Unit testing, Integration testing
TDD (Test Driven Development)
Mocking

by Micael Andersson

Goals of todays presentation

- Automated Testing Theory (Brief overview)
  - To introduce and motivate Automated Tests
  - To describe how Automated Testing fits within a Software Development Process
  - To provide a classification of Automated Testing Strategies and Tools
- Unit Testing (Brief overview)
  - Provide theoretical background for Unit Testing
  - Hands-on experience with Unit Testing tools and frameworks
- TDD - Test Driven Development (Agile development)
  - Process, In Practice, Benefits

Goals of next presentation

- Integration Testing
  - Provide theoretical background for and overview of Integration Testing
  - Overview of Integration Testing tools and frameworks
- Design for testability
  - Some design patterns
- Mocking ()
  - Basic principles
  - Hands-on experience with Mockito

Training material - Java

- These slides...
- Online resources
  - www.junit.org
  - code.google.com/p/mockito/
  - www.dbunit.org
  - strutstestcase.sourceforge.net
- Source code to exercises

Testing
and
Automated tests

Training provided by
Combitech Grow

About Tests …

Everybody knows they should, but few actually do

- “Why isn’t this tested before”?
  - Because it has been too expensive, difficult, cumbersome to test
  - Because we have been too busy
  - Because things have changed

Discussion:
In case the application you currently work on lacks tests;
- In your opinion; what’s the main reason for this?

Quality Assurance precedes Quality Assessment

- Testing is about Quality Assurance, not just Quality Assessment
- Quality Assessment only indirectly affect quality
- Testing reveals information
- Testing helps focus project activity

Test Automation Goals

Tests should be S.M.A.R.T:

- Self Checking
- Maintainable
- Act as documentation
- Repeatable and Robust
- To the point – provide ”defect triangulation”

Manual Tests are …

- Repetitive
- Error-prone
- Difficult to test other units than the User Interface

- yet …

- a (Manual) Test Process must be present in order to automate it!

COMBITECH

Critical Success Factors for Automated Tests

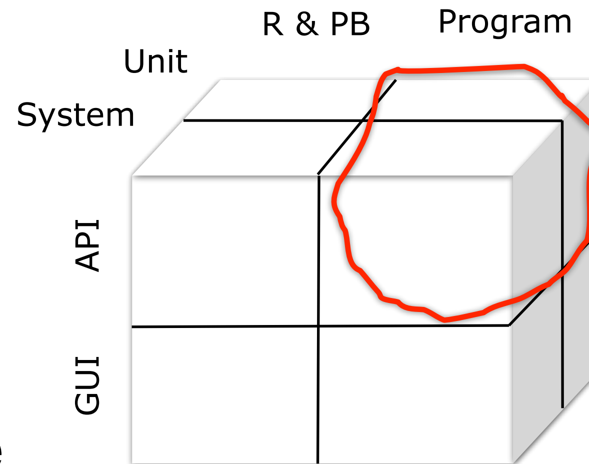
- **Repeatability and Consistency**
  - Once the test is complete, it should pass repeatedly, whether it executes by itself or within a test suite.
  - When a completed test fails, we need to quickly and accurately pinpoint the cause: did the test uncover a bug in the system, or is the test itself faulty?
- **Readability**
  - The tests are the definitive reference for the system requirements.
- **Maintainability**
  - Iterative, test-first development yields as much (or more) test code than system code
  - Thus we have to be as concerned (or more) with the maintenance costs of test code as compared to system code.

