Unit Testing

Presented by Aaron Korver

Agenda

- Unit Testing
- JUnit
- Best Practices
- Are you Mocking me?
- Test Driven Development
- Summary

Agenda

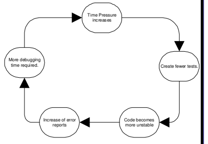
- **Unit Testing**
  - What is it?
  - Excuses for not Unit Testing.
  - Benefits
- JUnit
- Best Practices
- Are you Mocking me?
- Test Driven Development
- Integration Testing
- Code coverage tools
- Summary

What is a Unit Test?

- A unit test is a very small fast test which tests a single specific aspect of the system.
- A unit test runs in isolation of the system. It is not considered to be part of the system.
- Concentrates on the smallest unit of work in a system. With the idea being that if the foundation of the system is stable, the rest of the system will also be stable.

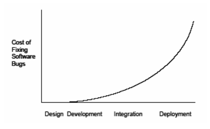
Excuses for not unit testing

- It takes too long to write a test.
  "I don't have time for testing."
  - How much time do you spend debugging code others have written?
  - How much time do you spend rewriting code you thought was working?
  - How much time do you spend in tracking down bugs?
  - This creates a vicious circle.


Time Pressure increases
Create fewer tests.
Code becomes more unstable
Increase of error reports
More debugging time required.

Excuses for not unit testing

- Testing takes time and time is money.
  - On average, fixing a bug after it has gone to a customer is 3-5 times more expensive than finding it up front.



Excuses for not unit testing

- It takes too long to run a test.
  - It shouldn't. Most tests run in a few milliseconds. If there are long running tests, they need to be either refactored or moved to a separate suite.
- Its not my job to test the code.
  - Is it your job to produce working code? If you consistently produce low quality code, it hurts your reputation.
- Testing is boring.
  - On the contrary, testing is an addicting challenging task. It also is what allows me to sleep well at night. Because all my tests passed, I know the system won't crash due to my code.

Benefits of Unit Testing

- Provides a safety net for developers.
- Promotes healthy code.
- Promotes good design, if code is easy to test, it is flexible.
- Gives confidence to the developers.
- Allows for refactoring without the fear of breaking things.

- Note that with all of these, the benefit is only as good as the tests and the test coverage.
- But, on the flip side, poor tests run often are still better than no test run at all.

Unit test framework requirements

- The developers should not have to learn a new language…therefore, the language used to specify the tests must be the programming language being used for the project.
- We need to be able to separate the application code from the test code.
- Test cases have to be executed and verified separately from one another.
- We need a way to arbitrarily group test cases into test results.
- The success or failure of a test run should be visible at a glance.
- Needs to be fast and easy to write test code.

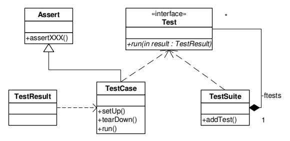
Agenda

- Unit Testing
- **JUnit**
  - Framework
  - Definitions
  - Examples
- Best Practices
- Are you Mocking me?
- Test Driven Development
- Integration Testing
- Code coverage tools
- Summary

Introducing Junit

- Part of the xUnit family.
- Other members of the family include C++Unit, PythonUnit, etc.
- Framework was designed by Erich Gamma and Kent Beck.
- Currently at version 3.8.1, though 4.0 is just around the corner.
- Multiple tools are available to help with Junit. Eclipse plugins, Ant tasks, etc.
- Also multiple addons are available due to the flexibility of the framework (Cactus, DBUnit, JunitPerf, etc).

Framework Design



JUnit Definitions

- Testcase – a collection of similar tests. Usually a one to one relationship with a class.
- Test – a function which exercises a specific method in a class.
- TestRunner – a utility that executes testcases and collects the results for display. Comes in GUI and Text flavors.
- Failure – an anticipated problem such as a failed assertion.
- Error – an unexpected problem such as an exception being raised.
- Assertion – a testable aspect of the values in a test.
- Fixture – the environment that is shared across multiple tests within a testcase.
- TestSuite – a collection of Testcases which follows the composite pattern.

