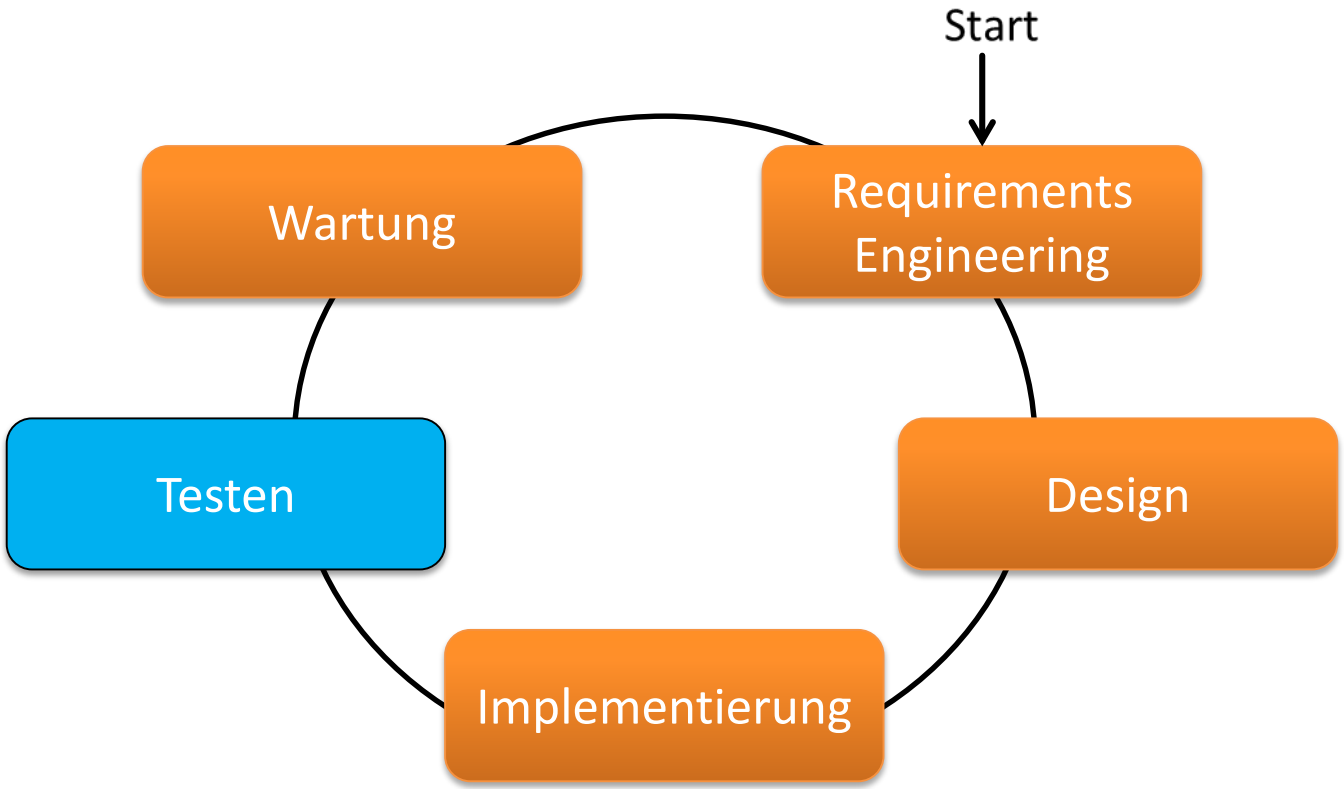
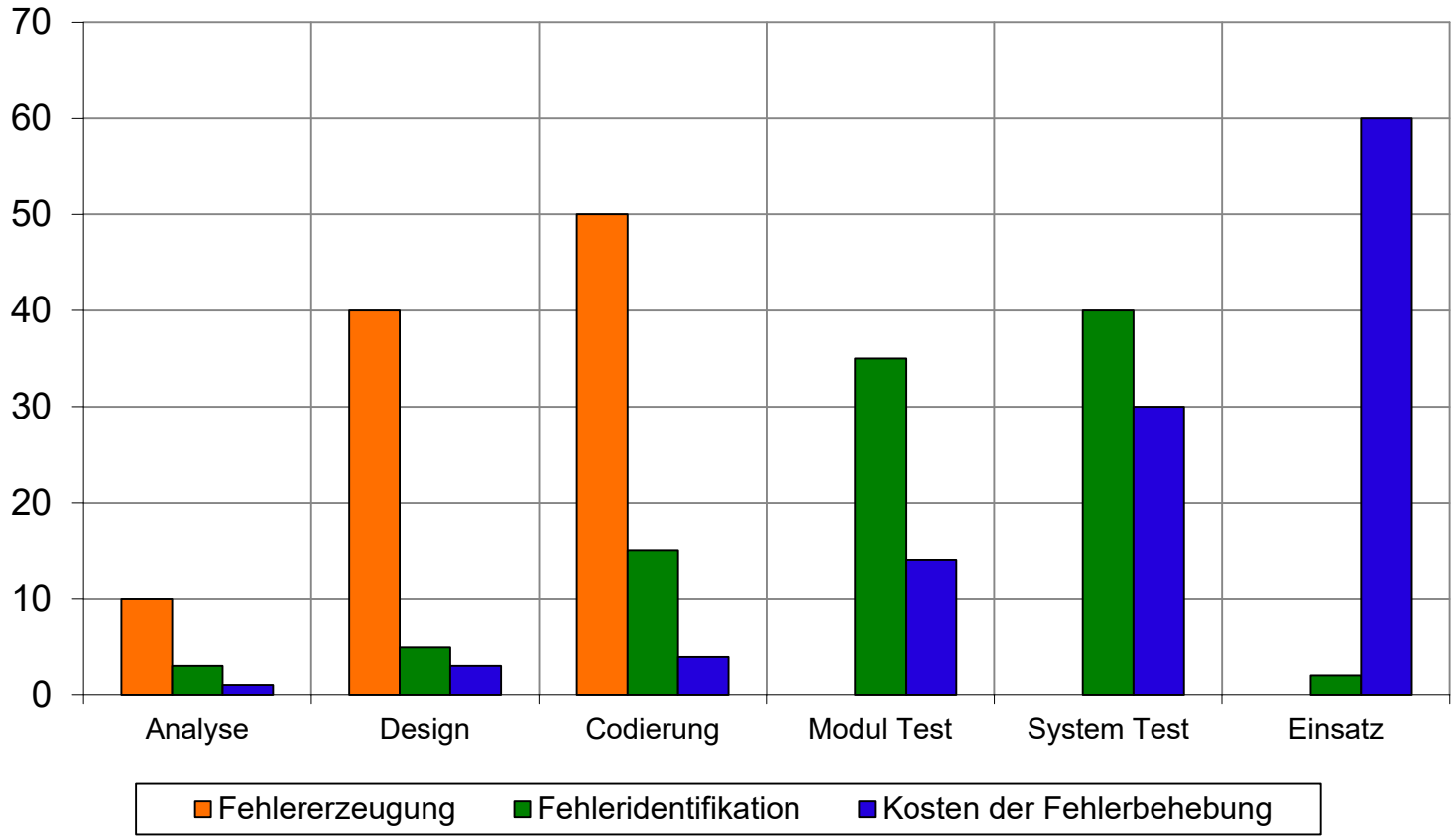
Testen

Authors of slides:
Norbert Siegmund
Janet Siegmund
Oscar Nierstrasz
Sven Apel

Einordnung


Start
Requirements Engineering
Design
Implementierung
Testen
Wartung

Kosten für die Behebung von Fehlern



Lernziele

- Notwendigkeit von Testen und Code Reviews verstehen
- Verschiedene Arten von Testen und Code Reviews kennen lernen

Ursprünge

- Margaret Hamilton

<- Quelltext für Apollo-Mission

- Ihre Tochter hat Software „getestet“
- Erst als Astronauten den selben „Fehler“ bei der Bedienung gemacht haben, wurde ein Fix eingebaut
- https://www.theguardian.com/technology/2019/jul/13/margaret-hamilton-computer-scientist-interview-software-apollo-missions-1969-moon-landing-nasa-women

Testen:
Einordnung und systematisches Vorgehen

Ziele I

- Google doc
- Sicherstellen, dass das Programm nicht abstürzt
- Regressionstest: keine neuen Fehler
- Sicherstellen, dass die Anforderungen erfüllt sind
- Sicherstellen, dass es keine Seiteneffekte gibt

Ziele II

"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence"

(*Edsger Dijkstra, The Humble Programmer, ACM Turing lecture, 1972*)

- Ziel von Testen:
  - Fehler finden
  - Vertrauen in Software herstellen
- Ein erfolgreicher Test findet Fehler

Herausforderungen

- Man muss annehmen, dass ein Programm fehlerhaft ist; nicht, dass es korrekt ist
  (Myers. A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections. 1978.)
- Entgegen jeder anderen Software-Entwicklungs-Aktivität (Fehler finden vs. Fehler vermeiden)
- Testen ist teuer
- Effektivität von Tests schwer zu messen
- Unvollständige, nicht-formalisierte und sich ändernde Anforderungen
- Integrationstest zwischen verschiedenen Produkten
- Steigende Anzahl von Versionen
- Patching nightmare

Von Microsoft Office EULA…

11. EXCLUSION OF INCIDENTAL, CONSEQUENTIAL AND CERTAIN OTHER DAMAGES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, **IN NO EVENT SHALL MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER** (INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR LOSS OF PROFITS OR CONFIDENTIAL OR OTHER INFORMATION, FOR BUSINESS INTERRUPTION, FOR PERSONAL INJURY, FOR LOSS OF PRIVACY, FOR FAILURE TO MEET ANY DUTY INCLUDING OF GOOD FAITH OR OF REASONABLE CARE, FOR NEGLIGENCE, AND FOR ANY OTHER PECUNIARY OR OTHER LOSS WHATSOEVER) **ARISING OUT OF OR IN ANY WAY RELATED TO THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT**, THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, OR OTHERWISE UNDER OR IN CONNECTION WITH ANY PROVISION OF THIS EULA, EVEN IN THE EVENT OF THE FAULT, TORT (INCLUDING NEGLIGENCE), STRICT LIABILITY, BREACH OF CONTRACT OR BREACH OF WARRANTY OF MICROSOFT OR ANY SUPPLIER, AND EVEN IF MICROSOFT OR ANY SUPPLIER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Von GPL

