Computer Engineering Master's Degree

Advanced Techniques and Tools for Software Development

Library management application

Student:
Mihail Teodor Gurzu
ID:
7060245

Professor:
Prof. Lorenzo Bettini

March, 2023

Contents

1 Introduction

In this report will be described the project developed for the final exam for the *Advanced Techniques and Tools for Software Development* course at the *University of Florence*. The project consists in a very basic management application for a library, of *books* and *users*, where the following operations can be performed:

- add users and books.
- remove users and books.
- a registered user can borrow a registered book if the book is available.
- when a book gets borrowed becomes unavailable until gets returned by the user.
- a user can return a borrowed book.
- when a book is returned it becomes available to be borrowed by any registered user.
- a user cannot be deleted if still has borrowed books to return.
- a book cannot be deleted if it is currently borrowed by a user.

The focus of the project is the correct usage of advanced techniques for build automation and continuous integration with a main focus on Test-Driven Development. In fact, all the main classes containing logic have been developed with a Test-Driven Development approach by first writing the suitable failing **Unit Test** with **JUnit 4** and then the logic of the class to make the test succeed. There have been tested also the integration between different classes by writing **Integration Tests** to verify the correct behavior of the interaction of different implemented classes or external libraries, and finally a few **End-To-End** tests check the correct functionality of the final application from a user perspective utilizing the **Graphical User Interface** developed with the **Swing** library in a TDD fashion as well.

During the development there have been utilized important tools like **Maven** for *build automation* and *dependency management*, **git** as a *version control system*, **github** as a *repository* and **github actions** as a *continuous integration* platform. Other advanced tools such as **Mutation Testing**, **Code Coverage** and **SonarQube** have been utilized to seek for code bugs and eventually to help their correct removal.

The application has been developed using **Java 8**, configuring the CI server to build the application on **JAVA 8 & 11**, performing code coverage and mutation testing on the Java 11 version.

There are two final versions of the application, due to the databases used. In fact, there has been developed a version using **MongoDB**, a NO-SQL database and **MySql**, an SQL database. To perform IT tests, we used **Testcontainers** to set up the required database in a **Docker container**.

The overall development has followed the practices described in detail in [1]

2 Implementation procedure

The application architecture is a variant of the **Model-View-Controller** called **Model-View-Presenter**, depicted in Figure 1.

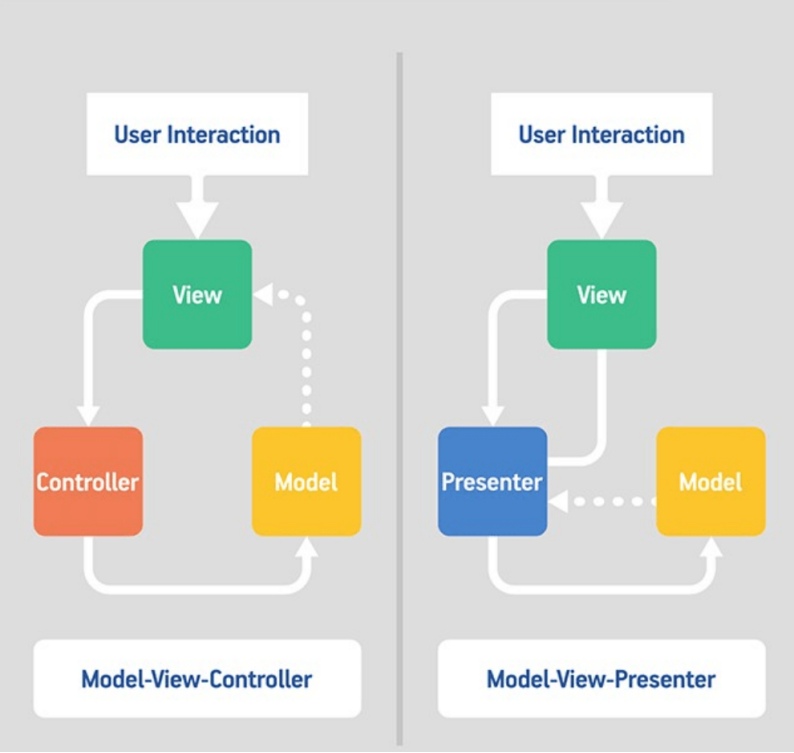

Figure 1: Model View Controller vs Model View Presenter architecture

2.1 Model

The model of the application is composed by two entities, *User* and *Book*. The two entities have basic attributes:

- **User**:
  - `int id`: stores the id of the user.
  - `String name`: stores the name of the user.
  - `Set<Book> rentedBooks`: stores the books rented by the user at that moment.
- **Book**:
  - `int id`: stores the id of the book.
  - `String title`: title of the book.
  - `String author`: author of the book.
  - `boolean available`: indicates the availability of the book (if the book is borrowed by a user it is not available). Defaults to true.

– `int userID`: indicates the id of the user which has currently borrowed the book. Defaults to -1.

The one-to-many relationship between User and Books has been modelled differently in the two databases used, but more on this in the relative sections.

2.2 Controller implementation

For the TDD implementation of the *Controller* class we have written a total of **25 UT** using **AssertJ**, a library which permits to write very readable tests, an important caracteristic for Unit Testing.

