UNIT-I

Introduction to UML

1. Importance of modeling, principles of modelling
2. object oriented modelling
3. conceptual model of the UML
4. Architecture
5. Software Development Life Cycle.

Structural Modeling

6. Classes
7. Relationships
8. common Mechanisms
9. Diagrams.
10. Advanced classes
11. advanced relationships
12. Object diagrams
13. common modeling techniques.

1Q.DEFINE UML?

Introduction of UML

- The Unified Modeling Language is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system.
- The UML is appropriate for modeling systems ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded systems. It is a very expressive language, addressing all the views needed to develop and then deploy such systems .The UML is a language for
  - Visualizing
  - Specifying
  - Constructing
  - Documenting

Visualizing
The UML is more than just a bunch of graphical symbols. Rather, behind each symbol in the UML notation is a well-defined semantics. In this manner, one developer can write a model in the UML, and another developer, or even another tool, can interpret that model unambiguously.

Specifying
The Specifying means building models that are precise, unambiguous, and complete. In particular, the UML addresses the specification of all the important analysis, design, and implementation decisions that must be made in developing and deploying a software-intensive system.

Constructing

The UML is not a visual programming language, but its models can be directly connected to a variety of programming languages. This means that it is possible to map from a model in the UML to a programming language such as Java, C++, or Visual Basic, or even to tables in a relational database or the persistent store of an object-oriented database.

This mapping permits forward engineering: The generation of code from a UML model into a programming language.

The reverse is also possible: You can reconstruct a model from an implementation back into the UML

Documenting

Documenting a healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include

- Requirements
- Architecture
- Design
- Source code
- Project plans
- Tests
- Prototypes
- Releases

2Q. WHAT IS MODEL? EXPLAIN THE IMPORTANCE AND PRINCIPLES OF MODELING?

Importance of Modeling

Model

- ✓ A model is a simplification of reality. A model provides the blueprints of a system. A model may be structural, emphasizing the organization of the system, or it may be behavioral, emphasizing the dynamics of the system.
- ✓ If we want to build a dog house, with a little planning, we'll likely end up with a dog house that's reasonably functional and we can do it with no one's help.
- ✓ If we want to build a house for a family, it's going to take a lot longer.
- ✓ In this case we need some detailed planning; we'll need to draw some blueprints, before we lay the foundation.
- ✓ If we want to build a high-rise office building, we'll have to do extensive planning; and we need all sorts of blueprints and models to communicate with one another.

Why do we model

We build models so that we can better understand the system we are developing. Through modeling, we achieve four aims.

1. Models help us to visualize a system as it is or as we want it to be.
2. Models permit us to specify the structure or behavior of a system.
3. Models give us a template that guides us in constructing a system.
4. Models document the decisions we have made.

We build models of complex systems because we cannot comprehend such a system in its entirety.

Principles of Modeling

There are four basic principles of model

1. The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.
2. Every model may be expressed at different levels of precision.
3. The best models are connected to reality.

4. No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

3Q. EXPLAIN THE OBJECT ORIENTED MODELING?

Object oriented modeling

In software, there are several ways to approach a model. The two most common ways are

1 Algorithmic perspective

2 Object-oriented perspective

1 Algorithmic Perspective

- ✓ The traditional view of software development takes an algorithmic perspective.
- ✓ In this approach, the main building block of all software is the procedure or function.
- ✓ This view leads developers to focus on issues of control and the decomposition of larger algorithms into smaller ones.
- ✓ As requirements change and the system grows, systems built with an algorithmic focus turn out to be very hard to maintain.

2 Object-oriented perspective

- ✓ The contemporary view of software development takes an object-oriented perspective.
- ✓ In this approach, the main building block of all software systems is the object or class.

- ✓ A class is a description of a set of common objects.
- ✓ Every object has identity, state, and behavior.
- ✓ Object-oriented development provides the conceptual foundation for assembling systems out of components using technology such as Java Beans or COM+.

4Q. EXPLAIN THE CONCEPTUAL MODEL OF UML (OR) EXPLAIN THE BUILDING BLOCK OF UML (OR) EXPLAIN THE THINGS IN UML (OR) EXPLAIN THE RLATIONSHIPS WITH AN EXAMPLE (OR) EXPLAIN THE DIAGRAMS (OR) EXPLAIN THE COMMON MECHANISMS?

Conceptual model of the UML

1. Things
2. Relationships
3. Diagrams

Things in the UML

There are four kinds of things in the UML:

1. Structural things
2. Behavioral things
3. Grouping things

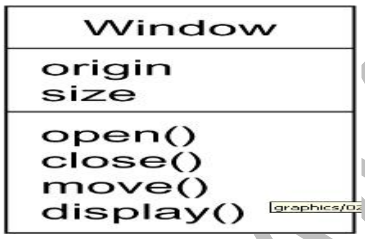
1. Structural things

Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things.

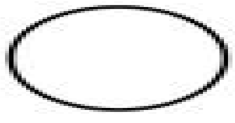
1. Classes
2. Interfaces
3. Collaborations

4. Use cases
5. Active classes
6. Components
7. Nodes

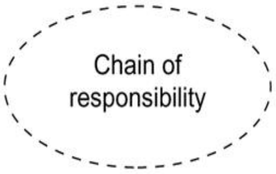
1. **<u>Class</u>**
   - ✓ Class is a description of a set of objects that share the <u>same attributes, operations, relationships, and semantics.</u>
   - ✓ A class implements one or more interfaces.
   - ✓ Graphically, a class is rendered as a <u>rectangle,</u> usually <u>including its name, attributes, and operations.</u>



2. **<u>Interface</u>**
   - ✓ Interface is a <u>collection of operations</u> that specify a service of a class or component.
   - ✓ An interface therefore describes the externally visible behavior of that element.
   - ✓ An interface might represent the <u>complete behavior of a class or component or only a part of that behavior.</u>
   - ✓ An interface is rendered as a <u>circle together with its name.</u> An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface

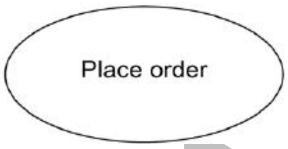


3. **<u>Collaboration</u>**
   - ✓ Collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements.
   - ✓ Graphically, collaboration is rendered as an <u>ellipse with dashed lines,</u> usually including only its name



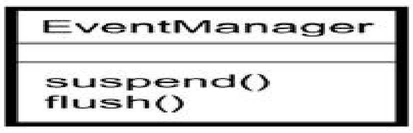
4. <u>Usecase</u>

- ✓ Use case is a description of <u>set of sequence of actions that a system performs that yields an observable result of value to a particular actor</u>
- ✓ Use case is used to structure the behavioral things in a model.
- ✓ A use case is realized by a collaboration. Graphically, a use case is rendered as an <u>ellipse with solid lines</u>, usually including only its name



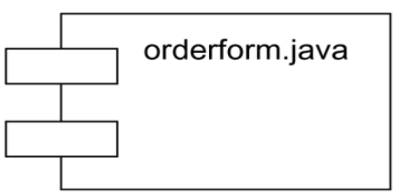
5. <u>Active class</u>

- ✓ Active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements.
- ✓ Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations



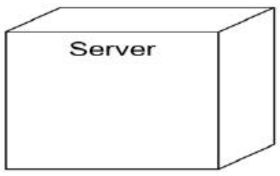
6. **<u>Component</u>**

- ✓ Component is a <u>physical and replaceable part of a system</u> that conforms to and provides the realization of a set of interfaces.
- ✓ Graphically, a component is rendered as a <u>rectangle with tabs</u>



7. **Node**

- ✓ Node is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability.
- ✓ Graphically, a node is rendered as a cube, usually including only its name



2. **Behavioral Things**

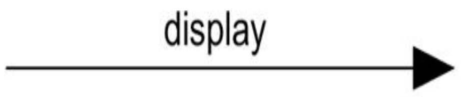
Behavioral Things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space.

In all, there are two primary kinds of behavioral things

1. Interaction
2. State machine

1. **Interaction**

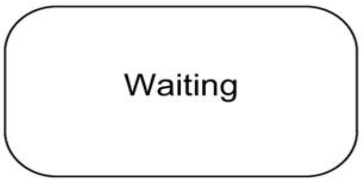
- ✓ Interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose
- ✓ An interaction involves a number of other elements, including messages, action sequences and links
- ✓ Graphically a message is rendered as a directed line, almost always including the name of its operation



2. **State Machine**

- ✓ State machine is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events

- ✓ State machine involves a number of other elements, including states, transitions, events and activities
- ✓ Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates

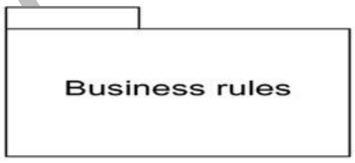


3. Grouping Things:-

- ✓ Grouping Things are the organizational parts of UML models. These are the boxes into which a model can be decomposed
- ✓ There is one primary kind of grouping thing, namely, packages.

1. Package:-

- ✓ A package is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package
- ✓ Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents

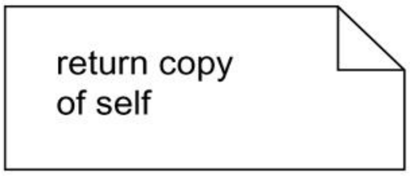


Annotational things

- ✓ Annotational things are the explanatory parts of UML models.
- ✓ These are the comments you may apply to describe about any element in a model.

Note

- ✓ A note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.
- ✓ Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment



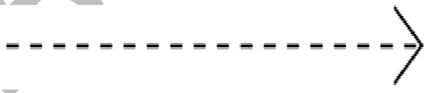
Relationships in the UML:

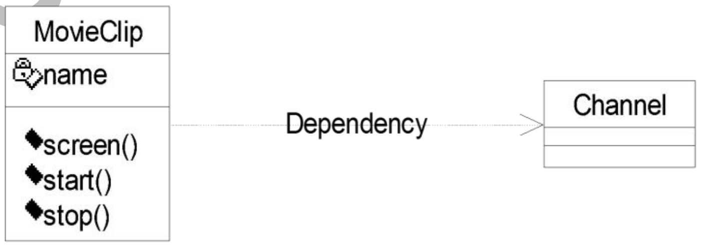
There are four kinds of relationships in the UML:

1. Dependency
2. Association
3. Generalization
4. Realization

1. Dependency:-

- ✓ Dependency is a semantic relationship between two things in which a change to one thing may affect the semantics of the other thing
- ✓ Graphically a dependency is rendered as a dashed line, possibly directed, and occasionally including a label

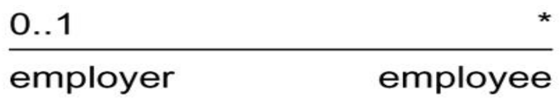


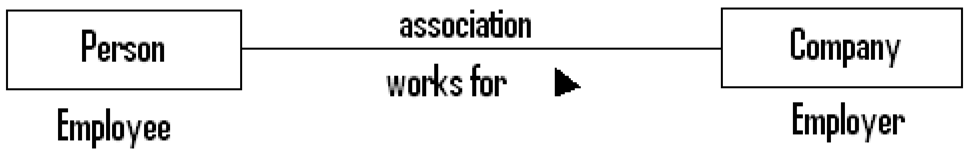


2. Association :

- ✓ Association is a structural relationship that describes a set of, a link being a connection among objects.

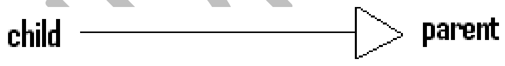
- ✓ Graphically an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names.





3. **Generalization:**

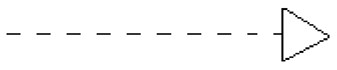
 It is denoted by a solid line with a hollow arrow head pointing to the parent

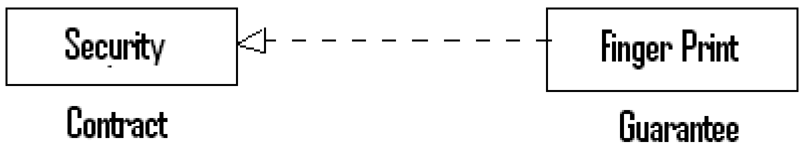


Generalization is a relationship in which the child will share the behavior of the parent.

4. **Realization**

- ✓ Realization is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out.
- ✓ It is denoted by dashed lines with a hollow arrow head.





Diagrams in the UML

- ✓ Diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).
- ✓ In theory, a diagram may contain any combination of things and relationships.
- ✓ For this reason, the UML includes nine such diagrams:

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. State chart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

There are 9 types of Diagrams in UML, which are classified into 2 types

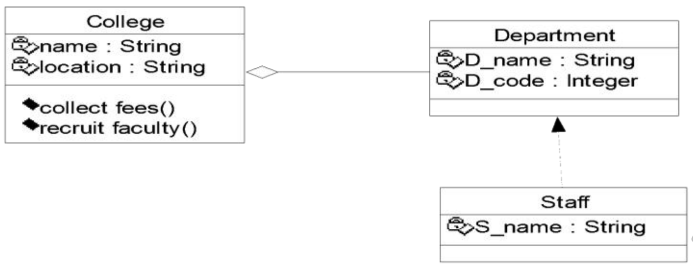
1. **Structural Diagrams** (static diagrams)
2. **Behavioral Diagrams** (Dynamic diagrams)

1. Structural Diagrams (static diagrams)

These are of 4 types

1. Class Diagram

- ✓ A Class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- ✓ A class consists of class name, attributes, operations and responsibilities.

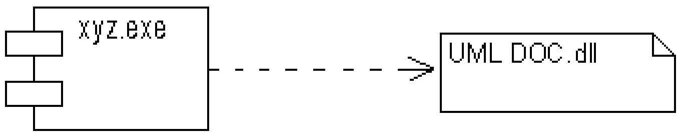


2. Object diagram

- ✓ Object diagrams represent static snapshots of instances of the things found in class diagrams.
- ✓ These diagrams address the static design view or static process view of a system.
- ✓ An object diagram shows a set of objects and their relationships.

3. Component diagram

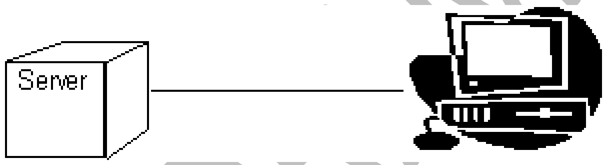
- ✓ A component diagram shows the organizations and dependencies among a set of components.
- ✓ Component diagrams address the static implementation view of a system.
- ✓ They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations
- ✓ It shows the organizations and dependencies among set of compo nets. The static(view) implementation view of a system. It displays the high level packaged structure of the code itself.



4. **Deployment diagram**

✓ It shows the configuration of runtime processing nodes and the components that live on them it address the static deployment view of architecture.

✓ It displays the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of code units.



2. **Behavioral Diagrams (Dynamic diagrams)**

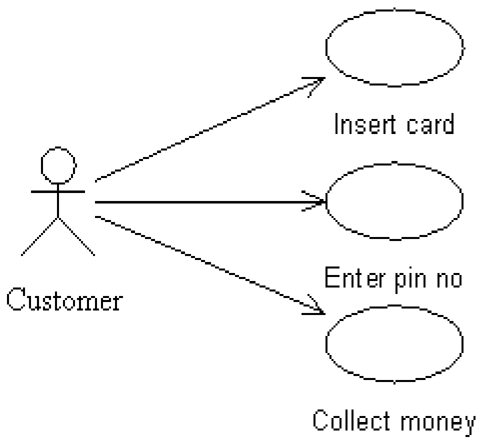
These are of 5 types

1. **Use cas e diagram**

✓ A use case diagram shows a set of use cases and actors and their relationships

✓ Use case diagrams address the static use case view of a system.

✓ These diagrams are especially important in organizing and modeling the behaviors of a system.

