JAVA ACADEMY - XIDERAL MONTERREY, N.L

EXERCISE - WEEK 3 BUILDER PATTERN

Made by:
Luis Miguel Sánchez Flores

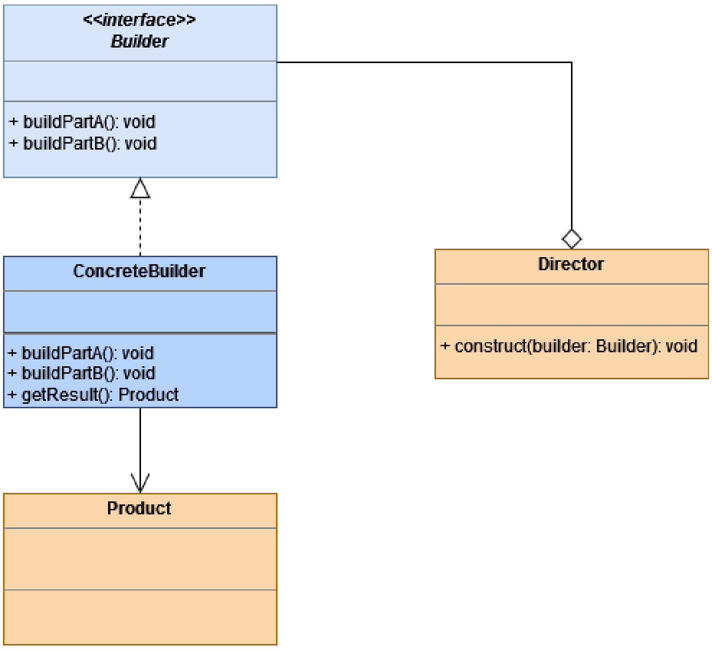
August 31st, 2024

INTRODUCTION:

The Builder design pattern is a creational pattern that streamlines the construction of objects by providing a simple and step-by-step process to establish both essential and optional attributes, as well as improve the readability of the code snippet by explicitly stating which components are being added to the object. The Builder pattern separates the construction of a complex object from its representation.

The Builder pattern usually contains the following elements:

- **Builder:** Offers an interface that declares the construction steps for creating the Product's components
- **Concrete Builder:** In charge of defining a specific implementation of the steps used to create the end product. Provides an interface for saving the product.
- **Product:** The final object or end product created by the concrete builder.
- **Director:** Constructs the product by establishing a specific order of the construction steps to create particular products. A Director is not required to be implemented



The Builder pattern is commonly used in scenarios where the creation of complex objects involves multiple steps or where the creation process needs to be flexible and customizable.

For example, universities are beginning to offer more flexible and customizable class schedules in the middle of students' careers to give students more control over course selection and completion timelines so that they are satisfied and engaged with their studies, especially those who have other commitments, whether work, family or personal, helping them to maintain high academic performance and increase their chances of success.

The use of the Builder pattern in this case can streamline the process of creating a flexible semester program, improving the experience for both the student and the administrators, by adapting a modular, step-by-step process to build flexible programs in an incremental and seamless manner, ensuring that each of the components that are integrated into the schedule are essential for the corresponding student in a given period of his or her college career, creating a flexible and modular schedule that can accommodate various course combinations.

JAVA EXAMPLE

A Java program was developed in which the Builder pattern is applied for the previous university case in order to build flexible student schedules, starting with the **ScheduleBuilder** class declared as abstract for the creation of the necessary components for a **ClassSchedule** object:

```java
// BUILDER
// Abstract class that declares the necessary components to create a ClassSchedule:

public abstract class ScheduleBuilder {

    int semester;
    ArrayList<Course> courses;
    int totalCredits;

    public abstract ScheduleBuilder addCourse(Course c);
    public abstract ScheduleBuilder setCredits();
    public abstract boolean isValidSchedule();
    public abstract int getMinimumCredits();

    public ScheduleBuilder(int semester) {
        this.semester = semester;
        courses = new ArrayList<>();
        totalCredits = 0;
    }

    public ClassSchedule getSchedule() {
        return new ClassSchedule(semester, courses, totalCredits);
    }


    public int getCredits() {
        return totalCredits;
    }
```

