SDUML LAB

UML

UML, or **Unified Modeling Language**, is a standardized way to visually represent the design of a system. It's widely used in software engineering to describe, specify, and document both software and non-software systems. UML provides a variety of diagrams to capture different aspects of a system, from its structure to its behavior.

Key Components of UML:

1. Structure Diagrams:

These diagrams represent the static aspects of a system, like its architecture and object relationships.

- **Class Diagram**: Describes the structure of a system by showing its classes, attributes, methods, and relationships among objects.
- **Object Diagram**: Shows instances of classes and their relationships at a particular moment.

- **Component Diagram**: Depicts the organization and dependencies of components in a system.
- **Deployment Diagram**: Describes the physical deployment of artifacts (like software components) on hardware nodes.
- **Package Diagram**: Organizes elements into related groups, showing how large systems are divided

2. Behavior Diagrams:

These diagrams represent dynamic aspects of the system, such as how the system behaves and interacts with users and other systems.

- **Use Case Diagram**: Describes how users (actors) interact with a system, capturing functional requirements.
- **Activity Diagram**: Represents workflows of stepwise activities and actions with support for choice, iteration, and concurrency.
- **State Machine Diagram**: Shows the lifecycle of an object, how it moves from one state to another in response to events.

Sequence Diagram: Models the sequence of messages passed between objects in a specific scenario or use case.

Collaboration Diagram: (also called Communication Diagram) Describes how objects interact and communicate, focusing on relationships.

Key Elements in UML Diagrams:

- **Actors**: Represent external entities (like users or other systems) that interact with the system.
- **Classes**: Describe the blueprint for objects (with attributes and methods).
- **Objects**: Instances of classes, representing real elements of a system.
- **Messages**: Communication between objects (methods invoked, data passed).
- **States**: The different stages in the lifecycle of an object, often seen in state machine diagrams.
- **Relationships**: Different ways entities are connected, such as inheritance (generalization), association, or dependency.

Benefits of Using UML:

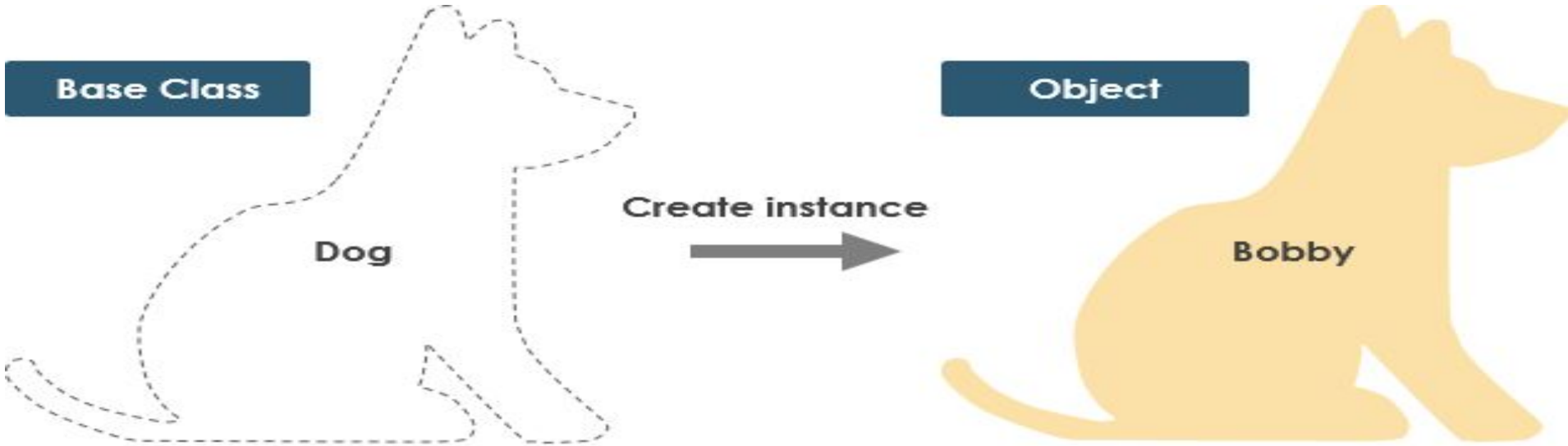
- **Standardized Communication**: UML provides a common visual language for developers, designers, and stakeholders, helping to avoid misunderstandings.
- **System Visualization**: UML diagrams allow complex systems to be broken down into manageable pieces, making it easier to understand and analyze system behavior and structure.
- **Flexibility**: UML can be applied across various domains, not just software engineering, to model any complex process or system.
- **Tool Support**: Many modeling tools support UML, allowing for automated generation of diagrams, code from models, and vice versa.

CLASS

- 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.

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

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

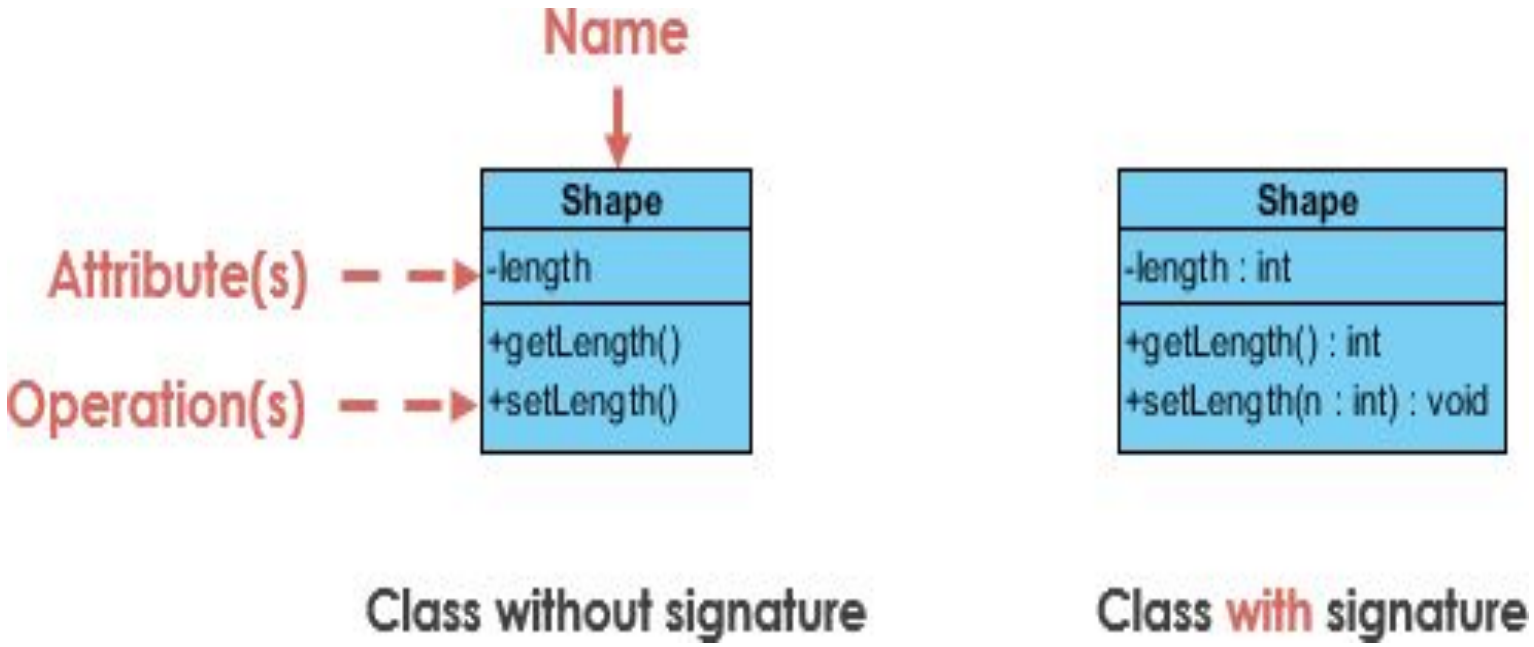


Properties	Methods
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

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

UML Class Notation :

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



Class Name:

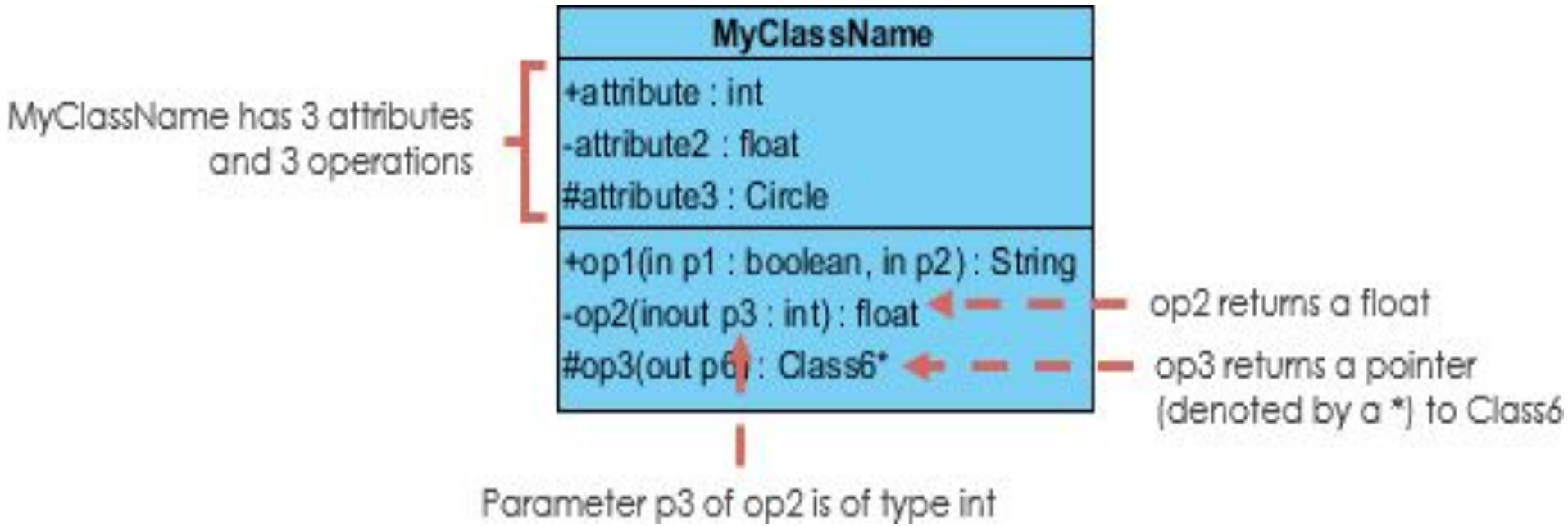
- The name of the class appears in the first partition.

Class Attributes:

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

Class Operations (Methods):

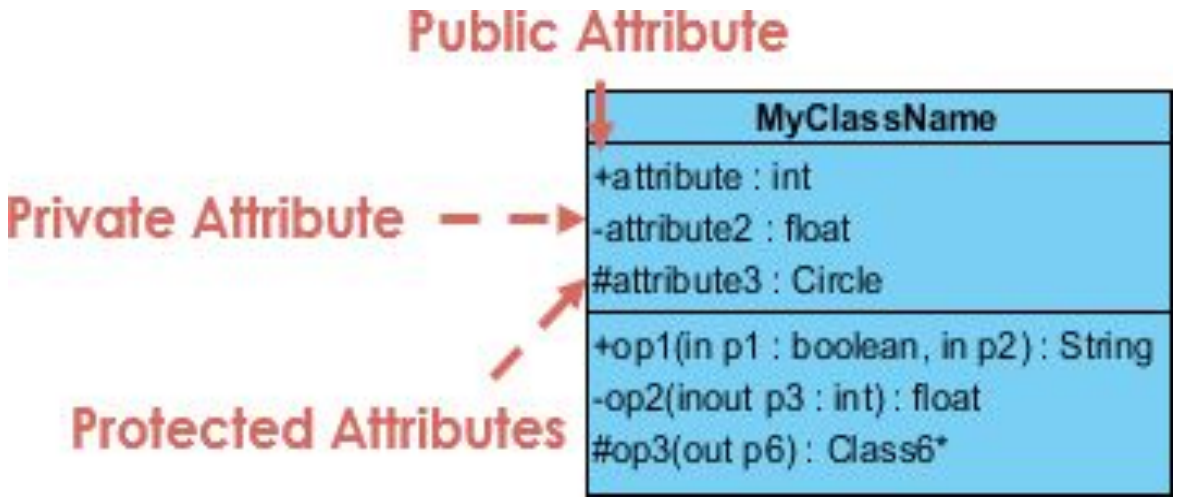
- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code


MyClassName
+attribute : int
-attribute2 : float
#attribute3 : Circle
+op1(in p1 : boolean, in p2) : String
-op2(inout p3 : int) : float
#op3(out p6) : Class6*
MyClassName has 3 attributes and 3 operations
op2 returns a float
op3 returns a pointer (denoted by a *) to Class6
Parameter p3 of op2 is of type int

Class Visibility :

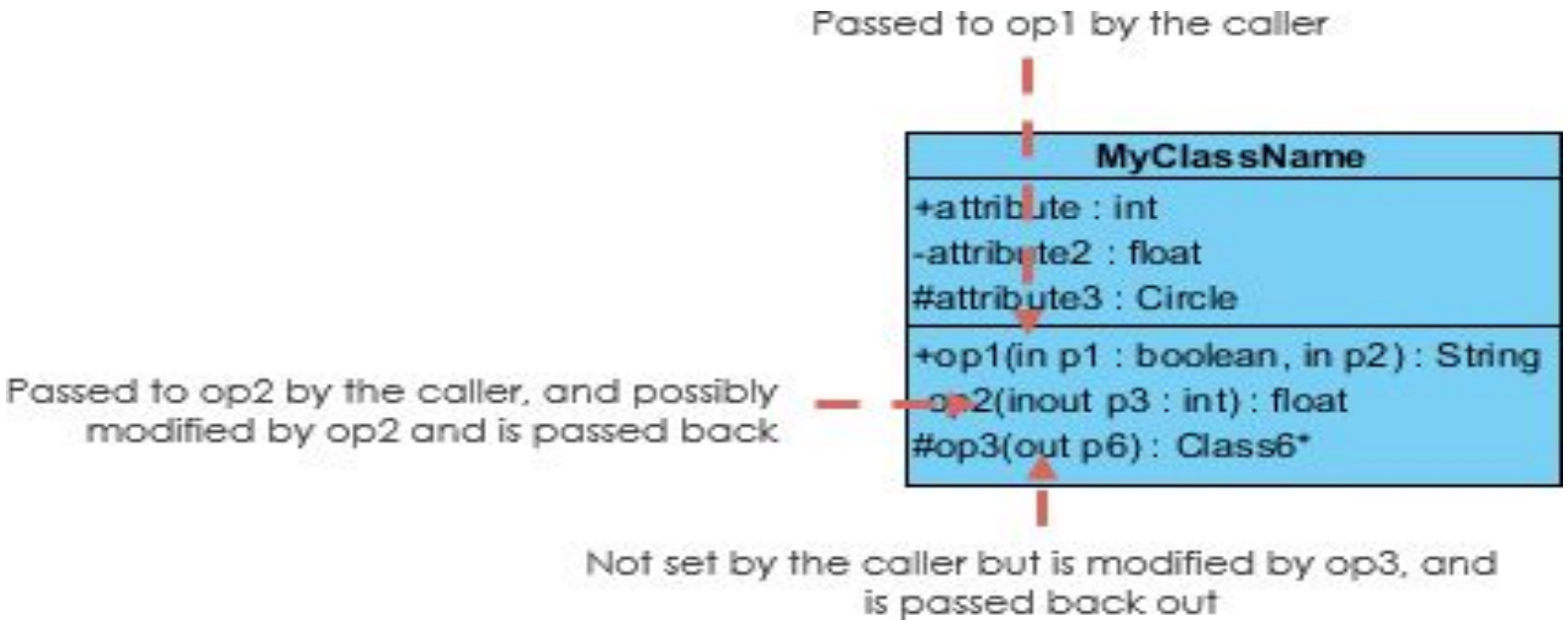
The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.

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



Parameter Directionality :

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

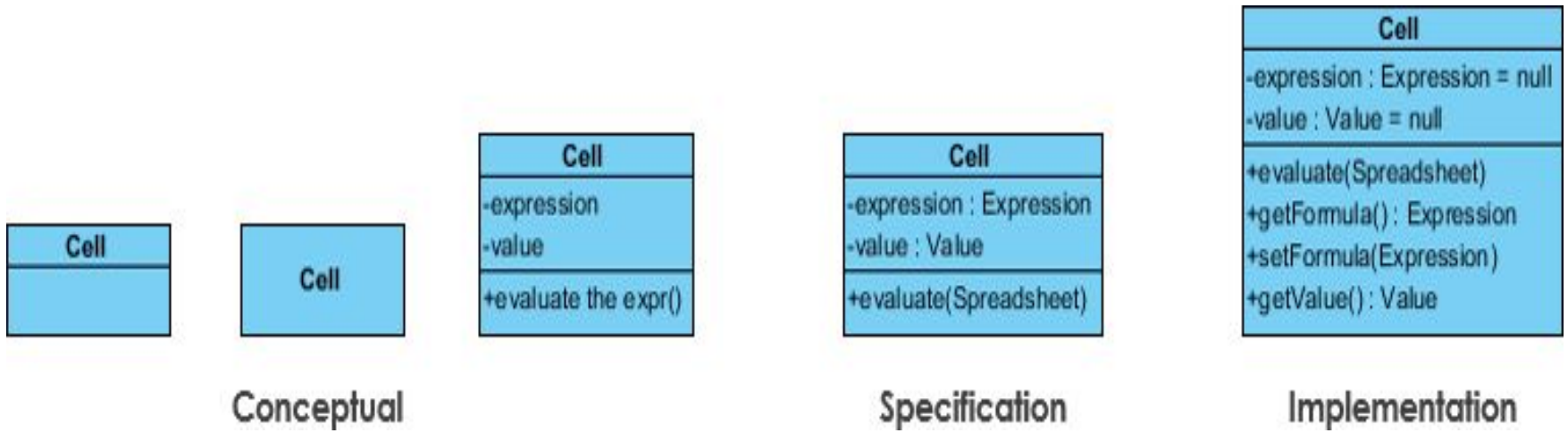


Perspectives of Class Diagram :

