Harrison Huston

Department Manager

Preface:

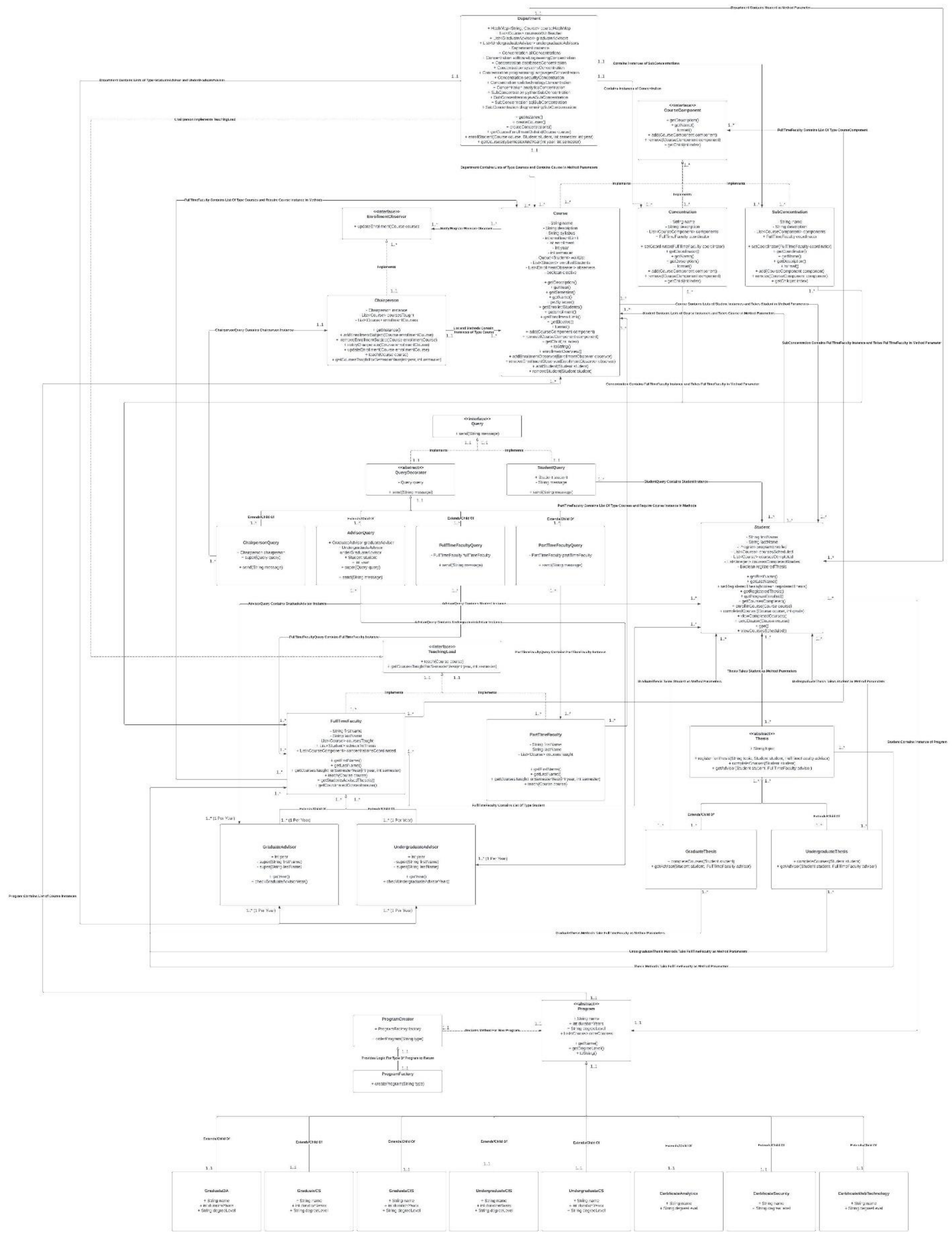
The program is located in the folder called DepartmentManager. The driver class is the Main.java file located within DepartmentManager > src > Main.java. Simply run the Main.java file to run the program. Additionally, within the same project folder, all images of the pattern diagrams and the class diagram will be included for readability, beyond what is included in this document.

Assumptions

Students take 2 courses per semester with 2 semesters per year. This means a bachelor's program would contain 16 courses, 12 of which are core courses, 3 are electives, and 1 is the thesis. A master's program would contain 8 courses, 4 of which are core courses, 3 are electives, and 1 is the thesis. A certificate program would contain 4 core courses, 0 electives, and no thesis.

Class Diagram

The below class diagram gives an overall depiction of the program, including the class structures, methods, variables, multiplicity, and relations to other classes. In looking at the below diagram, areas of interest should include the Student and Course classes, as these are potentially the most involved within the program. In addition, the Department class maintains much of the integrity and creational responsibilities of the program.

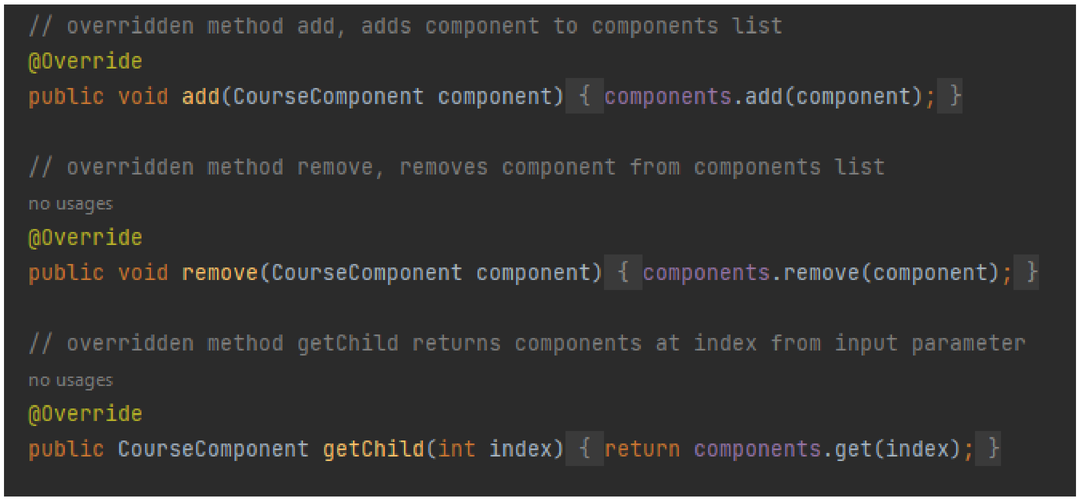

Department
+ HashMap<String, Course> courseHashMap
- List<Course> coursesWithTeacher
+ List<GraduateAdvisor> graduateAdvisors
+ List<UndergraduateAdvisor> undergraduateAdvisors
- Department instance
+ Concentration allConcentrations
- Concentration softwareEngineeringConcentration
+ Concentration databasesConcentration
+ Concentration systemsConcentration
+ Concentration programmingLanguagesConcentration
+ Concentration securityConcentration
+ Concentration webTechnologyConcentration
- Concentration analyticsConcentration
+ SubConcentration pythonSubConcentration
+ SubConcentration javaSubConcentration
- SubConcentration sqlSubConcentration
+ SubConcentration diagrammingSubConcentration
+ getInstance()
+ createCourses()
+ createConcentrations()
+ getCourseEnrollmentDetails(Course course)
+ enrollStudent(Course course, Student student, int semester, int year)
+ getCoursesBySemesterAndYear(int year, int semester)
Department Contains Student in Method Parameter
Department Contains Lists of Type GraduateAdvisor and UndergraduateAdvisor
Contains Instances of SubConcentrations
Contains Instance of Concentration
Chairperson Implements TeachingLoad
Department Contains Lists of Type Courses and Contains Course in Method Parameters
<<interface>>
CourseComponent
+ getDescription()
+ getName()
+ format()
+ add(CourseComponent component)
+ remove(CourseComponent component)
+ getChild(int index)
FullTimeFaculty Contains List Of Type CourseComponent
Implements
FullTimeFaculty Contains List Of Type Courses and Require Course Instance In Methods
<<interface>>
EnrollmentObserver
+ updateEnrollment(Course course)
Notify/Register/Remove Observer
Course
- String name
- String description
- String syllabus
- int enrollmentLimit
- int enrollment
- int year
- int semester
Queue<Student> waitList
- List<Student> enrolledStudents
- List<EnrollmentObserver> observers
- boolean elective
+ getDescription()
+ getYear()
+ getSemester()
+ getName()
+ getSyllabus()
+ getEnrolledStudents()
+ getEnrollment()
+ getEnrollmentLimit()
+ getElective()
+ format()
+ add(CourseComponent component)
+ remove(CourseComponent component)
+ getChild(int index)
+ toString()
+ enrollmentOverview()
+ addEnrollmentObserver(EnrollmentObserver observer)
+ removeEnrollmentObserver(EnrollmentObserver observer)
+ addStudent(Student student)
+ removeStudent(Student student)
Concentration
- String name
- String description
- List<CourseComponent> components
- FullTimeFaculty coordinator
+ setCoordinator(FullTimeFaculty coordinator)
+ getCoordinator()
+ getName()
+ getDescription()
+ format()
+ add(CourseComponent component)
+ remove(CourseComponent component)
+ getChild(int index)
SubConcentration
- String name
- String description
- List<CourseComponent> components
+ FullTimeFaculty coordinator
+ setCoordinator(FullTimeFaculty coordinator)
+ getCoordinator()
+ getName()
+ getDescription()
+ format()
+ add(CourseComponent component)
+ remove(CourseComponent component)
+ getChild(int index)
Chairperson
- Chairperson instance
List<Course> coursesTaught
- List<Course> enrollmentCourses
+ getInstance()
+ addEnrollmentSubject(Course enrollmentCourse)
+ removeEnrollmentSubject(Course enrollmentCourse)
+ notifyChairperson(Course enrollmentCourse)
+ updateEnrollment(Course enrollmentCourse)
+ teach(Course course)
+ getCoursesTaughtForSemesterYear(int year, int semester)
ChairpersonQuery Contains Chairperson Instance
List and Methods Contain Instances of Type Course
Course Contains Lists of Student Instances and Takes Student in Method Parameters
Student Contains Lists of Course Instances and Takes Course in Method Parameters
SubConcentration Contains FullTimeFaculty Instance and Takes FullTimeFaculty In Method Parameter
Concentration Contains FullTimeFaculty Instance and Takes FullTimeFaculty In Method Parameter
<<interface>>
Query
+ send(String message)
Implements
<<abstract>>
QueryDecorator
- Query query
+ send(String message)
StudentQuery
+ Student student
- String message
+ send(String message)
StudentQuery Contains Student Instance
PartTimeFaculty Contains List Of Type Courses and Require Course Instance In Methods
Extends/Child Of
ChairpersonQuery
- Chairperson chairperson
- super(Query query)
+ send(String message)
AdvisorQuery
+ GraduateAdvisor graduateAdvisor
- UndergraduateAdvisor underGraduateAdvisor
+ Student student
+ int year
+ super(Query query)
- send(String message)
FullTimeFacultyQuery
- FullTimeFaculty fullTimeFaculty
+ send(String message)
PartTimeFacultyQuery
- PartTimeFaculty partTimeFaculty
+ send(String message)
Student
- String firstName
- String lastName
- Program programEnrolled
- List<Course> coursesScheduled
- List<Course> coursesCompleted
- List<Integer> coursesCompletedGrades
- Boolean registeredThesis
+ getFirstName()
+ getLastName()
+ setRegisteredThesis(boolean registeredThesis)
+ getRegisteredThesis()
+ getProgramEnrolled()
+ getCoursesCompleted()
+ enrollInCourse(Course course)
+ completeCourse(Course course, int grade)
+ viewCompletedCourses()
+ dropCourse(Course course)
+ gpa()
+ viewCoursesScheduled()
AdvisorQuery Contains GraduateAdvisor Instance
AdvisorQuery Contains Student Instance
AdvisorQuery Contains UndergraduateAdvisor Instance
FullTimeFacultyQuery Contains FullTimeFaculty Instance
<<interface>>
TeachingLoad
+ teach(Course course)
+ getCoursesTaughtForSemesterYear(int year, int semester)
PartTimeFacultyQuery Contains PartTimeFaculty Instance
Thesis Takes Student as Method Parameters
GraduateThesis Takes Student as Method Parameters
UndergraduateThesis Takes Student as Method Parameters
FullTimeFaculty
- String firstName
- String lastName
List<Course> coursesTaught
+ List<Student> advisedInThesis
+ List<CourseComponent> concentrationsCoordinated
+ getFirstName()
+ getLastName()
+ getCoursesTaughtForSemesterYear(int year, int semester)
+ teach(Course course)
+ getStudentsAdvisedTheses()
+ getCoordinatedConcentrations()
PartTimeFaculty
- String firstName
String lastName
- List<Course> coursesTaught
+ getFirstName()
+ getLastName()
+ getCoursesTaughtForSemesterYear(int year, int semester)
+ teach(Course course)
FullTimeFaculty Contains List Of Type Student
<<abstract>>
Thesis
+ String topic
+ registerForThesis(String topic, Student student, FullTimeFaculty advisor)
+ completeCourses(Student student)
+ getAdvisor(Student student, FullTimeFaculty advisor)
Student Contains Instance of Program
1..* (1 Per Year)
GraduateAdvisor
+ int year
- super(String firstName)
- super(String lastName)
+ getYear()
+ checkGraduateAdvisorYear()
UndergraduateAdvisor
+ int year
- super(String firstName)
- super(String lastName)
+ getYear()
+ checkUndergraduateAdvisorYear()
GraduateThesis
+ completeCourses(Student student)
+ getAdvisor(Student student, FullTimeFaculty advisor)
UndergraduateThesis
+ completeCourses(Student student)
+ getAdvisor(Student student, FullTimeFaculty advisor)
Program Contains List of Course Instances
GraduateThesis Methods Take FullTimeFaculty as Method Parameters
UndergraduateThesis Methods Take FullTimeFaculty as Method Parameters
Thesis Methods Take FullTimeFaculty as Method Parameters
ProgramCreator
+ ProgramFactory factory
+ orderProgram(String type)
Declares Method For New Program
Provides Logic For Type Of Program to Return
ProgramFactory
+ createProgram(String type)
<<abstract>>
Program
+ String name
+ int durationYears
+ String degreeLevel
+ List<Course> coreCourses
+ getName()
+ getDegreeLevel()
+ toString()
GraduateDA
+ String name
+ int durationYears
+ String degreeLevel
GraduateCS
+ String name
+ int durationYears
+ String degreeLevel
GraduateCIS
+ String name
+ int durationYears
+ String degreeLevel
UndergraduateCIS
+ String name
+ int durationYears
+ String degreeLevel
UndergraduateCS
+ String name
+ int durationYears
+ String degreeLevel
CertificateAnalytics
+ String name
+ String degreeLevel
CertificateSecurity
+ String name
+ String degreeLevel
CertificateWebTechnology
+ String name
+ String degreeLevel

