JUnit

Test suites

- Obviously you have to test your code to get it working in the first place
  - You can do *ad hoc* testing (testing whatever occurs to you at the moment), or
  - You can build a test suite (a thorough set of tests that can be run at any time)
- Disadvantages of writing a test suite
  - It’s a lot of extra programming
    - *True*—but use of a good test framework can help quite a bit
  - You don’t have time to do all that extra work
    - *False*—Experiments repeatedly show that test suites reduce debugging time more than the amount spent building the test suite
- Advantages of having a test suite
  - Your program will have many fewer bugs
  - It will be a ***lot*** easier to maintain and modify your program
    - This is a *huge* win for programs that, unlike class assignments, get actual use!

Example: Old way vs. new way

-
  ```
  int max(int a, int b) {
      if (a > b) {
          return a;
      } else {
          return b;
      }
  }
  ```

-
  ```
  void testMax() {
      int x = max(3, 7);
      if (x != 7) {
          System.out.println("max(3, 7) gives " + x);
      }
      x = max(3, -7);
      if (x != 3) {
          System.out.println("max(3, -7) gives " + x);
      }
  }
  ```

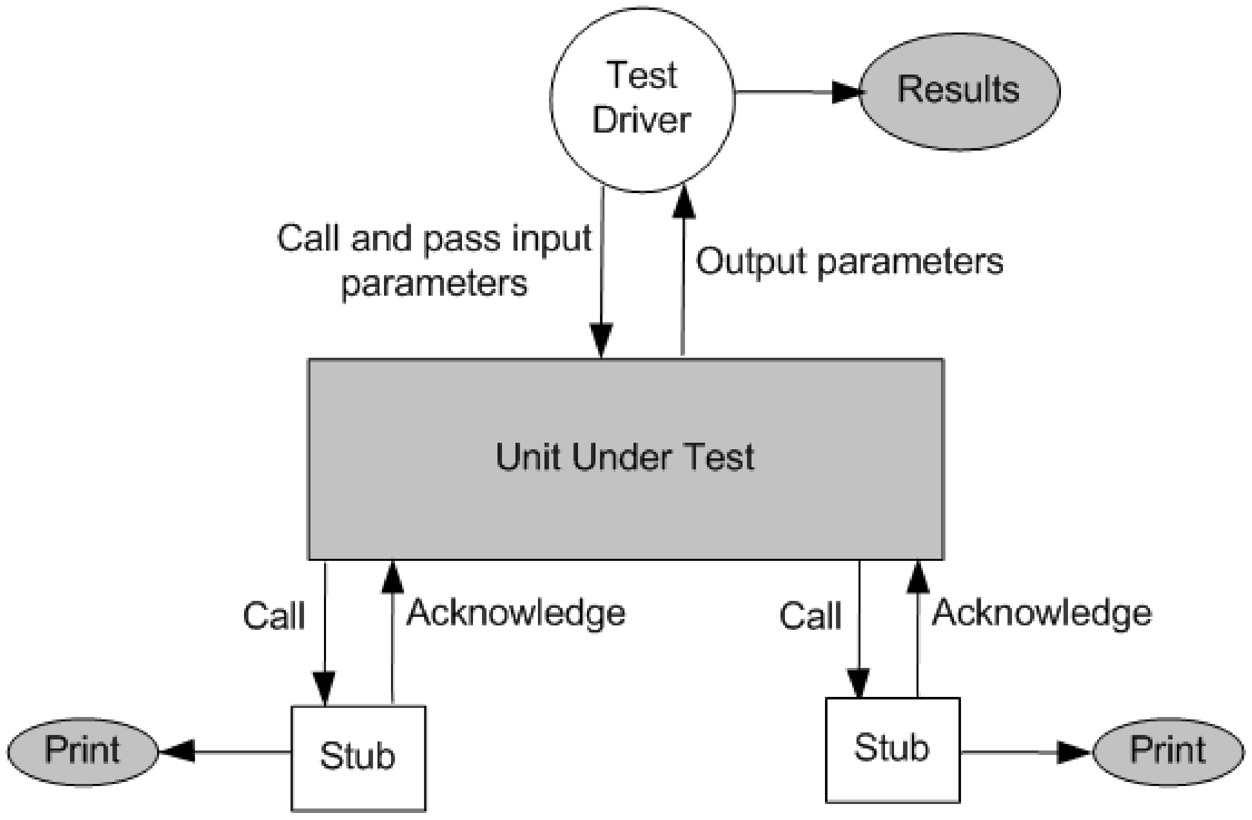
-
  ```
  public static void main(String[] args) {
      new MyClass().testMax();
  }
  ```

-
  ```
  @Test
  void testMax() {
      assertEquals(7, max(3, 7));
      assertEquals(3, max(3, -7));
  }
  ```

XP approach to testing

- In the Extreme Programming approach,
  - Tests are written before the code itself
  - If code has no automated test case, it is *assumed not to work*
  - A test framework is used so that automated testing can be done after every small change to the code
    - This may be as often as every 5 or 10 minutes
  - If a bug is found after development, a test is created to keep the bug from coming back
- Consequences
  - Fewer bugs
  - More maintainable code
  - Continuous integration—During development, the program *always works*—it may not do everything required, but what it does, it does right

Unit Testing



Selection of test data is broadly based on the following techniques:

- Control flow testing
  - Draw a control flow graph (CFG) from a program unit
  - Select a few control flow testing criteria
  - Identify a path in the CFG to satisfy the selection criteria
  - Derive the path predicate expression from the selection paths
  - By solving the path predicate expression for a path, one can generate the data
- Data flow testing
  - Draw a data flow graph (DFG) from a program unit and then follow the procedure described in control flow testing.
- Domain testing
  - Domain errors are defined and then test data are selected to catch those faults
- Functional program testing
  - Input/output domains are defined to compute the input values that will cause the unit to produce expected output values

