Software Test

- Motivation
- Testklassifikation
- Black Box Testtechniken
- White Box Testtechniken
- Testmetriken
- Grenzen des Software Tests

Testmetriken

Das Projekt hat Testverfahren und Testfälle definiert. Jetzt stellt sich die Frage:

Wie gut machen wir das? Wird ein Modul ausreichend getestet?

➔Quantifizierung durch Testmetriken

(Beurteilung von Testverfahren:

- Wie viele unentdeckte Fehler enthält das Programm?
- Wie leistungsfähig ist ein gegebenes Testverfahren?)

Überdeckungsmetriken

Im industriellen Bereich im Wesentlichen relevant:

- **Anweisungsüberdeckung**

M_{C0}=(Anzahl der überdeckten Knoten)/(Anzahl der Knoten) * 100 [%]

- **Zweigüberdeckung**

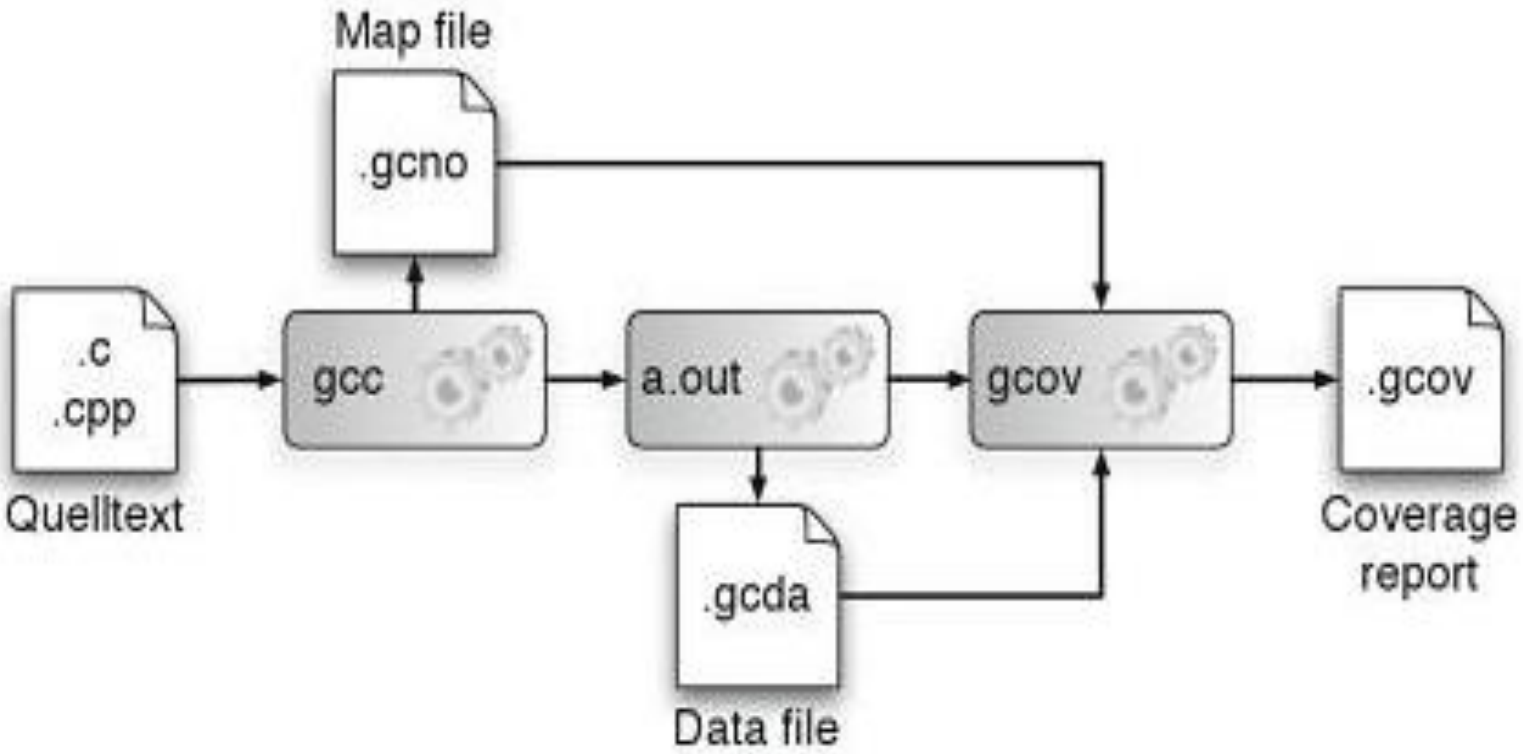
M_{C1}=(Anzahl der überdeckten Kanten)/(Anzahl der Kanten) * 100 [%]

Überdeckungsmetriken

Verwendung:

- Abnahme für Module: Abnahme erst bei Testüberdeckung größer als vereinbarte Schwelle.
- Vergleich von Modulen, um Schwächen in der Testumgebung aufzudecken und Testressourcen zielgerichtet zu planen.