Testability

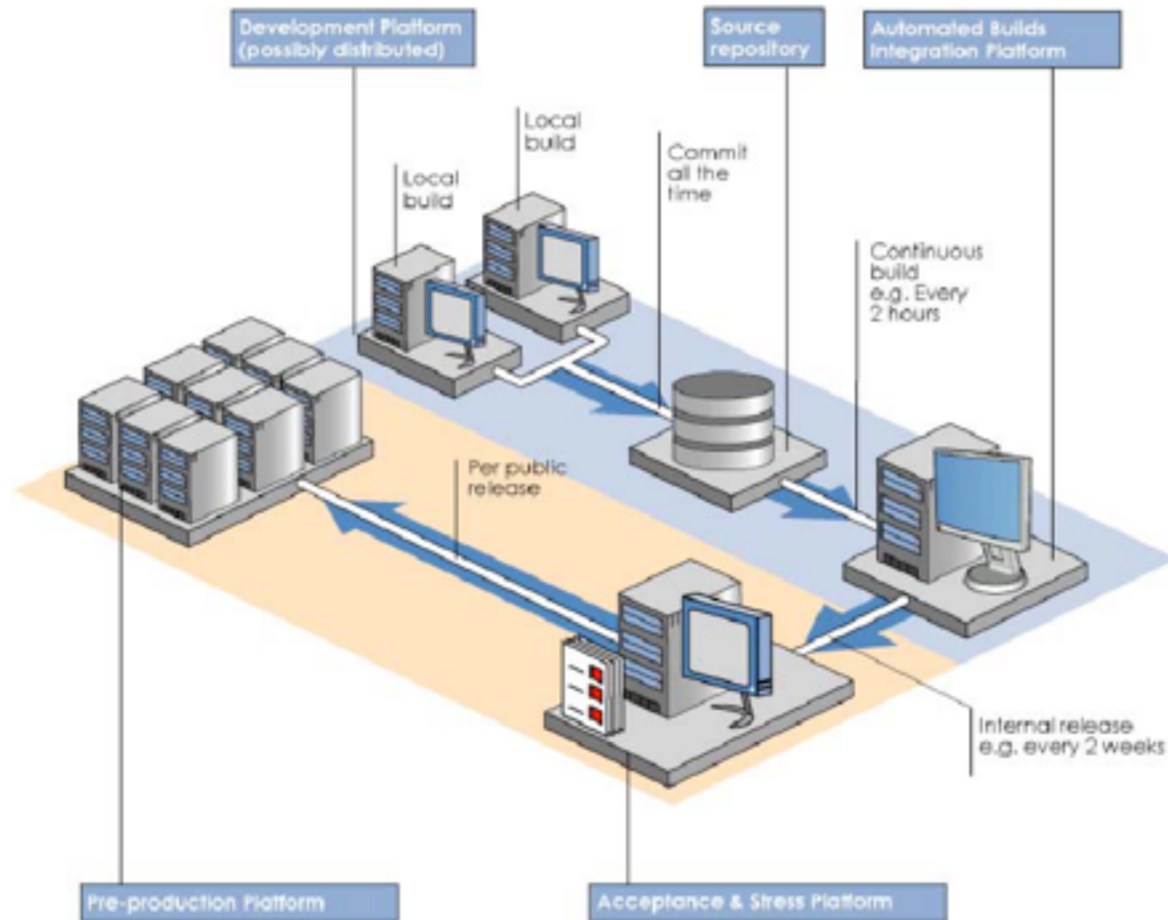
- Testability consists of two fundamental characteristics:
  - Visibility – the tester can see (and understand) what happens within the system (i.e. can observe important aspects of the internal state of the system)
  - Control – the tester can force interesting things to happen within the system (i.e. can control its behavior)
- Testability doesn't just happen. It must be designed and built into a system. Writing tests before designing and building the system (a.k.a. Test-First or Test-Driven Development) is a great way of achieving good testability.

Classifying Automated Tests

- **Granularity**
  - Entire system
  - Individual units
- **Point of Contact**
  - Existing User Interface
  - Testability API
- **Test Case Production**
  - Record & Play Back
  - Hand Written (programmatic)



Test (and build) automation

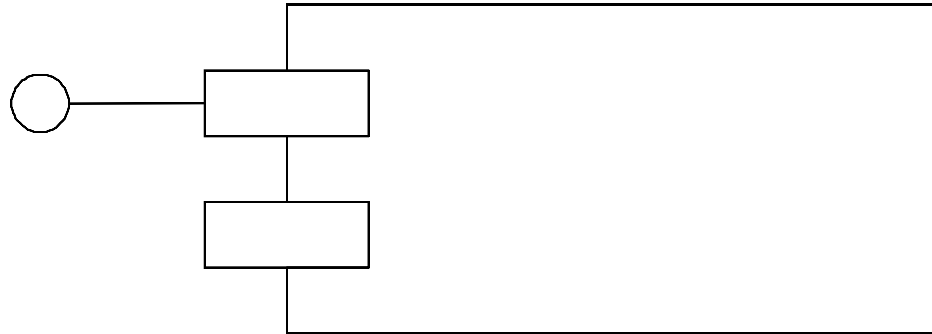


Introduction
to
Unit Testing with JUnit and Eclipse

Training provided by
Combitech Grow

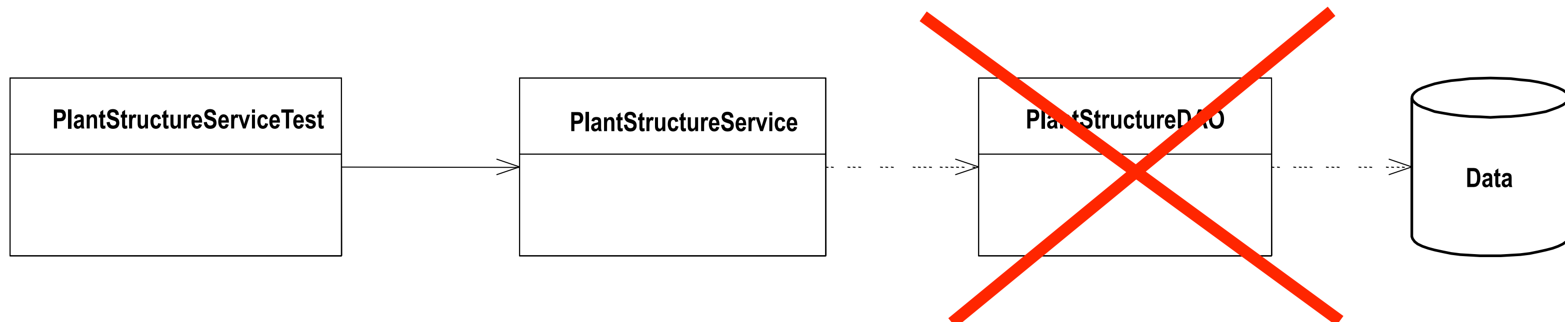
Unit Tests

- Black-box or White-box test of a *logical unit*, which verifies that the logical unit behaves correctly – *honors its contract*.



What exactly is a Unit Test?

- A self-contained software module (in OO languages typically a Class) containing one or more test scenarios which tests a Unit Under Test in isolation.
- Each test scenario is autonomous, and tests a separate aspect of the Unit Under Test.



