# Object Oriented Modeling And Designing(OOMD)

- Summary of all diagrams.
- for oral point of view.

Prepared By:-Er.Payal Dungarwal

# Introduction

- The **Unified Modeling Language**(UML) is one of the most exciting tools in the world of **system development** today
- The UML enables **system builder to create blueprint** that to capture their vision in standard, easy-to- understand way and communicate them to other
- **System:** It is combination of **software** and **hardware** that provides a solution for business problem

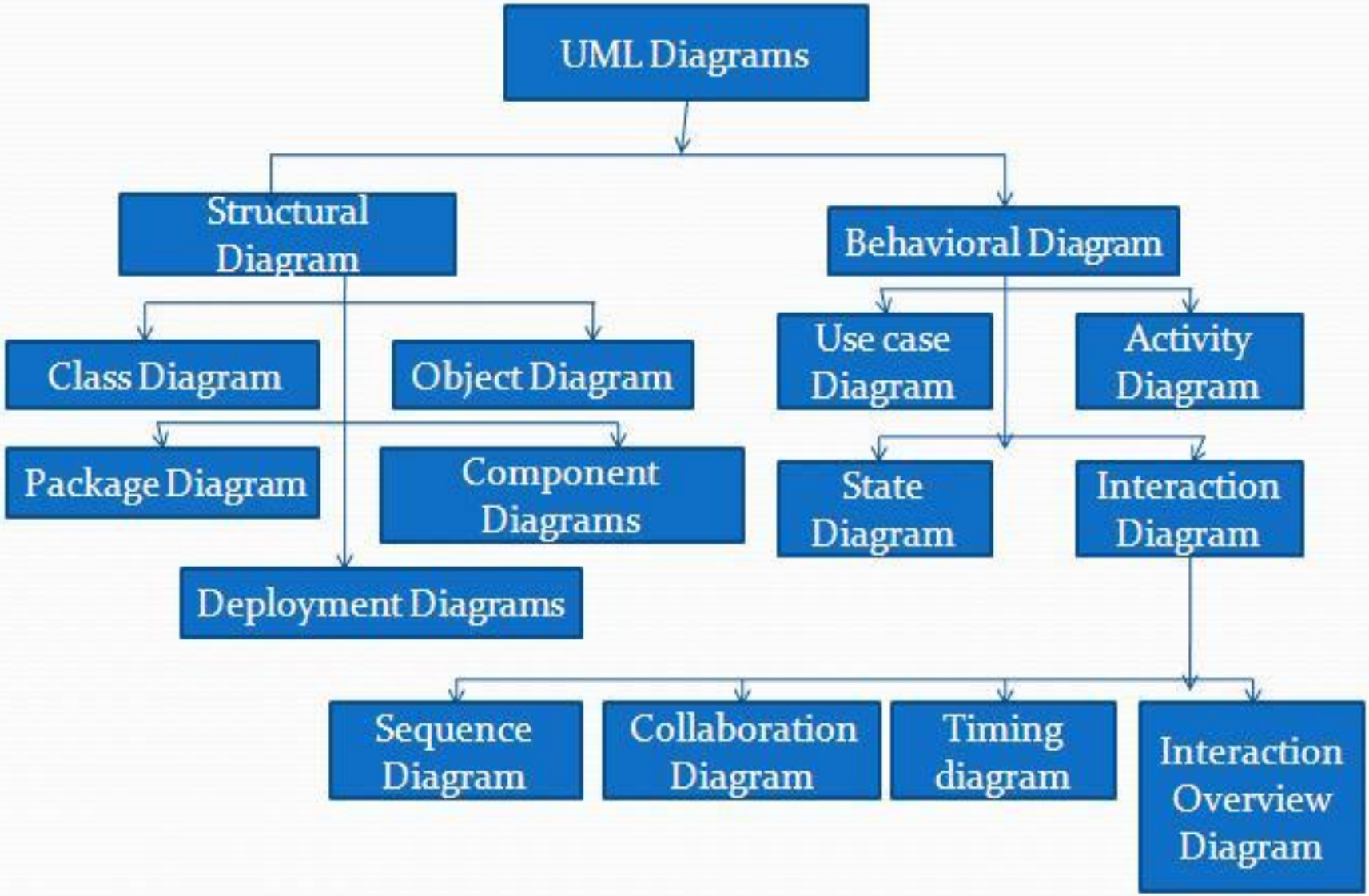
# Important Term

- **Client**: System development *is the creation of a system* for a client, the person has the *problem to be solved*
- **Analyst**: An Analyst *document the client's problem* and relays it to developer
- **Developer:** Programmers *who build the software* that solve the problem and *deploy the software* on computer hardware

# How the UML Came to be…

- The UML is the brainchild of **Grady Booch , James Rumbaugh** and **Ivar Jacobson**
- They are worked in Separate organizations ,In 1994 they was came together in **Rational Software Corporation**
- In 1997 they produced version 1.0 of UML and submitted it to the **OMG**
- Then the UML has become de **facto standard** in the *software Industry* and it continues to evolve

# UML Diagram Classification



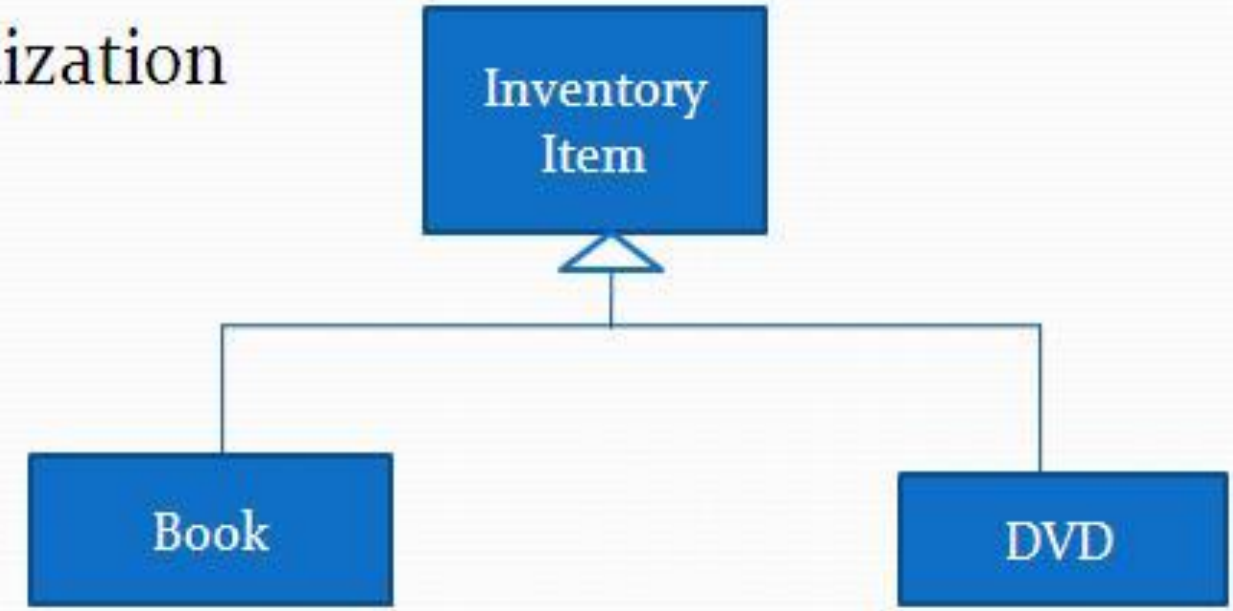
# Working with Relationships

- **Association**: When classes are connected together conceptually, that connection is called an **association**

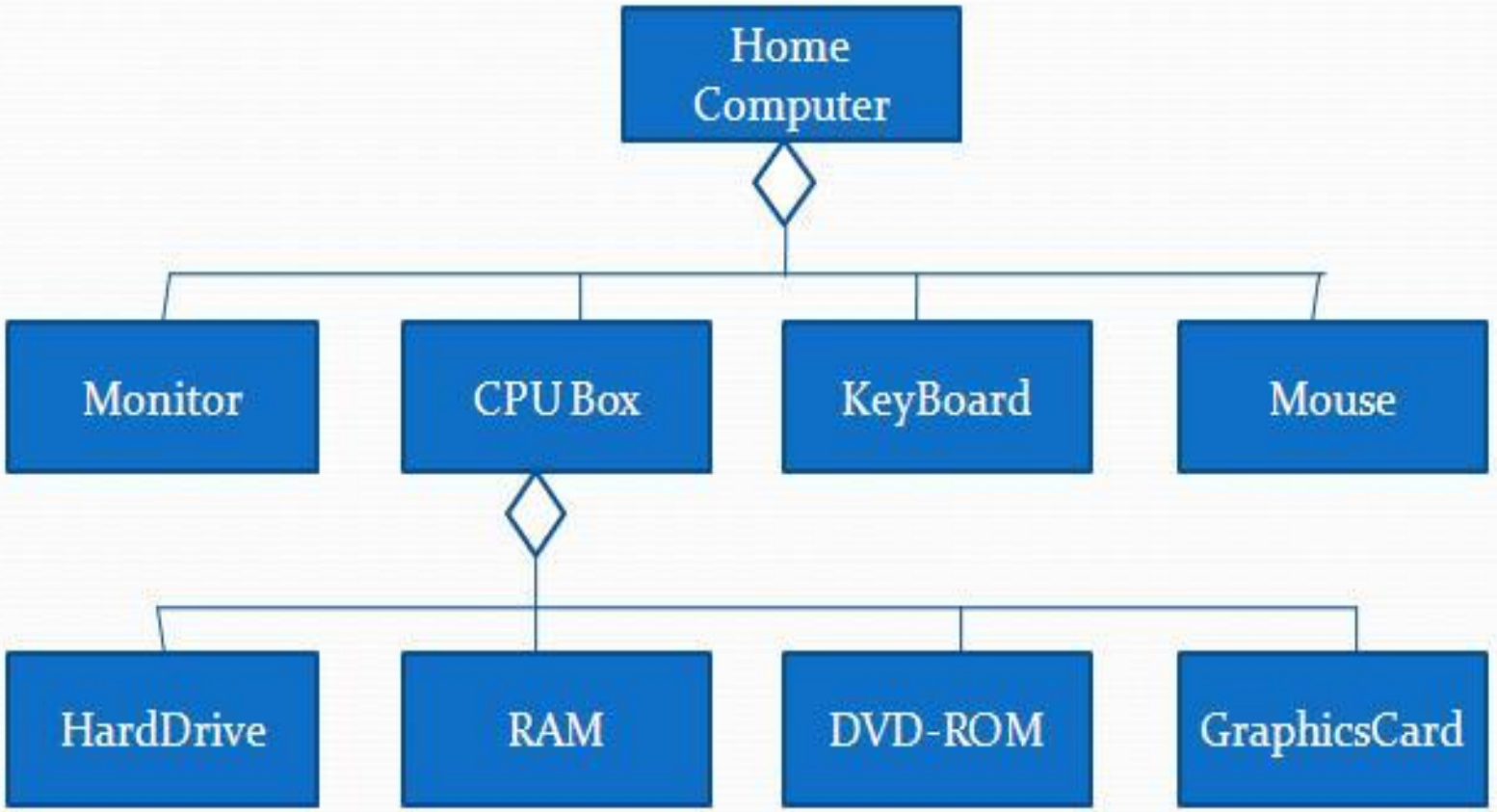
e.g.

