UNIFIED MODELING LANGUAGE

UML Views & Diagrams

There are five different views that the UML aims to visualize through different modeling diagrams. These five views are:

1. User's View
2. Structural Views
3. Behavioral Views
4. Environmental View
5. Implementation View

Now, these views just provide the thinking methodology and expectations (that people have formed the software) of different groups of people. However, we still need to diagrammatically document the system requirements, and this is done through 9 different types of UML diagrams. These diagrams are divided into different categories. These categories are nothing else but these views that we mentioned earlier.

The different software diagrams according to the views they implement are:

1) User's view

This contains the diagrams in which the user's part of interaction with the software is defined. No internal working of the software is defined in this model. The diagrams contained in this view are:

- Use case Diagram

2) Structural view

In the structural view, only the structure of the model is explained. This gives an estimate of what the software consists of. However, internal working is still not defined in this model. The diagram that this view includes are:

- Class Diagrams
- Object Diagrams

3) Behavioral view

The behavioral view contains the diagrams which explain the behavior of the software. These diagrams are:

- Sequence Diagram
- Collaboration Diagram
- State chart Diagram
- Activity Diagram

4) Environmental view

The environmental view contains the diagram which explains the after deployment behavior of the software model. This diagram usually explains the user interactions and software effects on the system. The diagrams that the environmental model contain are:

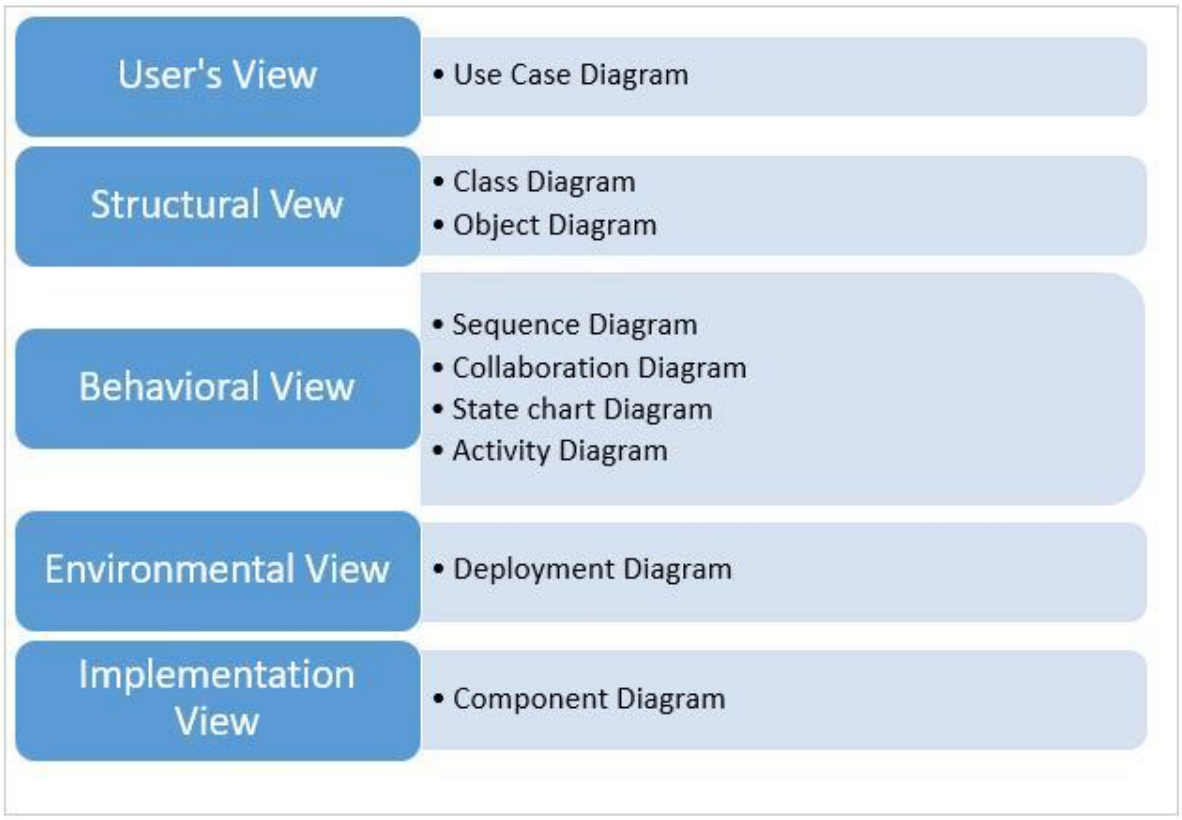
- Deployment diagram

5) Implementation view

The implementation view consists of the diagrams which represent the implementation part of the software. This view is related to the developer’s views. This is the only view in which the internal workflow of the software is defined. The diagram that this view contains is as follows:

- Component Diagram

The following diagram well explains the distribution of these 9 software diagrams according to the five mentioned views:



Object and Class Concepts

Objects--

The purpose of class modeling is to describe objects. For example , ***Joe Smith. Simplex company, process number 7648*** and ***the top window*** are objects.

An ***object*** is a concept, abstraction, or thing with identity that has meaning for an application.

Classes--

An object is an ***instance*** of a class. A ***class*** describes a group of objects with the same properties (attributes), behavior (operations), kinds of relationships, and semantics. ***Person, company, process,*** and ***window*** are all classes.

Class Diagrams--

There are two kinds of models of structure—classes diagrams and object diagrams.

Class diagrams provide a graphic notation for modeling classes and their relationships, thereby describing possible objects. Class diagrams are useful both for abstract modeling and for designing actual programs. An ***object diagram*** shows individual objects and their relationships. Figure shows a class (left) and instances (right) described by it.

