cs665 Final Project

Author: Yuqi Lin

Class Overview

Here's a brief overview of the key classes and their functionalities:

Department Class

It manages programs, courses, students, and faculties(one chairperson) within the department.

Key Methods:

- `getCourseBySemester()`: Retrieves courses offered by the department, organized by semester.
- `getStudentBySemester()`: Retrieves students enrolled in the department, organized by semester.
- `addFaculty()`: Adds a faculty member to the department.
- `addCourse()`: Adds a course to the department's offerings for a specific semester.
- `addStudent()`: Enrolls a student in the department, tracking them by semester.
- `setChairperson()`: Assigns a chairperson to oversee departmental activities.

Program

It is an abstract base class representing an academic program, such as a degree or certificate.

`Degree` and `Certificate` classes extend `Program`, each representing specific types of academic programs.

Concentration

It represents a specific area of study within a program. It can contain other sub-concentrations or courses, forming a hierarchical structure.

Key Methods:

- `add()`: Adds a sub-concentration or course to the concentration.
- `remove()`: Removes a sub-concentration or course from the concentration.
- `format()`: Returns a structured representation of the concentration and its components.

Course

It represents an academic course, including details like its syllabus, faculty, and enrollment.

Key Methods:

- `getName()`: Returns the name of the course.
- `getSemester()`: Retrieves the semester in which the course is offered.`enrollStudent()`: Enrolls a student in the course, or places them on a waitlist if the course is full.
- `dropStudent()`: Removes a student from the course and manages waitlist enrollments.

Faculty

It is an abstract class representing a faculty member responsible for teaching and advising students.

Key Methods:

- `getName()`: Returns the name of the faculty member.
- `addCourse()`: Assigns a course to the faculty member.
- `getCoursesBySemester()`: Retrieves courses taught by the faculty member in a given semester.

Inheritance:

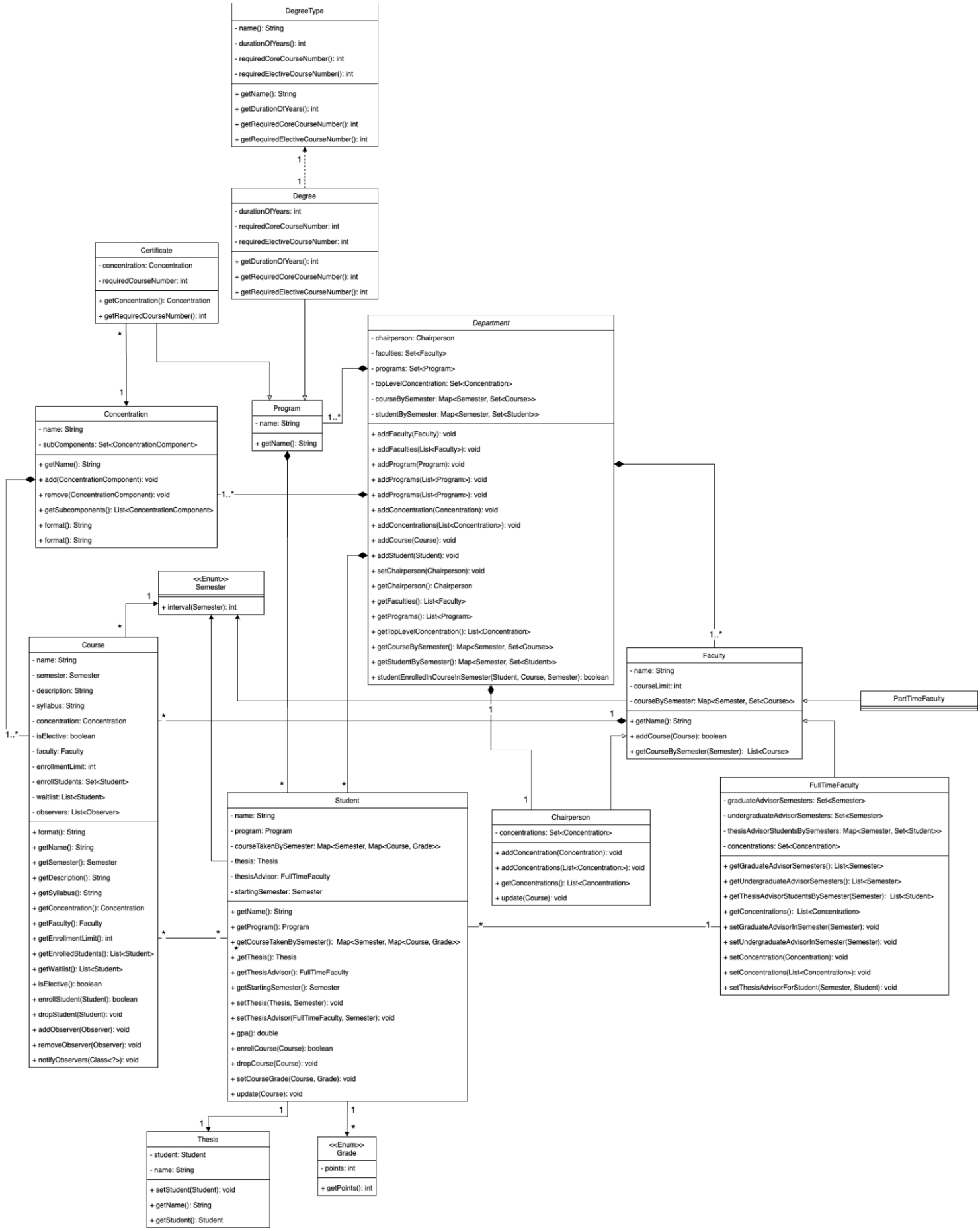
- `Chairperson`, `FullTimeFaculty`, and `PartTimeFaculty` classes extend `Faculty`, each representing different faculty roles.

