An Introduction into JUnit

Vorlesung Programmieren Wintersemester 2016/17
Erik Burger | 8 February 2017
Outline – Testing

1 Classification of Tests

2 Test-driven development
   1 Definition
   2 Refactoring
Outline – JUnit

3 Overview

4 Assertions

5 Fixtures
   1 Definition
   2 Example
   3 Parameterised Tests
   4 Test Suites

6 Eclipse Integration
   1 Test Runners
   2 Screenshot
Foreword

Program testing can be used to show the presence of bugs, but never to show their absence!

Dijkstra 1972
Classification of Tests

**Functional Tests**
- Correctness according to specification
- Concurrency/Thread safeness

**Non-Functional**
- Performance
- Security
- Usability
- Interoperability
- Reliability
Classification of Tests

Knowledge

- black-box tests
- white-box tests

Structure

- Unit
- Integration
- System
Test-Driven Development

“Test-infected development” (Erich Gamma)

- As a developer, you can’t stop coding until everything works
- Extreme Testing
- Testing is more important than coding
- Testing First
Test-Driven Development

1. add tests
2. fulfil tests
3. simplify/refactor code

JUnit: Failure

JUnit: OK
Object oriented classes often have dependencies on other classes
A lot of classes cannot be tested independently
→ micro integration tests
Starting from a certain degree of dependencies, test effort rises disproportionately high
Isolated Testing

- Test represent typical usage scenarios
- Less dependencies $\rightarrow$ easier to use
- High degree of dependencies
  - Lack of modularisation?
  - Bad design?
  - Bad code dependency management
- $\rightarrow$ Refactoring
Refactoring

Keep your code simple – always

**Refactor code**
- Before testing – fulfil test with simple code
- After testing – make code more simple

**Without changing semantics you can**
- Divide or merge classes
- Move methods and attribute to other classes
- Rename classes, methods, parameters, and variables
- Extract or integrate methods
- Replace inheritance by delegation and vice versa
- Introduce interfaces
Challenges for Testing Large-Scale Systems

Building test environments is time-consuming
Class dependencies require instantiation of multiple objects that have to be configured

Tests slow down if system borders are crossed
Interaction between database and middleware creates delays

Testing exceptional circumstances is non-trivial
Difficult to reproduce system faults

→ Keep tests simple, fast and coherent
Bad smells

See [Fowler 2004]

Duplicated logic
- Copies are difficult to handle
- Usually each copy start its own life
- Leads to spread information
- Hard to manage changes

Big classes and long methods
- Hinder mobility, localisation, usage, and documentation
- Medium for duplicated code
Bad smells

Bad naming and non-fitting code-structures
Increase effort for learning how the program works

Speculative generalisation
Elements of design formerly intended to make code flexible – but never used

Comments
- Often used as deodorant for bad smells
- Needless after refactoring
- Do not mix up with comments used for
  - Design decisions
  - Documentation of public interfaces (Javadoc)
  - References
JUnit

3. Overview

4. Assertions

5. Fixtures

6. Eclipse Integration
JUnit is a framework for writing tests

- JUnit uses Java’s reflection capabilities (Java programs can examine their own code)
- JUnit helps the programmer:
  - define and execute tests and test suites
  - formalize requirements and clarify architecture
  - write and debug code
  - integrate code and always be ready to release a working version
- JUnit is not included in Sun’s SDK, but almost all IDEs include it (e.g., Eclipse)
JUnit was written by Erich Gamma (of Design Patterns) and Kent Beck (creator of XP methodology). JUnit inspired various other unit testing frameworks for other programming languages, like NUnit (.NET), CppUnit (C++). JUnit is the de facto standard for test driven Java development.

