Assignment 5 - G03P02

Group information

- Ana Inês Oliveira de Barros - `up201806593@fe.up.pt`;
- João de Jesus Costa - `up201806560@fe.up.pt`

Structural Testing

Structural testing is a white-box testing technique that provides a systematic way to devise tests. The tests are based on different criteria. For this assignment, the criteria we will be focusing on are line and decision coverage.

We collected code coverage by using JaCoCo Code Coverage Library for Java. Unit tests were written using the JUnit framework.

We reached a line and branch coverage of at least 90% for all classes, except for the *GUI* package.

Previous assignments coverage

We wrote unit tests using black-box testing techniques for the previous two assignments (#2 and #3). The two images below show how much coverage these unit tests generated: an overall 5% instruction coverage and 6% decision coverage. Only the project package had been tested: 21% instruction coverage and 23% decision coverag

jTimeSched -- Time tracking application

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
de.dominik_geyer.jtimesched.gui		0%		0%	149	149	581	581	77	77	23	23
de.dominik_geyer.jtimesched.gui.table		0%		0%	63	63	248	248	29	29	9	9
de.dominik_geyer.jtimesched.project		21%		23%	77	104	237	307	38	55	2	5
de.dominik_geyer.jtimesched		0%		0%	12	12	40	40	7	7	2	2
de.dominik_geyer.jtimesched.misc		0%		n/a	2	2	6	6	2	2	1	1
Total	4,877 of 5,142	5%	277 of 297	6%	303	330	1,112	1,182	153	170	37	40

de.dominik_geyer.jtimesched.project

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ProjectSerializer		0%		0%	22	22	116	116	8	8	1	1
ProjectTableModel		24%		21%	26	34	62	82	9	11	0	1
Project		48%		50%	25	42	52	94	17	31	0	1
ProjectTime		47%		100%	3	5	5	13	3	4	0	1
ProjectException		0%		n/a	1	1	2	2	1	1	1	1
Total	985 of 1,250	21%	64 of 84	23%	77	104	237	307	38	55	2	5

As a result of this analysis, since there are no classes with both line and decision coverage at 90% or more, it was required to write unit tests for every class (except *GUI* related ones).

Unit tests

This section provides a brief description of the JUnit features explored and the result of each unit test we developed in order to increase project's code coverage.

JUnit Features

- Most tests work by comparing an expected result with a result. These types of comparisons use `assertEquals`. Other tests that require comparing a boolean value use `assertTrue` or `assertFalse`. In addition, tests where exceptions are expected, we use `assertThrows`.
- Some tests require a setup before running. For cases like this, we used `@BeforeEach` or `@BeforeAll`, depending on the situation. An example can be found on `ProjectSerializer.java`, where each test requires the creation of a temporary file and a `ProjectSerializer` instance. We also `@AfterEach` for some clean-up tasks (for example at `ProjectSerializer.java`).
- For tests that needed to be run multiple times with different values, we used `@ParametrizedTest`s along with `@MethodSource` to provide arguments to the test.
- We used `fail()` to explicitly create a failure for the `parseDateNullTest()` on `ProjectTimeTest.java`.

misc package

PlainTextFormatter class

1. **Format**
    - **Description**: Given a log record (a message, severity, and time), this test checks whether the output is formatted as intended.

Result: All tests pass successfully.

project package

Project class

- **Setup**: Before each test, a new project is created, its number of seconds today is set to 10, and the number of seconds overall is set to 30.

1. **Get elapsed seconds not running project**
    - **Description**: This test stops the project and tries to access the project's number of elapsed seconds. The test succeeds if a *ProjectException* is thrown.
2. **Start a running project**
    - **Description**: This test starts the project and attempts to start it again. The test succeeds if a *ProjectException* is thrown.
3. **Pause a paused project**
    - **Description**: This test pauses the project and attempts to pause it again. The test succeeds if a *ProjectException* is thrown.
4. **Toggle stopped project**
    - **Description**: Given a paused project, this test toggles the project and checks if the state of the project has changed. The test succeeds if a project ends up in the *running* state.
5. **Toggle running project**
    - **Description**: Given a running project, this test toggles the project and checks if the state of the project has changed. The test succeeds if a project ends up in the *paused* state.
6. **Reset today seconds**
    - **Description**: This test tries to reset a project for today. The test succeeds if it is successfully reset: the number of seconds today is 0, the quota for today is 0, and the

start time is equal to the moment the test executes.

Result: All tests pass successfully.

ProjectSerializer class

1. **Read/Write XML**
    - **Description**: This parameterized test receives different projects as input: one default project, a checked project, a project with a set color, a project with notes, and another with a null title. Then, the tests are saved in an *XML* file and recovered from the same file. The test succeeds if the projects are equal (read and write operations were correct).
2. **Write XML running project**
    - **Description**: Creates a new project, sets it as running and saves it to the XML file. Then, the project is read from the same XML file. The test succeeds if the project is the same and not running (running projects are saved as not running).
3. **Read XML running project**
    - **Description**: Creates a new project, sets it as running and saves it to the XML file. Then, the XML file is edited to set the project running and we proceed to read the project from the same XML file. The test succeeds if the project is the same and running.
4. **Read/Write XML project with no quota**
    - **Description**: Creates a new project, saves it to the XML file, and edits the file in order to remove the line corresponding to the project's quota. Then, the project is read from the same XML file. The test succeeds if the project is the same.
5. **Read/Write XML project no notes**
    - **Description**: Creates a new project, saves it to the XML file and edits the file in order to remove the line corresponding to the project's notes. Then, the project is read from the same XML file. The test succeeds if the project is the same.

Result: All tests pass successfully.

ProjectTableModel class

1. **Set value**
    - **Description**: The test starts by creating a *ProjectTableModel* that contains a single project. Then, this parametrized test receives different values as inputs for each settable project value (title, date, color, etc.) and checks if the value was correctly set.
2. **Set value at invalid column**
    - **Description**: The test starts by creating a *ProjectTableModel* that contains a single project. Then, it tries to set to true the value of the *STARTPAUSE* column. The test succeeds if the value is not set (these columns are not settable).
3. **Add project**
    - **Description**: The test starts by creating a *ProjectTableModel* that does not contain any project. Then, it tries to add a new project to the table. The test succeeds if the project is successfully added.
4. **Remove project**
    - **Description**: The test starts by creating a *ProjectTableModel* that contains a single project. Then, it tries to remove the project from the table. The test succeeds if the project is successfully removed.

5. **Get column class**
    - **Description**: The test starts by creating an empty *ProjectTableModel*. Then, this parametrized test receives inputs for each column on the table and corresponding class. The test succeeds if, for each column, the class type present on the table is correct (e.g., the title column should be *String*). Additionally, the value should be _String_for an invalid column.
6. **Is cell editable**
    - **Description**: The test starts by creating a *ProjectTableModel* that contains a single project. Then, this parametrized test receives different inputs consisting of different columns and an indication of whether the column is editable or not. The test succeeds if each column is editable or not, as expected.
7. **Column count**
    - **Description**: The test starts by creating an empty *ProjectTableModel*. Then, it checks the number of columns of the table. The test succeeds if the number of columns equals the expected (8 columns).
8. **Column names**
    - **javaDescription**: The test starts by creating an empty *ProjectTableModel*. Then, this parametrized test receives different inputs consisting of different columns and the corresponding names. The test succeeds if, for each column, its name on the table is correct.

Result: All tests pass successfully.

ProjectTime class

1. **Parse date**
    - **Description**: The test creates a date as a formatted *String* and attempts to parse it. The test succeeds if it correctly parses the date.
2. **Parse null date**
    - **Description**: The test creates a null date and attempts to parse it. The test succeeds if it throws a *NullPointerException*.

Code coverage achieved

The following image contains the final results of our work. We successfully achieved 89% of line coverage and 90% of branch coverage.

jTimeSched -- Time tracking application

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
de.dominik_geyer.jtimesched		8%		10%	10	13	37	42	5	8	1	2
de.dominik_geyer.jtimesched.project		99%		100%	0	104	5	307	0	55	0	5
de.dominik_geyer.jtimesched.misc		100%		n/a	0	2	0	6	0	2	0	1
Total	147 of 1,436	89%	9 of 94	90%	10	119	42	355	5	65	1	8

The lines not covered by our test are related to unreachable *catch* blocks in *try/catch* statements for `ProjectException`. This exception happens when the program tries to apply some action on a project that should be running, but it is not (or vice-versa). An example of this case can be seen in the code snippet below:

```
public int getSecondsToday() {
        int seconds = this.secondsToday;

        if (this.isRunning())
            try {
                seconds += this.getElapsedSeconds();
            } catch (ProjectException e) {
                e.printStackTrace();
            }

        return seconds;
    }
```

The other uncovered lines are on the main function of the project.