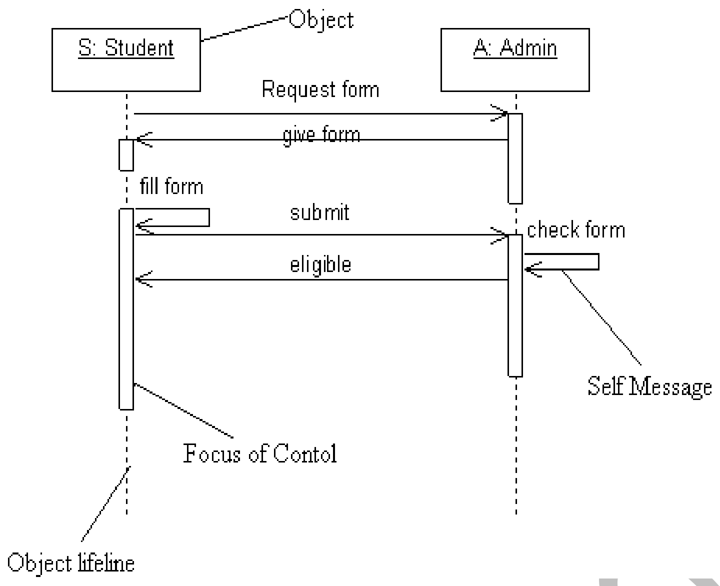


Interaction Diagrams

- ✓ Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams
- ✓ Interaction diagrams address the dynamic view of a system.

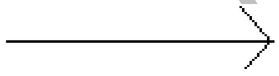
2. **A sequence diagram**

- ✓ Sequence diagram emphasizes the time ordering of messages. It mainly shows set of objects and the messages send /receive by those objects which is concerned with time ordering of messages.
- ✓ It displays the time sequence of the objects participating in the interaction. This consists of the vertical dimension (time) and horizontal dimension (different objects).



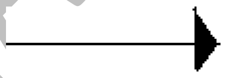
To show interaction between objects we use 3 types of messages.

Simple Messages:



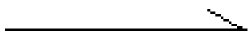
A Simple message shows how control is passed from one object to other without describing communication in detail i.e. without indicating whether it is synchronous or asynchronous message.

Synchronous Messages:



If sender object waits for a reply from receiver object from destination, such messages are called Synchronous messages. Here, only one object can send a message at a given instance of time.

Asynchronous Messages:



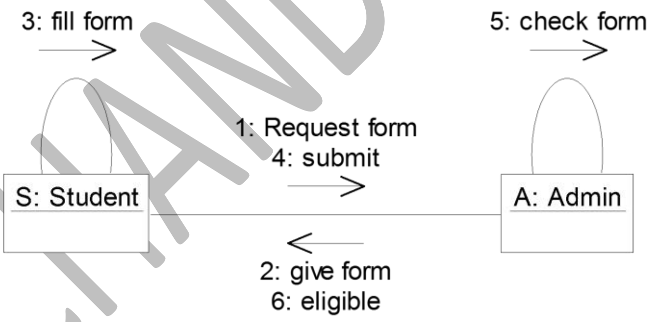
If sender object continues executing while target is processing the message then such messages are said to be Asynchronous messages. Here, multiple messages are executed at a time.

Object Lifeline: An Object life line is vertical dashed lines that represent the existence of an object over a period of time.

Focus of Control: It is represented by rectangle that shows the period of time during which an object performs some actions.

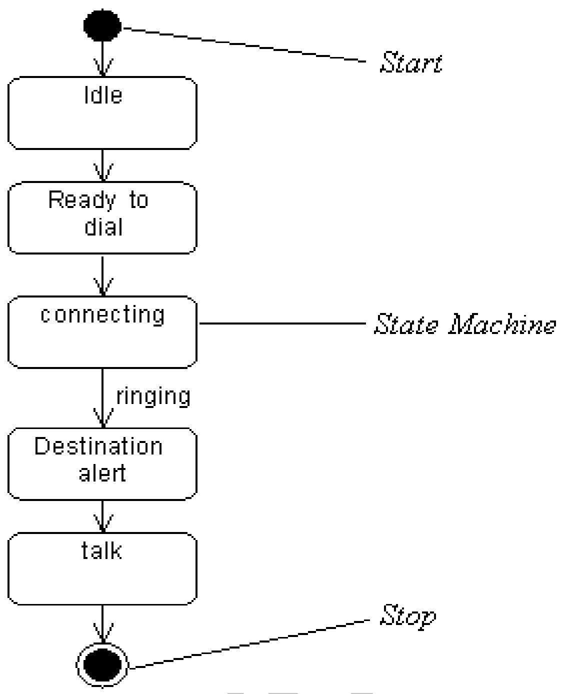
3. A collaboration diagram

- ✓ A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages
- ✓ Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.



4. Statechart diagram

- ✓ A statechart diagram shows a state machine, consisting of states, transitions, events, and activities.
- ✓ Statechart diagrams address the dynamic view of a system.
- ✓ They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object



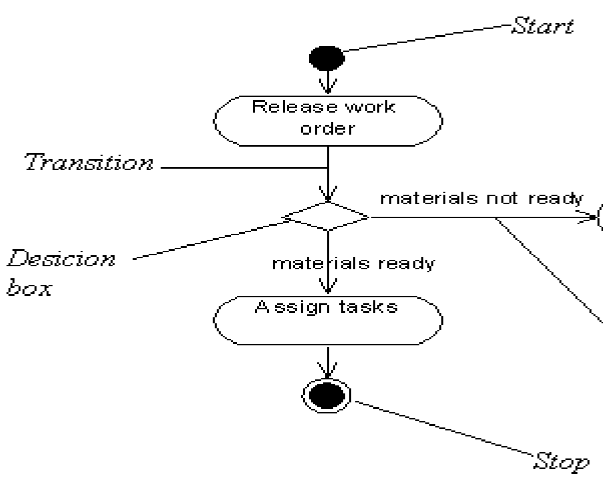
5. **Activity diagram**

- ✓ An activity diagram is a special kind of a statechart diagram that shows the flow from activity to activity within a system
- ✓ Activity diagrams address the dynamic view of a system
- ✓ They are especially important in modeling the function of a system and emphasize the flow of control among objects.

Activity: It is a major task that must take place in order to fulfill an operation contract.

Initial Activity: This shows the starting point of the flow. It is denoted by solid circle

Final Activity: This shows the end of the flow in the activity diagram. It is denoted by a solid circle nested in a circle.



<u>Decision Box</u>: A point in an Activity diagram where a <u>flow splits into several mutually exclusive guarded flows</u>. It has one incoming transition and two outgoing transitions.

<u>Forking and Joining</u>: We use synchronization bar to <u>specify the forking and joining of parallel flows of control.</u>

<u>A ***synchronization bar***</u> is a thick horizontal or vertical line.
A ***Fork*** may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control.

A ***<u>Join</u>*** may have two or more incoming transitions and one outgoing transition.

<u>Rules of the UML</u>

1. Names	What you can call things, relationships, and diagrams
2. Scope	The context that gives specific meaning to a name
3. Visibility	How those names can be seen and used by others
4. Integrity	How things properly and consistently relate to one another
5. Execution	What it means to run or simulate a dynamic model

The UML has semantic rules for

Models built during the development of a software-intensive system tend to evolve and may be viewed by many stakeholders in different ways and at different times. For this reason, it is common for the development team to not only build models that are well-formed, but also to build models that are

1. Elided Certain elements are hidden to simplify the view
2. Incomplete Certain elements may be missing
3. Inconsistent The integrity of the model is not guaranteed

Common Mechanisms in the UML

UML is made simpler by the presence of four common mechanisms that apply consistently throughout the language.

1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

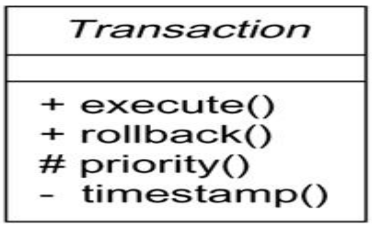
1. **Specification**

- ✓ Specification that provides a textual statement of the syntax and semantics of that building block.
- ✓ The UML's specifications provide a semantic backplane that contains all the parts of all the models of a system, each part related to one another in a consistent fashion.

2. **Adornments**

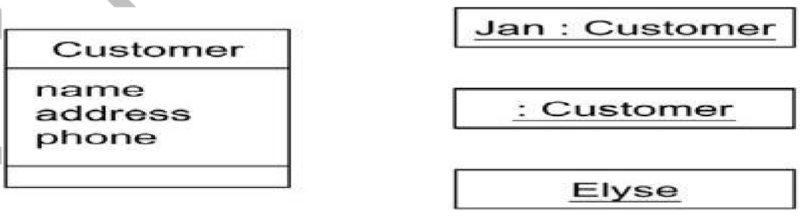
Adornments most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element.

A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations. Many of these details can be rendered as graphical or textual adornments to the class's basic rectangular notation.



3. Common Divisions

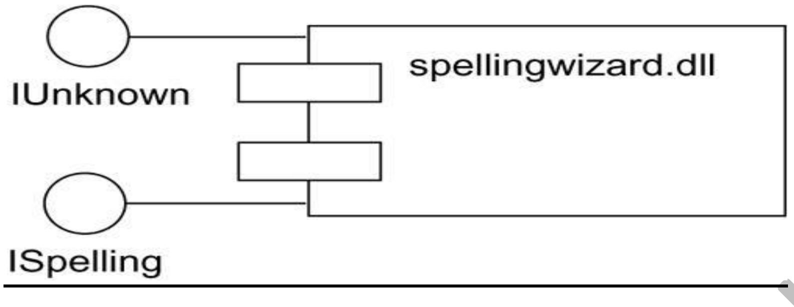
- ✓ In modelling object-oriented systems, the world often gets divided in at least a couple of ways.
- ✓ First, there is the division of class and object. A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, you can model classes as well as objects, as shown in Fig.
- ✓ In this figure, there is one class, named Customer, together with three objects: Jan (which is marked explicitly as being a Customer object), :Customer (an anonymous Customer object),and Elyse (which in its specification is marked as being a kind of Customer object, although it'snot shown explicitly here).



Class And objects

- ✓ Second, there is the separation of interface and implementation. An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics.

In the UML, you can model both interfaces and their implementations Shown in Fig



Interfaces And Implementation

In this figure, there is one component named spellingwizard.dll that implements two interfaces, IUnknown and ISpelling.

4. Extensibility Mechanisms

The UML's extensibility mechanisms include

a) Stereotypes

b) Tagged values

c) Constraints

a) **Stereotype**

Stereotype extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem.

b) Tagged value

A tagged value extends the properties of a UML building block, allowing you to create new information in that element's specification.

c) Constraint

A constraint extends the semantics of a UML building block, allowing you to add new rules or modify existing ones.

5Q. EXPLAIN THE ARCHITECTURE OF UML?

ARCHITECTURE

1 Use case view

2 Design View

3 Process View

4 Implementation View

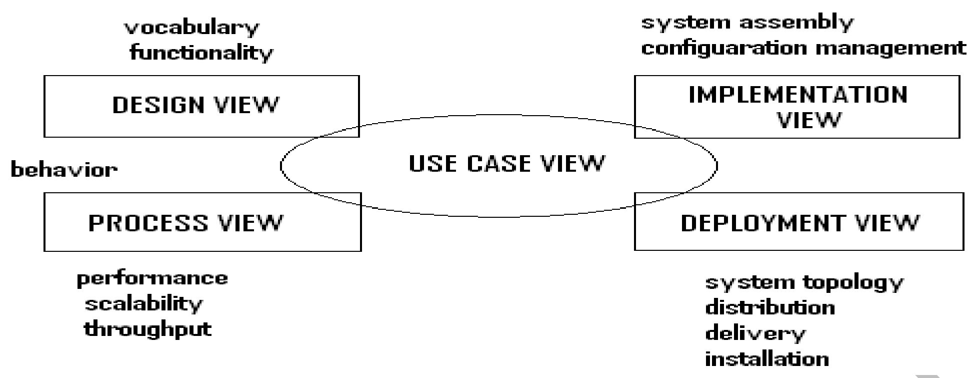
5 Deployment Diagram

A system's architecture is perhaps the most important artifact that can be used to manage these different viewpoints and so control the iterative and incremental development of a system throughout its life cycle.

Architecture is the set of significant decisions about

- ✓ The organization of a software system
- ✓ The selection of the structural elements and their interfaces by which the system is composed
- ✓ Their behavior, as specified in the collaborations among those elements
- ✓ The composition of these structural and behavioral elements into progressively larger subsystems

The architectural style that guides this organization: the static and dynamic elements and their interfaces, their collaborations, and their composition.



Modeling a System's Architecture

1. **Use case view**

✓ The use case view of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers.

With the UML, the static aspects of this view are captured in use case diagrams

✓ The dynamic aspects of this view are captured in interaction diagrams, state chart diagrams, and activity diagrams.

2. **Design View**

✓ The design view of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution.

✓ This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users.

3. **Process View**

✓ The process view of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms.

✓ This view primarily addresses the performance, scalability, and throughput of the system

4. **Implementation View**

- ✓ The implementation view of a system encompasses the components and files that are used to assemble and release the physical system.
- ✓ This view primarily addresses the configuration management of the system's releases, made up of somewhat independent components and files that can be assembled in various ways to produce a running system.

5. **Deployment view**

- ✓ The *deployment view* of a system encompasses the nodes that form the system's hardware topology on which the system executes.
- ✓ This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system. With the UML, the static aspects of this view are captured in deployment diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

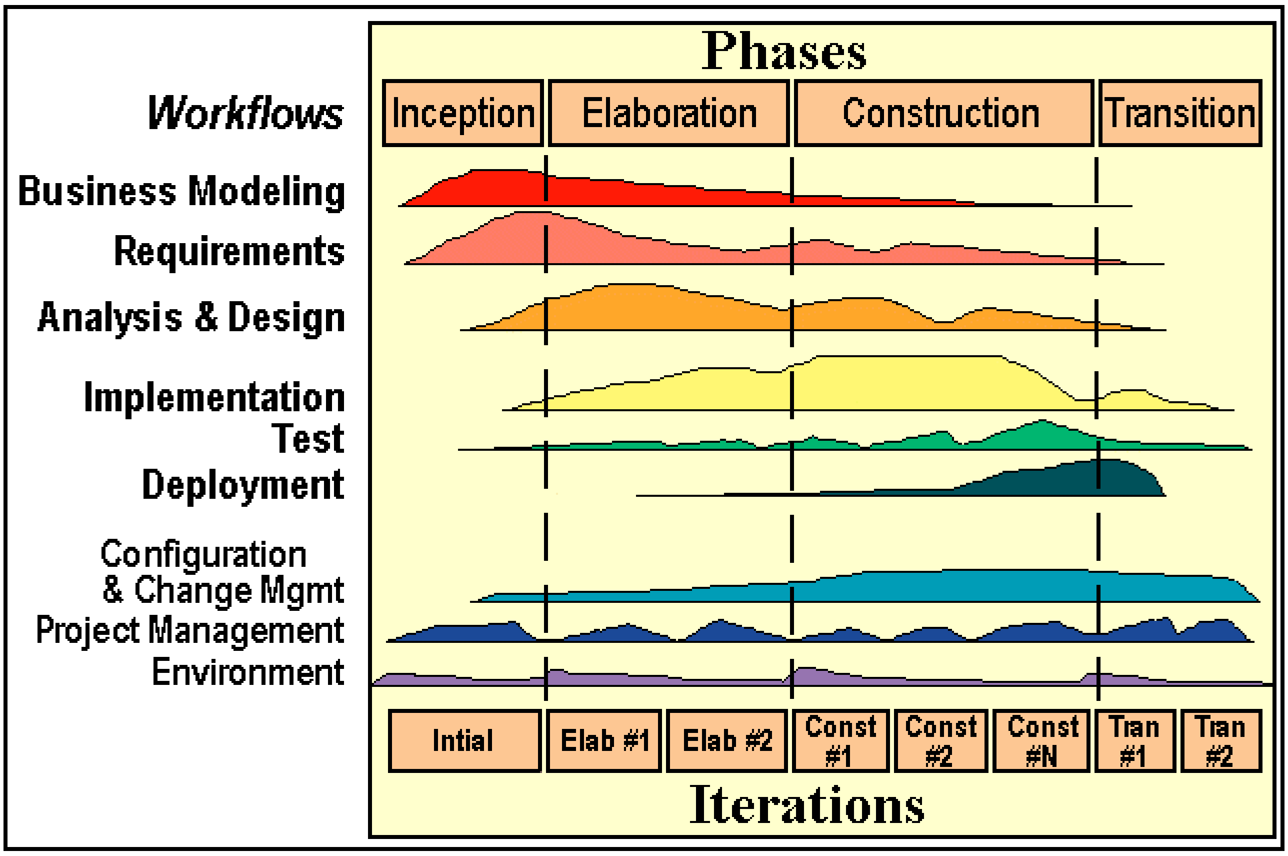
6Q. EXPLAIN THE SOFTWARE DEVELOPMENT LIFE CYCLE ?

