CLASS DIAGRAM

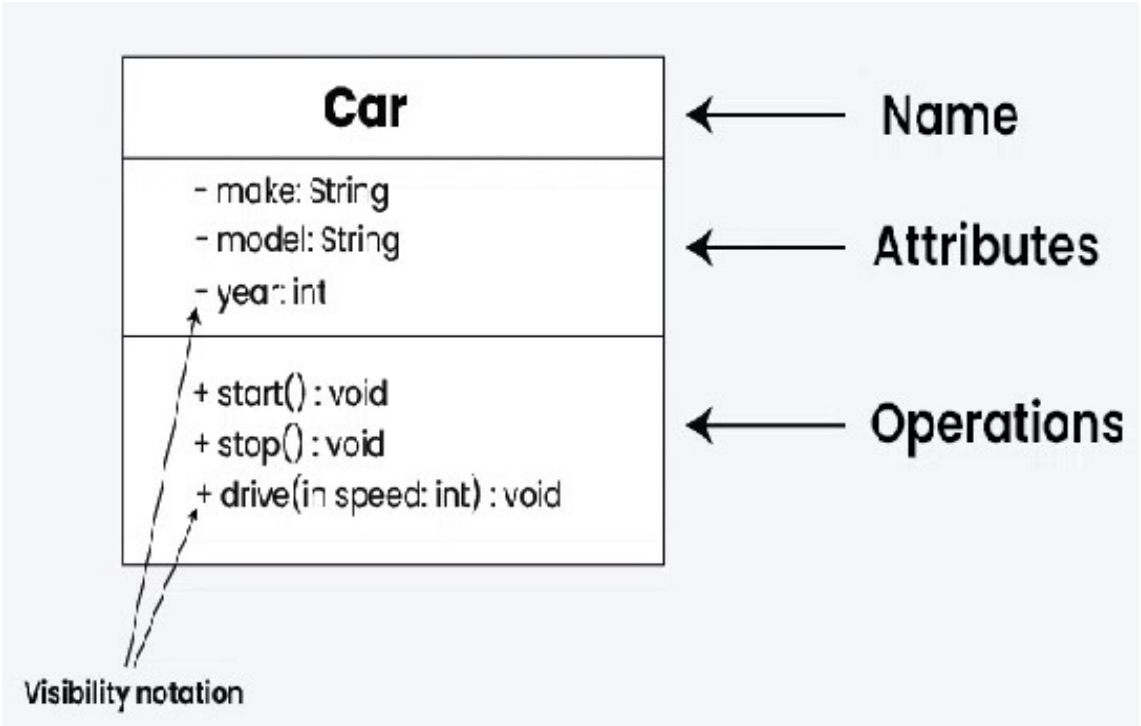
Class diagram is an UML (Unified Modeling Language) diagram used in software engineering to visually represent the structure and relationships of classes within a system.

Class diagrams provide a high-level overview of a system's design, helping to communicate and document the structure of the software.

Class diagrams are used for:

- Modeling the vocabulary of the system
- Modeling the collaborations
- Modeling the logical database schema

classes are depicted as boxes, each containing three compartments for the class name, attributes, and methods. Lines connecting classes illustrate associations, showing relationships such as one-to-one or one-to-many.



1. **Class Name:**
   - The name of the class is typically written in the top compartment of the class box and is centered and bold.

2. **Attributes:**
   - Attributes, also known as properties or fields, represent the data members of the class. They are listed in the second compartment of the class box and often include the visibility (e.g., public, private) and the data type of each attribute.
3. **Methods:**
   - Methods, also known as functions or operations, represent the behavior or functionality of the class. They are listed in the third compartment of the class box and include the visibility (e.g., public, private), return type, and parameters of each method.
4. **Visibility Notation:**
   - Visibility notations indicate the access level of attributes and methods. Common visibility notations include:
     - + for public (visible to all classes)
     - - for private (visible only within the class)
     - # for protected (visible to subclasses)
     - ~ for package or default visibility (visible to classes in the same package)

Purpose of Class Diagrams

The main purpose of using class diagrams is:

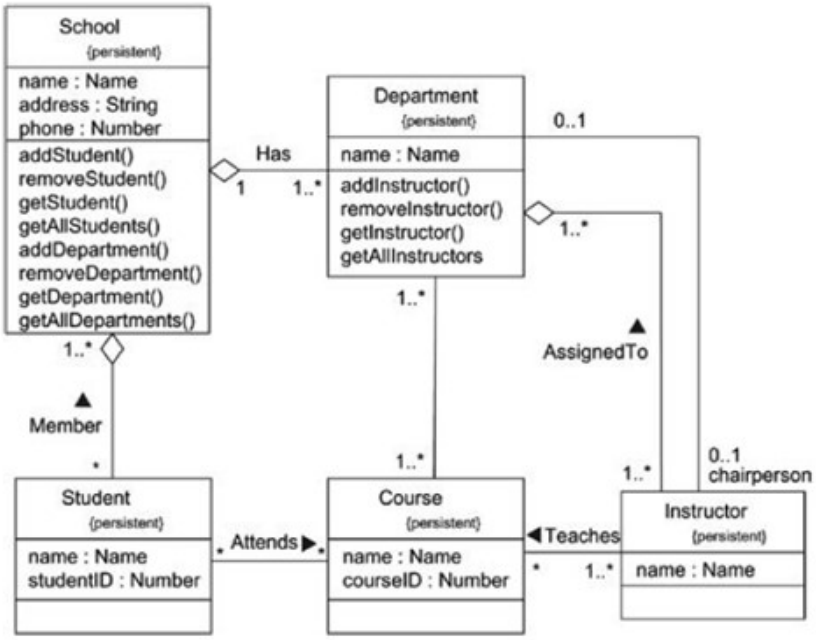
- This is the only UML that can appropriately depict various aspects of the OOPs concept.
- Proper design and analysis of applications can be faster and efficient.
- It is the base for deployment and component diagram.
- It incorporates forward and reverse engineering.

To model simple collaborations,

Identify the function or behavior of the part of a system you would like to model.
For each function or mechanism identify the classes, interfaces, collaborations and relationships between them.
Use scenarios (sequence of activities) to walk through these things.
For example, take a class and fill its responsibilities. Now, convert these responsibilities into attributes and operations.

Modeling a Logical Database Schema
To model a schema,

- Identify the classes whose state must be saved over the lifetime of the application.
- Create a class diagram and mark these classes as persistent by using tagged values.
- Provide the structural details for these classes like the attributes, associations with other classes and the multiplicity.
- Minimize the common patterns which complicate the physical database design like cyclic associations, one-to-one associations and n-ary associations.
- Provide the behavior for these classes by listing out the operations that are important for data access and integrity.
- Wherever possible, use tools to convert the logical design to physical design.



Forward and Reverse Engineering

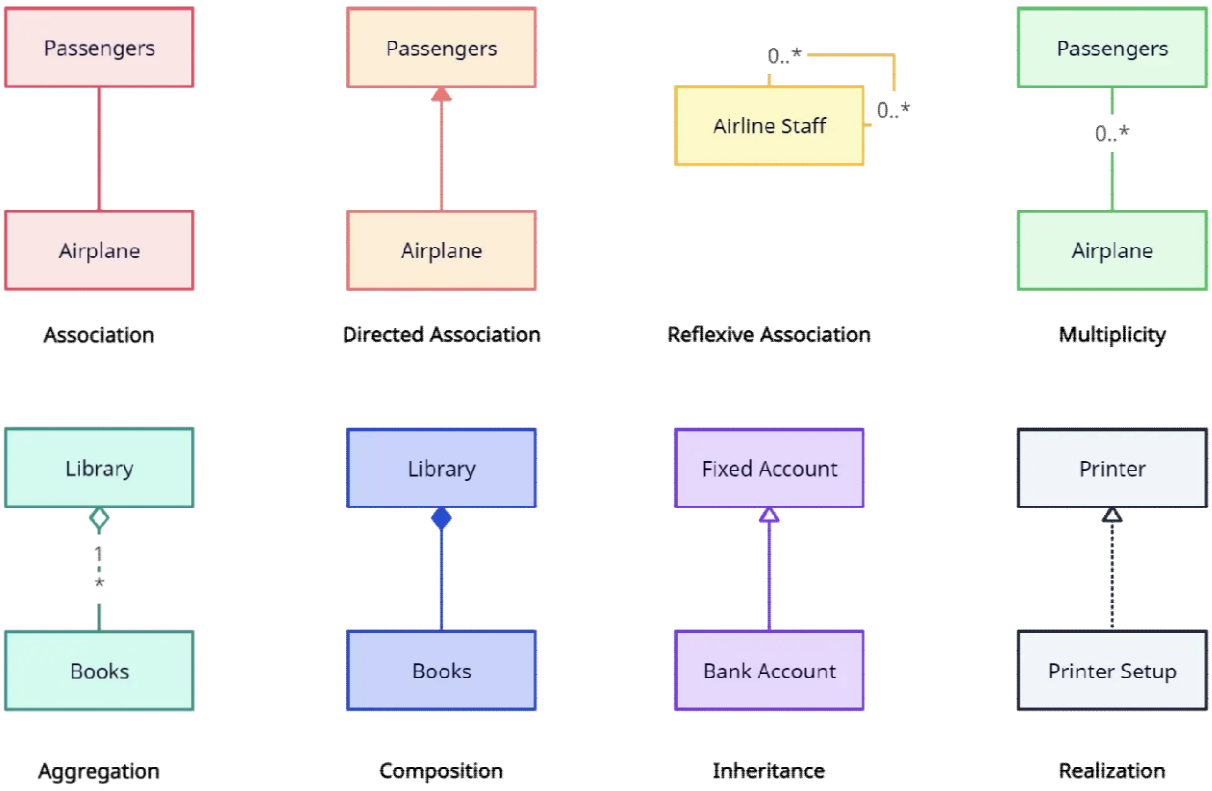
To forward engineer a class diagram,

- Identify the rules for mapping the models to the implementation language of your choice.
- Depending on the semantics of the language, you may want to restrict the information in your UML models. For example, UML supports multiple inheritance. But some programming languages might not allow this.
- Use tagged values to specify the target language.
- Use tools to convert your models to code.

