FloConsult SPRL

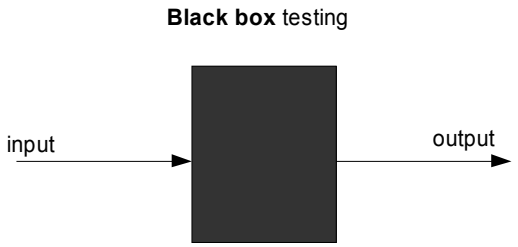
Introduction to unit testing with Java, Eclipse and Subversion

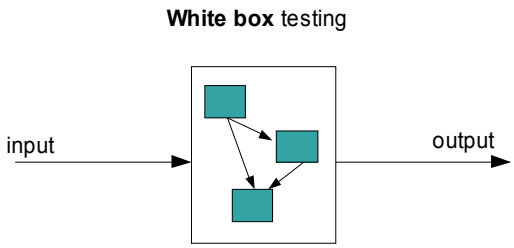
Table of Contents

1. About Unit Tests

1.1. Introduction

Unit testing is a key technique in today's agile software development. Unit tests are written by the developer during the coding activity, and thus pertain in the white box testing category:





Unit tests have following characteristics:

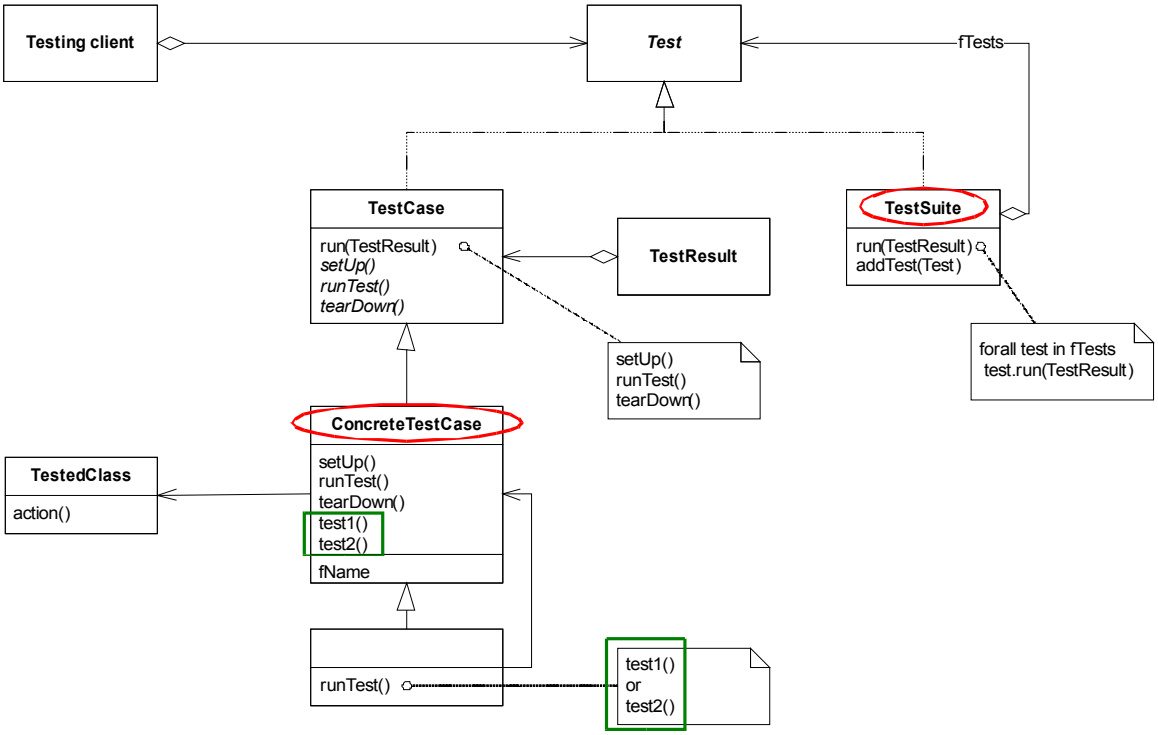
- they are **automatic** (do not require human intervention during the tests)
- they are **reproducible**
- they are **independent** from each other
- their **result** is either **OK** (all tests have been successful), or **NOK** (at least one test failed, in this case details are given to the user about the failure(s))
- may be combined in **test suites** to allow non-regression testing during the development

This document will present progressively concrete unit testing cases, using:

- **Java** as programming language (http://www.java.com/),
- **Eclipse** as Development Environment (http://www.eclipse.org/)
- **Subversion** as version control system (http://subversion.tigris.org/), its GUI for Windows **TortoiseSVN** (http://www.eclipse.org/), and its **Subclipse** plugin for Eclipse (http://subclipse.tigris.org/).
- **Enterprise Architect** as UML Modeling tool (http://www.sparxsystems.com/products/ea.html). .

1.2. Unit tests frameworks

JUnit is the original Java library for unit testing, ad such frameworks exist nowadays for many programming languages. They all aim at following the software design:



`test1()` and `test2()` are methods of the concrete test class `ConcreteTestCase`, one such class being developed per nominal application class (`TestClass`). They contain code testing the services of this application class.

Note that the "test-first" practice recommend writing those test methods before writing the nominal code, because it helps to keep the design simple from the start (see for example the article of Ron Jeffries http://www.xprogramming.com/xpmag/testFirstGuidelines.htm).

Besides the test methods, the developer may define for the concrete test case a `setUp()` method, that will be called be the framework *before* each test method, as well as a `tearDown()` method, called *after* each test.

In JUnit, the default implementation of `runTest()` uses the Java reflexion to invoke methods with names beginning with "`test`"; `runTest()` needs being redefined only if you want to override this behavior.

The `run()` template method calls `setUp()`, `runTest()`, `tearDown()`, what in turns implies concretely the execution of:

- `setUp(), test1(), tearDown()`
- `setUp(), test2(), tearDown()`
- ...

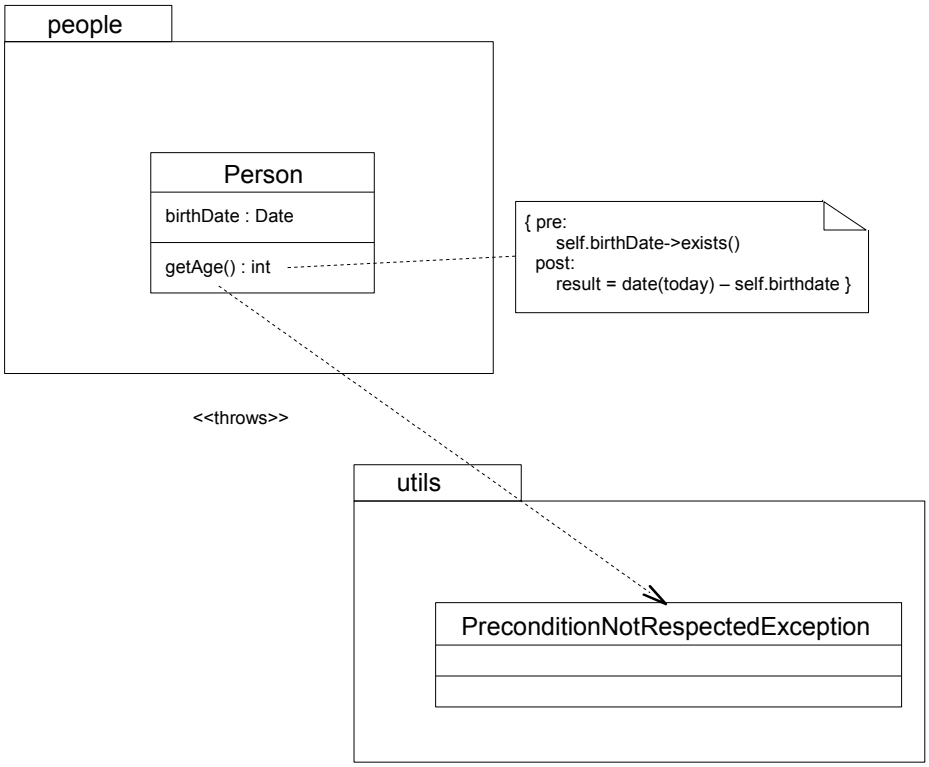
The number of tests executed and their result are recorded in the `TestResult` class.

