Test with JUnit

SOftEng
http://softeng.polito.it

Version 2 - April 2015



Motivation

- Why testing?
  - Improve software design
  - Make software easier to understand
  - Reduce debugging time
  - Catch integration errors
- In short, to produce better code
- Preconditions
  - Working code
  - Good set of unit tests

What is a test case

- A test case is a document that describes
  - an input, action, or event and
  - an expected response,
  - to determine if a feature of an application is working correctly

Good test case design

- A good test case satisfies the following criteria:
  - Reasonable probability of catching an error
  - Does interesting things
  - Doesn't do unnecessary things
  - Neither too simple nor too complex
  - Not redundant with other tests
  - Makes failures obvious
  - Mutually Exclusive, Collectively Exhaustive

Testing with JUnit

- JUnit is a testing framework for Java programs
  - Idea of Kent Beck
- It is a framework with unit-testing functionalities
- Integrated in Eclipse development Environment
- ***http://www.junit.org***

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

JUnit

- 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 (1)

- JUnit runs a suite of tests and reports results
- For *each* test method in the test suite:
  - JUnit calls the method
  - If the methods terminates without problems
    - The test is considered passed.

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

Assert*()

- For a condition
  - assertTrue("message when test fails", condition);
- If the tested condition is
  - True => execute the following instruction
  - False => break to the end of the test method, print out the optional message

Assert (2)

- For objects, int, long, byte:
  - assertEquals(expected value, expression);
  - Es. `assertEquals( 2 , unoStack.size() );`
- For floating point values:
  - assertEquals(expected value, expression, error);
  - Es. `assertEquals(1.0, Math.cos(3.14), 0.01);`

Test Example

```
public class StackTest extends TestCase {
  public void testStack() {
    Stack aStack = new Stack();
    assertTrue("Stack should be empty!",
                    aStack.isEmpty());
    aStack.push(10);
    assertTrue("Stack should not be empty!",
                    aStack.isEmpty());
    aStack.push(-4);
    assertEquals(-4, aStack.pop());
    assertEquals(10, aStack.pop());
  }
}
```

Extends TestCase

Test method name: testSomething

One or more assertions to check results

Testing a stack...

```
public void testStackEmpty() {
   Stack aStack = new Stack();
   assertTrue("Stack should be empty!",
                     aStack.isEmpty());
   aStack.push(10);
   assertTrue("Stack should not be empty!",
                     !aStack.isEmpty());
}
public void testStackOperations() {
   Stack aStack = new Stack();
   aStack.push(10);
   aStack.push(-4);
   assertEquals(-4, aStack.pop());
   assertEquals(10, aStack.pop());
}
```

Other assert*X* methods

```
assertTrue(boolean test)

assertFalse(boolean test)

assertEquals(expected, actual)

assertSame(Object expected,
           Object actual)

assertNotSame(Object expected,
              Object actual)

assertNull(Object object)
```

Other assert*X* methods

- `assertNotNull(Object object)`
- `fail()`
- All the above may take an optional String message as the first argument, e.g.

```
static void assertTrue(
                String message,
                boolean test)
```

Running a JUnit test case

- Running a JUnit test case :
  - Executes all public methods starting with “test”
  - Ignores the rest
- The class cancontain helper methods
  - They are not public
  - Or they don’t start with “test”

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 a piece of 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

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)

