Software Testing, Debugging and JUnit

Ananda Gunawardena

Hao Cen

Software Development Process

- A simplified process
  - User requirement → Development → Delivery
- A simplified process with software errors
  - User requirement → Development + Testing + Debugging → Delivery

Program Testing and Debugging

- Testing is to see if the program is correct.
  - Testing does not change your original program.
  - You get a Yes/No answer.
  - You need to what is correct before you test the program.
- Debugging is to remove the error after you know the program is wrong.
  - Debugging changes your original program.
  - Debugging succeeds if the corresponding test says yes.
- Both of them are time-consuming. The best way is to reduce software errors and defects from the beginning.

Program Testing

- Why testing
  - Program errors are inevitable in the development. If not corrected, they may cost your money, time, grade and job.
- Who tests
  - Programmer (Programmer Testing)
  - QA
- What to test
  - The whole software system
  - Individual classes (Object Testing)
- Types of testing
  - Exploratory testing
  - Automated testing

Example

- Program Requirement -- write a class to model money
  - The money class records its currency type and amount
  - Two Monies are considered equal if they have the same currency and value.
  - When you add two Moneys of the same currency, the resulting Money has as its amount the sum of the other two amounts.

```
package junitEx;

public class Money {
    private int amount;

    private String currency;

    public Money(int amount, String currency) {
          this.amount = amount;
          this.currency = currency;
      }

    public int getAmount() {
        return amount;
    }

    public String getCurrency() {
        return currency;
    }

    public boolean equals(Object other) {
       if (other == this) {
                    return true;
                }
       if (!(other instanceof Money)) {
          return false;
       }
       Money that = (Money) other;
       return  getCurrency().equals(that.getCurrency())
       && getAmount() == that.getAmount();
     }

    public Money add(Money m) {
        return new Money(this.getAmount() + m.getAmount(),
this.getCurrency());
    }
}
```

Example

- Test `Money`
  - Write a class `MoneyTest` to test `Money`
  - Create one or more Money objects
  - Invoke `equals()`, `add()`
  - Check these methods return expected results
- Demo

```
package junitEx;

import junit.framework.Assert;
import junit.framework.TestCase;

public class MoneyTest extends TestCase {
    private Money m1;
    private Money m11;
    private Money m2;

    protected void setUp() {
        m1= new Money(1, “USD");
        m11= new Money(1, "USD");
        m2= new Money(2, "USD");
    }


    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testEquals() {
        assertTrue(!m1.equals(null));
        assertEquals(m1, m1);
        assertEquals(m1, m11);
        assertTrue(!m1.equals(m2));
    }

    public void testAdd() {
        Money expected= new Money(3, "USD");
        Money result= m1.add(m2);
        assertEquals(expected, result);
    }
}
```

Unit Testing/Object Testing

- Goal – Take a single object (our unit) and test it by itself, make sure it behavior correctly according to a defined specification, without worrying about the role it plays in the surrounding system
  - If each object behavior correctly, the whole system is more likely to behave correctly
- How -- “if I invoke this method on that object, it should respond so.”
  - Create an object
  - Invoke a method
  - Check the result

Properties of Unit Testing

- Automated
  - Not having a human tester sitting in front of a screen, entering text, pushing a button and watching what happens. (this is end-to-end testing through an end user interface.)
  - Not `System.out.println()` or setting breakpoints, running coding in a debugger, and analyzing the value of variables. (this is code-level testing, i.e. debugging)
  - ➔ Need some code to represent the test
- Repeatable
  - Executing the same test several times under the same condition must yield the same results.
  - ➔ The code need to return the same results under the same condition.
- Self-verifying
  - Not having a programmer verify the expected results
  - ➔ The code needs to verify the expected results
- Can run simulatenously
  - Run individually, in batch, in a particular sequence
  - ➔ The code needs grouping mechnism.

