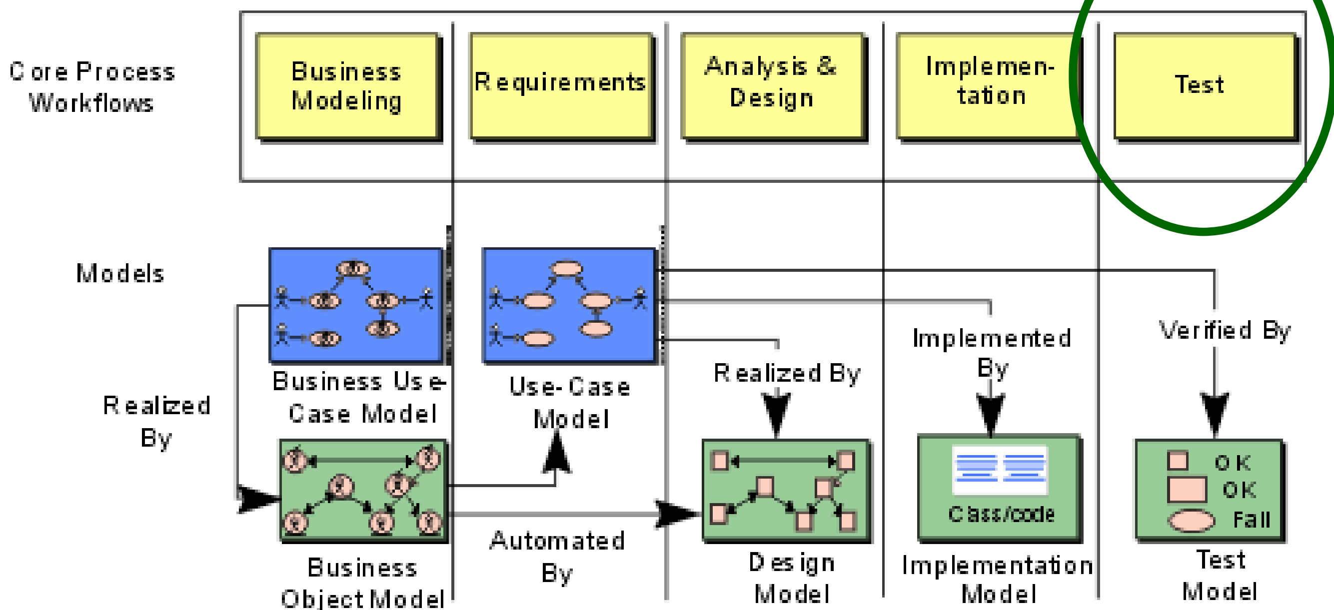
Testing
(Δοκιμές / Έλεγχοι)



However it is a bad practice to leave testing for the end

Testing

Testing is a form of insurance.

It costs more to repair software bugs when people are depending on the programs than in earlier stages before the systems are in use.

- *The purpose is not to demonstrate that the system is free of errors;*
- *The purpose is to detect as many errors as possible … :)*

Testing Philosophy

- It is dangerous to test early modules without an overall testing plan
- It may be difficult to reproduce sequence of events causing an error
- Testing must be done systematically and results documented carefully

Types/Stages of Testing

- **Unit testing**
  - Tests each module to assure that it performs its function
- **Integration testing**
  - Tests the interaction of modules to assure that they work together
- **System testing**
  - Tests to assure that the software works well as part of the overall system
- **Acceptance testing**
  - Tests to assure that the system serves organizational needs

Unit Testing

- Black Box Testing
  - Focuses on whether the unit meets requirements stated in specification
- White-Box Testing
  - Looks inside the module to test its major elements

Integration Testing

- User interface testing
  - Tests each interface function
- Use-case testing
  - Ensures that each use case works correctly
- Interaction testing
  - Tests each process in a step-by-step fashion
- System interface testing
  - Ensures data transfer between systems

