

**IARE**  
Institute of Aeronautical Engineering

An **Autonomous** Institute  
**NAAC** Accreditation with 'A' Grade  
Accredited by **NBA**  
Permanent Affiliation Status from **JNTUH**

# OBJECT ORIENTED ANALYSIS AND DESIGN

## Module- 1

# Introduction to UML

**Dr. C RAGHAVENDRA**

**Associate Professor**

**IT Dept. IARE**

- Importance of Modeling
- Principles of Modeling
- Object Oriented Modeling
- Conceptual model of UML
- Architecture
- Software Development life Cycle
- Classes
- Relationships
- Common mechanism and Diagrams

# Text Books:



1. Grady Booch, James Rumbaugh, Ivar Jacobson, "The Unified Modeling Language User Guide", Pearson Education, 2nd Edition, 2004.
2. Craig Larman, "Applying UML and Patterns", 3rd Edition, Pearson Education, 2011.

- **Importance of Modeling**
- **Principles of Modeling**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Importance of Modeling</b></li><li>• <b>Principles of Modeling</b></li></ul>	<ul style="list-style-type: none"><li>• Representing the importance of modeling concept for object oriented development in system.</li></ul>	<ol style="list-style-type: none"><li>1. Describe the importance of Object Oriented concepts, UML diagrams and their relationships.</li></ol>

# Outcome achieved



## Name of the topic:

<b>Students will be able to do:</b>	
1	Summarize the importance of modeling concept.
2	Describe the Object Oriented concepts, UML diagrams and their relationships.

# What is UML?



- “The Unified Modeling Language is a family of graphical notations, based by a single meta-model, that help in *describing* and *designing* software systems, particularly software systems built using the object-oriented style.”
- UML first appeared in 1997
- UML is standardized one and its content is controlled by the Object Management Group (OMG), a consortium of companies.

- UML combined the best from **object-oriented software modeling methodologies** that were in existence during the early 1990's.

## **Modeling:**

- Used to present a simplified view of reality in order to facilitate the design and implementation of object-oriented software systems.
- All creative disciplines use some form of modeling as part of the creative process.



# Language

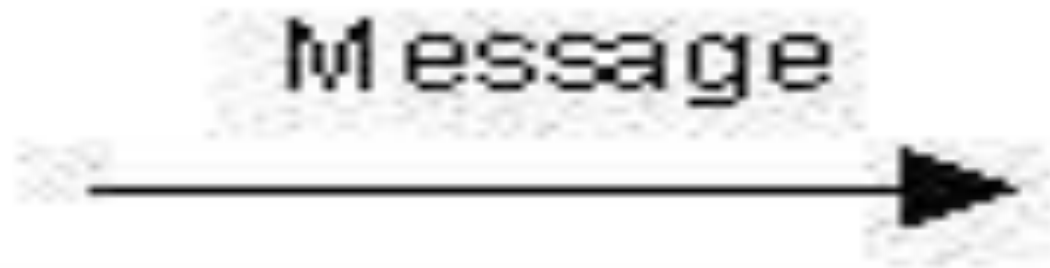
- UML is a **language** for documenting design
- Provides a record of what has been built.
- Useful for bringing new programmers up to speed.
- UML is primarily a graphical **language** that follows a precise syntax.
- UML 2.5 is the most recent version.

# Things

Things are the most important building blocks of UML. Things can be –

1. Behavioral
2. Grouping
3. Structural
4. Annotational

- **A behavioral thing** consists of the **dynamic parts of UML** models. Following are the behavioral things –
  - ✓ **Interaction** – Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.



- **State machine** – It defines the sequence of states an object goes through in response to events.
- Events are external factors responsible for state change.



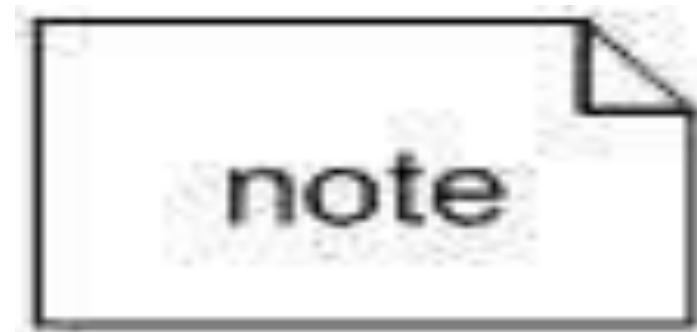
- **Grouping Things:**

- It can be defined as a **mechanism** to group elements of a UML model together. There is only **one** grouping thing available –
- **Package** – Package is the only one grouping thing available for gathering **structural and behavioral things**.



- **Annotational Things:**

- It can be defined as a mechanism to capture **remarks**, **descriptions**, and **comments** of UML model elements.
  - **Note** - It is the only one Annotational thing available. A note is used to render comments, constraints, etc. of an UML element.

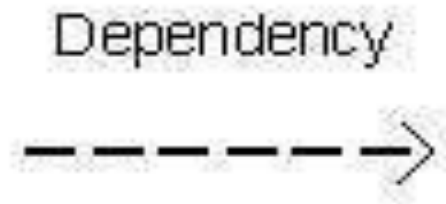


# Relationship

- It is another most important building block of UML.
- It shows how the elements are associated with each other and this association describes the functionality of an application.
- There are four kinds of relationships available.
  1. Dependency
  2. Association
  3. Generalization
  4. Relization

# 1. Dependency

- Dependency is a relationship between two things in which change in one element also affects the other.





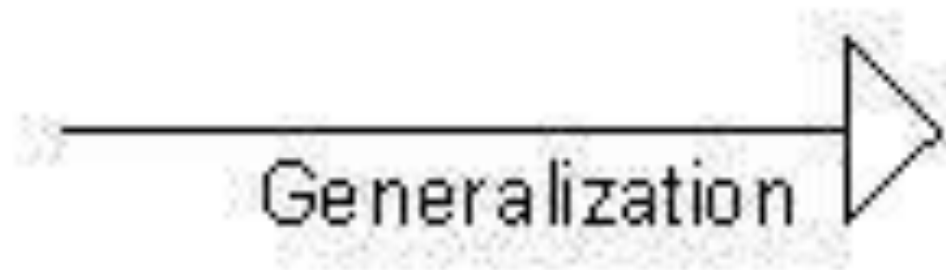
## 2. Association

- Association is basically a set of links that connects the elements of a UML model.
- It also describes how many objects are taking part in that relationship.



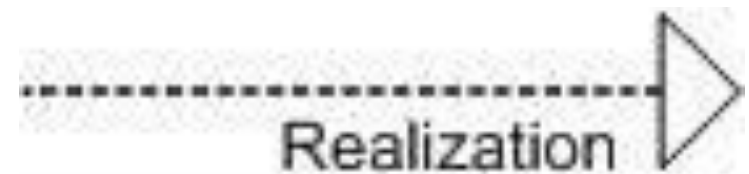
### 3. Generalization

- Generalization can be defined as *a relationship* which connects a **specialized element** with a **generalized element**.
- It basically describes the inheritance relationship in the real world of objects.



## 4. Realization

- Realization can be defined as a relationship in which two elements are connected.
- One element describes **some responsibility**, which is not implemented and the other one **implements** them.
- This relationship exists in case of interfaces.



- All the **elements, relationships** are used to make a complete UML diagram and the diagram represents a system.
- The **visual effect** of the UML diagram is the most important part of the entire process.
- All the other elements are used to make it complete.
- UML includes the following nine diagrams, the details of which are described in the subsequent chapters.

1. Class diagram
2. Object diagram
3. *Use case diagram*
4. *Sequence diagram*
5. Collaboration diagram
6. *Activity diagram*
7. Statechart diagram
8. Deployment diagram
9. Component diagram

- **Importance of Modeling**
- **Principles of Modeling**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Importance of Modeling</b></li><li>• <b>Principles of Modeling</b></li></ul>	<ul style="list-style-type: none"><li>• Representing the importance of modeling concept for object oriented development in system.</li></ul>	<ol style="list-style-type: none"><li>1. Describe the importance of Object Oriented concepts, UML diagrams and their relationships.</li></ol>

# Outcome achieved



## Name of the topic:

<b>Students will be able to do:</b>	
1	Summarize the importance of modeling concept.
2	Describe the Object Oriented concepts, UML diagrams and their relationships.



- UML can be described as the successor of object-oriented (OO) analysis and design.
- An object contains both data and methods that control the data.
- The data represents the state of the object.
- A class describes an object and they also form a hierarchy to model the real-world system.
- The hierarchy is represented as inheritance and the classes can also be associated in different ways as per the requirement.

- Objects are the real-world entities that exist around us and the basic concepts such as abstraction, encapsulation, inheritance, and polymorphism all can be represented using UML.
- UML is powerful enough to represent all the concepts that exist in object-oriented analysis and design.

Following are some fundamental concepts of the object-oriented world –

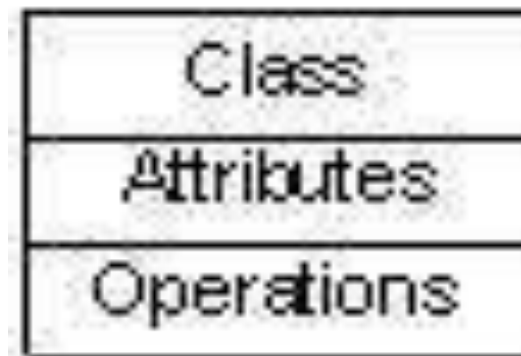
- **Objects** – Objects represent an entity and the basic building block.
- **Class** – Class is the blue print of an object.
- **Abstraction** – Abstraction represents the behavior of an real world entity.
- **Encapsulation** – Encapsulation is the mechanism of binding the data together and hiding them from the outside world.
- **Inheritance** – Inheritance is the mechanism of making new classes from existing ones.
- **Polymorphism** – It defines the mechanism to exists in different forms.

The purpose of OO analysis and design can be described as –

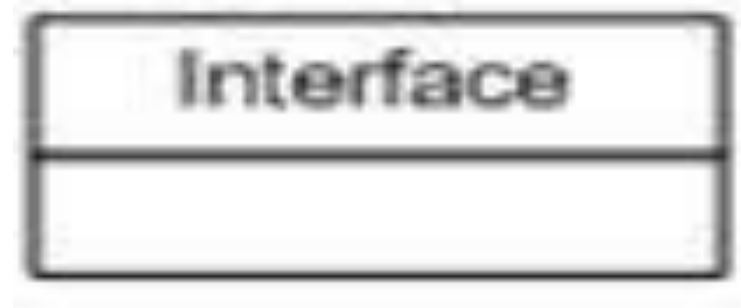
- Identifying the objects of a system.
- Identifying their relationships.
- Making a design, which can be converted to executables using OO languages.

# Structural Things

- Structural things define the static part of the model.
- They represent the physical and conceptual elements.
- Following are the brief descriptions of the structural things
- **Class** – Class represents a set of objects having similar responsibilities.



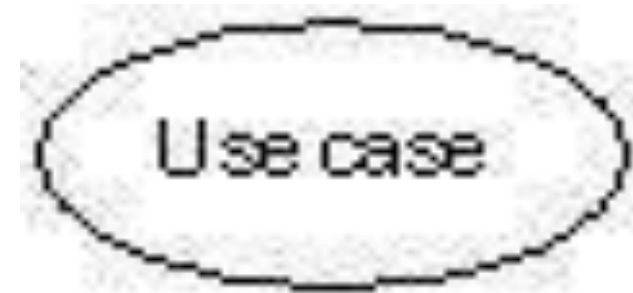
- **Interface** – Interface defines a set of operations, which specify the responsibility of a class.



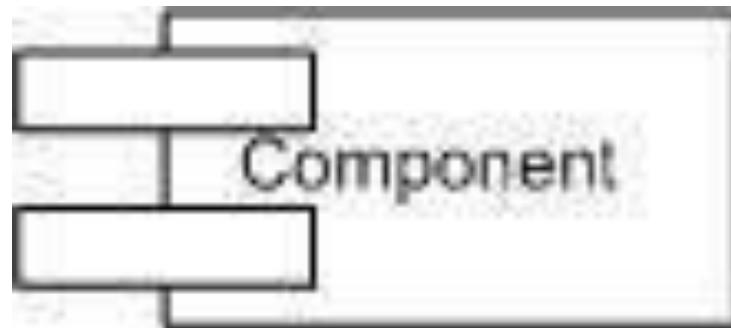
- **Collaboration** – Collaboration defines an interaction between elements.



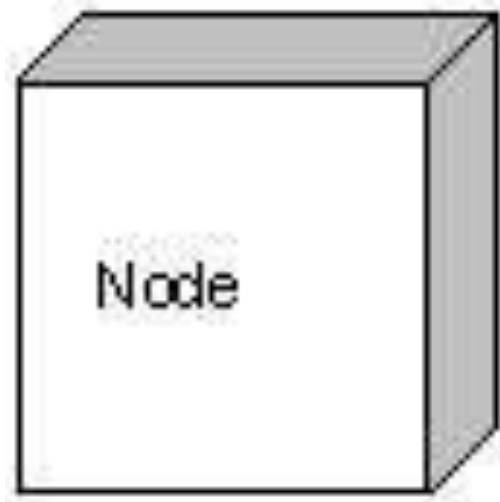
- **Use case** – Use case represents a set of actions performed by a system for a specific goal.



- **Component** – Component describes the physical part of a system.



- **Node** – A node can be defined as a physical element that exists at run time.





- **Structural things**
- **Structural diagrams**
- **Importance of Modeling**
- **Conceptual Modeling of UML**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Structural things</b></li><li>• <b>Structural diagrams</b></li><li>• <b>Importance of Modeling</b></li><li>• <b>Conceptual Modeling of UML</b></li></ul>	<ul style="list-style-type: none"><li>• Importance of Structural behaviour</li><li>• Representing the importance of modeling concept for object oriented development in system.</li></ul>	<ol style="list-style-type: none"><li>1. Importance of Structural behaviour</li><li>2. Describe the importance of Object Oriented concepts, UML diagrams and their relationships.</li></ol>

# Outcome achieved



## Name of the topic:

<b>Students will be able to do:</b>	
1	Importance of Structural behaviour
2	Summarize the role of modeling in real world environment.
3	Describe the Object Oriented concepts, UML diagrams and their relationships.

- UML plays an important role in defining different perspectives of a system. These perspectives are –
  1. Design
  2. Implementation
  3. Process
  4. Deployment
- The center is the Use Case view which connects all these four. A Use Case represents the functionality of the system.

- Hence, other perspectives are connected with use case.
- **Design** of a system consists of *classes, interfaces, and collaboration*. UML provides class diagram, object diagram to support this.
- **Implementation** defines the components assembled together to make a complete physical system. UML component diagram is used to support the implementation perspective.

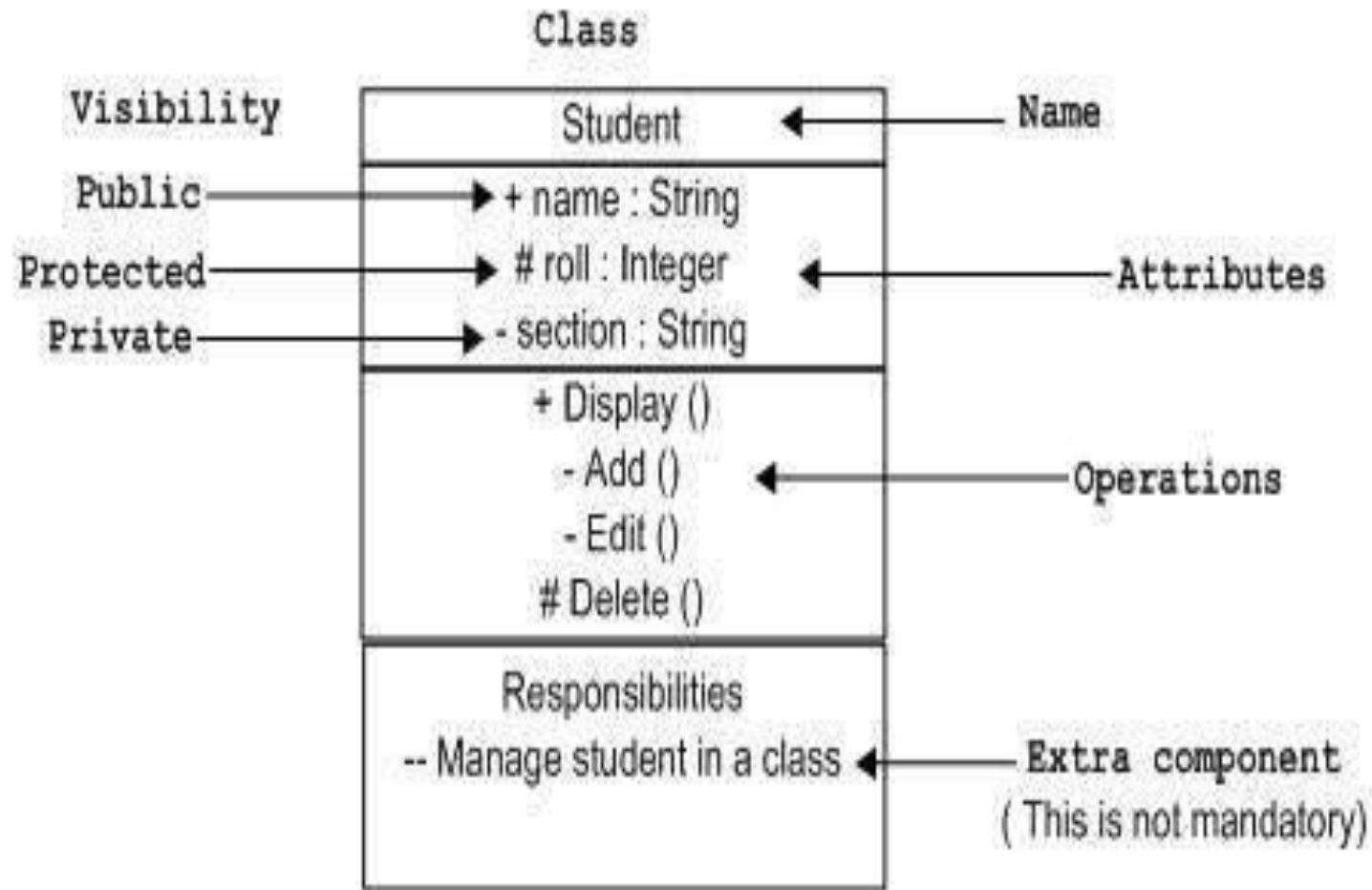
- **Process** defines the flow of the system. Hence, the same elements as used in Design are also used to support this perspective.
- **Deployment** represents the physical nodes of the system that forms the hardware. UML deployment diagram is used to support this perspective.

- **Structural model** represents the framework for the system and this framework is the place where all other components exist.
- **Behavioral model** describes the interaction in the system. It shows show the dynamic sequence of flow in a system.
- **Architectural model** represents the overall framework of the system.
- It contains both structural and behavioral elements of the system.
- Architectural model can be defined as the blueprint of the entire system.

- Graphical notations used in structural things are most widely used in UML.
- These are considered as the nouns of UML models.
  - Classes
  - Object
  - Interface
  - Collaboration
  - Use case
  - Active classes
  - Components
  - Nodes

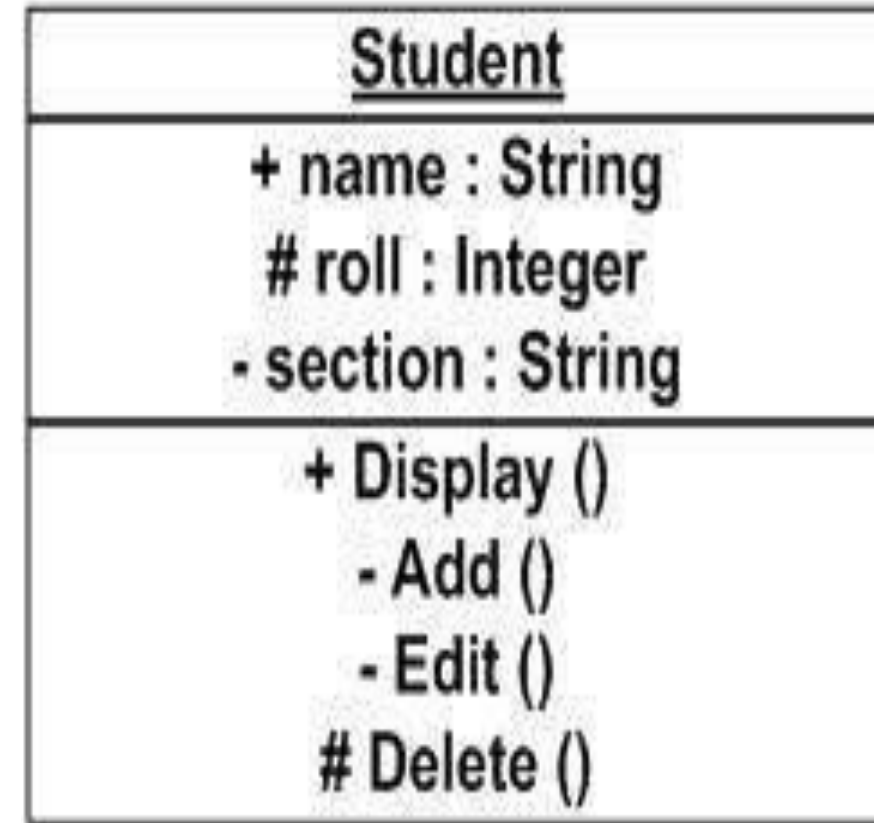


# • Class Notation



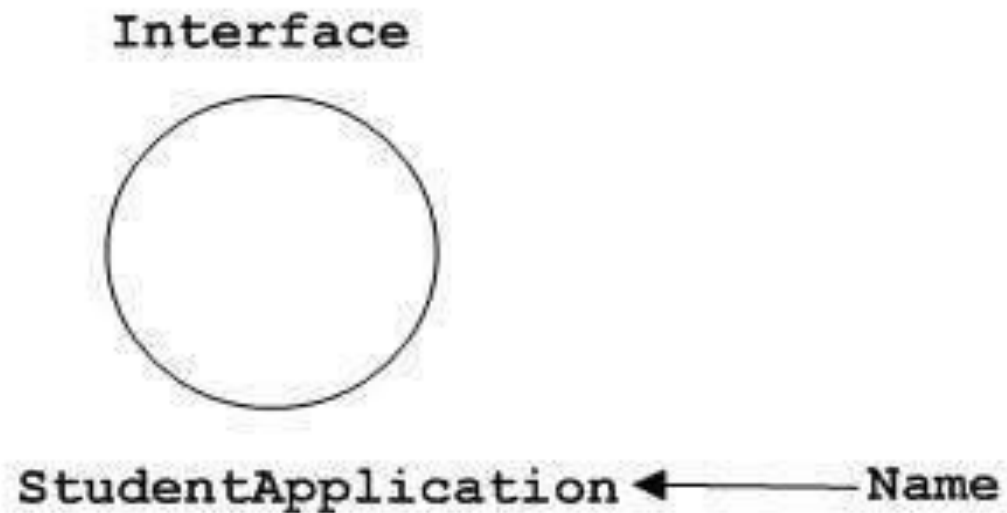
## • Object Notation

- The object is represented in the same way as the class. The only difference is the name which is underlined as shown in the following figure.
- As the object is an actual implementation of a class, which is known as the instance of a class. Hence, it has the same usage as the class.



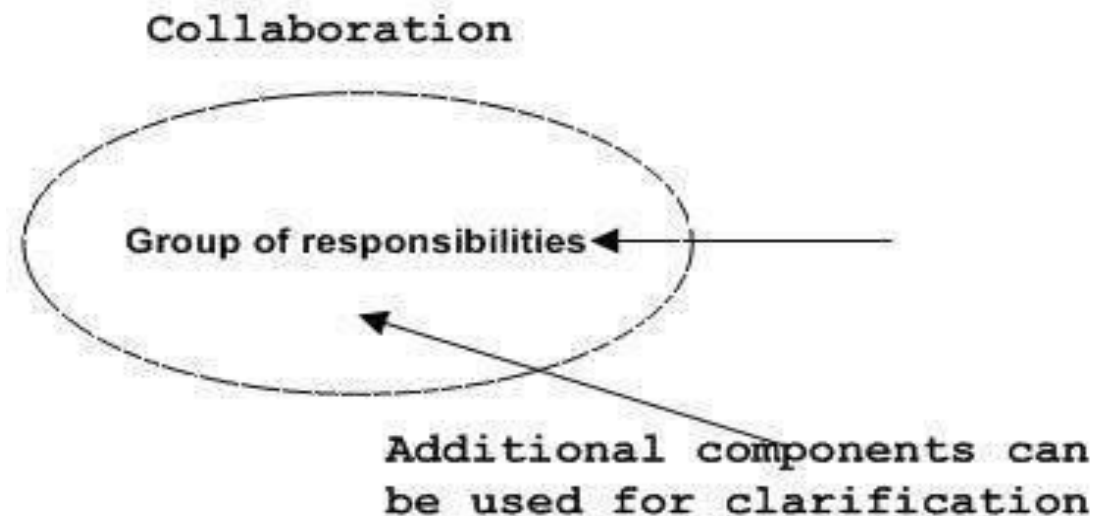
- **Interface Notation**

- Interface is represented by a circle as shown in the following figure. It has a name which is generally written below the circle.
- Interface is used to describe the functionality without implementation.



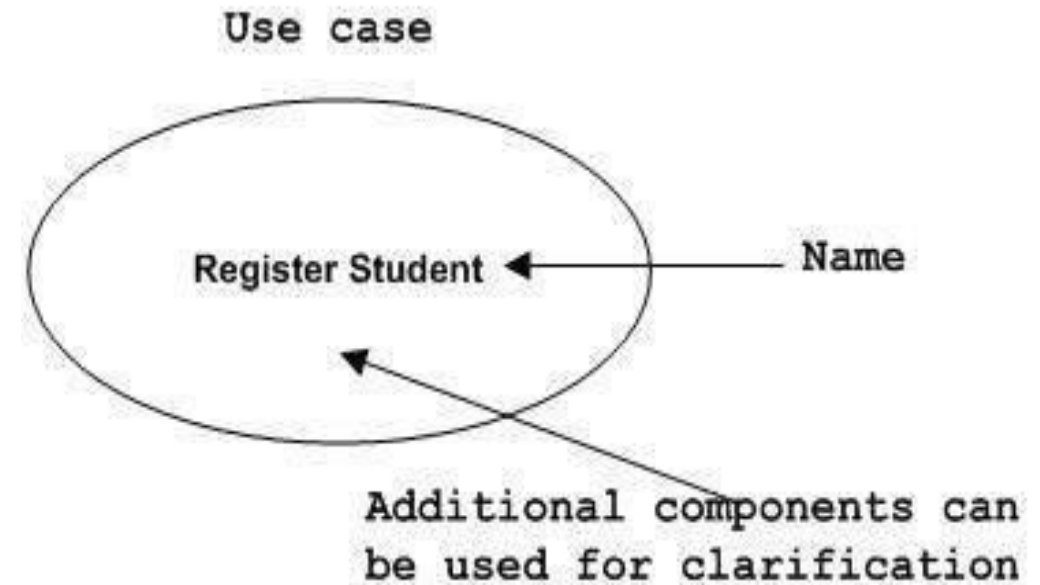
## • Collaboration Notation

- Collaboration is represented by a dotted ellipse as shown in the following figure. It has a name written inside the ellipse.
- Collaboration represents responsibilities.



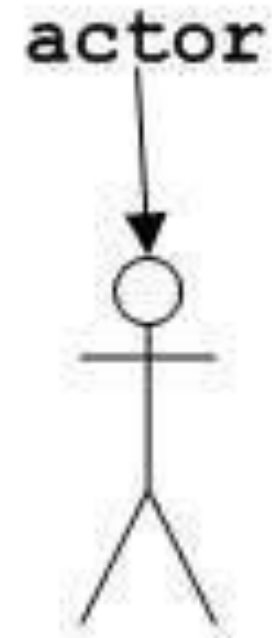
# Use Case Notation

- Use case is represented as an eclipse with a name inside it. It may contain additional responsibilities.
- Use case is used to capture high level functionalities of a system.



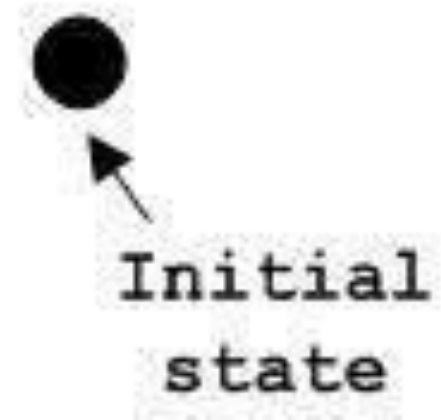
# Actor Notation

- An actor can be defined as some internal or external entity that interacts with the system.



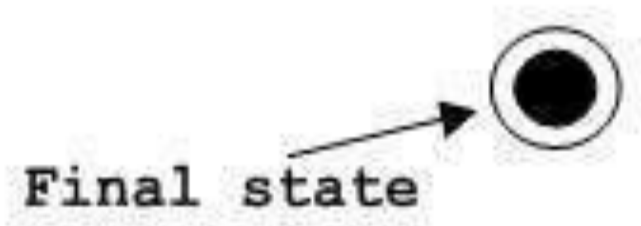
- **Initial State Notation**

- Initial state is defined to show the start of a process. This notation is used in almost all diagrams.



- **Final State Notation**

- Final state is used to show the end of a process.

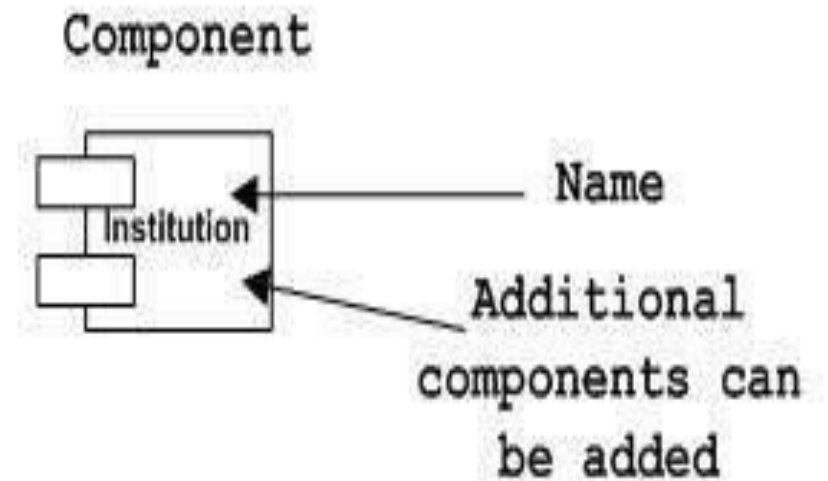


- **Component Notation**

- A component in UML is shown in the following figure with a name inside.

Additional elements can be added wherever required.

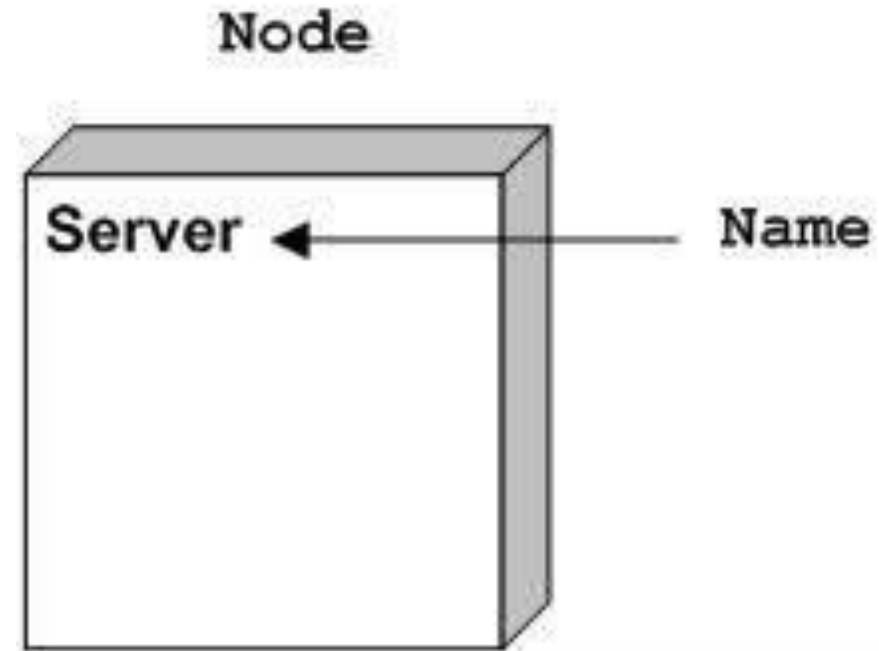
- Component is used to represent any part of a system for which UML diagrams are made.





- **Node Notation**

- A node represents the physical component of the system.
- Node is used to represent the physical part of a system such as the server, network, etc.



- The four structural diagrams are –
  - Class diagram
  - Object diagram
  - Component diagram
  - Deployment diagram

- **Class Diagram**

- Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, associations, and collaboration.
- Class diagram represents the object orientation of a system. Hence, it is generally used for development purpose.

- **Object Diagram**

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

- **Component Diagram**

- Component diagrams represent a set of components and their relationships.
- During the design phase, software artifacts (classes, interfaces, etc.) of a system are arranged in different groups depending upon their relationship. Now, these groups are known as components.
- Finally, it can be said component diagrams are used to visualize the implementation.

- **Deployment Diagram**

- Deployment diagrams are a set of nodes and their relationships.
- Component diagrams are dependent upon the classes, interfaces, etc. which are part of class/object diagram.
- Again, the deployment diagram is dependent upon the components, which are used to make component diagrams.

- **Behavioral Diagrams**
- **UML Architecture**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Behavioral Diagrams</b></li><li>• <b>UML Architecture</b></li></ul>	<ul style="list-style-type: none"><li>• Understand the role and function of each UML model in software development using object-oriented approach.</li></ul>	<ol style="list-style-type: none"><li>1. Design of a new models based on requirements and documentation process for future use.</li></ol>



# Outcome achieved



## Name of the topic:

<b>Students will be able to:</b>	
1	Describe the Behavioral Diagrams of UML and importance of each.

- UML has the following five types of behavioral diagrams –
  - Use case diagram
  - Sequence diagram
  - Collaboration diagram
  - Statechart diagram
  - Activity diagram

- Use case diagrams consists of **actors**, **use cases** and their **relationships**. The diagram is used to model the system/subsystem of an application.
- Use case diagrams are a set of use cases, actors, and their relationships.
- Hence, use case diagram is used to describe the relationships among the functionalities and their internal/external controllers. These controllers are known as actors.

- A single use case diagram captures a particular functionality of a system.
- To model a system, the most important aspect is to **capture the dynamic behavior**.
- Dynamic behavior means the behavior of the system when it is running/operating.
- Now as we have to discuss that the use case diagram is dynamic in nature, there should be some internal or external factors for making the interaction.

# Purpose of Use Case Diagrams



- Use case diagrams are used to gather the requirements of a system including internal and external influences.
- These requirements are mostly design requirements. Hence, when a system is analyzed to gather its functionalities, use cases are prepared and actors are identified.

In brief, the purposes of use case diagrams can be said to be as follows –

1. Used to gather the requirements of a system.
2. Used to get an outside view of a system.
3. Identify the external and internal factors influencing the system.
4. Show the interaction among the requirements and actors.

# How to Draw a Use Case Diagram?



- When the requirements of a system are analyzed, the functionalities are captured in use cases.
- **Use cases** are nothing but the system functionalities written in an organized manner.
- **Actors** can be defined as something that interacts with the system.

- When we are planning to draw a use case diagram, should have the following items identified.
  - ✓ Functionalities to be represented as use case
  - ✓ Actors
  - ✓ Relationships among the use cases and actors.
- After identifying the above items, we have to use the following guidelines to draw an efficient use case diagram



- The name of a use case is very important. The name should be chosen in such a way so that it can identify the functionalities performed.
- Give a suitable name for actors.
- Show **relationships** and **dependencies** clearly in the diagram.
- Do not try to include all types of relationships, as the main purpose of the diagram is to identify the requirements.
- Use notes whenever required to clarify some important points.



