

Functional Testing

(Testování funkčnosti)

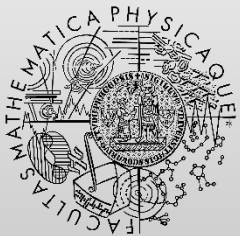
<http://d3s.mff.cuni.cz>

Department of
Distributed and
Dependable
Systems



Pavel Parízek

parizek@d3s.mff.cuni.cz



FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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
- Test oracle
 - Determines whether the program behaves correctly
 - “Oracle problem”: complex apps, user interface, automation

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 (end-to-end)
 - 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**, TestNG, PyUnit, CPPUnit, Google Test, MSTest, NUnit, xUnit, and many others

- Unit testing framework for Java
 - <https://github.com/junit-team/junit/wiki>
 - <http://junit.org/junit5/>
- 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/5.x: annotations**

Simple test case

```
import java.util.*;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class TestArrayList {
    @Test
    public void add() {
        List al = new ArrayList();
        int origSz = al.size();
        al.add("abc");
        int newSz = al.size();
        assertEquals(origSz+1, newSz, "new != orig+1");
        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)`
- `fail([String message])`

Running tests

- Many options
 - Command line (`java -cp ... <test runner>`)
 - Build tools (Ant, Maven, Gradle, ...)
 - Popular IDEs (Eclipse, NetBeans, IntelliJ IDEA)
 - Warning: make sure you have a proper setup (various caches, libraries, classpath, ...)
- Information
 - <https://junit.org/junit5/docs/current/user-guide/#running-tests>

What you should test

- Method contracts (API)
- All branches in the code
- All control-flow paths
- Special (corner) cases
 - “off by one”, bad inputs
- Positive testing
- Negative testing
- Regressions
 - Inputs triggering previously discovered bugs

Fixture

- Goal: prepare objects in a known state
 - Set up a fixed environment for each test cases
- Reset before each test case ➔ isolated tests
- Initialization
 - @BeforeEach
 - @BeforeAll
- Clean-up
 - @AfterEach
 - @AfterAll

Test case with a simple fixture

```
import org.junit.jupiter.api.*;

public class TestArrayList {
    private List al;

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

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

    @Test
    public void add() { ... }
}
```

Expected exceptions

@Test

```
public void testSomething() {  
    assertThrows(MyEx.class, () ->  
        doSomeUnsafeOperation());  
}
```

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
- Drawback: you need to write/produce lot more code

Parameterized tests

```
public class TestSquareRoot {

    public static Stream<Arguments> testData() {
        return Stream.of(
            arguments(1,1),
            arguments(4,2)
        );
    }

    @ParameterizedTest
    @MethodSource("testData")
    public void test(int expOutput, int valInput) {
        assertEquals(expOutput, Math.sqrt(valInput));
    }
}
```


Advanced features of JUnit (4)

- Matchers
 - `assertThat`
- Assumptions
- Rules
 - `TemporaryFolder`
 - `ErrorCollector`
- Categories
- Further information
 - <https://github.com/junit-team/junit/wiki>

JUnit 5 – new features

- Framework decomposed into several modules
- Distributed through Maven central repository
- User guide
 - <https://junit.org/junit5/docs/current/user-guide/>
- New syntax of annotations
 - @BeforeEach vs @Before, @AfterEach vs @After
 - @BeforeAll vs @BeforeClass, @AfterAll
- New modern API
 - Classes and interfaces => different imports
 - Named assertions, grouping via assertAll
 - Syntax for parameterized tests (data source)

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

```
@BeforeEach
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

Testing with mock objects

- Mock object
 - Stub that also checks whether it is used correctly by the object under test → “behavior verification”
- Frameworks
 - EasyMock (<https://easymock.org/>)
 - Mockito (<https://site.mockito.org/>)
 - Moq (<https://github.com/devlooped/moq>)
 - Rhino Mocks (<https://hibernatingrhinos.com/oss/rhino-mocks>)
 - Microsoft Fakes in Visual Studio
 - Only stubs, not full mocks
 - <https://learn.microsoft.com/en-us/visualstudio/test/isolating-code-under-test-with-microsoft-fakes?view=vs-2022>

Concurrency

- Testing does not work for concurrency
 - Programs with multiple threads
- Huge number of thread schedules
- Non-deterministic behavior
- Errors are hard to reproduce

Other artifacts and processes

- Configuration
- User interfaces
- Complete user scenarios (end-to-end)

Unit testing for C#/.NET/Windows

- MSTest (Visual Studio)
 - Annotations: [TestClass], [TestMethod]
 - Fixture: [TestInitialize], [TestCleanup]
 - Basic assertion statements
 - Assert.AreEqual(Object, Object, String)
 - IsTrue, IsNotNull, IsInstanceOfType, Fail, ...
 - More advanced: StringAssert, CollectionAssert
 - Parameterized tests: [DataRow]
- Other frameworks
 - NUnit: <http://nunit.org/>, <https://github.com/nunit>
 - xUnit.net: <https://xunit.net/>

Automation

- Generating tests with dynamic symbolic analysis
 - Manual writing of tests is very tedious
 - KLEE: <http://klee.github.io/>
 - IntelliTest: <https://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual/?view=vs-2022>
- Fuzzing techniques and tools

Fuzzing

- Search for inputs that may trigger some errors
 - Generating inputs [semi-] randomly (with constraints)
 - Visible failures: program crash, wrong output
- Useful for security bugs (critically important, hard-to-find)
- Interesting tools
 - SAGE & DART
 - Information and links: <https://patricegodefroid.github.io/>
 - AFL++ (Americal Fuzzy Loop): <https://aflplus.plus/>
 - JDart: <https://github.com/psycopaths/jdart>
 - OSS-Fuzz: <https://github.com/google/oss-fuzz>
- Literature
 - The Fuzzing Book (<https://www.fuzzingbook.org/>)
 - Fuzzing: Hack, Art, and Science. Communications of the ACM, Feb 2020
 - <https://cacm.acm.org/research/fuzzing/>

Related courses

- 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
 - <https://github.com/junit-team/junit/wiki>
 - <http://junit.org/junit5/>
- TestNG
 - <https://testng.org/doc/>
- MSTest
 - <https://learn.microsoft.com/en-us/visualstudio/test/unit-test-your-code?view=vs-2022>
- NUnit
 - <http://www.nunit.org>
 - <https://github.com/nunit/docs/wiki/NUnit-Documentation>
- CPPUnit
 - <http://sourceforge.net/projects/cppunit>
- Catch2
 - <https://github.com/catchorg/Catch2>
- Google Test
 - <https://github.com/google/googletest>