Final Project Summary

By: Timothy Flucker

# Overview

For this assignment, I used the template project which was to create a school department with faculty, students, courses, programs, academic concentrations, and degrees. This project was built using Gradle for dependency management, but otherwise uses basic Java functionality to achieve the requirements outlined in the assignment prompt.

The main class to run for this project is called “App.java” and is located in the “lib/src/main/java/cs665\_final\_project” package. This class can be run as a Java application and will create a base data set to work with, and then the following functionality, which print descriptive text as console output:

* Enroll students into courses
* Have some student’s drop courses
* Show how instructors and students are notified after a student drops and a waitlisted student is added to a course
* Additional functionality

All data for students and faculty randomly generated and so no two runs of the application will contain the same exact data. The number of these objects however will remain consistent. All other objects (departments, concentrations, programs, courses) are hard-coded and will be similar between executions.

**Apologies for any messiness with the console output. There is a lot of functionality and I didn’t have a lot of time to organize and optimize everything. Thank you!**

The following screenshot shows a basic class diagram for this application, focusing on the relationships between these objects and any hierarchical relationships they have in addition to any multiplicity. The original PNG files will be provided in the zip file containing all of the code.

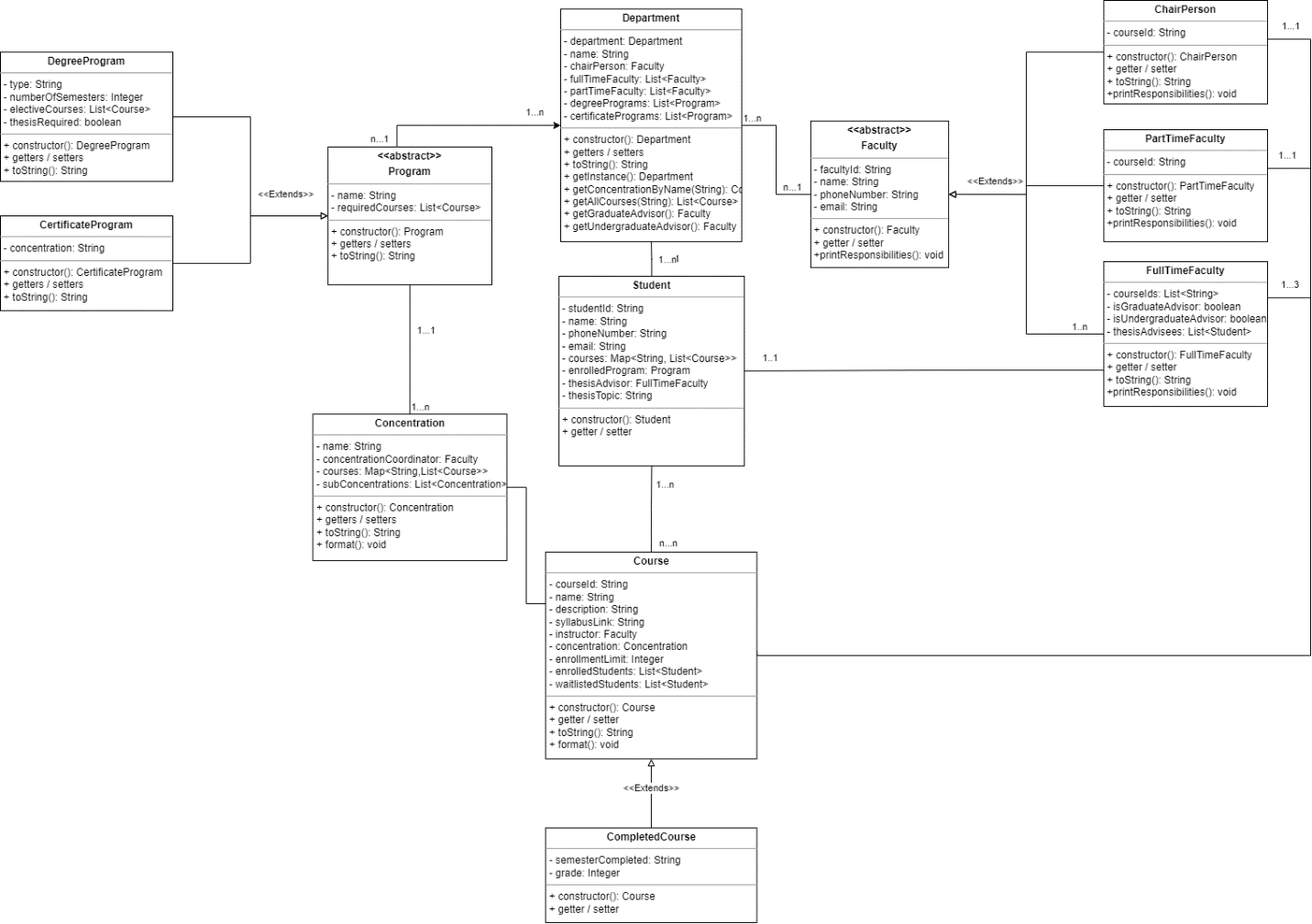


Figure : Class Diagram (Objects, relationships, multiplicity)

The sections below outline the various design patterns used and are organized into three main categories: Creational, Structural, and Behavioral.

# Creational Design Patterns

## Singleton Pattern

This relatively simple pattern was applied to the Department class which acts as the main entity object and container of all other relevant data for this application. Given its importance to the application’s data integrity and since the requirements only specify one Department object, the singleton allows our application to only have one instance of the Department object during project execution. Through the user of a private constructor and a public “getInstance” method, the singleton pattern is relatively easy to setup, however with its high utility, references to this method can be found throughout the project. It is called particularly heavily in the AbstractFactory related classes to ensure that all data created is properly associated to the Department.

Below is some sample console output for this design pattern. This highlighted message below is invoked in the private constructor for Department and only prints to the console once since only one instance will exist within the application runtime.

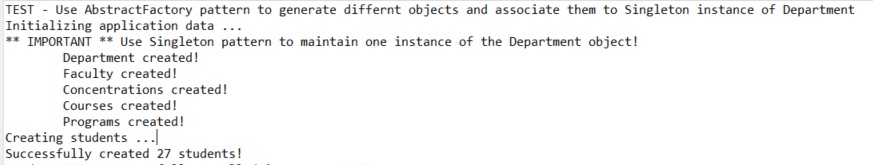


Figure : Singleton Console Output

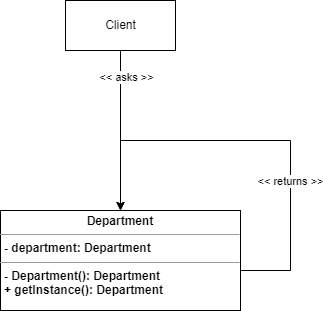
Below is a UML Class diagram showing the classes and relationships necessary for this pattern to function properly.

Figure : Singleton UML Diagram

## AbstractFactory Pattern

For my interpretation of the final project assignment, the abstract factory design pattern made the most sense to create the test data for the application in a controlled and organized manner.

Since this pattern is being used to generate data, the AbstractFactory interface only has one method: “create()” which has a void return type. This method has unique implementations across multiple different Factory classes for each of the main objects: Department, Concentration, Program, Faculty, and Course. These factory classes will create their respective model object, with hard-coded data, and associate it with the Department singleton instance. Certain factories benefit from more private static methods to get relevant data related to their model and associate it as necessary. An example of this would be the static methods of the DataCreationUtil class which are used in the FacultyFactory to create name, phoneNumber, and email addresses for each FullTimeFaculty and PartTimeFaculty instance.

Below is some sample console output for this design pattern. This text includes the singleton console output, however the tabbed lines are printed when a Factory class invokes their concrete implementation of the AbstractFactory ‘create’ method.

