**Unified Modeling Language(UML)**

**Introduction**

* A standardized, graphical “modeling language” for communicating software design.
* Allows implementation-independent specification of:
  + user/system interactions (required behaviors)
  + partitioning of responsibility (OO)
  + integration with larger or existing systems
  + data flow and dependency
  + operation orderings (algorithms)
  + concurrent operations
* Pretty pictures.
* UML is not “process”. (That is, it doesn’t tell you how to do things, only what you should do.)

There are three classifications of UML diagrams:

* **Behavior diagrams**.  A type of diagram that depicts behavioral features of a system or business process.  This includes activity, state machine, and use case diagrams as well as the four interaction diagrams.
* **Interaction diagrams**.  A subset of behavior diagrams which emphasize object interactions.  This includes communication, interaction overview, sequence, and timing diagrams.
* **Structure diagrams**.  A type of diagram that depicts the elements of a specification that are irrespective of time.  This includes class, composite structure, component, deployment, object, and package diagrams.

|  |  |
| --- | --- |
| **Diagram** | **Description** |
| [**Activity Diagram**](http://www.agilemodeling.com/artifacts/activityDiagram.htm) | Depicts high-level business processes, including data flow, or to model the logic of complex logic within a system. |
| [**Class Diagram**](http://www.agilemodeling.com/artifacts/classDiagram.htm) | Shows a collection of static model elements such as classes and types, their contents, and their relationships. |
| [**Communication Diagram**](http://www.agilemodeling.com/artifacts/communicationDiagram.htm) | Shows instances of classes, their interrelationships, and the message flow between them. Communication diagrams typically focus on the structural organization of objects that send and receive messages.  Formerly called a Collaboration Diagram. |
| [**Component Diagram**](http://www.agilemodeling.com/artifacts/componentDiagram.htm) | Depicts the components that compose an application, system, or enterprise. The components, their interrelationships, interactions, and their public interfaces are depicted. |
| [**Composite Structure Diagram**](http://www.agilemodeling.com/artifacts/compositeStructureDiagram.htm) | Depicts the internal structure of a classifier (such as a class, component, or use case), including the interaction points of the classifier to other parts of the system. |
| [**Deployment Diagram**](http://www.agilemodeling.com/artifacts/deploymentDiagram.htm) | Shows the execution architecture of systems.  This includes nodes, either hardware or software execution environments, as well as the middleware connecting them. |
| [**Interaction Overview Diagram**](http://www.agilemodeling.com/artifacts/interactionOverviewDiagram.htm) | A variant of an activity diagram which overviews the control flow within a system or business process.   Each node/activity within the diagram can represent another interaction diagram. |
| [**Object Diagram**](http://www.agilemodeling.com/artifacts/objectDiagram.htm) | Depicts objects and their relationships at a point in time, typically a special case of either a class diagram or a communication diagram. |
| [**Package Diagram**](http://www.agilemodeling.com/artifacts/packageDiagram.htm) | Shows how model elements are organized into packages as well as the dependencies between packages. |
| [**Sequence Diagram**](http://www.agilemodeling.com/artifacts/sequenceDiagram.htm) | Models the sequential logic, in effect the time ordering of messages between classifiers. |
| [**State Machine Diagram**](http://www.agilemodeling.com/artifacts/stateMachineDiagram.htm) | Describes the states an object or interaction may be in, as well as the transitions between states. Formerly referred to as a state diagram, state chart diagram, or a state-transition diagram. |
| [**Timing Diagram**](http://www.agilemodeling.com/artifacts/timingDiagram.htm) | Depicts the change in state or condition of a classifier instance or role over time.  Typically used to show the change in state of an object over time in response to external events. |
| [**Use Case Diagram**](http://www.agilemodeling.com/artifacts/useCaseDiagram.htm) | Shows use cases, actors, and their interrelationships. |

**Use Case Diagram**

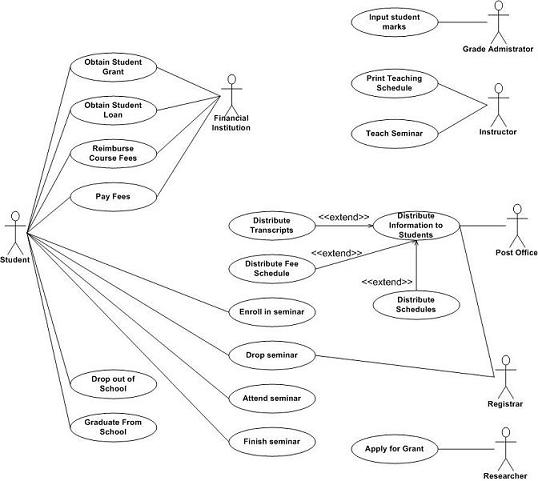
UML 2 use case diagrams overview the usage requirements for a system.

