

Vorlesung Programmieren

16. Testen und Assertions, JUnit

Prof. Dr.-Ing. Anne Koziolk

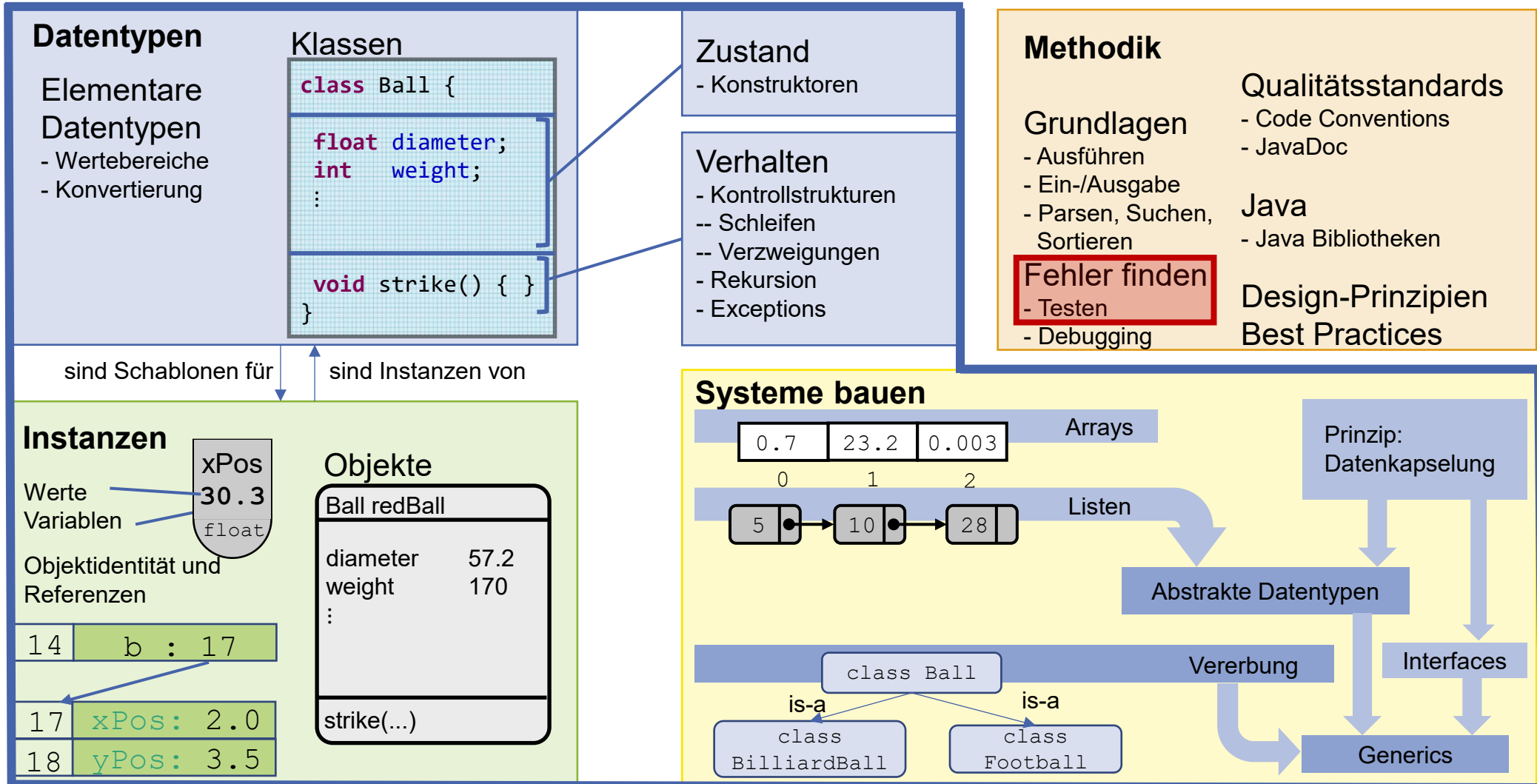


Termine und Fristen im Wintersemester 2024/25

- Übungsschein wird voraussichtlich am 07.02.2025 vorläufig und am 11.02.2025 endgültig in das Campus-Management-Portal eingestellt
 - Übungsschein ist Voraussetzung für die Anmeldung der Abschlussaufgaben
 - Die aktuellen Punkte sind über Artemis oder der zugehörigen Newsliste einsehbar
- Anmeldung zu den Abschlussaufgaben im Campus-Management-Portal vom 12.02.2025 bis 24.02.2025, jeweils 12:00 Uhr
 - Nach Anmeldeschluss ist keine Nachmeldung mehr möglich
 - Abgabe der Abschlussaufgaben nur nach Anmeldung
- Weitere Termine und Fristen wie immer auf der Veranstaltungsseite
 - <https://sdq.kastel.kit.edu/wiki/Programmieren>

Vorlesungsüberblick

Objektorientierte Programmierung in Java



Vorlesungsüberblick: Vorläufiger Semesterplan

21.10.2024	Erstsemesterbegrüßung: Einführung
23.10.2024	Organisatorisches; Ein Einfaches Programm, Objekte und Klassen
30.10.2024	Typen und Variablen
06.11.2024	Kontrollstrukturen (+Scanner)
13.11.2024	Konstruktoren und Methoden
20.11.2024	Arrays; Konvertierung, Datenkapselung, Sichtbarkeit
27.11.2024	Listen und Abstrakte Datentypen
04.12.2024	Vererbung
11.12.2024	Exceptions; Interfaces
18.12.2024	Generics; Rekursion
08.01.2025	Java-API; Objektorientierte Design-Prinzipien
15.01.2025	Best Practices; Finden und Beheben von Fehlern
22.01.2025	Testen und Assertions
29.01.2025	JUnit; Parsen, Suchen, Sortieren
05.02.2025	Vom Programm zur Maschine; Ausblick auf zukünftige Lehrveranstaltungen
12.02.2025	Wrap-Up

Lernziele

Testen und Assertions

- Sie kennen Kriterien für gute Testfälle und können anhand dieser Kriterien gute Testfälle erstellen
- Sie können Zusicherungen (Assertions) anwenden
- Sie können unterscheiden, welche Arten von Fehlern durch Testen entdeckt bzw. nicht entdeckt werden können
- Sie können begründen, warum Testen notwendig und sinnvoll ist
- Sie können das **JUnit**-Framework einsetzen
- Sie können parametrisierte Tests schreiben



Quelle: <http://phdcomics.com>

Murphy's Law

If it can go wrong, it will go wrong!

