Praktische Informatik 1

Hartmut Lackner

8.07.2010

11 Software-Qualität

1. Was ist Software-Qualität
   - Qualitätskriterien
   - Maßnahmen zur Qualitätssicherung
2. Softwaretest
   - Testen im Software-Entwicklungsprozess
   - Ebenen des Softwaretests
3. Unit-Test
   - Junit in Eclipse
   - Demo
4. Programmverifikation

Qualitätssicherung

- Wiederverwendung eines Bauteils aus der Ariane 4 für die Ariane 5
- Flugbahn der Ariane 5 weicht von der, der Ariane 4 ab → Höhere Beschleunigungswerte
- Konvertierung dieses Wertes als 64-bit floating point Variable in einen 16-bit signed Integerwert führt zum Überlauf → Exception
- Die Exception wird nicht abgefangen → Absturz des Flugsteuerungssystems
- Das abgestürzte System und dessen Backup-System senden Diagnosedaten an den OBC → Der OBC hält die Daten für Steuerdaten
- Der OBC übersteuert die Rakete → Die Rakete bricht auseinander
- Der OBC erkennt das Auseinanderbrechen → Selbstzerstörung

11.1 QUALITÄTSKRITERIEN

11.1 Qualitätskriterien

Qualität nach DIN:

> *„Qualität ist die Gesamtheit von Eigenschaften und Merkmalen eines Produkts oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse beziehen." (DIN 55350)*

Merkmale qualitativ hochwertiger Software-Produkte:

Extern (Benutzer)

- Funktionalität
- Benutzbarkeit
- Effizienz
- Performanz
- Robustheit
- Zuverlässigkeit
- Sicherheit

Intern (Entwickler)

- Wartbarkeit
- Wiederverwendbarkeit
- Änderbarkeit
- Übertragbarkeit
- Lesbarkeit

Methoden der Qualitätssicherung

Konstruktiv (Vorgehensmodelle)

- V-Modell
- Agile Prozesse (Extreme Programming)
- ...

Analytisch (Verfahren)

- Testen
  - Dynamisch
  - Statisch
- Verifizieren
  - Verifikation
  - Symbolische Ausführung
- Vermessen
  - Metriken
  - Analyse der Bindungsart

Ein studentischer Software-Entwicklungsprozess

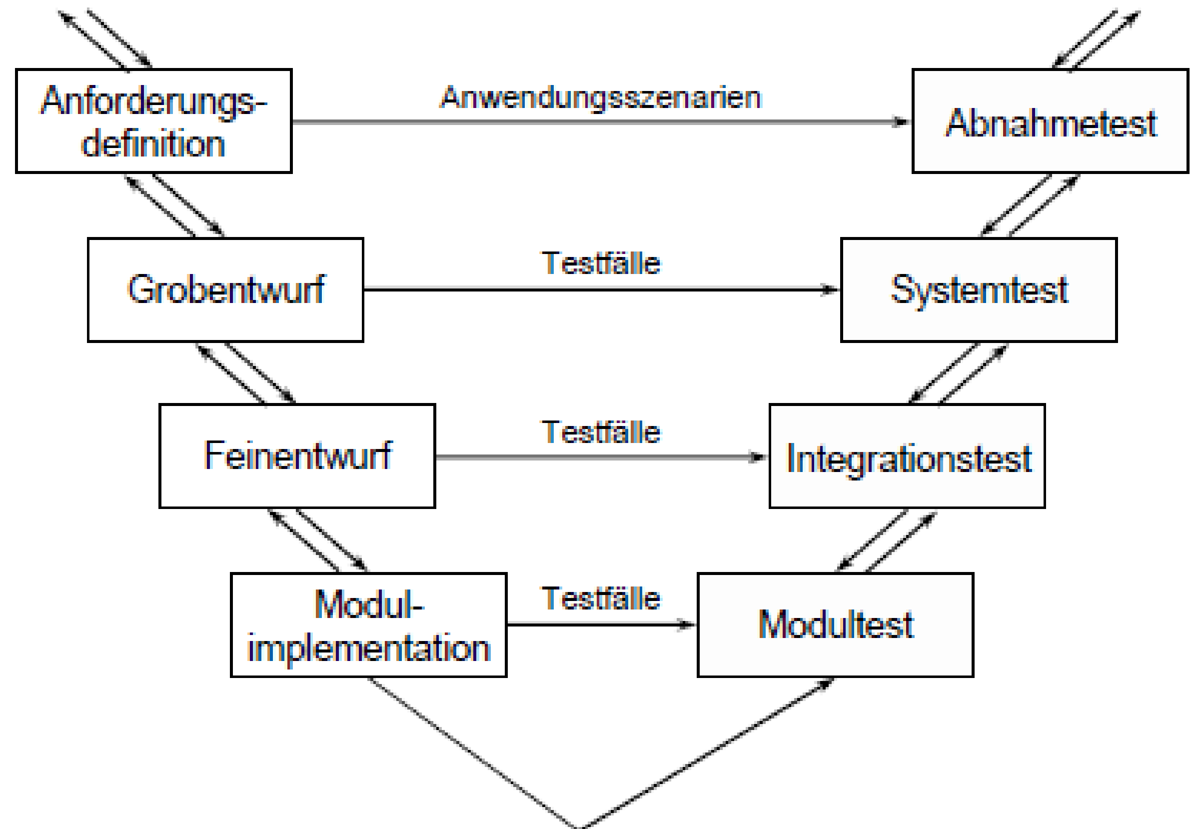
- Wie entwickelst Du Software (für die Übung)?
  - Abgabe ist am Montag nächster Woche um 16 Uhr

1. Verschieben der Programmiertätigkeit
   - Bei dem Wetter kann man gar nicht arbeiten
   - Am Wochenende spielt Deutschland!
2. Das Programm wird am Montag nach der Mittagspause zusammen gehackt.
3. Solange bis die Ausgaben den Erwartungen (grob) entsprechen, maximal bis 16 Uhr.

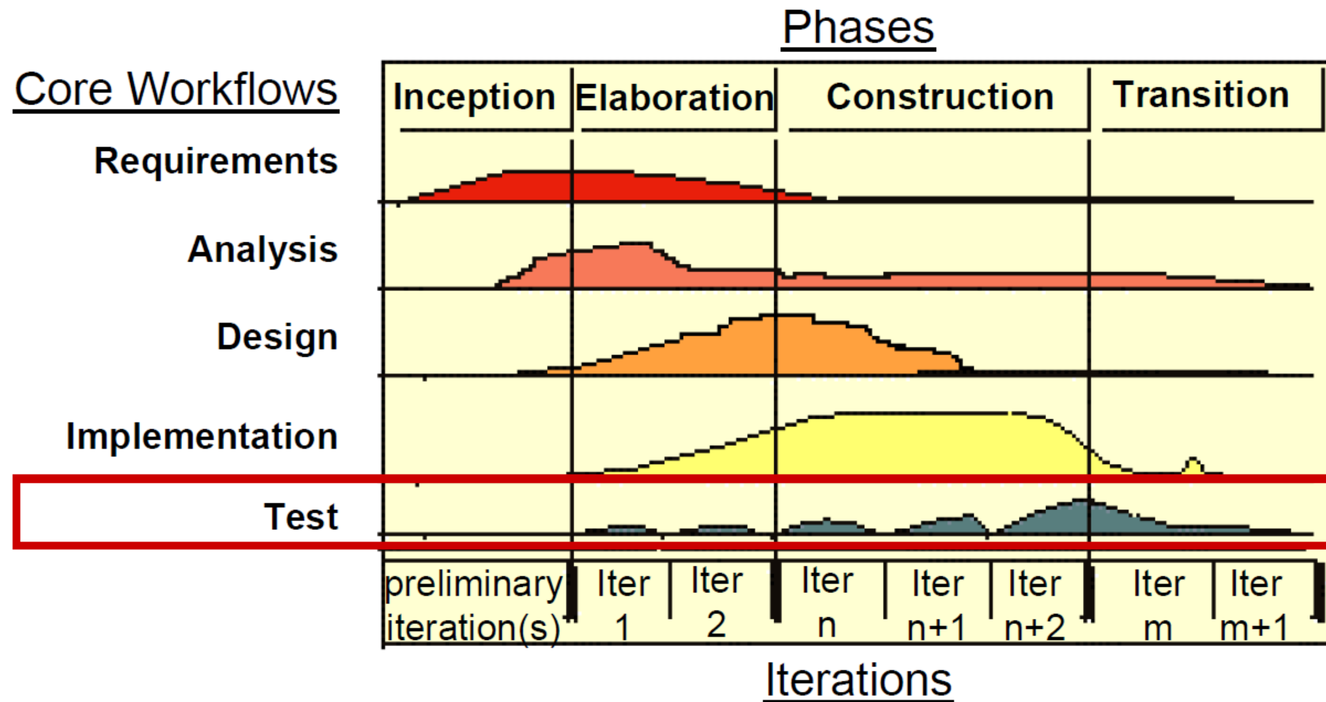
→ Abgabe

Ein professioneller Software-Entwicklungsprozess

- Das V-Modell als Beispiel für einen Entwicklungsprozess



Verteilung der Aufwände



- Aufwand für die einzelnen Phasen der Software-Entwicklung im Vergleich (normiert)

Kostenfortpflanzung eines Fehlers

		Zeitpunkt der Erkennung des Fehlers				
		Anforderungen	Architektur	Entwurf	Test	Release
Zeitpunkt der Einführung des Fehlers	Anforderungen	1x	3x	5-10x	10x	10-100x
	Architektur	-	1x	10x	15x	25-100x
	Implement.	-	-	1x	10x	10-25x

McConnell, Steve (2004). *Code Complete* (2nd ed.)

- Fehler möglichst frühzeitig erkennen
- In allen Phasen aktiv gegen Fehler vorgehen
- Systematische Erkennung aller Fehler ist nicht möglich

11.2 SOFTWARETEST

11.2 Softwaretest

"Test is an activity performed for evaluating product quality and for improving it, by identifying defects and problems."

(SWEBOK)

- Tests werden nur stichprobenartig durchgeführt.
- Der vollständige Test eines Programmes ist zu umfangreich (außer in trivialen Fällen)
- Deshalb gilt: Tests können stets nur die Anwesenheit, nie aber die Abwesenheit von Fehlern beweisen. (Djikstra)

Phase	Relativer Aufwand
Anforderungsanalyse	10%
Spezifikation	10%
Entwurf	15%
Implementierung	20%
Test	45%

J. Martin, C. McClure(1983)

Vorgehen zur Testerstellung

1. Testplanung
   - Definition der Testziele
   - Umfang und Komplexität der Tests
2. Testerstellung
   - Implementierung der Tests
3. Testausführung
   - Manuell oder automatisch
   - Nicht vom Entwickler durchzuführen
4. Testauswertung
   - Erstellung des Testprotokolls
5. Testanpassung (weiter mit 3)
   - Verfeinerung der Tests

Teststufen

- Abnahmetest / Akzeptanztest
  - Test der gelieferten Software durch den Kunden
- Systemtest
  - Test des gesamten Systems gegen die Anforderungen
- Integrationstest
  - Test der Zusammenarbeit voneinander abhängiger Komponenten
- Komponententest / Modultest
  - Test größerer Einheiten: Module oder Komponenten
- Unit Test
  - Kleinste zu testende Einheit: Units oder Klassen

Systemtest

- Konzentration auf Benutzersicht
  - Test über die GUI
  - Test über Sensorik/Aktuatorik („HiL“)
- Testwerkzeug stimuliert das System „von außen“ und überwacht die Reaktionen des Systems „nach außen“
  - automatischer Zugriff auf diese Schnittstellen ist nicht immer einfach

Beispiel: GUI-Test

- *Capture-Replay-Technik*
- bekannteste Vertreter: IBM Rational Robot, HP QuickTest Professional

- **Idee**
  - Aufzeichnung von Benutzerinteraktionen
  - Abspielen mit Vergleich auf Änderungen

- **Erweiterungen**
  - GUI-Map, Scripting Wizard
  - Alternativen, Wiederholungen
  - Checkpoints, Komparatoren

Vorgehensweise

- **pragmatisch**
  - Aufzeichnung von Experimenten, Replay bei neuen Produktversionen
- **systematisch**
  - Festlegung von Testzielen (Qualitätskriterien, Requirements)
  - Definition von Testfällen und Testsuiten
  - Umsetzung (Implementierung) der Testfälle
  - automatische Durchführung, Ergebnisabgleich und Korrektur

Capture-Replay im Browser

DEMO: SELENIUM

Integrationstest

- Vorgehensweisen: Bottom-up-, Top-down-, Big-Bang- und Sandwich-Methode
- Methode: Stümpfe und Treiber
  - Stumpf (stub): leere oder minimale „Pseudo-Methode“, liefert z.B. statt Berechnungswert eine Konstante
  - Treiber (driver): Methode, die nur zu Testzwecken andere Methoden aufruft (vgl. JUnit)
- Beispiel: Web-Shop

Integrations- und Systemtest

- Systemtest
  - geht davon aus, dass die Komponenten (korrekt) zusammengesetzt wurden
  - versucht, das korrekte Gesamtverhalten des Systems zu gewährleisen
  - Systematik: nach den Benutzungsschnittstellen
- Integrationstest
  - setzt auf getesteten, (korrekten) Komponenten auf
  - versucht die korrekte Interaktion der Komponenten zu garantieren
  - Systematik: nach den internen Schnittstellen des Systems

Komponententest

- Umfassender Test einer einzelnen Komponente
- Vorstufe des Integrationstests
- Nur schwierig vom Unit-Test abzugrenzen
- Häufig Synonym mit Unit-Testing verwendet
- Unit-Testing Frameworks konnen häufig auch zum Komponententest verwendet werden

Unit Test: Konventionen

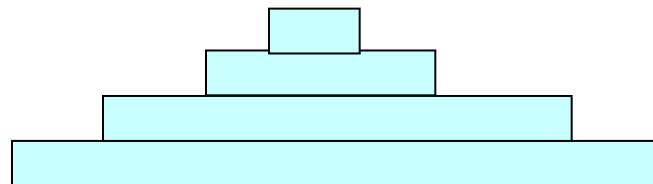
- Erstellung von Unit-Tests gehört zur sauberen Programmierung!
- Testfälle können in Testsuiten strukturiert werden (Baumartige Hierarchie)
- Methoden setUp() und tearDown() zum Aufbau bzw. Abbau dynamischer Datenstrukturen, die für jeden Test gebraucht werden; wird vor und nach *jedem* Einzeltest aufgerufen

Probleme

- Einkapselung: private Methoden können von anderen Klassen nicht aufgerufen werden
  - C++: Konzept der „Friend“-Klasse
  - Java: „Wrapper“-Methoden nur für Testzwecke?
- Vererbung: Wann werden ererbte Methoden getestet?
  - machen oft nur im vererbten Kontext Sinn
  - müssen immer wieder neu getestet werden?
- Polymorphie: gleiche Tests für unterschiedliche Funktionalitäten?

Vorgehensweise

- Bottom up:
  - Start mit den Klassen, die von keinen anderen abhängen
  - Test aller Methoden aller dieser Klassen
  - Abdeckungskriterien: Alle Datenfelder werden geschrieben, alle Anweisungen ausgeführt, alle Zweige durchlaufen
  - Danach werden die Klassen getestet, die auf den schon getesteten aufbauen
- Schichtenartige Systemsicht
  - Testsuiten werden baumartig gemäß dieser Hierarchie strukturiert



JUnit

- einfache, effektive Testumgebung
- integriert in Eclipse
  - automatische Generierung von Stümpfen
- benutzt Java als Testsprache für Java
  - CUnit, CPPUnit, NUnit, …
  - Test-Code als integraler Bestandteil der SW-Artefakte
- Kontrollierter Aufruf aller Methoden einer Klasse
- Nachbedingungen (Assertions) zur Verdiktbildung

in Eclipse

- SUT mit zu testenden Klassen erstellen
- neuen „JUnit Test Case“ erstellen
  - Stümpfe werden bereit gestellt
  - zur Methode myMethod() gehört testMyMethod()
- Testfälle eintragen

```
public final void testTriangleTypeEquilateral() {
 assertEquals(Type.equilateral,
  Triangle.triangleType(2,2,2));
}
public final void testTriangleTypeIsosceles() {
 assertEquals(Type.isosceles,
  Triangle.triangleType(1,2,2));
}
```

Testgetriebene Entwicklung mit Eclipse

DEMO: JUNIT

Systematischer Test

- Wann habe ich ein Programm ausreichend getestet?