Student

It represents a student enrolled in a program, managing their course enrollments, thesis, and academic progress.

Key Methods:

- `gpa()`: Calculates and returns the student's GPA.
- `getName()`: Returns the name of the student.`enrollCourse()`: Enrolls the student in a course after validating eligibility.
- `dropCourse()`: Allows the student to drop a course they are enrolled in.
- `setThesis()`: Assigns a thesis to the student if they meet the requirements.


DegreeType
- name(): String
- durationOfYears(): int
- requiredCoreCourseNumber(): int
- requiredElectiveCourseNumber(): int
+ getName(): String
+ getDurationOfYears(): int
+ getRequiredCoreCourseNumber(): int
+ getRequiredElectiveCourseNumber(): int
Degree
- durationOfYears: int
- requiredCoreCourseNumber: int
- requiredElectiveCourseNumber: int
+ getDurationOfYears(): int
+ getRequiredCoreCourseNumber(): int
+ getRequiredElectiveCourseNumber(): int
Certificate
- concentration: Concentration
- requiredCourseNumber: int
+ getConcentration(): Concentration
+ getRequiredCourseNumber(): int
Department
- chairperson: Chairperson
- faculties: Set<Faculty>
- programs: Set<Program>
- topLevelConcentration: Set<Concentration>
- courseBySemester: Map<Semester, Set<Course>>
- studentBySemester: Map<Semester, Set<Student>>
+ addFaculty(Faculty): void
+ addFaculties(List<Faculty>): void
+ addProgram(Program): void
+ addPrograms(List<Program>): void
+ addPrograms(List<Program>): void
+ addConcentration(Concentration): void
+ addConcentrations(List<Concentration>): void
+ addCourse(Course): void
+ addStudent(Student): void
+ setChairperson(Chairperson): void
+ getChairperson(): Chairperson
+ getFaculties(): List<Faculty>
+ getPrograms(): List<Program>
+ getTopLevelConcentration(): List<Concentration>
+ getCourseBySemester(): Map<Semester, Set<Course>>
+ getStudentBySemester(): Map<Semester, Set<Student>>
+ studentEnrolledInCourseInSemester(Student, Course, Semester): boolean
Concentration
- name: String
- subComponents: Set<ConcentrationComponent>
+ getName(): String
+ add(ConcentrationComponent): void
+ remove(ConcentrationComponent): void
+ getSubcomponents(): List<ConcentrationComponent>
+ format(): String
+ format(): String
Program
- name: String
+ getName(): String
<<Enum>>
Semester
+ interval(Semester): int
Course
- name: String
- semester: Semester
- description: String
- syllabus: String
- concentration: Concentration
- isElective: boolean
- faculty: Faculty
- enrollmentLimit: int
- enrollStudents: Set<Student>
- waitlist: List<Student>
- observers: List<Observer>
+ format(): String
+ getName(): String
+ getSemester(): Semester
+ getDescription(): String
+ getSyllabus(): String
+ getConcentration(): Concentration
+ getFaculty(): Faculty
+ getEnrollmentLimit(): int
+ getEnrolledStudents(): List<Student>
+ getWaitlist(): List<Student>
+ isElective(): boolean
+ enrollStudent(Student): boolean
+ dropStudent(Student): void
+ addObserver(Observer): void
+ removeObserver(Observer): void
+ notifyObservers(Class<?>): void
Faculty
- name: String
- courseLimit: int
- courseBySemester: Map<Semester, Set<Course>>
+ getName(): String
+ addCourse(Course): boolean
+ getCourseBySemester(Semester): List<Course>
PartTimeFaculty
Student
- name: String
- program: Program
- courseTakenBySemester: Map<Semester, Map<Course, Grade>>
- thesis: Thesis
- thesisAdvisor: FullTimeFaculty
- startingSemester: Semester
+ getName(): String
+ getProgram(): Program
+ getCourseTakenBySemester(): Map<Semester, Map<Course, Grade>>
+ getThesis(): Thesis
+ getThesisAdvisor(): FullTimeFaculty
+ getStartingSemester(): Semester
+ setThesis(Thesis, Semester): void
+ setThesisAdvisor(FullTimeFaculty, Semester): void
+ gpa(): double
+ enrollCourse(Course): boolean
+ dropCourse(Course): void
+ setCourseGrade(Course, Grade): void
+ update(Course): void
Chairperson
- concentrations: Set<Concentration>
+ addConcentration(Concentration): void
+ addConcentrations(List<Concentration>): void
+ getConcentrations(): List<Concentration>
+ update(Course): void
FullTimeFaculty
- graduateAdvisorSemesters: Set<Semester>
- undergraduateAdvisorSemesters: Set<Semester>
- thesisAdvisorStudentsBySemesters: Map<Semester, Set<Student>>
- concentrations: Set<Concentration>
+ getGraduateAdvisorSemesters(): List<Semester>
+ getUndergraduateAdvisorSemesters(): List<Semester>
+ getThesisAdvisorStudentsBySemester(Semester): List<Student>
+ getConcentrations(): List<Concentration>
+ setGraduateAdvisorInSemester(Semester): void
+ setUndergraduateAdvisorInSemester(Semester): void
+ setConcentration(Concentration): void
+ setConcentrations(List<Concentration>): void
+ setThesisAdvisorForStudent(Semester, Student): void
Thesis
- student: Student
- name: String
+ setStudent(Student): void
+ getName(): String
+ getStudent(): Student
<<Enum>>
Grade
- points: int
+ getPoints(): int