- **Generalization**: One class (the child class or subclass) can inherit attributes and operations from another (the parent class or super class).
- The parent class is more general than the child class.
- **O**bject-orientation refers to this as **Inheritance** The UML also refers to this as **Generalization**
- This type of connection stands for the phrase **is a kind of**

e.g. Generalization



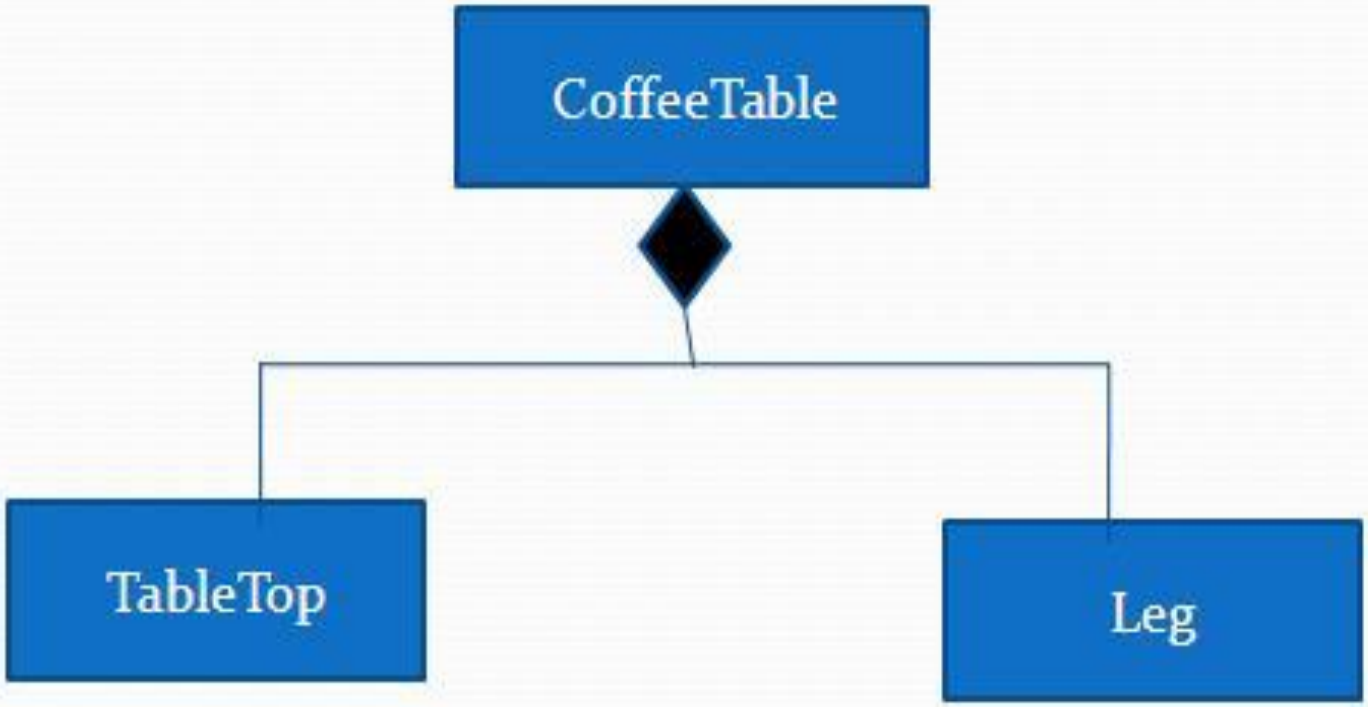
- **Aggregation: Sometimes** a class consists of a number of component classes. This is a special type of relationship called an **aggregation**
- The components and the class they constitute are in a *part-whole* association
- Example: Home Computer




Aggregation :Home Computer

- **Composites:** A **Composite** is a strong type of aggregation. Each component in a composite can belong to just one whole. The components of a coffee table—the tabletop and the legs—make up a composite. The symbol for a composite is the same as the symbol for an aggregation except the **diamond is filled**

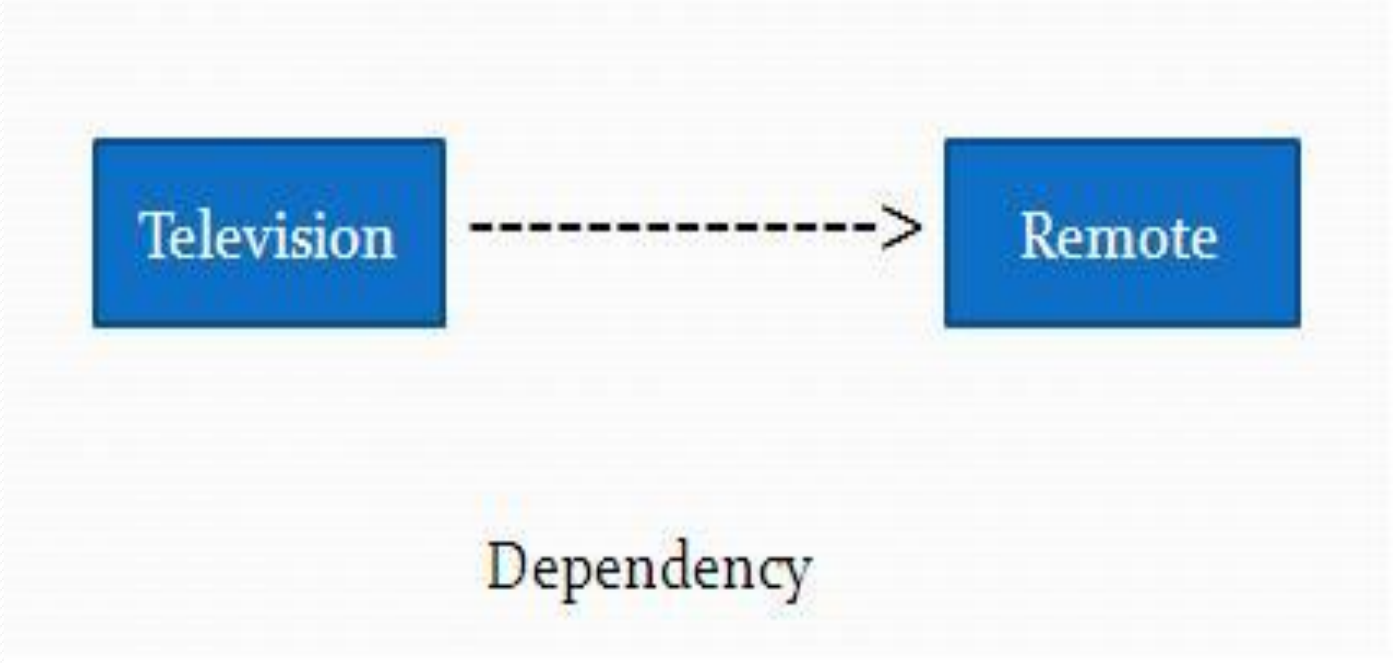
# Composition




Composite: Coffee Table

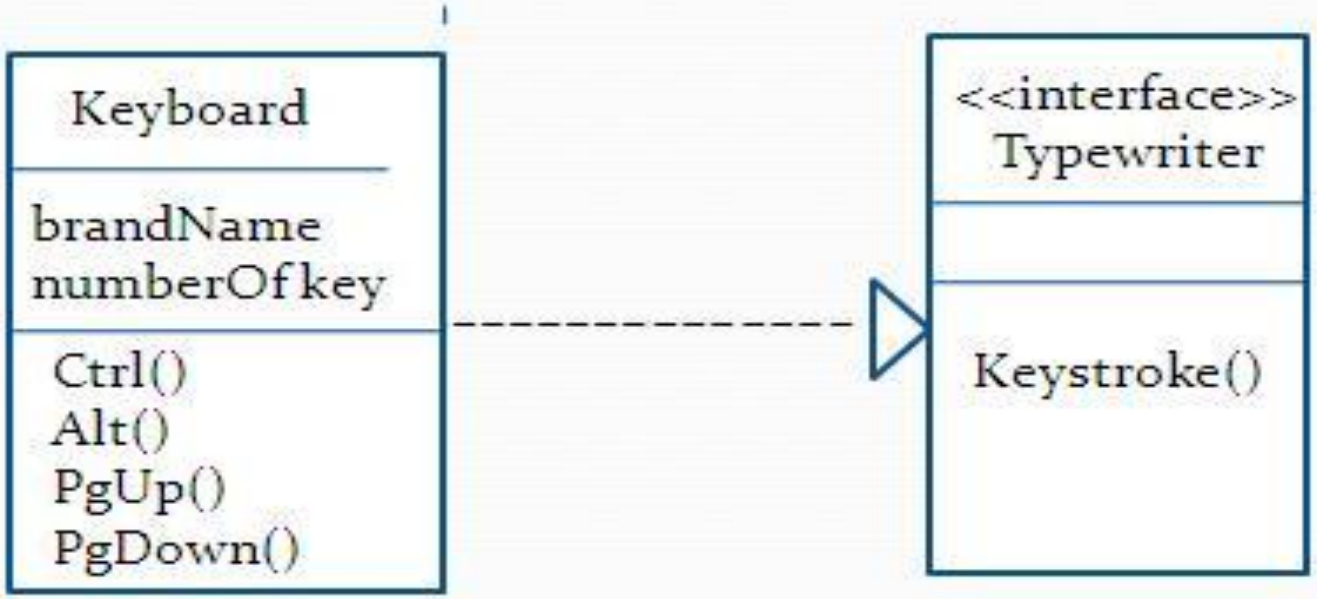
- **Dependency:** One class uses another ,i.e. Change in one class that affect change in another class

- E.g.



- **Realization:** The relationship between a class and interface is called realization
- This is modeled as a dashed line with a large open triangle adjoining and pointing to the interface

E.g.



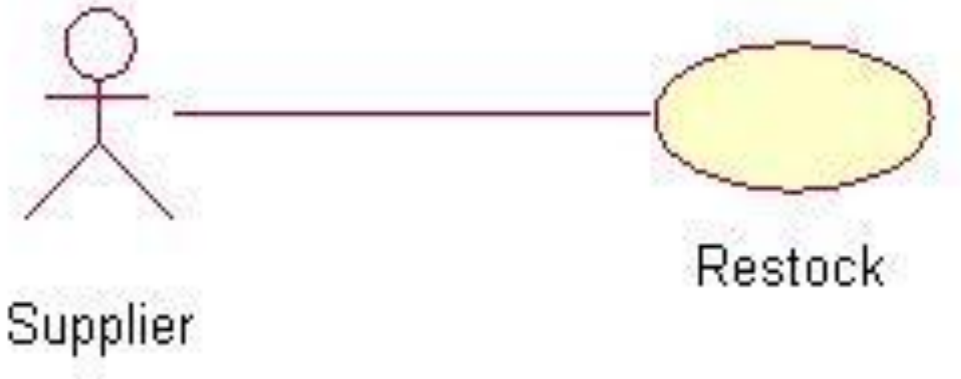
# Use Case diagram

- The use case is a powerful concept for **helping an analyst to understand how a system should behave**. It helps you gather requirements from the users' point of view
- it shows the **boundary** between the system and the outside world

**Representing a Use Case Model:**

- The actors, use cases, and interconnecting lines make up a **use case model**
- **Actor: An** actor initiates a use case, and an actor (possibly the initiator, but not necessarily) receives something of value from the use case.
- A **stick figure** represents an actor. The actor's name appears just below the actor
- **Use Case: U**se case is a *collection of scenarios*, and each scenario is a sequence of steps. **An ellipse** represents a use case. The name of the use case appears either inside the ellipse or just below it.
- An **association** line connects an actor to the use case, and represents communication between the actor and the use case. The association **line is solid**, like the line that connects associated classes.

# Use Case Diagram:




Use case dig:Represent Actor, association & Use case

- Actors are typically outside the system, whereas use cases are inside.

Visualizing Relationships Among Use Cases:

- **Inclusion: E**nables you to reuse one use case's steps inside another use case. It is basic steps to execute the use case operation
- To represent inclusion, you use the symbol you used for dependency between classes—a dashed line connecting the base use case with an arrowhead pointing to the depended-on use case. Near the line, you add the keyword **«include»**
- **Extension:** allows you to create a new use case by adding steps to an existing use case
- Like inclusion, you visualize extension with a dependency line (dashed line and arrowhead) along with a keyword. In this case the keyword is « **extend**» the base use case