Review the money test

- Test `Money`
  - Write a class `MoneyTest` to test `Money`
  - Create one or more Money objects
  - Invoke `equals()`, `add()`
  - Check these methods return expected results

```
package junitEx;

import junit.framework.Assert;
import junit.framework.TestCase;

public class MoneyTest extends TestCase {
    private Money m1;
    private Money m11;
    private Money m2;

    protected void setUp() {
        m1= new Money(1, “USD");
        m11= new Money(1, "USD");
        m2= new Money(2, "USD");
    }


    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testEquals() {
        assertTrue(!m1.equals(null));
        assertEquals(m1, m1);
        assertEquals(m1, m11);
        assertTrue(!m1.equals(m2));
    }

    public void testAdd() {
        Money expected= new Money(3, "USD");
        Money result= m1.add(m2);
        assertTrue(expected.equals(result));
    }
}
```

JUnit

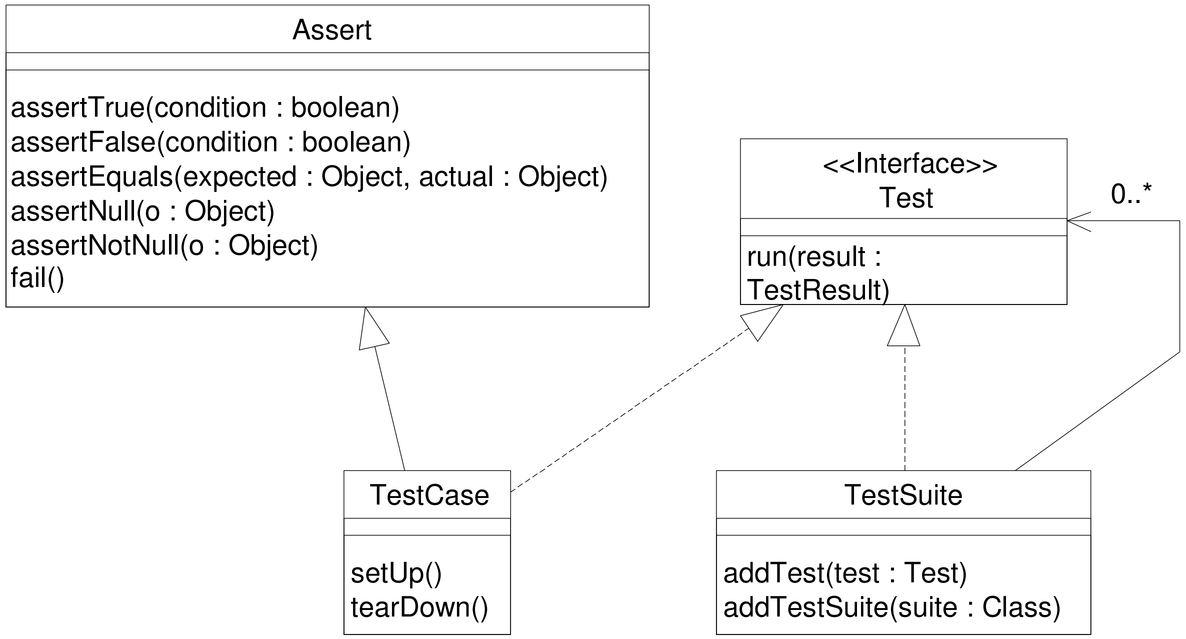
- Purpose
  - A Java library created as a framework for writing automated, self-verifying tests in Java
  - Each test is called a *test case*.
  - Related test cases can be grouped into a *test suite*.
  - The *test runner* lets you execute a test case or a test suite and tells if any one fails.
- Tests and Programmers
  - Programmers put their knowledge into the test.
  - The tests become programmer independent.
    - Any one can run the tests without knowing what they do. If they are curious, they only need to read the test code.

Writing tests in JUnit

- Steps to test a class xx
  - Create a class xxTest that extends TestCase
  - Declare xx objects as fields and create them in setup()
  - Create method testMethod() in xxTest to test method() in xx
  - Invoke method()
  - Assert if the expected results is the same as the returned results.
- Execution
  - If no assertion fails, the test passes, else the test fails.

```java
package junitEx;

import junit.framework.Assert;
import junit.framework.TestCase;

public class MoneyTest extends TestCase {
    private Money m1;
    private Money m11;
    private Money m2;

    protected void setUp() {
        m1= new Money(1, “USD");
        m11= new Money(1, "USD");
        m2= new Money(2, "USD");
    }


    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testEquals() {
        assertTrue(!m1.equals(null));
        assertEquals(m1, m1);
        assertEquals(m1, m11);
        assertTrue(!m1.equals(m2));
    }

    public void testAdd() {
        Money expected= new Money(3, "USD");
        Money result= m1.add(m2);
        assertTrue(expected.equals(result));
    }
}
```