Design Patterns Used

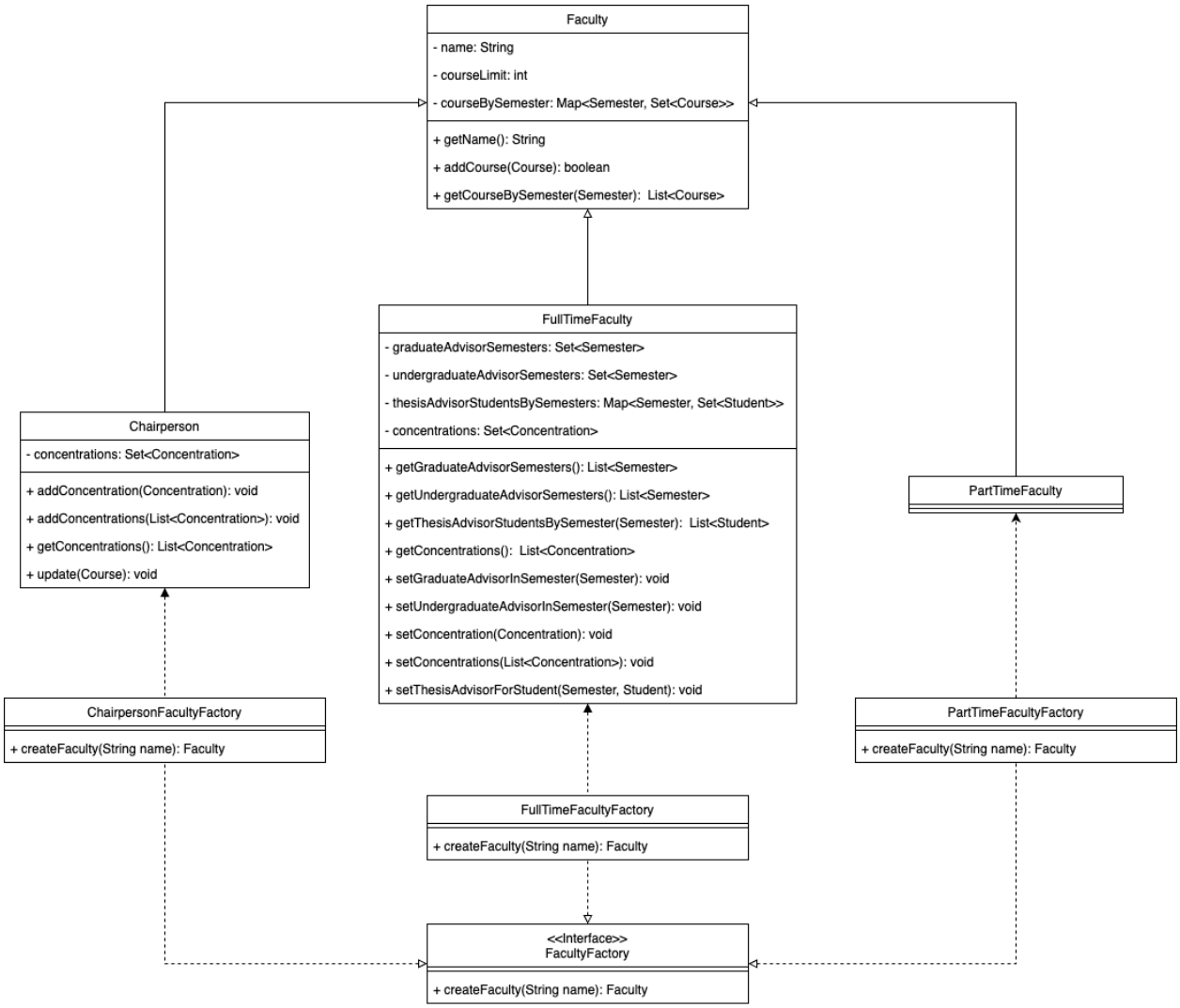
Factory Method Pattern (Faculty)

The Factory Method Pattern is used in the creation of `Faculty` objects, allowing for the creation of different types of faculty members (`Chairperson`, `FullTimeFaculty`, `PartTimeFaculty`).

Class Interaction:

- `FacultyFactory`: An interface that defines a method for creating `Faculty` objects.
- `ChairpersonFactory`, `FullTimeFacultyFactory`, `PartTimeFacultyFactory`: Concrete factory classes that implement the `FacultyFactory` interface to create specific types of `Faculty` objects.

This pattern simplifies the process of adding new types of faculty members. For instance, if a new faculty type is introduced, only a new factory class needs to be implemented without changing the core logic that relies on the `Faculty` interface.



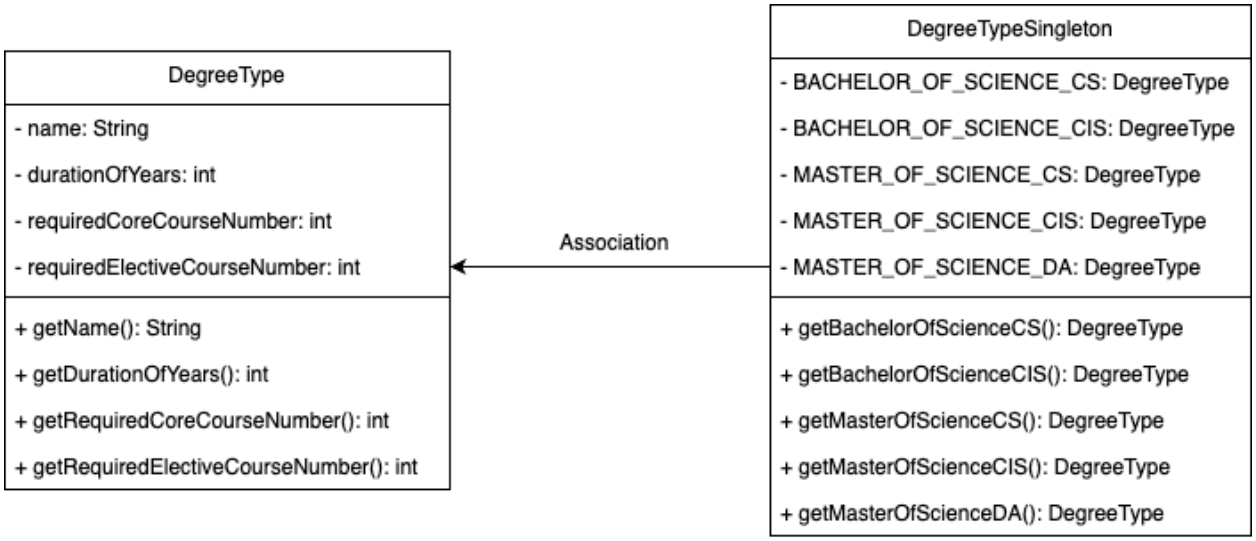
Singleton Pattern (DegreeType)

The Singleton Pattern is applied to the `DegreeType` class, ensuring that there is only one instance of each degree type (e.g., `Bachelor of Science in CS, Master of Science in CIS`) across the application.

Class Interaction:

- `DegreeTypeSingleton`: Provides access to the single instances of `DegreeType` objects.
- `Degree`: Uses `DegreeType` instances to define the duration and course requirements of specific degrees.

By using the Singleton Pattern, the system ensures that there is only one shared instance of each `DegreeType`, reducing memory usage and ensuring consistency across the application.



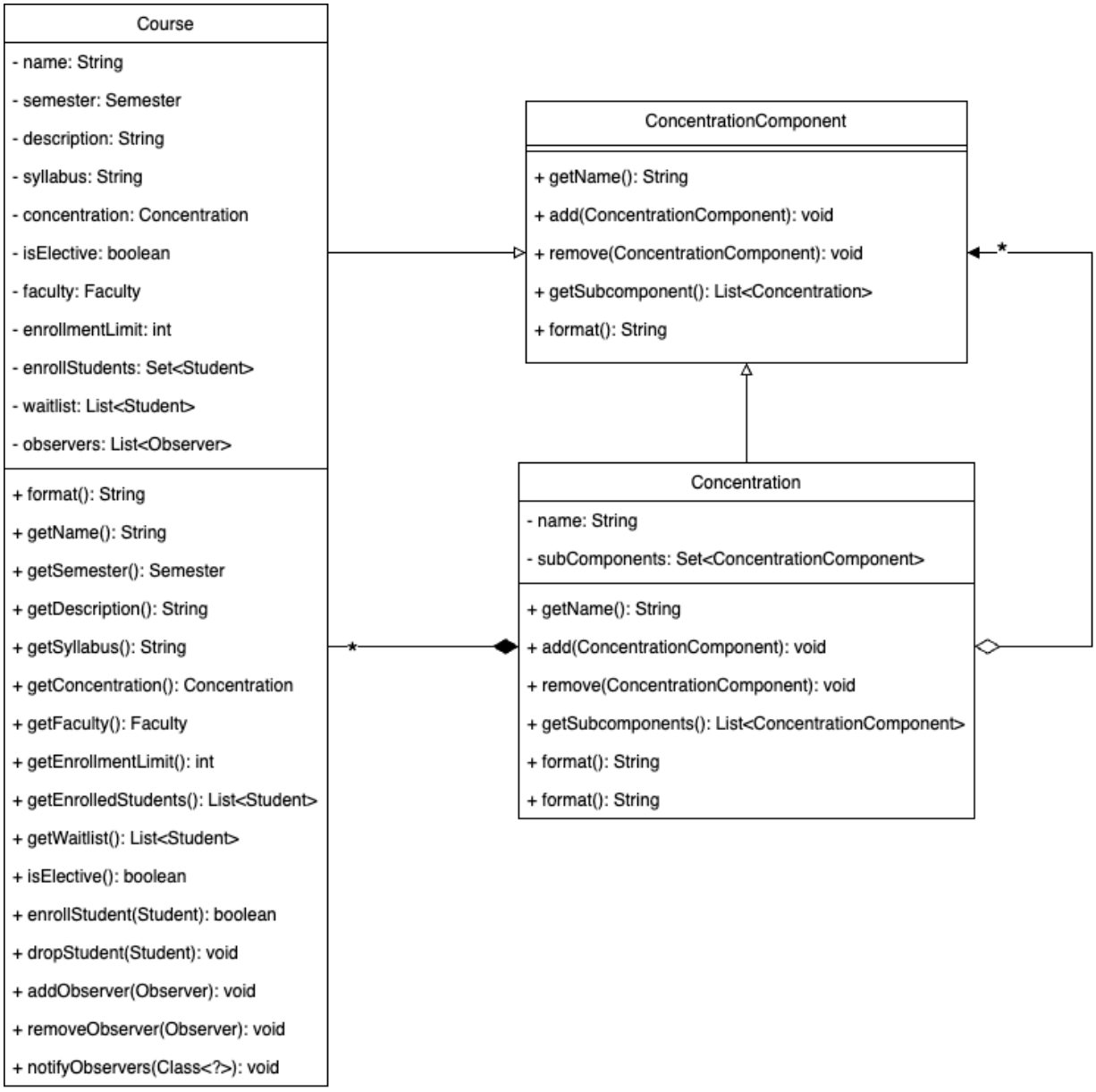
Composite Pattern (Concentration and Course)