Überdeckungsmetrik – Bsp gcov



Quelle des Bilds: D. Hoffmann, Software Qualität, 2. Auflage

Toolunterstützung bei Überdeckungsmetriken

Z.B. der Profiler gcov (Teil der GNU Compiler Collection gcc), https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

1. Kompilieren:

```
gcc -Wall -fprofile-arcs -ftest-coverage
-omanhattan manhattan.c
```

Die Optionen erzeugen zusätzliche output files, die nach den Testläufen durch den gcov befehl ausgewertet werden.

Gcov

- Ausführen:

```
D:\0_OTH\temp>manhattan 1 1
Returnvalue is 2

D:\0_OTH\temp>manhattan 2 2
Returnvalue is 4

D:\0_OTH\temp>manhattan 1 -1
Returnvalue is 2
```

Gcov

- **Analysieren**

gcov manhattan.c

➔

```
D:\0_OTH\temp>gcov manhattan.c
File 'manhattan.c'
Lines executed:86.67% of 15
Creating 'manhattan.c.gcov'
```

Gcov

- Analysieren:

Datei *manhattan.c.gcov* mit detaillierter Überdeckungsinformation der Testläufe.

Gcov ermittelt die **Zeilenüberdeckung, nicht die Anweisungsüberdeckung !**

➔ Vorsicht bei Kompilieren mit Optimierung!

Analyse

Ausschnitt aus manhattan.c.gcov

```
    3:    4:int manhattan(int a, int b){
    -:    5:
    3:    6:    if(a<0){
#####:    7:        a=-a;
    -:    8:    }
    3:    9:    if(b<0){
    1:   10:        b=-b;
    -:   11:    }
    -:   12:
    3:   13:    return a+b;
    -:   14:}
    -:   15:
    3:   16:int main(int argc,char **argv){
    3:   17:    int ret =0;
```

Nicht durchlaufen

1 mal durchlaufen

3 mal durchlaufen

Code Coverage Tools

- Gcov: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov
- Gcovr: setzt auf gcov auf und erlaubt neben vielen anderen Funktionen auch Zweigstatistiken: https://gcovr.com/en/stable/guide.html#
- Code Coverage Tools im Java Umfeld:
  - http://c2.com/cgi/wiki?CodeCoverageTools
  - http://en.wikipedia.org/wiki/Java_Code_Coverage_Tools

LCOV

Graphisches front-end für gcov.

Bsp eines einfachen Code Coverage resports (quelle: http://ltp.sourceforge.net/coverage/lcov/output/index.html)

LCOV - code coverage report

				Hit	Total	Coverage
Current view:	top level		**Lines:**	20	22	90.9 %
Test:	Basic example (view descriptions)		**Functions:**	3	3	100.0 %
Date:	2015-10-08 10:24:18		**Branches:**	8	10	80.0 %
Legend:	Rating:	low: < 75 % / medium: >= 75 % / high: >= 90 %				

Directory	Line Coverage		Functions		Branches	
example	90.0 %	9 / 10	100.0 %	1 / 1	75.0 %	3 / 4
example/methods	91.7 %	11 / 12	100.0 %	2 / 2	83.3 %	5 / 6

Generated by: LCOV version 1.12

Coverage Tools im Java Umfeld

- Code Coverage in Eclipse: ECLEmma: https://www.jacoco.org/index.html

```
1  package demo;
2
3  public class Main {
4
5      public static void main(String[] args) {
6          try {
7              System.out.println("OK");
8              int var = 1;
9              if( var == 1) {
10                 System.out.println("var ist 1");
11             }else {
12                 System.out.println("var ist nicht 1");
13             }
14         }catch(Exception e) {
15             System.out.println("Exception");
16         }
17         return;
18     }
19 }
20
21
```

Problems Javadoc Declaration Console Coverage

Main (27.05.2020 08:42:06)

Element	Coverage	Covered Branches	Missed Branch...	Total Branches
testprojekt	50,0 %	1	1	2
src	50,0 %	1	1	2
demo	50,0 %	1	1	2
Main.java	50,0 %	1	1	2
Main	50,0 %	1	1	2
main(String[])	50,0 %	1	1	2

Coverage Tools im Java Umfeld

- Integration in andere Tools (z.B. maven):

  JaCoCo: https://www.jacoco.org/jacoco/

Beispielprojekt auf der eLearning plattform.

