Unit Testing in Eclipse with JUnit[1]

© 2004 Michael Olan
Richard Stockton College
Pomona, NJ 08240

Audience: Introductory programming and problem solving (CS-1, CS-2), data structures

Eclipse Version: 3.0

Introduction to Unit Testing

Software testing is an important part of the software development process, but the topic is often neglected in the undergraduate computer science curriculum. Often, the beginner's way of testing code is to execute the program and see if the results seem reasonable. While this may work for very simple programs, it is not effective in projects of any realistic size. So the first design problem to solve is how to develop an effective testing strategy. A naïve approach might be to insert `System.out.println` statements at strategic points in the code to display intermediate results. This technique is not acceptable for a number of reasons. These output statements will need to be removed before releasing the production code, and likely re-inserted when new bugs are found later on. Large amounts of debugging output can result in "scroll blindness", where the programmer has a difficult time interpreting the results.

To be effective, tests must be repeatable to ensure that modifications have corrected the problem and not have not introduced new errors (*regression testing*). To be practical, such testing must be automated.

Unit testing is a particularly useful and natural way to test an object oriented development project. This technique, in conjunction with "Test Driven Development" (Extreme Programming) is an effective way to introduce good software testing practice in various software design courses. JUnit is useful in automating such testing, which encourages programmers to incorporate the technique. Unit testing with JUnit has the dual goals of increasing code quality and writing code faster. These apparently contradictory outcomes are accomplished by spending less time debugging, and enhancing one's confidence to improve existing code and add new features. A JUnit plug-in is built into Eclipse, further enhancing the tool.

A unit test exercises a "unit" of code in isolation and compares actual with expected results. In Java, the unit is usually a class. Unit tests invoke one or more methods from a class to produce observable results that are verified automatically.

Proponents of Extreme Programming (XP) recommend test-driven development as a desirable and effective way to develop software. This technique involves first designing test cases, and then developing class behavior that will satisfy the tests. Class implementation is done incrementally, with tests performed whenever a change is made.

> "If code has no automated test case written for it to prove that it works, it must be assumed not to work. An API that does not have an automated test case to show how it works must be assumed un-maintainable."
>
> *Java Tools for eXtreme Programming*, Hightower & Lesiecki

Guidelines for Software Testing

[1] This project was funded by an Eclipse Innovation Grant, IBM Corporation

- Tests should use the same language as the code being tested
- Test code should be separate from implementation and application code
- Test methods should be independent of the results of other methods
- Testing should be done often, and executing tests should be fast and easy
- Test cases should be logically grouped into test suites
- Tests should be self-checking, and produce immediate feedback that is be easy to read and interpret

JUnit

JUnit is a Java framework for writing and automating unit tests. In Java, a *unit* under test is usually a method. A typical unit test involves verifying that a method accepts input parameters in a certain range and returns the expected value or modifies the state of the object as expected for each input. In other words, does the method uphold the terms of its contract (specifications)?

Terminology

A *test case* is a set of tests that exercises some common behavior, such as for a class or a method. A test case defines a *test fixture*, that provides that resources (data) needed to run the test. A *test suite* is a collection of related test cases. A TestSuite class selects all methods whose names start with *test*.

All public, non-static, parameterless methods whose names begin with *test* are text methods.

The steps taken when JUnit executes a test case are as follows:

1. Execute the setUp method to initialize the test fixture.
2. Execute the textXXX method.
3. Execute the tearDown method.

Insert a diagram

Any assert method that fails will terminate the test method, i.e. there will be a maximum of one failure per test method. In most situations, at most a few assert method calls should be used in any test method.

Core JUnit Assert Methods

`assertTrue/False` – asserts that a condition is true/false
`assertEquals` – asserts that two objects are equal
`assertNull/NotNull` – asserts that an object is/is not null
`assertSame/NotSame` – asserts that two objects refer to the same object
`fail` – fails a test

There are 20 forms for the `assertEquals` method for many different types (primitives mostly). The `assertEquals` method uses == for comparing primitives, and `.equals` for objects. In the case of floating point numbers, an additional parameter representing the error tolerance level is required. See the API specifications for the `Assert` class for details.

Eclipse includes a plug-in that integrates JUnit into the Java IDE, that support creating and running unit tests.

Keep the bar green to keep the code clean. – The JUnit motto

A Simple Example

This first example will illustrate testing a simple class to represent a bank account. The requirements of the class are that instances maintain a `balance` that must be non-negative. The behavior of account objects include `deposit`, `withdraw`, `getBalance` and `equals`. The first two methods must provide an `amount` to be used in the transaction.

Before we begin, let's get an understanding of the behavior of an account object by designing a test plan.

For the `deposit` method (starting with `balance` = 200):

Test Case	Amount	Reason	Expected Result
1	300	normal	500
2	0	boundary	200
3	-300	error	?

For the `withdraw` method (starting with `balance` = 200):

Test Case	Amount	Reason	Expected Result
1	150	normal	50
2	200	boundary	0
3	0	boundary	200
4	-100	error	?
5	300	error	? (overdraft)

We'll build the class and corresponding test cases in Eclipse.

Let's start simple by defining the constructors and the `deposit` method for the `Account` class. Although the deposit method is very easy to write, we won't implement it yet.

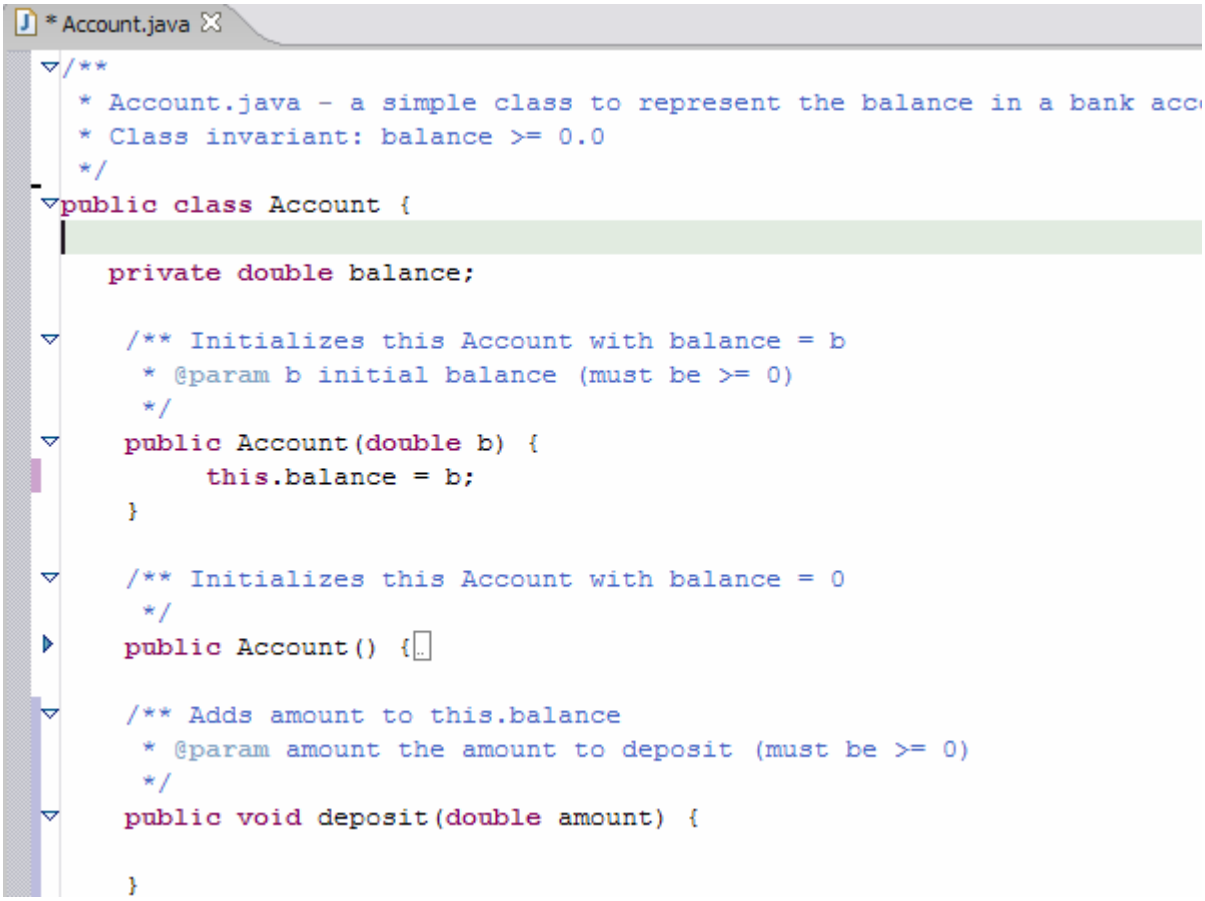

Fig. 1

Writing unit tests for simple constructors like these is usually considered overkill, so lets move on to testing the `deposit` method. This should be simple – construct a test fixture, call the `deposit` method, and determine if the result is as expected. With the `Account` class file selected, create a a test case usingt the *JUnit TestCase Wizard* as shown in Fig 2.