Individual Patterns

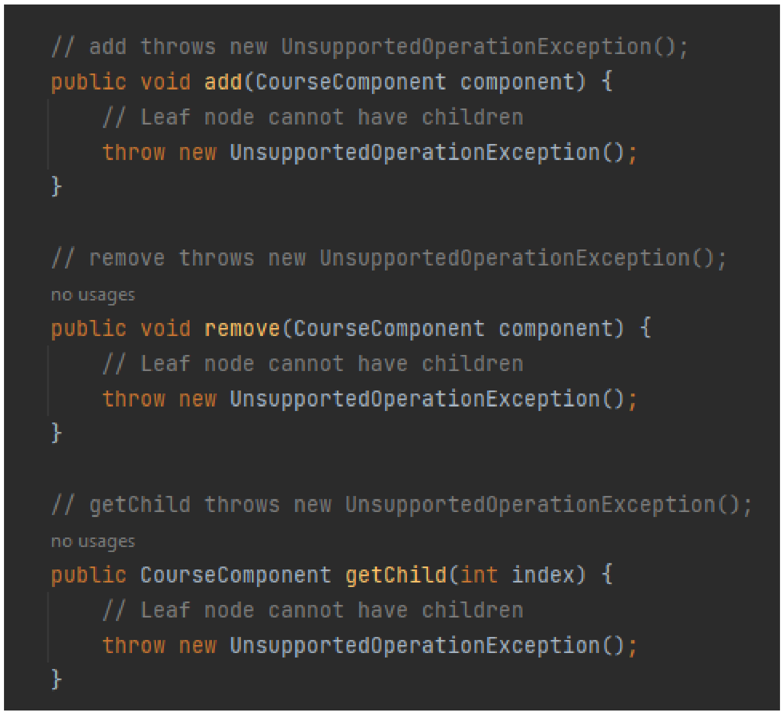
Structural Pattern : Composite

This pattern allows for the composition of objects into tree structures. This provides a hierarchical structure and allows for easier interactions with the structures. Due to these factors, it made sense to utilize this pattern for the course/concentration/sub concentration structure. In the design, the CourseComponent interface acted as the Component, with Concentration and SubConcentration classes acting as Composite, and the Course class as a Leaf.

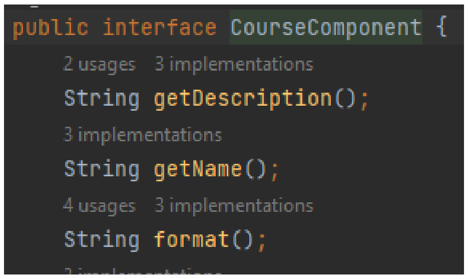
Important code in the implementation includes the Overridden add, remove, and getChild methods, which help maintain the pattern's structure and the Overridden getDescription, getName, and format methods which provide functionality to each class within the pattern.



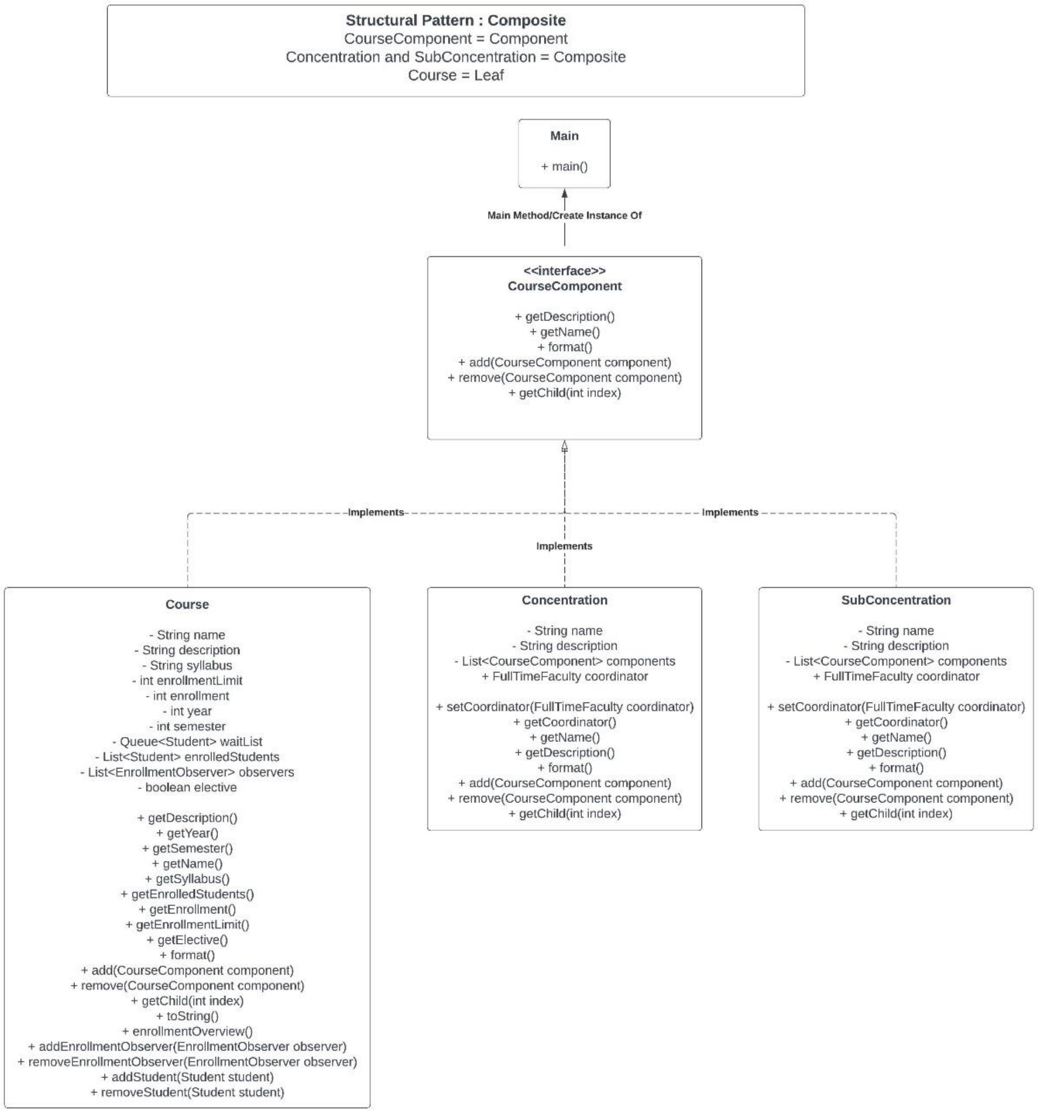
```
// overridden method add, adds component to components list
@Override
public void add(CourseComponent component) { components.add(component); }

// overridden method remove, removes component from components list
no usages
@Override
public void remove(CourseComponent component) { components.remove(component); }

// overridden method getChild returns components at index from input parameter
no usages
@Override
public CourseComponent getChild(int index) { return components.get(index); }
```



```
// add throws new UnsupportedOperationException();
public void add(CourseComponent component) {
    // Leaf node cannot have children
    throw new UnsupportedOperationException();
}

// remove throws new UnsupportedOperationException();
no usages
public void remove(CourseComponent component) {
    // Leaf node cannot have children
    throw new UnsupportedOperationException();
}

// getChild throws new UnsupportedOperationException();
no usages
public CourseComponent getChild(int index) {
    // Leaf node cannot have children
    throw new UnsupportedOperationException();
}
```



```
public interface CourseComponent {
    2 usages   3 implementations
    String getDescription();
    3 implementations
    String getName();
    4 usages   3 implementations
    String format();
```



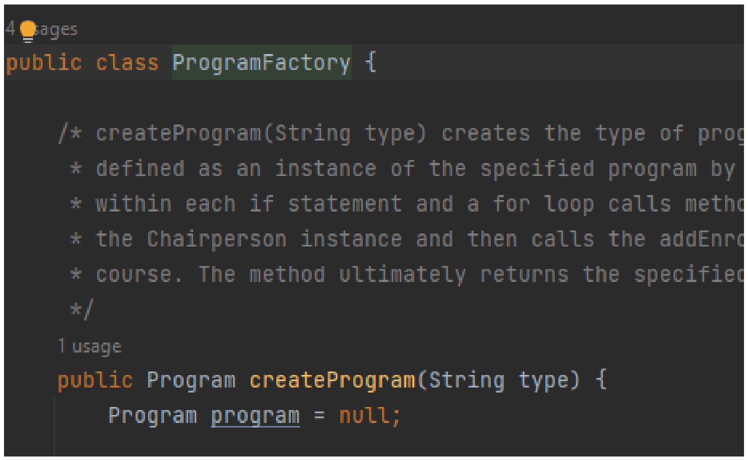
Structural Pattern : Composite
CourseComponent = Component
Concentration and SubConcentration = Composite
Course = Leaf

Main
+ main()
Main Method/Create Instance Of
<<interface>>
CourseComponent
+ getDescription()
+ getName()
+ format()
+ add(CourseComponent component)
+ remove(CourseComponent component)
+ getChild(int index)
Implements
Implements
Implements
Course
- String name
- String description
- String syllabus
- int enrollmentLimit
- int enrollment
- int year
- int semester
- Queue<Student> waitList
- List<Student> enrolledStudents
- List<EnrollmentObserver> observers
- boolean elective
+ getDescription()
+ getYear()
+ getSemester()
+ getName()
+ getSyllabus()
+ getEnrolledStudents()
+ getEnrollment()
+ getEnrollmentLimit()
+ getElective()
+ format()
+ add(CourseComponent component)
+ remove(CourseComponent component)
+ getChild(int index)
+ toString()
+ enrollmentOverview()
+ addEnrollmentObserver(EnrollmentObserver observer)
+ removeEnrollmentObserver(EnrollmentObserver observer)
+ addStudent(Student student)
+ removeStudent(Student student)
Concentration
- String name
- String description
- List<CourseComponent> components
+ FullTimeFaculty coordinator
+ setCoordinator(FullTimeFaculty coordinator)
+ getCoordinator()
+ getName()
+ getDescription()
+ format()
+ add(CourseComponent component)
+ remove(CourseComponent component)
+ getChild(int index)
SubConcentration
- String name
- String description
- List<CourseComponent> components
+ FullTimeFaculty coordinator
+ setCoordinator(FullTimeFaculty coordinator)
+ getCoordinator()
+ getName()
+ getDescription()
+ format()
+ add(CourseComponent component)
+ remove(CourseComponent component)
+ getChild(int index)

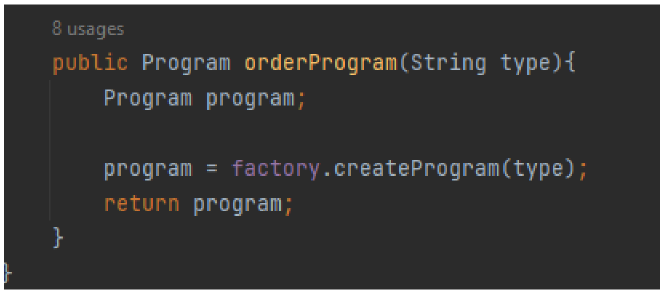
Creational Pattern : Factory Method

This pattern allows for the creation of objects in a superclass and allows the type of objects to be altered by the subclasses that create them. The pattern made sense to use as it pertained to creating the various programs offered, because they were not all entirely uniform. The pattern allowed for the creation of various graduate, undergraduate, and certificate programs with varying specifications. In the design within the project, the Program class is the Product, the GraduateDA, GraduateCS, GraduateCIS, UndergraduateCIS, UndergraduateCS, CertificateAnalytics, CertificateSecurity, and CertificateWebTechnology classes are Concrete Products, while the ProgramCreator is the Creator and the ProgramFactory is the Concrete Creator.

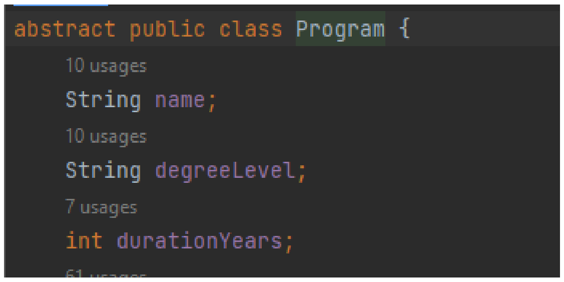
Important code in the implementation includes the createProgram method which helps decipher the program to be created, along with the orderProgram method. Additionally, the variables of each program help to distinguish it from the other Concrete Products.