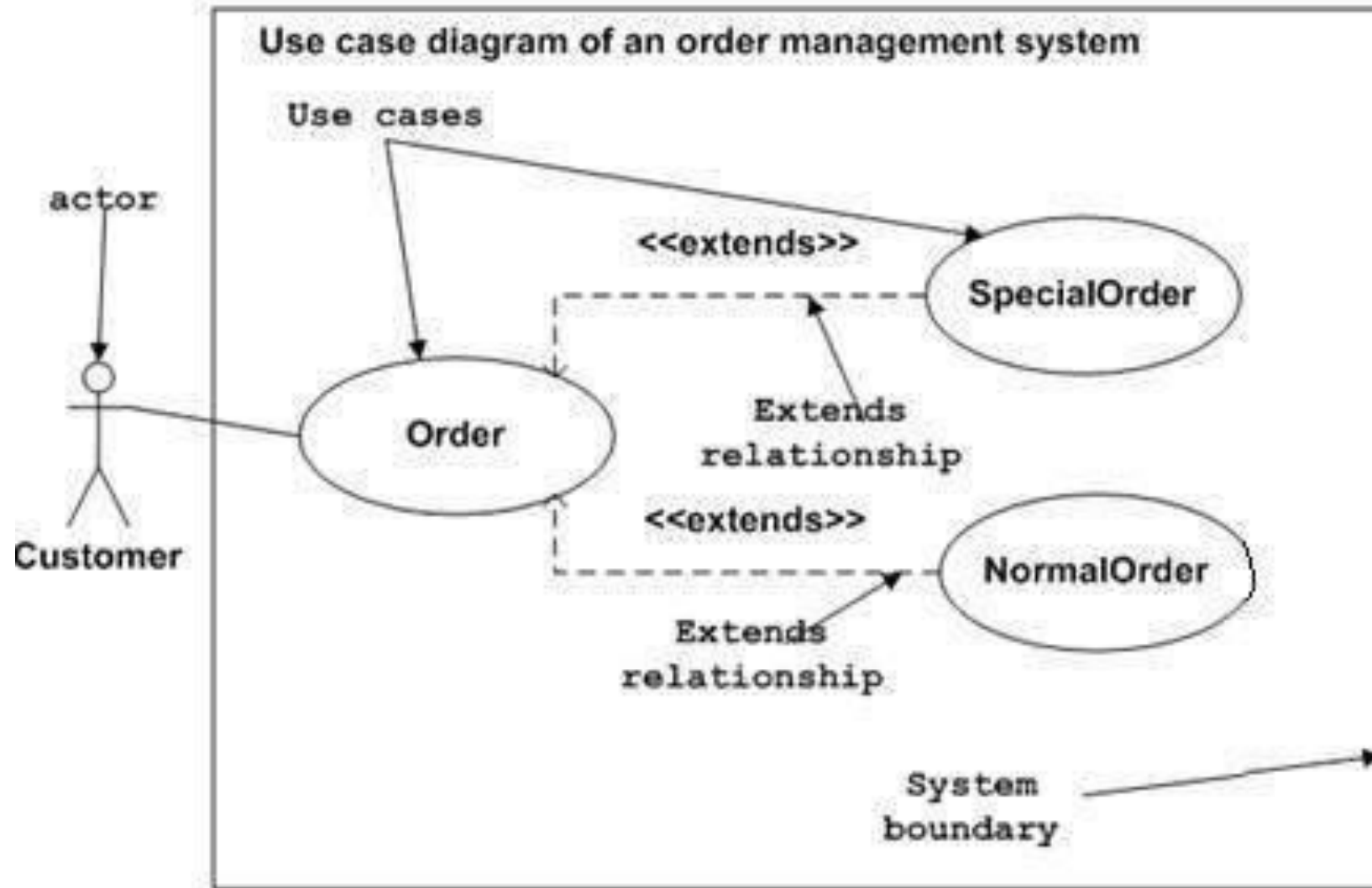


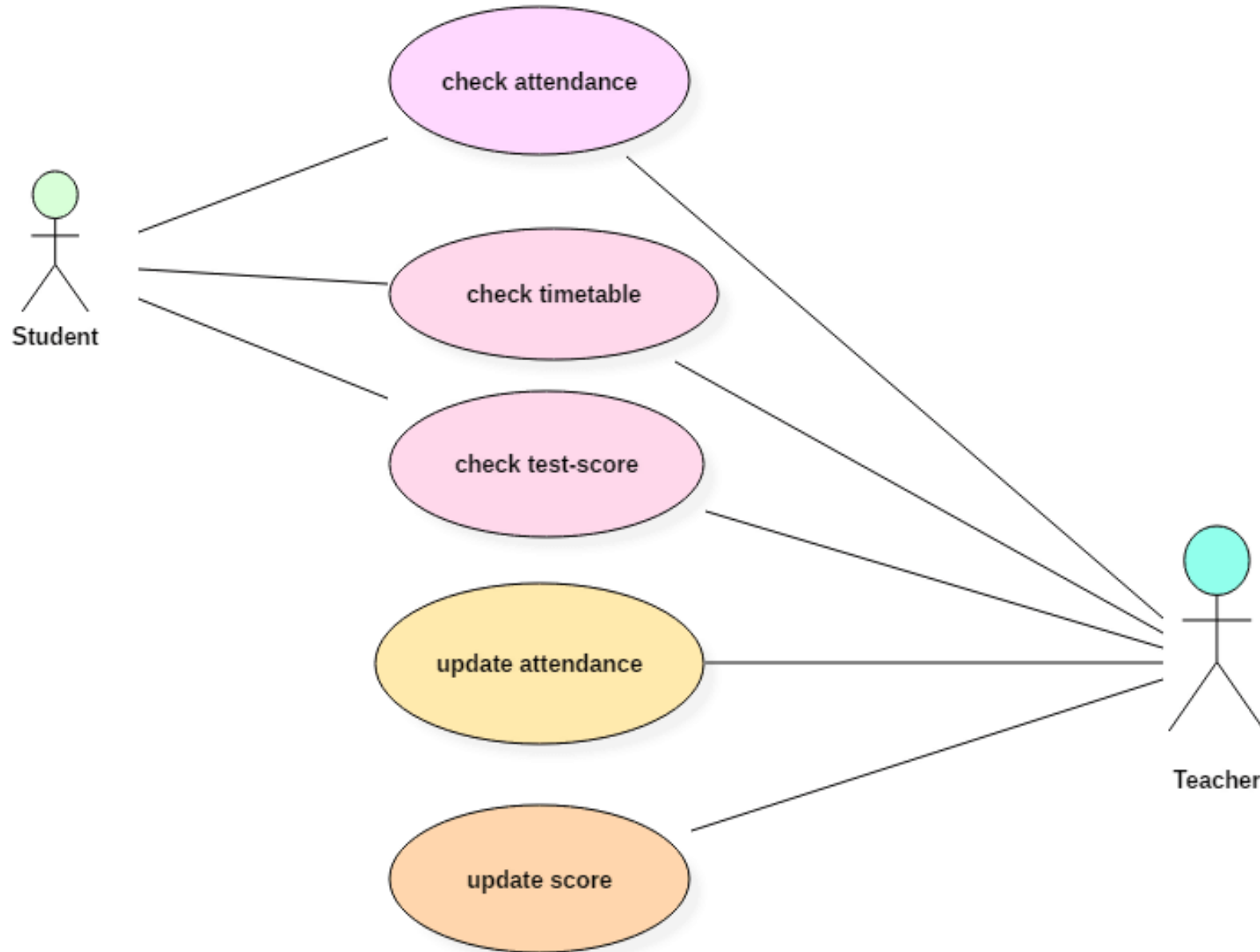
Figure: Sample Use Case diagram

- The sample use case diagram representing the order management system. Hence, if we look into the diagram then we will find three use cases (**Order, SpecialOrder, and NormalOrder**) and one actor which is the customer.
- The SpecialOrder and NormalOrder use cases are extended from *Order* use case. Hence, they have extended relationship.
- Another important point is to identify the **system boundary**, which is shown in the picture. The actor (Customer) lies outside the system as it is an external user of the system.

Use case diagrams can be used for –

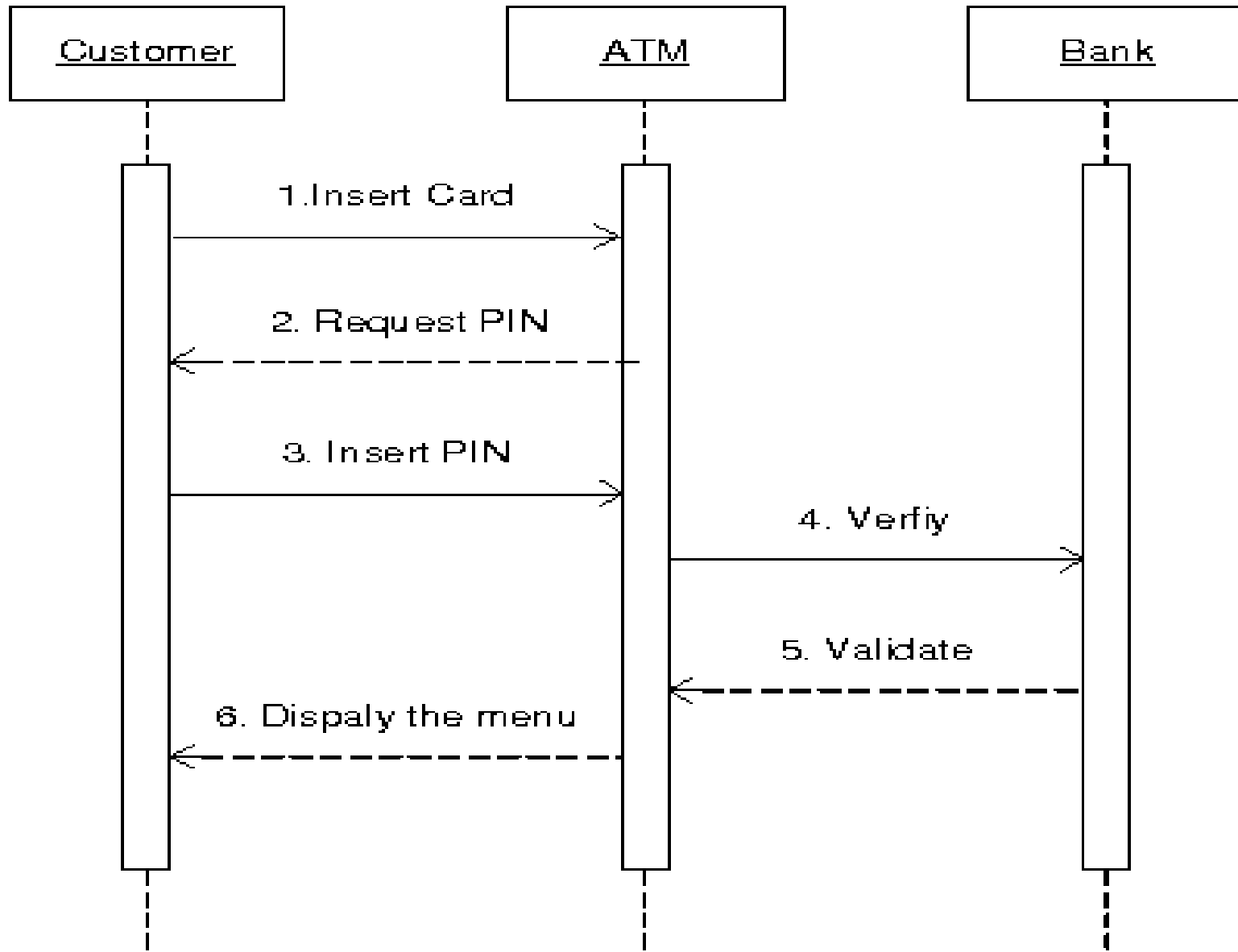
- Requirement analysis and high level design.
  - Model the context of a system.
  - Reverse engineering.
  - Forward engineering.
- In ***forward engineering***, use case diagrams are used to make test cases and in ***reverse engineering*** use cases are used to prepare the requirement details from the existing application.

# An example of a use-case diagram



# Sequence Diagram

- A sequence diagram is an interaction diagram. From the name, it is clear that the diagram deals with some sequences, which are the sequence of messages flowing from one object to another.
- Sequence diagram is used to visualize the sequence of calls in a system to perform a specific functionality.





- **Behavioral Diagrams**
- **Importance of Modeling**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Behavioral Diagrams</b></li></ul>	<ul style="list-style-type: none"><li>• Understand the role and function of each UML model in software development using object-oriented approach.</li></ul>	<ol style="list-style-type: none"><li>1. Design of a new models based on requirements and documentation process for future use.</li></ol>

# Outcome achieved



## Name of the topic:

### Students will be able to:

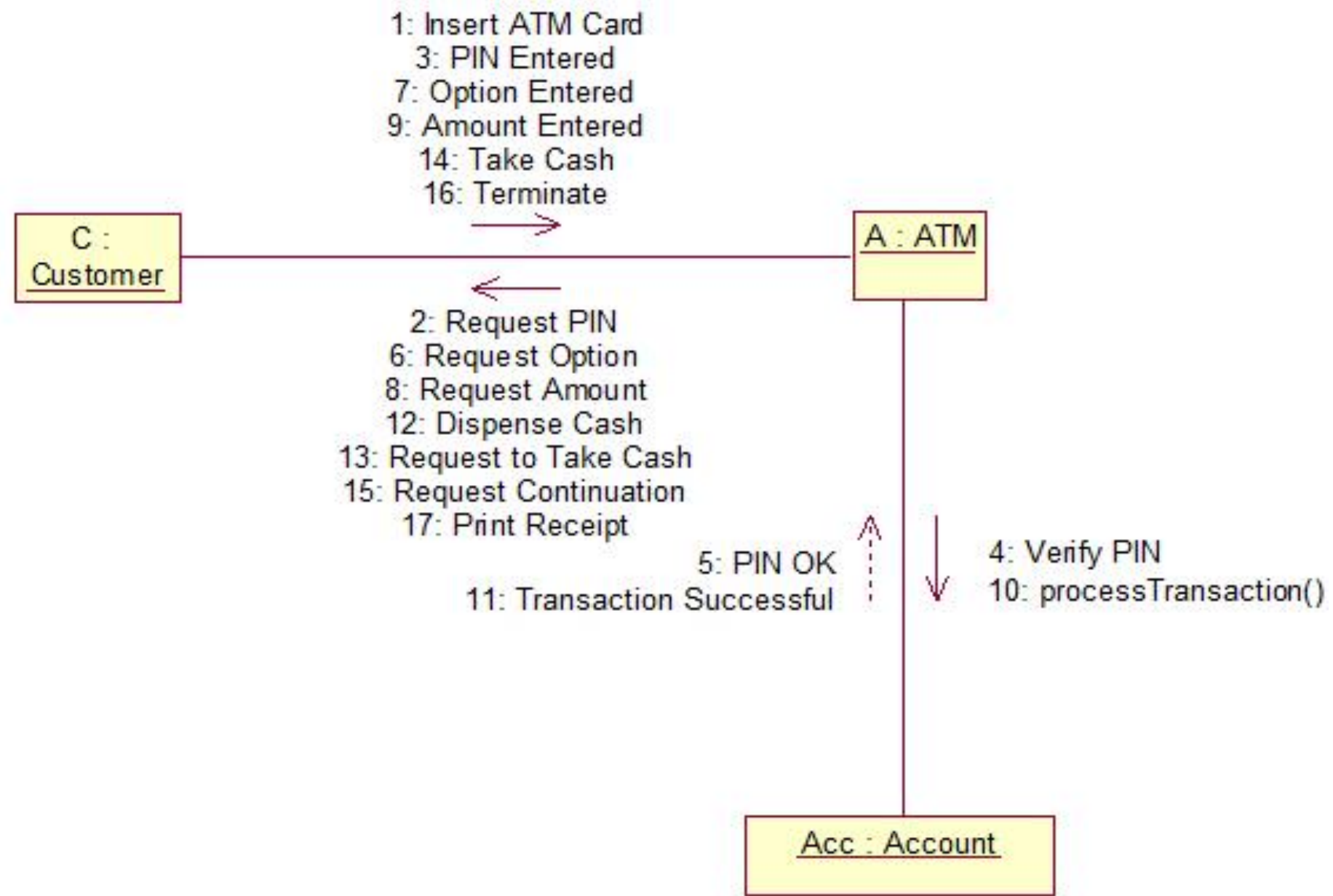
1

Describe the Behavioral Diagrams of UML and importance of each.

- UML has the following five types of behavioral diagrams –
  - Use case diagram
  - Sequence diagram
  - Collaboration diagram
  - Statechart diagram
  - Activity diagram

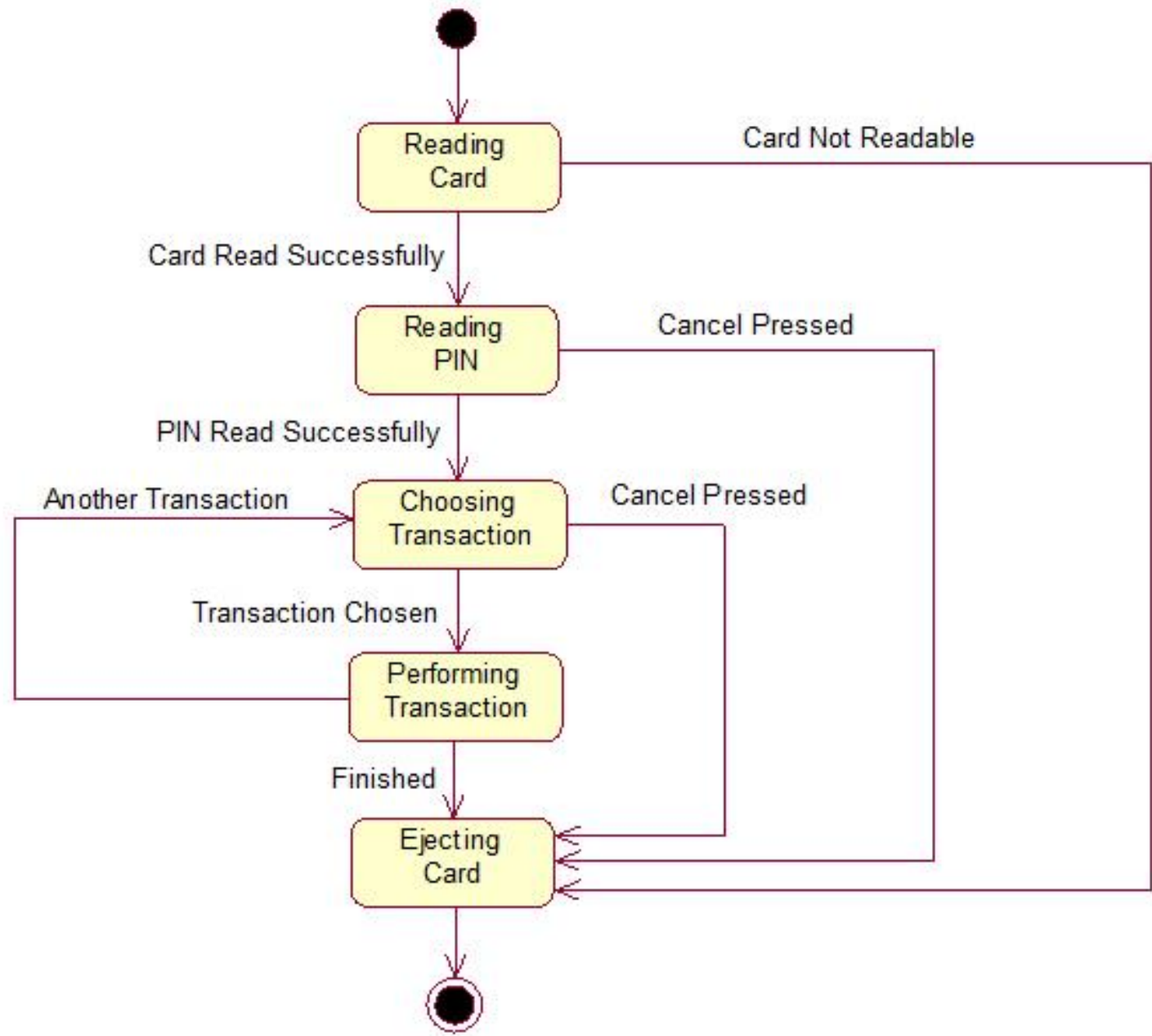
- **Collaboration Diagram**

- Collaboration diagram is another form of interaction diagram.
- It represents the structural organization of a system and the messages sent/received.
- The purpose of collaboration diagram is similar to sequence diagram.
- However, the specific purpose of collaboration diagram is to visualize the organization of objects and their interaction.



- **Statechart Diagram**

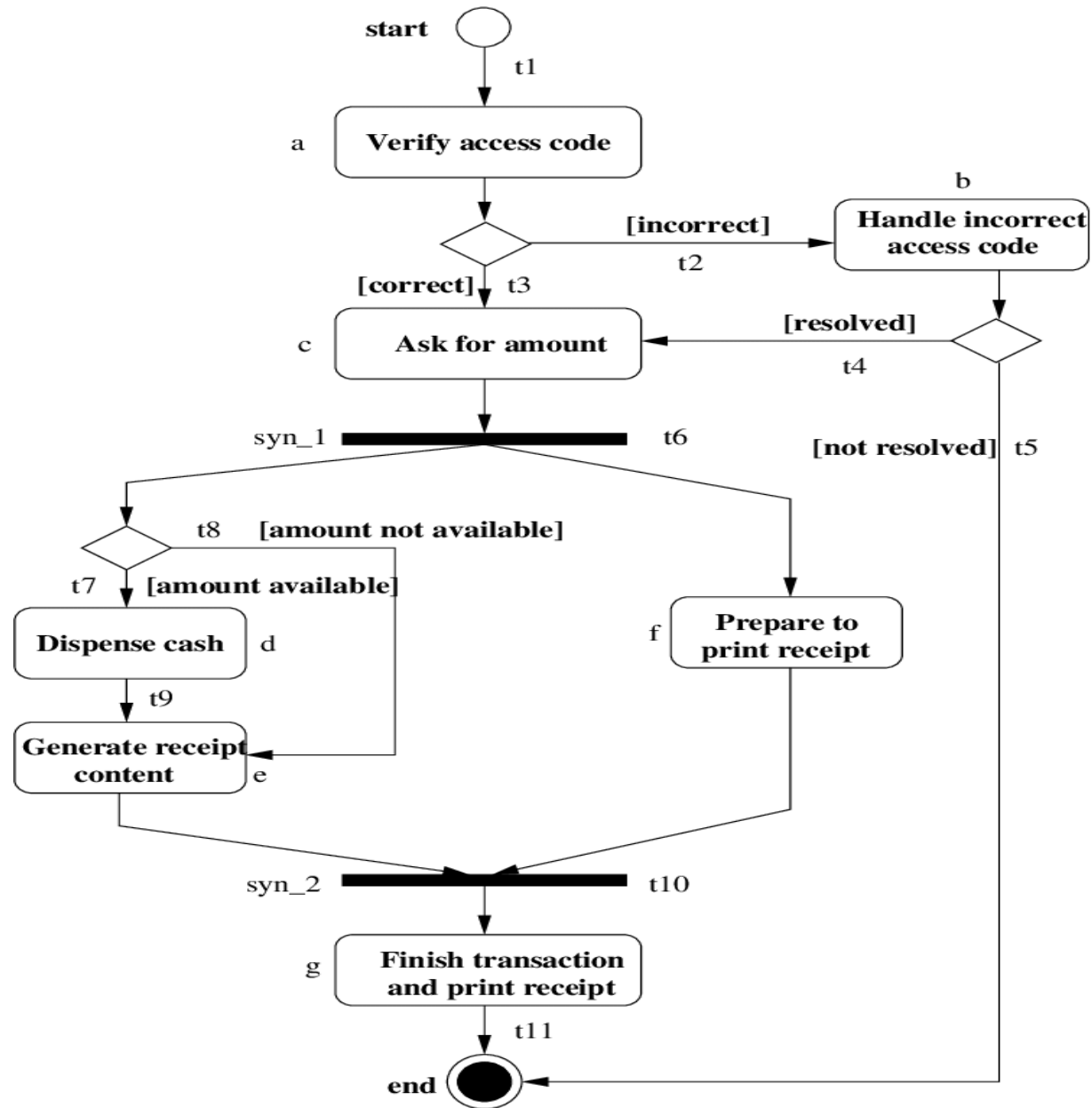
- Any real-time system is expected to be reacted by some kind of internal/external events.
- These events are responsible for state change of the system.
- State chart diagram is used to visualize the reaction of a system by internal/external factors.





- **Activity Diagram**

- Activity diagram describes the flow of control in a system.
- It consists of activities and links.
- The flow can be sequential, concurrent, or branched.
- Activities are nothing but the functions of a system.
- This is prepared to have an idea of how the system will work when executed.



# Importance of Modeling



- To know the importance of modeling let us assume that you are going to build a dog house, a house for your family and a high rise office for a client.
- In the case of a dog house you need minimal resources and the satisfaction of the dog is not that important.
- In the case of building a house for your family, you need to satisfy the requirements of your family members and the amount resources are non-trivial.

- In the case of building a high rise office, the amount of risk is very high.
- Unsuccessful software projects fail in their own unique ways, but all successful software projects are alike in many ways.
- There are many elements that contribute to a successful software organization; one common element is the use of modeling.

- Modeling is a proven and well-accepted engineering technique.
- We build architectural models of houses and high rises to help their users visualize the final product.
- Modeling is not only limited to the construction industry, it is also applied in the fields of aeronautics, automobile, picture, sociology, economics, software development and many more.

- We build models so that we can validate our theories or try out new ones with minimal risk and cost.
- **Why do we model?**
- We build models so that we can better understand the system we are developing.
- Every project can benefit from modeling. Modeling can help the development team better visualize the plan of their system and allow them to develop more rapidly by helping them build the right thing.

- **UML Architecture**
- **Classes and Relationships**
- **Common Mechanisms**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>UML Architecture</b></li><li>• <b>Classes and Relationships</b></li><li>• <b>Common Mechanisms</b></li></ul>	<ul style="list-style-type: none"><li>• Understand the role and function of each UML model in software development using object-oriented approach.</li><li>• Demonstrate the Conceptual model of UML architecture, common mechanisms</li></ul>	<ol style="list-style-type: none"><li>1. Recalling the procedure and architecture of SDLC.</li><li>2. Identify classes and their relationships</li></ol>



# Outcome achieved



## Name of the topic:

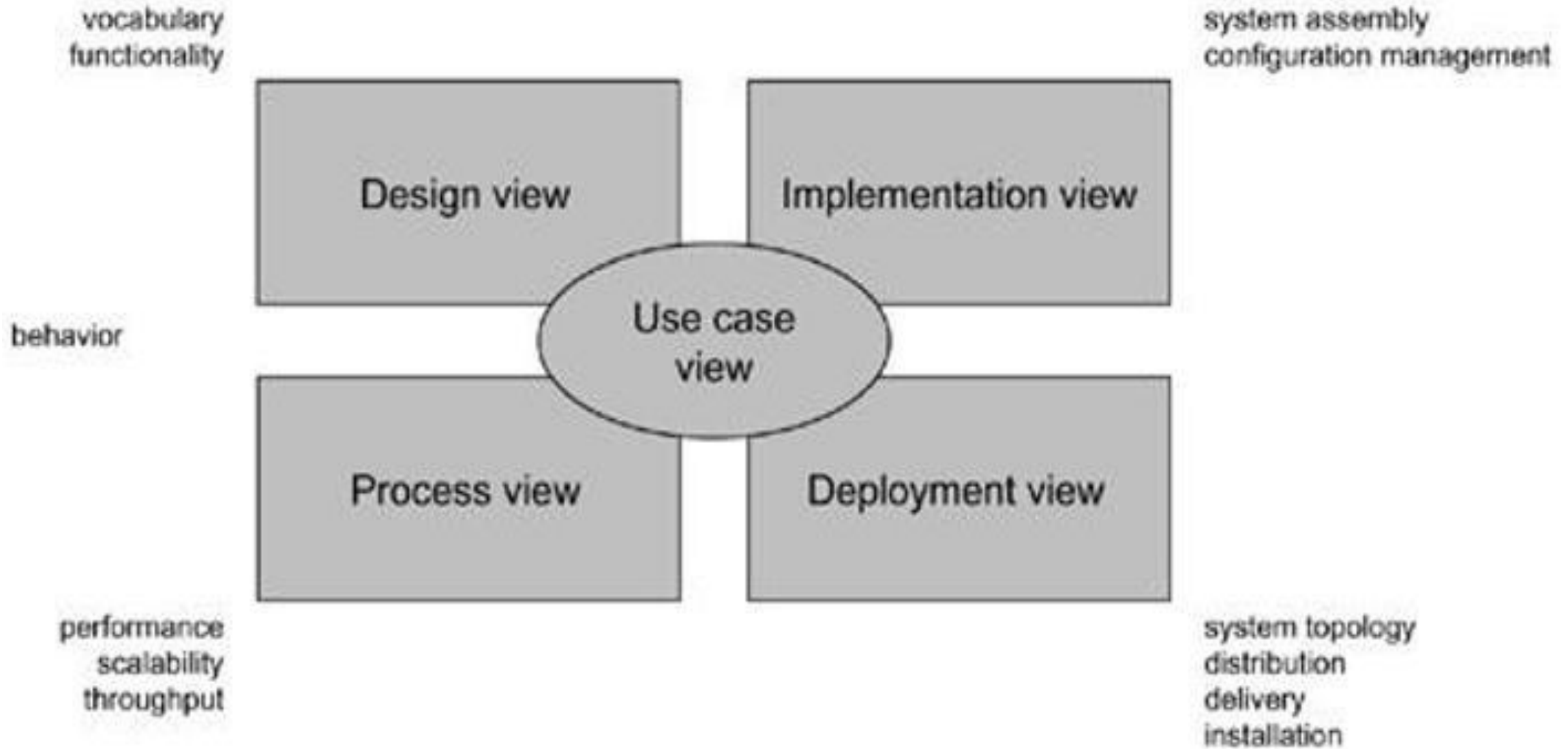
<b>Students will be able to:</b>	
1	Describe the importance of various common mechanisms used in classes and their associated relationships.
2	Apply architectural modelling techniques for design and drawing UML diagrams for real time applications.

## ***Through modeling, we achieve four aims:***

1. Models help us to visualize a system as it is or as we want it to be.
2. Models permit us to specify the structure or behavior of a system.
3. Models gives us a template that guides us in constructing a system.
4. Models document the decisions we have made.

- Any real-world system is used by **different users**.
- The users can be **developers**, testers, **business people**, **analysts**, and many more.
- Hence, before designing a system, the architecture is made with different perspectives in mind.
- The better we understand the better we can build the system.

# Architecture



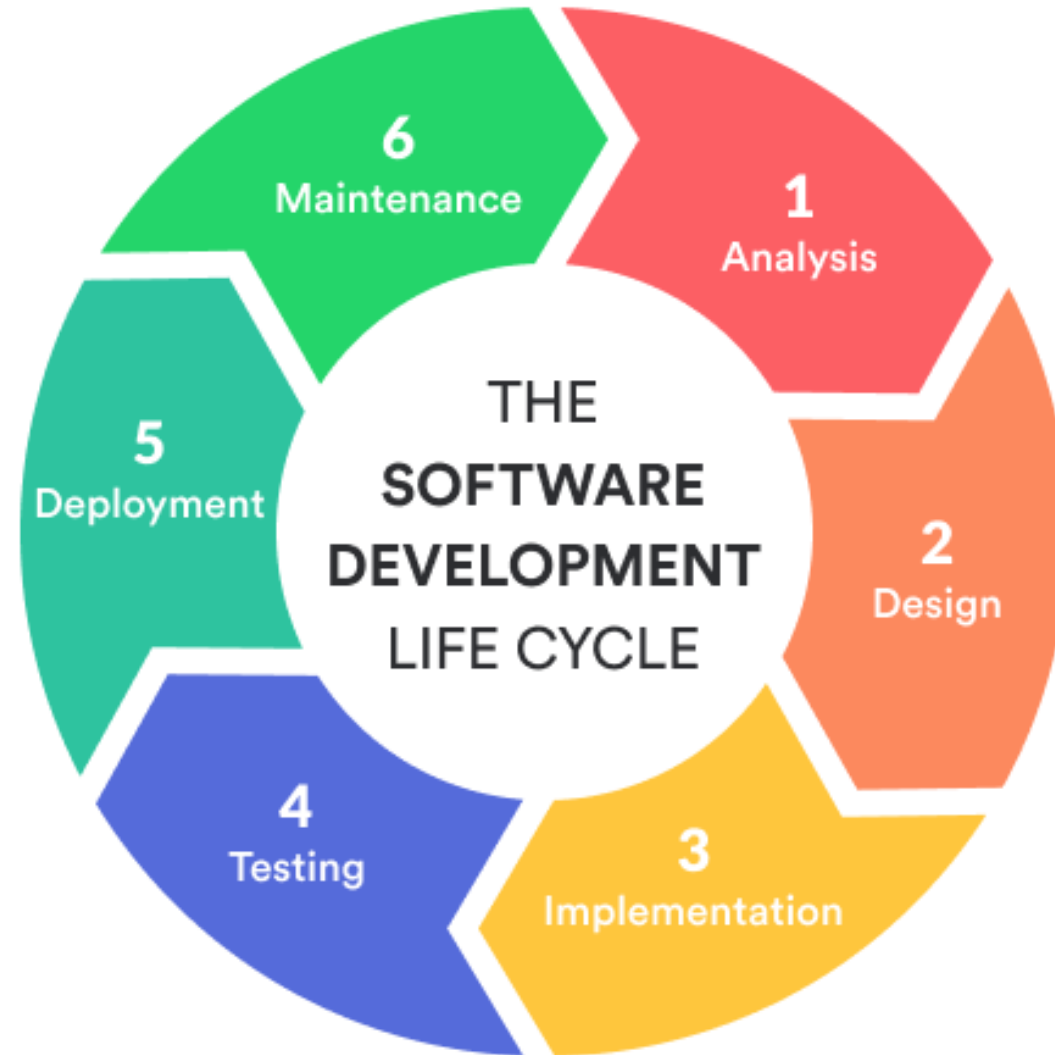
- UML is involved in each phase of the software development life cycle.
- The UML development process is
  - **Use case driven**
    - Use case driven means that use cases are used as a primary artifact for establishing the **desired behavior** of the system, for **verifying and validating** the system's architecture, for **testing**, and for **communicating** among the stakeholders of the project.

## – **Architecture-centric**

- Architecture-centric means that a system's architecture is used as a primary artifact for **conceptualizing**, **constructing**, **managing**, and **evolving** the system under development.

## – **Iterative and incremental**

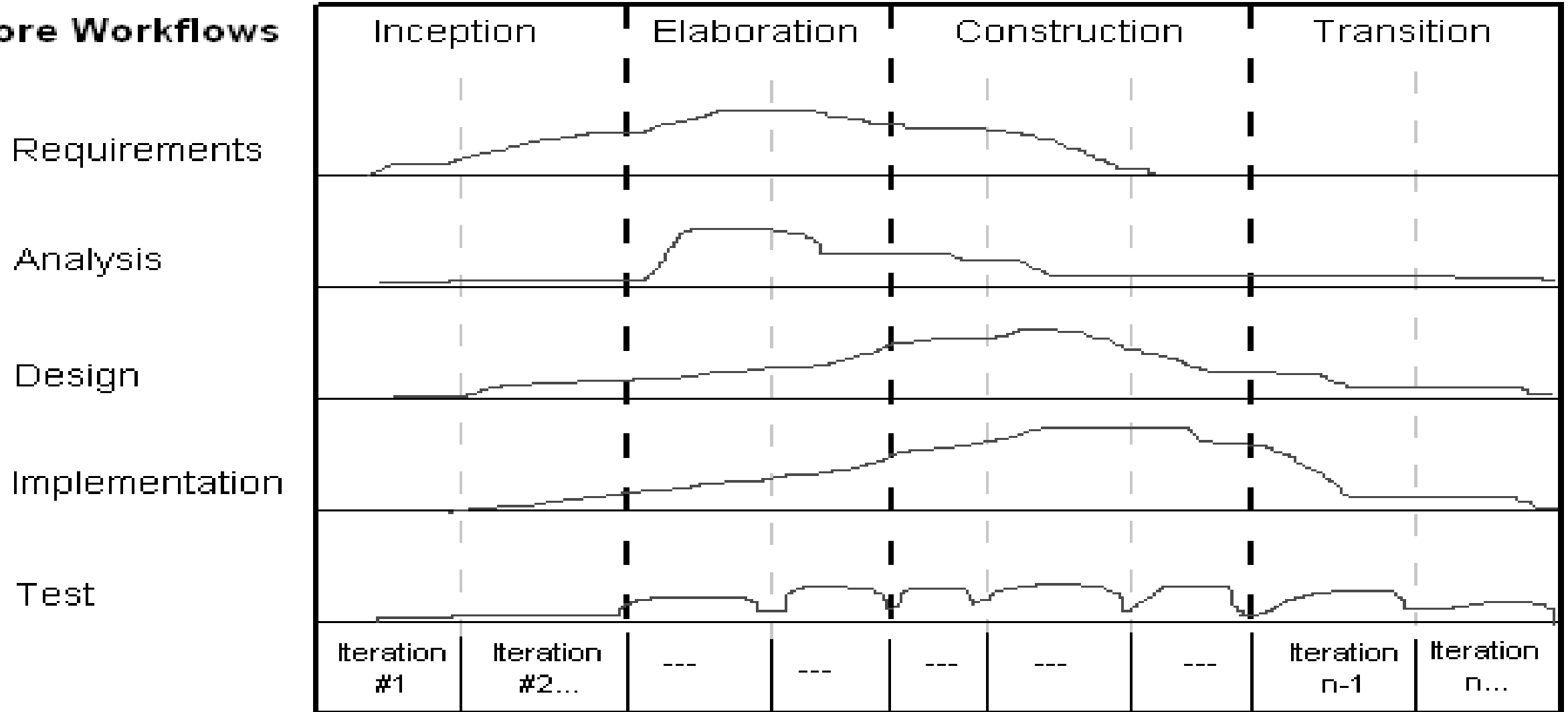
- It is an iterative process and involves managing a stream of executable releases.
- It involves the continuous integration of the system's architecture to produce these releases, with each new release representing incremental improvements over the other





## Phases

### Core Workflows



## Iterations

- Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.
- Class diagram is a static diagram.
- Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.
- Class diagram describes the attributes and operations of a class and also the constraints imposed on the system.

# Basic Class Diagram Symbols and Notations

## Classes

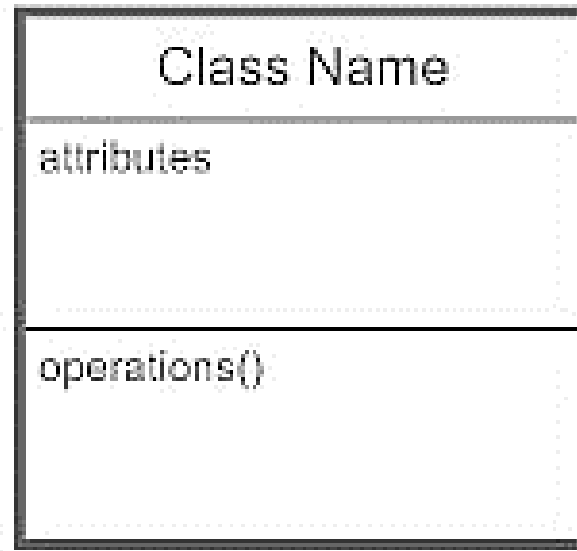
- Classes represent an abstraction of entities with common characteristics. Associations represent the relationships between classes.
- Illustrate classes with rectangles divided into compartments. Place the name of the class in the first partition (centered, bolded), list the attributes in the second partition (left-aligned, not bolded, and lowercase), and write operations into the third.



Class

- **Active Classes**

- Active classes initiate and control the flow of activity, while passive classes store data and serve other classes.



Active class

## Visibility

- Use visibility markers to signify who can access the information contained within a class.
- **Private visibility**, denoted with a - sign, hides information from anything outside the class partition.
- **Public visibility**, denoted with a + sign, allows all other classes to view the marked information.
- **Protected visibility**, denoted with a # sign, allows child classes to access information they inherited from a parent class.

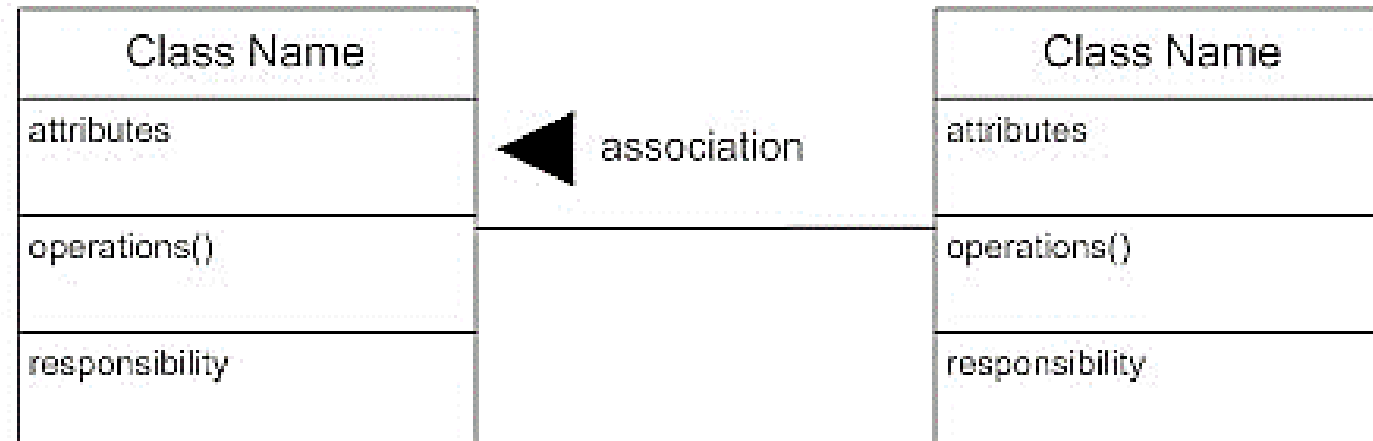
Class Name
attributes
+ public operation - private operation # protected operation

Visibility

Marker	Visibility
+	public
-	private
#	protected
?	package

# Associations

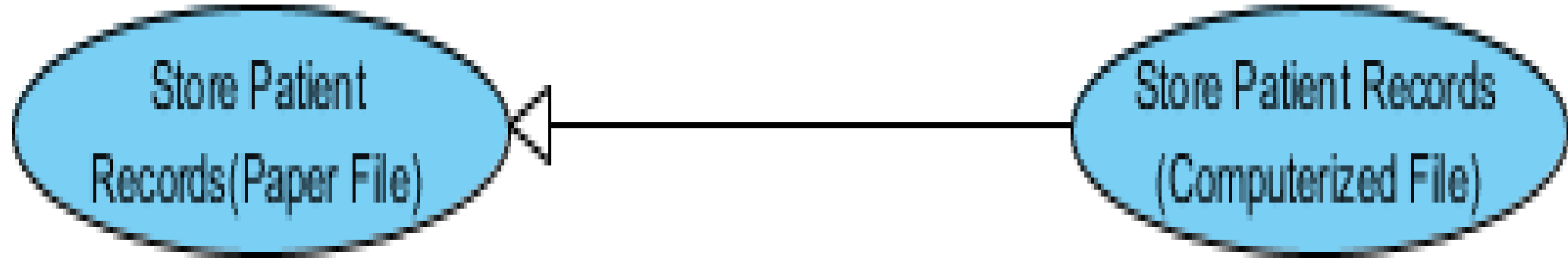
- Associations represent static relationships between classes. Place association names above, on, or below the association line. Use a filled arrow to indicate the direction of the relationship.