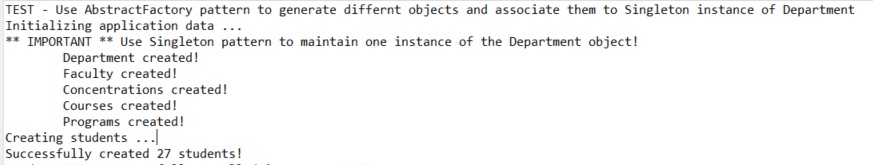


Figure : AbstractFactory Console Output

Below is a UML Class diagram showing the classes and relationships necessary for this pattern to function properly. The actual objects have been blanked for brevity.

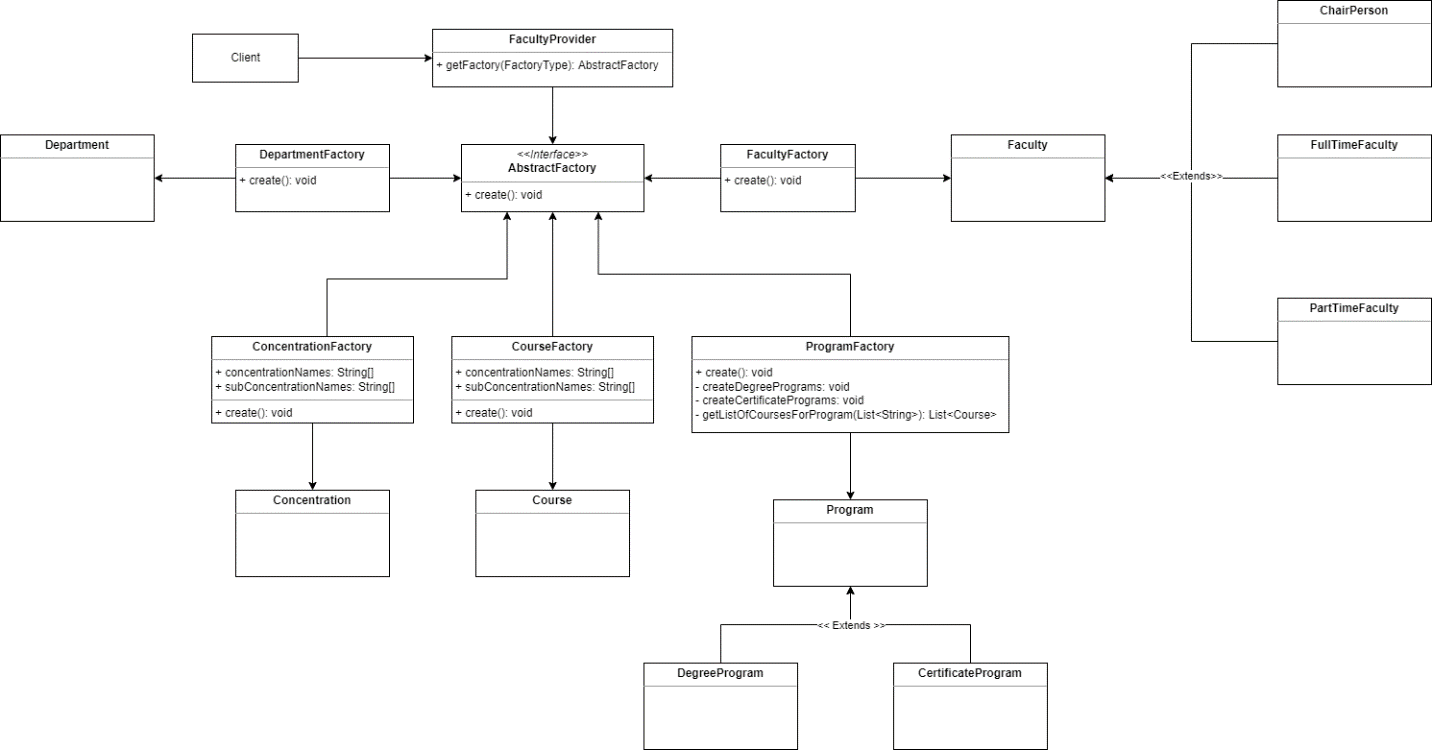


Figure : AbstractFactory UML Class Diagram

# Structural Design Patterns

## Façade Pattern

This pattern was used to add a layer of abstraction between the App.java in order to simplify how our application creates its first set of data. Since we are using an AbstractFactory pattern to create all of our data and the application has many objects which are all linked together, it makes abstract that logic into its own method which the main method invokes.

Below is some sample console output for this design pattern. This output is one of many examples of logic that is invoked by the façade pattern. This particular example is created when the ‘printCourseInformation’ method is invoked.

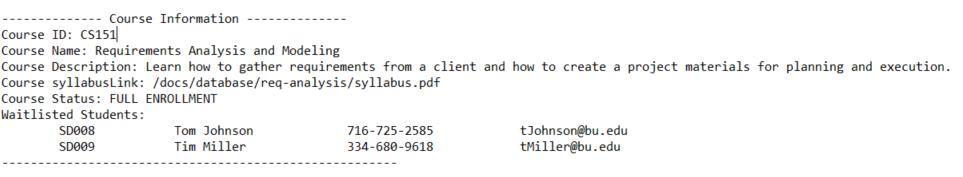


Figure : Facade Console Output

A “DepartmentFacade” class contains a method “initializeApplicationData” which instantiates the AbstractFactory and invokes the various “create” methods in order to generate all of the starting data. Other methods may be added to this façade in the future to support other patterns and to keep the main method lean and only invoking one-line methods to demonstrate functionality.

