

Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a diagramming language to specify, visualize and document models of Object Oriented software systems. UML is not a development method, it does not tell you how to implement your software, but it helps you to visualize your design and communicate with others. UML is controlled by the Object Management Group (OMG <http://www.omg.org/>) and is the industry standard for graphically describing software. UML is designed for Object Oriented software design and has limited use for other programming paradigms.

UML is composed of many model elements that represent the different parts of a software system. Some of these are:

- **Use Case Diagrams** show actors (people or other users of the system), use cases (the scenarios when they use the system), and their relationships
- **Class Diagrams** show classes and the relationships between them
- **Sequence Diagrams** show objects and a sequence of method calls they make to other objects.
- **Collaboration Diagrams** show objects and their relationship, putting emphasis on the objects that participate in the message exchange
- **State Diagrams** show states, state changes and events in an object or a part of the system
- **Activity Diagrams** show activities and the changes from one activity to another with the events occurring in some part of the system
- **Component Diagrams** show the high level programming components
- **Deployment Diagrams** show the instances of the components and their relationships.
- **Entity Relationship Diagrams** show data and the relationships and constraints between the data.

This is a huge area, but we focus on Use Cases, Class Diagrams, Sequence Diagrams, and State Diagrams.

Use-Case is an artifact that captures the system behavior to yield an observable result of value to those who interact with the system.

A use case typically includes the following information:

Name: The name of the use case

Brief Description: A brief description of the role and purpose of the use case

Flow of Events: A textual description of what the system does in regard to a use case scenario (not how specific problems are solved by the system). Write the description so that the customer can understand it. The flows can include a basic flow, alternative flows, and subflows.

Key scenarios: A textual description of the most important or frequently discussed scenarios

Special Requirements: A textual description that collects all of the requirements of the use case that are not considered in the use-case model, but that must be taken care of during design or implementation (for example, non-functional requirements)

Preconditions: A textual description that defines a constraint on the system when the use case starts


Post-conditions: A textual description that defines a constraint on the system when the use case ends

Extension points: A list of locations within the flow of events of the use case at which additional behavior can be inserted by using the extend-relationship.

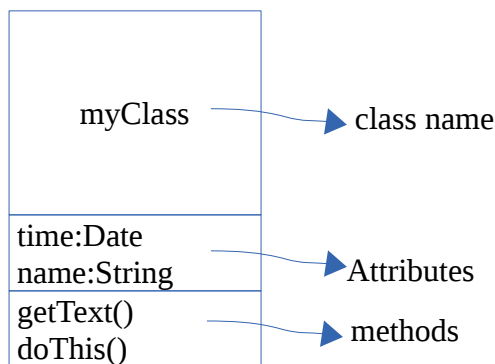
Class Diagrams

Unlike in Java, where type precedes a variable, in UML type follows attribute (variable,Field),
attribute:Type

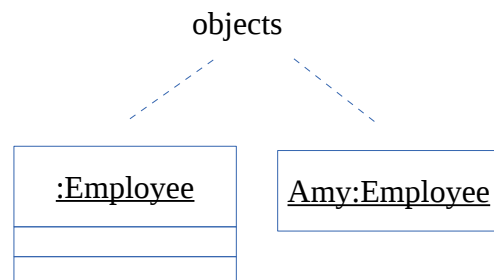
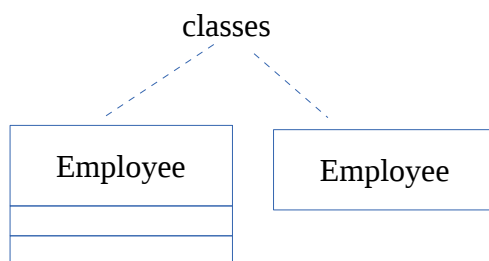
Examples

reason:String
value:Float
getMessage(n:Int):Text 

Representation of classes



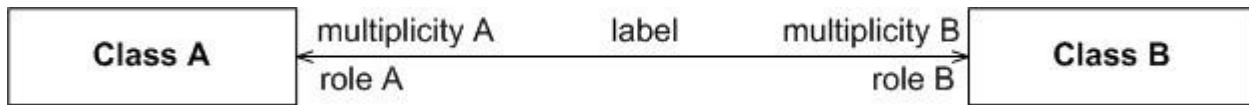
An object is the instantiation of a class - it is the object created from the class at runtime. **Object diagrams** are similar to class diagrams, except the object name and class they are instantiated from are underlined.