Software-Katastrophen



Bestrahlungsgerät Therac-25

<https://de.wikipedia.org/wiki/Therac-25>



»Blackout« in Nordamerika (2003)

https://en.wikipedia.org/wiki/Northeast_blackout_of_2003



Heartbleed (2012–2014)

<https://de.wikipedia.org/wiki/Heartbleed>

Was lernen wir daraus?

Umfassende Fehlerbehandlung erforderlich

- Jede Benutzereingabe sollte überprüft werden
- Jeder mögliche Fehlerfall behandelt

Ausgiebiges Testen unerlässlich!

»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

Software-Fehler (Wdh.)

- **Fault:** Ein Fehler bei der Entwicklung
- **Defect:** Ein Fault, der durch Testen gefunden wurde.
- **Bug:** Ein Defect, der von den Entwicklern anerkannt wurde.
- **Error:** Software-Zustand, durch Fault, Bug oder Defect verursacht.
- **Failure:** Unterschied zum von der Software **erwarteten Verhalten** zur Laufzeit (z.B. Absturz, Ausfall)

Woher wissen wir, was das **erwartete Verhalten** von Software ist?

→ Spezifikation von Anforderungen

Anforderung (requirement):

Aussage über eine zu erfüllende Eigenschaft oder zu erbringende Leistung eines Programms/Systems.

📄 Avizienis, A., Laprie, J. C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1), 11-33. DOI [10.1109/TDSC.2004.2](https://doi.org/10.1109/TDSC.2004.2)

Definition »Requirement«

DEFINITION. **Requirement** –

1. A need perceived by a stakeholder.
2. A capability or property that a system shall have.
3. A documented representation of a need, capability or property.

DEFINITION. **Requirements Specification** – A systematically represented collection of requirements, typically for a system, that satisfies given criteria.

M. Glinz (2024). *Requirements Engineering Glossary*, Version 2.1.0. International Requirements Engineering Board (IREB). Originally published in 2011.
Available at <http://www.ireb.org> (check-out CPRE Glossary)

Mögliche Gründe für fehlerhafte Software

- Fehler in der Entwicklung (fault/defect/bug/error)
- Anforderungen unvollständig/widersprüchlich/unmöglich zu implementieren
- Falsche Interpretation der Anforderungen
- Dokumentation falsch (Verhalten des Programms falsch beschrieben)

→ Testen hilft, Software-Fehler zu finden!

→ **Jedes Programm muss getestet werden!**

Testen: Vorgehen

Typischerweise **zweistufiges** Vorgehen:

1. Auswahl der Testfälle
 - Klassifizierung der möglichen Eingabedaten
 - Standardsituationen und Randfälle betrachten
2. Auswahl der Testdaten
 - Repräsentative Daten zu jedem Testfall

Man muss versuchen, **alle** Situationen abzudecken, in denen das Programm potentiell vom erwarteten Verhalten abweichen kann

Beispiel: Klassifikation Dreiecke

Der Benutzer gibt drei ganze Zahlen ein. Diese repräsentieren Seitenlängen eines Dreiecks.

Das Programm soll das Dreieck klassifizieren als

- gleichseitig
- gleichschenkelig
- Unregelmäßig

Was sind gute Testfälle?

- Klassifizierung der möglichen Eingabedaten
- Standardsituationen und Randfälle betrachten

Aufgabe: Was sind gute Testfälle?

Gültiges gleichseitiges Dreieck

Gültiges gleichschenkliges Dreieck

Gültiges unregelmäßiges Dreieck

Genau eine Seite Länge 0

Genau eine Seite negative Länge

Summe zweier Seiten > 0 ergibt dritte

Summe zweier Seiten $<$ dritte Seite

...

Example from [Myers2012]

Klassifikation Dreiecke: Spezifikation

```
enum TriangleKind { Equilateral, Isosceles, Other };  
class Triangle {  
    public Triangle(int a, int b, int c) {...}  
    // Specification: ...  
    public TriangleKind classify() {...}  
}
```

Klassifikation Dreiecke: Spezifikation

```
enum TriangleKind { Equilateral, Isosceles, Other };  
class Triangle {  
    public Triangle(int a, int b, int c) {...}  
    // Classify a triangle as equilateral, isosceles, other.  
    // classifyTriangle returns  
    // a) Equilateral: If all sides (a, b, c) are equal in length  
    // b) Isosceles:    If two sides (e.g., a, b) are equal in length (and > 0),  
    // c) Other:       If all sides have different lengths > 0  
    public TriangleKind classify() {...}  
}
```

Klassifikation Dreiecke: Spezifikation

```
enum TriangleKind { Equilateral, Isosceles, Other, Invalid };  
class Triangle {  
    public Triangle(int a, int b, int c) {...}  
    // Classify a triangle as equilateral, isosceles, other, or invalid.  
    // classifyTriangle returns  
    // a) Equilateral: If all sides (a, b, c) are equal in length and  
    //                  all of a, b, c are strictly positive.  
    // b) Isosceles:   If two sides (e.g., a, b) are equal in length (and > 0),  
    //                  and the triangle is neither Equilateral nor Invalid.  
    // c) Other:       If all sides have different lengths > 0 and the triangle  
    //                  is not Invalid.  
    // d) Invalid:     If at least one side's length is <= 0, or if the largest  
    //                  side's length is larger or equal to the sum of the  
    //                  other sides.  
    public TriangleKind classify() {...}  
}
```

Klassifikation Dreiecke: Testfälle

```
class TriangleTest {
    static void test_1() {
        // Test case: equilateral
        Triangle t = new Triangle(5,5,5);
        TriangleKind tk = t.classify();
        assert tk == TriangleKind.Equilateral;
    }
    static void test_2() {
        // Test case: isosceles
        Triangle t = new Triangle(3,1,3);
        TriangleKind tk = t.classify();
        assert tk == TriangleKind.Isosceles;
    }
    ...
    static void test_27() {
        // Test case: largest side greater equals sum of other sides
        Triangle t = new Triangle(4,4,8);
        TriangleKind tk = t.classify();
        assert tk == TriangleKind.Invalid;
    }
}
```

Kriterien für gute Testfälle

- Prüfen genau eine (möglichst spezifische) Eigenschaft ab
- Sind möglichst kurz und prägnant
- Können isoliert ausgeführt werden (d.h. keine Abhängigkeiten zwischen Tests)
- Sind gut dokumentiert
- Stellen Bezug zur Spezifikation klar dar
- Nutzen existierende Testing-Frameworks (z.B. **JUnit**)
- Sind möglichst vorausschauend (bzgl. Änderungen am Programm)

Folgen häufig dem Schema:

1. Initialisierung
2. Durchführung (z.B. Methodenaufruf)
3. Prüfung (z.B. mittels **assert**)
4. Finalisierung (»Aufräumen«; nicht immer notwendig)

Testen ist ...

