|  |
| --- |
| MTU Kerry |
| Implementing design patterns |
| Second year Java project |

|  |
| --- |
| Ivan Segade Carou t Number T002193587  [Date] |

Contents

[1 Introduction 2](#_Toc152769287)

[2 GitHub repositories 5](#_Toc152769288)

[3 Patterns used 6](#_Toc152769289)

[3.1 Strategy pattern 6](#_Toc152769290)

[3.1.1 Description of the implementation 7](#_Toc152769291)

[3.1.2 Open method 7](#_Toc152769292)

[3.1.3 Save method 8](#_Toc152769293)

[3.1.4 Search method 9](#_Toc152769294)

[3.2 Justification 10](#_Toc152769295)

[3.3 Drawbacks 11](#_Toc152769296)

[3.2. Adapter pattern 12](#_Toc152769297)

[3.2.1 Description of the implementation 12](#_Toc152769298)

[3.2.2 Justification 13](#_Toc152769299)

[3.2.3 Drawbacks 14](#_Toc152769300)

[3.3 Model View Controller (MVC) 15](#_Toc152769301)

[3.3.1 Description of the implementation 16](#_Toc152769302)

[3.3.2 Justification 18](#_Toc152769303)

[3.3.3 Drawbacks 19](#_Toc152769304)

[4. Testing 19](#_Toc152769305)

[5. Conclusion 23](#_Toc152769306)

[6. References 24](#_Toc152769307)

# 1 Introduction

For this assessment the project I picked was my second year Java project. This is based on project idea formulated when I was a fairly new student. I have attended other colleges before MTU and therefore consider myself experienced in this area. I have seen many instances in my past where existing systems covering this area have been too complex or functionally weak.

The original version of this Academy Management system was developed two years ago. Considering that my level of knowledge and experience was more limited at that time I am reviewing the system with design patterns and my own improved coding knowledge. The aim of this project is to improve the internal code structure of this existing project to streamline the code and improve its overall quality.

The project is a simple Academy Management system. The user can manage course, student and teachers’ details. The user can add and update student, teachers and courses. Also, the program managed the student assignments for each course.

The application saves the information in three different files: teachers.data, students.data and courses.data.

First an abstract class called Person was created. This class implements the Serializable interface in order to be able to handle the file system. Also, this class contains the follow private attributes:

* Strings id, name, surname, address, town, county, email and phone.
* Char: gender.
* Gregorian Calendar type dob.

The Person class holds all the getters and setters for the corresponding attributes and overrides the to String method to customise it for the project.

Then two more classes called Student and Teacher were created. These two new classes inherit from the Person class. Therefore, both Student and Teacher share the same attributes and getters and setters. The Student class does not add any new attribute, however the Teacher class has a new attribute for the department where the teacher works, as well as the corresponding getter and setter.

Moreover, another class called Course was created. This class implements the Serializable interface in order to be able to handle the file system. This class contains the follow private attributes:

* Strings id, name.
* Doubles: price (how much the student pays for the course) and pay (how much the teacher is paid).
* Teachers: teacher (the teacher that teaches the course).
* Students: students [] (An array that contains the students that are in the course).

A class called Academy was created to host the main window of the project. This window just contains a menu with the follow options: File, Teacher, Courses and Students.

The File options just has the Exit menu, where once pressed it displays a window message asking if the user wants to exit from the application. The Teachers and Students menus have the same options: Add, Update and View. For these options, a window with the corresponding form is displayed.

On the Add option, the corresponding form is displayed. There, the user can enter the required fields. Also, there are two buttons: Add and Cancel. The Cancel button closes the window without saving any data. The Add button saves the data after validating all the fields.

On the Update option, the corresponding form is displayed. First, the form asks the user for an Id to search the corresponding option. Then there are two buttons: Search and Cancel, The Cancel button closes the window without doing more operations. The Search button searches the corresponding element on the file. In the case of not finding the element, an error message is displayed. If the searched element is found, a new form is displayed with all the corresponding data. There, the user can modify any field, but the Id. Then, there are two buttons: Update and Cancel. The Cancel button closes the window without updating any data. The Update button updates the data on the corresponding file.

On the View option, the corresponding form is displayed. First, the form asks the user for an Id to search the corresponding option. Then there are two buttons: Search and Cancel. The Cancel button closes the window without doing more operations. The Search button searches the corresponding element on the file. In the case of not finding the element, an error message is displayed. If the searched element is found, a new window message is displayed with all the data of the searched element. In the case of not finding the element, an error message is displayed.

On the Courses menu there is an option called Assign Student. On this option there is a form that asks the user for a course and student. Then, in the case of not finding either course or student, and error message will be displayed. Otherwise, the user is able to enter the student and the course and thereby assign or unassign the student to the course. It displays error messages for duplicated assignments or assignments that do not exist. There is also a Cancel button that closes the window without doing more

In Figure 1.1 the original application before implementing the design patterns is outline while Figure 1.2 after implementing the design patterns is shown