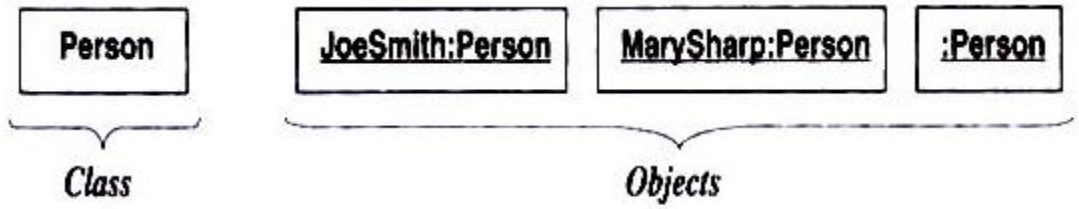


Figure **A class and objects.** Objects and classes are the focus of class modeling.

Values and Attributes--

A ***value*** is a piece of data. An ***attribute*** is a named property of a class that describes a value held by each object of the class. ***Name, birthdate,*** and ***weight*** arc attributes of ***Person*** objects. ***Color, model Year,*** and ***weight*** are attributes of ***Car*** objects. Each attribute has a value for each object.

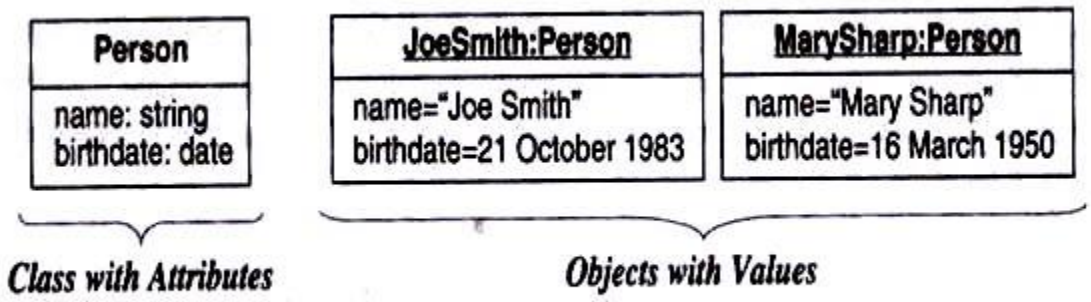


Figure **Attributes and values.** Attributes elaborate classes.

Operations and Methods--

An ***operation*** is a function or procedure that may be applied to or by objects in a class. ***Hire, fire,*** and ***payDividend*** are operations on class ***Company. Open, close, hide,*** and ***redisplay*** are operations on class ***Window.*** A ***method*** is the implementation of an operation for a class. For example, the class ***File*** may have an operation ***print.***

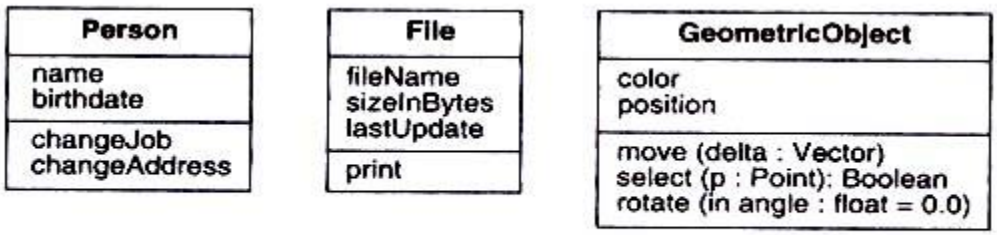


Figure **Operations.** An operation is a function or procedure that may be applied to or by objects in a class.

UML Class Notation--

A class represent a concept which encapsulates state (**attributes**) and behavior (**operations**). Each attribute has a type. Each **operation** has a **signature**. *The class name is the* ***only mandatory information***.

Class Name:

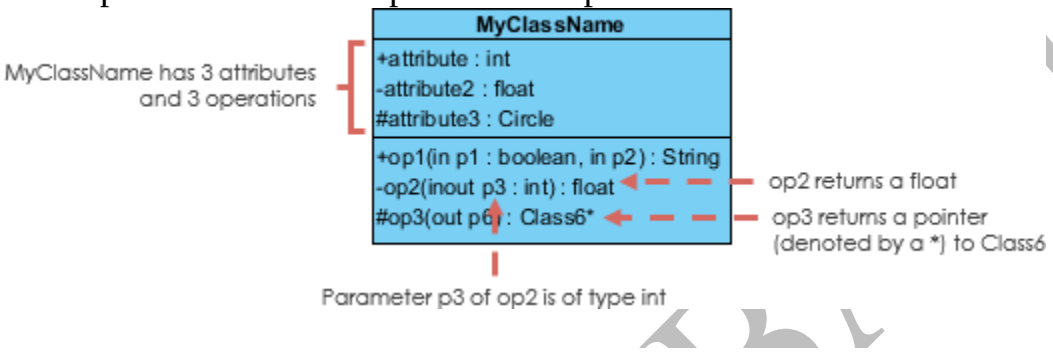
- The name of the class appears in the first partition.

Class Attributes:

- Attributes are shown in the second partition.
- The attribute type is shown after the colon.
- Attributes map onto member variables (data members) in code.

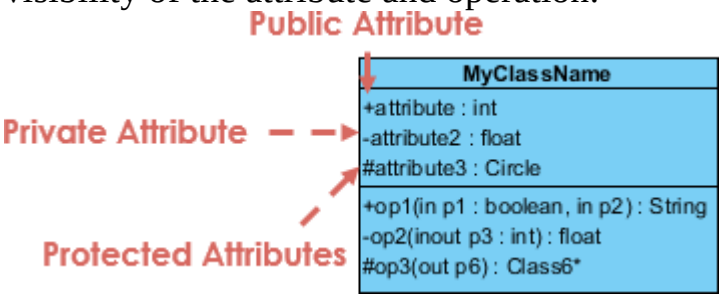
Class Operations (Methods):

- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code



Class Visibility

The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.