JUnit

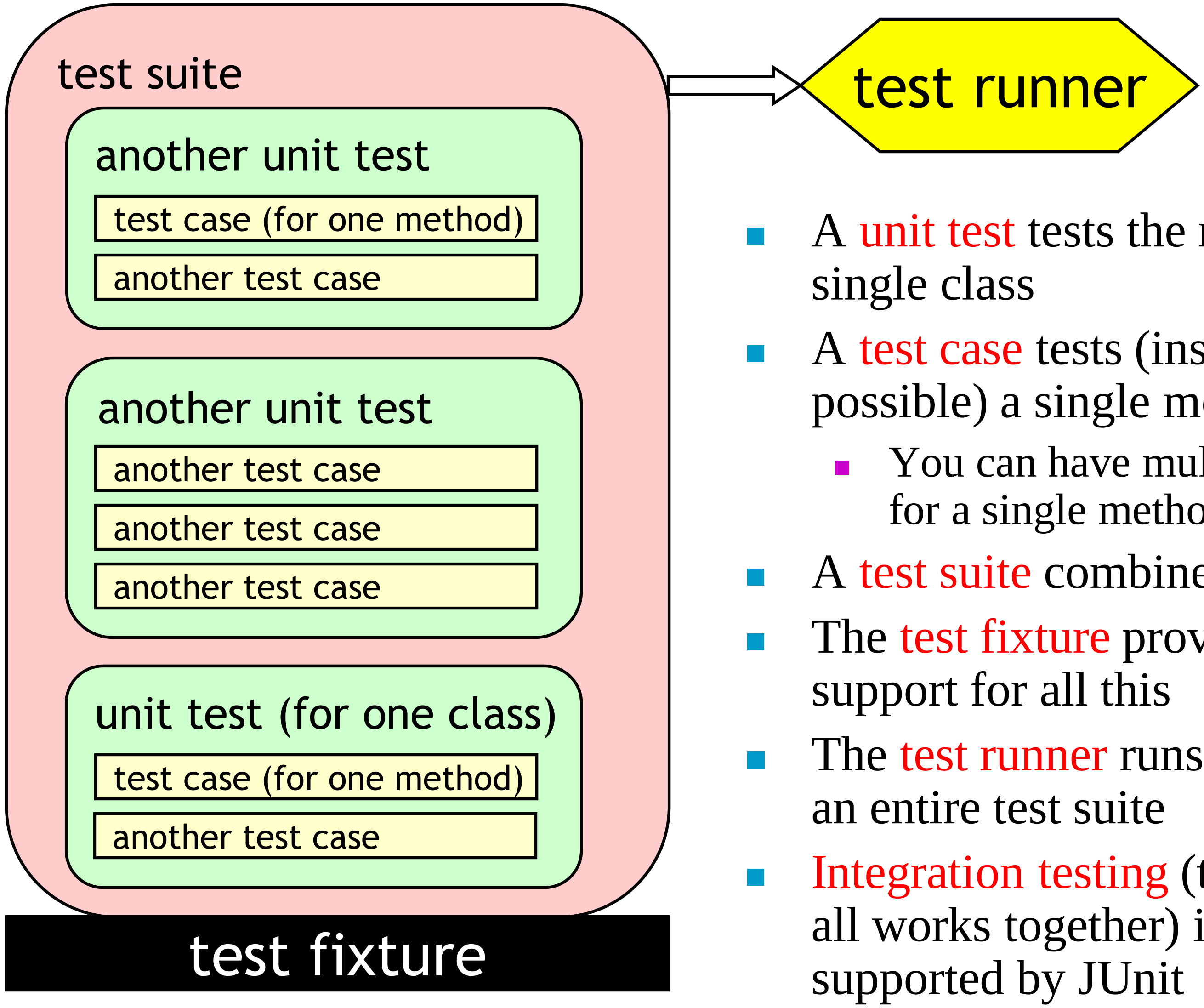
- JUnit is a framework for writing tests
  - JUnit was written by Erich Gamma (of *Design Patterns* fame) and Kent Beck (creator of XP methodology)
  - JUnit uses Java’s reflection capabilities (Java programs can examine their own code)
  - JUnit helps the programmer:
    - define and execute tests and test suites
    - formalize requirements and clarify architecture
    - write and debug code
    - integrate code and always be ready to release a working version
  - JUnit is not included in Sun’s SDK, but almost all IDEs include it

Terminology

- A test fixture sets up the data (both objects and primitives) that are needed to run tests
  - Example: If you are testing code that updates an employee record, you need an employee record to test it on
- A unit test is a test of a *single* class
- A test case tests the response of a single method to a particular set of inputs
- A test suite is a collection of test cases
- A test runner is software that runs tests and reports results

- An integration test is a test of how well classes work together
  - JUnit provides some limited support for integration tests

Once more, in pictures



- A unit test tests the methods in a single class
- A test case tests (insofar as possible) a single method
  - You can have multiple test cases for a single method
- A test suite combines unit tests
- The test fixture provides software support for all this
- The test runner runs unit tests or an entire test suite
- Integration testing (testing that it all works together) is not well supported by JUnit

JUnit

- JUnit is a framework for writing unit tests
  - A unit test is a test of a *single* class
    - A test case is a single test of a single method
    - A test suite is a collection of test cases
- Unit testing is particularly important when software requirements change frequently
  - Code often has to be refactored to incorporate the changes
  - Unit testing helps ensure that the refactored code continues to work

JUnit..

- JUnit helps the programmer:
  - Define and execute tests and test suites
  - Formalize requirements and clarify architecture
  - Write and debug code
  - Integrate code and always be ready to release a working version

What JUnit does

- JUnit runs a suite of tests and reports results
- For *each* test in the test suite:
  - JUnit calls `setUp()`
    - This method should create any objects you may need for testing

What JUnit does…

- JUnit calls *one* test method
  - The test method may comprise multiple test cases; that is, it may make multiple calls to the method you are testing
  - In fact, since it’s your code, the test method can do anything you want
  - The `setUp()` method ensures you *entered* the test method with a virgin set of objects; what you do with them is up to you
- JUnit calls `tearDown()`
  - This method should remove any objects you created

Creating a test class in JUnit

- Define a subclass of TestCase
- Override the setUp() method to initialize object(s) under test.
- Override the tearDown() method to release object(s) under test.
- Define one or more public testXXX() methods that exercise the object(s) under test and assert expected results.
- Define a static suite() factory method that creates a TestSuite containing all the testXXX() methods of the TestCase.
- Optionally define a main() method that runs the TestCase in batch mode.

