Test-Driven Development at the Unit Testing Level

Dr. Christoph Steindl

Motivation

- If you intend to test after you've developed the system, you won't have the time for testing.
  ➔ Write the tests before the code!
- If things get complicated, you might fear that „the system" doesn't work.
  ➔ Execute the tests and get positive feedback (everything still works) or get pointed to the bit that does not / no longer work.
- If you're overwhelmed by the complexity, you get frustrated.
  ➔ Start with the simplest thing and proceed in tiny steps!
- If you don't have tests for the code, you shouldn't use it / ship it.
  ➔ This can't happen if you write the test first (so you reach better test coverage than with functional tests).
- If performance is only considered late, you won't be able to just „add a little more performance" to the system.
  ➔ Re-use unit tests for performance tests even during development and don't start with performance tests late in the project!

Red – Green – Refactor

- **Red**
  - Write a little test that doesn‘t work (and perhaps doesn‘t even compile at first).
- **Green**
  - Make the test work quickly (committing whatever sins necessary)
- **Refactor**
  - Eliminate all of the duplication created in merely getting the test to work, improve the design.

Why?

- The test is the executable specification.
  - You start thinking about the goal first, then about the possible implementations.
  - You understand the program‘s behavior by looking at the tests. The tests tell you more than just an API description, they show the dynamics, how to use the API.
- You develop just enough.
  - You get to the goal as quick as possible.
  - You don‘t develop unnecessary code.
  - There is no code without a test.
  - There is no test without a user requirement.
- Once you get one test working, you know it is working now and forever.
  - You use the tests as regression tests.
- The tests give us the courage to refactor.
  - You can prove that everything still works after the refactoring by simply executing the tests.
- It‘s funnier that way, it reduces fear.

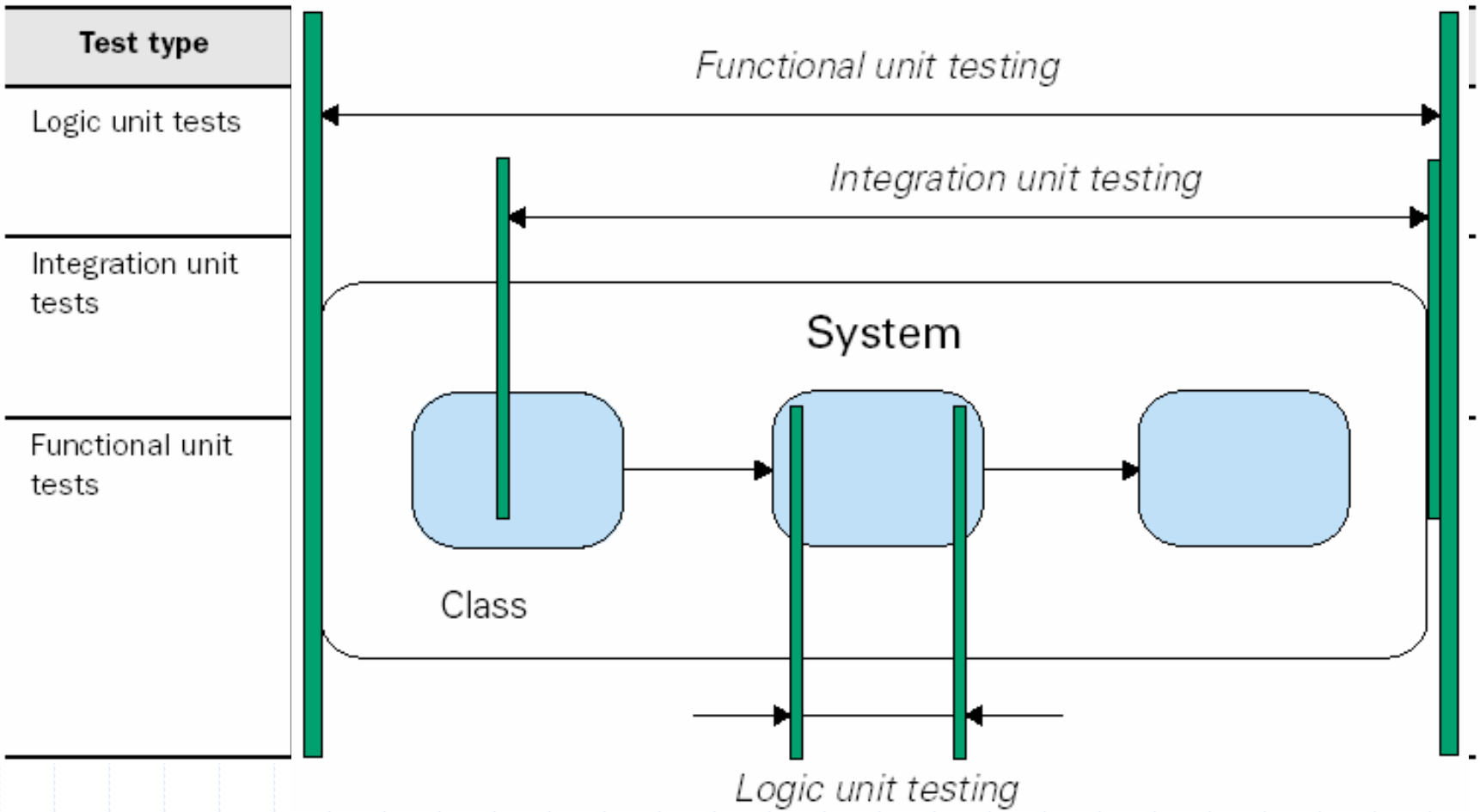
How?

