Unit Testing with JUnit

James Brucker

Many Levels of Software Testing

Software testing is critical!

❑ Testing of specification

❑ Unit Testing

❑ Integration Testing

❑ Acceptance Testing

❑ Usability Testing

❑ ...

Why Test?

1. Saves time!

- *Testing is faster than fixing "bugs".*

2. Testing finds more errors than debugging.

3. Prevent re-introduction of old faults (regression errors).

Programmers often recreate an error (that was already fixed) when they modify code.

4. Validate software: does it match the specification?

Psychological Advantages

- *Keeps you **focused** on current tasks.*

  - ***Test-driven development:***

    *write the tests **first** ... what the code should do.*

    Then write code that passes the tests

- *Increase **satisfaction**.*
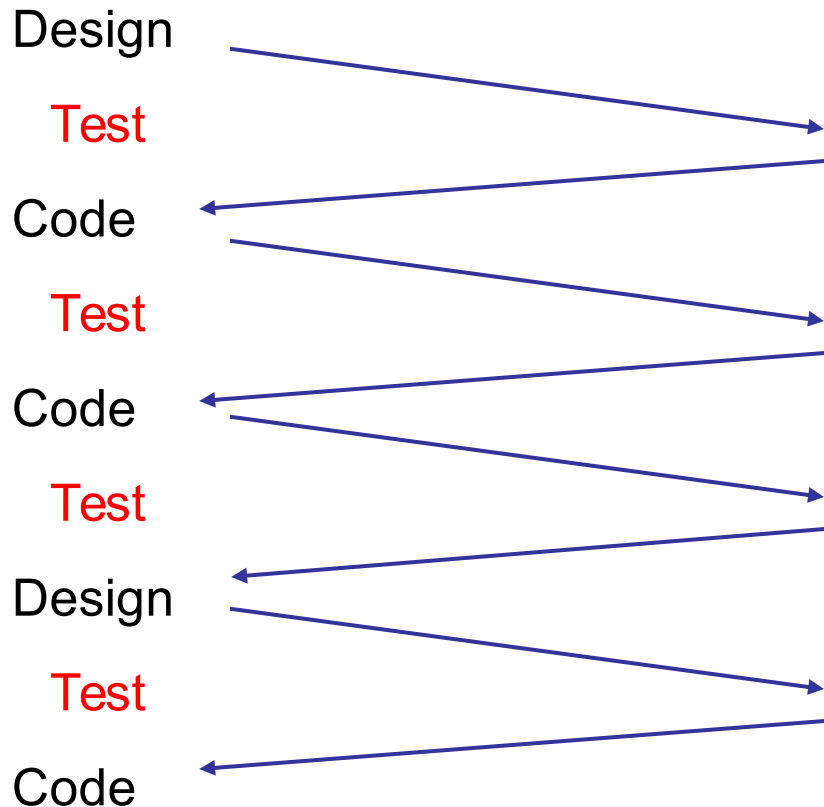
- ***Confidence** to make changes.*

Testing is part of development

Agile Development philosophy

- *Test early.*

- *Test continually!*

Design

Test

Code

Test

Code

Test

Design

Test

Code

When To Test?

*- Test **while** you are writing the source code*

*- **Retest** whenever you modify the source code*

The Cost of Fixing "faults"

Discover & fix a defect early is much cheaper (100X) than to fix it after code is integrated.
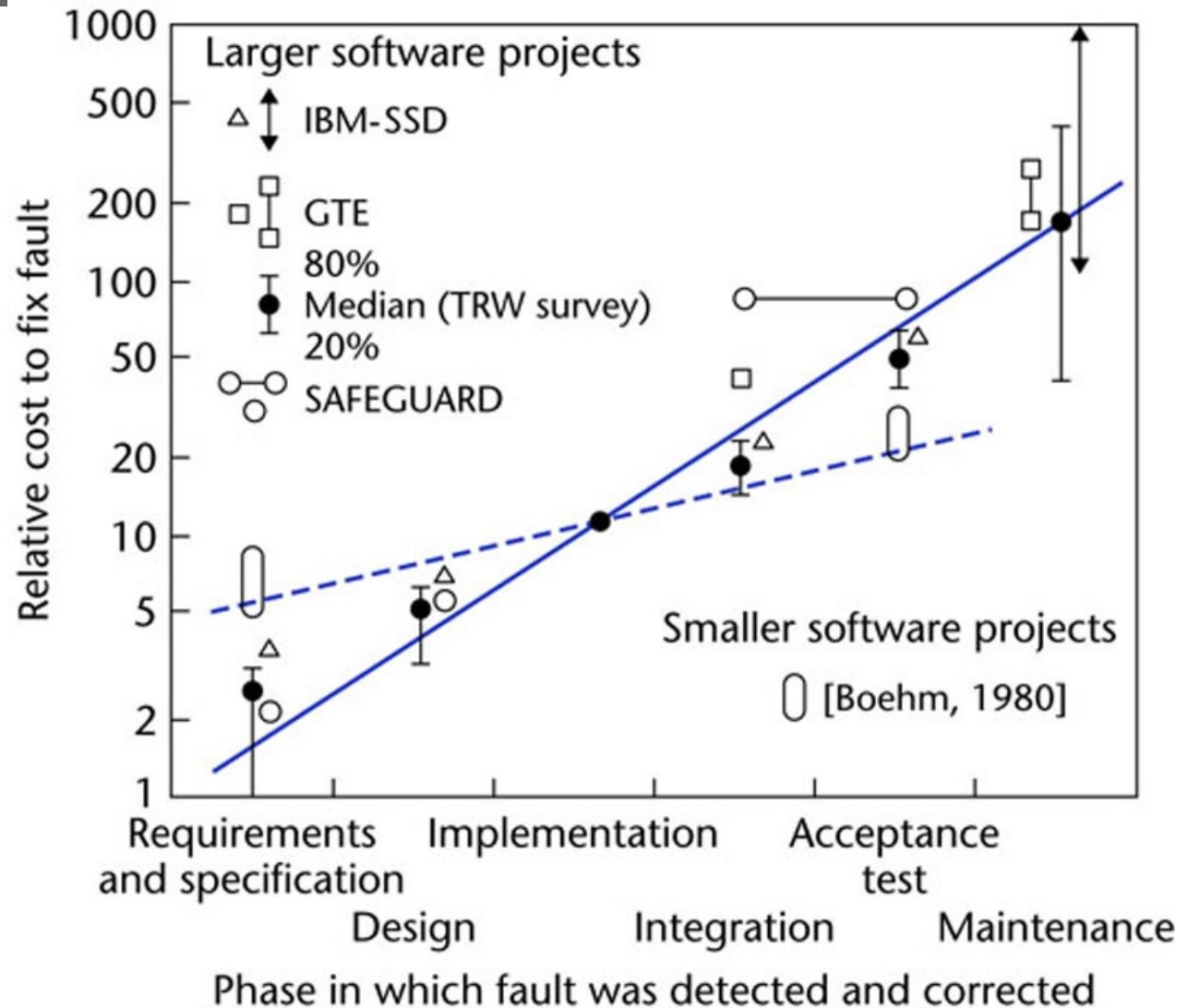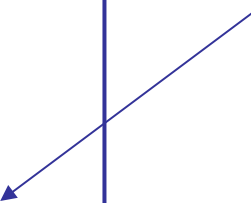

