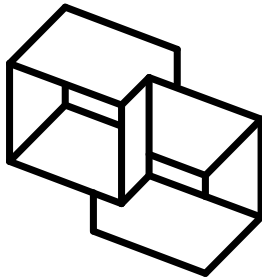©1999 McBreen.Consulting

Practical Objects

**Test Driven Software
Development using JUnit**

Pete McBreen, McBreen.Consulting
petemcbreen@acm.org

Test Driven Software Development???

The Unified Process is Use Case Driven

Requirements drive the development process, ***progress is
measured by how many Use Cases are done***

eXtreme Programming is Test Driven

Testable Requirements drive the process, ***progress is
measured by how many Functional Tests pass***

**JUnit is a freeware Java Testing Framework that
allows developers to test their own code**

Writing the tests first ***before writing the code*** ensures
that the unit tests are a real test of the implementation

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 2

---

eXtreme Programming states that *Pictures are nice, but tested code is better*

eXtreme Programming is a new methodology

It is a very high discipline process, but with the absolute minimum of mandated activities or deliverables

"Listening, Testing, Coding, Designing. **That's all there is to software. Anyone who tells you different is selling something." Kent Beck**

Extreme Programming is a minimal methodology that works with developer instincts

Delivery is a shared responsibility between developers and the user community

Design occurs by getting all developers to contribute to the product, rather than a design team and coding team

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 3

Adoption of eXtreme Programming is just starting, but we can apply the lessons now

eXtreme Programming is a developer scale, lightweight methodology based on four values

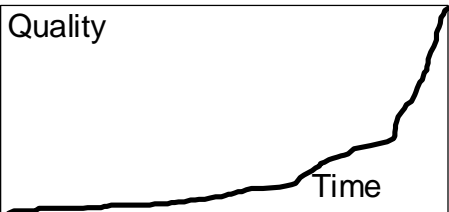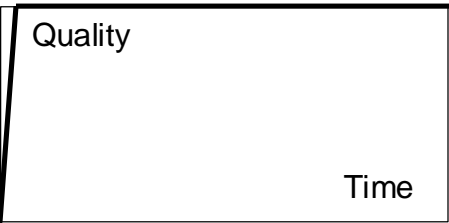Ultra high frequency *Communication*

Simplicity of both design and process

Feedback driven Activities

Courage - embrace change as a way of life, rather than build processes to limit change

***When we come to realize that change is the only reality, life is much easier* - Anon**

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 4

Applying the lessons of Extreme Programming

Steering projects

Applying the *Quality First* strategy

Using feedback to improve system quality

Big design up front does not work

Separating Business and Technical responsibilities

Incremental development means *incremental requirements capture*

Make it run, make it right, make it fast

Unit Testing and really meaning it

Avoiding *Integration Hell*

Refactoring to improve design quality

Experimentation requires a *high discipline process*

Use a process coach to tune and tailor your process

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 5

Quality First says do not add any features until you know *all existing code works*

Most projects "*add quality*" as time progresses
- Testing "*inspects quality*" into the software
- *Integration hell* is normal
- *It's feature complete, time to get the bugs out* ☺

XP gets quality to 100% as early as possible
- The system always runs with zero bugs
- Developers **incrementally capture requirements** and add features

Traditional Projects ship at the end

XP Projects can ship anytime

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 6

Quality First is a nice idea, but how do we manage to achieve it - by getting *feedback*

Feedback is essential for learning, we need to know the outcome to change our actions

> The greater the delay between action and visible outcome, the smaller are the chances of effective learning

In software development, *the best feedback occurs when we run a system*

> So it is useful to progress from Requirements, through Design and into Implementation quickly

> In order **to get the necessary speed, do incremental development**, developing just a small part each time

Just doing the activities faster is not a good idea

> Forcing the pace just increases errors

Make it run, make it right, make it fast

Developers should *make the design executable first* by writing the code

> *Running the test cases* will then identify any problems with the implementation or requirements

Then developers *make the design as simple as possible and correct*

> Rewrite for clarity, to reflect what has been learned so far and to meet project standards

Only make it fast if performance *measurements* indicate that it will be too slow for use

