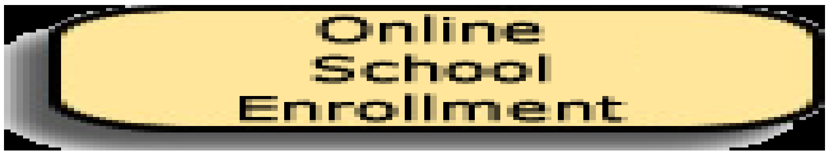


You are required to develop an automated Student Registration System (SRS). This system will enable students to register online for courses each semester.

As part of the exercise you will have to perform implementation (design) decisions. You should weigh the options available to you, and select the most appropriate one. Wherever you take a non-trivial implementation decision (two equivalent options, etc), document the considerations.
Read the entire exercise and supplementary files before you commence the implementation. Think how the system has to operate given the functional requirements and code limitations. Plan your code in a way that supports all the required scenarios.
Points to observe: What are the classes you chose to implement? What are the relationships between the classes? What is the responsibility of each service in these classes? Do you need additional helper classes? Consider these questions while designing your solution. Invest time in finding answers for these questions, it is time well invested as will save later code revisions, arising from ill thought out implementations.

Don't get caught up in form over substance! The model that you produce is only a means to an end.. and the process, notation, and tools that you use to produce the model are but a *means* to this end. If you get too hung up on which notation to use, or which process to use, or which toll to use, you may wind up spinning your wheels in 'analysis paralysis'. Don't lose sight of your ultimate goal: to build useable, flexible, maintainable, reliable, functionally correct software systems.

SRS Requirements Specification

When a student first enrolls at the university, the student uses the SRS to specify a major and a degree. During the registration period preceding each semester, the student is able to view the schedule of classes online, and choose whichever classes he or she wishes to attend, indicating the preferred section (day of week and time of day) if the class is offered by more than one professor. The SRS will verify whether or not the student has satisfied the necessary prerequisites for each requested course by referring to the student's online transcript of courses completed and grades received.

Assuming that (a) the prerequisites for the requested course(s) are satisfied, and (b) there is room available in each of the class(es), the student is enrolled in the class(es). It is the student's responsibility to drop the class if it is no longer desired. Students may drop a class up to the end of the first week of the semester in which the class is being taught. At the end of the semester grades are posted, possible grades are variations on A, B, C, D (i.e. A, A+, A-), and an F. A passing grade is anything above an F. Only if a student passes a course, is this course updated in his transcript.

Use Cases

In determining what the desired functionality of a system is to be, we must seek answers to the following questions:

- **Who** will want to use our system?
- What **services** will the system need to provide in order to be of value to them?
- When a user interacts with the system for a particular purpose, what is his/her expectation as to the **desired outcome**?

Use cases are a natural way to express the answers to these questions. Each use case is a simple statement, narrative and/or graphical in fashion, which describes a particular goal or outcome of the system, and by whom the outcome is expected. For example, one goal of the SRS is to 'enable a student user to register for a course'.
The purpose of thinking through all the use cases for a system is to explore the system's **functional requirements** thoroughly, so as to make sure a particular category of user, or potential purpose of the system, isn't overlooked. We differentiate between functional requirements and technical requirements. **Functional requirements** are those aspects of a system that have to do with how it is to operate or function from the perspective of someone using the system. **Technical requirements**, on the other hand, have more to do with how a system is to be built internally in order to **meet** the functional requirements.

The high-level use cases for the SRS are:

1. Enroll in university.
2. Register for a course.
3. Drop a course.
4. Determine a student's course load.
5. View schedule of classes.
6. Request a student roster for a given course.
7. Request a transcript for a given student.
8. Maintain course information: course id, course description, course instructor.
9. Post final semester grades for a given course.

We may decompose any one of the use cases into steps, each one representing a 'sub use case';

'Enroll in university' should be decomposed into:

I. Set student name
II. Set student ssn.
III. Declare degree.
IV. Declare major.

'Register for a course' should be decomposed into:

I. Verify that a student has met the prerequisites.
II. Check availability of a seat in the course.

'Post final semester grades for a given course' should be decomposed into:

I. Update student transcript with grade if student passed.
II. Remove course from student course load.

'Drop a course'

I. Check if a week into the semester has yet to pass.
II. If yes, then remove the course from student course load.
III. Otherwise do not remove course.

It is up to you to define the contract for the rest of the use cases, anything is acceptable as long as it is reasonable and well documented.