The implementation of this class has implied the definition of four *interfaces*:

- `TransactionManager`: All the operations that the controller requests to the repositories will be implemented with **transactions** in such a way, if a problem arises during a certain operation the state of the database is not changed at all, thus we ensure *atomicity*. The `TransactionManager` interface abstracts the implementation on a specific database and uses a functional interface, `TransactionCode` which represents a lambda that takes both a `UserRepository` and `BookRepository` and returns the result of the query which represents. The `TransactionManager` class will catch the eventual exceptions thrown by the repository implementation operations, will rollback the transaction and will throw the custom exception `Repository Exception` which will be finally cautgh by the controller.

- `UserRepository` and `BookRepository`: list the classic **CRUD** operations which will be performed on the particular database. The repository methods throw a custom exception `RepositoryException` in case something goes wrong which will be cautgh by the `TransactionManager` and rethrown to the controller which will send an appropriate message to the view in order to inform the user.

- `LibraryView`: declares all methods with which the controller communicates with the user interface, by sending information retrieved from the repositories and as a result of the user requests.

During the implementation of the *Controller* class, in the UT we have used Mocks, a mechanism provided by the **Mockito** framework which allows us to instantiate fake dependencies and to stub their methods occurrently, thus for the UT, the `LibraryView`, `UserRepository`, `BookRepository` and `TransactionManager` dependencies got mocked and the `TransactionManager.doInTransaction()` gets stubbed by using a Mockito answer, since we need to execute code based on the runtime value of the argument passed to the stubbed method.

Mockito also allows us to *spy* on a certain instance and we used this functionality in the Controller UT to verify for example in the `testBorrowBookWhenBookIsAvailable()` test that when a book is borrowed, its setter methods for the `available` and `userID` methods get effectively called and with the correct parameters.

Mockito allows us to verify as well the order of executions of certain operations and interactions with `Mockito.inOrder()` which takes as parameters mocked or spied instances.

2.3 Repositories Implementation

There have been implemented two versions for each `UserRepository` and `BookRepository`, one with *MongoDB* and one with *MySQL* databases. In order to be able to follow a TDD approach in the impolementation of these classes, we have exploited the mechanism offered by the **Testcontainers** library, which supports JUnit tests and provide lightweight, throwaway instances of common databases, run in **Docker containers**. This approach allows us to write tests interacting with a real version of the database, unlike other options such as utilizing *in memory databases* which usually are a simplified version of the original database.

For the TDD of these classes we wrote:

- **10** tests for `UserRepositoryMongo`.
- **7** tests for `BookRepositoryMongo`.
- **15** tests for `UserRepositoryMySql`.
- **12** tests for `BookRepositoryMySql`.

The *Docker Images* with which Testcontainers has been configured for these tests are:

- *mongo:6.0.3* for MongoDB.
- *mysql:8.0.32* for MySql.

Testcontainers makes sure that these containers are up and running before executing any test and at the end of the tests execution it will stop and remove the container independently of the outcome (this way we avoid all the manual configuration needed to assure that when tests are being executed the docker container is ready).

The mentioned tests which are a total of **44** tests, even if they are effectively **Integration tests** will be considered **Unit tests** since these tests have been written during the TDD implementation of the repositories and remembering that the interaction with the real database is needed to avoid the use of an in-memory database.

Before each test, we configure the database to a default situation, that being an empty database for the *MongoDB* and a configuration with an existing default user with `id = -1` for the `BookRepositoryMySql`, since the Book table has a foreign key to the User table. The MongoDb version doesn't use relations and the one to many relation is mapped as an *embedded list of book documents*. The *MySql* docker container is initialized using an `.sql` file which defines the tables and eventually an initial configuration of the database, a database containing only the *default-user* with `id = -1` in our case.

The Testcontainers library has been used as well for testing the implementations of the `TransactionManagerMongo` and `TransactionManagerMysql` classes which have been implemented following the respective API's. For this purpose there have been written:

- **2** tests for `TransactionManagerMongo`.
- **4** tests for `TransactionManagerMySql`.

2.4 LibraryViewSwing implementation

The view implementation has been carried out using the **Swing** library and the **Window Builder** tool available for Eclipse which makes possible to build the interface by positioning the desired 'blocks' and then generating automatically the corresponding code. The generated code will then be customized by adding events to buttons and list selections which on certain conditions will call the appropriate controller method.

As for the *controller* and the *repositories* we followed a TDD approach for the implementation of the view as well, by first writing the tests for the presence and default configuration of *buttons*, *labels* and *lists* using **AssertJ Swing**. After that we proceeded by implementing several *key listeners*, *key enablers* and *list selection listeners* which enable certain buttons, populate lists and call controller methods. Here are the main features of the GUI:

- the *Add User* and *Add Book* buttons are enabled only if all required respective fields are filled accordingly.
- the id field of both User and Book accept only integer values. If the entered values do not respect this bond, when clicking on the respective button for insertion, there will be displayed an error.
- the *Delete Selected* and *Delete* buttons used for User and Book deletion respectively are enabled only if a user or a book from the according list is selcted.
- the *Borrow* button is enabled only if a User and a Book are selected.
- the borrowed books list is populated only when a user is selected.
- the *Return* button is enabled only if a book is selected from the borrowed books list and a user is selected (the borrowed books list is populated only when a user is selcted).

