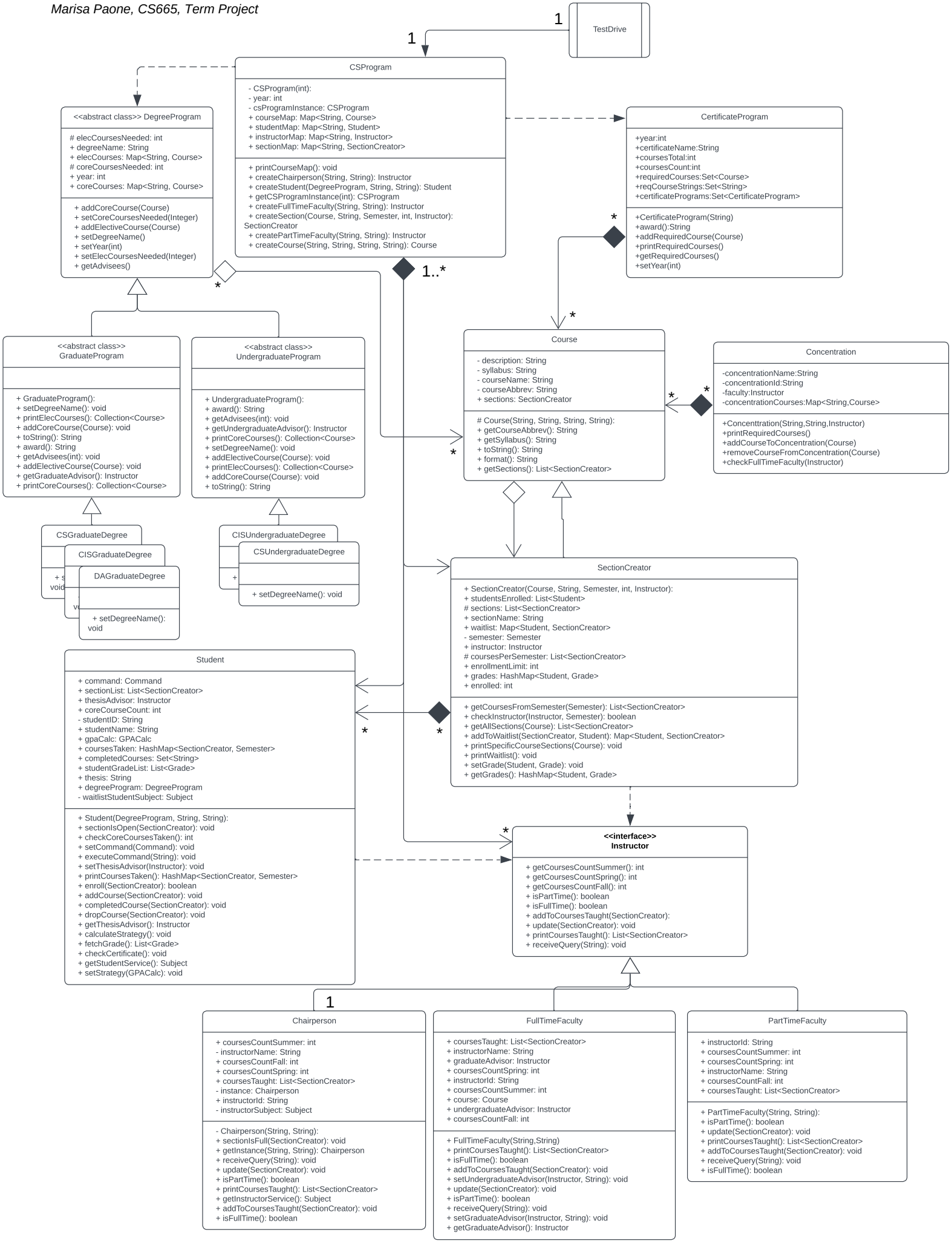
Marisa Paone, CS665, Term Project

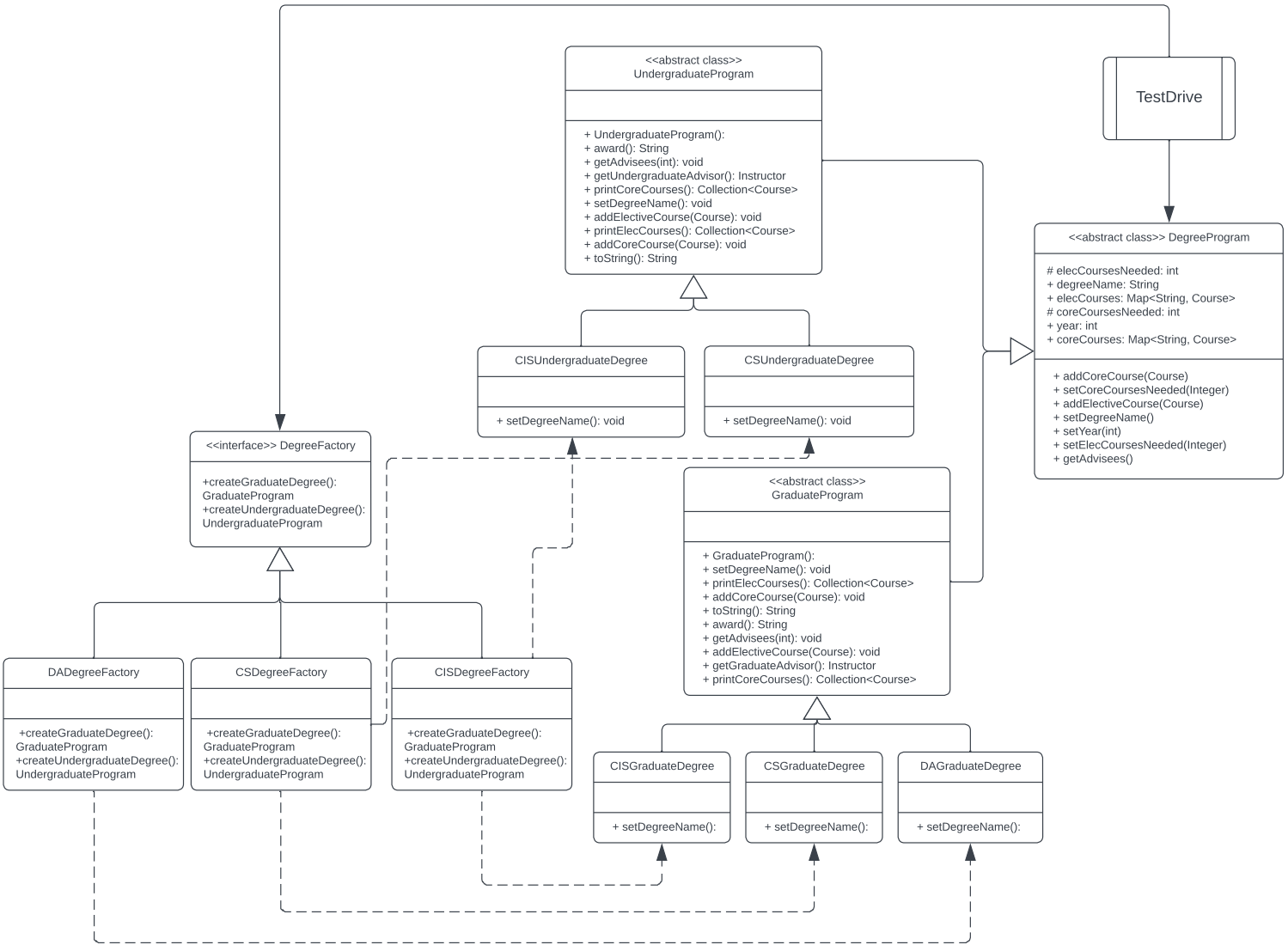


Dependencies are created when you are referencing a class or object in another class. They typically do not indicate a one to many or multiplicity relationship. My dependencies are indicated by dashed lines and they involve the CSProgram depending on the CertificateProgram and DegreeProgram. Without the certificates or degrees there would be no point to the program. The other dependency I have shown is the Student class depending on the Instructor interface. Since students have thesis advisors, instructors of courses, they have a way to send a message to their instructors, the students class depends on the instructor class.

