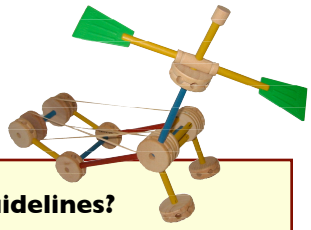
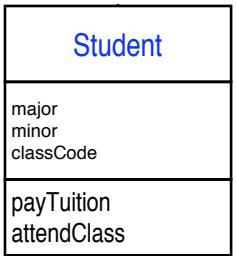
UML Guidelines

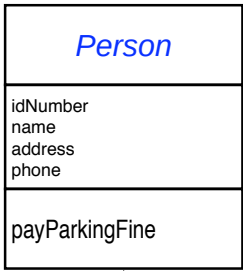
Abstract and Concrete Classes

Will there be instances of the class in the modeling world?



A **Concrete Class** indicates the existence of actual instances of this form in the modeling context while **Abstract Class** alone is a structuring tool to capture only a similarity between templates without the existence of these "abstractions" actually occurring. The italicized class name is the indicator of the abstract class. A non-italicized class name represents a concrete class for which there will be instances with values found in the modeling domain.





Why Guidelines?

In order for your documentation to be as useful as possible you need to be careful to use the diagraming discipline consistently. Diagrams are another form of "language" that you're using to communicate your ideas. If you were using a CASE tool to draw your diagrams you wouldn't be able to use bad "syntax." You might be able to make "logical" errors, but not syntax errors. If you're drawing these diagrams by hand or with a simple drawing tool you must be more careful to make your diagrams "syntactically" correct.

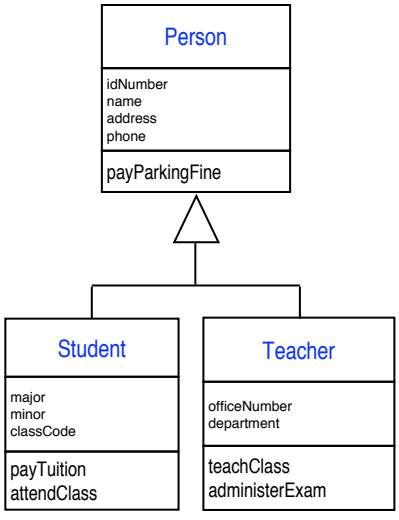
Here are some guidelines for drawing your diagrams that will make them more easily readable by your team mates and stakeholders.

- Professor Waguespack

Inheritance: Parent / Child

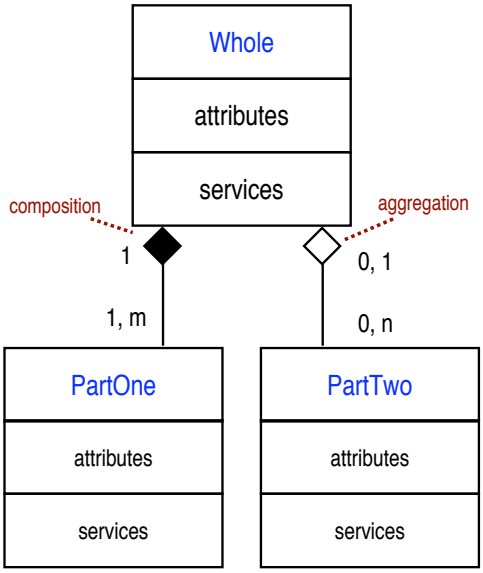
How are the classes the same and how are they different?

Generalization / Specialization is indicated with the "triangle" symbol over the connecting line between the bottom of the **parent class** (above) and the top of the **child class** (below) as shown in the insert. If there is more than one child each is hung on the horizontal connector. Inheritance flows from top to bottom. Notice that parent or child "classes" involved in "gen/spec" may themselves be either Abstract or Concrete Classes. Also notice that the relationship is between the "class" nature of these symbols rather than the "instances" that may occur.



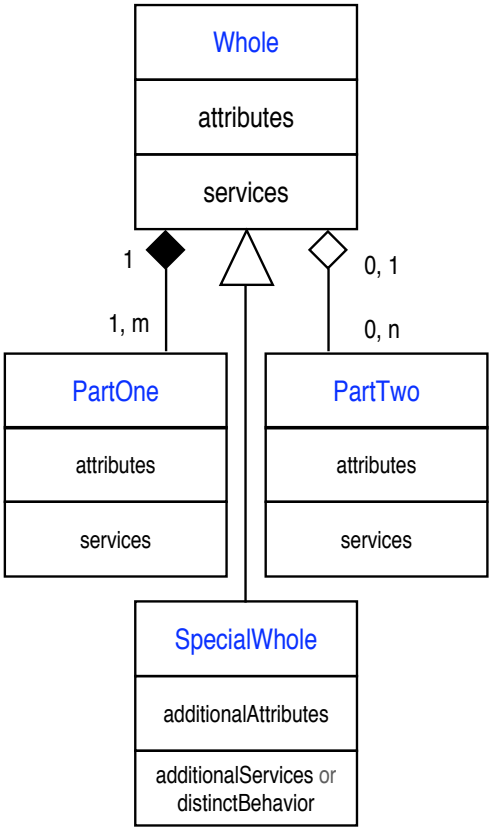
WHOLE / PART: COMPOSITION AND AGGREGATION: "OBJECTS NEED EACH OTHER!"

Although the Concrete Class depicts all the instances of the objects derived from this class in the problem space, it may be appropriate to indicate that there is some kind of collection object which is responsible for "finding," "enumerating," or "creating / destroying" individual member objects. For this purpose the diamond symbol is placed over the connecting line between the bottom of the "whole" down to the top of the "part." Whole/Part is a relationship between objects rather than classes. If this relationship carries an "existence" flavor, the diamond is black indicating a **Composition**. That is that the whole would not exist if it were not for the parts and / or vice versa. The parts "belong to" the whole and would probably not be found in the problem space except in their role as parts. If the collection relationship does not reach the level of "existence," the diamond may be left white indicating an **Aggregation**.



Therefore there must be an object to connect to in the whole and objects to play the role of parts. If the whole is not a concrete class then it must be a generalization of other concrete classes. Rather than draw each child class as a whole, you may choose to draw the abstract parent class as a whole from which the parts are connected as in the following example.

In this case the "Whole class" may own a collection of "Part One's" and "Part Two's." And



any child classes of "Whole class" would have a collection of "Part One's" and could have a collection of "Part Two's."

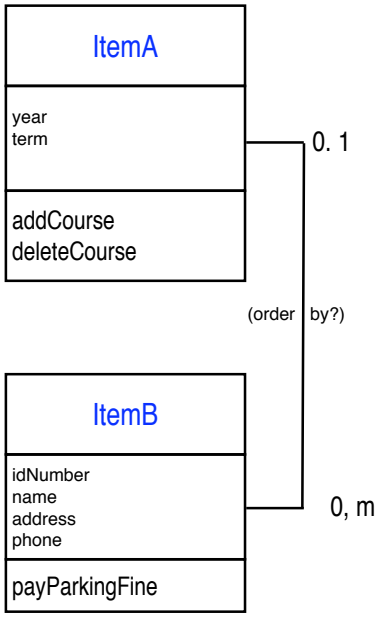
In a similar fashion, a "part" in the relationship may be an abstract rather than a concrete class. But, it must have a concrete child class down the line that would be the actual member of the "Whole's" collection. In the documentation the collection may have additional defined characteristics such as "ordered collection" or "indexed collection" indicating to the reader that the collection may be easily searched or enumerated if need be.

NOTICE that the cardinality (count constraints) on the vertical connection between whole and part indicates the constraints. The number next to the whole indicates the number of whole instances to which a part may "belong." The number next to the part is the number of

parts which may belong to each whole. The part cardinality may be [0, 1], [1, n], or [n, m] according to the business rules of the relationship. The whole cardinality in a **composition** is usually either [1] or [0,1] since a part can be "owned" by only one whole. Any exception to this should be carefully documented.

INSTANCE CONNECTIONS: "OBJECTS JUST KNOW ABOUT EACH OTHER!"