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

TestSuite

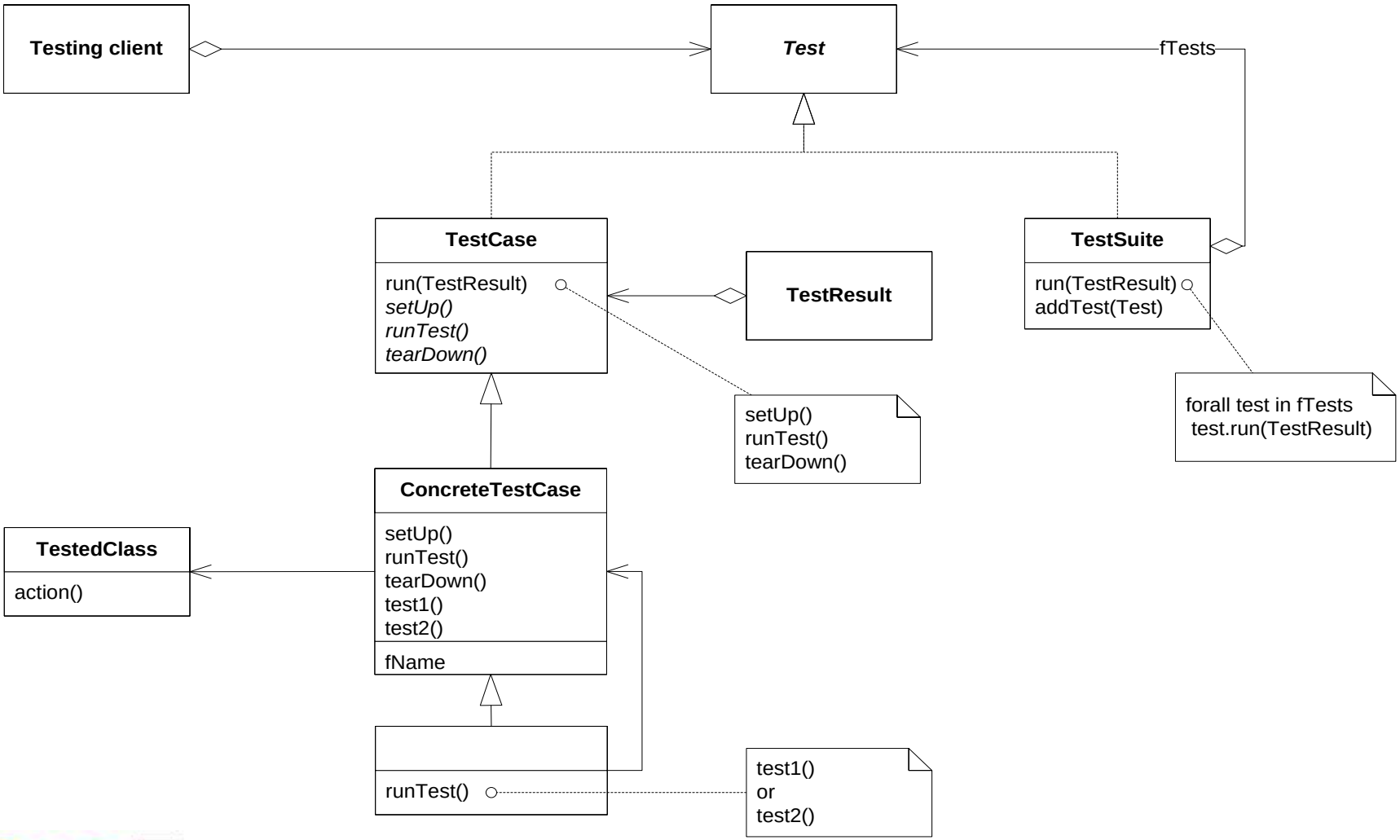
- Combine many test cases in a test suite:

```
public class AllTests extends TestSuite {

public AllTests(String name) {
   super(name);
}

public static TestSuite suite() {
   TestSuite suite = new TestSuite();
   suite.addTestSuite(StackTester.class);
   suite.addTestSuite(AnotherTester.class);
}
```

Organize The Tests

- Create test cases in the same package as the code under test
- For each Java package in your application, define a TestSuite class that contains all the tests for validating the code in the package
- Define similar TestSuite classes that create higher-level and lower-level test suites in the other packages (and sub-packages) of the application
- Make sure your build process includes the compilation of all tests

JUnit framework



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

Example: Counter class

- We write the test methods before we write the code
  - This has the advantages described earlier
  - Depending on the JUnit tool we use, we *may* have to create the class first, and we *may* have to populate it with stubs (methods with empty bodies)
- 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 extends
junit.framework.TestCase {
    Counter counter1;

    public CounterTest() { } // default ctor

    protected void setUp() {
        // creates a (simple) test fixture
        counter1 = new Counter();
    }
    protected void tearDown() { }
        // no resources to release
```

JUnit tests for Counter…

