Softwaretechnik

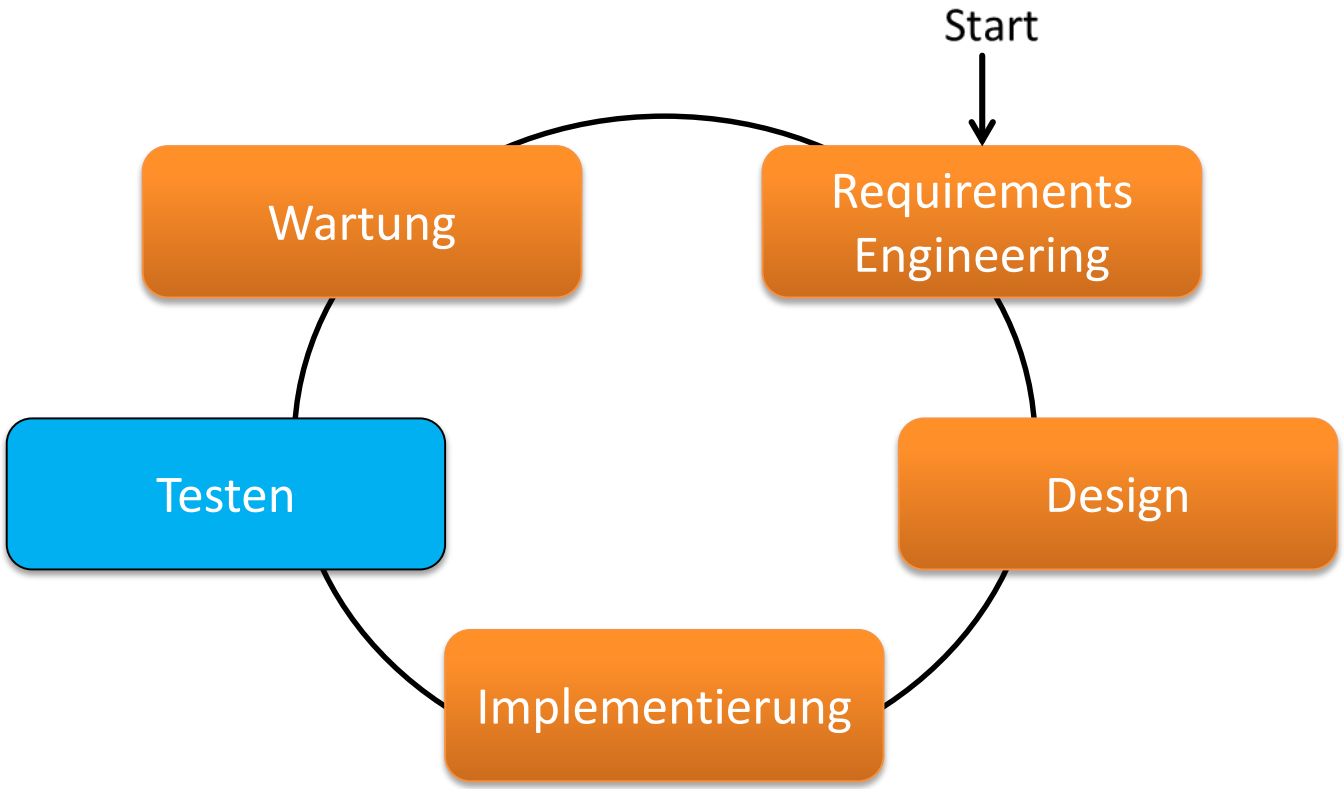
Testen

SOFTWARE SYSTEME

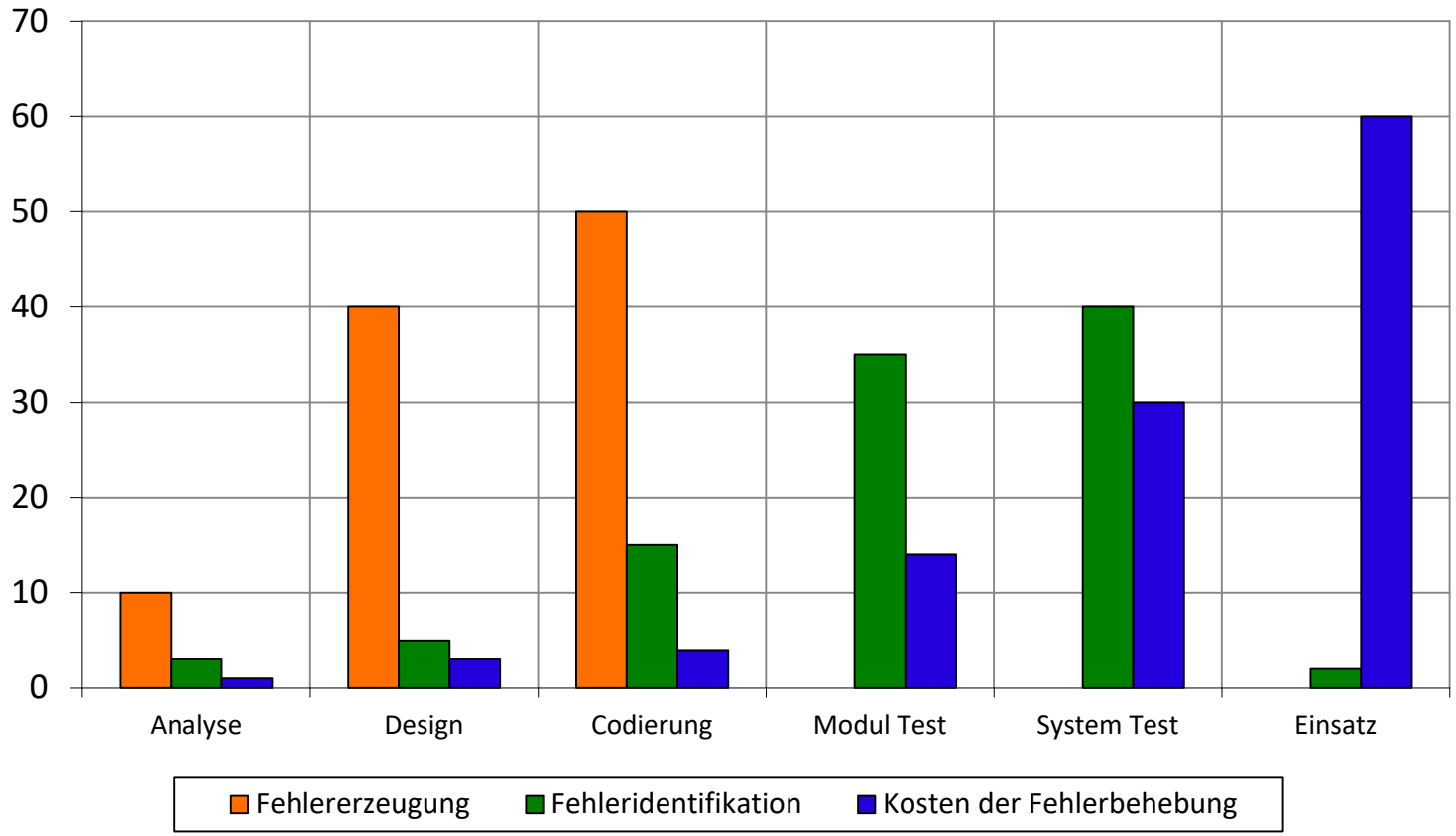
UNIVERSITÄT LEIPZIG

Prof. Dr.-Ing. Norbert Siegmund
Software Systems

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, Verifikation und Code Reviews kennen lernen

Testen:
Einordnung und systematisches Vorgehen

Ziele des Testings

"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
  - Sicherstellen, dass das Programm nicht abstürzt
  - Regressionstest: keine neuen Fehler einführen bei neuen commits
  - Sicherstellen, dass die Anforderungen erfüllt sind
  - Sicherstellen, dass es keine Seiteneffekte gibt

- Ein erfolgreicher Test findet Fehler

Herausforderungen

- Man muss annehmen, dass ein Programm fehlerhaft ist; nicht, dass es korrekt ist*1
- Entgegen jeder anderen Software-Entwicklungsaktivität (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

*1: Myers. A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections. 1978.

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.