Dynamischer Vergleich von tatsächlichem und erwünschtem (= spezifiziertem) Verhalten

Ziele:

- Nachweis, dass Funktionalität erfüllt ist
- Fehlerdetektion

Unterschied psychologisch wesentlich!

- Sollten Programmierer ihre eigenen Programme testen?
- Im allgemeinen: nicht nur, aber manchmal ist es aus organisatorischen oder Kostengründen erforderlich

 More details in Chapter „The Psychology and Economics of Software Testing“, [Myers2012]

Testfall:

- Ein Testfall besteht immer aus Eingabe und erwarteter Ausgabe (plus ggf. Ausführungsbedingungen). Für jeden Testfall sollte auch eine Begründung existieren, warum er ausgewählt wurde.

Testen ist nicht ...

- Fehlerlokalisierung (*fault localization*)
- Fehler beheben

Alternative Methoden

- Statische Codeanalyse, ob manuell oder automatisiert
 - Von Checkstyle ...
 - ... bis hin zu mathematischen Beweisen (formale Verifikation)
- Reviews
- Inspections
- Walkthroughs

Kategorien von Tests

Funktionale Tests

Korrektheit bzgl. Spezifikation

Nebenläufigkeit/Thread-Sicherheit

Nichtfunktionale Tests

Performance

Sicherheit

Benutzbarkeit

Interoperabilität

Zuverlässigkeit

Wissen

Black-Box-Tests

White-Box-Tests

Struktur

Unit

Integration

System

Warum Testen so schwierig ist

Was macht einen Testfall zu einem guten Testfall?

- Fähigkeit, Fehler zu finden (Fehler nur mit Spezifikation!)
 - Dann aber kein guter Testfall für ein fehlerfreies Programm
- Fähigkeit, wahrscheinliche Fehler mit angemessenem Aufwand zu finden
 - Kosten, Tests zu schreiben / auszuführen / auszuwerten
 - Kosten verbleibender Fehler, je nach Schwere
 - Testsuiten sollen aus Managementgründen so klein wie möglich (so groß wie nötig) sein und so kurze Testfälle wie möglich (so lang wie nötig) enthalten.
 - Leichte Fault-Lokalisierung

Erwartete Ausgabe ist essentiell. Also benötigen wir eine **Spezifikation**.

Testselektion

Approximation »guter« Testfälle

Zentrales Problem:

- Mögliche Eingabedaten in Blöcke gruppieren, so dass (möglichst) jedes Element eines Blocks denselben Fehler provoziert

Lösung:

- Auf der Basis von Anforderungen
- Auf der Basis von Wissen um typische Fehler
- Zufällig (gleichverteilt oder gemäß Nutzerprofilen)
- Strukturbasiert
 - »jede Zeile einmal ausführen«
 - »jede Bedingung einmal zu wahr und einmal zu falsch auswerten«
 - ...

Teststrategien

Datenbasiert, auf Basis der Eingabeparameter

- **Gruppierung von Daten**, die zu vergleichbarem Verhalten führen (wie im Dreieck-Beispiel)
- **Grenzwerte**
 - Beispiel natürliche Zahlen: -1, 0, 1, Integer.MAX_VALUE, ...
 - Beispiel Liste: leere Liste, null-Referenz, einelementige Liste
 - Beispiel Array: null-Referenz, Länge 0, Länge 1

Programmbasiert (z.B. *Coverage Criteria*)

- Jede Zeile (mind.) einmal ausgeführt
- Jede Bedingung (mind.) einmal zu wahr und zu falsch ausgewertet
- ...

Wann sind wir fertig?

Wenn die Zeit um ist

- Nutzlos. (Beispiel: Einfach nichts tun)
- In der Praxis Kriterium Nr. 1

Wenn keine weiteren Fehler mit existierenden Tests gefunden werden

- Nutzlos. (Beispiel: Irrelevante Tests auswählen)

Wenn ein strukturelles Kriterium erfüllt ist

- Etwa: Jede Zeile einmal ausgeführt
- Nützlich, wenn es einen Zusammenhang zwischen diesen strukturellen Kriterien und Fehlerdetektion gibt.

Wenn eine vorher spezifizierte Anzahl Fehler gefunden wurde

- Großartig, erfordert aber gute Abschätzungen

Wenn die Rate der gefundenen Fehler ausreichend abgenommen hat

- Auch sehr gut, wenn ehrlich angewendet

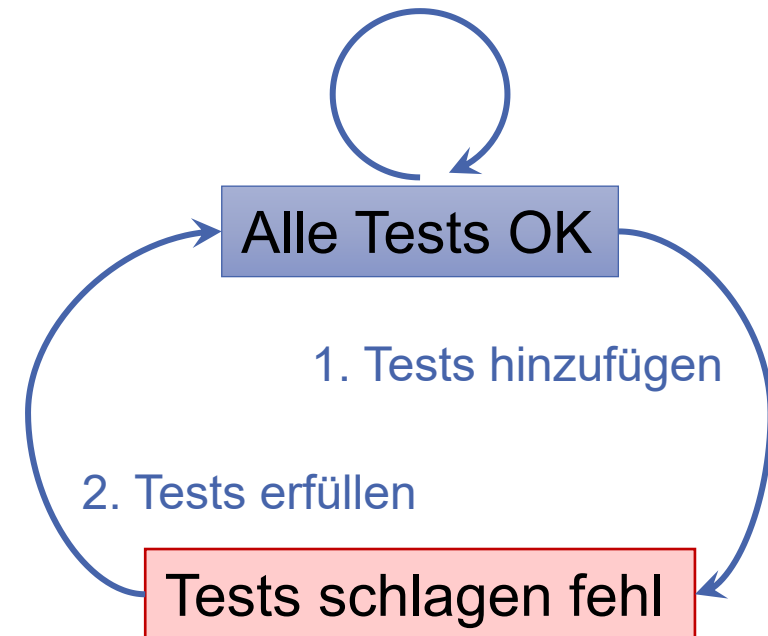
Ideal:
Mix aus
nützlichen
Kriterien

More details in Chapter „Higher-Order Testing“,
Section „Test Completion Criteria“ [Myers2012]

Testgetriebene Entwicklung

- Ausgiebiges Testen ist unerlässlich
- Trotzdem wird es oft unterlassen
 - Häufiger Grund: Zeitdruck
 - Teufelskreis:
Wenig Tests → geringere Produktivität
→ instabiler Code → höherer Zeitdruck
→ weniger Tests
- Besser: Tests gleich zusammen mit dem Programmcode schreiben
- Noch besser: Tests **vor** dem Programmcode schreiben

3. Code vereinfachen/Refactoring



Refactoring

Code so einfach wie möglich halten

- Vor dem Testen: Erfülle Tests mit einfachem Code
- Nach dem Testen: Code weiter vereinfachen

Refactorings:

- Änderungen der Struktur eines Programms, die das nach außen beobachtbare Verhalten nicht ändern.