They are useful for presentations to management and/or project stakeholders, but for actual development you will find that use casesprovide significantly more value because they describe "the meat" of the actual requirements.

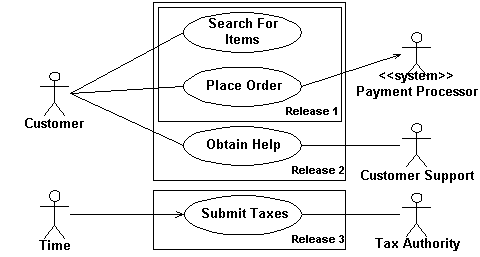
Use case diagrams depict:

* **Use cases**. A use case describes a sequence of actions that provide something of measurable value to an actor and is drawn as a horizontal ellipse.
* **Actors**. An actor is a person, organization, or external system that plays a role in one or more interactions with your system. Actors are drawn as stick figures.
* **Associations**.  Associations between actors and use cases are indicated in use case diagrams by solid lines. An association exists whenever an actor is involved with an interaction described by a use case.  Associations are modeled as lines connecting use cases and actors to one another, with an optional arrowhead on one end of the line. The arrowhead is often used to indicating the direction of the initial invocation of the relationship or to indicate the primary actor within the use case.  The arrowheads are typically confused with data flow and as a result I avoid their use.
* **System boundary boxes (optional)**. Draw a rectangle around the use cases, called the system boundary box, to indicates the scope of your system.  Anything within the box represents functionality that is in scope and anything outside the box is not.  System boundary boxes are rarely used, although on occasion I have used them to identify which use cases will be delivered in each major release of a system.  [**Figure 2**](http://www.agilemodeling.com/artifacts/useCaseDiagram.htm#Figure2SystemBoundaryBoxes) shows how this could be done.
* **Packages (optional)**.  Packages are UML constructs that enable you to organize model elements (such as use cases) into groups. Packages are depicted as file folders and can be used on any of the UML diagrams, including both use case diagrams and class diagrams. I use packages only when my diagrams become unwieldy, which generally implies they cannot be printed on a single page, to organize a large diagram into smaller ones.  [**Figure 3**](http://www.agilemodeling.com/artifacts/useCaseDiagram.htm#Figure3Packages) depicts how [**Figure 1**](http://www.agilemodeling.com/artifacts/useCaseDiagram.htm#Figure1) could be reorganized with packages.

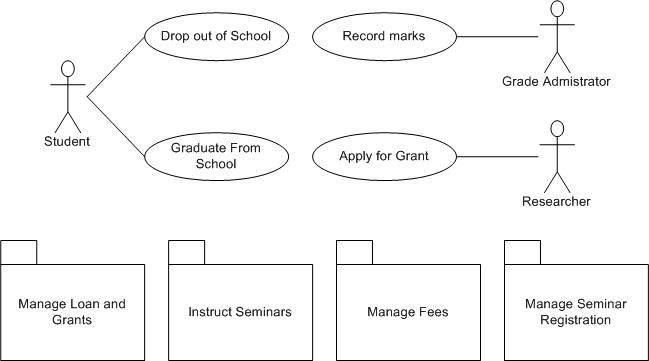
**Figure 1. System use case diagram.**



**Figure 2. Using System boundary boxes to indicate releases.**



**Figure 3. Applying packages to simplify use case diagrams.**



**Creating Use Case Diagrams**

Start by identifying as many actors as possible.

You should ask how the actors interact with the system to identify an initial set of use cases.

Then, on the diagram, you connect the actors with the use cases with which they are involved.

If an actor supplies information, initiates the use case, or receives any information as a result of the use case, then there should be an association between them.

Identify similarities between use cases, or between actors, and modeling the appropriate relationships between.