World's smallest Junit

```
import junit.framework.*;
public class SimpleTest extends TestCase {
    public void testMe(){
        assertTrue(true);
    }
}
```

- All test cases must extend the class TestCase in some form.
- All tests must start with the word "test".

Assertions

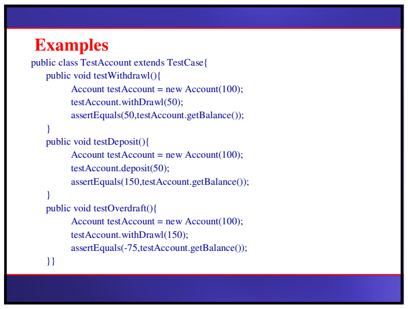
- Assertions are used to validate values in a test.
- There are many different validations that can be asserted.
- assertTrue()
- assertFalse()
- assertEquals()
- assertNull()
- assertNotNull()
- assertSame()
- assertNotSame()

Test Suites

- Suites are logical collections of TestCases.
- The TestSuite class is a Composite pattern, so a suite can contain other suites which contain other suites…etc
- Suites make it easier to run full collections of tests to validate our work.

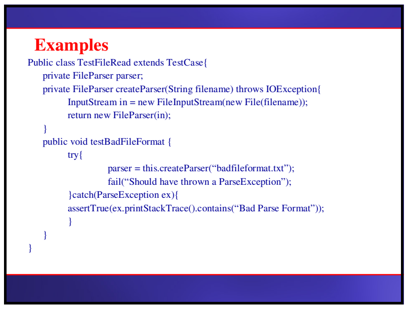
Examples

```
public void TestMath extends TestCase{
    public void testAddition(){
        assertEquals(2,Math.add(1,1));
    }
    public void testSubtraction(){
        assertEquals(0,Math.subtract(1,1));
    }
}
```



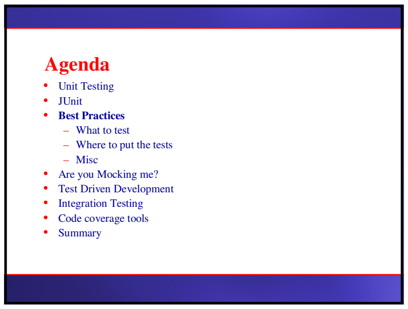
Examples

```
public class TestAccount extends TestCase{
    public void testWithdrawl(){
            Account testAccount = new Account(100);
            testAccount.withDrawl(50);
            assertEquals(50,testAccount.getBalance());
    }
    public void testDeposit(){
            Account testAccount = new Account(100);
            testAccount.deposit(50);
            assertEquals(150,testAccount.getBalance());
    }
    public void testOverdraft(){
            Account testAccount = new Account(100);
            testAccount.withDrawl(150);
            assertEquals(-75,testAccount.getBalance());
    }}
```



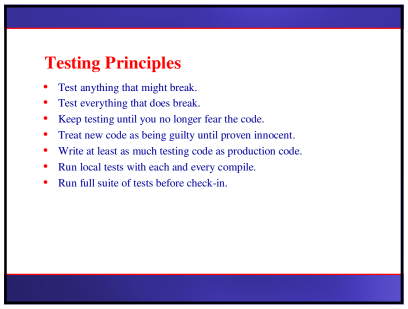
Examples

```
Public class TestFileRead extends TestCase{
    private FileParser parser;
    private FileParser createParser(String filename) throws IOException{
            InputStream in = new FileInputStream(new File(filename));
            return new FileParser(in);
    }
    public void testBadFileFormat {
            try{
                        parser = this.createParser("badfileformat.txt");
                        fail("Should have thrown a ParseException");
            }catch(ParseException ex){
            assertTrue(ex.printStackTrace().contains("Bad Parse Format"));
            }
    }
}
```