In total there have been written **40 UT** and the features listed above are being tested using the **GuiActionRunner.execute** utility method provided by AssertJ Swing which executes a lambda in the **Event Dispatch Thread (EDT)**, where AssertJ Swing checks the execution of operations on Swing components. For the UT of the view which have to implement and verify the interactions with the *LibraryController*, the controller is mocked.

Until now we described the TDD implementation of the *Controller*, *Repositories* and *View* by writing a total of **115 UT**.

2.5 Integration Tests

After implementing the single components in isolation, is time to test their interaction by writing some *Integration Tests*. It has been tested the integration between the following components:

- **Controller - Repository**: With these IT there has been tested the integration between the Controller and the two versions of Repositories, by using again Testcontainers and by mocking the view.
  - `LibraryControllerMongoIT`: **8 IT**.
  - `LibraryControllerMySqlIT`: **8 IT**.
- **Controller - View**: With these IT there has been tested the integration between the Controller and the Swing view. The user and book repositories have been mocked as well as the transaction manager.
  - `LibraryViewSwingIT`: **13 IT**.
- **MVC IT**: In these tests has been tested the integration of all components manually wired together. Testcontainers is used for the databases.
  - `LibraryViewSwingMongoIT`: **6 IT**.
  - `LibraryViewSwingMySqlIT`: **6 IT**.

Far this project there have been written a total of **41 IT**.

2.6 End To End tests

In e2e tests we run verifications directly on the whole running application by interacting directly and only with the user interface of the application. To run the application for testing we launch it with AssertJ Swing which supports launching an application from its `main()` method. To be able to handle command-line arguments we use **Picocli**, a Java command line parser, which allows us to specify the command line arguments by defining fields in a Java class with its annotations which will be initialized with the values passed from the command line arguments (for more details regarding the command line arguments refer to the Appendix A).

For the e2e tests we didn't use testcontainers, but configured Maven to start the needed containers. For the *Mongo* version of the application, the image used is *candis/mongo-replica-set:0.0.2* which is already configured as a replica set, while for the *MySql* version we used *mysql/mysql-server:8.0.32* image.

There have been written the following E2E tests:

- `LibrarySwingMongoAppE2E`: **7 E2E**.
- `LibrarySwingMySqlAppE2E`: **7 E2E**.

The test fixture this time is represented by a database already populated with 2 users and 2 books, where users have no borrowed books and all books are available. For the *MySql* version the User table contains actually 3 users due to the presence of the *default-user* with `id = -1`.

Summarizing there have been written a total of **170** tests:

- **115** Unit Tests;
- **41** Integration Tests;
- **14** End To End Tests;

subdivision which is in accordance with the testing pyramid in Figure 2.

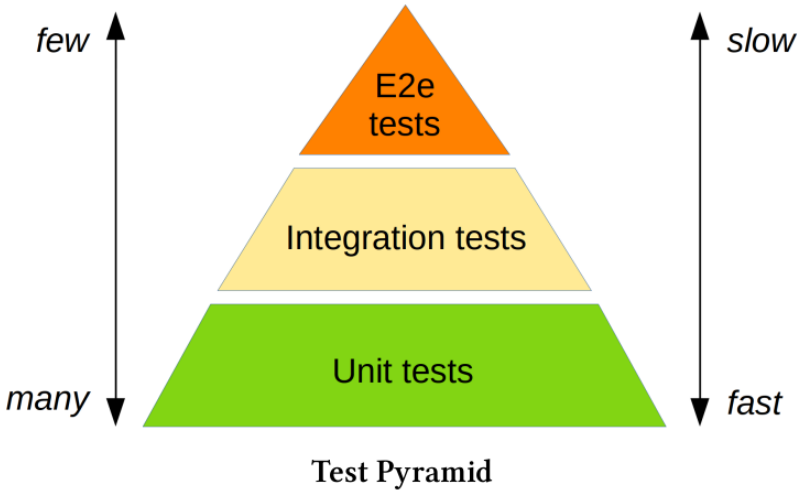

Figure 2: Test pyramid

3 Development

3.1 Maven

Another fundamental aspect on which we focused during the development of the application was the completely automation of the build process, thus we carried out **Build Automation** with **Maven**, which aims at simplifying the build process and dependency management. A Maven project must be described and configured in a single file `pom.xml` which is the XML representation of the **Project Object Model (POM)**. Let's take a closer look at the most important parts of the pom of our project.