**Use Case Relationships**

**Extends: <<extend>>**

An extend dependency, formerly called an extends relationship in [**UML**](http://www.agilemodeling.com/essays/umlDiagrams.htm) v1.2 and earlier, is a generalization relationship where an extending [**use case**](http://www.agilemodeling.com/artifacts/systemUseCase.htm) continues the behavior of a base use case. The extending use case accomplishes this by conceptually inserting additional action sequences into the base use-case sequence.

This allows an extending use case to continue the activity sequence of a base use case when the appropriate extension point is reached in the base use case and the extension condition is fulfilled. When the extending use case activity sequence is completed, the base use case continues.

In the use case diagram in figure 4 , the use case “Perform” extends the use case “Enroll in University,” the notation for doing so is simply a normal use-case dependency with the stereotype of <<extend>>. In this case, “Enroll in University” is the base use case and “Perform Security Check” is the extending use case.

**Includes: <<include>>**

An include dependency, formerly known as a uses relationship in UML v1.2 and earlier, is a generalization relationship denoting the inclusion of the behavior described by another use case.

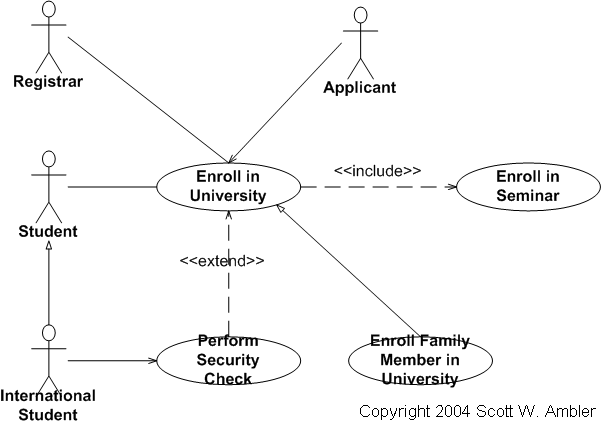
The best way to think of an include dependency is that it is the invocation of a use case by another one. In Figure 4 , you see the use case “Enroll in University” includes the use case “Enroll in Seminar,” the notation for doing so is simply a normal use-case dependency with the stereotype of <<include>>.

**Inheritance**

Use cases can inherit from other use cases, offering a third opportunity to indicate potential reuse. Figure 4 depicts an example of this, showing that “Enroll Family Member in University” inherits from the “Enroll In University” use case.

Inheritance between use cases is not as common as either the use of extend or include dependencies, but it is still possible. The inheriting use case would completely replace one or more of the courses of action of the inherited use case.

**Figure 4 shows the three types of relationships between use cases**



**Classes**

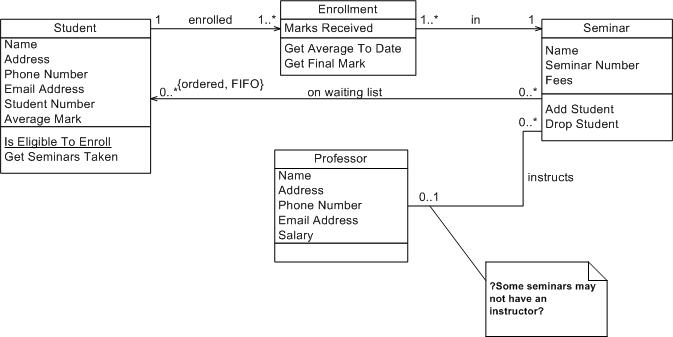
An object is any person, place, thing, concept, event, screen, or report applicable to your system.

Objects both know things (they have attributes) and they do things (they have methods).

A class is a representation of an object and, in many ways, it is simply a template from which objects are created.

Classes form the main building blocks of an object-oriented application.  Although thousands of students attend the university, you would only model one class, called *Student*, which would represent the entire collection of students.

**Figure 2. Initial conceptual class diagram.**



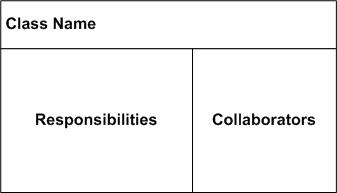
Classes are typically modeled as rectangles with three sections:

* The top section for the name of the class.
* The middle section for the attributes of the class.
* The bottom section for the methods of the class.