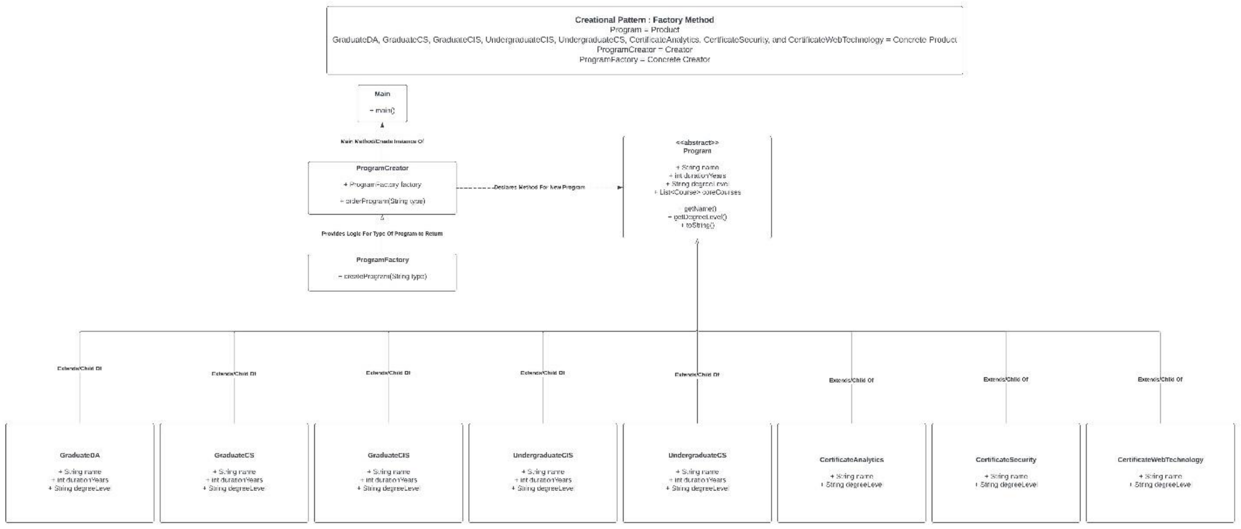
```
4 usages
public class ProgramFactory {

    /* createProgram(String type) creates the type of prog
     * defined as an instance of the specified program by
     * within each if statement and a for loop calls metho
     * the Chairperson instance and then calls the addEnro
     * course. The method ultimately returns the specified
     */
    1 usage
    public Program createProgram(String type) {
        Program program = null;
```



```
    8 usages
    public Program orderProgram(String type){
        Program program;

        program = factory.createProgram(type);
        return program;
    }
}
```



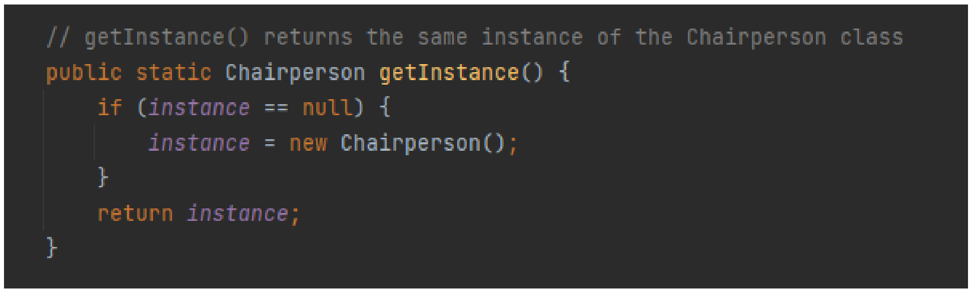
```
abstract public class Program {
    10 usages
    String name;
    10 usages
    String degreeLevel;
    7 usages
    int durationYears;
```


Creational Pattern : Factory Method
Program = Product
GraduateDA, GraduateCS, GraduateCIS, UndergraduateCIS, UndergraduateCS, CertificateAnalytics. CertificateSecurity, and CertificateWebTechnology = Concrete Product
ProgramCreator = Creator
ProgramFactory = Concrete Creator
Main
+ main()
Main Method/Create Instance Of
ProgramCreator
+ ProgramFactory factory
+ orderProgram(String type)
Declares Method For New Program
<<abstract>>
Program
+ String name
+ int durationYears
+ String degreeLevel
+ List<Course> coreCourses
getName()
+ getDegreeLevel()
+ toString()
Provides Logic For Type Of Program to Return
ProgramFactory
+ createProgram(String type)
Extends/Child Of
Extends/Child Of
Extends/Child Of
Extends/Child Of
Extends/Child Of
Extends/Child Of
Extends/Child Of
Extends/Child Of
GraduateDA
+ String name
+ int durationYears
+ String degreeLevel
GraduateCS
+ String name
+ int durationYears
+ String degreeLevel
GraduateCIS
+ String name
+ int durationYears
+ String degreeLevel
UndergraduateCIS
+ String name
+ int durationYears
+ String degreeLevel
UndergraduateCS
+ String name
+ int durationYears
+ String degreeLevel
CertificateAnalytics
+ String name
+ String degreeLevel
CertificateSecurity
+ String name
+ String degreeLevel
CertificateWebTechnology
+ String name
+ String degreeLevel

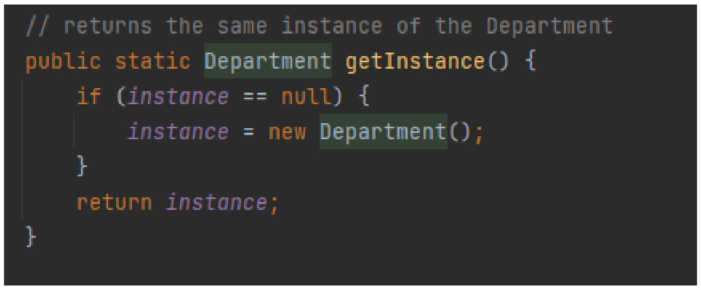
Creational Pattern : Singleton

This pattern ensures that a class has only one instance. It also allows for a global access point to this instance. This pattern was useful in regard to the Chairperson and Department classes. Both of which acted as the Singleton in their respective Singleton design pattern. Since there is only one department and one chairperson in the scope of the project, using this design pattern made perfect sense. Along with this, they needed to be easily accessed globally, as they both played important roles in the overall design and implementation of the project.

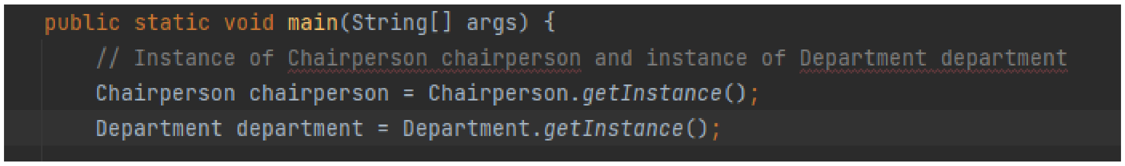
Important code in the Singleton pattern that is worth noting, is the getInstance method with the Department and Chairperson classes. This creates a new instance if the instance is null and returns that instance. If the instance is not null, it simply returns the instance. This ensures only a single instance exists at any given time. Calling the getInstance method allows for global access.



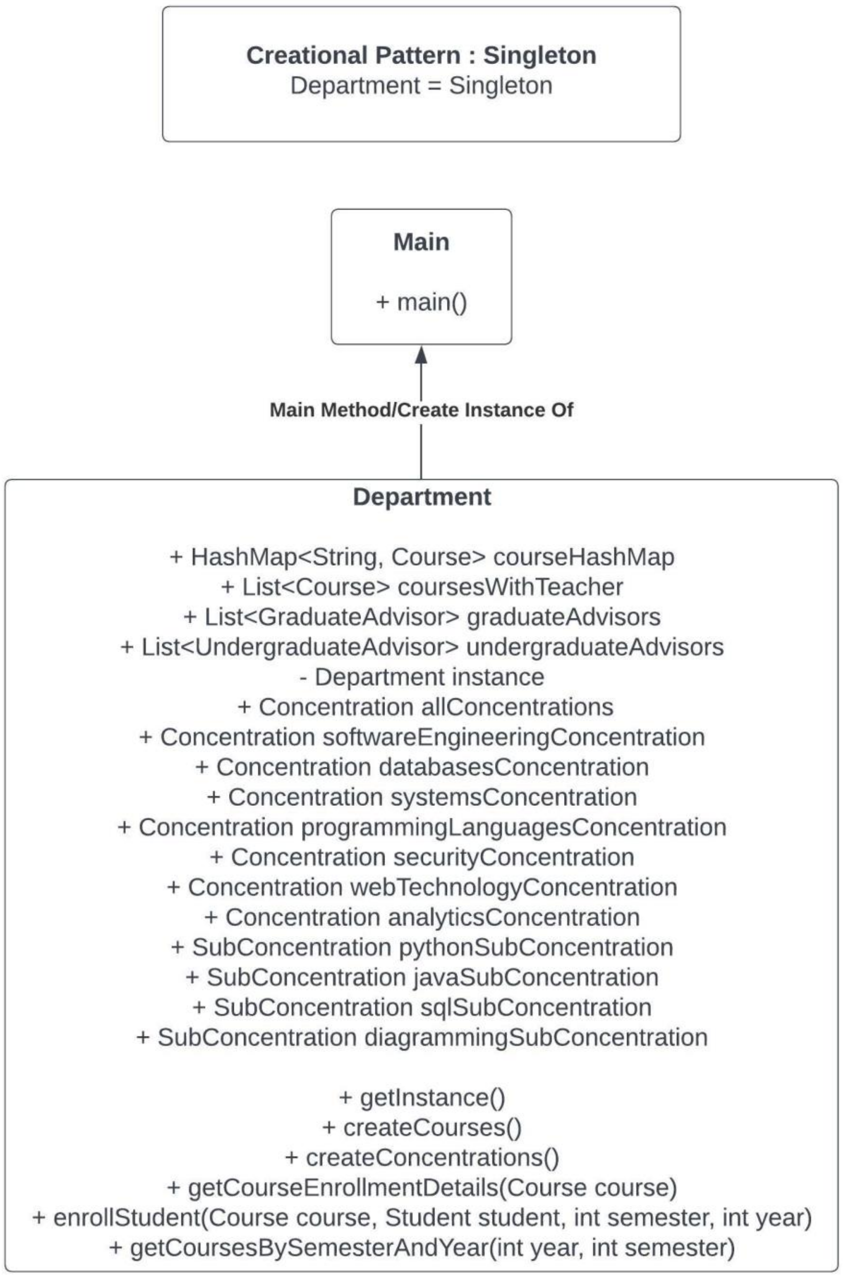
```java
// getInstance() returns the same instance of the Chairperson class
public static Chairperson getInstance() {
    if (instance == null) {
        instance = new Chairperson();
    }
    return instance;
}
```

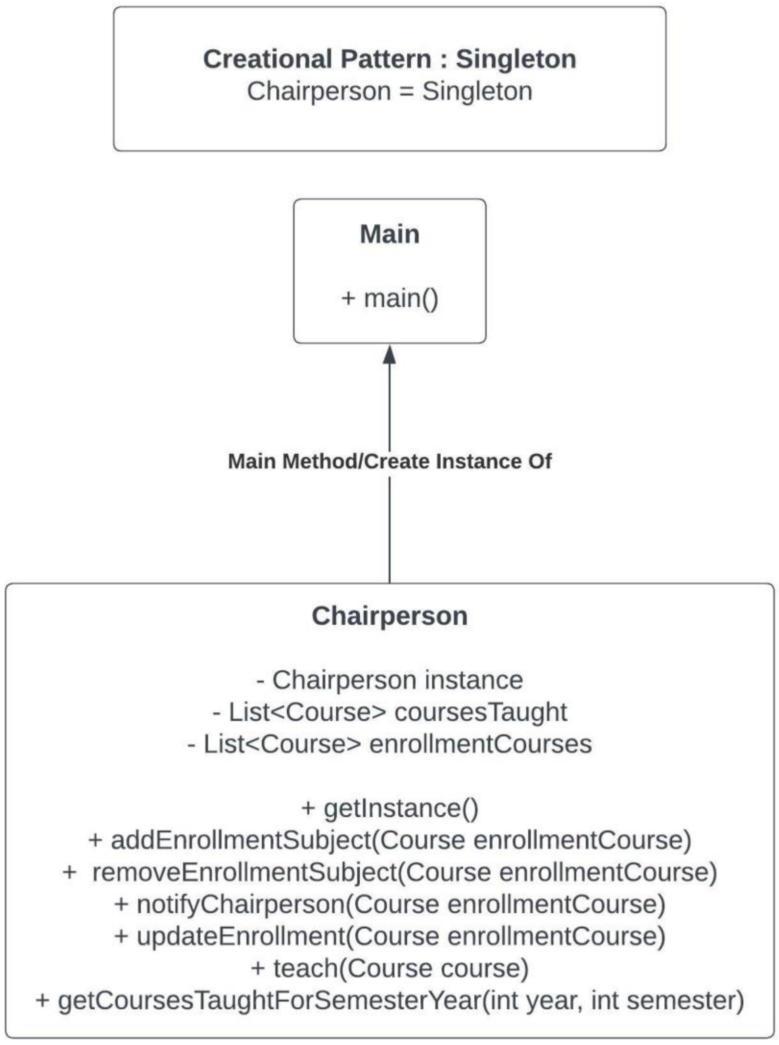


```java
// returns the same instance of the Department
public static Department getInstance() {
    if (instance == null) {
        instance = new Department();
    }
    return instance;
}
```



```java
public static void main(String[] args) {
    // Instance of Chairperson chairperson and instance of Department department
    Chairperson chairperson = Chairperson.getInstance();
    Department department = Department.getInstance();
```

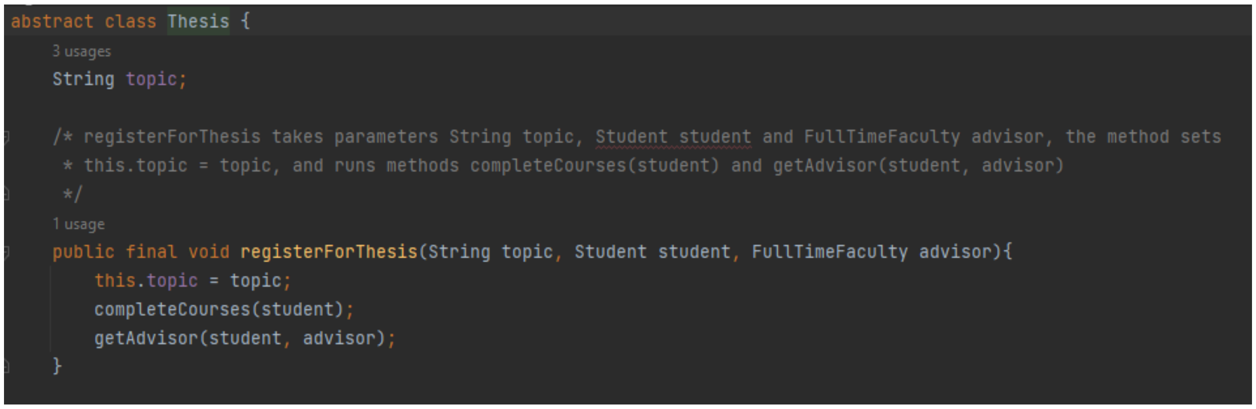

Creational Pattern : Singleton
Department = Singleton
Main
+ main()
Main Method/Create Instance Of
Department
+ HashMap<String, Course> courseHashMap
+ List<Course> coursesWithTeacher
+ List<GraduateAdvisor> graduateAdvisors
+ List<UndergraduateAdvisor> undergraduateAdvisors
- Department instance
+ Concentration allConcentrations
+ Concentration softwareEngineeringConcentration
+ Concentration databasesConcentration
+ Concentration systemsConcentration
+ Concentration programmingLanguagesConcentration
+ Concentration securityConcentration
+ Concentration webTechnologyConcentration
+ Concentration analyticsConcentration
+ SubConcentration pythonSubConcentration
+ SubConcentration javaSubConcentration
+ SubConcentration sqlSubConcentration
+ SubConcentration diagrammingSubConcentration
+ getInstance()
+ createCourses()
+ createConcentrations()
+ getCourseEnrollmentDetails(Course course)
+ enrollStudent(Course course, Student student, int semester, int year)
+ getCoursesBySemesterAndYear(int year, int semester)


Creational Pattern : Singleton
Chairperson = Singleton
Main
+ main()
Main Method/Create Instance Of
Chairperson
- Chairperson instance
- List<Course> coursesTaught
- List<Course> enrollmentCourses
+ getInstance()
+ addEnrollmentSubject(Course enrollmentCourse)
+ removeEnrollmentSubject(Course enrollmentCourse)
+ notifyChairperson(Course enrollmentCourse)
+ updateEnrollment(Course enrollmentCourse)
+ teach(Course course)
+ getCoursesTaughtForSemesterYear(int year, int semester)