- Aufruf mvn clean test

  ➔ Überdeckungsreport unter target/site/jacoco/index

- Aufruf von mvn install ➔ build failure, weil die konfigurierte Überdeckungsschwelle nicht erreicht wird.

Mutationstest

Bisher: Messen der Zeilen/Statement/Zweig Überdeckung.

Damit können Sie etwas über den Fleiß der Testentwickler aussagen, aber nicht wirklich über die Qualität der Tests.

Oder, um es mit Pedro Rijo zu sagen:

„Coverage sucks“

Methode, um die Qualität von Tests zu prüfen: **Mutationstests**

Mutationstest

Technik, um Testverfahren zu beurteilen.

Warum sind Mutationstests sinnvoll?

- ➔Die klassischen Metriken um Tests zu beurteilen, messen Zeilen-, Anweisungs- oder Zweigüberdeckungen.
- ➔Damit ist nichts darüber ausgesagt, ob Fehler auch gefunden werden. (Bsp. Tests, die alles überdecken, aber keine asserts verwenden)
- ➔Prüfung, ob Fehler gefunden werden, indem Fehler künstlich in den Code eingeführt werden.

Mutationstests

- Alte Idee
- Hoher Rechenaufwand
- mit zunehmender Leistung der Computer
- Und zunehmenden Qualitätsanforderungen wieder in den Fokus gerückt.

Unterstützt durch frameworks z.B. für Java:

- µJava (https://cs.gmu.edu/~offutt/mujava/)
- The Major mutation framework (https://mutation-testing.org/)
- PIT (https://pitest.org/)

PIT als Beispiel

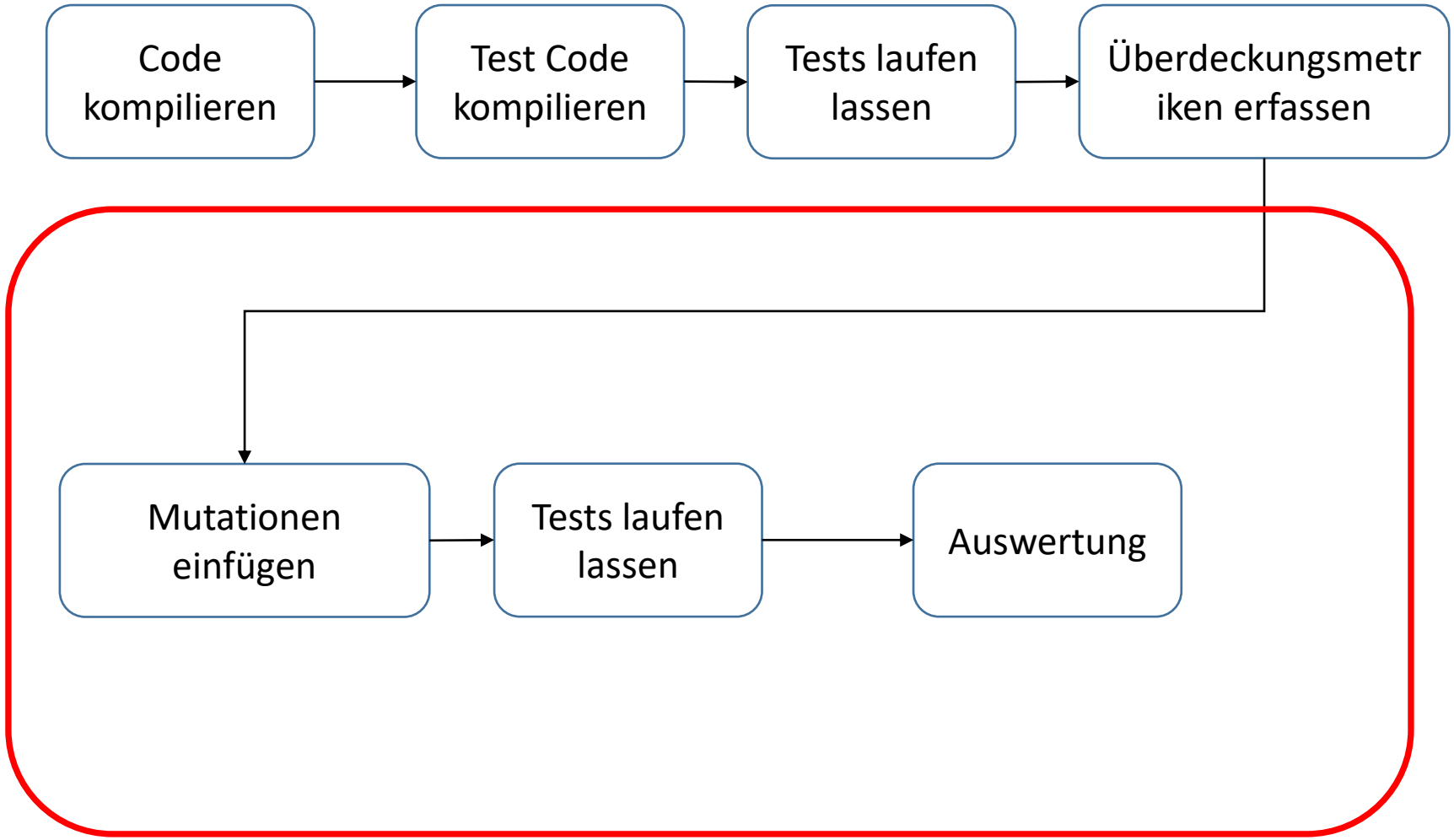
pitest.org

Real world mutation testing

PIT is a state of the art mutation testing system, providing gold standard test coverage for Java and the jvm. It's fast, scalable and integrates with modern test and build tooling.

Klingt doch gut.

PIT – Prinzip


Code kompilieren
Test Code kompilieren
Tests laufen lassen
Überdeckungsmetriken erfassen
Mutationen einfügen
Tests laufen lassen
Auswertung

PIT als Beispiel

- Quickstart mit maven, ant, cmd, gradle
- Zusätzliche Konfigurationsoptionen, z.B.
  - Packages, die analysiert werden sollen
  - Nur neue oder modifizierte Klassen.
  - Einschränkung der Tests
  - Anzahl der Threads
  - Zu verwendende Mutatoren
  - …

PIT – Mutatoren - Bsp

Siehe https://pitest.org/quickstart/basic_concepts/

CONDITIONALS_BOUNDARY_MUTATOR

Original Code

```
if ( i >= 0 ) {
    return "foo";
} else {
    return "bar";
}
```

Wird mutiert zu

Mutierter Code

```
if ( i > 0 ) {
    return "foo";
} else {
    return "bar";
}
```

PIT Mutationen

Siehe https://pitest.org/quickstart/mutators/

- Ca. 30 Mutationen
- Die Mutationen beziehen sich auf Aspekte der funktionalen Programmierung.
- Aspekte der Objektorientierung werden nicht berücksichtigt.

PIT – zusätzliche Aspekte

Plugin Konzept (aktuell instabil) für

- Mutation Result Listener
- Mutation Filter
- Mutation interceptor
- Test Prioritizer

https://pitest.org/quickstart/advanced/

Literatur zu Mutationstests und PIT

- Johannes Dienst, **Mutationstests mit PIT in Java,** 14.11.2017 https://m.heise.de/developer/artikel/Mutationstests-mit-PIT-in-Java-3888683.html?seite=all
- Kevin Wittek: **Mutation Testing: Attack of the Java Mutants,** 24. Juli 2019 https://jaxenter.de/mutant-testing-pit-java-84437
- https://www.baeldung.com/java-mutation-testing-with-pitest
- Pedro Rijo: **An intro to Mutation Testing - or why coverage sucks,** February 14, 2019 https://pedrorijo.com/blog/intro-mutation/

Software Test

- Motivation
- Testklassifikation
- Black Box Testtechniken
- White Box Testtechniken
- Testmetriken
- Grenzen des Software Tests
- Testautomatisierung

Grenzen des Software Tests

Software Tests sind oft schwierig und aufwändig.

Gründe

- Unklare oder fehlende Anforderungen
- Programmkomplexität
- Mangelnde Werkzeugunterstützung bei der Konstruktion
- Ausbildungs- und Fortbildungsdefizite
- Zeitprobleme

Schwierigkeit beim Test - Bsp

Kein Problem

```
float foo(unsigned char x)
{
    unsigned char i = 0;
     while (i<6){
        i++;
        x /=2;
     }
     return 1.0/(x-i);
}
```

Division durch 0
Bei bestimmten Eingabewerten

```
float foo(unsigned short x)
{
    unsigned char i = 0;
     while (i<6){
        i++;
        x /= 2;
     }
     return 1.0/(x-i);
}
```