> Measure and profile to find where the system is spending time, and if possible apply a hardware fix

Test everything that could possibly break

Relentless Testing means that the Quality of the complete system is always known

The benefit of *no integration surprises* is only available if you have a complete set of tests that you always run *no matter how small or insignificant the change*

Tests fall into one of two categories

Unit Tests are written by the Developers, *so that they know that every method works as intended*

Functional tests are owned by the Business, *so that they know that every requirement has been satisfied*

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 9

Unit Tests are owned by developers

For all classes

Test a single class and all it's collaborators

Everything that could possibly break

Run all unit tests before any release (or checkin)

Must always score 100% ! ! !

Unit testing requires no manual intervention

Don't let the sun set on bad code

Work in short episodes, checking in tested code frequently

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 10

Functional Tests are owned by customer

Test every scenario from every Use Case

End to end

Input through output

Check your export files, database contents, printouts

Specify exactly what a requirement means

Score increases show progress

Catch regressions

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 11

Continuous Integration is a way of avoiding *Integration Hell*

The longer you wait to integrate, the longer it takes, and the more painful it is

THEREFORE: don't wait at all!

Microsoft integrates daily to weekly

eXtreme Programming projects integrate multiple times PER DAY

Take small steps and always ensure that the complete system still passes all of the tests

Deleting tests to make your code pass the tests will soon be made a capital offense

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 12

To facilitate testing of Classes, a *Test Harness* is needed for each class that is developed

The *Unit Test* Harness must be capable of fully exercising all of the behaviors of the Class

It should **create** objects, **invoke all methods**, **destroy** the objects and **validate the performance**

The *Unit Test* Harness should also test all invariants and all method pre-conditions and post-conditions

All Tests should be run as a *Smoke Test* following all changes and *prior to source code check-in*

This prevents most errors from entering the master source

A Test Harness should require *no manual intervention* to minimize developer effort

Tests that require manual intervention are rarely used

To introduce the idea of a Test Harness, a very simple Counter Class will serve as an example

A *partial* listing for the Counter Class

```
class Counter {
    public int currValue;
    public int maxValue;

    public void increment() {
        if (currValue < maxValue)
            currValue = currValue + 1;
    }
    public void decrement() {
        currValue = currValue - 1;
    }
    public void setValue(int newValue) {
        if (newValue < maxValue)
            currValue = newValue;
    }
}
```

Counter
currValue
maxValue
increment( )
decrement( )
setValue( )

A Test Harness that simply prints the output requires a manual check of the results

```
class TestCounter {
    public static void main(String argv[]) {
        Counter myCounter = new Counter(10);
        myCounter.setValue(9);
        System.out.println("Counter "+ myCounter);
        myCounter.increment();
        System.out.println("Counter "+ myCounter);
        System.exit(0);
    }
}
```

To avoid manual checking, tests should only produce output *in case of error*

```
myCounter.setValue(9);
if (myCounter.currValue != 9)
    System.out.println("Error "+ myCounter);
```

Functional Test Frameworks are necessary for testing the interactions between objects

Functional Test Frameworks should be built to support testing of the complete system

Facades simplify this since a **Command Line** or **Batch** presentation can be created to drive the application

Use Cases are the source for Functional Tests

Just because a Class is internally correct, it does not mean that there will be no *interaction errors* between objects

Using test scripts to exercise the application is essential for regression testing

Tests can run unattended after every change to the system

The presentation logic will still need to be tested, but this will occur after the total application logic has been tested

The JUnit Testing Framework for Java is an example of a Test Harness

It provides `TestCase` and `TestSuite` classes for managing the execution of Tests

```
public class MoneyTest extends TestCase
```

A ***TestCase*** supports the ***suite( )*** method that returns a set of Tests that can be executed

A Test Failure is an anticipated problem that was tested for

A Test Error was an unexpected problem that threw an exception in the code being tested

JUnit supports both GUI and TextUI execution of the same test cases

A live example of providing alternate presentations for the same underlying application logic