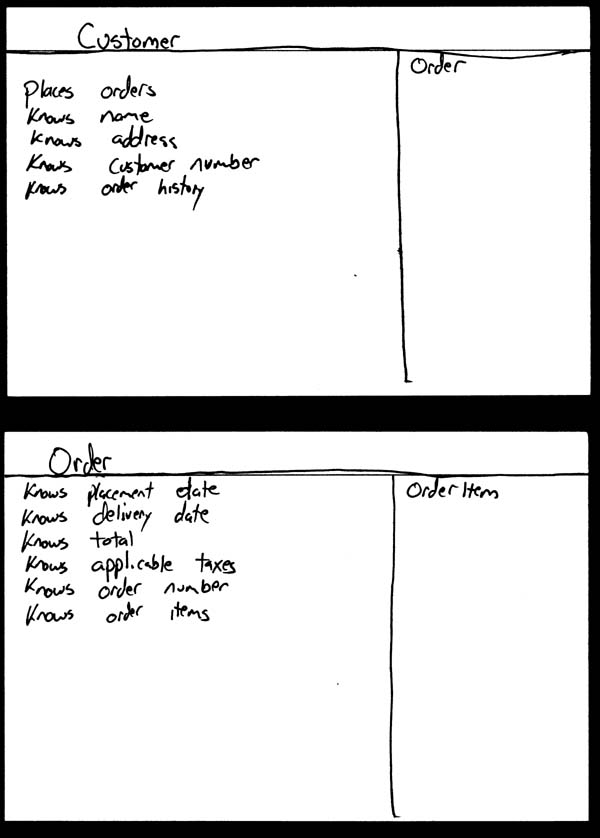
The initial classes of your model can be identified in the same manner as they are when you are Class Responsibility Collaborator (CRC) modeling , as will the initial responsibilities (its attributes and methods).

Example of CRC Modeling:

**Figure 1. CRC Card Layout.**



**Figure 2. Hand-drawn CRC Cards.**

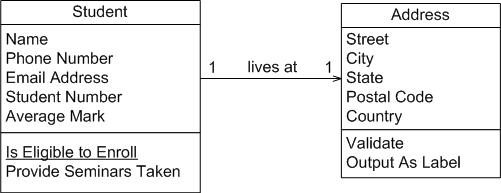


Attributes are the information stored about an object (or at least information  temporarily maintained about an object), while methods are the things an object or class do.

For example, students have student numbers, names, addresses, and phone numbers. Those are all examples of the attributes of a student. Students also enroll in courses, drop courses, and request transcripts. Those are all examples of the things a student does, which get implemented (coded) as methods. You should think of methods as the object-oriented equivalent of functions and procedures.

An important consideration the appropriate level of detail. Consider the *Student* class modeled in [Figure 2](http://www.agilemodeling.com/artifacts/classDiagram.htm#Figure2ConceptualDiagram) which has an attribute called *Address*. When you stop and think about it, addresses are complicated things. They have complex data, containing street and city information for example, and they potentially have behavior. An arguably better way to model this is depicted in [Figure 4](http://www.agilemodeling.com/artifacts/classDiagram.htm#Figure4StudentAddress). Notice how the *Address* class has been modeled to include an attribute for each piece of data it comprises and two methods have been added: one to verify it is a valid address and one to output it as a label (perhaps for an envelope). By introducing the *Address* class, the *Student* class has become more cohesive. It no longer contains logic (such as validation) that is pertinent to addresses. The *Address* class could now be reused in other places, such as the *Professor* class, reducing your overall development costs. Furthermore, if the need arises to support students with several addressesduring the school term, a student may live in a different location than his permanent mailing address, such as a dorminformation the system may need to track. Having a separate class to implement addresses should make the addition of this behavior easier to implement.

**Figure 4. Student and address (Conceptual class diagram).**



An interesting feature of the *Student* class is its *Is Eligible to Enroll* responsibility.  The underline indicates that this is a class-level responsibility, not an instance-level responsibility (for example *Provide Seminars Taken*).  A good indication that a responsibility belongs at the class level is one that makes sense that it belongs to the class but that doesn’t apply to an individual object of that class.  In this case this operation implements BR129 *Determine Eligibility to Enroll* called out in the *Enroll in Seminar* system use case.