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

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

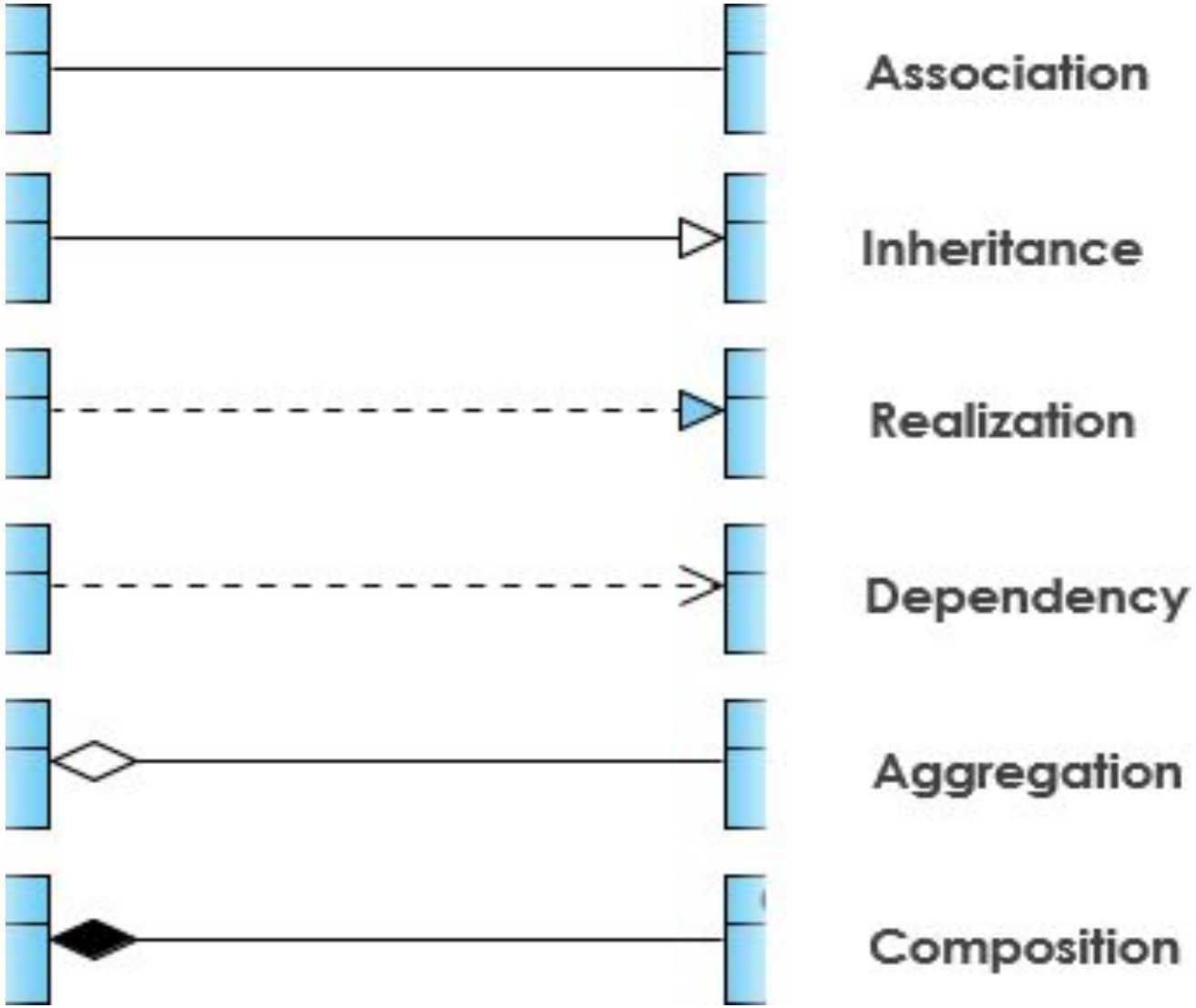
The perspective affects the amount of detail to be supplied and the kinds of relationships worth presenting. As we mentioned above, the class name is the only mandatory information.



Relationships between classes :

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

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

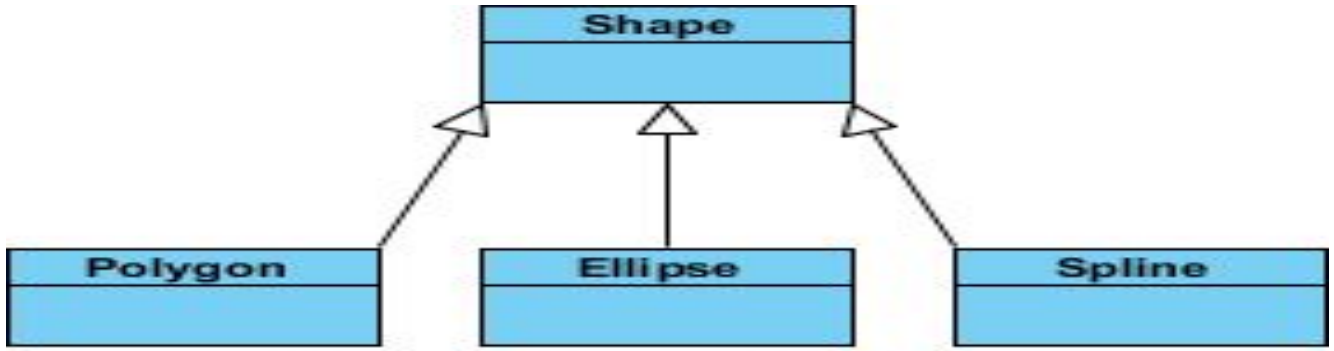

Association
Inheritance
Realization
Dependency
Aggregation
Composition

Inheritance (or Generalization):

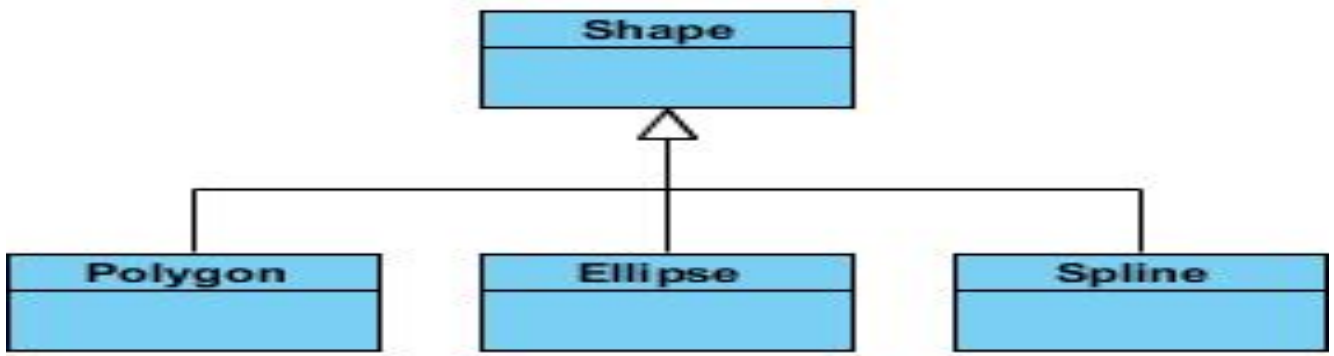
A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier.

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

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



Style 1: Separate target



Style 2: Shared target

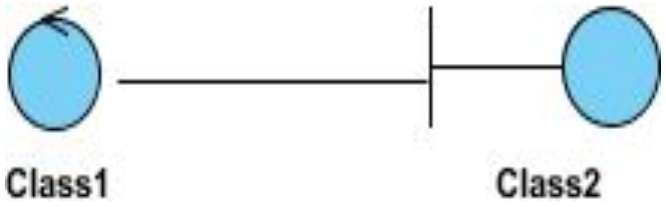
Association :

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

Simple Association

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

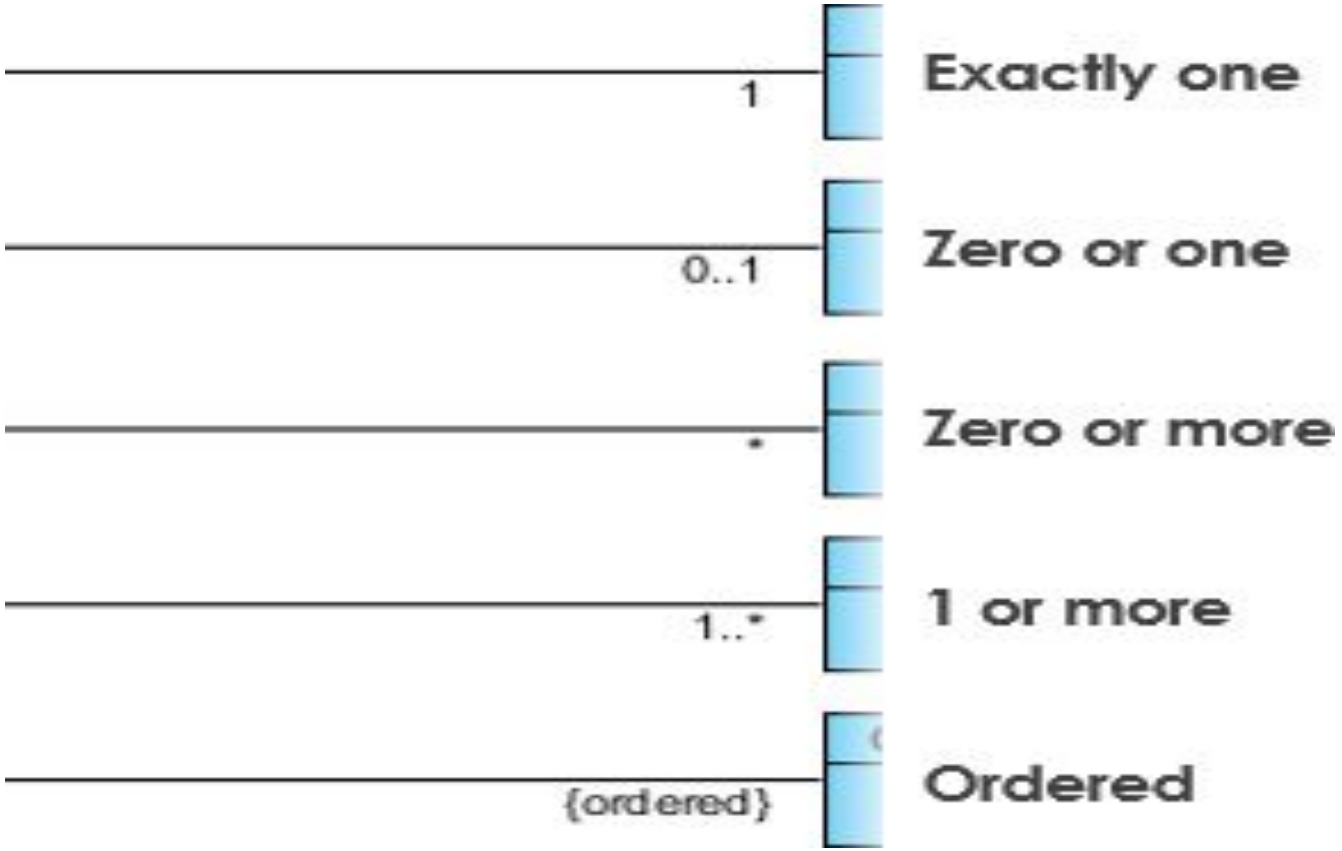
The figure below shows an example of simple association. There is an association that connects the <<control>> class Class1 and <<boundary>> class Class2. The relationship is displayed as a solid line connecting the two classes.



Cardinality :

Cardinality is expressed in terms of:

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

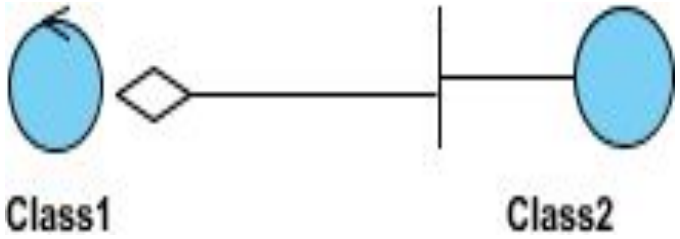


Aggregation :

A special type of association.

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

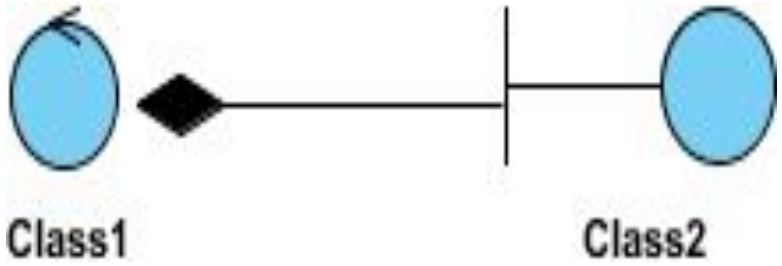
The figure below shows an example of aggregation. The relationship is displayed as a solid line with a unfilled diamond at the association end, which is connected to the class that represents the aggregate.



Composition :

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

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



Dependency :

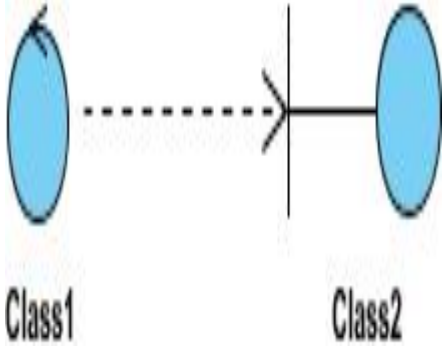
An object of one class might use an object of another class in the code of a method. If the object is not stored in any field, then this is modeled as a dependency relationship.

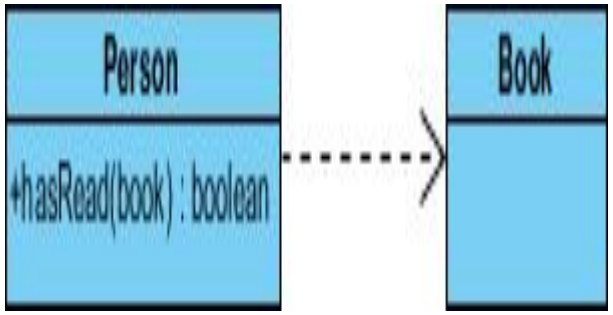
A special type of association.

Exists between two classes if changes to the definition of one may cause changes to the other (but not the other way around).

