Object Oriented Design

Class Diagram

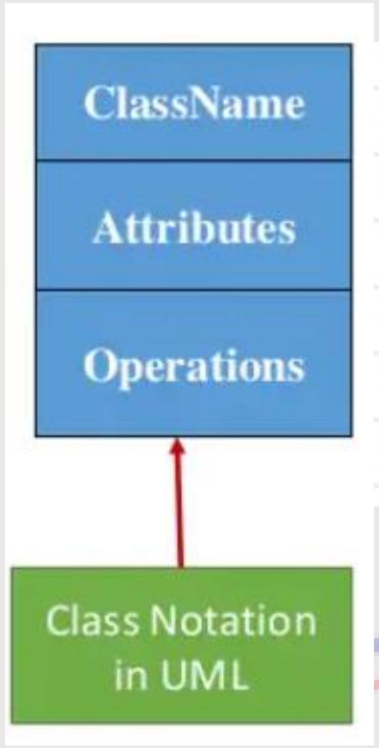
- It is a static structure that gives an overview of a software system by displaying classes, attributes, operations, and their relationships between each other.
- UML CLASS DIAGRAM is made up of a set of classes and a set of relationship between classes.
- This Diagram includes the class name, attributes, and operation in separate designated compartments.

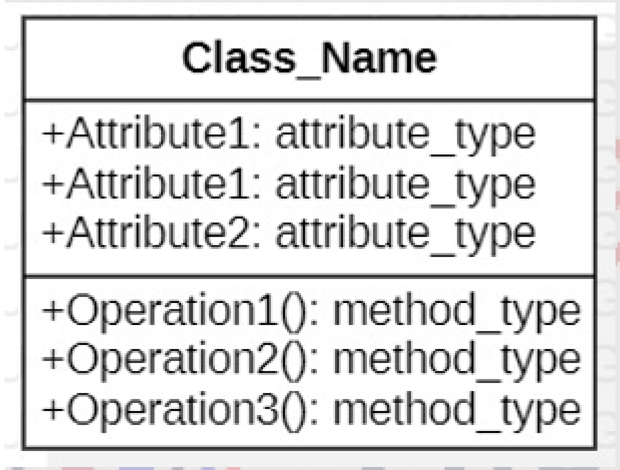
What is a Class?

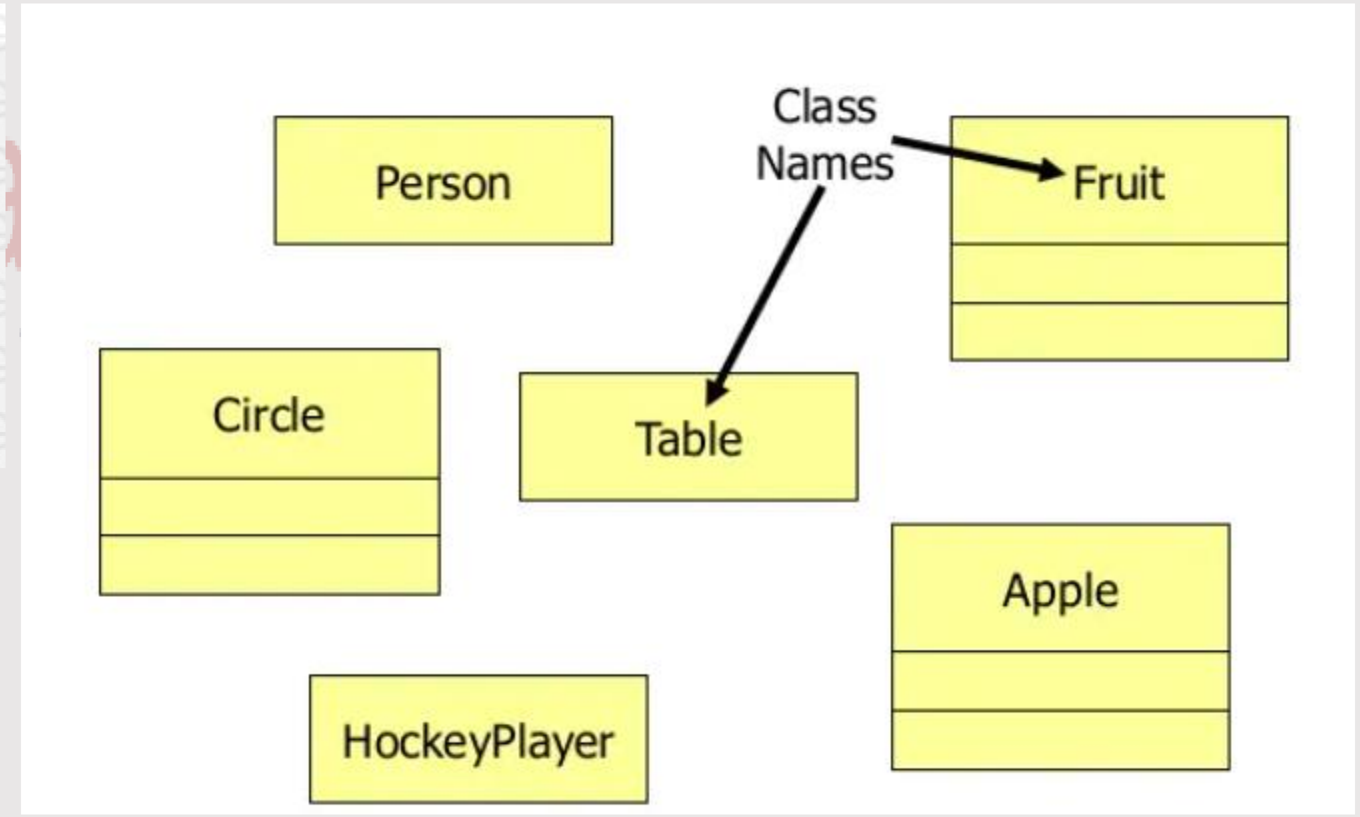
- A class is the blueprint of an object which can share the same relationships, attributes, operations, & semantics.
- **Structural features** (attributes) define what objects of the class "know"
  - Represent the state of an object of the class
  - Are descriptions of the structural or static features of a class
- **Behavioral features** (operations) define what objects of the class "can do"
  - Define the way in which objects may interact
  - Operations are descriptions of behavioral or dynamic features of a class
- **Following rules must be taken care of while representing a class**:
  - A class name should always start with a capital letter.
  - A class name should always be in the center of the first compartment.
  - A class name should always be written in **bold** format.

Class Notation

- **Class Name**
  - The name of the class appears in the first partition.

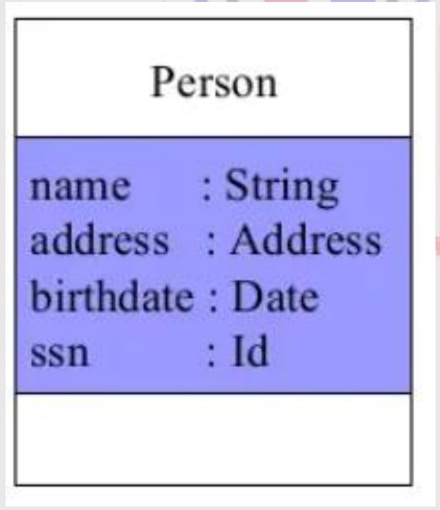


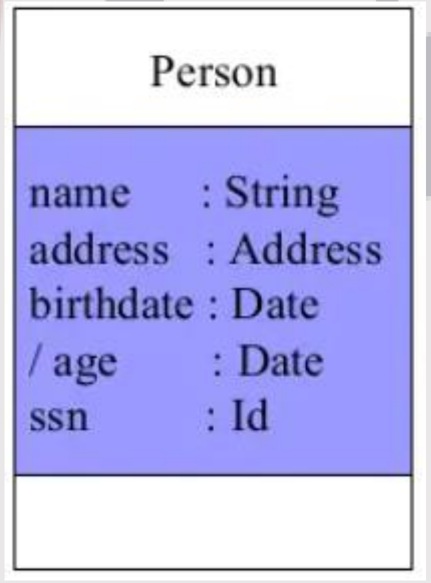

Class_Name
+Attribute1: attribute_type
+Attribute1: attribute_type
+Attribute2: attribute_type
+Operation1(): method_type
+Operation2(): method_type
+Operation3(): method_type

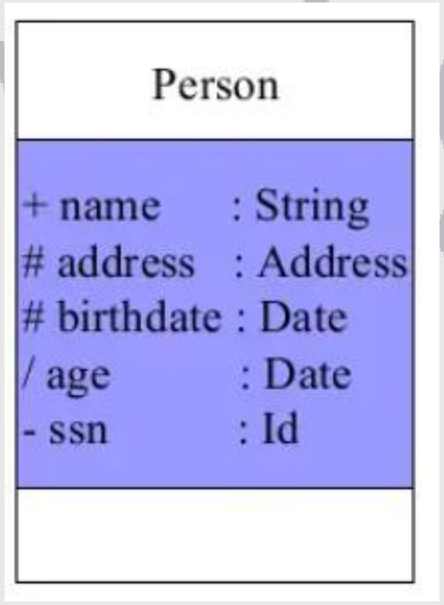


Class Notation

- **Class Attributes**
  - Attributes are shown in the second partition.
  - Attributes are named property of a class that describes the object being modelled.
  - Attributes are usually listed in the form- attributeName: Type
  - A derived attribute is designated by preceding '/' as in: / age: Date

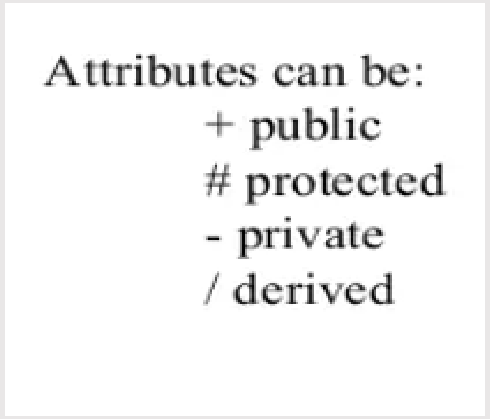






Attributes can be:
+ public
# protected
- private
/ derived

Attribute Modifiers


Attributes can be:
+ public
# protected
- private
/ derived

- In object-oriented design, there is a notation of visibility for attributes and operations. UML identifies four types of visibility: **public**, **protected**, **private**, and **package**.

•**Public attributes**: denoted with the modifier '+', can be accessed from outside the class

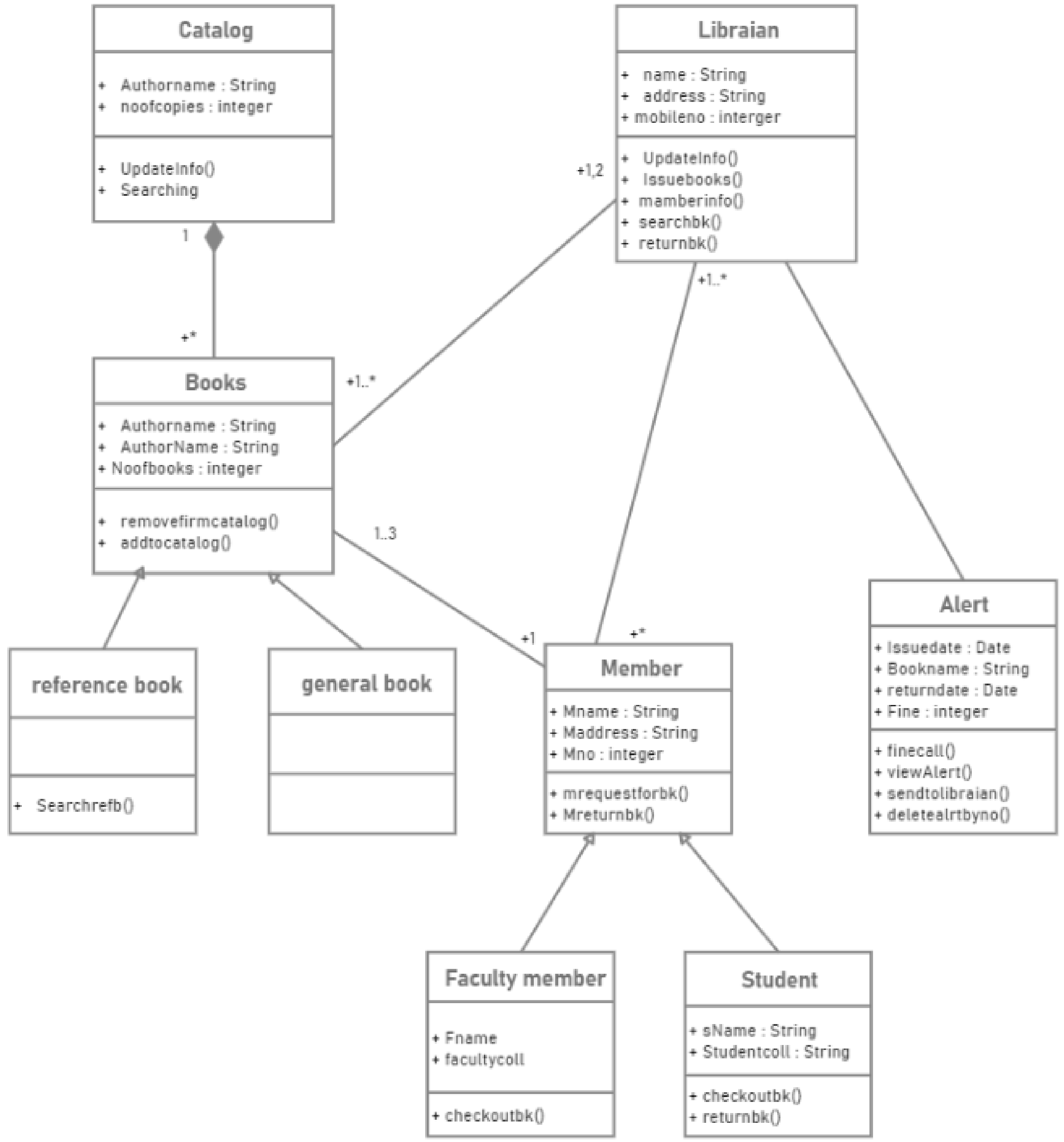
•**Private attributes:** denoted with the modifier '-', can only be accessed from within the class

•**Protected attributes:** denoted with the modifier '#', can be accessed from within the class, or any descendant (e.g. a subclass)

