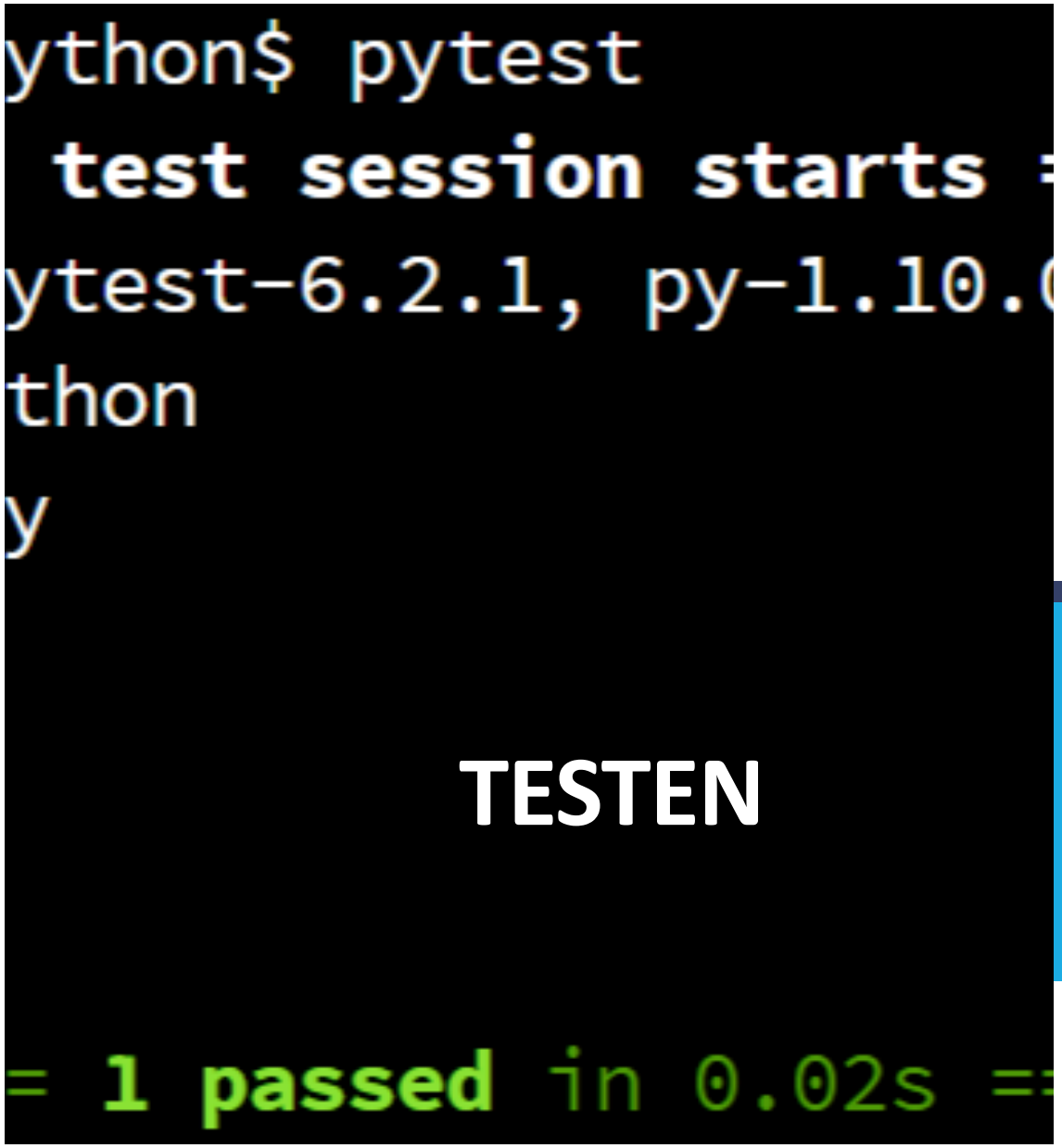

ython$ pytest
test session starts
ytest-6.2.1, py-1.10.
thon
y
= 1 passed in 0.02s =

TESTEN

Inhalt

- Klassifizierung von Testverfahren
- Test – Übersicht
- Testprozess
- Testkonzepte
- Testfälle ermitteln

Übersicht

Klassifizierung von Testverfahren

- Kriterien zur Klassifizierung
  - Wer testet?
  - Was wird getestet?
  - Wie wird getestet?
  - Wann wird getestet?
  - Warum wird getestet?

