# LU02a - Prüfstrategie

**Auf einen Blick**
Die Teststrategie legt den Rahmen für alle Tests eines Projekts fest: **was** wir testen, **wie gründlich**, **womit**, **in welcher Reihenfolge** und **wann wir aufhören**. Weil Zeit und Geld immer knapp sind, setzt sie Schwerpunkte dort, wo Fehler den grössten Schaden anrichten.

## Lernziele

Sie können

- erklären, wozu eine Teststrategie dient und wie sie sich vom Testplan unterscheidet,
- die fünf Leitfragen einer Teststrategie auf ein Projekt anwenden,
- Top-down, Bottom-up, hardest first, risikobasiertes und datengetriebenes Testen unterscheiden,
- Risiken mit Eintrittswahrscheinlichkeit und Schadensausmass bewerten.

## Teststrategie und Testplan

| | **Teststrategie** | **Testplan** |
|---|---|---|
| **Frage** | Wie gehen wir beim Testen grundsätzlich vor? | Wer testet was, wann und mit welchen Mitteln? |
| **Inhalt** | Ziele, Schwerpunkte, Vorgehen, Testende | Termine, Personen, Testfälle, Umgebung |
| **Gültigkeit** | ganzes Projekt, ändert sich selten | wird laufend angepasst |
| **Vergleich** | Spielidee einer Fussballmannschaft | Aufstellung und Matchplan für das nächste Spiel |

Eine Teststrategie muss kein dickes Dokument sein. Wichtiger ist, dass alle Beteiligten, also Auftraggeber, Entwicklung und Test, die Schwerpunkte kennen und mittragen.

## Fünf Leitfragen

Für jeden Teil des Systems beantwortet die Teststrategie diese Fragen:

| **Leitfrage** | **Beispiel: Webseite British Inheritance Tax** |
|---|---|
| **Testziele:** Was wollen wir wissen? | Berechnet die Seite die Steuer gemäss der britischen Regelung korrekt? Werden falsche Eingaben abgefangen? |
| **Intensität:** Wie gründlich testen wir? | Die Steuerberechnung sehr gründlich mit allen Grenzwerten, das Layout nur stichprobenweise. |
| **Methoden und Techniken:** Wie finden wir Testfälle? | Äquivalenzklassen und Grenzwerte (Blackbox), Unit-Tests für die Berechnungsfunktion (Glassbox). |
| **Infrastruktur:** Was brauchen wir? | Aktuelle Browser auf Windows und macOS, ein Smartphone, PyTest für automatisierte Tests. |

| Leitfrage | Beispiel: Webseite British Inheritance Tax |
|---|---|
| **Reihenfolge:** Was testen wir zuerst? | Zuerst die Berechnung, dann die Eingabeprüfung, zuletzt die Darstellung. |

# Vorgehensweisen

## Top-down

Wir testen zuerst die oberste Ebene, zum Beispiel die Steuerung oder die Benutzeroberfläche. Funktionen auf tieferen Ebenen, die noch nicht fertig sind, ersetzen wir durch **Stubs**.

**Stub:** ein Platzhalter für eine aufgerufene Funktion. Er liefert einen festen, vorher bekannten Wert zurück.

```
def steuersatz(jahre):
    # Stub: Die echte Berechnung ist noch nicht fertig
    return 0.24
```

**Vorteil:** Die Abläufe aus Sicht der Benutzer lassen sich früh prüfen.
**Nachteil:** Fehler in den Detailfunktionen werden erst spät entdeckt.

## Bottom-up

Wir beginnen mit den kleinsten Funktionen und setzen sie schrittweise zu grösseren Einheiten zusammen. Weil die aufrufenden Programmteile noch fehlen, rufen wir die Funktionen mit einem **Driver** auf.

**Driver:** ein kleines Programm, das die zu testende Funktion mit Testdaten aufruft und das Ergebnis prüft. Ein Unit-Test ist ein typischer Driver.

```
def test_steuersatz_viereinhalb_jahre():
    assert steuersatz(4.5) == 0.24
```

**Vorteil:** Die Grundbausteine sind früh zuverlässig getestet. Diese Strategie wird von Unit-Test-Werkzeugen wie PyTest direkt unterstützt und ist in der Praxis sehr verbreitet.
**Nachteil:** Das Programm als Ganzes ist erst spät testbar.