- Don‘t start with objects (or design, or ...), start with a test.
  - Write new code only if an automated test has failed.
  - First think of the goal, the required functionality.
  - Then think of the perfect interface for that operation (from the outside, black-box view).
- Run the test often – very often.
  - To determine whether you‘ve reached the goal.
  - To catch any bugs that have crawled back in.
- Make little steps (during coding and refactoring)
  - So little that you feel comfortable with them.
  - Make them larger if you feel.
  - Make them smaller if you don‘t proceed with your expected velocity.

Use Tools

- Framework for automating the unit tests
  - E.g. Junit
- Integrated development environment
  - For writing tests, using auto-completion and generation of missing code.
  - For running the tests
  - For refactoring
  - E.g. Eclipse
- Build environment
  - For executing tests automatically and during the build process
  - For computing code coverage
  - For generating test reports
  - E.g. Maven

Flavors of Unit Tests



How to Use Junit?

- Install the framework (see homework).
- Write tests (one for each feature)
  - Write the expected goal first.
  - Assert that the actual result equals the expected result.
  - Work backwards to compute the expected result.
  - Finally write the setup code needed at the start of the test.
- Combine tests to test suites.
- Extract common setup and tear down code into test fixtures.
- Execute tests or test suites (see homework).
- Organize your test suites.

Assert

Method	Description
`assertTrue`	Asserts that a condition is true. If it isn't, the method throws an `AssertionFailedError` with the given message (if any).
`assertFalse`	Asserts that a condition is true. If it isn't, the method throws an `AssertionFailedError` with the given message (if any).
`assertEquals`	Asserts that two objects are equal. If they are not, the method throws an `AssertionFailedError` with the given message (if any).
`assertNotNull`	Asserts that an object isn't null. If it is, the method throws an `AssertionFailedError` with the message (if any).
`assertNull`	Asserts that an object is null. If it isn't, the method throws an `AssertionFailedError` with the given message (if any).
`assertSame`	Asserts that two objects refer to the same object. If they do not, the method throws an `AssertionFailedError` with the given message (if any).
`assertNotSame`	Asserts that two objects do not refer to the same object. If they do, the method throws an `AssertionFailedError` with the given message (if any).
`fail`	Fails a test with the given message.

TestCase

Method	Description
`countTestCases`	Counts the number of `TestCases` executed by `run(TestResult result)`. (Specified by the `Test` interface.)
`createResult`	Creates a default `TestResult` object.
`getName`	Gets the name of a `TestCase`.
`run`	Runs the `TestCase` and collects the results in `TestResult`. (Specified by the `Test` interface.)
`runBare`	Runs the test sequence without any special features, like automatic discovery of test methods.
`runTest`	Override to run the test and assert its state.
`setName`	Sets the name of a `TestCase`.
`setUp`	Initializes the fixture, for example, to open a network connection. This method is called before a test is executed. (Specified by the `Test` interface.)
`tearDown`	De-initializes the fixture, for example, to close a network connection. This method is called after a test is executed. (Specified by the `Test` interface.)
`toString`	Returns a string representation of the `TestCase`.