Fig: Use Case dig for Soda machine

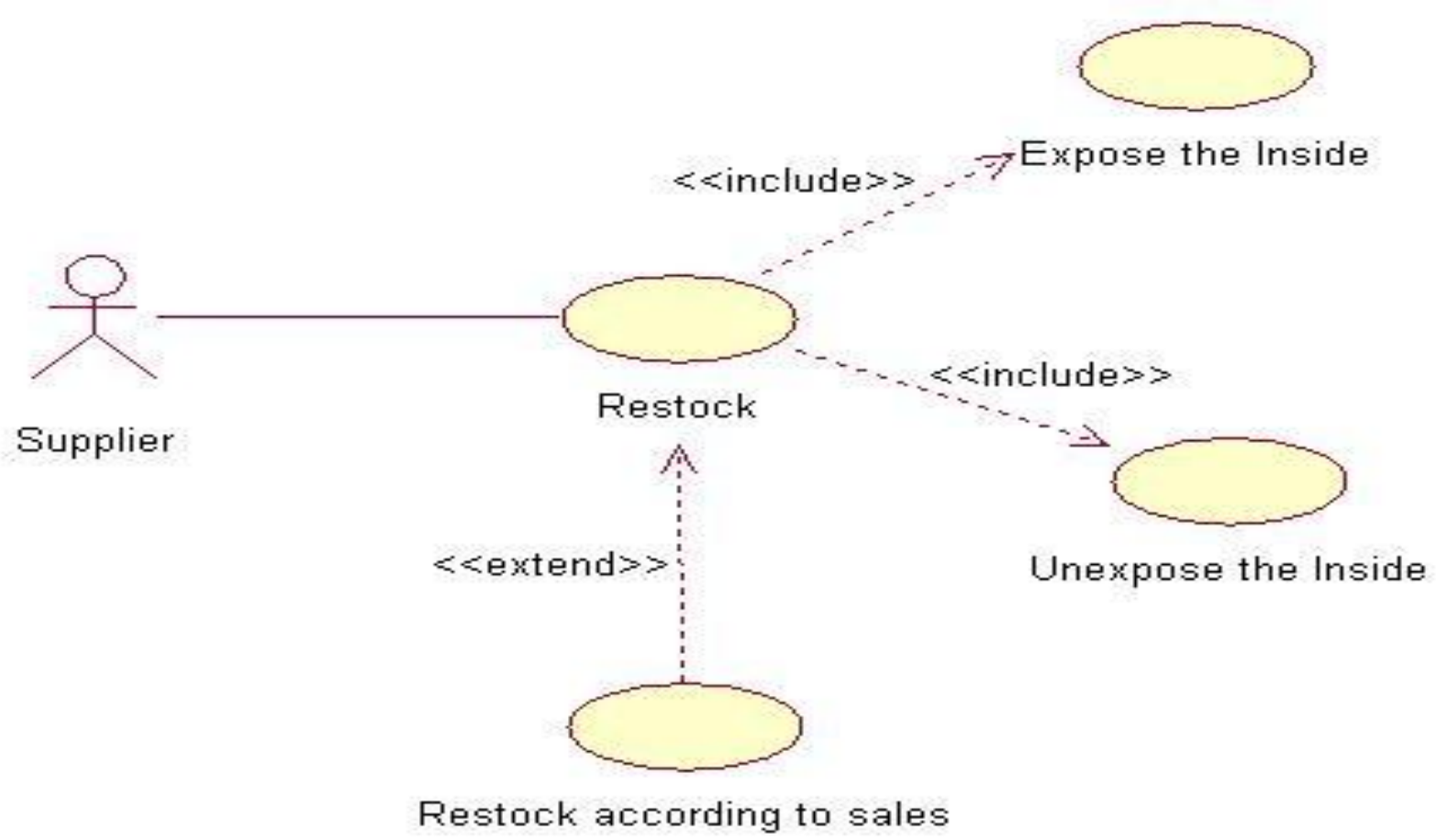



Fig:Include And Extend stereotype

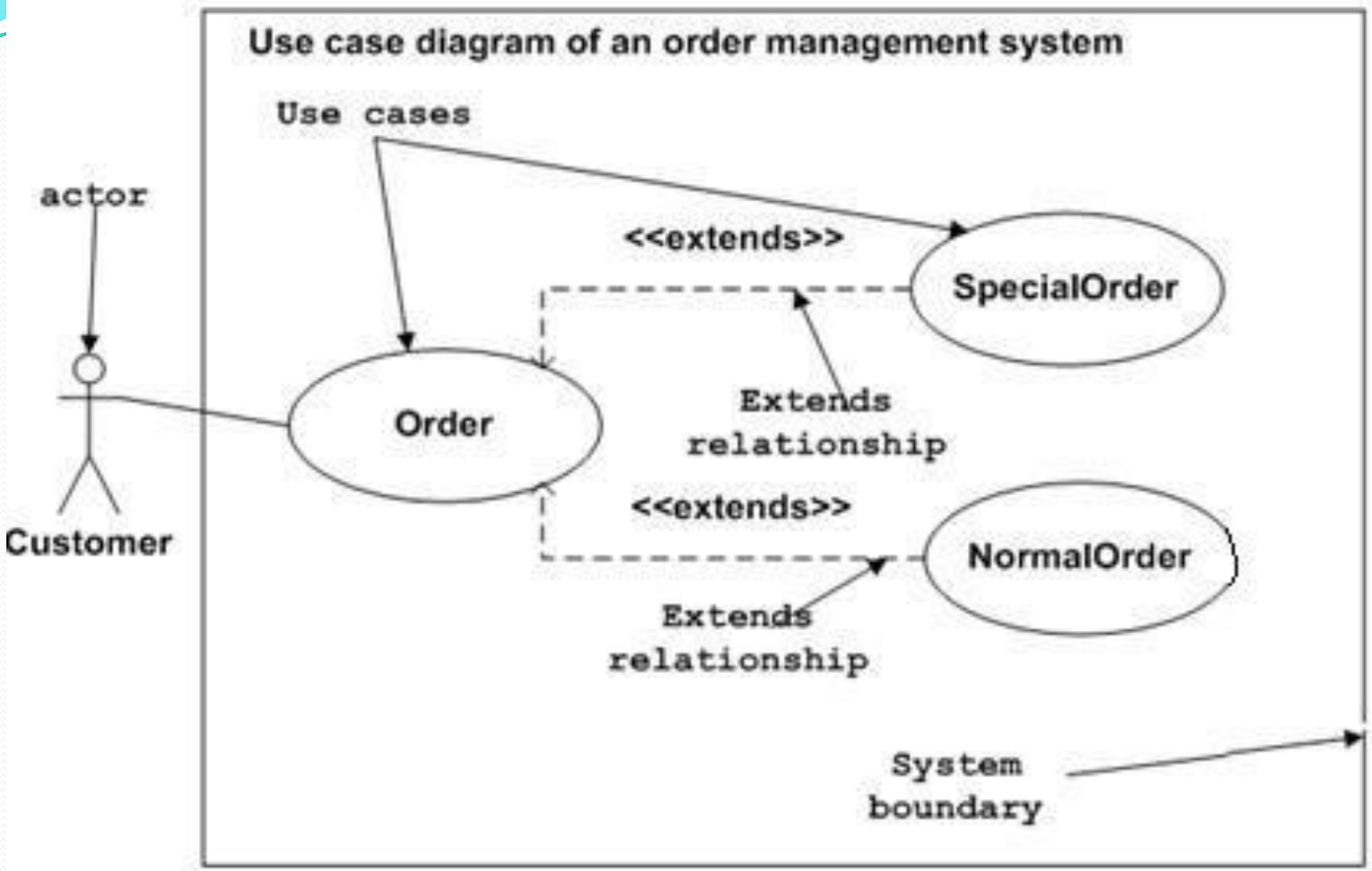



Figure: Sample Use Case diagram

# Activity Diagram

- The UML **activity diagram** is much like the **flowcharts** of old. It shows steps (**activities**) as well as decision points and branches. It's useful for showing what happens in a business process or an operation.
- It's an integral part of system analysis

Element of Activity Dig:

- **Starting point**: the activity diagram has a starting point represented by a **filled-in circle**
- **Activity:** activity is represented by a **rounded rectangle** narrower and more oval shaped than the state icon
- The processing within an activity goes to completion and then an automatic transmission to the next activity occurs
- **Transition:** An **arrow** represents the transition from one activity to the next
- **Endpoint**: endpoint represented by a **bull's-eye**

- Activity Diagram Representation

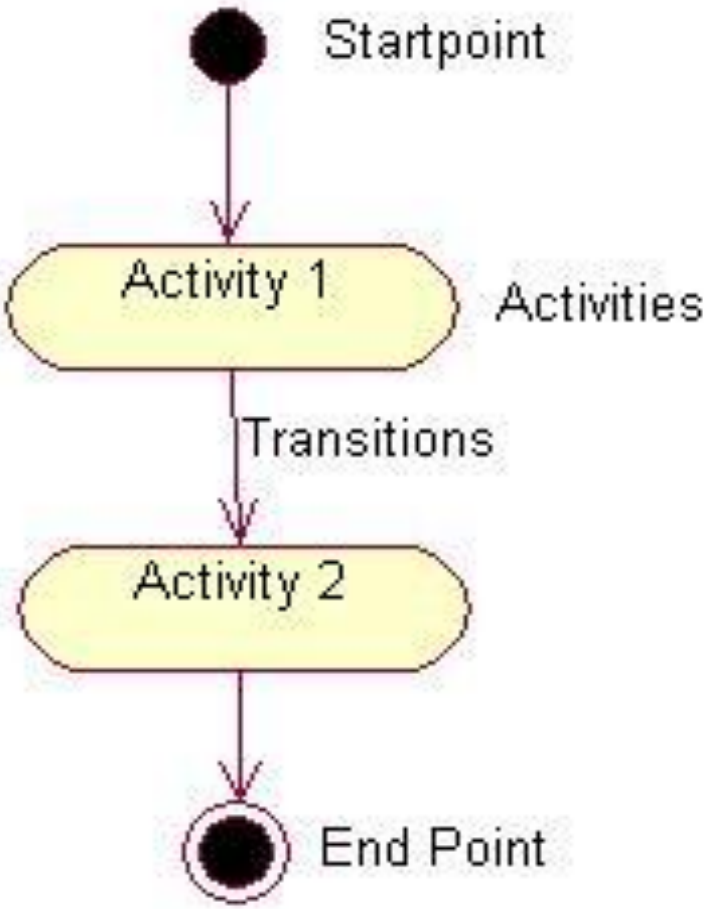



Figure:Transitions from one activity to another

- **<u>Concurrent Paths:</u> As** you model activities, you'll occasionally have to separate a transition into two separate paths that run at the same time (that is, concurrently) and then come together.
- To represent the S**plit(Fork)**, you use a *solid bold line* perpendicular to the transition and show the paths coming out of the line.
- To represent the M**erge(Join)**, show the paths pointing at another *solid bold line*
- **<u>Swimlanes:</u>** One of the handier aspects of the activity diagram is its ability to *expand* and show *who has the responsibility* for each activity in a process.
- you separate the diagram into parallel segments called **Swimlanes** Each Swimlanes shows the name of a role at the top and presents the activities of each role. Transitions can take place from one Swimlanes to another