Kriterien zur Klassifizierung

Wer testet?	Was wird getestet?	Wie wird getestet?	Wann wird getestet?	Warum wird getestet?
• Mensch (manuell) vs. Maschine (automatisch) • Entwickler vs. Benutzer	• Komponente (Unit-Test/Funktionstest /Klassentest) vs. Integration vs. System (End-to-End) • Testpyramide	• Bottom-Up vs. Top-Down • statisch (Kompilierzeit) vs. dynamisch (Laufzeit) • ohne Kenntnis des Codes (Blackbox) vs. mit Kenntnis des Codes (Whitebox) • explorativ • Schreibtischtest/ Review	• Vor vs. nach der Entwicklung • Abnahmetest	• Regressionstest • Lasttest • Belastungstest • Performancetest • Smoketest

Wer testet?

Kategorie	Beispiele
Manuell (Mensch)	Explorative Tests, GUI-Tests durch QA
Automatisch (Maschine)	Unit-Tests, CI/CD-basierte Tests, Lasttests
Entwickler	Unit-Tests, Whitebox-Tests
Benutzer	Abnahmetests, Beta-Tests, Usability-Tests

Was wird getestet?

Kategorie	Beispiele
Komponentenebene	Unit-Test, Klassentest, Modultest
Integrationsebene	Schnittstellentest, API-Test
Systemebene (End-to-End)	E2E-Test, UI-Test, Geschäftsprozess-Test
Testpyramide (nach Mike Cohn)	Basis: viele Unit-Tests Mittig: einige Integrationstests Spitze: wenige UI-/E2E-Tests

Wie wird getestet?

Methode	Beschreibung / Beispiel
Bottom-Up	Zuerst Unit-Tests, dann Integration
Top-Down	Erst UI-Ebene, dann tiefer
Statisch (zur Kompilierzeit)	Code Reviews, Linting, Statische Codeanalyse
Dynamisch (zur Laufzeit)	Unittest, Integrationstest, Lasttest
Blackbox-Test (ohne Codekenntnis)	UI-Test, Akzeptanztest
Whitebox-Test (mit Codekenntnis)	Unit-Test, Branch-Coverage-Test
Explorativ	Freies, erfahrungsbasiertes Testen
Schreibtischtest / Review	Durchsicht von Code oder Dokumenten ohne Ausführung

Wann wird getestet?

Zeitpunkt	Beispiele
Vor der Entwicklung	TDD (Test Driven Development), Review von Spezifikationen
Während / nach der Entwicklung	Unittests, CI/CD-Tests, Regressionstests
Vor Übergabe / Livegang	Abnahmetests, Beta-Tests, Smoketests

Warum wird getestet?

Testart	Zweck
Regressionstest	Sicherstellen, dass Änderungen keine bestehenden Funktionen beschädigen
Last-/Performancetest	Systemverhalten unter Belastung prüfen
Smoketest	Grundfunktionalität nach Build prüfen („Startet das System überhaupt?“)

