



TECHNICAL UNIVERSITY OF DENMARK

02160 AGILE OBJECT-ORIENTED SOFTWARE DEVELOPMENT

DEVELOPMENT OF DTU EDUCATION-MANAGEMENT SOFTWARE

Final Project Report

Written by:

Alison Davis s175909
Agathe Granier s176154
Clara Grau s176104
Moritz Hetzel s175942
Mingyuan Zang s171352
Jingkun Zhu s175886

Supervisors:

Barbara Weber
Andrea Burattin

May 1, 2018

Contents

1	Introduction	2
2	Design Objective	2
3	Methods and Tools	2
3.1	Behavioral-Driven Development	2
3.2	Cucumber tests	3
3.3	Continuous integration	3
4	User Stories and Resulting Scenarios	3
5	Class Design	6
6	Design Choices	6
6.1	Security	7
6.2	Collections	7
6.3	AbstractTableModel Extensions	7
6.4	Enumerations	8
6.5	Persistency Layer	8
6.6	Response Message	8
6.7	Date Class	9
6.8	Imported Superclasses	9
6.9	PDF Generation	9
7	UML Class Diagrams	10
8	GUI Design	10
8.1	Views	11
9	Design pattern: Model View Controller Implementation	11
10	Coverage	11
11	Work Distribution	12
12	User Manual	13
13	URLs to Screencast	14

1 Introduction

This report details the use of Agile Object-Oriented Programming principles to develop a new software for DTU's education management system. As developers, the initial goal was to implement a code model of the education registration system by realizing various scenarios based on user stories. These scenarios were individually implemented and tested with Cucumber and JUnit tests using a system of continuous integration. Following the creation of the core, the development team improved the code's ability to interact with users through the implementation of a graphic interface. This was done while still adhering to agile principles through the use of model view controllers.

2 Design Objective

The primary design of the program is inspired by the needs DTU would actually have to create students, teachers, classes, and study programs. It also allows these objects to interact with numerous dependencies—including, but not limited to, the assignment of: students and teachers to classes, classes to study programs, study programs to students, previously passed courses to students, and students to current courses. To allow the user to interact easily with the software, the final product objective has a graphical user interface which presents buttons, icons, and visual indicators to neatly accept user input and display windows able to describe this information stored in the program. To increase the simplicity and adaptability of the design, the design intent for the interface is for it to be user-independent. This means it does not specialize access depending on the user, whether that be a student, teacher, or administrator. This decision favors simplicity of design over security, as any user of the program is permitted access to all of its functionalities. Therefore, it is assumed that only the administrators will be granted access to the study management system.

3 Methods and Tools

One of the goals of the program is for it to be developed using agile object-oriented methods. Agile refers to an approach to software development that is collaborative and adaptive. Several specific methods and tools were employed specifically for this purpose.

3.1 Behavioral-Driven Development

The initial brainstorming of the functionalities of the program was done by generating many user stories on Trello for any possible way different subjects could want to use the program, as shown in Section 4. This process is known as behavioral-driven development (BDD). It uses these specific examples of necessary features to create clear design goals and avoid unwanted assumptions. These examples can then be directly implemented using Maven, a build management tool, to simultaneously automate the creation of code and unit-tests. Maven automates compilations, unit tests and deployments of applications that make up the project. Moreover, Maven manages dependencies. Of course, the repository/src only has to contain source files created by the developers for the project; that is why, external libraries used by the project have to be linked to other Maven artifacts and do not have to be copied in the repository/src of the project. All these links are defined in the pom.xml. After that, the Maven plugin manages the dependencies. These dependencies will download on the remote repositories the jar files indicate as dependencies if they are not in the local repository.

3.2 Cucumber tests

Cucumber is an open source BDD tool, which manages to express the behavior of a system in plain language. It allows the transformation of scenarios of a user story into automatically generated java tests. This conversion is possible thanks to a framework of JUnit tests. Each step of a scenario needs to have a matching step definition and then corresponding java methods. Thus, Cucumber allows automatic validation and supports JUnit. Thereby, it also functions as a guide for the programmer as it helps to keep track of the development progress. That is why it is also commonly known as an "executable specification".

