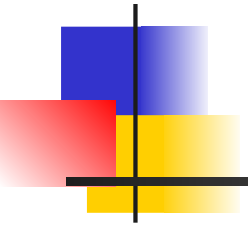


Unit Testing with JUnit

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 (TDD):***

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

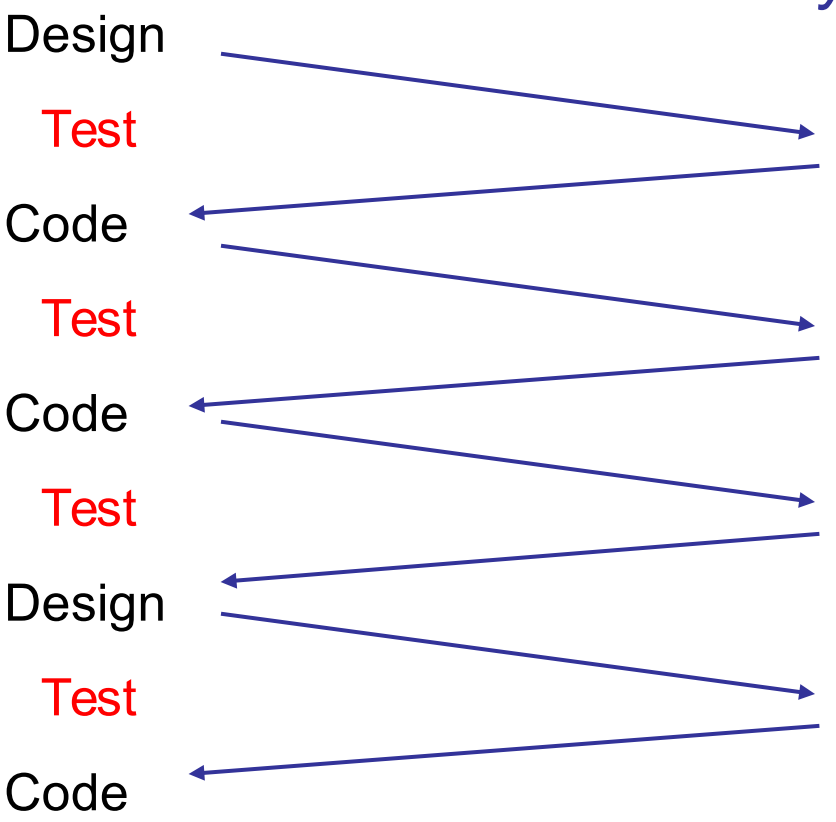
    Then write code that passes the tests

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

Testing is part of development

Agile Development Practices

- *Test early.*
- *Test continually!*



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.