JUnit Hierarchy


Assert
assertTrue(condition : boolean)
assertFalse(condition : boolean)
assertEquals(expected : Object, actual : Object)
assertNull(o : Object)
assertNotNull(o : Object)
fail()
<<Interface>>
Test
run(result : TestResult)
0..*
TestCase
setUp()
tearDown()
TestSuite
addTest(test : Test)
addTestSuite(suite : Class)

- Test Case, a single test, verifying a specific path through code.
- `TestCase`, such as `MoneyTest`, collects a multiple test cases into a single class. Your test classes will inherit from `TestCase` and will implement one or more test methods having this pattern: `testXxx()` for example, `testAdd()`
- Test Suite, a collection of tests that are run at the same time. Often construted dynamically from selected test cases.
- Test Runner, a utility to run test cases and test suites. May provide a graphical presentation, such as Eclispe view, or a command-line execution.

JUnit Hierarchy

Assert class provides the methods you will use to make assertions about the state of your objects. TestCase extends Assert so that you can write your assertions without having to refer to an outside class.

Method	What it does
`assertTrue(boolean condition)`	Fails if condition is false; passes otherwise.
`assertEquals(Object expected, Object actual)`	Fails if expected and actual are not equal, according to the equals() method; passes otherwise.
`assertEquals(int expected, int actual)`	Fails if expected and actual are not equal according to the == operator; passes otherwise. (overloaded for each primitive type:int, float, double, char, byte, long, short, and boolean)
`assertSame(Object expected, Object actual)`	Fails if expected and actual refer to different objects in memory; passes if they refer to the same object in memory. Objects that are not the same might still be equal according to the equals() method.
`assertNull(Object object)`	Passes if object is null; fails otherwise

JUnit Recipes

- Q: How do I verify the return value of a method?
  - Use `assertEquals(Object expected, Object actual)` if the method returns an object
  - Use `assertEquals(int expected, int actual)` or its overloaded version if the method returns a primitive

E.g. test if the Money add method returns the money with the sum

```
public void testAdd() {
        Money expected= new Money(3, "USD");
        Money result= m1.add(m2);
        assertEquals(expected, result);
    }
```

E.g. test if a new List is empty

```
public void testNewListIsEmpty(){
   List list = new ArrayList();
   assertEquals(0, list.size());
}
```

General form:

1. Create an object and put it in a known state
2. Invoke a method, which returns the “actural result”
3. Create the “expected retuls”
4. Invoke `assertEquals(expected, actual)`

JUnit Recipes

- Q: How do I verify `equals()`?

- Q: How do I write `equals()`?
  - Java API for `Object public boolean equals(Object obj)`
    - Overwritten for any value objects, an object that represents a value, such as `Integer`, `String`, `Money`. Two value objects are equal if if their values are equal even if they are in different memory locations.
    - == compares the memory location

```
String s1 = “cmu”; String s2 = “cmu”; String s3 = “cmu”;
s1.equals(s2); //returns true
s1.equals(s3); //returns true
S1 == s2 ;//returns false
```

    - If an object doesn’t overwrite `equals(), equals()` calls == by default.

```
Money m1 = new Money(1, “USD");
Money m2 = new Money(1, "USD");
m1.equals(m11) // true, if Money overwrites equals(),
m1.equals(m11) // false, if Money does not overwrite
  equals()
```

JUnit Recipes

- Q: How do I write `equals()`?
  - Properties of equals()
    - *reflexive*: x.equals(x) returns true.
    - *symmetric*: x.equals(y) returns true if and only if y.equals(x) returns true.
    - *transitive*: if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
    - *consistent*: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
    - *Not equal to null*: x.equals(null) returns false.