Mittl. Anzahl von Fehlern

- Industrie:
  - 30-85 Fehler per 1000 LOC (vor dem Testen);
  - 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 (ca. 2015)
  - 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* (siehe Requirements Engineering)
2. Software Design basierend auf *information hiding and encapsulation* (siehe SW-Qualitätvorlesung + Design Patterns)
3. Extensive *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

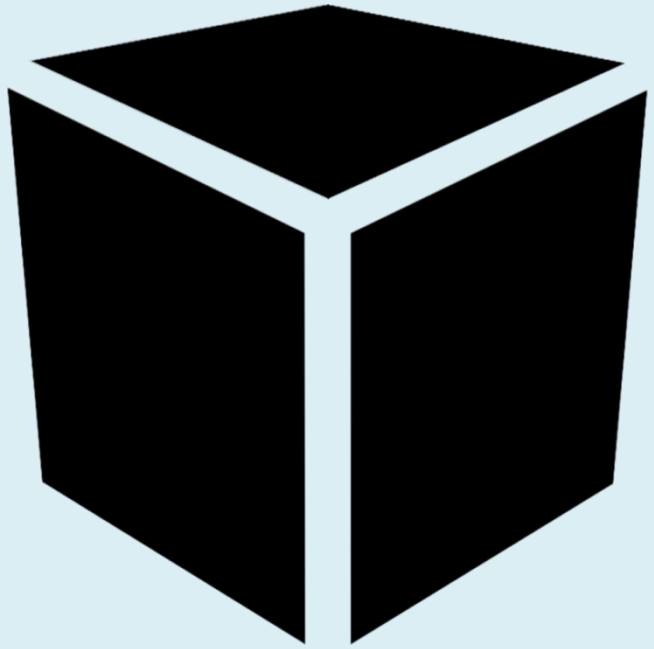
Verschiedene Features von Programmiersprachen und Systemen sind häufige Quellen von Fehlern:

- ***Goto Statements*** und anderen unstrukturierte Programmierkonstrukte machen Programme *schwer zu verstehen und zu modifizieren*.
  - Verwendet nur strukturierte Programmierkonstrukte
- ***Gleitkommazahlen*** sind *inhärent ungenau* und können zu fälschlichen Vergleichen führen.
  - Ganzzahlen sind sicherer für exakte Vergleiche
- ***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.
  - Minimiere inter-Process Abhängigkeiten
- ***Rekursion*** kann zu *verworrener Logik* führen und den Stack-Speicher überladen.
  - Verwende Rekursion in eine disziplinierten und kontrollierten Weise
- ***Interrupts*** erzwingen den Transfer der Kontrolle *unabhängig vom derzeitigen Kontext* und können daher zum Abbruch / Unterbrechung kritischer Operationen führen.
  - Minimiere die Verwendung von Interrupts; bevorzuge Exceptions

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
  - 4 Zähler müssen über 248 Tage emuliert werden

Boeing 787 Dreamliners contain a potentially catastrophic software bug

Beware of integer overflow-like bug in aircraft's electrical system, FAA warns.

DAN GOODIN - 5/1/2015, 7:55 PM

152

A software vulnerability in Boeing's new 787 Dreamliner jet has the potential to cause pilots to lose control of the aircraft, possibly in mid-flight, Federal Aviation Administration officials warned airlines recently.

The bug—which is either a classic integer overflow or one very much resembling it—resides in one of the electrical systems responsible for generating power, according to memo the FAA issued last week. The vulnerability, which Boeing reported to the FAA, is triggered when a generator has been running continuously for a little more than eight months. As a result, FAA officials have adopted a new airworthiness directive (AD) that airlines will be required to follow, at least until the underlying flaw is fixed.

"This AD was prompted by the determination that a Model 787 airplane that has been powered continuously for 248 days can lose all alternating current (AC) electrical power due to the generator control units (GCUs) simultaneously going into failsafe mode," the memo stated. "This condition is caused by a software counter internal to the GCUs that will overflow after 248 days of continuous power. We are issuing this AD to prevent loss of all AC electrical power, which could result in loss of control of the airplane."

The memo went on to say that Dreamliners have four main GCUs associated with the engine mounted generators. If all of them were powered up at the same time, "after 248 days of continuous power, all four GCUs will go into failsafe mode at the same time, resulting in a loss of all AC electrical power regardless of flight phase." Boeing is in the process of developing a GCU software upgrade that will remedy the unsafe condition. The new model plane previously experienced a battery problem that caused a fire while one aircraft was parked on a runway.

The memo doesn't provide additional details about the underlying software bug. Informed speculation suggests it's a signed 32-bit integer overflow that is triggered after 2^{31} centiseconds (i.e. 248.55 days) of continuous operation.

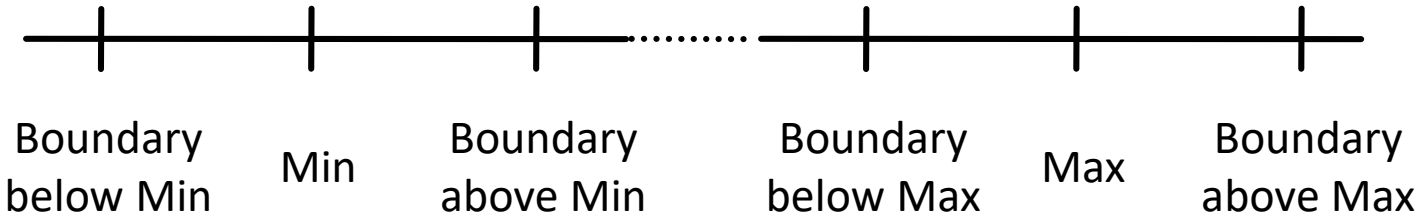
Äquivalenzklassen finden

- 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()`



- Tests für folgende Werte:

$$\underbrace{99 \quad 100 \quad 101}_{\text{lower boundary}} \qquad \underbrace{998 \quad 999 \quad 1000}_{\text{upper boundary}}$$

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

Zusammenfassung Black-Box Testen

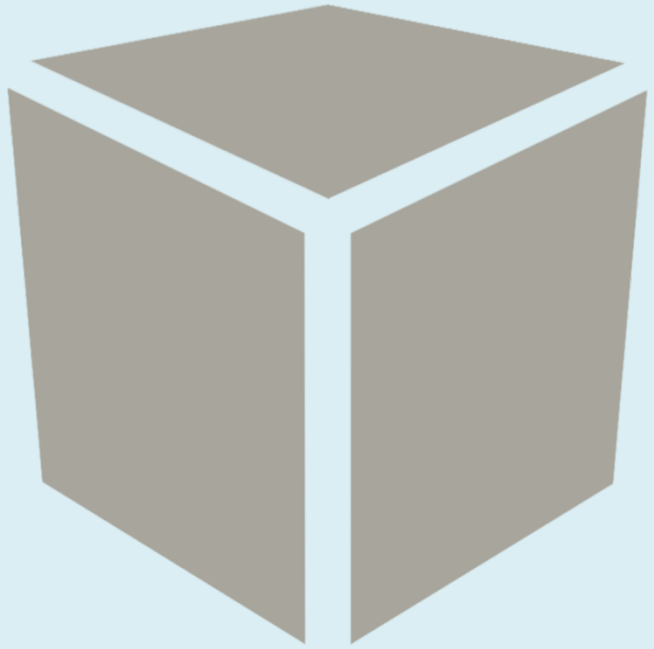
- Geeignet zum Finden von:
  - Inkorrekter oder fehlender Funktionalität (aus Spezifikation)
  - Schnittstellenfehler
  - Fehler in Datenstrukturen oder externen Zugriffen
  - Problemen von nicht-funktionalen Eigenschaften
  - Fehlern beim Ablauf von Prozessen
- Grenzen:
  - Spezifikation ist meist abstrakt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



White-Box Testing

- Idee: Code anschauen und systematisch alle Anweisungen, Bedingungen und Pfade mindestens einmal ausführen
- Ideal: alle Ausführungspfade testen (aber, nicht praktikabel)

- Stattdessen: Teste z.B. jedes Statement mind. 1 mal
- Beispiel:

```
if (x > 5) {
  System.out.println(‘‘hello’’);
} else {
  System.out.println(‘‘bye’’);
}
```

Es gibt zwei mögliche Pfade durch den Code,
$x > 5$ und $x \leq 5$.
Ziele darauf ab, jeden auszuführen.

Herausforderungen

- Oft durch Entwickler selbst durchgeführt
- Welcher Entwicklerin geht schon gern davon aus, dass ihr 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

Systematisches Vorgehen

- Wurde jede Funktion mindestens einmal ausgeführt?
- Wurde jede Anweisung mindestens einmal ausgeführt?
- Wurde jeder Zweig von if/case-Anweisungen ausgeführt?
- Wurde überprüft, ob jeder boolsche (Sub-) Ausdruck wahr und falsch werden kann?
- Ausführungspfade:
  - Wurde jeder mögliche Pfad ausgeführt?
  - Problem: Unendlich viele Pfade
  - Typischerweise Kompromiss: 0-1-viele Ausführungen
- Datenstrukturen: Jeder mögliche Zustand