Dependencies

```
<dependencyManagement>
        <dependencies>
                <dependency>
                        <groupId>org.apache.logging.log4j</groupId>
                        <artifactId>log4j-bom</artifactId>
                        <version>${log4j.version}</version>
                        <scope>import</scope>
                        <type>pom</type>
                </dependency>
        </dependencies>
</dependencyManagement>

<dependencies>
        <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>${junit.version}</version>
                <scope>test</scope>
        </dependency>
        <dependency>
                <groupId>org.assertj</groupId>
                <artifactId>assertj-swing-junit</artifactId>
                <version>${assertj.swing.version}</version>
                <scope>test</scope>
        </dependency>
        <dependency>
                <groupId>org.mockito</groupId>
                <artifactId>mockito-core</artifactId>
                <version>${mockito.version}</version>
                <scope>test</scope>
        </dependency>
        <dependency>
```

```
        <groupId>org.mongodb</groupId>
        <artifactId>mongo-java-driver</artifactId>
        <version>${mongodb.version}</version>
</dependency>
<dependency>
        <!-- required to see Mongo Java Driver logs -->
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>${logback.version}</version>
</dependency>
<dependency>
        <groupId>org.testcontainers</groupId>
        <artifactId>testcontainers</artifactId>
        <version>${testcontainers.version}</version>
        <scope>test</scope>
</dependency>
<dependency>
        <groupId>org.testcontainers</groupId>
        <artifactId>mongodb</artifactId>
        <version>${testcontainers.version}</version>
        <scope>test</scope>
</dependency>
<dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <version>${mysql.connector.version}</version>
</dependency>
<dependency>
        <groupId>org.testcontainers</groupId>
        <artifactId>mysql</artifactId>
        <version>${testcontainers.version}</version>
        <scope>test</scope>
</dependency>
<dependency>
        <!-- version taken from the imported BOM -->
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
</dependency>
<dependency>
        <!-- version taken from the imported BOM -->
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
</dependency>
<dependency>
        <groupId>info.picocli</groupId>
        <artifactId>picocli</artifactId>
```

```
            <version>${picocli.version}</version>
        </dependency>
</dependencies>
```

- **JUnit** is the most popular testing framework for Java developers. We used in this project Junit4 with version *4.13.2* specifying its scope to tests only.
- **assertj-swing-junit** is for **AssertJ Swing** which also includes **assertj-core** dependency. Version used is *3.17.1*.
- **mockito-core** is one of the most used frameworks for mocking, easy to use and allows to write very readable tests. Version used is *4.11.0*.
- **mongo-java-driver** dependency needed to interact with MongoDB from Java. Version used is *3.12.11*.
- **logback-classic** dependncy required to see Mongo Java Driver logs. Version used is *1.3.5*.
- **testcontainers** is the dependency that allows us to start throwaway instances of Docker containers directly from JUnit tests. Version used is *1.17.6*.
- **org.testcontainers.mongodb** is the dependency for the specific mongodb container for testcontainers. Version used is the one of testcontainers's.
- **mysql-connector-j** is a JDBC Type 4 driver, which means that it is pure Java implementation of the MySQL protocol and does not rely on the MySQL client libraries. Version used is *8.0.32*.
- **org.testcontainers.mysql** is the dependency for the specific mysql container for testcontainers. Version used is the one of testcontainer's.
- **log4j-api** and **log4j-core** are the dependencies for one of the most used Java libraries for logging events. These two dependencies are part of a multi-module library which provides a **BOM** artifact from which we can gather the version of these two dependendies. Version used is *2.19.0*.
- **picocli** is the dependency for the Java command line parser we used for our applications. Version used is *4.7.1*.

Plugin Management

The `<pluginManagement>` section is used for locking the version of the plugins and to avoid to use the default ones which depend on the Maven installation. Also we can specify a generic configuration of our plugins.

```
<build>
   <pluginManagement>
      <plugins>
```

```
        <plugin>
           <artifactId>maven-surefire-report-plugin</artifactId>
           <version>2.22.2</version>
        </plugin>
        <plugin>
           <artifactId>maven-site-plugin</artifactId>
           <version>3.9.1</version>
        </plugin>
       ...
```

These two plugins are part of the Maven installation and they generate a report on the tests executed during the build and generate also a `HTML` page displaying the reports.

```
<pluginManagement>
...
<plugin>
   <groupId>org.jacoco</groupId>
   <artifactId>jacoco-maven-plugin</artifactId>
   <version>0.8.8</version>
   <configuration>
      <excludes>
         <exclude>**/model/*.*</exclude>
         <exclude>**/library/view/swing/*.*</exclude>
         <exclude>**/library/app/swing/mongo/*.*</exclude>
         <exclude>**/library/app/swing/mysql/*.*</exclude>
      </excludes>
   </configuration>
   <executions>
      <execution>
         <goals>
            <!-- binds by default to the phase "initialize" -->
            <goal>prepare-agent</goal>
            <!-- binds by default to the phase "verify" -->
            <goal>report</goal>
         </goals>
      </execution>
   </executions>
</plugin>
```

This plugin provides the JaCoCo runtime agent to our tests and allows basic reports creation. This plugin is used to keep track of **code coverage** and generate code coverage reports.

The goal `prepare-agent` prepares a Maven property pointing to the JaCoCo Java agent which will be used during the tests for collecting data on code coverage

thus it is bounded to the phase `initialize` by default. The goal `report` takes the generated coverage files and generates a website with code coverage report and is bounded by default to phase `verify`, which comes after the tests have been run.

