DHBW Stuttgart

Duale Hochschule
Baden-Württemberg

# Software Engineering

Dr. Eugenie Giesbrecht

# Verbleibende Termine

| 15.12.25 | 09.01.26 | 16.01.26 | 30.01.26 |
|---|---|---|---|
| ganztägig | nachmittags | nachmittags | nachmittags |

- 09.01.26 - Bitte organisieren Sie sich so, dass jeder einen 10-minütigen Zeitblock erhält
- 16.01.26 - Das Projekt, das Sie für Teil 2 umsetzen möchten, soll in Gruppen definiert und präsentiert werden. Dabei sind die Idee, die Anforderungen und die geplante Architektur zu berücksichtigen. (Einzelprojekte sind auch möglich)
- 30.01.26 – Online-Test über die gesamte Vorlesung
- Abgabe des Portfolios bis 20.02.2026 23:59 Uhr

# Bewertungskriterien: Software Engineering (60 Punkte)

| Bereich | Punkte | Kriterien |
|---|---|---|
| A) Funktionale App:<br><br>Mobile/Web-App mit Python + Streamlit + Figma | 20 | 1. Bereitstellung der Designs für **Desktop und Mobile** (falls abweichend) Format: **.fig** oder **SVG**<br><br>2. Mindestens **5 funktionale Anforderungen** aus der vorgegebenen Liste (siehe Seite 6)<br>**3. UI:** Dokumentation:<br>Welche UI-Elemente unterstützen welche UI-Prinzipien?<br>Für jedes der 10 UI-Prinzipien je ein konkretes Beispiel (in der README)<br>**4.** Anwendung soll **ohne technische Fehler** funktionieren<br>**5. Zusätzliche Features**<br>Implementierung von weiteren funktionalen Anforderungen (>5) zusätzlich zur Basisliste |

# Bewertungskriterien: Software Engineering (60 Punkte)

| Bereich | Punkte | Kriterien |
|---|---|---|
| B)<br>Codequalität | 10 | **MVC-Architektur:** Kurze Beschreibung in README<br>*Warum ist MVC für TODO-App sinnvoll?*<br>*Wie wurde MVC in diesem Projekt umgesetzt?*<br>**Dateiorganisation**, **Lesbarkeit**<br>Klarer, gut verständlicher Code, Kommentare,<br>Konsistente Benennungskonventionen<br>**Testing**<br>Testumsetzung gemäß den heutigen Übungen<br><br>Keine UML-Diagramme oder Requirements-Tabellen erforderlich |

# Bewertungskriterien: Software Engineering (60 Punkte)

| Bereich | Punkte | Kriterien |
|---|---|---|
| C) (Online) Präsentation am 09. Januar | 20 | Verständnis der eigenen Lösung, Anpassungen live |
| D) Online-Multiple-Choice-Tests zu Beginn jeder Lektion im November und Dezember | 10 | Theoriewissen |

# Funktionale Anforderungen

| | | | |
|---|---|---|---|
| xXxXxXxXXXXx Xxxx | Das System muss Aufgaben persistent speichern (z. B. lokal/DB). | Backend | MUSS |
| xxxXxxxxXXxx | Die App soll es dem Nutzer ermöglichen, innerhalb von maximal 5 Sekunden eine neue Aufgabe mit Titel anzulegen. | Performance | MUSS |
| xxXXxxxXxxxxx | Todo löschen | Funktional | MUSS |
| xxXXxXxxxxxxx | Todo bearbeiten | Funktional | MUSS |
| xxXxxxx | Todo als erledigt markieren | Funktional | MUSS |
| xXxxxXXxX | Das System muss Aufgaben in einer Liste anzeigen. | Frontend | MUSS |
| xxxXxxxxXXxx | Der Nutzer soll Aufgaben bis zu fünf Kategorien zuordnen können, die er selbst erstellen und löschen kann. | Frontend | Soll |
| xXxxXXXx | Das System soll Aufgaben nach Status (offen/erledigt) filtern können. | Frontend | SOLL |
| xxxXxxxxXXxx | Jede Aufgabe soll ein Fälligkeitsdatum enthalten können, das über einen Kalenderpicker in unter 3 Klicks ausgewählt werden kann. | Frontend | Kann |