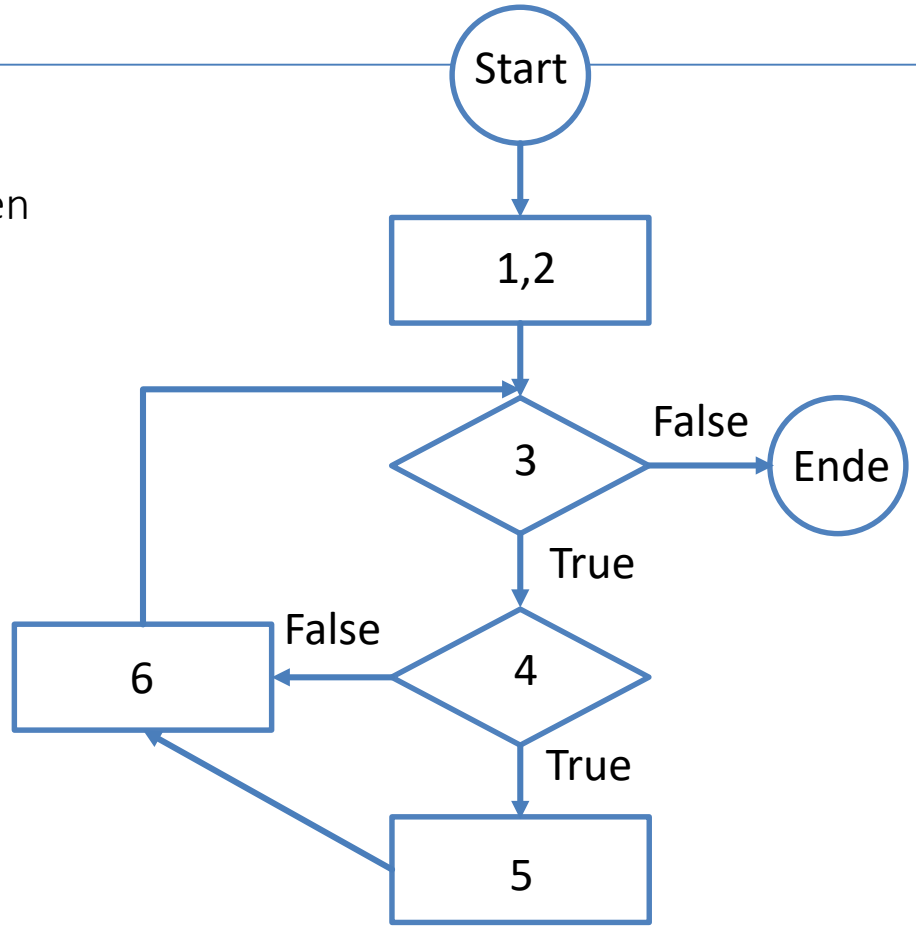
Arten

- Kontrollflussorientiert
  - Anweisungsüberdeckung (C0)
  - Kantenüberdeckung (C1)
  - Bedingungsüberdeckung (C2, C3)
  - Pfadüberdeckung (C4)
- Datenflussorientiert
  - Nicht behandelt

Kontrollflussgraph

- G = (V, E) wobei
  - V ist eine Menge von Basisblöcken
  - E ist eine Menge von gerichteten Kontrollflüssen

- Beispiel:



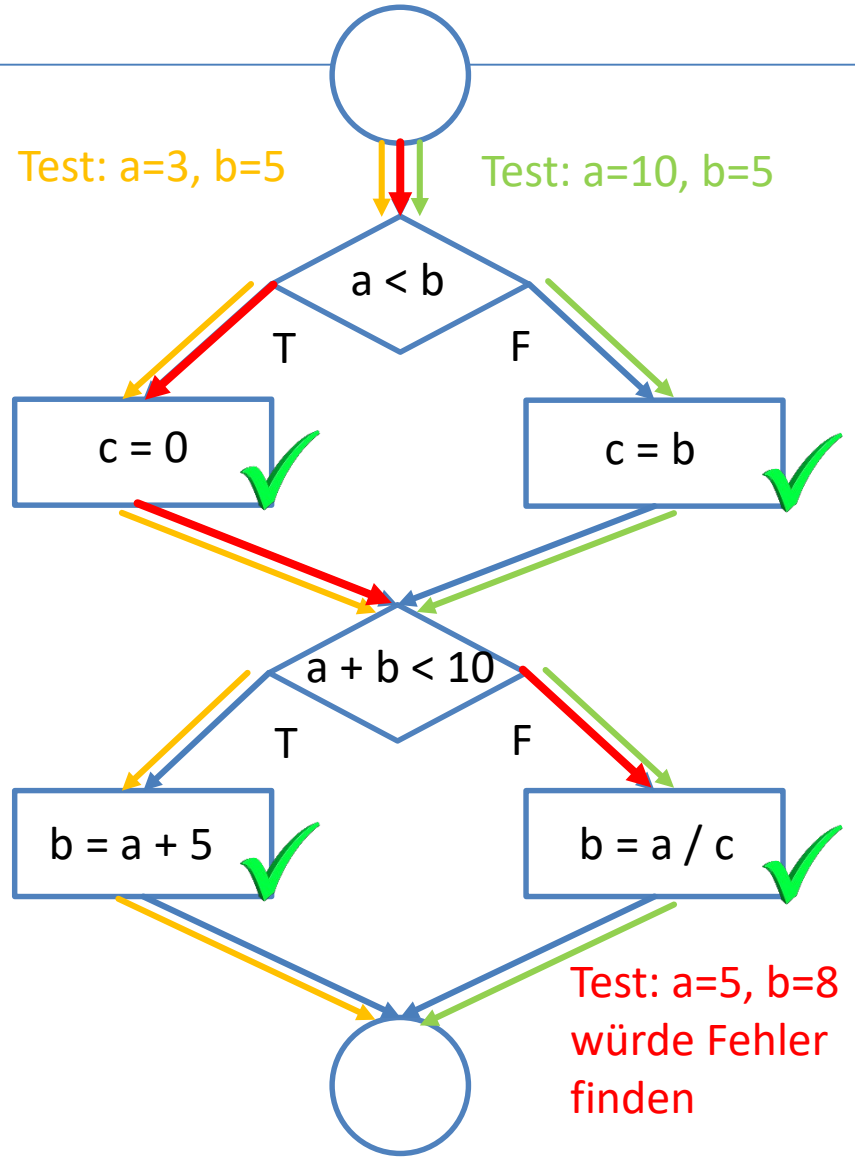

```
1. a = Read(b)
2. c = 0
3. while (a > 1)
4.     If (a^2 > c)
5.         c = c + a
6.     a = a - 2
```

Graph = Menge aller möglichen Ausführungen eines Programmes.
Pfad = Eine konkrete Ausführung eines Programmes.

Anweisungsüberdeckungsverfahren (C0- Test)

- Wähle Testmenge so, dass alle Anweisungen des Testobjektes mind. einmal ausgeführt wurden

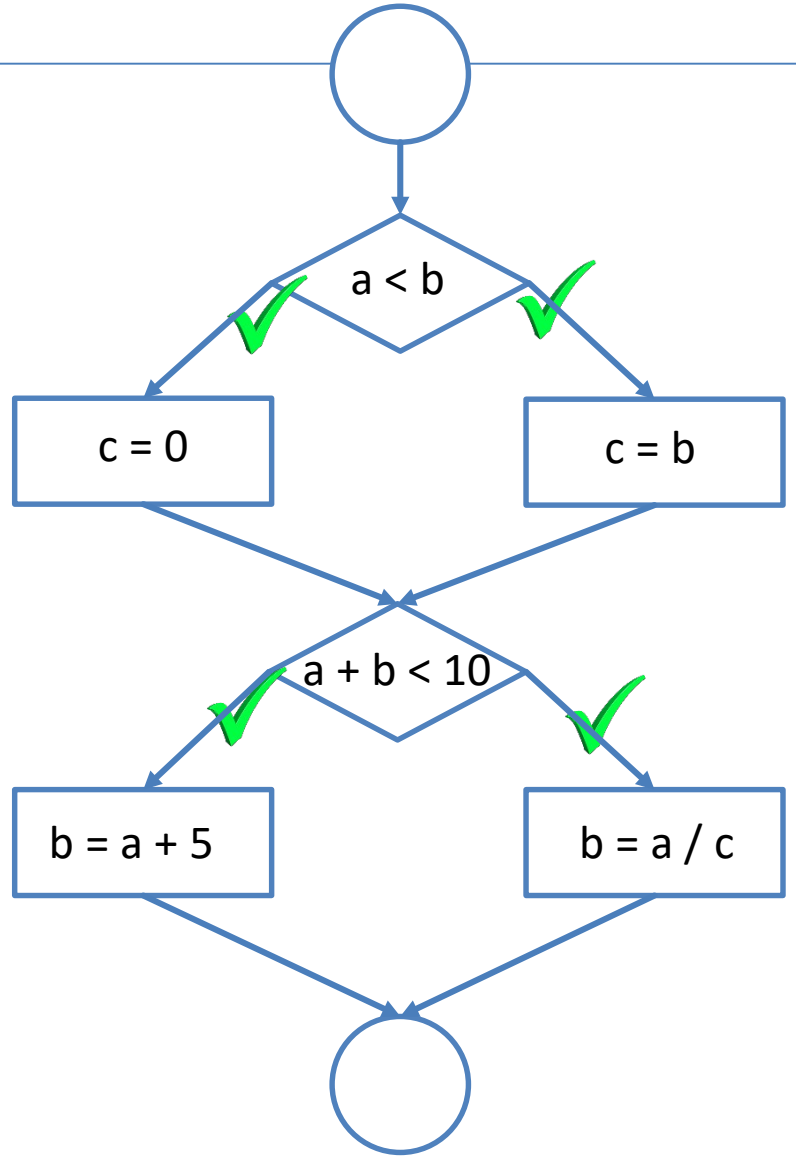
- Probleme:
  - Nicht alle Wege müssen geprüft werden
  - Schleifen werden unzureichend geprüft



Zweigüberdeckungstest (C1 - Test)