Software Development Life Cycle

UML is a software development life cycle or process independent language. But to get most out of UML, the software development process should have the following properties:

- Use case driven
- Architecture centric
- Iterative and Incremental

Rational Unified Process (RUP) is a software development process framework developed by Rational Corporation which satisfies the above three properties. The overall software development life cycle can be visualized as shown below:



Critical activities in each phase:

Inception:

- Business case is established
- 20% of the critical use cases are identified

Elaboration:

- Develop the architecture
- Analyze the problem domain (80% of use cases are identified)

Construction:

- Source code
- User manual
- Verification and validation of code

Transition:

- Deployment of software
- New releases
- Training

7Q. EXPLAIN THE CLASS DIAGRAMS IN UML?

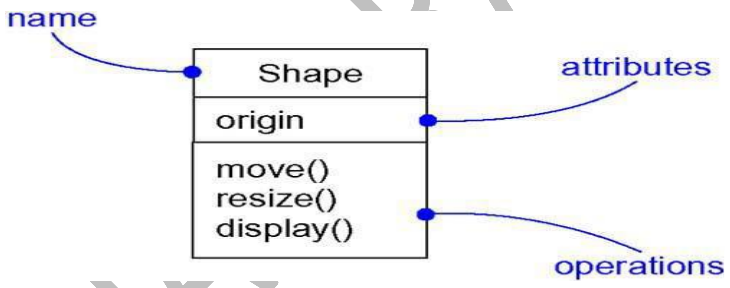
1. Class

A class is a description of a set of objects that share the same **attributes, operations, relationships, and semantics.**

A class implements one or more interfaces.

The UML provides a graphical representation of class

1. Class name
2. Attributes
3. Operations
4. Responsibilities

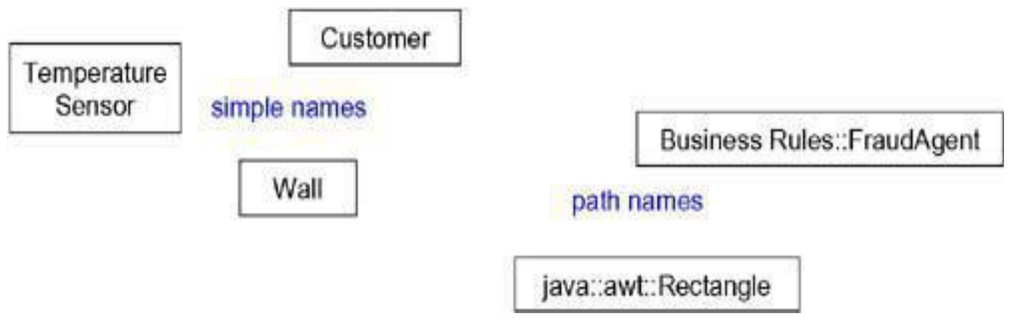


class

Terms and Concepts

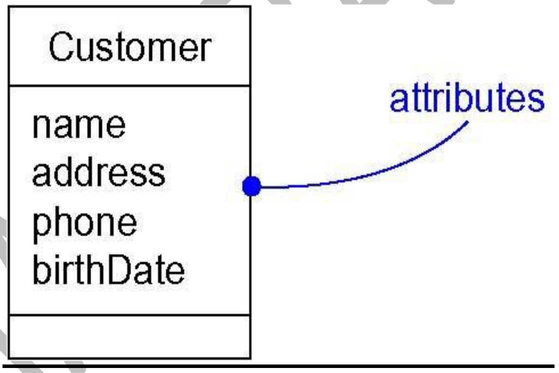
1. Names

- ✓ Every class must have a name that distinguishes it from other classes.
- ✓ **A name is a textual string** that name alone is known as a simple name; a path name is the class name prefixed by the name of the package in which that class lives.



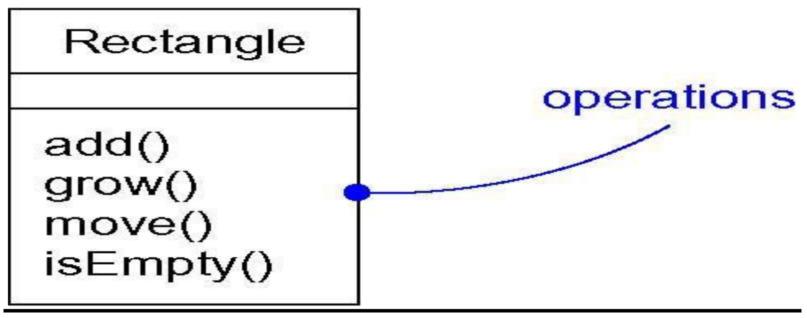
Simple and path names

2. **Attributes**
   - ✓ An attribute is a **named property of a class** that describes a range of values that instances of the property may hold.
   - ✓ A class may have any **number of attributes or no attributes** at all.
   - ✓ An attribute represents some property of thing you are modeling that is shared by all objects of that class
   - ✓ You can further specify an attribute by stating its class and possibly a default initial value



Attributes

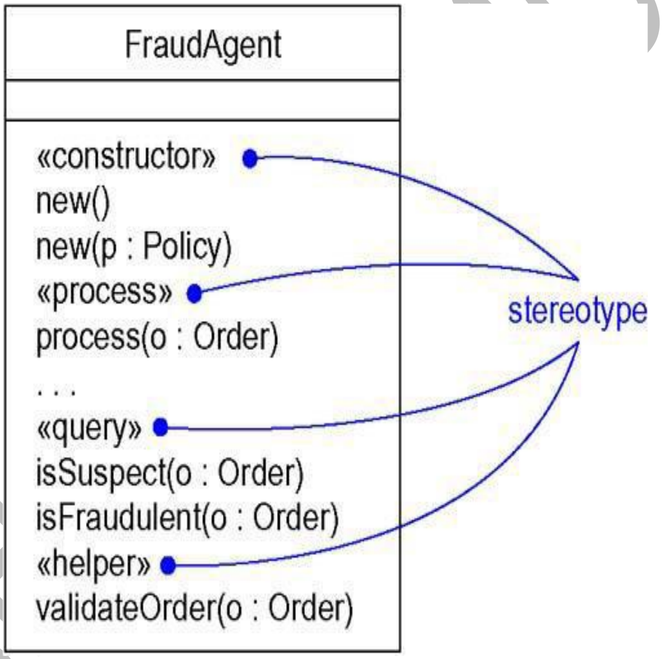
3. **Operations**
   - ✓ An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior.
   - ✓ A class may have any number of operations or no operations at all graphically; operations are listed in a **compartment just below the class attributes**.
   - ✓ You can specify an operation by stating its signature, covering the name, type, and default value of all parameters and a return type.



Operations

Organizing Attributes and Operations

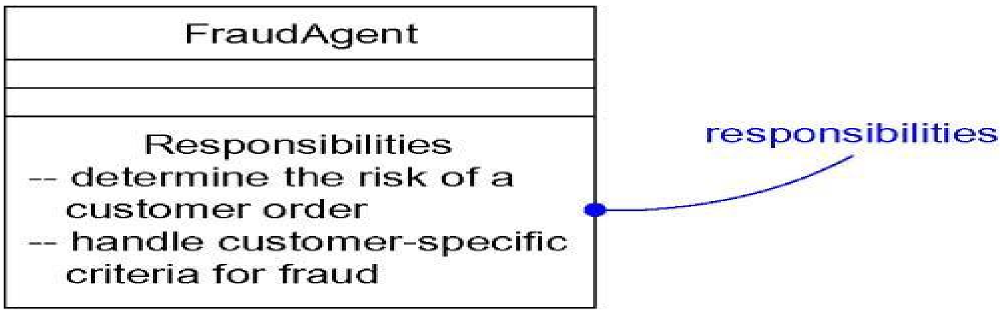
- ✓ To better organize long lists of attributes and operations, you can also prefix each group with a descriptive category by using stereotypes



Stereotypes for Class Features

4. Responsibilities

- ✓ A Responsibility is a contract or an obligation of a class.
- ✓ When you model classes, a good starting point is to specify the responsibilities of the things in your vocabulary.
- ✓ A class may have **any number of responsibilities**, although, in practice, every well-structured class has at least one responsibility and at most just a handful.
- ✓ Graphically, responsibilities can be drawn in a separate compartment at the **bottom of the class icon**


FraudAgent
Responsibilities
-- determine the risk of a customer order
-- handle customer-specific criteria for fraud
responsibilities

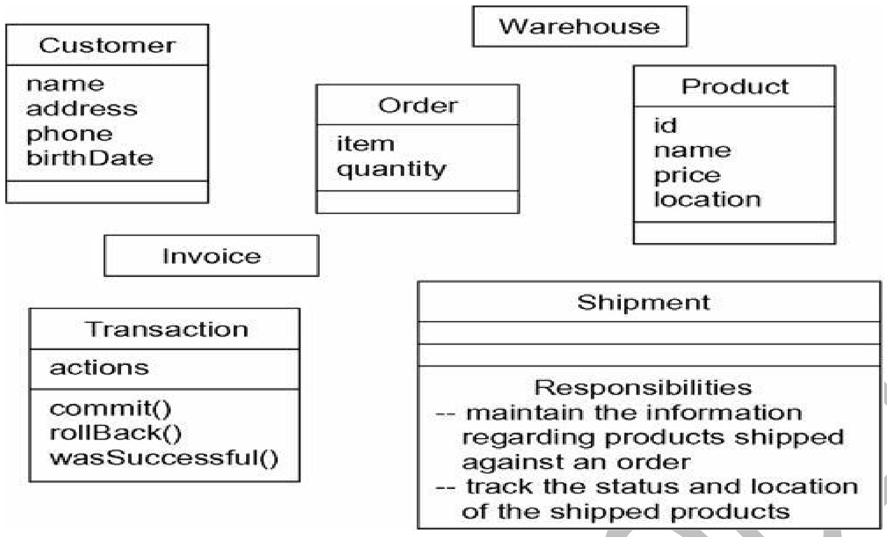
Responsibilities

8Q. EXPLAIN THE COMMON MODELING TECHNIQUES OF CLASS DIAGRAMS IN UML?

1 Modeling the Vocabulary of a System
2 Modeling the Distribution of Responsibilities in a System
3 Modeling Non software Things
4Modeling Primitive Types

1 Modeling the Vocabulary of a System

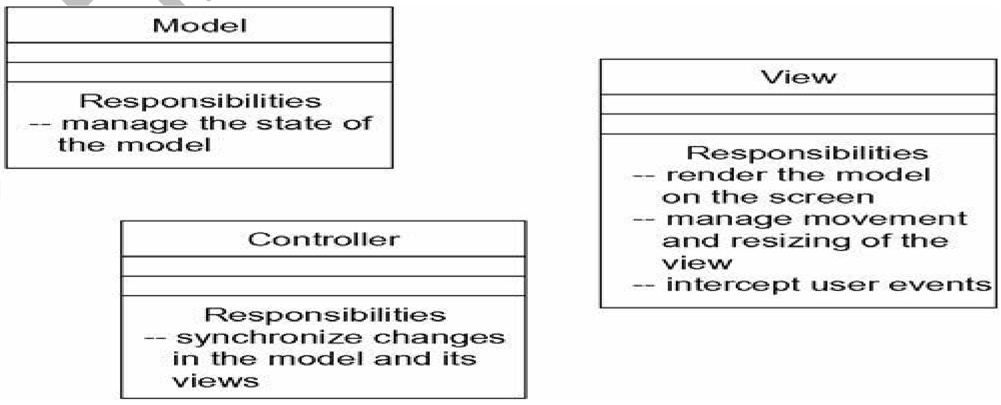
1. To model the vocabulary of a system
   - ✓ **Identify those things** that users or implementers use to describe the problem or solution. Use CRC cards and use case-based analysis to help find these abstractions.
   - ✓ For each abstraction, identify a **set of responsibilities**.
   - ✓ Provide the **attributes and operations** that are needed to carry out these responsibilities for each class.



Modeling the Vocabulary of a System

2 Modeling the Distribution of Responsibilities in a System

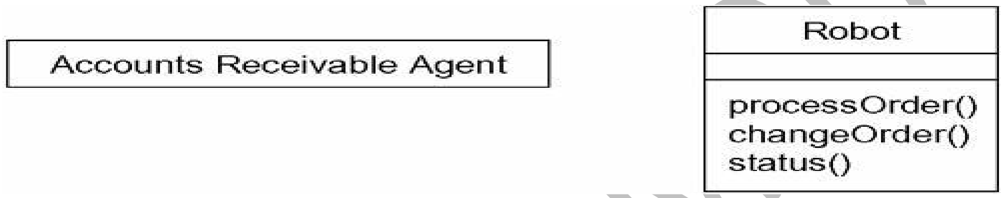
1. Once you start modeling more than just a handful of classes, you will want to be sure that your abstractions provide a balanced set of responsibilities.
2. To model the distribution of responsibilities in a system.
   - ✓ **Identify a set of classes** that work together closely to carry out some behavior.
   - ✓ Identify a **set of responsibilities** for each of these classes.
   - ✓ Consider the ways in which those classes **collaborate with one another**, and redistribute their responsibilities accordingly so that no class within collaboration does too much or too little.


Model
Responsibilities
-- manage the state of the model
View
Responsibilities
-- render the model on the screen
-- manage movement and resizing of the view
-- intercept user events
Controller
Responsibilities
-- synchronize changes in the model and its views

Modeling the Distribution of Responsibilities in a System

3 Modeling Non software Things

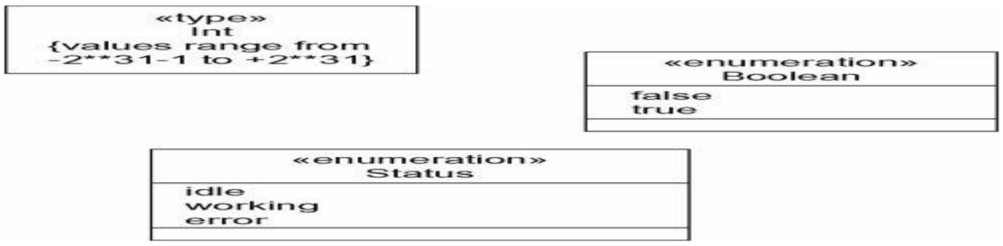
- ✓ Model the thing you are **abstracting as a class.**
- ✓ If you want to distinguish these things from the UML's defined **building blocks, create a newbuilding block** by using stereotypes to specify these new semantics and to give a distinctive visual cue.
- ✓ If the thing you are modeling is **some kind of hardware that itself contains software**, consider modeling it as a kind of node, as well, so that you can further expand on its structure.
- ✓



Modeling Non software Things

4 Modeling Primitive Types

- ✓ If you need to **specify the range of values associated** with this type, use constraints.
- ✓ As Figure shows, these things can be modelled in the UML as **types or enumerations, which are rendered just like classes but are explicitly marked via stereotypes**.
- ✓ Things like integers (**represented by the class Int)** are modeled as types, and you can explicitly indicate the range of values these things can take on by using a constraint. Similarly, enumeration types, such as Boolean and Status, can be modelled as enumerations, with their individual values provided as attributes.



Modeling Primitive Types

9Q. EXPLAIN THE RELATIONSHIPS IN UML?

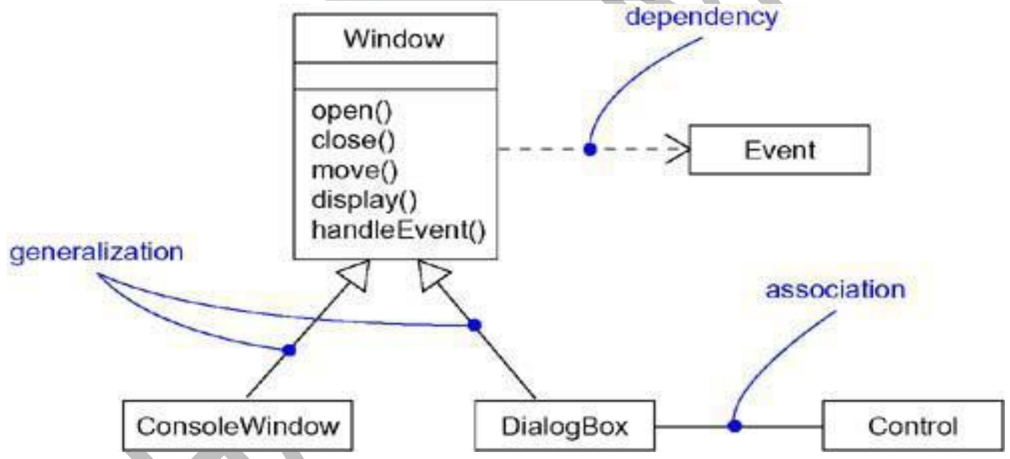
- ✓ In the UML, **the ways that things can connect to one another**, either logically or physically, are modeled as relationships.
- ✓ Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships

1 Dependency

2 Association

3 Generalization

4 Aggregation



Relationship

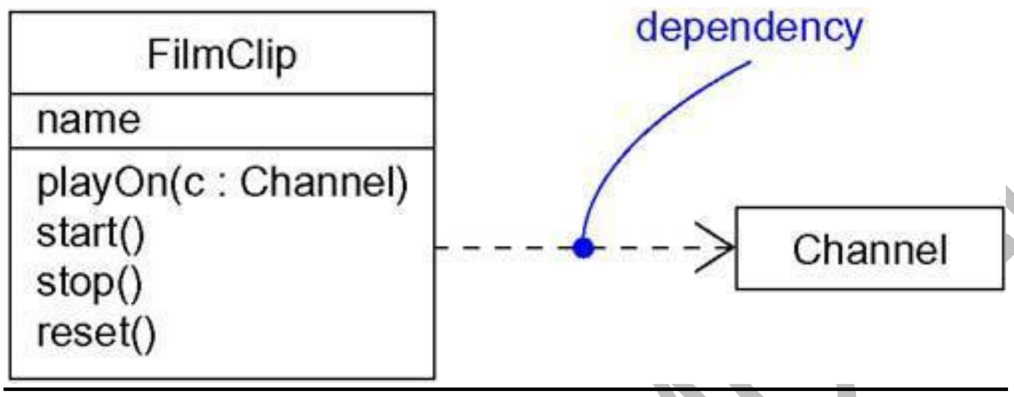
Terms and Concepts

- ✓ A relationship is a connection among things. In object-oriented modelling, the three most important relationships are dependencies, generalizations, and associations. Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships.

1 Dependency

- ✓ A dependency is a using relationship that states that a change in **specification of one thing may affect another thing** that uses it but not necessarily the reverse.

- ✓ Graphically dependency is rendered as a **dashed directed line**, directed to the thing being depended on.
- ✓ Most often, you will use dependencies in the context of classes to show that one class uses another class as an argument in the signature of an operation



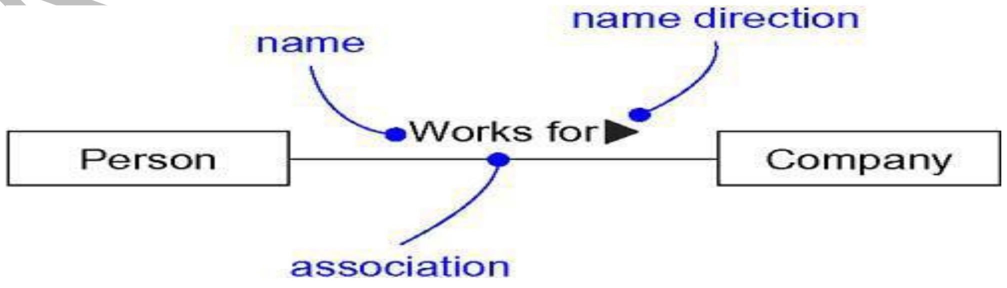
<u>Dependency</u>

2 <u>Association</u>

- ✓ An association is a structural relationship **that specifies that objects of one thing are connected to objects of another.**
- ✓ An association that connects exactly **two classes is called a binary association**.
- ✓ An associations that connect more than two classes; these are called **n-ary associations.**
- ✓ Graphically, an association is rendered as a solid line connecting the same or different classes.

<u>Name</u>

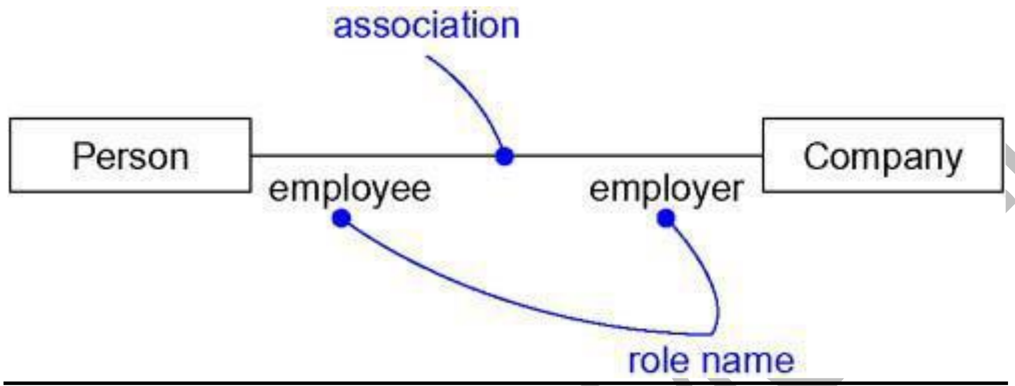
- ✓ An association can have a **name,** and you use that name to describe the nature of the relationship



<u>Association names</u>

Role

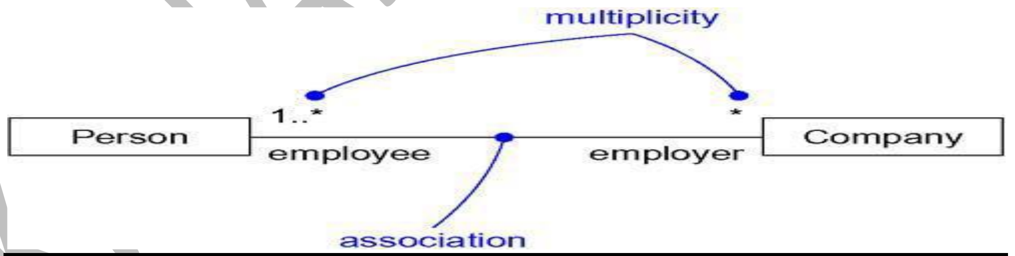
- ✓ When a class participates in an association, it has a **specific role that it plays** in that relationship;
- ✓ The same class can play the same or different roles in other associations.
- ✓ An instance of an association is called a **link**



Roles

Multiplicity

- ✓ In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association
- ✓ This "**how many**" is called the multiplicity of an association's role
- ✓ You can show a multiplicity of exactly **one (1), zero or one (0..1), many (0..*),** or **one or more (1..*)**. You can even state an exact number (for example)

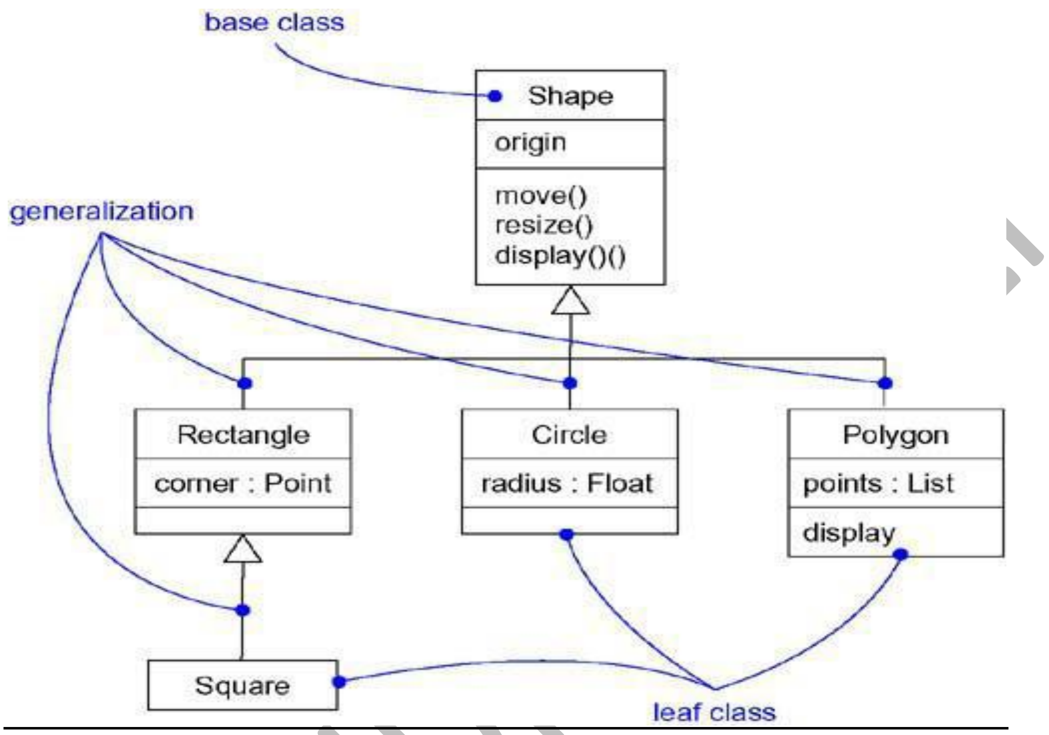


Multiplicity

3 **Generalization**

- ✓ A generalization is a relationship between a **general thing (called the super class or parent)** and a more specific kind of that thing (called the subclass or child).
- ✓ Generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations

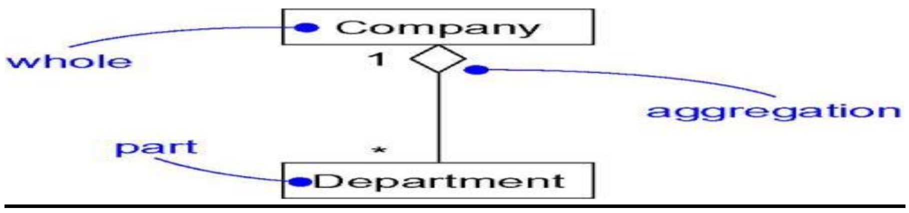
- ✓ Graphically generalization is rendered as **a solid directed line with a large** open arrowhead, pointing to the parent.



<u>Generalization</u>

4 <u>Aggregation</u>

- ✓ Sometimes, you will want to model a "**whole/part**" relationship, in **which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts").**
- ✓ This kind of relationship is called aggregation, which represents a "has-a" relationship, meaning that an object of the whole has objects of the part.



<u>Aggregation</u>

10 Q. EXPLAIN THE COMMON MODERN TECHNIQUES OF RELASTIONSHIPS IN UML?

1 Modeling Simple Dependencies

2 Modeling Single Inheritance

3 Modeling Structural Relationships

4 To model structural Relationships

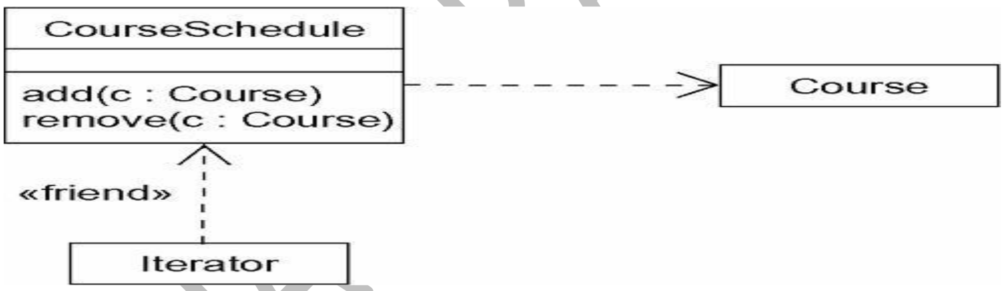
Modelling Simple Dependencies:

- ✓ The most common kind of dependency relationship is the connection between a **class that only uses another class as a parameter to an operation.**

To model this using relationship

- ✓ Create a dependency pointing from the class with the operation to the class used as a parameter in the operation.

.

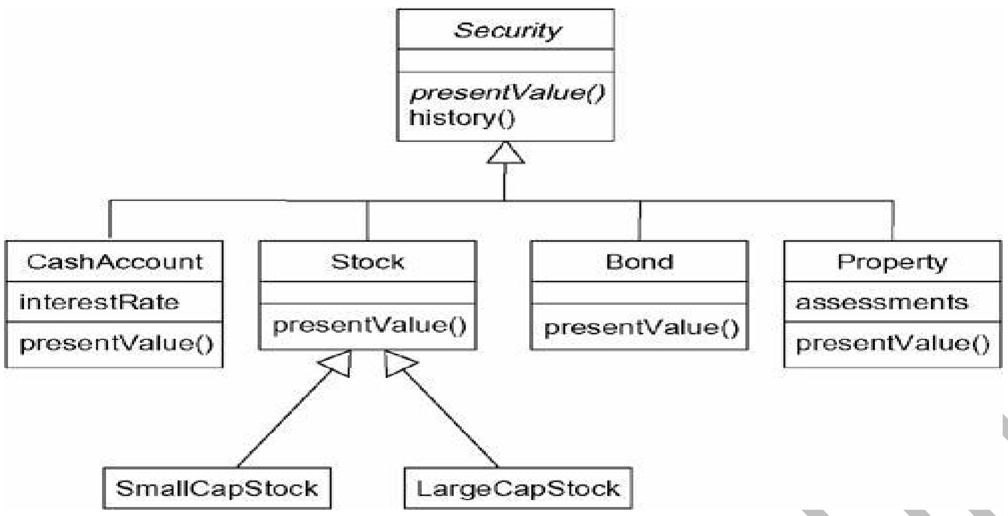


Modelling Simple Dependencies

4.2 Modeling Single Inheritance

To model inheritance relationships

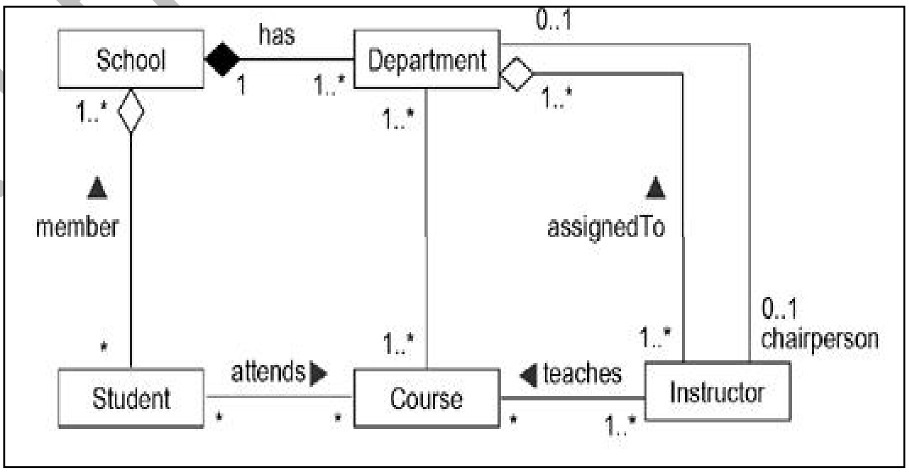
- ✓ Given a set of classes, look for **responsibilities, attributes, and operations** that are common to two or more classes.
- ✓ Elevate these **common responsibilities, attributes, and operations to a more general class**.



Modeling Single Inheritance

4.3 Modeling Structural Relationships

- ✓ An association specifies a structural path across which objects of the classes interact.
- ✓ For each pair of classes, if you need to navigate from objects of one to objects of another, specify an **association between the two**. This is a data-driven view of associations.
- ✓ For each pair of classes, if objects of one class need to **interact with objects of the other class** other than as parameters to an operation, specify an association between the two. This is more of a behavior-driven view of associations.
- ✓ For each of these associations, **specify a multiplicity** (especially when the multiplicity is not *, which is the default), as well as **role names** (especially if it helps to explain the model).



Structural relationships

11Q. EXPLAIN THE COMMON MECHANISMS IN UML?

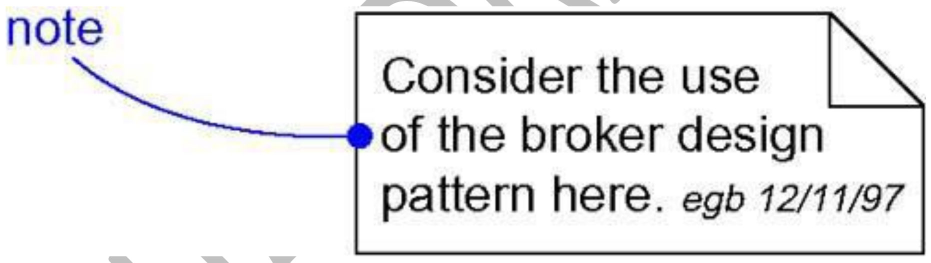
3. Common Mechanisms

1. Notes
2. Stereotypes,
3. Tagged values
4. Constraints

Terms and concepts

5.1 Notes

- ✓ A note is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements.
- ✓ Graphically, a note is rendered as a **rectangle with a dog-eared corner**, together with a **textual or graphical comment.**
- ✓ A note may contain any combination of **text or graphics.**

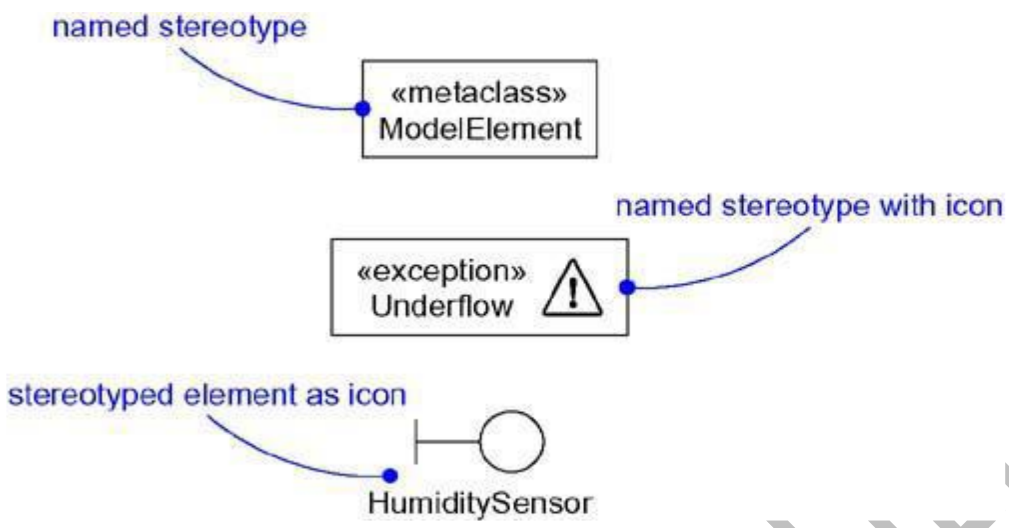

note
Consider the use of the broker design pattern here. egb 12/11/97

Notes

5.2 Stereotypes

✓ A stereotype is an extension of the vocabulary of the UML, allowing you **to create new kinds of building blocks** similar to existing ones but specific to your problem.

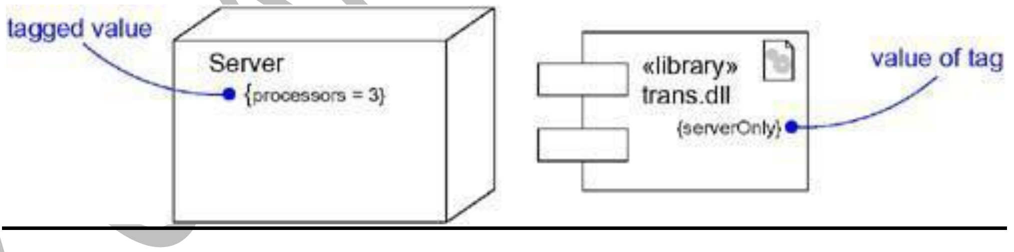
✓ Graphically, a stereotype is rendered as a name enclosed by **guillemets** << >>and placed above the name of another element



<u>**Stereotypes**</u>

5.3 <u>Tagged Values</u>

- ✓ Everything in the UML has its own set of properties: **classes have names, attributes, and operations; associations have names and two or more ends** (each with its own properties); and so on.
- ✓ With stereotypes, you **can add new things** to the UML; with tagged values, you can add new properties.

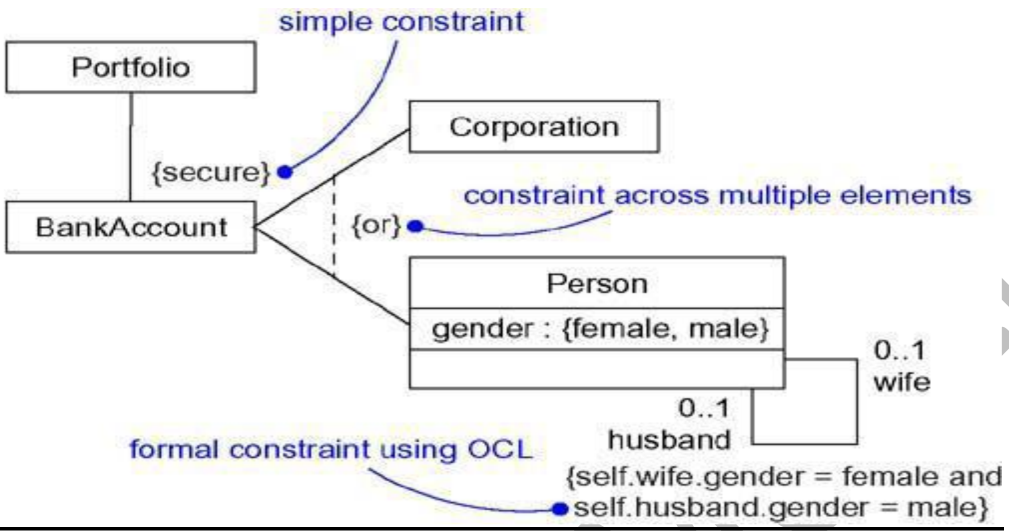


<u>**Tagged Values**</u>

5.4 <u>Constraints</u>

- ➢ A constraint **specifies conditions** that must be held true for the model to be well formed.
- ➢ A constraint is rendered as a **string enclosed by brackets** and placed near the associated element

➢ Graphically, a constraint is rendered as a string enclosed by brackets and placed near the associated element or connected to that element or elements by dependency relationships.


simple constraint
Portfolio
Corporation
{secure}
constraint across multiple elements
BankAccount
{or}
Person
gender : {female, male}
0..1
wife
0..1
husband
formal constraint using OCL
{self.wife.gender = female and
self.husband.gender = male}

Constraints

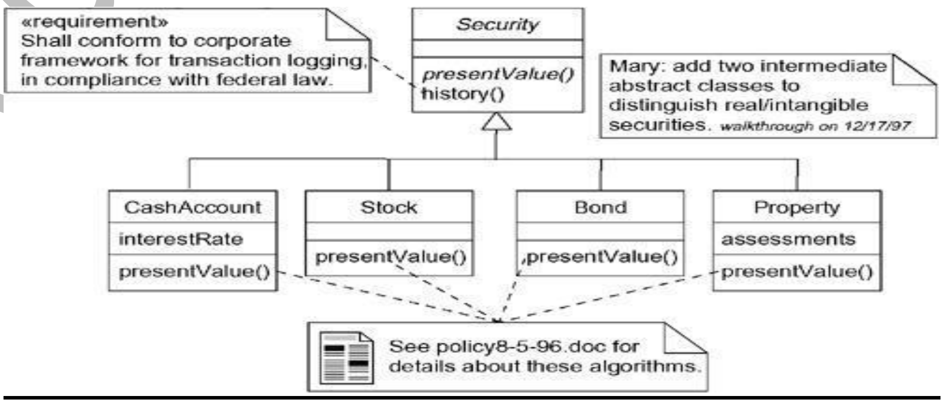
12Q. EXPLAIN THE COMMON MODELING TECHNIQUES FOR COMMON MECHANISMS IN UML?

Common Modeling Techniques for Common Mechanisms

1 Modeling Comments

✓ The most common purpose for which you'll use notes is to write down free-form **observations, reviews, or explanations.**

✓ If your comment is **lengthy or involves something richer than plain text**, consider putting your comment in an **external document and linking or embedding that document in a note attached to your model.**


«requirement»
Shall conform to corporate framework for transaction logging, in compliance with federal law.
Security
presentValue()
history()
Mary: add two intermediate abstract classes to distinguish real/intangible securities. walkthrough on 12/17/97
CashAccount
interestRate
presentValue()
Stock
presentValue()
Bond
presentValue()
Property
assessments
presentValue()
See policy8-5-96.doc for details about these algorithms.

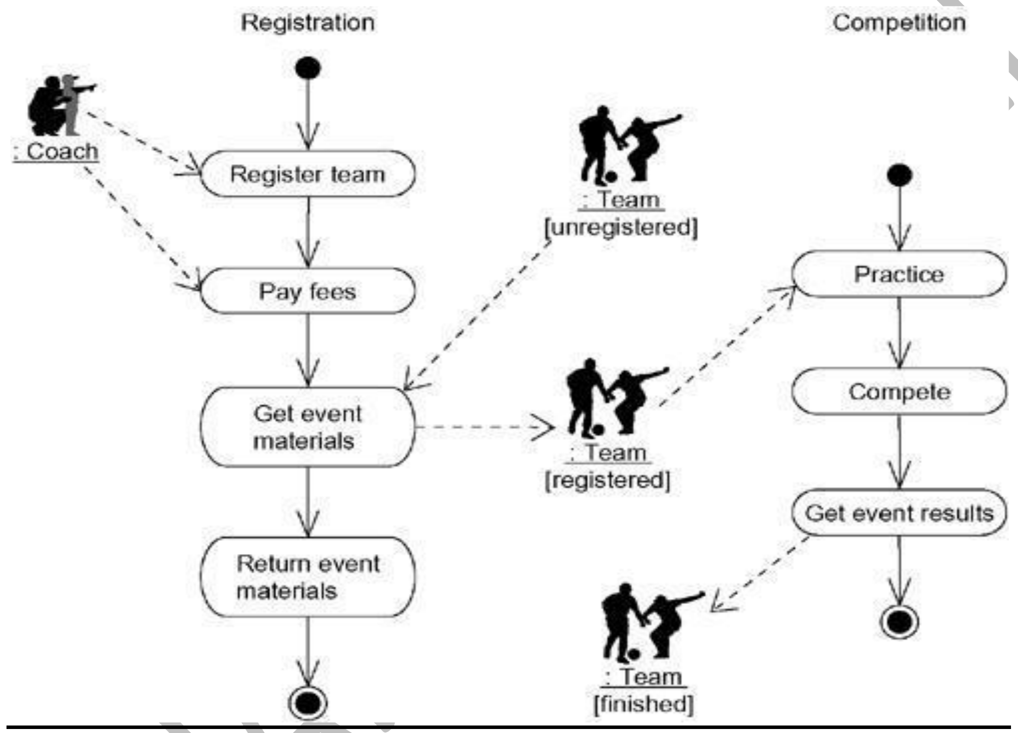
Modeling Comment

2 Modeling New Building Blocks

To model new building blocks,

identify the primitive thing in the UML that's most like what you want to model and define a **new stereotype** for that thing.

- ✓ As **Figure** shows, there are two things that standout• **Coach objects and Team objects**. These are not just plain kinds of classes. Rather, they are now primitive building blocks that you can use in this context.



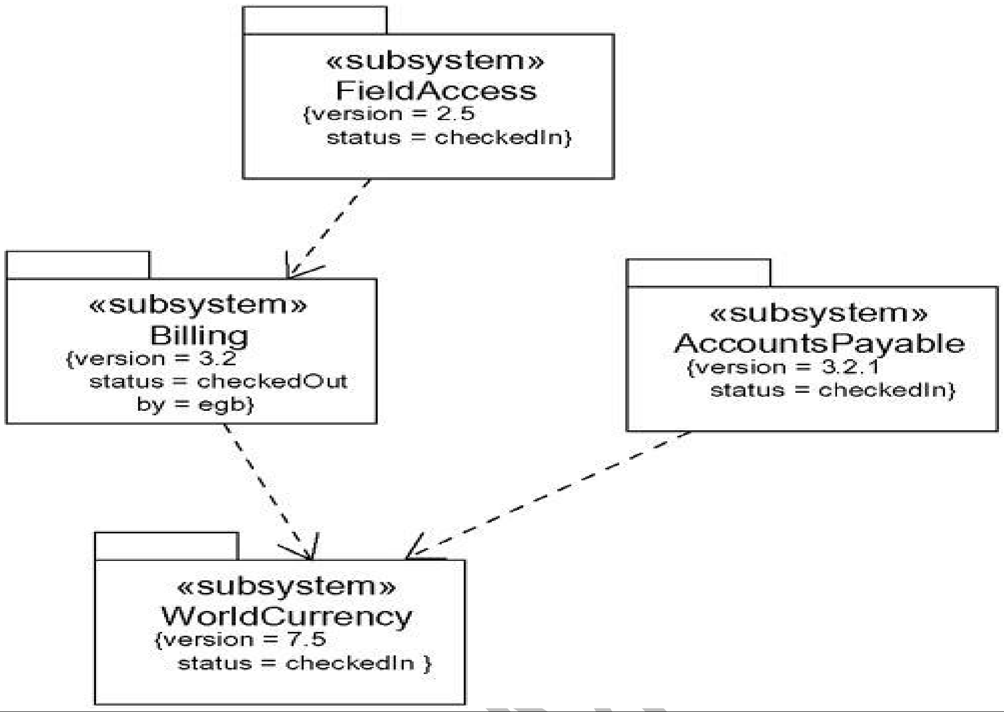
Modeling New Building Blocks

6.3 Modeling New Properties

- ✓ The basic properties of the **UML's building blocks—attributes and operations for classes, the contents of packages**, and so on.

To model new properties,

- ✓ If you're convinced there's no other way to express these semantics, add this new property to an individual element or a stereotype.



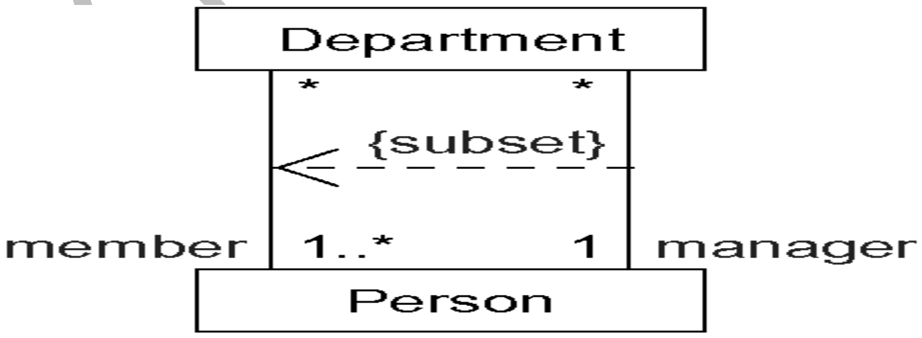
Modeling New Properties

6.3 **Modeling New Semantics**

To model new semantics,

- ✓ If you need to specify **your semantics** more precisely and formally, write your new **Semantics using OCL**(object constraint language).
- ✓ This diagram shows that **each Person may be a member of zero or more Departments and that each Department must have at least one Person as a member.**

.



Modeling New Semantics

13Q. EXPLAIN THE DIAGRAMS IN UML?

7 Diagrams

- ✓ Typically, you'll view the **Static parts of a system** using one of the four following diagrams.

1. Class diagram
2. Object diagram
3. Component diagram
4. Deployment diagram

- ✓ The five additional diagrams to view the **Dynamic parts of a system**.

1. Use case diagram
2. Sequence diagram
3. Collaboration diagram
4. Statechart diagram
5. Activity diagram

- ✓ The UML defines these nine kinds of diagrams.

- ✓ For this reason, the UML includes nine such diagrams:

10. Class diagram
11. Object diagram
12. Use case diagram
13. Sequence diagram
14. Collaboration diagram
15. State chart diagram
16. Activity diagram
17. Component diagram
18. Deployment diagram

There are 9 types of Diagrams in UML, which are classified into 2 types

3. **Structural Diagrams** (static diagrams)
4. **Behavioral Diagrams** (Dynamic diagrams)

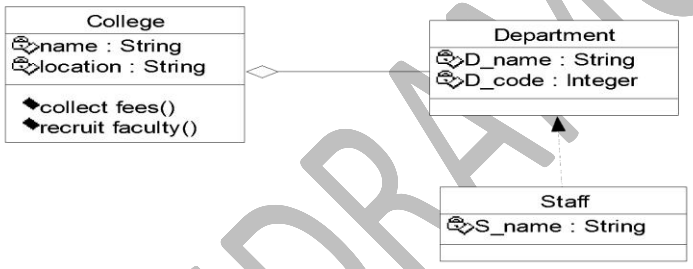
2. Structural Diagrams (static diagrams)

These are of 4 types

5. Class Diagram

✓ A Class diagram shows a set of classes, interfaces, and collaborations and their relationships.

✓ A class consists of class name, attributes, operations and responsibilities.



6. Object diagram

✓ Object diagrams represent static snapshots of instances of the things found in class diagrams.

✓ These diagrams address the static design view or static process view of a system.

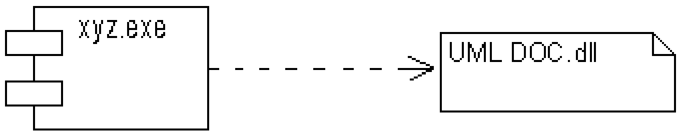
✓ An object diagram shows a set of objects and their relationships.

7. Component diagram

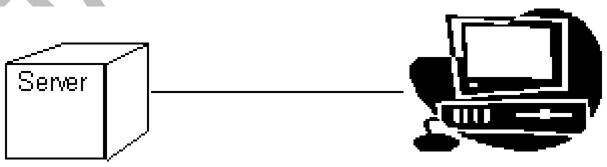
✓ A component diagram shows the organizations and dependencies among a set of components.

✓ Component diagrams address the static implementation view of a system.

- ✓ They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations
- ✓ It shows the organizations and dependencies among set of compo nets. The static(view) implementation view of a system. It displays the high level packaged structure of the code itself.



8. **Deployment diagram**
    - ✓ It shows the configuration of runtime processing nodes and the components that live on them it address the static deployment view of architecture.
    - ✓ It displays the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of code units.

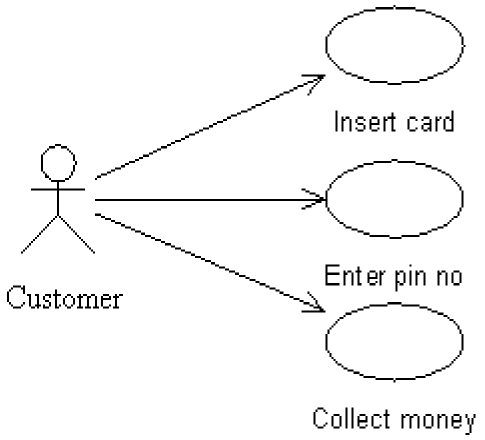


3. **Behavioral Diagrams (Dynamic diagrams)**

    These are of 5 types

6. **Use cas e diagram**
    - ✓ A use case diagram shows a set of use cases and actors and their relationships

- ✓ Use case diagrams address the static use case view of a system.
- ✓ These diagrams are especially important in organizing and modeling the behaviors of a system.

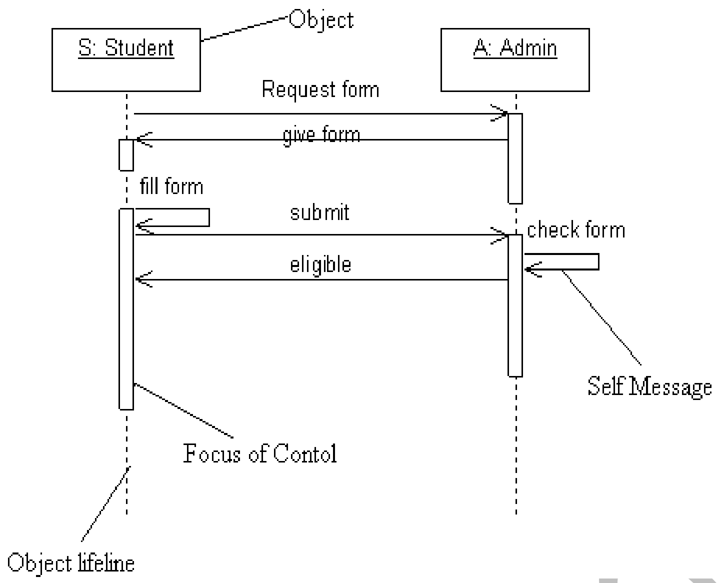


Interaction Diagrams

- ✓ Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams
- ✓ Interaction diagrams address the dynamic view of a system.

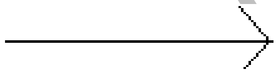
7. **A sequence diagram**

- ✓ Sequence diagram emphasizes the time ordering of messages. It mainly shows set of objects and the messages send /receive by those objects which is concerned with time ordering of messages.
- ✓ It displays the time sequence of the objects participating in the interaction. This consists of the vertical dimension (time) and horizontal dimension (different objects).



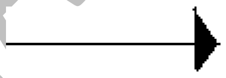
To show interaction between objects we use 3 types of messages.

Simple Messages:



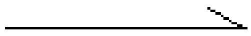
A Simple message shows how control is passed from one object to other without describing communication in detail i.e. without indicating whether it is synchronous or asynchronous message.

Synchronous Messages:



If sender object waits for a reply from receiver object from destination, such messages are called Synchronous messages. Here, only one object can send a message at a given instance of time.

Asynchronous Messages:



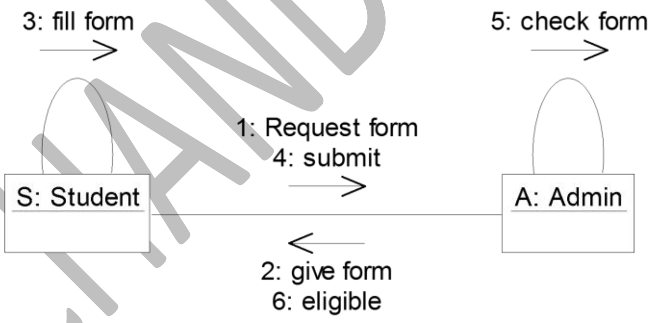
If sender object continues executing while target is processing the message then such messages are said to be Asynchronous messages. Here, multiple messages are executed at a time.

Object Lifeline: An Object life line is vertical dashed lines that represent the existence of an object over a period of time.

Focus of Control: It is represented by rectangle that shows the period of time during which an object performs some actions.

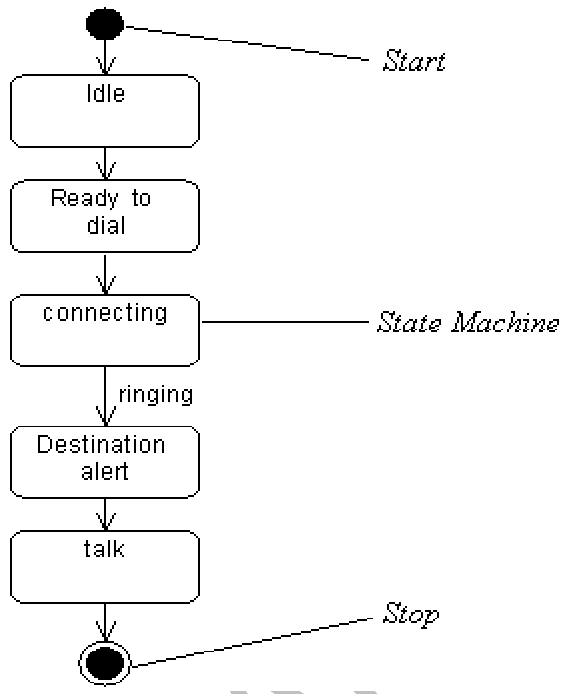
8. A collaboration diagram

- ✓ A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages
- ✓ Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.



9. Statechart diagram

- ✓ A statechart diagram shows a state machine, consisting of states, transitions, events, and activities.
- ✓ Statechart diagrams address the dynamic view of a system.
- ✓ They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object



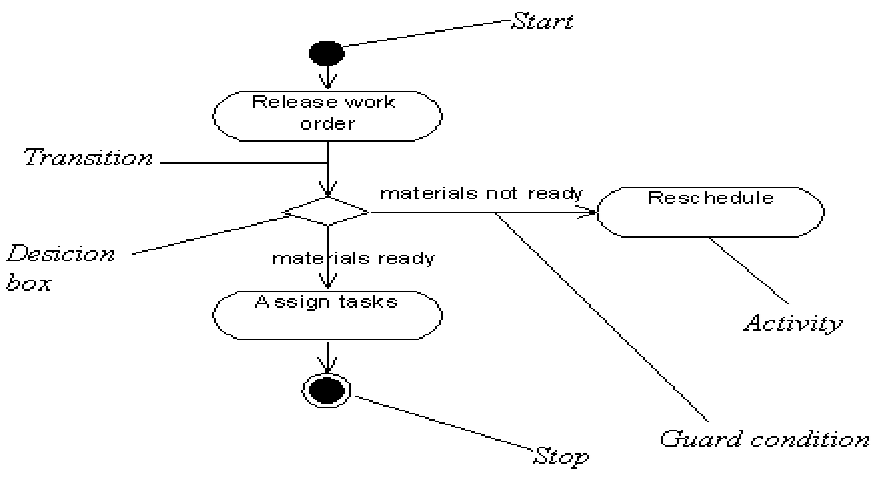
10. <u>**Activity diagram**</u>

- ✓ An activity diagram is a <u>special kind of a statechart diagram that shows the flow from activity to activity within a system</u>
- ✓ Activity diagrams address the <u>dynamic view of a system</u>
- ✓ They are especially important in modeling the function of a system and emphasize the flow of control among objects.

<u>**Activity**</u>**:** It is a major task that must take place in order to fulfill an operation contract.

<u>**Initial Activity**</u>*:* This shows the <u>starting point of the flow</u>. It is denoted by solid circle

<u>**Final Activity**</u>*:* This shows the <u>end of the flow in the activity diagram</u>. It is denoted by a solid circle nested in a circle.



Decision Box: A point in an Activity diagram where a flow splits into several mutually exclusive guarded flows. It has one incoming transition and two outgoing transitions.

Forking and Joining: We use synchronization bar to specify the forking and joining of parallel flows of control.

A ***synchronization bar*** is a thick horizontal or vertical line.
A ***Fork*** may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control.

A ***Join*** may have two or more incoming transitions and one outgoing transition.

14Q. EXPLAIN THE COMMON MODELING TECHNIQUES OF DIAGRAMS IN UML?

COMMON MODELING TECHNIQUES OF DIAGRAMS

1 Modeling Different views of a System
2 Modeling Different levels of Abstraction
3 Modeling Complex Views

1 Modeling Different Views of a System

To model a system from different views,

- ✓ **Decide which views you need to best express the architecture of your system** and to expose the technical risks to your project
- ✓ For each of these views, decide which artifacts you need **to create to capture the essential details of that view.**
- ✓

Use case view	Use case diagrams
Design view	Class diagrams (for structural modeling) Interaction diagrams (for behavioral modeling)
Process view	None required
Implementation view	None required
Deployment view	None required

Similarly, if yours is a **client/server system**, you'll probably want to include component diagrams and deployment diagrams to model the physical details of your system.

Use case view	Use case diagrams Activity diagrams (for behavioral modeling)
Design view	Class diagrams (for structural modeling) Interaction diagrams (for behavioral modeling) Statechart diagrams (for behavioral modeling)
Process view	Class diagrams (for structural modeling) Interaction diagrams (for behavioral modeling)
Implementation view	Component diagram
Deployment view	Deployment diagrams

8.2 Modeling Different Levels of Abstraction

Basically, there are two ways to model a system at different levels of abstraction:

1. By presenting diagrams with **different levels of detail** against the same model.
2. **By creating models** at different levels of abstraction with diagrams that trace from one model to another.

- ✓ To model a system at different levels of abstraction by presenting diagrams with **different levels** of detail,

8.3 Modeling Complex Views

To model complex views,

- ✓ First, convince yourself there's no **meaningful way to present this information** at a higher level of abstraction, perhaps eliding some parts of the diagram and retaining the detail in other parts.

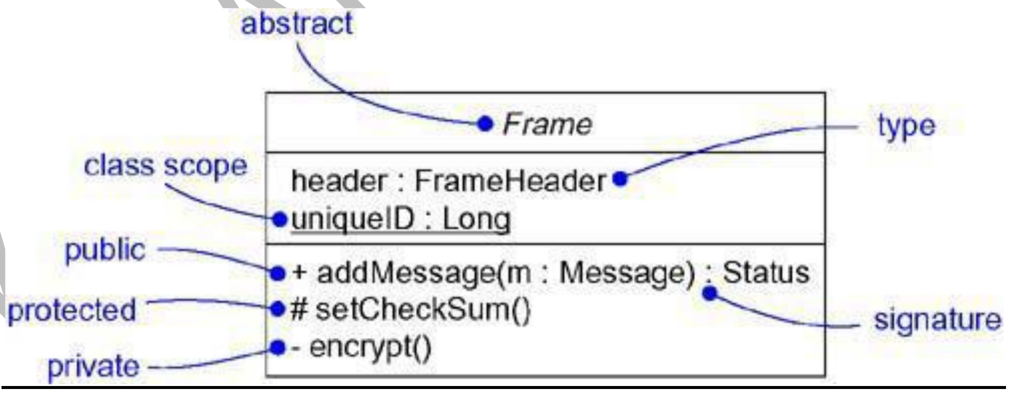
- ✓ If you've hidden as much detail as you can and your diagram is still complex, consider **grouping some of the elements in packages** or in higher level collaborations, then render only those packages or collaborations in your diagram.

.

15 Q. EXPLAIN THE ADVANCED CLASSES IN UML?

Advanced classes

9.1 Classifiers
9.2 Visibility
9.3 Scope
9.4 Abstract,Root,Leaf, and Polymorphic Elements
9.5 Multiplicity
9.6 Attributes
9.7 Operations
9.8 Template classes
9.9 Standard Elements



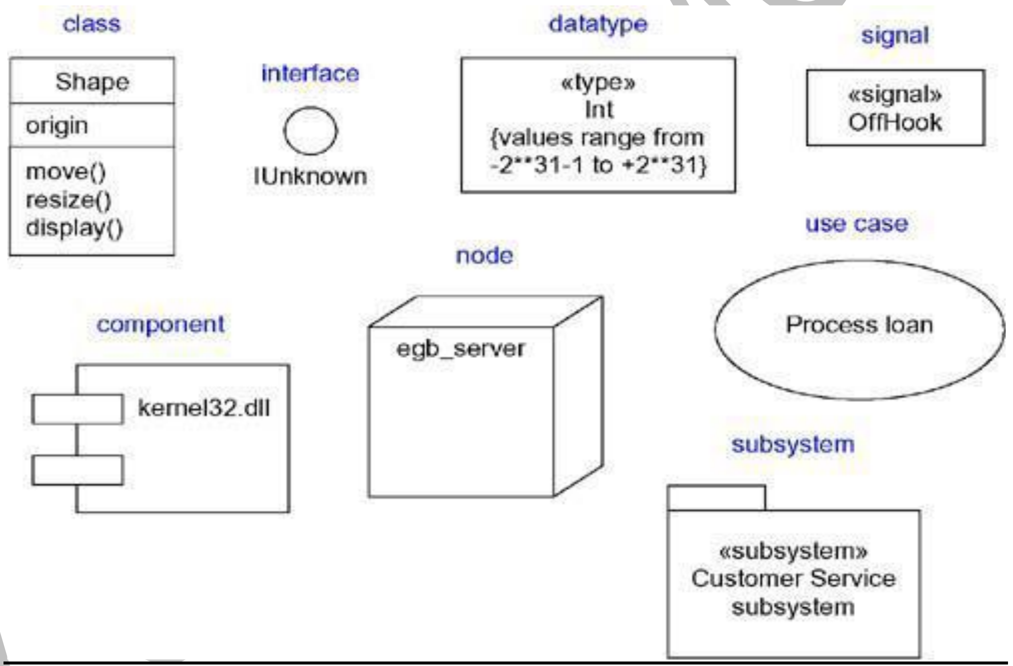
Advanced classes

Terms and concepts

9.1 Classifiers

✓

Interface	A collection of operations that are used to specify a service of a class or a component
Datatype	A type whose values have no identity, including primitive built-in types (such as numbers and strings), as well as enumeration types (such as Boolean)
Signal	The specification of an asynchronous stimulus communicated between instances
Component	A physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces
Node	A physical element that exists at run time and that represents a computational resource, generally having at least some memory and often processing capability
Use case	A description of a set of a sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor
Subsystem	A grouping of elements of which some constitute a specification of the behavior offered by the other contained elements

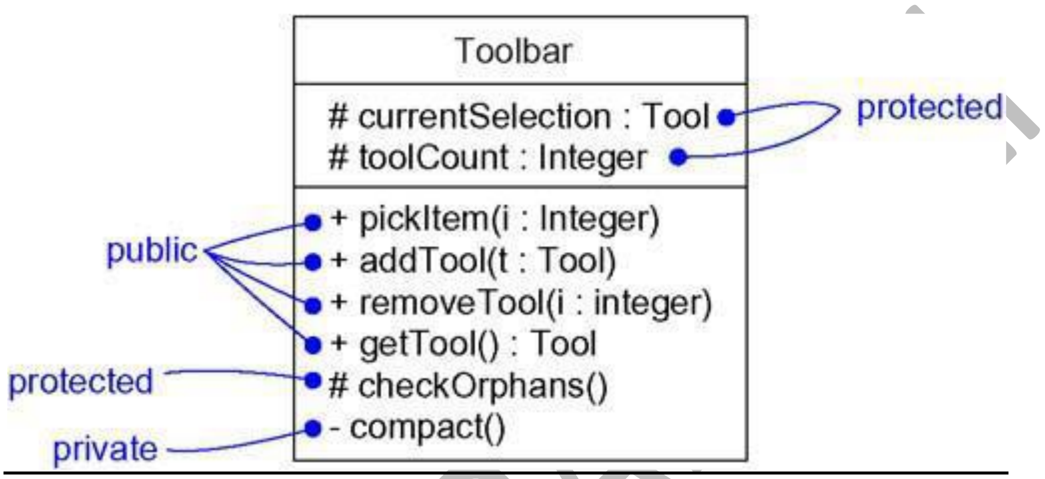


Classifiers

9.2 Visibility:

- ✓ One of the most important details you can specify for a classifier's attributes and operations is its visibility. The visibility of a feature specifies whether it can be used by other classifiers. In the UML, you can specify any of three levels of visibility.
- ✓ Figure shows a mix **of public, protected, and private** figures for the class Toolbar.

public	Any outside classifier with visibility to the given classifier can use the feature; specified by prepending the **symbol +**
protected	Any descendant of the classifier can use the feature; specified by prepending the **symbol #**
private	Only the classifier itself can use the feature; specified by prepending the **symbol -**



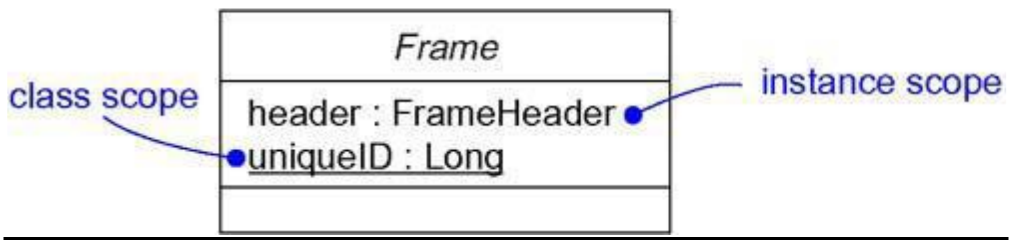
Visibility

9.3 Scope

✓ In the UML, you can specify **two kinds of owner scope**

1). Instance	Each instance of the classifier holds its own value for the feature.
2). Classifier	There is just one value of the feature for all instances of the classifier.

✓ As a figure shows, a feature that is **classifier scoped is rendered by underlining the feature' name**. No adornment means that the feature is **instance scoped.**

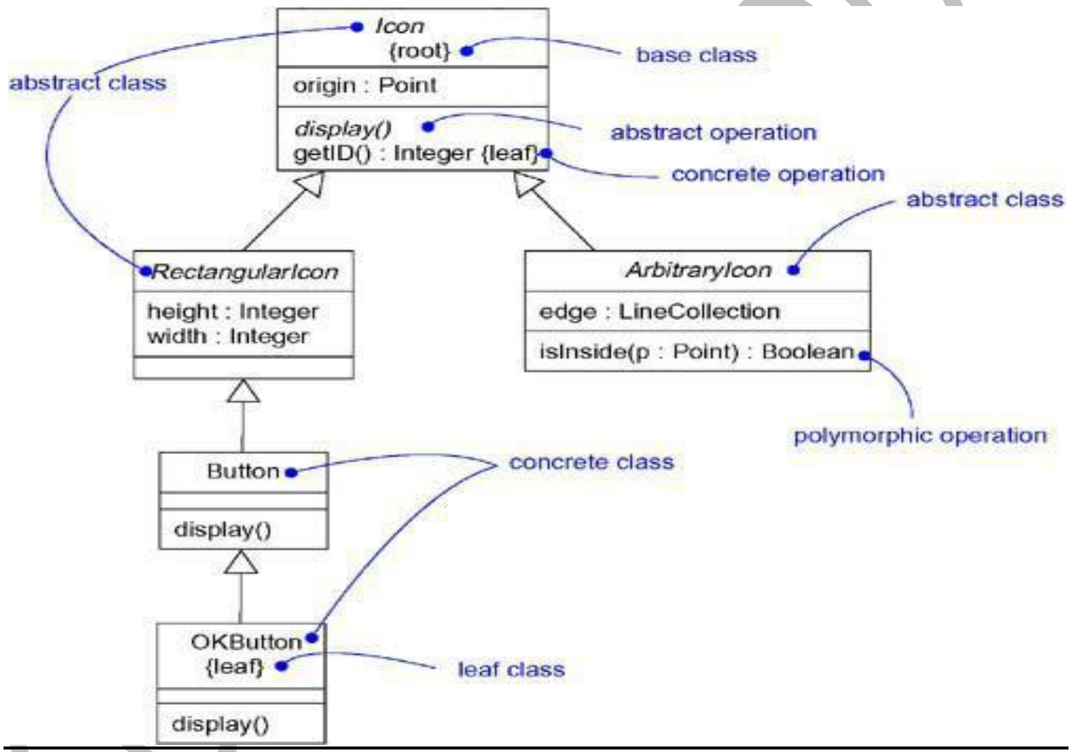


Scope

9.4 Abstract, Root, Leaf, and Polymorphic Elements

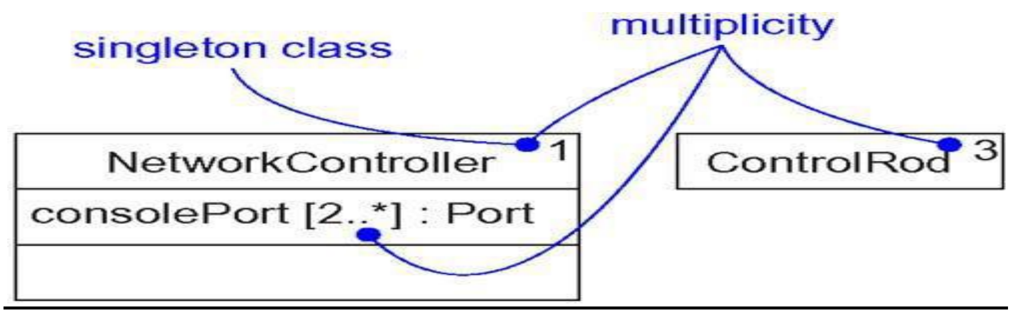
- ✓ Typically, an operation **is polymorphic, which means that, in a hierarchy of classes, you can specify operations with the same signature at different points in thehierarchy.** Ones in the child classes override the behavior of ones in the parent classes.
- ✓ For example, display and isInside are both polymorphic operations
- ✓ **Root class** :Specify that class may have no Parents
- ✓ **Leaf class** : Specify that class may have no Childs

Abstract and Concrete Classes and Operations



9.5 Multiplicity

- ✓ **The number of instances a class** may have is called its multiplicity. Multiplicity is a specification of the range of allowable cardinalities an entity may assume.
- ✓ In the UML, you can specify the multiplicity of a class by writing a multiplicity expression in the **upper-right corner of the class icon.**
- ✓ .



Multiplicity

9.6 Attributes

In its full form, the syntax of an attribute in the UML is

[visibility] name [multiplicity] [: type]

[= initial-value] [{property-string}]

9.7 Operations

In its full form, the **syntax** of an operation in the UML is

[visibility] name [(parameter-list)]

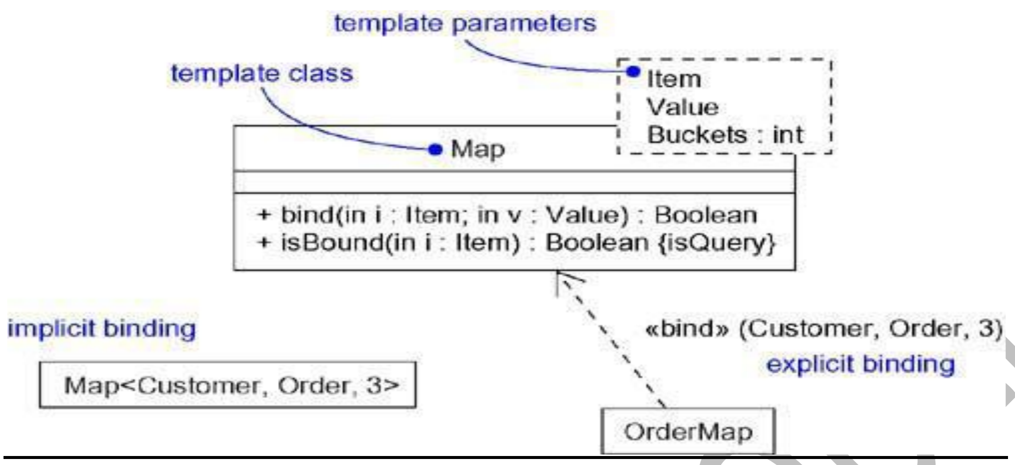
[: return-type] [{property-string}]

Direction may be any of the following values:

in	An input parameter; may not be modified
out	An output parameter; may be modified to communicate information to the caller
inout	An input parameter; may be modified

9.8 Template Classes

- ✓ A template is a **parameterized element.**
- ✓ A template includes **slots for classes, objects, and values, and these slots serve as the template's parameters**



Template Classes

9.9 Standard Elements

All of the UML's extensibility mechanisms apply to classes

The UML defines four standard stereotypes that apply to classes.

1. metaclass	Specifies a classifier whose objects are all classes
2. powertype	Specifies a classifier whose objects are the children of a given parent
3. stereotype	Specifies that the classifier is a stereotype that may be applied to other elements
4. utility	Specifies a class whose attributes and operations are all class scoped

16 Q. EXPLAIN THE COMMON MODELING TECHNIQUES OF ADVANCED CLASSES IN UML?

COMMON MODELING TECHNIQUES FOR ADVANCED CLASSES

10.1 Modeling the Semantics of a Class

- ✓ Specify the **responsibilities of the class.**
- ✓ Specify **the pre- and post conditions of each operation**, plus the invariants of the class as a whole, using structured textprecondition, post condition, and invariant) attached to the operation or classby a dependency relationship.
- ✓ Specify a **state machine for the class**. A state machine is a behavior that specifies the

- ✓ Specify a **collaboration that represents the class.**

17 Q. EXPLAIN THE ADVANCED RELATIONSHIPS IN UML?

Advanced Relationships

11.1 Dependency

- ✓ First, there are eight stereotypes that apply to dependency relationships among **classes and objects in class diagrams.**

1	**bind**	Specifies that the source instantiates the target template using the given actual parameters
2	**derive**	Specifies that the source may be computed from the target
3	**friend**	Specifies that the source is given special visibility into the target
4	**instanceOf**	Specifies that the source object is an instance of the target classifier
5	**instantiate**	Specifies that the source creates instances of the target
6	**powertype**	Specifies that the target is a powertype of the source; a powertype is a classifier whose objects are all the children of a given parent
7	**refine**	Specifies that the source is at a finer degree of abstraction than the target
8	**use**	Specifies that the semantics of the source element depends on the semantics of the public part of the target

There are two stereotypes that apply to dependency relationships among **packages.**

9	**access**	Specifies that the source package is granted the right to reference the elements of the target package
10	**import**	A kind of access that specifies that the public contents of the target package enter the flat namespace of the source, as if they had been declared in the source

Two stereotypes apply to dependency relationships among **use cases**:

11	**extend**	Specifies that the target use case extends the behavior of the source
12	**include**	Specifies that the source use case explicitly incorporates the behavior of another use case at a location specified by the source

There are three stereotypes when modeling **interactions among objects.**

13	**become**	Specifies that the target is the same object as the source but at a later point in time and with possibly different values, state, or roles
14	**call**	Specifies that the source operation invokes the target operation
15	**copy**	Specifies that the target object is an exact, but independent, copy of the source

One stereotype you'll encounter in the context of **state machines is**

16	**send**	Specifies that the source operation sends the target event

Finally, one stereotype that you'll encounter in the context of organizing the elements of your **system into subsystems and models is**

17	**trace**	Specifies that the target is an historical ancestor of the source

11.2 <u>Generalization</u>

- ✓ A generalization is a relationship between a **general thing (called the superclass or parent) and a more specific kind of that thing(called the subclass or child).**
- ✓

1	**implementation**	Specifies that the child inherits the implementation of the parent but does not make public nor support its interfaces, thereby violating substitutability

Next, there are four standard constraints that apply to generalization relationships

1	**complete**	Specifies that all children in the generalization have been specified in the model and that no additional children are permitted
2	**incomplete**	Specifies that not all children in the generalization have been specified (even if some are elided) and that additional children are permitted
3	**disjoint**	Specifies that objects of the parent may have no more than one of the children as a type
4	**overlapping**	Specifies that objects of the parent may have more than one of the children as a type

11.3 Association

✓

✓ For advanced uses, there are a number of other properties you can use to model subtle details, such as

Navigation

Vision

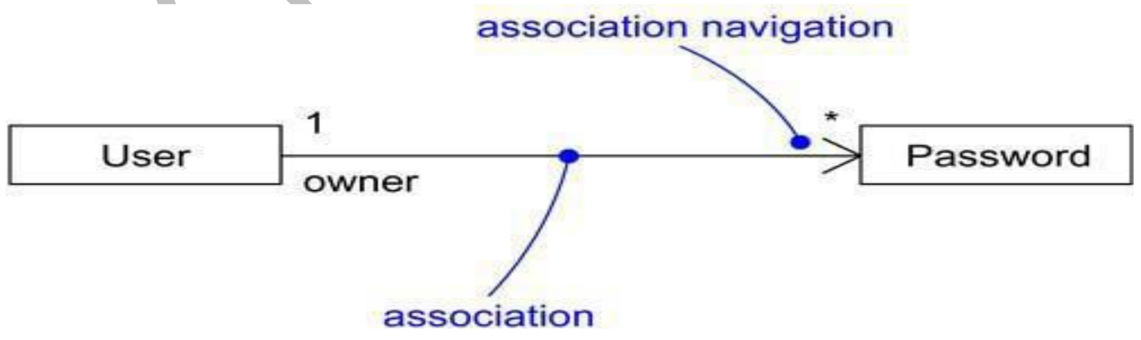
Qualification

Various flavors of aggregation.