- Fig: Concurrent Paths

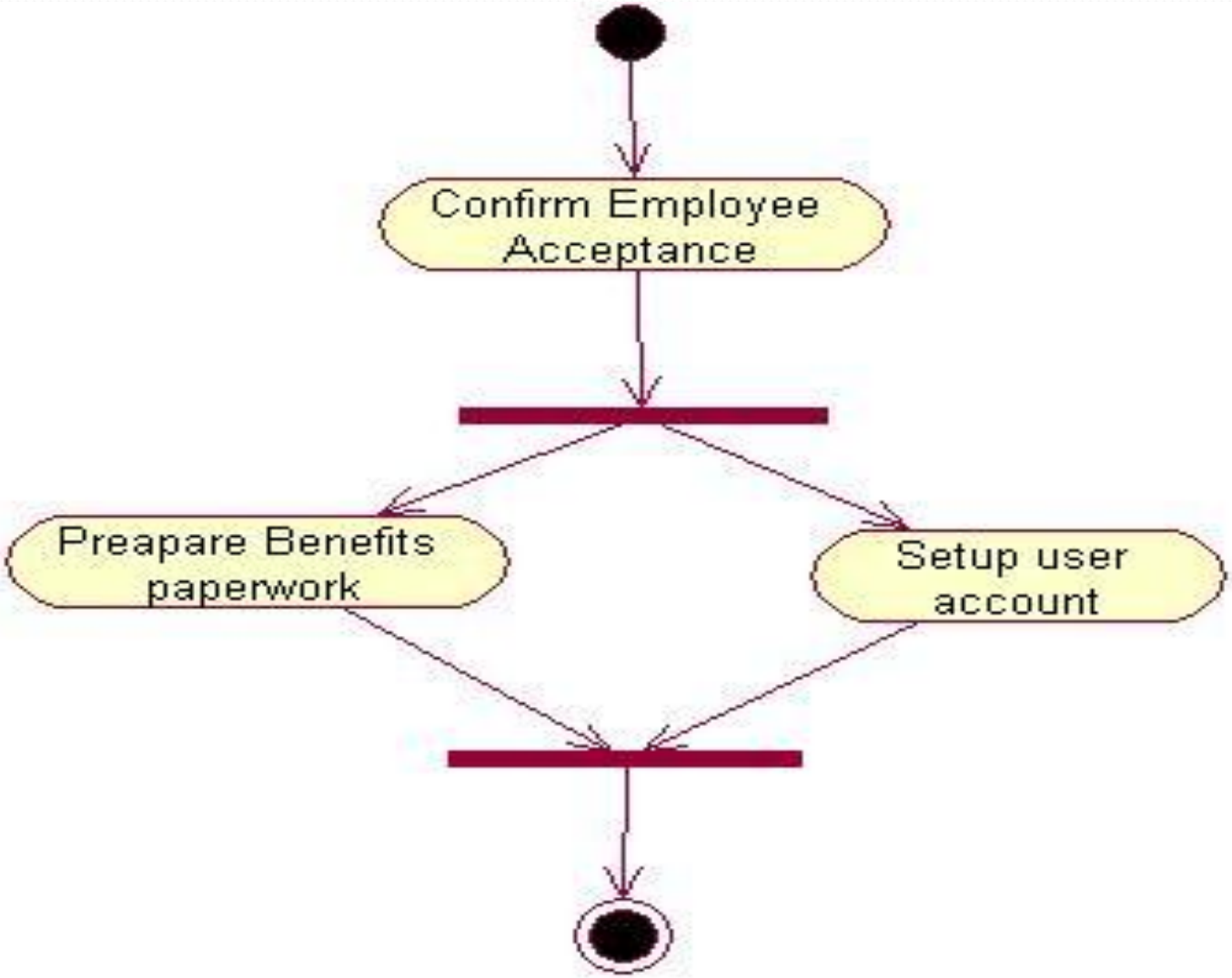
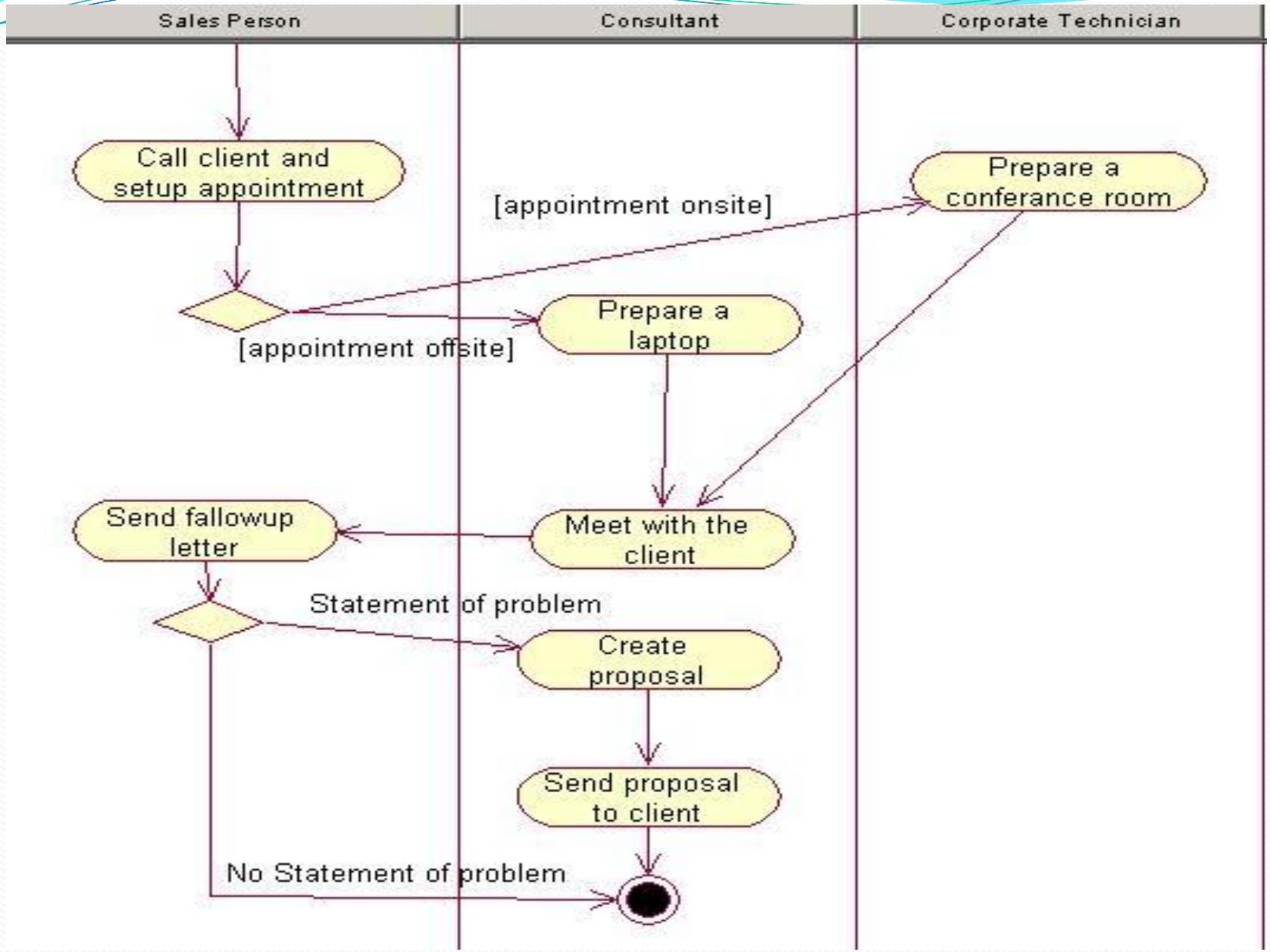
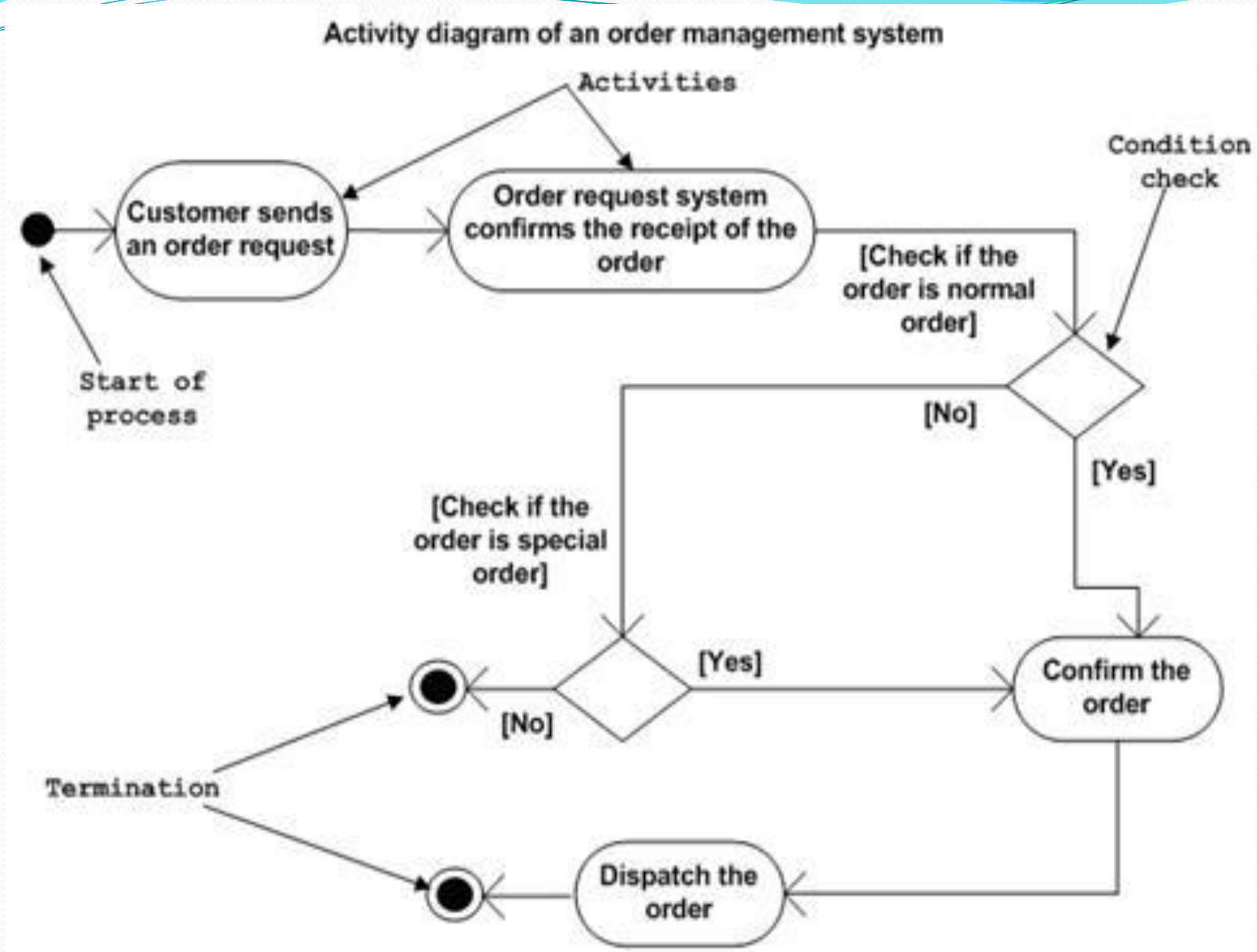



Figure:Hire Empolyee with fork and join node

- An Activity dig for Business process of meeting a new client with Swimlanes




Activity diagram of an order management system
Activities
Condition check
Customer sends an order request
Order request system confirms the receipt of the order
[Check if the order is normal order]
Start of process
[No]
[Yes]
[Check if the order is special order]
[Yes]
[No]
Confirm the order
Termination
Dispatch the order

# Class Diagram

- A **Class** is a **category** or **group of things** that have the **same attributes** and the **same behaviors.**
- A ***rectangle*** is the icon that represents the Class. It's divided into three areas. The uppermost area contains the ***name***, the middle area holds the ***attributes,*** and the lowest area holds the ***operations***
- UML, a multiword class name has initial ***capital letters*** for all the words and eliminates white space between each word (for example, ***WashingMachine***). Attribute names and Operation names follow the same convention, but the first letter of the first word isn't capitalized (for example***, acceptClothes()***). A pair of parentheses follows the name of each operation
- A **class diagram** consists of a number of these **rectangle icons** connected by **lines** that show how the classes relate to one another at **developer** side
- Class diagrams also help on the **analysis side**. They enable analysts to talk to clients in the clients' terminology to reveal important details about the problem they want solved

