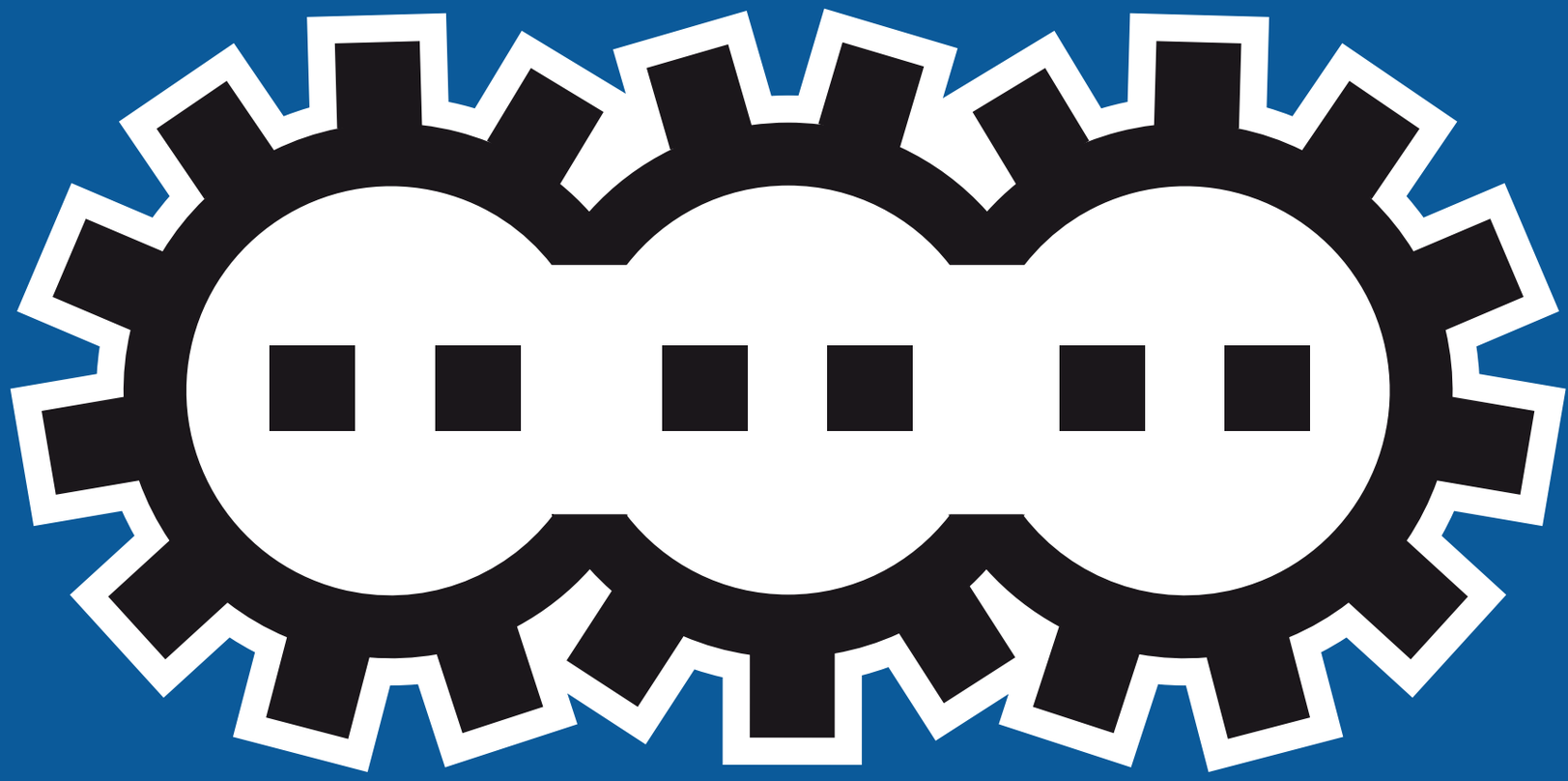
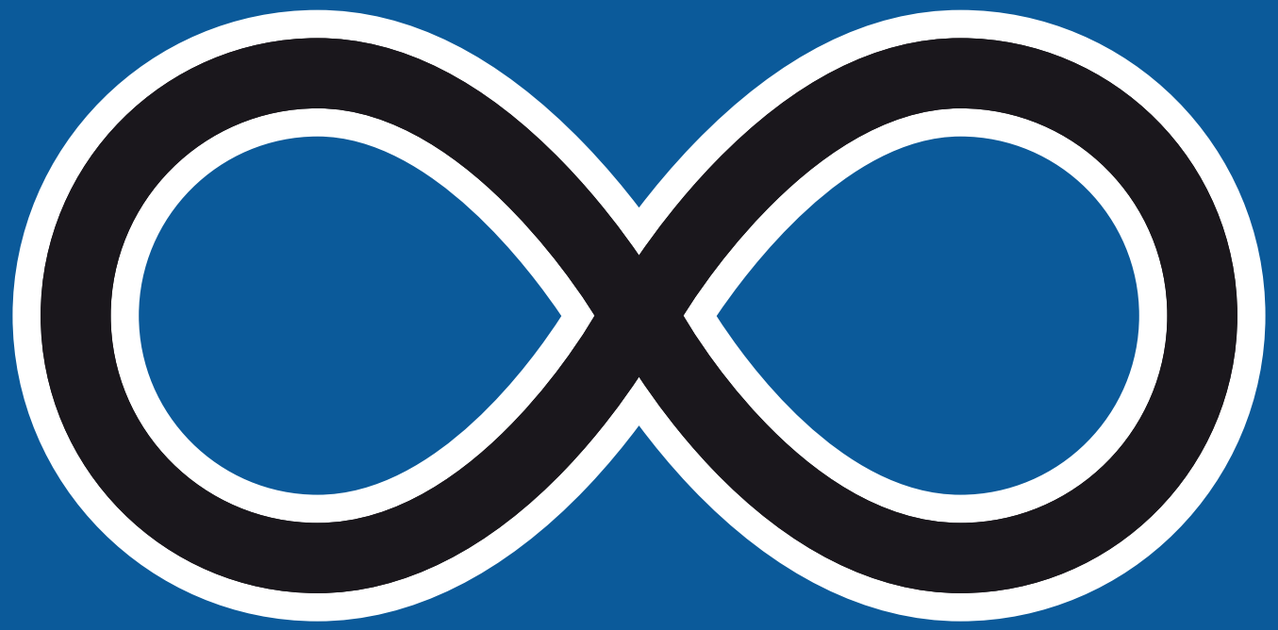


