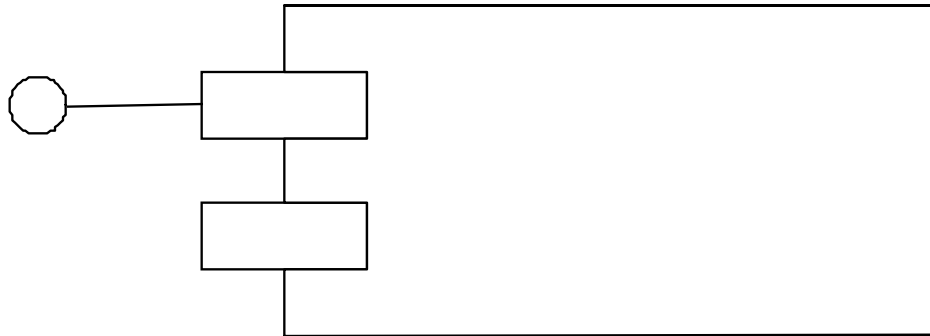


# **VOLVO**

**Introduction to  
Test Driven Development  
and Unit Testing in Eclipse**

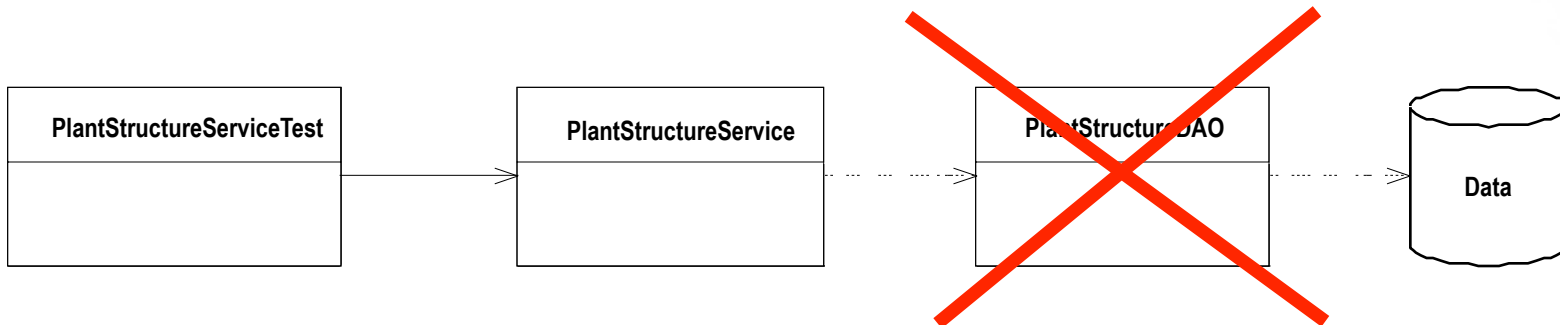
# Unit Tests

- 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

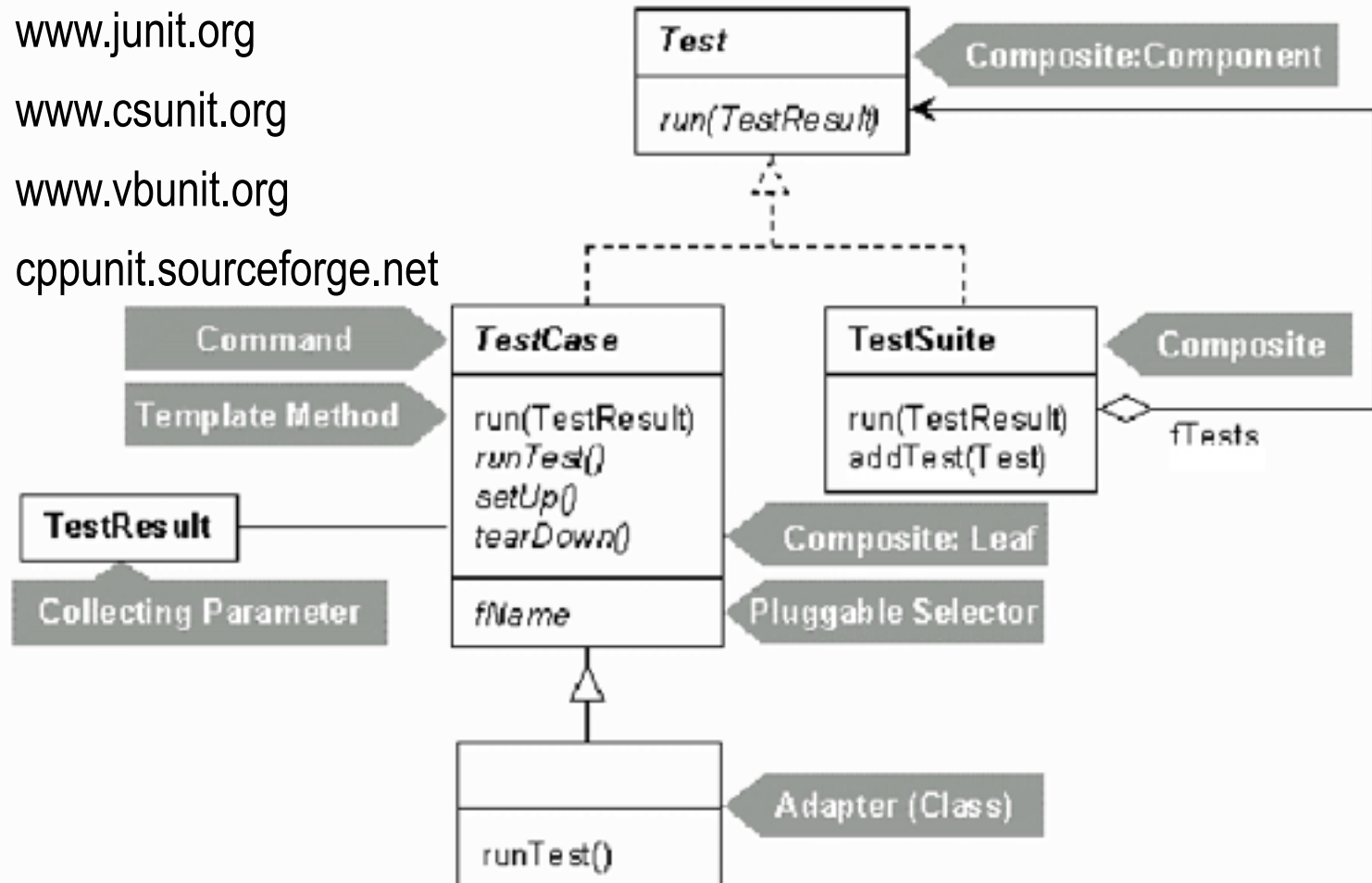
# Requirements on Unit Tests

- 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](http://www.junit.org)
- [www.csunit.org](http://www.csunit.org)
- [www.vbunit.org](http://www.vbunit.org)
- [cppunit.sourceforge.net](http://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.