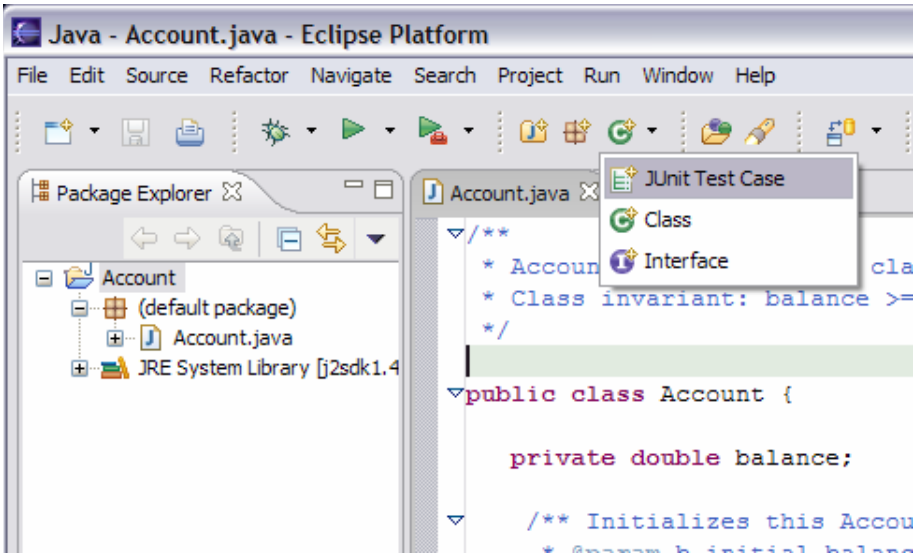

Fig 2

The first time you create a JUnit class, you will be prompted to add the `junit.jar` file to the project's build path. Click **Yes** (see Fig. 3).

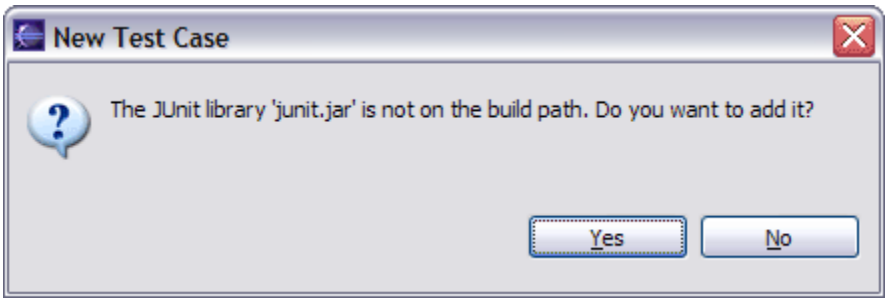

Fig 3

The wizard then gives a template with several fields already filled in. For this test case, the default class name (`AccountTest`) has been changed to `AccountDepositTest` to better reflect the method being tested. We've also checked the box to add a `setUp` method to the test class.

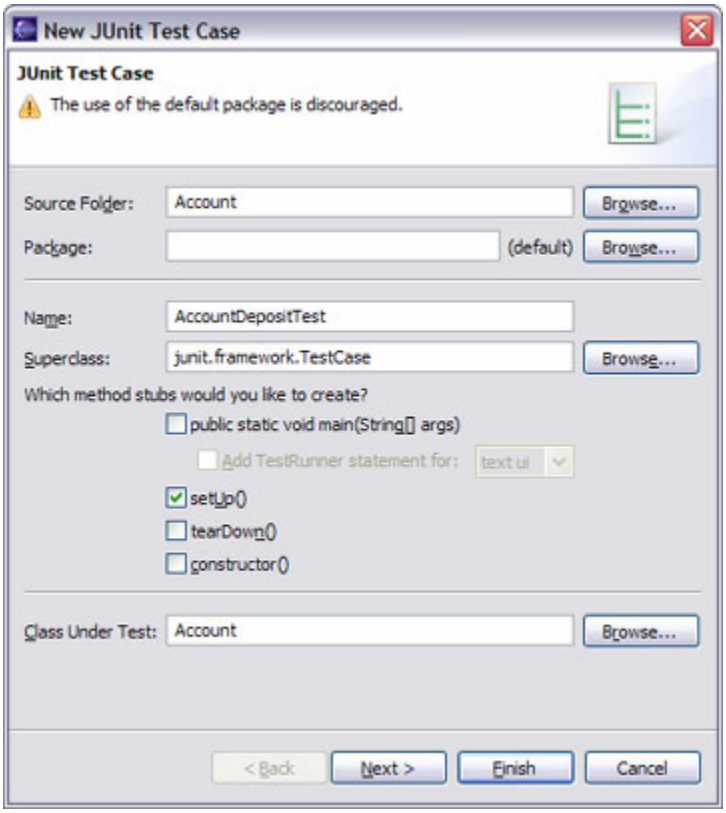

Fig 4

Next select the `deposit` method so that Eclipse will generate a test method stub for it, and click **Finish**.

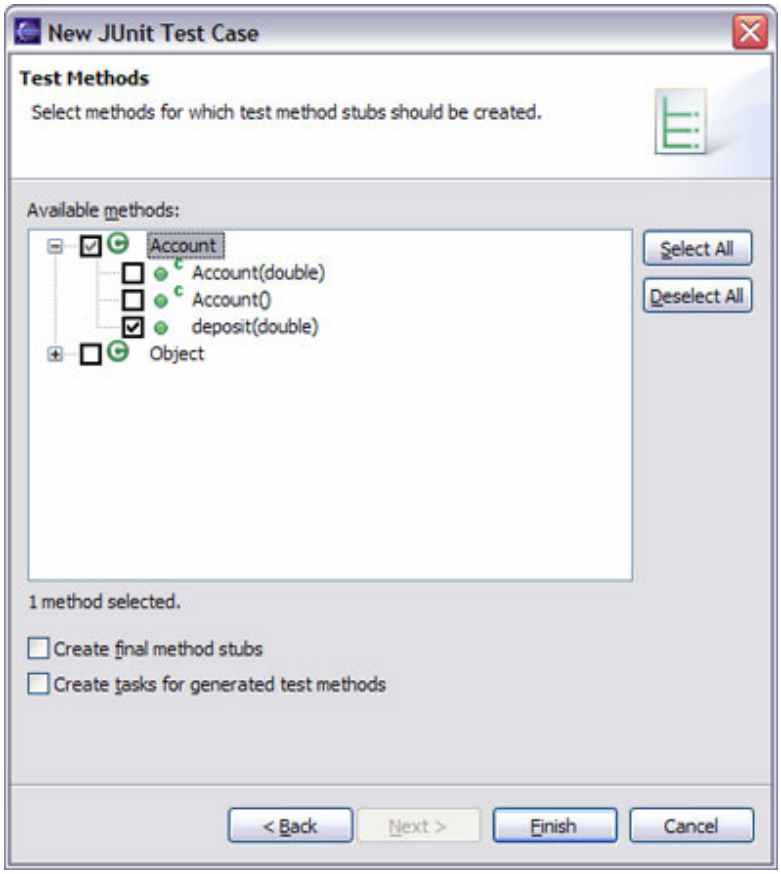

Fig 5

Fig 6 shows the test case skeleton generated by Eclipse.

