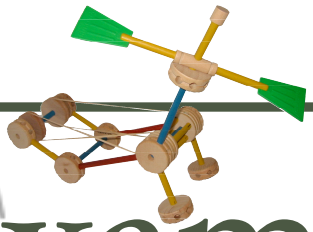


UML Examples

THE **CLASS** SYMBOL REPRESENTS THE TEMPLATE FOR OBJECTS OF THIS CLASS

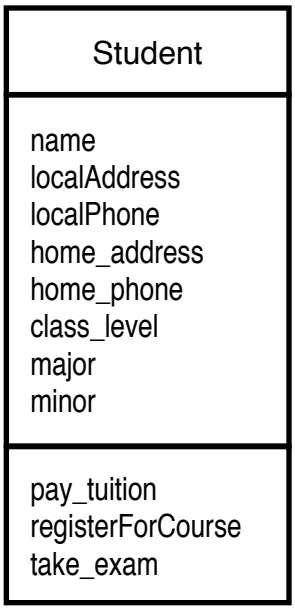
In UML-2 class diagrams there are no "object" symbols. The Class symbol is used to represent any and all possible instances of that "kind" of object in the drawing.

The Class symbol captures four important concepts: 1) that the class is either concrete or abstract, 2) what data attribute variables will all objects of this class have values for, 3) what services will every object of this class be able to perform and 4) what name best captures the stakeholders' understanding of this class's role in the model. Consider the following paragraph documenting the stakeholders' understanding of a domain concept.

"The college records information about every student including their name, their local and home address and phone numbers. The college also records their 'class level' (i.e. first semester freshman, second semester freshman, . . . second semester senior) and their academic major and minor disciplines. Every student is expected to be able to pay tuition, register for courses and take exams."

In this example, an instance of "student" represents the information the college records about a "real" student. The data elements are modeled as data attribute variables in the second pane of the symbol. Each of the "actions" expected of a student is modeled as a "service" in the third pane. The name of the Class seems best served by "Student," a singular noun most often used to refer to one of the many persons enrolled in the college taking courses.