System Testing

- Requirements Testing
  - Ensures that integration did not cause new errors
- Usability Testing
  - Tests how easy and error-free the system is in use
- Security Testing
  - Assures that security functions are handled properly
- Performance Testing
  - Assures that the system works under high volumes of activity
- Documentation Testing
  - Analysts check that documentation and examples work properly

Acceptance Testing

- Alpha Testing
  - Repeats tests by users to assure they accept the system
- Beta Testing
  - Uses real data, not test data

Case study: Java Unit Testing

Case: Testing Java Code

- JUnit is a freely available Java unit test framework. extensively used in the Java community because of its elegance of design and ease of use.
- Why use a testing framework?
  - Using a testing framework is beneficial because it forces you to explicitly declare the expected results of specific program execution routes.
  - When debugging it is possible to write a test which expresses the result you are trying to achieve and then debug until the test comes out positive.
  - By having a set of tests that test all the core components of the project it is possible to modify specific areas of the project and immediately see the effect the modifications have on the other areas by the results of the test, hence, side-effects can be quickly realized.

– JUnit promotes the idea of first testing then coding, in that it is possible to setup test data for a unit which defines what the expected output is and then code until the tests pass. It is believed by some that this practice of "test a little, code a little, test a little, code a little..." increases programmer productivity and stability of program code whilst reducing programmer stress and the time spent debugging. It also integrates with Ant (http://jakarta.apache.org/ant)

Short Introduction to JUnit

Suppose you want to test the following code

```
public class Math {
    static public int add(int a, int b) {
        return a + b;
    }
}
```

```
import junit.framework.*;
public class TestMath extends TestCase {
  public void testAdd() {
          int num1 = 3;
          int num2 = 2;
          int total  = 5;
          int sum = 0;
          sum = Math.add(num1, num2);
          assertEquals(sum, total);
    }
  }
```

To test this code, you need a second Java class that will

1) import junit.framework.*
2) extend TestCase.

Short Introduction to JUnit

- Notice that the routine is named testAddNumbers. This convention tells you that the routine is supposed to be a test and that it's targetting the "add" functionality.
- The last step is how to run your JUnit tests. You can do this several ways, including your command line, Eclipse or the JUnit Test Runner, or plain Ant. To run a Junit test with an Ant script, add this to your Ant script:

```
<junit printsummary="yes" haltonfailure="yes" showoutput="yes" >
          <classpath>
                  <pathelement path="${build}"/>
          </classpath>
        <batchtest fork="yes" todir="${reports}/raw/">
                  <formatter type="xml"/>
                  <fileset dir="${src}">
                          <include name="**/*Test*.java"/>
                  </fileset>
          </batchtest>
  </junit>
```

Assertion Statements

There is a list of the different types of assertion statements that are used to test your code. Any Java data type or object can be used in the statement. These assertions are taken from the JUnit API.

- **assertEquals**(expected, actual)
- **assertEquals**(message, expected, actual)
- **assertEquals**(expected, actual, delta) - used on doubles or floats, where delta is the difference in precision
- **assertEquals**(message, expected, actual, delta) - used on doubles or floats, where delta is the difference in precision
- **assertFalse**(condition)
- **assertFalse**(message, condition)
- **assertNotNull**(object)
- **assertNotNull**(message, object)

Assertion Statements (cont)

- **assertNotSame**(expected, actual)
- **assertNotSame**(message, expected, actual)
- **assertNull**(object)
- **assertNull**(message, object)
- **assertSame**(expected, actual)
- **assertSame**(message, expected, actual)
- **assertTrue**(condition)
- **assertTrue**(message, condition)
- **fail**()
- **fail**(message)
- **failNotEquals**(message, expected, actual)
- **failNotSame**(message, expected, actual)
- **failSame**(message)