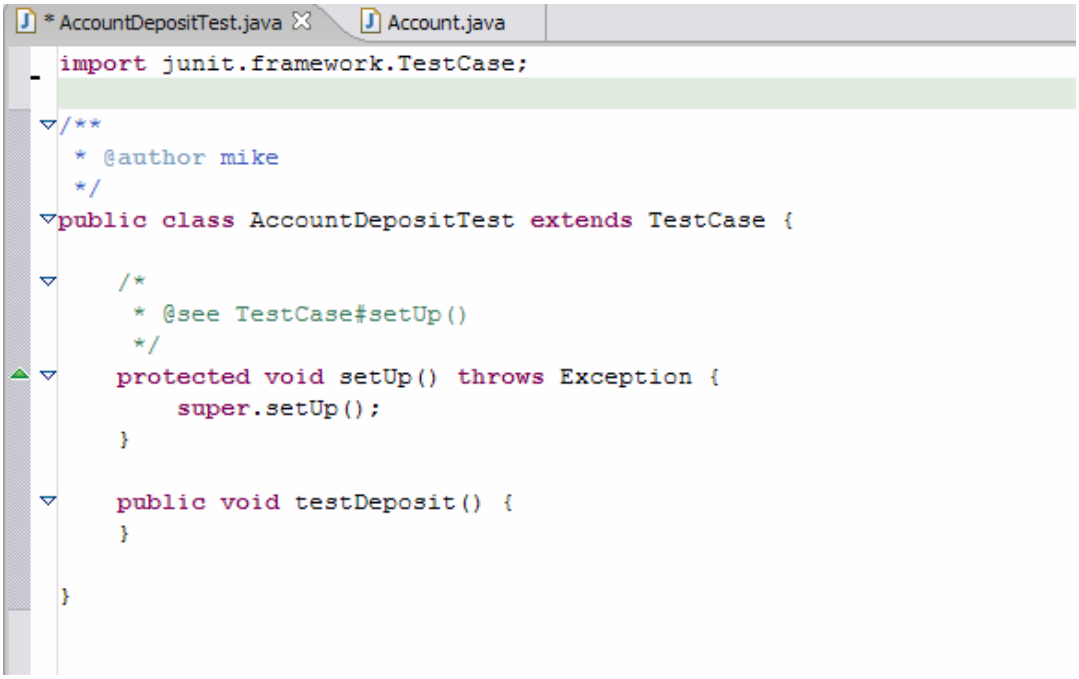

Fig 6

Next, add a test fixture - in this case, an `Account`. The account object is initialized in the `setUp` method.

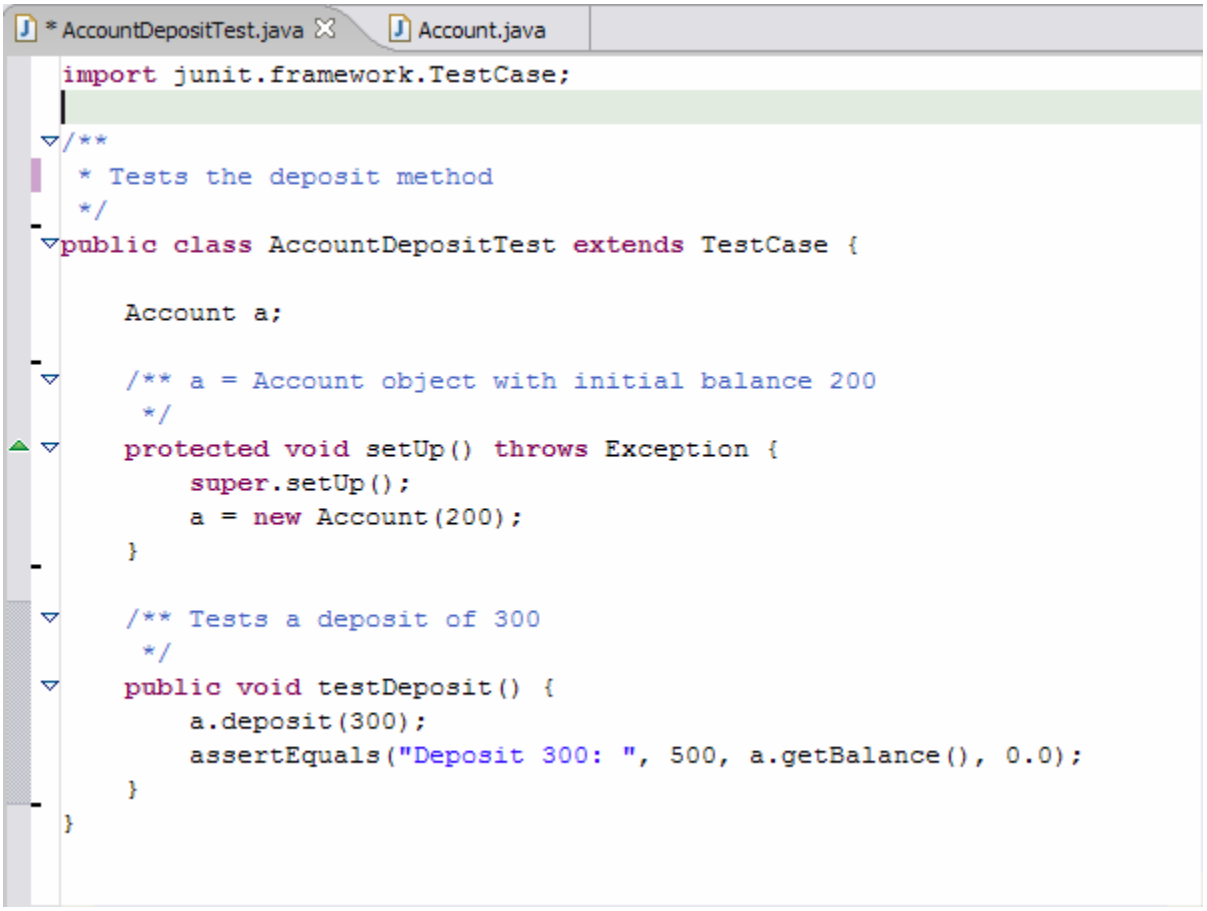

Fig 7

Note the extra parameter in the `assertEquals` method call that represents the error tolerance for comparing two `double` values.

Now run the test case by selecting **Run As -> JUnit Test** (see Fig. 8).

This opens a JUnit view which shows that the test failed (Fig. 9). This is no surprise, since the body of the `deposit` method has not yet been implemented.

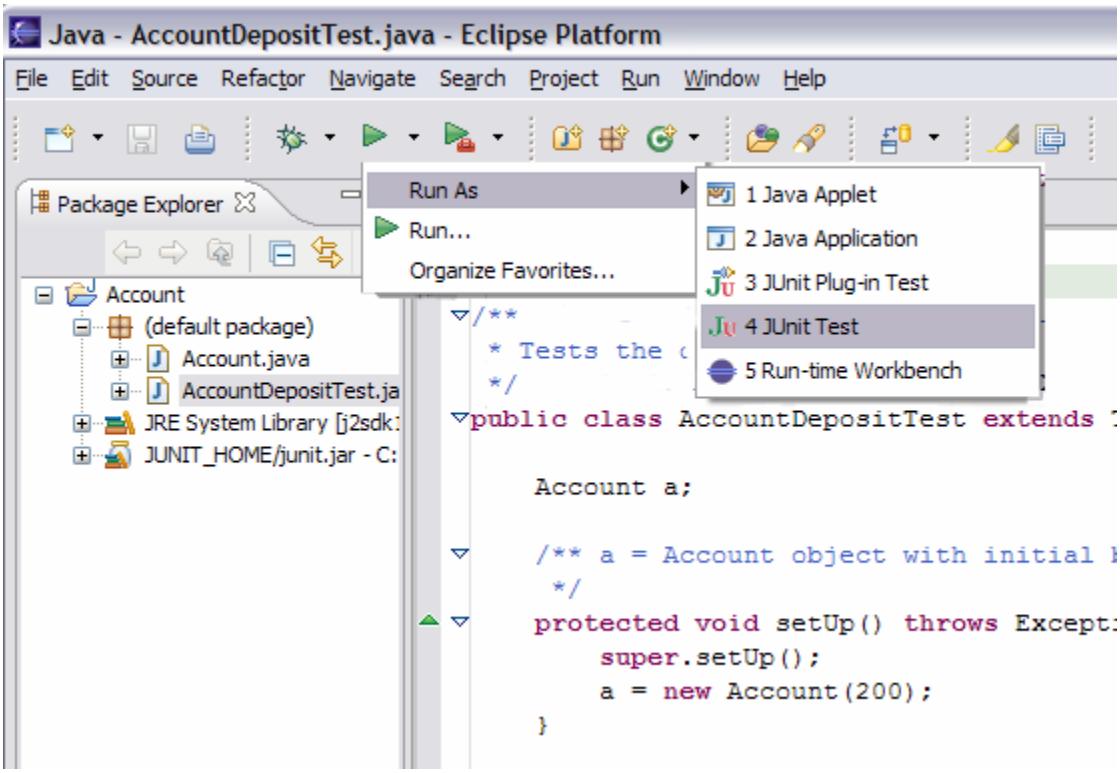

Fig 8

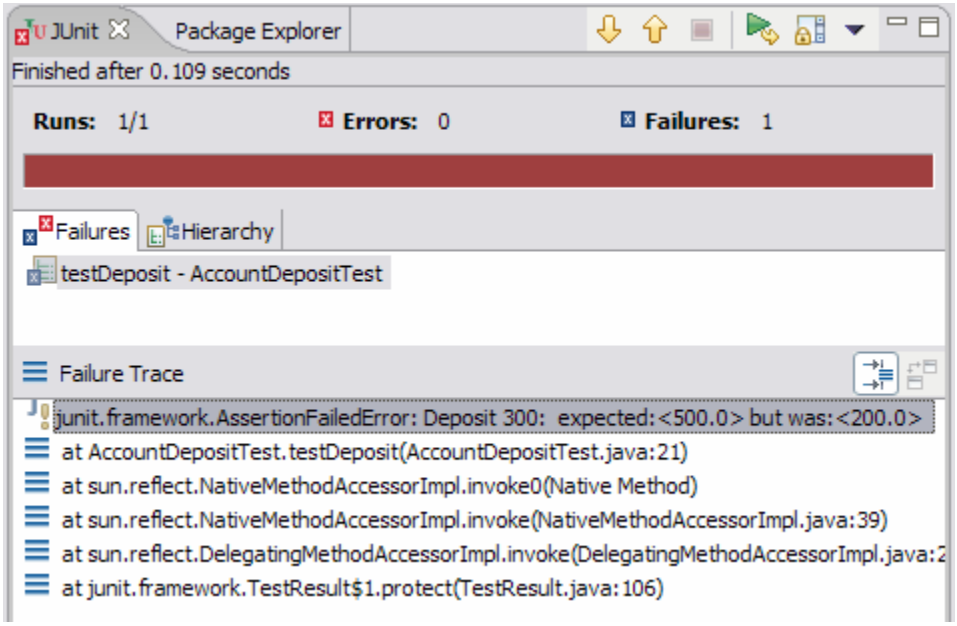

Fig 9

For illustration, let's do the minimal work needed to make this test work – add an assignment statement to `deposit` that stores 500 in `balance`.

```
public void deposit(double amount) {
    this.balance = 500;
}
```

Fig 10

Then run the test again by saving the modified `Account` class and clicking the **Rerun Last Test** button (since JUnit loads the latest class files, the most recently compiled updates are always used). This fixes the problem, at least for the test in question.

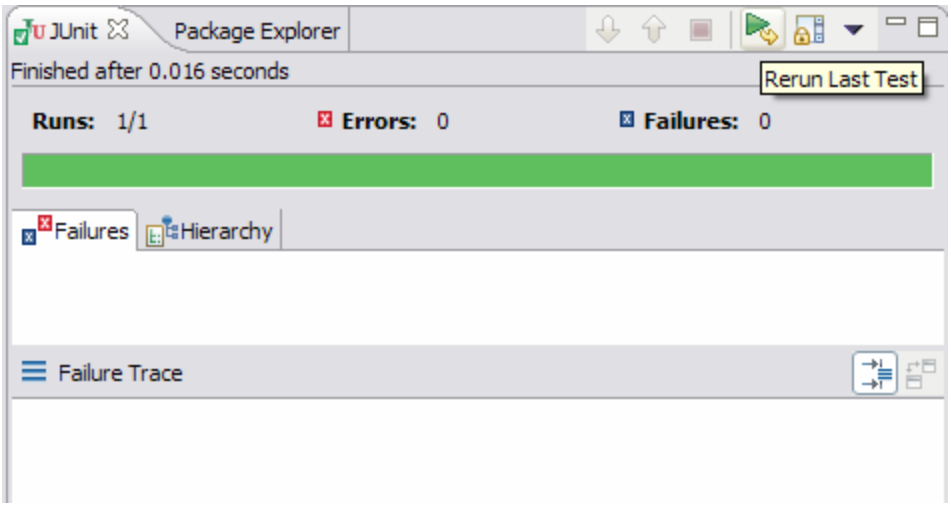

Fig 11

So now let's try another test.

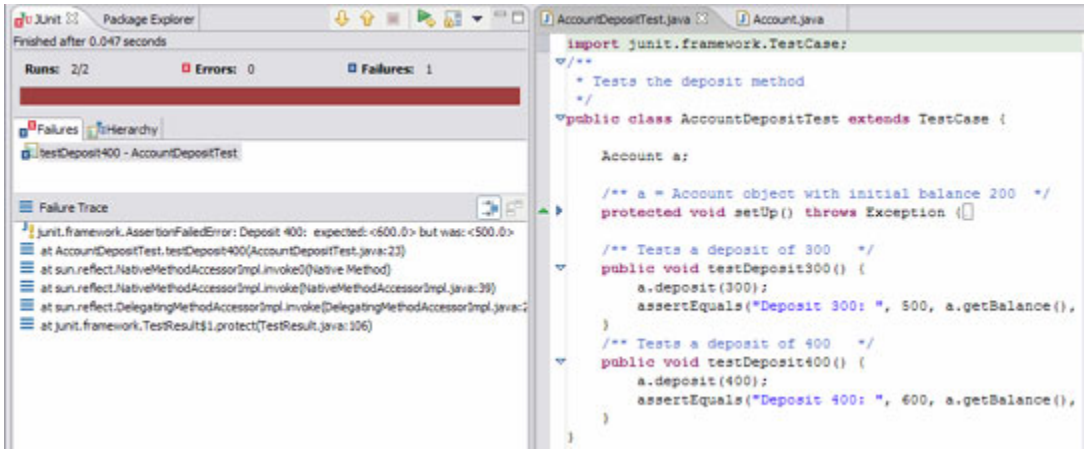

Fig 12

The name of the existing test was changed to `testDeposit300` to better reflect the action being performed. Then copy and paste the method and make appropriate changes to test a deposit of 400. Saving `AccountDepositTest.java` and rerunning the last test reveals that the implementation of `deposit` only works for a deposit that results in a `balance` of 500. So let's fix the `deposit` method to do the right thing.

```
public void deposit(double amount) {
    this.balance = this.balance + amount;
}
```

Fig 13

Save the change and rerun the last test. Success!

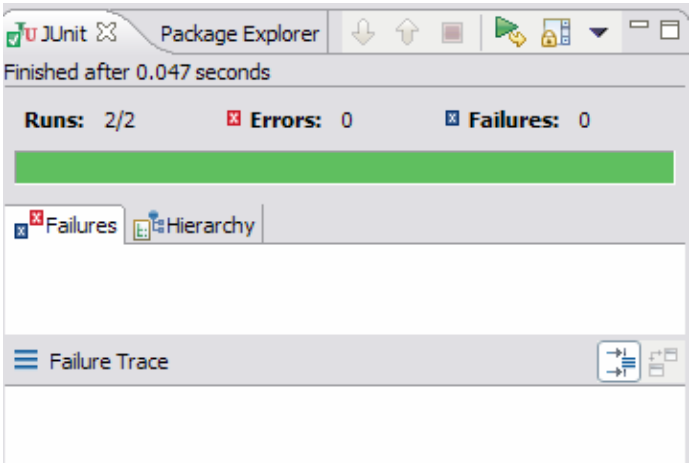

Fig 14

Sidebar : Debugging in Unit Tests

Double clicking on the failure trace line in the JUnit view will open the test case in an editor, and position the cursor in the test method where the failure occurred. But this does not give any information about the actual method where the problem lies. Occasionally, debugging is the best action to take. This is simple to do in Eclipse. First set a breakpoint (double click in the margin on the line where the breakpoint should be set).