Typische Refactoring-Operationen:

- Klassen aufteilen/kombinieren
- Methoden/Attribute in andere Klassen verschieben
- Umbenennen von Klassen, Methoden, Parametern, Variablen
- Methoden extrahieren/integrieren
- Vererbung durch Delegation ersetzen (und umgekehrt)
- Schnittstellen einführen



Martin Fowler, Kent Beck: *Refactoring: Improving the Design of Existing Code*.
2. Auflage, Addison-Wesley Professional, 2018

Statische Analyse

Wenn Auswahl der Testfälle so schwierig ist, warum nicht **alle Eingaben** testen?

- **Viel zu viele!**
- Management der Testfälle nicht mehr praktikabel
- Laufzeit der Testfälle zu groß

... oder geht es doch?

Idee: Transformiere Programm in logische Formel und versuche Einhaltung der Spezifikation zu beweisen

- **Verschiedene Logiken:** Dynamische Logik, Bit-Vektor-Logik, ...
 - **Spezifikation:** muss in formaler Logik vorliegen
 - **Beweisen:** manuell, Rechner-unterstützt, vollautomatisch
- Statische Programm-Analyse, formale Verifikation

ASSERTIONS

Assertions (Geek & Poke)



<http://geekandpoke.typepad.com/geekandpoke/2011/10/ddt.html>

Assertions (Geek & Poke)



<http://geekandpoke.typepad.com/geekandpoke/2011/10/ddt.html>

Assertions (Geek & Poke)



<http://geekandpoke.typepad.com/geekandpoke/2011/10/ddt.html>

Zusicherungen

Zusicherungen (assertions) sind ein weiteres wichtiges Hilfsmittel für die Entwicklung von Software und zur Vermeidung von Fehlern

Sie werden verwendet, um

- Restriktionen für Parameter und globale Variablen von Methoden anzugeben
→ **Vorbedingungen (preconditions)**
- den Effekt von Methoden formal zu beschreiben
→ **Nachbedingungen (postconditions)**
- an zentralen Programmpunkten wichtige Eigenschaften (z.B. mögliche Werte von Variablen) explizit festzuhalten

Mit Hilfe von Zusicherungen ist es möglich, Korrektheitsaussagen bzgl. eines gegebenen Programmes mathematisch zu beweisen

Zusicherungen

Beispiel:

Berechne das Skalarprodukt zweier Vektoren u und v .

```
double dotProduct(double[] u, double[] v) {  
    // hier muss gelten:  $u.length = v.length$   
    double s = 0;  
    for (int i = 0; i < u.length; i++) {  
        // hier gilt immer:  $s = \sum_{j=0}^{i-1} u_j \cdot u_v$   
        s += u[i] * v[i];  
    }  
    // hier gilt immer:  $s = \sum_{j=0}^{u.length-1} u_j \cdot u_v$   
    return s;  
}
```

Zusicherungen:

- **Vorbedingung:** muss vor Aufruf der Methode gelten
- **Schleifeninvariante:** muss bei jeder Schleifeniteration gelten
- **Nachbedingung:** muss nach Aufruf der Methode gelten

Assertions

Assertions ermöglichen es, angenommene erforderliche Bedingungen in Java zur Laufzeit zu prüfen.

Syntax

assert *Bedingung*;

assert *Bedingung*: "*Detaillierte Fehlermeldung*";

```
double dotProduct(double[] u, double[] v) {  
    // hier muss gelten: u.length = v.length  
    assert u.length == v.length;  
    double s = 0;  
    for (int i = 0; i < u.length; i++) {  
        ...  
    }  
}
```

Zusicherungen in Java

Java bietet die Möglichkeit, Zusicherungen mittels **assert** direkt in den Programmtext zu schreiben.

- **Vorteil:** Kann zur Laufzeit von der JVM nachgeprüft werden
- **Nachteil:** Es kann nur Java-Syntax verwendet werden, um die Bedingung auszudrücken

Häufig ist es einfacher, Zusicherungen in Kommentare zu schreiben.

- **Vorteil:** Volle sprachliche Freiheit
- **Nachteil:** Kann nicht (oder nur eingeschränkt) automatisiert überprüft werden

In jedem Fall: Assertions sind gute Dokumentation der Verantwortlichkeiten zwischen Methodennutzer und Methodenimplementierer. (*Design by contract*)

Zusicherungen in Java

Assertions werden standardmäßig nicht ausgewertet und überprüft!

- Aktivieren beim Programmstart mit der »enable-assertions«-Option:
- `java -ea TriangleTest`

Vor- und Nachbedingungen, Invarianten

Eine Zusicherung gibt eine Eigenschaft an, die bei der Ausführung des Programms an der entsprechenden Stelle erfüllt sein muss

- Eine **Vorbedingung (precondition)** muss vor der Abarbeitung des entsprechenden Methodenrumpfs erfüllt sein
- Eine **Nachbedingung (postcondition)** muss erfüllt sein, nachdem die Methode abgearbeitet wurde
- Eine **Klassen-Invariante** beschreibt den gültigen Zustand eines Objekts. Sie muss sowohl vor als auch nach jedem Methodenaufruf (Ausnahme: private Hilfsmethoden) des zugehörigen Objekts gültig sein.
 - Beispiel: `int length; /* ASSERT length >= 0 */`
 - Vor Ende jeder Public-Methode: `assert length >= 0`
- Eine **Schleifen-Invariante** ist eine Zusicherung, die am Anfang und Ende eines jeden Durchlaufs der zugehörigen Schleife erfüllt sein muss

Beispiel Invariante: Klasse zum Verwalten von Uhrzeit

```
public class Time {
    int hour, minute;

    // invariant: 0 <= hour && hour < 24
    // invariant: 0 <= minute && minute < 60

    public Time(int h, int m) {
        hour = h;
        minute = m;
        assert 0 <= hour && hour < 24;
        assert 0 <= minute && minute < 60;
    }

    public void addTime(Time t) {
        minute += t.minute;
        hour += t.hour;
        normalizeTime();
        assert 0 <= hour && hour < 24;
        assert 0 <= minute && minute < 60;
    }
}
```

```
private void normalizeTime() {
    if (minute >= 60) {
        minute -= 60;
        hour += 1;
    }
    if (hour >= 24) {
        hour -= 24;
    }
}

...
}
```

- Annahmen mit Assertions explizit machen

- Wo?
- Zusicherungen erfüllt?

Postcondition/Klasseninvariante kann verletzt sein!

Beitreten unter
slido.com
#Prog



Assertion oder If-Abfrage?

Assertion

- Annahme des Programmierers zur Korrektheit des Programms
- Auch zur Dokumentation
- Für nicht zur Laufzeit behandelbare Fehler
- Abschaltbar

Beispiel:

```
void setMinute(int m) {  
    assert m >= 0;  
    this.minute = m;  
}
```

If-Abfrage und Exceptions

- Prüfung von Eingabedaten des Benutzers
- Für Sonderfälle
- Für von aufrufenden Funktionen behandelbare Fehler
- Nicht abschaltbar

Beispiel:

```
void setMinute(int m) {  
    if (m < 0) {  
        throw new IllegalArgumentException();  
    }  
    this.minute = m;  
}
```

JUnit 5

Diese Folien enthalten Material aus:

JUnit-5-Tutorial von Lars Vogel: <http://www.vogella.com/tutorials/JUnit/article.html>

JUnit-Einführung von Alan Williams (inzwischen nicht mehr verfügbar)

JUnit – Übersicht

JUnit ist ein Open-Source-Framework zum Schreiben von Tests

- JUnit verwendet die *Java Reflection API* (Programme können sich selbst untersuchen)
- JUnit hilft Entwickler/-innen
 - bei der Definition und Ausführung von Tests und Test-Suiten
 - beim Formalisieren von Anforderungen und der Architektur
 - beim Schreiben und Debuggen von Code
 - bei Code-Integration, sodass ständig eine lauffähige Version des Programms besteht
- JUnit ist nicht Teil des Oracle-SDKs, doch von fast allen IDEs (z.B. Eclipse)
- Die Autoren von JUnit sind Erich Gamma (*Design Patterns*) und Kent Beck (*XP*)
- JUnit hat weitere Frameworks inspiriert (NUnit, CppUnit, PHPUnit, ABAP Unit, ...)
- JUnit ist der *De-Facto-Standard* für testgetriebene Entwicklung mit Java

Beispiel (JUnit 5)

```
package edu.kit.kastel;

import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

class TriangleTest {

    @Test
    public void equilateralTriangle() {
        Triangle triangle = new Triangle(5, 5, 5); // class Triangle is tested
        // assert statement
        assertEquals(TriangleKind.EQUILATERAL, triangle.classify(), "Triangle is not equilateral");
    }

    @Test
    public void isoscelesTriangle() {
        Triangle tester = new Triangle(3, 1, 3);
        assertEquals(TriangleKind.ISOSCELES, triangle.classify(), "Triangle is not isosceles");
    }
}
```