Test everything that could possibly break.
(Jeffries 2000)

- Wie hoch ist das *akzeptable Fehlerniveau*?
  → Aufwand wächst exponentiell zum Nutzen
- Tests schaffen Vertrauen in die Arbeit:
  → zu viele Tests bremsen die Entwicklung
  → zu wenige schaffen trügerisches Vertrauen
- Die Art der gemachten Fehler eines Teams ändert sich über die Projektlaufzeit

Testabdeckung

Kontrollflussorientierte Verfahren

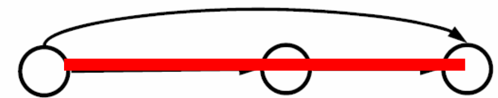
- Anweisungsüberdeckung (C0):
  - Jeder Ausdruck wurde mindestens einmal ausgeführt:
- Zweigüberdeckung (C1):
  - Alle Kanten des KFG durchlaufen
- Minimale mehrfach Überdeckung:
  - Jeder Teil einer Bedingung muss einmal zu True und zu False evaluiert worden sein
- Boundary-Interior-Pfadtest:
  - Schleifenkörper ein und zwei Mal durchlaufen
- Pfadüberdeckung (4)

Anweisungsüberdeckung

- Forderung: Jeder Ausdruck wurde mindestens einmal ausgeführt.
- D.h.: Jede Anweisung und jeder Ausdruck wird einmal ausgeführt
- Nur sehr schwaches Kriterium: 18% der Fehler werden gefunden

  → Warum?
- Nicht Praxis relevant

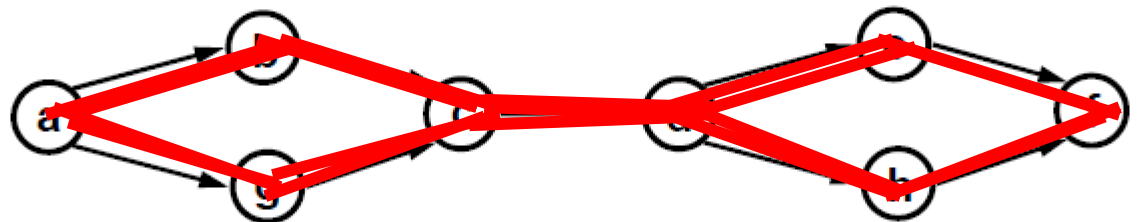
```
if B S1;
```



```
do S while B;
```

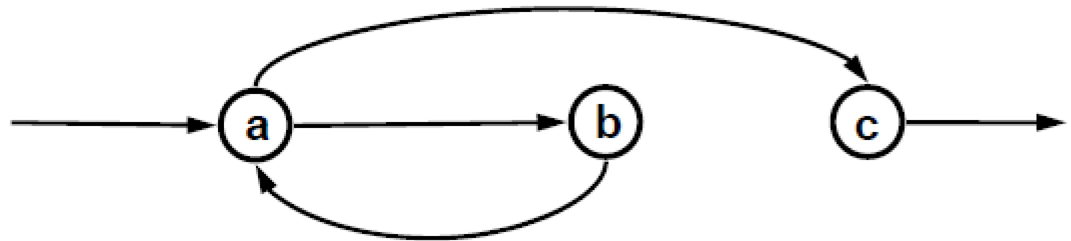
Zweigüberdeckung

- Forderung: Alle Kanten des Kontrollflussgraphen müssen durchlaufen werden.
- D.h.: Alle Verzweigungen müssen durchlaufen werden
- Stärkeres Kriterium als Anweisungsüberdeckung: 35% aller Fehler werden gefunden
- Für die Praxis wichtiges Kriterium



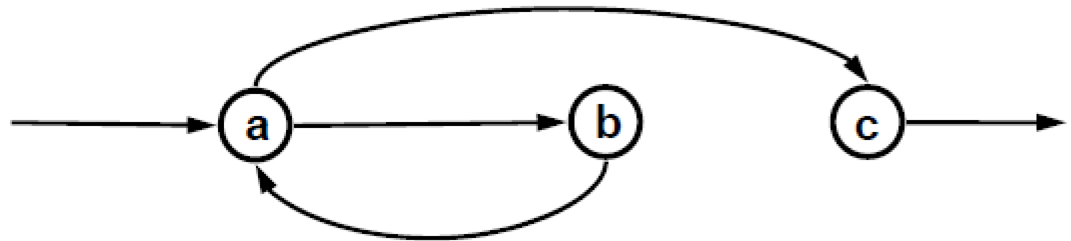
Pfadüberdeckung

- Forderung: Alle Pfade des KFG müssen durchlaufen werden
- Stärkstes Kriterium: Findet theoretisch alle Fehler
- In der Praxis nicht anwendbar