Figure 1.5

An Example

- A Coin Purse holds coins.

- It has a capacity that is fixed when the purse is created.
  - capacity is the number of <u>coins</u> (any type) that purse can hold

- You can insert and withdraw coins within capacity.

Purse
+ **Purse(capacity)**
+ **getCapacity( ) : int**
+ **getBalance( ) : int**
+ **isFull( ) : boolean**
+ **insert(Coin) : boolean**
+ **withdraw(amount) : Coin[*]**
+ **count( ) : int**

`insert` returns true if coin is inserted.

Test by writing Java code

```java
Purse purse = new Purse(2); // can hold 2 coins
// test insert and isFull methods
boolean result = purse.insert( new Coin( 10 ) );
if ( ! result ) out.println("ERROR: insert failed");
if ( purse.isFull() )
    out.println("ERROR: full");
balance = purse.getBalance( );
if ( balance != 10 )
    out.println("ERROR: balance is wrong" );
if ( purse.withdraw(5) != null )
    out.println("ERROR: withdraw is wrong");
if ( purse.withdraw(10) == null )
    out.println("ERROR: couldn't withdraw 10 Baht");
```

Test by writing Java code (2)

```java
// add 2 coins and test again
result = purse.insert( new Coin( 10 ) );
result = result && purse.insert( new Coin( 1 ) );
if ( ! result ) out.println("ERROR: insert failed");
if ( ! purse.isFull() )
   out.println("ERROR: should be full");
balance = purse.getBalance( );
if ( balance != 11 )
   out.println("ERROR: balance is wrong" );
if ( purse.withdraw(2) != null )
   out.println("ERROR: withdraw is wrong");
if ( purse.withdraw(11) == null )
   out.println("ERROR: couldn't withdraw 11 Baht");
```

Too Much Coding!

- A lot of code for a simple test.

- Would you write these tests?
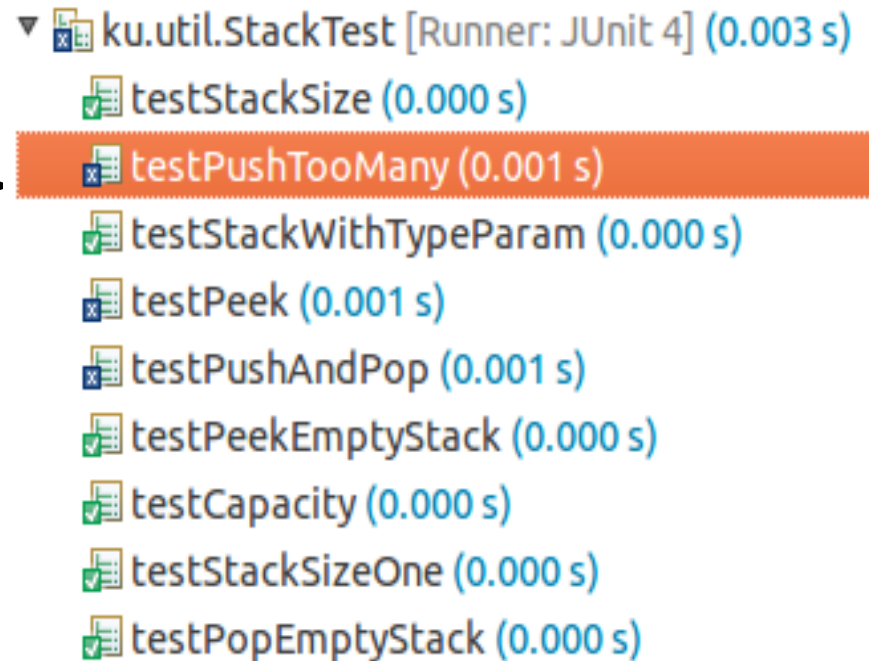
  No way.

