Functional Testing
(Testování funkčnosti)

http://d3s.mff.cuni.cz

Pavel Parízek
parizek@d3s.mff.cuni.cz
Software testing

• Purpose
  - Checking whether a given program satisfies certain requirements and expectations about its behavior

• Basic idea
  - Pick specific inputs (a set of values)
  - Run the program for each input
  - Inspect the output and final state

• Shows only presence of errors
  - You can try just few selected input values
Terminology

• Test case
  - Checks single requirement on the program behavior
  - Defines test input and expected output (final state)

• Test suite
  - Collection of related test cases

• Fixture
  - Common environment for test cases in a given suite
When to run tests

- **Development**
  1) Write code and some tests
  2) Run all tests and find bugs
  3) Fix bugs detected by tests
  4) Go to step 1 until deadline

- **Regressions**
  - Execute all passed tests after every modification
    - bug fix, refactoring, new unrelated feature, optimization
  - Goal: check whether everything still works then
Testing on different levels

- **Unit testing**
  - Small components (method, class)
  - Automatic easily repeatable tests
  - Provides clear answer (pass or fail)

- **Integration testing**
  - Checking interaction between components

- **System testing**
  - Whole system in a target environment
  - Requirements specified by customers
Unit testing

• Developers write code that
  ▪ Specifies test inputs and required properties
  ▪ Checks whether all tests successfully passed
    • Comparing expected outputs (and program state) with actual outputs

• Frameworks
  ▪ JUnit, PyUnit, CPPUnit, NUnit, xUnit, MSTest, ...
JUnit

• Unit testing framework for Java
  ▪ [http://junit.org](http://junit.org)
  ▪ [https://github.com/junit-team/junit/wiki](https://github.com/junit-team/junit/wiki)

• Key features
  ▪ Test cases are normal Java methods
  ▪ Test suites are normal Java classes
  ▪ Results analyzed in an automated way

• Versions
  ▪ JUnit 3.8.x: fixed method names, reflection
  ▪ JUnit 4.x: annotations
import java.util.*;
import org.junit.Test;
import static org.junit.Assert.*;

public class TestArrayList {
    @Test
    public void add() {
        List al = new ArrayList();
        int origSz = al.size();
        al.add("abc");
        int newSz = al.size();
        assertEquals("new != orig+1", origSz+1, newSz);
        assertTrue(al.contains("abc"));
    }
}

Assert statements

- public static void assertXY ([message], ...)
- assertEquals(T expected, T actual)
- assertArrayEquals(T[] expected, T[] actual)
- assertSame(Object expected, Object actual)
- assertTrue(boolean condition)
- assertFalse(boolean condition)
- assertNull(Object obj)
- assertNotNull(Object obj)
- assertNull(Object obj)
- fail([String message])
Running tests

- Command line
  
  ```
  java -cp lib/junit-4.11.jar:<dir with tests>
      org.junit.runner.JUnitCore <test class name>
  ```

- Ant
  
  ```
  <target name="run.tests" depends="build.tests">
    <junit haltonfailure="no">
      <formatter type="brief" usefile="false"/>
      <classpath refid="cp.run.tests"/>
      <batchtest>
        <fileset dir="${build.dir}"/>
        <include name="**/Test*.class"/>
      </fileset>
    </junit>
  </target>
  ```
What you should test

- Method contracts (API)
- All branches in the code
- All control-flow paths
- Special (corner) cases
  - “off by one”, bad inputs

- Regressions
  - Inputs triggering previously discovered bugs
Task 1

- Write unit tests for `java.util.ArrayList`
  - Selected methods: `add(o)`, `get(i)`, `remove(i)`, `remove(o)`, `clear()`, `size()`, `contains(o)`

- Try different assert statements

- Create also some failing tests
  - Inspect output of JUnit to see how it typically looks

- JUnit library
  - [http://d3s.mff.cuni.cz/teaching/software_development_tools/files/hamcrest-core-1.3.jar](http://d3s.mff.cuni.cz/teaching/software_development_tools/files/hamcrest-core-1.3.jar)
**Fixture**

- **Goal:** prepare objects in a known state
  - Set up a fixed environment for each test case

- **Reset before each test case ➔ isolated tests**

- **Initialization**
  - `@Before`
  - `@BeforeClass`

- **Clean-up**
  - `@After`
  - `@AfterClass`
import org.junit.*;

public class TestArrayList {
    private List al;

    @Before
    public void setUp() {
        al = new ArrayList();
        al.add("abc");
    }