3.3 Continuous integration

Throughout the creation of the project, the development team also followed the agile practice of continuous integration. That is to say, a method that ensures constant "production readiness." By developing user-stories into scenarios one at a time and not continuing until each individual function is fully operational and passing all new and previous tests, it ensures that the system is always running.

Using continuous integration involves a variety of practices, all common for software development in industry. First, this method seeks to never introduce regression into the code and therefore minimize cost of correction. In this way, continuous integration assures the quality and fluidity of development. This method refers to a practice of never introducing partially-developed or failing code, committing changes, controlling individual steps, and verification of every step by means of passing tests.

To implement continuous integration amongst the group of developers, a source code repository and an automated build tool were needed—leading to the implementation of Git and Maven.

The continuous integration method has lots of advantages. For one thing, each scenario, method, class or test is only added to the project only if it is working. For this reason, there will never be an impossible search for a mistake throughout the entire project at the end. Moreover, all the members of the group can work on the same project at the same time on their own computers, so long as they "pull" regularly to have the new version of the project.

On the other hand, this method of version control also requires attention and constant communication. At the early stages of the development process, difficulties with different user's changes made it impossible to merge changes in the Git repository due to conflicts in the code. Thus, at the end of the project, prior to the start of any work, the whole repository was always pulled. Furthermore, an on-going group chat was used to report what classes were being worked on by whom, so no two people were ever making changes on the same parts of the code at the same time.

4 User Stories and Resulting Scenarios

A selection of some of the previously mentioned User-Stories and their equivalent conversions into scenarios are shown below.

returnStudent.feature:

User Story 1: I want to search a student by first name in the student register where he and an other student with the same first name have already signed up. So I enter the first name in the search bar and the register returns a list with two student with the same first name.

```
@tag1
Scenario: Return student by first name
  Given a student with a first name "Alison" and last name "Davis" and with birthdate 08-27-1996
  And a student register where the student already is signed up
  And a student with a first name "Alison" and last name "Johnson" and with birthdate 02-14-1994
  And sign-up
  And a student with a first name "Moritz" and last name "Hetzel" and with birthdate 08-09-1996
  And sign-up
  When student register is searched by first name "Alison"
  Then an array list should be returned containing two students with first name "Alison"
```

returnTeacher.feature:

User Story 2: I want to search a teacher by serial number in the teacher register where he is already signed up. So I enter the serial number in the search bar and the register returns a teacher with the correct first name and last name.

```
@tag1
Scenario: Return teacher by serial number
  Given a teacher with a first name "Barbara" and last name "Weber"
  And a teacher register where the teacher already is signed up
  And a teacher with a first name "Jason" and last name "Weber"
  And teacher sign-up
  When teacher register is searched by serial number 100002
  Then a teacher with first name "Jason" and last name "Weber" should be returned
```

studentEnrollment.feature:

User Story 3: I want to enter the information of the course in which I want to enroll a student, but he does not have the prerequisites for this course, so the enrollment failed and I receive a error message.

```
@tag3
Scenario: Enrollment of student failed because of missing prerequisite
  Given a student with a first name "Moritz" and last name "Hetzel" and with birthdate 08-28-1996
  And a student register where the student already is signed up
  And a course register containing a course with name "Programming" and course time "1A" and ECTS points 10 and type "BSc"
  And containing the course with name "Physics" and course time "2A" and ECTS points 10 and type "BSc"
  And containing the course with name "Math1" and course time "2B" and ECTS points 10 and type "BSc"
  And course "Programming" with course time "1A" requires 910002 as prerequisite
  And course "Programming" with course time "1A" requires 910003 as prerequisite
  And student has passed course 910002
  When student enrolls in course with name "Programming" and course time "1A"
  Then an error message should be displayed that the enrollment failed
```

addCourse.feature:

User Story 4: I want to enter a new course and all its information in an empty course register, so the course is added to the course register and receives a course number.

```
@tag1
Scenario: Adding one course to an empty course register
  Given a course with name "Programming" and course time "1A" and ECTS points 10 and type "BSc"
  And an empty course register
  When adding course to the course register
  Then the course is created with a course number 910001
  And the course should be added to the course register
```

addProgramme.feature:

User Story 5: I want to enter the mandatory course "Programming" to the programme "Computer Science", so the mandatory course is added to programme.

@tag2

Scenario: Adding a mandatory course to a programme

Given a course register containing a course with name "Programming" and course time "1A" and ECTS points 10 and type "BSc"
And a programme register containing a programme with name "Computer Science" and type "BSc"
When adding the course as mandatory course to a programme
Then the mandatory course should be added to the programme

changeCourseInformation.feature:

User Story 6: I want to add a teacher assistant to a course, that he has already passed. So I enter the information about the course I want him to teach and he is added to the course as a teacher assistant.

@tag1

Scenario: Add teacher assistant to a course

Given a student with a first name "Moritz" and last name "Hetzel" and with birthdate 08-28-1996
And a student register where the student already is signed up
And a course register containing a course with name "Programming" and course time "1A" and ECTS points 10 and type "BSc"
And student has passed course 910001
When adding student as a teacher assistant to course 910001
Then the student is added as a teacher assistant to the course 910001

coursePlanner.feature:

User Story 7: I want to see the courses of a student, who is enrolled in a programme, that he has not completed yet. So I will receive a message displaying the missing courses of this programme.

@tag1

Scenario: Show the missing courses of a student

Given a course register containing a course with name "Programming" and course time "1A" and ECTS points 10 and type "BSc"
And containing the course with name "Physics" and course time "2A" and ECTS points 10 and type "BSc"
And containing the course with name "Math1" and course time "2B" and ECTS points 10 and type "BSc"
And a programme register containing a programme with name "Computer Science" and type "BSc"
And adding the course with name "Physics" and course time "2A" as elective course to a programme
And adding the course with name "Math1" and course time "2B" as mandatory course to a programme
And a student with a first name "Jingkun" and last name "Zhu" and with birthdate 06-04-1996
And a student register where the student already is signed up
And student enrolls in programme
And student has passed course 910001
When student queries the system the missing courses
Then the message about missing courses should be displayed

returnCourse.feature:

User Story 8: I want to search a course by keyword in the description or the course name, in a course register where a course with this keyword exists. So the register returns the course containing the keyword in its name or its description.

@tag1

Scenario: Return course by keyword

Given a course register containing a course with name "Programming" and course time "1A" and ECTS points 10 and type "BSc"
And adding description "This course teaches you basic object-oriented programming skills. The used programming language is Java." to course
And a course with name "Math1" and course time "2B" and ECTS points 10 and type "BSc"
And adding description "This course teaches you mathematic fundamentals. It gives you deeper insights to algebra and geometry." to course
And adding course to the course register
When searching the course register for a course with keyword "Java" in the description or the course name
Then course with name "Programming" is returned in the array list

User Story 9: I want to search a courses by type, in a course register where courses with this type exist. So the register returns courses with the type I have searched for.

@tag3

Scenario: Return course by type

Given a course register containing a course with name "Programming" and course time "1A" and ECTS points 10 and type "BSc"
And a course with name "Math1" and course time "2B" and ECTS points 10 and type "MSc"
When searching the course register for a course with type "BSc"
Then course with name "Programming" is returned in the array list

5 Class Design

While the primary functionalities required for the system were given, the class design could be approached in various ways. The software designed by the International Java Team was created using consideration of the SOLID design principles. The project does not demonstrate all these concepts, but they have motivated other decisions. The software has been developed with the purpose of making the software more understandable, flexible and maintainable based on five key design concepts:

1. **Single Responsibility Principle:** a class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class).
2. **Open/Closed Principle:** software entities should be open for extension, but closed for modification.
3. **Liskov Substitution Principle:** objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
4. **Interface Segregation Principle:** many client-specific interfaces are better than one general-purpose interface
5. **Dependency Inversion Principle:** one should depend upon abstractions, concretions.

One implementation of these principles is seen in the creation of a "Person" superclass, which is extended by both the Student and the Teacher classes. The advantage of creating this generalized class is that it increases the reusability of the code without having to write very similar methods. An example of how this could later be implemented is by including DTU staff and having person also extend new classes such as librarian or secretary. This idea has similar goals to the Open/Closed principle in that you do not directly have to change the person code to change all the objects that use it. This separation prevents the introduction of error into already working code. This relationship also respects the Liskov substitution Principle in that all features of person are applicable to both of the classes that extend person, with no unnecessary overlap.