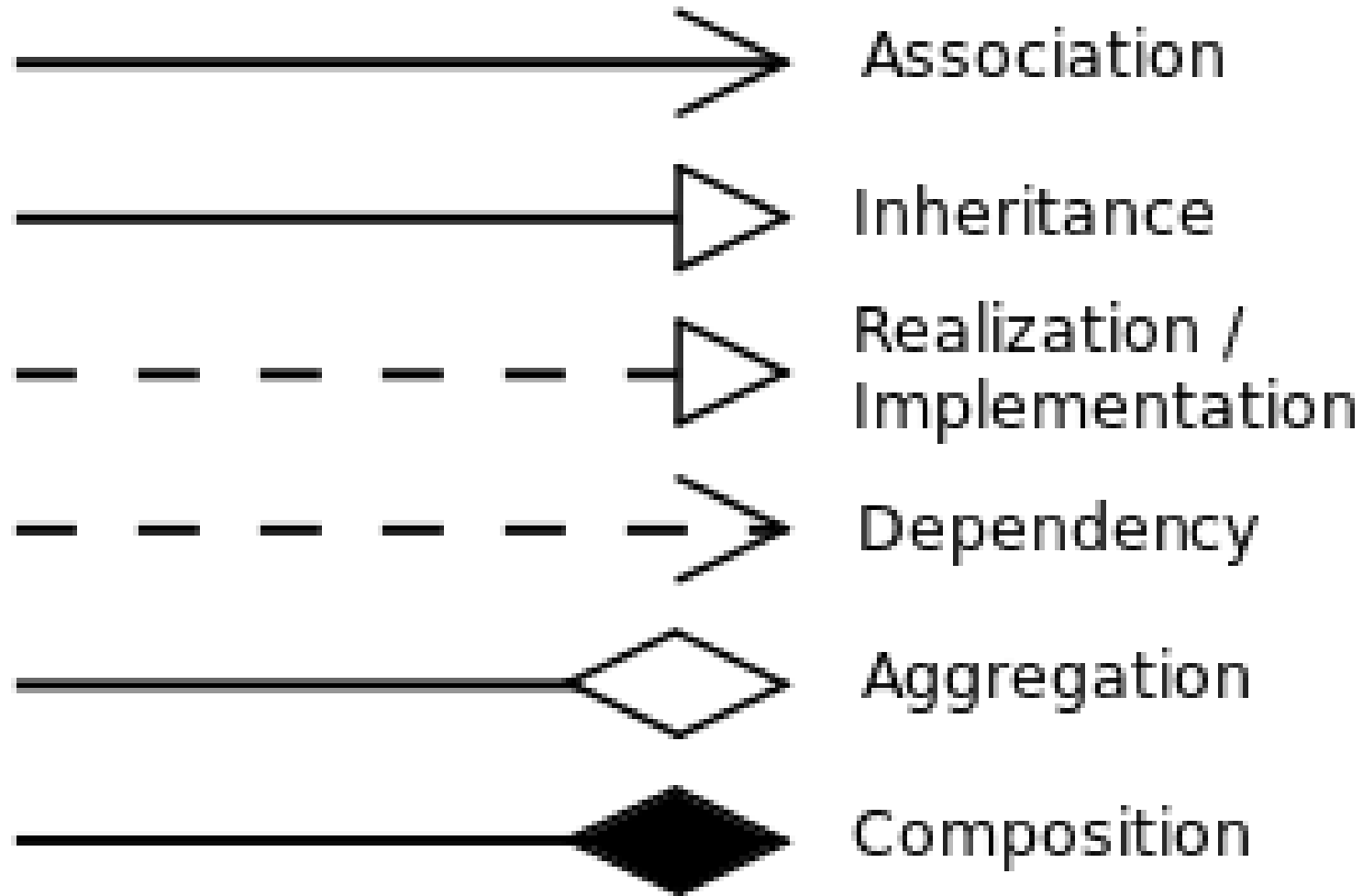


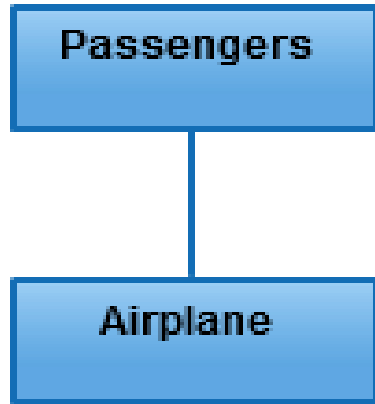
## Generalization

- A generalization relationship is a parent-child relationship between use cases.
- The child use case is an enhancement of the parent use case.
- Generalization is shown as a directed arrow with a triangle arrowhead.
- The child use case is connected at the base of the arrow. The tip of the arrow is connected to the parent use case.

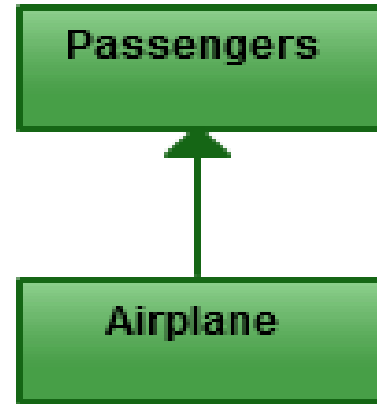








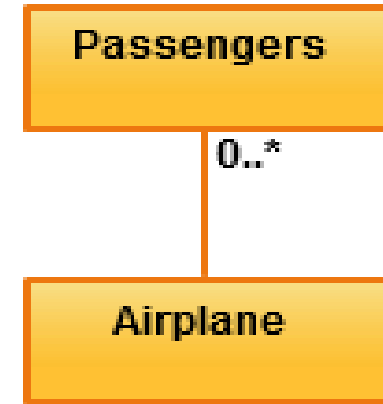
**Association**



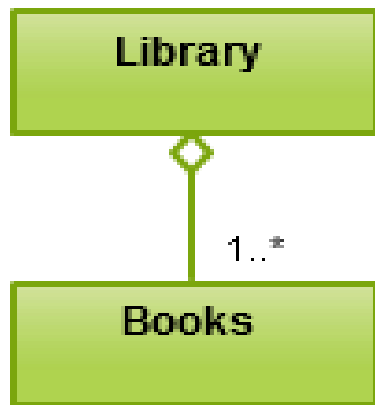
**Directed Association**



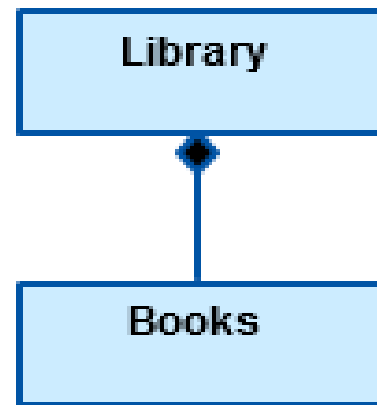
**Reflexive Association**



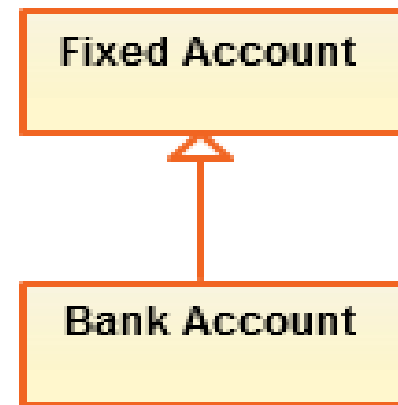
**Multiplicity**



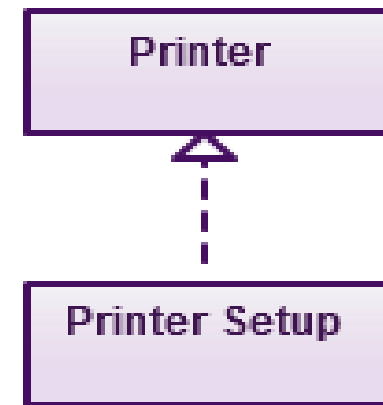
**Aggregation**



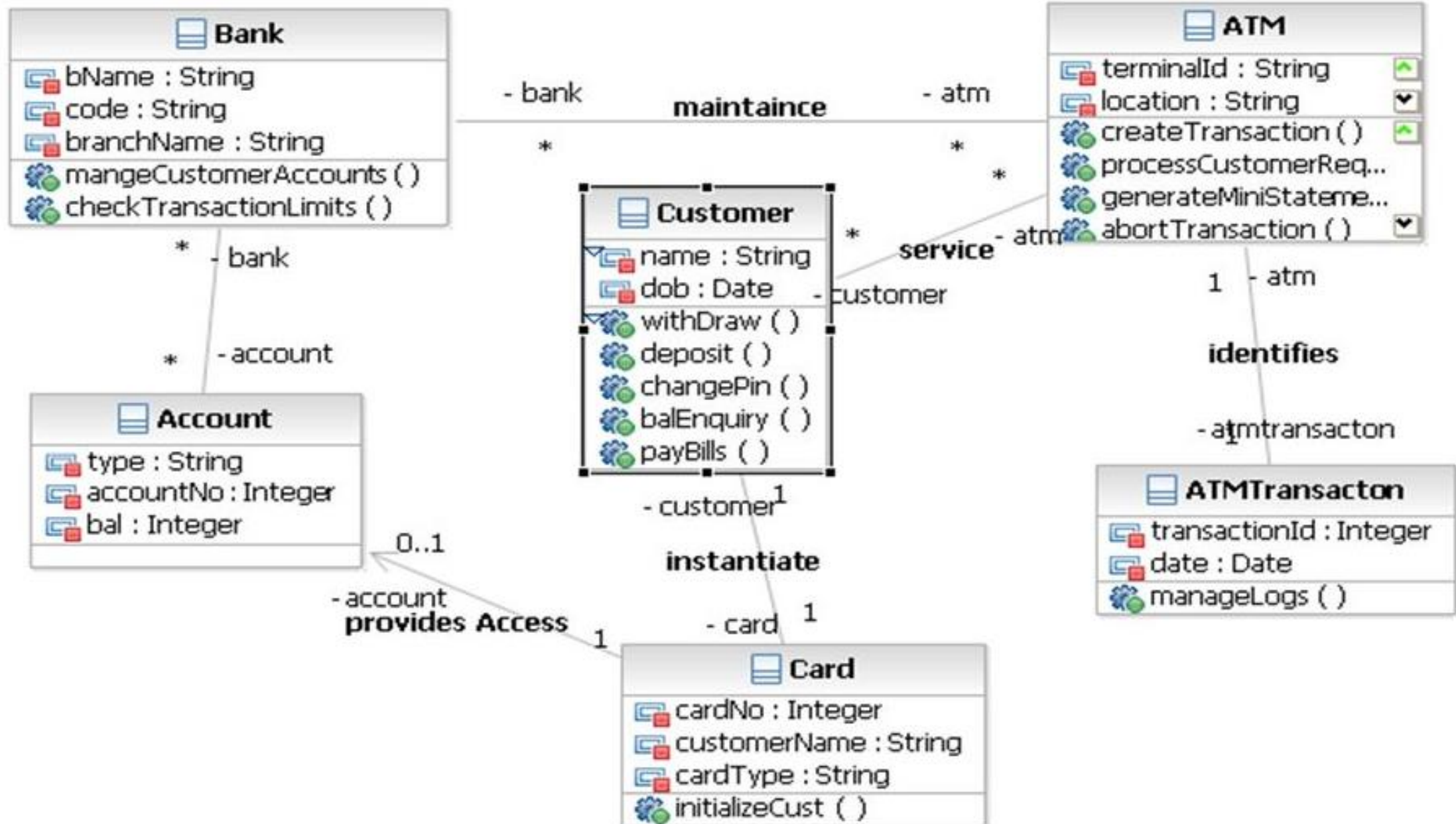
**Composition**



**Inheritance**



**Realization**



- **Classes and Relationships**
- **Common Mechanisms**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Classes and Relationships</b></li><li>• <b>Common Mechanisms</b></li></ul>	<ul style="list-style-type: none"><li>• Understand the role and function of each UML model in software development using object-oriented approach.</li><li>• Demonstrate the Conceptual model of UML architecture, common mechanisms</li></ul>	<ol style="list-style-type: none"><li>1. Recalling the procedure and architecture of SDLC.</li></ol>

# Outcome achieved



## Name of the topic:

<b>Students will be able to:</b>	
1	Describe the importance of various common mechanisms used in classes and their associated relationships.
2	Apply architectural modelling techniques for design and drawing UML diagrams for real time applications.

# Purpose of Class Diagrams



The purpose of the class diagram can be summarized as –

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.



# How to Draw a Class Diagram?

The following points should be remembered while drawing a class diagram –

- The name of the class diagram should be meaningful to describe the aspect of the system.
- Each element and their relationships should be identified in advance.
- Responsibility (attributes and methods) of each class should be clearly identified
- For each class, minimum number of properties should be specified, as unnecessary properties will make the diagram complicated.

# Where to Use Class Diagrams?



class diagrams are used for –

- Describing the static view of the system.
- Showing the collaboration among the elements of the static view.
- Describing the functionalities performed by the system.
- Construction of software applications using object oriented languages.

- UML provides common mechanism that apply throughout the language namely stereotype, tagged values and constraints

**1. Note:** It is a graphical symbol for rendering comments or constraints attached to an elements.

- Notes are used to specify things like requirements, observations, reviews and explanations.
- A note may contain any combination of text or graphics.
- A note represented by a rectangle with dog-eared corner along with text.

Note

**2. Adornments:** Adornments are added to an elements basic notation.

- It helps to visualize the details.
- It is represented by placing text at the element or with a graphic symbol to the basic notation.

**3. Stereotypes:** It provides new semantics to a mode. Which means additional features can be provided to a model, by extracting the existing properties, relevant to the situation being modelled.

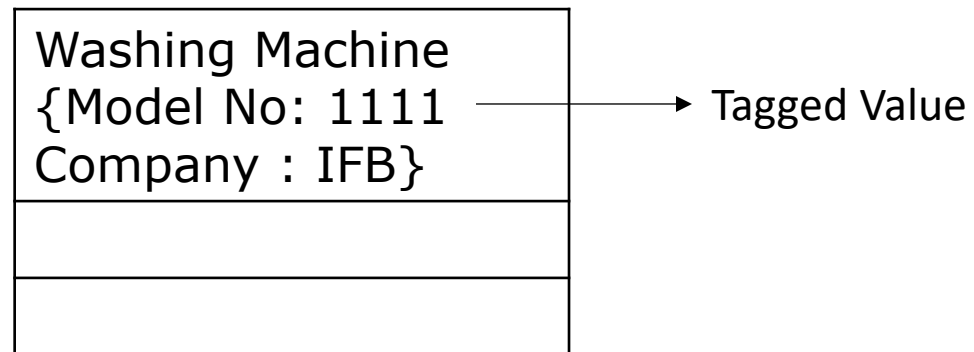
- These are included in separate rectangular boxes with type of information.
- EX: consider a class file. The major operation of files include read, write, append. When a given file name is typed, which is not existing, then a message 'File Not Found' can be represented using stereotypes in the following way.

# Representation of stereotypes



## 4. Tagged Values:

- These are used to provide the information related to an existing data
- EX: Consider a washing machine class, most probable tagged values for this class can be the “model no”, “company” etc.



**Fig: Representation of Tagged Value**



**5. Constraints:** Constraints specifies conditions that must be held true for the model to be well formed with constraints we can add new semantics or change existing rules.

**Ex:**

Cash Account..

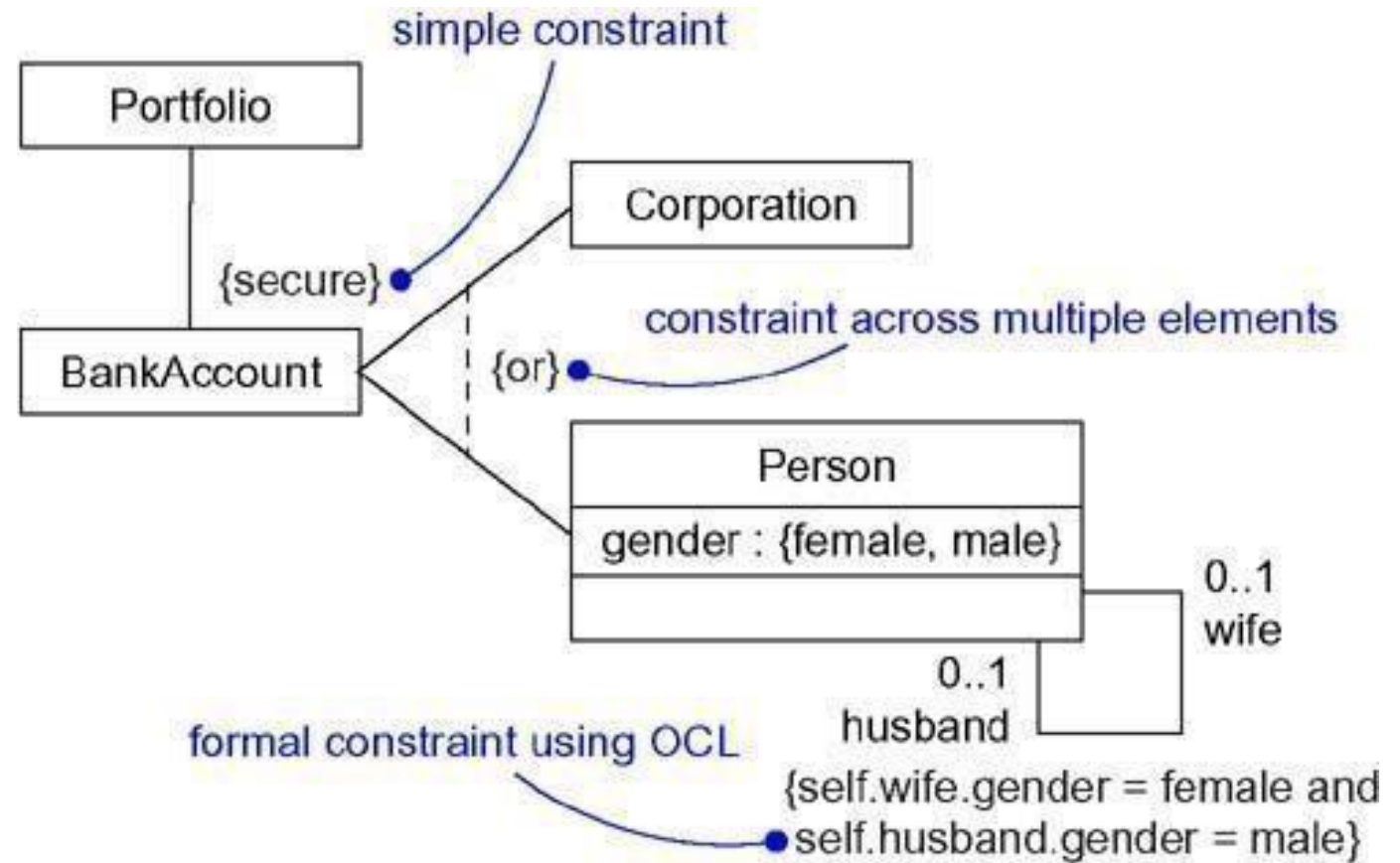
Current Balance: real;

Overdrawn : bool = (current balance < 100.0)

End

- Constraints will use boolean, relational, and arithmetic functions on values.
- Operations like  $\sim$ ,  $\wedge$ ,  $\vee$ ,  $\Rightarrow$ , and  $\sim =$  will be used for **not**, **and**, **or implies** and **not equal** to respectively.

- For example, some properties of associations (order and changeability) are rendered using constraint notation.



## 6. Standard Elements:

- ***Stereo type*** : It specifies that the classifier is a stereo type that may be applied to other elements.
- ***Documentation***: It specifies a comment, description or explanation of the elements to which it is attached.

- Class diagram is used for visualizing, specifying and documenting the structural modelling.

## **Contents of class diagram:**

1. Classes
2. Interfaces
3. Collaboration
4. Relationships

# Benefits of Class Diagram



- Class Diagram Illustrates data models for even very complex information systems
- It provides an overview of how the application is structured before studying the actual code. This can easily reduce the maintenance time
- It helps for better understanding of general schematics of an application.

# Essential elements of A UML class diagram

Essential elements of UML class diagram are:

## Class Name:

- Following rules must be taken care of while representing
- 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.
- An abstract class name should be written in italics format.

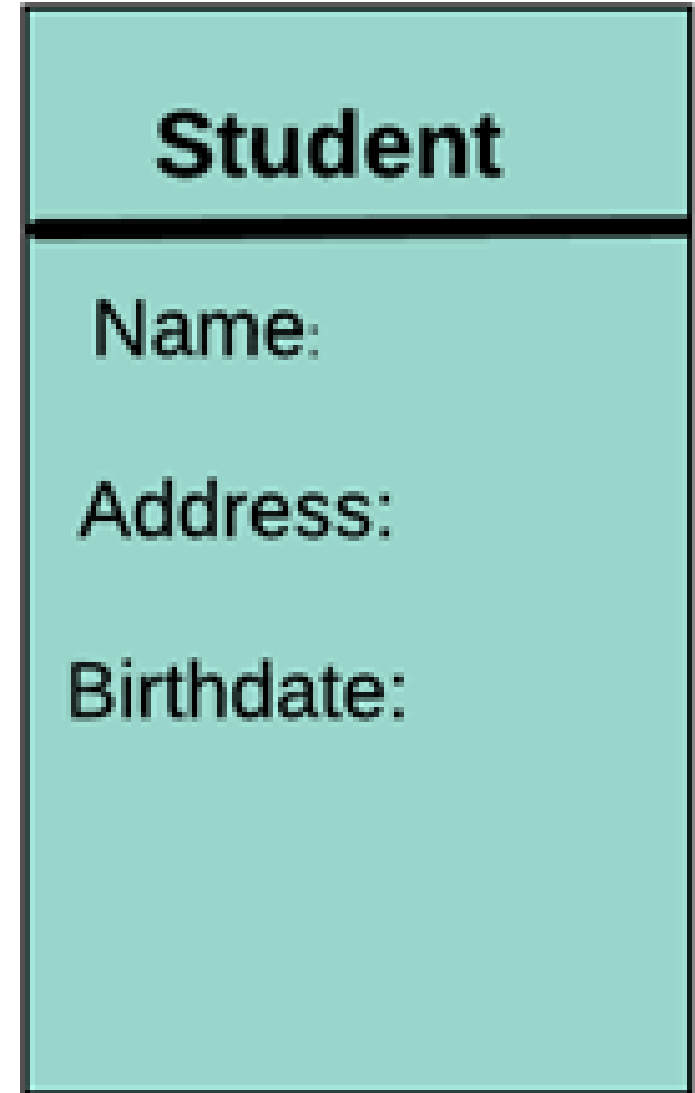
ClassName

attributes

operations

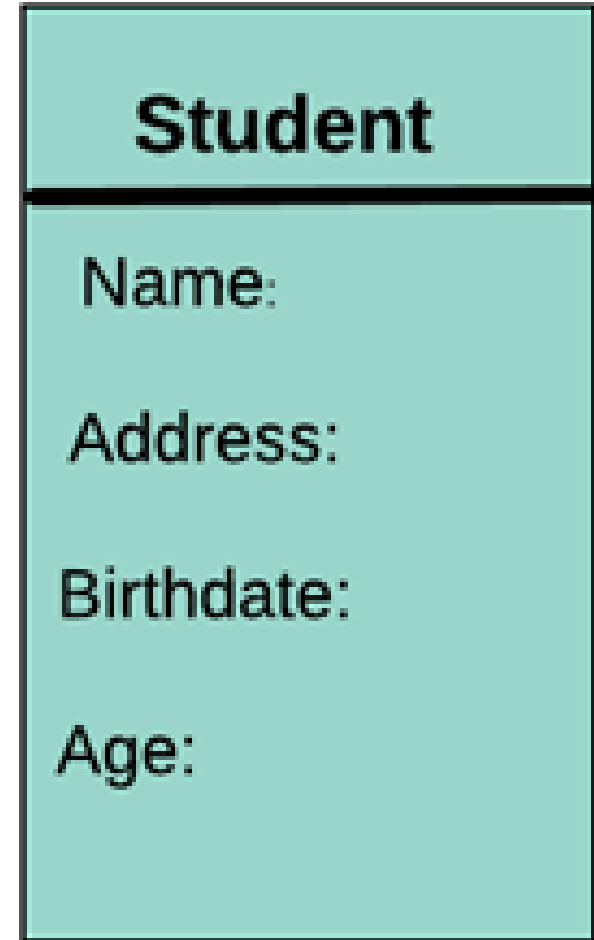
## Attributes:

- An attribute is named property of a class which describes the object being modeled.
- In the class diagram, this component is placed just below the name-compartment.





- A derived attribute is computed from other attributes.
- For example, an age of the student can be easily computed from his/her birth date.



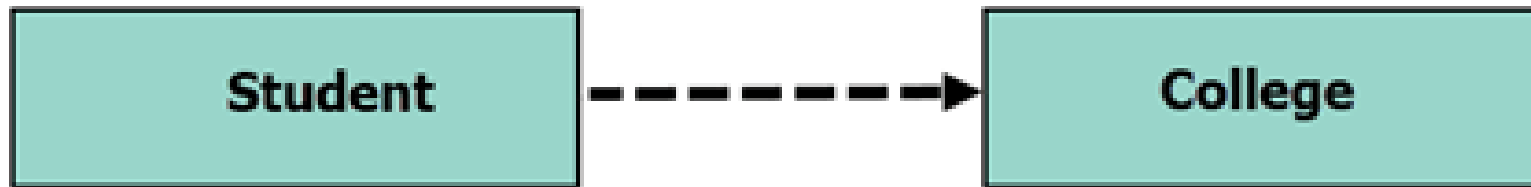
- Attributes must have a meaningful name that describes the use of it in a class.
- The attributes are generally written along with the visibility factor.
- Visibility describes the accessibility of an attribute of a class.
- Public, private, protected and package are the four visibilities which are denoted by +, -, #, or ~ signs respectively.

# Relationships

- There are mainly three kinds of relationships in UML:
  1. Dependencies
  2. Generalizations
  3. Associations

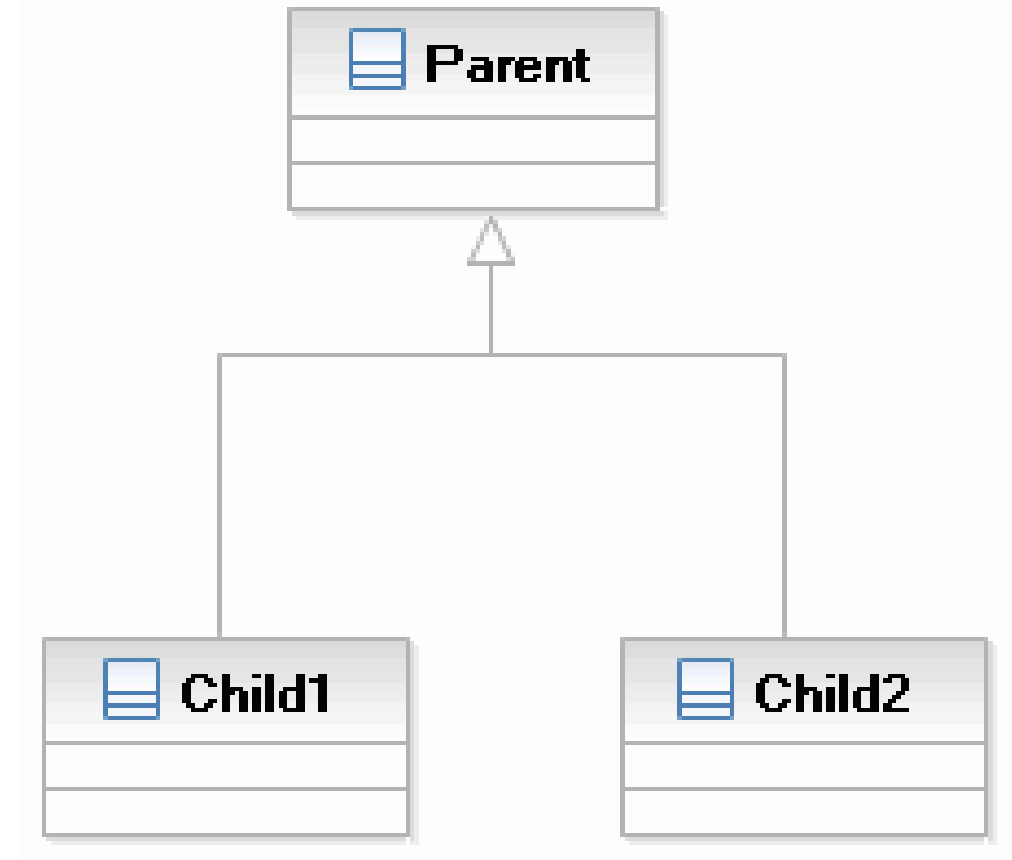
- A dependency means the relation between two or more classes in which a change in one may force changes in the other.
- However, it will always create a weaker relationship.
- Dependency indicates that one class depends on another.

In the following example, Student has a dependency on College

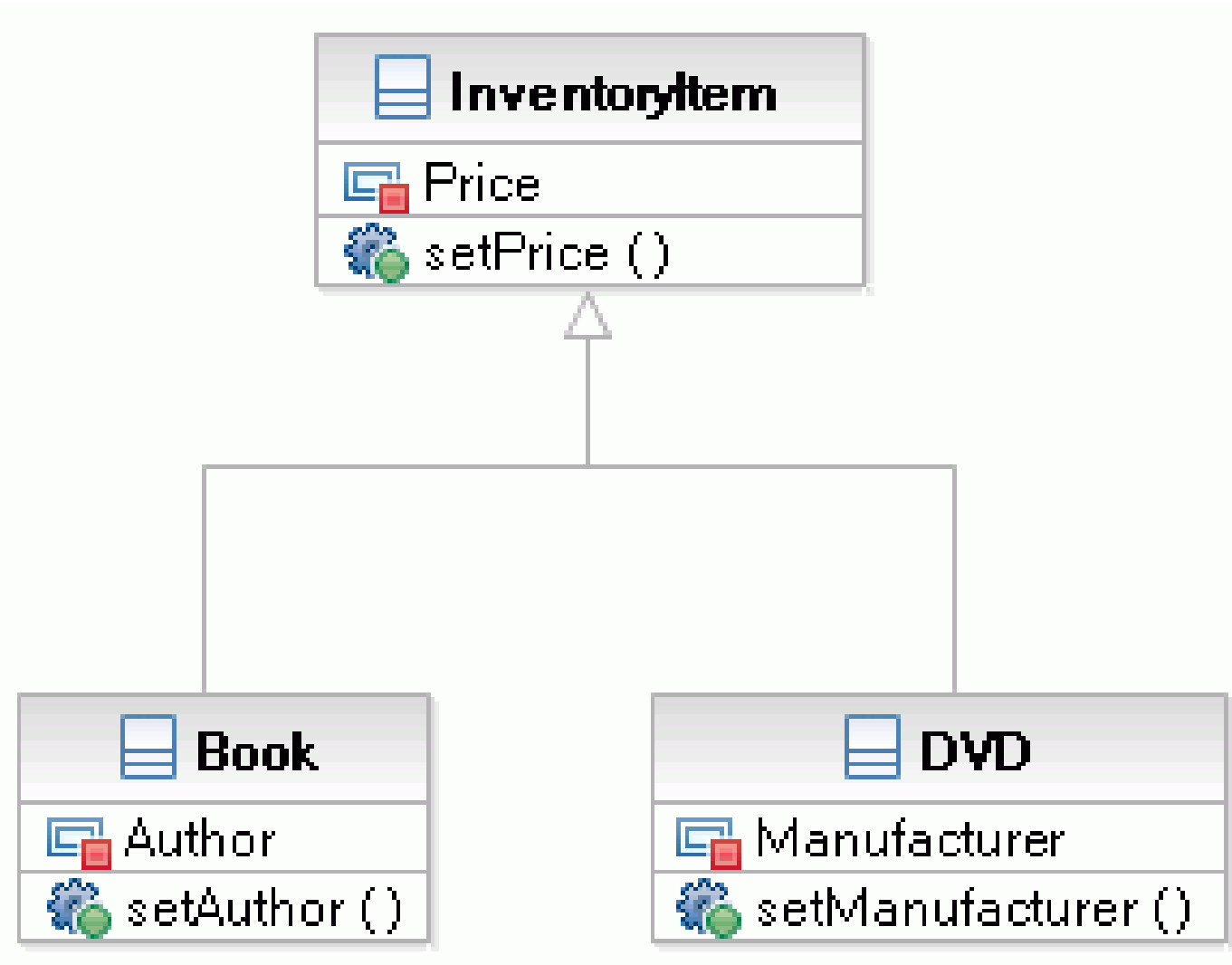


- In UML modeling, a generalization relationship is a relationship in which one model element (the child) is based on another model element (the parent).
- Generalization relationships are used in class, component, deployment, and use-case diagrams to indicate that the child receives all of the attributes, operations, and relationships that are defined in the parent.

- Generalization relationship can be used between actors or between use cases; however, it cannot be used between an actor and a use case.
- Generalization relationships do not have names.



# Example

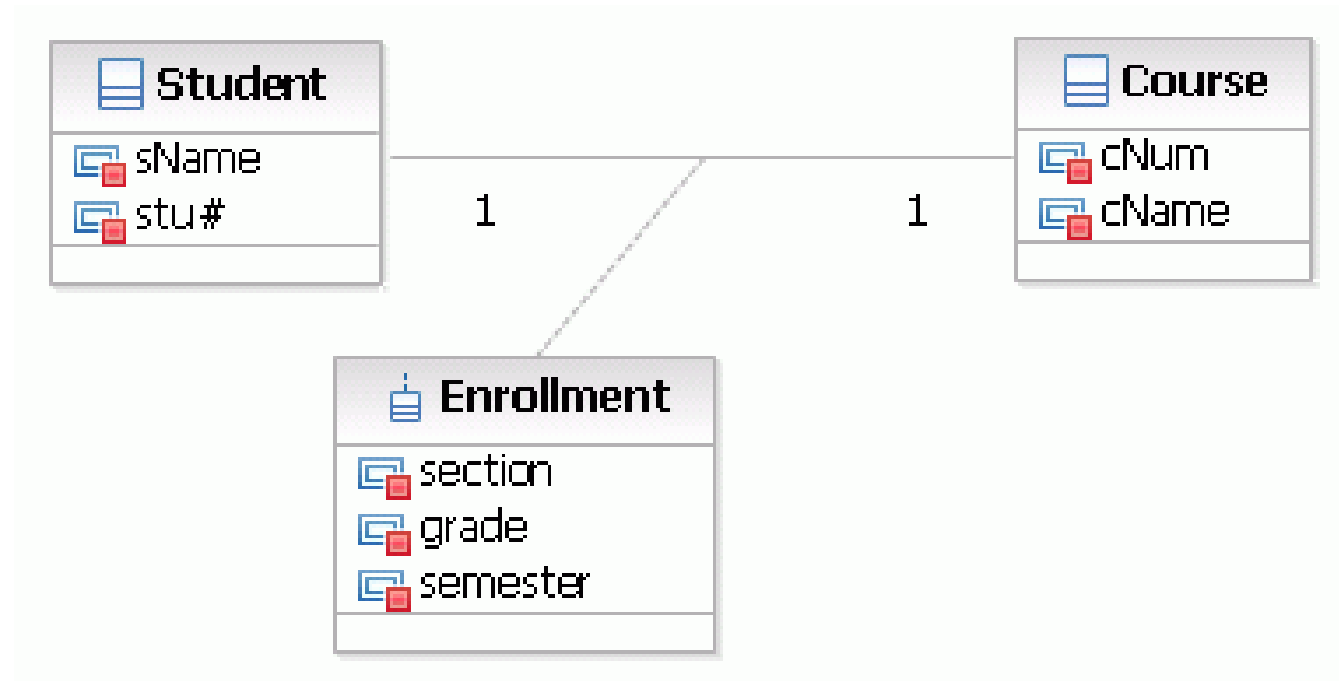


- The following figure illustrates an e-commerce application for a website that sells a variety of merchandise.
- The application has an InventoryItem class that is a parent class (also called a superclass).
- This class contains the attributes, such as Price, and operations, such as setPrice, that all pieces of merchandise use.

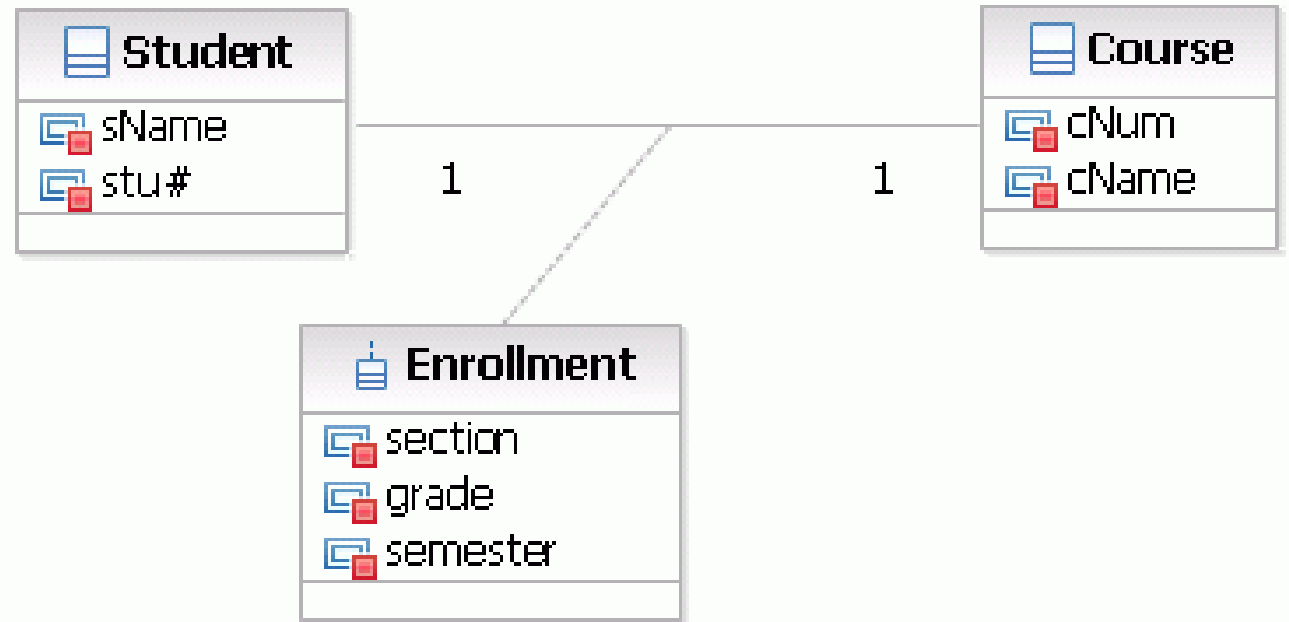


- After defining the parent class, a child class (also called a subclass) is created for each type of merchandise, such as books and DVDs.
- The book class uses the attributes and operations in the inventory class and then adds attributes such as author and operations such as setAuthor.
- A DVD class also uses the attributes and operations in the inventory class, but it adds attributes such as manufacturer and operations such as setManufacturer, which are different from those in the book class.