Smoke Tests

- A set of Unit Tests (which tests a set of logical units) executed as a whole provides a way to perform a *Smoke Test*: Turn it on, and make sure that it doesn't come smoke out of it!
- A relatively cheap way to see that the units "seems to be working and fit together", even though there are no guarantees for its overall function (which requires functional testing)

Developer testing vs Acceptance testing

- Unit Tests are written by developers, for developers.
- Unit Tests do not address formal validation and verification of correctness (even though it has indirect impact on it!) - Unit Tests prove that some code does what we intended it to do
- Unit Tests complements Acceptance Tests (it does not replace it)

Why should I (as a Developer) bother?

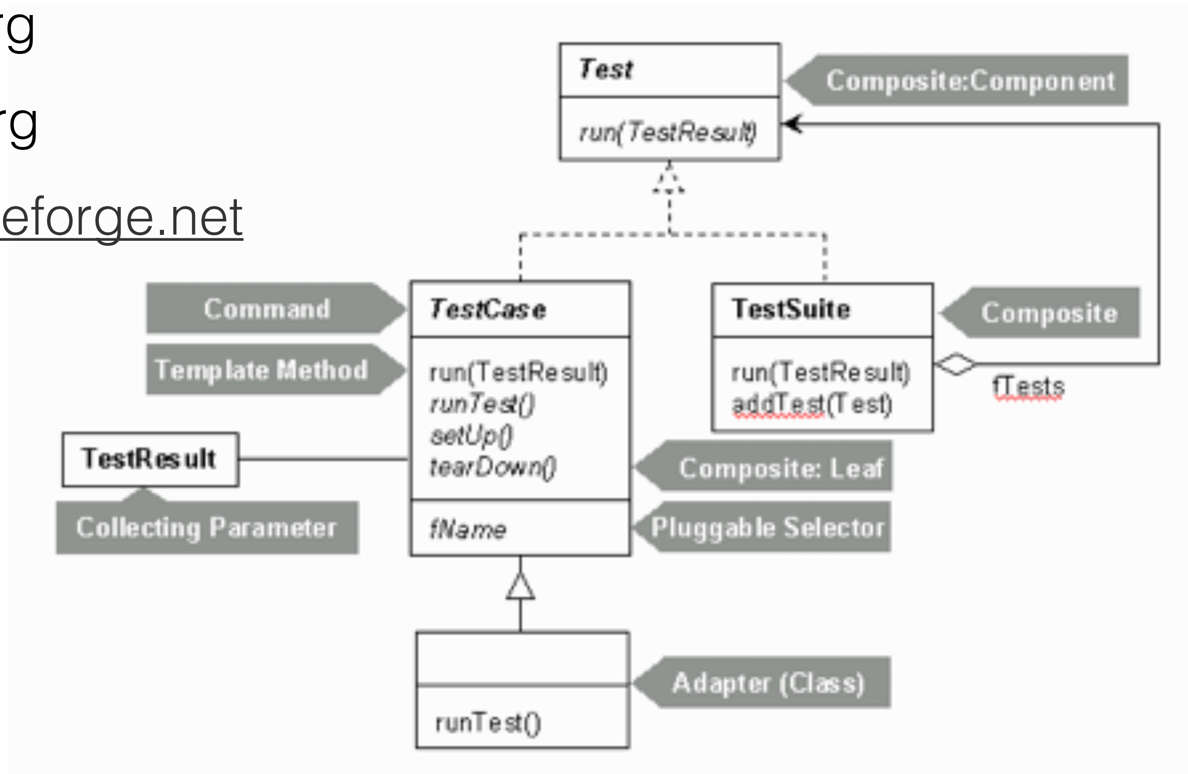
- Well-tested code works better. Customers like it better.
- Tests support refactoring. Since we want to ship useful function early and often, we know that we'll be evolving the design with refactoring.
- Tests give us confidence. We're able to work with less stress, and we're not afraid to experiment as we go.
- Hence Unit Testing will make my life easier
  - It will make my design better
  - It will give me the confidence needed to refactor when necessary
  - It will dramatically reduce the time I spend with the debugger
  - It will make me sleep better when deadlines are closing in

Requirements on Unit Tests

- Easy to write a test class
- Easy to find test classes
- Easy to test different aspects of a contract
- Easy to maintain tests
- Easy to run tests

XUnit: A Framework for Unit Tests

- www.junit.org
- www.csunit.org
- www.vbunit.org
- cppunit.sourceforge.net



JUnit Test Example

```
public interface Account {
   public void deposit(int amount);
   public void withdraw(int amount) throws AccountException;
   public int getBalance();
      …
}

public class AccountImplTest {
    @Test
    public void testWithdraw() throws AccountException {
        AccountImpl account = new AccountImpl("1234-9999", 2000);
        account.withdraw(300);
        Assert.assertEquals(1700, account.getBalance());
    }

    @Test
   public void testWithdrawTooMuch() throws AccountException { … }
    …
}
```

Naming Conventions and Directory Structure

- Unit Tests should be named after the Unit that is tested, with "Test" appended.
  A class usually represents a noun, it is a model of a concept. An instance of one of your tests would be a 'MyUnit test'. In contrast, a method would model some kind of action, like 'test [the] calculate [method]'.