A last important point in this design is the `TestSuite` notion, that allows calling recursively the `run()` methods on `TestCase` classes, allowing to call globally all the tests defined.

2. A first test class

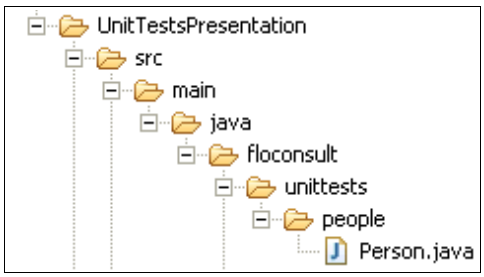
2.1. Problem description

Let us first think of a `Person` class, having a birth date as attribute, and an age calculation method:



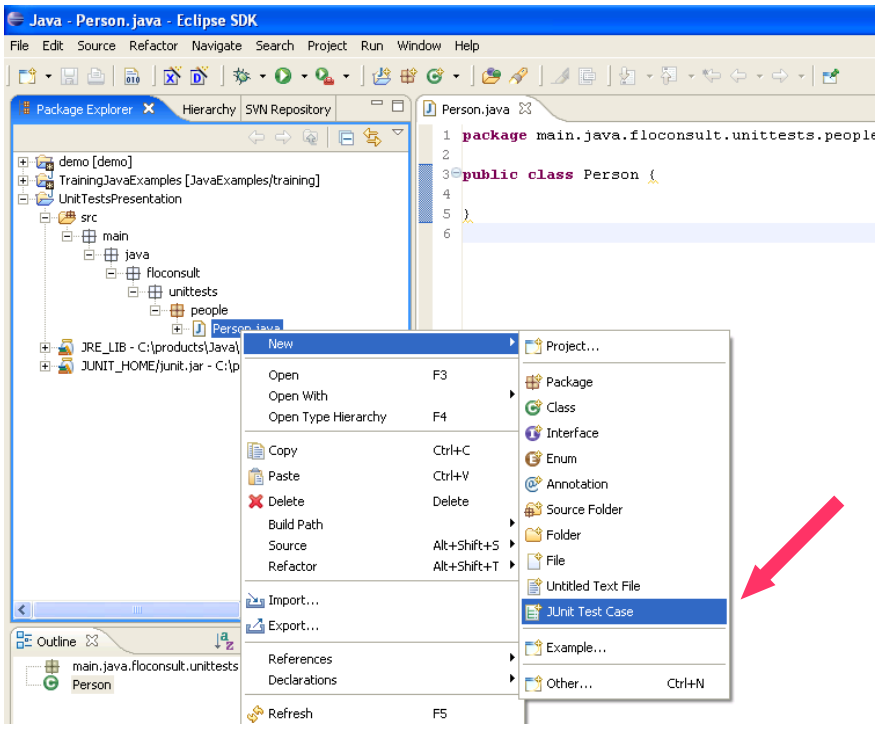
2.2. Code organisation

This class will be placed in the `people` package, itself in the `src.main.java.floconsult.unittests` packages:

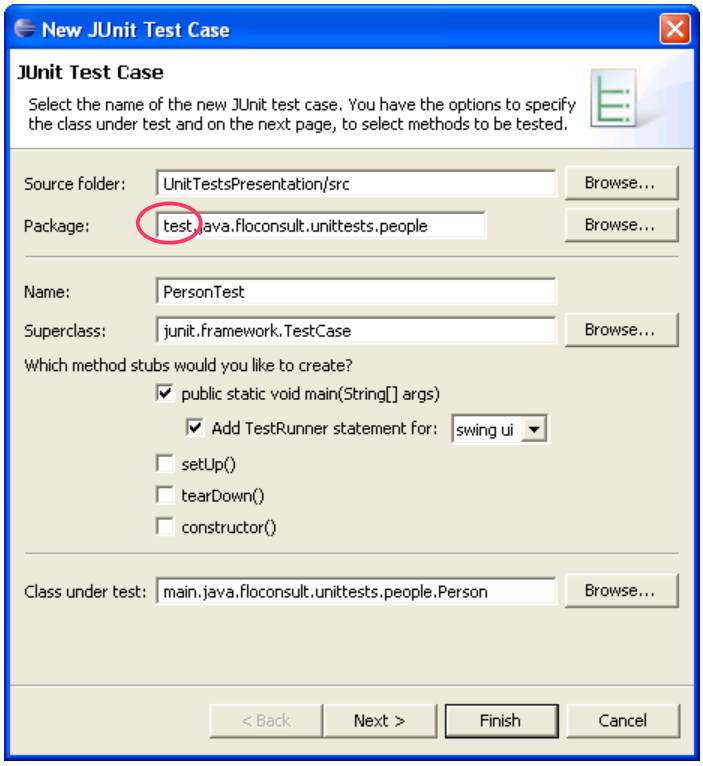


Note that the directory structure follows the recommendation of the project building tool Maven (http://maven.apache.org/maven-1.x/reference/conventions.html).

Using the JUnit enabled menus of Eclipse, we may directly create the associated test case (in the parallel package tree under `src\test`):

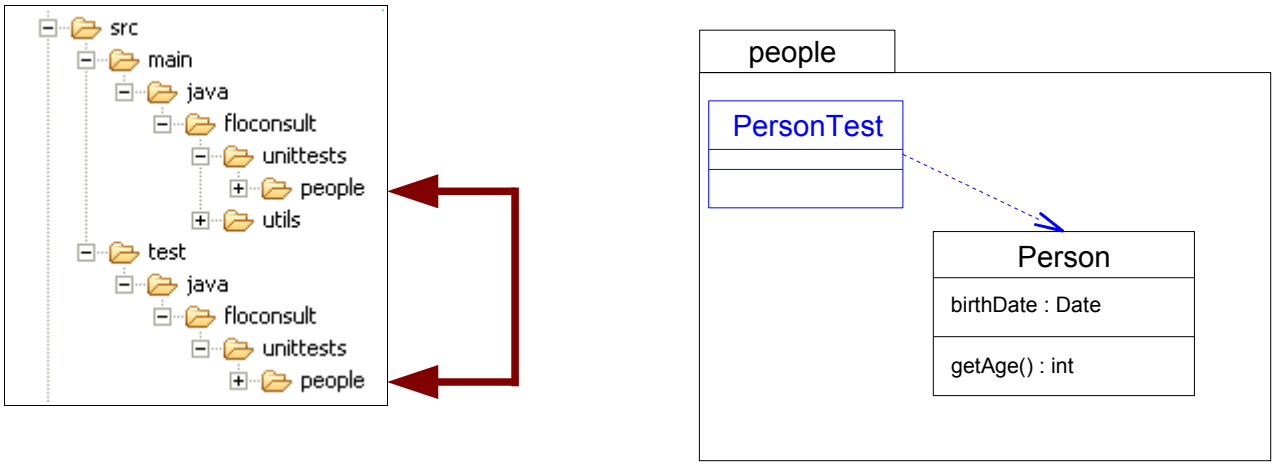


Here we chose to place the test class in the test part, and not to redefine the `setUp()` and `tearDown()` methods (the other data is left as proposed by Eclipse):