My agreggations/associations represented by the open diamond, involve the DegreeProgram consisting of many courses. A DegreeProgram is associated with the courses a student must take. The Course class is also associated with the SectionCreator class because there are many sections of the Courses objects that are created. Ex. cs520a, cs520b.

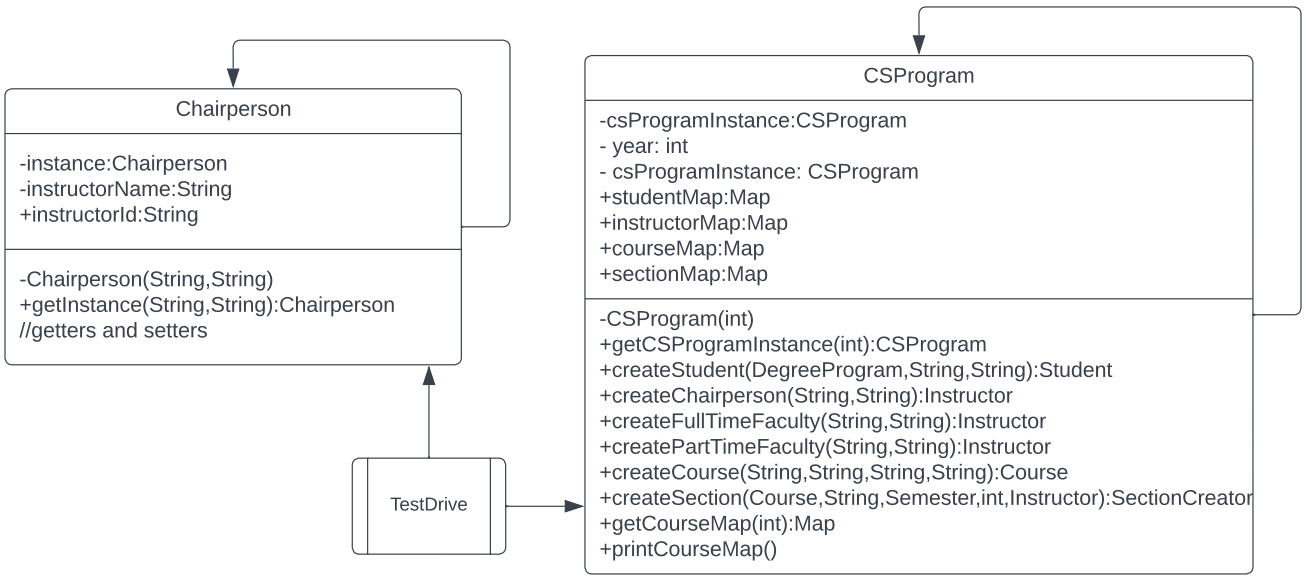
My composition relationships are plentiful (represented by the solid diamond). The CSProgram is composed of instructors, courses, sections, and students. Without these classes you wouldn't have a CS Program! Another composition relationship is the Concentration class is composed of courses. Without courses required there would be no point to having a Concentration class or awarding concentrations. This is similar to the CertificateProgram class as well - it is composed of required courses and if there was no Course class there would be no CertificateProgram class.

CREATIONAL #1:
ABSTRACT FACTORY PATTERN



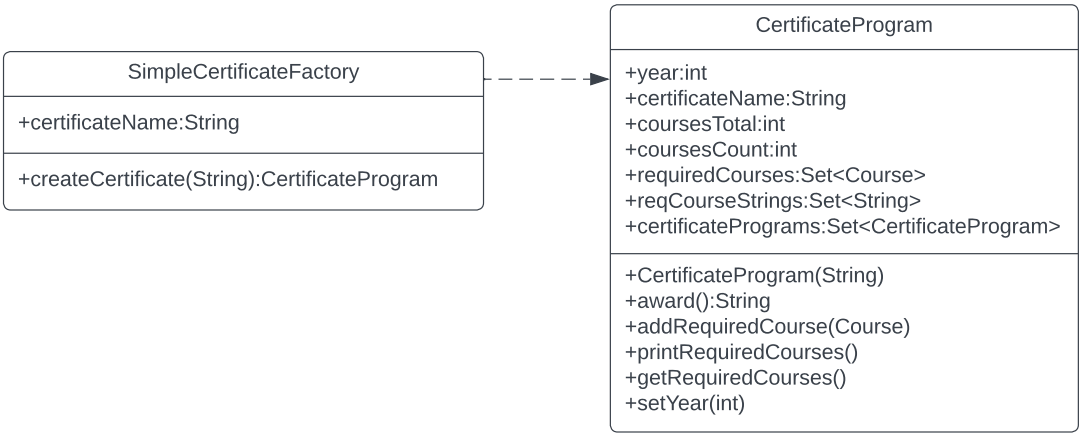
The abstract factory design pattern creates a coordinated family of families at runtime. It encapslates each family in a class whose methods return objects in that style. I chose this pattern to create all of the degrees because there were duplicates of CIS and CS degrees (grad and undergrad) but there was also a potential for the computer science program to add in more degrees like a software development grad and undergrad degree, so adding in a new degree factory for this would be easy. Refer to TestDrive lines 138-149.

CREATIONAL #2:
SINGLETON PATTERN



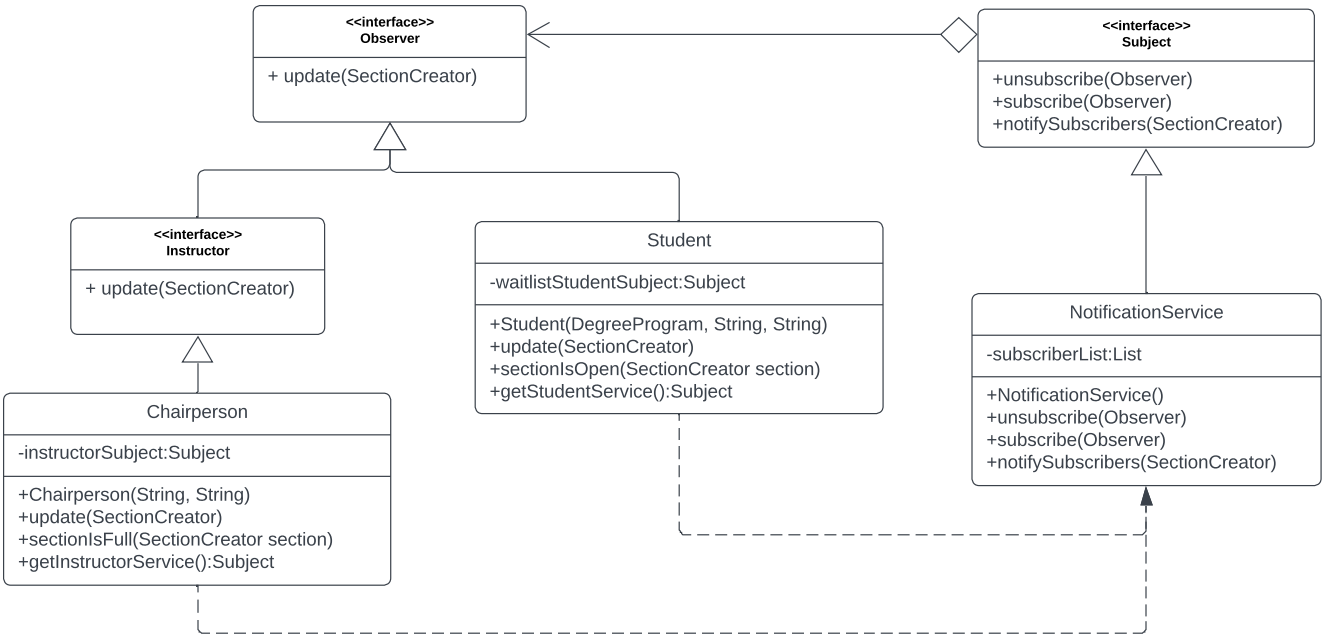
The singleton pattern is commonly used when there is only one of its kind. The singleton pattern ensures that a class has at most one instatiation accessible through the application. In our case, there is only one CSProgram, and one Chairperson. CSProgram is used as a single instance for student creation, instructor creation, course creation and section creation. Refer to Test Drive Linese 10, 335-343.

SIMPLE FACTORY PATTERN



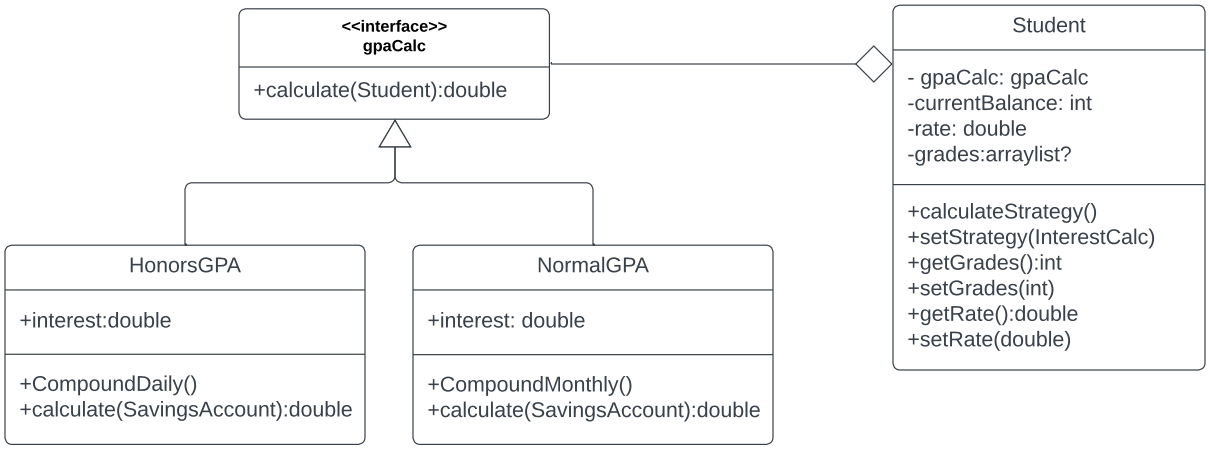
Although the simple factory pattern isn't a creational design pattern and more of an intermediate step towards the factory pattern, I thought I would include a diagram anyway. I could have created classes for each certificate which extended the Certificate Program and then used the CertificateFactory class, but I thought it would be better to create the certificates as objects instead of classes. The certificates I created in my TestDrive main class were a Security, Web Technology, and Analytics Certificate. The certificates would be awarded after a student gets a grade on a class (the student has completed the class) and the setGrade method in SectionCreator triggers the completedCourse method in the student class which then checks if you have fufilled a certificate's requirements. Refer to TestDrive Lines 419-444.

BEHAVIORAL #1:
OBSERVER PATTERN



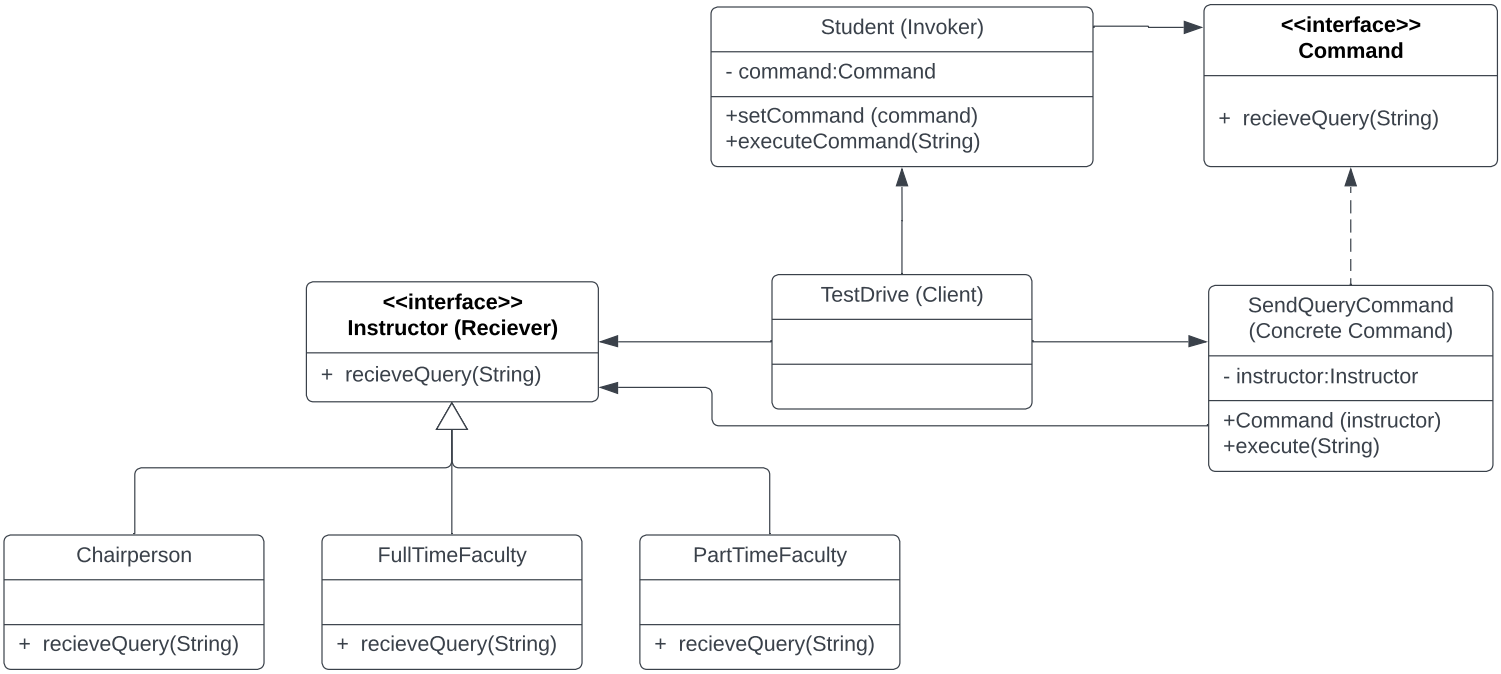
The observer pattern specifies communication between subjects and observers. It is typically used when there is a one to many relationship between objects. This pattern allows you to change or take action on a set of objects when and if the state of another object changes. I used the observer pattern for two purposes (subjects), one is for the Chairman to be notified when a course section is full, and the other reason is to add the student to a waitlist and notify them that they are on the waitlist when they try to enroll in a full course section. When another student drops a full course section, the next student on the waitlist is notified and automatically enrolled through the observer pattern. This happens through my dropCourse method in the Student class. Refer to TestDrive Lines 380-407.

BEHAVIORAL #2:
STRATEGY PATTERN



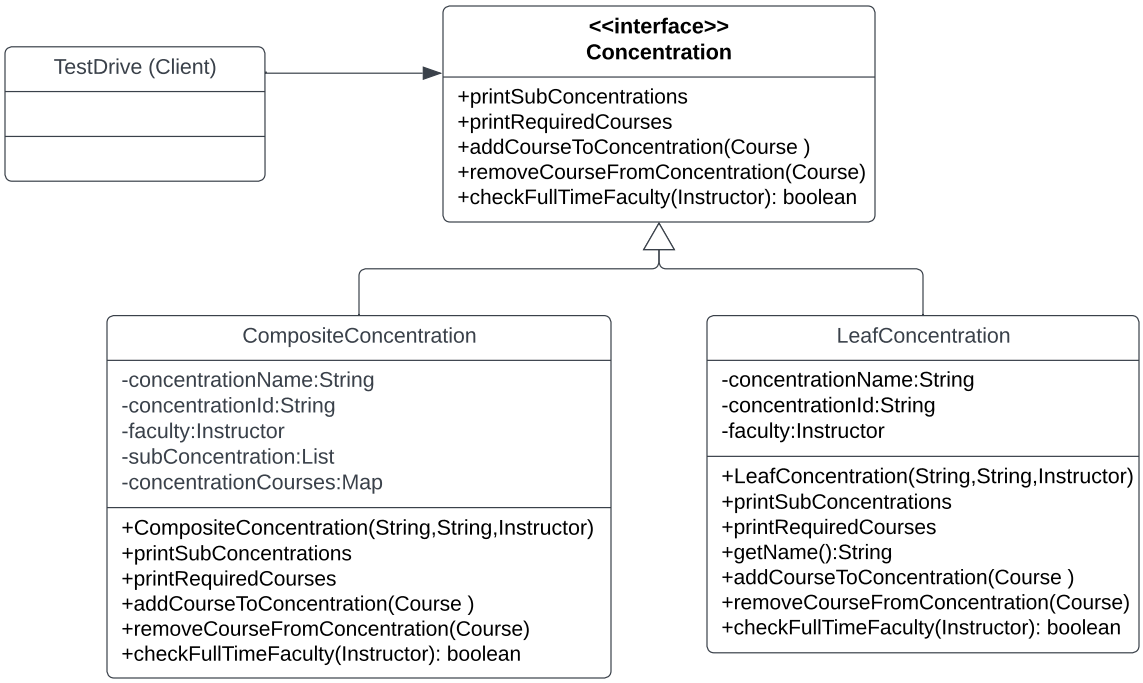
The strategy pattern defines a family of algorithms, encapsulates each one, and then makes them interchangable. Its best used when we have multiple algorithms for a specific task and the client wants to decide the actual implementation to be used at runtime. Calculating a students GPA with two different calculation methods is a perfect example of this. Sometimes we want to consider "H"'s (98-100) as valued higher than A's (95 and above). An H would demonstrate excellence and would raise the students GPA to be out of 5.0 instead of 4.0. But other times, employers or universities do not think H's should be awarded and are expecing GPA's on a 4.0 scale. This is where the strategy pattern comes in. At runtime, we can decide to calculate a students GPA the normal way with H's counting as A's, or the Honors way with H's counting as 5.0. Refer to TestDrive Lines 290-324.