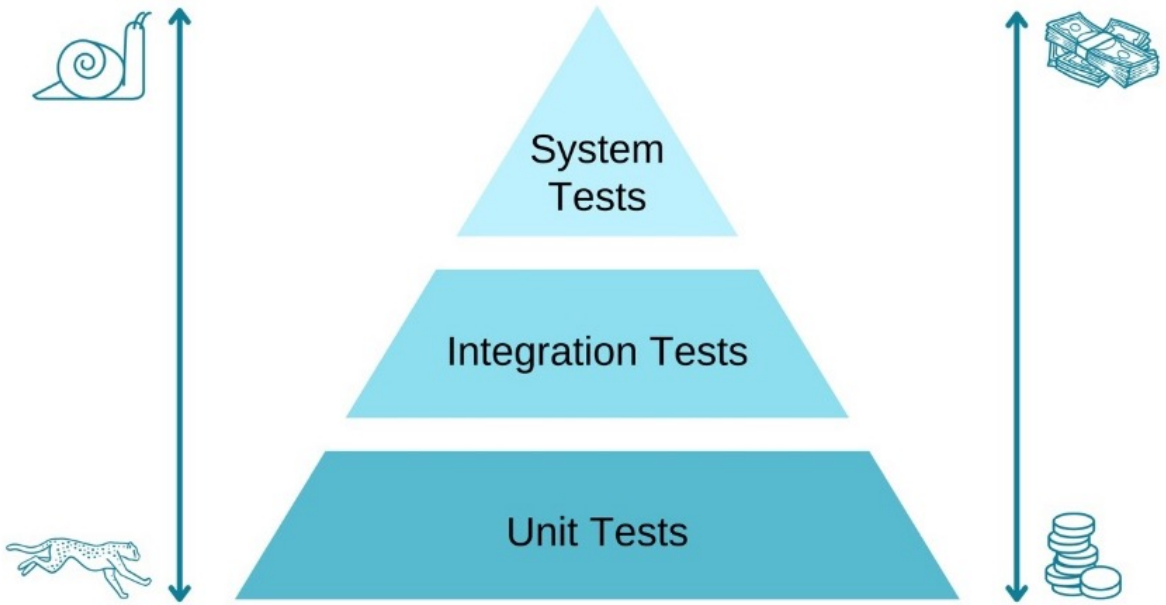
Übersicht

Testverfahren

- Testpyramide
- Bottom-Up-Test versus Top-Down-Test
- Blackbox-Test versus Whitebox-Test
- print-Debugging
- Schreibtischtest
- Code Review
- Explorative Tests
- Unit Test
- E2E-Test
- Integrationstest
- Abnahmetest
- Lasttest/Belastungstest/Performancetest
- Regressionstest
- Smoketest

Testpyramide

Je weiter oben, desto teurer und langsamer sind die Tests – also sollten viele Unit-Tests, wenige Integrationstests und ganz wenige System-/E2E-Tests geschrieben werden



Bottom-Up-Test versus Top-Down-Test

BOTTOM-UP

Erst werden untere Module (Units) getestet, dann Schritt für Schritt nach oben integriert

Frühzeitig stabilen Unterbau entwickeln

Jedoch geht UI unter

TOP-DOWN

Test startet bei höheren Modulen (z. B. GUI), untergeordnete Module werden schrittweise ersetzt oder eingebunden

Benutzerlogik früh getestet

Jedoch Mocks notwendig

Blackbox-Test versus Whitebox-Test

BLACKBOX-TEST

aus Benutzerperspektive getestet
Funktionalität von außen

benutzerorientiert, realitätsnah
begrenzte Abdeckung

WHITEBOX-TEST

Quellcode ist bekannt
interne Logik, Zweige, Schleifen

Codepfadabdeckung
nur durch Entwickler möglich

print-Debugging

- einfache und weit verbreitete Methode zur Fehlersuche in der Softwareentwicklung
- gezielte Ausgaben (z. B. mit print(), console.log(), System.out.println() o. ä.), um:
  - Programmlauf beobachten
  - Variablenwerte prüfen
  - Ablauf bestimmter Codeabschnitte nachvollziehen
- besonders hilfreich, wenn kein Debugger verfügbar oder das Problem schwer reproduzierbar ist

Beispiel in Python

```
def berechne_betrag(x):
    print("Funktion aufgerufen mit x =", x)
    if x < 0:
        print("Negativer Wert erkannt")
        return -x
    return x

print(berechne_betrag(-5))
```

Ausgabe:

```
Funktion aufgerufen mit x = -5
Negativer Wert erkannt
5
```

Schreibtischtest

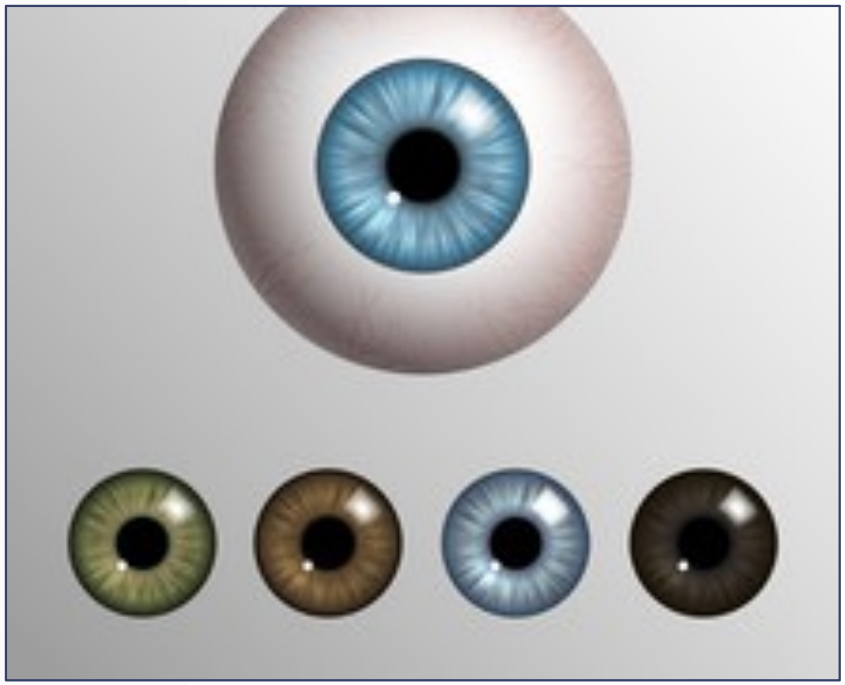
- Gedankliches Durchgehen des Codes oder Ablaufs
- auf Papier oder am Bildschirm
- ohne Ausführung
- Auch bekannt als „dry run“ oder „Walkthrough“
- Schnelles Testen ohne geringen Aufwand
- Jedoch können Fehler übersehen werden