# 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());
    }

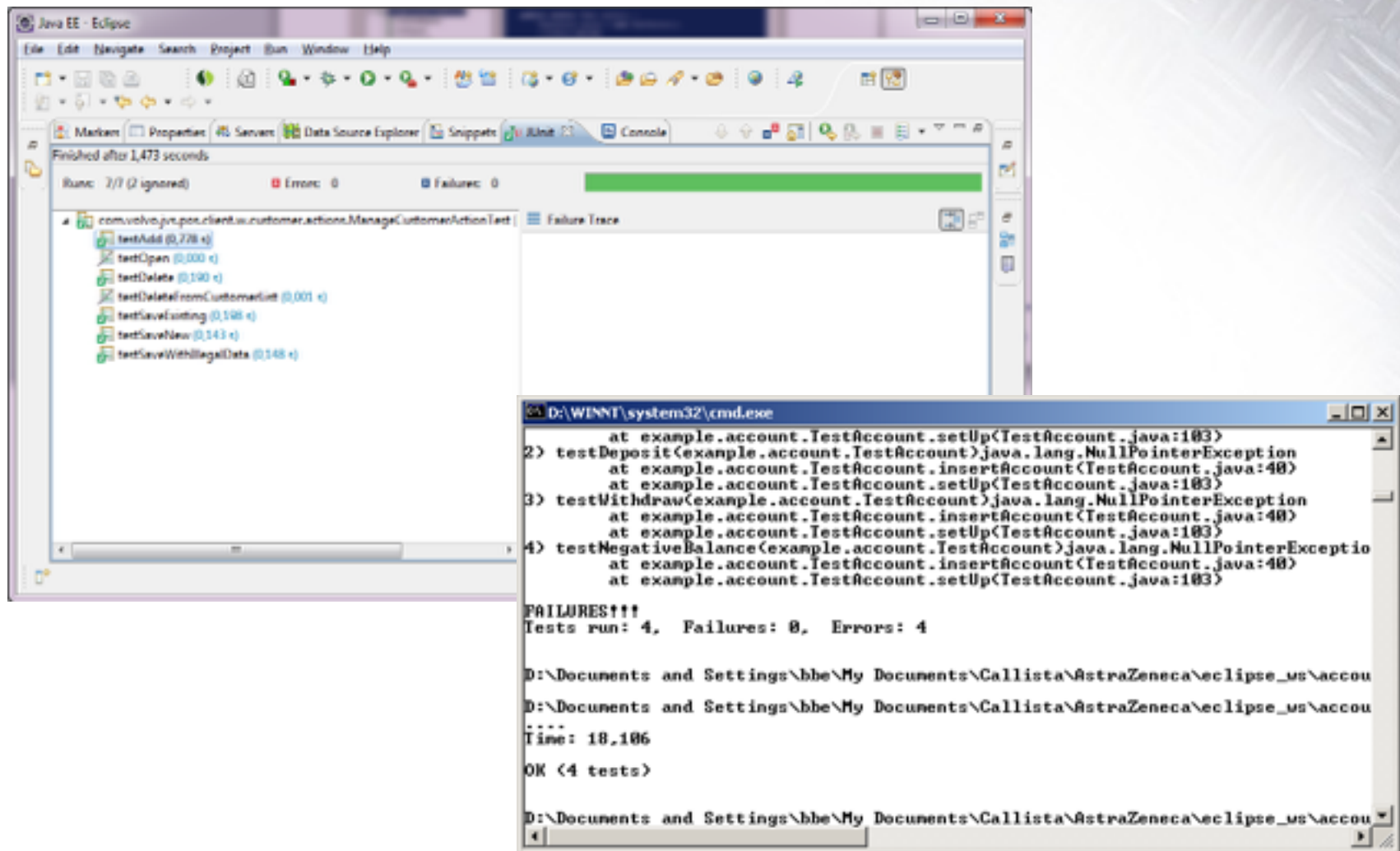
    @Test
    public void testWithdrawTooMuch() throws AccountException { ... }
    ...
}
```

All methods annotated with **@Test** are considered test scenarios