- For example, a class called Student represents a student and has an association with a class called Course, which represents an educational course. The Student class can enroll in a course.

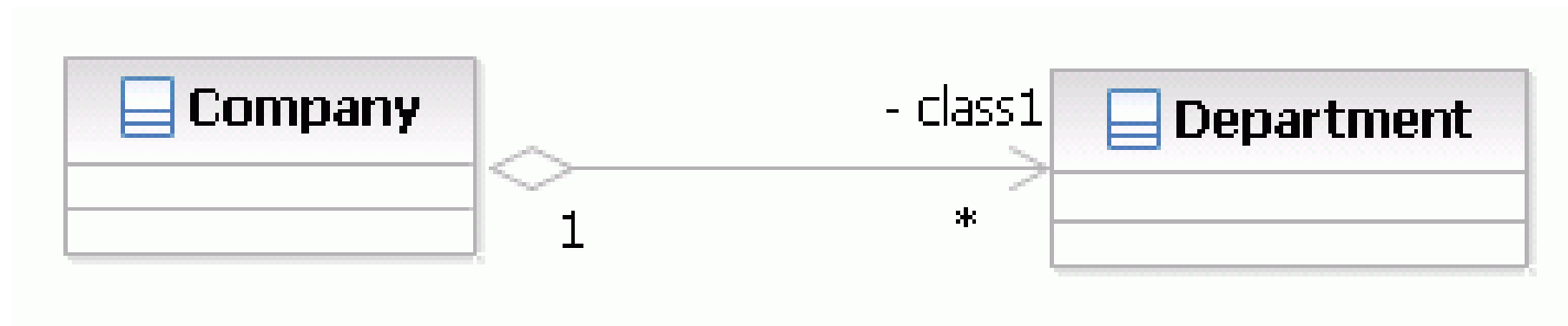


- An association class called Enrollment further defines the relationship between the Student and Course classes by providing section, grade, and semester information related to the association relationship.



- An aggregation describes a group of objects and how you interact with them.
- For example, a Department class can have an aggregation relationship with a Company class, which indicates that the department is part of the company.

- As the following figure illustrates, an aggregation association appears as a solid line with an unfilled diamond at the association end, which is connected to the classifier that represents the aggregate.
- Aggregation relationships do not have to be unidirectional.



- **Classes and Relationships**
- **Common Mechanisms**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Classes and Relationships</b></li><li>• <b>Common Mechanisms</b></li></ul>	<ul style="list-style-type: none"><li>• Demonstrate the Conceptual model of UML architecture, common mechanisms</li></ul>	1. Understand the concepts of classes, relationships and common mechanisms.

# Outcome achieved



## Name of the topic:

<b>Students will be able to:</b>	
1	Describe the importance of various common mechanisms used in classes and their associated relationships.



- A composition association relationship represents a whole–part relationship and is a form of aggregation.
- A composition association relationship specifies that the life time of the part classifier is dependent on the life time of the whole classifier.

- For example, a composition association relationship connects a Student class with a Schedule class, which means that if you remove the student, the schedule is also removed.



- As the following figure illustrates, a composition association relationship appears as a solid line with a filled diamond at the association end, which is connected to the whole, or composite, classifier.



## Aggregation

Aggregation indicates a relationship where the child can exist separately from their parent class.

**Example:** Automobile (Parent) and Car (Child). So, If you delete the Automobile, the child Car still exist.

## Composition

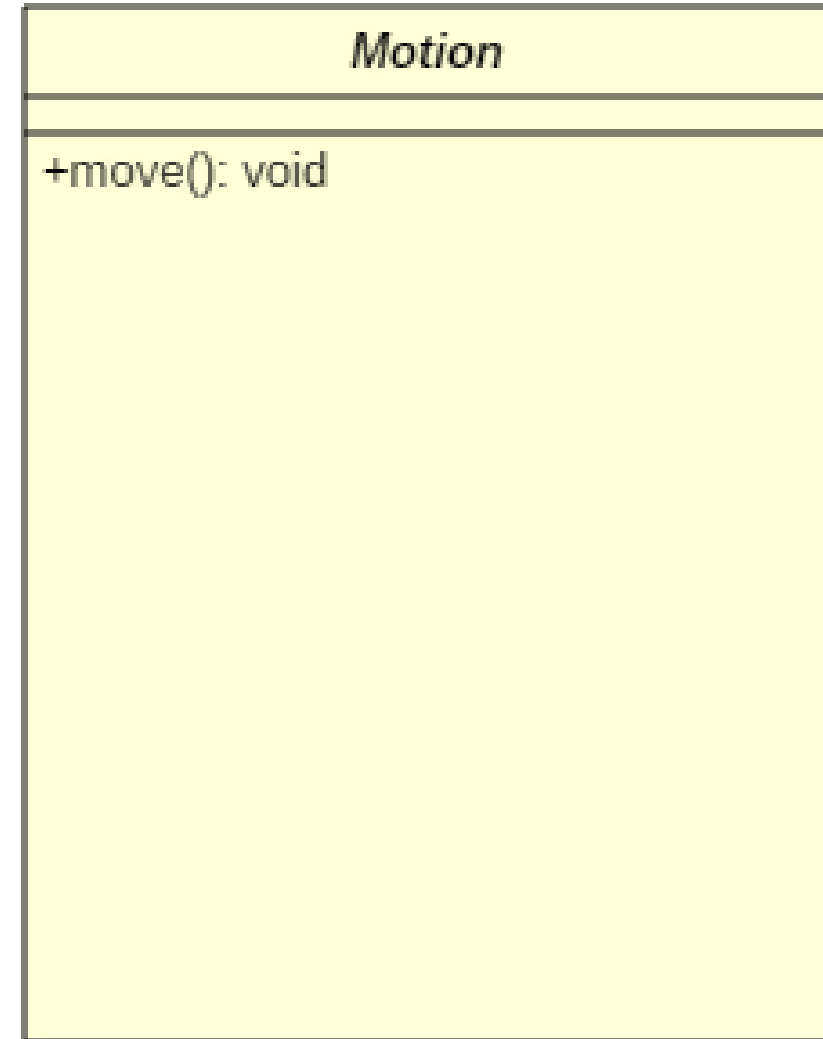
Composition display relationship where the child will never exist independent of the parent.

**Example:** House (parent) and Room (child). Rooms will never separate into a House.

- It is a class with an operation prototype, but not the implementation.
- An abstract is useful for identifying the functionalities across the classes.
- Suppose we have an abstract class called as a motion with a method or an operation declared inside of it. The method declared inside the abstract class is called a **move ()**.

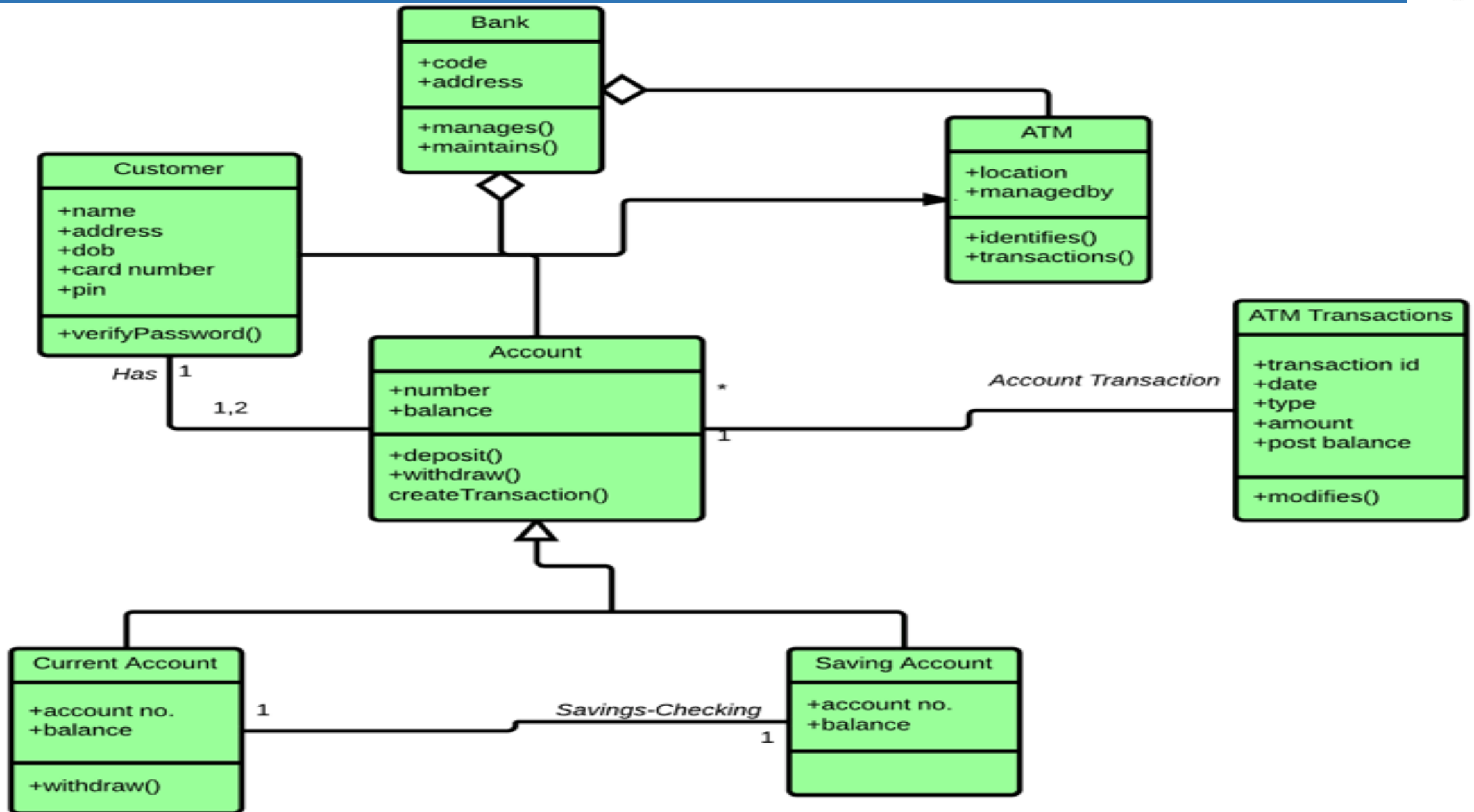
- This abstract class method can be used by any object such as a car, an animal, robot, etc. for changing the current position.
- It is efficient to use this abstract class method with an object because no implementation is provided for the given function.
- We can use it in any way for multiple objects.
- In UML, the abstract class has the same notation as that of the class. The only difference between a class and an abstract class is that the class name is strictly written in an italic font.

- The abstract class notation, there is the only a single abstract method which can be used by multiple objects of classes.



- Creating a class diagram is a straightforward process. It does not involve many technicalities. Here, is an example:
- ATMs system is very simple as customers need to press some buttons to receive cash.
- However, there are multiple security layers that any ATM system needs to pass.
- This helps to prevent fraud and provide cash or need details to banking customers.





- UML is the standard language for specifying, designing, and visualizing the artifacts of software systems
- A class is a blueprint for an object
- A class diagram describes the types of objects in the system and the different kinds of relationships which exist among them
- It allows analysis and design of the static view of a software application

- Class diagrams are most important UML diagrams used for software application development
- Essential elements of UML class diagram are
  - 1) Class
  - 2) Attributes
  - 3) Relationships
- Class Diagram provides an overview of how the application is structured before studying the actual code. It certainly reduces the maintenance time
- The class diagram is useful to map object-oriented programming languages like Java, C++, Ruby, Python, etc.



*Thank you*

- **ADVANCED BEHAVIORAL MODELING**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>ADVANCED BEHAVIORAL MODELING</b></li></ul>	<ul style="list-style-type: none"><li>• Understand the concepts of Advanced classes</li></ul>	<ol style="list-style-type: none"><li>1. Understand the concepts of Advanced classes</li></ol>

# Outcome achieved



## Name of the topic:

### Students will be able to:

1

Understand the concepts of Advanced classes include Classifiers, Visibility, Scope, Multiplicity, Attributes, Operations, Common, Modeling Techniques

## **UNIT -II**

# **ADVANCED BEHAVIORAL MODELING**



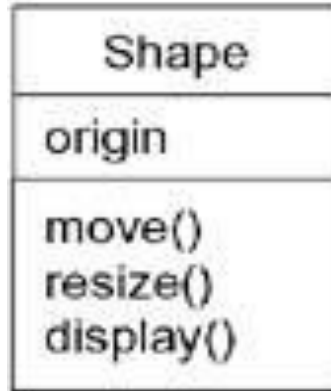
- Advanced classes, advanced relationships, interfaces, types and roles, packages, terms, concepts;
- Class and Object Diagrams: Terms, concepts, common modeling techniques for class and object diagrams.

- The fundamental building block in a object-oriented system is an object or class.
- It is only one of the general building blocks in UML, called as classifiers.
- A classifier describes a set of instances that have common behavioral and structural features (operations and attributes, respectively).
- A classifier is a mechanism which describes structural and behavioral features in a system.

- Other classifiers in UML are: interfaces, datatypes, signals, components, nodes, use cases and subsystems.
- Class is the frequently used classifier.
- Every classifier represents structural aspects in terms of *properties* and behavioral aspects in terms of *operations*.
- Beyond these basic features, there are several advanced features like multiplicity, visibility, signatures, polymorphism and others.

Classifier	Description
Class	A class is a blueprint or a template for its objects
Interface	An interface is a collection of operations that must be realized by another element
Datatype	A type whose values have no identity
Signal	An asynchronous event communicated between objects
Component	Physical replaceable part of the system
Node	A physical existing at runtime which represents a computational resource
Use case	Collection of actions that a role performs on the system or the system does for a user.
Sub system	A grouping of elements which specify the behavior of a part of the system

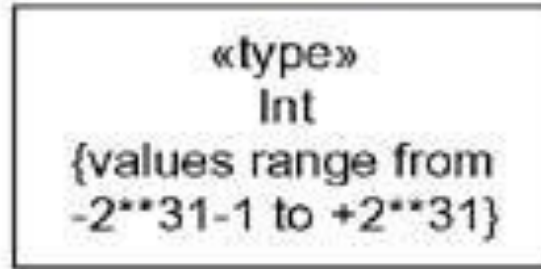
class



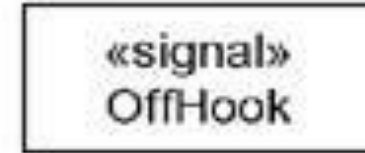
interface



datatype



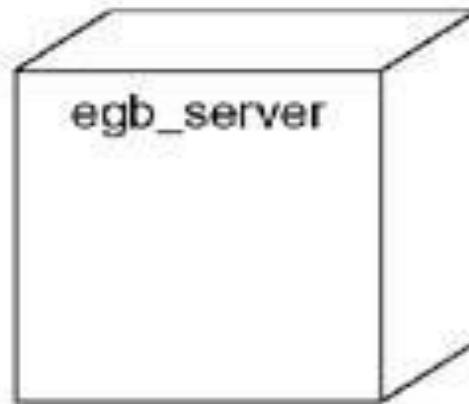
signal



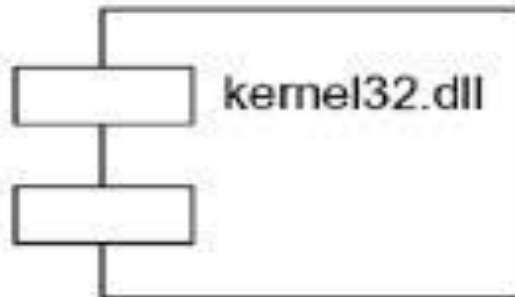
use case



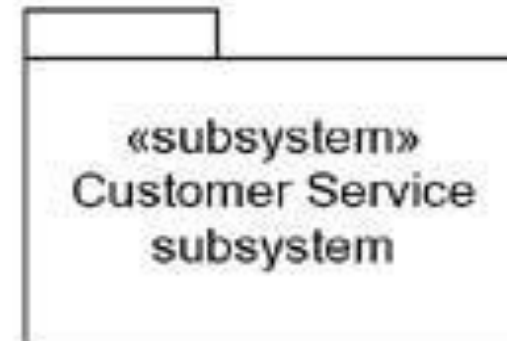
node



component



subsystem



**Visibility** - UML, you can specify any of three levels of visibility.

**1. public**

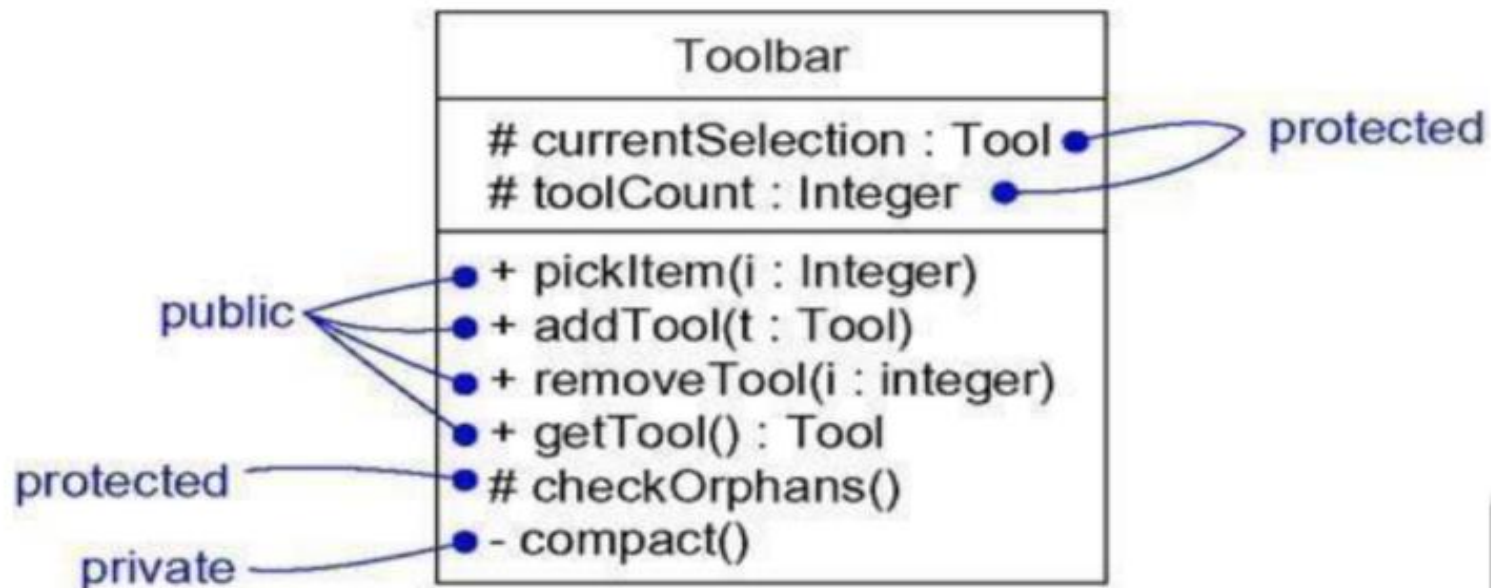
Any outside classifier with visibility to the given classifier can use the feature specified by prepending the symbol +.

**2. protected**

Any descendant of the classifier can use the feature; specified by prepending the symbol #.

**3. private**

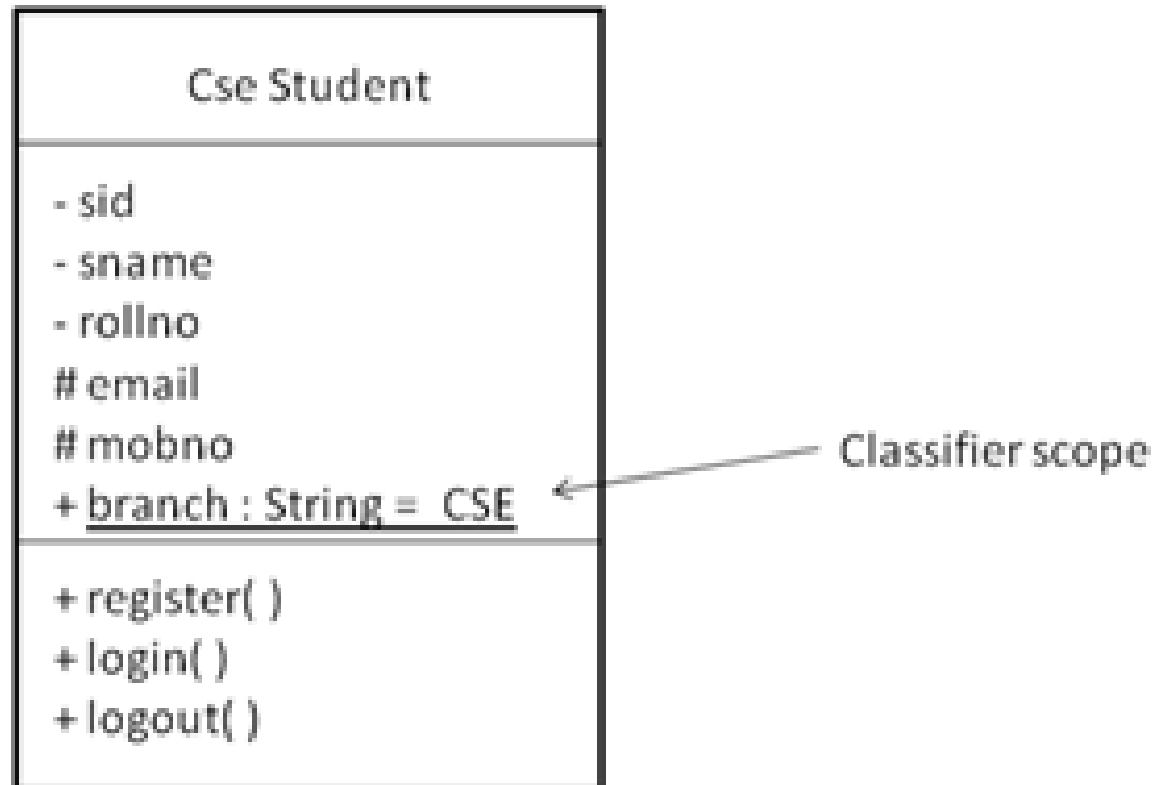
Only the classifier itself can use the feature; specified by prepending the symbol -.



- Another feature that can be applied to the classifier's attributes and operations is the scope.
- The scope of an attribute or an operation denotes whether they have their existence in all the instances of the classifier or only one copy is available and is shared across all the instances of the classifier.
- The scope specifiers in UML are:

Scope	Description
instance	Each instance of the classifier contains a value of the feature
classifier	One copy of the feature's value is shared by all the instances of a classifier

**Example:**





- Multiplicity can be set for attributes, operations and associations in a UML class diagram, and for associations in a use case diagram.
- The multiplicity is an indication of how many objects may participate in the given relationship, or the allowable number of instances of the element.

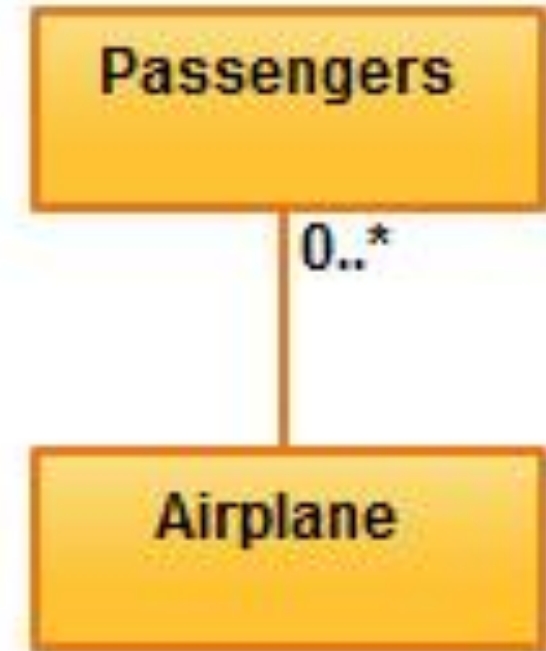
- When modeling classes it is reasonable to assume that a class can have any number of instances.
- In some cases a class might not have any instances at all.
- Such a class with no instances is called a utility class which contains only attributes and operations with the classifier scope.
- A class with exactly one instance is called a singleton class.

- Similarly other classes might have any number of instances which is the default in UML.
- The specification of how many instances a class can have is known as multiplicity.
- In UML, we can mention the multiplicity as an expression in the top right hand corner of the class icon.
- Attributes too can have multiplicity. The multiplicity of an attribute is represented in brackets after the attribute name.

# Example

Car	1
engine [1] wheel [4] door [4] horn	
move()	

- For example, one fleet may include multiple airplanes, while one commercial airplane may contain zero to many passengers. The notation  $0..*$  in the diagram means “zero to many”.



<b>Multiplicity</b>	<b>Option</b>	<b>Cardinality</b>
0..0	0	Collection must be empty
0..1		No instances or one instance
1..1	1	Exactly one instance
0..*	*	Zero or more instances
1..*		At least one instance
5..5	5	Exactly 5 instances
m..n		At least m but no more than n instances

- While modeling classes, it is reasonable to write only the attribute name.
- Along with the name, you can also specify multiplicity, visibility and scope.
- Apart from these features, we can also specify the type, initial value and changeability of each attribute.

- The syntax of an attribute in UML is:

***[visibility] name [multiplicity] [: type] [=initial value]  
[{property string}]***



# Examples:

origin	Name only
+ origin	Name and visibility
origin : Point	Name and type
head : *Item	Name and complex type
name [0..1] : String	Name, multiplicity and type
origin : Point = {0,0}	Name, type and initial value
id : Integer {frozen}	Name and property

- The three predefined properties that can be used with attributes in UML are:

changeable	No restrictions on modifying the attributes value
addOnly	For attributes with multiplicity greater than one, values may be added but not removed or altered once it is created
frozen	Once created the value of an attribute cannot be changed

## Student

- sid {frozen}

- sname

- rollno {frozen}

# email

+ marks {addOnly}

+ register( )

+ login( )

+ logout( )

- Along with the name we can also mention the visibility and scope.
- Apart from these basic details, we can also specify the return type, parameters, concurrency semantics and other features.
- The syntax of an operation in UML is:

***[visibility] name [(parameters-list)] [: return-type]  
[{property-string}]***

<code>display</code>	Name only
<code>+ display</code>	Name and visibility
<code>set(n : Name, s : String)</code>	Name and parameters
<code>getID() : Integer</code>	Name and return type
<code>restart() {guarded}</code>	Name and property

- The signature of parameter in the parameters list of an operation is as follows:

***[direction] name :type [=initial value]***

- Direction may be anyone of the following values:

<code>in</code>	An input parameter, may not be modified
<code>out</code>	An output parameter, may be modified
<code>inout</code>	An input parameter, may be modified

- The operation name, parameters list and its return type is collectively known as the signature of the operation.

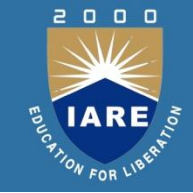
- There are four predefined properties (stereotypes) that can be applied to operations, in UML. They are:

isQuery	Execution of the operation does not change the state of the system.
sequential	Denotes that the operation is suitable for sequential access only.
guarded	Indicates that the operation guarantees integrity in case of concurrent access. Concurrent calls will be performed in a sequential manner.
concurrent	Denotes that the operation can be called concurrently and the integrity of the object will be preserved. Operation executes on the object concurrently.

- **Common Modeling Techniques**
- **Advanced Relationships**



# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Common Modeling Techniques</b></li><li>• <b>Advanced Relationships</b></li></ul>	<ul style="list-style-type: none"><li>• Describe Modeling the semantic of a class</li><li>• Understand the concepts of Advanced Relationships</li></ul>	<ol style="list-style-type: none"><li>1. Describe Modeling the semantic of a class</li><li>2. Understand the representation of different Advanced Relationships concepts.</li></ol>

# Outcome achieved



## Name of the topic:

<b>Students will be able to:</b>	
1	Describe Common Modeling Techniques
2	Understand the concepts of Advanced Relationships & graphics representations

## Modeling the semantic of a class

- To model the semantics of a class:
  1. Specify the responsibilities of the class in a note and attach it to the class with a dependency relationship.
  2. Specify the semantics of a class as a whole in a note stereotyped with "semantics".
  3. Specify the body of a method using structured text or a programming language in a note and attach it to the class or operation using a dependency relationship.

4. Specify the pre-conditions and post-conditions for operations using structured text in a note.
5. Specify a state machine for a class which represents different states the object undergoes.
6. Specify a collaboration that represents a class.

- The things in diagrams are connected with one another through relationships.
- So, relationships are the connections between things.
- In UML, the four important relationships are *dependency*, *generalization*, *association* and *realization*.
- Each type of relationship has its own graphical representation.

- ***Contents***

1. Dependency
2. Generalization
3. Association
4. Realization
5. Common Modeling Techniques

# 1. Dependency

- The dependency relationship is also known as using relationship i.e., if the specification of one thing changes then it will automatically affect the behavior of another thing that uses it.
- A dependency relationship is graphically represented as a dashed arrow.



- In general, a plain dependency relationship is reasonable for representing the using relationship between two things.
- But, if the user needs to specify extra information like the nature of the dependency relationship, he/she can use the predefined stereotypes that can be applied to dependency relationship.
- There are about 17 such stereotypes and are organized into 6 groups based on which things are participating in the dependency relationship.

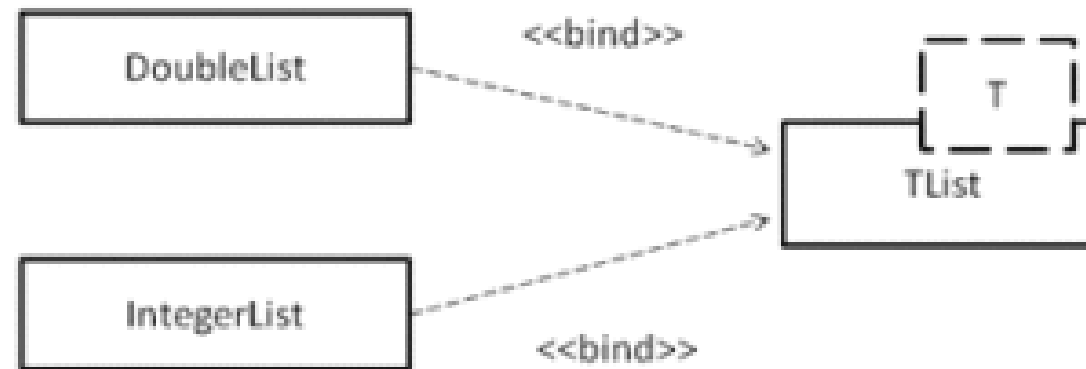


- First group consists of eight stereotypes that apply to dependency relationship between classes. They are as follows:

## 1. Bind:

- Specifies that the source instantiates (creates object) the target template using the given actual parameters.

- Ex:



## 2. Derive:

- Specifies that the source may be computed from the target.



### 3. friend:

- Specifies that the source is accessible by the target regardless of the visibility of the source element



## 4. instanceof:

- Specifies that the source object is an instance of the target classifier



## 5. instantiate:

- Specifies that the source creates instances of the target



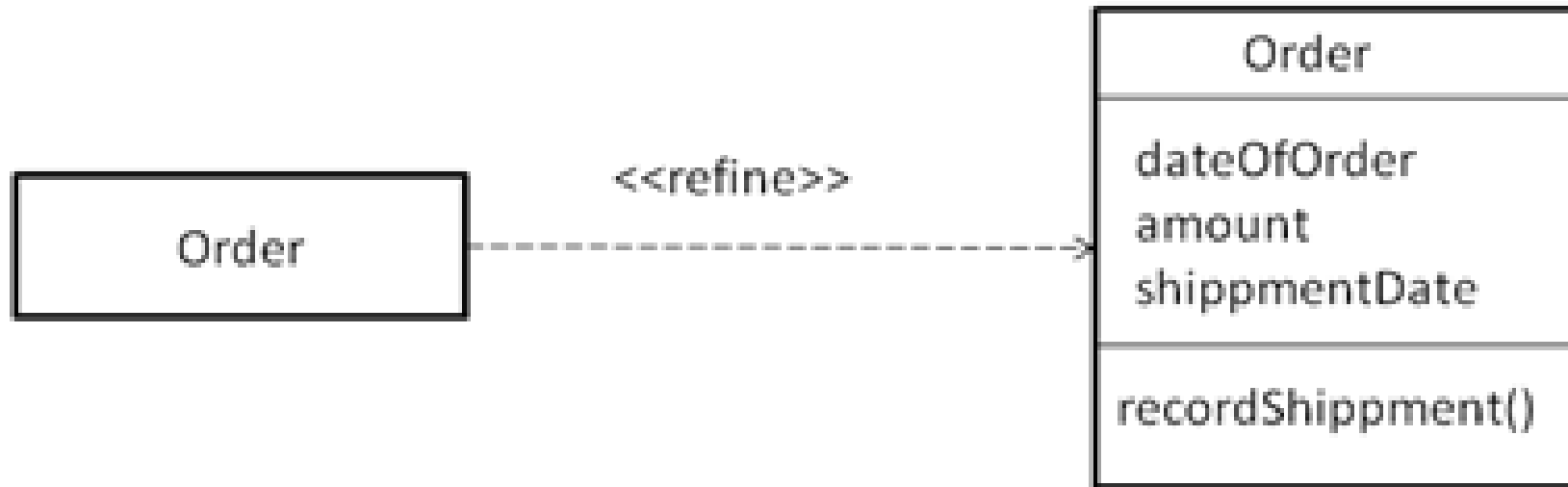
## 6. powertype:

- Specifies that the target is a powertype of the source.
- A powertype is a classifier whose objects are the children of a given parent.



## 7. refine:

- Specifies that the target is at a lower level of abstraction than the source



## 8. use:

- Specifies that the source element depends on the target for its functionality.

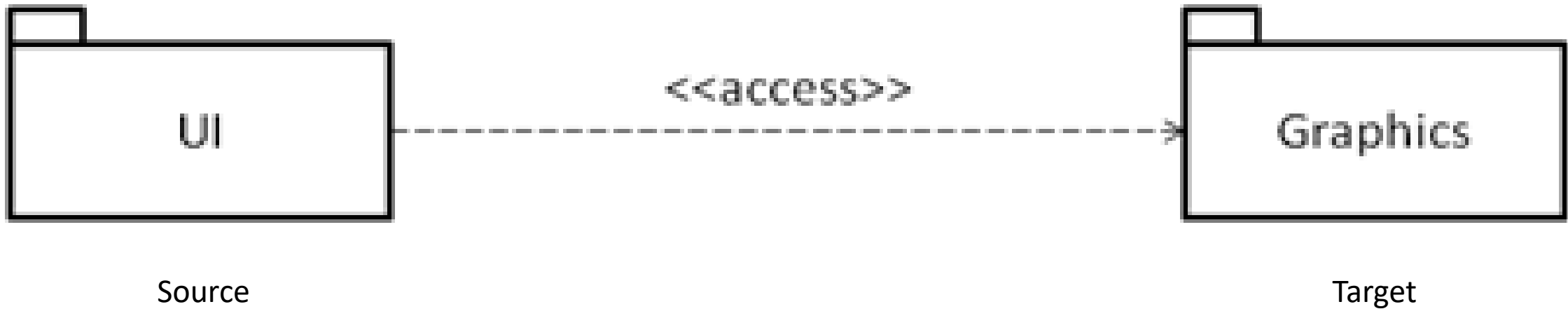




- There are two stereotypes that apply to dependencies between packages. They are:
  1. Access
  2. Import

## 1. Access:

- Specifies that the source package has the right to access the elements of the target package.



## 2. Import:

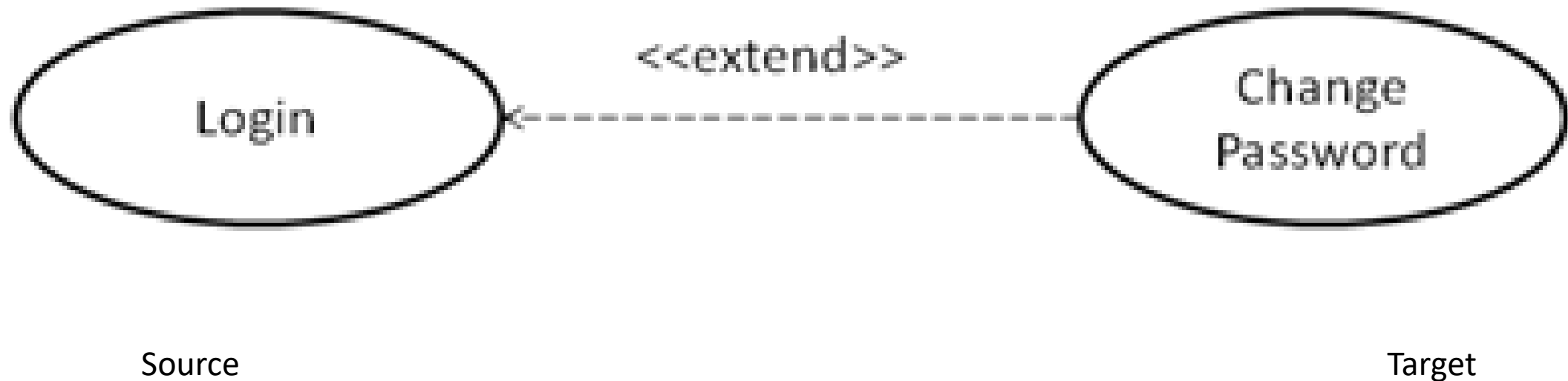
- Specifies that the public elements of the target package enter the namespace of the source package as if they were declared in the source



- Two stereotypes apply to dependency relationship among use case. They are:
  1. Extend
  2. Include

## 1. Extend:

- Specified that the target use case extends the functionality of the source use case



## 2. include:

- Specifies that the source use case incorporates the behaviour of the target use case to function as a whole



- Three stereotypes apply to interactions among objects. They are:

1. Become
2. Call
3. Copy

# 1. Become:

- Specifies that the source object becomes the target object at some point in time





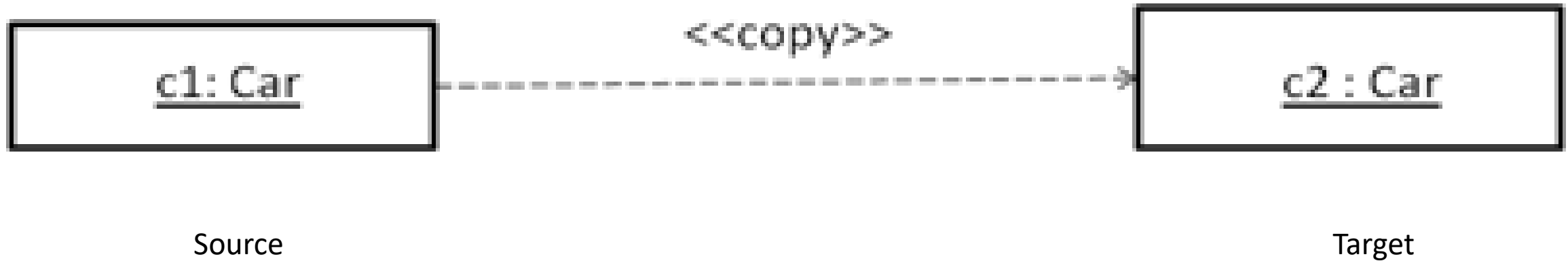
## 2. Call:

- Specifies that an operation in source calls another operation of the target



### 3. Copy:

- Specifies that the target is an exact, but independent copy of the source.