The Composite Pattern is utilized to manage the hierarchical structure of `Concentration` objects, where a concentration can contain multiple sub-concentrations or courses.

Class Interaction:

- `ConcentrationComponent`: An abstract class representing both individual courses and concentrations.
- `Concentration`: A composite class that can hold a collection of `ConcentrationComponent` objects (including other concentrations or courses).
- `Course`: Represents a leaf node in the composite structure, which cannot have sub-components.

This pattern allows the system to treat both individual courses and concentrations uniformly, simplifying the process of managing concentrations with complex structures.



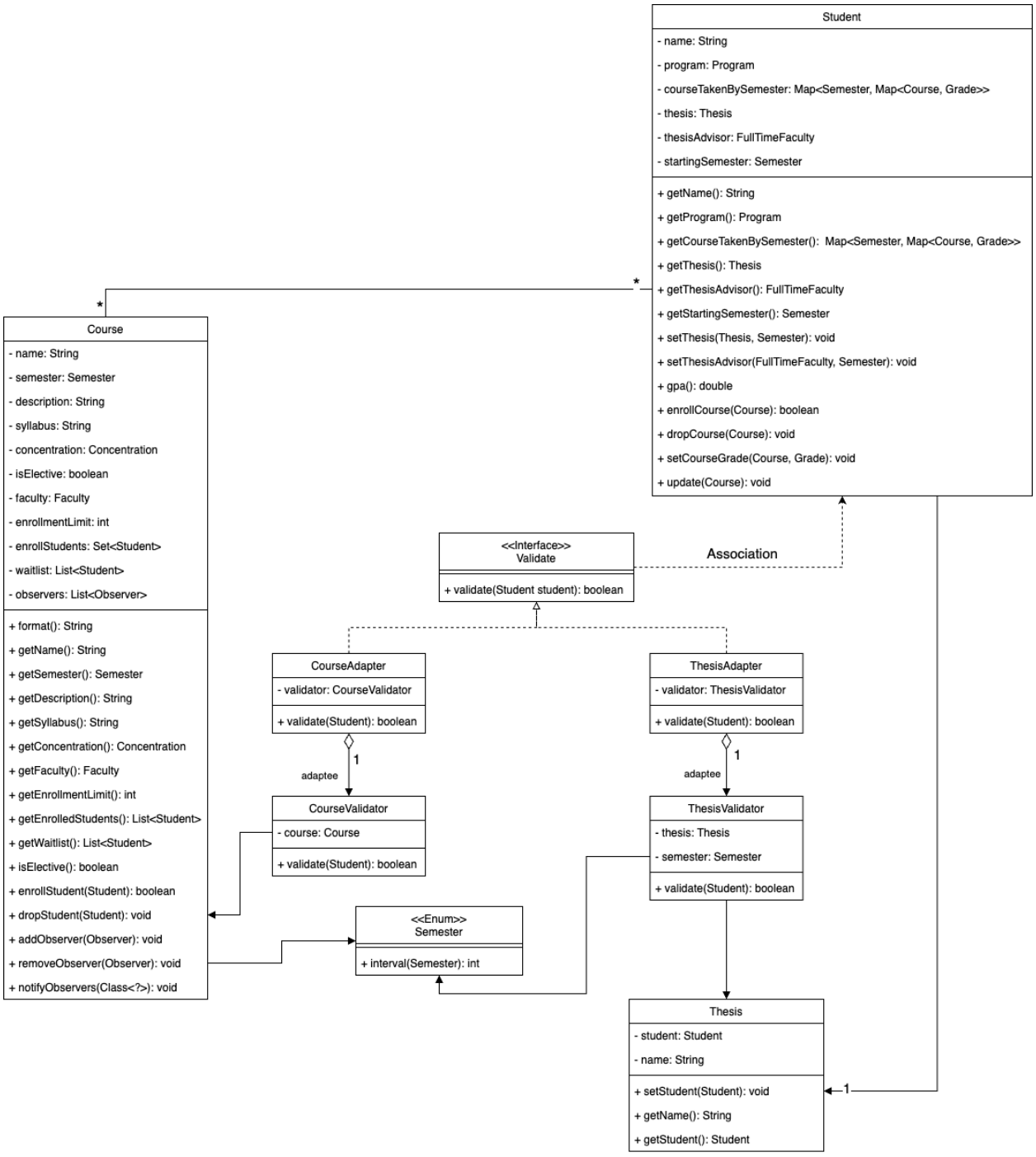
Adapter Pattern (xxAdapter and xxValidator)