Class1 depends on Class2

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

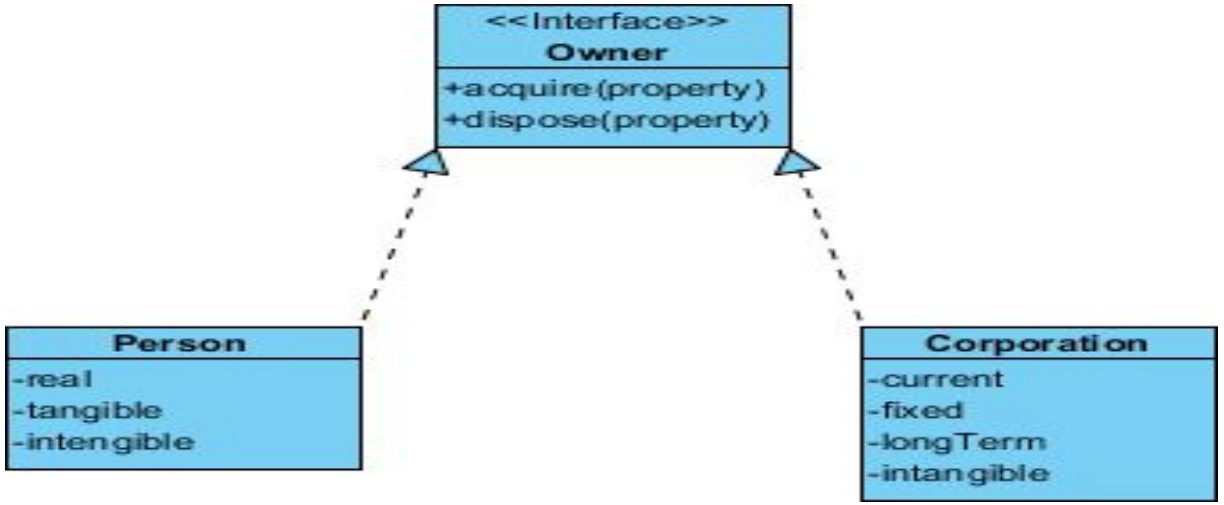


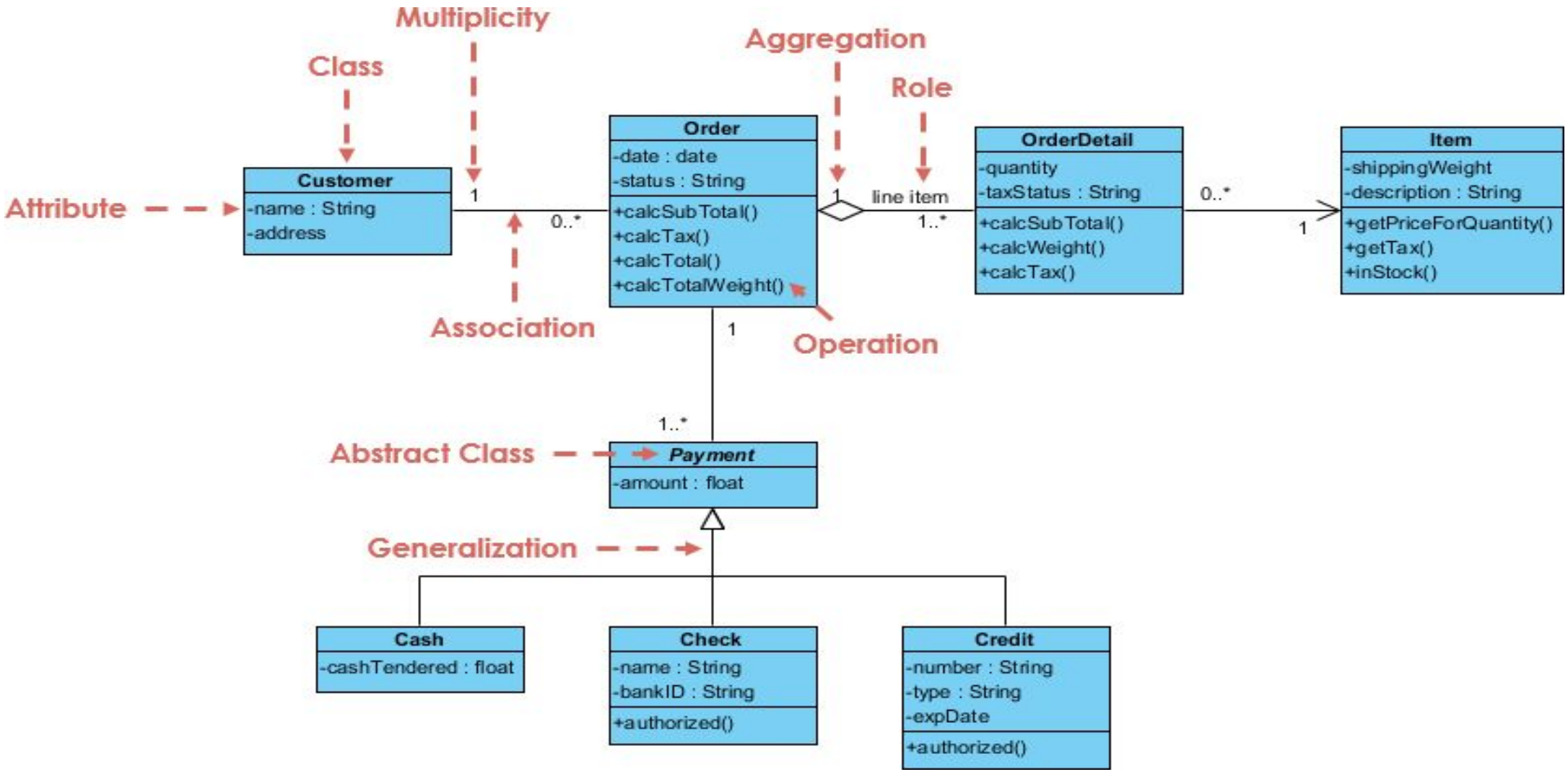


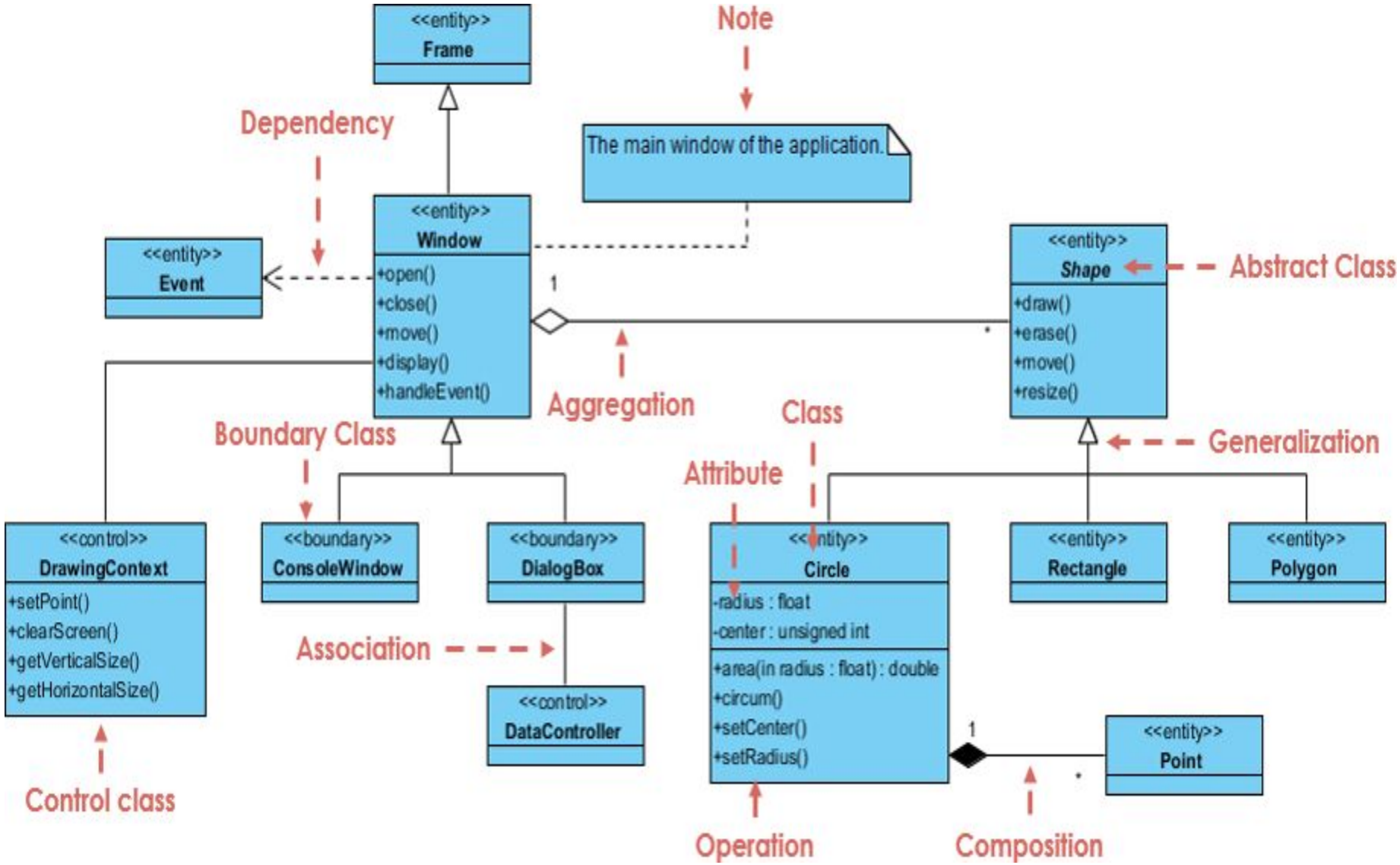
Realization :

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

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




Multiplicity
Class
Aggregation
Role
Attribute
Customer
-name : String
-address
1
0..*
Order
-date : date
-status : String
+calcSubTotal()
+calcTax()
+calcTotal()
+calcTotalWeight()
1
line item
1..*
OrderDetail
-quantity
-taxStatus : String
+calcSubTotal()
+calcWeight()
+calcTax()
0..*
1
Item
-shippingWeight
-description : String
+getPriceForQuantity()
+getTax()
+inStock()
Association
Operation
1
1..*
Abstract Class
Payment
-amount : float
Generalization
Cash
-cashTendered : float
Check
-name : String
-bankID : String
+authorized()
Credit
-number : String
-type : String
-expDate
+authorized()

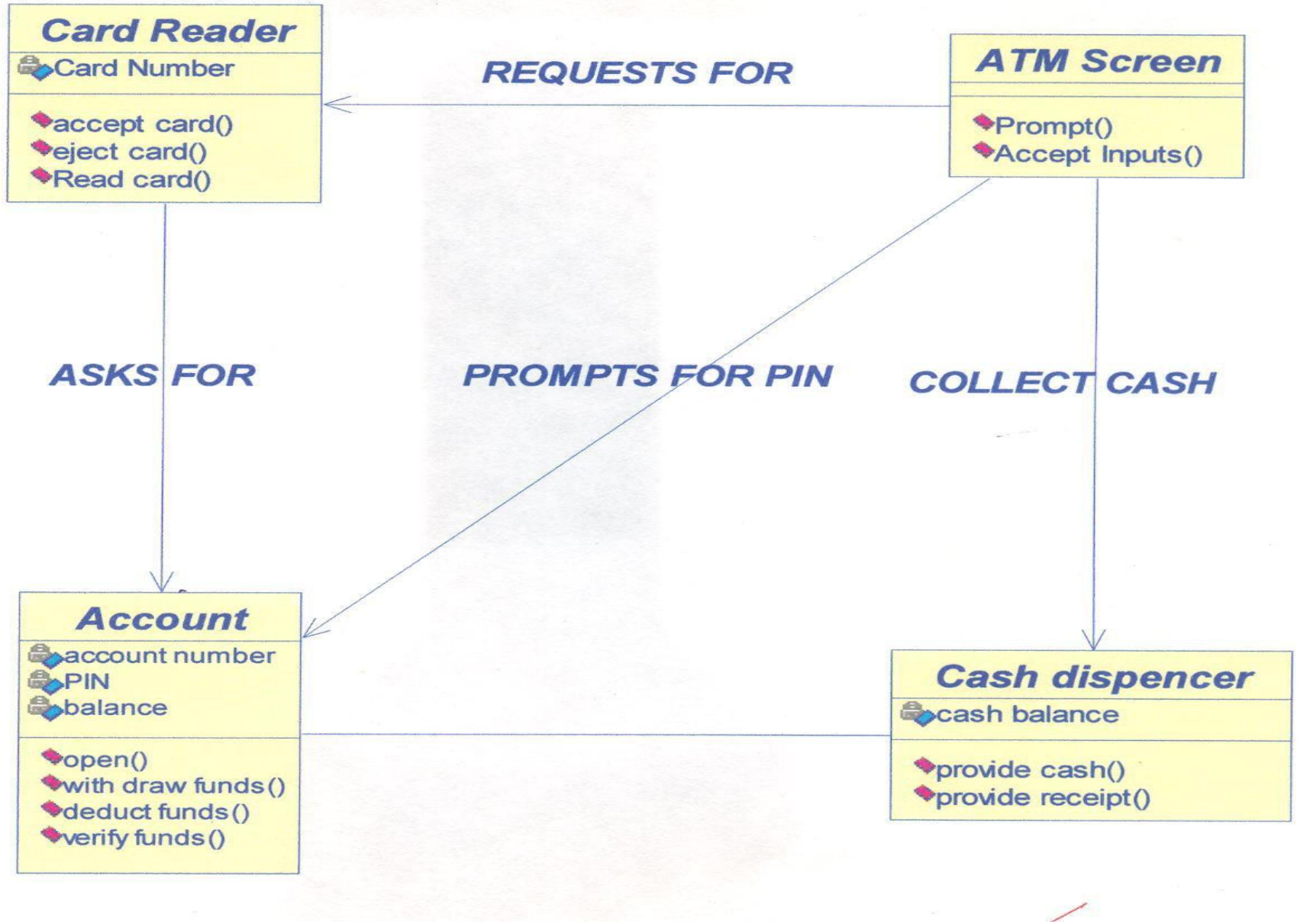

Note
<<entity>>
Frame
Dependency
The main window of the application.
<<entity>>
Window
+open()
+close()
+move()
+display()
+handleEvent()
<<entity>>
Event
1
*
<<entity>>
Shape
+draw()
+erase()
+move()
+resize()
Abstract Class
Aggregation
Boundary Class
Class
Attribute
Generalization
<<control>>
DrawingContext
+setPoint()
+clearScreen()
+getVerticalSize()
+getHorizontalSize()
<<boundary>>
ConsoleWindow
<<boundary>>
DialogBox
<<entity>>
Circle
-radius : float
-center : unsigned int
+area(in radius : float) : double
+circum()
+setCenter()
+setRadius()
<<entity>>
Rectangle
<<entity>>
Polygon
Association
<<control>>
DataController
1
*
<<entity>>
Point
Control class
Operation
Composition

Task1: Draw Class Diagram for ATM System.

Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, associations, and collaboration. Class diagrams basically represent the object-oriented view of a system, which is static in nature.

Active class is used in a class diagram to represent the concurrency of the system.

Class diagram represents the object orientation of a system. Hence, it is generally used for development purpose. This is the most widely used diagram at the time of system construction.

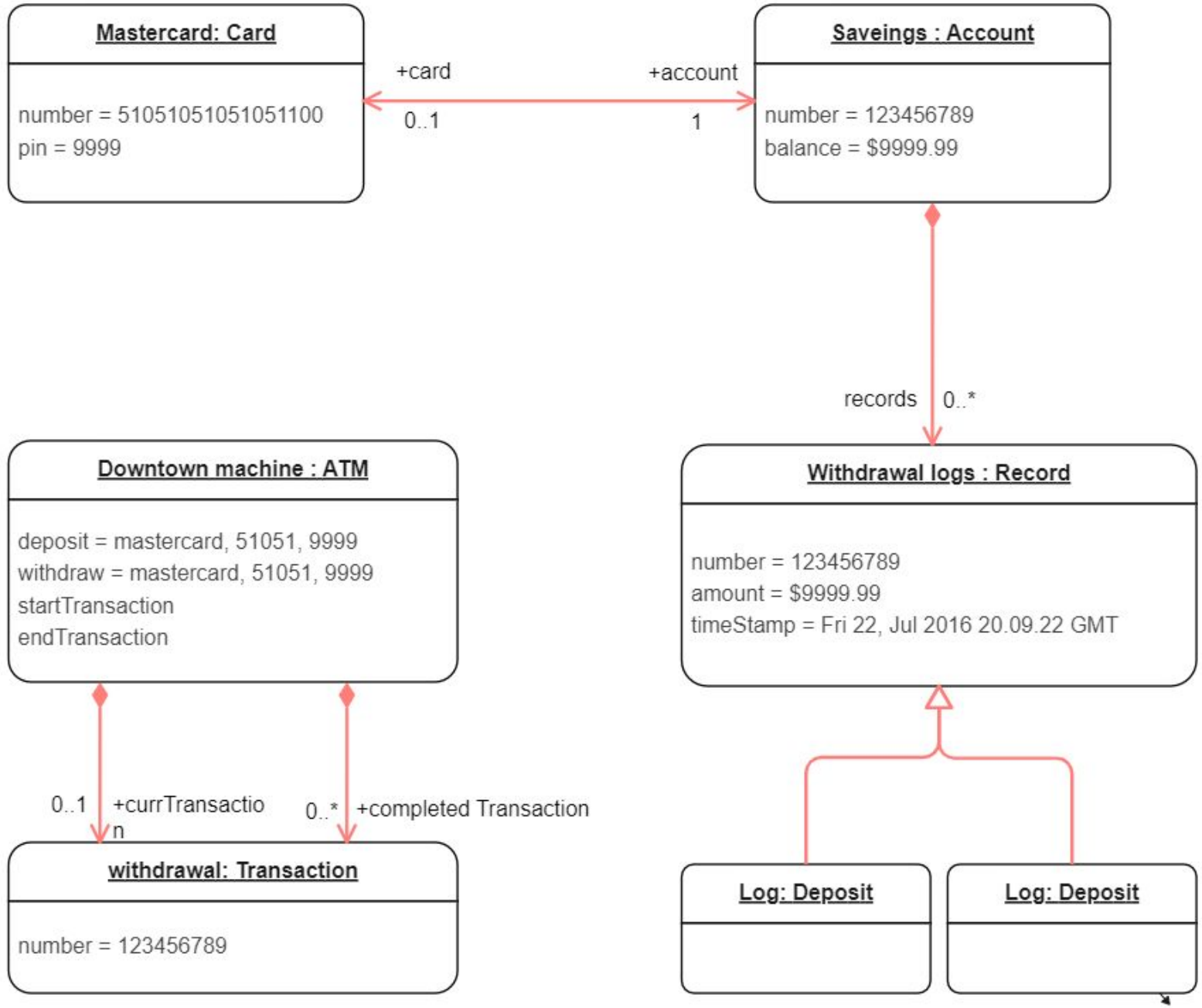

Card Reader
Card Number
accept card()
eject card()
Read card()
REQUESTS FOR
ATM Screen
Prompt()
Accept Inputs()
ASKS FOR
PROMPTS FOR PIN
COLLECT CASH
Account
account number
PIN
balance
open()
with draw funds()
deduct funds()
verify funds()
Cash dispencer
cash balance
provide cash()
provide receipt()

Task2: Draw Object Diagram for ATM System.

Object diagrams can be described as an instance of class diagram. Thus, these diagrams are more close to real-life scenarios where we implement a system.

Object diagrams are a set of objects and their relationship is just like class diagrams. They also represent the static view of the system.

The usage of object diagrams is similar to class diagrams but they are used to build prototype of a system from a practical perspective.


Mastercard: Card
number = 51051051051051100
pin = 9999
+card
0..1
+account
1
Saveings : Account
number = 123456789
balance = $9999.99
records
0..*
Downtown machine : ATM
deposit = mastercard, 51051, 9999
withdraw = mastercard, 51051, 9999
startTransaction
endTransaction
Withdrawal logs : Record
number = 123456789
amount = $9999.99
timeStamp = Fri 22, Jul 2016 20.09.22 GMT
0..1
+currTransactio
n
0..*
+completed Transaction
withdrawal: Transaction
number = 123456789
Log: Deposit
Log: Deposit

TASK 3:Draw Use Case Diagram for ATM System.

A **Use Case Diagram** visually represents the functionality of an ATM system and the interactions between users and the system. Here's how the use case diagram for an ATM system typically looks:

Actors:

1. **Customer**: A person using the ATM.
2. **Bank Server**: The back-end system that verifies transactions and account details.
3. **Technician**: A person who maintains or services the ATM.

Use Cases:

1. **Authenticate User**: The system verifies the user's credentials (Card + PIN).
2. **Check Balance**: Allows the customer to view their account balance.
3. **Withdraw Cash**: The customer can withdraw a specified amount of money.
4. **Deposit Cash/Cheque**: The customer deposits cash or a cheque into their account.

5.**Transfer Funds**: The customer transfers money between their own accounts or to other accounts

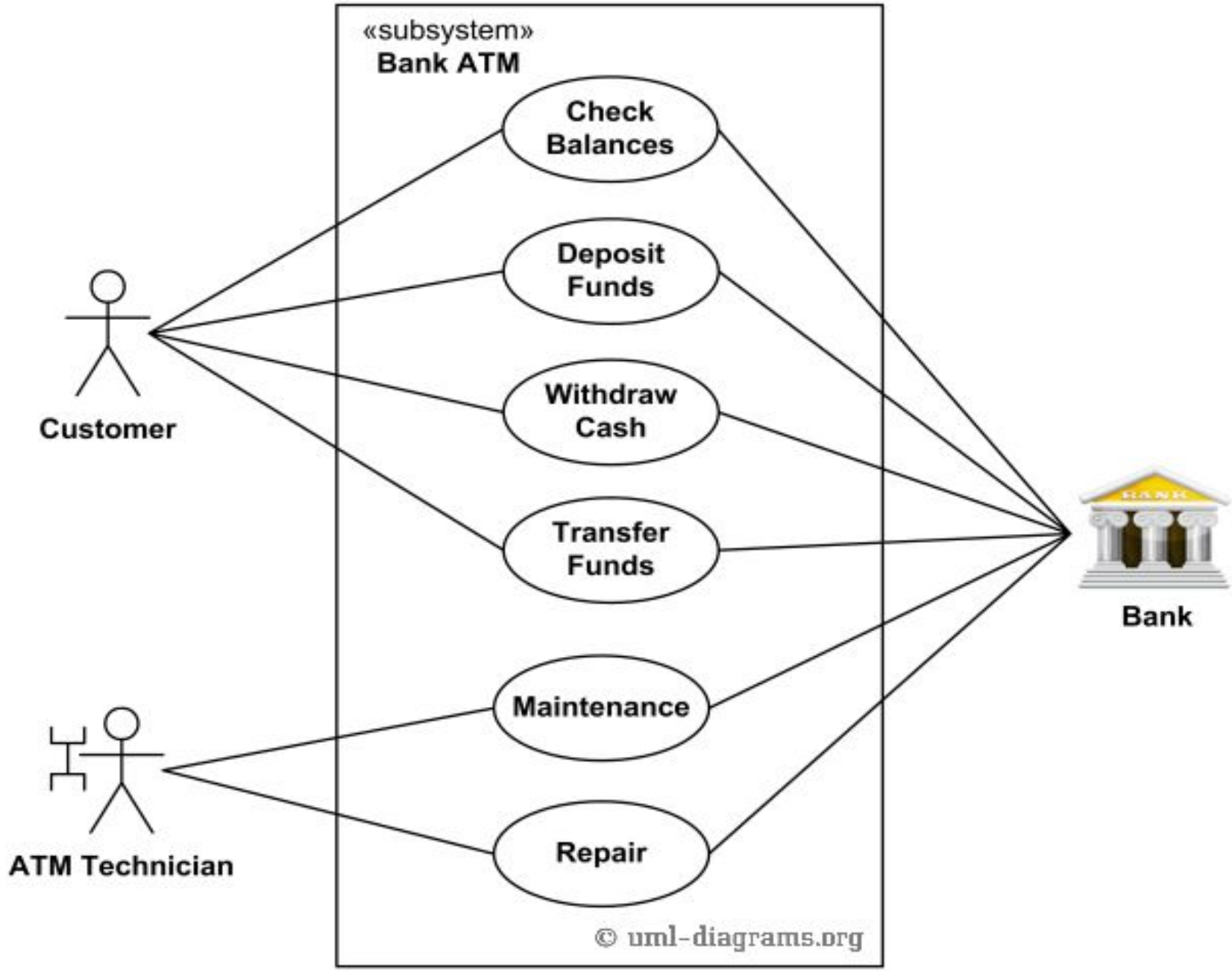
6.**Change PIN**: The customer can change their account PIN through the ATM.

7.**Print Receipt**: The customer receives a receipt after completing a transaction.

8.**Maintain ATM**: A technician can service the ATM and refill cash.

Relationships:

- **Customer** interacts with use cases like "Authenticate User", "Check Balance", "Withdraw Cash", "Deposit Cash", "Transfer Funds", "Change PIN", and "Print Receipt".
- **Technician** interacts with "Maintain ATM".
- **Bank Server** supports all customer transactions by providing back-end validation.


«subsystem»
Bank ATM
Check Balances
Deposit Funds
Withdraw Cash
Transfer Funds
Maintenance
Repair
Customer
ATM Technician
Bank
© uml-diagrams.org

Task4: Draw Sequence Diagram for ATM System.

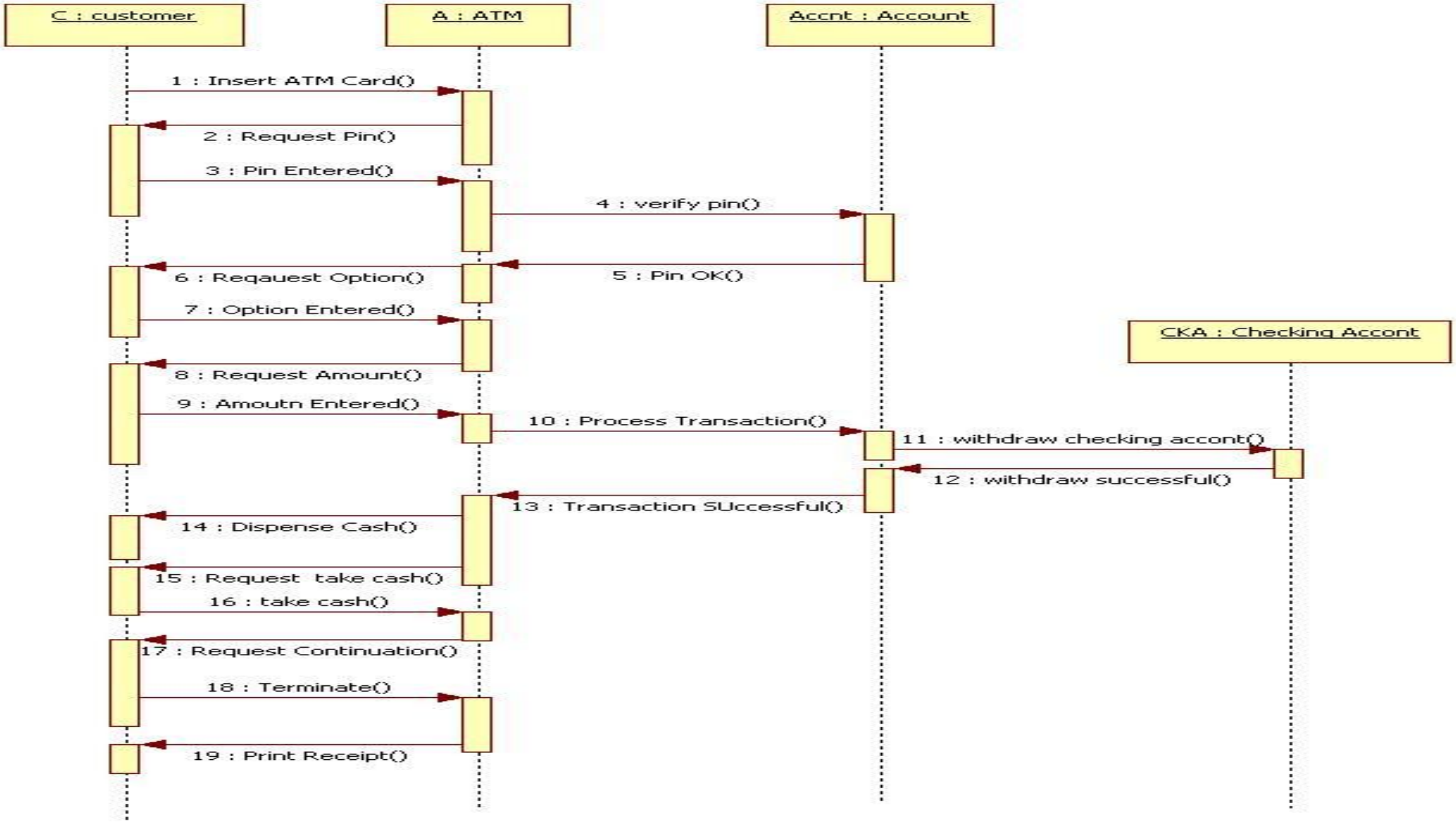
A **Sequence Diagram** shows how objects interact in a particular scenario of the system. Here's how a sequence diagram for an ATM system typically works when a customer performs a cash withdrawal.

Key Objects:

1. **Customer**: Initiates the interaction.
2. **ATM Machine**: Provides the interface for customer interaction.
3. **Bank Server**: Validates customer credentials and processes the transaction.
4. **Account**: Holds the customer's balance and handles fund transactions.

Scenario: Cash Withdrawal

1. **Customer** inserts card into the **ATM Machine**.
2. **ATM Machine** prompts the customer to enter the PIN.
3. **Customer** enters the PIN.
4. **ATM Machine** sends authentication request to the **Bank Server**.
5. **Bank Server** validates the credentials and confirms with the **ATM Machine**.
6. **ATM Machine** prompts the customer to select a transaction (withdraw cash).
7. **Customer** selects withdrawal and enters the amount.
8. **ATM Machine** sends the withdrawal request to the **Bank Server**.
9. **Bank Server** checks the balance in the **Account** and approves the transaction if funds are available.
10. **ATM Machine** dispenses cash to the **Customer**.
11. **ATM Machine** prints a receipt.
12. **ATM Machine** updates the transaction in the **Bank Server**.


C : customer
A : ATM
Accnt : Account
CKA : Checking Accont
1 : Insert ATM Card()
2 : Request Pin()
3 : Pin Entered()
4 : verify pin()
5 : Pin OK()
6 : Reqauest Option()
7 : Option Entered()
8 : Request Amount()
9 : Amoutn Entered()
10 : Process Transaction()
11 : withdraw checking accont()
12 : withdraw successful()
13 : Transaction SUccessful()
14 : Dispense Cash()
15 : Request take cash()
16 : take cash()
17 : Request Continuation()
18 : Terminate()
19 : Print Receipt()

Task5: Draw Collaboration Diagram for ATM System

A collaboration diagram for an ATM system typically illustrates the interactions between various objects (or entities) to achieve certain functionality like cash withdrawal, balance inquiry, or deposit. It represents the relationship between objects, showing how they collaborate with each other to complete a process. Below is a description of how the entities might interact in an ATM system:

Main Entities:

1. **Customer**: The user interacting with the ATM.
2. **ATM**: The machine providing banking services.
3. **Card Reader**: A subsystem of the ATM that reads the bank card.
4. **Bank Server**: The backend system that holds customer account information and processes requests.
5. **Cash Dispenser**: The component that releases cash to the customer.
6. **Receipt Printer**: Prints receipts after transactions.

Difference between sequence and collaboration diagrams :

Sequence and collaboration diagrams are both used in **UML (Unified Modeling Language)** to represent interactions between objects, but they do so in different ways. Here's a comparison between the two:

1. Purpose:

- **Sequence Diagram**: Focuses on the **order** of interactions between objects over time. It shows how processes interact through sequences of events.
- **Collaboration Diagram**: Focuses on the **relationships and structure** of objects, showing how they work together to achieve a particular task.

2. Representation:

- **Sequence Diagram**:
  - Represents the **time-based flow** of messages between objects.
  - The objects (participants) are placed at the top of the diagram, and interactions (messages) are represented vertically along a timeline.
  - The sequence in which messages are exchanged is shown from top to bottom.

•**Collaboration Diagram**:
  - Represents the **structural organization** of objects that interact.
  - Emphasizes the roles of objects and the relationships between them.
  - The messages are shown alongside lines that connect objects, without explicit focus on time ordering.

3. Focus:

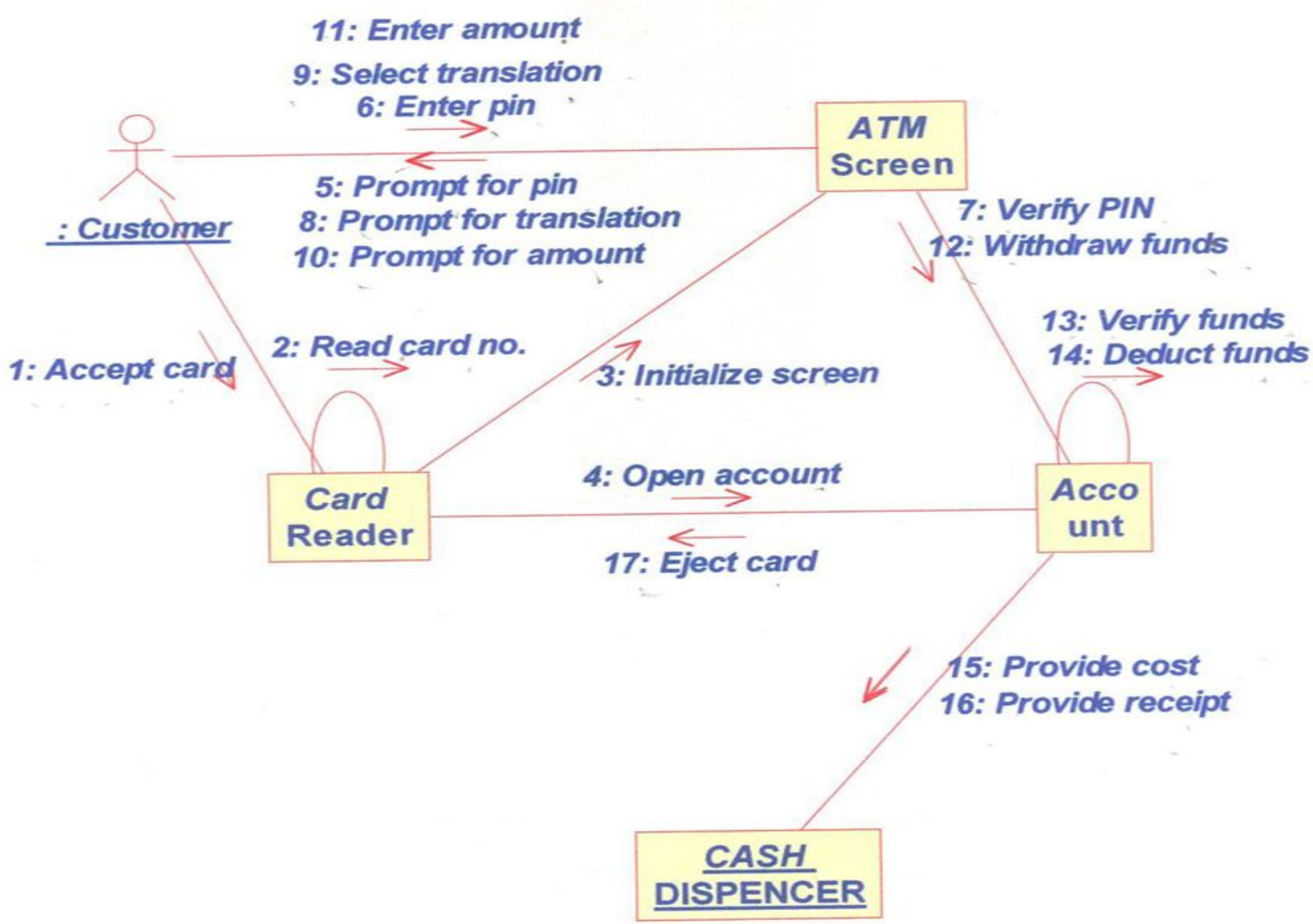
•**Sequence Diagram**: Focuses on **message sequence** and the **timing** of interactions.

•**Collaboration Diagram**: Focuses on how objects are **related** to one another and the communication that occurs among them.

4. Visualization:

•**Sequence Diagram**:
  - Messages are listed in order vertically.
  - You can see **lifelines** (vertical dashed lines) that indicate the object's lifespan during the interaction.
  - Arrows between objects indicate the flow of messages.

- **Collaboration Diagram**:
  - Objects are arranged arbitrarily, with lines between them showing associations.
  - The sequence of messages is usually indicated by numbering (e.g., 1, 2, 3, etc.) next to the messages on the lines between the objects


11: Enter amount
9: Select translation
6: Enter pin
ATM Screen
: Customer
5: Prompt for pin
8: Prompt for translation
10: Prompt for amount
7: Verify PIN
12: Withdraw funds
13: Verify funds
14: Deduct funds
1: Accept card
2: Read card no.
3: Initialize screen
Card Reader
4: Open account
Acco unt
17: Eject card
15: Provide cost
16: Provide receipt
CASH DISPENCER

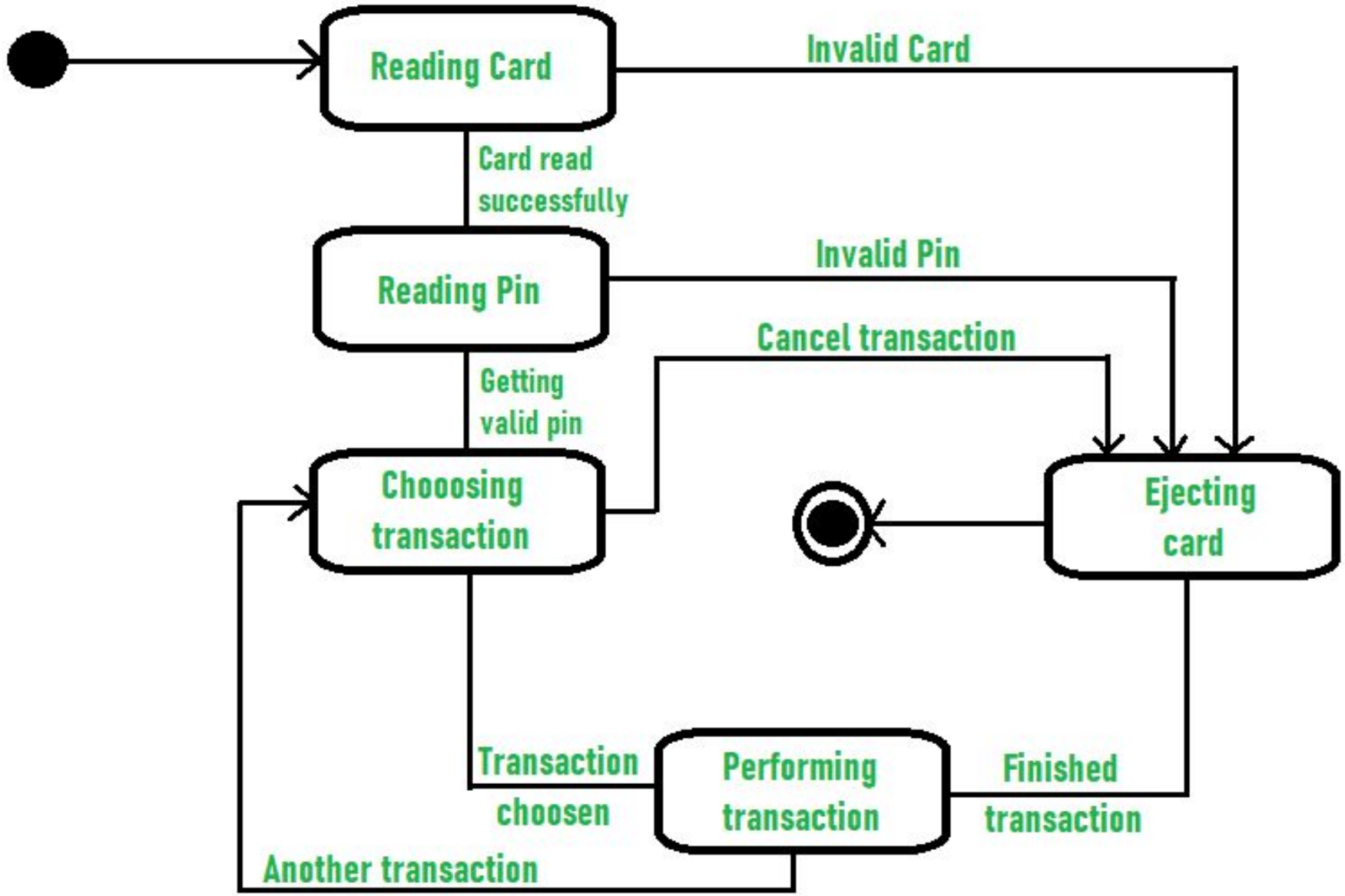
Task6: Draw State Chart Diagram for ATM System.

- Any real-time system is expected to be reacted by some kind of internal/external events. These events are responsible for state change of the system.
- Statechart diagram is used to represent the event driven state change of a system. It basically describes the state change of a class, interface, etc.

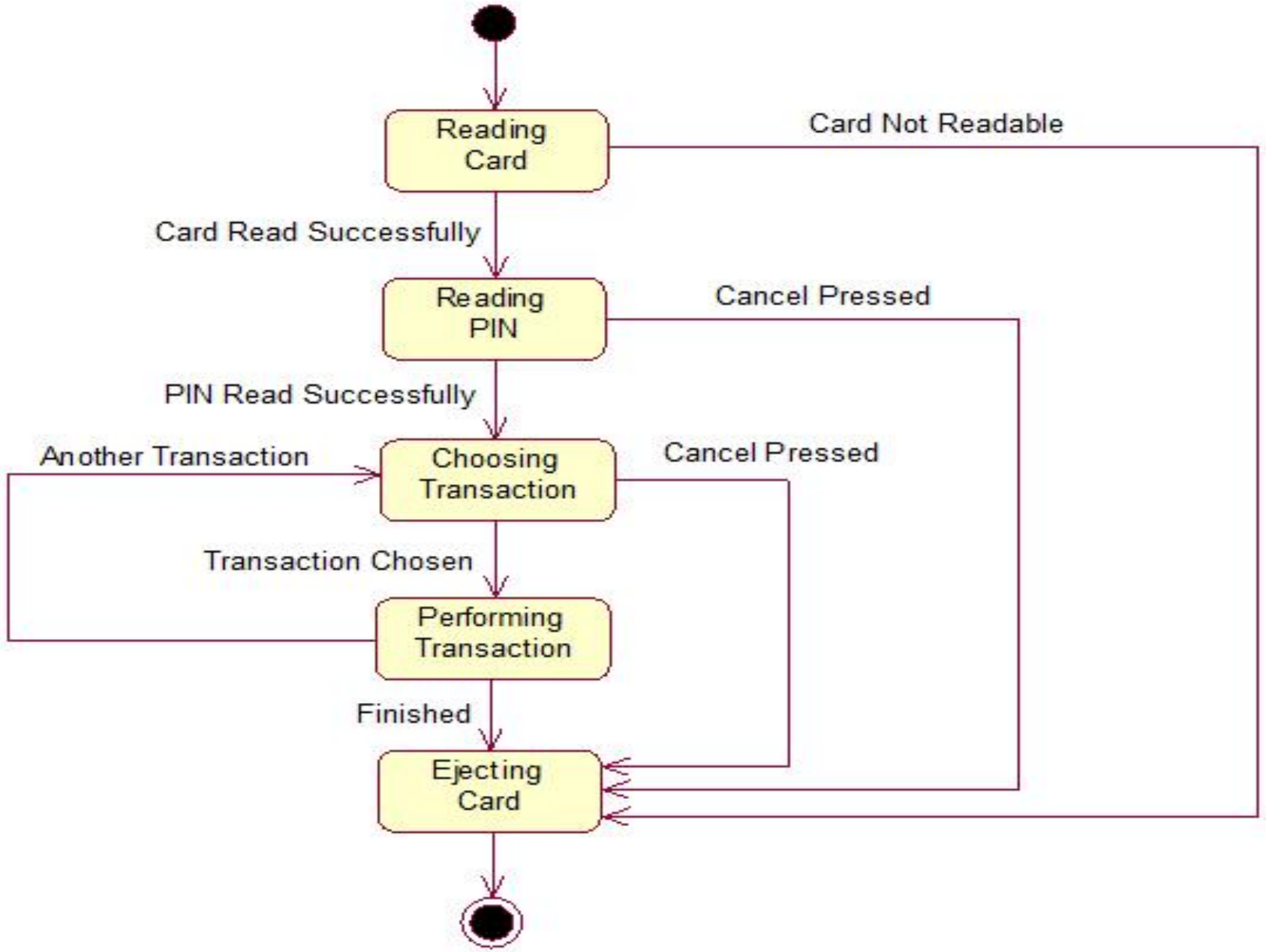
- State chart diagram is used to visualize the reaction of a system by internal/external factors.
- **State Chart Diagram (State Machine Diagram):**
- **Focuses on states and transitions of an object** during its lifecycle.
- It shows how an object changes from one state to another in response to events or conditions.
- It is used to model **reactive systems** that respond to internal or external events, where the object's state changes over time.

Elements:

- **State Chart Diagram:**
  - **States:** Represent the conditions or situations in which an object exists (e.g., idle, processing, completed).
  - **Transitions:** Represent the change from one state to another triggered by an event or condition.
  - **Events:** The triggers that cause transitions between states.
  - **Actions/Activities:** Can occur on transitions or within states.



State Transition Diagram for ATM System


Reading Card
Card Not Readable
Card Read Successfully
Reading PIN
Cancel Pressed
PIN Read Successfully
Another Transaction
Choosing Transaction
Cancel Pressed
Transaction Chosen
Performing Transaction
Finished
Ejecting Card

Task7: Draw Activity Diagram for ATM System.

Activity Diagram:

- **Focuses on the flow of activities or actions** that happen within a system or process.
- It models the sequence of tasks, decision points, and parallel activities.
- It is often used to describe **business processes, workflows**, or the logic of complex operations within the system.

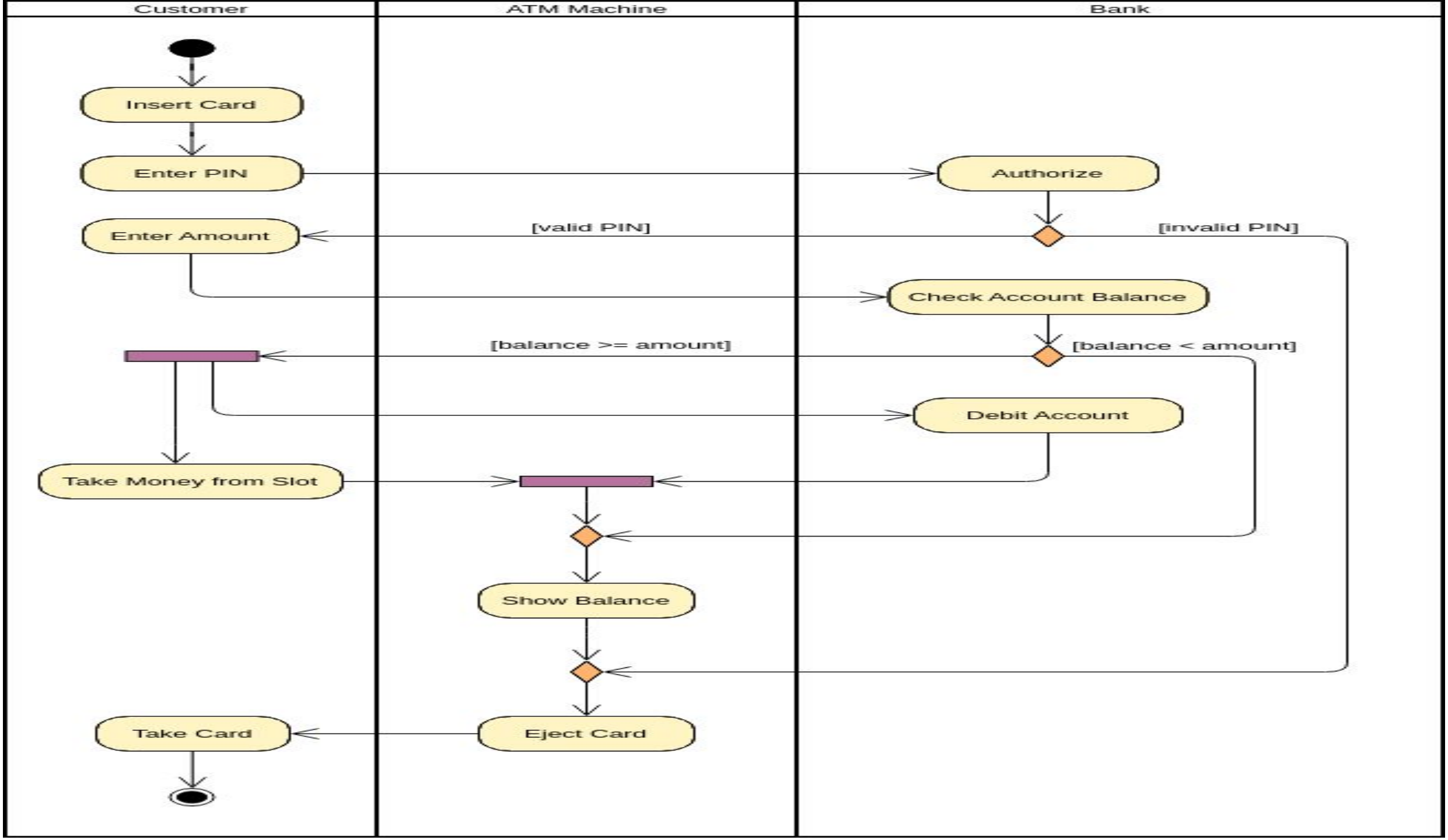
Elements:

Activities/Actions: Represent tasks, operations, or steps in a process.

Control Flows: Arrows indicating the order in which activities are executed.

Decision Points: Represent conditions where different paths are taken depending on a condition (e.g., yes/no).

Forks/Joins: Used to show parallel processes (fork) or synchronization (join).


Customer
ATM Machine
Bank
Insert Card
Enter PIN
Authorize
Enter Amount
[valid PIN]
[invalid PIN]
Check Account Balance
[balance >= amount]
[balance < amount]
Debit Account
Take Money from Slot
Show Balance
Take Card
Eject Card

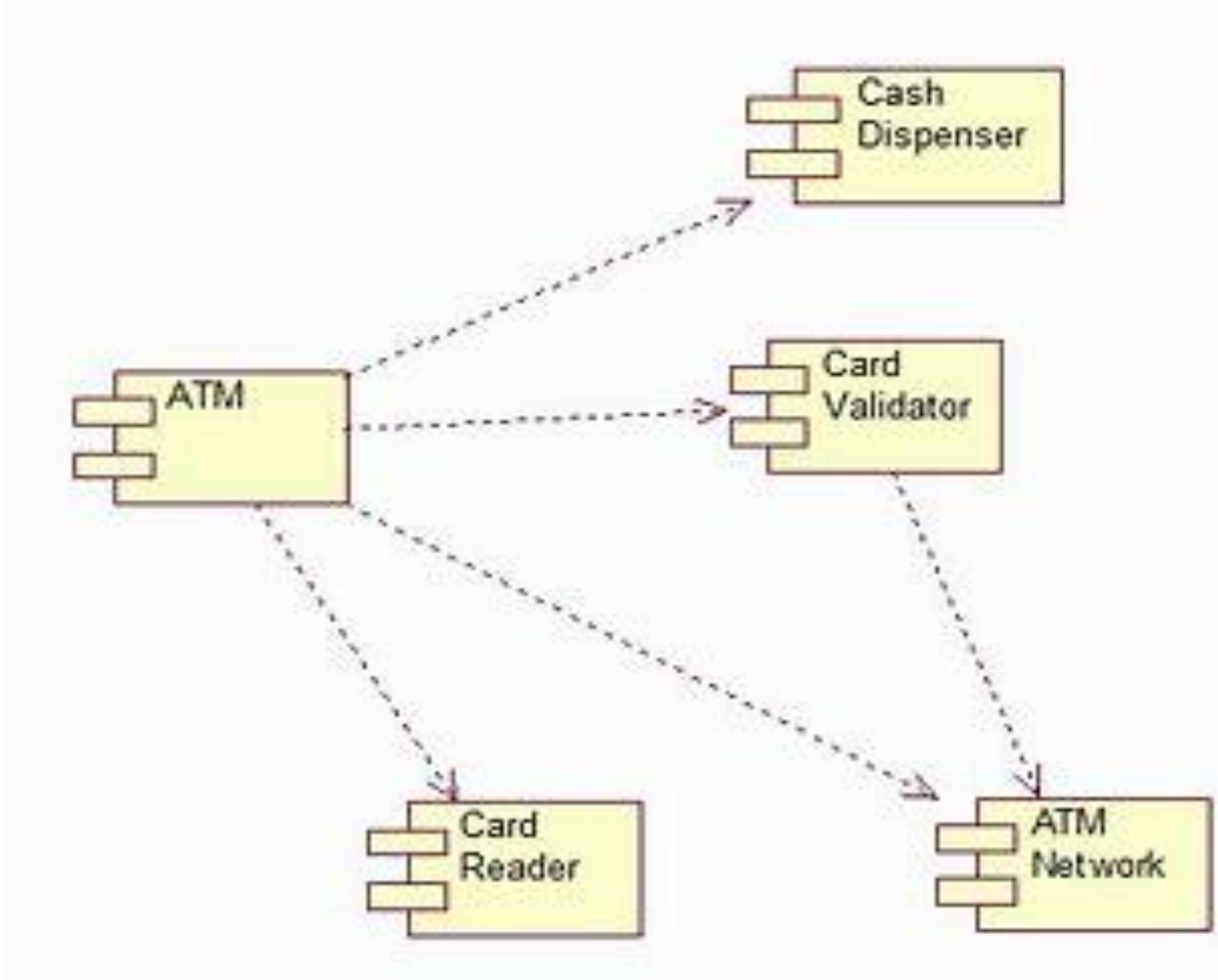
Task8: Draw Component Diagram for ATM System.

A **component diagram** is another UML (Unified Modeling Language) diagram type that models the structure of a system by showing its software components and their relationships. Component diagrams are ideal for representing the physical aspects of a system, including executables, libraries, and other components used in development.

Key Elements of a Component Diagram:

1. **Components**: Represent individual units within the system, like modules, executables, or libraries. Components are depicted as rectangles with the component icon (a smaller rectangle) at the top right.
2. **Interfaces**: Represent the services or operations provided and required by components. Interfaces are typically shown as circles (provided interfaces) or half circles (required interfaces).

1. **Relationships**:
    1. **Dependency**: Shows a relationship where one component depends on another.
    2. **Association**: Shows a connection or interaction between components.
    3. **Assembly Connector**: Connects required interfaces to provided interfaces, showing how components work together.


Cash
Dispenser
ATM
Card
Validator
Card
Reader
ATM
Network

Task9: Draw Deployment Diagram for ATM System.

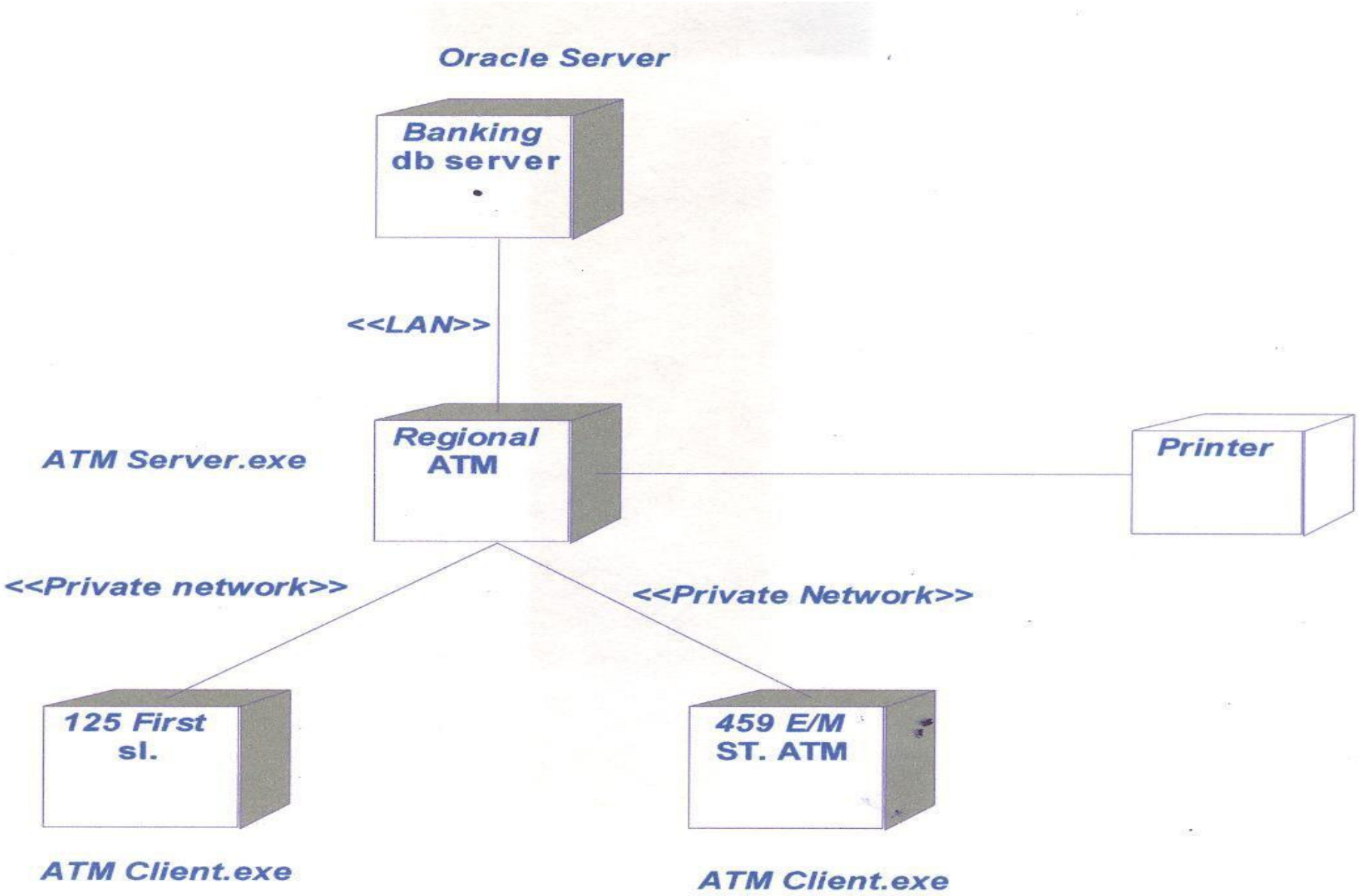
A **deployment diagram** is part of the Unified Modeling Language (UML) that models the physical deployment of artifacts on nodes in a system. It shows the hardware and software components in a system and their relationships, illustrating how software components are deployed across hardware nodes. Deployment diagrams are helpful in understanding the system's infrastructure, especially in distributed or cloud-based environments.

Key Components of a Deployment Diagram:

1. **Nodes**: Represent physical hardware or computational resources where the software will be deployed. This could be servers, virtual machines, devices, or cloud instances.
2. **Artifacts**: Represent software elements, such as applications, databases, or modules, that are deployed on nodes.

3.Relationships:

1. **Deployment Relationships**: Show where an artifact is deployed on a node.
2. **Communication Paths**: Represent connections or networks between nodes.


Oracle Server
Banking
db server
<<LAN>>
ATM Server.exe
Regional
ATM
Printer
<<Private network>>
<<Private Network>>
125 First
sl.
459 E/M
ST. ATM
ATM Client.exe
ATM Client.exe

Task10: Draw UML Behavioural diagrams for Remote Procedure Call Implementation.

- RPC stands for **Remote Procedure Call**, a protocol that allows programs to execute procedures (functions) on a remote server as if they were local.
- RPC abstracts the complexities of network communication, providing a simple interface for distributed computing.
- Remote Procedure Call or RPC is a powerful technique for constructing distributed, client-server-based applications.

Request flow:

- Client Call: The client invokes a procedure call as if it were local.
- Stub Generation: The client-side stub converts the procedure call into a network request (marshaling the arguments into a suitable format, such as binary, JSON, or XML).
- Transport: The request is sent over the network to the server.
- Server Stub: The server-side stub receives the request, unmarshals the data, and calls the corresponding local function.

- Response: The result of the function is serialized by the server stub and sent back to the client.
- Deserialization: The client stub deserializes the response and passes the result to the client application

Error Handling:

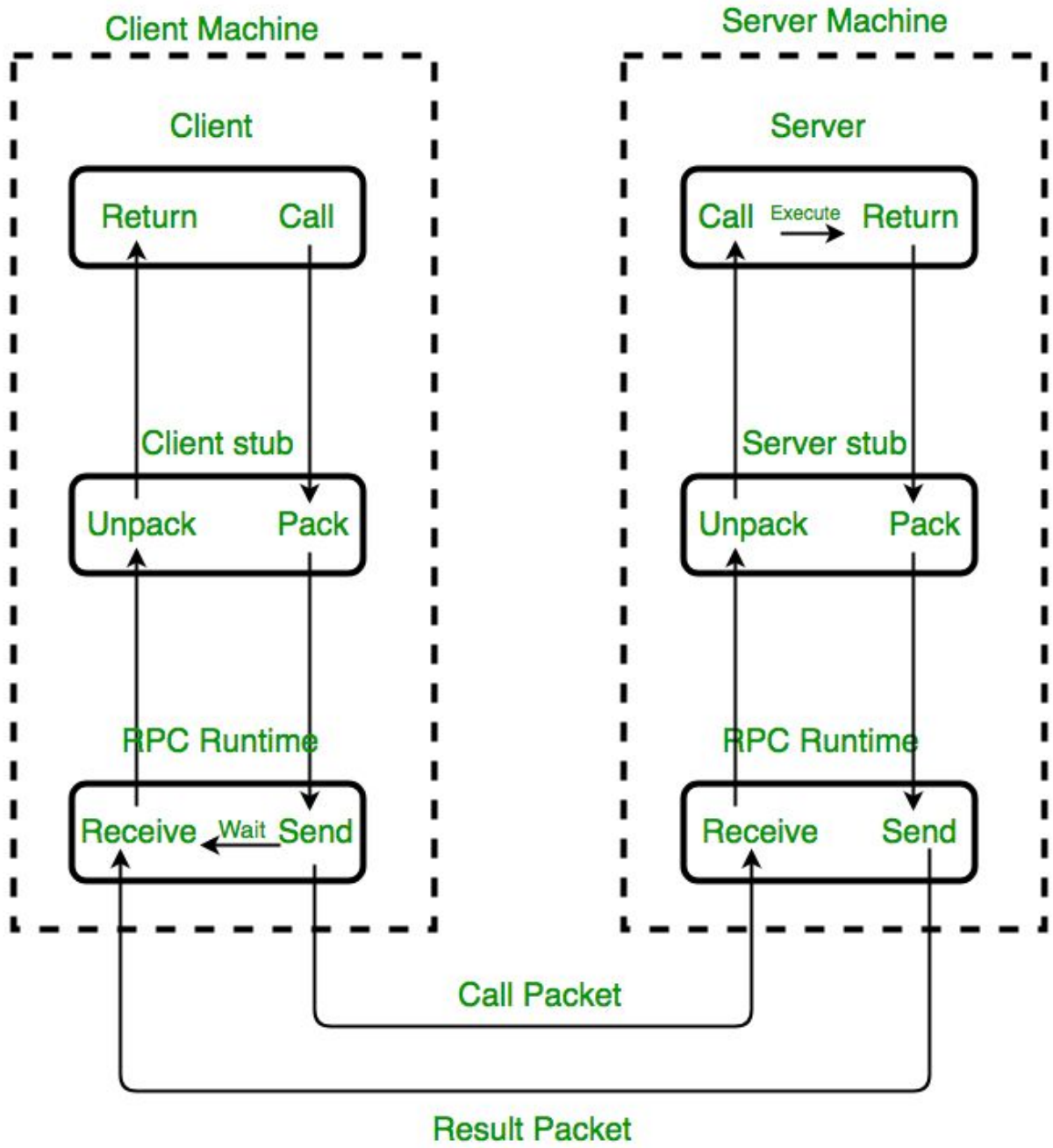
1. RPC handles failures, such as timeouts, network errors, or server crashes, by providing error codes or exceptions. These must be managed explicitly by developers.

Synchronous vs. Asynchronous:

2. **Synchronous RPC**: The client waits (blocks) until the server responds.
3. **Asynchronous RPC**: The client continues execution and processes the server's response later.

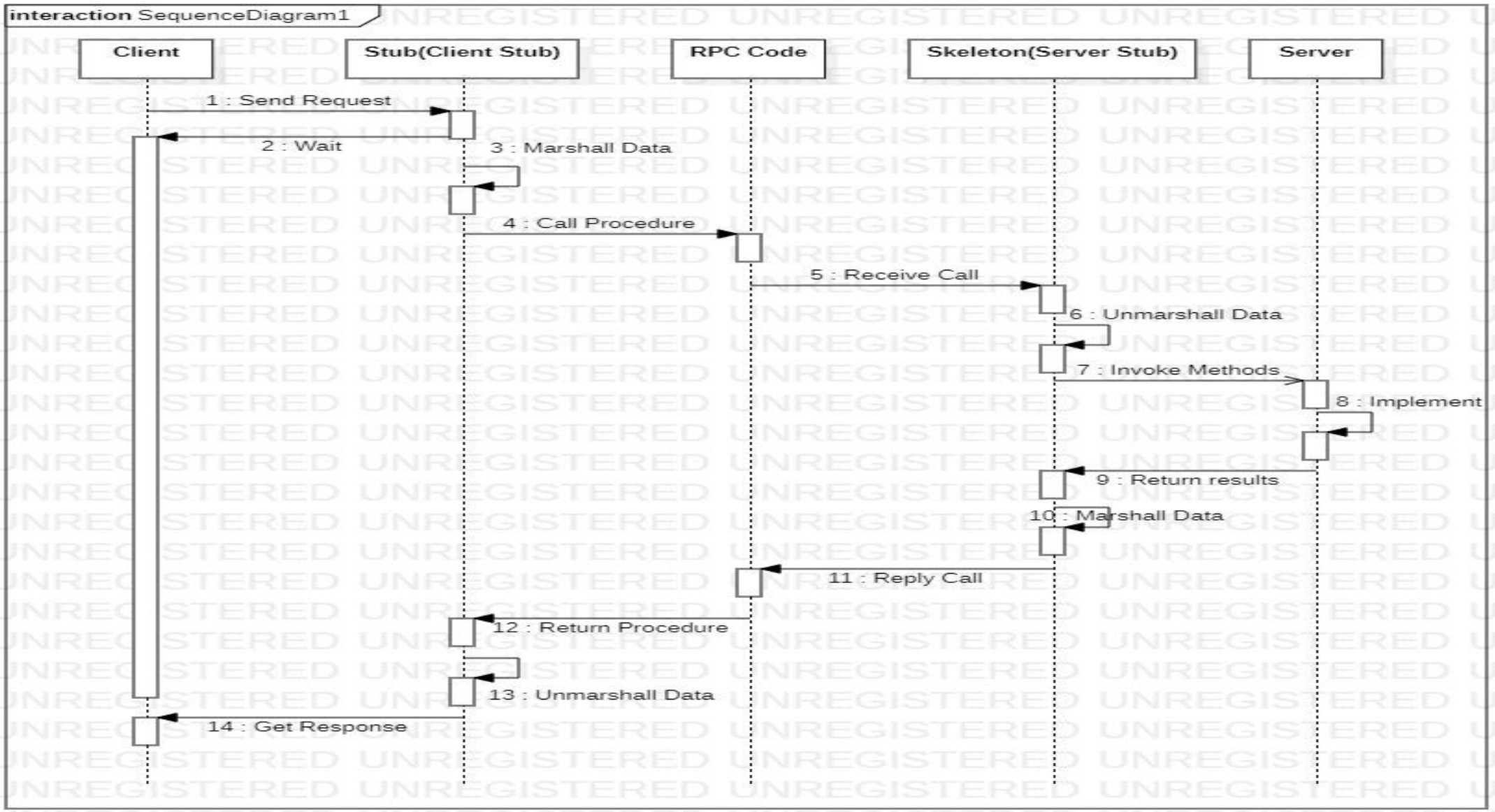
Advantages of RPC:

1. **Ease of Use**:
    1. Abstracts the network communication, making distributed computing more accessible.
2. **Language Neutrality**:
    1. Many RPC frameworks (e.g., gRPC, Thrift) allow communication between programs written in different languages.
3. **Efficiency**:
    1. Modern RPC systems optimize data transmission with compact serialization formats (e.g., Protocol Buffers).
4. **Scalability**:
    1. Facilitates microservices and distributed systems, allowing developers to scale components independently.

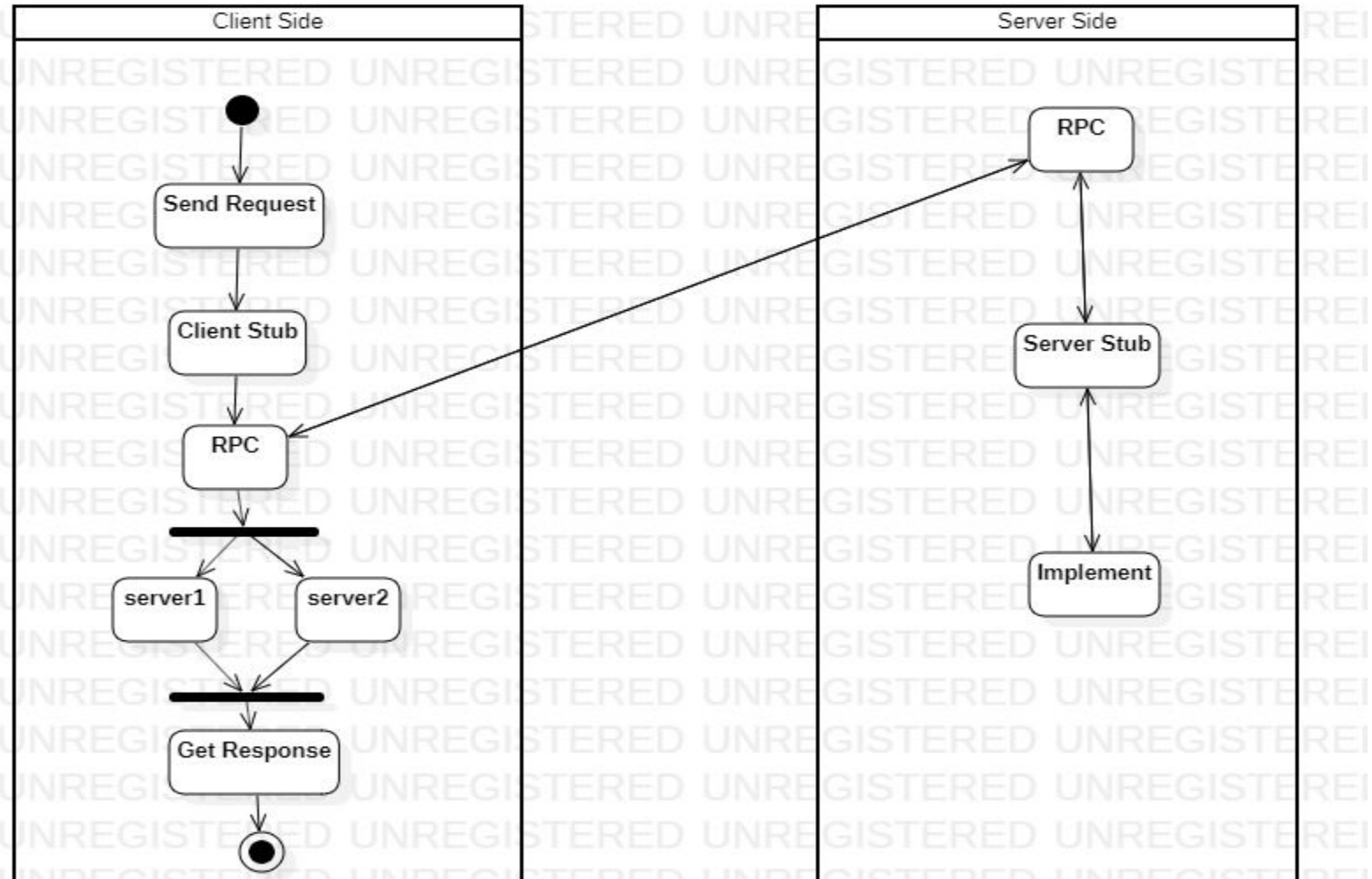


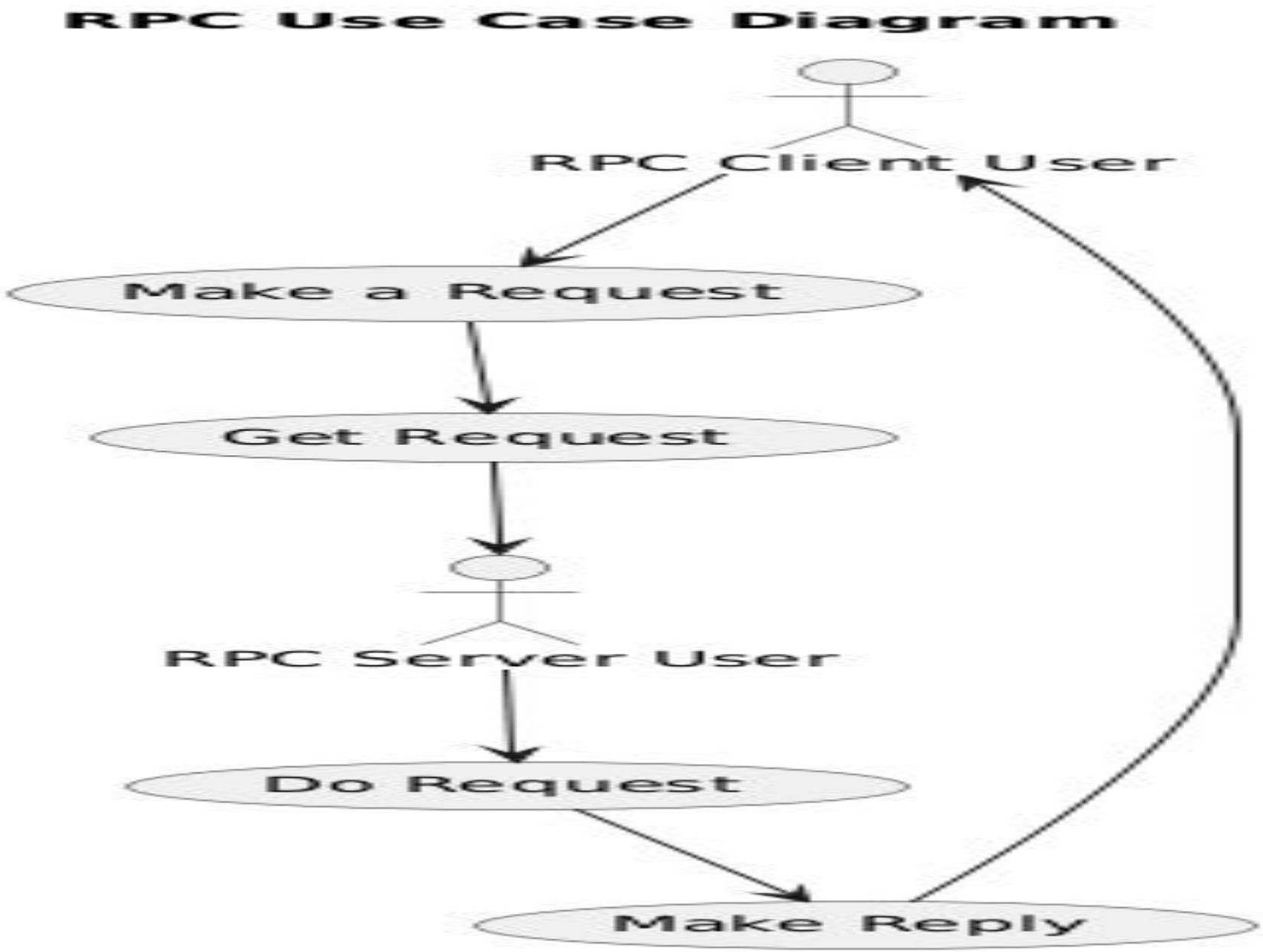
Implementation of RPC mechanism

Sequence diagram:

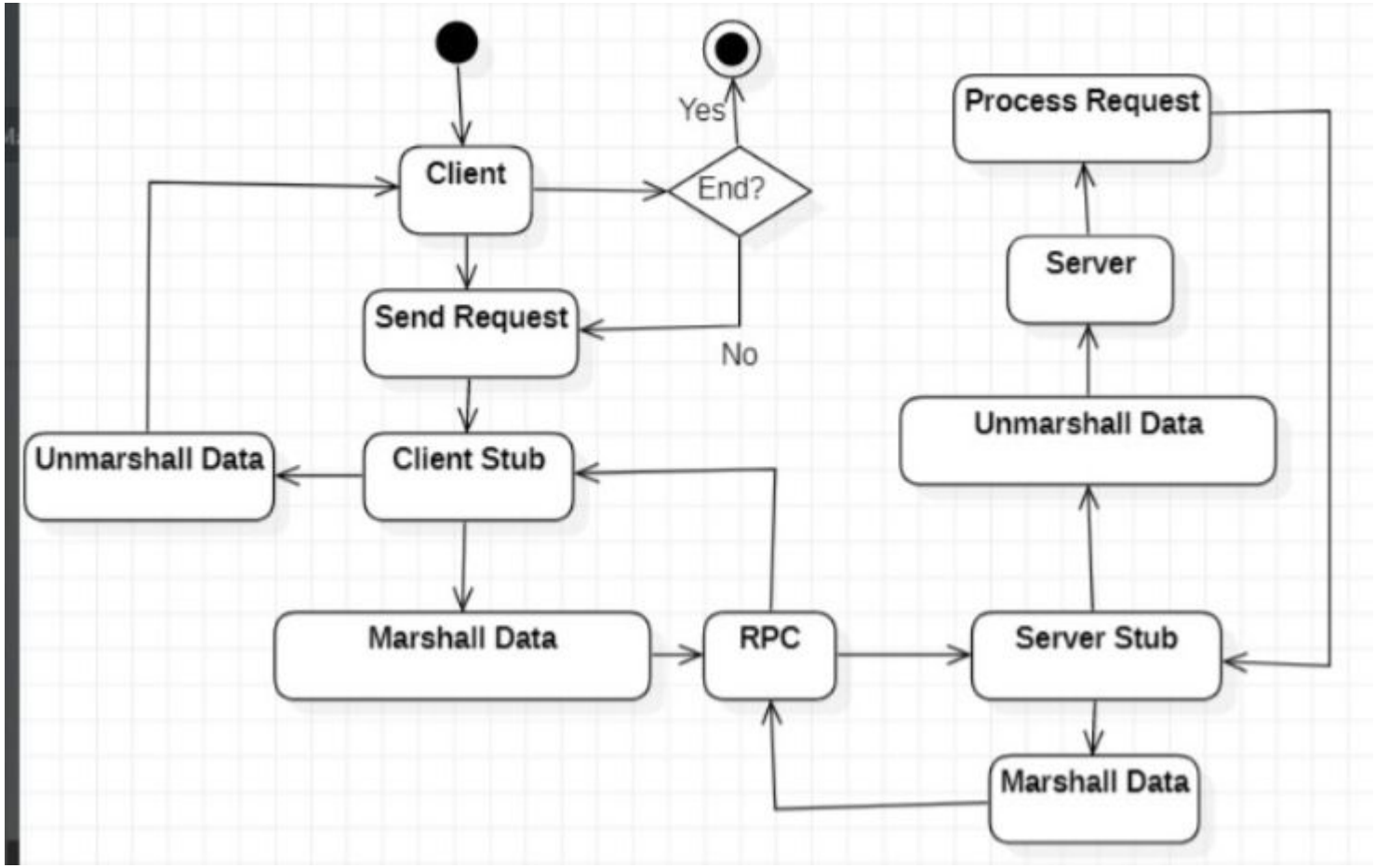


Activity diagram :



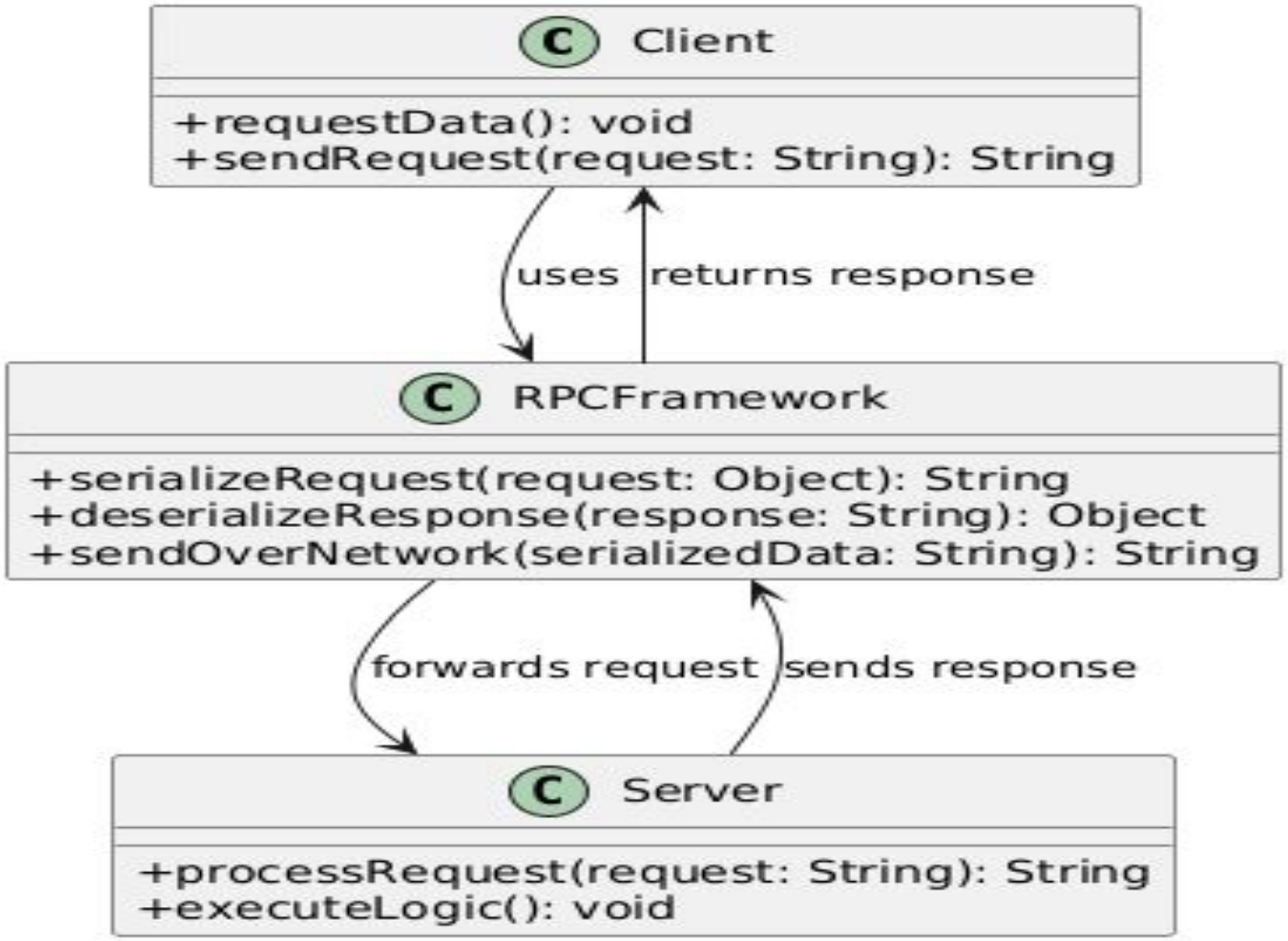

RPC Use Case Diagram
RPC Client User
Make a Request
Get Request
RPC Server User
Do Request
Make Reply

STATE CHART DIAGRAM :

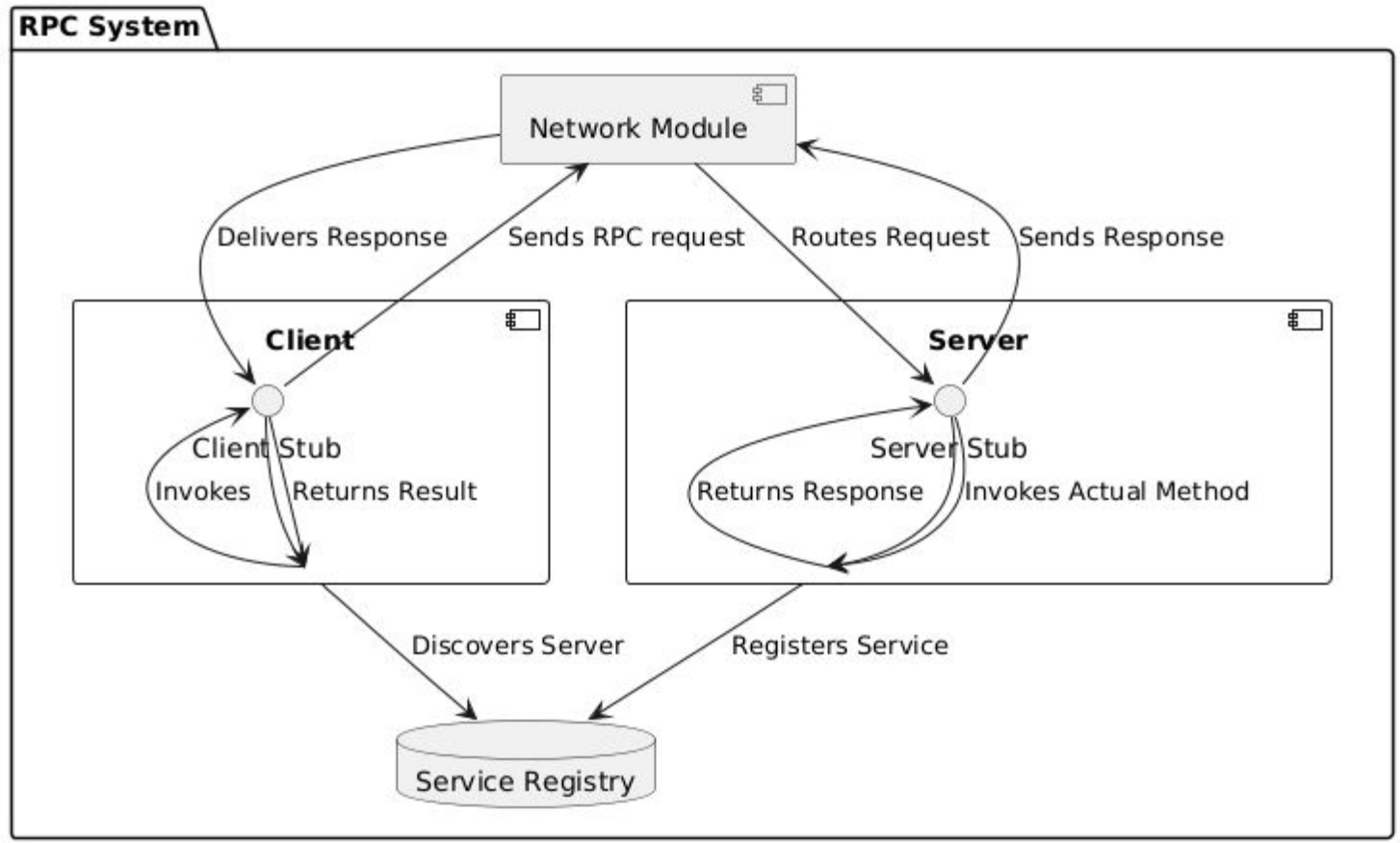


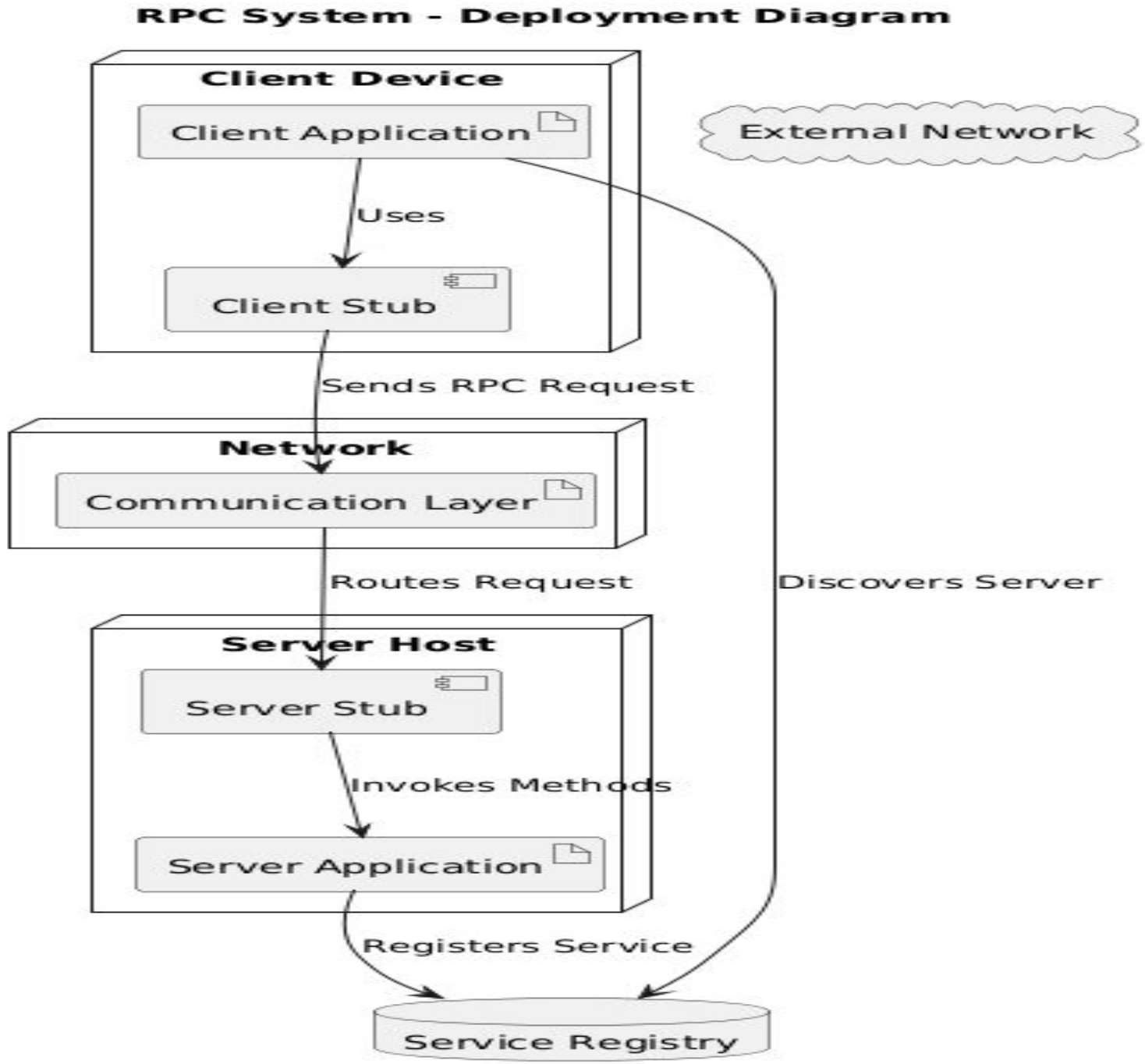
Task11: Draw UML Structural Diagrams for Remote Procedure Call Implementation.

CLASS DIAGRAM :



RPC System - Component Diagram



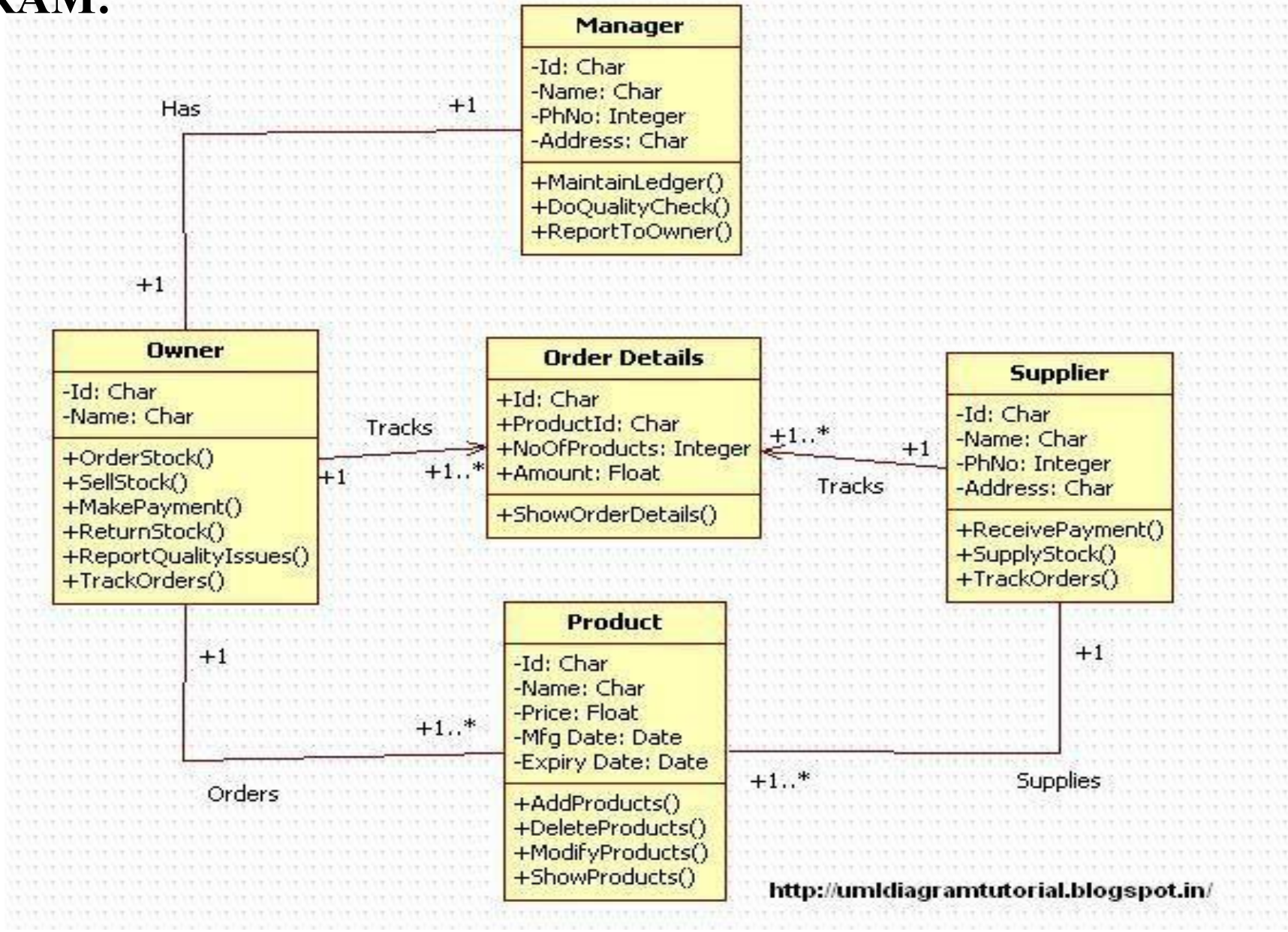

RPC System - Deployment Diagram
Client Device
Client Application
Uses
Client Stub
External Network
Sends RPC Request
Network
Communication Layer
Routes Request
Discovers Server
Server Host
Server Stub
Invokes Methods
Server Application
Registers Service
Service Registry

Task-12 UML Diagrams For Stock Maintenance System

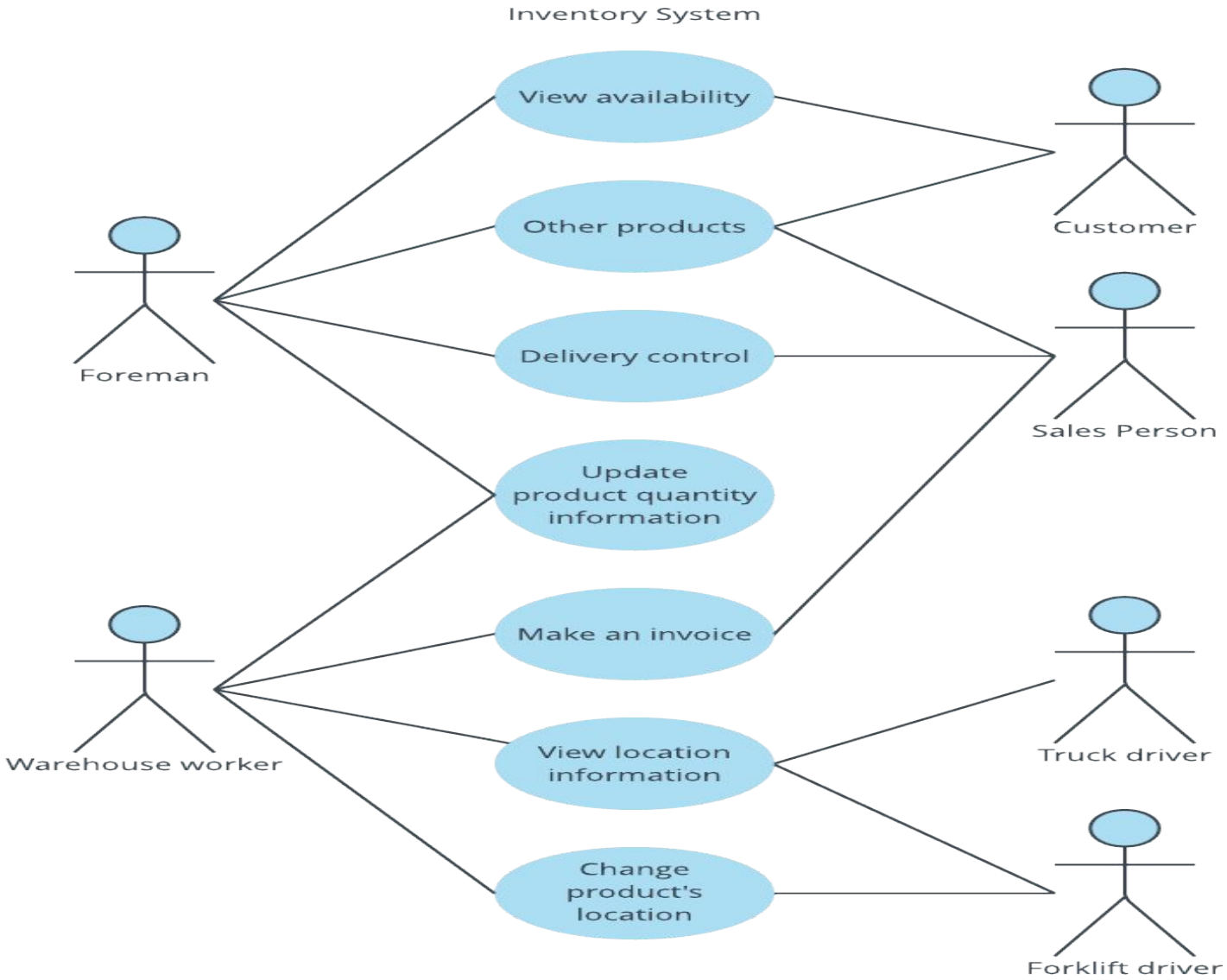
Stock maintenance :

- The stock maintenance system initial requirement is to develop a project about the mechanism of the stock maintenance system is caught from the customer.
- The requirement are analysed and refined which enables the end-users to efficiently use stock-maintenance system.
- The complete project is developed after the while project analysis explaining about the scope and the project statement is prepared.
- The process of stock maintenance is that the customer login to the particular site to place the order for the customer login to the particular site to place the order for the customer product.
- The stock maintenance system is described sequentially through steps.
- These steps are a customer login to the particular site.
- They fill the customer details.
- They place the order for their product.
- The vendor login and views the customer details and orders.

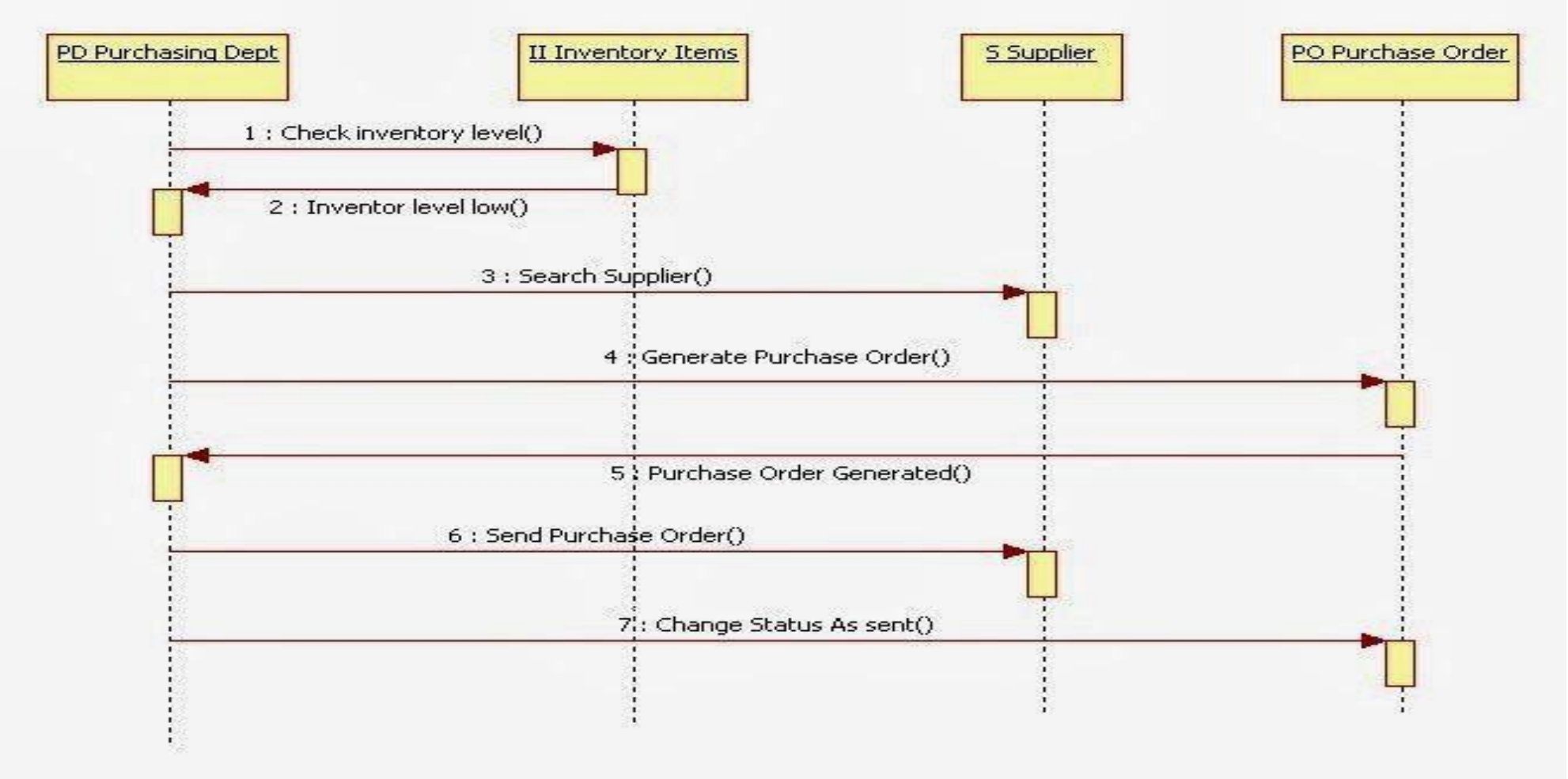
CLASS DIAGRAM:



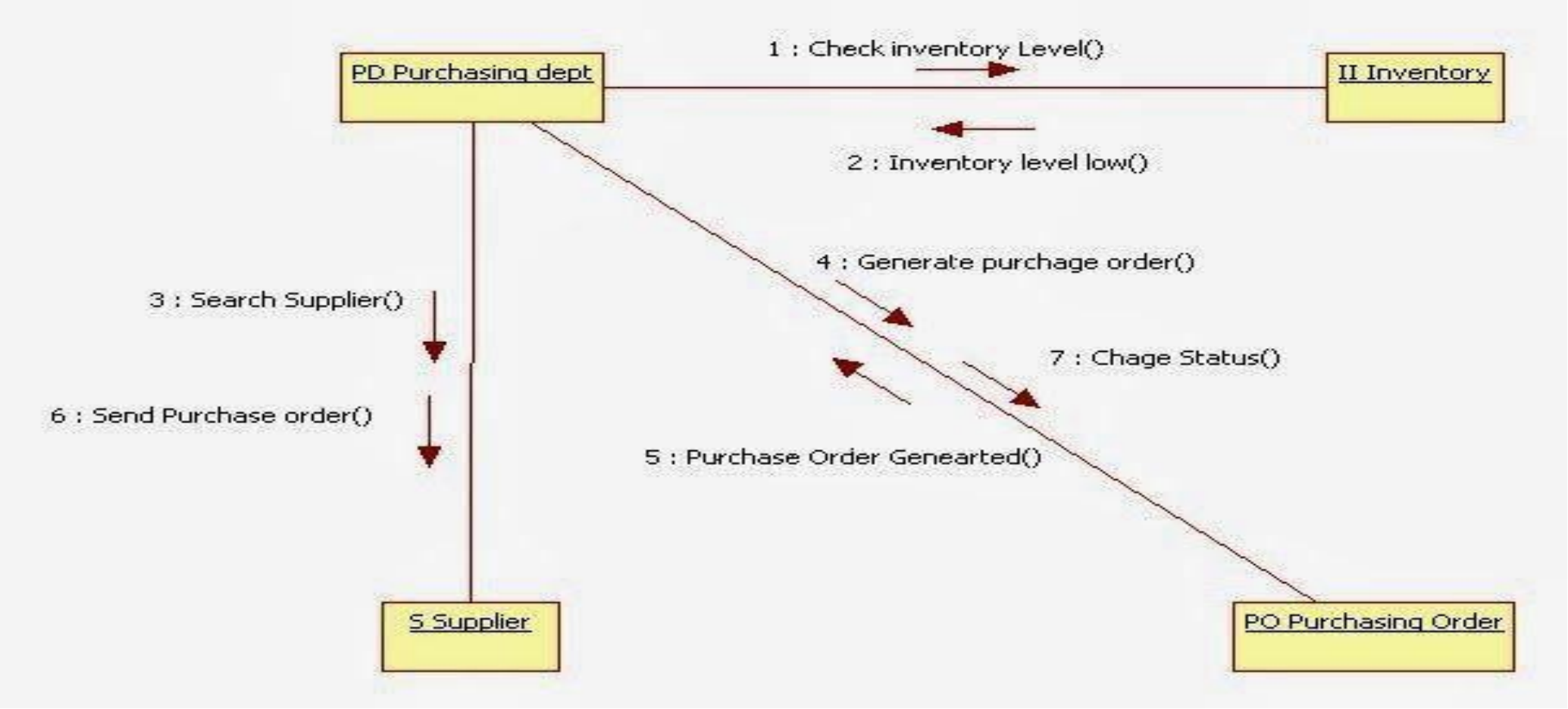
USE CASE DIAGRAM:



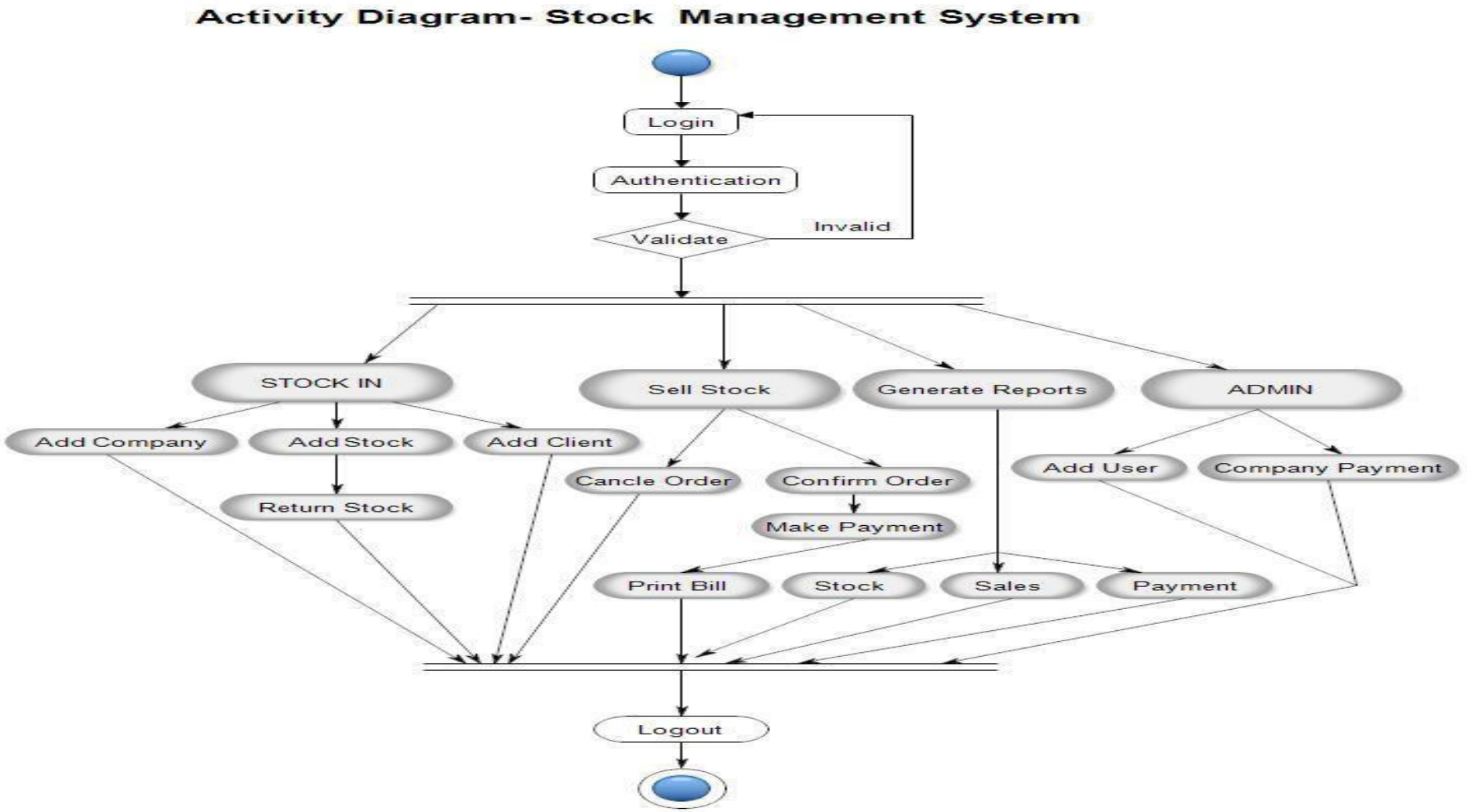
SEQUENCE DIAGRAM:



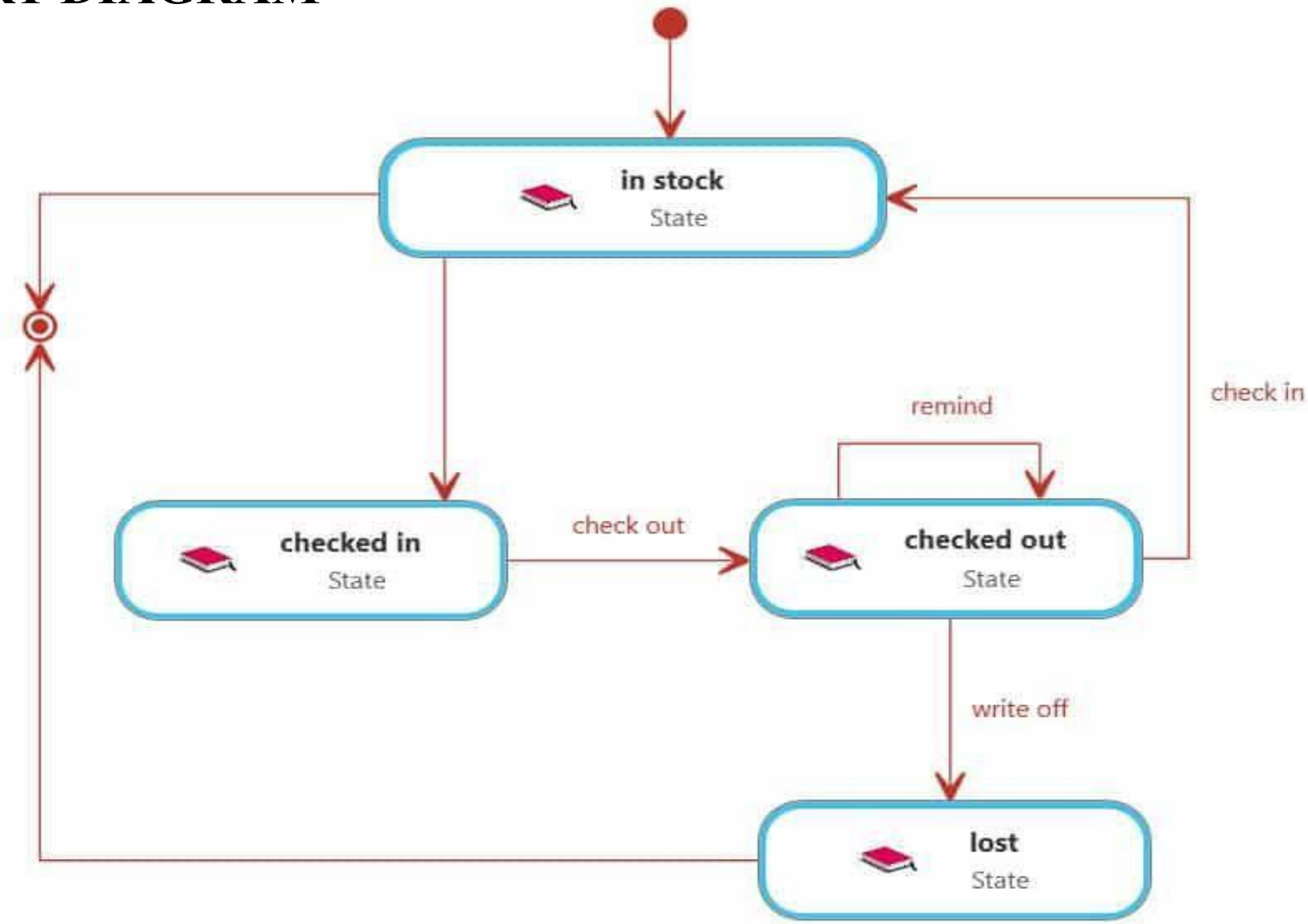
COLLABORATION DIAGRAM:



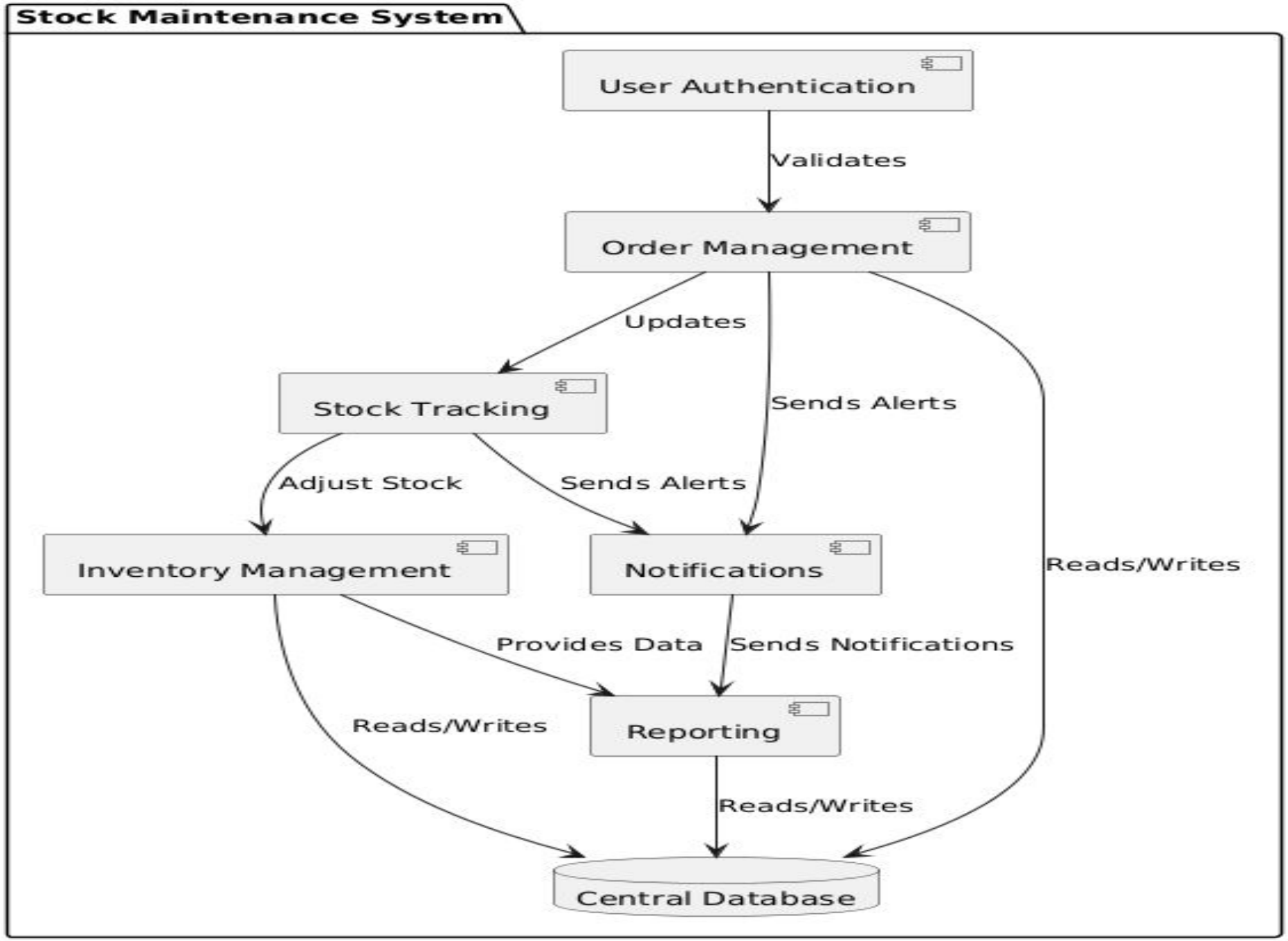
ACTIVITY DIAGRAM:



STATE CHART DIAGRAM



Stock Maintenance System - Component Diagram



Stock Maintenance System - Deployment Diagram