CSC 7203

J Paul Gibson, D311

paul.gibson@telecom-sudparis.eu

http://www-public.it-sudparis.eu/~gibson/Teaching/CSC7203/

JUnit

…/~gibson/Teaching/CSC7203/CSC7203-AdvancedOO-L9-JUnit.pdf

https://devops.com/unit-testing-dilemma/

JUnit

- A unit testing framework for the Java programming language.
- Key in the development of test-driven development
- Part of a family of unit testing frameworks which is collectively known as xUnit that originated with SUnit.
- JUnit is linked as a JAR at compile-time
- package **junit.framework** for JUnit 3.8 and earlier;
  package **org.junit** for JUnit 4 and later.
- On GitHub, it is (one of) the most commonly included external libraries (>30%)

Unit testing

- **unit testing**: Looking for errors in a subsystem in isolation.
  - Generally a "subsystem" means a particular class or object.
  - The Java library **JUnit** helps us to easily perform unit testing.

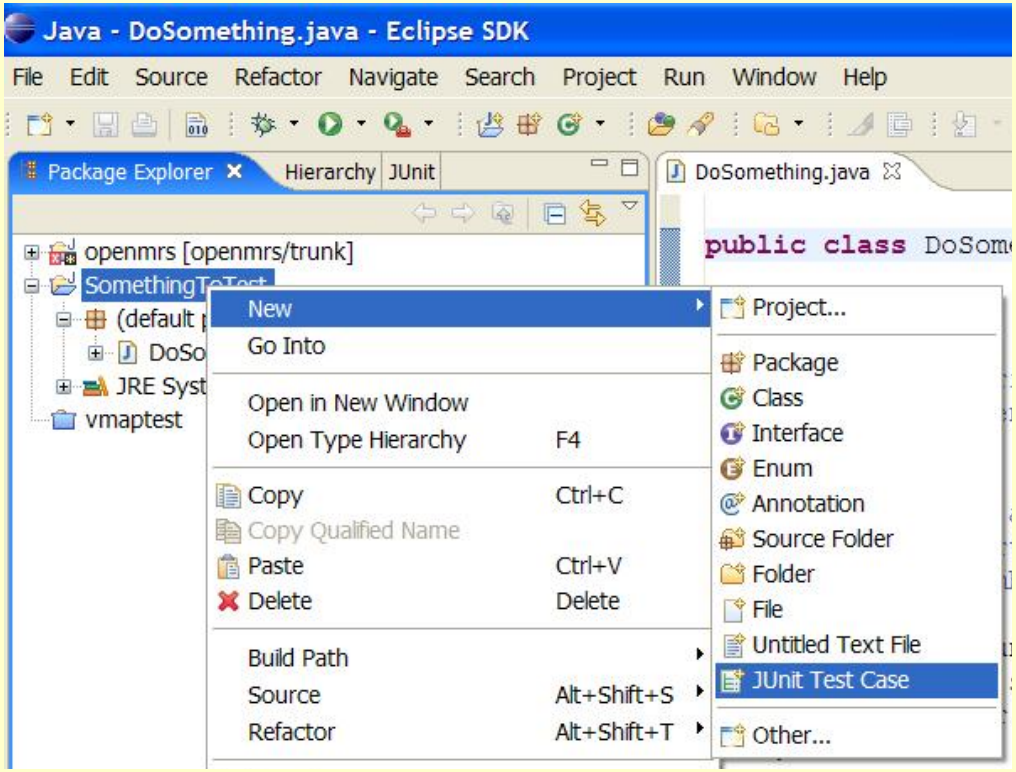
- The basic idea:
  - For a given class `Foo`, create another class `FooTest` to test it, containing various "test case" methods to run.
  - Each method looks for particular results and passes / fails.

- JUnit provides "**assert**" commands to help us write tests.
  - The idea: Put assertion calls in your test methods to check things you expect to be true. If they aren't, the test will fail.

JUnit and Eclipse

- To add JUnit to an Eclipse project, click:
  - **Project → Properties → Build Path → Libraries → Add Library... → JUnit → JUnit 4 → Finish**

- To create a test case:
  - right-click a file and choose **New → Test Case**
  - or click **File → New → JUnit Test Case**
  - Eclipse can create stubs of method tests for you.



A JUnit test class

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

public class name {
    ...