Another example

```
// A Class that adds up a string based on the ASCII values of its
// characters and then returns the binary representation of the sum.
public class BinString {
  public  BinString() {}

   public String convert(String s) {
      return binarise(sum(s));
   }

   public int sum(String s) {
      if(s=="") return 0;
      if(s.length()==1) return ((int)(s.charAt(0)));
      return ((int)(s.charAt(0)))+sum(s.substring(1));
   }

   public String binarise(int x) {
      if(x==0) return "";
      if(x%2==1) return "1"+binarise(x/2);
      return "0"+binarise(x/2);
   }
}
```

the test class

```
import junit.framework.*;
public class BinStringTest extends TestCase {
   private BinString binString;

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

   protected void setUp() {
      binString = new BinString();
   }

   public void testSumFunction() {
      int expected = 0;
      assertEquals(expected, binString.sum(""));
      expected = 100;
      assertEquals(expected, binString.sum("d"));
      expected = 265;
      assertEquals(expected, binString.sum("Add"));
   }
```

the test class (cont)

```
public void testBinariseFunction() {
      String expected = "101";
      assertEquals(expected, binString.binarise(5));
      expected = "11111100";
      assertEquals(expected, binString.binarise(252));
   }

   public void testTotalConversion() {
     String expected = "1000001";
     assertEquals(expected, binString.convert("A"));
   }
}
```