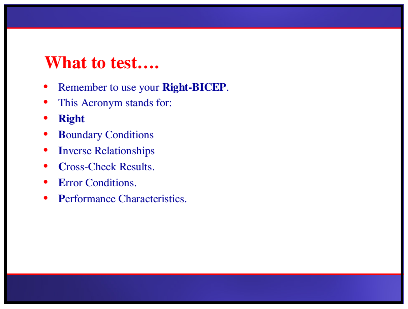
Agenda

- Unit Testing
- JUnit
- **Best Practices**
  - What to test
  - Where to put the tests
  - Misc
- Are you Mocking me?
- Test Driven Development
- Integration Testing
- Code coverage tools
- Summary



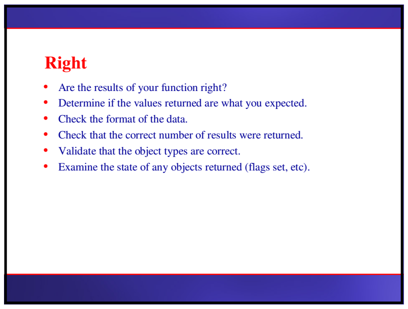
Testing Principles

- Test anything that might break.
- Test everything that does break.
- Keep testing until you no longer fear the code.
- Treat new code as being guilty until proven innocent.
- Write at least as much testing code as production code.
- Run local tests with each and every compile.
- Run full suite of tests before check-in.



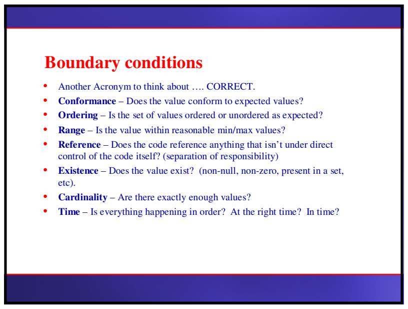
What to test….

- Remember to use your **Right-BICEP**.
- This Acronym stands for:
- **Right**
- **B**oundary Conditions
- **I**nverse Relationships
- **C**ross-Check Results.
- **E**rror Conditions.
- **P**erformance Characteristics.



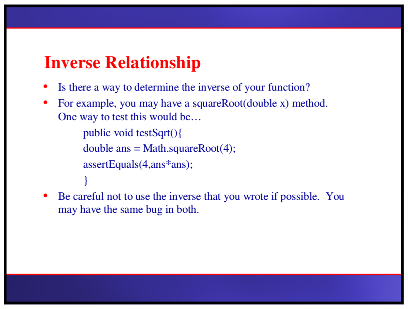
Right

- Are the results of your function right?
- Determine if the values returned are what you expected.
- Check the format of the data.
- Check that the correct number of results were returned.
- Validate that the object types are correct.
- Examine the state of any objects returned (flags set, etc).



Boundary conditions

- Another Acronym to think about …. CORRECT.
- **Conformance** – Does the value conform to expected values?
- **Ordering** – Is the set of values ordered or unordered as expected?
- **Range** – Is the value within reasonable min/max values?
- **Reference** – Does the code reference anything that isn't under direct control of the code itself? (separation of responsibility)
- **Existence** – Does the value exist? (non-null, non-zero, present in a set, etc).
- **Cardinality** – Are there exactly enough values?
- **Time** – Is everything happening in order? At the right time? In time?

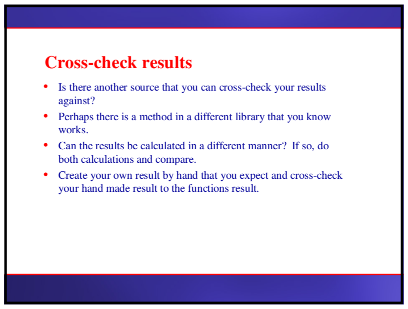


Inverse Relationship

- Is there a way to determine the inverse of your function?
- For example, you may have a squareRoot(double x) method. One way to test this would be…

```
      public void testSqrt(){
      double ans = Math.squareRoot(4);
      assertEquals(4,ans*ans);
      }
```

- Be careful not to use the inverse that you wrote if possible. You may have the same bug in both.