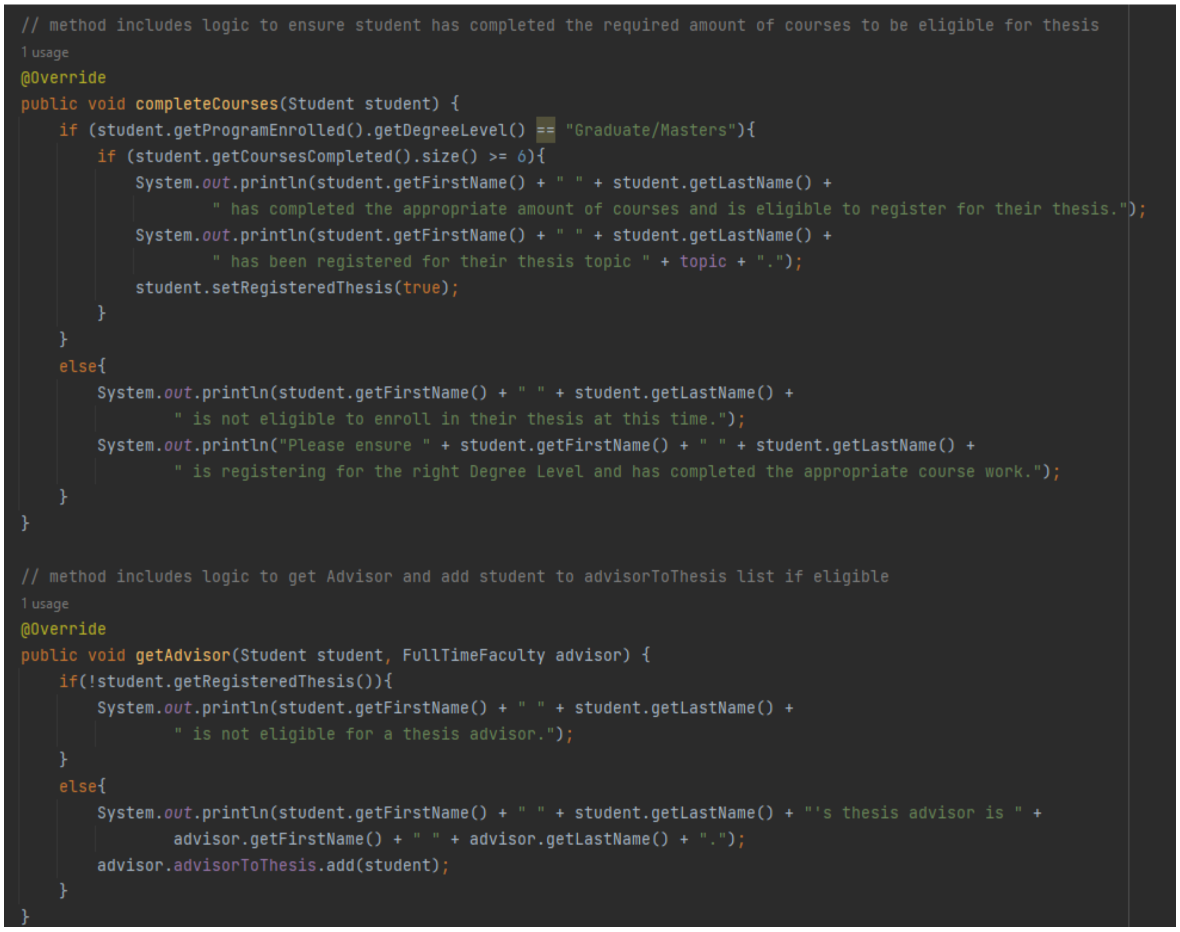
Behavioral Pattern : Template Method

This pattern allows for steps to be created as a part of a process that needs to be completed. The steps are defined in a superclass and then allowed to be overridden by subclasses, without changing the structure of the overall algorithm. The pattern worked well in implementing the thesis aspects of the project. The thesis required that certain criteria be met before a student was allowed to register for their thesis, lending itself to the Template Method. The pattern contains the abstract Thesis class, acting as the Abstract Class, with the UndergraduateThesis and GraduateThesis classes acting as Concrete Classes.

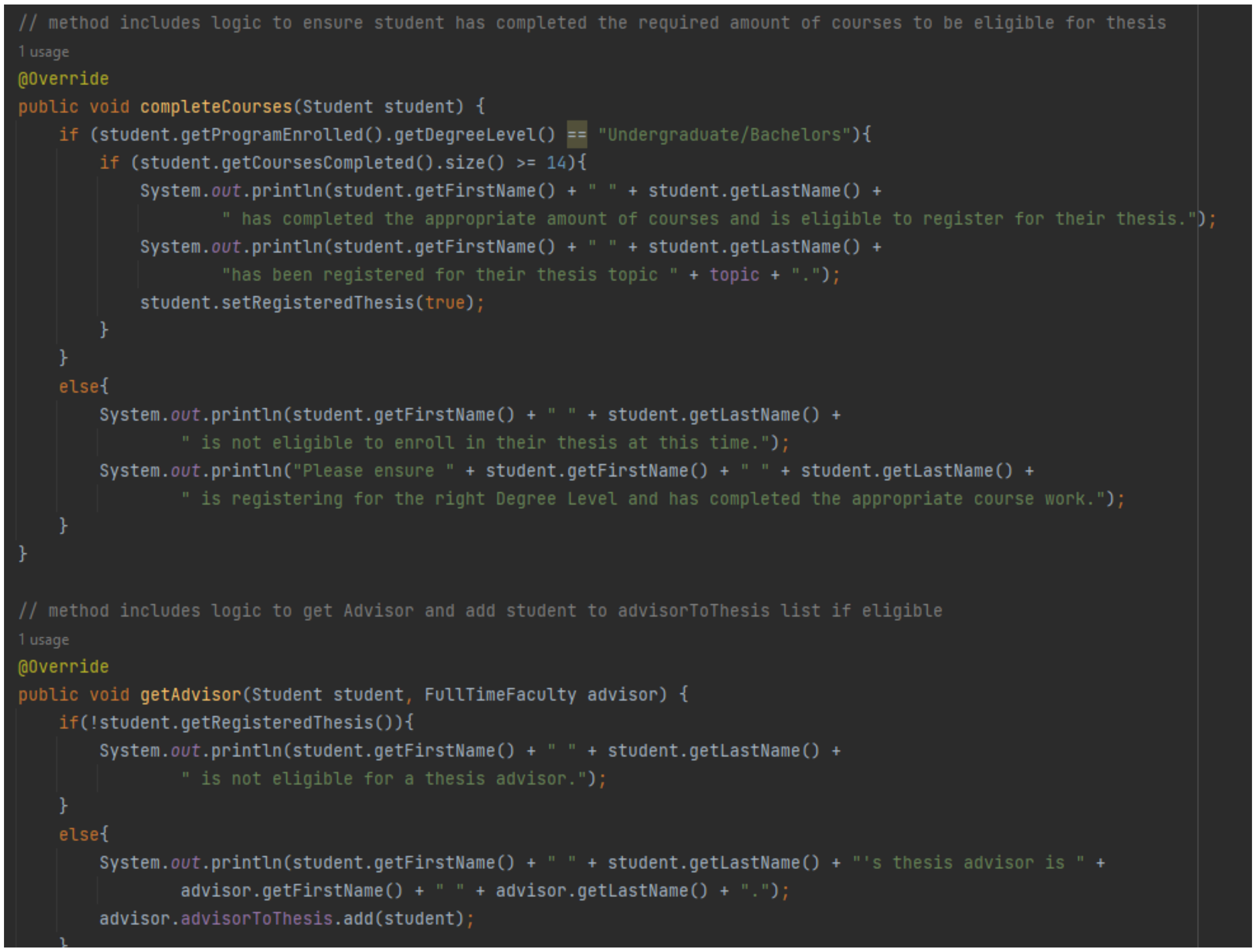
Important code includes the registerForThesis method in the abstract Thesis class, along with the Overridden methods completeCourses and getAdvisor. The structure of these methods allows for a step-by-step process to registering for the thesis.



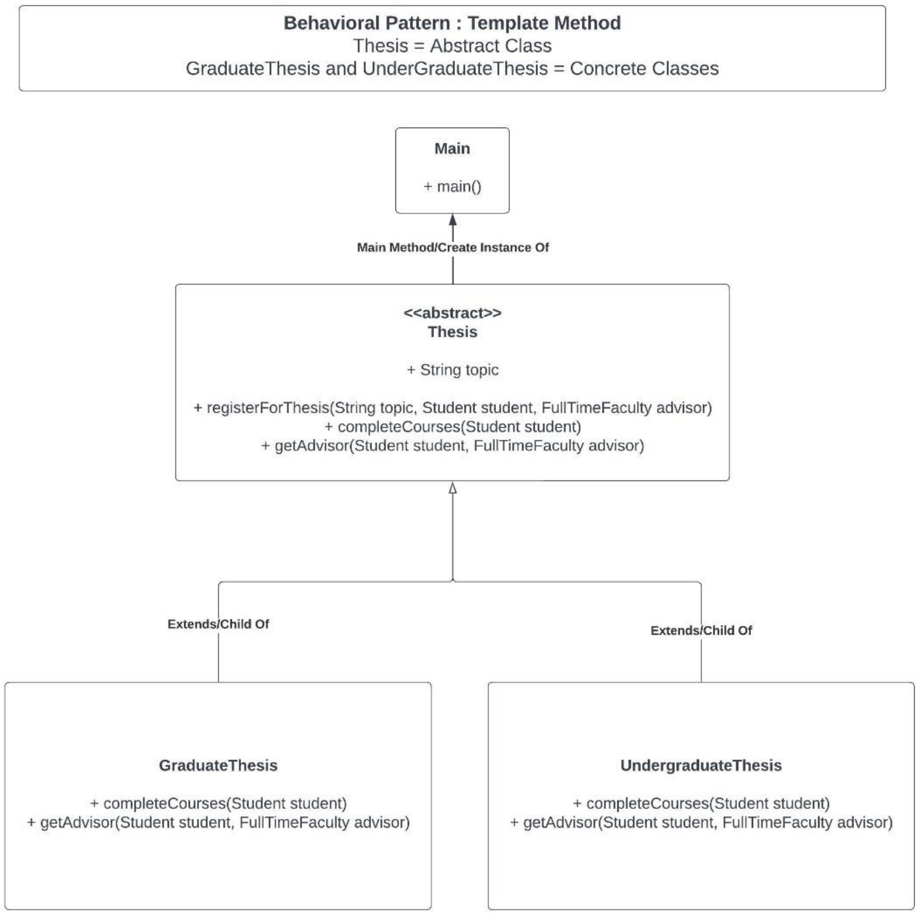
```
abstract class Thesis {
    3 usages
    String topic;

    /* registerForThesis takes parameters String topic, Student student and FullTimeFaculty advisor, the method sets
     * this.topic = topic, and runs methods completeCourses(student) and getAdvisor(student, advisor)
     */
    1 usage
    public final void registerForThesis(String topic, Student student, FullTimeFaculty advisor){
        this.topic = topic;
        completeCourses(student);
        getAdvisor(student, advisor);
    }
```



```java
// method includes logic to ensure student has completed the required amount of courses to be eligible for thesis
1 usage
@Override
public void completeCourses(Student student) {
    if (student.getProgramEnrolled().getDegreeLevel() == "Graduate/Masters"){
        if (student.getCoursesCompleted().size() >= 6){
            System.out.println(student.getFirstName() + " " + student.getLastName() +
                    " has completed the appropriate amount of courses and is eligible to register for their thesis.");
            System.out.println(student.getFirstName() + " " + student.getLastName() +
                    " has been registered for their thesis topic " + topic + ".");
            student.setRegisteredThesis(true);
        }
    }
    else{
        System.out.println(student.getFirstName() + " " + student.getLastName() +
                " is not eligible to enroll in their thesis at this time.");
        System.out.println("Please ensure " + student.getFirstName() + " " + student.getLastName() +
                " is registering for the right Degree Level and has completed the appropriate course work.");
    }
}

// method includes logic to get Advisor and add student to advisorToThesis list if eligible
1 usage
@Override
public void getAdvisor(Student student, FullTimeFaculty advisor) {
    if(!student.getRegisteredThesis()){
        System.out.println(student.getFirstName() + " " + student.getLastName() +
                " is not eligible for a thesis advisor.");
    }
    else{
        System.out.println(student.getFirstName() + " " + student.getLastName() + "'s thesis advisor is " +
                advisor.getFirstName() + " " + advisor.getLastName() + ".");
        advisor.advisorToThesis.add(student);
    }
}
```



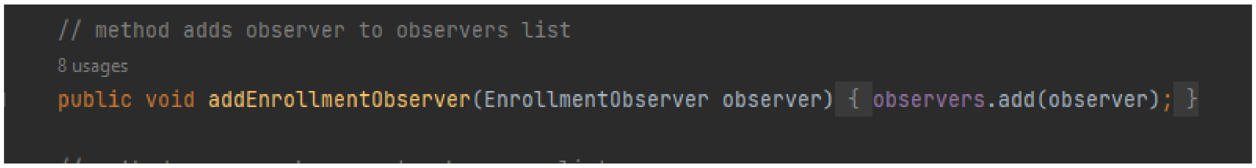
```
// method includes logic to ensure student has completed the required amount of courses to be eligible for thesis
1 usage
@Override
public void completeCourses(Student student) {
    if (student.getProgramEnrolled().getDegreeLevel() == "Undergraduate/Bachelors"){
        if (student.getCoursesCompleted().size() >= 14){
            System.out.println(student.getFirstName() + " " + student.getLastName() +
                    " has completed the appropriate amount of courses and is eligible to register for their thesis.");
            System.out.println(student.getFirstName() + " " + student.getLastName() +
                    "has been registered for their thesis topic " + topic + ".");
            student.setRegisteredThesis(true);
        }
    }
    else{
        System.out.println(student.getFirstName() + " " + student.getLastName() +
                " is not eligible to enroll in their thesis at this time.");
        System.out.println("Please ensure " + student.getFirstName() + " " + student.getLastName() +
                " is registering for the right Degree Level and has completed the appropriate course work.");
    }
}

// method includes logic to get Advisor and add student to advisorToThesis list if eligible
1 usage
@Override
public void getAdvisor(Student student, FullTimeFaculty advisor) {
    if(!student.getRegisteredThesis()){
        System.out.println(student.getFirstName() + " " + student.getLastName() +
                " is not eligible for a thesis advisor.");
    }
    else{
        System.out.println(student.getFirstName() + " " + student.getLastName() + "'s thesis advisor is " +
                advisor.getFirstName() + " " + advisor.getLastName() + ".");
        advisor.advisorToThesis.add(student);
```


Behavioral Pattern : Template Method
Thesis = Abstract Class
GraduateThesis and UnderGraduateThesis = Concrete Classes
Main
+ main()
Main Method/Create Instance Of
<<abstract>>
Thesis
+ String topic
+ registerForThesis(String topic, Student student, FullTimeFaculty advisor)
+ completeCourses(Student student)
+ getAdvisor(Student student, FullTimeFaculty advisor)
Extends/Child Of
Extends/Child Of
GraduateThesis
+ completeCourses(Student student)
+ getAdvisor(Student student, FullTimeFaculty advisor)
UndergraduateThesis
+ completeCourses(Student student)
+ getAdvisor(Student student, FullTimeFaculty advisor)

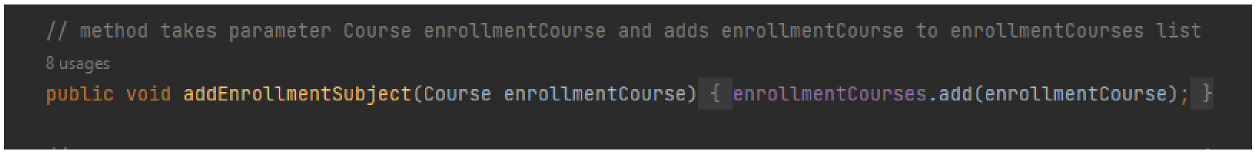
Behavioral Pattern : Observer

This pattern allows for a subscription mechanism. This allows for multiple objects to be notified as to changes or events that happen pertaining to an object they are observing. This pattern lent itself well to the chairperson and course scenario in the project description, in that the chairperson needed to be made aware of any time a course became full. Within the pattern, the Course class is the Publisher, the EnrollmentObserver is the Subscriber, and the Chairperson is the Concrete Subscriber.

