SEMESTER-IV

OBJECT ORIENTED SOFTWARE ENGINEERING

UNIT –I:

Software Engineering: Software engineering process paradigms, Process Models – Waterfall Model, Iterative Model, RAD Model, Prototype Model. Requirement Analysis, Analysis Model.

UNIT -II:

Introduction to OOAD – What is OOAD? – What is UML? What are the United process(UP) phases -Inception -Use case Modeling - Relating Use cases – include, extend and generalization.

UNIT -III:

Basic Structural Modeling: Classes, Relationships, common Mechanisms, and diagrams. Class & Object Diagrams: Terms, concepts, modeling techniques for Class & Object Diagrams.

UNIT -IV:

Basic BehavioralModeling-I: Interactions, Interaction diagrams, Activity Diagrams. UML state diagrams and modeling, UML deployment and component diagrams.

UNIT -V:

Object Oriented Testing: Overview of Testing, object oriented Testing, Types of Testing, Object oriented Testing strategies, Test case design for OO software.

PVV Durga PraSad **Department Of Computer Applications**

UNIT-I

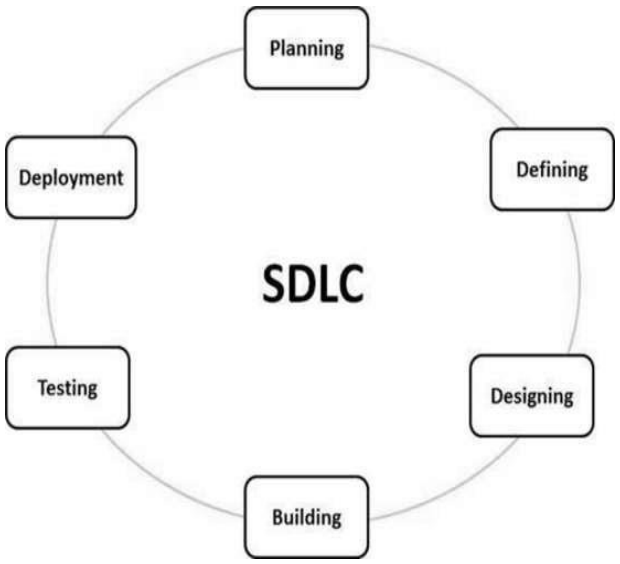
Software Engineering

*"softwareengineering"*istheprocessofsolvingcustomers'problemsbythesystematicdevelopmentandevolutionoflarge,high-qualitysoftwaresystemswithincost,timeandotherconstraints.

❖ Software Engineering process paradigms (or)SDLC:

- SDLC is a process followed for a software project, within a software organization.
- It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software.
- The life cycle defines a methodology for improving the quality of software and the overall development process.

The following figure is a graphical representation of the various stages of a typical SDLC.



A typical Software Development Life Cycle consists of the following stages −

Stage 1: Planning and Requirement Analysis

- Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry.
- This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational and technical areas.

- Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage.

Stage 2: Defining Requirements

- Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts.
- This is done through an **SRS (Software Requirement Specification)** document which consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3: Designing the Product Architecture

- Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification.
- This DDS is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity, budget and time constraints, the best design approach is selected for the product.
- A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any).

Stage 4: Building or Developing the Product

- In this stage of SDLC the actual development starts and the product is built,The programming code is generated as per DDS during this stage.
- If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle(inconvenience).
- Developers must follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers, etc. are used to generate the code.

Stage 5: Testing the Product

- This stage is usually a subset of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC.
- However, this stage refers to the testing only stage of the product where product defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

Stage 6: Deployment in the Market and Maintenance

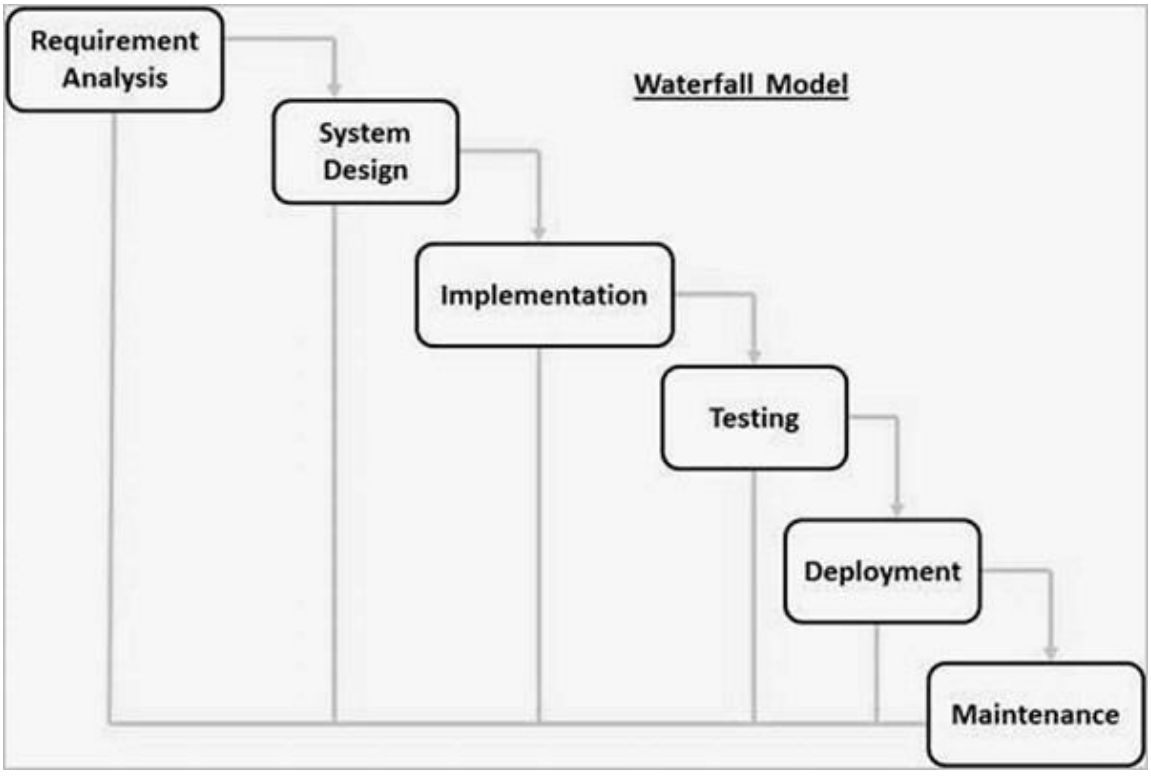
- Once the product is tested and ready to be deployed it is released formally in the appropriate market.

- Sometimes product deployment happens in stages as per the business strategy of that organization. The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing).
- After the product is released in the market, its maintenance is done for the existing customer base.

❖ Software process models:

Software process models are general approaches for organizing aproject intoactivities. They help the project manager and his or herteam to decide whatwork should be done and in what sequence to perform the work. The modelsshouldbeseenas***aidstothinking***,notrigidprescriptionsofthewayt odothings.Eachprojectwillendupwithitsownuniqueplan.

1. Waterfall Model:



- Rather than jumping in and immediately developing a product, the waterfallmodelsuggeststhatsoftwareengineersshouldworkinaseries ofstages.
- Beforecompletingeachstage,theyshouldperformqualityassurance(v erificationandvalidation) so that the next stage can bebuilt on a

good foundation.

- Thewaterfallmodelalsorecognizes,toalimitedextent,thatyousometimeshavetostepbacktoearlierstageswhenyoudiscoveraprobleminasubsequentstage.

The sequential phases in Waterfall model are −

- **Requirement Gathering and analysis** − All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.
- **System Design** − The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.
- **Implementation** − With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.
- **Integration and Testing** − All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.
- **Deployment of system** − Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.
- **Maintenance** − There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

Advantages:

- Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.

Disadvantages:

- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.

2.Iterative Model:

- In this Model, you can start with some of the software specifications and develop the first version of the software.
- After the first version if there is a need to change the software, then a new version of the software is created with a new iteration. Every release of the Iterative Model finishes in an exact and fixed period that is called iteration.
- The Iterative Model allows the accessing earlier phases, in which the variations made respectively. The final output of the project renewed at the end of the Software Development Life Cycle (SDLC) process.

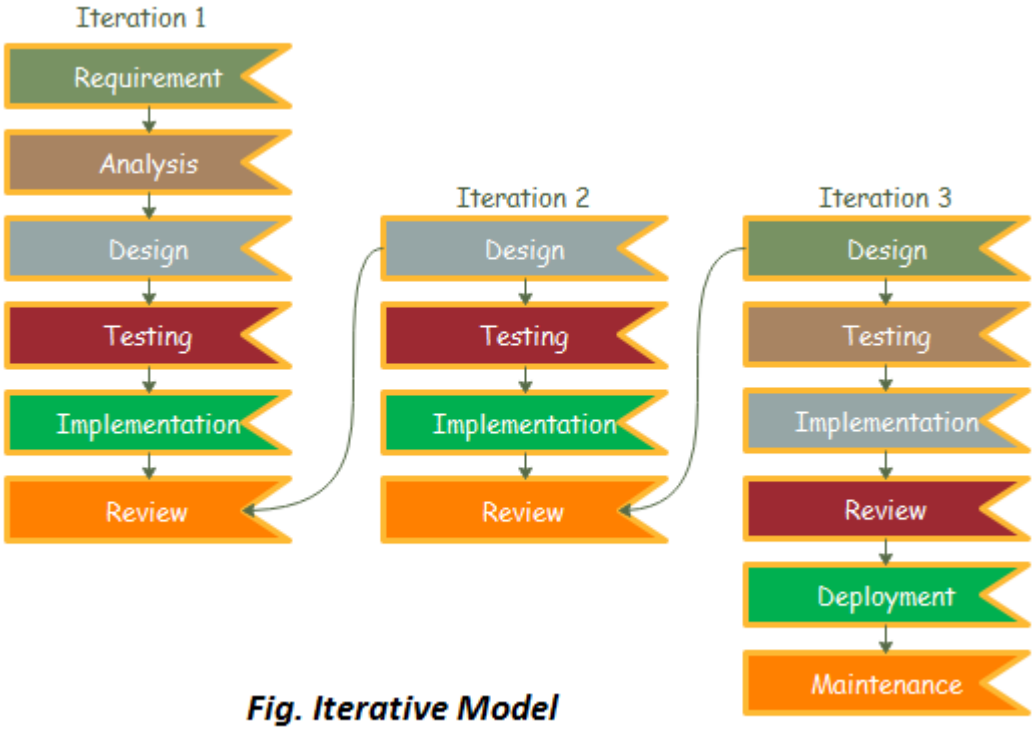


Fig. Iterative Model

The various phases of Iterative model are as follows:

- **Requirement gathering & analysis:** In this phase, requirements are gathered from customers and check by an analyst whether requirements will fulfil or not. Analyst checks that need will achieve within budget or not. After all of this, the software team skips to the next phase.
- **Design:** In the design phase, team design the software by the different diagrams like Data Flow diagram, activity diagram, class diagram, state transition diagram, etc.

- **Implementation:** In the implementation, requirements are written in the coding language and transformed into computer programmes which are called Software.
- **Testing:** After completing the coding phase, software testing starts using different test methods. There are many test methods, but the most common are white box, black box, and grey box test methods.
- **Deployment:** After completing all the phases, software is deployed to its work environment.
- **Review:** In this phase, after the product deployment, review phase is performed to check the behaviour and validity of the developed product. And if there are any error found then the process starts again from the requirement gathering.
- **Maintenance:** In the maintenance phase, after deployment of the software in the working environment there may be some bugs, some errors or new updates are required. Maintenance involves debugging and new addition options.

Advantages:

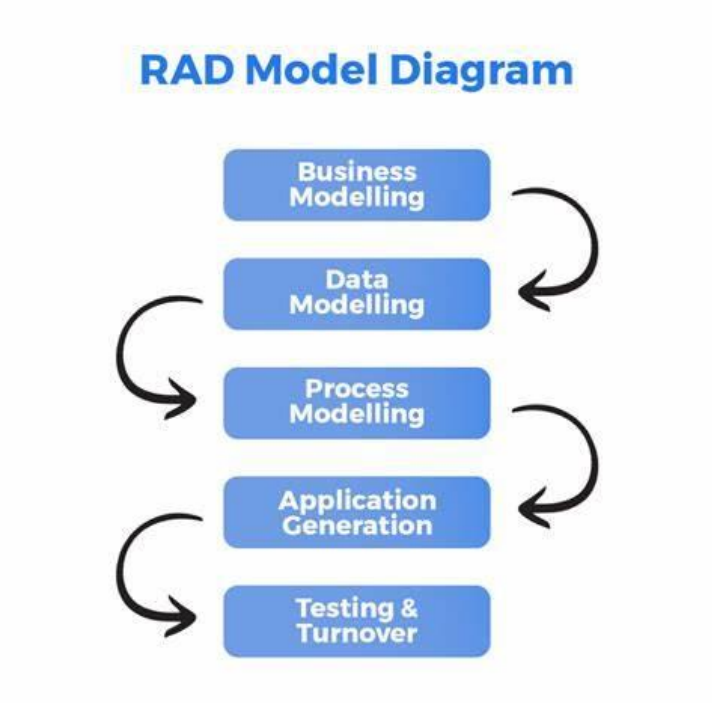
- Testing and debugging during smaller iteration is easy.
- A Parallel development can plan.
- Risks are identified and resolved during iteration.

Disadvantages:

- It is not suitable for smaller projects.
- More Resources may be required.
- Design can be changed again and again because of imperfect requirements.
- Requirement changes can cause over budget.

3.RAD Model:

- **RAD Model** or Rapid Application Development model is a software development process based on prototyping without any specific planning.
- In RAD model, there is less attention paid to the planning and more priority is given to the development tasks.
- It targets at developing software in a short span of time.



- It focuses on input-output source and destination of the information. It emphasizes on delivering projects in small pieces; the larger projects are divided into a series of smaller projects.
- The main features of RAD modeling are that it focuses on the reuse of templates, tools, processes, and code.

SDLC RAD modeling has following phases

- **Business Modeling:**On basis of the flow of information and distribution between various business channels, the product is designed.
- **Data Modeling:**The information collected from business modeling is refined into a set of data objects that are significant for the business.
- **Process Modeling:**The data object that is declared in the data modeling phase is transformed to achieve the information flow necessary to implement a business function.
- **Application Generation:**Automated tools are used for the construction of the software, to convert process and data models into prototypes.
- **Testing and Turnover:**As prototypes are individually tested during every iteration, the overall testing time is reduced in RAD.

Advantages:

- It is easier to transfer deliverables as scripts, high-level abstractions and intermediate codes are used.
- Due to code generators and code reuse, there is a reduction of manual coding.

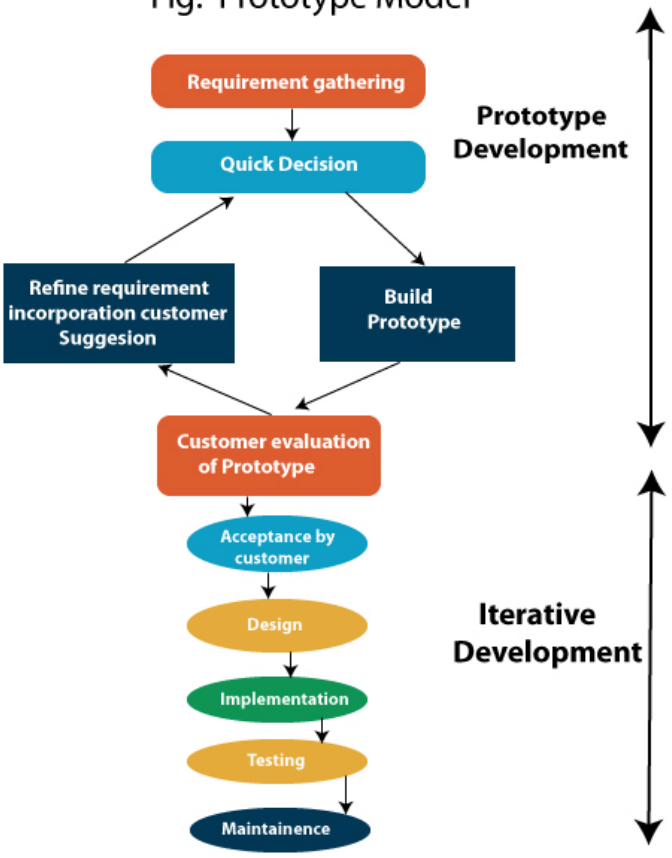
Disadvantages:

- When technical risk is high, it is not suitable.
- It can't be used for smaller projects.
- If developers are not committed to delivering software on time, RAD projects can fail.