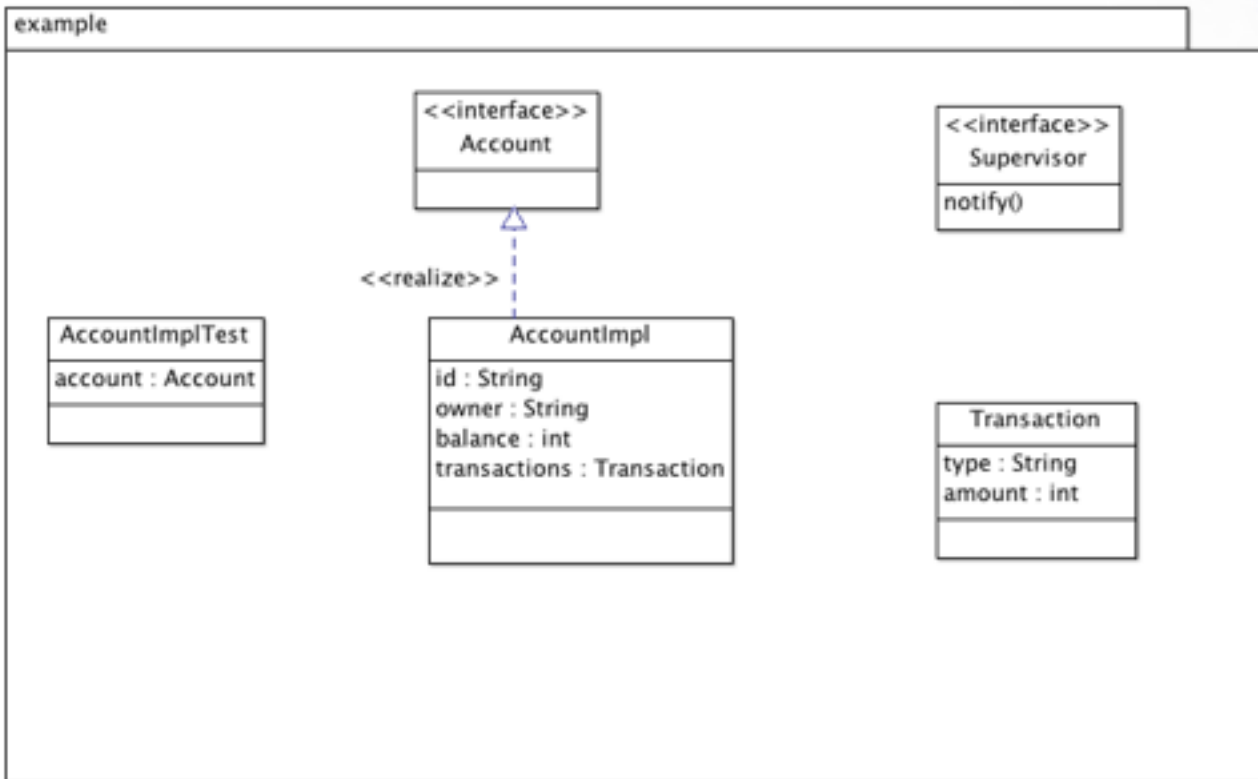
# Assert: Support for verifying conditions

- `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 description



# Exercise 1

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.

- 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() { ... }
```

# Typical unit test scenario

## – The Three A's

1. **Arrange** - Instantiate Unit under Test and set up test data
2. **Act** - Execute one or more methods on the Unit Under Test
3. **Assert** - 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  
}
```

- `Assert.assertTrue("Invariant violated",  
target.IsValid());`

# Ignore a Test

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

```
@Test
```

```
@Ignore("No, not right now but most definitely later")
```

```
    public void testThatDoesntWorkYet() {
```

```
        SomeUnit target = new SomeUnit();
```

```
        target.doSomethingThatDoesntWork();
```

```
        Assert.assertTrue(target.isValid());
```

```
    }
```

# Exercise 2

- Refactor your test data from the last example into a `@Before setUp()` method
- Add tests for the `withdraw()` method.

# Testing private or protected methods/members

In principle you got four options

- Don't test private methods. (Good or Bad?)
- Give the methods package access. (Good or Bad?)
- Use a nested test 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)

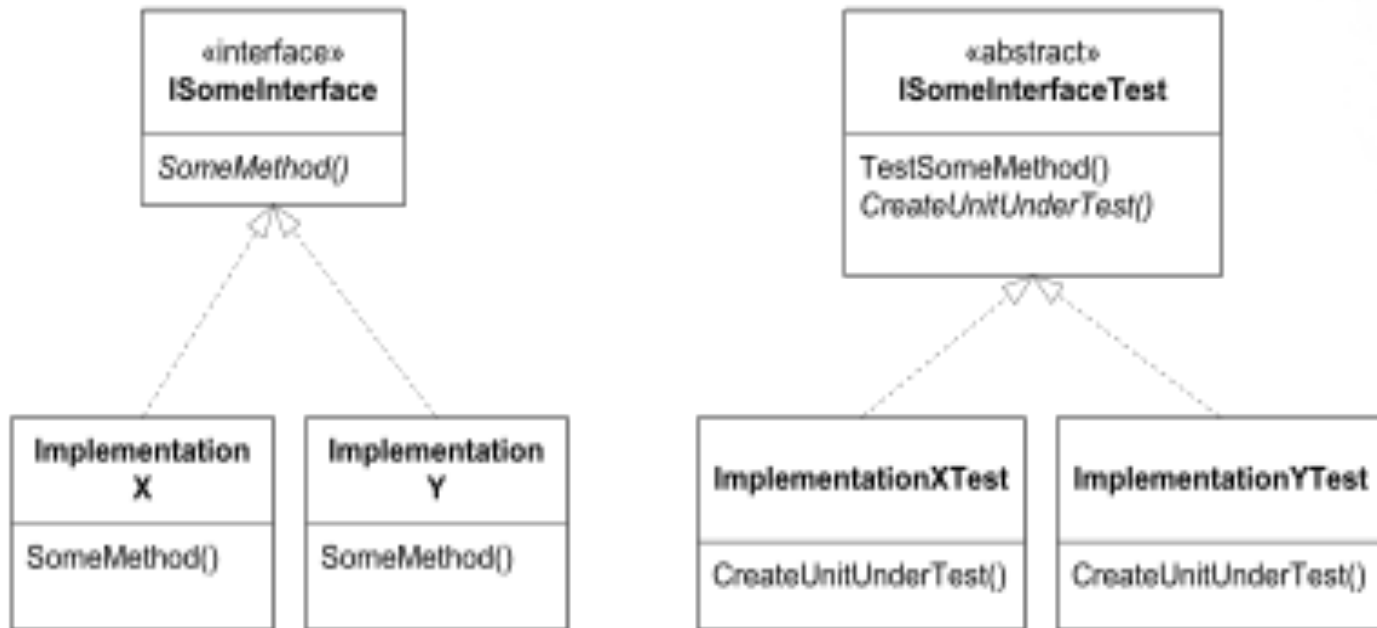
- Protected methods are visible by default when using the same parallel package structure for tests.
- The Subclass and Override idiom is used to write unit tests for protected methods:

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

public class MyClassTest {
    @Test
    public void TestProtectedMethod() {
        String expected = " MyClass: Hello";
        MyClass unitUnderTest = new MyClass() {
            public String myProtectedMethod(String s) {
                return super.myProtectedMethod(s);
            }
        };
        String actual = unitUnderTest.myProtectedMethod("Hello");
        Assert.assertTrue("Strings not equal", actual.equalsIgnoreCase(expected));
    }
}
```

# 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

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

public class ImplementationXTest extends AbstractSomeInterfaceTest {

    @Override
    protected SomeInterface createSomeInterface() {
        return new ImplementationX();
    }
}
```

## 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

# 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

# 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?