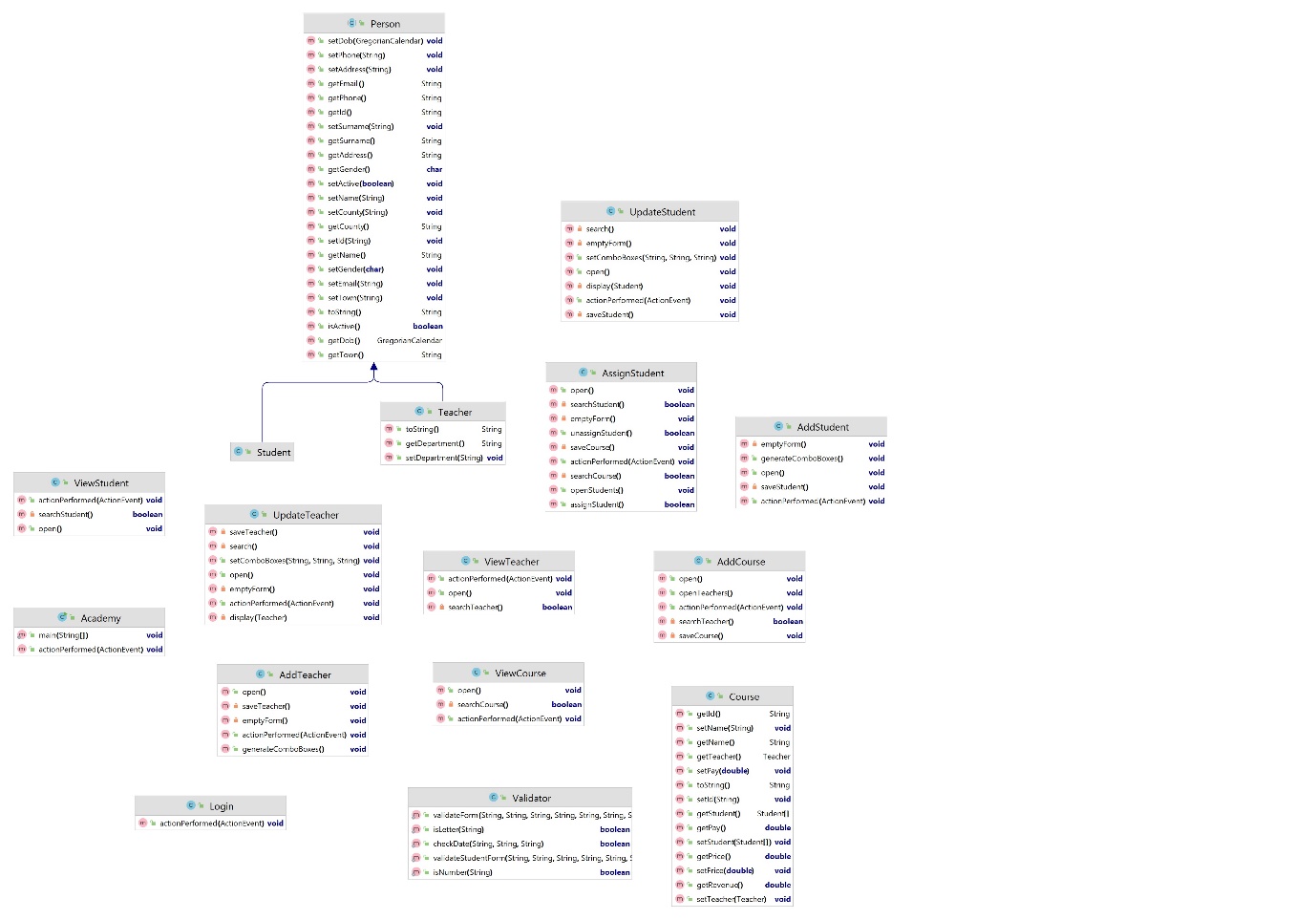


Figure 1.1 Old version diagram

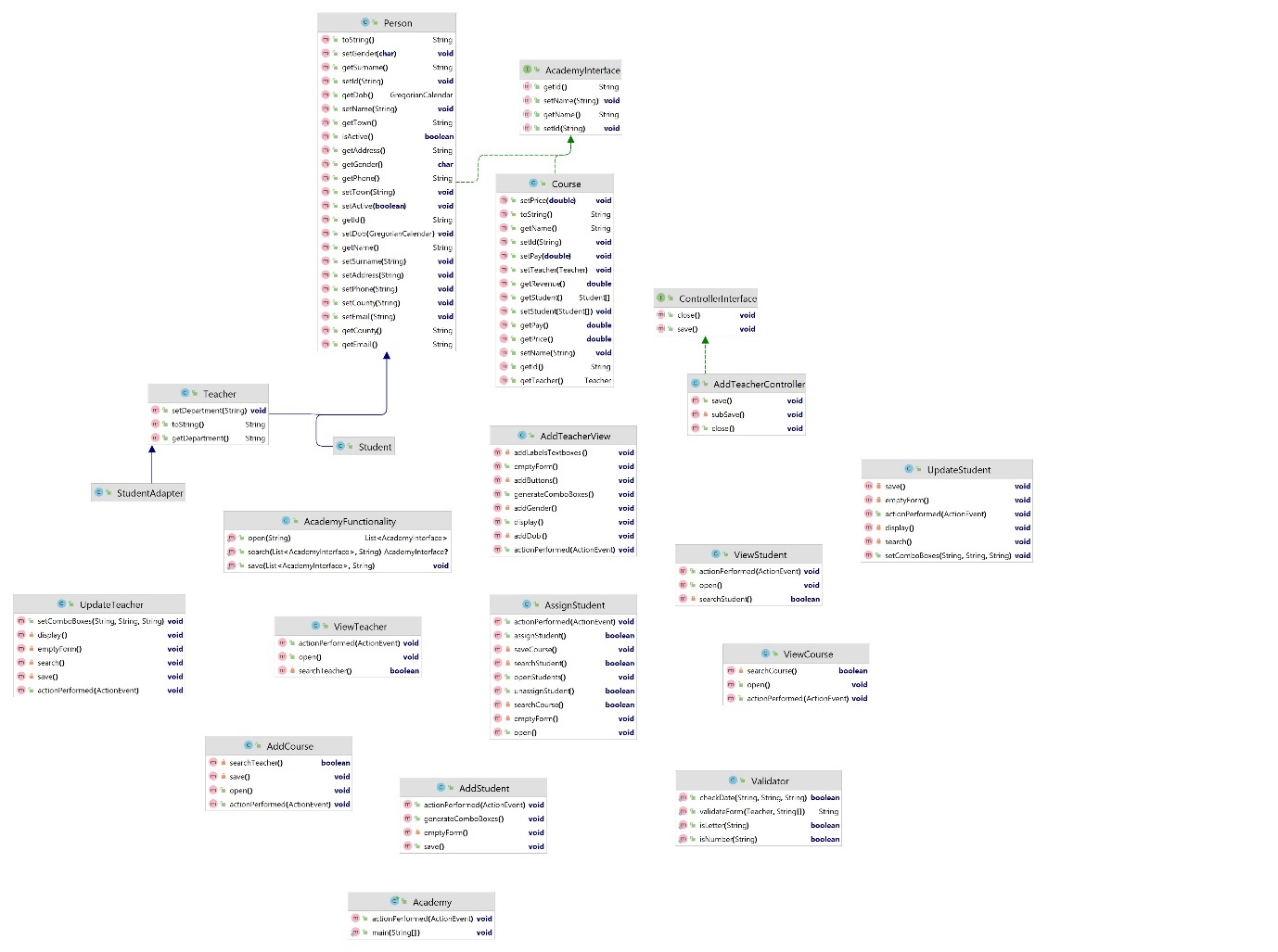


Figure 1.2 Current version diagram

# 2 GitHub repositories

The code of the original project and the current project after implementing the design patterns can be found on GitHub and the links to the repositories are:

Original project:

https://github.com/Segade/Java\_Project

Current project:

https://github.com/Segade/Java\_project\_DesignPattern

# 3 Patterns used

## 3.1 Strategy pattern

The strategy pattern is a behavioural pattern that is used for creating a family of classes. Each class is individual, but they can be interchangeable at runtime.

**A new interface class must be created. This new interface holds the common methods of all the classes to work with in order to be interchangeable. Then, the previous classes implement the new interface. As a result, the objects created from the family of classes can be used by the methods as the objects were the same class.**

**A real example of this design pattern is a credit purchase system. All the credit card provide the same information (the interface), but the type of credit card differs one from another.**

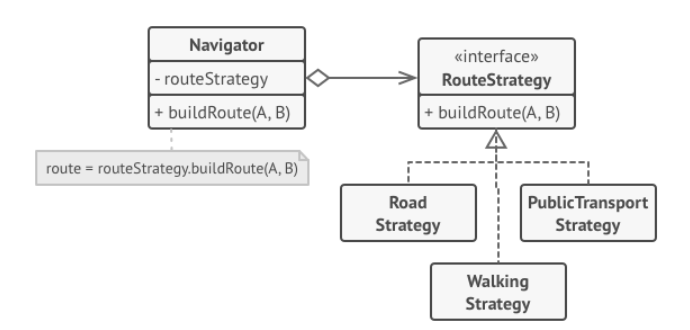


Figure 3.1 Strategy pattern diagram (Refactoring guru, 2023)

This pattern is used on this project to:

open the file where the data is and retrieve it.