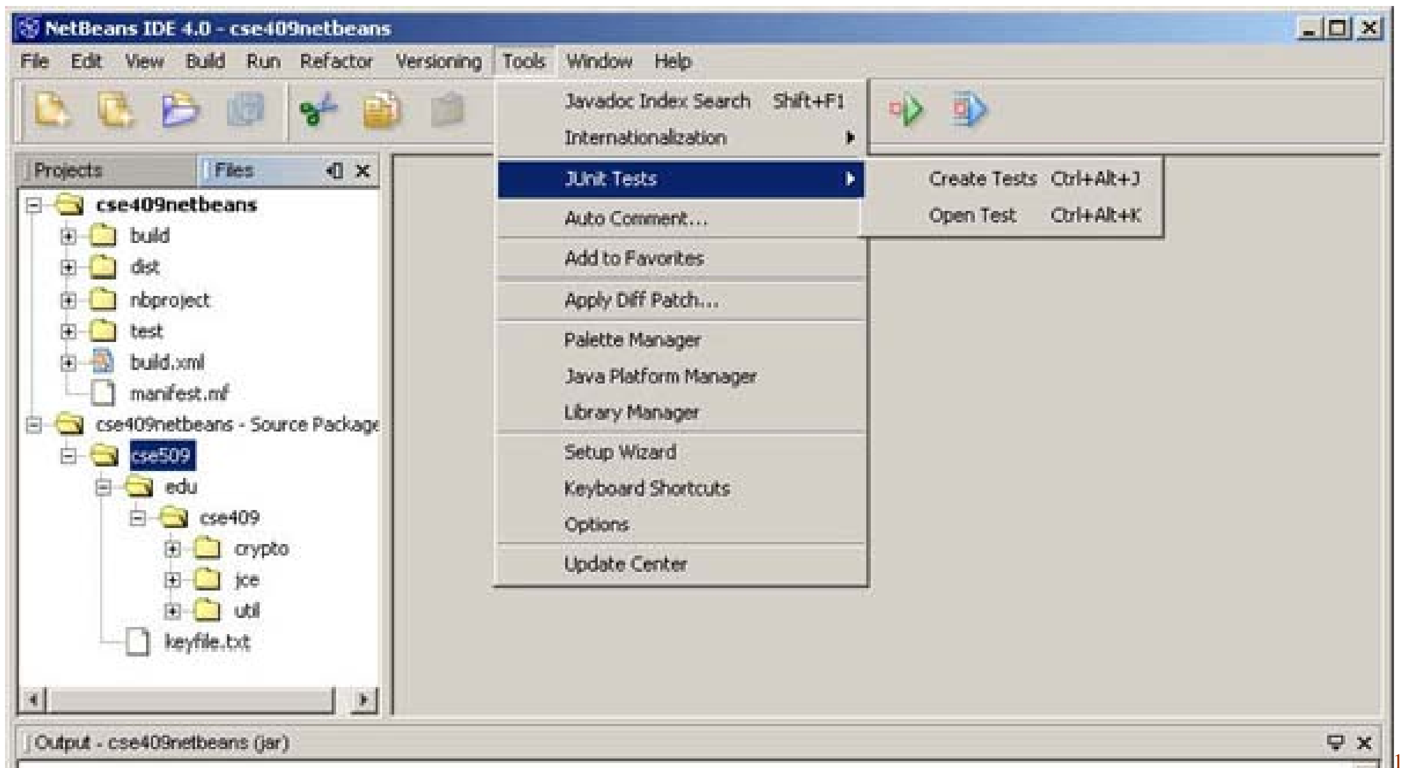
setUp and tearDown

- TestCase allows us to use the method ***setUp*** to set up any necessary variables or objects (***setUp*** is called before the evaluation of each test)
  - Note that *binString* was already declared at the top of the file like usual with `private BinString binString;`
- *setUp* has a brother called ***tearDown*** which is called after the evaluation of each test and can be used to dereference variables or whatever so that each test may be performed without issuing side-effects that may affect the outcome of the other tests.

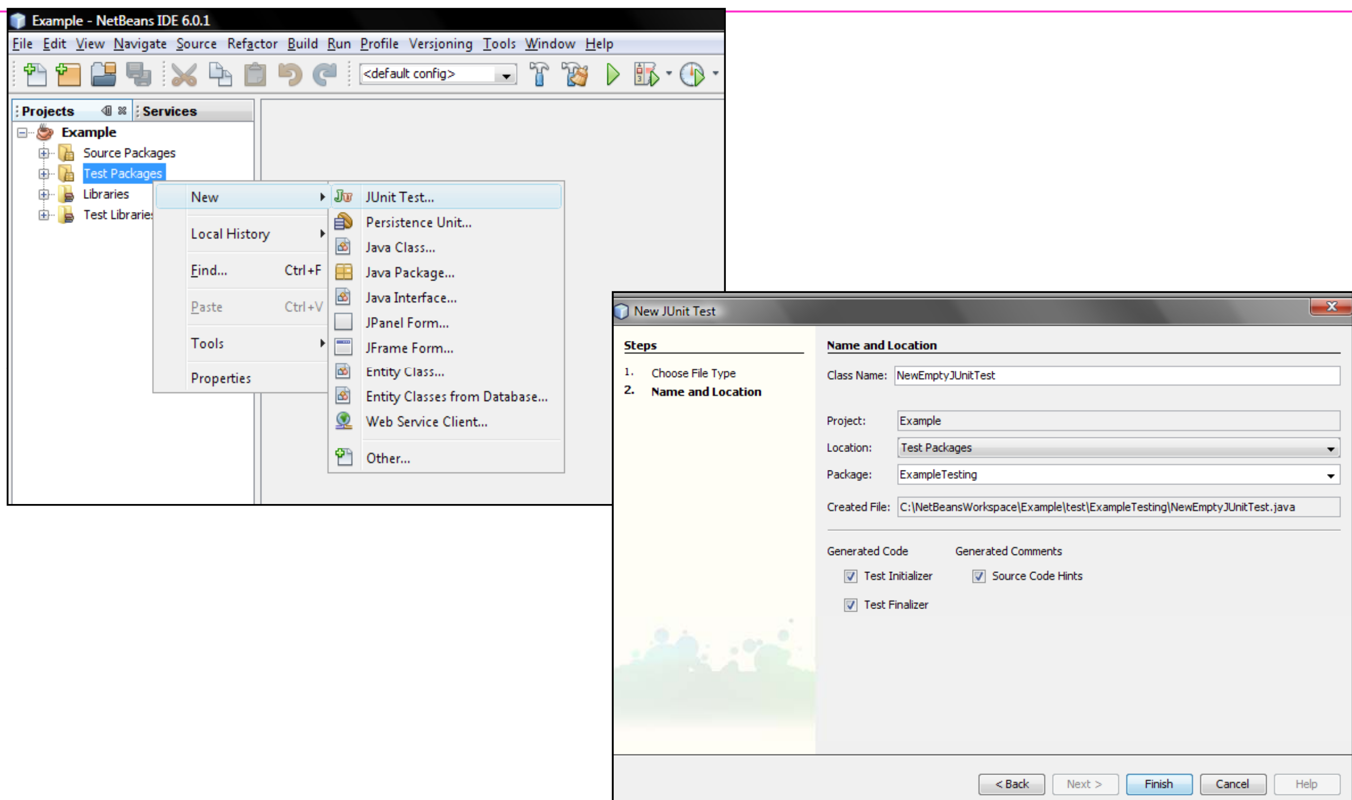
Tips

- Using an IDE (like Eclipse, NetBeans) you can create tests very quickly and easily
- Tests running can be automated