- + denotes public attributes or operations
- - denotes private attributes or operations
- # denotes protected attributes or operations

Links and Associations--A ***link*** is a physical or conceptual connection among objects. For example, Joe Smith ***Works-For*** Simplex company. A link is an instance of an association. An ***association*** is a description of a group of links with common structure and common semantics. For example ,a person ***WorksFor*** a company.

Multiplicity--***Multiplicity*** specifies the number of instances of one class that may relate to a single instance of an associated class. Multiplicity constrains the number of related objects. The UML specifies multiplicity with an interval, such as 1" (exactly one), "1...*" (one or more), or "3...5" (three to five, inclusive). The special symbol "*" is a shorthand notation that denotes "many" (zero or more).

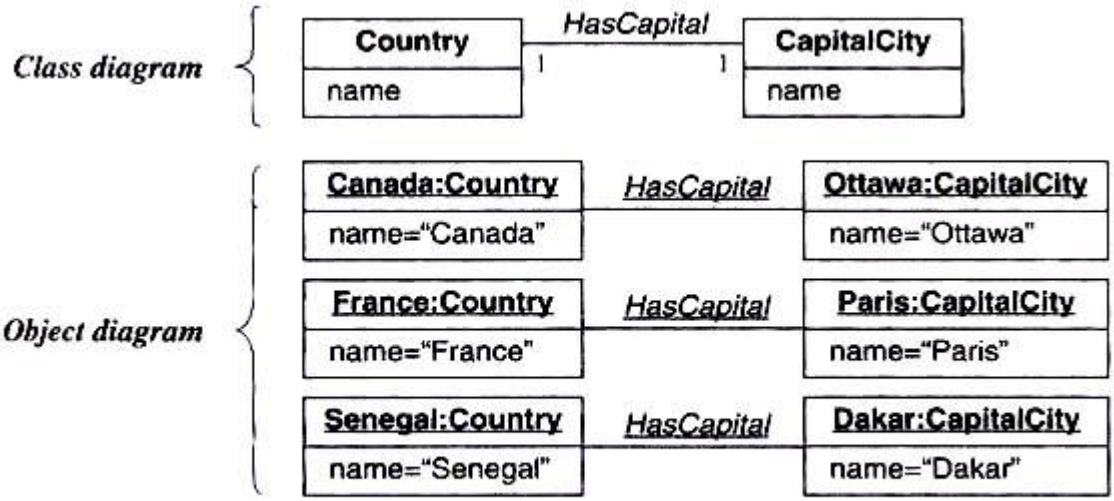
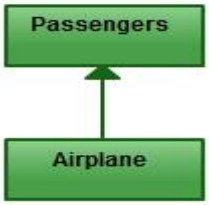


Figure **One-to-one association.** Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class.

Figure illustrates zero-or-one multiplicity. A workstation may have one of its windows designated as the console to receive general error messages.

Figure **Zero-or-one multiplicity**. It may be optional whether an object is involved in an association.

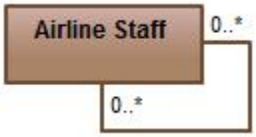
Directed Association



Directed Association

It refers to a directional relationship represented by a line with an arrowhead. The arrowhead depicts a container-contained directional flow.

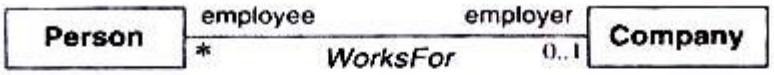
Reflexive Association



Reflexive Association

This occurs when a class may have multiple functions or responsibilities. For example, a staff member working in an airport may be a pilot, aviation engineer, a ticket dispatcher, a guard, or a maintenance crew member. If the maintenance crew member is managed by the aviation engineer there could be a managed by relationship in two instances of the same class.

Association End Names--The name given to the association end. In the figure ***Person*** and ***Company*** participate in association ***WorksFor.*** A person is an ***employee*** with respect to a company; a company is an ***employer*** with respect to a person. Use of association end names is optional.



employee	employer
Joe Doe	Simplex
Mary Brown	Simplex
Jean Smith	United Widgets

Figure **Association end names**. Each end of an association can have a name.

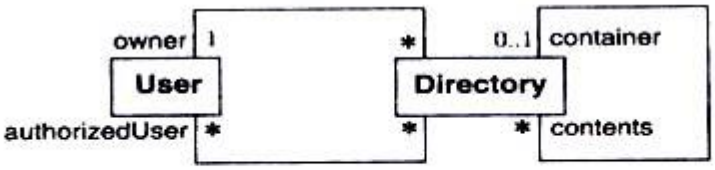


Figure **Association end names**. Association end names are necessary for associations between two objects of the same class. They can also distinguish multiple associations between a pair of classes.

Ordering--

The objects have an explicit order. The ordering is an inherent part of the association. We can indicate an ordered set of objects by writing "{ordered}" next to the appropriate association end.

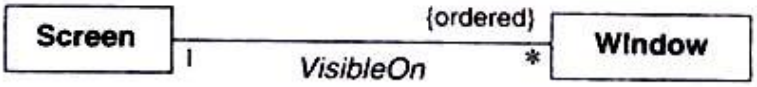


Figure **Ordering the objects for an association end**. Ordering sometimes occurs for "many" multiplicity.

Bags and Sequences--

A ***bag*** is a collection of elements with duplicates allowed. A ***sequence*** is an ordered collection of elements with duplicates allowed. In Figure an itinerary is a sequence of airports and the same airport can be visited more than once.

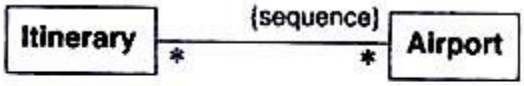


Figure **An example of a sequence**. An itinerary may visit multiple airports, so you should use *{sequence}* and not *{ordered}*.

Association Classes—

An ***association class*** is an association that is also a class. In Figure ***accessPermission*** is an attribute of ***AccessibleBy.*** The sample data at the bottomn of the figure shows the value for each link. The UML notation for an association class is a box (a class box) attached to the association by a dashed line.