- the MyUnit test --> MyUnitTest
- test the calculate method --> testCalculate()
- JUnit tests should be placed within the same Java package as the Unit under Test, but in a different directory structure.

COMBITECH

Test cases and test methods

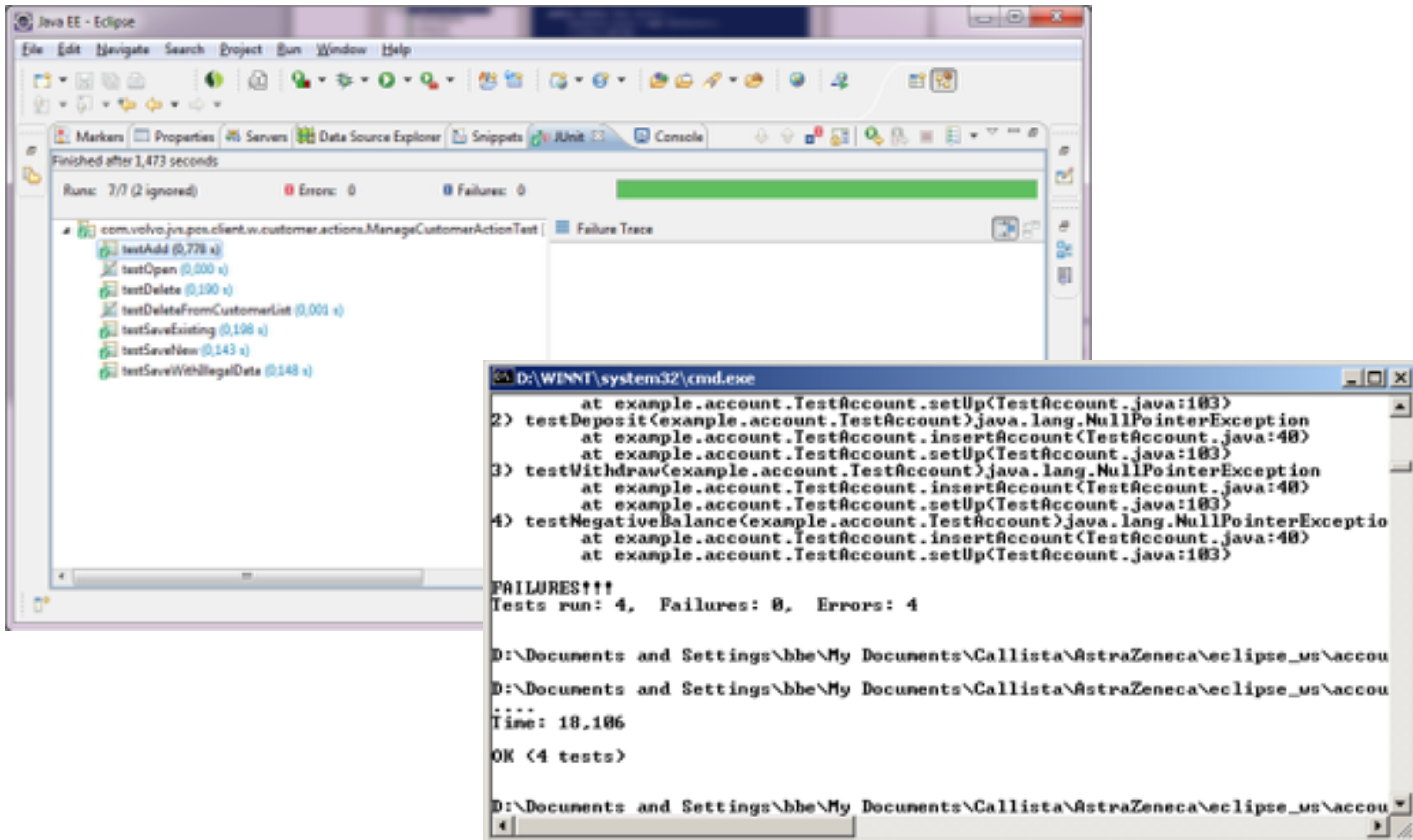
```
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class AccountImplTest {
    @Test
    public void testWithdraw() throws AccountException {
        AccountImpl account = new AccountImpl("1234-9999", 2000);
        account.withdraw(300);
        Assert.assertEquals(1700, account.getBalance());
    }

    @Test
   public void testWithdrawTooMuch() throws AccountException { … }
    …
}
```

All methods annotated with @Test are considered test scenarios

Assert: Support for verifying conditions

```
static void assertEquals(int expected, int actual);
          // Asserts that two ints are equal.

static void assertEquals(double expected, double actual, double delta);
          // Asserts that two doubles are equal concerning a delta.

static void assertEquals(java.lang.Object expected, java.lang.Object actual);
          // Asserts that two objects are equal.

static void assertFalse(java.lang.String message, boolean condition);
          // Asserts that a condition is false.

static void assertTrue (java.lang.String message, boolean condition);
          // Asserts that a condition is true.

static void assertNull(java.lang.String message, java.lang.Object object);
          // Asserts that an object is null.

static void assertNotNull(java.lang.String message, java.lang.Object object);
          // Asserts that an object isn't null.

// Etc…
```