We thus have the following directory organisation in our code directory, which at the same

time allows to separate the test classes from the main ones, while placing them in the same package from the "`java`" level (here `java.floconsult.unittests.people`).



2.3. Test implementation

To run our first tests, we first have to define the tests we think of:

```
public class PersonTest extends TestCase {

    public static void main(String[] args) {

    public void testNoBirthDate() {
        Person per = new Person();
        Exception e=null;
        try {
            per.getAgeInYears();
        } catch (PreconditionNotRespectedException e1) {
            e = e1;
            assertEquals(e1.getMessage(),
                    "to get an age the birthdate must be set");  //$NON-NLS-1$
        }
        assertNotNull(e);
    }

    public void testBirthDates() throws PreconditionNotRespectedException {
        Calendar now = new GregorianCalendar();

        int awaitedAge = 30;
        Calendar theBirthday = new GregorianCalendar(
                (now.get(Calendar.YEAR) - awaitedAge), now.get(Calendar.MONTH),
                now.get(Calendar.DAY_OF_MONTH));

        Person per = new Person();
        per.setBirthDate(theBirthday);

        assertEquals(per.getAgeInYears(), awaitedAge);
    }
}
```

If no birth date is set
→ the method should return an exception

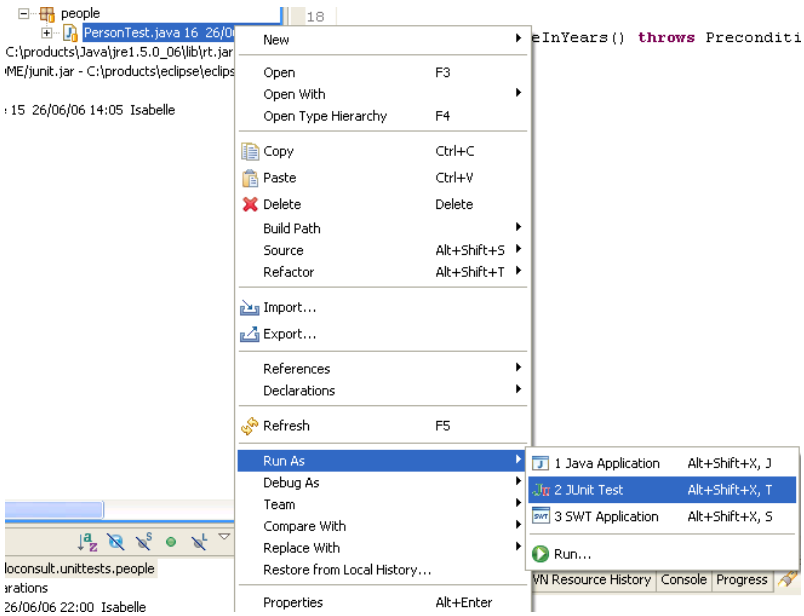
If person has a birth date such as his/her age would be 30
→ calculation correct

Then, to allow compiling the test code, let us define the skeleton (without method implementation) of the `Person` class:

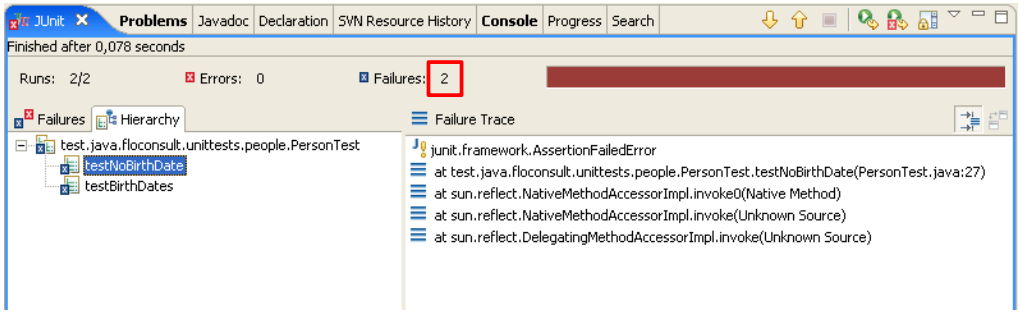
```
package main.java.floconsult.unittests.people;

import java.util.Calendar;

import main.java.floconsult.utils.PreconditionNotRespectedException;

public class Person {
    private Calendar birthDate;

    public Calendar getBirthDate() {
        return this.birthDate;
    }

    public void setBirthDate(Calendar birthDate) {
        this.birthDate = birthDate;
    }

    public int getAgeInYears() throws PreconditionNotRespectedException {
        return 0;
    }
}
```

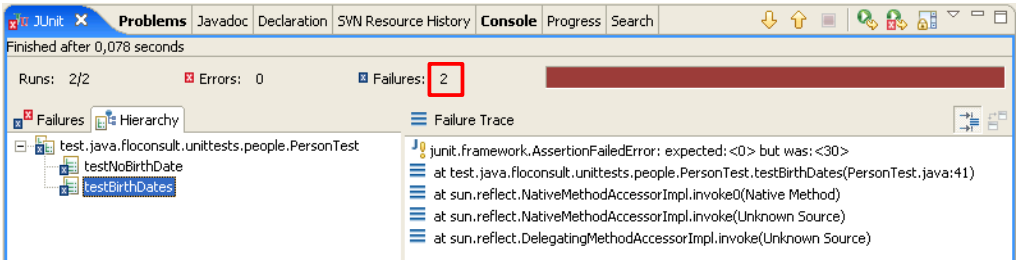
Service offered by the Person class (not yet implemented)

Running the `PersonTest` unit tests:



We have (as awaited as the `getAgeInYears()`) method is not yet implemented) both tests fail:

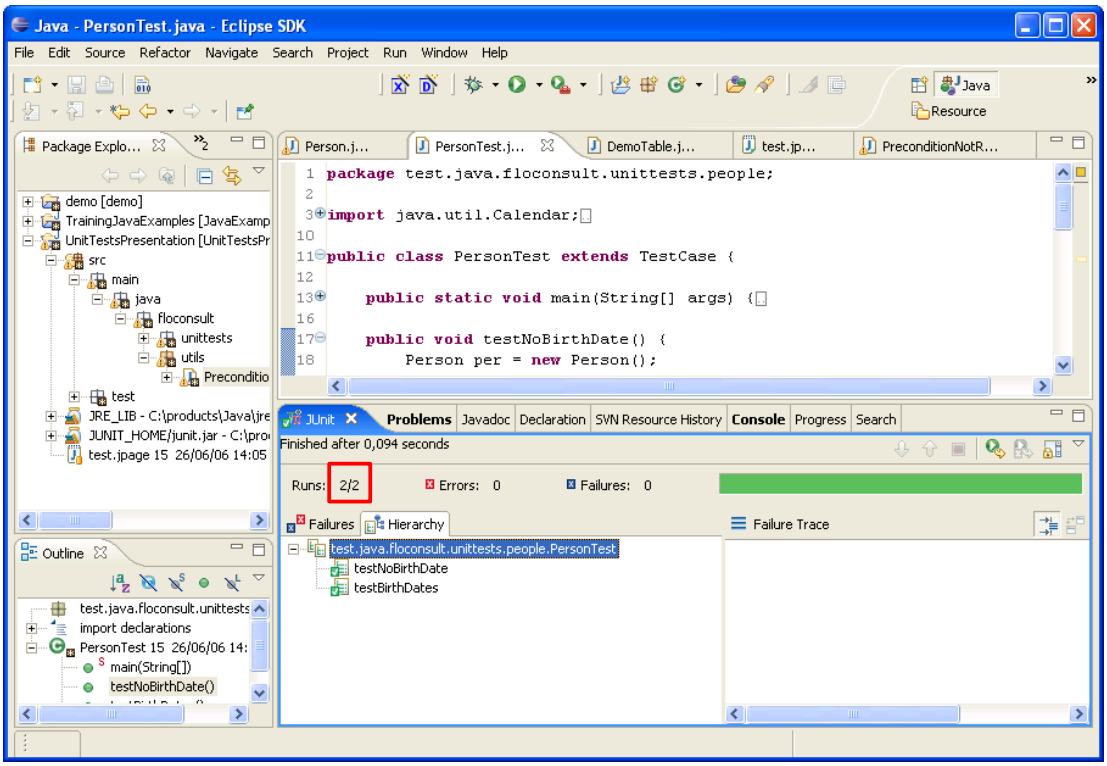




We are ready now for implementing the method:

```
public int getAgeInYears() throws PreconditionNotRespectedException {
    if (this.birthDate == null) {
        throw new PreconditionNotRespectedException(
                "to get an age the birthdate must be set"); //$NON-NLS-1$
    }

    Calendar now = new GregorianCalendar();
    Calendar bd = (Calendar) this.birthDate.clone();
    int age = now.get(Calendar.YEAR) - bd.get(Calendar.YEAR);
    bd.set(Calendar.YEAR, now.get(Calendar.YEAR));
    if (now.before(bd)) {
        age = age - 1;
    }
    return age;
}
```

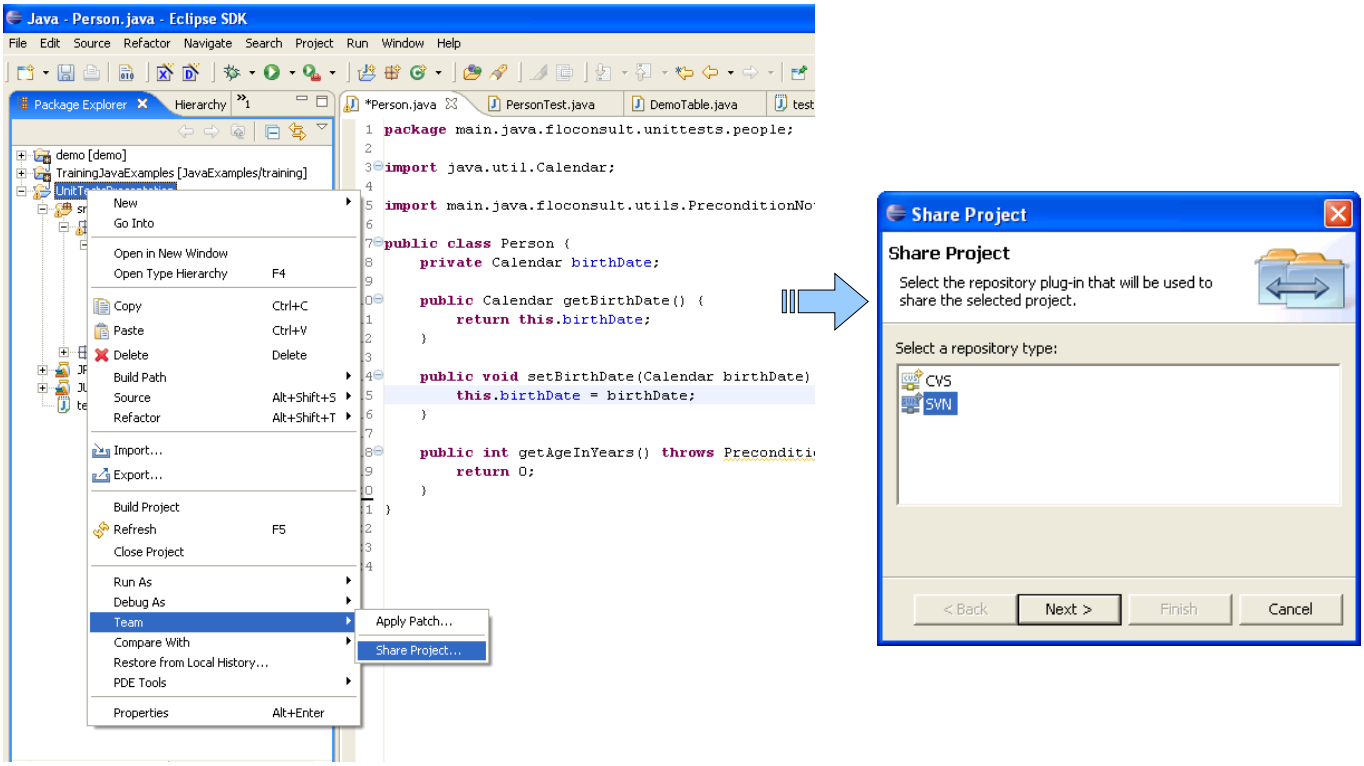
What allow us to see our first JUnit green bar, indicating the success of our 2 tests:



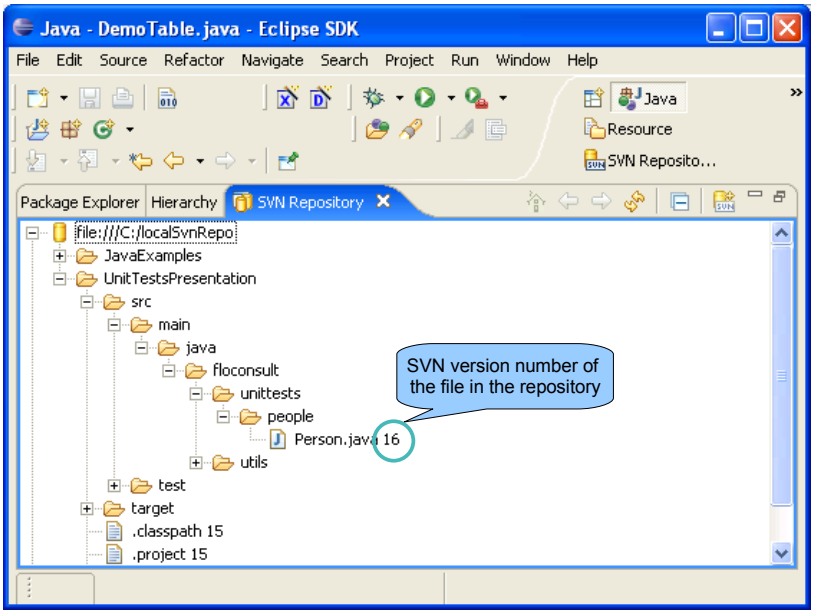
Before detailing the notion of test suites and of stubs, let us take a look at the version management of our project files using Subversion (SVN):