The **ClassSchedule** will behave as the **Product** of the pattern, being the complex product that would be developed from the Builder:

```java
// PRODUCT
// Represents the complex object built with the Builder pattern:

public class ClassSchedule {

    private int semester;
    private ArrayList<Course> courses;
    private int totalCredits;


    public ClassSchedule(int semester, List<Course> courses, int totalCredits) {
        this.semester = semester;
        this.courses = new ArrayList<>(courses);
        this.totalCredits = totalCredits;
    }

    // GETTERS

    public int getSemester() {
        return semester;
    }

    public List<Course> getCourses() {
        return courses;
    }
    public int getCredits() {
        return totalCredits;
    }
```

One of the **ScheduleBuilder** implementations is the **FlexibleScheduleBuilder** class, which is responsible for implementing the ScheduleBuilder methods, specifying the way in which the courses will be added to the schedule and setting the total number of credits from the courses currently in the list. A static and constant field is initialized that represents the minimum number of credits that the schedule must contain to be considered valid:

```java
// CONCRETE BUILDER
// Implements the Builder's functions to put togheter the necessary components of
// the ClassSchedule for a Student

public class FlexibleScheduleBuilder extends ScheduleBuilder {
    public static int MINIMUM_CREDITS = 8;

    public FlexibleScheduleBuilder(int semester) {
        super(semester);
    }

    @Override
    public ScheduleBuilder addCourse(Course c) {
        courses.add(c);
        return this;
    }

    @Override
    public ScheduleBuilder setCredits() {
        totalCredits = 0;
        courses.forEach(course -> totalCredits += course.getCredits());
        return this;
    }

    @Override
    public int getMinimumCredits() {
        return MINIMUM_CREDITS;
    }

    @Override
    public boolean isValidSchedule() {
```

The other classes developed for this example include the **Course** class, which serves as the basic unit for the schedule, the **Student** class with name and identification fields, and to which the schedule to be generated will be assigned by means of an **Administrator** object:

COURSE CLASS

```java
// Course class which serves as the basic unit for the schedule to generate.

public class Course {

    private String code;
    private String name;
    private int credits;

    public Course(String code, String name, int credits) {
        this.code = code;
        this.name = name;
        this.credits = credits;
    }

    // GETTERS AND SETTERS

    public String getCode() {
        return code;
    }

    public void setCode(String code) {
        this.code = code;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```java
// Overrides toString to showcase the code, name and credits of the Course object:
@Override
public String toString() {
    return "["+ code + " - " + name + " (" + credits + " credits)]";
}

// Override hashCode() and equals() for a proper comparison between two Course objects:
@Override
public int hashCode() {
    return Objects.hash(code, credits, name);
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Course other = (Course) obj;
    return Objects.equals(code, other.code) && credits == other.credits && Objects.equals
}
```

STUDENT CLASS

```java
// Auxiliary class Student to be assigned with a generated schedule:

public class Student {
    private final int ID;
    private String name;
    private ClassSchedule currentSchedule;

    private static int numStudents = 2743662;

    public Student(String name) {
        ID = ++numStudents;
        this.name = name;
    }

    // GETTERS AND SETTERS

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public ClassSchedule getCurrentSchedule() {
        return currentSchedule;
    }

    public void setCurrentSchedule(ClassSchedule currentSchedule) {
        this.currentSchedule = currentSchedule;
    }
```

ADMINISTRATOR CLASS

```java
// Auxiliary class that simulates an Administrator of the university:

public class Administrator {

    // Checks if the schedule created by the builder is valid in order to assign it to the student:
    public void assignSchedule(Student student, ScheduleBuilder sb) {

        // Check if the schedule has the minimum amount of credits to be accepted:
        if (sb.isValidSchedule()) {
            System.out.println("The given schedule is valid! Assigning it to student " + student.getN

            // Retrieve the schedule of the builder...
            ClassSchedule schedule = sb.getSchedule();

            // ..then set it up for the student:
            student.setCurrentSchedule(schedule);

        }
        // Otherwise, print out the following error:
        else {
            System.err.println("The given schedule is invalid. Please add more classes to meet the mi
            System.err.println(sb.getCredits() + " (SCHEDULE) < " + sb.getMinimumCredits() + " (MIN.

            
        }
    }
}
```

