UNIT – I

The UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML gives you a standard way to write a system's blueprints, covering conceptual things, such as business processes and system functions, as well as concrete things, such as classes written in a specific programming language, database schemas, and reusable software components.

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.

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.

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

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.

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+.

An Overview 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** means building models that are precise, unambiguous, and complete.
- **Constructing** the UML is not a visual programming language, but its models can be directly connected to a variety of programming languages
- **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

To understand the UML, you need to form a **conceptual model of the language**, and this requires learning three major elements:

1. Things
2. Relationships

3. Diagrams

Things in the UML

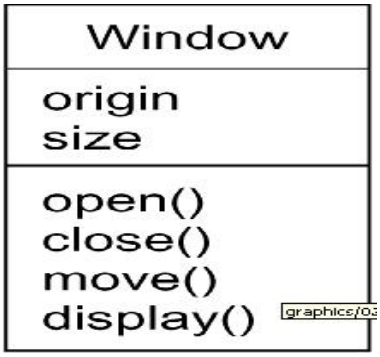
There are four kinds of things in the UML:

Structural things
Behavioral things
Grouping things
Annotational 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

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. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations.



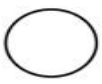
Interface

Interface is a collection of operations 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 complete behavior of a class or component or only a part of that behavior.

An interface is rendered as a circle together with its name. An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface

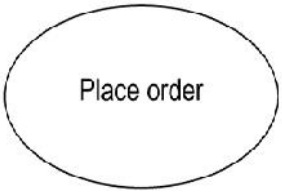


Collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Therefore, collaborations have structural, as well as behavioral, dimensions. A given class might participate in several collaborations.

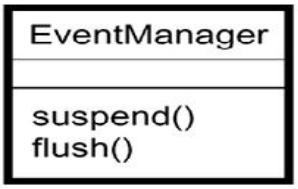
Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its name

Usecase

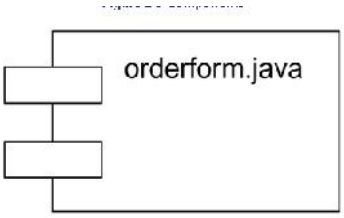
- Use case is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor
- Use case is used to structure the behavioral things in a model.
- A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse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



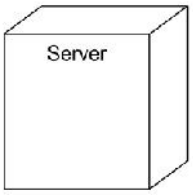
Active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations



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



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



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

Interaction
state machine

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

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

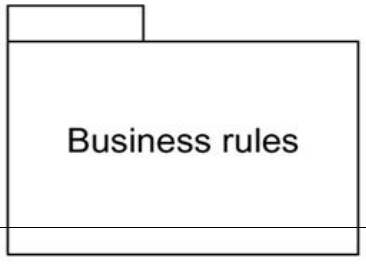
Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates

Grouping Things:-

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

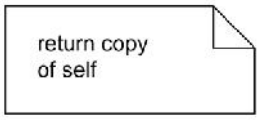
Package:-

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

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

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

Association is a structural relationship that describes a set of links, 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

Generalization is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent

Realization is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. Graphically a realization relationship is rendered as a cross between a generalization and a dependency relationship

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:
  - Class diagram
  - Object diagram
  - Use case diagram
  - Sequence diagram
  - Collaboration diagram
  - Statechart diagram
  - Activity diagram
  - Component diagram
  - Deployment diagram

Class diagram

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

Class diagrams that include active classes address the static process view of a system.

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

Use case 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

A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages

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

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

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

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

Deployment diagram

- A deployment diagram shows the configuration of run-time processing nodes and the components that live on them
- Deployment diagrams address the static deployment view of an architecture

Rules of the UML

The UML has semantic rules for

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

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

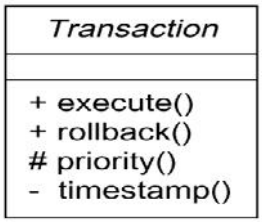
UML is made simpler by the presence of four common mechanisms that apply consistently throughout the language.

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

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

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.



Extensibility Mechanisms

The UML's extensibility mechanisms include

1. Stereotypes
2. Tagged values
3. Constraints

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
- A tagged value extends the properties of a UML building block, allowing you to create new information in that element's specification
- A constraint extends the semantics of a UML building block, allowing you to add new rules or modify existing ones

Architecture

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

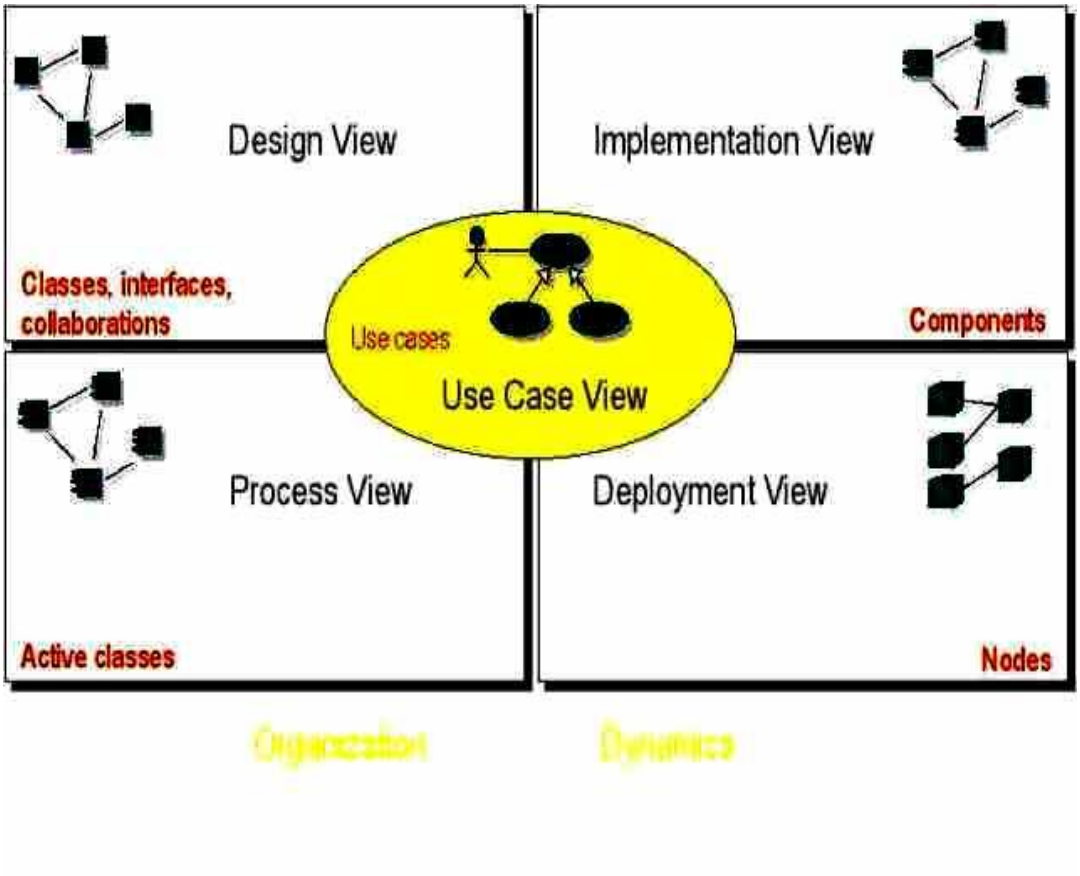
The selection of the structural elements and their interfaces by which the system is composed

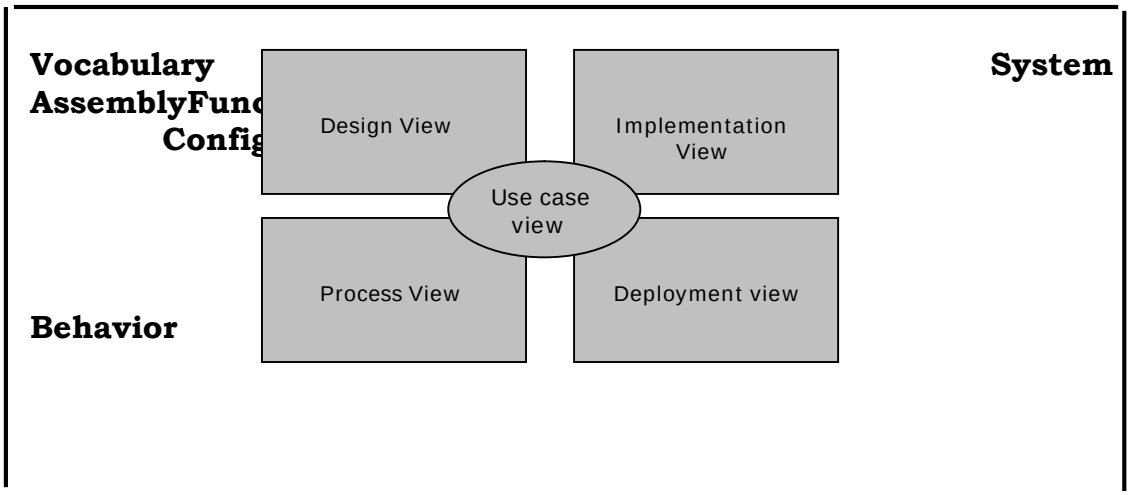
Their behavior, as specified in the collaborations among those elements

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.

Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns.





Modeling a System's Architecture

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.

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.

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

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.

Deployment Diagram

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.

Software Development Life Cycle(SDLC)

The UML is largely process-independent, meaning that it is not tied to any particular software development life cycle. However, to get the most benefit from the UML, it should consider a process that is

- Use case driven
- Architecture-centric
- Iterative and incremental

Use case driven means that use cases are used as a primary artifact for establishing the desired behavior of the system, for verifying and validating the system's architecture, for testing, and for communicating among the stakeholders of the project.

Architecture-centric means that a system's architecture is used as a primary artifact for conceptualizing, constructing, managing, and evolving the system under development.

An ***iterative process*** is one that involves managing a stream of executable releases. An is one that involves the continuous integration of the system's architecture to produce these releases, with each new release embodying incremental improvements over the other. Together, an iterative and incremental process is *risk-driven,* meaning that each new release is focused on attacking and reducing the most significant risks to the success of the project.

This use case driven, architecture-centric, and iterative/incremental process can be broken into phases. A *phase* is the span of time between two major milestones of the process, when a welldefined set of objectives are met, artifacts are completed, and decisions are made whether to move into the next phase.

There are four phases in the software development life cycle:

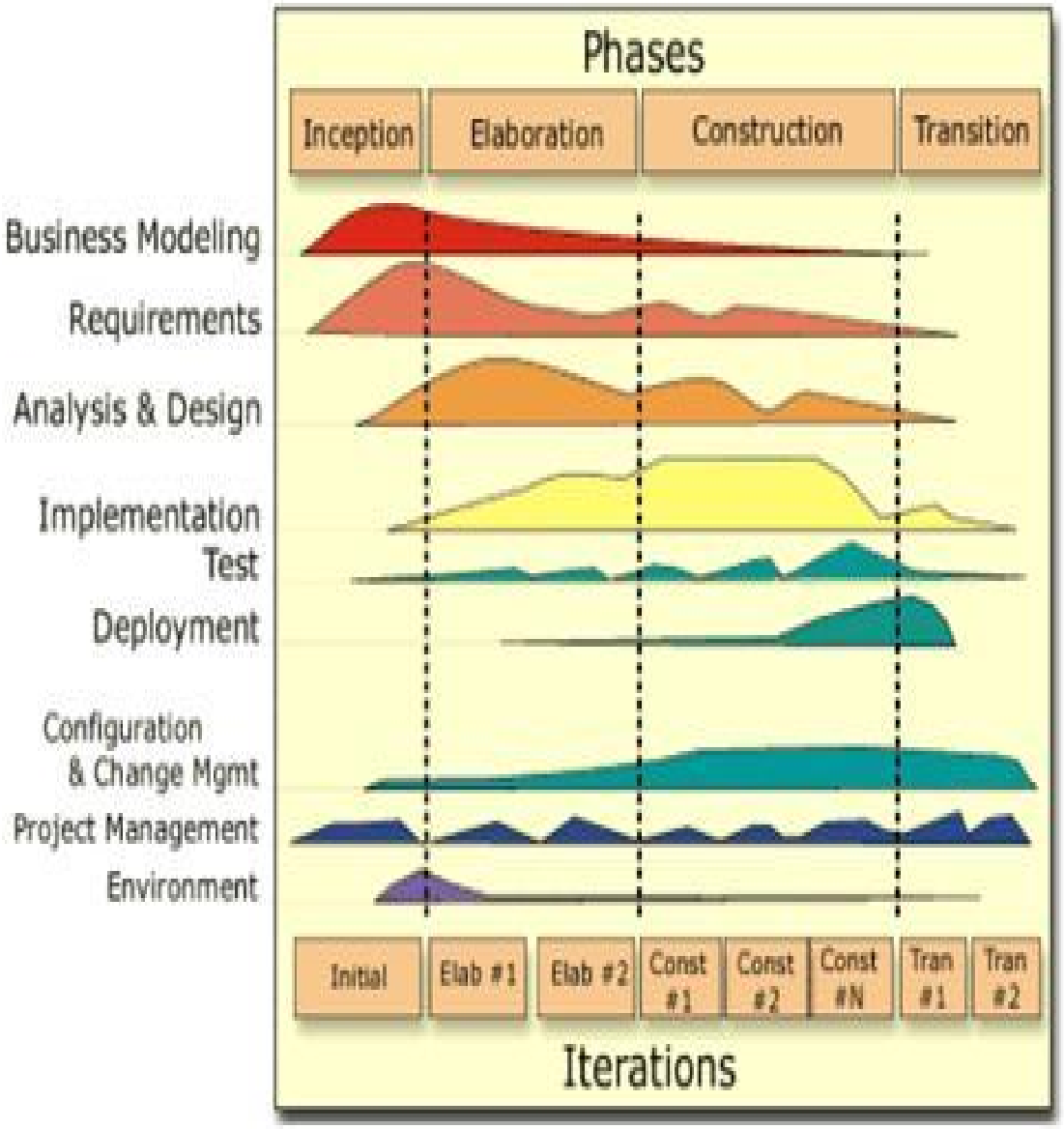
- Inception,
- Elaboration,
- Construction
- Transition.

Inception is the first phase of the process, when the seed idea for the development is brought up to the point of being at least internally sufficiently well-founded to warrant entering into the elaboration phase.

Elaboration is the second phase of the process, when the product vision and its architecture are defined. In this phase, the system's requirements are articulated, prioritized, and baselined. A system's requirements may range from general vision statements to precise evaluation criteria, each specifying particular functional or nonfunctional behavior and each providing a basis for testing.

Construction is the third phase of the process, when the software is brought from an executable architectural baseline to being ready to be transitioned to the user community. Here also, the system's requirements and especially its evaluation criteria are constantly reexamined against the business needs of the project, and resources are allocated as appropriate to actively attack risks to the project.

Transition is the fourth phase of the process, when the software is turned into the hands of the user community. Rarely does the software development process end here, for even during this phase, the system is continuously improved, bugs are eradicated, and features that didn't make an earlier release are added.


Phases
Inception
Elaboration
Construction
Transition
Business Modeling
Requirements
Analysis & Design
Implementation
Test
Deployment
Configuration
& Change Mgmt
Project Management
Environment
Initial
Elab #1
Elab #2
Const #1
Const #2
Const #N
Tran #1
Tran #2
Iterations

I-UNIT

DESCREPTIVE QUESTIONS

1. List the importance of UML modeling?
2. Compare and Contrast between Object Oriented Modeling and Conceptual Modeling?
3. List various steps involved in SDLC?
4. What does the word Unified in UML mean?
5. What are the thirteen diagramming or pictorial languages of UML?
6. What is the use of Model Driven Architecture (MDA)?
7. What is the goal of the Risk Aware principle?
8. What are the three categories of UML diagrams?
9. Compare and Contrast relationships between classes , interface and packages?
10. How do you show packages in a UML diagram? When working with multiple packages, what type of relationship is most commonly shown?
11. What are the two elements that are most commonly collected into packages?
12. How can the accessibility of the contents inside a package be determined?
13. What is the standard notation for specifying the diagrams inside a package?

