Unit 3
Unit testing with JUnit

Software Analysis and Design Project

Computer Science

Universidad Autónoma de Madrid

Unit testing

- Unit testing determines if individual modules of a system are fit for use.
  - In OO, a unit is a class and its methods.
- Unit tests must be:
  - Fully automated. (We are going to automate it with JUnit 4)
  - Complete, covering boundary cases.
  - Independent. Tests should never rely on other tests, or be affected by test ordering.

JUnit

- Framework to automate Unit Testing in Java
- For each class in the system, a testing class is created with methods (*tests*) that check the system class
- Each test, annotated with *@Test*, contains:
  - Code that sets up the required objects for the test
  - Execution of the test
  - Assertions to check that the test result meets certain conditions
- JUnit runs all tests, and shows a report with the execution results

JUnit – Example 1

Java Class:

```
public class Complex {
  private float real;
  private float imaginary;

  public Complex (float real, float imaginary) {
    this.real = real;
    this.imaginary = imaginary;
  }

  public float getReal(){
    return real;
  }

  public float getImaginary() {
    return imaginary;
  }

  public Complex add(Complex c) {
    return new Complex(
      real + c.getReal(),
      imaginary + c.getImaginary());
  }
}
```

Test Class:

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

public class ComplexTest {

  @Test
  public void testAdd() {
    // Create necessary objects
    Complex c1 = new Complex(3, 5);
    Complex c2 = new Complex(1, -1);

    // Execute operation
    Complex result = c1.add(c2);

    // Check result using asserts
    assertEquals(4f, result.getReal(), 0);
    assertEquals(4f, result.getImaginary(), 0);
  }
}
```

JUnit

- Common asserts:
  - assertTrue(<*bool_expr*>)
  - assertFalse(<*bool_expr*>)
  - assertEquals(...)
  - assertSame(<*expected object*>,<*actual object*>)
  - assertNotSame(<*unexpected object*>,<*actual object*>)
  - assertNull(<*object*>)
  - assertNotNull(<*object*>)
  - fail(<*message*>): fails a test with the given message

Complete list at:
http://junit.sourceforge.net/javadoc/org/junit/Assert.html

JUnit

- Methods in a test class are executed in the following order:
  - Static method with @*BeforeClass* annotation
  - For each method annotated with @*Test*:
    - Test class Constructor
    - Method annotated with @*Before* (set up)
    - Method annotated with @*Test*
    - Method annotated with @*After* (teardown)
  - Static method with @*AfterClass* annotation

JUnit – Example 2

Java Class:

```
public class Complex {
  private float real;
  private float imaginary;

  public Complex (float real, float imaginary) {
    this.real = real;
    this.imaginary = imaginary;
  }

  public float getReal(){
    return real;
  }

  public float getImaginary() {
    return imaginary;
  }

  public Complex add(Complex c) {
    return new Complex(
      real + c.getReal(),
      imaginary + c.getImaginary());
  }

  public Complex divide(Complex c) {
    ...
  }
}
```

Test Class:

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

public class ComplexTest {

  private Complex c1;
  private Complex c2;

  @Before
  public void setUp() throws Exception {
    c1 = new Complex(3, 5);
    c2 = new Complex(1, -1);
  }

  @Test
  public void testAdd() {
    Complex result = c1.sumar(c2);
    assertEquals(4f, result.getReal(), 0);
    assertEquals(4f, result.getImaginary(), 0);
  }

 @Test
  public void testDivide() {
    Complex result = c1.divide(c2);
    assertEquals(-1f, result.getReal(), 0);
    assertEquals(4f, result.getImaginary(), 0);
  }
}
```

Executed before each test

JUnit – Example 3

Java Class:

```
public class Complex {
  private float real;
  private float imaginary;

  public Complex (float real, float imaginary) {
    this.real = real;
    this.imaginary = imaginary;
  }

  public float getReal(){
    return real;
  }

  public float getImaginary() {
    return imaginary;
  }

  public Complex add(Complex c) {
    return new Complex(
      real + c.getReal(),
      imaginary + c.getImaginary());
  }

  public Complex divide(Complex c) throws Exception{
    ...
  }
}
```