General form

```
public boolean equals(Object other) {
  if (other == this) {
    return true;
  }
  if (! (other instanceof X)) {
    return false;
  }
  X that = (X) other;

//compare this and that field by field
//if all the important fields are equal
// return true;
//else return false;

}
```

e.g.

```
public boolean equals(Object other) {
  if (other == this) {
         return true;
      }
  if (!(other instanceof Money)) {
     return false;
  }
  Money that = (Money) other;
  return  getCurrency().equals(that.getCurrency())
  && getAmount() == that.getAmount();
}
```

JUnit Recipes

- Q: How do I verify `equals()`?

```
public class MoneyTest extends TestCase {
   private Money a,b,c,d
   protected void setUp(){
     a = new Money(100, “USD”);
     b = new Money(100, “USD”);
     c = new Money(200, “USD”);
     d = new Money(200, “USD”);
   }

   public void testEquals() {
        assertEquals(a,a); //reflexive
        assertEquals(a,b); //symmetric
        assertEquals(b,a);
        assertFalse(a.equals(c)) ;// symmetric
        assertFalse(c.equals(a))

        for (int I = 0; I < 1000; i++){ //consistent
               assertEquals(a,b);
               assertFalse(a.equals(c));
        }

        assertFalse(a.equals(null)); //not equal to null
     }
}
```

JUnit Recipes

- Q: How do I verify a method that returns void?
  - Test the method's observable side effect, if the method does something. If the method does nothing, delete the method.

E.g. Test the `clear()` method of a list

Java API: public void clear()

- Removes all of the elements from this list. The list will be empty after this call returns.

```
public void testListClear() {
    List list = new ArrayList();
    list.add("hello");
    list.clear();
    assertEquals(0, list.size());
    }
}
```

JUnit Recipes

- Q: How do I test a constructor?
  - If the constructor exposes readable properties (e.g. has getter methods), test their values against the values you had passed into the constructor.

E.g. Test the constructor of Integer

Java API: **Integer**(int value)

Constructs a newly allocated Integer object that represents the specified int value.

```
public void testInteger() {
       Integer integer = new Integer(123);
        assertEquals(123, integer.intValue());
       }
}
```

Ex. Test the constructor of TicTacToe

- If the constructor does not expose readable properties, write a method `isValid()` within that class, i.e. test within that class

```
public void testBankDepositeCommand() {
    BankDepositeCommand command = new BankDepositeCommand(“123”, “USD”, “June 5,
   2006”);
    assertTrue(command.isValid());
       }
}
```

JUnit Recipes

- Q: How do I test a getter?
  - If the getter simply returns the value, do not test it (too simply to break).
  - If the getter does some calculation before returns the value, test it.

E.g. No need to test the getter methods of Money

```
   public int getAmount() {
     return amount;
   }

   public String getCurrency() {
     return currency;
   }
```

Eg. Need to test the a special getter methods of Money

```
public int getAmountInHundreds() {
     return amount/100;
   }

public void testGetAmountInHundreds() {
   Money m = new Money(12300, “USD”);
   assertEquals(123, m.getAmountInHundreds());
}
```

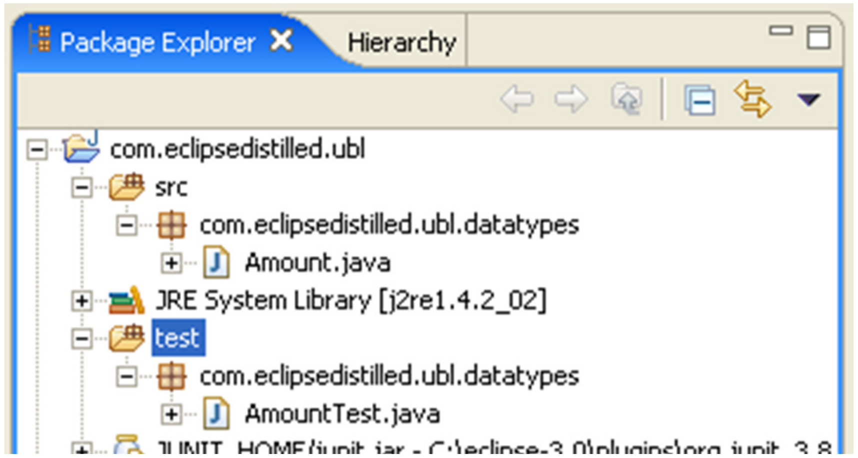
JUnit Recipes

- Q: How do I test a setter?
  - If the setter simply sets a new value, do not test it (too simply to break).
  - If the setter does some calculation before sets the value, test it.