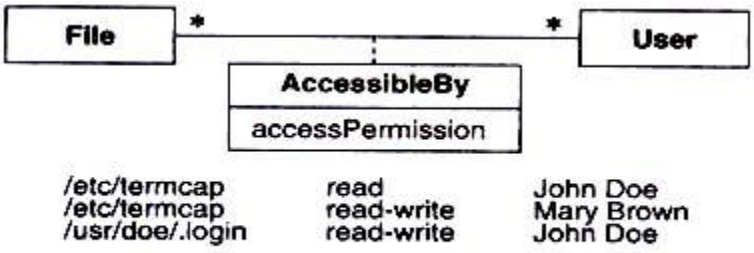


Figure **An association class.** The links of an association can have attributes.

Qualified Associations--

A ***qualified association*** is an association in which an attribute called the ***qualifier*** disambiguates the objects for a "many" association end. It is possible to define qualifiers for one-to-many and many-to-many associations. ***Bank*** and ***Account*** are classes and ***accountNumber*** is the qualifier. Qualification reduces the effective multiplicity of this association from one-to-many to one-to-one.

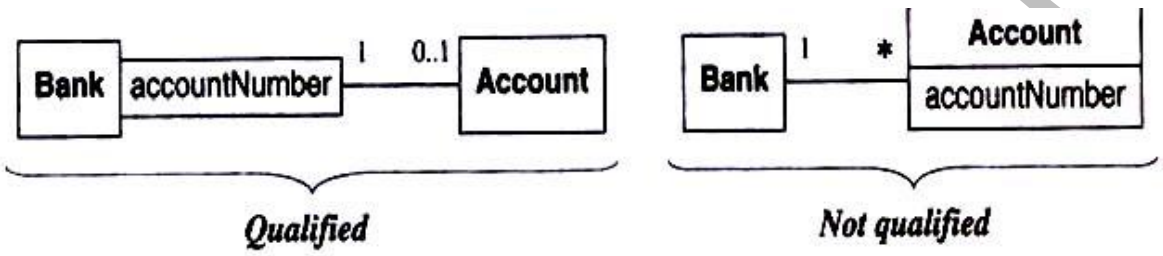


Figure **Qualified association.** Qualification increases the precision of a model.

N-ary Associations--

we may occasionally encounter ***n-ary associations*** (associations among three or more classes.)

Figure shows a genuine n-ary (ternary) association: Programmers use computer languages on projects. This n-ary association is an atomic unit and cannot he subdivided into binary associations without losing information. A programmer may know a language and work on a project, but might not use the language on the project. The UML symbol for n-ary associations is a diamond with lines connecting to related classes. If the association has a name, it is written in italics next to the diamond.

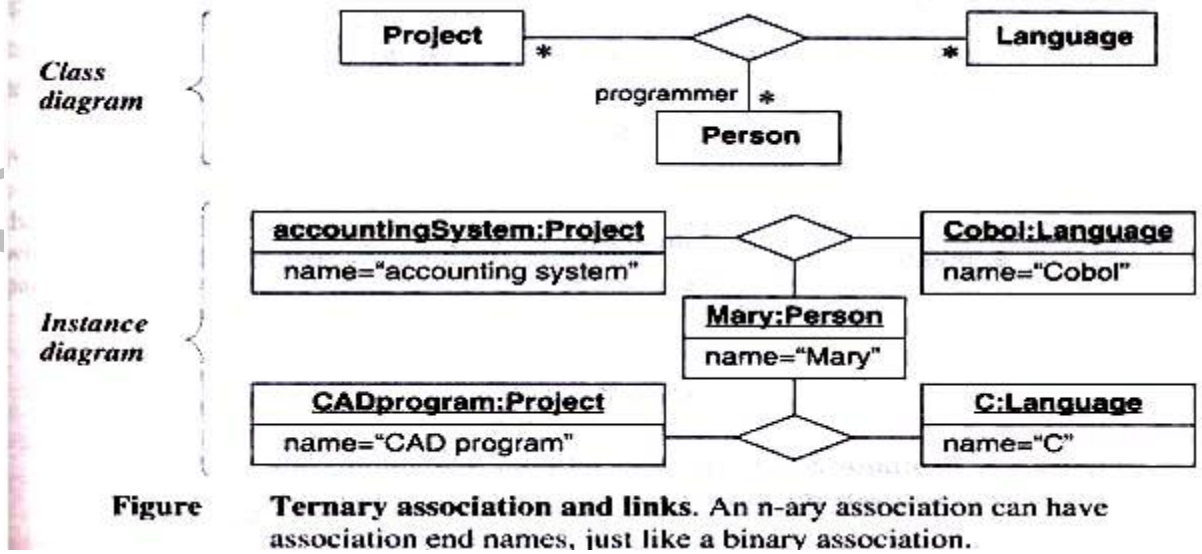


Figure **Ternary association and links.** An n-ary association can have association end names, just like a binary association.

Generalization and Inheritance--

Generalization is the relationship between a class (the ***superclass)*** and one or more variations of the class (the ***subclasses).*** Generalization organizes classes by their similarities and differences, structuring the description of objects. The superclass holds common attributes, operations, and associations; the subclasses add specific attributes, operations, and associations. Each subclass is said to ***inherit*** the features of its superclass. Generalization is sometimes called the "is-a" relationship, because each instance of a subclass is an instance of the superclass as well.