Similarly, the code consists of many different classes. The large number of classes originated from a desire to adhere to the single responsibility principle, where new responsibilities and features led to new classes. This includes separate Registers for students, teachers, courses, and programmes on top of the separate classes for each of these objects respectively. By treating the register as its own object, it makes it possible not only to hold a store of these objects somewhere, but it prevents the generation of sign up information such as student, serial, or course numbers from happening within the actual object class. By designating the creation of these features to the registers instead of the student, teacher, and course classes themselves, it prevents these objects from being contaminated with features that are not truly their responsibility. While students do possess emails, it should not be a responsibility of a student object to generate an email. It should simply be a feature of the student.

6 Design Choices

There are many decisions involved in coding. This section explores the decisions that were made involving security, collection usage for storing data, class extensions, dependencies, and other program decisions.

6.1 Security

Security within the classes was approached differently depending on the variable. Taking the person class for example, email and address are set as private variables so they may not be seen or changed by any other classes even in the package. On the other hand, first and last name are protected because they need to be visible to other classes that extend Person, but not to the whole world. This helps increase the safety of variables that should only be changed or used in certain classes. Another approach to security of the system is by user-input checks. These were added for all direct user inputs on the multi-option panes. The system is kept on track by not allowing users to enter anything but integers in fields such as ECTS or stopping the user from entering numbers as a name. The code's smoothness was also improved by issuing an error message to the user in these situations and keeping the user in a loop until he/she enters acceptable inputs or clicks cancel.

6.2 Collections

The program requires data storage of various groups of objects. For the general registers of students, teachers, courses, and programmes hash sets are used. These were deemed the most appropriate storage system due to the fact that they do not allow repeat items to be entered. They also have high efficiency in searching for elements in the set because of the hash code. The `equal()` method considers two objects equal only if the two objects have the same hash code, otherwise, there is no check of the equality of the two objects. Since exact repeats of any of these items was not desirable, this was a good solution. Similar collections exist within each student object such as current course list, passed course list, and missing course list. These are also created using hash sets for the same reason.

While the convenience of these sets in eliminating repetition was helpful, assigning the hash sets to the tables was not as practical as assigning array lists. An array list was not considered a good choice for the initial creation of these lists because it allows repetition, but the benefit of these when displaying the data is that they have order, unlike a hash set. Therefore, array lists were created from existing hash sets but only for table updates.

Another type of collection used was a `HashMap <Integer, Integer>`, with a key and the student's grade for of each course. The course number is chosen as key. This is very practical because for each student, the user can search the student's grade of a course in his `HashMap` using the course number. Moreover, each student has only one grade to each course (to know if he passes the course or not), so there is no duplication. The fact that there is no duplication is very important because in a `HashMap` the keys must be unique or else the second overwrites the first one.

6.3 AbstractTableModel Extensions

All of the register classes extend the `AbstractTableModel`, a class that is imported as a feature of Java Swing. This extension provides some conveniences for generating `TableModelEvents` and sending them on to the listeners. To create a subclass of `AbstractTableModel` it is only necessary to implement three methods:

- `public int getRowCount();`
- `public int getColumnCount();`
- `public Object getValueAt(int row, int column);`

While these are the minimum required implementations, the register classes in this project

implement the following functions:

- `int getRowCount()`
- `int getColumnCount()`
- `String getColumnName(int columnIndex)`
- `Object getValueAt(int rowIndex, int columnIndex)`
- `Class getColumnClass(int columnIndex)`
- `boolean isCellEditable(int rowIndex, int columnIndex)`
- `setValueAt(Object aValue, int rowIndex, int columnIndex)`

These functions of `AbstractTableModel` are redefined in the classes that control the tables to adapt them to our classes and our software. The additional features supported by `AbstractTableModel` are used to make the necessary features editable such as the address for students and teachers. The abstract table model also automatically creates the columns of the table given headers. These Strings are private and final because they do not change and they can be used without an instantiated object.

