# 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.

- 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

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.

- 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.

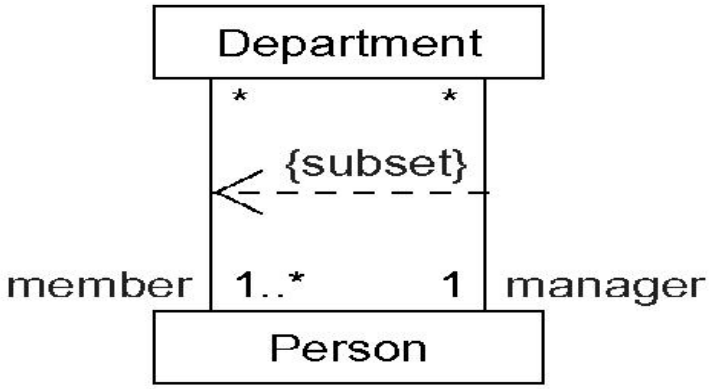


**Modeling Nonsoftware Things**

- 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**




## 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

- 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.

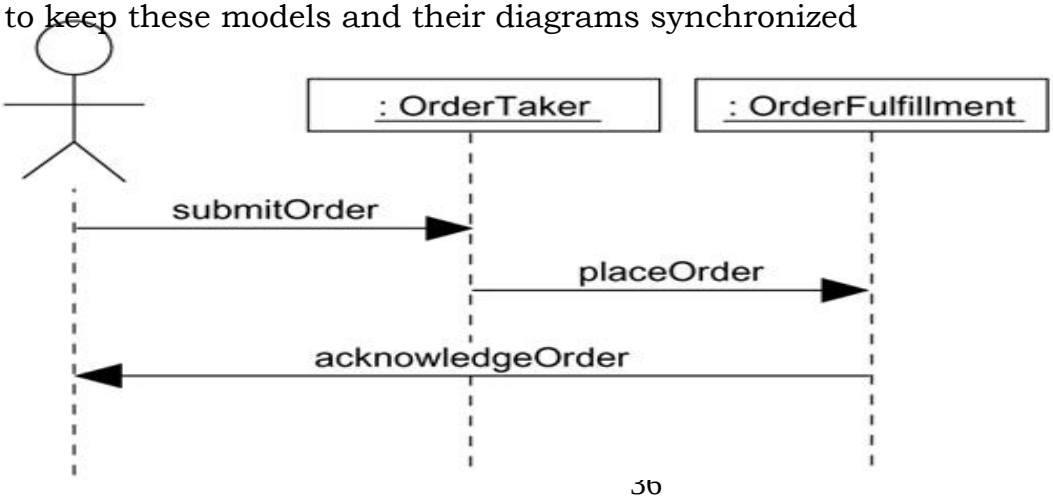
**Nodes and their components:**

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



- 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**

- 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.

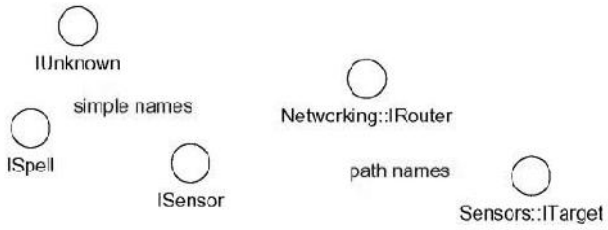
**role**

- 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

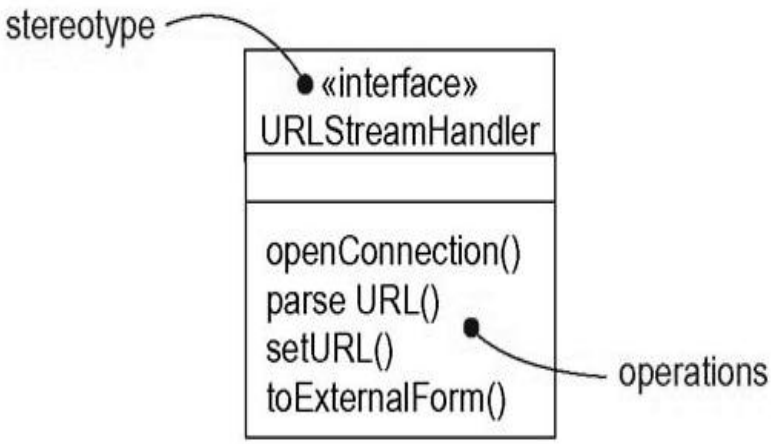



**Simple and Path Names**

**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**

## 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,

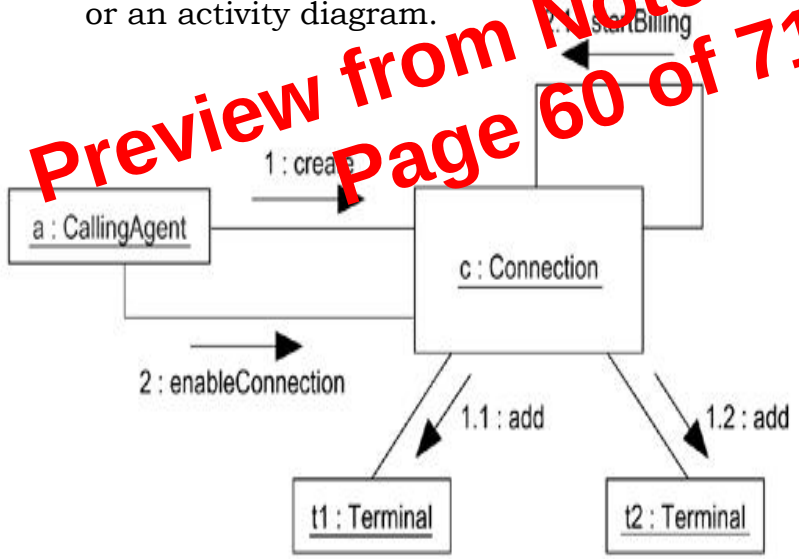
  - Identify those prototypical instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.

  - 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.





- 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.