    @Test
    public void name() {   // a test case method
        ...
    }
}
```

- A method with `@Test` is flagged as a JUnit test case.
  - All `@Test` methods run when JUnit runs your test class.

JUnit assertion methods

`assertTrue(`**`test`**`)`	fails if the boolean test is `false`
`assertFalse(`**`test`**`)`	fails if the boolean test is `true`
`assertEquals(`**`expected, actual`**`)`	fails if the values are not equal
`assertSame(`**`expected, actual`**`)`	fails if the values are not the same (by ==)
`assertNotSame(`**`expected, actual`**`)`	fails if the values *are* the same (by ==)
`assertNull(`**`value`**`)`	fails if the given value is *not* `null`
`assertNotNull(`**`value`**`)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

- Each method can also be passed a string to display if it fails:
  - e.g. `assertEquals(`**`"message", expected, actual`**`)`

ArrayIntList JUnit test

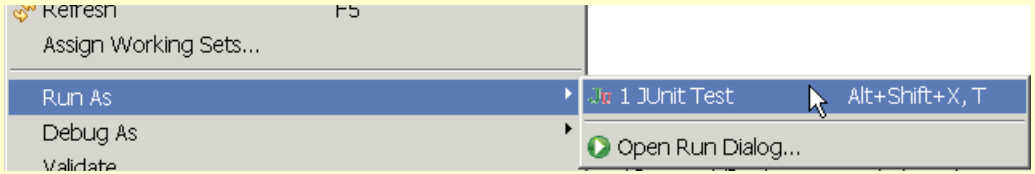
```
import org.junit.*;
import static org.junit.Assert.*;

public class TestArrayIntList {
    @Test
    public void testAddGet1() {
        ArrayIntList list = new ArrayIntList();
        list.add(42);
        list.add(-3);
        list.add(15);
        assertEquals(42, list.get(0));
        assertEquals(-3, list.get(1));
        assertEquals(15, list.get(2));
    }

    @Test
    public void testIsEmpty() {
        ArrayIntList list = new ArrayIntList();
        assertTrue(list.isEmpty());
        list.add(123);
        assertFalse(list.isEmpty());
        list.remove(0);
        assertTrue(list.isEmpty());
    }
    ...
```

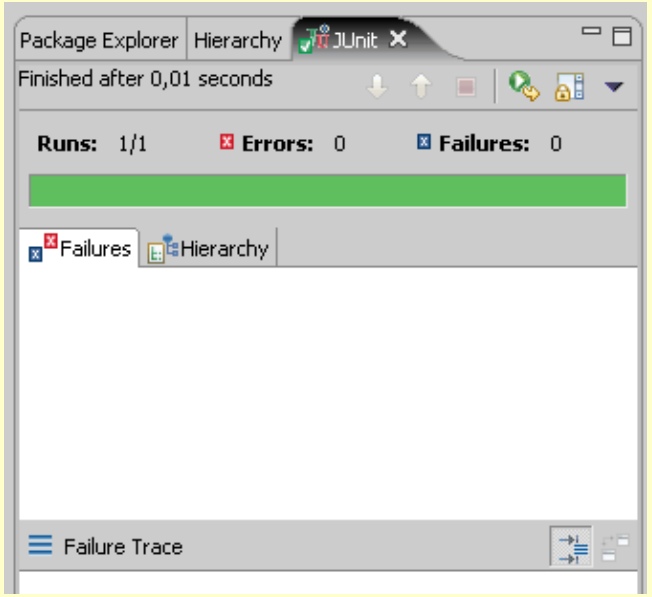
Running a test

- Right click it in the Eclipse Package Explorer at left; choose:
  Run As → JUnit Test

- The JUnit bar will show **green** if all tests pass, **red** if any fail.

- The Failure Trace shows which tests failed, if any, and why.





Well-structured assertions

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(2050, d.getYear());  // expected
        assertEquals(2, d.getMonth());    // value should
        assertEquals(19, d.getDay());     // be at LEFT
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals("year after +14 days", 2050, d.getYear());
        assertEquals("month after +14 days", 3, d.getMonth());
        assertEquals("day after +14 days", 1, d.getDay());
    }   // test cases should usually have messages explaining
}       // what is being checked, for better failure output
```

Expected answer objects

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals(expected, d);  // use an expected answer
    }                               // object to minimize tests

                                    // (Date must have toString
    @Test                           //  and equals methods)
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }
}
```

Naming test cases

```
public class DateTest {
    @Test
    public void test_addDays_withinSameMonth_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days", expected, actual);
    }
    // give test case methods really long descriptive names

    @Test
    public void test_addDays_wrapToNextMonth_2() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
    // give descriptive names to expected/actual values
}
```

Tests with a timeout

```
@Test(timeout = 5000)
public void name() { ... }
```

- The above method will be considered a failure if it doesn't finish running within 5000 ms

```
private static final int TIMEOUT = 2000;
...