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. **THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU**. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING **WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM** AS PERMITTED ABOVE, **BE LIABLE TO YOU FOR DAMAGES**, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES **ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM** (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Mittlere Anzahl von Fehlern

- Industrie:
  - 30-85 Fehler per 1000 LOC;
  - 0,5-3 Fehler per 1000 LOC werden nicht erkannt vor Auslieferung.
  - 1% der Software-Fehler rufen 50% der Crashes hervor
- Snapshot von Mozilla's Bugzilla Bug Datenbank
  - Gesamte Historie von Mozilla; alle Produkte und Versionen
  - 60.866 offene Bug reports
  - 109.756 zusätzliche reports markiert als Duplikate
- Snapshot von Mozilla's Talkback Crash Reporter
  - Firefox 2.0.0.4 der letzten 10 Jahre
  - 101.812 eindeutige Nutzer
  - 183.066 Crash reports
  - 6.736.697 Stunden von user-driven "testing"

Arten von Fehlern

Fehlerklasse	***Beschreibung***
Transient	Tritt nur bei **bestimmten Eingaben** auf
Permanent	Tritt bei **allen Eingaben** auf
Recoverable	System erholt sich **ohne Intervention eines Nutzers**
Unrecoverable	**Nutzerintervention** ist **benötigt** zur Wiederherstellung des Systems
Non-corrupting	Fehler korrumpiert **nicht** die Daten
Corrupting	Fehler **korrumpiert** die Daten

Fehlervermeidung

Fehlervermeidung ist abhängig von:

1. Einer genauen **Systemspezifikation** (besser formal)
2. Software Design basierend auf **information hiding and encapsulation**
3. **Validierungsreviews** während des Entwicklungsprozesses
4. Eine organisatorische **Philosophie von Qualität**, welche den Softwareprozess prägt
5. Geplante Phasen von **System Testen und Verifikation**, um Fehler zu entdecken und die Zuverlässigkeit zu ermitteln

Häufige Software-Fehler I

Verschiedene Features von Programmiersprachen und Systemen sind häufige Quellen von Fehlern:

- ***Goto Statements*** und andere unstrukturierte Programmierkonstrukte machen Programme **schwer zu verstehen und zu modifizieren**.
- ***Zeiger*** sind gefährlich, durch **Aliasing** und dem Risiko den **Speicher zu korrumpieren**
- ***Parallelisierung*** ist gefährlich, da **zeitliche Unterschiede** einen Einfluss auf das Programmverhalten haben können, die **schwer vorhersagbar** sind.
- ***Interrupts*** erzwingen den Transfer der Kontrolle **unabhängig vom derzeitigen Kontext** und können daher zum Abbruch / Unterbrechung kritischer Operationen führen (lieber exceptions)

Häufige Software-Fehler II

- ***Gleitkommazahlen*** sind **inhärent ungenau** und können zu fälschlichen Vergleichen führen.
  - Ganzzahlen sind sicherer für exakte Vergleiche

```
public static void loop() {
  double d = 0.0;
  while (d != 1.0) {
    d += 0.1;}}
```

- ***Rekursion*** kann zu **verworrener Logik** führen und den Stack-Speicher überladen.
  - Verwende Rekursion in einer disziplinierten und kontrollierten Weise

```
private static int fibRec(int n) {
  stepsRec++;
  if (n == 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else {
    return fibRec(n-1) + fibRec(n-2);
  }
}
```

`fibRec(10)` needs 177 steps

```
private static int fibIt(int n) {
  int x = 0, y = 1, z = 1;
  for (int i = 0; i < n; i++) {
    stepsIt++;
    x = y;
    y = z;
    z = x + y;
  }
  return x;
}
```

$$\mathrm{Fib}(n) = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & \text{otherwise} \end{cases}$$

`fibIt(10)` needs 10 steps

Strategien des Testens

- Black-Box-Tests
  - Ohne auf den Code zu schauen
  - Beziehung zwischen Eingaben und Ausgaben
- White-Box-Tests/Glass-Box-Tests
  - Code anschauen und systematisch versuchen, Fehler zu erzeugen
  - Ausführungspfade untersuchen

Black-Box Testen

Black-Box Testing

- Jede Funktionalität des Systems überprüfen
- Alles kann nicht mit vertretbarem Aufwand getestet werden

- Siehe Dreamliner (Ausfall aller Turbinen):
  - Entdeckung des Zählerüberlaufs erfordert zeitliche Simulation
  - 5 Zähler müssen über 180 Tage emuliert werden

- Google doc

Äquivalenzklassen finden

- Nur Zahlen
  - Beispiel: System fragt nach Zahlen zwischen 100 und 999
  - Dann teste mit: <100; 100-999; >999
  - Tests haben auch nicht charakteristische/valide Werte!
- Nur Buchstaben
- Kombination aus Zahlen und Buchstaben
- Sonderzeichen? Umlaute?

- Äquivalenzklassen zu finden ist oft nicht trivial

Grenzwerte Analysieren

- Eingaben an Grenzen machen oft Probleme
- Bsp: `ArrayIndexOutOfBoundsException()`

Erfahrung und Heuristik

- Bisherige Erfahrungen und Heuristiken nutzen
  - Sonderzeichen haben schon immer Probleme gemacht -> Sonderzeichen einschließen
  - Umlaute machen Probleme in anderen Sprachen -> Umlaute einschließen

Einfache Daten

- 3,14159265 ist schwerer manuell zu überprüfen als 2
- Z.B., wenn Code etwas verdoppeln soll

- Daten sollten daher nachvollziehbar ausgewählt werden (z.B. im Kopf ausrechenbar sein)

Systematisches Vorgehen

- Funktionalitäten aus Anforderungen ableiten und Eingabe- und zugehörige Ausgabedaten bestimmen
- Äquivalenzklassen von Eingaben testen
- Daten an Intervallgrenzen testen
- Inkorrekte Eingaben testen
- Jede definierte Fehlermeldung erzeugen
- Kombination von Funktionalitäten testen (fett+kursiv+Schriftgröße)
- Seltene Fälle testen

Wann Black-Box Testen?

- Geeignet zum Finden von:
  - Inkorrekter oder fehlender Funktionalität (aus Spezifikation)
  - Schnittstellenfehler
  - Fehler in Datenstrukturen oder externen Zugriffen
  - Problemen von nicht-funktionalen Eigenschaften
  - Fehlern beim Ablauf von Prozessen
- Grenzen:
  - Spezifikation ist meist abstrakt und spiegelt nicht Implementierung wieder
  - Ein externer Zustand kann mehreren internen Zuständen entsprechen (wie Testen?)
  - Nicht alle Elemente einer Äquivalenzklasse werden auch im Code gleich behandelt (fehlende Äquivalenzklassen)

Weitere Limitierungen von Black-Box Tests

Können Sie weitere Gründe benennen warum Black-Box Tests allein unzureichend sind?

- Spezifikationen und Sonderfälle können vergessen / übersehen werden
- Fehler, die unabhängig von der Eingabe sind, können evtl. nicht entdeckt werden (z.B. bei parallel laufenden Programmen)
- Ausnahmefälle (wie z.B. Hardwareausfall) und deren Fehlerbehandlungen können oft nicht ausreichend getestet werden

White-Box / Glass-Box Testen

Whitebox-Testing

```
public int gCD(int number1, int number2) {
  int temp;
  do {
    if (number1 < number2) {
      temp = number1;
      number1 = number2;
      number2 = temp;
    }
    temp = number1 % number2;
    if (temp != 0) {
      number1 = number2;
      number2 = temp;
    }
  } while (temp != 0);
  return number2;
}
```

Link zum Googledoc vom 13.12.

Aufgabe

```
   public int gCD(int number1, int number2) {
1.   int temp;
2.   do {
3.     if (number1 < number2) {
4.       temp = number1;
5.       number1 = number2;
6.       number2 = temp;
       }
7.     temp = number1 % number2;
8.     if (temp != 0) {
9.       number1 = number2;
10.      number2 = temp;
       }
11.  } while (temp != 0);
12.  return number2;
   }
```

Bsp: number1 = 6; number2 = 3
Zeile 7: temp = 0

Anweisungs-Überdeckung: 7 / 12

Aufgabe

```
   public int gCD(int number1, int number2) {
1.   int temp;
2.   do {
3.     if (number1 < number2) {
4.       temp = number1;
5.       number1 = number2;
6.       number2 = temp;
       }
7.     temp = number1 % number2;
8.     if (temp != 0) {
9.       number1 = number2;
10.      number2 = temp;
       }
11.  } while (temp != 0);
12.  return number2;
   }
```

