Fragen, Antworten, Kommentare zur aktuellen Vorlesung

Die Online-Befragung zur genutzten alternativen Veranstaltungsform und zur Lehrevaluation ist online. Bitte ausfüllen: https://forms.gle/fZTBWPmUhK4oqLHw5. Sie werden eventuell aufgefordert sich bei Google anzumelden, das ist nur notwendig, wenn Sie in der Bearbeitung eine Pause machen und das Teilergebnis zwischenspeichern wollen. Die Befragung endet am 19.12., die Ergebnisse stehen in einem nachfolgenden Fragen&Antworten-Dokument auf der Webseite der Veranstaltung.

Hinweis: Unter https://youtu.be/WNw3JbP2RCg finden Sie die Erklärungen von Beispiellösungen zu zwei Teilaufgaben der Messreihe Teil 2 online. Weiterhin gibt es ebenfalls ein Videos zur Aufgabe mit dem interaktiven Interaktionsbrett unter https://youtu.be/wVNGKb6U_ME und zum Basketballspiel in https://youtu.be/YhvkBYtCyZ4.

Hinweise:

- Die Unterlagen der letzten Veranstaltungen sind als vorzeitiges Weihnachtsgeschenk online (natürlich ohne die zeitnahen Fragen&Antworten-Dokumente).
- Eine Beispielklausur mit Online-Lösungsvideoversuch mit Korrektur ist schon etwas länger online. Die Klausurstruktur wird übernommen, die letzte Aufgabe kann beliebig ersetzt werden.
- Alle Inhalte der Vorlesung können in der Klausur vorkommen (als letzte Aufgabe).
- Das letzte Aufgabenblatt, das abgenommen wird ist Blatt 11, Fragen zu anderen Blättern können natürlich gestellt werden.

Frage: Ich habe beim googeln über Interfaces gesehen, das da auch eine Programmierung drin stattfinden kann. Das finde ich nicht in den Vorlesungsfolien.

Antwort: Das stimmt, dies ist nicht Teil der Veranstaltung, da die ergoogelte Idee nicht für die klassische Objektorientierung steht und Java-spezifisch ist. Zentrale Bedeutung hat das vorgestellte Interface-Konzept als vollständig abstrakte Klasse, die so als Vorgabe zur Realisierung dient und flexibel gegen andere Programmierungen ausgetauscht werden kann.

Das Schlüsselwort Interface gibt es nebenbei in C++ selbst nicht, aber die Idee als vollständig abstrakte Klasse findet sich dort genauso wieder und hat die gleiche elementare Bedeutung wie in anderen objektorientierten Sprachen.

In Java wurde das Interface-Konzept aufgeweicht, da für neue Java-Versionen der Wunsch entstand neue Methoden in Interfaces zu packen. Wird dies durch eine einfache Ergänzung gemacht, müssten in allen Klassen, die das Interface realisieren, diese Methode neu implementiert werden. Damit wäre eine neue Java-Version inkompatibel mit der vorherigen, was immer eine schlechte Idee ist (fragen sie PHP oder Python). Der leicht schmuddelige Trick ist, einfach zur neuen Methode doch eine konkrete Implementierung anzugeben, die mit dem Schlüsselwort default gekennzeichnet wird. Diese Methode kann dann in der realisierenden Klasse implementiert also überschrieben werden, muss es aber nicht. Das folgende Beispiel zeigt dies und noch eine Ergänzung.

```
public interface MeinInterface {

  public void zeich42();  // abstrakt, muss realisiert werden

  default public void zeich43() { // mit Standard-Implementierung
    System.out.println("XLIII");
  }

  default public void zeich44() {
    System.out.println("XLIV");
  }

  public static void klassenmethode() {
     System.out.println("geht in Java-Interface, "
         + "hat aber mit Vererbung nix zu tun.");
  }
}

public class MeineRealisierung implements MeinInterface{

  public MeineRealisierung(){
  }

  @Override
  public void zeich42(){
    System.out.println("42");
  }

  @Override
  public void zeich44(){
    System.out.println("44");
  }

  public static void klassenmethode() {
     System.out.println("geht in Java-Interface, "
         + "hat aber mit Vererbung nix zu tun.");
  }
}


public class Main {
  public static void main(String[] s) {
    MeinInterface mi = new MeineRealisierung();
    mi.zeich42();
    mi.zeich43();
    mi.zeich44();
    MeineRealisierung.klassenmethode(); //unsauber
    MeinInterface.klassenmethode(); // besser
  }
}
```

Die Ausgabe lautet:

```
42
XLIII
44
geht in Java-Interface, hat aber mit Vererbung nix zu tun.
geht in Java-Interface, hat aber mit Vererbung nix zu tun.
```

Weiterführend für Fortgeschrittenere: Die Erstellung von Tests ist eine zentrale Aufgabe während der Programmentwicklung. Tests können auch vor der Entwicklung geschrieben werden, dadurch besteht die Möglichkeit das zu programmierende Verhalten genau zu beschreiben. Wie immer bei Testfällen werden typische Fälle und jedwedes mögliche Randverhalten in Tests umgesetzt. Danach wird inkrementell die Software entwickelt und es sollten Schritt für Schritt mehr Tests grün werden. Da einige Tests laufen werden, auch wenn das zu entwickelnde Programm nichts macht, wird die Zahl der laufenden Tests nicht unbedingt kontinuierlich anwachsen. Das folgende Beispiel gibt einen kleinen unvollständigen Ausblick, was noch mit JUnit möglich ist. Beachten Sie, dass es unüblich ist Ausgaben in Tests zu programmieren, dies erfolgt hier zur Anschauung. Die zu testende Klasse hat eine Methode, die Zahlen addiert.

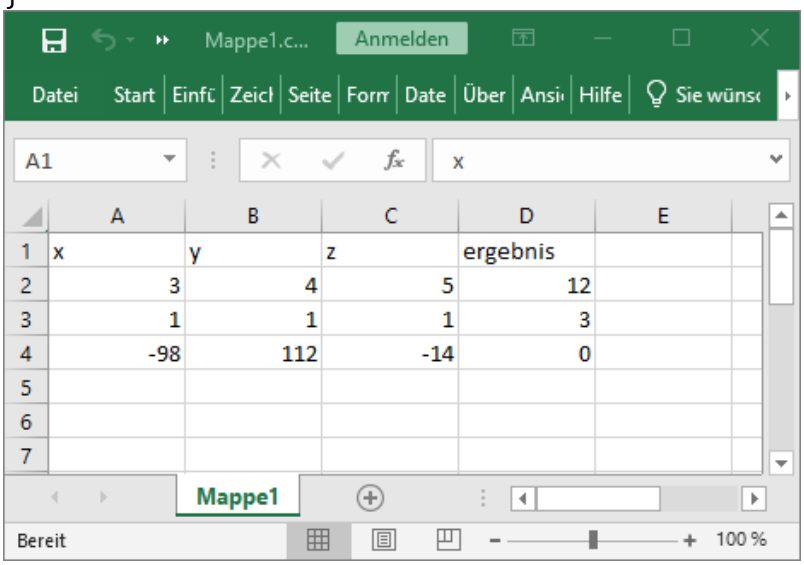
```
public class Adder {
  public int sum(int x, int y, int z) {
    return x + y + z;
  }
}
```

Die Beispieltestklasse sieht wie folgt aus.

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Assumptions;

import java.util.List;
import java.util.stream.Stream;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.CsvFileSource;
import org.junit.jupiter.params.provider.MethodSource;

public class AdderTest {

  private Adder sut; // System under Test
  private Adder sut2; // kann beliebig viele Testobjekte geben

  @BeforeAll
  public static void setUpBeforeClass() {
    //Dies wird einmal am Start fuer alle Tests gemacht
  }
```

```java
@AfterAll
public static void tearDownAfterClass() {
  //Dies wird einmal am Ende fuer alle Tests gemacht
}

@BeforeEach
public void setUp() {
  //Dies wird einmal vor jedem Test gemacht
  this.sut = new Adder();
  this.sut2 = new Adder();
}

@AfterEach
public void tearDown() {
  //Dies wird einmal mach jedem Test gemacht
}

@Test
void testSumErwartet() {
  System.out.println("test1");
  int erg = this.sut.sum(21, 21, 0);
  Assertions.assertEquals(42, erg
      , "Erwartet wurde 42, gefunden: " + erg);
}

@Test
void testSumVertauschbar() {
  System.out.println("test2");
  Assertions.assertEquals(this.sut.sum(1, 2, -1)
      , this.sut2.sum(-1, 1, 2));
}

@Test
void testSumMitAnnahme() {
  System.out.println("test3");
  // Test nur ausfuehren, wenn Annahme erfuellt
  Assumptions.assumeTrue(1 + 2 + 3 == 6);
  int erg = this.sut.sum(1, 2, 3);
  Assertions.assertEquals(6, erg
      , "Erwartet wurde 42, gefunden: " + erg);
}

// Stellen Sie sich Stream einfach als List vor; der
// ebenfalls mit einem Iterator alle Elemente durchlaufen kann.
public static Stream<Arguments> daten(){
  List<Arguments> testdaten = List.of(
      Arguments.of(0, 0, 0, 0),
      Arguments.of(-2, -1, -3, -6),
      Arguments.of(6, 7, 29, 42)
  );
  return testdaten.stream();
}
```

```java
  @ParameterizedTest
  @MethodSource({"daten"})
  public void testSumParametrisiert(int a, int b, int c, int erg) {
    System.out.println("Testpar mit " + a + "," + b + "," + c);
    Assertions.assertEquals(erg, this.sut.sum(a, b, c));
  }

  @ParameterizedTest
  @CsvFileSource(resources ={"/Mappe1.csv"}, numLinesToSkip = 1)
  public void testSumMitExcelSheet(int a, int b, int c, int erg) {
    System.out.println("Testcsv mit " + a + "," + b + "," + c);
    Assertions.assertEquals(erg, this.sut.sum(a, b, c));
  }
}
```



Die resultierende Ausgabe sieht wie folgt aus.