BEHAVIORAL #3:
COMMAND PATTERN



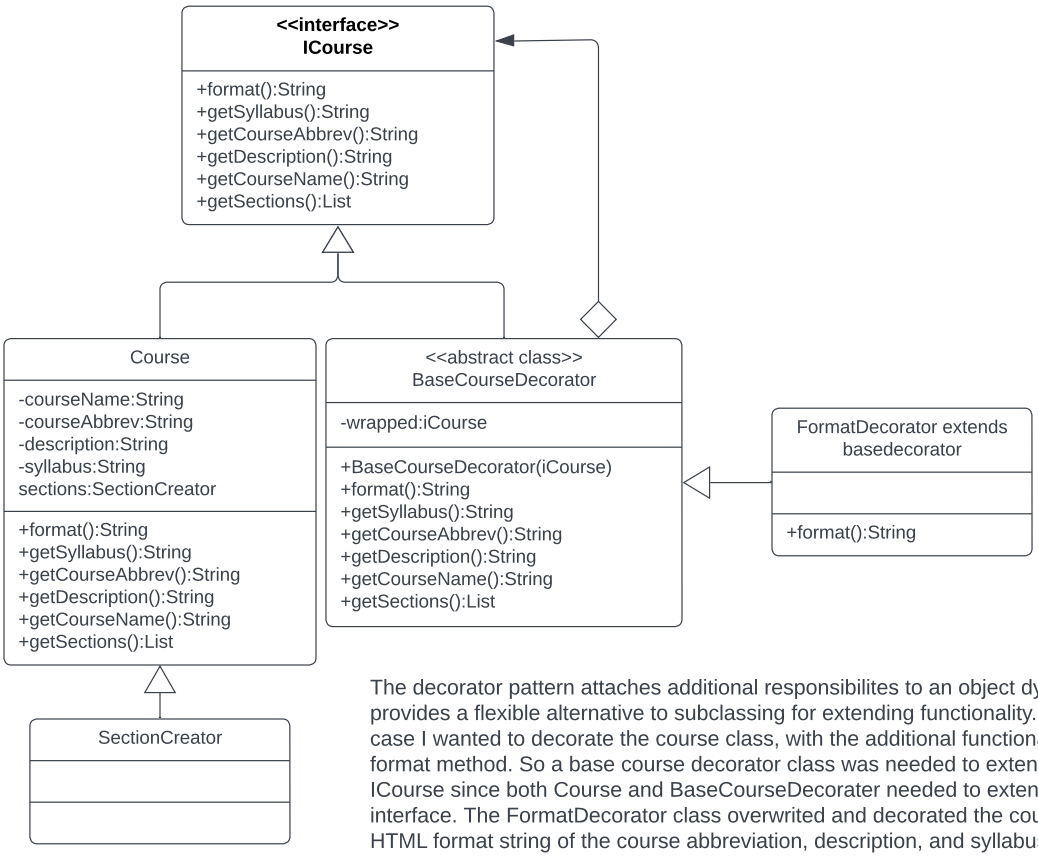
The command pattern makes the execution of operations of a program more flexible. Each command is captured in a class of its own. This pattern enables you to pass commands as method arguments, storing them inside other objects or switching commands at runtime. This command pattern was implemented to give students a way to communicate with their advisor, faculty, and chairperson. The student is able to set the command as a SendQueryCommand and inputs the instructor they want to send the message to, then the student will use the executeCommand method with their query. Refer to the TestDrive class lines 328-333.

STRUCTURAL #1:
COMPOSITE PATTERN



The composite pattern represents a tree of objects. It allows client code to access uniform functionality distributed in the subtree rooted by any node. I used this design pattern for the concentration. The composite pattern was used to structurally organize concentrations since a concentration can have subconcentrations. A CompositeConcentration has subconcentrations that are LeafConcentrations. However leaf concentrations can also exist on their own. Refer to TestDrive Lines 347-378.

STRUCTURAL #2:
DECORATOR PATTERN



The decorator pattern attaches additional responsibilites to an object dynamically. It provides a flexible alternative to subclassing for extending functionality. For this use case I wanted to decorate the course class, with the additional functionality of an HTML format method. So a base course decorator class was needed to extend the interface ICourse since both Course and BaseCourseDecorater needed to extend a shared interface. The FormatDecorator class overwrited and decorated the courses with an HTML format string of the course abbreviation, description, and syllabus. Refer to TestDrive Lines 409-417.