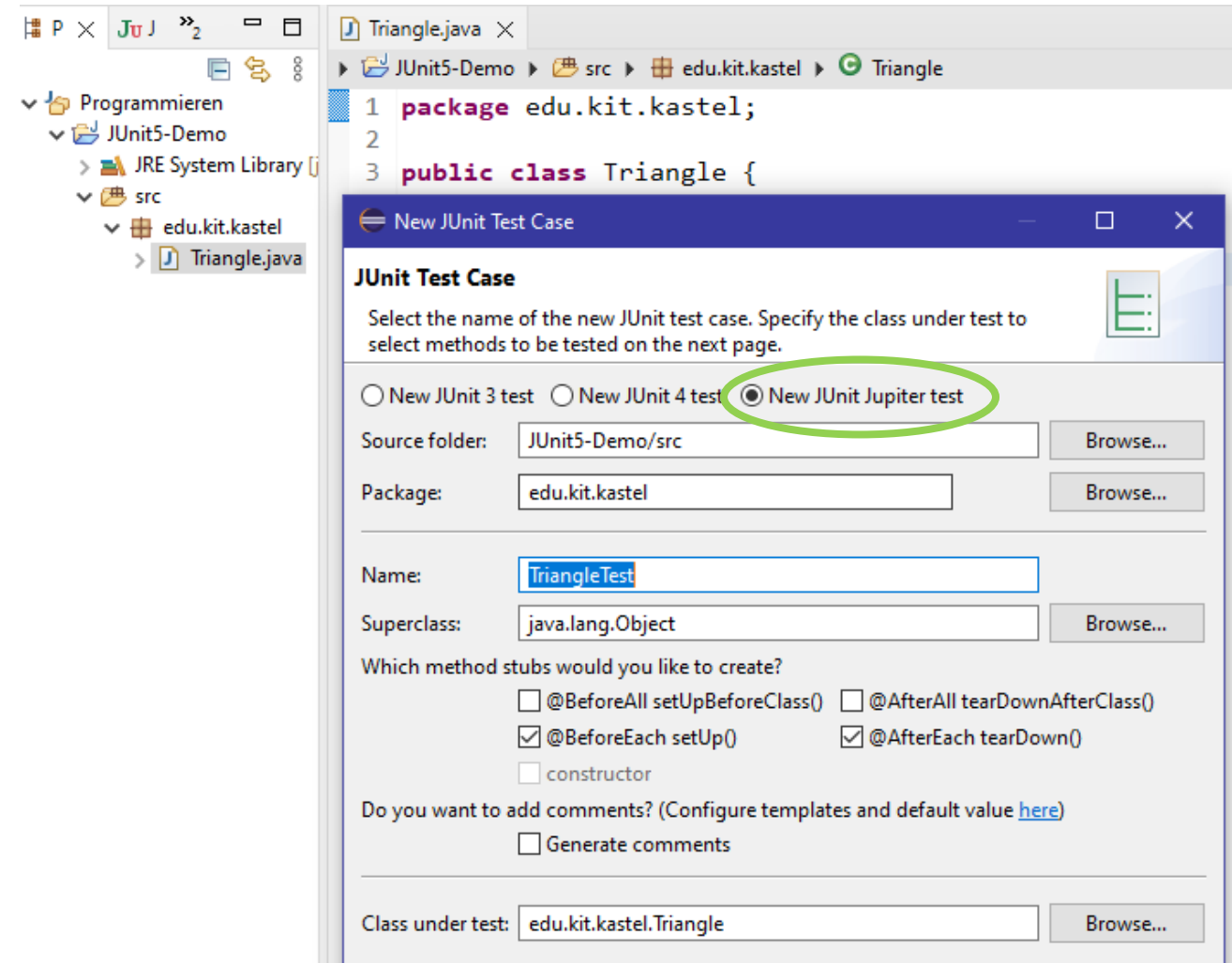
JUnit 5 Testfälle in Eclipse anlegen

■ Eclipse

- File > New > JUnit Test Case
- In Code on Class Name: CTRL + 1

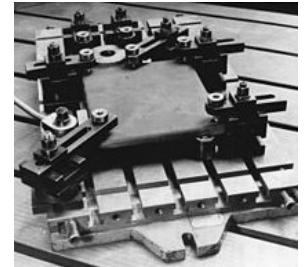
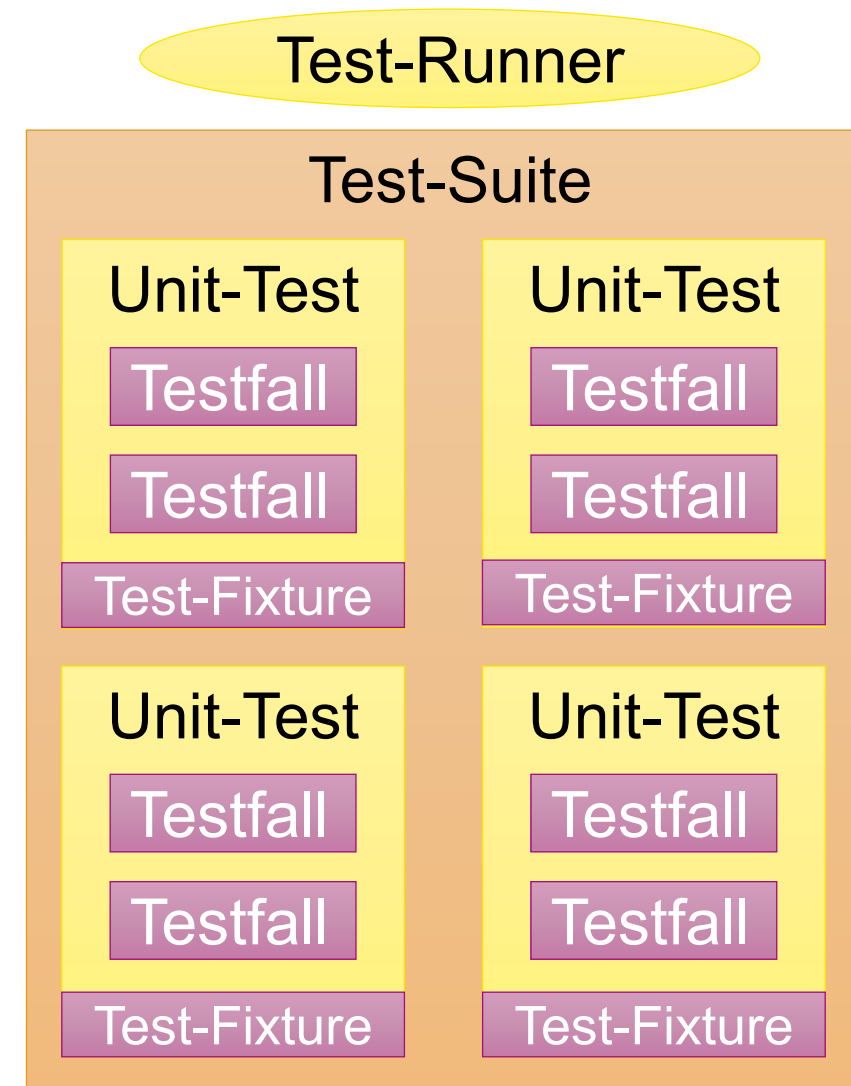
■ IntelliJ

- In Code on Class Name: Alt + Enter



Struktur/Terminologie von JUnit

- Ein **Testfall** testet eine einzelne Methode (soweit möglich)
- Ein **Unit-Test** testet Methoden einer einzelnen Klasse
- Eine **Test-Suite** kombiniert Unit-Tests
- Die **Test-Fixture** (*»Vorrichtung«*) liefert die Daten (Objekte, Werte), die zur Ausführung der Testfälle benötigt werden
- Der **Test-Runner** führt die Tests oder eine Suite aus und stellt die Ergebnisse dar



Was ist ein JUnit-Testfall?

Ein Testfall besteht aus einer oder mehreren Java-Methoden.

Hinzugefügt werden:

- Annotationen (@Test usw.)
- Assertions:
 - JUnit beinhaltet ein Assertions-Paket zur Überprüfung von
 - Gleichheit von Objekten
 - identischen Objektreferenzen
 - Objektreferenzen, die **null**/nicht **null** sind
 - Assertions werden verwendet, um das Test-Urteil zu bestimmen.

Test-Urteile

Ein Urteil (verdict) ist das Ergebnis der Ausführung eines einzelnen Testfalls.

- **Pass:** Der Testfall erfüllte seinen Zweck, und die getestete Software verhielt sich wie erwartet.
- **Fail:** Der Testfall erfüllte seinen Zweck, doch die getestete Software verhielt sich **nicht** wie erwartet.
- **Error:** Der Testfall erfüllte seinen Zweck **nicht**.
Mögliche Gründe:
 - Ein unerwartetes Ereignis trat während der Ausführung des Testfalls auf.
 - Der Test konnte nicht richtig aufgesetzt werden.
 - ...

Achtung: Die Terminologie hier entspricht nicht der Definition von Laprie (fault/defect/bug/error/failure) auf Folie 7

Assertions

- In der JUnit-Klasse `org.junit.jupiter.api.Assertions` sind JUnit-spezifische Assertions definiert (≠ Java-Schlüsselwort `assert`)
- Wird irgendeine dieser Assertions während des Testens verletzt, wird die Methodenausführung an diesem Punkt gestoppt, und das Testurteil ist **Fail**.
- Wird eine Ausnahme (*Exception*) während der Methodenausführung geworfen, ist das Testurteil **Error**.
- Wird keine Assertion verletzt, ist das Testurteil **Pass**.