# Bewertungskriterien: Fortgeschrittenes Software Engineering (60 Punkte)

| Bereich | Punkte | Kriterien |
|---|---|---|
| A) Funktionale App | 20 | Die Anwendung muss einen KI-Service integrieren und zusätzlich mindestens fünf funktionale sowie zwei nicht-funktionale Anforderungen erfüllen<br><br>**Qualitätsanforderungen**: UI-Konsistenz, Fehlerfreiheit, Cross-Plattform-Ausführbarkeit |
| B) Codequalität | 20 | **Architektur**<br>Nutzung einer in der Vorlesung behandelten Architektur**, jedoch nicht MVC**<br>**Verwendung von Design Patterns**<br>Einsatz der in der **Vorlesung vom 15.12** behandelten Entwurfsmuster (mindestens 2)<br>**Anwendung der UI-Prinzipien**<br>Umsetzung der in der **UI-Vorlesung** erläuterten Prinzipien<br>**Durchgeführte Tests**<br>4 Unit Tests, 2 Integrationstests, 1 Systemtest, 1 End-to-End-Test |

# Bewertungskriterien: Fortgeschrittenes Software Engineering (60 Punkte)

| Bereich | Punkte | Kriterien |
|---|---|---|
| C) Dokumentation | 10 | **1. Requirements: FR** und **NFR** im Tabellenformat (Spalten: **ID**, **Anforderungstyp**, **Beschreibung**)<br>**2. Umsetzung der UI-Prinzipien:** Übersicht, welche UI-Prinzipien konkret angewendet wurden<br>**3. UML-Diagramme**: Klassendiagramm, Komponentendiagramm, Aktivitätsdiagramm, Sequenzdiagramm<br>**4. Architektur und Design-Patterns**<br>Kurze Beschreibung der eingesetzten Architektur<br>Kurze Erläuterung der verwendeten Design-Patterns, z. B. „Das Factory Pattern wurde für X eingesetzt und mithilfe der Klassen X und Y umgesetzt.“<br>**5. Testdokumentation**<br>Kurze Beschreibung der Testfälle inklusive der jeweils abgedeckten Komponenten bzw. Systemteile |
| D) Online-Test (30.01.26) | 10 | Theoriewissen: Vorlesungsthemen |

# Software Testing

# Software Testing

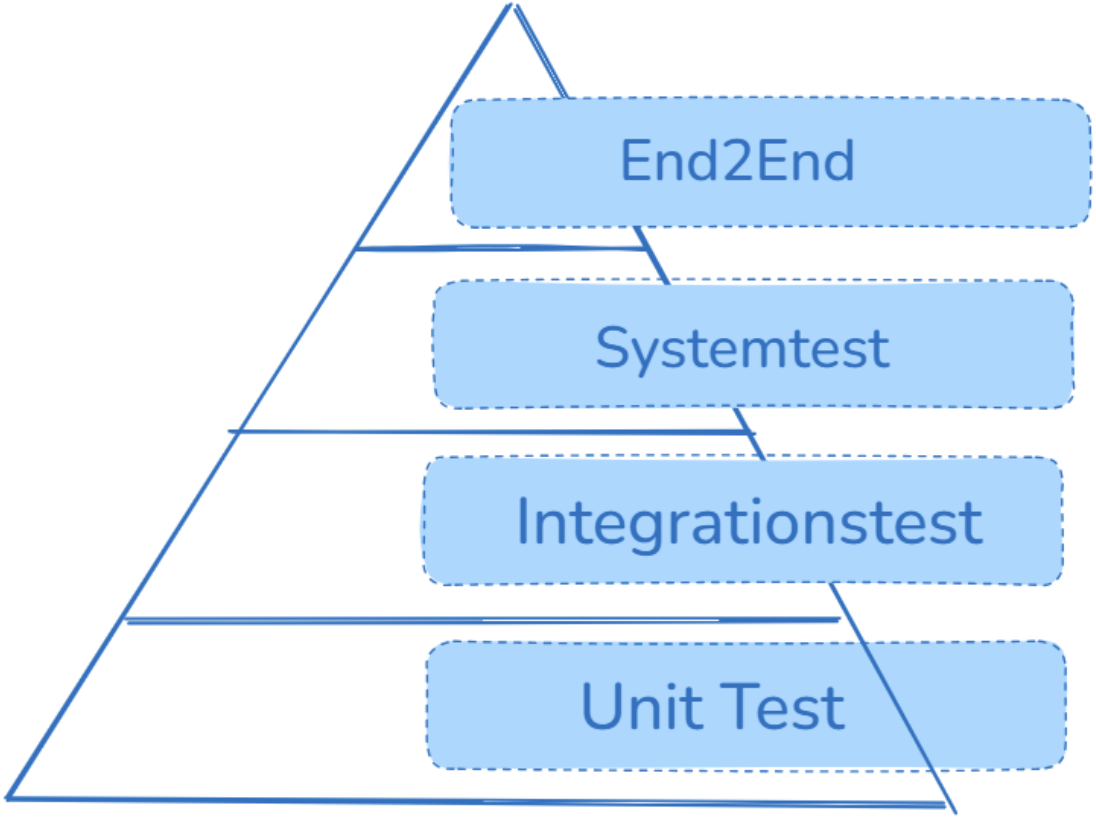
- «Testing in Software Engineering is a process of evaluating a software product to find whether the current software product meets the required conditions or not» (ANSI/IEEE 1059)
- Vorreiter – das „V-Modell“
- Softwaresysteme werden immer komplexer => Fokus auf bestimmte Bereiche legen und diese priorisieren

