HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

h_da

Agile Software Development

Part 7: Test-Driven Development

Prof. Dr. A. del Pino

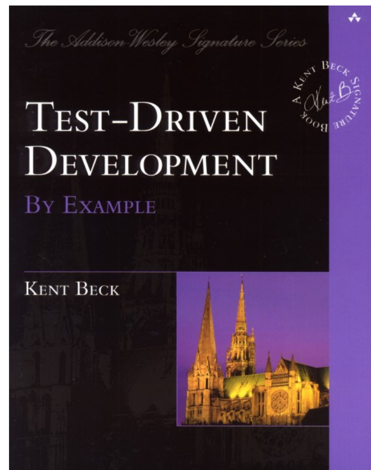
Test-Driven Development

Test-Driven Development (TDD)

The goal of TDD is best described by a phrase coined by Ron Jeffries:

> Clean code that works.

Literature:

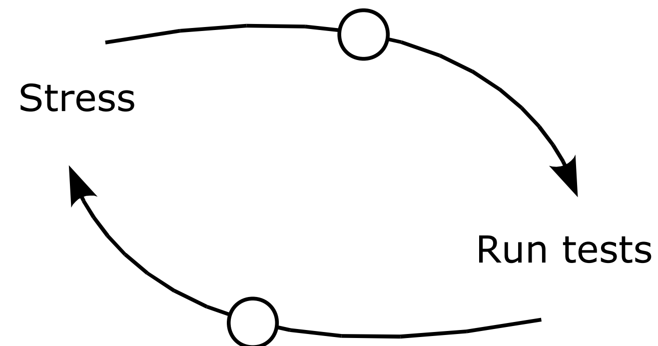


Kent Beck „Test-Driven Development",
Addison-Wesley, 2003

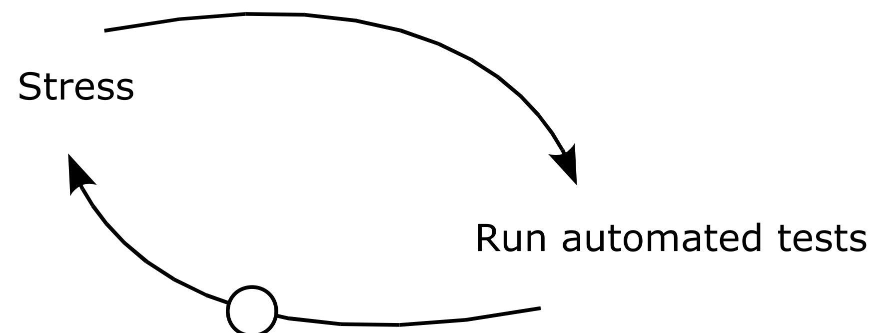
Test-Driven Development

When to test ?

Weinberg's influence diagram for „No time for testing“:



To get out of this death spiral, replace tests with automated tests:



Test-Driven Development

Rules

TDD is a software development style, where the following rules are applied:

> Write new code only if an automated test has failed.
>
> Eliminate duplication.
>
> Source: K. Beck „Test-Driven Development“, Addison-Wesley, 2003

Test-Driven Development
Implications

According to K. Beck this has the following implications:

> We must design organically, with running code providing feedback between decisions.
>
> We must write our own tests, because we can't wait 20 times per day for someone else to write a test.
>
> Our development environment must provide rapid response to small changes.
>
> Our designs must consist of many highly cohesive, loosely coupled components, just to make testing easy.
>
> Source: K. Beck „Test-Driven Development", Addison-Wesley, 2003

Test-Driven Development

The TDD Mantra

Trying to apply the rules leads to the following order of programming tasks:

> 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 in the process.
>
> Refactor – Eliminate all of the duplication created in merely getting the test to work.
>
> Source: K. Beck „Test-Driven Development“, Addison-Wesley, 2003

This sequence of programming tasks is what Kent Beck named the TDD mantra:

> Red/green/refactor – the TDD mantra.
>
> Source: K. Beck „Test-Driven Development“, Addison-Wesley, 2003

Test-Driven Development
Test construction

Develop small tests which are isolated from each other.

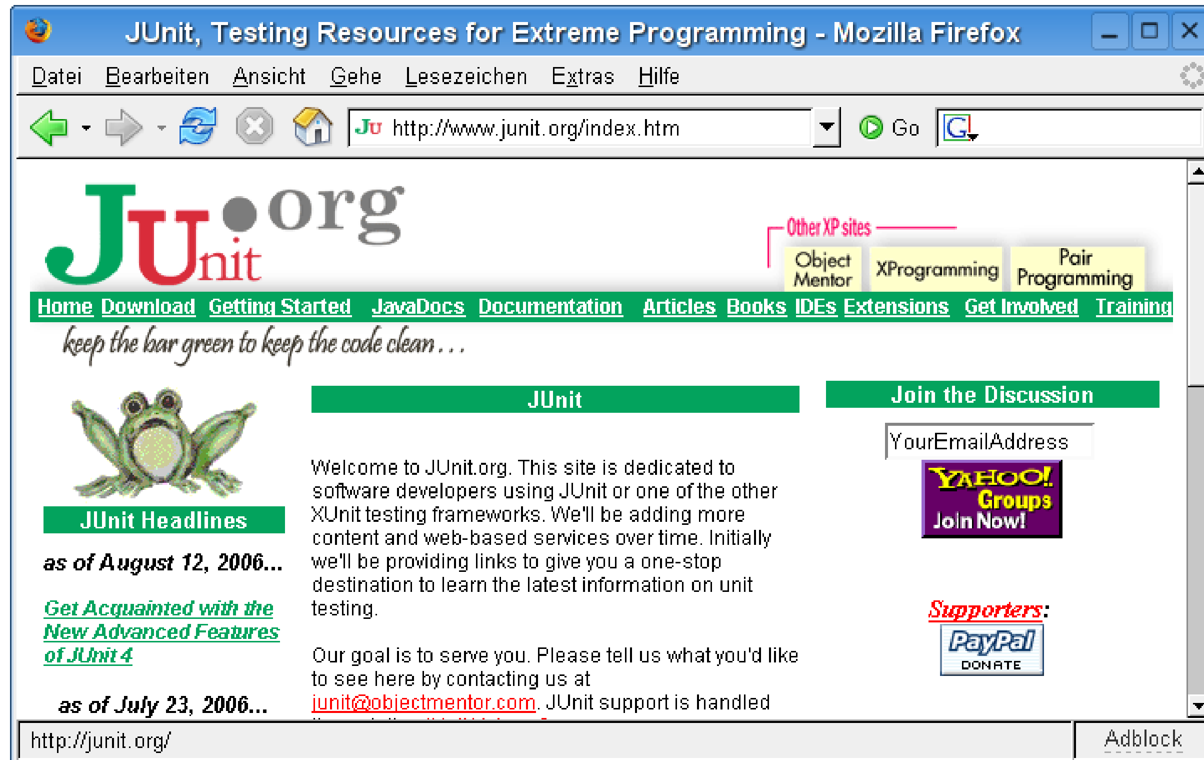
> If I had one test broken, I wanted one problem. If I had two tests broken, I wanted two problems.
>
> Source: K. Beck „Test-Driven Development“, Addison-Wesley, 2003