4.Prototype Model:

- The Prototyping Model is one of the most popularly used Software Development Life Cycle Models(SDLC models).Prototyping is nothing but the process of developing a working replication of a product or system.
- This model is used when the customers do not know the exact project requirements beforehand.
- In this model,a prototype of the end product is first developed,tested and refined as per customer feedback repeatedly till a final acceptance prototype is achieved which forms the basis for developing the final product.

Fig: Prototype Model



Steps of Prototype Model

The following steps are involved in the working of the Prototype model:

- **Requirement gathering and Analysis:** The users are interviewed to **"collect and define"** requirements for the software product, and various analysis techniques are applied to this information.
- **Quick Decision-Making:** A **preliminary design of the prototype** is made.
- **Building a Prototype:** It is a **small working model** of the software product being built.
- **User Evaluation:** The users evaluate the model for **strengths andweaknesses** and give their suggestions and comments to the developers.
- **Prototype Refinement:**The developers use the **feedback** collected from the users in the previous step to refine the prototype**.**
- **Building the Final Product and Maintenance:**The final product is built based on the **approved prototype** from the previous step. The product is then deployed in the user environment and undergoes routine maintenance.

Types of Prototype Models in Software Engineering

There are four types of Prototype Models available:

1. **Rapid Throwaway Prototyping**
2. **Evolutionary Prototyping**
3. **Incremental Prototyping**
4. **Extreme Prototyping**

Advantages:

- **Errors and missing functionalities** can be identified much early in the lifecycle because the users are actively involved.
- The customer gets to see partial products early in the lifecycle, hence ensuring customer satisfaction.

Disadvantages:

- Prototyping is a **slow and time taking process**.
- Documentation is poor as the requirements **change frequently**.

❖ Requirement Analysis:

- A**requirement**is a statement describing either 1) an aspect of what theproposedsystemmustdo,or2)aconstraintonthesystem'sdevelopment.Ineithercase,it mustcontributeinsomewaytowardsadequatelysolvingthecustomer'sproblem; thesetofrequirementsasawholerepresentsanegotiatedagreementamongallstakeholders.
- Requirements analysis is a set of operations that helps define users' expectations of the application you are building or modifying.
- Software engineering professionals sometimes call it requirement engineering, requirements capturing or requirement gathering.
- The process involves analyzing, documenting, validating and managing system or software requirements.

- **Requirements Analysis Process**

The software requirements analysis process involves the following steps/phases:

1. Eliciting requirements
2. Analyzing requirements
3. Requirements modeling
4. Review and retrospective

1- Eliciting requirements

The process of gathering requirements by communicating with the customers is known as eliciting requirements.

2- Analyzing requirements

This step helps to determine the quality of the requirements. It involves identifying whether the requirements are unclear, incomplete, ambiguous, and contradictory. These issues resolved before moving to the next step.

3- Requirements modeling

In Requirements modeling, the requirements are usually documented in different formats such as use cases, user stories, natural-language documents, or process specification.

4- Review and retrospective

This step is conducted to reflect on the previous iterations of requirements gathering in a bid to make improvements in the process going forward.

- **Requirements Analysis Techniques:**

There are different techniques used for business Requirements Analysis. Below is a list of different Requirements Analysis Techniques:

1. Business process modeling notation (BPMN)
2. UML (Unified Modeling Language)
3. Flowchart technique
4. Data flow diagram
5. Role Activity Diagrams (RAD)
6. Gantt Charts
7. IDEF (Integrated Definition for Function Modeling)
8. Gap Analysis

❖ Analysis Model:

- Analysis model operates as a link between the 'system description' and the 'design model'.
- In the analysis model, information, functions and the behaviour of the system is defined and these are translated into the architecture, interface and component level design in the 'design modeling'.

Elements of the analysis model

1. Scenario based element

- This type of element represents the system user point of view.
- Scenario based elements are use case diagram, user stories.

2. Class based elements

- The object of this type of element manipulated by the system.
- It defines the object,attributes and relationship.
- The collaboration is occurring between the classes.
- Class based elements are the class diagram, collaboration diagram.

3. Behavioral elements

- Behavioral elements represent state of the system and how it is changed by the external events.
- The behavioral elements are sequenced diagram, state diagram.

4. Flow oriented elements

- An information flows through a computer-based system it gets transformed.
- It shows how the data objects are transformed while they flow between the various system functions.
- The flow elements are data flow diagram, control flow diagram.

Analysis Rules of Thumb

The rules of thumb that must be followed while creating the analysis model.

The rules are as follows:

- The model focuses on the requirements in the business domain. The level of abstraction must be high i.e there is no need to give details.
- Every element in the model helps in understanding the software requirement and focus on the information, function and behaviour of the system.
- The consideration of infrastructure and nonfunctional model delayed in the design.
  For example, the database is required for a system, but the classes, functions and behavior of the database are not initially required. If these are initially considered then there is a delay in the designing.
- Throughout the system minimum coupling is required. The interconnections between the modules is known as 'coupling'.
- The analysis model gives value to all the people related to model.
- The model should be simple as possible. Because simple model always helps in easy understanding of the requirement.

UNIT-II

Introduction to OOAD

❖ OOAD:

- **Object-oriented analysis and design** (**OOAD**) is a technical approach for analyzing and designing an application, system, or business by applying object-oriented programming, as well as using visual modeling throughout the software development process to guide stakeholder communication and product quality.
- OOAD in modern software engineering is typically conducted in an iterative and incremental way.
- The outputs of OOAD activities are analysis models (for OOA) and design models (for OOD) respectively.

- **Object-Oriented Analysis(OOA):**
  - The purpose of any analysis activity in the software life-cycle is to create a model of the system's functional requirements that is independent of implementation constraints.
  - The main difference between object-oriented analysis and other forms of analysis is that by the object-oriented approach we organize requirements around objects, which integrate both behaviors (processes) and states (data) modeled after real world objects that the system interacts with.
  - In other or traditional analysis methodologies, the two aspects: processes and data are considered separately. For example, data may be modeled by ER diagrams, and behaviors by flow charts or structure charts.
  - Common models used in OOA are use cases and object models. Use cases describe scenarios for standard domain functions that the system must accomplish.
  - Object models describe the names, class relations (e.g. Circle is a subclass of Shape), operations, and properties of the main objects. User-interface mockups or prototypes can also be created to help understanding.
- **Object-Oriented Design(OOD):**
  - During object-oriented design (OOD), a developer applies implementation constraints to the conceptual model produced in object-oriented analysis.

- Such constraints could include the hardware and software platforms, the performance requirements, persistent storage and transaction, usability of the system, and limitations imposed by budgets and time.
- Concepts in the analysis model which is technology independent, are mapped onto implementing classes and interfaces resulting in a model of the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.
- Important topics during OOD also include the design of softwarearchitectures by applying architectural patterns and design patterns with the object-oriented design principles.

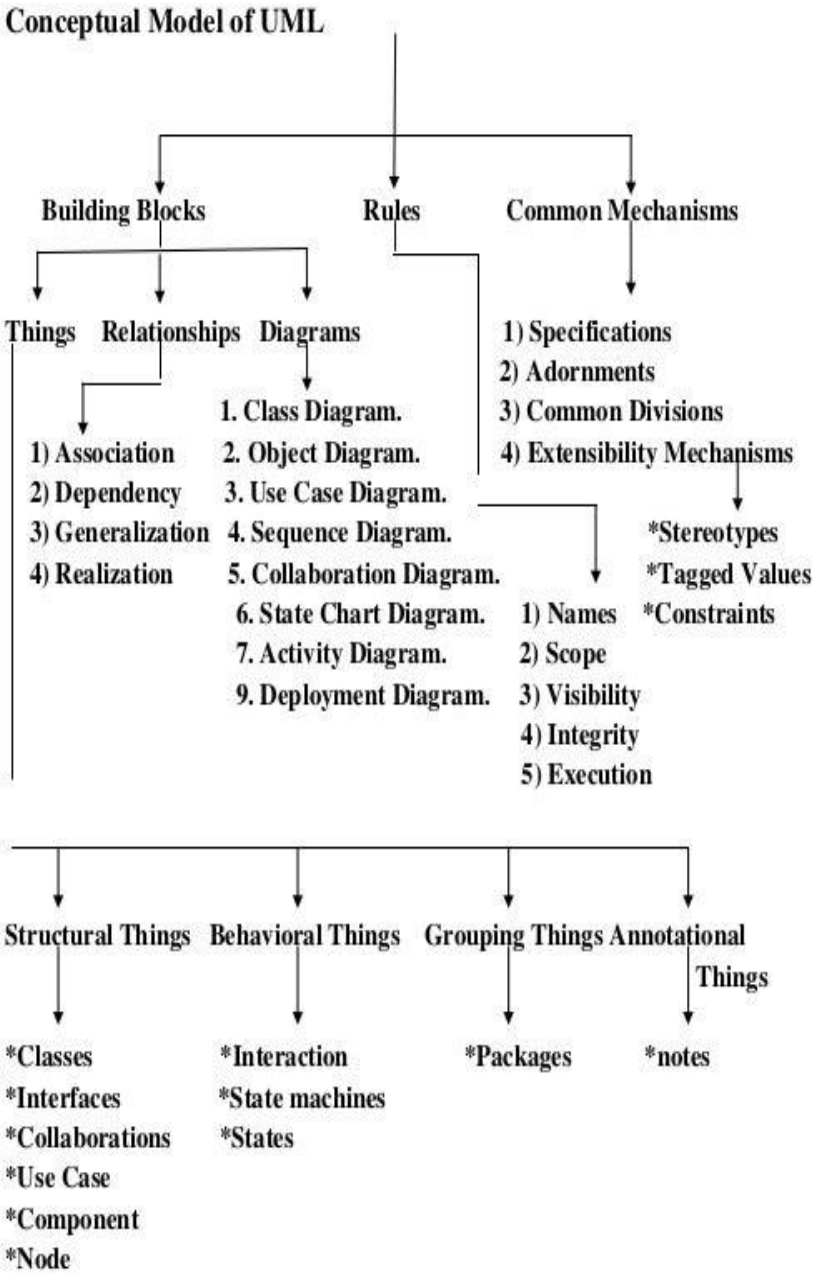
❖ UML:

- The UML stands for Unified modeling language, is a standardized general-purpose visual modeling language in the field of Software Engineering.
- It is used for specifying, visualizing, constructing, and documenting the primary artifacts of the software system.
- It helps in designing and characterizing, especially those software systems that incorporate the concept of Object orientation. It describes the working of both the software and hardware systems.
- The UML was developed in 1994-95 by Grady Booch, Ivar Jacobson, and James Rumbaugh at the Rational Software. In 1997, it got adopted as a standard by the Object Management Group (OMG).

• Modeling Concepts:

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

Conceptual Model of the UML:



- **Goals of UML:**
  - Since it is a general-purpose modeling language, it can be utilized by all the modelers.
  - UML came into existence after the introduction of object-oriented concepts to systemize and consolidate the object-oriented development, due to the absence of standard methods at that time.
  - The UML diagrams are made for business users, developers, ordinary people, or anyone who is looking forward to understand the system, such that the system can be software or non-software.
  - Thus it can be concluded that the UML is a simple modeling approach that is used to model all the practical systems.

- **Characteristics of UML:**
  The UML has the following features:
  - It is a generalized modeling language.
  - It is distinct from other programming languages like C++, Python, etc.
  - It is interrelated to object-oriented analysis and design.
  - It is used to visualize the workflow of the system.
  - Itisa pictorial language, used to generate powerful modelingartifacts(Objects).

❖ Use case Modeling:

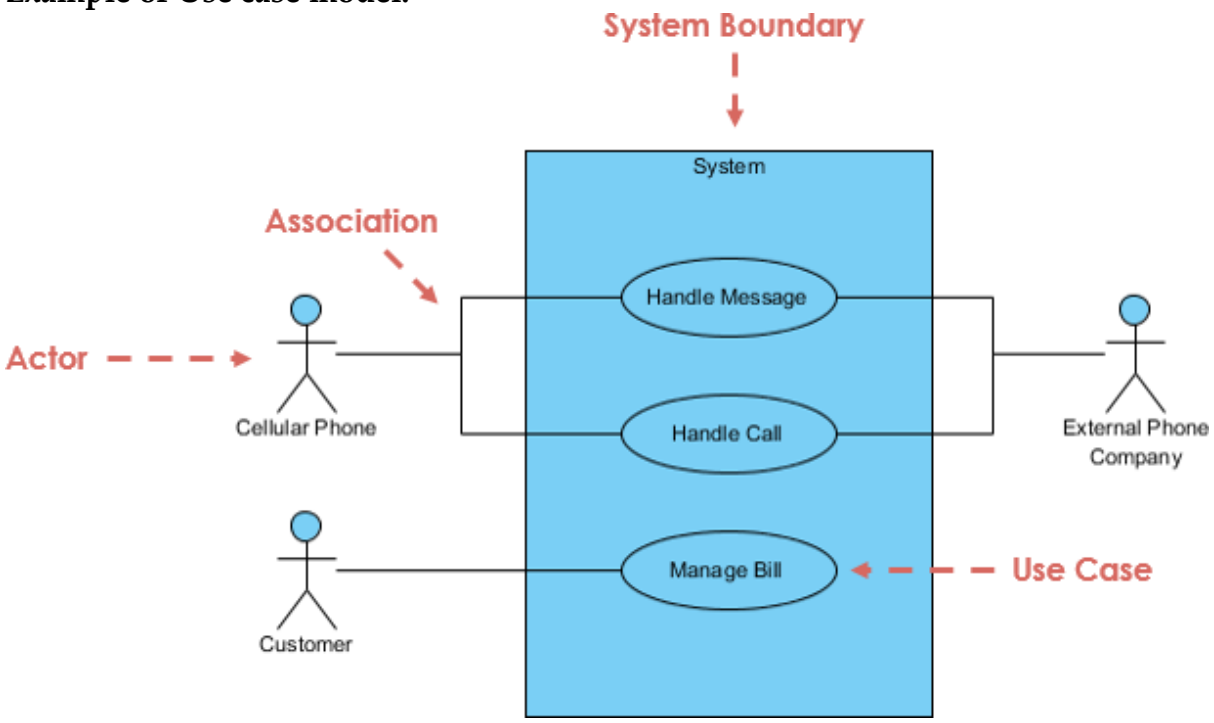
- Modeling is the process of creating a representation of the domain or thesoftware. Various modeling approaches can be used during both requirementsanalysisanddesign.
- Usecasemodelinginvolvesrepresentingthesequencesofactionsperformedbyt heusersofthesoftware.
- Packages may include a use-case model, which is used to organize the model to simplify the analysis, planning, navigation, communication, development and maintenance.
- Various model elements are contained in use-case model, such as
  i)Actors
  ii)Use cases and
  iii) Association between them.

- **Actors:**Usually, actors are people involved with the system defined on the basis of their roles. An actor can be anything such as human or another external system.
  The actor can be represented as

- **Use cases:**The use case defines how actors use a system to accomplish a specific objective. The use cases are generally introduced by the user to meet the objectives of the activities and variants involved in the achievement of the goal. Use cases can be represented as

- **Association:**Associations are another component of the basic model. It is used to define the associations among actors and use cases they contribute in. This association is called communicates-association.

Example of Use case model:



❖ Relating Use cases-include,extend and generalization:

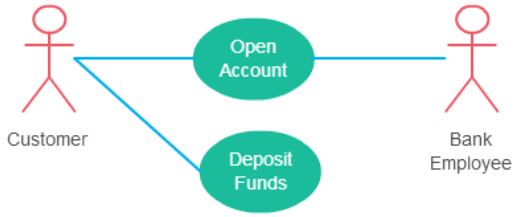
There can be 5 relationship types in a use case diagram.