Fixtures

- A fixture is just a some code you want run before every test
- You get a fixture by overriding the method
  - protected void setUp() { …}
- The general rule for running a test is:
  - protected void runTest() {
    setUp(); <run the test> tearDown();
    }
  - so we can override setUp and/or tearDown, and that code will be run prior to or after every test case

Implementing setUp() method

- Override setUp() to initialize the variables, and objects
- Since setUp() is your code, you can modify it any way you like (such as creating new objects in it)
- Reduces the duplication of code

Implementing the tearDown() method

- In most cases, the tearDown() method doesn't need to do anything
  - The next time you run setUp(), your objects will be replaced, and the old objects will be available for garbage collection
  - Like the finally clause in a try-catch-finally statement, tearDown() is where you would release system resources (such as streams)

The structure of a test method

- A test method doesn’t return a result
- If the tests run correctly, a test method does nothing
- If a test fails, it throws an AssertionFailedError
- The JUnit framework catches the error and deals with it; you don’t have to do anything

Test suites

- In practice, you want to run a group of related tests (e.g. all the tests for a class)
- To do so, group your test methods in a class which extends TestCase
- Running suites we will see in examples

Writing a JUnit test class, I

- Start by importing these JUnit 4 classes:

```
import org.junit.*;
import static org.junit.Assert.*; // note static import
```

- Declare your test class in the usual way

```
public class MyProgramTest {
```

- Declare an instance of the class being tested
- You can declare other variables, but *don't* give them initial values here

```
public class MyProgramTest {
    MyProgram program;
    int someVariable;
```

Writing a JUnit test class, II2

- Define a method (or several methods) to be executed *before each test*
- Initialize your variables in this method, so that each test starts with a fresh set of values

```
@Before
public void setUp() {
    program = new MyProgram();
    someVariable = 1000;
}
```

- You can define one or more methods to be executed after each test
- Typically such methods release resources, such as files
- Usually there is no need to bother with this method

```
@After
public void tearDown() {
}
```

A simple example

- Suppose you have a class Arithmetic with methods int multiply(int x, int y), and boolean isPositive(int x)

-
```
import org.junit.*;
import static org.junit.Assert.*;
```

-
```
public class ArithmeticTest {

    @Test
    public void testMultiply() {
        assertEquals(4, Arithmetic.multiply(2, 2));
        assertEquals(-15, Arithmetic.multiply(3, -5));
    }

    @Test
    public void testIsPositive() {
        assertTrue(Arithmetic.isPositive(5));
        assertFalse(Arithmetic.isPositive(-5));
        assertFalse(Arithmetic.isPositive(0));
    }


}
```

Assert methods I

- Within a test,
  - Call the method being tested and get the actual result
  - **Assert** what the correct result should be with one of the assert methods
  - These steps can be repeated as many times as necessary
- An assert method is a JUnit method that performs a test, and throws an AssertionError if the test fails
  - JUnit catches these Errors and shows you the result
- static void assertTrue(boolean ***test***)
  static void assertTrue(String ***message***, boolean ***test***)
  - Throws an AssertionErrorif the test fails
  - The optional *message* is included in the Error
- static void assertFalse(boolean ***test***)
  static void assertFalse(String ***message***, boolean ***test***)
  - Throws an AssertionErrorif the test fails

Example: Counter class

- For the sake of example, we will create and test a trivial “counter” class
  - The constructor will create a counter and set it to zero
  - The increment method will add one to the counter and return the new value
  - The decrement method will subtract one from the counter and return the new value
- We write the test methods before we write the code
  - This has the advantages described earlier
  - However, we usually write the method **stubs** first, and let the IDE generate the test method stubs
- Don’t be alarmed if, in this simple example, the JUnit tests are more code than the class itself

JUnit tests for Counter

```
public class CounterTest {
    Counter counter1; // declare a Counter here

    @Before
    void setUp() {
        counter1 = new Counter(); // initialize the Counter here
    }

    @Test
    public void testIncrement() {
        assertTrue(counter1.increment() == 1);
        assertTrue(counter1.increment() == 2);
     }

    @Test
    public void testDecrement() {
        assertTrue(counter1.decrement() == -1);
    }
}
```

- Note that each test begins with a *brand new* counter
- This means you don't have to worry about the order in which the tests are run

The Counter class itself

```
public class Counter {
        int count = 0;

        public int increment() {
            return count += 1;
        }

        public int decrement() {
            return count -= 1;
        }

        public int getCount() {
            return count;
        }
}
```

- Is JUnit testing overkill for this little class?
- The Extreme Programming view is: *If it isn't tested, it doesn't work*
- You are not likely to have many classes this trivial in a real program, so writing JUnit tests for those few trivial classes is no big deal
- Often even XP programmers don't bother writing tests for *simple* getter methods such as `getCount()`
- We only used `assertTrue` in this example, but there are additional assert methods

Warning: equals

- You can compare *primitives* with ==
- Java has a method *x*.equals(*y*), for comparing *objects*
  - This method works great for Strings and a few other Java classes
  - For objects of classes that *you* create, *you* have to define equals
- assertEquals(***expected***, ***actual***) uses == or equals
- To define equals for your own objects, define *exactly* this method:
  public boolean equals(Object ***obj***) { ... }
  - The argument must be of type Object, which isn't what you want, so you must cast it to the correct type (say, Person):

  -
    ```
    public boolean equals(Object something) {
        Person p = (Person)something;
        return this.name == p.name; // test whatever you like here
    }
    ```
- We'll talk *much* more about equals later

Assert methods II

- assertEquals(***expected***, ***actual***)
  assertEquals(String ***message***, ***expected***, ***actual***)
  - ***expected*** and ***actual*** must be both objects *or* the same primitive type
  - For objects, uses your equals method, ***if*** you have defined it properly, as described on the previous slide

- assertSame(Object ***expected***, Object ***actual***)
  assertSame(String ***message***, Object ***expected***, Object ***actual***)
  - Asserts that two arguments refer to the *same* object

- assertNotSame(Object ***expected***, Object ***actual***)
  assertNotSame(String ***message***, Object ***expected***, Object ***actual***)
  - Asserts that two objects do not refer to the same object

Assert methods III

- assertNull(Object ***object***)
  assertNull(String ***message***, Object ***object***)
  - Asserts that the object is null (undefined)

- assertNotNull(Object ***object***)
  assertNotNull(String ***message***, Object ***object***)
  - Asserts that the object is null

- fail()
  fail(String ***message***)
  - Causes the test to fail and throw an AssertionFailedError
  - Useful as a result of a complex test, when the other assert methods aren’t quite what you want

Writing a JUnit test class, III

- **This page is really only for expensive setup, such as when you need to connect to a database to do your testing**
  - If you wish, you can declare *one* method to be executed *just once*, when the class is first loaded

-
  ```
  @BeforeClass
  public static void setUpClass() throws Exception {
          // one-time initialization code
  }
  ```
  - If you wish, you can declare *one* method to be executed *just once*, to do cleanup after all the tests have been completed

-
  ```
  @AfterClass
  public static void tearDownClass() throws Exception {
          // one-time cleanup code
  }
  ```

Special features of @Test

  - You can limit how long a method is allowed to take
  - This is good protection against infinite loops
  - The time limit is specified in milliseconds
  - The test fails if the method takes too long

-
  ```
  @Test (timeout=10)
   public void greatBig() {
     assertTrue(program.ackerman(5, 5) > 10e12);
  }
  ```

  - Some method calls should throw an exception
  - You can specify that a particular exception is expected
  - The test will pass if the expected exception is thrown, and fail otherwise

-
  ```
  @Test (expected=IllegalArgumentException.class)
  public void factorial() {
     program.factorial(-5);
  }
  ```

Test-Driven Development (TDD)

- It is difficult to add JUnit tests to an existing program
  - The program probably wasn't written with testing in mind
- It's actually better to write the tests *before* writing the code you want to test
- This seems backward, but it really does work better:
  - When tests are written first, you have a clearer idea what to do when you write the methods
  - Because the tests are written first, the methods are necessarily written to be testable
  - Writing tests first encourages you to write simpler, single-purpose methods
  - Because the methods will be called from more than one environment (the "real" one, plus your test class), they tend to be more independent of the environment

Stubs

- In order to run our tests, the methods we are testing have to exist, but they don't have to be right
- Instead of starting with "real" code, we start with stubs—minimal methods that always return the same values
  - A stub that returns `void` can be written with an empty body
  - A stub that returns a number can return `0` or `-1` or `666`, or whatever number is most likely to be *wrong*
  - A stub that returns a `boolean` value should usually return `false`
  - A stub that returns an object of any kind (including a `String` or an array) should return `null`
- When we run our test methods with these stubs, we want the test methods to *fail!*
  - This helps "test the tests"—to help make sure that an incorrect method doesn't pass the tests

Ignoring a test

- The @Ignore annotation says to not run a test

```
@Ignore("I don’t want Dave to know this doesn’t work")
@Test
public void add() {
    assertEquals(4, program.sum(2, 2));
}
```

- You shouldn’t use @Ignore without a very good reason!

Test suites

- You can define a suite of tests

```
@RunWith(value=Suite.class)
@SuiteClasses(value={
                    MyProgramTest.class,
                    AnotherTest.class,
                    YetAnotherTest.class
                })
public class AllTests { }
```

JUnit in Eclipse

- If you write your method stubs first (as on the previous slide), Eclipse will generate test method stubs for you
- To add JUnit 4 to your project:
  - Select a class in Eclipse
  - Go to File → New... → JUnit Test Case
  - Make sure New JUnit 4 test is selected
  - Click where it says "Click here to add JUnit 4..."
  - Close the window that appears
- To create a JUnit test class:
  - Do steps 1 and 2 above, if you haven't already
  - Click Next>
  - Use the checkboxes to decide which methods you want test cases for; don't select Object or anything under it
    - I like to check "create tasks," but that's up to you
  - Click Finish
- To run the tests:
  - Choose Run → Run As → JUnit Test

Viewing results in Eclipse

Bar is green if *all* tests pass, red otherwise

Ran 10 of the 10 tests

No tests failed, but...

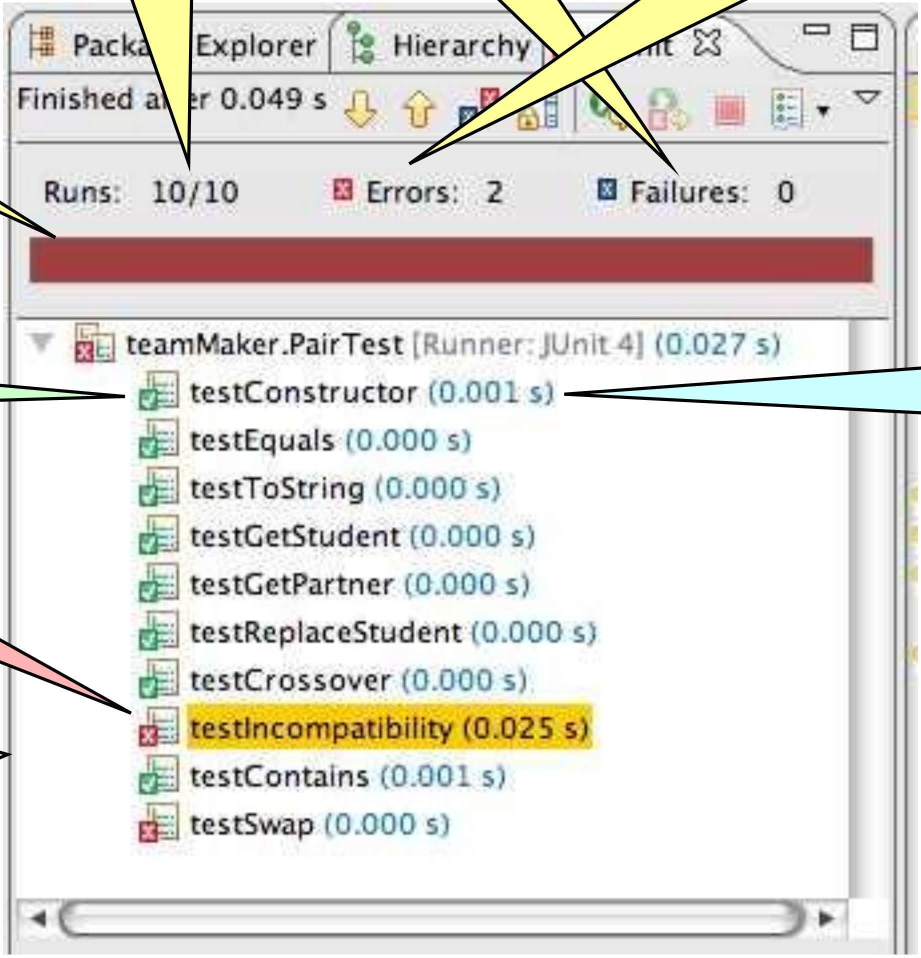
Something unexpected happened in two tests

This test passed

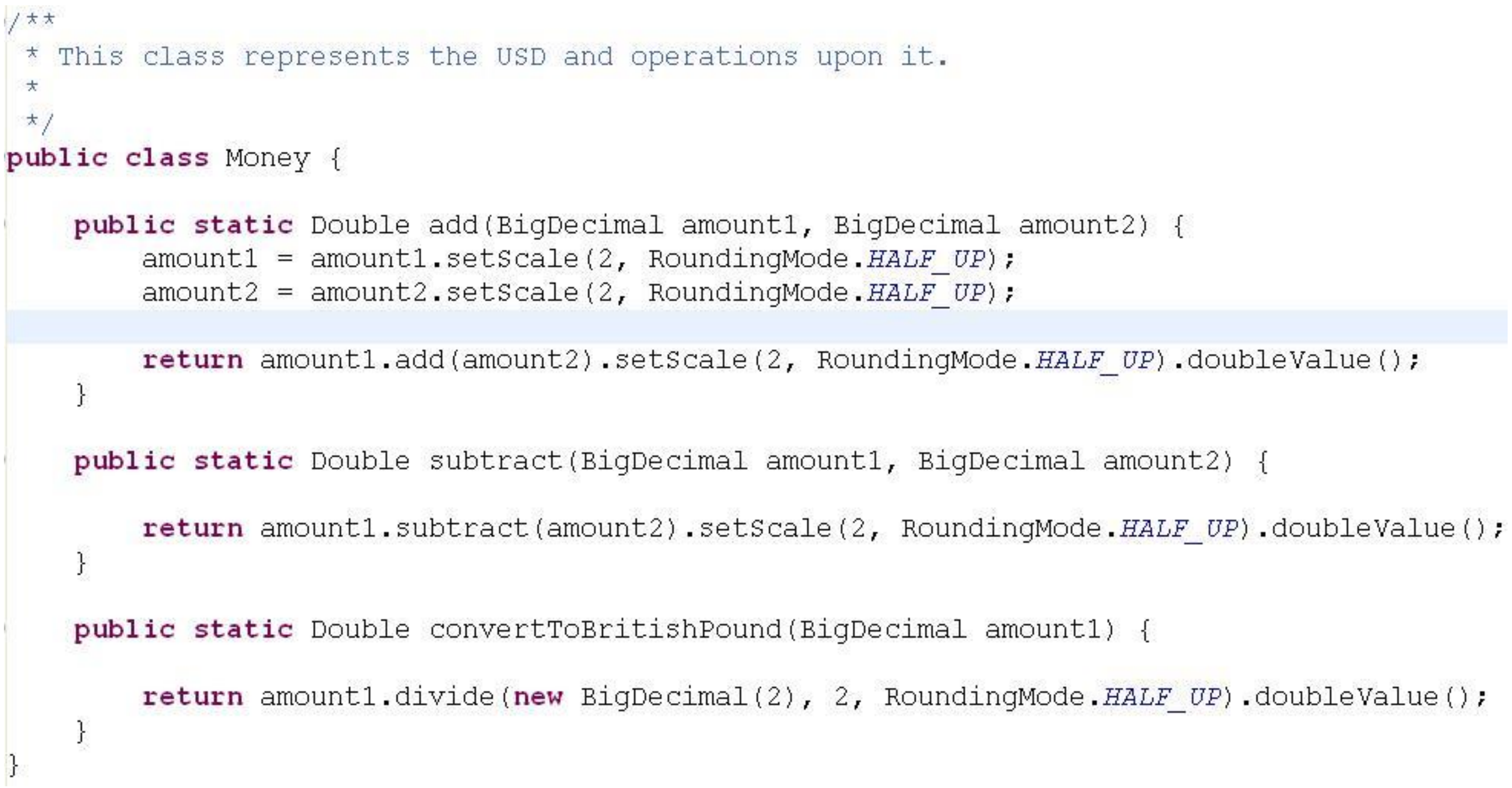
This is how long the test took

Something is wrong

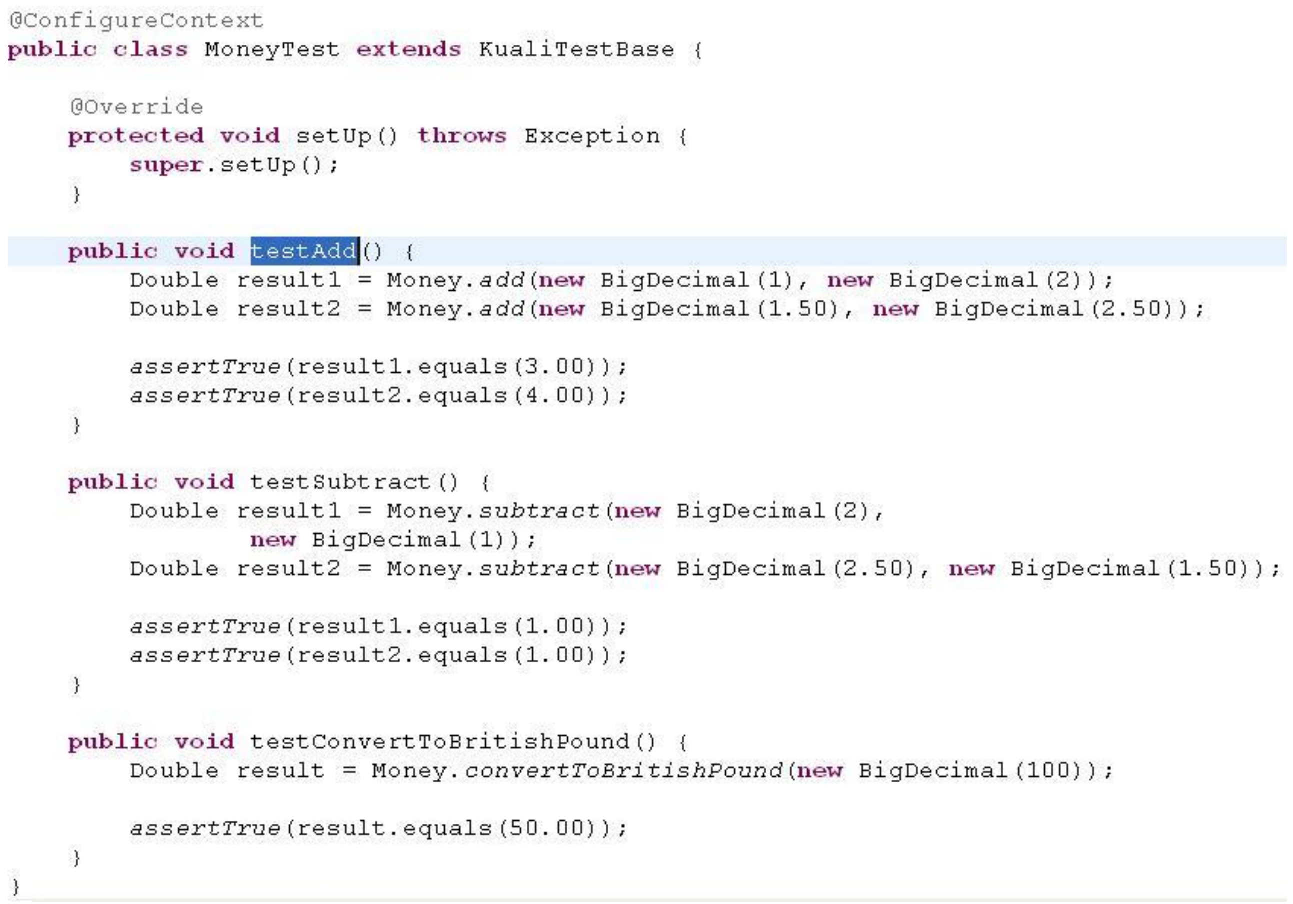
Depending on your preferences, this window might show *only* failed tests



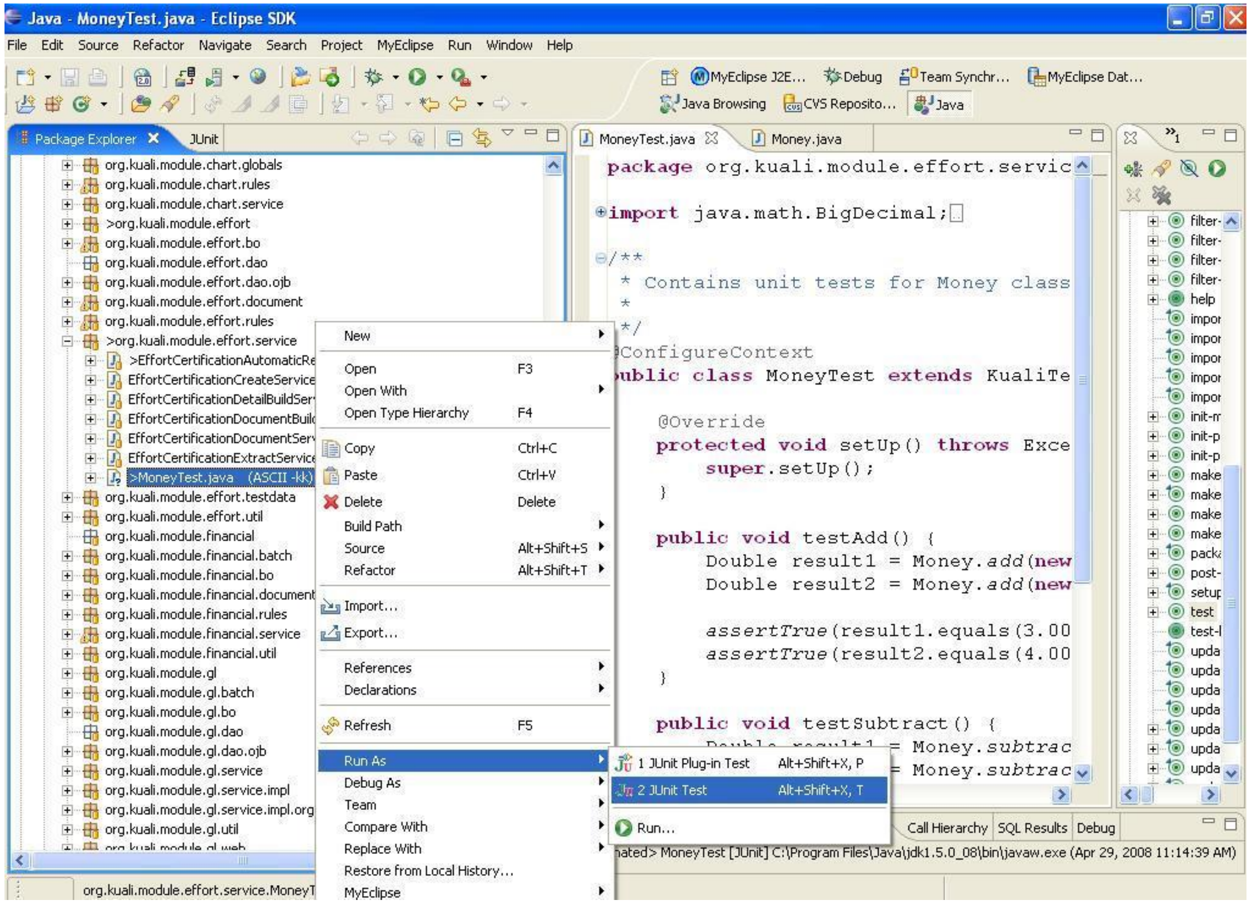
```
Finished after 0.049 s
Runs: 10/10      Errors: 2      Failures: 0

teamMaker.PairTest [Runner: JUnit 4] (0.027 s)
    testConstructor (0.001 s)
    testEquals (0.000 s)
    testToString (0.000 s)
    testGetStudent (0.000 s)
    testGetPartner (0.000 s)
    testReplaceStudent (0.000 s)
    testCrossover (0.000 s)
    testIncompatibility (0.025 s)
    testContains (0.001 s)
    testSwap (0.000 s)
```



```java
/**
 * This class represents the USD and operations upon it.
 *
 */
public class Money {

    public static Double add(BigDecimal amount1, BigDecimal amount2) {
        amount1 = amount1.setScale(2, RoundingMode.HALF_UP);
        amount2 = amount2.setScale(2, RoundingMode.HALF_UP);

        return amount1.add(amount2).setScale(2, RoundingMode.HALF_UP).doubleValue();
    }

    public static Double subtract(BigDecimal amount1, BigDecimal amount2) {

        return amount1.subtract(amount2).setScale(2, RoundingMode.HALF_UP).doubleValue();
    }

    public static Double convertToBritishPound(BigDecimal amount1) {

        return amount1.divide(new BigDecimal(2), 2, RoundingMode.HALF_UP).doubleValue();
    }
}
```



```
@ConfigureContext
public class MoneyTest extends KualiTestBase {

    @Override
    protected void setUp() throws Exception {
        super.setUp();
    }

    public void testAdd() {
        Double result1 = Money.add(new BigDecimal(1), new BigDecimal(2));
        Double result2 = Money.add(new BigDecimal(1.50), new BigDecimal(2.50));

        assertTrue(result1.equals(3.00));
        assertTrue(result2.equals(4.00));
    }

    public void testSubtract() {
        Double result1 = Money.subtract(new BigDecimal(2),
                new BigDecimal(1));
        Double result2 = Money.subtract(new BigDecimal(2.50), new BigDecimal(1.50));

        assertTrue(result1.equals(1.00));
        assertTrue(result2.equals(1.00));
    }

    public void testConvertToBritishPound() {
        Double result = Money.convertToBritishPound(new BigDecimal(100));

