

COURSE UNIT MANAGEMENT SYSTEM (WHITEBOARD)

DESCRIPTION

The “Whiteboard” application allows teaching faculty staff members (i.e. lecturers) to add in course units that they are interested in supervising. These course units can be divided into three subtypes; taught, research and internship. The taught course unit represents a typical class session running within the span of a particular semester in which the lecturer can make note of. The research course unit consists of duration and supervisor attributes. This means that it does not necessarily need to run within the span of a semester. The duration is determined in terms of years. Additionally, a supervisor can be chosen to manage this course unit should the lecturer feel the need to or if he/she does not want to directly manage it. Lastly, the internship course unit allows students to be allocated into an organization outside of the school for a particular length in time. Therefore, the lecturer has to define the organization (name) that will handle this course unit for a specified duration (also, in terms of years). Additionally, each course unit is accompanied by a limit. This limit represents the maximum number of students that can attend this course unit (i.e. seats available).

The data will then be saved into a choice of two different types of (local) file store. The first is a binary-formatted/machine-readable file. The other is a CSV-formatted file. The user/lecturer can select the file types that he/she would like to save in through the menu bar on the application’s graphical user interface. The application might be able to accommodate for more file types in the near future.

The functionality of the application is as follows:-

1. *Add New Course Unit* – The user can add/create as many course units as he/she wants. Each course unit can be different as well. It is also good to note that adding a new course unit (to the UI) does not mean that the data has been saved into a file store yet. Clicking on the *add* button merely updates the GUI and allows the user to enter the details of this new course unit. Saving is done through the *submit* button
2. *Delete Course Unit* – The user can remove a particular course unit on the interface/screen by clicking on the *bin* button. Note that there is no undo option once this operation completes.
3. *Delete ALL Course Units* – At the top of the screen and next to the *add* button, the user can choose to delete all of the course units shown on the interface/screen. Note that there is no undo option once this operation completes.
4. *Submit/Save* – Finally, once the user is satisfied with the new/updated course units, he/she can opt to save them into a selected file store format (as mentioned previously). This means that the data can be re-loaded at a later time for further changes or to accommodate for additional course units.

Regarding the details of each course unit, the only compulsory fields that the user has to fill in are the course unit’s code and name. The rest can be left as blank depending on the user’s preference. A validation mechanism has been implemented to ensure that this rule is followed.

DESIGN PATTERNS USED

Singleton Simple Factory

The singleton pattern means that the Factory class (CourseUnitCreationFactory.java) will only be instantiated once throughout the lifecycle of the application. The reason for using this pattern is because there is no need to have multiple instances of a class that only has one job during the application's run-time and also due to the fact that this is a very simple and small-scaled application. Furthermore, it can be extended to allow for multiple instances in the near future without affecting the class's clients. The Factory class will also be accessible by any other class. However, at the moment, only the Controller and main class is accessing it.

The simple Factory pattern is used when creating new course unit objects. Since there are three different types of this object, the Factory will decide on the right constructor to call depending on the parameters passed into its method from the Controller class. The Factory class is only assigned with the task of creating course unit instances according to the course unit type that has been selected through the user interface by the user. Each type corresponds to a course unit instance. For example, if the user selected TAUGHT as the course unit type, the Factory class will then react by instantiating the TaughtCourseUnit.java class before saving the new object in to the file store. The reason for using the Factory pattern was to avoid instantiating objects at various implementation classes. This reduces duplication of code significantly.

Java UI with a Controller

The application is accompanied by Java's Swing/AWT libraries. Therefore, a graphical user interface is implemented. There is only ever one screen/interface within the application. Of course, this could be extended in the near future. For now, every possible operation is and can be handled within a single interface.

The Controller (CourseUnitCreationController.java) class represents part of the GUI's event handling operation. It implements the event handling logic for the *submit* button. The rest of the event handling is done in the GUI class such as *add* and *delete* operations. This is due to the fact that the *submit* operation has a connection to the model (course unit) class. To avoid a direct interaction between the view and model class, the Controller class has to be there to implement the logic.

A Model-View-Controller pattern is adopted. The Controller class will handle the communication between the View/GUI and Model/Objects classes. It is mainly triggered when the application starts up and there is a need to load data from the file store. The data, if it exists, will then be passed to the Controller class (from Whiteboard.java) which, in turn, tells the UI class (MainFrame.java) to populate its fields accordingly. Also, the Controller handles the *submit* button's action listener. This event is triggered when the user decides to save the new/updated course units into the file store. The Controller will then receive the values from the UI and call the Factory class to construct the proper objects/instances. Once the instances have been created, they are then placed into a List before being submitted to the appropriate strategy class for saving/writing to file.

The Controller class is also partly responsible for delegating error messages to the view. For example, when the course unit code/name is empty/null, the Controller will catch the Null

Pointer Exception and call the appropriate method in the view/UI class to trigger the appropriate pop up message. Run Time Exceptions are also handled the same way.

Furthermore, the UI/view class has no knowledge of the model class. Any changes to the UI code would not affect the back-end implementation logic.