```
    public void testIncrement() {
        assertTrue(counter1.increment()== 1);
        assertTrue(counter1.increment()== 2);
     }
    public void testDecrement() {
        assertTrue(counter1.decrement()==-1);
    }
}  // End from last slide
```

The Counter class itself

```
public class Counter {
    int count = 0;
    public int increment() {
        return ++count;
    }
    public int decrement() {
        return --count;
    }
        public int getCount() {
            return count;
        } }
```

Why JUnit

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

Summary: elements of JUnit

- assert*()
  - Comparison functions
- TestCase
  - Class containing a set of tests
  - One per each class in production code
  - Many tests for each method of a class in production code
- TestSuite
  - Class containing a sequence of TestCase

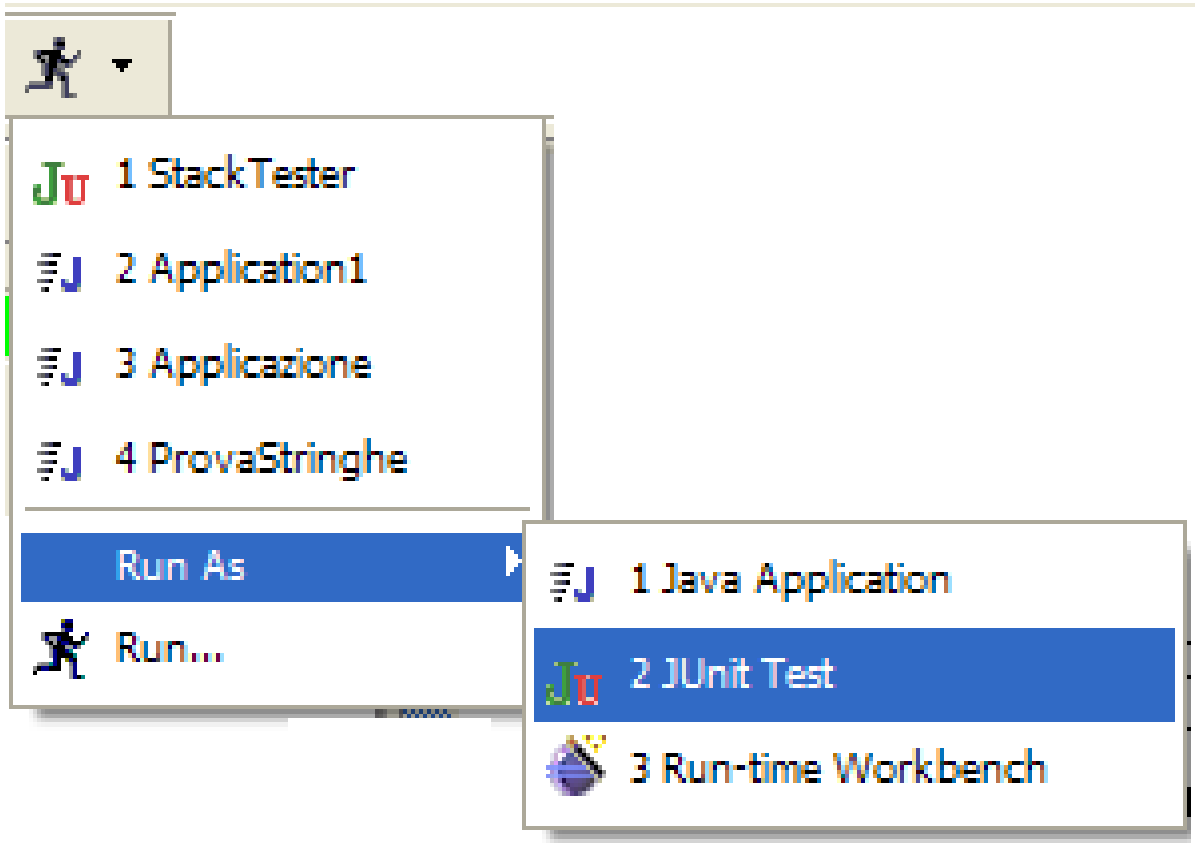
ECLIPSE JUNIT PLUG-IN

Junit in Eclipse – Setup

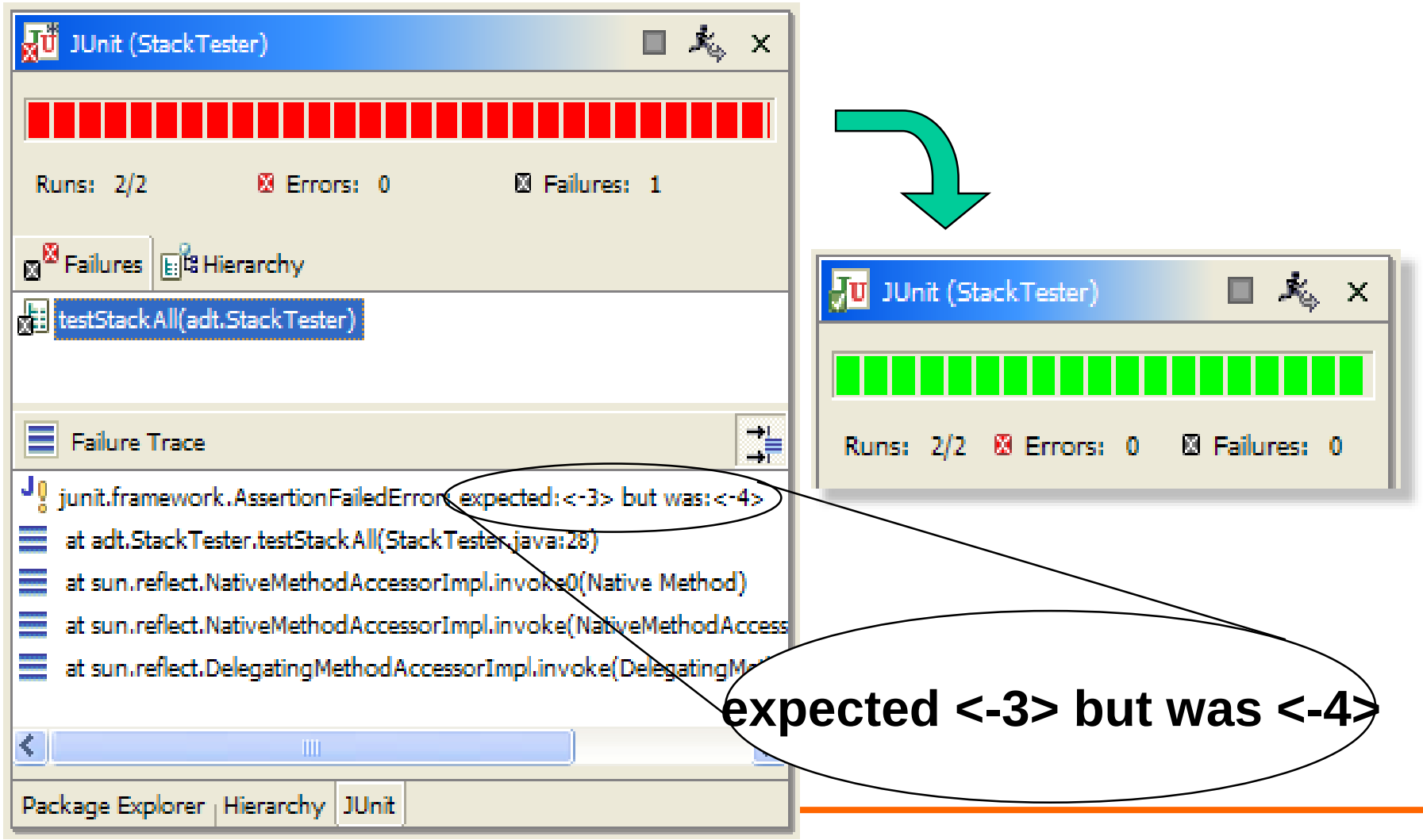
- In Eclipse
- open project's property window
- java build path
- libraries
- Add external jar
  - add org.junit

Junit in Eclipse – Run as JUnit Test

- Run
- Run As..
- Junit Test



Red / Green Bar