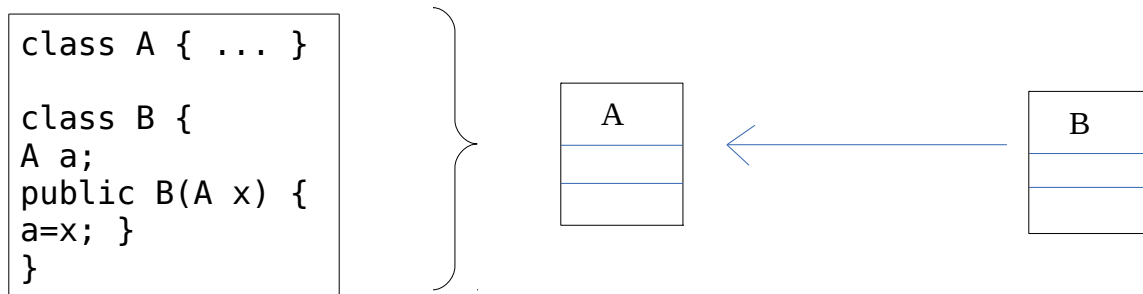
Relationships between classes

Association

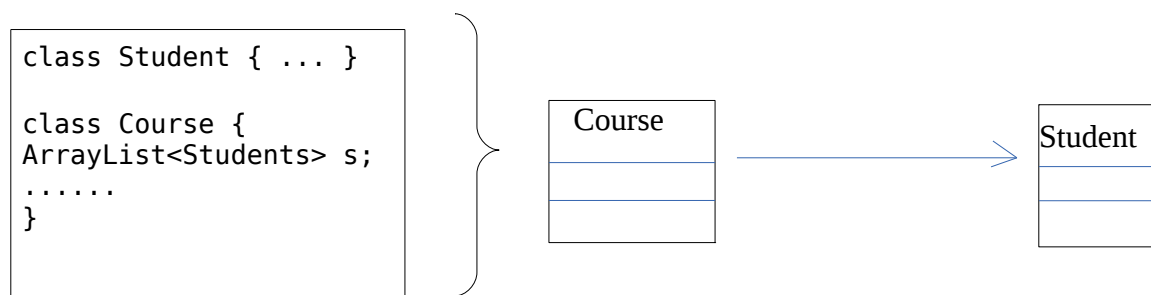
Relation between classes, with roles explicitly stated



Association is reference based relationship between two classes. Association by definition does not suggest a dependency.



The arrow shows the direction you can navigate (from class B you get to class A). This is unidirectional Association relationship.



If Course references Student, and Student references Course, the relationship is a bi-directional association.

Composition and **Aggregation** indicate something more about the association.

Composition indicates that one class belongs to the other. A polygon is made up of several points. If the polygon is destroyed, so are the points. (life-cycle dependency)

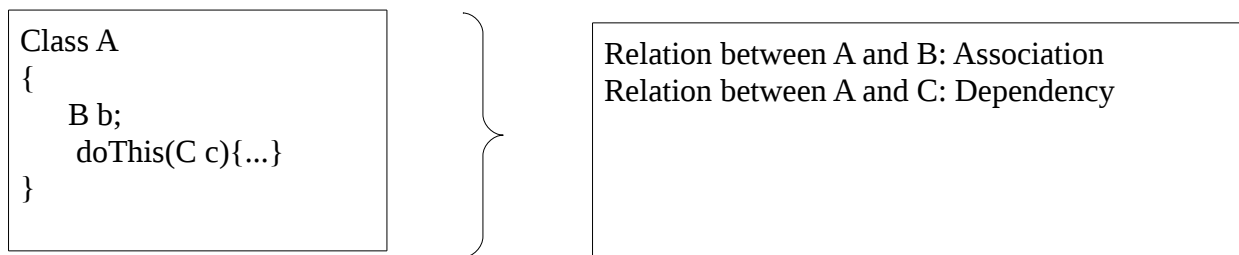
Aggregation ("has") is similar to composition, but it is a less rigorous way of grouping things. An order is made up of several products, but a product continues to exist even if the order is destroyed. (No life-cycle dependency).

A message queue aggregates messages, so a MessageQueue class aggregates Message class. Foundational types contained in a class are considered attributes rather than aggregations. If a Message class has an instance of Date class called timestamp, then timestamp is considered an attribute.

Dependency ("uses")









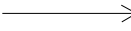
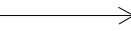




A class depends on another class if it manipulates objects of the other class. Dependency exists between two classes if a change in one may cause change to other.

Dependency is a weaker form of relationship which indicates that one class depends on another because it uses it at some point of time. One class depends on another if the latter is a parameter variable or local variable of a method of the former. This is different from an association, where an attribute of the former is an instance of the latter.



Inheritance ("is")

A class inherits from another if it incorporates the behavior of the other class.

	Dependency	this class is dependent upon  this class
	Aggregation	this class aggregates this  class
	Composition/ containment	This class is composed of  this class
	Association	This class is associated with this  class
	Directed association	From this class you can get to this  class
	Inheritance	This class inherits from  this class
	interface implementation	This class implements  this interface