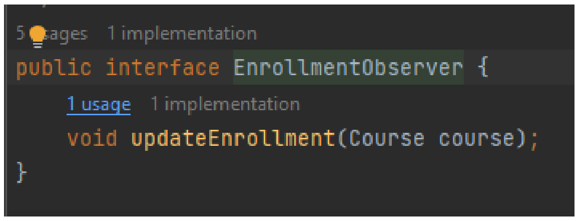
Important code to note in this pattern, is the addEnrollmentObserver method within the Course class, addEnrollmentSubject within the Chairperson class, and the Overridden updateEnrollment method.



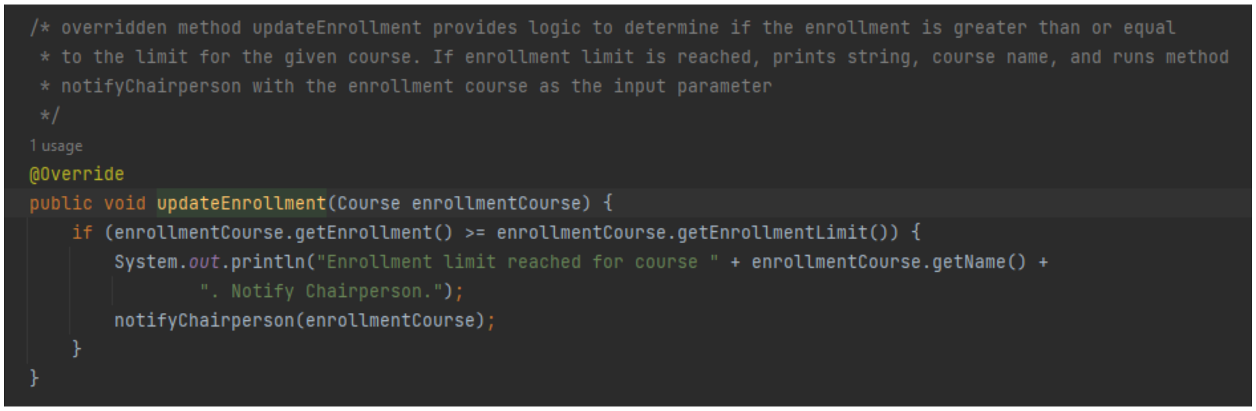
```
// method adds observer to observers list
8 usages
public void addEnrollmentObserver(EnrollmentObserver observer) { observers.add(observer); }
```



```
// method takes parameter Course enrollmentCourse and adds enrollmentCourse to enrollmentCourses list
8 usages
public void addEnrollmentSubject(Course enrollmentCourse) { enrollmentCourses.add(enrollmentCourse); }
```

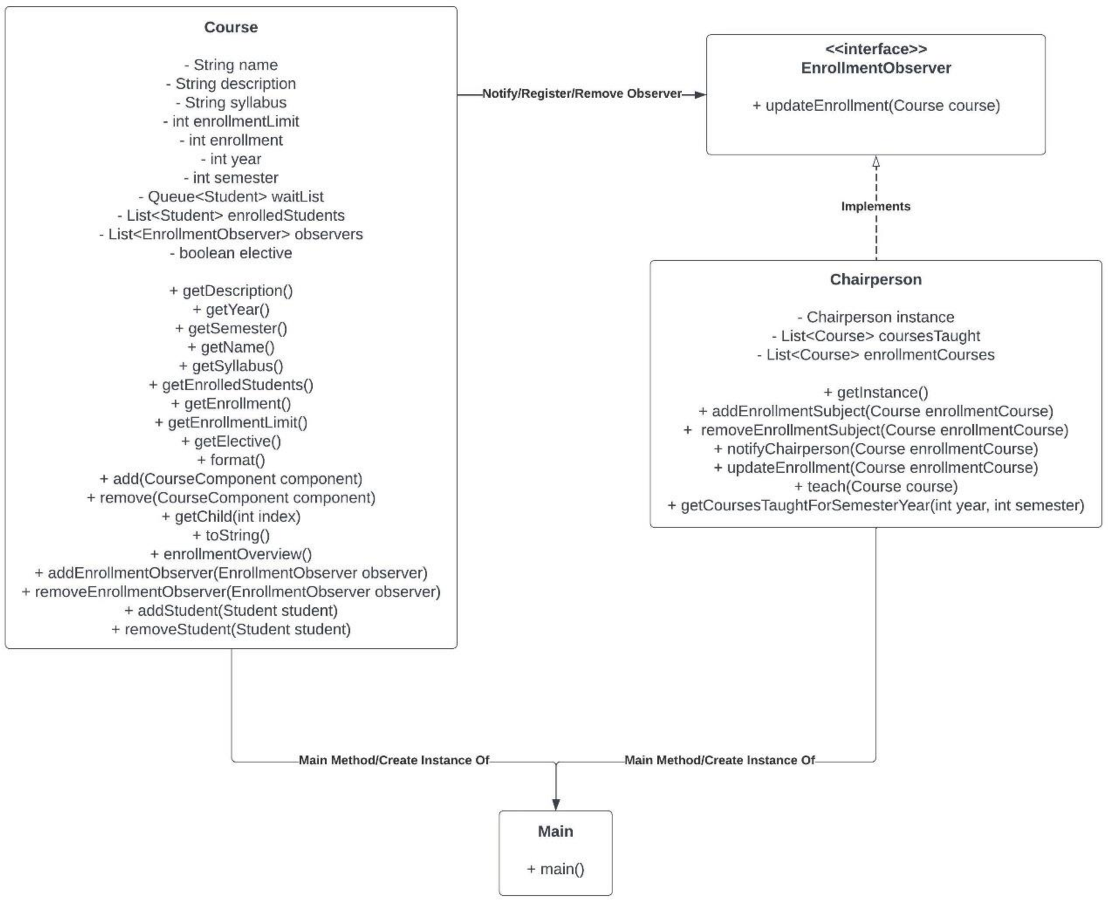


```
5 usages   1 implementation
public interface EnrollmentObserver {
    1 usage   1 implementation
    void updateEnrollment(Course course);
}
```



```
/* overridden method updateEnrollment provides logic to determine if the enrollment is greater than or equal
 * to the limit for the given course. If enrollment limit is reached, prints string, course name, and runs method
 * notifyChairperson with the enrollment course as the input parameter
 */
1 usage
@Override
public void updateEnrollment(Course enrollmentCourse) {
    if (enrollmentCourse.getEnrollment() >= enrollmentCourse.getEnrollmentLimit()) {
        System.out.println("Enrollment limit reached for course " + enrollmentCourse.getName() +
                ". Notify Chairperson.");
        notifyChairperson(enrollmentCourse);
    }
}
```

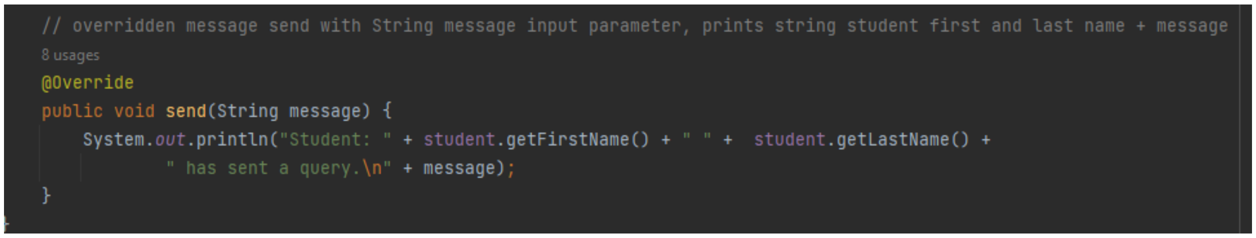
Behavioral Pattern : Observer
Course = Publisher
EnrollmentObserver = Subscriber
Chairperson = Concrete Subscriber


Course
- String name
- String description
- String syllabus
- int enrollmentLimit
- int enrollment
- int year
- int semester
- Queue<Student> waitList
- List<Student> enrolledStudents
- List<EnrollmentObserver> observers
- boolean elective
+ getDescription()
+ getYear()
+ getSemester()
+ getName()
+ getSyllabus()
+ getEnrolledStudents()
+ getEnrollment()
+ getEnrollmentLimit()
+ getElective()
+ format()
+ add(CourseComponent component)
+ remove(CourseComponent component)
+ getChild(int index)
+ toString()
+ enrollmentOverview()
+ addEnrollmentObserver(EnrollmentObserver observer)
+ removeEnrollmentObserver(EnrollmentObserver observer)
+ addStudent(Student student)
+ removeStudent(Student student)
Notify/Register/Remove Observer
<<interface>>
EnrollmentObserver
+ updateEnrollment(Course course)
Implements
Chairperson
- Chairperson instance
- List<Course> coursesTaught
- List<Course> enrollmentCourses
+ getInstance()
+ addEnrollmentSubject(Course enrollmentCourse)
+ removeEnrollmentSubject(Course enrollmentCourse)
+ notifyChairperson(Course enrollmentCourse)
+ updateEnrollment(Course enrollmentCourse)
+ teach(Course course)
+ getCoursesTaughtForSemesterYear(int year, int semester)
Main Method/Create Instance Of
Main Method/Create Instance Of
Main
+ main()

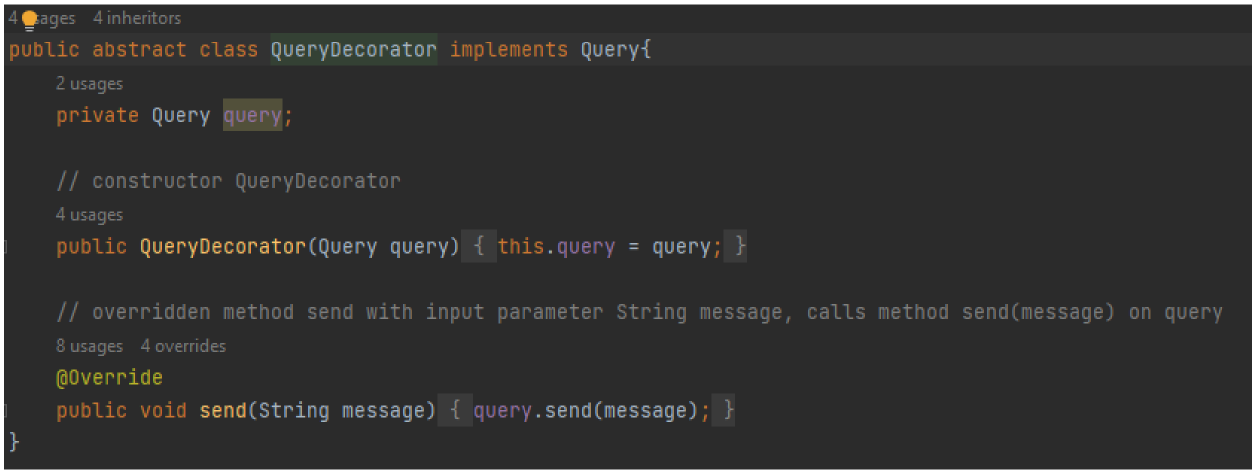
Structural Pattern : Decorator

The decorator pattern allows for new behaviors to be attached to objects. This is done by placing them within separate objects that contain the behaviors, often called wrapper objects. With these capabilities in mind, the pattern worked well in implementing the query aspect of the project. This required that queries be able to be sent to various "people/faculty" within the department. Because the queries differed in their recipients, the task was applicable to the Decorator pattern. Within the pattern, the Query interface acts as the Component, with the StudentQuery class acting as the Concrete Component, and the QueryDecorator class acting as the Base Decorator. The ChairpersonQuery, AdvisorQuery, FullTimeFacultyQuery, and PartTimeFacultyQuery classes are the Concrete Decorators.

Important aspects of the code include the Overridden send method in the StudentQuery class and QueryDecorator abstract class, as well as Overridden send methods within each Concrete Decorator. Each Concrete Decorator has their own implementation of the Overridden send method.