Save the new data on the file.

Search one specific record on the ArrayList.

These methods are stored on a new class called "AcademyFunctionality". As this new class is not instantiated, the methods will be static.

### 3.1.1 Description of the implementation

This pattern was implemented on this project in the following way -

A new interface called AcademyInterface was created. This new interface defines 4 methods which are common among the Teacher, Student and Course classes. The methods are: getId, setId, getName and setName. Then, the abstract Person and the Course class implement the new interface. The Teacher and Student classes inherit from the Person abstract class. This fact makes that these two classes belong to the AcademyInterface interface as well as the Course class. This means that Teacher, Student and Course objects can be used by the methods open, save and search hosted on the AcademyFunctionality class.

### 3.1.2 Open method

Previously implementing the strategy pattern on the project, each section (add and update for Teacher, Student and Course) contained their own open method to open the corresponding file and retrieve the data., All these functions used exactly the same code with the only difference of the object type used. Therefore, there was a large amount of redundant code across the whole application. After implementing the strategy pattern, the code dedicated to open and retrieve the data is centralised just in one specific place for the whole project, the AcademyFunctionality class.

As a result, all the open methods across the application were removed and their code was moved to a new open method hosted on the AcademyFunctionality class.

The new open method receives a String as parameter. This parameter is the file name where the data is extracted from. The data is stored on a List variable that host AcademyInterface objects. Therefore, both Teacher, Student and Course objects can be passed to this variable. Then, the open method returns the List variable with the data to the code that calls the function. The variable that receives the data is an ArrayList that holds AcademyInterface objects to be compatible between the two objects.

// ArrayList declaration and call to the open method

List<AcademyInterface> allTeachers = new ArrayList<AcademyInterface>();

allTeachers = AcademyFunctionality.open("teachers.data");

The main changes on this method are, the new heading, add a List object that holds the data extracted from the file and return the new List object.

public static List<AcademyInterface> open(String fileName) {

List<AcademyInterface> allItems = new ArrayList<AcademyInterface>();

// code to open the file

allItems = (ArrayList<AcademyInterface>) is.readObject();

// code to open the file

return allItems;

### 3.1.3 Save method

Previously implementing the strategy pattern on the project, each section (add and update for Teacher, Student and Course) contained their own save method to save the data on the corresponding file. All these functions used exactly the same code with the only difference of the object type used. Therefore, there was a large amount of redundant code across the whole application. After implementing the strategy pattern, the code dedicated to save the data is centralised just in one specific place for the whole project, the AcademyFunctionality class.

As a result, all the save methods across the application were removed and their code was moved to a new save method hosted on the AcademyFunctionality class.

method hosted on the AcademyFunctionality class.

The new save method hosted on the AcademyFunctionality class receives two parameters: A List object and a String variable. The List object is an AcademyInterface object, therefore it can be either Teacher, Student or Course objects. The String variable is the file name where the data is stored.

The method stores an AcademyInterface object in the corresponding file, depending if it is Teacher, Student or Course data. As the strategy pattern is used, the code does not differentiate the three class, therefore it deals with them as a unique class. The code that calls the save method passes the ArrayList of AcademyInterface definition. Previous passing the ArrayList, the new corresponding object is added to the ArrayList. Therefore, the ArrayList object contains the whole with new data added. As this ArrayList was declared as a List, it can be passed to the save method. A new corresponding object is added to the ArrayList

// ArrayList declaration and call to the save method

List<AcademyInterface> allTeachers = new ArrayList<AcademyInterface>();

Teacher teacher = new Teacher(/\*Parameters\*/);

allTeachers.add(teacher); allTeachers.add(teacher);

AcademyFunctionality.save(allTeachers, "teachers.data");

The main changes of this method are, the heading and the object name saved on the corresponding file.

public static void save(List<AcademyInterface> allItems, String fileName)

// code to save the data

objectOutStream.writeObject(allItems);

// code to save the data

### 3.1.4 Search method

Previously implementing the strategy pattern on the project, each section (update for Teacher, Student and Course) contained a search method with a loop statement that searched the required object on the ArrayList with all data. This loop was common for all the update sections, with the only difference of the object type used. Therefore, there was a large amount of redundant code across the whole application. After implementing the strategy pattern, the code dedicated to go through the ArrayList the data is centralised just in one specific place for the whole project, the AcademyFunctionality class.

As a result, all the loops in charge of searching the required object across the application were removed and the code was put on a new method called search hosted on the AcademyFunctionality class.

The new search method hosted on the AcademyFunctionality class receives two parameters: A List object and a String variable. The List object is an AcademyInterface object, therefore it can be either Teacher, Student or Course objects. The String variable is the Id of the object to be found on ArrayList that contains the whole data. As the getId method is defined on the AcademyInterface, both Teacher, Student and Course classes contain this method. Therefore, this method can be used independently which class is used. Thus, the function uses the same code to search the required object.

After processing the search, the method returns an AcademyInterface object. In the case of finding the required object, the function returns the specific object. In the case of not finding the required object, the function returns a null AcademyInterface object. The code that calls the search method must cast the result in order to be able to work with the methods that are not hosted on the AcademyInterface interface and correspond to the specific class.

The main change is that the loop that searches the required object is extracted and put on the AcademyFunctionality class

of this method are: the heading and that it returns a null object in case the record was not found.

// call to the search method

List<AcademyInterface> allTeachers = new ArrayList<AcademyInterface>();

Teacher teacher = null;

String id = searchTextField.getText();

teacher= (Teacher) AcademyFunctionality.search(allTeachers, id);

public static AcademyInterface search(List<AcademyInterface> allItems, String id){

// code for searching

if (x == allItems.size())

return null ;

return item;

## 3.2 Justification

Before using this pattern, each form had its own method for these operations. By using the Strategy pattern, the program just has one single method for each operation on all the project. This change reduces the number of methods in the whole application and the functions are localised in an individual class. The pattern makes the program more readable and easier to maintain and in the case of changing the storage technology only one class is affected. Moreover, the Single Responsibility principle is applied.

The Single responsibility is one of the SOLID object-oriented programming principals. These principals are a fundamental group of concepts that insurance that object-oriented programming is used correctly. The single responsibility refers to that each component (class) has a unique functionality. This principal makes components (classes) and methods easy to understand, and in the case of making any change, only one component is affected (Martin, 2008).