Executing JUnit Tests: Test Runners



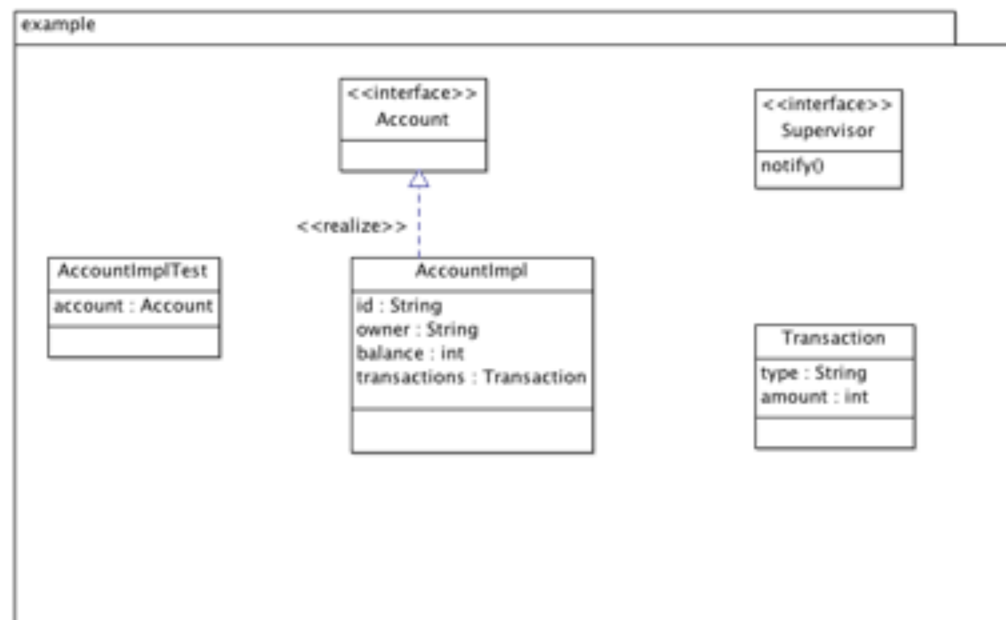
Exercise 1

(Writing unit tests to Account)

Training provided by
Combitech Grow

Exercise 1 description

Sometimes you need to write Unit tests to already existing software when you want to implement a change request e.t.c. In this example we have the source code but no tests, your task is to write them.



Exercise 1

- Create an Unit test case which tests the initial balance of an Account (i e. tests the constructor and GetBalance() method of Account).

```
@Test
public void testInitialBalance() { … }
```

- Add tests for the Deposit() method of Account.

```
@Test
public void testDeposit() { … }
```

Continued introduction to Unit Testing with JUnit and Eclipse

Training provided by
Combitech Grow

Typical unit test scenario – The Three A's

1. **A**rrange - Instantiate Unit under Test and set up test data
2. **A**ct - Execute one or more methods on the Unit Under Test
3. **A**ssert - Verify the results

```
public interface Account {
   public void deposit(int amount);
   public void withdraw(int amount) throws AccountException;
   public int getBalance();
    …
}
public class AccountImplTest {
     @Test
     public void testWithdraw() throws AccountException {
         AccountImpl account = new AccountImpl("1234-9999", 2000);       // ARRANGE
         account.withdraw(300);                                          // ACT
         Assert.assertEquals(1700, account.getBalance());          // ASSERT
     }
       … … …
}
```

General Rules of Thumb

- Create a single test class for each non-trivial application class you have.
- Give a readable, meaningful name to each test method. A good name candidates are to name the test method using the same name as the method that it is testing, with some additional info appended to the name. For instance if testing a method called "***Withdraw***" in an Account class, create a few test methods to test different ways of withdrawal:

```
@Test
public void testWithdrawTooMuch() throws AccountException {…}
@Test
public void withdrawBigAmount() throws AccountException {…}
@Test
public void withdrawNegativeAmount() throws AccountException {…}
```

- The scope of how much checking to do in a single test case (test method) is a judgment call. It is usually better to test only one scenario (and hence one potential error condition) in each test method. Remember : tests should be “to the point”.

Setup and teardown

- Methods annotated with **`@Before`** are executed *before every test method*.
- Methods annotated with **`@After`** are executed *after every test method*.

```
public class AccountImplTest {

        private AccountImpl account;

        @Before
        public void setUp() {
                account = new AccountImpl("1234-9999", 2000);
        }
        @Test
        public void testInitialBalance() {
                int actualBalance = account.getBalance();
                Assert.assertEquals(2000, actualBalance);
        }
        @Test
        public void testWithdraw() throws AccountException {
                account.withdraw(300);
                int actualBalance = account.getBalance();
                Assert.assertEquals(1700, actualBalance);
        }
    …
}
```

Working with Exceptions

- Unexpected exceptions thrown during execution of a test will be caught by the JUnit framework and reported as Errors (i.e. test will fail)
- A Test method must declare that it throws any checked exceptions that the Unit under Test may throw. If there are several checked exceptions that may occur, it is perfectly valid for a test method to declare throwing java.lang.Exception.
- Expected exceptions (exceptions that the test is expecting the Unit under Test should throw in a certain situation) are expressed using the **`@Test(expected=ExpectedException.class)`** attribute

```
@Test(expected=NastyException.class)
public void doSomethingNastyTest() {
    SomeUnit target = new SomeUnit();
    target.doSomethingNasty();
}
```

Working with Exceptions (Contd.)

- Or using the following idiom:

```
SomeUnit target = new SomeUnit();
try {
   target.doSomethingNasty();
   Assert.fail("NastyException expected");
} catch (NastyException expected) {
   // Expected
}
```

Ignore a Test

- To temporary ignore a test, use the Ignore attribute:

```
@Test
@Ignore("Not right now, but most definitely later")
public void testThatDoesNotWorkYet(){
    SomeUnit target = new SomeUnit();
    target.doSomethingThatDoesNotWork();
    Assert.assertTrue(target.isValid());
}
```

Exercise 2

(refactor unit tests to Account)

Training provided by
Combitech Grow

Exercise 2

- Refactor your test data from the last example into a @Before annotated setUp() method

- Add tests for the withdraw() method.

Continued introduction
to
Unit Testing with JUnit and Eclipse

Training provided by
Combitech Grow

Testing private or protected methods/members

JUnit will only test those methods in my class that are public or protected, but…

In principle you got four options

- Don't test private methods. (Good or Bad?)
- Give the methods package-private access. (Good or Bad?)
- Use a inner class or anonymous class. (Does it work?)
- Use reflection. (Is this good?)

http://stackoverflow.com/questions/34571/whats-the-proper-way-to-test-a-class-with-private-methods-using-junit

Testing private or protected methods/members

The best way to test a private method is via another public method. If this cannot be done, then one of the following conditions is true:

1. The private method is dead code.
2. There is a design smell near the class that you are testing.
3. The method that you are trying to test should not be private.

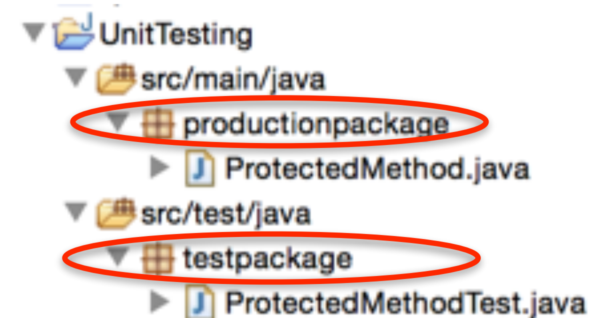
When I have private methods in a class that is sufficiently complicated that I feel the need to test the private methods directly, that could be a code smell: my class is too complicated.

But, it might also be SDK or Framework code or Security or encryption/decryption code. That type of code also need tests, but no publicity…

Testing protected methods (Java)

- Protected methods are visible by default when using the same parallel package structure for tests, but if in different packages, it will not work!

```
package productionpackage;
public class ProtectedMethod {
    protected String myProtectedMethod (String s) {
        return "MyClass: " + s;    }
}

package testpackage;
public class ProtectedMethodTest {

    @Test
    public void testProtectedMethod() {
        String expected = "MyClass: Hello";
        ProtectedMethod unitUnderTest = new ProtectedMethod();
        String actual = unitUnderTest.myProtectedMethod("Hello");
        boolean equal = actual.equalsIgnoreCase(expected);
        Assert.assertTrue("Strings not equal", equal);
    }
}
```



Will not work!

Testing protected methods (Java)

The Subclass and Override idiom is used to write unit tests for protected methods:

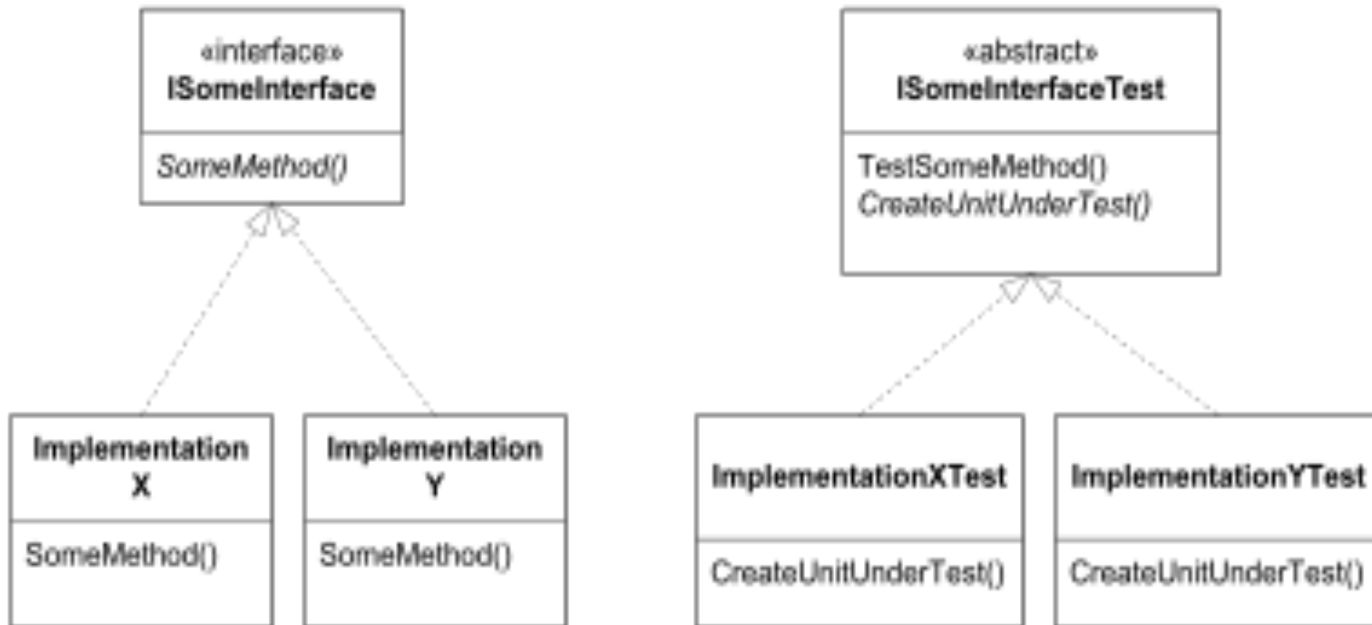
```
package productionpackage;
public class ProtectedMethodClass {
	protected String protectedMethod (String s) {
		return "Protected: " + s;	}
}

package testpackage;
public class ProtectedMethodClassTest {

	// Create an inner class to expose the protected method
	class ExposeProtectedMethodClass extends ProtectedMethodClass {
		public String exposeProtectedMethod(String s) {
			return super.protectedMethod(s);
		}
	}
	@Test
	public void testProtectedMethod() {
		String expected = "Protected: Hello";
		ExposeProtectedMethodClass unitUnderTest = new ExposeProtectedMethodClass();
		String actual = unitUnderTest.exposeProtectedMethod("Hello");
		boolean equal = actual.equalsIgnoreCase(expected);
		Assert.assertTrue("Strings not equal", equal);
	}
}
```