E.g. No need to test the getter methods of Money

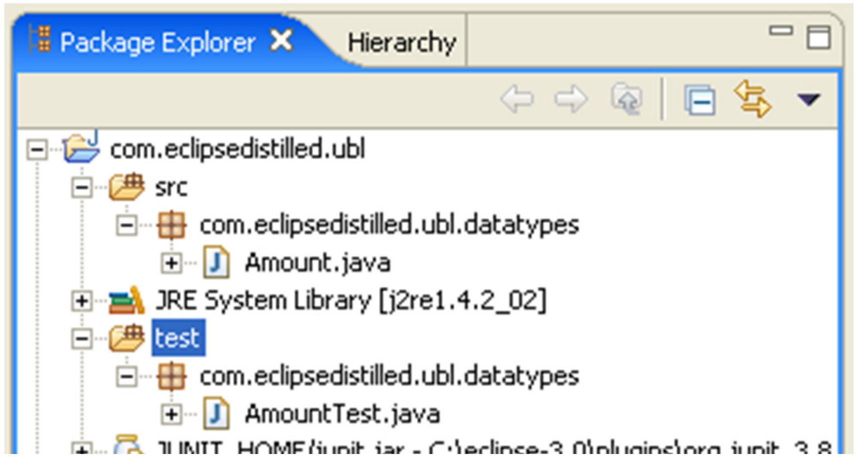
```
public int setAmount(double amount) {
   this.amount = amount;
 }

 public String setCurrency(String currency) {
   this.currency = currency;
 }
```

Writing JUnit Tests in Eclipse (example and pictures from ecliseDistilled)



```
com.eclipsedistilled.ubl
  src
    com.eclipsedistilled.ubl.datatypes
      Amount.java
  JRE System Library [j2re1.4.2_02]
  test
    com.eclipsedistilled.ubl.datatypes
      AmountTest.java
  JUNIT_HOME/junit.jar - C:\eclipse-3.0\plugins\org.junit_3.8
```

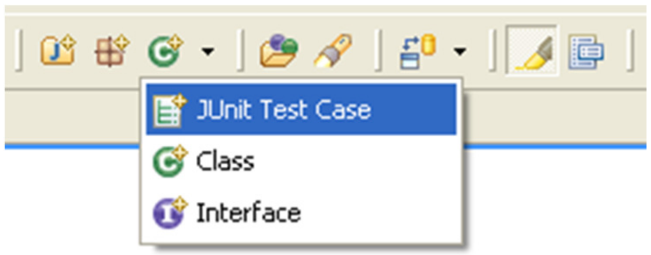
Organizing Unit Tests in Eclipse

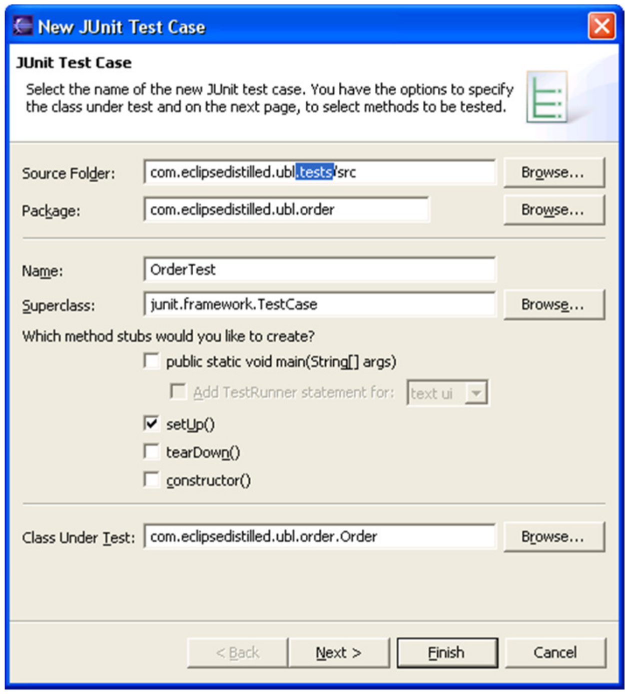


```
com.eclipsedistilled.ubl
  src
    com.eclipsedistilled.ubl.datatypes
      Amount.java
  JRE System Library [j2re1.4.2_02]
  test
    com.eclipsedistilled.ubl.datatypes
      AmountTest.java
  JUNIT_HOME/junit.jar - C:\eclipse-3.0\plugins\org.junit_3.8
```