- **Ziel:** Logik- und Verständnisfehler erkennen

Code Review

- Manuelle Prüfung des Quellcodes durch eine andere Person
- oft im Vier-Augen-Prinzip
- Wird vor dem Merge in Branches durchgeführt
- Sehr zeitintensiv, erhöht den Wissensaustausch

- **Ziel:**
  - Fehler erkennen
  - Verständlichkeit erhöhen
  - Coding-Standards sichern



Explorative Tests

- Erfahrungsbasiertes, nicht strukturiertes Testen durch Entwickler oder Tester, um durch Intuition und kreatives Vorgehen unerwartete Fehler zu entdecken
- Ohne vorher definierten Testfällen
- Wird häufig in frühen Projektphasen oder bei Prototypen eingesetzt
- Beispiel: Usability-Tests

Unit Test

- automatisierter Test, mit dem eine einzelne, isolierte Funktionseinheit (Unit) eines Programms überprüft wird (z. B. Methode, Funktion, Klasse)

- **Ziel/Nutzen**
  - Frühes Erkennen von Fehlern bei Änderungen
  - Refactoring unterstützen
  - Erhöhen Vertrauen in neue Releases
  - Grundpfeiler von Test Driven Development (TDD)
  - Unverzichtbar bei Continuous Integration (CI)

Eigenschaften „guter“ Unit Tests

Eigenschaft	Beschreibung
1. Korrekt	Der Test muss das richtige Verhalten testen und verlässliche Aussagen liefern – er darf nicht falsch positiv oder negativ ausfallen
2. Isoliert	Ein Unit-Test darf nur eine Funktion oder Methode testen und muss von anderen Modulen, Datenbanken, APIs etc. unabhängig sein
3. Schnell	Tests sollen innerhalb von Millisekunden laufen – damit sie regelmäßig und automatisiert ausgeführt werden können (z. B. in CI/CD)
4. Aussagekräftig	Der Testfallname und seine Fehlermeldung müssen klar zeigen, was getestet wurde und warum ein Test ggf. fehlschlägt
5. Wartbar	Unit-Tests sollen einfach lesbar und anpassbar sein, wenn sich der Code ändert – ohne viele Nebeneffekte
6. Einfach durchführbar	Der Test sollte automatisiert durchlaufen können, ohne manuelle Eingriffe oder komplexe Testdaten vorzubereiten

Unit Test – Test Doubles

- Platzhalter-Objekte, die echte Komponenten im Test simulieren, um das Verhalten einzelner Module zu testen, ohne von realen Abhängigkeiten (z. B. Datenbank, API) abhängig zu sein

Kriterium	Stub	Mock
Zweck	Liefert vordefinierte Werte zurück	Überprüft, ob bestimmte Methoden aufgerufen wurden
Fokus	Datenbereitstellung	Verhalten und Interaktion
Verwendung	Wenn der Test etwas zurückbekommen soll	Wenn der Test überprüfen soll, was passiert ist
Rückgabewerte	Ja – feste, vorher definierte Werte	Optional, aber meist irrelevant
Verifikation	Keine – der Test prüft nur das Ergebnis	Ja – der Test prüft auch Interaktionen
Typische Anwendung	API antwortet mit Dummy-Daten	Prüfen, ob logError() bei Fehler aufgerufen wird
Tool-Unterstützung	Einfach zu implementieren	Spezielle Mocking-Frameworks hilfreich