First Example

Calculator.java

```
public class Calculator
{
  public double add(double number1, double number2)
  {
    return number1 + number2;
  }
}
```

TestCalculator.java

```
import junit.framework.TestCase;

public class TestCalculator3 extends TestCase {
  public void testAdd() {
    Calculator calculator = new Calculator();
    double result = calculator.add(10, 50);
    assertEquals(60, result, 0);
  }
}
```

JUnit best practices: let the test improve the code

An easy way to identify exceptional paths is to examine the different branches in the code you're testing. By *branches*, we mean the outcome of `if` clauses, `switch` statements, and `try/catch` blocks. When you start following these branches, sometimes you may find that testing each alternative is painful. If code is difficult to test, it is usually just as difficult to use. When testing indicates a poor design (called a *code smell*, http://c2.com/cgi/wiki?CodeSmell), you should stop and refactor the domain code.

In the case of too many branches, the solution is usually to split a larger method into several smaller methods.[5] Or, you may need to modify the class hierarchy to better represent the problem domain.[6] Other situations would call for different refactorings.

A test is your code's first "customer," and, as the maxim goes, "the customer is always right."

How to Use Eclipse?

- Install it (see homework).
- Develop the program
  - Write tests
  - Use auto-completion to generate the missing code
  - Execute the tests
  - Refactor the code

How to Use Ant?

- Install it (see homework).
- Write the „build file“ (build.xml).
  - Name the project
  - Define the „default“ target
  - Define properties (global variables)
- Write the „targets“.
  - For compiling
  - For executing the tests
  - For creating the documentation (Javadoc, test results,...)
  - For bundling the deliverables
- Execute the targets (see homework).
- Use Grand (http://www.ggtools.net/grand/) to visualize a build file.

Build.xml for Ant (1/5)

```
<project name="sampling" default="test">

  <property file="build.properties"/>

  <property name="src.dir" location="src"/>
  <property name="src.java.dir" location="${src.dir}/java"/>
  <property name="src.test.dir" location="${src.dir}/test"/>

  <property name="target.dir" location="target"/>
  <property name="target.classes.java.dir"
     location="${target.dir}/classes/java"/>
  <property name="target.classes.test.dir"
     location="${target.dir}/classes/test"/>

  <property name="target.report.dir"
     location="${target.dir}/report"/>

</project>
```

- Name
- Default target
- Properties from a file
- Additional properties

Build.xml for Ant (2/5)

```
<target name="compile.java">
   <mkdir dir="${target.classes.java.dir}"/>
   <javac destdir="${target.classes.java.dir}">
     <src path="${src.java.dir}"/>
   </javac>
 </target>

 <target name="compile.test" depends="compile.java">
   <mkdir dir="${target.classes.test.dir}"/>
   <javac destdir="${target.classes.test.dir}">
     <src path="${src.test.dir}"/>
     <classpath>
       <pathelement location="${target.classes.java.dir}"/>
     </classpath>
   </javac>
 </target>

<target name="compile" depends="compile.java,compile.test"/>
```

Compile targets

Build.xml for Ant (3/5)

```
<target name="test" depends="compile">
   <mkdir dir="${target.report.dir}"/>
   <junit printsummary="yes" haltonerror="yes" haltonfailure="yes"
      fork="yes">
     <formatter type="plain" usefile="false"/>
     <formatter type="xml"/>
     <test name="junitbook.sampling.TestDefaultController"
          todir="${target.report.dir}"/>
     <classpath>
       <pathelement location="${target.classes.java.dir}"/>
       <pathelement location="${target.classes.test.dir}"/>
     </classpath>
   </junit>
 </target>
```

Test target

Build.xml for Ant (4/5)

Batch test target

```
<target name="test" depends="compile">
  <mkdir dir="${target.report.dir}"/>
  <property name="tests" value="Test*"/>
  <junit printsummary="yes" haltonerror="yes" haltonfailure="yes">
   <formatter type="plain" usefile="false"/>
   <formatter type="xml"/>
   <batchtest todir="${target.report.dir}">
    <fileset dir="${src.test.dir}">
     <include name="**/${tests}.java"/>
     <exclude name="**/AllTests.java"/>
    </fileset>
   </batchtest>
   <classpath>
    <pathelement location="${target.classes.java.dir}"/>
    <pathelement location="${target.classes.test.dir}"/>
   </classpath>
  </junit>
 </target>
```

Build.xml for Ant (5/5)

```
<target name="report" depends="test">
   <mkdir dir="${target.report.dir}/html"/>
   <junitreport todir="${target.report.dir}">
    <fileset dir="${target.report.dir}">
      <include name="TEST-*.xml"/>
    </fileset>
    <report todir="${target.report.dir}/html"/>
   </junitreport>
 </target>
```

Report target

```
<target name="clean">
   <delete dir="${target.dir}"/>
 </target>
```

Clean target

How to Use Maven?