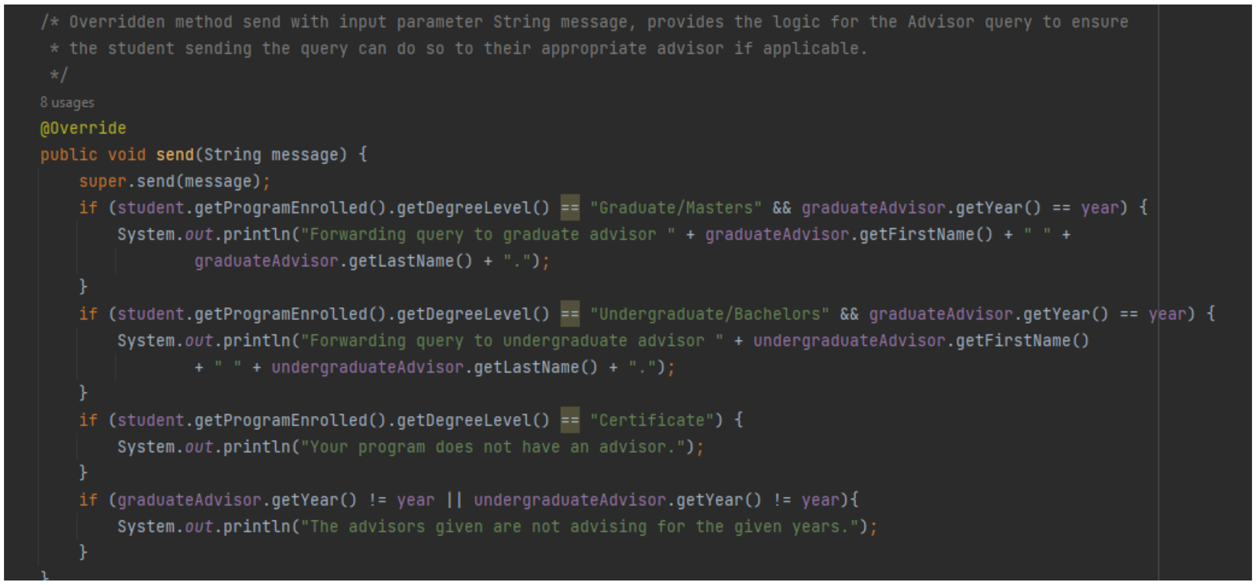


```
// overridden message send with String message input parameter, prints string student first and last name + message
8 usages
@Override
public void send(String message) {
    System.out.println("Student: " + student.getFirstName() + " " +  student.getLastName() +
            " has sent a query.\n" + message);
}
```

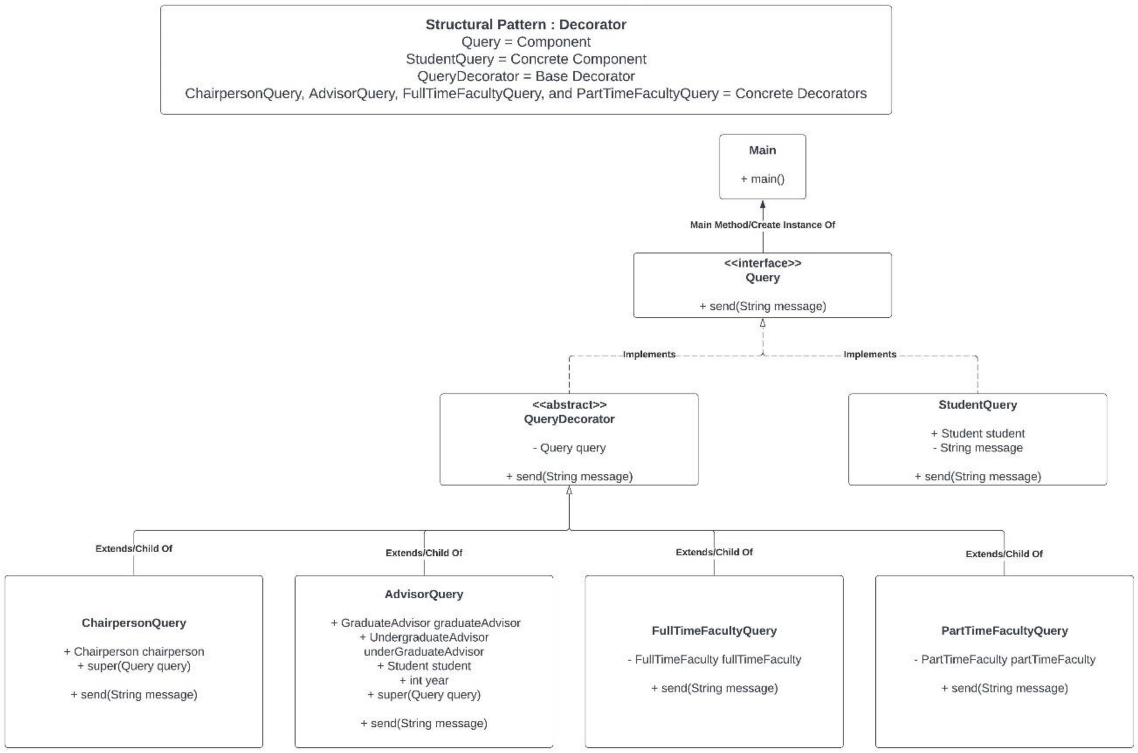


```
4 usages   4 inheritors
public abstract class QueryDecorator implements Query{
    2 usages
    private Query query;

    // constructor QueryDecorator
    4 usages
    public QueryDecorator(Query query) { this.query = query; }

    // overridden method send with input parameter String message, calls method send(message) on query
    8 usages   4 overrides
    @Override
    public void send(String message) { query.send(message); }
}
```

AdvisorQuery – Concrete Subscriber example



```java
/* Overridden method send with input parameter String message, provides the logic for the Advisor query to ensure
 * the student sending the query can do so to their appropriate advisor if applicable.
 */
8 usages
@Override
public void send(String message) {
    super.send(message);
    if (student.getProgramEnrolled().getDegreeLevel() == "Graduate/Masters" && graduateAdvisor.getYear() == year) {
        System.out.println("Forwarding query to graduate advisor " + graduateAdvisor.getFirstName() + " " +
                graduateAdvisor.getLastName() + ".");
    }
    if (student.getProgramEnrolled().getDegreeLevel() == "Undergraduate/Bachelors" && graduateAdvisor.getYear() == year) {
        System.out.println("Forwarding query to undergraduate advisor " + undergraduateAdvisor.getFirstName()
                + " " + undergraduateAdvisor.getLastName() + ".");
    }
    if (student.getProgramEnrolled().getDegreeLevel() == "Certificate") {
        System.out.println("Your program does not have an advisor.");
    }
    if (graduateAdvisor.getYear() != year || undergraduateAdvisor.getYear() != year){
        System.out.println("The advisors given are not advising for the given years.");
    }
}
```



Program Output

Program output is based on the final project description that was provided. The program can do more regarding capability than what is outputted, but the output does present the required tests to prove the program can cover all aspects of the final project description. There are additional layers of sophistication that are built into the program that I invite you to test if you would like.
** Note: some of the below output may be cut off to make the proof of concept for each test more visible. Output can be viewed by running Main.java. **

Initial Input

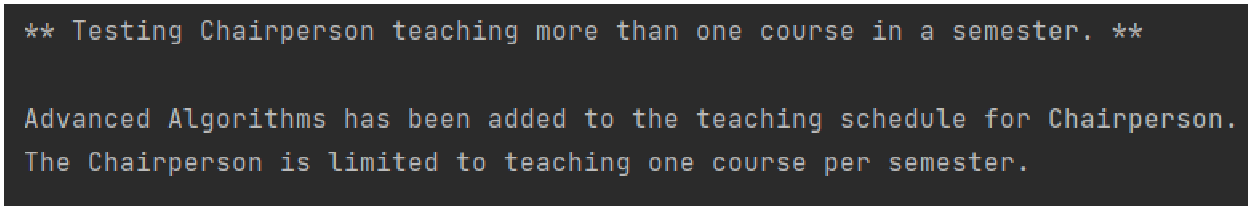
```
// Instance of Chairperson chairperson and instance of Department department
Chairperson chairperson = Chairperson.getInstance();
Department department = Department.getInstance();

// department method calls createCourses() and createConcentrations
department.createCourses();
department.createConcentrations();

// create programs (not all will be used in demonstration)
ProgramFactory factory = new ProgramFactory();
ProgramCreator store = new ProgramCreator(factory);
Program undergradCS = store.orderProgram("UndergraduateCS");
Program underGraduateCIS = store.orderProgram("UndergraduateCIS");
Program graduateCS = store.orderProgram("GraduateCS");
Program graduateDA = store.orderProgram("GraduateDA");
Program graduateCIS = store.orderProgram("GraduateCIS");
Program certificateSecurity = store.orderProgram("CertificateSecurity");
Program certificateWebTechnology =
store.orderProgram("CertificateWebTechnology");
Program certificateAnalytics = store.orderProgram("CertificateAnalytics");


// create full time faculty instances
FullTimeFaculty joeSmith = new FullTimeFaculty("Joe", "Smith");
FullTimeFaculty leahJohnson = new FullTimeFaculty("Leah", "Johnson");


// create part time faculty instances
PartTimeFaculty sandyElliot = new PartTimeFaculty("Sandy", "Elliott");
PartTimeFaculty tomPeters = new PartTimeFaculty("Tom", "Peters");

// create student instances
Student tedOrange = new Student("Ted", "Orange", graduateCS);
Student joeBlue = new Student("Joe", "Blue", graduateCIS);
Student evanBlack = new Student("Evan", "Black", graduateCIS);
```

Testing Chairperson teaching more than one course in a semester

Input:

```

System.out.println("** Testing Chairperson teaching more than one course in a
semester. **\n");
chairperson.teach(department.courseHashMap.get("CS511"));
chairperson.teach(department.courseHashMap.get("CS512"));
```

Output:



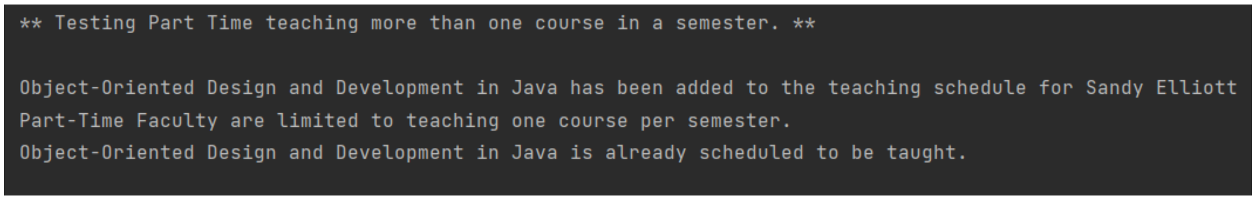
```
** Testing Chairperson teaching more than one course in a semester. **

Advanced Algorithms has been added to the teaching schedule for Chairperson.
The Chairperson is limited to teaching one course per semester.
```

Testing Part Time teaching more than one course in a semester

Input:

```
System.out.println("\n** Testing Part Time teaching more than one course in a
semester. **\n");
sandyElliot.teach(department.courseHashMap.get("CS513"));
sandyElliot.teach(department.courseHashMap.get("CS514"));
tomPeters.teach(department.courseHashMap.get("CS513"));
```

Output:



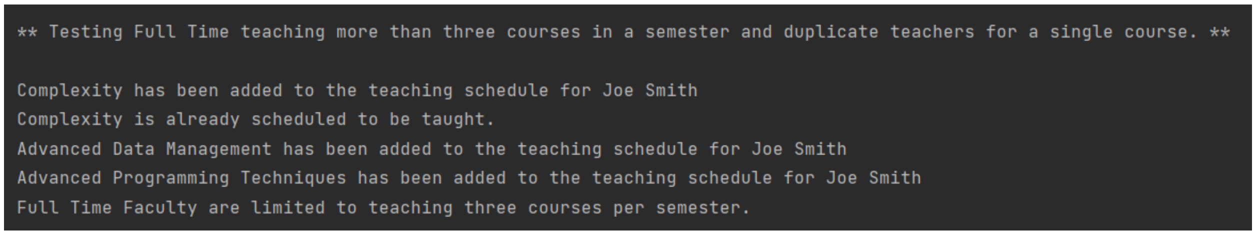
```
** Testing Part Time teaching more than one course in a semester. **

Object-Oriented Design and Development in Java has been added to the teaching schedule for Sandy Elliott
Part-Time Faculty are limited to teaching one course per semester.
Object-Oriented Design and Development in Java is already scheduled to be taught.
```

Testing Full Time teaching more than three courses in a semester and duplicate teachers for a single course

Input:

```
System.out.println("\n** Testing Full Time teaching more than three courses
in a semester and duplicate " +
        "teachers for a single course. **\n");
joeSmith.teach(department.courseHashMap.get("CS512"));
leahJohnson.teach(department.courseHashMap.get("CS512"));
joeSmith.teach(department.courseHashMap.get("CS516"));
joeSmith.teach(department.courseHashMap.get("CS517"));
joeSmith.teach(department.courseHashMap.get("CS521"));
```

Output:



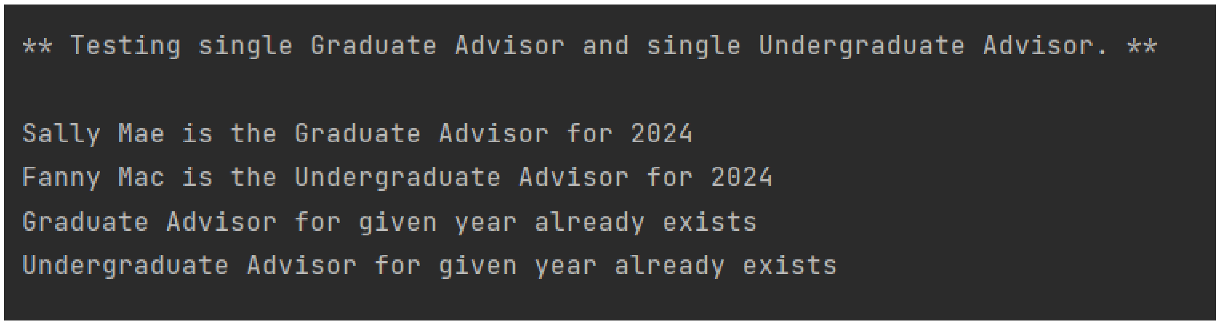
```
** Testing Full Time teaching more than three courses in a semester and duplicate teachers for a single course. **

Complexity has been added to the teaching schedule for Joe Smith
Complexity is already scheduled to be taught.
Advanced Data Management has been added to the teaching schedule for Joe Smith
Advanced Programming Techniques has been added to the teaching schedule for Joe Smith
Full Time Faculty are limited to teaching three courses per semester.
```

Testing single Graduate Advisor and single Undergraduate Advisor

Input:

```
System.out.println("\n** Testing single Graduate Advisor and single
Undergraduate Advisor. **\n");
GraduateAdvisor sallyMae = new GraduateAdvisor("Sally", "Mae", 2024);
UndergraduateAdvisor fannieMac = new UndergraduateAdvisor("Fanny", "Mac",
2024);
GraduateAdvisor duplicateForYearGraduate = new
GraduateAdvisor("DuplicateGraduate", "ForYear", 2024);
UndergraduateAdvisor duplicateForYearUndergraduate = new
UndergraduateAdvisor("DuplicateUndergraduate", "ForYear", 2024);
sallyMae.checkGraduateAdvisorYear();
fannieMac.checkUndergraduateAdvisorYear();
duplicateForYearGraduate.checkGraduateAdvisorYear();
duplicateForYearUndergraduate.checkUndergraduateAdvisorYear();
```