Boundary-Interior-Pfadtest

- Forderung: Mindestens einmalige Wiederholung von Zykluskörpern
- Innerhalb des Zykluskörpers wird Pfadüberdeckung gefordert
- Gutes Kriterium: 70% der Fehler werden aufgedeckt



Bedingungsüberdeckung

- Einfache Bedingungsüberdeckung:
  - Alle atomaren Bedingungen werden zu true und zu false ausgewertet
  - Schwächer als Zweigüberdeckung
- Mehrfache Bedingungsüberdeckung
  - Alle Werte-Kombinationen der atomaren Bedingungen treten auf
  - Für das Beispiel unmöglich
- Minimale mehrfache Bedingungsüberdeckung
  - Alle atomaren und zusammengesetzten Bedingungen werten zu true und zu false aus
  - Stärker als Zweigüberdeckung

Beispiel: if (ch == ‘a‘ || ch ==‘b‘) x = ‘y‘;

11.4
PROGRAMMVERIFIKATION

11.4 Programmverifikation

Versuch, die Korrektheit von Programmen mathematisch zu beweisen

- Assertions sind Zusicherungen, die an einer bestimmten Programmstelle gelten
  - beim Betreten einer Methode: Vorbedingung
  - vor Verlassen einer Methode: Nachbedingung
- Hoare-Kalkül: Systematisierung dieser Idee

  z.B. `assert (y<5); x=y*y; assert(x<25);`

- Hoare-Tripel: {precond} Prog {postcond}
    - Behauptung: „Falls precond vor Ausführung von Prog gilt, so gilt bei Terminierung hinterher postcond“
- Verifikationsaufgabe: Gegeben Hoare-Tripel (für komplexes Prog), zeige die Gültigkeit
- Programmentwicklung: Gegeben precond und postcond, konstruiere Prog
- Testentwicklung oftmals: Gegeben Prog, finde geeignete Vor- und Nachbedingungen

Hoare-Regeln

- Zuweisungsregel: {P[v/expr]} v=expr; {P}
  - Beispiel: {y*y>0} x=y*y; {x>0}
- Konsequenzregel: P'→P, {P}S{Q}, Q→Q' ➔ {P'}S{Q'}
  - Beispiel: y>0→ y*y>0 ➔ {y>0} x=y*y; {x>0}
- Sequenzregel: {P}S{Q}, {Q}T{R}➔{P}S;T{R}
  - Beispiel: {y>0} x=y*y; {x>0}, {x>0} z=x; {z>0} ➔
    {y>0} x=y*y; z=x; {z>0}
- Auswahlregel: {P&B}S{Q}, {P&!B}T{Q}
  ➔ {P}if (B) S; else T;{Q}
  - Beispiel: {true} if(y>0) z=y; else z=-y; {z>0}

Schleifen und Invarianten

- Iterationsregel:

  {I & B} S {I} → {I} while(B)S; {I & !B}

  - I ist die Schleifeninvariante, die durch S nicht verändert wird; Problem, I geeignet zu finden!
  - Beispiel: Sei INV= (j==Summe(i+1 .. n))

    {INV & i>0} j+=i--; {INV}

    {INV} while (i>0) j+=i--; {INV & i==0}

    j==0 & i==n → INV; INV & i==0 → j==Summe(1..n)

    also gilt:

    {j==0, i==n} while (i>0) j+=i--;{j==Summe(1..n)}

praktische Anwendung?

- Hoare-Kalkül galt lange Jahre als nicht anwendbar, zu kompliziert, mathematisch
- Unterstützt keine Zeiger, Parallelität
- Fortschritt bei Beweisunterstützungswerkzeugen
- in sicherheitskritischen Anwendungen keine Fehler tolerierbar, Standards fordern Verifikation
- bei FIRST Projekte, in denen Programme mit etlichen tausend LOCs verifiziert wurden
- Übung macht den Meister!
- Verifikation von Komponenten ersetzt den Test nicht

Danke!

FRAGEN?