The Adapter Pattern is used to allow different validation strategies to be applied to students based on their courses or thesis requirements.

Class Interaction:

- `Validate`: An interface that defines a method for validating students.
- `CourseAdapter` and `ThesisAdapter`: Adapters that allow different validation implementations (`CourseValidator` and `ThesisValidator`) to be used interchangeably.

This pattern provides flexibility in validation logic, enabling the system to easily apply different validation rules without changing the core logic of the `Student` class.



Command Pattern

RemoteControl

The `RemoteControl` class acts as an invoker in the Command pattern. It is responsible for storing a command and triggering its execution.

- `setCommand()`: Sets the command that the remote control will execute.
- `execute()`: Executes the stored command.

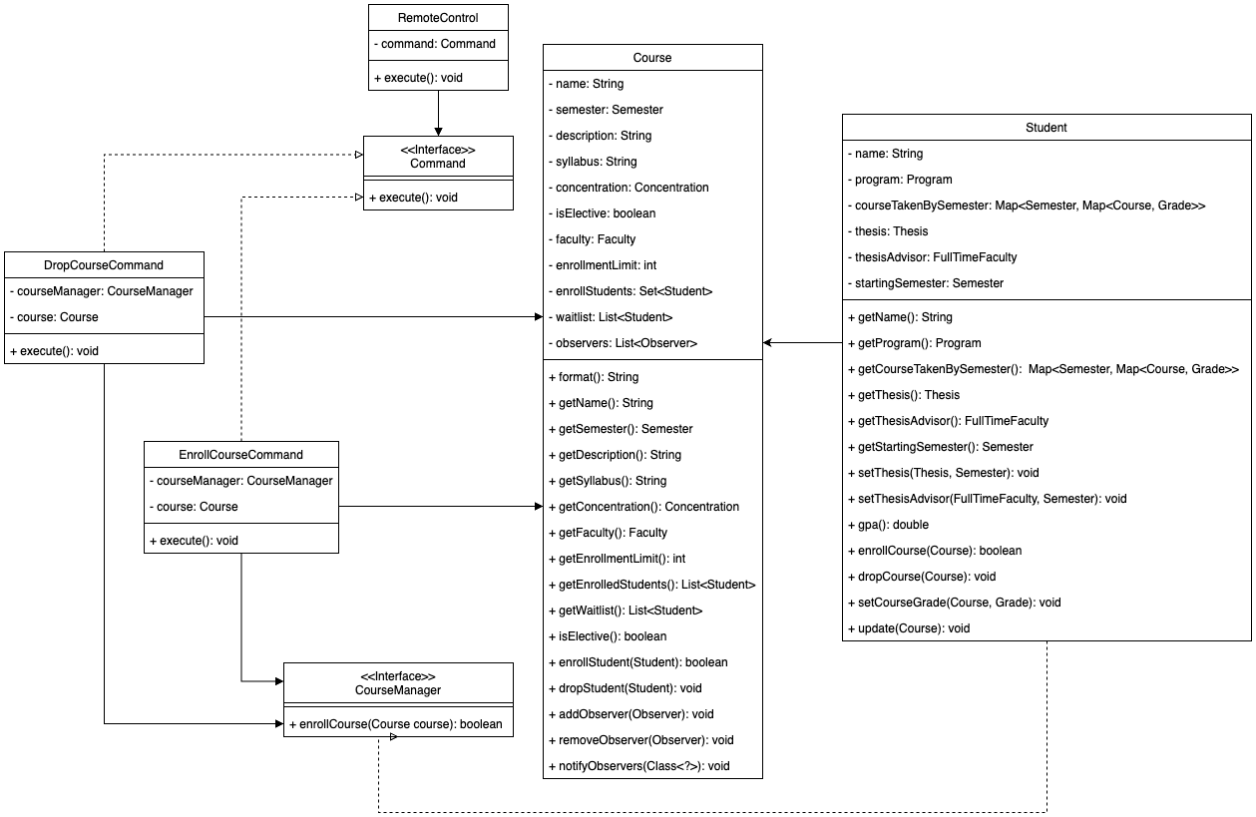
CourseManager

The `CourseManager` interface defines methods for managing course enrollments and drops, providing a layer of abstraction for these operations.