Insight

The test code is mostly redundant "boiler plate" code.

□ *Automate the redundant code.*

□ *Create a tool to perform tests and manage output.*

JUnit does it!

```java
public class StackTest {
    @Test
    public void testStackSize( ) {
        ...
    }
    @Test
    public void testPeek() {
        ...
    }
    @Test
    public void testPushAndPop() {
        ...
    }
```

Runs: 9/9   ☒ Errors: 0   ☒ Failures: 3

▼ 🔳 ku.util.StackTest [Runner: JUnit 4] (0.003 s)
   📋 testStackSize (0.000 s)
   📋 testPushTooMany (0.001 s)
   📋 testStackWithTypeParam (0.000 s)
   📋 testPeek (0.001 s)
   📋 testPushAndPop (0.001 s)
   📋 testPeekEmptyStack (0.000 s)
   📋 testCapacity (0.000 s)
   📋 testStackSizeOne (0.000 s)
   📋 testPopEmptyStack (0.000 s)

Structure of a Test Class

Class in Your Project

Test Class

```java
public class Purse {
    /** create coin purse */
    public Purse(int capacity) {
        ...
    }
    /** insert coins */
    public boolean insert(
        int tens, int fives, int ones){
        ...
    }
    /** get value of purse */
    public int getBalance( ) {
        ...
    }
}
```

```java
public class PurseTest {
    @Test
    public void testPurse( ) {
        // test the constructor
    }
    @Test
    public void testInsert() {
        // test insert method
    }
    @Test
    public void testGetBalance( ) {
        // test balance method
    }
}
```

Using JUnit for Testing

- JUnit makes it *easy* to write test cases

- JUnit automatically runs your tests

- JUnit reports failures

You can also use JUnit to...

- test for Exceptions

- limit the time that a test can run

Example: test the Math class

```java
import org.junit.*;
public MathTest  {
    @Test                           // @Test identifies a test method
    public void testMax( ) {        // any public void method name
        Assert.assertEquals( 7,  Math.max(3, 7) );
        Assert.assertEquals( 14, Math.max(14, -15) );
    }
```

JUnit test methods are in the `Assert` class.

assertEquals(expected, actual)

assertTrue(expression)

assertSame(obja, objb)

expected

result

actual

result

More Test Examples

```java
@Test
public void testMath( ) {

    double TOL = 1.0E-8; // tolerance

     Assert.assertEquals(

                1.414213562, Math.sqrt(2), TOL );

     Assert.assertEquals( "Rounding error!",

                8.0, Math.round(7.50000001), TOL );

 }
```

tolerance for floating point comparison

message to print if test fails (optional)

expected

result

Example: test the Purse constructor

```java
import org.junit.*;
public PurseTest  {
    /** test the constructor */
    @Test
    public void testPurseConstructor( ) {
        Purse p = new Purse( 10 );     // capacity 10
        Assert.assertEquals("Purse should be empty", 0, p.count() );
        Assert.assertEquals("Capacity should be 10", 10, p.getCapacity() );
        Assert.assertFalse(  p.isFull() );
    }
```

test insertCoin method

```java
import org.junit.*;
import static org.junit.Assert.* ; // import names of static methods
public PurseTest  {                 // into the name space
    @Test
    public void insertCoins() {
        Purse purse = new Purse( 2 );
        assertTrue("Couldn't add coin!", p.insertCoin( new Coin( 2 ) );
        assertEquals( 2, purse.getBalance( ) );
        assertFalse( purse.isFull( ) );
        assertTrue("Couldn't add note!", p.insertCoin(new BankNote(50) );
        assertEquals( 52, purse.getBalance( ) );
        assertTrue( purse.isFull( ) );
```

What can you Assert ?

JUnit Assert class provides many `assert` methods

```
Assert.assertTrue( 2*2 == 4 );

Assert.assertFalse( "Stupid Slogan", 1+1 == 3 );

Assert.assertEquals( new Double(2), new Double(2));

Assert.assertNotEquals( 1, 2 );

Assert.assertSame( "Yes", "Yes" ); // same object

Assert.assertNotSame("Yes", new String("Yes") );

double[] a = { 1, 2, 3 };

double[] b = Arrays.copyOf( a, 3 );

Assert.assertArrayEquals( a, b );

Assert.assertThat( patternMatcher, actualValue );
```

Use `import static Assert.*`

Tests almost always use static Assert methods:

```
    @Test
    public void testInsert( ) {
        Assert.assertTrue( 1+1 == 2 );
```

Use "`import static`" to reduce typing:

```
import static org.junit.Assert.*;
public class StupidTest {
    @Test
    public void testInsert( ) {
        assertTrue( 1+1 == 2 );
```

Test Methods are *Overloaded*

Assert.assertEquals is overloaded (many param. types)

```
assertEquals( expected, actual );
assertEquals( "Error message", expected, actual );
```

can be any primitive data type or String or Object

For floating point, you must specify a *tolerance* for values
  to be considered "equal".

```java
final static double TOL = 1.0E-8; // don't copy this
@Test
public void testPythagorus() {
    assertEquals( 5.0, Math.hypot(3.0,4.0), TOL );
}
```

AssertEquals for Floating Point

assertEquals for `float` and `double` require a **tolerance**

as allowance for limit on floating point accuracy.

```java
final static double TOL = 1.0E-8; // don't copy this
@Test
public void testPythagorus() {
    assertEquals( 5.0, Math.hypot(3.0,4.0), TOL );
}
@Test
public void testSquareRoot( ) {
    assertEquals( 1.41421356, Math.sqrt(2), TOL );
```

Expected Result

Actual Result

Tolerance for comparison

Running JUnit 4

1. Use Eclipse, Netbeans, or BlueJ (easiest)

   Eclipse, Netbeans, and BlueJ include JUnit.

2. Run JUnit from command line.

```
CLASSPATH=c:/lib/junit4.1/junit-4.1.jar;.

java org.junit.runner.JUnitCore PurseTest
```

3. Use Ant (automatic build and test tool)

JUnit 4 uses Annotations

- JUnit 4 uses annotations to identify methods

  @Test   a test method

  @Before a method to run before each test

  @After   a method to run after each test

  @BeforeClass   method to run one time before testing starts

Contents of a Test Class (JUnit 4.x)

```
import org.junit.*;                  // JUnit 4 uses package org.junit
import static org.junit.Assert.*;    // import static names from "Assert"
public PurseTest {                   // don't extend TestCase
    /** test the constructor */
    @Test                            // use @Test annotation
    public void testPurse( ) {       // any void method - no parameters
        Purse p = new Purse( 10 ); // capacity 10
        assertEquals("New Purse should be empty", 0, p.count() );
        assertEquals("Capacity should be 10", 10, p.capacity() );
    }
    /**test insert coins */
    @Test
    public void testInsert() {
        Purse p = new Purse( 10 ); // capacity 10
        boolean result = p.insert( new Coin( 5 ) );
        assertTrue("Couldn't insert coins!", result );
```

Before and After methods

@Before indicates a method to run before each test

@After indicates a method to run after each test

```
public PurseTest  {
    private Purse purse;
    @Before
    public void runBeforeTest( ) { purse = new Purse( 10 ); }
    @After
    public void runAfterTest( ) { purse = null; }

    @Test  public void testPurse( ) {
        Assert.assertEquals( 0, purse.count() );
        Assert.assertEquals( 10, purse.capacity() );
    }
```

Using @Before and @After

You want a *clean test environment* for each test.

This is called a "test fixture". Use @Before to initialize a test fixture. Use @After to clean up.

```
private File file; // fixture for tests writing a local file

@Before
public void setUp( ) {
    file = new File( "/tmp/tempfile" );
}
@After
public void tearDown( ) {
    if ( file.exists() ) file.delete();
}
```

@Before (setUp) and @After (tearDown)

- **@Before** - method that is run before every test case.

   setUp( ) is the traditional name.

- **@After** - method that is run after every test case.

   tearDown( ) is the traditional name.

setUp( )
↓
testFun1( )
↓
tearDown( )

one test case

setUp( )
↓
testFun2( )
↓
tearDown( )

another test
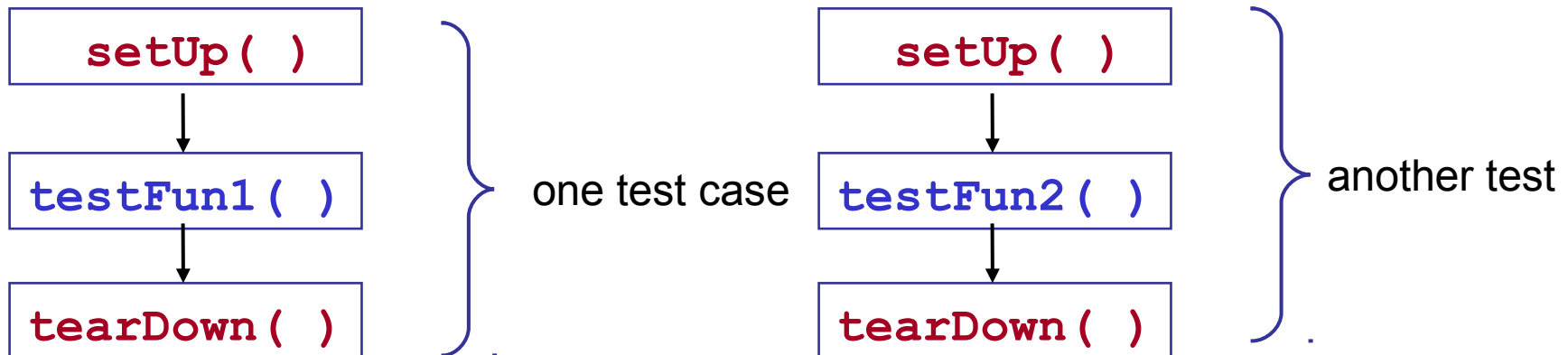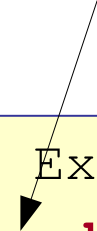
Testing for an Exception

- you can indicate that a test *should* throw an exception.

List should throw IndexOutOfBoundsException if you go beyond the end of the list.

```java
// this test should throw an Exception
@Test( expected=IndexOutOfBoundsException.class )
public void testIndexTooLarge() {
  List list = new ArrayList();
  list.add( "foo" );
  list.get( 1 ); // no such element!
}
```

Valid Arguments

- If an argument is invalid, Coin throws InvalidArgument Exception

```
// this test should throw an Exception
@Test( expected=InvalidArgumentException.class )
public void testRejectBadCoins() {
    Coin coin = new Coin(-1);
}
```

Limit the Execution Time

- specify a time limit (milliseconds) for a test

- if time limit is exceeded, the test fails

```java
// this test must finish in less than 500 millisec
@Test( timeout=500 )
public void testWithdraw() {
    // test fixture already created using @Before
    // method, and inserted coins, too
    double balance = purse.getBalance();
    assertNotNull( purse.withdraw( balance-1 ) );
}
```

fail!

- Signal that a test has failed:

```java
@Test
public void testWithdrawStrategy() {
    //TODO write this test
    @fail( "Test not implemented yet" );
}
```

What to Test?

- Test BEHAVIOR not just methods.

- One test may involve several methods.

- May have many tests for the same method.

Designing Tests

- "borderline" cases:

  - a Purse with capacity 0 or 1

  - if capacity is 2, can you insert 1, 2, or 3 coins?

  - can you withdraw 0? can you withdraw -1?

  - can you withdraw *exactly* amount in the purse?

- impossible cases:

  - can you withdraw negative amount?

  - can you withdraw balance+1 ?

  - can you withdraw Double.INFINITY ?

Designing Tests

❑ typical cases

- Purse capacity 5. Insert many different coins.

- When you withdraw, do coins match what you inserted?

Example: Purse

Test **behavior** ... not just methods

- "can I insert same coin twice?"

- "can I withdraw *all* the money?"

- "does withdraw always *exactly match* what I requested?"

Questions about JUnit 4

- Why use:

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

- How do you test if `Math.sin(Math.PI/2)` is 1 ???

- How do you test if a String named `str` is null ???

Using JUnit in BlueJ

1. From "Tools" menu select "Preferences..."

2. Select "Miscellaneous" tab.

3. Select "Show unit testing tools".
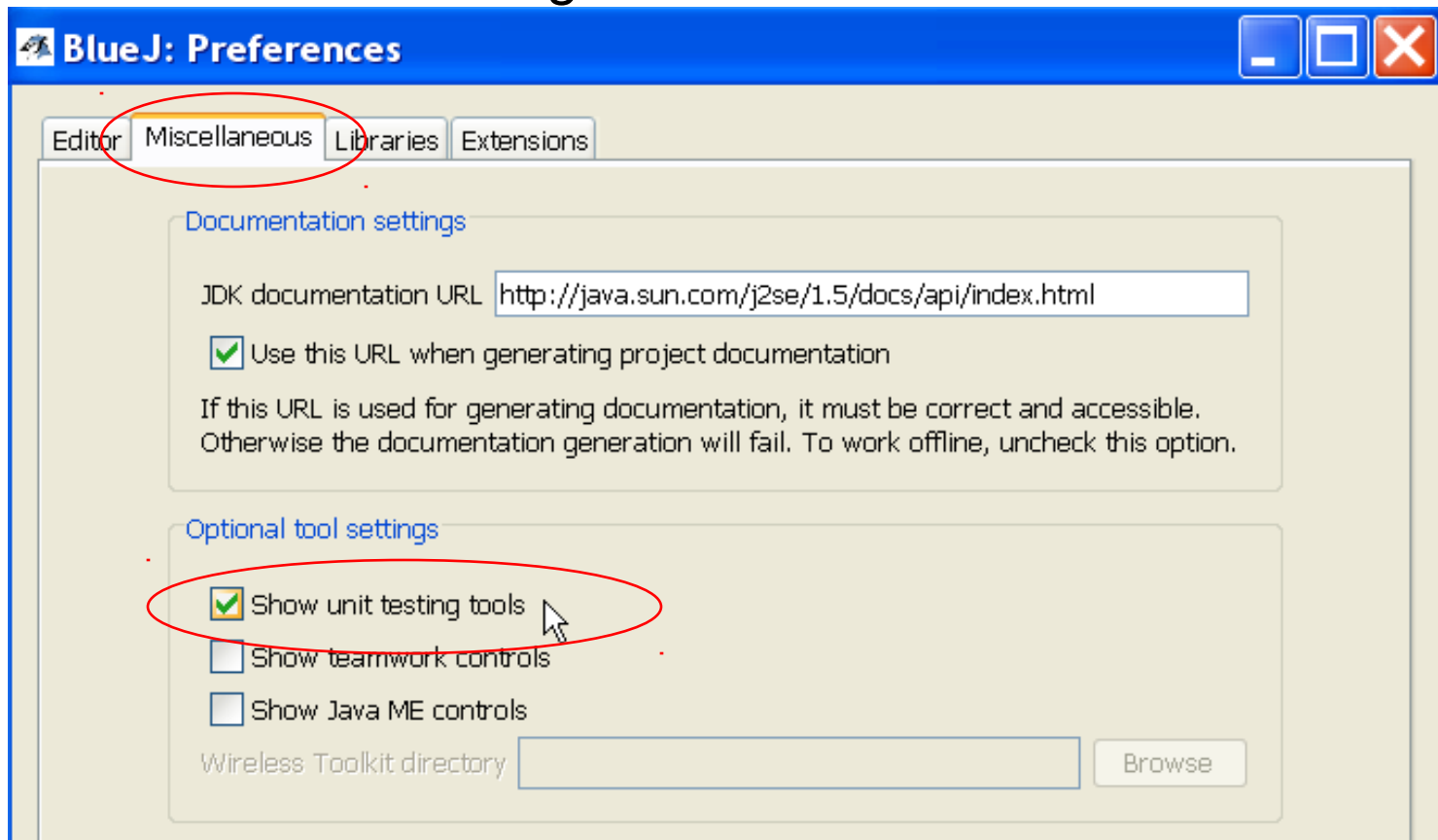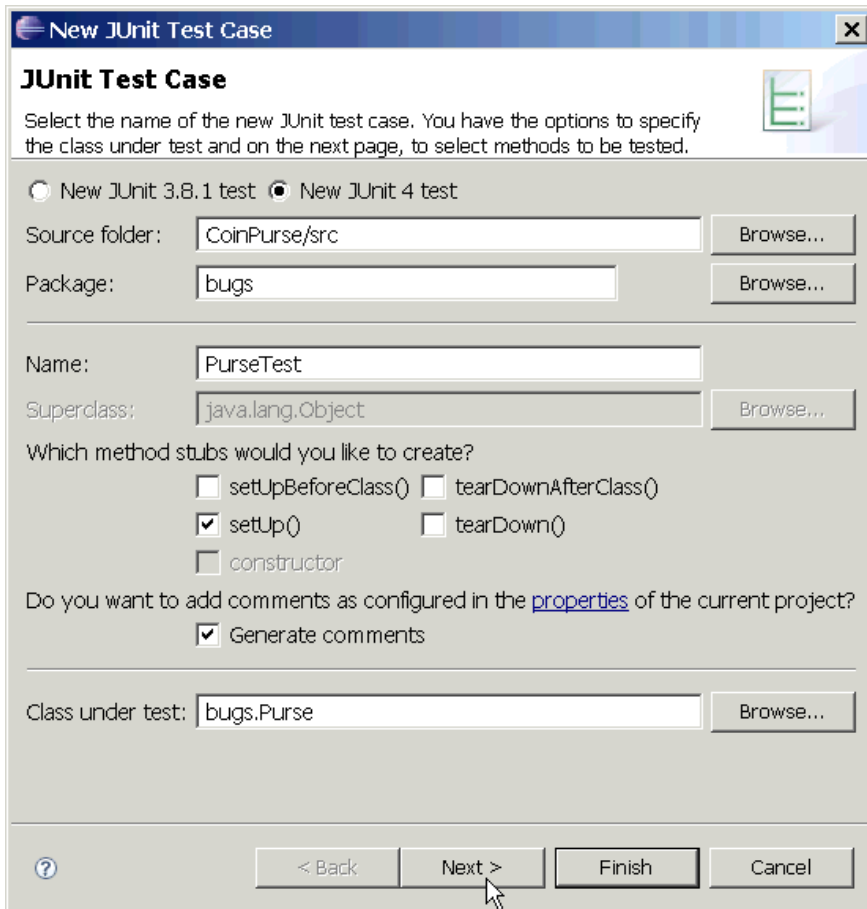
Using JUnit in Eclipse

- Eclipse includes JUnit 3.8 and 4.x libraries
  - you should use Junit 4 on your projects
- eclipse will manage running of tests.
  - *but,* you can write your own test running in the main method
- Select a source file to test and then...
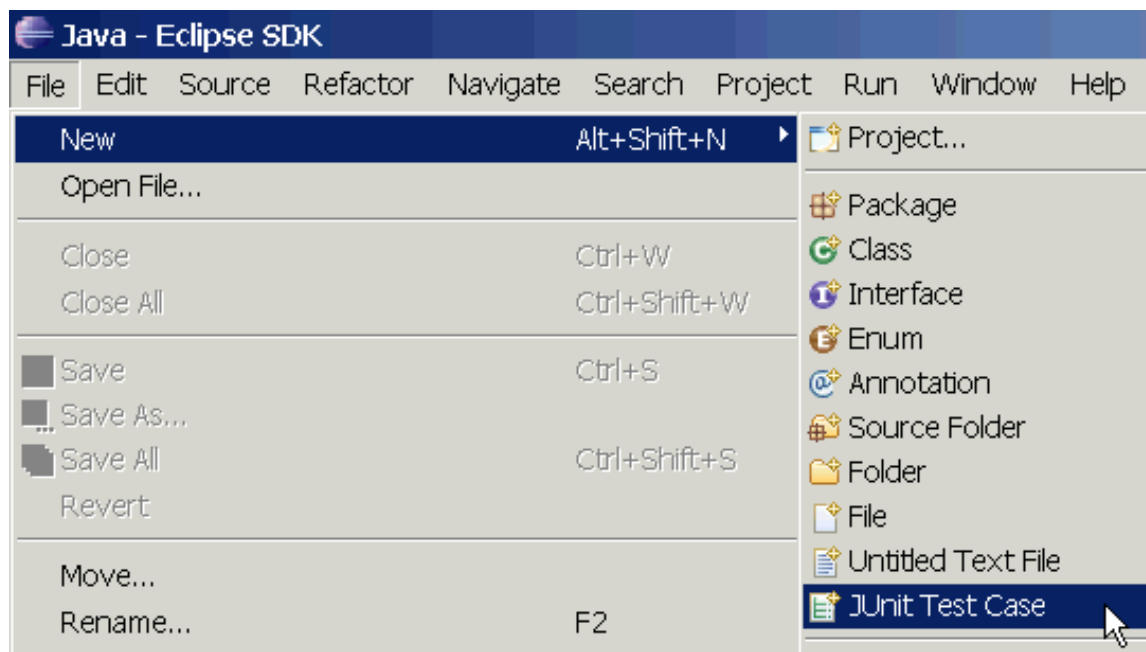
Using JUnit in Eclipse (2)

- Select test options and methods to test.

Using JUnit in Eclipse (3)

```java
/** Test of the Purse class
 * @author James Brucker
 */
public class PurseTest {
  private Purse purse;
  private static final int CAPACITY = 10;
  /** create a new purse before each test */
  @Before
  public void setUp() throws Exception {
    purse = new Purse( CAPACITY );
  }
  @Test
  public void testCapacity() {
    assertEquals("capacity wrong",
          CAPACITY, purse.capacity());
  }
```

Write your test cases. Eclipse can't help much with this.

Run JUnit in Eclipse (4)
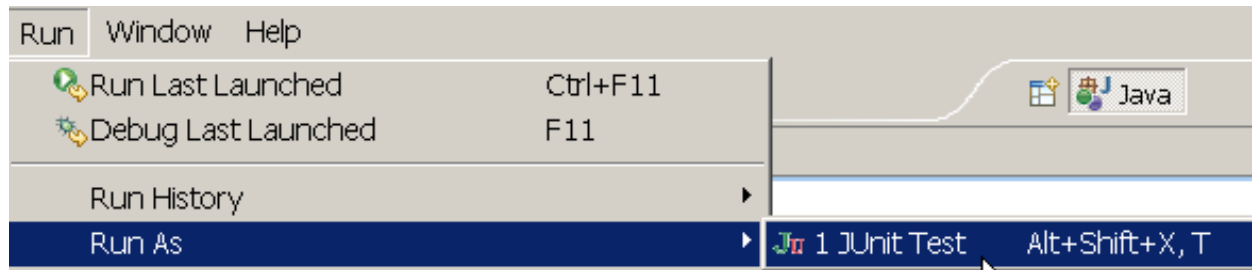
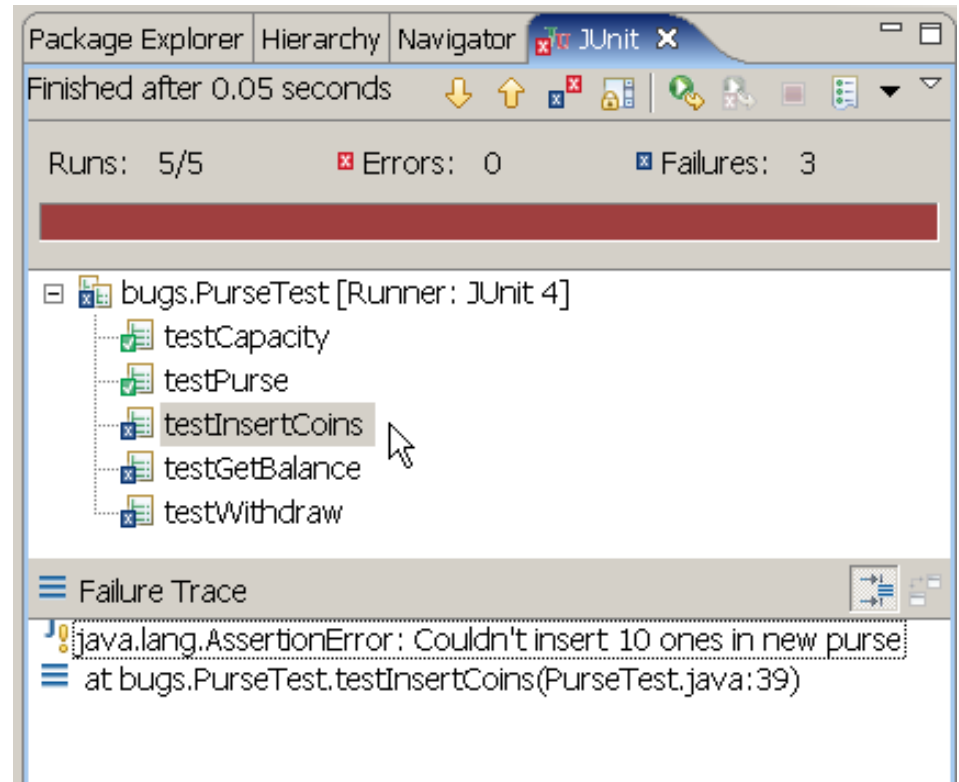Select the JUnit test case file and choose

Run => Run As => JUnit Test

Results appear in a new JUnit tab.

Click on any result for details and to go to the source code.

References

JUnit Home

`http://www.junit.org`

JUnit Software & documentation

`http://www.sf.net/projects/junit`

- Eclipse & Netbeans include Junit, but you still need to install JUnit to get documentation

Quick Starts

JUnit 4 in 60 Seconds

http://www.cavdar.net/2008/07/21/junit-4-in-60-seconds/

JUnit Tutorial by Lars Vogel

includes how to use JUnit in Eclipse.

http://www.vogella.de/articles/JUnit/article.html

JUnit 4 in 10 Minutes

on JUnit web site

Other Software for Testing

TestNG - a better JUnit

`http://www.testng.org`

`NUnit` - Unit testing for .Net Applications

`http://www.nunit.org`

JUnit 3.x

JUnit 3.x is *really old*.

But existing software still uses JUnit 3.x, so it is useful to know how to <u>read</u> JUnit 3 tests.

For new code, use the current version of JUnit.

Structure of a JUnit 4 Test Class

```java
import org.junit.*;              // package org.junit
import static org.junit.Assert.*;
public PurseTest  {             // don't extend TestCase
    Purse purse;
    /**test insert coins */

    @Test                       // use @Test annotation for tests
    public void testInsertCoins() {    // any method name is OK
        Purse p = new Purse( 1 );
        boolean result = p.insertCoin( new Coin( 5 ) );
        assertTrue("Couldn't insert coins!", result );
        assertFalse( p.insertCoin( new Coin(1) ); // should be full
        assertEquals( 5.0, p.getBalance( ), 0.001 );
    }
    @Before
    public void initialize( ) {            // any method name is OK
        purse = new Purse( 10 ); // capacity 10
    }
```

Structure of a JUnit 3 Test Class

```
import junit.framework.*;      // package junit.framework
import static junit.framework.Assert.*;
public PurseTest  extends TestCase {  // must extend TestCase
    Purse purse;

                                // No annotations

    public void testInsertCoins() {    // names must begin with "test"
        Purse p = new Purse( 1 );
        boolean result = p.insertCoin( new Coin( 5 ) );
        assertTrue("Couldn't insert coins!", result );
        assertFalse( p.insertCoin( new Coin(1) );
        assertEquals( 5.0, p.getBalance(), 0.001 );
    }

                                // no @Before annotation
    protected void setUp( ) {            // setUp method must use this name
        purse = new Purse( 10 ); // capacity 10
    }
```

Key Points in Using JUnit 3.x

1. Test class "`extends TestCase`"

2. JUnit package is `junit.framework`

   import junit.framework.*;

3. Import static methods:

   import static junit.framework.Assert.*;

4. Must use the naming convention:

   public void testGetBalance( ) { ... }

   protected void setUp( ) { ... }

   protected void tearDown( ) { ... }

5. no annotations (@Before, @After, @Test ...)

JUnit 3 Test Suite

□ For JUnit 3.x your need a method & a constructor:
- ■ PurseTest(string) constructor calls super(string)
- ■ suite( ) creates a test suite

```
import junit.framework.*;
public PurseTest extends TestCase {
    public PurseTest( String testmethod ) {
        super( testmethod );
    }
    /** create a test suite automatically */
    public static Test suite( ) {
        TestSuite suite = new TestSuite( PurseTest.class );
        return suite;
    }
}
```

This is standard form of the constructor; just copy it

Compiling and Running Tests

You invoke a JUnit **TestRunner** to run your test suite. JUnit 3.8 provides 3 test runners:

- junit.textui.TestRunner - console test runner

- junit.awtui.TestRunner - graphical using AWT

- junit.swingui.TestRunnger - graphical using Swing

```
> set CLASSPATH = /java/junit3.8.2/junit.jar;.
> javac PurseTest.java
> java junit.swingui.TestRunner PurseTest
```

Name of your test class as arg.

Another Way to Run Tests

Call test runner from your class's main method

- don't need to invoke junit.*.TestRunner on cmd line

```
public PurseTest extends TestCase {
  ...
  public static void main( String [] args ) {
    junit.swingui.TestRunner.run( PurseTest.class );
  }
```

```
> set CLASSPATH = /java/junit3.8.2/junit.jar;.
> javac PurseTest.java
> java PurseTest
```

Name of your test class as arg.

Selecting Tests to Run: `TestSuite`

❑ In the example we created a TestSuite using:

```
public static Test suite( ) {
    TestSuite suite = new TestSuite( PurseTest.class );
    return suite;
}
```

JUnit uses *reflection* to locate all methods named "`test*`".

❑ or can specify only the tests you want to run

```
/** create a custom test suite */
public static Test suite( ) {
    TestSuite suite = new TestSuite(  );
    suite.addTest( new PurseTest( "testPurse" ) );   // test the constructor
    suite.addTest( new PurseTest( "testInsertCoins") ); // insert coins
    return suite;
}
```

only run these test methods

JUnit 3 Adaptor for JUnit 4 test class

❑ You can run JUnit 3 test cases using JUnit 4 ...

```
import org.junit.Test;
import static org.junit.Assert.*;
            // import adaptor for JUnit 3
import junit.framework.JUnit4TestAdaptor;
public PurseTest  {      // don't extend TestCase

    /* JUnit 3 calls suite( ) to get a test suite */
    public static junit.framework.Test suite( ) {
       return new JUnit4TestAdaptor( PurseTest.class );
    }

    @Test
    ... rest of the JUnit 4 tests ...
```

Questions about JUnit 3

- What are the 2 forms of every `assert( )`?

- Why use:

    `import static junit.framework.Assert.*;`

- What is the name of ...

    - the test class for "`class LineItem`" ?

    - your test class `extends` what other class?

    - the test method for the `LineItem` constructor?

    - the test method for the `getItemID()` method?