## Hardest first

Die komplexesten oder wichtigsten Teile werden zuerst getestet. So bleibt genug Zeit, um dort gefundene Fehler zu beheben.

### Risikobasiertes Testen

Weil wir nie alles gleich gründlich testen können, planen wir die Tests anhand einer **Risikoanalyse**:

**Risiko = Eintrittswahrscheinlichkeit × Schadensausmass**

Wir bewerten jeden Teil des Systems, zum Beispiel auf einer Skala von 1 (gering) bis 3 (hoch):

| Teil des Systems | Wahrscheinlichkeit eines Fehlers | Schaden bei einem Fehler | Risiko | Testintensität |
|---|---|---|---|---|
| Steuerberechnung | 2 (viele Grenzfälle) | 3 (falsche Steuer) | **6** | sehr hoch |
| Prüfung der Eingaben | 3 (viele Varianten) | 2 (falsche Resultate) | **6** | sehr hoch |
| Zurücksetzen-Button | 1 | 1 | **1** | gering |
| Layout auf dem Smartphone | 2 | 1 | **2** | gering |

Teile mit hohem Risiko werden **zuerst** und **am gründlichsten** getestet.

### Datengetriebenes Testen

Der Testablauf wird einmal programmiert und dann automatisch mit vielen verschiedenen Datensätzen ausgeführt, zum Beispiel aus einer Tabelle.

```
import pytest

@pytest.mark.parametrize('jahre, satz', [
    (2.5, 0.40), (3.5, 0.32), (4.5, 0.24), (5.5, 0.16), (6.5, 0.08), (7.5,
0.0),
])
def test_steuersatz(jahre, satz):
    assert steuersatz(jahre) == satz
```

So lassen sich ganze Testfallsammlungen mit wenig Aufwand testen und erweitern.

## Qualitätsmerkmale nach ISO/IEC 25010

Die Normenreihe ISO/IEC 25000 beschreibt, wie Softwarequalität bewertet wird. Teil davon ist die Norm ISO/IEC 25010 mit diesen Qualitätsmerkmalen. Sie helfen, bei der Teststrategie nichts Wichtiges zu vergessen:

| Merkmal | Frage |
|---|---|
| Funktionale Eignung | Macht die Software, was sie soll, und rechnet sie richtig? |
| Leistungseffizienz | Ist sie schnell genug und sparsam mit Ressourcen? |
| Kompatibilität | Funktioniert sie mit anderen Systemen und Umgebungen? |
| Benutzbarkeit | Ist sie verständlich und einfach zu bedienen? |

| Merkmal | Frage |
|---|---|
| Zuverlässigkeit | Läuft sie stabil und erholt sie sich von Fehlern? |
| Sicherheit | Sind Daten vor unberechtigtem Zugriff geschützt? |
| Wartbarkeit | Lässt sie sich einfach ändern und testen? |
| Übertragbarkeit | Lässt sie sich auf andere Plattformen übertragen? |

## Wann hören wir auf zu testen?

Da man nie alle Fehler finden kann, legt die Teststrategie **Testende-Kriterien** fest, zum Beispiel:

- Alle Testfälle mit hohem Risiko sind bestanden.
- Es gibt keine offenen Hauptfehler mehr.
- Die Anweisungsabdeckung der Unit-Tests beträgt mindestens 80 % (siehe LU02c).
- Das Testbudget oder der Termin ist erreicht. Dann muss bewusst entschieden werden, welche Risiken man in Kauf nimmt.

## Kontrollfragen

1. Worin unterscheidet sich eine Teststrategie von einem Testplan?
2. Wozu braucht man beim Top-down-Test Stubs und beim Bottom-up-Test Driver?
3. Eine Funktion wird selten benutzt, ein Fehler würde aber Daten löschen. Wie hoch ist das Risiko und wie gründlich testen Sie?
4. Nennen Sie zwei sinnvolle Testende-Kriterien für ein Schulprojekt.

## Quellen

- Softwaretest
- andagon.com - Teststrategie

---

M450-LU02

Marcel Suter

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
**https://wiki.bzz.ch/modul/m450/learningunits/lu02/teststrategie**

Last update: **2026/11/11 10:54**