- Class Representation

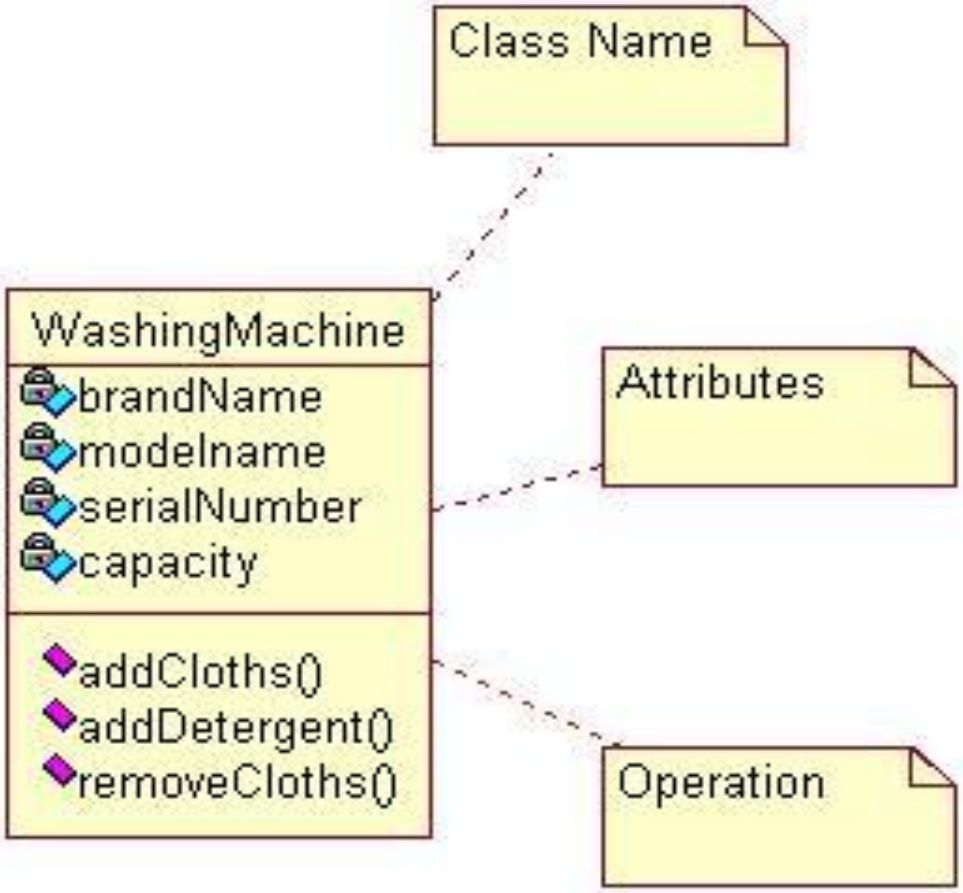
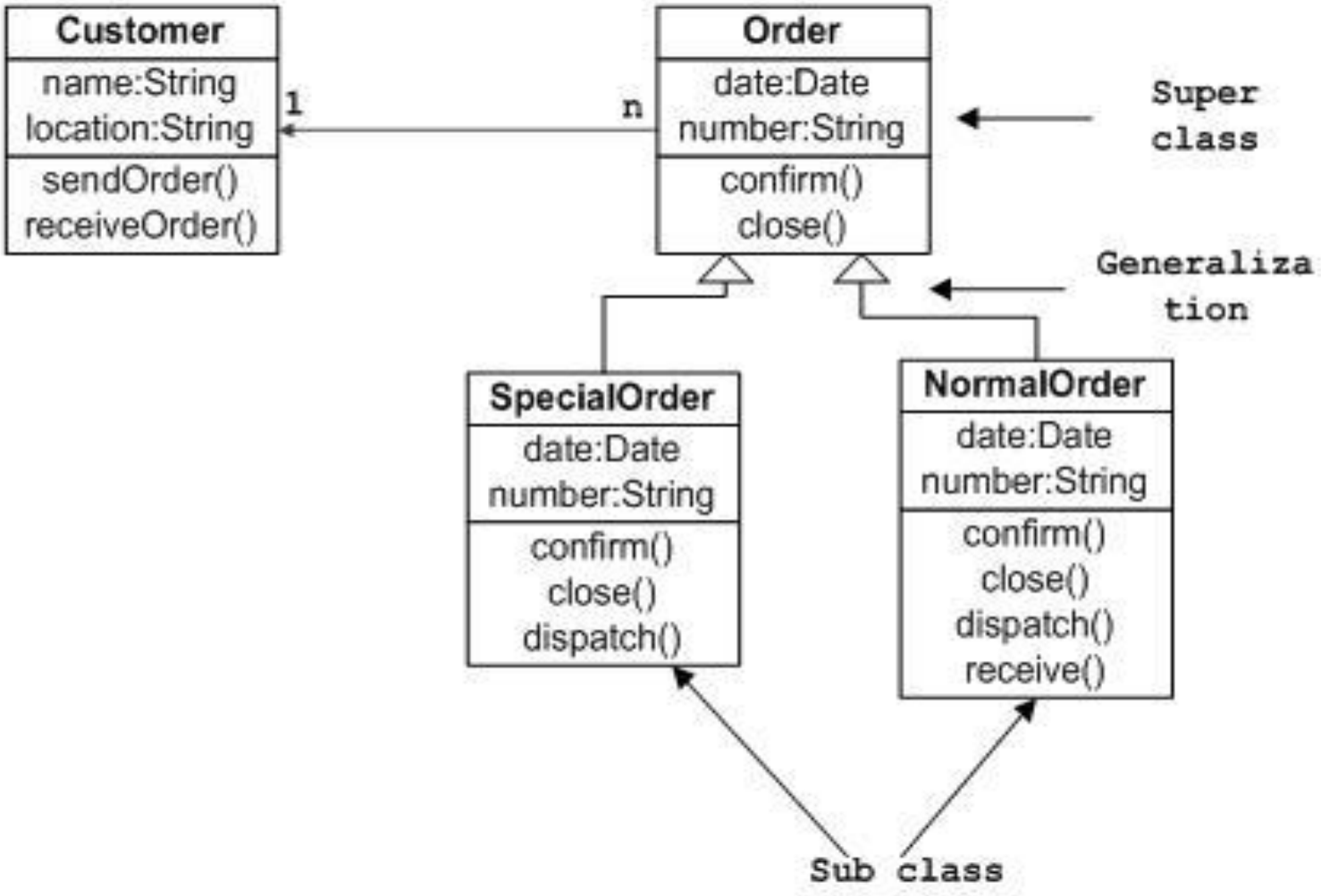



Fig Class Representation

## Sample Class Diagram



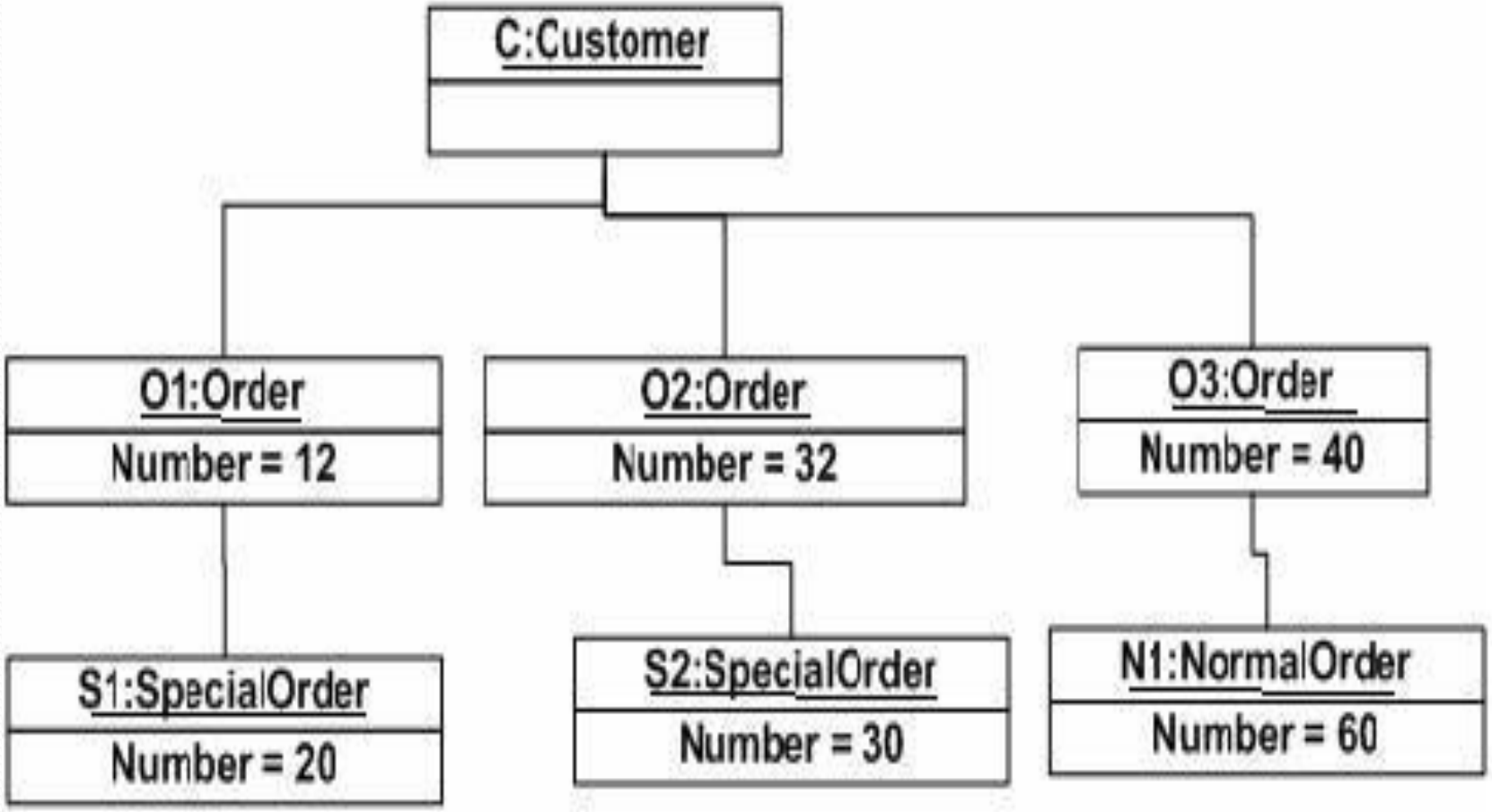
# Object Diagram

- **O**bject diagrams are **derived** from **class diagrams** so object diagrams are **dependent** upon class diagrams
- **O**bject diagrams represent an **instance** of a class diagram. The basic concepts are similar for class 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.*
- It means the **o**bject diagram is **more close** to the **actual system** behavior. The purpose is to capture the static view of a system at a particular moment

  *e.g. Taking snapshot of running Train*

  *Taking snapshot of running Movies*
- The icon is a **rectangle**, just like the class icon, but the **name is underlined**. In the icon on the *left*, the name of the specific *instance* is on the *left side* of a colon, and the name of the *class* is on the *right side* of the colon. The name of the instance begins with a *lowercase* letter

Object diagram of an order management system



# Package Diagram

- Sometimes ,you'll find the need **to organize the element of a diagram into a group** .you might want to show that a number of classes or components are part of particular subsystem. to do this ,you group them into a package
- **Representation :** you show a package using a rectangle with a tab attached to the top left
- **Importing :** Importing grants a one way permission for the elements in one package to access the element in another package
- In Uml you model an import relationship as dependency arrow with stereotype import
- **Exporting:** the public part of package are called its exports and represented by + sign. A package can only exports its public element no permission for private(-) and protected (#) element
- **Transitive property:** *Package A* imports *package B*, and *package B* imports *package C* but *package A* does not access *package C*

- Representation of package

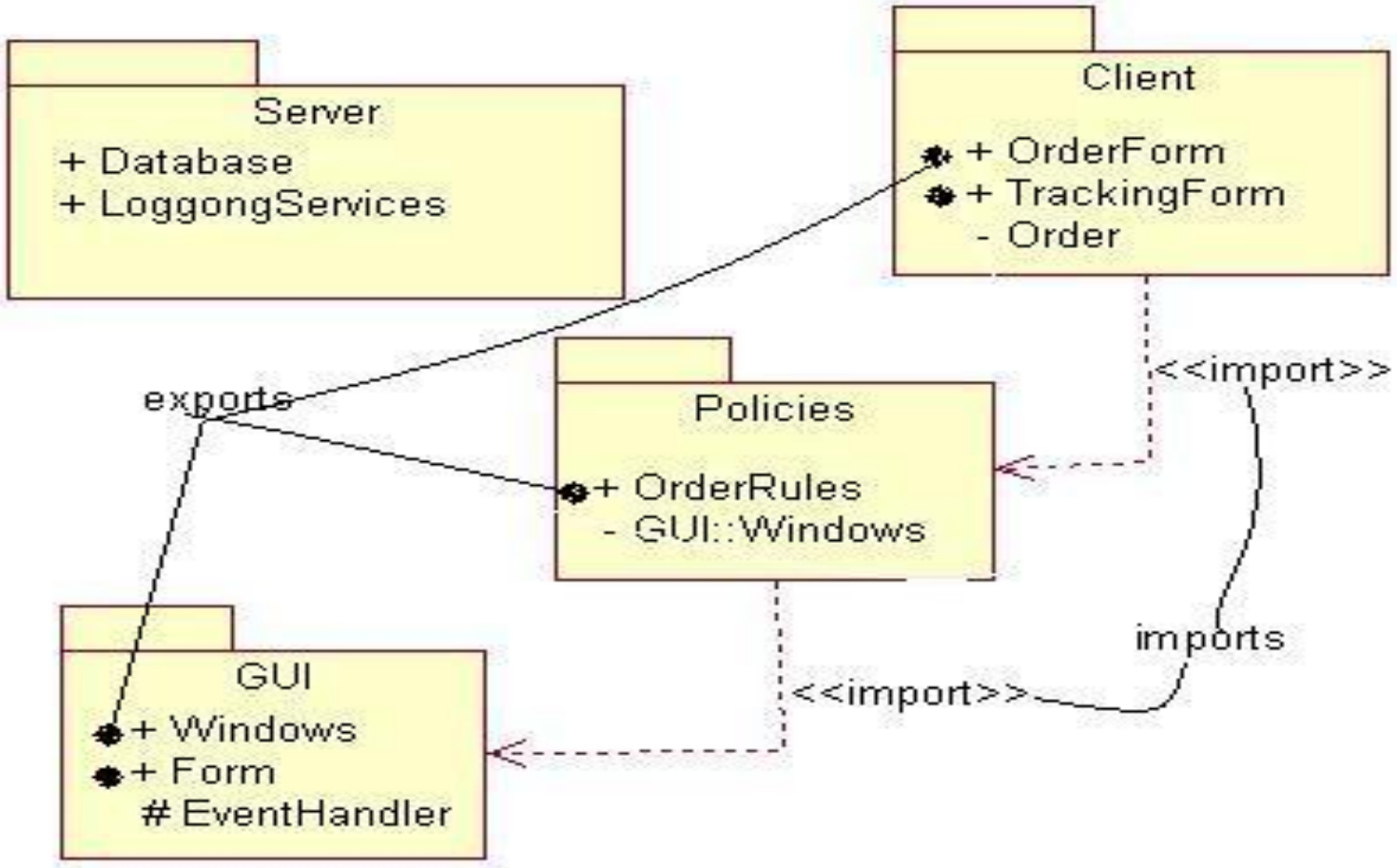



Figure: Importing and exporting

# Sequence Diagram

- **C**lass diagrams and **o**bject diagrams represent *static* information.
- In a functioning system, however o***bjects interact*** with one another, and these interactions occur **over time**. The UML sequence diagram shows the **time-based** dynamics of the interaction.
- The **sequence diagram** consists of **objects** represented in the usual way (as ***named rectangles*** with the ***name underlined***), **messages** represented as **solid-line arrows**, and **time** represented as a **vertical progression**.
- **Objects:** The objects are laid out near the top of the diagram from left to right. They're arranged in any order that simplifies the diagram. Extending downward from each object is a dashed line called the object's **lifeline**
- Along the lifeline is a narrow rectangle called an **activation .**The activation represents an execution of an operation the object carries out. The length of the rectangle signifies the activation's duration

- Sequence Diagram

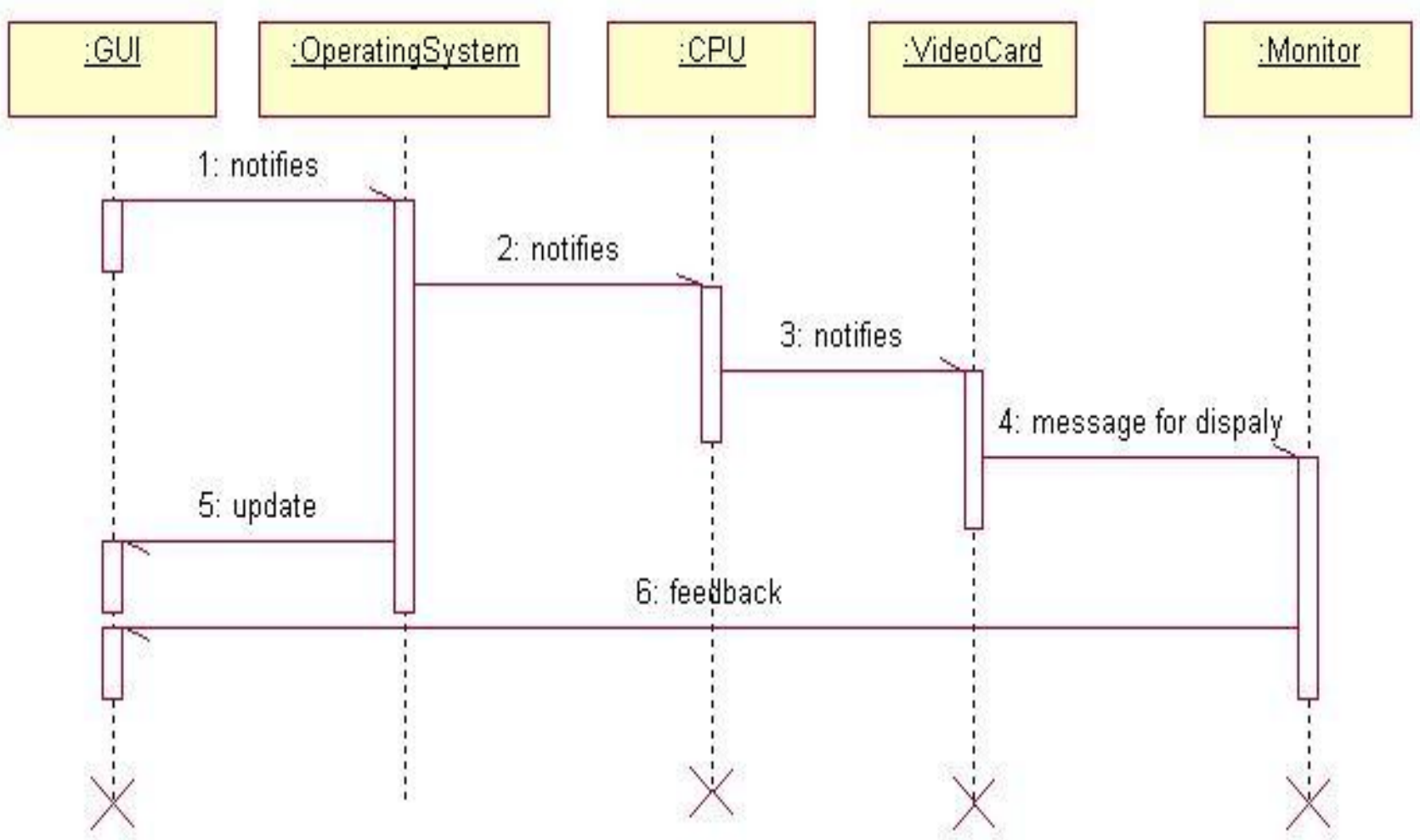
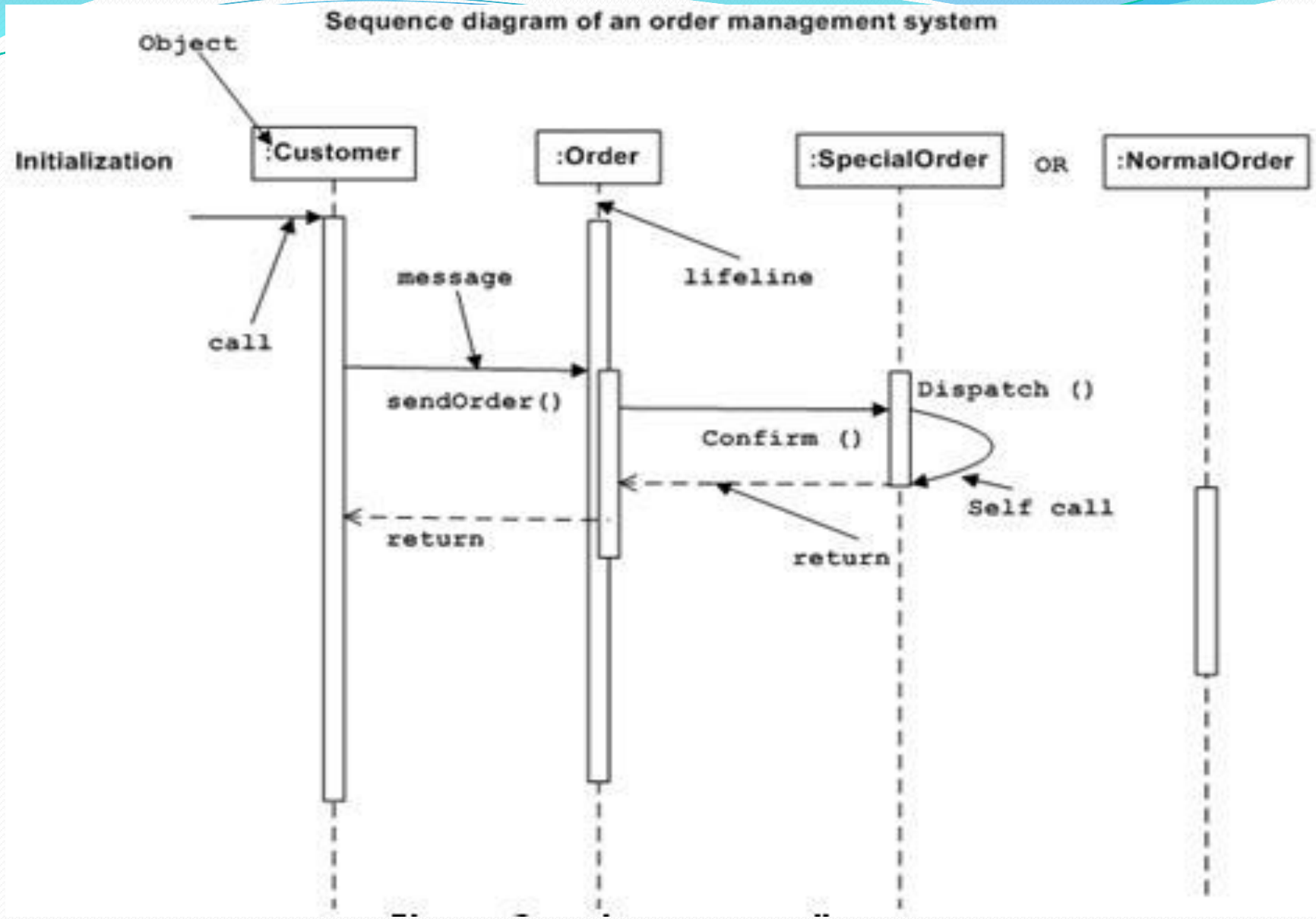



Fig:A GUI interacts with other object

- **Messages:** A message that goes from one object to another, goes from one object's lifeline to the other object's lifeline.
- An object can also send a message to itself that is from its lifeline back to its own lifeline
- UML represents a message as an arrow that starts at one lifeline and ends at another. The shape of the arrowhead shows what type of message it is.
- UML provides Three types of messages: ***Simple message,*** ***Synchronous message*** and **Asynchronous message**
- **Time :** The diagram represents time in the vertical direction. Time starts at the top and progresses toward the bottom. A message that's closer to the top occurs earlier in time than a message that's closer to the bottom
- Thus, the sequence diagram is **two-dimensional.** The left-to-right dimension is the layout of the objects, and the top-to-bottom dimension shows the passage of time

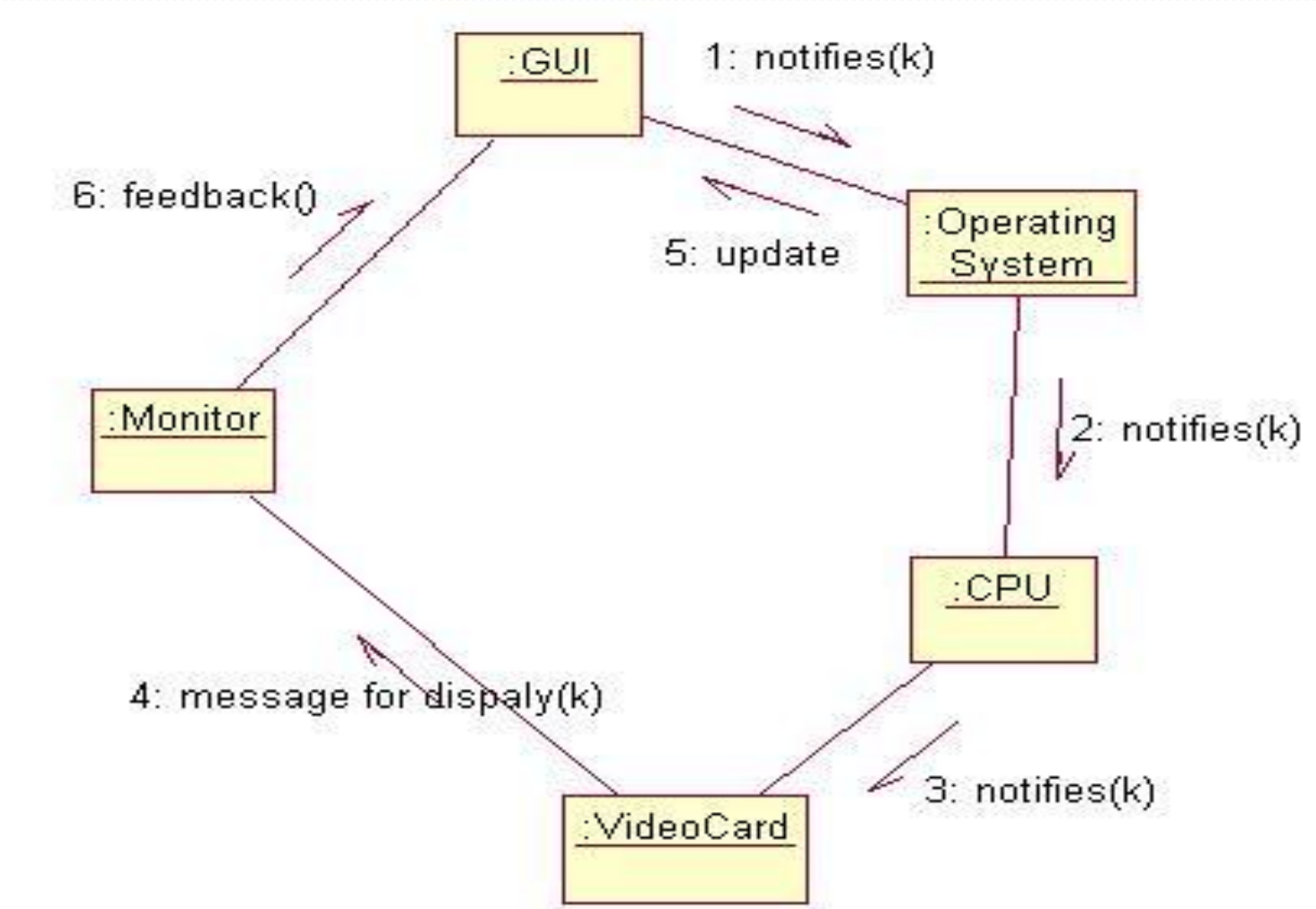

Sequence diagram of an order management system
Object
Initialization
:Customer
:Order
:SpecialOrder
OR
:NormalOrder
call
message
lifeline
sendOrder()
Dispatch ()
Confirm ()
Self call
return
return

# Collaboration Diagram

- Like the sequence diagram, the **communication** diagram shows how objects interact. It shows the objects along with the messages that travel from one object to another.
- Difference between Sequence dig & Communication dig: The sequence diagram is arranged according to **time**, the communication diagram according to **space**
- An object diagram shows objects and their relationships with one another. A communication diagram is an **extension of the object diagram**. In addition to the links among objects, the communication diagram shows the messages the objects send each other.
- The object diagram is the **snapshot:** It shows how instances of classes are linked together in an instant of time

  The communication diagram is the **movie**: It shows interactions among those instances over time

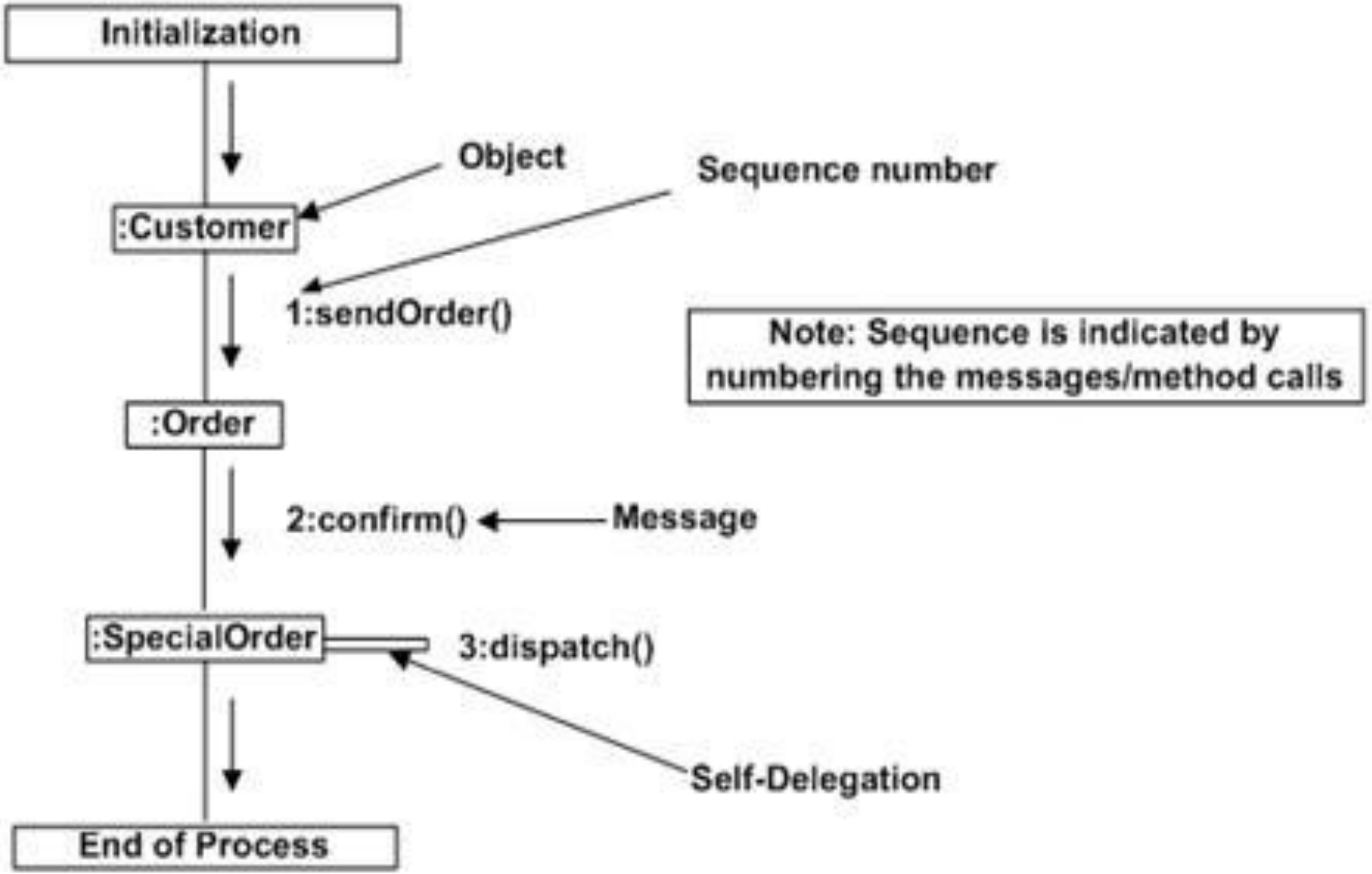
- Collaboration Diagram




A collaboration dig for the GUI

- **Message :**To represent a message, you draw an arrow near the link between two objects.
- The arrow points to the receiving object. A label near the arrow shows what the message is.
- The message typically tells the receiving object to execute one of its (the receiver's) operations.
- Arrowheads have the same meaning as in sequence diagrams.

Collaboration diagram of an order management system



# State diagram

- One way to characterize **change in a system** is to say that its **objects change their state** in response to **events** and to **time**
- E.g. When you throw a switch, a light changes its state from Off to On.

**The Fundamental Symbol Set:**

- The **rounded rectangle** that represents a state, along with the solid line and arrowhead that represent a transition.
- The arrow head points to the state being transitioned into. The figure also shows the solid circle that symbolizes a starting point and the bull's-eye that symbolizes an endpoint

**Adding Details to the State Icon:**

- The UML gives you the option of adding detail to these symbols.
- You can divide the state icon into **two areas**. The **top** area holds the **name of the state** and the **bottom** area holds **activities** that take place in that state

- State Diagram

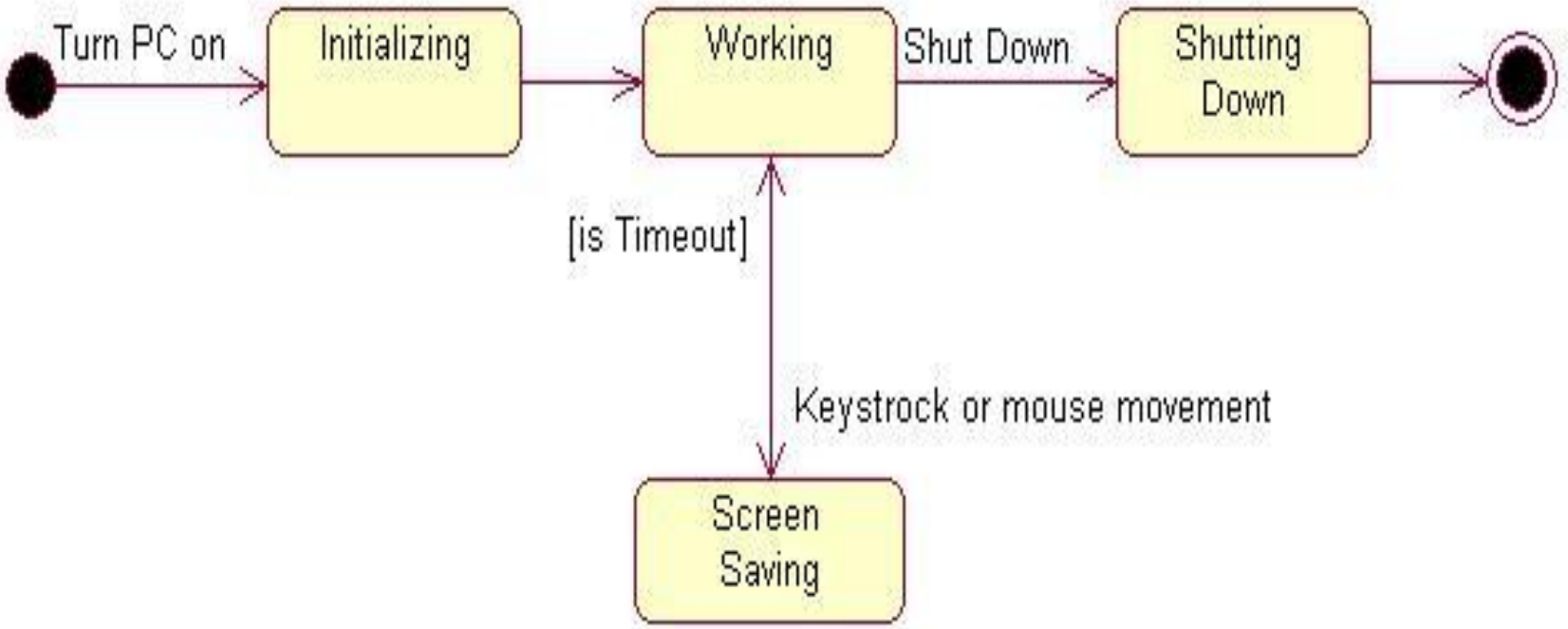



Fig:The state dig for GUI,with Sreensaving state and a guard condition.

**Adding Details to the Transitions**: Events and Actions

- You can also add some details to the transition lines. You can indicate an event that causes a transition to occur (a **trigger event** ) and the computation (the **action** ) that executes and makes the state change happen.
- To add **events** and **actions** you write them near the transition line, using a slash to separate a triggering event from an action.
- Sometimes an event causes a transition without an associated action, and sometimes a transition occurs because a state completes an activity This type of transition is called a **trigger less transition**

**Guard condition:** the condition responsible for state change and represented by **square bracket** having condition's name

e.g.[is timeout]

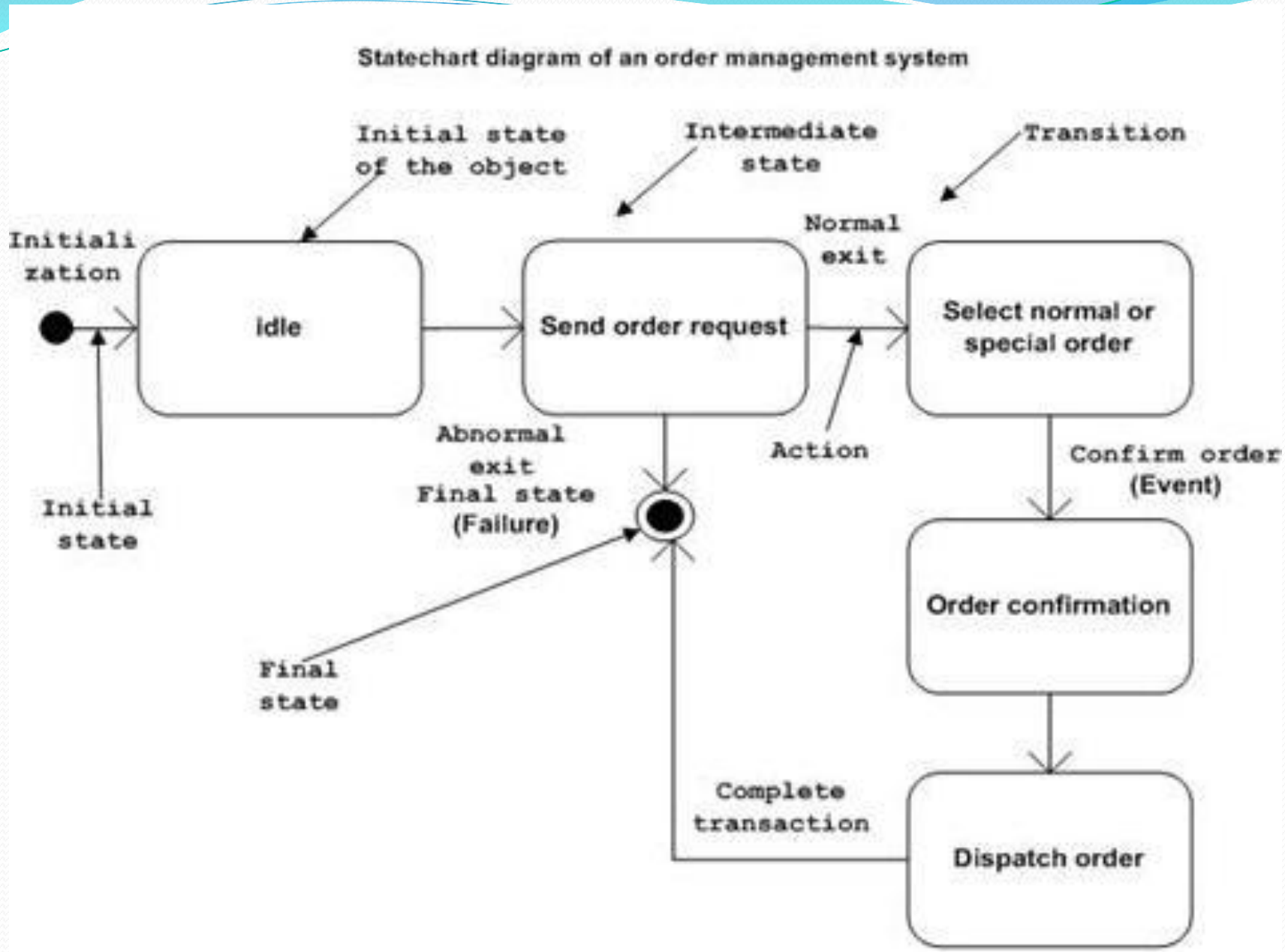
**Sub State:** the states reside within a state, they're called **sub state** substates come in two varieties: **sequential** and **concurrent**

Sequential substates **:** As the name implies, sequential substates occur one after the other.