Output:



```
** Testing single Graduate Advisor and single Undergraduate Advisor. **

Sally Mae is the Graduate Advisor for 2024
Fanny Mac is the Undergraduate Advisor for 2024
Graduate Advisor for given year already exists
Undergraduate Advisor for given year already exists
```

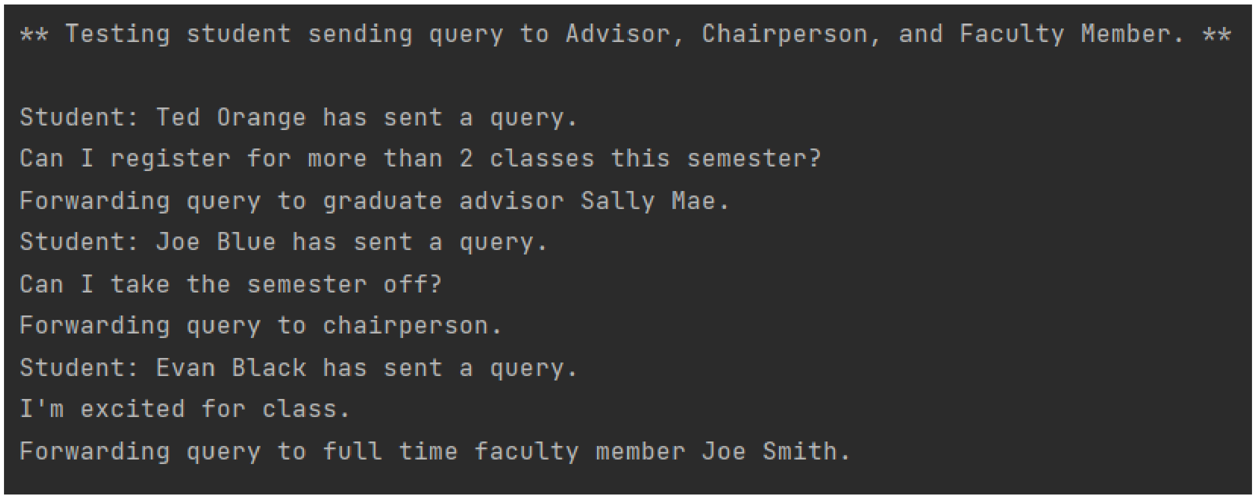
Testing student sending query to Advisor, Chairperson, and Faculty Member

Input:

```
System.out.println("\n** Testing student sending query to Advisor,
Chairperson, and Faculty Member. **\n");
// advisor query
Query tedOrangeQuery = new AdvisorQuery(new StudentQuery(tedOrange, "Can I
register for " +
        " more than 2 classes this semester?"), tedOrange, 2024, sallyMae,
fannieMac);
tedOrangeQuery.send("Can I register for more than 2 classes this semester?");
```

```
// chairperson query
Query joeBlueQuery = new ChairpersonQuery(new StudentQuery(joeBlue, "Can I
take the semester off?")
        , chairperson);
joeBlueQuery.send("Can I take the semester off?");

// full time faculty query
Query evanBlackQuery = new FullTimeFacultyQuery(new StudentQuery(evanBlack,
"I'm excited for class."),
        joeSmith);
evanBlackQuery.send("I'm excited for class.");
```

Output:



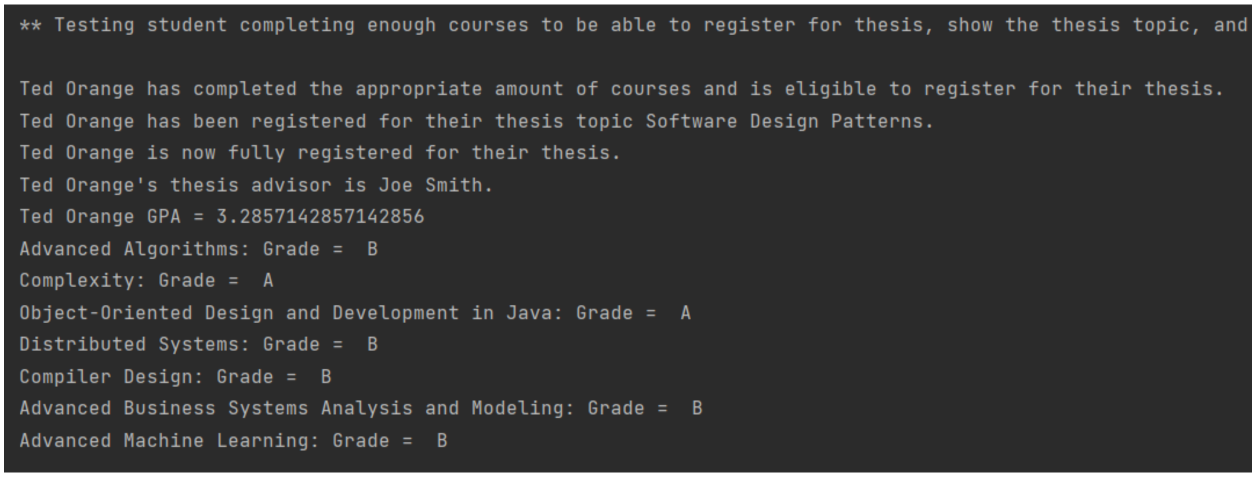
```
** Testing student sending query to Advisor, Chairperson, and Faculty Member. **

Student: Ted Orange has sent a query.
Can I register for more than 2 classes this semester?
Forwarding query to graduate advisor Sally Mae.
Student: Joe Blue has sent a query.
Can I take the semester off?
Forwarding query to chairperson.
Student: Evan Black has sent a query.
I'm excited for class.
Forwarding query to full time faculty member Joe Smith.
```

Testing student completing enough courses to be able to register for thesis, show the thesis topic, and the thesis advisor. Calculate the student's GPA and show the completed courses and their grades.

Input:

```
System.out.println("\n** Testing student completing enough courses to be able
to register for thesis, show " +
        "the thesis topic, and the thesis advisor. Calculate the student's
GPA and show the completed courses "
        + "and their grades.\n");

tedOrange.completedCourse(department.courseHashMap.get("CS511"), 3);
tedOrange.completedCourse(department.courseHashMap.get("CS512"), 4);
tedOrange.completedCourse(department.courseHashMap.get("CS513"), 4);
tedOrange.completedCourse(department.courseHashMap.get("CS514"), 3);
tedOrange.completedCourse(department.courseHashMap.get("CS515"), 3);
tedOrange.completedCourse(department.courseHashMap.get("CS520"), 3);
tedOrange.completedCourse(department.courseHashMap.get("CS525"), 3);
Thesis tedOrangeThesis = new GraduateThesis();
tedOrangeThesis.registerForThesis("Software Design Patterns", tedOrange,
joeSmith);
tedOrange.gpa();
tedOrange.viewCompletedCourses();
```

Output:



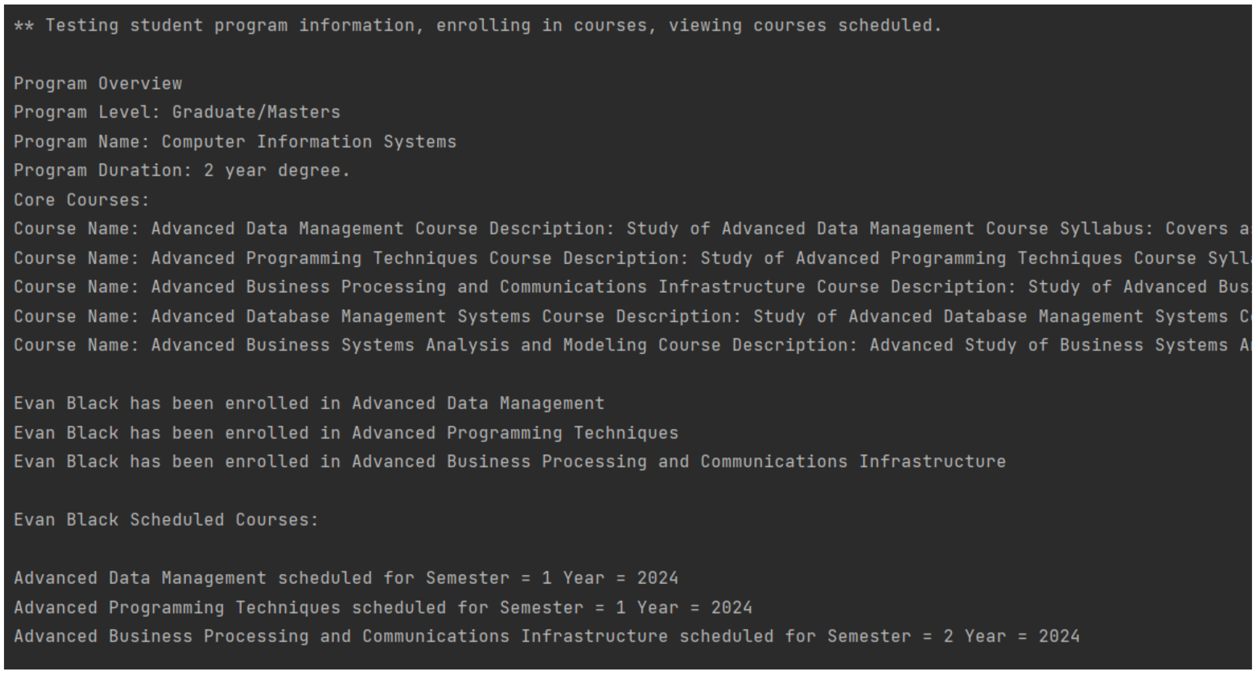
```
** Testing student completing enough courses to be able to register for thesis, show the thesis topic, and

Ted Orange has completed the appropriate amount of courses and is eligible to register for their thesis.
Ted Orange has been registered for their thesis topic Software Design Patterns.
Ted Orange is now fully registered for their thesis.
Ted Orange's thesis advisor is Joe Smith.
Ted Orange GPA = 3.2857142857142856
Advanced Algorithms: Grade =  B
Complexity: Grade =  A
Object-Oriented Design and Development in Java: Grade =  A
Distributed Systems: Grade =  B
Compiler Design: Grade =  B
Advanced Business Systems Analysis and Modeling: Grade =  B
Advanced Machine Learning: Grade =  B
```

Testing student program information, enrolling in courses, viewing courses scheduled.

Input:

```
System.out.println("\n** Testing student program information, enrolling in
courses, viewing courses scheduled.");
System.out.println(evanBlack.getProgramEnrolled());
evanBlack.enrollInCourse(department.courseHashMap.get("CS516"));
evanBlack.enrollInCourse(department.courseHashMap.get("CS517"));
evanBlack.enrollInCourse(department.courseHashMap.get("CS518"));
evanBlack.viewCoursesScheduled();
```

Output:



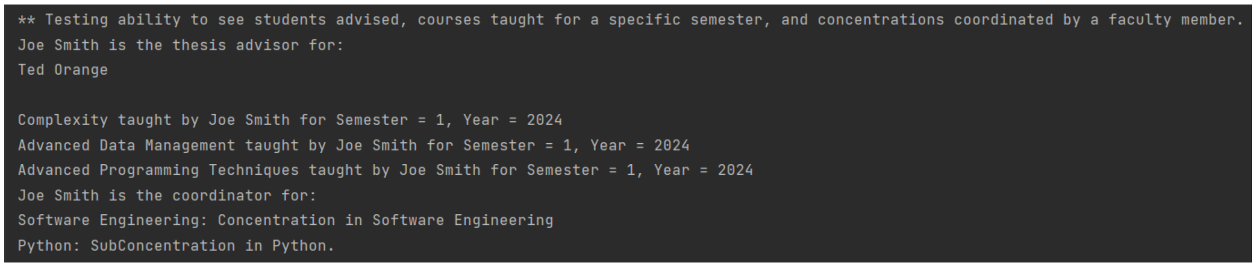
```
** Testing student program information, enrolling in courses, viewing courses scheduled.

Program Overview
Program Level: Graduate/Masters
Program Name: Computer Information Systems
Program Duration: 2 year degree.
Core Courses:
Course Name: Advanced Data Management Course Description: Study of Advanced Data Management Course Syllabus: Covers a
Course Name: Advanced Programming Techniques Course Description: Study of Advanced Programming Techniques Course Syll
Course Name: Advanced Business Processing and Communications Infrastructure Course Description: Study of Advanced Bus
Course Name: Advanced Database Management Systems Course Description: Study of Advanced Database Management Systems C
Course Name: Advanced Business Systems Analysis and Modeling Course Description: Advanced Study of Business Systems A

Evan Black has been enrolled in Advanced Data Management
Evan Black has been enrolled in Advanced Programming Techniques
Evan Black has been enrolled in Advanced Business Processing and Communications Infrastructure

Evan Black Scheduled Courses:

Advanced Data Management scheduled for Semester = 1 Year = 2024
Advanced Programming Techniques scheduled for Semester = 1 Year = 2024
Advanced Business Processing and Communications Infrastructure scheduled for Semester = 2 Year = 2024
```

Testing ability to see students advised, courses taught for a specific semester, and concentrations coordinated by a faculty member.

Input:

```java
System.out.println("\n** Testing ability to see students advised, courses
taught for a specific semester, " +
        "and concentrations coordinated by a faculty member.");
joeSmith.getStudentsAdvisedThesis();
System.out.println();
joeSmith.getCoursesTaughtForSemesterYear(2024, 1);
department.softwareEngineeringConcentration.setCoordinator(joeSmith);
department.pythonSubConcentration.setCoordinator(joeSmith);
joeSmith.getCoordinatedConcentrations();
```

Output:



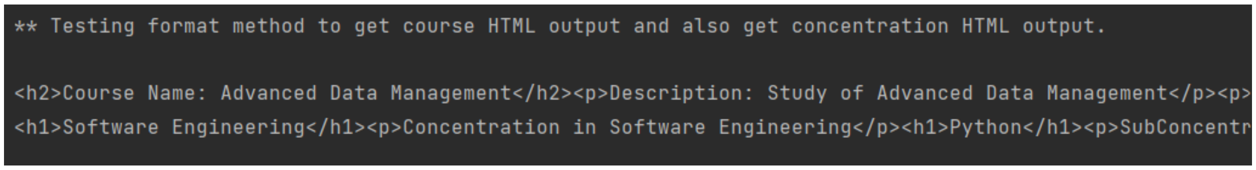
```
** Testing ability to see students advised, courses taught for a specific semester, and concentrations coordinated by a faculty member.
Joe Smith is the thesis advisor for:
Ted Orange

Complexity taught by Joe Smith for Semester = 1, Year = 2024
Advanced Data Management taught by Joe Smith for Semester = 1, Year = 2024
Advanced Programming Techniques taught by Joe Smith for Semester = 1, Year = 2024
Joe Smith is the coordinator for:
Software Engineering: Concentration in Software Engineering
Python: SubConcentration in Python.
```

Testing format method to get course HTML output and also get concentration HTML output.