        assertTrue(result.equals(50.00));
    }
}
```

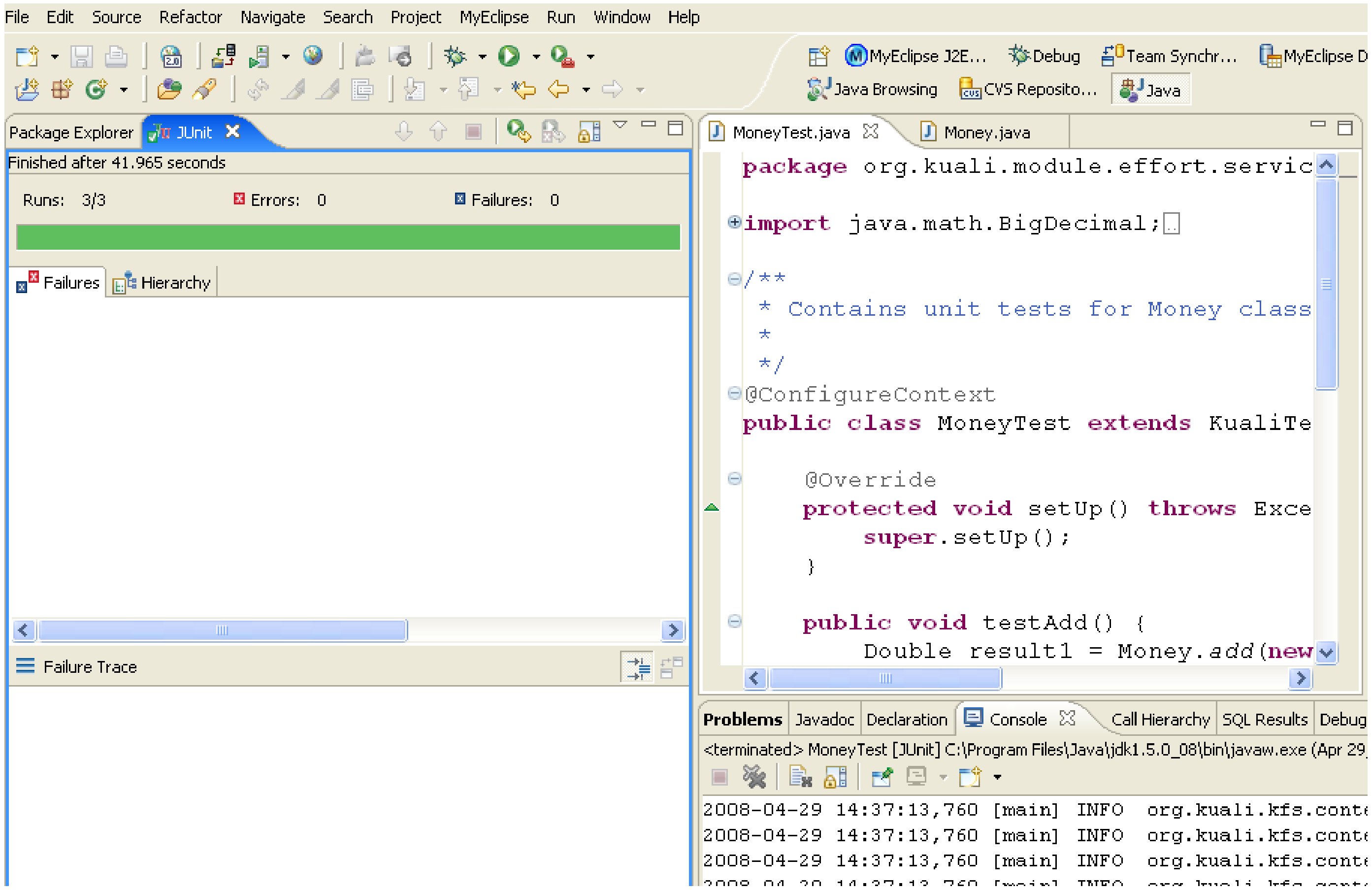

Java - MoneyTest.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project MyEclipse Run Window Help
MyEclipse J2E... Debug Team Synchr... MyEclipse Dat...
Java Browsing CVS Reposito... Java
Package Explorer JUnit
MoneyTest.java Money.java
New
Open F3
Open With
Open Type Hierarchy F4
Copy Ctrl+C
Paste Ctrl+V
Delete Delete
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
References
Declarations
Refresh F5
Run As
Debug As
Team
Compare With
Replace With
Restore from Local History...
MyEclipse
1 JUnit Plug-in Test Alt+Shift+X, P
2 JUnit Test Alt+Shift+X, T
Run...
Call Hierarchy SQL Results Debug

```
org.kuali.module.chart.globals
org.kuali.module.chart.rules
org.kuali.module.chart.service
>org.kuali.module.effort
org.kuali.module.effort.bo
org.kuali.module.effort.dao
org.kuali.module.effort.dao.ojb
org.kuali.module.effort.document
org.kuali.module.effort.rules
>org.kuali.module.effort.service
    >EffortCertificationAutomaticRe
    EffortCertificationCreateService
    EffortCertificationDetailBuildSer
    EffortCertificationDocumentBuil
    EffortCertificationDocumentSer
    EffortCertificationExtractServic
    >MoneyTest.java (ASCII -kk)
org.kuali.module.effort.testdata
org.kuali.module.effort.util
org.kuali.module.financial
org.kuali.module.financial.batch
org.kuali.module.financial.bo
org.kuali.module.financial.document
org.kuali.module.financial.rules
org.kuali.module.financial.service
org.kuali.module.financial.util
org.kuali.module.gl
org.kuali.module.gl.batch
org.kuali.module.gl.bo
org.kuali.module.gl.dao
org.kuali.module.gl.dao.ojb
org.kuali.module.gl.service
org.kuali.module.gl.service.impl
org.kuali.module.gl.service.impl.org
org.kuali.module.gl.util
```

```
package org.kuali.module.effort.servic

import java.math.BigDecimal;

/**
 * Contains unit tests for Money class
 *
 */
@ConfigureContext
ublic class MoneyTest extends KualiTe

    @Override
    protected void setUp() throws Exce
        super.setUp();
    }

    public void testAdd() {
        Double result1 = Money.add(new
        Double result2 = Money.add(new

        assertTrue(result1.equals(3.00
        assertTrue(result2.equals(4.00
    }

    public void testSubtract() {
        Double result1 = Money.subtrac
                       = Money.subtrac
```