[Gamma et al. 1994]
JUnit 3 vs JUnit 4

JUnit4

- JUnit4 was a complete redevelopment
- includes ideas from other frameworks and uses features of Java 1.5
- uses Java annotations (such as `@Test`)
- This lecture is based on JUnit 4
- JUnit 5 is under development for Java 8

Be careful

- Many (web) tutorials are still based on JUnit 3
- JUnit 4 is backwards compatible to version 3
- JUnit 4 is much cleaner
A test fixture sets up the data (both objects and primitives) that are needed to run tests

Example: If you are testing code that updates an employee record, you need an employee record to test it on

A unit test is a test of a single class

A test case tests the response of a single method to a particular set of inputs

A test suite is a collection of test cases

A test runner is software that runs tests and reports results

An integration test is a test of how well classes work together

JUnit provides some limited support for integration tests
A unit test tests the methods in a single class

A test case tests (insofar as possible) a single method

You can have multiple test cases for a single method

A test suite combines unit tests

The test fixture provides software support for all this

The test runner runs unit tests or an entire test suite

Integration testing is not well supported by JUnit
Test Case Verdicts

- **A verdict** is the declared result of executing a single test.
- **Pass**: the test case achieved its intended purpose, and the software under test performed as expected.
- **Fail**: the test case achieved its intended purpose, but the software under test did not perform as expected.
- **Error**: the test case did not achieve its intended purpose. Potential reasons:
  - An unexpected event occurred during the test case.
  - The test case could not be set up properly
What is a JUnit Test?

A test “script” is just a collection of Java methods.

General idea is to create a few Java objects, do something interesting with them, and then determine if the objects have the correct properties.

What is added? Assertions.

- A package of methods that checks for various properties:
  - “equality” of objects
  - identical object references
  - null / non-null object references

- The assertions are used to determine the test case verdict.
Organisation of JUnit Tests

- Each method represents a single test case that can independently have a verdict (Pass, Fail, Error).
- Normally, all the tests for one Java class are grouped together into a separate class.
- Naming convention:
  - Class to be tested: Value
  - Class containing tests: ValueTest
Writing a JUnit test class

Start by importing these JUnit 4 classes

```java
import org.junit.*
import static org.junit.Assert.*; // note static import
```

Declare your test class in the usual way

```java
public class MyProgramTest {
}
```

Declare an instance of the class being tested

```java
public class MyProgramTest {
    MyProgram program;
    int someVariable;
}
```
import org.junit.*;
import static org.junit.Assert.*;
public class ArithmeticTest {
    @Test
    public void testMultiply() {
        assertEquals(4, Arithmetic.multiply(2, 2));
        assertEquals(-15, Arithmetic.multiply(3, -5));
    }

    @Test
    public void testIsPositive() {
        assertTrue(Arithmetic.isPositive(5));
        assertFalse(Arithmetic.isPositive(-5));
        assertFalse(Arithmetic.isPositive(0));
    }
}
Assertions are defined in the JUnit class `Assert`.

- If an assertion is true, the method continues executing.
- If any assertion is false, the method stops executing at that point, and the result for the test case will be `fail`.
- If any other exception is thrown during the method, the result for the test case will be `error`.
- If no assertions were violated for the entire method, the test case will pass.

All assertion methods are `static` methods.
Assertion Methods

Boolean conditions are true or false

```java
assertTrue(condition)
assertFalse(condition)
```

Objects are null or non-null

```java
assertNull(object)
assertNotNull(object)
```

Objects are identical or not identical

i.e. two references to the same object `assertSame(expected, actual)`

```java
assertNotSame(expected, actual)
```
Assertion Methods

“Equality” of objects

$\text{assertEquals}(\text{expected, actual})$

valid if: $\text{expected.equals(actual)}$

“Equality” of arrays

$\text{assertArrayEquals}(\text{expected, actual})$

- arrays must have same length
- for each valid value for $i$, check as appropriate:

$\text{assertEquals}(\text{expected}[i], \text{actual}[i])$

$\text{assertArrayEquals}(\text{expected}[i], \text{actual}[i])$

There is also an unconditional failure assertion $\text{fail}()$ that always results in a fail verdict.
A test fixture is the context in which a test case runs.

Typically, test fixtures include:
- Objects or resources that are available for use by any test case.
- Activities required to make these objects available and/or resource allocation and de-allocation: “setup” and “teardown”.

- Allows multiple tests of the same or similar objects
- Share fixture code for multiple tests
Before/After

- **@Before**: Methods annotated with `@Before` are executed before every test.
- **@After**: Methods annotated with `@After` are executed after every test.
- If there are e.g. 10 tests, every `@Before` method is executed 10 times.
- More than one `@Before` or `@After` is allowed.
- Names of these methods are irrelevant, but must be `public void`
public class MoneyTest {
    private Money f12CHF;
    private Money f14CHF;
    private Money f28USD;

    @Before
    public void setUp() {
        f12CHF = new Money(12, "CHF");
        f14CHF = new Money(14, "CHF");
        f28USD = new Money(28, "USD");
    }
}
Setup and Teardown

**Setup**
Use the `@Before` annotation on a method containing code to run before each test case.

**Teardown (regardless of the verdict)**
Use the `@After` annotation on a method containing code to run after each test case. These methods will run even if exceptions are thrown in the test case or an assertion fails.