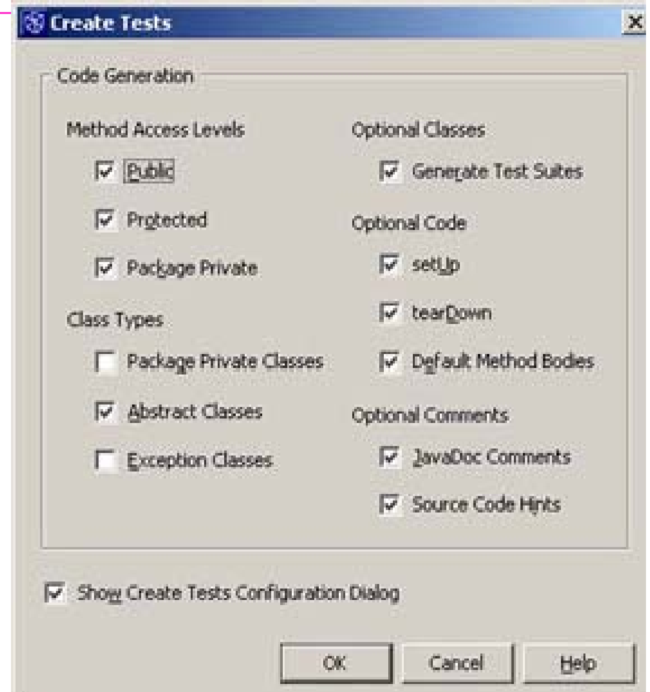
case NetBeans
creating test 1/2



case NetBeans
creating test 2/2



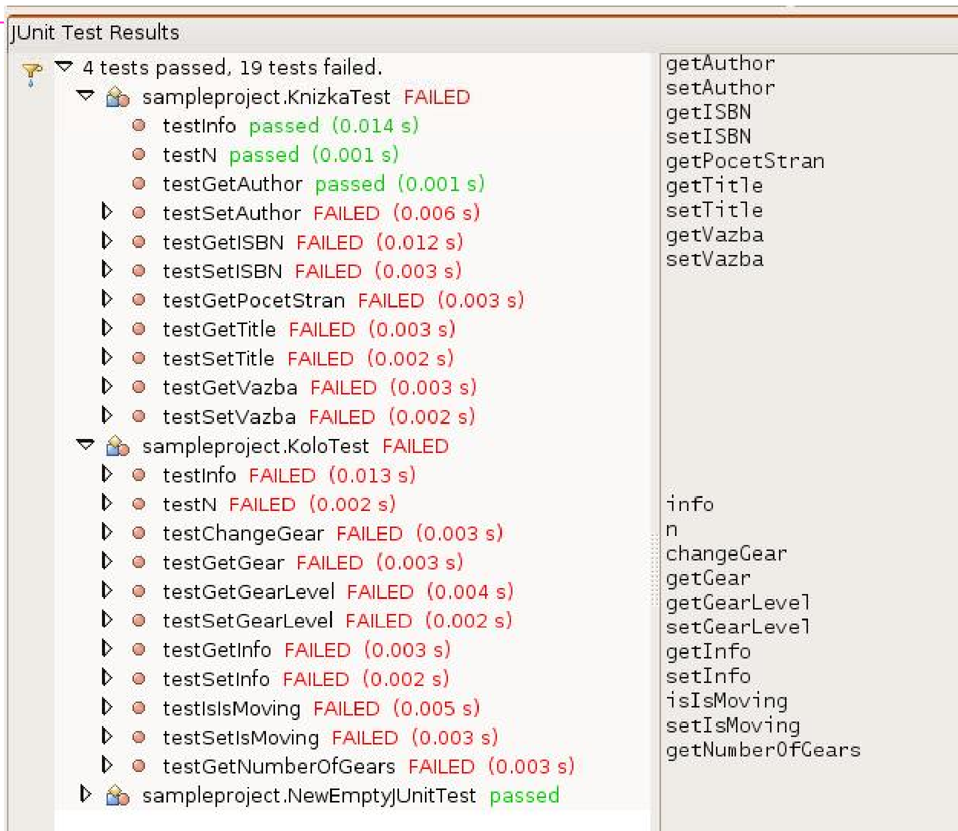
case NetBeans
selecting the elements for which tests should be created