We excluded some classes from code coverage analysis, such as the model classes, the Swing view class generated by Window Builder and the two app classes.

```
<plugin>
   <groupId>org.eluder.coveralls</groupId>
   <artifactId>coveralls-maven-plugin</artifactId>
   <version>4.3.0</version>
   <dependencies>
      <!-- this is required when using JDK 9 or higher
      since javax.xml.bind has been removed from the JDK -->
      <dependency>
         <groupId>javax.xml.bind</groupId>
         <artifactId>jaxb-api</artifactId>
         <version>2.3.1</version>
      </dependency>
   </dependencies>
</plugin>
```

The `coveralls-maven-plugin` relies on the `XML` report generated by JaCoCo and the goal `report` goal bounded to phase `verify` sends to the Coveralls Web page the coverage reports. For recent versions of Java it needs the `jsxb-api` dependency.

```
<plugin>
   <groupId>org.pitest</groupId>
   <artifactId>pitest-maven</artifactId>
   <version>1.10.3</version>
   <executions>
      <execution>
         <phase>verify</phase>
         <goals>
            <goal>mutationCoverage</goal>
         </goals>
      </execution>
   </executions>
   <configuration>
      <excludedClasses>
         <param>com.gurzumihail.library.model.*</param>
         ↪  <param>com.gurzumihail.library.transaction_manager.mongo.*</param>
         ↪  <param>com.gurzumihail.library.transaction_manager.mysql.*</param>
         <param>com.gurzumihail.library.view.swing.*</param>
```

```
            <param>com.gurzumihail.library.app.swing.mongo.*</param>
            <param>com.gurzumihail.library.app.swing.mysql.*</param>
        </excludedClasses>
        <excludedTestClasses>

            ↪  <param>com.gurzumihail.library.transaction_manager.mongo.*</param>

            ↪  <param>com.gurzumihail.library.transaction_manager.mysql.*</param>
            <param>com.gurzumihail.library.view.swing.*</param>
            <param>com.gurzumihail.library.app.swing.mongo.*</param>
            <param>com.gurzumihail.library.app.swing.mysql.*</param>
        </excludedTestClasses>
        <mutators>
            <mutator>STRONGER</mutator>
        </mutators>
        <mutationThreshold>100</mutationThreshold>
    </configuration>
</plugin>
```

PIT's goal `mutation-coverage` is bounded to phase `verify` and we have excluded few classes and test classes from mutation testing since it is worth mutation-testing only classes with important logic. The mutators are set to `STRONGER` and the threshold to `100` thus no survived mutants are permitted, otherwise the build would fail.

```
<plugin>
    <groupId>org.sonarsource.scanner.maven</groupId>
    <artifactId>sonar-maven-plugin</artifactId>
    <version>3.9.1.2184</version>
</plugin>
```

Is the default scanner for Maven projects and we lock the version in the plugin manager.

We have seen several plugins configured in the `<pluginManagement>` section. There have been defined a few Maven profiles in order to be able to activate some of these plugins such as *JaCoCo*, *PIT*, *Coveralls* only when required since some of them can be quite expensive in terms of computation time. Let's take a look at some of these profiles which are defined in the `<profile>` section:

```
<profiles>
    <profile>
    ...
    </profile>
    <profile>
    ...
    </profile>
</profiles>
```

```
<profile>
   <id>jacoco</id>
   <build>
      <plugins>
         <plugin>
            <!-- configured in pluginManagement -->
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
         </plugin>
      </plugins>
   </build>
</profile>
```

This profile activates the *JaCoCo* plugin configured in pluginManagement to operate code coverage.

```
<profile>
   <id>coveralls</id>
   <build>
      <plugins>
         <plugin>
            <!-- configured in pluginManagement -->
            <!-- JaCoCo report is required by coveralls-maven-plugin
            ↪  -->
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
         </plugin>
         <plugin>
            <!-- configured in pluginManagement -->
            <groupId>org.eluder.coveralls</groupId>
            <artifactId>coveralls-maven-plugin</artifactId>
            <executions>
               <execution>
                  <phase>verify</phase>
```

```
                <goals>
                    <goal>report</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
  </plugins>
 </build>
</profile>
```

This profile sends code coverage information to Coveralls. Since it needs the coverage reports produced by jacoco, in order to avoid a build failure due to not enabling the jacoco profile, we enable it directly from this profile. We bound the *report* goal of coveralls to the *verify* phase since it is not bounded by default to any Maven phase.

```
<profile>
  <id>pit</id>
  <build>
    <plugins>
      <plugin>
        <!-- configured in pluginManagement-->
        <groupId>org.pitest</groupId>
        <artifactId>pitest-maven</artifactId>
      </plugin>
    </plugins>
  </build>
</profile>
```

This profile activates the PIT mutation testing plugin already configured in the pluginManagement section.

More on the usage of these profiles in the next section.

The last profile is added in order to pass the argument for opening the unnamed modules for jdk versions higher that or equal to 9, otherwise we would get an annoying warning which however doesn't harm the overall build.

```
<profile>
  <id>jmv-9+</id>
  <activation>
    <jdk>[9,)</jdk>
  </activation>
  <properties>
    <argLine>--add-opens=java.base/java.util=ALL-UNNAMED</argLine>
```

```
   </properties>
</profile>
```

Let's now take a look at the configuration of other plugins in the `<plugins>` section:

```
<plugin>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>build-helper-maven-plugin</artifactId>
   <version>${codehaus.mojo.version}</version>
   <executions>
      <execution>
         <id>add-test-source</id>
         <phase>generate-test-sources</phase>
         <goals>
            <goal>add-test-source</goal>
         </goals>
         <configuration>
            <sources>
               <source>src/it/java</source>
               <source>src/e2e/java</source>
            </sources>
         </configuration>
      </execution>
      <execution>
         <id>add-IT-resource</id>
         <phase>generate-test-resources</phase>
         <goals>
            <goal>add-test-resource</goal>
         </goals>
         <configuration>
            <resources>
               <resource>
                  <directory>src/it/resources</directory>
               </resource>
            </resources>
         </configuration>
      </execution>
      <execution>
         <id>add-E2E-resource</id>
         <phase>generate-test-resources</phase>
         <goals>
            <goal>add-test-resource</goal>
         </goals>
         <configuration>
            <resources>
```

```
                <resource>
                   <directory>src/e2e/resources</directory>
                </resource>
             </resources>
          </configuration>
       </execution>
    </executions>
</plugin>
```

With this plugin we keep separate the UT, IT, and E2E tests by adding the respective *test source folders* for the last two and also we configure the *test resource folders* for the IT tests and E2E tests, needed for the `INIT.sql` file for Mysql.

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-failsafe-plugin</artifactId>
   <version>${failsafe.version}</version>
   <executions>
      <execution>
         <id>default-IT</id>
         <goals>
            <goal>integration-test</goal>
            <goal>verify</goal>
         </goals>
      </execution>
      <execution>
         <id>e2e-tests</id>
         <goals>
            <goal>integration-test</goal>
            <goal>verify</goal>
         </goals>
         <configuration>
            <excludes>
               <exclude>**/*IT.java</exclude>
            </excludes>
            <includes>
               <include>**/*E2E.java</include>
            </includes>
         </configuration>
      </execution>
   </executions>
</plugin>
```

The plugin for unit tests is *surefire* whose goal `test` is bound by default to the phase `test`, where only UT are executed. The `maven-failsafe-plugin` is

the plugin for IT but it is not enabled by default. Thus we configure it here by enabling it's goals `integration-test` and `verify` which are bounded to phases `integration-tests` and `verify` respectively. In order to execute the E2E tests as well, we configured the plugin to do so in another `<execution>`, by explicitly specifying to exclude the IT and to execute only the E2E tests once the IT have been executed in the default execution.

```
<plugin>
   <groupId>io.fabric8</groupId>
   <artifactId>docker-maven-plugin</artifactId>
   <version>${docker.maven.version}</version>
   <executions>
      <execution>
         <id>mongodb</id>
         <phase>pre-integration-test</phase>
         <goals>
            <goal>start</goal>
         </goals>
         <configuration>
            <images>
               <image>
                  <alias>mongodb</alias>
                  <name>${docker.image.mongodb}</name>
                  <run>
                     <ports>
                        <port>27017:27017</port>
                        <port>27018:27018</port>
                        <port>27019:27019</port>
                     </ports>
                     <wait>
                        <log>connecting to: mongodb</log>
                        <time>60000</time>
                     </wait>
                  </run>
               </image>
            </images>
         </configuration>
      </execution>
      <execution>
         <id>mysql</id>
         <phase>pre-integration-test</phase>
         <goals>
            <goal>start</goal>
         </goals>
         <configuration>
            <images>
```

