"ALEXANDRU-IOAN CUZA" UNIVERSITY FROM IAȘI

FACULTY OF COMPUTER SCIENCE

Quality of Software Systems: Documentation

Onofrei Tudor-Cristian

Constantinescu George-Gabriel

Lupu Cezar

Filimon Daniel-Dumitru

Baciu Dragoș

Master's program: Software Engineering

Contents

1 Project Description **2**

1.1 Context 2

1.2 Implementation 3

1.2.1 Used Technologies 3

1.2.2 Development Phase 3

2 User Guide **6**

3 Unit Testing **8**

4 Assertions **20**

5 Contributions **25**

Chapter 1

Project Description

1.1 Context

The developed application is intended to facilitate the creation of the timetable for a certain faculty; the implementation has to offer a list of functionalities:

1. the management of the teachers
2. the management of the rooms
3. the management of the disciplines of study, organized by years
4. the management of student groups
5. the possibility of adding/removing a course/seminary/laboratory class

The application should notify the user about all the problems encountered in the timetable creation; the situations that may generate exceptions are:

- Classes may only be scheduled on weekdays (Monday-Friday), between hours 8-20.
- Course classes may only be scheduled in the course rooms (due to the large number of students), while laboratory classes may only be scheduled in the laboratory rooms. There is no restriction regarding seminary classes.
- Course classes are taught to whole study years, while seminary/laboratory classes are taught to groups.

The program must assist the user:

- by providing a graphical (not necessarily web-based) interface that eases the manipulation of the entities involved by notifying the user about any issues: broken restrictions; overlappings (e.g., two classes at the same time in the same room; two classes taught by the same teacher at the same time); etc.
- by generating the HTML files for publishing the timetable, in a manner that is similar to the current timetable of the faculty

1.2 Implementation

1.2.1 Used Technologies

- Java SE, version 1.8 for the main implementation
- Java Swing, for the user graphic interface creation
- PostgreSQL, for the database management system
- JUnit, for unit testing purposes and assertions
- Mockito, for unit testing mocking
- Jacoco, for code coverage metrics
- Apache Netbeans, as the main IDE
- pgAdmin4, as the administration and development platform for PostgreSQL

1.2.2 Development Phase

The first issue taken into consideration was the database structure creation, which can be observed in the Figure 1.1.

After the database structure was created, the graphical user interface was designed. At this step of the implementation, Java Swing was solely the GUI library used.

For the GUI, there were used combo box items in order to enforce the choices the user could make in relation with the database records; a more comprehensive guide and description of the GUI can be consulted in the *User Guide* section.