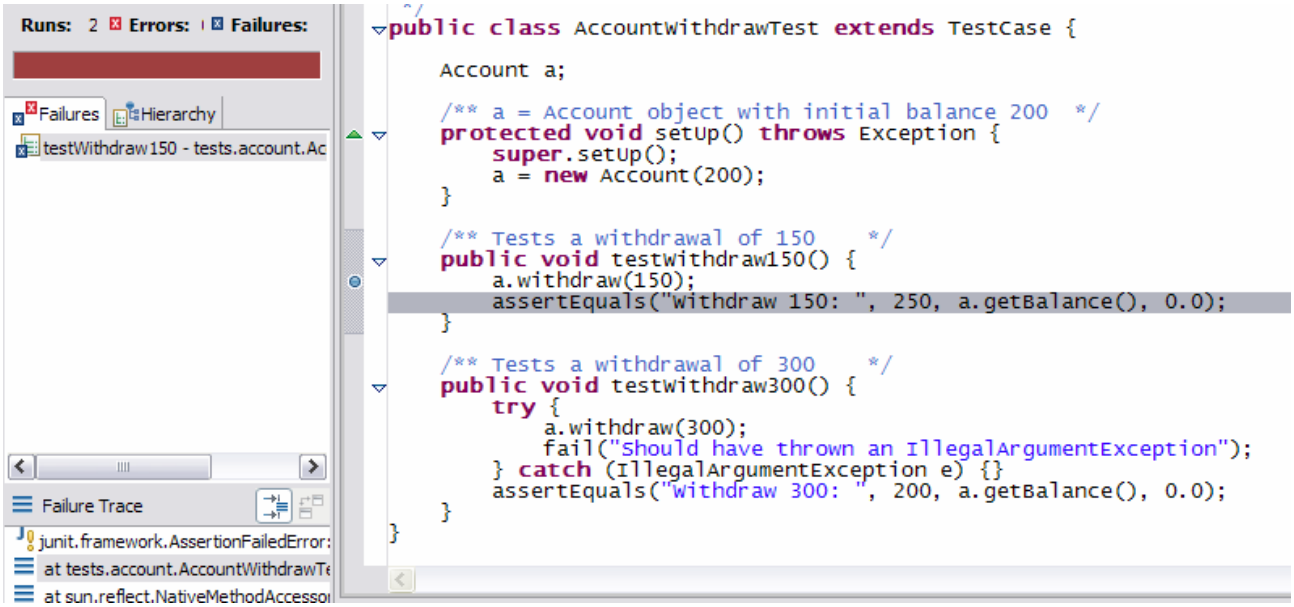

Fig S1

Then from the **Debug** menu, select **Debug As -> JUnit Test**. This will open a Debug perspective, pausing execution at the breakpoint. Now all the usual debugging facilities are available, such as stepping into the method where the problem occurs.

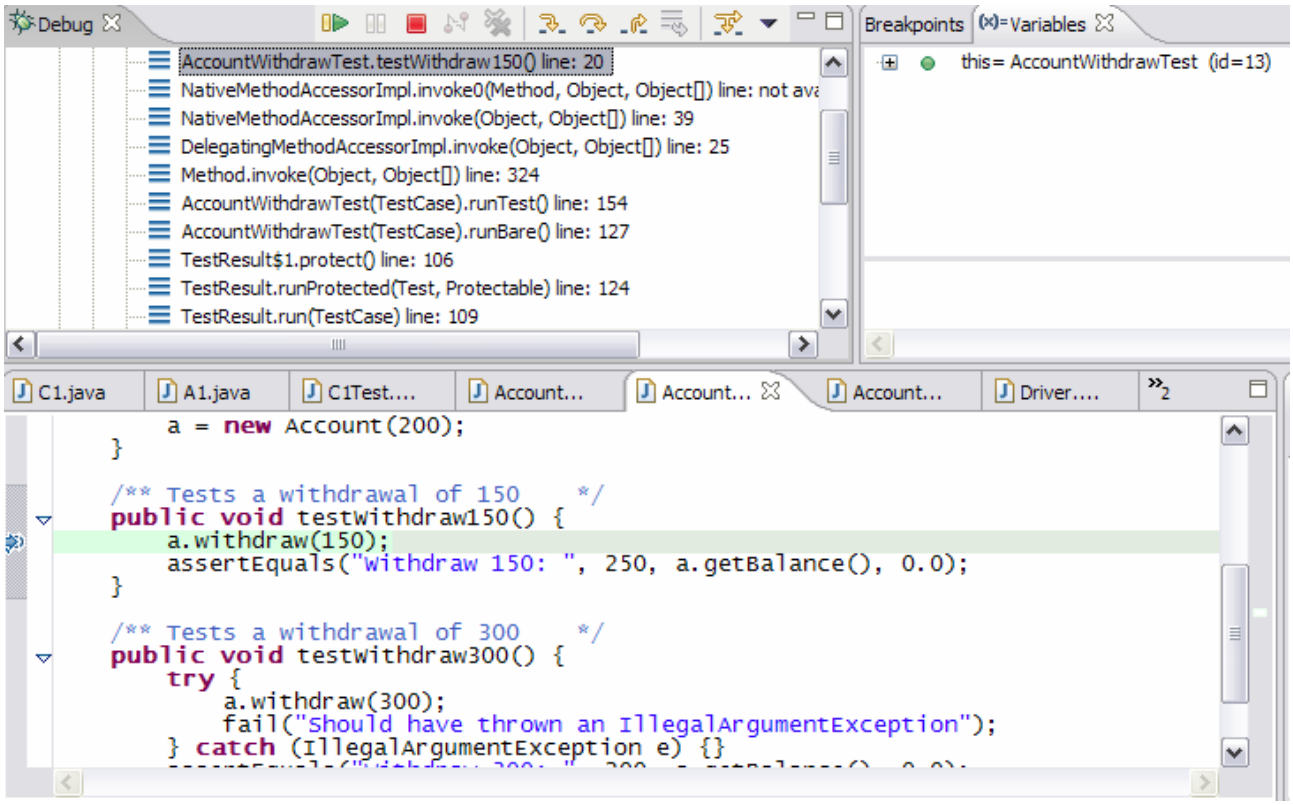

Fig S2

End of Sidebar

At this point, adding a boundary case (deposit 0) will round out the test case for `deposit`. Note that a test with a negative amount should be added to see how the method reacts to invalid data. We'll examine this later.

Now let's test and implement the `withdraw` method. It seems similar to `deposit`, so as a first attempt, we'll add a statement that subtracts `amount` from the `balance`. We'll also add another test case with a test for withdrawing 150 from an Account with a balance of 200. Success!

AccountWithdrawTest.java | AccountDepositTest.java | Account.java

```
import junit.framework.TestCase;

/**
 * Tests the withdraw method
 */
public class AccountWithdrawTest extends TestCase {

    Account a;

    /** a = Account object with initial balance 200  */
    protected void setUp() throws Exception {
        super.setUp();
        a = new Account(200);
    }

    /** Tests a withdrawal of 150    */
    public void testWithdraw150() {
        a.withdraw(150);
        assertEquals("Withdraw 150: ", 50, a.getBalance(), 0.0);
    }
}
```

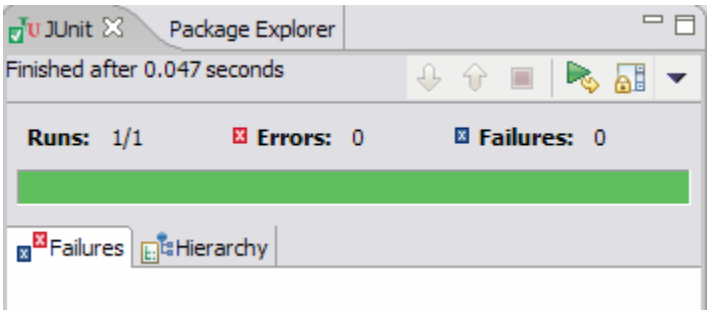

Fig 15

Note that the class invariant requires a non-negative Account balance. The implication for `withdraw` is that `amount` must be ≤ `balance`. We'd better test this by attempting to withdraw more than the balance in the account. The correct action in this case would be to reject the transaction and leave the balance unchanged.