How should tests be constructed to achieve this ? Which property has a set of such tests ?

Test-Driven Development
JUnit

The following example in Java 1.5 makes use of the JUnit 4.* framework.

JUnit was developed by Erich Gamma and Kent Beck, is in the open source and available for download at www.junit.org:



Test-Driven Development

Managing Bookshelves in a Library

Let's write an application to manage bookshelves for a library. Bookshelves are used to keep books, so let's start with books.

Test-Driven Development

Adding a test method

A book has a title which should be retrievable by the *getTitle()* method:

```
import org.junit.Test;
import static org.junit.Assert.*;
public class BookTest
    @Test
    public void getTitle() {
        String author = "Kent Beck";
        String title = "Test-Driven Development";
        Book b = new Book(author, title);
        assertEquals(title, b.getTitle());
    }
}
```

Test-Driven Development

Author

Similar, a book has also an author which should be retrievable by the getAuthor() method. Note, that instead of changing the title test, we write the author test in an own method:

```
@Test
public void getAuthor() {
    String author = "Kent Beck";
    String title = "Test-Driven Development";
    Book b = new Book(author, title);
    assertEquals(author, b.getAuthor());
}
```

Now let's code a little bit and get the test to work (BookTest.java, Book.java).

Test-Driven Development

To pass or not to pass

Automated tests lead sooner or later to a situation where a value or object computed during the test execution must be compared with a known value or object.

- Based in the result of that comparison, the test either passed or failed.
- In JUnit, these comparisons are done with methods in the statically imported Assert class:
- Assert contains many variations of the comparison methods, including more specialized methods to compare an object against null, or a boolean expression against the true or false constants.

```
import static org.junit.Assert.*;
```

Test-Driven Development

@Ignore

Although automated tests should generally be small and easy to develop, it might happen that one or more tests are temporarily not yet ready for usage.

- In such situations tag such tests with the @Ignore annotation.
- It is good practice to provide a valid reason as an argument to the annotation, which will appear in the test log.

```
import org.junit.Ignore;
// ...
@Ignore(„Do not check the author“)
@Test
public void getAuthor() {
      // …
}
```

Test-Driven Development

Exceptions

Sometimes you want to construct tests which check, whether an exception is thrown, for example, to figure out whether the code recognizes properly bad input data.

- If the exception is not thrown, then it means that the code works not as expected, and hence, the test should fail.
- Such tests can be constructed by explictly identifying the expected exception type in the test annotation:

```
@Test(expected = NullPointerException.class)
public void getAlternativeBookshelf() {
      // …
}
```

Test-Driven Development
Timeouts

In some situations it is not only relevant whether a test passes or not, but how fast the test executes.

- Such performance oriented tests can be constructed by specifying a timeout in milliseconds.
- The test automatically fails if the test method needs more time than specified.

```
@Test(timeout = 3000)
public void loginToServer() {
      // …
}
```

Test-Driven Development
@Before and @After

A test class contains usually several tests. In our example, the BookTest class contains two tests to check the author and the title of a book.

- In both tests, String variables for the author and book title are defined, initialized and used to create a Book object.
- A SetUp method is a method which is annotated with the @Before tag and is always executed before a @Test tagged method is executed. It contains initializations which are useful for all tests.
- A TearDown method is a method which is annotated with the @After tag and is always executed after a @Test tagged method has been run.
- SetUp and TearDown methods are used for the establishment and cleanup of well defined objects where the tests are executed. Such environments are called test fixtures.

Let's have a look at the source code (BookTest2.java).

Test-Driven Development

@BeforeClass and @AfterClass

We have seen that the @Before and @After annotations are executed for each of the exisiting tests in a test class.

- To contrast, the @BeforeClass is executed once before any of the test methods in a particular test class are executed.
- An @AfterClass annotated method is executed after all test methods of that test class have been completed.

Test-Driven Development

Building test suites

To run an automated test it must be started. This is not a problem for a particular test, but what happens if you develop a system and have 500 test classes ? Which tests to run ?

Try always to run all tests. To do this in a more convenient way, build an AllTests test class:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
      BookshelfTest.class,
      BookTest.class
      })
public class AllTests {
}
```