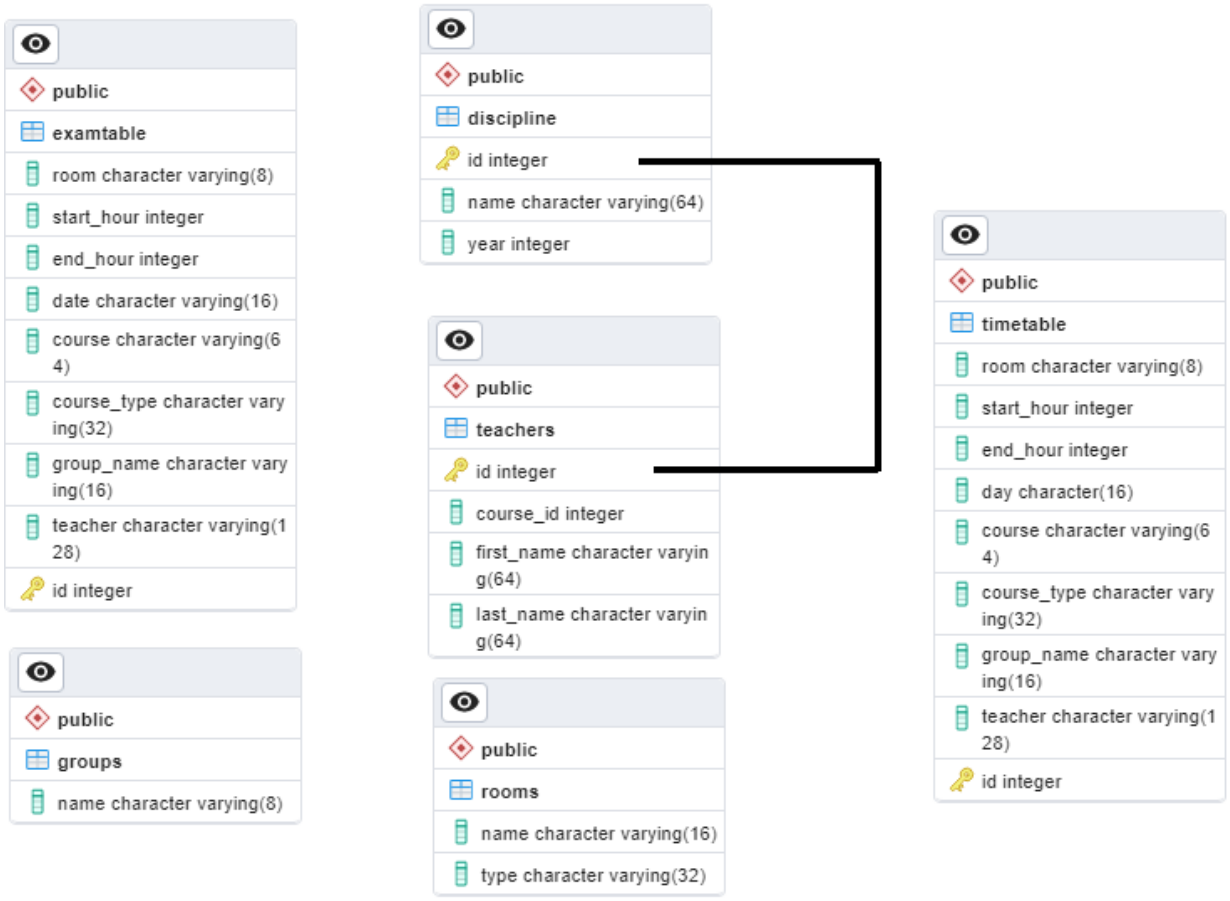

Figure 1.1: Database Structure

The database was populated using a configuration file, where the most important data for the timetable creation were stored.

The configuration file is defined under the resources folder of the project and it holds basic properties for the database setup. The properties defined represent the necessary connection parameters for the database connection, the parameters that are used to generate the basic entries in each table except the timetable, exam table and teachers and another final property that allows the initialization of the data inside the database to be toggled.

The most important properties are:

1. *setup.groups.years*: The number of years that attend the courses.
2. *setup.groups.count*: The number of groups in total for each year.
3. *setup.groups.halfs*: The number of half years for each year.
4. *setup.rooms*: The total number of rooms.

5. *setup.rooms.types*: A distribution of rooms that are distributes and sum exactly to 100. (eg: Seminary-40,Laboratory-40,Course-20)

6. *setup.teachers*: A number of teacher which is used to validate the number of total disciplines that are added.

7. *setup.discipline*: The disciplines and their respective year. (eg: PCD-4,CSS-4,Maths-1)

8. *setup.initiate*: Specifies if the application has to initialize the database with the data found in the configuration file.

The application uses a total of seven classes to provide the functionalities for adding new items to the timetable and exam table for groups, but also for the export of the results for each one of the existing group. The classes used are as following:

1. *ConfigReader*: A Singleton class that is used to read the properties from the configuration file.

2. *JDBCConnection*: A Singleton class which is used in order to maintain a single active connection to the database during a session; the interaction with the graphical widgets triggers certain events used in order to synchronize the values displayed in the graphical components

3. *RegistrationTimetable*: A class that represents an entry in the timetable/exam table.

4. *Timetable*: Main class of the application that runs the database setup and enables the graphical user interface.

5. *TiemtableUI*: The main user interface that provides functionalities for adding new timetable items and exam items that also validates the user input and provides feedback to the user.

6. *ExportTimetable*: A class used to export the timetable and exam table for each group. A HTML file is created for each group that has at least one entry for the timetable or exam table.

7. *DatabaseSetup*: A class used for importing the groups, disciplines, years of study from the *configuration file* - in case the *setup.initiate* parameter is set to *true*

Chapter 2

User Guide

Firstly, the graphical user interface was designed; the application is single-windowed, offering a simple and complete experience even to the most unexperienced user, as it can be seen in Figure 1.1.

The user interface is organised in 3 channels:

1. **Channel 1**: This is the channel that offers the user the chance of adding a record in the timetable; the user has to select the student group, choice which will trigger the event that updates the disciplines and the registrations in the associated timetables to the associated group.

   If the user selects a certain discipline, another event is created so that the teachers associated to that discipline are listed in the combo box menu.

   When the user selects the course type, the rooms list is updated properly, according to the discipline type; if the type is ”Course Exam” or ”Lab Exam”, the input for the ”start” and ”end” hours changes to an input field that should be a valid date.

   If the course type is different than exam, selecting the type will also filter the rooms accordingly.

   Fixing the start hour is preventing the selection of an end hour before the fixed start hour.

2. **Channel 2**: After the user has made the selection for a new timetable registration, the choice can be added in the timetable by pressing the ”Submit” button; submitting a ”course” discipline record will update the timetables of all the groups from the specific year.

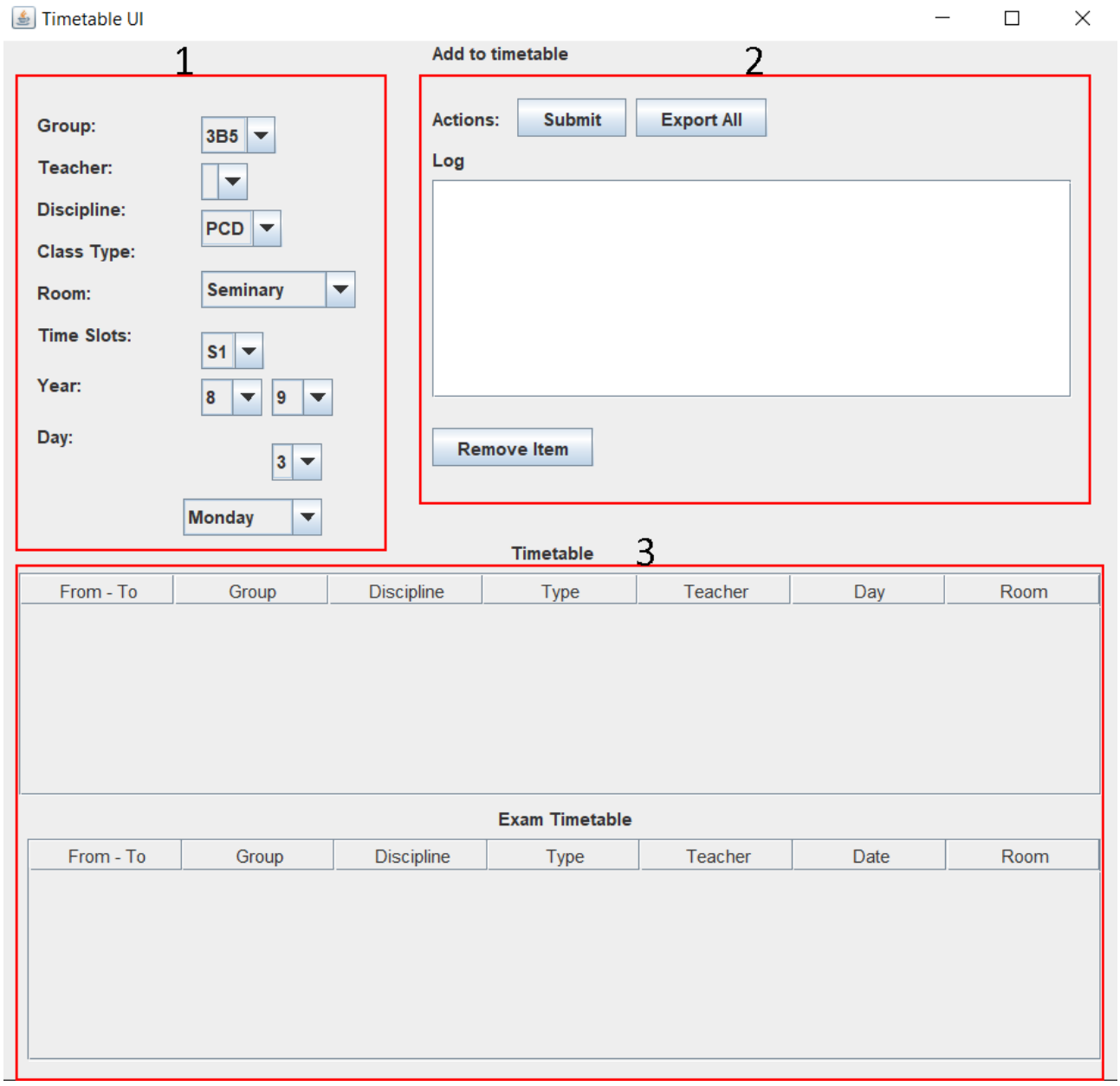

Figure 2.1: The graphical user interface

If any restrictions are broken, the user is notified in the ”Log” window about the issues.

If the user considers the timetable as being complete, it can be exported using the ”Export all” button, which will generate the timetables separately for every study group.

3. **Channel 3**: This section holds the generated timetables for a single group; if the user decides to remove a certain record from the timetable, by clicking that record and pressing the ”Remove Item” button from Channel 2 will delete the record from the database.

Chapter 3

Unit Testing

In order to check the errors within the implementation, there were written certain tests in order to check the correctness and the completeness of the implementation; every unit test checked a certain aspect:

- *ConfigReader*: Here, it was checked the singleton implementation of the ConfigReader. It was checked if two consecutive calls of *getInstance()* method return the same not-null instance.

```
@Test
public void testGetInstanceJDBCConnection() {
    ConfigReader myFirstInstance = ConfigReader.getInstance();
    ConfigReader mySecondInstance = ConfigReader.getInstance();
    assertNotNull(myFirstInstance);
    assertNotNull(mySecondInstance);
    assertSame(myFirstInstance, mySecondInstance );
}
```

- *DatabaseSetup*: It was checked that singleton implementation of the setup of the database is followed. Mocking techniques were used in order to simulate specific behaviour for a regular scenario. Also, it was tested if the database is set up or not, if the teachers are mismatched with the courses and if rooms are evenly distributed according to the fixed distribution.

```
@Mock
private ConfigReader configReader = mock(ConfigReader.class);
```

```
@Mock
private JDBCConnection jdbcConnection =
    mock(JDBCConnection.class);

@Mock
private Connection connection;

@Mock
private Statement statement;

@Mock
private PreparedStatement preparedStatement;
@Mock
private ResultSet resultSet;

private List<String> logMessages;

@BeforeEach
public void setUp() throws SQLException {
    MockitoAnnotations.openMocks(this);
    logMessages = new ArrayList<>();

    when(ConfigReader.getInstance())
        .thenReturn(configReader);
    when(JDBCConnection.getInstance())
        .thenReturn(jdbcConnection);
    when(jdbcConnection.getConnection())
        .thenReturn(connection);
    when(connection.createStatement())
        .thenReturn(statement);
    when(connection.prepareStatement(any()))
```

```
            .thenReturn(preparedStatement);
        when(statement.executeQuery(any())).thenReturn(resultSet);
    }
```

- *JDBCConnectionTest*: It was checked if using two different instances of JDBC connections are successful. Also it was checked if a current connection exists after using specific method from JDBC java class.

```
    @Test
    public void testGetInstanceJDBCConnection() throws SQLException {
        JDBCConnection myFirstInstance = JDBCConnection.getInstance();
        JDBCConnection mySecondInstance = JDBCConnection.getInstance();
        assertNotNull(myFirstInstance);
        assertNotNull(mySecondInstance );
        assertSame(myFirstInstance, mySecondInstance );
    }


    @Test
    public void testConnectionJDBCConnectionNotNull() throws
    ↪  SQLException {
        JDBCConnection myInstance = JDBCConnection.getInstance();
        assertNotNull(myInstance .getConnection());
    }
```