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 17

The Graphical User Interface provides simple feedback on the overall Progress

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 18
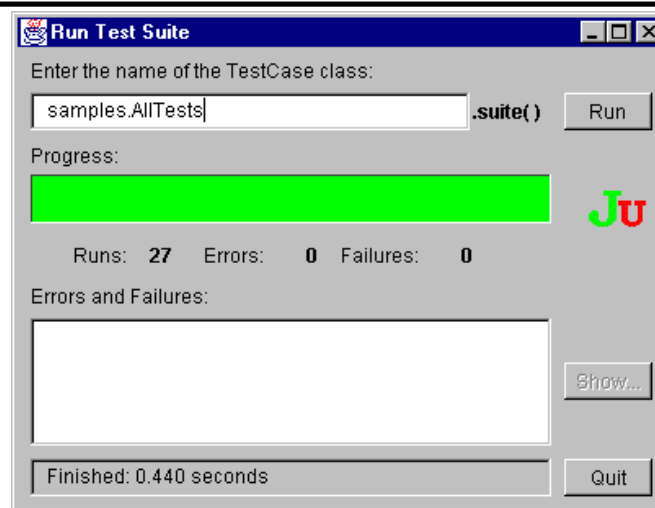
Errors and Failures are highlighted by the coloring in the progress bar

Run Test Suite

Enter the name of the TestCase class:

samples.AllTests   .suite( )   Run

Progress:

JU

Runs: **27**   Errors: **0**   Failures: **1**

Errors and Failures:

Failure: samples.money.MoneyTest.testMoneyEquals:exp

Show...

Finished: 0.490 seconds   Quit
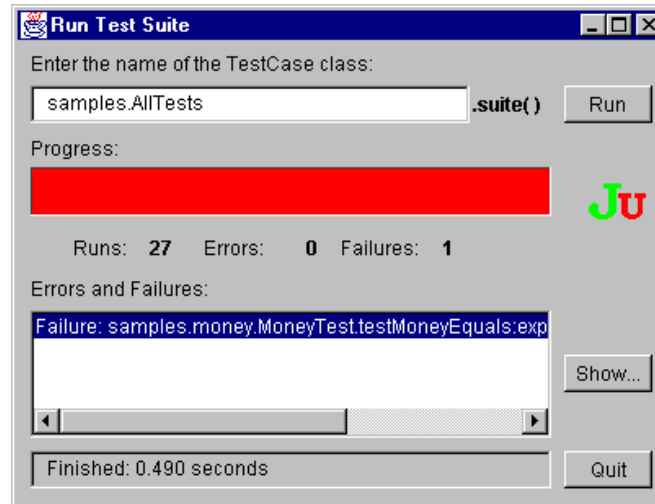
The details of each Error and Failure are captured by the thrown assertion

Stack Trace

```
test.framework.AssertionFailedError: expected:"[12 CHF]"but was:"[13 CHF]"
        at java.lang.Throwable.<init>(Compiled Code)
        at java.lang.Error.<init>(Compiled Code)
        at test.framework.AssertionFailedError.<init>(AssertionFailedError.java:13)
        at test.framework.TestCase.assert(TestCase.java:100)
        at test.framework.TestCase.assertEquals(Compiled Code)
        at samples.money.MoneyTest.testMoneyEquals(MoneyTest.java:143)
        at test.framework.TestCase.runTest(Compiled Code)
        at test.framework.TestCase.run(Compiled Code)
        at test.framework.TestSuite.run(Compiled Code)
        at test.framework.TestSuite.run(Compiled Code)
        at test.ui.TestRunner.run(TestRunner.java:1248)
        at java.lang.Thread.run(Compiled Code)
```

Close

Created by Erich Gamma and Kent Beck, *JUnit* is freely available for download and use

Documentation and source code is available at http://members.pingnet.ch/gamma/junit.htm

Instructions for use are provided on this webpage titled
Test Infected: Programmers Love Writing Tests

JUnit was ported to C++ by Michael Feathers, CppUnit is at http://www.armaties.com

There are some minor operational differences due to the
differences in language capabilities