- Wähle Testmenge so, dass alle Kanten des Testobjektes mind. einmal ausgeführt wurden
  - Inkludiert C0 Test
  - Minimaler Test
  - Auch Zweige ohne Code werden ausgeführt

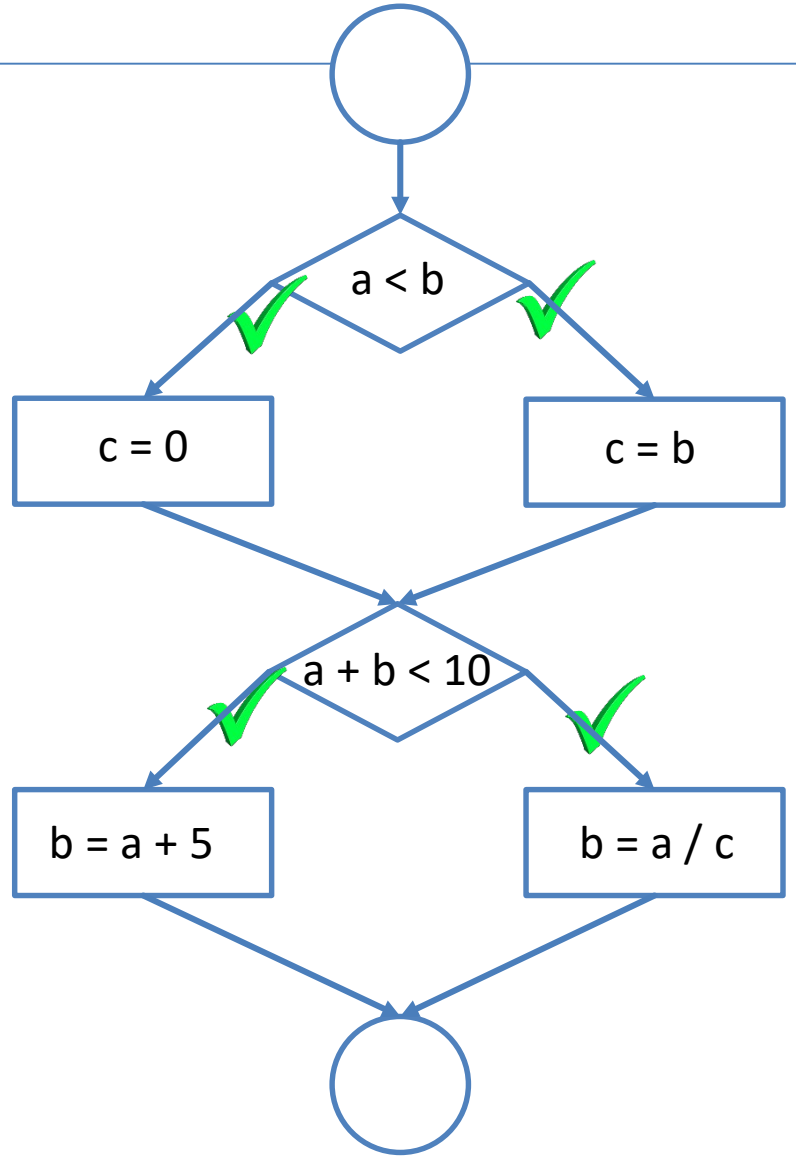
- Probleme:
  - Nicht alle Wege müssen geprüft werden
  - Schleifen werden unzureichend geprüft



Bedingungsüberdeckungstest (C2 / C3 Test)

- Wähle Testmenge so, dass alle Teilbedingungen des Testobjektes überdeckt werden
  - Alle Teilformeln von Bedingungen auf true und false prüfen (C2)
  - Alle Wahrheitskombinationen atomarer Teilformeln auf true und false prüfen (C3)

- Probleme:
  - C3 führt zu exp. Anstieg der Testfälle
  - Schleifen werden unzureichend geprüft



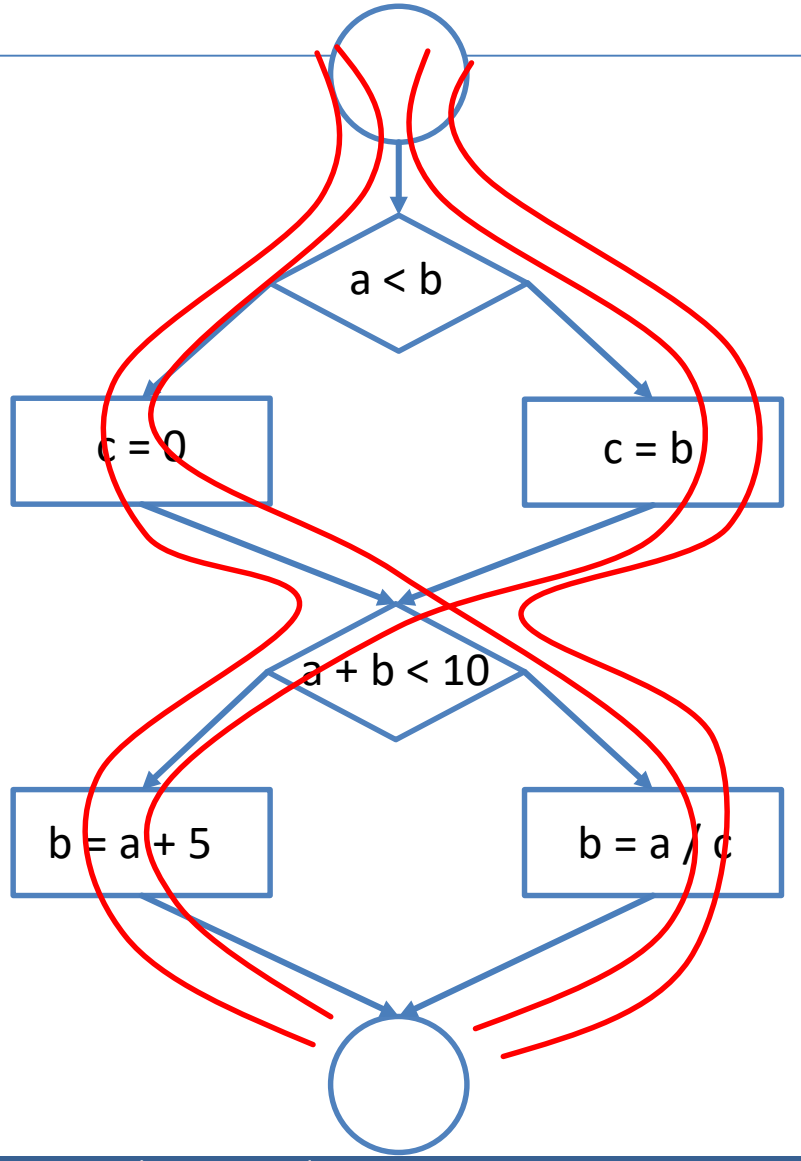
Pfadüberdeckungsverfahren

- 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 haben häufig fälschliche Annahme, dass ein bestimmter Pfad nicht ausgeführt wird
  - Tippfehler sind zufällig; somit wahrscheinlicher in ungetesteten Pfaden

Beispiel

- 2 Bedingungen = 2^2 = 4 Pfade
- Was ist mit Schleifen?



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 Ausführungspfade und Fallunterscheidungen
- White-Box Tests ergänzen Black-Box Tests
- Viele Tools, die Testabdeckung messen und visualisieren (https://about.codecov.io/blog/the-best-code-coverage-tools-by-programming-language/)
- Trotz systematischem Vorgehen kann man nie sicher sein, dass der Quelltext fehlerfrei ist
  - Dijkstra hat nach 40 Jahren immer noch Recht

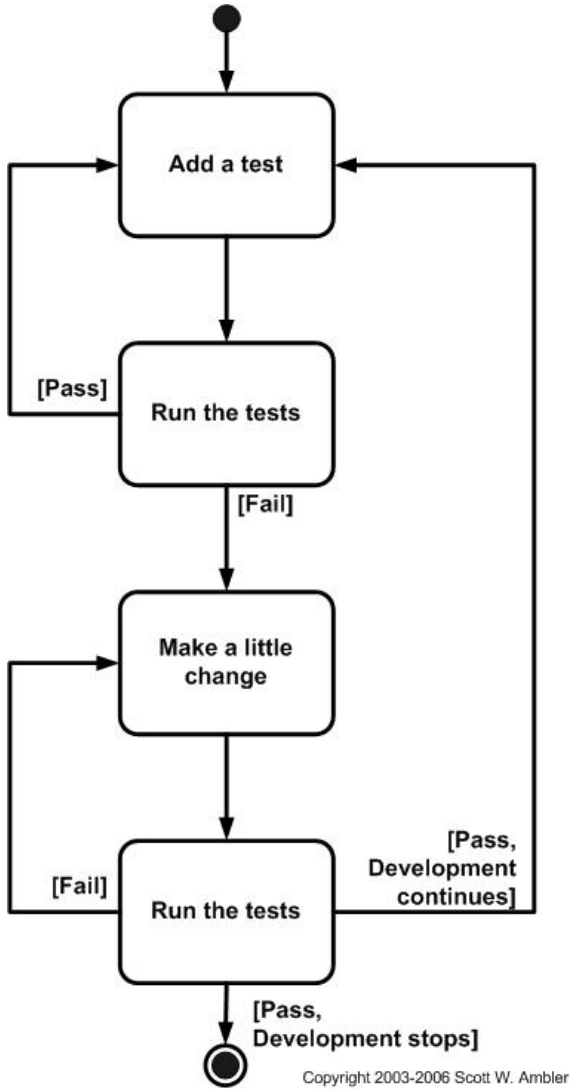
Konkrete Testverfahren

Testverfahren

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

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 verweigern das Hinzufügen auch nur einer einzelnen Zeile Code solange dafür kein Test existiert
- Sobald ein Test vorhanden ist, setzt der Programmierer 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 Tests