- *ExportTimetableTests*: The main purpose of this test class was to ensure that specific operations regarding timetable exporting are followed. It was tested that cleanup can be done in our database, if a timetable which was exported exists and can be opened, mocking up some specific operations regarding timetable changing (specific classes or exams) and checking that after the changes were done, the generated HTML file is according to the changes and also checking the integrity of our tables.

```
    @AfterEach
    public void cleanUp() throws SQLException {
        conn = JDBCConnection.getInstance().getConnection();
```

```
        Statement stmt = conn.createStatement();

        stmt.execute("DELETE FROM groups");
        stmt.execute("DELETE FROM timetable ");

        conn.close();
    }

    @Test
    public void testTableNoInformation() throws SQLException {
        ExportTimetable.createHTML();

        URL url = ClassLoader.getSystemResource("resources/1A1.html");
        assertFalse(url != null);
    }

    @Test
    public void testTableWithInformation() throws SQLException {
        conn = JDBCConnection.getInstance().getConnection();
        Statement stmt = conn.createStatement();

        stmt.execute("INSERT INTO groups (name) VALUES ('1A2')");
        stmt.execute("INSERT INTO timetable (start_hour, end_hour,
        ↪  group_name, course, course_type, teacher, day, room) "
                + "VALUES (8, 10, '1A2', 'Matematica', 'seminary',
                ↪  'Dragos', 'Monday', '101')");
        stmt.execute("INSERT INTO timetable (start_hour, end_hour,
        ↪  group_name, course, course_type, teacher, day, room) "
                + "VALUES (10, 12, '1A2', 'Css', 'seminary', 'Cezar',
                ↪  'Monday', '202')");

        ExportTimetable.createHTML();
```

```
String resourcePath = "src/main/resources/1A2.html";
try {
    File file = new File(resourcePath);
    try (Scanner scanner = new Scanner(file)) {
        Document doc = Jsoup.parse(scanner.nextLine());
        Elements timeTableRows = doc.select("tbody tr");
        ArrayList<String> contents1 = new ArrayList<>();
        ArrayList<String> contents2 = new ArrayList<>();
        int rowIndex = 1;

        assertTrue(timeTableRows.size() == 2);

        for (Element rowElement : timeTableRows) {
            Elements row = rowElement.select("td");

            for(Element cell : row) {
                String value = cell.text();

                if (rowIndex == 1) contents1.add(value);
                else contents2.add(value);
            }
            rowIndex = 2;
        }

        assertTrue(checkIntegrityRows(contents1, "8 - 10",
        ↪ "1A2", "Matematica", "seminary", "Dragos", "Monday",
        ↪ "101"));
        assertTrue(checkIntegrityRows(contents2, "10 - 12",
        ↪ "1A2", "Css", "seminary", "Cezar", "Monday",
        ↪ "202"));
    }
```

```
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

        stmt.execute("DELETE FROM groups WHERE name = '1A2'");
        stmt.execute("DELETE FROM timetable WHERE course =
        ↪  'Matematica'");
        stmt.execute("DELETE FROM timetable WHERE course = 'Css'");
        conn.close();
    }
```

- *RegistrationTimetableTests*: It was checked if a specific week day/course/type/group names/teacher/room set in the GUI is received correctly. There is also a check for the start and end hour of each event which includes seminary, course or exam. Besides, it is tested that changes made for a specific scenario is properly indexed in our database and retrieved correctly (with the interaction with the GUI).

```
    @Test
    public void testSetDay() {
        RegistrationTimetable rt = new RegistrationTimetable();
        MockitoAnnotations.openMocks(this); // Initialize the annotated
        ↪  mocks

        rt.setDay("Monday");
        assertTrue("Monday".equals(rt.getDay()));
    }

    @Test
    public void testSetCourse() {
        RegistrationTimetable rt = new RegistrationTimetable();
        MockitoAnnotations.openMocks(this); // Initialize the annotated
        ↪  mocks
```

```
        rt.setCourse("Matematica");
        assertTrue("Matematica".equals(rt.getCourse()));
    }

    @Test
    public void testSetCourseType() {
        RegistrationTimetable rt = new RegistrationTimetable();
        MockitoAnnotations.openMocks(this); // Initialize the annotated
        ↪ mocks

        rt.setCourseType("seminar");
        assertTrue("seminar".equals(rt.getCourseType()));
    }

    @Test
    public void testToString() {
        RegistrationTimetable rt = new RegistrationTimetable();
        MockitoAnnotations.openMocks(this); // Initialize the annotated
        ↪ mocks

        rt.setDay("Monday");
        rt.setCourse("Matematica");
        rt.setCourseType("seminar");
        rt.setTeacher("Dragos");
        rt.setGroupName("A1");
        rt.setRoom("101");
        rt.setStartHour(10);
        rt.setEndHour(12);

        String expectedString = "RegistrationTimetable{" +
                "startHour=" + rt.getStartHour() +
```

```
                ", endHour=" + rt.getEndHour() +
                ", room='" + rt.getRoom() + '\'' +
                ", day='" + rt.getDay() + '\'' +
                ", course='" + rt.getCourse() + '\'' +
                ", courseType='" + rt.getCourseType() + '\'' +
                ", groupName='" + rt.getGroupName() + '\'' +
                ", teacher='" + rt.getTeacher() + '\'' +
                '}';

        assertTrue(expectedString.equals(rt.toString()));
    }
```

- *TimetableUITest*: The main purpose of this test class is to ensure that all GUI capabilities are followed properly. A possible scenario for populating a TimetableUI instance is run. Besides is tested that a room for course is suitable for a course; a room for laboratory is suitable for a laboratory; tests the logical constraints of the timetable; if the discipline combo box is populated with the expected number of items; if the class combo box is populated correctly; if the timeslot for starting time combo box is populated correctly and overlapping scenarios.

```
@Test
public void testPopulateExamTable() throws SQLException {
    JDBCConnection jdbcConnectionMock =
    ↪ Mockito.mock(JDBCConnection.class);
    Connection connectionMock = Mockito.mock(Connection.class);
    PreparedStatement preparedStatementMock =
    ↪ Mockito.mock(PreparedStatement.class);
    ResultSet resultSetMock = Mockito.mock(ResultSet.class);

    String group = "Group1";
    String[] columns = {"id", "From - To", "Group", "Discipline",
    ↪ "Type", "Teacher", "Date", "Room"};
```

```
String[][] timetableData = {{"1", "10 - 12", "Group1", "Math",
↪  "Exam", "John Doe", "2023-05-20", "Room1"}};

Mockito.when(jdbcConnectionMock.getConnection())
    .thenReturn(connectionMock);
Mockito.when(connectionMock.prepareStatement(Mockito.anyString()))
    .thenReturn(preparedStatementMock);
Mockito.when(preparedStatementMock.executeQuery())
    .thenReturn(resultSetMock);
Mockito.when(resultSetMock.next())
    .thenReturn(true, false);
Mockito.when(resultSetMock.getInt("id"))
    .thenReturn(1);
Mockito.when(resultSetMock.getInt("start_hour"))
    .thenReturn(10);
Mockito.when(resultSetMock.getInt("end_hour"))
    .thenReturn(12);
Mockito.when(resultSetMock.getString("group_name"))
    .thenReturn("Group1");
Mockito.when(resultSetMock.getString("course"))
    .thenReturn("Math");
Mockito.when(resultSetMock.getString("course_type"))
    .thenReturn("Exam");
Mockito.when(resultSetMock.getString("teacher"))
    .thenReturn("John Doe");
Mockito.when(resultSetMock.getString("date"))
    .thenReturn("2023-05-20");
Mockito.when(resultSetMock.getString("room"))
    .thenReturn("Room1");

TimetableUI timetableUI = new TimetableUI();
```

```
    MockitoAnnotations.openMocks(this); // Initialize the annotated
    ↪ mocks

    timetableUI.populateExamTable();

    Assertions.assertEquals(0, timetableUI.jTable2.getRowCount());
}

@Test
public void testSuitableCourseRoom() {
    TimetableUI timetableUI = new TimetableUI();
    MockitoAnnotations.openMocks(this);
    String type = "Course";
    String room = "C1";
    boolean result = timetableUI.isRoomSuitable(type, room);
    assertTrue(result);
}

@Test
public void testIsCourseNotTotallyOverlapped() {
    // Test case 1: The courses are not totally overlapsed
    RegistrationTimetable oldOne = new RegistrationTimetable();
    oldOne.setStartHour(9);
    oldOne.setEndHour(11);
    oldOne.setRoom("L1");
    oldOne.setDay("Monday");
    oldOne.setCourse("Matematica");
    oldOne.setCourseType("Laboratory");
    oldOne.setGroupName("1A2");
    oldOne.setTeacher("Teacher A");

    RegistrationTimetable newOne = new RegistrationTimetable();
```

```
newOne.setStartHour(11);
newOne.setEndHour(13);
newOne.setRoom("L1");
newOne.setDay("Monday");
newOne.setCourse("Matematica");
newOne.setCourseType("Course");
newOne.setGroupName("1A3");
newOne.setTeacher("Teacher B");

TimetableUI timetableUI = new TimetableUI();
boolean expected = true;
boolean actual = timetableUI.isCourseNotTotallyOverlapped(oldOne,
↪  newOne);
assertEquals(expected, actual);

// Test case 2: courses are totally overlapping
oldOne = new RegistrationTimetable();
oldOne.setStartHour(14);
oldOne.setEndHour(16);
oldOne.setRoom("L1");
oldOne.setDay("Monday");
oldOne.setCourse("Matematica");
oldOne.setCourseType("Laboratory");
oldOne.setGroupName("1A2");
oldOne.setTeacher("Teacher A");

newOne = new RegistrationTimetable();
newOne.setStartHour(11);
newOne.setEndHour(17);
newOne.setRoom("L1");
newOne.setDay("Monday");
newOne.setCourse("Matematica");
```

```
    newOne.setCourseType("Laboratory");
    newOne.setGroupName("1A3");
    newOne.setTeacher("Teacher B");

    expected = false;
    actual = timetableUI.isCourseNotTotallyOverlapped(oldOne, newOne);
    assertEquals(expected, actual);
}

@Test
public void testPopulateGroupComboBox() throws SQLException {
    TimetableUI TUI = new TimetableUI();
    conn = JDBCConnection.getInstance().getConnection();
    Statement stmt = conn.createStatement();
    int groupNumber = 0;

    ResultSet rs = stmt.executeQuery("SELECT COUNT(*) AS total FROM
    ↪  groups;");
    while (rs.next()) {
        groupNumber = rs.getInt("total");
    }

    TUI.populateGroupComboBox();
    javax.swing.JComboBox<String> groupComboBox =
    ↪  TUI.getGroupComboBox();

    assertTrue(groupComboBox.getItemCount() == groupNumber);
    conn.close();
}
```