Both implementations provide

A README file for installation

A cookbook for how to define new test cases

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 21

Using JUnit, an instance of a subclass of TestCase is created for each Unit Test

```java
import test.framework.*;
public class CounterTest extends TestCase {
    protected Counter myCounter;
    public CounterTest(String name) {
        super(name);
    }
    protected void setUp() {
        myCounter = new Counter(10);
    }
    public void testSetValue() {
        myCounter.setValue(9);
        assertEquals(9 , myCounter.currValue);
    }
    public static Test suite() {
        TestSuite suite= new TestSuite();
        suite.addTest(new CounterTest(" testSetValue"));
        return suite;
    }
}
```

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 22

The JUnit TestCase class allows a set of tests to be run independently

The *setUp()* method is invoked prior to running the method named in the TestCase constructor

```
suite.addTest(new CounterTest(" testSetValues"));
```

If this method is not present, a ***NoSuchMethodException*** will be reported as an ***Error*** by the Test Harness

```
Run: 1 Failures: 0 Errors: 1
```

The *tearDown()* method is invoked after running the method to clean up if necessary

TestCase can also support a *main* method

```
public static void main (String[] args) {
    test.textui.TestRunner.run (suite());
}
```

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 23

Designing unit tests

When to write a Unit Test is a simple decision

Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead. -- Martin Fowler

So if a **Method** could be affected by a **coding error**, write tests that ***validate that the Implementation is correct***

Functional Tests are written for every Use Case as well, but defining these is the responsibility of the Users

The Unit Tests are complete when the tests can detect any change in the software

If it is possible to change the software without the tests catching the problem then the test suite is incomplete

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 24

Designing functional tests from use cases uses the *failure conditions* from the scenarios

For each Failure Condition in a Use Case, Test Plans need test cases to cause that failure

And the usual boundary testing applies with values just above and below the threshold to detect problems
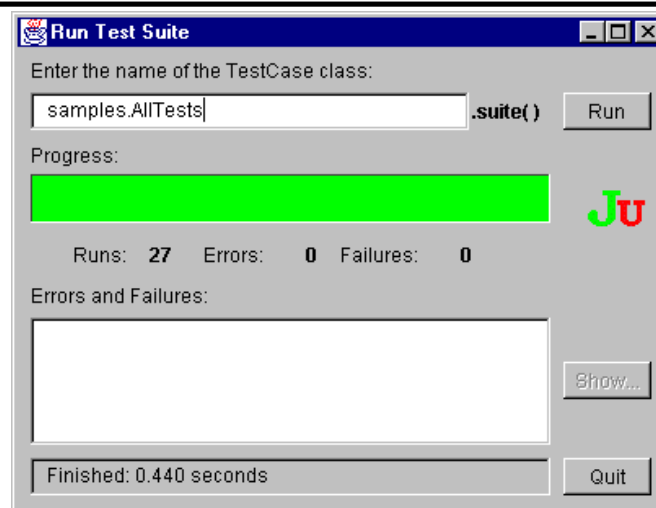
4a. Customer has exceeded credit limit
- Test 1 - Outstanding balance = credit limit
- Test 2 - Outstanding balance + order value = credit limit

Dependencies between the various tests can be identified from the different scenarios

Conditions that are only recognized in recoverable scenarios should not affect normal operation

- e.g. Preferred customer status only affects operation if Customer has exceeded credit limit

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 25

JUnit Demonstration

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 26

Closing Thoughts and Question Time

Software development is possible without a good testing harness, but it is a stressful practice

Using a good testing framework allows for less stress and much more confidence in the code

It is much faster to *Write the unit tests first*, then implement the methods

If your implementation environment does not have a port of JUnit, write your own

☺ **Remember that *software development is meant to be fun*, if it isn't, the process is wrong**

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 27

Links and references

The eXtreme Programming Website
  http://www.xprogramming.com/

Manifesto for software development
  http://members.aol.com/acockburn/manifesto.html

Software Development as a Cooperative Game
  http://members.aol.com/humansandt/papers/asgame/asgame.htm

Test Infected: Programmers Love Writing Tests
  http://members.pingnet.ch/gamma/junit.htm

Test Driven Software Development Using JUnit
  http://www.cadvision.com/roshi/talks/TestDriven.html

©McBreen.Consulting   Practical Objects: Test Driven Software Development using JUnit   Page 28