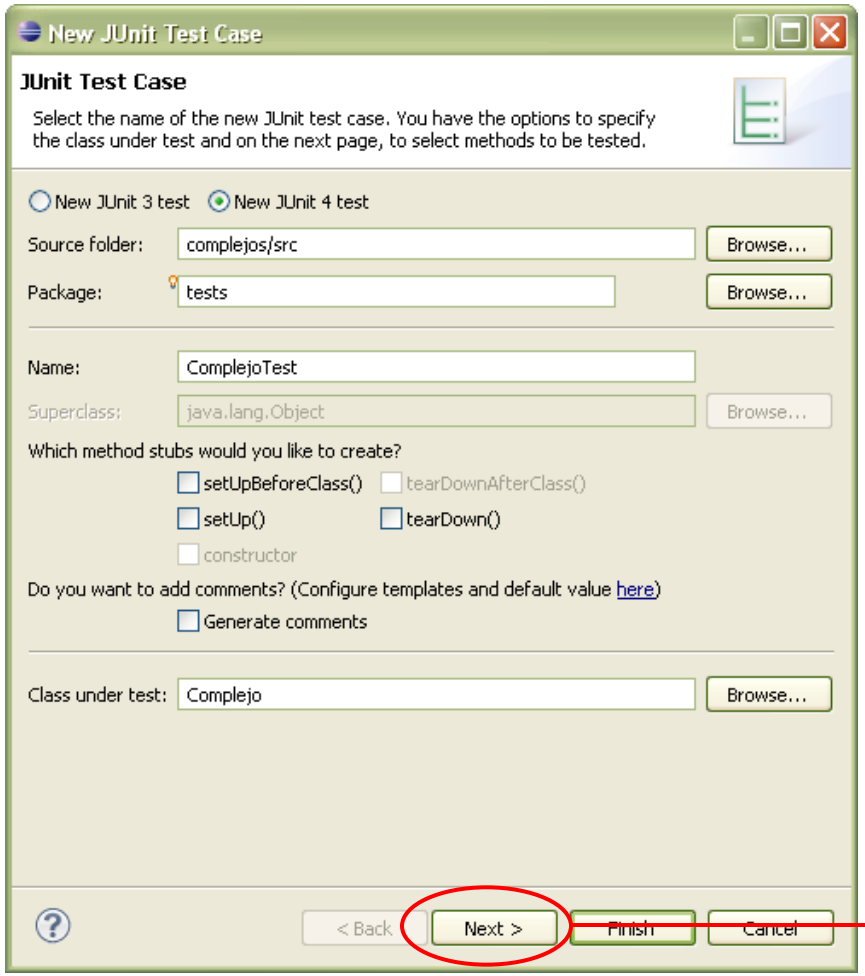
Test Class:

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

public class ComplexTest {

  private Complex c1;
  private Complex c2;

  @Before
  public void setUp() throws Exception {
    c1 = new Complex(3, 5);
    c2 = new Complex(1, -1);
  }