@Test(timeout = TIMEOUT)
public void name() { ... }
```

- Times out / fails after 2000 ms

Pervasive timeouts

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_withinSameMonth_1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days", expected, d);
    }


    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_wrapToNextMonth_2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }

    // almost every test should have a timeout so it can't
    // lead to an infinite loop; good to set a default, too
    private static final int DEFAULT_TIMEOUT = 2000;
}
```

Testing for exceptions

```
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```

- Will pass if it *does* throw the given exception.
  - If the exception is *not* thrown, the test fails.
  - Use this to test for expected errors.

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayIntList list = new ArrayIntList();
    list.get(4);   // should fail
}
```

Setup and teardown

```
@Before
public void name() { ... }
@After
public void name() { ... }
```

- methods to run before/after each test case method is called

```
@BeforeClass
public static void name() { ... }
@AfterClass
public static void name() { ... }
```

- methods to run once before/after the entire test class runs

Trustworthy tests

- Test one thing at a time per test method.
  - 10 small tests are much better than 1 test 10x as large.

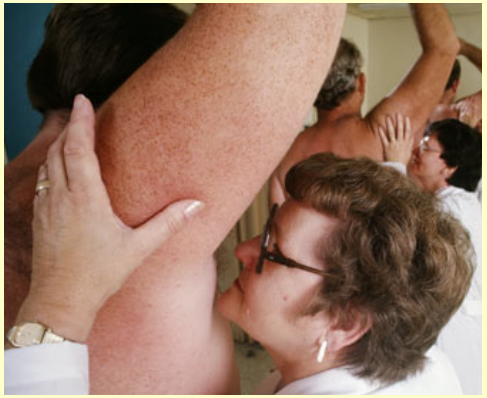
- Each test method should have few (likely 1) assert statements.
  - If you assert many things, the first that fails stops the test.
  - You won't know whether a later assertion would have failed.

- Tests should avoid logic.
  - minimize `if/else`, `loops`, `switch`, etc.
  - avoid `try/catch`
    - If it's supposed to throw, use `expected=` ... if not, let JUnit catch it.

- **Torture** tests are okay, but only *in addition to* simple tests.

Test case "smells"

- Tests should be self-contained and not care about each other.



- "**Smells**" (bad things to avoid) in tests:
  - *Constrained test order* : Test A must run before Test B.
    (usually a misguided attempt to test order/flow)
  - *Tests call each other* : Test A calls Test B's method
    (calling a shared helper is OK, though)
  - *Mutable shared state* : Tests A/B both use a shared object.
    (If A breaks it, what happens to B?)

Test suites

- **test suite**: One class that runs many JUnit tests.
  - An easy way to run all of your app's tests at once.

```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestCaseName.class,
    TestCaseName.class,
    ...
    TestCaseName.class,
})
public class name {}
```

Test suite example

```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    WeekdayTest.class,
    TimeTest.class,
    CourseTest.class,
    ScheduleTest.class,
    CourseComparatorsTest.class
})
public class HW2Tests {}
```

JUnit summary

- Tests need *failure atomicity* (ability to know exactly what failed).
  - Each test should have a clear, long, descriptive name.
  - Assertions should always have clear messages to know what failed.
  - Write many small tests, not one big test.
    - Each test should have roughly just 1 assertion at its end.
- Always use a `timeout` parameter to every test.
- Test for expected errors / exceptions.
- Choose a descriptive assert method, not always `assertTrue`.
- Choose representative test cases from equivalent input classes.
- Avoid complex logic in test methods if possible.
- Use helpers, `@Before` to reduce redundancy between tests.

JUnit 4 introduced parameterized tests.

Allow a developer to run the same test over and over again using different values.