- `enrollCourse()`: Enrolls a course using the command pattern.
- `dropCourse()`: Drops a course using the command pattern.

Student Class (as part of Command Pattern)

In the context of the Command pattern, the `Student` class implements the `CourseManager` interface, allowing students to enroll in and drop courses through command execution.

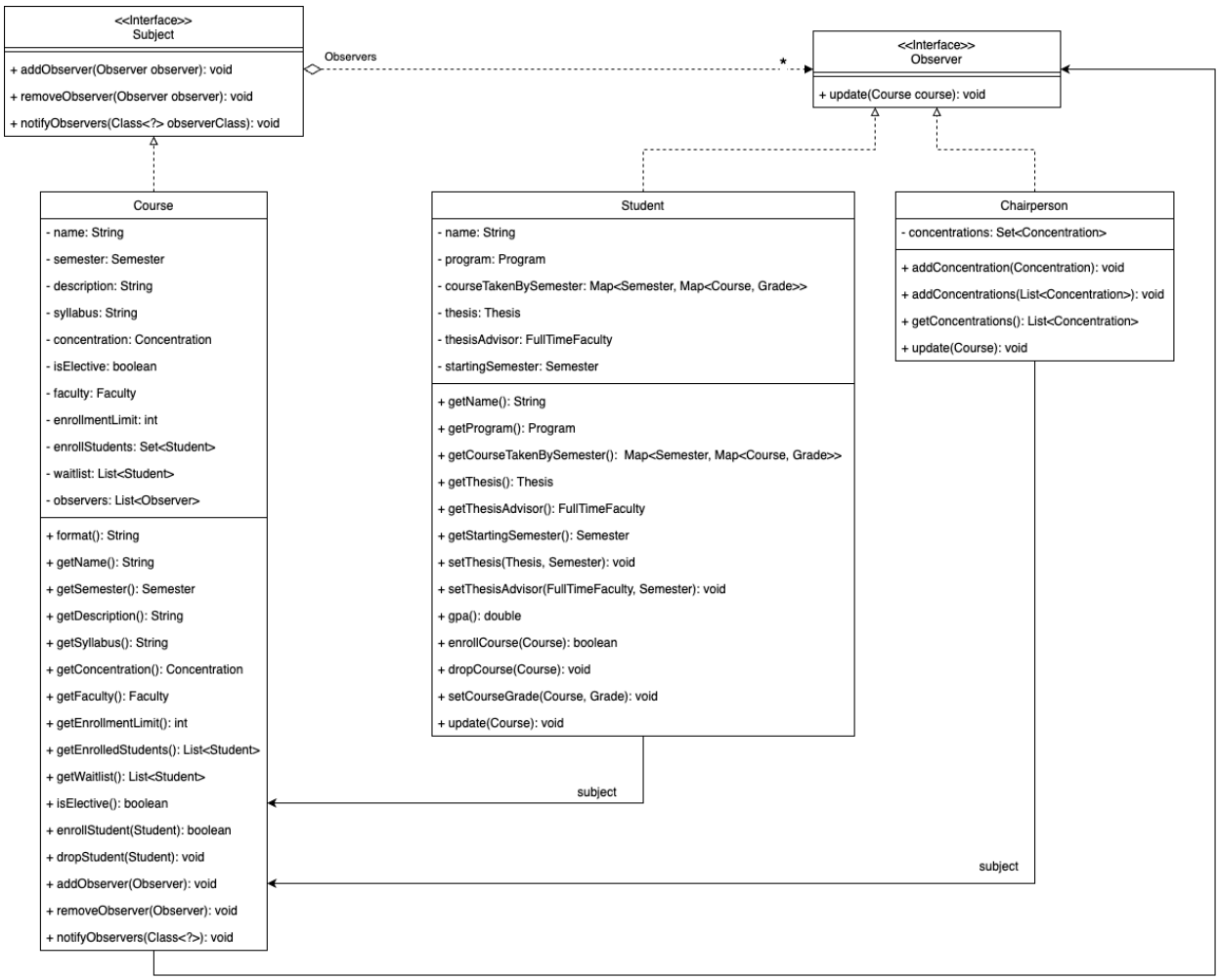


Observer Pattern (Course, Student and Chairperson)

The Observer Pattern is employed to notify relevant parties when certain events occur, such as when a course reaches its enrollment limit.

Class Interaction:

- `Observer`: An interface that defines an update method for receiving notifications.
- `Student`, `Chairperson`: Concrete observers that implement the `Observer` interface to respond to course-related events.
- `Course`: The subject that maintains a list of observers and notifies them of significant events, like when a waitlisted student is enrolled.



Test

Adapter Pattern Test

The `AdapterPatternTest` is designed to test the implementation of the Adapter pattern in the project, specifically focusing on validating whether a student can successfully enroll in a course or set up a thesis.

Data Initialization:

- **Course Creation**: The test begins by creating several courses, such as "Introduction to Java" and "Introduction to Python." These courses are initialized with various attributes like elective status, semester, and enrollment limits.
- **Student Creation**: Several student instances are created, such as "Yuqi Lin," who is enrolled in different programs, including degrees and certificates.
- **Thesis Setup**: A thesis is created, and the test attempts to assign it to students to check if they meet the necessary conditions.