…

  @Test (expected = ArithmeticException.class)
  public void testDivideByZero() {
    Complex zero = new Complex(0,0);
    Complex result = c1.divide(zero);
  }
}
```

Test will pass if an ArithmeticException is thrown

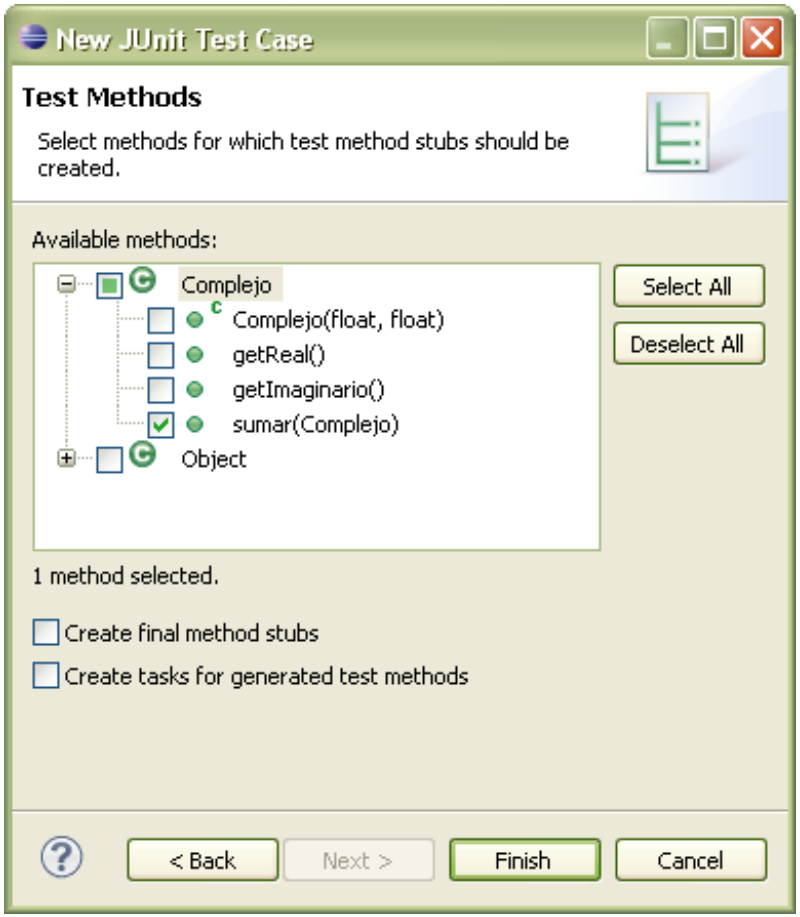
Using JUnit in Eclipse (i)

Step 1:
new JUnit Test Case



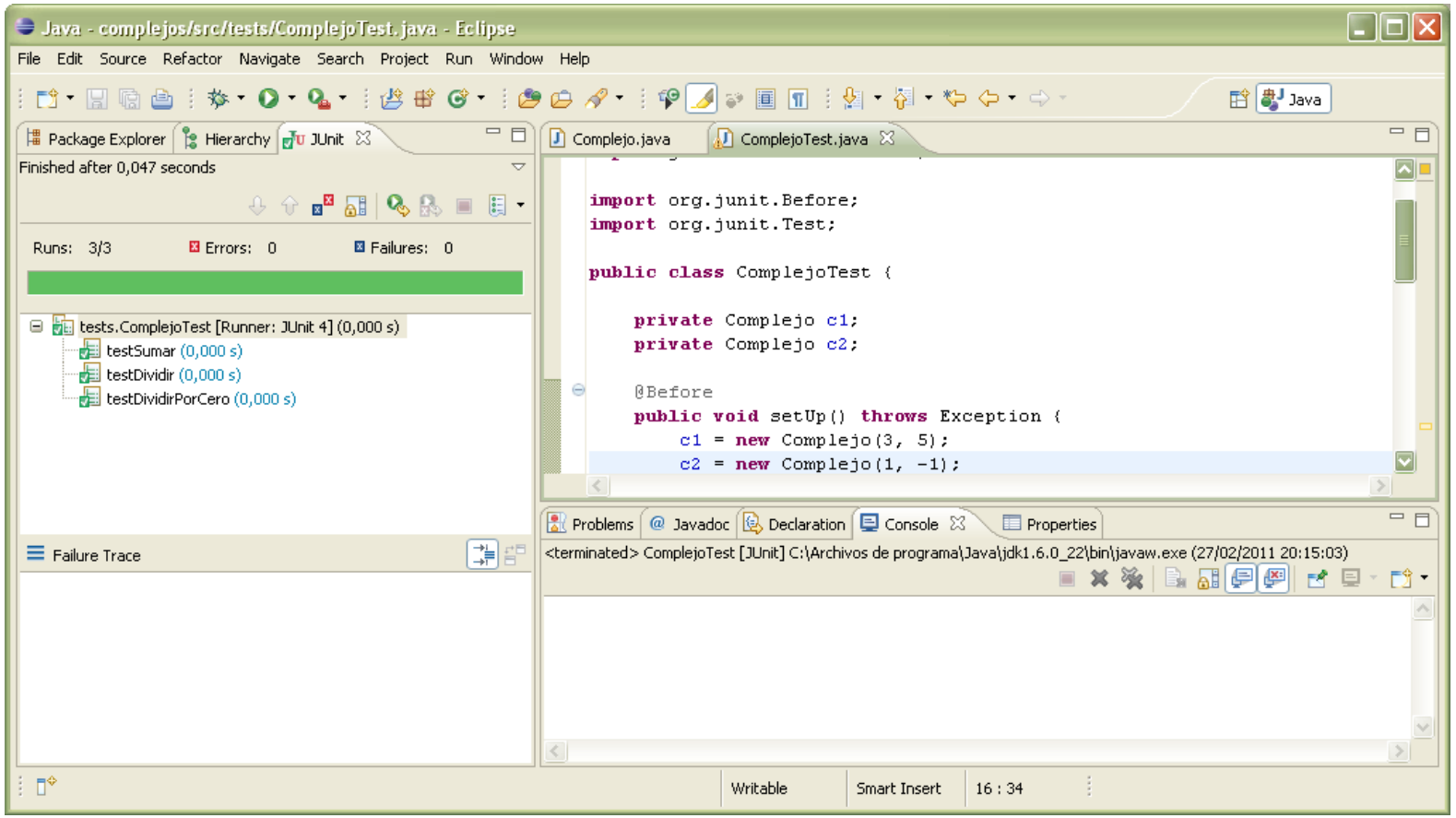
Using JUnit in Eclipse (ii)

Step 2:
Select methods to include in test Unit



Using JUnit in Eclipse (iii)

Report of Test Units execution:



When?

- ***Unit testing***: Test a unit of code (class in OO), before integrating it with other classes. Tests are written while coding the application.
- ***Regression testing***: To ensure that a change in code does not introduce new faults. For example, to test if a change in one part affects other parts of the application.

- ***Test-driven development (TDD)***: The developer writes the test units before starting a new class. After that, he writes the minimum amount of code to pass those tests, and later he refactors it to improve code quality.

Bibliography

- http://junit.sourceforge.net/
- **Test Driven: TDD and Acceptance TDD for Java Developers. Koskela,** Manning Publications, 2007.
- **Pruebas de software y JUnit: un análisis en profundidad y ejemplos prácticos**. Bolaños, Sierra, Alarcón. Prentice-Hall, 2008. INF/681.3.06/BOL