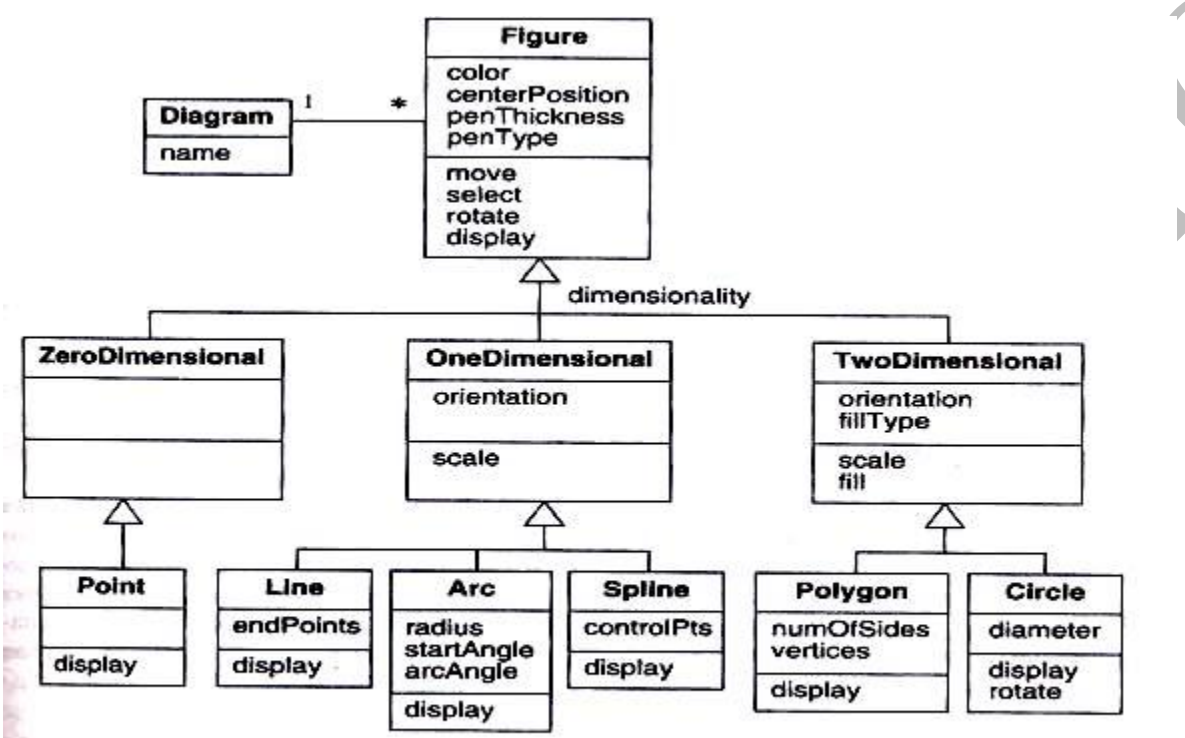
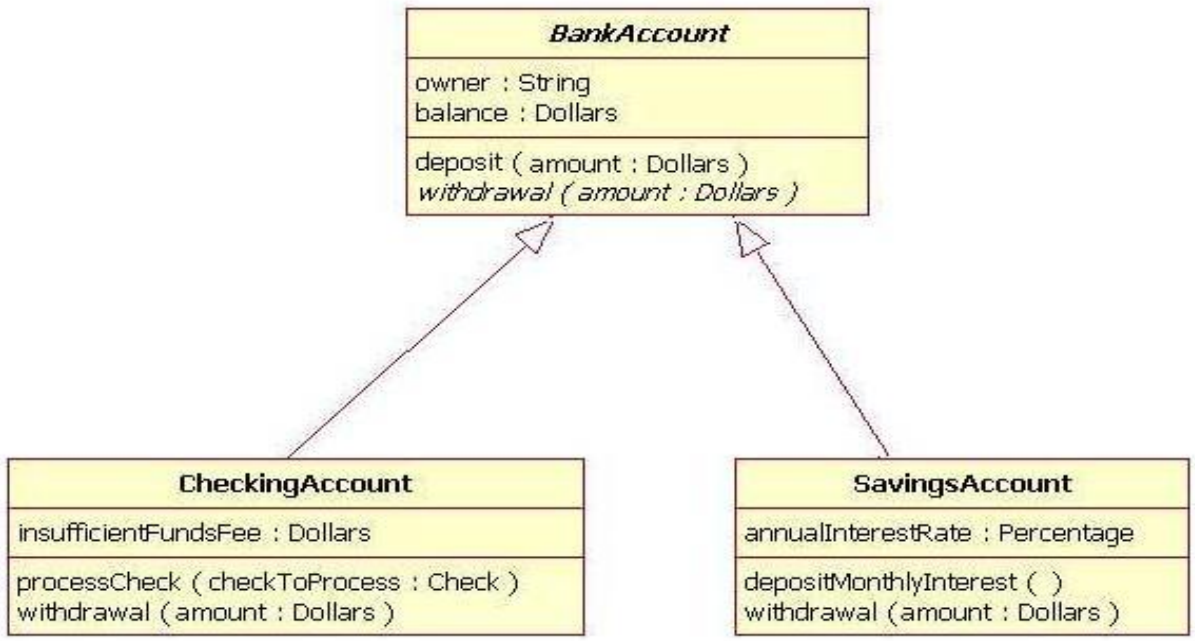


Figure **Inheritance for graphic figures.** Each subclass inherits the attributes, operations, and associations of its superclasses.