# Dimensionen

Klassifikation der Testverfahren nach ...

- Stufe
- Schicht
- Methode
- Fokus

# Dimension: Stufe (Testpyramide)



# Grundlagen der Unit Tests

1. Zweck von Unit Tests
2. Charakteristika guter Unit Tests:

| Buchstabe | Bedeutung | Erklärung |
|---|---|---|
| F | Fast | Tests müssen schnell laufen, sonst werden sie nicht genutzt. |
| I | Independent | Tests dürfen sich nicht gegenseitig beeinflussen. |
| R | Repeatable | Tests müssen deterministisch sein (kein Zufall, keine externen Ressourcen). |
| S | Self-Validating | Jeder Test endet mit klaren Pass/Fail-Asserts. |
| T | Timely | Tests sollten während oder vor der Implementierung geschrieben werden. |

# 3. Struktur eines Unit Tests: AAA-Muster

Ein verbreitetes Muster ist **Arrange – Act – Assert**:

- **Arrange**: Testdaten und Objekte vorbereiten
- **Act**: Die zu testende Funktion ausführen
- **Assert**: Ergebnis prüfen

```
# Arrange
item = TodoItem("Task")
# Act
item.mark_done()
# Assert
assert item.done is True
```

# 4. Test Isolation

Ein Unit Test darf nicht abhängig sein von:

- Datenbanken
- Filesystem
- Netzwerk
- externer Zeit
- globalen Zuständen
- Reihenfolge der Tests

Falls notwendig: **Mocking** (unittest.mock).

# 5. Code Coverage

- **Coverage** misst, wie viel Prozent der Codebasis durch Tests ausgeführt werden.
  Typische Zielwerte in Projekten:
- 70–80 %: solide
- 90 %+: meist nur für sicherheitskritische Systeme notwendig
- Wichtig: hohe Coverage bedeutet nicht, dass der Code korrekt ist – nur dass er ausgeführt wurde.
- **coverage.py**

# 6. Anti-Patterns

- Tests mit Logik (Schleifen, if-Bedingungen, Berechnungen)
- Tests, die nicht deterministisch sind (Random, Zeit, Netzwerk)
- Tests, die zu viele Fälle abdecken (Genereller Test statt kleiner Tests)
- Zu große Setup-Blöcke
- Testen privater Methoden (deutet auf schlechte Architektur)

# 7.Frameworks

In Python sind üblich:

- **pytest** (modern, flexibel, Industrie-Standard)
- **unittest** (Standardbibliothek)
- **hypothesis** (property-based testing)
- **coverage.py** (Coverage-Analyse)

**pytest** ist am weitesten verbreitet

# Aufgabe: Unit Tests für TODO-App

Erstellen Sie eine vollständige Suite von **Unit Tests** für die TODO-App. Verwenden Sie dafür das Framework **pytest**. Die Tests sollen die Logik der Anwendung vollständig abdecken, ohne externe Systeme einzubeziehen.

