Unit - 1

Requirement Engineering:

Requirement Engineering is the process of proactively working with stakeholders to gather, articulate, and negotiate their needs, thereby establishing a clear scope and boundary for a project.

What are Requirements?

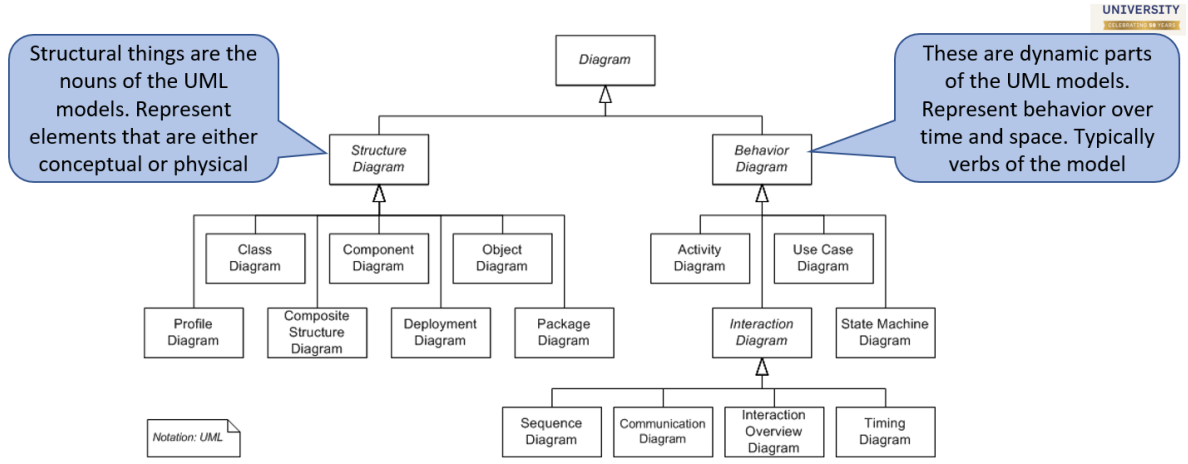
- Requirements are properties that software must exhibit to solve a particular problem.
- Requirements specify the externally visible behavior of what the system should do, not how it should do it.
- Requirements can be individual or a set of requirements.

Requirement Analysis Process:

1. **Understand Requirements:**
    - Dive deep into both product and process perspectives.
2. **Classification and Organization:**

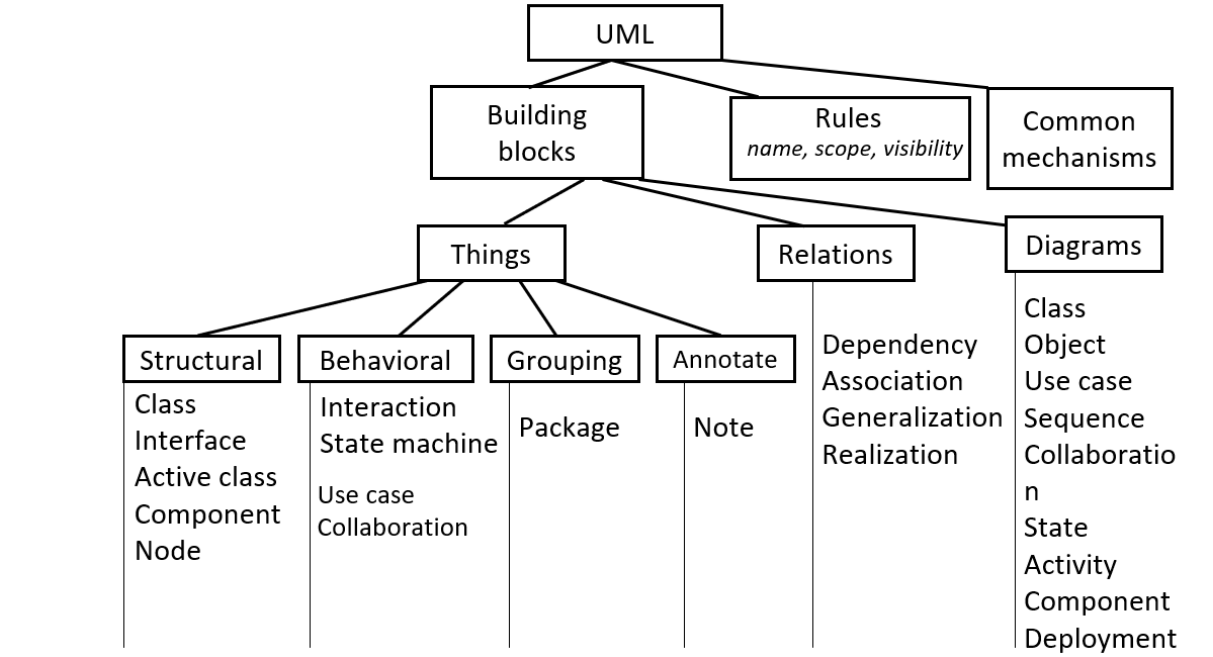
- Organize requirements into coherent clusters:
    - Functional, Non-Functional, and Domain requirements.
    - System and User requirements.

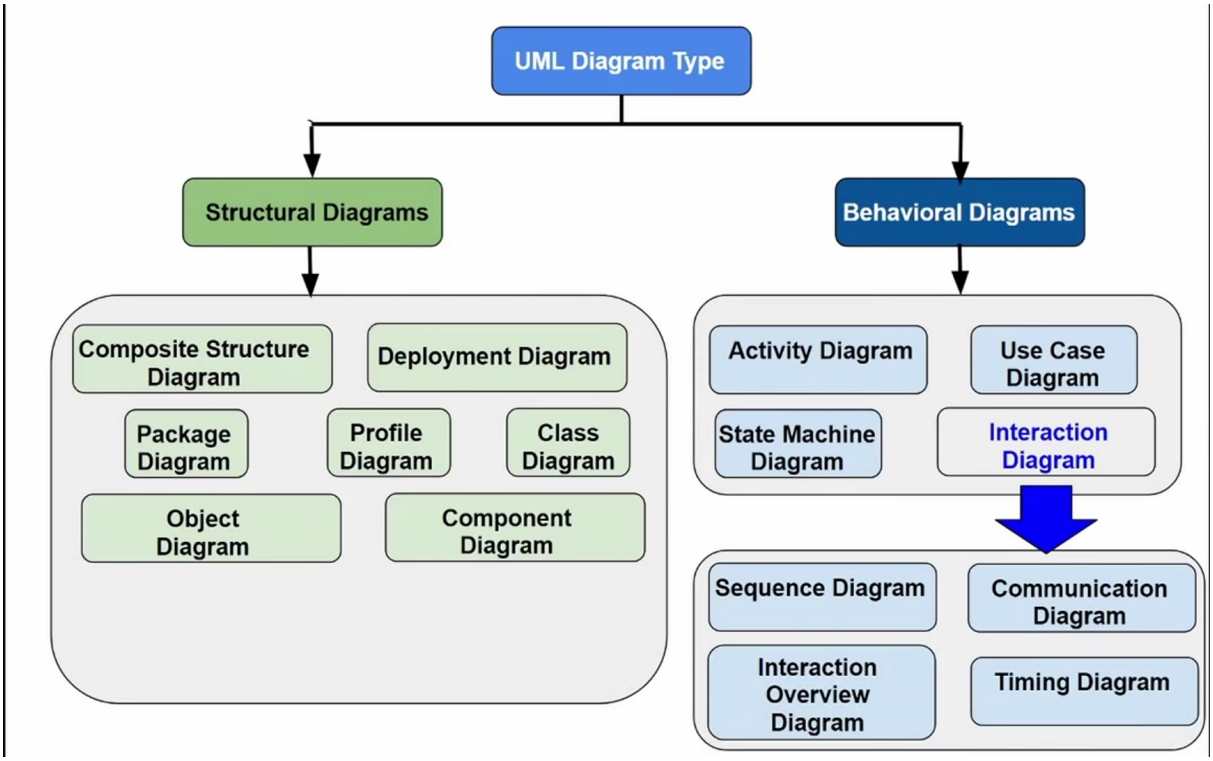
3. **Modeling Requirements:**
    - Create models to represent requirements.
4. **Analysis of Requirements:**
    - Analyze requirements, possibly using a fishbone diagram.
5. **Conflict Recognition and Resolution:**
    - Recognize and resolve conflicts (e.g., functionality vs. cost vs. timeliness).
6. **Requirement Negotiation:**
    - Negotiate requirements with stakeholders.
7. **Requirement Prioritization:**
    - Prioritize requirements using MoSCoW:
        - Must have
        - Should have
        - Could have
        - Won't have
8. **Risk Identification:**
    - Identify any potential risks associated with the requirements.
9. **Decision Making:**
    - Decide on Build vs. Buy (Commercial Off-The-Shelf Solution) and refine requirements accordingly.



Aspect	Structural Models	Behavioral Models
Focus	Focuses on the structure and organization of a system, representing its components and their relationships.	Focuses on the interactions and behaviors of components within a system.
Representation	Typically represented using diagrams such as class diagrams, entity-relationship diagrams, or architecture diagrams.	Often represented using diagrams such as state diagrams, activity diagrams, or sequence diagrams.
Emphasis	Emphasizes static aspects of a system.	Emphasizes dynamic aspects of a system.
Purpose	Primarily used for understanding system architecture, design, and composition.	Primarily used for understanding system behavior, interactions, and functionality.
Examples	Class diagrams, component diagrams, deployment diagrams.	State diagrams, activity diagrams, sequence diagrams.
Time Dimension	Generally time-independent.	Often incorporates the notion of time to represent dynamic behavior.
Abstraction Level	Usually higher-level abstraction.	May include both higher-level and detailed behavioral models.
Predictability	Provides insight into the structure of the system but may not predict actual system behavior accurately.	More closely aligns with predicting system behavior based on defined scenarios and interactions.

Analysis	Often used for system architecture analysis and design.	Often used for analyzing system behavior, identifying bottlenecks, and refining system functionality.





UML is made up of three conceptual elements. They are Building blocks , Rules and Common Mechanisms

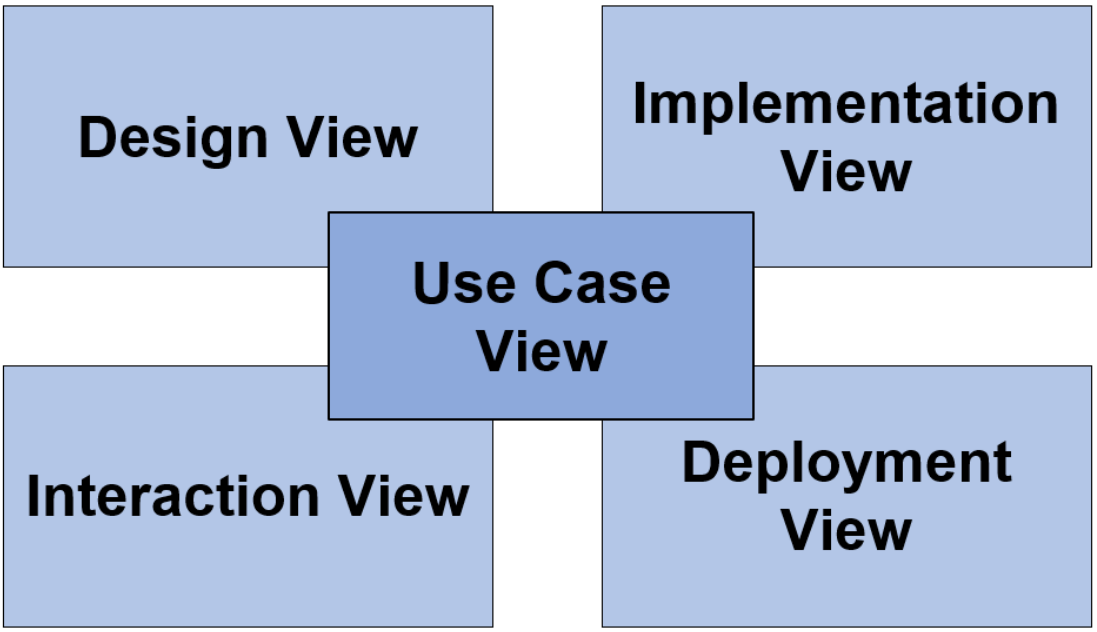
- **Building blocks** which make up the UML
- **Rules** that dictate how these building blocks can be put together
- **Common mechanisms** that apply through out UML

Things are abstractions in the model.

Relationship ties together the things or abstractions

Diagrams are graphical representation of set of elements, often rendered as a set of connected graph of vertices (things) and arcs (relationships)

Views of a Solution System



Use Case View

Describes behavior of system as seen by end users, analysts, and testers

Design View

Describes Classes, Interfaces, and Collaborations that form the vocabulary of the problem and its solution

Interaction View

Describes flow of control among its various parts Address performance, scalability, and throughput

State Diagram

Shows states, transitions, events, and activities depicting the dynamic view of internal object states

Sequence Diagram

Shows interactions between objects as a time-ordered view

Collaboration Diagram

Similar to sequence diagram, but in a spatial view, using numbering to indicate time order

Deployment Diagram

Shows the configuration of run-time processing nodes and the components that are deployed on them

Use case Modelling: Use case Diagrams

- Describes the **interaction of users and the system**
- Describes **what functionality does a system provides to its users**

Actor/s: One or set of objects who directly interacts with the system

- Every actor has a defined purpose while interacting with the system.
- An actor can be a person, device or another system.

Use case: A piece of functionality that a system offers to its users.

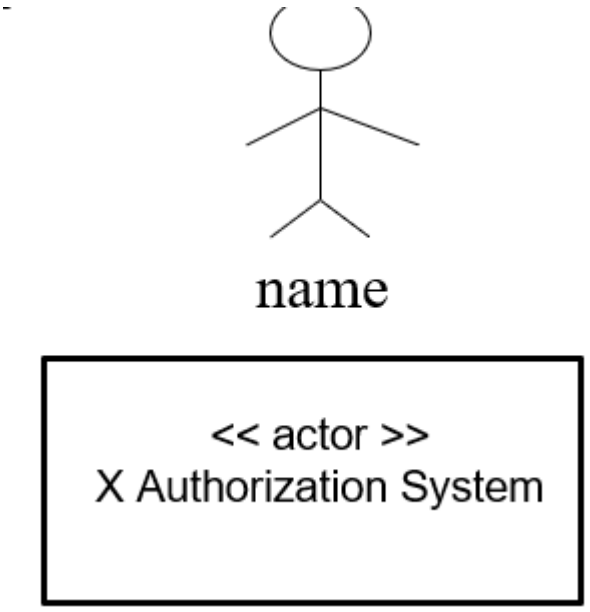
- Set of all use cases defines the entire functionality of the system.
- Also define the error conditions that may occur while interacting with the system

Rules in Use Case

1. Use cases describe interactions between actors and the system to achieve specific outcomes.
2. They consist of multiple sequences of actions representing different scenarios.
3. Each use case results in an observable outcome valuable to the actors involved.

4. Variants accommodate different conditions or exceptions within the main sequence.
5. Use case names start with a verb to denote the action or goal.
6. Sequences within a use case depict interactions initiated by actors and responses from the system.
7. They specify desired system behavior under various conditions.
8. Use cases focus on delivering value to actors by fulfilling their needs or business objectives.

Actors

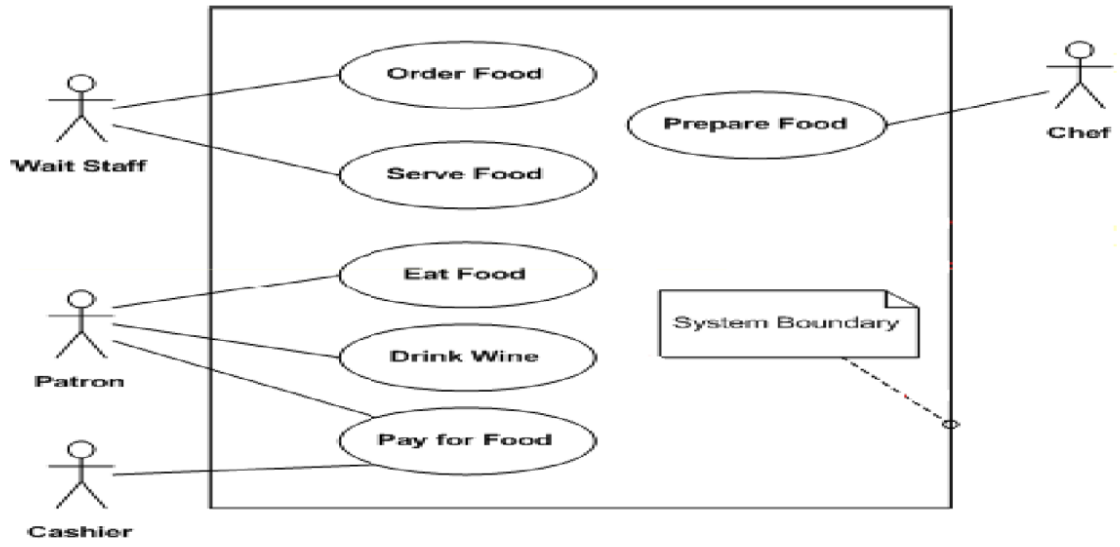


- Actors represent different user roles or system components interacting with use cases.
- They can be human users or automated processes/systems.
- Named using nouns to describe their role within the system.
- Actors either require assistance from the system or are necessary for executing system functions.
- They exist external to the system and trigger the execution of use cases.
- Actors have responsibilities towards the system, providing inputs, and expectations from it in terms of outputs or responses.

Components of a Use Case:

- Title or Reference Name
- Author/Date
- Modification/Date
- Purpose
- Overview
- Cross References
- Actors
- Pre Conditions
- Post Conditions
- Normal flow of events
- Alternative flow of events
- Exceptional flow of events
- Implementation issues

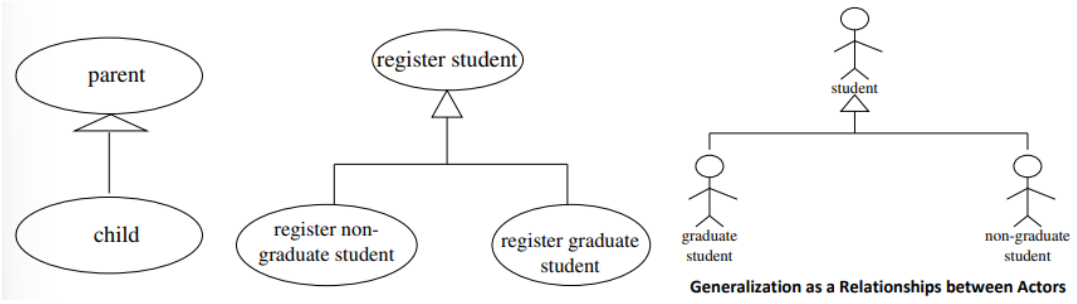
Restaurant Use Case



Relationships between Use Cases

Generalization:

- **Description**:
  - Use cases that represent specialized versions of other use cases, **inheriting** their behavior and meaning.
  - The child use case shares the same relationship with actors as the parent use case.
- **Example**:
  - Parent Use Case: "Manage Account"
  - Child Use Case: "Manage Savings Account"
  - Explanation:
    - The "Manage Savings Account" use case inherits the behavior and meaning of the "Manage Account" use case, as both involve managing accounts.
    - However, the "Manage Savings Account" use case may add specific behaviors related to managing savings features, such as interest calculation or deposit limits, while still maintaining the core functionality of managing accounts.

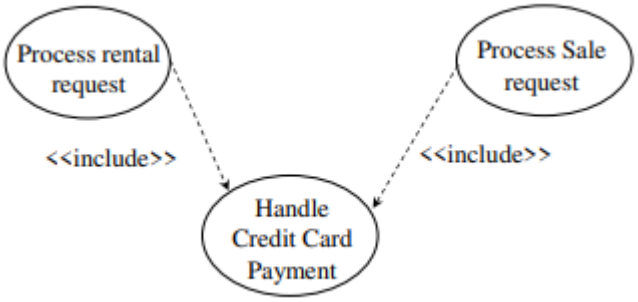


Generalization as a Relationships between Actors

Include:

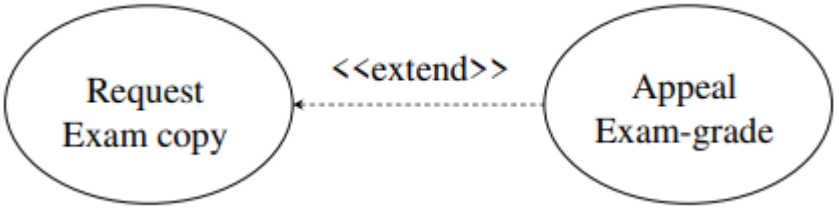
- **Description**:
  - Use cases that are included as parts of other use cases to factor common behavior.
  - The base use case explicitly incorporates the behavior of another use case at specified locations.
- **Example**:
  - Base Use Case: "Order Food Online"

- Included Use Case: "Select Payment Method"
- Explanation:
  - The "Select Payment Method" use case is included as part of the "Order Food Online" use case to handle the common behavior of selecting a payment method during the order process.
  - This enables the "Order Food Online" use case to focus on the overall ordering process without needing to describe the payment selection flow multiple times.



Extend:

- **Description**:
  - Use cases that extend the behavior of other core use cases to model optional behavior or branching under specific conditions.
  - The base use case may stand alone but can be extended by another use case at certain extension points.
- **Example**:
  - Base Use Case: "Book Flight"
  - Extending Use Case: "Upgrade Seat"
  - Explanation:
    - The "Upgrade Seat" use case extends the behavior of the "Book Flight" use case, allowing users to upgrade their seat selection after booking a flight.
    - The "Book Flight" use case can function independently, but when a user chooses to upgrade their seat, the behavior specified in the "Upgrade Seat" use case is invoked at the designated extension point.



Use Case Specification

1. **Name** (Must start with a verb)
2. **Summary**
3. **Actors**
4. **Pre-conditions** Conditions that must exist *before* the use case is executed
5. **Description** Textual description (may include steps to execute) and typically is the primary functionality
6. **Exceptions** These are paths which will need to handle exceptions which could be all to provide handling of things which are not provide you with a primary functionality including things like power failure
7. **Alternate Flows** Handles the other functionality paths for the summary these could be some in the exceptions too
8. **Post-conditions** Conditions that must exist *after* the use case is executed

Use Case Specification: Order Food Online

- **Summary:**
  - Allows users to order food online from a restaurant's menu.
- **Actors:**
  - Customer, Restaurant Staff.
- **Pre-conditions:**
  - The user must be logged into the online ordering platform.
  - The restaurant must have an active menu available for online ordering.
- **Description:**
  - The user selects desired items from the restaurant's menu.
  - The user adds selected items to the shopping cart.

- The user proceeds to checkout and provides delivery address and payment details.
- The order is confirmed, and the user receives an order confirmation.

- **Exceptions:**
  - Payment failure: If the payment transaction fails, the user is prompted to try again or choose an alternative payment method.
  - Menu items unavailable: If any selected items are unavailable, the user is notified and prompted to remove or replace them.
  - Network error: In case of network failure during checkout, the user is prompted to retry or check their internet connection.
- **Alternate Flows:**
  - Special Instructions: User may provide special instructions for food preparation or delivery.
  - Order Modification: User can modify the order (add/remove items) before final confirmation.
- **Post-conditions:**
  - The order is successfully placed in the restaurant's system.
  - The user receives an order confirmation with estimated delivery time.