- One stereotype applies to dependencies in the context of state machines: ***send***

## **1. Send:**

- Specifies that the source, an operation whose target is a signal, sends the target signal

- One stereotype applies to dependencies in the context of subsystem: ***trace***

## 1. Trace:

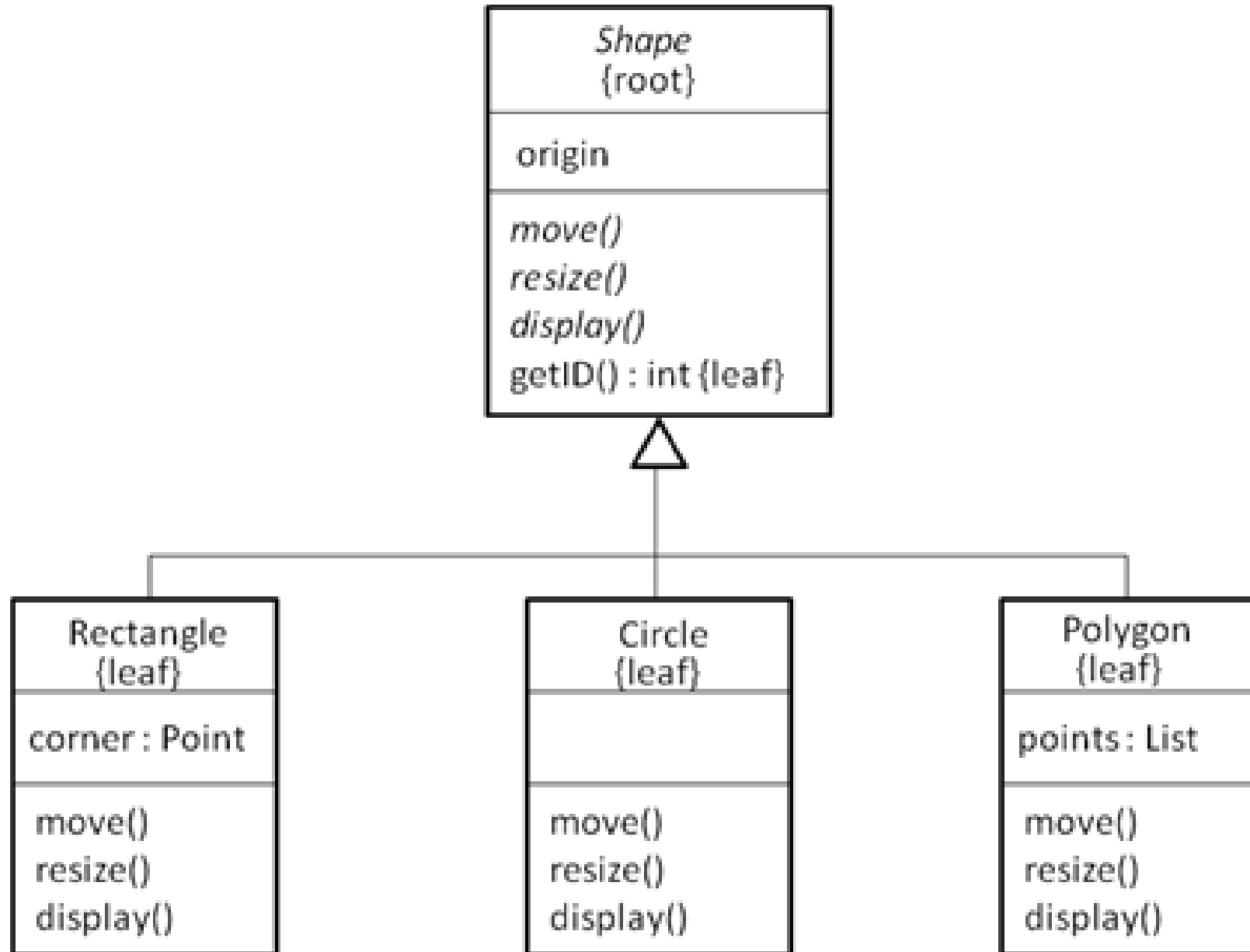
- Specifies that the source element has a conceptual connection to the target which individually belong to different models



## 2. Generalization



- A generalization relationship represents generalization-specialization relationship between classes.
- The class with the general structure and behavior is known as the parent or superclass and the class with specific structure and behavior is known as the child or subclass.
- Consider the below class hierarchy:



- Shape class is the parent or super class and the remaining three classes namely Rectangle, Circle and Polygon are the child or subclasses of the Shape class.
- A subclass in the generalization relationship automatically inherits the state and behavior of the superclass.
- The generalization relationship is also known as the “is-a” relationship.

- If a class has only one parent, such inheritance is known as single inheritance and if a class has one or more parents, such inheritance is known as multiple inheritance.
- To represent extra semantics in a generalization relationship, UML provides one stereotype and four constraints.
- The stereotype on generalization relationship is ***implementation***.



## 1. Implementation:

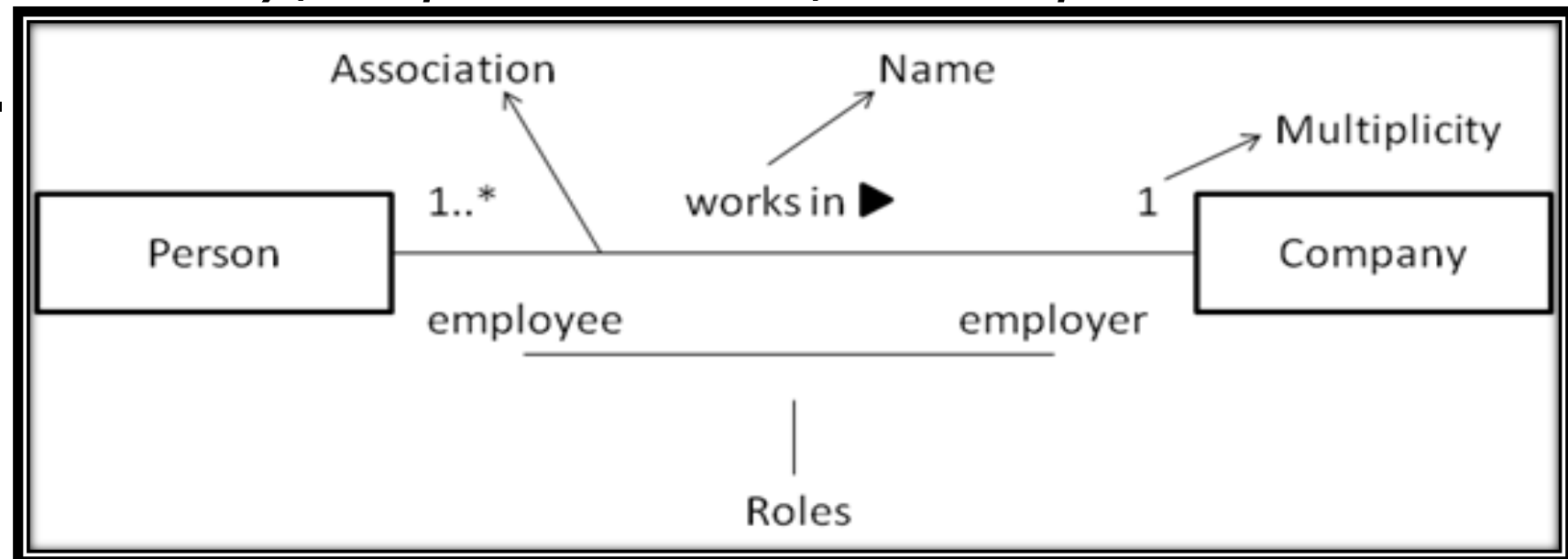
- Specifies that the child inherits the implementation of the parent but does not support its interfaces, there by violating the principles of substitutability.
- There are four standard constraints that apply to the generalization relationship are *complete*, *incomplete*, *disjoint*, *overlapping*.

# 3. Association



- Association is a structural relationship which denotes a connection between two or more things.
- The association relationship can represent either physical or logical connections between things.
- The graphical representation of the association relationship is a solid line.

- The four basic adornments for an association relationship are: **name**, **role** at each end of the association, **multiplicity** at each end of the association and **aggregation**.
- Over these basic features, there are other advanced features like: *navigation*, *visibility*, *qualification*, *composition* and *association classes*.



- **Navigation**

- Given an association between two things, we can navigate from one thing to another and vice versa.
- By default the navigation of an association is bidirectional.
- In some cases we may need to navigate in only direction.



- **Visibility**

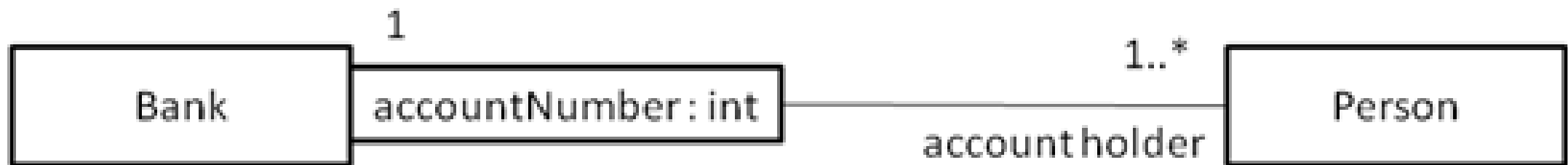
- Given an association, we can navigate from one object to another.
- However, in some situations we may want to limit the visibility (access) of an object to the objects outside the association relationship.
- To limit the visibility of an object, we can use the visibility specifiers: ***public (+)***, ***private (-)*** and ***protected (#)***.

- ***For example,*** a usergroup object can access the user object and an user object can access the password.
- If, private visibility is specified for the password object, the usergroup object cannot access the password of the user object.

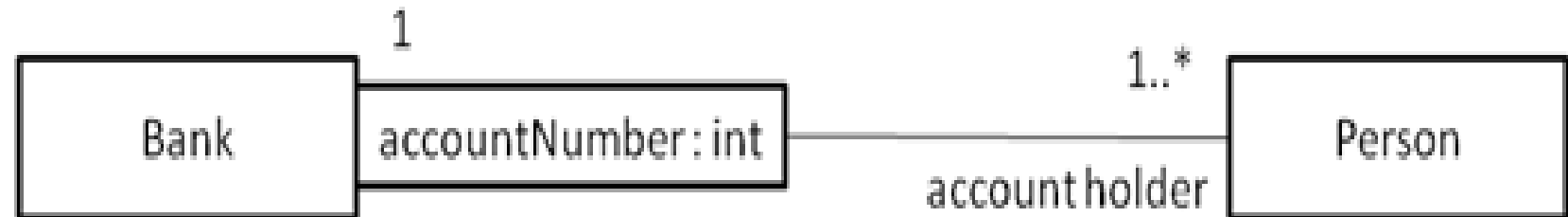


- **Qualification**

- In binary associations, where two classes are connected together, if one object of a class has to identify a set of instances of another class, we can use qualifiers.
- A qualifier is a set of attributes of an association which is used to identify a set of instances of a class.



- Each qualifier is represented with a name and type in a rectangle box at the qualified end of the association relationship.
- For example, a bank object is able to recognize the person (account holder) based on the account number.
- So, account number is the qualifier which is used to identify an account holder.





- **Composition**

- The composition is a flavor of association relationship.
- Composition as well as aggregation relationships represent whole-part relationships, in which one thing is a part of the other thing.
- There is a simple difference between the association and composition relationships.

- In ***composition***, the lifetime of the part is dependent of the lifetime of the whole thing.
- Where as in ***aggregation***, the lifetime of the part is independent of the whole thing.
- Composition is graphically represented by adorning the association relationship with a filled diamond head near the whole end.



- **Advanced Relationships**
- **Interfaces, types and roles**
- **Packages**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Advanced Relationships</b></li><li>• <b>Interfaces, types and roles</b></li><li>• <b>Packages</b></li></ul>	<ul style="list-style-type: none"><li>• Understand the concepts of Advanced Relationships</li><li>• Interfaces, types and roles</li><li>• Identify purpose of Packages</li></ul>	<ol style="list-style-type: none"><li>1. Understand the representation of different Advanced Relationships concepts.</li><li>2. Understand usage of interface, types and roles</li><li>3. Identify the importance of Package</li></ol>

# Outcome achieved



## Name of the topic:

<b>Students will be able to:</b>	
1	Understand the concepts of Advanced Relationships & graphics representations
2	Understand usage of interface, types and roles
3	Identify the importance of Package

- ***Contents***

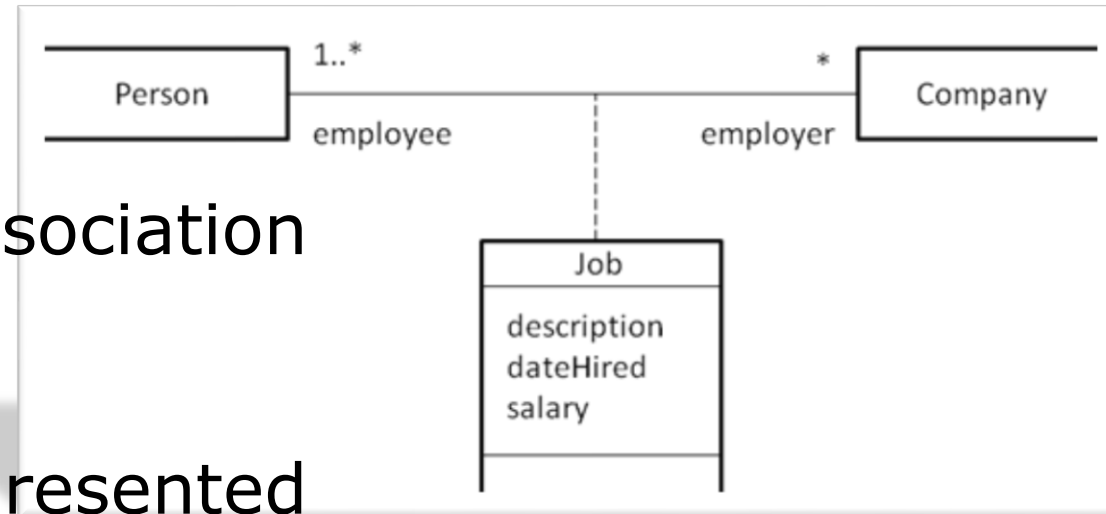
1. Dependency
2. Generalization
3. Association
4. Realization
5. Common Modeling Techniques

## • Association Class

- Sometimes in an association relationship, the association might have attributes or properties like a class does.

- In such cases, it is modeled as an association class.

- An association class in UML is represented with a class icon attached to the association with a dashed line.



- Apart from the above advanced features, UML provides six constraints that can be applied to an association. They are:

implicit	Specifies that the relationship is not physical, but rather conceptual
ordered	Specifies that the objects at one end of the association are in a certain order
changeable	Links between objects can be added, removed
addOnly	New links may be added from an object on the opposite side of the association
frozen	After adding a link it might not be modified later
xor	Indicates that among a set of associations, only one can be applied for each object



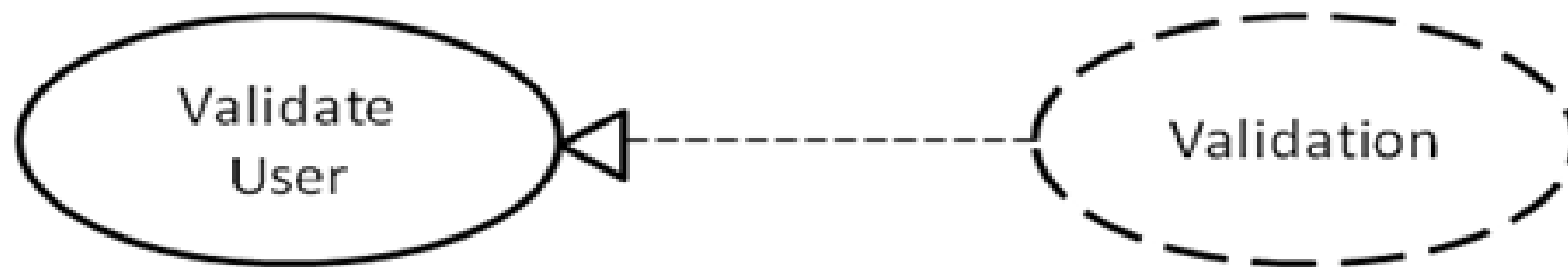
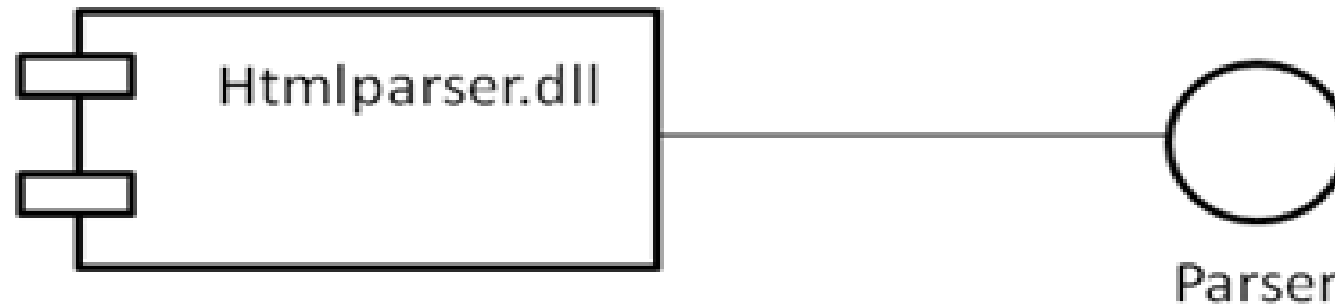
- Association is a relationship between classifiers which is used to show that instances of classifiers could be either linked to each other or combined logically or physically into some aggregation.
- UML specification categorizes association as semantic relationship.

## Example: Unary Association



# 4. Realization

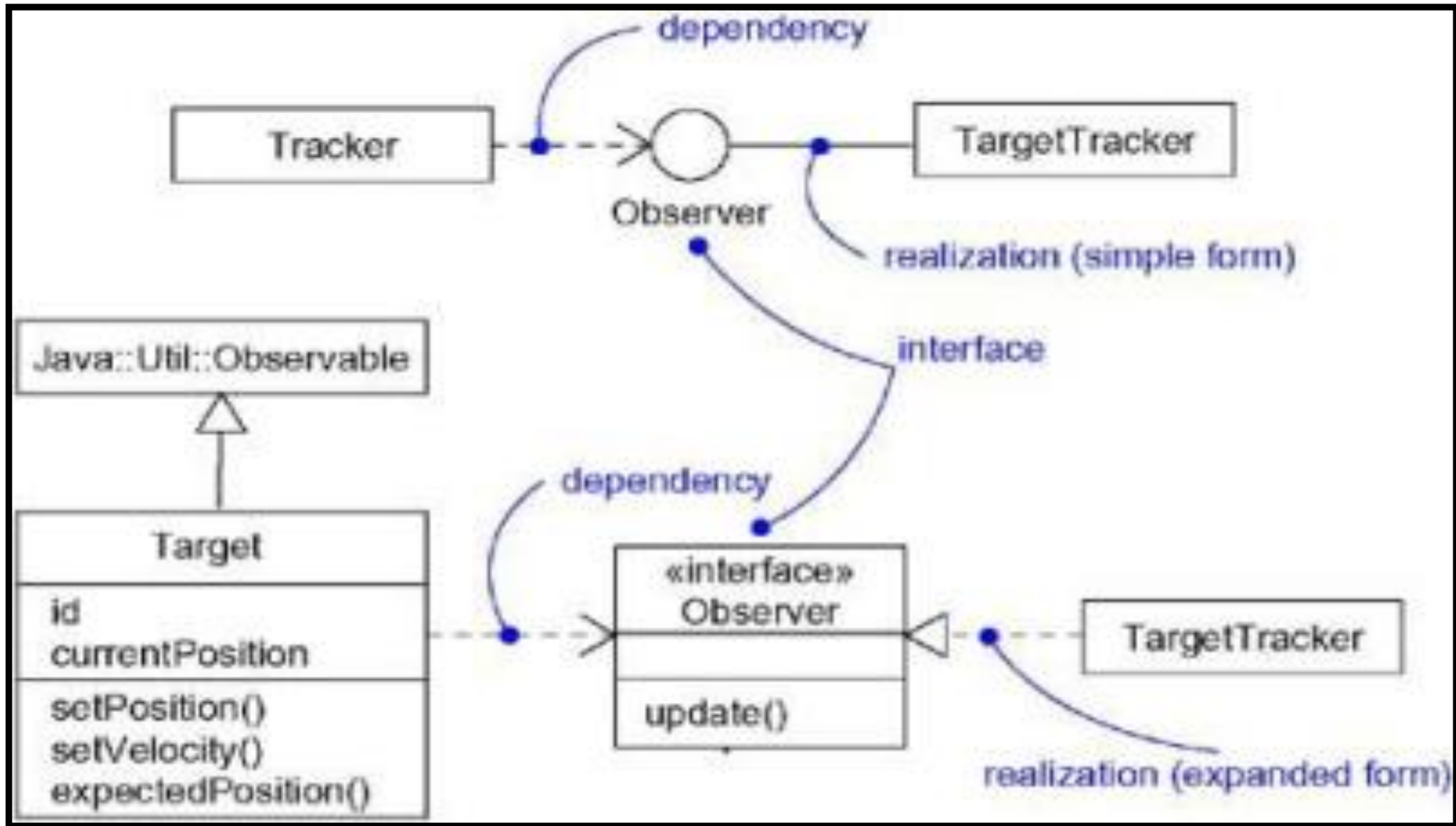
- Realization is a semantic relationship between classifiers, where one classifier provides the specification which is implemented by the other classifier.
- Realization can exist between an interface and class, interface and a component and between a use case and collaboration.
- Realization is graphically represented as dashed line with a hollow arrow head pointing towards the classifier which provides the specification.



- **Modeling Webs of Relationships:** To model webs of relationships,
  1. Apply use cases and scenarios to find the relationships between the abstractions in the system.
  2. Start by modeling the structural relationships (associations) between the things. These specify the structure of the system.
  3. Then, model the generalization-specialization relationships.
  4. Finally, after modeling the remaining relationships go for dependency relationships.
  5. After representing all the relationships, transform their basic representation by applying the advanced features to your intent.

- **Interface**

- An interface is a collection of operations that are used to specify a service of a class or a component.
- Graphically, an interface is rendered(represented) as a circle; in its expanded form, an interface may be rendered as a stereotyped class as shown in below figure 1.
- An interface name must be unique within its enclosing package.

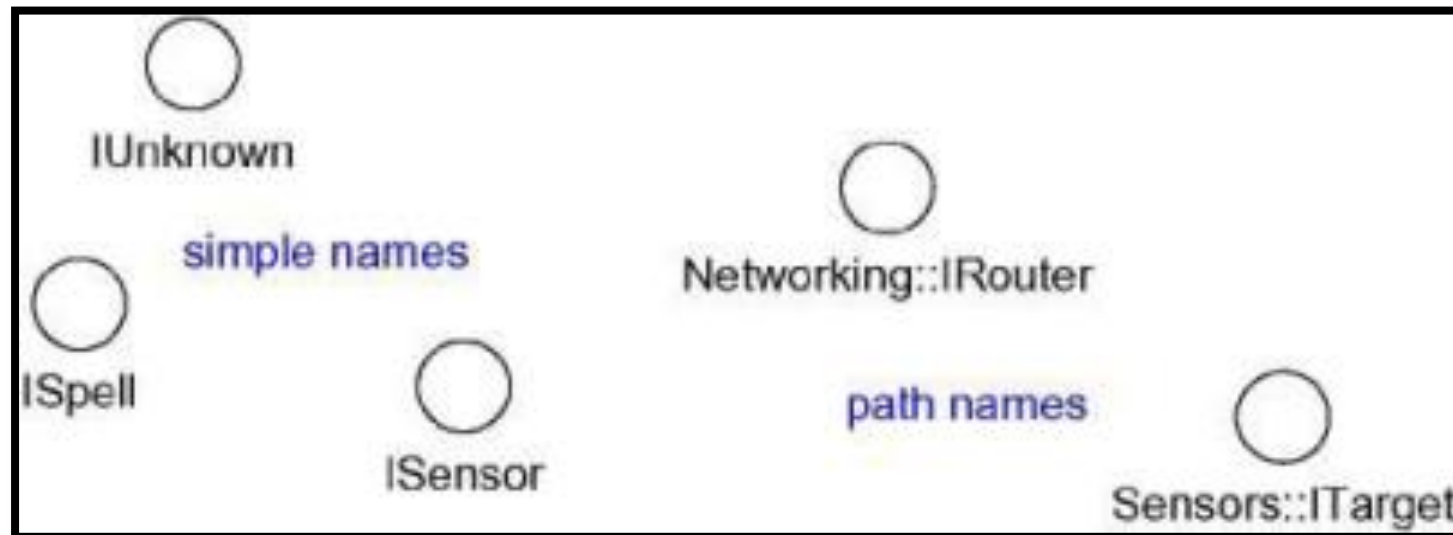


**Figure: 1**

- There are two naming mechanism;
  - ✓ a simple *name* (only name of the interface),
  - ✓ a path name is the interface name prefixed by the name of the package in which that interface lives represented in Figure: 2.
- To distinguish an interface from a class, prepend an 'I' to every interface name.



- Operations in an interface may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.
- Interface don't have attributes.



**Figure: 2**

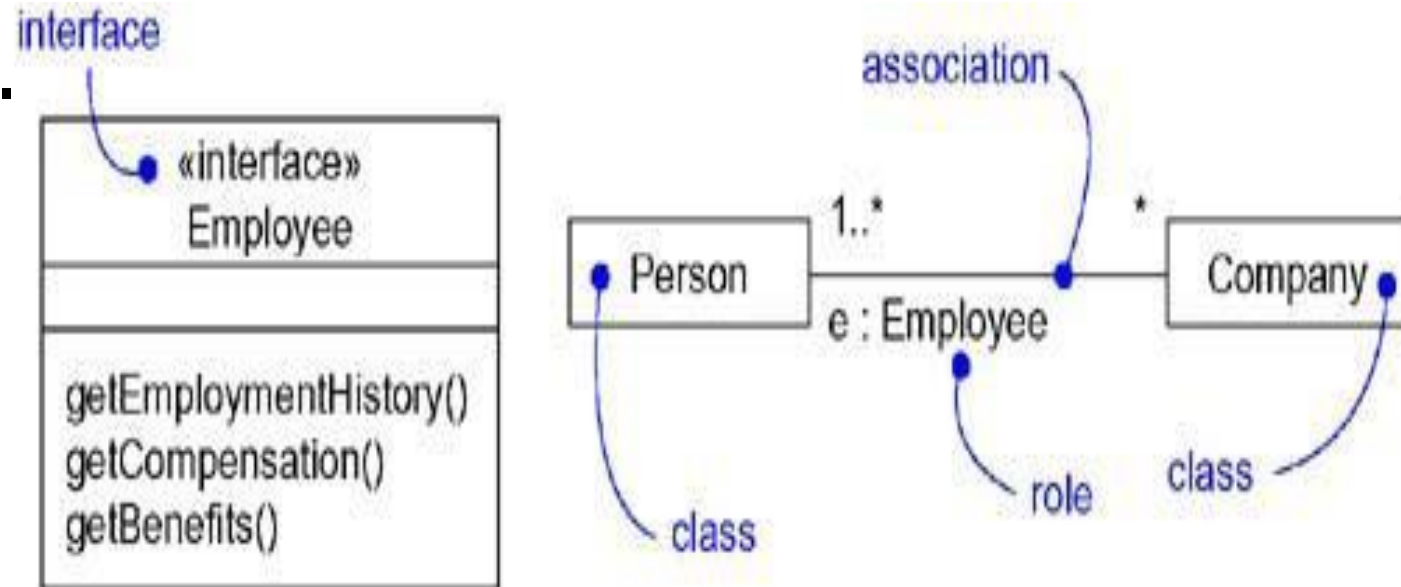
- **Interface relationships**

- An interface may participate in generalization, association, dependency and realization relationships.
- **Note:** Interfaces may also be used to specify a contract for a use case or subsystem.

- A type is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object of that type.
- To distinguish a type from an interface or a class, prepend a 'T' to every type.
- Stereotype type is used to formally model the semantics of an abstraction and its conformance to a specific interface.

- A role indicates a behavior of an entity participating in a particular context. Or, a role is the face that an abstraction presents to the world.
- For example, consider an instance of the class Person.
- Depending on the context, that Person instance may play the role of Mother, Comforter, PayerOfBills, Employee, Customer, Manager, Pilot, Singer, and so on.
- When an object plays a particular role, it presents a face to the world, and clients that interact with it expect a certain behavior depending on the role that it plays at the time.

- **For example,** an instance of Person in the role of Manager would present a different set of properties than if the instance were playing the role of Mother.
- Figure:3 indicates a role employee played by person and is represented statically there.



**Figure: 3**

- The above fig 3 describes the **Person** presents the role of **Employee** to the Company, and in that context, only the properties specified by Employee are visible and relevant to the Company.

- **Static and Dynamic modeling in UML**

- A class diagram that indicates a particular role is useful for modeling the static binding of an abstraction to its interface.
- To model the dynamic binding of an abstraction to its interface by using the become stereotype in an interaction diagram, showing an object changing from one role to another.

- **To model a dynamic type,**
- Specify the different possible types of that object by rendering each type as a class stereotyped as type (if the abstraction requires structure and behavior) or as interface (if the abstraction requires only behavior).
- Model all the roles the class of the object may take on at any point in time. It can be done in two ways:



- First, in a class diagram, explicitly type each role that the class plays in its association with other classes.
- Doing this specifies the face instances of that class put on in the context of the associated object.
- Second, also in a class diagram, specify the class-to-type relationships using generalization

- In an interaction diagram, properly render each instance of the dynamically typed class.
- Display the role of the instance in brackets below the object's name.
- To show the change in role of an object, render the object once for each role it plays in the interaction, and connect these objects with a message stereotyped as become.
- They are represented in the following figures:

Figure:4 Static modeling

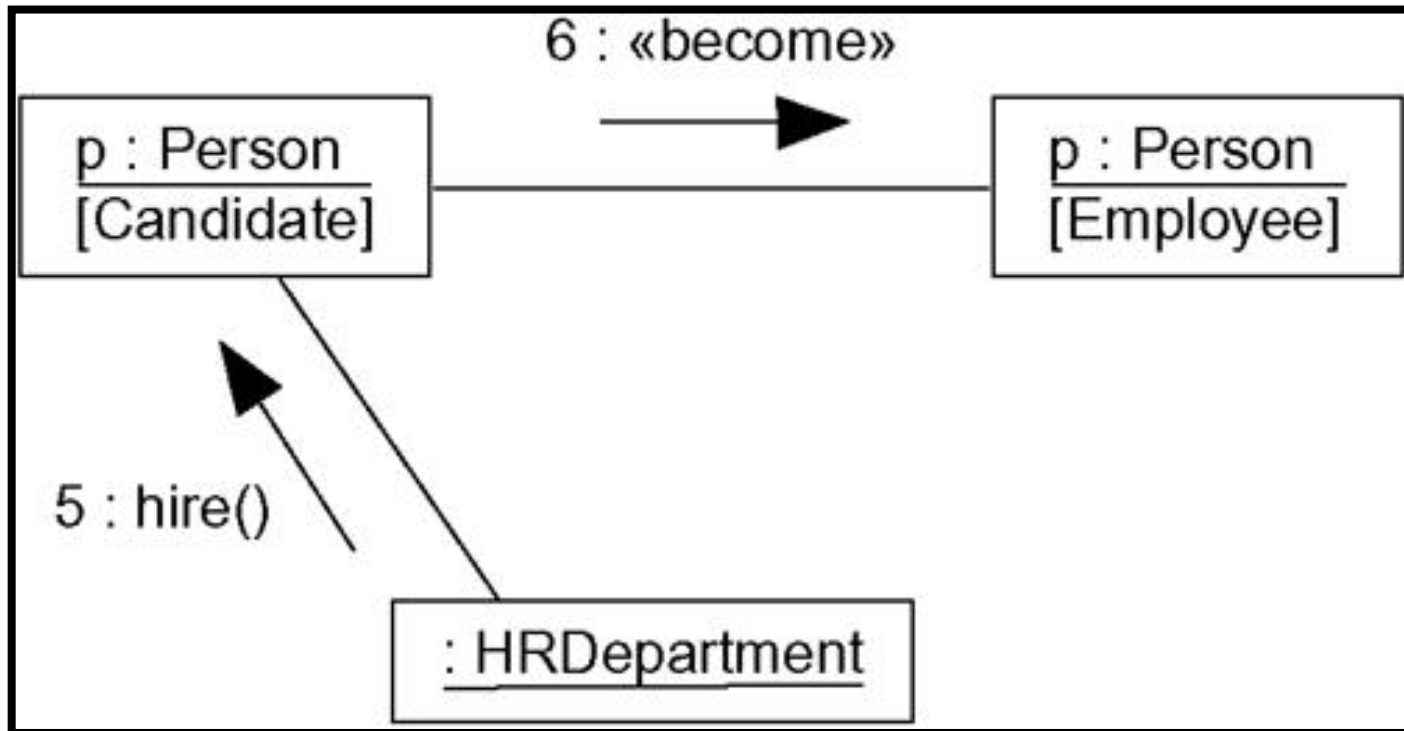
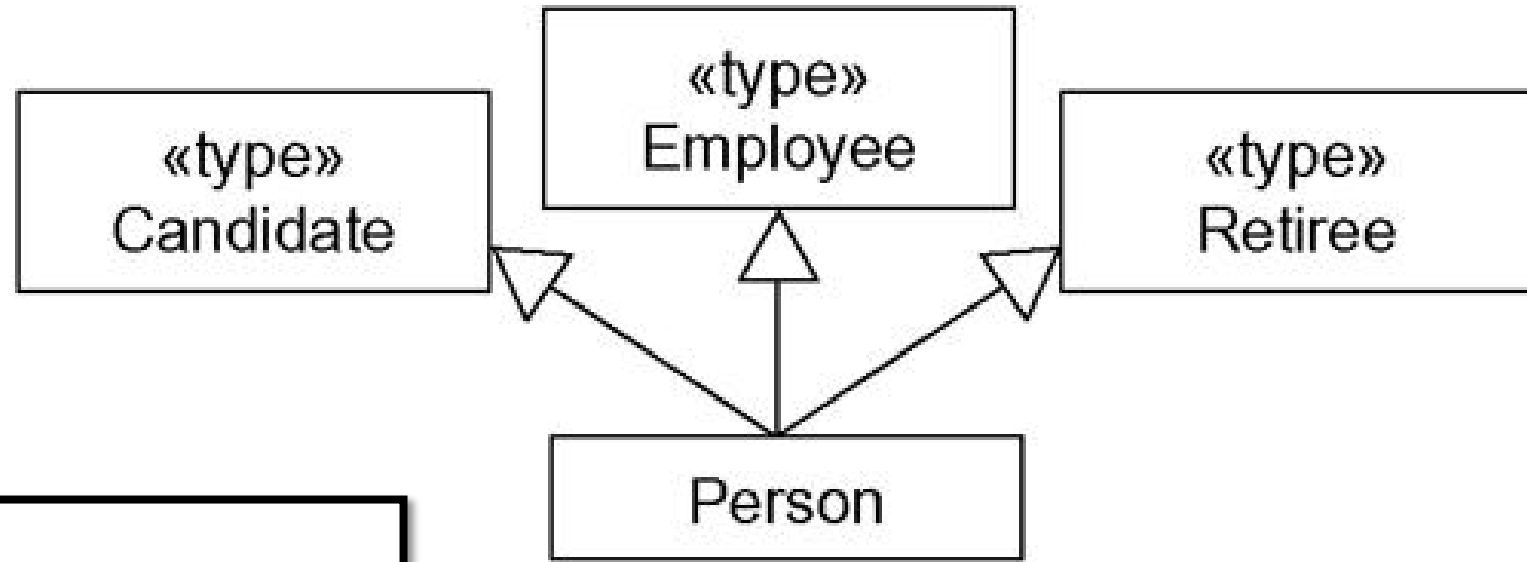
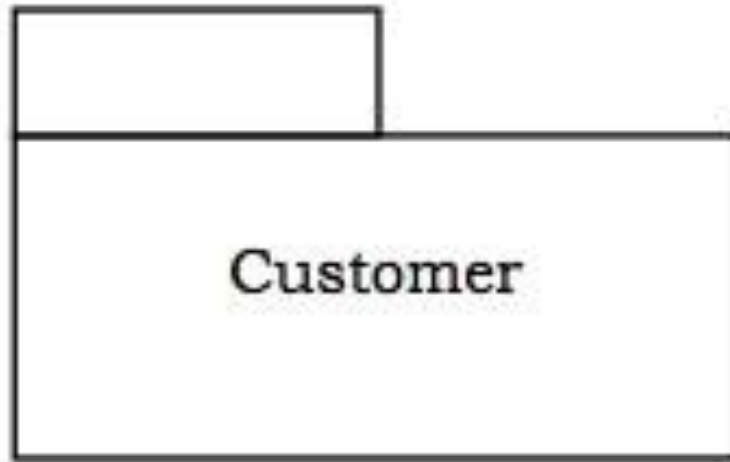


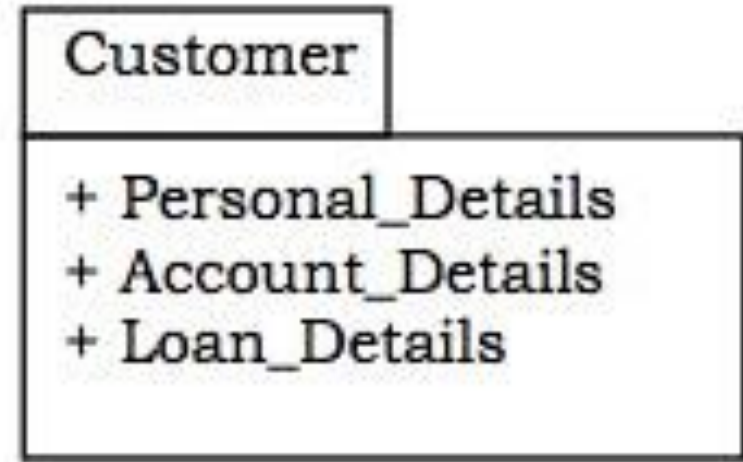
Figure:5 Dynamic-modeling

- Figure:4 shows statically that instances of the Person class may be any of the three types namely, Candidate, Employee, or Retiree.
- Figure:5 shows the dynamic nature of a person's type.
- In this fragment of an interaction diagram, p (the Person object) changes its role from Candidate to Employee.

- A package is an organized group of elements. A package may contain structural things like classes, components, and other packages in it.
- **Notation** – Graphically, a package is represented by a tabbed folder. A package is generally drawn with only its name.
- However it may have additional details about the contents of the package. See the following figures.



(a)



(b)

# Purpose of Package Diagrams

- Package diagrams are used to structure high level system elements.
- Packages are used for organizing large system which contains diagrams, documents and other key deliverables.
- ✓ Package Diagram can be used to simplify complex class diagrams, it can group classes into packages.
- ✓ A package is a collection of logically related UML elements.
- ✓ Packages are depicted as file folders and can be used on any of the UML diagrams.

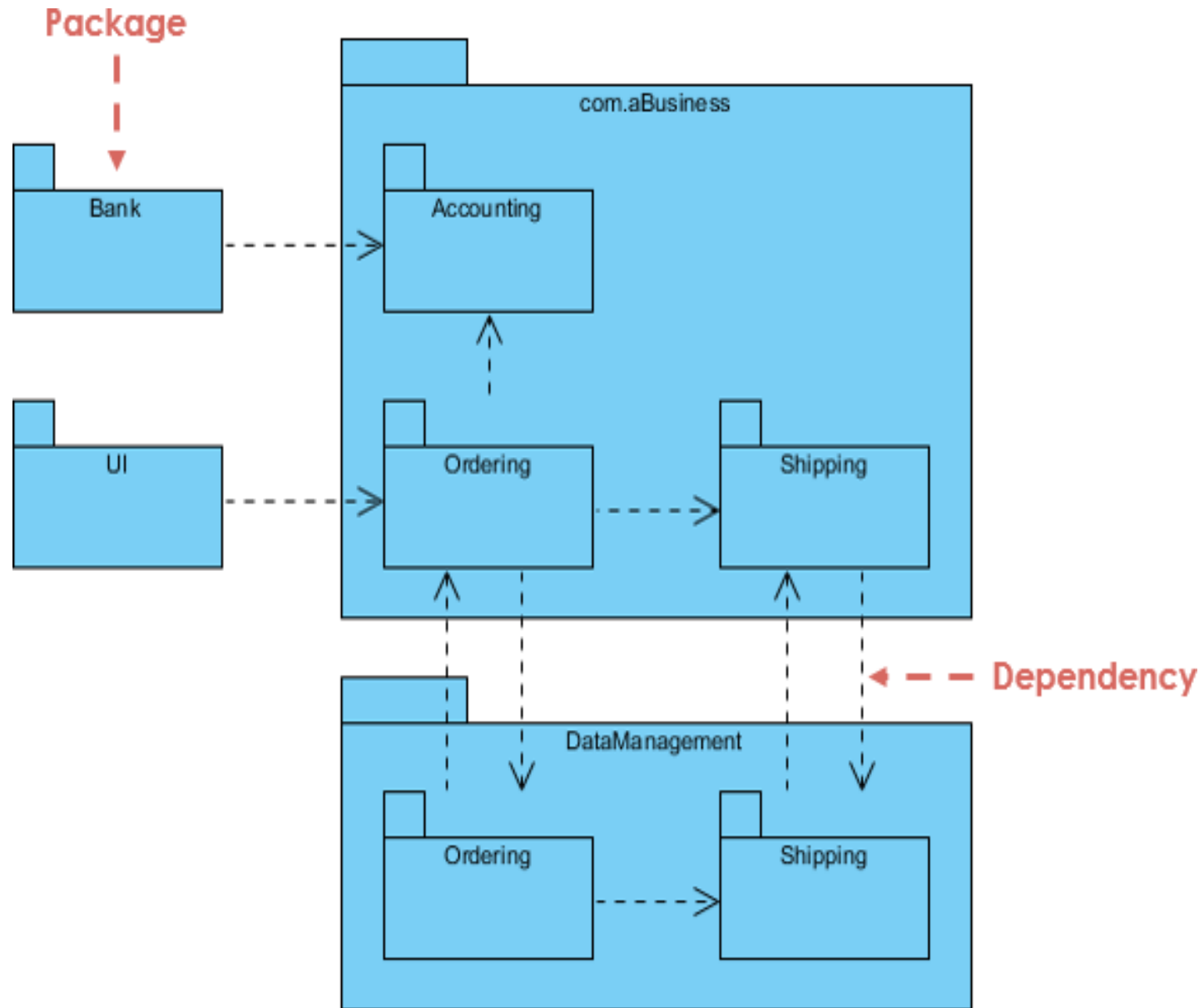
# Package Diagram at a Glance



- Package diagram is used to simplify complex class diagrams, you can group classes into packages.
- A package is a collection of logically related UML elements.
- The below diagram is a business model in which the classes are grouped into packages:
  - ✓ Packages appear as rectangles with small tabs at the top.
  - ✓ The package name is on the tab or inside the rectangle.



- ✓ The dotted arrows are dependencies.
- ✓ One package depends on another if changes in the other could possibly force changes in the first.

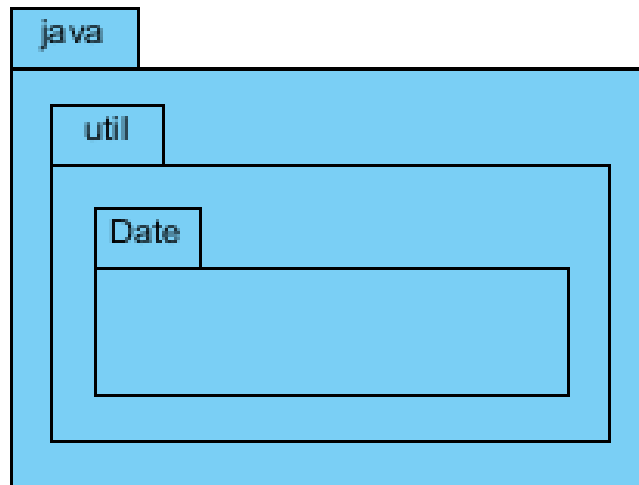


- Package diagram follows hierarchal structure of nested packages.
- There are few constraints while using package diagrams, they are as follows.
  - ✓ Package name should not be the same for a system, however classes inside different packages could have the same name.
  - ✓ Packages can include whole diagrams, name of components alone or no components at all.
  - ✓ Fully qualified name of a package has the following syntax.

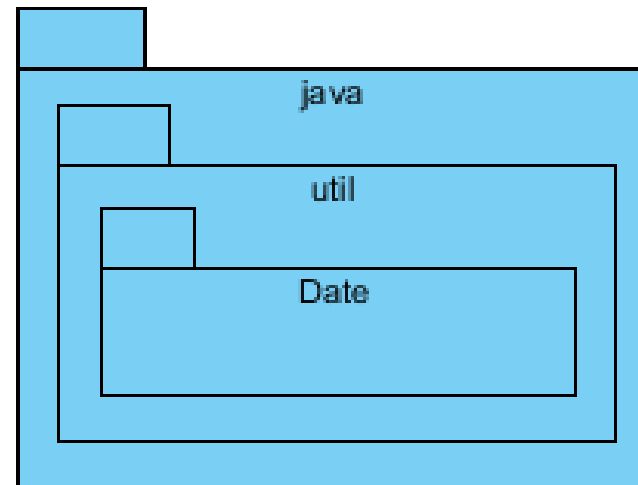
Name owing the package :: Name of the package

java :: util :: Date

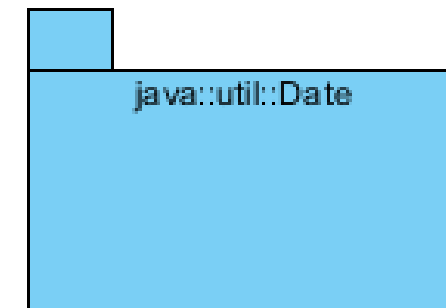
Packages can be represented by the notations with some examples shown below:



Nested, with captions in tab



Nested, with captions in package body



Fully qualified

- **Packages**
- **Terms And Concepts**
- **Common Modeling Techniques**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Packages</b></li><li>• <b>Terms And Concepts</b></li><li>• <b>Common Modeling Techniques</b></li></ul>	<ul style="list-style-type: none"><li>• Identify purpose of Packages</li><li>• Understand the concepts of Class and Object diagrams</li></ul>	<ol style="list-style-type: none"><li>1. Identify the importance of Package</li><li>2. Understand the concepts of Class and Object diagrams</li></ol>

# Outcome achieved



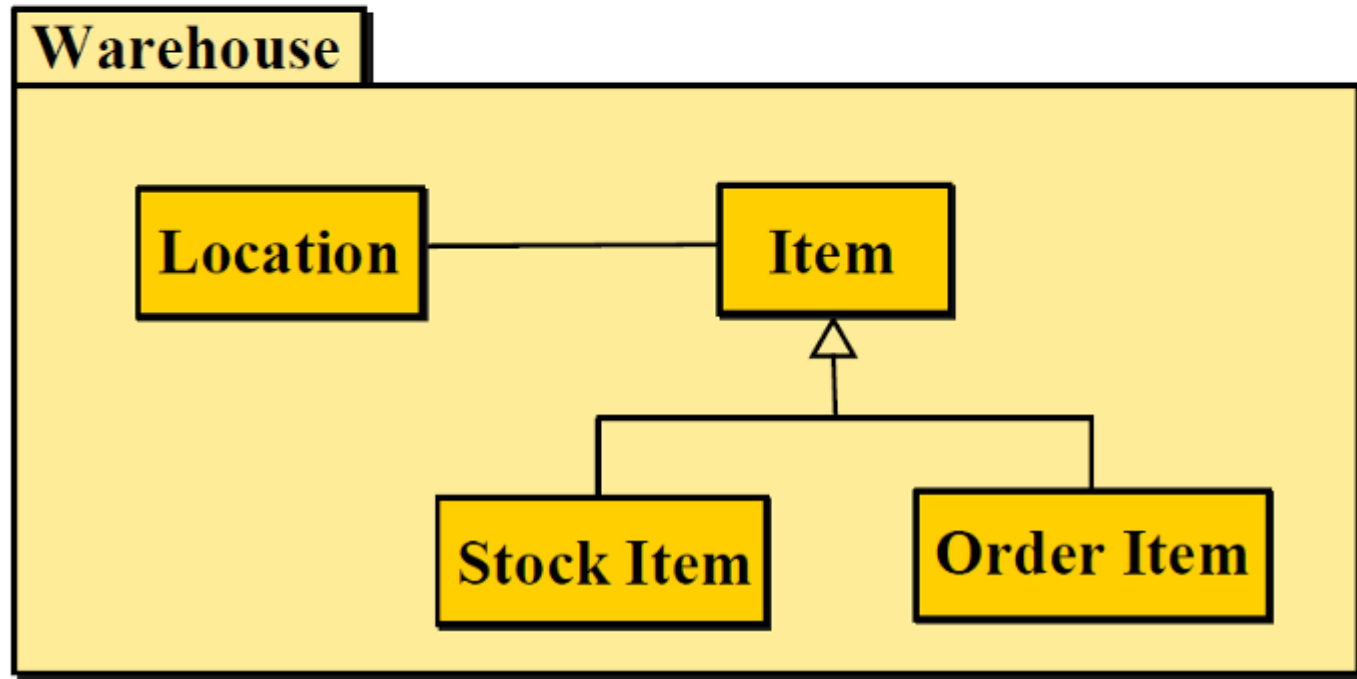
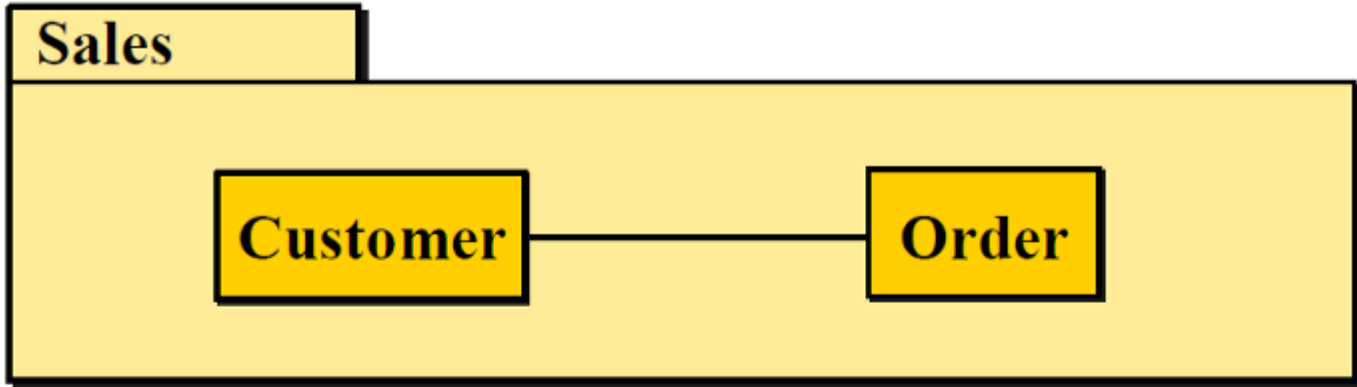
## Name of the topic:

<b>Students will be able to:</b>	
1	Identify the importance of Package
2	Understand the concepts of Class and Object diagrams

# Package Diagram — Dependency Notation



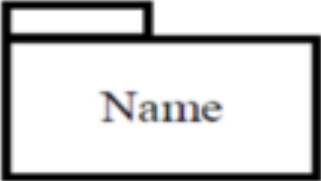
- There are two sub-types involved in dependency.
- They are `<<import>>` & `<<access>>`.
- Though there are two stereotypes users can use their own stereotype to represent the type of dependency between two packages.





# Key Elements of Package Diagram

- Packages are used to organize a large set of model elements:

<b>Construct</b>	<b>Description</b>	<b>Syntax</b>
<b>Package</b>	A grouping of model elements.	
<b>Import</b>	A dependency indicating that the public contents of the target package are added to the namespace of the source package.	<p>«import» - - - - -&gt;</p>
<b>Access</b>	A dependency indicating that the public contents of the target package are available in the namespace of the source package.	<p>«access» - - - - -&gt;</p>

# When to Use Packages?



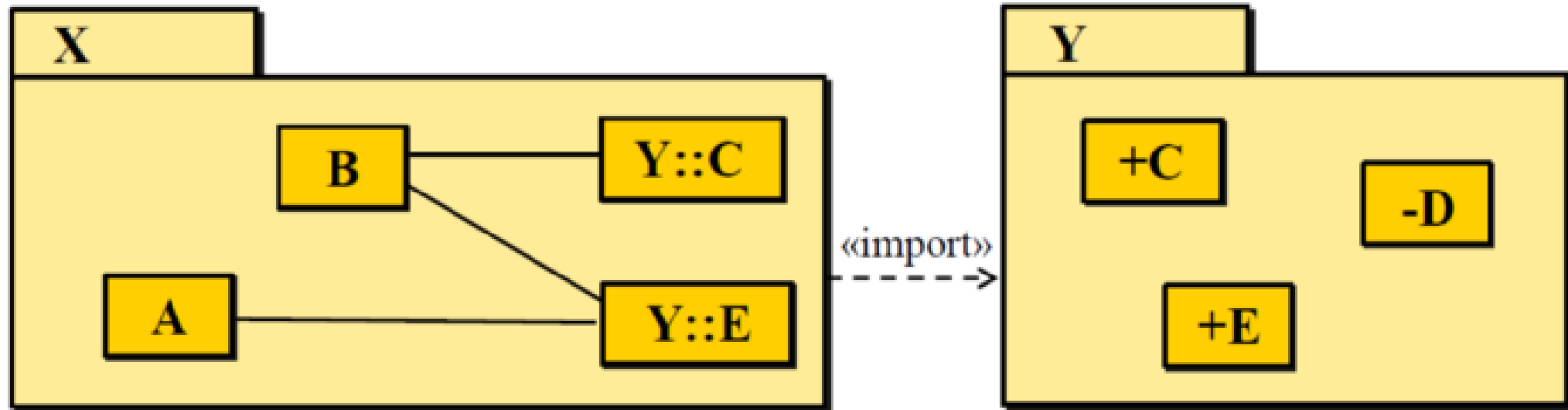
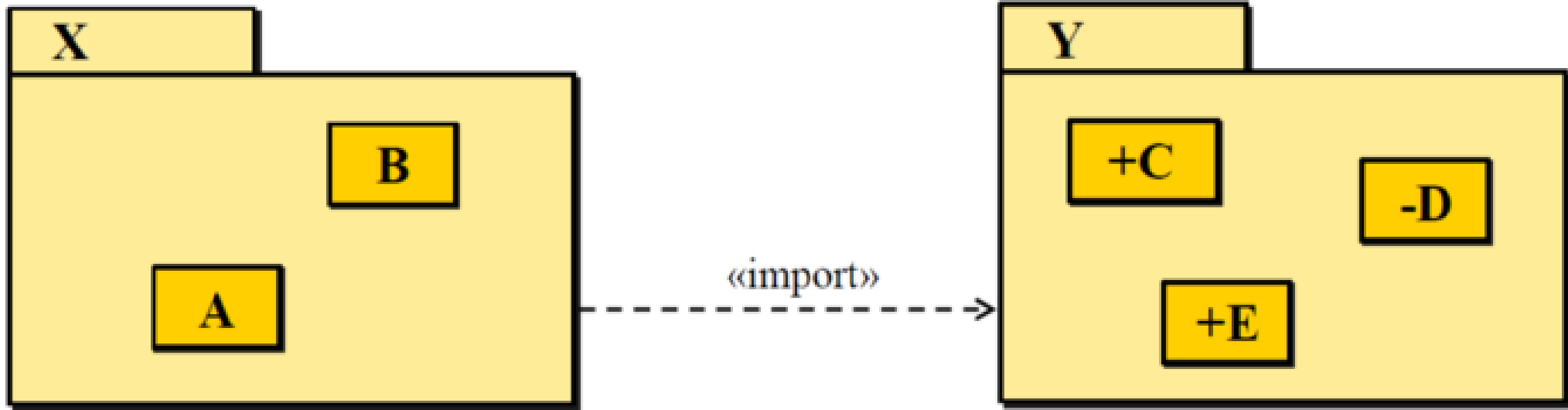
- To organize a large model
- To group related elements
- To separate namespaces

# Visibility of Packages



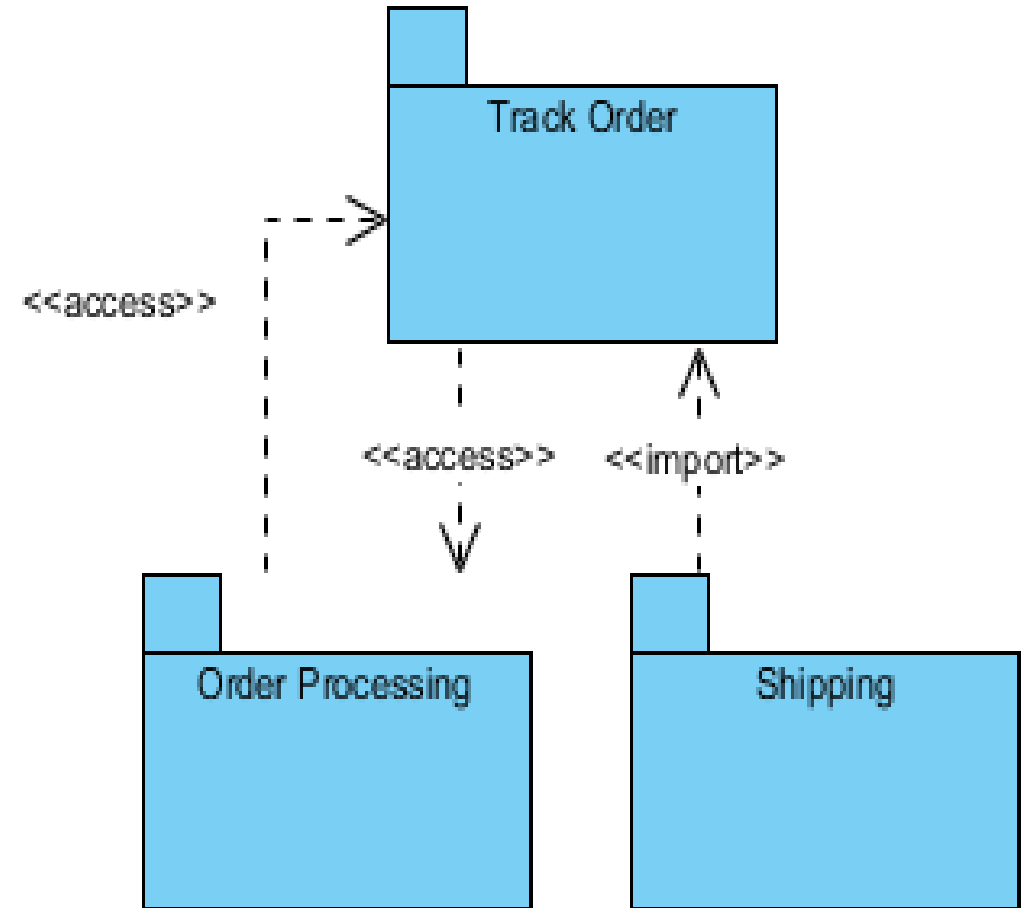
- A public element is visible to elements outside the package, denoted by '+'
- A protected element is visible only to elements within inheriting packages, denoted by '#'
- A private element is not visible at all to elements outside the package, denoted by '-'
- Same syntax for visibility of attributes and operations in classes

# Import Relationship between Packages

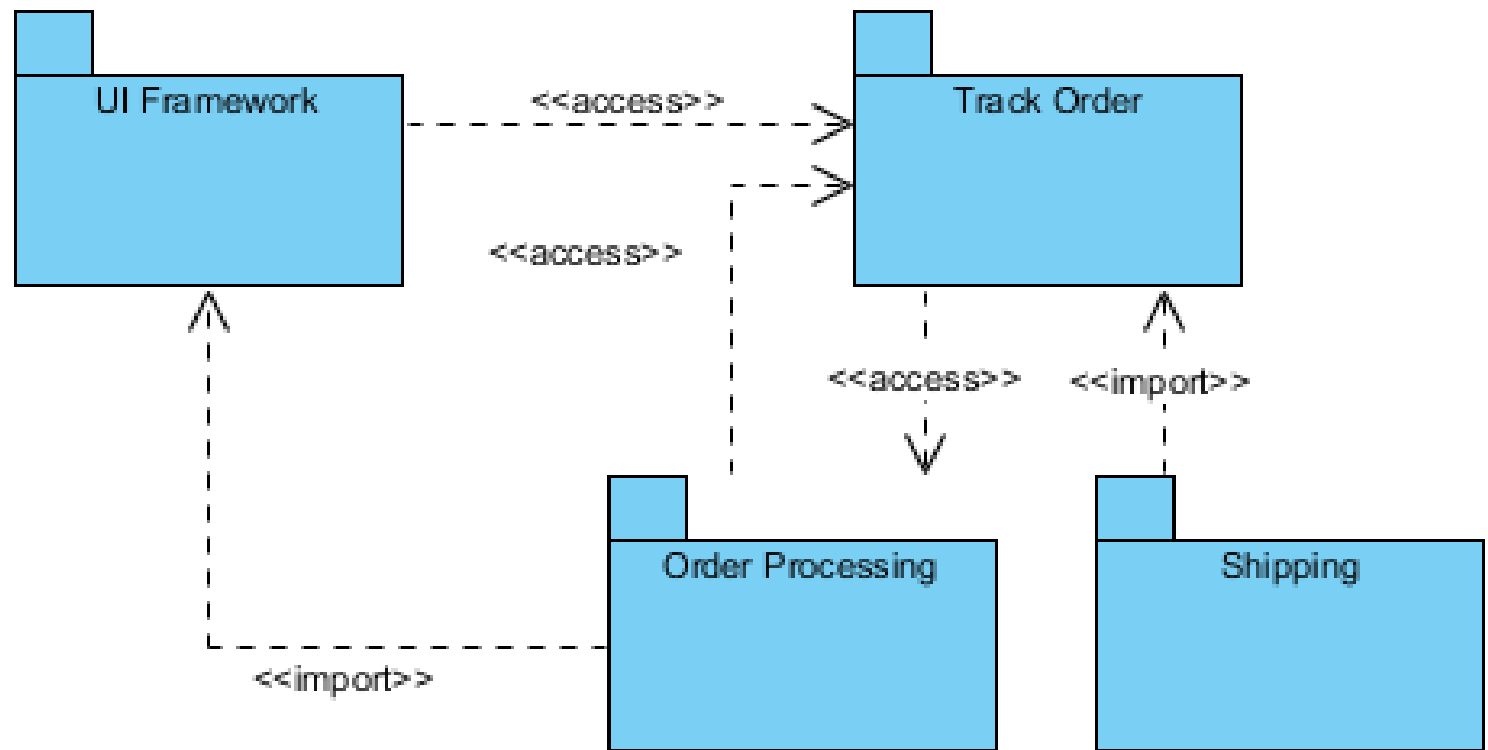


# Import and Access

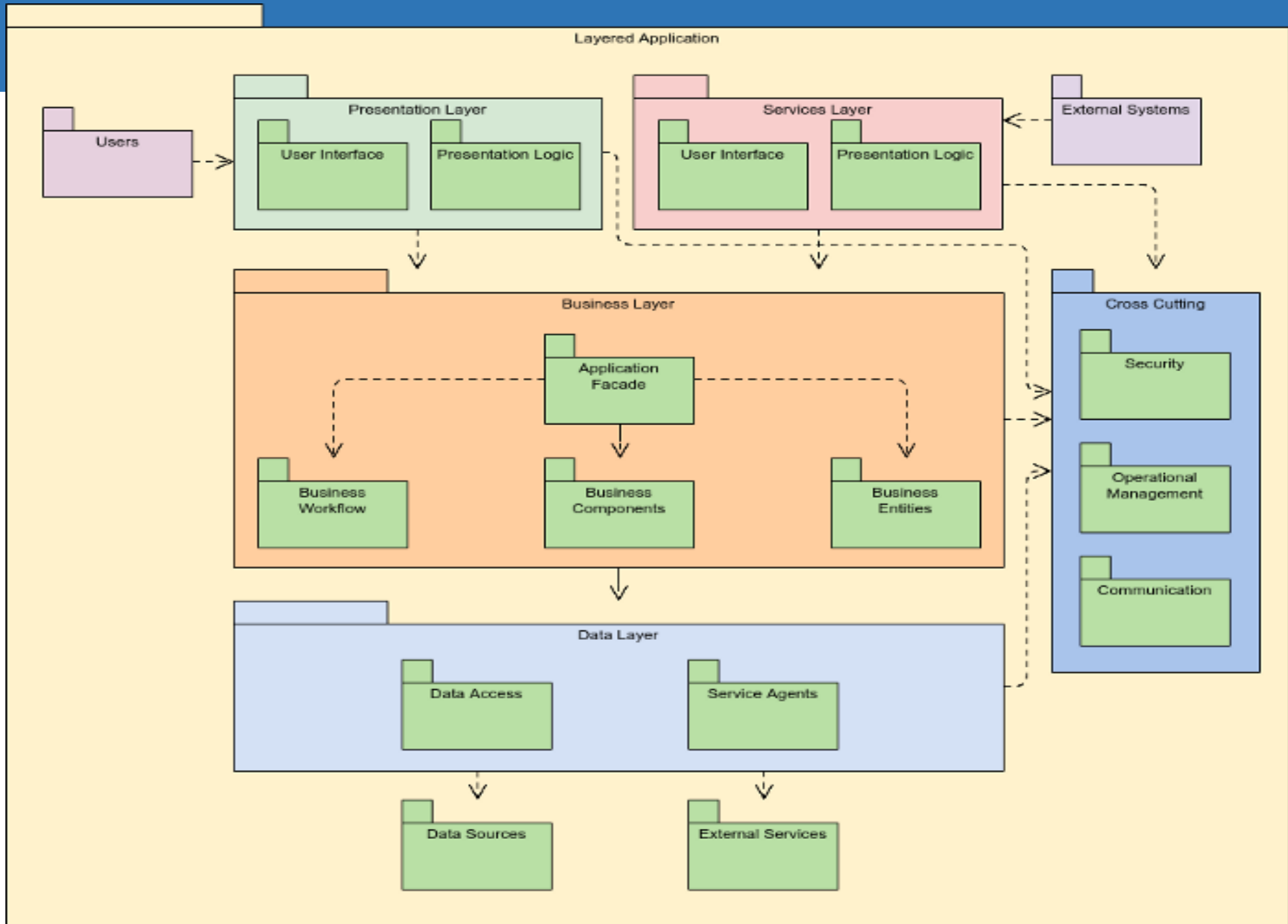
- To know shipping information, "Shipping" can import "Track Order" to make the navigation easier.
- `<<import>>` - one package imports the functionality of other package
- `<<access>>` - one package requires help from functions of other package.



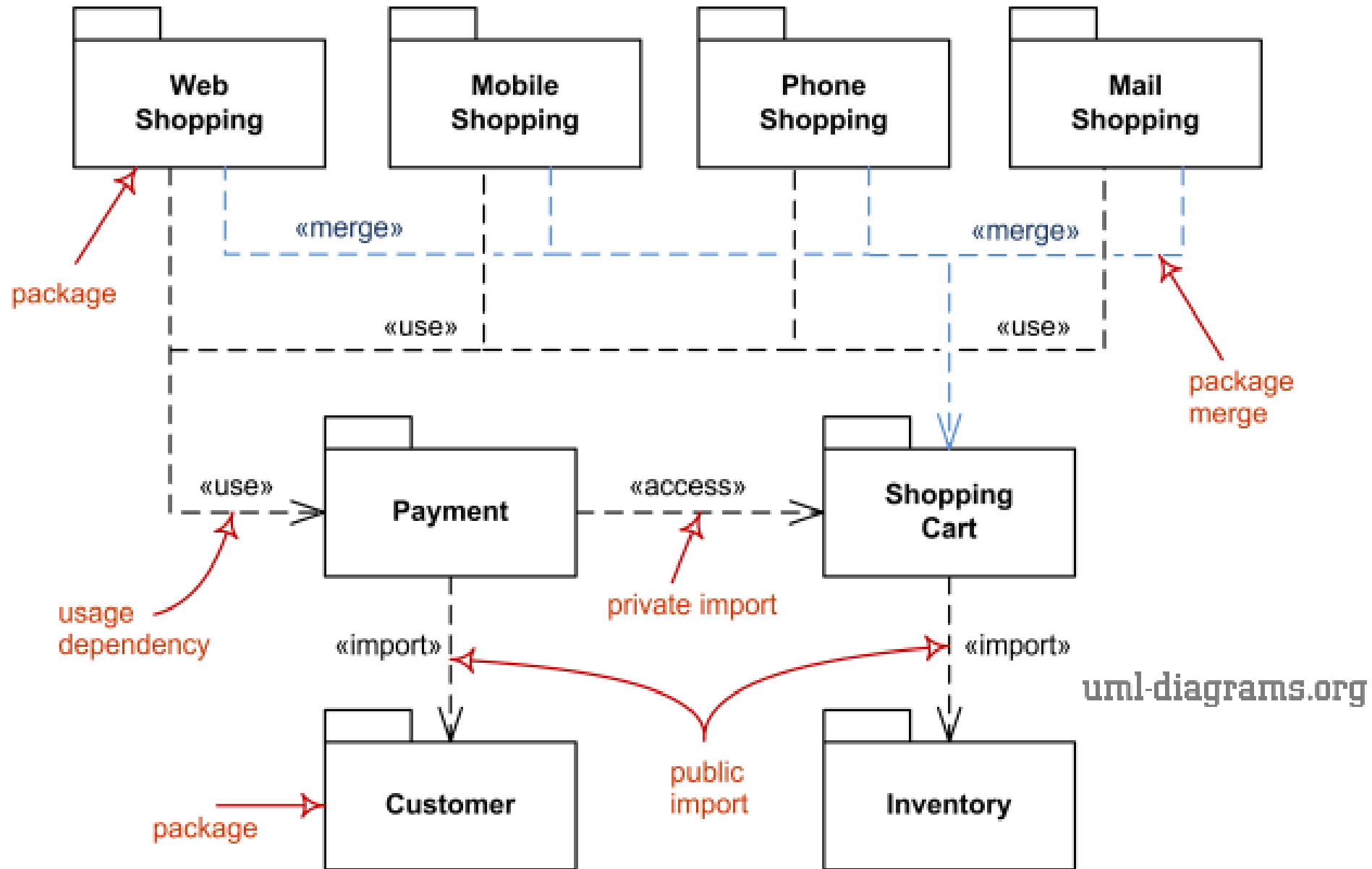
- Finally, Track Order dependency to UI Framework is also mapped which completes our Package Diagram for Order Processing subsystem.



- A package diagram is often used to describe the hierarchical relationships (groupings) between packages and other packages or objects.
- A package represents a namespace.
- Package Diagram Example — Layering Structure







- A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.
- Graphically, a class is rendered as a rectangle.

## **Common Properties :**

- A class diagram is just a special kind of diagram and shares the same common properties as do all other diagrams.
- A name and graphical content that are a projection into a model.

- What distinguishes a class diagram from all other kinds of diagrams is its particular content.
- Class diagrams commonly contain the following things:
  - ✓ Classes
  - ✓ Interfaces
  - ✓ Collaborations
  - ✓ Dependency, generalization, and association relationships

- Like all other diagrams, class diagrams may contain notes and constraints.
- Class diagrams may also contain packages or subsystems

## **1. To model the vocabulary of a system**

- Modeling the vocabulary of a system involves making a decision about which abstractions are a part of the system under consideration and which fall outside its boundaries.
- You use class diagrams to specify these abstractions and their responsibilities

## **2. To model simple collaborations**

- A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements.

## **3. To model a logical database schema**

- Think of a schema as the blueprint for the conceptual design of a database.

- In many domains, you'll want to store persistent information in a relational database or in an object-oriented database.
- You can model schemas for these databases using class diagrams

# Common Modeling Techniques



1. Modeling Simple Collaborations
2. Modeling a Logical Database Schema
3. Forward and Reverse Engineering



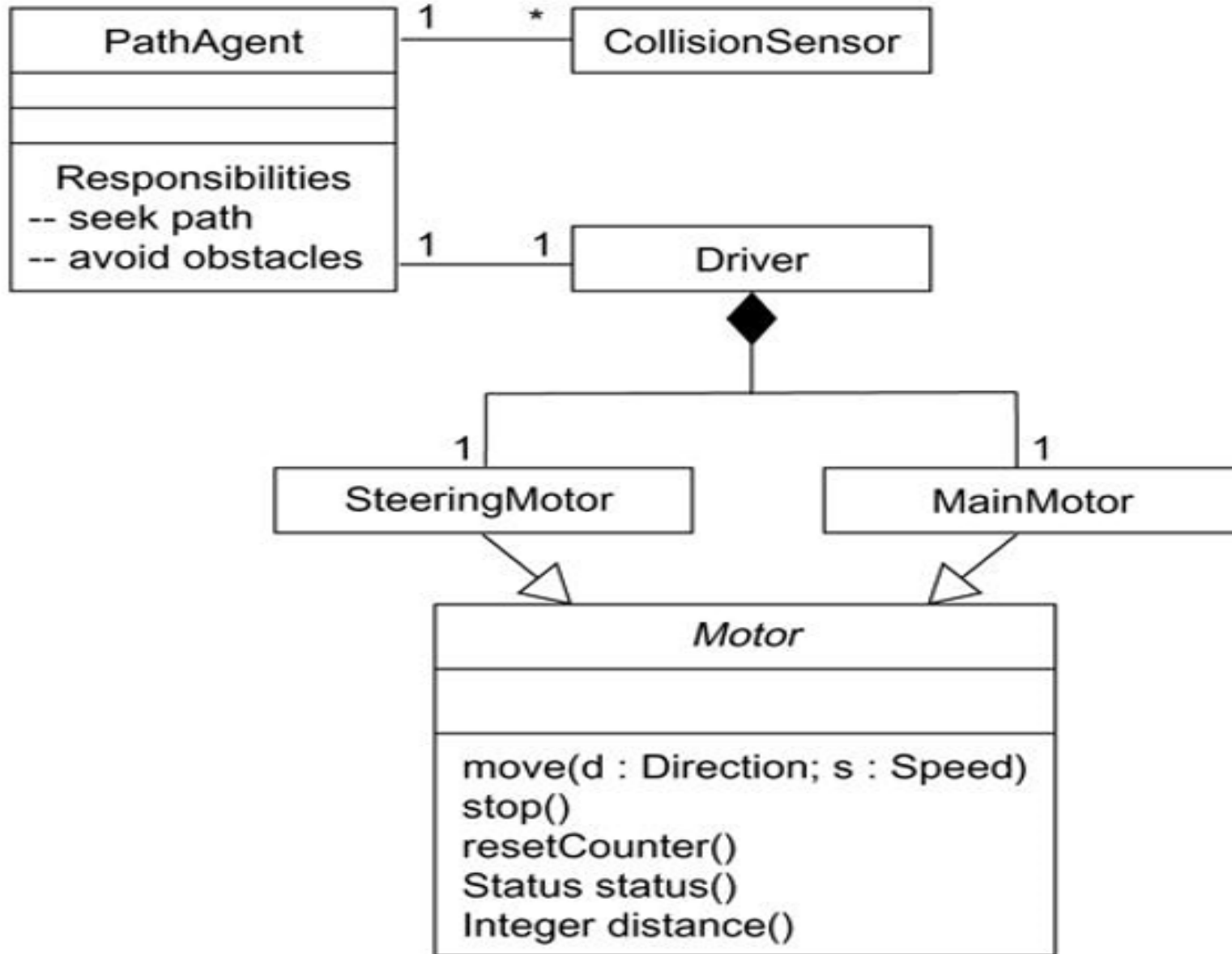
# 1. Modeling Simple Collaborations

- Identify the mechanism you'd like to model.
- A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.
- For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration.
- Identify the relationships among these things, as well.

- Use scenarios to walk through these things.
- Along the way, you'll discover parts of your model that were missing and parts that were just plain semantically wrong.
- Be sure to populate these elements with their contents.
- For classes, start with getting a good balance of responsibilities.
- Then, over time, turn these into concrete attributes and operations.

- For example, Figure shows a set of classes drawn from the implementation of an autonomous robot.
- The figure focuses on the classes involved in the mechanism for moving the robot along a path.
- You'll find one abstract class (Motor) with two concrete children, SteeringMotor and MainMotor.
- Both of these classes inherit the five operations of their parent, Motor.