- Ziel: Individuelle Module (meist auf Klassenebene) und Funktionen werden getestet, um deren korrekte Funktionsweise sicherzustellen

- Typischerweise automatisiert
- Oft durch Entwickler 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

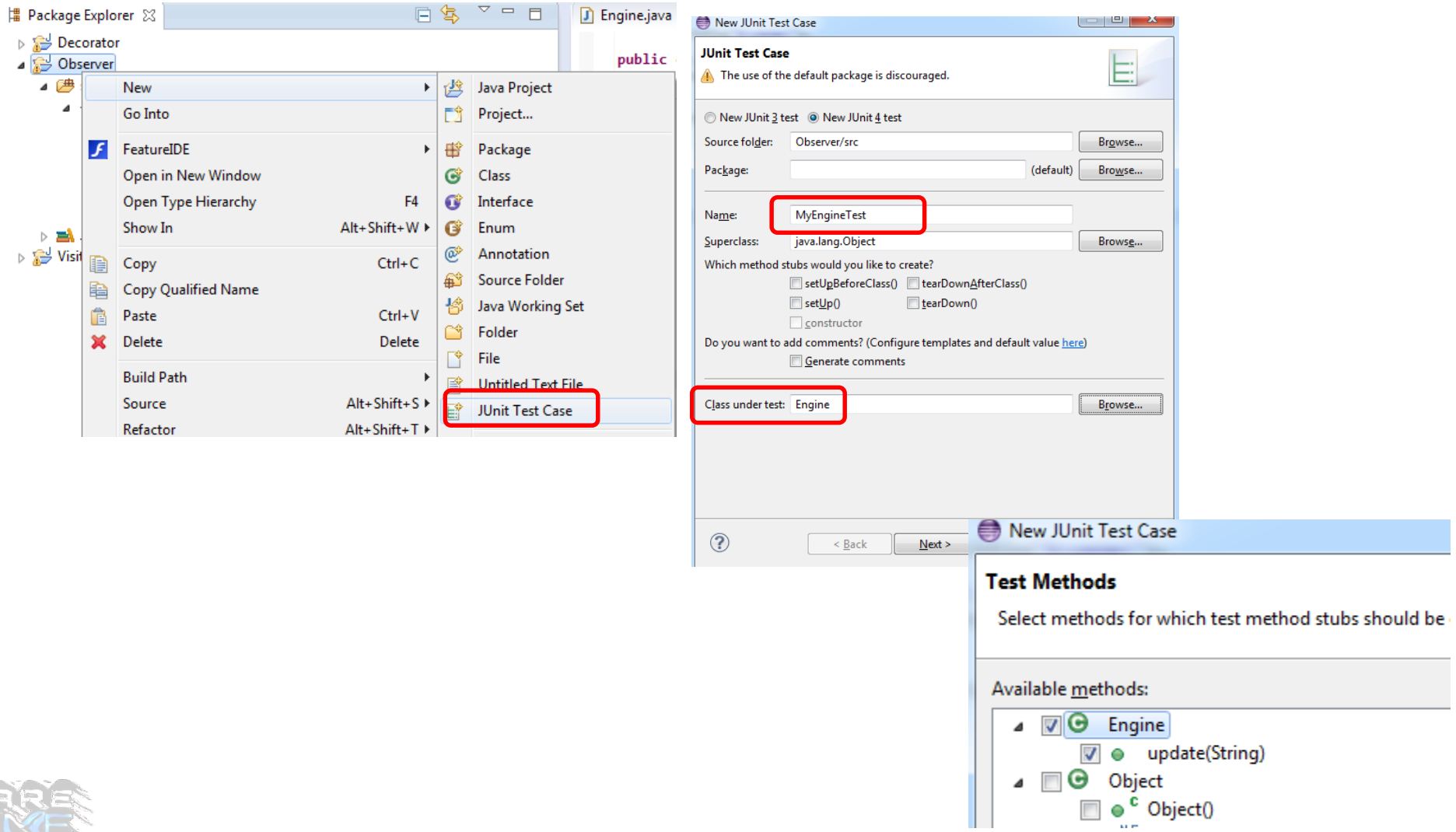
```
@Test
public void multiplicationOfZeroIntegersShouldReturnZero() {
  // MyClass is tested
  MyClass tester = new MyClass();
  // Tests
  assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
  assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
  assertEquals("0 x 0 must be 0", 0,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

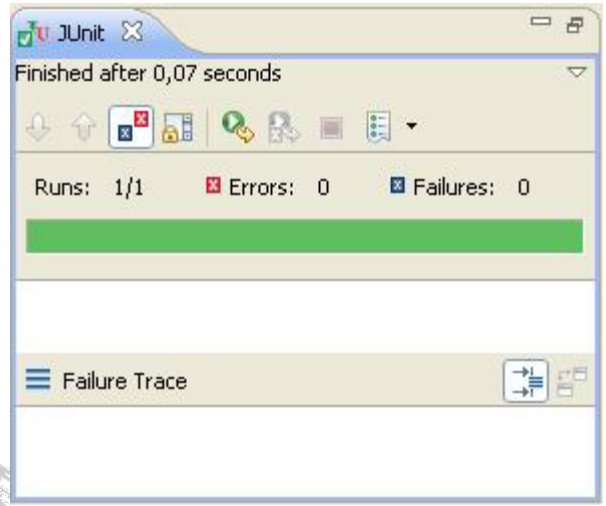
Annotation	Description
@Test public void method()	The `@Test` annotation identifies a method as a test method.
@Test (expected = Exception.class)	Fails if the method does not throw the named exception.
@Test(timeout=100)	Fails if the method takes longer than 100 milliseconds.
@Before public void method()	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.

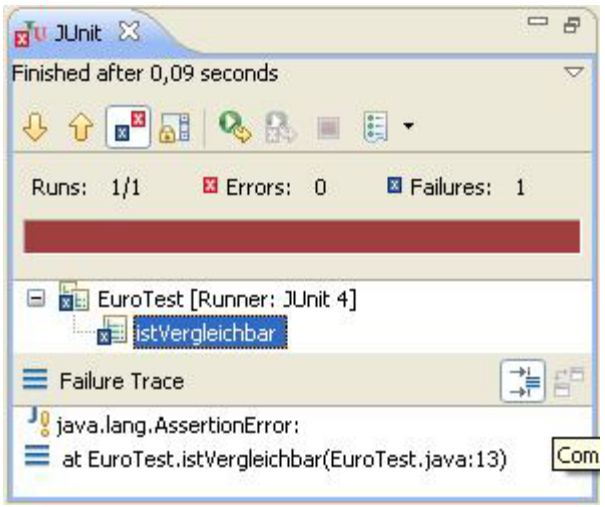
JUnit in Eclipse



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")));
    }

}
```





2. Integrations-Tests (IT)

- Module werden kombiniert und als Gruppe getestet
- Nach Unit-Tests, vor System-Tests
- Zweck: Erfüllen Module im Zusammenspiel funktionale und nicht-funktionale Anforderungen?
- IT basiert auf Black-Box-Tests
- IT oft als Top-down IT or Bottom-up IT

IT: Top-Down

1. Kontrollflussmodul führt Test aus, alle untergeordneten Komponenten werden durch Stubs ausgetauscht
2. Untegordnete Komponenten werden durch eigentliche Komponenten 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. Komponente, die Tests koordiniert
3. Cluster wird getestet
4. Getestet Cluster werden kombiniert und wieder getestet, bis höchste Ebene erreicht ist

Aufgabe

- Was ist besser, Top-Down oder Bottom-Up Tests?