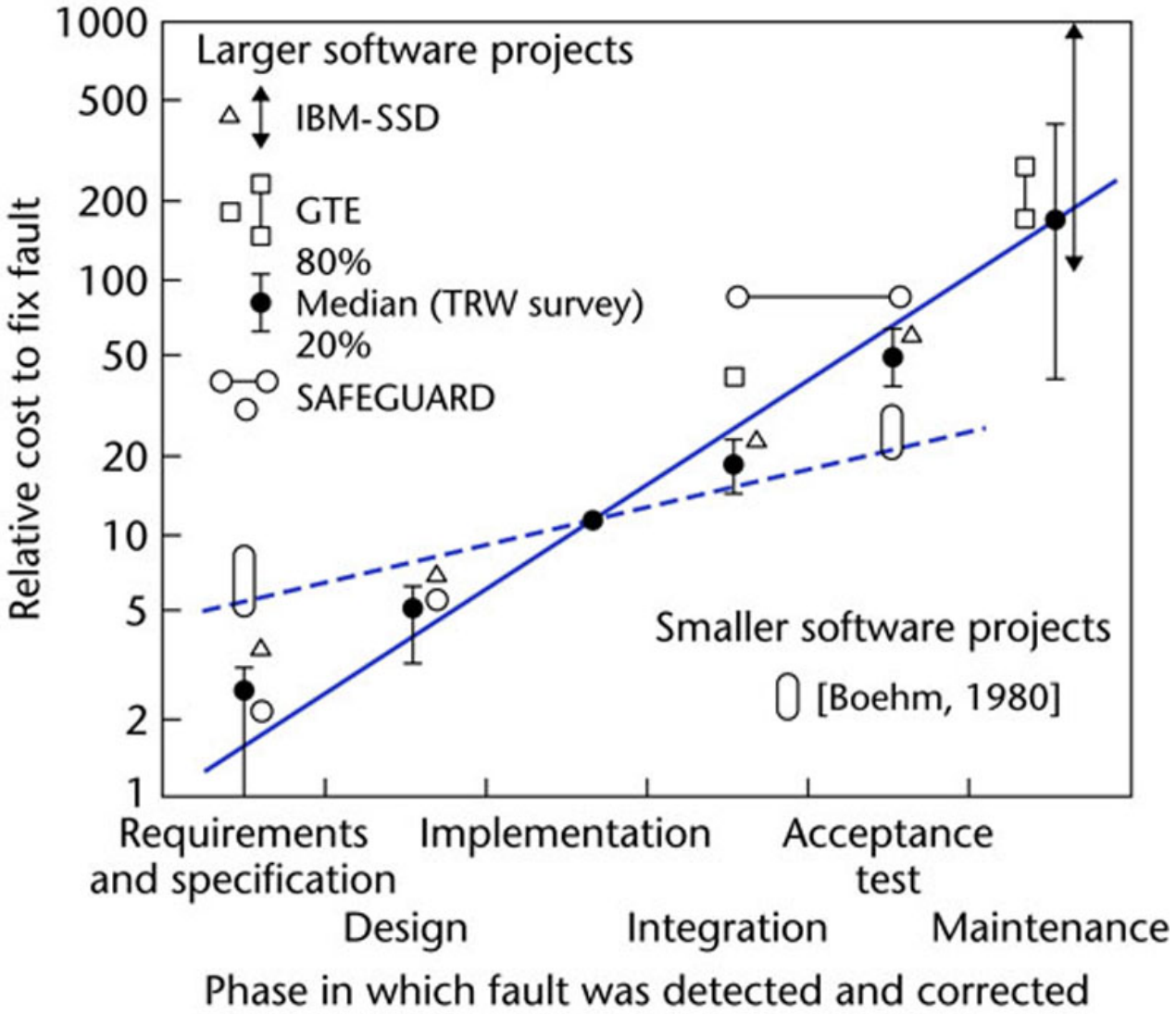


Figure 1.5

What to Test?

In unit testing, we test functions or methods in classes.

How to Test?

We can't test all possible input / output.

- Divide input into categories, sets, or classes.
- Or, discover "rules" that govern different sets of input.
- Test a few samples from each set, category, or class.
  - Test boundary values.
  - Test "typical" values.
  - Test "extreme" values.
  - Test impossible values.
  - Try to make things fail.

Example

- Stack implements common stack data structure.
- Has a fixed capacity and methods shown below.
- Throws StackException if you do something stupid.

Stack<T>
+ Stack(capacity) + capacity( ): int + size( ): int + isEmpty( ): boolean + isFull( ): boolean + push(T): void + pop( ): T + peek( ): T

What to Test?

Border Case:

Stack with capacity 1

1. no elements in stack
   capacity() is 1
   isEmpty() -> true
   isFull() -> false
   size() -> 0
   peek() returns ???
2. push one element on stack
   isEmpty() -> false
   isFull() -> true
   size() -> 1
3. can peek()?
   push one element
   peek() returns element
   stack does not change
4. push element, peek it, then pop
   pop -> returns same object
   test all methods
   idea: a helper method for all
   tests of an empty stack or full stac

Test for Methods

push( )	Hard to test by itself! Need to use peek( ), pop( ), or size( ) to verify something was pushed. 1. Stack of capacity 2. push(x) verify size=1 peek()==x, not full, not empty push(y) verify again pop(y) push(x) - should have 2 items both == x

Test by writing Java code

```
Stack stack = new Stack(1);
// test empty stack behavior
if ( ! stack.isEmpty() )
    out.println("error: should be empty");
if ( stack.isFull() )
   out.println("error: should not be full");
if ( stack.capacity() != 1 )
   out.println("error: capacity incorrect" );
if ( stack.size() != 0 )
   out.println("error: size should be 0" );
if ( stack.peek() != null ) // what should it do?
   out.println("error: peek() should be null");
```

More Java code...

```
// Test a Stack with 1 element
Stack stack = new Stack(1);
Object arg = "push-me";
if ( ! stack.push(arg) )
    out.println("error: should be able to push");
if ( stack.isEmpty() )
    out.println("error: should NOT be empty");
if ( ! stack.isFull() )
   out.println("error: should be full");

                    And so on...
```

Too Slow, too boring

- A lot of redundant code... even for simple tests.