SOFTWAREENTWICKLUNG

IM TEAM MIT OPEN-SOURCE-WERKZEUGEN

06 - Softwaretests

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen



Wiederholung

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Maven Lifecycle

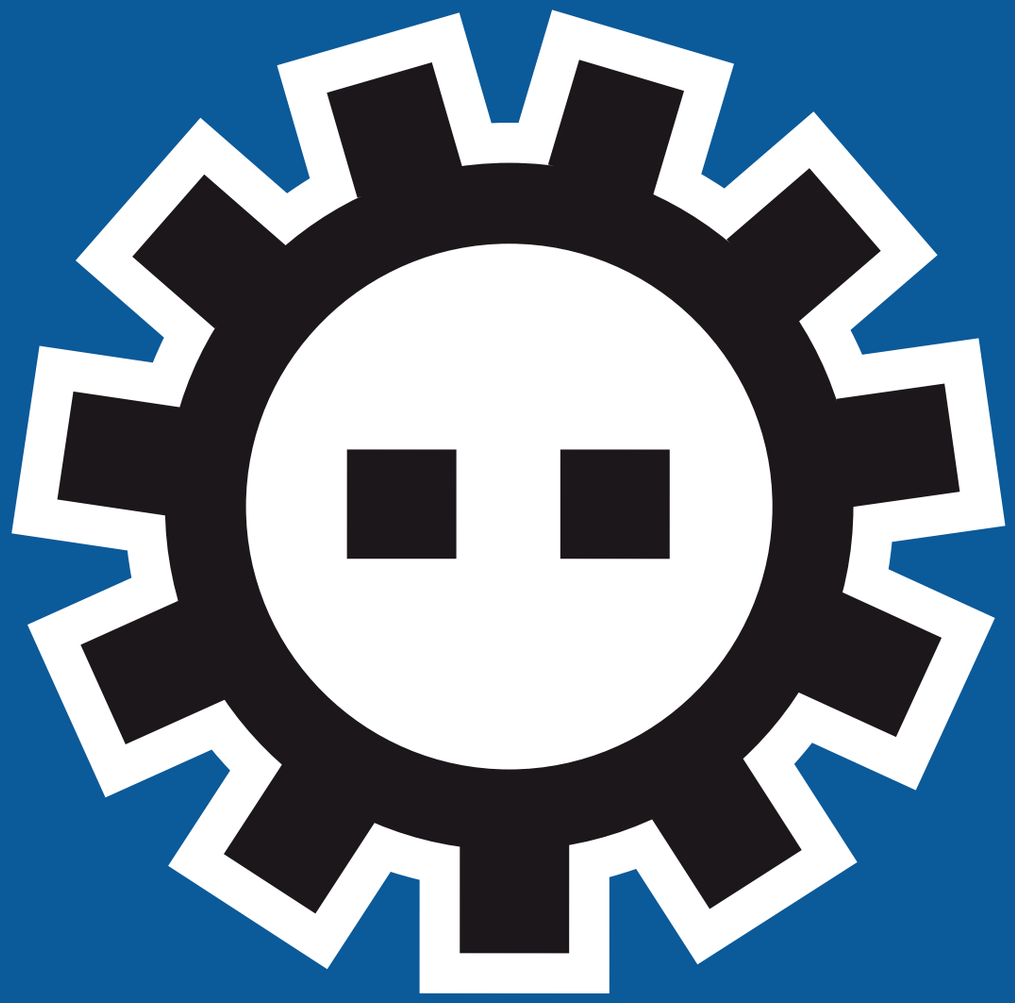
- Zentrales Konzept von Maven:
  - die Ausführung von vordefinierten Arbeitsabläufen (Lifecycle)
- Jeder Lifecycle besteht aus sequentiellen Phasen
- In die Phasen können sich Plugins einhängen, die Operationen durchführen
  - Standardplugins sind vordefiniert und müssen höchstens konfiguriert werden
    - » z.B. Java-Dateien kompilieren, Jar-Bibliothek erstellen
  - weitere Plugins können bei Bedarf in den Lifecycle eingehängt werden
- Nicht immer muss der komplette Lifecycle durchlaufen werden
  - z.B.: `mvn compile` → alle Phasen bis *compile* werden durchlaufen

Default Lifecycle (Auswahl)

Default Lifecycle	
Phase	Beschreibung
`validate`	überprüft, ob die Projektkonfiguration korrekt ist
`generate-sources`	Erzeugung von Quelldateien aus Metadaten, z.B. aus Annotationen
`process-resources`	Kopiert und ersetzt Variablen in Konfigurationsdateien
`compile`	Kompilierung der Quelldateien
`test-compile`	Kompilierung der Test-Quelldateien
`test`	Ausführung der Unit-Tests
`package`	Erstellung eines Programmpakets, z.B. eines WAR für eine Web-Anwendung
`install`	legt das erzeugte Paket im lokalen Maven-Repository ab
`deploy`	legt das erzeugte Paket im entfernten Maven-Repository ab

Ausblick

- Maven dient als Fundament, um die nächsten Schritte umzusetzen:
- Software-Tests mit Junit
  - innerhalb des Maven-Lifecycles, in der Phase *test*
- Continuous Integration
  - zentraler Server führt den Maven-Lifecycle aus
- Repository Manager
  - eigener Maven-Repository-Server
- Messung der Software-Qualität
  - Reporting-Plugins messen und dokumentieren Software-Qualität



Motivation

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Fallbeispiel F-16

- 1978: Entwicklung des F-16 Kampfflugzeugs
- Steuersoftware des Autopilots wurde aus einem Raketenprogramm übernommen
- Raketensteuerung wurde aus Platzgründen optimiert:
  - beim Überflug über den Äquator folgt sie den Koordinaten mit umgekehrten Vorzeichen
  - Rakete dreht sich beim Äquatorüberflug auf den Kopf
- Folge:
  - F-16 dreht sich beim Äquatorüberflug auf den Kopf

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Fallbeispiel Ariane 5

- 1996: Entwicklung und Test der Ariane 5 Rakete
- Über 500 Millionen US-Dollar Verlust
- Steuersoftware wurde aus der (langsameren) Ariane 4 übernommen
- Geschwindigkeit wurde in eine interne Einheit (16-bit signed integer) umgerechnet
  - bei einer Geschwindigkeit von über 32768 führte die Konvertierung zu einem Überlauf
- Folgefehler führten zum Verlust der Kontrolle und der automatischen Selbstzerstörung

Fallbeispiel ATC

- 14. September 2004: LA Air Traffic Control
- Funkmodul zur Kommunikation schaltet sich unerwartet ab
  - 400 anfliegende Flugzeuge können nicht mehr geleitet werden
- Notlösung: Fluglotsen telefonierten per Handy mit Piloten
- Erkanntes Problem:
  - interner Timer hat einen Wertebereich von 2^{32}ms
  - bei einem Überlauf schaltet sich das System ab
  - 2^{32}ms entsprechen ungefähr 50 Tage
- Lösung: alle 30 Tage wird das System neu gestartet…

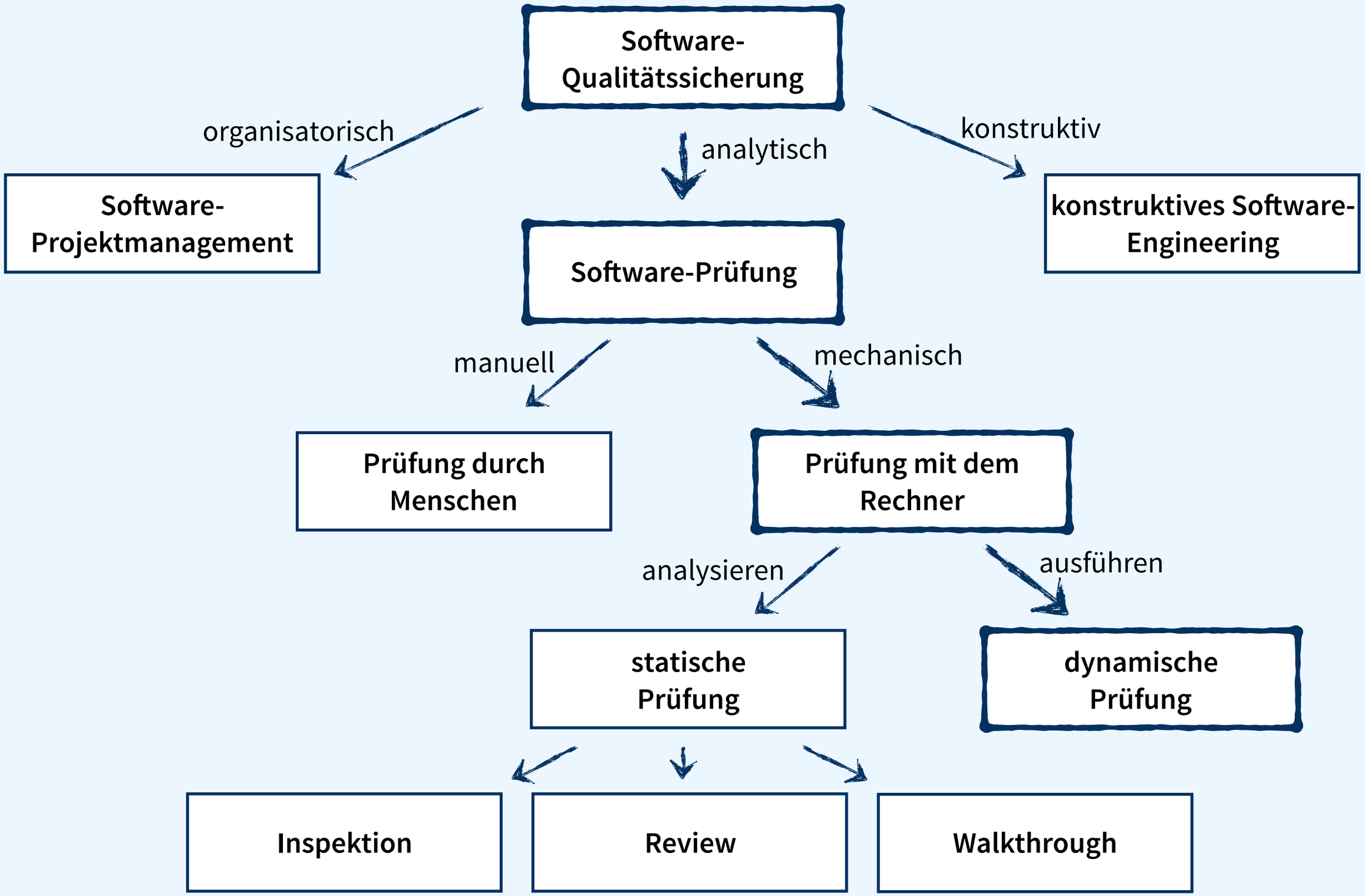
Quelle: Linda Geppert, IEEE spectrum

weitere Fallbeispiele

- 1962: Die NASA verlor ihre Venus-Sonde Mariner 1, und damit 80 Millionen US-Dollar, aufgrund eines fehlenden Bindestrichs im Programmcode.
- 1971: Bei einem meteorologischen Experiment mit 141 Wetterballons rund um die Erde und dem 1. französischen Anwendungssatelliten explodieren 72 Ballons. Die Datenübertragung zum Satelliten der Ballons konnte zwei Kommandos empfangen: "Sende Daten" und "Zerstöre Dich selbst" (Notfall). Der Satellit wünschte Datenempfang die Ballons erhielten aber das Kommando zur Zerstörung.
- Im September 1994 wurde in Bayreuth drei Parksündern wegen der Verwechslung einer Codenummer eine falsche Anzeige geschickt. Die Anzeige lautete „Vorbereitung eines Angriffskrieg“.

Quelle: certitudo GmbH

Software-Qualitätssicherung



Quelle: J. Ludewig, H. Lichter. Software Engineering, 2010

Programmtest

- Wir müssen die Fehlerfreiheit des fertigen Software-Systems nachweisen
- Aufgrund des erforderlichen Aufwands sind wir dazu in der Praxis nicht in der Lage
- Unser Ziel ist es daher, möglichst viele der vorhandenen Fehler durch Programmtests zu finden

➡ **Testen** ist die Ausführung eines Programms auf einem Rechner mit dem Ziel, Fehler zu finden (Myers 1979).

Programmtest

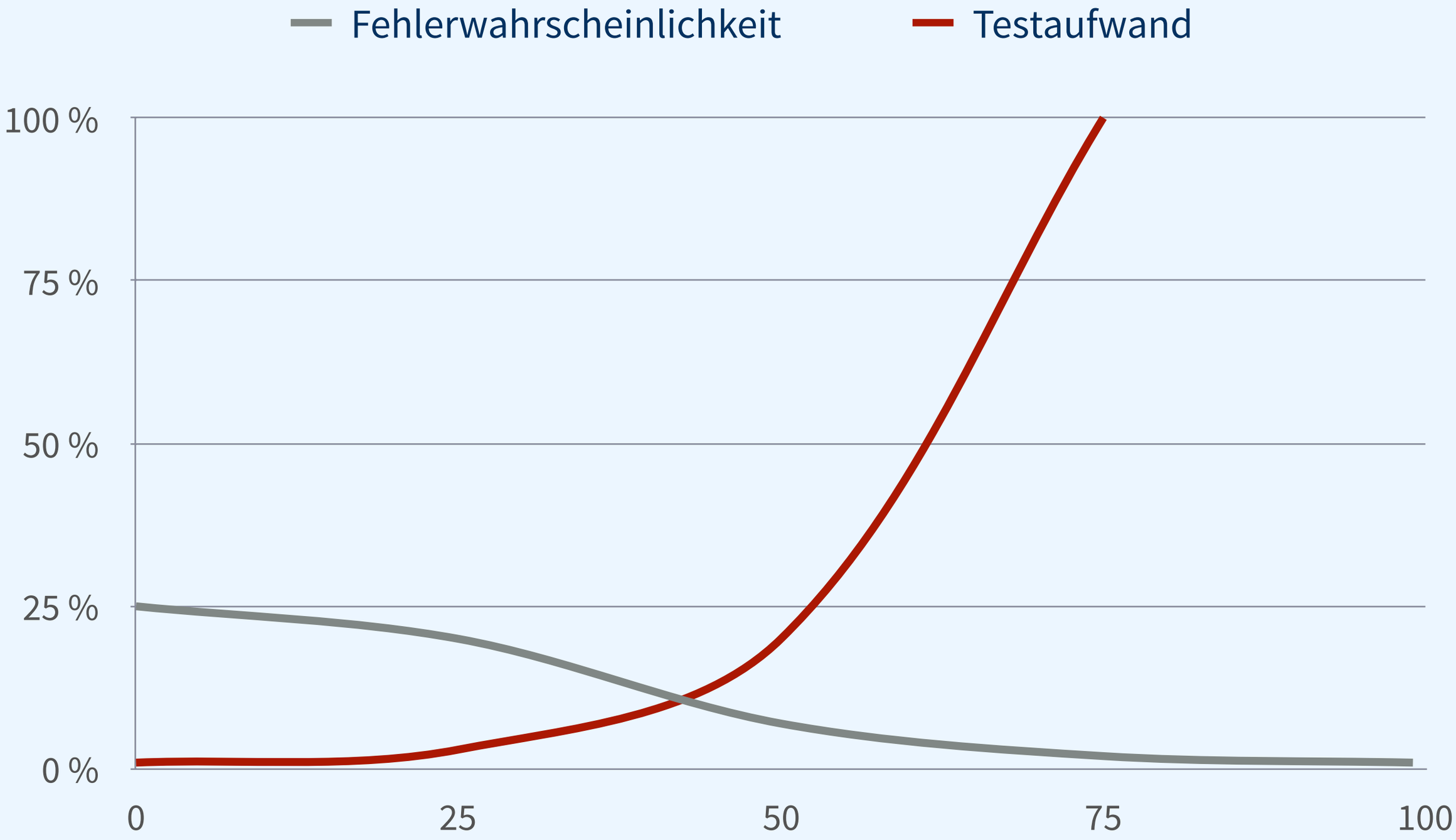
- Das übersetzte Programm wird dazu ausgeführt und es werden konkrete Werte eingegeben
- Die Ausgaben (Ist-Werte) werden mit den Soll-Werten (aus der Spezifikation abgeleitet) verglichen
- Bei einer Abweichung zwischen Ist- und Soll-Wert liegt ein Fehler vor
- Ein spontanes Ausprobieren von Eingaben ist weder effizient noch sonderlich effektiv