- Jeder Test soll klar das AAA-Muster verwenden.
- Jeder Test muss unabhängig voneinander lauffähig sein.
- Ihre Tests sollen mindestens 80 % des Codes abdecken (Sie können coverage.py nutzen, um das zu messen).

**Erstellen Sie mindestens folgende Unit Tests:**

- Hinzufügen eines neuen TODO-Items
- Entfernen eines Items
- Markieren als erledigt / nicht erledigt
- Bearbeiten eines Items
- Optional: Fehlerfälle oder Randbedingungen (z. B. leere Titel, doppelte Titel)

**Abgabe: test_unit.py**

# Integrationstest

# Zielsetzung

Integrationstests sollen sicherstellen, dass:

- Schnittstellen korrekt umgesetzt sind
- Komponenten die erwarteten Parameter weitergeben
- Rückgabewerte korrekt interpretiert werden
- reale Interaktionen funktionieren (z. B. echter DB-Zugriff, echtes Dateisystem – falls vorgesehen)
- typische Fehler beim Zusammenspiel früh sichtbar werden

Integrationstests prüfen also **nicht** einzelne Funktionen, sondern **Interaktionsketten**.

# Charakteristika guter Integrationstests

- Sie testen **mehrere Komponenten gemeinsam**, aber nicht das ganze System.
- Sie sollen **realistisch**, aber immer noch **kontrollierbar und deterministisch** sein.
- Nur dort sollte gemockt werden, wo externe Systeme ansonsten unkontrollierbar wären (API, Netzwerk, Drittanbieter).

# Typische Beispiele bei einer TODO-App

**Integrationstests könnten prüfen:**

- Eine Aufgabe wird erstellt, gespeichert und anschließend direkt wieder gefunden
- Ein Task wird über mehrere Schichten hindurch korrekt aktualisiert
- Fehlerhafte Eingaben führen in allen Schichten zu konsistentem Verhalten
- *TodoService* ruft korrekt das *TODORepository* auf und gibt valide Datentypen zurück

# Anti-Patterns bei Integrationstests

- Übermäßig große End-to-End-Szenarien
- Unkontrollierbare Abhängigkeiten (z. B. echte externe APIs)
- Nicht deterministische Tests
- Instabile oder flüchtige Testdaten

# Aufgabe: Integrationstests für eine TODO-App

Erstellen Sie 3 **Integrationstests, z.B.**

- Mehrere Aufgaben erstellen und sicherstellen, dass die Repository-Schicht konsistent bleibt
- Aktualisieren eines vorhandenen Items (z. B. Status ändern) -> → Status korrekt im Repository aktualisiert
- Leere Aufgabe → Repository wirft kontrollierten Fehler
- Löschen eines Items und Überprüfung, ob es aus dem Repository verschwindet

Abgabe: test_intergration.py

# Systemtest

# Zielsetzung

Systemtests prüfen:

- vollständige technische End-to-End-Abläufe, ohne Endnutzer
- funktionale Anforderungen
- nichtfunktionale Anforderungen (teilweise), z. B.:
  - Performance
  - Robustheit
  - Fehlerverhalten
  - Schnittstellenkonsistenz

# Merkmale guter Systemtests

- Testen das **System als Ganzes**, nicht einzelne Module
- Fokussieren auf **Use Cases**, nicht auf Methoden
- Nutzen eine **Umgebung, die der Produktion möglichst ähnlich ist**
- Sind **deterministisch**, soweit möglich

# Anti-Patterns

- Mocks einsetzen (Systemtests sollen realistisch sein)
- Instabile Tests aufgrund von Zeitabhängigkeiten oder asynchronen Prozessen
- Kombination von Funktionalität und technischem Detailwissen
- Testen von Dingen, die Unit- oder Integrationstests übernehmen sollten

# Aufgabe: Systemtests für die TODO-App

**Ziel:** Prüfen, dass **das komplette System technisch korrekt arbeitet**, inkl. Service, Repository und evtl. UI-Elemente (kontrolliert, ohne echten Browser).