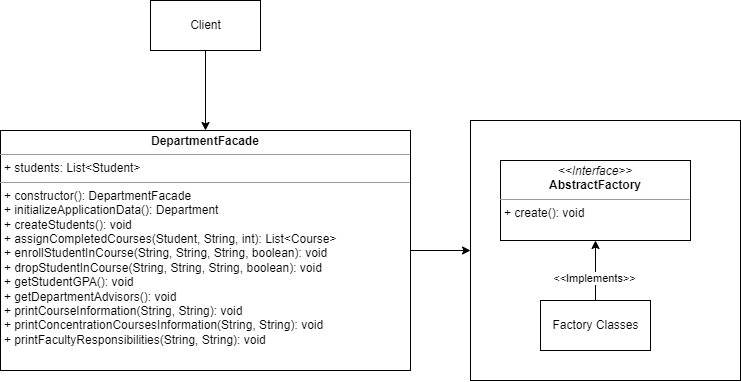


Figure : Facade UML Class Diagram

## Proxy pattern

The proxy design pattern was implemented as an intermediary step to perform validations and verifications before modifying data in the Department object. This is extremely important for the data integrity of our application since the data structure of the Department object is massive and complex. Ensuring that correct values and proper references to child objects are valid will help eliminate, or at least minimize, orphan data.

Below is some sample console output for this design pattern. The highlighted text is returned in the ‘verifyStudentId’ method of IVerify, if the provided studentId doesn’t exist with the List<Student> related to the Departmetnt.

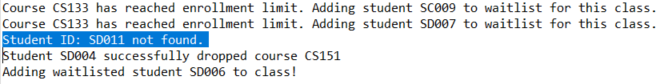


Figure : Proxy Console Output

Currently the proxy pattern is being used in the “enrollStudent” and “dropStudent” methods to verify the String “studentId” and “coursed” values to ensure that they correspond to valid data in the application. Since we are using these values to search through lists of objects, the proxy pattern allows use to retrieve those particular data sets only once and then hold them in memory for any future invocations of the enroll / drop methods.