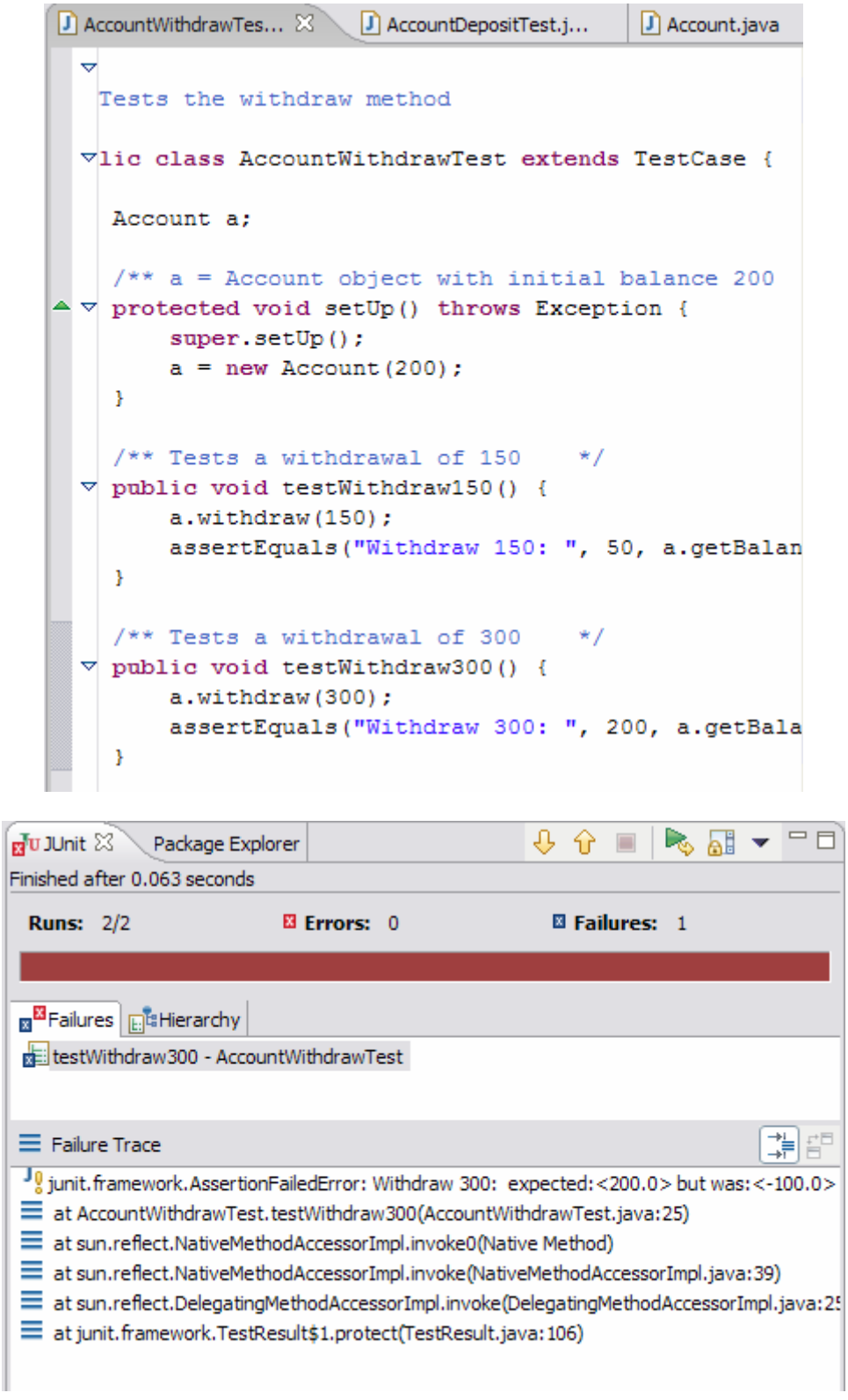

Fig 16

It is no surprise that the test fails.

The preferred fix for this problem is to have the withdraw method throw an exception if the parameter does not meet the method's preconditions. Making the appropriate modifications, and rerunning the test results in an error. This indicates that the withdraw method did throw an exception as shown in the stack trace.

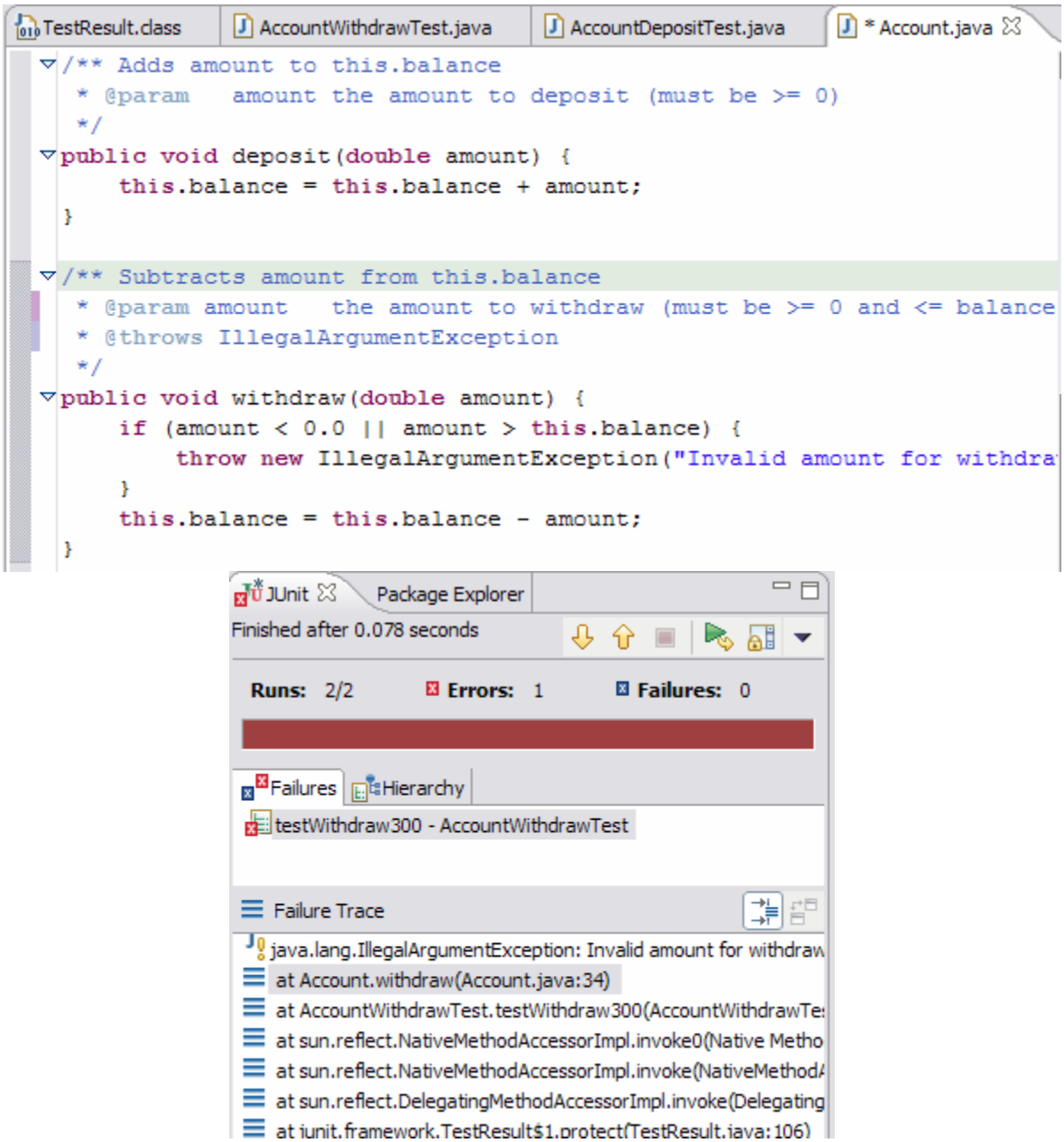

Fig 17

While this error does indicate the test succeeded, we don't want the red bar. Rewriting the test with the call to withdraw in a try-catch block puts the method call in the proper context. The call to fail will occur only if an exception was not thrown when it should have occurred.

AccountWithdrawTest.java | AccountDepositTest.java | Account.java

```
import junit.framework.TestCase;

/**
 * Tests the withdraw method
 */
public class AccountWithdrawTest extends TestCase {

    Account a;

    /** a = Account object with initial balance 200  */
    protected void setUp() throws Exception {

    /** Tests a withdrawal of 150    */
    public void testWithdraw150() {

    /** Tests a withdrawal of 300    */
    public void testWithdraw300() {
        try {
            a.withdraw(300);
            fail("Should have thrown an IllegalArgumentException");
        } catch (IllegalArgumentException e) {}
        assertEquals("Withdraw 300: ", 200, a.getBalance(), 0.0);
    }
```

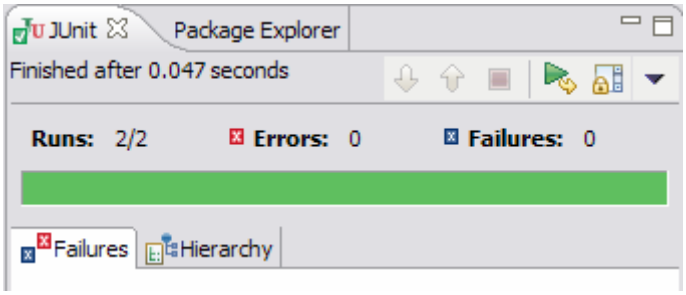

Fig 18

Now adding a similar exception to deposit and a corresponding test will complete the development of these two methods.

Note also that the constructor should also throw an exception if an attempt is made to create an account with a negative balance. The class should be appropriately modified, and a test written to verify that it does in fact work.

Sidebar : Running individual test methods

It is possible to run any test method individually from the context menu in the Outline view. Right click on the method to be run, select **Run -> JUnit Test** and only the selected method will be executed.

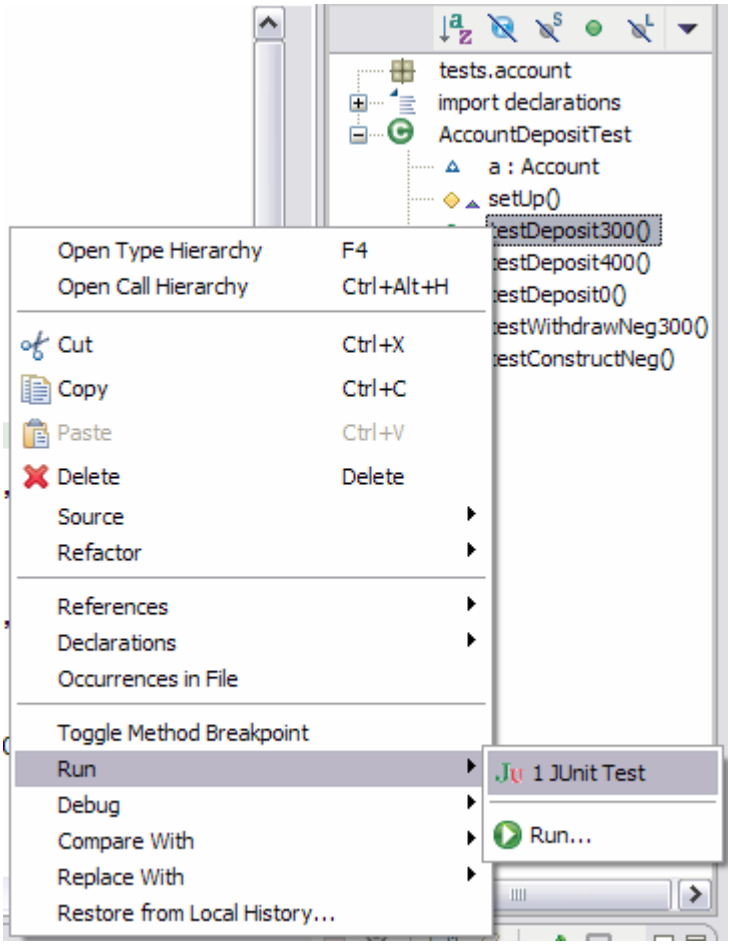

Fig S3

End of Sidebar

Test Suite

A group of related test can be aggregated into a test suite and run together. This is done in Eclipse by using the **New -> Other** wizard and selecting **JUnit Test Suite** (Fig. 19).

The wizard will use the default name **AllTests** for the class and select all classes in the project with names ending in *Test*.

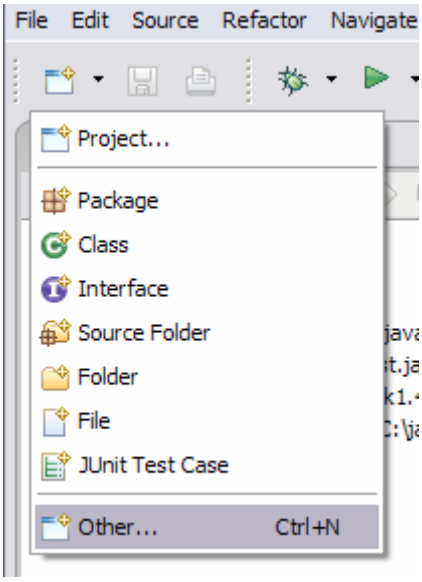
File Edit Source Refactor Navigate
Project...
Package
Class
Interface
Source Folder
Folder
File
JUnit Test Case
Other... Ctrl+N

Fig 19

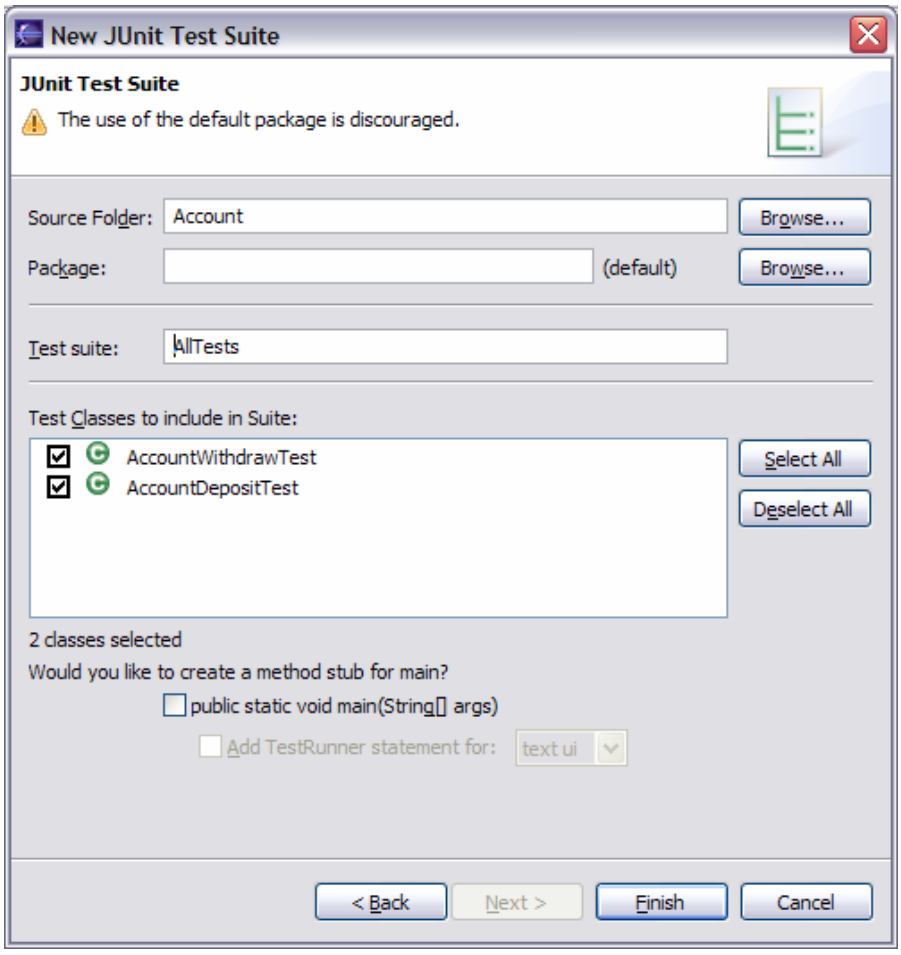
New JUnit Test Suite
JUnit Test Suite
The use of the default package is discouraged.
Source Folder: Account
Browse...
Package:
(default)
Browse...
Test suite: AllTests
Test Classes to include in Suite:
AccountWithdrawTest
AccountDepositTest
Select All
Deselect All
2 classes selected
Would you like to create a method stub for main?
public static void main(String[] args)
Add TestRunner statement for: text ui
< Back
Next >
Finish
Cancel

