

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) Formulieren Sie drei funktionale Anforderungen und zwei nicht-funktionale Anforderungen für ein Bibliotheksverwaltungssystem, das Ausleihe, Rückgabe und Reservierung unterstützt.
- (b) Beschreiben Sie in Klartext eine einfache Klassenbeschreibung mit den Klassen *Benutzer*, *Buch* und *Ausleihe*. Geben Sie kurze Attribute in Stichpunkten an und erläutern Sie Beziehungen zwischen den Klassen.
- (c) Nennen Sie drei Techniken der Anforderungsanalyse und erläutern Sie kurz, wofür sie verwendet werden.
- (d) Erläutern Sie den Unterschied zwischen Lastenheft und Pflichtenheft in der Softwareentwicklung. Formulieren Sie je ein Zielbeispiel pro Dokumentenart.

Aufgabe 2.

- (a) Beschreiben Sie eine modulare Architektur für eine E-Commerce-Plattform mit drei Kernkomponenten: *Benutzermanagement*, *Produktkatalog* und *Bestellwesen*. Diskutieren Sie Kopplung, Kohäsion und sinnvolle Schnittstellen.
- (b) Skizzieren Sie drei typische Schichten einer Softwarearchitektur und erläutern Sie, wie lose Kopplung zwischen ihnen erreicht wird.
- (c) Nennen Sie drei Vorteile der modularen Softwareentwicklung und eine potenzielle Herausforderung.
- (d) Geben Sie eine einfache Schnittstelle zwischen dem Produktkatalog und dem Bestellwesen in Form einer textuellen API-Skizze an.

Aufgabe 3.

- (a) Beschreiben Sie die wichtigsten Arten von Softwaretests: Unit-Tests, Integrationstests, Systemtests und Abnahmetests. Geben Sie Zweck und typische Kennzahlen zu jeder Art an.
- (b) Skizzieren Sie eine kleine Unit-Test-Suite für eine einfache Funktion $add(a, b)$. Beschreiben Sie je zwei Tests, die Randfälle und Sicherheitsaspekte abdecken.
- (c) Welche Qualitätskennzahlen würden Sie heranziehen, um den Zustand der Software zu bewerten? Nennen Sie drei Kennzahlen und erläutern Sie kurz ihren Zweck.

Aufgabe 4.

- (a) Beschreiben Sie vier Programmierparadigmen, die im Kurs behandelt werden: objektorientierte, funktionale, logische und modellorientierte Programmierung. Geben Sie zu jedem Paradigma ein kurzes Beispiel in Textform an.
- (b) Diskutieren Sie Vor- und Nachteile der objektorientierten Programmierung im Hinblick auf Modularisierung und Wiederverwendbarkeit.
- (c) Skizzieren Sie eine kleine Zustandsmaschine für den Prozess *Bestellung abschließen* mit den Zuständen *Offen*, *Beahlt*, *Versendet* und *Abgeschlossen*. Beschreiben Sie geeignete Übergänge zwischen den Zuständen.
- (d) Welche Prinzipien der Softwarearchitektur unterstützen die Umsetzung verschiedener Paradigmen? Nennen Sie zwei zentrale Prinzipien.

Aufgabe 5.

- (a) Beschreiben Sie ein exemplarisches agiles Vorgehensmodell mit kurzen Phasen, Rollen und Artefakten.
- (b) Skizzieren Sie eine einfache Backlog-Struktur für ein Modul *Bibliotheksausleihe* inklusive typischer User Stories und Akzeptanzkriterien.
- (c) Welche organisatorischen Aspekte sind bei der Teamarbeit besonders wichtig? Nennen Sie drei Aspekte.
- (d) Welche Metriken eignen sich zur Messung des Projektfortschritts? Nennen Sie zwei zentrale Metriken.

Lösungen

Lösung zu Aufgabe 1.

(a) Funktionale Anforderungen (mind. drei) und nicht-funktionale Anforderungen (mind. zwei) für ein Bibliotheksverwaltungssystem, das Ausleihe, Rückgabe und Reservierung unterstützt:

- **Funktionale Anforderungen**

- Eine registrierte Benutzereinheit kann ein verfügbares Exemplar eines Mediums ausleihen. Eine Ausleihe erzeugt einen Datensatz **Ausleihe** mit Referenzen auf Benutzer und Exemplar sowie Ausleihdatum und voraussichtlichem Rückgabedatum.
- Ein Exemplar kann zurückgegeben werden; der Zustand des Exemplar-Wickets wird aktualisiert und die zugehörige Ausleihe wird geschlossen.
- Ein Medium kann reserviert werden. Falls das Medium aktuell ausgeliehen ist, wird die Reservierung in einer Warteschlange geführt und der Benutzer benachrichtigt, sobald das Medium verfügbar ist.
- Verfügbarkeit prüfen: Vor der Ausleihe wird geprüft, ob das gewünschte Exemplar verfügbar ist (kein aktiver Ausleihevorgang, keine Sperren).
- Verwaltung von Strafen und Mahnungen: Überfällige Ausleihen lösen Mahnungen aus und blockieren ggf. eine neue Ausleihe.

- **Nicht-funktionale Anforderungen**

- **Verfügbarkeit/Performance:** Systemverfügbarkeit von mindestens 99,5% im Jahresmittel; Antwortzeit für einfache Abfragen ≤ 2 s; Skalierbarkeit bei steigender Benutzerzahl.
- **Sicherheit/Datenintegrität:** Zugriffskontrollen (Rollenkonzepte), saubere Trennung von Benutzer- und Bibliotheksdaten, Auditing von Ausleihen/Rückgaben.

(b) Klartextklassenbeschreibung und Beziehungen

- *Benutzer*

- Attribute (Beispiele): *benutzerId*, *name*, *kontakt*, *bibliotheksausweisNr*, *kontoStatus*

- *Buch*

- Attribute (Beispiele): *isbn*, *titel*, *autor*, *verlag*, *jahr*, *status* (*verfügbar*, *ausgeliehen*, *reserviert*)

- *Ausleihe*

- Attribute (Beispiele): *ausleiheId*, *benutzerId*, *isbn*, *ausleihDatum*, *rueckgabeVoraussicht*, *rueckgabeDatum*, *status*

- **Beziehungen**

- Ein *Benutzer* kann 0..n *Ausleihen* haben.
- Ein *Buch* kann 0..n *Ausleihen* haben (im Sinne von Ausleihhistorie).
- Eine *Ausleihe* verknüpft genau einen *Benutzer* mit genau einem *Buch*.

(c) Techniken der Anforderungsanalyse und Zweck

- *Interviews* mit Stakeholdern, um Bedürfnisse, Ziele und Prioritäten zu erfassen.
- *Use-Case-Analyse* bzw. *Use-Case-Modellierung* zur Beschreibung der Interaktion von Akteuren mit dem System.
- *Prototyping* (z. B. UI-Prototypen) zur frühzeitigen Validierung von Anforderungen und zur Reduktion von Missverständnissen.

(d) Unterschied Lastenheft vs Pflichtenheft; je ein Zielbeispiel

- *Lastenheft*: Beschreibt das Zielsystem aus Sicht des Auftraggebers, z. B. „Das System soll die Ausleihe, Rückgabe und Reservierung von Medien unterstützen.“
- *Pflichtenheft*: Konkretisiert die Umsetzungsvorgaben, z. B. „Die Ausleihe erfolgt über eine REST-API `/api/ausleihe`, unterstützt durch eine relationale DB; maximale Antwortzeit 2 s; Authentisierung über OAuth2.“

Lösung zu Aufgabe 2.

(a) Modulare Architektur der E-Commerce-Plattform

- *Kernkomponenten:*
 - *Benutzermanagement* (User Service): Verwaltung von Benutzerkonten, Authentifizierung, Berechtigungen.
 - *Produktkatalog* (Catalog Service): Verwaltung von Produkten, Kategorien, Preise, Verfügbarkeit.
 - *Bestellwesen* (Order Service): Warenkorb, Bestellabwicklung, Zahlung, Auftragsstatus.
- *Kopplung/Kohäsion:* Hohe Kohäsion innerhalb der Dienste; lose Kopplung zwischen Diensten über klar definierte Schnittstellen (APIs) und Messaging.
- *Schnittstellen:* API-Verträge (z. B. REST/GraphQL); asynchrone Events (z. B. Message Bus) für lose Kopplung; gemeinsame Domänenmodelle über bounded contexts.
- *Technische Aspekte:* Eigene Persistenz je Service; eventbasierte Benachrichtigungen bei Bestellstatusänderungen; zentrale Authentisierung.

(b) Drei Schichten einer Softwarearchitektur

- *Präsentationsschicht* (UI/API-Frontend)
- *Anwendungsschicht* (Dienstlogik, Orchestrierung der Use Cases)
- *Domänen-/Datenzugriffsschicht* (Domänenlogik, Persistenz)

Lose Kopplung wird erreicht durch: klar definierte Schnittstellen, Abstraktion, Dependency-Injection, getrennte Datenmodelle pro Schicht, asynchrone Kommunikation where sinnvoll.

(c) Drei Vorteile der modularen Softwareentwicklung und eine potenzielle Herausforderung

- Vorteile: (i) Parallele Entwicklung und klare Verantwortlichkeiten, (ii) Bessere Wartbarkeit und Wiederverwendbarkeit, (iii) Leichtere Skalierung einzelner Komponenten.
- Herausforderung: Komplexe Interaktion zwischen Modulen, API-Verträge müssen versioniert und abwärtskompatibel gehalten werden.

(d) Schnittstelle zwischen Produktkatalog und Bestellwesen (textuelle API-Skizze)

- Endpunkte (Beispiele):
 - GET /catalog/products/productId – Produktdetails
 - POST /orders – neue Bestellung erstellen
- Beispiel Request (JSON):

```
{
  "userId": "u123",
  "items": [
    {"productId": "p987", "quantity": 2},
    {"productId": "p654", "quantity": 1}
  ],
  "shippingAddress": {
    "line1": "Musterstraße 1",
    "city": "Beispielstadt",
    "postalCode": "12345",
    "country": "DE"
  },
  "paymentMethod": "credit_card"
}
```

- Erwartete Response (Beispiel):

```
{ "orderId": "o555", "status": "CREATED", "total": 129.50 }
```

Lösung zu Aufgabe 3.

(a) Wichtige Arten von Softwaretests

- *Unit-Tests*: Zweck: Prüfung einzelner Funktionen/Methoden isoliert von Abhängigkeiten; typische Kennzahlen: Code Coverage, Anzahl fehlschlagender Tests pro Build.
- *Integrationstests*: Zweck: Prüfung des Zusammenspiels mehrerer Module/Unit-Komponenten; Kennzahlen: Fehlerquote beim Zusammenspiel, Time-to-Detection.
- *Systemtests*: Zweck: Prüfung des Gesamtsystems aus Sicht des Endnutzers; Kennzahlen: Abdeckung end-to-end-Szenarien, Defect-Density auf Systemebene.
- *Abnahmetests (User Acceptance Testing, UAT)*: Zweck: Validierung der Geschäftskriterien durch den Auftraggeber; Kennzahlen: Erfüllungsgrad der Akzeptanzkriterien, Freigabewahrscheinlichkeit.

(b) Kleine Unit-Test-Suite für eine Funktion $add(a, b)$

- Randfall 1: `add(0, 0)` -> Erwartung: 0.
- Randfall 2: `add(Integer.MAX_VALUE, 1)` (overflow-Szenario in 32-Bit-Ganzzahlen) -> Erwartung: Overflow-Fehler bzw. definierte Overflow-Behandlung.
- Sicherheitsaspekt 1: `add("a", 1)` -> Erwartung: Typfehler/Exception.
- Sicherheitsaspekt 2: `add(null, 5)` -> Erwartung: `NullPointerException` bzw. definierte Fehlerbehandlung.

(c) Qualitätskennzahlen zur Bewertung des Softwarezustands

- *Code Coverage*: Anteil des Codes, der durch Tests abgedeckt ist; Ziel 80–90%.
- *Defect Density*: Anzahl der gefundenen Fehler pro Kilozeile Code (KLOC); dient der Qualitätsbewertung.
- *Build Health / Build Success Rate*: Anteil erfolgreich abgeschlossener Builds über die Zeit; Indikator für Stabilität der Build-Pipeline.

Lösung zu Aufgabe 4.

(a) Vier Programmierparadigmen und kurze Beispiele

- *Objektorientierte Programmierung (OOP)*: Organisation des Codes in Objekten mit Zustand und Verhalten; Beispiel: *Klasse Kunde* mit Methode `gebuehrBerechnen()`.
- *Funktionale Programmierung*: Fokus auf reinen Funktionen und unveränderlichen Daten; Beispiel: Verwendung von `map`/`reduce` zur Verarbeitung einer Liste von Bestellungen.
- *Logische Programmierung*: Regeln und Abfragen, oft deklarativ (z. B. Prolog-Ansatz); Beispiel: `fact kauft(Kunde, Produkt)`, `regel(Kunde)`.
- *Modellorientierte Programmierung (modellorientierte/Model-Driven)*: Arbeiten mit Modellen und Transformationen; Beispiel: UML-basierte Modellierung von Zuständen und Transformationsregeln in einer Code-Generierungspipeline.

(b) Vor- und Nachteile der objektorientierten Programmierung in Bezug auf Modularisierung und Wiederverwendbarkeit

- Vorteile: klare Abstraktionen durch Klassen/Objekte, einfache Wiederverwendung durch Vererbung/ Komposition, gute Modularisierung durch Kapselung.
- Nachteile: Overhead durch komplexe Vererbungsstrukturen; Gefahr von Zuweisung von zu engen oder zu breiten Verantwortlichkeiten; Versions- und Abhängigkeitsmanagement bei vielen Klassen.

(c) Zustandsmaschine für *Bestellung abschließen*

- **Zustände**: Offen, Beahlt, Versendet, Abgeschlossen
- **Übergänge (Beispiele)**:
 - Offen – *Zahlung bestätigt* → Beahlt
 - Beahlt – *Versand initiiert* → Versendet
 - Versendet – *Lieferung bestätigt* → Abgeschlossen
- Optional: weitere Übergänge wie *Stornierung* (von Offen oder Beahlt), *Rückfrage* etc.

(d) Zwei Prinzipien der Softwarearchitektur, die die Umsetzung verschiedener Paradigmen unterstützen

- *Trennung der Belange (Separation of Concerns, SoC)*: Liefert klare Verantwortlichkeiten und erleichtert den Einsatz unterschiedlicher Paradigmen in separaten Modulen.
- *Abstraktion über Schnittstellen (Interface-basierte Abstraktion / Dependency Inversion)*: Unterstützt lose Kopplung und ermöglicht modulübergreifende Integration verschiedener Paradigmen durch definierte Verträge.

Lösung zu Aufgabe 5.

(a) Exemplarisches agiles Vorgehensmodell (Phasen, Rollen, Artefakte)

- *Phasen/Iterationen*: Planen, Implementieren, Testen, Review, Retrospektive in kurzen Sprints (z. B. 2 Wochen).
- *Rollen*: Product Owner (priorisiert das Backlog), Scrum Master (entfernt Hindernisse, stellt Prozesse sicher), Entwicklungsteam (umsetzt).
- *Artefakte*: Product Backlog (priorisierte Anforderungen), Sprint Backlog (ausgewählte Aufgaben pro Sprint), Inkrement (funktionsfähige, potenziell auslieferbare Software am Sprint-Ende).

(b) Backlog-Struktur für das Modul *Bibliotheksausleihe*; typische User Stories und Akzeptanzkriterien

- *Epic: Bibliotheksausleihe*
 - **User Story 1**: Als registrierter Benutzer möchte ich ein Exemplar ausleihen können, damit ich es nach Hause nehmen kann.
 - * Akzeptanzkriterien:
 - Verfügbarkeit des Exemplar prüfen; bei verfügbarer Ware Ausleihe zulassen.
 - Rückgabedatum wird gesetzt (z. B. two weeks from Ausleihdatum).
 - Ausleihe-Record wird erzeugt und dem Benutzer angezeigt.
 - **User Story 2**: Als registrierter Benutzer möchte ich ein ausgeliehenes Exemplar zurückgeben können.
 - * Akzeptanzkriterien:
 - Rückgabe erhöht Verfügbarkeit des Exemplar.
 - Ausleihe wird geschlossen; ggf. Vorabwarnung bei Überfälligkeit.
 - **User Story 3**: Als registrierter Benutzer möchte ich eine Reservierung vornehmen können, falls das Exemplar aktuell ausgeliehen ist.
 - * Akzeptanzkriterien:
 - Reservierung wird in Warteschlange aufgenommen.
 - Benachrichtigung, wenn das Exemplar verfügbar wird.
 - **User Story 4**: Als Administrator möchte ich Ausleihen/Rückgaben nachvollziehen und Mahnungen generieren können.
 - * Akzeptanzkriterien:
 - Historie ist auditierbar; Mahnworkflow bei Überfälligkeit.

(c) Organisatorische Aspekte bei der Teamarbeit (drei Beispiele)

- Kommunikation (regelmäßige Abstimmungen, klare Kommunikationswege, transparente Statusberichte).
- Rollen- und Verantwortlichkeitsverteilung (klar definierte Verantwortlichkeiten, z. B. PO, Tech Lead, Dev-Team).

- Koordination und Abstimmung (Daily Stand-ups, gemeinsame Definition of Done, regelmäßige Retrospektiven).

(d) Zwei zentrale Metriken zur Messung des Projektfortschritts

- *Velocity* (Geschwindigkeit des Teams; z. B. gelöste Story Points pro Sprint).
- *Burndown/Burnup* (Verlauf der verbleibenden Arbeit oder des geleisteten Umfangs über die Zeit).