Framework: **pytest**

**Szenarien**:

- Aufgabe anlegen → geprüft wird Speicherung und Abruf
- Aufgabe als erledigt markieren → Status korrekt im System
- Aufgabe löschen → System konsistent
- Fehlerfälle → kontrollierte Fehlermeldung

**Abgabe**: system_test.py

# End2End-Test (E2E)

# Ziele von E2E-Tests

- **Validierung realer Benutzerflüsse:** Hier testen wir den gesamten Workflow, vom User Interaktionen mit dem UI bis zum Backend
  - **Beispiel TODO-App**: Aufgabe anlegen → markieren → löschen → alles korrekt in UI und Datenbank.
- **Sicherstellung von Systemstabilität unter realistischen Bedingungen**
  - Reale Daten, Browser-Umgebung, UI-Interaktionen.

# Automatisierung von End-to-End Tests

**Typische Werkzeuge:**

- Selenium **- Klassiker, platformübergreifend**
- Playwright – modernes Browser-Automation-Framework, unterstützt Chromium, Firefox, WebKit
- Cypress- beliebt für Webanwendungen, vor allem Frontend-lastig

# Playwright in VS Code

- https://playwright.dev/docs/getting-started-vscode


https://playwright.dev/docs/getting-started-vscode

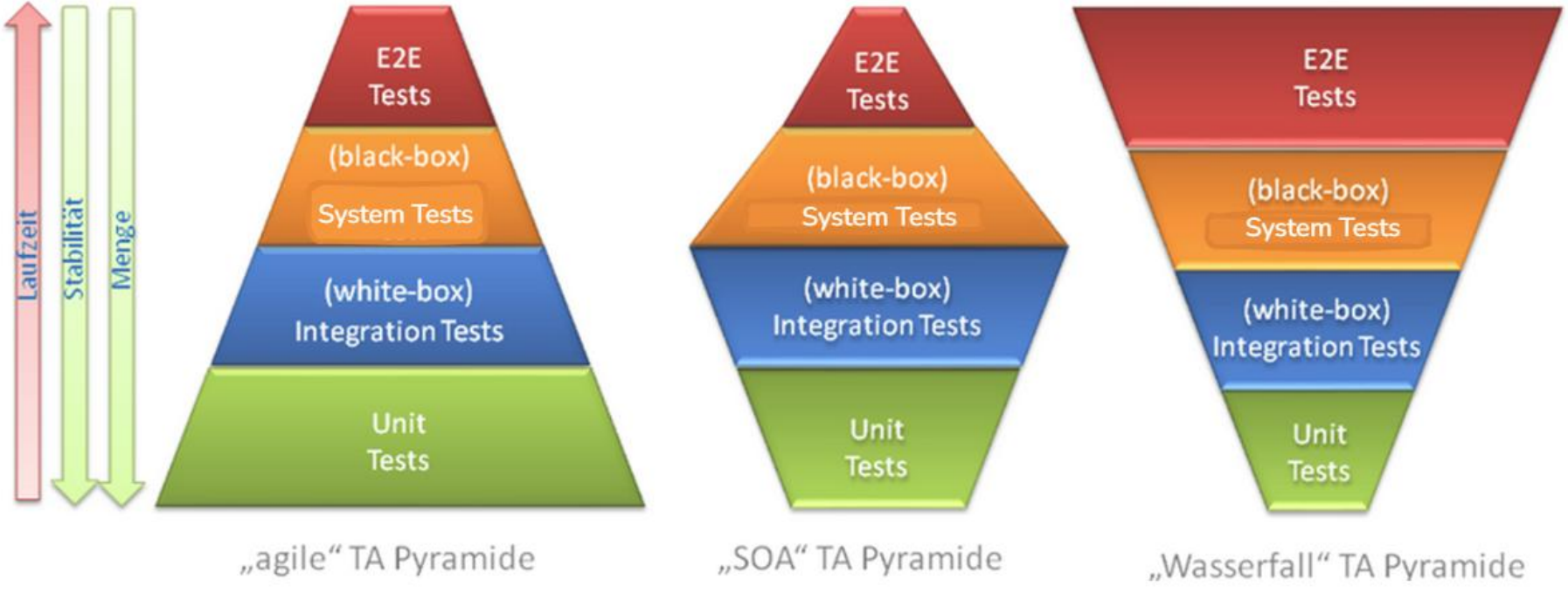
# Aufgabe: End-to-End-Test (E2E) für die TODO-App