Cross-check results

- Is there another source that you can cross-check your results against?
- Perhaps there is a method in a different library that you know works.
- Can the results be calculated in a different manner? If so, do both calculations and compare.
- Create your own result by hand that you expect and cross-check your hand made result to the functions result.

Error Conditions

- Force checked errors to be thrown.
- Try to create environmental errors as well.
- Example of environmental errors are: Server down, disk full, system load, timeouts.

Performance

- This can be a test suite all by itself.
- Use a timer class or another decorator such as JunitPerf to test timings of methods.
- Use very large datasets and compute things such as mean average time for method calls.
- These tests could take a long time, so best to run these every night or maybe over lunch time.

Characteristics of a good test

- Good tests are A TRIP.
- **A**utomatic
- **T**horough
- **R**epeatable
- **I**ndependent
- **P**rofessional

Automatic

- Tests need to be automatic.
- Automatic as in NO Human interaction.
- All input, all button pushing, everything needs to be automatic.

Thorough

- All aspects of the function or class is tested.
- This is measured with code coverage.
- Remember to test each branch of “if-else” statements.
- Remember to test all iterations of loops.
- A measure of thoroughness is called “code coverage”.

Repeatable

- Unit tests need to be repeatable.
- Unit tests need to be repeatable.
- The tests will be run thousands of times by numerous people.
- Use of mock objects help to decouple us from real world things which could prevent repeatability.

Independent

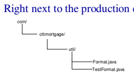
- Tests should not rely on each other.
- Only test one thing at a time.
- There is NO guarantee about the order of testcase execution when using JUnit.
- If a test needs certain state before it is run utilize the setup() and teardown() methods.
- Tests should not rely on an outside resource (database, Servlet engine, etc).

Professional

- Test code is still published to everyone.
- Any formatting or layout standards should be obeyed.
- Other people will be looking at the tests (sometimes more closely than at the real code).
- Follow the same OO principles in tests as you would in real code. This means refactoring, no duplicate code, low coupling, etc.

Where to put the tests?

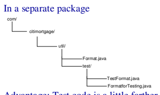
- Right next to the production code in the same package.



- Advantages: Can access protected members of the class. Makes it easy to find the tests.
- Disadvantages: Makes the workspace cluttered. Difficult to separate tests from production code when deploying.
- Recommended for small projects.

Where to put the tests?

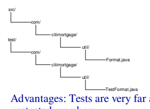
- In a separate package



- Advantage: Test code is a little farther away from prod code.
- Disadvantage: Can’t get to protected members anymore for testing.
- You can get around the disadvantage by creating a class which extends your class and exposes the protected member. Or you could use reflection.

Where to put the tests?

- Parallel source trees.



- Advantages: Tests are very far away now. Also can access protected members.
- Disadvantages: Sometimes can be difficult to find a test. Could led to CLASSPATH issues.

Naming conventions

- TestCases start with the word “Test”. This helps when doing builds as you can filter out all test classes easily. This also helps for when searching for classes with Eclipse’s easy assist to not get the classes confused by mistake.
- Tests all start with the word “test”.
- Mark incomplete tests with some sort of notation. Try placing an underscore in front of the method name.
- Use good names for methods. testConvertEuroToUSD() is a much better name than test1()

AllTests.java

- There is a quasi-standard for test suites. In each package of tests a suite named AllTests.java is created as a point for running the tests.
- At a top level is one additional AllTests.java which will run everything.

```
class AllTests{
    public static Test suite(){
        TestSuite suite = new TestSuite(“All the tests of MyProj”);
        suite.addTest(myproj.pack1.AllTests.suite());
        suite.addTest(myproj.pack2.AllTests.suite());
        suite.addTest(myproj.pack1.sub1.AllTests.suite());
        return suite;
    }
}
```

The psychology of testing

- As a developer we have an ego.
- Because of this ego, we are sometime blind to what are obvious things to test. Simply because we think the code could “never” break, we wrote it after-all….
- When writing unit tests, you must put on your testing hat. Examine the code as if it was written by another person.
- Thus test quality can be increased when working in pairs. Some companies separate the test case creators from the development creators.