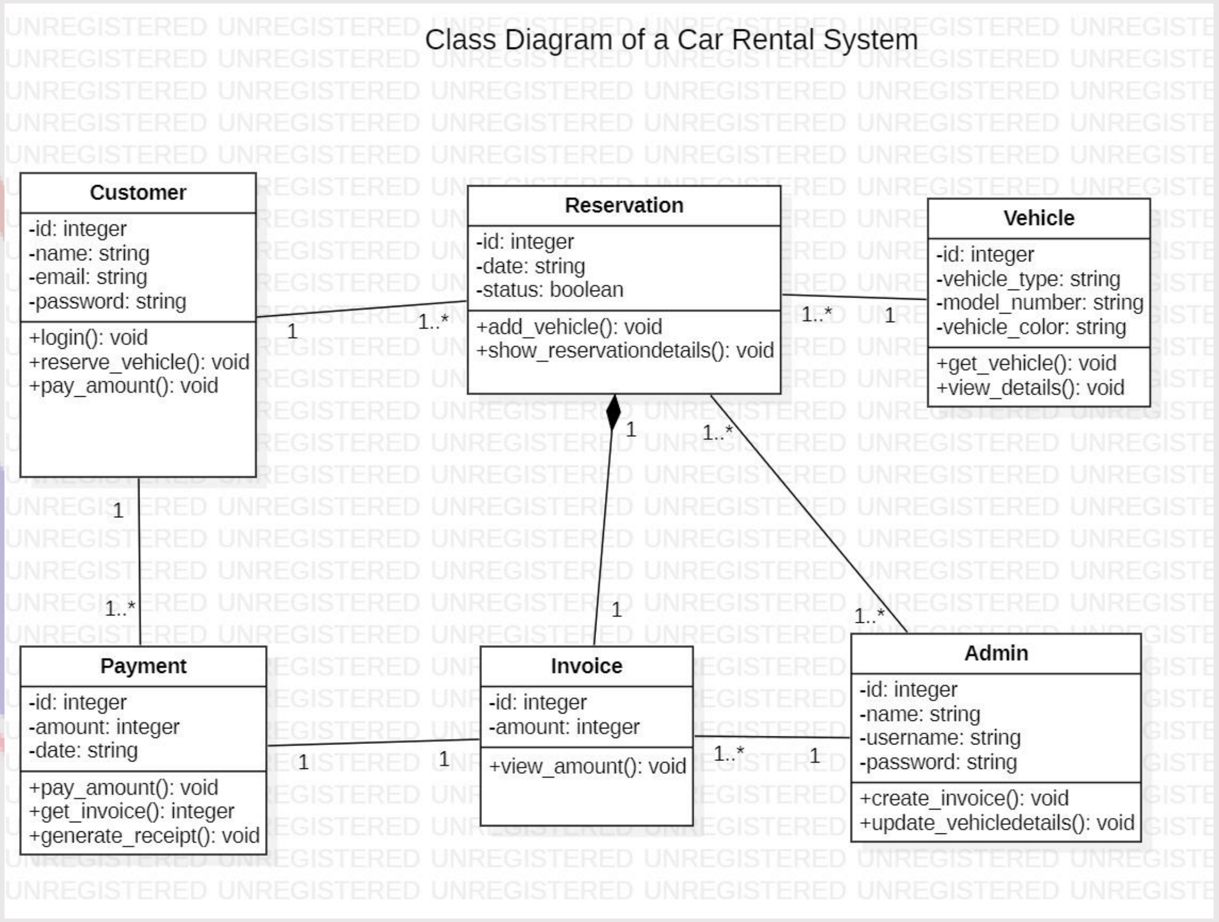
Class Notation

- **Class Operations** (Methods)
  - Operations are shown in the third partition. They are services the class provides.
  - Operations map onto class methods in code

Class Diagram of Library Management System



Class Diagram Example

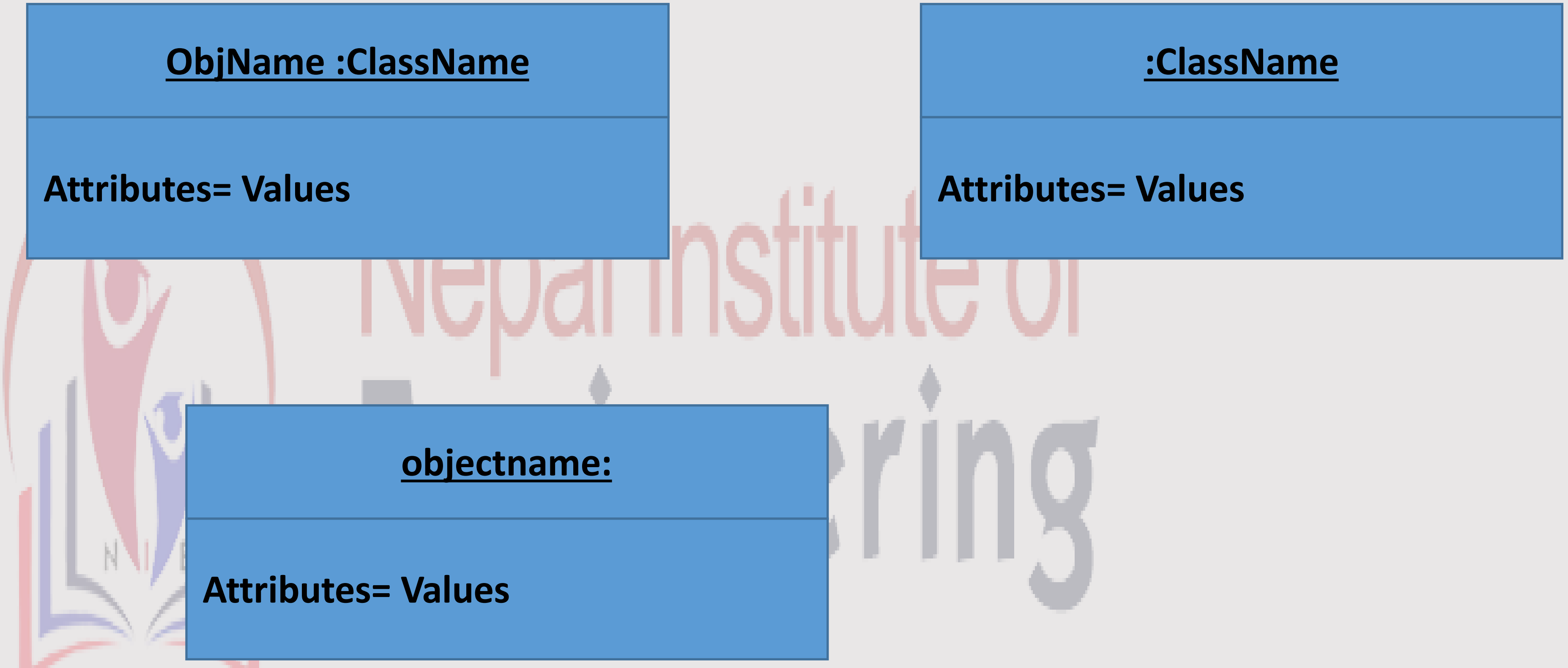


Object Diagram

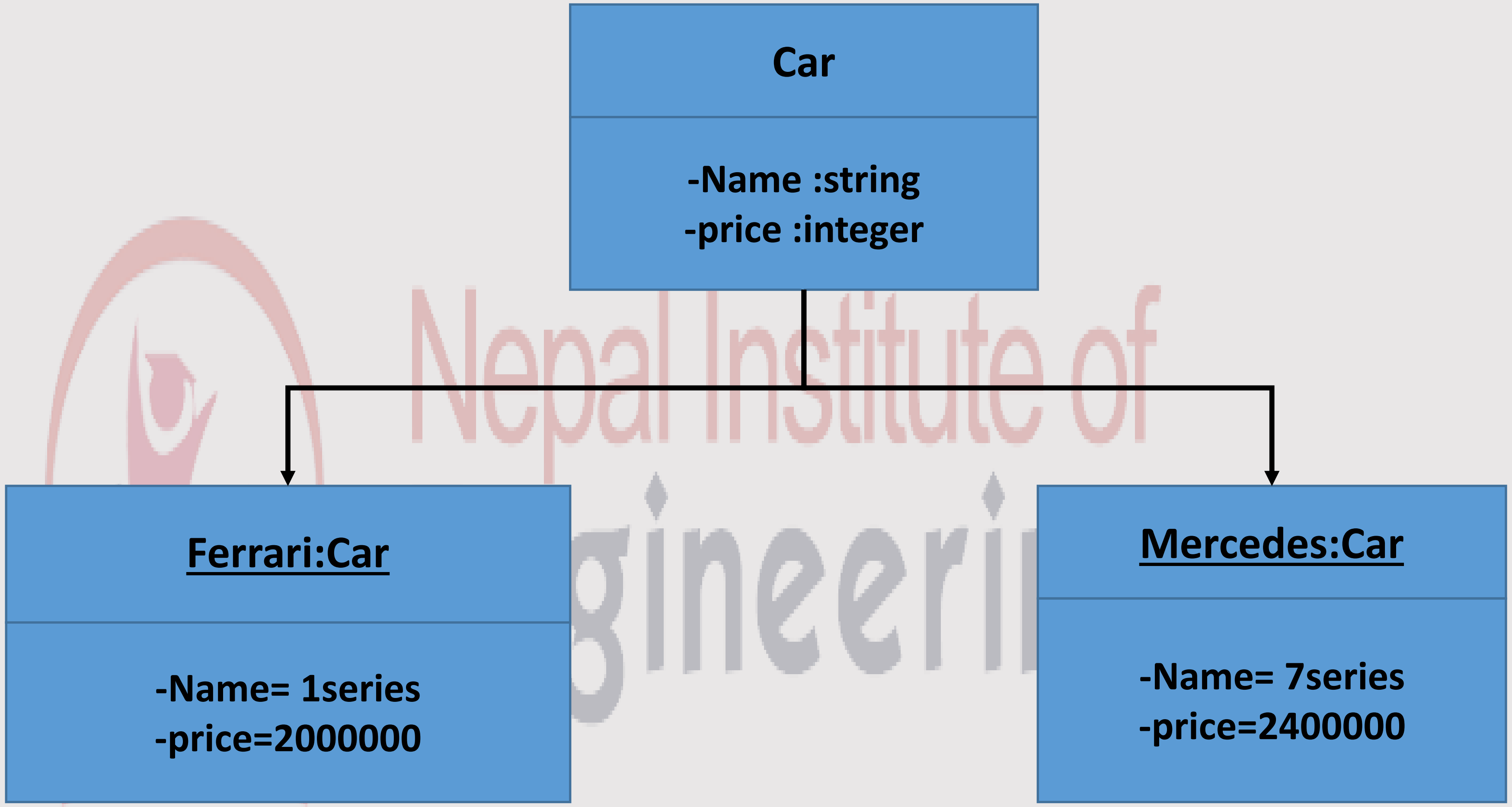
- Object diagrams are derived from class diagrams so object diagrams are dependent upon class diagrams.
- Object diagrams represent an instance of a class diagram. The basic concepts are similar for class diagrams and object diagrams.
- Object diagrams also represent the static view of a system but this static view is a snapshot of the system at a particular moment.
- Captures Run time/ Real Time changes of a system
- It includes attributes and data values of any instance.

Object Notation

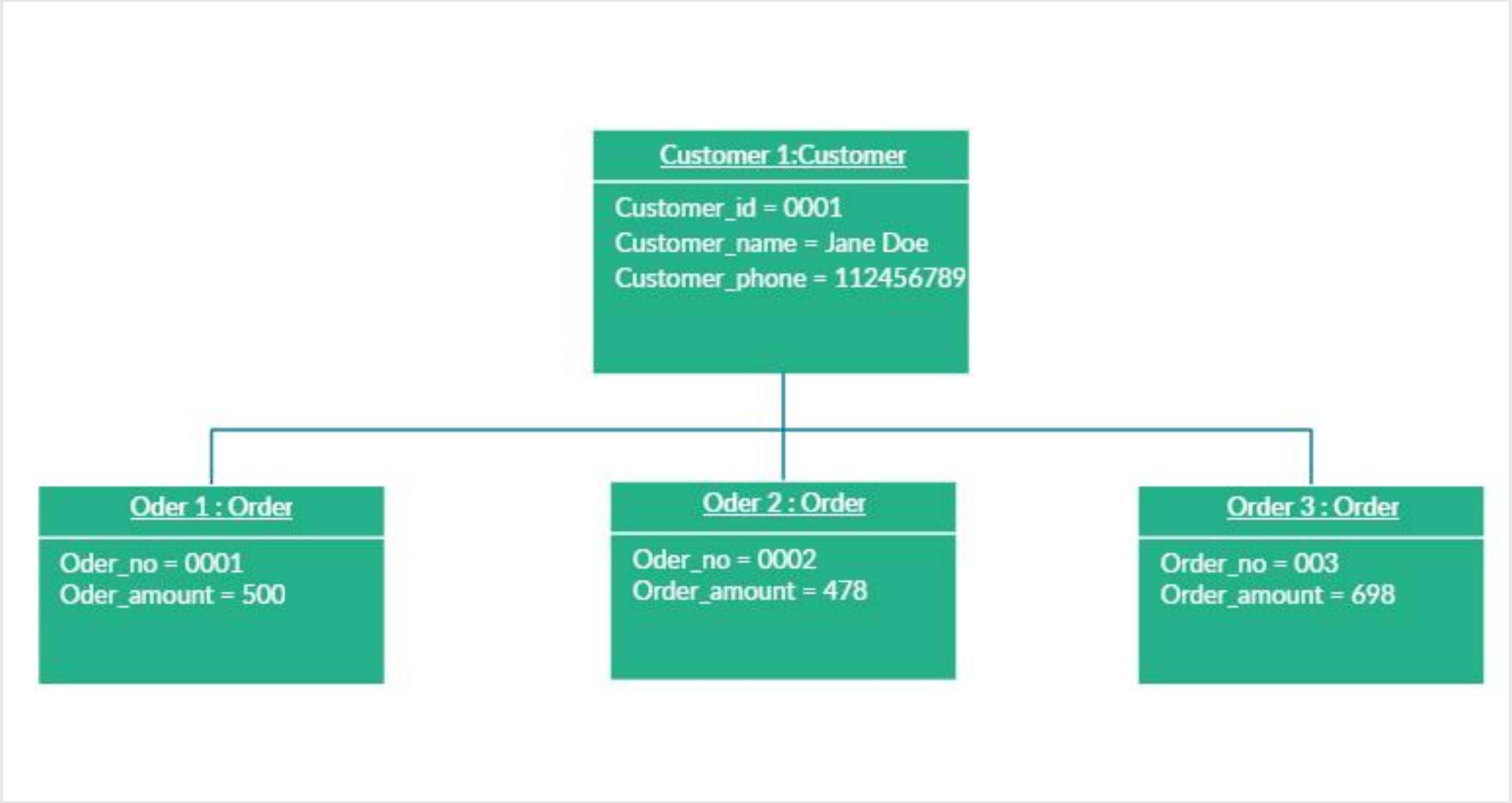
->



Objects Example



Object Diagram Example

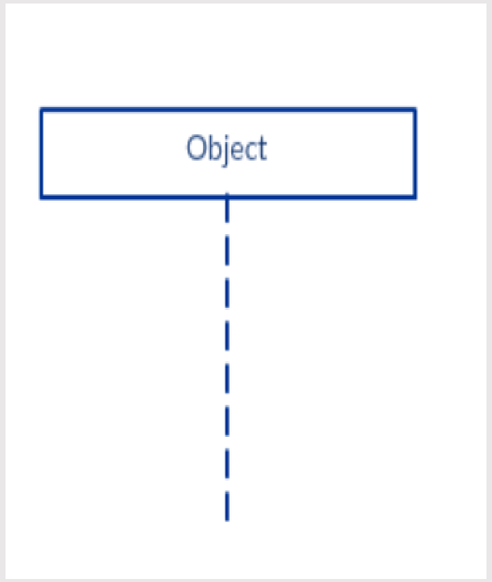


Sequence Diagram

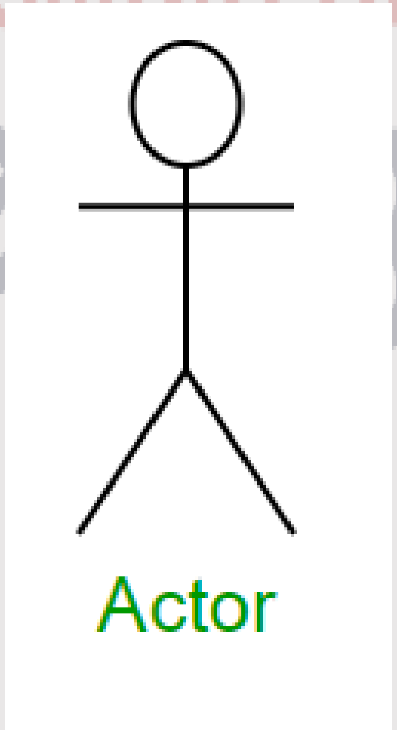
- The most commonly used **interaction** diagram.
- Also known as **Event diagram**.
- A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place.
- Sequence diagrams describe how and in what order the objects in a system function.
- They illustrate how the different parts of a system interact with each other to carry out a function, and the order in which the interactions occur when a particular use case is executed.
- In simpler words, a sequence diagram shows different parts of a system work in a ‘sequence’ to get something done.

Sequence Diagram Notations

- **Lifeline:** An individual participant in the sequence diagram is represented by a lifeline.