```
                <image>
                    <alias>mysql</alias>
                    <name>${docker.image.mysql}</name>
                    <run>
                        <ports>
                            <port>3306:3306</port>
                        </ports>
                        <wait>
                            <tcp>
                                <ports>
                                    <port>3306</port>
                                </ports>
                            </tcp>
                            <time>60000</time>
                        </wait>
                        <env>
                            <MYSQL_DATABASE>library</MYSQL_DATABASE>
                            <MYSQL_USER>root</MYSQL_USER>
                            <MYSQL_ROOT_PASSWORD>password
                            </MYSQL_ROOT_PASSWORD>
                            <MYSQL_ROOT_HOST>%</MYSQL_ROOT_HOST>
                        </env>
                        <volumes>
                            <bind>
                                <volume>src/e2e/resources/database:
                                /docker-entrypoint-initdb.d</volume>
                            </bind>
                        </volumes>
                    </run>
                </image>
            </images>
        </configuration>
    </execution>
    <execution>
        <id>docker-stop</id>
        <phase>post-integration-test</phase>
        <goals>
            <goal>stop</goal>
        </goals>
    </execution>
  </executions>
</plugin>
```

The `docker-maven-plugin` is used for setting up the necessary containers with *mongodb* and *mysql* for the E2E tests end for being able to run the applications. We have configured in total three executions, one for starting the mongodb image,

one for the mysql image and the last one for stopping and removing the images.

The first two executions are bounded to the `pre-integrations-test` phase, while the last one to the `post-integration-phase`. For each image we can specify some further configurations like the ports exposed to the host or evironment variables for example for the mysql image. For the mysql image we specify also the location of the `INIT.sql` file used for database initial configuration.

For both containers there has been defined a wait strategy in order to assure that once the plugin has finished his goal's execution, the containers are effectively ready and operative. For *mongodb* we configured a wait based on the specified log and for *mysql* on the response on requested port. Both strategies have a timeout of 60s even if the *mongodb* container takes 30-45 seconds to be operative and *mysql* around 10s.

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-assembly-plugin</artifactId>
   <executions>
      <execution>
         <id>build-mongo-app</id>
         <phase>package</phase>
         <goals>
            <goal>single</goal>
         </goals>
         <configuration>
            <finalName>MONGO_app</finalName>
            <archive>
               <manifest>
                  <mainClass>com.gurzumihail.library.app.swing.mongo.
                  LibrarySwingMongoApp</mainClass>
               </manifest>
            </archive>
               <descriptors>
                  <descriptor>src/main/resources/MONGO.xml
                  </descriptor>
               </descriptors>
         </configuration>
      </execution>
      <execution>
         <id>build-mysql-app</id>
         <phase>package</phase>
         <goals>
            <goal>single</goal>
         </goals>
         <configuration>
            <finalName>MYSQL_app</finalName>
            <archive>
```

```
            <manifest>
               <mainClass>com.gurzumihail.library.app.swing.mysql.
               LibrarySwingMySqlApp</mainClass>
            </manifest>
         </archive>
         <descriptors>
            <descriptor>src/main/resources/MYSQL.xml
            </descriptor>
         </descriptors>
      </configuration>
   </execution>
</executions>
</plugin>
```

The `maven-assembly-plugin` is configured to build the two *FatJars*, one for each version of the application including only the necessary dependencies for each version. Its goal `single` is bounded to phase `package` in each of the two executions. Each execution then relies on the `xml` file indicated in the `<descriptor>` section which specifies essentially the dependencies to exclude from the jars in the `<dependencySet>` section:

```
<dependencySets>
   <dependencySet>
      <outputDirectory>/</outputDirectory>
      <useProjectArtifact>false</useProjectArtifact>
      <unpack>true</unpack>
      <scope>runtime</scope>
      <excludes>
         <exclude>org.mongodb:mongo-java-driver</exclude>
         <exclude>ch.qos.logback:logback-classic</exclude>
      </excludes>
   </dependencySet>
</dependencySets>
```

Above we can see a snippet of the configuration file for the *mysql* fatjar, where we exclude the dependencies needed by *mongodb*.

Sonar configuration

Sonar Clound needs some further configuration which we carried out in the `pom`'s `<properties>` section. First we have to explicitly exclude the classes from code coverage even if already excluded in the `jacoco` plugin configuration since SonarCloud does not consider them.

```
<sonar.coverage.exclusions>
   **/model/*,
   **/library/view/swing/*,
   **/library/app/swing/mongo/*,
   **/library/app/swing/mysql/*,
</sonar.coverage.exclusions>
```

We also disabled some sonar rules:

```
<sonar.issue.ignore.multicriteria>e11, e12, e13, e14,
↪  e15</sonar.issue.ignore.multicriteria>
<!-- Disable rule for "Tests should include assertions":
SonarQube is not aware of AssertJ Swing -->
<sonar.issue.ignore.multicriteria.e11.ruleKey>
   java:S2699
</sonar.issue.ignore.multicriteria.e11.ruleKey>
<sonar.issue.ignore.multicriteria.e11.resourceKey>
   **/LibraryViewSwingTest.java
</sonar.issue.ignore.multicriteria.e11.resourceKey>

<!-- Disable rule for "Rename this local variable to match the
↪  regular expression '^[a-z][a-zA-Z0-9]*$'."-->
<sonar.issue.ignore.multicriteria.e12.ruleKey>
   java:S117
</sonar.issue.ignore.multicriteria.e12.ruleKey>
<sonar.issue.ignore.multicriteria.e12.resourceKey>
   **/LibraryViewSwing.java
</sonar.issue.ignore.multicriteria.e12.resourceKey>

<!-- Disable rule for "Test classes should comply with a naming
↪  convention " -->
<sonar.issue.ignore.multicriteria.e13.ruleKey>
   java:S3577
</sonar.issue.ignore.multicriteria.e13.ruleKey>
<sonar.issue.ignore.multicriteria.e13.resourceKey>
   **/LibrarySwingMongoAppE2E.java
</sonar.issue.ignore.multicriteria.e13.resourceKey>

<sonar.issue.ignore.multicriteria.e14.ruleKey>
   java:S3577
</sonar.issue.ignore.multicriteria.e14.ruleKey>
<sonar.issue.ignore.multicriteria.e14.resourceKey>
   **/LibrarySwingMySqlAppE2E.java
</sonar.issue.ignore.multicriteria.e14.resourceKey>
```

```
<!-- Disable rule for "Reduce cognitive Complexity" -->
<sonar.issue.ignore.multicriteria.e15.ruleKey>
   java:S3776
</sonar.issue.ignore.multicriteria.e15.ruleKey>
<sonar.issue.ignore.multicriteria.e15.resourceKey>
   **/LibraryViewSwing.java
</sonar.issue.ignore.multicriteria.e15.resourceKey>
```