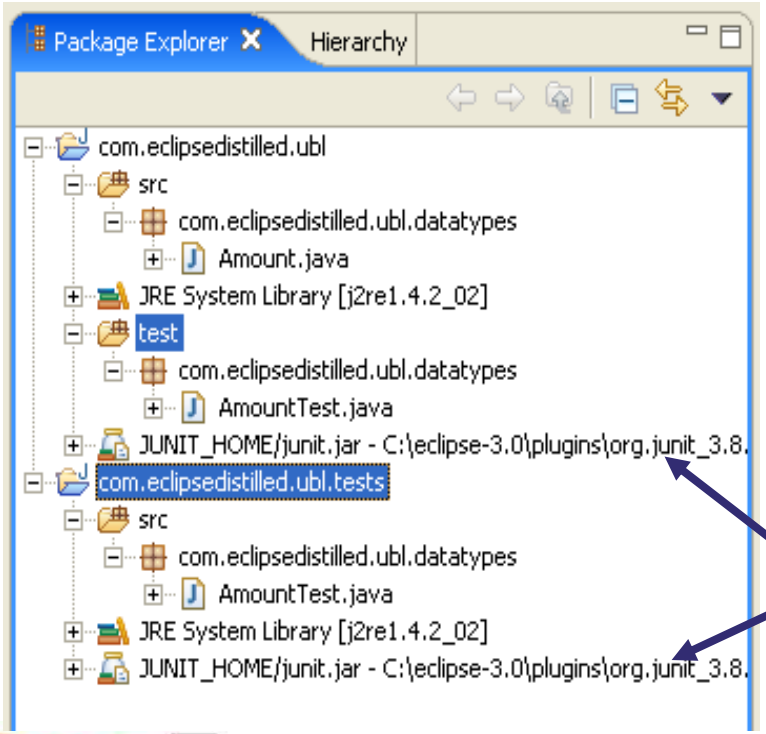


Unit Testing New Code

Unit testing can support several general strategies for validating the behavior of your software. When developing new code, write tests that:

- Specify the intended outcome of code yet to be written; then write code until the tests pass. This is the ideal test-first strategy advocated by agile development principles.
- Specify the correct operation for bug reports; then modify the code until these bug-fix tests pass.

Organizing Unit Tests in Eclipse

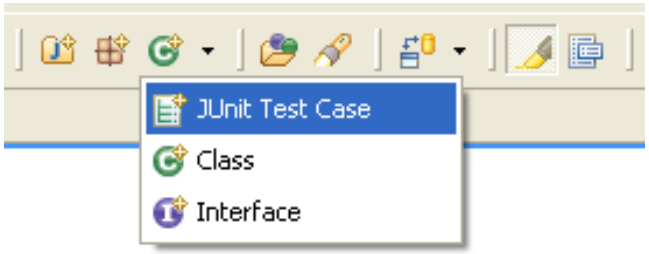


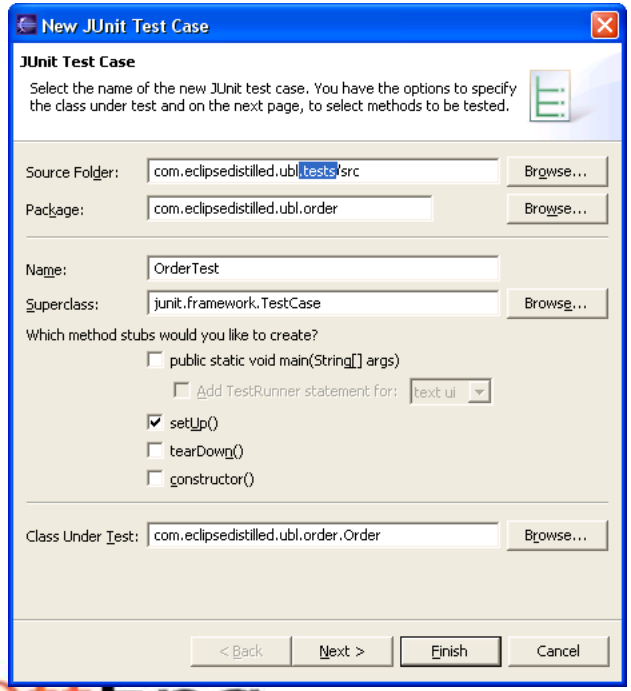
