Programmieren in Java

Vorlesung 05: Testen

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2017

Inhalt

Korrektur: Komplexität von ArrayList

Implementierung von ArrayList

```
1  public class ArrayList<E> {
2    private int size;
3    private E[] elements;
4    // invariant: 0 <= size <= elements.length
5
6    boolean add(E elem) {
7      if (size == elements.length) {
8        int newLength = 2 * size;
9        E[] nextElements = new E[newLength];
10       for (int i = 0; i < size; i++) {
11         nextElements[i] = elements[i];
12       }
13       elements = nextElements;
14     }
15     elements[size] = elem;
16     size++;
17     return true;
18   }
19 }
```

Korrektur — Fortsetzung

```
1 E get(int index) {
2   if (index < 0 || index >= size) {
3     throw new IndexOutOfBoundsException("ArrayList.get");
4   }
5   return elements[index];
6 }
```

Komplexität (ArrayList)

- **get** — $O(1)$
- **add** — $O(1)$ amortisiert
  d.h. das Einfügen von n Elementen braucht $O(n)$ Zeit

Komplexität (LinkedList)

- **get** — $O(n)$
- **add** — $O(1)$

Aus der Praxis

Codebeispiele

- `ListOperations.java`
- `anagram.java`
- `caesar code.java`
- `timetowords.java`

Best practice

ListOperations

- Kleine Methoden; eine Funktionalitaet pro Methode
- Magische Konstanten
- Regex und `switch` sind deine Freunde

Best practice

ListOperations

- Kleine Methoden; eine Funktionalitaet pro Methode
- Magische Konstanten
- Regex und `switch` sind deine Freunde

anagram

- `nextLine()` vs. `next()`
- Exceptions nicht zum Spass

Best practice

ListOperations

- Kleine Methoden; eine Funktionalitaet pro Methode
- Magische Konstanten
- Regex und `switch` sind deine Freunde

anagram

- `nextLine()` vs. `next()`
- Exceptions nicht zum Spass

caesar code

- Casts zwischen `char` und `int` verwenden!

Best practice

ListOperations

- Kleine Methoden; eine Funktionalitaet pro Methode
- Magische Konstanten
- Regex und `switch` sind deine Freunde

anagram

- `nextLine()` vs. `next()`
- Exceptions nicht zum Spass

caesar code

- Casts zwischen `char` und `int` verwenden!

timetowords

- Spass, oder?

Testen

Testen

Ziel des Testens

Gibt es Fehler?

Definition

Unter Testen versteht man den Prozess des Planens, der Vorbereitung und der Messung, mit dem Ziel, die Eigenschaften eines IT-Systems festzustellen und den Unterschied zwischen dem tatsächlichen und dem erforderlichen Zustand aufzuzeigen.

Testen

Ziel des Testens

Gibt es Fehler?

Definition

Unter Testen versteht man den Prozess des Planens, der Vorbereitung und der Messung, mit dem Ziel, die Eigenschaften eines IT-Systems festzustellen und den Unterschied zwischen dem tatsächlichen und dem erforderlichen Zustand aufzuzeigen.

Durchführung des Tests

- Ausführen eines Programms mit der Absicht Fehler zu finden
- Keine Garantie für Korrektheit des Programms

Testen

Ziel des Testens

Gibt es Fehler?

Definition

Unter Testen versteht man den Prozess des Planens, der Vorbereitung und der Messung, mit dem Ziel, die Eigenschaften eines IT-Systems festzustellen und den Unterschied zwischen dem tatsächlichen und dem erforderlichen Zustand aufzuzeigen.

Durchführung des Tests

- Ausführen eines Programms mit der Absicht Fehler zu finden
- Keine Garantie für Korrektheit des Programms

Edsger W. Dijkstra (Turing Award 1972)

Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.

Arten des Testens

- Funktionale Eigenschaften
  - Komponententest
  - Integrationstest
  - Systemtest
  - Abnahmetest
  - Regressionstest
- Nicht-funktionale Eigenschaften
  - Performancetest
  - Lasttest
  - Stresstest
  - Benutzbarkeitstest
  - Wartbarkeitstest
  - Zuverlässigkeitstest
  - Portabilitätstest

Funktionale Eigenschaften — Komponententest

Fragestellung

Verhält sich der Code gemäß seiner funktionalen Spezifikation?

Prüfung

- Tester implementiert ein Orakel
- Orakel erhält Eingabe und die berechnete Ausgabe
- Entscheidet ob die beobachtete Ein-/Ausgabe die Spezifikation erfüllt

Güte eines Tests

Auswahl der Eingaben

- Unmöglich sämtliche Eingaben zu testen!
- Es reicht, Äquivalenzklassen von Eingaben zu testen, die gleich behandelt werden
- Bestimmung der Äquivalenzklassen i.a. ein unlösbares Problem :-(

Güte eines Tests — Abdeckung (coverage)

- Funktionsabdeckung:
  jede Funktion wird durch mindestens einen Testfall aufgerufen
- Zeilenabdeckung:
  jede Zeile wird durch mindestens einen Testfall ausgefuehrt
- Entscheidungsabdeckung: jeder Zweig einer Bedingung muss durch mindestens einen Testfall ausgefuehrt werden
- Bedingungsabdeckung:
  jede Teilbedingung muss durch mindestens einen Testfall `true` bzw. `false` werden
- Flugsoftware muss MC/DC (modified condition/decision coverage) erfüllen; trotzdem sind Fehler vorhanden

Hier: Nur Komponententest

- Komponententest mit JUnit4 (siehe Anleitung)
- Spezifikation: Aufgabentext
- Orakel: bereitgestellt oder selbst
- Zusätzlich: Zufallstests (gegen Orakel oder Eigenschaften)

Warum JUnit?

Idee

Schreibe Code, der die Tests ausführt

Warum?

- Große Testsuites möglich
- Tests können zusammen mit dem Produktionscode abgelegt werden
- Nach einer Änderung können die Tests einfach wiederholt werden (Fehler behoben?)
- Nach einer Erweiterung können (alte) Tests einfach wiederholt werden (Regressionstest, keine neuen Fehler eingeführt?)

Anatomie eines Testfalls

set up Initialisierung der Testumgebung (fixture)

— zu testenden Code aufrufen

— Ergebnis mit Orakel prüfen

tear down Finalisierung der Testumgebung, Freigabe von Ressourcen, etc

Basic JUnit Usage

```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3
4  public class Ex1Test {
5    @Test
6    public void testFind_min() {
7      int[] a = {5, 1, 7};
8      int res = Ex1.find_min(a);
9      assertEquals(1, res);
10   }
11
12   @Test
13   public void testInsert() {
14     int x[] = {2, 7};
15     int n = 6;
16     int res[] = Ex1.insert(x, n);
17     int expected[] = {2, 6, 7};
18     assertArrayEquals(expected, res);
19   }
20 }
```

Beispiel Testorakel

vgl. Aufgabe line-intersection

```
1 public boolean checkIntersection(
2   double[] computedResult, // null or array of length 2
3   double a1, double b1, // line 1: y = a1*x + b1
4   double a2, double b2, // line 2: y = a2*x + b2
5   double eps // allowed rounding error
6 )
```

Sollte prüfen

1. `computedResult` ist `null` gdw. line1 und line2 parallel
2. `computedResult` ist zwei-elementiges Array mit den Koordinaten eines Punktes $\{x_0, y_0\}$ gdw. ein Punkt (x_1, y_1) existiert, der sowohl auf line1 als auch auf line2 liegt und $x_0 \approx x_1$ und $y_0 \approx y_1$ bis auf Rundungsfehler kleiner gleich `eps`.

Beispiel Testorakel

vgl. Aufgabe line-intersection

```
1 public boolean checkIntersection(
2   double[] computedResult, // null or array of length 2
3   double a1, double b1, // line 1: y = a1*x + b1
4   double a2, double b2, // line 2: y = a2*x + b2
5   double eps // allowed rounding error
6 )
```

Sollte prüfen

1. `computedResult` ist `null` gdw. line1 und line2 parallel
2. `computedResult` ist zwei-elementiges Array mit den Koordinaten eines Punktes $\{x_0, y_0\}$ gdw. ein Punkt (x_1, y_1) existiert, der sowohl auf line1 als auch auf line2 liegt und $x_0 \approx x_1$ und $y_0 \approx y_1$ bis auf Rundungsfehler kleiner gleich `eps`.

Aufgabe: Konstruiere Eingabe, die Rundungsfehler hervorruft

Anzahl der Tests pro Testmethode

Best practise

Only one test per test case method.

Preamble – Fixture

- Often several tests need to set up in the same or a similar way.
- This common setup of a set of tests is called *preamble*, or *fixture*.
- Write submethods which perform the common setup, and which are called from each test case.
- A slightly more convenient (but less flexible) way is to use the JUnit `@Before` and `@After` annotations. Thus annotated methods run before and after *each test case*.

Abnormal Termination

- JUnit propagates the result of an assertion by throwing an exception.
- Default treatment: report *failure* if the IUT throws an exception.
- Most of the time: correct behavior (no unhandled exceptions in the IUT).
- To override this behaviour, there are two options:
  - Catch and analyse exceptions thrown by IUT in the test case method, or
  - Give an `expected` optional element of the `@Test` annotation.

Exceptions – Example

Exception means failure:

```
1  @Test public void test_find_min_1() {
2    int[] a = {};
3    int res = Ex1.find_min(a);
4  }
```

Exceptions – Example

Exception means failure:

```
1  @Test public void test_find_min_1() {
2    int[] a = {};
3    int res = Ex1.find_min(a);
4  }
```

Exception means success:

```
1  @Test(expected=Exception.class) public void test_find_min_1() {
2    int[] a = {};
3    int res = Ex1.find_min(a);
4  }
```

Enums

Enum — Aufzählungstypen

```
1 public enum PartialOrdering {
2   LESS, EQUAL, GREATER, INCOMPARABLE
3 }
```

- Erzeugt einen neuen (Referenz-) Typ mit genau den aufgelisteten Elementen
- `PartialOrdering` verhält sich wie eine Klasse
- Können im Code als Konstanten verwendet haben
- Verwendbar im `switch`
- Vergleich mit ==
- Weitere Methoden siehe Dokumentation von java.lang.Enum
- Weitere Möglichkeiten siehe Enum Tutorial

Exceptions

Exceptions

- Eine Java Methode kann einen Fehlerzustand durch eine *Exception* melden.
- Eine Exception wird durch eine **throw**-Anweisung ausgelöst und kann durch eine **catch**-Anweisung abgefangen werden.
- Jede Exception wird durch ein *Exception-Objekt* repräsentiert

$\Rightarrow$ Die **throw**-Anweisung nimmt ein Exception-Objekt als Parameter

- Konvention: vermeide Exceptions in Konstruktoren

Arten von Fehlern

Java unterscheidet 3 Arten von Fehlern:

1. Ausnahmen, auf die man nur wenig Einfluss nehmen kann:
   - Kein Speicher mehr verfügbar
   - Stack overflow (rekursive Aufrufe)
2. Ausnahmen, die durch korrekte Programmierung vermeidbar sind:
   - Dereferenzierung von **null**
   - Arrayzugriff außerhalb der Grenzen
3. Ausnahmen, die nicht vermeidbar sind, aber behandelt werden können:
   Alternativer Rückgabewert
   - Datei kann nicht gelesen werden
   - Netzwerkverbindung ist gestört

Klassifizierung in Checked/Unchecked Exceptions

Art der Fehler:

- durch korrekte Programmierung vermeidbar
- unvermeidbar aber behandelbar

Klassifizierung in Checked/Unchecked Exceptions

Art der Fehler:

- durch korrekte Programmierung vermeidbar
- unvermeidbar aber behandelbar

Vermeidbare Fehler werfen **unchecked** Exceptions

- Objekt nicht initialisiert (NullPointerException)
- Division durch Null (ArithmeticException)

Subklassen von `RuntimeException`

Klassifizierung in Checked/Unchecked Exceptions

Art der Fehler:

- durch korrekte Programmierung vermeidbar
- unvermeidbar aber behandelbar

Vermeidbare Fehler werfen **unchecked** Exceptions

- Objekt nicht initialisiert (NullPointerException)
- Division durch Null (ArithmeticException)

Subklassen von `RuntimeException`

Unvermeidbare Fehler werfen **checked** Exceptions

- Datei wurde nicht gefunden (FileNotFoundException)
- Allgemeine Exception für Servlets (ServletException)

Subklassen von `Exception` mit Ausnahme von `RuntimeException`

Checked/Unchecked Exceptions

- Checked Exceptions müssen in der Signatur der Methode aufgeführt werden

```
1 String readLineFromFile(String filename) throws FileNotFoundException {
2     // ...
3 }
```

- Compiler und Programmierer wissen, dass eine FileNotFoundException geworfen werden kann
- Programmierer kann angemessen darauf reagieren: Zum Beispiel Hinweis an Benutzer, dass die Datei nicht existiert

Fangen einer Exception

Fangen einer Exception:

```
1 try { /* Code der Ausnahme produziert */ } catch (Throwable e) { /* Behandlung */ }
```

- Es *kann* jede Subklasse von Throwable gefangen werden
- Checked und unchecked Exceptions können gefangen werden
- Es können mehrere **catch** hintereinander stehen

Fangen einer Exception

Fangen einer Exception:

```
1 try { /* Code der Ausnahme produziert */ } catch (Throwable e) { /* Behandlung */ }
```

- Es *kann* jede Subklasse von Throwable gefangen werden
- Checked und unchecked Exceptions können gefangen werden
- Es können mehrere **catch** hintereinander stehen

Beispiel:

```
1 try {
2   readLineFromFile("invalid file name");
3 } catch (FileNotFoundException e) {
4     System.out.println("File does not exist, please choose another one!");
5 } catch (AnotherException e) {
6     // ...
7 }
```

Exceptions behandeln

Ist es sinnvoll jede Exception zu fangen?

Exceptions nur fangen, wenn man die Fehlersituation bereinigen oder melden kann.

Exceptions behandeln

Ist es sinnvoll jede Exception zu fangen?

> **Exceptions nur fangen, wenn man die Fehlersituation bereinigen oder melden kann.**

Checked Behandeln, Benutzer über Fehler informieren[1], und ggf. nochmals mit neuem Wert versuchen

Unchecked Möglichkeiten:

- Fehler im Source beheben
- Benutzer informieren **und** Programmierer informieren: Stacktrace, Logausgabe, Bugreport
- Ignorieren, d.h. kein try-catch

[1] Natürlich mit klarer Fehlerbeschreibung

Exceptions behandeln

Ist es sinnvoll jede Exception zu fangen?

> **Exceptions nur fangen, wenn man die Fehlersituation bereinigen oder melden kann.**

Checked Behandeln, Benutzer über Fehler informieren[1], und ggf. nochmals mit neuem Wert versuchen

Unchecked Möglichkeiten:

- Fehler im Source beheben
- Benutzer informieren **und** Programmierer informieren: Stacktrace, Logausgabe, Bugreport
- Ignorieren, d.h. kein try-catch

Niemals Exceptions fangen, und "verschlucken":

```
1 try { callMethod(); } catch (Exception e) { }
```

[1] Natürlich mit klarer Fehlerbeschreibung

Next Up: Klassen und Objekte

Fragen