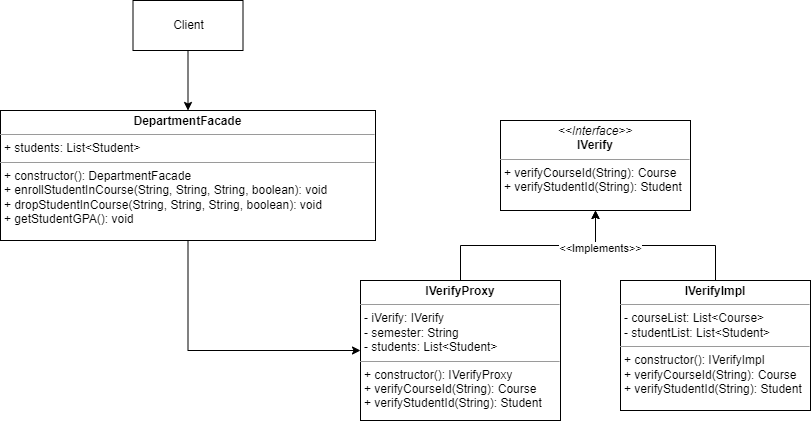


Figure : Proxy UML Class Diagram

# Behavioral Design Patterns

## Observer Pattern

The observer pattern was used in this application to monitor how students enroll and drop classes. Since each class has a waitlist, it is important to immediately fill any open spots with waitlisted students and to notify the course instructor about updates to the course roster.

Below is some sample console output for this design pattern. It shows the console output returned when students enroll for a class and there is space, when they enroll for a course and are put on the waitlist, and when another student drops the course and the first waitlisted student is added to the course and auto-enrolled. **The example console outputs relate to different courses and are shown for ease of demonstration, but full logs can be seen when running the application.**

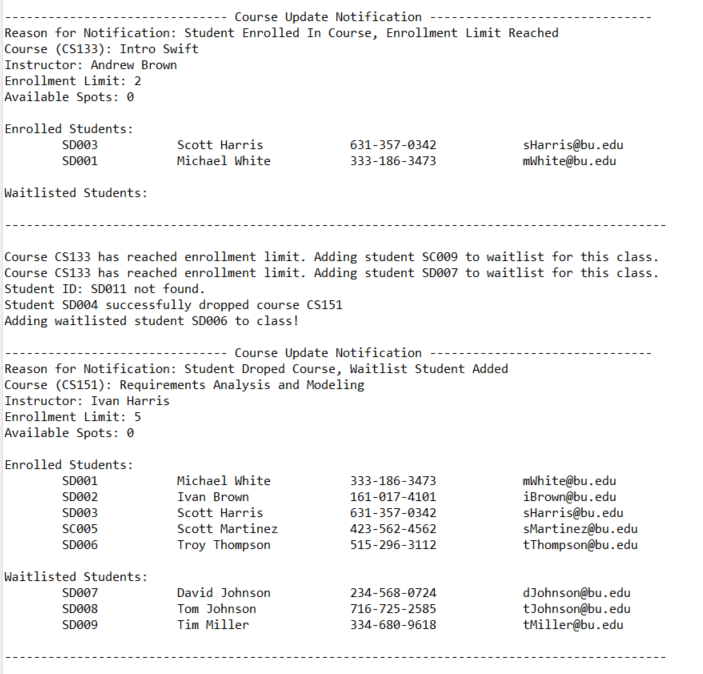


Figure : Observer Console Output

This pattern was implemented using a Subject class which takes in a Course when instantiated, and an abstract Observer class which defines two relevant method: enrollStudent() and dropStudent(). These methods have concrete implementations in the “CourseObserver” class, which updates the enrolledStudents (List<Student>) and waitlistedStudents(LinkedList<Student>) attributes of the Course class. For student enrollment, the code checks to see if the size of the enrolledStudents list is less than the enrollmentLimit attribute. If the size is less than this limit, then the student is successfully enrolled and added to the Course, however if the size is equal to or greater than the limit, then the student is added to the waitlistedStudents linked list. A linked list data structure is used to take advantage of Queue functionality and take the first element as spots become available when students drop from the class. Once the Course object has been updated, then mock “update” is sent to the course instructor, and for the purposes of this application it is printed to the console.

