MSc/ICY Software Workshop
Testing (Revisited)
Functions
Interfaces

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

26 October 2016

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 (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.

JUnit testing with Eclipse

- JUnit: unit testing library for Java
- Integrated into Eclipse/NetBeans/XCode/... IDEs
- Separates source code from testing code
- IDEs provide lots of nifty inspection tools
- Well documented

Testing

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

How to test?

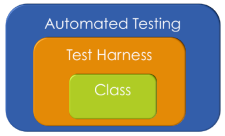
- All the code should be covered by tests (e.g. both branches of an if-then-else).
- You should test sufficiently many *typical* cases.
- You should test border cases (e.g. for an array `a` whether `a[0]` and `a[a.length-1]` are properly initialized/changed).
- You should write your code so that it is 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).

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";
    // ...
  }
}
```


Unit testing

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

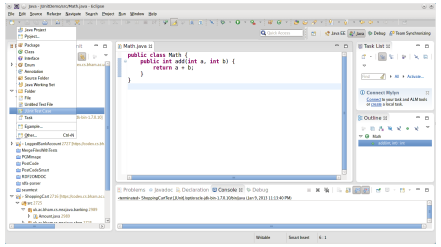


Eclipse example

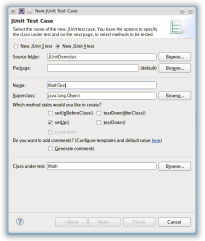
Outline:

1. creating JUnit test
2. writing test cases
3. coding by intent
   1. getting it wrong ⇒ test fails
   2. getting it right ⇒ test passes

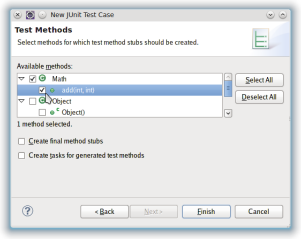
Eclipse example: Creating JUnit test



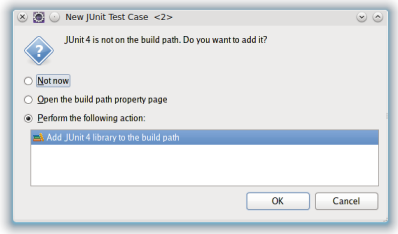
Eclipse example (Cont'd): Creating JUnit test



Eclipse example (Cont'd): Creating JUnit test



Eclipse example (Cont'd): Creating JUnit test



Eclipse example (Cont'd): Creating JUnit test

Math.java MathTest.java

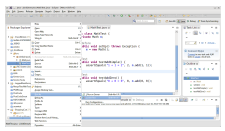
```
import static org.junit.Assert.*;

public class MathTest {

    @Before
    public void setUp() throws Exception {
    }

    @Test
    public void testAdd() {
        fail("Not yet implemented");
    }

}
```


Eclipse example (Cont'd): Writing test cases

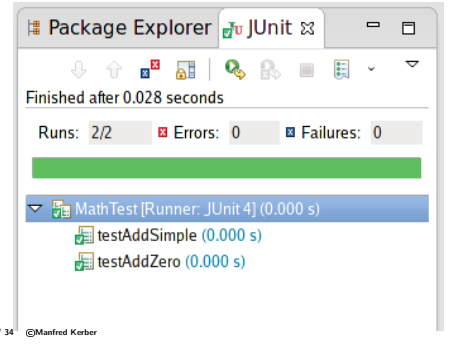
Math.java MathTest.java

```
public class MathTest {
    private Math m;

    @Before
    public void setUp() throws Exception {
        m = new Math();
    }

    @Test
    public void testAddSimple() {
        assertEquals("1 + 1 = 2", 2, m.add(1, 1));
    }

    @Test
    public void testAddZero() {
        assertEquals("0 + 0 = 0", 0, m.add(0, 0));
    }
}
```


Eclipse example (Cont'd): Running a test



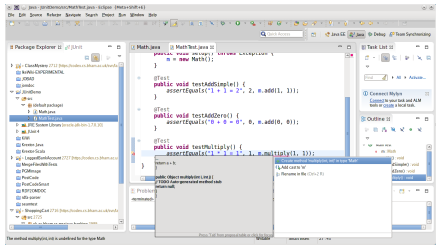
Eclipse example (Cont'd): Running a test



Eclipse example (Cont'd): Test-driven development

```
@Test
public void testMultiply() {
    assertEquals("1 * 1 = 1", 1, m.multiply(1, 1));
}
```

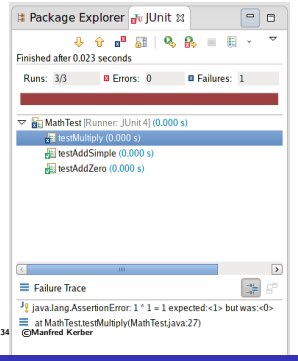

Eclipse example (Cont'd): Test-driven development



Eclipse example (Cont'd): Test-driven development

```
public int multiply(int a, int b) {
    return a - b;
}
```


Eclipse example (Cont'd): Test-driven development



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 if 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 first argument **message**.
- Last 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("failure - arrays not same", expected, actual);
  }

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

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


JUnit Assertions (Cont'd)

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

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

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

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


JUnit Assertions (Cont'd)

```
// JUnit Matchers assertThat
@Test
public void testAssertThatBothContainsString() {
  assertThat("albumen",
        both(containsString("a")).and(containsString("b")));
}

@Test
public void testAssertThathasItemsContainsString() {
  assertThat(Arrays.asList("one", "two", "three"),
             hasItems("one", "three"));
}

@Test
public void testAssertThatEveryItemContainsString() {
 assertThat(Arrays.asList(new String[] {"fun", "ban", "net"}),
            everyItem(containsString("n")));
}
```

JUnit Assertions (Cont'd)

```
@Test
public void testAssertThatHamcrestCoreMatchers() {
 assertThat("good",allOf(equalTo("good"),startsWith("good")));
 assertThat("good",not(allOf(equalTo("bad"),equalTo("good"))));
 assertThat("good", anyOf(equalTo("bad"), equalTo("good")));
 assertThat(7, not(CombinableMatcher.<Integer>
                   either(equalTo(3)).or(equalTo(4))));
 assertThat(new Object(), not(sameInstance(new Object())));
}

@Test
public void testAssertTrue() {
  assertTrue("failure - should be true", true);
 }
}
```


JUnit Annotations

Statement	What it does
`@Test`	Marks a test method
`@Test(timeout=1000)`	Test fails if timeout (in ms) exceeded
`@Test(expException)`	Test fails if defined exception is not thrown
`@Before @After`	Code that should be executed before or after every test
`@BeforeClass @AfterClass`	One-off setup/teardown code (e.g. database login)
`@Ignore`	Ignore this test

Usage:

```
@Test(timeout=1000)
public void underOneSecond() {}
```


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);
  }
}
```


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.**

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:
@Rule public ExpectedException exception =
            ExpectedException.none();
@Test public void dateTest9() {
    exception.expect(IllegalArgumentException.class);
    exception.expectMessage("Invalid date in Date");
    new Date(31, "April", 2016);
}
```

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

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.

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()`

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