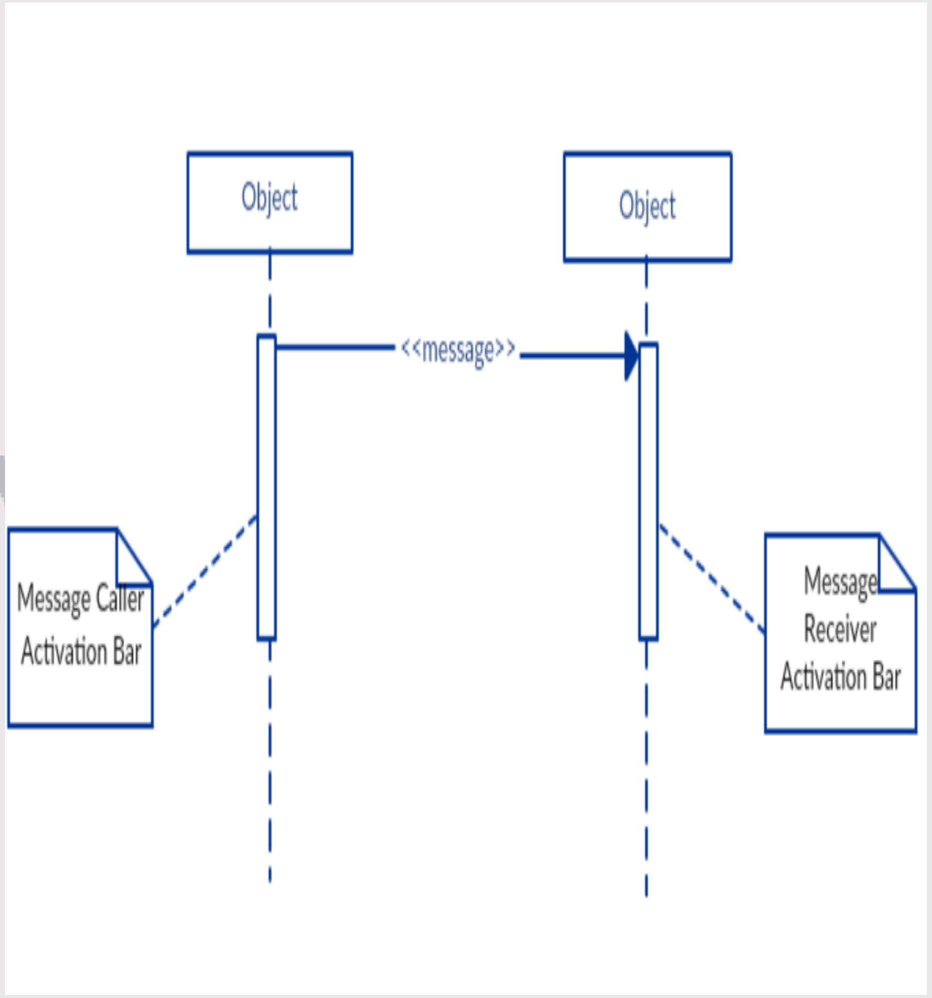
- **Actor:** It represents the role, which involves human users and external hardware or subjects.



Sequence Diagram Notation

Activation Bars:

- The use of the activation bar on the lifelines of the Message Caller (the object that sends the message) and the Message Receiver (the object that receives the message) indicates that both are active/is instantiated during the exchange of the message.

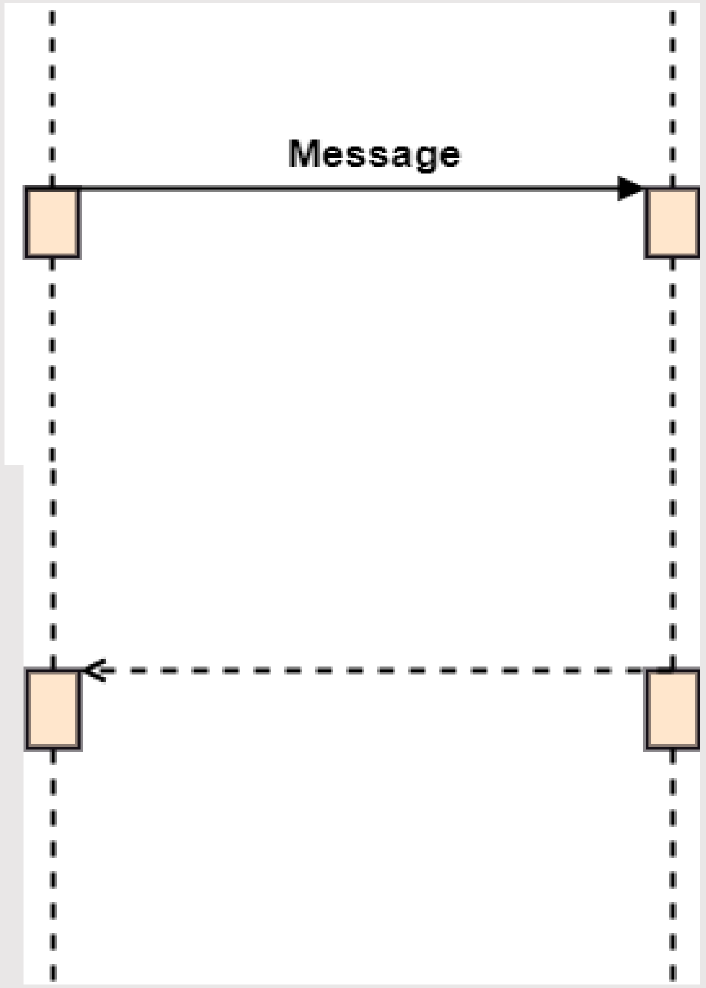

Object
Object
<<message>>
Message Caller Activation Bar
Message Receiver Activation Bar

Sequence Diagram Notation

Messages:

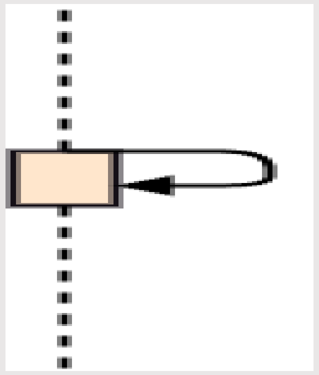
- **Call Message:** It defines a particular communication between the lifelines of an interaction, which represents that the target lifeline has invoked an operation.

- **Reply Message:** It defines a particular communication between the lifelines of interaction that represent the flow of information from the receiver of the corresponding caller message.

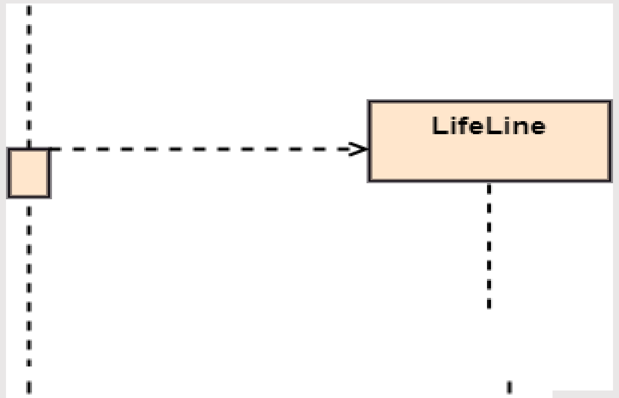


Sequence Diagram Notation

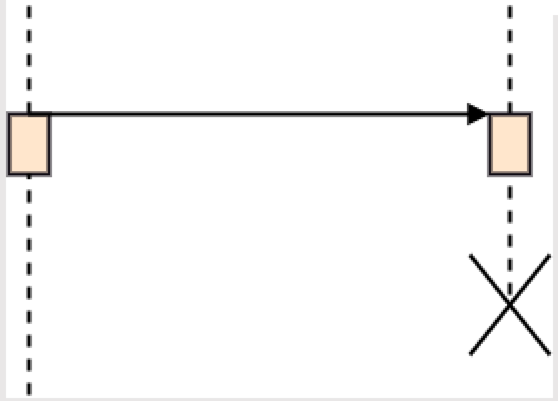
- **Self Message:** It describes a communication, particularly between the lifelines of an interaction that represents a message of the same lifeline, has been invoked.



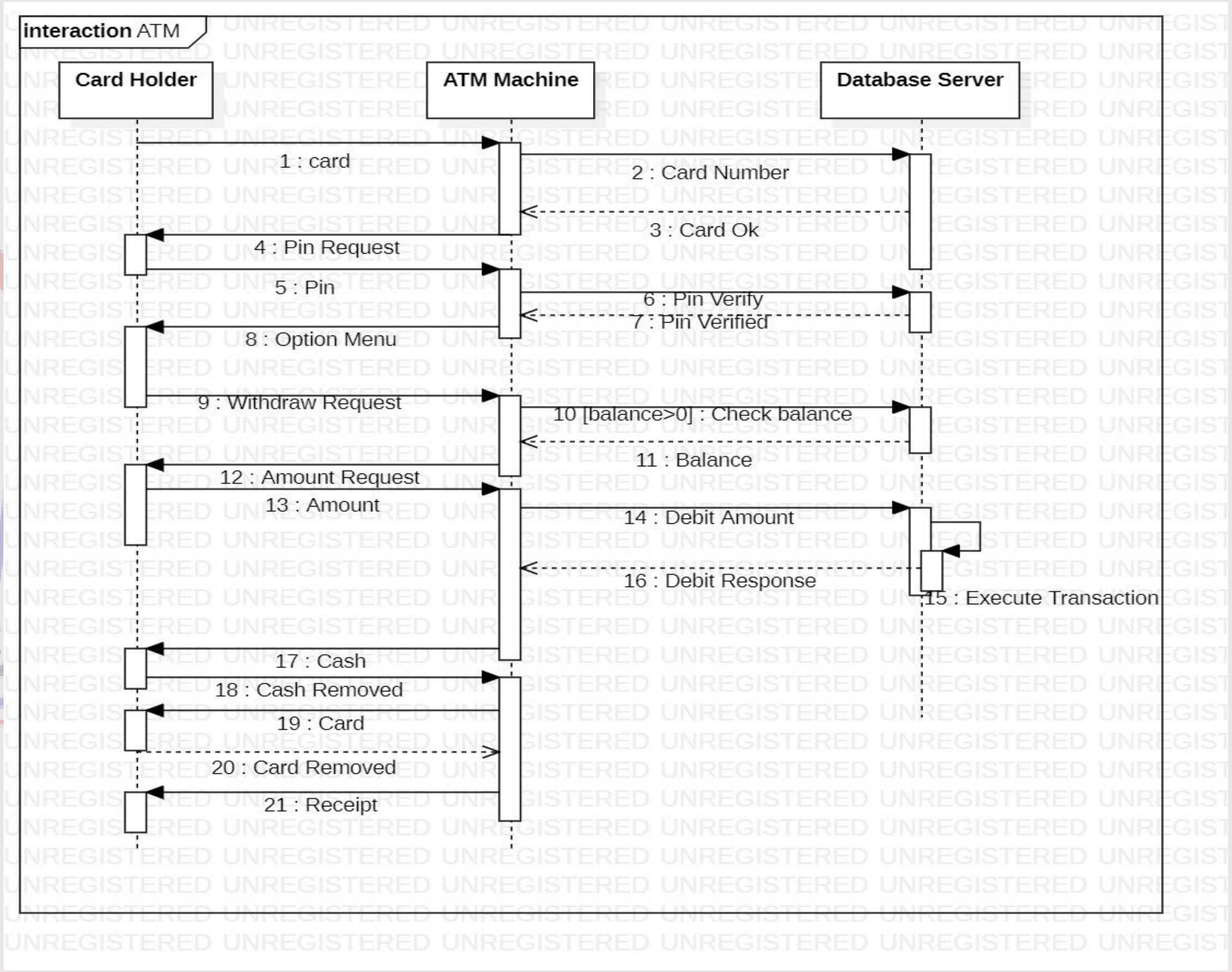
- **Create Message:** It describes a communication, particularly between the lifelines of an interaction describing that the target (lifeline) has been instantiated.



- **Destroy Message:** It describes a communication, particularly between the lifelines of an interaction that depicts a request to destroy the lifecycle of the target.



Examples of Sequence Diagram



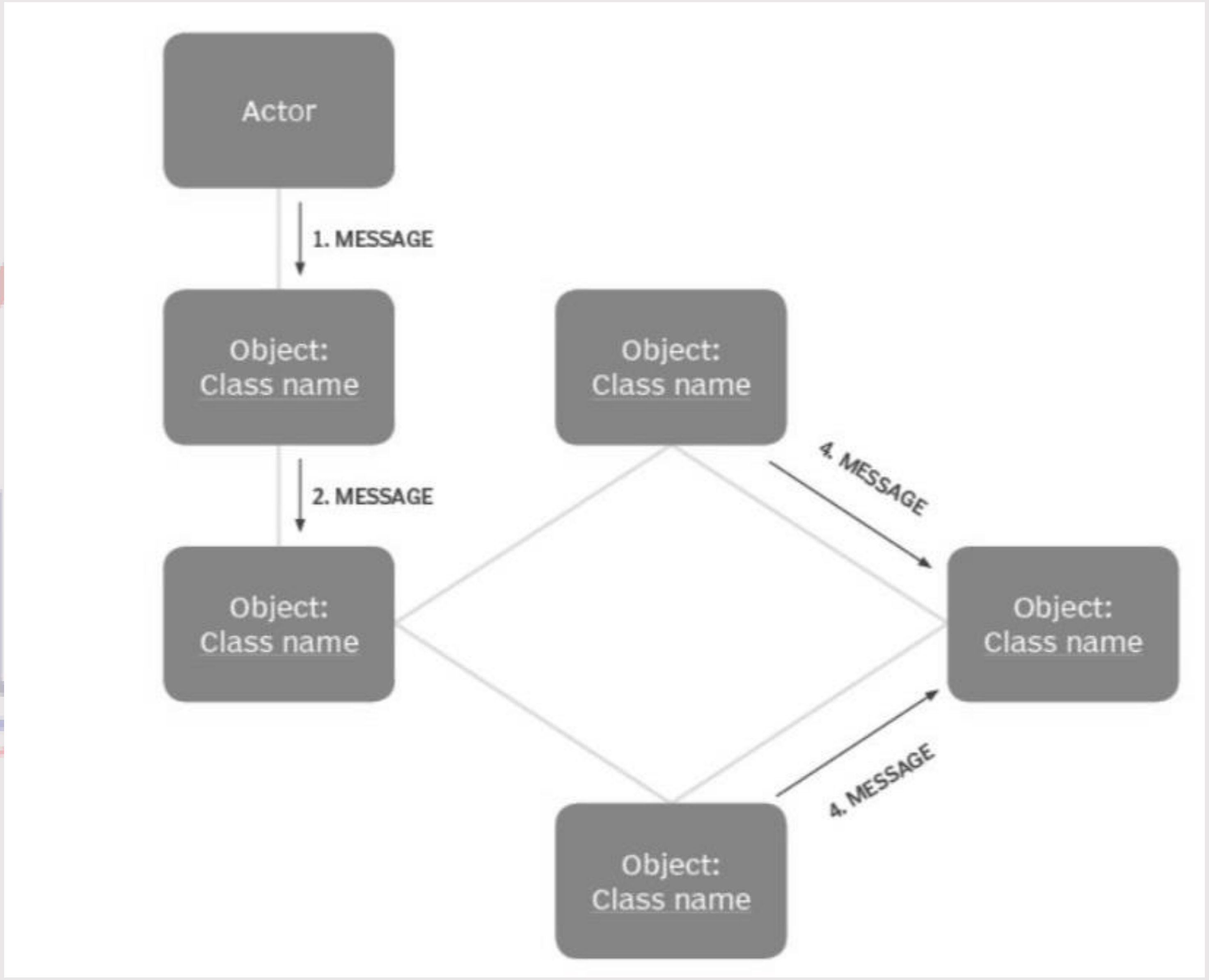
Collaboration Diagram

- They are used to understand the object architecture within a system rather than the flow of a message as in a sequence diagram.
- Also called communication diagram
- It emphasizes the structural aspects of an interaction diagram – how lifeline connects.
- It depicts the relationships and interactions among software objects.
- It allows you to focus on the elements rather than focusing on the message flow as described in the sequence diagram.
- Sequence diagrams can be easily converted into a collaboration diagram as collaboration diagrams are not very expressive.
- Messages passed over sequencing is indicated by numbering each message hierarchically.