SHORT QUESTIONS

1. Draw the Architecture of UML ?
2. What does USDP stand for?
3. List the principles of UML?
4. Define the terms:
   a) Classes b) Interface c) Packages
5. Draw the notations for Classes , Interface and Packages?
6. List various types of relationships?

OBJECTIVE QUESTIONS

1. Which one of the following is not principal of modeling ? [ ]
   a)Choose your models well
   b)Every model may not be expressed at different at different levels of decision
   c)The best models are connected to reality.
   d) No single model is sufficient.
2. Which one of the following view express the requirements of the system ?
   a)USE CASE b)Design c)Process
   d)Implementation.
3. UML is a _____ modeling language. [ ]
   a)general-purpose b)object-purpose c)architecture – purpose d) code-purpose
4. WE build models so that we can better _____ the system we are developing.
   a)misunderstand b)understand c) guide
   d)misguide.
5. _____ is a central past of all the activities that lead up to the deployment of good software.
   a) Modeling b)Coding c)Testing d)Analysis
6. Models tells us to _____ a system as it is (or) as we want it to be.
   a) Visualize b) Specify c) constructing
   d)Document.
7. In _____ any program can call any other program. [ ]
   a)Modular b)object c)procedural
   d)Component
8. An object contains _____ [ ]
   a)Attributies & Methods b)only attributes c)Only Methods
   d)Classes
9. A _____ is a generic template for objects.
   a)Program b)Class c)Procedure d)Method
10. The best models are connected to _____.
[ ]

a)Reality b)Functionality c)Casuality d)visualityl.

11. Which on of the following is grouping thing?

12. An _____ is a class which objects own one or more processes or threads and therefore can initiate control activity.

13. An interface is rendered as _____

14. An _____ is a collection of operations that specify a service of a class or component.

15. _____ helps us in protecting privacy of Objects.

16. _____ things are the names of UML models.

17. _____ things are the dynamic parts of UML models.

18. _____ is a special kind of association, representing a structural relationship between a whole and its parts.

19. A _____ relationship is rendered as a solid line with a hollow arrow head pointing to the parent.

20. Aggregation is a _____ kind of relationship.

UNIT-1 ANSWERS

Multiple Choice

1.(B) 2.(A) 3(A) 4(B) 5(A) 6(c) 7(d) 8(A) 9(B 10(A)

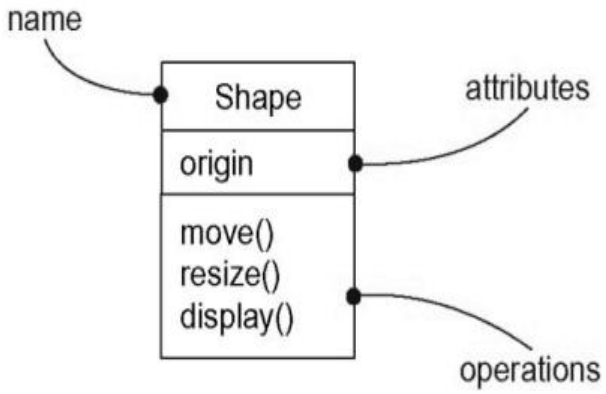
Fill in the Blanks

11.Package 12. Active class 13. Circle 14. Interface 15. Encapsulation 16. Structural 17. Behavioral 18. Aggregation 19. Generalization 20. has-a

UNIT – II

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



Graphical Representation of Class in UML

Terms and Concepts

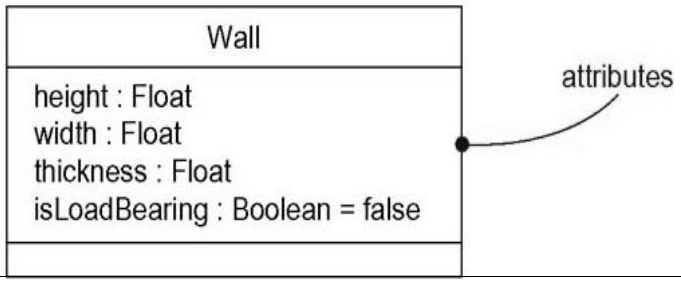
Names

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 Name **Path Name**

Attributes

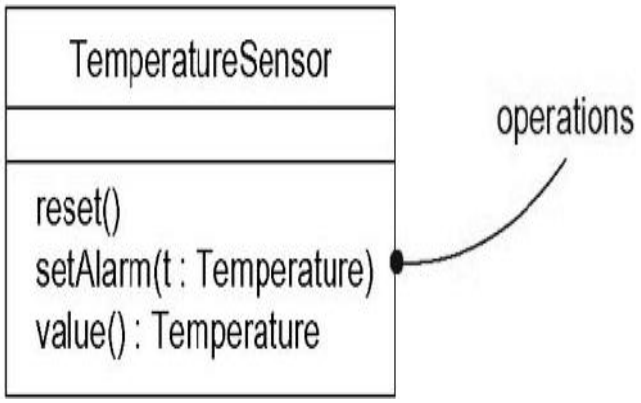
- An attribute is a named property of a class that describes a range of values that instances of the property may hold.
- A class may have any number of attributes or no attributes at all.
- An attribute represents some property of thing you are modeling that is shared by all objects of that class
-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



Attributes and Their Class

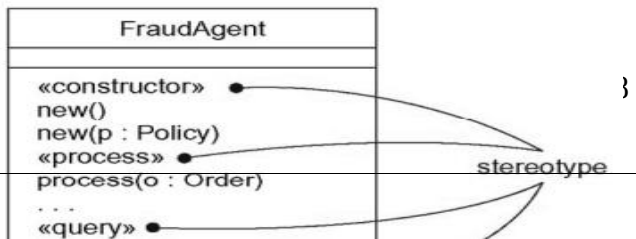
Operations

- 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



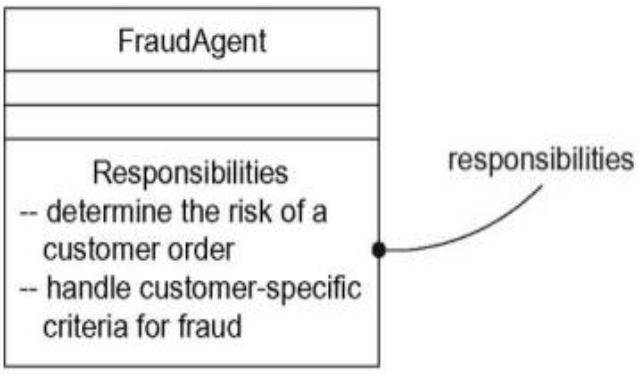
Organizing Attributes and Operations

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



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



Common Modeling Techniques

Modeling the Vocabulary of a System

- You'll use classes most commonly to model abstractions that are drawn from the problem you are trying to solve or from the technology you are using to implement a solution to that problem.
- They represent the things that are important to users and to implementers

- 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 Distribution of Responsibilities in a System

- 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.
- 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.
  - Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions, collapse tiny classes that have trivial responsibilities into larger ones, and reallocate responsibilities so that each abstraction reasonably stands on its own.
  - Consider the ways in which those classes collaborate with one another, and redistribute their responsibilities accordingly so that no class within a collaboration does too much or too little.

Modeling Nonsoftware Things

- Sometimes, the things you model may never have an analog in software
- Your application might not have any software that represents them
- To model nonsoftware 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 new building 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 Primitive Types

At the other extreme, the things you model may be drawn directly from the programming language you are using to implement a solution.

Typically, these abstractions involve primitive types, such as integers, characters, strings, and even enumeration types

To model primitive types

Model the thing you are abstracting as a type or an enumeration, which is rendered using class

notation with the appropriate stereotype.

If you need to specify the range of values associated with this type, use constraints.

Relationships

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

In object-oriented modeling, there are three kinds of relationships that are most important:

Dependencies

Generalizations

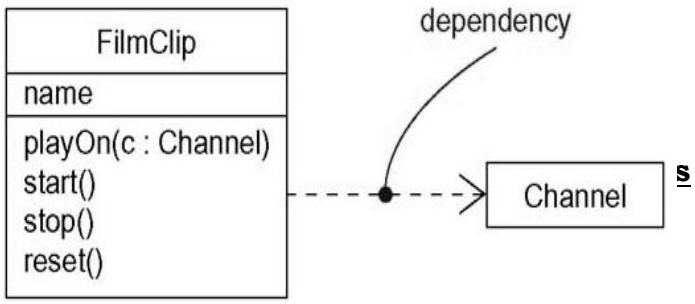
Associations

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.

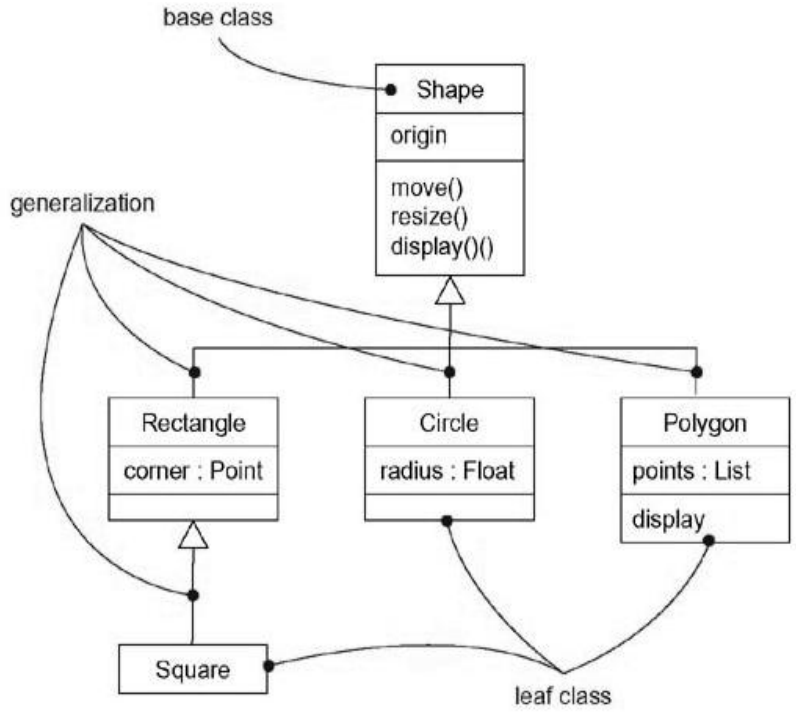
Most often, you will use dependencies in the context of classes to show that one class uses another class as an argument in the signature of an operation



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

- An operation of a child that has the same signature as an operation in a parent overrides the operation of the parent; this is known as polymorphism.
- Graphically generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent



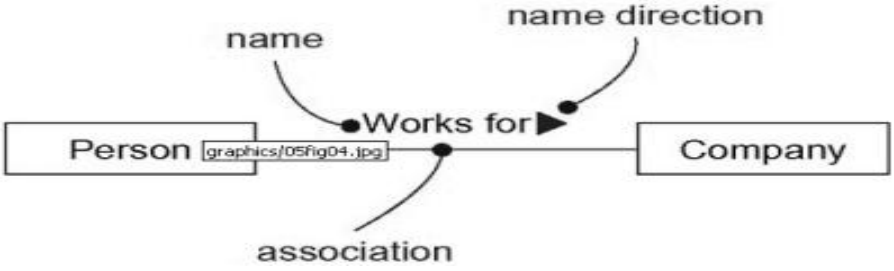
Generalization

Association

- 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.
- Beyond this basic form, there are four adornments that apply to association

Name

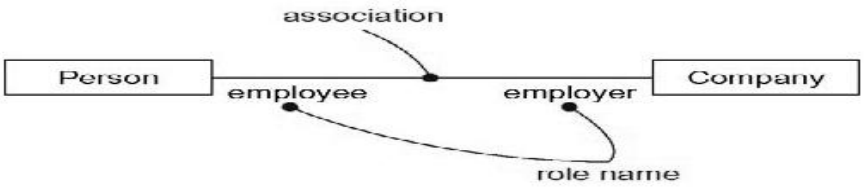
- An association can have a name, and you use that name to describe the nature of the relationship



Association Names

Role

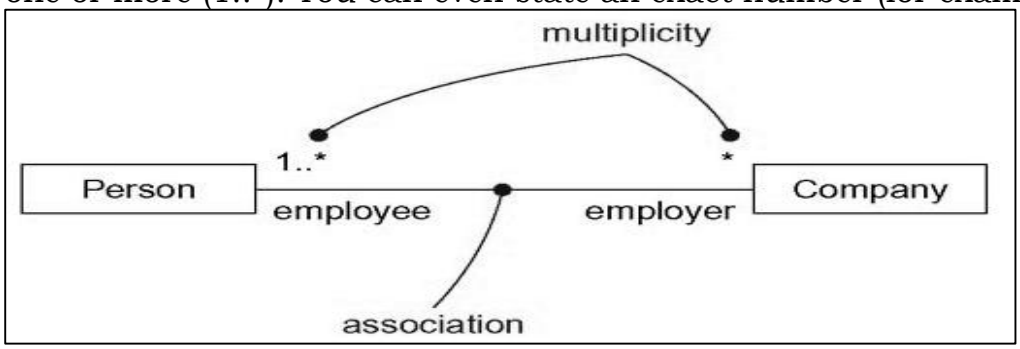
- When a class participates in an association, it has a specific role that it plays in that relationship;
- The same class can play the same or different roles in other associations.
- An instance of an association is called a link



Role Names

Multiplicity

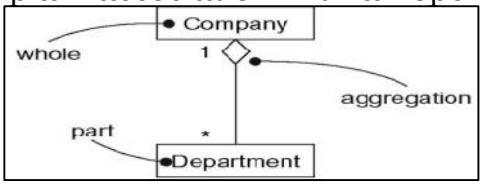
- In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association
- This "how many" is called the multiplicity of an association's role
- You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..*), or one or more (1..*). You can even state an exact number (for example, 3).



Multiplicity

Aggregation

- 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
- Aggregation is really just a special kind of association and is specified by adorning a plain association with an open diamond at the whole end



Aggregation

Common Modeling Techniques

Modeling 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.

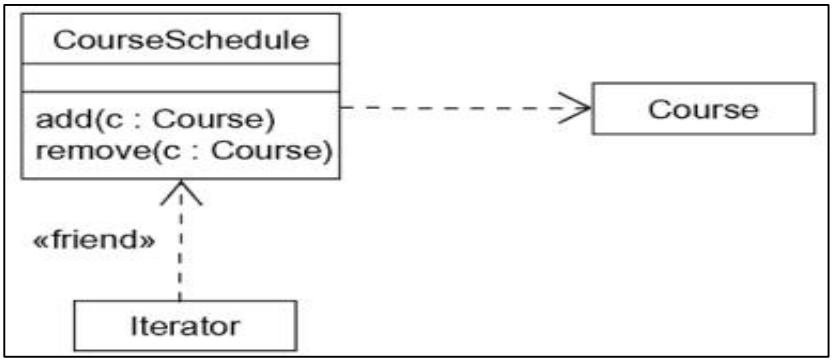
To model this using relationship

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

The following figure shows a set of classes drawn from a system that manages the assignment of students and instructors to courses in a university.

This figure shows a dependency from CourseSchedule to Course, because Course is used in both the add and remove operations of CourseSchedule.

The dependency from Iterator shows that the Iterator uses the CourseSchedule; the CourseSchedule knows nothing about the Iterator. The dependency is marked with a stereotype, which specifies that this is not a plain dependency, but, rather, it represents a friend, as in C++.

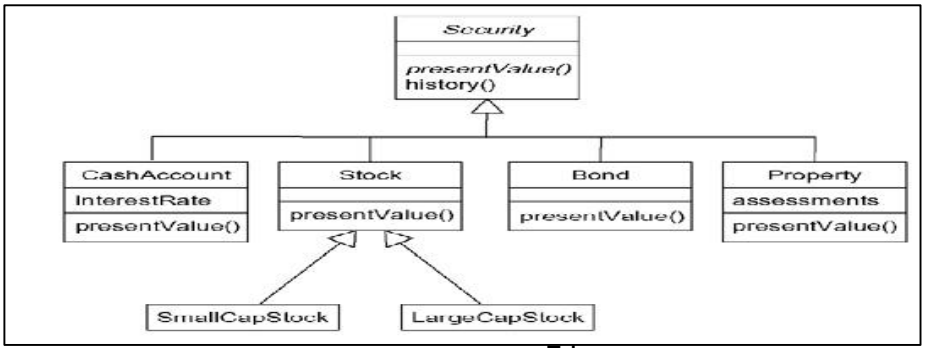


Dependency Relationships

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. If necessary, create a new class to which you can assign these
- Specify that the more-specific classes inherit from the more-general class by placing a generalization relationship that is drawn from each specialized class to its more-general parent.



Inheritance Relationships

Modeling Structural Relationships

- When you model with dependencies or generalization relationships, you are modeling classes that represent different levels of importance or different levels of abstraction
- Given a generalization relationship between two classes, the child inherits from its parent but the parent has no specific knowledge of its children.
- Dependency and generalization relationships are one-sided.
- Associations are, by default, bidirectional; you can limit their direction
- Given an association between two classes, both rely on the other in some way, and you can navigate in either direction
- An association specifies a structural path across which objects of the classes interact.

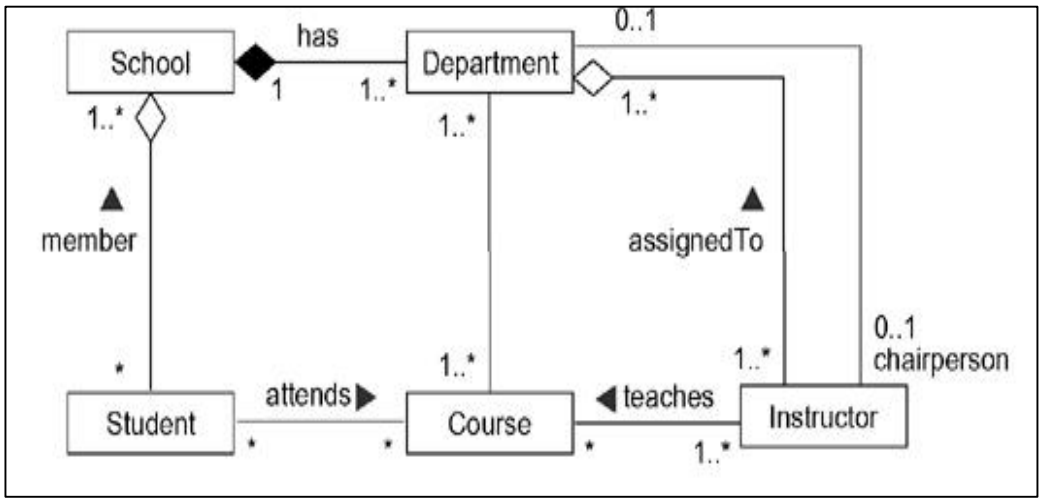
To model structural relationships

- 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).

- If one of the classes in an association is structurally or organizationally a whole compared with the classes at the other end that look like parts, mark this as an aggregation by adorning the association at the end near the whole



Structural Relationships

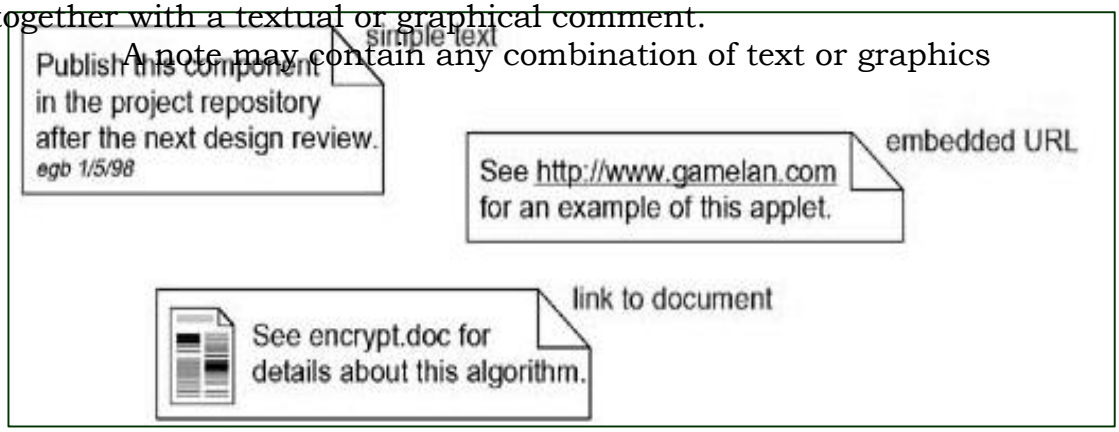
Common Mechanisms

Note

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

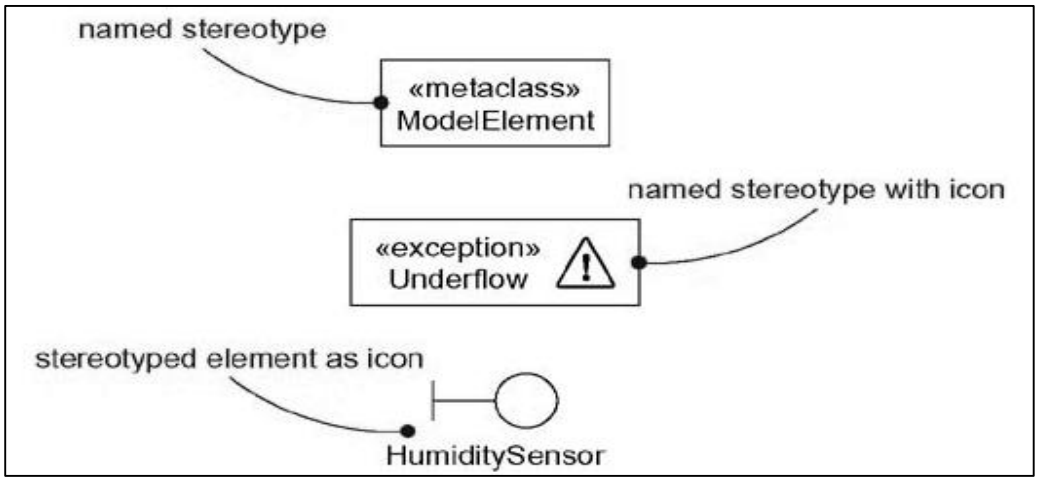


Notes

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.

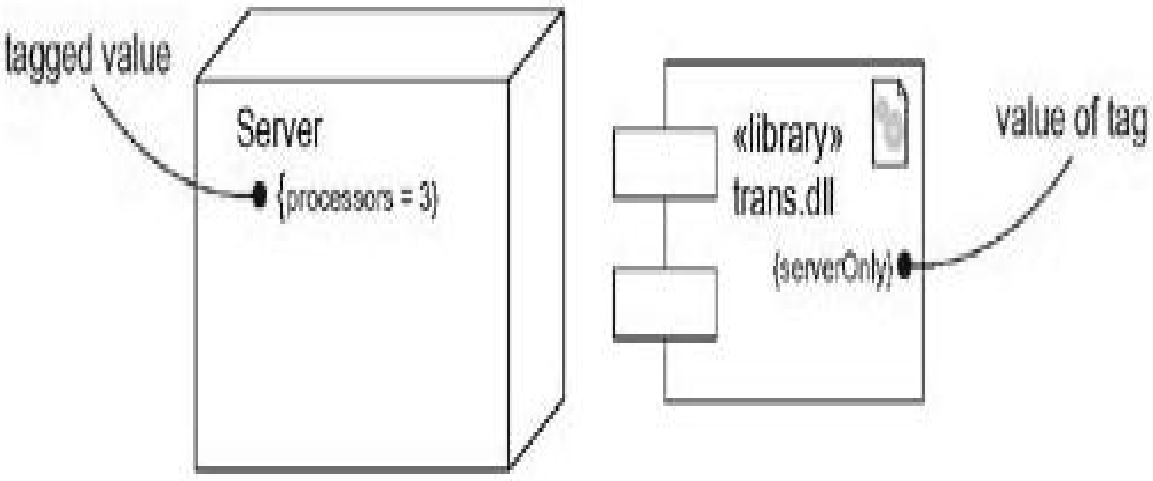
Graphically, a stereotype is rendered as a name enclosed by guillemets and placed above the name of another element



Stereotypes

Tagged Values

- Every thing 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.
- A tagged value is not the same as a class attribute. Rather, you can think of a tagged value as metadata because its value applies to the element itself, not its instances.
- A tagged value is an extension of the properties of a UML element, allowing you to create new information in that element's specification.
- Graphically, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element.
- In its simplest form, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element.
- That string includes a name (the tag), a separator (the symbol =), and a value (of the tag).

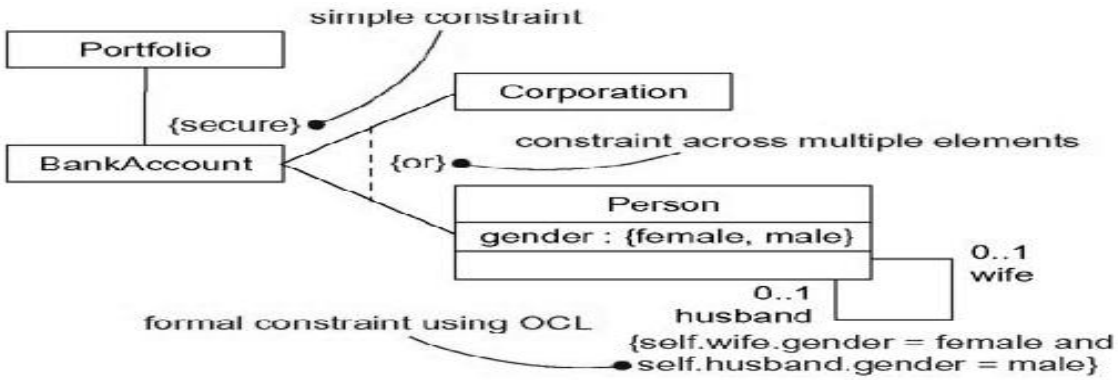


Constraints

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.



Common Modeling Techniques

Modeling Comments

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

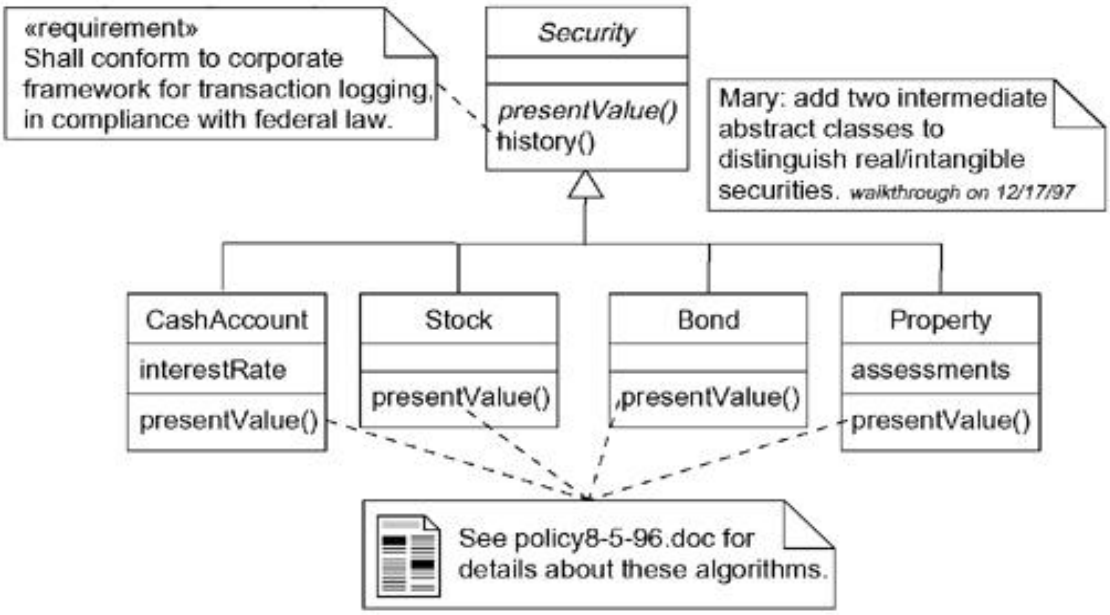
By putting these comments directly in your models, your models can become a common repository for all the disparate artifacts you'll create during development.

To model a comment,

Put your comment as text in a note and place it adjacent to the element to which it refers

Remember that you can hide or make visible the elements of your model as you see fit.

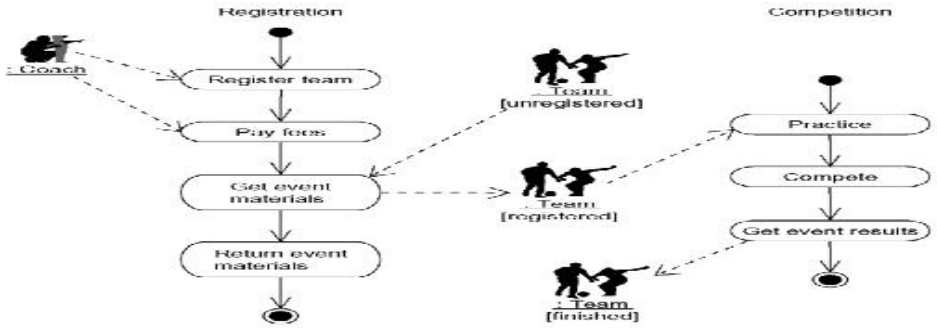
If your comment is lengthy or involves something richer than plain text, consider putting your comment in an external document and linking or embedding that document in a note attached to your model



Modeling Comments

Modeling New Building Blocks

- The UML's building blocks—classes, interfaces, collaborations, components, nodes, associations, and so on—are generic enough to address most of the things you'll want to model.
- However, if you want to extend your modeling vocabulary or give distinctive visual cues to certain kinds of abstractions that often appear in your domain, you need to use stereotypes
- To model new building blocks,
  - Make sure there's not already a way to express what you want by using basic UML
  - If you're convinced there's no other way to express these semantics, identify the primitive thing in the UML that's most like what you want to model and define a new stereotype for that thing
  - Specify the common properties and semantics that go beyond the basic element being stereotyped by defining a set of tagged values and constraints for the stereotype.
  - If you want these stereotype elements to have a distinctive visual cue, define a new icon for the stereotype



Modeling New Building Blocks

Modeling New Properties

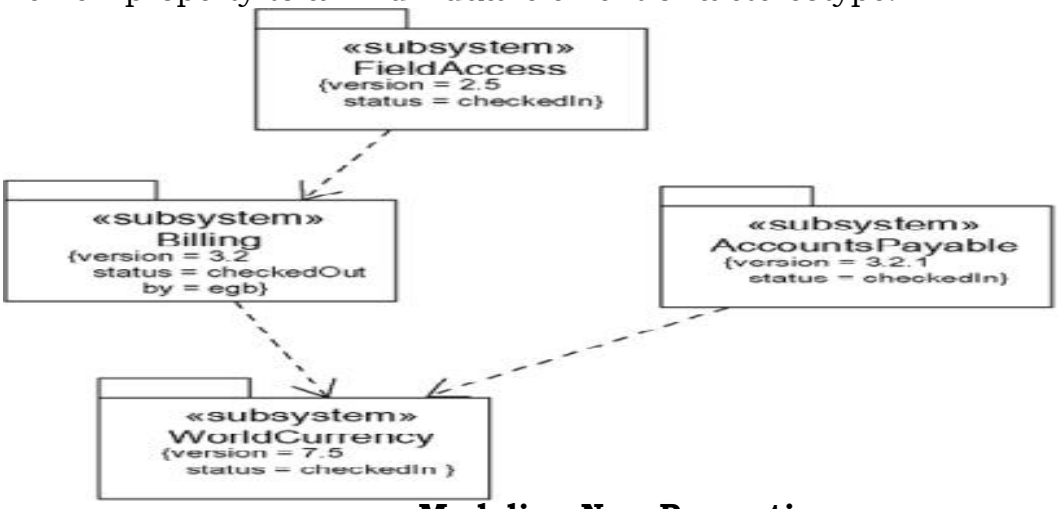
The basic properties of the UML's building blocks—attributes and operations for classes, the contents of packages, and so on—are generic enough to address most of the things you'll want to model.

However, if you want to extend the properties of these basic building blocks, you need to use tagged values.

To model new properties,

First, make sure there's not already a way to express what you want by using basic UML

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

Modeling New Semantics

When you create a model using the UML, you work within the rules the UML lays down

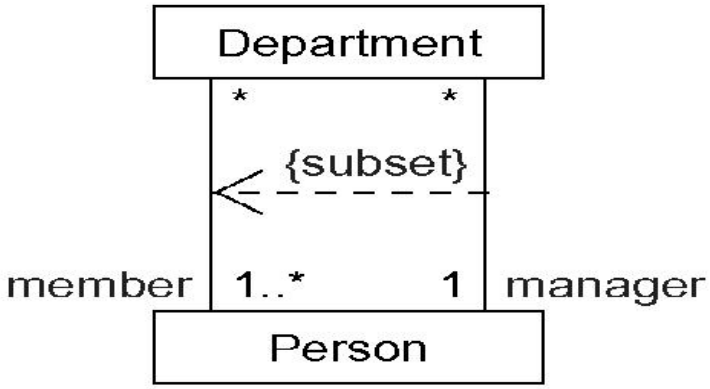
However, if you find yourself needing to express new semantics about which the UML is silent or that you need to modify the UML's rules, then you need to write a constraint.

To model new semantics,

First, make sure there's not already a way to express what you want by using basic UML

If you're convinced there's no other way to express these semantics, write your new semantics as text in a constraint and place it adjacent to the element to which it refers

If you need to specify your semantics more precisely and formally, write your new semantics using OCL.



Modeling New Semantics

Diagrams

- When you view a software system from any perspective using the UML, you use diagrams to organize the elements of interest.
- The UML defines nine kinds of diagrams, which you can mix and match to assemble each view.
- Of course, you are not limited to these nine diagrams. In the UML, these nine are defined because they represent the most common packaging of viewed elements. To fit the needs of your project or organization, you can create your own kinds of diagrams to view UML elements in different ways.
- You'll use the UML's diagrams in two basic ways:
  - to specify models from which you'll construct an executable system (forward engineering)
  - and to reconstruct models from parts of an executable system (reverse engineering).

System

- A system is a collection of subsystems organized to accomplish a purpose and described by a set of models, possibly from different viewpoints

SubSystem

- A subsystem is a grouping of elements, of which some constitute a specification of the behavior offered by the other contained elements.

Model

- A model is a semantically closed abstraction of a system, meaning that it represents a complete and self-consistent simplification of reality, created in order to better understand the system. In the context of architecture

View

- view is a projection into the organization and structure of a system's model, focused on one aspect of that system

Diagram

- A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).
- A diagram is just a graphical projection into the elements that make up a system
- Each diagram provides a view into the elements that make up the system
- Typically, you'll view the static parts of a system using one of the four following diagrams.
  Class diagram
  Object diagram
  Component diagram

Deployment diagram

You'll often use five additional diagrams to view the dynamic parts of a system.

Use case diagram
Sequence diagram
Collaboration diagram
Statechart diagram
Activity diagram

The UML defines these nine kinds of diagrams.

- Every diagram you create will most likely be one of these nine or occasionally of another kind
- Every diagram must have a name that's unique in its context so that you can refer to a specific diagram and distinguish one from another
- You can project any combination of elements in the UML in the same diagram. For example, you might show both classes and objects in the same diagram

Structural Diagrams

- The UML's four structural diagrams exist to visualize, specify, construct, and document the static aspects of a system.
- The UML's structural diagrams are roughly organized around the major groups of things you'll find when modeling a system.
  - Class diagram : Classes, interfaces, and collaborations
  - Object diagram : Objects
  - Component diagram : Components
  - Deployment diagram : Nodes

Class Diagram

- We use class diagrams to illustrate the static design view of a system.
- Class diagrams are the most common diagram found in modeling object-oriented systems.
- A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- Class diagrams that include active classes are used to address the static process view of a system.

Object Diagram

- Object diagrams address the static design view or static process view of a system just as do class diagrams, but from the perspective of real or prototypical cases.
- An object diagram shows a set of objects and their relationships.
- You use object diagrams to illustrate data structures, the static snapshots of instances of the things found in class diagrams.

Component Diagram

- We use component diagrams to illustrate the static implementation view of a system.
- A component diagram shows a set of components and their relationships.
- Component diagrams are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

Deployment Diagram

- We use deployment diagrams to illustrate the static deployment view of an architecture.

- A deployment diagram shows a set of nodes and their relationships.
- Deployment diagrams are related to component diagrams in that a node typically encloses one or more components.

Behavioral Diagrams

- The UML's five behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system.
- The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.
  - Use case diagram : Organizes the behaviors of the system
  - Sequence diagram : Focused on the time ordering of messages
  - Collaboration diagram : Focused on the structural organization of objects that send and receive messages
  - Statechart diagram : Focused on the changing state of a system driven by events
  - Activity diagram : Focused on the flow of control from activity to activity

Use Case Diagram

- A use case diagram shows a set of use cases and actors and their relationships.
- We apply use case diagrams to illustrate the static use case view of a system.
- Use case diagrams are especially important in organizing and modeling the behaviors of a system.

Sequence Diagram

- We use sequence diagrams to illustrate the dynamic view of a system.
- A sequence diagram is an interaction diagram that emphasizes the time ordering of messages.
- A sequence diagram shows a set of objects and the messages sent and received by those objects.
- The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.

Collaboration Diagram

- We use collaboration diagrams to illustrate the dynamic view of a system.
- A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.
- A collaboration diagram shows a set of objects, links among those objects, and messages sent and received by those objects.
- The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.

***** Sequence and collaboration diagrams are isomorphic, meaning that you can convert from one to the other without loss of information.

Statechart Diagram

We use statechart diagrams to illustrate the dynamic view of a system.

They are especially important in modeling the behavior of an interface, class, or collaboration.

A statechart diagram shows a state machine, consisting of states, transitions, events, and activities.

Statechart diagrams emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

Activity Diagram

We use activity diagrams to illustrate the dynamic view of a system.

Activity diagrams are especially important in modeling the function of a system.

Activity diagrams emphasize the flow of control among objects.

An activity diagram shows the flow from activity to activity within a system.

An activity shows a set of activities, the sequential or branching flow from activity to activity, and objects that act and are acted upon.

Common Modeling Techniques

Modeling Different Views of a System

- When you model a system from different views, you are in effect constructing your system simultaneously from multiple dimensions
- 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.
  - As part of your process planning, decide which of these diagrams you'll want to put under some sort of formal or semi-formal control.
  - For example, if you are modeling a simple monolithic application that runs on a single machine, you might need only the following handful of diagrams

• 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.
  - Finally, if you are modeling a complex, distributed system, you'll need to employ the full range of the UML's diagrams in order to express the architecture of your system and the technical risks to your project, as in the following.

• 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

Modeling Different Levels of Abstraction

- Not only do you need to view a system from several angles, you'll also find people involved in development who need the same view of the system but at different levels of abstraction
- Basically, there are two ways to model a system at different levels of abstraction:
    - By presenting diagrams with different levels of detail against the same model
    - 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,
    - Consider the needs of your readers, and start with a given model
    - If your reader is using the model to construct an implementation, she'll need diagrams that are at a lower level of abstraction which means that they'll need to reveal a lot of detail
    - If she is using the model to present a conceptual model to an end user, she'll need diagrams that are at a higher level of abstraction which means that they'll hide a lot of detail
    - Depending on where you land in this spectrum of low-to-high levels of abstraction, create a diagram at the right level of abstraction by hiding or revealing the following four categories of things from your model:

Building blocks and relationships:

- Hide those that are not relevant to the intent of your diagram or the needs of your reader.

Adornments:

- Reveal only the adornments of these building blocks and relationships that are essential to understanding your intent.

Flow:

- In the context of behavioral diagrams, expand only those messages or transitions that are essential to understanding your intent.

Stereotypes:

- In the context of stereotypes used to classify lists of things, such as attributes and operations, reveal only those stereotyped items that are essential to understanding your intent.

- The main advantage of this approach is that you are always modeling from a common semantic repository.

- The main disadvantage of this approach is that changes from diagrams at one level of abstraction may make obsolete diagrams at a different level of abstraction.

To model a system at different levels of abstraction by creating models at different levels of abstraction,

- Consider the needs of your readers and decide on the level of abstraction that each should view, forming a separate model for each level.
- In general, populate your models that are at a high level of abstraction with simple abstractions and your models that are at a low level of abstraction with detailed abstractions. Establish trace dependencies among the related elements of different models.
- In practice, if you follow the five views of an architecture, there are four common situations you'll encounter when modeling a system at different levels of abstraction:

Use cases and their realization:

Use cases in a use case model will trace to collaborations in a design model.

Collaborations and their realization:

Collaborations will trace to a society of classes that work together to carry out the collaboration.

Components and their design:

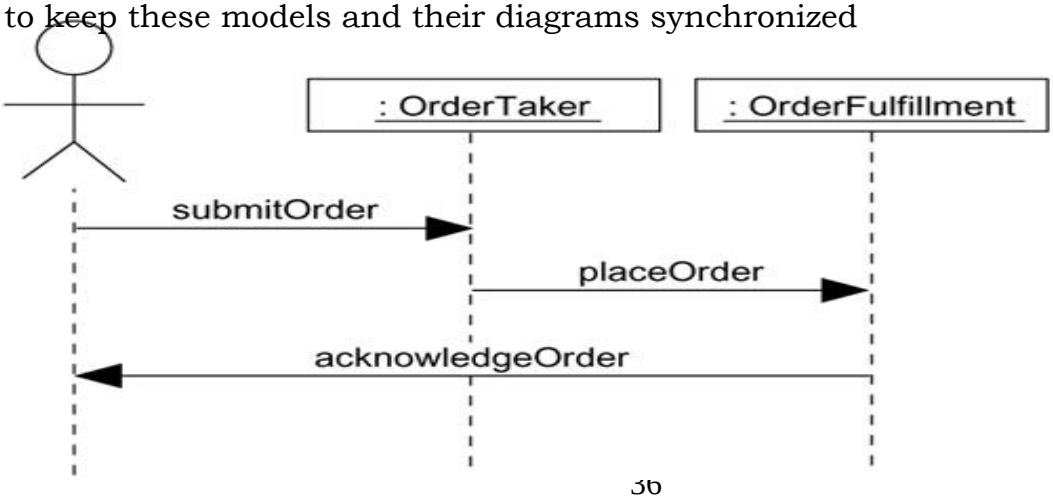
Components in an implementation model will trace to the elements in a design model.

Nodes and their components:

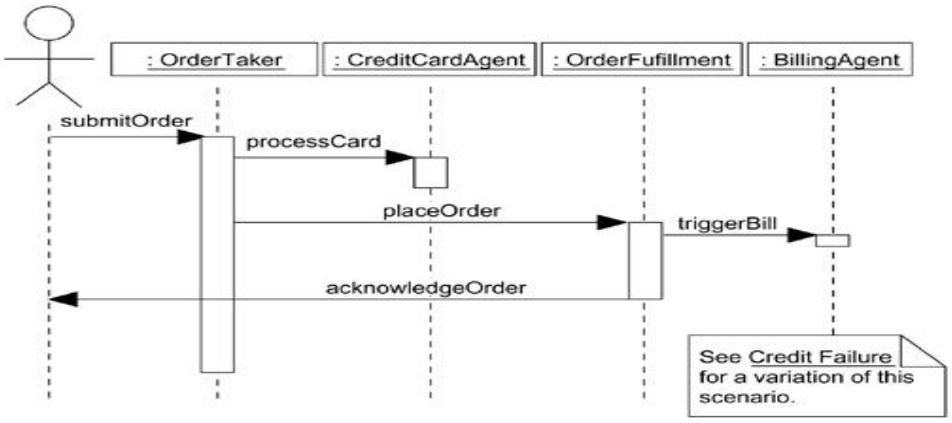
Nodes in a deployment model will trace to components in an implementation model.

The main advantage of the approach is that diagrams at different levels of abstraction remain more loosely coupled. This means that changes in one model will have less direct effect on other models.

The main disadvantage of this approach is that you must spend resources to keep these models and their diagrams synchronized



Interaction Diagram at a High Level of Abstraction



Interaction at a Low Level of Abstraction

* Both of these diagrams work against the same model, but at different levels of detail.

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.
  - If your diagram is still complex, use notes and color as visual cues to draw the reader's attention to the points you want to make.
  - If your diagram is still complex, print it in its entirety and hang it on a convenient large wall. You lose the interactivity an online version of the diagram brings, but you can step back from the diagram and study it for common patterns.

Advanced Structural Modeling

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

Dependency

- A dependency is a using relationship, specifying that a change in the specification of one thing may affect another thing that uses it, but not necessarily the reverse. Graphically, a dependency is rendered as a dashed line
- A plain, unadorned dependency relationship is sufficient for most of the using relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines a number of stereotypes that may be applied to dependency relationships.
- There are 17 such stereotypes, all of which can be organized into six groups.

- 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

bind:

bind includes a list of actual arguments that map to the formal arguments of the template.

derive

When you want to model the relationship between two attributes or two associations, one of which is concrete and the other is conceptual.

friend

When you want to model relationships such as found with C++ friend classes.

instanceOf

When you want to model the relationship between a class and an object in the same diagram, or between a class and its metaclass.

instantiate

when you want to specify which element creates objects of another.

powertype

when you want to model classes that cover other classes, such as you'll find when modeling databases

refine

when you want to model classes that are essentially the same but at different levels of abstraction.

use

when you want to explicitly mark a dependency as a using relationship

* 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

* We'll use become and copy when you want to show the role, state, or attribute value of one object at different points in time or space

* You'll use call when you want to model the calling dependencies among operations.

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

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

* We'll use send when you want to model an operation dispatching a given event to a target object.
* The send dependency in effect lets you tie independent state machines together.

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

* We'll use trace when you want to model the relationships among elements in different models

Generalization

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).

In a generalization relationship, instances of the child may be used anywhere instances of the parent apply—meaning that the child is substitutable for the parent.

A plain, unadorned generalization relationship is sufficient for most of the inheritance relationships you'll encounter. However, if you want to specify a shade of meaning,

The UML defines one stereotype and four constraints that may be applied to generalization relationships.

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

?implementation

- We'll use implementation when you want to model private inheritance, such as found in C++.

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

complete

- We'll use the complete constraint when you want to show explicitly that you've fully specified a hierarchy in the model (although no one diagram may show that hierarchy);

incomplete

- We'll use incomplete to show explicitly that you have not stated the full specification of the hierarchy in the model (although one diagram may show everything in the model).

Disjoint & overlapping

- These two constraints apply only in the context of multiple inheritance.
- We'll use disjoint and overlapping when you want to distinguish between static classification (disjoint) and dynamic classification (overlapping).

Association

An association is a structural relationship, specifying that objects of one thing are connected to objects of another.

We use associations when you want to show structural relationships.

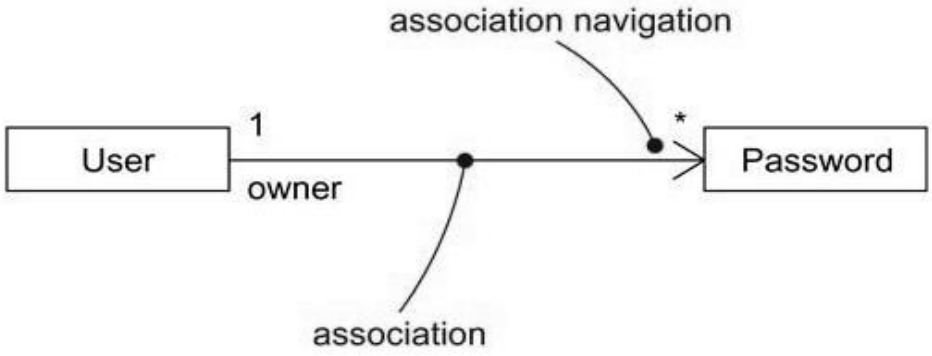
There are four basic adornments that apply to an association: a name, the role at each end of the association, the multiplicity at each end of the association, and aggregation.

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

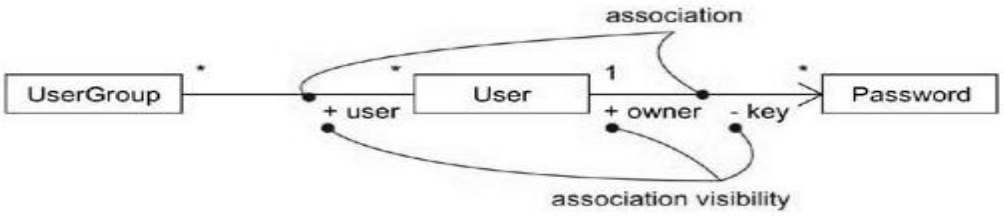
- unadorned association between two classes, such as Book and Library, it's possible to navigate from objects of one kind to objects of the other kind. Unless otherwise specified, navigation across an association is bidirectional.
- However, there are some circumstances in which you'll want to limit navigation to just one direction.



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.
- However, there are circumstances in which you'll want to limit the visibility across that association relative to objects outside the association.
- In the UML, you can specify three levels of visibility for an association end, just as you can for a class's features by appending a visibility symbol to a role name the visibility of a role is public.
- Private visibility indicates that objects at that end are not accessible to any objects outside the association.
- Protected visibility indicates that objects at that end are not accessible to any objects outside the association, except for children of the other end.



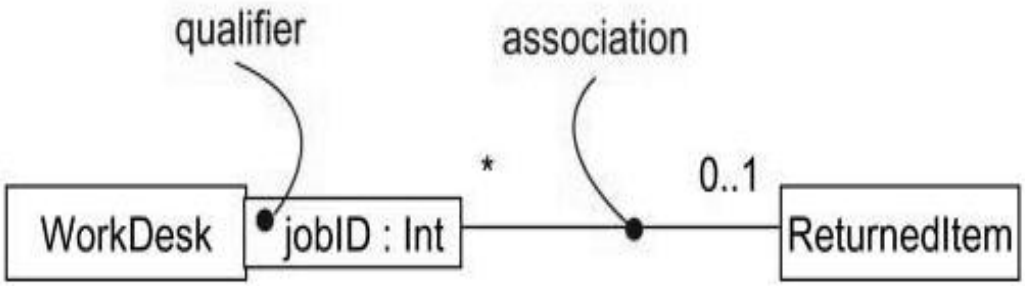
Visibility

Qualification

In the context of an association, one of the most common modeling idioms you'll encounter is the problem of lookup. Given an object at one end of an association, how do you identify an object or set of objects at the other end?

In the UML, you'd model this idiom using a qualifier, which is an association attribute whose values partition the set of objects related to an object across an association.

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



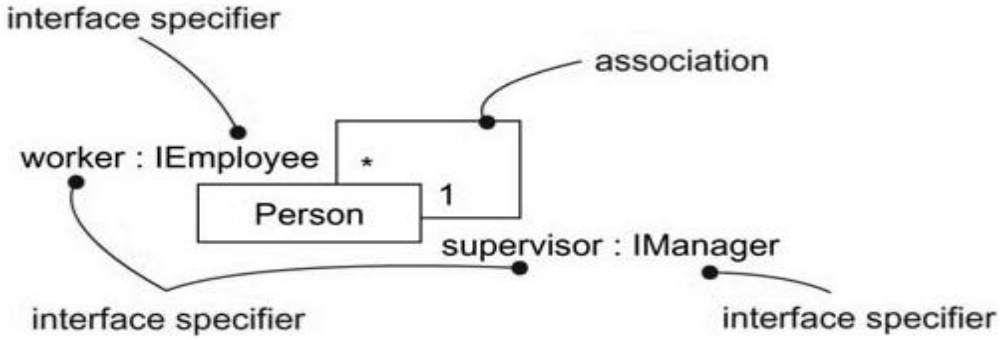
Qualification

Interface Specifier

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

Collectively, the interfaces realized by a class represent a complete specification of the behavior of that class.

However, in the context of an association with another target class, a source class may choose to present only part of its face to the world



a Person class may realize many interfaces: IManager, IEmployee, IOfficer, and so on

you can model the relationship between a supervisor and her workers with a one-to-many

association, explicitly labeling the roles of this association as supervisor and worker

- In the context of this association, a Person in the role of supervisor presents only the IManager face to the worker; a Person in the role of worker presents only the IEmployee face to the supervisor. As the figure shows, you can explicitly show the type of role using the syntax rolename : iname, where iname is some interface of the other classifier.

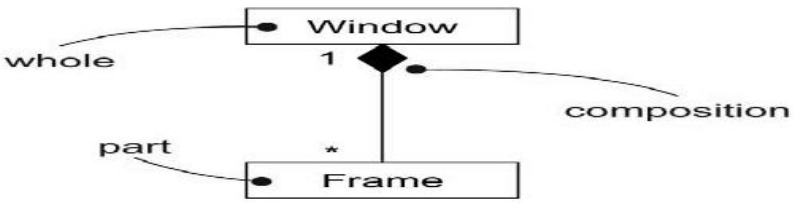
Composition

* 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.
* Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and

die with it. Such parts can also be explicitly removed before the death of the composite.

* This means that, in a composite aggregation, an object may be a part of only one composite at a time
* In addition, in a composite aggregation, the whole is responsible for the disposition of its parts, which

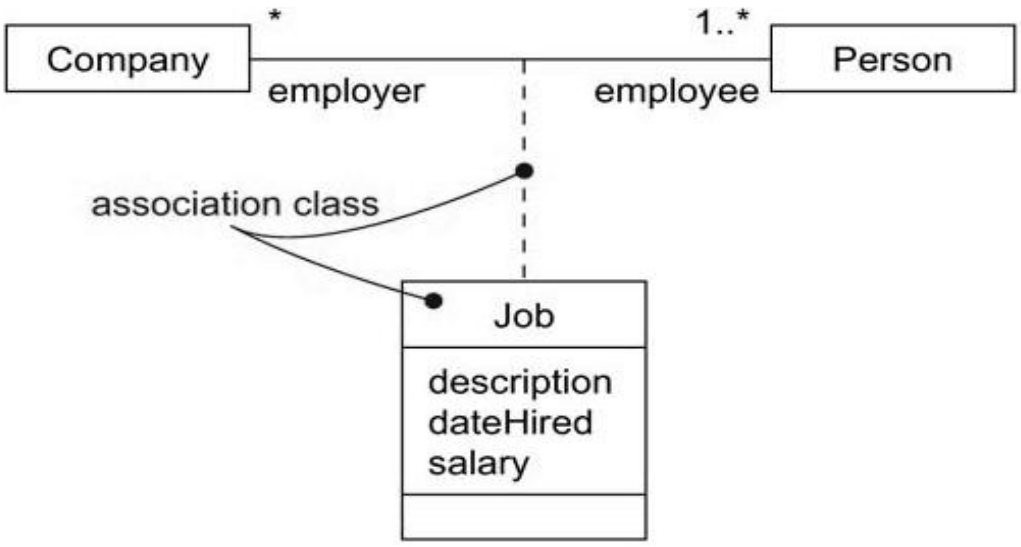
means that the composite must manage the creation and destruction of its parts



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.
* We render an association class as a class symbol attached by a dashed line to an association



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

implicit

* if you have an association between two base classes, you can specify that same association between two children of those base classes

* you can specify that the objects at one end of an association (with a multiplicity greater than one) are ordered or unordered.

ordered

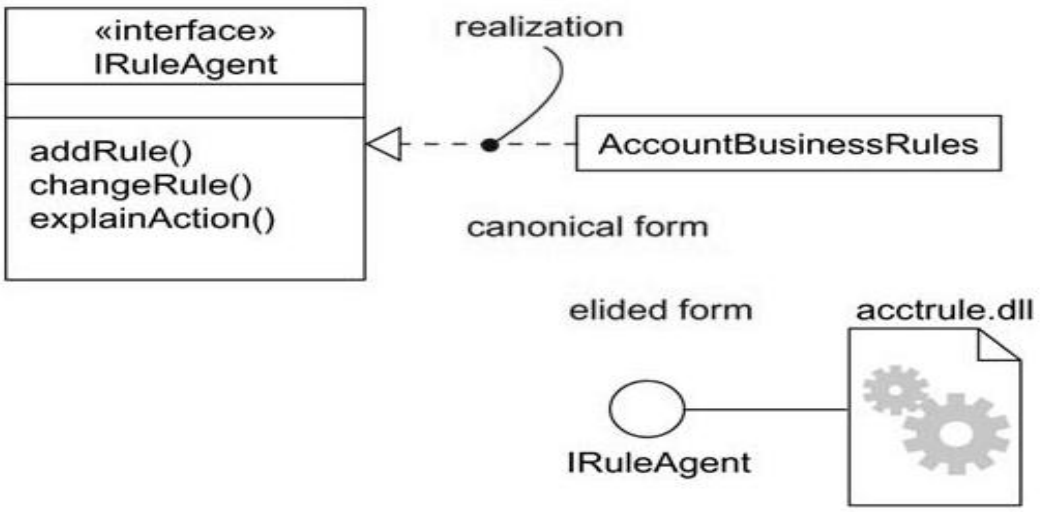
* For example, in a User/Password association, the Passwords associated with the User might be kept in a least-recently used order, and would be marked as ordered.

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

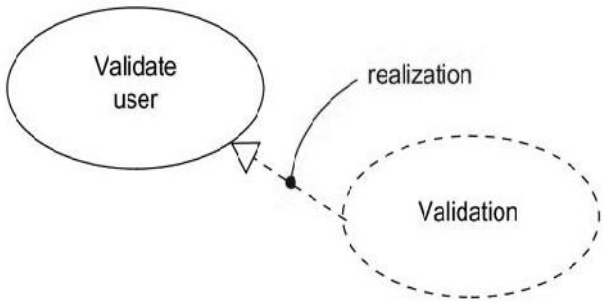
1	**xor**	Specifies that, over a set of associations, exactly one is manfest for each associated object

Realization

1. Realization is sufficiently different from dependency, generalization, and association relationships that it is treated as a separate kind of relationship.
2. A realizationis a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.
3. Graphically, a realization is rendered as a dashed directed line with a large open arrowhead pointing to the classifier that specifies the contract.
4. You'll use realization in two circumstances: in the context of interfaces and in the context of collaborations
5. Most of the time, you'll use realization to specify the relationship between an interface and the class or component that provides an operation or service for it
6. You'll also use realization to specify the relationship between a use case and the collaboration that realizes that use case



Realization of an Interface



Realization of a Use Case

Common Modeling Techniques

Modeling Webs of Relationships

1. When you model the vocabulary of a complex system, you may encounter dozens, if not hundreds or thousands, of classes, interfaces, components, nodes, and use cases.
2. Establishing a crisp boundary around each of these abstractions is hard
3. This requires you to form a balanced distribution of responsibilities in the system as a whole, with individual abstractions that are tightly cohesive and with relationships that are expressive, yet loosely coupled
4. 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.
    - Only after completing the preceding steps should you look for dependencies; they generally represent more-subtle forms of semantic connection.
    - For each kind of relationship, start with its basic form and apply advanced features only as absolutely necessary to express your intent.
    - Remember that it is both undesirable and unnecessary to model all relationships among a set of abstractions in a single diagram or view. Rather, build up your system's relationships by considering different views on the system. Highlight interesting sets of relationships in individual diagrams.

Interfaces, type and roles

Interface

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

type

- A type is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object.

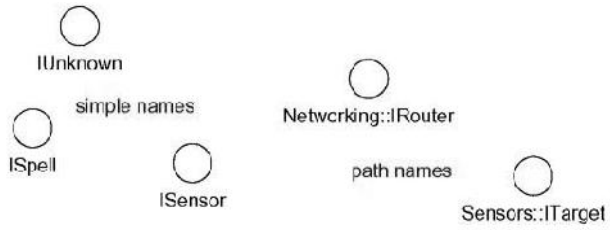
role

- A role is the behavior of an entity participating in a particular context.

an interface may be rendered as a stereotyped class in order to expose its operations and other properties.

Names

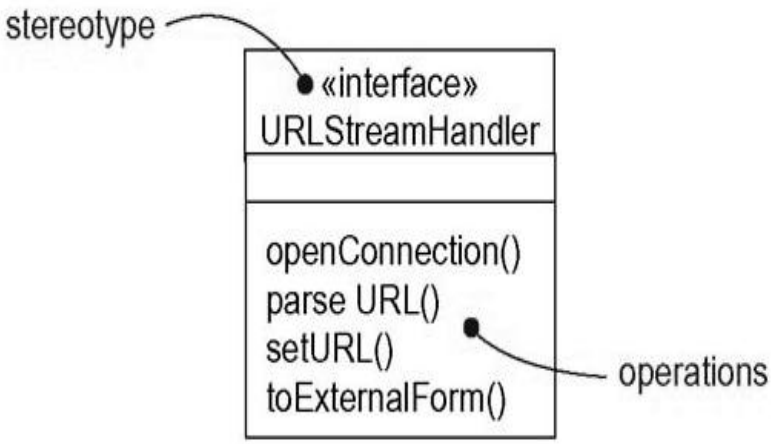
- Every interface must have a name that distinguishes it from other interfaces.
- A name is a textual string. That name alone is known as a simple name;
- A path name is the interface name prefixed by the name of the package



Simple and Path Names

Operations

- An interface is a named collection of operations used to specify a service of a class or of a component.
- Unlike classes or types, interfaces do not specify any structure (so they may not include any attributes), nor do they specify any implementation
- These operations may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.
- you can render an interface as a stereotyped class, listing its operations in the appropriate compartment. Operations may be drawn showing only their name, or they may be augmented to show their full signature and other properties



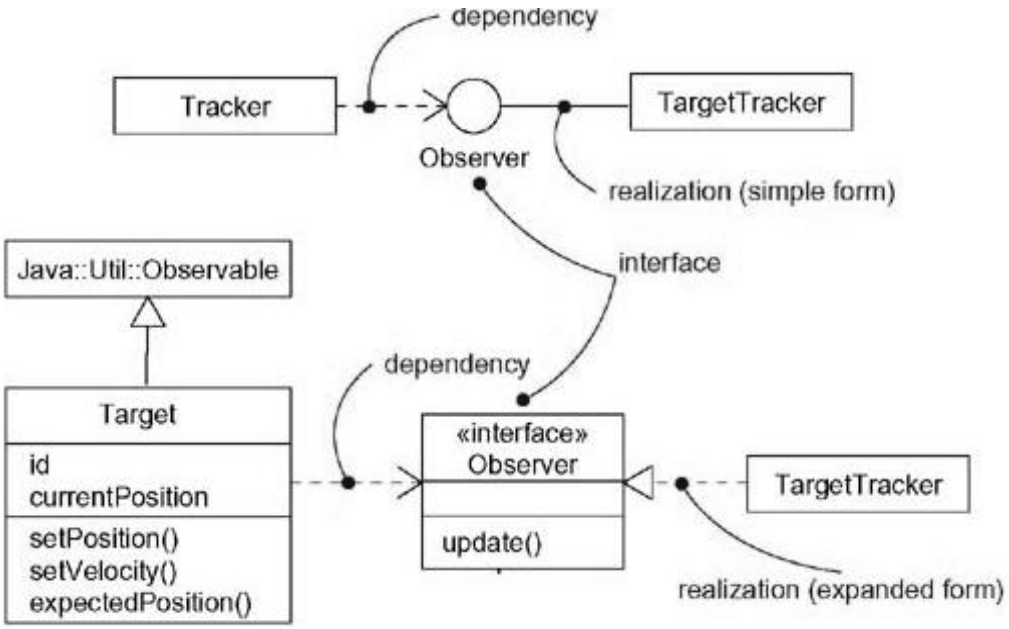
Operations

Relationships

- Like a class, an interface may participate in generalization, association, and dependency relationships. In addition, an interface may participate in realization relationships.
- An interface specifies a contract for a class or a component without dictating its implementation. A class or component may realize many interfaces
- We can show that an element realizes an interface in two ways.

o First, you can use the simple form in which the interface and its realization relationship are
rendered as a lollipop sticking off to one side of a class or component.

o Second, you can use the expanded form in which you render an interface as a stereotyped class,
which allows you to visualize its operations and other properties, and then draw a realization
relationship from the classifier or component to the interface.



Realizations

Understanding an Interface

- In the UML, you can supply much more information to an interface in order to make it understandable and approachable.

- First, you may attach pre- and postconditions to each operation and invariants to the class or component as a whole. By doing this, a client who needs to use an interface will be able to understand what the interface does and how to use it, without having to dive into an implementation.

- We can attach a state machine to the interface. You can use this state machine to specify the legal partial ordering of an interface's operations.

- We can attach collaborations to the interface. You can use collaborations to specify the expected behavior of the interface through a series of interaction diagrams.

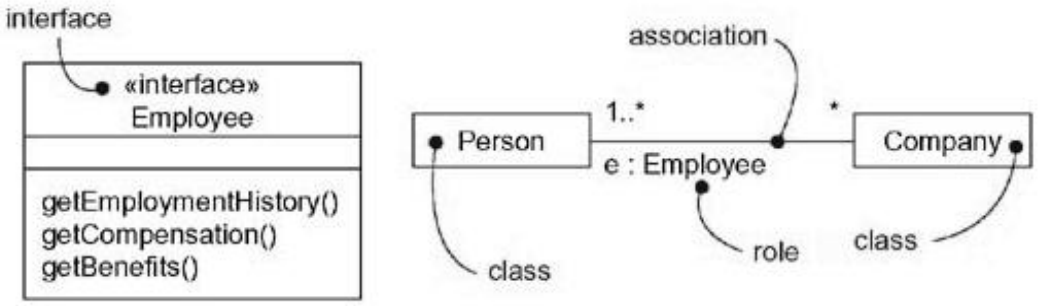
Types and Roles

A role names a behavior of an entity participating in a particular context. Stated another way, a role is the face that an abstraction presents to the world.

For example, consider an instance of the class Person. Depending on the context, that Person instance may play the role of Mother, Comforter, PayerOfBills, Employee, Customer, Manager, Pilot, Singer, and so on. When an object plays a particular role, it presents a face to the world, and clients that interact with it expect a certain behavior depending on the role that it plays at the time.

an instance of Person in the role of Manager would present a different set of properties than if the instance were playing the role of Mother.

In the UML, you can specify a role an abstraction presents to another abstraction by adorning the name of an association end with a specific interface.



Roles

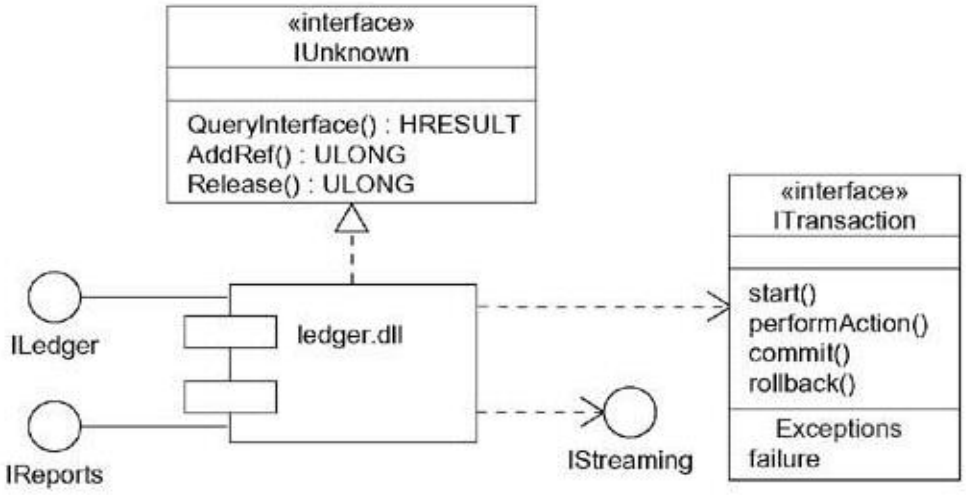
- A class diagram like this one is useful for modeling the static binding of an abstraction to its interface. You can model the dynamic binding of an abstraction to its interface by using the become stereotype in an interaction diagram, showing an object changing from one role to another.
- If you want to formally model the semantics of an abstraction and its conformance to a specific interface, you'll want to use the defined stereotype type
- Type is a stereotype of class, and you use it to specify a domain of objects, together with the operations (but not the methods) applicable to the objects of that type. The concept of type is closely related to that of interface, except that a type's definition may include attributes while an interface may not.

Common Modeling Techniques

➢ Modeling the Seams in a Systemeling the Seams in a System

- The most common purpose for which you'll use interfaces is to model the seams in a system composed of software components, such as COM+ or Java Beans.

- Identifying the seams in a system involves identifying clear lines of demarcation in your architecture. On either side of those lines, you'll find components that may change independently, without affecting the components on the other side,



Modeling the Seams in a System

- The above Figure shows the seams surrounding a component (the library ledger.dll) drawn from a financial system. This component realizes three interfaces: IUnknown, ILedger, and IReports. In this diagram, IUnknown is shown in its expanded form; the other two are shown in their simple form, as lollipops. These three interfaces are realized by ledger.dll and are exported to other components for them to build on.
- As this diagram also shows, ledger.dll imports two interfaces, IStreaming and ITransaction, the latter of which is shown in its expanded form. These two interfaces are required by the ledger.dll component for its proper operation. Therefore, in a running system, you must supply components that realize these two interfaces.
- By identifying interfaces such as ITransaction, you've effectively decoupled the components on either side of the interface, permitting you to employ any component that conforms to that interface.

➢ Modeling Static and Dynamic Typeseling Static and Dynamic Types

- Most object-oriented programming languages are statically typed, which means that the type of an object is bound at the time the object is created.
- Even so, that object will likely play different roles over time.
- Modeling the static nature of an object can be visualized in a class diagram. However, when you are modeling things like business objects, which naturally change their roles throughout a workflow,
- To model a dynamic type

- Specify the different possible types of that object by rendering each type as a class stereotyped as type (if the abstraction requires structure and behavior) or as interface (if the abstraction requires only behavior).

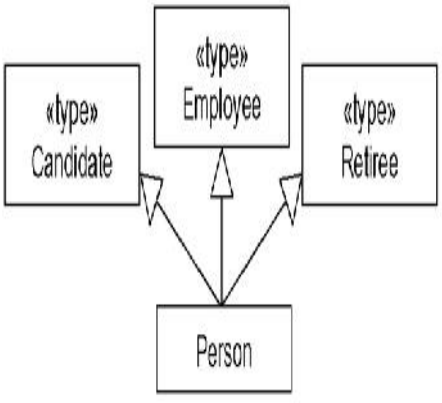
- Model all the roles the of the object may take on at any point in time. You can do so in two ways:

  1.) First, in a class diagram, explicitly type each role that the class plays in its association with Other classes. Doing this specifies the face instances of that class put on in the context of the associated object.

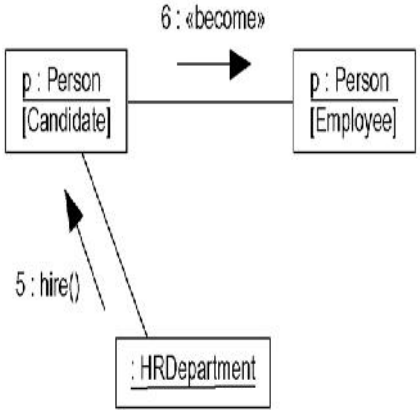
  2.) Second, also in a class diagram, specify the class-to-type relationships using generalization.

- In an interaction diagram, properly render each instance of the dynamically typed class. Display the role of the instance in brackets below the object's name.

- To show the change in role of an object, render the object once for each role it plays in the interaction, and connect these objects with a message stereotyped as become.



Modeling Static Types



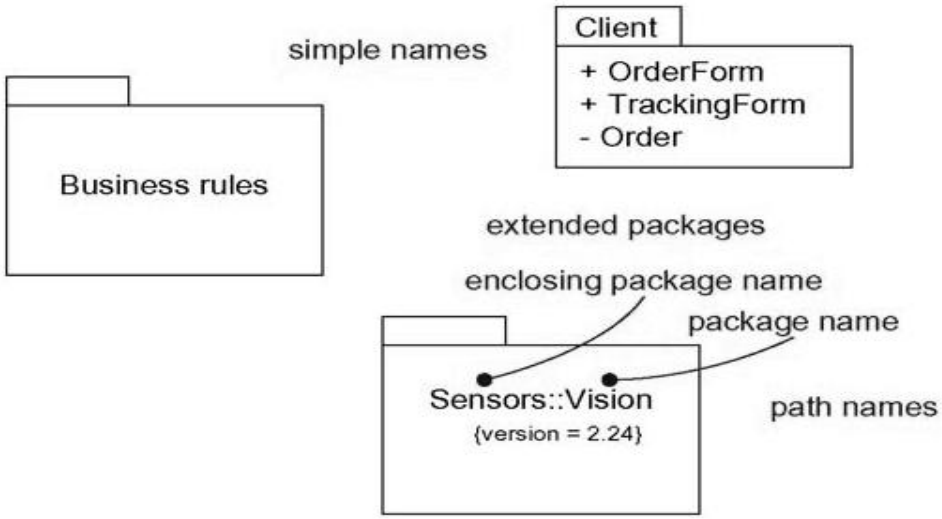
Modeling Dynamic Types

Package

"A package is a general-purpose mechanism for organizing elements into groups." Graphically, a package is rendered as a tabbed folder.

Names

- Every package must have a name that distinguishes it from other packages. A name is a textual string.
- That name alone is known as a simple name; a path name is the package name prefixed by the name of the package in which that package lives
- We may draw packages adorned with tagged values or with additional compartments to expose their details.



Simple and Extended Package

Owned Elements

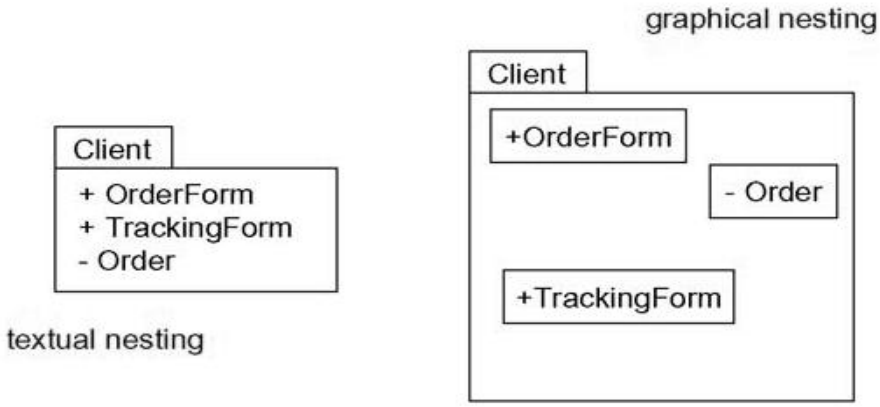
A package may own other elements, including classes, interfaces, components, nodes, collaborations, use cases, diagrams, and even other packages.

Owning is a composite relationship, which means that the element is declared in the package. If the package is destroyed, the element is destroyed. Every element is uniquely owned by exactly one package.

Elements of different kinds may have the same name within a package. Thus, you can have a class named Timer, as well as a component named Timer, within the same package.

Packages may own other packages. This means that it's possible to decompose your models hierarchically.

We can explicitly show the contents of a package either textually or graphically.



Owned Elements

Visibility

You can control the visibility of the elements owned by a package just as you can control the visibility of the attributes and operations owned by a class.

Typically, an element owned by a package is public, which means that it is visible to the contents of any package that imports the element's enclosing package.

Conversely, protected elements can only be seen by children, and private elements cannot be seen outside the package in which they are declared.

We specify the visibility of an element owned by a package by prefixing the element's name with an appropriate visibility symbol.

Importing and Exporting

In the UML, you model an import relationship as a dependency adorned with the stereotype import

Actually, two stereotypes apply here—import and access— and both specify that the source package has access to the contents of the target.

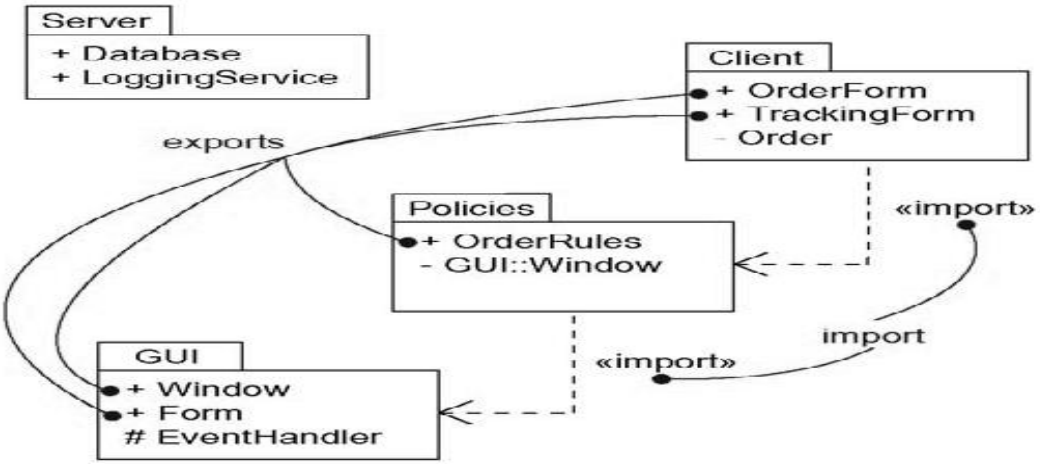
Import adds the contents of the target to the source's namespace

Access does not add the contents of the target

The public parts of a package are called its exports.

The parts that one package exports are visible only to the contents of those packages that explicitly import the package.