Violates 2 Key Development Practices

1. *don't repeat yourself*

2. *automate repetitive tasks*

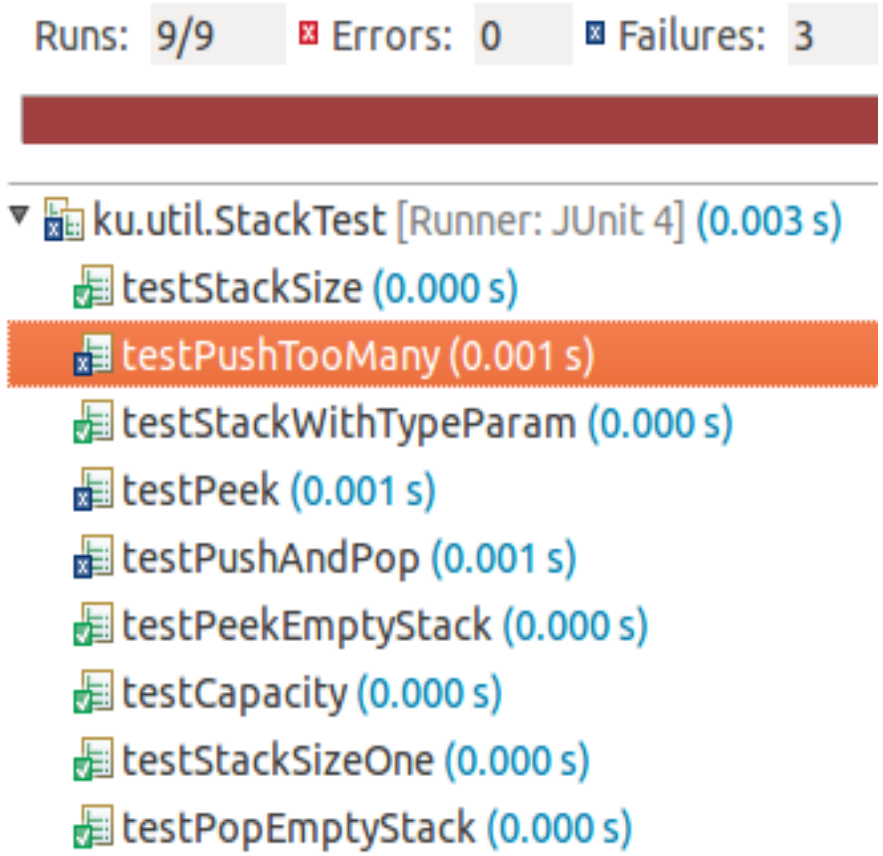
Insight

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

- *Automate the redundant code.*
- *Create an automatic tool to perform tests and manage output.*

JUnit does it

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



Using JUnit for Testing

- makes it *easy* to write test cases
- automatically runs your tests
- reports failures with context information

JUnit can also...

- test for Exceptions
- limit the execution time
- use parameters to vary the test data

Example: test the Math class

```
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

Tests Using Floating Point Values

```
@Test
public void testMath( ) {
    double TOL = 1.0E-8; // tolerance
     Assert.assertEquals(
                  1.414213562, Math.sqrt(2), TOL );
     Assert.assertEquals( "Should round up",
                  8.0, Math.round(7.50000001), TOL );
  }
```

tolerance for floating point comparison

message to print if test fails (optional)

expected result

Unit Testing Vocabulary

Test Suite - collection of unit tests. A test class.

Test Case - test method (`@Test`).

Test Fixture - attributes or local var that is being tested.

Test Runner - code that runs the tests, collects results.

Example: test the Stack constructor

```
import org.junit.*;
import static org.junit.Assert.* ; // import names of all static methods
public StackTest  {
    @Test
    public void testStackConstructor( ) {
        Stack stack = new Stack(5);
        assertEquals("Stack should be empty", 0, stack.size() );
        assertEquals("Capacity should be 5", 5, stack.capacity() );
        assertFalse(  stack.isFull() );
        assertTrue( stack.isEmpty() );
    }
```

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 ArithmeticTest {
    @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

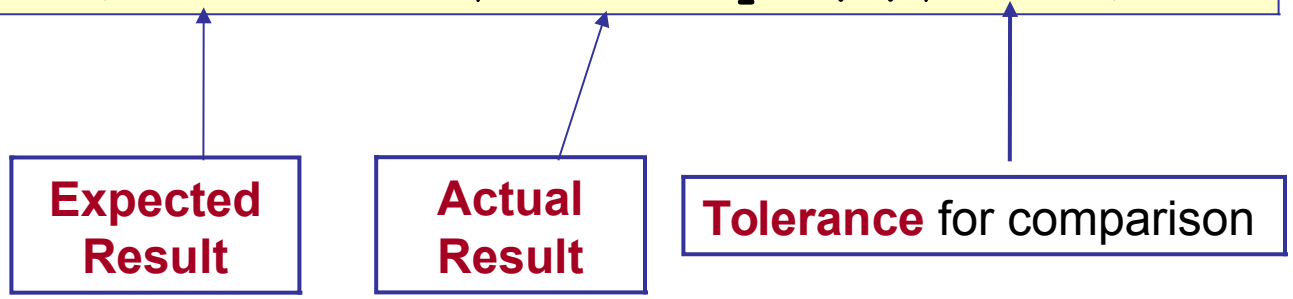
```
// assertSame(a,b)  tests  a == b

assertSame( expected, actual );
```

AssertEquals for Floating Point

assertEquals for `float` and `double` require a **tolerance** as allowance for limit on floating point accuracy.

```
final static double TOL = 1.0E-8; // be careful
@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 );
```



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

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() );
    }
```

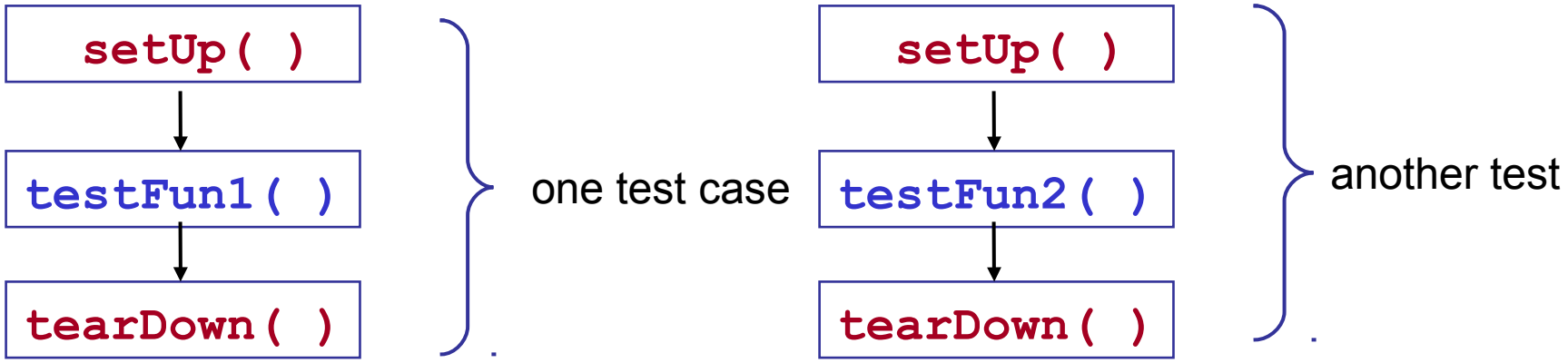
@Before (setUp) and @After (tearDown)

- **`@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.



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.

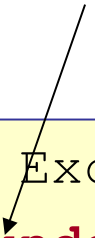
```
private File file; // fixture for tests writing a local file

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

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.



```
// 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!
}
```

Stack Example

- If you pop an empty stack it throws StackException

```
@Test( expected=StackException.class )
public void testPopEmptyStack() {
    Stack stack = new Stack(3);
    Object x = stack.pop( );
}
```

Limit the Execution Time

- specify a time limit (milliseconds) for a test
- if time limit is exceeded, the test fails

```
// 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 ) );
}
```

fail!

- Signal that a test has failed

```
@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 several tests for the same method, each testing different *behavior* or *test cases*.

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? -1?
- can you withdraw balance+1 ?
- can you withdraw Double.INFINITY ?

Designing Tests

typical cases

- Purse capacity 10. Insert many different coins.
- When you withdraw, do coins match what you inserted?

extreme cases

Purse with capacity 9.999,999.

Insert 9,999,999 of 1 Trillion Zimbabwe dollars.

Is balance correct? Can you withdraw everything?

Test *Behavior,* not *methods*

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

Stack:

- can I push until stack is full, then pop each one?
- do peek() and pop() return same object as push-ed?

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 ???

(2 ways)

Fluent JUnit

`Assume.that( actual, matcher )`

`Assume.assumeTrue( stack.isEmpty() )`

- skip a test unless some conditions are true

`Theories` - define more complex test conditions.

See:

https://dzone.com/articles/parameterized-tests-and-theories

Parameterized Tests

We want to test the `isPrime(long)` method.

```
public class MathUtil {
    /**
     * Test if a number is prime.
     * @param n the numbe to test
     */
    public static boolean isPrime(long n) {
        //TODO complete the code
        return false;
    }
}
```

Redundant Tests

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

public class MathUtilTest {
    @Test
    public void testPrimeNumbers( ) {
       long[] primes = [2,3,5,29,163,839,...];
       for(long p: primes)
           assertTrue( MathUtil.isPrime(p) );
    }
    @Test
    public void testNonprimeNumbers() {
       long[] nonprime = [4,99,437,979,3827,...];
       for(long n: nonprime)
           assertFalse( MathUtil.isPrime(n) );
    }
}
```

Parameters for Unit Tests

JUnit **Parameterized Tests**

- set parameters as attributes or method arguments

- you inject (set) values directly to attributes.

- See JUnit docs for "Parameterized Tests"

Maybe Better (and simpler):

JUnitParams: add-on with easier syntax for parameters:

https://github.com/Pragmatists/JUnitParams

Tutorial: https://www.baeldung.com/junit-params

Using Parameter class

```
@RunWith(Parameterized.class)
public class TestMathUtil {
   private long input;
   private boolean expected; // expected result

   public MathUtilTest(long n, boolean result) {
       this.input = n;
       this.expected = result;
    }
   @Test
   public void testPrimeNumber( ) {
     assertEquals(expected,MathUtil.isPrime(input));
   }
}

// ...continued
```

Method defines parameter values

```
@Parameterized.Parameters
public static Collection makeTestValues() {
    // collection of (input,result) pairs
    return Arrays.asList( new Object[][] {
          {2, true},
          {3, true},
          {4, false},
          {19, true},
          {21, false},
          ...
     });
  }
```

Use @Parameterized.Parameters annotation.
Each item in the Collection is injected into the test constructor (MathUtilTest(long,boolean)) before running one test.

More Parameterized Tests

Previous example is too simple to show usefulness.

You can *inject values* directly into attributes (fields) or as method parameters.

JUnitParams

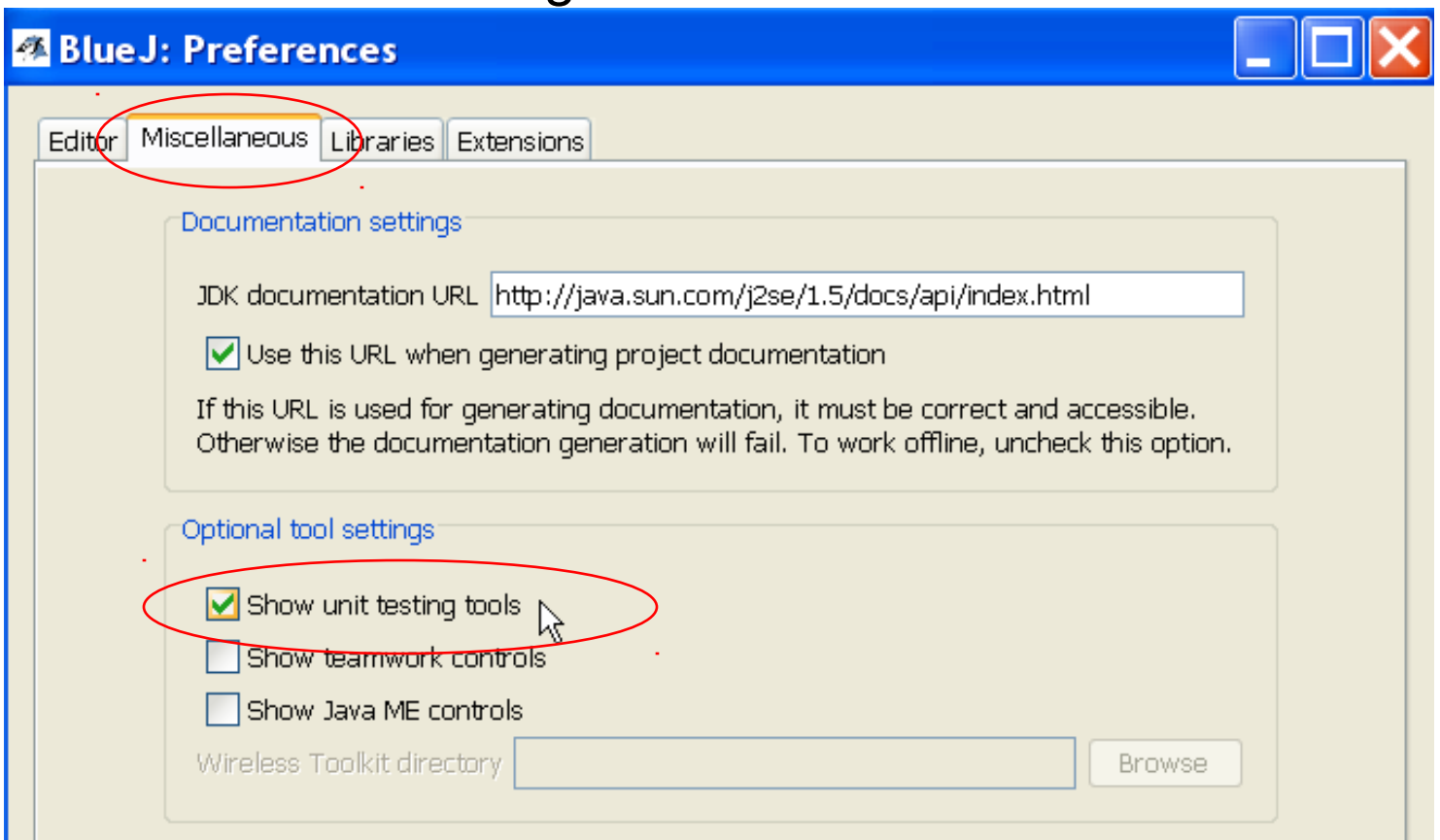
JUnitParams is an open-source add-on Test Runner for JUnit.

- Less coding to define parameters
- Many ways to inject values
- Easier to read - data is closer to test method
- https://github.com/Pragmatists/JUnitParams

Tutorial: https://www.baeldung.com/junit-params

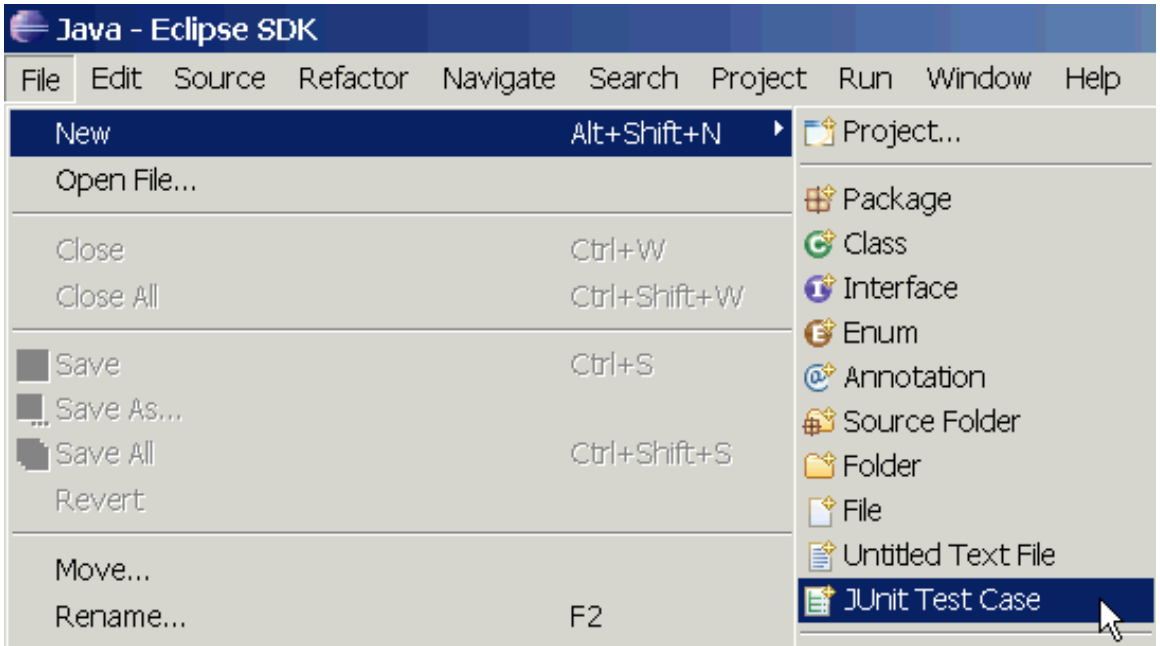
Using JUnit in BlueJ

1. From "Tools" menu select "Preferences..."
2. Select "Miscellaneous" tab.
3. Select "Show unit testing tools".



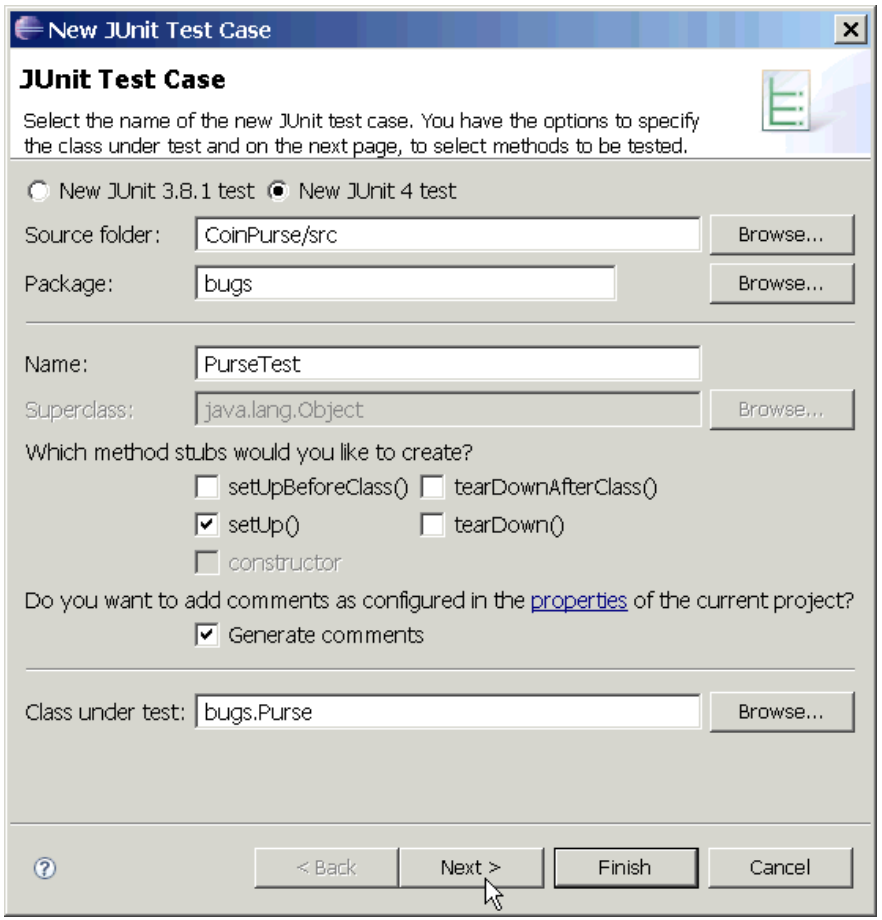
Using JUnit in Eclipse

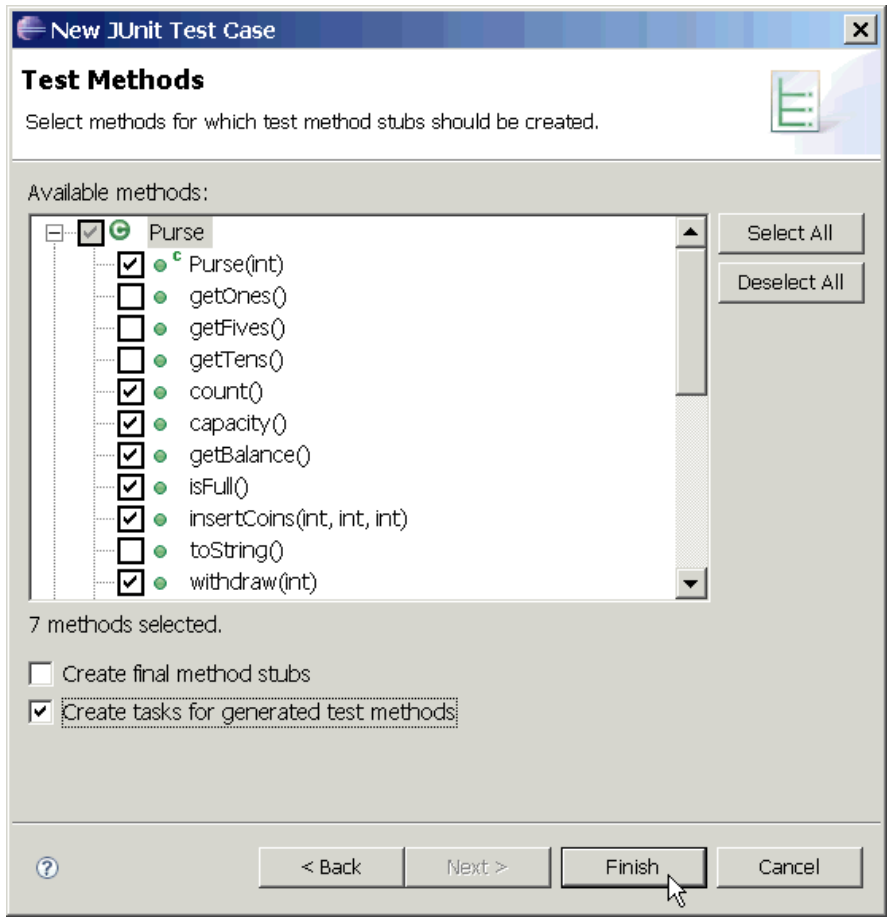
- Eclipse includes JUnit 3.8 and 4.x libraries
  - you should use Junit 4 on your projects
- eclipse will manage running of tests.
  - *but,* you can write your own test running in the main method
- Select a source file to test and then...



Using JUnit in Eclipse (2)

- Select test options and methods to test.



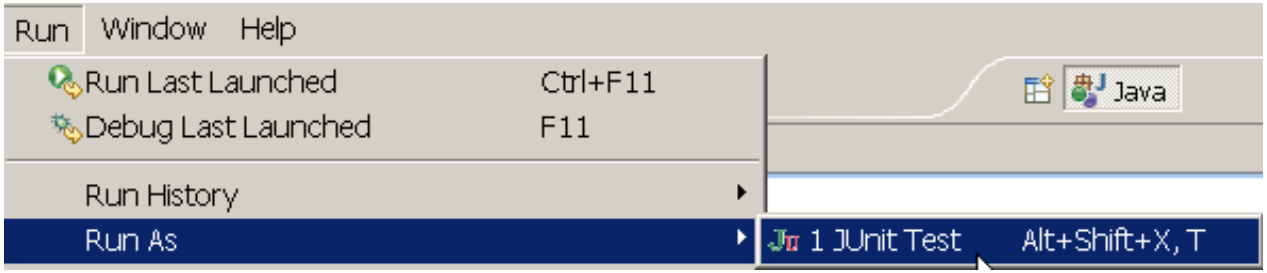


Using JUnit in Eclipse (3)

```
/** 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)

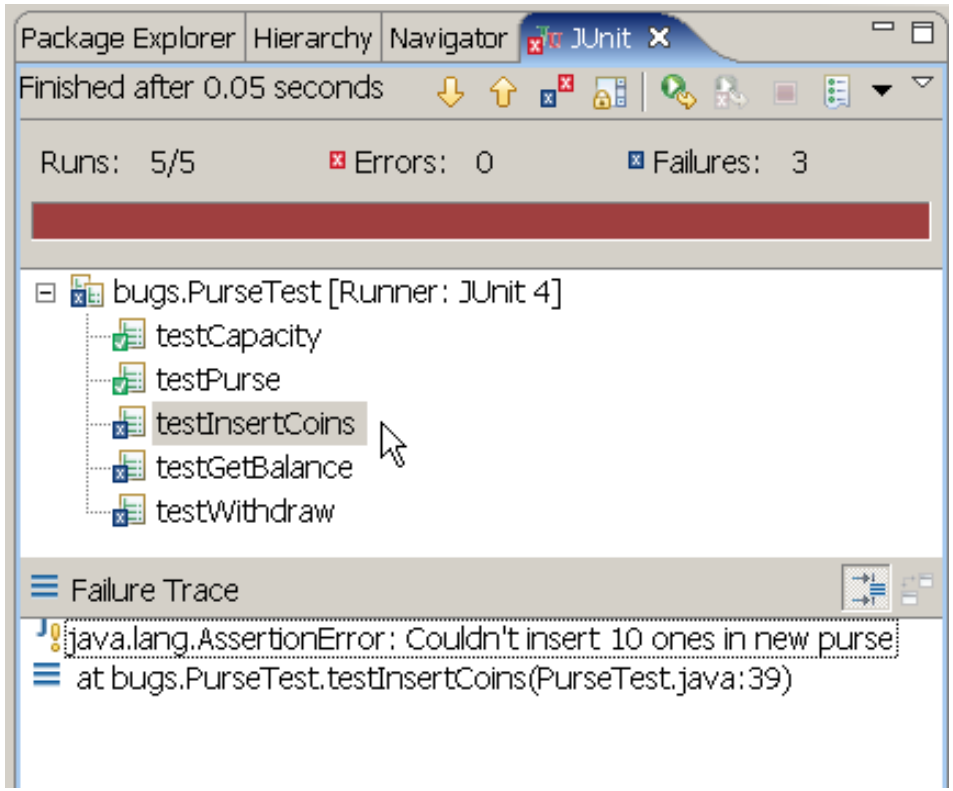


Select the JUnit test case file and choose

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

JUnit 5 - The new version of JUnit

TestNG - a "better" JUnit, but not widely used

`http://www.testng.org`

`NUnit` - Unit testing for .Net Applications

`http://www.nunit.org`