- Association between actor and use case
- Generalization of an actor

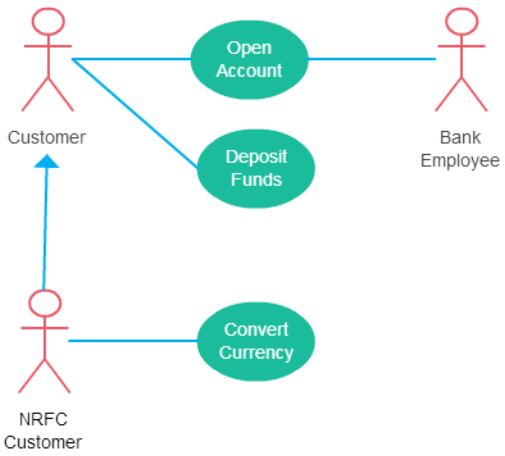
- Extend between two use cases
- Include between two use cases
- Generalization of a use case

➢ **Association Between Actor and Use Case**:
This one is straightforward and present in every use case diagram. Few things to note.

- An actor must be associated with at least one use case.
- An actor can be associated with multiple use cases.
- Multiple actors can be associated with a single use case.



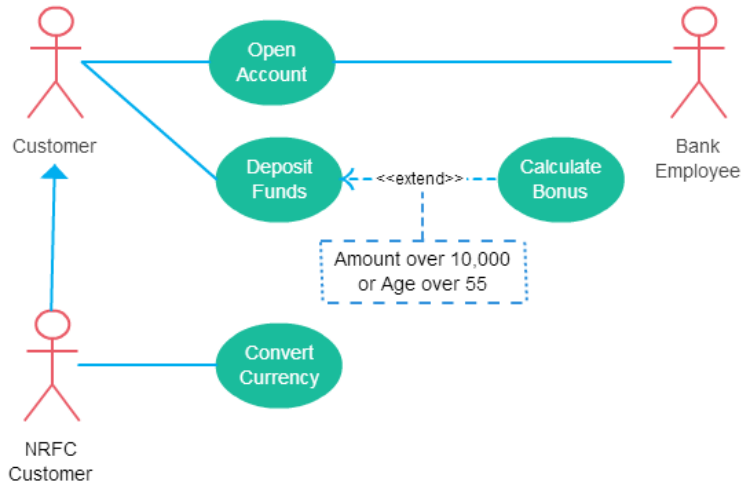
➢ **Generalization of an Actor:**Generalization of an actor means that one actor can inherit the role of the other actor. The descendant inherits all the use cases of the ancestor. The descendant has one or more use cases that are specific to that role. Let's expand the previous use case diagram to show the generalization of an actor.



➢ **Extend Relationship Between Two Use Cases:**Many people confuse the extend relationship in use cases. As the name implies it extends the base use case and adds more functionality to the system. Here are a few things to consider when using the **<<extend>>** relationship.

- **The extending use case is dependent on the extended (base) use case**. In the below diagram the "Calculate Bonus" use case doesn't make much sense without the "Deposit Funds" use case.
- **The extending use case is usually optional** and can be triggered conditionally. In the diagram, you can see that the extending use case is triggered only for deposits over 10,000 or when the age is over 55.
- **The extended (base) use case must be meaningful on its own**. This means it should be independent and must not rely on the behavior of the extending use case.

Lets expand our current example to show the <<extend>> relationship.

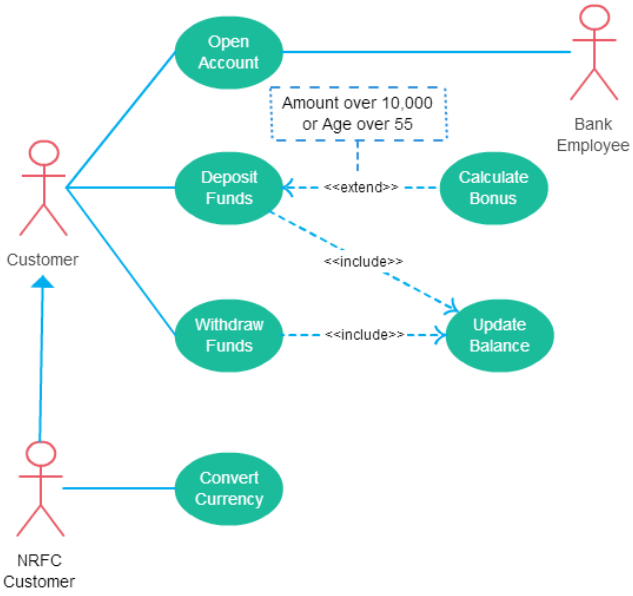


Although extending use case is optional most of the time it is not a must. An extending use case can have non-optional behavior as well. This mostly happens when your modeling complex behaviors.

➢ **Include Relationship Between Two Use Cases:**Include relationship show that the behavior of the included use case is part of the including (base) use case. The main reason for this is to reuse common actions across multiple use cases. In some situations, this is done to simplify complex behaviors. Few things to consider when using the <<include>> relationship.

- The base use case is incomplete without the included use case.
- The included use case is mandatory and not optional.

Lest expand our banking system use case diagram to show include relationships as well.



- **Generalization of a Use Case:**This is similar to the generalization of an actor. The behavior of the ancestor is inherited by the descendant. This is used when there is common behavior between two use cases and also specialized behavior specific to each use case.

For example, in the previous banking example, there might be a use case called "Pay Bills". This can be generalized to "Pay by Credit Card", "Pay by Bank Balance" etc.

Inception:

- The inception phase is a unified process that helps business owners validate their business idea, define risks, clarify the project's requirements, and make the development process run smoothly.
- When you start a new mobile app project, there is tons of work to be done - selecting the best mobile app development team, shaping the business idea, clarifying requirements, creating technical documentation, and finding the most appropriate tech solutions.

- **Phases in Inception:**

Step 1. Initiation

During this planning phase of the software development life cycle, our business analyst schedules a call with you to clarify your project objectives and strategy, including target users, monetization models, an estimated number of users, and project scaling perspectives.

The main activities during the unified process of the inception phase are:

- Introduce the team to stakeholders

- Review your business case and project goals
- Define the current state of the project

Step 2. Research
Next, the Business Analyst (BA) conducts market and competitor research to find out whether there is a place for such a business case in the market and to show how many similar projects already exist.
Main activities include:

- Define business goals and needs
- Demo of an existing product
- Requirements elicitation session
- Identify stakeholders' concerns, risks, and issues

Step 3. Gathering requirements
We write down technical requirements for your project's business case and start creating project technical documentation with use cases, user stories, suitable technologies, and third-party integrations.
Main activities are:

- Technical assessment session
- Requirements elicitation session
- Collect functional and non-functional requirements
- UI/UX review session
- Identify and verify solution use cases

Step 4. Prototyping
Using the use cases of your project, our designer starts making layouts, wireframes, and prototypes while consulting with you on each result.
Our team is busy with the following actions:

- Define scope boundaries
- Prioritize the scope
- Define MVP scope
- Prepare outcome documentation
- Validate outcomes with stakeholders

Step 5. Preparation for the development stage

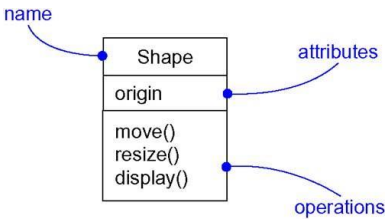
Now, using a technical project specification, we prioritize features to identify the MVP's scope, i.e., the list of functions sufficient to verify your business model and estimate the time and money required to implement one or other features. Main activities are:

- Finalizing outcome documents
- Sending outcomes to the client
- Final meeting to present deliverables to all client stakeholders

UNIT-III

❖ Classes:

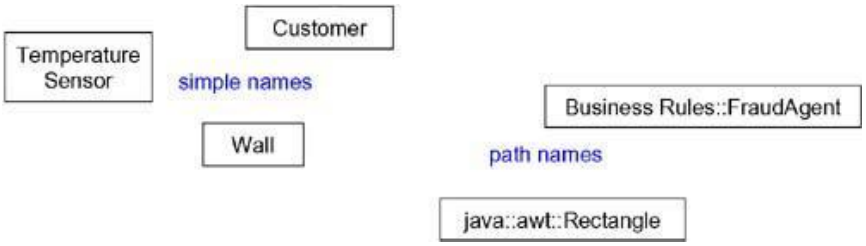
Classes are the most important building block of any object-oriented system. A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.



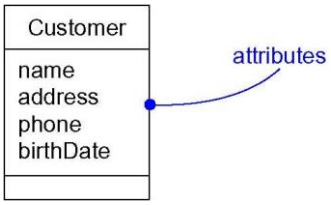
Terms and Concepts:

A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.Graphically, a class is rendered as a rectangle.

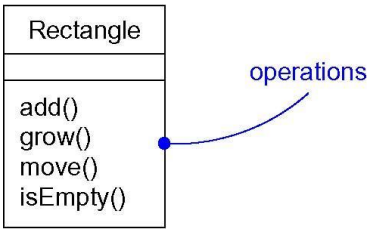
- **Names:**
  Every class must have a name that distinguishes it from other classes. A name is a textual string.



- **Attributes:**
  A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing you are modeling that is shared by all objects



- **Operations:**
  An operation is the implementation of a service that can be requested from any object of the class to affect behavior.

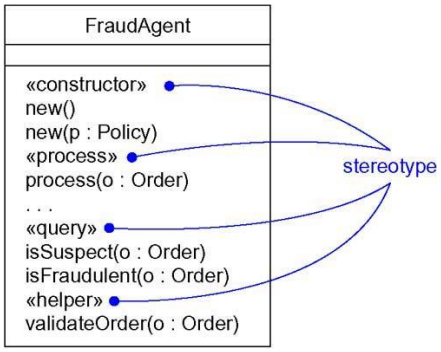


- **Organizing Attributes and Operations:**

➢ When drawing a class, you don't have to show every attribute and every operation at once.

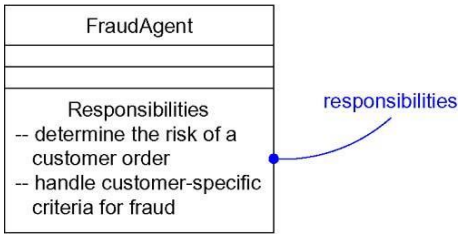
➢ For these reasons, you can merge a class, meaning that you can choose to show only some or none of a class's attributes and operations.

➢ You can indicate that there are more attributes or properties than shown by ending each list with an ellipsis ("...")



- **Responsibilities:**

A responsibility is a contract or an responsibility of a class. When you create a class, you are making a statement that all objects of that class have the same kind of state and the same kind of behavior.



Modeling Techniques:

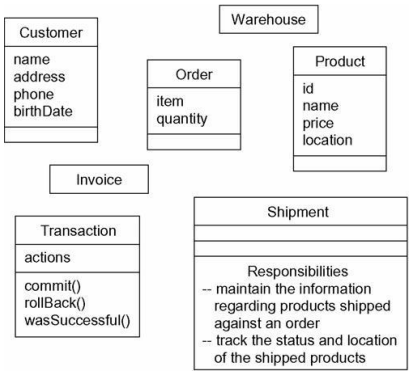
- **Modeling the Vocabulary of a System:**

To model the vocabulary of a system,

➢ Identify those things that users or implementers use to describe the problem or solution. Use CRC cards and use case-based analysis to help find these abstractions.

➢ For each abstraction, identify a set of responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all your classes.

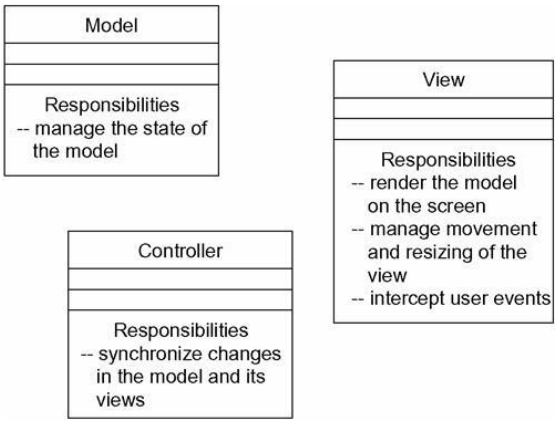
➢ Provide the attributes and operations that are needed to carry out these responsibilities for each class.



- **Modeling the Distribution of Responsibilities in a System:**

To model the distribution of responsibilities in a system,

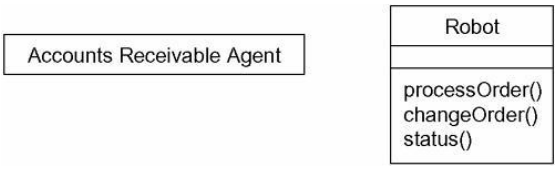
- Identify a set of classes that work together closely to carry out some behavior.
- Identify a set of responsibilities for each of these classes.
- Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions, collapse tiny classes that have trivial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



- **ModelingNonsoftware Things:**

To model nonsoftware things,

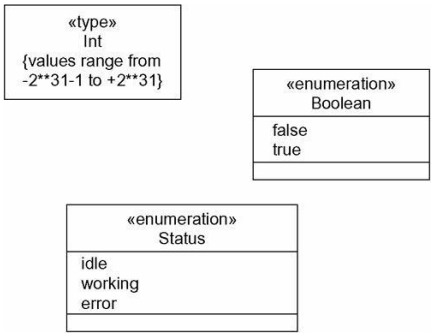
- Model the thing you are abstracting as a class.
- If you want to distinguish these things from the UML's defined building blocks, create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.
- If the thing you are modeling is some kind of hardware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



- **Modeling Primitive Types:**

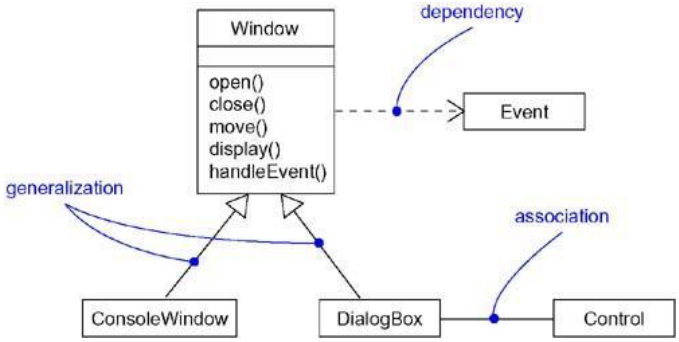
To model primitive types,

- Model the thing you are abstracting as a class or an enumeration, which is rendered using class notation with the appropriate stereotype.
- If you need to specify the range of values associated with this type, use constraints.



❖ Relationships:

- In the UML, the ways that things can connect to one another, either logically or physically, are modeled as relationships.
- In object-oriented modeling, there are threekinds of relationships that are most important: dependencies, generalizations, and associations.

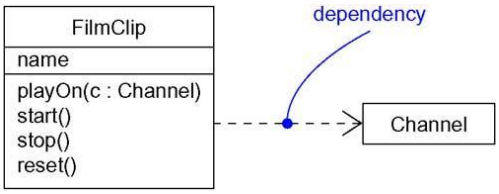


Terms and Concepts:

Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships.

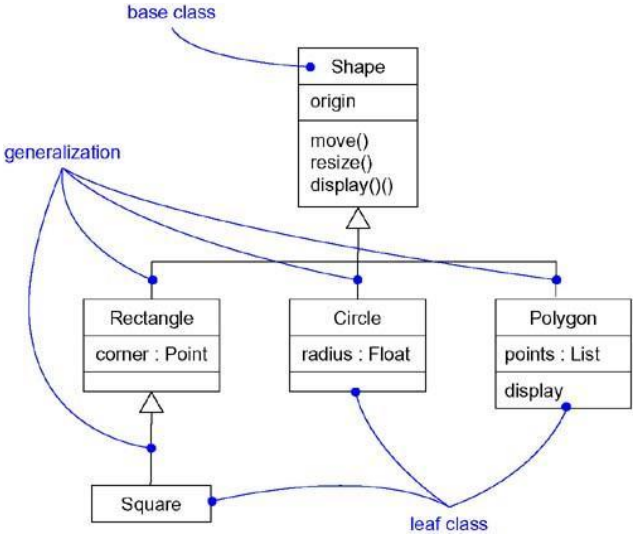
- **Dependencies:**

Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on. Choose dependencies when you want to show one thing using another.



- **Generalization:**

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

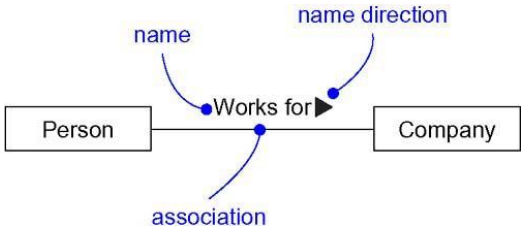


- **Associations:**

An association is a structural relationship that specifies that objects of one thing are connected to objects of another. Although it's not as common, you can have associations that connect more than two classes; these are called n-ary associations. Beyond this basic form, there are four adornments that apply to associations.

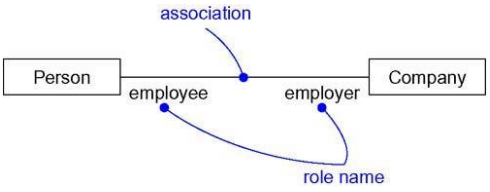
- **Name:**

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



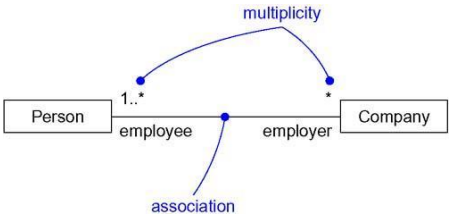
- **Role:**

When a class participates in an association, it has a specific role that it plays in that relationship; a role is just the face the class at the far end of the association presents to the class at the near end of the association.



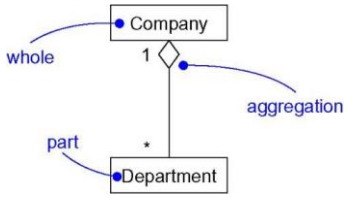
- **Multiplicity :**

An association represents a structural relationship among objects. In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association. This "how many" is called the multiplicity of an association's role.



- **Aggregation :**

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

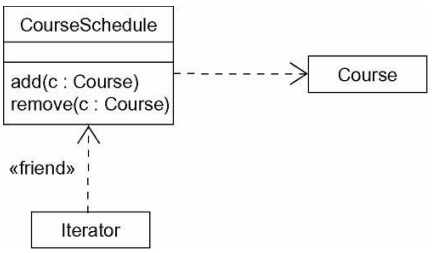


Modeling Techniques:

- **Modeling Simple Dependencies:**

To model this using relationship,

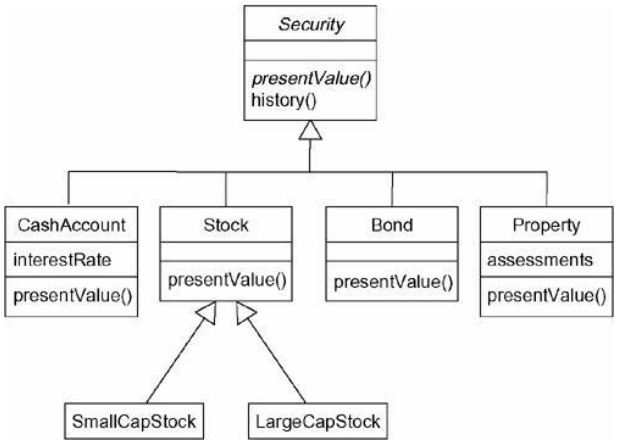
- Create a dependency pointing from the class with the operation to the class used as a parameter in the operation.



- **Modeling Single Inheritance In modeling :**

To model inheritance relationships,

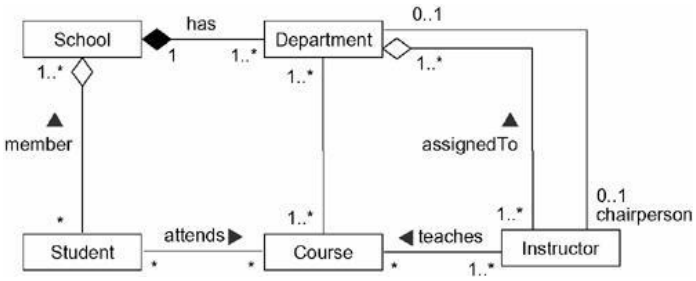
- Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.
- Elevate these common responsibilities, attributes, and operations to a more general class. If necessary, create a new class to which you can assign these elements (but be careful about introducing too many levels).
- Specify that the more-specific classes inherit from the more-general class by placing a generalization relationship that is drawn from each specialized class to its more-general parent.



- **Modeling Structural Relationships:**

To model structural relationships,

- For each pair of classes, if you need to navigate from objects of one to objects of another, specify an association between the two. This is a data-driven view of associations.
- For each pair of classes, if objects of one class need to interact with objects of the other class other than as local variables in a procedure or parameters to an operation, specify an association between the two. This is more of a behavior-driven view of associations.



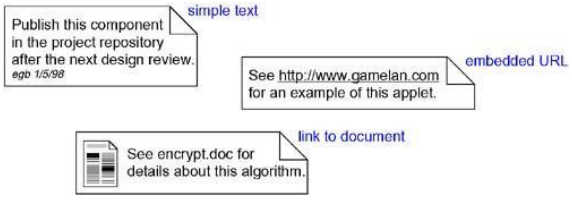
❖ Common mechanisms:

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.

Terms and Concepts:

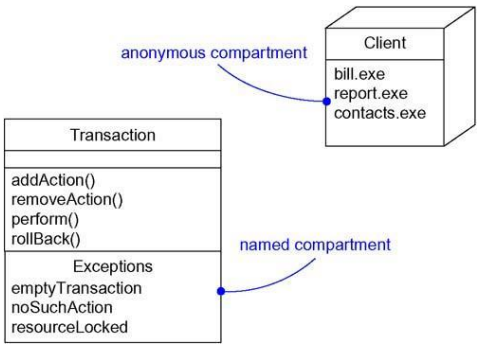
- **Notes**:

A note that renders a comment.Notes are used to specify things like requirements, observations, reviews, and explanations, in addition to rendering constraints. A note may contain any combination of text or graphics



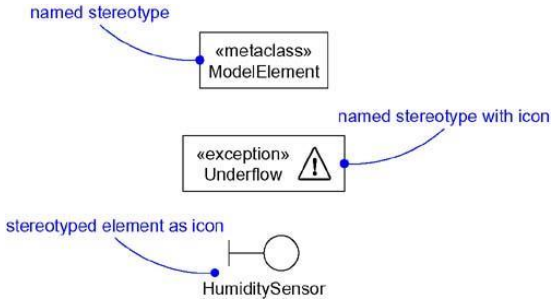
- **Other Adornments:**

Adornments are textual or graphical items that are added to an element's basicnotation and are used to visualize details from the element's specification



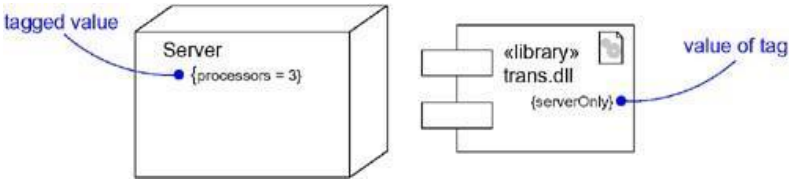
- **Stereotypes:**

The UML provides a language for structural things, behavioral things, grouping things, and notational things. These four basic kinds of things address the overwhelming majority of the systems you'll need to model. In its simplest form, a stereotype is rendered as a name enclosed by guillemets (for example, «name») and placed above the name of another element.



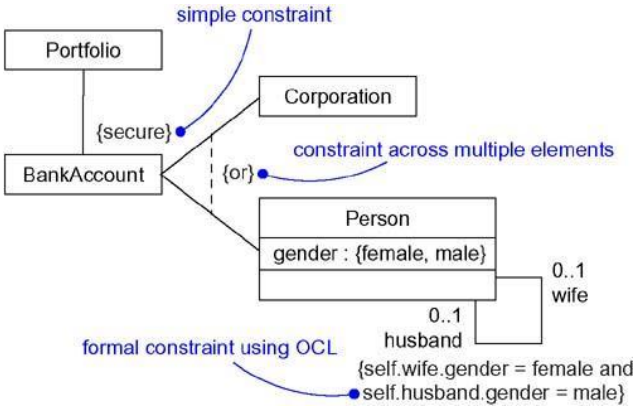
- **Tagged Values:**

Every thing in the UML has its own set of properties: classes have names, attributes, and operations; associations have names and two or more ends, each with its own properties; and so on. With stereotypes, you can add new things to the UML; with tagged values, you can add new properties to a stereotype.



- **Constraints:**

With constraints, you can add new semantics or extend existing rules. A constraint specifies conditions that a run-time configuration must satisfy to conform to the model.

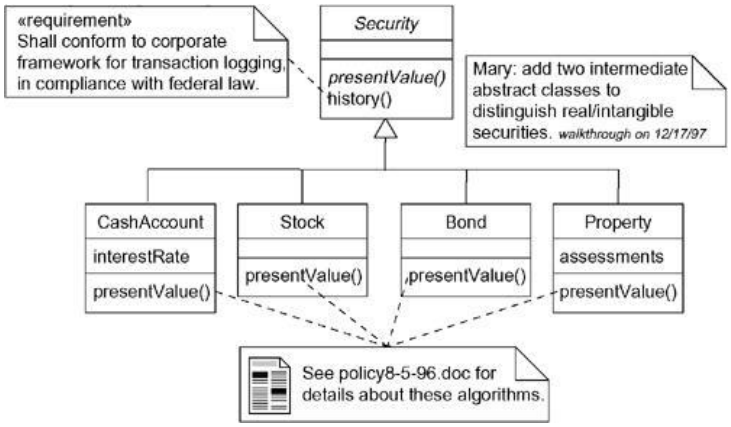


Modeling Techniques:

- **Modeling Comments:**

To model a comment,

- Put your comment as text in a note and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a note to its elements using a dependency relationship.
- Remember that you can hide or make visible the elements of your model as you see fit. This means that you don't have to make your comments visible everywhere the elements to which it is attached are visible. Rather, expose your comments in your diagrams only insofar as you need to communicate that information in that context.
- If your comment is lengthy or involves something richer than plain text, consider putting your comment in an external document and linking or embedding that document in a note attached to your model.
- As your model evolves, keep those comments that record significant decisions that cannot be inferred from the model itself, andunless they are of historic interestdiscard the others.

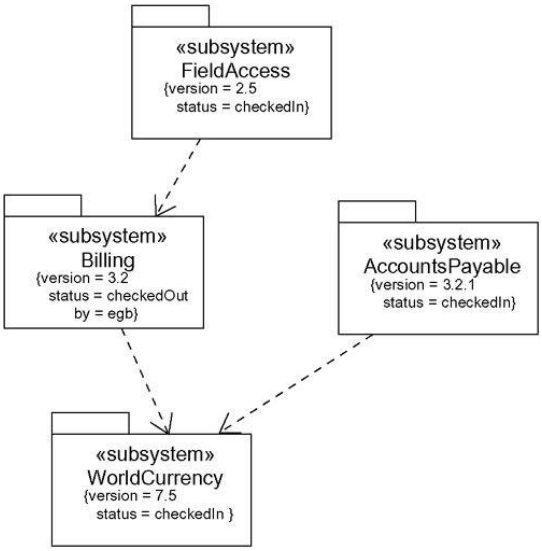


- **Modeling New Properties:**

To model new properties,

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

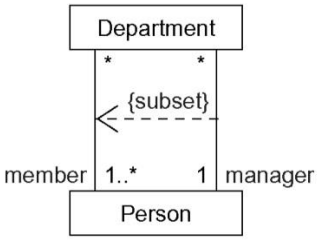
- If you're convinced there's no other way to express these semantics, define a stereotype and add the new properties to the stereotype. The rules of generalization applytagged values defined for one kind of stereotype apply to its children.



- **Modeling New Semantics:**

To model new semantics,

- First, make sure there's not already a way to express what you want by using basic UML.
- If you're convinced there's no other way to express these semantics, write your new semantics in a constraint placed near the element to which it refers. You can show a more explicit relationship by connecting a constraint to its elements using a dependency relationship.
- If you need to specify your semantics more precisely and formally, write your new semantics using OCL.



❖ Diagrams:

- Diagrams are the means by which you view these building blocks. A diagram is a graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).
- You use diagrams to visualize your system from different perspectives. Because no complex system can be understood in its entirety from only one perspective, the UML defines a number of diagrams so that you can focus on different aspects of your system independently.

- Good diagrams make the system you are developing understandable and approachable.
- Choosing the right set of diagrams to model your system forces you to ask the right questions about your system and helps to illuminate the implications of your decisions.

In UML,diagrams are into two parts.They are:

I. Structural Diagrams :

- The UML's structural diagrams exist to visualize, specify, construct, and document the static aspects of a system.
- Just as the static aspects of a house encompass the existence and placement of such things as walls, doors, windows, pipes, wires, and vents, so too do the static aspects of a software system encompass the existence and placement of such things as classes, interfaces, collaborations, components, and nodes.
- The UML's structural diagrams are roughly organized around the major groups of things you'll find when modeling a system.

1.Class diagram- Classes, interfaces, and collaborations
2.Component diagram- Components
3.Object diagram -Objects
4.Deployment diagram -Nodes

II. Behavioral Diagrams:

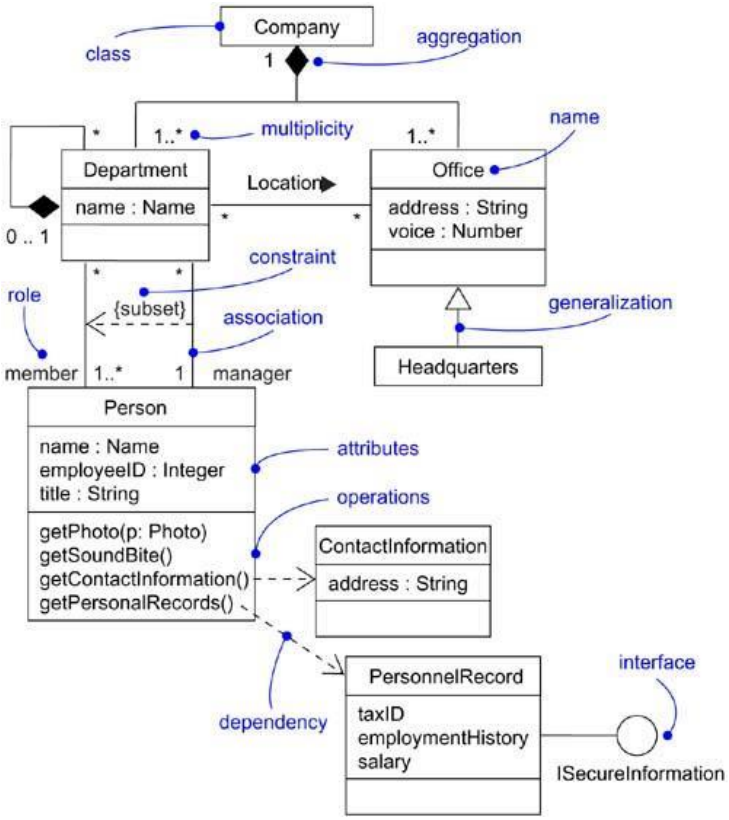
- The UML's behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system.
- Just as the dynamic aspects of a house encompass airflow and traffic through the rooms of a house, so too do the dynamic aspects of a software system encompass such things as the flow of messages over time and the physical movement of components across a network.
- The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.

1.Use case diagram -Organizes the behaviors of the system
2.Sequence diagram- Focuses on the time ordering of messages
3.Collaboration diagram -Focuses on the structural organization of objects that send and receive messages
4.State diagram -Focuses on the changing state of a system driven by events
5.Activity diagram -Focuses on the flow of control from activity to activity

❖ Class Diagram:

- Class diagrams are the most common diagram found in modeling object-oriented systems. A class diagram shows a set of classes, interfaces, and collaborations and their relationships.

- You use class diagrams to model the static design view of a system. For the most part, this involves modeling thevocabulary of the system, modeling collaborations, or modeling schemas.
- Class diagrams are also the foundation for a couple of related diagrams: component diagrams and deployment diagrams.
- Class diagrams are important not only for visualizing, specifying, and documenting structural models, but also for constructing executable systems through forward and reverse engineering.