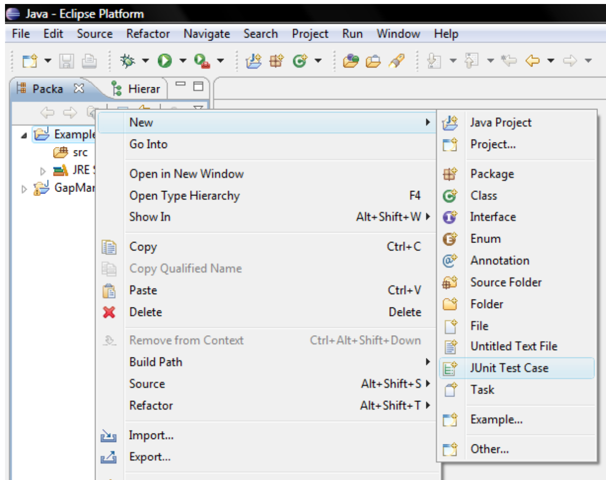
We are now presented with a Create Tests window. You will notice several check boxes. If the methods that you are trying to test are not static meaning that they need an instance of the object to be called then you will probably want to specify a **setUp** method for the test case. This will be where you can instantiate your object before running the tests. Also if your tests allocate any special resources you may want to select **tearDown** so you can properly dispose of the resources. After we are done with this you may click the next button.

- An alternate way to create tests for individual classes is to right click on the file that you wish to create a test for then goto JUnit Tests and then Create Tests. This will allow you to create a single test for a particular class.

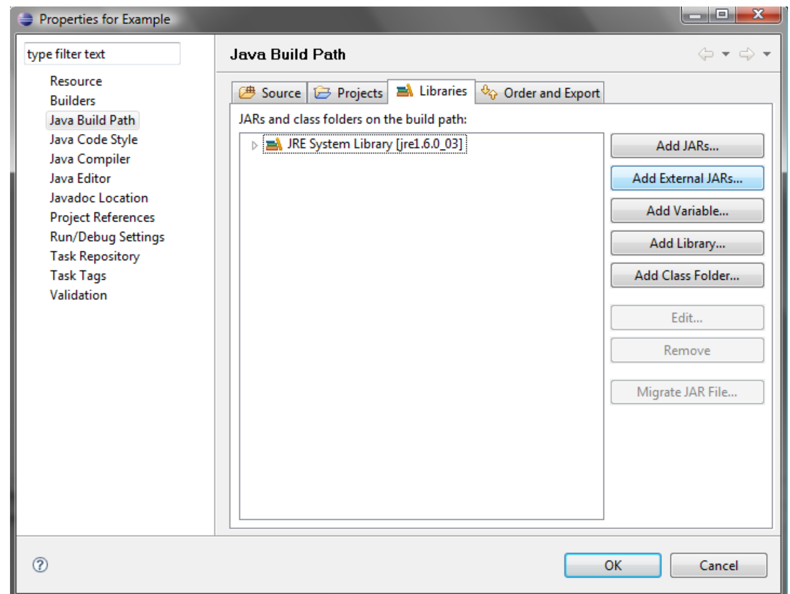
The results of running the tests



case Eclipse
creating test 1/2



In Eclipse we must add the JUnit Library explicitly



case Eclipse
creating test 2/2