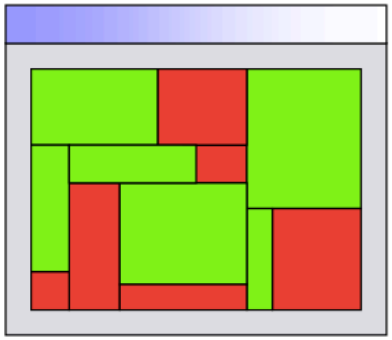
There are five steps that you need to follow to create a parameterized test.
•Annotate test class with @RunWith(Parameterized.class).
•Create a public static method annotated with @Parameters that returns a Collection of Objects (as Array) as test data set.
•Create a public constructor that takes in what is equivalent to one "row" of test data.
•Create an instance variable for each "column" of test data.
•Create your test case(s) using the instance variables as the source of the test data.

The test case will be invoked once for each row of data.

https://examples.javacodegeeks.com/core-java/junit/junit-parameterized-test-example/

Code coverage of Unit Tests

Coverage Units



- Control Flow Coverage
  - → Classes
  - → Methods
  - → Lines
  - → Statements
  - → Branches
  - → Paths

$$\text{Coverage Ratio} = \frac{\text{Covered Units}}{\text{Total Units}}$$

Code coverage of Unit Tests

Statement Coverage

```
public int clip(int lower, int upper, int x) {
    if (x < lower) {
        x = lower;
    }
    if (x > upper) {
        x = upper;
    }
    return x;
}
```

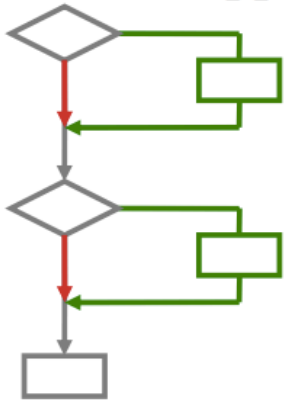
Test Set for Full Statement Coverage:

```
clip(1, 9, 0)
clip(1, 9, 10)
```

Code coverage of Unit Tests

Branch Coverage

```
public int clip(int lower, int upper, int x) {
    if (x < lower) {
        x = lower;
    }
    if (x > upper) {
        x = upper;
    }
    return x;
}
```



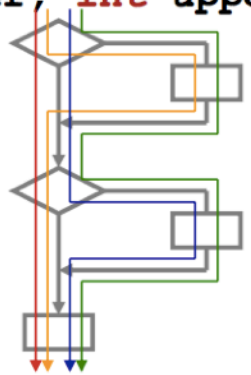
Test Set for Full Branch Coverage:

```
clip(1, 9, 0)
clip(1, 9, 10)
```

