

Test-Driven Development

Eugene Wallingford
University of Northern Iowa

wallingf@cs.uni.edu

SugarLoafPLoP'04
August 13, 2004



Abstract

In recent years, several agile software methodologies have encouraged a style of development in which programmers write the test for a piece of code *before* writing the code itself. Proponents of Test-Driven Development (TDD) argue that this style leads to simpler code that is easier to understand and easier to maintain. Further, when practiced in conjunction with aggressive refactoring, TDD is an effective way to generate simple and robust software designs.

This tutorial introduces the audience to the ideas of Test-Driven Development. First, it describes the basic practices of TDD and their role in agile software methodologies. It then shows how TDD works in concert with refactoring to create good programs. Finally, it discusses how tools such as automated testing frameworks support programmers in the disciplines of TDD.

This tutorial demonstrates TDD with examples using Java and JUnit, though the ideas presented are independent of any particular programming language or development environment.

Principles of Agile Software Development

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

Values of Agile Software Development

Communication

Simplicity

Feedback

Courage

One Set of Agile Practices: Extreme Programming

planning game

small releases

simple design

collective ownership

test-first programming

common coding standards

constant refactoring

sustainable pace

pair programming

on-site customer

continuous integration

follow system metaphor

A New Way to Write Code

Usual: Write some code, and then write the test of the code.

... It is too easy to write a test **to** the code, not **of** the code

... It is easy to write a **lot** of code before trying to test it

In TDD: Write a test, and only then the code to pass it.

... "Think before you act"

... Plus, now you will know when you are done coding

The Cycle of TDD

1. *Choose a requirement*
2. *Write a test*
3. *Write just enough code to compile the test*
4. *Run the test — it fails*
5. *Write code to pass the test*
6. *Run the test — it passes*

Write a Test for a Requirement

A Movie may have multiple ratings.
It can report its average rating.

```
public static void main( String[] args )
{
    Movie returnOfTheKing = new Movie();
    returnOfTheKing.addRating( 3 );
    returnOfTheKing.addRating( 5 );
    System.out.println( "Average rating: " +
                        returnOfTheKing.averageRating() +
                        " = " + 4 );
}
```

Compile and Run the Test

In order to **compile** this test, we need a Movie class with a default constructor and two methods, `addRating(int)` and `averageRating()`. Neither has to do anything useful!

Then we run the test, expecting it to fail. It does.

Write the Code to Pass the Test

The `Movie` must remember the values sent with an `addRating(int)` message.

In response to an `averageRating()` message, the `Movie` must compute the average value.

Run the test after each change to the code. When we have made both of these changes, the test passes.

The Cycle of TDD

1. *Choose a requirement*
2. *Write a test*
3. *Write just enough code to compile the test*
4. *Run the test — it fails*
5. *Write code to pass the test*
6. *Run the test — it passes*

7. *Refactor the code mercilessly*

An Old Idea in a New Form

... the programmer should let correctness proof and program grow hand in hand. ... If one first asks oneself what the structure of a convincing proof would be and, having found this, then constructs a program satisfying this proof's requirements then these correctness concerns turn out to be a very effective heuristic guidance.

— Edsger Dijkstra
"The Humble Programmer"
Turing Award Lecture 1972

A Key Insight: A Test is not a "Test"

Test-Driven Development is not so much about testing

Tests aren't tests —
but *specifications of expected behavior*

Tests act as assertions, as delimiters of correctness

Each test captures a single requirement in executable code

The Elements of Test-Driven Development

1. You write each test before you write the code it tests.
2. The test *determines* what code you write.
3. You maintain and run all tests throughout the course of your project.

A side effect:

an exhaustive test suite of all the code in your system

A Larger View of Tests

Tests implement small-scale *design decisions*:

... What objects do we need?

... What classes do we need?

... What are their names?

... Which objects communicate?

... By which messages?

Interactions with Other Agile Practices

Simple design

Satisfy the requirement — and no more

Constant refactoring

Evolve a meaningful large-scale design

Continuous integration

Run all tests, not just new ones

Pair programming

Ensure that programmers remain disciplined

Need for Software Tools

When done by hand, these practices are tedious and prone to human error. Software tools can provide support by automating as much of these tasks as possible.

Constant refactoring:	refactoring browser
Continuous integration:	automated testing framework

JUnit and its relatives are the most common family of tools for supporting TDD.

The JUnit Testing Framework

JUnit is a testing framework that takes care of much of the tedium involved in setting up and running any number of tests.

The basic framework provides:

- Test
- TestCase
- TestSuite
- assertions

A Demonstration of TDD using JUnit

Let's reconsider our simple `Movie` example above, using JUnit instead of a bare Java `main` method...

What happens if we add a second requirement?

A `Movie` can tell us if it has received a particular rating.

Practical Issues in TDD

- ... choosing an appropriate size test
- ... selecting which requirement to address next
- ... using stub objects for expensive or complex collaborations
- ... using mock objects and interaction-based testing
- ... letting clients specify tests

Selecting the Next Requirement

The Planning Game or some other technique identifies a set of requirements to address in the current iteration.

But in what order shall we work within the set?

Selecting the Next Requirement

Approaches:

... Work in random order, to avoid too much design too soon

... Work in an order that avoids needless undoing of work

This is an area in which experience and style provide the most guidance. Is there a right answer?

Stub Objects

Problem: Testing some behaviors is not straightforward.

... they involve expensive resources, e.g., a database

... they involve external resources, e.g., a web connection

... they require non-trivial collaborations, e.g., a network error

Solution:

Design and build *stub objects* that stand as proxies for the problem collaborator.

Mock Objects

Problem:

Testing some behaviors is more about *interaction* than about change in system state.

Solution:

Use *mock objects* that record expected interactions and verify them in the tests.

Bridging Gap Between Specification and Tests

Problem:

The client is the source of specification for the program but usually cannot write tests as code.

Solution:

Use a framework that lets clients specify tests in an intuitive form, and extract tests as code.

Reading

Dave Astels, *Test-Driven Development*, Prentice Hall, 2003

Kent Beck, *Test-Driven Development*, Addison Wesley, 2002

Edsger Dijkstra, "The Humble Programmer",
Communications of the ACM, 1972

Martin Fowler, "Mocks Aren't Stubs",
<http://www.martinfowler.com/>

Software

JUnit

<http://www.junit.org/>

Other testing frameworks

<http://c2.com/cgi-bin/wiki?TestingFramework>

<http://www.xprogramming.com/software.htm>

Mock Objects

<http://www.mockobjects.com/>

<http://www.easymock.org/>

<http://www.jmock.org/>

FIT

<http://www.XXXXX.org/>

Resources

By next week, you will be able to find these slides and links to other resources at:

<http://www.cs.uni.edu/~wallingf/>