Input:

```
System.out.println("\n** Testing format method to get course HTML output and
also get concentration HTML " +
        "output.\n");
// testing course
System.out.println(department.courseHashMap.get("CS516").format());
// testing concentration w/ subconcentration
System.out.println(department.softwareEngineeringConcentration.format());
```

Output:



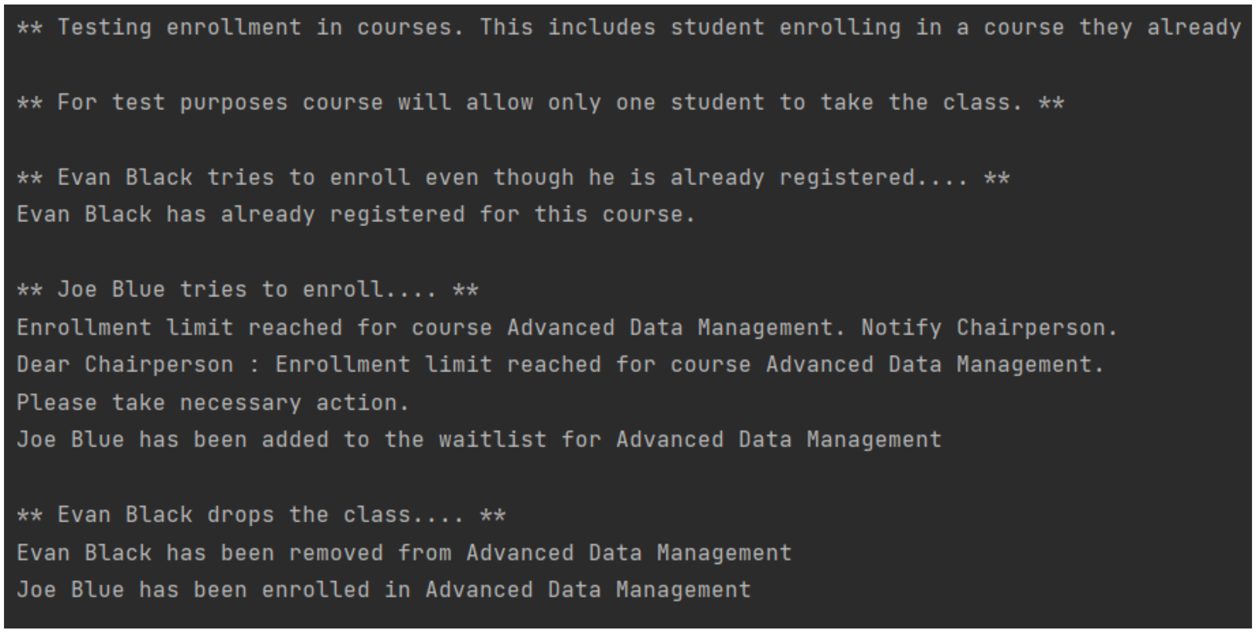
```
** Testing format method to get course HTML output and also get concentration HTML output.

<h2>Course Name: Advanced Data Management</h2><p>Description: Study of Advanced Data Management</p><p>
<h1>Software Engineering</h1><p>Concentration in Software Engineering</p><h1>Python</h1><p>SubConcentr
```

Testing enrollment in courses. This includes student enrolling in a course they already have registered for, student being wait listed, student dropping course, and wait listed student being added. This will also include the notifications to the chairperson and students, as well as auto enrollment.

Input:

```
System.out.println("\n** Testing enrollment in courses. This includes student
enrolling in a course they " +
        "already have registered for, student being wait listed, student
dropping course, and wait listed "
        + "student being added. This will also include the notifications to
the chairperson and students, " +
        "as well as auto enrollment.\n");

System.out.println("** For test purposes course will allow only one student
to take the class. **");
System.out.println("\n** Evan Black tries to enroll even though he is already
registered.... **");
// student registering for class they already registered for
evanBlack.enrollInCourse(department.courseHashMap.get("CS516"));
System.out.println("\n** Joe Blue tries to enroll.... **");
joeBlue.enrollInCourse(department.courseHashMap.get("CS516"));
System.out.println("\n** Evan Black drops the class.... **");
evanBlack.dropCourse(department.courseHashMap.get("CS516"));
```

Output:



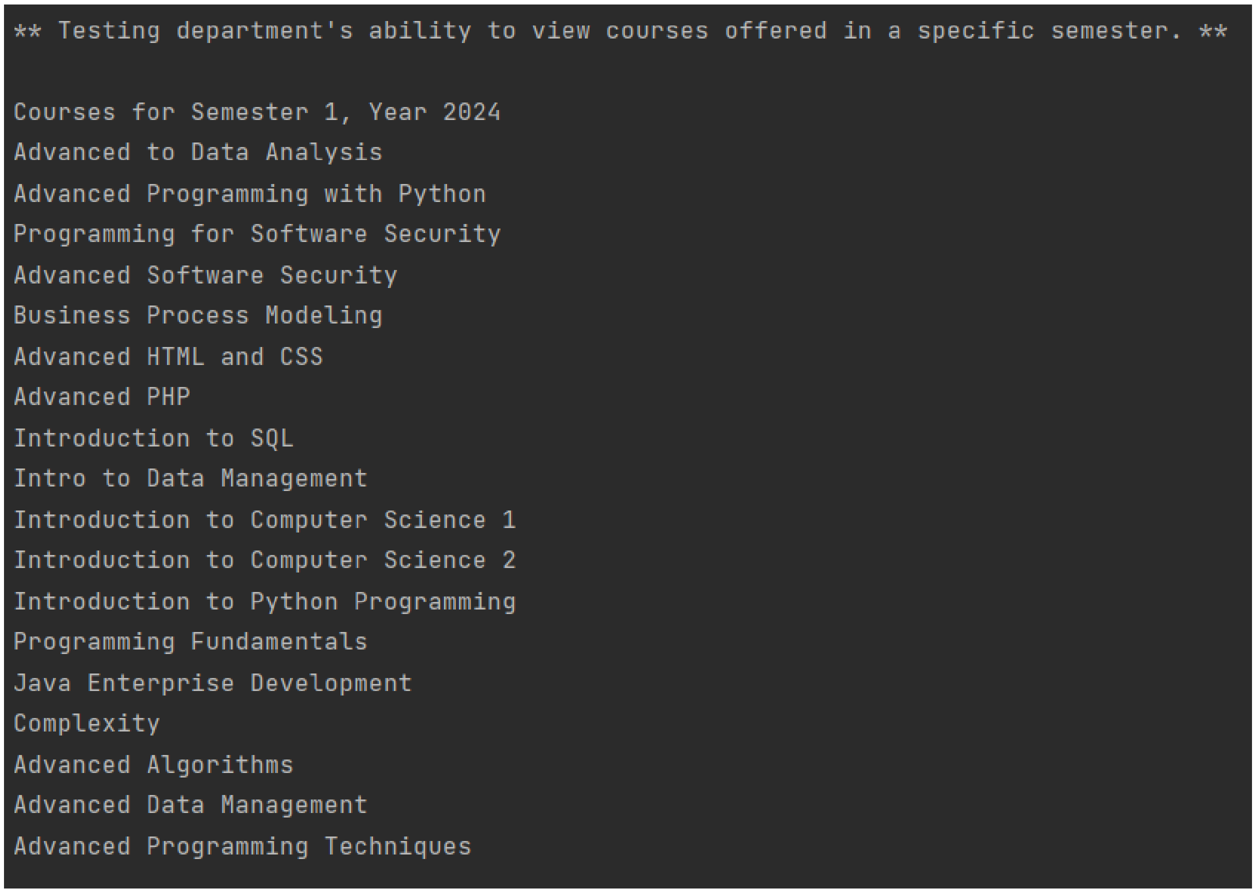
```
** Testing enrollment in courses. This includes student enrolling in a course they already

** For test purposes course will allow only one student to take the class. **

** Evan Black tries to enroll even though he is already registered.... **
Evan Black has already registered for this course.

** Joe Blue tries to enroll.... **
Enrollment limit reached for course Advanced Data Management. Notify Chairperson.
Dear Chairperson : Enrollment limit reached for course Advanced Data Management.
Please take necessary action.
Joe Blue has been added to the waitlist for Advanced Data Management

** Evan Black drops the class.... **
Evan Black has been removed from Advanced Data Management
Joe Blue has been enrolled in Advanced Data Management
```

Testing department's ability to view courses offered in a specific semester.

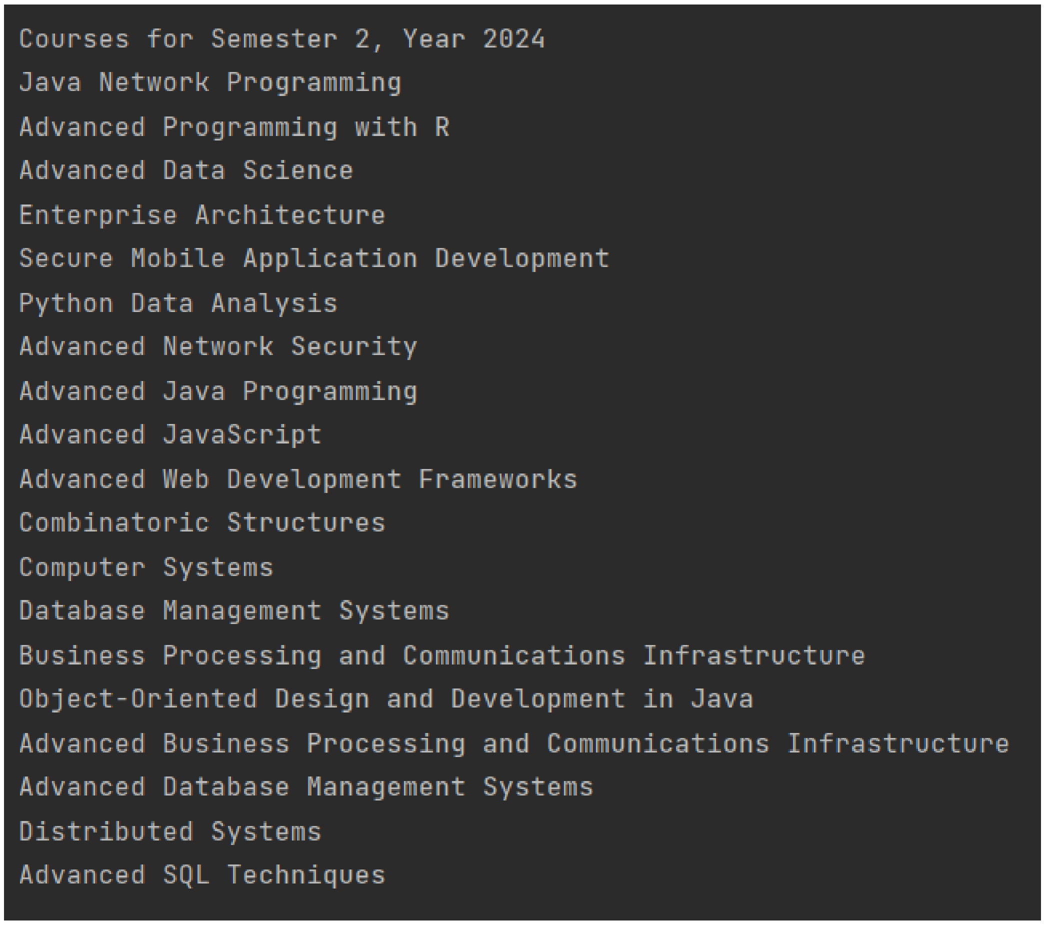
Input:

```
System.out.println("\n** Testing department's ability to view courses offered
in a specific semester. **\n");
department.getCoursesBySemesterAndYear(2024, 1);
System.out.println("\n");
department.getCoursesBySemesterAndYear(2024, 2);
System.out.println("\n");
department.getCoursesBySemesterAndYear(2025, 2);
```

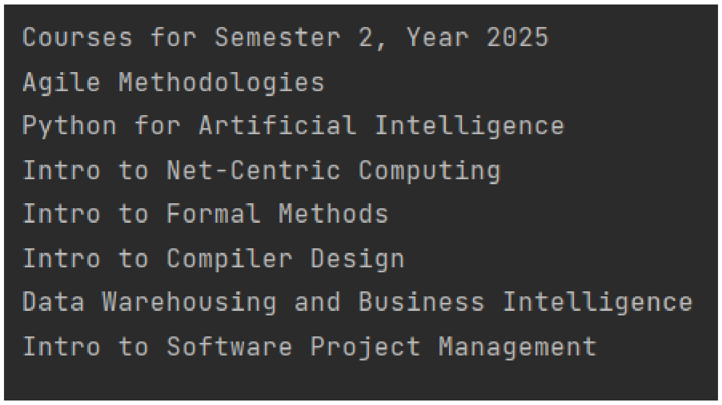
Output:



```
** Testing department's ability to view courses offered in a specific semester. **

Courses for Semester 1, Year 2024
Advanced to Data Analysis
Advanced Programming with Python
Programming for Software Security
Advanced Software Security
Business Process Modeling
Advanced HTML and CSS
Advanced PHP
Introduction to SQL
Intro to Data Management
Introduction to Computer Science 1
Introduction to Computer Science 2
Introduction to Python Programming
Programming Fundamentals
Java Enterprise Development
Complexity
Advanced Algorithms
Advanced Data Management
Advanced Programming Techniques
```



```
Courses for Semester 2, Year 2024
Java Network Programming
Advanced Programming with R
Advanced Data Science
Enterprise Architecture
Secure Mobile Application Development
Python Data Analysis
Advanced Network Security
Advanced Java Programming
Advanced JavaScript
Advanced Web Development Frameworks
Combinatoric Structures
Computer Systems
Database Management Systems
Business Processing and Communications Infrastructure
Object-Oriented Design and Development in Java
Advanced Business Processing and Communications Infrastructure
Advanced Database Management Systems
Distributed Systems
Advanced SQL Techniques
```



```
Courses for Semester 2, Year 2025
Agile Methodologies
Python for Artificial Intelligence
Intro to Net-Centric Computing
Intro to Formal Methods
Intro to Compiler Design
Data Warehousing and Business Intelligence
Intro to Software Project Management
```