Chapter 4

Assertions

In order to further check the implementation of the application, assertions were inserted in the application code in order to check the preconditions, postconditions and the invariants of certain loops and functions; the assertions test:

- if the values returned or given as input are not null:

```
assert instance != null : "ConfigReader instance is null.";
```

- the integrity of the offered data as input/output for a key-value pair of the *ConfigReader* file:

```
public void setProperty(String key, String value)
{
    assert key != null : "Key cannot be null";
    assert value != null : "Value cannot be null";
    properties.setProperty(key, value);
}
```

- if the database connection remains opened after the instance creation:

```
assert instance != null : "Instance is null";
assert !instance.getConnection().isClosed() : "Connection is
↪  closed";
```

- if the start and end hours are from the specified interval (8-20):

```
public void setStartHour(int starHour) {
    assert starHour < 8 && starHour > 18 : "Start hour must have the
    ↪  value between 8 and 16";
    this.startHour = starHour;
}

public void setEndHour(int endHour) {
    assert endHour < 9 && endHour > 20: "End hour must have the
    ↪  value between 9 and 20";
    this.endHour = endHour;
}
```

- if the year number is greater than zero:

```
int years =
↪  Integer.parseInt(config.getProperty("setup.groups.years"));
String[] yearItems = new String[years];
for (int i = 0; i < years; i++) {
    yearItems[i] = Integer.toString(i + 1);
    assert yearItems[i] != null && Integer.parseInt(yearItems[i]) >
    ↪  0;
}
```

- if the timetable is not null after adding/retrieve records:

```
try {
    Connection conn = JDBCConnection.getInstance().getConnection();
    assert conn != null : "Database connection is null";
    PreparedStatement ptmt = conn.prepareStatement(sql);
    ptmt.setString(1, group);
    ResultSet rs = ptmt.executeQuery();
    assert rs != null;
    while (rs.next()) {
```

```
            int i = 0;
            String[] item = new String[columns.length];
            item[i++] = String.valueOf(rs.getInt("id"));
            item[i++] = new
            ↪  StringBuilder().append(rs.getInt("start_hour")).append("
            ↪  -").append(rs.getInt("end_hour")).toString();
            item[i++] = rs.getString("group_name");
            item[i++] = rs.getString("course");
            item[i++] = rs.getString("course_type");
            item[i++] = rs.getString("teacher");
            item[i++] = rs.getString("date");
            item[i++] = rs.getString("room");
            timetable.add(item);
        }
        assert !timetable.isEmpty();
    }
```