- Top-Down:
  - Gesamtsystem im Blick
  - Ganze Use-cases testen
  - Prozess (also Ablauf und Reihenfolge mehrerer Aktionen) kann getestet werden

- Buttom-Up:
  - Lokalisierung von Fehlern ist einfach
  - Fokus auf komplizierte Kombinationen von Komponenten

3. 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:
  - Verantwortlicher Entwickler 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/)

4. Acceptance-Tests

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

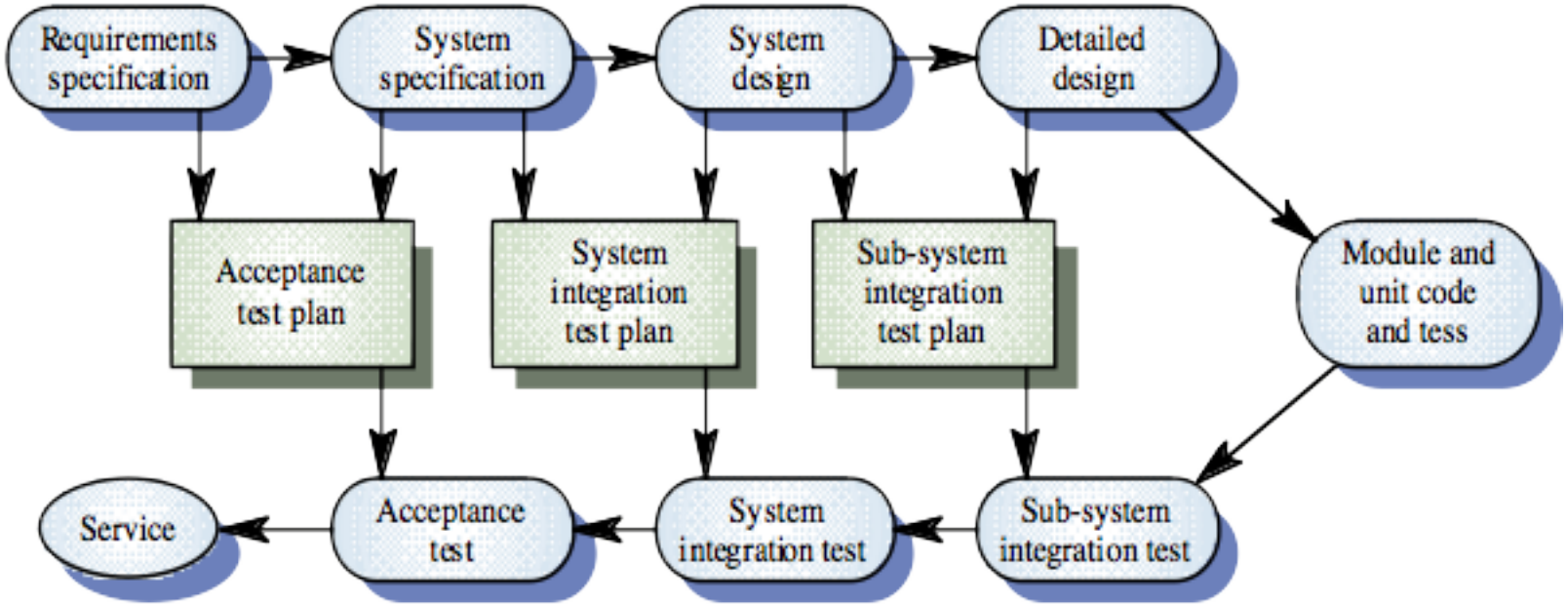
Wann findet man die meisten Fehler?

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

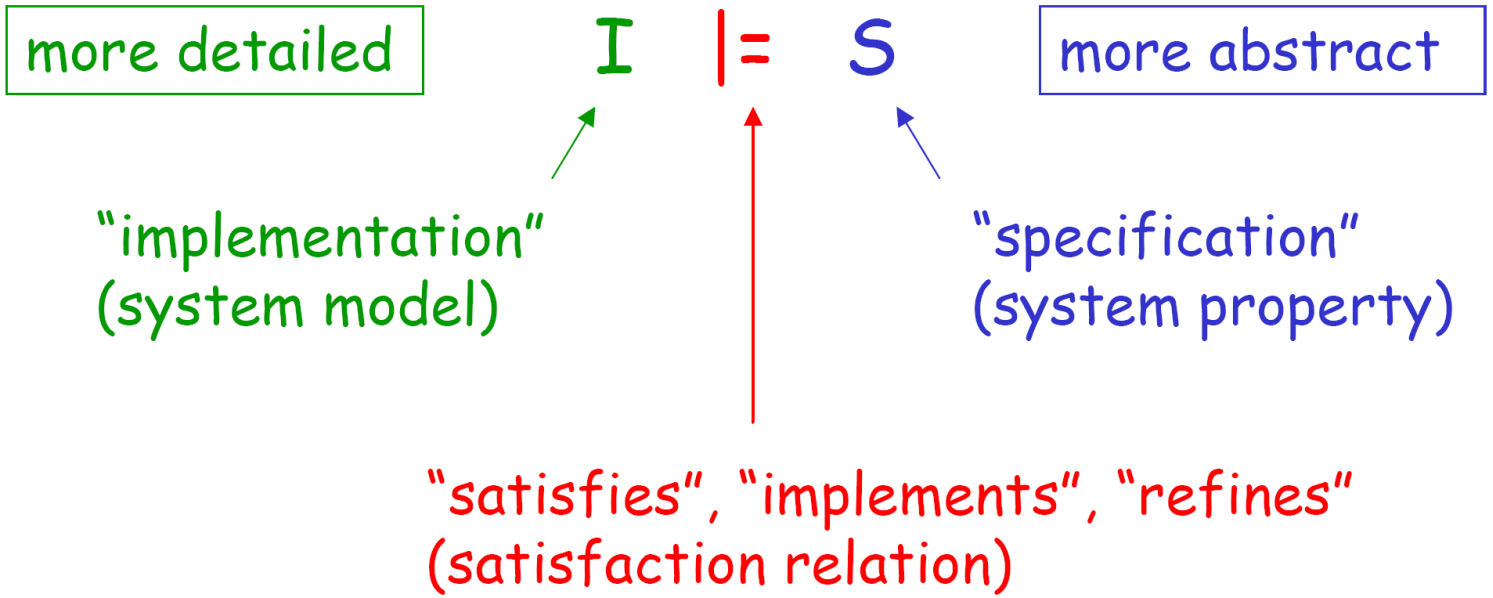
Statische Programmanalyse:

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

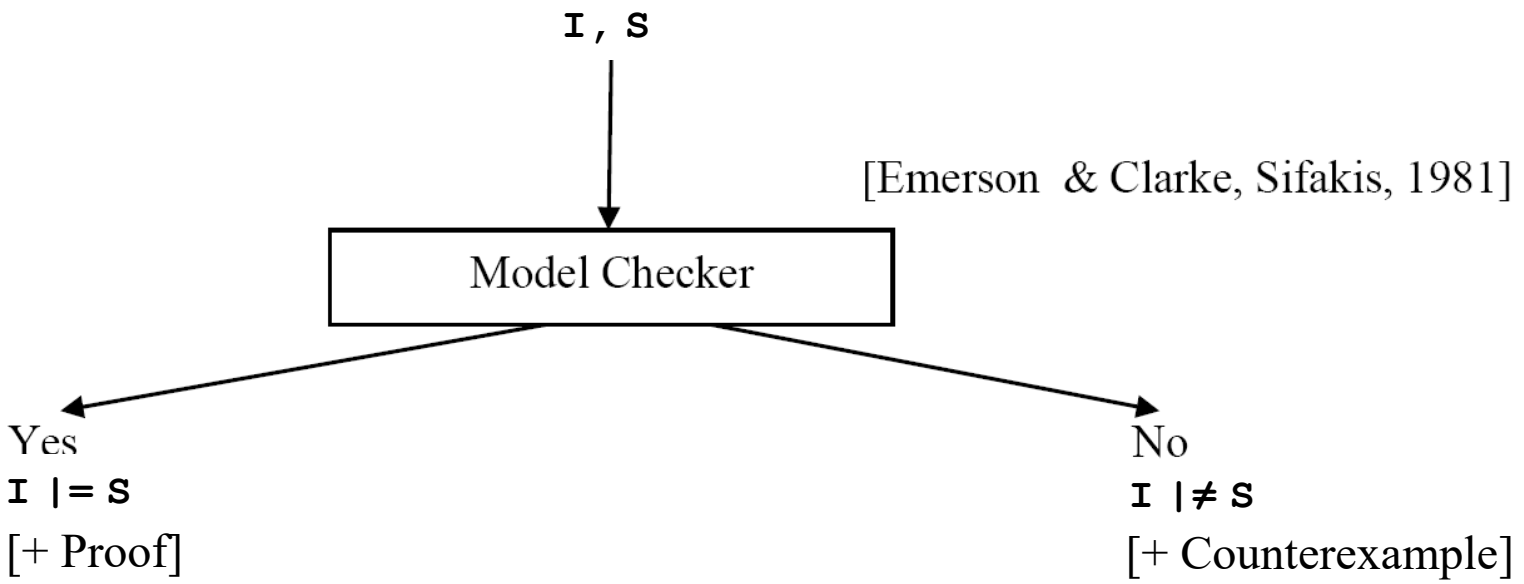
Mathematisch-basierte Verifikation:

- 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

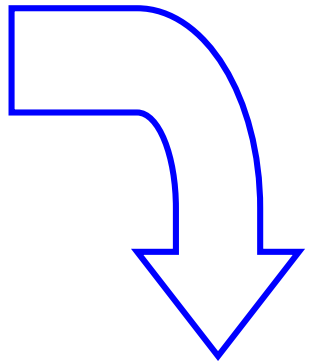


Model Checking



Beispiel

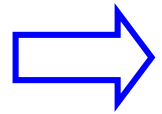
```
import java.util.Random;
public class Rand {
  public static void main (String[] args) {
    Random random = new Random(42);  // (1)

    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);
  }
}
```



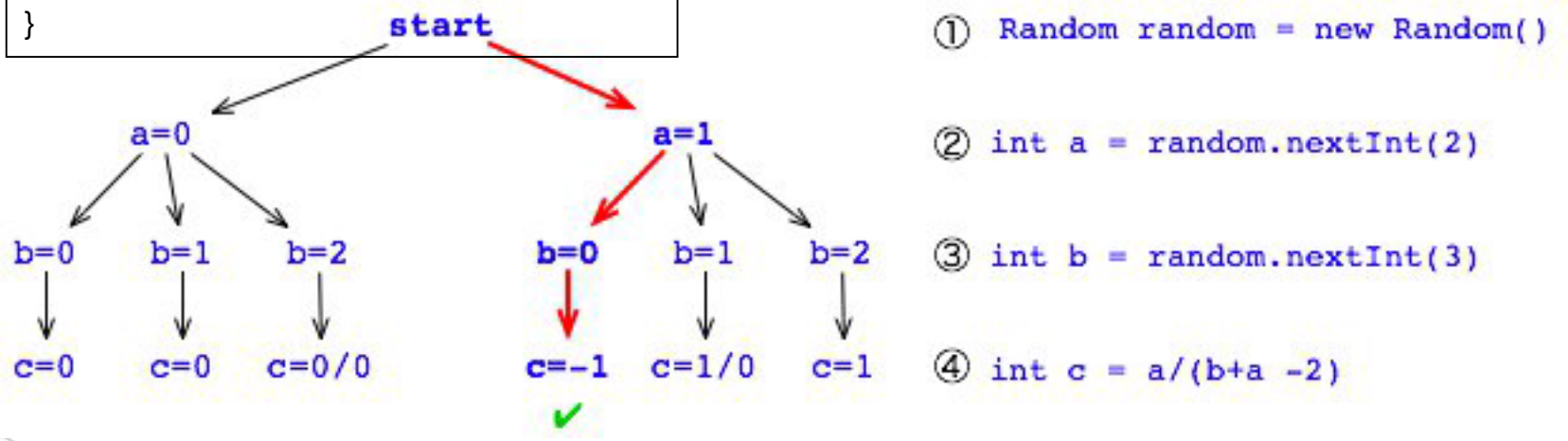
```
> java Rand
a=1
b=0
c=-1
>
```

Test Run

```
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);
  }
}
```



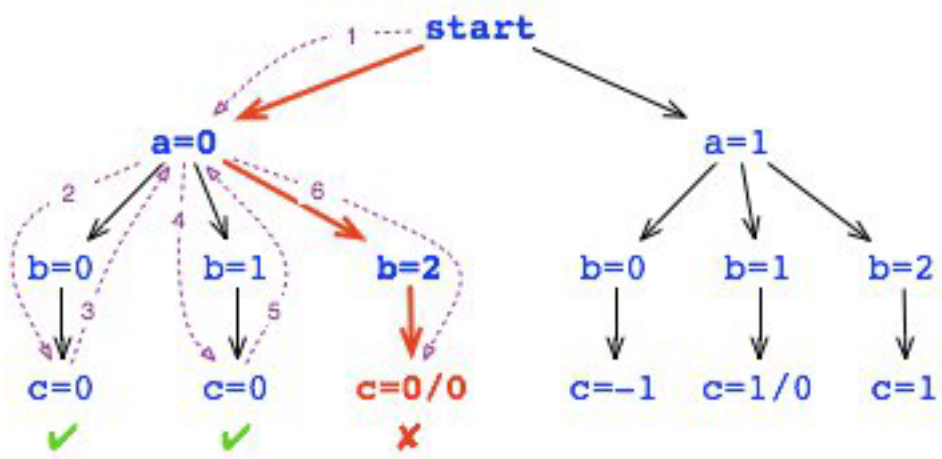
```
> java Rand
a=1
b=0
c=-1
>
```



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

```
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;
}
```

Vorteile

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

Nachteile

- Findet nicht alle Fehler
- Ausbremsen des Systems

Deduktive Verifikation

```
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;
}
```

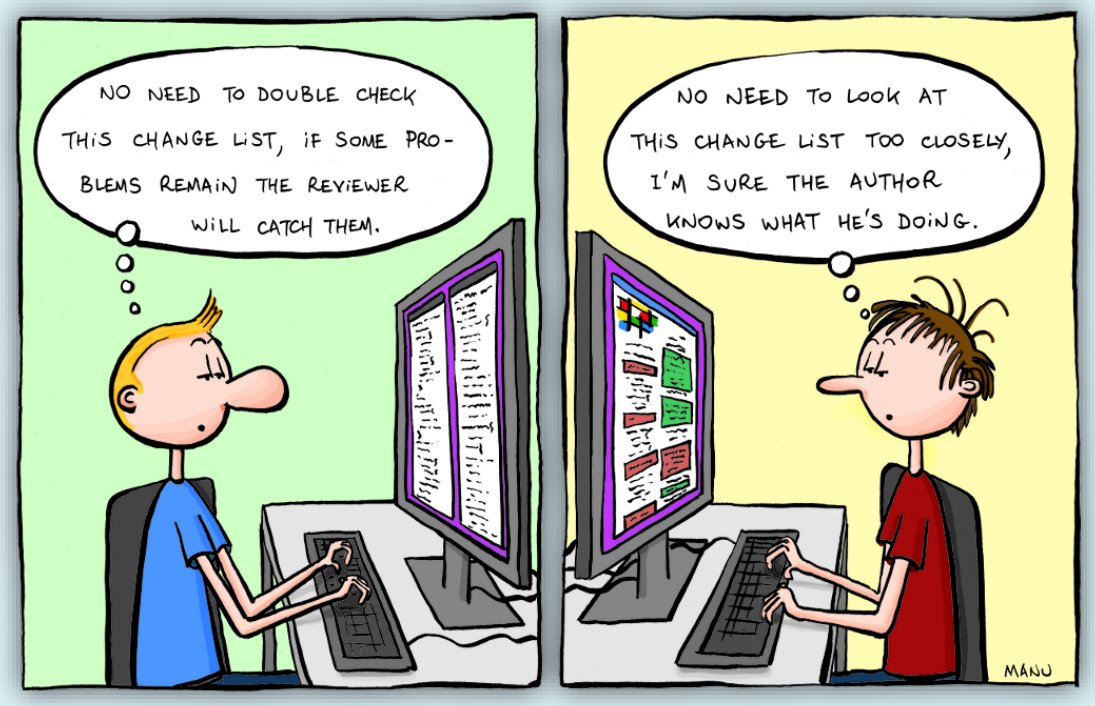
Vorteile

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

Nachteile

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

Code Reviews

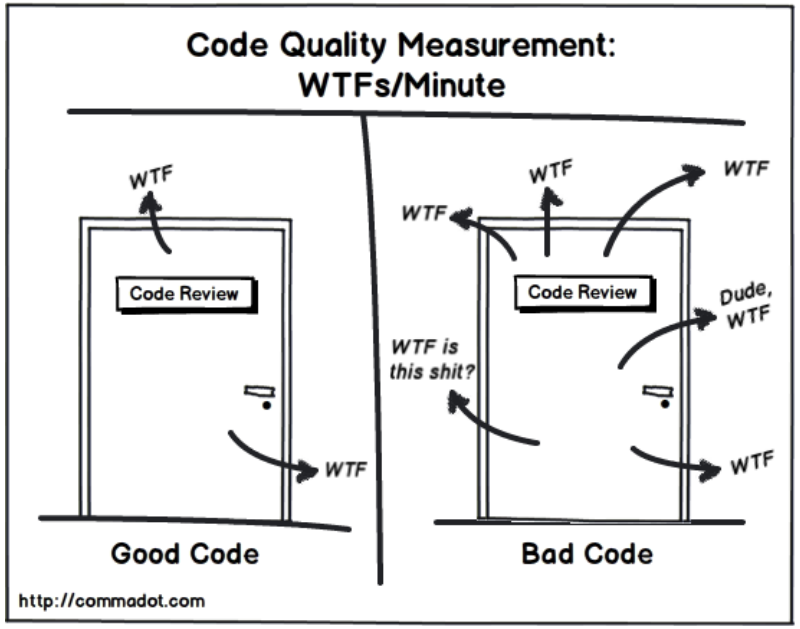


Beispiel

- Code Reviews
  - Kannst du mal auf den Code schauen? Ich finde das Problem nicht… Es kann nicht daran liegen, weil ich X gemacht habe. Und es kann auch nicht daran liegen, weil ich Y gemacht. Und es kann auch nicht—Moment, es **kann** daran liegen. 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



Pair Programming

- Zwei Entwicklerinnen arbeiten zusammen an einem Rechner
- Eine schreibt Code, während die Andere jede getippte Zeile überprüft
- Beide Rollen werden regelmäßig gewechselt

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

Allgemeines Vorgehen bei Code Reviews

- Entwicklerin stellt Code zur Verfügung
- Projektleiterin setzt Meeting an
- Teilnehmer bereiten sich vor
- Meeting findet statt
- Projektmanagerin bekommt Bericht

- Verschiedene Umsetzungen: Walkthroughs, Inspection, (Formal) technical review
- IEEE1028 Standard