Symbols/ Notations

1. **Objects:** The representation of an object is done by an object symbol with its name and class underlined, separated by a colon.
2. **Actors:** In the collaboration diagram, the actor plays the main role as it invokes the interaction. Each actor has its respective role and name. In this, one actor initiates the use case.
3. **Links:** Links connect objects with actors and are depicted using a solid line between two elements. Each link is an instance where messages can be sent.
4. **Messages:** Messages between objects are shown as a labeled arrow placed near a link. These messages are communications between objects that convey information about the activity and can include the sequence number.

Components of Collaboration Diagram



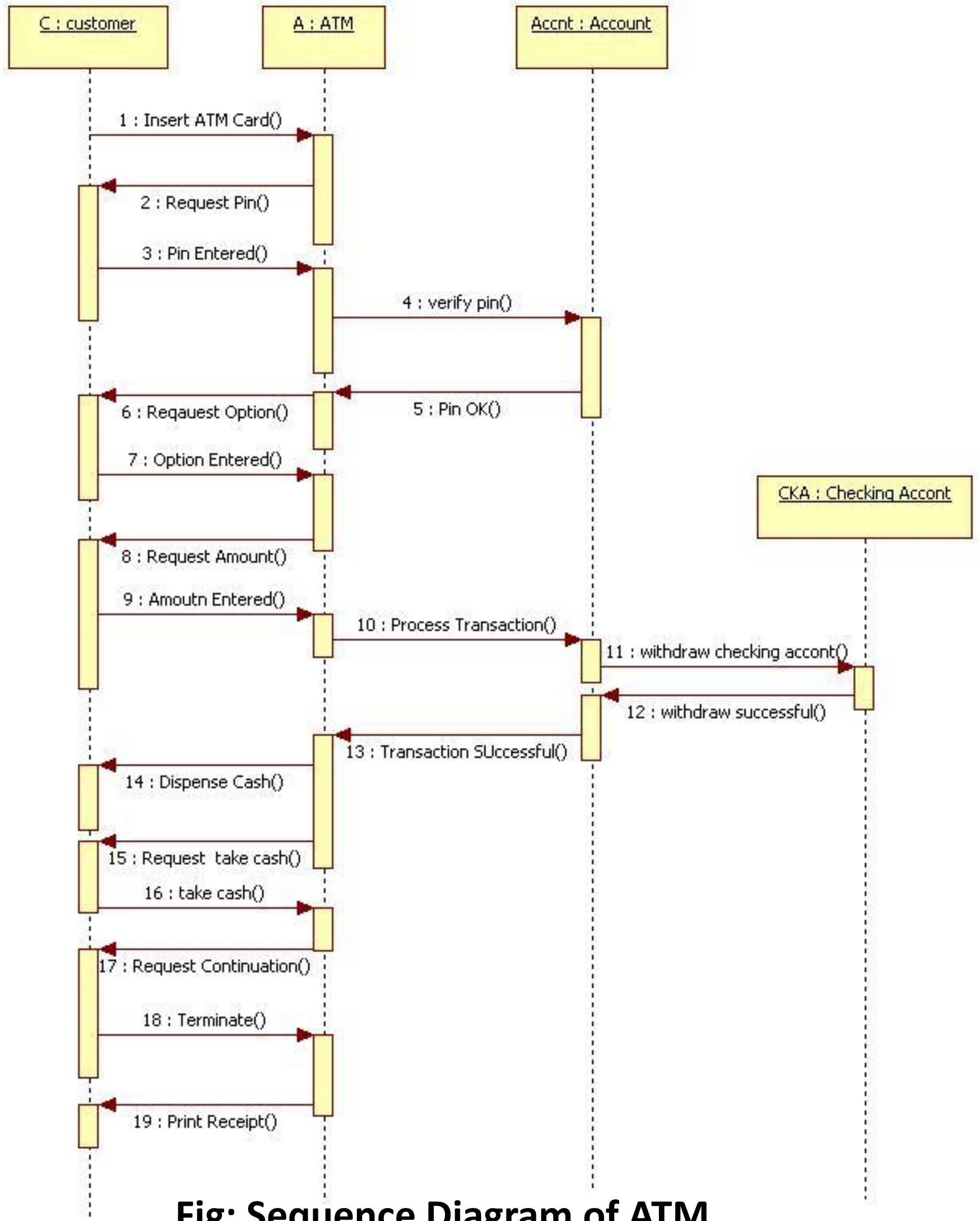


Fig: Sequence Diagram of ATM

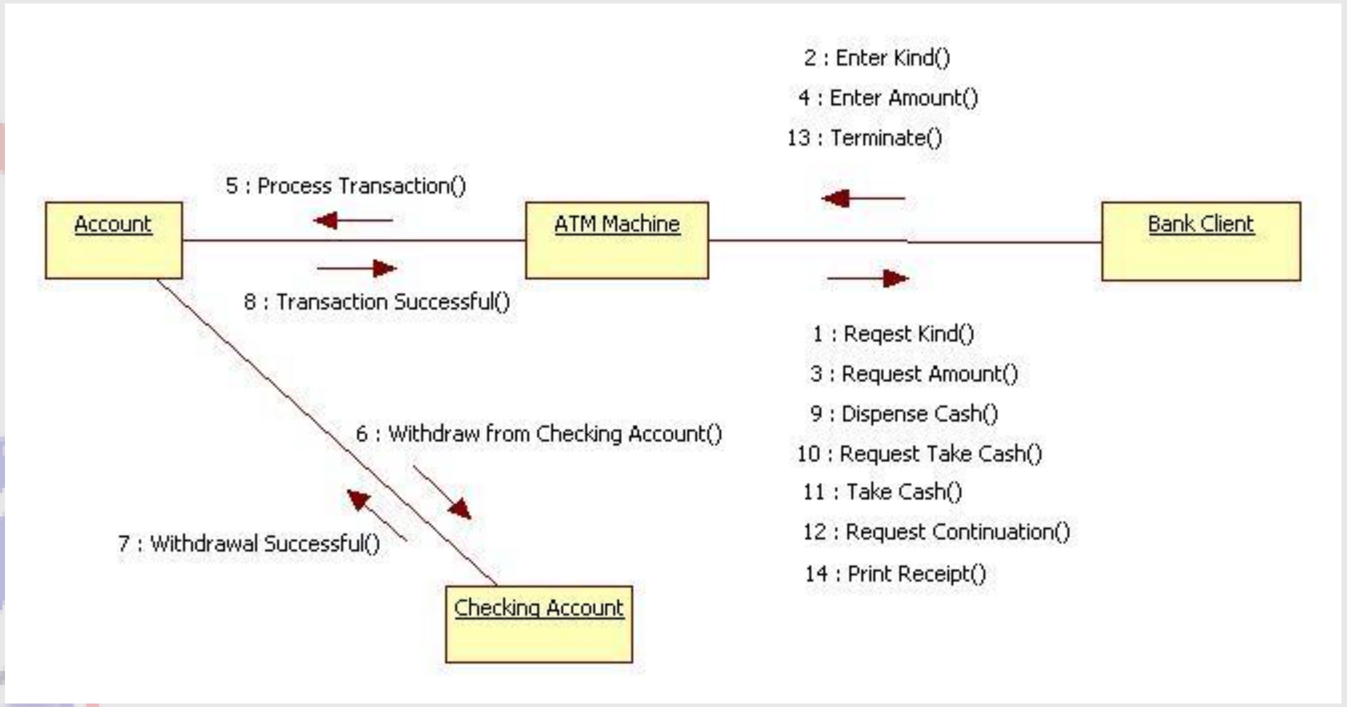


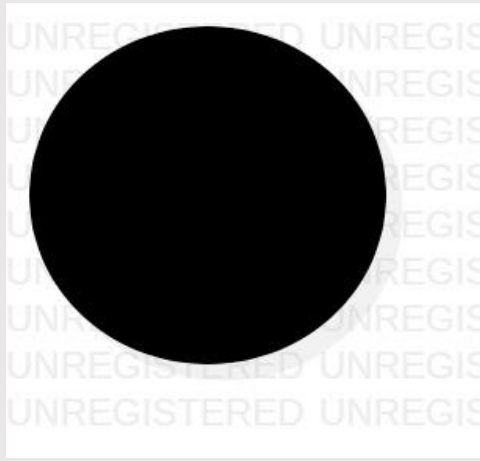
Fig: Collaboration Diagram of ATM

Activity Diagram

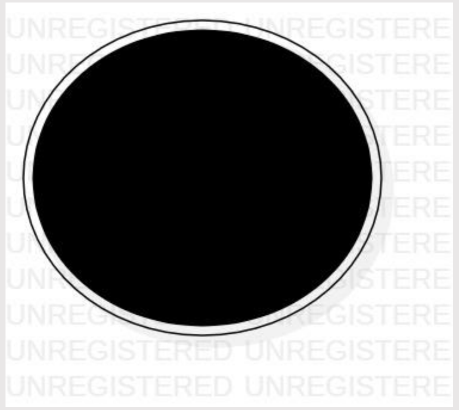
- An activity diagram visually presents a series of actions or flow of control in a system similar to a flowchart or a data flow diagram.
- It is a diagram to represent the flow of control among the activities in a system.
- It describe how activities are coordinated to provide a service which can be at different levels of abstraction.
- It focuses on the execution and flow of the behavior of a system instead of implementation.

Symbols/Notations

- **Initial State:**
  - ✓ A filled circle followed by an arrow represents the object's initial state.
  - ✓ It indicates the initial stage or beginning of the set of actions.

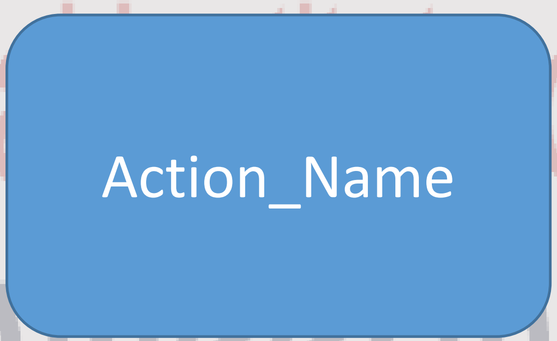


- **Final State:**
  - ✓ It is the stage where all the control flows and object flows end.
  - ✓ An arrow pointing to a filled circle nested inside another circle represents the object's final state.



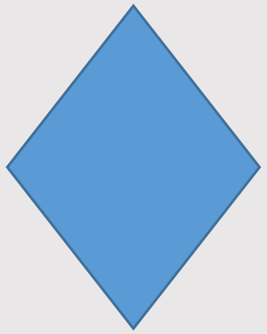
Symbols/Notations

- **Action Box**: An activity represents execution of an action on objects or by objects. The users or software perform a given task.



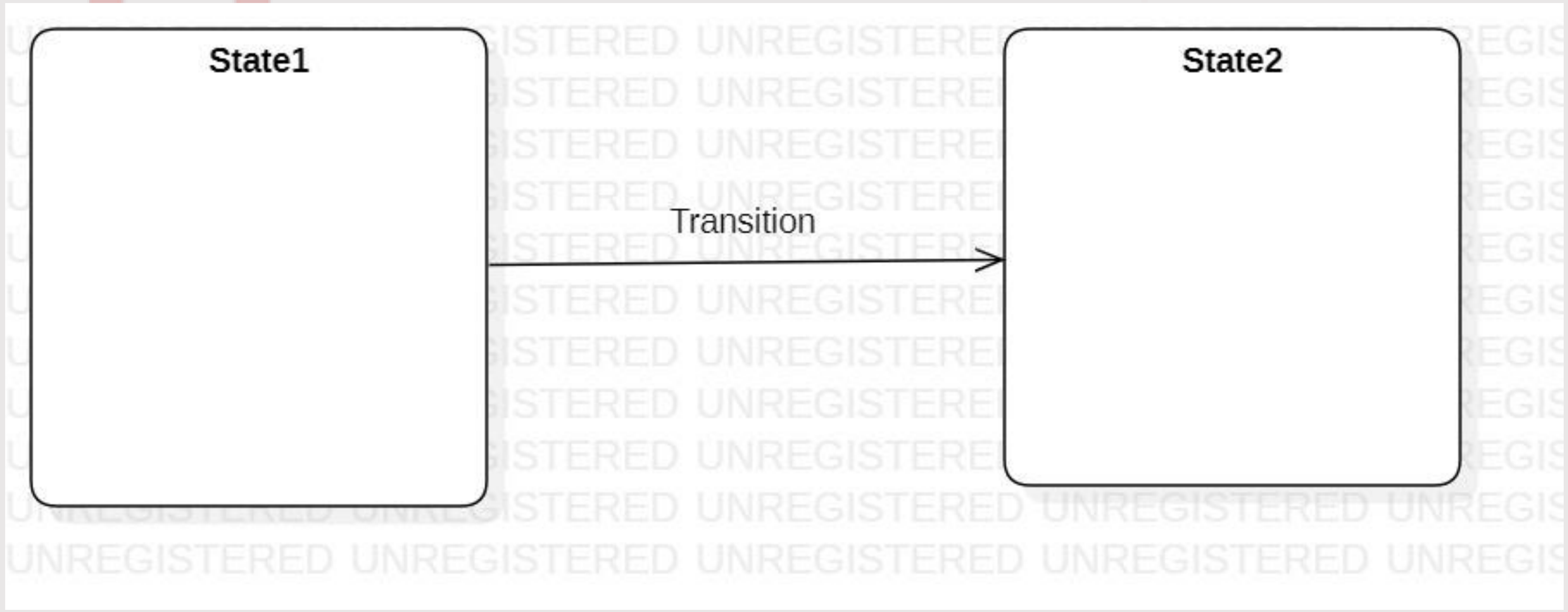
Decision Box:

It is a diamond shape box which represents a decision with alternate paths. It represents the flow of control.



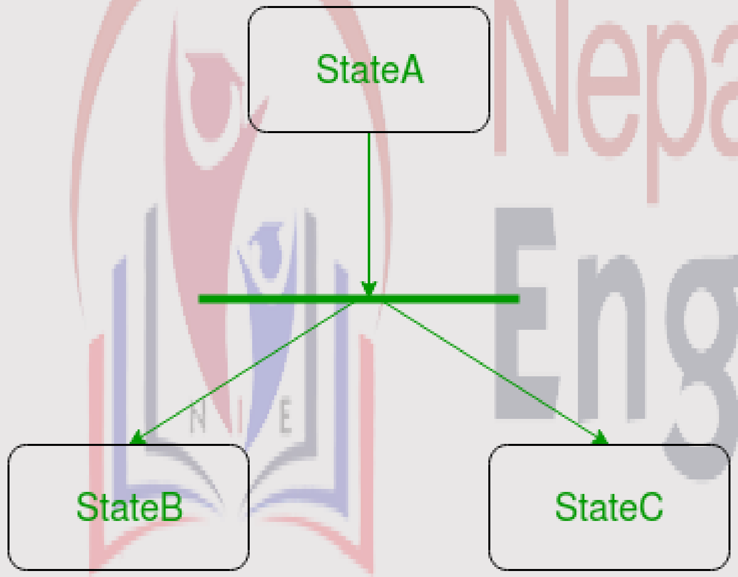
Symbols/Notations

- **Action Flow:** A solid arrow represents the path between different states of an object. Label the transition with the event that triggered it and the action that results from it. A state can have a transition that points back to itself.