To reverse engineer the code,

- Identify the rules for mapping the implementation language to a model.
- Using a tool, navigate to the code that you want to reverse engineer. Use the tool to generate a new model.

Types of UML Relationships used in Class Diagram



1. Association

An association represents a bi-directional relationship between two classes. It indicates that instances of one class are connected to instances of another class.
Associations are typically depicted as a solid line connecting the classes, with optional arrows indicating the direction of the relationship.

2. Directed Association

A directed association in a UML class diagram represents a relationship between two classes where the association has a direction, indicating that one class is associated with another in a specific way.

- In a directed association, an arrowhead is added to the association line to indicate the direction of the relationship. The arrow points from the class that initiates the association to the class that is being targeted or affected by the association.
- Directed associations are used when the association has a specific flow or directionality, such as indicating which class is responsible for initiating the association or which class has a dependency on another.

3. Aggregation

Aggregation is a specialized form of association that represents a “whole-part” relationship. It denotes a stronger relationship where one class (the whole) contains or is composed of another class (the part).
Aggregation is represented by a diamond shape on the side of the whole class. In this kind of relationship, the child class can exist independently of its parent class.

4. Composition

Composition is a stronger form of aggregation, indicating a more significant ownership or dependency relationship. In composition, the part class cannot exist independently of the whole class.
Composition is represented by a filled diamond shape on the side of the whole class.

5. Generalization (Inheritance)

Inheritance represents an “is-a” relationship between classes, where one class (the subclass or child) inherits the properties and behaviors of another class (the superclass or parent).

Inheritance is depicted by a solid line with a closed, hollow arrowhead pointing from the subclass to the superclass.

6. Realization (Interface Implementation)

Realization indicates that a class implements the features of an interface. It is often used in cases where a class realizes the operations defined by an interface.
Realization is depicted by a dashed line with an open arrowhead pointing from the implementing class to the interface.

Let's consider the scenario where a "Person" and a "Corporation" both realizing an "Owner" interface.

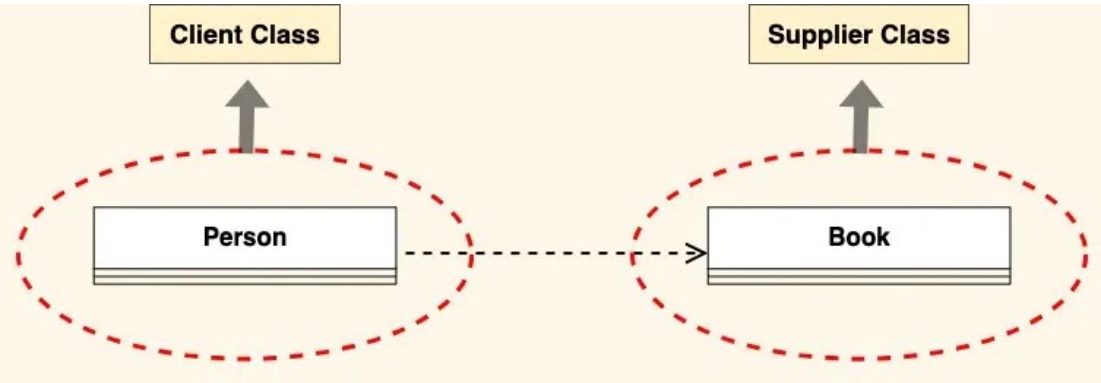
- **Owner Interface:** This interface now includes methods such as "acquire(property)" and "dispose(property)" to represent actions related to acquiring and disposing of property.
- **Person Class (Realization):** The Person class implements the Owner interface, providing concrete implementations for the "acquire(property)" and "dispose(property)" methods. For instance, a person can acquire ownership of a house or dispose of a car.
- **Corporation Class (Realization):** Similarly, the Corporation class also implements the Owner interface, offering specific implementations for the "acquire(property)" and "dispose(property)" methods. For example, a corporation can acquire ownership of real estate properties or dispose of company vehicles.

7. Dependency Relationship

A dependency exists between two classes when one class relies on another, but the relationship is not as strong as association or inheritance.
It represents a more loosely coupled connection between classes. Dependencies are often depicted as a dashed arrow.

Let's consider a scenario where a Person depends on a Book.

- **Person Class:** Represents an individual who reads a book. The Person class depends on the Book class to access and read the content.
- **Book Class:** Represents a book that contains content to be read by a person. The Book class is independent and can exist without the Person class.



8. Usage(Dependency) Relationship

A usage dependency relationship in a UML class diagram indicates that one class (the client) utilizes or depends on another class (the supplier) to perform certain tasks or access certain functionality. The client class relies on the services provided by the supplier class but does not own or create instances of it.

- Usage dependencies represent a form of dependency where one class depends on another class to fulfill a specific need or requirement.
- The client class requires access to specific features or services provided by the supplier class.
- In UML class diagrams, usage dependencies are typically represented by a dashed arrowed line pointing from the client class to the supplier class.
- The arrow indicates the direction of the dependency, showing that the client class depends on the services provided by the supplier class.

The “Car” class may need to access methods or attributes of the “FuelTank” class to check the fuel level, refill fuel, or monitor fuel consumption.

In this case, the “Car” class has a usage dependency on the “FuelTank” class because it utilizes its services to perform certain tasks related to fuel management.

How to draw Class Diagrams

Drawing class diagrams involves visualizing the structure of a system, including classes, their attributes, methods, and relationships. Here are the steps to draw class diagrams:

1. **Identify Classes:**

- Start by identifying the classes in your system. A class represents a blueprint for objects and should encapsulate related attributes and methods.

2. **List Attributes and Methods:**
   - For each class, list its attributes (properties, fields) and methods (functions, operations). Include information such as data types and visibility (public, private, protected).
3. **Identify Relationships:**
   - Determine the relationships between classes. Common relationships include associations, aggregations, compositions, inheritance, and dependencies. Understand the nature and multiplicity of these relationships.
4. **Create Class Boxes:**
   - Draw a rectangle (class box) for each class identified. Place the class name in the top compartment of the box. Divide the box into compartments for attributes and methods.
5. **Add Attributes and Methods:**
   - Inside each class box, list the attributes and methods in their respective compartments. Use visibility notations (+ for public, – for private, # for protected, ~ for package/default).
6. **Draw Relationships:**
   - Draw lines to represent relationships between classes. Use arrows to indicate the direction of associations or dependencies. Different line types or notations may be used for various relationships.
7. **Label Relationships:**
   - Label the relationships with multiplicity and role names if needed. Multiplicity indicates the number of instances involved in the relationship, and role names clarify the role of each class in the relationship.
8. **Review and Refine:**
   - Review your class diagram to ensure it accurately represents the system's structure and relationships. Refine the diagram as needed based on feedback and requirements.
9. **Use Tools for Digital Drawing:**
   - While you can draw class diagrams on paper, using digital tools can provide more flexibility and ease of modification. UML modeling tools, drawing software, or even specialized diagramming tools can be helpful.

Use cases of Class Diagrams

System Design:

- During the system design phase, class diagrams are used to model the static structure of a software system. They help in visualizing and organizing classes, their attributes, methods, and relationships, providing a blueprint for system implementation.

Communication and Collaboration:

- Class diagrams serve as a visual communication tool between stakeholders, including developers, designers, project managers, and clients. They facilitate discussions about the system's structure and design, promoting a shared understanding among team members.

Code Generation:

- Some software development environments and tools support code generation based on class diagrams. Developers can generate code skeletons, reducing manual coding efforts and ensuring consistency between the design and implementation.

Testing and Test Planning:

- Testers use class diagrams to understand the relationships between classes and plan test cases accordingly. The visual representation of class structures helps in identifying areas that require thorough testing.

Reverse Engineering:

- Class diagrams can be used for reverse engineering, where developers analyze existing code to create visual representations of the software structure. This is especially helpful when documentation is scarce or outdated.

Aggregation vs Composition

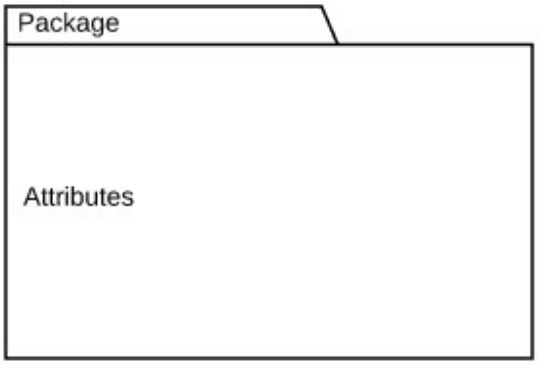
Features	Aggregation	Composition
Dependency	In an aggregation relationship, a child can exist independent of a parent.	In a composition relationship, the child cannot exist independent of the parent.
Type of Relationship	It constitutes a **Has-a** relationship.	It constitutes **Part-of** relationship.

Type of Association	It forms a **weak** association.	It forms a **strong** association.
Examples	A doctor has patients when the doctor gets transfer to another hospital, the patients do not accompany to a new workplace.	A hospital and its wards. If the hospital is destroyed, the wards also get destroyed.