Navigation

✓ **Given a plane,unadorned association between two classes**, such as Book and Library, it's possible to navigate from objects of one kind to objects of the other kind.

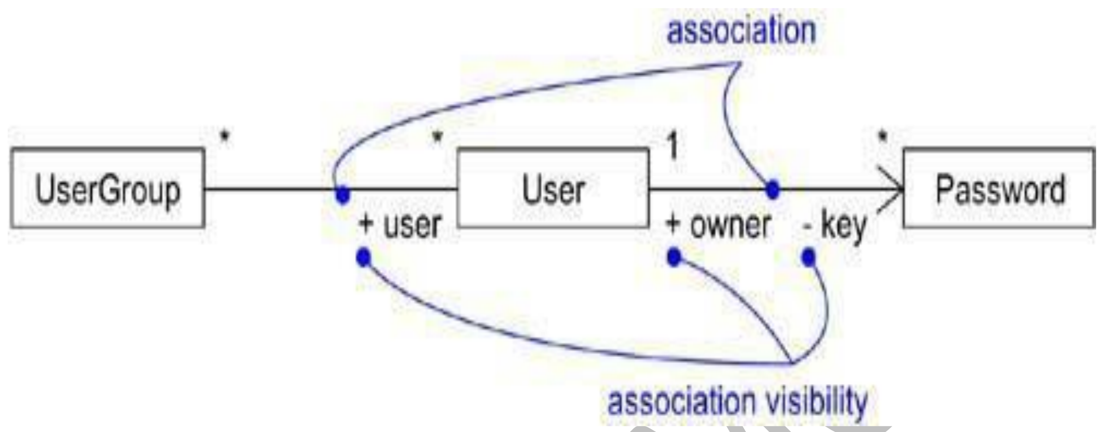
✓



Navigation

Visibility

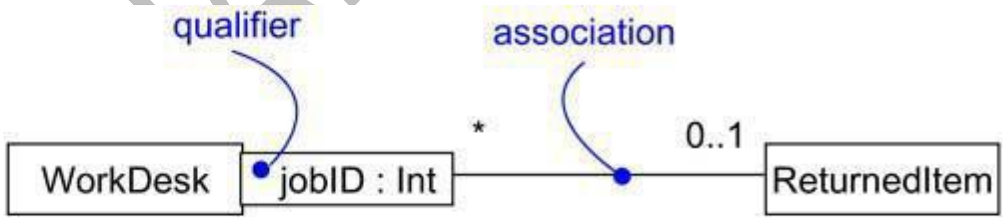
- ✓ **Given an association between two classes, objects of one class can see and navigate to objects of the other**, unless otherwise restricted by an explicit statement of navigation.
- ✓



Visibility

Qualification

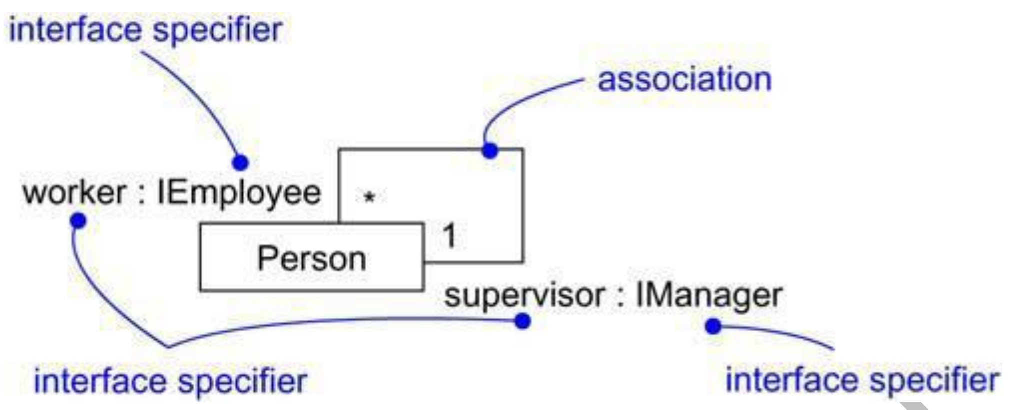
- ✓ You render a qualifier as a small rectangle attached to the end of an association, placing the attributes in the rectangle



Qualification

Interface Specifier

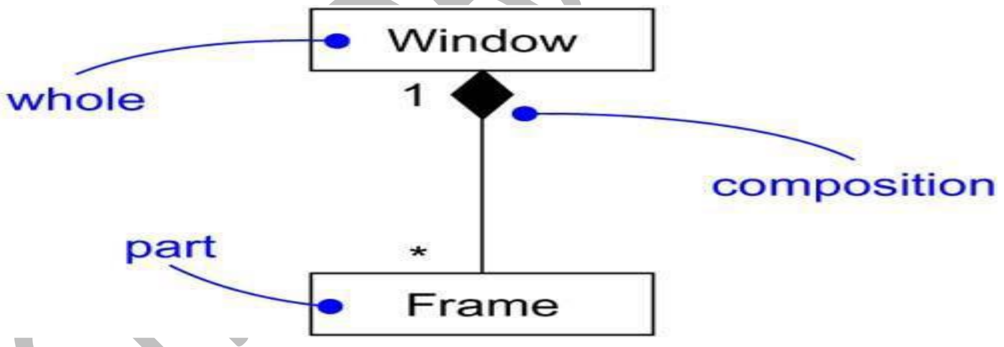
- ✓ An interface is a collection of operations that are used to **specify a service of a class or a component.**



Interface Specifier

Composition

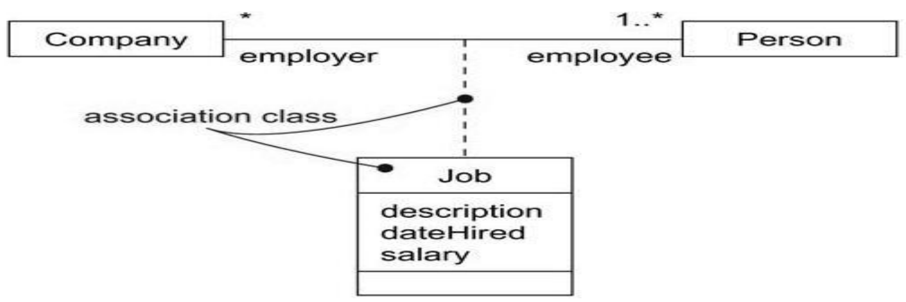
- ✓ Simple aggregation is entirely conceptual and does nothing more than distinguish a "whole" from a "part."
- ✓ **Composition is a form of aggregation, with strong ownership and coincident lifetime as part of the whole.**
- ✓



Composition

Association Classes

- ✓ In an association between two classes, the association itself might have properties.
- ✓ An association class can be seen as **an association that also has class properties, or as a class that also has association properties.**
- ✓



Association Classes

Constraints

UML defines five constraints that may be applied to association relationships.

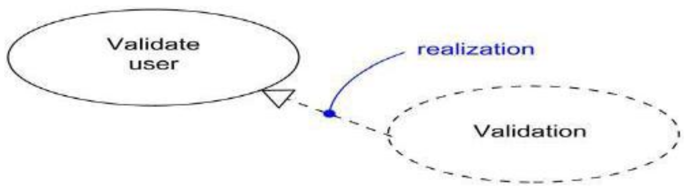
1	**implicit**	Specifies that the relationship is not manifest but, rather, is only conceptual
2	**ordered**	Specifies that the set of objects at one end of an association are in an explicit order
3	**changeable**	Links between objects may be added, removed, and changed freely
4	**addOnly**	New links may be added from an object on the opposite end of the association
5	**frozen**	A link, once added from an object on the opposite end of the association, may not be modified or deleted

Finally, there is one constraint for managing related sets of associations:

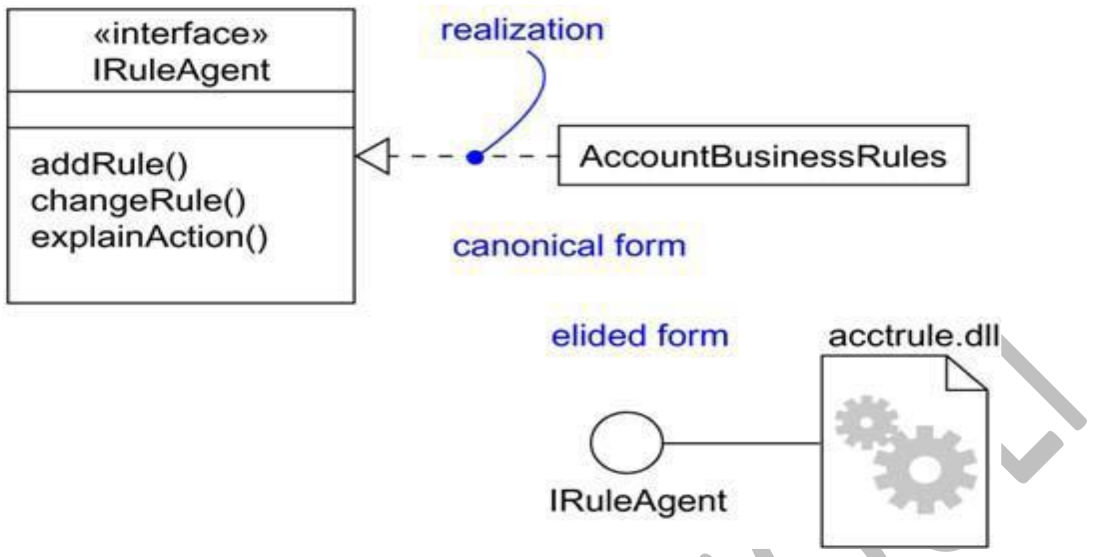
1	**xor**	Specifies that, over a set of associations, exactly one is manfest for each associated object

Realization

- ✓ Realization is sufficiently different from dependency, generalization, and association relationships that it is treated as a separate kind of relationship.
- ✓ A realization is **a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.**
- ✓ Graphically, a realization is rendered as a dashed directed line with a large open arrowhead pointing to the classifier that specifies the contract.
- ✓



Realization of an Interface



Realization

18 Q. EXPLAIN THE COMMON MODELING TECHNIQUES ADVANCED RELATIONSHIPS IN UML?

Common Modeling Techniques for advanced Relationships

12.1 Modeling Webs of Relationships

When you model these webs of relationships,

- ✓ **Don't begin in isolation**. Apply use cases and scenarios to drive your discovery of the relationships among a set of abstractions.
- ✓ In general, **start by modeling the structural relationships that are present**. These reflect the static view of the system and are therefore fairly tangible.
- ✓ Next**, identify opportunities for generalization/specialization relationships**; use multiple inheritance sparingly.

19 Q. EXPLAIN THE OBJECT DIAGRAMS IN UML?

Object Diagram

- An object diagram is a diagram that shows a **set of objects and their relationships at a point in time.**
- Graphically, an o**bject diagram is a collection of vertices and arcs**

- An object diagram is a special kind of diagram and shares the same common properties as all other diagrams—that is, a name and graphical contents that are a projection into a model

Contents

- Object diagrams commonly contain
    - Objects
    - Links
- Like all other diagrams, object diagrams may contain notes and constraints.
- Object diagrams may also contain packages or subsystems

Common Uses

- You use object diagrams to model the static design view or static process view of a system just as you do with class diagrams
- When you model the static design view or static process view of a system, you typically use object diagrams in one way:
    - To model object structures

Modeling Object Structures

- Modeling object structures involves taking a snapshot of the objects in a system at a given moment in time.
- An object diagram represents one static frame in the dynamic storyboard represented by an interaction diagram

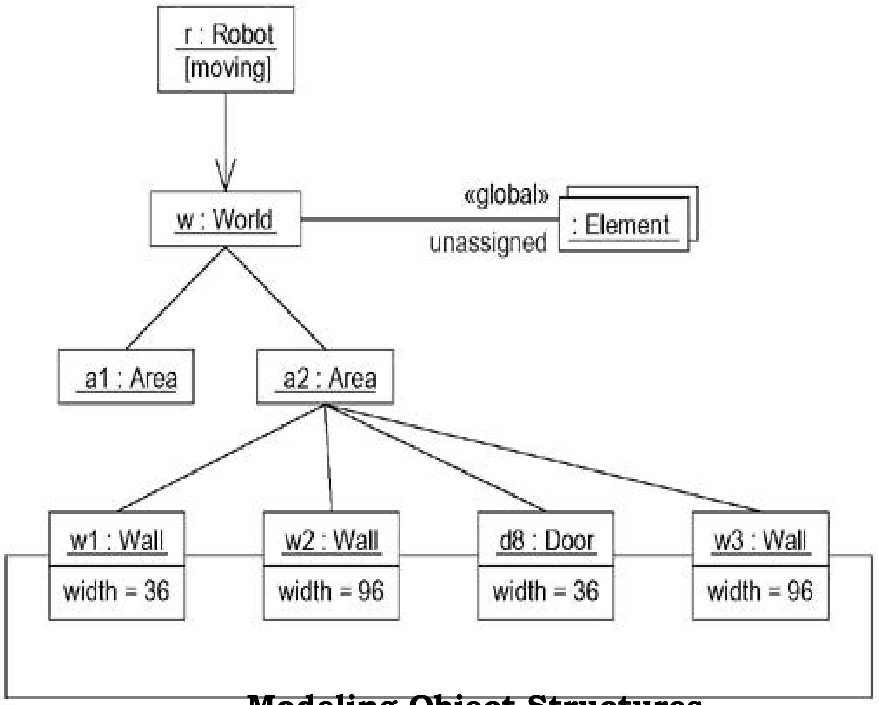
20 Q. EXPLAIN THE COMMON MODELING TECHNIQUES OF OBJECT DIAGRAMS IN UML?

Common Modeling Techniques

Modeling Object Structures

- An object diagram shows one set of objects in relation to one another at one moment in time.
- To model an object structure,
- **Identify the mechanism you'd like to model**.
    - **For each mechanism, identify the classes, interfaces, and other elements that participate in this collaboration**;
    - identify the relationships among these things, as well.
    - Consider one scenario that walks through this mechanism.
    - Expose the state and attribute values of each such object, as necessary, to understand the scenario.

- Similarly, expose the links among these objects, representing instances of associations among them.



Modeling Object Structures

Forward and Reverse Engineering

- **Forward engineering an object diagram is theoretically possible but pragmatically of limited value**
- Component instances and node instances are things that live outside the running system and are amenable to some degree of forward engineering.
- Reverse engineering an object diagram is a very different thing
- To reverse engineer an object diagram,

  - Chose the target you want to reverse engineer. Using a tool or simply walking through a scenario, stop execution at a certain moment in time.

  - Identify the set of interesting objects that collaborate in that context and render them in an object diagram.

  - As necessary to understand their semantics, expose these object's states.

  - As necessary to understand their semantics, identify the links that exist among these objects.