Agenda

- Unit Testing
- JUnit
- Best Practices
- **Are you Mocking me?**
  - Dummy and Mock objects
  - Examples
- Test Driven Development
- Integration Testing
- Code coverage tools
- Summary

Why use Mock Objects?

- Mock objects allow us to test without bringing in external baggage.
- Most helpful when you need to have control over a third party system (such as a mainframe).
- Great for isolating behavior in a system in order to create very fine grain tests.

What are Mock Objects?

- A false implementation of an interface or class that mimics the external behavior of the true implementation.
- An object which observes how other objects interact with it’s methods and compares actual behavior with expected behavior.

Examples

- See WSAD

Spring’s Mock Objects

- Two separate packages, one for web and one for JNDI
- Very useful for testing the MVC layer.
- Most of the MVC classes need a request and response object.
- `protected ModelAndView showForm(HttpServletRequest request, HttpServletResponse response, BindException errors)throws Exception {`

```
•  MockHttpServletRequest request = new MockHttpServletRequest();
•  MockHttpServletResponse response = new
   MockHttpServletResponse();
•          UserSession user = new UserSession();
•          SessionUtil.setUserSession(request, user);
•          request.addParameter("txtSearch", "2915");
•          BindException errors = new BindException(new Object(),
   ""); ModelAndView mav = homePage.showForm(request, response,
   errors);
•          assertEquals("homepage", mav.getViewName());
•          assertEquals(4, mav.getModel().size());
•          assertNotNull(mav.getModel().get("account"));
•          assertNotNull(mav.getModel().get("primary"));
•          assertNotNull(mav.getModel().get("secondary"));
•          assertNotNull(mav.getModel().get("quicklink"));
```

EasyMock

- Instead of creating the mock object by hand, this library uses the Proxy facilities provided by the language to create implementations.
- Normally only used for mocks of interfaces, however, there is an extension for concrete classes.

Agenda

- Unit Testing
- JUnit
- Best Practices
- Are you Mocking me?
- **Test Driven Development**
- Integration Testing
- Code coverage tools
- Summary

What is it?

- Test Driven Development is a style of development that focuses on writing tests before you write production code.
- It involves lots of small baby steps to solve small requirements of the system.

How does it work?

- TDD follows a certain cycle of work in lots of micro-iterations.

1. Write a test case to solve a requirement.
2. Run the test case to prove that it fails.
3. Write code to make the test pass.
4. Refactor code to remove any duplication. At this point good design just starts to happen.
5. Re-run all tests to make sure everything still works.
6. Go back to step 1 until all requirements are met.

RED/GREEN/REFACTOR

Benefits of using TDD

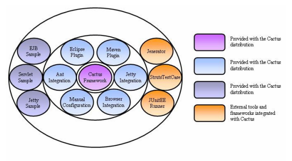
- Code that works.
- Fast feedback
- Flexibility – this is a key benefit as you will always be changing what doesn’t work as part of the Refactor stage.
- A constant stream of slow forward progress, with continuously working code.
- Happy customers
- Satisfied developers
- Did I mention the code always works?

Agenda

- Unit Testing
- JUnit
- Best Practices
- Are you Mocking me?
- Test Driven Development
- **Integration Testing**
  - Cactus
  - DbUnit
  - Spring
- Code coverage tools
- Summary

Cactus

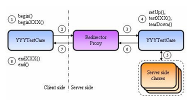
- This package is from Apache.



The Cactus Ecosystem

Cactus

- Cactus allows for “incontainer” testing.
- This means that you can do integration testing with the actual resources that your application is using.
- The tests are executed on the server itself.