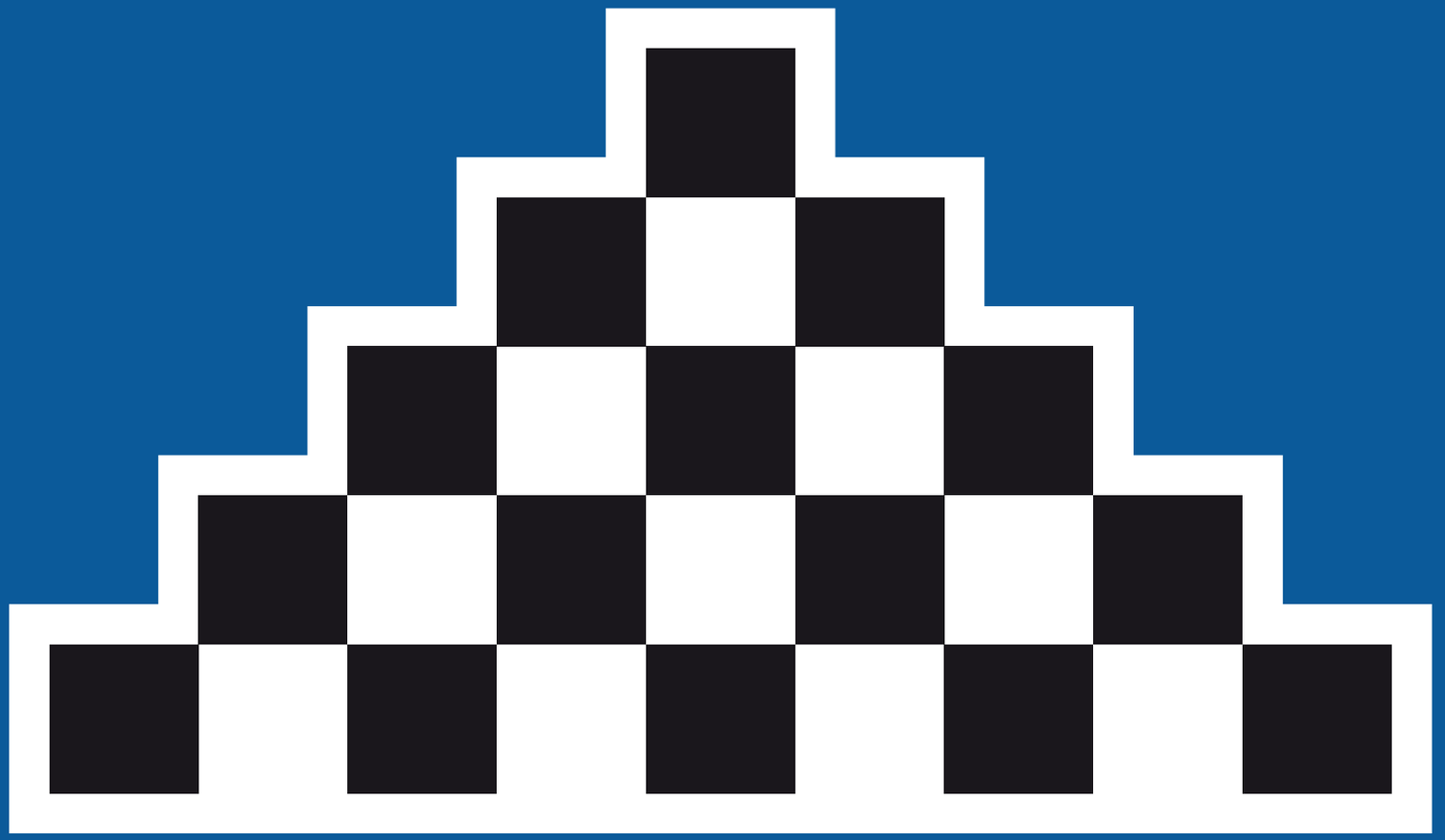
schlechte Tests

- Ausführen und Ausprobieren eines Programms
- Untersuchung eines Programms im Debugger
- Manuelle Durchsicht des Programmcodes und "error guessing"
- Analyse eines Programms durch Software-Werkzeuge
  (z.B. die Erhebung von Metriken)