Terms and concepts:

- **Common Properties:**
  A class diagram is just a special kind of diagram and shares the same common properties as do all other diagrams that is,a name and graphical content that are a projection into a model.
- **Contents:**
  Class diagrams commonly contain the following things:
    - Classes
    - Interfaces
    - Collaborations
    - Dependency, generalization, and association relationships

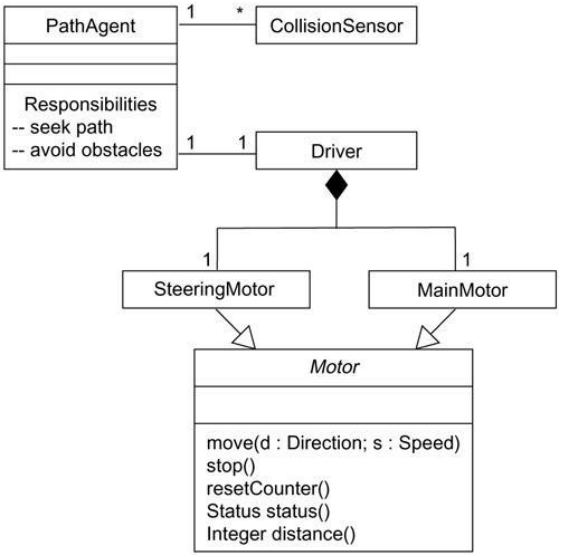
Like all other diagrams, class diagrams may contain notes and constraints.Class diagrams may also contain packages or subsystems

Modeling techniques:

- **Modeling Simple Collaborations:**

To model a collaboration,

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

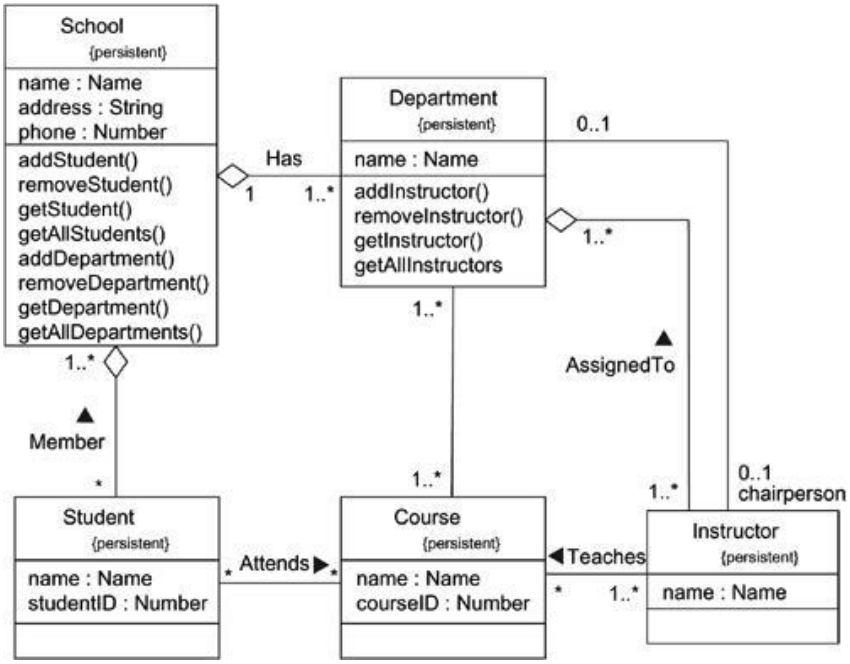


- **Modeling a Logical Database Schema:**

To model a schema,

- Identify those classes in your model whose state must transcend the lifetime of their applications.
- Create a class diagram that contains these classes and mark them as persistent (a standard tagged value). You can define your own set of tagged values to address database-specific details.
- Expand the structural details of these classes. In general, this means specifying the details of their attributes and focusing on the associations and their cardinalities that structure these classes.
- Watch for common patterns that complicate physical database design, such as cyclic associations, one-to-one associations, and n-ary associations. Where necessary, create intermediate abstractions to simplify your logical structure.

- Consider also the behavior of these classes by expanding operations that are important for data access and data integrity. In general, to provide a better separation of concerns, business rules concerned with the manipulation of sets of these objects should be encapsulated in a layer above these persistent classes.
- Where possible, use tools to help you transform your logical design into a physical design.

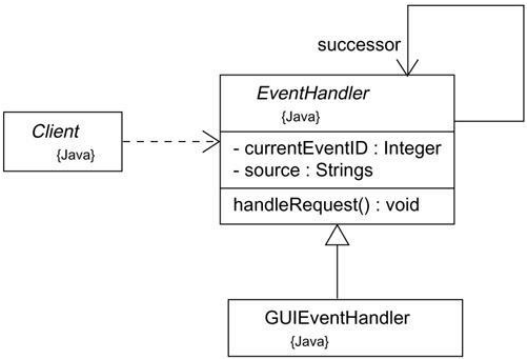


- **Forward and Reverse Engineering:**

Forward engineering is the process of transforming a model into code through a mapping to an implementation language.

To forward engineer a class diagram,

- Identify the rules for mapping to your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Depending on the semantics of the languages you choose, you may have to constrain your use of certain UML features. For example, the UML permits you to model multiple inheritance, but Smalltalk permits only single inheritance. You can either choose to prohibit developers from modeling with multiple inheritance (which makes your models language-dependent) or develop idioms that transform these richer features into the implementation language (which makes the mapping more complex).
- Use tagged values to specify your target language. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages.



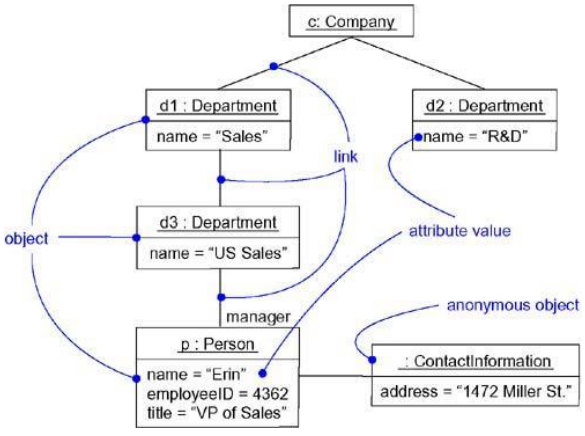
Reverse engineering is the process of transforming code into a model through a mapping from a specific implementation language.

To reverse engineer a class diagram,

- Identify the rules for mapping from your implementation language or languages of choice.This is something you'll want to do for your project or your organization as a whole.
- Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered.
- Using your tool, create a class diagram by querying the model. For example, you might start with one or more classes, then expand the diagram by following specific relationships or other neighboring classes. Expose or hide details of the contents of this class diagram as necessary to communicate your intent.

❖ Object Diagram:

- 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.
- You use object diagrams to model the static design view or static process view of a system. This involves modeling a snapshot of the system at a moment in time and rendering a set of objects, their state, and their relationships.
- Object diagrams are not only important for visualizing, specifying, and documenting structural models, but also for constructing the static aspects of systems through forward and reverse engineering.



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 diagramsthat is, a name and graphical contents that are a projection into a model.

- **Contents:**

Object diagrams commonly contain

- Objects
- Links

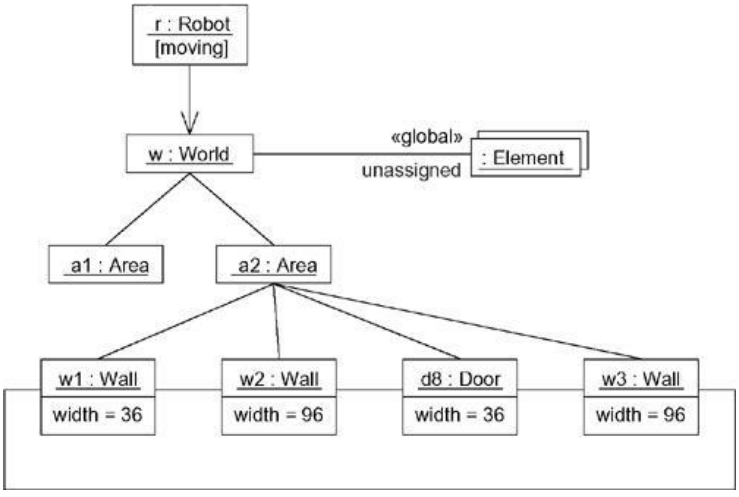
Like all other diagrams, object diagrams may contain notes and constraints.

Modeling Techniques:

- **Modeling Object Structures:**

To model an object structure,

- Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modelingthat results from the interaction of a society of classes, interfaces, and other things.
- For each mechanism, identify the classes, interfaces, and other elements that participate in this collaboration; identify the relationships among these things, as well.
- Consider one scenario that walks through this mechanism. Freeze that scenario at a moment in time, and render each object that participates in the mechanism.
- Expose the state and attribute values of each such object, as necessary, to understand the scenario.
- Similarly, expose the links among these objects, representing instances of associations among them.



- **Forward and Reverse Engineering:**

Forward engineering (the creation of code from a model) an object diagram is theoretically possible but pragmatically of limited value.

Reverse engineering (the creation of a model from code) an object diagram is a very different thing.

To reverse engineer an object diagram,

- Chose the target you want to reverse engineer. Typically, you'll set your context inside an operation or relative to an instance of one particular class.
- Using a tool or simply walking through a scenario, stop execution at a certain moment in time.
- Identify the set of interesting objects that collaborate in that context and render them in an object diagram.
- As necessary to understand their semantics, expose these object's states.
- As necessary to understand their semantics, identify the links that exist among these objects.
- If your diagram ends up overly complicated, prune it by eliminating objects that are not germane to the questions about the scenario you need answered. If your diagram is too simplistic, expand the neighbors of certain interesting objects and expose each object's state more deeply.

UNIT-IV

❖ Interactions:

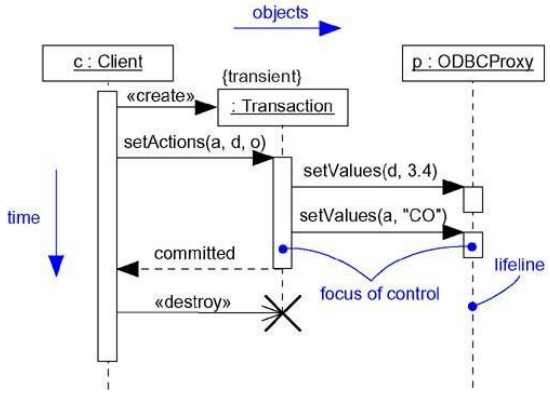
- An interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose.
- The behavior of a society of objects or of an individual operation may be specified with an interaction.
- An interaction involves a number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

❖ Interaction diagrams:

- Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams. It illustrates dynamic view of a system.
- Interaction diagrams are isomorphic, meaning that you can convert from one to the other without loss of information.
- Interaction diagrams commonly contain
  - Objects ·
  - Links
  - Messages
- An interaction diagram is basically a projection of the elements found in an interaction. The semantics of an interaction's context, objects and roles, links, messages, and sequencing apply to interaction diagrams.

Sequence Diagram:

- A sequence diagram is an interaction diagram that emphasizes the time ordering of messages.
- A sequence diagram shows a set of objects and the messages sent and received by those objects.

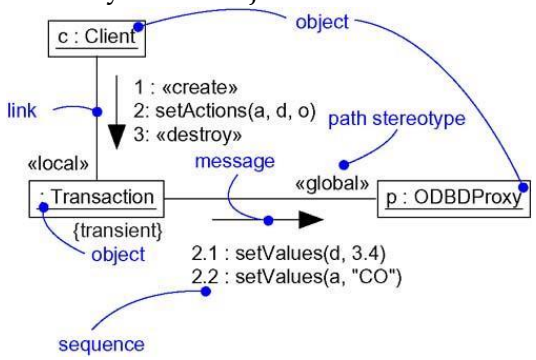


- First, there is the object lifeline. An object lifeline is the vertical dashed line that represents the existence of an object over a period of time.

- Most objects that appear in an interaction diagram will be in existence for the duration of the interaction, so these objects are all aligned at the top of the diagram, with their lifelines drawn from the top of the diagram to the bottom.
- Objects may be created during the interaction. Their lifelines start with the receipt of the message stereotyped as create.
- Objects may be destroyed during the interaction. Their lifelines end with the receipt of the message stereotyped as destroy (and are given the visual cue of a large X, marking the end of their lives).

Collaboration Diagram:

- A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.
- A collaboration diagram shows a set of objects, links among those objects, and messages sent and received by those objects.

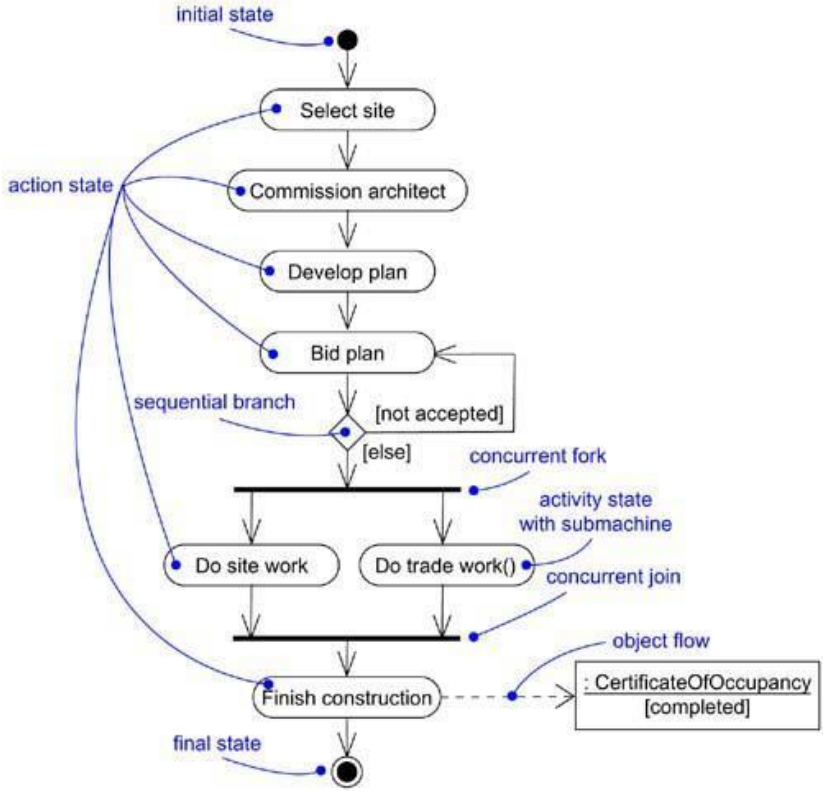


- First, there is the path. To indicate how one object is linked to another, you can attach a path stereotype to the far end of a link (such as<<local>>, indicating that the designated object is local to the sender).
- Typically, you will only need to render the path of the link explicitly for local, parameter, global, and self (but not association) paths.
- Second, there is the sequence number. To indicate the time order of a message, you prefix the message with a number (starting with the message numbered 1), increasing monotonically for each new message in the flow of control (2, 3, and so on).
- To show nesting, you use Dewey decimal numbering (1 is the first message; 1.1 is the first message nested in message 1; 1.2 is the second message nested in message 1; and so on).
- You can show nesting to an arbitrary depth. Note also that, along the same link, you can show many messages (possibly being sent from different directions), and each will have a unique sequence number.

❖ Activity Diagrams:

- An activity diagram shows the flow from activity to activity. An is an ongoing nonatomic execution within a state machine.

- Activities ultimately result in some action, which is made up of executable atomic computations that result in a change in state of the system or the return of a value.
- Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation, such as evaluating an expression.
- Activity diagrams may stand alone to visualize, specify, construct, and document the dynamics of a society of objects, or they may be used to model the flow of control of an operation.

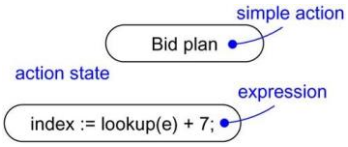


Activity diagrams commonly contain

- Activity states and action states
- Transitions ·
- Objects

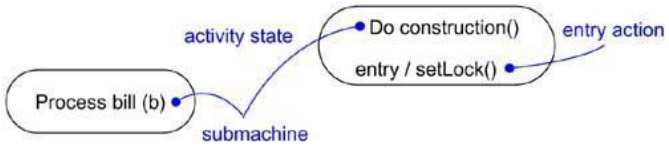
✓ **Action States and Activity States :**

Executable, atomic computations are called Action Statesbecause they are states of the system, each representing the execution of an action



- Figure shows representation of an action state using a lozenge shape (a symbol with horizontal top and bottom and convex sides).