Code coverage of Unit Tests

Path Coverage

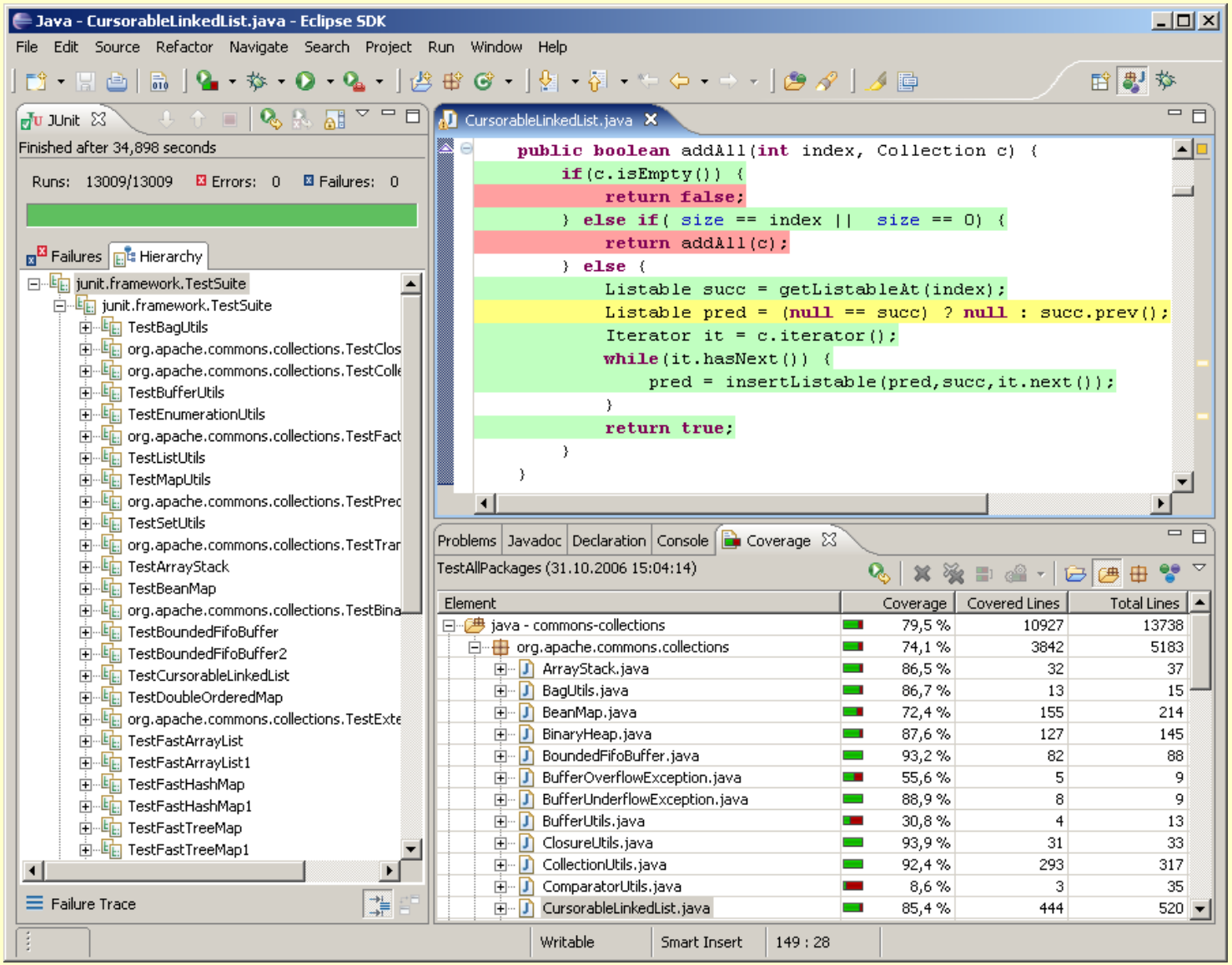
```
public int clip(int lower, int upper, int x) {
    if (x < lower) {
        x = lower;
    }
    if (x > upper) {
        x = upper;
    }
    return x;
}
```



Test Set for Full Path Coverage:

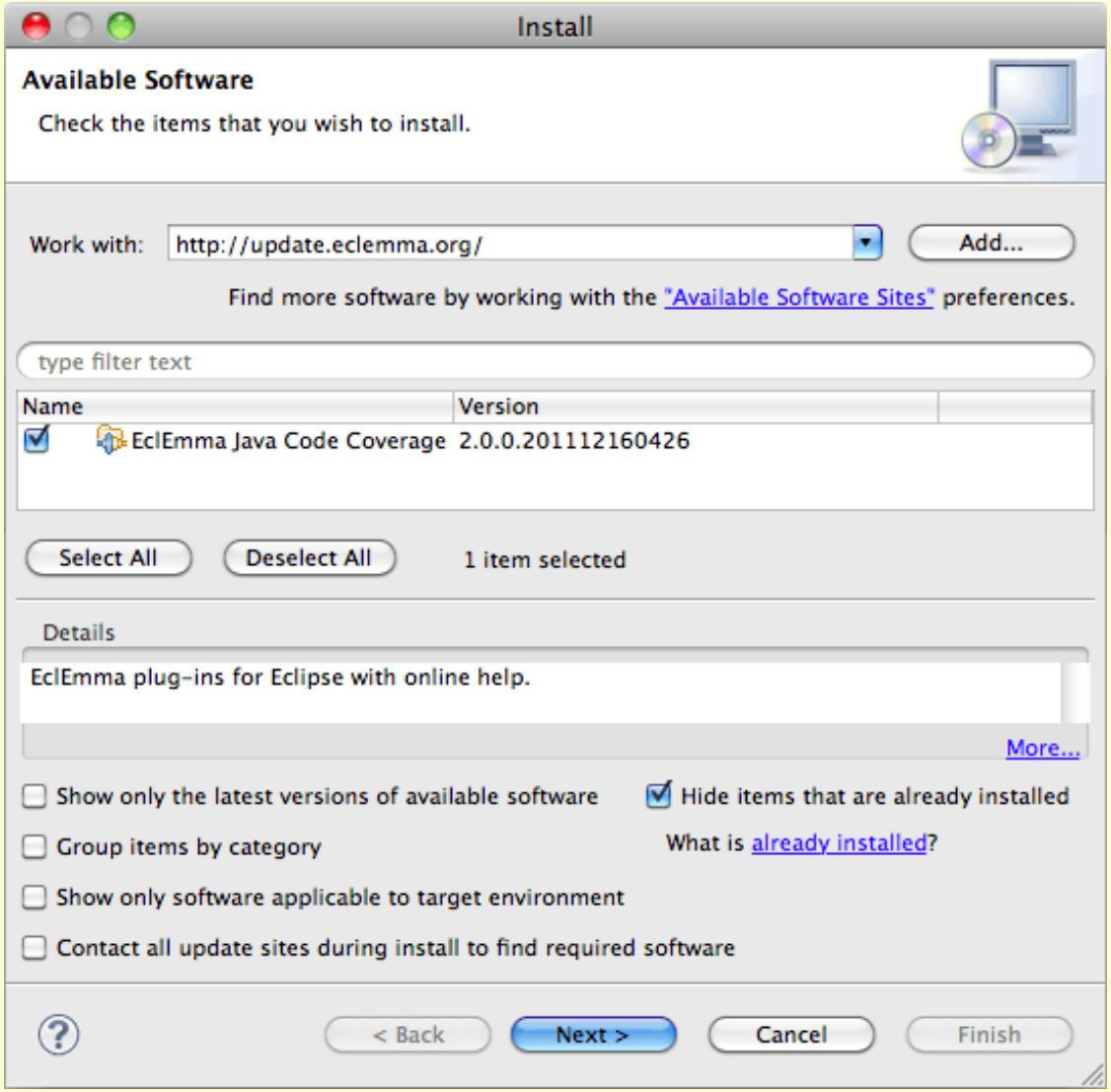
```
clip(1, 9, 0)
clip(1, 9, 10)
clip(1, 9, 5)
clip(9, 1, 5)
```