Cactus and Spring

-
```
public class TestMyService extends ServletTestCase
{
   WebApplicationContext ctx;
   /**
    * Constructor for TestMyService.
    * @param arg0
    */
   public TestMyService(String arg0)
   {
      super(arg0);
   }
   /*
    * @see TestCase#setUp()
    */
   protected void setUp() throws Exception
   {
      //config is an instance of
      //org.apache.cactus.server.ServletConfigWrapper
      //which inherits from javax.servlet.ServletConfig
   ctx = WebApplicationContextUtils.getWebApplicationContext(config.getServletContext());
   }

   public void testXXX()
   {
      Bean1 bean1 = (Bean1)ctx.getBean("bean1Service");
      Bean2 bean2 = (Bean2)ctx.getBean("bean2Service");
      Map types = bean1.getTypes();
      User uesr = bean2.getUser("user1", "pass1");
      for(Iterator iterator = types.entrySet().iterator(); iterator.hasNext();)
      {
         Map.Entry entry = (Map.Entry)iterator.next();
         System.out.println(entry.getKey() + " = " + entry.getValue());
      }
   }
}
```

DbUnit

- DbUnit is another tool to help with integration testing.
- It solves the problem of how to make sure that the database is in the same state for every test.
- Remember that tests need to be repeatable, so the database must be in some known state.
- It does this by using XML to create datasets and then import this data into the database.

Spring’s Transaction TestCases

- **AbstractTransactionalSpringContextTests**
- This is a super class that you can extend to get a rollback() at the end of your tests.
- The idea is that you start with the database in state A, do some work, verify that work, then rollback.
- With this parent class the rolling back will happen automagically for you.

Agenda

- Unit Testing
- JUnit
- Best Practices
- Are you Mocking me?
- Test Driven Development
- Integration Testing
- **Code coverage tools**
  - Emma
  - Cobertura
- Summary

What is code coverage?

- Code coverage is the measure of the executable amount of code that was executed during a given execution.
- In other words, it is a measurement to see how much code your tests currently cover.
- 80-90% is extremely good test coverage.

Emma and Cobertura

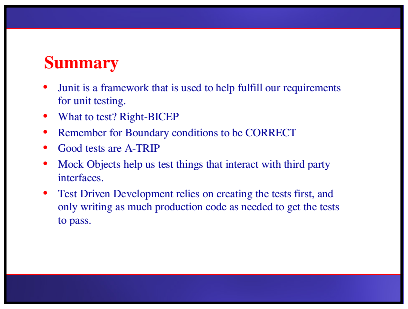
- These are both tools that are used to help determine code coverage.
- Both work on the idea of modifying the byte code.
- When the byte code has been modified then the unit tests are run. Because of the additional information in the byte code, these tools are able to produce reports that show the code coverage.
- Cobertura is also able to fail a build if the code coverage is too low.

Commercial code coverage tools

- Clover
- Agitar

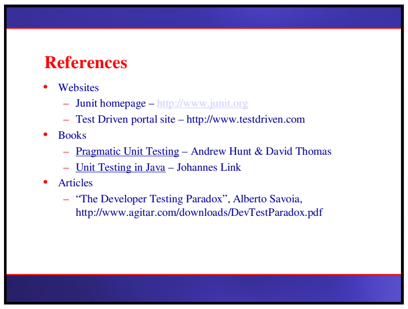
Agenda

- Unit Testing
- JUnit
- Best Practices
- Are you Mocking me?
- Test Driven Development
- Integration Testing
- Code coverage tools
- **Summary**



Summary

- Junit is a framework that is used to help fulfill our requirements for unit testing.
- What to test? Right-BICEP
- Remember for Boundary conditions to be CORRECT
- Good tests are A-TRIP
- Mock Objects help us test things that interact with third party interfaces.
- Test Driven Development relies on creating the tests first, and only writing as much production code as needed to get the tests to pass.



References

- Websites
  - Junit homepage – http://www.junit.org
  - Test Driven portal site – http://www.testdriven.com
- Books
  - Pragmatic Unit Testing – Andrew Hunt & David Thomas
  - Unit Testing in Java – Johannes Link
- Articles
  - “The Developer Testing Paradox”, Alberto Savoia, http://www.agitar.com/downloads/DevTestParadox.pdf