E2E-Test

End-to-End-Test

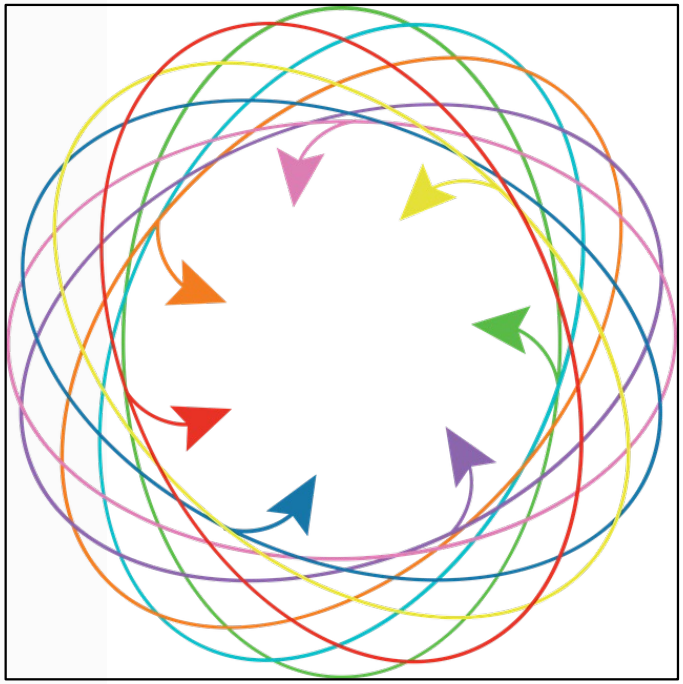
- funktionaler Test, bei dem ein komplettes Anwendungsszenario von Anfang bis Ende durchlaufen wird – aus Benutzersicht
- Überprüfen der gesamten Systemkette (inkl. Benutzeroberfläche, Backend, Datenbank, API, Drittsystemen)

- **Ziel:**
  - Gewährleisten der Gesamtsystemfunktionalität aus Sicht des Endanwenders

Integrationstest

- Test des Zusammenspiels mehrerer Softwarekomponenten oder Module

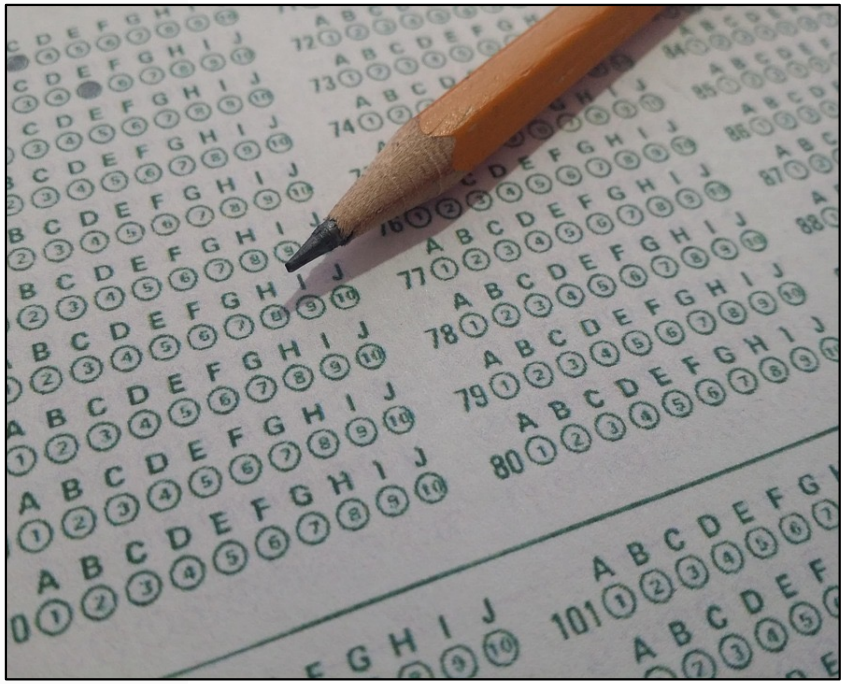
- **Ziel**
  - Fehler in Schnittstellen, Datenflüssen oder Kommunikation zwischen Modulen aufdecken
- **Vorteile:**
  - Erkennt Schnittstellenfehler
  - Testet realistische Abläufe
- **Nachteile:**
  - Fehlerursache schwerer zu isolieren
  - komplexere Testumgebungen nötig



Abnahmetest

- Test durch den Kunden/Auftraggeber
- Prüfen, ob die Software alle Anforderungen erfüllt
- Sorgt für rechtliche Klarheit
- Erfolgreiche Abnahme ist Voraussetzung für Projektschluss, Abrechnung und/oder Go-Live

- **Herausforderungen**
  - Fehlerhafte Spezifikation führen zu “falschen” Tests
  - Aufwendig bei komplexen Systemen