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

Testaufwand





Grundlagen

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

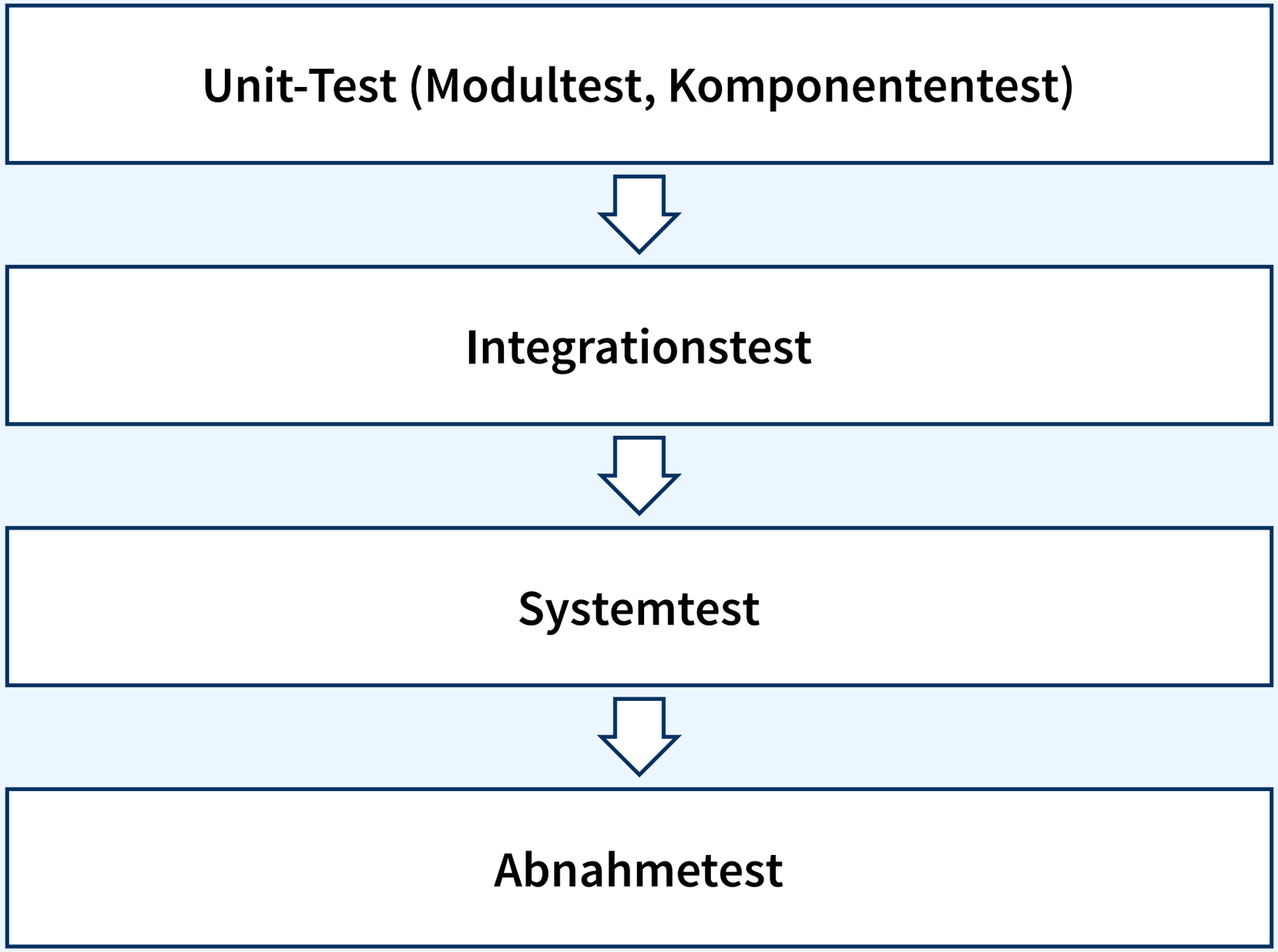
systematische Tests

- Randbedingungen (z.B. Systemumgebung) sind definiert
- Eingaben werden systematisch ausgewählt
- Soll-Werte werden vor dem Test festgelegt
- Der Testverlauf wird dokumentiert
- Test und Korrektur finden getrennt statt

➡ Ein *systematischer Test* ist reproduzierbar (vorausgesetzt, der Startzustand ist reproduzierbar). Damit ist der Test *objektiv* und *wiederholbar*.

Prüfebenen

Tests lassen sich nach der Prüfebene in verschiedene *Teststufen* einteilen:



Unit-Test

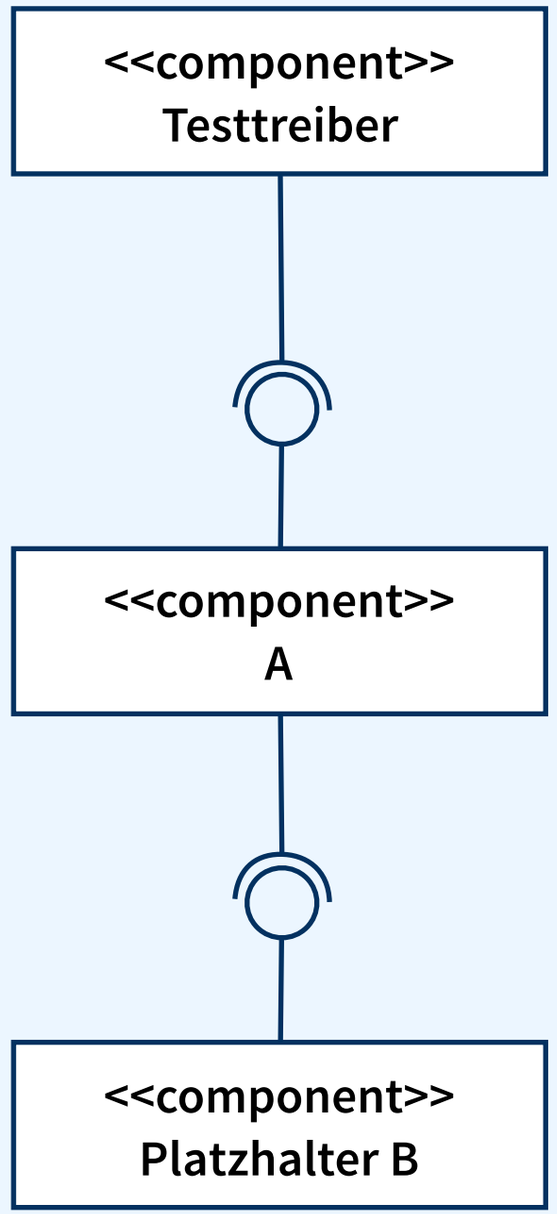
- Es werden Programmteile getestet, die eine ausreichende Größe für einen eigenständigen Test haben:
  - Funktionen/Methoden
  - Klassen
  - Komponenten / Verbund von Klassen
- Wird in der Implementierung durchgeführt
- Ein Testtreiber versorgt das Modul über Aufrufe der Schnittstelle mit Testdaten und nimmt die Antwort des Moduls entgegen
- Im Mittelpunkt steht der funktionale Test
- Optional
  - Test auf Robustheit
  - Test auf Effizienz

Integrationstest

- Es wird getestet, ob das Zusammenspiel der integrierten Module / Komponenten funktioniert
- An den Schnittstellen können verschiedene Fehler auftreten:
  - inkompatible Schnittstellenformate
  - Protokollfehler
  - semantische Fehler (unterschiedliche Einheiten)
  - Timing-Probleme
  - Kapazitäts- und Lastprobleme

Integrationstest - Aufbau

- Wenn Komponente A getestet werden soll, wird ggf. ein *Testtreiber* benötigt
  - ▸ versorgt die Schnittstelle der Komponente A mit Testdaten
- Falls die Komponente A Dienste einer Komponente B nutzt, die noch nicht integriert ist, wird ein *Platzhalter* (*stub*) für B benötigt
  - ▸ vertritt die fehlende Komponente und liefert entweder
    - » konstante Werte, oder
    - » simuliert das Verhalten der späteren Komponente in Ausschnitten



Systemtest

- Test des *komplett integrierten* Systems
- Test aus Sicht des Anwenders
- Validation, ob die Anforderungen vollständig und angemessen umgesetzt wurden
- Testumgebung sollte der späteren Produktivumgebung möglichst ähnlich sein
- Testtreiber und Platzhalter sind durch reale Komponenten ersetzt

Abnahmetest

- Erfolgt als letzter Test vor der Inbetriebnahme
- Die Abnahmekriterien sind im Entwicklungsvertrag festgeschrieben (Individualsoftware)
- Kunde ist in die Festlegung der Akzeptanztests involviert
- Test aus Anwendersicht in der Abnahmeumgebung des Kunden

Konstruktion von Testfällen

Grenzen von Programmtests

Falls es für ein Programm n mögliche Eingaben gibt, und wir alle n Eingaben testen, dann haben wir einen vollständigen Test durchgeführt

- Treten bei einem vollständigen Test keine Fehler auf, dann ist die Korrektheit des Programms nachgewiesen
- **Beispiel:**
  - Wir wollen einen vollständigen Test für die Methode `public static long abs(long a)` der Klasse `java.lang.Math` durchführen
  - Da eine long-Variable eine Breite von 64-Bit hat, gibt es 2^{64} Testfälle
  - Falls ein Test nur eine Mikrosekunde dauern würde, müssten wir über 500000 Jahre auf den Nachweis der Korrektheit warten!

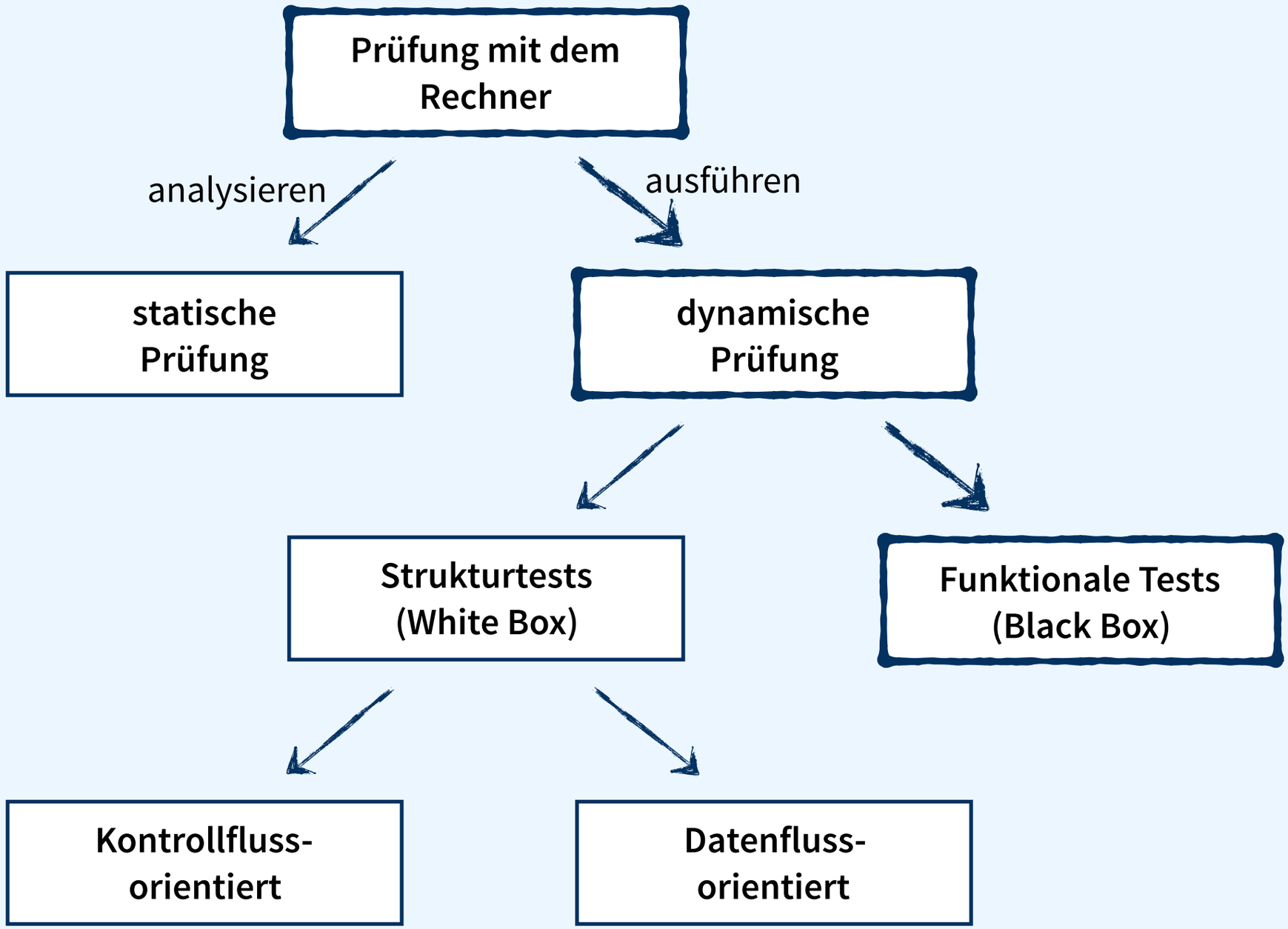
Testfälle

- Wir müssen also mit einer kleinen Teilmenge von Eingaben auskommen
- Um mit den wenigen Eingaben trotzdem viele Fehler finden zu können, müssen wir die Eingaben geeignet wählen
- Die Festlegung der Eingabedaten reicht für einen systematischen Test nicht aus
- Ein *Testfall* beschreibt:
  - die Ausgangssituation (Vorbedingungen, Randbedingungen)
  - Testdaten, die die vollständige Ausführung des Testobjekts bewirken
  - Sollwerte (das erwartete Ergebnis bzw. Verhalten)

Testfälle

- Die Sollwerte eines Testfalls werden aus der Spezifikation abgeleitet
- Oft können Testszenarien als Folge von Testfällen gebildet werden
- Ein guter Testfall sollte die folgenden Eigenschaften besitzen:
  - *fehlersensitiv*
    (er zeigt mit hoher Wahrscheinlichkeit einen Fehler an)
  - *repräsentativ*
    (er steht stellvertretend für viele andere Testfälle)
  - *redundanzarm*
    (er überdeckt keine anderen Testfälle)

Klassifizierung der Testverfahren



Quelle: Balzert, Lehrbuch der Softwaretechnik

Black-Box-Tests

Werden auf Basis der in der Spezifikation geforderten Eigenschaften gewählt, die innere Beschaffenheit des Programms spielt keine Rolle.

- Der Black-Box-Test ist die wichtigste Form des dynamischen Tests
- Die Testfälle können/sollen bereits aufgestellt werden, sobald die entsprechende Spezifikation vorliegt
- Auswahl an Verfahren zur Bestimmung von Testfällen:
  - Funktionale Äquivalenzklassenbildung
  - Grenzwertanalyse
  - Test spezieller Werte
  - Zufallstest

Äquivalenzklassenbildung

Angenommen, wir haben eine Menge E={e_1, e_2, …, e_n} von Eingaben, für die sich das Testobjekt gleich verhält

- Dann reicht es aus, nur eine Eingabe aus der Menge zu testen (z.B. e_1)
- Wenn diese Eingabe fehlerfrei ist, dann kann man davon ausgehen, dass auch die anderen Eingaben korrekt funktionieren
- Auch Äquivalenzklassen mit ungültige Eingabewerten definieren!
- Wie verhält sich das Testobjekt mit ungültigen Eingaben?
  - angemessene Reaktion auf die ungültige Eingabe (Meldung)
  - ignorieren der ungültigen Eingabe
  - automatische Korrektur der ungültigen Eingabe
  - Fehlberechnung, Programmabbruch (unerwünscht)

Grenzwertanalyse

Falls die Elemente einer Äquivalenzklasse eine natürliche Ordnung besitzen, dann sollten Elemente aus der Äquivalenzklasse gewählt werden, die an den Grenzen liegen

