Test Driven Development (TDD)

Jari Aalto

Introduction (1/2)

- Test Driven Development (TDD) consists of short development iterations where the **test cases** covering a new functionality **are written first**.
- It is based on small increments of code.
- It's a process. Usually you cannot see in the end product whether TDD was applied or not.
- Automated test cases are an integral part of delivered software.

Introduction (2/2)

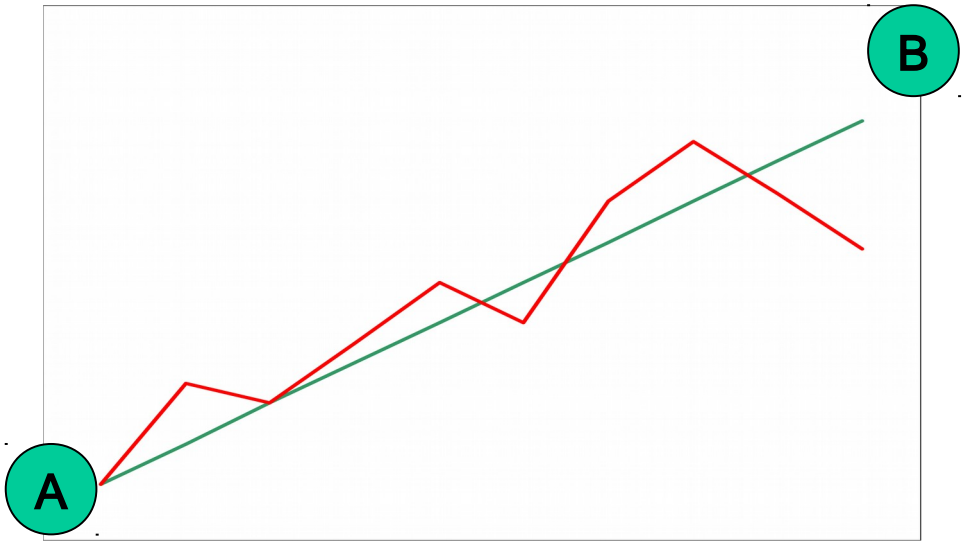
- Test cases are developed before product code is written.
- The development of the product is driven by the test case.
- Most of the documentation is incorporated in the source code.
- *TDD is the key factor in Agile Software Development methodology.*

TDD steps

- Create **unit test module**, to hold the automated test cases.
- Design and **implement test case**; document its motivation in the source code. This test case will fail, lacking an implementation.
- Implement functionality to **make the test case pass**.
- **Run tests**, fix defects found.
- **Refactore code**, run tests. Repeat until ready.

TDD and Keys to Success

- Write only **small fragments** of code at a time, and **run all test cases after each change**.
- If a defect is found while using the code: first, add a test case to detect that defect and only then repair the defect. Small steps to reach A-B:



TDD … Why?

- Software of some complexity is inevitable to have mistakes and defects**.**
- Defects reduces the product's quality.
- The longer a defect remains undetected, the **higher the cost of repairing it**.
- *Therefore, one should attempt to detect defects as soon as possible.*

Automated tests (1/3)

- **Manual testing** is time consuming, prone to error, and hard to reproduce
- **Automated test** cases are faster, more accurate, and reproducible.
- Automated test cases also serve as *documentation*: how to invoke and how to decide between passing and failing.
- Automated test cases *serve as a client*: the tests help to stabilize interfaces before releasing them.
- *Without test cases it is hard to localize and repair defects, leading to frustrating and unpredictable debugging work*

Automated tests (2/3)

- *Ad hoc debugging work is a waste of time*, because it does not help you to do better in the future.
- ...and systematic debugging boils down to writing automated test cases anyway. So, why not write them in the first place?
- Good automated test cases make debugging obsolete.

Automated tests (3/3)

- Automated test cases can be re-run anytime; in particular, they can be re-run after changing the software (*regression testing*). They serve as a safety net when refactoring the code.
- *Automation ensures that that production code always works and is release ready.*

Why tests first?

- Thinking *client first* makes sure you write code that meets the expectations (the interface, the contract).
- Think it as a *preventive measure*. If you write code first, it may be impossible to test afterwards (messy, too long code base, multiple tasks). Writing tests first forces to think keeping the code lean and simple.
- *Hand-in-hand: having automated test cases ready before implementing a module, allows you to focus on implementing the module and getting it to work. The is no time gaps between code/test (agile).*

Contract: an example in Java

- **The javadoc is the contract – the formalized specification.** Make sure it is as clear as possible. It's impossible to design and write good test cases without a good documentation.

```
/**
 * <Short description>
 *
 * <Long description, multiple lines, paragraphs>
 * <Detailed explanation and possibly call examples>
 *
 * @param one  <detailed information, range of values>
 * @param two  <detailed information, range of values>
 * @return     <detailed information, range of values>
 */
```

Do not use `@throws` tag: http://www.javapractices.com/topic/TopicAction.do?Id=171

TDD traffic lights (red, green)

Test

Add failing test
Red light

Code

Make test run
Green light

Refactor

Make code better
Keep light green

Critical view

- What about a system that goes through a ton of changes due to the organic nature fo the application?
  - *That effectively causes having to rewrite/change/delete most of the existing tests!*
- You can't really refactor tests if changes in code are massive in scale.
  - *It is best to delete the tests affected, and re-create them from scratch.*
- **Summary**: Double work. Hard to do and justify if it has a significant impact on project deadlines.