Walkthroughs

- Entwicklerin "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:
  - Entwicklerin tendieren dazu, ihren Code zu rechtfertigen
  - Es ist schwieriger, Code und Entwickelnde 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:
  - Guter Moderator 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
- Vorteil:
  - Unabhängig vom Entwickler
  - Festgeschriebener Ablauf
- Nachteil:
  - Aufwand

Personal Review

- Informell durch Entwicklerin selbst
- Kann jede Entwicklerin einfach durchführen

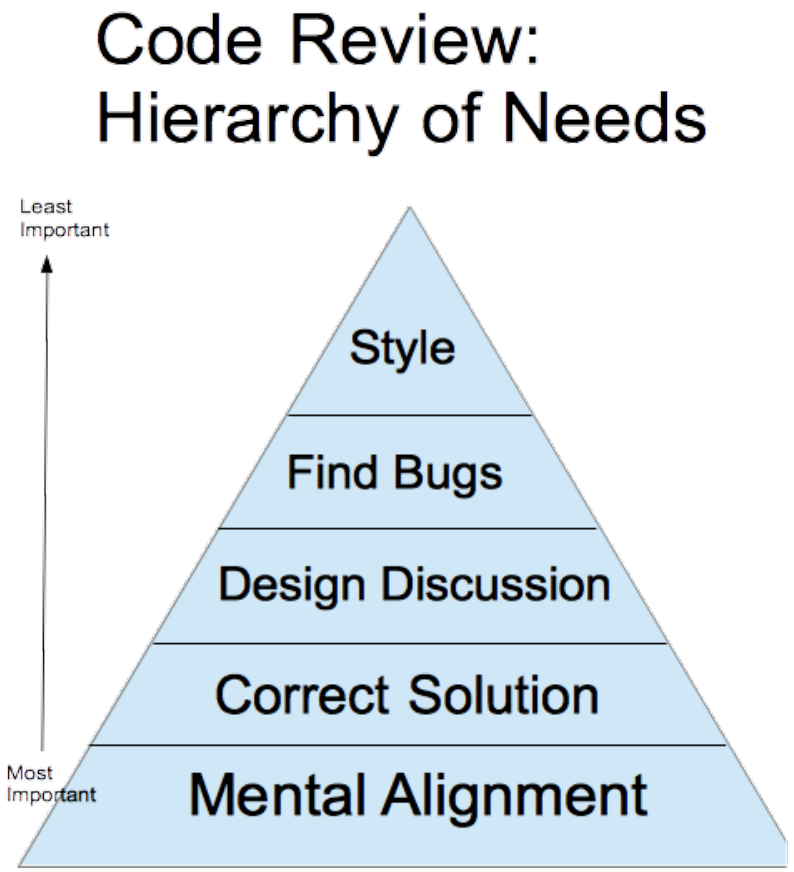
- Nicht besonders objektiv

Was sollte wann gereviewed werden

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

Review benötigt	Review nicht zwingend
Komplizierte Algorithmen	Triviale Prozesse
Kritische Pfade, wo Fehler zu Systemfehlern führen würden	Pfade, wo Fehler keine oder nur geringe Auswirkungen auf das System haben
Teile mit neuer Technologie, Frameworks, etc.	Teile, die ähnlich zu bereits zuvor positiv begutachteten Teilen sind
Teile, die von unerfahrenen Teammitgliedern entwickelt worden	Wiederverwendete oder redundante Teile

Ziele



http://blakesmith.me/2015/02/09/code-review-essentials-for-software-teams.html

Teammitglieder zusammen halten (gleiches Verständnis über Aufgaben und Code)

Gute Pull Requests definieren:
- Richtiges Item für jetzige Situation?
- Zustimmung des Teams, ob das der richtige Change ist?
- Wie kann ich die Änderung in kleine Teile zerlegen, die einfach zu reviewen sind?
- Wie kann ich die Änderungen Testen und auf Korrektheit überprüfen?

Gemeinsames mentales Modell hilft:
- Risiken zu vermeiden
- Änderungen schneller nachzuvollziehen
- Besseres Softwaredesign zu entwickeln

Pull Request: Schlechtes Beispiele

Title: Fix uninitialized memory bug
Description:
This is the bug Bob and I talked about earlier. I had trouble with the compiler but managed to make this work. Let me know what you guys think.

Kein Kontext, unklarer Titel

Wo ist der Bug?

Wie kritisch ist die Änderung?

Was war dieser Bug von Bob?

Welche Probleme gab es mit dem Compiler?

blakesmith.me/2015/02/09/code-review-essentials-for-software-teams.html

Pull Requests: Gutes Beispiele

Title: Fix process crash on startup from uninitialized memory [#54633]
Description: This bug was causing process crashes on boot due to a memory initialization error in our statistics Counter class. I talked this over with Bob, and we both agree the crashes are a rare edge case that don't warrant a hot-fix release. Here's a summary of the changes:
- Moved the underlying int variable into the class initializer to prevent ununitialized memory in the Counter.
- Reworked the Counter interface to simplify caller conditional logic and prevent further off-by-one counting problems.
- Added a unit test that exposes the crash
Testing:
I've verified the test suite still passes, and verified manually that the crash doesn't happen locally.

Titel mit Link zu Subsystem (z.B. Bug Repo, Issue Tracker, Commit Message, etc.)
Klarer Titel -> Prozess Crash durch nicht-initialisierten Speicher
Beschreibung gibt Auskunft, wo der Bug und unter welchen Bedingungen auftrat
Kritikalität ist beschrieben
Zusammenfassung der Änderung als strukturierte Liste (mentales Bild + Erwartungen vor Code Review)
Testing gibt dem Reviewer vertrauen, dass die Änderungen durchdacht und überprüft sind.

blakesmith.me/2015/02/09/code-review-essentials-for-software-teams.html

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 Teamleiterin

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

Aufgaben der Gutachterin

- Vorbereitung durch Begutachtung des Materials und aktive Teilnahme
- Professionelles Verhalten: Berechtigte positive und negative Kommentare. Gut: Fragen als Feedback

“This design is broken.”

Warum? Wie kann es verbessert werden?

“I don’t like this change.”

Warum? Was würde dir besser gefallen?

“Can you rewrite this to be more clear?”

Was ist das Problem hier? Wie sollte ich es umschreiben? Was ist unklar?

“How does this code handle negative integers?”

“This section is confusing to me, I don’t understand why class A is talking to class B”

“It looks like you broke an interface boundary here. How will that affect the user?”

Spezifisches Feedback damit der Entwickler über die Auswirkung der Änderung nachdenken kann (auch wenn der Reviewerin der Crash her schon klar ist). Testing Gap? Issues wie unten könnten auch beabsichtigt sein. Gebt den Entwicklern die Möglichkeit der Begründung.

blakesmith.me/2015/02/09/code-review-essentials-for-software-teams.html

Bericht

- Zusammenfassung
- Gefundene Probleme
- Empfehlung

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

Wann Review?

- Vor dem Commit oder danach?

Pre-Commit Review

- Stellt sicher, dass Code Guidelines und Qualitätsstandards eingehalten werden
- Hilft, dass eigenes Review auch durchgeführt und nicht auf andere Gutachter abgeschoben wird (siehe Comic)
- Verhindert, dass andere Entwicklerinnen durch selbst eingeführte Bugs leiden

Aber:

- Reduziert Produktivität, da nicht weiter am Code gearbeitet werden kann, bis Review erfolgt ist
- Nach dem Review könnte die Entwicklerin aber eine geänderte (nicht-gereviewte) Version (versehentlich/absichtlich) Commiten

Post-Commit Review

- Entwicklerin kann Änderung commiten und weiter arbeiten
- Andere Entwicklerinnen sehen frühzeitig die Änderung und können ihre eigene Arbeit daran anpassen
- Bei komplexen Änderungen kann dies in mehrere Teile zerlegt und diese individuell gereviewt werden

Aber:

- Erhöhte Chance, dass schlechter Code ins Repository geht und somit das gesamte Team betrifft
- Falls Probleme bei Review auftreten, dann braucht es evtl. eine Zeit, bis die Entwicklerin wieder an dem betroffenen Modul arbeitet

Best Practices

- Pre-Commit für kritischen Code (major changes)
- Post-Commit für triviale Änderungen

- Commit früh und oft, um viele kleine Reviews zu ermöglichen
- Entwicklerinnen sollten Commit gut dokumentieren (z.B. gute Commit Message / Pull Request)
- Commit Code zu einem Feature / Test Branch
- Track Bugs, um sicherzustellen, dass sie auch tatsächlich gefixt werden
- Review den Code und nicht die Entwicklerin

Tools

- Review Ninja (http://review.ninja)
- Gerrit (https://code.google.com/p/gerrit/)
- FishEye(https://www.atlassian.com/software/fisheye/overview)

- SimpleCov(code coverage, https://github.com/colszowka/simplecov)
- Flog (code complexity,http://ruby.sadi.st/Flog.html)
- Reek (code smells,https://github.com/troessner/reek)
- Cane (code quality, https://github.com/square/cane)
- Rails_best_practices(Rails specific, https://github.com/flyerhzm/rails_best_practices)

List by HPI: Software Engineering II

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