2.4. Managing the project versions with Subversion

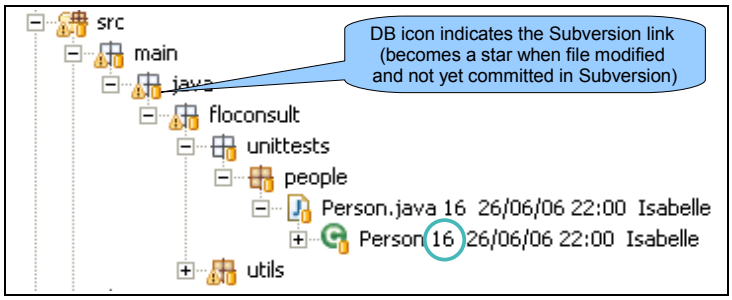
Considering we place the project in Subversion before implementing the method:



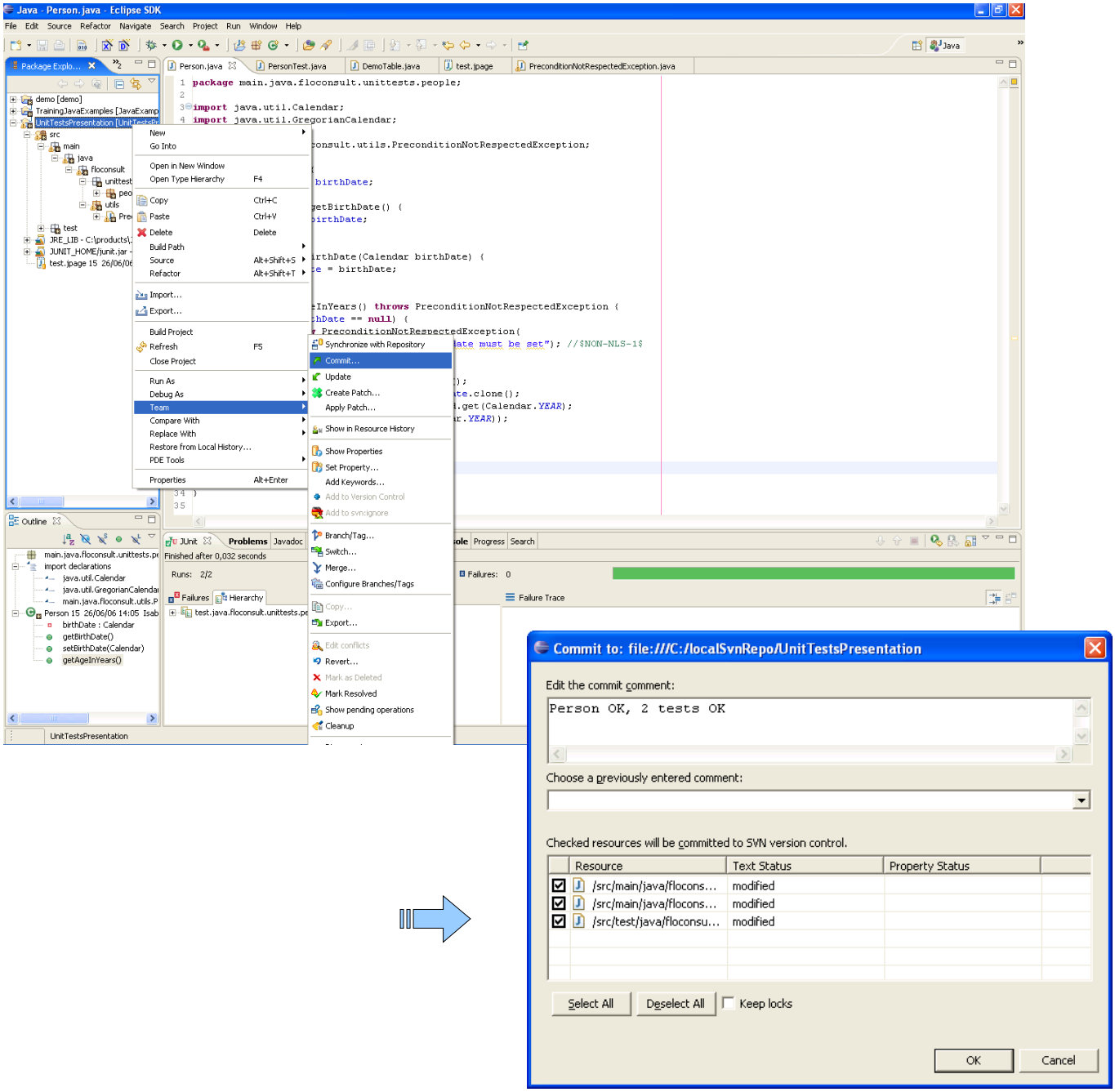
Then the user has to select the SVN repository to use (here `file:///C:/localSvnRepo/`), and confirm the project import into this repository. This done, the project may be seen in the SVN Repository view in Eclipse:



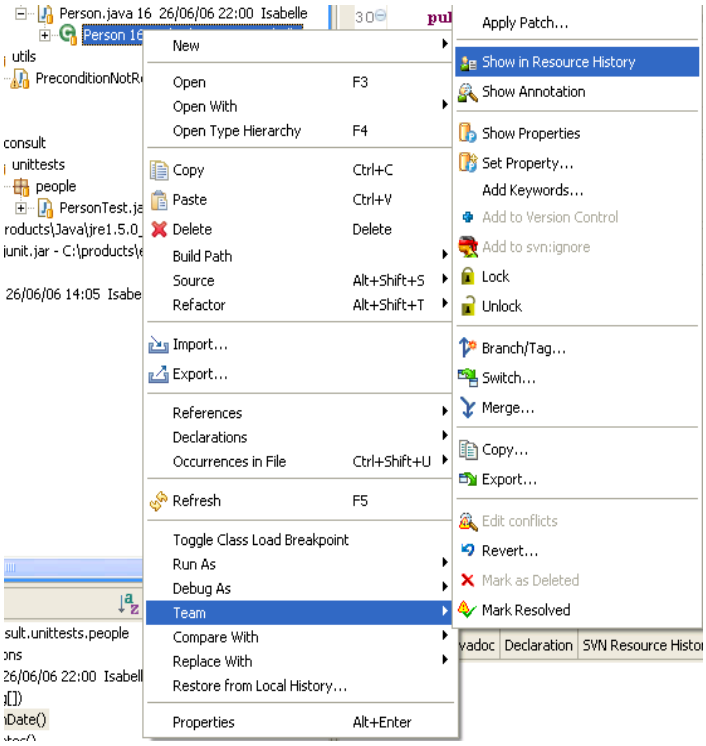
Or directly in the Package Explorer view:

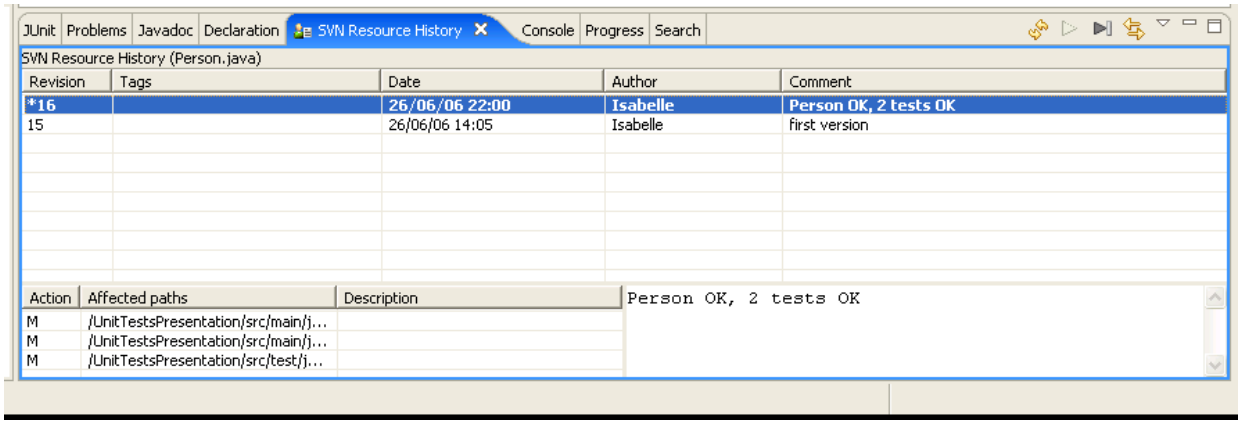


After the successful run of the tests, it is a good rule of thumb to commit the modifications in the SVN repository:

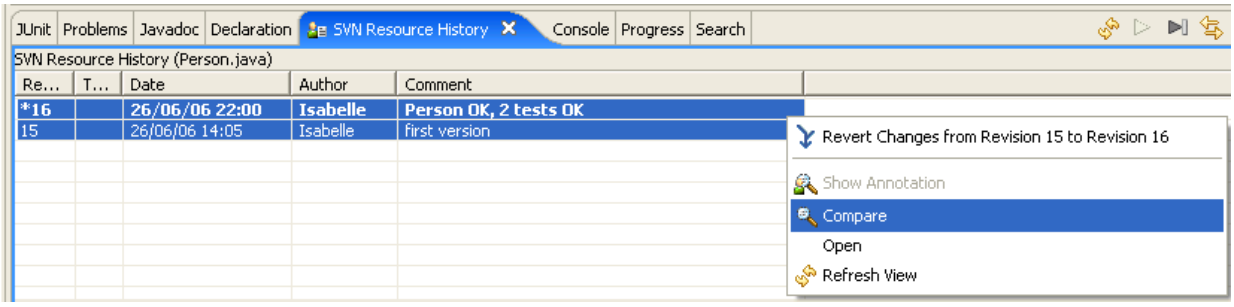


As soon as a file is managed under Subversion, we may look at its history in Subversion:





Selecting 2 versions, we may compare the files:



Java - Compare Person.java 16 and 15 - Eclipse SDK

File Edit Navigate Search Project Run Window Help

Person.java | PersonTest.java | DemoTable.java | test.jpage | PreconditionNotRespectedException.java | Compare Person.java 16 and 15

Java Structure Compare

- Compilation Unit
  - Import Declarations
  - Person
    - getAgeInYears()

Java Source Compare

Person.java 16

```
package main.java.floconsult.unittests.people;

import java.util.Calendar;
import java.util.GregorianCalendar;

import main.java.floconsult.utils.PreconditionNotRespectedE

public class Person {
    private Calendar birthDate;

    public Calendar getBirthDate() {
        return this.birthDate;
    }

    public void setBirthDate(Calendar birthDate) {
        this.birthDate = birthDate;
    }

    public int getAgeInYears() throws PreconditionNotRespec
        if (this.birthDate == null) {
            throw new PreconditionNotRespectedException(
                    "to get an age the birthdate must be se
        }

        Calendar now = new GregorianCalendar();
        Calendar bd = (Calendar) this.birthDate.clone();
        int age = now.get(Calendar.YEAR) - bd.get(Calendar.
        bd.set(Calendar.YEAR, now.get(Calendar.YEAR));
        if (now.before(bd)) {
            age = age - 1;
        }
        return age;
    }
}
```

Person.java 15

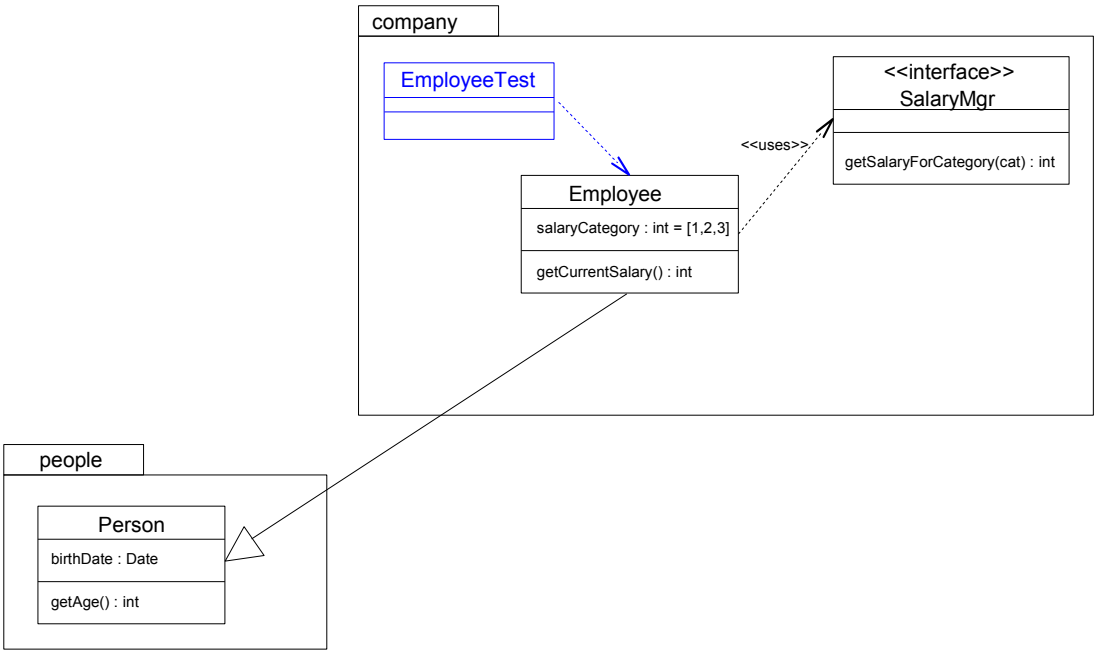
```
package main.java.floconsult.unittests.people;

import java.util.Calendar;

import main.java.floconsult.utils.PreconditionNotRespec

public class Person {
    private Calendar birthDate;

    public Calendar getBirthDate() {
        return this.birthDate;
    }

    public void setBirthDate(Calendar birthDate) {
        this.birthDate = birthDate;
    }

    public int getAgeInYears() throws PreconditionNotRe
        return 0;
    }
}
```

This to demonstrate the very nice integration of Subversion in Eclipse (using the plug-in given in #1.1.Introduction).