- Create second source folder
  - Do not put JUnit tests in the same source folder as your project code
  - A second source folder allows clear separation
- File>New>Source Folder

Creating Unit Tests in Eclipse




New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

Source Folder: com.eclipsedistilled.ubl.tests/src

Package: com.eclipsedistilled.ubl.order

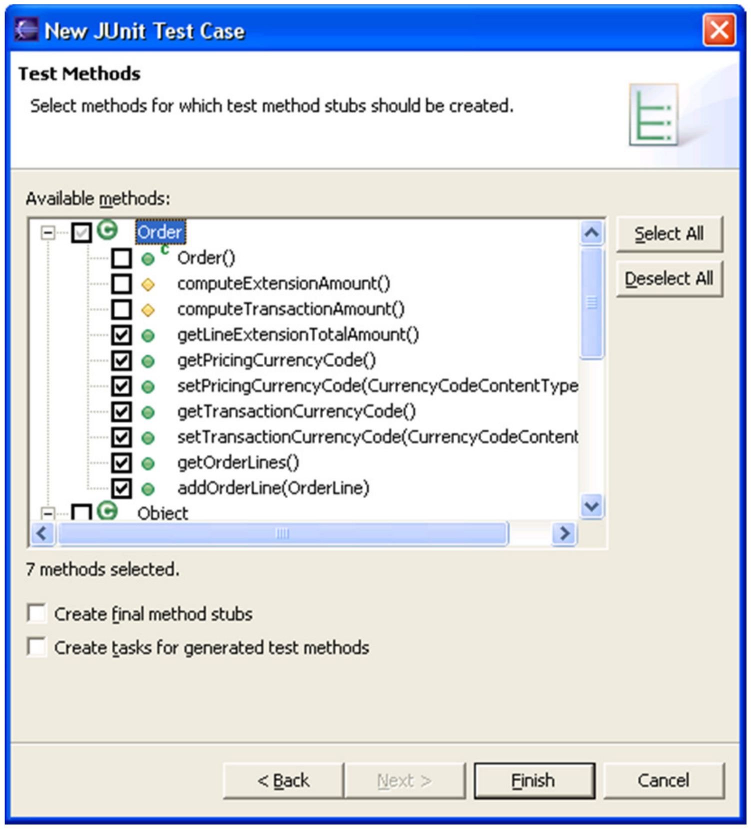
Name: OrderTest

Superclass: junit.framework.TestCase

Which method stubs would you like to create?

- [ ] public static void main(String[] args)
  - [ ] Add TestRunner statement for: text ui
- [x] setUp()
- [ ] tearDown()
- [ ] constructor()

Class Under Test: com.eclipsedistilled.ubl.order.Order


New JUnit Test Case

Test Methods

Select methods for which test method stubs should be created.

Available methods:

- Order
  - [ ] Order()
  - [ ] computeExtensionAmount()
  - [ ] computeTransactionAmount()
  - [x] getLineExtensionTotalAmount()
  - [x] getPricingCurrencyCode()
  - [x] setPricingCurrencyCode(CurrencyCodeContentType
  - [x] getTransactionCurrencyCode()
  - [x] setTransactionCurrencyCode(CurrencyCodeContent
  - [x] getOrderLines()
  - [x] addOrderLine(OrderLine)
- [ ] Object

7 methods selected.