Instance Connections (also called Simple Associations) depict relationships between objects that are less restrictive than **composition** or **aggregation**. Both participants in an instance connection would "exist" regardless of the other. In this case Item A is connected to from 0 to m instances of Item B and Item B is associated with 0 or 1 Item A. [1,1], [1,m] and [m, n] are all possible including occasions when the 1's may be 0's as well. These cardinalities would be determined by the business rules of the problem space. Since neither of these objects "owns" the other the connecting line is suggested to attach at either side of each symbol clearly distinguishing the relationship from that of whole / part.



Simple Associations are usually needed to depict "awareness" of one object for another when there is no other awareness such as **composition** or **aggregation**. For instance, in order for an object to send a message to another object there must be some way of "finding" it - either through a whole/part relationship or an instance connection. When you draw message connections check to see if the sender would have a way to find the receiver to whom the message is sent.

CARDINALITY: COMPOSITION, AGGREGATION, INSTANCE CONNECTION: WHAT COUNTS?

All associations (including composition and aggregation) require explicit specification of cardinality. The counts are placed nearest the class whose cardinality is indicated. Although all three nominal cardinalities are legal (1-1, 1-m, and m-m), there are virtually no circumstances where (m-m) improves understanding of the object model. Using "normalized" cardinality (1-1, 1-m) also improves your chances of recognizing when a class is really a "container" rather than a collection of instances. For example to aggregate transcripts to students (1-1) means that each student has one transcript object. If you describe a transcript object as a record of a course with a grade, that also means that "each student takes one and only one course!" That's probably not what you intended! Being explicit every time with cardinality helps to avoid these logical oversights.

SERVICES: SERVICES BELONG IN THE OBJECTS RESPONSIBLE FOR PROVIDING THEM

Services determine the actions in an object model. The service resides in the object that is responsible for providing the behavior to the model whole. As such, service names should always be command verbs that are "spoken" by the sending object. If the action required is to

respond with some information (e.g. get a value from an attribute of the receiving object) then the service name may be seen more like the action / inquiry of the sending object (e.g. "GetChar," or "IsCharEqual") but, the service still must reside in the receiving object.

Since child classes inherit all the services and attributes of their parent class, you may have a choice of where to place a service in a "family tree." Generally, you should place the service as high in the tree as it denotes a behavior familiar to all the child classes below it. If a child class needs to perform the familiar behavior somewhat differently, then the implementation of the service will be **overridden** in that child class. But since the service name (name and any parameters needed) is identical to that of the parent class, a client object sending a message to the object will not have to "distinguish" the object's class from that of its parent. And thus the client will be exercising the **polymorphism** that the identically named services provide.

Service names reside in the class that performs that behavior. To perform a behavior the object must either have attributes or direct access to companion classes that enable accomplishing that behavior. That's why service names are present tense imperative; so you can "address the object using the service name as a command." When you place a service name in a class, ask yourself, "Does an object of this class know how to perform this behavior?" and "Is this behavior really this class's responsibility?"

Services that create objects (i.e. when a new member is created to add to the "parts" of a whole/part relationship) should be placed in objects that are superior to the objects to be created. A superior object would be one that already exists to create the new object. So, parts should not "create themselves.," but rather might be better created by their **whole**, or at least by an object that otherwise already exists. It is however, common for a new object once created by its "owner" – to have a service that initializes and populates the values of its attributes.

Services are defined in the class that is responsible for the behavior they accomplish. To that extent, the description of a service should reflect the encapsulated responsibility of the hosting object. These descriptions should not refer to the objects that invoke them and should refer to other objects only if those objects' services are needed to accomplish this object's behavior. To that end, service descriptions are quite modular and thus easily reusable because the "intent" of the object that invokes this service is not germane to the service being provided. Service descriptions do not need to be algorithmic, but they must identify "what happens" to accomplish the behavior: setting attributes, sending messages to other objects, returning calculated results, etc.

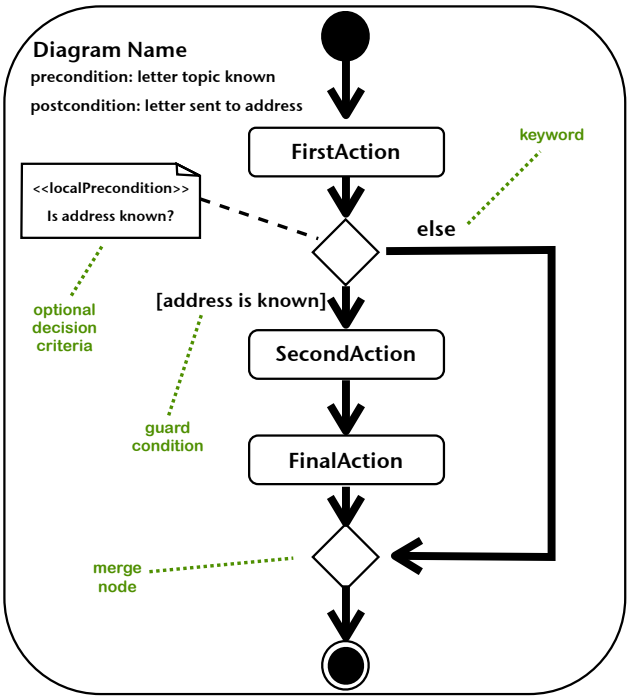
SCENARIO, USE CASE, ACTIVITY DIAGRAM, SEQUENCE DIAGRAM, OBJECT MODELS IN ANALYSIS

Scenarios are stories collected during requirements analysis that attempt to gather a "broad" understanding of the problem domain and the meaningful elements and events that exist there. Scenarios are usually text descriptions based on stakeholder interviews; stories about what's going on in the problem domain of interest. They often correspond to the direct contact with domain experts during the interviewing process. They are the source of domain experience that is distilled and formalized in the modeling process.

UML separates out class structure and relationships, object interaction through messages, and behavior externally visible to a user in several different diagrams. **Use cases** attempt to "homogenize" the terms, actions, and actors, allowing the analyst and users to converse

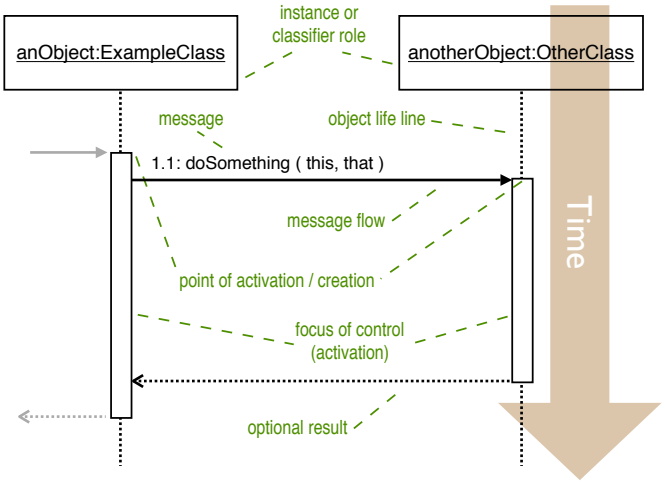
in the same "glossary" of domain labels. The actions in the use cases should be recognizable in the object model, perhaps not as a single service, but as initiated by some particular object in the model.

The **activity diagram** is a means of visualizing the actions that occur in the modeling domain.



They may represent the action sequence that accomplishes a use case from the actors' perspective or they may represent the detailed steps taken to implement the behavior that is indicated by a service name in a class definition. Activity diagrams are most easily read when the *flow of control* proceeds generally from top to bottom and from left to right in the diagram. The action nodes may represent a single atomic computation or a more complex collection of steps depending on the level of abstraction that the diagram is intended to represent.

The **sequence diagram** pulls together the objects and actions laying out the "time line" of messages/responses and (when detailed the exceptions, "extensions") that occur in completing a domain visible action. Because this is a domain descriptive modeling activity (focused primarily on requirements for behavior in the problem domain) the modeling artifacts are primarily those visible to the users (actors) and thus do not focus on files, nodes, servers, and components that might be the case in the design or implementation stage. In analysis the focus is on "what's happening in the problem domain."



In the end it should be possible to follow a **sequence diagram** to "trace" the actions of the players defined in the **class diagram** taken to result in the "actor visible outcomes" described in a **use case**. Eventually the use case becomes the final "testing" pattern to validate that the **object model** defines all the necessary objects and their interacting relationships and behaviors.