TDD and xUnit

- Various frameworks (Wikipedia) for programming languages have come to be known collectively as xUnit.
- Allow testing of different elements (units) of software, such as functions and classes.
- Concept of *fixtures*: pieces of code that set up the initial conditions, run tests, and clean up env.
- Effectively *Industry standard*: provides an automated solution with no need to write the tests.
- http://en.wikipedia.org/wiki/XUnit

TDD and JUnit (1/4)

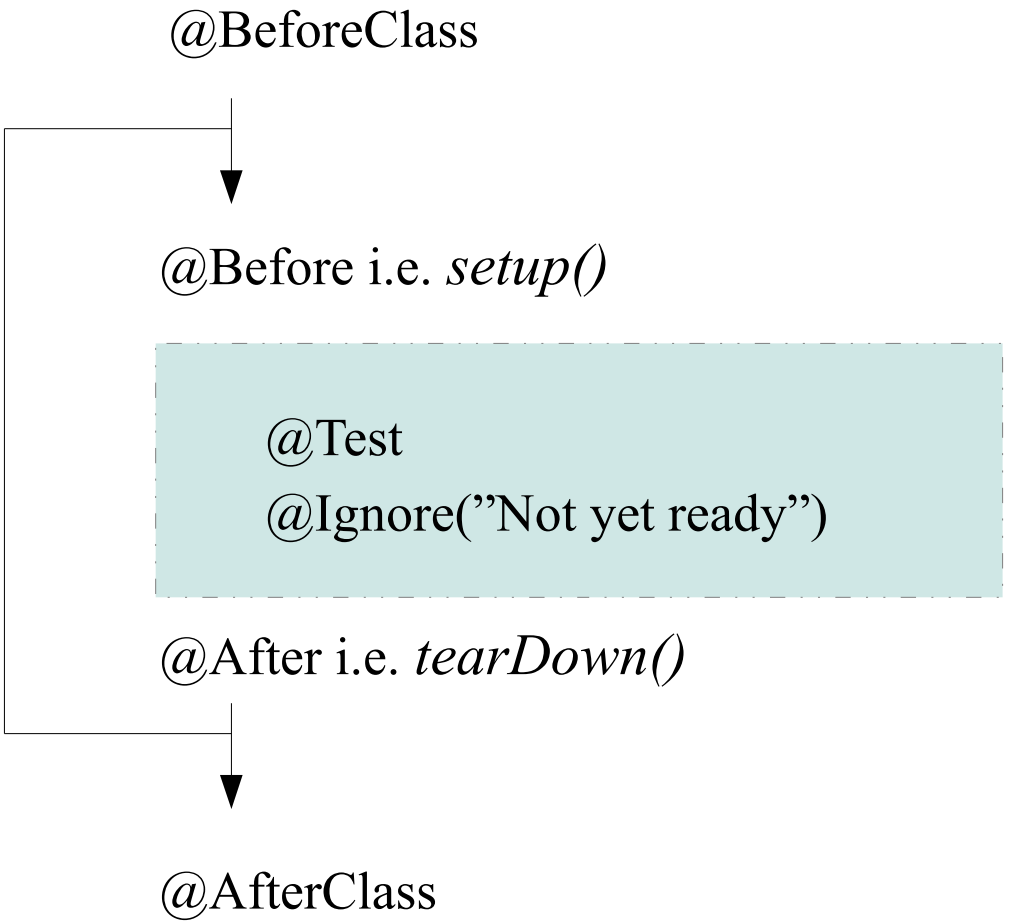
- Not yet part of the java standard base.
- Download JAR files from http://junit.org
  - *The JAR files provide the framework under package* ***org.junit***
  - ***Warning****: junit.framework.* was the old interface.*
- Fixtures in in Junit4: there is no `setup()` or `tearDown()` methods but `@Before` and `@After` annotations.
- http://junit.sourceforge.net/javadoc
- http://en.wikipedia.org/wiki/JUnit

TDD and JUnit (2/4)

- The annotations are form of syntactic metadata that can be added to Java source code. In Java this is implemented in form of @tags.
- In Java 1.5 the annotations become part of language.
- *Test suite* means bundling a few units together. In JUnit, both @RunWith and @Suite annotation are used to run the suite test (see an example).
- JUnit provides e.g. annotations: `@Before`, `@BeforeClass`, `@After`, `@AfterClass`, `@Test`, `@Ignore`, `@Test(timeout=<millisec>)`, `@Test(expected=<exception name>.class)`

TDD and JUnit (3/4)

The order of execution of the annotations in a fixture



JUnit @Ignore annotation

- Used to disable a test or a group of tests.
- Runner still reports that the test was not run.
- You can pass in a string to explain why the test was ignored. Use it instead of deleting a test.
  - `@Ignore("Test unfinished")`
- A class with @Ignore: none of the tests will be executed.
- Can mark tests which fail because of a known bug.
  - `@Ignore("Broken test. See bug#123456")`
- http://junit.sourceforge.net/javadoc/org/junit/Ignore.html

JUnit Notes

- You cannot print messages easily using `System.out`. See stackoverflow question ”JUnitCore.runClasses doesn't print anything”.

