



Introduction to UML

UML is an acronym for Unified Modeling Language. As its name indicates, it is a graphical language, used to create visual models of software intensive systems. The UML is an industry standard for object oriented design notation, supported by the Object Management Group (OMG). The UML represents a collection of best engineering practices for modeling large and complex systems. The UML uses graphical notations to design software projects. It is a visual language with graphical symbols used for visualizing, specifying, constructing and documenting various artifacts of a software system.

Visualizing

The UML is the language used by every stakeholder involved in the software intensive system, to better understand the conceptual models of the system. UML can build models of the complex systems which are difficult to comprehend mentally. The UML models the software systems for better communication among all parties involved for good team work resulting into successful software projects. UML helps the project teams to communicate, explore potential designs and validate architectural design of the software systems.

Specifying

The UML builds models which are precise, simple, complete and unambiguous. The

LEARNING OBJECTIVES

After studying the chapter the students familiarize themselves with the following concepts:

- ◆ History of UML
- ◆ Principles and Importance of Modeling
- ◆ Understanding UML with its Rules and Building Blocks

UML addresses the tasks of design, analysis and implementation to develop and deploy software intensive systems. By using UML we can model application's structure, behavior, architecture and data.

Constructing

The UML is used for forward and reverse engineering. UML models can be mapped to any object-oriented programming languages such as Java, C++ and Visual basic. We can carry forward engineering, where source code can be generated by using system models. UML can also be applied for reverse engineering, where system models are generated from the source code.

Documenting

The UML can be used for documenting various features and characteristics of any software-intensive system. UML can document system's requirements, system's architecture, test cases, project plan and release specifications of any software development project.



1.1 History of UML

In the late 1980s and early 1990s, people used a variety of object-oriented design techniques and notations. Different software development companies were using different notations to analyze, design and document their object-oriented systems. These diverse notations used, lead to confusion and ambiguity.

UML was developed to standardize the large number of object-oriented modeling notations that existed and used extensively in the early 1990s. The major ones used were, Object Management Technology notations developed by Rumbaugh in 1991, Booch's methodology notations developed by Grady Booch in 1991, Object-Oriented Software Engineering notations developed by Jacobson in 1992, Odell's methodology notations developed by Odell in 1992, and Shaler and Mellor methodology notations developed by Shaler in 1992. The UML adopted many concepts from all these techniques and notations. Later, UML was adopted by Object Management Group (OMG) as a de facto standard in 1997.



1.2 Importance of Modeling

What is a Model ?

A model is a simplification of reality. A model can be considered as a blue print. A blueprint describes an idea, a feature and a process involved. A blueprint can be defined as a paper based technical drawing, an architecture of a system or

an engineering design. More generally, the term blueprint refers any detailed plan.

A potter, who makes a pot, has a model of the pot he is making, in his mind. The model could be conceptual and visual image of the pot, describing its size, shape and appearance. A pot model is simple, so the potter can easily comprehend it in his mind. The same potter, can make different varieties of pots based on different visual models of them, he has in his mind.

The following diagrams specify making of different pots based on the model the potter has in his mind.



Fig. 1.1(a) Pot Model 1



Fig. 1.1(b) Pot Model 2



Fig. 1.1(c) Pot Model 3

Some of the models could be little complex to have them in mind. In such cases we have paper representation describing the model. The model can be a descriptive text or a collection of graphical figures. A tailor who is stitching a dress has the model of the dress in the form of specifications, requirements and measures given by the client.

The following diagrams indicate the model for stitching a suit.



Fig. 1.2(a) Taking the Model



Fig. 1.2(b) Construction



Fig. 1.2(c) Deployment

The above models are quite easy to comprehend mentally. In cases of handling complex situations visual models can play a major role in understanding the system. Imagine how difficult it would be to understand the layout of a building without a set of visual plans or models.



Fig. 1.3(a) Model of a Construct



Fig.1.3(b) The actual Construct

A model is the simplified conceptual picture of the thing that is getting developed into a physical reality. A model is a complete description of the system specified textually or graphically, representing different aspects of it. Suppose, any aeronautical industry, developing a new fighter aircraft, the process starts with the design containing visual models representing different aspects of the new fighter aircraft. We cannot build a running prototype of the physical aircraft without having a model of it on a paper or on a computer, specifying all the features and details of it. The following gives the model of it representing different aspects and its physical reality when it is built.

Models representing Different aspects of Aircraft

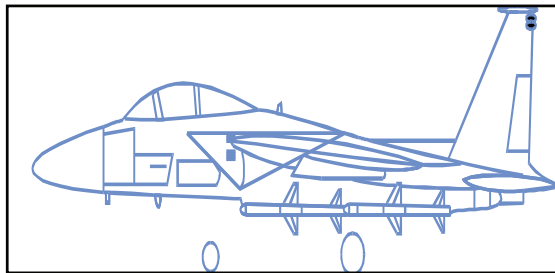


Fig. 1.4(a) Side View

The Physical reality

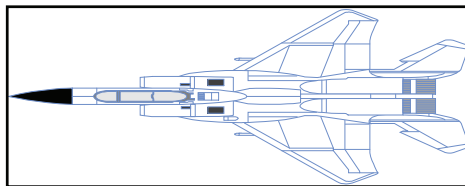


Fig. 1.4(b) Top View



Fig. 1.4(c) The Real Aircraft when developed

Hope you have understood what a model is. The same visual models are applicable even in the software engineering domain. Any software project must start with analysis and design using visual models of the system. Hence a

model is the simplification of reality to start with. The following section answers why we need models.

Why we need Models ?

Modeling plays a significant role in large projects belonging to the various engineering disciplines. Models are essential parts of any software engineering projects. A model plays a major roll in software development as blueprints and other plans such as site maps, elevation photos, and physical models play in the building of a skyscraper construct. Modeling achieves the following objectives:

- ❖ Helps us to visualize a system as we want it to be.
- ❖ Permits us to specify the structure or behavior of a system.
- ❖ Gives us a template that guides us in constructing a system.
- ❖ Documents the decisions we have made.

We build models of complex systems because we cannot comprehend such a system in its entirety. We build models to better understand the system we are developing. Any software application is built without building models of it, is bound to fail. Software Modeling is an important aspect as it ensures software quality. Software source code and models are mutually related. We can generate source code from models, and models could be automatically created using source code. Models enhance communication among team members belonging to a project. Models ensure better planning, risk reduction, and reduced costs etc. There are many elements that contribute to a successful software organization; one common basis is the use of modeling. Modeling is a proven and well-accepted engineering technique.

The model would provide a way to understand the business, a basis for the physical structure needed to support the business. The model also helps us to understand the project, and to comprehend the business in general. The model not only reflects the modeler's interpretation of the project scope and business needs, but it also provides a means to communicate with the client's business community. Another important aspect is that the model helps us understand our project within the context of the overall enterprise.

You can develop visual, system analysis models for any software systems using UML. UML helps in building a data model, using class and object diagrams. It helps in building visual models, depicting the system's functionality, using use case and activity diagrams. Visual models can be created, for specifying the entire system's behavior, using state chart diagram and interaction diagrams. UML will be discussed extensively in later chapters.



1.3 Principles of Modeling

A model is a descriptive, functional, or physical representation of a system. Modeling is a way of thinking and reasoning about systems. The goal of

modeling is to come up with a representation that is easy to use in describing systems in a mathematically consistent manner.

The four basic principles of modeling are as follows:

- ❖ **Principle one:** The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.
- ❖ **Principle two:** Every model may be expressed at different levels of precision.
- ❖ **Principle three:** The best models are connected to reality.
- ❖ **Principle four:** No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

Principle One

The Choice of Model Is Important. In software, the models you choose greatly affect your world view. Each world view leads to a different kind of system, with different costs and benefits. If you build a system through the eyes of a database developer, you will likely end up with entity-relationship models that push behavior into stored procedures and triggers. If you build a system through the eyes of an object-oriented developer, you will end up with a system that has its architecture centered around many classes and patterns of interaction that direct how those classes work together.