Code coverage of Unit Tests



http://www.eclemma.org

Code coverage of Unit Tests



http://www.eclemma.org/installation.html

Code coverage of Unit Tests

Source Code Annotation

Line coverage and branch coverage of the active coverage session is also directly displayed in the Java source editors.

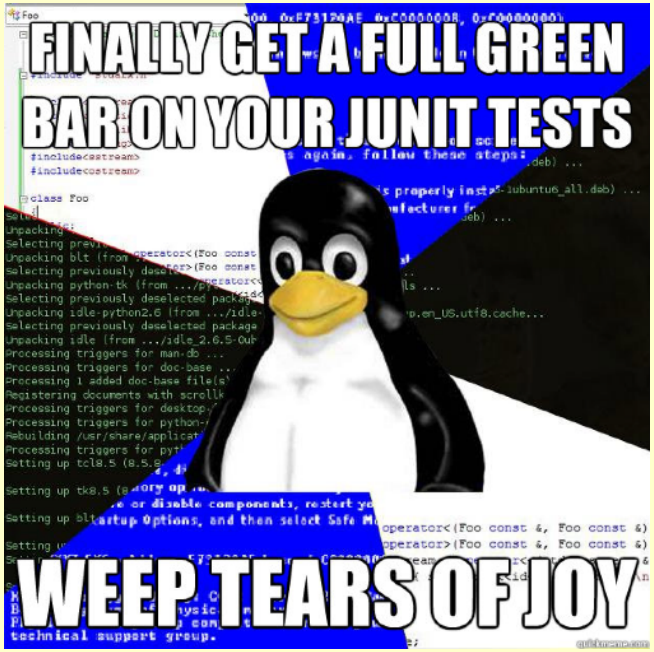
```
    public boolean addAll(int index, Collection c) {
        if(c.isEmpty()) {
            return false;
        } else if(_size == index || _size == 0) {
            return addAll(c);
        } else {
            Listable succ = getListableAt(index);
            Listable pred = (null == succ) ? null : succ.prev();
            Iterator it = c.iterator();
            while(it.hasNext()) {
                pred = insertListable(pred,succ,it.next());
            }
            return true;
        }
    }
```

1 of 2 branches missed.

Press 'F2' for focus

Source lines containing executable code get the following color code:

- green for fully covered lines,
- yellow for partly covered lines (some instructions or branches missed) and
- red for lines that have not been executed at all.



All green 100% is not enough (usually)… but it is better than ….?