Fig—Inheritance example

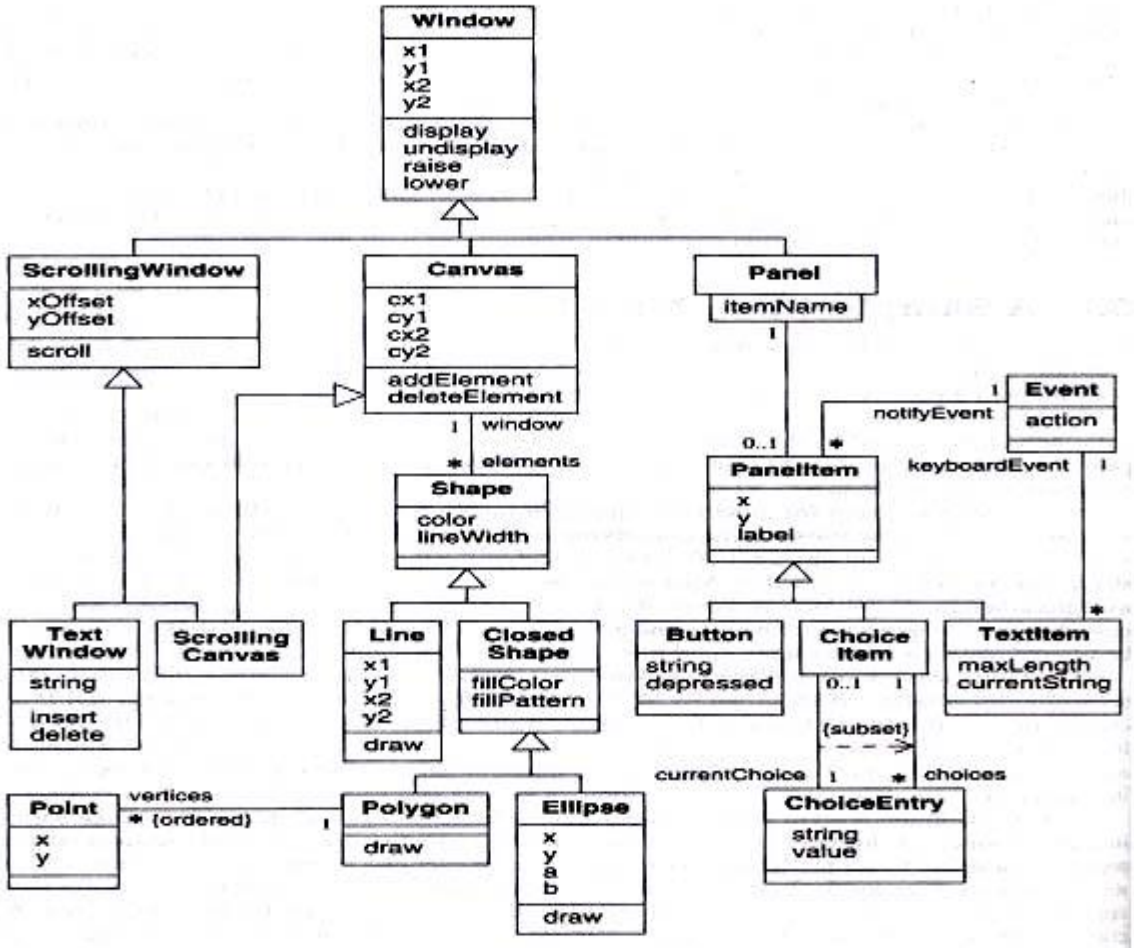


Figure **Class model of a windowing system**

Multiple Inheritance--

Multiple inheritance permits a class to have more than one superclass and to inherit features from all parents. The advantage of multiple inheritance is greater power in specifying classes and an increased opportunity for reuse. The disadvantage is a loss of conceptual and implementation simplicity.

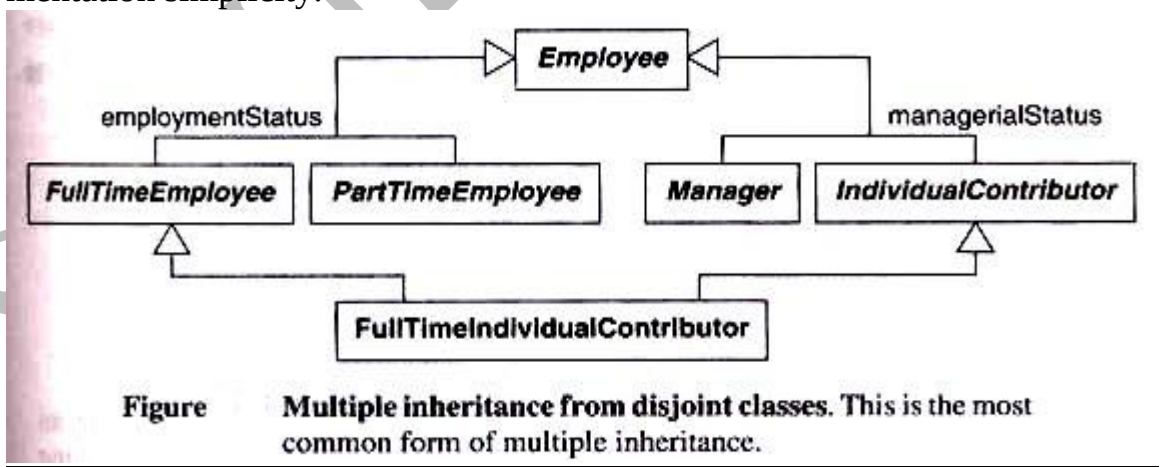


Figure **Multiple inheritance from disjoint classes.** This is the most common form of multiple inheritance.

Multiple inheritance can also occur with overlapping classes. In Figure ***Amphibi-onsVehicle*** is both ***LandVehicle*** and ***WaterVehicle. LandVehicle*** and ***WaterVehicle*** overlap, because some vehicles travel on both land and water. The UML uses a constraint to indicate an

overlapping generalization set; the notation is a dotted line cutting across the affected generalizations with keywords in braces.

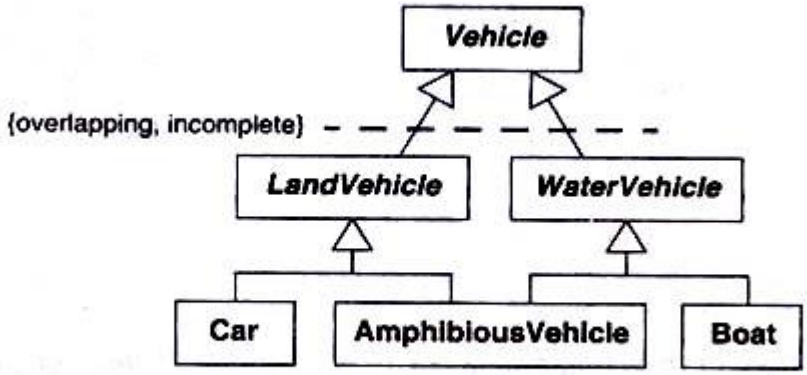


Figure **Multiple inheritance from overlapping classes**. This form of multiple inheritance occurs less often than with disjoint classes.

