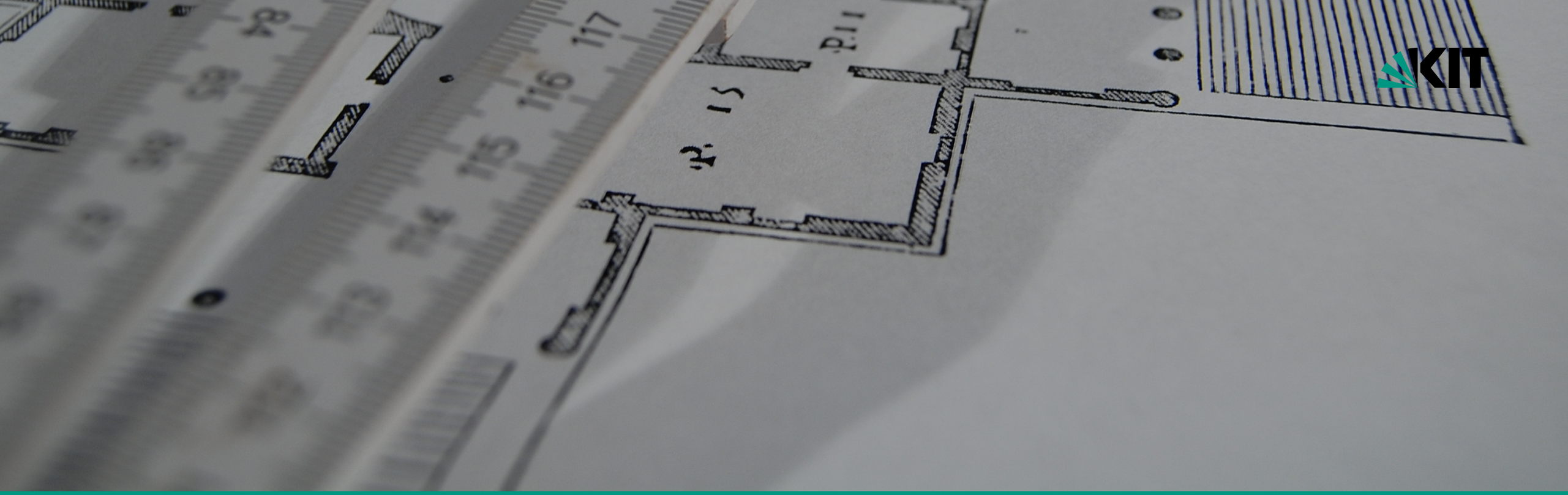

Testing, Assertions und Lambdas

Programmieren Tutorium Nr.12

Aleksandr Zakharov | 10. Februar 2026

Testen

Testen

Jedes Programm sollte getestet werden. Wichtig sind dabei:

- Auswahl der Testfälle
  - Klassifizierung der möglichen Eingabedaten
  - Standardsituationen **und** Randfälle beachten
- Auswahl der Testdaten
  - Repräsentative Daten zu jedem Testfall

$\Longrightarrow$ Versuchen *alle* Situationen abzudecken, in denen das Programm potenziell schief gehen kann.

Testen

Ein klassisches Quiz

Der Benutzer gibt drei ganze Zahlen ein. Diese Zahlen repräsentieren die Länge der Seiten eines Dreiecks. Das Programm klassifiziert das Dreieck als gleichseitig, gleichschenklig, oder weder noch.
Nenne geeignete Testfälle.

Testen

16 Testfälle!

- Gültiges weder-noch-Dreieck
- Gültiges gleichseitiges Dreieck
- Gültiges gleichschenkliges Dreieck (plus Permutationen, z.B. 4,3,3;3,4,3; 3,3,4)
- Genau eine Seite Länge Null
- Eine Seite negative Länge
- Summe zweier nichtnegativer Seiten ergibt dritte Seite (plus Permutationen)
- Summe zweier nichtnegativer Seiten ist kleiner als dritte Seite (plus Permutationen)
- Alle Seiten Länge Null
- Nicht-ganzzahlige Werte
- $\neq$ 3 Parameter
- Erwartete Ausgabe?

Testen

Allgemein

- Dynamischer Vergleich von tatsächlichen und erwünschtem (= spezifizierten) Programmverhalten
- Ziele: Nachweis, dass Funktionalität erfüllt ist & Fehlerdetektion
- Unterschied psychologisch wesentlich
  - Sollten Programmierer ihre eigenen Programme testen?
  - Im Allgemeinen nein; aber das ergibt organisatorische und Kostenprobleme

Testen

Allgemein

- Dynamischer Vergleich von tatsächlichen und erwünschtem (= spezifizierten) Programmverhalten
- Ziele: Nachweis, dass Funktionalität erfüllt ist & Fehlerdetektion
- Unterschied psychologisch wesentlich
  - Sollten Programmierer ihre eigenen Programme testen?
  - Im Allgemeinen nein; aber das ergibt organisatorische und Kostenprobleme

Testfall

Ein Testfall besteht immer aus Ein- und erwarteter Ausgabe (plus ggf. Ausführungsbedingungen). Für jeden Testfall sollte auch eine Begründung existieren, warum er ausgewählt wurde. Im Entwicklungsmodell XP sind Testfälle die (einzige) Spezifikation!

Testen - Drei Fehlerarten

Failure

- Unterschied zwischen Spezifikation und beobachtbarem Verhalten
- Manifestation zur Laufzeit
- Wenn Testfälle als eigenständiges Programm implementiert werden, kann der Vergleich von erwarteter und tatsächlicher Ausgabe als Assertion formuliert werden. Gilt auch für einzelne Funktionen.
- Gute Frameworks, etwa *JUnit*, verfügbar

Testen - Drei Fehlerarten

Error

- Unterschied zwischen tatsächlichem und erwünschtem internen Programmzustand
- Kann, muss aber nicht zu einer Failure führen
- Kann zur Laufzeit „repariert“ (z.B. durch Redundanz)
- Kann mit Assertions überprüft werden

Fault

- Tatsächlicher oder vermuteter Grund für die Abweichung von tatsächlichem und erwünschtem beobachtbaren Verhalten oder dem Programmzustand

Testen

Testen ist nicht . . .

- Qualitätsverbesserung
  - Das wird uA. durch Debugging erreicht
- Fehlerlokalisierung (fault localization)
- Statische Codeanalyse, ob manuell oder automatisiert
  - Formale Beweise
  - Manche nennen verschiedene Lesetechniken (reviews, inspections, walkthroughs) „statisches Testen“. Wir nicht.

Testen

Testen ist schwer . . .

Was macht einen Testfall zu einem guten Testfall?

- Fähigkeit, Fehler (failures) zu finden
- Fähigkeit, potentielle Fehler zu finden
  - Fähigkeit, wahrscheinliche Fehler mit angemessenem Aufwand zu finden
    - Kosten: Tests schreiben/ausführen/auszuwerten
    - Kosten: verbleibende Fehler, je nach Schwere
    - Leichte Fault-Lokalisierung

Ausgezeichnet! Und viel zu abstrakt!

Testen

Testselektion

- Approximation „guter“ Testfälle
- Zentrales Problem
  - Mögliche Eingabedaten in Blöcke gruppieren, so dass (möglichst) jedes Element eines Blocks denselben Fehler provoziert
  - (Erfordert natürlich a priori-Wissen um mögliche Fehler)

Testen

Testselektion

- Approximation „guter“ Testfälle
- Zentrales Problem
  - Mögliche Eingabedaten in Blöcke gruppieren, so dass (möglichst) jedes Element eines Blocks denselben Fehler provoziert
  - (Erfordert natürlich a priori-Wissen um mögliche Fehler)
- Lösung
  - Auf der Basis von Anforderungen
  - Mit Wissen um typische Fehler
  - Strukturbasiert („jede Zeile einmal ausführen“, „jede Bedingung einmal zu wahr und einmal zu falsch auswerten“, usw.)

Testen

Anwendungsbasiertes Testen

- Datenbasiert, aus Basis der Eingabeparameter
- Kontrollflussbasiert, auf Basis angenommener „typischer“ oder „relevanter“ Interaktionen mit dem Programm (also Programmabläufe)

Testen

Anwendungsbasiertes Testen

- Datenbasiert, aus Basis der Eingabeparameter
- Kontrollflussbasiert, auf Basis angenommener „typischer“ oder „relevanter“ Interaktionen mit dem Programm (also Programmabläufe)

Category Partition Method

- Identifiziere individuelle funktionale Einheiten, die separat getestet werden können. Bestimme für jede Einheit die das Verhalten der Einheit beeinflussenden Parameter und relevante Umgebungsvariablen. (= Komponenten einzeln testen)
- Bestimme die individuellen Wahlmöglichkeiten für jeden Parameter und jede Umweltvariable.
- Bestimme Abhängigkeiten und Constraints zwischen den Wahlmöglichkeiten.
- Erzeuge alle relevanten, die Constraints erfüllenden, Kombinationen von Werten für die Kategorien.
- Transformiere diese Werte in ausführbare Testfälle inkl. der erwarteten Ausgaben.

Testen

Grenzwerttesten

Wenn die Datentypen der Kategorien eine natürliche Ordnung mit Grenzen aufweisen, dann ergeben diese Grenzen (und Werte „leicht links“ und „rechts davon“) zusätzliche natürliche Kandidaten für Blöcke der partitionierten Kategorie.

- Beispiel natürliche Zahlen: -1, 0, 1 (und beliebige positive Werte)
- Beispiel Liste: leere Liste (null), einelementige Liste (und diverse nicht-leere Listen)
- Beispiel Array: Länge Null, Länge 1 (und diverse größere Arrays)
- Beispiel String: null, Länge Null, Länge 1 (und größere Längen)

Diese Art des Testens beruht auf einem intuitiven Fehlermodell: An den Grenzfällen machen Programmierer Fehler (off by one!).

Testen

Beispiel

Kategorien für das Testen eines Sortieralgorithmus:

- Länge (null, 0, 1, > 1)
- Sortierung (aufsteigend, absteigend, identische Werte, nein)
- Einschluss negativer Elemente (einige negative, alle negativ, alle positiv)
- ...

Testen

Und wie teste ich jetzt?

- Simple Variante: im Programm verstreute main-Methoden mit *System.out.println()* (später wieder entfernen)
- Statisches Codeanalyse: PMD, Checkstyle, Entwicklungsumgebung, . . .
- Assertions & JUnit

Assertions

Allgemein

Zusicherungen (engl. Assertions) sind ein weiteres wichtiges Hilfsmittel für die Entwicklung von Software.
Sie werden verwendet, um

- Restriktionen für Parameter und globale Variablen von Methoden anzugeben (Vorbedingung)
- den Effekt von Methoden formal zu beschreiben (Nachbedingung)
- an zentralen Programmpunkten wichtige Eigenschaften (z.B. mögliche Werte von Variablen) explizit festzuhalten

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

Keine Verwendung in diesem Modul!

Assertions

Java

- Java bietet die Möglichkeit Assertions direkt in den Programmtext zu schreiben (`assert`)
- Automatisierte Überprüfung, aber nur während der Laufzeit
- Eingeschränkt durch Java-Syntax im Vergleich zu Kommentaren
- Assertions werden standardmäßig nicht ausgewertet. Sie müssen beim Programmstart mit `java -ea MyClass` aktiviert werden (**e**nable **a**ssertions)

Syntax

```
assert Boolesche Bedingung;
assert Boolesche Bedingung: "Detaillierte Fehlermeldung";
```

Assertions

Beispiel

```
double multiplyScalar(double[] u, double[] v) {
    assert u.length == v.length; // Precondition
    double s= 0;
    for (int i = 0; i < u.length; i++) {
        assert s == sum(i, u, v); // Invariante
        s += u[i] * v[i];
    }
    assert s == sum(u.length - 1, u, v); // Postcondition
    return s;
}
```

Assertions

Eigenschaften

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

- Eine **Precondition** muss vor der Abarbeitung des entsprechenden Methodenrumpfs erfüllt sein
- Eine **Postcondition** muss erfüllt sein, nachdem die Methode abgearbeitet wurde
- Eine **Instanz-Invariante** ist eine Zusicherung, die sowohl vor als auch nach jedem Methodenaufruf (Ausnahme: private Hilfsmethoden) des zugehörigen Objekts gültig sein muss. Beispiel: `int length; assert length >= 0;`
- Eine **Schleifen-Invariante** ist eine Zusicherung, die am Anfang und Ende eines jeden Durchlaufes der zugehörigen Schleife erfüllt sein muss

Assertions

assert vs if

```
void setBalance(double b) {
    assert (b >= 0);
    this.balance = b;
}

void setBalance(double b) throws IllegalArgumentException {
    if (b < 0) {
        throw new IllegalArgumentException();
    }
    this.balance = b;
}
```

Assertions

assert

- Dokumentation: Nur Überprüfungszweck
- Zur eigentlichen Laufzeit abschaltbar

if

- wird immer ausgeführt
- ggf. teuer zur Laufzeit

Lambda-Ausdrücke

Anonyme Klassen

- Manchmal möchte man ein einmaliges Objekt erschaffen
  - “Anonyme Klasse”
  - Erweitern existierende Klassen

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button geklickt!");
    }
});
```

Lambda Ausdrücke

- Kompakte Schreibweise einer anonymen Klasse
- Syntax:

```
(Parameterliste)    -> Ausdruck
()                  -> Ausdruck
(Parameterliste)    -> {
                        Ausdruck1;
                        ...
                        return Wert; // Optional, nur wenn es einen
                          R ckgabetyp geben soll
                      }
```

Beispiel (1/2)

```
List<Integer> list = new ArrayList:>();
// Add some elements//.

// ------ 1. Variante: Anonyme Klasse ------
list.sort(new Comparator<Integer>() {
    @Override
    public int compare(Integer x, Integer y) {
        return y - x;
    }
});

// ------ 2. Variante: Lambda Ausdruck ------
list.sort( (x, y) -> y - x );
```

Beispiel (2/2)

```java
// ------ 3. Variante: Lambda Ausdruck als Variable ------
Comparator<Integer> reverseIntegerComparator = (x, y) -> (y - x);
list.sort(reverseIntegerComparator);

// ------ 4. Variante: Wiederverwenden von anderen Methoden ------
list.sort(MyClass::compareIntegersReversed);
```

Parameter und Rückgabetyp müssen stimmen! (und static!)

```java
public class MyClass {

    public static int compareIntegersReversed(int x, int y) {
        return y - x;
    }

}
```

Aufgabe 1

Wie sieht die Methodenreferenz (Method Reference) für folgenden Lambda-Ausdruck aus?

Ausdruck: `(str) -> new ArrayList<>(str)`

1. `new::ArrayList`
2. `ArrayList::new`
3. `ArrayList()::new`
4. `create::ArrayList()`
5. `ArrayList::create`
6. `new::ArrayList()`

Aufgabe 1

Wie sieht die Methodenreferenz (Method Reference) für folgenden Lambda-Ausdruck aus?

Ausdruck: `(str) -> new ArrayList<>(str)`

1. `new::ArrayList`
2. `ArrayList::new`
3. `ArrayList()::new`
4. `create::ArrayList()`
5. `ArrayList::create` → wäre okay, wenn es so eine Methode gäbe
6. `new::ArrayList()`

Aufgabe 2

```java
import java.util.*;

public class Aufgabe {
    public static void main(String[] args) {
        List<String> cities = Arrays.asList("Berlin", "Ulm", "M nchen", "
          D sseldorf");

        //Sortiere nach L nge des Strings (s1, s2).
        Collections.sort(cities, _______);

        System.out.println(cities);
    }
}
```

Aufgabe 2

Lösung

- `(s1, s2) -> Integer.compare(s1.length(), s2.length())`
- Auch möglich:
  - `(s1, s2) -> s1.length() - s2.length()`
  - `‘Comparator.comparingInt(String::length)`

Aufgabe 3

Aufgabe

Ladet euch die Datei ArrayUtils.java aus Ilias runter und schreibt Testfälle für die Methoden. Welches Fehlverhalten könnt ihr finden?

Lösung 3

Funktion	Fehlverhalten	Sinnvolle Testfälle
`findFirst()`	■ Berechnet die i-te Stelle, nicht Index i	■ Standard ■ Kein passendes Element im Array ■ Array ist `null`
`findAll()`	■ Kein passendes Element → Gibt leeres Array zurück ■ Findet Elemente ganz am Anfang des Arrays nicht	■ Wie findFirst() ■ Nur gesuchte Elemente ■ Gesuchtes Element ganz am Anfang / Ende
`incrementAll()`	■ -	■ Array ist `null` ■ Increment: -1, 0, 1, Irgendeine “mittlere” Zahl, `Integer.MAX_VALUE`

Lösung 3

Funktion	Fehlverhalten	Sinnvolle Testfälle
`insert()`	■ Schlägt fehl, falls das Array `null` ist	■ Array: `null`, length = 0, Standard
`sortedJoin()`	■ Funktioniert nur mit sortierten Arrays ■ Endlosschleife, falls zwei Elemente gleich	■ Wie bei `insert()` ■ Arrays mit gleicher/ unterschiedlicher Länge ■ Unsortiertes Array ■ Values: -1, 0, 1, zwei gleiche
`map()`	■ Wendet die Funktion nicht auf das letzte Element an	■ Array: Wie bei `insert()` ■ Function: `null`, Addition, Multiplikation, Division (by 0), Alles auf eine Zahl setzen, eine invalide Funktion

Optional<T>

Was und warum?

- Container Objekt, das vielleicht ein Objekt vom Typen T enthält
- Ist dafür da um `null` Checks zu verhindern
- Ist meist Rückgabetyp von Methoden
- Beispiel: Eine API Abfrage gibt ein gültiges Ergebnis oder null zurück
- Man könnte stattdessen `Optional<Response>` zurück geben
- Hier ist die Dokumentation

Optional<T>

Was und warum?

- Container Objekt, das vielleicht ein Objekt vom Typen T enthält
- Ist dafür da um `null` Checks zu verhindern
- Ist meist Rückgabetyp von Methoden
- Beispiel: Eine API Abfrage gibt ein gültiges Ergebnis oder null zurück
- Man könnte stattdessen `Optional<Response>` zurück geben
- Hier ist die Dokumentation

Aber das klingt ja irgendwie nichtdeterministisch...

Optional<T>

Was und warum?

- Container Objekt, das vielleicht ein Objekt vom Typen T enthält
- Ist dafür da um `null` Checks zu verhindern
- Ist meist Rückgabetyp von Methoden
- Beispiel: Eine API Abfrage gibt ein gültiges Ergebnis oder null zurück
- Man könnte stattdessen `Optional<Response>` zurück geben
- Hier ist die Dokumentation

Aber das klingt ja irgendwie nichtdeterministisch... Ist es aber nicht!

Optional<T>

Methoden von Optional

- `static Optional<T> empty()` Gibt ein neues leeres Optional zurück. Das ist quasi äquivalent zu `null`
- `static Optional<T> of(T value)` Erzeugt ein neues Optional wo `value` drin ist
- `boolean isPresent()` Gibt `true` zurück wenn das Optional nicht leer ist
- `boolean isEmpty()` Gegenteil von `isPresent()`
- `T get()` Gibt zurück was im Optional drin ist. Achtung: Diese Methode darf nicht benutzt werden ohne vorher zu checken ob etwas im Optional drin ist! Lieber nächste Methode verweden...
- `T orElseThrow()` Gibt den Wert im Optional zurück, falls was drin ist. Auch hier muss man vorher checken ob das der Fall ist.
- `void ifPresent(Consumer<? super T> action)` Führt das lambda aus falls das Optional nicht leer ist
- Noch mehr sehr nützliche Methoden...

Optional<T>

Ein Code Beispiel

```
private Optional<Response> getResponse(){
    //response might be null here
    Response response = ...;
    //this returns the response wrapped in an Optional or an empty Optional
    return new Optional.ofNullable(response);
}
//get the response or an empty Optional
Optional<Response> response = getResponse();
//run lambda function if response is present
response.ifPresent(res -> {
    switch(res.getStatusCode()){
        //Do anything here
    }
});
```