Sometimes the strategy pattern and the state pattern are confused. In the state pattern the state of the object alters the perform of the object. Also, the object may have different states during its life cycle along with the performance. In this case, neither the Teacher nor the student and nor the Course suffer any change on their state nor different performance during their life cycle is required. For this reason, this pattern was discarded from the project (Kovko, 2023).

## 3.3 Drawbacks

Some of the drawbacks of this strategy pattern are:

This pattern adds new class to the program. Thus, clarity of the project may be affected. Also, complexity may increase.

This pattern increases the number of classes. Thus, it may slow down the throughput of the program as more memory usage is needed. This is a big issue especially for larger projects.

An incompatible strategy applied is not detected by the compilation as an error. So, to find this mistake may take too much time and will generate a runtime error.

In the case of using too many strategies, code duplication may appear. This decreases the quality of the code. In the case of needing to change the pattern developers would have to make the change in all the parts of the code that are affected.

If the interface suffers a modification, it may affect the whole program. As a result, there would be too many modifications, with the danger that this fact is presumed. Therefore, a deep knowledge of the project by the developers is required.

## 3.2. Adapter pattern

The adapter pattern is a structural pattern that allows that incompatible classes can work together without modifying the source code. A new class is created called wrapper, and performs as mediator between the other classes. The adapter class implements the expected class called target. The class to be adapted, called adaptee, is passed as parameter on the constructor. Then, the adapter overrides the target class methods with the adaptee corresponding methods within. A real example of this pattern is on the java Arrays.AsList method. This converts a regular array into a List object (tutorialkart, 2023)

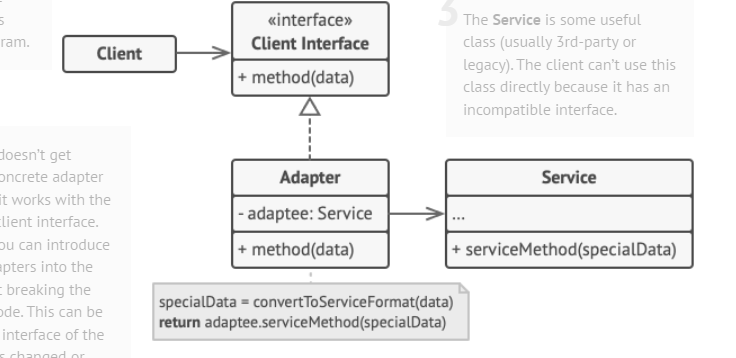


Figure 3.2 Adapter pattern diagram (Refactoring guru, 2023)

This pattern is used on this project to:

Pass a Teacher class to the validateForm method within the Validator class.

The validateForm method is used to validate either teacher or student's form.

### 3.2.1 Description of the implementation

For this purpose, a new class called StudentAdapter is created. This new class extends from the Teacher class (The target class), so it will be treated the same as the Teacher class.

The StudentAdapter constructor receives a Student class (the adaptee) as a parameter. Then the super class constructor (Teacher class) is used to set all the inherited attributes from the Teacher class. To enter the student object values to the super constructor, the getters methods that belong to the Student class are used. The Teacher and Student classes share all the attributes, but the department attribute. Therefore, a “student” string value is entered on the super constructor in order to pass the validation. As a result, the StudentAdapter class contains all a Teacher attributes and methods, since it extends from the Teacher class. However, as the constructor receives the values from a student object, the StudentAdapter object contains the student values but uses the Teacher methods to work with them, because the StudentAdapter class is considered the same family of the Teacher class.

public class StudentAdapter extends Teacher{

public StudentAdapter(Student student){

super(student.getId(), student.getName(), student.getSurname(), student.getAddress(), student.getTown(), student.getCounty(), student.getDob(), student.getEmail(), student.getPhone(),student.getGender(), "student");

} // end constructor

} // end of class

The main change on this method is the heading. With the new heading most of the attributes are replaced with the Teacher class that contains the values to be validated.

public static String validateForm(Teacher person, String[] date){

### 3.2.2 Justification

The benefit of using this pattern is that the number of parameters passed to the validateForm method decreases by 7 values. With this, the project removes the code smell called "Long Parameter List".

Previously, the heading of the method contained up to 11 parameters.

public static String validateForm(String name, String surname, String address, String town, String county, String myDay, String myMonth, String myYear, String phone, String email, String department){

Thanks to the adapter pattern, up to 8 parameters can be encapsulated in one single variable (in a Teacher object). In addition, the variables related to the data of birth were stored in an array, therefore the method heading just contains 2 parameters.

public staticString validateForm(Teacher person, String[] date){

The adapter pattern is created to connect two classes, where one class masks the second one and then both perform as similar. The goal was to convert the Student class into a Teacher class, in order to be able to pass both classes to the ValidationForm method as a similar variable. The reason is that the Student class needs to add the department property, which is used on the validation method. Therefore, to mask the Student class as a Teacher, the adapter class needs to extend the Teacher class and overrides its methods with the Student methods within. As a result, the methods are originally from the Teacher class, but their code is substituted by the Student class.

In the decorator pattern, the classes join their code in order to perform together and extend new functionality to the solution without altering the interfaces. The decorator pattern uses recursive composition, something not applicable to the adapter pattern. As the purpose is not to add new functionality to the solution, if not just to substitute the Student class by the Teacher class and keep the classes independent from each other. For this reason, this pattern was discarded for this project (refactoring, 2023).

The bridge pattern is usually confused with the adapter pattern because the bridge pattern wraps the class to substitute. However, the bridge pattern is created to be able to modify the wrapped class during runtime to meet the need in the specific moment. In this case, the Student class is not modified in any moment. For this reason, this pattern was discarded from this project (doeken, 2023).

### 3.2.3 Drawbacks

Some of the drawbacks of this adapter pattern are:

This pattern adds a new class to the project. So, the pattern makes the project more complex and less clear by incrementing the overall number of classes and the run time is increased by this change.

If there is any change in the interfaces, the adapter class will have to be modified too. Therefore, the number of places to be modified on the project is affected as well, this means tightly coupling.

If multiple adapter classes are created, may lead to code duplication and poor maintainability. In the case of needing a change the pattern developers would have to make the change in all the parts of the code that are affected.

Converting one class to another may be resource-intensive. This makes the program less efficient. This is a big issue especially for larger projects.

## 3.3 Model View Controller (MVC)

The MVC (Model View Controller) design pattern is a software architectural pattern that splits out the application into three interconnected layers (Data model, presentation and control information). With this, applications achieve higher reusability, maintainability, lose coupling and modularity. This pattern is widely used in web Frameworks i.e. Angular, React, Vue.. Even though these web frameworks implement variations of this pattern. An example that implements pure MVC pattern is ASP.NET (Microsoft, 2023) . On ASP.NET the view layer uses Razor syntax to generate HTML code. The controller layer handles the requests and the model layer handles the bind the web page to the data storage system along with the validation rules applied in the application.

* Model layer: It handles the data used on the application and the business logic.
* View layer: It handles the user interface and data representation.

It manages user interactions. Also, it shows data sent from the model layer.

-Controller layer: It handles the flow of data between the view layer and the model layer. It receives the user input, then it gets the corresponding data from the model layer and modify the view according to the required data (Leff & Rayfield, 2001).

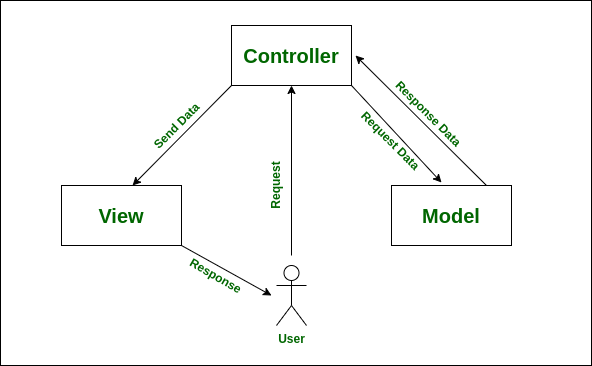


Figure 3.3 MVC pattern diagram (geeksforgeeks, 2023):

This pattern is used on this project on:

Add Teacher section. Originally all the code related to this section was hosted in one single class AddTeacher.java. In order to implement the MVC design pattern, a new class called AddTeacherController was created and the AddTeacher class was renamed to AddTeacherView for clarification.

### 3.3.1 Description of the implementation

The code that is trigger when the user presses the Add button and performs the save process for collecting the values from the textboxes, comboboxes and radio buttons and also handles the validation is moved to a new method called save on the AddTeacherController class. The save method that is hosted on the AddTeacherView class is moved to the AddTeacherController class. As there is already a save method and this method calls the previous save method, this method is renamed with subSave. The subSave method is the method that calls the AcademyFunctionality save method which stores the data.

The line of code that is triggered when the user presses the Cancel button and closes the window and is hosted on the AddTeacherView class is moved to the AddTeacherController class. This line is put on a method called close. The reason of this is, although there is only one line of code, the controller class take over the process of closing the window. In the case of additional code is added in the future for this operation, the AddTeacherController class must contain the corresponding code.

The AddTeacherView class declares a ControllerInterface object. The reason of this is that the AddTeacherView class must call the methods that handle users’ actions and that substitute the code moved to the AddTeacherController. The controller is passed as a parameter on the AddTeacherView constructor. The reason of this is if the view is required to be called from another part of the project, the AddTeacherView does not what class makes the call. Therefore, the application increases loosely coupling in the solution.

In addition, the declaration of the List object as an ArrayList that stores the data retrieved from the open method, is moved from the AddTeacherView class to the AddTeacherController class. The reason of this is that the open method (on AcademyFunctionality) is also moved to the controller in order to maintain the logical. The call to the open method is put on the AddTeacherController constructor to perform the operation from the instantiation of the controller.

List<AcademyInterface> allTeachers = new ArrayList<AcademyInterface>();

Moreover, a new AddTeacherView object is declared on the AddTeacherController. The reason of this is that the controller class must handle the view layer of the application. The instantiation of the AddTeacherView class is put on the AddTeacherController constructor in order to show the window once the controller is instantiated as well. Furthermore, the lines of code on the AddTeacherView class that show the window are moved to a new method called display. This new method remains on the AddTeacherView. The reason of this is that the controller takes over the decision of when the window must be shown. The call to this method is done on the AddTeacherController constructor.

AddTeacherView view;

public AddTeacherController(){

view = new AddTeacherView(this);

view.display();

allTeachers = AcademyFunctionality.open("teachers.data");

} // end constructor

A new interface class called ControllerInterface was created in order to keep the same structure for all the controller classes. The methods defined on this class are save and close. The reason of this is that the MVC pattern meant to be implemented on the whole project. Therefore, an interface must be created in order to insure consistency across the application. For this purpose, the AddTeacherController implements this interface.

public class AddTeacherController implements ControllerInterface{

On this design pattern the AcademyFunctionality class is the model layer. This class handles all the processes to open and save the data into the storage system. On this project the storage system are just simple files for each Teacher, Student and Course data. In the case of requiring to change this system, for example, to a database, the AcademyFunctionality class will handle the modification and the other layers do not suffer the change.

### 3.3.2 Justification

With the MVC design pattern the application achieves a clearer separation among the responsibilities of the application. Each layer does not interfere with the others, therefore, any change made on one layer does not affect the other two, which means more loosely coupling in the project. With this separation, the code is more understandable, testable, maintainable and reusable.

In the case of making a change on the view layer, the modem and controller layers must not be affected by that change. Therefore, there is less probability of getting the project crash because the change moved something that it should not, and he same with the other layers.

Also, the model layer (the AcademyFunctionality class) that deals with the storage system is common for the whole project. Thanks to the MVC design pattern, this part of the application is separated to the rest of the code. Because of the lack of time, this pattern was not implemented on the whole project. In the case of implementing the MVC design patter across the entire application, all the controllers classes would call the AcademyFunctionality class for the storage processes. This reduce the need of duplication of classes.

Furthermore, with the MVC design pattern the same view class may be called from different controllers. That is not the case on this project, but if this would be needed in the future, there would not be need of creating new similar windows.

Another similar design pattern that might be used in this project is Model-View-View model pattern (MVVM). The reason because this pattern was not used is that the MVVM pattern does not centralise the functionality on the controller, when on the MVVm the view centralises the functionality. Therefore, in the case of needing to use the same controller class to utilise another view class, it would be more complex. This would be useful if there is a new teacher section with a new window, but the same controller may be used. On the MVVM the view layer is responsible of makes the corresponding changes on the form, rather than the controller class. Therefore, with MVVM reusability is not so guaranteed in this project (Krstanović, 2023).

The event-driven pattern is designed to handle queues of asynchronous processes within a system, where the execution of one event depends on the execution of another one and events do not have direct connection between them. In this project, only users’ events are handled and there are no additional events to process or that depend on users events to wait for. For this reason, this design pattern was discarded from this project (Seetharamugn, 2023).

The microservice pattern is designed to divide the whole application into smaller pieces. Each piece has one unique purpose and can work separately to the other pieces of services. Then all the pieces are implemented together for the entire application. In this project, the pieces are the layers (mode, view and controller) and even though each one has a different purpose, both are part of the same section and cannot work one without the other ones. For this reason, this design pattern was discarded from this project (Kong, 2023).

One benefit is that the application gets more scalable, since the workload is distributed among the three layers. Therefore, the resources are optimised.

With this pattern applications get more flexible. Even though it is used on this project, each layer may be made on different technologies i.e. Django for the model, React for the view, and Flask for the controller.

### 3.3.3 Drawbacks

The MVC design pattern requires three layers (classes or file) for each section of the project. This increments complexity, functions and overhead of the program. This factor may make the program harder to debug and refactor. This is a big issue especially for larger projects.