Actor Actions	System Actions
1. Begins when a Customer arrives at ATM	
2. Customer inserts a Credit card into ATM	3. System verifies the customer ID and status
5. Customer chooses "Withdraw" operation	4. System asks for an operation type
7. Customer enters the cash amount	6. System asks for the withdrawal amount
	8. System checks if withdrawal amount is legal
	9. System dispenses the cash
	10. System deduces the withdraw amount from account
	11. System prints a receipt
13. Customer takes the cash and the receipt	12. System ejects the cash card

Alternative flow of events:

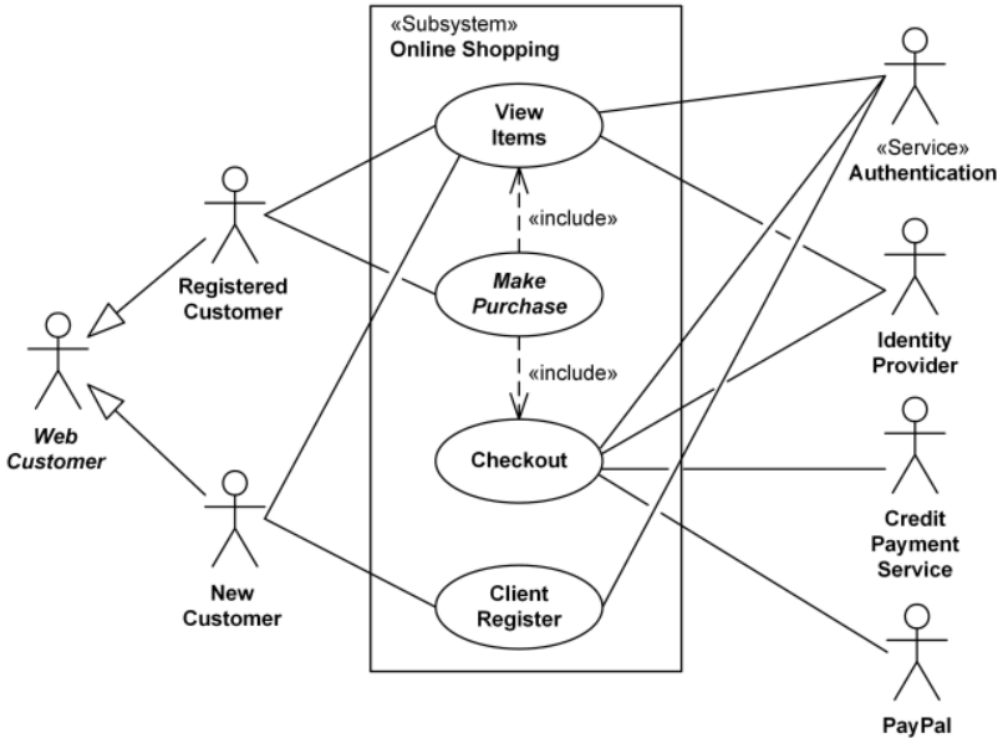
Step 3: Customer authorization failed. Display an error message, cancel the transaction and eject the card.

Step 8: Customer has insufficient funds in its account. Display an error message, and go to step 6.

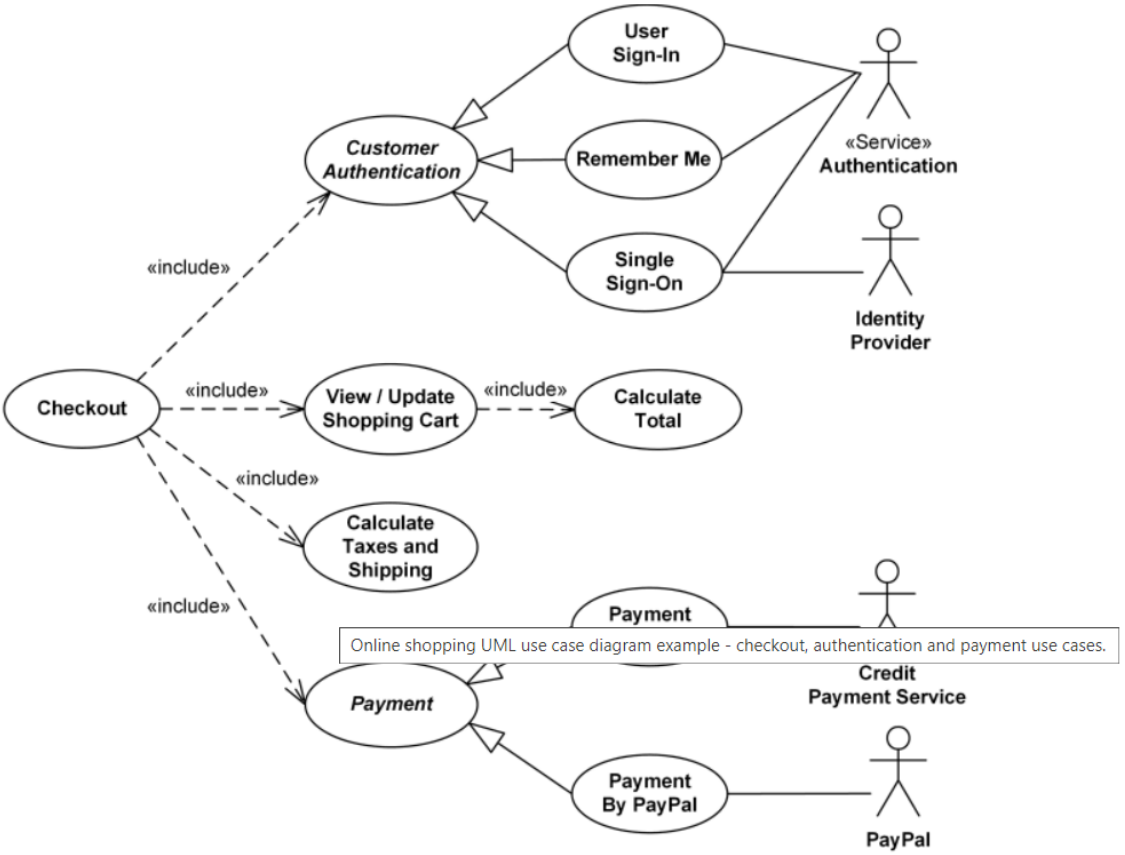
Step 8: Customer exceeds its legal amount. Display an error message, and go to step 6.

Exceptional flow of events:

Power failure in the process of the transaction before **step 9**, cancel the transaction and eject the card



Online shopping UML use case diagram example - view items use case.



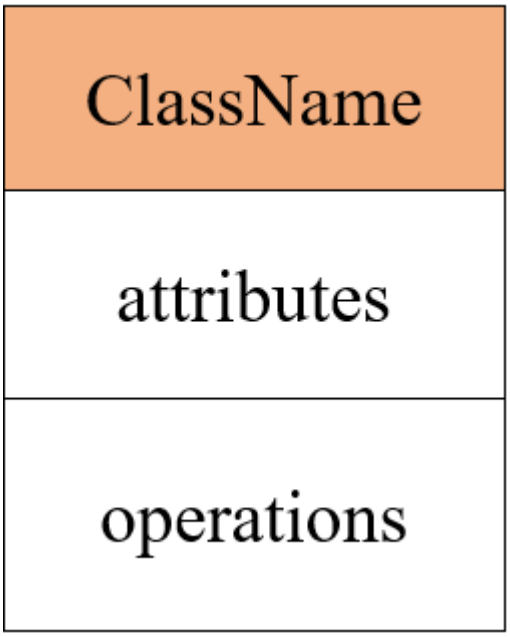
Class Modelling: UML Class Diagrams

- **Definition:**
  - Class diagrams capture the static structure of object-oriented systems, illustrating the objects, relationships, attributes, and operations of each class.
- **Visualization Tool:**
  - Class diagrams provide a graphical notation for constructing and visualizing object-oriented systems, aiding in understanding system structure.
- **Direct Mapping:**
  - They can be directly mapped with object-oriented languages, facilitating the implementation process.
- **Static Structure Focus:**
  - Class diagrams focus on the static structure of systems, depicting how objects are structured rather than their dynamic behavior.

- **Support for Architectural Design:**
  - They support architectural design by helping to organize and define the structure of a system.
- **Identification and Interaction:**
  - Class diagrams identify the classes present in the system, their interrelationships, and how they interact with each other.
- **Components:**
  - A UML class diagram consists of a set of classes and a set of relationships between classes, defining the structure and connections within the system.

Class

- A class is a description of a set of objects that share the same attributes, operations, relationships and semantics.
- The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.

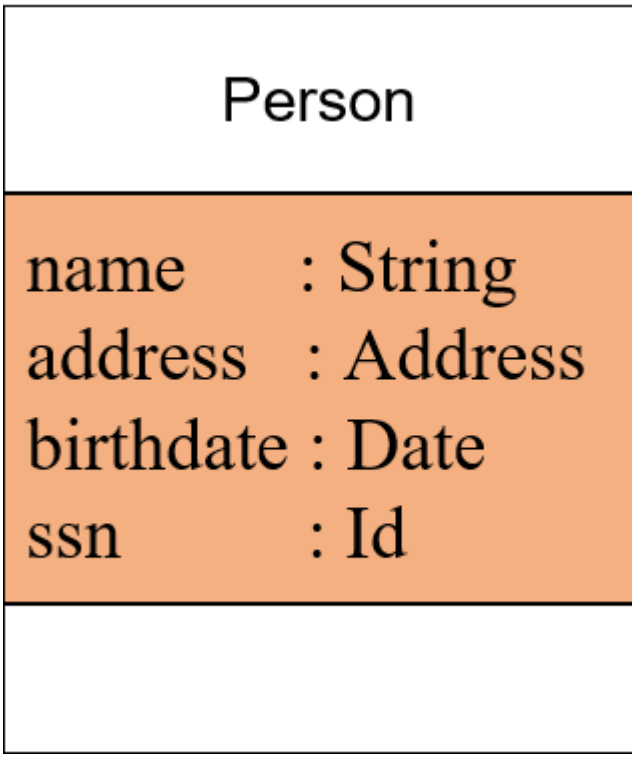


Attributes

Each class can have attributes.

An *attribute* is a named property of a class that describes the object being modeled.

Each attribute has a type.

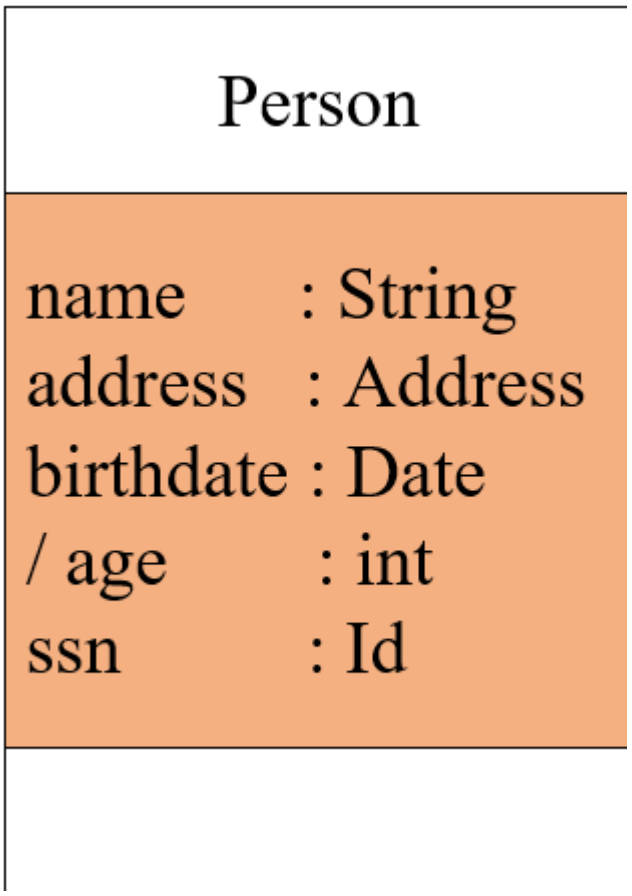


Derived Attribute

A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist.

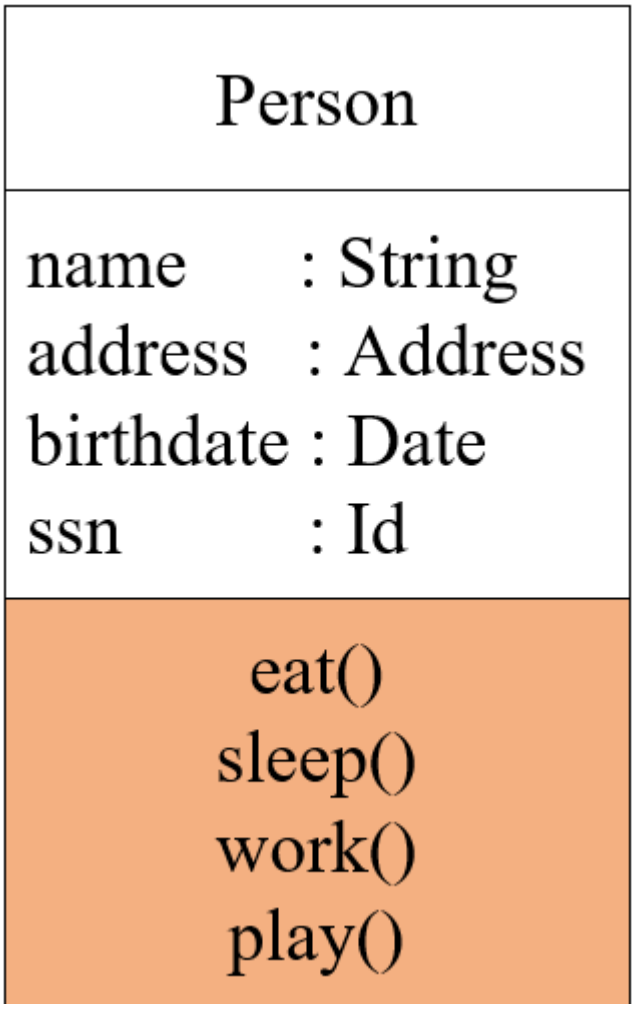
For example, a Person's age can be computed from his birth date.

A derived attribute is designated by a preceding '/' as in: / age : int

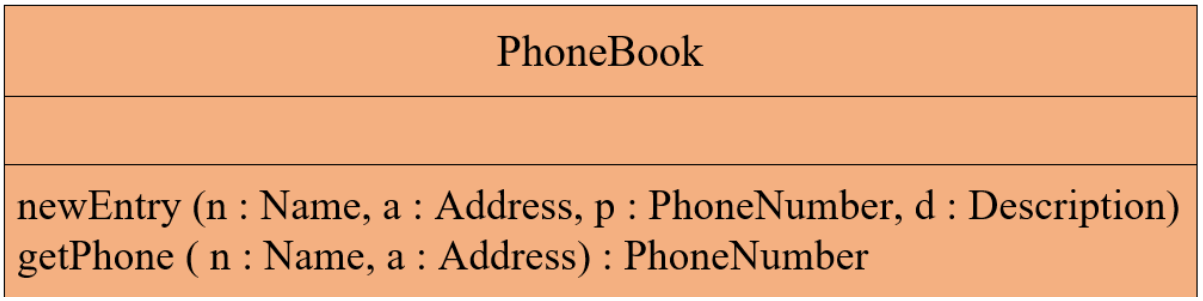


Class Operations

Operations describe the class behavior and appear in the third compartment.



You can specify an operation by stating its signature: listing the name, type, and default value of all parameters, and, in the case of functions, a return type.

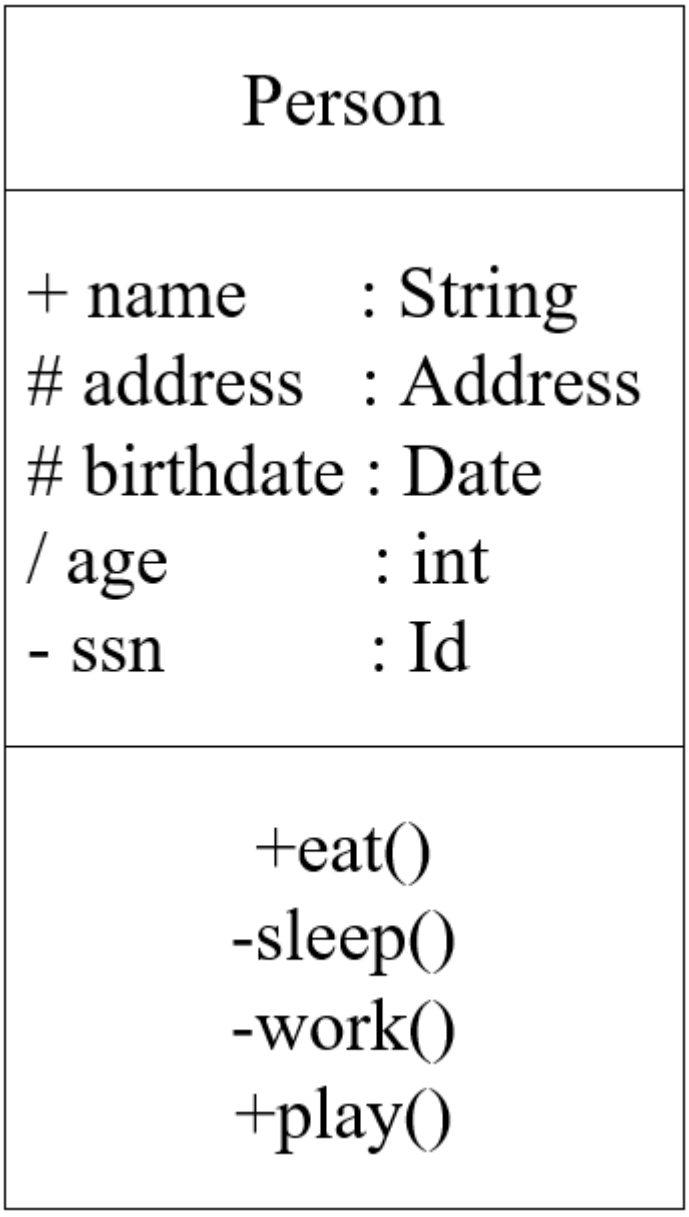


Visibility

attributes and operations can be declared with different visibility modes:

- public: any class can use the feature (attribute or operation)

- protected: any descendant of the class can use the feature
- private: only the class itself can use the feature.

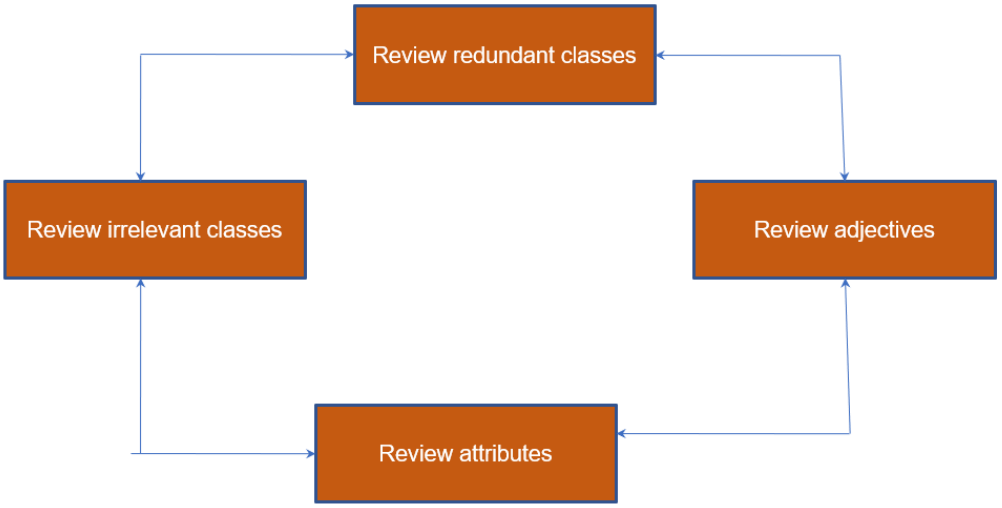


1. **Noun Phrase Approach**:
    - Identify nouns and noun phrases in the requirement document.
    - Consider nouns as potential classes and verbs as methods of the classes.
    - Change plurals to singular forms.

- Categorize nouns into relevant, fuzzy, and irrelevant classes.
- Eliminate irrelevant classes.
- Select candidate classes from relevant and fuzzy categories.
- Choose class names carefully.
- Incrementally and iteratively refine the list of classes.
- Select classes based on guidelines:
    - Redundant classes: Choose one class if multiple describe the same information.
    - Attribute classes: Define as attributes if used only as values.
    - Irrelevant classes: Ensure classes have clear purposes.
- The process is iterative and not sequential.

2. **Common Class Patterns Approach**:
    - Identify common patterns in classes across domains or systems.
    - Use established patterns to identify classes in the current system.
    - Patterns may include design patterns, architectural patterns, or domain-specific patterns.
    - Apply patterns to find relevant classes efficiently.
3. **Classes, Responsibilities, and Collaborators (CRC) Approach**:
    - Identify classes based on their responsibilities and collaborations.
    - Create CRC cards for each class to define its responsibilities and interactions with other classes.
    - Collaboratively refine and iterate on CRC cards to identify classes effectively.
4. **Use Case Driven Approach**:
    - Extract classes from use case descriptions.
    - Identify actors, actions, and objects involved in each use case.
    - Map these elements to potential classes in the system.
    - Ensure classes align with the functional requirements specified in the use cases.

- Iterate on the process based on feedback and further analysis.



Common Class Patterns Approach:

1. **Concept Class**:
   - Represents principles or abstract concepts used to organize or track business activities.
   - Examples: Performance, Quality.
2. **Event Class**:
   - Represents points in time that must be recorded, often associated with attributes such as who, what, when, where, how, or why.
   - Examples: Landing, Interrupt.
3. **Organization Class**:
   - Represents collections of people, resources, facilities, or groups to which users belong.
   - Examples: Accounting Department, Project Team.
4. **People Class**:
   - Specifies the different roles users play in interacting with the application.
   - Divided into two types:

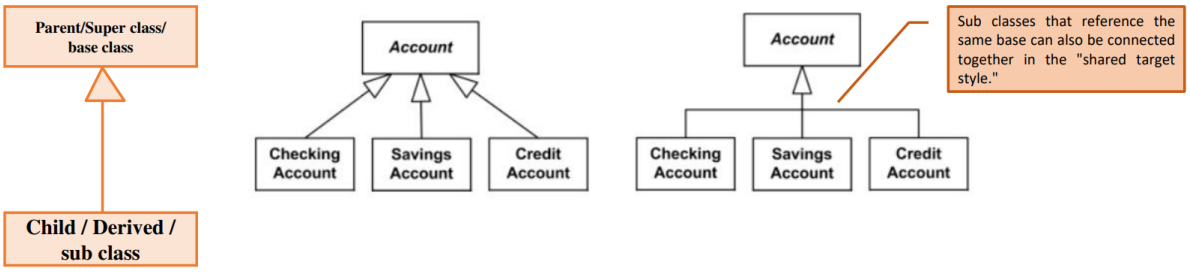
- Users of the system (e.g., operators, clerks).
- Individuals who do not use the system but whose information is kept by the system (e.g., customers, suppliers).

5. **Place Class**:
   - Represents physical locations that the system must keep information about.
   - Examples: Buildings, Stores, Offices.
6. **Tangible Things and Device Class**:
   - Represents physical objects or groups of objects that are tangible, including devices with which the application interacts.
   - Examples: Cars, Pressure Sensors.

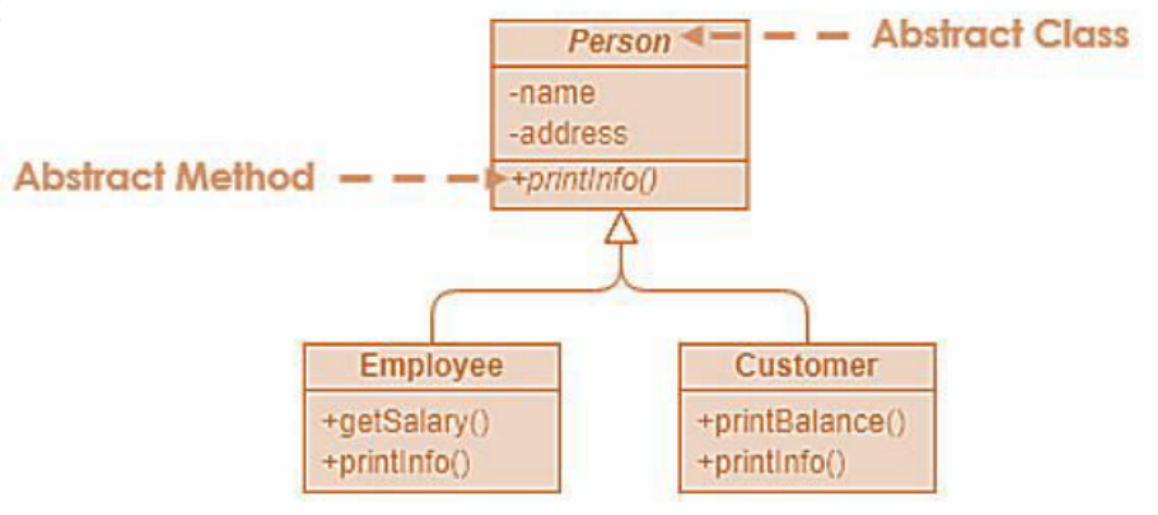
OO-Relationships

Generalization (Inheritence)

- Also known as Inheritance.
- It represents “is a” or “is a kind of” relationship since the child class is a type of the parent class.
- It is used to showcase reusable elements in the class diagram.
- The child classes “inherit” the common functionality(attributes and methods) defined in the parent class.
- A generalization is shown as a line with a hollow triangle as an arrowhead.
- The arrowhead points to the super class



Abstraction

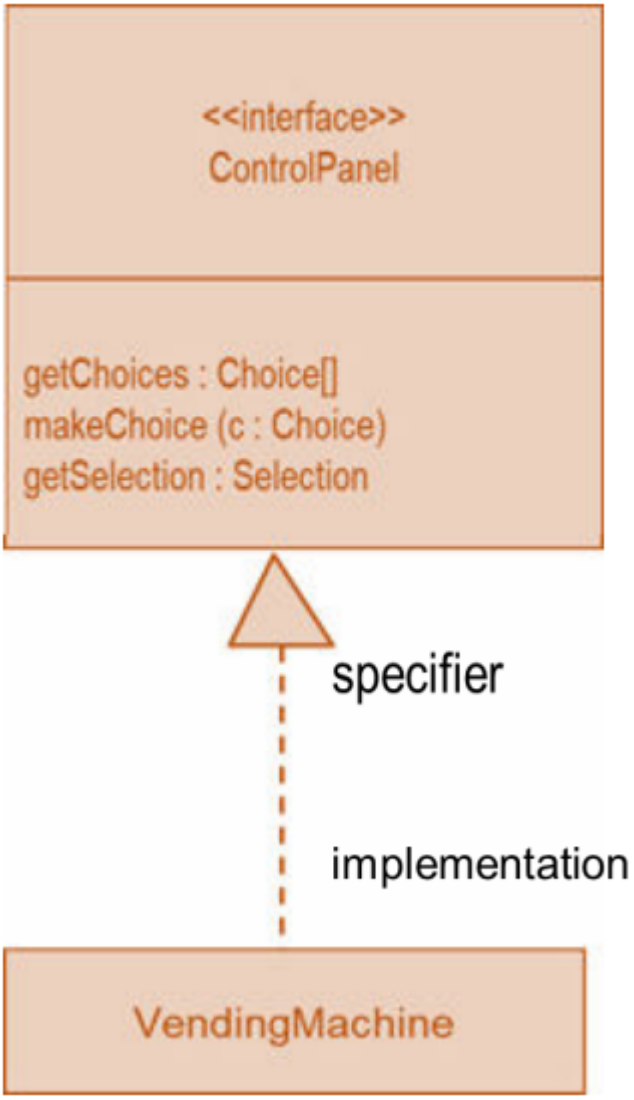


1. **Definition**: Abstraction involves defining the essential characteristics and behaviors of an object while concealing its implementation details.
2. **Abstract Classes**:
   - Abstract classes cannot be instantiated on their own.
   - They serve as blueprints for other classes.
   - Abstract classes may contain one or more abstract methods, which are declared but not implemented.
   - Abstract classes define a common interface and behavior for subclasses.
3. **Purpose**:
   - Encapsulates common functionality and behavior.
   - Provides a template for subclasses to follow.
   - Ensures consistency and interoperability within an inheritance hierarchy.
4. **Visual Distinction**:
   - Abstract classes are typically denoted by italicizing their names or using a textual annotation like {abstract}.

- This helps developers easily identify abstract classes within the codebase.

5. **Abstract Methods**:
    - Abstract methods do not have an implementation in the abstract class.
    - They are meant to be implemented by subclasses.
    - Abstract methods establish a contract that subclasses must fulfill.
6. **Creating Abstract Methods**:
    - Declare a method within the abstract class without providing an implementation.
    - In some programming languages, abstract methods are denoted by italicizing their names.
7. **Benefits**:
    - Promotes code reusability by defining common behaviors.
    - Enhances maintainability by encapsulating implementation details.
    - Supports extensibility by allowing subclasses to provide specific implementations.

Realization / Interfaces



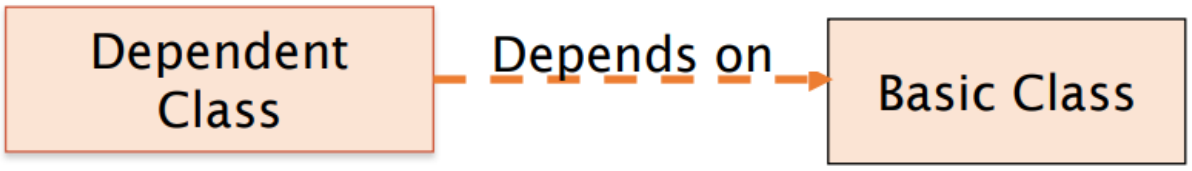
1. **Definition**: Realization is a specialized abstraction relationship between two entities, typically an interface and a class. It defines a contract specified by an interface that another entity (class) guarantees to fulfill by implementing the operations defined in that contract.
2. **Interface and Class Relationship**:
    - An interface defines a set of functionalities as a contract.
    - The class "realizes" this contract by implementing the functionality defined in the interface.
3. **Visual Representation**:
    - Realization is depicted as a dashed directed line with an open arrowhead pointing to the interface from the implementing class.

- This graphical representation signifies that the class realizes the contract specified by the interface.

4. **Purpose**:
   - Encourages loose coupling between components by defining contracts.
   - Facilitates polymorphism, allowing different classes to be substituted where the interface is expected.
   - Enables code reuse and flexibility by separating interface definition from implementation details.
5. **Benefits**:
   - Promotes modular design and maintainability by clearly defining contracts.
   - Enhances code readability and understandability by explicitly showing relationships between entities.
   - Facilitates robustness and scalability by allowing for interchangeable implementations.
6. **Implementation**:
   - The class implementing an interface must provide concrete implementations for all the methods specified in the interface.
   - This ensures that the contract defined by the interface is fulfilled by the implementing class.

Dependency

- Dependency means “One class uses the other”
- A dependency relationship indicates that a change in one class may affect the dependent class, but not necessarily the reverse.



Association Relationships

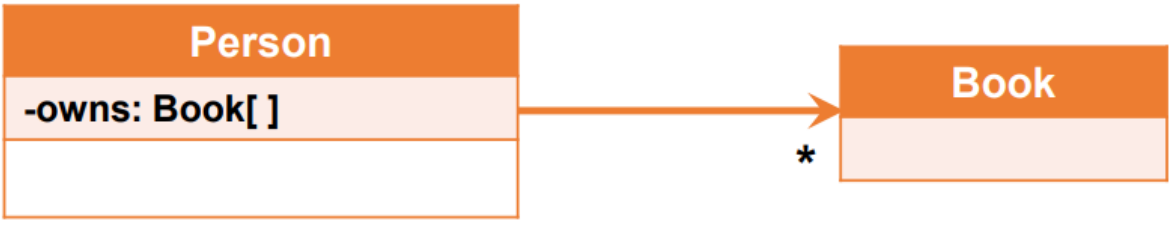
If two classes in a model need to communicate with each other, there must be link between
them. An association denotes that link.

Association is of two types.

- Unidirectional Association
- Bidirectional Association

Unidirectional Association:

- In a unidirectional association, two classes are related, but only one class "knows" about the relationship.
- Represented as a solid line with an open arrowhead pointing towards the known class.
- The known class includes a role name and multiplicity value.
- Example: `OrderDetails` class knows about its relationship with the `Item` class, but `Item` class doesn't have knowledge of the association.



An object might store another object in a field

Multiplicity Adornments:

- Multiplicity indicates the number of possible instances of one class associated with a single instance of the other end.

- Represented as single numbers or ranges on the association line.

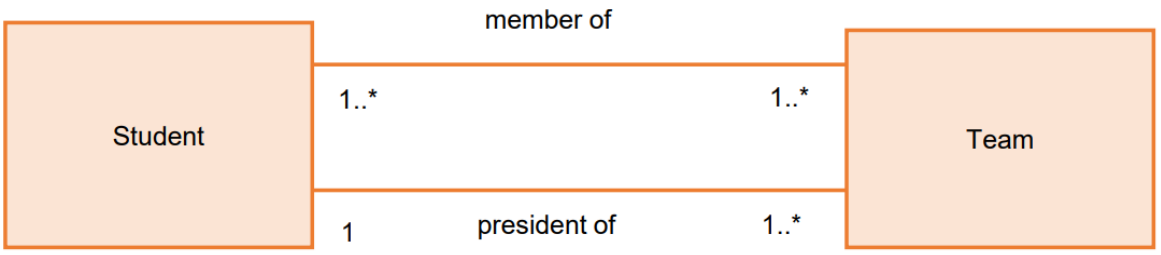
Multiplicities	Meaning
0..1	zero or one instance.
0..* *or* *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance
n..m	***n*** to ***m*** instances (n and m stand for numbers, e.g. **0..4**, **3..15**)
n	exactly n instance (where n stands for a number, e.g. **3**)

The example indicates that a *Student* has one or more *Instructors*:

The example indicates that every *Instructor* has one or more *Students*:

Bidirectional Association:

- Indicates associations where both classes involved have knowledge of the relationship.
- Represented with lines connecting both classes, often with role names and multiplicity adornments on both ends.
- Example: `Student` is a member of a `Team`, and a `Team` has members who are `Student`s.



Role Names:

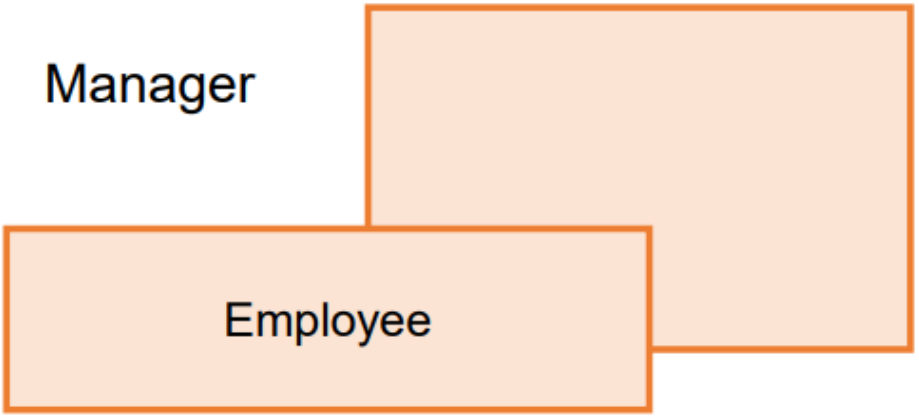
- Indicate the behavior or role of an object within the association.
- Used to clarify the nature of the relationship between classes.
- Example: A `Student` "learns from" an `Instructor`.

Association Naming:

- Associations can be given names to provide additional context or clarity.
- Example: `Student` has a "membership" with a `Team`.

Self Association:

- A class can have an association with itself.
- Useful for modeling relationships within a single class.
- Example: `Employee` can have a relationship with another `Employee` if one is a `Manager`.



Aggregation

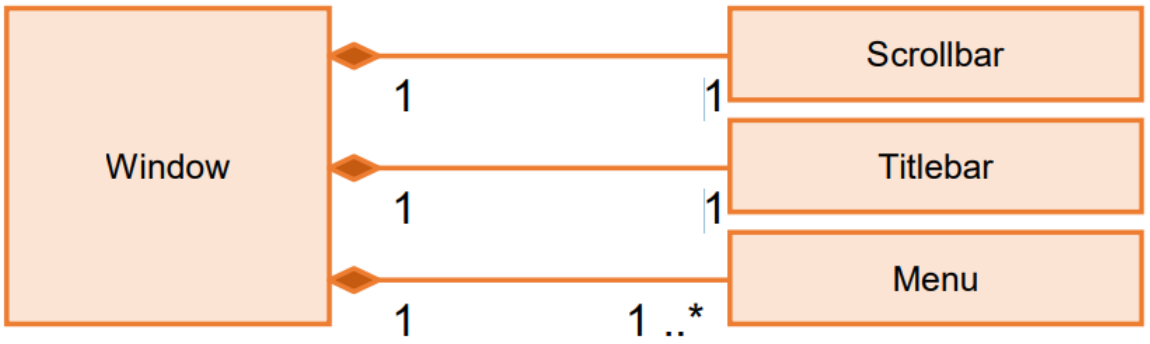
1. **Definition**: Aggregation is a modeling concept used to represent a "has a" relationship between objects, where one object contains or owns another object. It signifies a whole-part relationship where the part (constituent) can exist independently of the whole (aggregate).

1. **Representation**:
    - Aggregations are denoted by a hollow-diamond adornment on the association line between classes.
    - This visual indicator signifies that the aggregate class contains or owns instances of the constituent class.
2. **Example**:
    - Consider the relationship between a `Car`, `Engine`, and `Transmission`.
    - A `Car` aggregates `Engine` and `Transmission` objects.
    - While the `Engine` and `Transmission` are part of the `Car`, they can also exist independently, for example, during maintenance or replacement.
3. **Ownership**:
    - Aggregation implies ownership or containment, where one object (aggregate) possesses or controls another object (constituent).
    - For instance, a `Person` may own multiple `Book` objects.
    - While a `Book` is not physically part of the `Person`, it is considered part of the person's property.
4. **Multiplicity**:
    - Multiplicity indicates the number of instances of one class associated with a single instance of the other class.
    - In the example, each `Person` typically owns multiple `Book` objects (`1 *`).

Composition

1. **Definition**: Composition represents a strong ownership relationship between a whole and its parts, where the parts are integral to the existence of the whole. Unlike aggregation, compositions signify that the lifetime of the parts is directly tied to the lifetime of the whole.
2. **Representation**:
    - Compositions are denoted by a filled-diamond adornment on the association line between classes.
    - This visual indicator signifies that the whole class has exclusive ownership and control over the parts.
3. **Example**:

- Consider the relationship between a `Window` and its components such as `Scrollbar`, `Titlebar`, and `Menu`.
- These components are integral to the existence of the `Window`.
- The lifetime of the `Scrollbar`, `Titlebar`, and `Menu` instances is directly tied to the lifetime of the `Window`.



4. **Strong Ownership and Lifetime**:
    - Composition implies strong ownership and coincident lifetime of parts by the whole.
    - Parts created within a composition cannot exist independently of the whole and are typically destroyed when the whole is destroyed.
    - For example, if the `Window` is closed or destroyed, its components (Scrollbar, Titlebar, Menu) are also typically destroyed.
5. **Multiplicity**:

- Multiplicity indicates the number of instances of one class associated with a single instance of the other class.
- In the example, each `Window` typically has one instance of each component (`1`), implying a one-to-one relationship.

Feature	Composition	Aggregation
Ownership	Strong ownership: Parts are owned exclusively by the whole.	Weak ownership: Parts are independent entities and can be shared among multiple wholes.
Lifetime	Coincident lifetime: Parts' lifetimes are tied to the whole.	Independent lifetime: Parts can exist independently of the whole.
Dependency	Parts are integral to the existence of the whole.	Parts are loosely coupled with the whole.
Representation	Denoted by a filled-diamond adornment on the association line.	Denoted by a hollow-diamond adornment on the association line.
Example	A car and its engine; when the car is destroyed, the engine is also destroyed.	A library and its books; books can be shared among different libraries.
Multiplicity	Typically one-to-one relationship.	Multiplicity can vary; typically one-to-many or many-to-many relationship.

Enumeration

1. **Definition**: An enumeration, often denoted by the <<enumeration>> stereotype, is a user-defined data type that represents a set of named constants. It consists of a name and an ordered list of enumeration literals.
2. **Representation**:
    - Enumerations are typically depicted using the <<enumeration>> stereotype, which is placed above the name of the enumeration.
    - The enumeration literals, which represent the named constants, are listed below the enumeration name.

- Each enumeration literal has an implicit ordinal value based on its position in the ordered list.

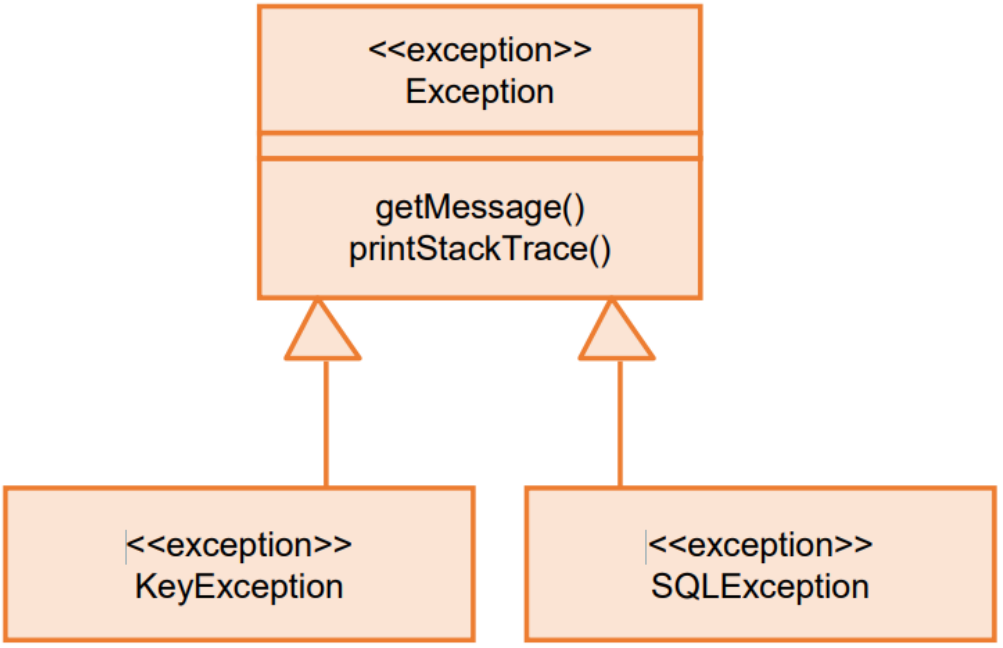
3. **Purpose**:
   - Enumerations are used to define a set of possible values for a variable.
   - They improve code readability and maintainability by providing meaningful names for constants.
   - Enumerations enhance type safety by restricting variable values to a predefined set of options.
4. **Example**:
   - In the provided example, the enumeration <<enumeration>> `Boolean` represents the Boolean data type with two literals: `false` and `true`.
   - This enumeration defines the only two possible values for variables of type `Boolean`.
5. **Use Cases**:
   - Enumerations are commonly used to represent categories, states, or options in software systems.
   - Examples include days of the week, months of the year, status codes, and menu options.
6. **Benefits**:
   - Enhances code clarity by replacing magic numbers or strings with descriptive names.
   - Facilitates code maintenance and refactoring by centralizing definitions of constants.
   - Improves type safety by ensuring that variables only hold valid values from the predefined set.

Exceptions

1. **Definition**: Exceptions, denoted by the <<exception>> stereotype, represent exceptional conditions or errors that occur during the

execution of a program. They are a mechanism for handling errors and abnormal situations in software.

2. **Representation**:
    - Exceptions are modeled similarly to other classes, with the <<exception>> stereotype placed in the name compartment to indicate that it represents an exception class.
    - Exception classes typically have meaningful names that describe the specific type of error or exceptional condition they represent.
3. **Example**:



- In the provided example, three exception classes are defined: `KeyException`, `SQLException`, and a generic `Exception`.
- Each exception class is tagged with the <<exception>> stereotype, indicating its role as an exception.

4. **Methods**:
    - Exception classes can have methods that provide information about the exception or aid in debugging.
    - Common methods include `getMessage()` to retrieve an error message associated with the exception and `printStackTrace()` to print the stack

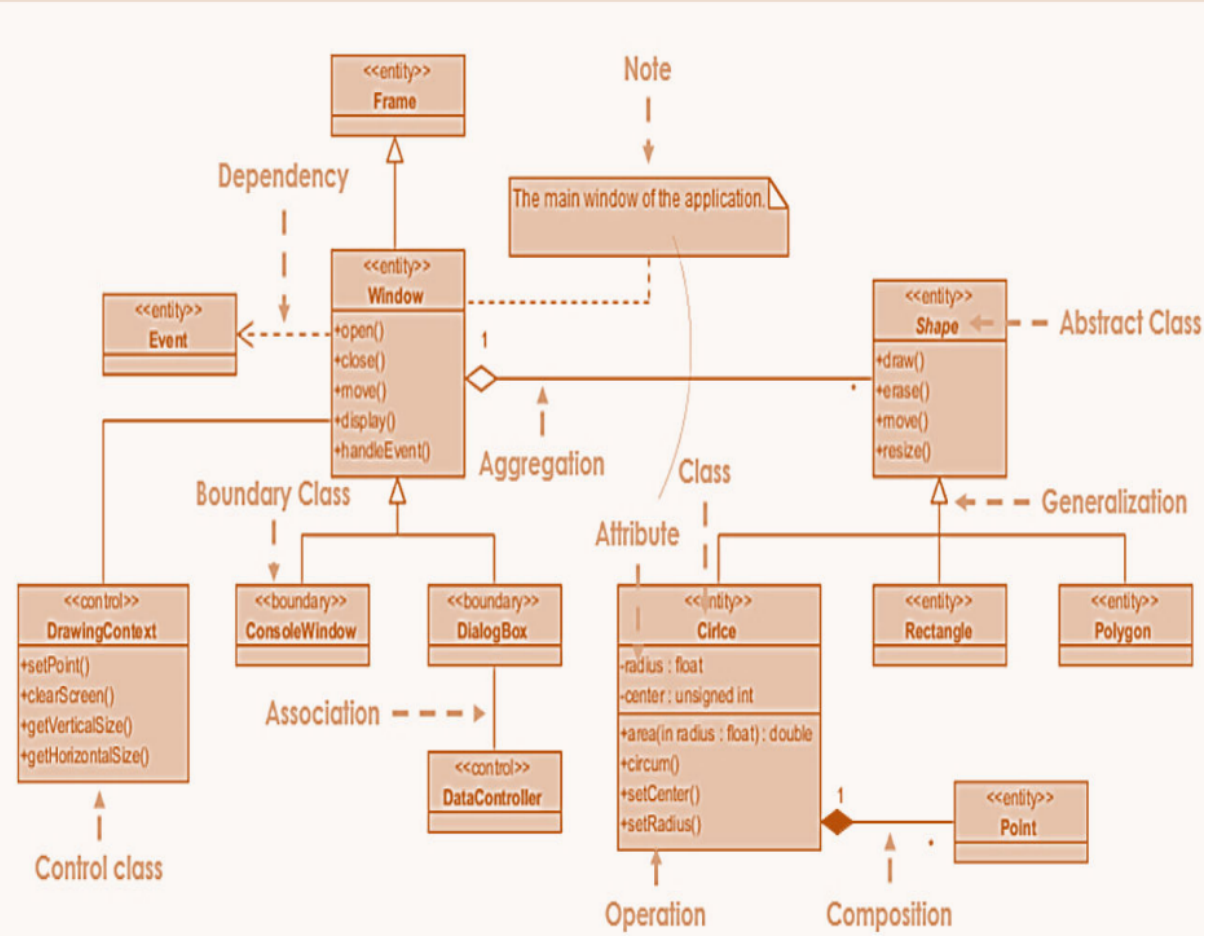
trace of the exception's occurrence.

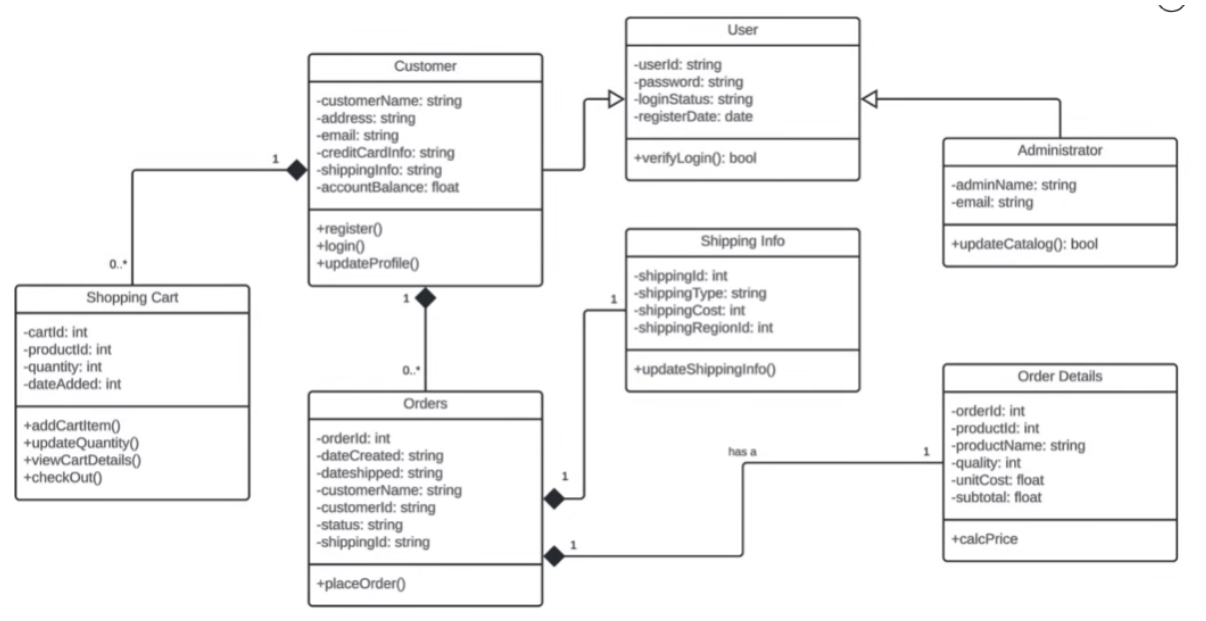
5. **Usage**:
    - Exceptions are thrown when an exceptional condition occurs during program execution.
    - They can be caught and handled by appropriate exception handling mechanisms to gracefully manage errors and maintain program stability.
6. **Best Practices**:
    - Define specific exception classes for different types of errors to provide more precise error handling.
    - Catch exceptions at appropriate points in the code and handle them gracefully to prevent program crashes and data corruption.

Question

1. **Shape**:
    - Abstract class, shown in italics.
    - Superclass.
    - Generalization/Inheritance relationship with Circle, Rectangle, and Polygon.
    - Attributes and method names are not specified.
2. **Circle**:
    - Derived from Shape.
    - Has attributes: radius and center.
    - Methods: area(), circum(), setCenter(), and setRadius().
    - Parameter radius is an in parameter of type float.
    - Method area() returns a value of type double.
    - Aggregation relationship with Window (part-of relationship).
3. **Rectangle**:
    - Derived from Shape.
    - Specific attributes and method names are hidden.

4. **Polygon**:
    - Derived from Shape.
    - Specific attributes and method names are hidden.
5. **DialogBox**:
    - Associated with DataController.
6. **DataController**:
    - Associated with DialogBox.
7. **Window**:
    - Aggregates Shape (part-of relationship).
    - Shape can exist without Window.
8. **Event**:
    - Window is dependent on Event, but Event is not dependent on Window.
9. **Point**:
    - Part-of relationship with Circle (Composition).
    - Point cannot exist without a Circle.

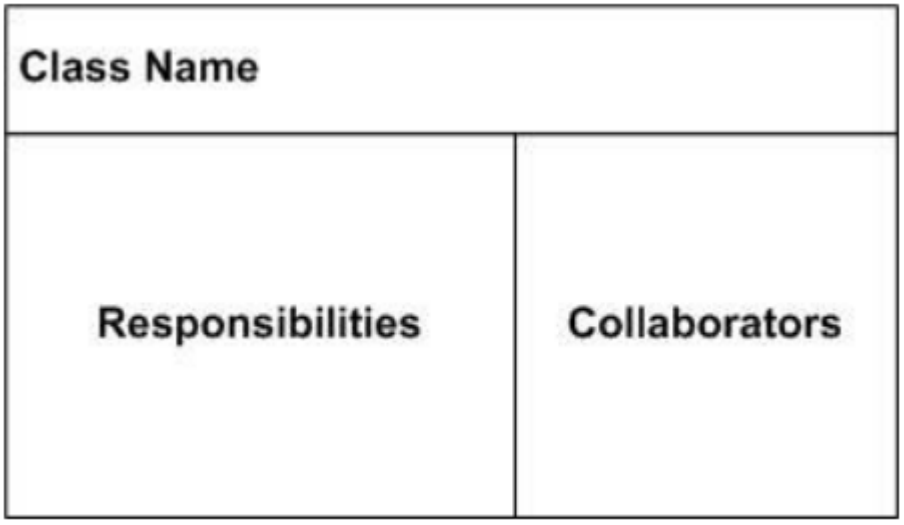




CRC (Class-Responsibility-Collaborators)

- CRC cards are tool used for brainstorming in OO design and agile methodologies

● CRC cards are created from Index cards
● Each member of the project team write one CRC card for each relevant class of their design
● Objects need to interact with other objects (Collaborators) in order to fulfill their responsibilities
● Since the cards are small, prevents to get into details and give too many responsibilities to a class
● They can be easily placed on the table and rearranged to describe and evolve the design



CRC Cards: Class, Responsibility, Collaboration

1. **Class**:
    - Object-oriented class name.
    - Includes information about super- and sub-classes.
2. **Responsibility**:
    - Stores information pertinent to its purpose.
    - Defines the actions and behaviors it can perform.
    - Specifies the behavior for which an object of this class is accountable.
3. **Collaboration**:
    - Describes relationships with other classes.
    - Indicates which other classes this class interacts with or uses in its operations.

Class Name	
Responsibilities	**Collaborators**
Class: Account	
Responsibilities	**Collaborators**
Know balance	
Deposit Funds	Transaction
Withdraw Funds	Transaction, Policy
Standing Instructions	Transaction, StandingInstruction Policy, Account

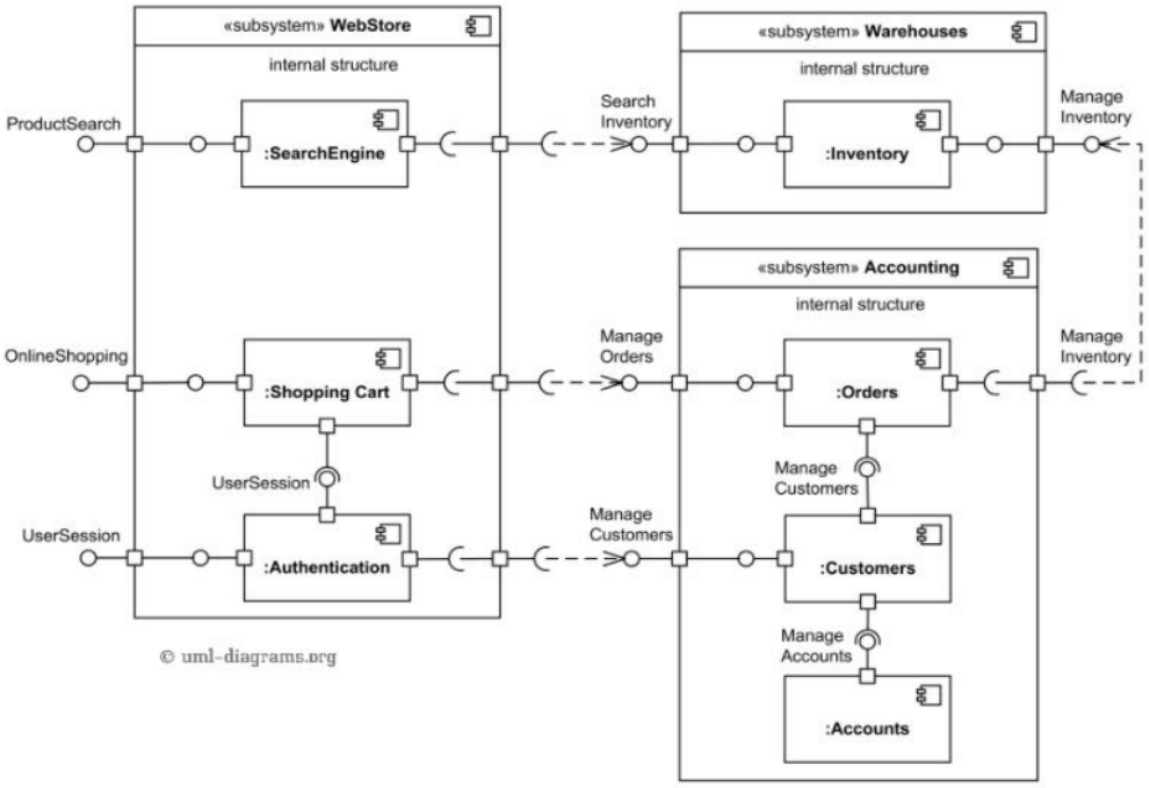
Ex: CRC card

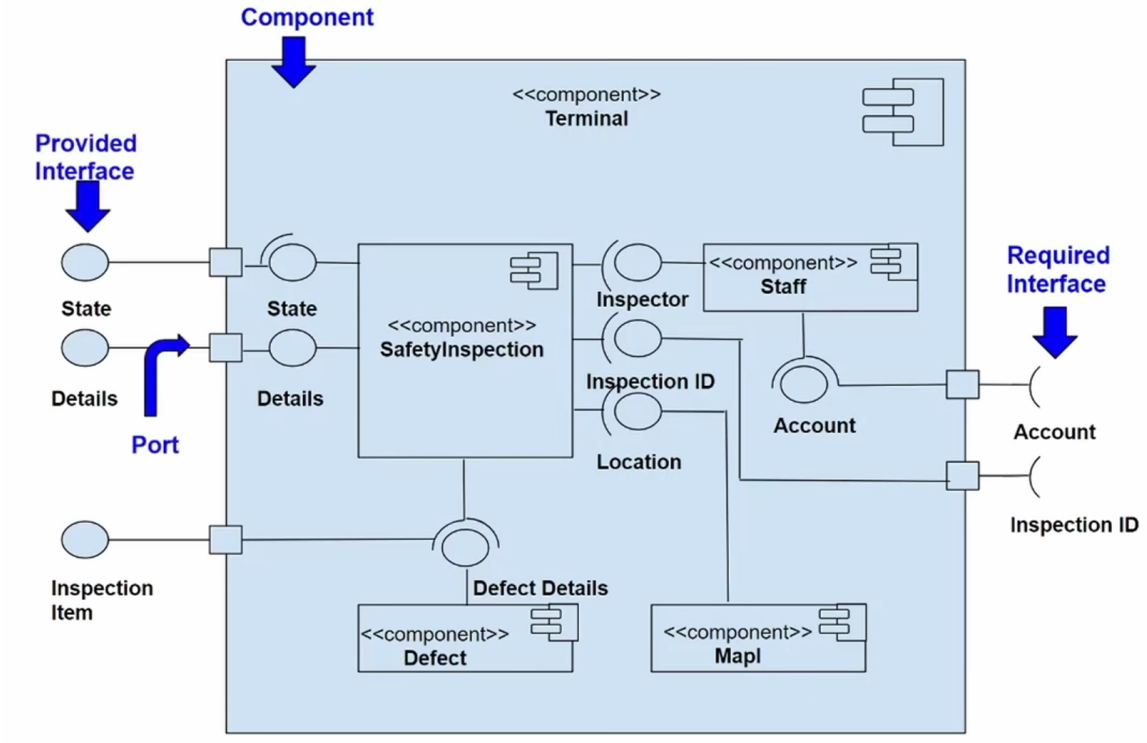
Component Model

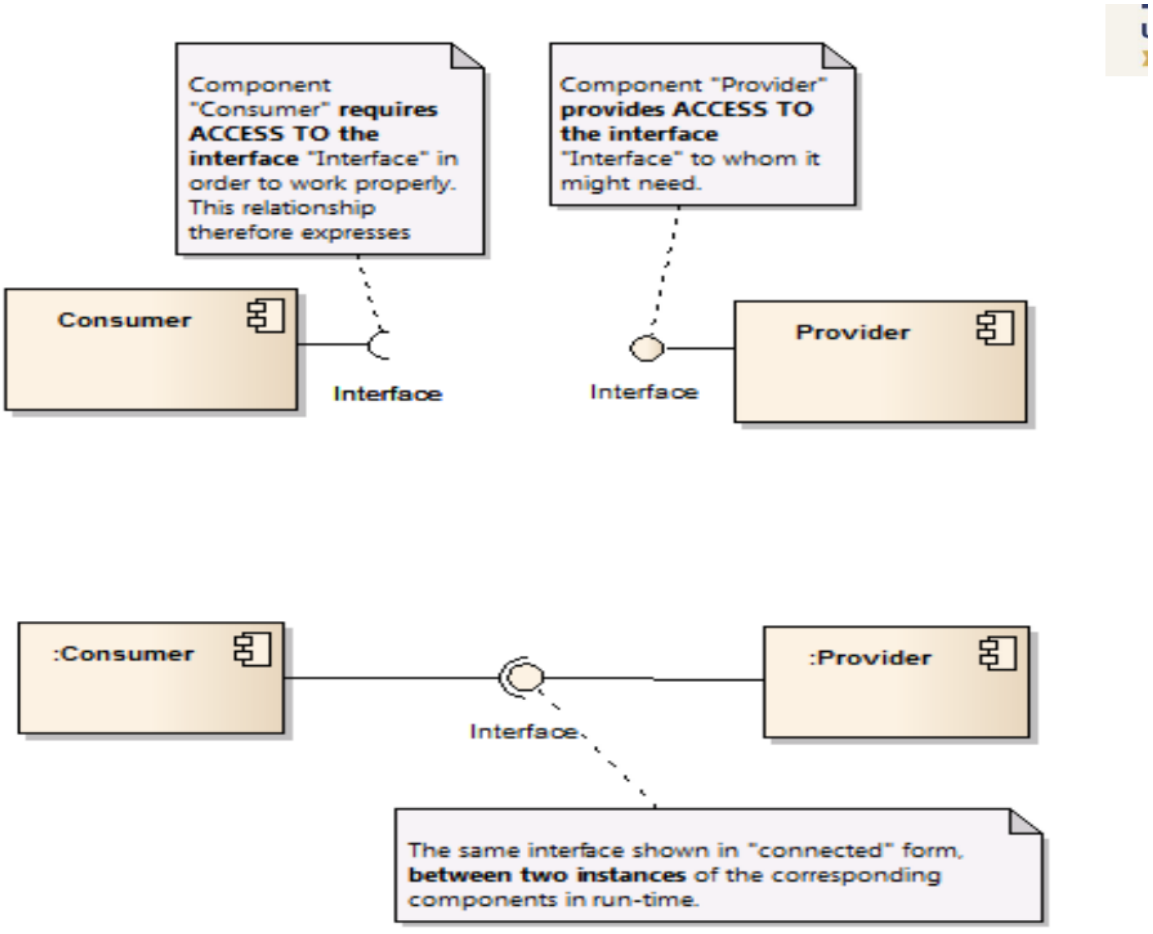
1. **Definition**:
   - A component is a self-contained, replaceable unit of a system.
   - It encapsulates implementation details, exposing only its interfaces.
   - It provides a set of interfaces that define how it can be accessed and utilized by other parts of the system.
2. **Modularity**:

- Components break down a large system into smaller, manageable subsystems.
- Each component represents a distinct piece of functionality within the system.
- Components allow for modular design and development, promoting ease of maintenance and scalability.

3. **Interfaces**:
   - Components require interfaces to interact with other components or external systems.
   - They realize provided interfaces, which define the functionality they offer to other components.
   - They may also rely on required interfaces, specifying the functionality they depend on from other components.
4. **Encapsulation**:
   - Components encapsulate their contents, hiding implementation details.
   - They expose only the necessary interfaces, abstracting away internal complexities.
5. **Interaction**:
   - Components interact with each other based on their defined interfaces.
   - Interactions occur on a need-to-know basis, with each component communicating with essential elements to fulfill its purpose.
6. **Component Diagram**:
   - A component diagram visually represents the components of a system and their relationships.
   - It breaks down the system into high levels of functionality, showing how components collaborate to achieve system goals.
   - Each component diagram depicts the system's architecture and helps in understanding the system's structure and dependencies.


«subsystem» WebStore
internal structure
:SearchEngine
:Shopping Cart
:Authentication
UserSession
UserSession
ProductSearch
OnlineShopping
Search Inventory
Manage Orders
Manage Customers
«subsystem» Warehouses
internal structure
:Inventory
Manage Inventory
«subsystem» Accounting
internal structure
:Orders
:Customers
:Accounts
Manage Customers
Manage Accounts
Manage Inventory
© uml-diagrams.org


Component
<<component>>
Terminal
Provided Interface
State
Details
Port
State
Details
<<component>>
SafetyInspection
Inspector
Inspection ID
Location
<<component>>
Staff
Account
Required Interface
Account
Inspection ID
Inspection Item
Defect Details
<<component>>
Defect
<<component>>
MapI

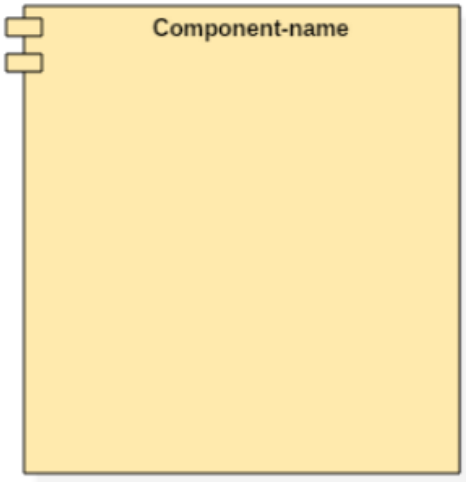


Nodes and Components:

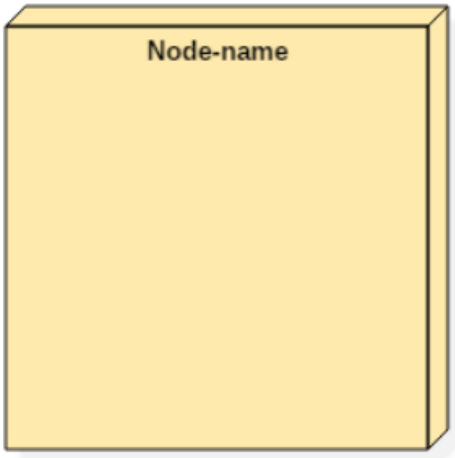
- **Components** are elements that participate in the execution of a system. They represent the physical packaging of logical elements. Components encapsulate functionality and are represented in UML with the <<component>> stereotype.
- **Nodes** are entities that execute components. They represent the physical deployment of components in a system. Nodes can be physical hardware devices, such as servers or computers, or software containers, such as virtual machines or Docker containers.

Component Diagram Notations

Component Notation in Component Diagram



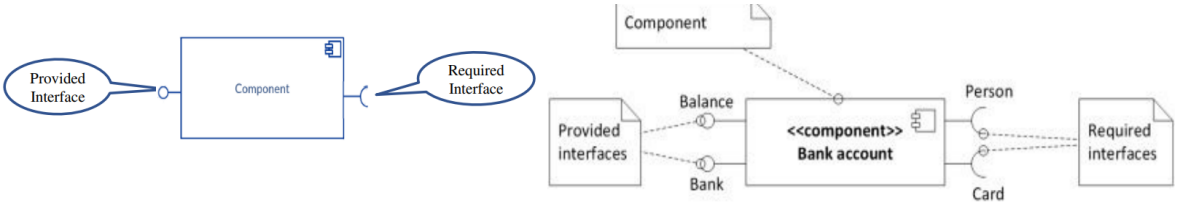
Node Notation in Component Diagram



What are nodes and components?
Components are things that participate in the execution of a system; nodes are things that execute components. Components represent the physical packaging of logical elements; nodes represent the physical deployment of components.

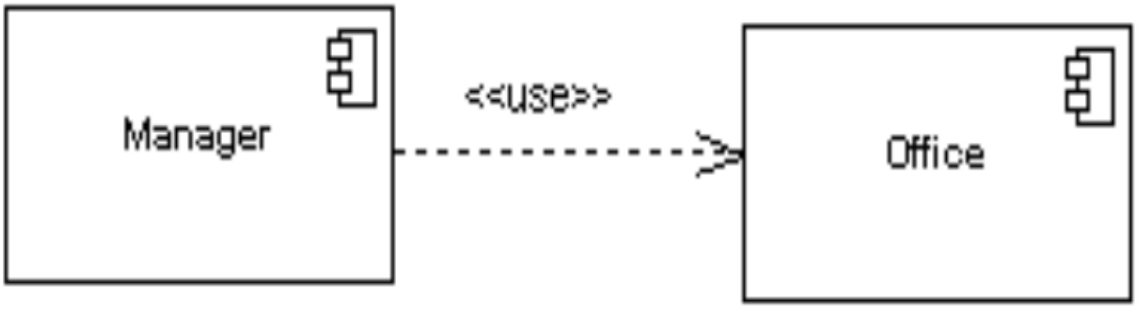
Interface in Component Diagram:

- An **interface** is a named set of public features that separate the specification of functionality from its implementation.
- It declares a contract that may be realized by zero or more classifiers, such as classes or subsystems.
- Interfaces cannot be instantiated and define a set of operations or methods that must be implemented by classes or subsystems that realize them.
- In UML, provided and required interfaces are used to specify the services a component offers and requires, respectively.



Component Diagram Usage Connector:

- Components can be connected by **usage dependencies**.
- A usage dependency indicates that one component requires another component for its full implementation.
- It is shown as a dashed arrow with the <<use>> keyword, pointing from the dependent component to the one it depends on.



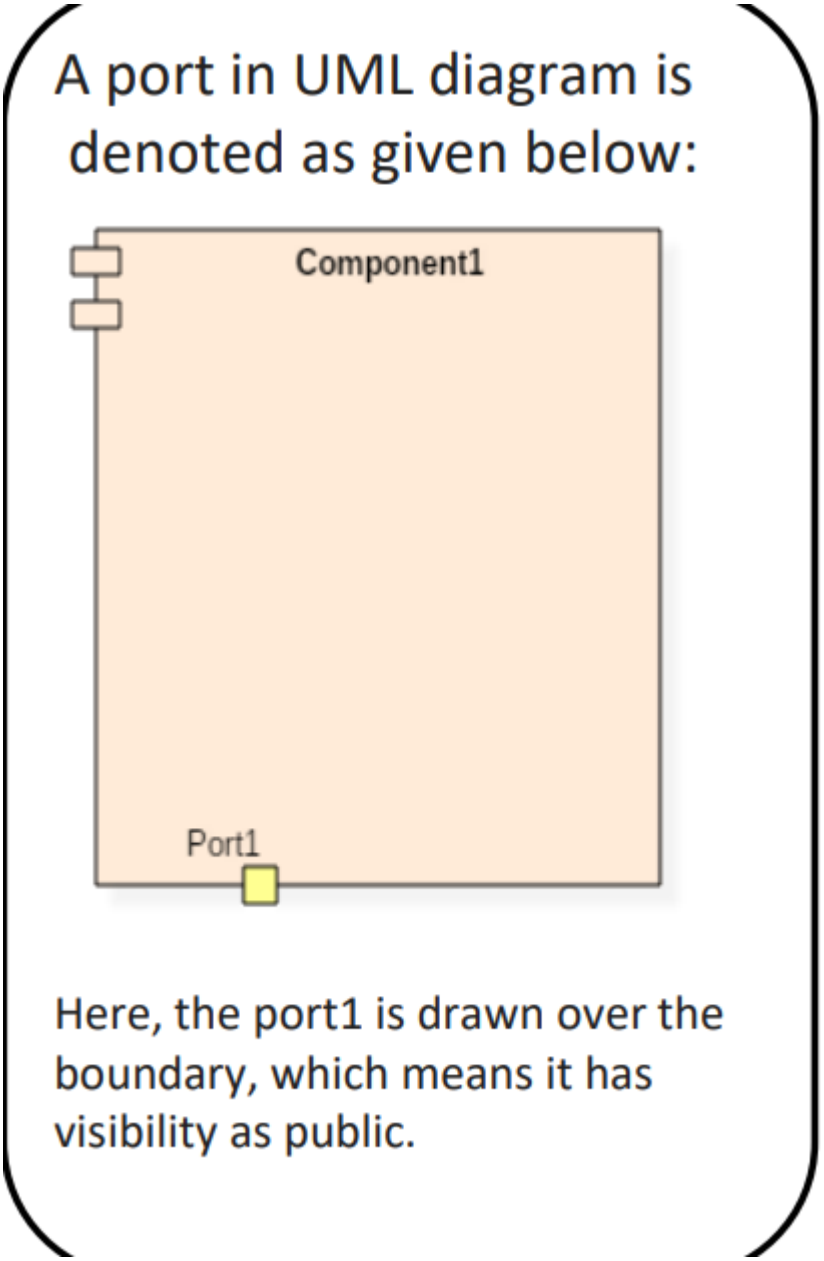
Subsystem:

- A **subsystem** is a logical construct used to break down a larger system into smaller, more manageable units.
- Subsystems act as decomposition units for larger systems, making it easier to manage each subsystem efficiently.
- Subsystems cannot be instantiated at runtime but can be initialized. They create a single system when connected.

Port:

- A **port** is an interaction point between a classifier (such as a component or subsystem) and its external environment.
- It groups a semantically cohesive set of provided and required interfaces.
- Ports can have visibility, such as public, protected, or private, indicating the accessibility of the interfaces they expose.

- Ports can also have multiplicity, indicating the number of instances of the port classifier.



To draw a Component Diagram, follow these steps:

1. **Identify Components**: Understand all the components within the system and their functionalities. List down each component and describe its purpose and behavior.
2. **Explore Components**: Dive deeper into each component to understand its connections with other physical artifacts in the system. Identify the

interfaces it provides and requires.

3. **Identify Relationships**: Determine the relationships among various artifacts, libraries, and files in the system. Understand how components interact with each other and with external systems.
4. **Create Diagram**: Use a UML modeling tool or a drawing tool to create the Component Diagram.
    - Place each component as a labeled box in the diagram.
    - Use the <<component>> stereotype to denote each component.
    - Draw lines between components to represent relationships, such as dependencies, associations, or usage.
5. **Label Diagram**: Label the diagram appropriately to indicate the purpose and context of the components and their relationships.
6. **Review and Refine**: Review the diagram to ensure it accurately represents the structure and organization of components in the system. Refine as needed to improve clarity and completeness.

Why use Component Diagrams?

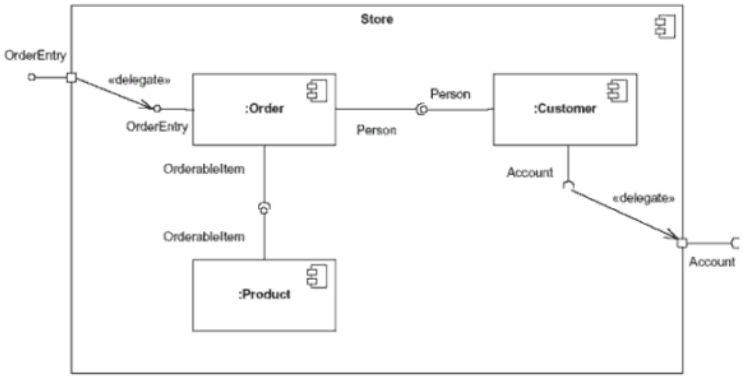
- **Representation at Runtime**: Component Diagrams represent the components of a system at runtime, helping in understanding the system's structure and behavior during execution.
- **Testing**: They aid in testing the system by visualizing the connection between various components, facilitating the identification of potential testing scenarios.
- **Visualization of Connections**: Component Diagrams visualize the connections between components, enabling stakeholders to understand how different parts of the system interact.

Internal and External Views of a Component:

- **External View**: Shows publicly visible properties and operations of a component. It represents the black box view, focusing on the component's interfaces and interactions with other components.
- **Internal View**: Shows the realizing classes/components nested within the component shape. It represents the white box view, providing insights into the internal structure and implementation details of the component.

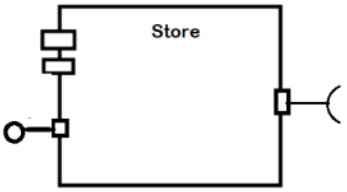
Internal and External view of a Component

Internal View



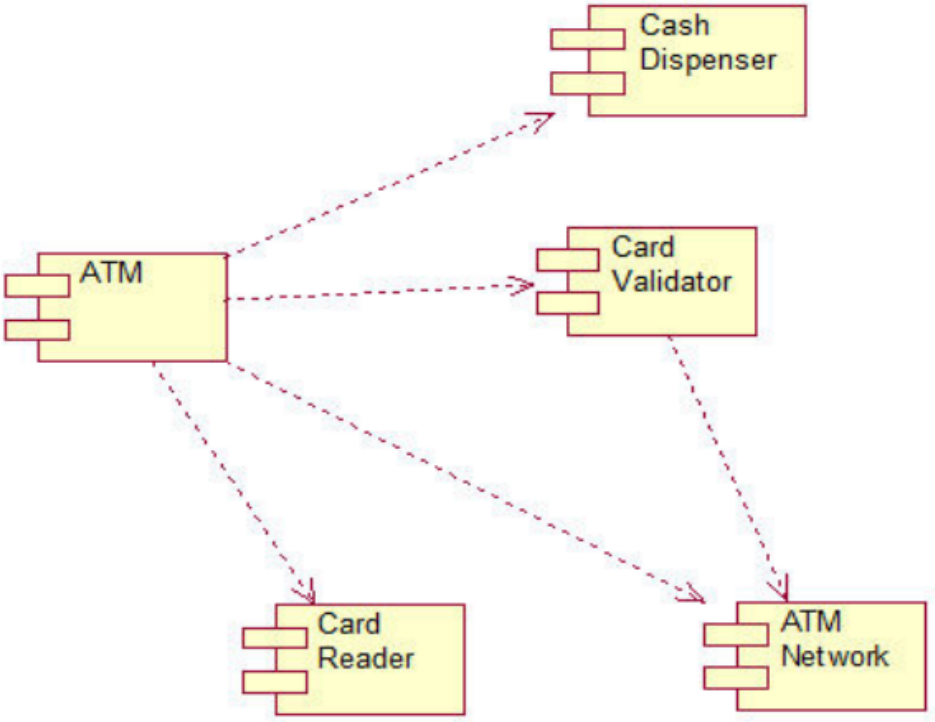
An internal view (or white box view) of a component is where the realizing classes/components are nested within the component shape

External View



An external view (or black box view) shows publicly visible properties and operations

Example – Component Diagram of an ATM



Deployment Model

Introduction to Deployment Diagrams:

Deployment diagrams are used to map the software architecture designed during development onto the physical system architecture where it will be executed. **They provide a visual representation of how software components are deployed on hardware or software nodes, especially in distributed systems where components are distributed across multiple physical nodes.**

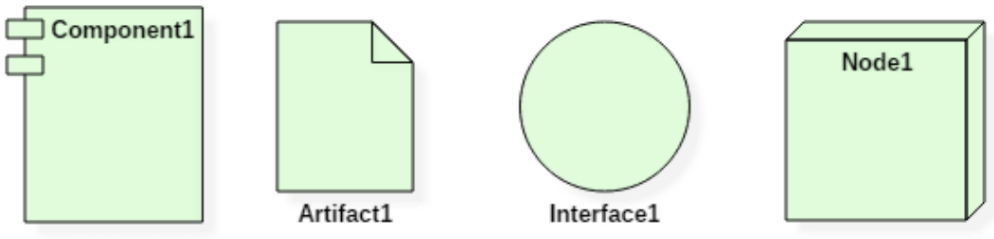
Purpose of Deployment Diagrams:

- **Visualizing Topology**: They illustrate the topology of the physical components of a system and how software components are deployed on them.
- **Describing Hardware Components**: They describe the hardware components used to deploy software components, including servers, computers, routers, etc.
- **Describing Runtime Nodes**: They describe the runtime processing nodes, which represent the physical hardware where the software components will be executed.

Notations in Deployment Diagrams:

- Deployment diagrams model the runtime architecture of a system and show the configuration of hardware elements (nodes) and how software elements and artifacts are mapped onto those nodes.
- Notations include symbols representing nodes, artifacts, and relationships between them.

Some of the deployment diagram symbols and notations



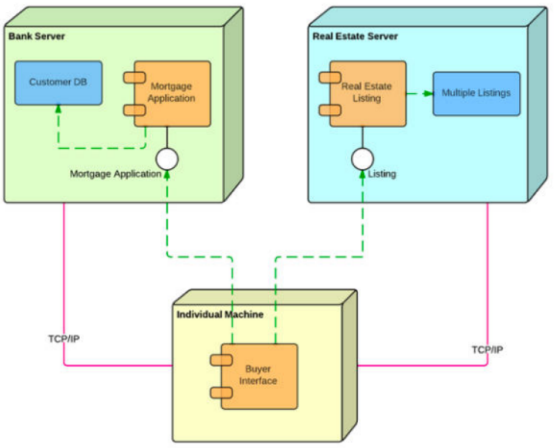
Uses of Deployment Diagrams:

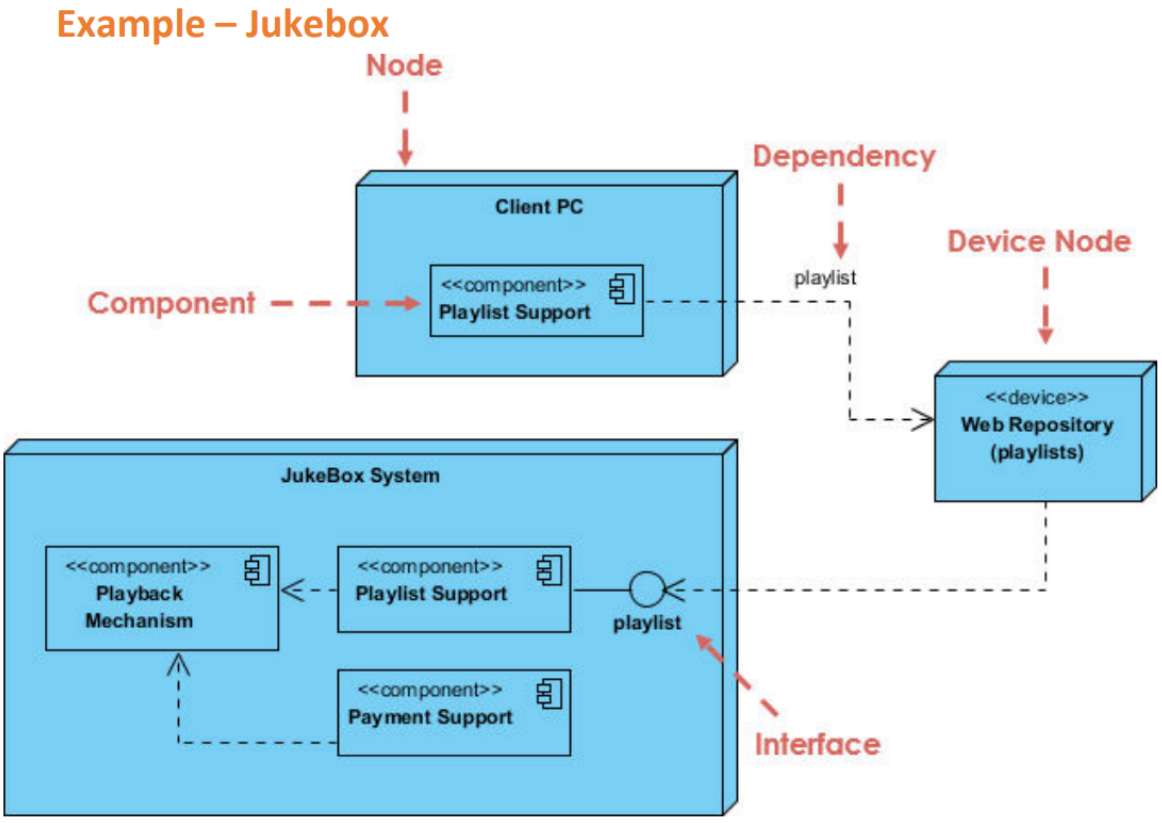
- **System Engineering**: They are useful for system engineers to understand the runtime architecture of a system and how its components are deployed.

- **Controlling Parameters**: An efficient deployment diagram is crucial as it controls parameters such as performance, scalability, maintainability, portability, and understandability of the system.
- **Visualizing, Specifying, and Documenting Systems**: Deployment diagrams are important for visualizing, specifying, and documenting embedded systems, client/server architectures, and distributed systems. They also aid in managing executable systems through forward and reverse engineering processes.

Elements in a Deployment Diagram

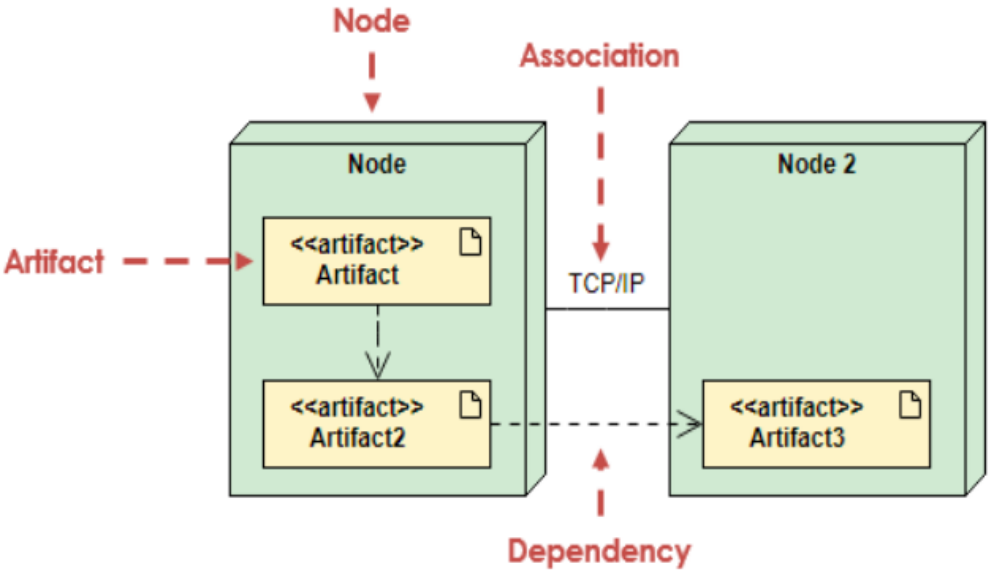
- **Artifact:** A product developed by the software, symbolized by a rectangle with the name and the word "artifact" enclosed by double arrows.
- **Association**: A line that indicates a message or other type of communication between nodes.
- **Component:** A rectangle with two tabs that indicates a software element.
- **Dependency:** A dashed line that ends in an arrow, which indicates that one node or component is dependent on another.
- **Interface:** A circle that indicates a contractual relationship. Those objects that realize the interface must complete some sort of obligation.
- **Node:** A hardware or software object, shown by a three-dimensional box.





Elements in a Deployment Diagram:

1. **Artifact**:
    - Artifacts represent concrete elements in the physical world that result from a development process.
    - Examples include executable files, libraries, archives, database schemas, configuration files, etc.
    - Artifacts are concrete entities caused by the development process and are deployed on nodes.



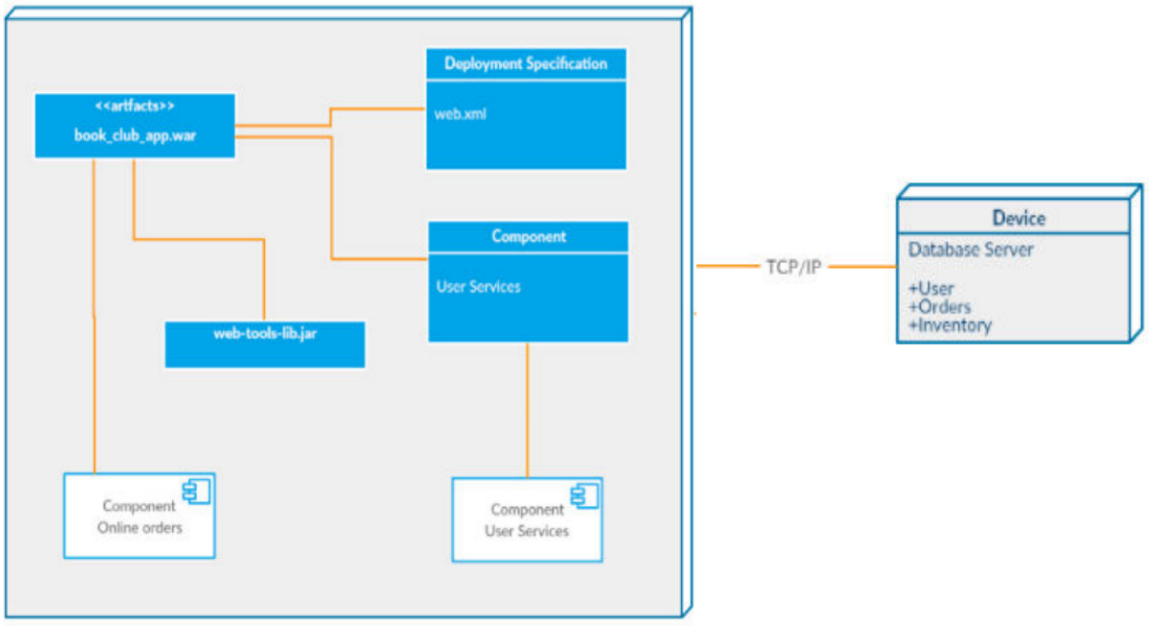
2. **Component with Artifacts**:
    - Represents concrete real-world entities related to software development.
    - These are things that participate in the execution of a system or executable entities residing in nodes.
    - They represent the physical packaging of logical elements, and artifacts are deployed on these nodes.
    - Each artifact has a filename indicating its physical location.
3. **Associations**:
    - Associations represent relationships between different elements in the deployment diagram.
4. **Nodes**:
    - Nodes are computational resources on which component artifacts are deployed for execution.
    - They serve as the execution environment for components and artifacts.

How to Draw a Deployment Diagram:

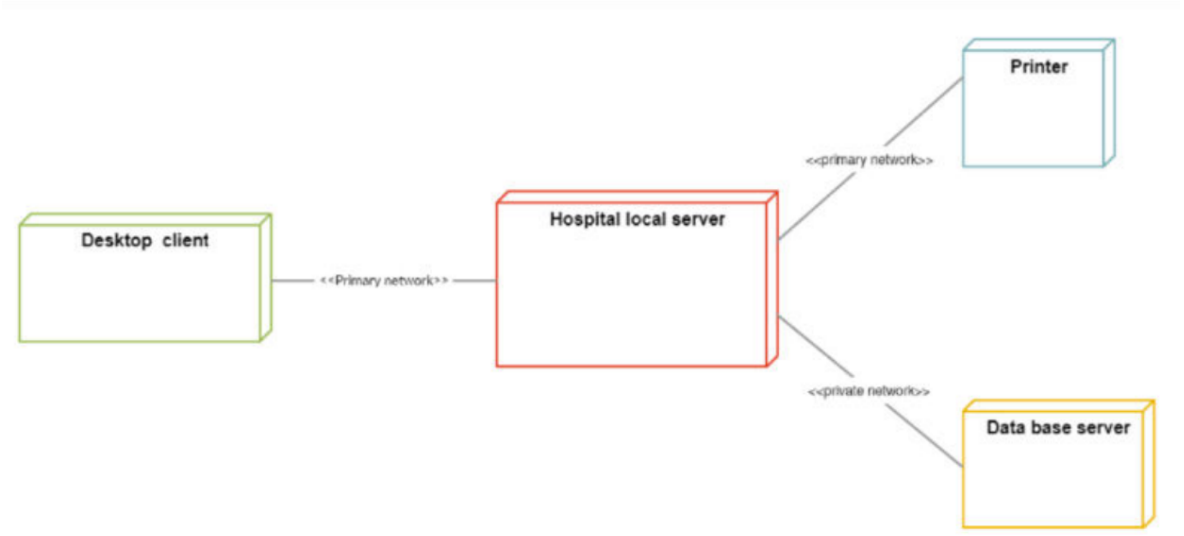
1. **Identify Purpose**: Determine the purpose of the deployment diagram and identify the nodes and devices within the system to be visualized.

2. **Define Relationships**: Determine the relationships between nodes and devices, and add communication associations to the diagram.
3. **Identify Additional Elements**: Identify other elements such as components, active objects, etc., and add them to the diagram as needed.
4. **Add Dependencies**: Add dependencies between components and objects as required to complete the diagram.

Example – Online Shopping System

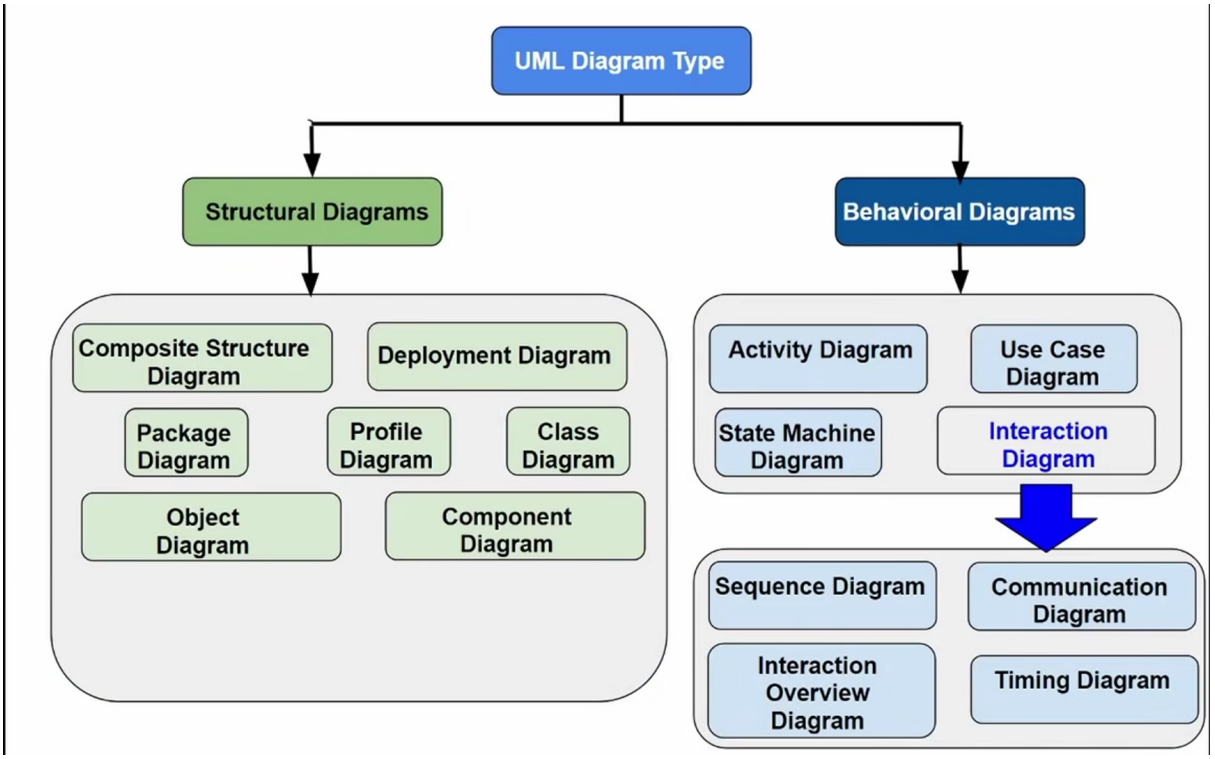


Example – Hospital Management System



Activity Modelling: UML Activity Diagrams, Modelling and Guidelines

Interaction Modeling



Use Case Model:

- Describes how a system interacts with outside actors.
- Each use case represents a piece of functionality that the system provides to its users.
- Helpful for capturing informal requirements and understanding system functionalities at a high level.

Sequence Model:

- Sequence diagrams provide more detail than use case diagrams.
- They show the messages exchanged among a set of objects over time.
- Messages can include both asynchronous signals and procedure calls.
- Sequence diagrams depict the behavior sequences seen by users of a system.

Activity Model:

- Activity diagrams provide even further detail than sequence diagrams.
- They show the flow of control among the steps of a computation.
- Activity diagrams can also represent data flows in addition to control flows.
- Document the steps necessary to implement an operation or a business process referenced in a sequence diagram.

Activity Diagram: Notations and Symbols

1. An activity diagram visually presents a series of actions or flow of control in a system similar to a flowchart or a data flow diagram.
2. Activity diagrams are often used in business process modeling. They can also describe the steps in a use case diagram.
3. Activities modeled can be sequential and concurrent.
4. In both cases an activity diagram will have a beginning (an initial state) and an end (a final state).

Initial State or Start Point:

- A small filled circle followed by an arrow represents the initial action state or the starting point of an activity diagram.
- In diagrams with swimlanes, the start point is placed in the top left corner of the first column.

Activity or Action State:

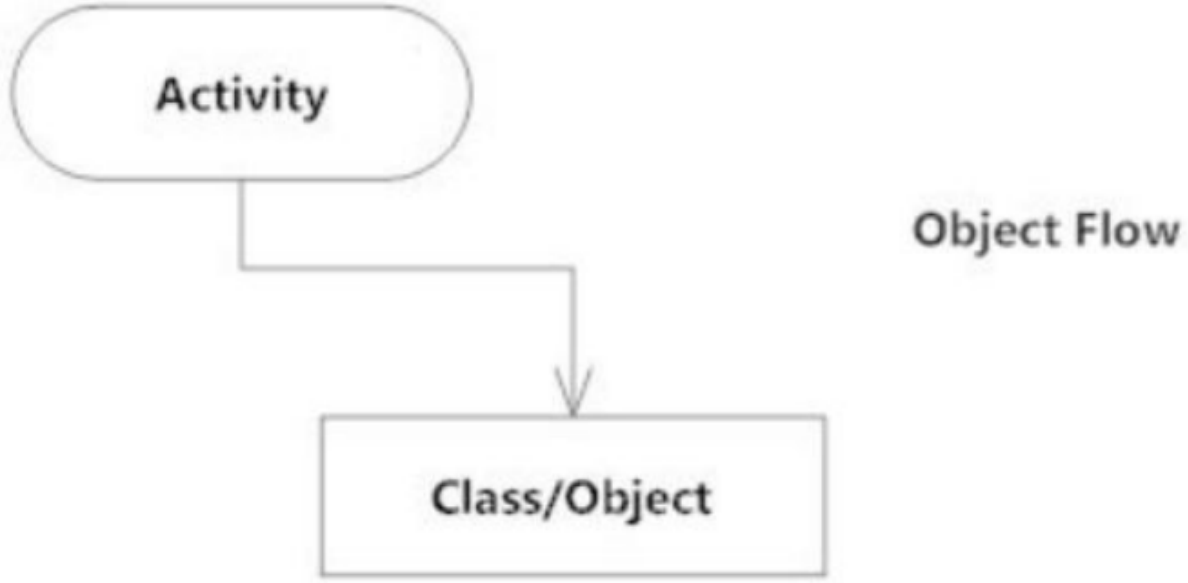
- An action state represents non-interruptible actions of objects.
- It is depicted as a rectangle with rounded corners.

Action Flow:

- Action flows illustrate transitions from one action state to another.
- They are drawn with arrowed lines.

Object Flow:

- Refers to the creation and modification of objects by activities.
- An arrow from an action to an object indicates that the action creates or influences the object, while an arrow from an object to an action indicates that the action uses the object.

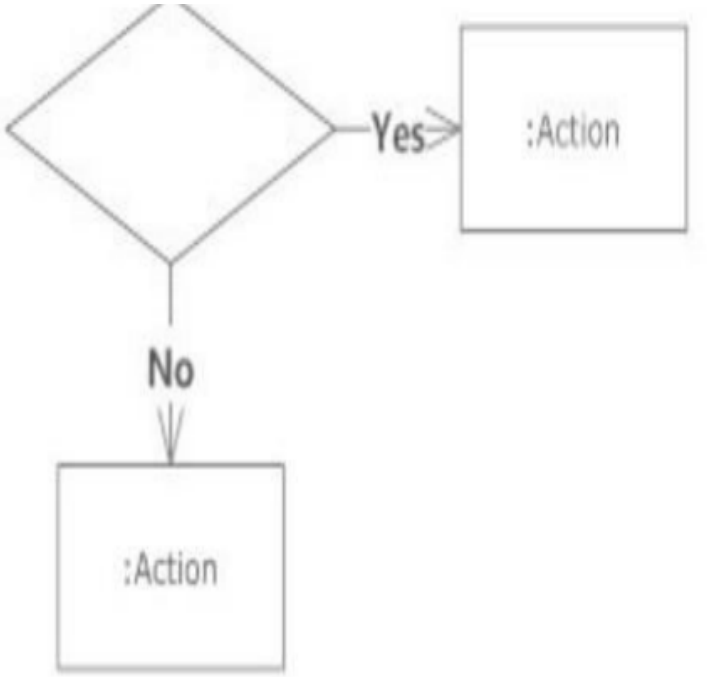


Decisions and Branching:

- Represented by a diamond shape with alternate paths.
- Outgoing paths are labeled with conditions or guard expressions, with one path typically labeled as "else".

Guards:

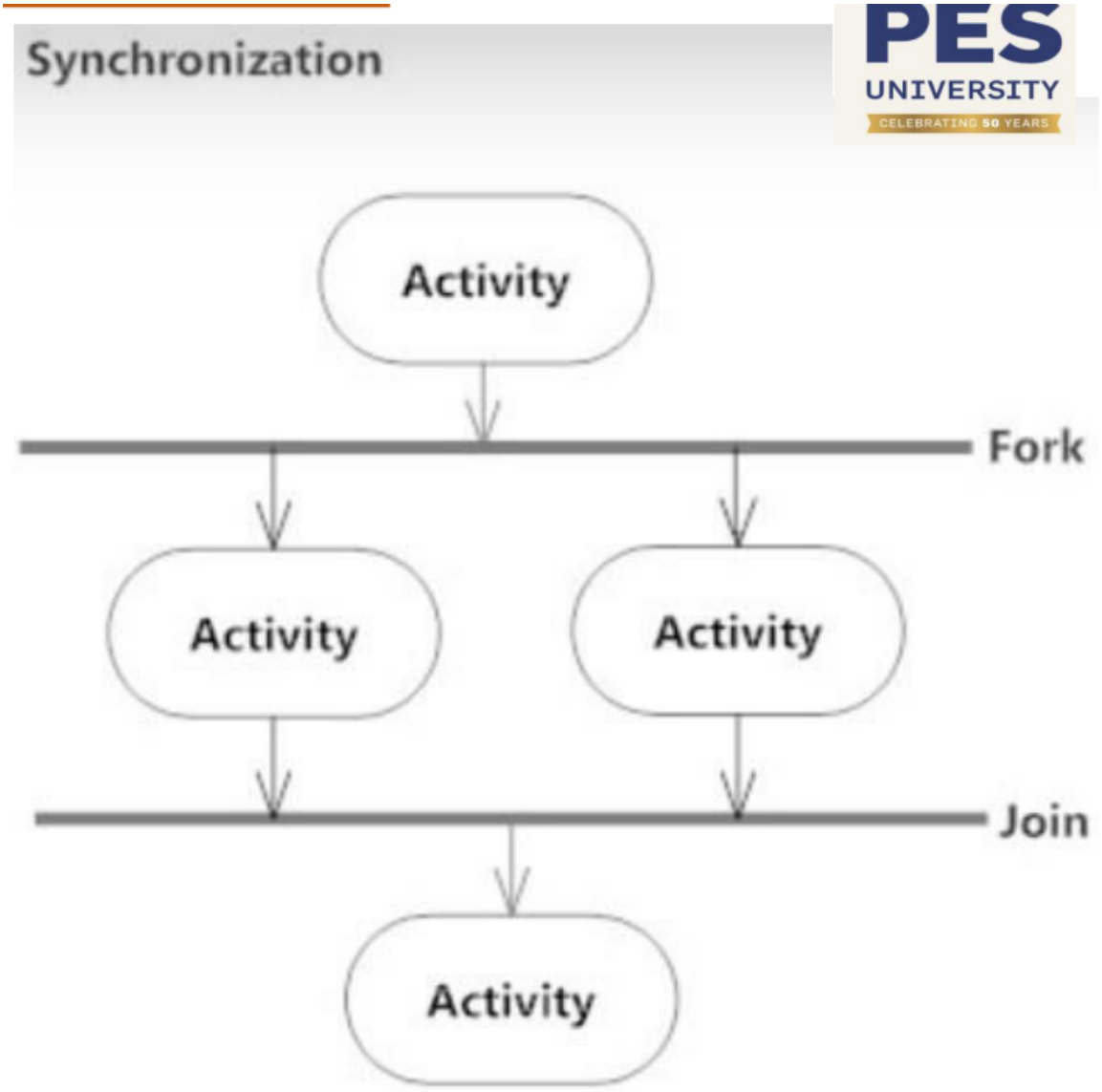
- Statements written next to a decision diamond that must be true before moving to the next activity.
- Useful for ensuring specific conditions are met before progressing.



Guard Symbols

Synchronization:

- Forks split a single incoming flow into multiple concurrent flows, while joins merge multiple concurrent flows back into a single outgoing flow.
- Used together, forks and joins illustrate parallel occurrences of activities.

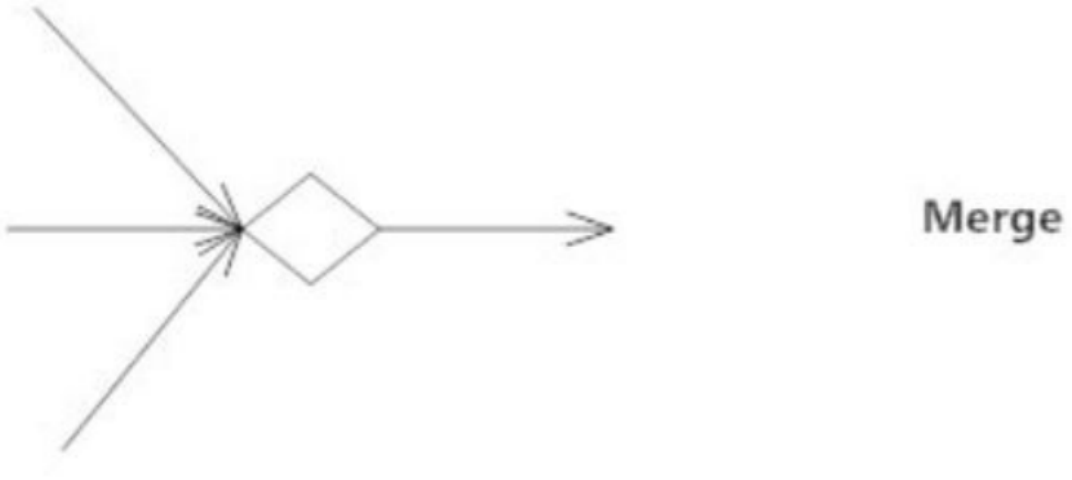


Time Event:

- Represents an event that pauses the flow for a specific duration.
- Depicted by an hourglass symbol.

Merge Event:

- Brings together multiple flows that are not concurrent.

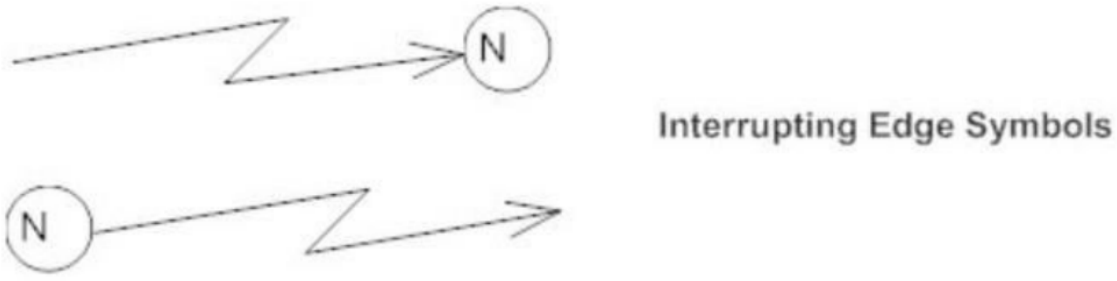


Sent and Received Signals:

- Represent how activities can be modified from outside the system.
- Usually depicted as pairs of sent and received signals, indicating that the state can't change until a response is received.

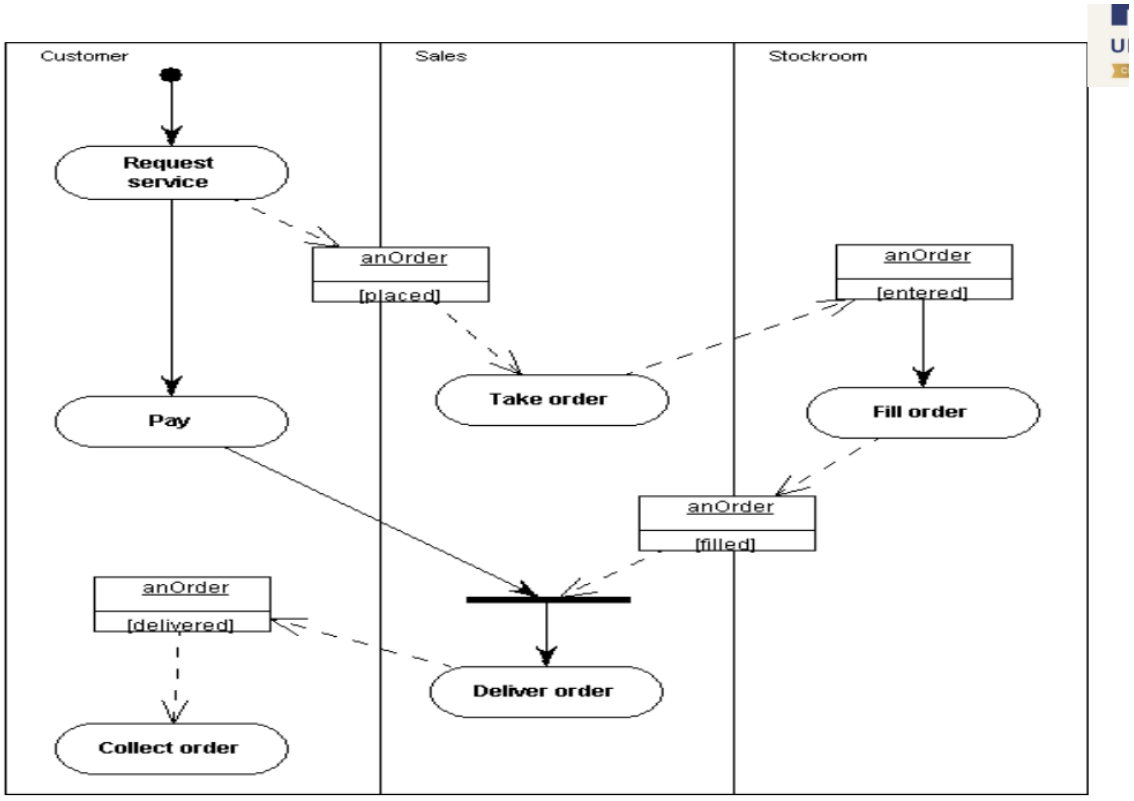
Interrupting Edge:

- Denotes an event, such as a cancellation, that interrupts the flow, often represented by a lightning bolt symbol.



Swimlanes:

- Group related activities into columns to organize and structure the diagram.



Final State or End Point:

- An arrow pointing to a filled circle nested inside another circle represents the final action state or endpoint of the activity diagram.

Construct an Activity diagram

Step 1: Identify Action Steps

Identify the various activities and actions that constitute your business process or system.

Step 2: Identify Actors

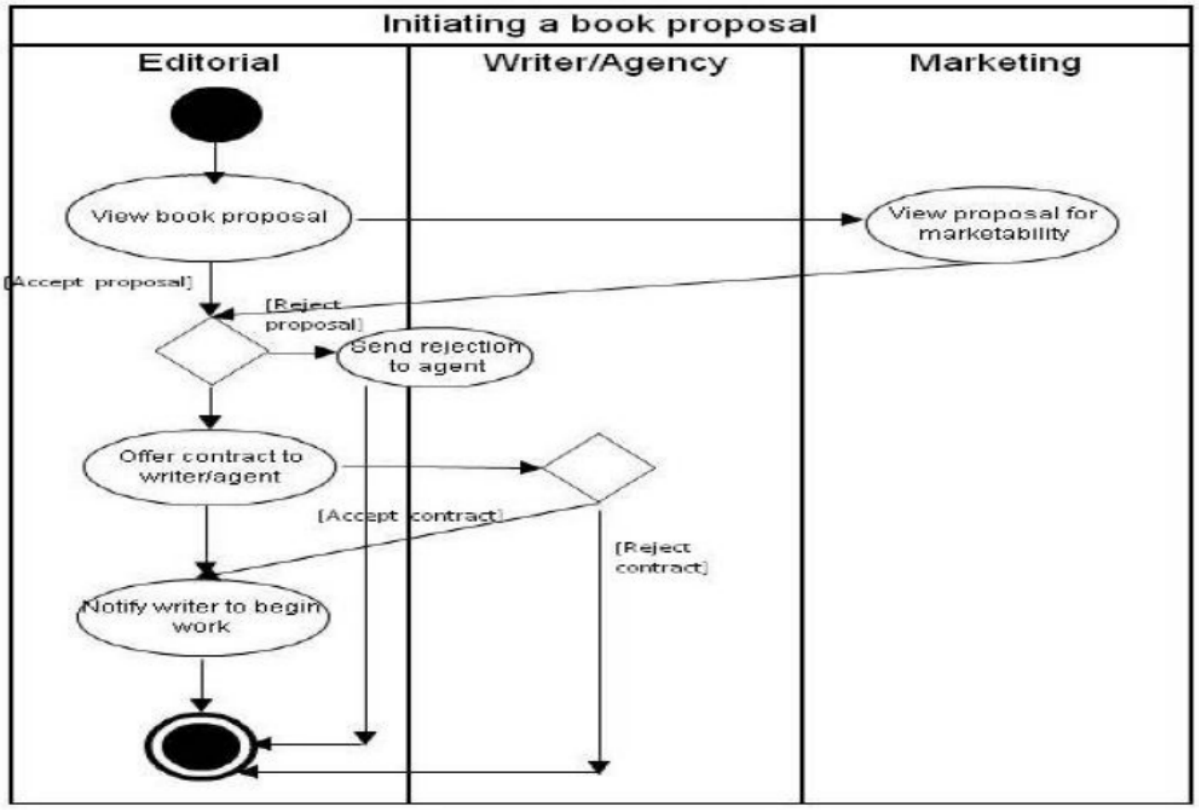
Determine the actors involved in the system. Understanding the actors makes it easier to discern each action they are responsible for.

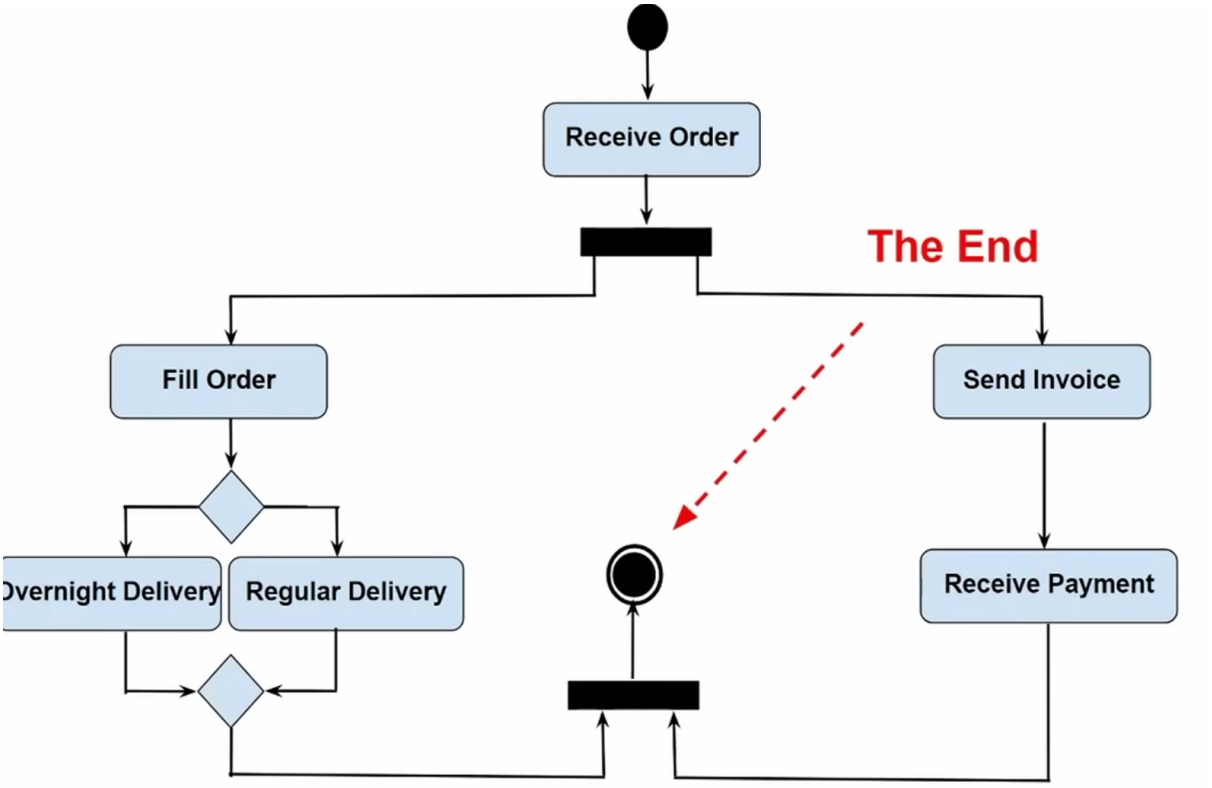
Step 3: Establish Flow Among Activities

Determine the order in which actions are processed. Identify conditions that must be met to carry out certain processes, such as simultaneous actions, branching in the diagram, or dependencies between actions.

Step 4: Add Swimlanes

Group actions based on responsibility and assign them to swimlanes. Each swimlane represents a group of actors responsible for carrying out specific actions. Grouping actions under swimlanes helps organize and clarify the responsibilities of different entities involved in the system.





Behaviour Modelling: Sequence Diagram

A Sequence diagram is a structured representation of behavior as a series of sequential steps over time. It serves several purposes:

- **Depict Workflow and Message Passing:** Sequence diagrams illustrate how elements cooperate over time to achieve a result. They show the flow of information and responsibility throughout the system.
- **Capture Flow of Information and Responsibility:** Sequence diagrams help capture the flow of information and responsibility early in the analysis phase. Messages exchanged between elements in a sequence diagram eventually translate into method calls in the class model.
- **Model Use Case Scenarios:** Sequence diagrams can be used to create explanatory models for use case scenarios. By including an Actor and elements involved in the use case, you can represent the sequence of steps that the user and the system undertake to complete the required tasks.

Sequence Diagram Construction

- Sequence elements are arranged horizontally, with messages passing back and forth between elements.

- Messages in a Sequence diagram can be of various types and can reflect the operations and properties of the source and target elements.
- An Actor element represents the user initiating the flow of events.
- Stereotyped elements, such as Boundary, Control, and Entity, can be used to illustrate screens, controllers, and database items, respectively.
- Each element in a sequence diagram has a dashed stem called a Lifeline, which represents where the element exists and potentially takes part in the interactions.

Actor

Actor is a pivotal concept in system design, serving as a representation of any entity that interacts with the system, regardless of whether it's a human user, a machine, or even another system or subsystem. Actors are typically associated with Use Cases and play a crucial role in defining system requirements.

1. **User of the System:** An Actor represents entities that interact with the system from the outside or system boundary. This interaction could occur through various mediums such as graphical user interfaces, batch interfaces, or other means.
2. **Use Case Interaction:** An Actor's interaction with the system is documented in Use Case scenarios. These scenarios detail the functions that the system must provide to satisfy user requirements. Actors help define the scope of a system and the functionalities it should support.
3. **Representation in Sequence Diagrams:** In Sequence diagrams, Actors are depicted using rectangle notation. They represent the roles played by users or other entities in the sequence of interactions depicted in the diagram.
4. **Stereotyped Elements:** Enterprise Architect supports stereotyped Actor elements for business modeling purposes. These elements help in representing Actors in the system and their roles in interacting with the system components.

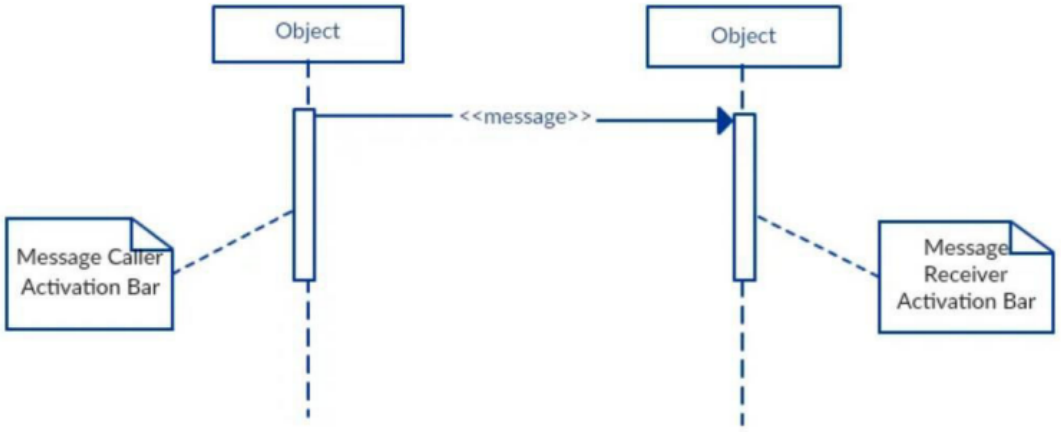
Lifeline

1. **Individual Participant:** A Lifeline represents a single participant or object in an interaction scenario depicted in a Sequence diagram. Each Lifeline corresponds to a specific object or component involved in the sequence of events.

2. **Distinct Connectable Element:** Lifelines are distinct and identifiable elements within a Sequence diagram. They depict the existence and potential participation of objects or components in the interactions illustrated in the diagram.
3. **Instance Classifier Specification:** In Enterprise Architect, you can specify the representation of a Lifeline by right-clicking on it and selecting the 'Advanced | Instance Classifier' option. This allows you to choose the appropriate project classifiers to associate with the Lifeline.
4. **Availability in Sequence Diagrams:** Lifelines are a fundamental element in Sequence diagrams and are commonly used to depict the flow of messages and interactions between objects or components over time.
5. **Different Lifeline Elements:** While Sequence diagrams primarily use Lifelines, it's worth noting that there are different Lifeline elements for Timing diagrams as well. These include State Lifelines and Value Lifelines, each serving specific purposes within the context of Timing diagrams. However, regardless of the diagram type, the fundamental meaning and purpose of Lifelines remain consistent.

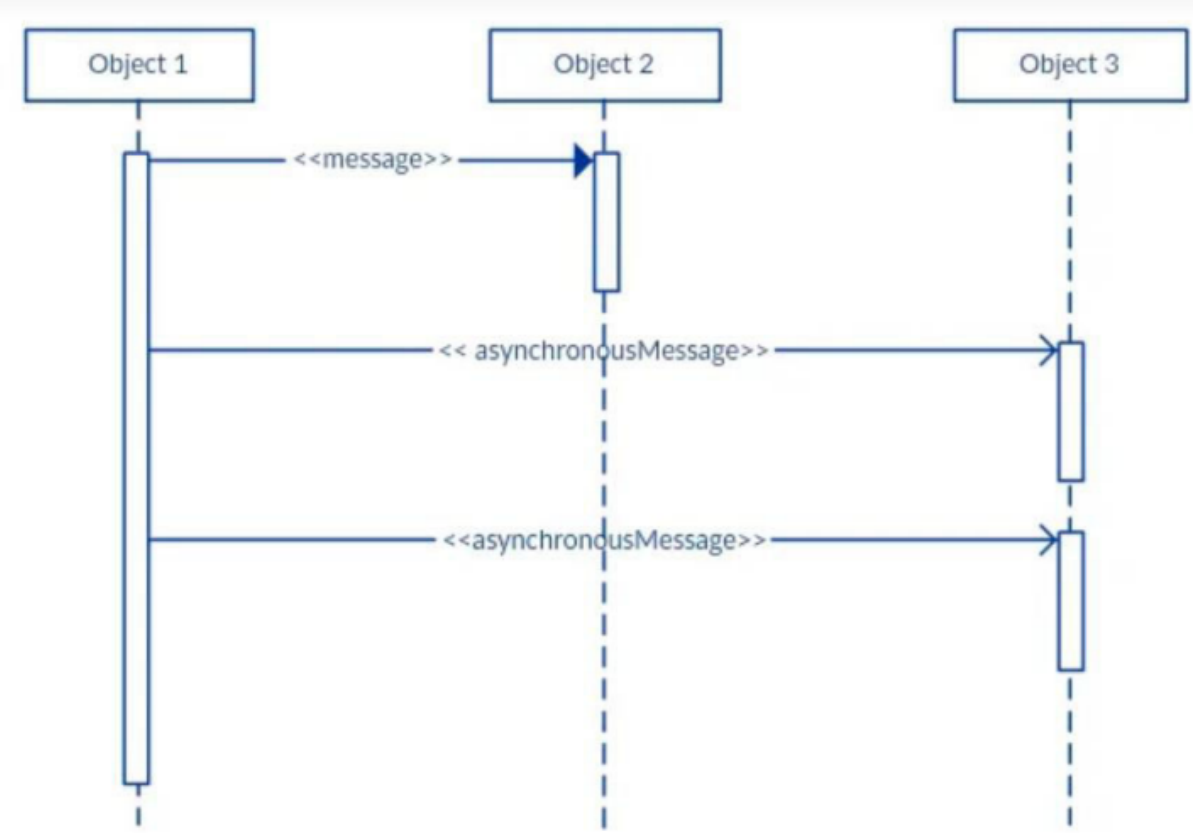
Activation Bar (Execution Occurrence):

- Indicates the period during which an object is active or performing some action.
- Shows when messages are being sent and received.



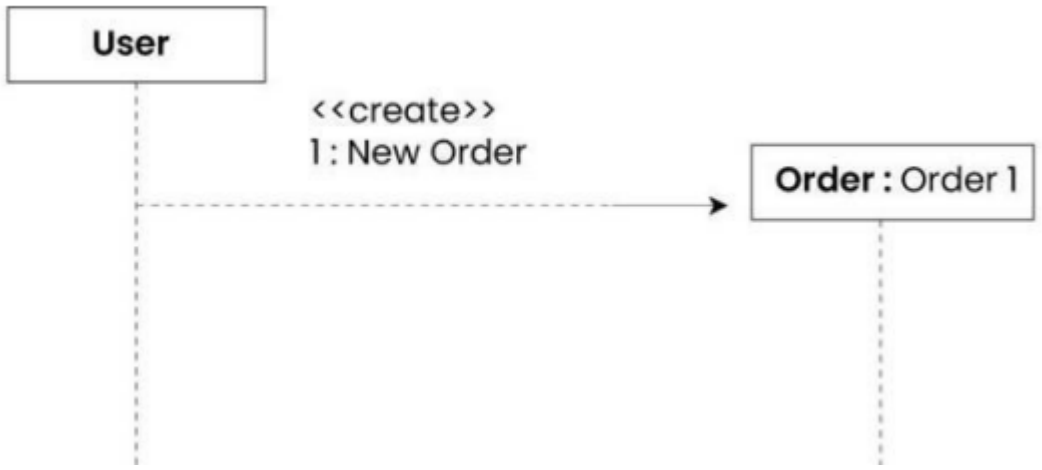
Message:

- Represents communication or interaction between objects.
- **Messages can be synchronous (solid line with filled arrowhead), asynchronous (dashed line with open arrowhead), or a return message (dotted line).**



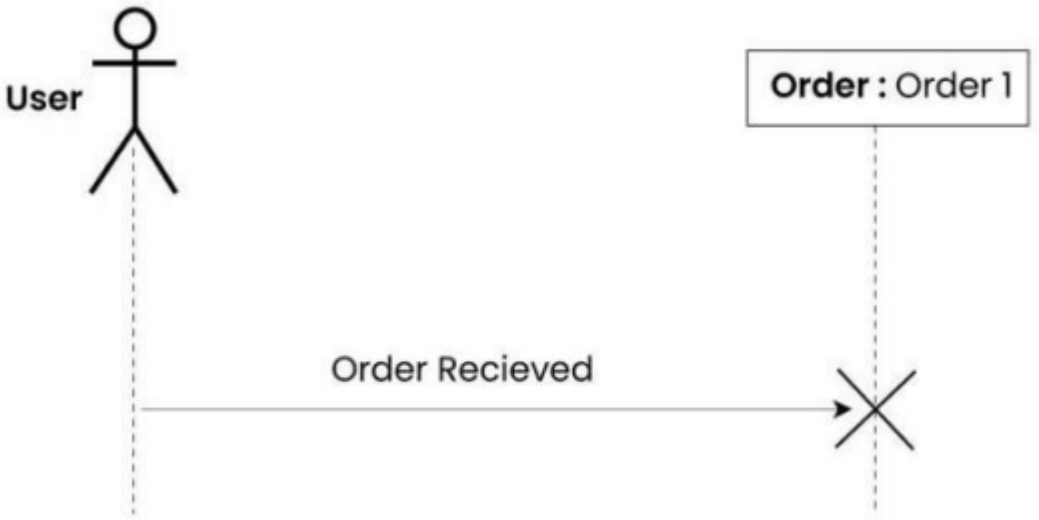
Create Message:

- Used to instantiate a new object in the sequence diagram.
- Represented by a dotted arrow with the word 'create' labeled on it.



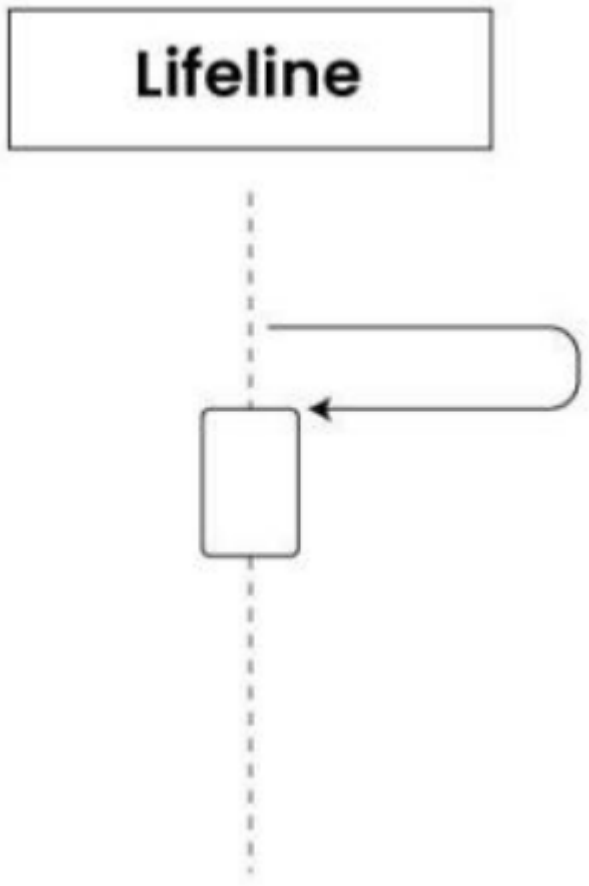
Destroy/Delete Message:

- Denotes the termination or destruction of an object.
- Symbolized by an X-shaped arrowed line.



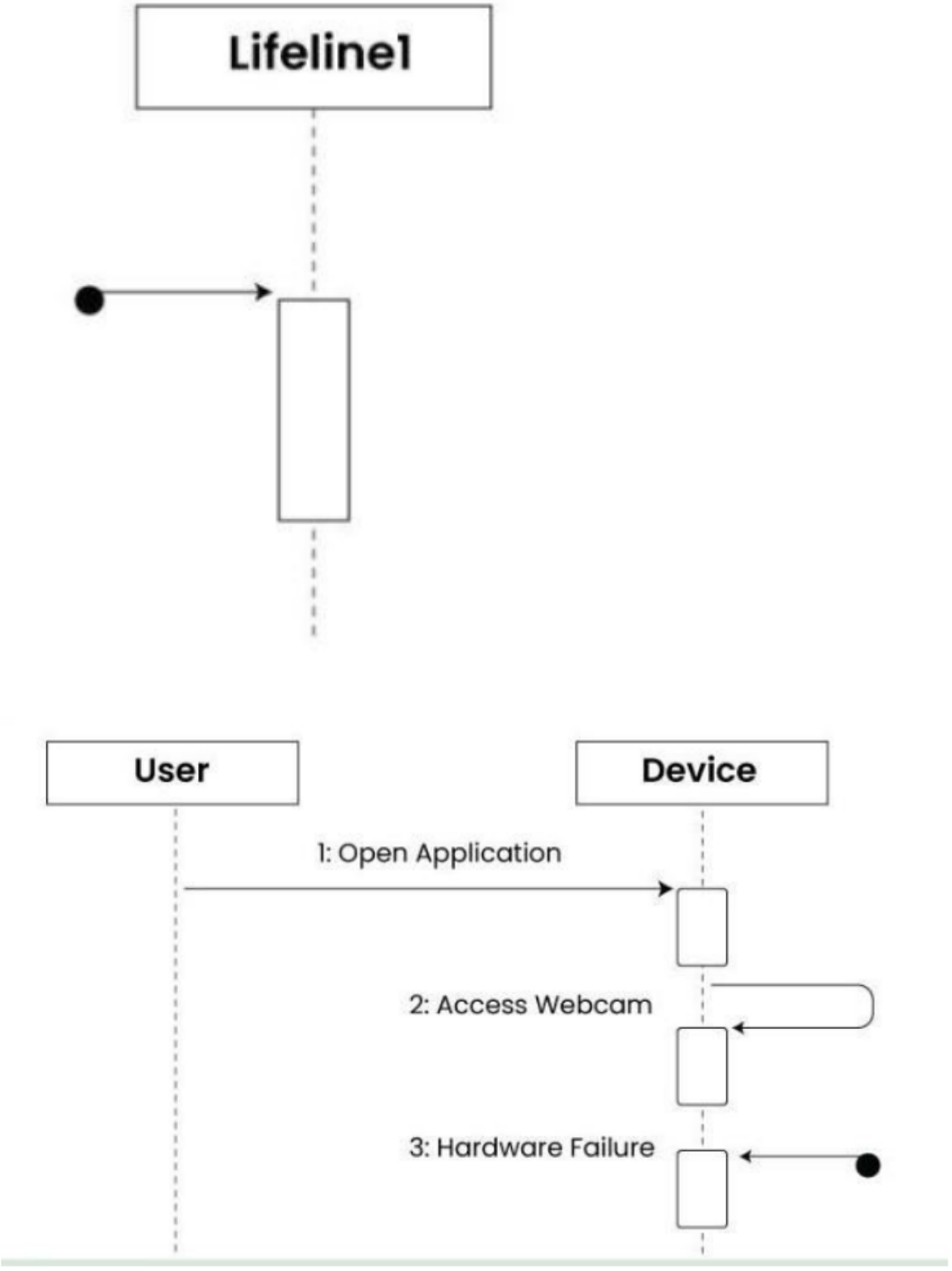
Self Message:

- Denotes a message sent by an object to itself.
- Represents actions or method calls within the same object.
- Symbolized by a loop arrowed line.



Found Message:

- Represents a scenario where an unknown source sends the message.
- Symbolized by an arrow directed towards a lifeline from an endpoint.

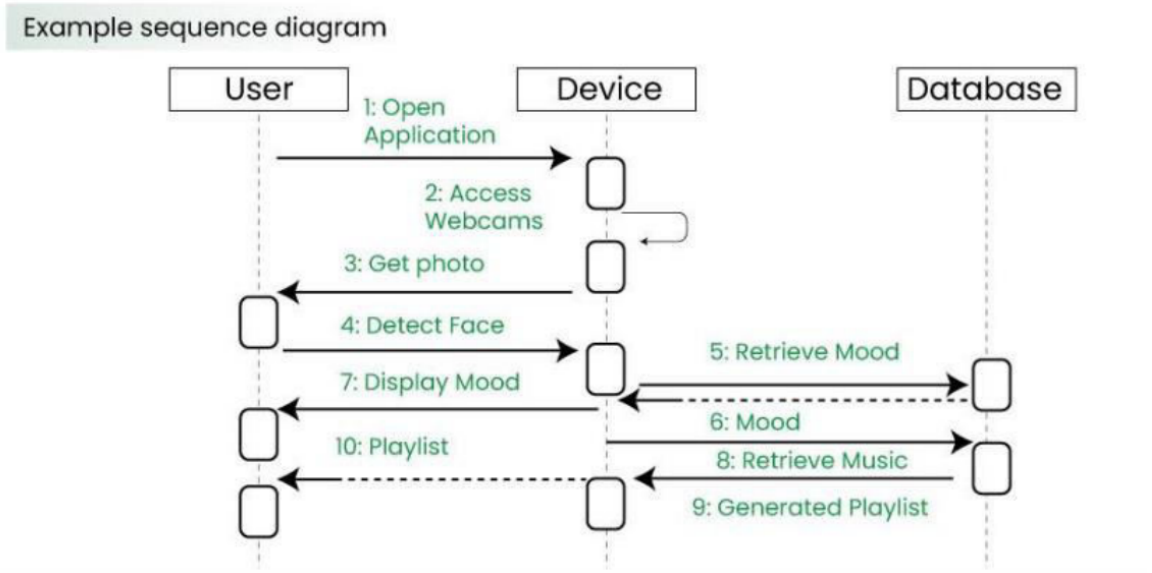


Guide to constructing a Sequence Diagram:

1. **Identify the Use Case:** Clearly define the scenario or use case that you want to represent with the sequence diagram. Understand the purpose of the interaction and the actors involved.

2. **Identify Objects/Participants:** Identify the main objects or participants (actors, classes, components) that are part of the interaction. These will be represented by lifelines in the sequence diagram.
3. **Draw Lifelines:** Draw vertical dashed lines to represent the lifelines of each identified object or participant. Lifelines indicate the existence of an object over time.
4. **Arrange Lifelines:** Position the lifelines vertically, ensuring they are spaced appropriately. This layout represents the temporal order of the participating objects.
5. **Determine the Order of Execution:** Identify the order in which the objects will execute their actions. This helps in organizing the sequence of messages in the diagram.
6. **Draw Messages:** Use arrows to draw messages between the lifelines to represent interactions or communications between objects. Different types of messages (synchronous, asynchronous, etc.) can be indicated by varying line styles and arrowheads.
7. **Label Messages:** Label each message with the method or operation being called, along with any parameters or return values. This adds clarity to the diagram and helps in understanding the purpose of each message.
8. **Include Create and Destroy Messages:** If necessary, include create and destroy messages to represent the instantiation and termination of objects. These are particularly relevant for dynamic interactions.
9. **Add Conditions and Loops (Optional):** If the sequence involves conditions or loops, you can use combined fragments or loop/alt/option boxes to indicate these structures within the sequence diagram.
10. **Include Additional Details:** Add annotations, notes, or comments to provide additional information or context that might be important for understanding the sequence of events.
11. **Review and Refine:** Review the sequence diagram to ensure it accurately represents the desired interaction. Refine the diagram as needed to improve clarity and readability.
12. **Consider Time Constraints (Optional):** If necessary, include duration constraints to specify the minimum and maximum time duration of certain interactions.

13. **Validate and Iterate:** Validate the sequence diagram with stakeholders or team members to ensure it accurately captures the intended behavior. Iterate on the diagram based on feedback received.
14. **Create Additional Diagrams (Optional):** If the interaction is complex, consider creating separate sequence diagrams for different aspects of the interaction or breaking it down into smaller, more manageable parts.



UML State Machine Diagrams and Models

Behavior modeling is a crucial aspect of system design and analysis, aiming to provide clarity on internal processes, business processes, or interactions between different systems. In the realm of Unified Modeling Language (UML), behavioral diagrams are employed to depict elements of a system that are time-dependent, illustrating dynamic concepts and their relationships.

State Diagram:

- A state diagram, also known as a state machine diagram, describes state-dependent behavior for an object.
- It showcases how an object responds differently to the same event based on its current state.
- State diagrams offer a reductionist view of behavior, focusing on individual objects and their states.

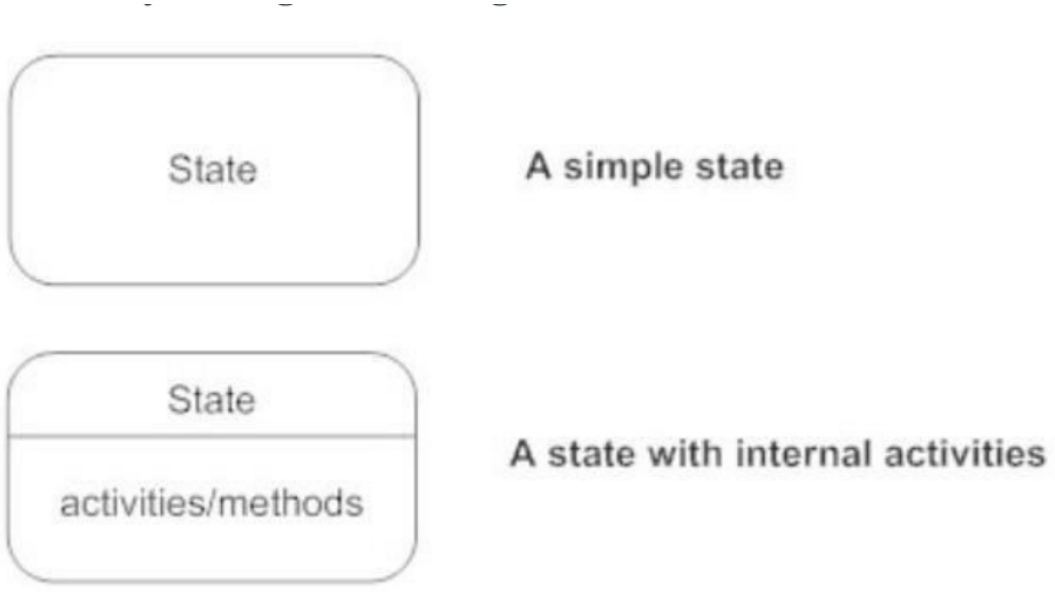
State Diagram vs. Flowchart:

- A flowchart primarily illustrates processes executed within a system that result in changes to the state of objects.
- Conversely, a state diagram specifically showcases the actual changes in state, rather than the processes or commands responsible for those changes.

In essence, while both state diagrams and flowcharts capture aspects of system behavior, state diagrams provide a more focused view on the state-dependent dynamics of objects, offering clarity on how they transition between different states in response to events.

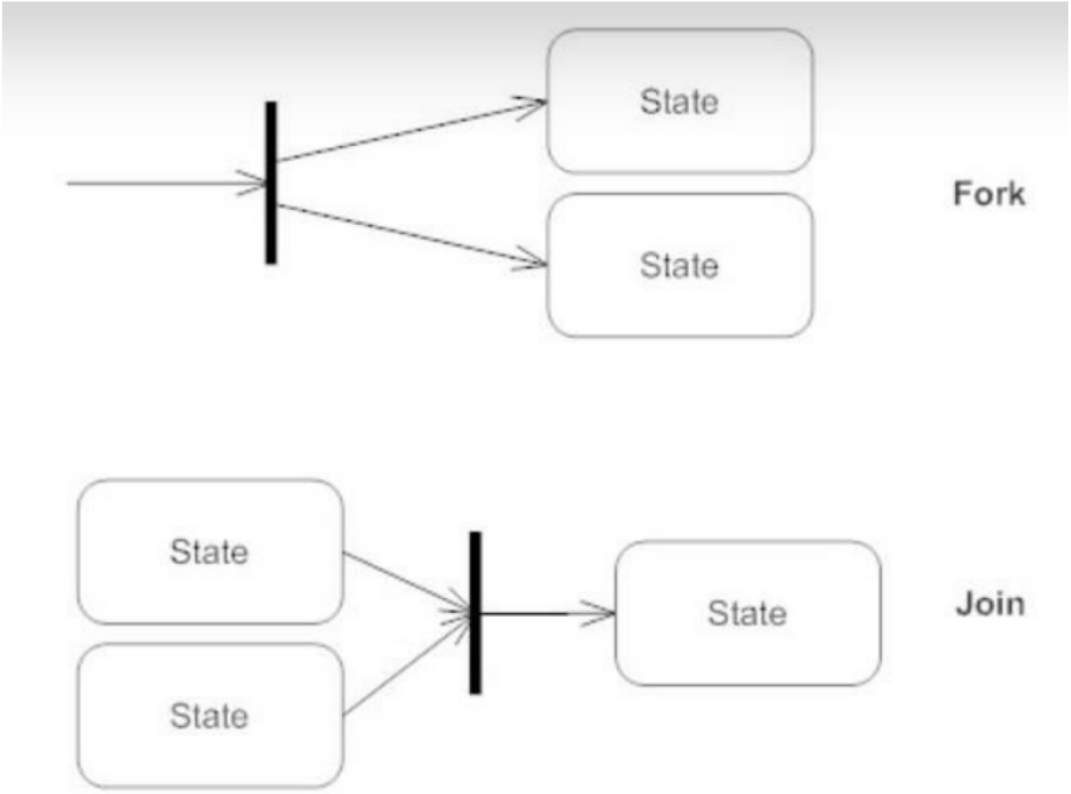
Guide to state diagram:

1. **Identify Initial and Final States**: Determine the initial state of the object, where it starts, and the final state, where it ends.
2. **Define States**: Think about the different states the object might transition through during its lifecycle. For example, in an e-commerce scenario, a product might have states like available, sold out, in cart, purchased, etc.

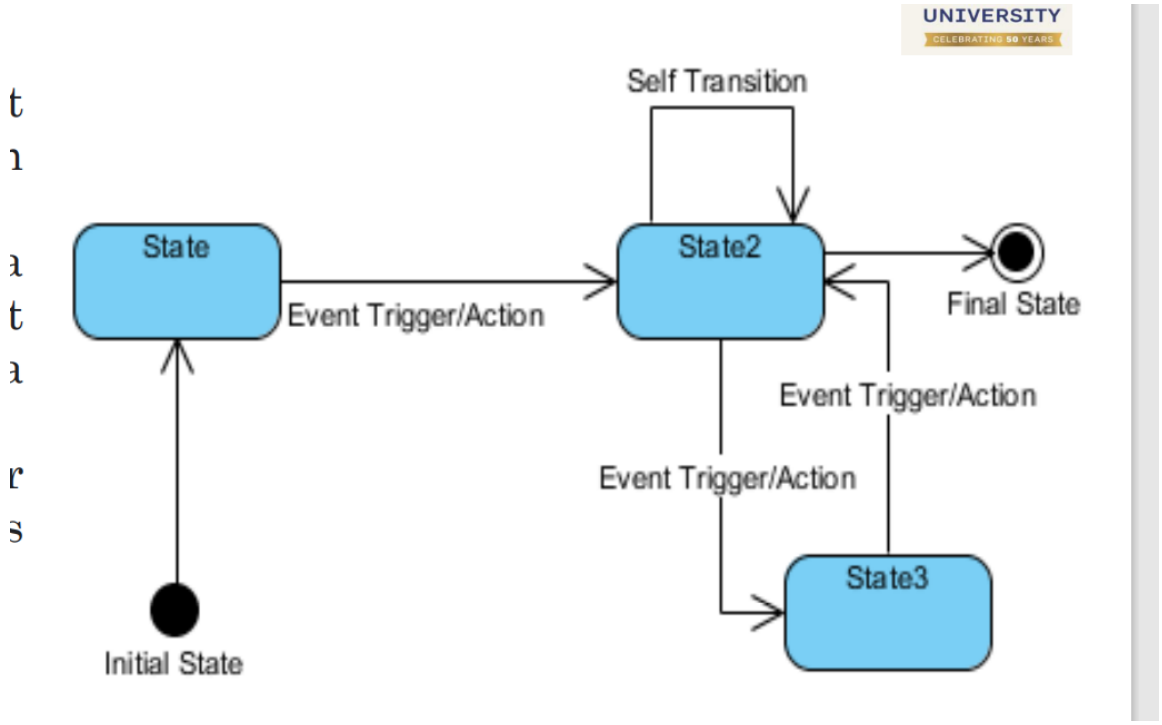


1. **Consider Transitions**: Determine the transitions between states. Certain transitions might only be applicable when the object is in specific states. For example, a product can only be purchased if it's available; it can't be purchased if it's already sold out.

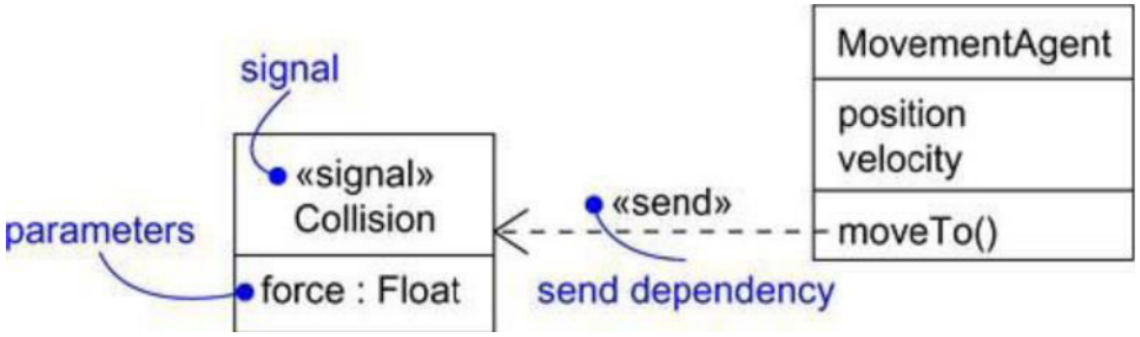
2. **Draw States**: Represent each state using rectangles with rounded corners. Label each state appropriately to describe its meaning.
3. **Draw Transitions**: Use solid arrows to indicate transitions between states. Label each transition with the event that triggers it and the action that results from it. Additionally, transitions can loop back to the same state or lead to other states.
4. **Initial and Final States**: Represent the initial state with a filled circle followed by an arrow. Similarly, represent the final state with an arrow pointing to a filled circle nested inside another circle.
5. **Synchronization and Splitting of Control**: If the object's behavior involves concurrent activities, use short heavy bars to represent synchronization and splitting of control. These bars indicate where a single transition splits into multiple concurrent transitions and where concurrent transitions converge back into one.



6. **Triggers, Events, and Actions**: Specify triggers for transitions, which are events that initiate state changes. Additionally, include guard conditions, which are Boolean conditions that must be satisfied for a transition to occur. Describe the action or activity that happens when a transition occurs.
    a. A **trigger** is an event that initiates a transition from one state to another.
    b. A **guard** condition is a Boolean condition that must be satisfied for a transition to occur.
    c. An **effect** is the action or activity that happens when a transition occurs.

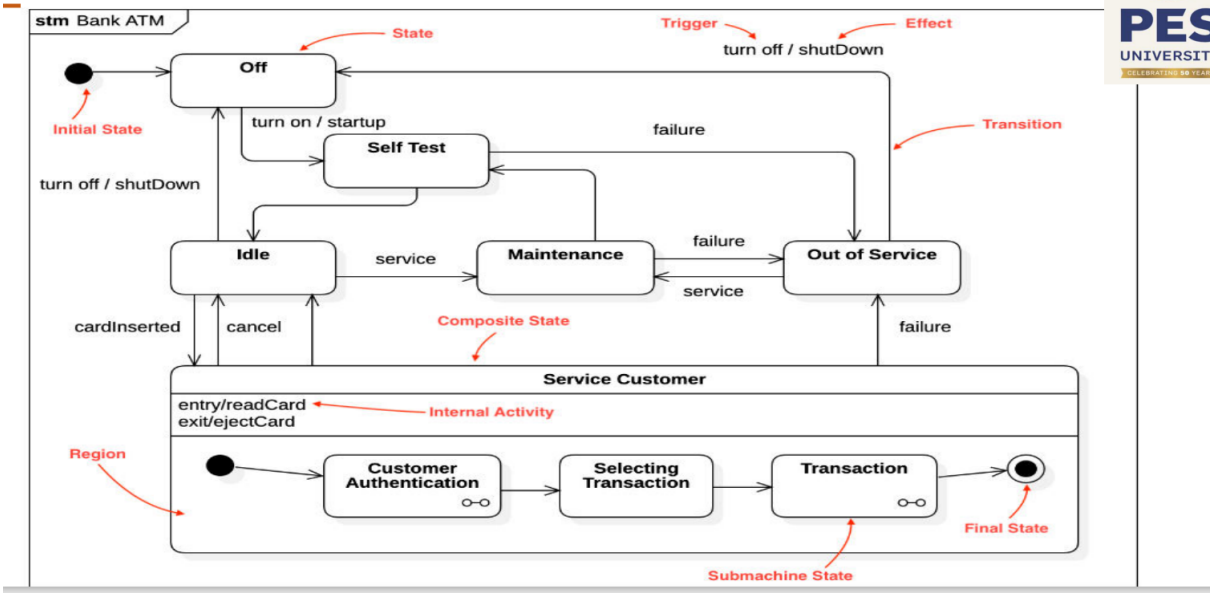


7. **Adding Call, Change, Time, and Signal Events**: Depending on the scenario, you may need to add call events (representing requests to invoke an operation), change events (triggered by a change in a condition), time events (representing the passage of time), or signal events (representing specific messages received by the object).



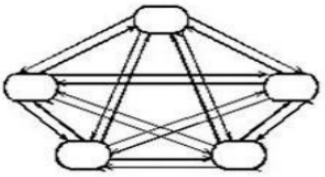
By following these steps and incorporating the appropriate notations and symbols, you can effectively draw a state diagram to model the behavior of your system or object.

Bank ATM



Advanced state Modelling

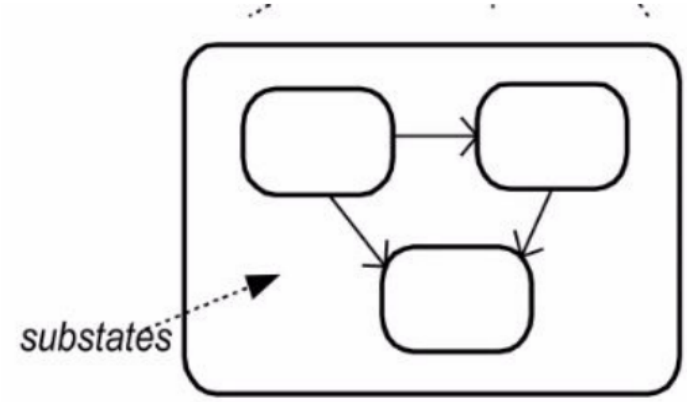
- N independent boolean attributes that affect control. Representing such an object a single flat state diagram would require 2^n States. By partitioning the state into n independent state diagrams, however, only 2n states are required.
- State diagram in Figure in which n^2 transitions are needed to connect every state to every other state. It can be reduced as low as n transitions.



Nested State Diagram

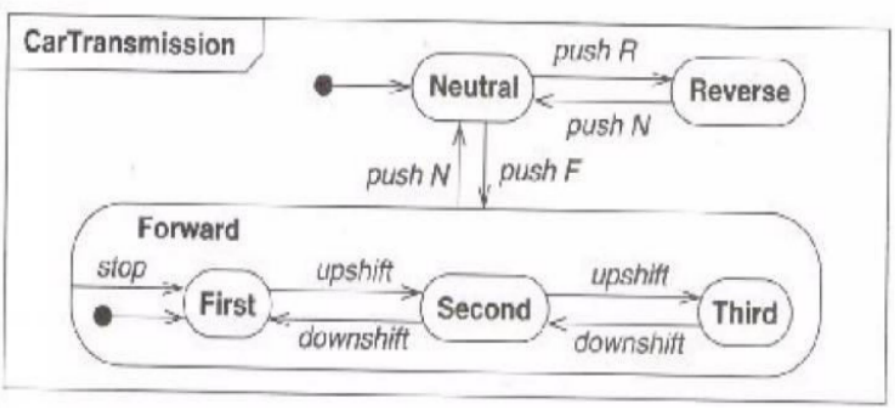
Super or Substate

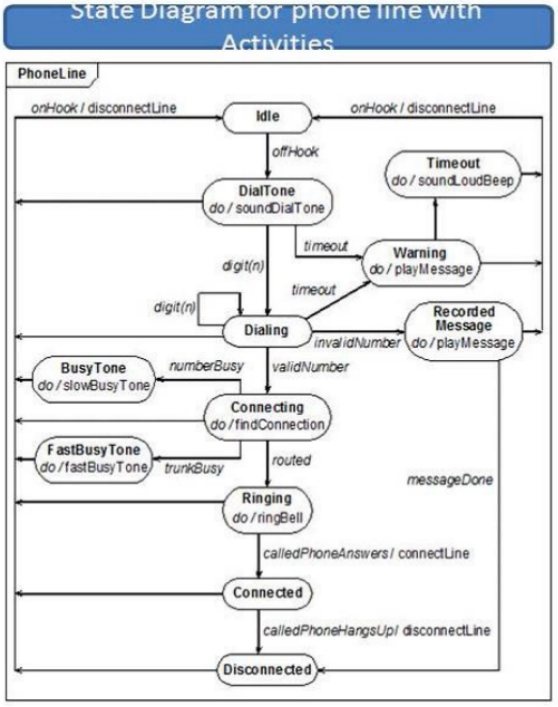
- When one state is complex, you can include substates in it.
  - drawn as nested rounded rectangles within the larger state

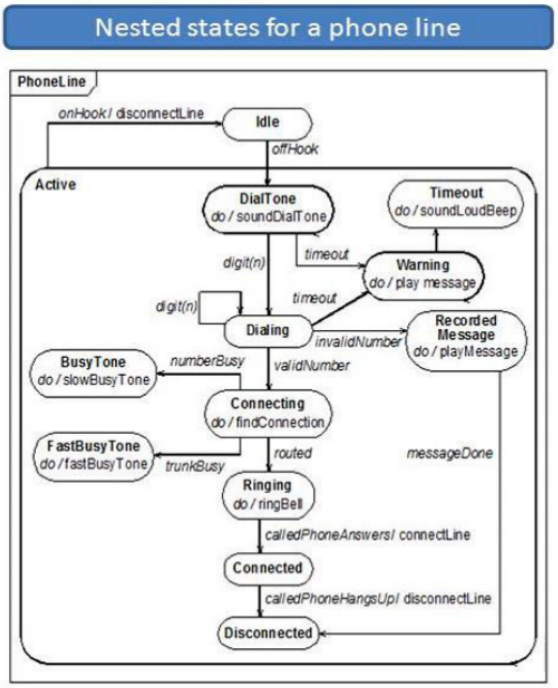


A state may be represented as nested substates.

- In UML, substates are shown by nesting them in a superstate box.
- A substate inherits the transitions of its superstate.

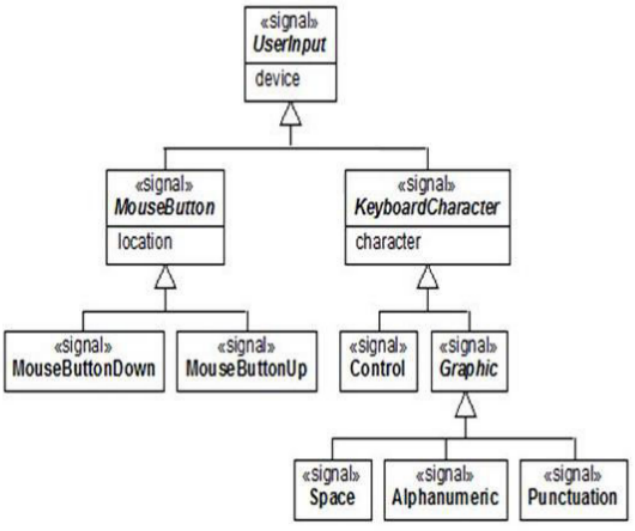






Signal Concurrency (Inheritence)

- Organize signals into a generalization hierarchy with inheritance of signal attributes.



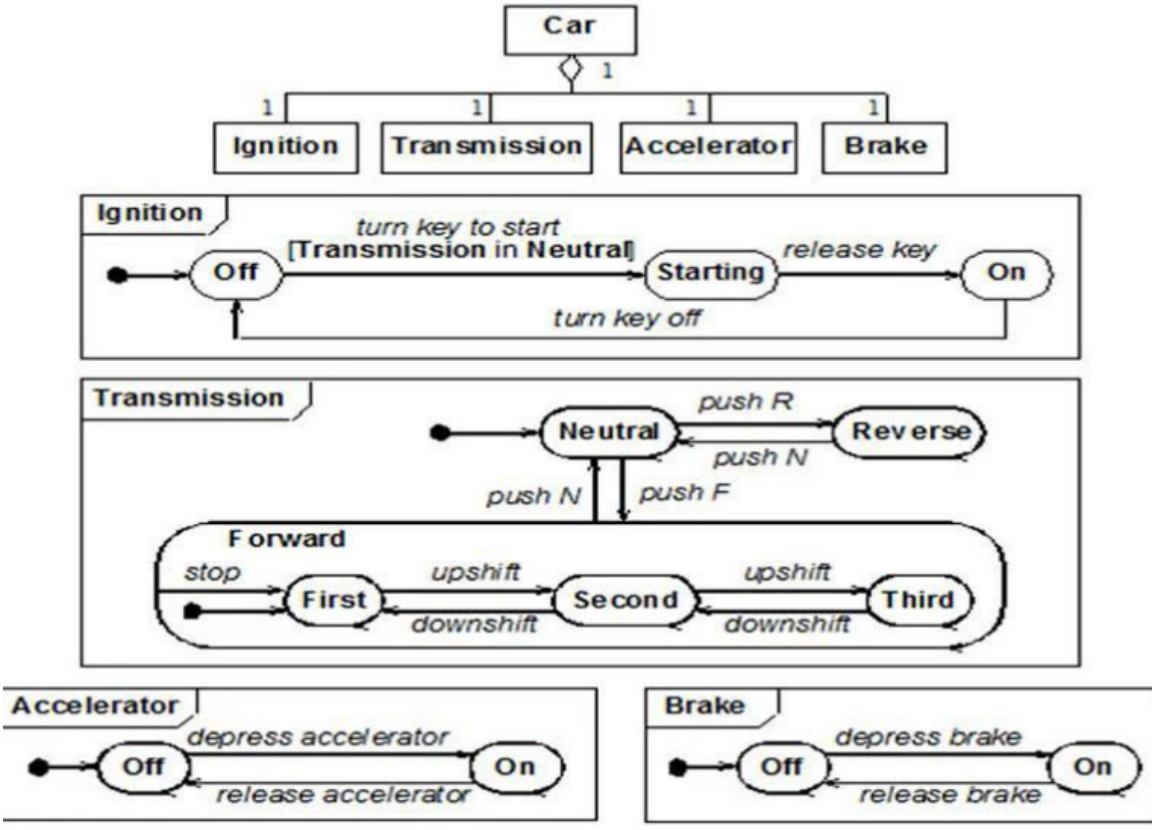
It supports two types of concurrency-

1. Aggregation Concurrency

A state diagram for an assembly is a collection of state diagram, one for each part. The aggregate state corresponds to the combined states of all the parts.

Aggregation is "**and-relationship**".

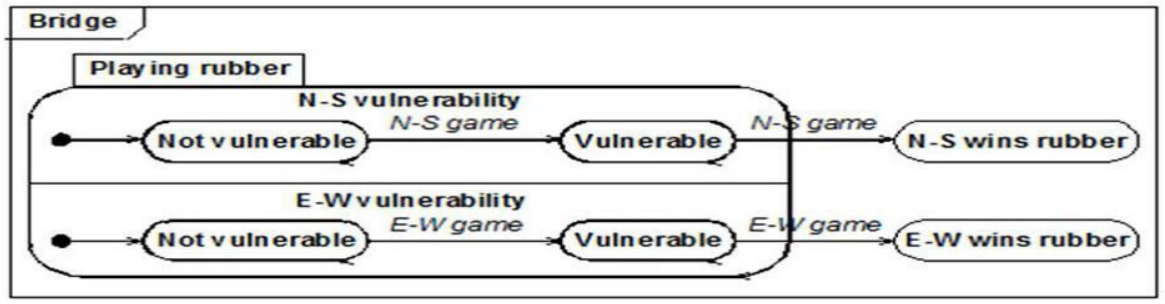
Aggregate state is one state from the first diagram, and a state from second diagram and a state from each other diagram. In the more interesting cases, the part states interact.



2. Concurrency within an object

The state model implicitly supports concurrency among objects. In general, objects are autonomous entities that can act and change state independent of one another.

Objects need not be completely independent and may be subject to shared constraints that cause some correspondence among their state changes.



Concurrency in State Models

State models provide a way to represent complex systems by modeling the different states an object can be in and the transitions between them. Concurrency within this model means the ability for multiple objects or parts of a system to operate simultaneously or in an interleaved way.

1. Aggregation Concurrency

- **Multiple Independent Objects:** In its most basic form, a state model implicitly supports concurrency simply because objects are modeled as independent entities. Multiple objects can act and change their states without direct interference from each other. This assumes no shared constraints between these objects.
- **State Diagrams Working in Parallel:** Think of a system with an ATM machine, a customer account database, and a display for customer transactions. Each of these components can be modeled with its own state diagram, and the overall system state is a combination of the states of each individual object. The ATM state machine, the account database state machine, and the display state machine operate in parallel, handling their respective activities.

2. Concurrency within an Object

- **Substates:** A single object might have multiple aspects that can operate relatively independently. To model this, its state diagram is broken into concurrent substates.
- **Example: Order Processing** An order object could have substates like "Payment Processing", "Shipping Status", and "Customer Notification". Each of these substates can have its own set of transitions and events, operating in parallel within the overall order object.

Illustrative Example: Heart Monitoring Device

Consider a heart monitoring device:

- **Aggregation Concurrency:** The device might have separate components for power supply and the heart monitoring application itself. Each has its own state model, potentially changing state independently.
- **Concurrency within an Object:** The heart monitoring application could be further divided into substates:
  - "Recording Heartbeat": Tracks heart rate and rhythm.
  - "Analyzing Data": Detects anomalies or arrhythmias.
  - "Alert System": Manages alarms or notifications.

How State Models Facilitate Concurrency

- **Modularization:** State models break down complex systems into more manageable components, allowing these components to operate with a degree of autonomy.
- **Event-Driven:** State machines react to events, and multiple events may occur simultaneously, driving state transitions in different parts of the system.
- **Synchronization:** Where needed, objects or substates can coordinate through shared events or synchronization mechanisms.