Import and access dependencies are not transitive



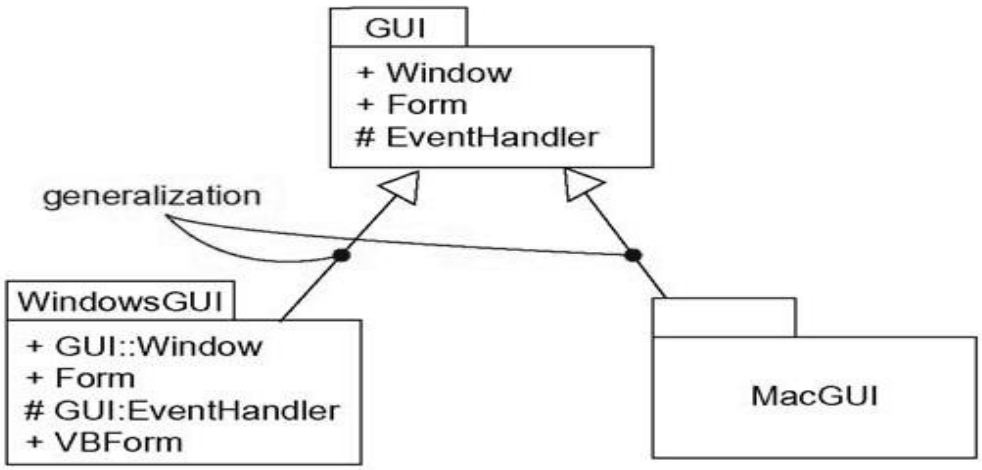
Importing and Exporting

Generalization

There are two kinds of relationships you can have between packages: import and access dependencies used to import into one package elements exported from another and generalizations, used to specify families of packages

Generalization among packages is very much like generalization among classes

Packages involved in generalization relationships follow the same principle of substitutability as do classes. A specialized package (such as WindowsGUI) can be used anywhere a more general package (such as GUI) is used



Generalization Among Packages

Standard Elements

- All of the UML's extensibility mechanisms apply to packages. Most often, you'll use tagged values to add new package properties (such as specifying the author of a package) and stereotypes to specify new kinds of packages (such as packages that encapsulate operating system services).
- The UML defines five standard stereotypes that apply to packages

1. facade	Specifies a package that is only a view on some other package
2. framework	Specifies a package consisting mainly of patterns
3. stub	Specifies a package that serves as a proxy for the public contents of another package
4. subsystem	Specifies a package representing an independent part of the entire system being modeled
5. system	Specifies a package representing the entire system being modeled

The UML does not specify icons for any of these stereotypes

Common Modeling Techniques

Modeling Groups of Elements

The most common purpose for which you'll use packages is to organize modeling elements into groups that you can name and manipulate as a set.

There is one important distinction between classes and packages:

Packages have no identity (meaning that you can't have instances of packages, so they are invisible in the running system);

classes do have identity (classes have instances, which are elements of a running system).

To model groups of elements,

Scan the modeling elements in a particular architectural view and look for clumps defined by elements that are conceptually or semantically close to one another.

Surround each of these clumps in a package.

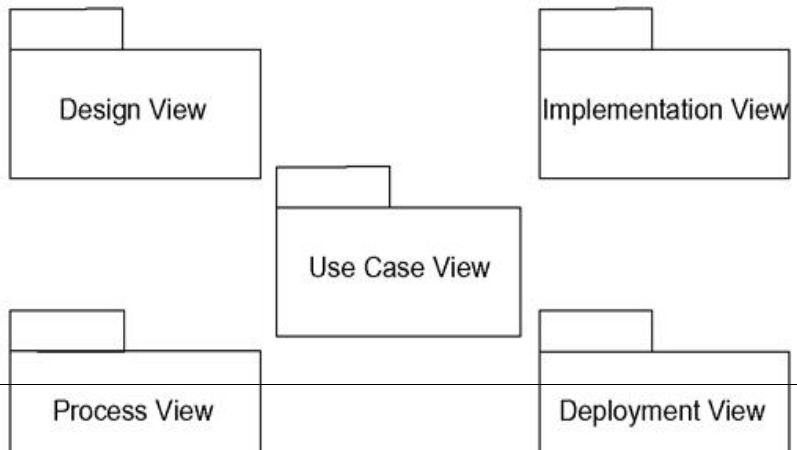
For each package, distinguish which elements should be accessible outside the package. Mark them public, and all others protected or private. When in doubt, hide the element.

Explicitly connect packages that build on others via import dependencies

In the case of families of packages, connect specialized packages to their more general part via generalizations

Modeling Architectural Views

- We can use packages to model the views of an architecture.
- Remember that a view is a projection into the organization and structure of a system, focused on a particular aspect of that system.
- This definition has two implications. First, you can decompose a system into almost orthogonal packages, each of which addresses a set of architecturally significant decisions.(design view, a process view, an implementation view, a deployment view, and a use case view)
- Second, these packages own all the abstractions germane to that view.(Implementation view)
- To model architectural views,
  - Identify the set of architectural views that are significant in the context of your problem. In practice, this typically includes a design view, a process view, an implementation view, a deployment view, and a use case view.
  - Place the elements (and diagrams) that are necessary and sufficient to visualize, specify, construct, and document the semantics of each view into the appropriate package.
  - As necessary, further group these elements into their own packages.
  - There will typically be dependencies across the elements in different views. So, in general, let each view at the top of a system be open to all others at that level.



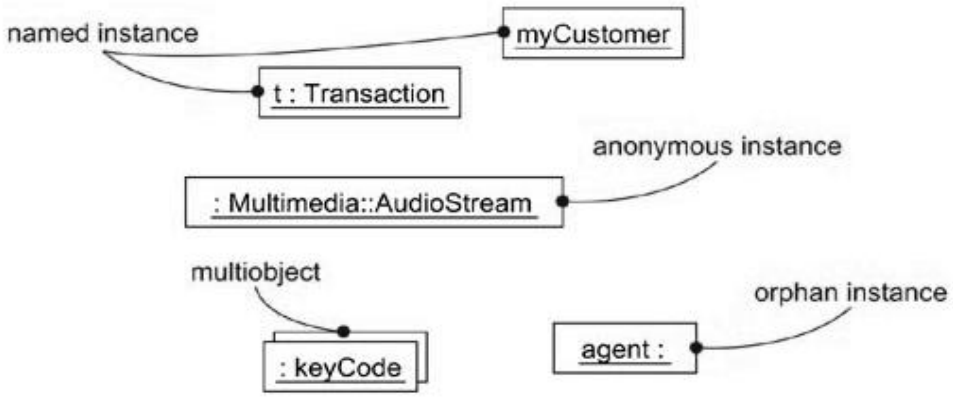
Modeling Architectural Views

Instances

- An instance is a concrete manifestation of an abstraction to which a set of operations can be applied and which has a state that stores the effects of the operations.
- Graphically, an instance is rendered by underlining its name.

Abstractions and Instances

- Most instances you'll model with the UML will be instances of classes although you can have instances of other things, such as components, nodes, use cases, and associations
- In the UML, an instance is easily distinguishable from an abstraction. To indicate an instance, you underline its name.
- We can use the UML to model these physical instances, but you can also model things that are not so concrete.



Named, Anonymous, Multiple, and Orphan Instances

Names

- Every instance must have a name that distinguishes it from other instances within its context.
- Typically, an object lives within the context of an operation, a component, or a node.
- A name is a textual string. That name alone is known as a simple name. or it may be a path name

Operations

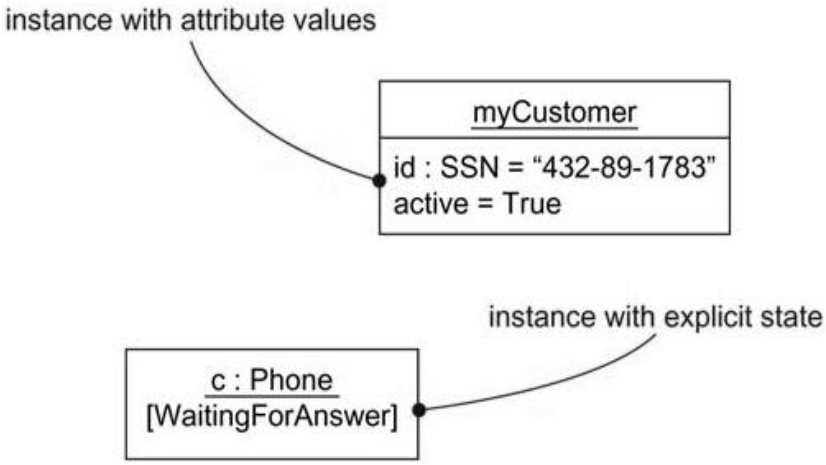
- The operations you can perform on an object are declared in the object's abstraction
- For example, if the class Transaction defines the operation commit, then given the instance t : Transaction, you can write expressions such as t.commit()

State

An object also has state. An object's state is therefore dynamic. So when you visualize its state, you are really specifying the value of its state at a given moment in time and space.

It's possible to show the changing state of an object by showing it multiple times in the same interaction diagram, but with each occurrence representing a different state.

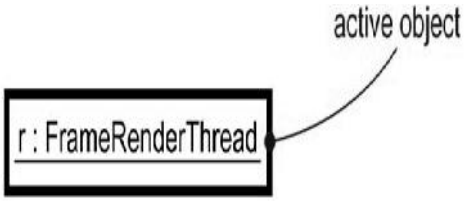
When you operate on an object, you typically change its state;
when you query an object, you don't change its state



Other Features

Processes and threads are an important element of a system's process view, so the UML provides a visual cue to distinguish elements that are active from those that are passive.

You can declare active classes that reify a process or thread, and in turn you can distinguish an instance of an active class

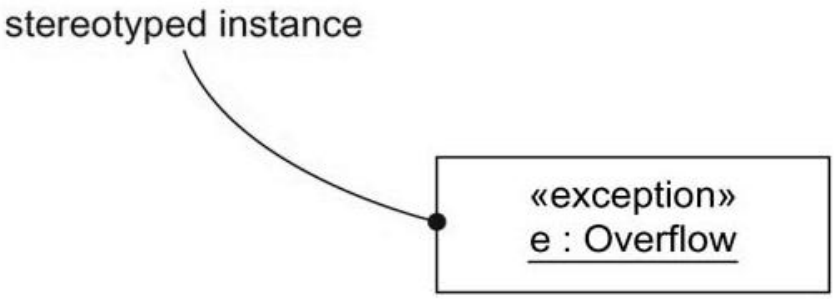


Active Objects

- There are two other elements in the UML that may have instances
- The first is a link. A link is a semantic connection among objects. An instance of an association is therefore a link. A link is rendered as a line
- The second is a class-scoped attribute and operation. A class-scoped feature is in effect an object in the class that is shared by all instances of the class.

Standard Elements

- All of the UML's extensibility mechanisms apply to objects. Usually, however, you don't stereotype an instance directly, nor do you give it its own tagged values. Instead, an object's stereotype and tagged values derive from the stereotype and tagged values of its associated abstraction.



Stereotyped Objects

The UML defines two standard stereotypes that apply to the dependency relationships among objects and among classes:

1. instanceOf	Specifies that the client object is an instance of the supplier classifier
2. instantiate	Specifies that the client class creates instances of the supplier class

There are also two stereotypes related to objects that apply to messages and transitions:

1. become	Specifies that the client is the same object as the supplier, but at a later time and with possibly different values, state, or roles
2. copy	Specifies that the client object is an exact but independent copy of the supplier

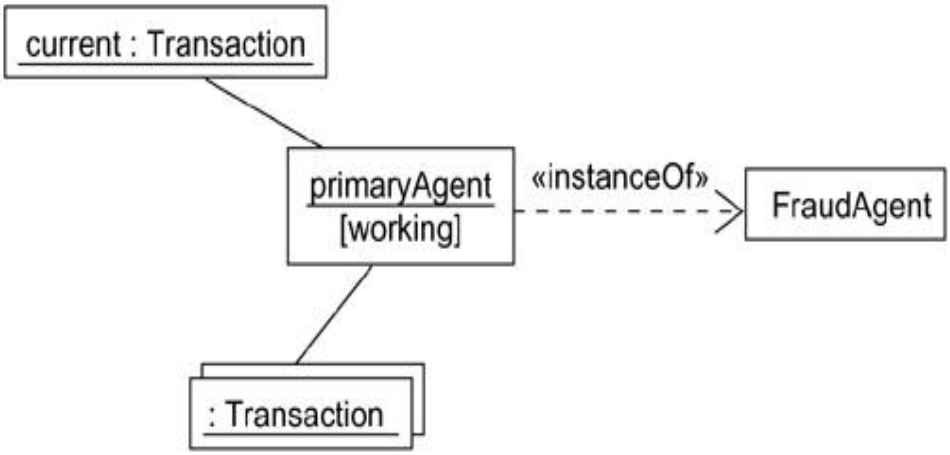
The UML defines a standard constraint that applies to objects:

transient	Specifies that an instance of the role is created during execution of the enclosing interaction but is destroyed before completion of execution

Common Modeling Techniques

Modeling Concrete Instances

- When you model concrete instances, you are in effect visualizing things that live in the real world
- One of the things for which you'll use objects is to model concrete instances that exist in the real world
- To model concrete instances,
  - Identify those instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.
  - Render these objects in the UML as instances. Where possible, give each object a name. If there is no meaningful name for the object, render it as an anonymous object.
  - Expose the stereotype, tagged values, and attributes (with their values) of each instance necessary and sufficient to model your problem.
  - Render these instances and their relationships in an object diagram or other diagram appropriate to the kind of the instance.



Modeling Concrete Instances

Modeling Prototypical Instances

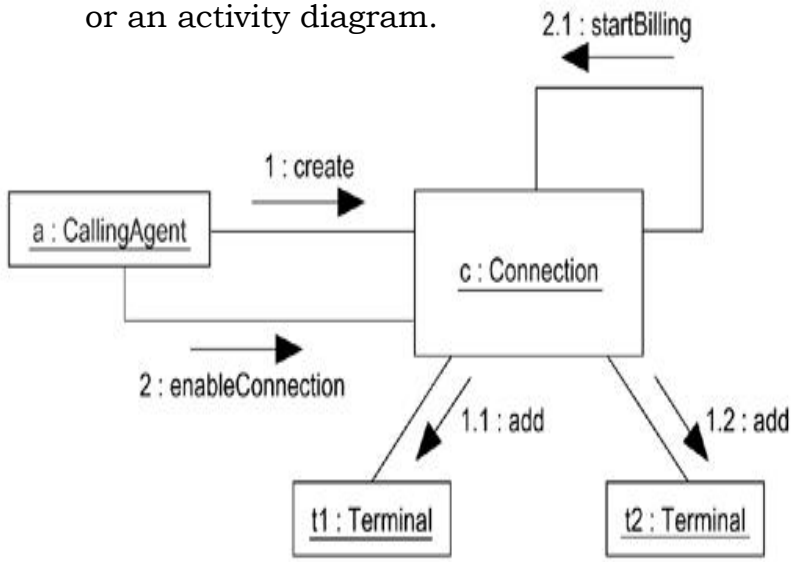
- Perhaps the most important thing for which you'll use instances is to model the dynamic interactions among objects. When you model such interactions, you are generally not modeling concrete instances that exist in the real world.
- These are prototypical objects and, therefore, are roles to which concrete instances conform.
- Concrete objects appear in static places, such as object diagrams, component diagrams, and deployment diagrams.
- Prototypical objects appear in such places as interaction diagrams and activity diagrams.
- To model prototypical instances,

  - Identify those prototypical instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.

  - Render these objects in the UML as instances. Where possible, give each object a name. If there is no meaningful name for the object, render it as an anonymous object.

  - Expose the properties of each instance necessary and sufficient to model your problem.

  - Render these instances and their relationships in an interaction diagram or an activity diagram.



Classes and Object Diagrams

Class Diagrams

- A class diagram shows a set of classes, interfaces, and collaborations and their relationships.
- Graphically, a class diagram is a collection of vertices and arcs.