- Second source folder
  - Do not put JUnit tests in the same source folder as your project code
  - A second source folder allows clear separation
- Separate Project
  - Preferred configuration
  - No unit test libraries are added to your primary project classpath

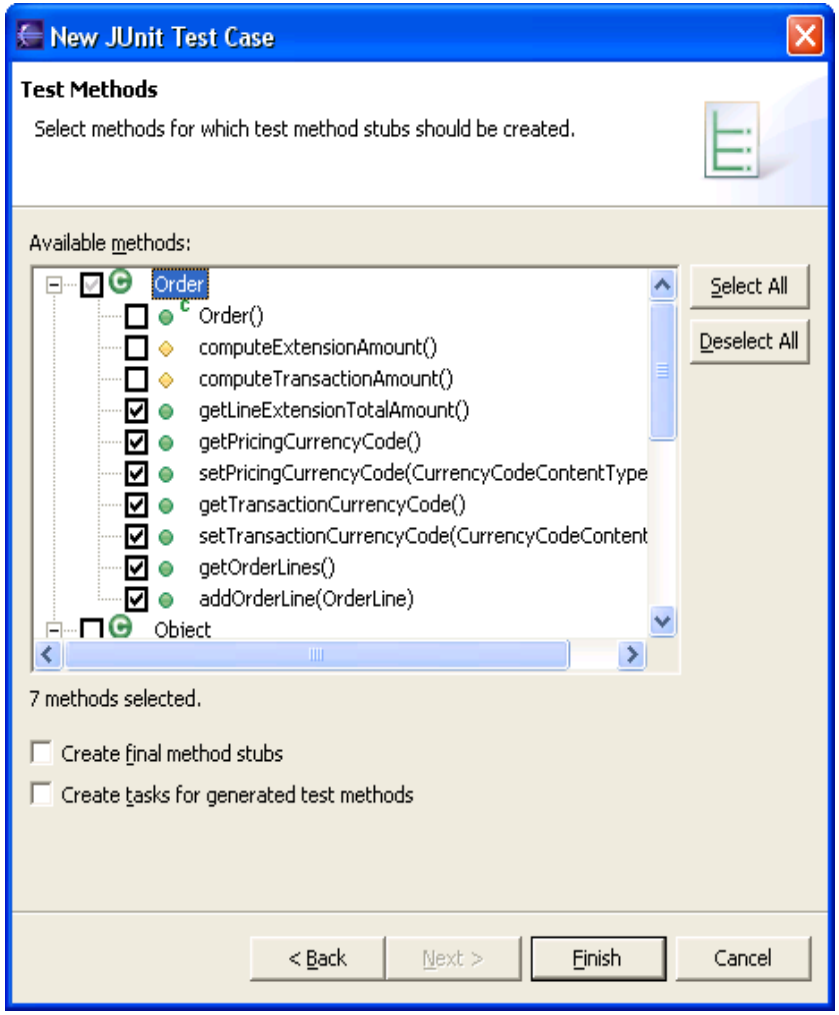
Add junit.jar to the project classpath

SoftEng
http://softeng.polito.it

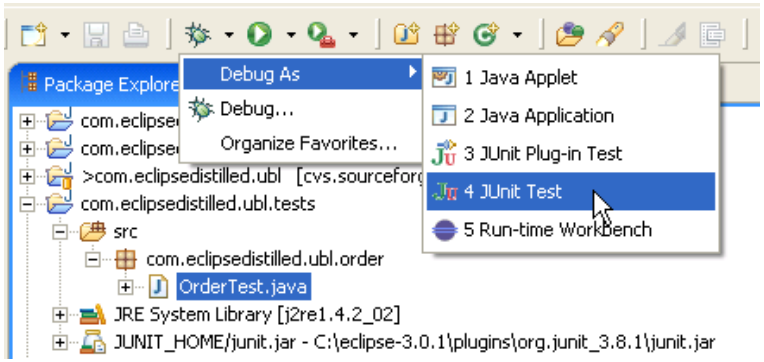
JUnit Test Case Wizard

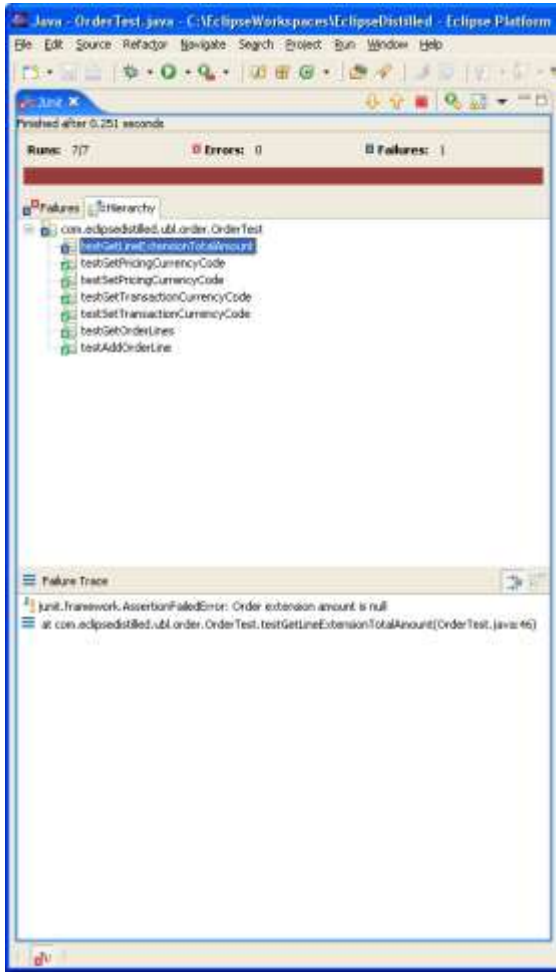






Running JUnit in Eclipse





- Configure JUnit View as a **Fast View**

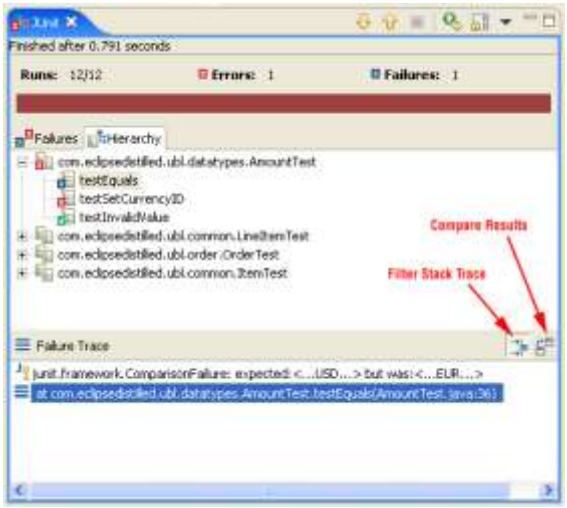