We can live with this since the exposure is done in test package, that will be stripped out in the production code!

COMBITECH

Testing Interfaces or Abstract Classes (Java only)

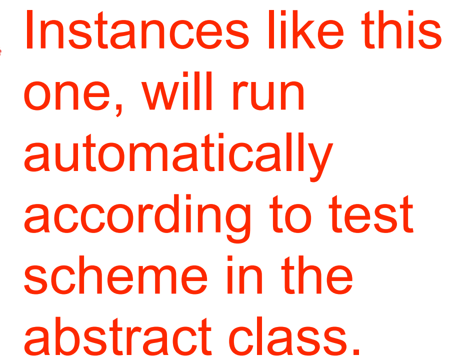
- Sometimes, you want to write tests for an Interface or Abstract Class, and have those tests executed against all implementations.
- Specify the tests in an Abstract Test class, with one concrete Test class for each concrete implementation



Testing Interfaces - Java example

```java
package somepackage;
public class ImplementationX implements SomeInterface {
        @Override
        public boolean someMethod() {
                return false;
        }
}
```

```java
package somepackage;
import org.junit.*;

public abstract class AbstractSomeInterfaceTest {
    private SomeInterface unitUnderTest;
    @Before
    public void setUp() {
        unitUnderTest = implementSomeInterfaceTest();
    }
    @Test
    public void testSomeMethodReturnsTrue () {
        Assert.assertTrue("someMethod() should return true", unitUnderTest.someMethod());
    }
    protected abstract SomeInterface implementSomeInterfaceTest();
}

public class ImplementationXTest extends SomeInterfaceTest {
      @Override
      protected SomeInterface implementSomeInterfaceTest() {
         return new ImplementationX();
      }
}
```



Exercise 3
(refactor unit tests to Account)

Training provided by
Combitech Grow

Exercise 3 - As Demo

- Refactor your test case from the last example into an abstract test case for the interface Account, with a concrete test of the Account implementation
- Show it by running project TDD_Account_Solution concrete implementation ConcretAccountTest.

What should be tested?

- Everything that could possibly break!
- Corollary: Don’t test stuff that is too simple to break!
- Typical problematic areas:
  - Boundary conditions
    - **C**onformance
    - **O**rdering
    - **R**ange
    - **R**eference
    - **E**xistence
    - **C**ardinality
    - **T**ime
  - Error conditions

Exercise 4

Given the following interface for a fax sender service:

```
/* Send the named file as a fax to the given phone number.
* Phone numbers should be of the form 0nn-nnnnnn where n is
* digit in the range [0-9]
*/
public boolean SendFax(String phone, String filename) {
    . . .
}
```

What tests for boundary conditions can you think of?

Introduction to TDD (Test-Driven Development)

Training provided by
Combitech Grow

COMBITECH

Test-Driven Development

Unit Tests may be written very early. In fact, they may even be written before any production code exists:

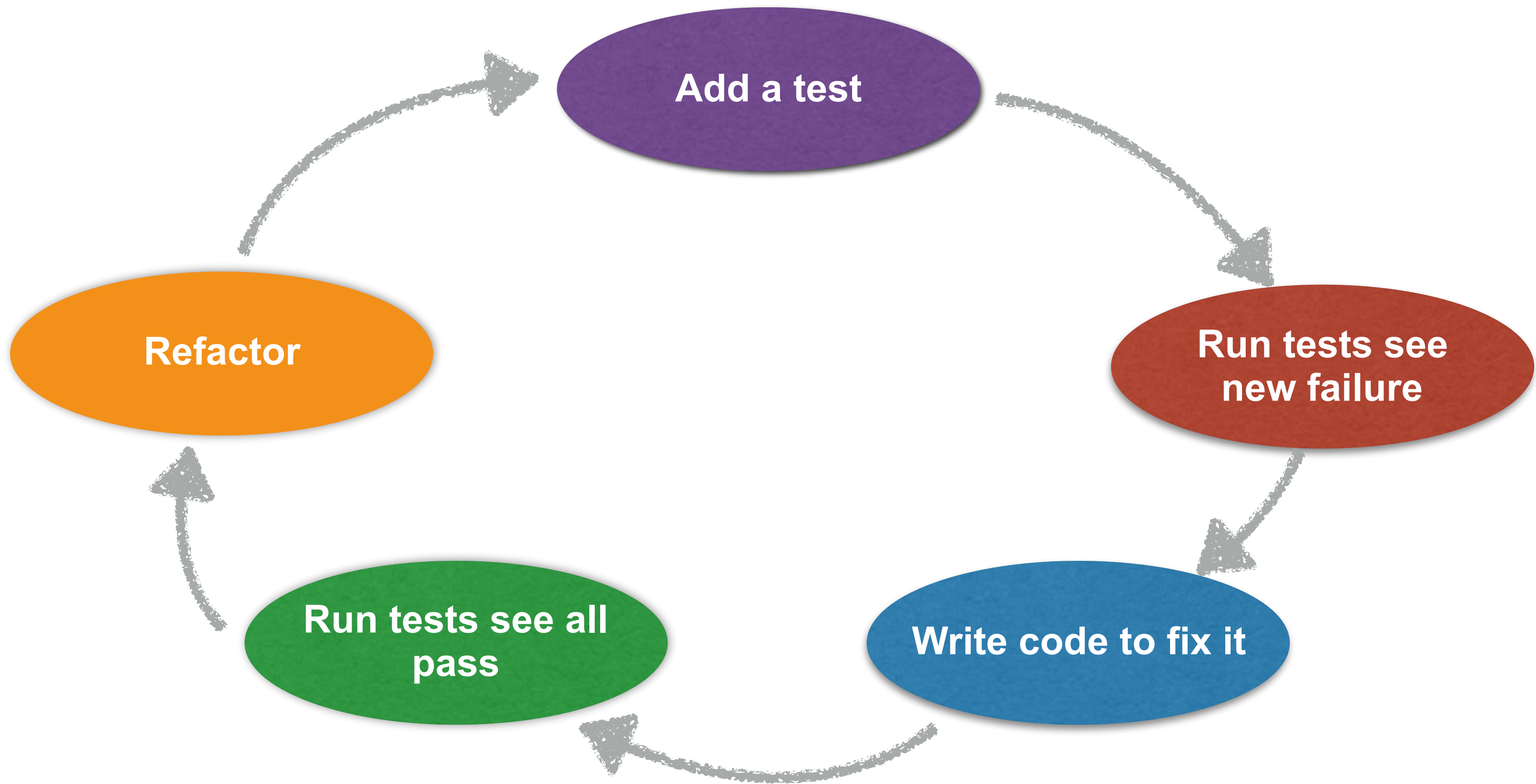
- Write a test that specifies a tiny bit of functionality
- Ensure the test fails (you haven't built the functionality yet!)
- Write the code necessary to make the test pass
- Refactoring the code to remove redundancy

There is a certain rhythm to it: Design a little – test a little – code a little – design a little – test a little – code a little – ...

Test-Driven Development process

1. Think about what you want to do.
2. Think about how to test it.
3. Write a small test. Think about the desired API.
4. Write just enough code to fail the test.
5. Run and watch the test fail (and you'll get the "**Red Bar**").
6. Write just enough code to pass the test (and pass all your previous tests).
7. Run and watch all of the tests pass (and you'll get the "**Green Bar**").
8. If you have any duplicate logic, or inexpressive code, **refactor** to remove duplication and increase expressiveness.
9. Run the tests again (you should still have the "Green Bar").
10. Repeat the steps above until you can't find any more tests that drive writing new code.

Test-Driven Development process (TDD process)



Simple Design

- "Simplicity is more complicated than you think. But it's well worth it."

  Ron Jeffries

- Satisfy Requirements
  - No Less
  - No More

You can use your developer intuition to find best choice

COMBITECH

Simple Design Criteria

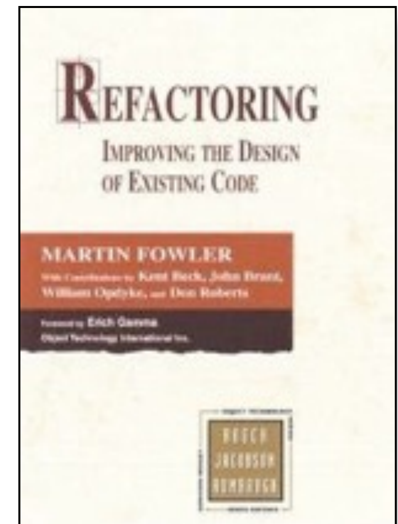
- In Priority Order
  - The code is appropriate for the intended audience
  - The code passes all the tests
  - The code communicates everything it needs to
  - The code has the smallest number of classes
  - The code has the smallest number of methods

Should we then have all code in one class and only have the one method the “main”-method?

Of course not, but why?

Refactoring

- Definition: Improve the code without changing its functionality
- Code needs to be refined as additional requirements (tests) are added
- For more information see *Refactoring: Improve the Design of Existing Code* – Martin Fowler



Working Breadth First - Using a Test List

- Work Task Based
  - 4-8 hour duration (maximum)
- Brainstorm a list of developer tests
- Do not get hung up on completeness… you can always add more later
- Describes completion requirements

COMBITECH

Red/Green/Refactor