Realization

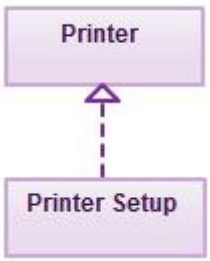


Fig--Realization

denotes the implementation of the functionality defined in one class by another class. To show the relationship in UML, a broken line with an unfilled solid arrowhead is drawn from the class that defines the functionality of the class that implements the function. In the example, the printing preferences that are set using the printer setup interface are being implemented by the printer.

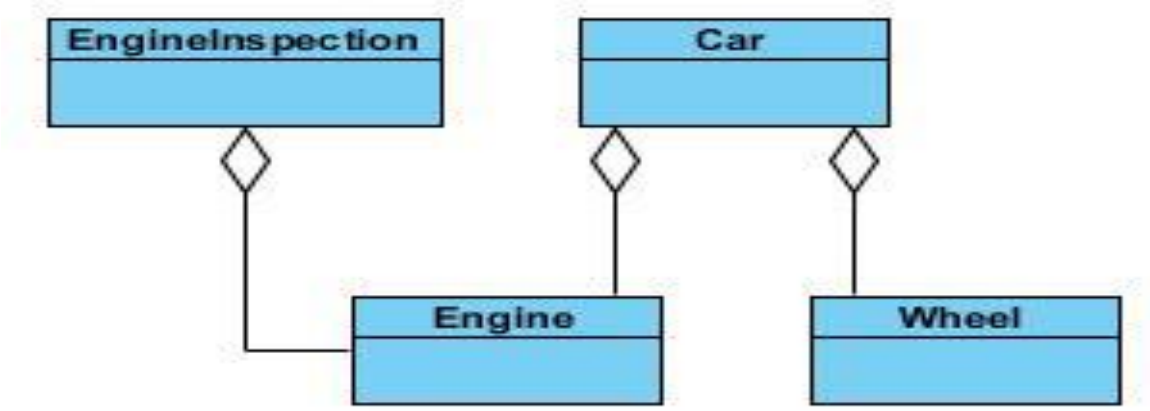
Aggregation--

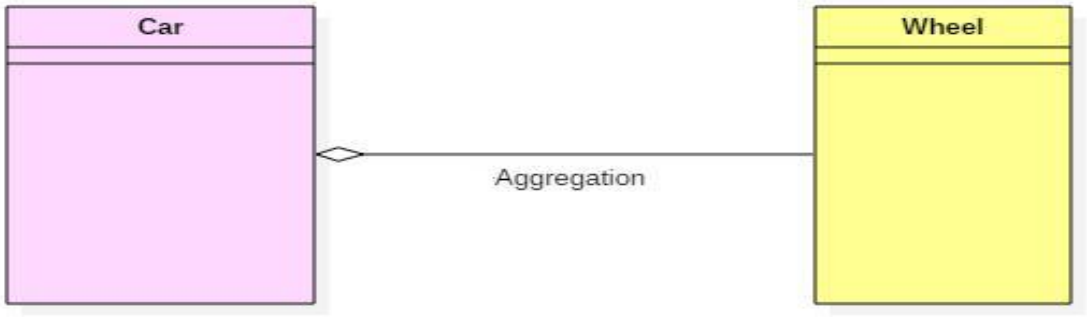
Aggregation as relation of an assembly class to ***one*** constituent part class. An assembly with many kinds of constituent parts corresponds to many aggregationsThis definition emphasizes that aggregation is a special form of binary association.

The most significant property of aggregation is ***transitivity***— that is, if A is part of ***B*** and ***B*** is part of ***C,*** then ***A*** is part of ***C.*** Aggregation is also ***antisymmetric***—that is, if A is part of ***B,*** then is not part of ***A.***

Examples

Examples--





Aggregation

Composition--

Composition is a form of aggregation with two additional constraints. A constituent part can belong to at most one assembly. It has a coincident lifetime with the assembly. Thus composition implies ownership of the parts by the whole.

In Figure a company consists of divisions, which in turn consist of departments: a company is indirectly a composition of departments. A company is not a composition of its employees, since company and person are independent objects of equal stature.