Bsp: number1 = 3; number2 = 6

Anweisungs-Überdeckung: 10 / 12

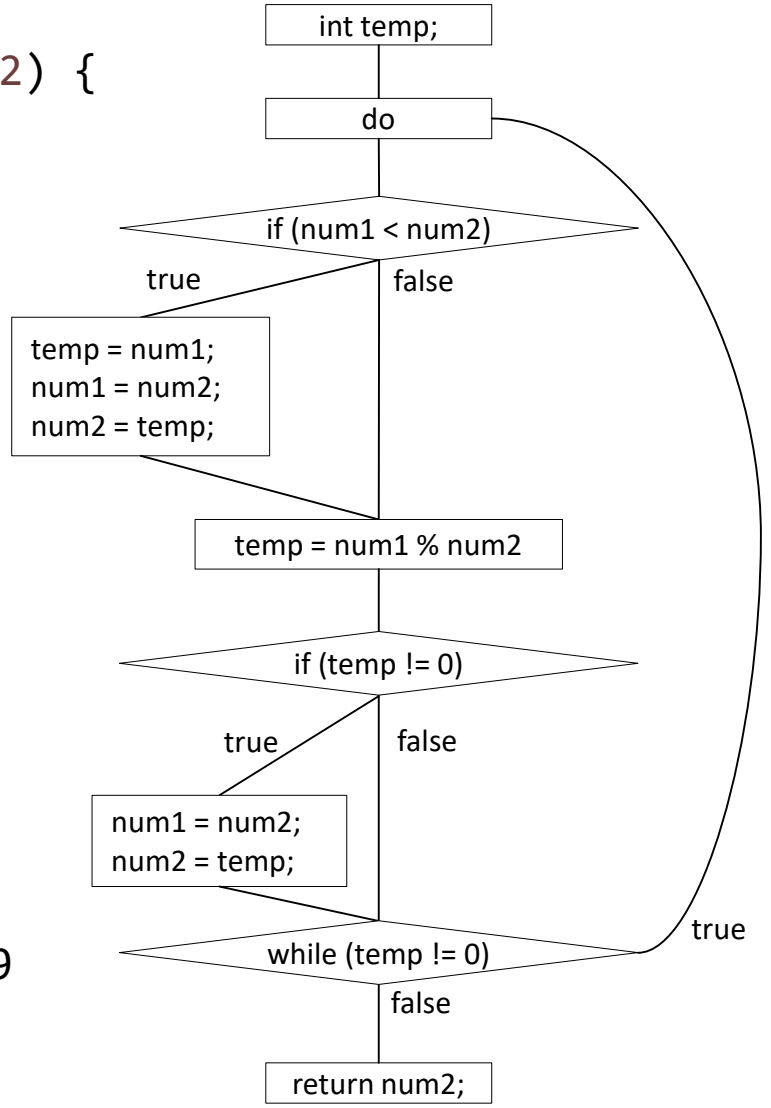
Bsp: number1 = 6; number2 = 9

Anweisungs-Überdeckung: 12 / 12

Zweig-Überdeckung

```
public int gCD(int number1, int number2) {
  int temp;
  do {
    if (number1 < number2) {
      temp = number1;
      number1 = number2;
      number2 = temp;
    }
    temp = number1 % number2;
    if (temp != 0) {
      number1 = number2;
      number2 = temp;
    }
  } while (temp != 0);
  return number2;
}
```

Bsp: number1 = 6; number2 = 9

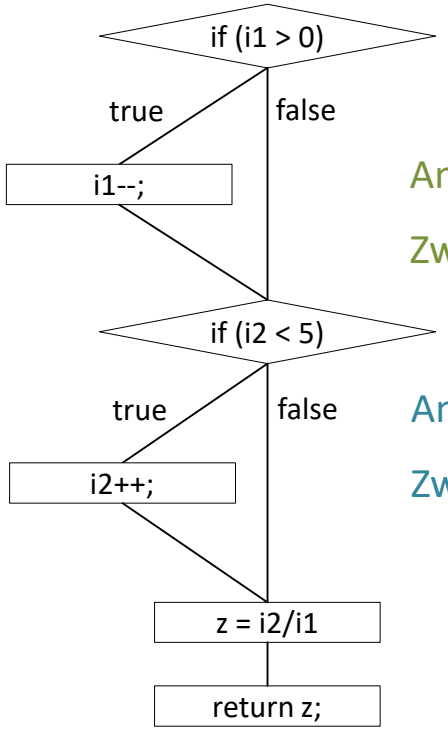


Anweisungs- vs. Zweigüberdeckung

```
1. public static int calculate(int input1, int input2){
2.   if (input1 > 0)
3.     input1--;
4.   if (input2 < 5)
5.     input2++;
6.   int z = input2/input1;
7.   return z;
8. }
```

Bsp: input1 = 2; input2 = 3

Bsp: input1 = 0; input2 = 5



Anweisungsüberdeckung: 100%

Zweigüberdeckung: 71%

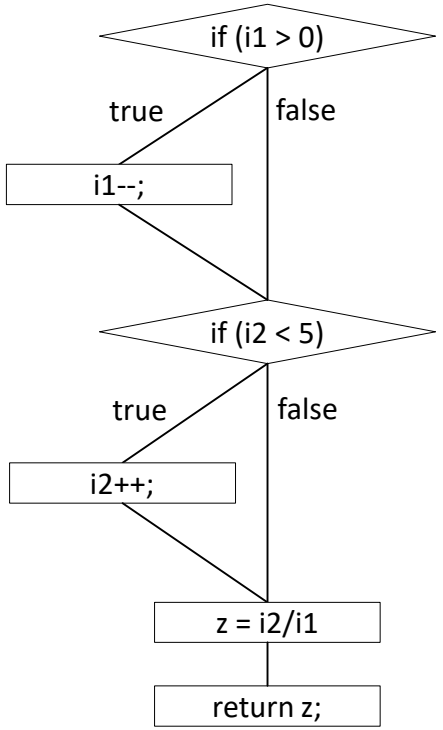
Anweisungsüberdeckung: 67%

Zweigüberdeckung: 43%

Link zur Google-Zeichnung im Google-Doc

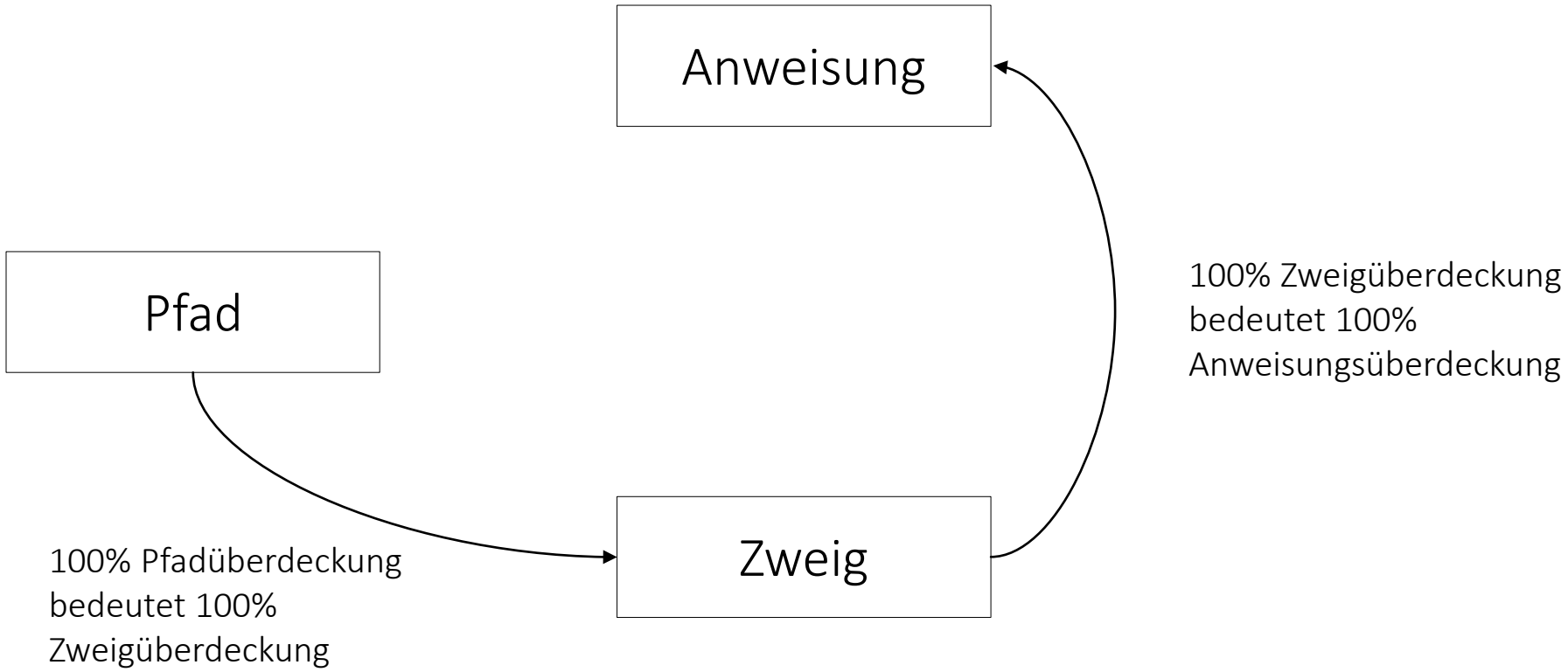
Pfadüberdeckung

```
1. public static int calculate(int input1, int input2){
2.   if (input1 > 0)
3.     input1--;
4.   if (input2 < 5)
5.     input2++;
6.   int z = input2/input1;
7.   return z;
8. }
```



4 Ausführungspfade
Testfälle zur 100%-Abdeckung:
input1 = 2, input2 = 2
input1 = -1, input2 = 5
input1 = 2, input2 = 5
input1 = -1, input2 = 2

