CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 1
Unit - 1
Year and Semester: IIIyear &II Semester
Subject**: Object Oriented Analysis and Design**
Branch: **CSE**
Faculty: **Dr** **,** Professor (CSE)

12IT302CV Object Oriented Analysis and Design

Instruction: 4 Periods/Week Credits: 4 Sessional Marks: 25
End Examination: 75 Marks End Exam Duration : 3 Hours

Unit I Introduction to UML – CO1: Importance of Modeling – CO1, Principles of Modeling – CO1, Object Oriented modeling – CO1, Conceptual Model of the UML – CO1, Architecture – CO1, Software Development Life Cycle – CO1.
Unit II Basic Structural Modeling - CO2: Classes – CO2, Relationships – CO2, Common Mechanisms – CO2, and diagrams – CO2.
Advanced Structural Modeling – CO2: Advanced Classes – CO2, advanced relationships CO2, interfaces, Types and Roles – CO2, Packages – CO2.
Unit III Class & Object Diagrams – CO1 & CO2: Terms - CO1 & CO2, concepts - CO1 & CO2, modeling techniques for Class & Object Diagrams - CO1 & CO2.
Unit IV Basic Behavioral Modeling-I – CO2: Interactions CO2, Interaction diagrams CO2.
Unit V Basic Behavioral Modeling-II - CO2 & CO3: Use cases - CO2 & CO3, Use case Diagrams - CO2 & CO3, Activity Diagrams - CO2 & CO3.
Unit VI Advanced Behavioral Modeling - CO2 & CO3: Events and signals - CO2 & CO3, state machines - CO2 & CO3, processes and Threads - CO2 & CO3, time and space - CO2 & CO3, state chart diagrams - CO2 & CO3.
Unit VII Architectural Modeling – CO4: Component – CO4, Deployment – CO4, Component Diagrams – CO4 and Deployment diagrams – CO4.
Unit VIII Case Study: The Unified Library application – CO5.

Text Books:

1. **The Unified Modeling Language User Guide, Ivar Jacobson and Grady Booch, James Rumbaugh, Pearson Education, 2009.**
2. **UML 2 Toolkit, Magnus Penker, Brian Lyons, David Fado and Hans-Erik Eriksson, Wiley-Dreamtech India Pvt.Ltd., 2004.**

References:

1. **Fundamentals of Object Oriented Design in UML, Meilir Page-Jones, Pearson Education, 2000.**
2. **Modeling Software Systems Using UML2, Pascal Roques, Wiley-Dreamtech India Pvt. Ltd., 2007.**
3. **Object Oriented Analysis & Design, Atul Kahate, 1st Edition, McGraw-Hill Companies, 2007.**

CVR COLLEGE OF ENGINEERING
An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 1
Unit - 1
Year and Semester: IIIyear &II Semester
Subject**: Object Oriented Analysis and Design**
Branch: **CSE**
Faculty: **Dr** **,** Professor (CSE)

12IT302CV Object Oriented Analysis and Design

Unit I

Unit I
Introduction to UML – CO1

Unified Modeling Language (UML) is a general purpose modelling language. The main aim of UML is define a standard way to **visualize** the way a system has been designed. It is quite similar to blueprints used in other fields of engineering.

UML is **not a programming language**, it is rather a visual language. We use UML diagrams to portray the **behavior and structure** of a system. UML helps software engineers, businessmen and system architects with modelling, design and analysis. The Object Management Group (OMG) adopted Unified Modelling Language as a standard in 1997. Its been managed by OMG ever since. International Organization for Standardization (ISO) published UML as an approved standard in 2005. UML has been revised over the years and is reviewed periodically.

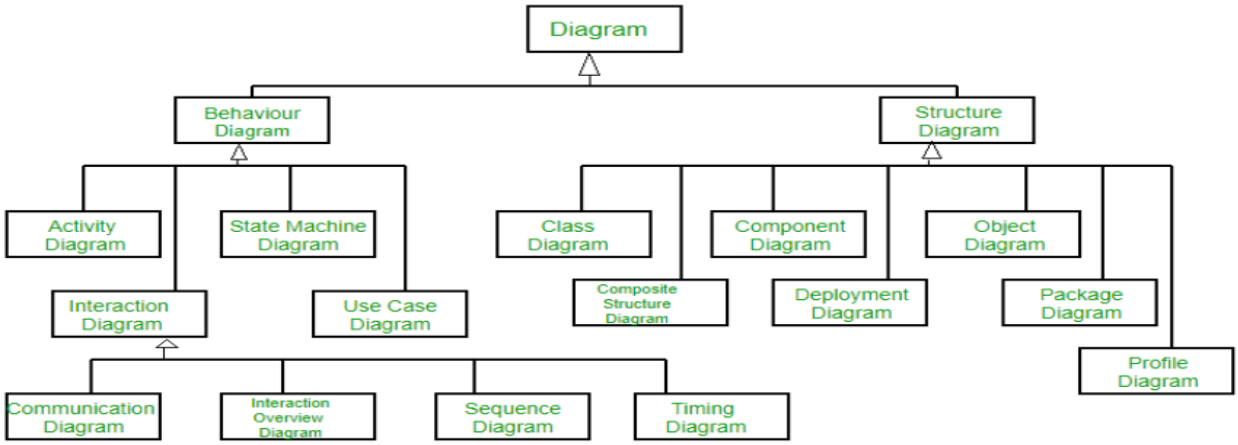
Uses of UML:

- Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them.
- Businessmen do not understand code. So UML becomes essential to communicate with non programmers essential requirements, functionalities and processes of the system.
- A lot of time is saved down the line when teams are able to visualize processes, user interactions and static structure of the system.

UML is linked with **object oriented** design and analysis. UML makes the use of elements and forms associations between them to form diagrams. Diagrams in UML can be broadly classified as:

1. **Structural Diagrams –** Capture static aspects or structure of a system. Structural Diagrams include: Component Diagrams, Object Diagrams, Class Diagrams and Deployment Diagrams.
2. **Behavior Diagrams –** Capture dynamic aspects or behavior of the system. Behavior diagrams include: Use Case Diagrams, State Diagrams, Activity Diagrams and Interaction Diagrams.

The image below shows the hierarchy of diagrams according to UML 2.2



Object Oriented Concepts Used in UML –

1. **Class –** A class defines the blue print i.e. structure and functions of an object.
2. **Objects –** Objects help us to decompose large systems and help us to modularize our system. Modularity helps to divide our system into understandable components so that we can build our system piece by piece. An object is the fundamental unit (building block) of a system which is used to depict an entity.
3. **Inheritance –** Inheritance is a mechanism by which child classes inherit the properties of their parent classes.
4. **Abstraction –** Mechanism by which implementation details are hidden from user.
5. **Encapsulation –** Binding data together and protecting it from the outer world is referred to as encapsulation.
6. **Polymorphism –** Mechanism by which functions or entities are able to exist in different forms.

Additions in UML 2.0 –

- Software development methodologies like agile have been incorporated and scope of original UML specification has been broadened.
- Originally UML specified 9 diagrams. UML 2.x has increased the number of diagrams from 9 to 13. The four diagrams that were added are : timing diagram, communication diagram, interaction overview diagram and composite structure diagram. UML 2.x renamed statechart diagrams to state machine diagrams.
- UML 2.x added the ability to decompose software system into components and sub-components.

Structural UML Diagrams –

1. **Class Diagram –** The most widely use UML diagram is the class diagram. It is the building block of all object oriented software systems. We use class diagrams to depict the static structure of a system by showing system's classes,their methods and attributes. Class diagrams also help us identify relationship between different classes or objects.
2. **Composite Structure Diagram –** We use composite structure diagrams to represent the internal structure of a class and its interaction points with other parts of the system. A composite structure diagram represents relationship between parts and their configuration which determine how the classifier (class, a component, or a deployment node) behaves. They represent internal structure of a structured classifier making the use of parts, ports, and connectors. We can also model collaborations using composite structure diagrams. They are similar to class diagrams except they represent individual parts in detail as compared to the entire class.

3. **Object Diagram –** An Object Diagram can be referred to as a screenshot of the instances in a system and the relationship that exists between them. Since object diagrams depict behaviour when objects have been instantiated, we are able to study the behaviour of the system at a particular instant. An object diagram is similar to a class diagram except it shows the instances of classes in the system. We depict actual classifiers and their relationships making the use of class diagrams. On the other hand, an Object Diagram represents specific instances of classes and relationships between them at a point of time.
4. **Component Diagram –** Component diagrams are used to represent the how the physical components in a system have been organized. We use them for modelling implementation details. Component Diagrams depict the structural relationship between software system elements and help us in understanding if functional requirements have been covered by planned development. Component Diagrams become essential to use when we design and build complex systems. Interfaces are used by components of the system to communicate with each other.
5. **Deployment Diagram –** Deployment Diagrams are used to represent system hardware and its software.It tells us what hardware components exist and what software components run on them.We illustrate system architecture as distribution of software artifacts over distributed targets. An artifact is the information that is generated by system software. They are primarily used when a software is being used, distributed or deployed over multiple machines with different configurations.
6. **Package Diagram –** We use Package Diagrams to depict how packages and their elements have been organized. A package diagram simply shows us the dependencies between different packages and internal composition of packages. Packages help us to organise UML diagrams into meaningful groups and make the diagram easy to understand. They are primarily used to organise class and use case diagrams.

Behavior Diagrams –

1. **State Machine Diagrams –** A state diagram is used to represent the condition of the system or part of the system at finite instances of time. It's a behavioral diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as **State machines** and**State-chart Diagrams** . These terms are often used interchangeably.So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli.
2. **Activity Diagrams –** We use Activity Diagrams to illustrate the flow of control in a system. We can also use an activity diagram to refer to the steps involved in the execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram.An activity diagram focuses on condition of flow and the sequence in which it happens. We describe or depict what causes a particular event using an activity diagram.
3. **Use Case Diagrams –** Use Case Diagrams are used to depict the functionality of a system or a part of a system. They are widely used to illustrate the functional requirements of the system and its interaction with external agents(actors). A use case is basically a diagram representing different scenarios where the system can be used. A use case diagram gives us a high level view of what the system or a part of the system does without going into implementation details.
4. **Sequence Diagram –** A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place.We can also use the terms

event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.

5. **Communication Diagram –** A Communication Diagram(known as Collaboration Diagram in UML 1.x) is used to show sequenced messages exchanged between objects. A communication diagram focuses primarily on objects and their relationships. We can represent similar information using Sequence diagrams,however, communication diagrams represent objects and links in a free form.
6. **Timing Diagram –** Timing Diagram are a special form of Sequence diagrams which are used to depict the behavior of objects over a time frame. We use them to show time and duration constraints which govern changes in states and behavior of objects.
7. **Interaction Overview Diagram –** An Interaction Overview Diagram models a sequence of actions and helps us simplify complex interactions into simpler occurrences. It is a mixture of activity and sequence diagrams.

Importance of Modeling – CO1

UML is a pictorial language used to make software blueprints. **UML** can be described as a general purpose visual **modeling** language to visualize, specify, construct, and document software system. Although **UML** is generally used to **model** software systems, it is not limited within this boundary.

The Unified Modeling Language (**UML**) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. ... It has been found that all the **UML** diagrams plays vital role in s/w development.

Research has found that **modeling** decreases student error, positively affects the perceived **importance** of a task and increases self-regulated learning. For effective **modeling**, teachers should use think-aloud to make **important** connections and share their expert thinking with their students.

Modeling serves multiple **purposes** in the software world, some of which are systems **modeling** for system determination (such as for real time system behavior) or scientific **modeling** used for better understanding of scientific phenomenon (which can also be used in real time systems to perform real time operations).

A UML class diagram is not only used to **describe** the object and information structures in an application, but also show the communication with its users. It provides a wide range of usages; from modeling the static view of an application to describing responsibilities for a system.

Modeling Benefits:

Manage Complexity.

Modeling is essential in complexity management. Modeling benefits include:

- Viewing systems from multiple perspectives
- Discovering causes and effects using model traceability
- Improving system understanding through visual analysis
- Discovering errors earlier and reducing system defects
- Exploring alternatives earlier in the system lifecycle
- Improving impact analysis, identifying potential consequences of a change, or estimating modifications to implement a change

- Simulating system solutions without code generation

Preserve Knowledge and Corporate Memory

Modeling helps enterprises preserve knowledge and corporate memory by:

- Storing the corporate memory in a versioned repository
- Enabling quick and easy understanding of our systems within an organization by all members of our teams
- Assisting new team members in getting up to speed quickly

Reuse

Modeling helps us reuse parts of existing information and knowledge in our new projects, saving time and money.

Automate

Modeling facilitates automation including these examples:

- Automate generation of a real working system or part of a system from models
- Automate repeatable tasks by writing scripts
- Use thousands of shortcuts and features for getting the expected results in a single click

The popularity of modeling is increasing. And with good reason, because modeling (especially based on standards) provides a means for communication, thinking and complexity management.

Modeling is essential for these major activities:

Sketches/Prototype Designs/Throw-away Models

Although these are the simplest forms of modeling activities, modeling is still much better than doing no modeling at all. Transform our words into pictures – they are better understood. Transform our ideas into a model – now they have structure. Frankly, in most of these cases, paper or a whiteboard will substitute for a modeling tool. If we still want software for this task, we can even consider a drawing tool. The major requirements for a tool are **Fast and Inexpensive.**

Code Engineering

The architect's dream is to model the system and then … launch it. However, the reality is not so glamorous. Usually modeling tools generate skeletons of code, saving some time, but on the other hand consuming time for creating code-level class models. There is a reason for such behavior – models usually do not have enough information to produce full code, and it would take a lot of time if we wanted to model all those little details.

Do not use code generation. Concentrate on higher-level architecture and design, and leave code to coders.

Use MDA (Model Driven Architecture) tools. These tools can (in combination with modeling tools) generate 60-100% of our system – code, persistence, etc. However, they have to be chosen wisely, since they provide the most value for repetitive/machine-like tasks. For example, if we are modeling a mobile application which slightly differs from platform to platform, then it makes sense to generate code for different platforms from a single source – the model. This way we save the money for maintaining two, three or even more similar code bases which are still different.

Code reverse engineering into a model is a different story. This is good for quick code reviews – reverse, visualize, review, throw away. There is no need to maintain those models, since it should be easy to reverse engineer them from the latest version of our code base.

Creating Complex Systems

Creating a complex system is a complex task. There are certain qualities in a modeling tool that can make this task easier. The primary goal for a modeling tool is to create a collaborative environment that allows users to create and validate design ideas, prevent errors early, and communicate those

ideas/designs to those who will further refine them. Let's break down the benefits we should expect from a modeling tool for production of complex systems.

Robustness and ease of use. If it takes more time to manage the tool than to do the actual work – we need a different tool.

Standards-compliant. Communication in producing a complex system is a very important component; therefore, we need a modeling solution that:

Is based on standards, since learning a proprietary methodology is expensive and unnecessary.

Forces to create standards-compliant models. Creating non-standard models introduces the risk of interpreting them differently. Ambiguity leads to communication problems.

Centralized repository. We really do not want to look for the latest version of our models somewhere on the network. And if we are working with an older version, it will be difficult to incorporate changes to the latest one.

Teamwork. Complex systems require large teams, so our selected tool has to allow for multiple people working on the same model, preventing conflicts while allowing model branches. Plus, we might not want a team developer to have the ability to modify high-level architecture.

Analysis/validation functions. The larger the model gets, the easier it is to miss things. We want the tool to:

Provide multiple views of the model, allowing users to study the model from different perspectives.

Automatically validate the model, allowing the user to identify obvious model gaps or inconsistencies.

Document output. Unfortunately, there is a lot of paperwork circulating in the IT world. However, a carefully selected modeling tool should do it for us – the information is in the model, we just need to "transform" it to a paper format.

Simulation. For really complex systems, mission-critical systems, or hardware systems, errors that could be prevented in the design/analysis phase are very expensive in the production phase. The best way to make sure that the system works as described in the model is to simulate it.

Support for all types of related models. The last thing we want to have is multiple types of models, created by different modeling tools. That means we will struggle with traceability, synchronization of data, etc.

Multiplatform. If our company is not strict about the operating system (Windows/MacOS/Linux/etc.), our selection of a modeling tool should be multiplatform as well.

Creating Integrated Models for our Business

If we want to improve our business, we need to know three things: **Current situation, Future (improved) situation and How to get there.**

So it starts with knowing the current situation:

Traceability. In the case of business processes, we need to know who is responsible for this task, why this task is performed, and how it is automated by the IT system.

Integrated models. Businesses involve different types of models such as organizational charts, business processes, etc. We need those models to be tightly integrated.

Ability to customize. Every business is different, so the modeling solution which we select needs to take that into account. Our modeling solution should be able to adapt, to match the specific details that make up our business unique.

Web output. Information that is in the model is important to all the people in our company. Our colleagues should not need a special tool to access the models.

Principles of Modeling – CO1

The use of modeling has a rich history in all the engineering disciplines. That experience suggests four basic principles of modeling. First,

The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.

In other words, choosing our models well. The right models will brilliantly illuminate the most wicked development problems, offering insight that we simply could not gain otherwise; the wrong models will mislead us, causing us to focus on irrelevant issues.

Setting aside software for a moment, suppose we are trying to tackle a problem in quantum physics. Certain problems, such as the interaction of photons in space-time, are full of wonderfully hairy mathematics. Choose a different model and suddenly this inherent complexity becomes doable, if not exactly easy. In this field, this is precisely the value of Feynmann diagrams, which provide a graphical rendering of a very complex problem. Similarly, in a totally different domain, suppose we are constructing a new building and we are concerned about how it might behave in high winds. If we build a physical model and then subject it to wind tunnel tests, we might learn some interesting things, although materials in the small don't flex exactly as they do in the large. Hence, if we build a mathematical model and then subject it to simulations, we will learn some different things, and we will also probably be able to play with more new scenarios than if we were using a physical model. By rigorously and continuously testing our models, we'll end up with a far higher level of confidence that the system we have modeled will behave as we expect it to in the real world.

In software, the models we choose can greatly affect our world view. If we build a system through the eyes of a database developer, we will likely focus on entity-relationship models that push behavior into triggers and stored procedures. If we build a system through the eyes of a structured analyst, we will likely end up with models that are algorithmic-centric, with data flowing from process to process. If we build a system through the eyes of an object-oriented developer, we'll end up with a system whose architecture is centered around a sea of classes and the patterns of interaction that direct how those classes work together. Executable models can greatly help testing. Any of these approaches might be right for a given application and development culture, although experience suggests that the object-oriented view is superior in crafting resilient architectures, even for systems that might have a large database or computational element. That fact notwithstanding, the point is that each world view leads to a different kind of system, with different costs and benefits.

Second,

Every model may be expressed at different levels of precision.

If we are building a high rise, sometimes we need a 30,000-foot view for instance, to help our investors visualize its look and feel. Other times, we need to get down to the level of the studs for instance, when there's a tricky pipe run or an unusual structural element.

The same is true with software models. Sometimes a quick and simple executable model of the user interface is exactly what we need; at other times we have to get down and dirty with the bits, such as when we are specifying cross-system interfaces or wrestling with networking bottlenecks. In any case, the best kinds of models are those that let us choose our degree of detail, depending on who is doing the viewing and why they need to view it. An analyst or an end user will want to focus on issues of

what; a developer will want to focus on issues of how. Both of these stakeholders will want to visualize a system at different levels of detail at different times.

Third,

The best models are connected to reality.

A physical model of a building that doesn't respond in the same way as do real materials has only limited value; a mathematical model of an aircraft that assumes only ideal conditions and perfect manufacturing can mask some potentially fatal characteristics of the real aircraft. It's best to have models that have a clear connection to reality, and where that connection is weak, to know exactly how those models are divorced from the real world. All models simplify reality; the trick is to be sure that our simplifications don't mask any important details.

In software, the Achilles heel of structured analysis techniques is the fact that there is a basic disconnect between its analysis model and the system's design model. Failing to bridge this chasm causes the system as conceived and the system as built to diverge over time. In object-oriented systems, it is possible to connect all the nearly independent views of a system into one semantic whole.

Fourth,

No single model or view is sufficient. Every nontrivial system is best approached through a small set of nearly independent models with multiple viewpoints.

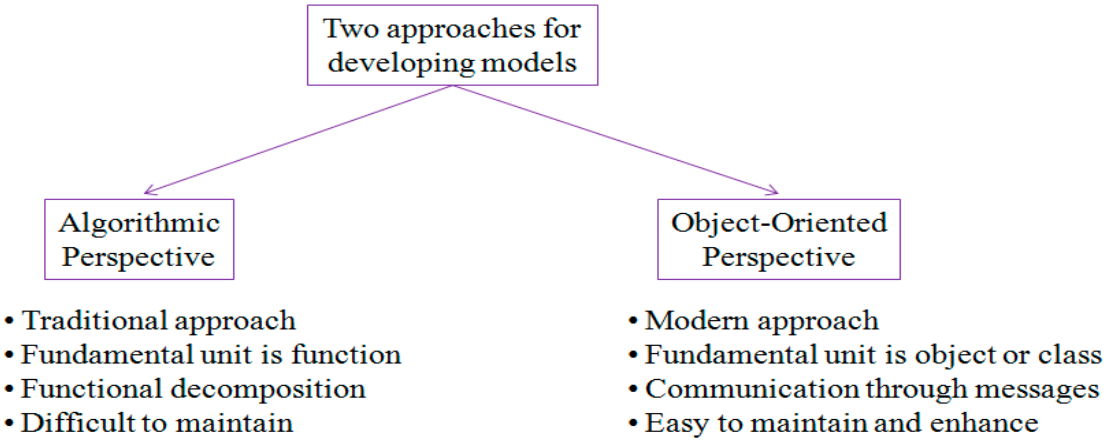
If we are constructing a building, there is no single set of blueprints that reveal all its details. At the very least, we'll need floor plans, elevations, electrical plans, heating plans, and plumbing plans. And within any kind of model, we need multiple views to capture the breadth of the system, such as blueprints of different floors.

The operative phrase here is "nearly independent." In this context, it means having models that can be built and studied separately but that are still interrelated. As in the case of a building, we can study electrical plans in isolation, but we can also see how they map to the floor plan and perhaps even their interaction with the routing of pipes in the plumbing plan.

The same is true of object-oriented software systems. To understand the architecture of such a system, we need several complementary and interlocking views: a use case view (exposing the requirements of the system), a design view (capturing the vocabulary of the problem space and the solution space), an interaction view (showing the interactions among the parts of the system and between the system and the environment), an implementation view (addressing the physical realization of the system), and a deployment view (focusing on system engineering issues). Each of these views may have structural, as well as behavioral, aspects. Together, these views represent the blueprints of software.

Object Oriented modeling – CO1

- **Two most common ways in modeling software systems are**
    - **Algorithmic - Procedures or functions**
    - **Object oriented - Objects or classes**



Object-oriented modeling (OOM) is the construction of objects using a collection of objects that contain stored values of the instance variables found within an object. Unlike models that are record-oriented, object-oriented values are solely objects.

The object-oriented modeling approach creates the union of the application and database development and transforms it into a unified data model and language environment. Object-oriented modeling allows for object identification and communication while supporting data abstraction, inheritance and encapsulation.

Object-oriented modeling is the process of preparing and designing what the model's code will actually look like. During the construction or programming phase, the modeling techniques are implemented by using a language that supports the object-oriented programming model.

OOM consists of progressively developing object representation through three phases: analysis, design, and implementation. During the initial stages of development, the model developed is abstract because the external details of the system are the central focus. The model becomes more and more detailed as it evolves, while the central focus shifts toward understanding how the system will be constructed and how it should function.

Object oriented modeling is entirely a new way of thinking about problems. This methodology is all about visualizing the things by using models organized around real world concepts. Object oriented models help in understanding problems, communicating with experts from a distance, modeling enterprises, and designing programs and database. We all can agree that developing a model for a software system, prior to its development or transformation, is as essential as having a blueprint for large building essential for its construction. Object oriented models are represented by diagrams. A good model always helps communication among project teams, and to assure architectural soundness. It is important to note that with the increasing complexity of systems, importance of modeling techniques increases. Because of its characteristics Object Oriented Modeling is a suitable modeling technique for handling a complex system. OOM basically is building a model of an application, which includes implementation details of the system, during design of the system.

As we know, any system development refers to the initial portion of the software life cycle: analysis, design, and implementation. During object oriented modeling identification and organization of application with respect to its domain is done, rather than their final representation in any specific programming language. We can say that OOM is not language specific. Once modeling is done for an application, it can be implemented in any suitable programming language available.

OOM approach is a encouraging approach in which software developers have to think in terms of the application domain through most of the software engineering life cycle. In this process, the developer is forced to identify the inherent concepts of the application. First, developer organize, and understood

the system properly and then finally the details of data structure and functions are addressed effectively.

In OOM the modeling passes through the following processes: • System Analysis • System Design • Object Design, and • Final Implementation. System Analysis: In this stage a statement of the problem is formulated and a model is build by the analyst in encouraging real-world situation. This phase show the important properties associated with the situation. Actually, the analysis model is a concise, precise abstraction and agreement on how the desired system must be developed. We can say that, here the objective is to provide a model that can be understood and criticized by any application experts in the area whether the expert is a programmer or not. System Design: At this stage, the complete system architecture is designed. This is the stage where the whole system is divided into subsystems, based on both the system analysis model and the proposed architecture of the system. Object Design: At this stage, a design model is developed based on the analysis model which is already developed in the earlier phase of development. The object design decides the data structures and algorithms needed to implement each of the classes in the system with the help of implementation details given in the analysis model. Final Implementation: At this stage, the final implementation of classes and relationships developed during object design takes place a particular programming language, database, or hardware implementation (if needed). Actual implementation should be done using software engineering practice. This helps to develop a flexible and extensible system. Whole object oriented modeling is covered by using three kinds of models for a system description. These models are: • object model, • dynamic model, and • functional model. Object models are used for describing the objects in the system and their relationship among each other in the system. The dynamic model describes interaction among objects and information flow in the system. The data transformations in the system are described by a functional model. All three models are applicable during all stages of development. These models bear the responsibility of acquiring implementation details of the system development. It is important to note that we cannot describe a system completely until unless all three modes are described properly. In block 3 of this course, we will discuss these three models in detail. Before we discuss the characteristics of object oriented modeling, let us see how object oriented development is different from structured development of the system. In the structured approach, the main emphasis is on specifying and decomposing system functionality. Structured approach is seen as the most direct way of implementing a desired goal. A structured approach has certain basic problems, such as, if the requirements of system change then a system based on decomposing functionality may require massive restructuring, and, the system gradually become unmanageable. In contrast to the structured approach, the basic focus of object oriented approach is to identify objects from the application domain, and then to associate procedures (methods) around these identified objects.

We can say that object oriented development is an indirect way of system development because in this approach a holistic view of application domain is considered, and objects are identified in the related problem domain. A historic view of application helps in realizing the situations and characteristics of the system. Taking a holistic view of the problem domain rather than considering functional requirements of a single problem give an edge to object oriented development. Once the objects are created with the needed characteristics, they communicate with each other by message passing during problem solving.

Conceptual Model of the UML – CO1

To understand the UML, we need to form a conceptual model of the language, and this requires learning three major elements: the UML's basic building blocks, the rules that dictate how those building blocks may be put together, and some common mechanisms that apply throughout the UML.

Once we have grasped these ideas, we will be able to read UML models and create some basic ones. As we gain more experience in applying the UML, we can build on this conceptual model, using more advanced features of the language.

Building Blocks of the UML

The vocabulary of the UML encompasses three kinds of building blocks:

1. Things 2. Relationships 3. Diagrams

Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.

Things in the UML

There are four kinds of things in the UML:

1. Structural things 2. Behavioral things 3. Grouping things 4. Annotational things

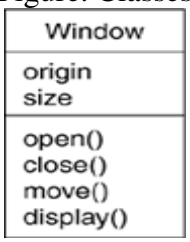
These things are the basic object-oriented building blocks of the UML. We use them to write well-formed models.

Structural Things

Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. Collectively, the structural things are called *classifiers*.

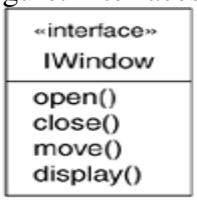
A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations, as in Figure.

Figure. Classes



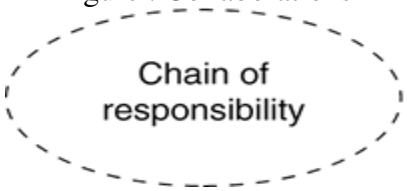
An *interface* is a collection of operations that specify a service of a class or component. An interface therefore describes the externally visible behavior of that element. An interface might represent the complete behavior of a class or component or only a part of that behavior. An interface defines a set of operation specifications (that is, their signatures) but never a set of operation implementations. The declaration of an interface looks like a class with the keyword «interface» above the name; attributes are not relevant, except sometimes to show constants. An interface rarely stands alone, however. An interface provided by a class to the outside world is shown as a small circle attached to the class box by a line. An interface required by a class from some other class is shown as a small semicircle attached to the class box by a line, as in Figure.

Figure. Interfaces



A *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. Collaborations have structural, as well as behavioral, dimensions. A given class or object might participate in several collaborations. These collaborations therefore represent the implementation of patterns that make up a system. Graphically, a collaboration is rendered as an ellipse with dashed lines, sometimes including only its name, as in Figure.

Figure . Collaborations



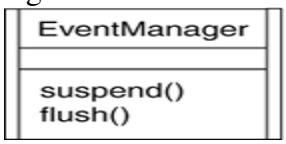
A *use case* is a description of sequences of actions that a system performs that yield observable results of value to a particular actor. A use case is used to structure the behavioral things in a model. A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name, as in Figure.

Figure. Use Cases

The remaining three things active classes, components, and nodes are all class-like, meaning they also describe sets of entities that share the same attributes, operations, relationships, and semantics. However, these three are different enough and are necessary for modeling certain aspects of an object-oriented system, so they warrant special treatment.

An *active class* is a class whose objects own one or more processes or threads and therefore can initiate control activity. An 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 as a class with double lines on the left and right; it usually includes its name, attributes, and operations, as in Figure.

Figure. Active Classes



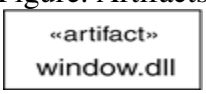
A component is a modular part of the system design that hides its implementation behind a set of external interfaces. Within a system, components sharing the same interfaces can be substituted while preserving the same logical behavior. The implementation of a component can be expressed by wiring together parts and connectors; the parts can include smaller components. Graphically, a component is rendered like a class with a special icon in the upper right corner, as in Figure.

Figure. Components

The remaining two elements artifacts and nodes are also different. They represent physical things, whereas the previous five things represent conceptual or logical things.

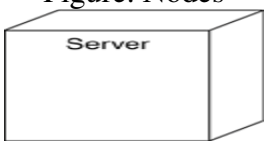
An *artifact* is a physical and replaceable part of a system that contains physical information ("bits"). In a system, we'll encounter different kinds of deployment artifacts, such as source code files, executables, and scripts. An artifact typically represents the physical packaging of source or run-time information. Graphically, an artifact is rendered as a rectangle with the keyword «artifact» above the name, as in Figure.

Figure. Artifacts



A *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. A set of components may reside on a node and may also migrate from node to node. Graphically, a node is rendered as a cube, usually including only its name, as in Figure.

Figure. Nodes



These elements classes, interfaces, collaborations, use cases, active classes, components, artifacts, and nodes are the basic structural things that we may include in a UML model. There are also variations on these, such as actors, signals, and utilities (kinds of classes); processes and threads (kinds of active classes); and applications, documents, files, libraries, pages, and tables (kinds of artifacts).

Behavioral Things

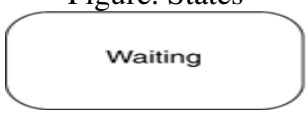
Behavioral things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are three primary kinds of behavioral things.

First, an *interaction* is a behavior that comprises a set of messages exchanged among a set of objects or roles within a particular context to accomplish a specific purpose. The behavior of a society of objects or of an individual operation may be specified with an interaction. An interaction involves a number of other elements, including messages, actions, and connectors (the connection between objects). Graphically, a message is rendered as a directed line, almost always including the name of its operation, as in Figure.

Figure. Messages

Second, a *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. The behavior of an individual class or a collaboration of classes may be specified with a state machine. A state machine involves a number of other elements, including states, transitions (the flow from state to state), events (things that trigger a transition), and activities (the response to a transition). Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates, if any, as in Figure.

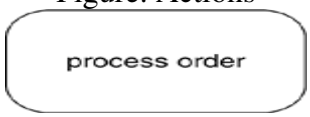
Figure. States



Third, an activity is a behavior that specifies the sequence of steps a computational process performs.

In an interaction, the focus is on the set of objects that interact. In a state machine, the focus is on the life cycle of one object at a time. In an activity, the focus is on the flows among steps without regard to which object performs each step. A step of an activity is called an *action*. Graphically, an action is rendered as a rounded rectangle with a name indicating its purpose. States and actions are distinguished by their different contexts.

Figure. Actions



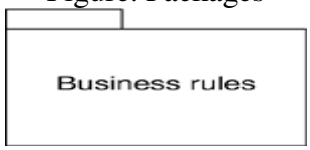
These three elements interactions, state machines, and activities are the basic behavioral things that we may include in a UML model. Semantically, these elements are usually connected to various structural elements, primarily classes, collaborations, and objects.

Grouping Things

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

A *package* is a general-purpose mechanism for organizing the design itself, as opposed to classes, which organize implementation constructs. Structural things, behavioral things, and even other grouping things may be placed in a package. Unlike components (which exist at run time), a package is purely conceptual (meaning that it exists only at development time). Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents, as in Figure.

Figure. Packages

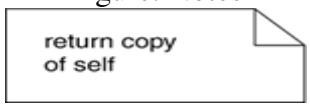


Packages are the basic grouping things with which we may organize a UML model. There are also variations, such as frameworks, models, and subsystems (kinds of packages).

Annotational Things

Annotational things are the explanatory parts of UML models. These are the comments we may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of annotational thing, called a note. A *note* is simply a symbol for rendering constraints and comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment, as in Figure.

Figure. Notes



This element is the one basic annotational thing we may include in a UML model. We'll typically use notes to adorn our diagrams with constraints or comments that are best expressed in informal or formal text. There are also variations on this element, such as requirements (which specify some desired behavior from the perspective of outside the model).

Relationships in the UML

There are four kinds of relationships in the UML:

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

These relationships are the basic relational building blocks of the UML. We use them to write well-formed models.

First, a *dependency* is a semantic relationship between two model elements in which a change to one element (the independent one) may affect the semantics of the other element (the dependent one). Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label, as in Figure.

Figure. Dependencies

Second, an *association* is a structural relationship among classes that describes a set of links, a link being a connection among objects that are instances of the classes. Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and end names, as in Figure

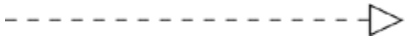
Figure. Associations

Third, a *generalization* is a specialization/generalization relationship in which the specialized element (the child) builds on the specification of the generalized element (the parent). The child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent, as in Figure.

Figure. Generalizations

Fourth, a *realization* is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. We'll encounter realization relationships in two places: between interfaces and the classes or components that realize them, and between use cases and the collaborations that realize them. Graphically, a realization relationship is rendered as a cross between a generalization and a dependency relationship, as in Figure.

Figure. Realizations



These four elements are the basic relational things we may include in a UML model. There are also variations on these four, such as refinement, trace, include, and extend.

Diagrams in the UML

A *diagram* is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and paths (relationships). We draw diagrams to visualize a system from different perspectives, so a diagram is a projection into a system. For all but the most trivial systems, a diagram represents an elided view of the elements that make up a system. The same element may appear in all diagrams, only a few diagrams (the most common case), or in no diagrams at all (a very rare case). In theory, a diagram may contain any combination of things and relationships. In practice, however, a small number of common combinations arise, which are consistent with the five most useful views that comprise the architecture of a software-intensive system. For this reason, the UML includes thirteen kinds of diagrams:

1. Class diagram 2. Object diagram 3. Component diagram 4. Composite structure diagram
5. Use case diagram 6. Sequence diagram 7. Communication diagram 8. State diagram
9. Activity diagram 10. Deployment diagram 11. Package diagram 12. Timing diagram
12. Interaction overview diagram

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

An *object diagram* shows a set of objects and their relationships. 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 as do class diagrams, but from the perspective of real or prototypical cases.

A *component diagram* is shows an encapsulated class and its interfaces, ports, and internal structure consisting of nested components and connectors. Component diagrams address the static design implementation view of a system. They are important for building large systems from smaller parts. (UML distinguishes a *composite structure diagram*, applicable to any class, from a component diagram, but we combine the discussion because the distinction between a component and a structured class is unnecessarily subtle.)

A *use case diagram* shows a set of use cases and actors (a special kind of class) 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.

Both sequence diagrams and communication diagrams are kinds of interaction diagrams. An *interaction diagram* shows an interaction, consisting of a set of objects or roles, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system. A *sequence diagram* is an interaction diagram that emphasizes the time-ordering of messages; a *communication diagram* is an interaction diagram that emphasizes the structural organization of the objects or roles that send and receive messages. Sequence diagrams and communication diagrams represent similar basic concepts, but each diagram emphasizes a different view of the concepts. Sequence diagrams emphasize temporal ordering, and communication diagrams emphasize the data structure through which messages flow. A *timing diagram* (not covered in this book) shows the actual times at which messages are exchanged.

A *state diagram* shows a state machine, consisting of states, transitions, events, and activities. A state diagrams shows the dynamic view of an object. They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems

An *activity diagram* shows the structure of a process or other computation as the flow of control and data from step to step within the computation. 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.

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. A node typically hosts one or more artifacts.

An *artifact diagram* shows the physical constituents of a system on the computer. Artifacts include files, databases, and similar physical collections of bits. Artifacts are often used in conjunction with deployment diagrams. Artifacts also show the classes and components that they implement. (UML treats artifact diagrams as a variety of deployment diagram, but we discuss them separately.)

A *package diagram* shows the decomposition of the model itself into organization units and their dependencies.

A *timing diagram* is an interaction diagram that shows actual times across different objects or roles, as opposed to just relative sequences of messages. An *interaction overview diagram* is a hybrid of an activity diagram and a sequence diagram. These diagrams have specialized uses and so are not discussed in this book. See the *UML Reference Manual* for more details.

This is not a closed list of diagrams. Tools may use the UML to provide other kinds of diagrams, although these are the most common ones that we will encounter in practice.

Rules of the UML

The UML's building blocks can't simply be thrown together in a random fashion. Like any language, the UML has a number of rules that specify what a well-formed model should look like. A *well-formed model* is one that is semantically self-consistent and in harmony with all its related models.

The UML has syntactic and semantic rules for

- Names What we can call things, relationships, and diagrams
- Scope The context that gives specific meaning to a name
- Visibility How those names can be seen and used by others
- Integrity How things properly and consistently relate to one another
- 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

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

These less-than-well-formed models are unavoidable as the details of a system unfold and churn during the software development life cycle. The rules of the UML encourage us but do not force us to address the most important analysis, design, and implementation questions that push such models to become well-formed over time.

Common Mechanisms in the UML

A building is made simpler and more harmonious by the conformance to a pattern of common features. A house may be built in the Victorian or French country style largely by using certain architectural patterns that define those styles. The same is true of the UML. It 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

Specifications

The UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block. For example, behind a class icon is a specification that provides the full set of attributes, operations (including their full signatures), and behaviors that the class embodies; visually, that class icon might only show a small part of this specification. Furthermore, there might be another view of that class that presents a completely different set of parts yet is still consistent with the class's underlying specification. We use the UML's graphical notation to visualize a system; we use the UML's specification to state the system's details. Given this split, it's possible to build up a model incrementally by drawing diagrams and then adding semantics to the model's specifications, or directly by creating a specification, perhaps by reverse engineering an existing system, and then creating diagrams that are projections into those specifications.

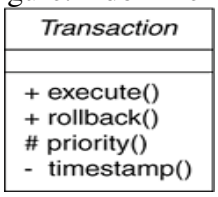
The UML's specifications provide a semantic backplane that contains all the parts of all the models of a system, each part related to one another in a consistent fashion. The UML's diagrams are thus simply visual projections into that backplane, each diagram revealing a specific interesting aspect of the system.

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. For example, the notation for a class is intentionally designed to be easy to draw, because classes are the most common element found in modeling object-oriented systems. The class notation also exposes the most important aspects of a class, namely its name, attributes, and operations.

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. For example, Figure shows a class, adorned to indicate that it is an abstract class with two public, one protected, and one private operation.

Figure. Adornments



Every element in the UML's notation starts with a basic symbol, to which can be added a variety of adornments specific to that symbol.

Common Divisions

In modeling object-oriented systems, the world often gets divided in several ways.

First, there is the division of class and object. A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, we can model classes as well as objects, as shown in Figure. Graphically, the UML distinguishes an object by using the same symbol as its class and then simply underlying the object's name.

Figure. Classes and Objects

In this figure, there is one class, named `Customer`, together with three objects: `Jan` (which is marked explicitly as being a `Customer` object), `:Customer` (an anonymous `Customer` object), and `Elyse` (which in its specification is marked as being a kind of `Customer` object, although it's not shown explicitly here).

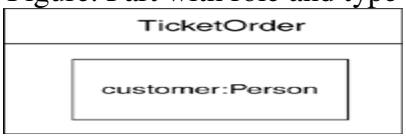
Almost every building block in the UML has this same kind of class/object dichotomy. For example, we can have use cases and use case executions, components and component instances, nodes and node instances, and so on.

Second, there is the separation of interface and implementation. An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics. In the UML, we can model both interfaces and their implementations, as shown in Figure.

Figure. Interfaces and Implementations

In this figure, there is one component named **SpellingWizard.dll** that provides (implements) two interfaces, IUnknown and ISpelling. It also requires an interface, IDictionary, that must be provided by another component.
Almost every building block in the UML has this same kind of interface/implementation dichotomy. For example, we can have use cases and the collaborations that realize them, as well as operations and the methods that implement them.
Third, there is the separation of type and role. The type declares the class of an entity, such as an object, an attribute, or a parameter. A role describes the meaning of an entity within its context, such as a class, component, or collaboration. Any entity that forms part of the structure of another entity, such as an attribute, has both characteristics: It derives some of its meaning from its inherent type and some of its meaning from its role within its context (Figure).

Figure. Part with role and type



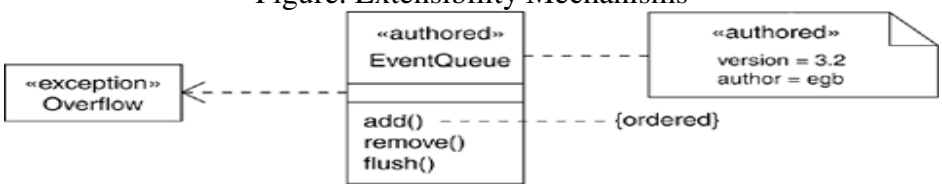
Extensibility Mechanisms

The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time. For this reason, the UML is opened-ended, making it possible for us to extend the language in controlled ways. The UML's extensibility mechanisms include

- Stereotypes
- Tagged values
- Constraints

A *stereotype* extends the vocabulary of the UML, allowing we to create new kinds of building blocks that are derived from existing ones but that are specific to our problem. For example, if we are working in a programming language, such as Java or C++, we will often want to model exceptions. In these languages, exceptions are just classes, although they are treated in very special ways. Typically, we only want to allow them to be thrown and caught, nothing else. We can make exceptions first-class citizens in our models meaning that they are treated like basic building blocks by marking them with an appropriate stereotype, as for the class **Overflow** in Figure.
A *tagged value* extends the properties of a UML stereotype, allowing us to create new information in the stereotype's specification. For example, if we are working on a shrink-wrapped product that undergoes many releases over time, we often want to track the version and author of certain critical abstractions. Version and author are not primitive UML concepts. They can be added to any building block, such as a class, by introducing new tagged values to that building block. In Figure, for example, the class **EventQueue** is extended by marking its version and author explicitly.
A *constraint* extends the semantics of a UML building block, allowing us to add new rules or modify existing ones. For example, we might want to constrain the **EventQueue** class so that all additions are done in order. As Figure shows, we can add a constraint that explicitly marks these for the operation **add.**

Figure. Extensibility Mechanisms



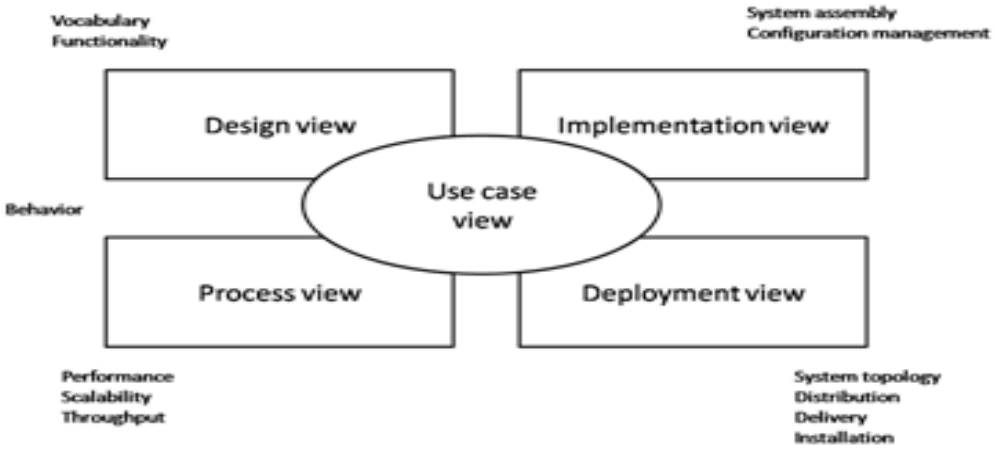
Collectively, these three extensibility mechanisms allow us to shape and grow the UML to our project's needs. These mechanisms also let the UML adapt to new software technology, such as the likely emergence of more powerful distributed programming languages. We can add new building blocks, modify the specification of existing ones, and even change their semantics. Naturally, it's important that we do so in controlled ways so that through these extensions, we remain true to the UML's purpose the communication of information.

Architecture – CO1

A model is a simplified representation of the system. To visualize a system, we will build various models. The subset of these models is a view. Architecture is the collection of several views.

The stakeholders (end users, analysts, developers, system integrators, testers, technical writers and project managers) of a project will be interested in different views.

Architecture can be best represented as a collection five views: 1) Use case view, 2) Design/logical view, 3) Implementation/development view, 4) Process view and 5) Deployment/physical view.



Developed by Philippe Kruchten

Software architecture involves the high level structure of software system abstraction, by using decomposition and composition, with architectural style and quality attributes. A software architecture design must conform to the major functionality and performance requirements of the system, as well as satisfy the non-functional requirements such as reliability, scalability, portability, and availability.

The five views can be summarized as shown in the below table:

View	Stakeholders	Static Aspects	Dynamic Aspects
Use case view	End users Analysts Testers	Use case diagrams	Interaction diagrams Statechart diagrams Activity diagrams
Design view	End users	Class diagrams	Interaction diagrams Statechart diagrams Activity diagrams
Implementation view	Programmers Configuration managers	Component diagrams	Interaction diagrams Statechart diagrams Activity diagrams
Process view	Integrators	Class diagrams (active classes)	Interaction diagrams Statechart diagrams Activity diagrams
Deployment view	System Engineer	Deployment diagrams	Interaction diagrams Statechart diagrams Activity diagrams

A software architecture must describe its group of components, their connections, interactions among them and deployment configuration of all components.

A software architecture can be defined in many ways −

- **UML (Unified Modeling Language)** − UML is one of object-oriented solutions used in software modeling and design.
- **Architecture View Model (4+1 view model)** − Architecture view model represents the functional and non-functional requirements of software application.
- **ADL (Architecture Description Language)** − ADL defines the software architecture formally and semantically.

UML

UML stands for Unified Modeling Language. It is a pictorial language used to make software blueprints. UML was created by Object Management Group (OMG). The UML 1.0 specification draft was proposed to the OMG in January 1997. It serves as a standard for software requirement analysis and design documents which are the basis for developing a software.

UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document a software system. Although UML is generally used to model software system, it is not limited within this boundary. It is also used to model non software systems such as process flows in a manufacturing unit.

The elements are like components which can be associated in different ways to make a complete UML picture, which is known as a **diagram**. So, it is very important to understand the different diagrams to implement the knowledge in real-life systems. We have two broad categories of diagrams and they are further divided into sub-categories i.e. **Structural Diagrams** and **Behavioral Diagrams**.

Structural Diagrams

Structural diagrams represent the static aspects of a system. These static aspects represent those parts of a diagram which forms the main structure and is therefore stable.

These static parts are represented by classes, interfaces, objects, components and nodes. Structural diagrams can be sub-divided as follows −

- Class diagram
- Object diagram
- Component diagram

- Deployment diagram
- Package diagram
- Composite structure

The following table provides a brief description of these diagrams −

Sr.No.	Diagram & Description
1	**Class** Represents the object orientation of a system. Shows how classes are statically related.
2	**Object** Represents a set of objects and their relationships at runtime and also represent the static view of the system.
3	**Component** Describes all the components, their interrelationship, interactions and interface of the system.
4	**Composite structure** Describes inner structure of component including all classes, interfaces of the component, etc.
5	**Package** Describes the package structure and organization. Covers classes in the package and packages within another package.
6	**Deployment** Deployment diagrams are a set of nodes and their relationships. These nodes are physical entities where the components are deployed.

Behavioral Diagrams

Behavioral diagrams basically capture the dynamic aspect of a system. Dynamic aspects are basically the changing/moving parts of a system. UML has the following types of behavioral diagrams −

- Use case diagram
- Sequence diagram
- Communication diagram
- State chart diagram
- Activity diagram
- Interaction overview
- Time sequence diagram

The following table provides a brief description of these diagram −

Sr.No.	Diagram & Description
1	**Use case** Describes the relationships among the functionalities and their internal/external controllers. These controllers are known as actors.

2	**Activity** Describes the flow of control in a system. It consists of activities and links. The flow can be sequential, concurrent, or branched.
3	**State Machine/state chart** Represents the event driven state change of a system. It basically describes the state change of a class, interface, etc. Used to visualize the reaction of a system by internal/external factors.
4	**Sequence** Visualizes the sequence of calls in a system to perform a specific functionality.
5	**Interaction Overview** Combines activity and sequence diagrams to provide a control flow overview of system and business process.
6	**Communication** Same as sequence diagram, except that it focuses on the object's role. Each communication is associated with a sequence order, number plus the past messages.
7	**Time Sequenced** Describes the changes by messages in state, condition and events.

Architecture View Model

A model is a complete, basic, and simplified description of software architecture which is composed of multiple views from a particular perspective or viewpoint.

A view is a representation of an entire system from the perspective of a related set of concerns. It is used to describe the system from the viewpoint of different stakeholders such as end-users, developers, project managers, and testers.

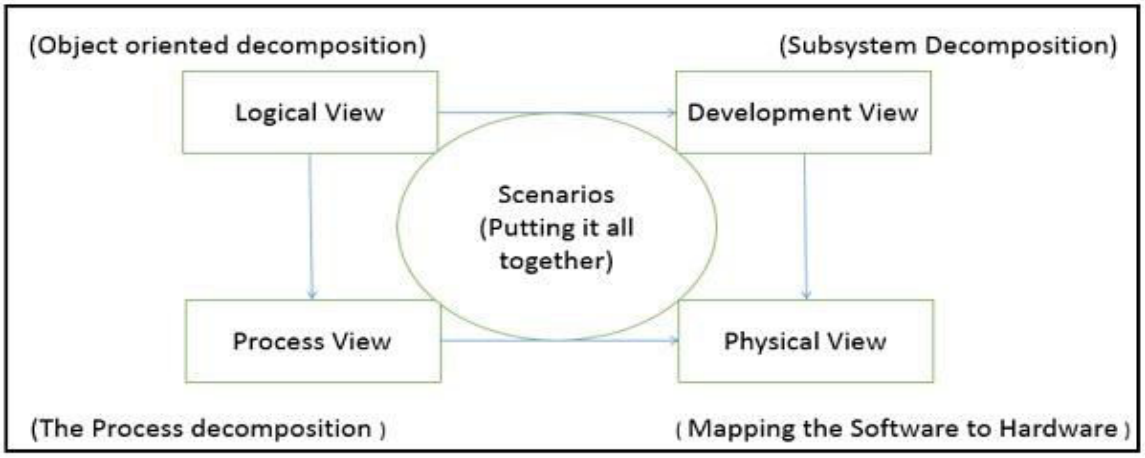
4+1 View Model

The 4+1 View Model was designed by Philippe Kruchten to describe the architecture of a software–intensive system based on the use of multiple and concurrent views. It is a **multiple view** model that addresses different features and concerns of the system. It standardizes the software design documents and makes the design easy to understand by all stakeholders.

It is an architecture verification method for studying and documenting software architecture design and covers all the aspects of software architecture for all stakeholders. It provides four essential views –

- **The logical view or conceptual view** − It describes the object model of the design.
- **The process view** − It describes the activities of the system, captures the concurrency and synchronization aspects of the design.
- **The physical view** − It describes the mapping of software onto hardware and reflects its distributed aspect.
- **The development view** − It describes the static organization or structure of the software in its development of environment.

This view model can be extended by adding one more view called **scenario view** or **use case view** for end-users or customers of software systems. It is coherent with other four views and are utilized to illustrate the architecture serving as "plus one" view, (4+1) view model. The following figure describes the software architecture using five concurrent views (4+1) model.



Reason for calling it 4+1 instead of 5 is:

The **use case view** has a special significance as it details the high level requirement of a system while other views details — how those requirements are realized. When all other four views are completed, it's effectively redundant. However, all other views would not be possible without it. The following image and table shows the 4+1 view in detail −

	Logical	Process	Development	Physical	Scenario
Description	Shows the component (Object) of system as well as their interaction	Shows the processes / Workflow rules of system and how those processes communicate , focuses on dynamic view of system	Gives building block views of system and describe static organization of the system modules	Shows the installation, configuration and deployment of software application	Shows the design is complete by performing validation and illustration
Viewer / Stake holder	End-User, Analysts and Designer	Integrators & developers	Programmer and software project managers	System engineer, operators, system administrators and system installers	All the views of their views and evaluators
Consider	Functional	Non	Software	Nonfunction	System

	requirements	Functional Requirements	Module organization (Software management reuse, constraint of tools)	al requirement regarding to underlying hardware	Consistenc y and validity
UML – Diagram	Class, State, Object, sequence, Communicatio n Diagram	Activity Diagram	Component, Package diagram	Deployment diagram	Use case diagram

Architecture Description Languages (ADLs)

An ADL is a language that provides syntax and semantics for defining a software architecture. It is a notation specification which provides features for modeling a software system's conceptual architecture, distinguished from the system's implementation.

ADLs must support the architecture components, their connections, interfaces, and configurations which are the building block of architecture description. It is a form of expression for use in architecture descriptions and provides the ability to decompose components, combine the components, and define the interfaces of components.

An architecture description language is a formal specification language, which describes the software features such as processes, threads, data, and sub-programs as well as hardware component such as processors, devices, buses, and memory.

It is hard to classify or differentiate an ADL and a programming language or a modeling language. However, there are following requirements for a language to be classified as an ADL −

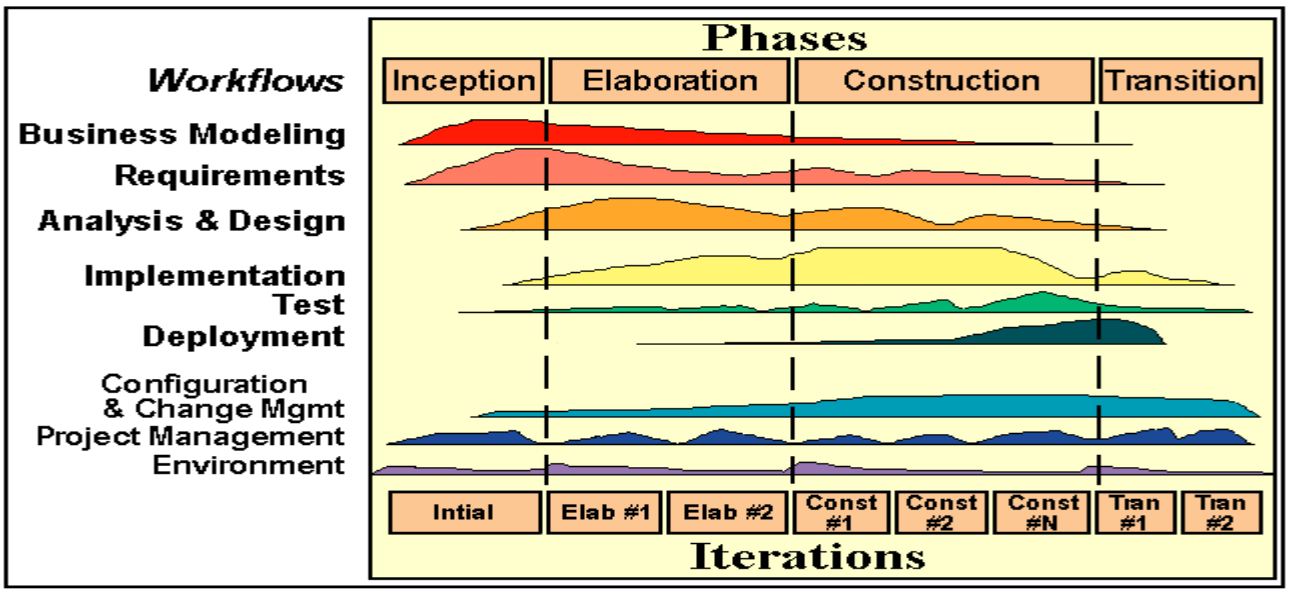
- It should be appropriate for communicating the architecture to all concerned parties.
- It should be suitable for tasks of architecture creation, refinement, and validation.
- It should provide a basis for further implementation, so it must be able to add information to the ADL specification to enable the final system specification to be derived from the ADL.
- It should have the ability to represent most of the common architectural styles.
- It should support analytical capabilities or provide quick generating prototype implementations.

Software Development Life Cycle – CO1

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

- Use case driven
- Architecture centric
- Iterative and Incremental

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



Critical activities in each phase:

Inception:

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

Elaboration:

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

Construction:

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

Transition:

- Deployment of software
- New releases
- Training

CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 3
Unit - 3
Year and Semester: IIIyear &II Semester
Subject**: Object Oriented Analysis and Design**
Branch: **CSE**
Faculty: **Dr** **,** Professor (CSE)

12IT302CV Object Oriented Analysis and Design

Instruction: 4 Periods/Week Credits: 4 Sessional Marks: 25
End Examination: 75 Marks End Exam Duration : 3 Hours

Unit I Introduction to UML – CO1: Importance of Modeling – CO1, Principles of Modeling – CO1, Object Oriented modeling – CO1, Conceptual Model of the UML – CO1, Architecture – CO1, Software Development Life Cycle – CO1.
Unit II Basic Structural Modeling - CO2: Classes – CO2, Relationships – CO2, Common Mechanisms – CO2, and diagrams – CO2.
Advanced Structural Modeling – CO2: Advanced Classes – CO2, advanced relationships CO2, interfaces, Types and Roles – CO2, Packages – CO2.
Unit III Class & Object Diagrams – CO1 & CO2: Terms - CO1 & CO2, concepts - CO1 & CO2, modeling techniques for Class & Object Diagrams - CO1 & CO2.
Unit IV Basic Behavioral Modeling-I – CO2: Interactions CO2, Interaction diagrams CO2.
Unit V Basic Behavioral Modeling-II - CO2 & CO3: Use cases - CO2 & CO3, Use case Diagrams - CO2 & CO3, Activity Diagrams - CO2 & CO3.
Unit VI Advanced Behavioral Modeling - CO2 & CO3: Events and signals - CO2 & CO3, state machines - CO2 & CO3, processes and Threads - CO2 & CO3, time and space - CO2 & CO3, state chart diagrams - CO2 & CO3.
Unit VII Architectural Modeling – CO4: Component – CO4, Deployment – CO4, Component Diagrams – CO4 and Deployment diagrams – CO4.
Unit VIII Case Study: The Unified Library application – CO5.

Text Books:

1. **The Unified Modeling Language User Guide, Ivar Jacobson and Grady Booch, James Rumbaugh, Pearson Education, 2009.**
2. **UML 2 Toolkit, Magnus Penker, Brian Lyons, David Fado and Hans-Erik Eriksson, Wiley-Dreamtech India Pvt.Ltd., 2004.**

References:

1. **Fundamentals of Object Oriented Design in UML, Meilir Page-Jones, Pearson Education, 2000.**
2. **Modeling Software Systems Using UML2, Pascal Roques, Wiley-Dreamtech India Pvt. Ltd., 2007.**
3. **Object Oriented Analysis & Design, Atul Kahate, 1st Edition, McGraw-Hill Companies, 2007.**

CVR COLLEGE OF ENGINEERING
An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 3
Unit - 3
Year and Semester: IIIyear &II Semester
Subject**: Object Oriented Analysis and Design**
Branch: **CSE**
Faculty: **Dr** **,** Professor (CSE)

12IT302CV Object Oriented Analysis and Design

Unit III

Unit III
Class & Object Diagrams – CO1 & CO2

A static **UML object diagram** is an instance of a **class diagram**; it shows a snapshot of the detailed state of a system at a point in time, thus an **object diagram** encompasses **objects** and their relationships at a point in time. It may be considered a special case of a **class diagram** or a communication **diagram**. The use of **object diagrams** is fairly limited, mainly to show examples of data structures. **Class diagram** shows a collection of declarative (static) model elements, such as **classes**, types, and their contents and relationships. **Object diagram** encompasses **objects** and their relationships at a point in time.

Terms, concepts, modeling techniques for Class Diagrams – CO1 & CO2

Terms and concepts: A class diagram is a diagram that shows a set of classes, interfaces, and collaborations and their relationships. Graphically, a class diagram is a collection of vertices and arcs. Common Properties: A class diagram is just a special kind of diagram and shares the same common properties as do all other diagrams name and graphical content that are a projection into a model. What distinguishes a class diagram from other kinds of diagrams is its particular content.

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, both of which are used to group elements of our model into larger chunks. Sometimes we'll want to place instances in our class diagrams as well, especially when we want to visualize the (possibly dynamic) type of an instance.

Common Uses : We use class diagrams to model the static design view of a system. This view primarily supports the functional requirements of a system the services the system should provide to its end users. When we model the static design view of a system, we'll typically use class diagrams in one of three ways.

1. To model 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. We use class diagrams to specify these abstractions and their responsibilities.

2. To model 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. For example, when we're modeling the semantics of a transaction in a distributed system,

we can't just stare at a single class to understand what's going on. Rather, these semantics are carried out by a set of classes that work together. We use class diagrams to visualize and specify this set of classes and their relationships.

3. To model a logical database schema Think of a schema as the blueprint for the conceptual design of a database. In many domains, we'll want to store persistent information in a relational database or in an object-oriented database. We can model schemas for these databases using class diagrams.

Common Modeling Techniques: 1. Modeling Simple Collaborations To model a collaboration, • Identify the mechanism we'd like to model. A mechanism represents some function or behavior of the part of the system we 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, we'll discover parts of our 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 in to concrete attributes and operations.

2. Modeling a Logical Database Schema To model a schema,

• Identify those classes in our model whose state must transcend the lifetime of their applications. • Create a class diagram that contains these classes. We can define our own set of stereotypes and 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 multiplicities that relate these classes.

• Watch for common patterns that complicate physical database design, such as cyclic associations and one-to-one associations. Where necessary, create intermediate abstractions to simplify our 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 us transform our logical design into a physical design.

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. In fact, this is a major reason why we need models in addition to code. Structural features, such as collaborations, and behavioral features, such as interactions, can be visualized clearly in the UML, but not so clearly from raw code.

To forward engineer a class diagram,

• Identify the rules for mapping to our implementation language or languages of choice. This is something we'll want to do for our project or our organization as a whole.

• Depending on the semantics of the languages we choose, we may want to constrain our use of certain UML features. For example, the UML permits us to model multiple inheritance, but Smalltalk permits only single inheritance. We can choose to prohibit developers from modeling with multiple inheritance (which makes our models language-dependent), or we can develop idioms that transform these richer features into the implementation language (which makes the mapping more complex).

• Use tagged values to guide implementation choices in our target language. We can do this at the level of individual classes if we need precise control. We can also do so at a higher level, such as with collaborations or packages.
• Use tools to generate code.

```
public abstract class EventHandler {
EventHandler successor;
private Integer currentEventID;
private String source;
EventHandler() {}
public void handleRequest() {}
}
```

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 we'll need to build useful models. At the same time, reverse engineering is incomplete. There is a loss of information when forward engineering models into code, and so we can't completely recreate a model from code unless our 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 our implementation language or languages of choice. This is something we'll want to do for our project or our organization as a whole.
• Using a tool, point to the code we'd like to reverse engineer. Use our tool to generate a new model or modify an existing one that was previously forward engineered. It is unreasonable to expect to reverse engineer a single concise model from a large body of code. We need to select portion of the code and build the model from the bottom.
• Using our tool, create a class diagram by querying the model. For example, we 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 our intent.
• Manually add design information to the model to express the intent of the design that is missing or hidden in the code.

Terms, concepts, modeling techniques for Object Diagrams – CO1 & CO2

Object diagrams model the instances of things contained in class diagrams. An object diagram shows a set of objects and their relationships at a point in time.
Terms and concepts: 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.
Common Properties: 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. What distinguishes an object diagram from all other kinds of diagrams is its particular content.
Contents: Object diagrams commonly contain •Objects •Links Like all other diagrams, object diagrams may contain notes and constraints. Sometimes we'll want to place classes in our object diagrams as well, especially when we want to visualize the classes behind each instance.
Common Uses: We use object diagrams to model the static design view or static process view of a system just as we do with class diagrams, but from the perspective of real or prototypical instances.

This view primarily supports the functional requirements of a system that is, the services the system should provide to its end users. Object diagrams let us model static data structures.

When we model the static design view or static process view of a system, we typically use object diagrams in one way: To model 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. We use object diagrams to visualize, specify, construct, and document the existence of certain instances in our system, together with their relationships to one another.

Common Modeling Techniques: Modeling Object Structures To model an object structure,

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

• Create a collaboration to describe a mechanism.

• 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.

Forward And Reverse Engineering

Forward engineering (the creation of code from a model) 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, we cannot exactly instantiate these objects from the outside.

Reverse engineering (the creation of a model from code) an object diagram can be useful. In fact, while we are debugging our system, this is something that we or our tools will do all the time. For example, if we are chasing down a dangling link, we'll want to literally or mentally draw an object diagram of the affected objects to see where, at a given moment in time, an object's state or its relationship to other objects is broken.

To reverse engineer an object diagram,

•Choose the target we want to reverse engineer. Typically, we'll set our 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.

CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 4
Unit - 4
Year and Semester: IIIyear &II Semester
Subject**: Object Oriented Analysis and Design**
Branch: **CSE**
Faculty: **Dr** **,** Professor (CSE)

12IT302CV Object Oriented Analysis and Design

Instruction: 4 Periods/Week Credits: 4 Sessional Marks: 25
End Examination: 75 Marks End Exam Duration : 3 Hours

Unit I Introduction to UML – CO1: Importance of Modeling – CO1, Principles of Modeling – CO1, Object Oriented modeling – CO1, Conceptual Model of the UML – CO1, Architecture – CO1, Software Development Life Cycle – CO1.
Unit II Basic Structural Modeling - CO2: Classes – CO2, Relationships – CO2, Common Mechanisms – CO2, and diagrams – CO2.
Advanced Structural Modeling – CO2: Advanced Classes – CO2, advanced relationships CO2, interfaces, Types and Roles – CO2, Packages – CO2.
Unit III Class & Object Diagrams – CO1 & CO2: Terms - CO1 & CO2, concepts - CO1 & CO2, modeling techniques for Class & Object Diagrams - CO1 & CO2.
Unit IV Basic Behavioral Modeling-I – CO2: Interactions CO2, Interaction diagrams CO2.
Unit V Basic Behavioral Modeling-II - CO2 & CO3: Use cases - CO2 & CO3, Use case Diagrams - CO2 & CO3, Activity Diagrams - CO2 & CO3.
Unit VI Advanced Behavioral Modeling - CO2 & CO3: Events and signals - CO2 & CO3, state machines - CO2 & CO3, processes and Threads - CO2 & CO3, time and space - CO2 & CO3, state chart diagrams - CO2 & CO3.
Unit VII Architectural Modeling – CO4: Component – CO4, Deployment – CO4, Component Diagrams – CO4 and Deployment diagrams – CO4.
Unit VIII Case Study: The Unified Library application – CO5.

Text Books:

1. **The Unified Modeling Language User Guide, Ivar Jacobson and Grady Booch, James Rumbaugh, Pearson Education, 2009.**
2. **UML 2 Toolkit, Magnus Penker, Brian Lyons, David Fado and Hans-Erik Eriksson, Wiley-Dreamtech India Pvt.Ltd., 2004.**

References:

1. **Fundamentals of Object Oriented Design in UML, Meilir Page-Jones, Pearson Education, 2000.**
2. **Modeling Software Systems Using UML2, Pascal Roques, Wiley-Dreamtech India Pvt. Ltd., 2007.**
3. **Object Oriented Analysis & Design, Atul Kahate, 1st Edition, McGraw-Hill Companies, 2007.**

12IT302CV Object Oriented Analysis and Design

Unit IV

Unit IV
Basic Behavioral Modeling-I – CO2

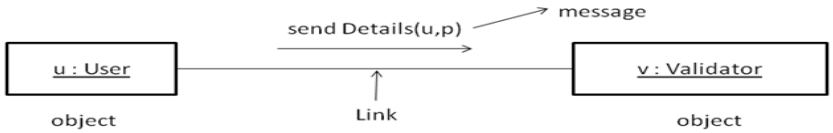
UML 2.1 defines thirteen basic diagram types, divided into two general sets: structural **modeling** diagrams and **behavioral modeling** diagrams. The **UML** is used to define a software system – to detail the artifacts in the systems, to document and construct; it is the language the blueprint is written in.

Interactions – CO2

In UML, the dynamic aspects of a system can be modeled using interactions. Interactions contain messages that are exchanged between objects. A message can be an invocation of an operation or a signal. The messages may also include creation and destruction of other objects.

We can use interactions to model the flow of control within an operation, a class, a component, a use case or the system as a whole. Using interaction diagrams, we can model these flows in two ways: one is by focusing on how the messages are dispatched across time and the second is by focusing on the structural relationships between objects and then consider how the messages are passed between the objects.

Graphically a message is rendered as a directed line with the name of its operation as show below:



Interaction (Definition)

An interaction is a behavior that contains a set of messages exchanged among a set of objects within a context to accomplish a purpose. A message is specification of a communication between objects that conveys information with the expectation that the activity will succeed.

Objects and Roles

The objects that participate in an interaction are either concrete things or prototypical things. As a concrete thing, an object represents something in the real world. For example, p an instance of the class Person, might denote a particular human. Alternately, as a prototypical thing, p might represent any instance of Person.

Links

A link is a semantic connection among objects. In general, a link is an instance of association. Wherever, a class has an association with another class, there may be a link between the instances of the two classes.

Wherever there is a link between two objects, one object can send messages to another object. We can adorn the appropriate end of the link with any of the following standard stereotypes:

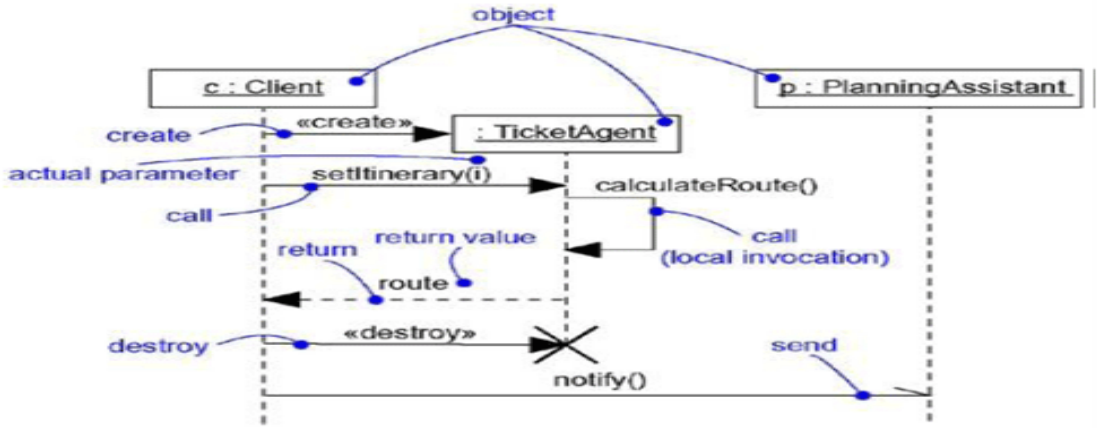
association	Specifies that the corresponding object is visible by association
self	Specifies that the corresponding object is visible as it is the dispatcher of the operation
global	Specifies that the corresponding object is visible as it is in an enclosing scope
local	Specifies that the corresponding object is visible as it is in local scope
parameter	Specifies that the corresponding object is visible as it is a parameter

Messages

A message is the specification of communication among objects that conveys information with the expectation that activity will succeed. The receipt of a message instance may be considered an instance of an event.

When a message is passed, the action that results is an executable statement that forms an abstraction of a computational procedure. An action may result in a change of state. In UML, we can model several kinds of actions like:

Call	Invokes an operation on an object
Return	Returns a value to the caller
Send	Sends a signal to the object
Create	Creates an object
Destroy	Destroys an object



Sequencing

When an object passes a message to another object, the receiving object might in turn send a message to another object, which might send a message to yet a different object and so on.

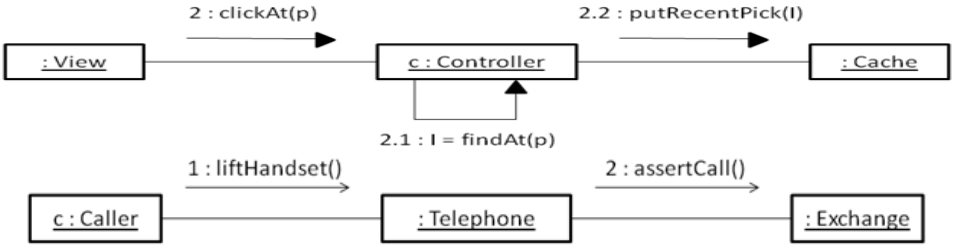
This stream of messages forms a sequence. So, we can define a sequence as a stream of messages. Any sequence must have a beginning. The start of every sequence is associated with some process or thread.

To model the sequence of a message, we can explicitly represent the order of the message relative to the start of the sequence by prefixing the message with a sequence number set apart by a colon separator.

Most commonly, we can specify a procedural or nested flow of control, rendering using a filled solid arrowhead. Less common but also possible, we can specify a flat flow of control, rendered using a stick arrowhead.

We will use flat sequences only when modeling interactions in the context of use cases that involve the system as a whole, together with actors outside the system.

In all other cases, we will use procedural sequences, because they represent ordinary, nested operation calls of the type we find in most programming languages.



Representation

When we model an interaction, we typically include both objects and messages. We can visualize those objects and messages involved in an interaction in two ways: by emphasizing the time ordering of messages and by emphasizing the structural organization of the objects that send and receive messages.

In UML, the first kind of representation is called a sequence diagram and the second kind of representation is called a collaboration diagram. Both sequence and collaboration diagrams are known as interaction diagrams.

Sequence diagrams and collaboration diagrams are isomorphic, meaning that we can take one and transform it into the other without loss of information. Sequence diagram lets us to model the lifeline of an object. An object's lifeline represents the existence of the object at a particular time.

A collaboration diagram lets us to model the structural links that may exist among the objects in the interaction.

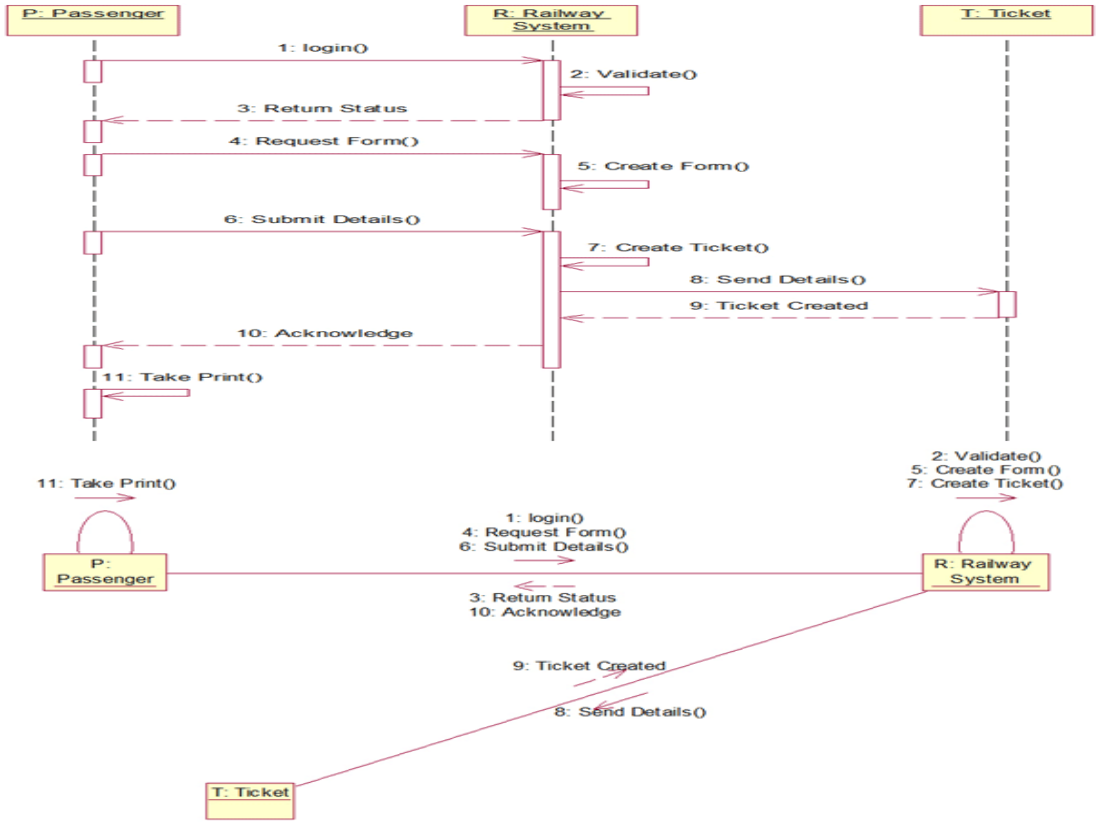
Common Modeling Techniques

Modeling a flow of control

To model a flow of control,

1. Set the context for the interaction, whether it is the system as a whole, a class or an individual operation.
2. Identify the objects and their initial properties which participate in the interaction.
3. Identify the links between objects for communication through messages.
4. In time order, specify the messages that pass from object to object. Use parameters and return values as necessary.
5. To add semantics, adorn each object at every moment in time with its state and role.

Consider the following example of railway reservation system's sequence and collaboration diagrams:



Interaction diagrams – CO2

Introduction

An interaction diagram represents an interaction, which contains a set of objects and the relationships between them including the messages exchanged between the objects.

A sequence diagram is an interaction diagram in which the focus is on time ordering of messages. Collaboration diagram is another interaction diagram in which the focus is on the structural organization of the objects. Both sequence diagrams and collaboration diagrams are isomorphic diagrams.

Common Properties

Interaction diagrams share the properties which are common to all the diagrams in UML. They are: a name which identifies the diagram and the graphical contents which are a projection into the model.

Contents

Interaction diagrams commonly contain:

1. Objects 2. Links 3. Messages

Like all other diagrams, interaction diagrams may contain notes and constraints.

Sequence Diagrams

A sequence diagram is one of the two interaction diagrams. The sequence diagram emphasizes on the time ordering of messages. In a sequence diagram, the objects that participate in the interaction are arranged at the top along the x-axis.

Generally, the object which initiates the interaction is placed on the left and the next important object to its right and so on. The messages dispatched by the objects are arranged from top to bottom along the y-axis. This gives the user the detail about the flow of control over time.

Sequence diagram has two features that distinguish them from collaboration diagrams. First, there is the object lifeline, which is a vertical dashed line that represents the existence of an object over a period of time. Most of the objects are alive throughout the interaction.

Objects may also be created during the interaction with the receipt of the message stereotyped with create. Objects may also be destroyed during the interaction with the receipt of the message stereotyped with destroy.

Second, there is focus of control which is represented as a thin rectangle over the life line of the object. The focus of control represents the points in time at which the object is performing an action. We can also represent recursion by using a self message.

Collaboration Diagrams

A collaboration diagram is one of the two interaction diagrams. The collaboration diagram emphasizes on the structural organization of the objects in the interaction.

A collaboration diagram is made up of objects which are the vertices and these are connected by links. Finally, the messages are represented over the links between the objects. This gives the user the detail about the flow of control in the context of structural organization of objects that collaborate.

Collaboration diagram has two features that distinguish them from the sequence diagrams. First, there is a path which indicates one object is linked to another. Second, there is a sequence number to indicate the time ordering of a message by prefixing the message with a number.

We can use Dewey decimal numbering system for the sequence numbers. For example a message can be numbered as 1 and the next messages in the nested sequence can be numbered 1.1 and so on.

Common Uses

We use interaction diagrams to model the dynamic aspects (interactions) of the system. When we use an interaction diagram to model some dynamic aspect of a system, we do so in the context of the system as a whole, a subsystem, an operation or a class. We typically use the interaction diagrams in two ways:

1. To model flows of control by time ordering
2. To model flows of control by organization

Common Modeling Techniques

Modeling flow of control by time ordering

To model a flow of control by time ordering,

1. Set the context for the interaction, whether it is a system, subsystem, operation or class or one scenario of a use case or collaboration.
2. Identify the objects that take part in the interaction and lay them out at the top along the x-axis in a sequence diagram.
3. Set the life line for each object.
4. Layout the messages between objects from the top along the y-axis.
5. To visualize the points at which the object is performing an action, use the focus of control.
6. To specify time constraints, adorn each message with the time and space constraints.
7. To specify the flow of control in a more formal manner, attach pre and post conditions to each message.

Modeling flow of control by organization

To model a flow of control by organization,

1. Set the context for the interaction, whether it is a system, subsystem, operation or class or one scenario of a use case or collaboration.
2. Identify the objects that take part in the interaction and lay them out in a collaboration diagram as the vertices in a graph.

3. Set the initial properties of each of these objects.
4. Specify the links among these objects.
5. Starting with the messages that initiate the interaction, attach each subsequent message to the appropriate link, setting its sequence number, as appropriate. Use Dewey numbering system to specify nested flow of control.
6. To specify time constraints, adorn each message with the time and space constraints.
7. To specify the flow of control in a more formal manner, attach pre and post conditions to each message.