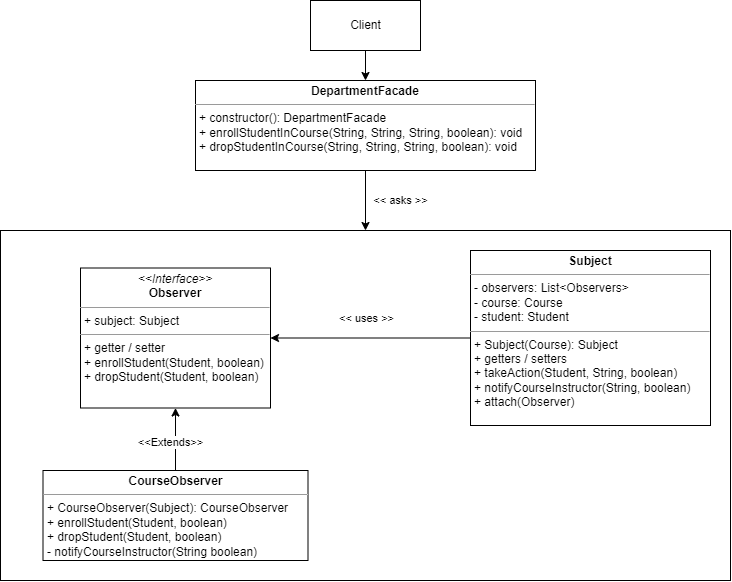


Figure 11: Observer UML Class Diagram

## Mediator

The mediator pattern was used in this application to abstract functionality related to various query functionalities, such as getting the undergraduate and graduate advisors of the department, getting a student’s gpa, and retrieving a concentration by name. These methods have been abstracted into their own Mediator class so that the methods of the Object don’t contain complicated logic and are easily maintained.

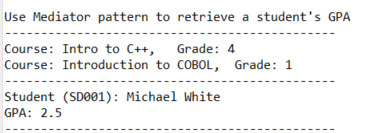
Below is some sample console output for this design pattern.

Figure : Mediator Console Output

This pattern was implemented for the Student and Department objects. Methods, such as “getGPA” for the Student object, were created in the their relevant object and reference a static method in its respective Mediator class. This class contains the concrete logic to calculate values and print to the console. Due to their utility, some of these methods are also referenced in the DepartmentFacade class.

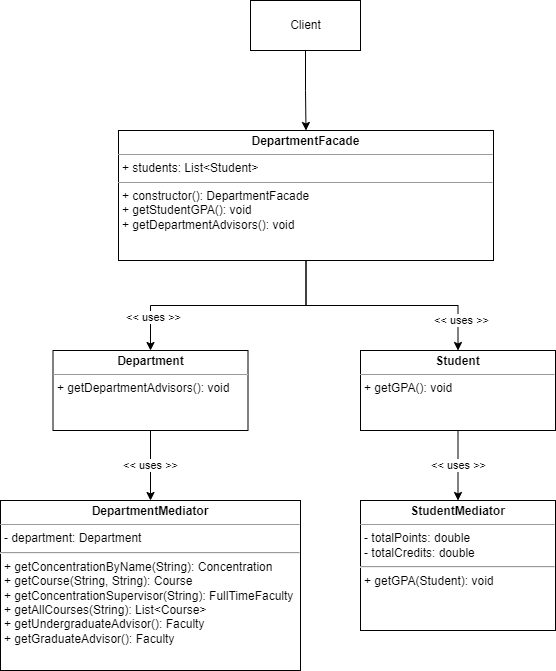


Figure : Mediator UML Class Diagram

**Special thank you to the facilitators of CS665 for all of their help and assistance with clarifying the final assignment and homework assignments. I really appreciate all of the hard work you do and the helpful feedback and the quick and clear communication you provided via email. Thank you!**