- The two classes are, in turn, shown as parts of another class, Driver.
- The class PathAgent has a one-to-one association to Driver and a one-to-many association to CollisionSensor.
- No attributes or operations are shown for PathAgent, although its responsibilities are given



- **Common Modeling Techniques**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Common Modeling Techniques</b></li></ul>	<ul style="list-style-type: none"><li>• Understand the concepts of Class and Object diagrams</li></ul>	<ol style="list-style-type: none"><li>1. Understand the concepts of Class and Object diagrams</li></ol>

# Outcome achieved



## Name of the topic:

<b>Students will be able to:</b>	
1	Understand the concepts of Class and Object diagrams



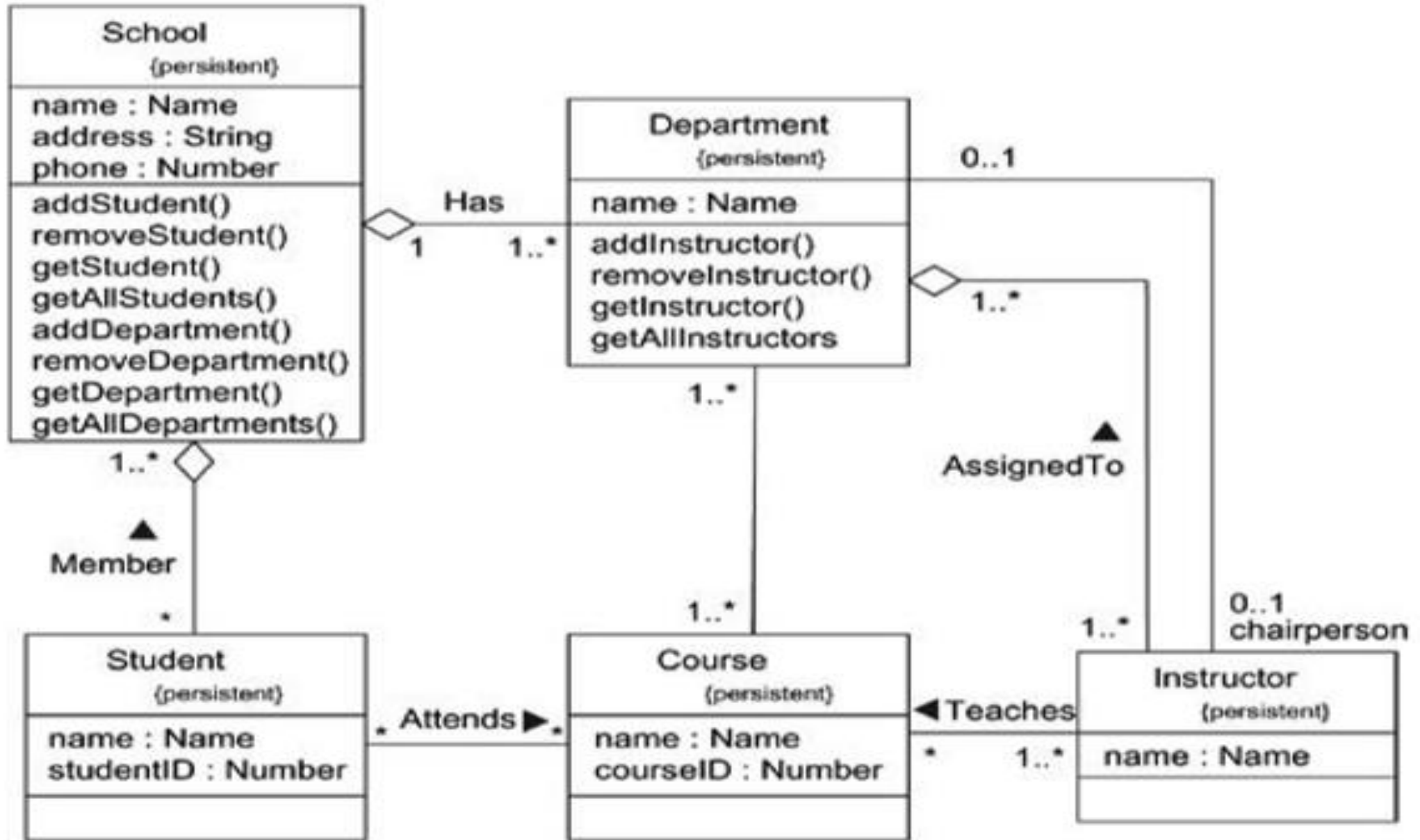
## 2. Modeling a Logical Database Schema

- Identify those classes in your model whose state must go beyond the lifetime of their applications.
- Create a class diagram that contains these classes and mark them as persistent (a standard tagged value).
- You can define your own set of tagged values to address database-specific details.

- Expand the structural details of these classes.
- In general, this means specifying the details of their attributes and focusing on the associations and their cardinalities that structure these classes.
- Watch for common patterns that complicate physical database design, such as cyclic associations, one-to-one associations, and n-ary associations.
- Where necessary, create intermediate abstractions to simplify your logical structure.

- Consider also the behavior of these classes by expanding operations that are important for data access and data integrity.
- In general, to provide a better separation of concerns, business rules concerned with the manipulation of sets of these objects should be encapsulated in a layer above these persistent classes.
- Where possible, use tools to help you transform your logical design into a physical design.

- Figure shows a set of classes drawn from an information system for a school.
- find the classes named Student, Course, and Instructor.
- There's an association between Student and Course, specifying that students attend courses.
- Furthermore, every student may attend any number of courses and every course may have any number of students.

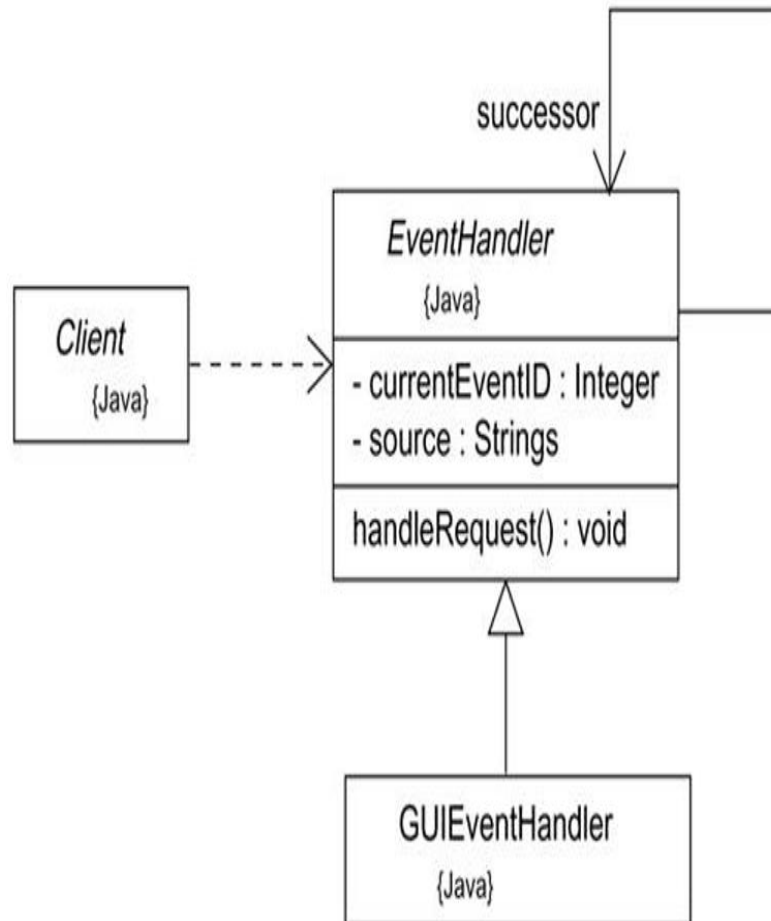


- Identify the rules for mapping to your implementation language or languages of choice.
- Depending on the semantics of the languages choose, may have to constrain use of certain UML features.
- For example, the UML permits you to model multiple inheritance.
- Use tagged values to specify your target language.

- Can do this at the level of individual classes if you need precise control.
- Can also do so at a higher level, such as with collaborations or packages.
- Use tools to forward engineer of models.
- Figure illustrates a simple class diagram specifying involves three classes: Client, EventHandler, and GUIEventHandler.

- Client and EventHandler are shown as abstract classes, whereas GUIEventHandler is concrete.
- EventHandler has the usual operation expected of this pattern (handleRequest), although two private attributes have been added for this instantiation.





```

public abstract class EventHandler {
    EventHandler successor;
    private Integer currentEventID;
    private String source;

    EventHandler() {}
    public void handleRequest() {}
}
  
```

# **MODULE-III**

# **ARCHITECTURAL MODELING**

- **Interaction diagrams**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Interaction Diagrams</b></li></ul>	<ul style="list-style-type: none"><li>• Describe interaction diagrams and their modeling techniques</li></ul>	<ul style="list-style-type: none"><li>• Apply advanced behavioral modeling techniques in design and drawing UML diagrams for various systems</li></ul>

# Outcome achieved



## Name of the topic:

### Students will be able to:

1

Describe interaction diagrams and their modeling techniques

- This is having dynamic behaviour and it is used to describe interactions among the different elements in the model.
- This interactive behavior is represented in UML by two diagrams known as **Sequence diagram** and **Collaboration diagram**.
- *Sequence diagram* emphasizes on time sequence of messages and *collaboration diagram* emphasizes on the structural organization of the objects that send and receive messages.

# Purpose of Interaction Diagrams



- The purpose of interaction diagrams is to visualize the interactive behavior of the system.

The purpose of interaction diagram is –

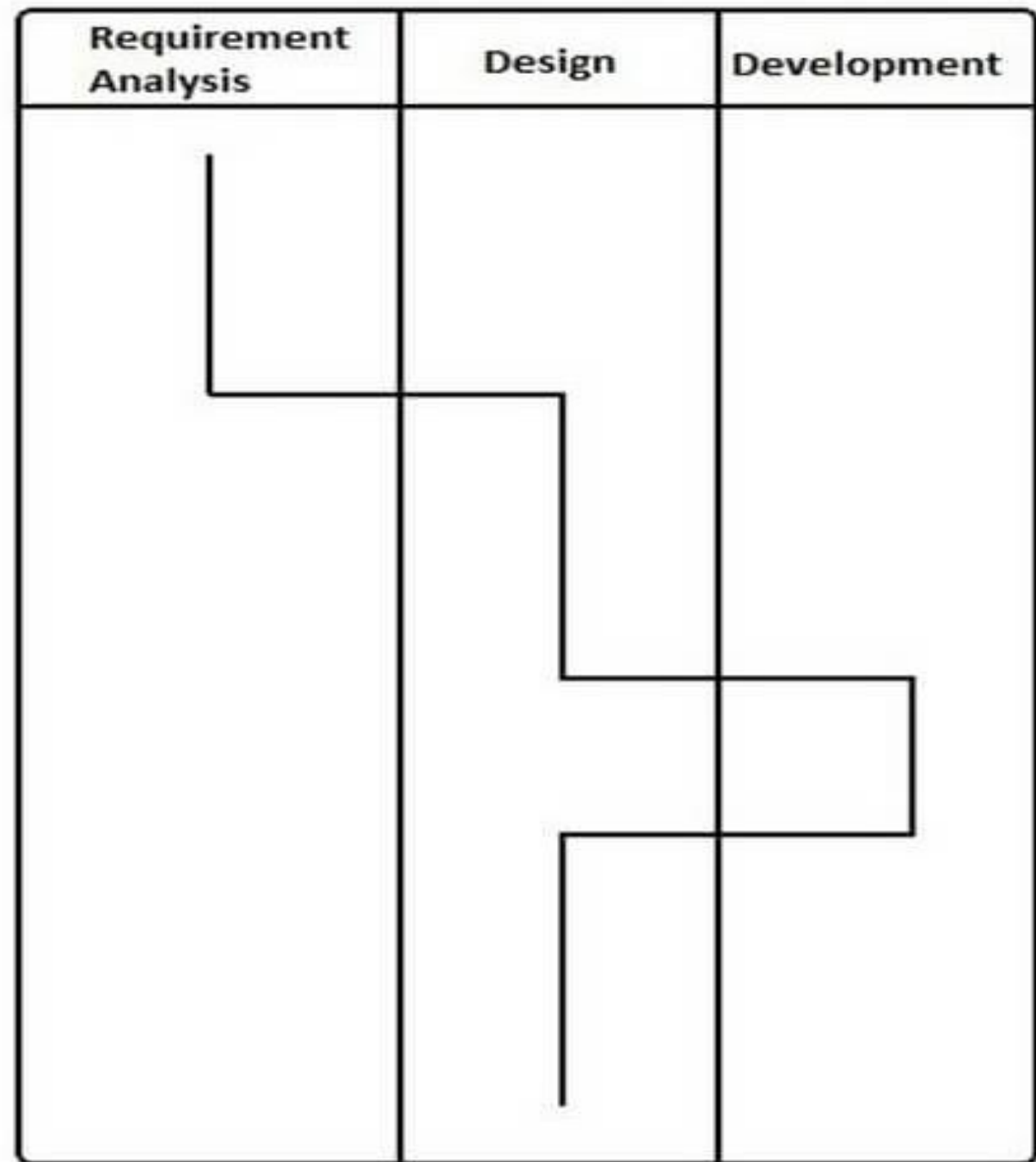
- To capture the dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe the structural organization of the objects.
- To describe the interaction among objects.

- Following are the different types of interaction diagrams defined in UML:
  1. Sequence diagram
  2. Collaboration diagram
  3. Timing diagram



# Timing Diagram

- The timing diagram is merely just a waveform or a graph which is used to describe the state of a lifeline at any instance of time.



- The output of the previous phase at that given instance of time is given to the second phase as an input.
- Thus, the timing diagram can be used to describe SDLC (Software Development Life Cycle) in UML.

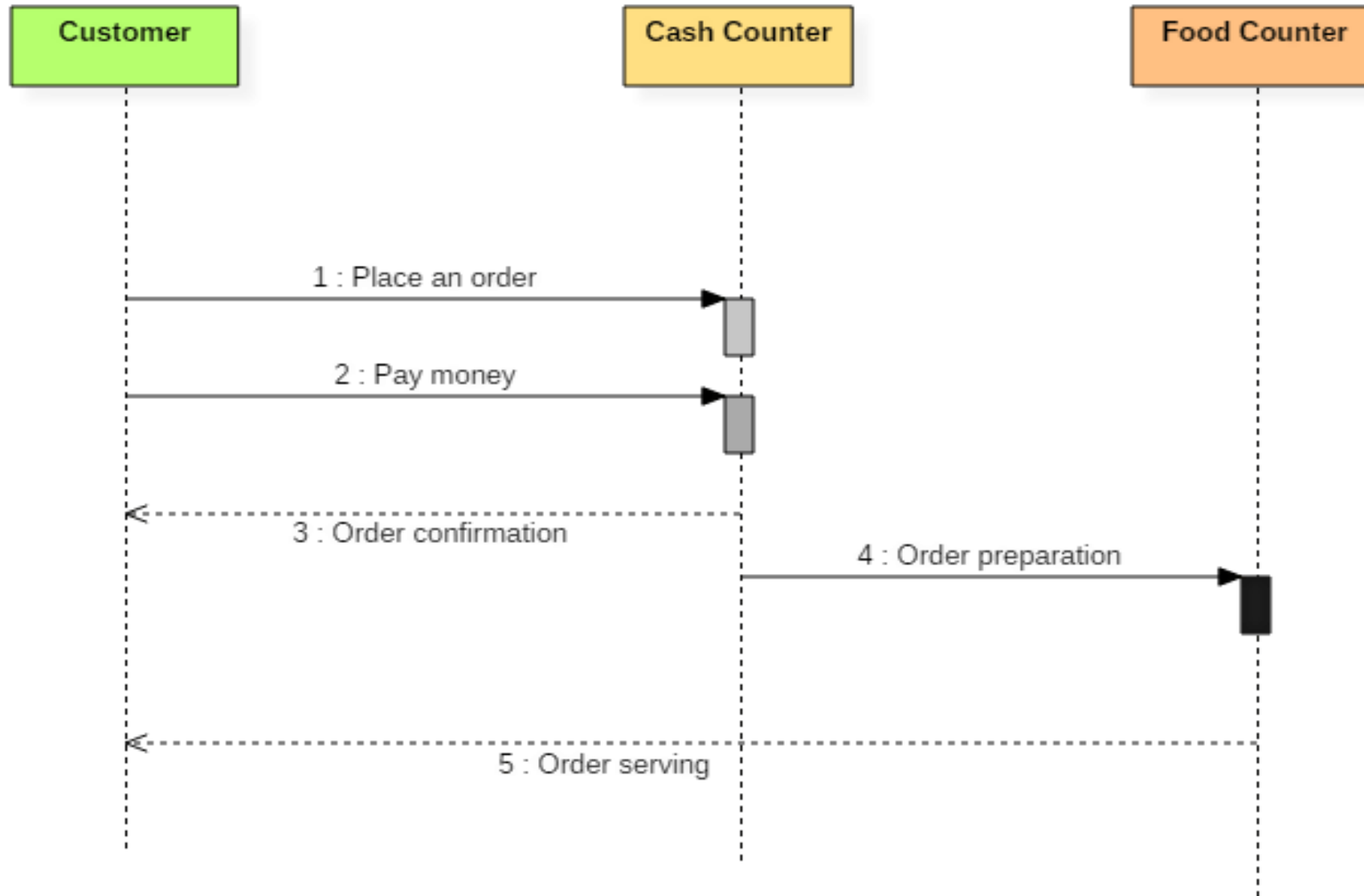
## **Drawbacks of a Timing Diagram**

- Timing diagrams are ***difficult to understand.***
- Timing diagrams are ***difficult to maintain.***

- In UML, timing diagrams are read from left to right according to the name of a lifeline specified at the left edge.
- Timing diagrams are used to explain the detailed time processing of a particular object.

- The purpose of a sequence diagram in UML is to visualize the sequence of a message flow in the system.
- sequence diagram is used to capture the behavior of any scenario.
- In a sequence diagram, a lifeline is represented by a vertical bar.
- A message flow between two or more objects is represented using a vertical dotted line

interaction McDonald's Order System



- **Benefits of a Sequence Diagram**

- Sequence diagrams are used to explore any real application or a system.
- Sequence diagrams are used to represent message flow from one object to another object.
- Sequence diagrams are easier to maintain.
- Sequence diagrams can be easily updated according to the changes within a system.
- Sequence diagram allows reverse as well as forward engineering.

- Forward engineering a object diagram is theoretically possible but practically of limited value as the objects are created and destroyed dynamically at runtime, we cannot represent them statically.
- To reverse engineering a object diagram,
  1. Choose the target (context) you want to reverse engineer.
  2. Use a tool to stop execution at a certain moment in time.

3. Identify the objects that collaborate with each other and represent them in an object diagram.
4. To understand their semantics, expose these object's states.
5. Also identify the links between the objects to understand their semantics.

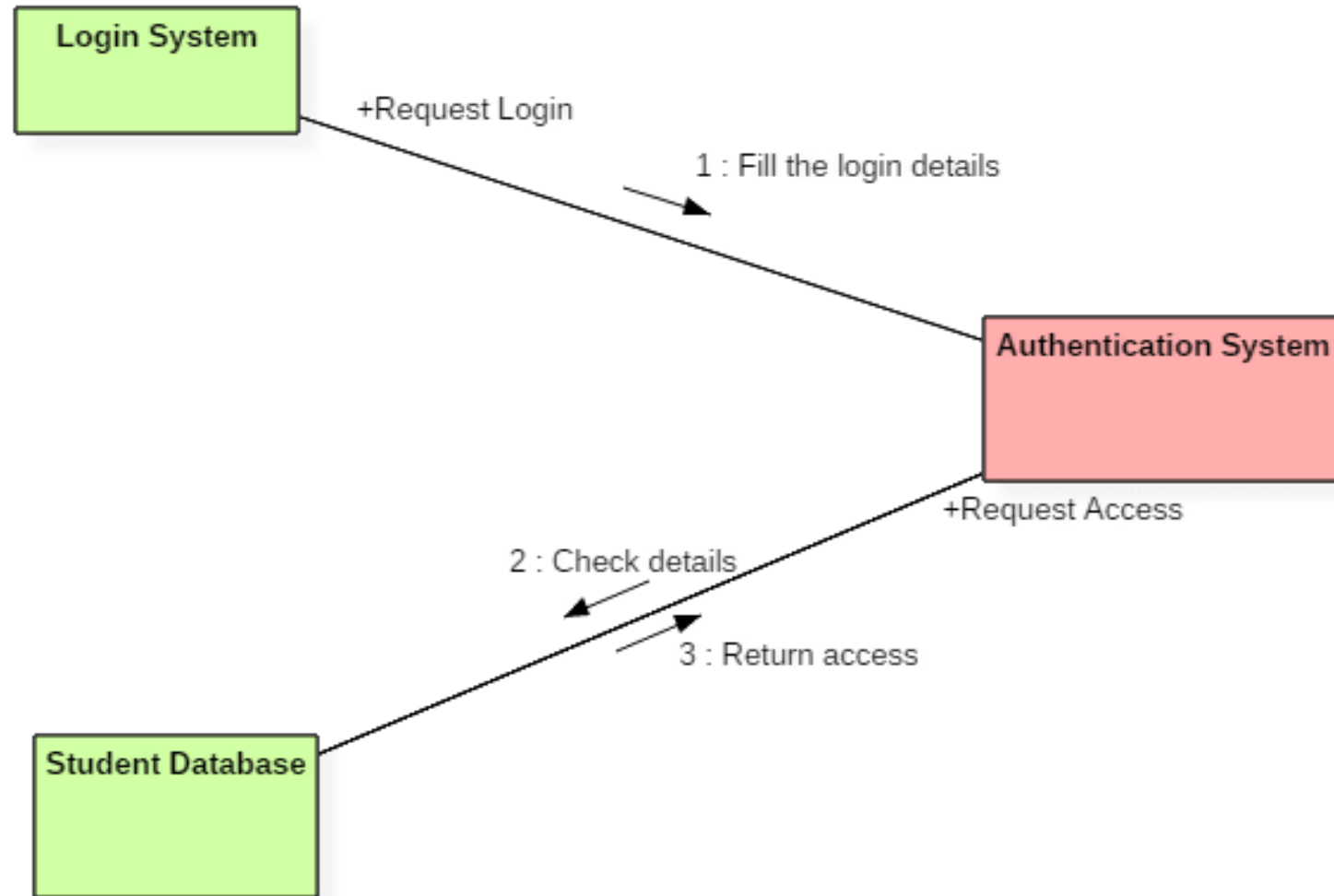


<b>S.N O</b>	<b>Forward Engineering</b>	<b>Reverse Engineering</b>
1.	In forward engineering, the application are developed with the given requirements.	In reverse engineering or backward engineering, the information are collected from the given application.
2.	Forward Engineering is high proficiency skill.	Reverse Engineering or backward engineering is low proficiency skill.
3.	Forward Engineering takes more time to develop an application.	While Reverse Engineering takes less time to develop an application.
4.	The nature of forward engineering is Prescriptive.	The nature of reverse engineering or backward engineering is Adaptive.
5.	In forward engineering, production is started with given requirements.	In reverse engineering, production is started by taking existing product.
6.	The example of forward engineering are construction of electronic kit, construction DC MOTOR etc.	The example of backward engineering are research on Instruments etc.

- It is also called as a ***communication diagram***.
- It emphasizes the structural aspects of an interaction diagram - how lifeline connects.
- Its syntax is similar to that of sequence diagram except that lifeline don't have tails.
- Messages passed over sequencing is indicated by numbering each message hierarchically.
- Compared to the sequence diagram communication diagram is semantically weak.

- 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.
- While modeling collaboration diagrams w.r.t sequence diagrams, some information may be lost.

interaction Student Management System



## **Drawbacks of a Collaboration Diagram**

- Collaboration diagrams can become complex when too many objects are present within the system.
- It is hard to explore each object inside the system.
- Collaboration diagrams are time consuming.
- The state of an object changes momentarily, which makes it difficult to keep track of every single change that occurs within an object of a system.

- **Activity Diagram**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Activity Diagram</b></li></ul>	<ul style="list-style-type: none"><li>• Identify importance of Activity Diagram</li></ul>	<ul style="list-style-type: none"><li>• Identify importance of Activity Diagram</li></ul>

# Outcome achieved



## Name of the topic:

<b>Students will be able to:</b>	
1	Describe message flow among several activities.



- It captures the dynamic behavior of the system.
- activity diagram is used to show message flow from one activity to another
- Activity diagrams are not only used for visualizing the dynamic nature of a system, but they are also used to construct the executable system by using forward and reverse engineering techniques.

- In UML, an activity diagram provides a view of the behavior of a system by describing the sequence of actions in a process.
- Activity diagrams are similar to flowcharts because they show the flow between the actions in an activity;
- In activity diagrams, you use activity nodes and activity edges to model the flow of control and data between actions.
- Activity diagrams are helpful in the following phases of a project:

- Before starting a project, you can create activity diagrams to model the most important workflows.
- During the **requirements phase**, you can create activity diagrams to illustrate the flow of events that the use cases describe.
- During the **analysis** and **design phases**, you can use activity diagrams to help define the behavior of operations.

# Activity Diagram Notations

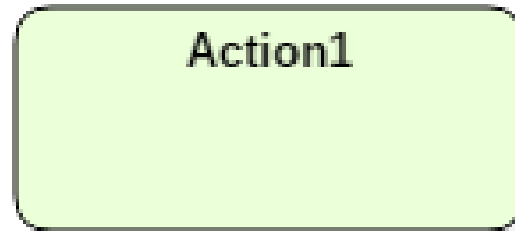


- Activity diagrams symbol can be generated by using the following notations:
- **Initial states:** The starting stage before an activity takes place is depicted as the initial state
- **Final states:** The state which the system reaches when a specific process ends is known as a Final State
- **State or an activity box or action box**
- **Decision box:** It is a diamond shape box which represents a decision with alternate paths. It represents the flow of control.

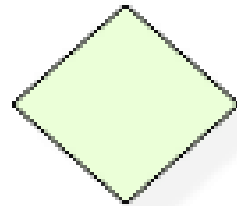
# Activity Diagram Notation and Symbol



initial-state



action-box



decision-box



final-state

Following rules must be followed while developing an activity diagram,

- All activities in the system should be named.
- Activity names should be meaningful.
- Constraints must be identified.
- Activity associations must be known.

- It does not show any message flow from one activity to another.
- Activity diagram is sometimes considered as the flowchart.
- Before drawing an activity diagram, we should identify the following elements
  1. Activities
  2. Association
  3. Conditions
  4. Constraints

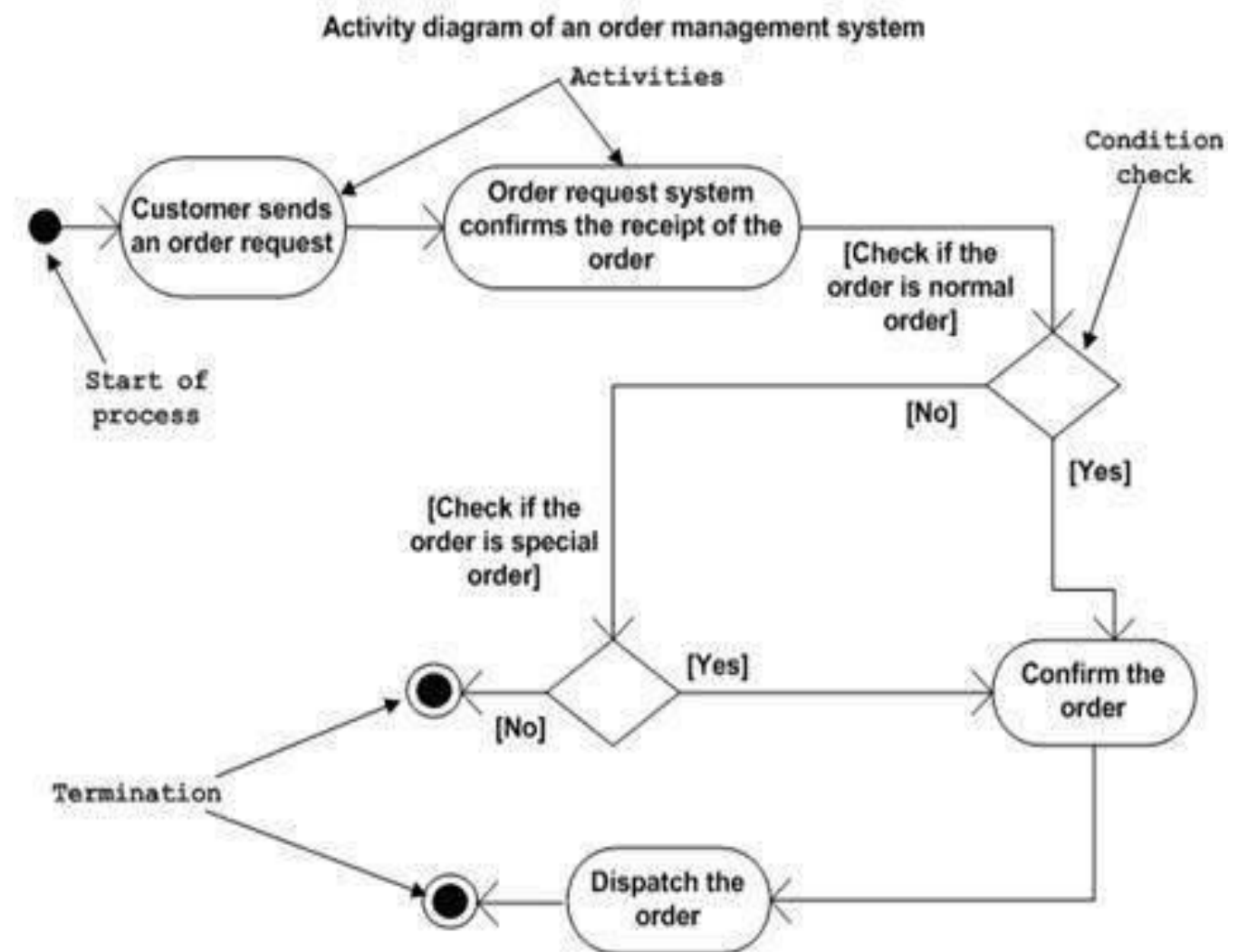
• Following diagram is drawn with the four main activities

✓ Send order by the customer

✓ Receipt of the order

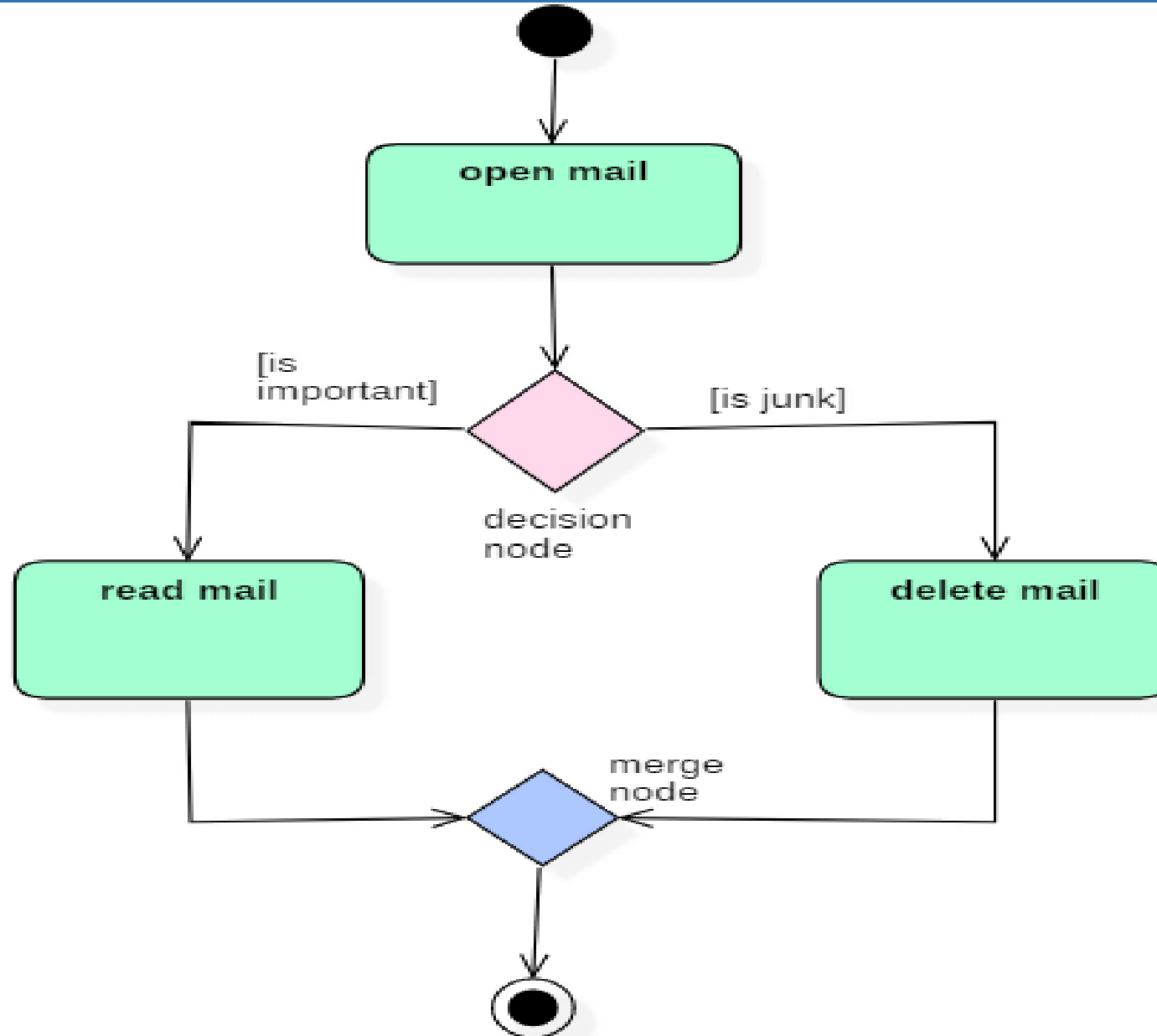
✓ Confirm the order

✓ Dispatch the order





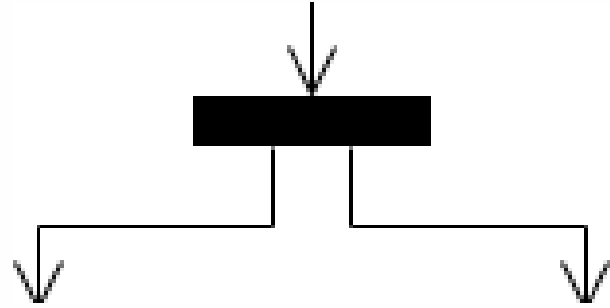
# Ex: processing e-mails



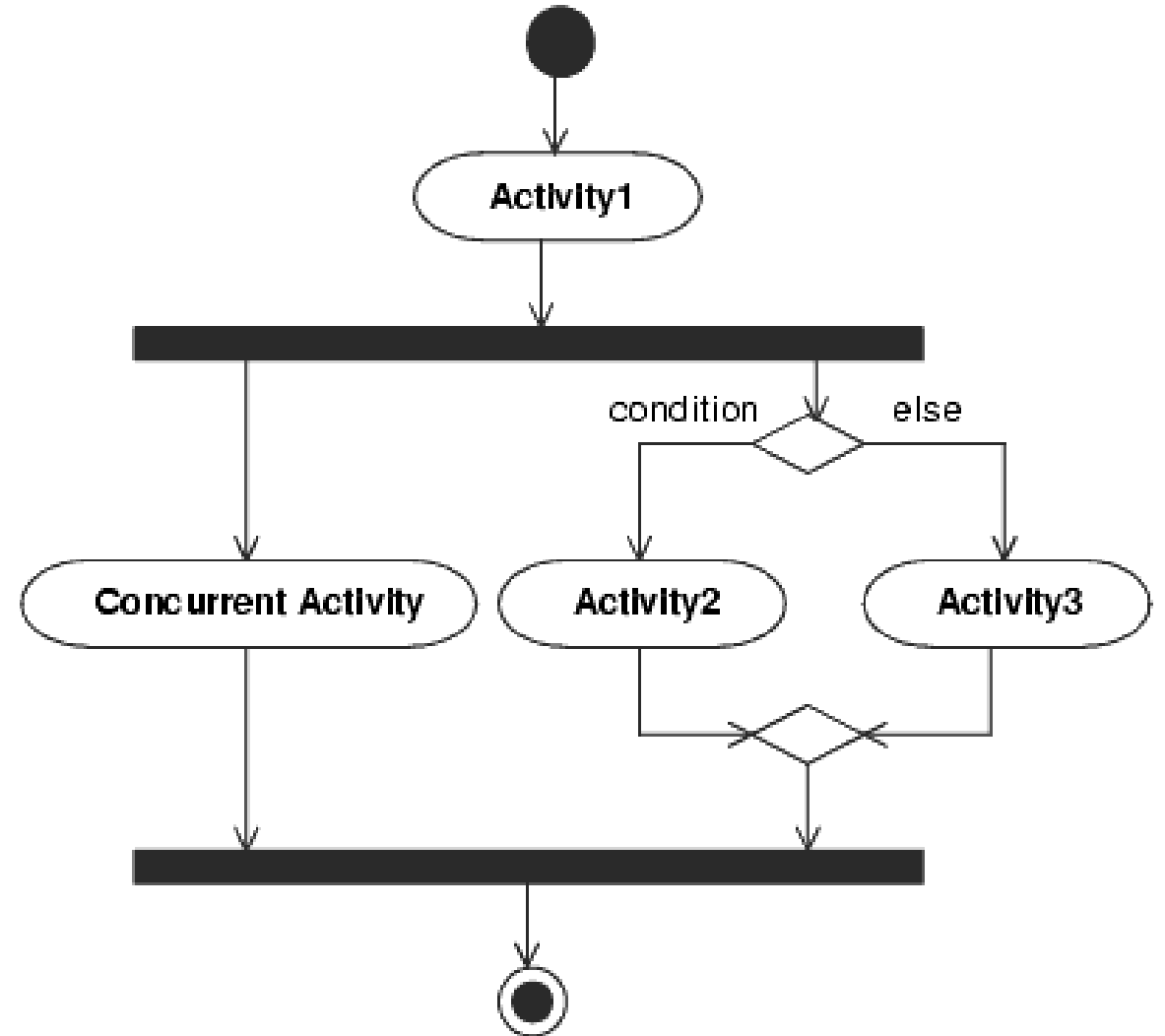
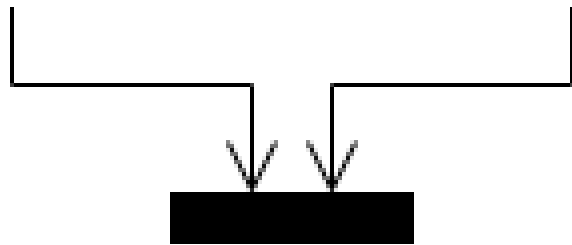
- A **fork** node has one incoming edge and numerous outgoing edges.
- It is similar to one too many decision parameters.
- When data arrives at an incoming edge, it is duplicated and split across numerous outgoing edges simultaneously.
- A **join** node is opposite of a fork node as it has many incoming edges and a single outgoing edge.
- It performs logical AND operation on all the incoming edges. This helps you to synchronize the input flow across a single output edge.