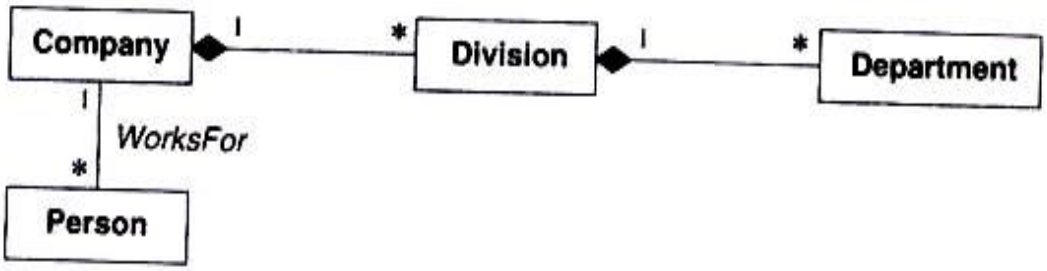
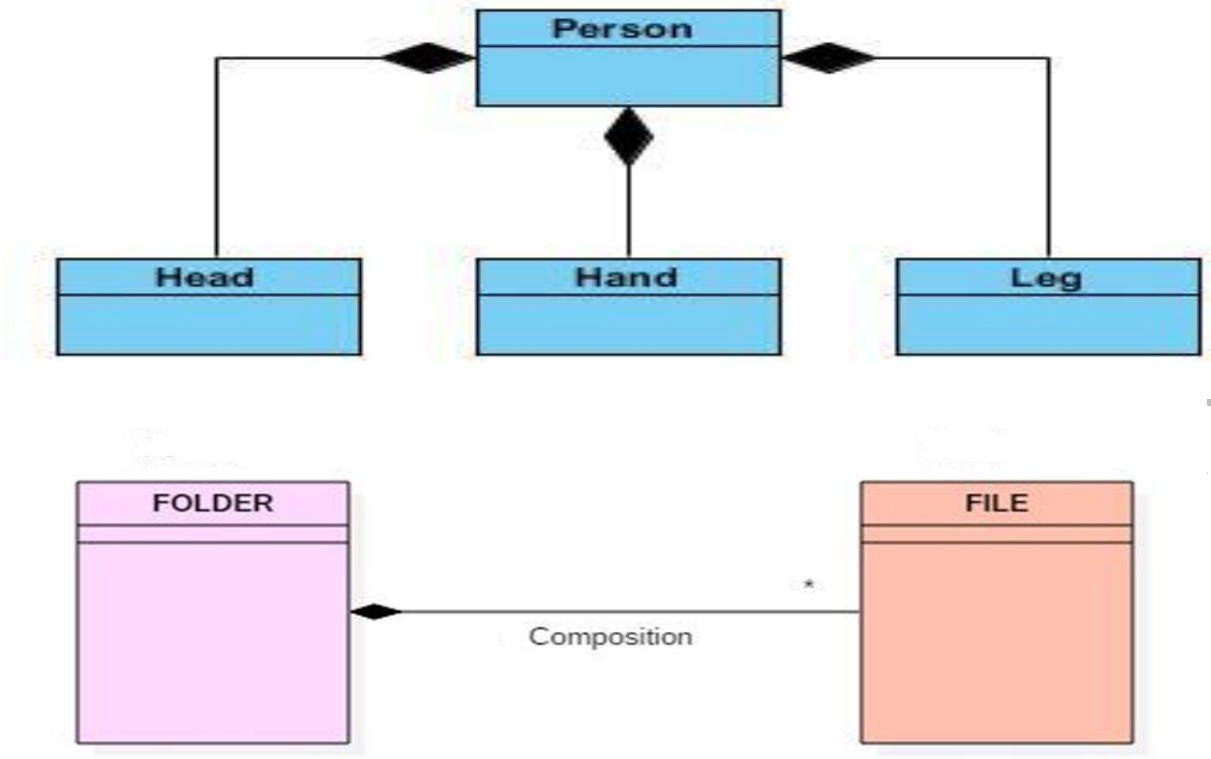


Figure **Composition.** With composition a constituent part belongs to at most one assembly and has a coincident lifetime with the assembly.

Examples--



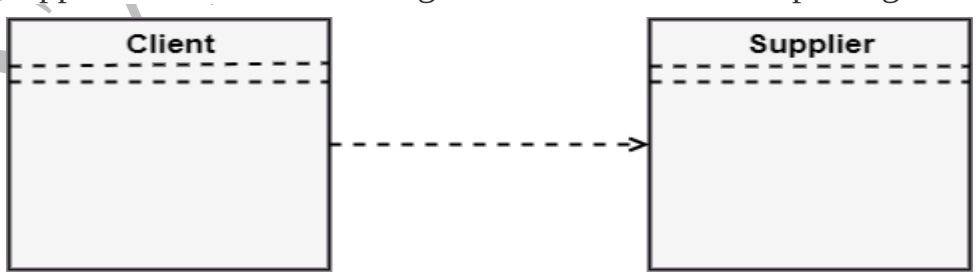
Composition

Packages--A ***package*** is a group of elements (classes, associations, generalizations, and lesser packages) with a common theme. A package partitions a model, making it easier to understand and manage.

Figure **Notation for a package.** Packages let you organize large models so that persons can more readily understand them.

Dependency

Dependency depicts how various things within a system are dependent on each other. In UML, a dependency relationship is the kind of relationship in which a client (one element) is dependent on the supplier (another element). It is used in class diagrams, component diagrams, deployment diagrams, and use-case diagrams, which indicates that a change to the supplier necessitates a change to the client. An example is given below:



Types of Dependency Relationship

Following are the type of dependency relationships, keywords, or stereotypes given below:

- **<<derive>>** -It is a constraint that specifies the template can be initialized by the source at the target location utilizing given parameters.
- **<<derive>>** -It represents that the source object's location can be evaluated from the target object.
- **<<friend>>** -It states the uniqueness of the source in the target object.
- **<<instanceOf>>** -It states that an instance of a target classifier is the source object.
- **<<instantiate>>** -It defines the capability of the source object, creating instances of a target object.
- **<<refine>>** -It states that the source object comprises of exceptional abstraction than that of the target object.
- **<<use>>** -When the packages are created in UML, the use of stereotype is used as it describes that the elements of the source package can also exist in the target package. It specifies that the source package uses some of the elements of the target package.
- **<<substitute>>** -The substitute stereotype state that the client can be substituted at the runtime for the supplier.
- **<<access>>** -It is also called as private merging in which the source package accesses the element of the target package.
- **<<import>>** -It specifies that target imports the source package's element as they are defined within the target. It is also known as public merging.
- **<<permit>>** -It describes that the source element can access the supplier element or whatever visibility is provided by the supplier.
- **<<extend>>** -It states that the behavior of the source element can be extended by the target.
- **<<include>>** -It describes the source element, which can include the behavior of another element at a specific location, just like a function call in C/C++.
- **<<become>>** -It states that target is similar to the source with distinct roles and values.
- **<<call>>** -It specifies that the target object can be invoked by the source.
- **<<copy>>** -It states that the target is an independent replica of a source object.
- **<<parameter>>** -It describes that the supplier is a parameter of the client's actions.

- **<<send>>** -The client act as an operation, which sends some unspecified targets to the supplier.

Examples---