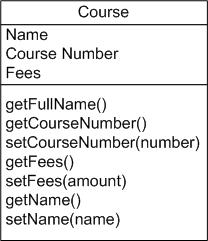
The *Seminar* class of [Figure 2](http://www.agilemodeling.com/artifacts/classDiagram.htm#Figure2ConceptualDiagram) is refactored into the classes depicted in [Figure 5](http://www.agilemodeling.com/artifacts/classDiagram.htm#Figure5SeminarNormalized). Refactoring such as this is called class normalization ([Ambler 2004](http://www.ambysoft.com/agileDatabaseTechniques.html)), a process in which you refactor the behavior of classes to increase their cohesion and/or to reduce the coupling between classes. A seminar is an offering of a course, for example, there could be five seminar offerings of the course "CSC 148 Introduction to Computer Science."  The attributes *name* and*fees* where moved to the *Course* class and *courseNumber* was introduced. The *getFullName()* method concatenates the course number, "CSC 148" and the course name "Introduction to Computer Science" to give the full name of the course. This is called a getter method, an operation that returns a data value pertinent to an object. Although getter methods, and the corresponding setter methods, need to be developed for a class they are typically assumed to exist and are therefore not modeled (particularly on conceptual class diagrams) to not clutter your models.

**Figure 5. Seminar normalized (Conceptual class diagram).**



[Figure 6](http://www.agilemodeling.com/artifacts/classDiagram.htm#Figure6Course) depicts *Course* from [Figure 5](http://www.agilemodeling.com/artifacts/classDiagram.htm#Figure5SeminarNormalized) as it would appear with its getter and setter methods modeled. Getters and setters are details that are not appropriate for conceptual models and in my experience aren’t even appropriate for detailed design diagrams – instead I would set a coding guideline that all properties will have getter and setter methods and leave it at that. Some people do choose to model getters and setters but I consider them visual noise that clutter your diagrams without adding value.

**Figure 6. Course with accessor methods (Inching towards a design class diagram).**



**Associations**

Objects are often associated with, or related to, other objects. For example, as you see in Figure 2 several associations exist: Students are ON WAITING LIST for seminars, professors INSTRUCT seminars, seminars are an OFFERING OF courses, a professor LIVES AT an address, and so on. Associations are modeled as lines connecting the two classes whose instances (objects) are involved in the relationship.

When you model associations in UML class diagrams, you show them as a thin line connecting two classes, as you see in [Figure 6](http://www.agilemodeling.com/artifacts/classDiagram.htm#Figure6AssociationNotation). Associations can become quite complex; consequently, you can depict some things about them on your diagrams. The label, which is optional, although highly recommended, is typically one or two words describing the association. For example, professors instruct seminars.

**Figure 6. Notation for associations.**

http://www.agilemodeling.com/images/models/classDiagramAssocationNotation.jpg

It is not enough simply to know professors instruct seminars. How many seminars do professors instruct? None, one, or several? Furthermore, associations are often two-way streets: not only do professors instruct seminars, but also seminars are instructed by professors. This leads to questions like: how many professors can instruct any given seminar and is it possible to have a seminar with no one instructing it? The implication is you also need to identify the multiplicity of an association. The multiplicity of the association is labeled on either end of the line, one multiplicity indicator for each direction ([Table 1](http://www.agilemodeling.com/artifacts/classDiagram.htm#Table1MultiplicityIndicators) summarizes the potential multiplicity indicators you can use).

Table 1. Multiplicity Indicators.

|  |  |
| --- | --- |
| Indicator | Meaning |
| 0..1 | Zero or one |
| 1 | One only |
| 0..\* | Zero or more |
| 1..\* | One or more |
| n | Only *n* (where *n* > 1) |
| 0..n | Zero to *n* (where *n* > 1) |
| 1..n | One to *n* (where *n*> 1) |

Another option for associations is to indicate the direction in which the label should be read. This is depicted using a filled triangle, called a direction indicator, an example of which is shown on the *offering of* association between the *Seminar* and *Course* classes of [Figure 5](http://www.agilemodeling.com/artifacts/classDiagram.htm#Figure5SeminarNormalized). This symbol indicates the association should be read “a seminar is an offering of a course,” instead of “a course is an offering of a seminar.” Direction indicators should be used whenever it isn’t clear which way a label should be read. My advice, however, is if your label is not clear, then you should consider rewording it.