# Fork and Join nodes

- Fork



- Join



# **MODULE-IV**

# **ADVANCED BEHAVIORAL MODELING**

- Events and signals,
- state machines,
- processes and threads,
- time and space,
- state chart and state chart diagrams.
- Case study: The next gen POS system

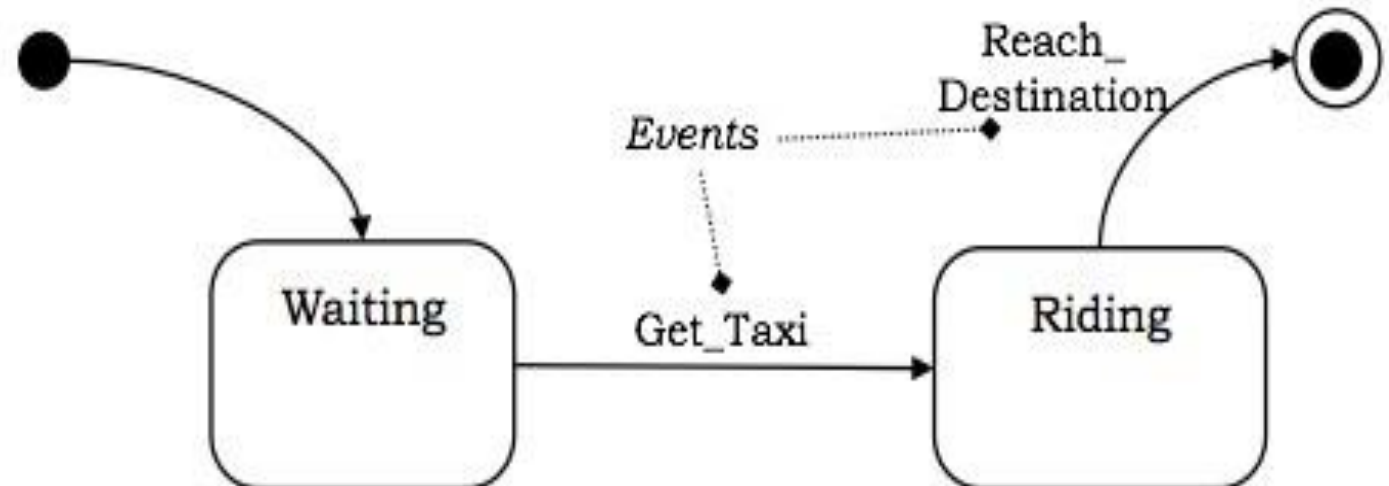
- In the real world, things happen, often simultaneously and unpredictably. "Things that happen" are called 'events'.
- Events include signals, calls, the passage of time, or a change in state.
- Events may be synchronous or asynchronous.

## Kinds of events

- Events may be external or internal. External events are those that pass between the system and its actors.
- Internal events are those which pass among the objects that reside within the system.
- There are four kinds of events: signals, calls, the passing of time, and a change in state,

# Example

- The transition from Waiting state to Riding state takes place when the person gets a taxi.
- Likewise, the final state is reached, when he reaches the destination.
- These two occurrences can be termed as events Get\_Taxi and Reach\_Destination.





# Types of Events

1. Signals
2. Calls
3. Time Event
4. Change of state event

# External and Internal Events



- External events are those events that pass from a user of the system to the objects within the system.
- ***For example,*** mouse click or key–press by the user are external events.
- Internal events are those that pass from one object to another object within a system.
- ***For example,*** stack overflow, a divide error, etc.

- **Signals**
- **State machines**
- **Processes and threads**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Signals</b></li><li>• <b>State machines</b></li><li>• <b>Processes and threads</b></li></ul>	<ul style="list-style-type: none"><li>• Identify the importance of Signals in UML diagrams</li><li>• State machines</li><li>• Processes and threads</li></ul>	<ul style="list-style-type: none"><li>• Identify the importance of Signals in UML diagrams</li><li>• State machines</li><li>• Processes and threads</li></ul>

# Outcome achieved



## Name of the topic:

<b>Students will be able to:</b>	
1	Describe the role of Signals while modeling UML diagrams
2	Identify the importance of State machines
3	Describe Processes and threads in UML

- In UML models, *signals* are model elements.
- Signals specify one-way, asynchronous communications between active objects.
- **A signal represents an object that is dispatched (thrown) asynchronously by one object and then received (caught) by another.**
- Exceptions are an example of a kind of signal.

- Signals may have attributes and these attributes of a signal serve as its parameters.
- Signals may be involved in generalization relationships, enabling the modeling of hierarchies of events.
- The execution of an operation can also send signals.
- When modeling a class or an interface, an important part of specifying the behavior of the element is specifying the signals that can be sent by its operations.

- You can add signals to the class diagrams in your model to represent the following functions:
  - *Trigger* events in objects in models that represent event-driven systems
  - Exceptions thrown by an operation when something unexpected occurs in a software system
- A signal has a name describing its purpose in the system.



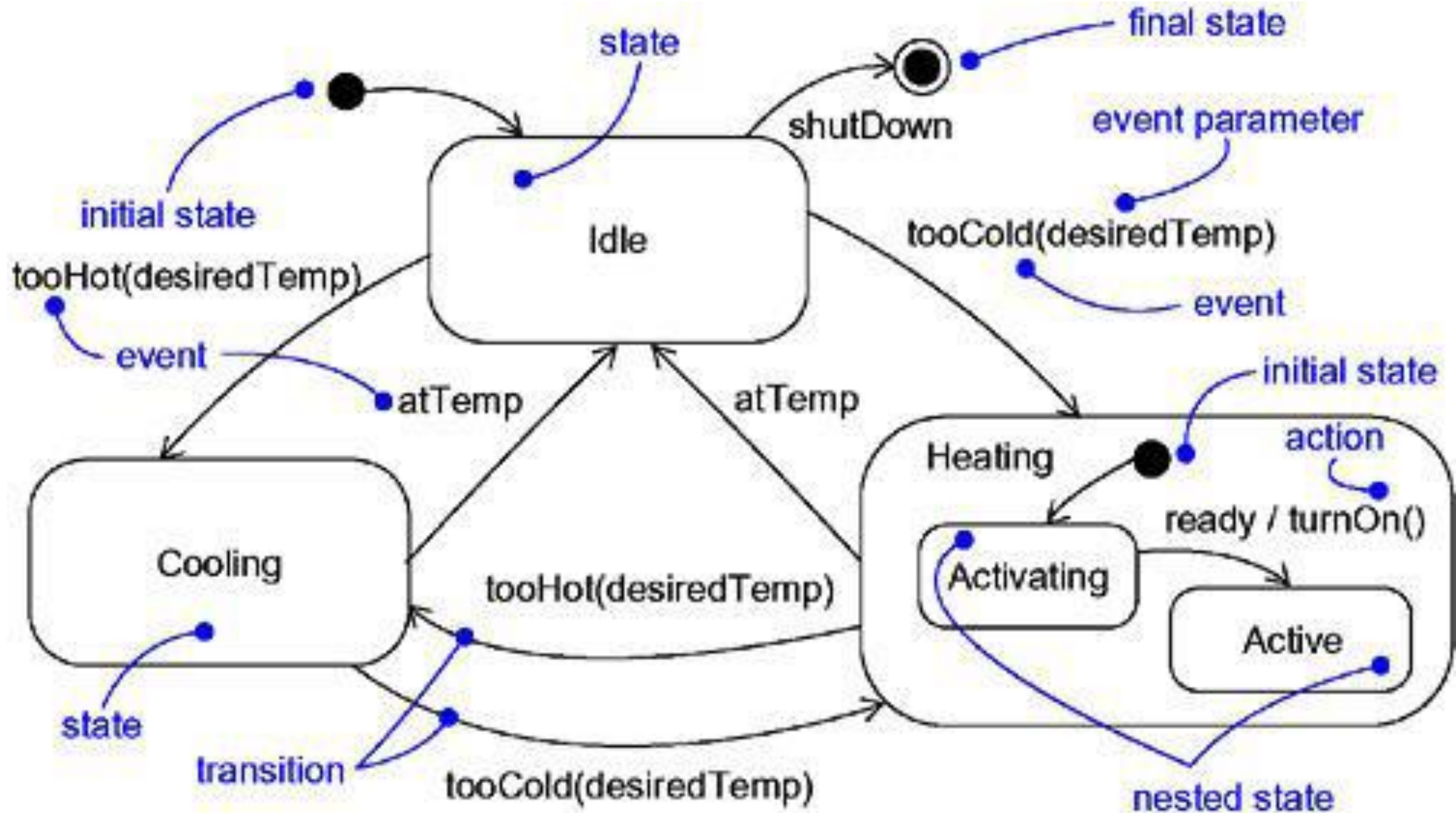
- As the following figure illustrates, the UML notation for a signal is a rectangle with two compartments.
- The upper compartment contains the «signal» keyword and the signal's name.
- The lower compartment contains the signal's attributes.



## Terms and Concepts

- A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- A *state* is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
- An object remains in a state for a finite amount of time.

# Example



# Common Modeling Techniques :

## Modeling the Lifetime of an Object



- Starting from the initial state to the final state, lay out the top-level states the object may begin.
- Connect these states with transitions triggered by the appropriate events. Continue by adding actions to these transitions.
- Identify any entry or exit actions.
- Expand these states as necessary by using sub-states.

- Check that all actions mentioned in the state machine are sustained by the relationships, methods, and operations of the enclosing object.
- After rearranging your state machine, check it against expected sequences again to ensure that you have not changed the object's semantics.

## Transitions

- A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.

## Action

- An action is an executable atomic computation. Actions may include operation calls the creation or destruction of another object, or the sending of a signal to an object.

# Processes and Threads



- A **process** is a heavyweight flow that can execute concurrently with other processes.
- A **thread** is a lightweight flow that can execute concurrently with other threads within the same process.
- An active object is an object that owns a process or thread and can initiate control activity.
- Processes and threads are rendered as stereotyped active classes.
- An active class is a class whose instances are active objects.

## Classes and Events

- Active classes are just classes which represents an independent flow of control
- Active classes share the same properties as all other classes.
- When an active object is created, the associated flow of control is started; when the active object is destroyed, the associated flow of control is terminated

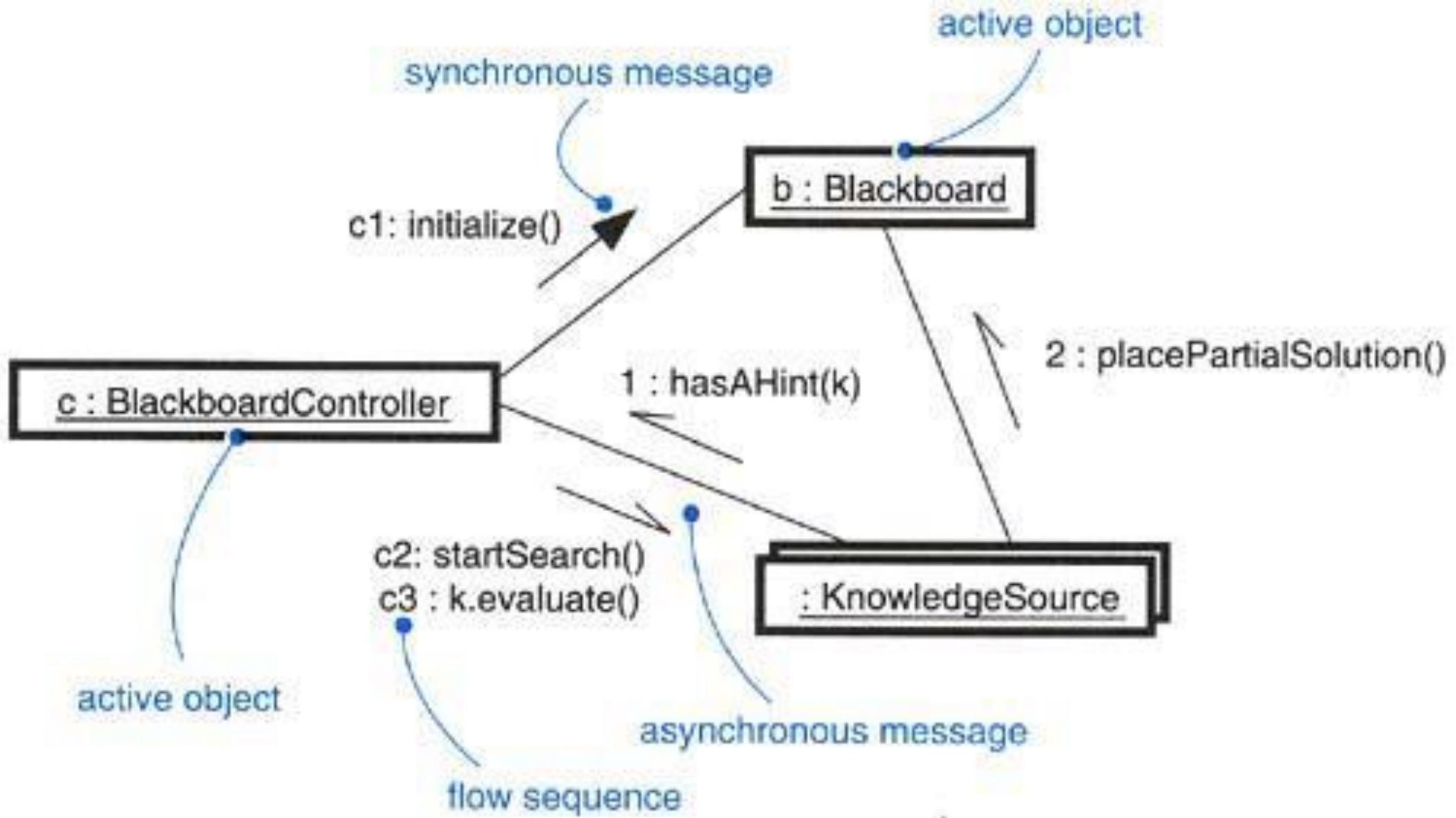


- Two standard stereotypes that apply to active classes are,  
**<<process>>** – Specifies a heavyweight flow that can execute concurrently with other processes.
- **<<thread>>** – Specifies a lightweight flow that can execute concurrently with other threads within the same process

- In a sequential system, there is a single flow of control. i.e., one thing, and one thing only, can take place at a time.
- In a concurrent system, there is multiple simultaneous flow of control i.e., more than one thing can take place at a time.

- In a system with both active and passive objects, there are **FOUR** possible combinations of interaction.
  1. A message may be passed from one **passive object** to another.
  2. A message may be passed from one **active object** to another.
  3. A message may be passed from an **active object** to a passive object.
  4. A message may be passed from a **passive object** to an active one.

- In **inter-process communication** there are two possible styles of communication.
  1. **Active object** might **synchronously** call an operation of another object.
  2. **Active object** might **asynchronously** call an operation of another object.
- A **synchronous message** is rendered as a **full arrow** and an **asynchronous message** is rendered as a **half arrow**.



- **Time and Space and its Common modeling Techniques with examples**

# Course outcome / Topic learning outcome



## List the course outcome / Topic outcome

<b>Name of the Topic covered</b>	<b>Topic Learning Outcome</b>	<b>Course Outcome</b>
<ul style="list-style-type: none"><li>• <b>Time and Space</b></li></ul>	<ul style="list-style-type: none"><li>• <b>Categorize</b> advanced behavioral modeling for visualizing flow control of objects and activities of specified case study like next gen POS system</li></ul>	<ul style="list-style-type: none"><li>• <b>Categorize</b> advanced behavioral modeling for visualizing flow control of objects and activities of specified case study like next gen POS system</li></ul>

# Outcome achieved



## Name of the topic:

### Students will be able to:

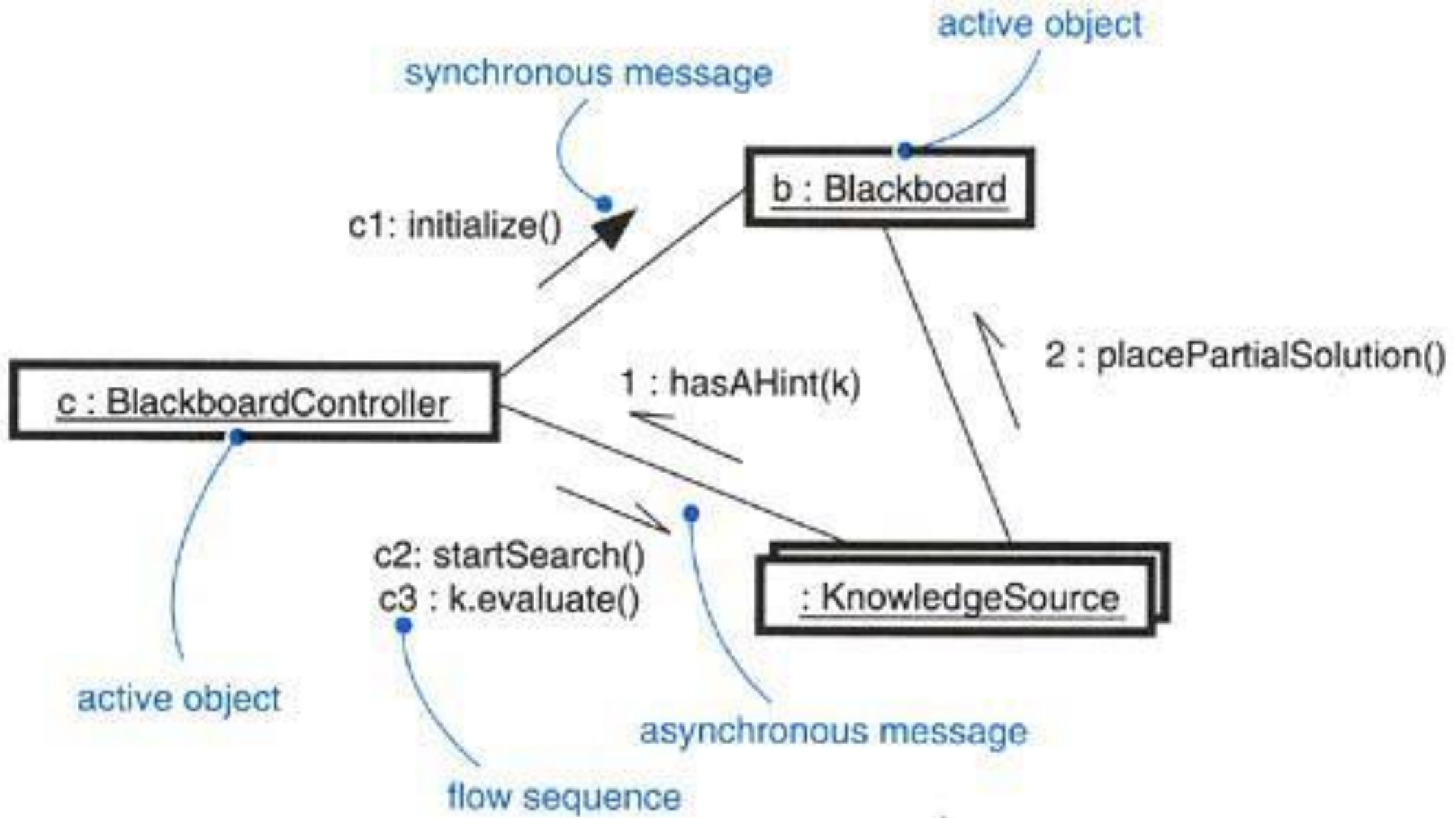
1

**Categorize** advanced behavioral modeling for visualizing flow control of objects and activities of specified case study like next gen POS system



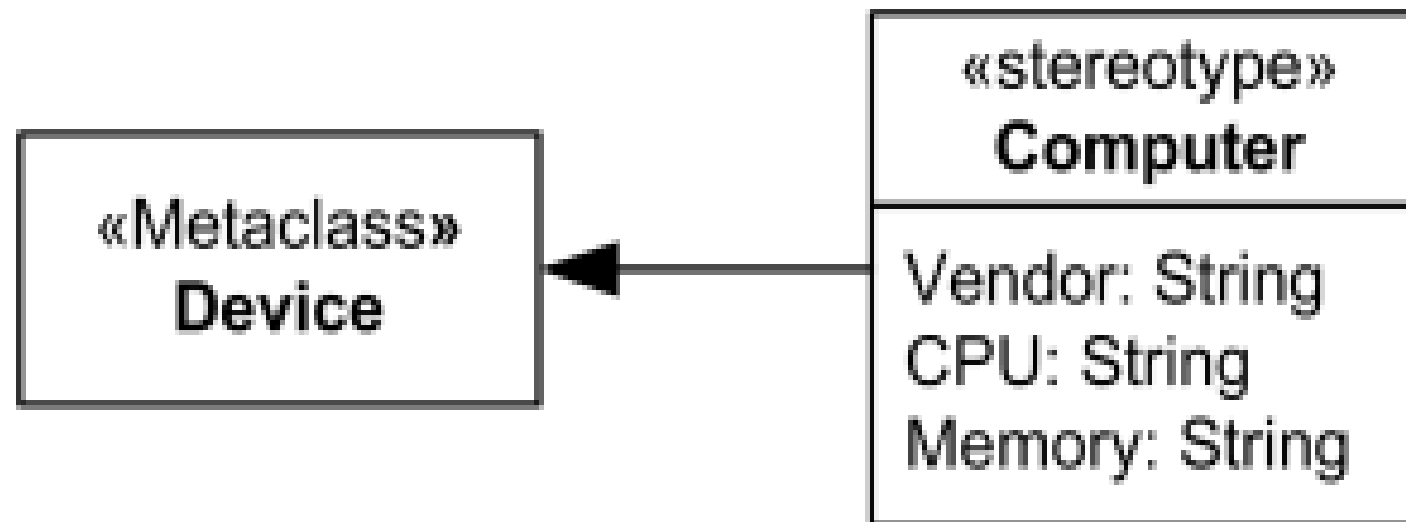
- In a system with both active and passive objects, there are **FOUR** possible combinations of interaction.
  1. A message may be passed from one **passive object** to another.
  2. A message may be passed from one **active object** to another.
  3. A message may be passed from an **active object** to a passive object.
  4. A message may be passed from a **passive object** to an active one.

- In **inter-process communication** there are two possible styles of communication.
  1. **Active object** might **synchronously** call an operation of another object.
  2. **Active object** might **asynchronously** call an operation of another object.
- A **synchronous message** is rendered as a **full arrow** and an **asynchronous message** is rendered as a **half arrow**.



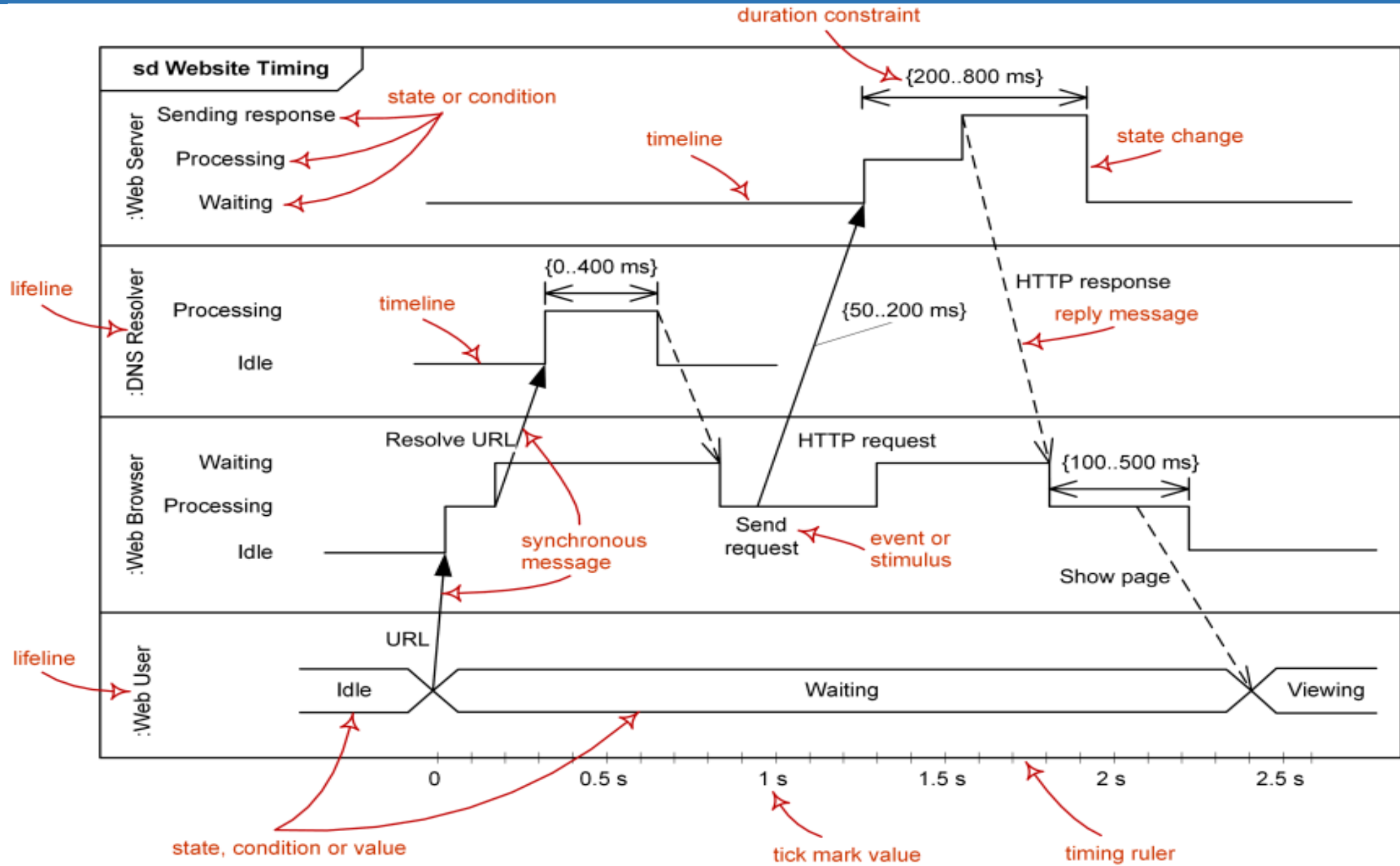
- A ***time expression*** is an expression that **evaluates** to an **absolute** or **relative value of time**
- A ***timing constraint*** is a **semantic statement** about the relative or absolute value of time
- *Location* is the placement of a component on a node. Graphically, location is rendered as a tagged value.

- **stereotype** is a class, it may have properties.
- Properties of a **stereotype** are referred to as **tag definitions**.
- When a **stereotype** is applied to a model element, the values of the properties are referred to as tagged values.

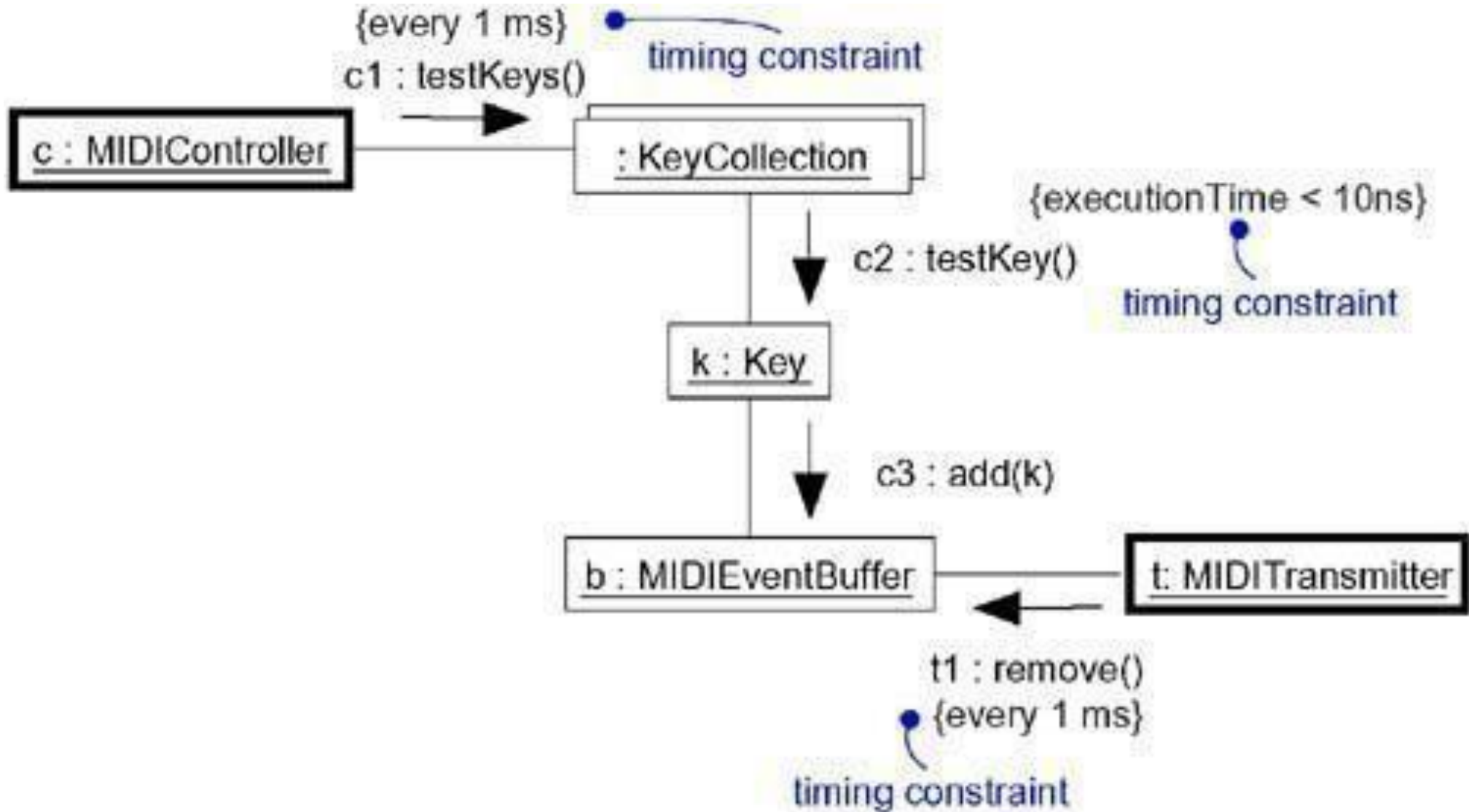


- Real time systems are, by their very name, time-critical systems.
- Events may happen at regular or irregular times; the response to an event must happen at **predictable absolute times** or at **predictable times** relative to the event itself.
- The passing of messages represents the dynamic aspect of any system, so when you model the time critical nature of a system with the UML, you can give a name to each message in an interaction to be used as a timing mark.

# Ex:



# Ex:





# State Chart Diagram



- State chart diagram shows a state machine, consisting of states, transitions, events and activities
- It address the dynamic view of a system
- Especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive system.

- A *statechart diagram* shows a state machine, emphasizing the flow of control from state to state.
- A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- A *state* is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event.

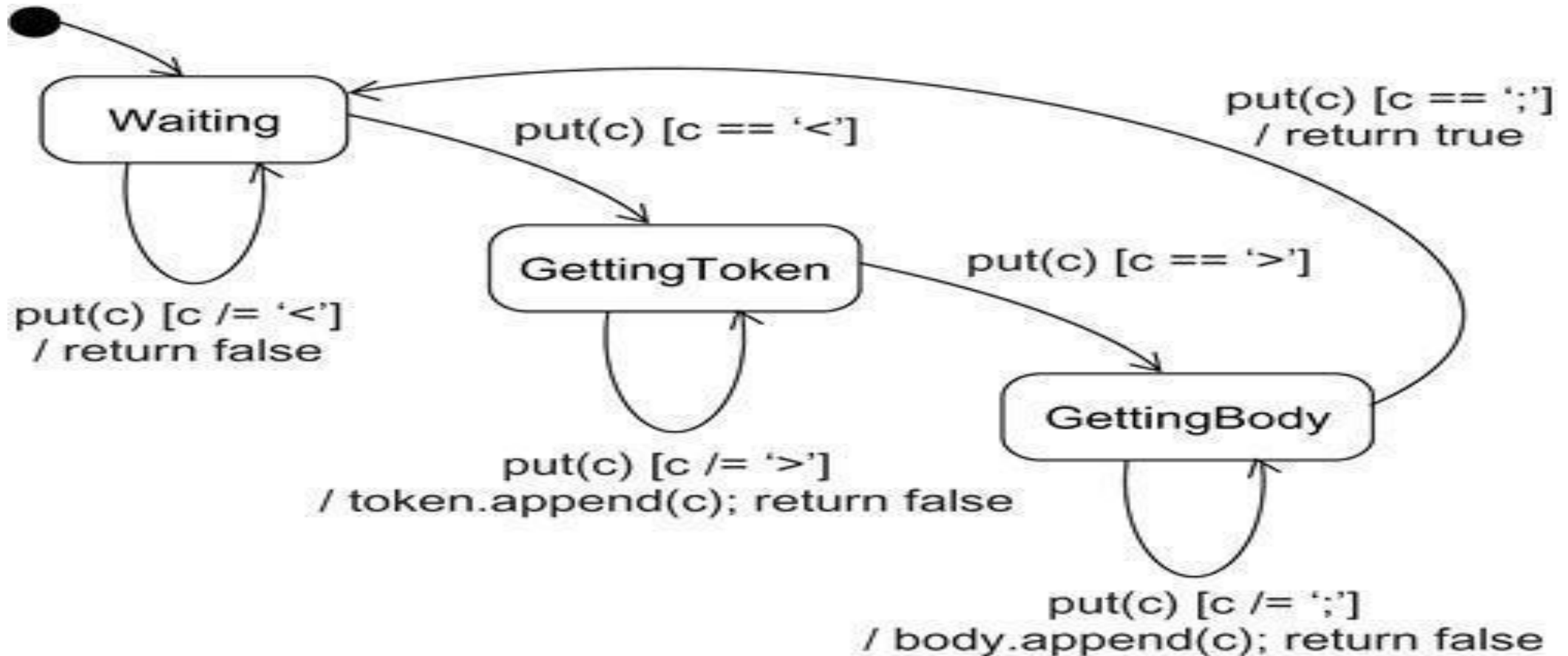
- A statechart diagram will use to model some dynamic aspect of a system.
- It is also used in the context of virtually any modeling element.
- Typically, use statechart diagrams in the context of the system as a whole, a subsystem, or a class.
- It can also be used to attach a statechart diagrams to use cases.

- **To model a reactive object,** Choose the context for the state machine, whether it is a class, a use case, or the system as a whole.
- Choose the initial and final states for the object.
- To guide the rest of your model, possibly state the pre- and post conditions of the initial and final states, respectively.

- Decide on the stable states of the object by considering the conditions in which the object may exist for some identifiable period of time.
- Start with the high-level states of the object and only then consider its possible sub-states.

- Decide on the meaningful partial ordering of stable states over the lifetime of the object.
- Decide on the events that may trigger a transition from state to state.
- Model these events as triggers to transitions that move from one states to another.
- Attach actions to these transitions (as in a Mealy machine) and/or to these states (as in a Moore machine).

- Consider ways to simplify your machine by using sub-states, branches, forks, joins, and history states.



***Forward engineering*** (the creation of code from a model) is possible for statechart diagrams, especially if the context of the diagram is a class.

- The forward engineering tool must generate the necessary private attributes and final static constants.

***Reverse engineering*** (the creation of a model from code) is theoretically possible, but practically not very useful.

- The choice of what constitutes a meaningful state is in the eye of the designer.



- Reverse engineering tools have no capacity for abstraction and therefore cannot automatically produce meaningful statechart diagrams.

## **A well-structured statechart diagram**

- It is focused on communicating one aspect of a system's dynamics.
- Contains only those elements that are essential to understanding that aspect.
- Provides detail consistent with its level of abstraction (expose only those features that are essential to understanding).
- Uses a balance between the styles of Mealy and Moore machines.

- **When you draw a statechart diagram**

- Give it a name that communicates its purpose.
- Start with modeling the stable states of the object, then follow with modeling the legal transitions from state to state.
- Address branching, concurrency, and object flow as secondary considerations, possibly in separate diagrams.
- Lay out its elements to minimize lines that cross.

# Case study: The next gen POS system



- A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store.
- It includes hardware components such as a computer and bar code scanner, and software to run the system.
- It interfaces to various service applications, such as a thirdparty tax calculator and inventory control.
- These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).

# Case study: The next gen POS system



- A POS system increasingly must support multiple and varied client-side terminals and interfaces.
- These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.
- Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added.
- Therefore, will need a mechanism to provide this flexibility and customization.

# Case study: The next gen POS system

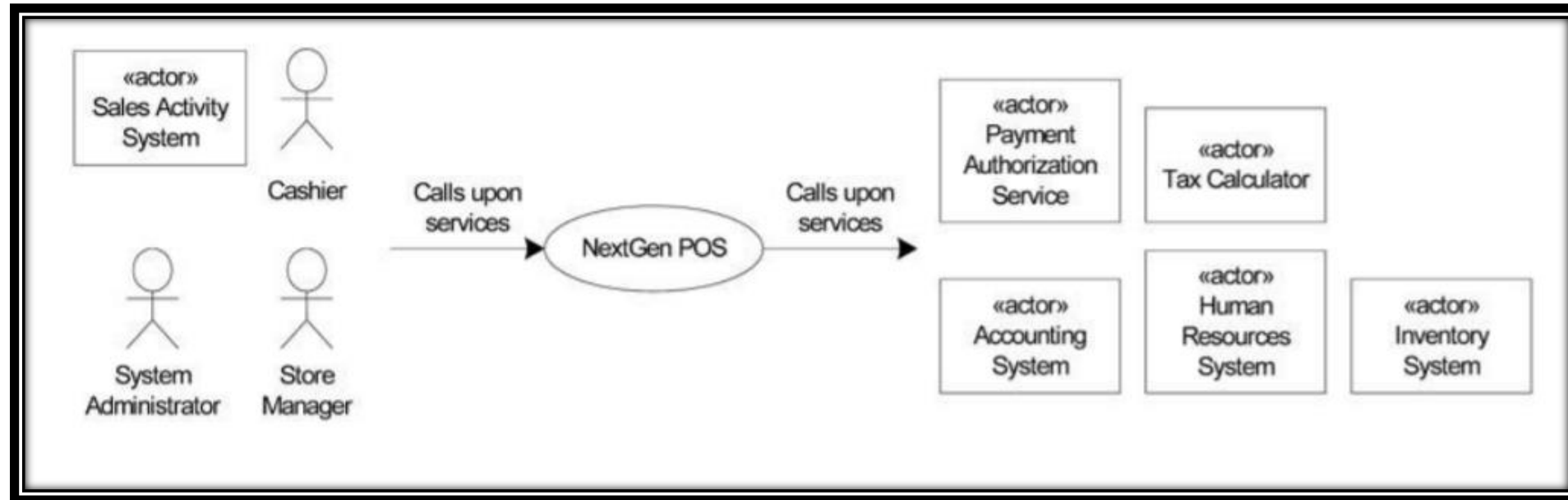


## User-Level Goals

- The users (and external systems) need a system to fulfill these goals:
- Cashier: process sales, handle returns, cash in, cash out
- System administrator: manage users, manage security, manage system tables
- Manager: start up, shut down
- Sales activity system: analyze sales data .....

# Case study: The next gen POS system

## User-Level Goals



## QUESTIONS

- What is Statechart diagram?
- When to draw a Statechart diagram?
- What are the basic components of a Statechart diagram?
- What are the common modeling techniques of Statechart diagram?
- Define the case study: The next gen POS system



Thank  
you