  Test results summary on status bar icon

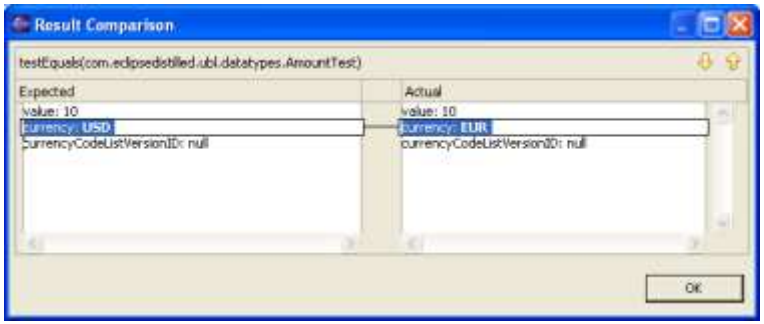
- Double click error trace to go to test/app source

- **Failure**: JUnit assertion or fail was invoked
- **Error**: Unexpected error, e.g. NullPointerException

SoftEng
http://softeng.polito.it

Additional Controls in JUnit View





- Filter Stack Trace
  - remove stack trace entries related to JUnit infrastructure
  - filter is configurable in preferences

    Java > JUnit
- Compare Results
  - available when `assertEquals()` is used to compare two string value

…use JUnit

Keep the bar green to keep the code clean…