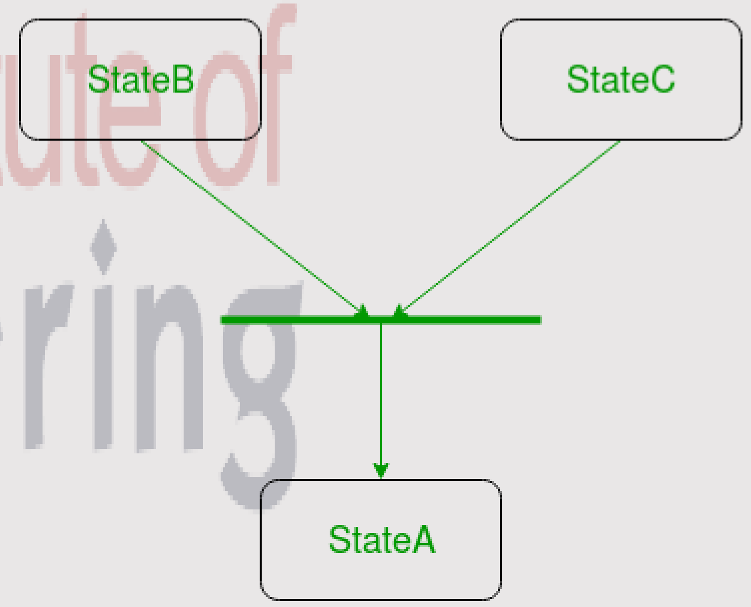


Symbols/Notations

- **Synchronization and Splitting of Control**
  A short heavy bar with two transitions entering it represents a synchronization of control. The first bar is often called a fork where a single transition splits into concurrent multiple transitions. The second bar is called a join, where the concurrent transitions reduce back to one.



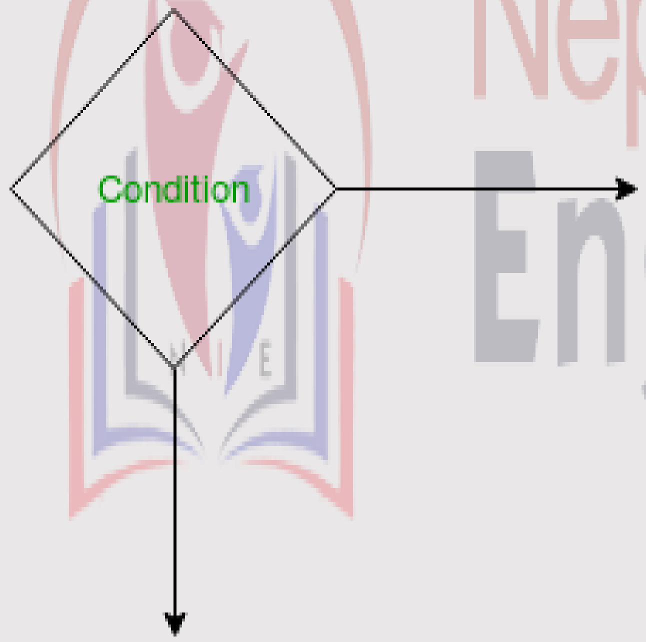
Fork

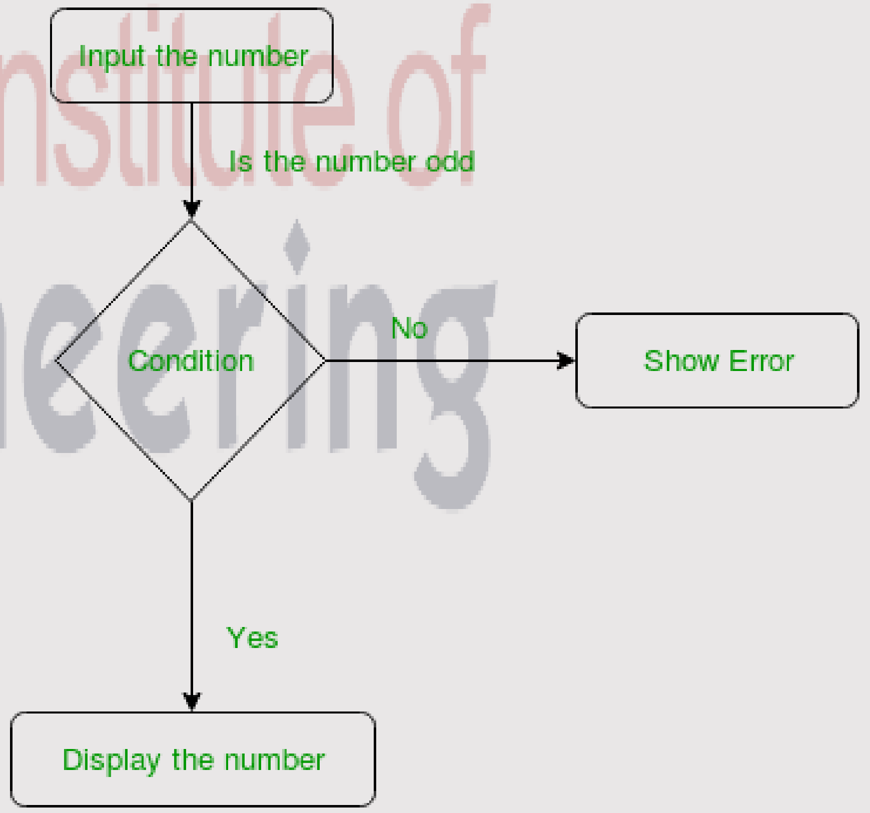


Join

Symbols/Notations

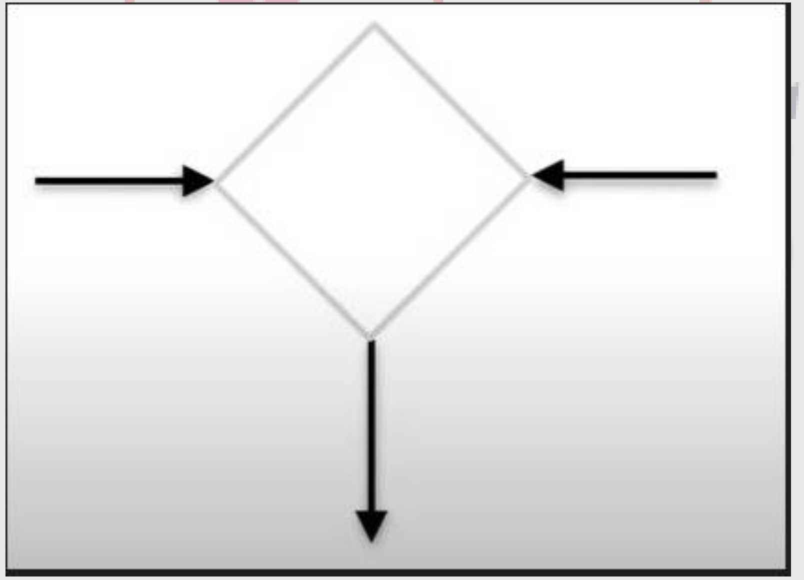
- **Decision node and Branching :** When we need to make a decision before deciding the flow of control, we use the decision node.

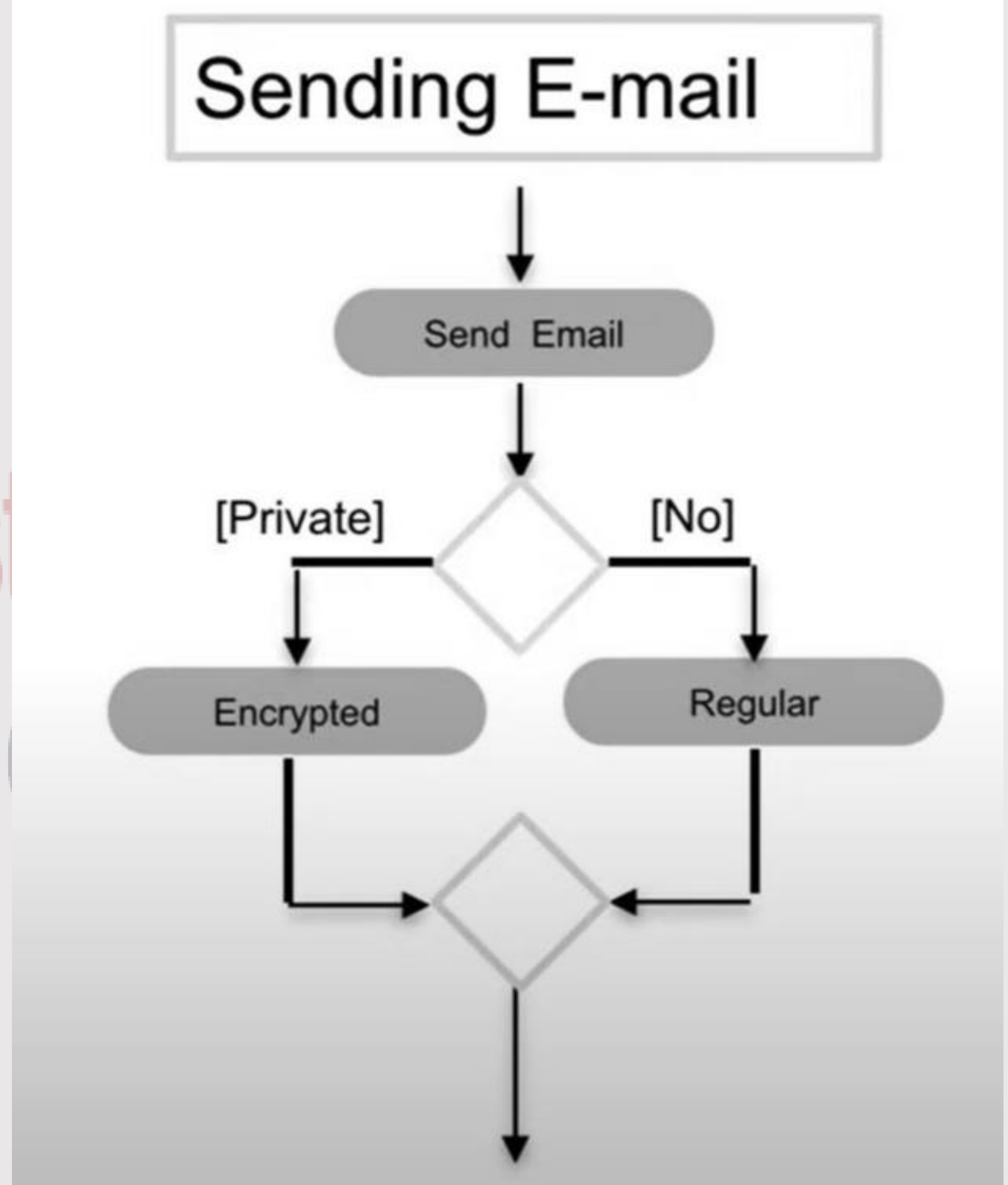




Symbols/Notations

- **Merge Node** : Two activities are merged with condition and only one activity flows forward.





Example of Activity Diagrams

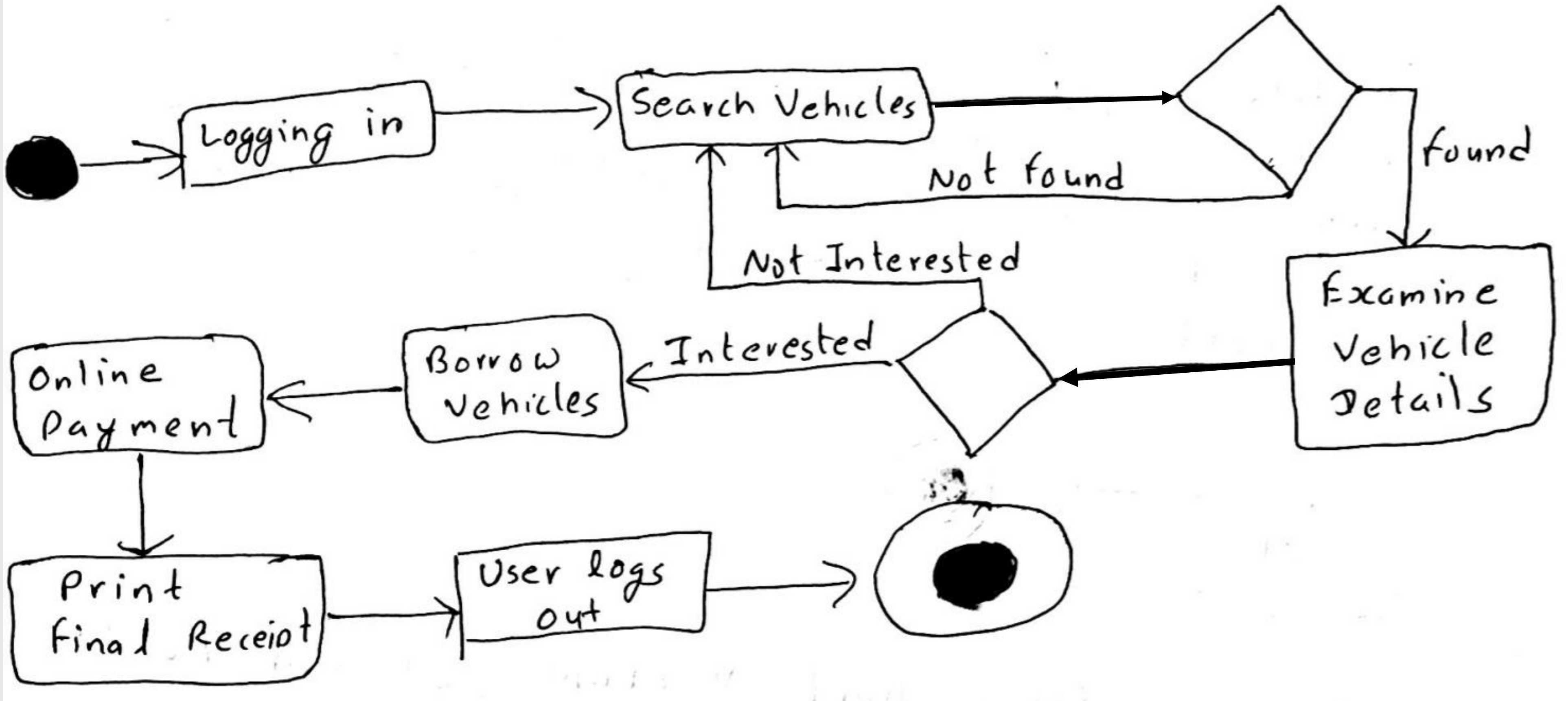


fig: Activity Diagram Of Car Rental System

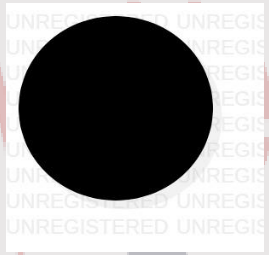
State Chart Diagram

- Also called: **State Chart, State Machines, State machine diagram**
- A **state diagram** is used to represent the condition of the system or part of the system at finite instances of time.
- Statechart diagrams are used to describe various states of an entity within the application system.
- Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered.

Symbols/Notations

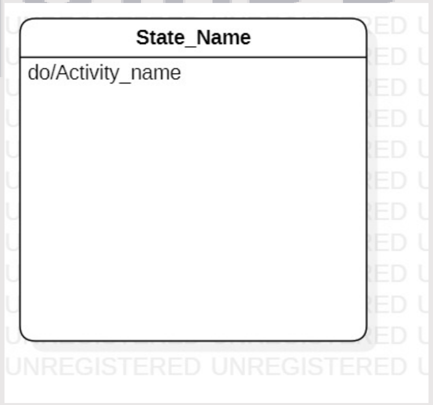
Initial State:

This shows the starting point of state chart diagram that is where the activity starts.



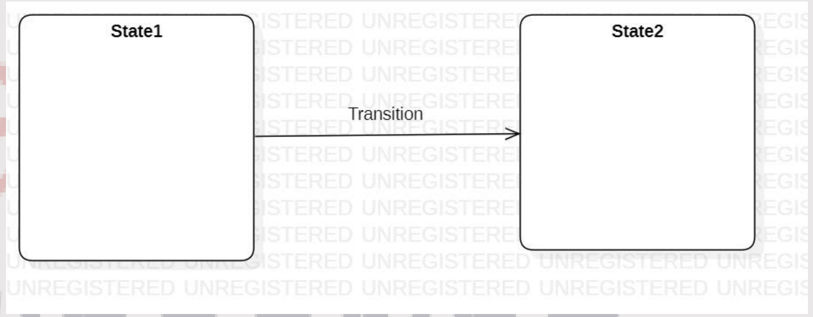
States:

States represent situations during the life of an object in which some action is performed.

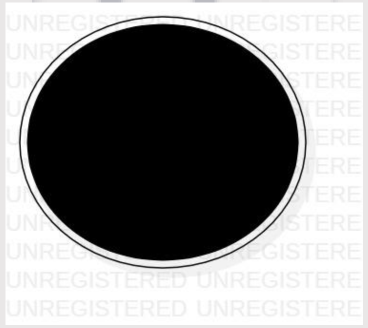


Symbols/Notations

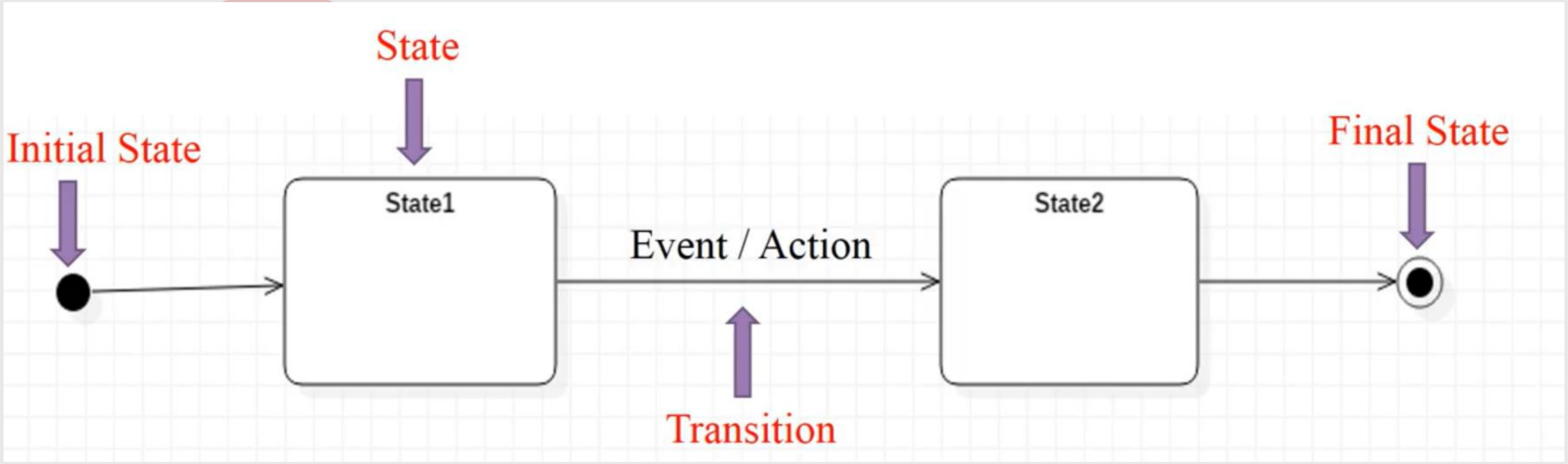
- Transition: It is relationship between two states which indicates that an object in the first state will enter the second state and performs certain specified actions.



- Final state: The end of the state chart diagram is represented by solid circle surrounded by a circle.



Example of State Chart Diagrams



Example of State Chart Diagrams

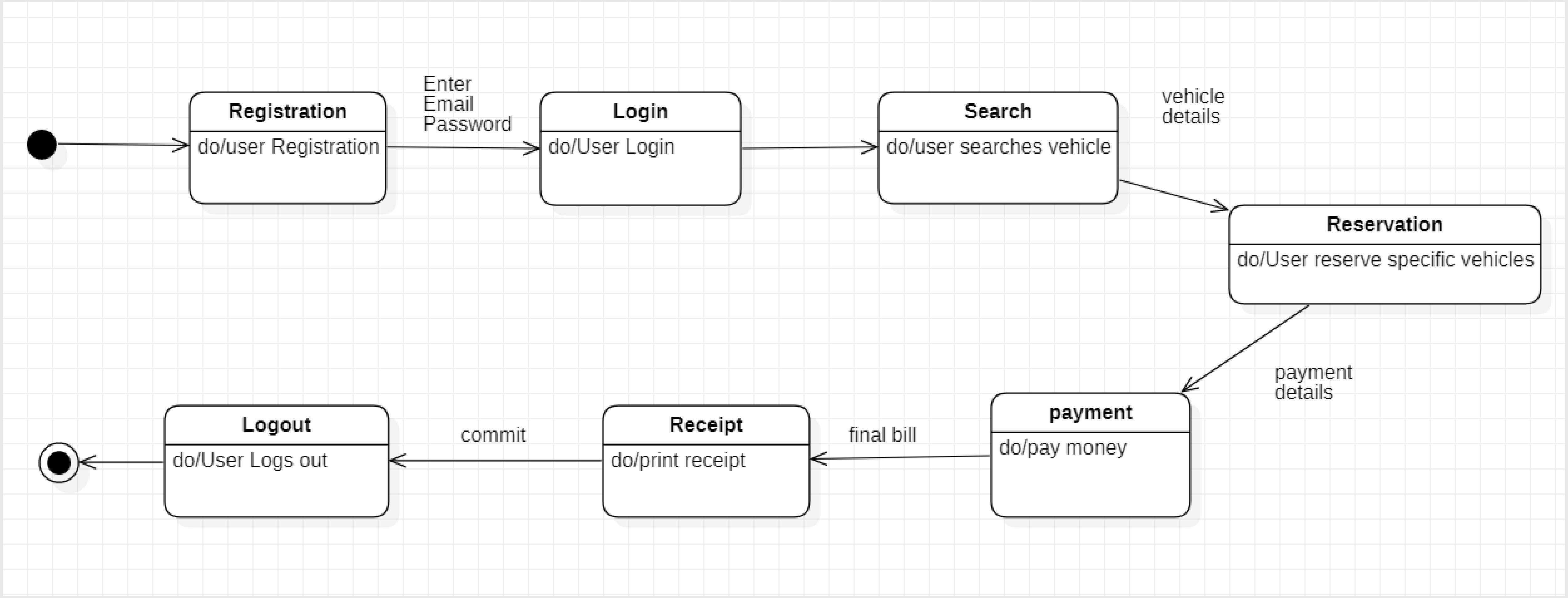


Fig: State Chart Diagram of Car Rental System

Analysis to Design

- The requirement study & OOA focuses on learning to “do the right thing” i.e. understanding some of the goals for the problem domain
- By contrast, the design work stresses “ do the right thing”, that is, skillfully designing a solution to satisfy the requirements.
- If we follow UP guidelines, perhaps 10% of the requirements were investigated during inception & slightly deeper investigation in first iteration of elaboration.
- Then further emphasis towards designing a solution for this iteration in terms of collaborating software objects

Analysis to Design

- Here in each iteration, the focus from requirement analysis is shifted to the design.
- These discoveries will both clarify the purpose of the design work of this iteration &
- Refine the requirements understanding for future iterations.
- Over the course of these early elaboration iterations, the requirement discovery will stabilize.
- So, by the end of elaboration perhaps 80% of the requirements are reliably defined in detail.

Object Oriented Analysis to Design

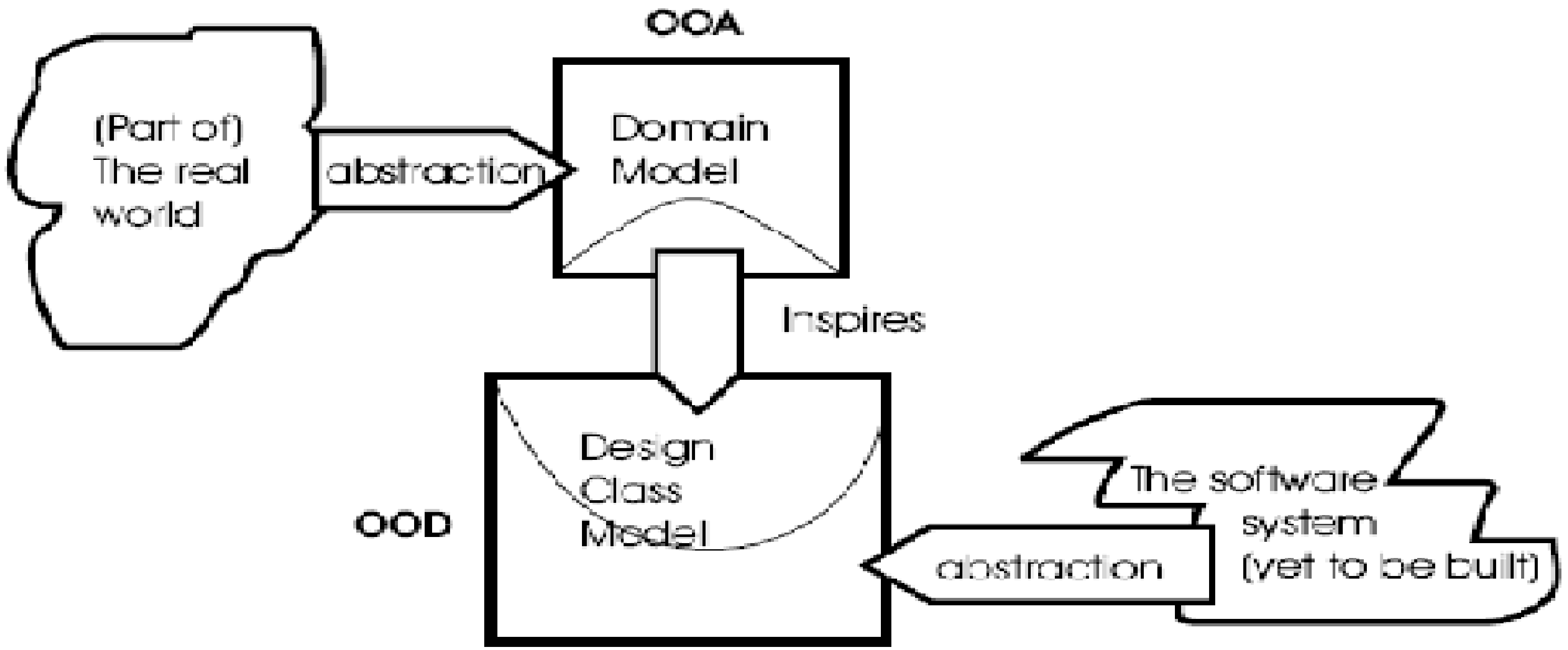


Fig: The phases of OOAD and how the transition from OOA to OOD works

Object Oriented Design

- Object-oriented design is the discipline of defining the objects and their interactions to solve a problem that was identified and documented during object-oriented analysis.
- OOD transforms the analysis model created using OOA into a design model that serves as a blueprint for software construction.
- OOD results in a design that achieves a number of different levels of modularity.
- Subsystems: Major system components.
- Objects: Data and the operations.
- Four important software design concepts:
  - Abstraction
  - Information Hiding
  - Functional Independence
  - Modularity

Input (sources) for object-oriented design

- Problem Statement/ Requirement Documentations
- Use Case Diagram
- Conceptual Model
- Relational Data Model(If Applicable)

Output (deliverables) of object-oriented design

- Class Diagram
- Object Diagram
- Component Diagram
- Sequence Diagram
- Activity Diagram
- State Chart Diagram

Patterns

- A general repeatable solution to a commonly occurring problem in software design
- Design patterns are programming language independent strategies for solving a common problem i.e., represents an idea, not a particular implementation.
- A design pattern isn't a finished design that can be transformed directly into code. A design pattern is a description of template for how to solve a problem that can be used in many different solutions and scenarios.
- A pattern is usually described in four components; these components explain what it is about and how it is to be used.

Components of Patterns

A pattern is usually described in four components; these components explain what it is about and how it is to be used.

- **Pattern name:** The pattern name is used to identify the pattern once it has been introduced. It is a way to communicate the pattern to other people and is therefore vital in spreading its reach, this is a fact mentioned in.
- **Problem description:** In this section, the problem is described that is the reason for applying the pattern. It may be accompanied by a list of preconditions that must be fulfilled – only when these conditions are met, the pattern is applied.
- **Solution:** Here the workings of the pattern are explained: which classes interact when and how these interactions are achieved. The description is on an abstract level to make sure that it can be applied in many situations.
- **Consequences**: Here the effects of the pattern are explained. This might be both advantages and disadvantages of applying the pattern. The consequences are often related to the impact on flexibility, extensibility and portability the application of the pattern has

GRASP Pattern

- **General Responsibility Assignment Software Patterns (or Principles)**
- A set of patterns for assigning responsibilities to software objects
- It helps in guiding object-oriented design by clearly outlining WHO does WHAT.
- Which object or class is responsible for what action or role?
- GRASP also helps us define how classes work with one another. The key point of GRASP is to have efficient, clean, understandable code

GRASP Pattern

1. Information Expert (or Expert)
2. Creator
3. Low Coupling
4. High Cohesion
5. Controller
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

1. Information Expert

- **Problem**: Which class should be responsible for doing certain things?
- **Solution**: Assign responsibility to the information expert – the class that has the information necessary to fulfill the required responsibility. The expert pattern expresses the common intuition that objects do things related to the information they have

POS Example

- Let's consider **Point of Sale** (POS) application: Here is a brief overview of the **POS** application.
- POS application for a shop, restaurant, etc. that registers sales.
- Each sale is of one or more items of one or more product types and happens at a certain date.
- A product has a specification including a description, unitary price, an identifier.
- The application also registers payments (say, in cash) associated with sales.
- Payment is for a certain amount, equal to or greater than the total of the sale.

POS System

- POS is known as Point of Sale.
- POS is an intermediary system used to complete the online transaction between merchant and buyer.
- It is an electronic hardware system, where customer gets the calculated total amount of the acquired services and also have the payment system enabled into it.
- In Nepal also, there are many POS system utilized in shopping centers, banks, schools, grocery store and many other places.
- Never be confuse that, POS is not only a ATM swapping machine seen in shopping centers but it is also a complete billing system seen like in Bhatbhateni super market.

POS System

- POS system is coupled with hardware and software interconnected making it function properly. The users simply swap the ATM card into this machine, enter their card pin number and the payment is done.
- In Nepal, commercial banks are leading in providing this system to the merchants.

Payment made Easy !

Use your debit or credit card at any Point of Sale (POS) device

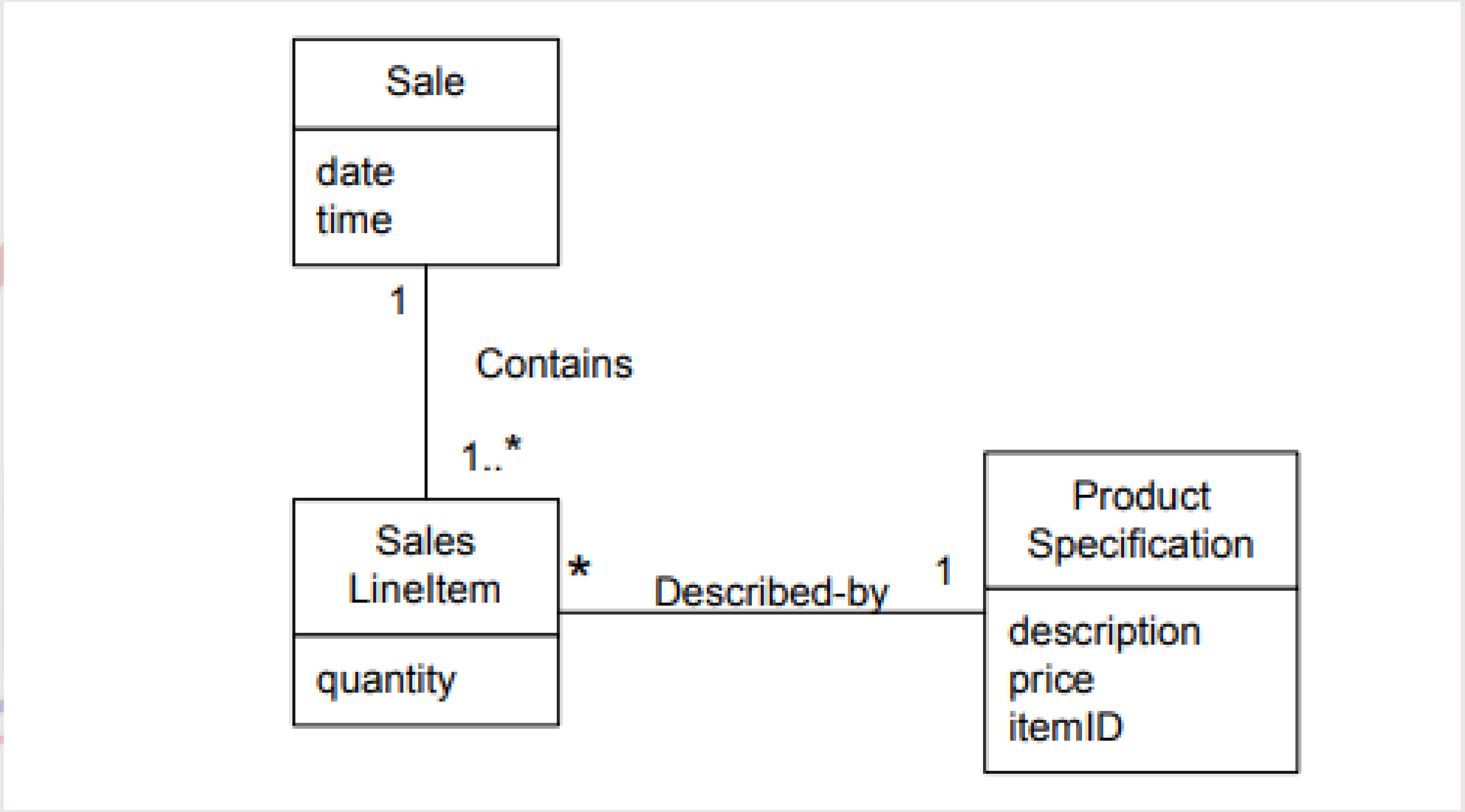
- Cash less transaction
- Higher purchasing power
- Global visa card accepted.
- Easy cash management
- Convenient and Secure mode of payment

हामी संग-संगै

1. Information Expert

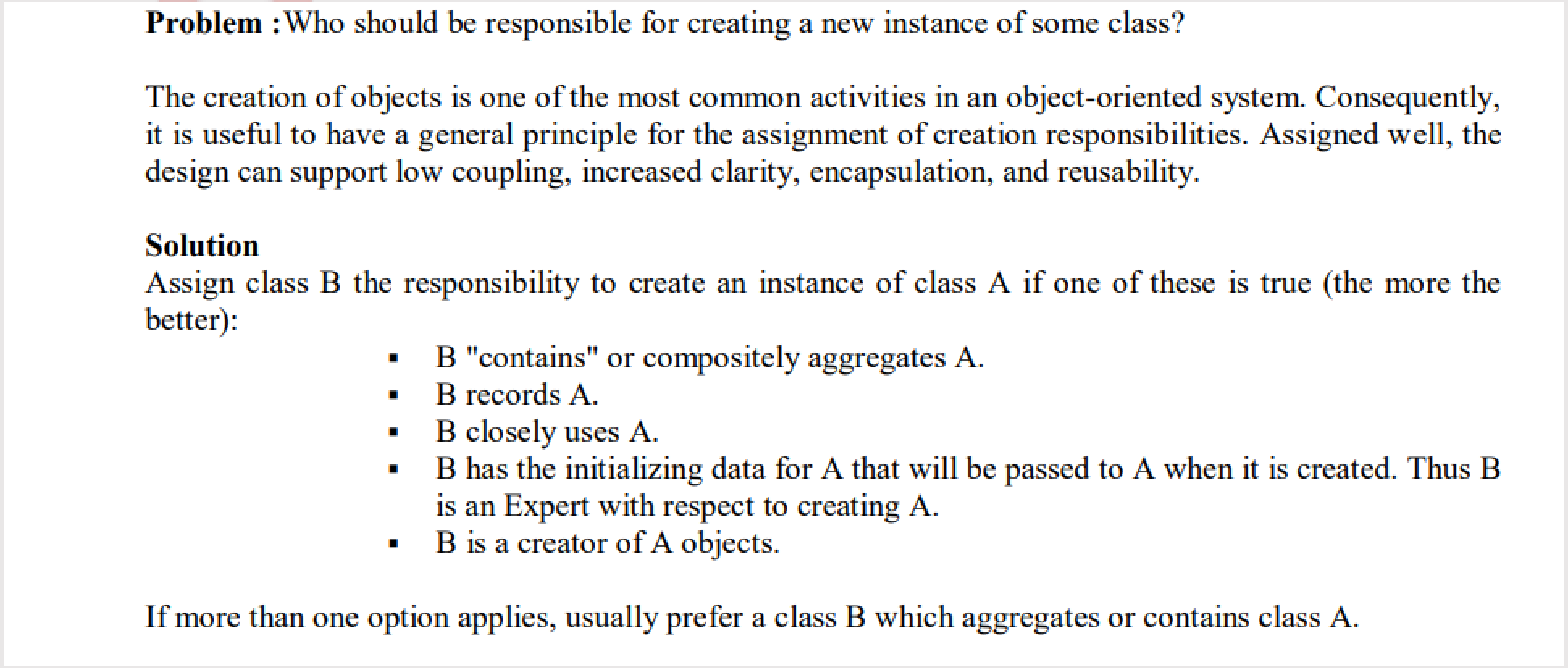
- In the POS application, some class needs to know the grand total of a sale.
- Start assigning responsibilities by clearly stating the responsibility. By Information Expert, we should look for that class of objects that has the information needed to determine the total. It is necessary to know about all the *SalesLineItem* instances of a sale and the sum of their subtotals.
- A *Sale* instance contains these; therefore, by the guideline of Information Expert, *Sale* is a suitable class of object for this responsibility; it is an information expert for the work.

1. Information Expert



2. Creator

- Creator is a GRASP pattern which helps to decide which class should be responsible for creating a new instance of a class.
- Object creation is an important process, and it is useful to have a principle in deciding who should create an instance of a class.



Problem : Who should be responsible for creating a new instance of some class?

The creation of objects is one of the most common activities in an object-oriented system. Consequently, it is useful to have a general principle for the assignment of creation responsibilities. Assigned well, the design can support low coupling, increased clarity, encapsulation, and reusability.

Solution
Assign class B the responsibility to create an instance of class A if one of these is true (the more the better):

- B "contains" or compositely aggregates A.
- B records A.
- B closely uses A.
- B has the initializing data for A that will be passed to A when it is created. Thus B is an Expert with respect to creating A.
- B is a creator of A objects.

If more than one option applies, usually prefer a class B which aggregates or contains class A.

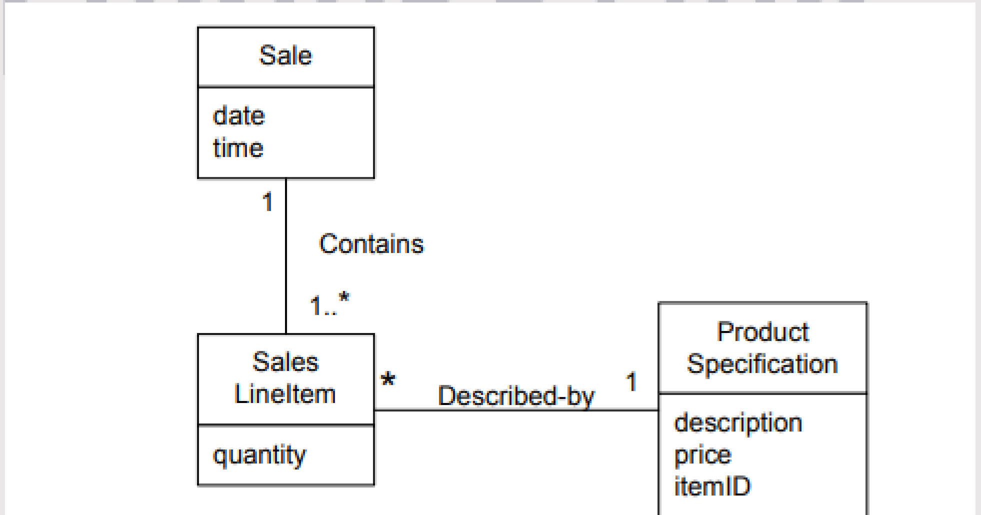
Creator Example

Problem statement

- Who should be responsible for creating a *SalesLineItem* instance?

Solution

- In the **POS** application, who should be responsible for creating a *SalesLineItem* instance? By **Creator**, we should look for a class that aggregates, contains, and so on, *SalesLineItem* instances.
- From the diagram, The *Sale* class contains (in fact, aggregates) many *SalesLineItem* objects, the **Creator pattern** suggests that *Sale* class is a good candidate to have the responsibility of creating *SalesLineItem* instances.



3. Low Coupling

- Coupling is the measure of low strongly one element is connected to the other elements.
- An element if does not depend on too many other elements like classes, subsystems systems it is having low coupling.
- A class with high coupling relies on many other classes.
- Low coupling tends to reduce time, effort and defects in modifying software.
- **Problem**: How to support low dependency, low change impact, and increased reuse?
- **Solution**: Assign a responsibility so that coupling remains low.

3. Low Coupling

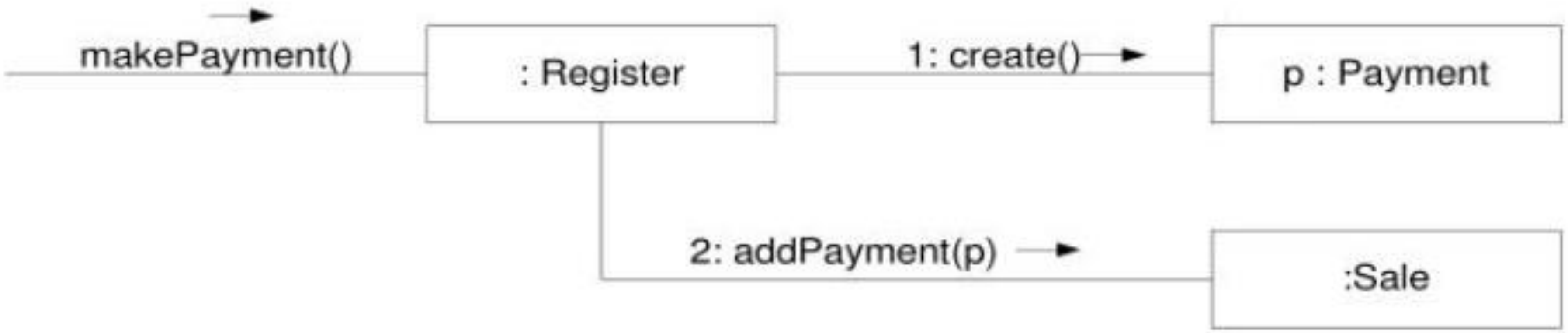
Example (from POS system):

Consider Payment, Register, Sale

Need to create a Payment and associate it with a Sale, who is responsible?

Creator => since a Register “records” a Payment it should have this responsibility

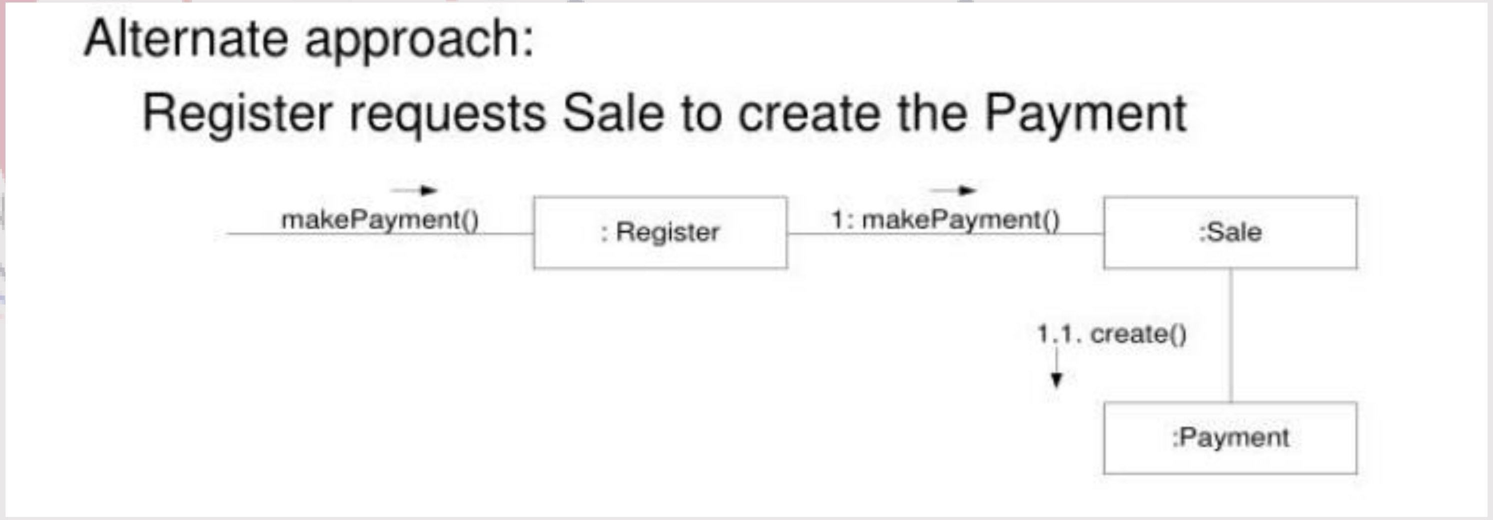
Register creates Payment *p* then sends *p* to a Sale => coupling of Register class to Payment class



3. Low Coupling

Consider Coupling in two approaches:

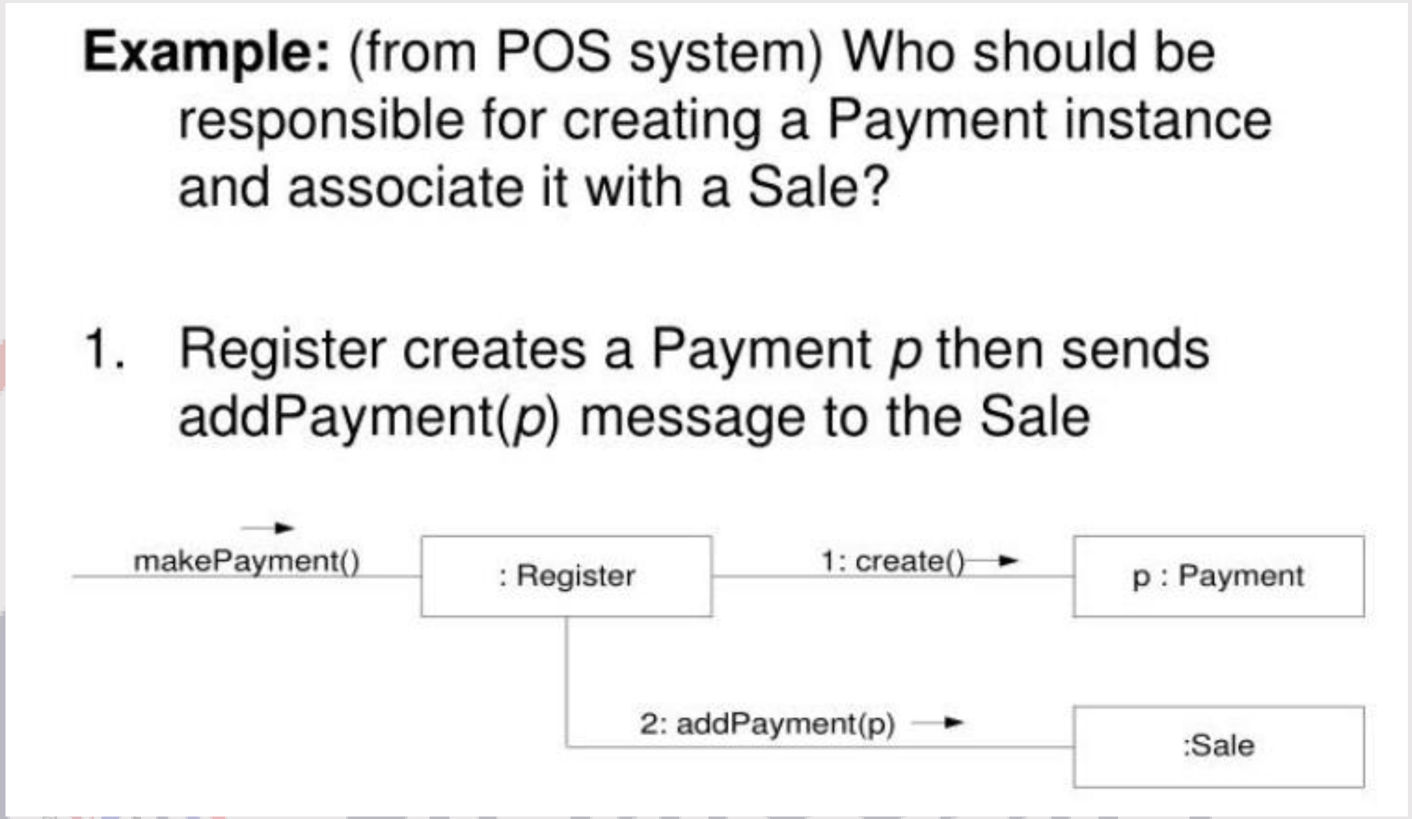
- In both cases, sale need to know about a payment.
- However a register needs to know about a payment first but not in second
- Second approach has lower coupling.



4. High Cohesion

- **Cohesion** is used to indicate the degree to which a class has a single, well-focused responsibility.
- **Cohesion** is a measure of how the methods of a class or a module are meaningfully and strongly related and how focused they are in providing a well-defined purpose to the system.
- **Problem:** How to keep classes focused, understandable and manageable?
- **Solution:** Assign responsibilities so that cohesion remains high. Try to avoid classes to do too much different things.

4. High Cohesion



Example: (from POS system) Who should be responsible for creating a Payment instance and associate it with a Sale?

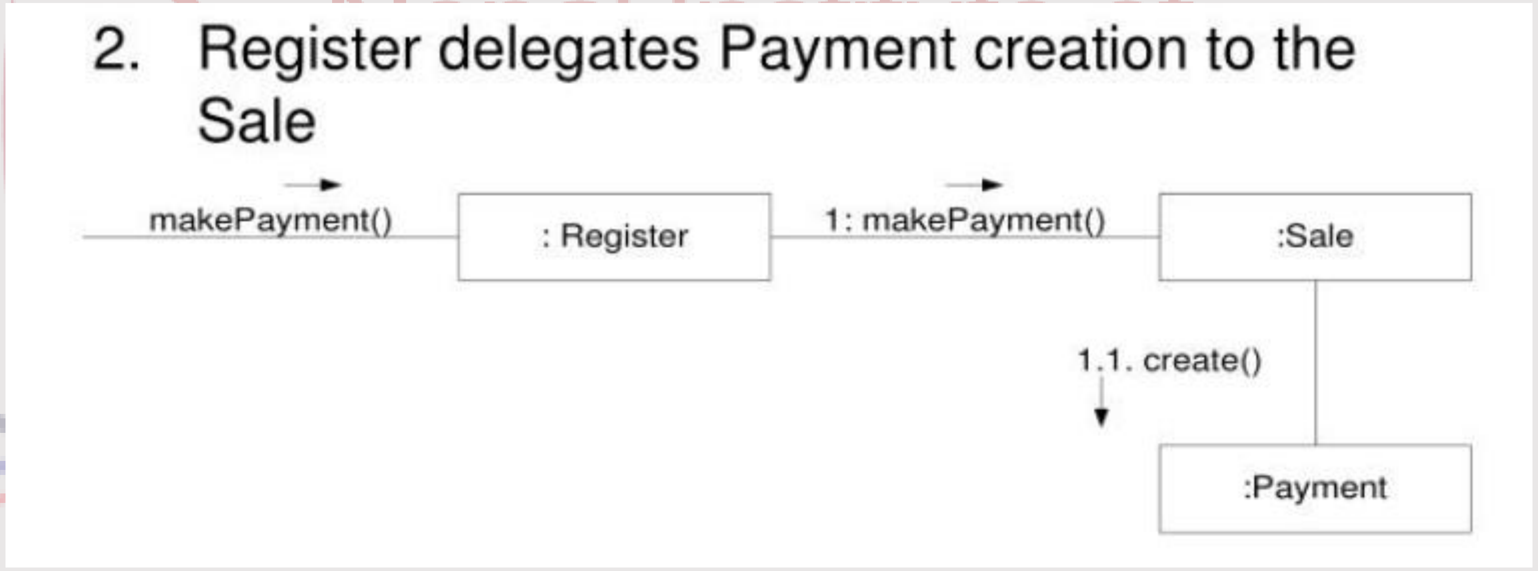
1. Register creates a Payment *p* then sends addPayment(*p*) message to the Sale

- Register is responsible for system operation makePayment()

- In isolation no problem
- But if we start assigning additional system operations to Register then will violate high cohesion.

4. High Cohesion

- This second approach will lead to high cohesion for Register class.
- **Note:** This design supports both low coupling and high cohesion
- High cohesion like low coupling is an evaluation principle.


2. Register delegates Payment creation to the Sale

5. Controller

- A controller is defined as the first object beyond the user interface (UI) layer that is responsible for the receiving or handling a system operation message.
- **Example**:

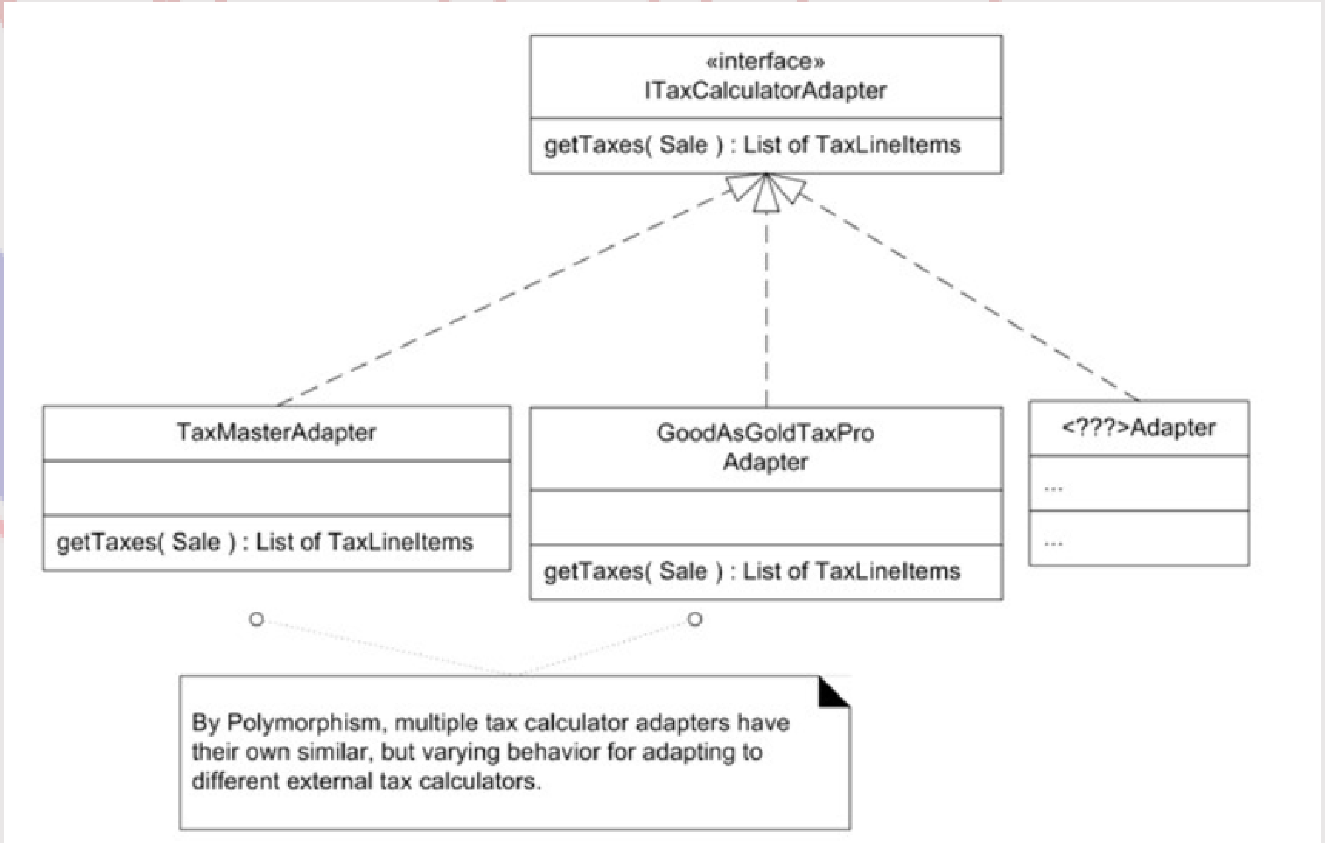
1) A cashier is POS terminal pressing ―EndSale‖ button indicating ―sale has ended
2) Writer using word processor presses ―spell check‖ button to perform checking of spelling

5. Controller

- **Problem**: Who should be responsible for handling an input system event?
- **Solution:** Assign the responsibility for receiving and/or handling a system event to one of the following choices:
  - Objects that represent overall system, device or subsystem
  - Objects that represent a use case scenario within the system event occurs.

6. Polymorphism

- Polymorphism extends beyond inheritance; it's about working with different types through a shared interface.
- Let’s say you like to sale NextGen POS system to different countries. You need to integrate different external third- party tax calculators such as TaxMasterAdapter GoodAsGoldTaxProAdapter etc.



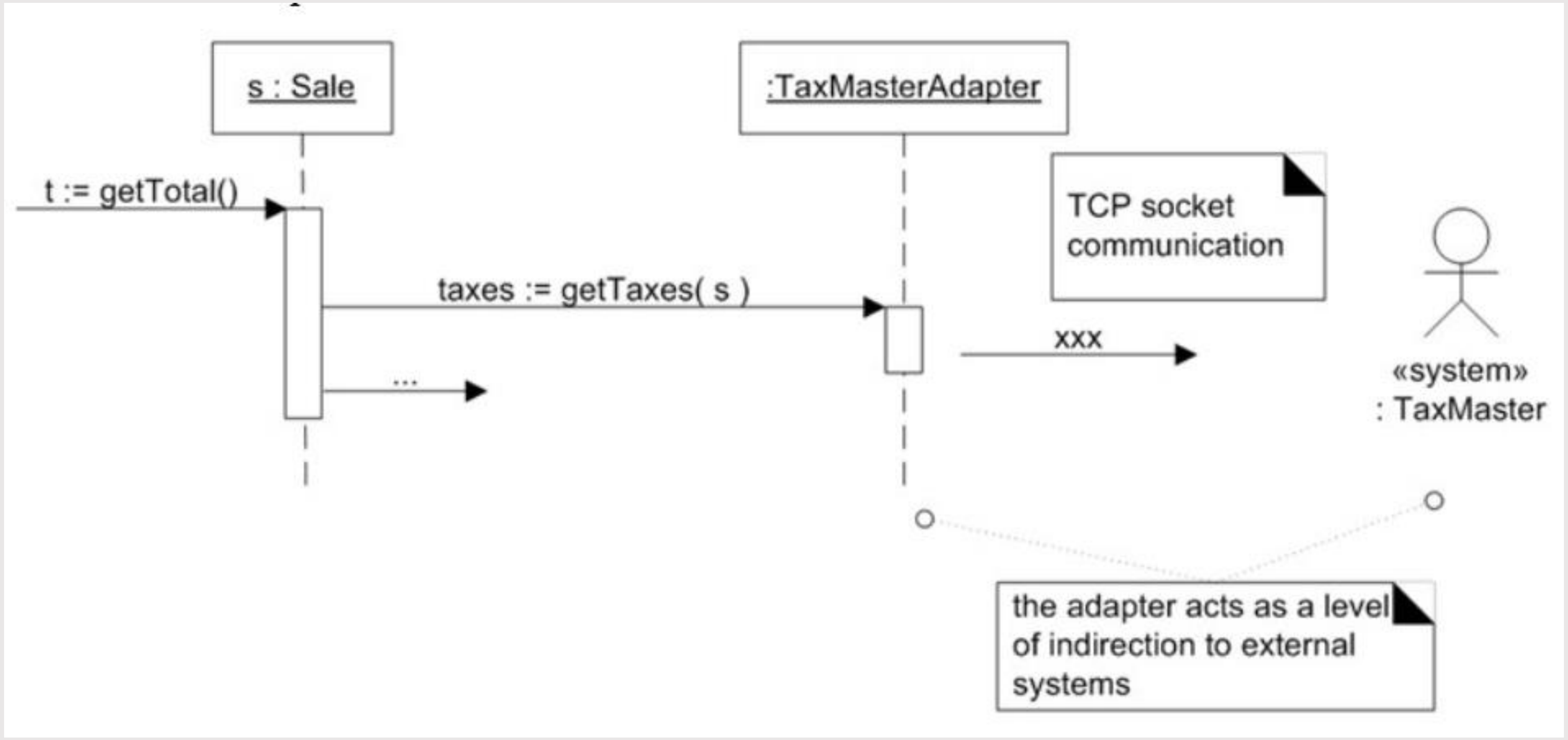
«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List of TaxLineItems
TaxMasterAdapter
getTaxes(Sale) : List of TaxLineItems
GoodAsGoldTaxPro Adapter
getTaxes(Sale) : List of TaxLineItems
<???>Adapter
...
...

By Polymorphism, multiple tax calculator adapters have their own similar, but varying behavior for adapting to different external tax calculators.

7. Pure Fabrication

- What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate?
- Solution: Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept – something made up, to support high cohesion, low coupling, and reuse. Finally, a pure fabrication implies making something up, which we do when we're desperate!

7. Pure Fabrication



8.Indirection

- Involves creating a "middleman" class (like a Controller) in the application.
- Adding this "extra" class can surprisingly simplify the application.
- Controllers reduce direct connections between objects.
- Results in improved code reusability and readability.
- Aids in achieving Low coupling.

9. Protected Variations

- How to design objects, subsystems and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
- Solution: Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

Determining Visibility

- **Visibility** is the ability of an object to "see" or have a reference to another object. More generally, it is related to the issue of scope: Is one resource (such as an instance) within the scope of another?
- There are four common ways that visibility can be achieved from object A to object B:
  - **Attribute visibility**: B is an attribute of A.
  - **Parameter visibility**: B is a parameter of a method of A.
  - **Local visibility**: B is a (non-parameter) local object in a method of A.
  - **Global visibility**: B is in some way globally visible.

Attribute Visibility

- **Attribute visibility** from A to B exists when B is an attribute of A.
- It is a relatively permanent visibility because it persists as long as A and B exist. This is a very common form of visibility in object-oriented systems.

Parameter Visibility

- Parameter visibility from A to B exists when B is passed as a parameter to a method of A.
- It is a relatively temporary visibility because it persists only within the scope of the method.
- After attribute visibility, it is the second most common form of visibility in object-oriented systems.

Local Visibility

- **Local visibility** from A to B exists when B is declared as a local object within a method of A.
- It is a relatively temporary visibility because it persists only within the scope of the method.
- After parameter visibility, it is the third most common form of visibility in object-oriented systems.

Global Visibility

- Global visibility from A to B exists when B is global to A.
- It is a relatively permanent visibility because it persists as long as A and B exist.
- It is the least common form of visibility in object-oriented systems.

THANK
YOU