6.4 Enumerations

Two enumerations were used to enhance the comprehensibility of the code and to provide type-safe storage of constant data. The first is `Level` which is used for teachers. `Level` has 3 options: Assistant, Associate, or Professor. The other enumeration used is `Type` referring to the type of degree a class or programme is classified as and also has 3 possibilities: BSc, MSc, or PhD.

6.5 Persistency Layer

In 'ProgramRegister' and 'fileRegister' the optional functionality 4 is implemented. Data has to be stored to keep it even if eclipse is closed and then opened again. That is what `IOStream.java` is used for. It allows Objects to be put into a file in order to make a persistency layer. In the files of our persistency layer, bytes are expected, therefore, it is necessary to convert the data. 'inputCollection' allows the reading of data into the file (for example a text file) and 'outputCollection' allows the writing of new data in the file (for example, to add a course). When something is written, the file has to be refreshed. At the beginning of execution, it is necessary to check if the file '.xml' exists, if it does not exist, this file will be created later automatically when the output stream is called.

6.6 Response Message

The `ResponseMessage.java` class and its function was created to prevent all malfunctions of the system and to display an error message in these cases. For example, if a student who has already signed up wants to sign up again, our response message is going to be "Student is already in the register". This error message allows Java not to stop the code if it detects an error and to inform programmers about what the user has done wrong.

Moreover, when the action is successfully made, the response message still informs the programmer. This feature was helpful in testing the scenarios by getting descriptive feedback from the program of what exactly happened.

6.7 Date Class

It was decided that date should be its own object, not just a String, since it is repeatedly used and could be implemented in more classes if the code were to be extended in the future. For the Date.java class, the developers implemented a vector of integers which contains the number of days for each month. This table is private, static and final. 'Final' prevents the modification of the variable.

Then, the software has to check if the date entered by the user is valid. Therefore, a method which returns a true if the date is valid is run. This method checks if the number of the month is between 1 and 12, if the number entered for the day is between 1 and the number of day of the month entered. We check also if the year entered is a leap year, in this case the 29th of February is also a valid date. In this method, the exception 'InvalidDateException' is thrown.

The use of exceptions was another programmer decision unique to this model. Exceptions are also Objects in java. To create a new and adapted exception, it is necessary to create a new class, for example, 'public class InvalidDateException' which extends Exception and create a simple constructor. Whenever a method has to throw an exception, the method must declare it using throw statement. For example, in Date it is written:

```
throw new InvalidDateException("Invalid birthdate");
```

Therefore, when an exception is thrown, we return the exception to InvalidDateExcpetions (the exception class) and then, this class manages the exception.

6.8 Imported Superclasses

Two examples of the use of imported super classes are seen in Course.java, which implements Serializable and Comparable super classes.

Comparable defines a natural ordering. What this means that the code defines when one object should be considered "less than" or "greater than" another. For example, to sort a bunch of integers, is simple because it is only necessary to put them in a sorted collection. However, sorting custom objects is not so easily defined. Now the easiest way to sort them is to define them with a natural ordering by implementing Comparable, which means there is a standard way these objects are defined to be ordered. This same goal can also be achieved by defining a comparator. The Serializable interface allows the representation of an object as a sequence of bytes including the type of the object, data stored in its attributes and the type of these. It is required to compile the class list pdf file.

6.9 PDF Generation

It was chosen to implement optional functionality 5, which is meant to employ methods that allow the production of a PDF with the list of all students enrolled in a course. This is completed in the GeneratePDF.java.

To make this possible, pdf.PdfWriter and pdf.PdfTable are imported into GeneratePDF.java. The first step is the creation of a new document and a new 'writer' to put information on this document. Using the methods implemented in PdfTable, the developer can choose the width and the spaces before and after the table. The different loops in the code situate the data in an aesthetically pleasing manner in the PDF.