- [ ] Create final method stubs
- [ ] Create tasks for generated test methods

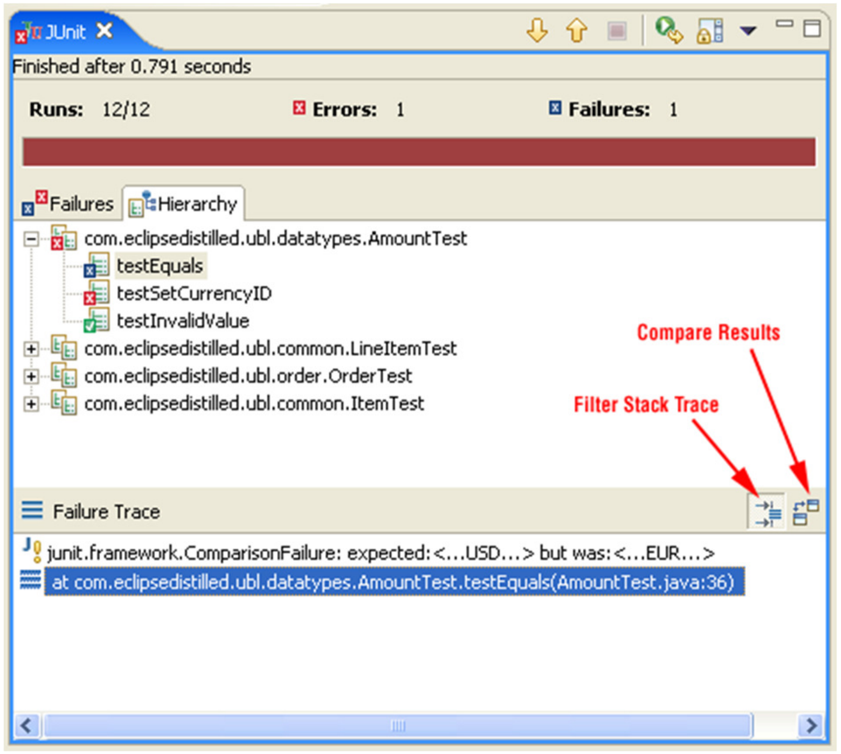
Genearated Test Caes

```
public class OrderTest extends TestCase {

    /*
     * @see TestCase#setUp()
     */
    protected void setUp() throws Exception {
        super.setUp();
    }

    public void testGetLineExtensionTotalAmount() {
    }

    public void testGetPricingCurrencyCode() {
    }

    public void testSetPricingCurrencyCode() {
    }

    public void testGetTransactionCurrencyCode() {
    }

    public void testSetTransactionCurrencyCode() {
    }

    public void testGetOrderLines() {
    }

    public void testAddOrderLine() {
    }
}
```

Write the test methods

```
public class OrderTest extends TestCase {

    private LineItem lineItem1;
    private Item item1;
    private Order order1;

    protected void setUp() throws Exception {
        item1 = new Item();
        lineItem1 = new LineItem(item1);
        order1 = new Order();
        order1.addOrderLine(new OrderLine(lineItem1));
    }

    public void testGetLineExtensionTotalAmount() {
        try {
            AmountType amount = order1.getLineExtensionTotalAmount();
            assertNotNull("Order extension amount is null", amount);
            assertEquals("Order extension amount does not equal
    300.00",
                        amount.get_value(), new BigDecimal("300.00"));
        } catch (PriceException e) {
            fail("Unexpected PriceException");
        }
    }
```

Running JUnit in Eclipse



Finished after 0.791 seconds

Runs: 12/12 Errors: 1 Failures: 1

```
com.eclipsedistilled.ubl.datatypes.AmountTest
    testEquals
    testSetCurrencyID
    testInvalidValue
com.eclipsedistilled.ubl.common.LineItemTest
com.eclipsedistilled.ubl.order.OrderTest
com.eclipsedistilled.ubl.common.ItemTest
```

```
junit.framework.ComparisonFailure: expected:<...USD...> but was:<...EUR...>
at com.eclipsedistilled.ubl.datatypes.AmountTest.testEquals(AmountTest.java:36)
```

Run as > JUnit Test

Filter Stack Trace: remove stack trace entries related to JUnit infrastructure

Good Practices in Testing

- “Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead” -- Martin Fowler
- During Development- When you need to add new functionality to the system, write the tests first. Then, you will be done developing when the test runs.
- During Debugging- When someone discovers a defect in your code, first write a test that will succeed if the code is working. Then debug until the test succeeds.
- In the end your tests should cover your whole application.
- Run all tests every time you change a method.

Debugging

- After JUnits returns failures, you need to find out where the errors are
- Two approaches in debugging
  - Add System.out.println() to print out suspected values
  - Use a debugger.

- Demo

Useful Links and Books

- Junit tutorial
  - http://junit.sourceforge.net/doc/testinfected/testing.htm
- Unit Testing in Eclipse Using JUnit
  http://open.ncsu.edu/se/tutorials/junit/#section7_0
- Eclispe Distilled
- JUnit Recipes