- **Ziel:** Prüfen, dass ein Benutzer alle Abläufe erfolgreich durchführen kann, inklusive Frontend (Streamlit), Backend und Persistenz.
- **Framework**: pytest + Playwright
- **Mögliche Szenarien:**
  - Aufgabe über UI anlegen → sichtbar und gespeichert
  - Aufgabe als erledigt markieren → Status sichtbar und gespeichert
  - Aufgabe löschen → aus UI und Persistenz entfernt
  - Fehlerfall: leere Aufgabe → Fehlermeldung sichtbar
  - Optional: App neu starten → Aufgaben korrekt wiederhergestellt

# Abgrenzung von Tests

| Testtyp | Fokus | Scope |
|---|---|---|
| **Unit-Test** | einzelne Funktionen / Klassen | minimal |
| **Integrationstest** | Zusammenspiel von Modulen | mittlerer Umfang |
| **Systemtest** | gesamtes System technisch, kontrolliert | gesamtes System, aber nicht echte Benutzerflows |
| **E2E-Test** | reale Benutzerflows über alle Komponenten | gesamtes System, inkl. UI, Persistenz, Schnitt |

# Testautomatisierungs-Pyramide




https://www.testautomatisierung.org/tutorial-testautomatisierung-auf-verschiedenen-stufen-1-ueberblick/

# Dimensionen

Klassifikation der Testverfahren nach ...

- Stufe
- Schicht
- Methode
- Fokus

# Dimension: Schicht

- UI
- API
- Datenhaltung
- Geschäftslogik
- Middleware
- Netzwerk
- Systemschicht
- Sicherheit
- Performance
- Konfiguration

# Dimensionen

Klassifikation der Testverfahren nach ...

- Stufe
- Schicht
- Methode
- Fokus

# Dimension: Methode

- **Statische** Testverfahren
  - z.B. „Reviews“ wie Code-Review, statische Code-Analyse (automatisierte Verfahren wie Pylint für Python), Regelkonformität und Codierungsrichtlinien
- **Dynamische** Testverfahren
- **Diversifizierende** Verfahren – testen zwei Versionen einer Software gegeneinander

# Testmethoden (White-, Black-, Grey-Box)

Beschreiben **die Herangehensweise** beim Testen

- **White-Box** → Unit-Tests, Pfadüberdeckung, statische Code-Analyse
- **Black-Box** → UI-Tests, End-to-End-Tests, funktionale Tests
- **Grey-Box** → Integrationstests, API-Tests mit teilweisem Codewissen

# Dimensionen

Klassifikation der Testverfahren nach ...

- Stufe
- Schicht
- Methode
- Fokus

# Dimension: Fokus

- Funktionale Tests
- Nicht-funktionale Tests (Leistungstests)

# Funktionale Tests

- Black-Box-Testen
- White-Box-Testen
- Graue-Box-Testen

# Nicht-funktionale Tests (Leistungstests)

# Beispiele für nicht-funktionale Tests

- Usability Tests
- Akzeptanztests
- Leistungs- und Performancetests
- Recovery Tests
- Portabilitätstests

# Usability Tests

- Usability-Testing hilft sicherzustellen, dass die Software einfach zu benutzen ist und eine positive Benutzererfahrung bietet

  - **Fokus:** Bei diesem Test geht es um **die Benutzeroberfläche (UI)** und die **Bedienbarkeit** der Software

  - **Testmethoden:** Usability-Tests beinhalten oft Beobachtungen von Benutzern, das Durchführen von Aufgaben, Umfragen, Interviews und das Sammeln von Feedback

# Last- und Performance Tests

- **Leistungs- und Performancetests**: wie gut eine Software unter realen oder extremen Bedingungen arbeitet
- **Recovery Tests**: wie gut eine Software nach einem Ausfall oder Fehler wieder in den Normalbetrieb zurückkehrt
- **Portabilitätstests**: wie gut eine Software auf verschiedenen Plattformen, Betriebssystemen oder Geräten funktioniert

# Best Practices

1. Kontinuierliche Tests
2. Testautomatisierung
3. Defekt- oder Fehlerverfolgung
4. Metriken, KPIs und Berichte
5. Code Coverage erhöhen
6. Schlaue End-to-End-Tests