Beziehung der Überdeckungskriterien

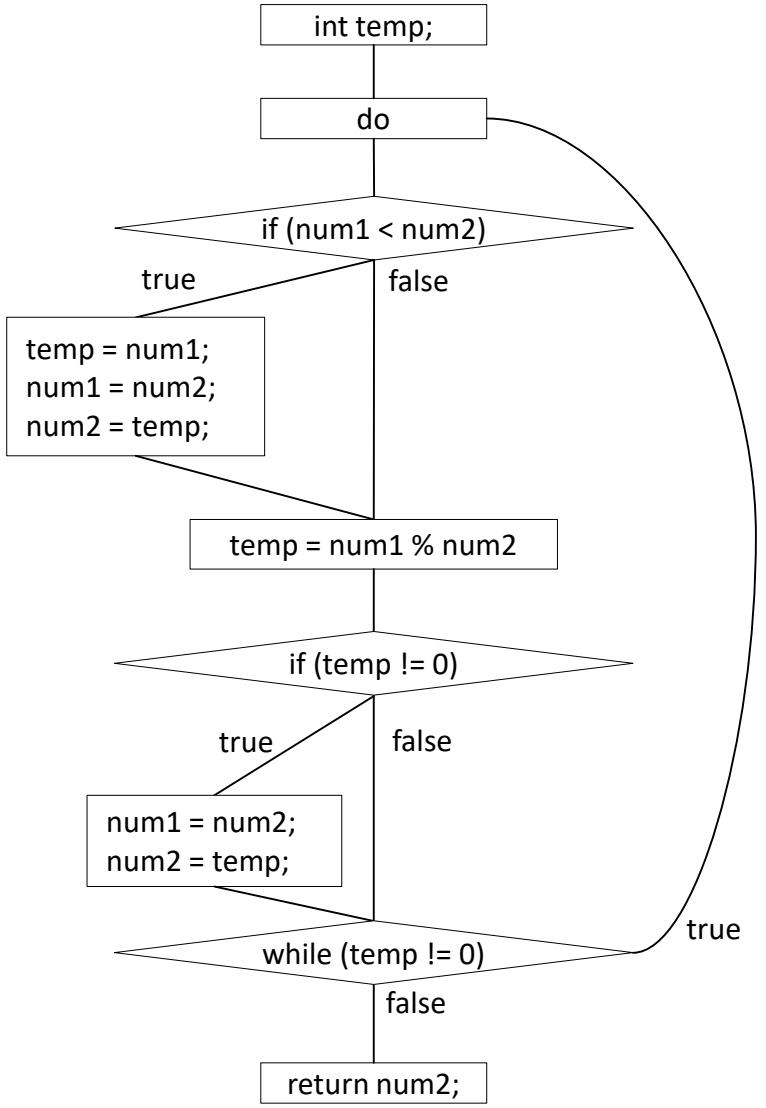


Pfadüberdeckung

```
public int gCD(int number1, int number2) {
  int temp;
  do {
    if (number1 < number2) {
      temp = number1;
      number1 = number2;
      number2 = temp;
    }
    temp = number1 % number2;
    if (temp != 0) {
      number1 = number2;
      number2 = temp;
    }
  } while (temp != 0);
  return number2;
}
```

Wie viele Ausführungspfade?

Zu viele, um sie alle zu testen



Praxis

- 100% Zweigüberdeckung als Minimalziel

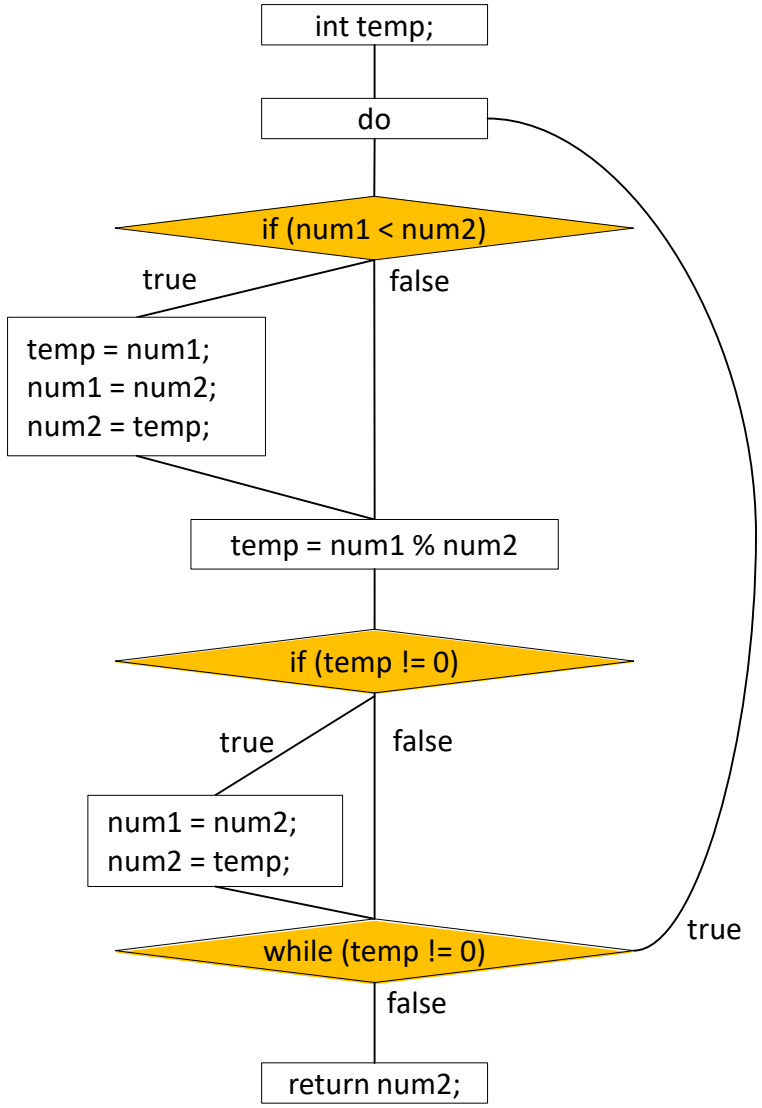
Bedingungsüberdeckung

- Testen der Belegungen von Bedingungen

Bsp: num1 = 6, num2 = 3
Bedingungsüberdeckung: 50% (3/6)

Bsp: num1 = 3, num2 = 6
Bedingungsüberdeckung: 50% (3/6)

Bsp: num1 = 6, num2 = 9
Bedingungsüberdeckung: 100%



Herausforderungen

- Oft durch Entwickler selbst durchgeführt
- Welcher Entwickler geht schon gern davon aus, dass sein Programm Fehler hat
- Man testet oft das Verhalten, was man sowieso bereits im Kopf / programmiert hat
  - Daher: Andere Personen wichtig zum Testen, um alternative Herangehensweise / Benutzungen / etc vom System zu testen

Pfadüberdeckungsverfahren

- Wähle eine Testmenge so, dass alle Pfade vom Eingangs- bis zum Ausgangsknoten durchlaufen werden
  - Probleme bei Schleifen (insb. while)
  - Anz. der Wiederholungen in Schleifen wird meist eingeschränkt
- Motivation:
  - Logische Fehler und inkorrekte Annahmen sind umgekehrt proportional zur Wahrscheinlichkeit der Ausführung des Pfades
  - Entwickler:innen haben häufig fälschliche Annahme, dass ein bestimmter Pfad nicht ausgeführt wird
  - Tippfehler sind zufällig; somit wahrscheinlicher in ungetesteten Pfaden

Probleme der Pfadüberdeckung

- Sehr schnell zu viele Pfade
  - Sollte nur bei kritischen Modulen verwendet werden
  - Spezialfälle für Schleifen
    - Kein Durchlauf
    - 1 Durchlauf
    - 2 Durchläufe
    - M Durchläufe bei m < n : n = max. Anzahl Durchläufe
    - n-1, n, n+1 Durchläufe
- Zyklomatische Komplexität als Maß der Pfadüberdeckung

Grenzen und Zusammenfasung

- 100%-Testabdeckung praktisch nicht möglich, besonders bei Pfadüberdeckung und Fallunterscheidungen
- White-Box Tests ergänzen Black-Box Tests
- Viele Tools, die Testabdeckung messen und visualisieren (http://c2.com/cgi/wiki?CodeCoverageTools)
- Trotz systematischem Vorgehen kann man nie sicher sein, dass der Quelltext fehlerfrei ist
  - Dijkstra hat nach 40 Jahren immer noch Recht

Konkrete Testverfahren

Testverfahren

1. Unit-Tests
2. Test-Driven Development
3. Integrations-Tests
4. System-Tests
5. Acceptance-Tests

1. Unit Tests

- Ziel: Individuelle Komponenten / Module werden getestet, um deren korrekte Funktionsweise sicherzustellen

- Typischerweise automatisiert
- Oft durch Entwickler:in spezifiziert
- Fokus auf eine Funktion/Methode/Modul
- Stubs/Mock-Objekte, wenn dabei andere Module aufgerufen werden
  - Stubs/Mocks emulieren anderen Objekte/Methoden im Programm, welche notwendig sind, um das eigentliche Modul zu testen

JUnit – Unit Testing Framework

- JUnit Test ist eine Methode in einer Klasse, die nur für das Testen verwendet wird
  - Annotationen markieren Methoden, die einen Test spezifizieren (@org.junit.Test)
- Innerhalb der Methode wird eine Methode des Frameworks verwendet, welche das erwartete Ergebnis gegen das des ausgeführten Codes vergleicht

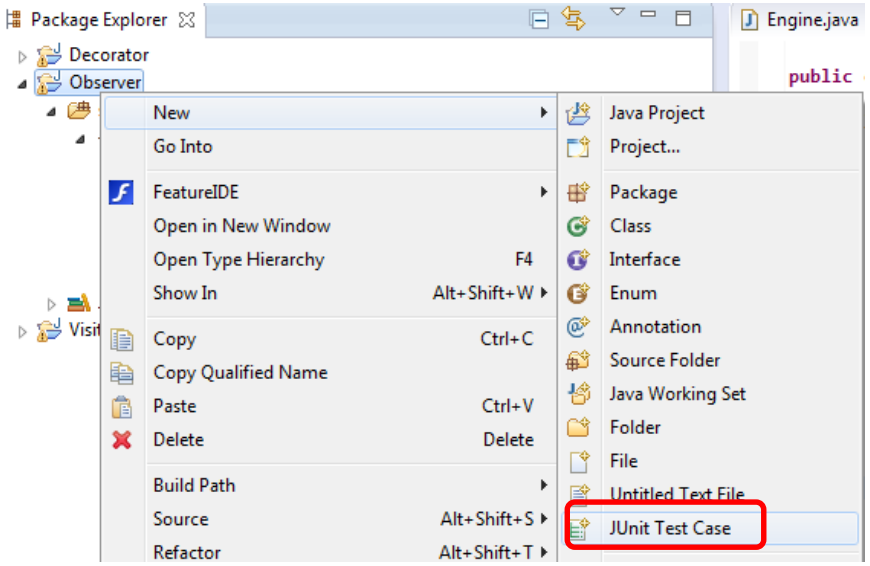
```
@Test public void multiplicationOfZeroIntegersShouldReturnZero() {
  // MyClass is tested
  MyClass tester = new MyClass();
  // Tests
  assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
  assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
  assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
}
```

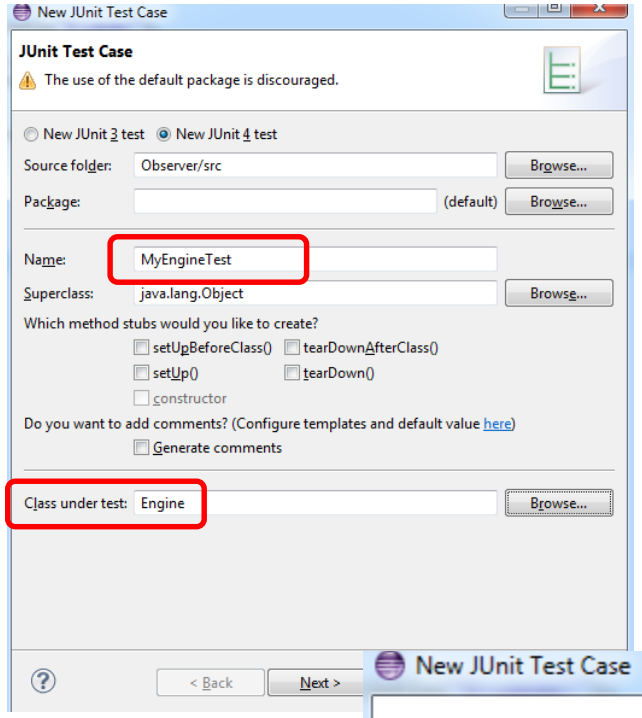
JUnit – Methoden und Annotationen

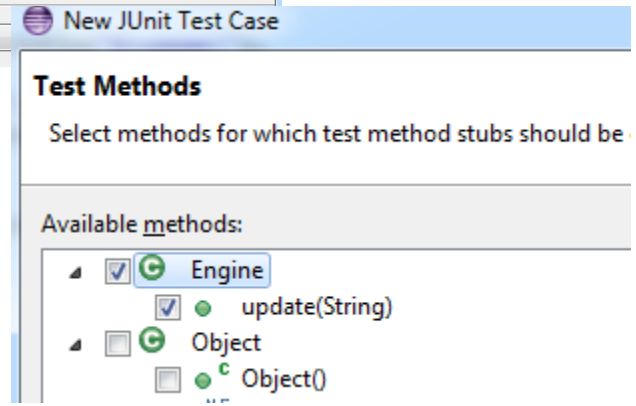
Statement	Description
fail(message)	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The message parameter is optional.
assertTrue([message,] boolean condition)	Checks that the boolean condition is true.
assertFalse([message,] boolean condition)	Checks that the boolean condition is false.
assertEquals([message,] expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([message,] expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message,] object)	Checks that the object is null.
assertNotNull([message,] object)	Checks that the object is not null.
assertSame([message,] expected, actual)	Checks that both variables refer to the same object.
assertNotSame([message,] expected, actual)	Checks that both variables refer to different objects.

Annotation	Description
@Test public void method()	The `@Test` annotation identifies a method as a test method.
@Test (expected = Exception.class)	Fails if the method does not throw the named exception.
@Test(timeout=100)	Fails if the method takes longer than 100 milliseconds.
@Before public void method()	This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
@After public void method()	This method is executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
@BeforeClass public static void method()	This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as `static` to work with JUnit.
@AfterClass public static void method()	This method is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as `static` to work with JUnit.
@Ignore or @Ignore("Why disabled")	Ignores the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled.

JUnit in Eclipse

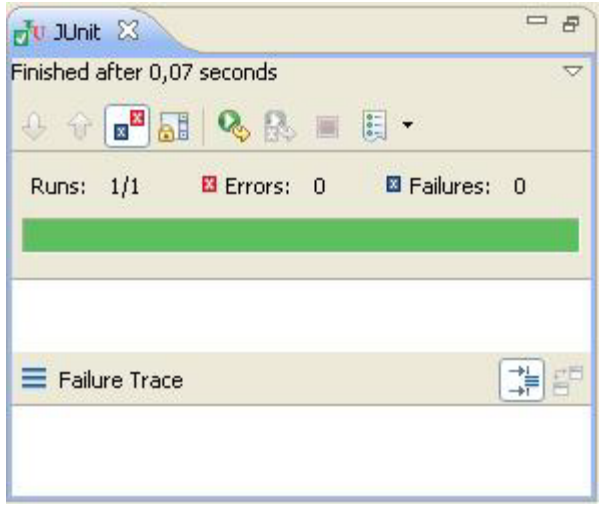


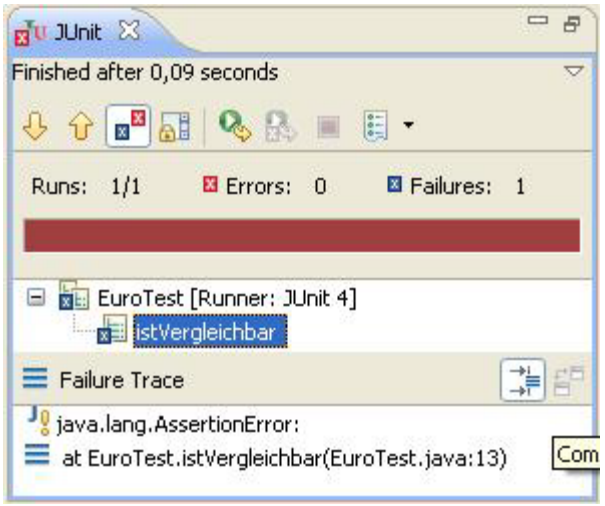




JUnit – Beispiel

```java
import static org.junit.Assert.*;


public class MyEngineTest {

    @Test
    public void testUpdate() {
        Engine eng = new Engine();
        assertTrue("Test ob Engine Geräusche macht beim fahren", "Brummmm!".equals(eng.update("fährt")));
        assertTrue("Test ob Engine Geräusche macht beim fahren", "zzz".equals(eng.update("parkt")));
    }

}
```



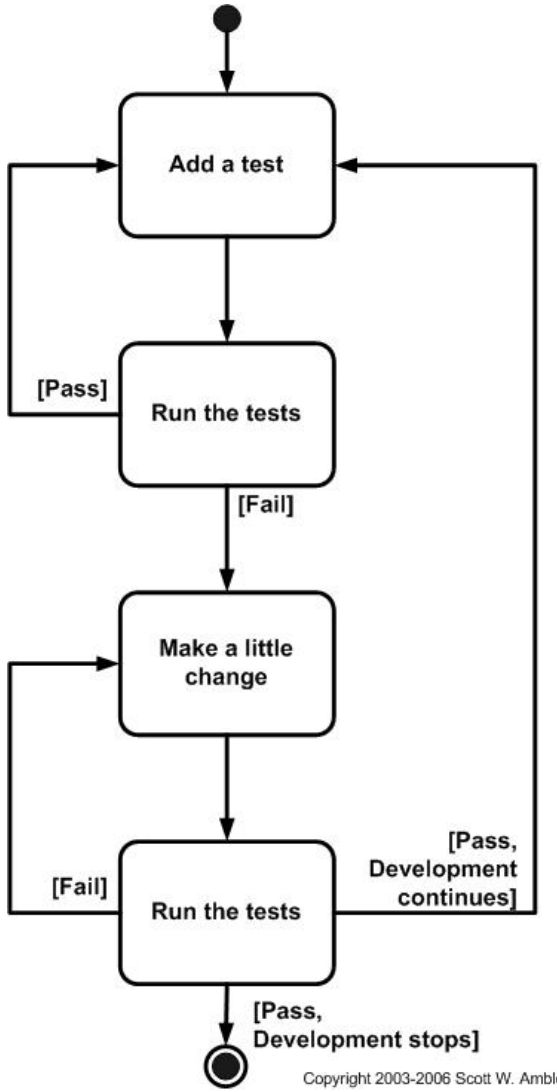


Frameworks zum Testen

- https://site.mockito.org

2. Test-Driven Development

- Evolutionärer Entwicklungsansatz, welcher die folgenden Methoden kombiniert:
  - Test-first design: zuerst Test schreiben bevor der Code geschrieben wird, welcher getestet werden soll
  - Refactoring



Test-Driven Development II

- Umgekehrter Entwicklungsansatz
  - Ist das existierende Design das best-möglichste Design, um ein Feature zu implementieren?
    - Falls ja, setze mit nächstem Feature fort
    - Falls nein, refaktorisiere es lokal, so dass das neue Feature so einfach wie möglich hinzugefügt werden kann
  - Ergebnis: kontinuierliche Verbesserung der Qualität des Designs

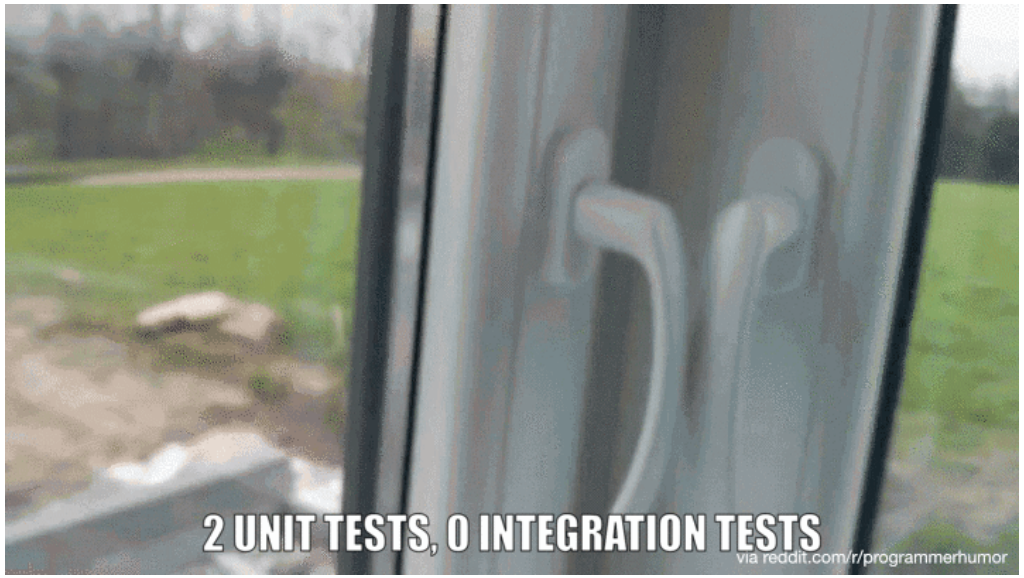
Prinzipien von TDD

- Schreibe Test-Code vor dem funktionellen Code
- Sehr kleine Schritte --- ein Test und eine kleine Einheit korrespondierenden Codes zur gleichen Zeit
- Programmierer:innen verweigern das Hinzufügen auch nur einer einzelnen Zeile Code solange dafür kein Test existiert
- Sobald ein Test vorhanden ist, setzen Programmierer:innen alles daran, dass die Test Suite erfolgreich durchläuft

> “If it's worth building, it's worth testing.
>
> If it's not worth testing, why are you wasting your time working on it?”
>
> –Scott W. Ambler

Unit Test vs. Integration Test

- Module werden kombiniert und als Gruppe getestet
- Nach Unit-Tests, vor System-Tests



3. Integrations-Tests

- Zweck: Erfüllen Module im Zusammenspiel funktionale und nicht-funktionale Anforderungen?
- Basiert auf Black-Box-Tests
- Top-down oder Bottom-up

Integrations-Tests: Top-Down

1. Kontrollflussmodul führt Test aus, alle untergeordneten Module werden durch Stubs ausgetauscht
2. Untegordnete Module werden durch eigentliche Module Schritt für Schritt getauscht
3. Nach jedem Tausch wird getestet

IT: Bottom-Up

1. Low-level-Module, die bestimmte Funktion ausführen, werden zu Cluster kombiniert
2. Modul, das Tests koordiniert
3. Cluster wird getestet
4. Getestete Cluster werden kombiniert und wieder getestet, bis höchste Ebene erreicht ist

4. System-Tests

- Black-Box-Test des kompletten Systems
- Konzentration auf:
  - Fehler, die aus Interaktionen zwischen Sub-Systemen herkommen
  - Validierung, dass das gesamte System funktioniert und nicht-funktionale Anforderungen erfüllt
- Orientierung oft an use cases
- Meistens durch separates Test-Team
- Viele verschiedene Arten von System-Tests
  - GUI, Usability, Performance, Barrierefreiheit, Stresstests,…

Regressions-Test

- Idee: Nach **jeder** Änderungen werden **alle** Testfälle wieder ausgeführt
- **Sicherstellung**, dass alles, was **vor** der **Änderung** funktioniert hat, auch **nach** der Änderung weiterhin funktioniert
- Tests müssen **deterministisch** und **wiederholbar** sein
- Tests sollten gesamte Funktionalität umfassen
  - Jedes Interface
  - Alle Grenzsituationen /-fälle
  - Jedes Feature
  - Jede Zeile Code
  - Alles was irgendwie falsch gehen kann

Nightly/Daily Builds

- Release eines großen Projekts wird zu fest definiertem Zeitpunkt erstellt
- Zeitpunkt: Wenn keine Änderungen am Code zu erwarten sind
- Gefundener Fehler ist großes Problem:
  - Verantwortliche:r Entwickler:in muss ggfs. nachts das Problem beheben
- Nach einem Build oft Regressions-Tests

“Treat the daily build as the heartbeat of the project. If there is no heartbeat, the project is dead.” -Jim McCarthy (http://www.mccarthyshow.com/)

5. Acceptance-Tests

- Erfüllt das System alle spezifierten Anforderungen?
- Funktionstest, die Kund:in ausführt, um Qualität zu bewerten
- Echte statt simulierte Daten
- Alpha-Tests:
  - White-Box-Tests durch Entwickler:inen
  - Danach Black-Box-Test durch andere Teams
- Beta-Tests:
  - Endnutzer:innen testen System

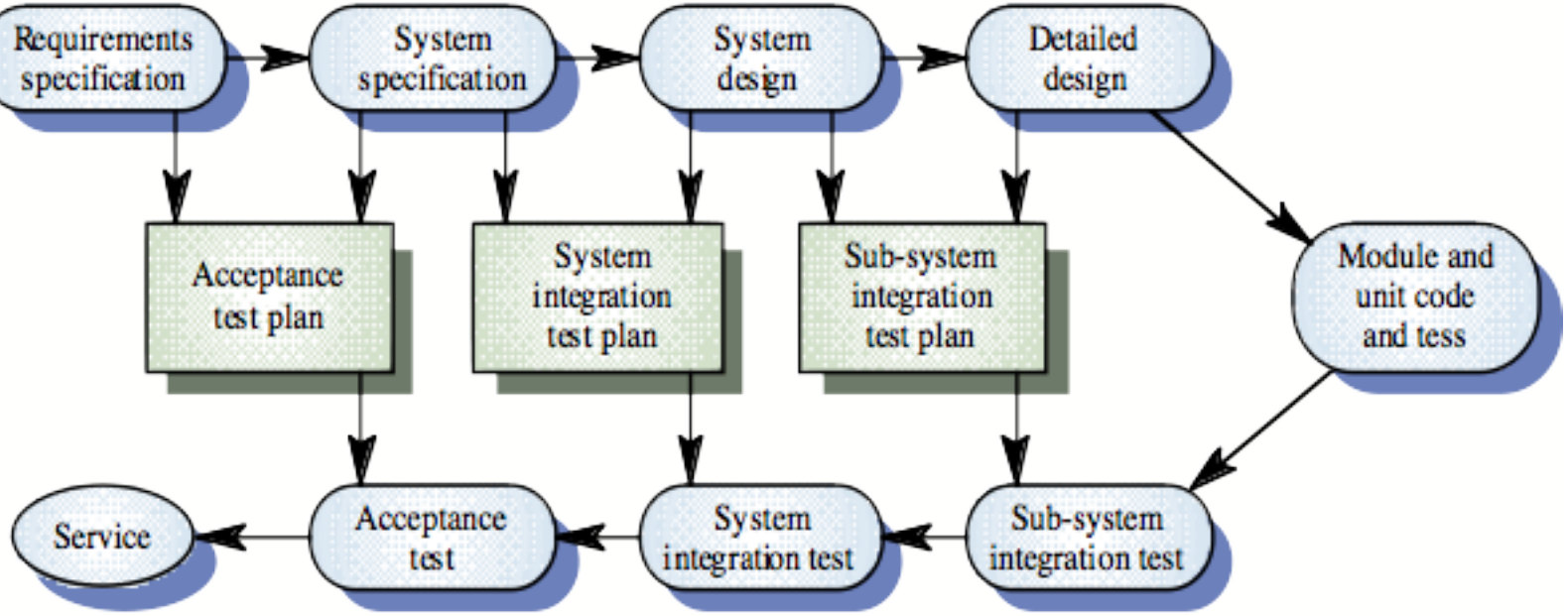
Wann findet man die meisten Fehler?

Testing technique	Rate
Unit test	30%
Integration test	35%
System test	40%
Beta test	bis zu 75%

- Rate:
  - Anzahl gefundener Fehler beim Anwenden einer Technik
  - Gefundene Fehler pro Strategie überlappen sich teilweise

Wie spielt alles zusammen?

- Testplan muss erstellt werden, **sobald die Anforderungen formuliert** sind, und **ständig verfeinert** werden
- Plan sollte **regulär** überarbeitet und Tests **wiederholt** und **erweitert** werden



Design für Testen

- Stelle sicher, dass Komponenten in Isolation getestet werden können
  - Minimiere Abhänigkeiten zu anderen Komponenten
  - Biete Konstruktoren an, um Objekte für das Testen zu erstellen

- Design Techniken existieren für verbesserte Testbarkeit
  - Benutze Interfaces, um Mock Objekte oder Stubs zu nutzen

Nochmal: THE limitation of testing

- *"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."*
  - E.W. Dijkstra

- Keine Fehler bei Tests kann bedeuten:
  - Es gibt keine Fehler
  - Testfälle sind unvollständig

- Lösung: Verifikation von Software (nicht Teil der Vorlesung)

Software Verifikation und Modell-basiertes Testen

Was bedeutet fehlerfrei?

- Programm kann kompiliert werden
- Main-Methode wirft keine Exception
- Generell keine NullpointerException, ClassCastException
- Keine Zugriffe auf nicht-initialisierten Speicher

- Sagt alles noch nichts über „richtige“ Ergebnisse aus

  → Spezifikation

Verifikation und Validation

Verifikation:

- Bauen wir das **Produkt richtig**?
  - D.h., entspricht es der Spezifikation?

Validation:

- Bauen wir das **richtige Produkt**?
  - D.h., entspricht es den Nutzererwartungen?

Statische Verifikation

Programminspektionen:

- Kleine Teams prüfen systematisch den Code
- Checklisten sind oft erforderlich
  - z.B., “Sind alle Invarianten, Vor- und Nachbedingungen überprüft?” ...

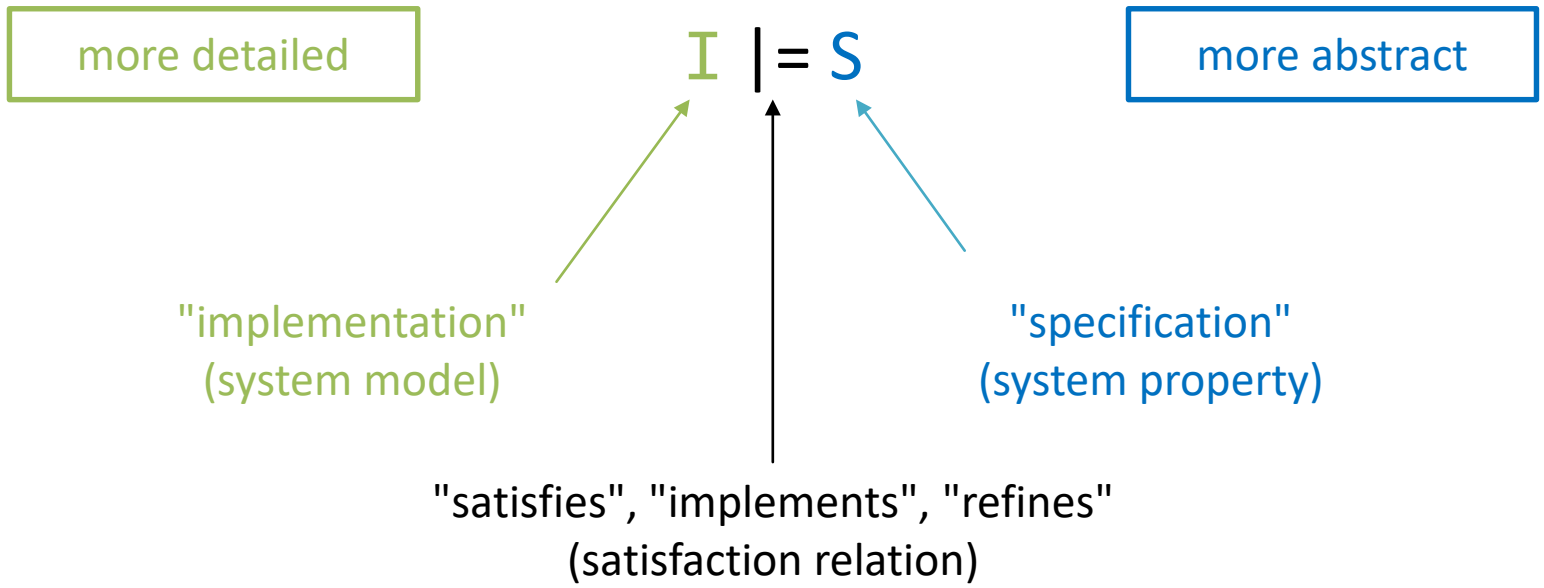
Statische Programmanalyse:

- Komplementiert Compiler-Checks für gewöhnliche Fehler
  - z.B., Variable benutzt vor Initialisierung

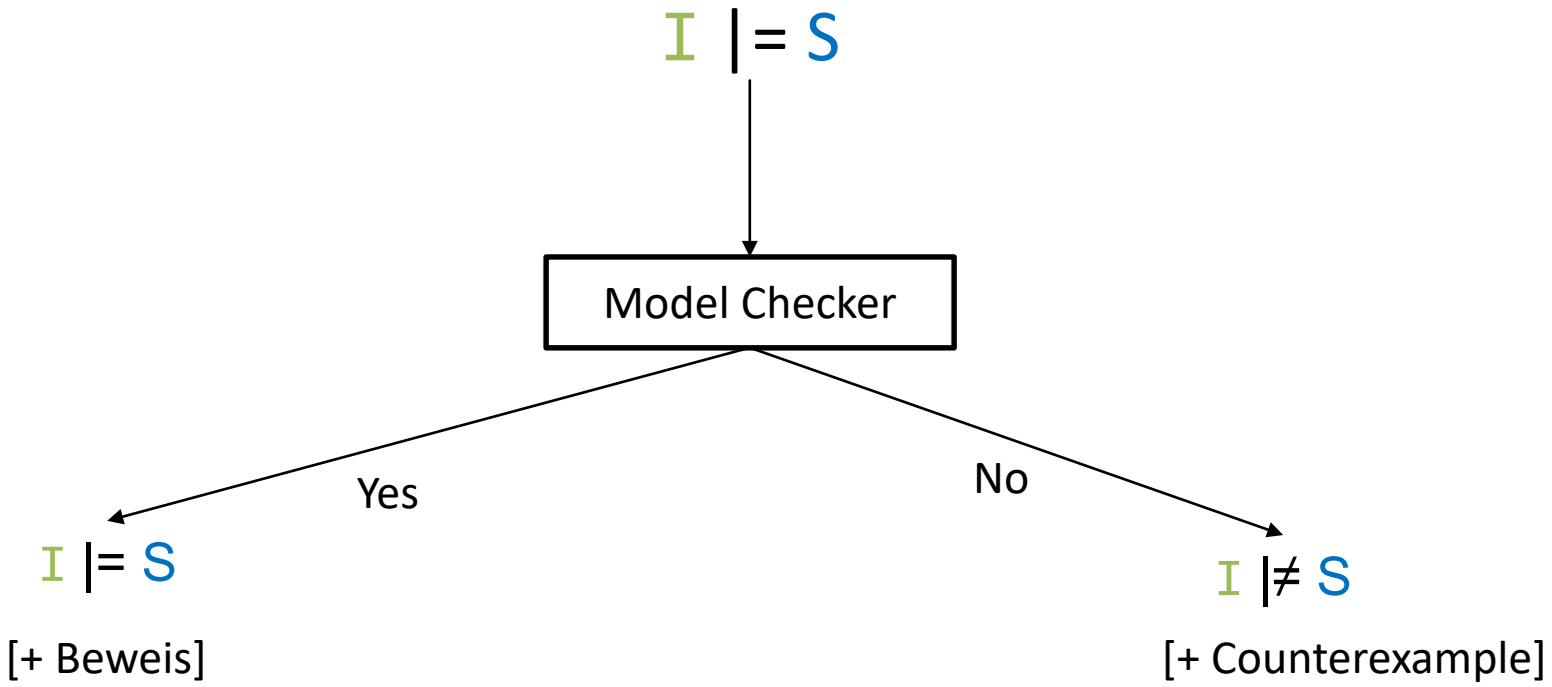
Mathematisch-basierte Verifikation:

- Verwendung von mathematischen Herleitung zur Demonstration, dass das Programm die Spezifikationen erfüllt
  - z.B., dass Invarianten nicht verletzt werden, Schleifen terminieren, etc
  - z.B., Model-checking Tools

Model Checking

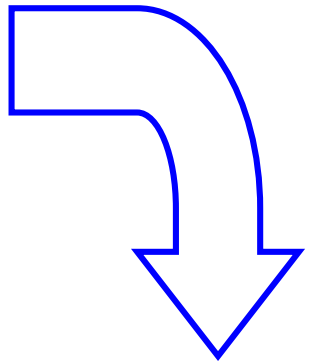


Model Checking



Beispiel

```
import java.util.Random;
public class Rand {
  public static void main (String[] args) {
    Random random = new Random(42);  // (1)

    int a = random.nextInt(2);       // (2)
    System.out.println("a=" + a);

    //... lots of code here

    int b = random.nextInt(3);       // (3)
    System.out.println("b=" + b);

    int c = a/(b+a -2);              // (4)
    System.out.println("c=" + c);
  }
}
```

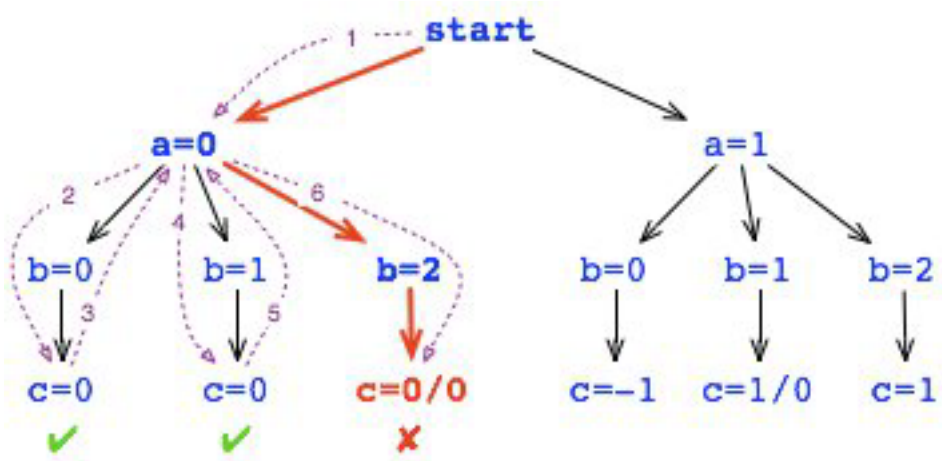


```
> java Rand
a=1
b=0
c=-1
>
```

Model Checking

```
import java.util.Random;
public class Rand {
  public static void main (String[] args) {
    Random random = new Random(42);  // (1)

    int a = random.nextInt(2);                  // (2)
    System.out.println("a=" + a);

    //... lots of code here

    int b = random.nextInt(3);                  // (3)
    System.out.println("b=" + b);

    int c = a/(b+a -2);                         // (4)
    System.out.println("c=" + c);
  }
}
```

Idee: Prüfe alle möglichen Werte von Ausdrücken



Informelle Spezifikation

Beispiel: JavaDoc

```
/**
 * The method takes a given array of integers, sorts and returns the array.
 *
 * @param  values a potentially unsorted array
 * @return  array sorted in ascending order
 */
public int[] sort(int[] values) {
  for (int j = values.length; j > 1; j--) {
    for (int i = 0; i < j - 1; i++) {
      if (values[i] > values[i + 1]) {
        int temp = values[i];
        values[i] = values[i + 1];
        values[i + 1] = temp;
      }
    }
  }
  return values;
}
```

Problem 1: Nur durch manuelles Testen überprüfbar

Problem 2: Mehrdeutigkeiten

Formale Spezifikation

Beispiel: Java Modeling Language (JML)

```
/*@
 @ requires values != null;
 @ ensures (\forall int i; 0 < i && i < values.length; \result[i-1] <= \result[i]);
 @*/
public int[] sort(int[] values) {
  for (int j = values.length; j > 1; j--) {
    for (int i = 0; i < j - 1; i++) {
      if (values[i] > values[i + 1]) {
        int temp = values[i];
        values[i] = values[i + 1];
        values[i + 1] = temp;
      }
    }
  }
  return values;
}
```

- Runtime Assertions
- Deduktive Verifikation
- Statische Analysen

Runtime Assertions

Vorteile

- Präzise Fehlerlokalisierung (Blame assignment)
- Keine False-Positives
- Automatisierbar
- Orthogonal zum Testen

Nachteile

- Findet nicht alle Fehler
- Ausbremsen des Systems

```
public int[] sort(int[] values) {

  assert values != null;

  for (int j = values.length; j > 1; j--) {
    for (int i = 0; i < j - 1; i++) {
      if (values[i] > values[i + 1]) {
        int temp = values[i];
        values[i] = values[i + 1];
        values[i + 1] = temp;
      }
    }
  }

  for (int i = 1; i < values.length; i++)
    assert values[i-1] <= values[i];

  return values;
}
```

Deduktive Verifikation

Vorteile

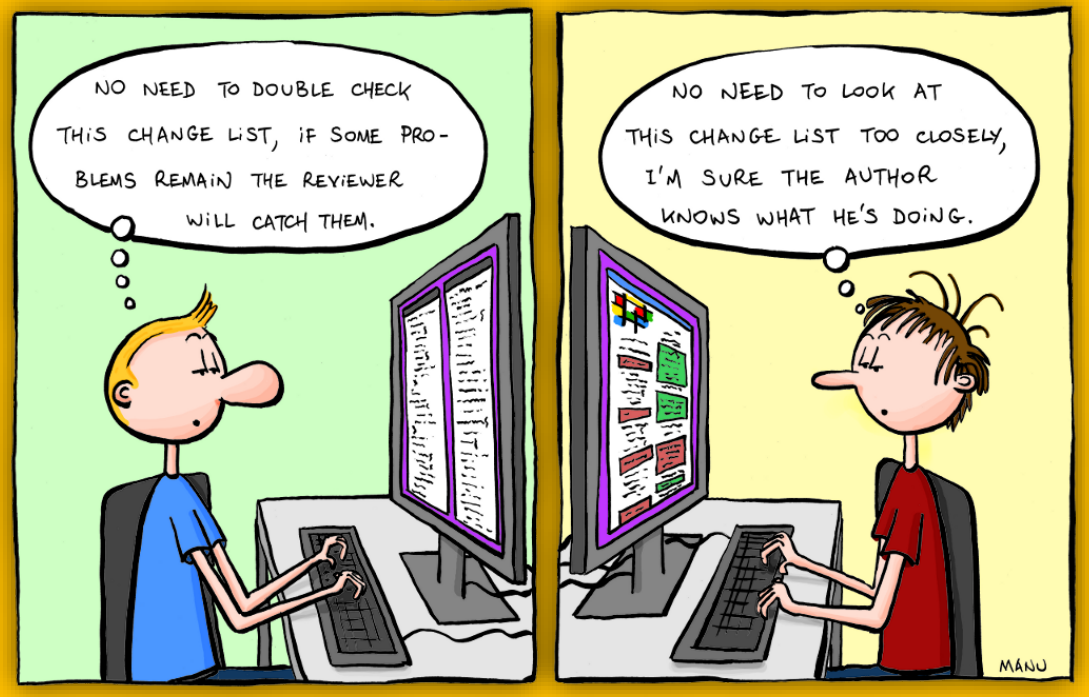
- Präzise Fehlerlokalisierung (Blame assignment)
- Findet alle Fehler
- Keine Laufzeitbeinflussung

Nachteile

- False Positives (Unentscheidbarkeit)
- Interaktion teilweise notwendig (Schleifeninvarianten)
- Großer Aufwand u. Expertise nötig

```
public int[] sort(int[] values) {
  //@ assert values != null;
  for (int j = values.length; j > 1; j--) {
    //@ assert (\forall int k; j-1 < k && k
< values.length; values[k-1] <= values[k]);
    for (int i = 0; i < j - 1; i++) {
      if (values[i] > values[i + 1]) {
        int temp = values[i];
        values[i] = values[i + 1];
        values[i + 1] = temp;
      }
    }
    //@ assert (\forall int k; j-2 < k && k
< values.length; values[k-1] <= values[k]);
  }
  //@ assert (\forall int k; 0 < k && k <
values.length; values[k-1] <= values[k]);
  return values;
}
```

Code Reviews



Beispiel

- Code Reviews
  - Kannst du mal auf den Code schauen? Ich finde das Problem nicht… Es kann nicht daran liegen, dass ich X gemacht habe. Und es kann auch nicht daran liegen, dass ich Y gemacht. Und es kann auch nicht—Moment, es **kann** daran liegen… Wieso lief das eigentlich bis hier? Danke, du hast mir sehr geholfen!

Code Reviews

- Eine Familie verschiedener Techniken
  - Pair Programming
  - Walkthroughs
  - Inspections
  - Personal reviews
  - Formal technical reviews
- Review/inspizieren:
  - Zur genauen Begutachtung
  - Mit einem Auge auf Korrektur und Bewertung
- Menschen (peers) sind die Begutachter:innen

Pair Programming

- Zwei Entwickler:innen arbeiten zusammen an einem Rechner
- Eine:r schreibt Code, während der/die andere jede getippte Zeile überprüft
- Beide Rollen werden regelmäßig gewechselt

- Vorteile
  - Wissen verteilt sich zwischen Programmierer:innen
  - Anzahl Fehler wird reduziert, Produktivität wird gesteigert
  - Keine Vorbereitung notwendig
- Nachteile
  - Paar muss miteinander klarkommen
  - Ggf. brauchen Entwickler:innen länger

Allgemeines Vorgehen bei Code Reviews

- Entwickler:innen stellen Code zur Verfügung
- Projektleiter:in setzt Meeting an
- Teilnehmende bereiten sich vor
- Meeting findet statt
- Projektmanager:in bekommt Bericht

- Verschiedene Umsetzungen: Walkthroughs, Inspection, (Formal) technical review

Walkthroughs

- Entwickler:in "führt" andere Personen durch den Quelltext
- Personen geben Feedback über mögliche Fehler, Einhaltung von Standards,…

- Vorteile:
  - Größere Gruppen können teilnehmen, dadurch mehr Wissensaustausch
  - Kaum Vorbereitungszeit
- Nachteile:
  - Entwickler:innen tendieren dazu, ihren Code zu rechtfertigen
  - Es ist schwieriger, Code und Entwickler:in zu trennen

Inspections

- Teammitglieder schauen sich Material an
- Team trifft sich und diskutiert über Material
- Ggf. werden nur ausgewählte Aspekte betrachtet
- Vorteile:
  - Fokus auf wichtige Dinge (wenn man sie kennt)
- Nachteil:
  - Gute Moderatorion notwendig

Formal Technical Review

- Eingeplantes Meeting mit festgeschriebenem Ablauf
- Ergebnis wird in Bericht zusammengefasst
- Fokus auf technische Aspekte, z.B. Abweichung von Anforderungen oder Standards
- Unabhängiges Team ohne Entwickler:innen
- Vorteil:
  - Unabhängig von Entwickler:in
  - Festgeschriebener Ablauf
- Nachteil:
  - Aufwand

Personal Review

- Informell durch Entwickler:in selbst
- Kann jeder Entwickler:in einfach durchführen

- Nicht besonders objektiv

Was sollte wann reviewed werden

- Jedes Projekt-Artefakt:
  - Anforderungen
  - Design
  - Code
  - Dokumentation
  - …
- Meetings sind fest eingeplant und sollten definierte Dauer haben

Aufgaben des Teams

- Guten Review erstellen
  - Team ist verantwortlich für Review, nicht das Produkt
- Probleme finden (nicht beheben)

- Entscheidung treffen:
  - Akzeptiert, akzeptiert mit kleinen Änderungen (einstimmig)
  - Bedeutende Änderungen, abgelehnt (ein Veto reicht)

Aufgaben der Teamleiter:innen

- Voreilige Reviews vermeiden
- Guten Review sicherstellen…
- …oder Gründe für Scheitern berichten
  - Fehlendes Material
  - Fehlende, unvorbereitete Gutachter:innen
- Meetings koordinieren
  - Material verteilen
  - Zeitplan für Meeting (und dessen Einhaltung)
  - Ort für Meeting

Aufgaben der Gutachter:innen

- Vorbereitung durch review des Materials
- Aktiv teilnehmen
- Professionelles Verhalten
- Berechtigte positive und negative Kommentare

Bericht

- Zusammenfassung
- Gefundene Probleme
- Empfehlung

- Projektleiter über Status informieren
- Frühwarnsystem für mögliche Probleme
- Logbuch für Fortschritt und involvierte Leute

Testen vs. Reviews

- Testen verläuft in 2 Phasen:
  - Testfälle finden Fehler
  - Ursache von Fehlern muss gefunden werden
- Bei Reviews findet man Fehler und deren Ursache in einem Schritt
- Erst Review, dann Testen
- Erst Testen, dann Verifikation

Was Sie mitgenommen haben sollten:

- Warum brauchen wir Tests/Verifikation/Code Reviews?
- Was kann man mit Tests nicht zeigen? Warum?
- Nennen/Erklären Sie die 4 vorgestellten Ebenen von Tests.
- Erklären Sie Black-Box/White-Box/Regressions/ Nightly/Daily-Builds.
- Nennen/Erklären Sie den Vorteil von Code Reviews gegenüber testen.
- Welche Strategie würden Sie einem kleinen Unternehmen (3 Mitarbeiter) empfehlen, um möglichst fehlerfreie Software auszuliefern? Begründen Sie Ihre Entscheidung.

Literatur

- McConnell. Code Complete. 2004. [Chapter 20-22] (contains also references of further interesting papers)
- Sommerville. Software Engineering. 2002 [Chapter 22-23]
- Beckert and others. Verification of object-oriented software: The KeY approach. 2007
- Code Reviews: http://www.baskent.edu.tr/~zaktas/courses/Bil573/IEEE_Standards/1028_2008.pdf
- Wikipedia