```
nated> MoneyTest [JUnit] C:\Program Files\Java\jdk1.5.0_08\bin\javaw.exe (Apr 29, 2008 11:14:39 AM)
```

org.kuali.module.effort.service.MoneyT

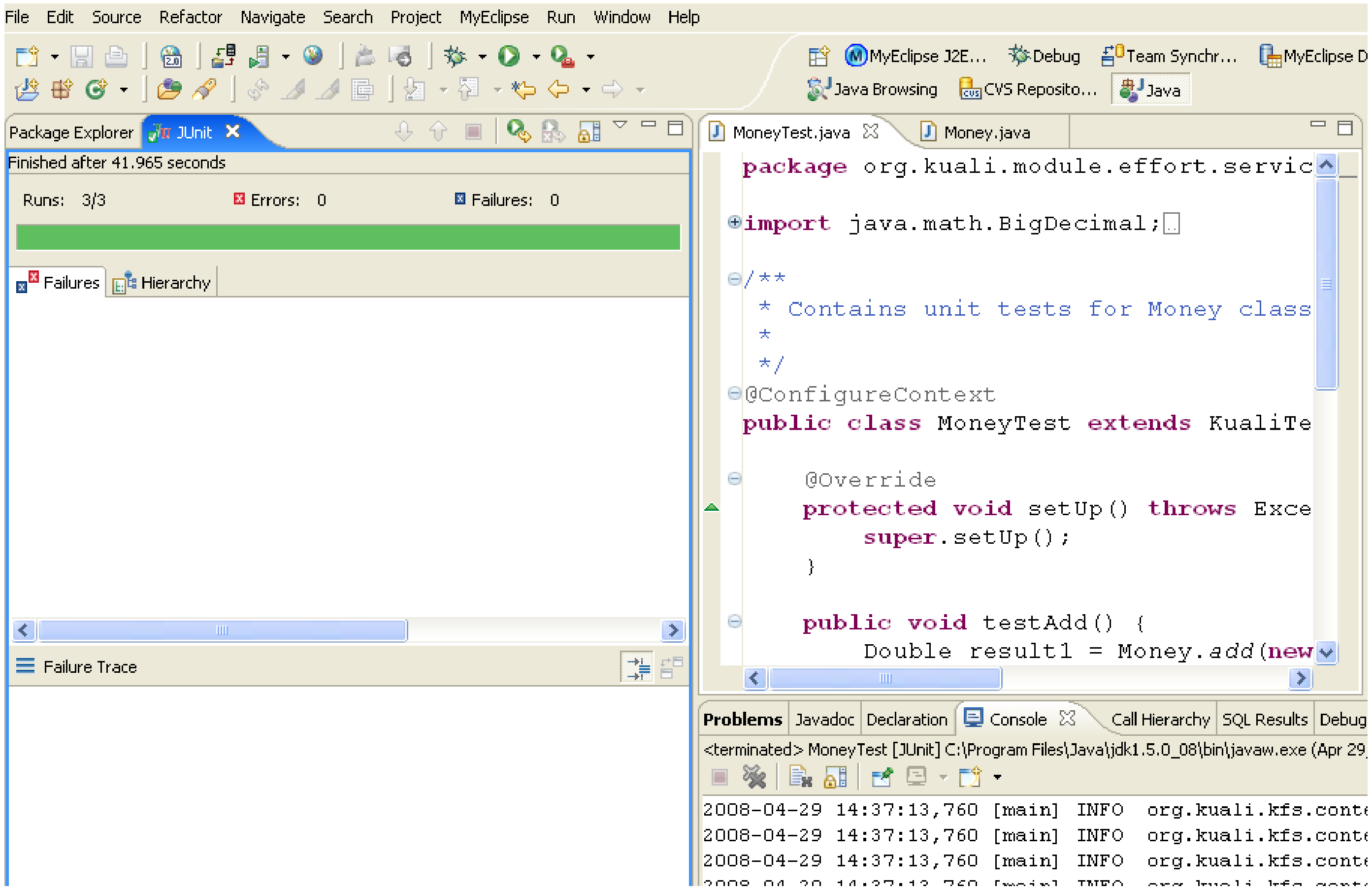


Finished after 41.965 seconds

Runs: 3/3 Errors: 0 Failures: 0

```
package org.kuali.module.effort.servic

import java.math.BigDecimal;

/**
 * Contains unit tests for Money class
 *
 */
@ConfigureContext
public class MoneyTest extends KualiTe

    @Override
    protected void setUp() throws Exce
        super.setUp();
    }

    public void testAdd() {
        Double result1 = Money.add(new
```

```
<terminated> MoneyTest [JUnit] C:\Program Files\Java\jdk1.5.0_08\bin\javaw.exe (Apr 29
2008-04-29 14:37:13,760 [main] INFO  org.kuali.kfs.conte
2008-04-29 14:37:13,760 [main] INFO  org.kuali.kfs.conte
2008-04-29 14:37:13,760 [main] INFO  org.kuali.kfs.conte
```



Finished after 41.965 seconds

Runs: 3/3 Errors: 0 Failures: 0

```
package org.kuali.module.effort.servic

import java.math.BigDecimal;

/**
 * Contains unit tests for Money class
 *
 */
@ConfigureContext
public class MoneyTest extends KualiTe

    @Override
    protected void setUp() throws Exce
        super.setUp();
    }

    public void testAdd() {
        Double result1 = Money.add(new
```

<terminated> MoneyTest [JUnit] C:\Program Files\Java\jdk1.5.0_08\bin\javaw.exe (Apr 29

```
2008-04-29 14:37:13,760 [main] INFO  org.kuali.kfs.conte
2008-04-29 14:37:13,760 [main] INFO  org.kuali.kfs.conte
2008-04-29 14:37:13,760 [main] INFO  org.kuali.kfs.conte
```

Recommended approach

- Write a test for some method you intend to write
  - If the method is fairly complex, test only the simplest case
- Write a stub for the method
- Run the test and make sure it fails
- Replace the stub with code
  - Write just enough code to pass the tests
- Run the test
  - If it fails, debug the method (or maybe debug the test); repeat until the test passes
- If the method needs to do more, or handle more complex situations, add the tests for these first, and go back to step 3

Why JUnit

- Allow you to write code faster while increasing quality
- Elegantly simple
- Check their own results and provide immediate feedback
- Tests are inexpensive
- Increase the stability of software
- Developer tests
- Written in Java
- Free
- Gives proper understanding of unit testing

Problems with unit testing

- JUnit is designed to call methods and compare the results they return against expected results
  - This ignores:
    - Programs that do work in response to GUI commands
    - Methods that are used primary to produce output

Problems with unit testing…

- Heavy use of JUnit encourages a “functional” style, where most methods are called to compute a value, rather than to have side effects
  - This can actually be a good thing
  - Methods that *just* return results, without side effects (such as printing), are simpler, more general, and easier to reuse

The End

If you don't unit test then you aren't a software engineer, you are a typist who understands a programming language.

--Moses Jones

1. Never underestimate the power of one little test.
2. There is no such thing as a dumb test.
3. Your tests can often find problems where you're not expecting them.
4. Test that everything you say happens actually does happen.
5. If it's worth documenting, it's worth testing.

--Andy Lester