- Install it (see homework).
- Generate the project template and directory structure.
- Adapt the project template.
- Generate Eclipse files by: „maven eclipse“
- Compile the sources by: „maven java:compile“
- Execute the tests by: „maven test“
- Generate Ant „build.xml“ by: „maven ant“
- Generate CruiseControl file by: „maven cruisecontrol“
- Run CruiseControl by: „maven cruisecontrol:run“
- Generate Reports by: „maven site“

Project.xml for Maven (1/4)

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<project>
  <pomVersion>3</pomVersion>
  <id>junitbook-sampling</id>
  <name>JUnit in Action - Sampling JUnit</name>
  <currentVersion>1.0</currentVersion>
  <organization>
    <name>Manning Publications Co.</name>
    <url>http://www.manning.com/</url>
    <logo>http://www.manning.com/front/dance.gif</logo>
  </organization>
  <inceptionYear>2002</inceptionYear>
  <package>junitbook.sampling</package>
  <logo>/images/jia.jpg</logo>
</project>
```

- Name,...

Project.xml for Maven (2/4)

```
<description>
  Chapter 3 presents a sophisticated test case to show how JUnit
  works with larger components. The subject of our case study is
  a component found in many applications: a controller. We
  introduce the case-study code, identify what code to test, and
  then show how to test it. Once we know that the code works as
  expected, we create tests for exceptional conditions, to be
  sure our code behaves well even when things go wrong.
</description>
<shortDescription>
  Chapter 3 of JUnit in Action: Sampling JUnit
</shortDescription>

<url>http://sourceforge.net/projects/junitbook/</url>
```

Description

Project.xml for Maven (3/4)

```
<developers>
  <developer>
    <name>Vincent Massol</name>
    <id>vmassol</id>
    <email>vmassol@users.sourceforge.net</email>
    <organization>Pivolis</organization>
    <roles>
      <role>Java Developer</role>
    </roles>
  </developer>
</developers>
```

Developers

Project.xml for Maven (4/4)