- `java:S2699 - Tests should include assertions`: We need to disable this rule since Sonar does not consider the assertions made with AssertJ Swing. We excluded this rule from the analysis of class `LibraryViewSwingTest.java`, the class with GUI UTs.
- `java:S117 - Rename this local variable to match the regular expression` $\hat{}[a-z][a-zA-Z0-9]*\$$: The Window Builder automatically names variables in a non complying manner with this rule thus we considered this rule as a false positive for the class `LibraryViewSwing.java`.
- `java:S3577 - Test classes should comply with a naming convention`: We used as a convention during the development and build automation with Maven to name the End-to-End test classes `**/*E2E.java` which is not complying with the `S3577` rule. We have thus excluded this rule from the analysis of classes `LibrarySwingMongoAppE2E.java` and `LibrarySwingMySqlAppE2E.java`.
- `java:S3776 - Reduce cognitive complexity`: In order to implement the desired functionality of our GUI, we had to insert a check of whether the inserted id's are integers for both users and books. By doing this we had to use `Integer.parseInt()` which throws a `NumberFormatException`. Every `if, else` in the `try-catch` block is then considered as a double negative point for the cognitive complexity. We think however that the code is still straightforward and choose to exclude this rule from the analysis of class `LibraryViewSwing.java`.

3.2 Continuous Integration with GitHub Actions

In order to configure and to instruct GitHubActions we have to create one or more configuration `YAML` files which define a specific *workflow*. For our project we created one workflow, which activates both on *push* and *pull-requests*. This workflow builds the application on a *Linux* environment running *ubuntu-latest*. Initially there were three workflows, the missing ones being the *windows* and *macos* workflows but since gitHub Actions windows environments don't support testcontainers the respective workflow has been canceled. The *macos* environments needed an installation of docker at each run and only this installation took more than 10 minutes, thus this workflow has been cancelled as well. This workflow runs the following command:

```
- name: Build with Maven
  run: >
    xvfb-run mvn verify ${{ matrix.additional-maven-args }}
```

where the additional maven arguments are specified only for the Java 11 build:

```
    strategy:
      matrix:
        # test against several Java versions:
        include:
          - java: 8
          - java: 11
            additional-maven-args: >
              -Pcoveralls,pit, sonar:sonar
              -Dsonar.organization=mihailteodor-github
              -Dsonar.host.url=https://sonarcloud.io
              -Dsonar.projectKey=MihailTeodor_attsw-final-project
              -DrepoToken=$COVERALLS_REPO_TOKEN
              -DpullRequest=${{ github.event.pull_request.number }}
```

For the Java 11 build we activate the `coveralls` profile which activates the `jacoco` plugin as well, `pit` and `sonar` for code coverage tracking and code quality control.

References

[1] Lorenzo Bettini. *Test-Driven Development, Build Automation, Continuous Integration*. 2021.

A How to use the code

From the `pom.xml`'s location directory, run the maven command:

```
mvn clean verify
```

which will build and run all tests.

There are two profiles that can be used for code coverage `jacoco` and mutiation testing `pit`. Run

```
mvn clean verify -Pjacoco,pit
```

to build the application and run all tests, with the addition of test coverage and mutation testing altogether.

In the target folder are generated two FatJARs:

- `MONGO_app-jar-with-dependencies.jar`
- `MYSQL_app-jar-with-dependencies.jar`

for the application using **MongoDB** and **MySql** respectively.
In order to run these jars:

- first run from the `pom.xml`'s directory the maven command

  ```
  mvn docker:start@mongodb
  ```

  or

  ```
  mvn docker:start@mysql
  ```

  which will start up the container for the **MongoDB** or **MySql** respectively.

- then, you can run the desired application, e.g.

  ```
  java -jar MYSQL_app-jar-with-dependencies.jar
  ```

  for the MySql version from the target directory.

- finally, use

  ```
  mvn docker:stop@mongodb
  ```

  or

  ```
  mvn docker:stop@mysql
  ```

  to stop and remove the desired container.

Each application can be used with an already existing database, by specifying the following arguments:

- **MONGO** database configuration
  - `--mongo-host-1` -> MongoDB host-1 address (default = localhost)
  - `--mongo-host-2` -> MongoDB host-2 address (default = localhost)
  - `--mongo-host-3` -> MongoDB host-3 address (default = localhost)
  - `--mongo-port-1` -> MongoDB host-1 port (default = 27017);
  - `--mongo-port-2` -> MongoDB host-2 port (default = 27018);
  - `--mongo-port-3` -> MongoDB host-3 port (default = 27019);
  - `--db-name` -> Database name (default = library);
  - `--db-user-collection` -> user collection name (default = user);
  - `--db-book-collection` -> book collection name (default = book);
- **MYSQL** database configuration
  - `--mysql-host` -> MySql host address (default = localhost);
  - `--mysql-port` -> MySql host port (default = 3306);
  - `--db-name` -> Database name (defaulkt = library);
  - `--db-user` -> username (default = root);
  - `--db-password` -> password (default = password);