Fig 20

Now suppose we extend the `Account` class so that an attempt to withdraw an amount greater than the current balance will result in an overdraft fee being charged against the account.

```
/**
 * OverdraftAccount.java - Bank accounts that charges a fee for attempts
 * withdraw more than the current balance
 */
public class OverdraftAccount extends Account {

    private static final double OVERDRAFT_FEE = 20.0;
    /**..
    public OverdraftAccount() {
        super();
    }
    /**..
    public OverdraftAccount(double b) {
        super(b);
    }

    /** Subtracts amount from this.balance
     * (or subtracts OVERDRAFT_FEE from this.balance if amount > this.balance)
     * @param amount   the amount to withdraw (must be >= 0 and <= balance)
     * @throws IllegalArgumentException
     */
    public void withdraw(double amount) {
        try {
            super.withdraw(amount);
        } catch (IllegalArgumentException e) {
            super.withdraw(OVERDRAFT_FEE);
            throw e;
        }
    }
}
```

Fig 21

The following test case verifies that the withdraw method works properly.

```
import junit.framework.TestCase;
/**
 * Test case for withdraw from OverdraftAccount
 */
public class OverdraftAccountTest extends TestCase {

    Account a;
    /** Initialize a with balance 200    */
    protected void setUp() throws Exception {
        super.setUp();
        a = new OverdraftAccount(200);
    }

    /** Test: Withdraw 150   */
    public void testWithdrawOK() {
        try {
            a.withdraw(150);
            assertEquals("withdraw 150: ", 50, a.getBalance(), 0.0);
        } catch (IllegalArgumentException e) {}
    }

    /** Error test: Attempt to withdraw 250  */
    public void testWithdrawOver() {
        try {
            a.withdraw(250);
            assertEquals("withdraw 250: ", 180, a.getBalance(), 0.0);
            fail("Should have thrown exception");
        } catch (IllegalArgumentException e) {}
    }
}
```

Fig 22

Repeating the **New** wizard to create an **AllTests** test suite will update the test suite to include the `OverdraftAccountTest` case. Running the updated test suite verifies that the new implementation has not broken any of the existing code.

Another test not included here is one to examine the result of attempting to withdraw an excess amount from an account with balance less than `OVERDRAFT_FEE`. This would result in a negative balance, but this would violate the class invariant. An appropriate solution for this case is beyond the scope of this discussion, but it does point out how unit testing can help to detect such situations early in the development cycle.

Organizing Files

Larger projects can benefit from the organization provided by using packages. In particular, it is sometimes recommended that test files be stored in a separate package from the source code to make deployment of the project easier. It is easier to exclude test code from JAR files when they are stored in separate directories. The following shows a possible configuration that separates testing code from implementation code.

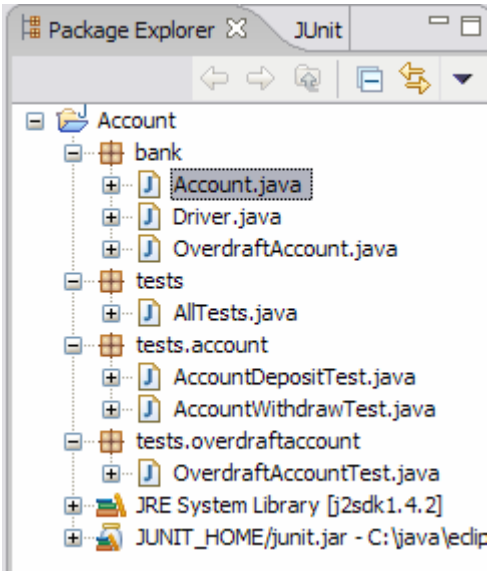

Fig 23

The next figure shows the hierarchy tab of the JUnit view for this package structure.

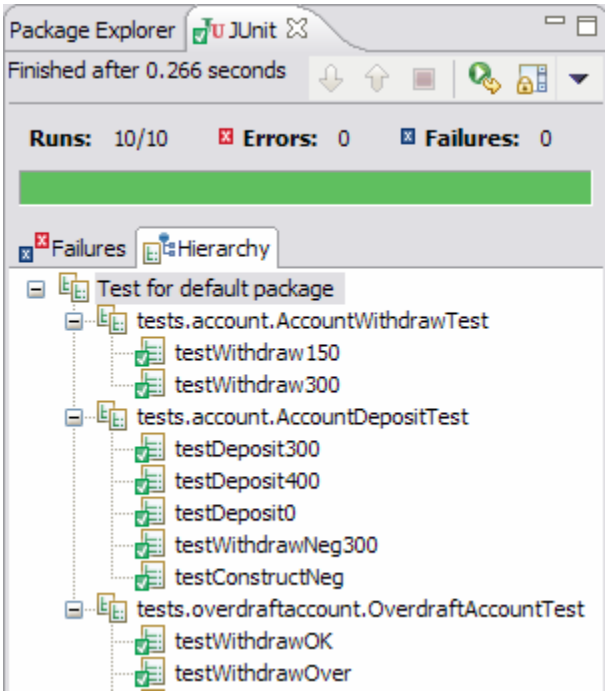

Fig 24

Summary

Follow the Extreme Programming technique of *code a little, test a little*. Some suggestions are to run tests after every 10 minutes of coding.

Use and develop specifications and tests together, but independent of any application code. This will improve the reusability of the module under development, and encourage refactoring to improve overall quality.

Exercises

Design a test plan and implement unit test cases for the Queue class (array implementation w/wraparound) Find and correct any errors detected in the implementation.

Sidebar : Differences between Eclipse 2.1.x and 3.0.

The Eclipse 2.1.x JUnit Test class wizard does not have a prompt for adding `junit.jar` to the project class path. Before writing JUnit tests, it is necessary to open the project's **Property** page, and in the **Java Build Path** select the **Libraries** tab, and add junit.jar (found in the junit.org plug-in directory).

End of Sidebar

References:

Beck, Kent *Test-Driven Development By Example*, Addison-Wesley, 2003.

Daum, Berthold *Eclipse 2 for Java Developers*, John Wiley & Sons, 2003.

Fowler, Martin *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.

Gallardo, David, Burnette, Ed, and McGovern, Robert, Hightower, Richard and Lesiecki, Nicholas *Java Tools for eXtreme Programming: Mastering Open Source Tools, Including Ant, JUnit, and Cactus*, John Wiley & Sons, 2002.

JUnit Web Site, www.junit.org

Link, Johannes *Unit Testing in Java*, Morgan Kaufman Publishers, 2002.

Massol, Vincent *JUnit in Action*, Manning Publications, 2004.

Olan, Michael " Unit Testing: Test Early, Test Often", *Journal of Computing Sciences in Colleges*, October 2003.

Prohorenko, Alexander and Prohorenko, Olexiy "Using JUnit With Eclipse IDE", O'Reilly on Java.com, http://www.onjava.com/lpt/a/4524, February 2004.

"Writing and running JUnit tests" in Eclipse Help.