7 UML Class Diagrams

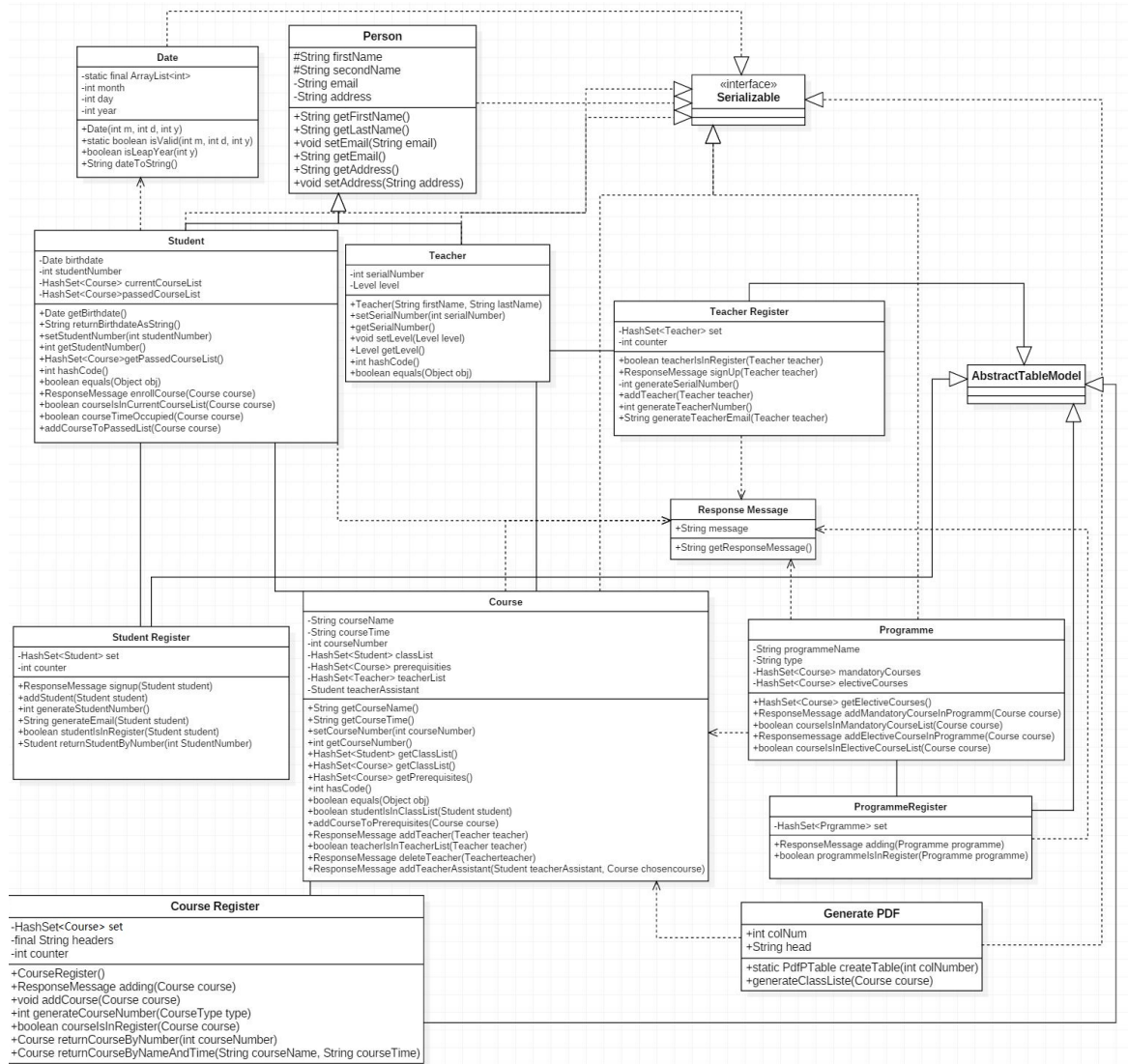


Figure 1: UML Class Diagram

8 GUI Design

The GUI design was completed separately and simultaneously with the model design. While all code features were considered during GUI design, the attachment of the full model to the GUI called for many alterations due to oversights or misconceptions regarding how the program needs to interact with the user. While some changes were still necessary for this reason, the majority of the view codes were able to remain unedited during the connection process. This was made possible by the creation of controller classes. In a model-view-controller (MVC) pattern, like the one used here, models and views can both maintain their individual integrity. This increases the reusability of code and supports the idea of continuous integration by never dismantling model or GUI code that is already working.

8.1 Views

The views are responsible for taking user input and recognizing user actions. All classes in the *application.Views* folder are responsible for a single view, each having just one top-level container. There are five primary types of views used in the program: the selection view, inventory views, detail views, multi-option input panes, and search views. Each of these views contain different combinations of mid-level containers such as JPanels and JScrollPanels to organize the view. Finally, each has a unique organization of widgets designed to accept forms of user input. While the types of views are similar for student, teacher, course, and programme—each shows different information and requires unique inputs, resulting in the need for each to have its own view class. On the other hand, a common functionality amongst all the views is the use of the `gridBagsLayout`. This allows components to be constrained to an invisible grid in each view port.

9 Design pattern: Model View Controller Implementation

In accordance with the MVC pattern as mentioned in Section 8, the code is divided into three main subsections: the model, the view, and the controller. The relation could be seen in figure 2.

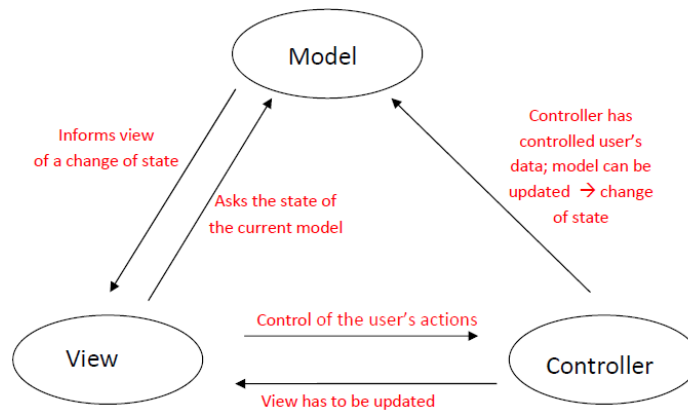


Figure 2: MVC Pattern

In the project, the view represents the graphic interface that the user can see, the model contains the data (it is often, one or many java objects). The controller is also an object and makes the connection between the view and the model when the user makes an action. This object controls the data.

10 Coverage

Based on the BBD principles, several scenarios have been set to realize the user stories and different functions of the classes. To cover as many scenarios as possible, failure scenarios have been created. For example, if a number is entered as the first name, then an error message will pop out to cancel this turn of sign-up.

Based on these scenarios, some methods like "HashCode" and "Equals" still can not be covered. To test these parts, JUnit tests have also been added. The final coverage of only the model is shown in Figure 3.
















▼ application.model		96.7 %	2,523	86	2,609
> CourseRegister.java		94.4 %	357	21	378
> StudentRegister.java		94.1 %	332	21	353
> TeacherRegister.java		94.6 %	371	21	392
> ProgrammeRegister.java		91.4 %	148	14	162
> Course.java		97.9 %	366	8	374
> Student.java		99.8 %	408	1	409
> CourseType.java		100.0 %	48	0	48
> Date.java		100.0 %	147	0	147
> InvalidDateException.java		100.0 %	4	0	4
> Level.java		100.0 %	48	0	48
> Person.java		100.0 %	23	0	23
> Programme.java		100.0 %	141	0	141
> ResponseMessage.java		100.0 %	9	0	9
> Teacher.java		100.0 %	121	0	121

Figure 3: Coverage

11 Work Distribution

In early stages, pair-programming was imperative to getting all group members working cohesively and not over-writing one-another's work. Later the responsibilities became more diverse. Table 1 shows individual responsibilities of the developers:

Table 1: Group member responsibilities

Group Member	Responsibilities
Alison Davis	Initial model views, model view controllers, work on student/teacher/course scenarios, additional JUnit testing, Report sections
Moritz Hetzel	Improve the graphic interface, Mandatory1, Mandatory2, Mandatory3, Programme view, Controller, Optional6, Report
Mingyuan Zang	Improve graphic interface, Mandatory2, Optional5, Mandatory3, JUnit testing, StarUML, Report
Jingkun Zhu	Improve the graphic interface, Mandatory2, Mandatory4, Optional1, Optional3, Optional4, JUnit testing, Report
Clara Grau	Mandatory1, Mandatory3, Users stories, Design choices, explanation of the implementation and Maven (report)
Agathe Granier	Mandatory2, Mandatory3, explanation of the implementation of the software and MVC (report), StarUML, Optional5

12 User Manual

Run the application and an interface with four buttons will be shown. Each of the buttons can achieve functions as shown below.

- Student:

1. Sign-up: Enter the Student interface and click the button [Add Student] to input the student name, address and birthdate. After the confirmation, the student number and student email based on the number will automatically appear. Student can be removed by selecting the student and click [Remove Selected Student].
2. Change Information: To change information, user can double click the address he wants to change and enter the new address.
3. Enroll Programme: Student can enroll programme by clicking on the button [Enroll Programme] and enter the programme name. After the enrollment succeed, the course planner showing the missing courses of this programme can be checked by clicking [Detail View].
4. Register Course Grades: User can click [Add Completed Course] and enter the course number and grade. If the course number is invalid, an error message will appear. The grade ranks from 0 to 10. The grade lower than 3 means the student fails to pass the course. The passed courses can be checked in [Detail View].
5. Course Calendar: Having the student enrolled in courses, the course calendar of the student can be checked in [Detail View].

- Teacher:

1. Sign-up: Enter the Teacher interface and click the button [Add Teacher] to input the teacher name, office address and choose the teacher level. After the confirmation, the serial number and name based email will be generated automatically. Teacher can be removed by selecting the teacher and click [Remove Selected Teacher].
2. Change Information: Address and teacher level can be changed by double clicking the targeted address or level and enter the new information.

- Course:

1. Add Course: Enter the course interface and click the button [Add Course] to input the information like course name, course time, type, etc. After the confirmation, course number will be generated automatically. The course can be removed by clicking [Remove Selected Course].
2. Add Teacher: Select the course and click [Add Teacher]. Then enter the teacher's serial number. The responsible teacher can be removed by clicking [Remove teacher] and entering the serial number. Check the result in [Detail View].
3. Add Student Teacher: Select the course and click [Add Student Teacher]. Enter the desired student number. If the student haven't passed the course before, then he won't be added to the list. The student teacher can be removed by clicking [Remove teacher] and entering the student number. Check the result in [Detail View].
4. Add Prerequisites: Click [Add Prerequisite] and enter the course number to set prerequisites. The result can be checked in [Detail View].
5. Enroll Student: Select the course and click [Enroll Student]. Enter the student number and check the student list in [Detail View].
6. Print Participant List: The participant list can be saved as pdf file to default path by

clicking the [Participant List].

- Programme:

1. Add Programme: Enter the programme interface and click [Add Programme]. Enter the programme name and choose the type. The programme can be removed by clicking [Remove Selected Programme].

2. Add Courses: Click [Add Mandatory Course] and enter the course number to set mandatory course list. Similar way to set the elective course list. The results can be checked in [Detail View].

- Search Functions:

1. Student: By clicking the [Search] in student interface, students can be searched by first name, last name and student number.

2. Teacher: By clicking the [Search] in teacher interface, teachers can be searched by first name, last name, email and serial number.

3. Course: By clicking the [Search] in course interface, courses can be searched by keywords in course name and course description or by course type or number. To find the course with most students attending, click [Courses with most students attending]. To view the courses' average grades, click [Courses with highest average grades].

- Persistency Layer:

The persistency layer function can save the data to the default saving path when the user closes the application. The data will remain the same when the user reopens the application. To get an empty register, the user can delete the ".xml" file in the saving path.

13 URLs to Screencast

1. Example of creating a course and showing the search functionalities: <https://youtu.be/vGks8oB-GXg>

2. Example of enrolling a student: <https://www.youtube.com/watch?v=YSJdZf6kySI>

3. Example of generating a PDF: <https://youtu.be/c7M0hieQSVs>

4. Example of using the persistency layer: <https://www.youtube.com/watch?v=1tBM-f2dko8>

5. Example of teacher sign-up and course details: <https://www.youtube.com/watch?v=CTzLXypcHYs>