SOFTWARE ENGINEERING

UNIT 3

Software Design

Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programming language.

Objectives of Software Design

Following are the purposes of Software design:

1. **Correctness:** Software design should be correct as per requirement.
2. **Completeness:** The design should have all components like data structures, modules, and external interfaces, etc.
3. **Efficiency:** Resources should be used efficiently by the program.
4. **Flexibility:** Able to modify on changing needs.
5. **Consistency:** There should not be any inconsistency in the design.
6. **Maintainability:** The design should be so simple so that it can be easily maintainable by other designers.

Software Design Principles

Software design principles are concerned with providing means to handle the complexity of the design process effectively. Effectively managing the complexity will not only reduce the effort needed for design but can also reduce the scope of introducing errors during design.

Following are the principles of Software Design

Problem Partitioning

For small problem, we can handle the entire problem at once but for the significant problem, divide the problems and conquer the problem it means to divide the problem into smaller pieces so that each piece can be captured separately.

For software design, the goal is to divide the problem into manageable pieces.

Benefits of Problem Partitioning

1. Software is easy to understand
2. Software becomes simple
3. Software is easy to test
4. Software is easy to modify
5. Software is easy to maintain
6. Software is easy to expand

These pieces cannot be entirely independent of each other as they together form the system. They have to cooperate and communicate to solve the problem. This communication adds complexity.

Abstraction

An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about the internal details of the implementation. Abstraction can be used for existing element as well as the component being designed.

Here, there are two common abstraction mechanisms

1. Functional Abstraction
2. Data Abstraction

Functional Abstraction

i. A module is specified by the method it performs.

ii. The details of the algorithm to accomplish the functions are not visible to the user of the function.

Functional abstraction forms the basis for **Function oriented design approaches**.

Data Abstraction

Details of the data elements are not visible to the users of data. Data Abstraction forms the basis for **Object Oriented design approaches**.

Software Design Levels

Software design yields three levels of results:

- **Architectural Design -** The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.
- **High-level Design-** The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.
- **Detailed Design-** Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

Characteristics	Good Design	Bad Design
Change	Change in one part of the system does not always require a change in another part of the system.	One conceptual change requires changes to many parts of the system.
Logic	Every piece of logic has one and only one home.	Logic has to be duplicated.
Nature	Simple	Complex
Cost	Small	Very high
Link	The logic link can easily be found.	The logic link cannot be remembered.
Extension	System can be extended with changes in only one place.	System cannot be extended so easily.

Design concepts

- **The set of fundamental software design concepts are as follows:**

1. Abstraction: Abstraction is the act of representing essential features without including the background details or explanations. the abstraction is used to reduce complexity and allow efficient design and implementation of complex software systems. Many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. As different levels of abstraction are developed, you work to create both procedural and data abstractions. A procedural abstraction refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. A data abstraction is a named collection of data that describes a data object

2. Architecture: Software architecture alludes to “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system” Architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

•Shaw and Garlan describe a set of properties that should be specified as part of an architectural design:

- Structural properties
- Extra-functional properties.
- Families of related systems.

•The aim of the software design is to obtain an architectural framework of a system.

•The more detailed design activities are conducted from the framework.

3.Patterns

A design pattern describes a design structure and that structure solves a particular design problem in a specified content.

•The intent of each design pattern is to provide a description that enables a designer to determine

(1) whether the pattern is applicable to the current work,

(2) whether the pattern can be reused (hence, saving design time), and

(3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

4. Separation of Concerns

• Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A concern is a feature or behavior that is specified as part of the requirements model for the software. Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement.

5. Modularity

•A software is separately divided into name and addressable components. Sometime they are called as modules which integrate to satisfy the problem requirements.

•Modularity is the single attribute of a software that permits a program to be managed easily.

•Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called module.

6. Information hiding

Information Hiding The principle of information hiding suggests that modules be “characterized by design decisions that hides from all others.” In other words, modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

7. Functional independence

The functional independence is the concept of separation and related to the concept of modularity, abstraction and information hiding.

•The functional independence is accessed using two criteria i.e Cohesion and coupling.

Cohesion:

Cohesion is an extension of the information hiding concept.

•A cohesive module performs a single task and it requires a small interaction with the other components in other parts of the program.

Coupling

Coupling is an indication of interconnection between modules in a structure of software.

8.Refinement

- Refinement is a top-down design approach.
- It is a process of elaboration.
- A program is established for refining levels of procedural details.
- A hierarchy is established by decomposing a statement of function in a stepwise manner till the programming language statement are reached.

Abstraction enables you to specify procedure and data internally but suppress the need for “outsiders” to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses.

9. Refactoring:

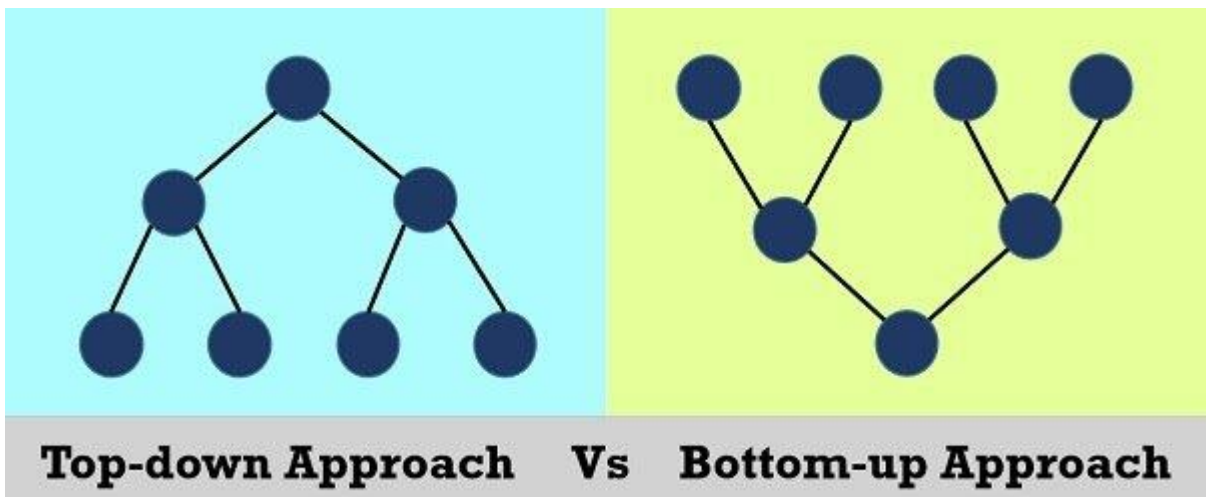
- It is a reorganization technique which simplifies the design of components without changing its function behaviour.
- Refactoring is the process of changing the software system in a way that it does not change the external behaviour of the code still improves its internal structure.

•: “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.”

DESIGN APPROACHES

1.Top-Down Design: This strategy starts with a high-level view of the system and gradually breaks it down into smaller, more manageable components.
2.Bottom-Up Design: This strategy starts with individual components and builds the system up, piece by piece.

Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.
Bottom-up approach is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.
Both top-down and bottom-up approaches are not practical individually .Instead a good combination of both is used.



Software Design Strategies

1.Structured Design

Structured design is mostly based on ‘divide and conquer’ strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.
The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution.
These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely -
Cohesion - grouping of all functionally related elements.
Coupling - communication between different modules.
A good structured design has high cohesion and low coupling arrangements.

2.Function oriented design

Function oriented design inherits some properties of structured design where divide and conquer methodology is used.
This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation.. These functional modules can share information among themselves by means of information passing and using information available globally.
Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Design Process

•The whole system is seen as how data flows in the system by means of data flow diagram.
•DFD depicts how functions changes data and state of entire system.
•The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
•Each function is then described at large.

3. Object Oriented Design

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategies focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

OBJECT ORIENTED DESIGN CONCEPTS

- **Objects:** Objects are all the entities involved in the solution design. Persons, banks, companies, and users are all examples of objects. Every object has some properties associated with it, along with some methods for performing operations on those attributes.
- **Class:** Classes are generic descriptions of objects. An object is a class instance. A class defines all the properties an object can have and the methods that represent the object's functionality.
- **Abstraction:** Abstraction is used in object-oriented design to deal with complexity. Abstraction is the removal of the unnecessary and the amplification of the necessary.
- **Encapsulation:** It is also known as information concealing. The processes and data are tied to a single unit. Encapsulation not only groups together an object's vital information but also restricts access to the data and operations from the outside world.
- **Inheritance:** OOD allows similar classes to be stacked hierarchically, with lower or sub-classes being able to import, implement, and reuse variables and functions from their immediate superclasses. This OOD characteristic is known as inheritance. This facilitates the definition of specialized classes as well as the creation of generic classes.
- **Polymorphism:** OOD languages give a technique for assigning the same name to methods that perform similar functions but differ in arguments. This is referred to as polymorphism, and it allows a single interface to perform functions for multiple types. The relevant piece of the code is run depending on how the service is invoked.

Design classes

- The model of software is defined as a set of design classes.
- Every class describes the elements of problem domain and that focus on features of the problem which are user visible.
- Five different types of design classes, each representing a different layer of the design architecture, can be developed:

Design classes

- **User interface classes** define all abstractions that are necessary for human computer interaction (HCI). The design classes for the interface may be visual representations of the elements of the metaphor.
- **Business domain classes** are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- **Process classes** implement lower-level business abstractions required to fully manage the business domain classes.
- **Persistent classes** represent data stores (e.g., a database) that will persist beyond the execution of the software.
- **System classes** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

four characteristics of a well-formed design class:

 Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

 Primitiveness. Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.

 High cohesion. A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.

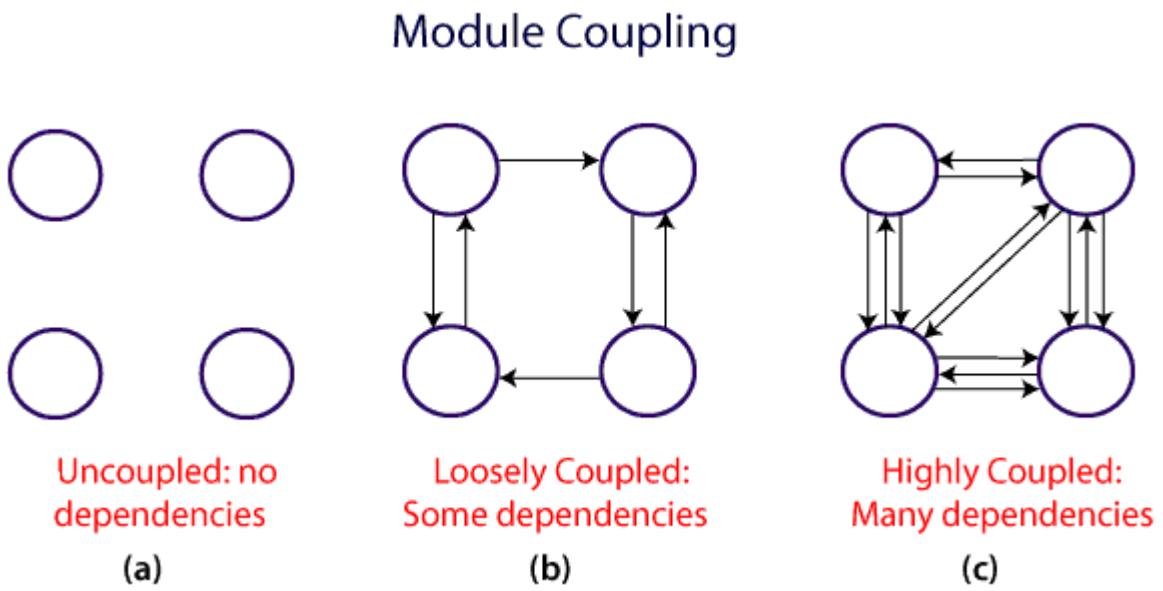
 Low coupling. Within the design model, it is necessary for design classes to collaborate with one another. If a design model is highly coupled, the system is difficult to implement, to test, and to maintain over time.

Coupling and Cohesion

Module Coupling

In software engineering, the coupling is the degree of interdependence between software modules. Two modules that are tightly coupled are strongly dependent on each other. However, two modules that are loosely coupled are not dependent on each other. **Uncoupled modules** have no interdependence at all within them.

The various types of coupling techniques are shown in fig:

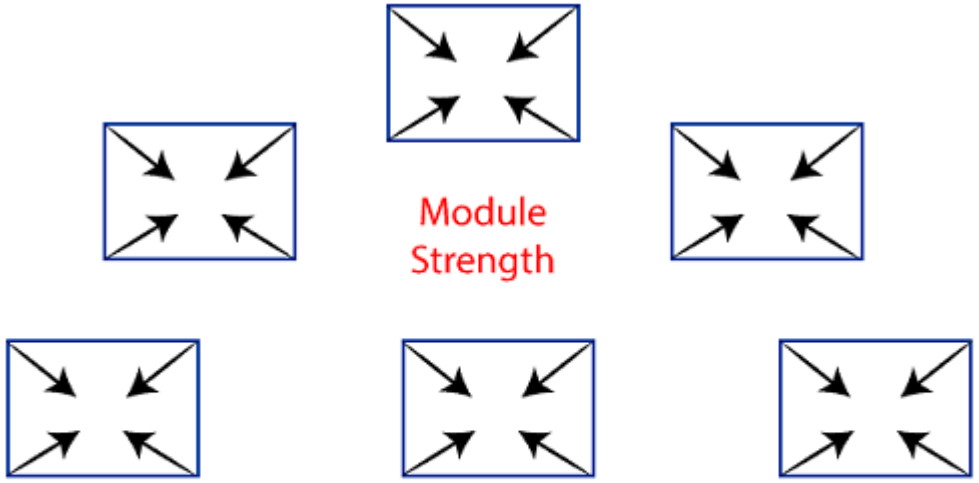


TYPES OF MODULES COUPLING:

No Direct Coupling: There is no direct coupling between M1 and M2.
Content Coupling: Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.
Common Coupling: Two modules are common coupled if they share information through some global data items.
Control Coupling: Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.
Stamp Coupling: Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.
Data Coupling: When data of one module is passed to another module, this is called data coupling.
Ideally no coupling is considered to be best.

Cohesion

In computer programming, cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.
Cohesion is an **ordinal** type of measurement and is generally described as "high cohesion" or "low cohesion."



Cohesion= Strength of relations within Modules

Types of Modules Cohesion:

Coincidental Cohesion: A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all.
Logical Cohesion: A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.
Temporal Cohesion: When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.
Procedural Cohesion: A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.
Communicational Cohesion: A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.
Sequential Cohesion: A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.
Functional Cohesion: Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.

DESIGN MODEL

The design principles and concepts establish a foundation for the creation of the design model that encompasses representation of data, architecture, interface and components. Like the analysis model before it, each of these design representations is tied to the others, and all can be traced back to software requirements.

The entity-relationship diagrams(ERD),data flow diagrams(DFD),the state transition diagram(STD) and the data dictionaries(DD) that are constructed during the requirements phase are directly mapped on to the corresponding design model as below.

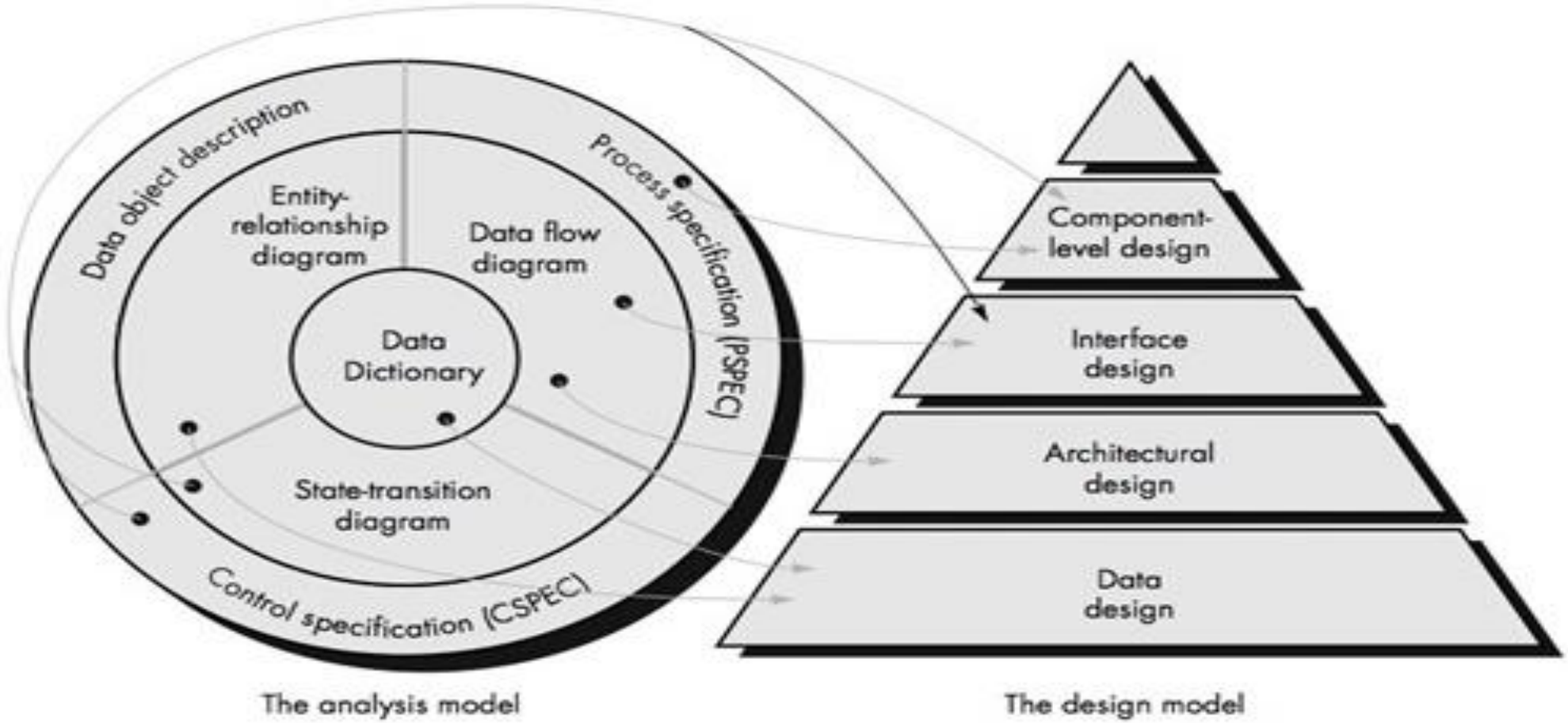


FIGURE 13.1 Translating the analysis model into a software design

•**Data design:** It represents the data objects and their interrelationship in an entity-relationship diagram. Entity-relationship consists of information required for each entity or data objects as well as it shows the relationship between these objects. It shows the structure of the data in terms of the tables. It shows three type of relationship – One to one, one to many, and many to many. In one to one relation, one entity is connected to another entity. In one many relation, one Entity is connected to more than one entity. un many to many relations one entity is connected to more than one entity as well as other entity also connected with first entity using more than one entity.
•**Architectural design:** It defines the relationship between major structural elements of the software. It is about decomposing the system into interacting components. It is expressed as a block diagram defining an overview of the system structure – features of the components and how these components communicate with each other to share data. It defines the structure and properties of the component that are involved in the system and also the inter-relationship among these components.

•**User Interfaces design:** It represents how the Software communicates with the user i.e. the behavior of the system. It refers to the product where user interact with controls or displays of the product. For example, Military, vehicles, aircraft, audio equipment, computer peripherals are the areas where user interface design is implemented. UI design becomes efficient only after performing usability testing. This is done to test what works and what does not work as expected. Only after making the repair, the product is said to have an optimized interface.

•**Component level design:** It transforms the structural elements of the software architecture into a procedural description of software components. It is a perfect way to share a large amount of data. Components need not be concerned with how data is managed at a centralized level i.e. components need not worry about issues like backup and security of the data.

UML

Unified Modeling Language (UML)

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

Do we really need UML?

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

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

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

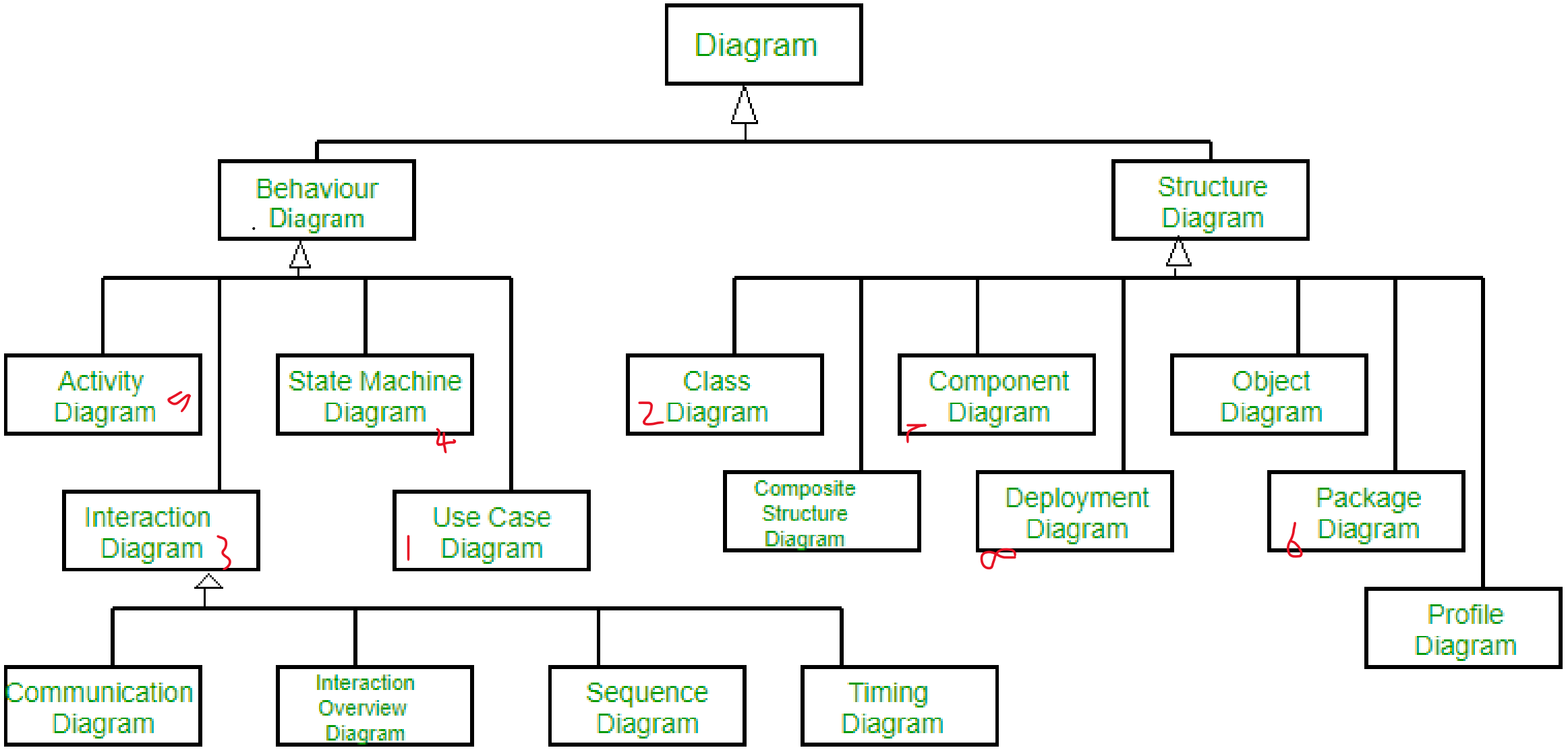


Diagram
Behaviour Diagram
Structure Diagram
Activity Diagram
State Machine Diagram
Interaction Diagram
Use Case Diagram
Class Diagram
Component Diagram
Composite Structure Diagram
Deployment Diagram
Object Diagram
Package Diagram
Profile Diagram
Communication Diagram
Interaction Overview Diagram
Sequence Diagram
Timing Diagram

1.USE CASE DIAGRAM

A use case diagram is used to represent the dynamic behavior of a system. It encapsulates the system's functionality by incorporating use cases, actors, and their relationships. It models the tasks, services, and functions required by a system/subsystem of an application. It depicts the high-level functionality of a system and also tells how the user handles a system.

In the Unified Modeling Language (UML), a use case diagram can summarize the details of your system's users (also known as actors) and their interactions with the system. To build one, you'll use a set of specialized symbols and connectors. An effective use case diagram can help your team discuss and represent:

- Scenarios in which your system or application interacts with people, organizations, or external systems
- Goals that your system or application helps those entities (known as actors) achieve
- The scope of your system

A use case diagram doesn't go into a lot of detail—for example, don't expect it to model the order in which steps are performed. Instead, a proper use case diagram depicts a high-level overview of the relationship between use cases, actors, and systems. Experts recommend that use case diagrams be used to supplement a more descriptive textual use case.

- Use cases are represented with a labeled oval shape. Stick figures represent actors in the process, and the actor's participation in the system is modeled with a line between the actor and use case. To depict the system boundary, draw a box around the use case itself.

UML use case diagrams are ideal for:

•Representing the goals of system-user interactions
•Defining and organizing functional requirements in a system
•Specifying the context and requirements of a system
•Modeling the basic flow of events in a use case

Use case diagram components

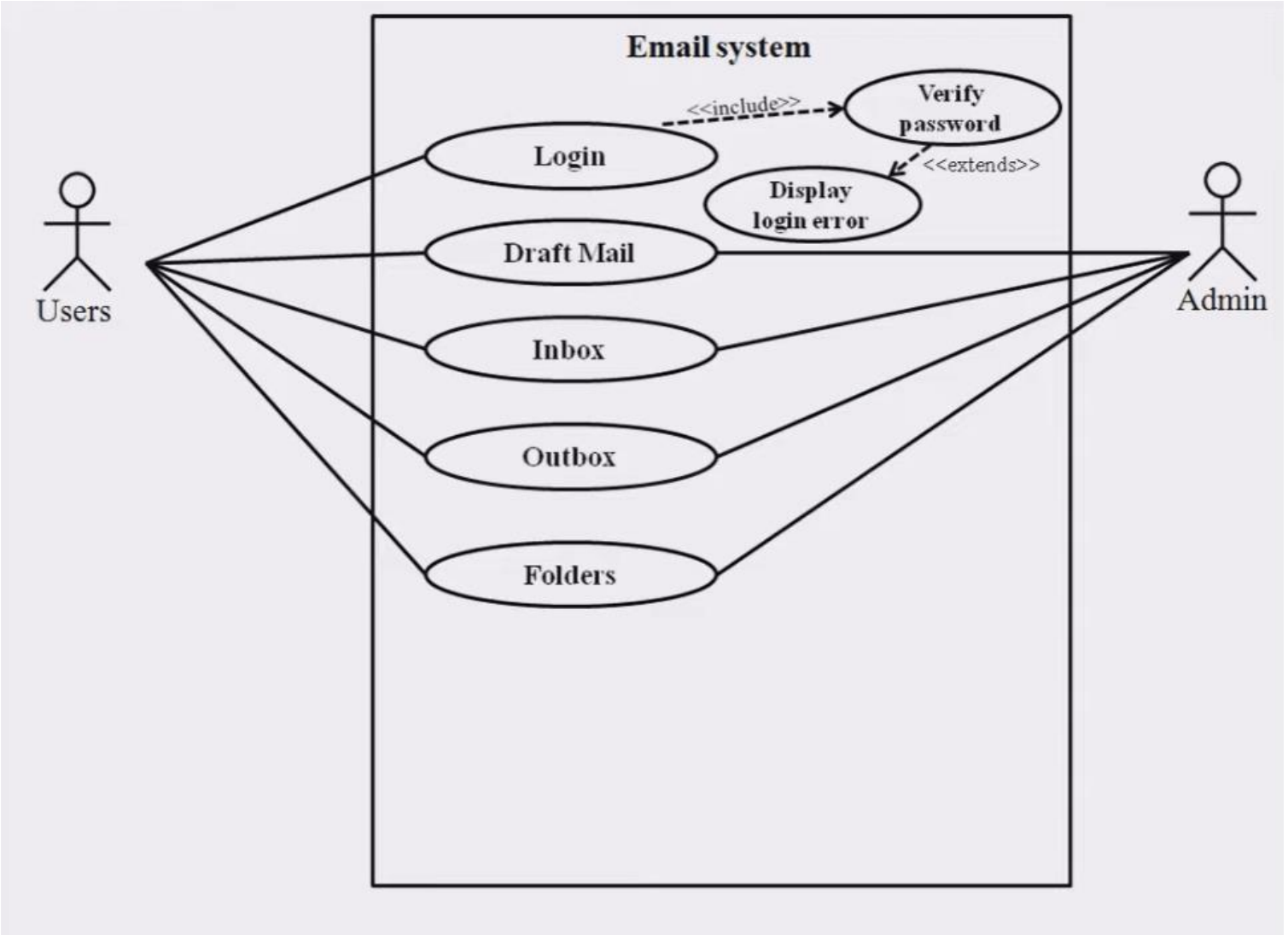
- **Actors**: The users that interact with a system. An actor can be a person, an organization, or an outside system that interacts with your application or system. They must be external objects that produce or consume data.
- **System**: A specific sequence of actions and interactions between actors and the system. A system may also be referred to as a scenario.
- **Goals**: The end result of most use cases. A successful diagram should describe the activities and variants used to reach the goal.

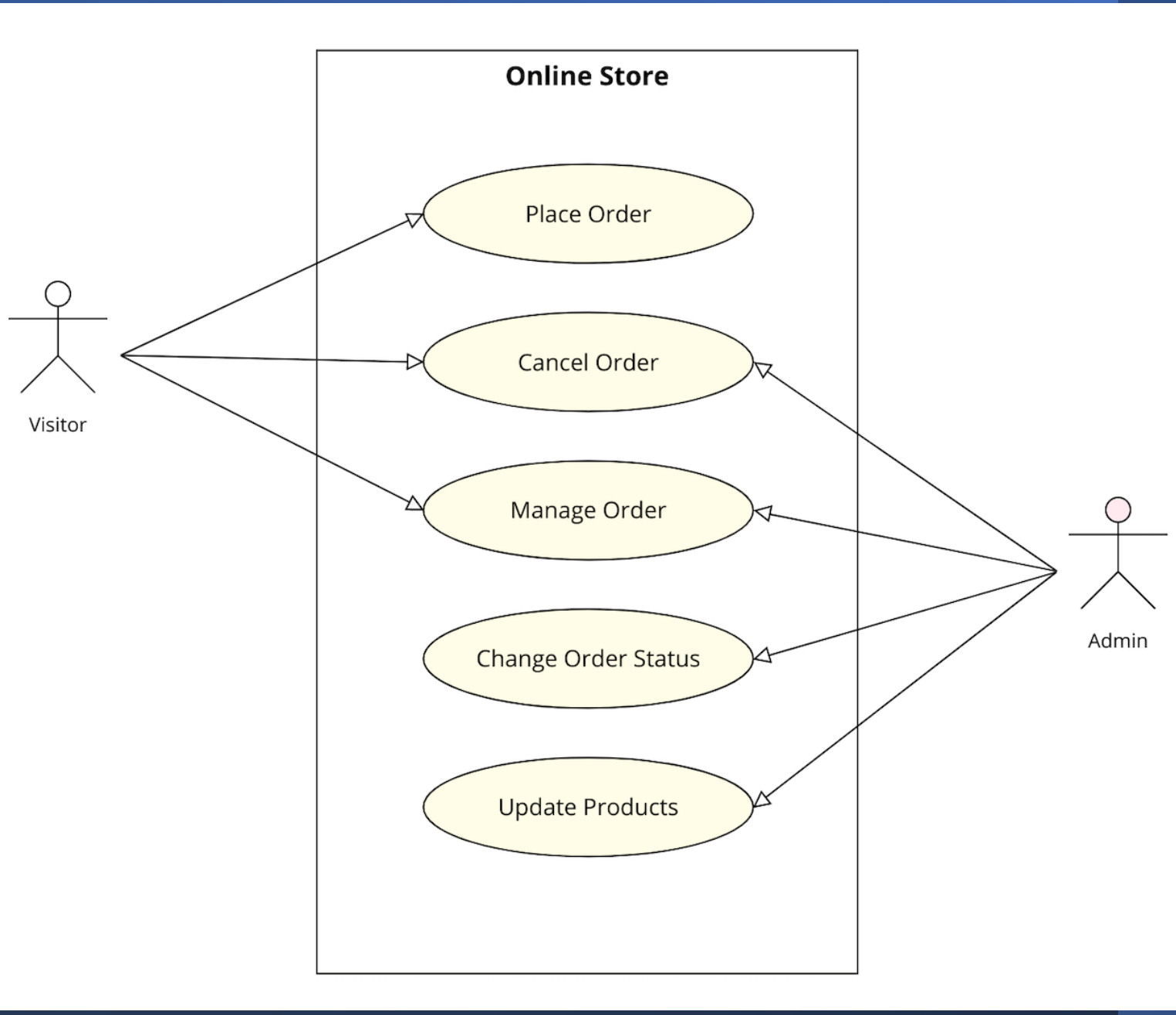
Use case diagram symbols and notation

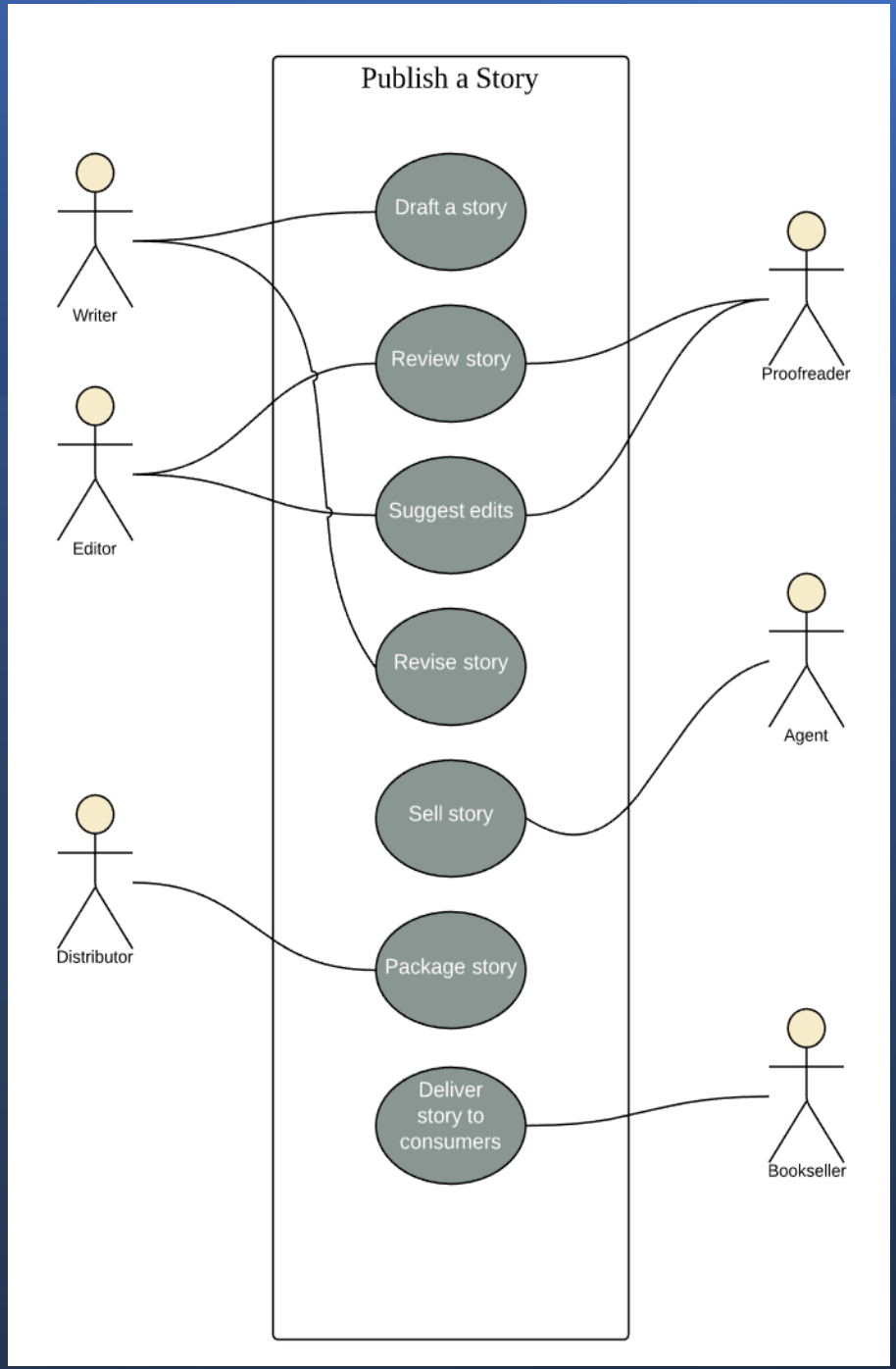
- **Use cases**: Horizontally shaped ovals that represent the different uses that a user might have.
- **Actors**: Stick figures that represent the people actually employing the use cases.
- **Associations**: A line between actors and use cases. In complex diagrams, it is important to know which actors are associated with which use cases.
- **System boundary boxes**: A box that sets a system scope to use cases. All use cases outside the box would be considered outside the scope of that system.

•**Include Use Case** - The include use case never stand alone. When an actor initiates any base use case then base use case executes included use case.

•**Extend Use Case** - The base use case may stand alone, but under certain conditions, its behavior may be extended by behavior of another use case.




Online Store
Place Order
Cancel Order
Manage Order
Change Order Status
Update Products
Visitor
Admin


Publish a Story
Draft a story
Review story
Suggest edits
Revise story
Sell story
Package story
Deliver story to consumers
Writer
Editor
Distributor
Proofreader
Agent
Bookseller

2.Class Diagram

The class diagram depicts a static view of an application. It represents the types of objects residing in the system and the relationships between them. A class consists of its objects, and also it may inherit from other classes. A class diagram is used to visualize, describe, document various different aspects of the system, and also construct executable software code.

It shows the attributes, classes, functions, and relationships to give an overview of the software system. It constitutes class names, attributes, and functions in a separate compartment that helps in software development. Since it is a collection of classes, interfaces, associations, collaborations, and constraints, it is termed as a structural 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.

Purpose of Class Diagrams

1.It analyses and designs a static view of an application.
2.It describes the major responsibilities of a system.
3.It is a base for component and deployment diagrams.
4.It incorporates forward and reverse engineering.

Benefits of Class Diagrams

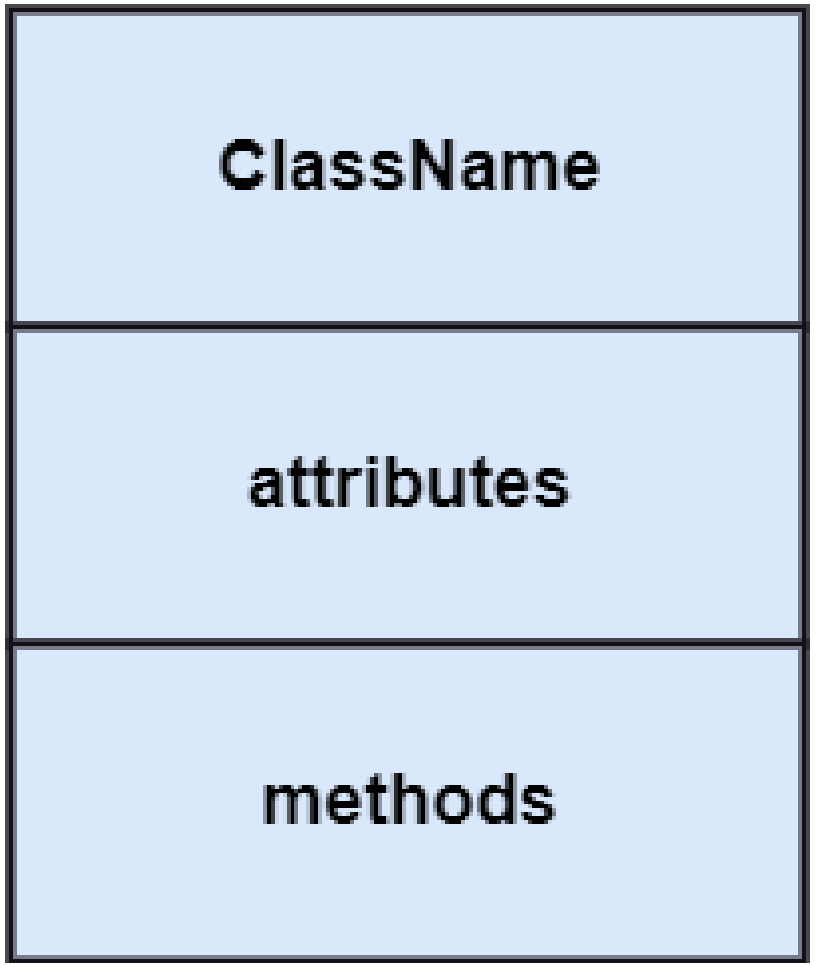
1.It can represent the object model for complex systems.
2.It reduces the maintenance time by providing an overview of how an application is structured before coding.
3.It provides a general schematic of an application for better understanding.
4.It represents a detailed chart by highlighting the desired code, which is to be programmed.
5.It is helpful for the stakeholders and the developers.

Basic components of a class diagram

The standard class diagram is composed of three sections:

- Upper section: Contains the name of the class. This section is always required, whether you are talking about the classifier or an object.
- Middle section: Contains the attributes of the class. Use this section to describe the qualities of the class. This is only required when describing a specific instance of a class.
- Bottom section: Includes class operations (methods). Displayed in list format, each operation takes up its own line. The operations describe how a class interacts with data.

- Each class is represented by a rectangle having a subdivision of three compartments class name, attributes, and methods(operations).
- There are three types of modifiers that are used to decide the visibility of attributes and operations.
    - + is used for public visibility(for everyone)
    - # is used for protected visibility (for friend and derived)
    - – is used for private visibility (for only me)

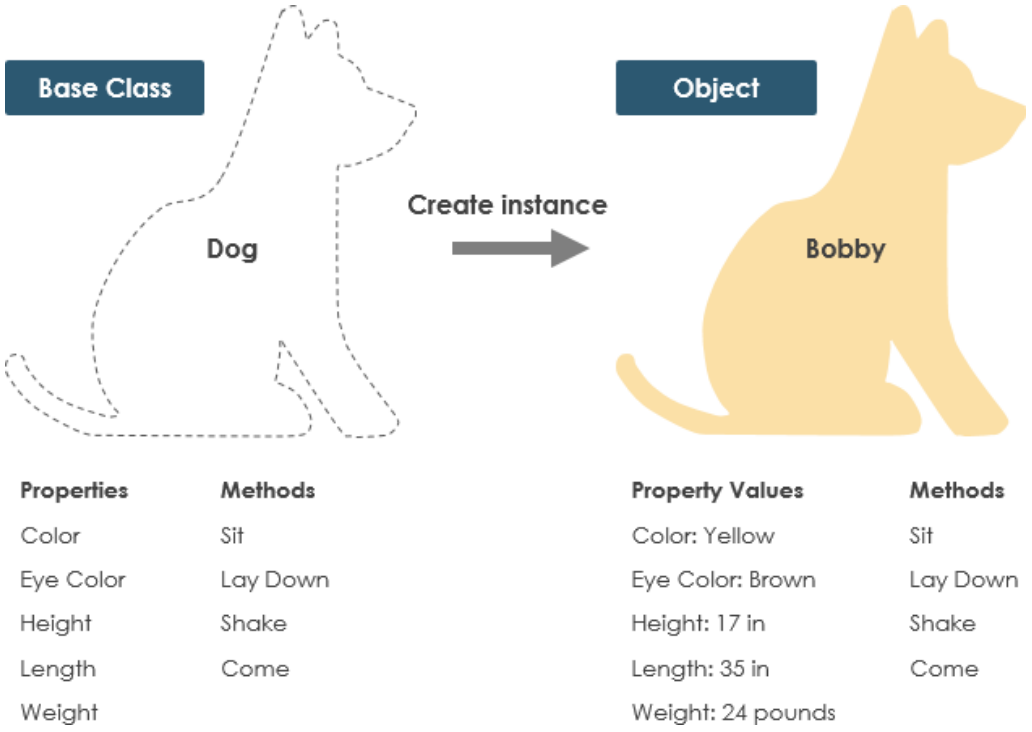


What is a Class?

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.

Example:

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

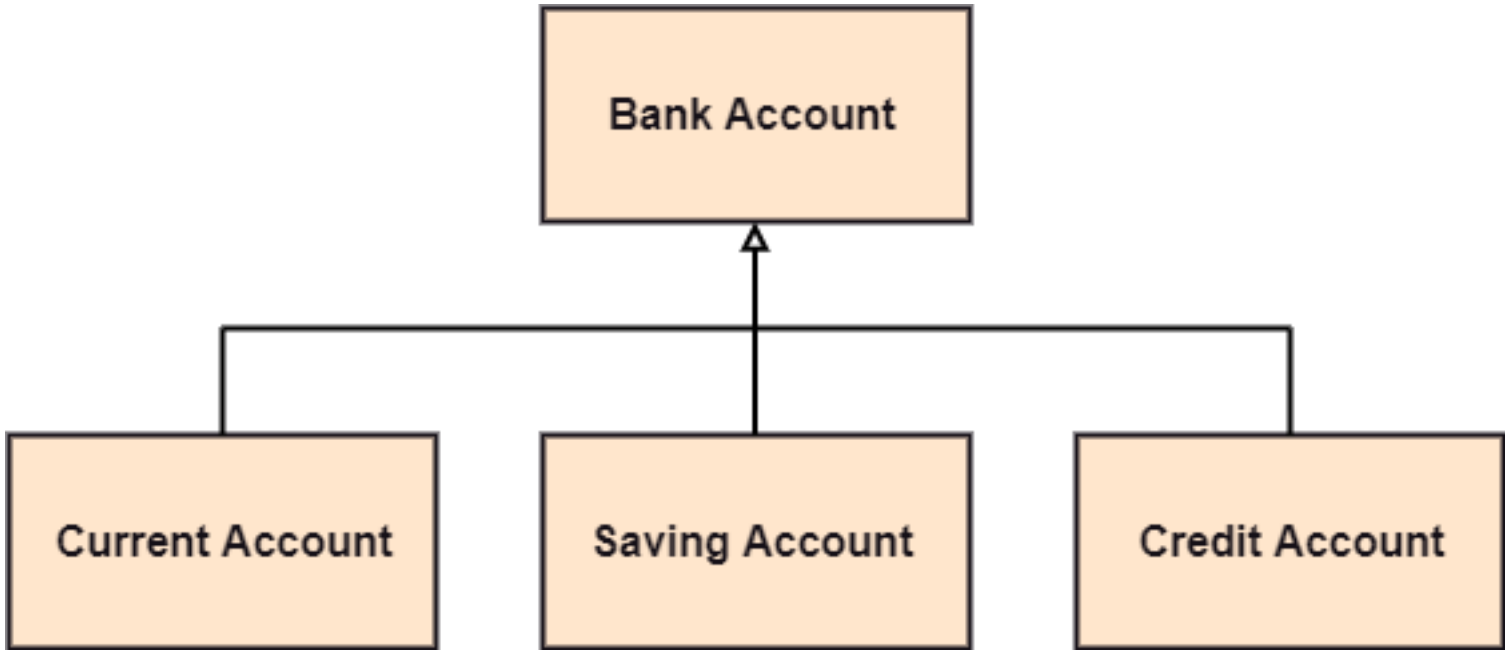


Relationships

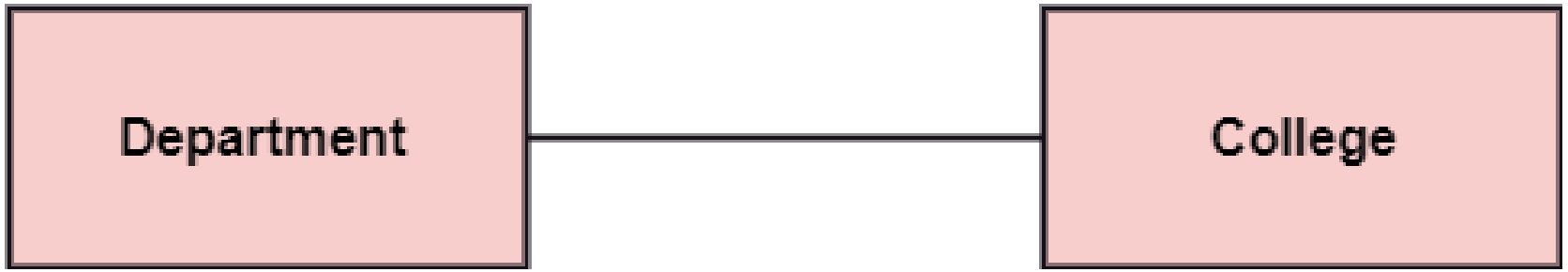
In UML, relationships are of three types:

1.Dependency: A dependency is a semantic relationship between two or more classes where a change in one class cause changes in another class.It forms a weaker relationship.
In the following example, Student_Name is dependent on the Student_Id.

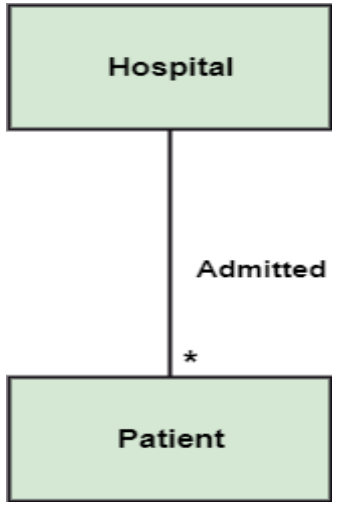
2.Generalization: A generalization is a relationship between a parent class (superclass) and a child class (subclass). In this, the child class is inherited from the parent class.
For example, The Current Account, Saving Account, and Credit Account are the generalized form of Bank Account.



3.Association: It describes a static or physical connection between two or more objects. It depicts how many objects are there in the relationship. For example, a department is associated with the college.

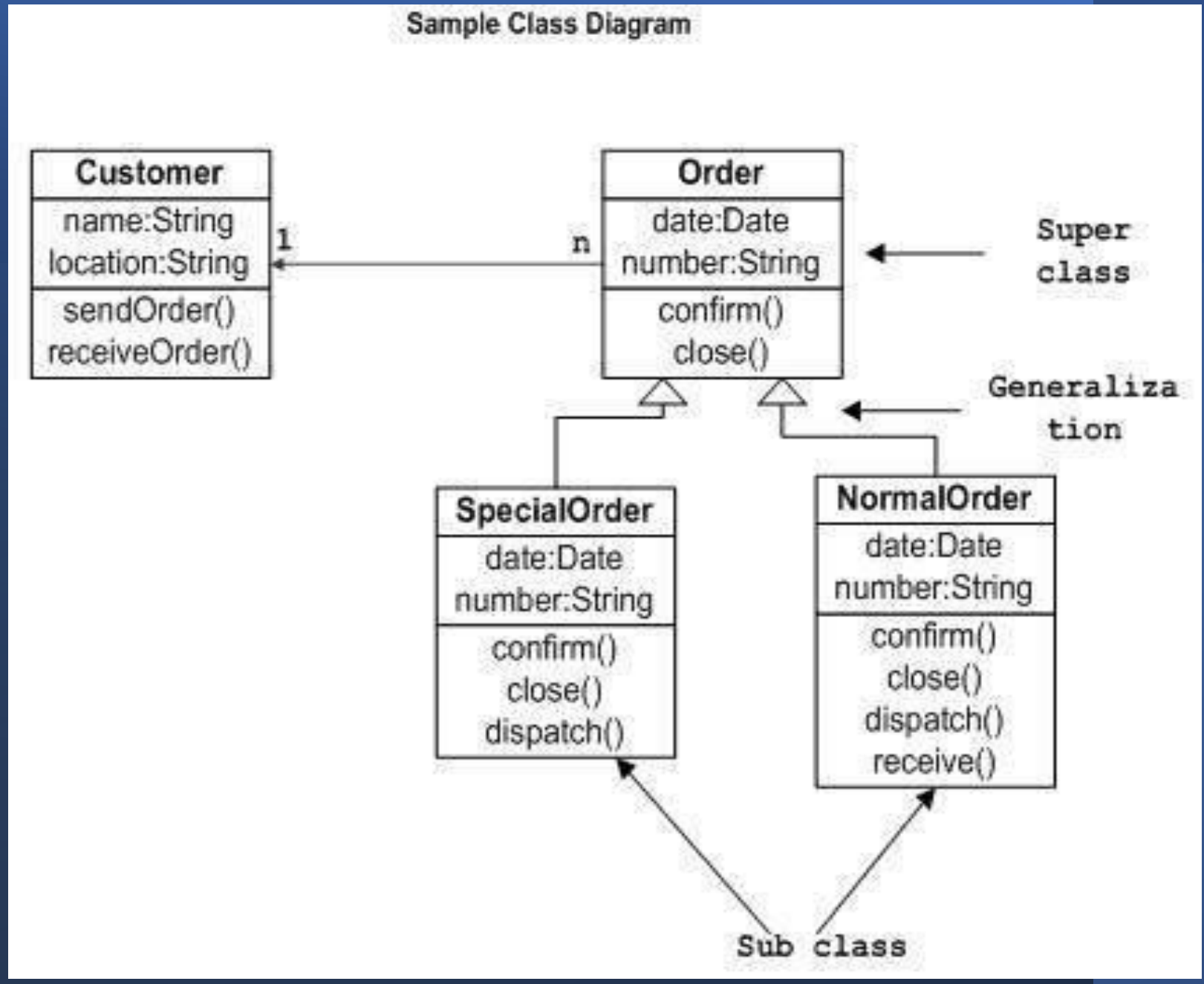


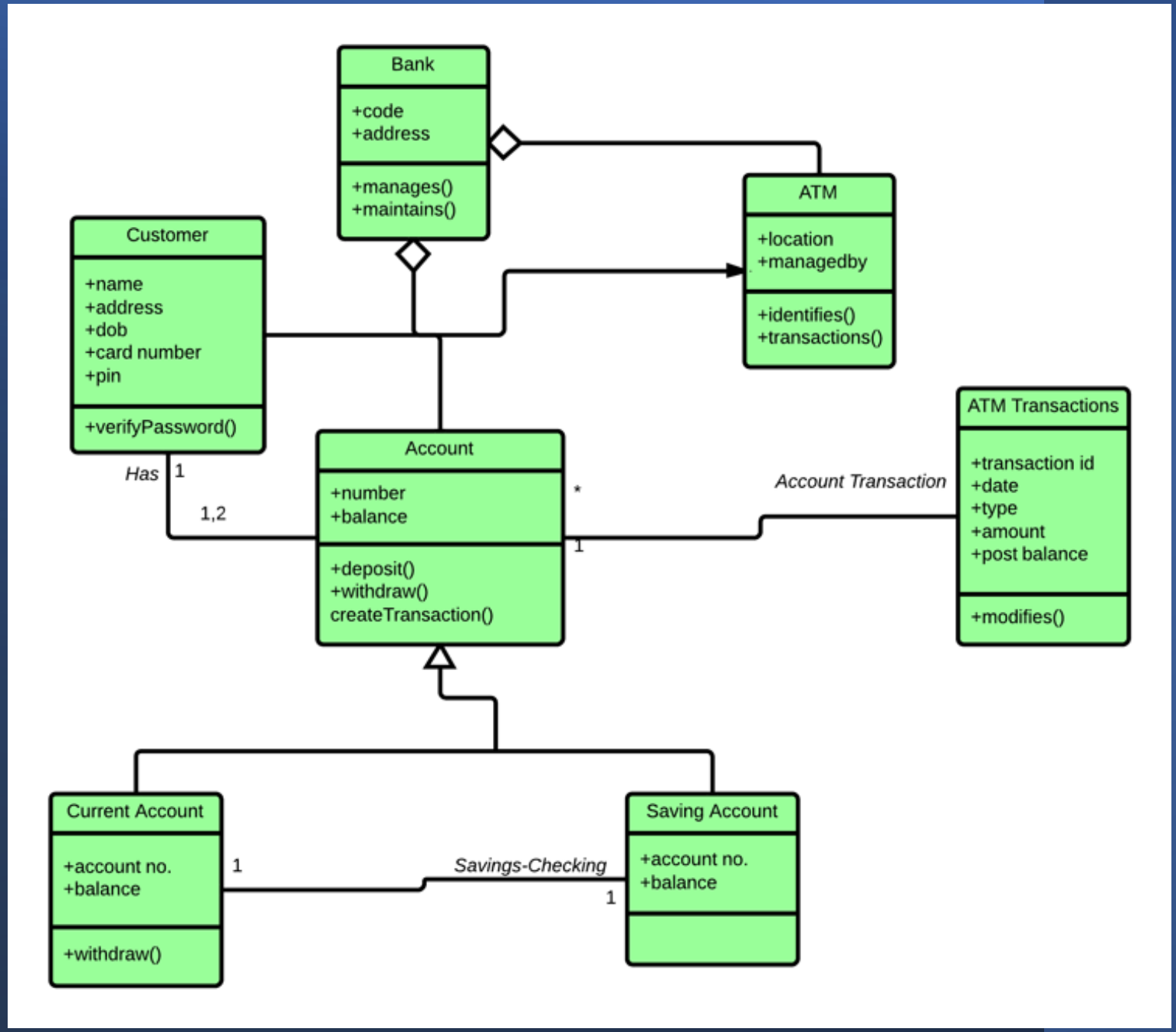
Multiplicity: It defines a specific range of allowable instances of attributes. In case if a range is not specified, one is considered as a default multiplicity.
For example, multiple patients are admitted to one hospital.



Aggregation: An aggregation is a subset of association, which represents has a relationship. It is more specific then association. It defines a part-whole or part-of relationship. In this kind of relationship, the child class can exist independently of its parent class.
The company encompasses a number of employees, and even if one employee resigns, the company still exists.

Composition: The composition is a subset of aggregation. It portrays the dependency between the parent and its child, which means if one part is deleted, then the other part also gets discarded. It represents a whole-part relationship.
A contact book consists of multiple contacts, and if you delete the contact book, all the contacts will be lost.


Sample Class Diagram
Customer
name:String
location:String
sendOrder()
receiveOrder()
1
n
Order
date:Date
number:String
confirm()
close()
Super class
Generalization
SpecialOrder
date:Date
number:String
confirm()
close()
dispatch()
NormalOrder
date:Date
number:String
confirm()
close()
dispatch()
receive()
Sub class


Bank
+code
+address
+manages()
+maintains()
ATM
+location
+managedby
+identifies()
+transactions()
Customer
+name
+address
+dob
+card number
+pin
+verifyPassword()
Has
1
1,2
Account
+number
+balance
+deposit()
+withdraw()
createTransaction()
*
1
Account Transaction
ATM Transactions
+transaction id
+date
+type
+amount
+post balance
+modifies()
Current Account
+account no.
+balance
+withdraw()
1
Savings-Checking
1
Saving Account
+account no.
+balance

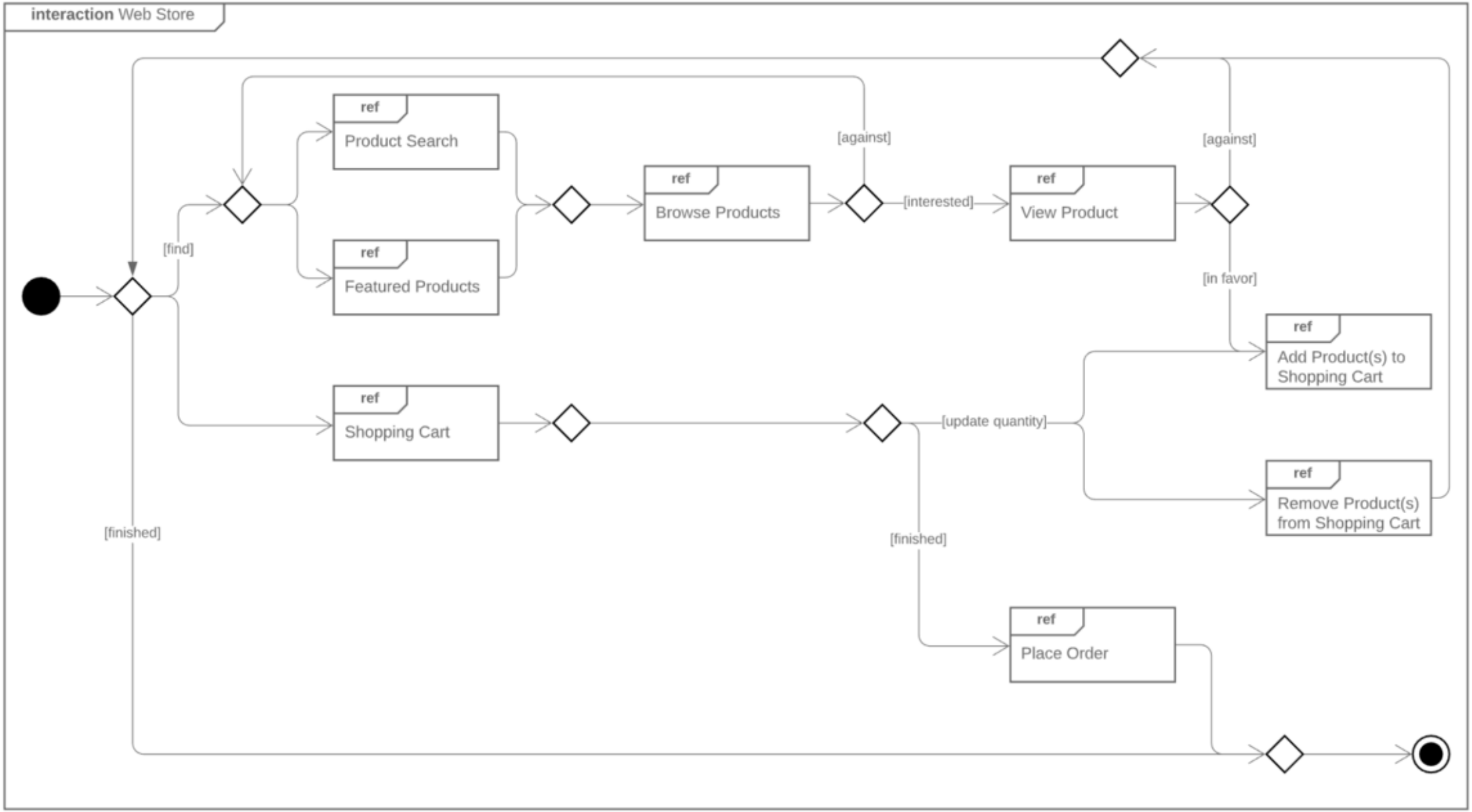
3.INTERACTION DIAGRAM

As its name might suggest, an interaction diagram is a type of UML diagram that's used to capture the interactive behavior of a system. Interaction diagrams focus on describing the flow of messages within a system, providing context for one or more lifelines within a system. In addition, interaction diagrams can be used to represent the ordered sequences within a system and act as a means of visualizing real-time data via UML.

The interaction diagram helps to envision the interactive (dynamic) behavior of any system. It portrays how objects residing in the system communicates and connects to each other. It also provides us with a context of communication between the lifelines inside the system.

Following are the purpose of an interaction diagram given below:

1.To visualize the dynamic behavior of the system.
2.To envision the interaction and the message flow in the system.
3.To portray the structural aspects of the entities within the system.
4.To represent the order of the sequenced interaction in the system.
5.To visualize the real-time data and represent the architecture of an object-oriented system.


interaction Web Store
ref
Product Search
ref
Featured Products
ref
Browse Products
ref
View Product
ref
Shopping Cart
ref
Add Product(s) to Shopping Cart
ref
Remove Product(s) from Shopping Cart
ref
Place Order
[find]
[against]
[interested]
[against]
[in favor]
[update quantity]
[finished]
[finished]

Types of interaction diagrams in UML

Interaction diagrams are divided into four main types of diagrams:

- Communication diagram
- Sequence diagram
- Timing diagram
- Interaction overview diagram

Each type of diagram focuses on a different aspect of a system’s behavior or structure.

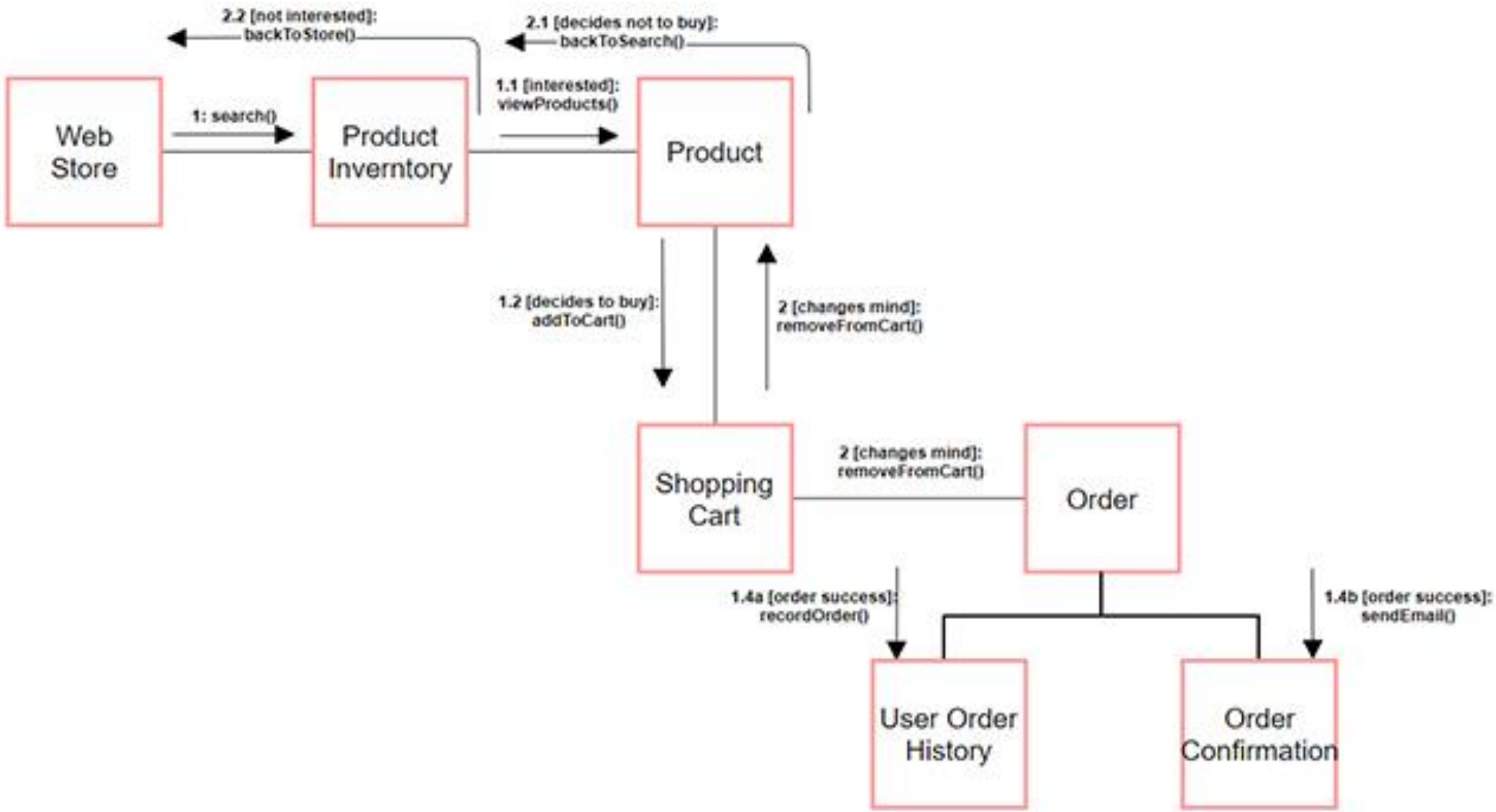
3.1 Communication diagram (or collaboration diagram)

In UML, communication diagrams depict the relationships and interactions among various software objects. They emphasize the structural aspects of an interaction diagram, focusing on object architecture rather than the flow of messages.

A communication diagram provides the following benefits:
- They emphasize how lifelines connect.
- They focus on elements within a system rather than message flow.
- They provide an added emphasis on organization over timing.

Communication diagrams can also have these possible downsides:
- They can become very complex.
- They make it difficult to explore specific objects within a system.
- They can be time-consuming to create.

Online Store Communication Diagram



3.2 Sequence diagram

Another option for depicting interactions is using sequence diagrams. These diagrams revolve around five main events:

- Order placement
- Payment
- Order confirmation
- Order preparation
- Order deliver

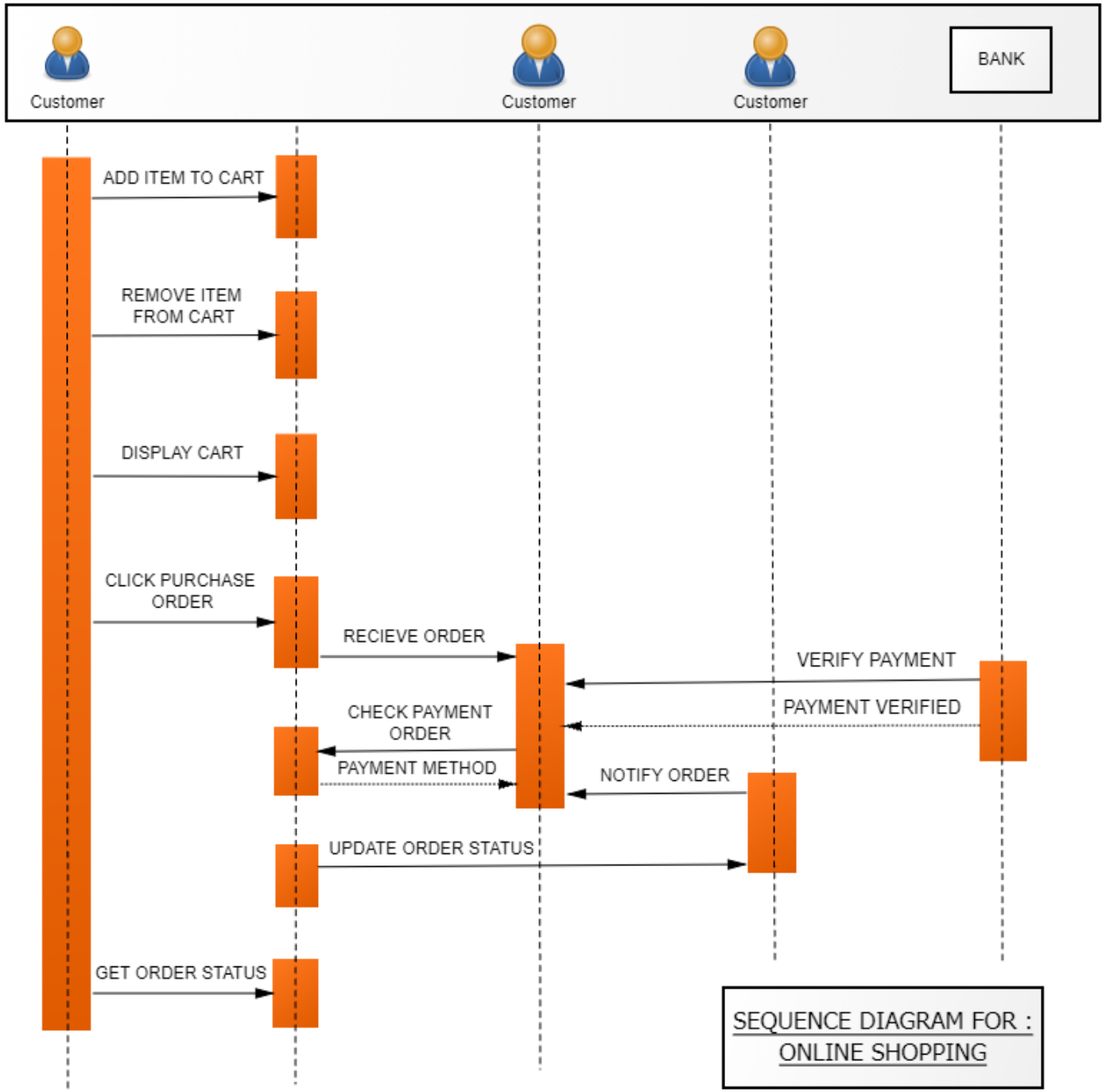
If the sequence of events changes, it can cause delays, or the system may crash. It's important to select the notation that matches the particular sequence within your diagram.

A sequence diagram provides the following benefits:

- They're easy to maintain and generate.
- They're easy to update according to changes in a system.
- They allow for reverse and forward engineering.

Sequence diagrams can also have these possible downsides:

- They can become complex, with too many lifelines and varied notations.
- They're easy to produce incorrectly and depend on your sequence being entered correctly.


Customer
Customer
Customer
BANK
ADD ITEM TO CART
REMOVE ITEM FROM CART
DISPLAY CART
CLICK PURCHASE ORDER
RECIEVE ORDER
VERIFY PAYMENT
PAYMENT VERIFIED
CHECK PAYMENT ORDER
PAYMENT METHOD
NOTIFY ORDER
UPDATE ORDER STATUS
GET ORDER STATUS
SEQUENCE DIAGRAM FOR :
ONLINE SHOPPING

3.3 Timing diagram

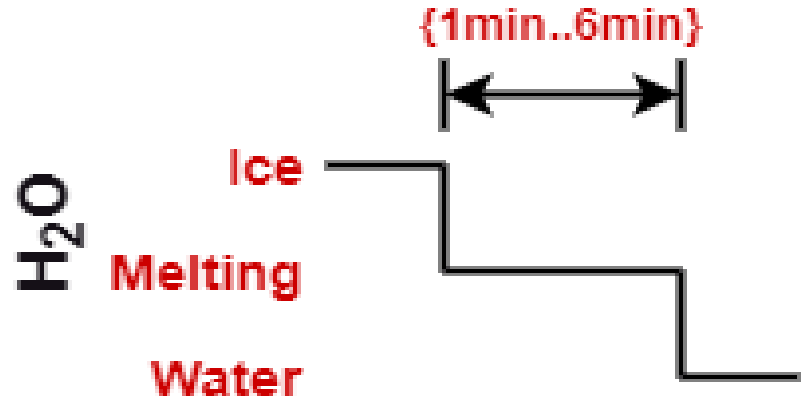
Another diagram option can be to use timing diagrams. These are visuals used to depict the state of a lifeline at any instance in time, denoting the changes in an object from one form to another. Waveforms are used within timing diagrams to visualize the flow within the software program at various instances of time.

A timing diagram offers the following benefits:

- They allow for forward and reverse engineering.
- They can represent the state of an object at an exact instance in time.
- They can keep track of any and all changes within a system.

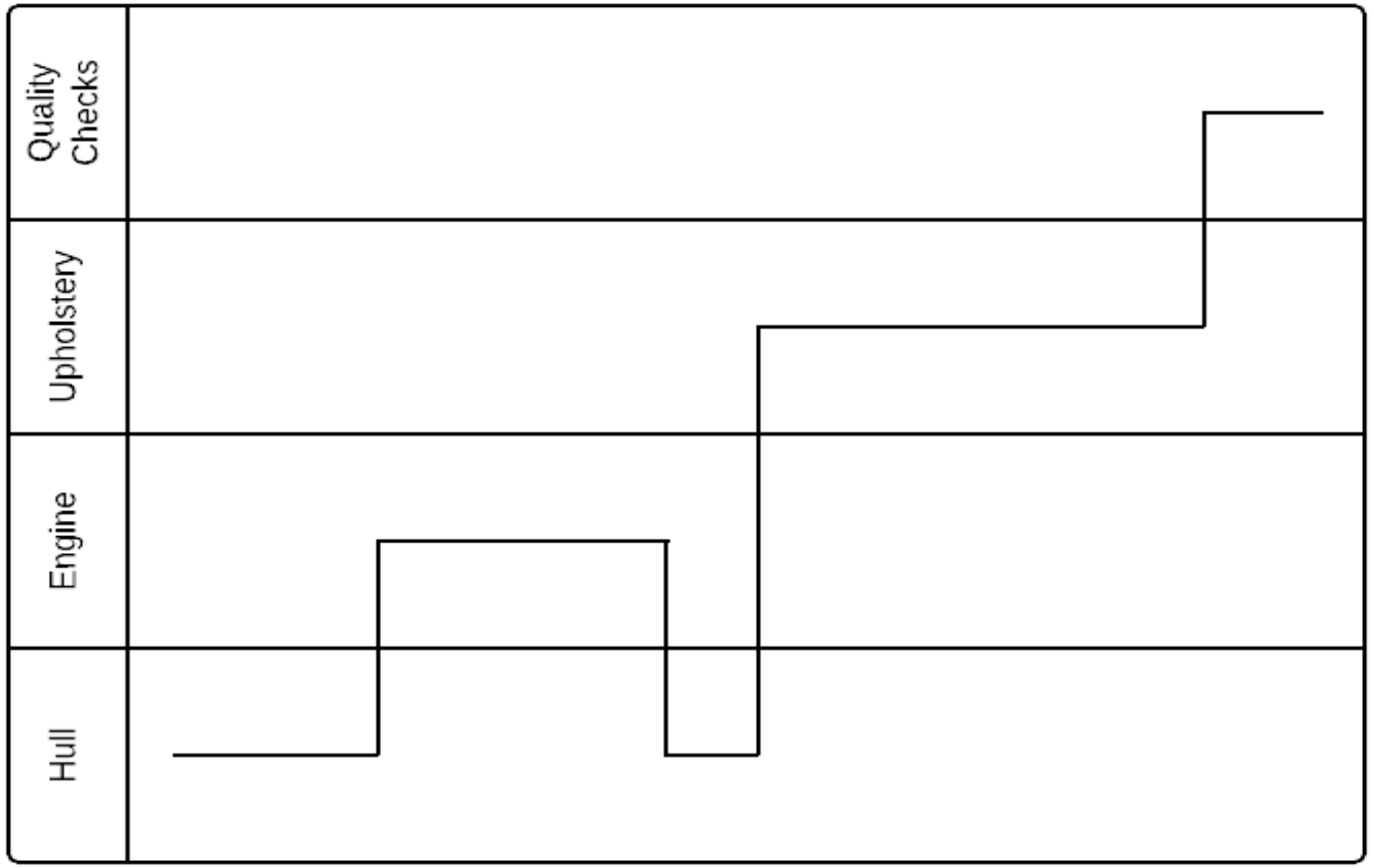
You should also consider these potential downsides of using a timing diagram:

- They can be difficult to understand.
- They can be hard to maintain over time.



Boat manufacturing timing diagram example

In this simplified example of a boat manufacturing plant, a timing diagram shows that too much time is spent on the upholstery stages of production. As a result, factory administrators may assign more employees to the upholstery stations or seek out ways to increase efficiency. If administrators can effectively use a timing diagram to increase efficiency, the process can be significantly improved, decreasing both time and money spent on the process.

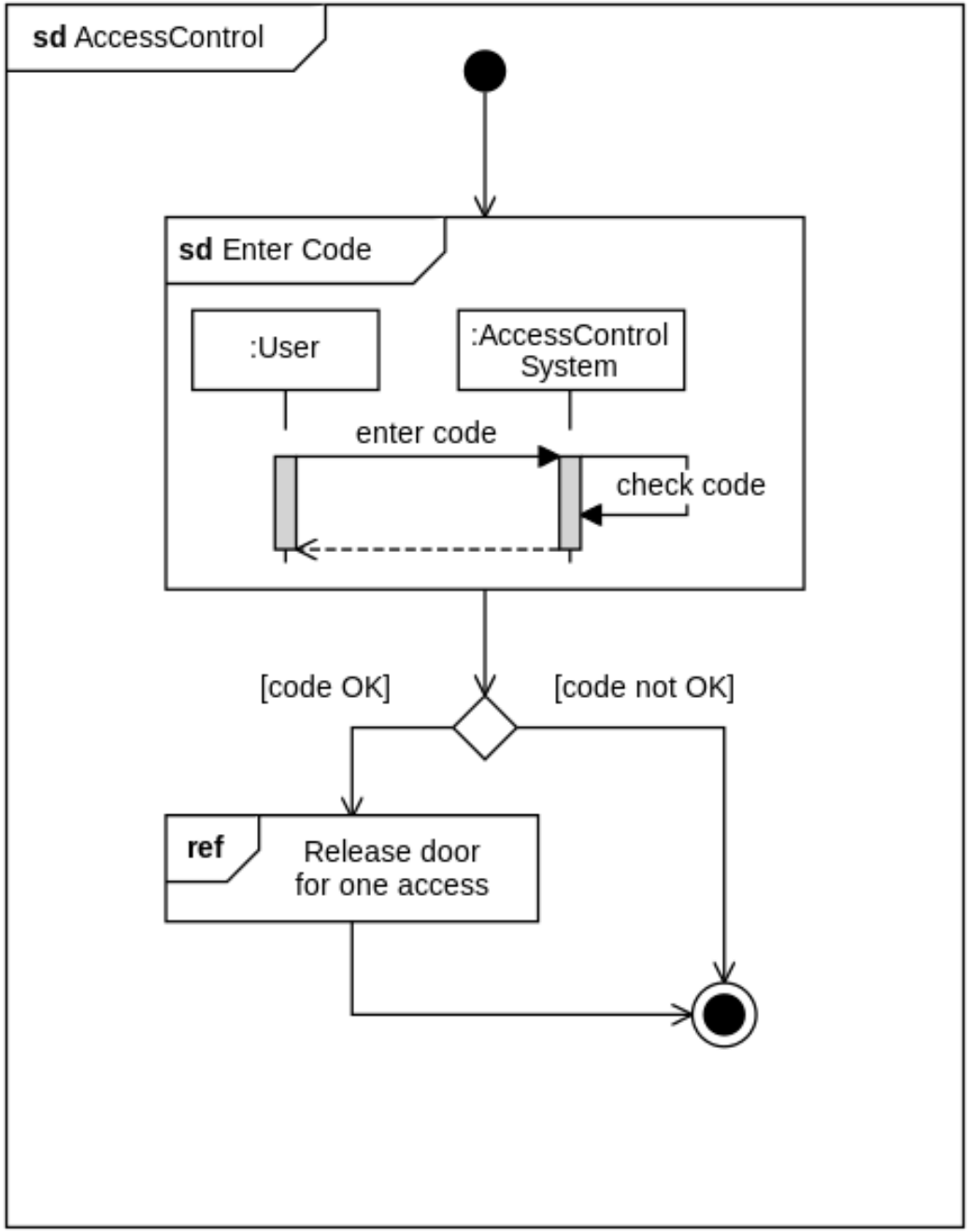


3.4 Interaction Overview Diagram

An Interaction Overview Diagram is a high-level diagram used to show the flow of interactions between different parts of a system or between various systems or components.

- It provides an overview of how various interactions, typically represented by sequence diagrams or communication diagrams, are organized and connected.
- Interaction overview diagrams are often used to show the overall structure of interactions in complex scenarios, making them easier to understand.
- They can include elements like decision nodes, merge nodes, and interaction fragments to represent conditional flows and loops within the interactions.
- Interaction overview diagrams are especially useful when you want to present a simplified view of complex interactions.
- Interaction overview diagrams focus on the overview of the flow of control where the nodes are **interactions** (sd) or **interaction use** (ref).

Interaction(sd)
An Interaction diagram of any kind may appear inline as an Activity Invocation.
Interaction Use(ref)
Large and complex sequence diagrams could be simplified with interaction uses. It is also common to reuse some interaction between several other interactions.



4. State Machine Diagram

The state machine diagram is also called the Statechart or State Transition diagram, which shows the order of states underwent by an object within the system. It captures the software system's behavior. It models the behavior of a class, a subsystem, a package, and a complete system.

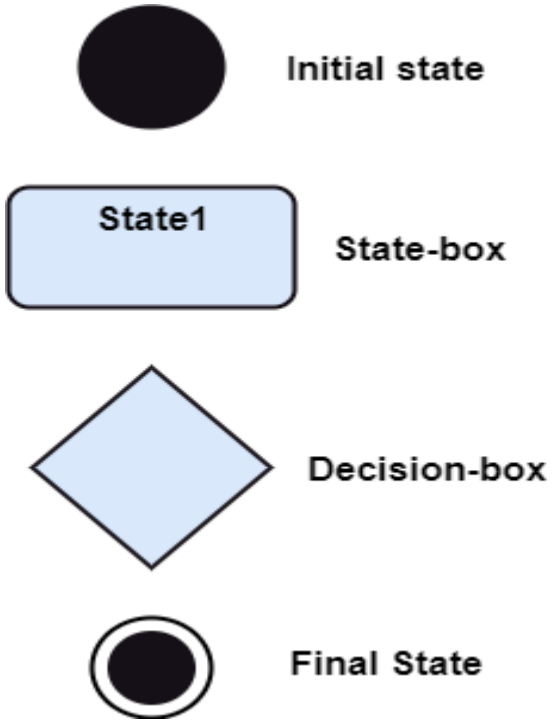
Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. The most important purpose of Statechart diagram is to model lifetime of an object from creation to termination.

Following are the main purposes of using Statechart diagrams −

- To model the dynamic aspect of a system.
- To model the life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model the states of an object.

How to Draw a State-chart Diagram?

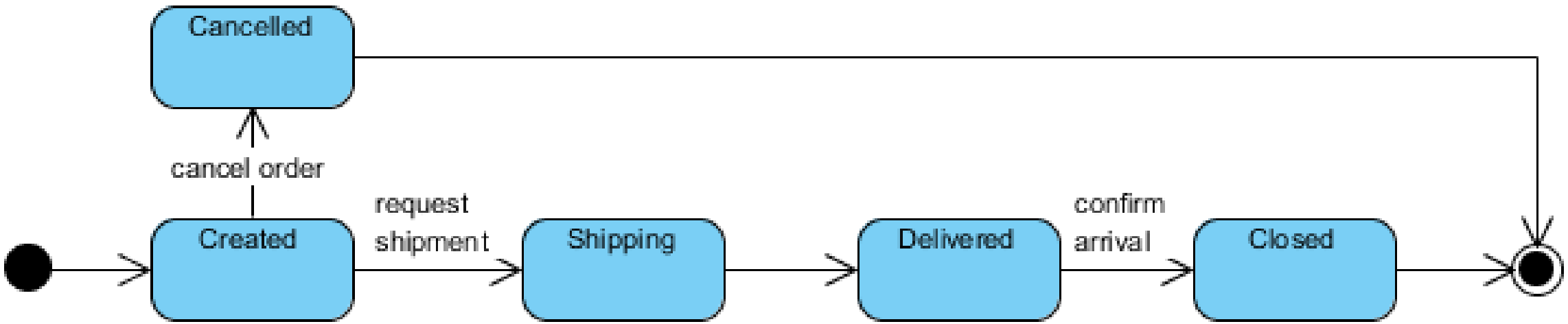
- Identify the important objects to be analyzed.
- Identify the states.
- Identify the events.



Notation of a State Machine Diagram

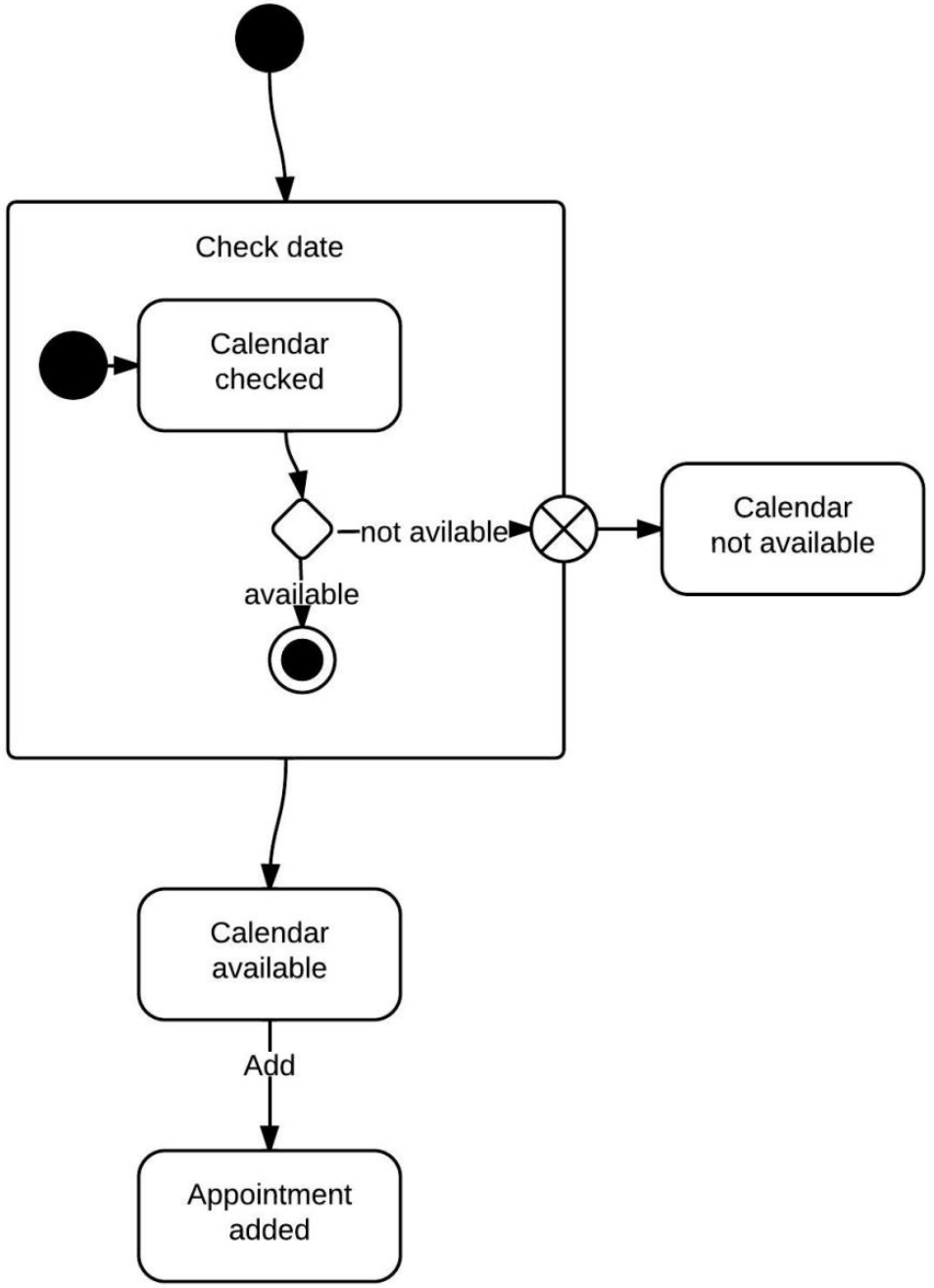
Following are the notations of a state machine diagram enlisted below:

1.Initial state: It defines the initial state (beginning) of a system, and it is represented by a black filled circle.
2.Final state: It represents the final state (end) of a system. It is denoted by a filled circle present within a circle.
3.Decision box: It is of diamond shape that represents the decisions to be made on the basis of an evaluated guard.
4.Transition: A change of control from one state to another due to the occurrence of some event is termed as a transition. It is represented by an arrow labeled with an event due to which the change has ensued.
5.State box: It depicts the conditions or circumstances of a particular object of a class at a specific point of time. A rectangle with round corners is used to represent the state box.


Cancelled
cancel order
Created
request
shipment
Shipping
Delivered
confirm
arrival
Closed

Calendar availability state diagram example

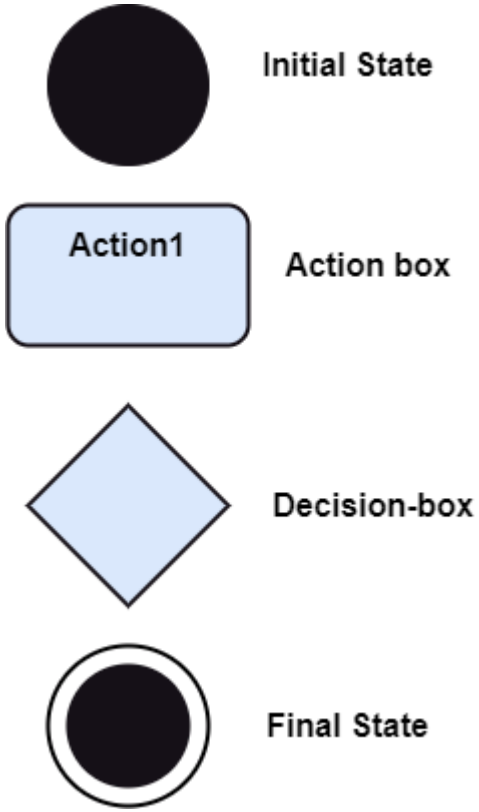
This state machine diagram example shows the process by which a person sets an appointment on their calendar. In the “Check date” composite state, the system checks the calendar for availability in a few different substates. If the time is not available on the calendar, the process will be escaped. If the calendar shows availability, however, the appointment will be added to the calendar.



5. ACTIVITY DIAGRAM

ACTIVITY DIAGRAM is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system. The basic purpose of activity diagrams is to capture the dynamic behavior of the system.. It is also called object-oriented flowchart.

The purpose of an activity diagram can be described as –
•Draw the activity flow of a system.
•Describe the sequence from one activity to another.
•Describe the parallel, branched and concurrent flow of the system.



Notation of an Activity diagram

Activity diagram constitutes following notations:
Initial State: It depicts the initial stage or beginning of the set of actions.
Final State: It is the stage where all the control flows and object flows end.
Decision Box: It makes sure that the control flow or object flow will follow only one path.
Action Box: It represents the set of actions that are to be performed.

Components of an Activity Diagram

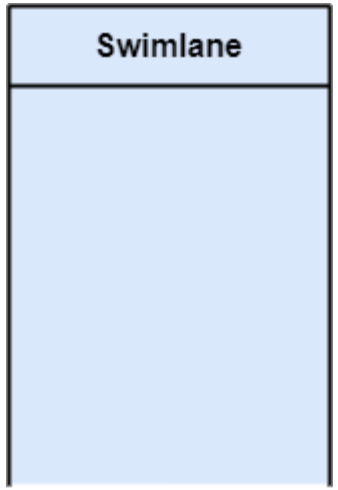
Activities

The categorization of behavior into one or more actions is termed as an activity. In other words, it can be said that an activity is a network of nodes that are connected by edges. The edges depict the flow of execution. It may contain action nodes, control nodes, or object nodes.

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.

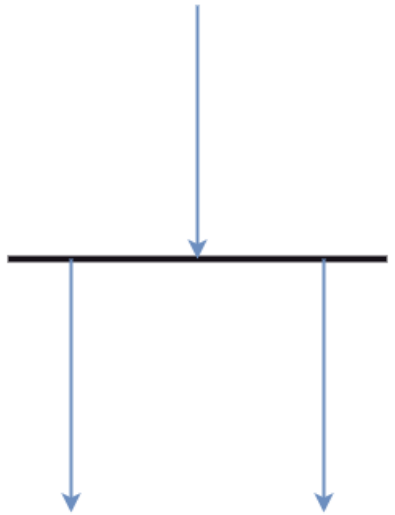
Activity partition /swimlane

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. But it is used to add more transparency to the activity diagram.



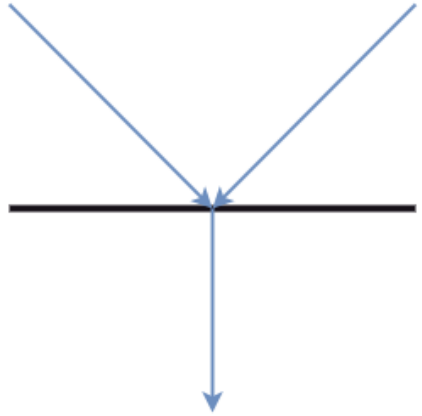
Forks

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

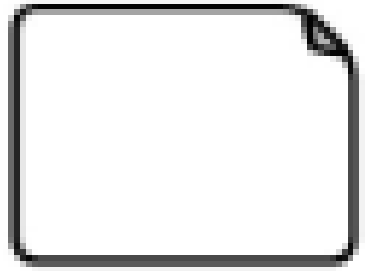


Join Nodes

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.



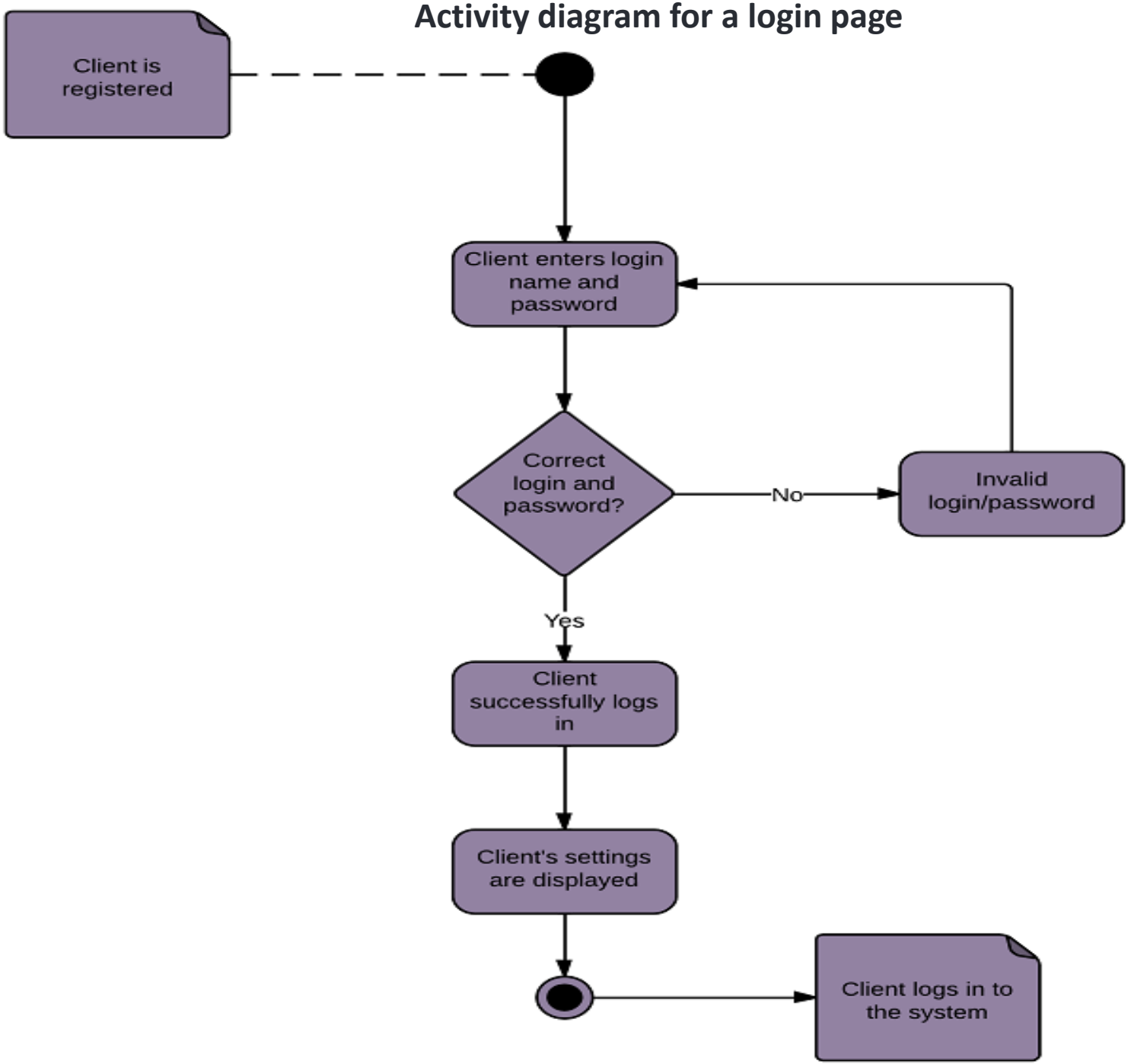
Note symbol
Allows the diagram creators or collaborators to communicate additional messages that don't fit within the diagram itself. Leave notes for added clarity and specification.



Send signal symbol
Indicates that a signal is being sent to a receiving activity.

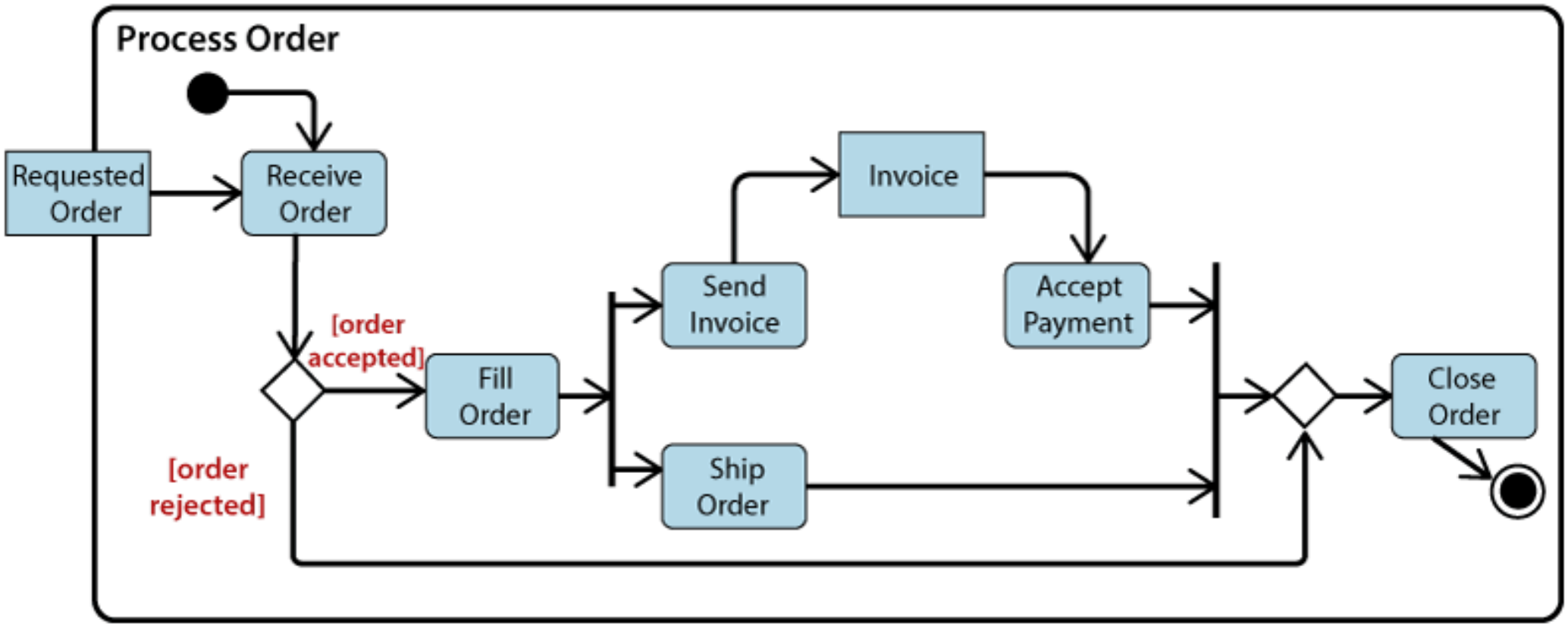
Receive signal symbol
Demonstrates the acceptance of an event. After the event is received, the flow that comes from this action is completed

Activity diagram for a login page



Example of an Activity Diagram

An example of an activity diagram showing the business flow activity of order processing is given below. Here the input parameter is the Requested order, and once the order is accepted, all of the required information is then filled, payment is also accepted, and then the order is shipped. It permits order shipment before an invoice is sent or payment is completed.



6. Package diagrams

Package diagrams are structural diagram which is commonly used to simplify complex class diagrams and organize classes into packages. A package is a collection of related UML elements including diagrams, documents, classes, and event packages. Aside from that, the package diagram offers valuable high-level visibility for large projects and systems.

A package in the Unified Modeling Language helps:

1.To group elements
2.To provide a namespace for the grouped elements
3.A package may contain other packages, thus providing for a hierarchical organization of packages.
4.UML elements can be grouped into packages.

Benefits of a package diagram

A well-designed package diagram provides numerous benefits to those looking to create a visualization of their UML system or project.

- They provide a clear view of the hierarchical structure of the various UML elements within a given system.
- These diagrams can simplify complex class diagrams into well-ordered visuals.
- They offer valuable high-level visibility into large-scale projects and systems.
- Package diagrams can be used to visually clarify a wide variety of projects and systems.
- These visuals can be easily updated assystems and projects evolve.

Basic components of a package diagram

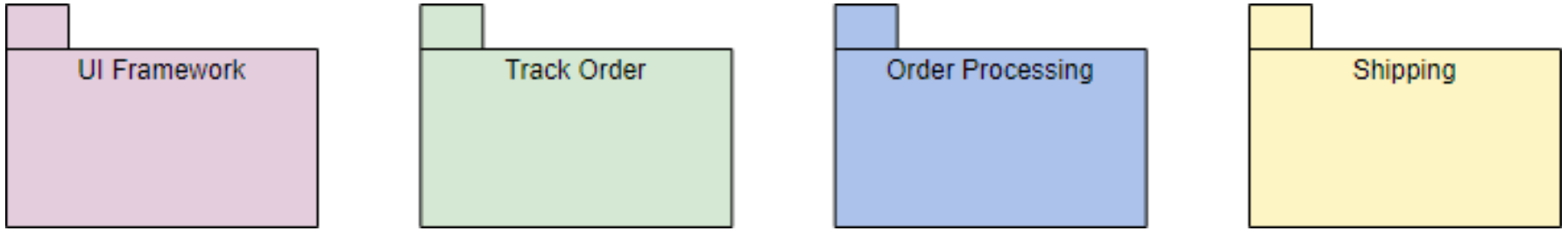
Package	Groups common elements based on data, behavior, or user interaction
Dependency	Depicts the relationship between one element (package, named element, etc) and another. Dependencies are divided into two groups: **access** and **import** dependencies.

<<import>> - one package imports the functionality of other package
<<access>> - one package requires help from functions of other package

EXAMPLE

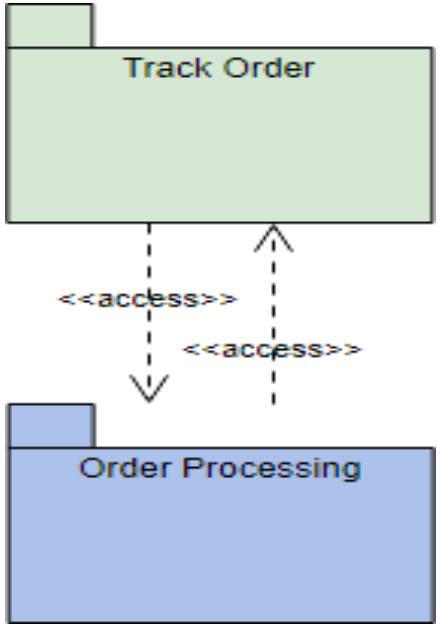
The following example shows the Track Order Service for an online shopping store. Track Order Service is responsible for providing tracking information for the products ordered by customers. Customer types in the tracking serial number, Track Order Service refers the system and updates the current shipping status to the customer.

Step 1 - Identify the packages present in the system
1.There is a **"track order"** service, it has to talk with other module to know about the order details, let us call it **"Order Processing"**.
2.Next after fetching Order Details it has to know about shipping details, let us call that as **"Shipping"**.
3.Finally if knows the status of the order it has to update the information to the user, let us call this module as **"UI Framework"**.

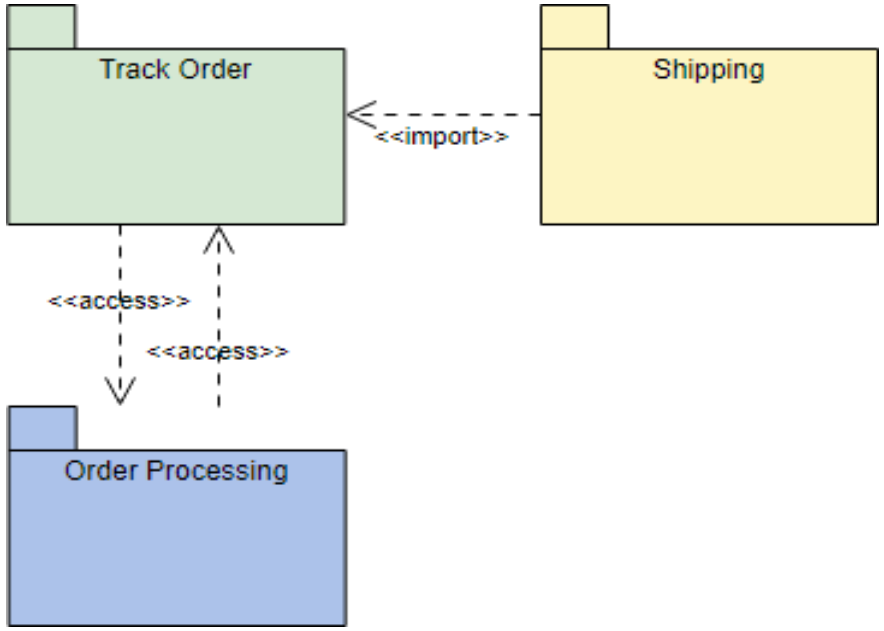


Step 2 - Identify the dependencies

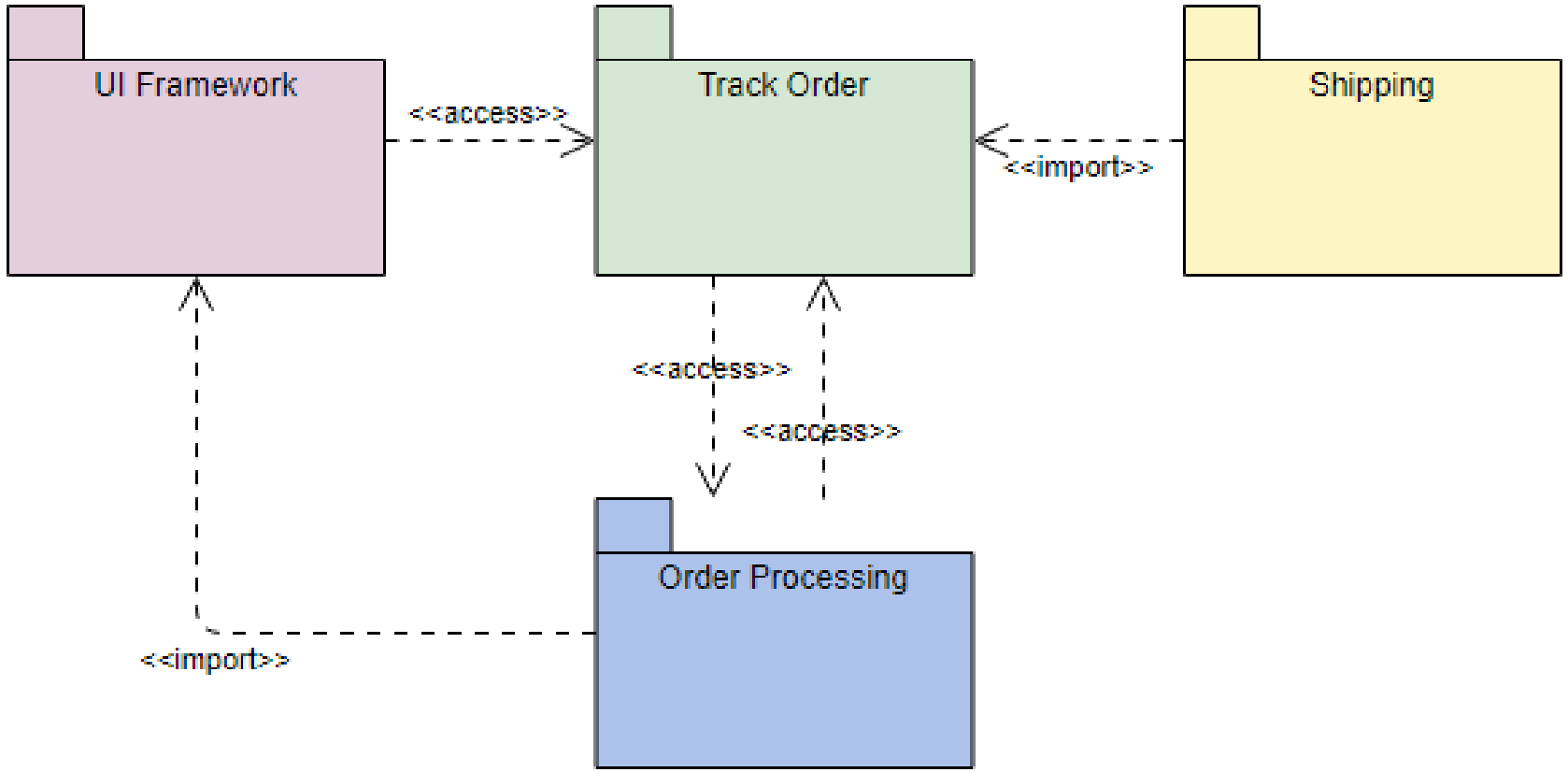
1."Track order" package should get order details from **"Order Processing"** and on the other hand, "Order Processing" also requires the tracking information from the "Track Order" package, thus, the two modules are accessing each other which suffices <<access>> dual dependency.



2.To know shipping information, "Shipping" requires to import "Track Order" to complete the shipping process.



Step 3 - Finally, Track Order dependency to UI Framework is also mapped in to the diagram which completes the Package Diagram for Track Order subsystem.



7. Component diagram

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc.

Component diagrams can also be described as a static implementation view of a system.

The purpose of the component diagram can be summarized as –

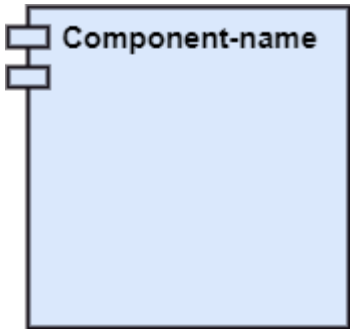
- To represent the components of any system at runtime.
- It helps during testing of a system.
- It visualizes the connection between various components.

Following are some artifacts that are needed to be identified before drawing a component diagram:

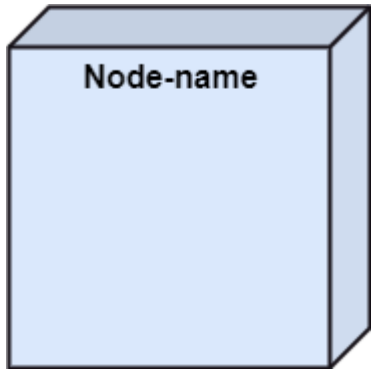
- Files used in the system.
- Libraries and other artifacts relevant to the application.
- Relationships among the artifacts.

Notation of a Component Diagram

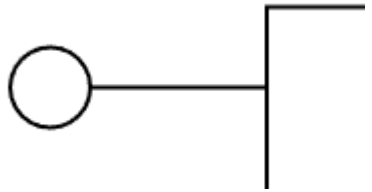
a) A component



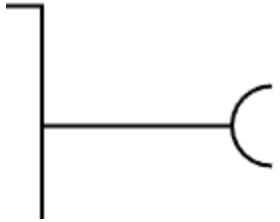
b) A node



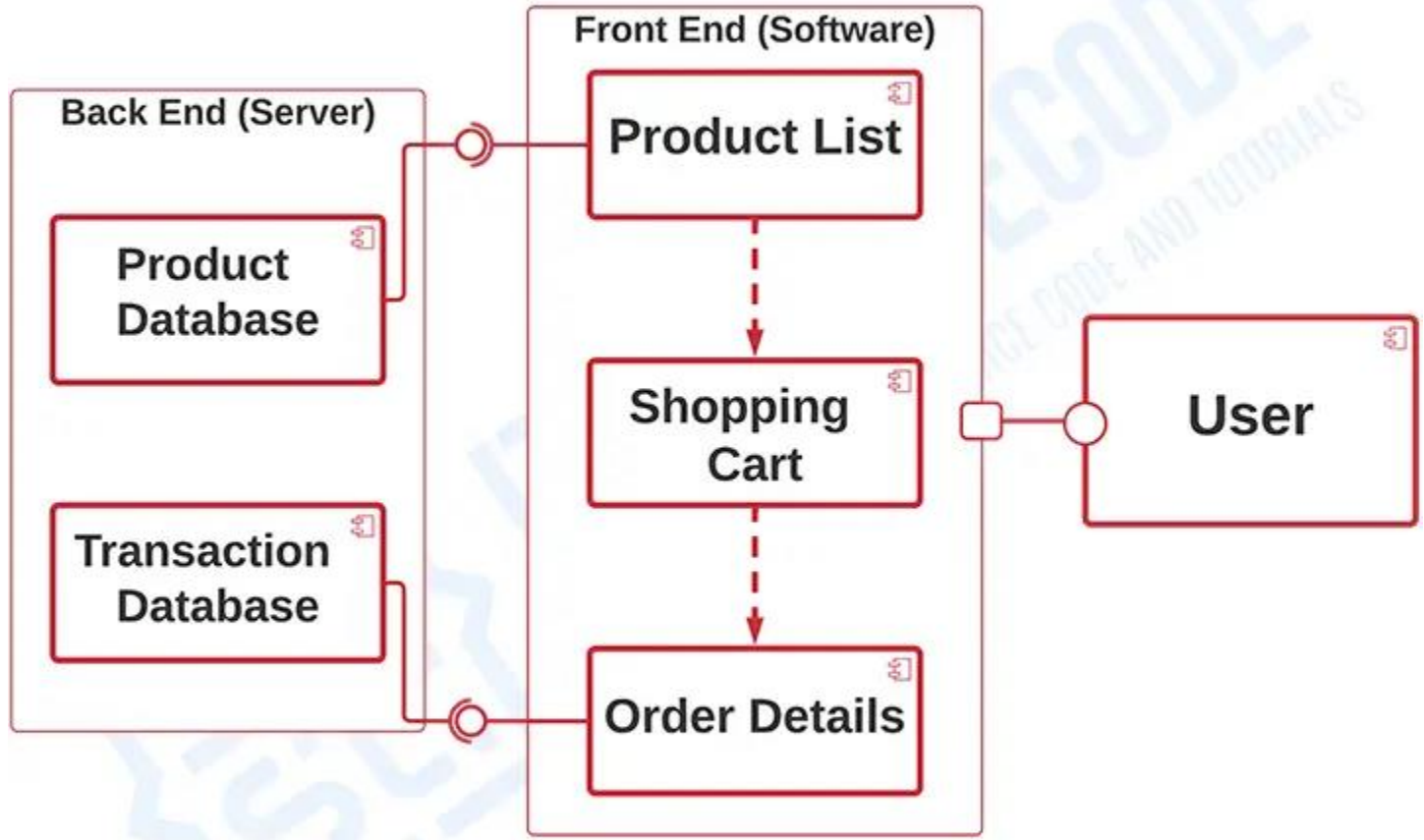
Provided interfaces:



Required interfaces:



ONLINE SHOPPING SYSTEM



COMPONENT DIAGRAM

8. Deployment Diagram

The deployment diagram visualizes the physical hardware on which the software will be deployed. It portrays the static deployment view of a system. It involves the nodes and their relationships.
It ascertains how software is deployed on the hardware. It maps the software architecture created in design to the physical system architecture, where the software will be executed as a node. Since it involves many nodes, the relationship is shown by utilizing communication paths.

Both the deployment diagram and the component diagram are closely interrelated to each other as they focus on software and hardware components. The component diagram represents the components of a system, whereas the deployment diagram describes how they are actually deployed on the hardware.

Following are the purposes of deployment diagram enlisted below:

1.To envision the hardware topology of the system.

2.To represent the hardware components on which the software components are installed.

3.To describe the processing of nodes at the runtime.

Symbol and notation of Deployment diagram

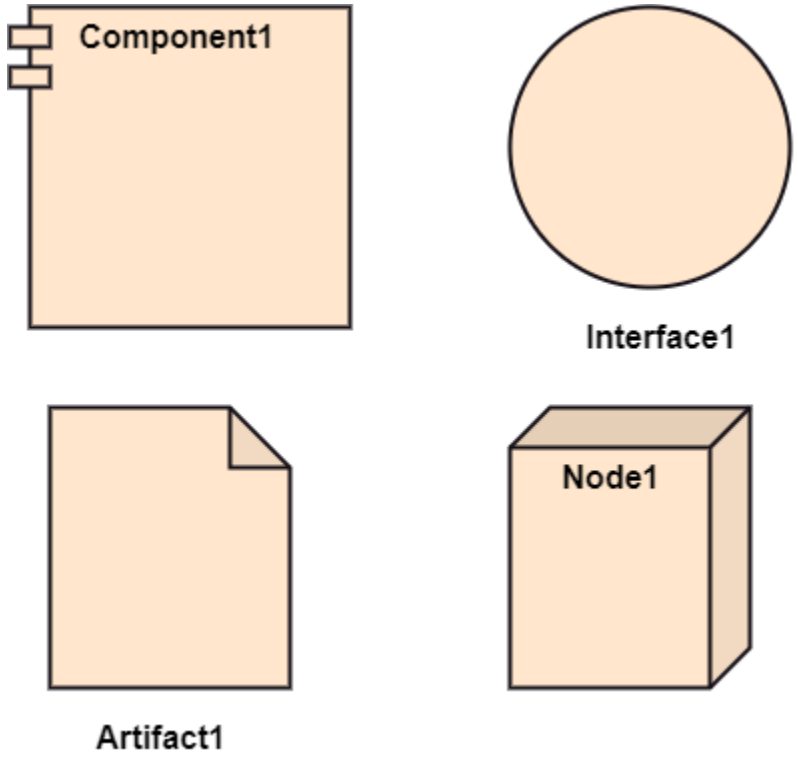
The deployment diagram consist of the following notations:

1.A component

2.An artifact

3.An interface

4.A node



NOTE: A node, represented as a cube, is a physical entity that executes one or more components, subsystems or executables. A node could be a hardware or software element.

Artifacts are concrete elements that are caused by a development process. Examples of artifacts are libraries, archives, configuration files, executable files etc.

A deployment diagram plays a critical role during the administrative process, and it must satisfy the following parameters,

- High performance
- Maintainability
- Scalability
- Portability
- Easily understandable

Deployment diagrams can be used –

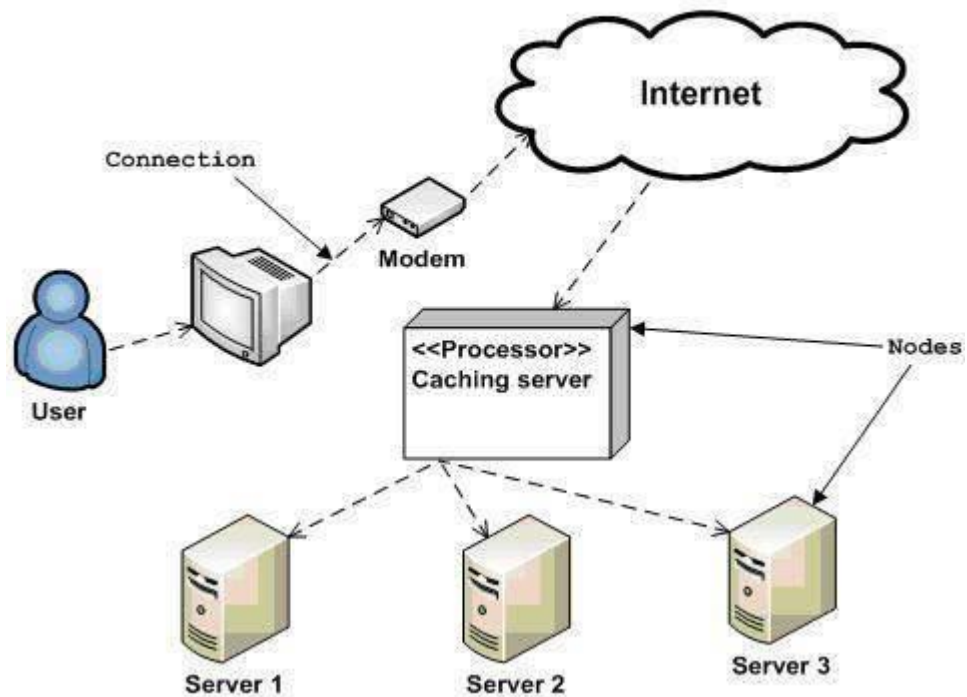
- To model the hardware topology of a system.
- To model the embedded system.
- To model the hardware details for a client/server system.
- To model the hardware details of a distributed application.
- For Forward and Reverse engineering.

Following is a sample deployment diagram to provide an idea of the deployment view of order management system. Here, we have shown nodes as –

•Monitor
•Modem
•Caching server
•Server

The application is assumed to be a web-based application, which is deployed in a clustered environment using server 1, server 2, and server 3. The user connects to the application using the Internet. The control flows from the caching server to the clustered environment.

Deployment diagram of an order management system



HOSPITAL MANAGEMENT SYSTEM