- Die Erfahrung hat gezeigt, dass Eingaben, die die Grenzen von Äquivalenzklassen bilden, häufig Fehler aufdecken
- Die Annäherung an die Grenzen kann vom gültigen und vom ungültigen Bereich der Äquivalenzklasse aus erfolgen
- In einigen Situationen können Fehler nur gefunden werden, wenn die Grenzen mit drei Werten geprüft werden:
  - der exakte Grenzwert
  - der zum Grenzwert benachbarte Wert in der Äquivalenzklasse
  - der zum Grenzwert benachbarte Wert außerhalb

Test spezieller/zufälliger Werte

Bestimmte Eingaben, unabhängig von der konkreten Spezifikation, decken häufig Fehler auf:

- eine leere Eingabe
- Eingabe mit Sonder- und Steuerzeichen

Eingaben können aber auch zufällig generiert werden

- Die Datentypen der Eingaben müssen berücksichtigt werden
- Der Vorteil von Zufallstests ist, dass auch nicht naheliegende Eingaben überprüft werden
- Der Nachteil ist, dass Fehler nur zufällig gefunden werden

Testautomatisierung

- Der Vorteil systematischer Tests ist die Wiederholbarkeit der Tests
- Wiederholung eines Tests auf Grund von:
  - Programmänderungen (Fehlerkorrektur) → Fehlernachtest
  - Programmerweiterungen → Regressionstests
- Mit der Wiederholung von Tests können Modifikationen am Programm abgesichert werden
- Eine häufige Wiederholung erfordert eine Testautomation
- Die Testautomation kann auf Basis existierender Test-Frameworks und Werkzeugen erfolgen (Reduzierung des Aufwands)
- Als Beispiel betrachten wir das JUnit-Framework für Java- Programme

Werkzeuge

Prof. Dr. Stefan Betermieux | Fakultät Informatik | Hochschule Furtwangen

JUnit

- JUnit bietet Unterstützung bei der Erstellung und der Durchführung von Programmtests
  - Entlastung von Routinetätigkeiten bei der Testfallerstellung
  - Automatisierung der Testdurchführung
  - Zählen und Berichten von Fehlern
- In der Praxis treten im Wesentlichen die Versionen JUnit 3 (Altsysteme) und JUnit 4 auf
- Unter JUnit 3 wird für jede zu testende Anwendungsklasse eine korrespondierende Testklasse geschrieben, die von der JUnit-Klasse `junit.framework.TestCase` abgeleitet ist
- Als Beispiel schreiben wir einen Testfall, um die Methode `toUpperCase()` der Klasse `String` zu testen

JUnit 3

```java
import junit.framework.TestCase;

public class StringTest extends TestCase {
  public void testStringtoUpperCase() {
    final String eingabe = "Das ist ein Testfall.";
    final String sollWert = "DAS IST EIN TESTFALL.";
    assertEquals(sollWert, eingabe.toUpperCase());
  }
}
```

- Jede Testmethode (Testfall) muss:
  - mit dem Präfix `test` beginnen
  - muss die Sichtbarkeit `public` haben
  - muss den Rückgabetyp `void` besitzen
- Empfehlung: Testklasse hat den Namen der Anwendungsklasse, ergänzt um das Postfix `Test`

JUnit assert-Methoden

Methode	Beschreibung
assertEquals(String message, Object a, Object b)	Prüft `a.equals(b)`
assertEquals(String message, int a, int b)	Prüft `(a==b)`
assertEquals(String message, double a, double b)	Prüft `abs(a-b)<=epsilon`
assertSame(String message, Object a, Object b)	Prüft `(a==b)`
assertTrue(String message, boolean b)	Prüft `(b == true)`
assertNotNull(String message, Object a)	Prüft `(a!=null)`
fail(String mesage)	Immer fehlschlagen

JUnit 4

- JUnit 4 modernisiert und vereinfacht die Testfallerstellung
  - Testklasse muss nicht von der Basisklasse `TestCase` abgeleitet werden (`assert`-Methoden aus globalem Import)
  - verwendet Annotationen, um Testfälle auszuzeichnen
  - Testmethoden benötigen kein Präfix `test` (trotzdem sinnvoll)

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


public class StringTest{
  @Test
  public void testStringtoUpperCase() {
    final String eingabe = "Das ist ein Testfall.";
    final String sollWert = "DAS IST EIN TESTFALL.";
    assertEquals(sollWert, eingabe.toUpperCase());
  }
}
```

JUnit 4 Annotationen

Annotation	Beschreibung
`@Test`	Kennzeichnet eine Methode (`public`, `void` und ohne Parameter) als Testfall
`@Before`	Die gekennzeichnete Methode wird vor jeder Testmethode aufgerufen
`@After`	Die gekennzeichnete Methode wird nach jeder Testmethode aufgerufen
`@BeforeClass`	Die gekennzeichnete statische Methode wird für eine Testklasse einmal ausgeführt, bevor die Testmethoden gestartet werden
`@AfterClass`	Die gekennzeichnete statische Methode wird für eine Testklasse einmal ausgeführt, nachdem die Testmethoden ausgeführt wurden

Erwartete Exceptions

- Auf ungültige Eingaben muss das zu testende Programm geeignet reagieren
  - z.B. mit einer Exception
- Wie können wir im Testfall eine Exception »erwarten«?
  - z.B. Division durch null wirft eine `ArithmeticException`

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

public class StringTest{
  @Test
 public void testDivisionDurchNull(){
    try{
      calculator.divide(2,0); // triviale Anwendungsklasse, dividiert a/b
      fail("Erwartete Ausnahme wurde nicht geworfen");
    } catch(ArithmeticException e){
    }
  }
}
```

Erwartete Exceptions

- Eleganter: Parameter `expected` zur `@Test`-Annotation erklärt eine Exception zum erwarteten Ergebnis
  - Test schlägt fehl, wenn die Exception nicht geworfen wird
- Weiterer Parameter timeout zur `@Test`-Annotation:
  - Test schlägt fehl, wenn der Test nicht innerhalb der angegebenen Zeitspanne (in ms) abgeschlossen ist

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

public class StringTest{
  @Test(expected=ArithmeticException.class, timeout=1000)
  public void testDivisionDurchNull() {
    calculator.divide(2,0);
  }
}
```

Parametrisierte Testfälle

- Häufig muss die gleiche Methode mit verschiedenen Parametern getestet werden → Äquivalenzklasssen
- JUnit 4 bietet parametrisierte Testfälle mittels eines speziellen TestRunners:
  - die Testklasse stellt die Testfalldaten in einer Sammlung zur Verfügung
  - die Testfalldaten enthalten den Soll-Wert und die Eingabewerte
  - ein spezieller Runner ruft dann für alle Testfalldaten aus der Sammlung die Test-Methode auf
  - für jeden Testfall wird eine neue Instanz der Testklasse erzeugt
  - der Runner übergibt die Testfalldaten dem Konstruktor der Testklasse

Parametrisierte Testfälle

```java
@RunWith(value = Parameterized.class)
public class CalculatorParameterizedTest {

  private int para1, para2, soll;

  public CalculatorParameterizedTest(int soll, int para1, int para2) {
    this.para1 = para1;
    this.para2 = para2;
    this.soll = soll;
  }