SUMMARY

- *TDD is the corner stone of being Agile.*
- If you're not writing automated unit tests, you're not developing agile software.
- The de facto is to use xUnit frameworks.
- In java, this means using JUnit. See IDE support in Eclipse, Netbeans etc.
- *When you do test-first, you write **simpler, more maintainable code** than you otherwise would.*

JUnit and command line

- Use classpath option (-cp) to inlcude JAR files. In Windows use path separator ”;” and in Linux use ”:”
- It is not possible to run a single test. See Stackoverflow thread.
- JAR files depend on http://junit.org release. Files hamcrest-core-3.x.jar junit-4.x.jar
- http://junit.sourceforge.net/javadoc/org/junit/runner/package-summary.html

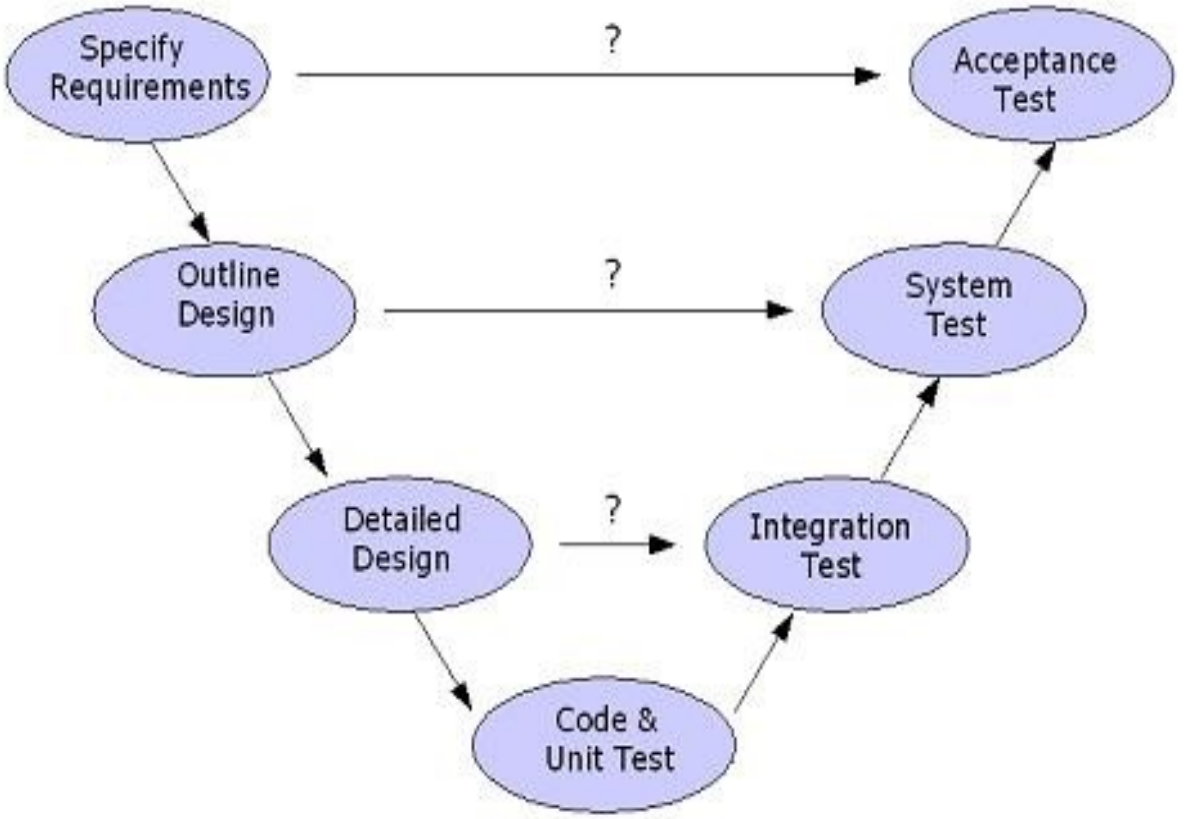
```
java -cp ".:JAR:JAR" \
     org.junit.runner.JUnitCore \
     <[package.]classname>
```

APPENDIX: JUnit 3.x changes

- Test classes do not have to extend from `junit.framework.TestCase`.
- Test methods do not have to be prefixed with `test`.
- @Test annotations: can take a parameter for *timeout* and type of *exception* thrown.
- JUnit4Adapter enables running the new JUnit4 tests using the old JUnit runners.
- Old JUnit tests can be run in the new JUnit4 runner.

APPENDIX: Testing and V-model

- The classic representation of testing phases:



APPENDIX: Areas of Testing

- Remember: **regression Testing**