- if the total number of disciplines and the total number of exams added together do not exceed the total size of the timetable:

```
while (rs.next()) {
    int i = 0;
    String[] item = new String[columns.length];
    item[i++] = String.valueOf(rs.getInt("id"));
    item[i++] = new
    ↪  StringBuilder().append(rs.getInt("start_hour")).append(" -
    ↪  ").append(rs.getInt("end_hour")).toString();
    item[i++] = rs.getString("group_name");
    item[i++] = rs.getString("course");
    item[i++] = rs.getString("course_type");
    if(item[i - 1].split(" ").length > 1 && item[i - 1].split("
    ↪  ")[1].equals("Exam")){
        totalExams++;
```

```
    }
    else{
        totalDisciplines++;
    }
    item[i++] = rs.getString("teacher");
    item[i++] = rs.getString("day");
    item[i++] = rs.getString("room");
    timetable.add(item);
}
assert (totalDisciplines + totalExams) <= timetable.size();
```

- if the current index does not exceed

```
String[][] dataTimetable = new String[totalDisciplines][];
String[][] dataExamtable = new String[totalExams][];

int i = 0;
int ii = 0;
for(int j = 0; j < timetable.size(); j++){
    String[] item = timetable.get(j);
    if(item[4].split(" ").length > 1 && item[4].split("
    ↪  ")[1].equals("Exam")){
        assert i <= dataExamtable.length;
        dataExamtable[i] = item;
        i++;
    }
    else{
        assert ii <= dataTimetable.length;
        dataTimetable[ii] = item;
        ii++;
    }
}
```

- if the start hour is before 20:

```
public void updateTimeSlotEndComboBox(int startHour) {
    int numberOfHours = 20 - startHour;
    String[] items = new String[numberOfHours];

    for (int i = startHour + 1, j = 0; i <= 20; i++, j++) {
        items[j] = String.valueOf(i);
    }
    assert startHour < 20;

    DefaultComboBoxModel model = new DefaultComboBoxModel(items);
    timeSlotEndComboBox.setModel(model);
}
```

- if the number of possible courses is evenly distributed:

```
int sumRoomTypes = Arrays.stream(separateRoomTypes).mapToInt(value ->
↪  Integer.parseInt(value.split("-")[1])).sum();
if (sumRoomTypes != 100){
    LOG.severe("Setup room types are not evenly distributed");
    return;
}
assert sumRoomTypes == 100 : "Setup room types are not evenly
↪  distributed";
```

Chapter 5

Contributions

- **Constantinescu George-Gabriel**:
    1. database design and population, export timetable functionality, exam timetable design and implementation, constraints for exams implementation
    2. database setup unit testing and assertions
    3. Unit testing chapter from the documentation
    4. Javadoc for unit testing classes
- **Filimon Dănuț-Dumitru**:
    1. GUI design and implementation, configuration file reader implementation, adding and removing records from timetable, database singleton implementation
    2. GUI unit testing and assertions
    3. Project description chapter from the documentation
    4. Javadoc for the GUI implementation
- **Onofrei Tudor-Cristian**:
    1. database design, exam timetable design and implementation, constraints for exams implementation, database population script
    2. timetable records unit testing and assertions
    3. Assertions and User Guide chapters from the documentation
    4. Javadoc for the configuration reader implementation

- **Lupu Cezar-Justinian**:
  1. timetable records design and implementation, submit constraints checks, logging system
  2. registration timetable, database setup unit testing and assertions
  3. User Guide chapter from the documentation
  4. Javadoc for constraints implementation and unit testing classes
- **Baciu Dragoș**:
  1. Combo box synchronizations, GUI design and implementation, timetable records design and implementation, submit constraints checks
  2. Configuration reader unit testing and assertions
  3. Unit testing chapter from the documentation
  4. Javadoc for the timetable export implementation