MSc/ICY Software Workshop
Testing (Revisited)
Functions
Interfaces

Manfred Kerber www.cs.bham.ac.uk/~mmk

Tuesday 29 October 2019

Testing

(partly based on material by Christoph Lange and Chris Bowers)

How to test?

- All the code should be covered by tests (e.g. both branches of an if-then-else). That is, you want at the very least a 100% coverage. Much better is to test every combination of code runs, e.g., if you had 2 if-then-else constructs test "then1 with then2", "then1 with else2", "then2 with else1", and "then2 with else2". [This can result in many cases.]
- You should test sufficiently many *typical* cases.
- You should test *border* cases (e.g. for an array `a` whether `a[0]` and `a[a.length-1]` are properly initialized/changed/used).
- You should write your code so that it is testable as far as possible (e.g., do not write a sophisticated print method since that cannot be tested by JUnit), but write a sophisticated `toString` method (since it can be tested by JUnit). Also subdivide methods into smaller methods.

A quote by E. W. Dijkstra (2000)

A programmer has to be able to demonstrate that his program has the required properties. If this comes as an afterthought, it is all but certain that he won't be able to meet this obligation only if he allows this obligation to influence his design, there is hope that he can meet it. Pure a posteriori verification denies you that wholesome influence and is therefore putting the cart before the horse.

Test-Driven Development

- Program by intent
  1. Start by defining a set of test cases
     - Pay attention to border cases!
  2. Write code that passes the tests
- ```
  public class TestWordStemmer {
    public void testStemmer() {
      WordStemmer stemmer = new WordStemmer ();
      assert stemmer.stem("helping") == "help";
      assert stemmer.stem("hungrily") == "hungry";
      assert stemmer.stem("friendliness") == "friend";
      assert stemmer.stem("play") == "play";
      assert stemmer.stem("playing") == "play";
      assert stemmer.stem("player") == "play";
      // ...
    }
  }
  ```


  ```

# Test-Driven Development (Cont'd)

- Use tests as a template to create code
-

```
public class WordStemmer {
  public String stem(String word) {
    String stemmed_word;
    // do stemming
    return stemmed_word;
  }
}
```

Find the right pattern to make defining test cases easier.

Unit testing

- **Unit** – small functional part of application that can be **tested independently**.
- A unit could be an **individual class or individual method**
- A **set of test cases** are constructed to form a **test harness**

JUnit assertions

Statement	What it does
`fail()`	Lets the test fail. Useful for checking code is not reached under certain conditions.
`assertTrue(boolCond)`	Checks that a condition is true
`assertEquals(expected, actual, [tolerance])`	Checks that two values are the same. Not a deep check, i.e., be careful with non-primitive values.
`assertArrayEquals (expArray, actualArray)`	checks whether two arrays are equal.
`assertNull(object)`	Check that an object is null
`assertNotNull(object)`	Check that an object is not null
`assertSame(expected, actual)`	Check that both object references are the same.
`assertNotSame(expected, actual)`	Check that both object references are not the same.

- All methods take an optional last argument **message**.
- Argument tolerance for tests involving floating point numbers.

JUnit Assertions

```
/* from https://github.com/junit-team/junit/wiki/Assertions */
public class AssertTests {
    @Test
    public void testAssertArrayEquals() {
        byte[] expected = "trial".getBytes();
        byte[] actual = "trial".getBytes();
        assertArrayEquals(expected, actual, "failure - arrays not same"
    }

    @Test
    public void testAssertEquals() {
        assertEquals("text", "text", "failure - strings are not equal")
    }

    @Test
    public void testAssertFalse() {
        assertFalse(false, "failure - should be false");
    }
```


JUnit Assertions (Cont'd)

```
@Test
public void testAssertNotNull() {
    assertNotNull(new Object(), "should not be null");
}

@Test
public void testAssertNotSame() {
    assertNotSame(new Object(), new Object(),
                  "should not be same Object");
}

@Test
public void testAssertNull() {
    assertNull(null, "should be null");
}

@Test
public void testAssertSame() {
    Integer aNumber = Integer.valueOf(768);
    assertSame(aNumber, aNumber, "should be same");
}
```


JUnit Test for Exception

```
in Date.java:
public Date(int day, String month, int year) {
    if (admissible(day, month, year)) {
        this.day = day; ...
    } else {
        throw new
            IllegalArgumentException("Invalid date in Date");
    }
}

in DateTest.java:

@Test public void dateTest11() {
    Exception e = assertThrows(IllegalArgumentException.class,
                                () -> {
            new Date(31, "April", 2019);});
    assertEquals("Invalid date in Date", e.getMessage());
}
```

Test succeeds if correct exception with correct error message is thrown.

JUnit Annotations

Statement	What it does
`@Test`	Marks a test method
`@BeforeEach`	Code that should be executed before each test
`@Disabled`	Ignore this test

JUnit cannot Test Everything

- Input/output is hard to test (need to maintain separate test files, prepare strings/arrays that simulate file contents if code to be tested supports streams).
- GUIs are even harder to test (separation of GUI and underlying logic helps, e.g. model-view-controller design pattern)

More info online:

- JUnit homepage: http://junit.org
- Eclipse JUnit tutorial http://www.vogella.com/articles/JUnit/article.html

JUnit cannot Test Everything

- Input/output is hard to test (need to maintain separate test files, prepare strings/arrays that simulate file contents if code to be tested supports streams).
- GUIs are even harder to test (separation of GUI and underlying logic helps, e.g. model-view-controller design pattern)

More info online:

- JUnit homepage: http://junit.org
- Eclipse JUnit tutorial http://www.vogella.com/articles/JUnit/article.html

Final Point to take home:

- Kiss (Keep it simple, stupid!)
- Design by contract.
- Use design patterns.
- Try to write beautiful code.

Functions

Unlike methods, functions (also called lambda-expressions) can be called as arguments in methods. Syntax example:

```
import java.util.function.Function;
public class FunMain {
  public static void printN(Function<Integer,Integer> f, int n){
      for (int i = 0; i <= n; i++){
          System.out.print(f.apply(i) + " ");
      }
      System.out.println();
  }
  public static final Function<Integer,Integer> f0 =
      x -> {return x * x + x - 7;};
  public static void main(String[] args) {
      printN(x -> {return x * x;}, 10);
      printN(x -> {return x + 1;}, 10);
      printN(f0, 10);
  }
}
```


Function primeNumber

The function determines whether a number is a prime number.

```
/** The function primeNumber returns true if and only if
 *  the number is divisble only by 1 and itself.
 */
public static final Function<Integer,Boolean>
    primeNumber = x -> {
    if (x < 2) {
        return false;
    }
    for (int i = 2; i < x; i++) {
        if (x % i == 0) {
            return false;
        }
    }
    return true;
};
```


Functions - Average Income

Assume we want to compute the average net income, given gross incomes and a tax function:

```
public static Double netAverage(ArrayList<Double> a,
                                Function<Double,Double> tax) {
    double average = 0;
    for (Double gross : a) {
        average += gross - tax.apply(gross);
    }
    return average/a.size();
}

public static Function<Double,Double> tax1 =
    x -> {return (x <= 30000) ? 0.2*x : 0.3*x;};
```


Functions - Average Income (Cont'd)

```
// Tax computed with current income tax rules of the UK:

public static Function<Double,Double> tax1 =
    x -> {
    if (x <= 11850){
        return 0.0;
    }
    else if (x <= 34500){
        return 0.2 * (x - 11850.0);
    }
    else if (x <= 150000){
        return 0.4 * (x - 34500.0) + 0.2 * 34500.0;
    }
    else { // x > 150000
        return 0.45 * (x - 150000.0) +
        0.4 * (150000.0 - 34500.0)   +
        0.2 * 34500.0;
        }
    };
```


Object-Oriented Programming

Distinguish:

- **Classes**, **e.g.**, `BankAccount`, `Customer`
- **Objects**, **e.g.**, `bankAccountJohn`, `customerMary`
  created by a Constructor, e.g.
  `public BankAccount (Customer customer, String password)`
- **Methods**, **e.g.** `getBalance()`

Classes

- Only **one class declaration** in a particular file can be **public**. **It corresponds to the file name.**
- **Classes can be nested. Inner classes are invisible from the outside and corresponding methods cannot be called from the outside.**

Interface

There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, interfaces are such contracts.

from `http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html`

Interface (Cont'd)

Allow to apply same method to unrelated objects, e.g. employees and invoices (as expenditure) The example from [Deitel & Deitel, 2010, p. 427 ff.]

```
public interface Payable {
    int getPaymentAmount();
    // no implementation, only head of method
}
public class Invoice implements Payable{
    // implementation of getPaymentAmount()
    public int getPaymentAmount(){
        ...
        return ...;
    }
}
public class Employee implements Payable{
    // implementation of getPaymentAmount()
    public int getPaymentAmount(){
    ...
    return ...;
    }
```

Structure