The arrowheads on the end of the line indicate the directionality of the association.  A line with one arrowhead is uni-directional whereas a line with either zero or two arrowheads is bidirectional.  Officially you should include both arrowheads for bi-directional assocations, however, common practice is to drop them (as you can see, I prefer to drop them).

At each end of the association, the role, the context an object takes within the association, may also be indicated. My style is to model the role only when the information adds value, for example, knowing the role of the *Student*class is enrolled student in the enrolled in association doesn’t add anything to the model. I follow the AM practice [Depict Models Simply](http://www.agilemodeling.com/practices.htm#DepictModelsSimply) and indicate roles when it isn’t clear from the association label what the roles are, if there is a recursive association, or if there are several associations between two classes.

**Inheritance Relationships**

Similarities often exist between different classes.

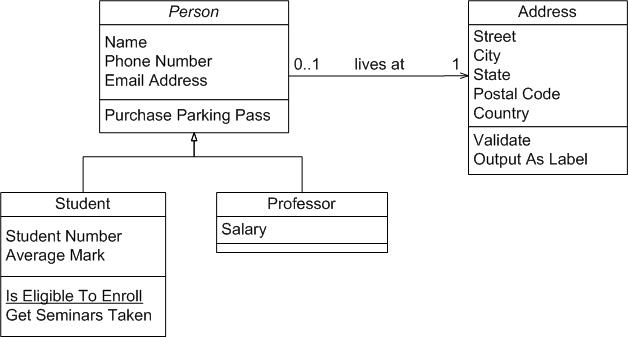
Very often two or more classes will share the same attributes and/or the same methods. Because you don’t want to have to write the same code repeatedly, you want a mechanism that takes advantage of these similarities. Inheritance is that mechanism.

Inheritance models “is a” and “is like” relationships, enabling you to reuse existing data and code easily.

When *A* inherits from *B,* we say *A* is the subclass of *B* and *B* is the superclass of *A.* Furthermore, we say we have “pure inheritance” when *A* inherits all the attributes and methods of *B.* The UML modeling notation for inheritance is a line with a closed arrowhead pointing from the subclass to the superclass.

Many similarities occur between the *Student* and *Professor* classes of Figure 2. Not only do they have similar attributes, but they also have similar methods. To take advantage of these similarities, I created a new class called*Person* and had both *Student* and *Professor* inherit from it, as you see in [Figure 7](http://www.agilemodeling.com/artifacts/classDiagram.htm#Figure7InheritanceHierarchy).  This structure would be called the *Person* inheritance hierarchy because *Person* is its root class. The *Person* class is abstract: objects are not created directly from it, and it captures the similarities between the students and professors. Abstract classes are modeled with their names in italics, as opposed to concrete classes, classes from which objects are instantiated, whose names are in normal text. Both classes had a name, e-mail address, and phone number, so these attributes were moved into *Person*. The *Purchase Parking Pass* method is also common between the two classes, something we discovered after [Figure 2](http://www.agilemodeling.com/artifacts/classDiagram.htm#Figure2ConceptualDiagram)was drawn, so that was also moved into the parent class. By introducing this inheritance relationship to the model, I reduced the amount of work to be performed. Instead of implementing these responsibilities twice, they are implemented once, in the *Person* class, and reused by *Student* and *Professor*.

**Figure 7. Inheritance hierarchy.**

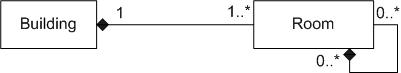


**Composition Associations**

Sometimes an object is made up of other objects.

For example, an airplane is made up of a fuselage, wings, engines, landing gear, flaps, and so on. Figure 8 presents an example using composition, modeling the fact that a building is composed of one or more rooms, and then, in turn, that a room may be composed of several subrooms (you can have recursive composition).  In UML 2, aggregation would be shown with an open diamond.

**Figure 8. Modeling composition.**



**UML sequence diagrams**

UML sequence diagrams model the flow of logic within your system in a visual manner, enabling you both to document and validate your logic, and are commonly used for both analysis and design purposes.