Principle Two

While you are building models, the levels of precision may differ based on what context you are in. While you are building a big apartments complex, sometimes the buyer is interested to see the front elevation, and yet other times he is interested in the internal architecture of an apartment. The best kinds of models are those that let you choose your degree of detail, depending on who is doing the viewing and why they need to view it. For example, when you are developing a GUI system, a quick executable model of the user interface without bothering about other details or quality constraints could be your intention. Other times, when you are dealing with cross-system interfaces of network bottlenecks, you need to model down to the bit level.

Principle Three

Suppose, you have a mathematical model of the house that exists only in ideal environment, that is it cannot withstand, sun light and rains. Such model is the one which is away from the physical reality. When you have a model of the car, the model has to be connected to reality. The car when physically manufactured it should withstand all natural conditions such as it should run in sun light, rains and it should withstand snow fall etc. Even in case of software development, the models of it have to be build in a way, when software is ready, it should work in a realistic environment. It can happen, only when you develop models

connected to reality. We know that, there can be several independent views of a system represented by different models. All these models are assembled into one semantic whole model of the system.

Principle Four

Suppose you are constructing a shopping complex, there may not be single set of blueprints that reveal all its details. You have separate models for floor plans, elevations, electrical plans, and plumbing plans. Although these models are nearly independent, still they are interrelated. The model representing electrical plans can be studied in isolation, but you understand their mapping to the models of the floor plan and the plumbing plan. This is also applicable for object-oriented software systems. To understand the architecture of such a system, you need nearly independent but interrelated views such as use case view, a design view, a process view, an implementation view, and a deployment view. These views, together represent the system, which is under development. The following diagram indicates how all these views which are nearly independent but are interrelated.

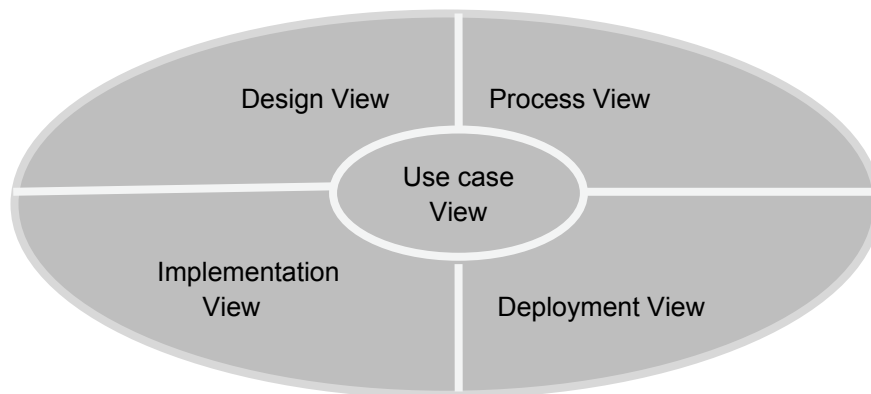


Fig.1.5 Views of an Object Oriented system



1.4 Object-Oriented Modeling

Architects build many kinds of models. These models could be structural models that make people visualize and specify parts of systems and how they relate to one another. They may also build dynamic models to understand the behavior of the structure during cyclones and earthquakes.

In software, there are two most common ways to approach a model. You can build a model from an algorithmic approach and an object-oriented approach. In algorithmic approach, the basic building blocks of software are procedures or functions. A procedure or a function contains a set of instructions to accomplish a task or a purpose. In this approach, larger algorithms are decomposed into

smaller ones and developed independently and later integrated. The systems built with this approach have problems in maintaining the software as requirements change or when the system grows.

In object-oriented approach, the major building blocks are objects or classes. A class is a category for a set of common objects. Every object has its own name to distinguish it from others. An object has state, in the form of attributes or data associated with it. It has behavior specified in the form of operations. The whole behavior of an object-oriented system can be expressed in the form of interactions among objects that constitute the system. The major advantage of an object-oriented approach is reusability. We can build a new system based on already available and fully tested objects by assembling them as we build a vehicle by assembling the various parts of it which are independently manufactured by independent vendors. New object-oriented systems can be built by assembling already existing components in the software market, such as Java beans or COM, DCOM.

Visualizing, specifying, constructing, and documenting systems which are built based on object-oriented approach is the primary purpose of UML.



1.5 Understanding UML

UML is a modeling language whose vocabulary and rules focus on the conceptual and physical representation of a system. Some things are best modeled textually; other are best modeled graphically. UML is a visual modeling language having graphical symbols as its vocabulary. A modeling language such as UML, is thus a standard language for software blueprints. The UML addresses the specification of all the important analysis, design and implementation decisions relating to any software systems. UML is not only a visual programming language, but its models can be directly connected to a variety of programs, in fact, source code can be generated directly from UML models. UML models can be used for analyzing the problem-domain which includes simplifying the reality, capturing requirements, visualizing the system in its entirety, and specifying the structure and/or behavior of the system. UML models can be applied for designing the solution which includes documenting the solution in terms of its structure and/or behavior. UML provides the notations for documenting some of the artifacts such as requirements, system design and test cases and test procedures.

The UML is largely process independent. However, to get the most benefit from the UML, you should consider a process that is:

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

The UML is not limited to only modeling softwares. You can also model non software systems such as:

- ❖ Work Flow in the legal system.
- ❖ The Patient Healthcare system.
- ❖ The Design of hardware.



1.6 Building Blocks of the UML

The vocabulary of UML include three kinds of building blocks:

- ❖ Things.
- ❖ Relationships.
- ❖ Diagrams.

The Things are:

- ❖ Structural Things.
Classes, Interfaces, Collaborations, Use cases, Active classes,
Components, Nodes.
- ❖ Behavioral Things.
Interactions, State Machines.
- ❖ Grouping Things.
Packages.
- ❖ Annotational Things.
Notes

The Relationships are:

- ❖ Dependency.
- ❖ Association.
- ❖ Realization.
- ❖ Generalization.

The Diagrams are:

- ❖ Class Diagram.
- ❖ Object Diagram.
- ❖ Use case Diagram.
- ❖ Sequence Diagram.
- ❖ Collaboration Diagram.
- ❖ State chart Diagram.
- ❖ Activity Diagram.
- ❖ Component Diagram.
- ❖ Deployment Diagram.

Structural Things

Structural things are nouns of the UML models. These are mostly static parts of the model, representing elements which are conceptual or physical. There are seven structural things supported by UML.

- ❖ Class
- ❖ Interface

- ❖ Collaboration
- ❖ Use case
- ❖ Active Class
- ❖ Component
- ❖ Node

Class

A class is a category of set of objects that share common attributes, operations, relationships and semantics. Attributes are the named properties of a class depicting its state, whereas operations are the services offered by a class depicting its behavior. A class is graphically represented as a rectangle containing three components, indicating name, attributes and operations.

Example:

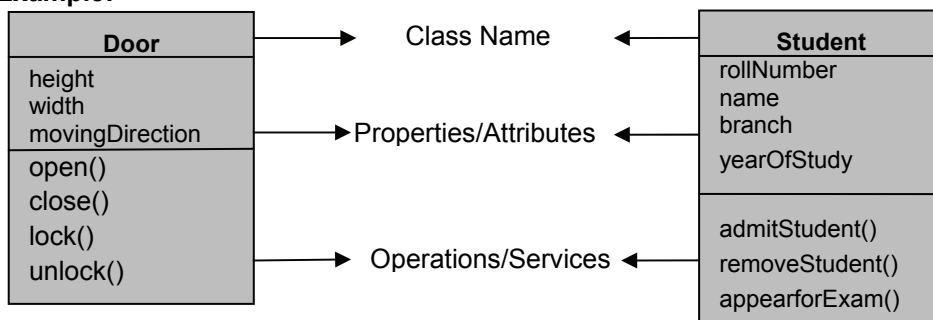


Fig. 1.6 (a) Class Definition (Door)

Fig. 1.6(b) Class Definition (Student)

Interface

It is a collection of operations, which specify a service of a class or a component. Interface contains operations, but not their implementations. A TV Remote is an interface for the service of a class named TV Set. An interface is attached to the class or component that realizes an interface. An interface shares the same features as a class; in other words, it contains attributes and methods. The only difference is that the methods are only declared in the interface and will be implemented by the class implementing the interface. An interface is graphically represented as a circle with its name in UML.

Example:

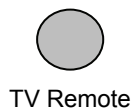


Fig. 1.7(a) Defining Interface

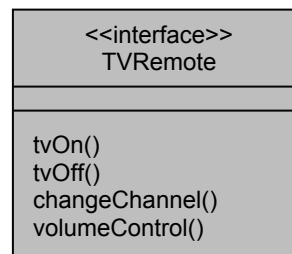


Fig. 1.7(b) Another way of defining Interface

Collaboration

It defines an interaction among different elements of UML to provide some cooperative behavior. Collaborations specify structure as well as behavior. These are implementations of patterns that make up the system. A collaboration is graphically represented as a dashed ellipse.

Example:

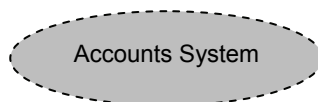


Fig 1.8(a) Collaboration

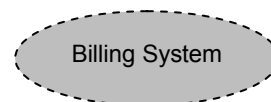


Fig 1.8(b) Collaboration

Use case

It specifies the behavior of a system as a whole or a part of it in the form of set of functions. It is realized by a collaboration. A use case is graphically represented as a full ellipse.

Example:

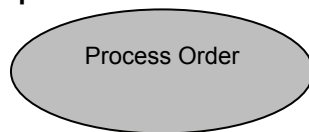


Fig 1.9(a) Use case Definition

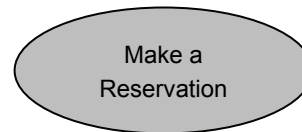


Fig 1.9(b) Use case Definition

Active Class

It is a class whose instance is an active object. An active object is an object that owns a process or thread. It can initiate control activity. An active class is graphically represented as an ordinary class but with thick boundary.

Example:

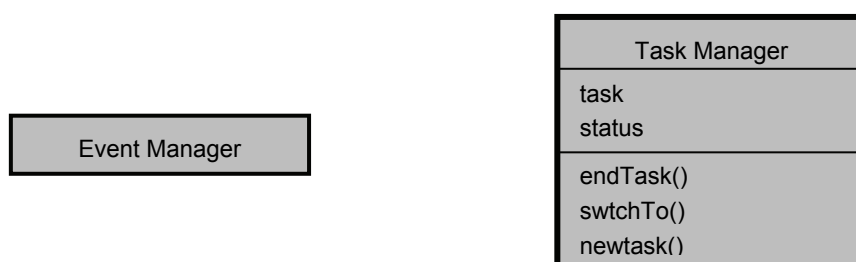


Fig. 1.10 Defining Active Class

Component

It is a physical and replaceable part of the system. A component typically manifests itself as a piece of software. Graphically, a component is represented as a rectangle with tabs, usually including only its name.

Example:

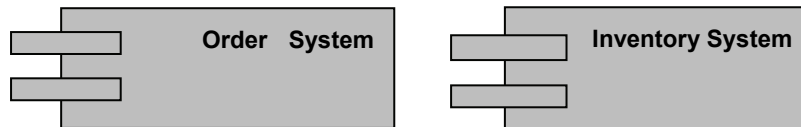


Fig. 1.11 Component Definition

Node

It is a physical element that exists at run-time and represents a computational resource. It is typically, a hardware resource of the system. It has at least some memory and processing capability. A node can be a personal computer, a workstation, mini or main frame computer or an electronic device with some memory and computing power. A node is graphically represented as a cube containing its name.

Example:

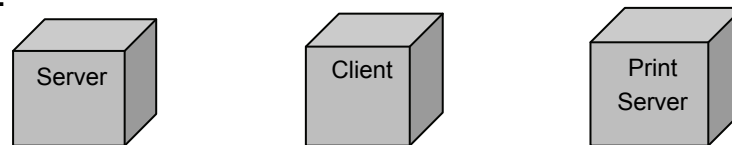


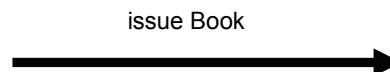
Fig. 1.12 Defining Nodes

Behavioral Things

These are the the verbs of UML models; usually the dynamic parts of the system in question. They represent the behavior of the system over time and space. There are two kinds of behavioral things:

- ❖ **Interaction:** some behavior constituted by messages exchanged among objects; the exchange of messages is with a view to achieving some purpose. An interaction specifies the behavior of a society of objects or of an individual operation of a class. A message is graphically represented as a directed line including the name of its operation.

Example:



- ❖ **State Machine:** A behavior that specifies the sequence of “states” an object goes through, during its lifetime. A “state” is a condition or situation during the lifetime of an object during which it exhibits certain characteristics and/or performs some function. A state machine can be

used to indicate the behavior of a class or a collaboration of classes. A state machine involves state (the current status), transition (flow from one state to the other state), event (that causes transition), and activities (response to a transition). A state machine is graphically represented as a rounded rectangle indicating the name of its current status and its sub states, if any.

Example:

Fig. 1.13 Defining State

Grouping Things

These are the organizational parts of the UML models. They provide higher level of abstraction. These are the containers into which a model can be placed. There is only one kind of grouping thing, named package.

Package

It is a general-purpose element that comprises UML elements - structural, behavioral or even grouping things. Packages are conceptual groupings of the system and need not necessarily be implemented as cohesive software modules. A package is graphically represented as a tabbed folder including its name.

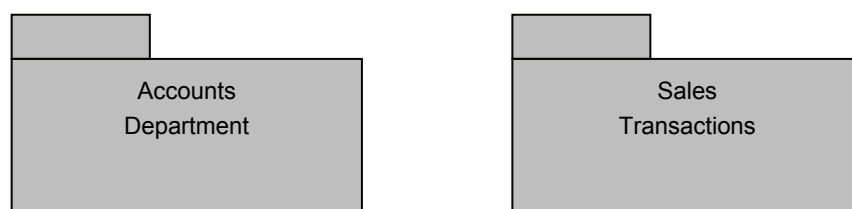
Example:

Fig.1.14 Defining Packages

Annotational Things

These are the explanatory part of the UML model; adds information/meaning to the model elements. There is only one kind of annotational thing, named Note.

Note

It is a graphical notation for attaching constraints and/or comments to elements of the model. Using notes you can attach explanatory comments to an element

or a collection of elements. It is graphically represented as a rectangle with a dog-eared corner containing a textual or graphical comment.

Example:

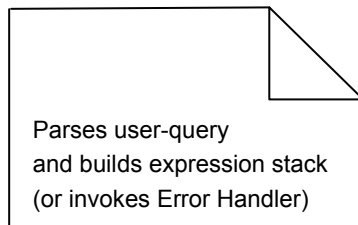


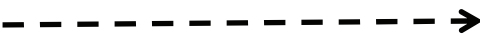
Fig.1.15 Defining Note

Relationships

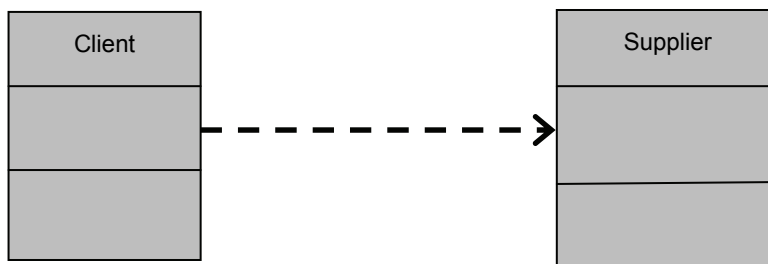
There are four relationships supported by UML models.

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. Usually a class may depend on another class or on interface. Typically, dependency relationships do not have names. As the following figure illustrates, a dependency is displayed in the diagram editor as a dashed line with an open arrow that points from the client to the supplier.

Notation: 
(arrow-head points to the independent thing)

Example:



Note: Client class depends on the Supplier class.

Fig 1.16 Defining Dependency

Association

An association represents a structural relationship that connects two classifiers, such as classes or use cases, that describes the reasons for the relationship and

the rules that govern the relationship. Like attributes, associations record the properties of classifiers. An association appears as a solid line between two classifiers, and association ends indicate the roles played by them including properties such as multiplicity and constraints.

Notation

Example:

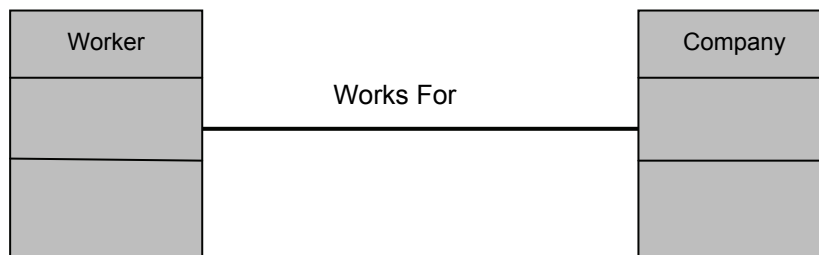


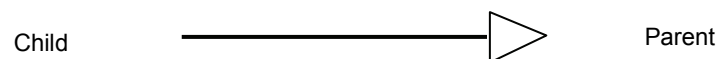
Fig.1.17 Defining Association

Note: Worker works for a Company. Worker and Company play roles in association. More than one worker can work in a single Company indicating multiplicity at association ends.

Generalization

It is a relationship in which one model element (the child) is based on another model element (the parent). The parent element is the generalized one and the child element is considered as the specialized one. This relationship is applicable to class, component, deployment, and use-case diagrams to indicate that the child receives all of the attributes, operations, and relationships that are defined in the parent. The model elements in a generalization relationship must be the same type. For example, a generalization relationship can be used between actors or between use cases; however, it cannot be used between an actor and a use case. The child model elements in generalizations inherit the attributes, operations, and relationships of the parent, you must only define for the child the attributes, operations, or relationships that are distinct from the parent. Graphically, a generalization relationship is displayed in the diagram editor as a solid line with a hollow arrowhead that points from the child model element to the parent model element.

Notation



Example:

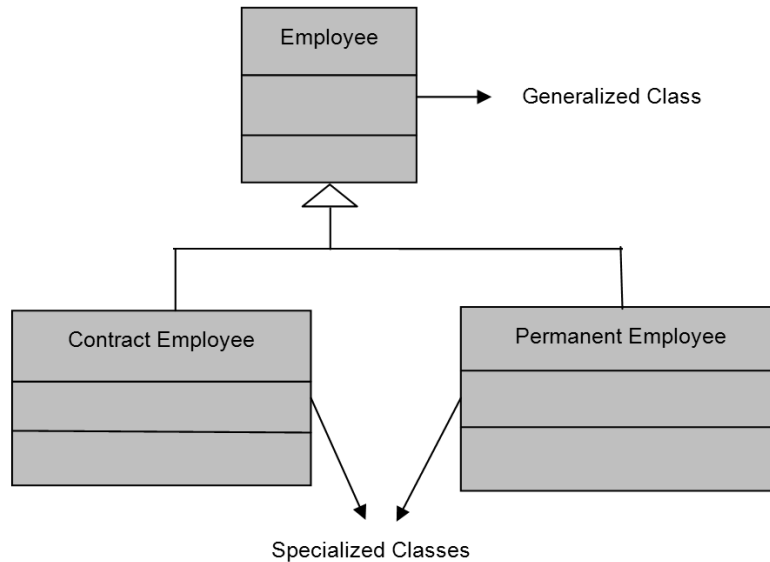
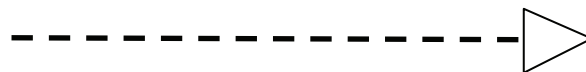


Fig 1.18 Defining Inheritance

Realization

In UML modeling, a realization relationship is a relationship between two model elements, in which one model element, implements or executes the behavior that the other model element, specifies. This relationship is available between interfaces and the classes or the components that realize them and also between use cases and the collaborations that realize them. A realization is indicated by a dashed line with an unfilled arrowhead towards the supplier. Realizations can only be shown on class or component diagrams.

Notation



Example:

1. This relationship between a class and an interface indicates that the interface specifies the behavior to be carried out in the form of its operations, and the class implements that behavior.

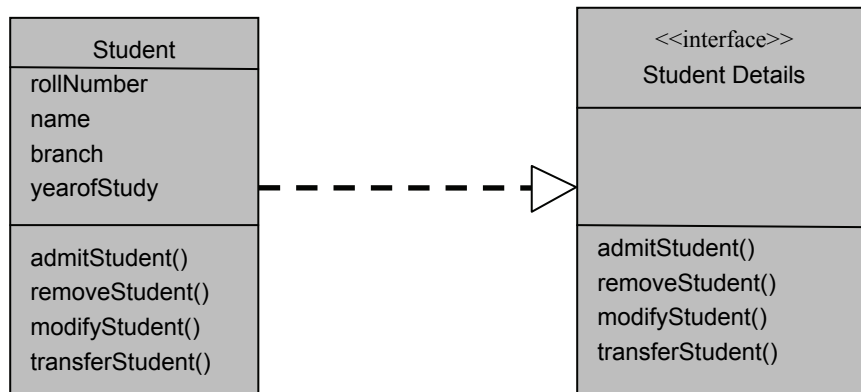


Fig.1.19 Defining Realization

2. This relationship between a use case and the collaboration indicates that the use case specifies the behavior to be carried out and the collaboration implements it.



Fig.1.20 Defining Realization among use cases

Diagrams in the UML

A UML diagram is a graphical presentation of the UML model. It is represented as a connected graph containing vertices (things) connected by arcs (relationships). The UML diagrams help us to visualize a system from different perspectives. UML diagrams let developers and customers view a software system from different perspectives and in varying degrees of abstraction. UML includes nine diagrams - each capturing a different dimension of a software system architecture.

Class Diagram

Class diagrams are widely used to describe the types of objects in a system and their relationships. It includes elements of a model, such as classes, interfaces and collaborations and how they are interrelated. Class diagrams indicate the structural or static part of the system. Class diagrams address static design view of the system. Class diagrams may also include design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system, conceptual, specification, and

implementation. These perspectives become evident as the diagram is created and help improve the design.

Object Diagram

Class diagrams are conceptual ones, but object diagrams are physical. An object diagram contains set of objects and how they are connected or linked together. It indicates the snapshot of an instance of a class diagram at a particular instance of time. Object diagrams address the real and prototypical perspectives of any software intensive system. Objects are nothing but instances of classes.

Use Case Diagram

It is a graphical representation consisting of use cases and actors and how they are related. Use case diagrams address the dynamic part of the system. They are helpful in modeling the behavior of the system. A use case is a set of scenarios that describe an interaction between a user and a system. The two main components of a use case diagram are use cases and actors. An actor represents a user or another system that will interact with the system you are modeling. A use case is an external view of the system that represents some action the user might perform in order to accomplish a task.

Interaction Diagrams

Interaction diagrams model the behavior of a use case by depicting the way a set of objects interact in order to complete a task. There are two kinds of interaction diagrams, namely sequence and collaboration diagrams. Both of these diagrams are isomorphic, i.e., they are semantically equivalent and one can be generated from the other without the loss of any information. Interaction diagrams address the dynamic view of the system.

Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass among themselves. Sequence diagrams emphasize the time ordering of messages. Sequence diagrams contain objects with their life lines coming from top to bottom. The interactions among objects is specified with the messages passed among them.

Collaboration diagrams demonstrate the structural organization of the objects that interact, it shows how objects are statically connected. If you have a sequence diagram, you can transform it into a collaboration diagram and vice versa. They show the relationship between objects and the order of messages passed between them. The objects are listed as icons and arrows indicate the messages being passed between them. The numbers next to the messages are

called sequence numbers. As the name suggests, they show the sequence of the messages as they are passed between the objects.

State chart Diagram

It is a graphical representation consisting of states, transitions, events and activities. A state machine displays the sequences of states that an object goes through during its life time in response to received external or internal events, together with its responses and actions. State diagrams are used to describe the behavior of a system. State diagrams, describe all of the possible states of an object goes through as events occur. Each diagram usually represents objects of a single class and track the different states of its objects through the system.

Activity Diagram

It is a special kind of state chart diagram where most of the states are action states and most of the transitions are triggered by completion of the actions in the source states. This diagram focuses on flows driven by internal processing. Activity diagrams describe the workflow behavior of a system. Activity diagrams can show activities that are conditional or parallel. Activity diagrams emphasize the flow of control among objects as activities are carried out by the system.

Component Diagram

A component diagram shows how the various components that constitute a system are organized and physically related to each other. It displays the high level packaged structure of the code itself. Component diagrams show the software components of a system and how they are related to each other. Dependencies among components are shown, including source code components, binary code components, and executable components. Some components exist at compile time, at link time, at run time as well as at more than one time. Component diagram addresses the static implementation view of the system.

Deployment Diagram

A deployment diagram shows the configuration of nodes (processing elements) and the components that live on them. It gives the static deployment view of the system's architecture. This diagram displays the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of code units.

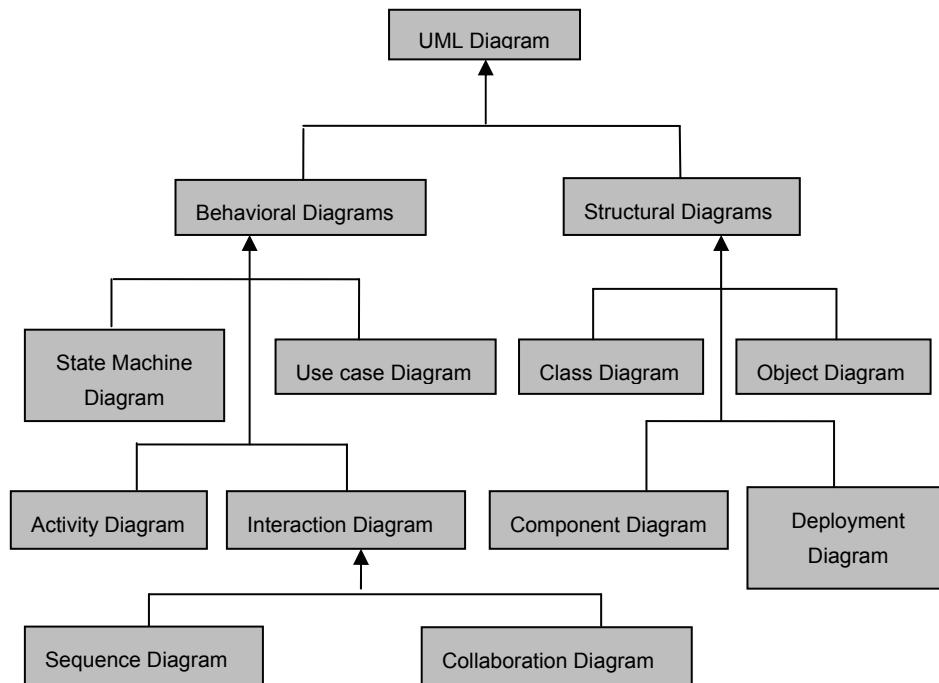


Fig.1.21 Classification of UML Diagrams



1.7 Rules of the UML

UML being a visual modeling language, has number of rules indicating how UML's building blocks can be combined together to build various system models. These UML rules specify what a well-formed model should look like. Well-formed means that a model or model fragment adheres to all semantic and syntactic rules that apply to it.

Example of a semantic rule: If the class is concrete, there should be methods to realize all its operations.

Example of a syntactic rule: A class is drawn as a solid-outline rectangle with three compartments separated by horizontal lines. The first compartment contain the name of the class. The second and third compartments contain attributes and operations belonging to the class. UML has semantic rules for:

- ❖ Names
- ❖ Scope
- ❖ Visibility
- ❖ Integrity
- ❖ Execution

Name: The names you can call for things, relationships, and diagrams. There should be identifiable and distinguished names, to identify elements such as class, object, association, state, process, inheritance, and final state etc.

Scope: It is the context that gives specific meaning to the element named. For example, the scope could be instance scope meaning the named element appears in every instance of the class with different values. If the scope is classifier scope, the element will have only one value across the class, i.e., for all instances.

Visibility: This specifies how an element can be seen and used by others. For example, the visibility could be public, meaning the element is accessible by everyone, private, meaning no outsider can access this element, protected, only the class and inherited classes can access.

Integrity: This identifies how the things are properly defined and how they are related to each other. Here the relationships among the things have to be consistent. The data they contain should be relevant and accurate.

Execution: What it means if the model is built and run. What it will convey if the dynamic model of the system is executed or simulated.

However, during iterative, incremental development it is expected that models will be incomplete and inconsistent. Because, models built during the development of any software intensive system, tend to evolve as requirements are gathered and understood. These models are understood by different participants (stake holders) in different ways at different times.



1.8 Common Mechanisms in the UML

Building an apartment complex is made simpler if it conforms to certain patterns of common features. For example, the model of front room, the kitchen room pattern, and the bathroom pattern or its look and feel are some common patterns with their features are to be considered when building a flat. Similarly, building models of software systems becomes simpler in UML by using some common mechanisms that apply consistently throughout the UML, a visual modeling language.

There are four types of common mechanisms supported by UML. They are as follows:

- ❖ Specifications.
- ❖ Adornments.
- ❖ Common Divisions.
- ❖ Extensibility Mechanisms.

Specifications

As we know that UML is a graphical language used to model software intensive systems. UML not only have the facility of creating the building blocks graphically, it also has provision for textual statements describing the syntax and semantics of those visual building blocks. UML graphical notation helps us in visualizing the system, whereas the specification state the system's details.

By using a specification, we are basically specifying something in detail so that the role and meaning of the thing being specified is more clear and concise. For example, we can give a class a detailed specification by defining a full set of attributes, operations, full signatures of operations, and behaviors. Then we will have a clear picture of the capabilities, responsibilities and limitations of that class. Specifications can be included in the class, or specified separately.

Example:

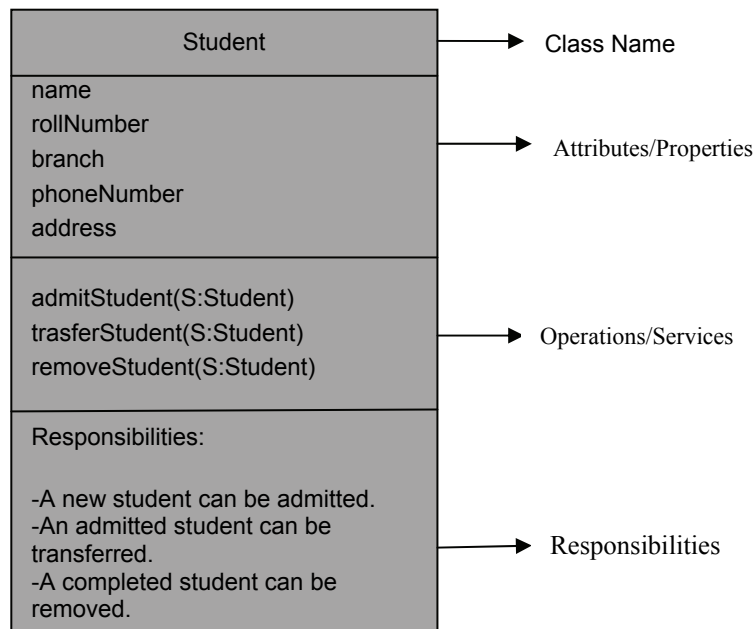


Fig.1.22 Specifying a Class

Adornments

Adornments are textual or graphical items, which can be added to the basic notation of a UML building block in order to visualize some details from its specification. For example, let us consider association, which in its most simple notation consists of one single line. Now, this can be adorned with some additional details, such as the role and the multiplicity of each end.

Example:

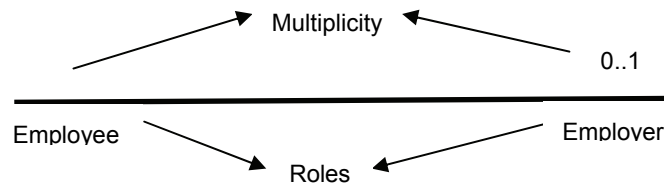


Fig 1.23 Specifying Adornments

Note: In the above association, there are two roles, one Employee and another Employer. An Employer can have more than one Employee working for him. An Employee either can work for an Employer or cannot work.

A class's specification may include details other than general information, such as whether it is abstract (A class cannot have instances) or the visibility of its attributes and operations. These details can be specified as graphical or textual adornments. The following example specify a class adorned to indicate that it is an abstract class with two public, one protected, and one private attributes.

Example:

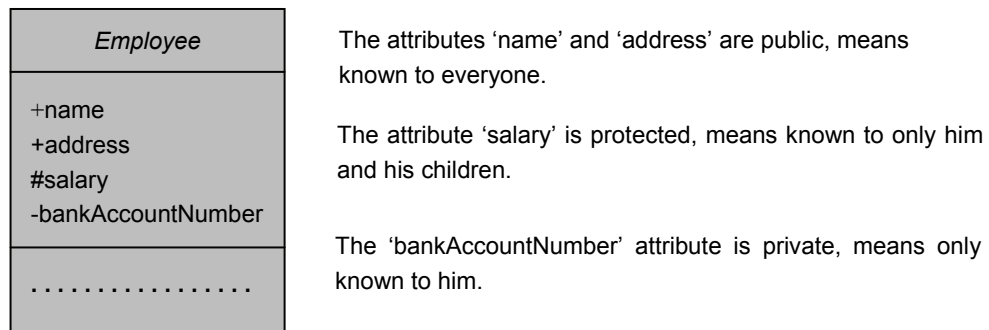
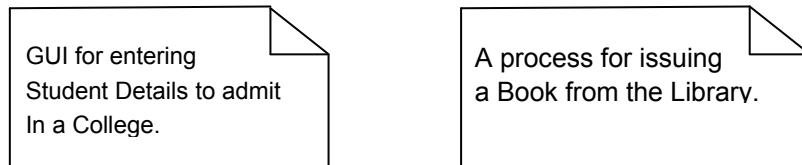


Fig. 1.24 Specifying Adornments in a Class

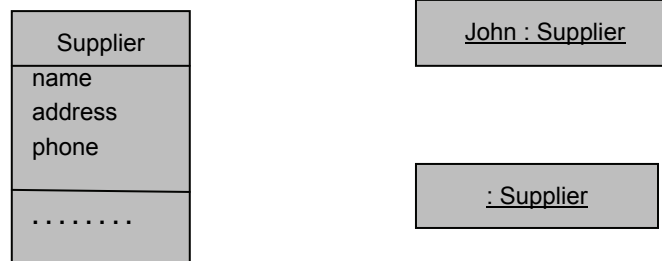
The most important kind of adornments is a Note. A Note is a graphical symbol used for adding constraints or comments to an element/group of elements. By using Notes, we can attach additional information to a model such as observations, requirements, and explanations. The contents specified in Notes do not change the meaning of the model to which it is attached.

Example:**Fig.1.25** Adornments in the form of Notes**Common Divisions**

These are used to distinguish between two things that might appear to be quite similar, or closely related to one another. There exist two main common divisions:

- ❖ Abstraction verses Manifestation.
- ❖ Interface verses Implementation.

In the former case, we mainly distinguish a class and an object. A class is an abstraction; and the object is the concrete manifestation of that class. Most UML building blocks have this kind of class/object distinction. For example, we have use cases and use case instances, components and component instances, and nodes and node instances.

Example:**Fig.1.26(a)** Supplier Class.**Fig.1.26(b)** Objects of Supplier.

Note: The top most object represents a supplier named John. In our subject terminology, John is an instance of the class, named Supplier. The second object represents a Supplier without knowing him. Such an object is called anonymous object. We use it only when we refer just an instance of the class without bothering about the details of that instance.

In the later case, we distinguish Interface and Implementation and how they are related. In this case, we address that an Interface declares some kind of contract or agreement, where as an implementation is a concrete realization of that contract. The Implementation is responsible for carrying out the interface.

Example:

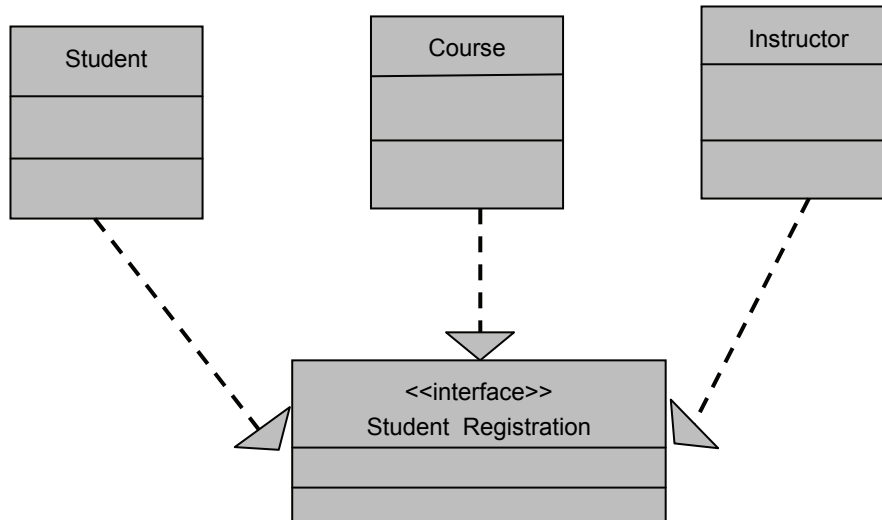


Fig.1.27 Implementing Interface

Extensibility Mechanisms

UML provides a standard graphical language for creating software blue prints, but not sufficient to express all possible things that happen across all models across all domains. Because of this constraint, UML is made open-ended, making you to extend the language capabilities. Extensibility mechanisms allow you to extend the UML by adding new building blocks, creating new properties and specifying new semantics in order to make the language suitable for modeling your specific problem domain. The UML's extensibility mechanisms include the following:

- ❖ Stereotypes
- ❖ Tagged Values
- ❖ Constraints

Stereotypes

They extend the vocabulary of the UML by creating new model elements derived from existing ones but that have specific properties suitable for your domain/problem. Each stereotype defines a set of properties that are received by elements of that stereotype. For Example:

1. If you are modeling a network oriented system, you definitely need symbols for routers and hubs. The standard UML is not having model elements to represent them. You can use stereotypes to model hubs and routers. We can make use of stereotyped nodes to model them so that they appear as primitive building blocks. The hub and router can be

graphically represented as follows. Graphically, a stereotype is rendered as a name enclosed by guillemets (« » or, if guillemets proper are unavailable, << >>) and placed above the name of another element.

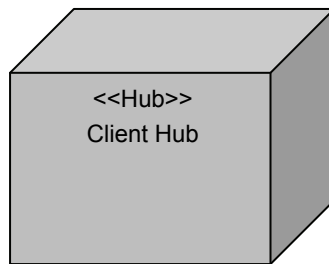


Fig. 1.28(a) Stereotype for Hub

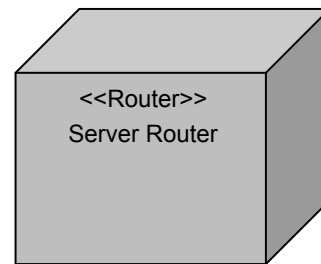


Fig.1.28(b) Stereotype for Router

- Another example, in java or in C++, you sometimes have to model exceptions as classes. You only want them to be thrown and caught. You can model them like basic building blocks, by marking them with a suitable stereotype.

Example:

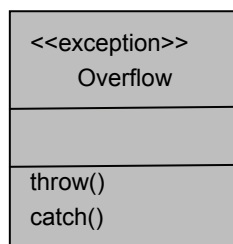


Fig. 1.29(a) Stereotype for exception

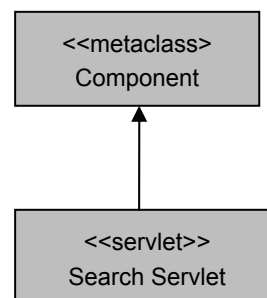
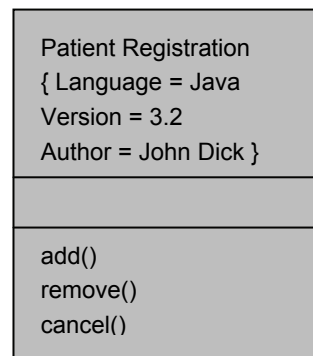


Fig. 1.29(b) Stereotypes for Component and Servlet

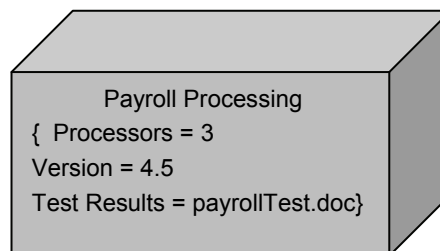
Tagged values

Tagged values are properties for specifying keyword-value pairs of model elements, where the keywords are attributes. They allow you to extend the properties of a UML building block so that you create new information in the specification of that element. Tagged values can be defined for existing model elements, or for individual stereotypes, so that everything with that stereotype has that tagged value. It is important to mention that a tagged value is not equal to a class attribute. Instead, you can regard a tagged value as being a metadata, since its value applies to the element itself and not to its instances. For example:

1. One of the most common uses of a tagged value is to specify properties that are relevant to code generation or configuration management. So, for example, you can make use of a tagged value in order to specify the programming language to which you map a particular class, or you can use it to denote the author and the version of a component. Graphically, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element. The string consists of a name (the tag), a separator (the symbol =), and a value (of the tag).

Example:**Fig.1.30** Using Tagged Values

2. As another example, where tagged values can be useful, consider the release team of a project, which is responsible for assembling, testing, and deploying releases. In such a case it might be feasible to keep track of the version number and test results for each main subsystem, and so one way of adding this information to the models is to use tagged values.

Example:**Fig.1.31** Using Tagged Values**Constraints**

Constraints are properties for specifying semantics and/or conditions that must be held true at all times for the elements of a model. They allow you to extend the semantics of a UML building block by adding new rules, or modifying

existing ones. For example, when modeling hard real time systems it could be useful to adorn the models with some additional information, such as time budgets and deadlines. By making use of constraints these timing requirements can easily be captured. Graphically, a constraint is rendered as a string enclosed by brackets, which is placed near the associated element(s), or connected to the element(s) by dotted lines. This notation can also be used to adorn a model element's basic notation, in order to visualize parts of an element's specification that have no graphical cue.

Example:

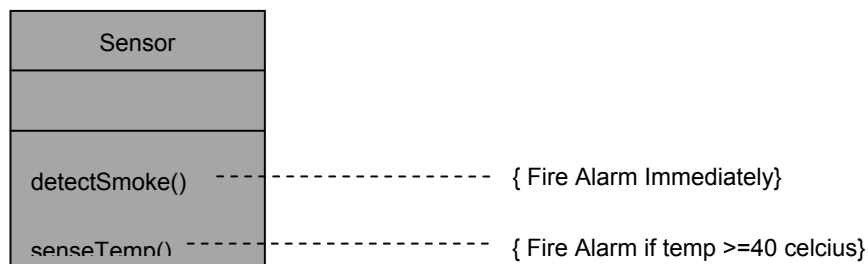


Fig.1.32 Specifying Constraints for Sensor

Note: In the above diagram specifies the details of a sensor class. Whenever sensor detects smoke, it immediately raises the fire alarm. If the sensor senses the temperature above 40 celcius, it raises the fire alarm.

In the following example, an engineering college takes admissions based on the ranks in the engineering entrance test.

Example:

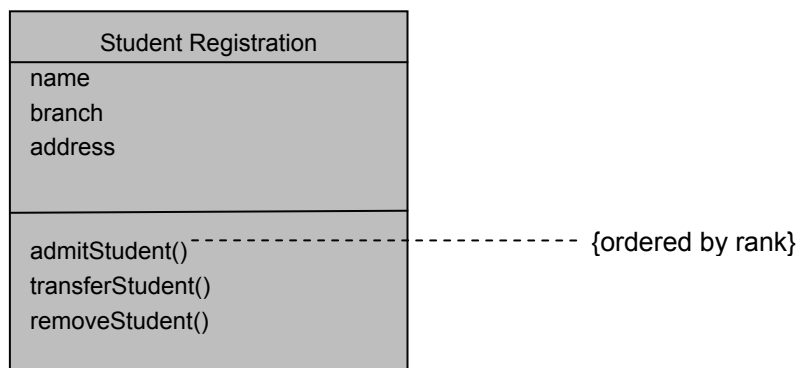


Fig.1.33 Student Admissions are in the order of the rank obtained



1.9 System Architecture

Any system models developed by UML demands that the system be viewed from different perspectives based on different stakeholders. The different stakeholders are end users, system analysts, developers, testers, system integrators, technical writers, project leaders and project managers and others who are directly or indirectly connected with the system. Each of these stakeholders look at the system in different ways at different situations.

Here, the system's architecture is the main way of realizing these different views and to control system development life cycle. The software architecture of a system is the set of static structures of the system needed to reason about the system, which comprise software elements, relations among them, and properties of both. The term system architecture also refers to documentation of a system's software architecture. Documenting software architecture facilitates understanding and communication between various stake holders of the software intensive system. An architecture documents early decisions about high-level design, and allows reuse of design components and design patterns between different software projects.

Technically, system architecture can be defined as understanding the components that make the system, how they work together and how they interact with each other, and the world around them. The exact definition of system architecture according to SEI (Software Engineering Institute) can be as follows:

“The architecture of a software-intensive system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.”

Architecture specification includes the decisions about the following:

- ❖ The organization of the system.
- ❖ The structural elements and their interfaces that the system is composed off.
- ❖ The behavior specified by behavioral elements.
- ❖ The composition of these structural and behavioral elements.

Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, and economic and technology constraints. The architecture of a software intensive system, can be best depicted by five related views as follows:

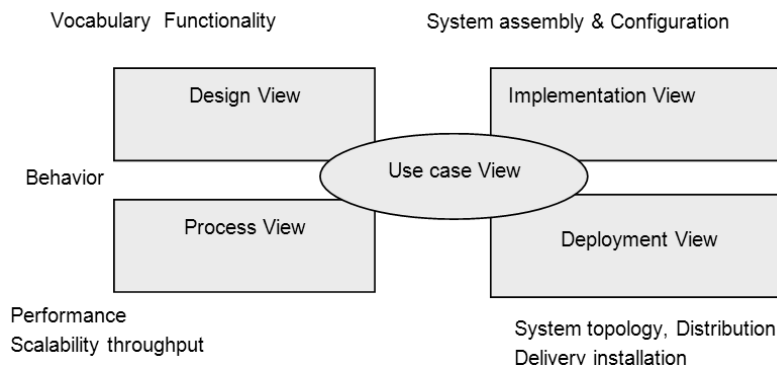


Fig.1.34 Five State View of a System

➤ 1.10 The Problem of Architectural Description

When you start the task of designing the architecture of a system, you will find that you have some questions to answer about the architecture. These could be as follows:

- ❖ What are the main functional elements of your system architecture?
- ❖ How will these elements interact with one another and with the outside world?
- ❖ What information will be managed, stored, and presented?
- ❖ What physical hardware and software elements will be required to support these functional and information elements?
- ❖ What operational features and capabilities will be provided?
- ❖ What development, test, support, and training environments will be provided?

A major role of an architect is to provide answers to these questions in a form that is understandable to the people who are concerned with the system. Most probably this can be achieved by creating an architectural description.

Definition: An architectural description (AD) is a set of artifacts that documents an architecture in a way its stakeholders can understand and demonstrates that the architecture has met their concerns.

Use case View

The purpose of this view is to see the system as a set of activities or transactions. It is a technique to capture business processes from end user's perspective. It expresses the system's behavior as seen by users, system analysts and testers. It gives the elements and various other sources that shape the architecture of the system. Use case diagrams are the ways and means of

expressing system requirements according to the client's perspective, and system functionalities according to the developers and testers perspective. The static aspect of this use case view is indicated by Use case Diagrams. The dynamic or the behavioral aspect of the system is expressed by interaction diagrams (state chart and activity diagrams).

Design View

This is the structural view of the system which gives an idea of what a given system is made of. This view encompasses classes, interfaces, and collaborations that define the vocabulary of a system. This view expresses functional requirements of the system. The static aspect of this view is captured in class and object diagrams. The dynamic aspect of this view is captured through interaction diagrams. This view extensively expresses the problem definition and the way the solution is built for the problem.

Process View

Through this view you can understand the behavior of a system. This view encompasses the threads and processes defining concurrency and synchronization mechanisms involved in the system. It not only addresses the processes that constitute your system, it also gives information about performance, scalability, and throughput of the system under consideration. This view includes various diagrams such as, the state diagram, activity diagram, sequence diagram, and collaboration diagram.

Implementation View

This view includes various components and files used to assemble and release the system for customer base. This view is the procedure depicting how the system is assembled from its components and files that establish a running system. It addresses the configuration management of the system's release. It gives the picture of grouped modules that constitute your system. The static aspects of this view are captured in component diagrams, and the dynamic aspects of it are captured through interaction diagrams, state chart diagrams and activity diagrams.

Deployment View

This is used to identify the deployment modules for a given system. This view encompasses the nodes that form the hardware topology on which the system executes. This view addresses the distribution, delivery, and installation of the parts that make up the physical system. You find the static aspects of this view , in deployment diagrams and dynamic aspects in interaction diagrams, state chart diagrams, and activity diagrams.



1.11 Software Development Life Cycle

The models developed by UML are process-independent, means that they do not depend on any particular software development methodology. To utilize UML for a greater extent, it is better to have processes that are use case driven, architecture centric, and iterative and incremental.

Use Case Driven

The desired behavior of the system is established by use cases. Use cases are used as primary source for verifying, and validating the system's architecture. Use cases are used as the major resources for establishing testing, and communication among various stakeholders of the system.

Architecture-Centric

The system's architecture is considered as the primary artifact. The system's architecture is taken as the basis for conceptualizing, managing, constructing and evolving the system under consideration.

Iterative and Incremental

Here, the iterative process refers to the management of the stream of executable releases. Incremental process refers to the continuous integration of system's architecture for the releases. Each new release will be the improvement of the previous release in terms of end user's perspective of the system in its functionality.

Any process with the above mentioned characteristics can be broken into phases. A phase is the span of time between two major milestones of the process. There are four phases in the software development life cycle, and they are Inception, Elaboration, Construction, and Transition.

Inception

This is the first phase in the software development process. It involves the basic idea of what to implement. The end of this phase begins the next phase, that is Elaboration.

Elaboration

It is the second phase of the software development process, which include the definition of the product vision and its architecture. In this phase, system's requirements are considered and formulated. This phase also specify functional or non functional behavior of the system. This phase also forms the basis for testing the system.

Construction

In this phase the system is made ready to be transferred to the user community. In this phase, the system's requirements, and its evaluation criteria are constantly examined against the business needs of the system.

Transition

In this phase, the system is fully tested and it is delivered to the end user. But this phase is not the end of the development process. In this phase still requirements are gathered, bugs are fixed, and new enhancements are taken from the user and they are implemented in the new incremental release of the system.



Essay Questions

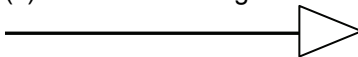
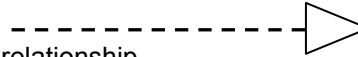
1. What is UML? Explain it briefly.
2. Give importance of modeling.
3. Explain principles of modeling.
4. What is object oriented modeling?
5. Briefly explain the following
 - (a) Things
 - (b) Relationships
6. Briefly explain the following
 - (a) Relationships
 - (b) Diagrams
7. Explain common mechanisms in UML
8. Explain extensibility mechanisms in UML.



Objective Type Questions

1. What is a model?
 - (a) Simplification of reality
 - (b) It is a blue print
 - (c) Both a & b
 - (d) None
2. A model is needed for
 - (a) Visualizing the system
 - (b) Specifying structure & behavior of a system
 - (c) Documenting the decisions we make
 - (d) All the above.
3. UML builds data model by using
 - (a) Class diagrams
 - (b) Object diagrams
 - (c) Both a & b
 - (d) None

4. The system's functionality can be specified by using
 - (a) Use case diagrams
 - (b) Activity diagrams
 - (c) Both a & b
 - (d) component diagrams
5. The best models are connected to reality is what principle of modeling
 - (a) Principle One
 - (b) Principle Two
 - (c) Principle Three
 - (d) Principle Four
6. The building blocks of UML are
 - (a) Things
 - (b) Relationships
 - (c) Diagrams
 - (d) All the above
7. Classes and interfaces come under the category of
 - (a) Behavioral things
 - (b) Structural things
 - (c) Annotational things
 - (d) Grouping things
8. Packages come under the category of
 - (a) Structural things
 - (b) Behavioral things
 - (c) Annotational things
 - (d) Grouping things
9. The set of objects that share common attributes, operations, relationships and semantics is called
 - (a) Interface
 - (b) Class
 - (c) Component
 - (d) Node
10. A collection of operations, which specify a service of a class or a component is
 - (a) Interface
 - (b) Class
 - (c) Node
 - (d) Component
11. A collaboration is graphically rendered as
 - (a) Solid ellipse
 - (b) Dotted ellipse
 - (c) Rectangle
 - (d) A tagged rectangle.
12. A physical and replaceable part of the system is called
 - (a) Class
 - (b) Object
 - (c) Component
 - (d) Use case
13. ----- represents a computational resource of a system
 - (a) class
 - (b) Object
 - (c) Component
 - (d) Node
14. The behavioural things are
 - (a) Interactions
 - (b) State machine
 - (c) Both
 - (d) None

15. A Note comes under the category of
- (a) Grouping Things
 - (b) Structural Things
 - (c) Behavioral things
 - (d) Annotational Things
16.  Is a graphical symbol for
- (a) Realization Relationship
 - (b) Generalization Relationship
 - (c) Dependency Relationship
 - (d) Association Relationship
17.  is a graphical symbol for ----- relationship
- (a) Association
 - (b) Dependency
 - (c) Composition
 - (d) Realization
18. ----- diagrams represent static design view of the system
- (a) Class Diagrams
 - (b) Object Diagrams
 - (c) Both a and b
 - (d) None
19. The diagram which emphasizes the time ordering of messages is
- (a) Collaboration diagram
 - (b) State chart
 - (c) Activity diagram
 - (d) Sequence diagram
20. State chart diagram contain
- (a) States
 - (b) Transitions
 - (c) Events and activities
 - (d) All the above
21. The diagram which indicate static implementation view of the system is
- (a) Activity diagram
 - (b) Deployment Diagram
 - (c) Component diagram
 - (d) Object diagram
22. These are the common mechanisms supported by UML
- (a) Specifications
 - (b) Adornments
 - (c) Common divisions
 - (d) All the above
23. The Extensibility mechanisms supported by UML are
- (a) Stereotypes
 - (b) tagged values
 - (c) constraints
 - (d) All the above
24. The view which encompasses the threads and processes is
- (a) design view
 - (b) process view
 - (c) use case view
 - (d) implementation view

25. It is the phase in which the system is fully tested
- (a) inception
 - (b) construction
 - (c) transition
 - (d) elaboration



Answers

- | | | | | | |
|---------|---------|---------|---------|---------|---------|
| 1. (c) | 2. (d) | 3. (c) | 4. (c) | 5. (c) | 6. (d) |
| 7. (b) | 8. (d) | 9. (b) | 10. (a) | 11. (b) | 12. (c) |
| 13. (d) | 14. (c) | 15. (d) | 16. (b) | 17. (d) | 18. (c) |
| 19. (d) | 20. (d) | 21. (c) | 22. (d) | 23. (d) | 24. (b) |
| 25. (c) | | | | | |