APPLYING UNIT TESTING

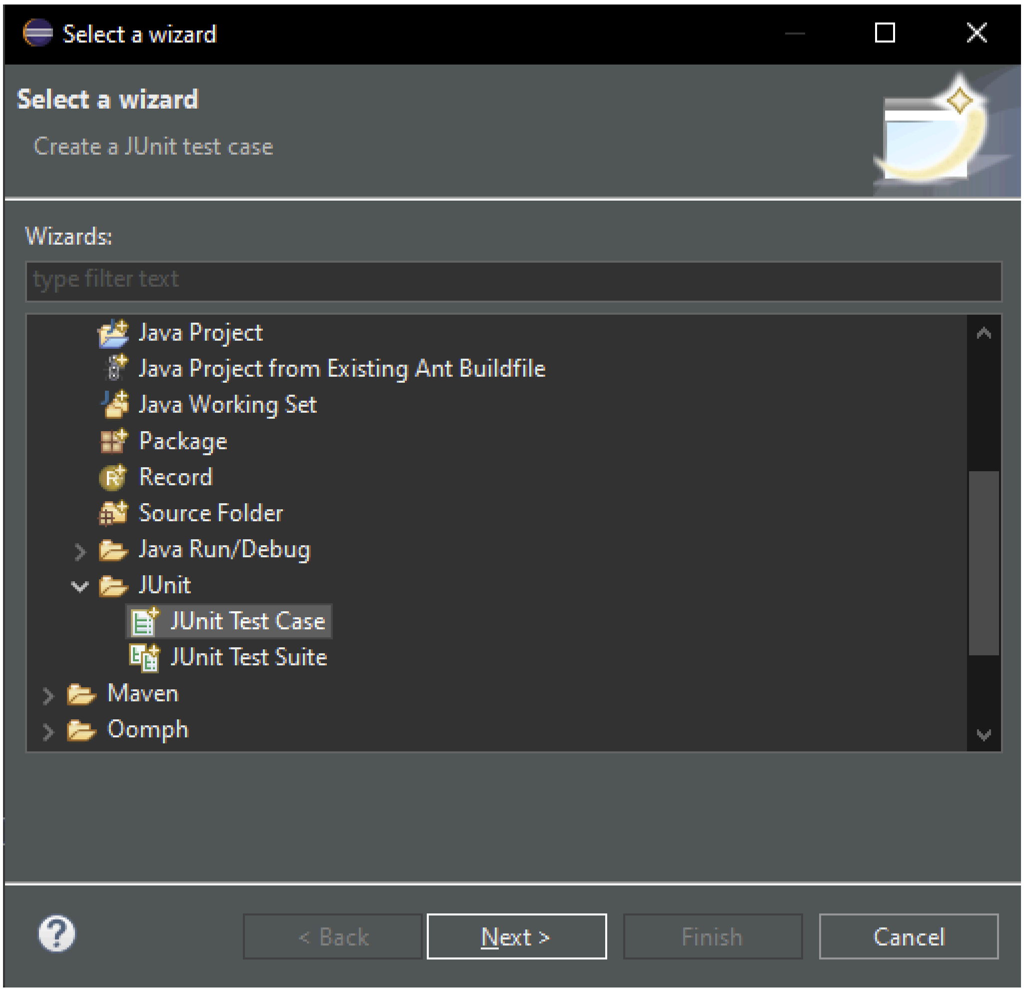
At first glance, the code developed may look very good and perform its functions/operations correctly. However, most if not all code may contain subtle bugs or edge cases that aren't immediately apparent, which clever bad actors could exploit to cause damage or catastrophic problems.
Likewise, what was once decent code can become outdated and problematic when imminent updates begin to arrive, resulting in a costly maintenance process.

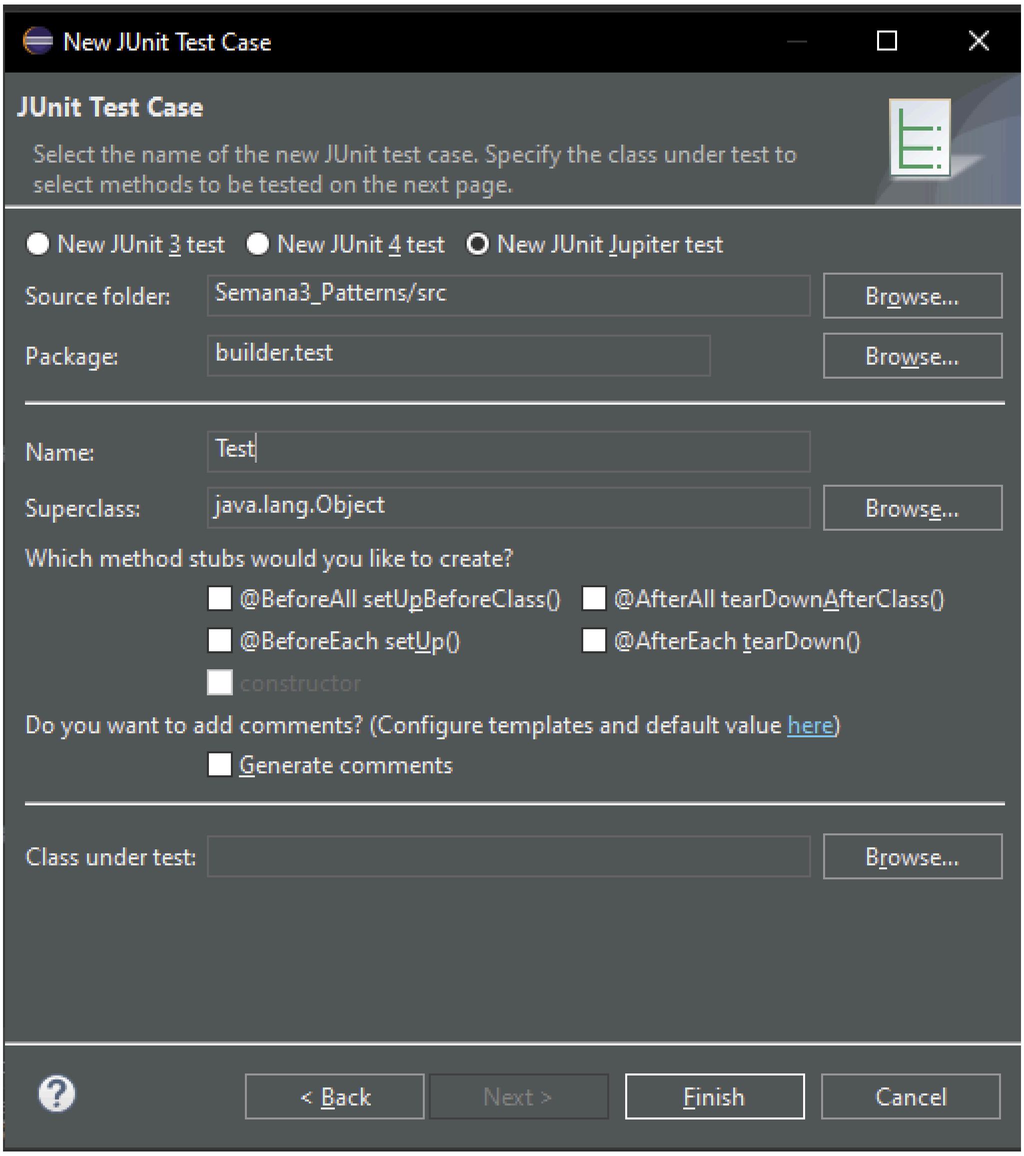
Because of this, there are several practices and tools that programmers can implement to ensure that the code being developed is of high quality and reliable both in the development process and in the maintenance of the program. One of these methods is known as **unit testing**.

Unit testing, as the name implies, involves testing individual pieces of code, such as cycles, conditions, functions or entire classes, ensuring that each piece of code developed works as expected. Unit testing offers benefits such as:

- Verify that individual components function correctly
- Catch bugs early in the development cycle
- Serve as documentation for the code's intended behavior
- Simply refactoring and code changes.

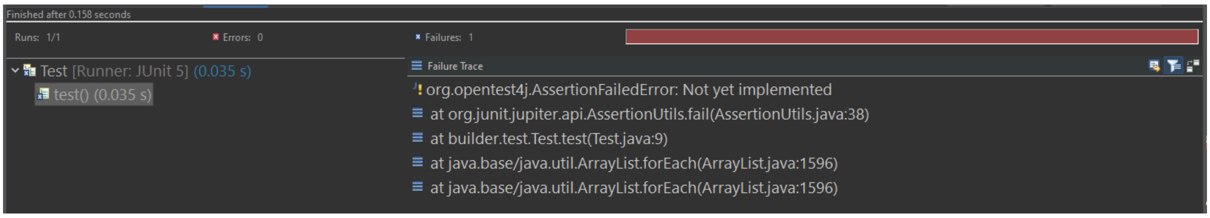
In the Java language, the *de facto* tool for applying unit tests is **JUnit**, which contains sophisticated functionalities that ensure the correct execution of the codes through test cases and/or test suites that can be developed automatically and easily.
JUnit can be imported as a dependency in a Maven or Gradle project.
Some IDEs already integrate JUnit to start generating unit tests quickly. In the case of Eclipse, just select the JUnit option in the wizard windows, either as part of a new project or within an existing project:


Select a wizard
Select a wizard
Create a JUnit test case
Wizards:
type filter text
Java Project
Java Project from Existing Ant Buildfile
Java Working Set
Package
Record
Source Folder
Java Run/Debug
JUnit
JUnit Test Case
JUnit Test Suite
Maven
Oomph
< Back
Next >
Finish
Cancel



Automatically, Eclipse will generate a new java file that imports utility methods known as **assertions** to test the code's functionality. It also integrates a test method (indicated with the @Test annotation) that will demonstrate whether the test was successful or not, once executed:

```java
package builder.test;

import static org.junit.jupiter.api.Assertions.*;

class Test {

    @org.junit.jupiter.api.Test
    void test() {
        fail("Not yet implemented");
    }

}
```



The power of JUnit is leverage to verify the different functions of the classes, especially those in the concrete Builder that is FlexibleScheduleBuilder, work as they should. For example, we checked if the Builder manages to add the course to the list, if it manages to correctly generate the schedule based on the classes selected, and assure that incomplete schedules are rejected by the minimum amount of credits required:

```java
// Test if the ScheduleBuilder is able to add the course into the list correctly:
@Test
void testAddedCourse() {
    System.out.println("TEST #1 - TEST ADD COURSE");
    Course courseToAdd = COURSE_CATALOG.get("CS301");

    ClassSchedule cs = new FlexibleScheduleBuilder(1)
                          .addCourse(courseToAdd)
                          .getSchedule();

    assertEquals(courseToAdd, cs.getCourses().get(0));

    System.out.println("TEST #1 - TEST ADD COURSE FINISHED\n");
}
```

```java
// Test if the ScheduleBuilder provides the correct ClassSchedule, based on
// the provided courses shown below:
@Test
void testClassScheduleGeneration() {
    System.out.println("TEST #2 - GENERATE CLASS SCHEDULE");

    List<Course> testCourses = getTestCourses(
            "ART310",
            "MUS121"
    );

    ClassSchedule expectedSchedule = new ClassSchedule(3, testCourses, 5);

    ClassSchedule cs = new FlexibleScheduleBuilder(3)
                          .addCourse(COURSE_CATALOG.get("ART310"))
                          .addCourse(COURSE_CATALOG.get("MUS121"))
                          .setCredits()
                          .getSchedule();

    assertEquals(expectedSchedule, cs);
    System.out.println("TEST #2 - GENERATE CLASS SCHEDULE FINISHED\n");
}
```

```
// Test out if the created schedule DOES NOT satisfy the min. amount of credits required:
@Test
void testLackOfMinimumCredits() {

    System.out.println("TEST #3 - INSUFICIENT AMOUNT OF CREDITS");

    boolean isValid = new FlexibleScheduleBuilder(0)
                            .addCourse(COURSE_CATALOG.get("MUS121"))
                            .addCourse(COURSE_CATALOG.get("ART310"))
                            .setCredits()
                            .isValidSchedule();

    assertFalse(isValid);

    System.out.println("TEST #3 - INSUFICIENT AMOUNT OF CREDITS FINISHED\n");

}
```

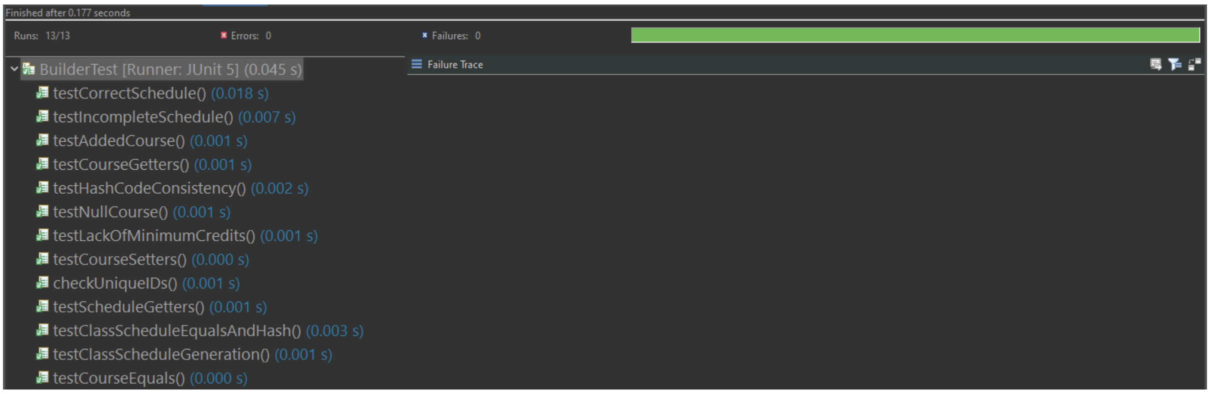
Furthermore, complex tasks were developed to see if the Administrator object is able to identify both correct and incorrect schedules, ensuring that only the valid generated schedules are assigned to the student:

```
    // Complex test to verify if the Administrator is able to assign
    // a correct schedule is assigned to the student:
    @Test
    void testCorrectSchedule() {

            System.out.println("TEST #4 - CORRECT CLASS SCHEDULE");

           List<Course> listCourses = getTestCourses(
                   "CS101",
                   "MATH123",
                   "ENG110"
                   );

           // The expected schedule to be assigned to the student:
           ClassSchedule expectedSchedule = new ClassSchedule(
                   5, listCourses, 11
           );


            // Add a new administrator:
            Administrator admin = new Administrator();

            // Create a new Student to add the flexible schedule:
            Student student = new Student("John Doe");
```

```java

        // Generate a flexible schedule builder
        ScheduleBuilder flexSchedule = new FlexibleScheduleBuilder(5)
                                        .addCourse(COURSE_CATALOG.get("CS101"))
                                        .addCourse(COURSE_CATALOG.get("MATH123"))
                                        .addCourse(COURSE_CATALOG.get("ENG110"))
                                        .setCredits();


        // Asign the schedule to the student:
        admin.assignSchedule(student, flexSchedule);

        // Print out the student's schedule:
        System.out.println("STUDENT SCHEDULE : \n" + student.getCurrentSchedule());

        // Check if the student's schedule is the one expected:
        assertEquals(expectedSchedule, student.getCurrentSchedule(), "This is an error");

        System.out.println("TEST #4 - CORRECT CLASS SCHEDULE FINISHED\n");
    }
```

```java
    // Complex test to verify if the Administrator is able to assign
    // a correct schedule is assigned to the student:
    @Test
    void testIncompleteSchedule() {

        System.out.println("TEST #5 - INCOMPLETE CLASS SCHEDULE (less than Min. amount of credits required)");

        Administrator admin = new Administrator();

        Student student = new Student("Ashley Cole");

        ScheduleBuilder flexSchedule = new FlexibleScheduleBuilder(7)
                .addCourse(COURSE_CATALOG.get("BIOL205"))
                .setCredits();

        admin.assignSchedule(student, flexSchedule);

        assertNull(student.getCurrentSchedule());

        System.out.println("TEST #5 - INCOMPLETE CLASS SCHEDULE FINISHED\n");
    }
```

Running the JUnit test file will prompt the JUnit window in Eclipse that lets developers know the status for each of the tests created. In this case, all of the tests were able to produce the expected outcomes, and mark as successful as shown below:



Alongside the unit tests, sophisticated IDEs like Eclipse can enhance the testing phase by providing additional tools such as **EclEmma** that consists of a **code coverage tool. Coverage** helps the developer in pointing out which lines of code were executed by the unit tests, and serves as a useful metric for assessing the completeness of the test suite, giving a clear picture of which code snippets have been tested.

In Eclipse, you only need to execute the "Coverage As" option at the "Run" tab, which will display a window that prints out the percentage of code that has been covered by the unit tests. In this case, more than 95% of the Builder code has been covered by the developed unit tests: