Lucia Picos Maiztegui

Book inventory

Introduction

In this Maven project is implemented a Book inventory for a bookshop which wants to know how many books they have in stock, how much money they earn selling their books.

To do this the starting point was creating the Maven project in Eclipse and creating a git repository connected to a GitHub repository. Once that was set up, we could start implementing the first classes of our project.

Implementation and Unit Tests

For this project we implemented a class Book, with every identification tool needed to know the different books there are in a bookshop and some characteristics that will be of use later. We also implemented a class BankAccount that serves to keep the balance of the bookshop. At last, the class GestInventory is created to manage every move that can be made.

In this class there are various functions that implement the inventory, a function for new books that get added into the inventory, books that are removed, books that are already in the inventory, but the bookshop buys more copies of it and what happens if someone buys a book.

To implement all these functions and using TDD, unit tests were being created before the implementation of the functions in a class GestInventoryTest. Each test is carefully done to avoid missing code coverage percentage.

Code coverage and mutation tests

The code percentage is registered online in Coveralls. But to see it off-line we use the JaCoCo reports explained in this subject's book. In these reports, the code from the class Book is excluded because it doesn't need to be tested 100% (not every setter and getter).

The exact same thing happens when we try to run a PIT mutation test, the class Book gets excluded. At first running this test there were some changes that needed to be made to the unit test in order to kill every mutant.

During all these changes the pom.xml suffered different alterations in which different plugins were introduced (maven plugins, coveralls, jacoco even changes to dependencies). Once all unit tests were correctly done, we started using continuous integration with Github Actions.

Continuous Integration

For this purpose, the first yaml archive was created. In this file at first, we only added CI with Maven and the necessary things to send the reports to the Coveralls page. Then we added the dependencies and changes needed to create a profile in the pom.xml that starts a Docker container which can execute the tests. This profile work when using the verify command.

Integration tests

Starting with the integration tests the source folder IT was created even tough there were some problems with using the Build Helper Maven Plugin and in the end, I had to search for another form of getting the folder made.

In this folder the integration tests are implemented, this test can be run with the maven command verify, not the same as the test command because not every time that we made little changes the integration tests need to be checked.

During all this, different branches in our git project were made and they were merged with the master branch with Pull requests that we can check in the history of the GitHub repository.

For the time being there is a warning when building the project (Parameter `localRepository' is deprecated core expression; Avoid use of ArtifactRepository type. If you need access to local repository, switch to '${repositorySystemSession}' expression and get LRM from it instead) , with all my researches everyone says that this warning is a problem of the actual version of a maven plugin and for now it can not be solved.

GUI tests

The next step are GUI tests. First unit tests. For this we create a new branch GUI and we install the WindowBuilder plugin in which designing the window it is not needed to write all the code by ourselves. In these phases, we change a few things, creating an interface for the Gestion and we implement it in the new JFrame which will be our window.

First, we need to implement the window tests, for the buttons and fields that we created. After checking all those we can start to write the unit tests similar to our previous unit tests but we need a new implementation of the function in the Swing Frame.

Maven commands

At last, to implement SonarCloud we need to create a new yaml file, because SonarCloud needs Java 11 and our first yaml file implements Java 8. We add this to our continuous integration characteristics, creating the secret token in our Github repository and making sure that the needed report files are sent to the SonarCloud web page.

The project can be run with different Maven commands or with only one if we make it really big or if we just want to check something specific. Different Maven command will activate different sections of the project. The build of the project can be made with a simple mvn clean install which will generate some needed files and install some dependencies if there are not already in our cache.

To create the Jacoco files we can use the command: mvn clean install verify test jacoco:report . If we want to run the tests with Docker the Docker profile with a Maven command: mvn clean install -Pdocker test. If we want to check the PIT mutation test we can also do it with a maven command: clean install test org.pitest:pitest-maven:mutationCoverage and we can check that every mutant is killed.