Concurrent Substates: the two sequence running simultaneously

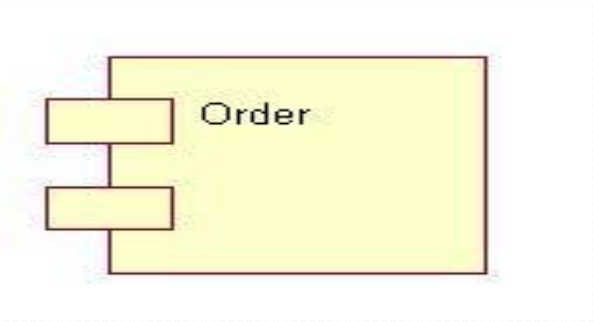
**History States:** The UML supplies a symbol that shows that a composite state **remembers its active sub states** when the object **transitions out** of the composite state.

- The symbol is the letter ***H*** enclosed in a **small circle** connected by a **solid line** to the remembered sub states, with an **arrowhead** pointing to that **sub states**


Statechart diagram of an order management system
Initial state of the object
Intermediate state
Transition
Initiali zation
Normal exit
idle
Send order request
Select normal or special order
Abnormal exit Final state (Failure)
Action
Confirm order (Event)
Initial state
Order confirmation
Final state
Complete transaction
Dispatch order

# Component Diagram

- A software component is a **modular** part of a system. Because it's the **software implementation** of one or more **classes,**
- A component resides in a computer, not in the mind of an analyst. A component provides interfaces to other components
- data files, tables, executables, documents, and dynamic link libraries were defined as components
- A component is **physical** and **replaceable** and **reusable** software part of system
- **Representing a Component in UML**
- the component diagram's main icon is a **rectangle** that has two rectangles overlaid on its left side



**Types of Component**

***Deployment component***, which form the basis of executable system.eg DLLs, Executable, ActiveX controls and Java Beans.

***Work product Component***, from which deployment component are created e.g. Data files and source code files

***Executable component***, created as result of running system

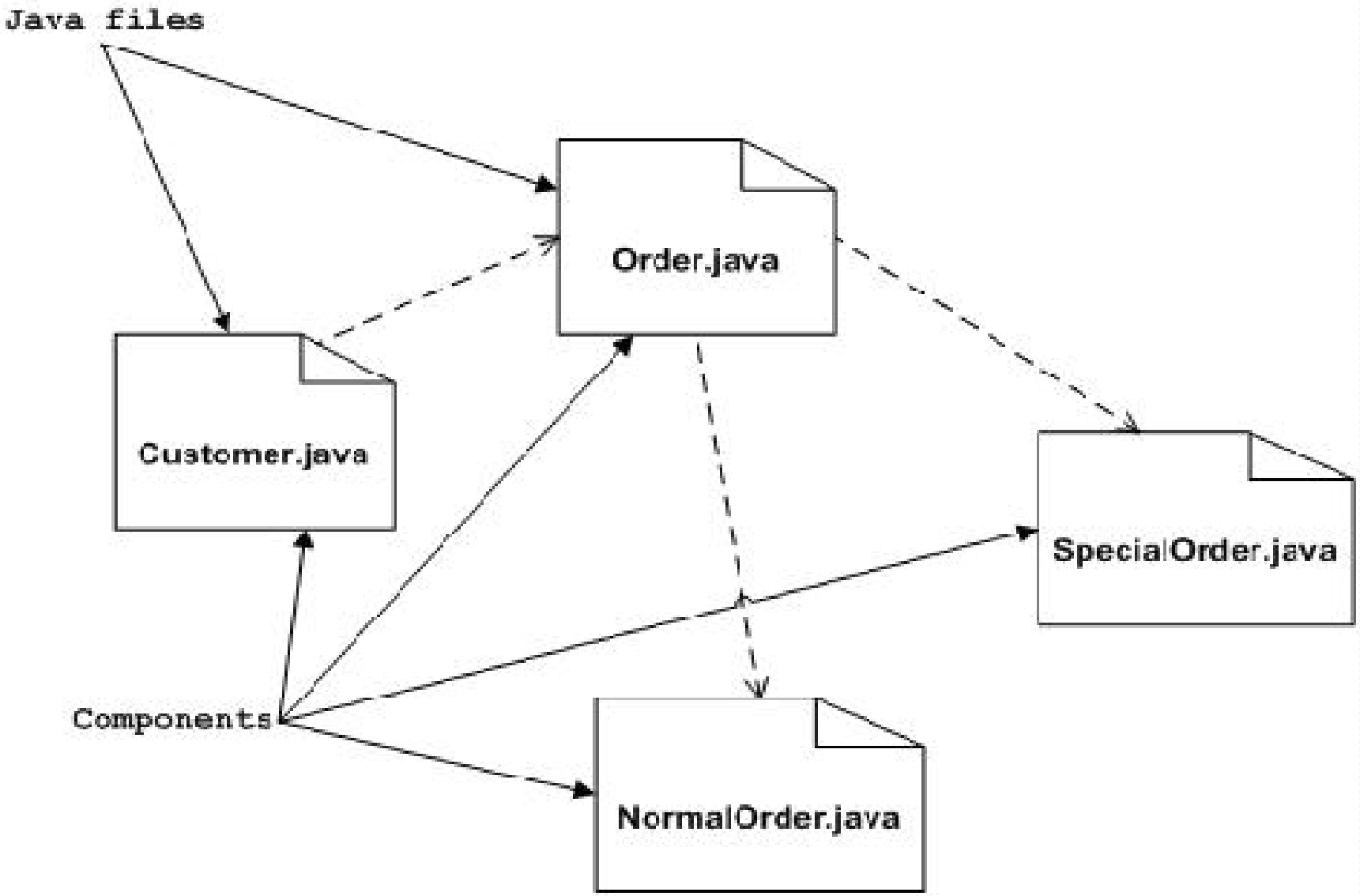
**Replacement and Reuse :** Interfaces figure heavily into the important concepts of component replacement and component reuse. You can replace one component with another if the new component conforms to the same interfaces as the old one

**Representing Interfaces:**

A component and the interfaces it realizes can be represented in two ways.

- The first shows *the interface as a rectangle* that contains interface-related information. It's connected to the component by the *dashed line and large open triangle* that indicate ***realization***.
- The second way. It's iconic: You represent the interface as *a small circle* connected to the component *by a solid line*
- A component *provides* interfaces that allow other components to access it.

Component diagram of an order management system



# Deployment Diagram

- A deployment diagram shows how **artifacts** (**component**) are **deployed** on **system hardware**, and how the pieces of hardware connect to one another
- The name *Deployment* itself describes the purpose of the diagram. Deployment diagrams are used for describing the hardware components where software components are deployed. Component diagrams and deployment diagrams are closely related.
- **A node** is main hardware item , a generic name for a computing resource.
- Two types of nodes

  A ***processor node*** *:* a node that can execute a component and

  A ***device node :*** a peripheral piece of hardware that doesn't execute components but typically interfaces in some way with the outside world
- A **line** joining two cubes represents a **connection** between two nodes. you can use a *stereotype* to provide information about the connection.

  Basically connection between the node is **Association** type but you can also use *aggregation and dependency*

- Deployment Diagram

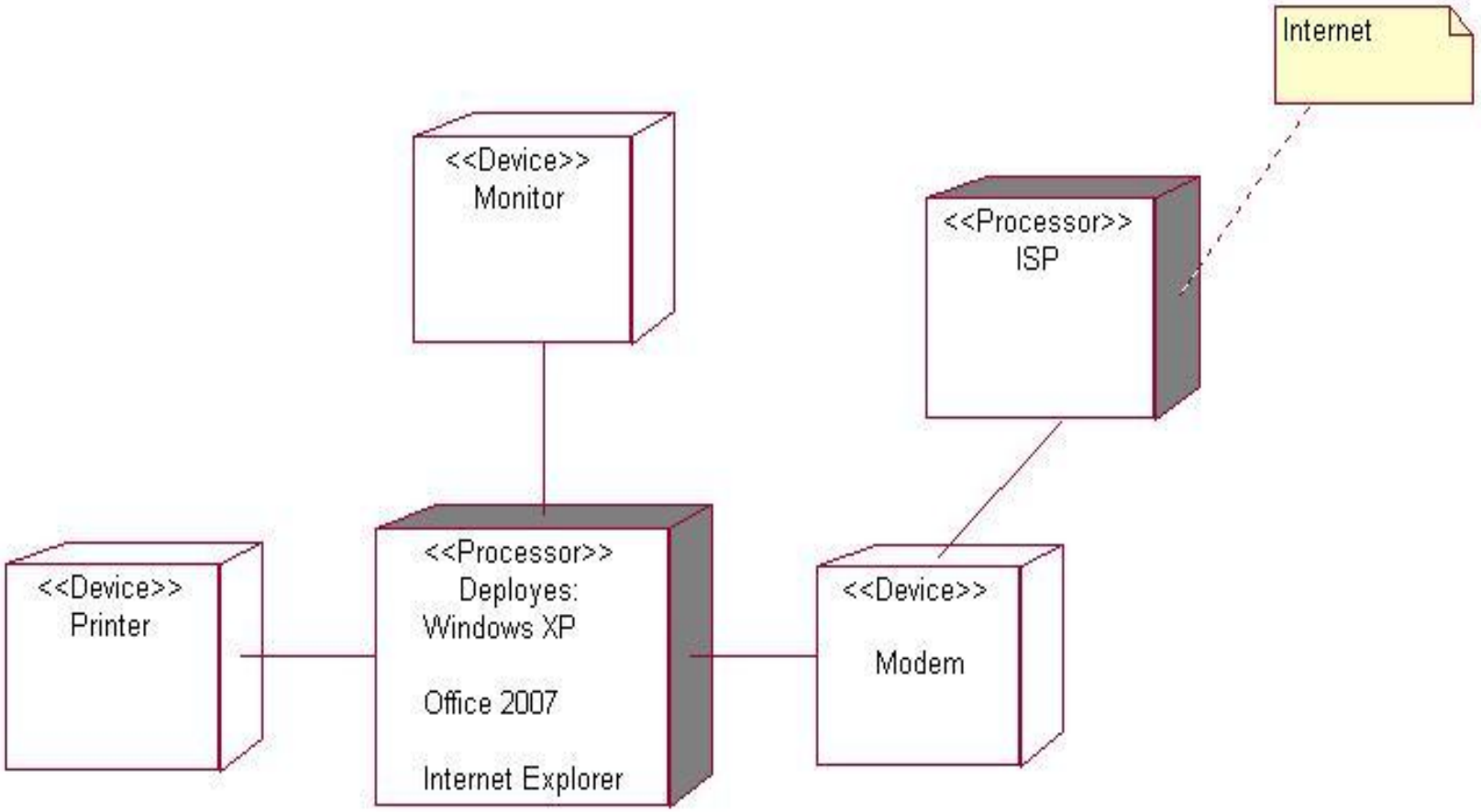



Fig:Home System

Deployment diagram of an order management system