PACKAGE DIAGRAM

A package diagram is a type of Unified Modeling Language (UML) diagram mainly used to represent the organization and the structure

Notation:



Relationships used inn Package Diagram: Dependencies

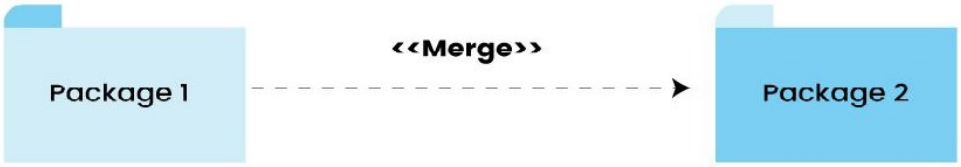
Depicts the relationship between one element (package, named element, etc) and another

Namespace Relationships: Merge, Import, Access

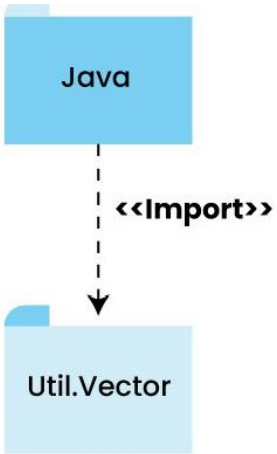
Package:
This basic building block of the Package Diagram is a Package. It is a container for organizing different diagram elements such as classes, interfaces, etc. It is represented in the Diagram in a folder-like icon with its name.

NameSpace:
It represents the package's name in the diagram. It generally appears on top of the package symbol which helps to uniquely identify the package in the diagram.

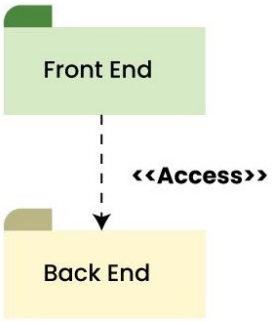
Package Merge:
It is a relationship that signifies how a package can be merged or combined. It is represented as a direct arrow between two packages. Signifying that the contents of one package can be merged with the contents of the other.



Package Import:
It is another relationship that shows one package's access to the contents of a different package. It is represented as a Dashed Arrow.



Access Relationship:
This type of relationship signifies that there is a access relationship between two or more packages, meaning that one package can access the contents of another package without importing it.

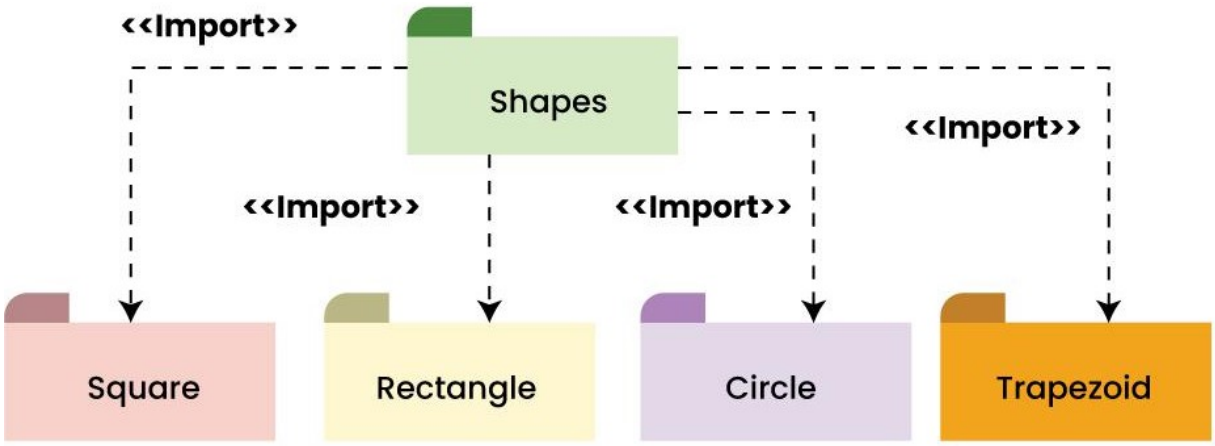


Element:

An element can be a single unit inside of a package, it can be a class, an interface or subsystems.

For example, if we consider a Class, then there can be many functions inside it and many variables or subclass can also be present, all of these are considered as an Element of that Class and they are connected with the Main Class, without the existence of the main class, they will also not exist or have no relevance.

Constraint: It is like a condition or requirement set related to a package. It is represented by curly braces.



Purpose of Package Diagram

- Package Diagram can be used to simplify complex class diagrams, it can group classes into packages.
- A package is a collection of logically related UML elements.
- Packages are depicted as file folders and can be used on any of the UML diagrams.

Use Cases of Package Diagrams

Package Diagram is an essential part of System Design, it is a versatile tool to model and depict the structure of any organization or system's elements. Some of the use cases of Package Diagrams are listed below:

System Structure Visualization:
Package Diagram is helpful to represent the system's structure in a diagrammatic format. It also helps in organizing the elements into smaller and compact packages. This type of visualization help in understanding the structure of the system in a more concise manner.

Module and Component Management:
Package Diagram is useful to organize and manage modules or components within a system. Package acts as a container for items which are related to each other. It helps in grouping and categorizing parts of the system.

Dependency Management:
Developers use Package Diagram to represent dependency between the different packages, this is useful to represent the system's architecture neatly and show the potential impact of the changes.

System Decomposition:
System Architects tend to break down a complex problem into a group of smaller and easily managable problem at the initial stage of the System Design. This helps in designing the components easier and easier implementation.

Versioning and Release Planning:
Project Managers often use Package Diagram to plan release of the product, ensuring that all the newly added components and changes are well understood and coordinated.

Package Diagram Best Practices

Clearly Define the Package Names: It is always recommended to clearly define the package names in such a manner that there is no duplicate present in the same package which might create confusion.

Organize Packages Hierarchically:

Organize the packages in a hierarchical manner so that it represents the proper structure of the system and the relationship of the various parts of the system.

Maintain Modularity:

Make the packages concise, use a single package to represent a single function or element. Don't use overcomplicated and complex packages.

Document Dependencies:

Clearly mention and document the dependencies between the packages using the suitable symbols .This includes dependencies such as associations, generalizations, or dependencies.

Use Colors and Styles Sparingly:

Use different colors and styles to differentiate between different packages, but ensure the used colors are meaningful and consistent.

Benefits of Package Diagram

Clarity and Understanding: It provides a visual representation of the system's architecture, showing that how each elements are organized and interact with each other.

Modularity and Encapsulation: Package Diagram encourages a modular approach to represent any system in form of smaller and easy to understand packages. It also supports the encapsulation of elements into packages which share a same trait.

Communication and Collaboration: It servers as a common language of commmunication between the developers and the stakeholders.

Dependency Identification: By showing the dependency between the different packages, it become easy to identify and manage dependencies between the packages, also it becomes easy to address issues related to coupling and cohesion.

Scalability and Maintainability: Package Diagram highly encourages Scalability and Maintainability due to it's modular nature. Any package can be changed or modified independently without harming the others, but the developers need to keep in mind about the relation or dependency that package has with others and make changes in them too.

Importance of Package Diagram

1. Organizing Large Systems
2. Dependency Management

3. High-Level Overview
4. Reuse of Components
5. Encapsulation
6. Guiding Development
7. Team Collaboration
8. Version Control and Maintenance
9. Software Documentation

.

Challenges of Package Diagrams

Overly Detailed or Abstract: Keeping the balance between providing enough detail and maintaining simplicity and abstraction can be challenging task. Any single package of a Package Diagram must not be overwhelmed with information so that it becomes complex to comprehend.
Dynamic Aspects Missing: The main focus of Package Diagram is Static Structural Aspects of elements of the system. It might not catch the dynamic nature of the system such as the runtime behaviors or interaction between different components of the system.

Overemphasis on Dependencies: While it is a must to show the dependencies amongst the packages in a Package Diagram, putting extra emphasis on them might create clutter and complex diagram, which becomes challenging to understand. Emphasis should be put on to display the dependencies in a clear and concise manner without making the diagram hard to understand
Limited Support for Behavioral Aspects: It is not well suited when it comes to dynamic or behavioral attributes of a system, like the change of behavior of the system during its runtime, or the interaction between different packages during the runtime.

Pitfalls of Package Diagrams
Misinterpretation of Relationships: If the relationships are not defined or represented correctly, some misinterpretation might arise regarding them. This misinterpretation might lead into incorrect assumptions about the system's architecture and behavior.
Inconsistent Naming Conventions: Giving un-appropriate name to packages which might lead into confusion regarding the system. It also

create confusion amongst the team members, as they do not understand the purpose each package due to their inconsistent name.

Ignoring Updates / Changes: If the package diagram is not updated accordingly with the changes made in the system, then it will become outdated and useless, eventually it will lost its significance as a visualization and documentation tool.

Lack of involvements from Stakeholders: If the stakeholders don't provide their input while making of the package diagram, it might not align with their need and actual requirement of the project.

Package Diagrams in Software Development

Package Diagram plays a significant role in the field of Software Development as a visual representation of an architecture of a system. It helps in understanding , organizing and communication of the structural component of a system.

Package Diagram provides a high level view of the system's architecture by grouping related elements together to form packages, and representing the dependencies and interactions with each other.

Package Diagram promotes modularity and encapsulation, which allows the developers to create software with clear boundaries among its components.

By visually representing classes, interfaces, functions and other elements in a concise manner by including them in a package, software developers can manage the complexity of large software/systems.

Package Diagram also used as a communication tool between the team members and the stakeholders, as it becomes easy to show the internal architecture of the system and provide a detail view of it, as it assist in finding the dependencies between the elements of the software, it also helps in terms of the maintenance of the system.

COMPONENT DIAGRAM

A component diagram is used to break down a large object-oriented system into the smaller components, so as to make them more manageable. It models the physical view of a system such as executables, files, libraries, etc. that resides within the node.

It visualizes the relationships as well as the organization between the components present in the system. It helps in forming an executable system. A component is a single unit of the system, which is replaceable and executable. The implementation details of a component are hidden, and it necessitates an interface to execute a function. It is like a black box whose behavior is explained by the provided and required interfaces.

Notations

Relationships used in Component Diagram

- dependency, aggregation, constraint, generalization, association, and realization

Purpose of a Component Diagram

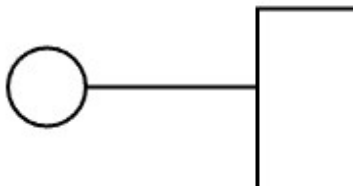
Since it is a special kind of a UML diagram, it holds distinct purposes. It describes all the individual components that are used to make the functionalities, but not the functionalities of the system. It visualizes the physical components inside the system. The components can be a library, packages, files, etc.

The component diagram also describes the static view of a system, which includes the organization of components at a particular instant. The collection of component diagrams represents a whole system.

The main purpose of the component diagram are enlisted below:

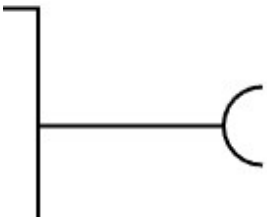
- It envisions each component of a system.
- It constructs the executable by incorporating forward and reverse engineering.
- It depicts the relationships and organization of components.
- Why use Component Diagram?
- The component diagrams have remarkable importance. It is used to depict the functionality and behavior of all the components present in the system, unlike other diagrams that are used to represent the architecture of the system, working of a system, or simply the system itself.

Provided interfaces: A straight line from the component box with an attached circle. These symbols represent the interfaces where a component produces information used by the required interface of another component.



Component Diagram Required Interfaces

Required interfaces: A straight line from the component box with an attached half circle (also represented as a dashed arrow with an open arrow). These symbols represent the interfaces where a component requires information in order to perform its proper function.



The component diagram can be used:

- To model the components of the system.
- To model the schemas of a database.
- To model the applications of an application.
- To model the system's source code.

In UML, the component diagram portrays the behavior and organization of components at any instant of time. The system cannot be visualized by any individual component, but it can be by the collection of components.

Reasons for the requirement of the component diagram:

- It portrays the components of a system at the runtime.
- It is helpful in testing a system.
- It envisions the links between several connections.
- When to use a Component Diagram?
- It represents various physical components of a system at runtime.
- It is helpful in visualizing the structure and the organization of a system.
- It describes how individual components can together form a single system.

When to use component diagram:

- To divide a single system into multiple components according to the functionality.
- To represent the component organization of the system.

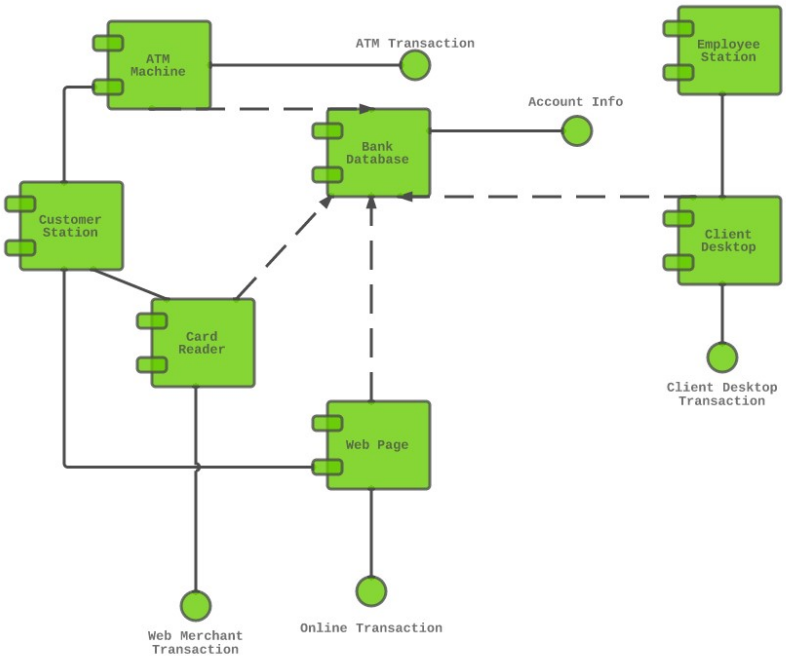
How to Draw a Component Diagram?

- The component diagram is helpful in representing the physical aspects of a system, which are files, executables, libraries, etc. The main purpose of a component diagram is different from that of other diagrams. It is utilized in the implementation phase of any application.

- Once the system is designed employing different UML diagrams, and the artifacts are prepared, the component diagram is used to get an

idea of implementation. It plays an essential role in implementing applications efficiently.

Component Diagram for ATM System



DEPLOYMENT DIAGRAM

Deployment diagram is a kind of structure diagram used in modeling the physical aspects of an object-oriented system. They are often be used to model the static deployment view of a system (topology of the hardware).

Deployment diagram illustrates how software architecture, designed on a conceptual level, translates into the physical system architecture where the software will run as nodes.
It maps out the deployment of software components onto hardware nodes and depicts their relationships through communication paths, enabling a visual representation of the software's execution environment across multiple nodes.

Deployment Diagram at a Glance

Deployment diagrams are important for visualizing, specifying, and documenting embedded, client/server, and distributed systems and also for managing executable systems through forward and reverse engineering.

A deployment diagram is just a special kind of class diagram, which focuses on a system's nodes. Graphically, a deployment diagram is a collection of vertices and arcs.

Relationships and other notations

- Dependency
- Association relationships.
- May also contain notes and constraints.

Key elements of a Deployment Diagram

1. Nodes - These represent the physical hardware entities where software components are deployed, such as servers, workstations, routers, etc.

2. Components - Represent software modules or artifacts that are deployed onto nodes, including executable files, libraries, databases, and configuration files.

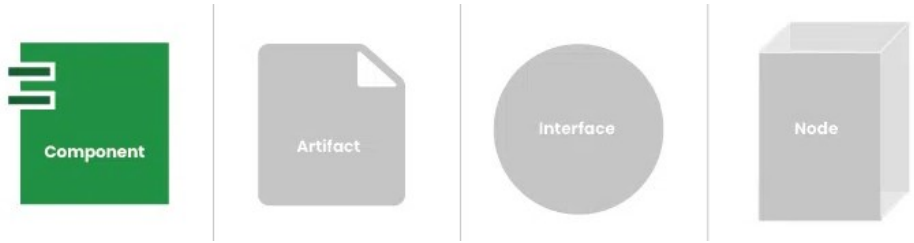
3. Artifacts = Physical files deployed onto nodes, embodying the actual implementation of software components, such as executables, scripts, databases, etc.

4. Dependencies - Reflect relationships or connections between nodes and components, indicating communication paths, deployment constraints, or other dependencies.

5. Associations - Show relationships between nodes and components, signifying that a component is deployed on a particular node, thus mapping software components to physical nodes.

6. Deployment Specification - Describes the configuration and properties of nodes and components, encompassing hardware specifications, software configurations, communication protocols, etc.

7. Communication Paths - Represent channels or connections facilitating communication between nodes and components, including network connections, communication protocols, etc.



When to Use Deployment Diagram

- What existing systems will the newly added system need to interact or integrate with?
- How robust does system need to be (e.g., redundant hardware in case of a system failure)?
- What and who will connect to or interact with system, and how will they do it
- What middleware, including the operating system and communications approaches and protocols, will system use?
- What hardware and software will users directly interact with (PCs, network computers, browsers, etc.)?
- How will you monitor the system once deployed?
- How secure does the system needs to be (needs a firewall, physically secure hardware, etc.)?

Purpose of Deployment Diagrams

- They show the structure of the run-time system
- They capture the hardware that will be used to implement the system and the links between different items of hardware.
- They model physical hardware elements and the communication paths between them
- They can be used to plan the architecture of a system.
- They are also useful for Document the deployment of software components or nodes

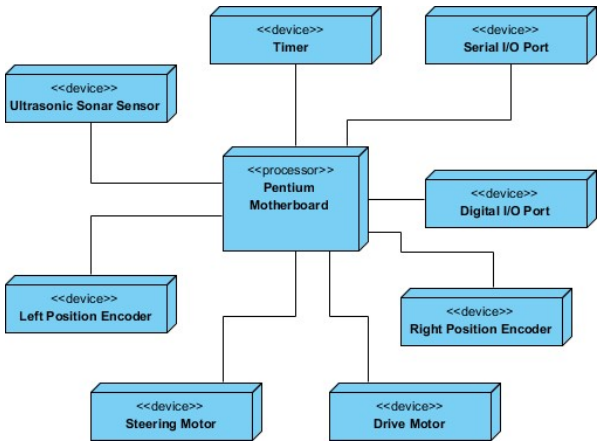
Steps for Modeling an Embedded System

- Identify the devices and nodes that are unique to your system.
- Provide visual cues, especially for unusual devices, by using the UML's extensibility mechanisms to define system-specific stereotypes with appropriate icons. At the very least, you'll want to distinguish processors (which contain software components) and devices (which, at that level of abstraction, don't directly contain software).
- Model the relationships among these processors and devices in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.
- As necessary, expand on any intelligent devices by modeling their structure with a more detailed deployment diagram.

Deployment Diagram for Embedded System

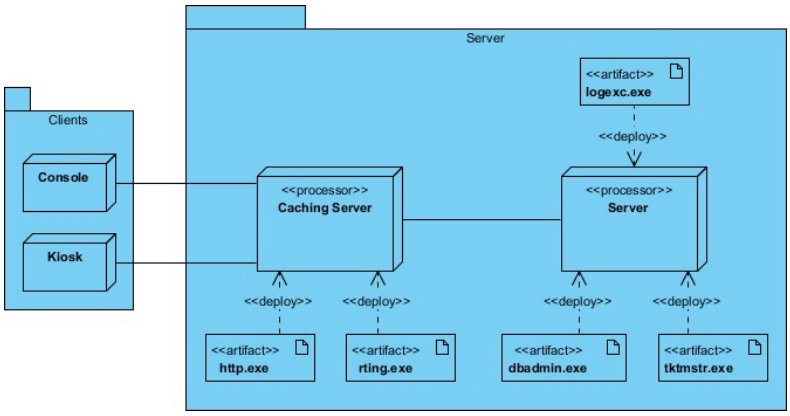
Steps for Modeling a Client/Server System

- Identify the nodes that represent your system's client and server processors.
- Highlight those devices that are germane to the behavior of your system. For example, you'll want to model special devices, such as credit card readers, badge readers, and display devices other than monitors, because their placement in the system's hardware topology are likely to be architecturally significant.
- Provide visual cues for these processors and devices via stereotyping.
- Model the topology of these nodes in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.



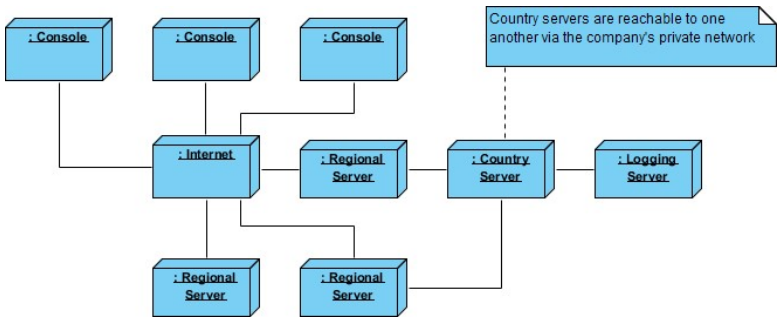
Steps for Modeling a Client/Server System

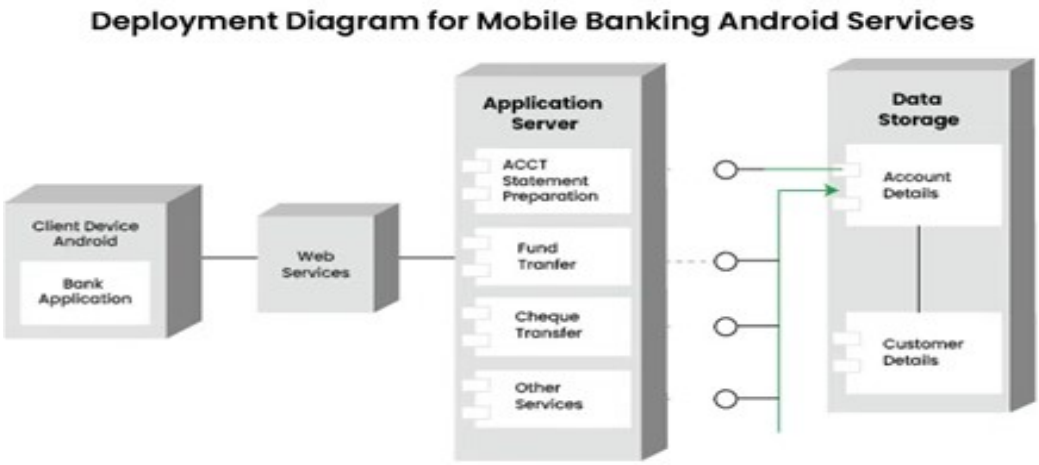
- Identify the nodes that represent your system's client and server processors.
- Highlight those devices that are germane to the behavior of your system. For example, you'll want to model special devices, such as credit card readers, badge readers, and display devices other than monitors, because their placement in the system's hardware topology are likely to be architecturally significant.
- Provide visual cues for these processors and devices via stereotyping.
- Model the topology of these nodes in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.



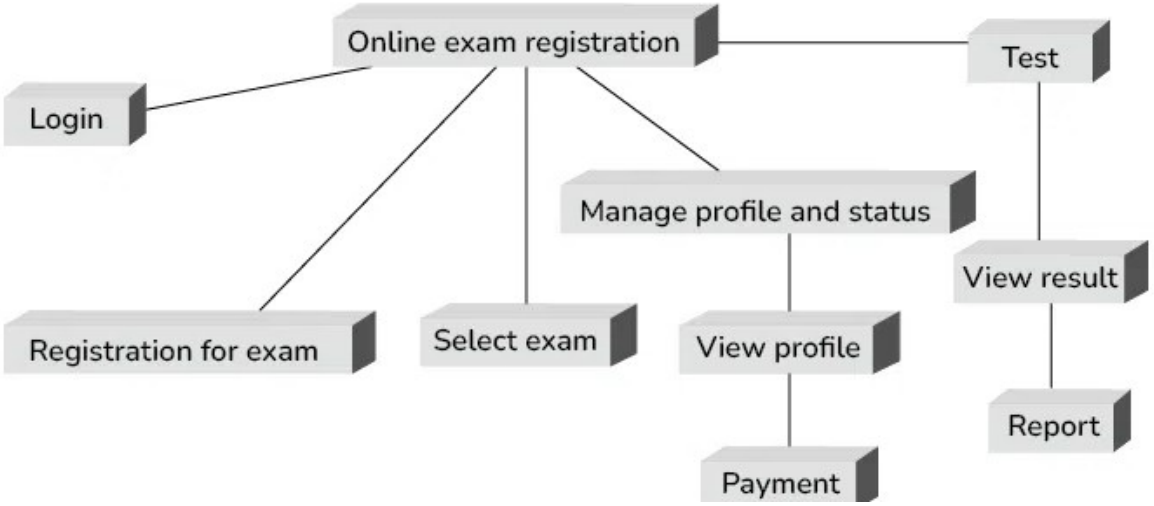
Deployment Diagram Example - Modeling a Distributed System

- Identify the system's devices and processors as for simpler client/server systems.
- If you need to reason about the performance of the system's network or the impact of changes to the network, be sure to model these communication devices to the level of detail sufficient to make these assessments.
- Pay close attention to logical groupings of nodes, which you can specify by using packages.
- Model these devices and processors using deployment diagrams. Where possible, use tools that discover the topology of your system by walking your system's network.
- If you need to focus on the dynamics of your system, introduce use case diagrams to specify the kinds of behavior you are interested in, and expand on these use cases with interaction diagrams.
- When modeling a fully distributed system, it's common to reify the network itself as an node. i.e. Internet, LAN, WAN as nodes





Deployment Diagram for Online Exam Registration System

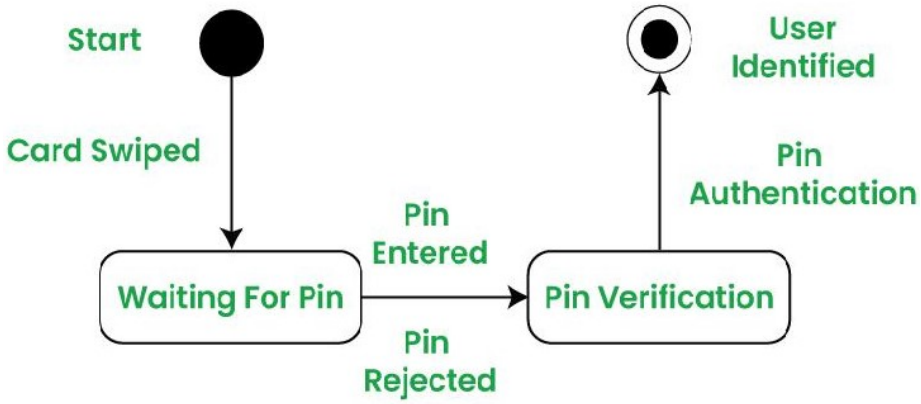


STATE DIAGRAM

A state diagram is used to represent the condition of the system or part of the system at finite instances of time.
It is a behavioral diagram and it represents the behavior using finite state transitions.

- State Machine diagrams are also referred to as State Machines Diagrams and State-Chart Diagrams.
- These terms are often used interchangeably. So simply, a state machine diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli.
- Each and every class has a state but we don’t model every class using State Machine diagrams.
- We prefer to model the states with three or more states.

State Machine for User Verification



Basic components and notations of a State Machine diagram

Initial state

We use a black filled circle represent the initial state of a System or a Class.

Initial State

Transition

We use a solid arrow to represent the transition or change of control from one state to another.
The arrow is labeled with the event which causes the change in state.

Transition

State

We use a rounded rectangle to represent a state. A state represents the conditions or circumstances of an object of a class at an instant of time.

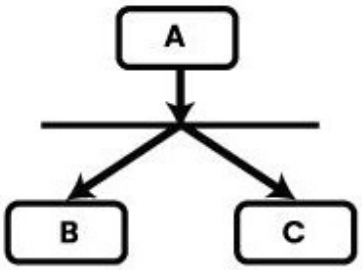
State

Name	A textual string which distinguishes one state from other states
Entry/exit actions	Actions executed on entering and exiting the state
Internal transitions	Transitions that are handled without causing a change in state
Substates	The nested structure of a state, involving disjoint or concurrent substates
Deferred events	A list of events that are not handled in that state, but are postponed or queued for handling by the object in another state

Fork

We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent state and outgoing arrows towards the newly created states.

We use the fork notation to represent a state splitting into two or more concurrent states.

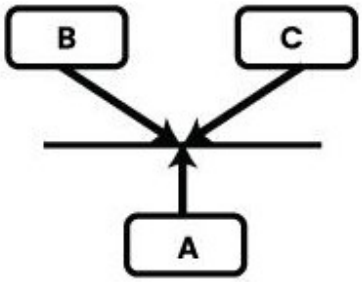


Fork

Join

We use a rounded solid rectangular bar to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state.

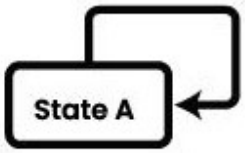
We use the join notation when two or more states concurrently converge into one on the occurrence of an event or events.



Join

Self transition

We use a solid arrow pointing back to the state itself to represent a self transition. There might be scenarios when the state of the object does not change upon the occurrence of an event.
We use self transitions to represent such cases.

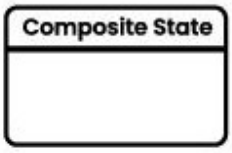


Self Transition

Source state	The state affected by the transition
Event trigger	The event whose reception changes the state of the object
Guard condition	A boolean expression which is evaluated when the transition is triggered
Action	An executable atomic computation that is performed on or by an object
Target state	The state that is active after the completion of the transition

Composite state

We use a rounded rectangle to represent a composite state also. We represent a state with internal activities using a composite state.



Composite State

Final State

We use a filled circle within a circle notation to represent the final state in a state machine diagram.

Final State

Guard Condition

A Boolean expression that must be true for a transition to occur

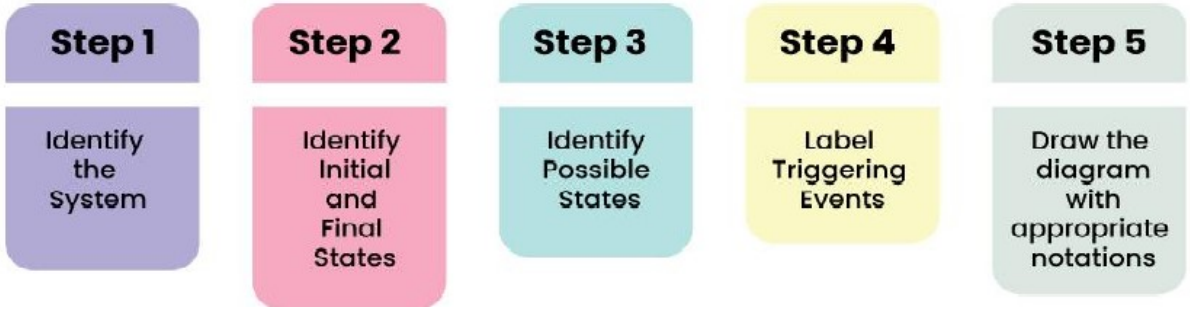
Notation: Appears in square brackets [ ] next to the transition arrow.

Choice Pseudo-State

Definition: Used for branching transitions based on multiple conditions.

Notation: Represented by a diamond shape, similar to a decision point.

How to draw State Machine Diagram



Step1. Identify the System:

- Understand what your diagram is representing.
- Whether it's a machine, a process, or any object, know what different situations or conditions it might go through.

Step2. Identify Initial and Final States:

- Figure out where your system starts (initial state) and where it ends (final state).
- These are like the beginning and the end points of your system's journey.

Step3. Identify Possible States:

- Think about all the different situations your system can be in.
- These are like the various phases or conditions it experiences.
- Use boundary values to guide you in defining these states.

Step4. Label Triggering Events:

- Understand what causes your system to move from one state to another.
- These causes or conditions are the events.

- Label each transition with what makes it happen.

Step5. Draw the Diagram with appropriate notations:

- Now, take all this information and draw it out.
- Use rectangles for states, arrows for transitions, and circles or rounded rectangles for initial and final states.

Uses of State Machine Diagram

- State Machine Diagrams are particularly useful for modeling and visualizing the dynamic behavior of a system.
- They showcase how a system responds to events and transitions between different states.
- We use it to state the events responsible for change in state (we do not show what processes cause those events).
- Objects go through different states during their existence, and these diagrams help in illustrating these states and the transitions between them.
- In embedded systems, where hardware interacts with software to perform tasks, State Machine Diagrams are valuable for representing the control logic and behavior of the system.

When to use a State Machine Diagram?

The state machine diagram implements the real-world models as well as the object-oriented systems. It records the dynamic behavior of the system, which is used to differentiate between the dynamic and static behavior of a system.

It portrays the changes underwent by an object from the start to the end. It basically envisions how triggering an event can cause a change within the system.

State machine diagram is used for:

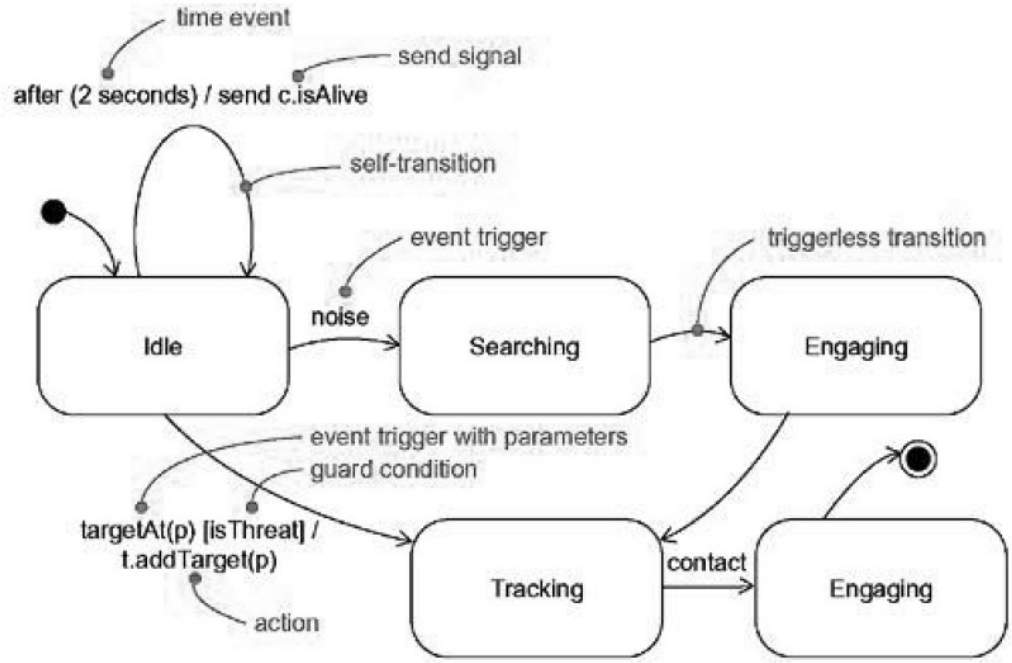
For modeling the object states of a system
For modeling the reactive system as it consists of reactive objects
For pinpointing the events responsible for state transitions
For implementing forward and reverse engineering

To model the lifetime of an object:

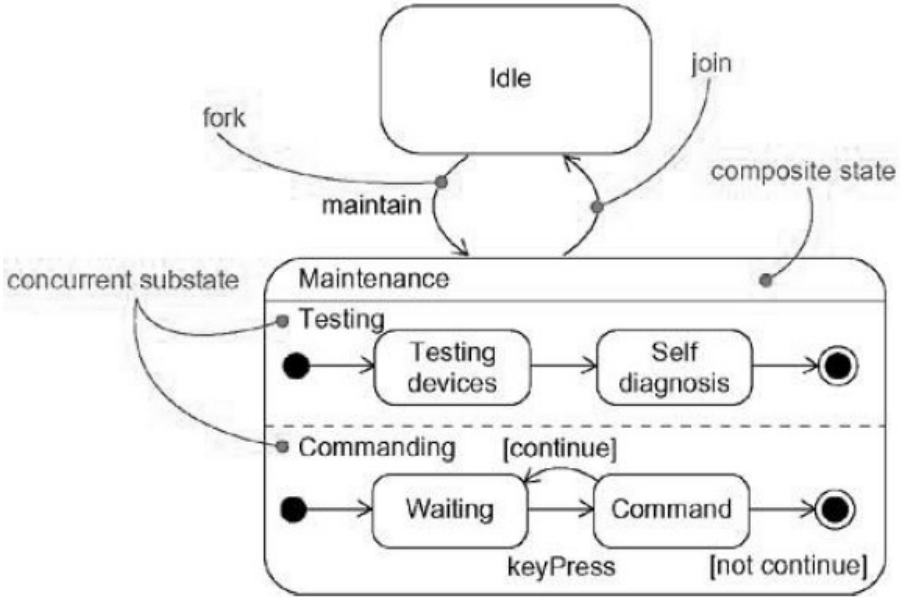
Set the context for the state machine, whether it is a class, a use case, or the system as a whole.

- Establish the initial and final states for the object.
- Consider the events to which this object may respond.
- Starting from the initial state to the final state, lay out the top level states the object may be in. Connect these states with transitions triggered by the appropriate events.
- Identify any entry or exit actions.
- Expand these states as necessary by using sub states.
- Check for the consistency of events in the object interface with the events in the state machine.
- Check that all actions mentioned in the state machine are sustained by the relationships, methods and operations of the enclosing object.
- Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses.
- After rearranging, again check it against the expected sequences to ensure that you have not changed the object's semantics.

States and Transitions



Concurrent States



State Machine vs Flow Chart

State Machine Diagram	Flow Chart
An State Machine Diagram is associated with the UML(Unified Modeling Language)	A Flow Chart is associated with the programming.
The basic purpose of a state machine diagram is to portray various changes in state of the class and not the processes or commands causing the changes.	A flowchart on the other hand portrays the processes or commands that on execution change the state of class or an object of the class.
Primarily used for systems, emphasizing their states and transitions.	Often used for processes, procedures, or algorithms involving actions and decisions.

ACTIVITY DIAGRAM

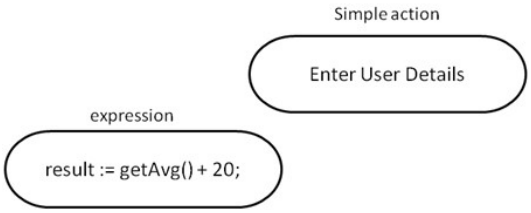
Activity diagram is used to demonstrate the flow of control within the system rather than the implementation. It models the concurrent and sequential activities.

The activity diagram helps in envisioning the workflow from one activity to another. It put emphasis on the condition of flow and the order in which it occurs. The flow can be sequential, branched, or concurrent, and to deal with such kinds of flows, the activity diagram has come up with a fork, join, etc.

Activity diagrams mainly contain:

- Activity states and action states
- Transitions
- Objects

In UML, an action states is represented using a lozenge symbol (rounded rectangle)



Usage

- To model a workflow
- To model an operation

To model a workflow:

- Establish a focus for the workflow.
- Select the objects that have the high-level responsibilities for parts of the overall workflow. Create a swimlane for each of these objects.
- Identify the pre-conditions of the workflow's initial state and the post-conditions of the workflow's final state.
- Starting at the workflow's initial state layout the actions and activities that take place and render them as actions states or activity states.
- For complex actions, or for sets of actions that appear multiple times, collapse them into activity states and provide a separate activity diagram for them.
- Connect the action and activity states by transitions. Consider branching and then consider forking and joining.
- If there are important objects that are involved in the workflow, render them in the activity diagram as well.

To model an operation:

- Collect the abstractions involved in an operation like: parameters, attributes of the enclosing class and the neighboring class.
- Identify the operation's pre-conditions and the operation's post-conditions.
- Beginning at the operation's initial state, define the actions and activities and render them as action states and activity states respectively.

- Use branching as necessary to specify conditional paths and iteration.
- If this operation is owned by an active class, use forking and joining as necessary to specify parallel flows of control.

Components

The control flow of **activity** is represented by control nodes and object nodes that illustrates the objects used within an activity.

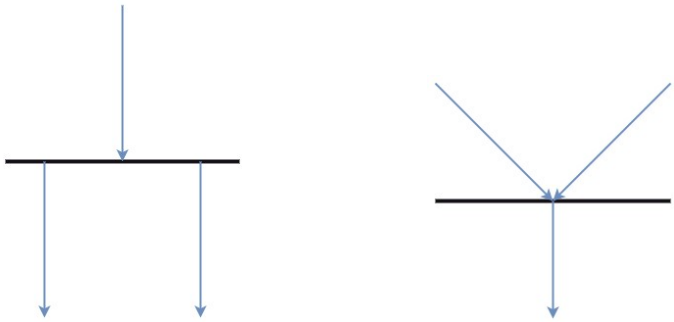
The activities are initiated at the initial node and are terminated at the final node.

The **swimlane** is used to cluster all the related activities in one column or one row. It can be either vertical or horizontal. It used to add modularity to the activity diagram. It is not necessary to incorporate swimlane in the activity diagram.

Forks and join nodes generate the concurrent flow inside the activity.

Fork - A fork node consists of one inward edge and several outward edges.

It is the same as that of various decision parameters. Whenever a data is received at an inward edge, it gets copied and split crossways various outward edges. It split a single inward flow into multiple parallel flows.



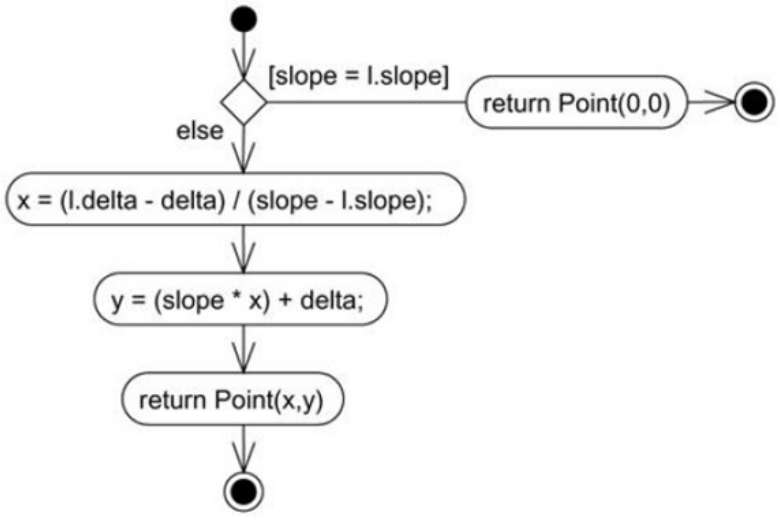
Fork **Join**

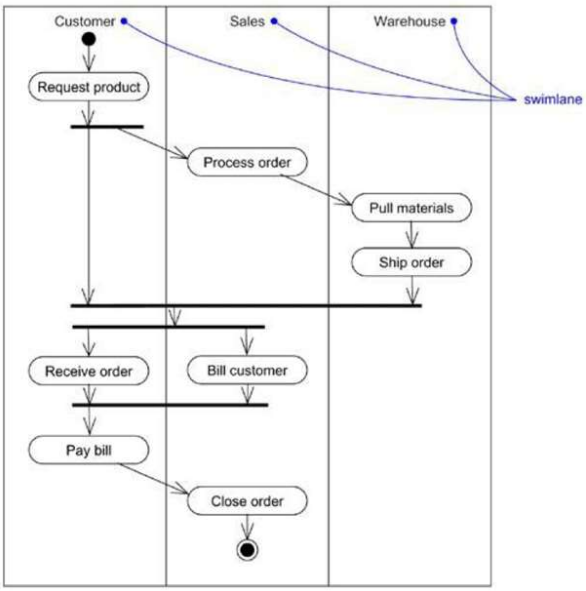
Join nodes are the opposite of fork nodes. A Logical AND operation is performed on all of the inward edges as it synchronizes the flow of input across one single output (outward) edge.

Pins - It is a small rectangle, which is attached to the action rectangle. It clears out all the messy and complicated thing to manage the execution flow of activities.

Rules of Activity Diagram

- A meaningful name should be given to each and every activity.
- Identify all of the constraints.
- Acknowledge the activity associations.





When to use Activity Diagram

- To graphically model the workflow in an easier and understandable way.
- To model the execution flow among several activities.
- To model comprehensive information of a function or an algorithm employed within the system.
- To model the business process and its workflow.
- To envision the dynamic aspect of a system.
- To generate the top-level flowcharts for representing the workflow of an application.
- To represent a high-level view of a distributed or an object-oriented system.

COLLABORATION DIAGRAM

Collaboration Diagram is a type of Interaction Diagram that visualizes the interactions and relationships between objects in a system.
It shows how objects collaborate to achieve a specific task or behavior.

Collaboration diagrams are used to model the dynamic behavior of a system and illustrate the flow of messages between objects during a particular scenario or use case.

Relationships: **Link** ————

Importance of Collaboration Diagrams

Collaboration diagrams play a crucial role in system development by facilitating understanding, communication, design, analysis, and documentation of the system's architecture and behavior.

Visualizing Interactions:

- These diagrams offer a clear visual representation of how objects or components interact within a system.
- This visualization aids stakeholders in comprehending the flow of data and control, fostering easier understanding.

Understanding System Behavior:

- By depicting interactions, collaboration diagrams provide insights into the system's dynamic behavior during operation.
- Understanding this behavior is crucial for identifying potential issues, optimizing performance, and ensuring the system functions as intended.

Facilitating Communication:

- Collaboration diagrams serve as effective communication tools among team members.
- They facilitate discussions, enabling refinement of the system's design, architecture, and functionality. Clearer communication fosters better collaboration and alignment.

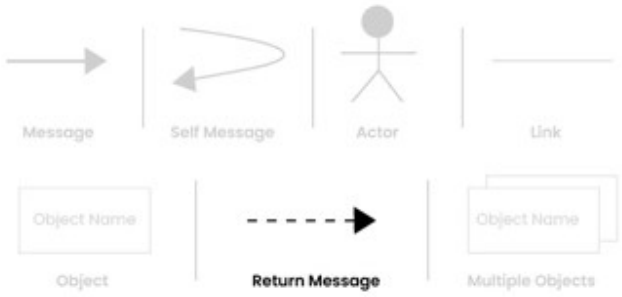
Supporting Design and Analysis:

- These diagrams assist in designing and analyzing system architecture and functionality.
- They help identify objects, their relationships, and message exchanges, which is vital for creating efficient and scalable systems.

Documentation Purposes:

- Collaboration diagrams serve as valuable documentation assets for the system.
- They offer a visual representation of the system's architecture and behavior, serving as a reference for developers, testers, and other stakeholders throughout the development process.

Components and their Notations in Collaboration Diagrams



Objects/Participants - Objects are represented by rectangles with the object's name at the top. Each object participating in the interaction is shown as a separate rectangle in the diagram. Objects are connected by lines to indicate messages being passed between them.

Multiple Objects - Multiple objects are represented by rectangles, each with the object's name inside, and interactions between them are shown using arrows to indicate message flows.

Actors - They are usually depicted at the top or side of the diagram, indicating their involvement in the interactions with the system's objects or components. They are connected to objects through messages, showing the communication with the system.

Messages - Messages represent communication between objects. Messages are shown as arrows between objects, indicating the flow of communication. Each message may include a label indicating the type of message (e.g., method call, signal). Messages can be asynchronous (indicated by a dashed arrow) or synchronous (solid arrow).

Self Message - This is a message that an object sends to itself. It represents an action or behavior that the object performs internally, without involving any other objects. Self-messages are useful for modeling scenarios where an object triggers its own methods or processes.

Links - Links represent associations or relationships between objects. Links are shown as lines connecting objects, with optional labels to indicate the nature of the relationship. Links can be uni-directional or bi-directional, depending on the nature of the association.

Return Messages - Return messages represent the return value of a message. They are shown as dashed arrows with a label indicating the

return value. Return messages are used to indicate that a message has been processed and a response is being sent back to the calling object.

To draw a collaboration diagram:

- **Step 1: Identify Objects/Participants:** Begin by identifying the objects or participants involved in the system. These can be classes, modules, actors, or any other relevant entities.
- **Step 2: Define Interactions:** Determine how these objects interact with each other to accomplish tasks or scenarios within the system. Identify the messages exchanged between objects during these interactions.
- **Step 3: Add Messages:** Draw arrows between lifelines to represent the messages exchanged between objects. Label each arrow with the name of the message and, if applicable, any parameters or data being transmitted.
- **Step 4: Consider Relationships:** If there are associations or dependencies between objects, represent them using appropriate notations, such as dashed lines or arrows.
- **Step 5: Documentation:** Once finalized, document the collaboration diagram along with any relevant explanations or annotations. Ensure that the diagram effectively communicates the system's interactions to stakeholders.

Use cases of Collaboration Diagrams

- **Software Development:** Collaboration diagrams help developers understand how different parts of a system interact, aiding in building and testing software.
- **System Analysis and Design:** They assist in visualizing system interactions, aiding analysts and designers in refining system architecture.
- **Team Communication:** Collaboration diagrams facilitate team discussions and decision-making by providing a clear visual representation of system interactions.
- **Documentation:** They are essential for documenting system architecture and design decisions, serving as valuable reference material for developers and testers.
- **Debugging and Troubleshooting:** Collaboration diagrams help trace message flow and identify system issues, aiding in debugging and troubleshooting efforts.

When to use Collaboration Diagram

- To model collaboration among the objects or roles that carry the functionalities of use cases and operations.
- To model the mechanism inside the architectural design of the system.
- To capture the interactions that represent the flow of messages between the objects and the roles inside the collaboration.
- To model different scenarios within the use case or operation, involving a collaboration of several objects and interactions.
- To support the identification of objects participating in the use case.
- In the collaboration diagram, each message constitutes a sequence number, such that the top-level message is marked as one and so on. The messages sent during the same call are denoted with the same decimal prefix, but with different suffixes of 1, 2, etc

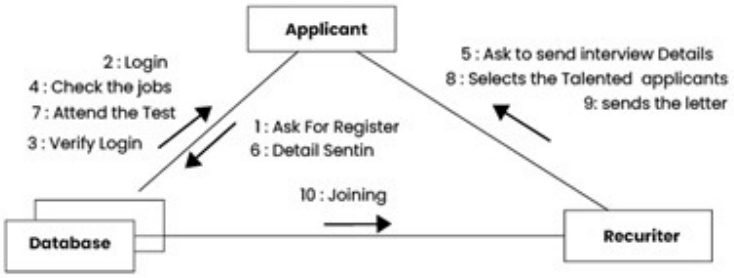
Benefits of Collaboration Diagrams

- The collaboration diagram is also known as Communication Diagram.
- It mainly puts emphasis on the structural aspect of an interaction diagram, i.e., how lifelines are connected.
- The syntax of a collaboration diagram is similar to the sequence diagram; just the difference is that the lifeline does not consist of tails.
- The messages transmitted over sequencing is represented by numbering each individual message.
- The collaboration diagram is semantically weak in comparison to the sequence diagram.
- The special case of a collaboration diagram is the object diagram.
- It focuses on the elements and not the message flow, like sequence diagrams.
- Since the collaboration diagrams are not that expensive, the sequence diagram can be directly converted to the collaboration diagram.
- There may be a chance of losing some amount of information while implementing a collaboration diagram with respect to the sequence diagram.

Challenges of Collaboration Diagrams

Complexity: Keeping collaboration diagrams clear in large systems with many objects and interactions can be tough.
Ambiguity: Sometimes, interpreting the interactions in diagrams isn't straight forward, leading to mis-understandings.
Dynamic Systems: Diagrams might not fully capture systems where interactions change over time, like those with continously processing and improvising.
Scalability: Managing diagrams becomes harder as systems grow, requiring efforts to keep them manageable.
Maintainability: Updating diagrams to reflect system changes can be challenging, especially in large and fast evolving systems.
Communication: Ensuring that diagrams effectively convey complex interactions to all stakeholders can be challenging.

Example: Job Recruitment System



Applicant –attend test –> Database
Applicant –provide details –> Database
Recruiter –verify login–> Database
Recruiter <– confirms login –> Database
Recruiter –check jobs positions –> Database
Recruiter –select talented applicant –> Applicant
Recruiter –send interview details –> Applicant
Recruiter –send joining letter –> Applicant
Database –send jobs –> Recruiter

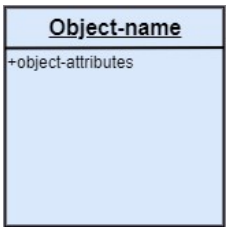
Drawbacks of a Collaboration Diagram

- Multiple objects residing in the system can make a complex collaboration diagram, as it becomes quite hard to explore the objects.
- It is a time-consuming diagram.
- After the program terminates, the object is destroyed.
- As the object state changes momentarily, it becomes difficult to keep an eye on every single that has occurred inside the object of a system.

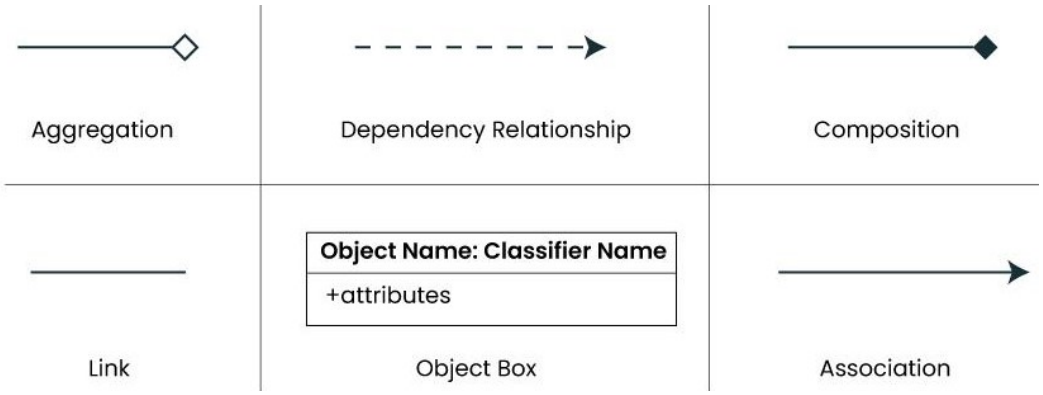
OBJECT DIAGRAM

- Object diagrams are dependent on the class diagram as they are derived from the class diagram. It represents an instance of a class diagram.
- The objects help in portraying a static view of an object-oriented system at a specific instant.
- Both the object and class diagram are similar to some extent; the only difference is that the class diagram provides an abstract view of a system. It helps in visualizing a particular functionality of a system.

Notation of an Object Diagram



Relationships used in Object Diagram



Purpose of Object Diagram
The object diagram holds the same purpose as that of a class diagram.

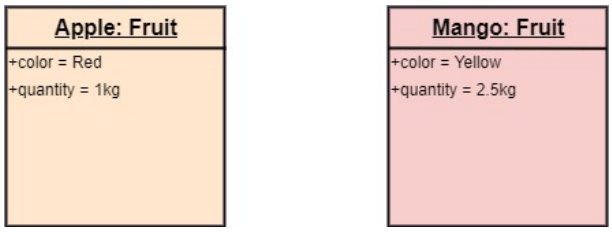
The class diagram provides an abstract view which comprises of classes and their relationships, whereas the object diagram represents an instance at a particular point of time.

The object diagram is actually similar to the concrete (actual) system behavior. The main purpose is to depict a static view of a system.

Purpose:

- It is used to perform forward and reverse engineering.
- It is used to understand object behavior and their relationships practically.
- It is used to get a static view of a system.
- It is used to represent an instance of a system.

Example of Object Diagram



How to draw an Object Diagram?

- All the objects present in the system should be examined before start drawing the object diagram.
- Before creating the object diagram, the relation between the objects must be acknowledged.
- The association relationship among the entities must be cleared already.
- To represent the functionality of an object, a proper meaningful name should be assigned.
- The objects are to be examined to understand its functionality.

- **Identify Classes**: Determine the classes relevant to the scenario you want to depict. Classes are the blueprints that define the attributes and behaviors shared by their instances.
- **Identify Objects:** Identify specific instances or objects of each class that you want to include in the diagram. These represent the actual things in your system.
- **Create Object Boxes:** Draw rectangles to represent the specific instances or objects of each class. Write the name of each object inside the box.
- **Add Attributes and Values:** Inside each object box, list the attributes of that object along with their specific values.
- **Draw Relationships:** Connect the object boxes with lines to represent relationships or associations between instances. Use arrows to indicate the direction of the association if necessary.
- **Label Relationships:** Label the relationships with multiplicity and role names if needed. Label the association lines with a verb or phrase to describe the nature of the relationship.
- **Review and Refine:** Review your Object diagram to ensure it accurately represents the system's structure and relationships. Refine the diagram as needed based on feedback and requirements.

Applications of Object diagrams

The following are the application areas where the object diagrams can be used.

- To build a prototype of a system.
- To model complex data structures.
- To perceive the system from a practical perspective.
- Reverse engineering.

Class Diagram vs Object Diagram

S. No.	Class Diagram	Object Diagram
1.	It depicts the static view of a system.	It portrays the real-time behavior of a system.
2.	Dynamic changes are not included	Dynamic changes are captured in the

	in the class diagram.	object diagram.
3.	The data values and attributes of an instance are not involved here.	It incorporates data values and attributes of an entity.
4.	The object behavior is manipulated in the class diagram.	Objects are the instances of a class.

****