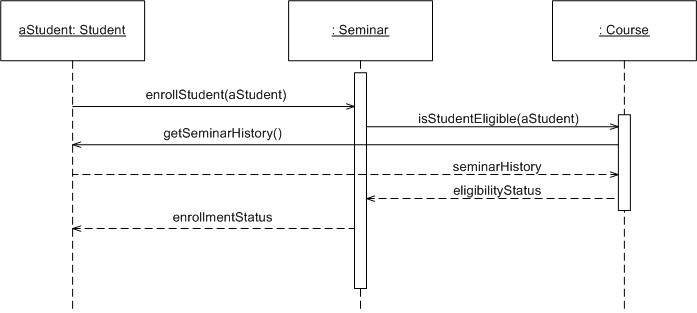
Sequence diagrams are the most popular UML artifact for dynamic modeling, which focuses on identifying the behavior within your system.

Sequence diagrams, along with [**class diagrams**](http://www.agilemodeling.com/artifacts/classDiagram.htm) and [**physical data models**](http://www.agiledata.org/essays/dataModeling101.html) are the most important design-level models for modern business application development.

Sequence diagrams are typically used to model:

* **Usage scenarios**.  A usage scenario is a description of a potential way your system is used. The logic of a usage scenario may be part of a use case, perhaps an alternate course. It may also be one entire pass through a use case, such as the logic described by the basic course of action or a portion of the basic course of action, plus one or more alternate scenarios. The logic of a usage scenario may also be a pass through the logic contained in several use cases. For example, a student enrolls in the university, and then immediately enrolls in three seminars.
* **The logic of methods**.   Sequence diagrams can be used to explore the logic of a complex operation, function, or procedure.  One way to think of sequence diagrams, particularly highly detailed diagrams, is as [**visual object code**](http://www.agilemodeling.com/artifacts/sequenceDiagram.htm#VisualCoding).
* **The logic of services**.  A service is effectively a high-level method, often one that can be invoked by a wide variety of clients.  This includes web-services as well as business transactions implemented by a variety of technologies such as CICS/COBOL or CORBA-compliant object request brokers (ORBs).

Example:



**Elements of a sequence diagram**

* The boxes across the top

The boxes across the top of the diagram represent classifiers or their instances, typically use cases, objects, classes, or actors.

Because messages can be sent to both objects and classes, objects respond to messages through the invocation of an operation and classes do so through the invocation of static operations, it makes sense to include both on sequence diagrams.

Because actors initiate and take an active part in usage scenarios, they can also be included in sequence diagrams.

* Object Lifelines

The dashed lines hanging from the boxes are called object lifelines, representing the life span of the object during the scenario being modeled.

* Activation Boxes

The long, thin boxes on the lifelines are activation boxes, also called method-invocation boxes, which indicate processing is being performed by the target object/class to fulfill a message.

* Labeled Arrows(Messages)

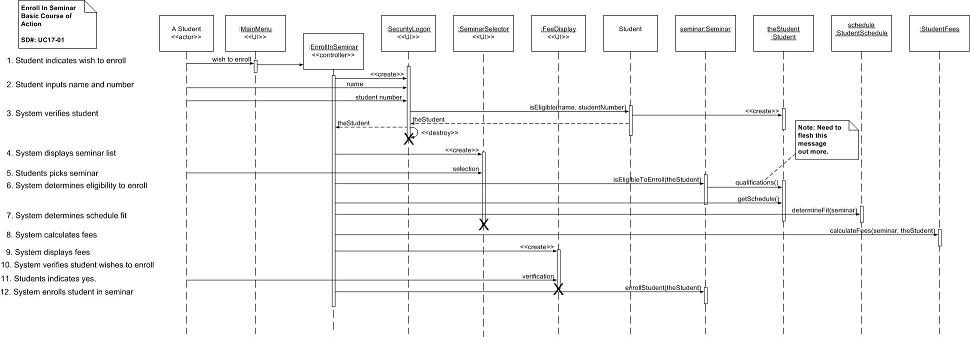
Messages are indicated on UML sequence diagrams as labeled arrows, when the source and target of a message is an object or class the label is the signature of the method invoked in response to the message.

However, if either the source or target is a human actor, then the message is labeled with brief text describing the information being communicated.

* *X*

The *X* at the bottom of an activation box, is a UML convention to indicate an object has been removed from memory.

[view](UML%20-%20Sequence%20diagram.jpg)

[](UML%20-%20Sequence%20diagram.jpg)

Reference: <http://www.agilemodeling.com/essays/umlDiagrams.htm>