Modelling Static/Data Aspects of the System

Having employed a use case analysis to determine the SRS requirements specification, the next stage of modeling is determining how these requirements are going to be met in an object-oriented fashion. Objects form the building blocks of an OO system, and classes are templates used to define and instantiate objects. An OO model, then, must specify:

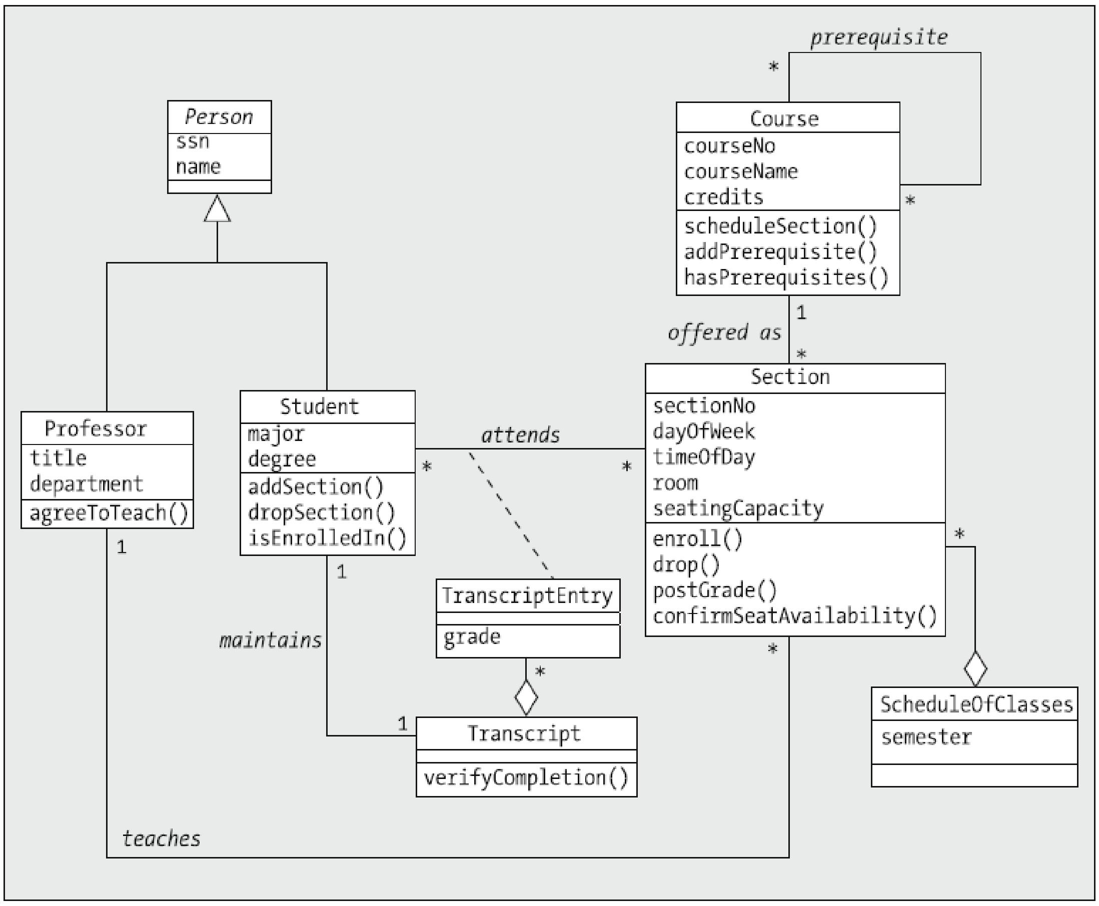
- What classes of objects are needed to be created and instantiated in order to represent the proper abstraction: in particular, their attributes, methods, and structural relationships with one another. This is called the static model as once established is not supposed to change (unless you did not model correctly).
- How these objects will need to collaborate in carrying out the overall requirements, or 'mission', of the system: The ways in which objects interact can change literally from one moment to the next based upon the circumstances that are in effect. One moment, a *Course* object may be registering a *Student* object, and the next it might be responding to a query by a *Professor* object as to the current student head count. We refer to the process of detailing object collaborations as preparing the **dynamic model**.

The static and dynamic models are simply two different sides of the same coin: they jointly comprise the blueprint of the object-oriented implementation of the SRS.

Identifying appropriate classes

The first challenge in object modeling is determining what classes are necessary as our system building blocks. Unfortunately, the process of class identification is 'fuzzy'; it relies heavily on intuition, prior modeling experience, and familiarity with the subject area, or **domain**, of the system to be developed. So, how does an object modeling novice ever get started? One tried and true (but somewhat tedious) procedure for identifying candidate classes is to use the 'hunt and gather' method: that is, to hunt for and gather a list of all nouns/noun phrases from the project documentation set and to then use a process of elimination to whittle this list down into a set of appropriate classes. In the case of the SRS, our documentation set thus far consists of: The requirements specification; The use case model.