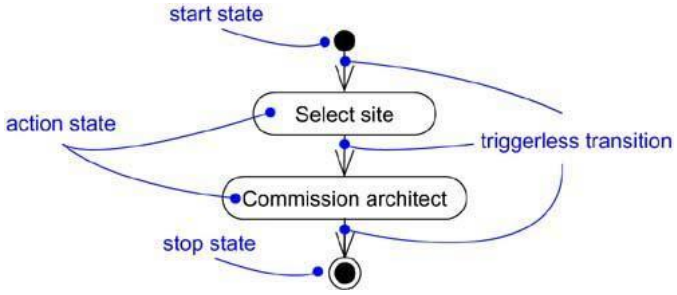
- Action states can't be decomposed. Furthermore, action states are atomic, meaning that events may occur, but the work of the action state is not interrupted.
- Activity states are not atomic, meaning that they may be interrupted and, in general, are considered to take some duration to complete.
- Activity states can be further decomposed.



- Figure above shows that there's no notational distinction between action and activity states, except that an activity state may have additional parts, such as entry and exit actions (actions which are involved on entering and leaving the state, respectively) and submachine specifications.

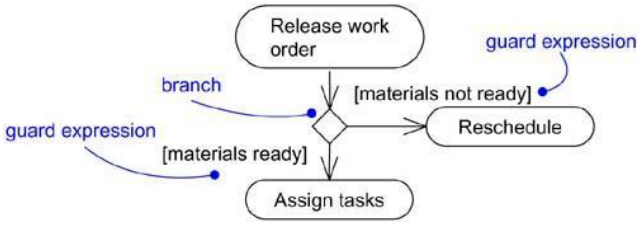
✓ **Transitions :**

A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.



✓ **Branching :**

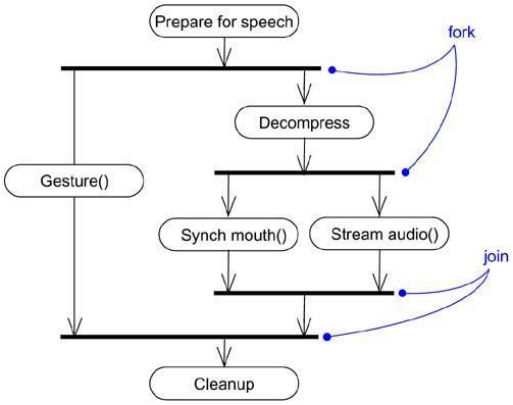
Branches are a notational convenience, semantically equivalent to multiple transitions with guards.



✓ **Forking and Joining :**

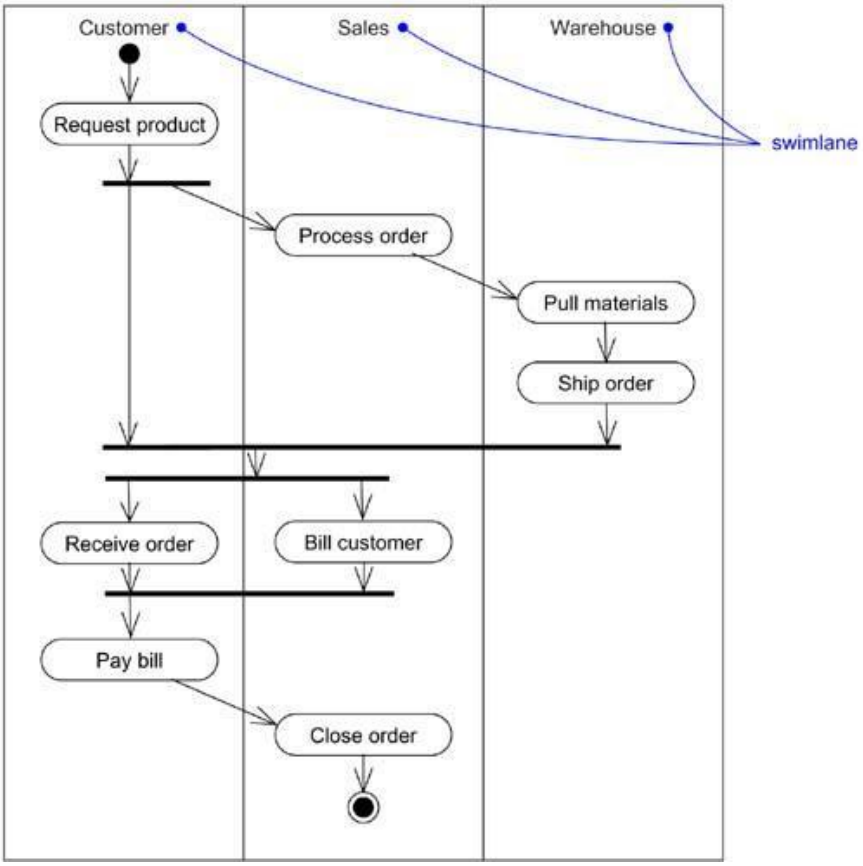
- A fork represents the splitting of a single flow of control into two or more concurrent flows of control.
- A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control.
- A join represents the synchronization of two or more concurrent flows of control. A join may have two or more incoming transitions and one outgoing transition.

- Above the join, the activities associatedwith each of these paths continues in parallel.
- Joinsand forks should balance, meaning that the number of flows that leave afork should match the number of flows that enter its corresponding join.



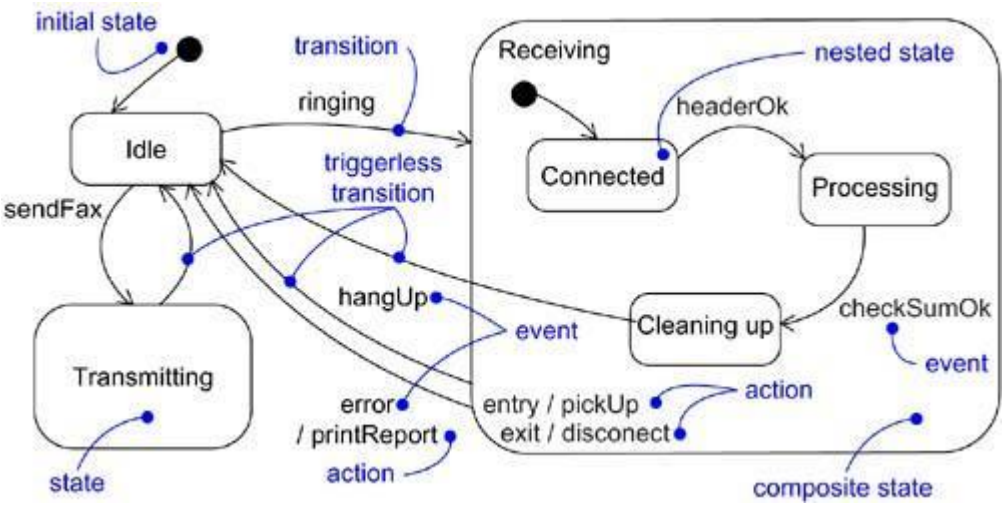
✓ **Swimlanes :**

- At times we find it useful, to partition the activity states on an activity diagram intogroups, each group representing the business organization responsible forthose activities.
- Each such group is called a swimlane because, visually, each group is divided from its neighbor by a vertical solid line, as shown in Figure.
- A swimlane specifies a locus of activities.Acitivity diagrams are used
    - To model a workflow
    - To model an operation



❖ Statechart Diagrams:

- Statechart diagram shows flow of control from state to state. Statechart diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems.
- A statechart diagram shows a state machine, consisting of states, transitions, events, and activities. Statechart diagrams address the dynamic view of a system.
- They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.
- Statechart diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering.



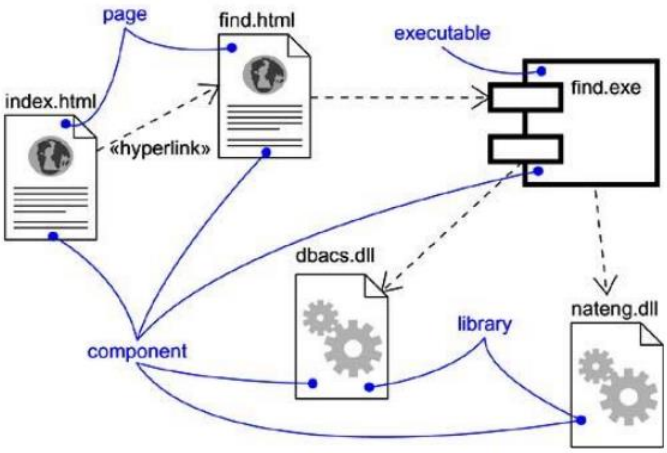
- A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- A **state** is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- An **event** is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition.
- A **transition** is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- An **activity** is ongoing nonatomic execution within a state machine.
- An **action** is an executable atomic computation that results in a change in state of the model or the return of a value. Graphically, a statechart diagram is a collection of vertices and arcs.
- Statechart diagrams commonly contain
  - Simple states and composite states
  - Transitions, including events and action

❖ Component Diagrams:

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

- A component diagram shows the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system.

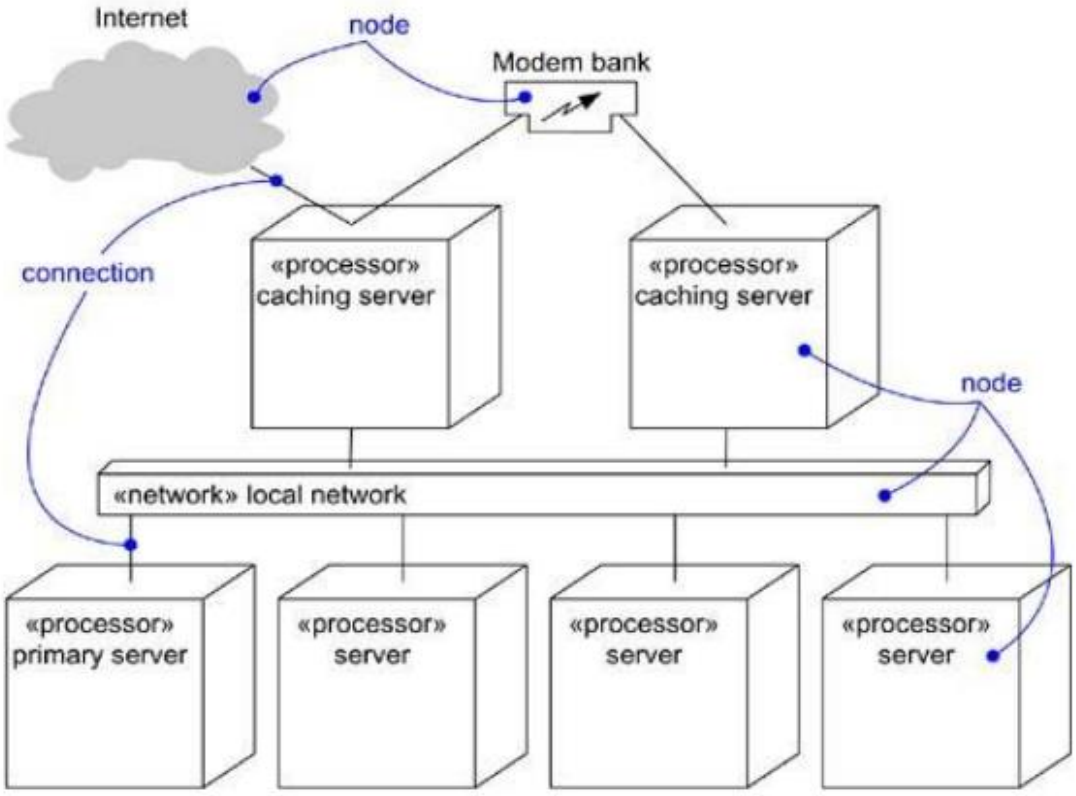
- They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.
- In many ways, components are like classes: Both have names; both may realize a set of interfaces; both may participate in dependency, generalization, and association relationships; both may be nested; both may have instances; both may be participants in interactions.
- However, there are some significant differences between components and classes.
    - Classes represent logical abstractions; components represent physical things that live in the world of bits. In short, components may live on nodes, classes may not.
    - Components represent the physical packaging of otherwise logical components and are at a different level of abstraction.
    - Classes may have attributes and operations directly.
    - In general, components only have operations that are reachable only through their interfaces.
- A component diagram shows a set of components and their relationships. Graphically, a component diagram is a collection of vertices and arcs.



- Component diagrams commonly contain
    - Components
    - Interfaces
    - Dependency, generalization, association, and realization relationships
- Common Uses of Component Diagram
    - To model source code
    - To model executable releases
    - To model physical databases
    - To model physical databases

❖ Deployment Diagrams:

- 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.
- They are related to component diagrams in that a node typically encloses one or more components.
- Graphically, a deployment diagram is a collection of vertices and arcs.
- Deployment diagrams are one of the two kinds of diagrams used in modeling the physical aspects of an object-oriented system.



- In the diagram,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. Graphically, a node is rendered as a cube.

- Deployment diagrams commonly contain
  - Nodes
  - Dependency and association relationships

- Deployment diagrams are used in one of three ways.
  1. To model embedded systems
  2. To model client/server systems
  3. To model fully distributed systems

UNIT-V

❖ Overview of Testing:

Software Testing is evaluation of the software against requirements gathered from users and system specifications. Testing is conducted at the phase level in software development life cycle or at module level in program code. Software testing comprises of Validation and Verification.

Validation:

Validation is process of examining whether or not the software satisfies the user requirements. It is carried out at the end of the SDLC. If the software matches requirements for which it was made, it is validated.

- Validation ensures the product under development is as per the user requirements.
- Validation answers the question – "Are we developing the product which attempts all that user needs from this software ?".
- Validation emphasizes on user requirements.

Verification:

Verification is the process of confirming if the software is meeting the business requirements, and is developed adhering to the proper specifications and methodologies.

- Verification ensures the product being developed is according to design specifications.
- Verification answers the question– "Are we developing this product by firmly following all design specifications ?"
- Verifications concentrates on the design and system specifications.

Classification of Software Testing:

Software Testing can be broadly classified into two types:

- **Manual**:
  - This testing is performed without taking help of automated testing tools. The software tester prepares test cases for different sections and levels of the code, executes the tests and reports the result to the manager.
  - Manual testing is time and resource consuming. The tester needs to confirm whether or not right test cases are used. Major portion of testing involves manual testing.
- **Automated**:
  This testing is a testing procedure done with aid of automated testing tools. The limitations with manual testing can be overcome using automated test tools.

Testing approaches:

Tests can be conducted based on two approaches –

- **Functionality testing/Black-box testing:**

➢ The technique of testing in which the tester doesn't have access to the source code of the software and is conducted at the software interface without any concern with the internal logical structure of the software is known as black-box testing.

➢ It is carried out to test functionality of the program. It is also called 'Behavioral' testing.

➢ In this testing method, the design and structure of the code are not known to the tester, and testing engineers and end users conduct this test on the software.

Black-box testing techniques:

i. **Equivalence class** :
The input is divided into similar classes. If one element of a class passes the test, it is assumed that all the class is passed.

ii. **Boundary values** :
The input is divided into higher and lower end values. If these values pass the test, it is assumed that all values in between may pass too.

iii. **Cause-effect graphing**:
In both previous methods, only one input value at a time is tested. Cause (input) – Effect (output) is a testing technique where combinations of input values are tested in a systematic way.

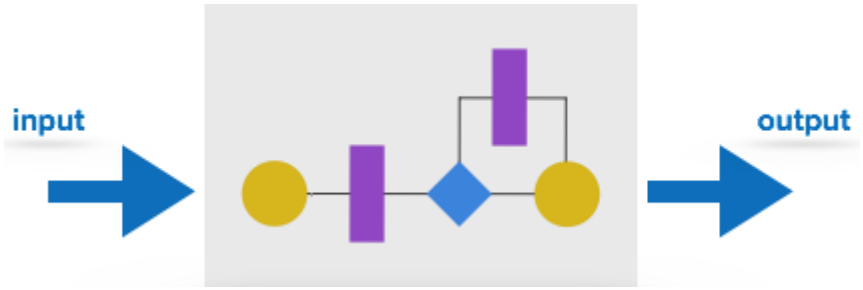
iv. **Pair-wise Testing** :
The behavior of software depends on multiple parameters. In pairwise testing, the multiple parameters are tested pair-wise for their different values.

v. **State-based testing** :
The system changes state on provision of input. These systems are tested based on their states and input.

- **Implementation testing/White-box testing:**

➢ The technique of testing in which the tester is aware of the internal workings of the product, has access to its source code, and is conducted by making sure that all internal operations are performed according to the specifications is known as white box testing.

➢ It is conducted to test program and its implementation, in order to improve code efficiency or structure. It is also known as 'Structural' testing.



- In this testing method, the design and structure of the code are known to the tester. Programmers of the code conduct this test on the code.

White-box testing techniques:

i. **Control-flow testing** - The purpose of the control-flow testing to set up test cases which covers all statements and branch conditions. The branch conditions are tested for both being true and false, so that all statements can be covered.

ii. **Data-flow testing** - This testing technique emphasis to cover all the data variables included in the program. It tests where the variables were declared and defined and where they were used or changed.

❖ Types of Testing:

There are different types of testing few of them are give as:

- ✓ Unit Testing
- ✓ Integration Testing
- ✓ Function Testing
- ✓ Structural Testing
- ✓ Use Case/Scenario Based Testing
- ✓ Regression Testing
- ✓ Performance Testing
- ✓ System Testing
- ✓ Acceptance Testing
- ✓ Installation Testing

1. Unit Testing:

- Unit Testing is defined as a type of software testing where individual components of a software are tested.
- In unit testing, the aim is to test whether the components behave according to the requirements.
- The purpose is to validate that each unit of the software performs as designed.
- Unit Testing of software product is carried out during the development of an application.
- An individual component may be either an individual function or a procedure.

- Unit Testing is typically performed by the developer.

2. **Integration Testing:**
   - Integration testing deals with testing individual components or modules after they are combined in a group.
   - It is performed in an integrated hardware and software environment to ensure that the entire system functions properly.
   - Once all the modules have been unit tested, integration testing is performed.
   - The purpose of the integration testing is to expose faults in the interaction between integrated units.
   - Integration testing is of four types:

     (i) Top-down (ii) Bottom-up

     (iii) Sandwich (iv) Big-Bang
3. **Function Testing:**
   - Functional Testing is a type of software testing in which the system is tested against the functional requirements and specifications.
   - The purpose of Functional tests is to test each function of the software application, by providing appropriate input, verifying the output against the Functional requirements.
   - Functional testing ensures that the requirements or specifications are properly satisfied by the application.
   - Functional testing mainly involves black box testing.
   - It is not concerned about the source code of the application.
   - The testing can be done either manually or using automation.
   - This type of testing is particularly concerned with the result of processing.
   - This testing checks User Interface, APIs, Database, Security, Client/Server communication and other functionality of the Application Under Test.
4. **Structural Testing**:
   - Structural testing is a type of software testing which uses the internal design of the software for testing.
   - In other words the software testing which is performed by the team which knows the development phase of the software, is known as structural testing
   - Structural testing is basically related to the internal design and implementation of the software.
   - It basically tests different aspects of the software according to its types.
   - Structural testing is just the opposite of behavioral testing.
5. **Use Case/Scenario Based Testing:**
   - Use Case Testing is a software testing technique that helps to identify test cases that cover entire system on a transaction by transaction basis from start to end.

- Test cases are the interactions between users and software application.
- Use case testing helps to identify gaps in software application that might not be found by testing individual software components.
- Use Case Testing is a functional black box testing technique.
- Scenario testing is performed to ensure that the end to end functioning of software and all the process flow of the software are working properly.

6. **Regression Testing :**
   - Regression Testing is the process of testing the modified parts of the code and the parts that might get affected due to the modifications to ensure that no new errors have been introduced in the software after the modifications have been made.
   - This type of testing makes sure that the whole component works properly even after adding components to the complete program.
   - Regression Testing is nothing but a full or partial selection of already executed test cases which are re-executed to ensure existing functionalities work fine.
   - Regression testing is a black box testing techniques.
   - Regression tests are also known as the Verification Method.
7. **Performance Testing:**
   - Performance Testing is a type of software testing that ensures software applications to perform properly under their expected workload.
   - Performance Testing is the process of analyzing the quality and capability of a product.
   - It is a testing method performed to determine the system performance in terms of speed, reliability and stability under varying workload.
   - Performance testing is also known as Perf Testing, load testing.
   - It is designed to test the run-time performance of software within the context of an integrated system.
8. **System Testing:**
   - System Testing is a level of testing that validates the complete and fully integrated software product.
   - In system testing, integration testing passed components are taken as input.
   - System testing detects defects within both the integrated units and the whole system.
   - This software is tested such that it works fine for the different operating systems.
   - It is covered under the black box testing technique
   - It mainly focus on the required input and output without focusing on internal working.

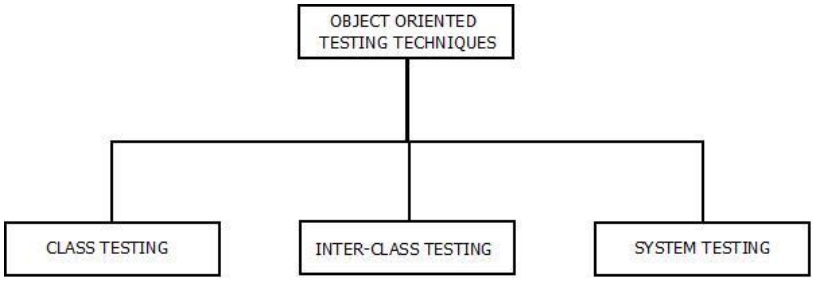
9. **Acceptance Testing:**
   - Acceptance Testing is a method of software testing where a system is tested for acceptability.
   - Acceptance testing ensures that the end-user (customers) can achieve the goals set in the business requirements, which determines whether the software is acceptable for delivery or not.
   - Acceptance Testing is the last phase of software testing performed after System Testing and before making the system available for actual use.
   - It is also known as user acceptance testing (UAT).
   - It is kind of black box testing where two or more end-users will be involved.
10. **Installation Testing:**
   - Installation Test is performed to check if a product works according to expectations after installation.
   - The installation testing ensures that the software application has been successfully installed with all its inherent features or not.
   - In this installation testing checking full or partial upgrades and other features install/uninstall processes are included.
   - It is also named as implementation testing.
   - Mainly done in the end phase.
   - Testing the procedures to achieve an installed software system that can be used are known as installation testing.

❖ Object Oriented Testing :

- **Whenever large scale systems are designed, object oriented testing is done rather than the conventional testing strategies as the concepts of object oriented programming is way different from that of conventional ones.**
- **The whole object oriented testing revolves around the fundamental entity known as "class".**
- **With the help of "class" concept, larger systems can be divided into small well defined units which may then be implemented separately.**
- **The object oriented testing can be classified as like conventional systems. These are called as the levels for testing.**

Object Oriented Testing : Levels / Techniques

- **The levels of object oriented testing can be broadly classified into three categories. These are:**



Object Oriented Testing : Techniques

1. **Class Testing:**
   - **Class testing is also known as unit testing.**
   - **In class testing, every individual classes are tested for errors or bugs.**
   - **Class testing ensures that the attributes of class are implemented as per the design and specifications. Also, it checks whether the interfaces and methods are error free of not.**
2. **Inter-Class Testing:**
   - **It is also called as integration or subsystem testing.**
   - **Inter class testing involves the testing of modules or sub-systems and their coordination with other modules.**
3. **System Testing:**
   - **In system testing, the system is tested as whole and primarily functional testing techniques are used to test the system. Non-functional requirements like performance, reliability, usability and test-ability are also tested.**

❖ Object oriented Testing strategies:

There are two strategies:

- Unit Testing in the Object Oriented context
- Integration Testing in the Object Oriented context

A. Unit Testing in the Object Oriented context :

- When object–oriented software is considered, the concept of the unit changes.
- Encapsulation drives the definition of classes and objects.
- An encapsulated class is usually the focus of unit testing.
- However, Operations (methods) within the class are the smallest testable units.
- As a class can contain a number of different classes, the tactics applied to unit testing must change.

B. Integration Testing in the Object Oriented context:

- As object–oriented software does not have hierarchical control structure, traditional top–down and bottom–up integration strategies.
- There are two different strategies for integration testing of OO systems:
  - thread based testing,

- use-based testing

✓ **Thread based testing:**

- thread based testing integrates the set of classes required to respond to one input or event for the system.
- Each thread is integrated and tested individually.

✓ **Use-based testing:**

- use-based testing begins the construction of the system by testing classes/independent class
- After the independent classes are tested, the next layers of classes/dependent classes, that use the independent classes are tested.
- This sequence of testing layers of dependent classes continues until the entire system is constructed.

❖ Test case design for OO software:

- A good test case design technique is crucial to improving the quality of the Software testing process.
- This helps to improve the overall quality and effectiveness of the released software.
- Test case design refers to how you set-up your test cases. The main purpose of test case design techniques is to test the functionalities and features of the software with the help of effective test cases.
- It is important that the tests are designed well, or it could fail to identify bugs and defects in software during testing.
- There are many different test case design techniques used to test the functionality and various features of your software.
- The test case design techniques are broadly classified into three major categories.

A. Specification-Based or Black-Box techniques:

The technique enables testers to develop test cases that provide full test coverage. The Specification-based or black box test case design techniques are divided further into 5 categories. These categories are as follows:

i. **Boundary Value Analysis (BVA):**

- This technique is applied to explore errors at the boundary of the input domain.
- BVA catches any input errors that might interrupt with the proper functioning of the program.

ii. **Equivalence Partitioning (EP):**

- In Equivalence Partitioning, the test input data is partitioned into a number of classes having an equivalent number of data.
- The test cases are then designed for each class or partition.
- This helps to reduce the number of test cases.

iii. **Decision Table Testing:**

- In this technique, test cases are designed on the basis of the Decision Tables that are formulated using different combinations of inputs and their corresponding outputs based on various conditions and scenarios adhering to different business rules.

iv. **State Transition Diagrams:**

- In this technique, the software under test is perceived as a system having a finite number of states of different types.
- The transition from one state to another is guided by a set of rules.
- The rules define the response to different inputs.
- This technique can be implemented on the systems which have certain workflows within them.

v. **Use Case Testing:**

- A use case is a description of a particular use of the software by a user.
- In this technique, the test cases are designed to execute different business scenarios and end-user functionalities.
- Use case testing helps to identify test cases that cover the entire system.

B. Structure-Based or White-Box techniques:

- The structure-based or white-box technique design test cases based on the internal structure of the software.
- This technique exhaustively tests the developed code.
- Developers who have complete information of the software code, its internal structure, and design help to design the test cases.
- This technique is further divided into five categories.

i. **Statement Testing & Coverage:**

- This technique involves execution of all the executable statements in the source code at least once.
- The percentage of the executable statements is calculated as per the given requirement.
- This is the least preferred metric for checking test coverage.

ii. **Decision Testing Coverage:**

- This technique is also known as branch coverage
- This is a testing method in which each one of the possible branches from each decision point is executed at least once to ensure all reachable code is executed.
- This helps to validate all the branches in the code.
- This helps to ensure that no branch leads to unexpected behavior of the application.

iii. **Condition Testing:**

- Condition testing also is known as Predicate coverage testing.
- In this technique each Boolean expression is predicted as TRUE or FALSE.
- All the testing outcomes are at least tested once.
- This type of testing involves 100% coverage of the code.
- The test cases are designed as such that the condition outcomes are easily executed.

iv. **Multiple Condition Testing:**

- The purpose of Multiple condition testing is to test the different combination of conditions to get 100% coverage.
- To ensure complete coverage, two or more test scripts are required which requires more efforts.

v. **All Path Testing**

- In this technique, the source code of a program is leveraged to find every executable path.
- This helps to determine all the faults within a particular code.

C. Experience-Based techniques:

- These techniques are highly dependent on tester's experience to understand the most important areas of the software.
- The outcomes of these techniques are based on the skills, knowledge, and expertise of the people involved.
- The types of experience-based techniques are as follows:

i. **Error Guessing:**

- In this technique, the testers anticipate errors based on their experience, availability of data and their knowledge of product failure.
- Error guessing is dependent on the skills, intuition, and experience of the testers.

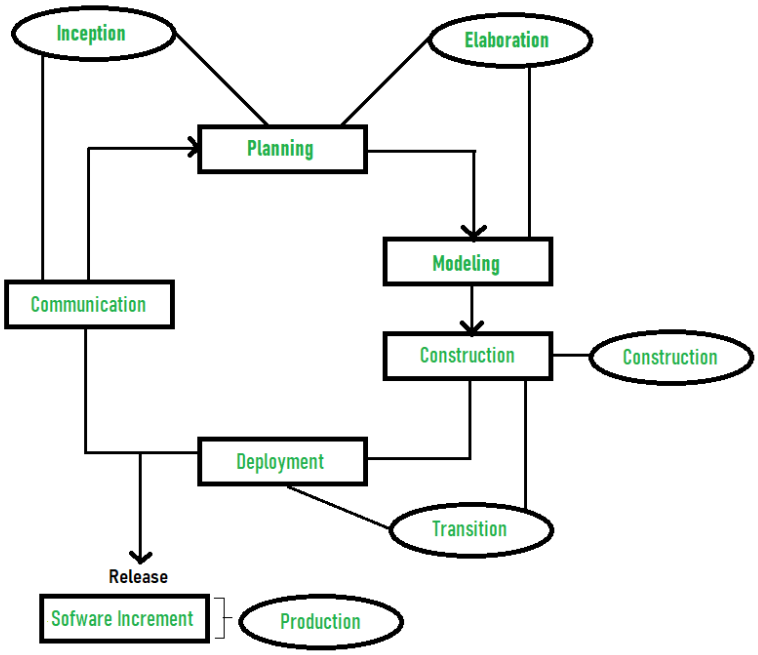
ii. **Exploratory Testing:**

- This technique is used to test the application without any formal documentation.
- There is minimum time available for testing and maximum for test execution.
- In Exploratory Testing**,** the test design and test execution are performed concurrently.

➢ Rational Unified Process (RUP) or Unified Process Model

Rational Unified Process (RUP) is a software development process for object-oriented models. It is also known as the Unified Process Model. It is created by Rational corporation and is designed and documented using UML (Unified Modeling Language). This process is included in IBM Rational Method Composer (RMC) product. IBM (International Business Machine Corporation) allows us to customize, design, and personalize the unified process. RUP is proposed by Ivar Jacobson, Grady Bootch, and James Rambaugh. Some characteristics of RUP include use-case driven, Iterative (repetition of the process), and Incremental (increase in value) by nature, delivered online using web technology, can be customized or tailored in modular and electronic form, etc. RUP reduces unexpected development costs and prevents wastage of resources.

Phases of RUP: There is total of five phases of the life cycle of RUP:



1. **Inception –**

- Communication and planning are the main ones.
- Identifies the scope of the project using a use-case model allowing managers to estimate costs and time required.
- Customers' requirements are identified and then it becomes easy to make a plan for the project.
- The project plan, Project goal, risks, use-case model, and Project description, are made.
- The project is checked against the milestone criteria and if it couldn't pass these criteria then the project can be either canceled or redesigned.

2. **Elaboration –**

- Planning and modeling are the main ones.

- A detailed evaluation and development plan is carried out and diminishes the risks.
- Revise or redefine the use-case model (approx. 80%), business case, and risks.
- Again, checked against milestone criteria and if it couldn't pass these criteria then again project can be canceled or redesigned.
- Executable architecture baseline.

3. **Construction –**
   - This phase of RUP often takes the longest because you create, write, collaborate and test your software and applications, focusing on the features and components of the system and how well they function.
   - You typically start by incrementally expanding upon the baseline architecture, building code and software until it's complete.
   - You manage costs and quality in this phase, intending to produce a completed software system and user manual.
   - Review the software user stability and transition plan before ending the RUP construction phase.
4. **Transition –**

   - The transition stage releases the project to the user, whether that's the public or internal users like employees.
   - A transition phase is rarely perfect and often includes making system adjustments based on practical and daily usage.
   - Ensuring a smooth transition and rectifying software issues timely can help make this stage a success.

Elements often involved in the transition period include:

- *Beta testing*
- *Education and training*
- *Deployment and data analytics*
- *Collection of user feedback*

5. **Production –**
   - This last phase of the RUP process includes software deployment, intending to gain user acceptance.
   - You maintain and update the software accordingly, often based on feedback from people who use the software, app, program or platform.

This last stage usually includes:

- *Packaging, distribution and installation*
- *User help and assistance platform availability*
- *Data migration*
- *Continued user acceptance initiatives*

Advantages:

1. It provides good documentation, it completes the process in itself.
2. It provides risk-management support.
3. It reuses the components, and hence total time duration is less.
4. Good online support is available in the form of tutorials and training.

Disadvantages:

1. Team of expert professional is required, as the process is complex.
2. Complex and not properly organized process.
3. More dependency on risk management.
4. Hard to integrate again and again.

(unit-3)

➢ UML Class Diagram

The **UML** Class diagram is a graphical notation used to construct and visualize object oriented systems. A class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's:

- classes,
- their attributes,
- operations (or methods),
- and the relationships among objects.

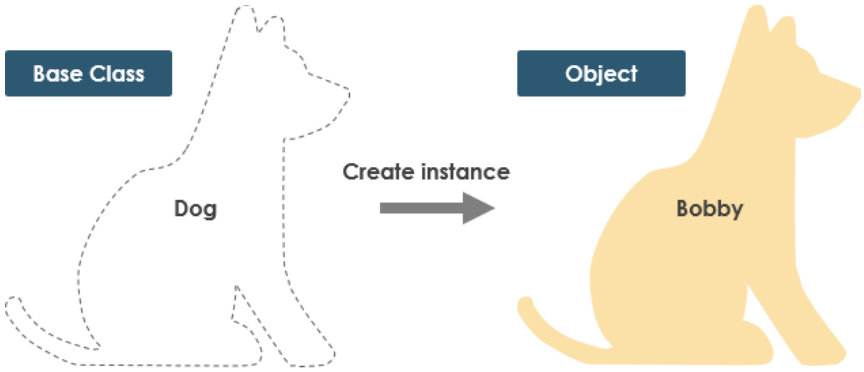
What is a Class?

A Class is a blueprint for an object. Objects and classes go hand in hand. We can't talk about one without talking about the other. And the entire point of Object-Oriented Design is not about objects, it's about classes, because we use classes to create objects. So a class describes what an object will be, but it isn't the object itself.

In fact, classes describe the type of objects, while objects are usable instances of classes. Each Object was built from the same set of blueprints and therefore contains the same components (properties and methods). The standard meaning is that an object is an instance of a class and object - Objects have states and behaviors.

Example

A dog has states - color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.



Properties	Methods
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

Property Values	Methods
Color: Yellow	Sit
Eye Color: Brown	Lay Down
Height: 17 in	Shake
Length: 35 in	Come
Weight: 24 pounds	

UML Class Notation

A class represent a concept which encapsulates state (**attributes**) and behavior (**operations**). Each attribute has a type. Each **operation** has a **signature**. *The class name is the **only mandatory information***.

Class Name:

- The name of the class appears in the first partition.

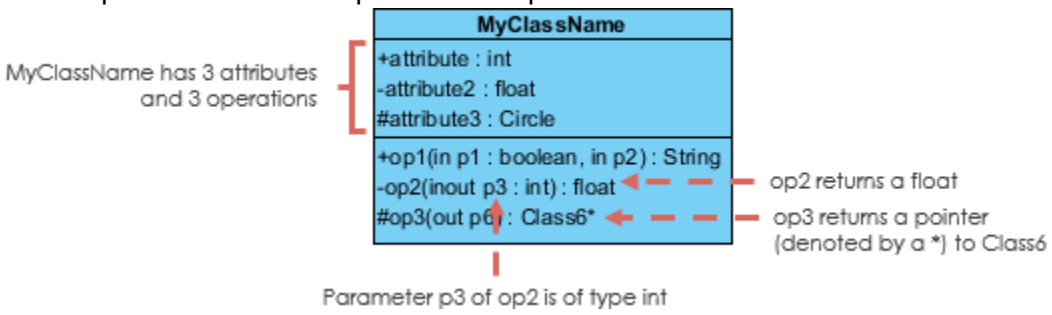
Class Attributes:

- Attributes are shown in the second partition.
- The attribute type is shown after the colon.
- Attributes map onto member variables (data members) in code.

Class Operations (Methods):

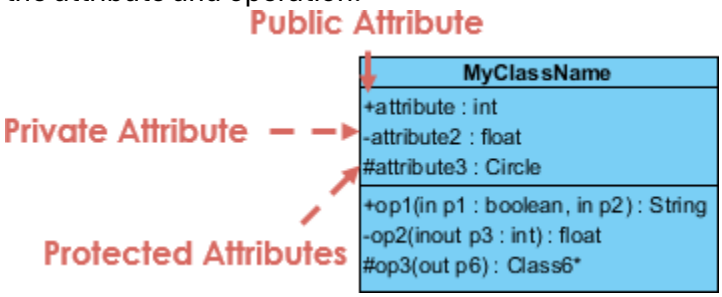
- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.

- The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code



Class Visibility

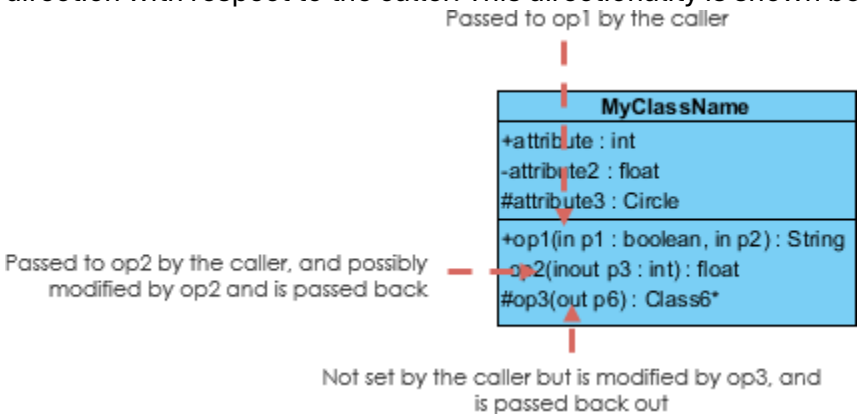
The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.



- \+ denotes public attributes or operations
- \- denotes private attributes or operations
- # denotes protected attributes or operations

Parameter Directionality

Each parameter in an operation (method) may be denoted as in, **out** or **inout** which specifies its direction with respect to the caller. This directionality is shown before the parameter name.



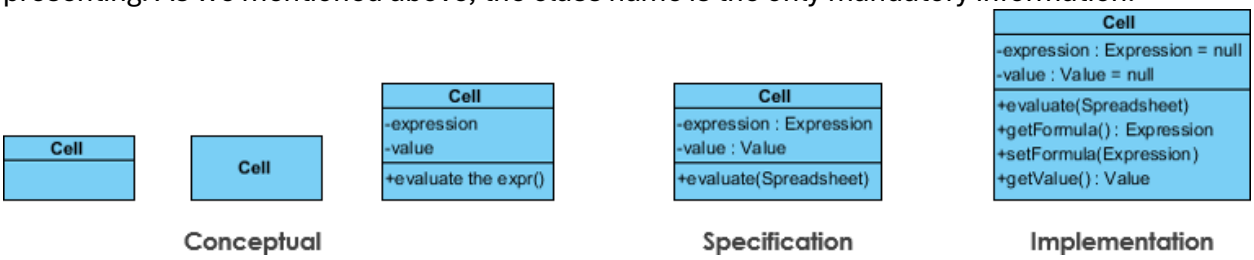
Perspectives of Class Diagram

The choice of perspective depends on how far along you are in the development process. During the formulation of a **domain model**, for example, you would seldom move past the **conceptual perspective**. **Analysis models** will typically feature a mix of **conceptual and specification perspectives**. **Design model** development will typically start with heavy emphasis on the **specification perspective**, and evolve into the **implementation perspective**.

A diagram can be interpreted from various perspectives:

- **Conceptual**: represents the concepts in the domain
- **Specification**: focus is on the interfaces of Abstract Data Type (ADTs) in the software
- **Implementation**: describes how classes will implement their interfaces

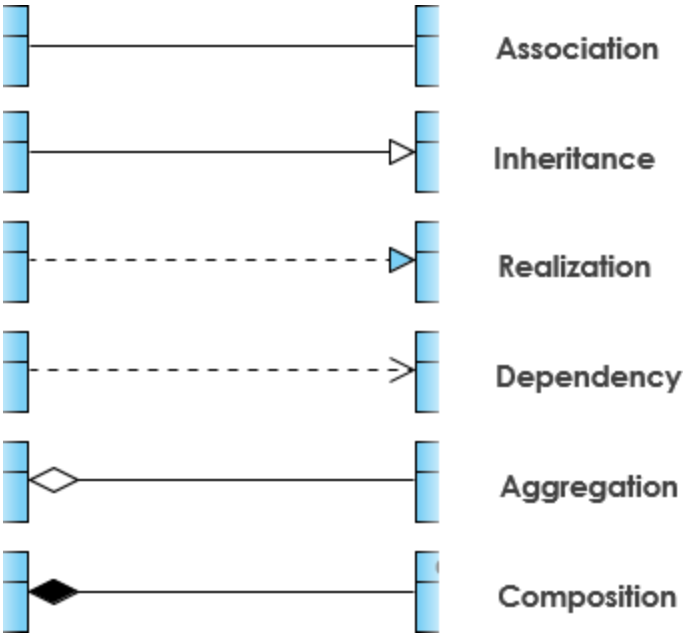
The perspective affects the amount of detail to be supplied and the kinds of relationships worth presenting. As we mentioned above, the class name is the only mandatory information.



➢ Relationships between classes

UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. Can you describe what each of the relationships mean relative to your target programming language shown in the Figure below?

If you can't yet recognize them, no problem this section is meant to help you to understand UML class relationships. A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:

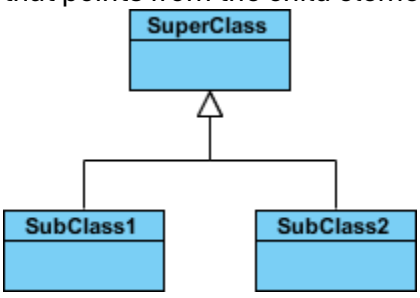


Inheritance (or Generalization):

A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier.

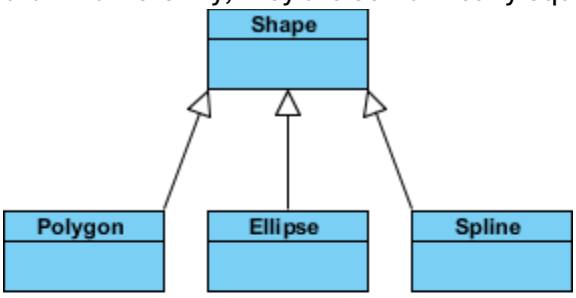
- Represents an "is-a" relationship.
- An abstract class name is shown in italics.
- SubClass1 and SubClass2 are specializations of SuperClass.

The figure below shows an example of inheritance hierarchy. SubClass1 and SubClass2 are derived from SuperClass. The relationship is displayed as a solid line with a hollow arrowhead that points from the child element to the parent element.

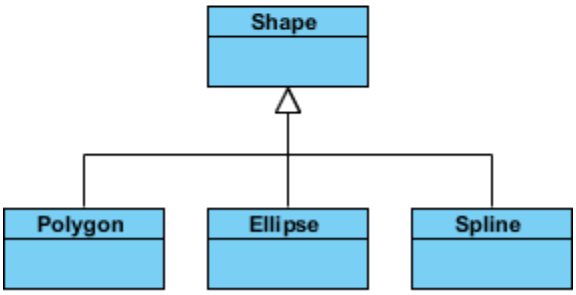


Inheritance Example - Shapes

The figure below shows an inheritance example with two styles. Although the connectors are drawn differently, they are semantically equivalent.



Style 1: Separate target



Style 2: Shared target

Association

Associations are relationships between classes in a UML Class Diagram. They are represented by a solid line between classes. Associations are typically named using a verb or verb phrase which reflects the real world problem domain.

Simple Association

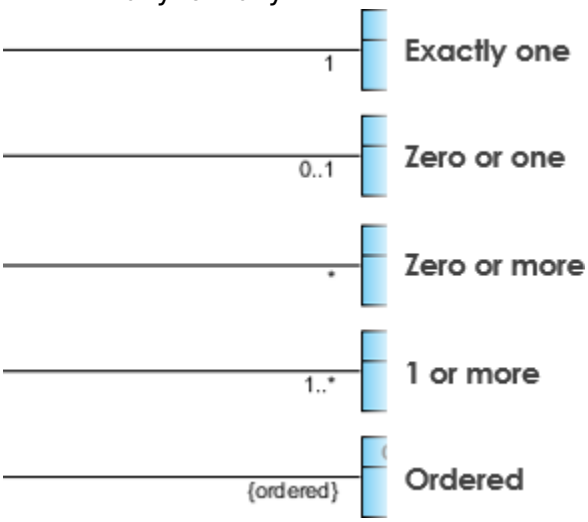
- A structural link between two peer classes.
- There is an association between Class1 and Class2

The figure below shows an example of simple association. There is an association that connects the <<control>> class Class1 and <<boundary>> class Class2. The relationship is displayed as a solid line connecting the two classes.

Cardinality

Cardinality is expressed in terms of:

- one to one
- one to many
- many to many

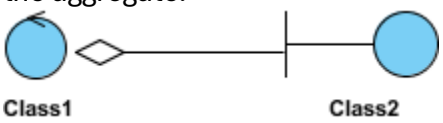


Aggregation

A special type of association.

- It represents a "part of" relationship.
- Class2 is part of Class1.
- Many instances (denoted by the *) of Class2 can be associated with Class1.
- Objects of Class1 and Class2 have separate lifetimes.

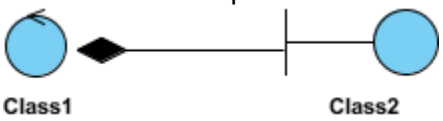
The figure below shows an example of aggregation. The relationship is displayed as a solid line with a unfilled diamond at the association end, which is connected to the class that represents the aggregate.



Composition

- A special type of aggregation where parts are destroyed when the whole is destroyed.
- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.

The figure below shows an example of composition. The relationship is displayed as a solid line with a filled diamond at the association end, which is connected to the class that represents the whole or composite.

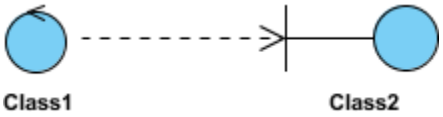


Dependency

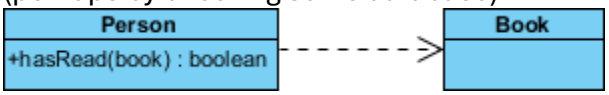
An object of one class might use an object of another class in the code of a method. If the object is not stored in any field, then this is modeled as a dependency relationship.

- A special type of association.
- Exists between two classes if changes to the definition of one may cause changes to the other (but not the other way around).
- Class1 depends on Class2

The figure below shows an example of dependency. The relationship is displayed as a dashed line with an open arrow.



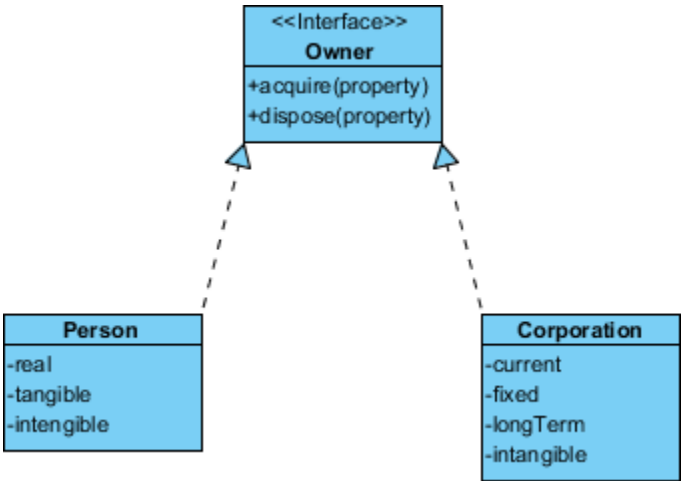
The figure below shows another example of dependency. The Person class might have a hasRead method with a Book parameter that returns true if the person has read the book (perhaps by checking some database).



Realization

Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In other words, you can understand this as the relationship between the interface and the implementing class.

For example, the Owner interface might specify methods for acquiring property and disposing of property. The Person and Corporation classes need to implement these methods, possibly in very different ways.



Class Diagram Example: Order System