```
<build>
  <sourceDirectory>src/java</sourceDirectory>
  <unitTestSourceDirectory>src/test</unitTestSourceDirectory>
  <unitTest>
    <includes>
      <include>**/Test*.java</include>
    </includes>
    <excludes>
      <exclude>**/AllTests.java</exclude>
      <exclude>**/TestDefaultController?.java</exclude>
    </excludes>
  </unitTest>
 </build>
```

Directories

Benefits of Using Maven

- Standardizes the directory structure and targets.

Overview

Document	Description
Metrics	Report on source code metrics.
Checkstyle	Report on coding style conventions.
Change Log	Report on the source control changelog.
Developer Activity	Report on the amount of developer activity.
File Activity	Report on file activity.
Project License	Displays the primary license for the project.
JavaDocs	JavaDoc API documentation.
JavaDoc Report	Report on the generation of JavaDoc.
Source Xref	A set of browsable cross-referenced sources.
Test Xref	A set of browsable cross-referenced test sources.
Unit Tests	Report on the results of the unit tests.
Link Check Report	Report on the validity of all links in the documentation.
Task List	Report on tasks specified in the source code.

Stubs

- fake the behavior of real code that may exist or that may not have been written yet.
- allow you to test a portion of a system without the other part being available.
- do not change the code you're testing but instead adapt to provide seamless integration.
- usually hard to write, especially when the system to fake is complex.
- needs to implement the same logic as the code it is replacing, and that is difficult to get right for complex logic.
- can be difficult to maintain because they're complex.
- are better adapted for replacing coarse-grained portions of code.
- they're more a vestige of the past, when the general consensus was that tests should be a separate activity and should not modify the code.

DEFINITION *stub*—A stub is a portion of code that is inserted at runtime in place of the real code, in order to isolate calling code from the real implementation. The intent is to replace a complex behavior with a simpler one that allows independent testing of some portion of the real code.

Mock Objects

- allow testing in isolation
- allow you to unit-test at the finest possible level and develop method by method, while providing you with unit tests for each method.
- allows to test code that has not yet been written (as long as you at least have an interface to work with).
- allows to unit-test one part of the code without waiting for all the other parts.
- allows to write focused tests that test only a single method, without side effects resulting from other objects being called from the method under test.
- do not implement any logic: They are empty shells that provide methods to let the tests control the behavior of all the business methods of the faked class.

DEFINITION *mock object*—A *mock object* (or *mock* for short) is an object created to stand in for an object that your code will be collaborating with. Your code can call methods on the mock object, which will deliver results as set up by your tests.

Expectations

- Write your test case with a mock object.
- Tell the mock object what it will experience (= predict the behaviour of the code to be tested).
- Let your code (to be tested) use the mock object instead of the (to be) environment (= generate the actual behaviour).
- Tell the mock object to verify whether it has been exercised by the code in the predicted way (= compare the actual to the predicted behaviour).

DEFINITION *expectation*—When we're talking about mock objects, an *expectation* is a feature built into the mock that verifies whether the external class calling this mock has the correct behavior. For example, a database connection mock could verify that the `close` method on the connection is called exactly once during any test that involves code using this mock.

When to Use Mock Objects

- The real object has non-deterministic behavior.
- The real object is difficult to set up.
- The real object has behavior that is hard to cause (such as a network error).
- The real object is slow.
- The real object has (or is) a UI.
- The test needs to query the object, but the queries are not available in the real object (for example, “was this callback called?”)
- The real object does not yet exist.

In-Container Testing

- When components (to be tested) interact with their container, and the container services are available only when the container is running, test the component within the container.
- Mock Objects allow *outside-the-container testing,* where as Cactus allows *in-container testing* for J2EE components like servlets and JSPs.
- Although the concept is generic, the tools that implement in-container unit-testing are very specific to the underlying API being tested.
- For a test to run in the container, you need to start the container beforehand (start container, deploy code, start test outside the container, call code inside the container, verify the result outside the container).
- Configuration becomes more difficult, since you have to configure the container within the test (before you start the container).

DEFINITION *component/container*—A component is a piece of code that executes inside a container. A container is a receptacle that offers useful services for the components it is hosting, such as life cycle, security, transaction, distribution, and so forth.

When to Use Mock Objects vs. In-Container Testing

- The main difference is that In-Container Testing performs not only unit tests but also integration tests and, to some extent, functional tests. The added benefits come at the cost of added complexity.
- Mock-object tests are usually harder to write, because you need to define the behavior of all calls made to the mocks. For example, if your method under test makes 10 calls to mocks, then you need to define the behavior for these 10 calls as part of the test setup.
- In-container testing provides real objects for which you only need to set some initial conditions.
- If the application to unit-test is already written, it usually has to be refactored to support mock-object testing. Extra refactoring is generally not needed with in-container testing.
- A good strategy is to separate the business-logic code from the integration code (code that interacts with the container), and then:
  - Use mock objects to test the business logic.
  - Use in-container testing to test the integration code.

References

- Kent Beck: Test-Driven Development: By Example, Addison-Wesley, 2002.
- David Astels: Test-Driven Development: A Practical Guide, Prentice Hall, 2003.
- Frank Westphal: Testgetriebene Entwicklung, dpunkt.verlag, 2004
- Vincent Massol: Junit in Action, Manning Publications, 2003.
- J. B. Rainsberger: Junit Recipes, Manning Publications, 2004.
- Andrew Hunt, David Thomas: Pragmatic Unit Testing, Pragmatic Bookshelf, 2004.
- Johannes Link, Peter Fröhlich: Unit Tests mit Java, dpunkt.verlag, 2002.

Online References

- Unit Testing:
  - xUnit: http://www.xprogramming.com/software.htm
  - Junit: http://junit.org
  - NUnit: http://sourceforge.net/projects/nunit/
- Automated Build:
  - Ant: http://ant.apache.org/
  - Grand: http://www.ggtools.net/grand/
  - NAnt: http://nant.sourceforge.net/
- Integrated Development Environment:
  - Eclipse: http://eclipse.org
  - SharpDevelop: http://www.icsharpcode.net/OpenSource/SD/
- Build Environment:
  - Maven: http://maven.apache.org/

Online References

- On Ward Cunningham‘s original wiki:
  - http://c2.com/cgi/wiki?TestDrivenProgramming
  - http://c2.com/cgi/wiki?TestFirstDesign
- Various Resources
  - http://www.testdriven.com/
  - http://www.xprogramming.com/xpmag/testFirstGuidelines.htm
  - http://www.objectmentor.com/writeUps/TestDrivenDevelopment
  - „Aim, Fire“: http://www.computer.org/software/homepage/2001/05Design/index.htm