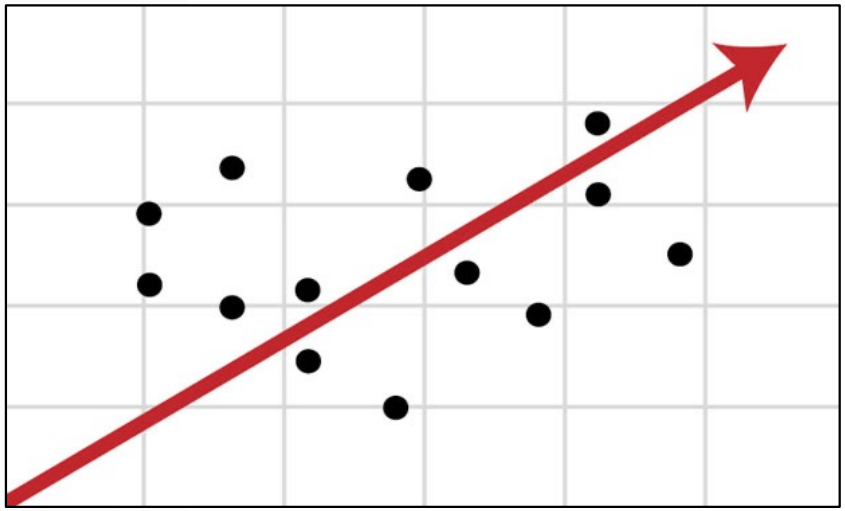


Lasttest/Belastungstest/Performancetest

Testart	Ziel
Lasttest	Test unter erwarteter Last (z. B. 100 Nutzer gleichzeitig)
Belastungstest	Test bei überhöhter Last (Stressszenario – z. B. Black Friday)
Performancetest	Allgemeine Prüfung von Reaktionszeit, Speicherverbrauch, Stabilität

Regressionstest

- Wiederholung von Tests nach einer Änderung (Bugfix, neues Feature)
- Prüfen, ob bestehende Funktionen weiterhin korrekt funktionieren
- Wichtig zur Qualitätssicherung bei Updates
- Erhöht den Testumfang, daher ist eine gute Strategie notwendig



Smoketest

- Schneller, oberflächlicher Test, ob die Software überhaupt grundsätzlich läuft
- Wird oft nach einem neuen Build oder Deployment ausgeführt zur schnellen Funktionsprüfung (funktioniert Login, lädt die Hauptseite)
- Sollte automatisiert durchgeführt werden

- **ACHTUNG:**
  - Deckt nur grobe Fehler auf
  - Kein Tiefentest

Übersicht

Testprozess

- Überblick
- Auswahl des Testverfahrens
- Kriterien für Testergebnisse definieren
- Testdaten generieren und auswählen
- Testprotokoll und Auswertung
- Metriken

Testprozess

- Auswahl des Testverfahrens
- Kriterien für Testergebnisse definieren
- Testdaten generieren und auswählen
- Testprotokoll und Auswertung

Auswahl des Testverfahrens

- welche Art von Testverfahren sollen zur Anwendung kommen – abhängig von Projektphase, Testziel und Umfang

Auswahlkriterien	Mögliche Verfahren
Komponententest / Funktionstest	Unit-Test, Whitebox-Test
Integration von Modulen	Integrationstest
Gesamtverhalten	Systemtest, End-to-End-Test
Kundenanforderungen	Abnahmetest, Akzeptanztest
Leistung unter Last	Last-/Stabilitäts-/Performancetest
Wartbarkeit und Wiederholbarkeit	Regressionstest

- Zudem wird entschieden:
  - manuell vs. Automatisiert / Blackbox- vs. Whitebox-Testing / statisch oder dynamisch / Kombinationen?

Kriterien für Testergebnisse definieren

- Wann gilt ein Test als bestanden?
- Was sind Akzeptanzkriterien für das Verhalten?
- Welche Metriken sollen erhoben werden? z. B.:
  - Antwortzeiten unter 1 Sekunde
  - Fehlerquote ≤ 1 %
  - 100 % Testabdeckung kritischer Pfade
  - Kein kritischer Fehler im Log
- Kriterien werden oft in Testfällen oder Abnahmekriterien dokumentiert und später zur Auswertung herangezogen

Testdaten generieren und auswählen

- Methoden der Testdatengenerierung:
  - Manuell (gezielt ausgedachte Daten für bestimmte Szenarien)
  - Automatisch (Testgeneratoren)
  - Grenz- & Äquivalenzwertanalyse
  - Produktivdaten (anonymisiert)

- Wichtig:
  - Testdaten sollten realistisch, repräsentativ, vollständig, aber auch gezielt fehlerhaft sein, um auch Robustheit und Validierung zu testen.

Testprotokoll und Auswertung 1/2

dokumentierte Nachverfolgung und Bewertung

TESTPROTOKOLL

- Testfall-ID, Tester, Datum, getestete Funktion
- Eingabedaten, erwartetes und tatsächliches Ergebnis
- Status (bestanden/fehlgeschlagen/offen)
- evtl. verknüpfte Fehler-ID (z. B. im Bugtracker)

AUSWERTUNG

- Vergleich Ist vs. Soll
- Berechnung von Kennzahlen (z. B. Fehlerdichte, Erfolgsquote)
- Visualisierung (Diagramme, Metriken)
- Entscheidung: Abnahme, Nachbesserung oder Retest
- ggf. Rückmeldung ins Team/an Entwickler

Testprotokoll und Auswertung 2/2

- Typische Schritte bei der Auswertung von Testergebnissen

Schritt	Inhalt / Ziel
1. Ergebnissammlung	Alle Testergebnisse automatisiert oder manuell erfassen
2. Abgleich Ist vs. Soll	Vergleichen von tatsächlichem Verhalten mit erwarteten Ergebnissen
3. Fehlerklassifikation	Fehler nach Typ, Schwere, Häufigkeit kategorisieren (z. B. mit Bug-Tracking-System)
4. Statistik und Metriken	Testabdeckung, Fehlerrate, Durchlaufquote, Performance-Kennzahlen berechnen
5. Visualisierung/Reporting	Diagramme, Heatmaps, Berichte zur besseren Übersicht für Teams / Management
6. Entscheidung/Maßnahmen	Korrekturen einleiten, Nachtests planen, Freigabe oder Release stoppen

Metriken

Metrik	Bedeutung
Testabdeckung (%)	Wie viel Code wurde durch Tests ausgeführt?
Fehlerrate	Fehler pro Testfall oder pro Zeit
Pass/Fail-Quote	Verhältnis bestandener zu fehlgeschlagenen Tests
Defect Density	Fehler pro 1.000 Zeilen Code (KLOC)
Wiederholbarkeit	Liefern Tests bei gleicher Umgebung konsistente Ergebnisse?

Übersicht

Testkonzepte

- Testdefinition
- Testumfang
- Testdatengeneratoren

Testdefinition

Element	Beschreibung
Welche Art von Test	Unit-Test/Integrationstest/Lasttest etc.
Testobjekt	Welche Funktion/Methode/Komponente/System wird getestet?
Eingabedaten	Welche konkreten Werte oder Bedingungen werden übergeben oder simuliert?
Erwartetes Ergebnis	Was soll genau passieren? Was ist das Soll-Verhalten bei diesen Eingaben?
Testschritte	Wie wird getestet? (z. B. Methode aufrufen, Eingabe simulieren)
Bewertungskriterium	Wann gilt der Test als bestanden/fehlgeschlagen?
Testumgebung	(optional) Rahmenbedingungen: z. B. Betriebssystem, Versionsstand, Abhängigkeiten

Testumfang

Aspekt	Bedeutung / Beschreibung
Funktionaler Umfang	Welche Module, Features oder Use Cases werden getestet?
Technischer Umfang	Welche Technologien, Schnittstellen, Protokolle etc. sind im Test enthalten?
Datenumfang	Wie viele Datensätze/welche Datenmengen werden verarbeitet?
Nutzerlast	Wie viele gleichzeitige Benutzer oder Sessions werden simuliert?
Systemgrenzen	Wann liegt eine Überlastung oder Fehlersituation vor? (z. B. Speicher, Zeit, CPU)
Testzeitraum	Über welchen Zeitraum wird getestet? (z. B. 8 h Dauerlauf)
Grenzwerte	Welche Maximalwerte werden angesetzt? (z. B. max. 10.000 gleichzeitige Requests)

Testdatengeneratoren

- Werkzeuge oder Verfahren, die automatisch Testdaten oder komplette Testfälle erzeugen
- **Ziel**
  - Tests effizient erstellen
  - hohe Testabdeckung erzielen
  - Fehlerquellen systematisch aufdecken
  - manuelle Arbeit reduzieren

Art	Beschreibung
Zufallsgeneratoren	Erzeugen zufällige Testdaten innerhalb definierter Grenzen (z. B. Strings, Zahlen, Zeitstempel)
Modellbasierte Generatoren	Erzeugen Testfälle basierend auf Zustandsdiagrammen, Flussdiagrammen, Use Cases etc.
Codebasierte Generatoren	Nutzen den Quellcode (z. B. zur Pfadüberdeckung) – Whitebox-orientiert
Spezifikationsbasierte Generatoren	Erzeugen Testfälle aus Anforderungen, Tabellen, Regeln – Blackbox-orientiert
Grenzwert-/Äquivalenzgeneratoren	Automatische Generierung relevanter Grenz- oder Äquivalenzwerte

Übersicht

Testfälle ermitteln

- Entscheidungstabellen
- Coverage
- Äquivalenzklassen
- Grenzwertanalyse/Extremwertetest

Entscheidungstabellen

- Darstellung komplexer Abhängigkeiten zwischen mehreren Bedingungen und den jeweils auszuführenden Aktionen in übersichtlicher, vollständiger und widerspruchsfreier Weise
- **Regel:**
  - Vorschrift, welche Aktionen bei Vorliegen einer gegebenen Kombination von Bedingungen durchzuführen sind
- **Regelwerk:**
  - Zusammenstellung unterschiedlicher Regeln
- **Einsatz:**
  - Entwurf von Computerprogrammen
  - Testdatenkonstellationen

Entscheidungs-tabellen - Struktur

Einer Auflistung der zu berücksichtigenden Bedingungen

Einer Auflistung der möglichen Aktionen

Einem Bereich, in dem die möglichen Bedingungskombinationen zusammengestellt sind

Einem Bereich, in dem jeder Bedingungskombination die jeweils durchzuführenden Aktivitäten zugeordnet sind

Tabellenbezeichnung	R1	R2	R3	R4	R5	R6	R7	R8
Bedingungen								
1 Lieferfähig?	j	j	j	j	n	n	n	n
2 Angaben vollständig?	j	j	n	n	j	j	n	n
3 Bonität in Ordnung?	j	n	j	n	j	n	j	n
Aktionen								
A Lieferung mit Rechnung	x							
B Lieferung als Nachnahme		x						
C Angaben vervollständigen			x	x				
D Mitteilen: nicht lieferbar					x	x	x	x

Entscheidungs-tabellen - Eigenschaften

- Vollständigkeit
  - sämtliche möglichen Bedingungskombinationen sind erfasst
- Konsolidierung
  - = Zusammenfasung
  - Führen zwei Regeln zur selben Aktion oder Aktionsfolge UND unterscheiden sich diese nur in einer Bedingungsanzeige, so können diese beiden Regeln konsolidiert werden
- Redundanz
  - Keine identischen Fälle
- Widerspruchsfreiheit (Konsistenz)
  - Keine wiedersprüchlichen Regeln
  - Keine Irrelevanzzeiger

Coverage

STATEMENT COVERAGE

Anweisungsüberdeckung

- Ziel:
  - Alle einzelnen Anweisungen im Code mindestens einmal durch einen Testfall ausführen
- Misst, wie viel Prozent des Quellcodes durchlaufen wurden
- Fokus: Habe ich jede Zeile mindestens einmal getestet?

BRANCH COVERAGE

Zweigüberdeckung

- Ziel:
  - Jeder mögliche Entscheidungsweg (z. B. true/false) wird mindestens einmal durchlaufen
- Deckt beide Richtungen von Bedingungen ab
- Geht über die bloße Anweisungsüberdeckung hinaus

PATH COVERAGE

Pfadüberdeckung

- Ziel:
  - Alle möglichen Ausführungspfade durch das Programm testen Deckt beide Richtungen von Bedingungen ab
- Sehr detailliert – besonders in Schleifen/Kombinationen
- Exponentielle Anzahl an Pfaden → nicht immer vollständig realisierbar

Äquivalenzklassen

- **Ziel:**
  - Eingabemengen in gleichwertige Gruppen (Klassen) aufteilen
  - einen Repräsentanten pro Klasse zu testen
- Annahme: Wenn ein Wert aus der Klasse funktioniert, tun es auch alle anderen
- Spart viele überflüssige Tests

- **Beispiel: Alter in einer App (gültig: 18–99)**
- → Äquivalenzklassen:
  - Gültig: 18–99 (z. B. Testwert = 30)
  - Ungültig: < 18 (z. B. 12)
  - Ungültig: > 99 (z. B. 120)

Grenzwertanalyse/Extremwertetest

- **Ziel:**
  - Gezieltes Testen von Grenzen von Eingabebereichen
- Annahme: Fehler liegen oft genau dort
- Wählt Testwerte am, unter und über den Grenzwerten
- Ergänzt die Äquivalenzklassen sinnvoll

- **Beispiel: PW-Länge von 8 bis 20 Zeichen**

Testwert	Begründung
7	Unterer Randbereich (ungültig)
8	Untere Grenze (gültig)
20	Obere Grenze (gültig)
21	Oberhalb der Grenze (ungültig)