    @After
    public void tearDown() {
        al = null;
    }

    @Test
    public void add() {
        ...}
}
Expected exceptions

```java
@Test(expected=MyEx.class)
public void testSomething() {
    doSomeOperationThatThrowsException();
}
```
Task 2

- Extend your tests for `ArrayList`

- Define common fixtures
  - Extract duplicate initialization code

- Test against expected exceptions
  - `get(i)`: `IndexOutOfBoundsException`
Recommended practice

- Place tests in the same package as target classes
  - Directory layout
    
    ```
    src/main/cz/cuni/mff/myapp/MyClass.java  
    src/tests/cz/cuni/mff/myapp/TestMyClass.java  
    ```

- Define single assertion in each test method
  - JUnit reports only the first failed assert in a test case
  - Multiple assertions ➔ some failures possibly missed
@RunWith(Parameterized.class)
public class TestSquareRoot {

    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][]{{1,1}, {4,2}});
    }

    private int valInput;
    private int expOutput;

    public TestSquareRoot(int i, int e) {
        valInput = i; expOutput = e;
    }

    @Test
    public void test() {
        assertEquals(expOutput, Math.sqrt(valInput));
    }
}
<target name="test">
    <junit fork="on">
        <formatter type="xml" />
        <classpath refid="...">
            <batchtest>...
            </batchtest>
        </classpath>
    </junit>
</target>

<target name="report">
    <mkdir dir="reports"/>
    <junitreport todir="./reports">
        <fileset dir="."/>
        <include name="TEST-*.xml"/>
    </fileset>
    <report format="frames" todir="./reports/html"/>
</junitreport>
</target>
Task 3

• Use some parameterized tests

• Try reports in HTML (with Ant)
Advanced features

- Matchers
  - `assertThat`
- Assumptions
- Rules
  - `TemporaryFolder`
  - `ErrorCollector`
- Categories

Further information
- [https://github.com/junit-team/junit/wiki](https://github.com/junit-team/junit/wiki)
Testing methods

- **Black-box testing**
  - Zero knowledge about the implementation (no access)
  - Tests based only on specification and interfaces (API)
  - Checking outputs against expectations for input values

- **White-box testing**
  - Full knowledge of the implementation (access to code)
  - Tester can modify the system a little bit for easy testing

- **Grey-box testing**
  - Tester knows the system (code), but cannot modify it
Dependencies among objects

- Units typically have dependencies
  - Very hard to test such units in full isolation
  - Approach: complex fixtures and test cases
  - Example
    ```java
    @Before
    public void setUp() {
        java.sql.Connection db = ... // complex init
        PersistenceMngr pm = new MyPersistenceMngr(db);
    }
    ```

- Possible solutions
  - dummy objects, fake, stubs, mock objects
Dependencies among objects

- Dummy objects
  - Passed around but never used (e.g., parameter list)
- Fake
  - Working simpler implementation (e.g., in-memory DB)
- Stub
  - “empty” implementation with predefined responses to method calls
- Mock object
  - Stub that also checks whether it is used correctly by the object under test ➔ “behavior verification”
  - Frameworks: jMock, Mockito
Unit testing for Windows/.NET

- MSTest (Visual Studio)
  - Annotations: `[TestClass], [TestMethod]
  - Basic assertion statements
    - `Assert.AreEqual(Object, Object, String)
    - `IsTrue, `IsNotNull, `IsInstanceOfType, `Fail, ...
  - More advanced: `StringAssert, `CollectionAssert

- Other frameworks
  - `xUnit.net: [http://xunit.github.io/](http://xunit.github.io/)
• More general information about testing
  ▪ NTIN070: Testování software (ZS)

• But you can do better than simple unit testing ...
  ▪ NSWI126: Pokročilé nástroje pro vývoj a monitorování software (LS)

• ... and you can even model, analyze, and verify program behavior
  ▪ NSWI101: Modely a verifikace chování systémů (ZS)
  ▪ NSWI132: Analýza programů a verifikace kódu (LS)
Links

- JUnit
  - http://junit.org
  - https://github.com/junit-team/junit/wiki

- NUnit
  - http://www.nunit.org

- CPPUnit
  - http://sourceforge.net/projects/cppunit/
Homework

- Assignment
  - [http://d3s.mff.cuni.cz/~parizek/teaching/sdt/](http://d3s.mff.cuni.cz/~parizek/teaching/sdt/)

- Deadline

- Homework targets Java and JUnit
  - Alternative 1: C# and suitable framework
    - They use same concepts but little bit different syntax
    - Write similar test cases for the corresponding classes from the .NET base class library (e.g., SortedDictionary)
  - Alternative 2: In fact, any other language with support for unit testing can be used
    - For example: C++, Python, Scala