Such an analysis could yield the following list of classes as depicted by the class diagram below. These are the system objects you are required to implement. You may need additional 'helper' classes, in order to implement the system requirements.



OO Feature	Embodied in the SRS Class Diagram As Follows:
Inheritance	The Person class serves as the base class for the Student and Professor subclasses.
Aggregation	We have two examples of this: the Transcript class represents an aggregation of TranscriptEntry objects, and the ScheduleOfClasses class represents an aggregation of Section objects.
One-to-one association	The *maintains* association between the Student and Transcript classes.
One-to-many association	The *teaches* association between Professor and Section; the *offered as* association between Course and Section.
Many-to-many association	The *attends* association between Student and Section; the *prerequisite* (reflexive) association between instances of the Course class.
Association class	The TranscriptEntry class is affiliated with the *attends* association.
Reflexive association	The *prerequisite* association between instances of the Course class.
Abstract class	The Person class will be implemented as an abstract class.
Metadata	Each Course object embodies information that is relevant to multiple Section objects.
Static attributes	Although not specifically illustrated in the class diagram, we'll take advantage of static attributes when we code the Section class.
Static methods	Although not specifically illustrated in the class diagram, we'll take advantage of static methods when we code the TranscriptEntry class.

The class diagram above depicts the core functionality of our application, but we would like to be able to support different types of students, namely **UndergraduateStudent** and **GraduateStudent**. A course can only be taken by a GraduateStudent if the student majors in the program the Course belongs to. UndergraduateStudents can take courses from any program they like, as long as they take at least 6.0 credits each semester belonging to the program they are majoring in. Thus when an UndergraduateStudent attempts registering to a course it will be allowed to register if this course belongs to his major, or he has already registered to 2 courses of his major this semester. General Student objects have no restrictions on their allowed course registration. Supporting more specific types of the same object, is achieved in OO programming via inheritance. The term **polymorphism** refers to the ability of two or more objects belonging to ***different*** classes to respond to exactly the ***same*** message (method call) in different class-specific ways. In our SRS application, we'll declare a HashMap called studentBody to hold references to Student objects. We'll then populate the HashMap with Student object references—some GraduateStudents, some UndergraduateStudents and some generic Students, randomly mixed. We will then envoke the same message call (sometimes iteratively), and note that each object will respond in its class specific way. One of the benefits of polymorphism is It ***minimizes "ripple effects"*** on client code when new subclasses are added to the class hierarchy of an existing application, resulting in ***dramatically reduced maintenance costs***.

The SRS 'Driver' Program

Now that we have defined all the classes called for by our model of the SRS, we need a way to implement the application logic, and enable users to access this logic. The standard is to build a GUI (Graphical User Interface) front-end, but this is out of the scope of this exercise. So instead we will provide you with a command line driven program (SRSDriver.java) to instantiate objects of varying types and to invoke their critical methods, displaying the results to the command-line window. You are required to implement the mandatory classes depicted in the class diagram, and the two additional classes specified, namely, GraduateStudent and UndergraduateStudent. You may implement any helper classes you deem necessary, in order to render the SRSDriver.java useable. We provide templates for all the required classes (including a full implementation of ScheduleOfClasses). In addition we supply a partial implementation of UniversityRegistry, which is a class required by our implementation of the SRS driver. It contains a reference to all the objects comprising the SRS application. We also provide two enum classes Day and EnrollmentStatus. You are free to add any members and methods to the classes defined by the requirements, as well as additional helper classes, as long as you document sufficiently.

The SRSDriver requires you display all the objects comprising the SRS application, namely, faculty, studentBody, courseCatalog, and scheduleOfClasses, via a specific display method, whose template api is defined in UniversityRegistry. The display method is required to sort the objects according to an identifier, and then display. The faculty and studentBody should be sorted according to their name, while the courseCatalog according to courseNo, and scheduleOfClasses according to sectionNo. We do not allow using the java Collections sort algorithm, and require you to implement your own sorting algorithm, which is required to be merge sort: http://en.wikipedia.org/wiki/Merge_sort. We allow only one sorting implementation.

The exact implementation requirements are documented in the template classes provided. You must add documentation to all the methods and attributes of the supplied template classes, and of-course to your own additions. Part of your grade will be determined by the quality of your documentation.

Good Luck!