Common Properties

Contents

- Class diagrams commonly contain the following things:
  - Classes
  - Interfaces
  - Collaborations
  - Dependency, generalization, and association relationships

- Like all other diagrams, class diagrams may contain notes and constraints
- Class diagrams may also contain packages or subsystems

Note: Component diagrams and deployment diagrams are similar to class diagrams, except that instead of containing classes, they contain components and nodes

Common Uses

- You use class diagrams to model the static design view of a system. This view primarily supports the functional requirements of a system
- We'll typically use class diagrams in one of three ways:

1. To model the vocabulary of a system
2. To model simple collaborations
3. To model a logical database schema

Modeling the vocabulary of a system

- Modeling the vocabulary of a system involves making a decision about which abstractions are a part of the system under consideration and which fall outside its boundaries

Modeling simple collaborations

- A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements.

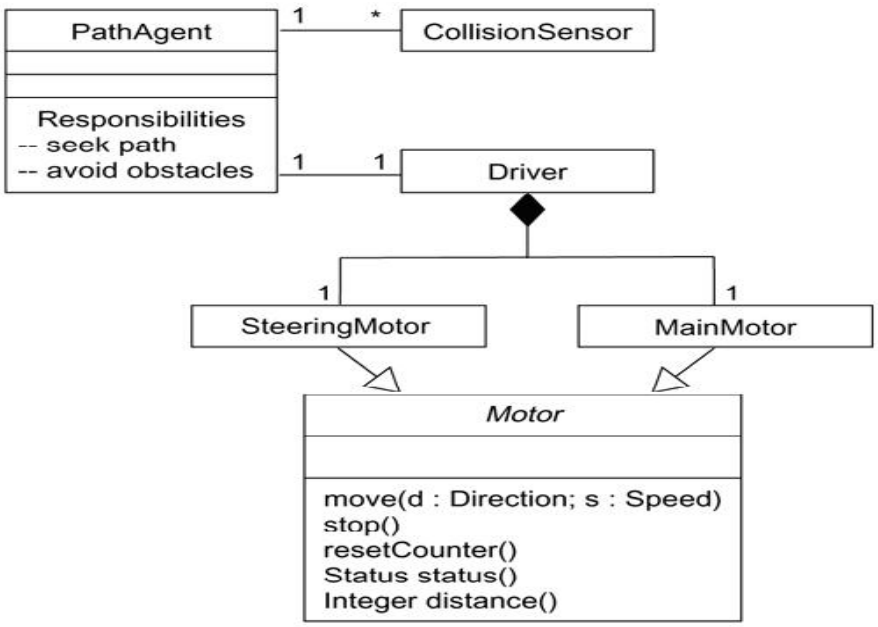
Modeling logical database schema

- We can model schemas for these databases using class diagrams.

Common Modeling Techniques

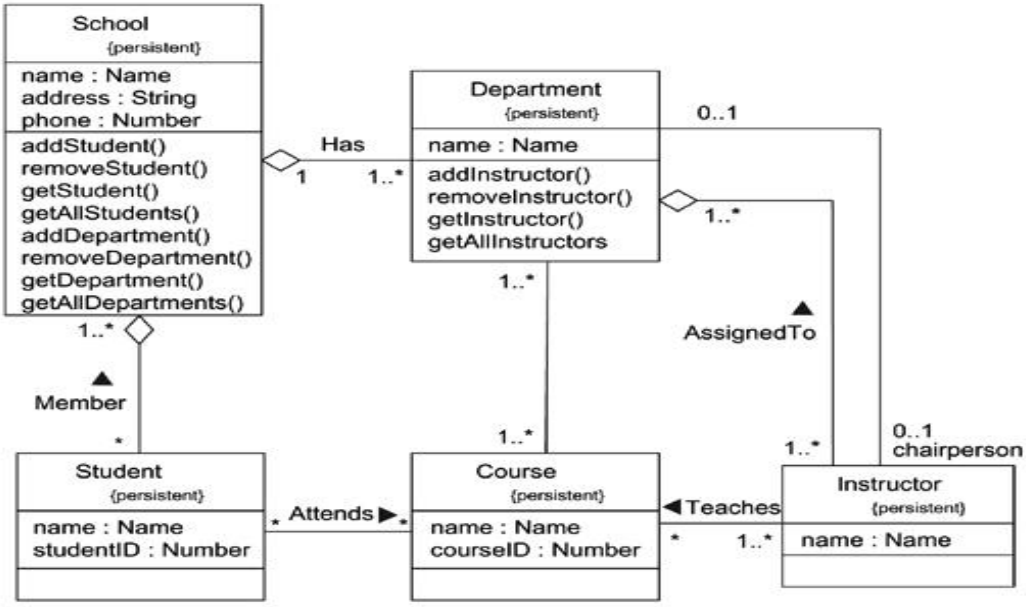
Modeling Simple Collaborations

- When you create a class diagram, you just model a part of the things and relationships that make up your system's design view. For this reason, each class diagram should focus on one collaboration at a time.
- To model a collaboration
    - Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.
    - For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration. Identify the relationships among these things, as well.
    - Use scenarios to walk through these things. Along the way, you'll discover parts of your model that were missing and parts that were just plain semantically wrong.
    - Be sure to populate these elements with their contents. For classes, start with getting a good balance of responsibilities. Then, over time, turn these into concrete attributes and operations.



Modeling a Logical Database Schema

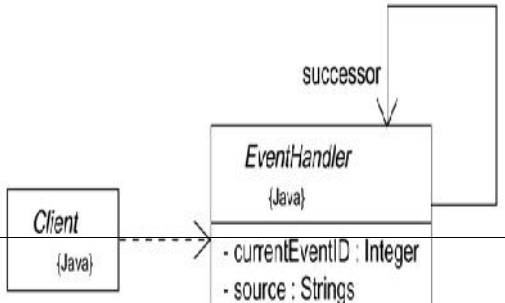
- The UML is well-suited to modeling logical database schemas, as well as physical databases themselves.
- The UML's class diagrams are a superset of entity-relationship (E-R) diagrams, Whereas classical E-R diagrams focus only on data, class diagrams go a step further by permitting the modeling of behavior, as well. In the physical database these logical operations are generally turned into triggers or stored procedures.
- To model a schema,
    - Identify those classes in your model whose state must transcend the lifetime of their applications.
    - Create a class diagram that contains these classes and mark them as persistent (a standard tagged value). You can define your own set of tagged values to address database-specific details.
    - Expand the structural details of these classes. In general, this means specifying the details of their attributes and focusing on the associations and their cardinalities that structure these classes.
    - Watch for common patterns that complicate physical database design, such as cyclic associations, one-to-one associations, and n-ary associations. Where necessary, create intermediate abstractions to simplify your logical structure.
    - Consider also the behavior of these classes by expanding operations that are important for data access and data integrity. In general, to provide a better separation of concerns, business rules concerned with the manipulation of sets of these objects should be encapsulated in a layer above these persistent classes.
    - Where possible, use tools to help you transform your logical design into a physical design.



Modeling a Schema

Forward and Reverse Engineering

- **Forward engineering** is the process of transforming a model into code through a mapping to an implementation language
- Forward engineering results in a loss of information, because models written in the UML are semantically richer than any current object-oriented programming language.
- To forward engineer a class diagram,
  - Identify the rules for mapping to your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
  - Depending on the semantics of the languages you choose, you may have to constrain your use of certain UML features. For example, the UML permits you to model multiple inheritance, but Smalltalk permits only single inheritance. You can either choose to prohibit developers from modeling with multiple inheritance (which makes your models language-dependent) or develop idioms that transform these richer features into the implementation language (which makes the mapping more complex).
  - Use tagged values to specify your target language. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages.
  - Use tools to forward engineer your models.



Forward Engineering

- **Reverse engineering** is the process of transforming code into a model through a mapping from a specific implementation language.
- Reverse engineering results in a flood of information, some of which is at a lower level of detail than you'll need to build useful models.
- Reverse engineering is incomplete. There is a loss of information when forward engineering models into code, and so you can't completely recreate a model from code unless your tools encode information in the source comments that goes beyond the semantics of the implementation language.
- To reverse engineer a class diagram,

    - Identify the rules for mapping from your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.

    - Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered.

    - Using your tool, create a class diagram by querying the model. For example, you might start with one or more classes, then expand the diagram by following specific relationships or other neighboring classes. Expose or hide details of the contents of this class diagram as necessary to communicate your intent.

Object Diagram

- An object diagram is a diagram that shows a set of objects and their relationships at a point in time.
- Graphically, an object 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

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,

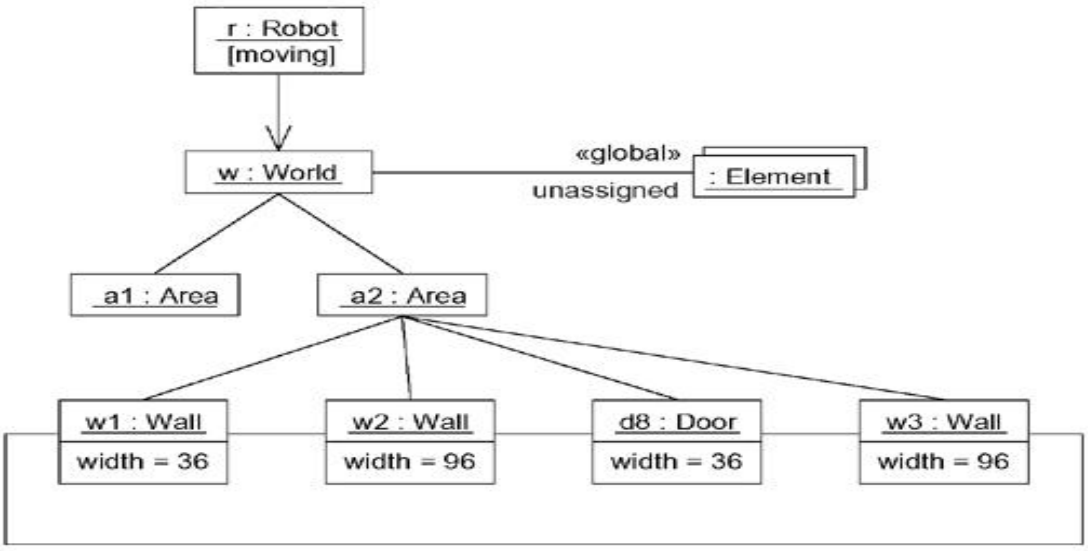
    - Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.

    - 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. Freeze that scenario at a moment in time, and render each object that participates in the 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
- In an object-oriented system, instances are things that are created and destroyed by the application during run time. Therefore, you can't exactly instantiate these objects from the outside.
- 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. Typically, you'll set your context inside an operation or relative to an instance of one particular class.

    - 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.

- If your diagram ends up overly complicated, prune it by eliminating objects that are not germane to the questions about the scenario you need answered. If your diagram is too simplistic, expand the neighbors of certain interesting objects and expose each object's state more deeply.

II UNIT

Essay Questions

1. Define the terms:
   a) Classes b) Interface c) Packages
2. Draw the notations for Classes , Interface and Packages?
3. List various types of relationships?
4. Compare and Contrast relationships between classes , interface and packages?
5. How do you show packages in a UML diagram?
6. When working with multiple packages, what type of relationship is most commonly shown?
7. What are the two elements that are most commonly collected into packages?
8. How can the accessibility of the contents inside a package be determined?
9. What is the standard notation for specifying the diagrams inside a package?
10. How do you document inheritance in UML?
11. What is the purpose of documenting the inside of a class?
12. What is an Object diagram and interface ?
13. What are the benefits of programming with interfaces?
14. List three ways to draw an interface in a Class diagram.

Short Questions

1.What is cardinality?
2.What are associations?
3.What is dependency?
4.What is an interface?
5.What is responsibility-driven design?
6.Draw the notations for Classes , Interface and Packages?
7.How do you show packages in a UML diagram?
8.List various types of relationships?
9.What is cardinality?
10.What is an Object diagram?
11.List three ways to draw an interface in a Class diagram?

Assignment Questions

1. Define the terms:
   A) Classes
   B) Interface
   C) Packages
2. What is the purpose of documenting the inside of a class?
3. When working with multiple packages, what type of relationship is most commonly shown?
4. What are the two elements that are most commonly collected into packages?
5. How can the accessibility of the contents inside a package be determined?
6. What is the standard notation for specifying the diagrams inside a package?
7. How do you document inheritance in UML?
8. Compare and Contrast relationships between classes , interface and packages?

Objective Questions

1. Which on of the following is grouping thing? [ ]
   a) Class b)Package c)Use Case d) Collaboration
2. An _____ is a class which objects own one or more processes or threads and therefore can initiate control activity. [ ]
   a)Active class b)Abstract class c)Alternate class d)Interface class
3. An interface is rendered as _____ [ ]
   a) Cube b)Square c)Rectangle d)Circle
4. An _____ is a collection of operations that specify a service of a class or component.
   a) Interface b) Active class c) Use case d) interaction [ ]
5. _____ helps us in protecting privacy of Objects.
   a)Encapsulation b)Abstraction c)Polymorphism d)Inheritance [ ]
6. _____ things are the names of UML models.
   a)Structural b)Behavioral c)Grouping d)Annotational [ ]
7. _____ things are the dynamic parts of UML models. [ ]
   a)Structure b)Behavioral c) grouping d) Annotational
8. _____ is a special kind of association, representing a structural relationship between a whole and its parts. [ ]
   a)Dependency b)Aggregation c)Generalization d) Realization
9. A _____ relationship is rendered as a solid line with a hollow arrow head pointing to the parent. [ ]
   a) Dependency b)Aggregation c)Generalization d)Realization
10. Aggregation is a _____ kind of relationship. [ ]
    a)is- a b)to-a c)was-a d)has-a
11. A _____ is a contract or an obligation of a class. [ ]
    A)Usability B)responsibility C)package D)State

12.A _____ is just the face the class at the neor end of the association presents to the other end of the association. [ ]
A)name B)role C)multiplicity D)aggregation

13.An _____ is a named property of a class that describes a range of values that instance of the property may held. [ ]
A)item B)Attribute C)operation D)Entity

14. A _____ is a description of a set of objects that store the same attributes , operations ,relationships & semantics.
[ ]
A)Class B)Entity C)Function D)Procedure

15.An _____ is a named property of a class that describes a range of values that instance of the property may held.
[ ]
A)item B)Operation C)Attribute D)Entity

16.You use _____ diagrams to illustrate data structures , the static snapshots of instances of the things found in class diagrams.
A)Usecase B)object C)collaboration D)sequence [ ]

17.Name:string_____. [ ]
B)+ origin. C)head:*item. D)namejO.1J : string.

18.Notes may be attached to more than one element by using _____ [ ]
A)associations B)dependences C) generalizations D) aggregations

19.The visibility of private is _____. [ ]
A)+ B)# C)- D)I

20..A _____ relationship is rendered as a solid line with a hoiiow arrowhead pointing to

21.A _____is a semantic relationship between two things in which a change to one thing

22.A _____is a semantic relationship between classifiers where in one classifier specifies a
contract that another classifier gnarantees to carry out.

23.UML includes _____ diagrams.

24. _____ is a specification of the range of allowable cordinalities that a set may assume.

25._____stereotype specifies that the corresponding objects is visible because it is in an local scope.

26._____ diagrams permit you to model the lifetime of an object.

27.Collaboration diagrams use _____relationship

28._____diagrams do a better job of visualizing complex iteration and branching and of visualizing multiple concurrent flow of control than do sequence diagrams.

29._____ is defined as that you can take one and transform it into the other without loss of information

30. a procedural or nested flow of control rendered using _____

UNIT –II

MULTIPLE CHOICE

1) B, 2) A ,3) D, 4) A, 5)A, 6) A, 7) B, 8) B ,9) C ,10) D, 11) B, 12) B, 13)B ,14) A, 15) C,16) B, 17) B ,18) D ,19) D,

FILL IN THE BLANKS

20. GENERALIZATION, 21. DEPENDENCY, 22). REALIZATION, 23) NINE, 24) MULTIPLICITY, 25) SELF , 26) .SEQUENCE , 27) MESSAGE ,28) USE CASE, 29) DEPLOYMENT, 30) A FILLED SOLID ARROW HEAD.