Some techniques that software designers use in choosing and designing classes

Imagine that we are designing a registration system for your school. Where should we begin? A useful way to start would be to look at the system from a functional point of view, as follows:

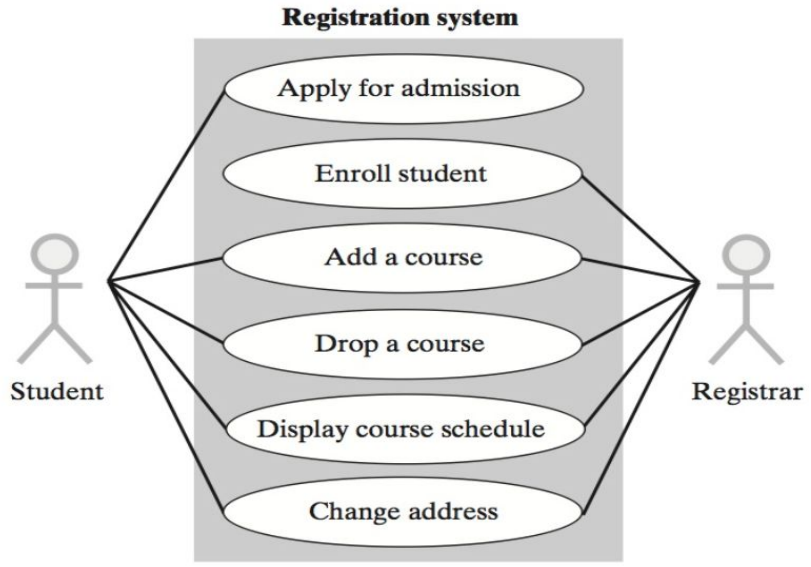
● Who or what will use the system? A human user or a software component that interacts with the system is called an actor. So a first step is to list the possible actors. For a registration system, two of the actors could be a student and the registrar.

● What can each actor do with the system? A scenario is a description of the interaction between an actor and the system. For example, a student can add a course. This basic scenario has variations that give rise to other scenarios. For instance, what happens when the student attempts to add a course that is closed? Our second step, therefore, is to identify scenarios. One way to do this is to complete the question that begins “What happens when...”.

● Which scenarios involve common goals? For example, the two scenarios we just described are related to the common goal of adding a course. A collection of such related scenarios is called a use case. Our third step, then, is to identify the use cases.

You can get an overall picture of the use cases involved in a system you are designing by drawing a use case diagram. Figure below is a use case diagram for our simple registration system. Each actor— the student and the registrar—appears as a stick figure. The box represents the registration system, and the ovals within the box are the use cases. A line joins an actor and a use case if an interaction exists between the two.

A use case diagram for a registration system



Some use cases in this example involve one actor, and some involve both. For example, only the student applies for admission, and only the registrar enrolls a student. However, both the student and the registrar can add a course to a student's schedule.

Although drawing a use case diagram is a step in the right direction, it does not identify the classes that are needed for your system. Several techniques are possible, and you will probably need to use more than one.
One simple technique is to describe the system and then identify the nouns and verbs in the description. The nouns can suggest classes, and the verbs can suggest appropriate methods within the classes. Given the imprecision of natural language, this technique is not foolproof, but it can be useful.
For example, we could write a sequence of steps to describe each use case in Figure below.. Figure below gives a description of the use case for adding a course from the point of view of a student. Notice the alternative actions taken in Steps 2a and 4a when the system does not recognize the student or when a requested course is closed.

A description of a use case for adding a course

System: Registration
Use case: Add a course
Actor: Student
Steps:

1. Student enters identifying data.
2. System confirms eligibility to register.
   a. If ineligible to register, ask student to enter identification data again.
3. Student chooses a particular section of a course from a list of course offerings.
4. System confirms availability of the course.
   a. If course is closed, allow student to return to Step 3 or quit.
5. System adds course to student's schedule.
6. System displays student's revised schedule of courses.

What classes does this description suggest?
Looking at the **nouns**, we could decide to have classes to represent a student, a course, a list of all courses offered, and a student's schedule of courses.
The **verbs** suggest actions that include confirming whether a student is eligible to register, seeing whether a course is closed, and adding a course to a student's schedule.

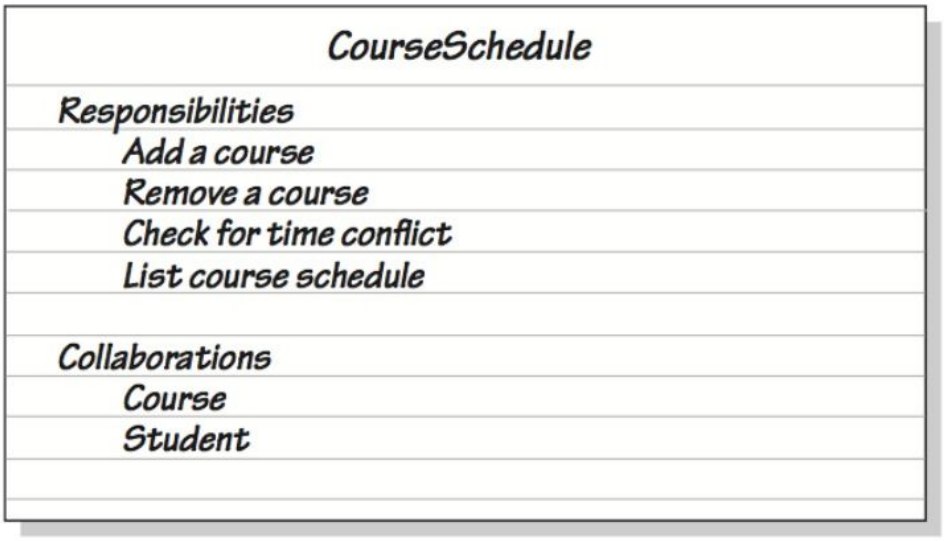
One way to assign these actions to classes is to use CRC cards.

A simple technique for exploring the purpose of a class uses index cards. Each card represents one class. You begin by choosing a descriptive name for a class and writing it at the top of a

card. You then list the actions that represent the class's responsibilities. You do this for each class in the system. Finally, you indicate the interactions, or collaborations, among the classes. That is, you write on each class's card the names of other classes that have some sort of interaction with the class. Because of their content, these cards are called class-responsibility-collaboration, or CRC, cards.

For example, Figure below shows a CRC card for the class CourseSchedule that represents the courses in which a student has enrolled. Notice that the small size of each card forces you to write brief notes. The number of responsibilities must be small, which suggests that you think at a high level and consider small classes. The size of the cards also lets you arrange them on a table and move them around easily while you search for collaborations.

A class-responsibility-collaboration (CRC) card

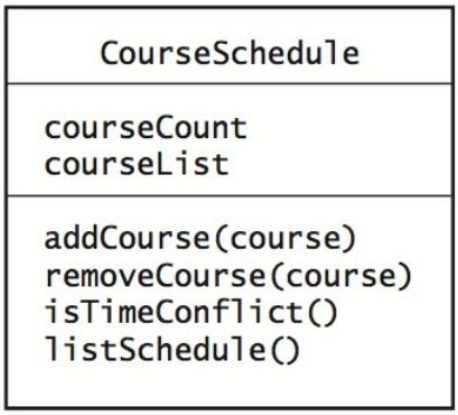


A simple technique for exploring the purpose of a class uses index cards. Each card represents one class. You begin by choosing a descriptive name for a class and writing it at the top of a card. You then list the actions that represent the class's responsibilities. You do this for each class in the system. Finally, you indicate the interactions, or collaborations, among the classes. That is, you write on each class's card the names of other classes that have some sort of interaction with the class. Because of their content, these cards are called class-responsibility-collaboration, or CRC, cards.

For example, Figure above shows a CRC card for the class CourseSchedule that represents the courses in which a student has enrolled. The number of responsibilities must be small, which suggests that you think at a high level and consider small classes. The size of the cards also lets you arrange them on a table and move them around easily while you search for collaborations.

The Unified Modeling Language: The use of case diagram in is part of a larger notation known as the Unified Modeling Language, or UML. Designers use the UML to illustrate a software system's necessary classes and their relationships. The UML gives people an overall view of a complex system more effectively than either a natural language or a programming language can. English, for example, can be ambiguous, and Java code provides too much detail. Providing a clear picture of the interactions among classes is one of the strengths of the UML. Besides the use case diagram, the UML provides a class diagram that places each class description in a box analogous to a CRC card. The box contains a class's name, its attributes (data fields), and operations (methods). For example, Figure below shows a box for the class CourseSchedule. Typically, you omit from the box such common operations as constructors, get methods, and set methods.

A class representation that can be a part of a class diagram

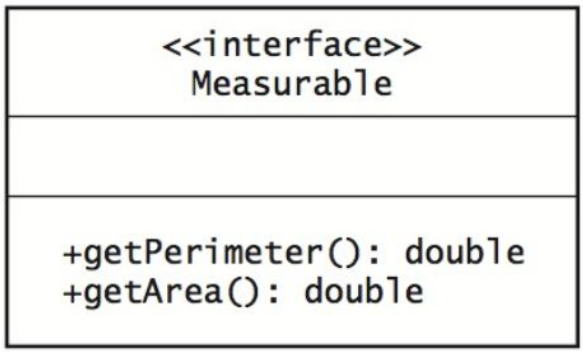


As your design progresses, you can provide more detail when you describe a class. You can indicate the visibility of a field or method by preceding its name with + for public, - for private, and # for protected.

You also can write the data type of a field, parameter, or return value after a colon that follows the particular item. Thus, in Figure below you can write the data fields as:
-courseCount: integer
-courseList: List

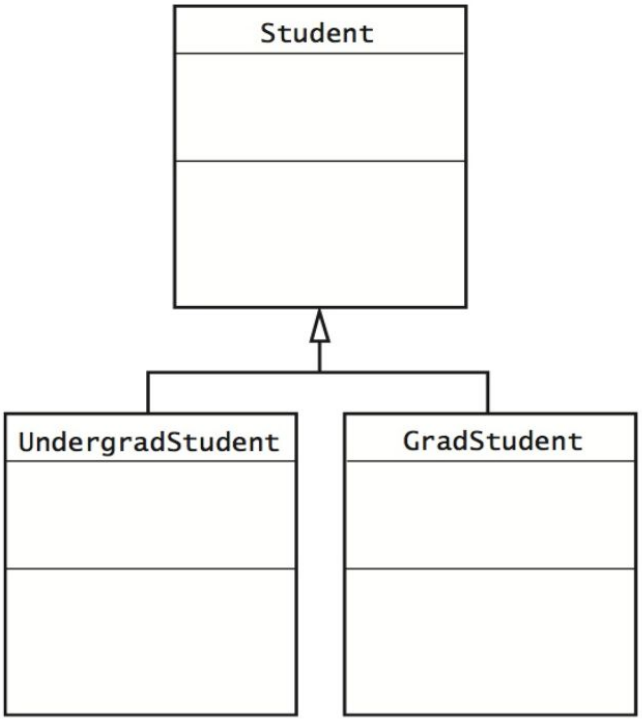
and the methods as
+addCourse(course: Course): void
+removeCourse(course: Course): void
+isTimeConflict(): boolean
+listSchedule(): void
You represent an interface in UML much as you represent a class, but you precede its name with <>.

UML notation for the interface `Measurable`



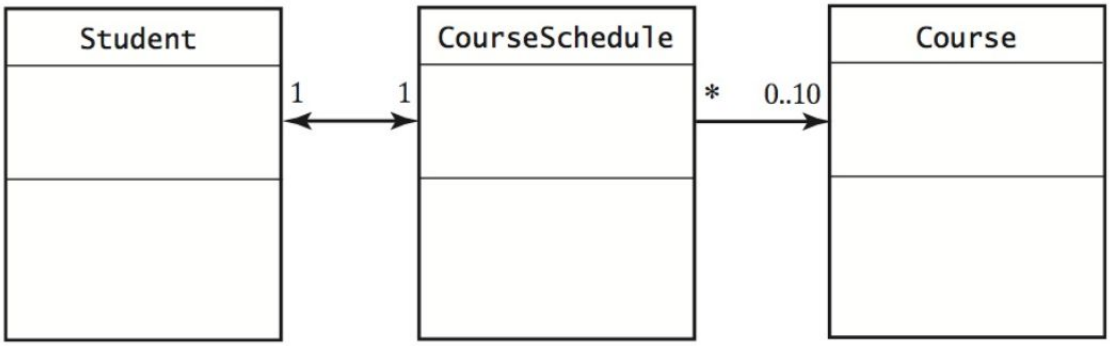
In a class diagram, lines join the class boxes to show the relationships among the classes, including any inheritance hierarchy. For example, the class diagram in Figure below shows that the classes UndergradStudent and GradStudent are each derived from the class Student. An arrow with a hollow head points to the superclass. Within the UML, the superclass Student is said to be a generalization of UndergradStudent and GradStudent. If a class implements an interface, you draw an arrow having a dotted shaft and hollow head from the class to the interface.

A class diagram showing the base class `Student` and two subclasses



Association: An association is a relationship between two objects of different classes. Basically, an association is what a CRC card calls a collaboration. For example, relationships exist among the classes Student, CourseSchedule, and Course. Figure below shows how the UML pictures these relationships as arrows. The arrow between the classes CourseSchedule and Course, for example, indicates a relationship between objects of the class CourseSchedule and objects of the class Course. This arrow points toward Course and indicates responsibilities. Thus, a CourseSchedule object should be able to tell us the courses it contains, but a Course object need not be able to tell us to which schedules it belongs. The UML calls this aspect of the notation the navigability.

Part of a UML class diagram with associations



This particular association is said to be **unidirectional**, since its arrow points in one direction. An association indicated by a line with arrowheads on both ends is called bidirectional. For example, a Student object can find its course schedule, and a CourseSchedule object can discover the student to which it belongs. You can assume that the navigability of an association represented by a line without arrowheads is unspecified at the present stage of the design.
At the ends of each arrow are numbers. At the head of the arrow between CourseSchedule and Course, you see the notation 0..10. This notation indicates that each CourseSchedule object is associated with between zero and ten courses. At the other end of this arrow is an asterisk. It has the same meaning as the notation 0..infinity. Each Course object can be associated with many, many course schedules—or with none at all. The class diagram also indicates a relationship between one Student object and one CourseSchedule object. This notation on the ends of an arrow is called the association's **cardinality or multiplicity**.

Points to remember:

- When you first start to write programs, you can easily get the impression that each program is designed and written from scratch. On the contrary, most software is created by combining already existing components with new components. This approach saves time and money. In addition, the existing components have been used many times and so are better tested and more reliable.

For example, a highway simulation program might include a new highway object to model a new highway design, but it would probably model automobiles by using an automobile class that had already been designed for some other program. As you identify the classes that you need for your project, you should see whether any of the classes exist already. Can you use them as is, or would they serve as a good base class for a new class?

- As you design new classes, you should take steps to ensure that they are easily reusable in the future. You must specify exactly how objects of that class interact with other objects. This is the principle of encapsulation You must also design your class so that the objects are general and not tailored too much for one particular program. For example, if your program requires that all simulated automobiles move only forward, you should still include a reverse in your automobile class. Some other simulation may require automobiles to back up.

Hacking together a solution to your specific problem would take less time. But the payback for your effort will come later on, when you or another programmer needs to reuse an interface or a class. If you planned for the future when you wrote those components, every use of them will be faster and easier. Actual software developers use these principles to save time over the long term, because saving time saves them money.