In the case of applications where the layers have a big dependency (tight coupling), high modularity decreases code quality, since managing a tight coupling application with several classes is inefficient. Therefore, the use of the MVC design pattern cannot be used on every application with different layers, since each layer must be well separated from each other.

# 4. Testing

On this section there are the results of testing the original application and the new version. After running the testing, the figures proved that the implementation of the three design patterns improved the quality of the application. Just to point out the decrease of the figures on three sections where the reduction was higher: bugs from 34 to 18, Security hot spots from 39 to 23 and duplication from 34.40% to 22.50%.

\*Figure 4.1 is the original version: figure 4.2 original overview with figure 4.3 and Figure 4.4 outlining the new version

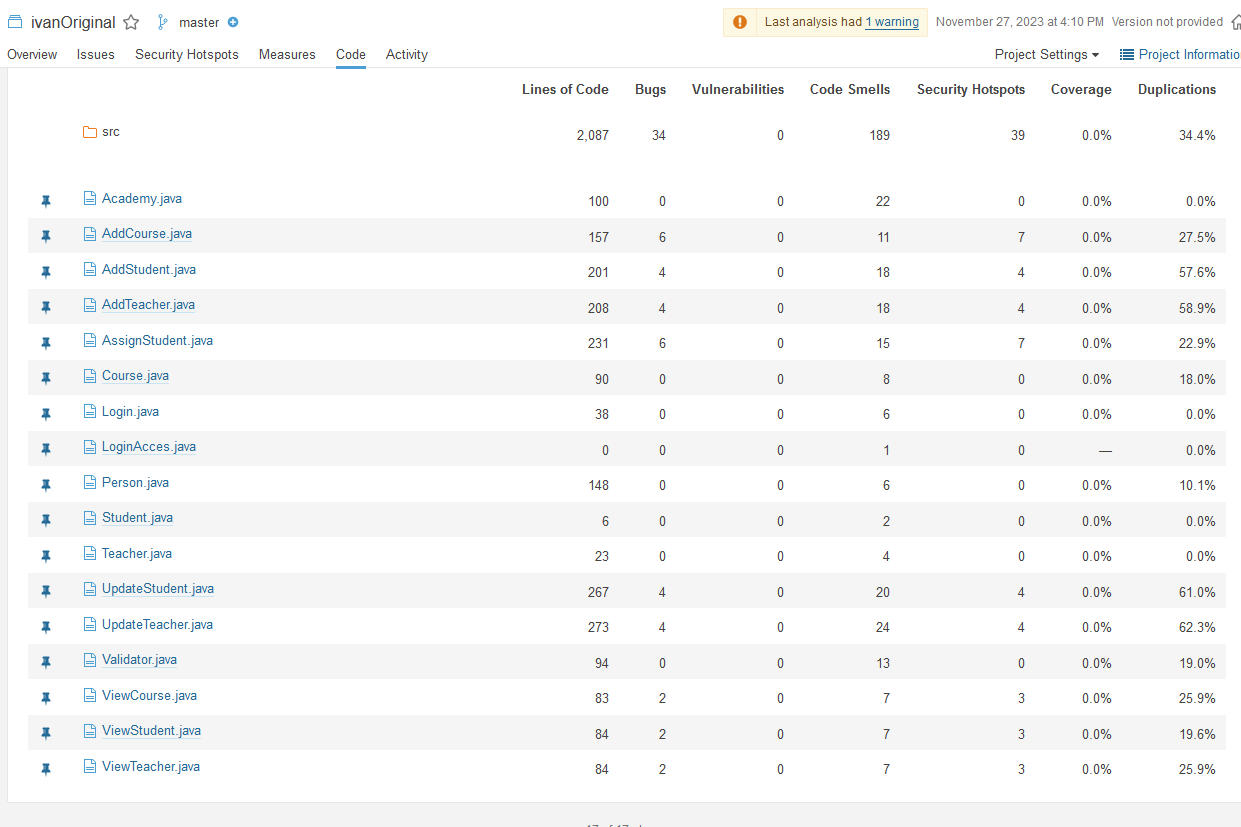


Figure 4.1 Old version bugs and code smells

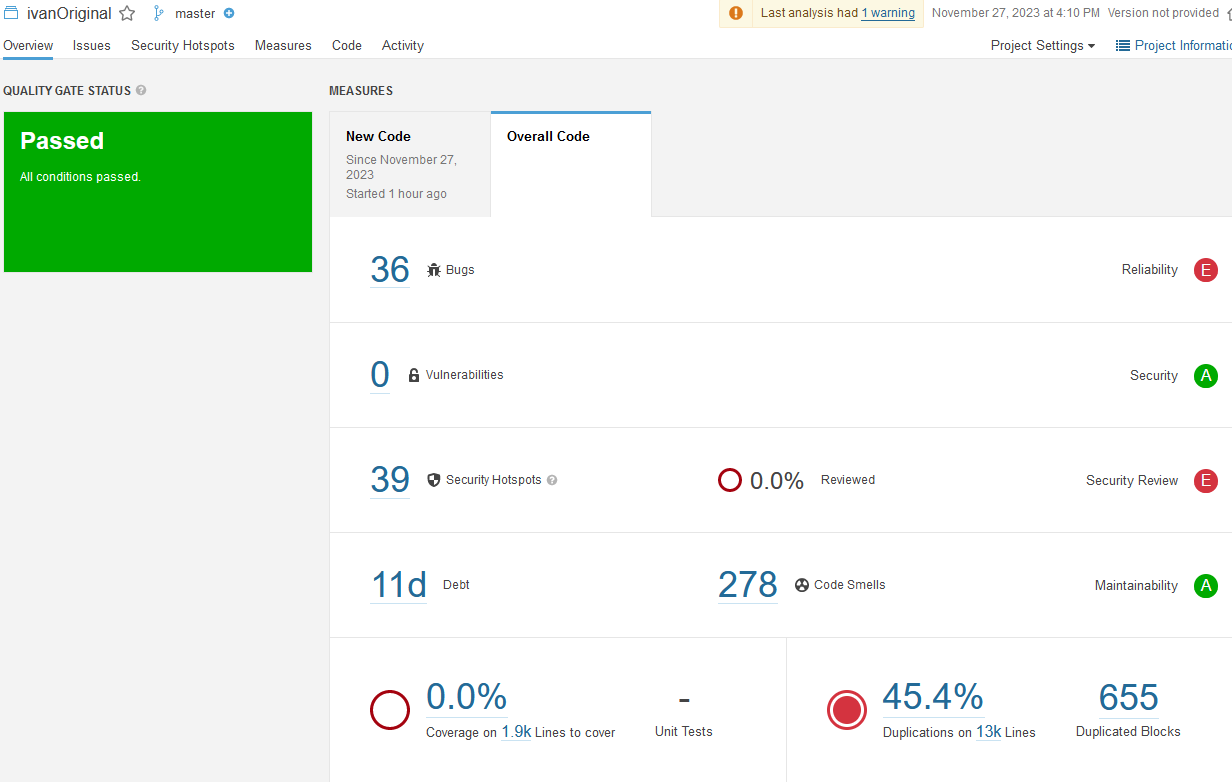


Figure 4.2 Original version overview

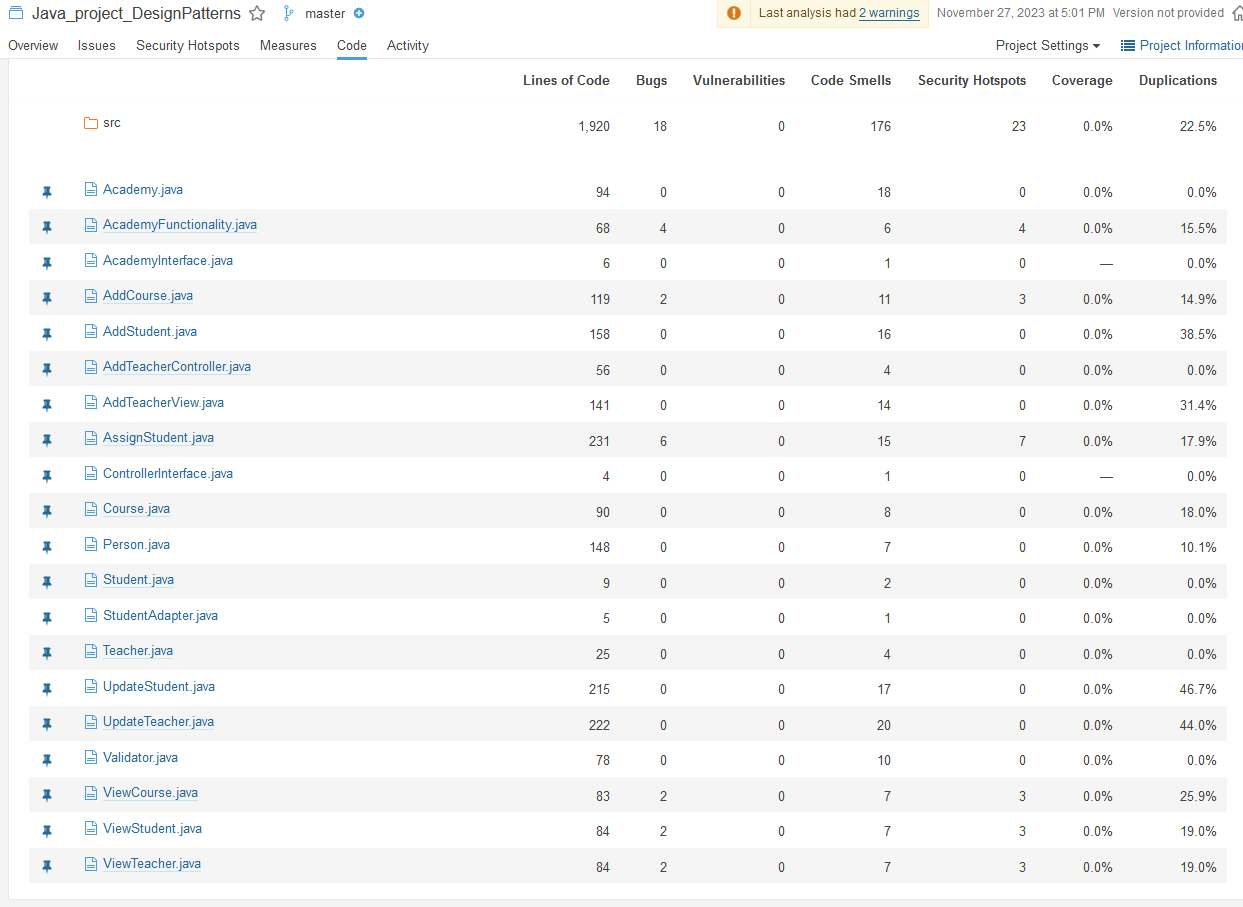


Figure 4.3 Current version bugs and code smells

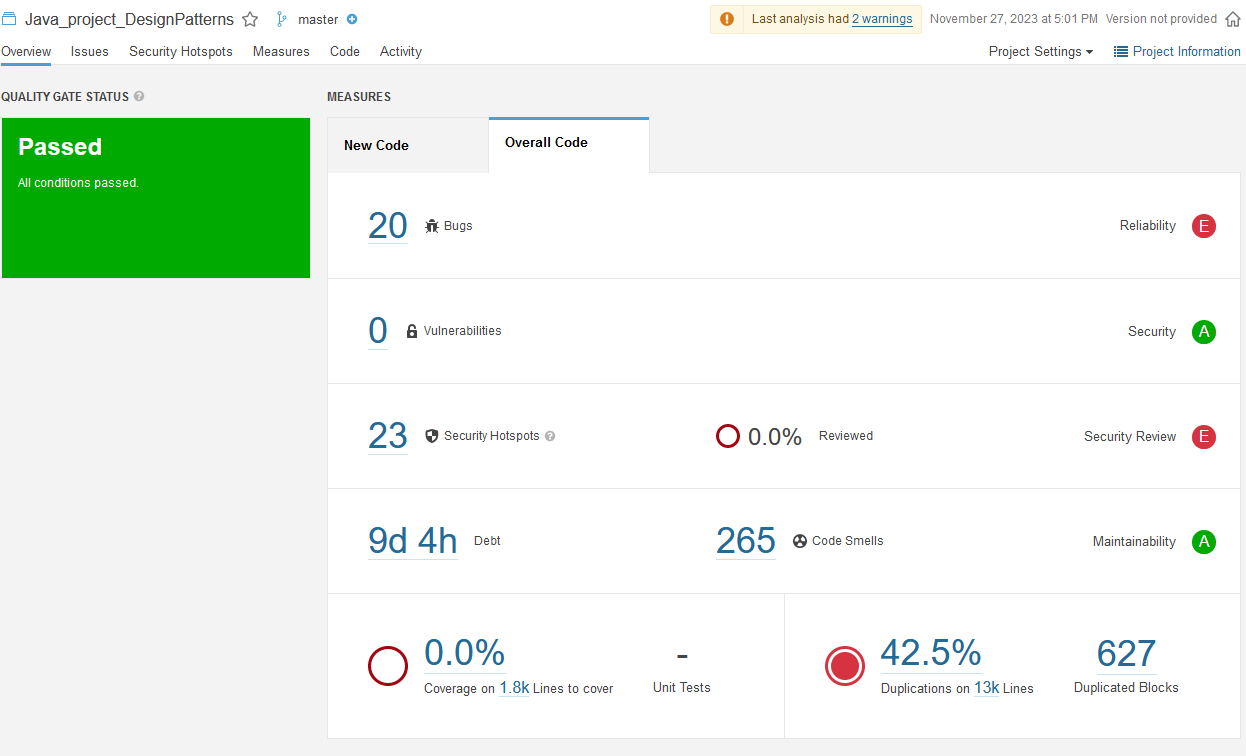


Figure 4.4 Current version overview

# 5. Conclusion

The design patterns used for this project increased its quality, since the application reduced a large number of redundant lines of code and removed some code smells. Most of the redundant code was moved to new classes. This increased the complexity of the application since finding a specific piece of code gets harder. However, the application, thanks to the design patterns keeps it’s logic. Therefore, the developer can figure out where the piece of code may be. The number of classes was increased too, however this change improved clarity of the code also.