Test:

- **Course Enrollment Test**: The test attempts to enroll students in courses using the `CourseAdapter`. For example, it checks whether a student can enroll in an elective course during different academic years, ensuring that the `CourseAdapter` correctly enforces the degree requirements.
- **Thesis Setup Test**: The test tries to set up a thesis for students using the `ThesisAdapter`. It verifies whether students in different programs and at different academic stages are allowed to set up a thesis.

Command Pattern Test

The `CommandPatternTest` is focused on testing the Command pattern's implementation, particularly how course enrollment and dropping actions are encapsulated as commands.

Data Initialization:

- **Faculty and Course Creation**: The test initializes faculty (using the `PartTimeFacultyFactory`) and courses such as "Introduction to Java." These courses are linked with the faculty and have specific attributes like semester and enrollment limits.
- **Student Creation**: Students such as "Tom" and "Mary" are created and enrolled in different programs like a Master's degree or a certificate.

Test:

- **Course Enrollment and Dropping**: The test uses commands (`EnrollCourseCommand` and `DropCourseCommand`) to execute course enrollment and dropping actions. A `RemoteControl` is used to set and execute these commands.

Composite Pattern Test

The `CompositePatternTest` is intended to verify the implementation of the Composite pattern, particularly how concentrations and courses are organized in a hierarchical structure.

Data Initialization:

- **Concentration and Sub-Concentration Creation**: The test creates a main concentration ("Programming Languages") and several sub-concentrations (e.g., "Object Oriented Languages").
- **Course Assignment**: Courses such as "Introduction to Java" are created and added to the appropriate sub-concentrations.

Test:

- **Hierarchical Structure Test**: The test assembles the concentration and its sub-components, verifying that the structure behaves as expected.

Factory Method Pattern Test

The `FactoryMethodPatternTest` tests the Factory Method pattern's implementation, specifically in creating different types of faculty members.

Data Initialization:

- **Faculty Creation**: The test uses various factory classes (`ChairpersonFactory`, `FullTimeFacultyFactory`, `PartTimeFacultyFactory`) to create different types of faculty members, such as a chairperson, full-time faculty, and part-time faculty.

Test:

- **Factory Method Verification**: The test checks that the correct type of faculty is created by each factory, ensuring that the Factory Method pattern is correctly implemented.

Singleton Pattern Test

The `SingletonPatternTest` is designed to verify the Singleton pattern's implementation, specifically ensuring that only one instance of each `DegreeType` exists.

Data Initialization:

- **DegreeType Access**: The test retrieves instances of `DegreeType` using the `DegreeTypeSingleton` class.

Test:

- **Singleton Instance Check**: The test compares instances of the same `DegreeType` to confirm that they are indeed the same object, proving that the Singleton pattern is working correctly.

Observer Pattern Test

The `ObserverPatternTest` aims to test the Observer pattern's implementation, particularly in managing course enrollment and notifying observers when significant events occur.

Data Initialization:

- **Course and Faculty Setup**: A course ("Introduction to Python") is created and associated with a faculty member.
- **Student Creation**: Several students are created and enrolled in the course.

Test:

- **Observer Notification Test**: The test checks whether students and the chairperson are notified when the course reaches its enrollment limit or when a student drops the course.

Department Info Test

The `DepartmentInfoTest` is designed to test the integration of different classes and patterns within the department, including course management, student enrollment, and faculty assignment.

Data Initialization:

- **Department Setup**: A department is created, and courses, students, and faculty are added.
- **Student Enrollment**: Students are enrolled in courses, and their enrollment status is tracked.

Test:

- **Course and Student Information**: The test prints out details about courses by semester and the status of student enrollments in those courses.

Student InfoTest

The `StudentInfoTest` is focused on course enrollment, GPA calculation, thesis assignment, and the retrieval of comprehensive student information.

Data Initialization:

- **Faculty and Course Creation**: A FullTimeFaculty member named "Yuqi" is created, along with two courses: "Introduction to Java" and "Introduction to Erlang." These courses are associated with the faculty member.
- **Student Creation and Enrollment**: Student "Tom" is created and enrolled in the `Bachelor of Science in CS` program, and two courses.
- **Thesis Assignment**: Student Tom did a thesis titled "How to improve garbage collection in Java," with "Yuqi" as his thesis advisor.
- **Grading**: Grades are assigned to Tom for each course: `A` for "Introduction to Java" and `B` for "Introduction to Erlang."

Test:

- **Printing Student Information**: The test prints out Tom's details, including the courses he has taken by semester, his GPA, and the thesis information.

Faculty Info Test

The `FacultyInfoTest` is focused on the courses a faculty member teaches and the students they advise across different semesters.

Data Initialization:

- **Faculty and Course Creation**: A FullTimeFaculty member named "Yuqi" is created. Two courses, "Introduction to Java" and "Introduction to Erlang," are created and assigned to "Yuqi."
- **Department Setup**: A department instance is created, and the two courses are added to it.
- **Student Creation and Thesis Assignment**: Two students, "Tom" and "Mary," are created and enrolled in different programs. Each student is assigned a thesis with "Yuqi" as their advisor in different semesters.

Test:

- **Printing Faculty Information**: The test prints out details about "Yuqi," including the faculty type, the courses being taught in various semesters, and the students being advised for their theses.