  @Parameters
  public static Collection<Integer[]> daten() {
    return Arrays.asList(new Integer[][] {
        {1,  2, 2},
        {2,  4, 2},
        {3,  7, 2},
        {5, 10, 2}
    });
  }

  @Test
  public void test() {
    Calculator cal = new Calculator();
    assertEquals(soll, cal.divide(para1, para2));
  }
}
```

Verzeichnis für Testklassen

- Kompilierte Testklassen (und Testframeworks) sollten später nicht ausgeliefert werden
  - separate Verzeichnisse für Anwendungs- und Testcode
  - Anwendungs- und Testcode sollten aber im gleichen Package liegen

- Maven unterstützt dies durch eine vorgegebene Verzeichnisstruktur:

projekt/	Hauptverzeichnis des Projekts
+-**src/**	Alle Quelldateien
| +-**main/**	Quelldateien für die Anwendung
| | +-**java/**	Java Quellcode
| | | +-Calculator.java	
| +-**test/**	Quelldateien für den Test der Anwendung
| | +-**java/**	Java Quellcode der Testklassen
| | +-CalculatorTest.java	
…	

Ausführen der Tests

- JUnit kann auf verschiedene Weisen ausgeführt werden
  - aus der Kommandozeile
  - aus einem Maven-Build heraus
  - innerhalb einer Entwicklungsumgebung
- Der manuelle Aufruf aus der Kommandozeile heraus kommt nur selten vor
  - besser Testklassen automatisiert ausführen
- Beispielausgabe in der Kommandozeile:

```
$ java -cp … org.junit.runner.JUnitCore CalculatorTest

JUnit version 4.11
..
Time: 0,006

OK (2 tests)
```

JUnit mit Maven

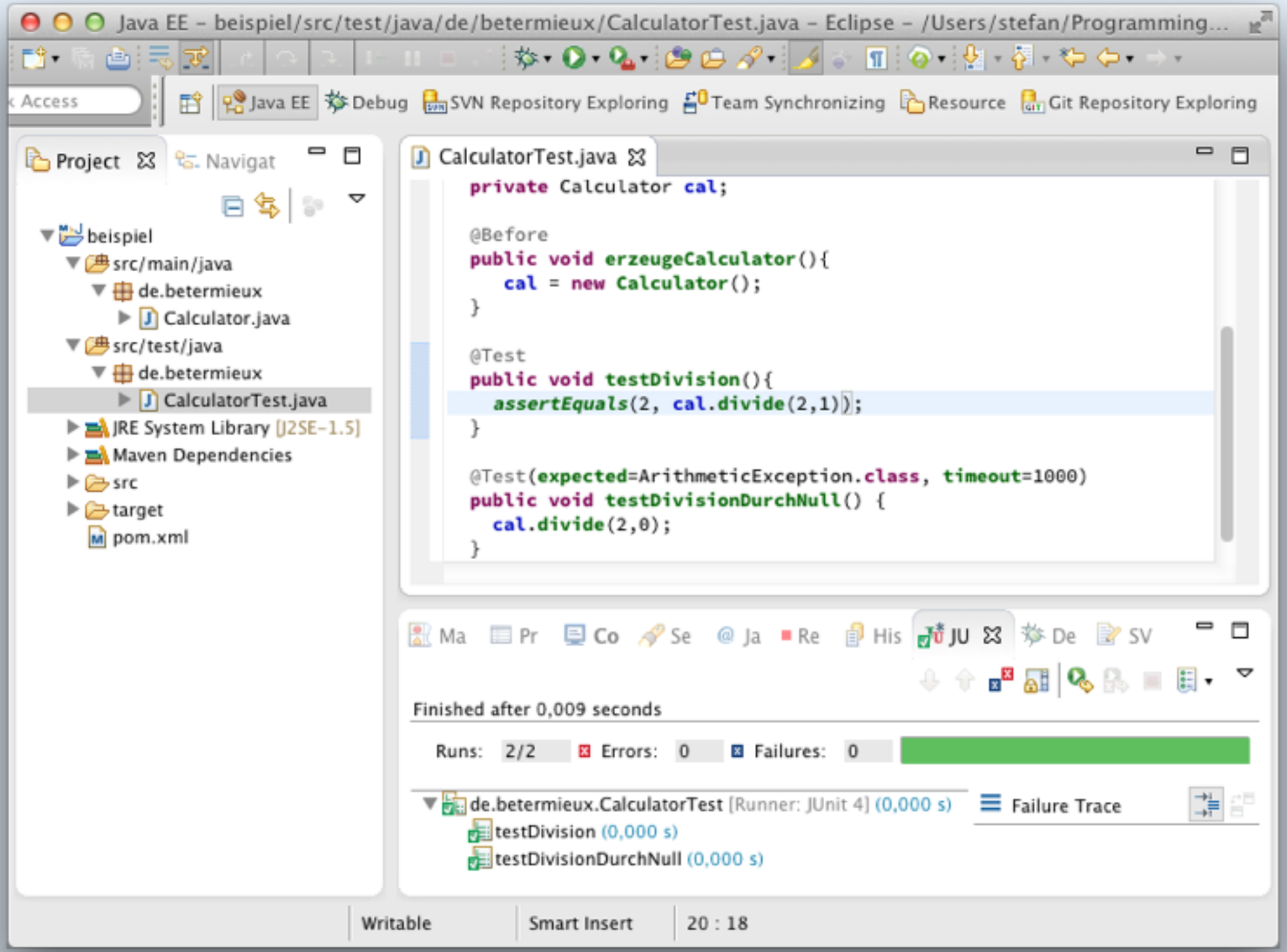
- Der Testdurchlauf ist Teil des Default Lifecycle
  - nach dem kompilieren der Anwendungsklassen
  - vor dem Paketieren
  - Testklassen im Verzeichnis `src/test/java` werden automatisch ausgeführt
  - wenn ein Fehler auftritt wird der Build-Prozess abgebrochen
- Beispiel-Ausgabe von Maven mit Testklassen:

```
$ mvn test
…
[INFO]
[INFO] --- maven-surefire-plugin:2.10:test (default-test) @ beispiel ---
[INFO] Surefire report directory: /Users/stefan/Temp/beispiel/target/surefire-reports
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running de.betermieux.CalculatorTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.046 sec
```

JUnit in Eclipse

- JUnit wird standardmäßig in Eclipse unterstützt
  - rechtsklick auf Testklasse → Run as… → JUnit Test
  - neue View mit den Testergebnissen öffnet sich
- JUnit-Tests müssen in Eclipse immer manuell angestoßen werden
  - schöner wäre eine automatische Ausführung bei jedem Abspeichern
  - das Infinitest-Plugin unterstützt dieses Vorgehen
    - `http://infinitest.github.io/`
    - Eclipse Marketplace → nach InfiniTest suchen

JUnit in Eclipse



Zusammenfassung

Schwierigkeiten

- Testen der grafischen Benutzeroberfläche
  - besser im Rahmen von Akzeptanztest, dann aber früher im Entwicklungszyklus
- Asynchroner Programmcode
  - Race-Conditions, etc…
- Komplexe Methoden
  - ist ein Code-Smell → besser Refactoring betreiben
- Zufalls- und datumsabhängige Funktionen
  - schwer reproduzierbar

Danke