**It is allowed to have any number of these annotations**
All methods annotated with `@Before` will be run before each test case, but they may be run in any order.
@BeforeClass: executed once before a test suite
@AfterClass: executed once after a test suite
Only one @BeforeClass and @AfterClass allowed
Methods must be static
public class MoneyTest {
    private static string currency;

    @BeforeClass
    public static void setGlobalCurrency() {
        this.currency = "CHF";
    }

    @Before
    public void setUp() {
        m12 = new Money(12, this.currency);
        m14 = new Money(14, this.currency);
    }
}

- Exceptions that are expected on test executing
- Annotation using `@Test`
- `@Test(expected=MyException.class)`
- If no exception is thrown, or an unexpected exception occurs, the test will fail.
- That is, reaching the end of the method with no exception will cause a test case failure.
Expected Exceptions – example

```java
new ArrayList<Object>().get(0);
```

- Should throw an `IndexOutOfBoundsException`

```java
@Test(expected = IndexOutOfBoundsException.class)
public void empty() {
    new ArrayList<Object>().get(0);
}
```
Ignore/Timeout

### Ignore
- Tests annotated using `@Ignore` are not executed
- Test runner *reports* that test was not run
- `@Ignore("Reason")` allows to specify a reason why a test was not run

### Timeout
- Test allows to specify a timeout parameter
- `@Test(timeout=10)` fails if the test takes more than 10 milliseconds
Parameterised Tests

Motivation

If you want a test to run with several parameter values, you’d have to

- loop over a collection of values
- which means if there was a failure, the loop wouldn’t terminate
- write unique test cases for each test data combination
- which could prove to be a lot of coding

Support in JUnit

With JUnit, you can create highly flexible testing scenarios easily
Creating a parameterised test

1. Create a generic test and decorate it with the `@Test` annotation.
2. Create a static feeder method that returns a Collection type and decorate it with the `@Parameters` annotation.
3. Create class members for the parameter types required in the generic method defined in Step 1.
4. Create a constructor that takes these parameter types and correspondingly links them to the class members defined in Step 3.
5. Specify the test case be run with the Parameterized class via the `@RunWith` annotation.
Parameterised Test – Example

```java
@RunWith(Parameterized.class)
public class ParameterizedTest {
    private int numberToTest;
    private int rest;
    public ParameterizedTest(Integer pValue, Integer rValue) {
        numberToTest = pValue.intValue();
        rest = rValue.intValue();
    }
    @Parameters
    public static List<Integer[]> testValues() {
        return Arrays.asList(new Integer[][] {
            {1,1}, {3,1}, {6,0}, {7,1}, {9,1}
        });
    }
    @Test
    public void isOdd() {
        assertTrue(numberToTest % 2 == rest);
    }
}
```
Test Suites

Creating a test suite

- Tests can be combined to test suites
- suites can contain other suites
- useful for partitioning your test scenarios
- well supported by Test Runners (see example)
Test Suite – Example

```java
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    MyTest1.class,
    MyTest2.class,
    MyTest3.class
})

public class AllTests {
}
```
Running JUnit Tests

- The JUnit framework does not provide a graphical test runner. Instead, it provides an API that can be used by IDEs to run test cases and a textual runner that can be used from a command line.
- Eclipse and Netbeans each provide a graphical test runner that is integrated into their respective environments.
Test Runners

With the runner provided by JUnit:

- When a class is selected for execution, all the test case methods in the class will be run.
- The order in which the methods in the class are called (i.e. the order of test case execution) is not predictable.

Other Runners

- Test runners provided by IDEs may allow the user to select particular methods, or to set the order of execution.
- It is good practice to write tests with are independent of execution order, and that are without dependencies on the state any previous test(s).
Eclipse TestRunner

- JUnit included
- GUI provided
- easy re-running of tests
- test history