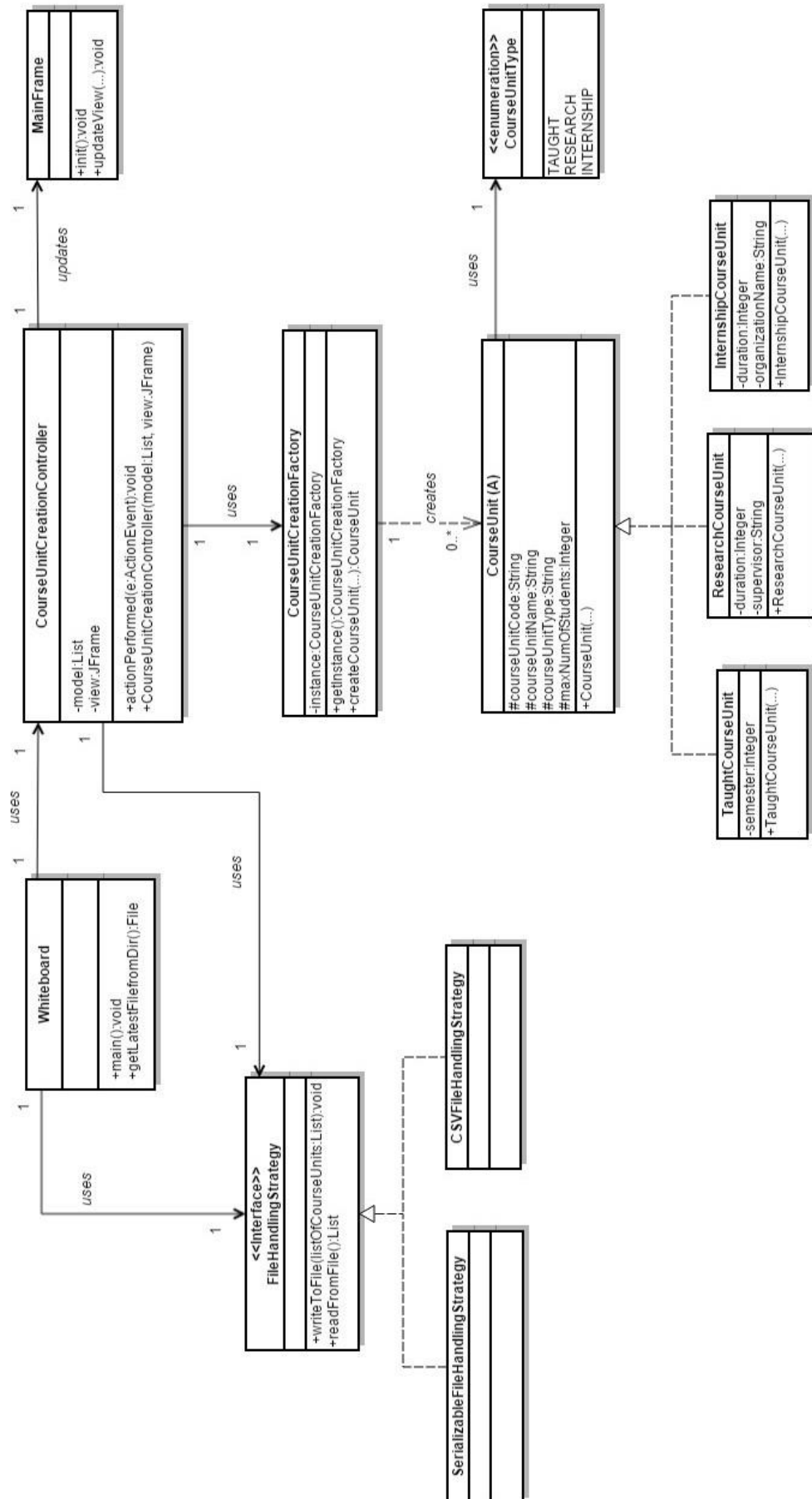
Strategy

This pattern is mainly used when saving and/or loading of data is involved. The idea behind using this pattern was to allow for a dynamic (run-time) switch of implementation code/algorithm according to the user's interaction with the application. With a given (user) option, there is no way of knowing which implementation code to use during compile time. That is why the strategy pattern was adopted.

The user is allowed to select the file type that the data should be stored into and so, the application has to select the proper file saving strategy from there. At this moment, the only two options are saving the file in serializable format and CSV format. The option is located on the menu bar of the application. Once the option is chosen and the *submit* button is clicked on, the application will then have to dynamically initialize the correct strategy class. For example, if the user opt to save the data in CSV format, the application will call the constructor of the CSVFileHandlingStrategy.java class and from there, trigger the correct method call (depending on whether the operation is a write to or read from file).

When starting up the application, data is loaded from the latest modified file store since there is no option for the user (right now) to choose which file store to load the data from. The way that this is handled right now is by allowing the decision to be dynamically handled by the application itself. The application compares both file store formats (if they exist) according to their latest modification date/time and from there, load the data accordingly. This way, the user will always be able to get the most up-to-date set of data. Additionally, if one file gets deleted, the application will just read from the other provided that the other file store still exists. Further features such as allowing the user to choose which file store to load the data from might be implemented in the near future.




With this pattern, not only does the code logic become more cohesive and separated from one another, it is also easier to add in new file format handling strategies in the near future. Adding in a new strategy would only involve creating a new and separate Java class without touching any of the existing strategy classes.



Course Unit Management System

File

Add/Remove Course Units


Course Unit Code:

Course Unit Name:

Course Unit Type: ▼

Semester:

Maximum No. of Students:



Course Unit Code:


Course Unit Name:

Course Unit Type: ▼

Duration (year):

Supervisor:

Maximum No. of Students:




File


Save File As... ▶

☐ Serializable file


☒ CSV file



DONE!

 [SUCCESS] Course Units have been saved to file!




Confirmation

 Are you sure that you want to remove ALL course units?

Confirmation

 Are you sure that you want to remove this course unit?