Alle Assertion-Methoden sind *statische Methoden* (Tipp: Statisch importieren).

Assertion-Methoden

Statement	Beschreibung
<code>fail([msg])</code>	Produziert immer Fail .
<code>fail([msg,] throwable)</code>	Produziert immer Fail , mit throwable-Nachricht
<code>assertTrue(boolean condition[, msg])</code>	Prüft, ob boolsche Bedingung wahr.
<code>assertFalse(boolean condition[, msg])</code>	Prüft, ob boolsche Bedingung falsch.
<code>assertEquals(expected, actual[, msg])</code>	Prüft Gleichheit zweier Werte. (Bei Arrays: Referenz, nicht Inhalt).
<code>assertEquals(expected, actual, delta[, msg])</code>	Prüft Gleichheit zweier Fließkommazahlen. Toleranz=Anzahl identischer Dezimalstellen
<code>assertNull(object[, msg])</code>	Prüft, ob Objekt null ist
<code>assertNotNull(object[, msg])</code>	Prüft, ob Objekt nicht null ist
<code>assertSame(expected, actual[, msg])</code>	Prüft, ob beide Variablen auf dasselbe Objekt verweisen
<code>assertNotSame(expected, actual[, msg])</code>	Prüft, ob beide Variablen auf versch. Objekte verweisen
<code>assertThrows(expectedType, executable[, msg])</code>	Prüft, ob Exception eines bestimmten Typs geworfen wurde

Gleichheits-Assertions

Gleichheit von Objekten

`assertEquals(expected, actual)` ist gültig gdw. `expected.equals(actual)`

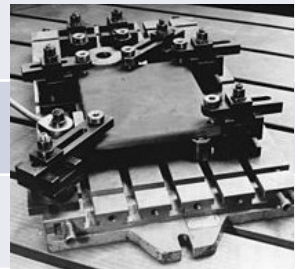
Gleichheit von Arrays

`assertArrayEquals(expected, actual)`

- Arrays müssen dieselbe Länge haben
- Ist gültig, wenn für jeden gültigen Wert von `i` gilt:
- `assertEquals(expected[i], actual[i])`

Testfälle und Test-Fixture in JUnit 5

Annotation	Beschreibung
<code>import org.junit.jupiter.api.*</code>	Import-Anweisung für folgende Annotationen
<code>@Test</code>	Identifiziert Methode als Test-Methode
<code>@BeforeEach</code>	Wird vor jedem Test ausgeführt
<code>@AfterEach</code>	Wird nach jedem Test ausgeführt
<code>@BeforeAll</code>	Wird einmal vor Beginn aller Tests ausgeführt
<code>@AfterAll</code>	Wird einmal nach Ende aller Tests ausgeführt
<code>@Disabled</code> / <code>@Disabled("Why disabled")</code>	Test ist deaktiviert
<code>@Timeout(value=100, unit=TimeUnit.SECONDS)</code>	Wenn Dauer der Ausführung > 100 <i>Sekunden</i> : Fehlschlag des Tests
<code>@DisplayName</code>	Individueller Name für den Test (statt Methodename)



Hinweis: `@Test(expected=Exception.class)` wurde in JUnit 5 abgelöst durch `assertThrows`

@BeforeEach/@AfterEach in JUnit 5

- Methoden mit der Annotation @BeforeEach werden vor jedem Test ausgeführt
- Man kann mehrere Methoden mit @BeforeEach annotieren. Sie werden alle vor jedem Test ausgeführt, jedoch ist die Reihenfolge beliebig.
- Methoden mit der Annotation @AfterEach werden nach jedem Test ausgeführt, auch wenn eine Exception geworfen wurde oder wenn der Test fehlschlägt.
- Methoden mit der Annotation @BeforeEach/@AfterEach dürfen beliebig benannt werden, müssen jedoch **void** als Rückgabetyp haben.
- Methoden mit der Annotation @BeforeAll/@AfterAll dürfen beliebig benannt werden, müssen jedoch **static** sein.

Test-Suiten in JUnit 5

- Tests können zu **Test-Suiten** kombiniert werden
- Suiten können andere Suiten enthalten
- Partitionierung der Testfälle möglich
- Werden gut durch die Test-Runner unterstützt

```
import org.junit.platform.suite.api.Suite;
import org.junit.runners.Suite.SuiteClasses;

@Suite
@SuiteClasses({ TriangleTest.class, CircleTest.class })
public class GeometryTests {

}
```

Parametrisierte Tests in JUnit 5

Um einen Test mit verschiedenen Parameter-Werten auszuführen, müsste man:

- über eine Menge von Werten iterieren
→ bei Test-Fehlschlag während Iteration:
möglicherweise Endlosschleife
- einzelne Testfälle für jede Kombination schreiben
→ hoher manueller Aufwand

JUnit bietet Unterstützung für
parametrisierte Tests.

Erstellung eines einfachen parametrisierten Tests:

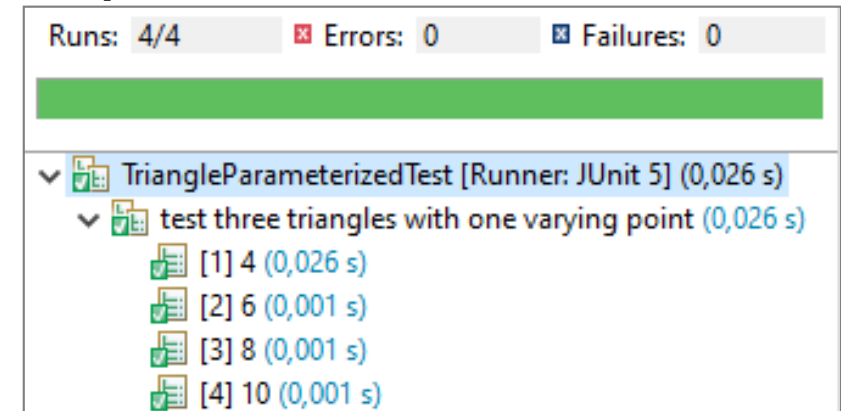
1. Test mit Annotation
`@ParameterizedTest` schreiben
2. Parameter der Methode hinzufügen,
z.B. `int index` oder `String name`
3. Testfall ergänzen mit der Annotation für die Parameter:
`@ValueSource(ints = {0, 2, -1})` oder
`@ValueSource(strings = {"A", "B"})`

Parametrisierte Tests: Beispiel

```
public class TriangleParameterizedTest {  
  
    @ParameterizedTest  
    @ValueSource(ints = {4, 6, 8, 10}) // Testdaten  
    @DisplayName("test four triangles with one varying point")  
    public void testIsoscelesTriangle(int variableParameter) {  
        Triangle triangle = new Triangle(2, 2, variableParameter);  
        assertEquals(TriangleKind.ISOSCELES, triangle.classify(),  
            "Triangle is not isosceles");  
    }  
}
```

- Parametrisierte Tests in JUnit 5 sind deutlich einfacher als in JUnit 4
- Annotationen für häufige Werte:
 - @NullSource
 - @EmptySource
 - @NullAndEmptySource

Eclipse:



IntelliJ:

✓ TriangleParameterizedTest	49 ms
✓ test three triangles with one varying point	49 ms
✓ [1] 4	45 ms
✓ [2] 6	2 ms
✓ [3] 8	1 ms
✓ [4] 10	1 ms

Verschiedene Datenquellen

Direkte Werte als Datenquelle

1. Test mit Annotation
`@ParameterizedTest` schreiben
2. Parameter der Methode hinzufügen, z.B.
`int index` oder `String name`
3. Testfall ergänzen mit der Annotation für die Parameter:
`@ValueSource(ints = {0, 2, -1})` oder
`@ValueSource(strings = {"A", "B"})`

Weitere Datenquellen:

- Enums mit `@EnumSource`
- CSV-Dateien mit `@CsvSource` und `@CsvFileSource`
- ArgumentsProvider Implementierungen mit `@ArgumentsSource`

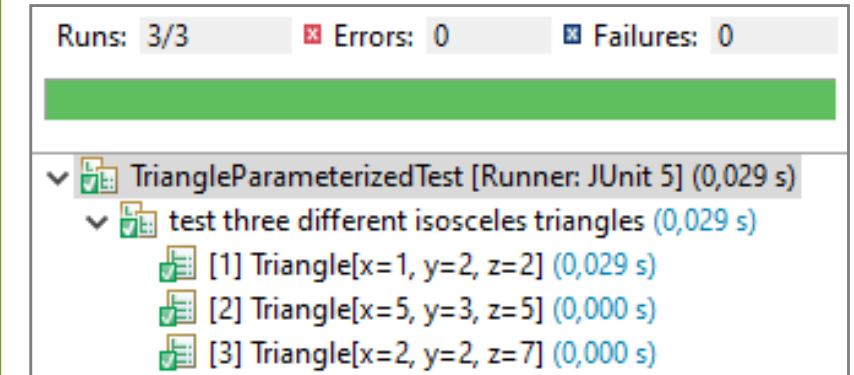
Methoden als Datenquelle

1. Test mit Annotation
`@ParameterizedTest` und `Parameter` schreiben
2. Statische Methode zur Datenlieferung schreiben, die eine Collection zurückgibt
3. Testfall ergänzen mit der Annotation:
`@MethodSource("nameOfMethod")`

Parametrisierte Tests: Beispiel mit Methode

```
public class TriangleParameterizedTest {  
    // Testdaten  
    private static Collection<Triangle> isoscelesTriangles() {  
        return List.of(new Triangle(1, 2, 2),  
                        new Triangle(5, 3, 5),  
                        new Triangle(2, 2, 7));  
    }  
  
    @ParameterizedTest  
    @MethodSource("isoscelesTriangles")  
    @DisplayName("test three different isosceles triangles")  
    void testIsoscelesTriangle(Triangle tester) {  
        assertEquals(TriangleKind.ISOSCELES, tester.classify(),  
                     "Triangle is not isosceles");  
    }  
}
```

Eclipse:



IntelliJ:

✓ TriangleParameterizedTest	36 ms
✓ test three different isosceles triangles	36 ms
✓ [1] Triangle[x=1, y=2, z=2]	35 ms
✓ [2] Triangle[x=5, y=3, z=5]	1 ms
✓ [3] Triangle[x=2, y=2, z=7]	

- Komplexe Testfälle können auch in JUnit 5 Methoden als Datenquellen nutzen
- Mehrere Parameter via `Arguments.of`

Ausführen von Tests

Mit dem in JUnit enthaltenen Test-Runner:

- Alle Test-Methoden in der Klasse werden ausgeführt
- Die Reihenfolge des Methodenaufrufs (d.h. der Testfälle) ist nicht vorhersagbar

Andere Test-Runner

- Test-Runner in IDEs (z.B. Eclipse) *können* es dem/-r Benutzer/-in ermöglichen, eine Teilmenge der Testmethoden auszuwählen oder eine Reihenfolge zu bestimmen
- Gute Praxis: Tests so schreiben,
 - dass sie unabhängig von der Ausführungsreihenfolge sind
 - dass sie unabhängig vom Zustand von vorherigen Tests sind
- Stattdessen `@BeforeEach/@BeforeAll` usw. verwenden

Zusammenfassung

- Ausgiebiges Testen ist unerlässlich
- Ziele des Testens:
 - Nachweis, dass Funktionalität erfüllt ist
 - Entdecken von Fehlern
- Tests sollten so viele Situationen wie möglich abdecken
- Kriterien für gute Testfälle und Teststrategien verbessern die Qualität der Tests
- Zusicherungen (Assertions) in Java können erwartete Bedingungen zur Laufzeit prüfen
- JUnit ist ein mächtiges Framework zum Testen von Java-Code
- Parametrisierte Tests ermöglichen das automatische Testen großer Mengen an Testdaten



Literaturhinweis – Weiterlesen

Dietmar Ratz, Jens Scheffler, Detlef Seese und Jan Wiesenberger: *Grundkurs Programmieren in Java*, 7. Auflage, 2014 (mit Java 8), Hansa-Verlag

- Kapitel 10 »Exceptions and Errors«
 - Abschnitt 10.3 »Assertions«

Martin Fowler, Kent Beck: *Refactoring: Improving the Design of Existing Code*. 2. Auflage, Addison-Wesley Professional, 2018

[Myers2012] Myers, Glenford J., et al. *The art of software testing*. Vol. 2. Chichester: John Wiley & Sons, 2012. (available online via KIT library at <https://ebookcentral.proquest.com/lib/karlsruhetech/detail.action?docID=697721>)