Perhaps the strategy pattern was used to an extreme point. Because the application treats both teachers, students and courses the same, when they are actually very different parts of the project. Therefore, this use of the strategy pattern might not be the most elegant application of it, since each section of the program must be differentiated for clarifications reasons. By dealing with the three sections as one this can lead to confusion. Casting must be applied when the data has to be manipulated in order to access the non-common methods. This increases complexity in the application.

For this project the fact that the three classes used utilise the same interface may give some problems. On this project the storage system used is a simple file system. Therefore, there is no complexity on handling the data. In the case of using databases rather than file system for storing the data, some problems may arise. Databases have a higher degree of complexity because databases require a more structured data setup and are more restrictive. In this project, data is saved and retrieved regardless of which class is applicable. In databases this process cannot be implemented, as data has to be well defined.

The use of the adapter pattern removed one code smell, which improved the quality of the code. Perhaps, this pattern could have been avoided if the original design was better. The original idea was to have on the Person class just the common attributes and methods for teachers and students. That was the reason of putting the department attribute along with the getters and setters on the Teacher class. By moving the department attributes and its getters and setters to the Person class, the use of the adapter pattern could have been discarded. To pass the validation (the initial idea of application of the adapter pattern), a default value could be set. Another question might be if it would be suitable to put the department attribute on the Person class when not all the inherited classes use it.

The use of the MVC pattern improved the clarity of the application and is more decoupled. Now it is obvious in which class each process is handled. Actually, this pattern could add a new layer to the project. For example, in the add teacher and update teacher sections share exactly the same form to save and display teacher’s information. Although the update window adds the search section, it uses the same form for the rest of the process. Therefore, a new layer could be added to the project that holds the common form and the windows can call the corresponding form when necessary. This would add even more complexity to the program, but it would add more reusability too.

(Freeman, Robson, Sierra, & Bates, 2021) provides a good guidance on the topic of design patterns, giving good explanations of each design pattern.

This assessment proved that make changes to an existing application is complex, because there is the fear of changing a part of the code, which must be modified, but another part of the application may be affected and programmers must be aware of that. Therefore, sometimes the solution to fix the code made may not be the most elegant i.e. the implementation of the adapter pattern. Perhaps, adding the department attribute to the Person class would be more professional, but when refactoring the code the adapter pattern was the best solution. Move the department attribute to the Person class would avoid the creation of the adapter class, and as a result there would be one less class on the application, which means less complexity. Therefore, not always design patterns are the solution. Developers must be aware that not always design patterns have to be implemented (Wendorff, 2002).

This assessment proves the point that developers must think deeply how to make the program before coding. Developers must think that programs may suffer changes in the future, therefore the application must be designed for accepting future modifications. Also, code must be clear and understandable, since this application was made two years ago, and there were some parts that were hard to understand, and the person that made the program originally and the person that did this assessment are the same.

Moreover, refactoring this project, just partially, took a large period of time. In the real world, this period of time may not be acceptable on a real company, since it had been a waste of money. Therefore, developers must have a good think before coding in order to avoid catastrophic consequences in the future.

# 6. References

doeken. (2023, 11 14). *Adapter Pattern vs. Bridge Pattern | doeken.org.* Retrieved from doeken.org: https://doeken.org/blog/adapter-vs-bridge-pattern

Freeman, E., Robson, E., Sierra, K., & Bates, B. (2021). *Head First Design Patterns.* Sebastopol: O'reilly .

geeksforgeeks. (2023, 11 23). *MVC Design Pattern - GeeksforGeeks*. Retrieved from geeksforgeeks.org: https://www.geeksforgeeks.org/mvc-design-pattern/

Kong. (2023, 11 30). *Microservice Architecture pattern.* Retrieved from microservices.io: https://microservices.io/patterns/microservices.html

Kovko, E. (2023, 11 14). *State Pattern vs. Strategy Pattern | Baeldung on Computer Science.* Retrieved from baeldung.com: https://www.baeldung.com/cs/design-state-pattern-vs-strategy-pattern#:~:text=Strategy%20vs.,of%20the%20elements%20in%20action.

Krstanović, M. (2023, 11 20). *Understanding the Differences: MVC vs. MVVM – BrightMarbles.* Retrieved from brightmarbles.io: https://brightmarbles.io/blog/differences-between-mvc-and-mvvm/#:~:text=In%20MVC%2C%20the%20View%20communicates,the%20ViewModel%20through%20data%20binding.

Leff, A., & Rayfield, J. (2001). Web-application development using the Model/View/Controller design pattern. *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference* (pp. 1 - 20). Seattle, WA, USA: IEEE.

Martin, R. C. (2008). Chapter 13: Concurrency. In *Clean code* (p. 181 ). New Jersey: Pearson education Inc.

Microsoft. (2023, 11 22). *ASP.NET MVC Pattern | .NET*. Retrieved from dotnet.microsoft.com: https://dotnet.microsoft.com/en-us/apps/aspnet/mvc

refactoring. (2023, 11 15). *Adapter*. Retrieved from refactoring.guru: https://refactoring.guru/design-patterns/adapter#:~:text=Adapter%20provides%20a%20completely%20different,possible%20when%20you%20use%20Adapter.

Refactoring guru. (2023, 10 14). *Adapter*. Retrieved from refactoring.guru: https://refactoring.guru/design-patterns/adapter

Refactoring guru. (2023, 10 14). *Strategy*. Retrieved from refactoring.guru: https://refactoring.guru/design-patterns/strategy

Seetharamugn. (2023, 11 30). *The Complete Guide to Event-Driven Architecture | by Seetharamugn | Medium.* Retrieved from medium.com: https://medium.com/@seetharamugn/the-complete-guide-to-event-driven-architecture-b25226594227#:~:text=Event%2Ddriven%20architecture%20is%20a%20method%20of%20developing%20enterprise%20IT,events%20occur%20throughout%20the%20business.

tutorialkart. (2023, 11 22). *Java Arrays asList() - Examples*. Retrieved from tutorialkart.com: https://www.tutorialkart.com/java/java-util-arrays-aslist/#gsc.tab=0

Wendorff, P. (2002). Assessment of design patterns during software reengineering: lessons learned from a large commercial project. *Proceedings Fifth European Conference on Software Maintenance and Reengineering.* Lisbon: IEEE.