3. Stubs

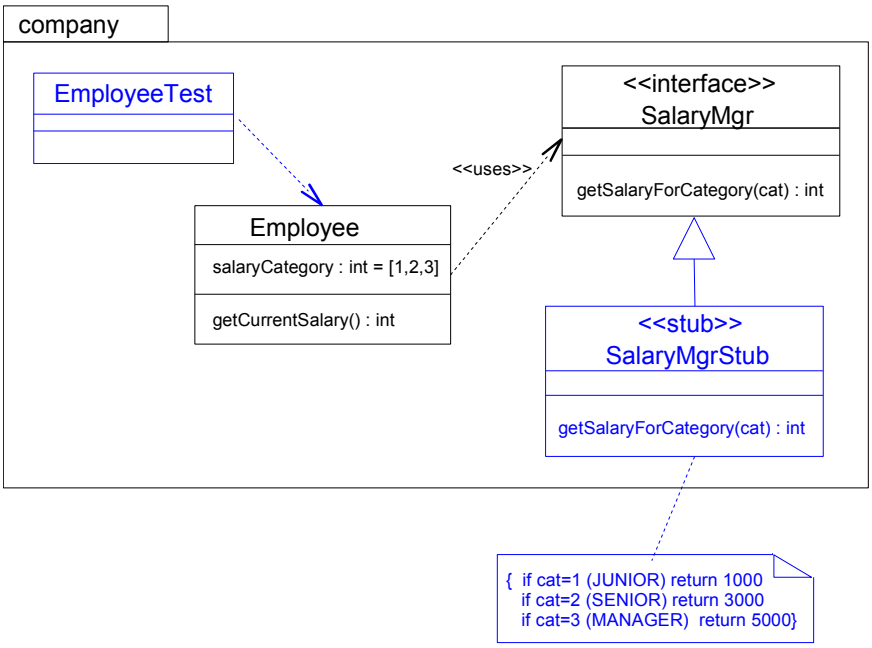
3.1. A second package

Let us now consider a package "`company`", with an `Employee`, being a `Person`, and having a salary category, which, allows to calculate his/her salary:

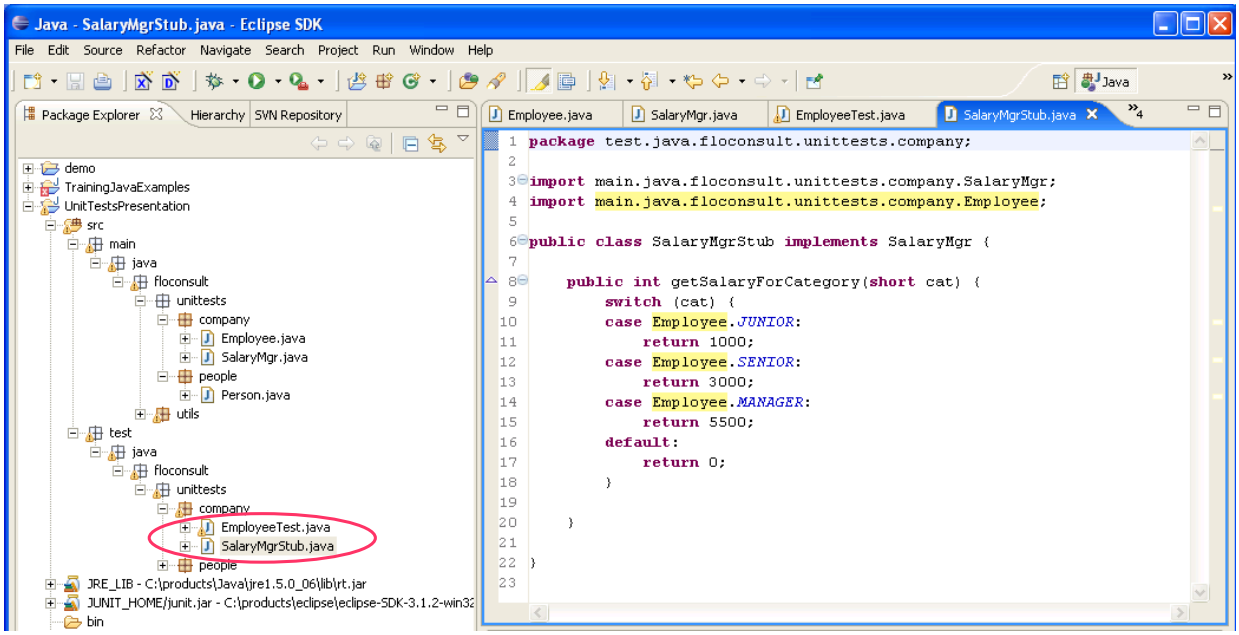


The calculation of the salary needs a class implementing the interface `SalaryMgr`, which is not detailed here.

To unit test the `Employee` class, we cannot afford having a dependence on an external unknown class, and we thus have to introduce an implementation of `SalaryMgr`, having a perfectly predictable behaviour. This is a "**Stub**" as represented hereunder:



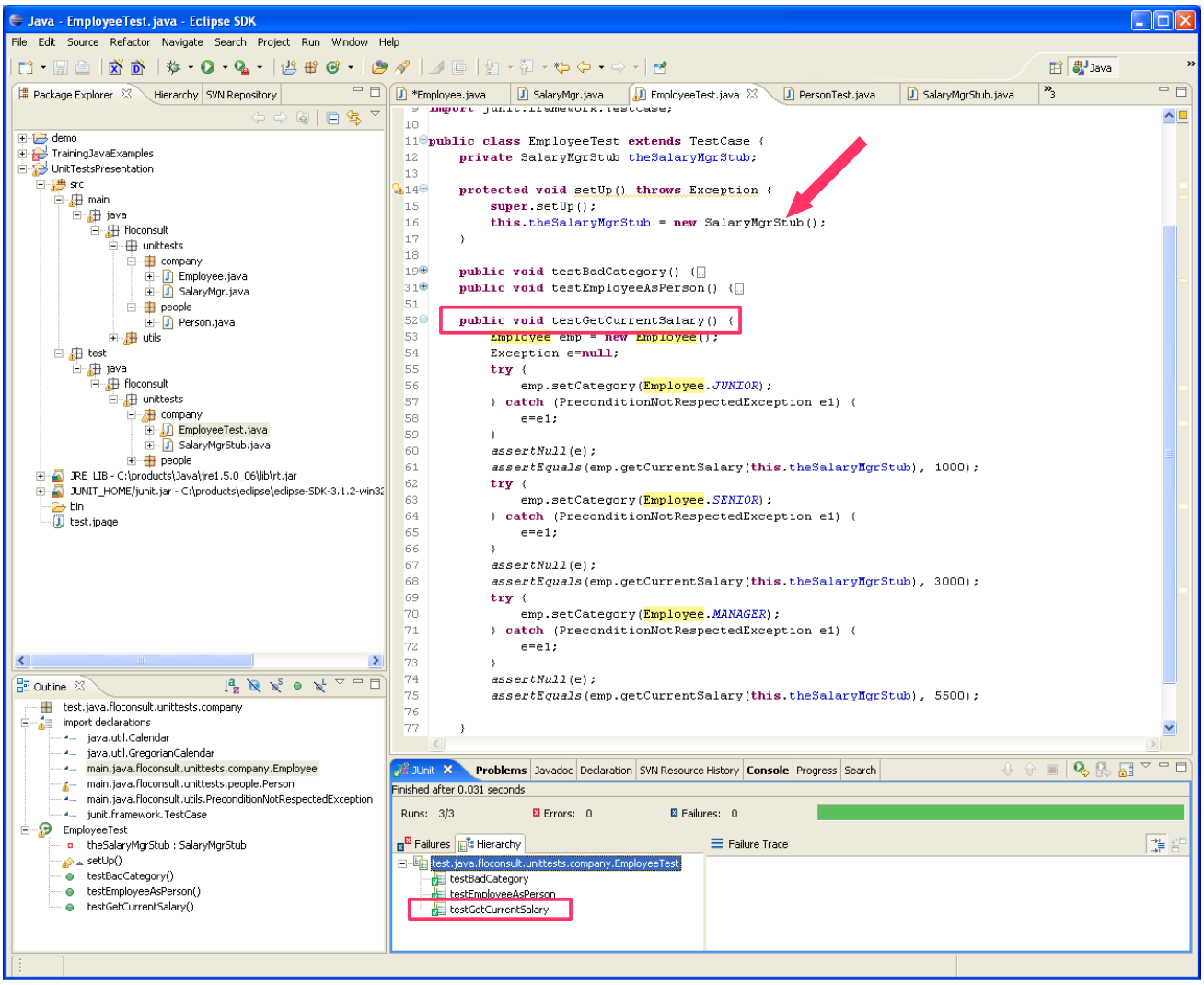
We will place this stub in the test part of the project directory, in the company package:



What allow us to test the `getCurrentSalary()` method of the `Employee` class:

```
public class Employee extends Person {
    public static final short JUNIOR = 1;
    public static final short SENIOR = 2;
    public static final short MANAGER = 3;

    private short category = Employee.JUNIOR;

    public short getCategory() {

    public void setCategory(short category) throws PreconditionNotRespectedException {
        if ((category != JUNIOR) && (category != SENIOR) && (category != MANAGER)) {
            throw new PreconditionNotRespectedException(
                    "Employee may only be JUNIOR/SENIOR/MANAGER"); //$NON-NLS-1$
        }
        this.category = category;
    }

    public int getCurrentSalary(SalaryMgr theMgr) {
        return theMgr.getSalaryForCategory(this.category);
    }
}
```

This Stub is instantiated in the `SetUp()` method of the `EmployeeTest` class (recall that this method is called before each test execution), and passed to the `getCurrentSalary()` method to allow testing it:



Obviously, in this very simple example, `Employee` does not add any behaviour to the one of the `SalaryMgr`, but this technique of stubbing is applicable to any situation were we want our classes, and thus our unit tests, to be independent of external (unpredictable) behaviours.

4. Test suites

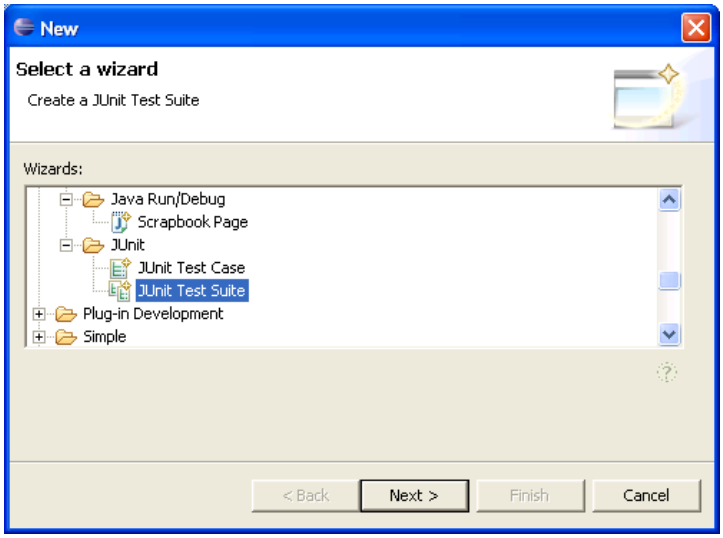
4.1. Grouping unit tests in suites

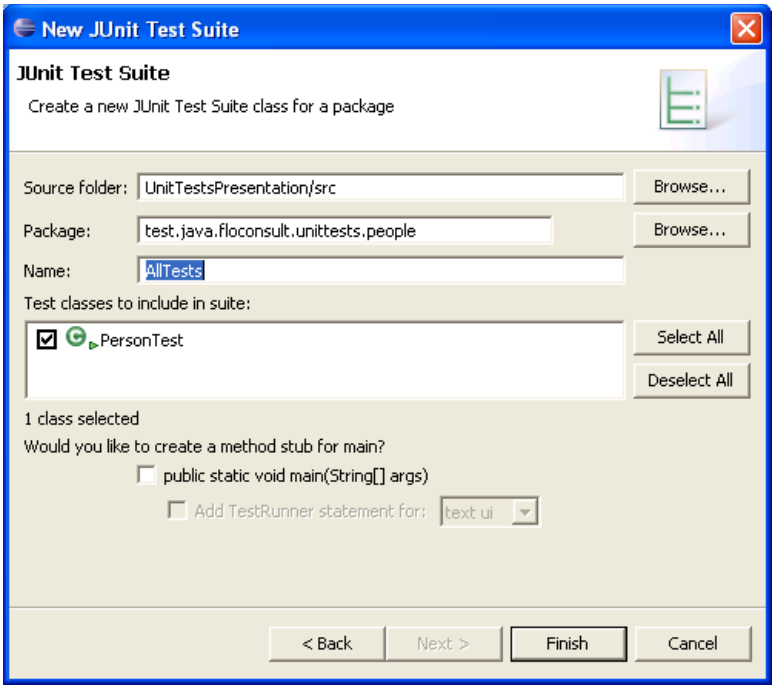
A last important notion in this introduction to unit testing with Java and Eclipse is the notion of test suites.

In the previous paragraphs, we successively tested the `Person` class of the `people` package (2 tests), and the `Employee` class of the `company` package (3 tests).

As said in the introduction, unit tests may be combined in suites, allowing to (re-)execute all the existing unit tests, and to verify that the system did not “regress” following modifications or additions in the code.

Again, Eclipse offers a wizard to ease the creation of test suites. First, let us create the test suites at the `people` level:





This suite class executes all the test classes in the package:

```
public class AllTests {

    public static Test suite() {
        TestSuite suite = new TestSuite(
                "Test for test.java.floconsult.unittests.company");
        //$JUnit-BEGIN$
        suite.addTestSuite(EmployeeTest.class);
        //$JUnit-END$
        return suite;
    }
}
```

We similarly create a test suite at the company level, and then a “parent” test suite at the higher level, which call the lower-level test suites.

Executing this suite runs the 5 tests at once: