Università degli Studi di Firenze

Advanced Programming Techniques - Project Report

Federico Palai, Ubaldo Puocci

Anno Accademico 2019/20

Contents

1 Introduction

This project is a simple example to show how to implement an application following the practices draw by the TDD method.

2 System Description

The application developed is a simple agenda for schools. In particular is able to store information about the students and the course thought and the relation between them. It also offer the opportunity to insert a new student and/or course and change the relations between them. Following the complete list of operations the application supports:

- Insert a new student
- Insert a new course
- Associate a course to a student
- Associate a student to a course
- Remove a student
- Remove a course
- Disassociate a course from a student
- Disassociate a student from a course

The final user can interact with the application choosing between a CLI (Command Line Interface) or a GUI (Graphical User Interface).

3 Technologies and Framework used

The TDD has been supported by several tools among the ones saw in the course. Following is described the developing environment used for this project:

- *Ubuntu 20.04 LTS*: the OS used entirely for the development
- *Eclipse 2020-06*: the IDE used
- *Java 8*
- *JUnit 4*: the library used to perform tests both Unit and Integration tests
- *AssertJ*: library used for insert assertions within the tests
- *JaCoCo* (and the *EclEmma* plugin for Eclipse): tool used for the Code Coverage

- *PIT* (and the *Pitclipse* plugin for Eclipse): framework used for the Mutation Testing
- *Maven* (and the *M2Eclipse* plugin for Eclipse): tool used for the build automation of Java projects
- *Mockito*: framework used for mocking dependencies during Unit tests
- *Git* (and the *GitKraken* interface for Git)
- *GitHub*: used for hosting the project and for CI integration
- *Docker*: used for virtualize the MongoDB Database in a container
- *Swing* (and the *WindowBuilder* plugin for Eclipse): toolkit used for developing the graphic interface in Java
- *AssertJ Swing*: framework used for testing the graphic interface build with Swing
- *Travis CI*: server CI used for build the application remotely
- *SonarQube* (and the *SonarLint* plugin for Eclipse): platform used for inspect the code e estimating the code quality. It can be used alongside JaCoCo to obtain also the code coverage
- *SonarCloud* (remote interface for *SonarQube*)
- *MongoDB*: non relational DBMS, based on document

4 System Description

The basic functionalities we wanted to implement in the application were:

Create New Student

The user can fill a form with the student name, the student ID and create a new profile for that student. The system will create the corresponding object and initialize it without courses associated.

Remove a Student

The user can remove a student profile by using its ID. Of course this will not delete also the course(s) associated with the student.

Show the Student Courses

The user can select a student profile by using its ID and see all the courses with which he is associated.

Remove a Student Course

The user can select a student and one of his course by using the ID for both and remove that course from the list of associated course of the student.

Create New Course

The user can fill a form with the course name, the course ID, the curse CFU and create a new profile for that course. The system will create the corresponding object and initialize it without students associated.

Remove a Course

The user can remove a course profile by using its ID. Of course this will not delete also the student(s) associated with the course.

Show the Course Students

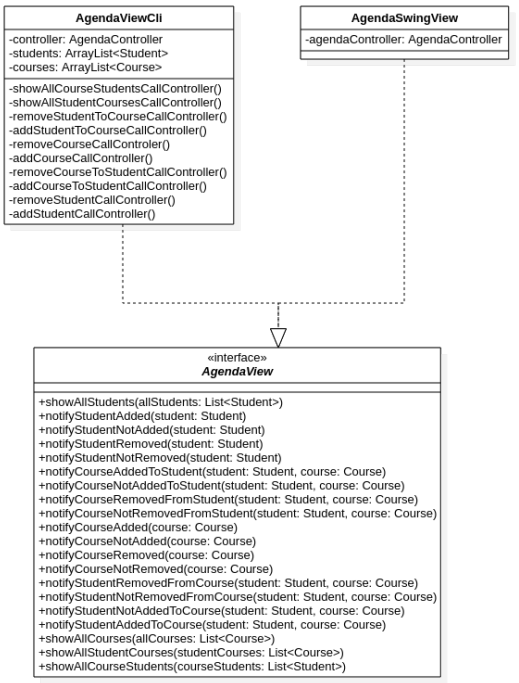
The user can select a course profile by using its ID and see all the students with which it is associated.

Remove a Course Student

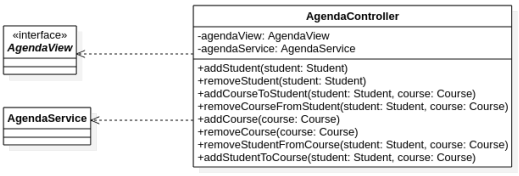
The user can select a course and one of his student by using the ID for both and remove that student from the list of associated course of the course.

To better understand the architecture used, following is presented the UML draw during the project analysis phase. In this model can be distinguish five main layers:

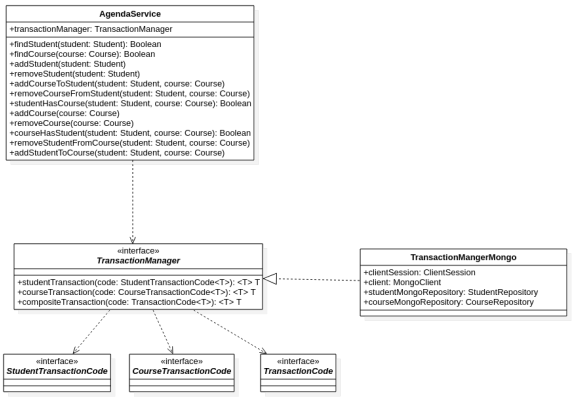
- *View*: this is the layer used to present the data. The way those data are presented, by using a CLI rather then a GUI is up to the user. The application offer the opportunity to choose between both of them hence the presence of an interface implemented by the two different classes



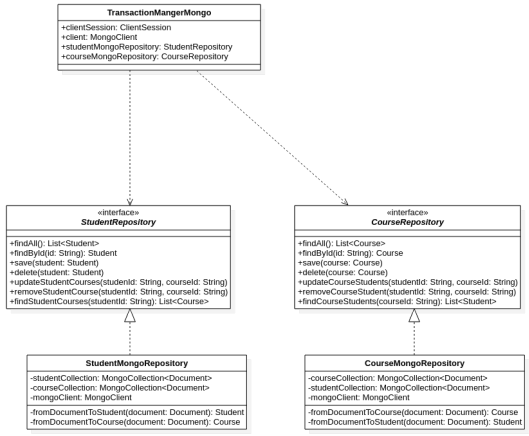
- *Controller*: the user does not interact directly with the View, instead every request is handled by this layer. On its behalf the Controller use the Service to interact with the Database and View to show the user the results obtained.



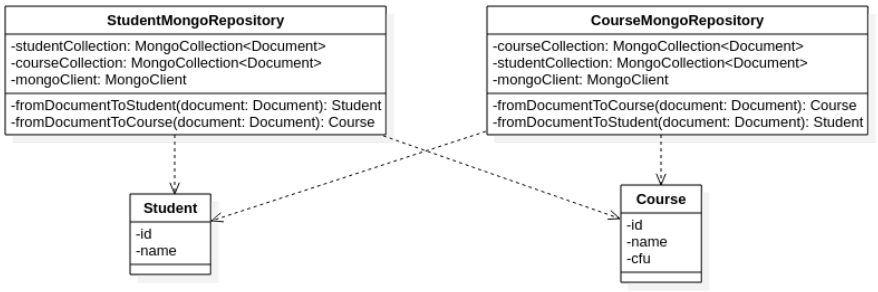
- *Service*: this layer contains the logic of the application. It interface with the Repository classes to obtain data and in particular it uses the Transaction Manager to execute every operation within a transaction.



- *Repository*: this layer is the interface used to talk with the Database both for writing and reading. We defined an interface which is implemented by the actual class, one for each entity (Student and Course), to interact with the correct collection.



- *Model*: this layer contains the domain classes. Those classes describe the entities used within the application. The entities are:
  - *Student*: this class represents the student, identified by an unique ID and by the name
  - *Course*: this class represents the course, identified by an unique ID, the name and the CFU



5 Testing and Developing

Unit 188 IT 79 E2E 41

Following the TDD process we also used the so called *Test Pyramid* hence dividing the test in three categories. The frameworks used for testing were *JUnit*, used for unit, integration and end-to-end tests, *AssertJ* used for assertions.

The total amount of tests for our project is 308 and they are divided as such:

- 188 *Unit Tests*
- 79 *Integration Tests*
- 41 *End-2-End Tests*

In the following sections we will see the detail of the three phases of the pyramid.

5.1 Unit Testing

In the first phase the single classes are tested in isolation i.e. independently to the other classes they interact with. Those components are simulated using a mock version obtained thanks to the *Mockito* library.

At this phase the main goal was to cover as much as possible all the cases for the method under testing. The tests wrote in this phase are strictly linked with the code coverage, measured using both *JaCoCo* (local) and *SonarCloud* (remote).

Some classes are not covered by tests at all. This is because these classes have no logic (most of them are generated automatically using Eclipse).

A special mention is necessary for the test involving the GUI and the CLI.

- Because the GUI was developed using *Swing* we used the corresponding library *AssertJSwing* for the tests. This library use an object of class `FrameFixture` to simulate the user interaction with the GUI.

- As for the CLI, on the other hand, no third-party libraries were used. Therefore, again, AssertJ were was for testing it. More interestingly, we used a single thread approach simulating the controller methods using Mockito. The CLI class takes the I/O streams as input in the constructor, this way during the tests is possible to use an object of class `ByteArrayStream` to simulate the user input and an object of class `PrintStream` to check the output.

5.2 Integration Testing

During this second phase, the goal is to test the correct behavior of the components even when several of them are used together and with other third-party components (e.g. a database).

At this level only the so called positive cases are tested. Moreover a "real" database is used, dockerizing a Mongo container.

We first tested the interaction between repository and db, then the interaction between the transaction manager and the underlying components and finally the interaction between the controller and the view with the corresponding underlying layers.

5.3 E2E Testing

This last phase is used to test the whole application. Because the application has two different UIs, we wrote two different test classes.

GUI Test

In the setup phase, we insert in the db two different students and two different courses, pairing each course to a student. After that, we run the application using the AssertJSwing API, also used for assertions.

CLI Test

Again, the db is filled with two students and two courses in the setup phase. After that we started the docker container using the `Runtime` class API while the application is launched using the `ProcessBuilder` class API, also used to retrieve the necessary I/O streams in order to simulate user interaction. Interestingly enough is to note that we used the `.jar` file to run the application during these tests because there are no AssertJ-like libraries to run CLI tests.

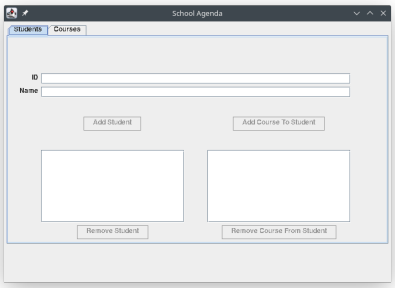


Figure 1: Screenshot of the implemented GUI

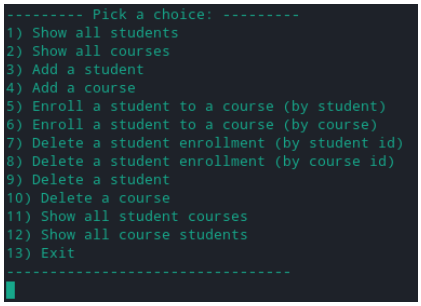


Figure 2: Screenshot of the implemented CLI

6 MongoDB Advanced Usage

For this project we used some peculiarities of MongoDB. In particular we decided to encapsulate every operation on the db within a transaction because of situations that require atomicity of reads and writes to multiple documents or in a single or multiple collections. To use the transactions is necessary to configure the db as *replica set* in order to support the usage of sessions allowing the execution of multi-collection transactions. To avoid manually configuration of the mongo instance to act as such we used an alternative mongo docker image, called *krnbr/mongo*, instead of the normal one.

The usage of the transaction in the code can be spotted because every mongo API used to communicate with the db takes as additional parameter an object of class `Session`, created from a `MongoClient` object.

A practical example to better understand the use of transactions in our project is the following:

- Student A is removed from the db
- Consequently, student A needs to be removed from every course that was enrolled in

If the database is abruptly stopped halfway through this operations, it can revert itself back to a consistent state prior to the failed transaction.

In our code the transaction is executed by the Repository class correspondent to the Mongo collection to modify. The repository, in turn, is executed by a lambda function within the real implementation of the `TransactionManager`, i.e. the `TransactionManagerMongo` class. This way, we can pass to the repository the client session needed to actually execute the transaction.

7 Continuous Integration

We used GitHub as VCS, and we used Gitkraken as a Git GUI client to cooperate and manage the work between the developers involved in the project. The workflow used during the project development was, for every single developer, the following:

1. Write a new feature
2. Create a new branch
3. Commit and push the changes to the new branch
4. Check if the current build is successful
5. If the build is successful, open a Pull Request asking another developer to review your changes
6. The review checks all the changes in the project and approves them one by one
7. The Pull Request is eventually merged into the `master` and the current branch is deleted

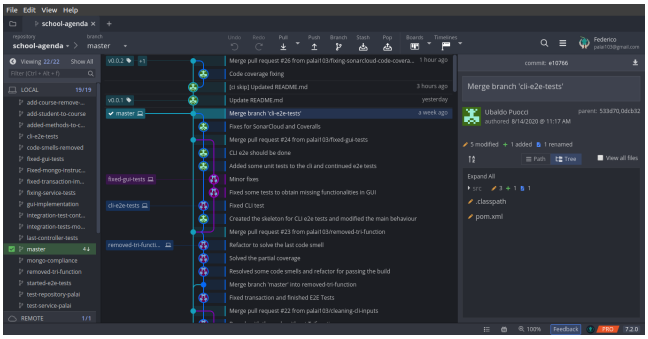


Figure 3: A screenshot showing the GitKraken GUI

After every push committed from Gitkraken the CI process was triggered. In particular, we used Travis as a CI server, SonarCloud and Coveralls to check additional info about the code (the former mainly for code smells and duplication in the code, the latter for code coverage).

All these services are directly triggered from Travis CI, in particular a maven command is defined in the `.travis.yml` file in the root of the project indicating that additional phases must be run.

In general, this approach has been used for the whole project, with some exceptions: classes that had no logic or that were impossible (for their nature) to test were excluded from the code coverage or code duplication (or both) calculation from SonarCloud and Coveralls.

8 References

GitHub Repository: https://github.com/palai103/school-agenda