7 Testen

7.1 Test – Grundlagen

Was heißt Qualität?

- Gesamtheit von Eigenschaften und Merkmalen eines Produktes oder einer Dienstleistung, die sich auf deren Eignung zur Erfüllung festgelegter oder vorausgesetzter Erfordernisse beziehen

Qualitätsdimensionen

1) **Produktqualität(=Softwarequalität)**
   - Qualitätsmerkmale (u.a.Funktionalität, Performanz) der **Software**
2) **2. Prozessqualität**
   - Qualitätsmerkmale (Termin, Kosten) des Entwicklung**sprozesses**

- Test ist eine (unter mehreren) Qualitätssicherungsmaßnahme

Bedeutung von Softwaretests

- vollständiges Testen ist bei komplexen Programmen **nicht** möglich
- Ziel: **Testfälle** (= Stichproben) finden, mit denen die Wahrscheinlichkeit am höchsten ist, um festzustellen, ob Software korrekt funktioniert

Testverfahren

1. **White-Box-Test (Glass-Box-Test)**
   - Ziel: Überprüfung des Source Code
   - innere logische Struktur des zu testenden Elements muss bekannt sein
   - möglichst alle Zeilen des Source Code sollen durchlaufen werden (siehe Überdeckungsgrade)

   ⇒ Einsatz von Test-Werkzeugen erforderlich

von innen!

2. **Black-Box-Test**
   - Ziel: prüft funktionale Korrektheit des Systems aus Anwendersicht
   - zu testendes Element wird nur von außen betrachtet, innere Struktur nicht bekannt
   - Frage: reagiert das System korrekt auf alle Eingabesituationen?

von außen!

Testfallentwurfsverfahren

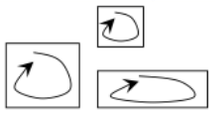
- Zusammenfassung Testfallentwurfsverfahren

	Testfallentwurfs-verfahren	Passende Testverfahren
1.	Äquivalenzklassenbildung	Black Box
2.	Grenzwertanalyse	Black Box
3.	Überdeckungsgrade	White Box
4.	Bedingungsüberdeckung	White Box
5.	Erfahrungsbasierte Verfahren	i.W. Black Box

Teststufen

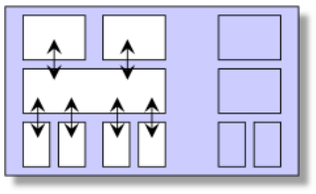
- Tests begleiten den gesamten Entwicklungsprozess
- Integrationstest: es werden nur die Ein- und Ausgaben und das Verhalten an den Schnittstellen beobachtet, innere Struktur ist egal, da die davor im Komponententest getestet wurde

1. Komponententest (Unit-Test)
- Testobjekt: „autarke“ Komponente, z.B. Klasse, Modul, Programm
- Testverfahren: i.d.R. White-Box-Test
- alle Programmteile müssen getestet werden



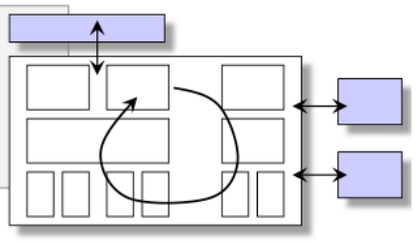
1. Code Check

2. Integrationstest
- Testobjekt: teilintegrierte Komponenten mit Komponenteninteraktion, z.B. Subsystem, mehrere Subsysteme, Schnittstellen
- Testverfahren: Black-Box-Test

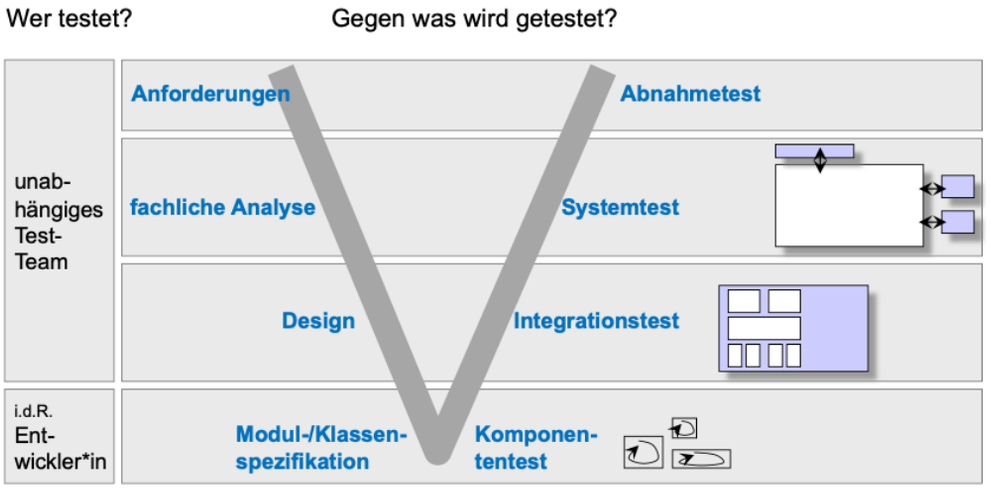


2. Integrationstest der Komponenten

3. Systemtest
- Testobjekt: Gesamtsystem mit Nachbarsystemen
- Testverfahren: Black-Box-Test



3. Funktionstest des Systems



Bestimmung von Testfällen

- Problem: es können nicht alle Eingaben überprüft werden, d.h. ein vollständiger Test ist nicht möglich
- Ziel: mit möglichst wenig Testfällen (Stichproben) möglichst große Wirkung erzielen

- **Testfall**(-beschreibung) (test case) umfasst (zumindest):
  a) Voraussetzungen, die für Durchführung gegeben sein müssen
  b) Werte aller Eingabedaten
  c) erwartete Ausgaben/ Ergebnisse (= Soll-Resultat (expected result))

Testfallentwurfsverfahren: ein guter Testfall ist
- **repräsentativ**: steht stellvertretend für möglichst viele andere Testfälle
- **fehlersensitiv**: weist eine hohe Wahrscheinlichkeit auf, einen Fehler zu entdecken
- **redundanzarm**: prüft nicht erneut, was andere Testfälle auch schon prüfen

7.1.1 Bestimmung von Testfällen – Äquivalenzklassen

Äquivalenzklassenbildung

- Äquivalenzklasse: Zusammenfassung von Eingaben, die zu einem gleichen Systemverhalten führen
- Bei Eingaben teste, hier dann typische Werte für a) dann zwischen 1 und 31, aber nicht die Grenzen. Die Grenzen sind besondere Werte

Äquivalenzklassen

- für jede Funktion/ Methode werden Eingabebereich und Systemreaktionen (disjunkt) klassifiziert
  - alle Eingaben(-bereiche) mit ähnlicher Systemreaktion bilden eine Äquivalenzklasse
  - für jede Äquivalenzklasse werden nur wenige (typische) Eingaben (Repräsentanten) betrachtet und entsprechende Testfälle formuliert

Beispiel: Eingabebereich ist ein zusammenhängender Wertebereich, z.B.1 ≤ Tage ≤ 31

- Äquivalenzklassen für dieses Beispiel: (Eingabeber. / Systemreaktion)
  a) 1 ≤ Tage ≤ 31 / (gültig)
  b) Tage<1 /(ungültig)
  c) Tage>31 /(ungültig)

Erzeugung von Testfällen aus Äquivalenzklassen

1) **gültige Äquivalenzklassen:**
   - möglichst viele Äquivalenzklassen in einem Test kombinieren
2) **2. ungültige Äquivalenzklassen:**
   - ein Testfall pro ungültiger Äquivalenzklasse (kombiniert mit Werten aus ausschließlich gültigen Klassen)
   - für alle ungültigen Eingabewerte muss Fehlerbehandlung existieren

Beispiel : Äquivalenzklassen mit Testfällen für dieses Beispiel

a) 1 ≤ Tage ≤ 31 → Testeingabe: 5 / Ergeb.: gültig
b) Tage<1 → Testeingabe: -23 / Ergeb.: ungültig
c) Tage>31 → Testeingabe: 777 / Ergeb.: ungültig

7.1.2 Grenzwertanalyse

Grenzwertanalyse (für Black-Box-Test)

- **Grenzwerte** (von Daten) sind **Werte**, die entweder **gerade noch innerhalb** oder **gerade außerhalb des Wertebereichs** liegen

- Grenzwerte stellen häufige Fehlerquellen dar
- Grenzwerte liegen u.a. an den Grenzen der Äquivalenzklassen

Grenzwertfehler, z.B.

- verwechselte Vergleichsoperatoren, z.B. "<" statt "≤"
- Rundungsfehler, Konvertierungsfehler, falsch abgebrochene Schleifen
- bei Strings: leere Strings, Strings mit Maximallänge

- **für jeden Parameter einer Methode werden die Grenzwerte betrachtet und für jeden Repräsentanten entsprechende Testfälle formuliert**

Grenzwertanalyse für dieses Beispiel

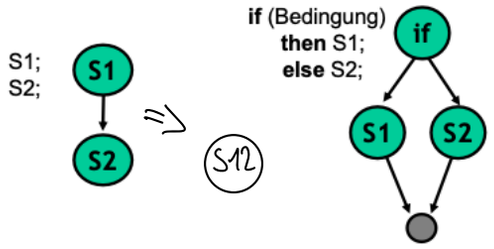
- zusätzliche Testfälle für die Werte **(-1), 0, 1, 31, 32,** da diese mit einer überdurchschnittlichen Wahrscheinlichkeit Fehler produzieren

– 7 Testfälle

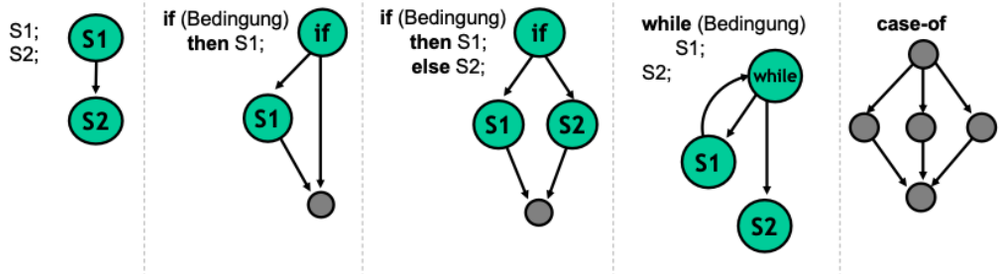
7.1.3 Überdeckungsgrad

Überdeckungsgrade (für White-Box-Test)

- welche Ausführungspfade des Codes werden durch Testfälle überprüft ?
- werden überhaupt alle Code-Anweisungen durchlaufen oder gibt es Code-Teile, die beim Testen nie ausgeführt werden ?
- **Ziel: durch Testfälle möglichst große Testabdeckung (Code Coverage), d.h. getestete Code-Teile, erreichen**
- **betrachtet wird der Kontrollfluss im Code** (dargestellt als Kontrollflussgraph mit Anweisungsfolgen)



- sequentielle Anweisungen werden i.d.R. zu einem Knoten zusammengefasst

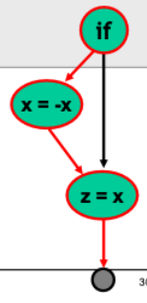


- Ziel: Durchlauf „repräsentativer" Pfade beim Testen

- Bei Eingabe x=-2 ist c0=3/3=1 => 100% => alle Anweisungen werden ausgeführt

- **Überdeckungsgrade** (Code Coverage)
  - **C_x-Überdeckung**, wobei x die geforderte Länge der möglichen Kantenfolgen angibt, welche durch Testfälle abgedeckt sein müssen

- **C_0-Überdeckung** (Knoten-/ Anweisungsüberdeckung (statement coverage))
  - alle Anweisungen sollen mindestens einmal ausgeführt werden, d.h. alle Knoten des KFG müssen mind. einmal besucht werden, also Überdeckung aller Kantenfolgen der Länge 0
  - Testmaß: $C_0 = \frac{\text{(Anzahl besuchter Knoten)}}{\text{(Anzahl aller Knoten)}}$
  - z.B. jede if-Anweisung soll ausgeführt werden
  - z.B.:
```
if (x < 0) then x=-x;
z = x;
```
  - (x = -2) erfüllt Anweisungsüberdeckung



$x = -2$

$C_0 = \frac{3 \text{ besuchte Knoten}}{3 \text{ ges Knoten}} = 100\%$

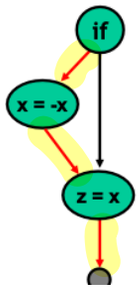
- **C_1-Überdeckung** (Zweigüberdeckung (branch coverage))
  - alle Verzweigungen/ Kanten im Kontrollflussgraphen werden mindestens in einem Testfall durchlaufen
  - Testmaß: $C_1 = \frac{\text{(Anzahl besuchter Kanten)}}{\text{(Anzahl aller Kanten)}}$
  - z.B.:
```
if (x < 0) then x=-x;
z = x;
```
  - Testeingaben (x = -2) und (x = 2) erfüllen Zweigüberdeckung

Eingabe

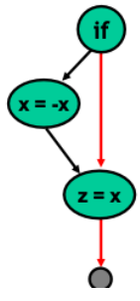
- x= -2 → c1=3/4 (75%)
- x= 2 → c1=2/4 (50%)

→ zusammen 100%

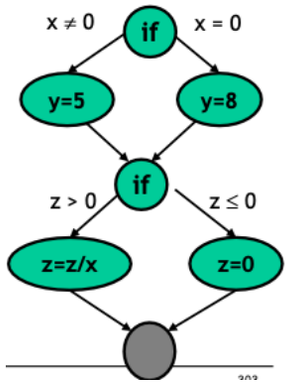
(x = -2)
Überdeckung 3/4



(x = 2)
Überdeckung 2/4



- **$C_{unendlich}$-Überdeckung** (Pfadüberdeckung)
  - jede mögliche Folge von Anweisungen durch Testfälle abdecken



- ist i. allg. nur in Näherung zu erreichen
- bei Schleifen gibt es meist unendlich viele Pfade
- Boundary-Tests: Anzahl der Schleifendurchgänge beschränken

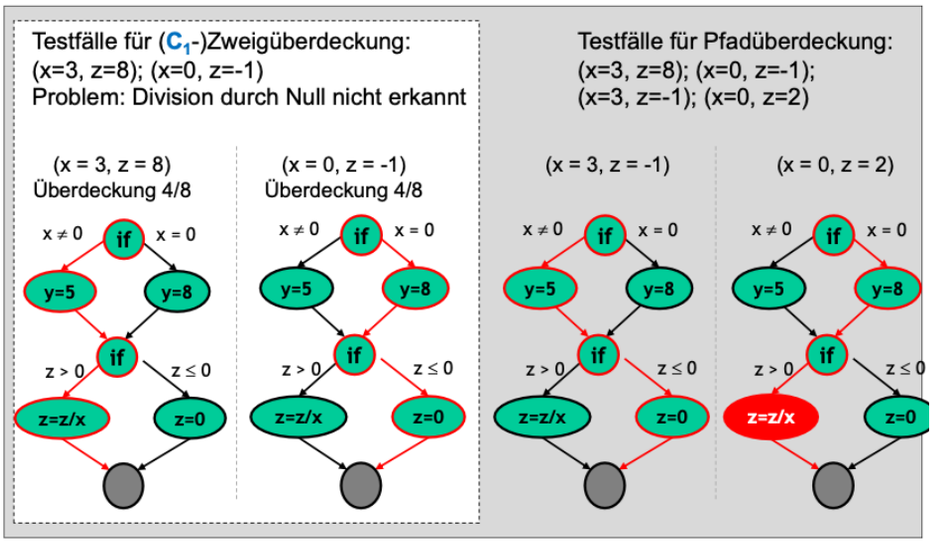
Bsp:

- ```
  IF (x!=0)THEN y=5;        ELSE y=8;
  IF (z>0) THEN z=z/x;      ELSE z=0;
  ```
- Problem: mögliche Division durch Null erkennen (wenn x=0 und z > 0)

Jeden möglichen Pfad ablaufen. Hier 4
Eingabe

- x=3 u. z=8 → c1=4/8 => 50%
- x=0 u. z=-1 → c1=50%
  → zusammen 100%
  → dann automatisch auch 100% c0 Überdeckung
- cUnendlich noch nicht erfüllt 2 Pfade fehlen noch
- x=3 u. z=-1
- x=0 u. z = 2 →Fehler durch 0 teilen durch cUnendlich-Überdeckung erkannt, aber nicht durch c1-Überdeckung

In der Realität ist c0-Überdeckung=100% und c1-Überdeckung so hoch wie möglich (meistens 80%) angestrebt



Bewertung Überdeckungsgrade

Anweisungsüberdeckung (C0-Überdeckung):

- Vorteile: einfach, geringe Anzahl von Testfällen (Eingabedaten), nicht ausführbare Programmteile werden erkannt
- Nachteil: logische Aspekte werden nicht überprüft

Folgerung: Zweigüberdeckung (C1-Überdeckung) ist anzustreben

- impliziert Anweisungsüberdeckung
  (= jede C1-Überdeckung ist auch eine C0-Überdeckung)
- erfordert die Ausführung aller Zweige eines KFG, indem jede Entscheidung mindestens einmal wahr und einmal falsch
- Nachteile der Zweigüberdeckung:
  - Kombinationen von Zweigen sind unzureichend geprüft
  - komplexe Bedingungen werden nicht analysiert
  - Schleifen werden nur unzureichend analysiert
  ```

7.1.4 Bedingungsüberdeckung

- Ziel: Testfälle speziell für Bedingungen in bedingten Anweisungen und Schleifen auswählen

> • **Bedingungsüberdeckung** (condition coverage)
>
> – Überprüfung der Kombinationen des logischen Ausdrucks in einer Bedingung/ Schleife
>
> – Testmaß:
> (Anzahl wahrer + Anzahl falscher Ausdrücke) / 2*alle Ausdrücke

– z.B.
```
if ((A and B) or (C and D))
    then statement1;
    else statement2;
```

– mit den atomaren Prädikaten `A, B, C, D`

einfache Bedingungsüberdeckung (simple condition coverage)

- jedes atomare Prädikat muss je einmal TRUE und einmal FALSE ergeben
- Testmaß#:

$$C2 = \frac{\text{(Anzahl wahre Atome) + (Anzahl falsche Atome)}}{\text{2 * (Anzahl alle Atome)}}$$

– `if ((A and B) or (C and D))`

Testfall	A	B	C	D	Resultat
1	true	true	false	false	true
2	false	false	true	true	true

- Achtung: einfache Bedingungsüberdeckung subsumiert nicht die Anweisungs- und Zweigüberdeckung

Bedingungs-/ Entscheidungsüberdeckung (condition/decision coverage)

– jedes atomare Prädikat muss je einmal TRUE und einmal FALSE ergeben
– und logischer Gesamtausdruck muss je einmal TRUE und einmal FALSE ergeben

– `if ((A and B) or (C and D))`

Testfall	A	B	C	D	Resultat
1	true	true	true	true	true
2	false	false	false	false	false

– **Bedingungs-/Entscheidungsüberdeckung subsumiert Zweigüberdeckung (und damit auch Anweisungsüberdeckung)**

Mehrfach-Bedingungsüberdeckung (multiple condition coverage)

– Test aller Wahrheitswertekombinationen der Einzelterme
– Anzahl Testfälle: 2n, d.h. i.d.R. nicht praktisch anwendbar

– `if ((A and B) or (C and D))`

Testfall	A	B	C	D	Resultat
1	false	false	false	false	false
2	true	false	false	false	false
3	true	true	false	false	true
...	...	...	...	...	...
16	true	true	true	true	true

7.2 Test – Ergebnisse

Produkte im Arbeitsschritt Test

1) **Testplan**
   - Rahmenbedingungen, Teststrategie, Planung des Testablaufs
2) **Testfälle**
   - Voraussetzungen, Eingabedaten und erwartete Ausgabedaten für ein Testobjekt (Komponente, integrierte Komponenten, Gesamtsystem)
3) **Testbericht/-dokumentation**
   - Protokoll/ Überblick über gesamten Testvorgang
4) **Fehlerbericht**
   - detaillierte Beschreibung aller Abweichungen vom in den Testfällen spezifizierten Verhalten des Systems

7.3 Test – Vorgehen

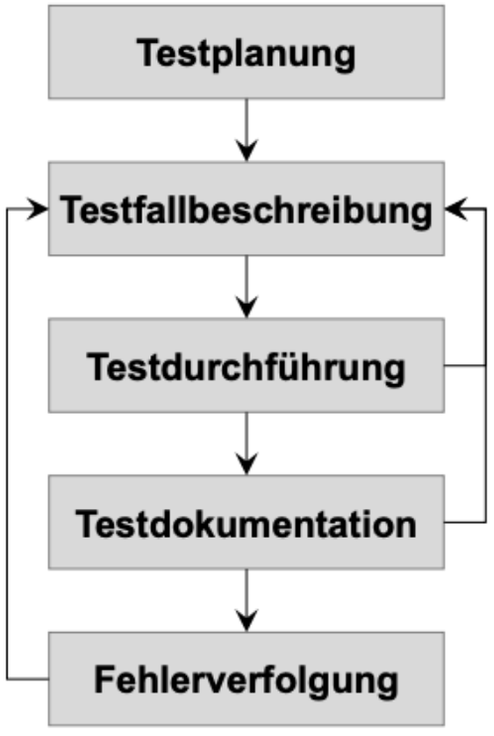
1. Systematischer Testprozess

- anzustrebende Ziele des Arbeitsschritts Test:

1) **Reproduzierbarkeit**
   - Testergebnisse entstehen nicht zufällig, sondern systematisch und nachvollziehbar und lassen sich wiederholen
2) **Planbarkeit**
   - Aufwand und Nutzen (gefundene Fehler) der Tests können prognostiziert werden
3) **Wirtschaftlichkeit**
   - Aufwand und Nutzen werden optimiert, durch Testfall- Optimierung, Werkzeugeinsatz, optimierten Testprozess
4) **Risiko-und Haftungsreduktion**
   - systematischer Testprozess führt zu (gewisser) Sicherheit, dass kritische Fehler nicht auftreten

2. Systematisches Vorgehen

- Arbeitsschritt Test kann als Projekt im Projekt betrachtet werden
  - Tests werden geplant, entworfen, implementiert und ausgeführt
- Tätigkeiten im Arbeitsschritt Test



7.5 Test-Driven Development

- Testen in klassischer Softwareentwicklung: Code-and-Test
  1. zuerst Anforderungen implementieren,
  2. dann Testfälle zum Prüfen der Anforderungen schreiben (u.U. auch parallel zur Implementierung)
- mögliche negative Erfahrungen:
  - schlechte Testbarkeit der Software
  - Erstellung der Tests unter Zeitdruck (da Software ja bereits „fertig“)
  - Gefahr, dass die Testfälle entsprechend des Codes und nicht der Anforderungen entworfen werden („Betriebsblindheit“, „um Fehler herum testen“)
- ein Testfall kann zwei unterschiedliche Resultate haben:
  1) **bestanden(grün):** Testergebnis entspricht den Erwartungen
  2) **nichtbestanden(rot):** Testergebnis weicht von Soll-Ergebnis ab
- Ziel bei klassischer Entwicklung: alle Testfälle sind grün

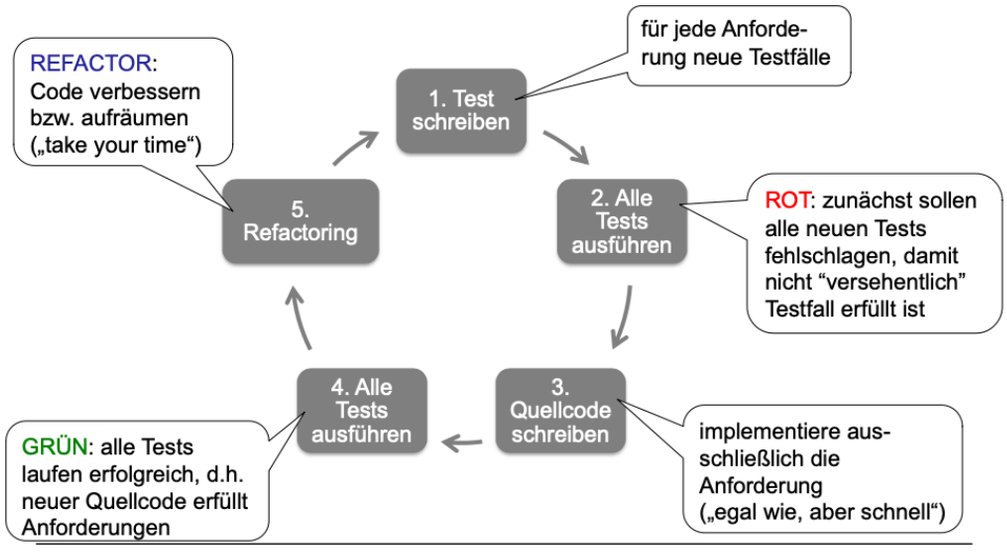
Integration von Test und Entwicklung

- tests first-Ansatz

> • **Test-Driven Development (TDD – Testgetriebene Entwicklung):**
>
> Entwicklungsstil, bei dem
>
> 1. die Testfälle **zuerst** geschrieben werden (d.h. vor der Implementierung),
> 2. eine umfassende Menge von Entwickler*innen-Tests gepflegt wird,
> 3. kein Code produktiv geht, ohne das dazugehörige Test existieren.

- TDD entstand aus agiler Softwareentwicklung (siehe SE2)
- TDD führt zu
  - **benutzerorientierter Programmierung**: zunächst Sicht auf Verwendung einer Funktion, bevor diese implementiert wird
  - gut testbarem Code, da Testen die Implementierung steuert
  - besseren Testfällen, da diese entsprechend der Anforderungen entworfen werden

• **Mikro-Iterationen** über 5 Schritte („Rot, Grün, Refactor“-Prinzip)



Bewertung

TDD-Vorteile

- Testfälle beziehen sich auf konkrete Code-Bereiche (d.h. kein Code ohne dazugehörige Testfälle)
- **zwingt frühzeitig über das API/ die Schnittstelle nachzudenken**
- **Mikro-Iterationen führen zu weniger Fehlern**
- Bestand an Unit-Tests dokumentiert den Code
- Work for Outcome: YAGNI (You Aren‘t Gonna Need It)
- Ping-Pong passt gut zu Pair Programming (Paarprogrammierung)

TDD-Kritik

- **kann falsch eingesetzt werden**, insbesondere von unerfahrenen Programmierern
- kann Hürde für größere Code-Änderungen darstellen (da dann viele Testfälle fehlschlagen würden)
- **Fokussierung auf Testbarkeit statt auf Problemlösung**
- Probleme bei „bad data“ Testfällen, also wenn Testdaten unvollständig oder fehlerhaft sind
- **erfordert ggf. höheren Aufwand** (reduziert dafür aber das Risiko!)

TDD Einsatz

- auf Komponenten(Unit-)Tests ausgelegt (Programmieren im Kleinen) – zusätzliche Integrations- und Systemtests i.d.R. notwendig

---

7.6 Zusammenfassung

- **Test** ist eine analytische QS-Maßnahme
- Test**verfahren**: White-Box-, Black-Box-Test
- Test**stufen**: Komponenten-, Integrations-, Systemtest
- **Testfallbestimmung**: Äquivalenzklassen, Grenzwertanalyse, Überdeckungsgrade, Bedingungsüberdeckung, erfahrungsbasiert
- Arbeitsschritt Test: (Integrations-)Test des Gesamtsystems bzw. Stufe
  - Testplanung, (2) Testfälle, (3) Testdurchführung, (4) Testdokumentation, (5) Fehlerverfolgung
- **automatisierte** Durchführung der Tests und Auswertung der Ergebnisse
- **TDD** als eigener Entwicklungsstil
- es gibt **keine fehlerfreie Software**!!!