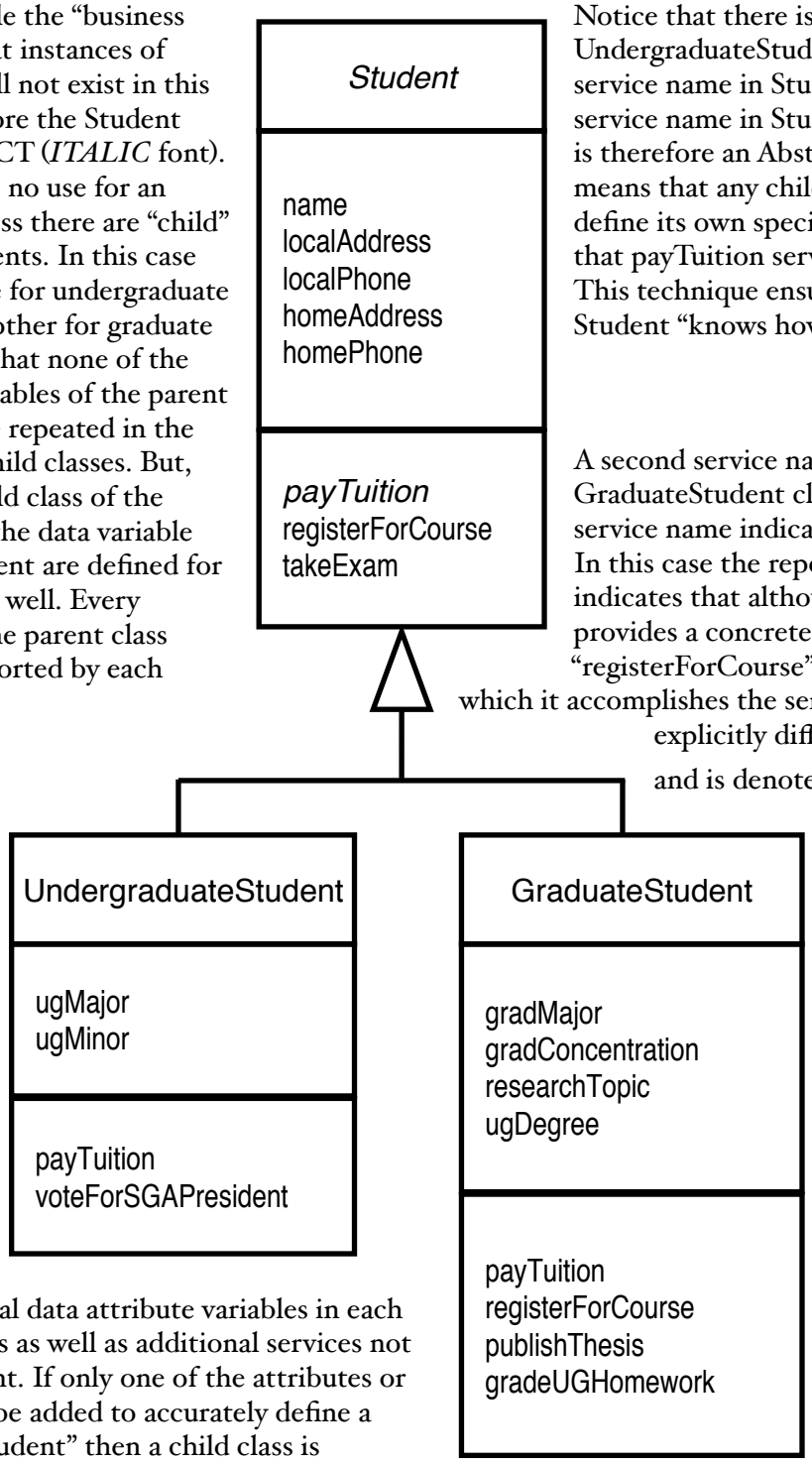
Notice that the name of the Class is not specified in an *ITALIC* font and therefore this Class is concrete meaning that actual objects, instances of Class Student, can / will exist in this modeling world. In order to model an Abstract Class (*ITALIC* font), there would be some explicit indication that no objects of this Class would exist (or be allowed) such as "this is an abstraction for the specific Classes of students that will exist (e.g. undergraduate student or graduate student).



The syntax for naming attributes and services is intended to make the names as readable and understandable as possible. The example above uses two different techniques: 1) use an underscore to separate words in a "multiple-word" name or 2) capitalize the first letter of what would otherwise be a separate word if you omit underscores.

Consider the example of an abstract class that follows on the next page. . .

In this example the "business rule" indicates that instances of "plain student" will not exist in this model and therefore the Student Class is ABSTRACT (*ITALIC* font). In general there is no use for an abstract class unless there are "child" classes that it parents. In this case there are two: one for undergraduate students and the other for graduate students. Notice that none of the data attribute variables of the parent class (Student) are repeated in the symbols for the child classes. But, since each is a child class of the Student Class all the data variable attributes of Student are defined for each child class as well. Every service name in the parent class must also be supported by each child class.



Notice that there is one service name in UndergraduateStudent that is the same as a service name in Student. That's because the service name in Student is in *ITALICS* and is therefore an Abstract Service – which means that any child class of Student must define its own special method, "HOW," that payTuition service is implemented. This technique ensures that any child of Student "knows how to payTuition."

A second service name in the GraduateStudent class is the same as a service name indicated for the parent class. In this case the repeated service name indicates that although Graduate_Student provides a concrete (not abstract) service "registerForCourse" the exact steps with which it accomplishes the service (the "HOW") is explicitly different in the child class and is denoted by repeating the service name. This indicates that the method of implementing the service is "OVERRIDDEN" in the child class. Since the service name is omitted in the UndergraduateStudent class that indicates that whatever method for implementing the service was defined in the Student Class is exactly the same method that every instance of UndergraduateStudent will apply.

The only reason for having child (specialization) classes is that there is some definition in the child class that is not completed by the inherited attributes and services of the parent class. In this example there are additional data attribute variables in each of the child classes as well as additional services not found in the parent. If only one of the attributes or services needs to be added to accurately define a "special kind of student" then a child class is appropriate!