Object Oriented Analysis and Design with Java

Dr. Kamatchi Priya L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Object Oriented Analysis and Design with Java

Overview

Dr. Kamatchi Priya L

Department of Computer Science and Engineering

What we Know?

Traditional SDLC

- We have seen in Software Engineering course, the design step falls between understanding your requirements and building the product.

Structured Systems Analysis & Design(SSAD)

Structured Systems Analysis & Design (SSAD) is a framework of activities and tasks that need to be accomplished to develop an information system. This approach is also known as top-down design, modular programming, and stepwise refinement.

Object Oriented Analysis and Design with Java

What will we study?

What is OOAD?

One approach to help make the design process easier is the object-oriented(OO) approach. This allows for the description of **concepts** in the problem and solution spaces as **objects.**

What is Object Oriented Approach?

- In software development lifecycle we can apply and implement OO concepts by following three steps.

OO Analysis --> OO Design --> OO implementation by using OO languages(Java)

Object Oriented Analysis and Design with Java

Context of Object-oriented Approach

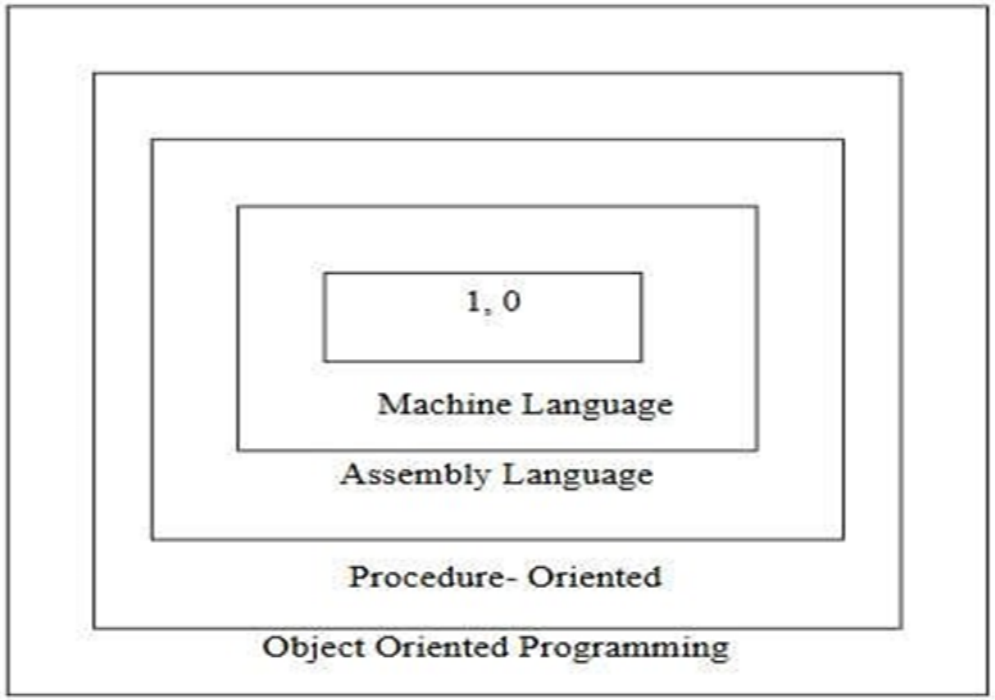
Software Crisis

- Developments in software technology continue to be dynamic.
- New tools and techniques are announced in quick succession.
- This has forced the software engineers and industry to continuously look for new approaches to software design and development.
- These rapid advances appear to have created a situation of crisis within the industry

The following issues need to be addressed to face the crisis:

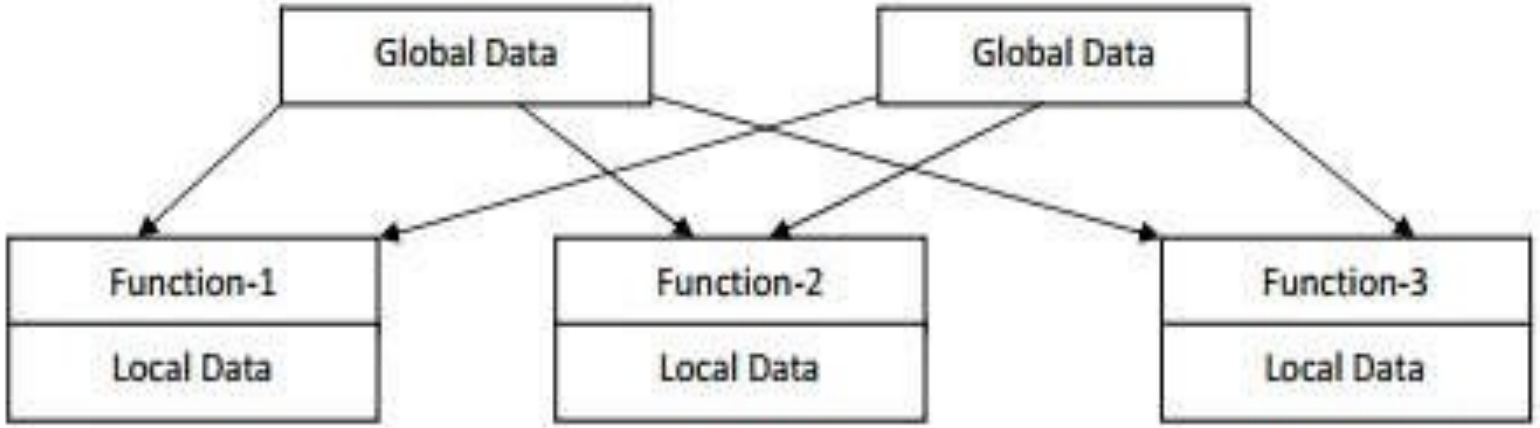
- How to represent real-life entities of problems in system design?
- How to design system with open interfaces?
- How to improve the quality of software?
- How to ensure reusability and extensibility of modules?
- How to develop modules that are tolerant of any changes in future?
- How to improve software productivity and decrease software cost?

Software Evolution concepts: Programming Knowledge



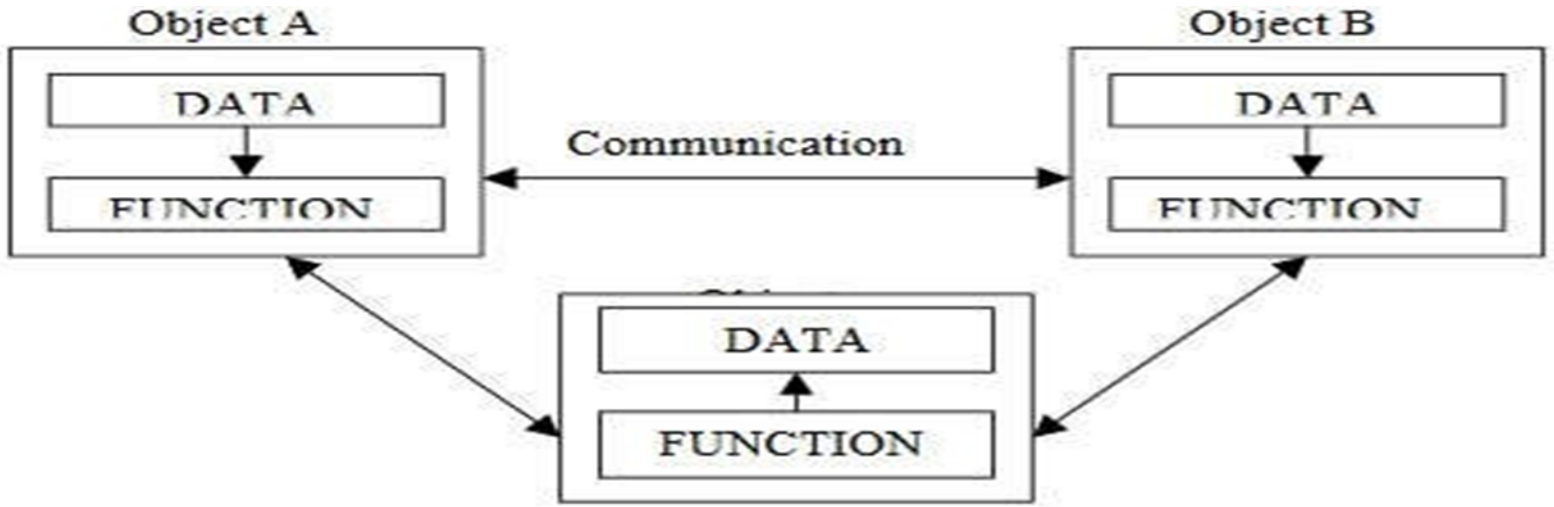
Procedure-oriented programming

- Conventional programming, using high level languages such as COBOL, FORTRAN and C, is commonly known as procedure- oriented programming (POP) .
- In the procedure-oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks .
- The primary focus is on **functions** :



Object-oriented Programming

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data may be hidden and may not be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design



Object-based Programming

Object-oriented Programming Language	Object-based Programming Language
All the characteristics and features of object-oriented programming are supported.	All characteristics and features of object-oriented programming, such as inheritance and polymorphism are not supported.
These types of programming languages don't have a built-in object. Example: C++.	These types of programming languages have built-in objects. Example: JavaScript has a window object.
Java is an example of object-oriented programing language which supports creating and inheriting (which is reusing of code) one class from another.	VB is another example of object-based language as you can create and use classes and objects, but inheriting classes is not supported.

Object Oriented Analysis and Design with Java

Syllabus

Unit 1: Advanced OO, Object Oriented Analysis and Static Models and Diagrams

Unit 2: Object Orientated Programming and Architecture design

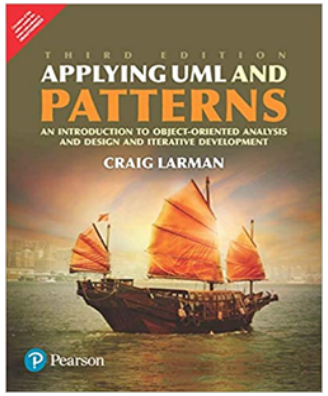
Unit 3: Design principles

Unit 4: OO Design Patterns & Anti-Patterns

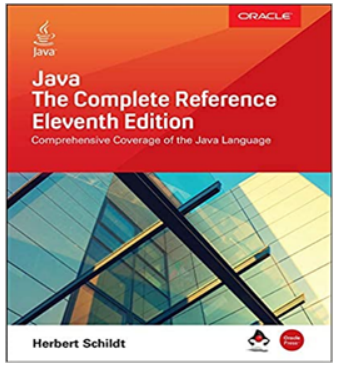
Object Oriented Analysis and Design with Java

Text Books

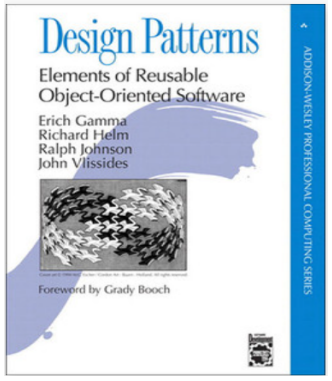
T1: "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development", by Craig Larman, 3rd Edition, Pearson 2015.



T2:"Java the Complete Reference", Herbert Schildt ,McGraw-Hill ,11th Edition, 2018.



T3. "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, 1st Edition, Pearson 2015



PES UNIVERSITY
CELEBRATING 50 YEARS

Reference Books

R1. “Head First Design Patterns”, Freeman, Eric,and Elisabeth Robson, O'Reilly Media, 2020.

R2. "Object-Oriented Software Engineering: Practical Software Development using UML and Java", Timothy C. Lethbridge, Robert Laganière, Second edition, Mc Gram Hill,2016

R3. “Object-Oriented Modelling and Design with UML”, Michael R Blaha and James R Rumbaugh, 2nd Edition, Pearson 2007.

Evaluation Policy

Evaluation Component		Marks
ISA	ISAs – CBT	30
	Lab Assignment (10 Nos.)	10
	Hands on Assignment on MVC Framework using Spring	2
	Mini Project	8
	Total	**50**
ESA	**Pen and Paper Mode (100 marks)**	**50**

THANK YOU

Dr. Kamatchi Priya L
Department of Computer Science and Engineering

priyal@pes.edu

Object Oriented Analysis and Design using Java

Prof. Vinay Joshi
Department of Computer Science and Engineering

Requirement Engineering

RE is usually the first step in any software intensive development lifecycle irrespective of model

- Usually difficult, error prone and costly
- Critical for successful development of all downstream activities
- Errors introduced during requirements phase if not handled properly will propagate into the subsequent phases
- Unnecessary, Late or invalid requirements can make the system cumbersome or slip
- Requirement errors are expensive to fix at a later stage

Object Oriented Analysis and Design using Java

Requirement Engineering

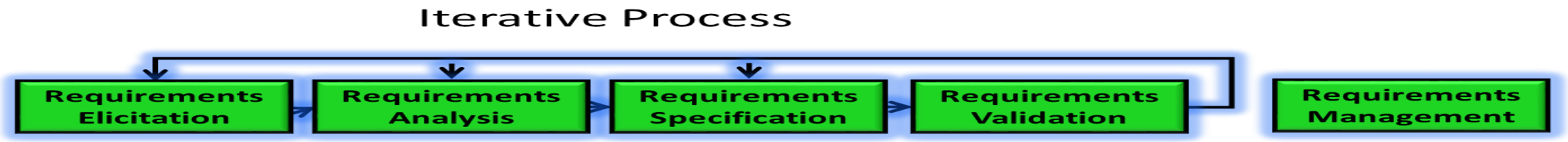
Requirement Engineering

- Requirement is the Property which must be exhibited by software developed/adapted to solve a particular problem
- Requirement should specify the externally visible behavior of **what** and **not how**
- Requirements can be looked at as
  - Individual requirements
  - Set of requirements

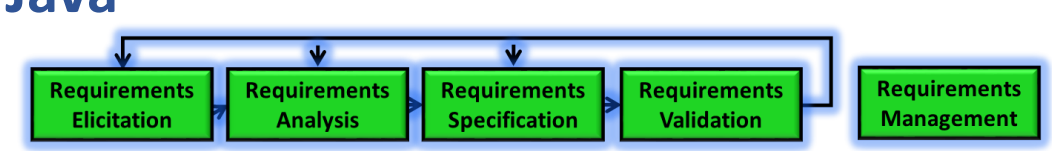
Requirement Engineering

Is a process of working proactively with all stakeholders gathering their needs, articulating their problem, identify and negotiate potential conflicts thereby **establishing a clear scope and boundary for a project.**

It can also be described as a process of ensuring that the stakeholders have been identified and they have been given an **opportunity to explain their problem and needs** and describe what they would like the new system to do.

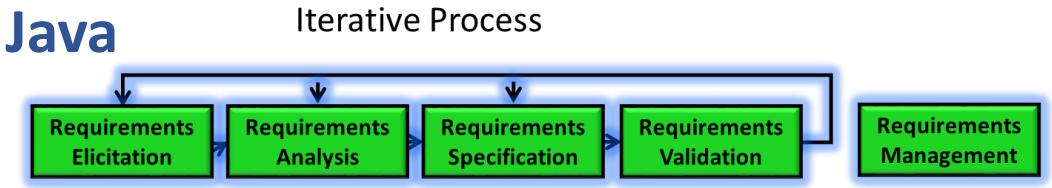


Requirement Analysis Process



1. **Understand the requirements in depth, both from a product and process perspective**
2. **Classify and Organize the requirements into coherent clusters**
   - **Functional, Non-Functional & Domain requirements**
   - **System and User Requirements**
3. **Model the requirements**
4. **Analyze the requirements (if necessary) using fish bone diagram**

Requirement Analysis Process



5. Recognize and resolve conflicts (e.g., functionality v. cost v. timeliness)
6. Negotiate Requirements
7. Prioritize the requirements (MoSCoW -Must have, Should have, Could have, Wont have)
8. Identify risks if any
9. Decide on Build or Buy (Commercial Of The Shelf Solution) and refine requirements

What is UML

Unified Modeling Language

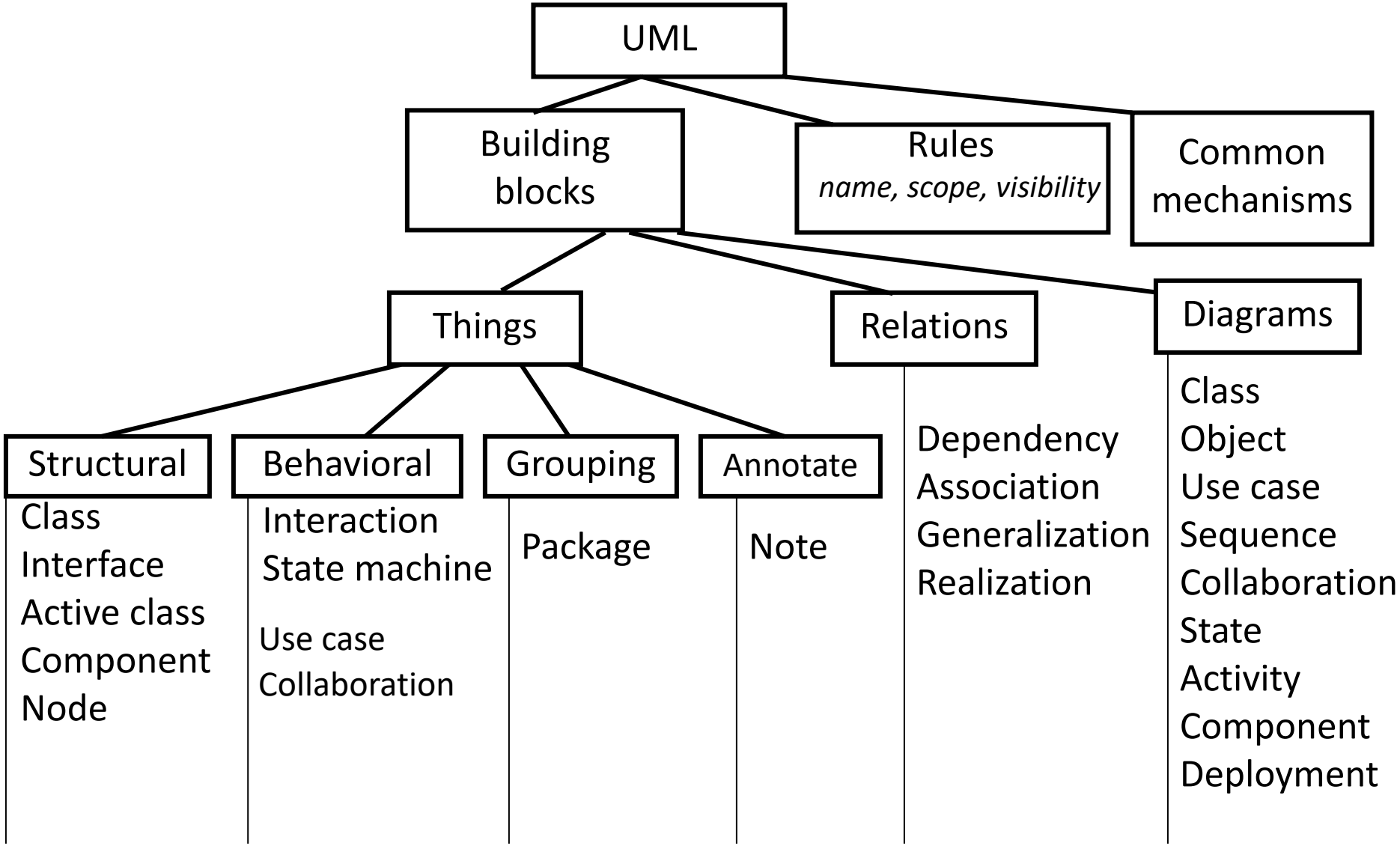
- Language to express different types of models
- Language defines
- Syntax – Symbols and rules for using them
- Semantics – What these symbols represent

UML can be used to

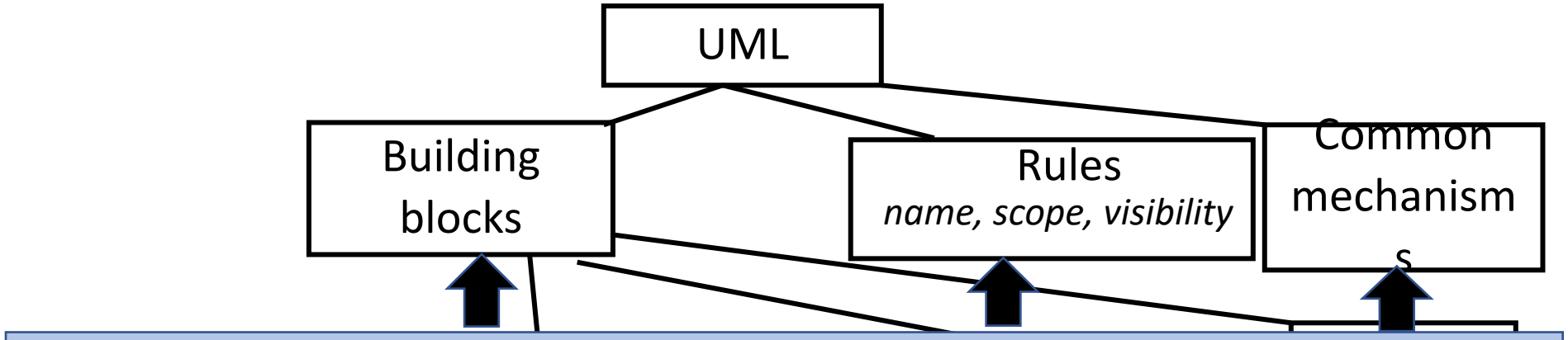
- Visualize (Graphical Notation)
- Specify (Complete and Unambiguous)
- Construct (Code Generation)
- Document (Design, Architecture, etc.)

the artifacts of software-intensive systems

Conceptual Models of UML



Conceptual Models of UML

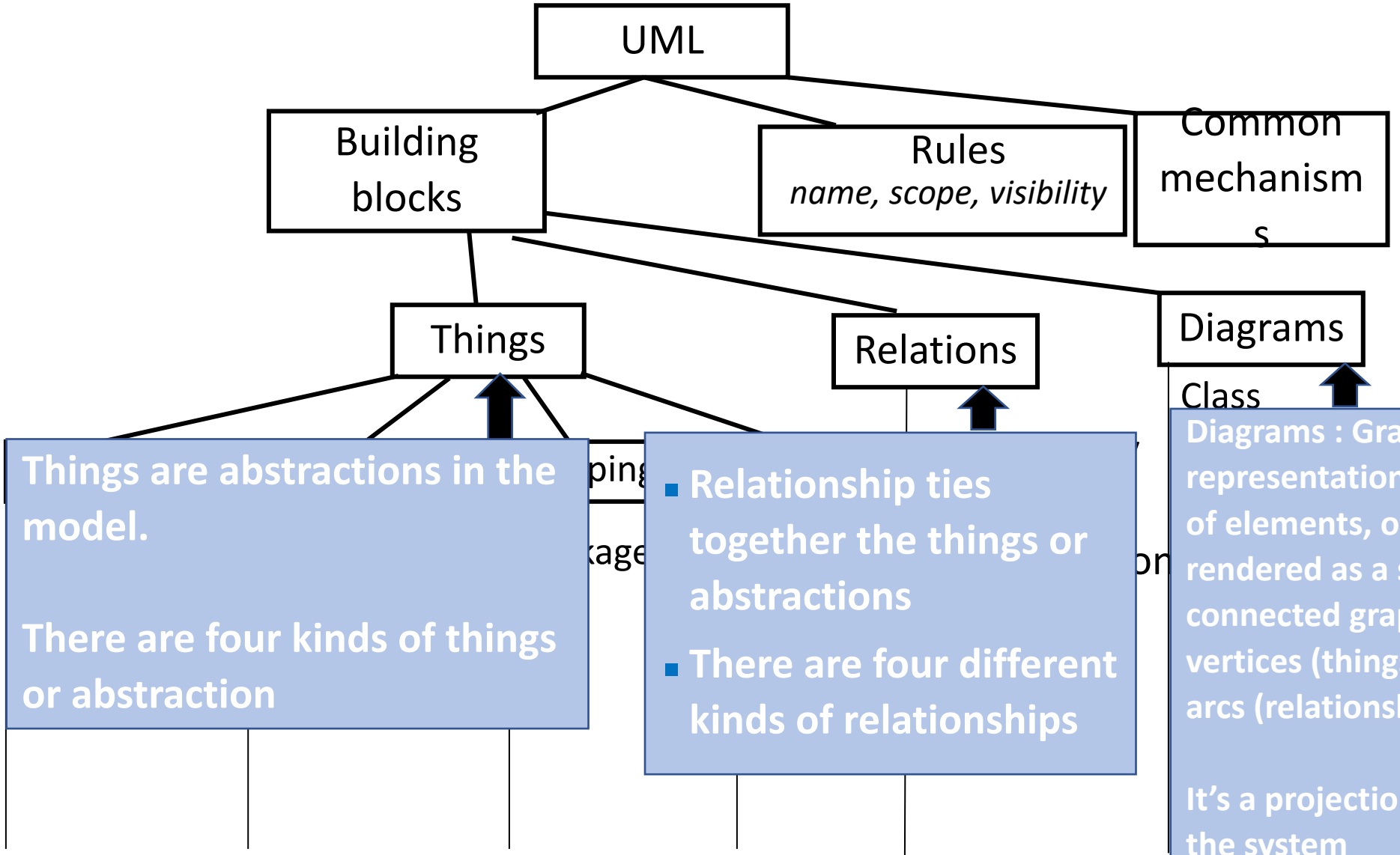


UML is made up of three conceptual elements. They are

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

Interface Active class Component Node	State machine Use case Collaboration			Realization	Collaboration State Activity Component Deployment

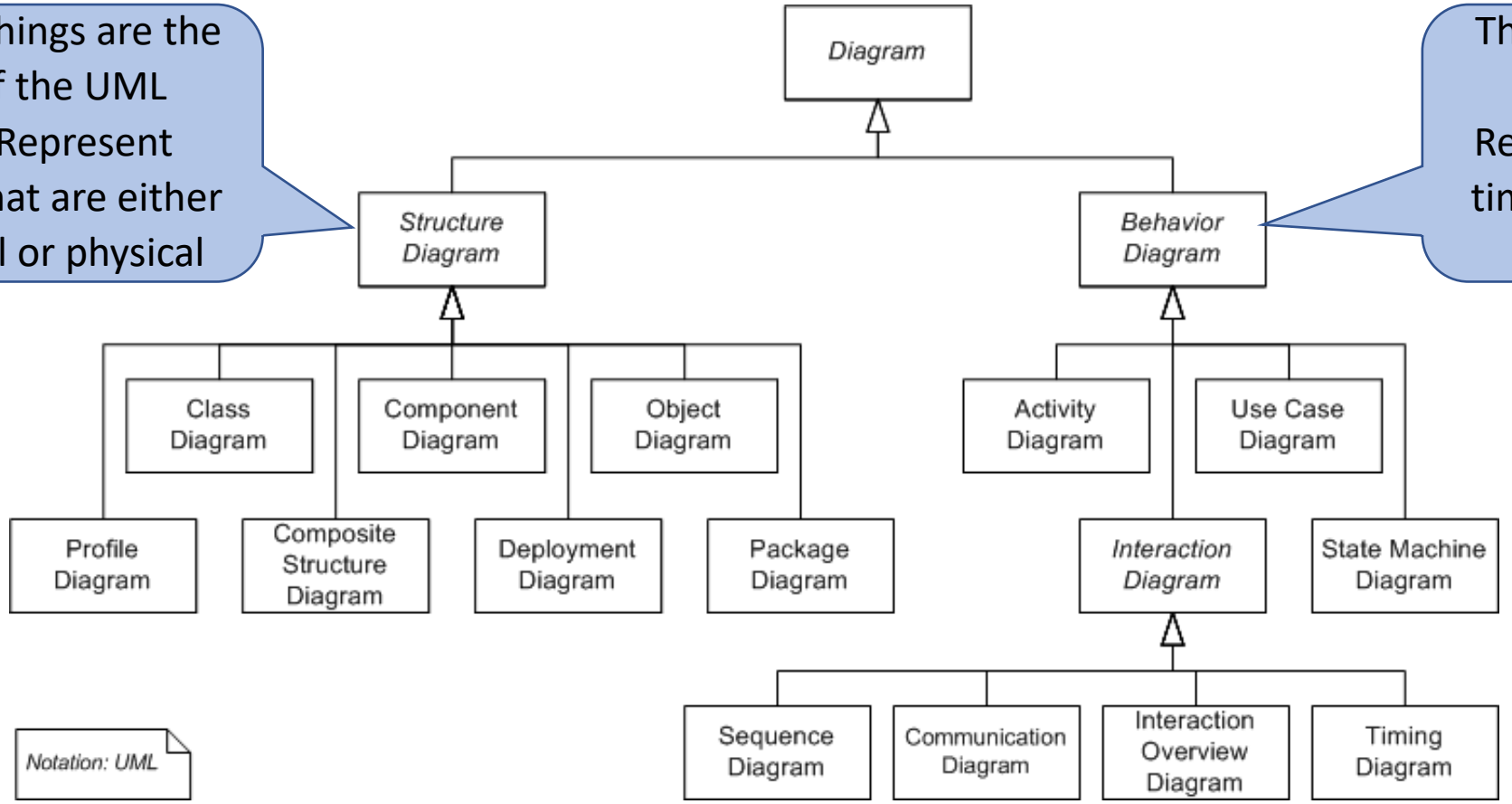
Conceptual Models of UML



Object Oriented Analysis and Design using Java

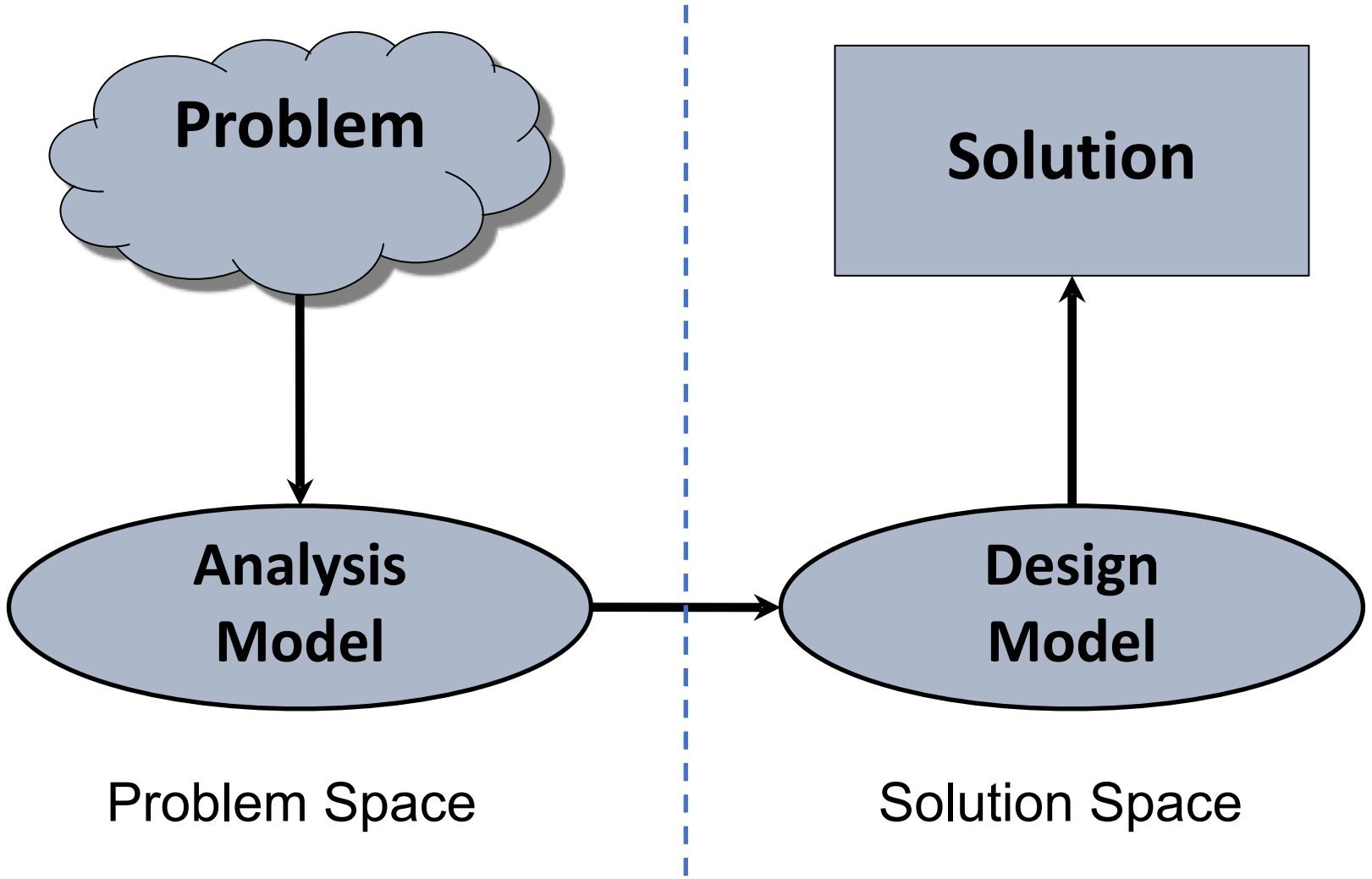
UML Diagrams

Structural things are the nouns of the UML models. Represent elements that are either conceptual or physical

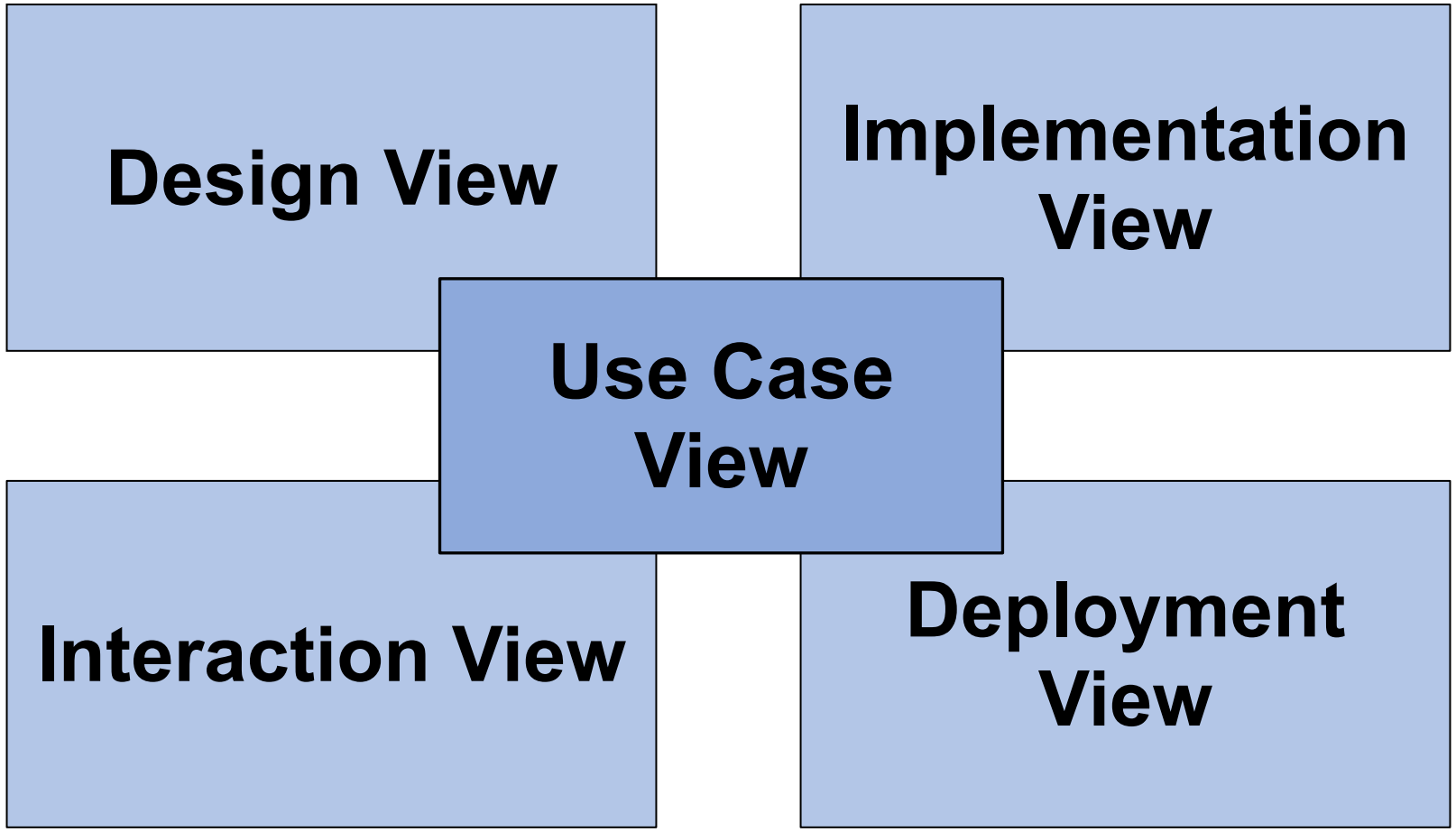


These are dynamic parts of the UML models. Represent behavior over time and space. Typically verbs of the model

Models in Solution Development



Views of a Solution System



Views of a Solution System

- Use Case View
  - Describes behavior of system as seen by end users, analysts, and testers
- Design View
  - Describes Classes, Interfaces, and Collaborations that form the vocabulary of the problem and its solution
- Interaction View
  - Describes flow of control among its various parts
  - Address performance, scalability, and throughput

Object Oriented Analysis and Design using Java

UML Diagrams –Summary

- State Diagram
  - Shows states, transitions, events, and activities depicting the dynamic view of internal object states
- Sequence Diagram
  - Shows interactions between objects as a time-ordered view
- Collaboration Diagram
  - Similar to sequence diagram, but in a spatial view, using numbering to indicate time order
- Deployment Diagram
  - Shows the configuration of run-time processing nodes and the components that are deployed on them

THANK YOU

Vinay Joshi
Department of Computer Science and Engineering

vinayj@pes.edu

Object Oriented Analysis and Design Using Java

Prof. Vinay Joshi
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Object Oriented Analysis and Design using Java

Use case Modelling: Use case Diagrams

Prof. Vinay Joshi
Department of Computer Science and Engineering

Use case Modelling - Agenda

- Introduction To Use case Modelling
- Use case diagrams
- Use case
- Actor
- Use case description
- Example use case diagrams
- Relation between these use case and Actor
- Relation between the use cases
- Use case specification
- Practice: ATM
- Few inputs

Use case Modelling: Introduction

- Describes the **interaction of users and the system**
- Describes **what functionality does a system provides to its users**.
- Use case model has **two important elements** - **actors** and **use cases**.
- **Actor/s:** One or set of objects who directly interacts with the system
  - Every actor has a defined purpose while interacting with the system.
  - An actor can be a person, device or another system.
- **Use case:** A piece of functionality that a system offers to its users.
  - Set of all use cases defines the entire functionality of the system.
  - Also define the error conditions that may occur while interacting with the system

Use case Diagrams

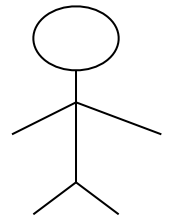
- Incorporates both **actor and use cases and also the relationship** between them in the **graphical representation**.
- Used to visualize, specify, construct, and document the (intended) behavior of the system, during requirements capture and analysis.
- Provide a way for developers, domain experts and end-users to Communicate.
- Serve as basis for testing

Use case

- Use cases **specify the desired behavior**.
- A use case is
  - a description of a set of sequences of actions a system performs to yield an observable result of value to an actor.
  - includes variants
- Name starts with a verb.
- Each sequence represent an interaction of actors with the system.

Actor

- Represents a set of roles that users of use case play when interacting with these use cases.
- Can **be human or automated systems.**
- Actor is someone interacting with use case (system function). **Named by noun.**
- Actors are entities which require help from the system to perform their task or are needed to execute the system's functions.
- Actors are not part of the system.
- An **Actor triggers use case**
- Actor has responsibility toward the system (inputs), and have expectations from the system (outputs).



name

<< actor >>
X Authorization System

How to create Use case diagrams?

- List main system functions (use cases) in a column
- Draw ovals around the function labels
- Draw system boundary
- Draw actors and connect them with use cases
- Specify include and extend relationships between use cases

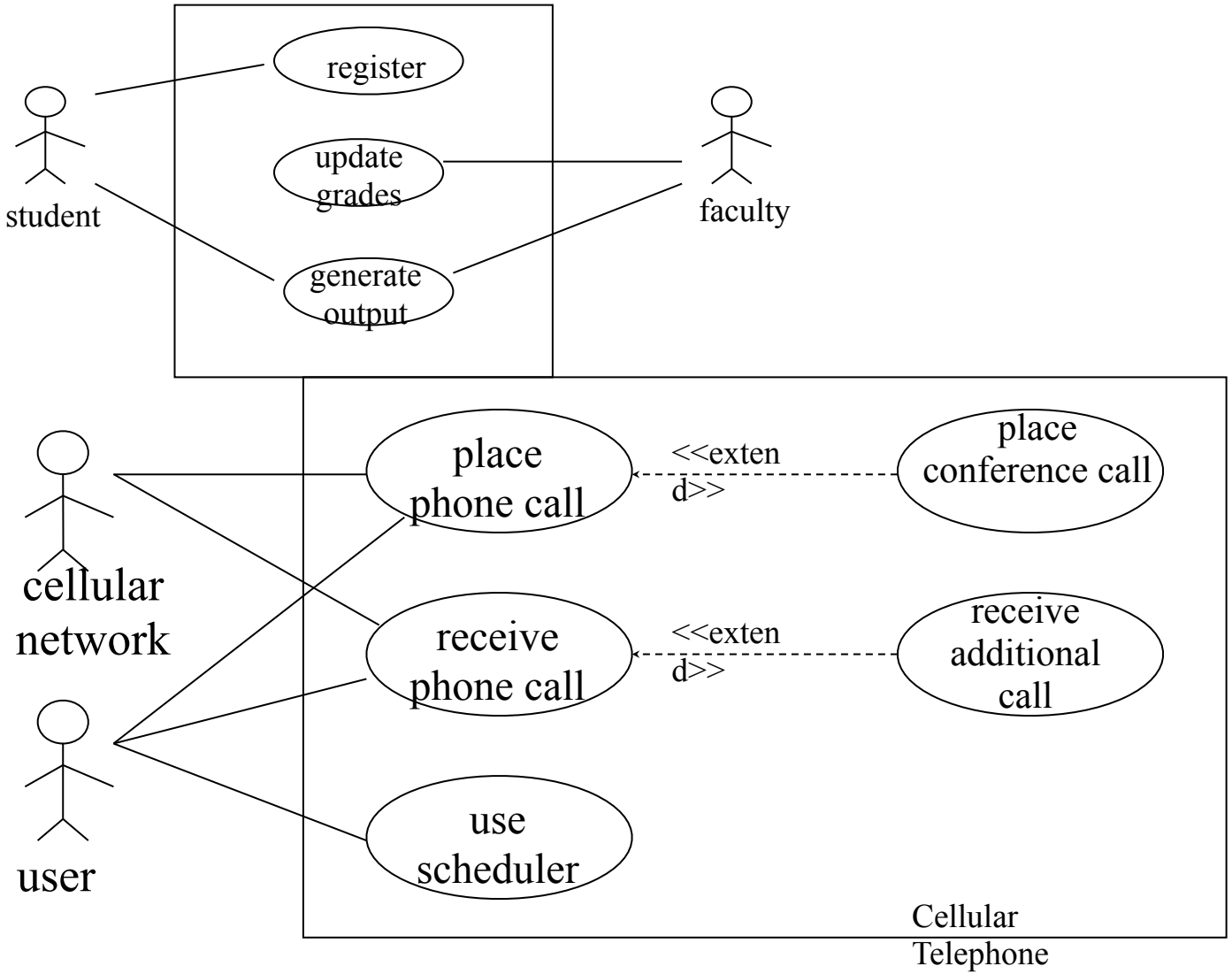
Use case description

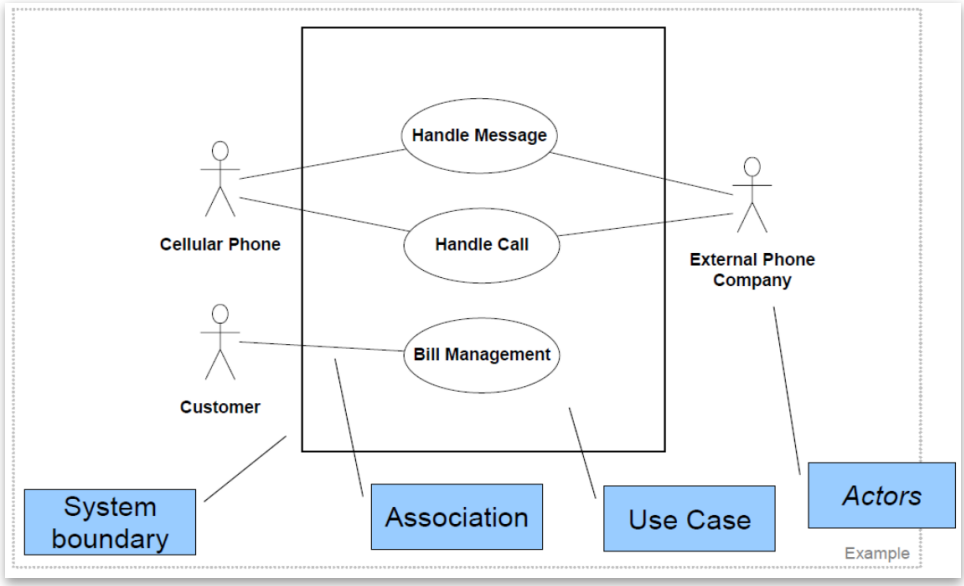
Each use case may include all or part of the following

- Title or Reference Name - meaningful name of the UC
- Author/Date - the author and creation date
- Modification/Date - last modification and its date
- Purpose - specifies the goal to be achieved
- Overview - short description of the processes
- Cross References - requirements references
- Actors - agents participating
- Pre Conditions - must be true to allow execution
- Post Conditions - will be set when completes normally
- Normal flow of events - regular flow of activities
- Alternative flow of events - other flow of activities
- Exceptional flow of events - unusual situations
- Implementation issues - foreseen implementation problems

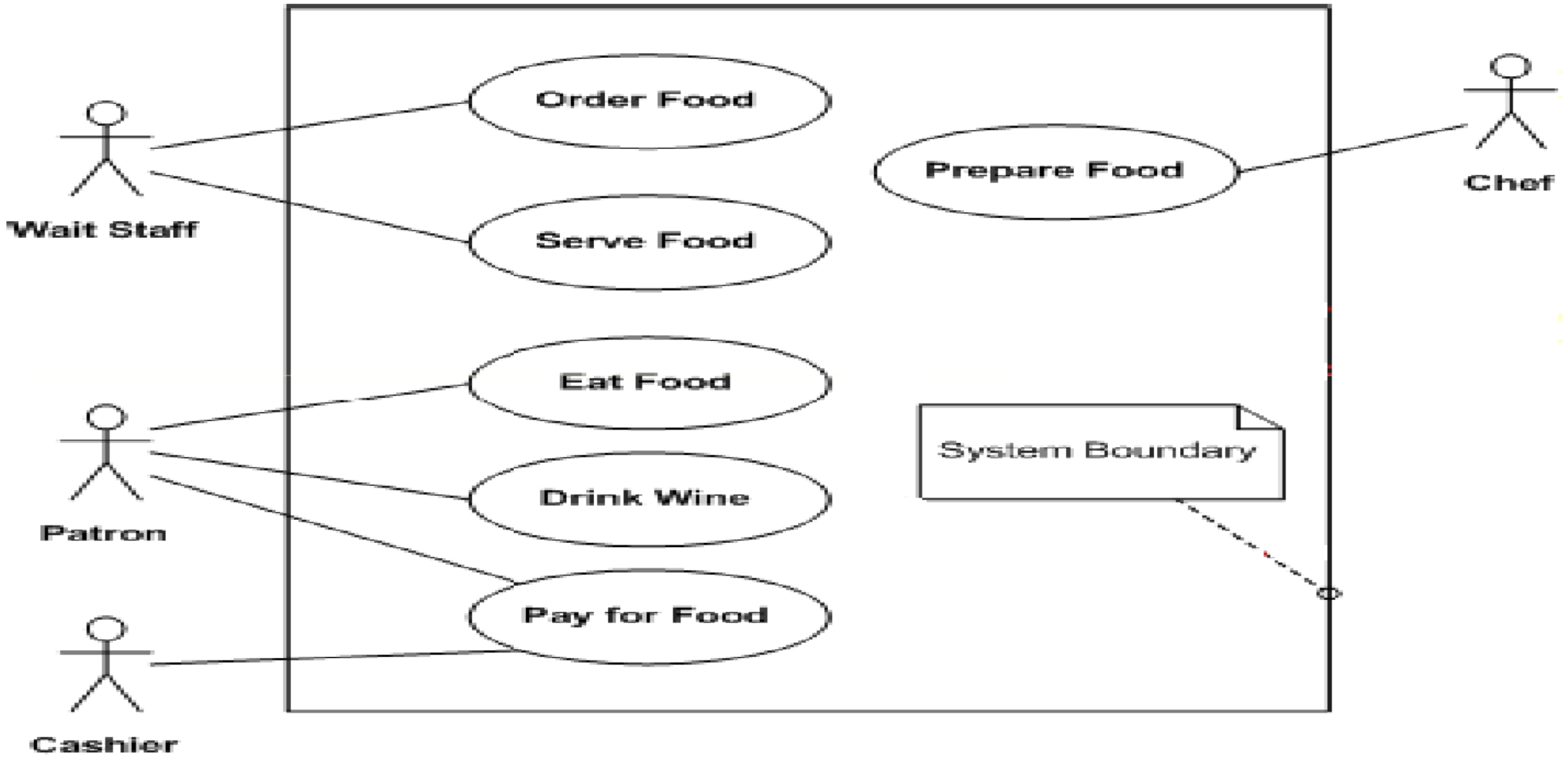
Object Oriented Analysis and Design using Java

Example use case diagram



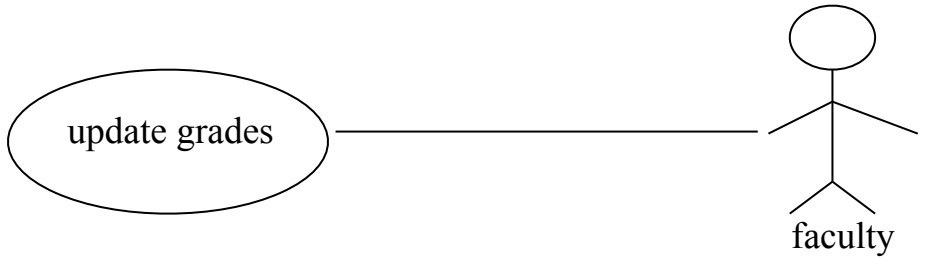


Example use case diagram for a restaurant



Relationships between Use Cases and Actors

- Actors may be connected to use cases by associations, indicating that the actor and the use case communicate with one another using messages.



Relationship between use cases

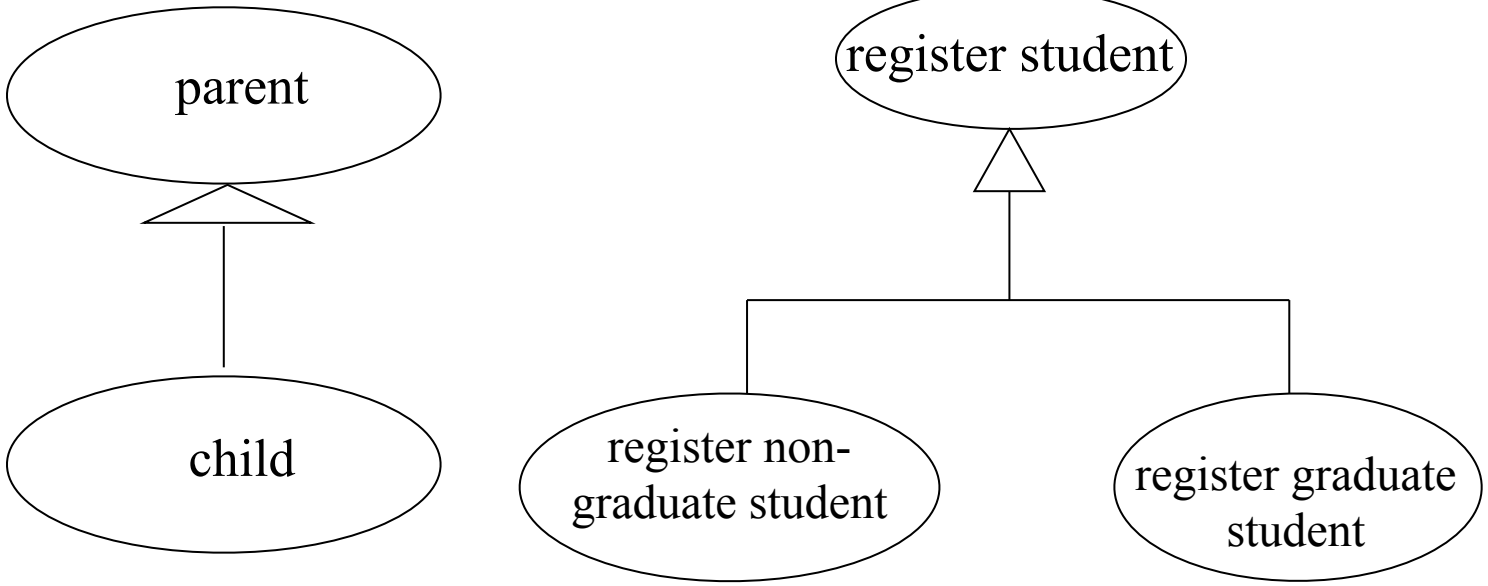
- **Generalization** - Use cases that are specialized versions of other use cases.

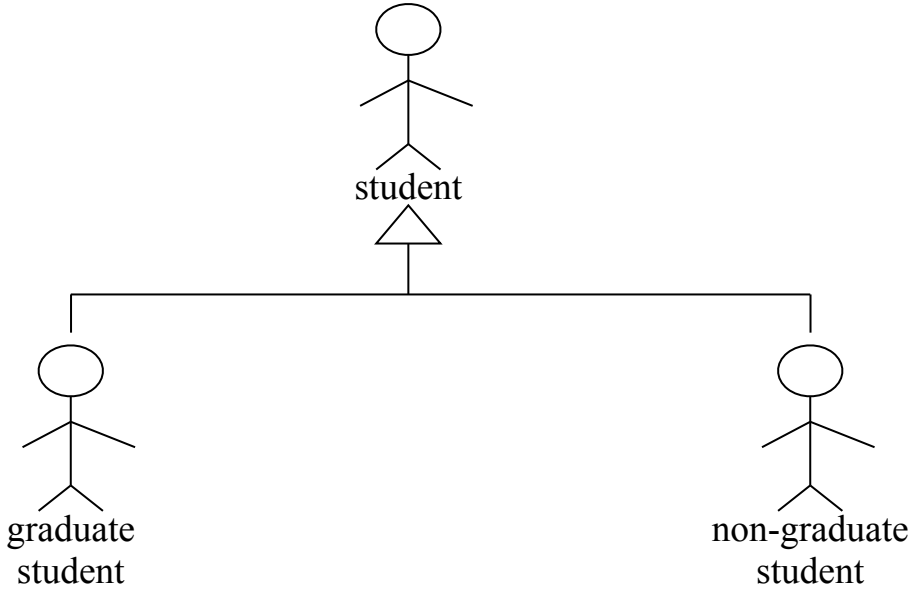
- **Include** - Use cases that are included as parts of other use cases. Enable to factor common behavior.

- **Extend** - Use cases that extend the behavior of other core use cases. Enable to factor variants.

Generalization

- The child use case inherits the behavior and meaning of the parent use case.
- The child may add to or override the behavior of its parent.
- Share the same relationship to the actor

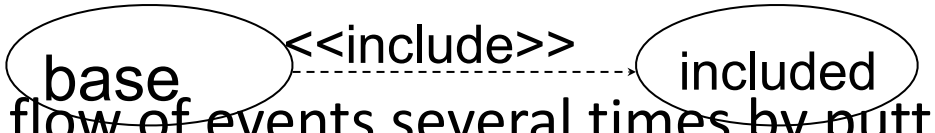


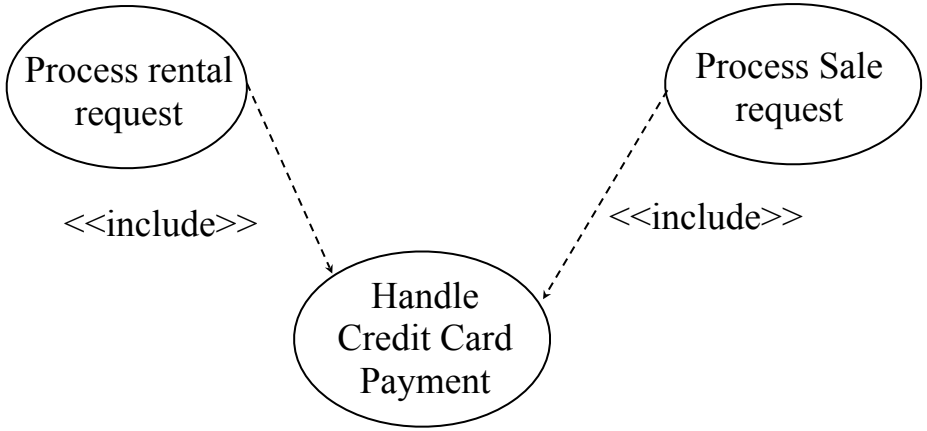


Generalization as a Relationships between Actors

Include

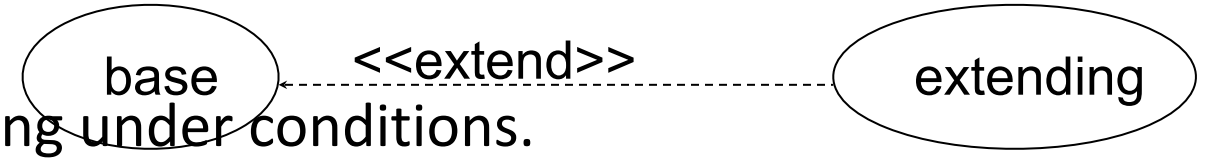
- The base use case explicitly incorporates the behavior of another use case at a location specified in the base.
- The included use case never stands alone. It only occurs as a part of some larger base that includes it.
- Enables to avoid describing the same flow of events several times by putting the common behavior in a use case of its own

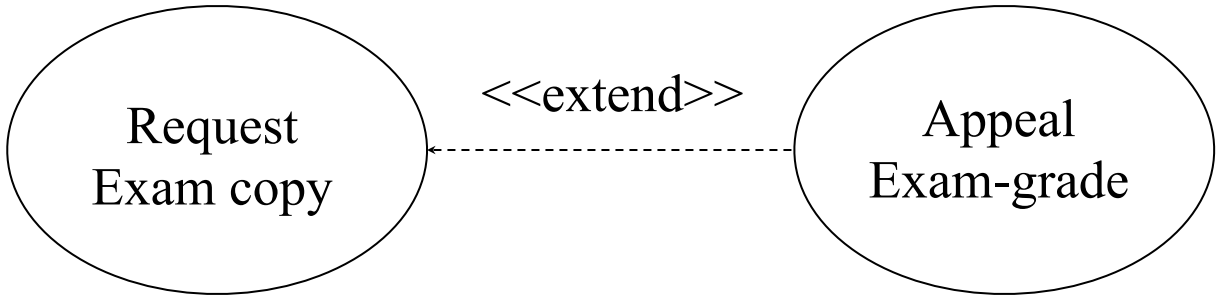




Extend

- The base use case implicitly incorporates the behavior of another use case at certain points called extension points.
- The base use case may stand alone, but under certain conditions its behavior may be extended by the behavior of another use case.
- Enables to model optional behavior or branching under conditions.





Use case specification

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

Sample use case specification

- Name: Transfer Funds
- Summary/Overview : Transfer funds from one account to another
- Actor: Customer
- Pre-conditions: Source account must have sufficient funds
- Description:
  a. Customer identifies the accounts from which and to which funds have to be transferred
  b. Enters the amount to be transferred
  c. Confirm the transaction
- Exceptions
  - Cancel, Insufficient funds, Cannot identify destination account
  - Needs to handle ..say power failure, slow network
- Alternate Flows
  - Handles all the alternate paths (Accounts belonging to the same customer)
- Post-conditions: Funds transferred and account balances updated

Practice: ATM

- Use Case: Withdraw money
- Actors: Customer
- Pre Condition:
  - The ATM must be in a state ready to accept transactions
  - The ATM must have at least some cash on hand that it can dispense
  - The ATM must have enough paper to print a receipt for at least one transaction
- Post Condition:
  - The current amount of cash in the user account is the amount before the withdraw minus the withdraw amount
  - A receipt was printed on the withdraw amount
  - The withdraw transaction was audit in the System log file

ATM : System and user Actions

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

Alternative flow of events:

Step 3: Customer authorization failed. Display an error message, cancel the transaction and eject the card.
Step 8: Customer has insufficient funds in its account. Display an error message, and go to step 6.
Step 8: Customer exceeds its legal amount. Display an error message, and go to step 6.

Exceptional flow of events:

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

Few Inputs

- One method to identify use cases is actor-based:
  - Identify the actors related to a system or organization.
  - For each actor, identify the processes they initiate or participate in.
- A second method to identify use cases is event-based:
  - Identify the external events that a system must respond to.
  - Relate the events to actors and use cases.
- The following questions may be used to help identify the use cases for a system:
  - What are tasks of each actor ?
  - Will any actor create, store, change, remove, or read information in the system ?
  - What use cases will create, store, change, remove, or read this information ?
  - Will any actor need to inform the system about sudden, external changes ?
  - Does any actor need to be informed about certain occurrences in the system ?
  - Can all functional requirements be performed by the use cases ?

THANK YOU

Prof. Vinay Joshi

Department of Computer Science and Engineering

vinayj@pes.edu

Object Oriented Analysis and Design Using Java

UE21CS352B
UNIT-1

Lecture:04-Class Modelling-UML Class Diagrams

Dr. Sudeepa Roy Dey
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Object Oriented Analysis and Design Using Java

Class Modelling: UML Class Diagrams

Dr. Sudeepa Roy Dey

Department of Computer Science and Engineering

With grateful thanks for contribution of slides to:
Prof Ruby D, Asst Professor at the Department of CSE, PESU-EC

Class Modelling

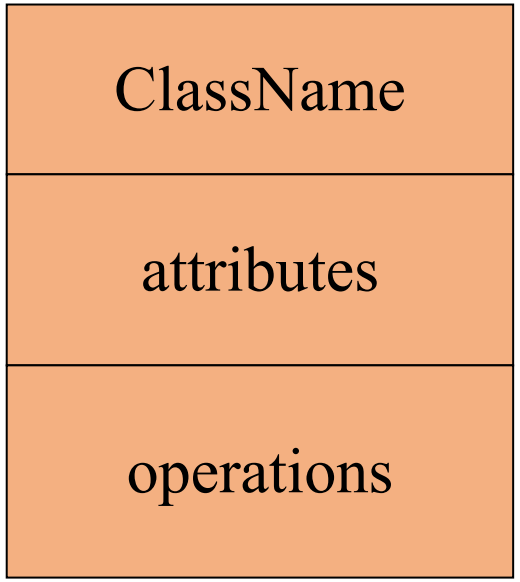
- A class model captures the static structure of a system by characterizing the objects in the system, the relationships between the objects and the attributes and operations for each class of objects.
- **Class diagram** is a graphical notation used to construct and visualize object oriented systems.
- Class diagram can be mapped directly with object oriented languages.
- Class diagrams capture the static structure of Object-Oriented systems as how they are structured rather than how they behave.
- It supports architectural design.
- They identify what classes are there, how they interrelate and how they interact.
- A UML class diagram is made up of:

  - A set of classes and
  - A set of relationships between classes

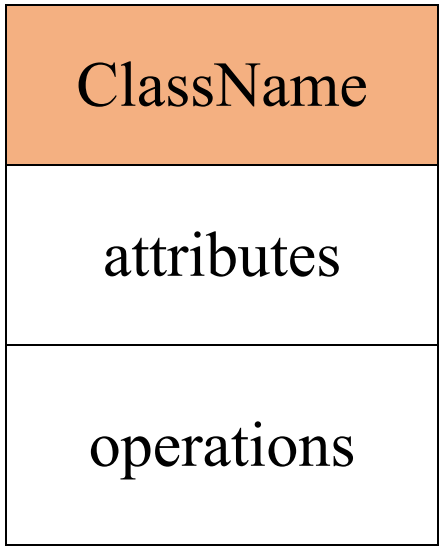
Class

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, usually including its name, attributes, and operations in separate, designated compartments.

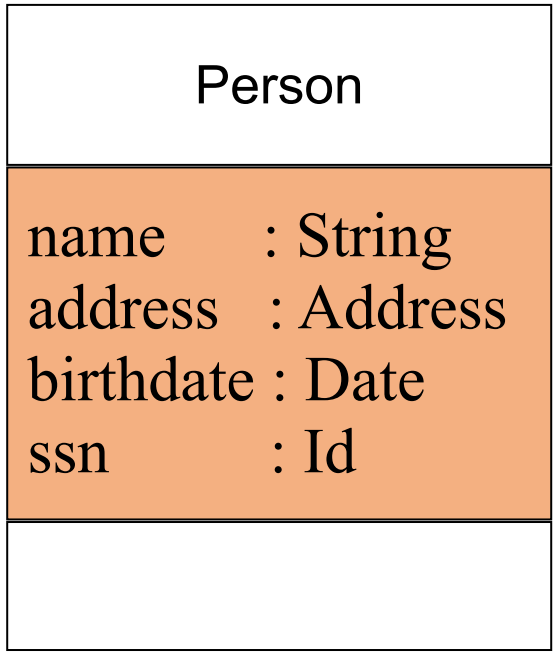


Class



The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.

Attributes



Each class can have attributes.

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

Each attribute has a type.

In the class diagram, attributes appear in the second compartment just below the name-compartment.

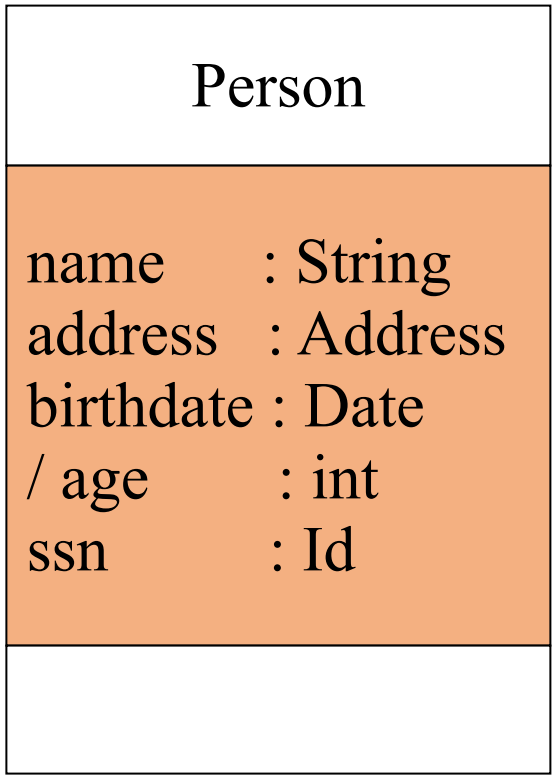
Derived Attribute

Attributes are usually listed in the form:

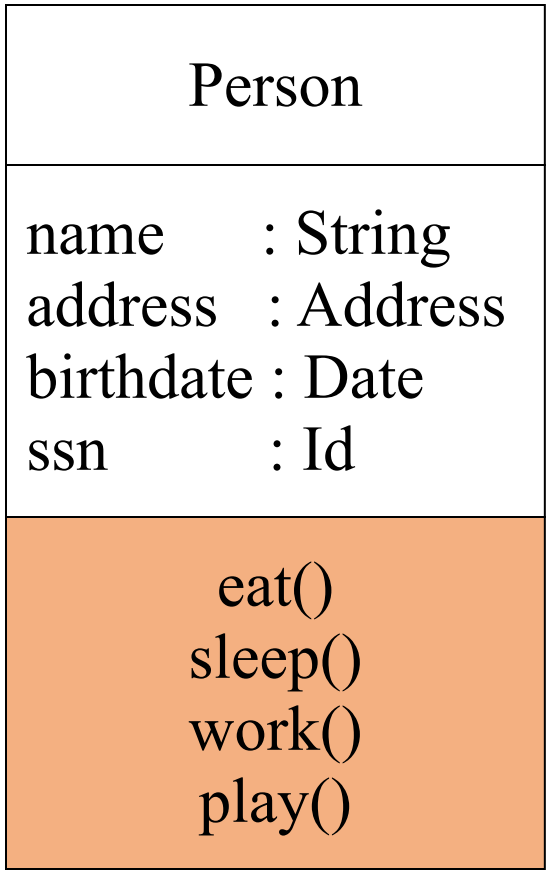
attributeName : Type

A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist.
For example,
a Person's age can be computed from his birth date.
A derived attribute is
designated by a preceding '/' as in:

/ age : int

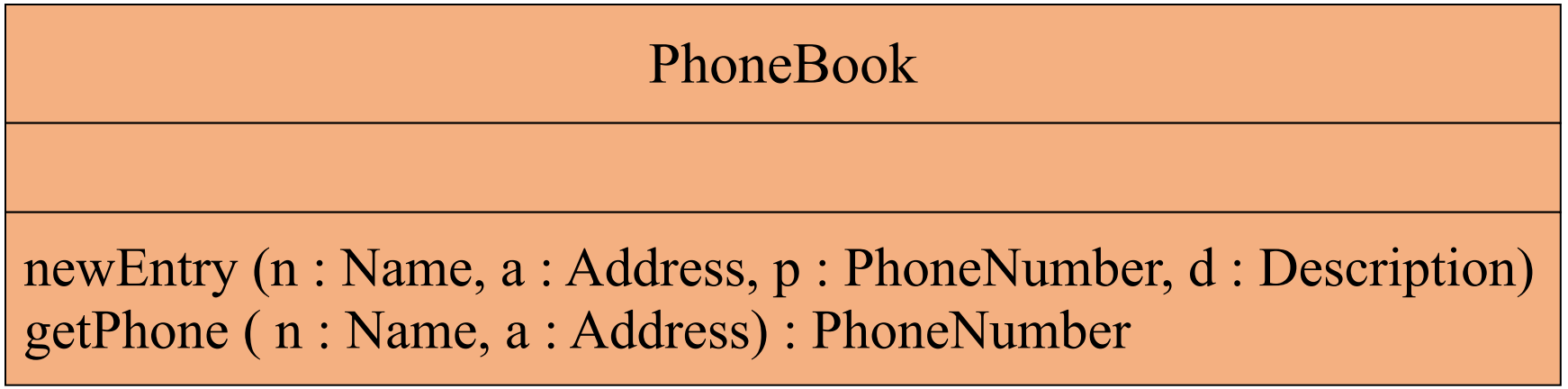


Class Operations



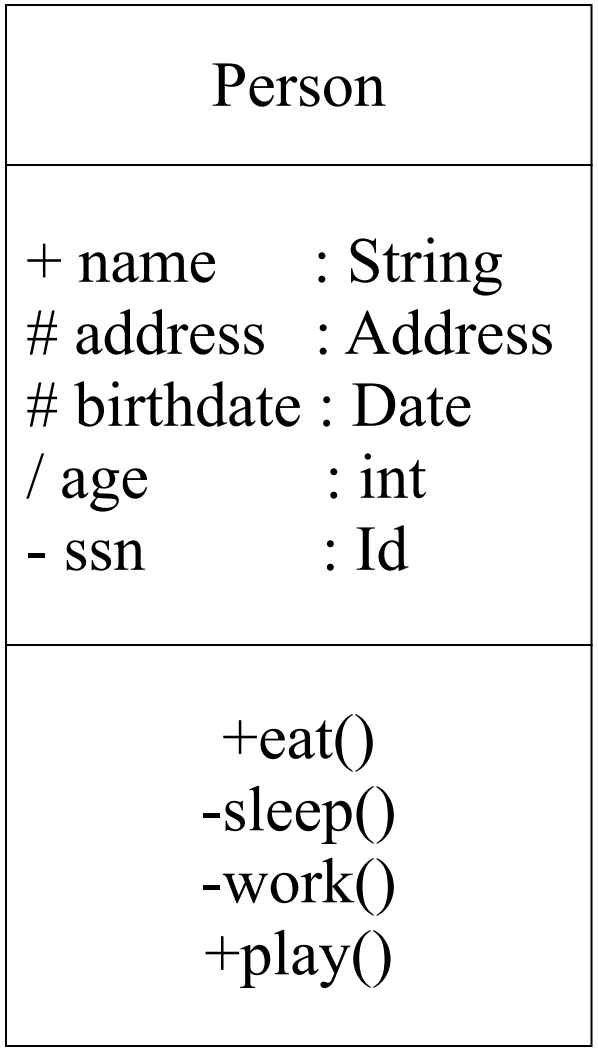
Operations describe the class behavior and appear in the third compartment.

Class operations



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

Visibility



attributes and operations can be declared with different visibility modes:

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

protected: any descendant of the class can use the feature;

- private: only the class itself can use the feature.

Finding Classes

Finding classes in use case, or in text descriptions:
Look for nouns and noun phrases in the description of a use case or a problem statement;
These are only included in the model if they explain the nature or structure of information in the application.

Don't create classes for concepts which:
Are beyond the scope of the system;
Refer to the system as a whole;
Duplicate other classes;
Are too vague or too specific (few instances)

Finding classes in other sources:
Reviewing background information;
Users and other stakeholders;
Analysis patterns;
CRC (Class Responsibility Collaboration) cards.

Approaches for identifying classes

1. Noun phrase approach
2. Common class patterns approach
3. Classes, responsibilities and collaborators (CRC) approach (to be discussed next lecture)
4. Use case driven approach (discussed in previous session)

1. Noun phrase approach

- Proposed by Rebecca, brian and lauren.
- The requirement document need to be read and noun phrase should be identified.
- Nouns to be considered as classes and verbs to be methods of the classes.
- All plurals changed to singular.
- Nouns are listed and divided into three categories:
  relevant, fuzzy and irrelevant classes
- Irrelevant classes need to be identified .
- Candidate classes can be identified from other two categories.

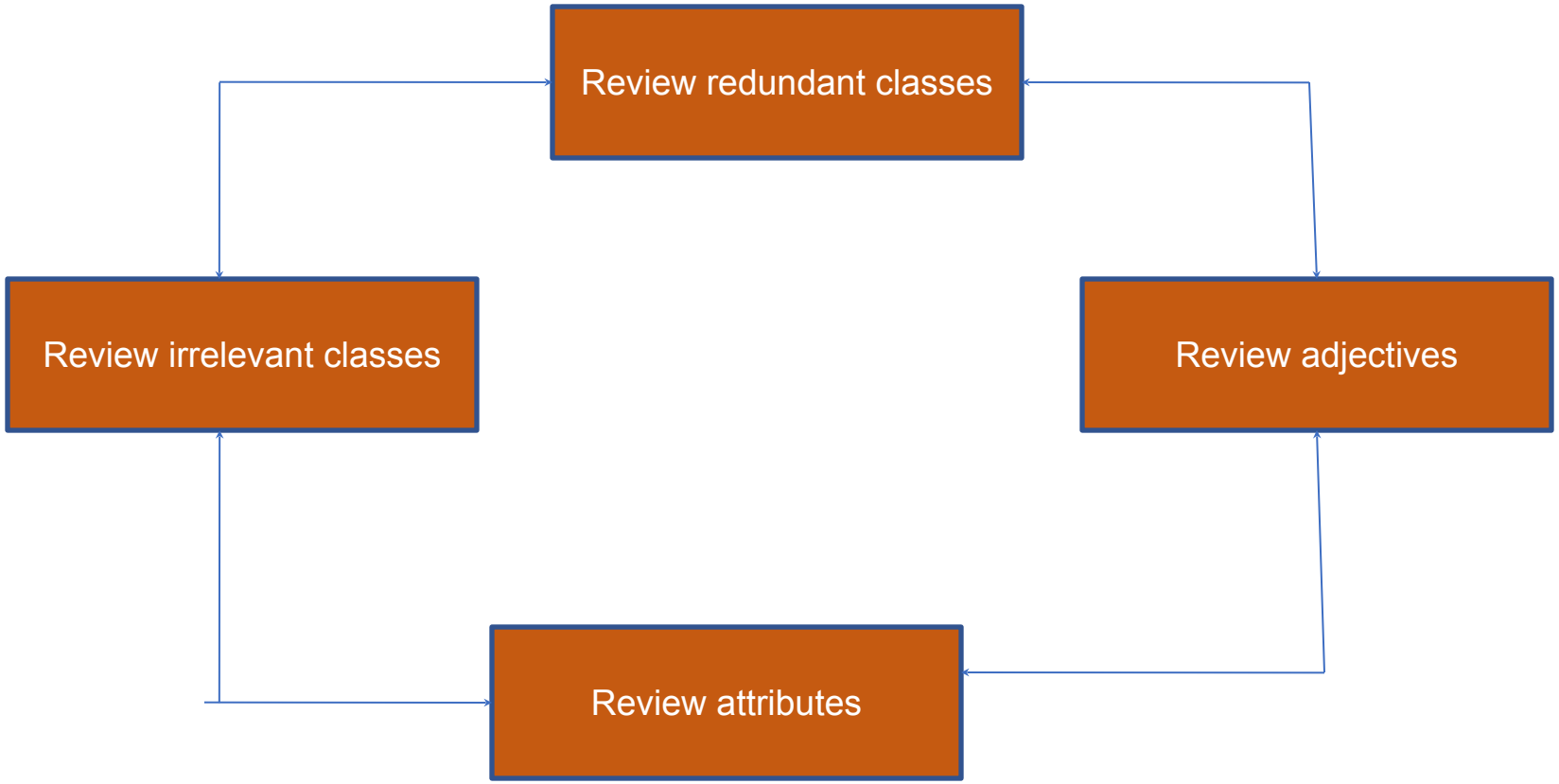
- Identifying tentative classes:
  - Look for noun and noun phrases in the use cases.
  - Some classes are implicit or taken from general knowledge.
  - Carefully choose and define class names.
- Finding classes is an incremental and iterative process.

selecting classes from the relevant and fuzzy categories

- Guidelines to select candidate classes from the relevant and fuzzy categories:
  - Redundant classes: select one class if more than one class describes the same information. This is part of building a common vocabulary for the whole system. Choose the word that is used by the user.
  - Attribute classes: Tentative objects that are used only as values should be defined or restarted as attributes and not as a class. For example client status can be the attribute not the class.

- Irrelevant classes: class should be defined clearly with its purpose. If the purpose of statement is not possible for any class, class should be eliminated.

- The process of identifying classes and refining is the iterative process.
- This is not the sequential process. You can move back and forth.


Review redundant classes
Review irrelevant classes
Review adjectives
Review attributes

2. Common class patterns approach

- Researchers like shlaer and mellor proposed some patterns for identifying the candidate class and objects
- Concept class:
  - Emphasis principles that are not tangible but used to organize or keep track of business activities or communications.
  - Ex: performance

- Event class:
  - These are points in time that must be recorded.
  - Things happen, usually to something else at given date and time or as a step in an ordered sequence.
  - Associated with things remembered are attributes such as who,what, when, where, how or why.
  - Ex: Landing,interrupt

- Organization class:
  - it's the collection of people, resources,facilities or groups to which the users belong.
  - Ex: An accounting department might be considered a potential class.

- People class:
  - It specifies different roles users play in interacting with the application.

  - *Two types:*
    - the users of the system (operator or clerk) or who interacts with the system.
    - Who do not use the system but whom information is kept by the system.

- Place class:
  - Places are physical locations that the system must keep information.
  - Ex: buildings, stores and offices
- Tangible things and device class:
  - This includes physical objects or groups of objects that are tangible.
  - The devices with which the application interacts.
  - Ex: cars,pressure sensors

References

[1] "**Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development**", by Craig Larman, 3rd Edition, Pearson 2015.

[2] **"Object-Oriented Software Engineering: Practical Software Development using UML and Java"**, Timothy C. Lethbridge, Robert Laganière, Second edition, Mc Gram Hill,2016

[3] **"Object-Oriented Modelling and Design with UML"**, Michael R Blaha and James R Rumbaugh, 2nd Edition, Pearson 2007.

[4] *https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/*

THANK YOU

Dr. Sudeepa Roy Dey

Department of Computer Science and Engineering

sudeepar@pes.edu

Object Oriented Analysis and Design Using Java

UE21CS352B
UNIT-1

Lecture:05-Class Modelling: OO-Relationships

Dr. Sudeepa Roy Dey
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Object Oriented Analysis and Design Using Java

Class Modelling: OO-Relationships

Dr. Sudeepa Roy Dey

Department of Computer Science and Engineering

With grateful thanks for contribution of slides to:
Prof Ruby D, Asst Professor at the Department of CSE, PESU-EC

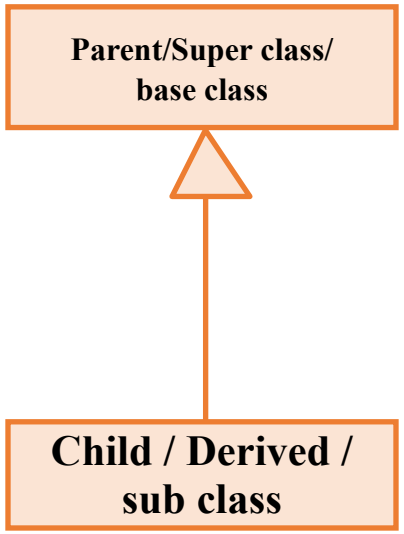
OO-Relationships

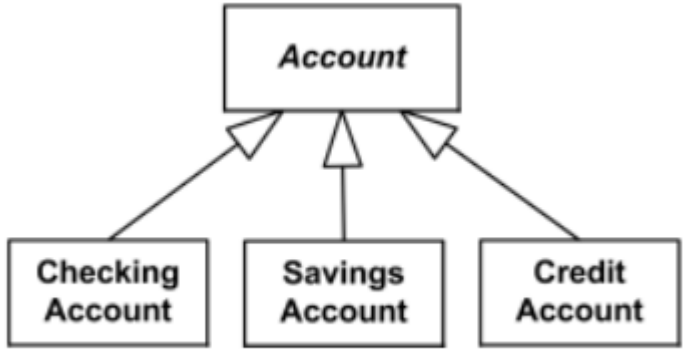
In UML, object interconnections (logical or physical), are modeled as relationships. The type of relationships are listed below:

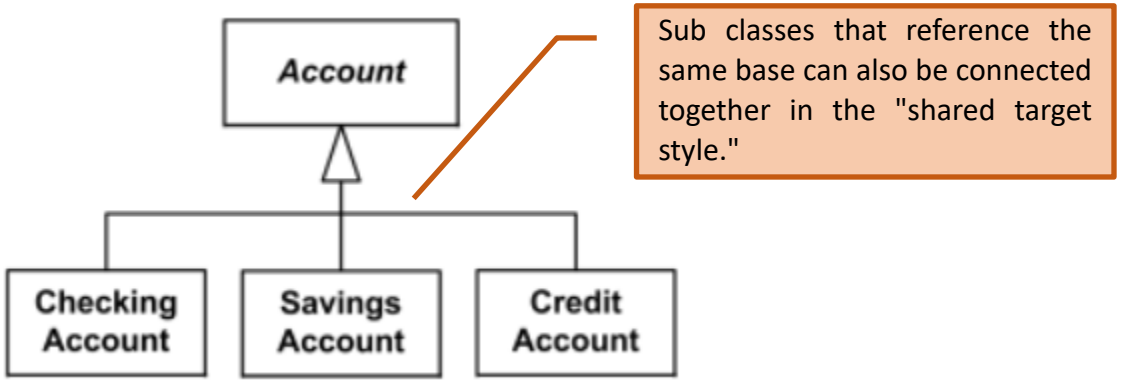
- Generalization- an inheritance relationship
- Abstraction
- Realization - interface implementation
- Dependency
- Association – using relationship
    - Aggregation
    - Composition

Generalization

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

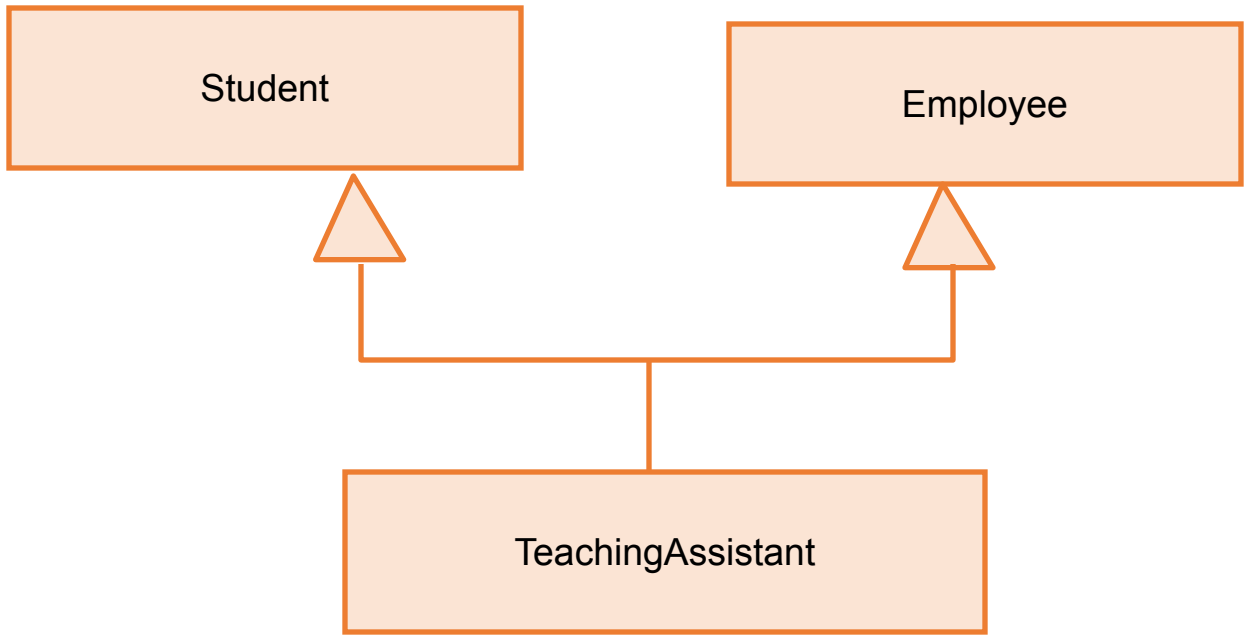






Generalization Relationships (Cont'd)

UML permits a class to inherit from multiple super classes, although some programming languages (*e.g.,* Java) do not permit multiple inheritance.

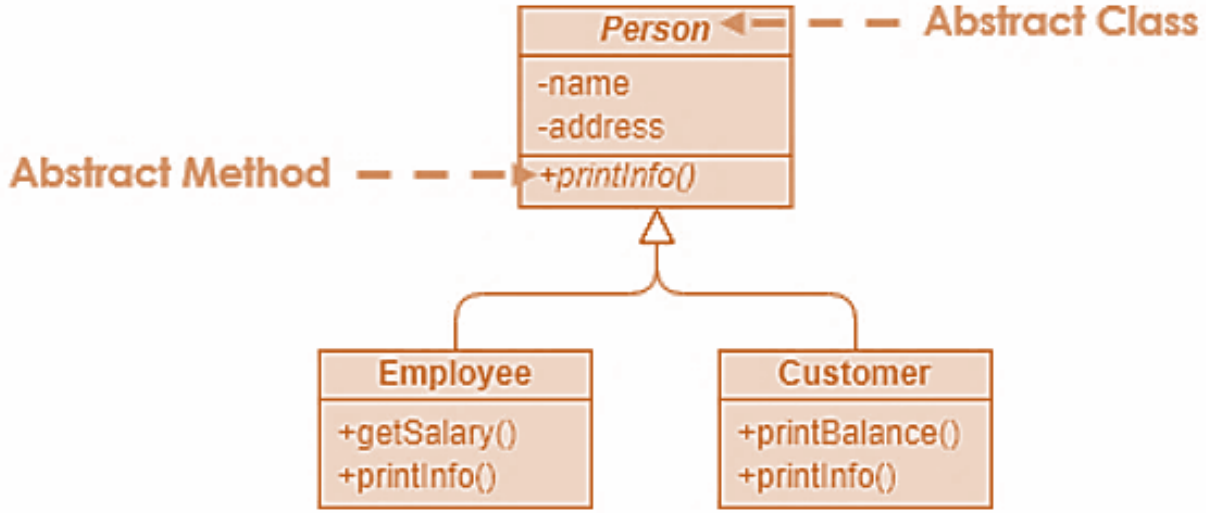


Abstraction

In an inheritance hierarchy, subclasses implement specific details, whereas the parent class defines the framework its subclasses. The parent class also serves as a template for common methods that will be implemented by its subclasses.

The name of an abstract Class is typically shown in italics; alternatively, an abstract Class may be shown using the textual annotation, also called stereotype {abstract} after or below its name.

An abstract method is a method that do not have implementation. In order to create an abstract method, create a operation and make it italic



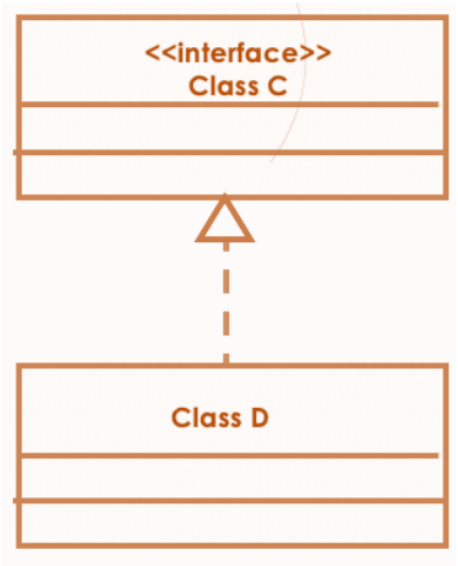
Object Oriented Analysis and Design Using Java

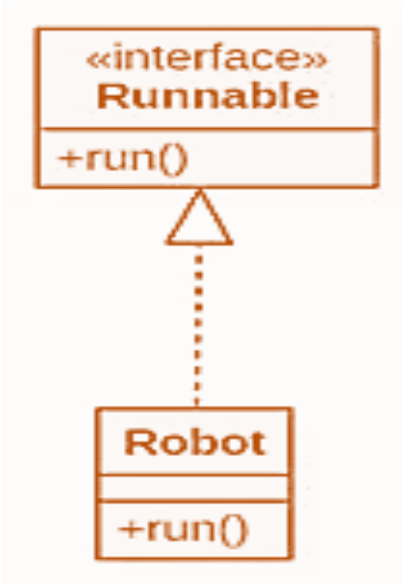
Realization / Interfaces

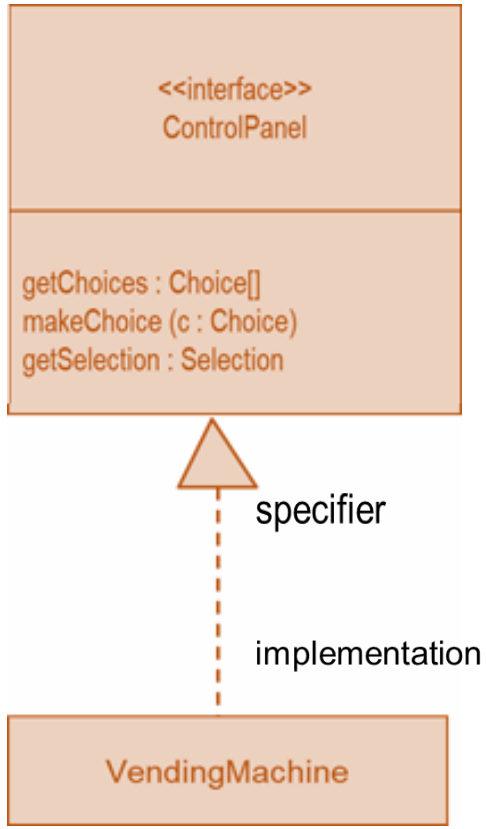
Realization is a specialized abstraction relationship between two things where one thing (an interface) specifies a contract that another thing (a class) guarantees to carry out by implementing the operations specified in that contract.

Interface defines a set of functionalities as a contract and the other entity “realizes” the contract by implementing the functionality defined in the contract.

Realization is shown as a dashed directed line with an open arrowhead pointing to the interface.





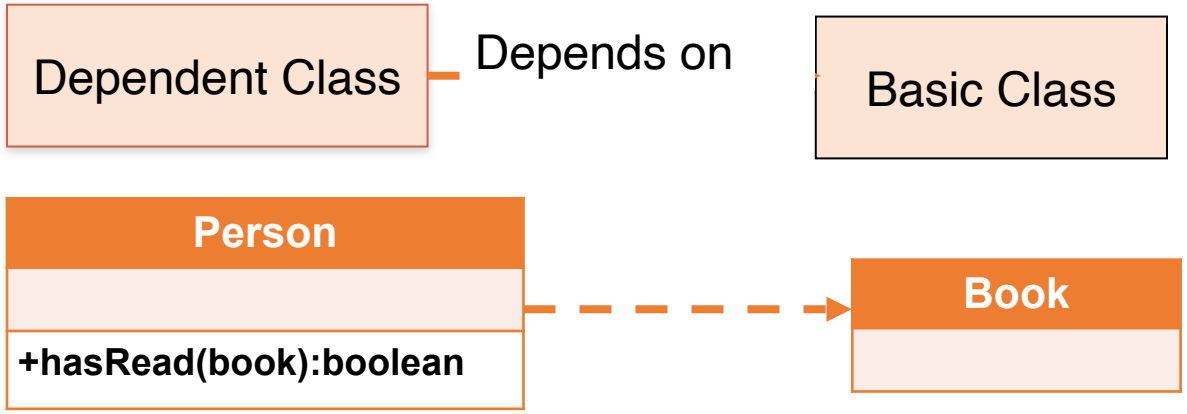


Dependency

•Dependency means "**One class uses the other**"

•A dependency relationship indicates that a **change** in **one class** may **affect** the **dependent class**, but not necessarily the reverse.

•A dependency relationship is often used to show that a **method has object** of a class **as arguments**.



An object of one class might use an object of another class in the code of a method. If the object is not stored in any field, then this is modeled as a dependency relationship. For example, the Person class might have a hasRead method with Book as a parameter that returns true if the person has read the book.

Object Oriented Analysis and Design Using Java

Association Relationships

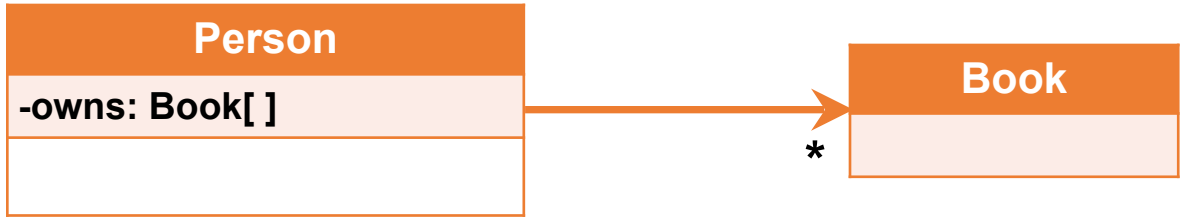
If two classes in a model need to communicate with each other, there must be link between them.
An *association* denotes that link.

We can constrain the association relationship by defining the *navigability* of the association.
Association is of two types.
They are

- Unidirectional Association
- Bidirectional Association

Association Relationships (Cont'd)

- In a unidirectional association, two classes are related, but only one class "knows" that the relationship exists.
- A unidirectional association is drawn as a solid line with an open arrowhead pointing to the known class.
- Uni-directional association includes a role name and a multiplicity value, but only for the known class.
- In this example Orderdetails class only knows that it has a relationship with Item class but Item class does not know about their relationship.



An object might store another object in a field. For example, people own books, which might be modeled by an owns field in Person objects. However, a book might be owned by a very large number of people, so the reverse direction might not be modeled. The * in the figure indicate that a person can own any number of books.

Association Relationships (Cont'd)

We can indicate the *multiplicity* of an association by adding *multiplicity adornments* to the line denoting the association.

The multiplicity of an association is the number of possible instances of the class associated with a single instance of the other end.

Multiplicities are single number or ranges of numbers.

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

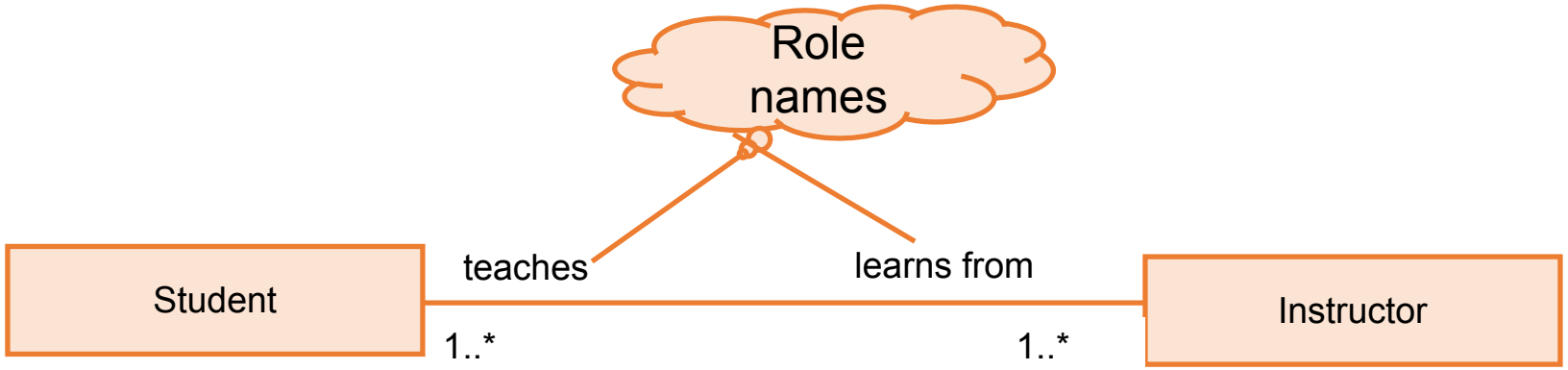
Association Relationships (Cont'd)

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

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

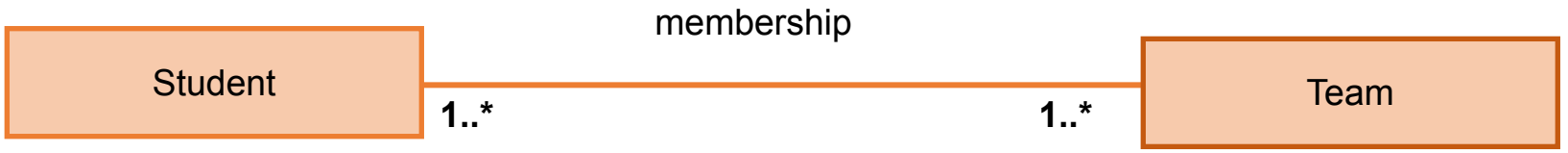
Association Relationships (Cont'd)

We can also indicate the behavior of an object in an association (*i.e.,* the *role* of an object) using *role names.*



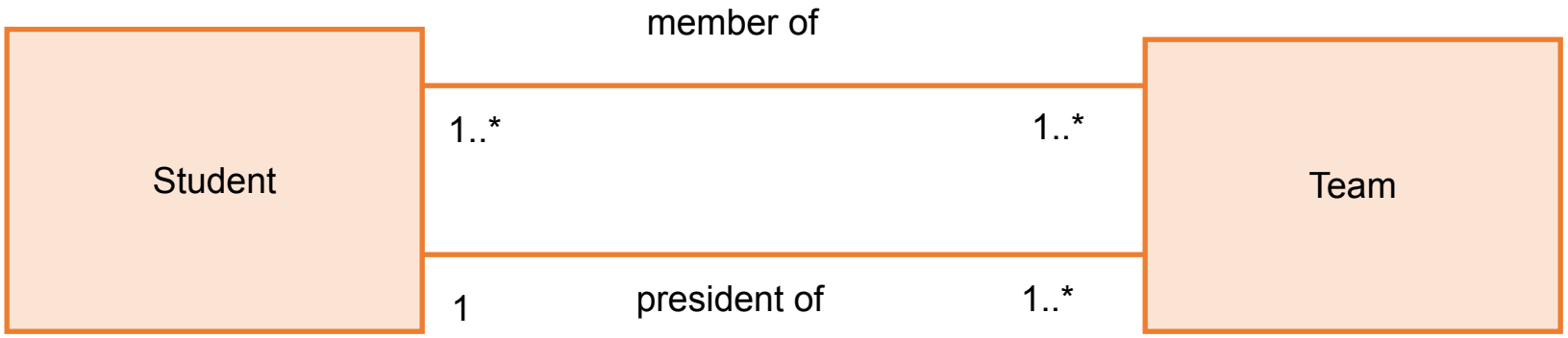
Association Relationships (Cont'd)

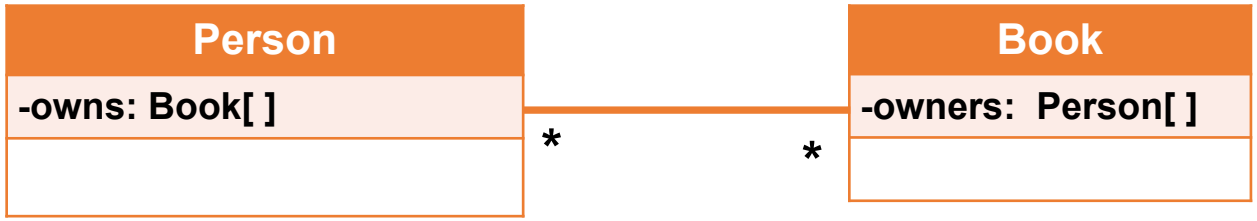
We can also name the association.



Association Relationships (Cont'd)

We can specify dual associations using bidirectional association

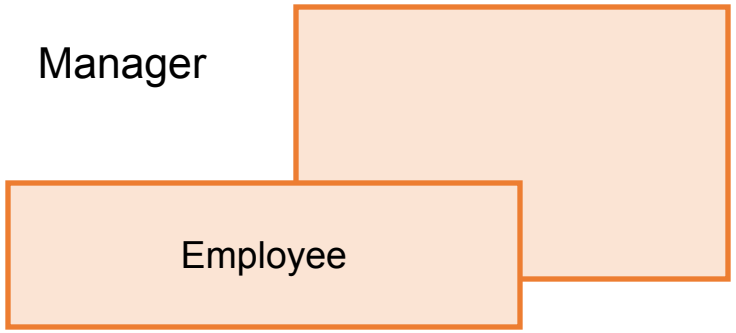




Two objects might store each other in fields. For example, in addition to a Person object listing all the books that the person owns, a Book object might list all the people that own it

Association Relationships (Cont'd)

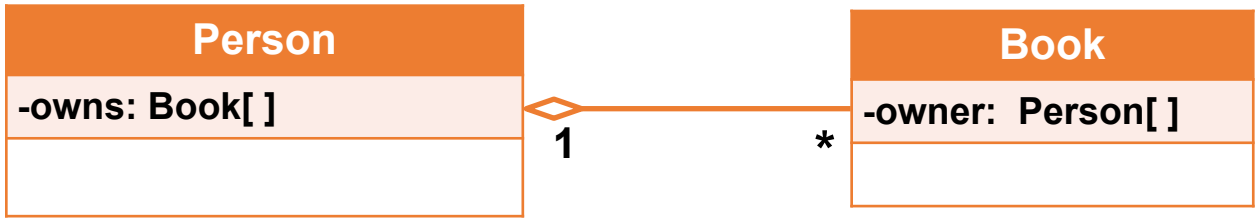
A class can have a *self association*.



Aggregation

We can model objects that contain other objects by way of special associations called *aggregations* and *compositions. It is also known as "has a" relationship.*

An *aggregation* specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment on the association.

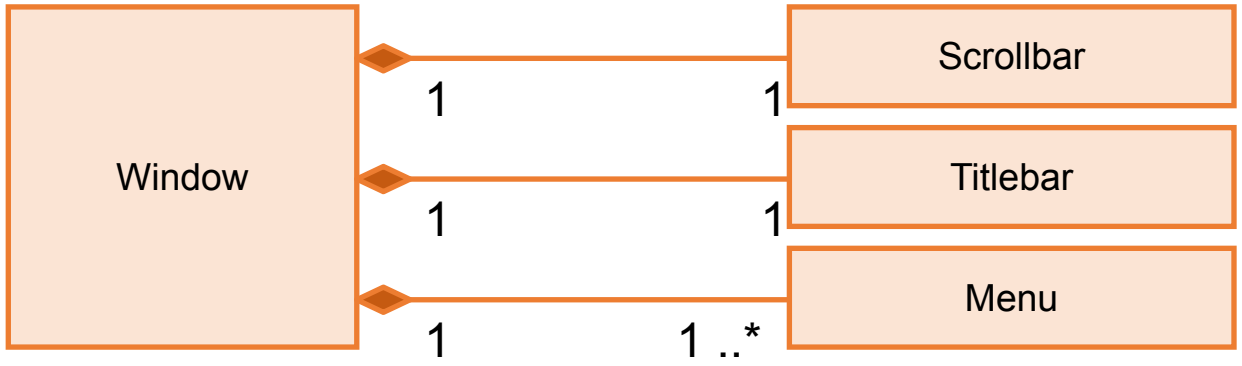


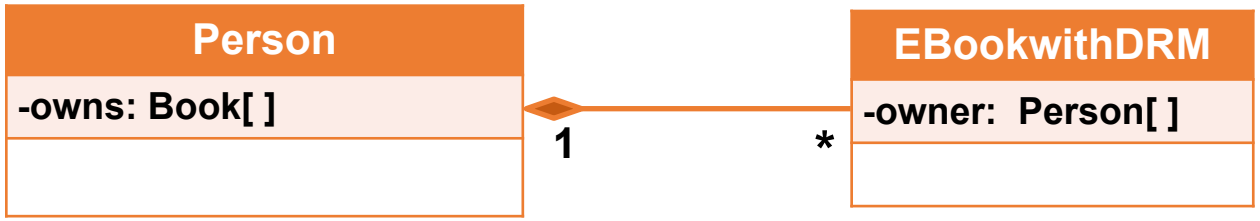
One object A has or owns another object B, and/or B is part of A. For example, suppose there are different Book objects for different physical copies. Then the Person object has/owns the Book object, and, while the book is not really part of the person, the book is part of the person's property. In this case, each book will (usually) have one owner. Of course, a person might own any number of books.

Object Oriented Analysis and Design Using Java

Composition

A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.,* they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.

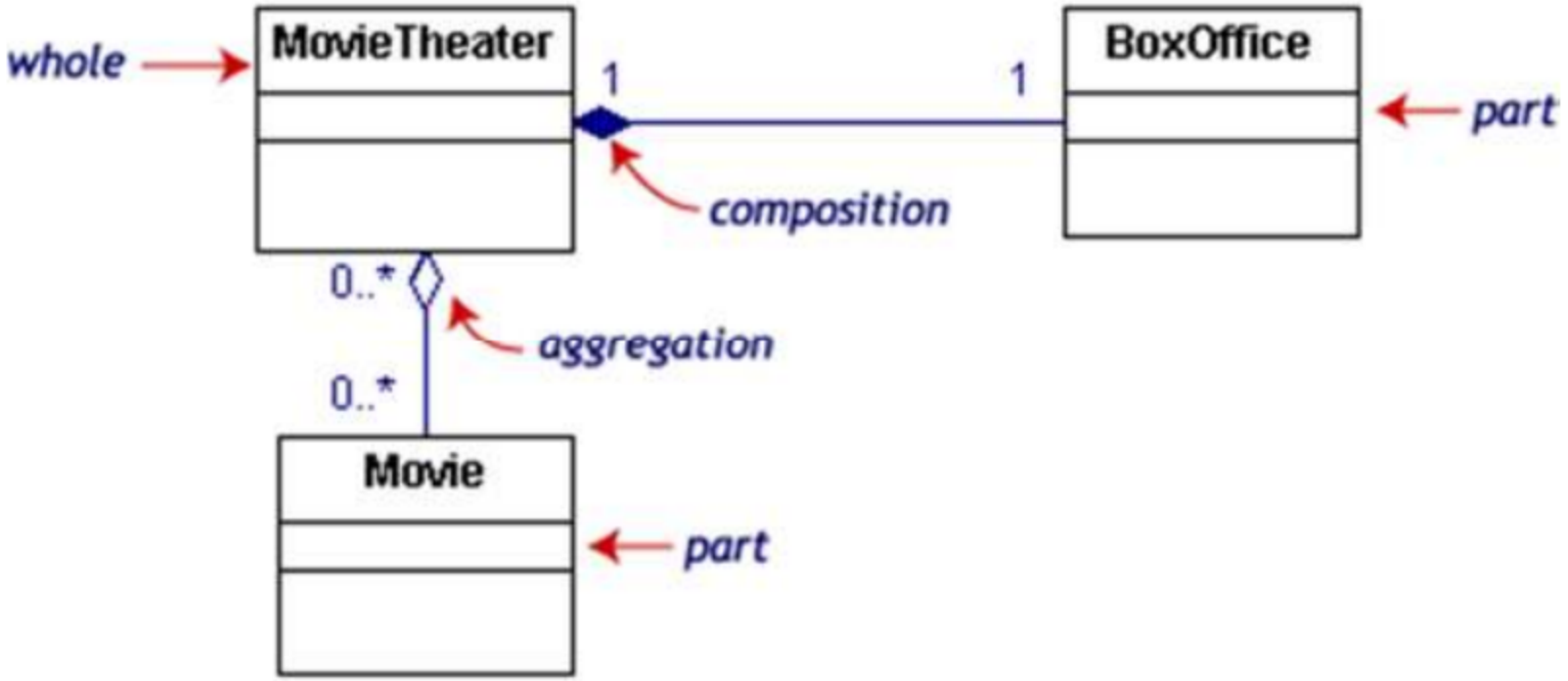




In addition to an aggregation relationship, the lifetimes of the objects might be identical, or near For example, in an idealized world of electronic books with DRM (Digital Rights Management), a person can own an ebook, but cannot sell it. After the person dies, no one else can access the ebook. [This is idealized, but might be considered less than ideal.]

Object Oriented Analysis and Design Using Java

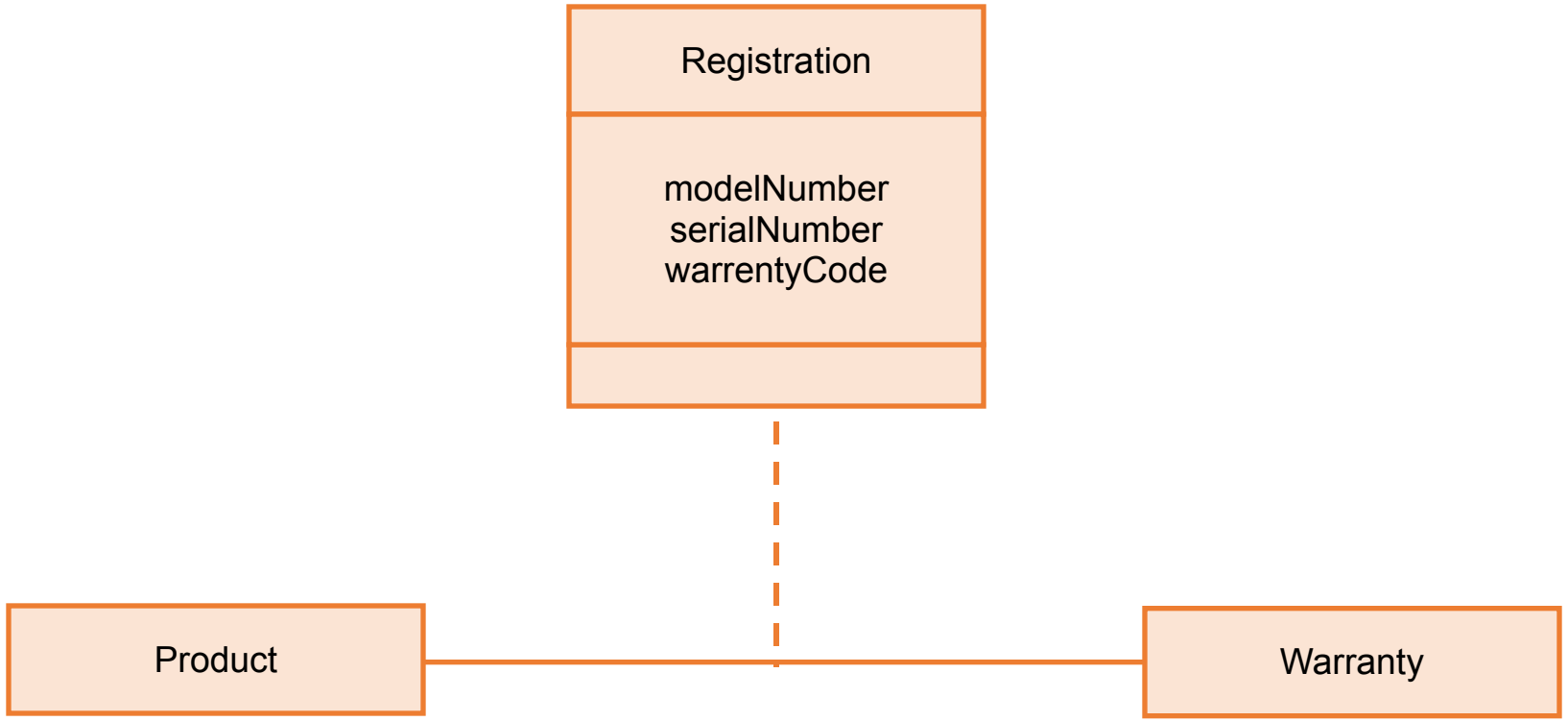
Composition / Aggregation Example



If the movie theater goes away
so does the box office => composition
but movies may still exist => aggregation

Association Relationships (Cont'd)

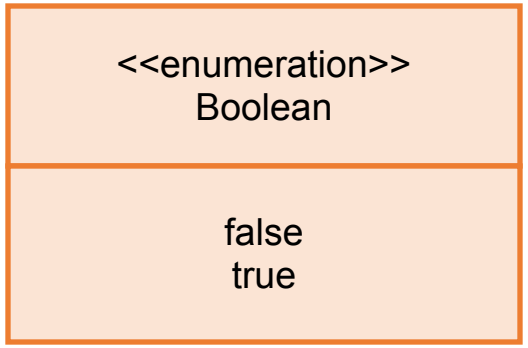
Associations can also be objects themselves, called *link classes* or an *association classes*.
An association class is represented like a normal class, but it is linked to an association line with a dotted line.



Object Oriented Analysis and Design Using Java

Enumeration

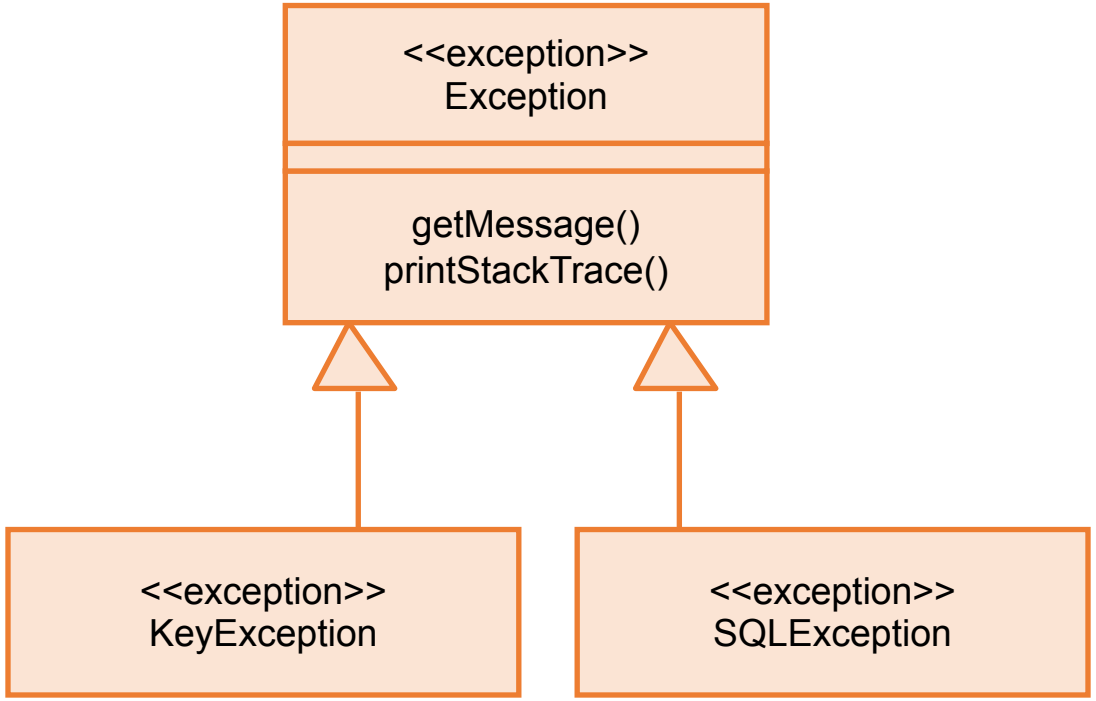
An *enumeration* is a user-defined data type that consists of a name and an ordered list of enumeration literals.



Exceptions

Exceptions can be modeled just like any other class.

Notice the <<exception>> stereotype in the name compartment.



How to make a class diagram

1.Identify all the classes participating in the software solution.

2. Draw them in a class diagram.

3. Identify the attributes.

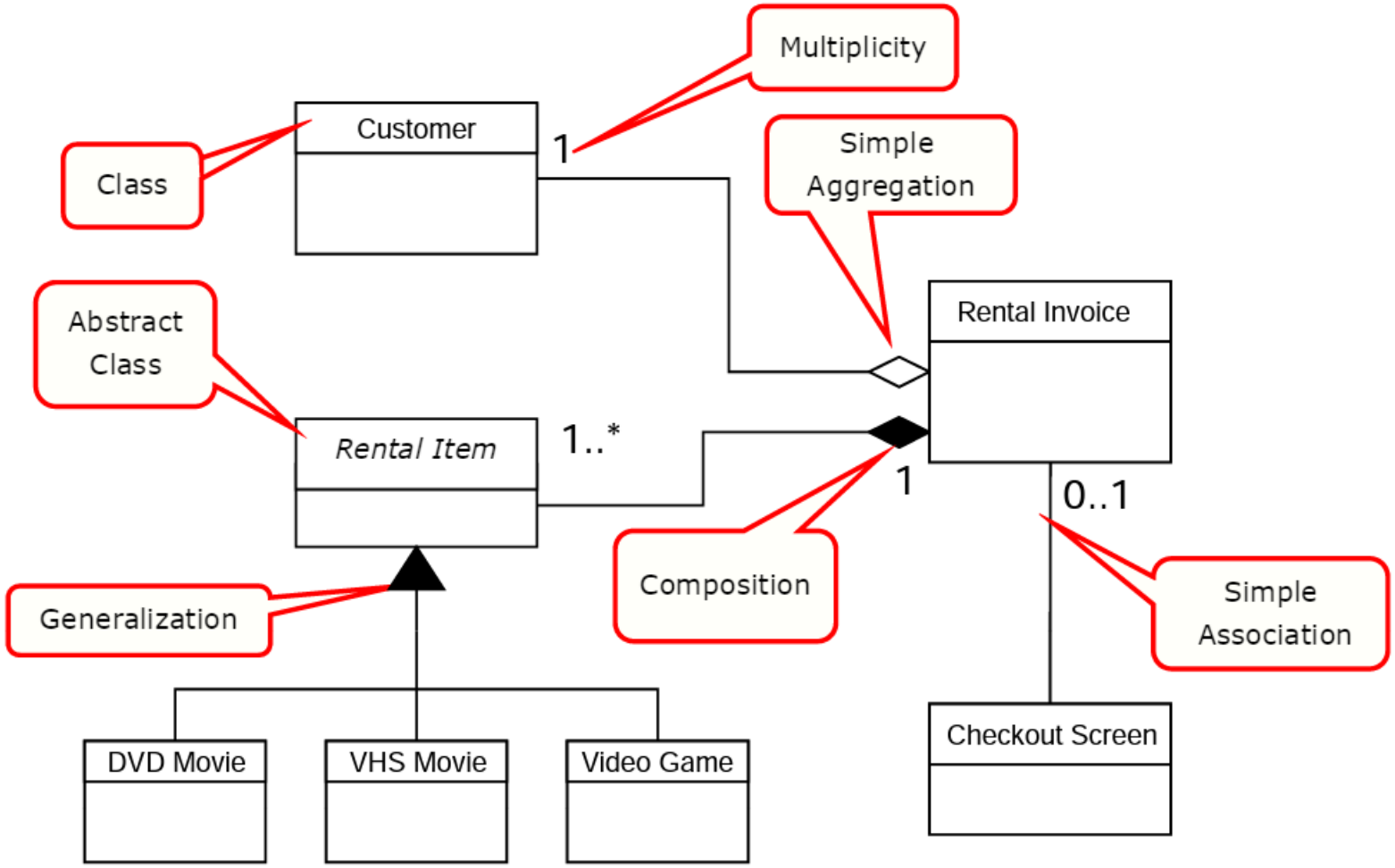
4. Identify the methods.

5. Add associations, generalizations, aggregations and dependencies.

6. Add other stuff (roles, constraints, ...)

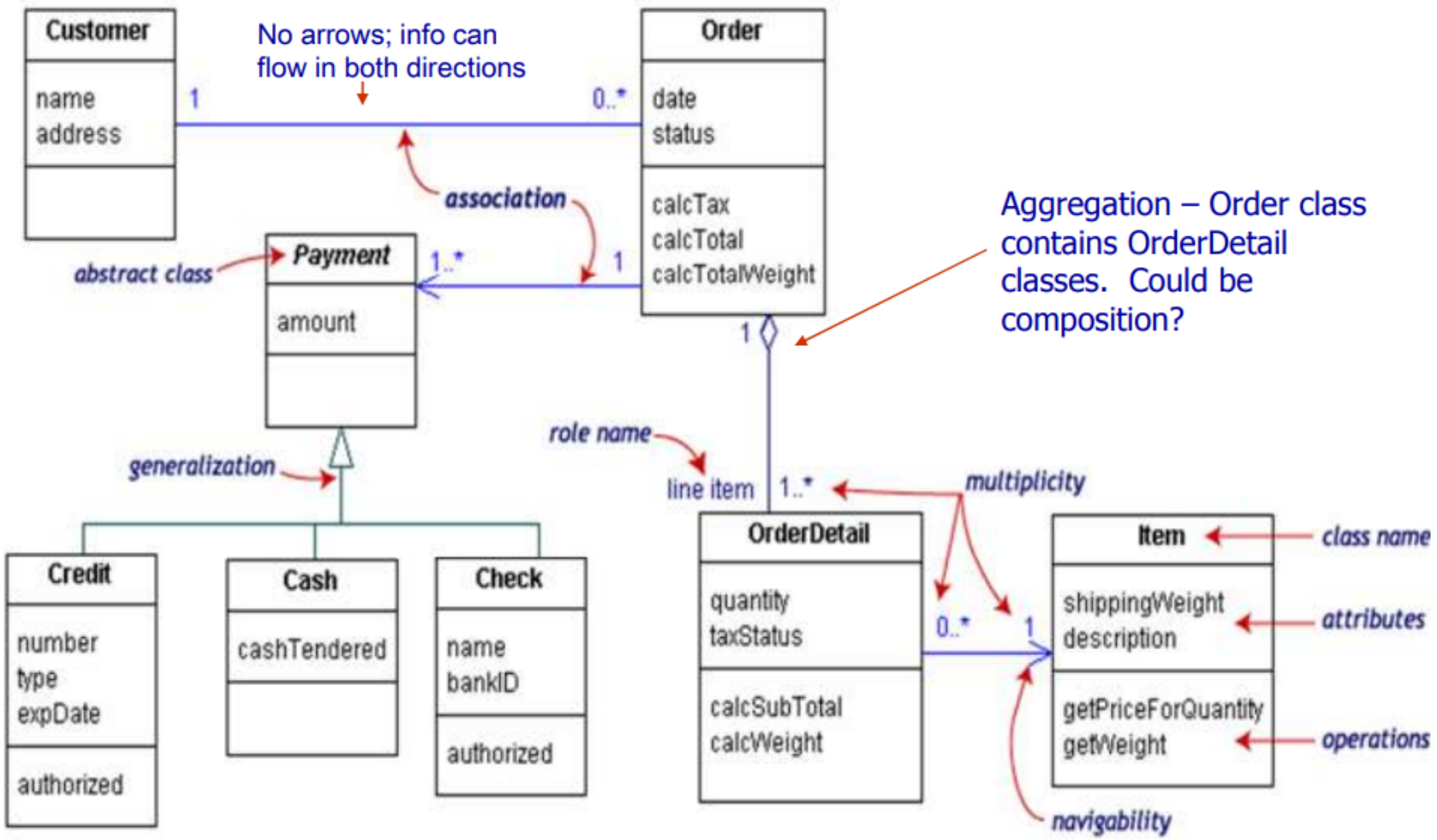
Object Oriented Analysis and Design Using Java

Example class diagram –Video Store



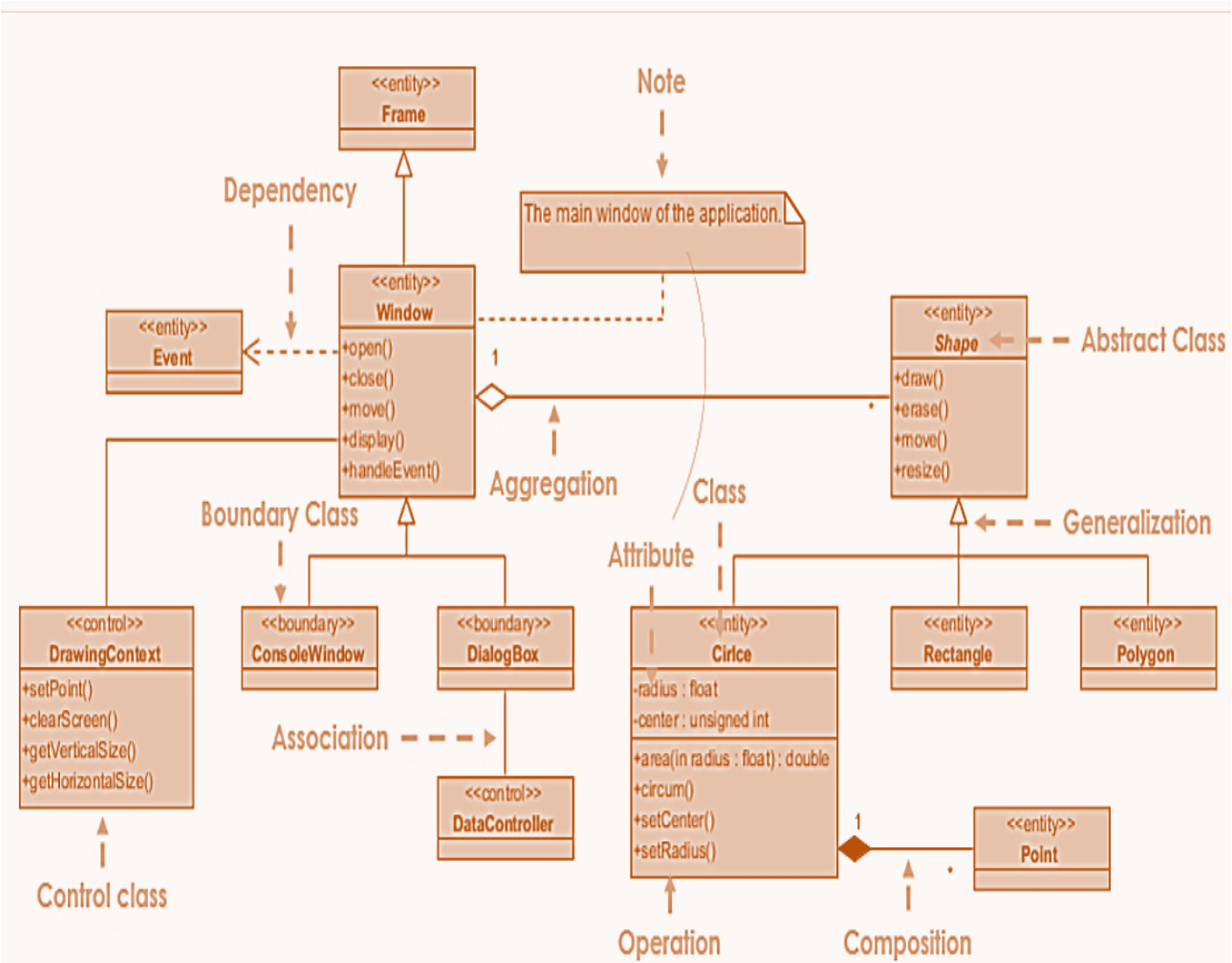
Object Oriented Analysis and Design Using Java

Example class diagram –Video Store



Object Oriented Analysis and Design Using Java

Example class diagram



Shape is an abstract class. It is shown in Italics.
Shape is a superclass. Circle, Rectangle and Polygon are derived from Shape. This is a generalization / inheritance relationship.
There is an association between DialogBox and DataController.
Shape is part-of Window. This is an aggregation relationship.
Shape can exist without Window.
Point is part-of Circle. This is a composition relationship.
Point cannot exist without a Circle.
Window is dependent on Event. However, Event is not dependent on Window.
The attributes of Circle are radius and center. This is an entity class.
The method names of Circle are area(), circum(), setCenter() and setRadius().
The parameter radius in Circle is an in parameter of type float.
The method area() of class Circle returns a value of type double.
The attributes and method names of Rectangle are hidden.

References

[1] "**Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development**", by Craig Larman, 3rd Edition, Pearson 2015.

[2] **"Object-Oriented Software Engineering: Practical Software Development using UML and Java"**, Timothy C. Lethbridge, Robert Laganière, Second edition, Mc Gram Hill,2016

[3] **"Object-Oriented Modelling and Design with UML"**, Michael R Blaha and James R Rumbaugh, 2nd Edition, Pearson 2007.

[4] https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

[5] https://online.visual-paradigm.com/diagrams/tutorials/class-diagram-tutorial/

THANK YOU

Dr. Sudeepa Roy Dey

Department of Computer Science and Engineering

sudeepar@pes.edu

Object Oriented Analysis and Design Using Java

UE21CS352B
UNIT-1

Lecture:06-CRC for a case study

Dr. Sudeepa Roy Dey
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Object Oriented Analysis and Design Using Java

CRC for a case study

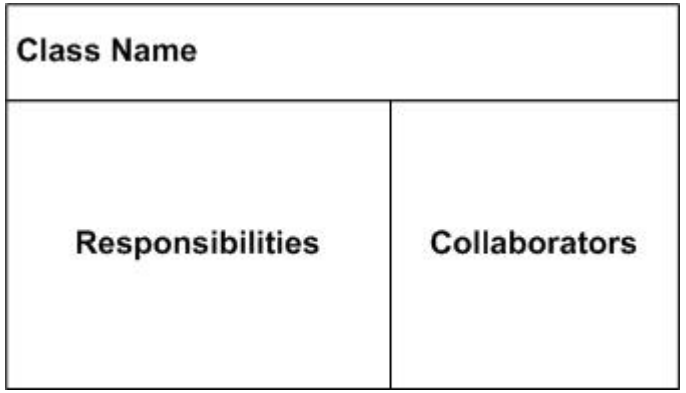
Dr. Sudeepa Roy Dey

Department of Computer Science and Engineering

With grateful thanks for contribution of slides to:
Prof. Ruby D, Asst Professor at the Department of CSE, PESU-EC

CRC (Class-Responsibility-Collaborators)

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



What is a CRC Card?

CRC stands for Class, Responsibility and Collaboration.

- Class
  - An object-oriented class name
  - Include information about super- and sub-classes
- Responsibility
  - What information this class stores
  - What this class does
  - The behaviour for which an object is accountable
- Collaboration
  - Relationship to other classes
  - Which other classes this class uses

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

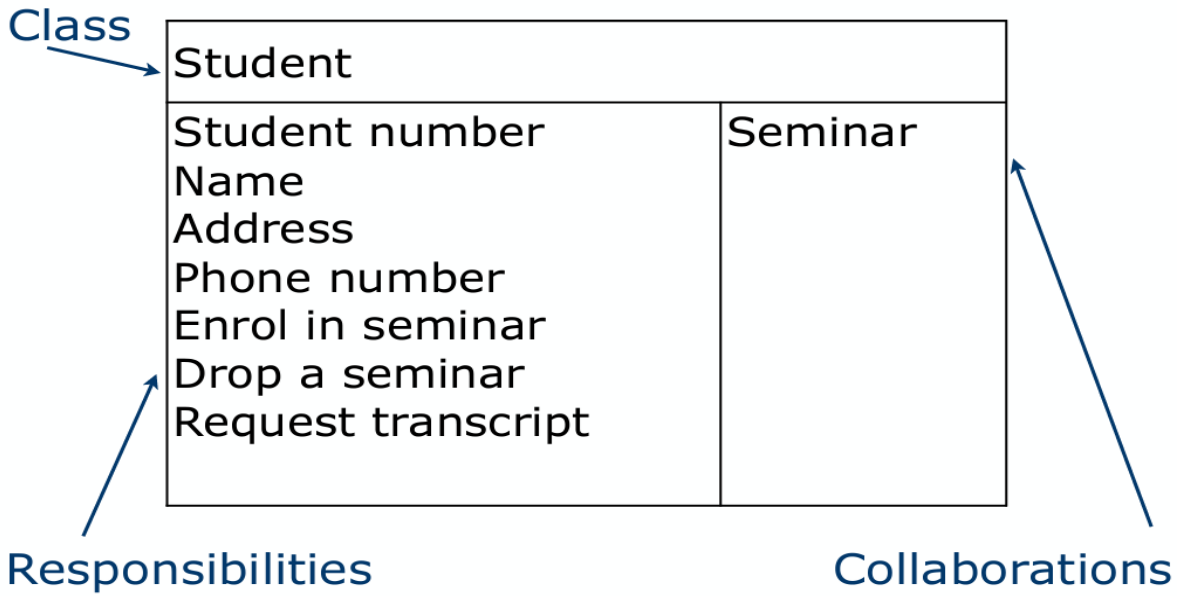
Ex: CRC card

Object Oriented Analysis and Design Using Java

Example: Student CRC card.

A responsibility is anything that a class knows or does. The things a class knows and does constitute its responsibilities.

For example, students have names, addresses, and phone numbers. These are the things a student knows. Students also enroll in seminars, drop seminars, and request transcripts. These are the things a student does.



students only have information about themselves (their names and so forth), and not about seminars. What the student needs to do is collaborate/interact with the card labeled *Seminar* to sign up for a seminar. Therefore, *Seminar* is included in the list of collaborators of *Student*.

Refer to link : https://agilemodeling.com/artifacts/crcmodel.htm for details

- CRC cards are an aid to a group role-playing activity.
- Index cards are used in preference to pieces of paper due to their robustness and to the limitations that their size (approx. 15cm x 8cm) imposes on the number of responsibilities and collaborations that can be effectively allocated to each class.
- A class name is entered at the top of each card and responsibilities and collaborations are listed underneath as they become apparent.
- For the sake of clarity, each collaboration is normally listed next to the corresponding responsibility.
- From a UML perspective, use of CRC cards is in analyzing the object interaction that is triggered by a particular use case scenario.

The process of using CRC cards is usually structured as follows.

1.Conduct a session to identify which objects are involved in the use case.

2. Allocate each object to a team member who will play the role of that object.

3. Act out the use case.

4. Identify and record any missing or redundant objects.

References

[1] "**Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development**", by Craig Larman, 3rd Edition, Pearson 2015.

[2] **"Object-Oriented Software Engineering: Practical Software Development using UML and Java"**, Timothy C. Lethbridge, Robert Laganière, Second edition, Mc Gram Hill,2016

[3] **"Object-Oriented Modelling and Design with UML"**, Michael R Blaha and James R Rumbaugh, 2nd Edition, Pearson 2007.

[4] *https://agilemodeling.com/artifacts/crcmodel.htm*

THANK YOU

Dr. Sudeepa Roy Dey

Department of Computer Science and Engineering

sudeepar@pes.edu

Object Oriented Analysis and Design Using Java

UE21CS352B

Prof. Nivedita Kasturi
Department of Computer Science and Engineering

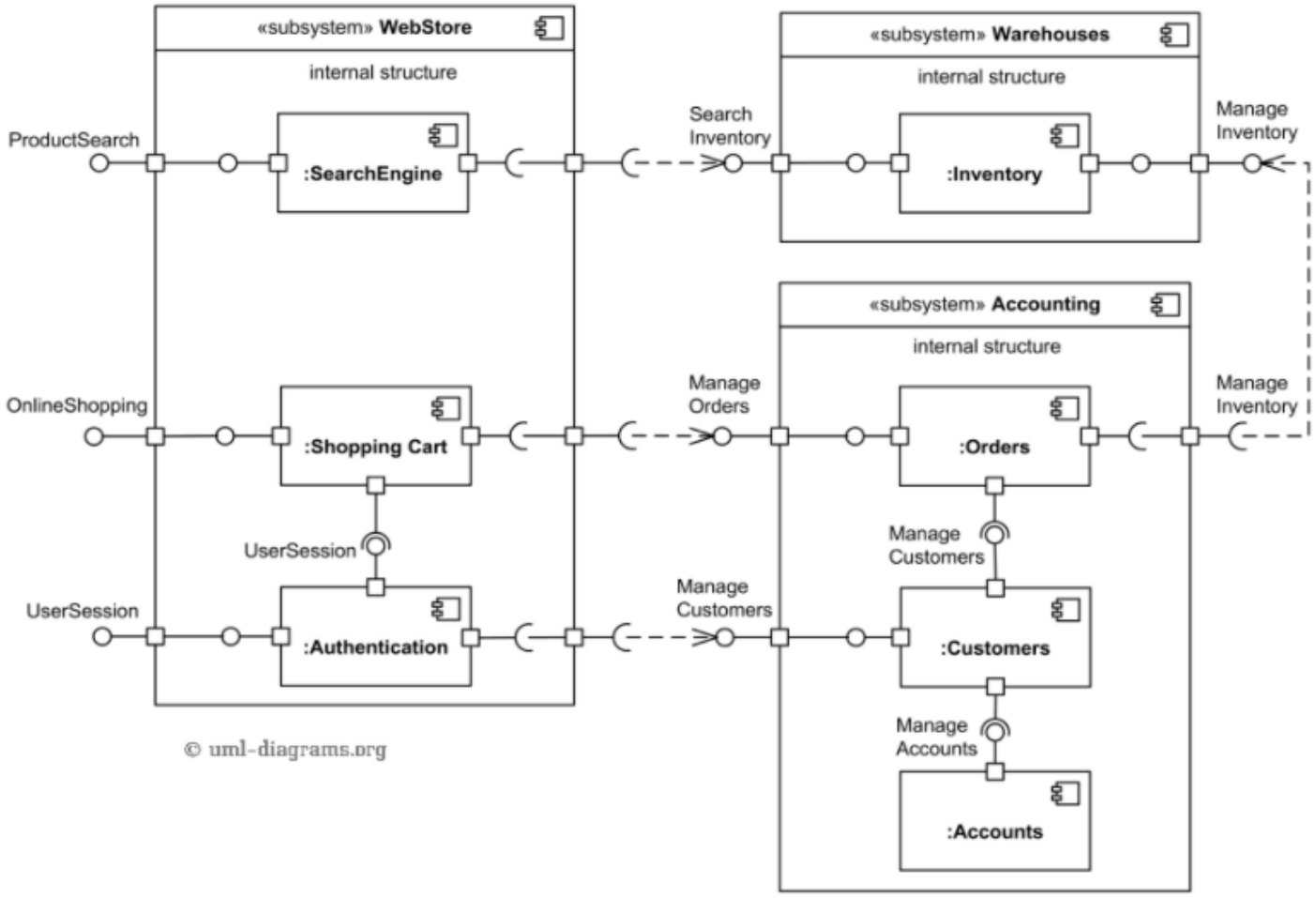
Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE21CS352B: Object Oriented Analysis and Design with Java

Component Model

Prof. Nivedita Kasturi
Department of Computer Science and Engineering

UML Component Diagram



Online shopping UML component diagram example with three related subsystems - WebStore, Warehouses, and Accounting.

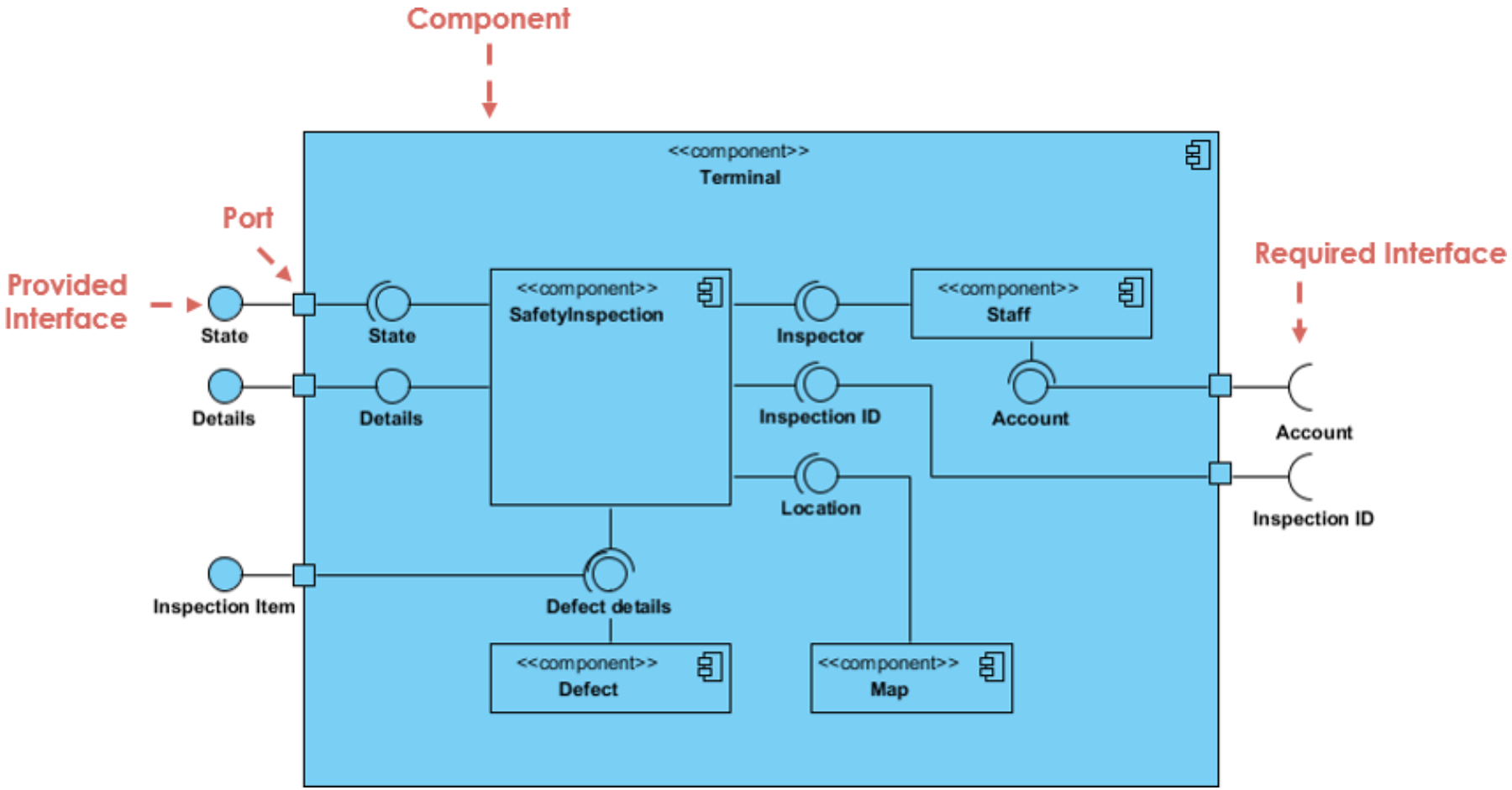
- **Used for modelling large systems into smaller subsystems** which can be easily managed
- Used to represent different components of a system.
- When modelling large OO-systems, break down the system into manageable subsystems.

What is Component in OOAD?

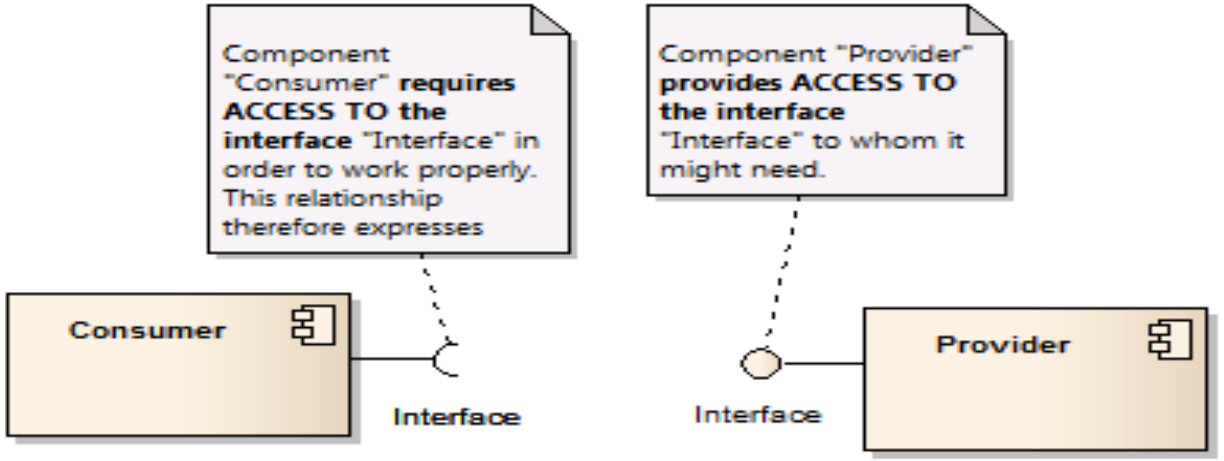
- Component is a *replaceable and executable piece of a system whose implementation details are hidden.*
- Component provides the set of *interfaces that a component realizes* or implements.
- Components require interfaces to carry out a function.
- **It is a modular part of a system that encapsulates its contents.**
- They are the logical elements of a system that plays an essential role during the execution of a system.
- *A component is similar to a **black box** whose external behaviour is defined by a provided interface and required interfaces.*

Component in OOAD?

- A component diagram breaks down the actual system under development into various high levels of functionality.
- *Each component is responsible for one clear aim* within the entire system and only interacts with other essential elements on a need-to-know basis.

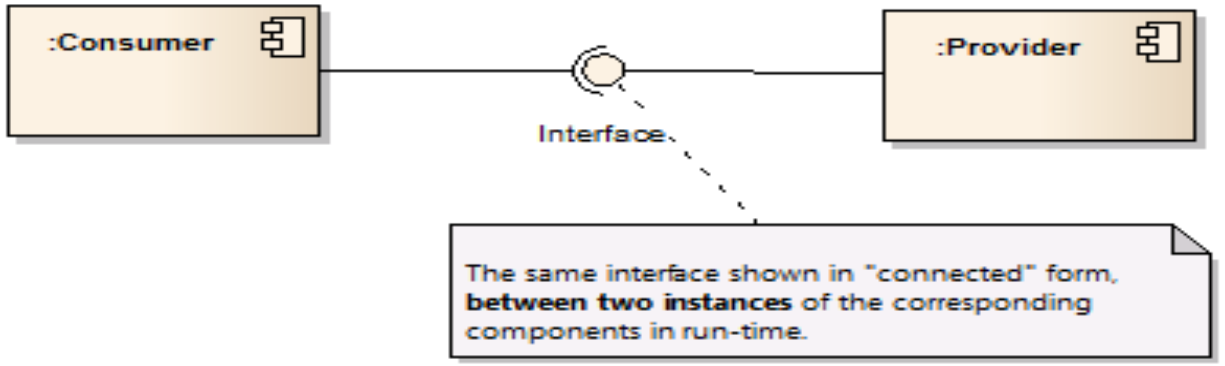


Provided and required interface always refer to the concept of interface, indicating the point of view. The diagrams sheds some light on the subject.



On the implementation level a provided interface is the interface implemented by a class (in the most common sense, e.g. a class B implements the interface I).

Required interface would be any use of an interface by a component (e.g. if a class A defines a method that has the interface I as a parameter, this means that class A has a required interface I).



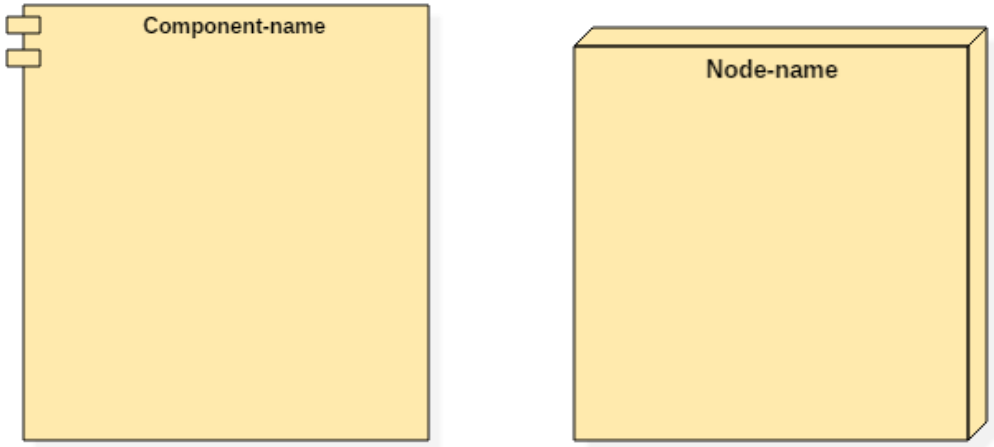
Structure of a UML Component

- A component is represented with classifier rectangle stereotypes as: **<< component >>:**
- *Component details are hidden for the outside world.*
- The name of a component is placed at the center of a rectangle.
- A component icon is displayed at the upper right corner of a rectangle, **which is optional.**

Component Diagram Notations

Component Notation in Component Diagram

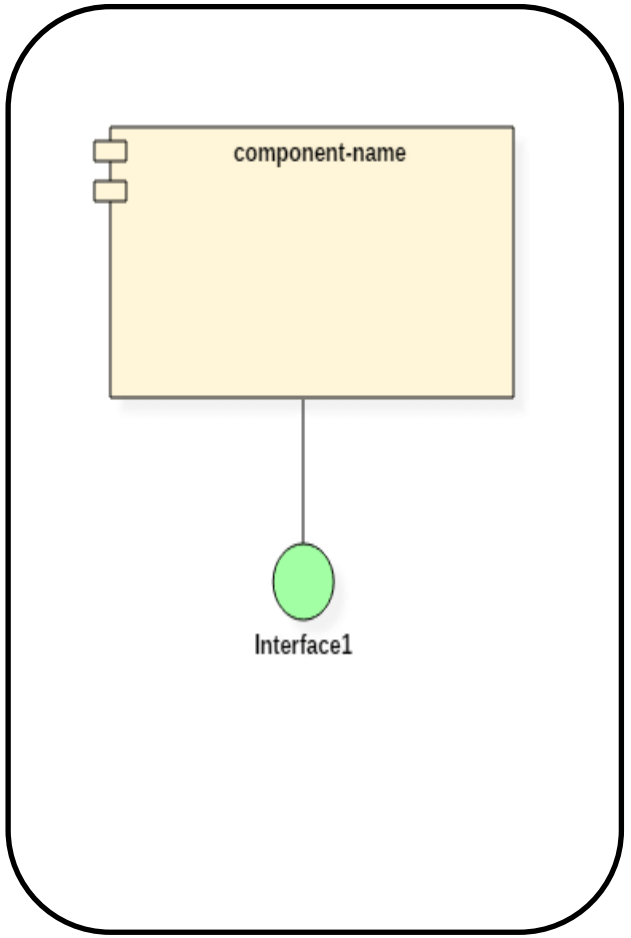
Node Notation in Component Diagram



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

Interface in Component Diagram

- The Interface is a named **set of public features**.
- It separates the *specification of functionality from its implementation by a class diagram* or a subsystem.
- An interface symbol cannot be instantiated.
- It declares a **contract that may be realized** by zero or more classifiers such as a class or a subsystem.
- Anything that realizes an interface accepts the functionalities of the interface and agrees to abide by the contract defined by the interface.
- If the implementation language does not support interfaces, the use of abstract classes, interfaces are named just like classes, in **UpperCamelCase**.



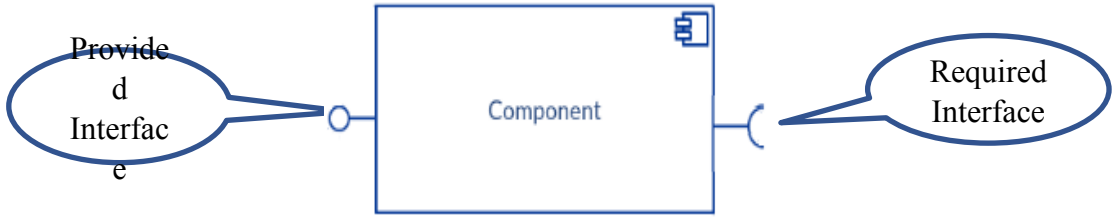
Interface in Component Diagram

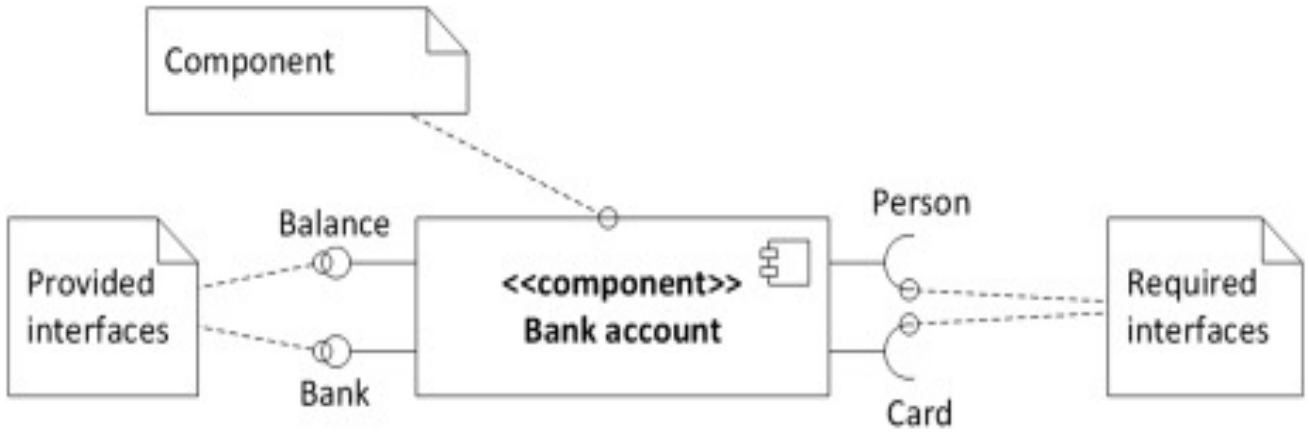
The two types of Interfaces in Component Diagram are **Provided interfaces and Required interfaces**

We can connect provided and required interfaces using assembly connector.

Advantages: It increases the flexibility and extensibility of a class and it decreases the implementation dependencies.

Disadvantages: Extra flexibility leads to complex classes and too many interfaces make systems hard to understand.



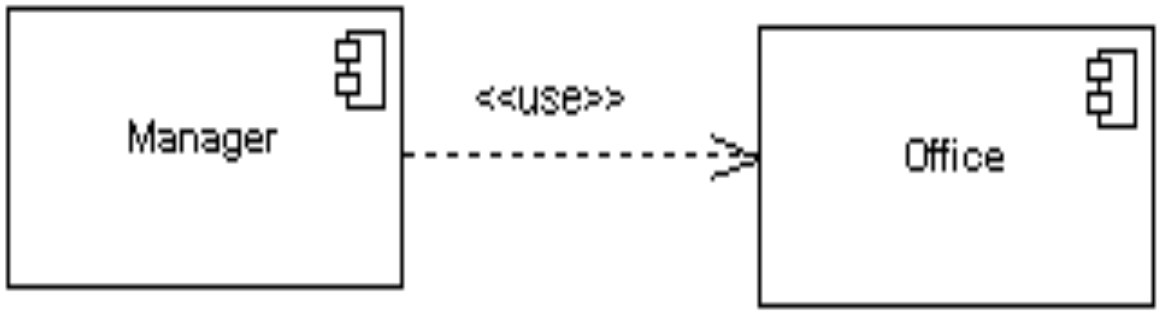


Component Diagram – Usage Connector

Components can be **connected by usage dependencies.**

Usage Dependency

- *A usage dependency is relationship which one element requires another element for its full implementation*
- It is a dependency in which the client requires the presence of the supplier
- **It is shown as a dashed arrow with a <<use>> keyword**
- The arrowhead point from the dependent component to the one of which it is dependent on



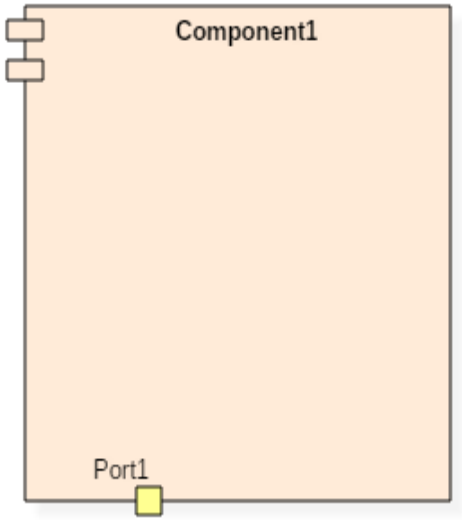
Component Diagram Subsystems

- It is a **component base that acts as a decomposition unit** for larger systems.
- It is a logical construct which is used to break down an extensive system into smaller systems which are known as subsystems.
- This process makes it easy to manage each subsystem efficiently.
- A subsystem cannot be instantiated during runtime, but their contents can be initialized. When subsystems are connected, it creates a single system.

Port in Component Diagram

- **A Port is an interaction point between a classifier and an external environment.**
- It groups semantically cohesive set of provided and required interfaces.
- A port can be used in UML **without specifying the name of the port.**
- A port may have visibility. When a port is drawn over the boundary of a classifier, then it means that the port is **public**. It also means that all the interfaces used are made as public.
- When a port is drawn inside the classifier, then it is either **protected** or **private**.
- A port also has **multiplicity that indicates the number** of instances of the port classifier will have.

A port in UML diagram is denoted as given below:



Here, the port1 is drawn over the boundary, which means it has visibility as public.

How to Draw Component Diagram

A component is nothing but an **executable piece of a system.** Various components together make a single system. Component diagrams are useful during the implementation phase of any system.

Step 1) Before modeling the component diagram, **one must know all the components within the system.** The working of each component should be mentioned. Component diagrams are used to analyze the execution of a system.

Step 2) One should also **explore each component in depth to understand the connection of a component to other** physical artifacts in the system.

Step 3) Identifying the **relationship amongst various artifacts, libraries**, and files are the essential things required during modeling of a component diagram.

Why use Component Diagram?

- These are the **static diagrams** of the unified modelling language.
- **A component diagram is used to represent the structure and organization of components during any instance of time.**
- Component diagrams are used for modelling the subsystems.
- These subsystems collectively represent the entire working view of any system.
- **A single component cannot visualize the whole system**, but the collection of multiple components can.

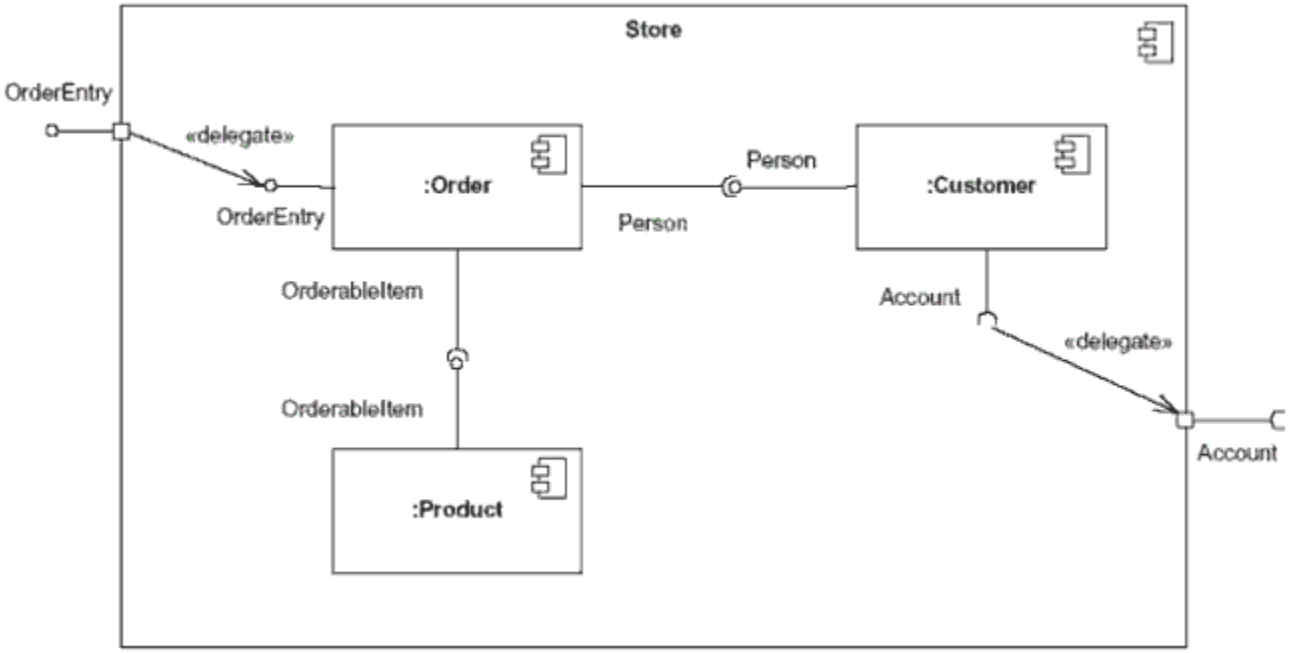
Why use Component Diagram?

So, Component Diagrams are used for:

- To represent the **components of any system at runtime.**
- It helps during **testing of a system.**
- It visualizes the **connection between various components.**

Internal and External view of a Component

Internal View

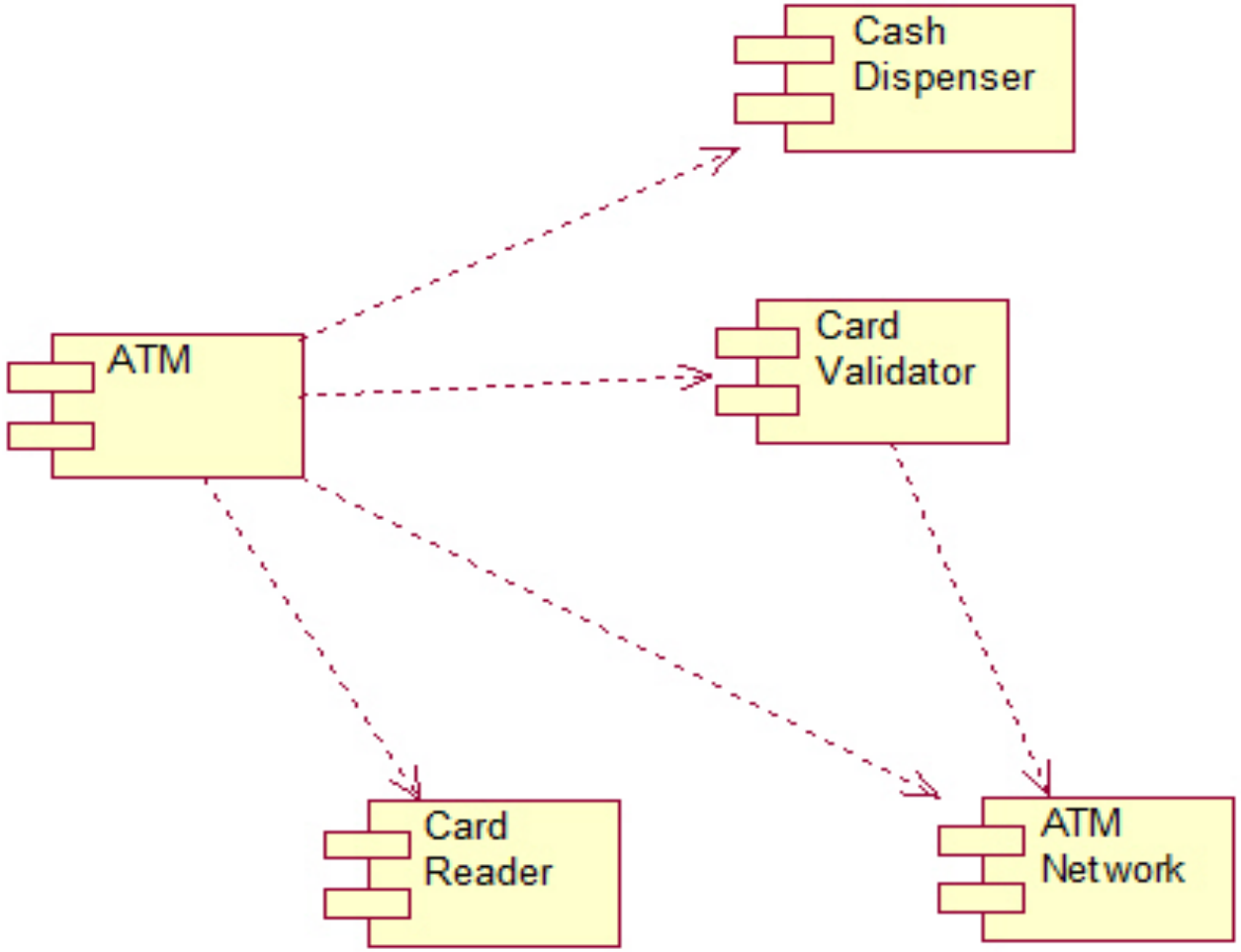


An internal view (or white box view) of a component is where the realizing classes/components are nested within the component shape

External View

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

Example – Component Diagram of an ATM



THANK YOU

Prof. Nivedita Kasturi

Department of Computer Science and Engineering

niveditak@pes.edu

Object Oriented Analysis and Design Using Java

UE21CS352B

Prof. Nivedita Kasturi
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE21CS352B: Object Oriented Analysis and Design with Java

PES UNIVERSITY
CELEBRATING 50 YEARS

Deployment Model

Prof. Nivedita Kasturi
Department of Computer Science and Engineering

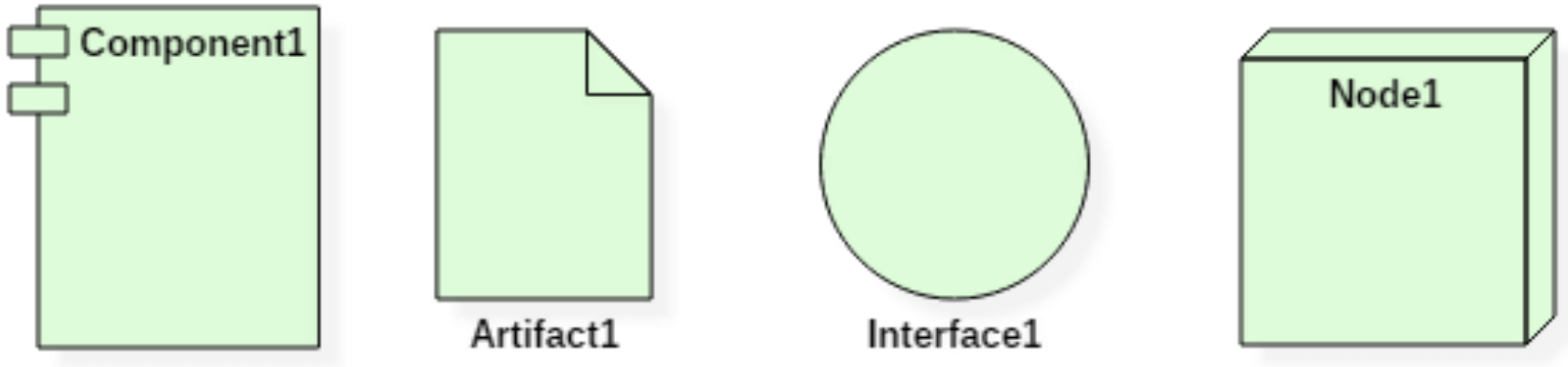
Introduction

- Software components developed as part of the development process would need to be deployed on some set of hardware or software to be executed.
- Deployment diagram maps the software architecture created in design to the physical system architecture that executes it. In distributed systems, it models the distribution of the software across the physical nodes.
- Deployment diagrams shows how the components described in component diagrams are deployed in hardware.
- Deployment diagrams are used to visualize
  - The topology of the physical components of a system, where the software components are deployed.
  - Describe the hardware components used to deploy software components.
  - Describe the runtime processing nodes (physical hardware) .

Deployment Diagram – Notations

- Deployment diagram, models the run-time architecture of a system.
- It shows the configuration of the hardware elements (nodes) and shows how software elements and artifacts are mapped onto those nodes.

Some of the deployment diagram symbols and notations

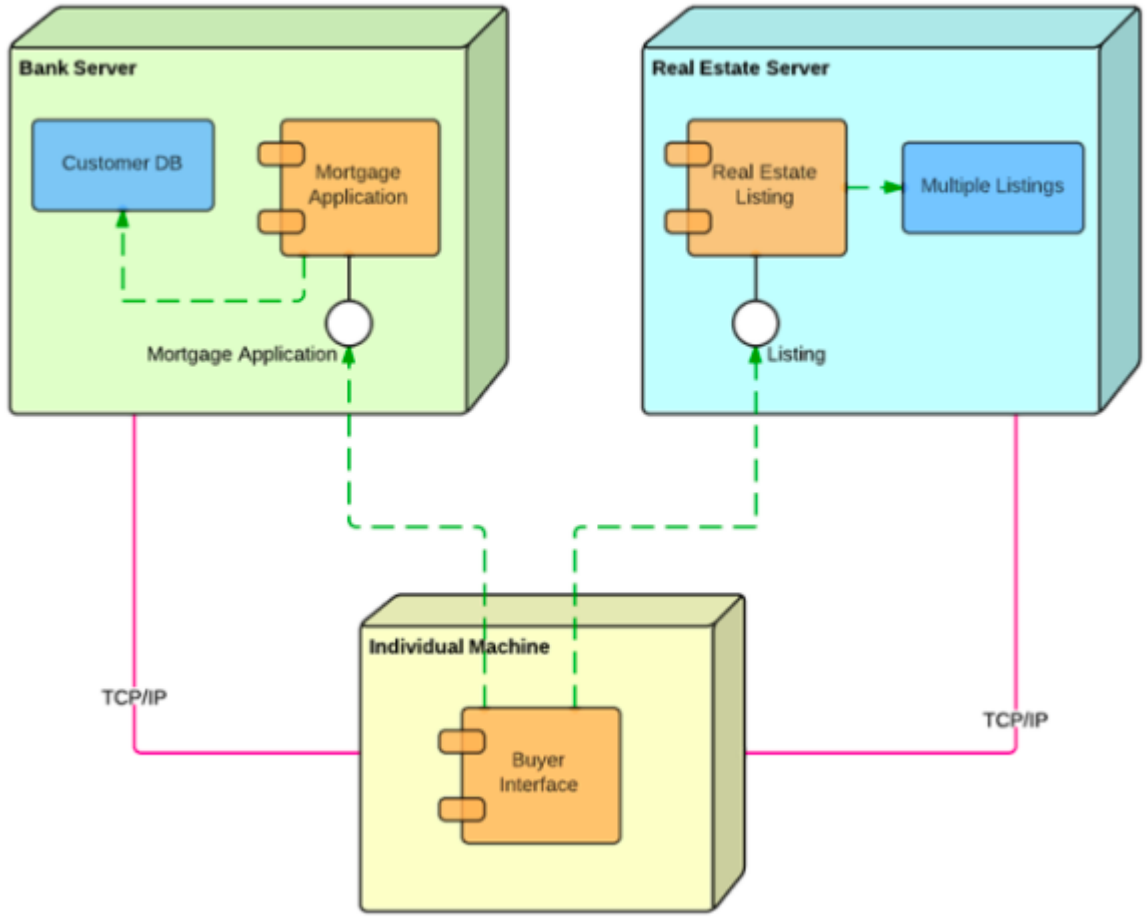


Uses of a Deployment Diagram

- Deployment diagrams are useful for system engineers.
- An efficient deployment diagram is very important as it controls the following parameters −
  - Performance
  - Scalability
  - Maintainability
  - Portability
  - Understandability
- Deployment diagrams are important for visualizing, specifying, and documenting embedded, client/server, and distributed systems and also for managing executable systems through forward and reverse engineering.

Elements in a Deployment Diagram

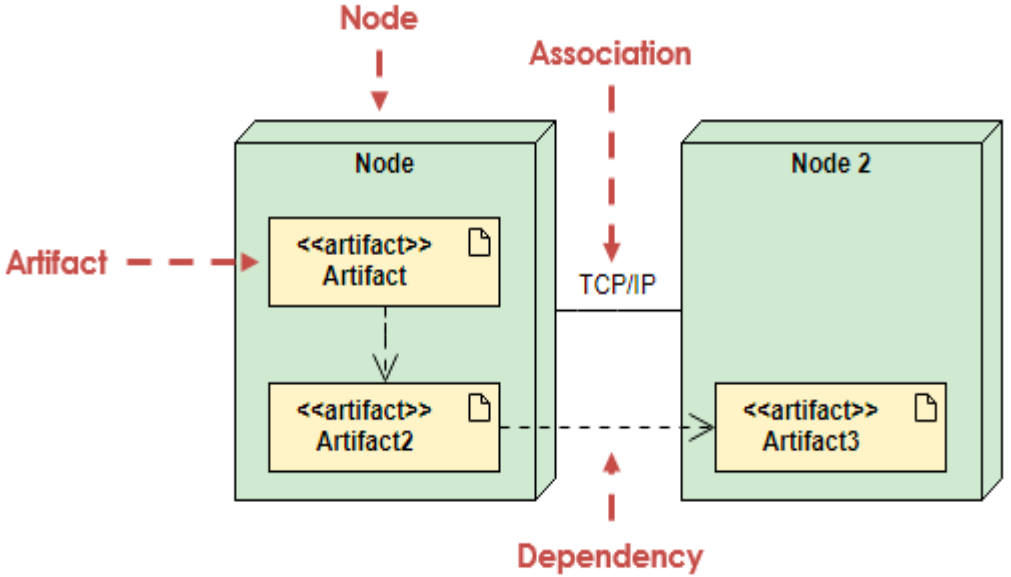
- **Artifact:** A product developed by the software, symbolized by a rectangle with the name and the word "artifact" enclosed by double arrows.
- **Association**: A line that indicates a message or other type of communication between nodes.
- **Component:** A rectangle with two tabs that indicates a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



Artifact

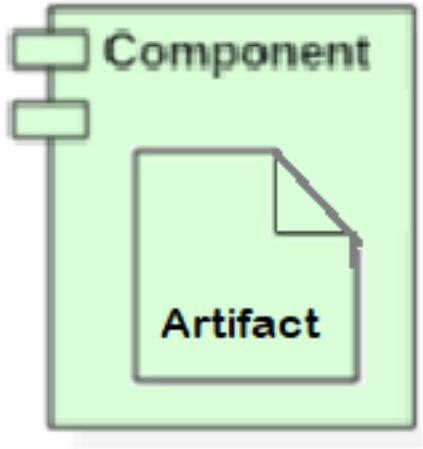
- Artifacts represent concrete elements in the physical world that are the result of a development process. Examples of artifacts are executable files, libraries, archives, database schemas, configuration files, etc.

- Artifacts are concrete elements that are caused by a development process. Examples of artifacts are libraries, archives, configuration files, executable files etc.



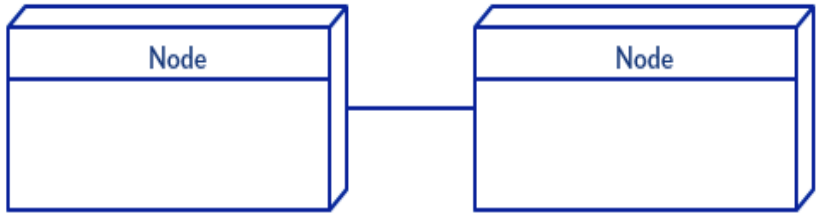
Component with Artifacts

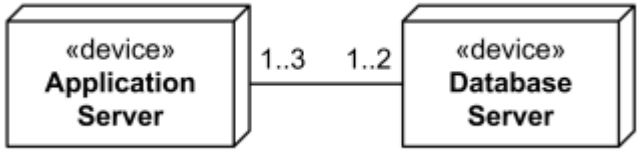
- Component with Artifacts represents the concrete real-world entity related to software development.
- Things that participate in the execution of a system or executable entities that reside in nodes.
- Represent the physical packaging of the logical elements.
- Artifacts are deployed on the nodes.
- Each artifact has a filename in its specification that indicates the physical location of the artifact.



Associations

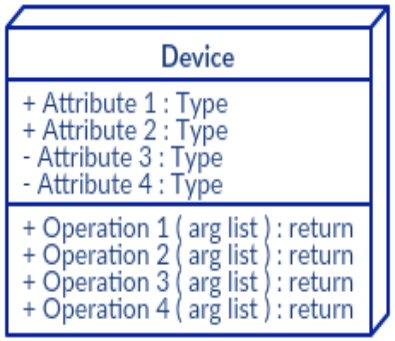
Communication Association

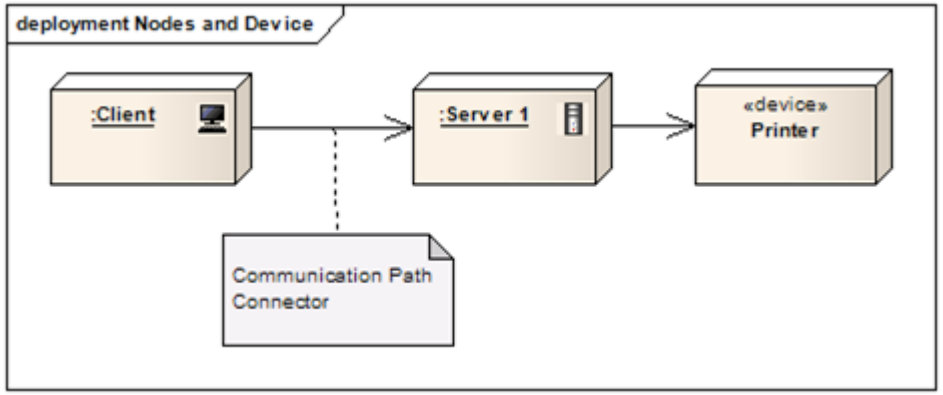




This is represented by a solid line between two nodes. It shows the path of communication between nodes.

Devices

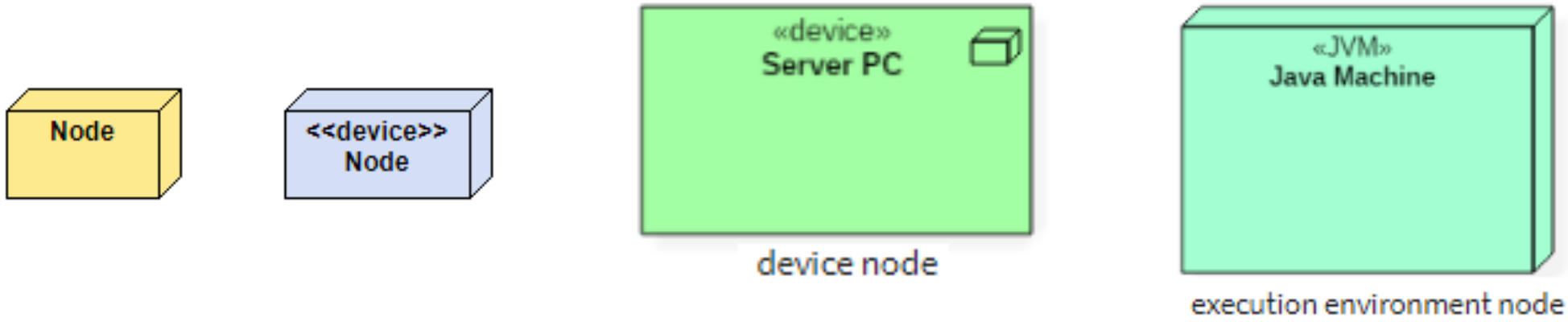




A device is a node that is used to represent a physical computational resource in a system. An example of a device is an application server.

Nodes

- Node : Node is a computational resource upon which component artifacts are deployed for execution.



- There are two types of Nodes:
  - **Device Node**
  - **Execution Environment Node**

Device nodes are physical computing resources with processing memory and services to execute software, such as typical computers or mobile phones.

An execution environment node (EEN) is a software computing resource that runs within an outer node and which itself provides a service to host and execute other executable software elements.

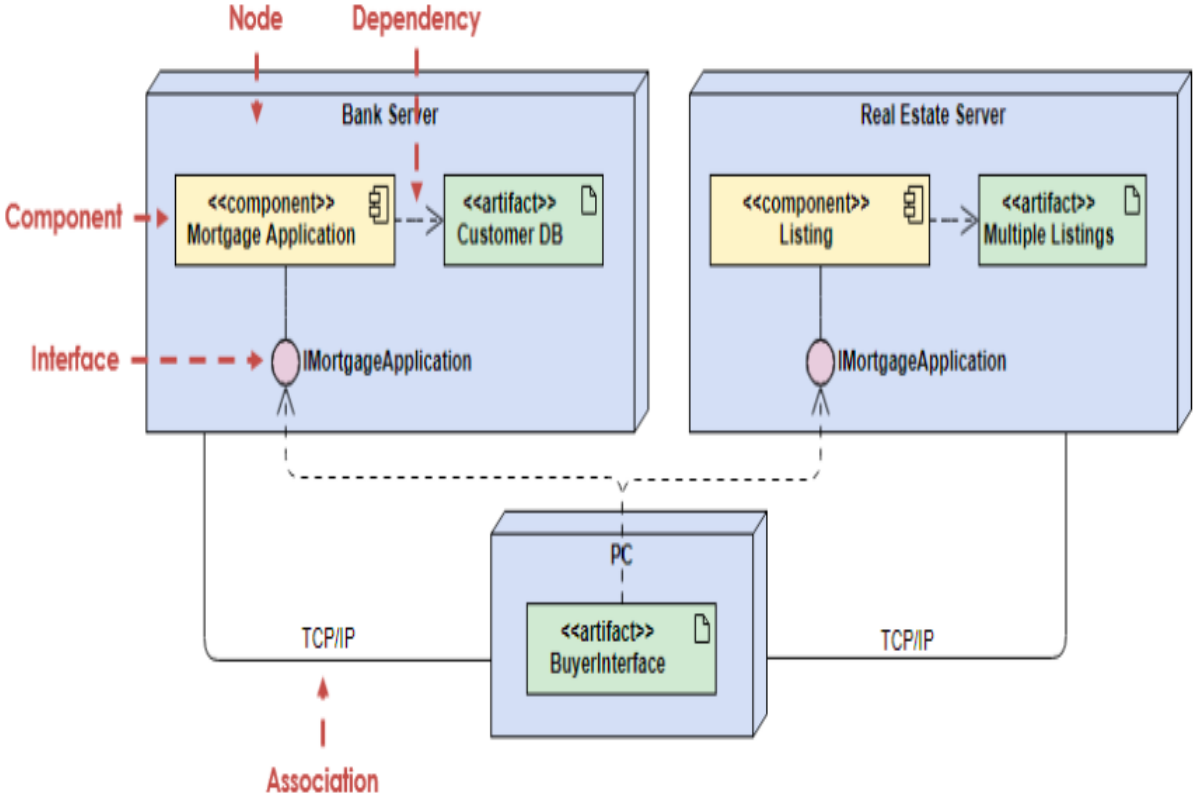
How to Draw deployment Diagram

Step 1: Identify the purpose of your deployment diagram. And to do so, you need to identify the nodes and devices within the system you'll be visualizing with the diagram.

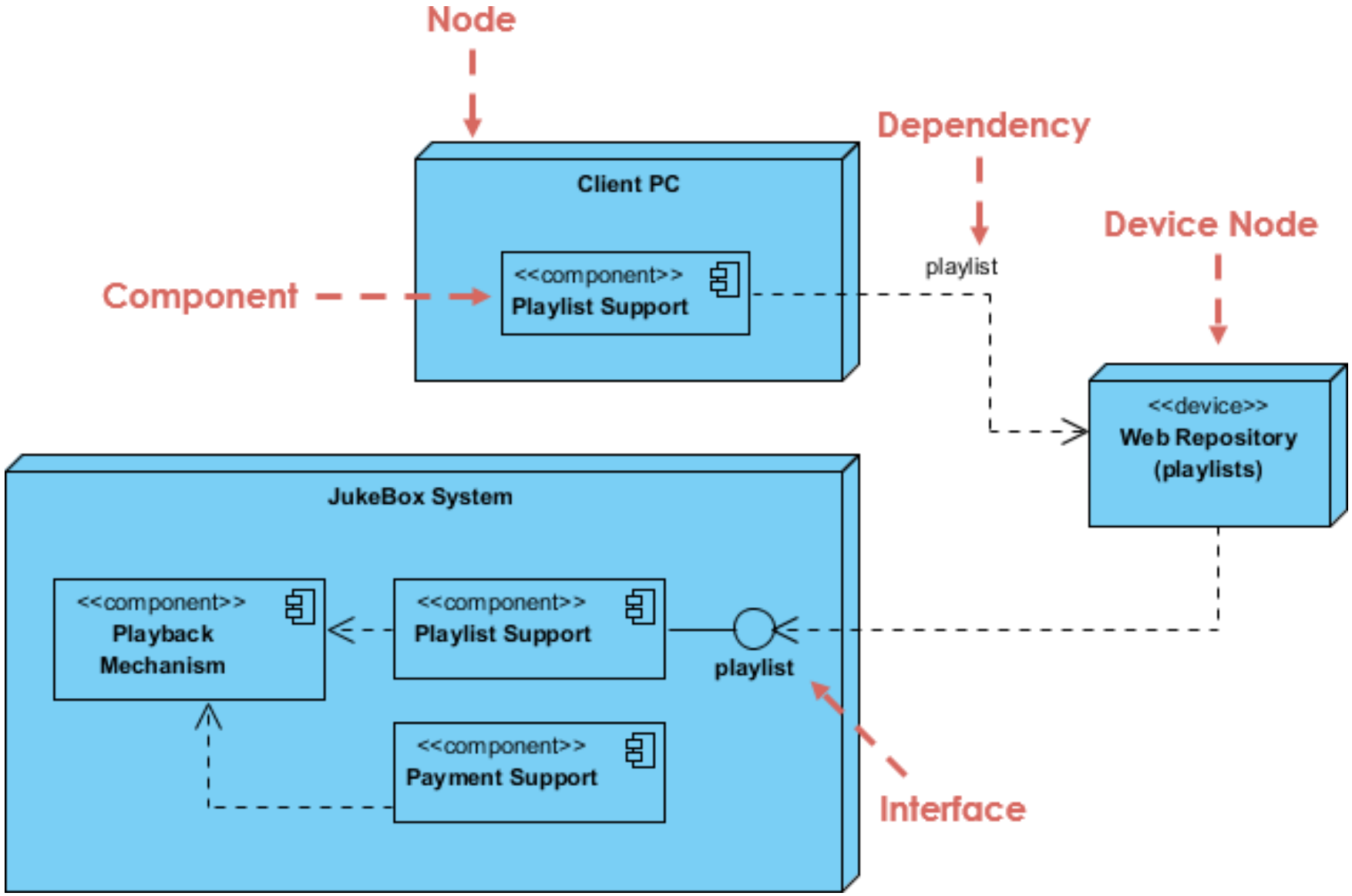
Step 2: Figure out the relationships between the nodes and devices. Once you know how they are connected, proceed to add the communication associations to the diagram.

Step 3: Identify what other elements like components, active objects you need to add to complete the diagram.

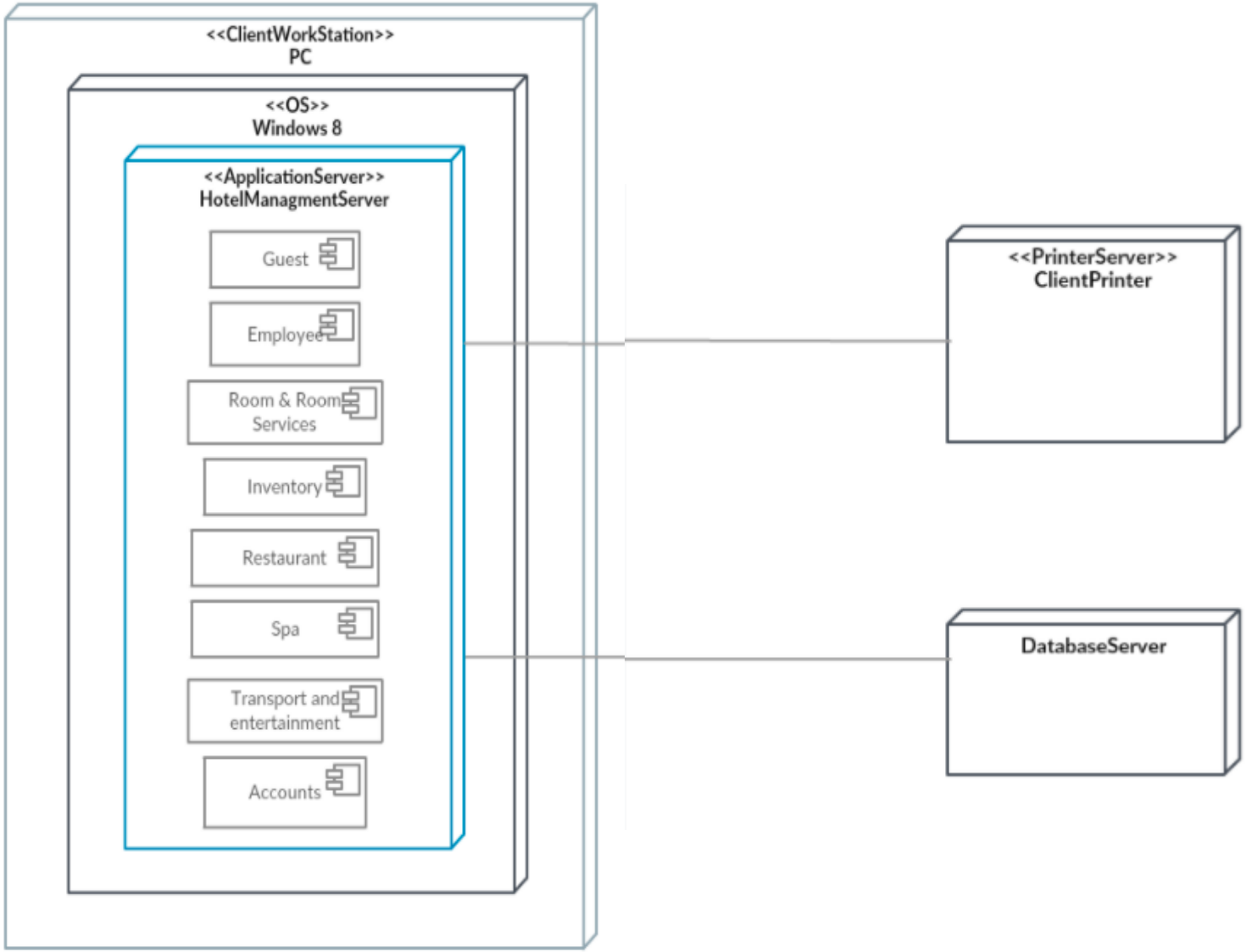
Step 4: Add dependencies between components and objects as required.



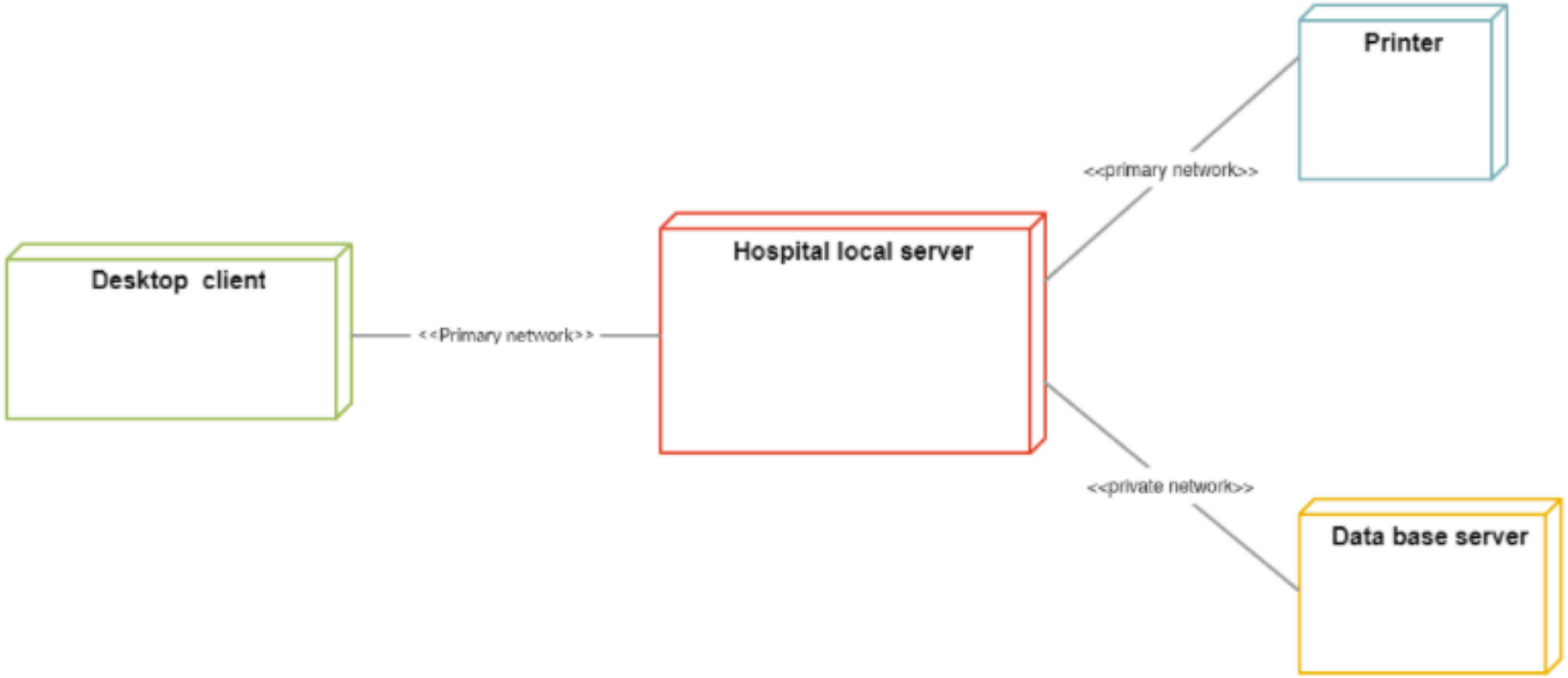
Example – Jukebox



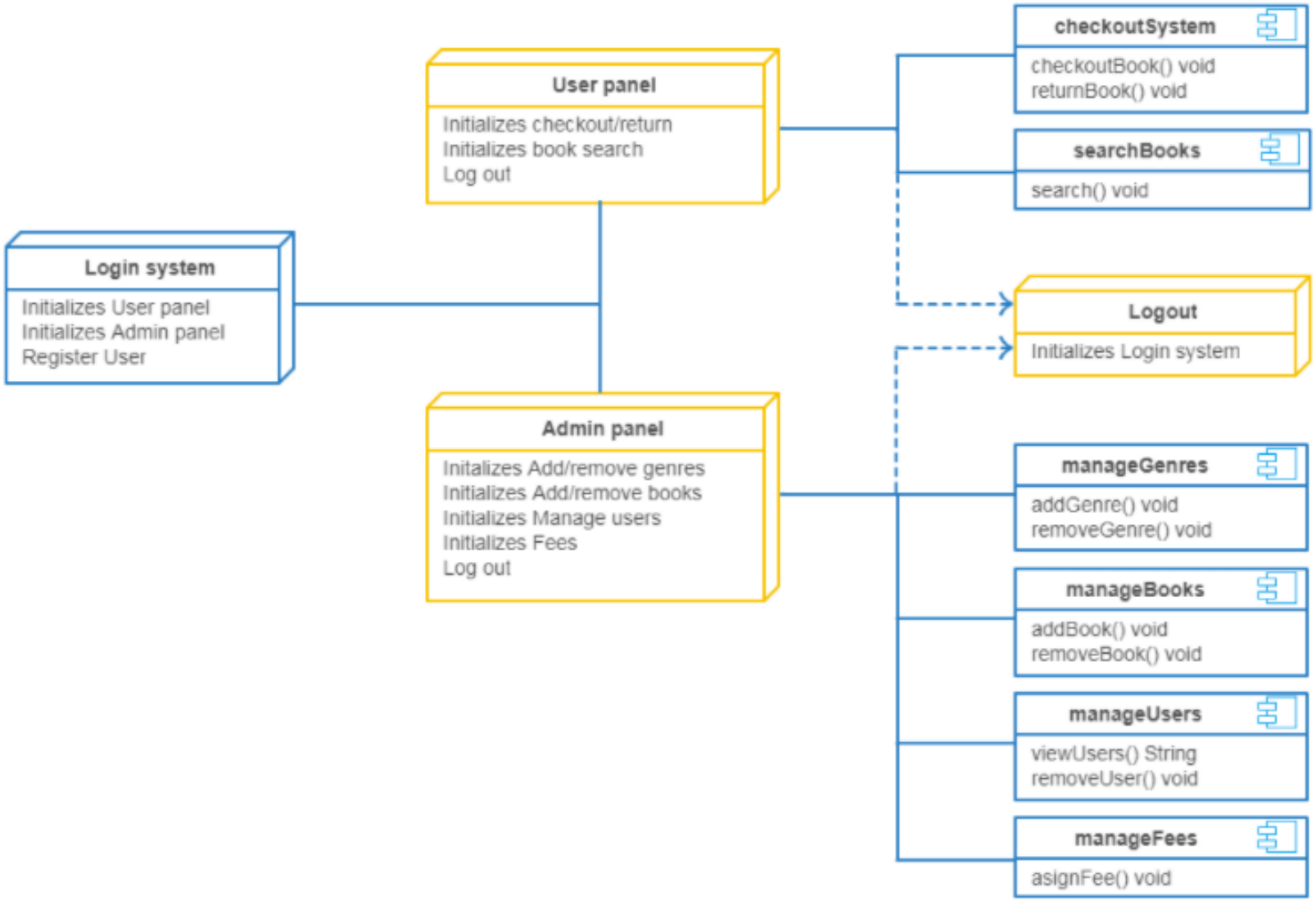
Example – Hotel Management System



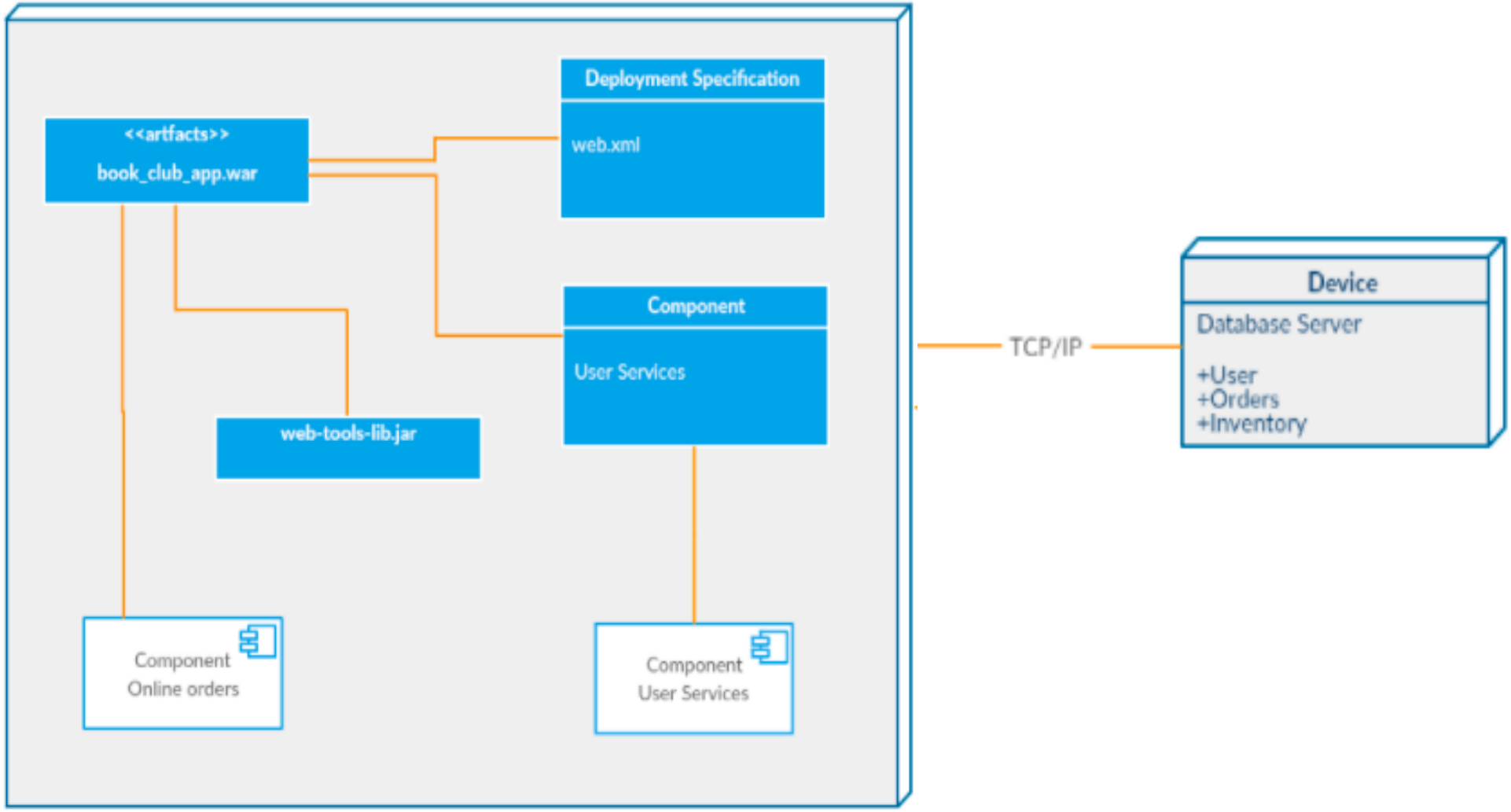
Example – Hospital Management System



Example – Library Management System



Example – Online Shopping System



Example – Modeling a Distributed System

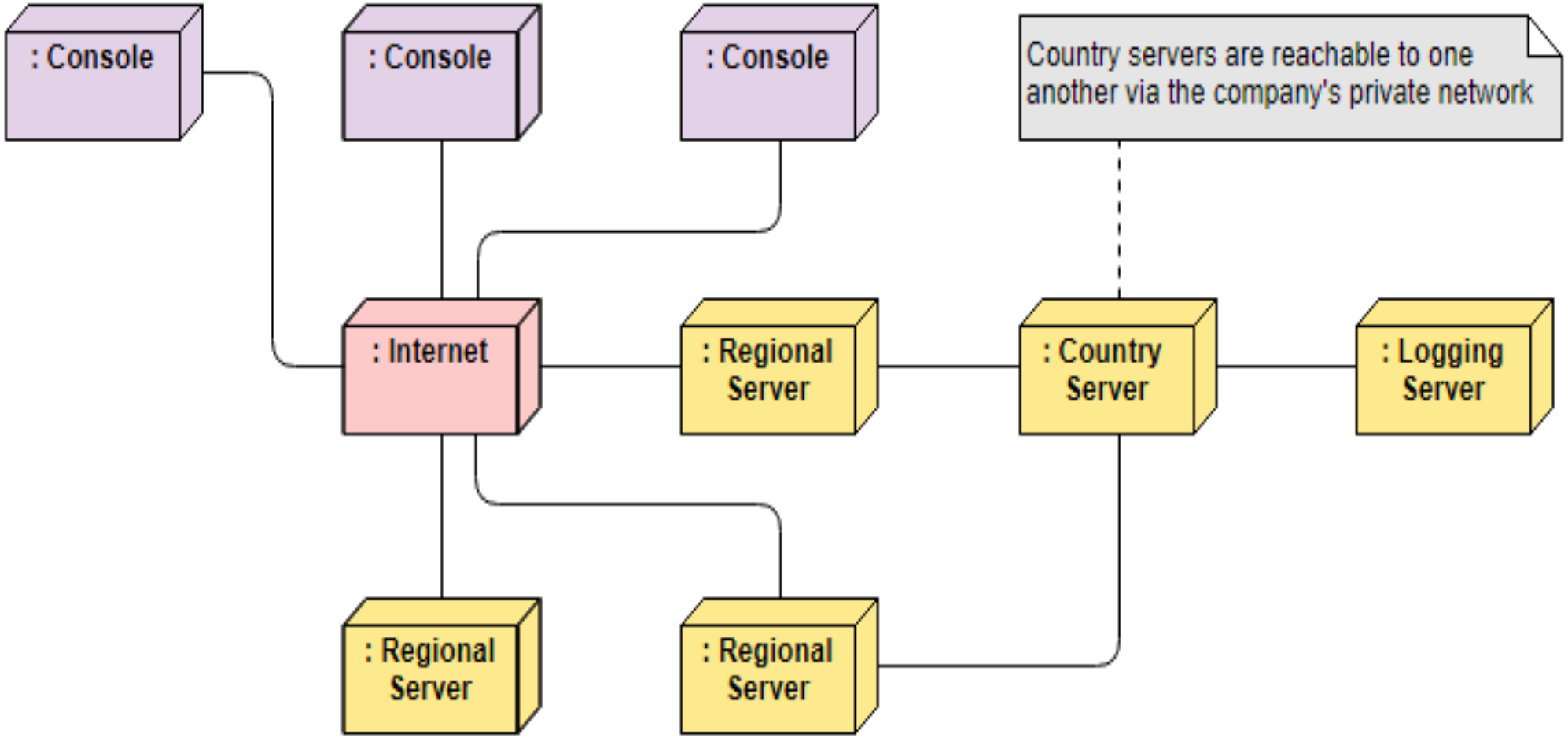
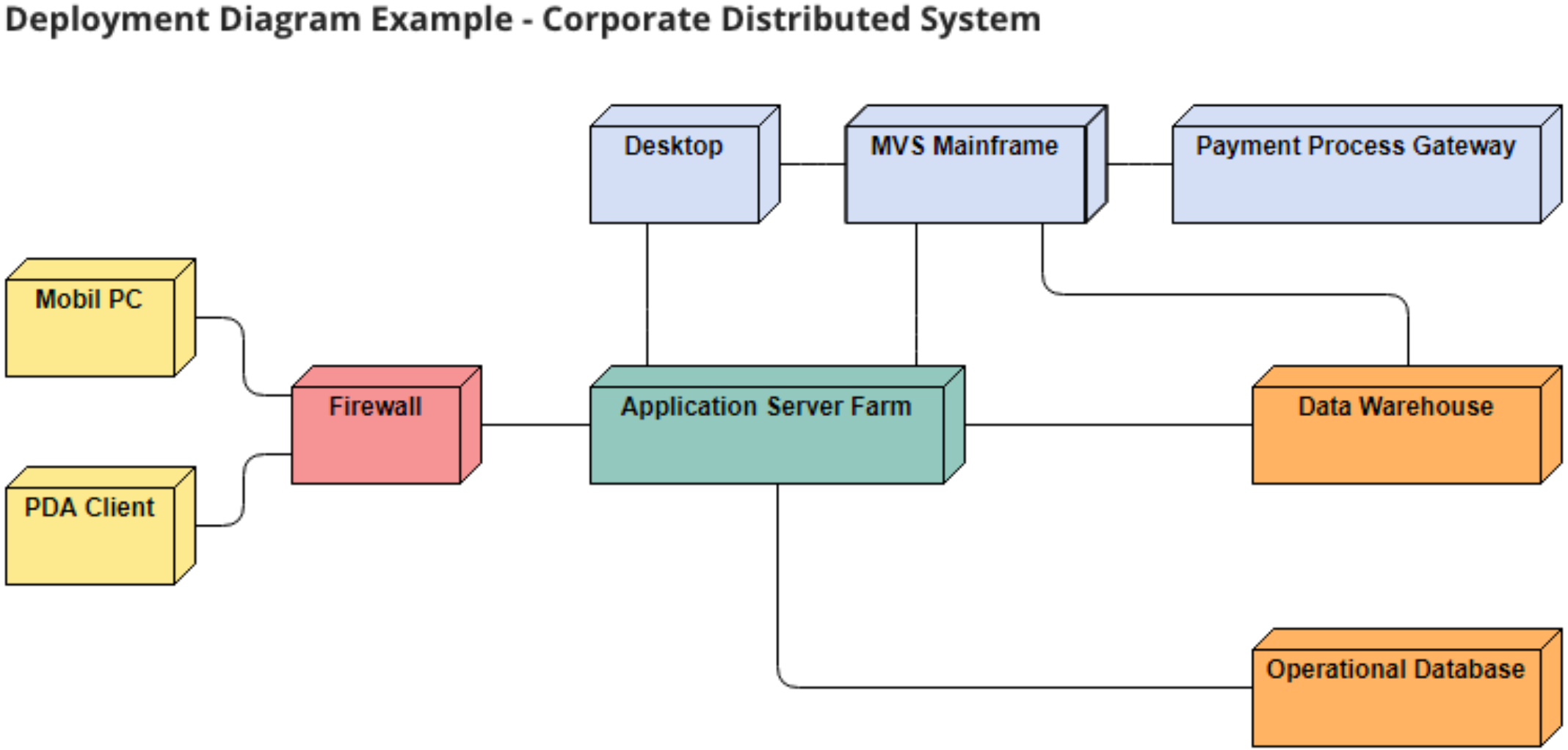


Illustration of a Deployment Diagram



THANK YOU

Prof. Nivedita Kasturi

Department of Computer Science and Engineering

Niveditak@pes.edu

Object Oriented Analysis and Design using Java

Dr. Kamatchi Priya L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Object Oriented Analysis and Design using Java

Activity Modelling: UML Activity Diagrams, Modelling and Guidelines

Dr. Kamatchi Priya L

Department of Computer Science and Engineering

Interaction Modeling

The interaction model describes how objects interact to produce useful results. It is a holistic view of behavior across many objects

Interactions can be modeled at different levels of abstraction.

- Use Case Model
- Sequence Model
- Activity Model

Interaction Modeling

Use Case Model:

At a high level, use cases describe how a system interacts with outside actors. Each use case represents a piece of functionality that a system provides to its users. Use cases are helpful for capturing informal requirements.

Sequence Model:

Sequence diagrams provide more detail and show the messages exchanged among a set of objects over time. Messages include both asynchronous signals and procedure calls. Sequence diagrams depicts the behavior sequences seen by users of a system.

Interaction Modeling

Activity Model:

- Activity diagrams provide further detail and show the flow of control among the steps of a computation.
- Activity diagrams can show data flows as well as control flows.
- Activity diagrams document the steps necessary to implement an operation or a business process referenced in a sequence diagram.

Activity Diagram

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

Activity Diagram: Notations and Symbols

Initial State or Start Point

A small filled circle followed by an arrow represents the initial action state or the start point for any activity diagram. For activity diagram using swimlanes, make sure the start point is placed in the top left corner of the first column.

Start Point/Initial State

Activity Diagram: Notations and Symbols

Activity or Action State

An action state represents the non-interruptible action of objects. A rectangle with rounded corners represent an activity or an action state.

Activity

Activity Diagram: Notations and Symbols

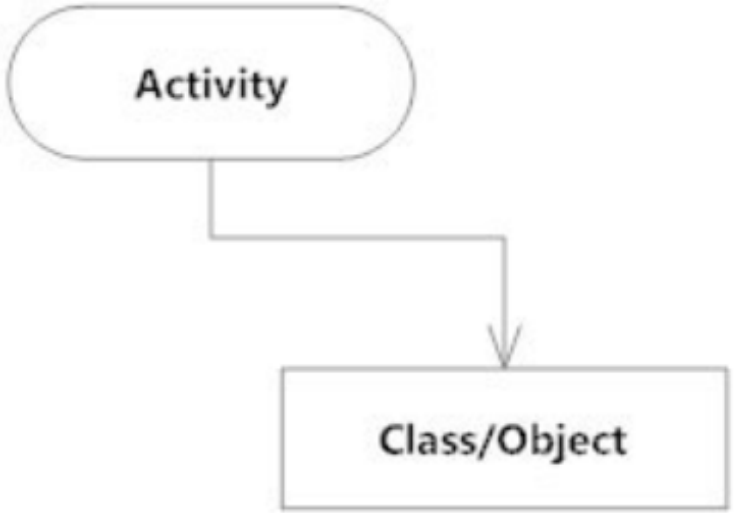
Action Flow:

Action flows, also called edges and paths, illustrate the transitions from one action state to another. They are usually drawn with an arrowed line.

Activity Diagram: Notations and Symbols

Object Flow:

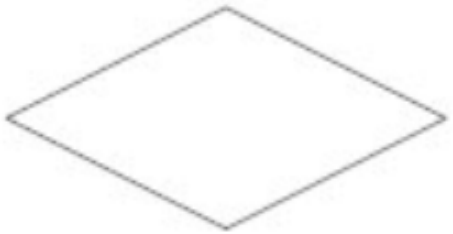
- Object flow refers to the creation and modification of objects by activities.
- An object flow arrow from an action to an object means that the action creates or influences the object.
- An object flow arrow from an object to an action indicates that the action state uses the object.



Activity Diagram: Notations and Symbols

Decisions and Branching:

A diamond represents a decision with alternate paths. When an activity requires a decision prior to moving on to the next activity, add a diamond between the two activities. The outgoing alternates should be labeled with a condition or guard expression. One of the paths can be labeled as "else“.

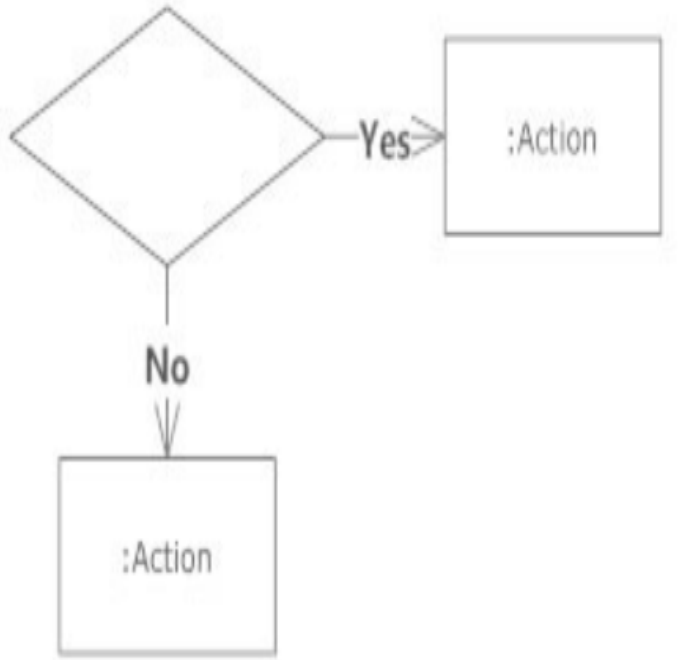


Decision Symbol

Activity Diagram: Notations and Symbols

Guards:

In UML, guards are a statement written next to a decision diamond that must be true before moving next to the next activity.
These are not essential, but are useful when a specific answer, such as "Yes, three labels are printed," is needed before moving forward.

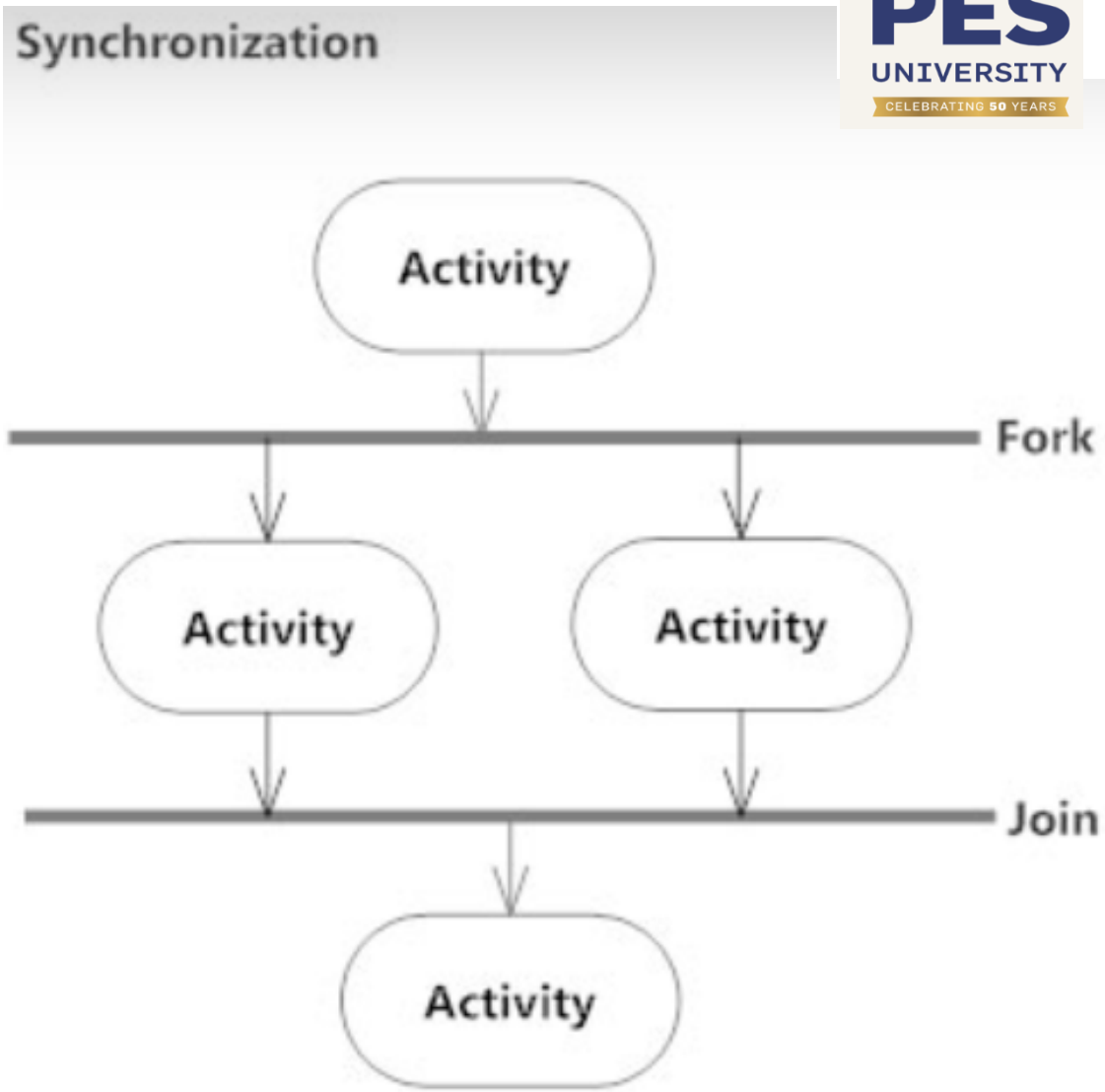


Guard Symbols

Activity Diagram: Notations and Symbols

Synchronization :

- A fork is used to split a single incoming flow into multiple concurrent flows. It is represented as a straight, slightly thicker line in an activity diagram.
- A join is used to join multiple concurrent flows back into a single outgoing flow.
- A fork and join mode used together are often referred to as synchronization. It helps to show parallel occurrence of activities.



Activity Diagram: Notations and Symbols

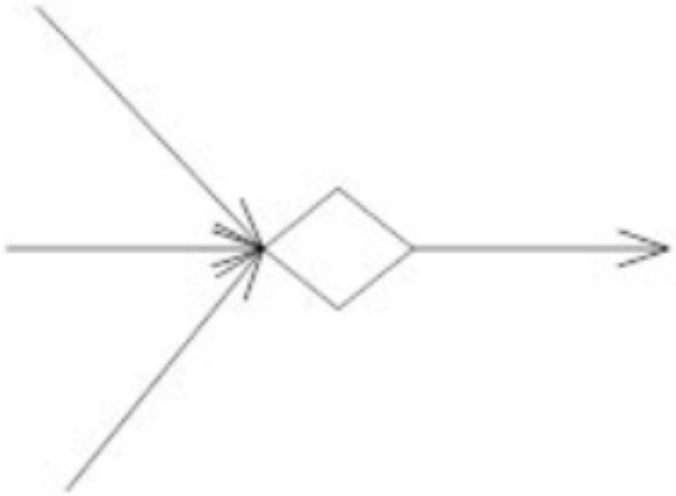
Time Event:

- This refers to an event that stops the flow for a time
- An hourglass depicts it.

Activity Diagram: Notations and Symbols

Merge Event:

A merge event brings together multiple flows that are not concurrent.



Merge

Activity Diagram: Notations and Symbols

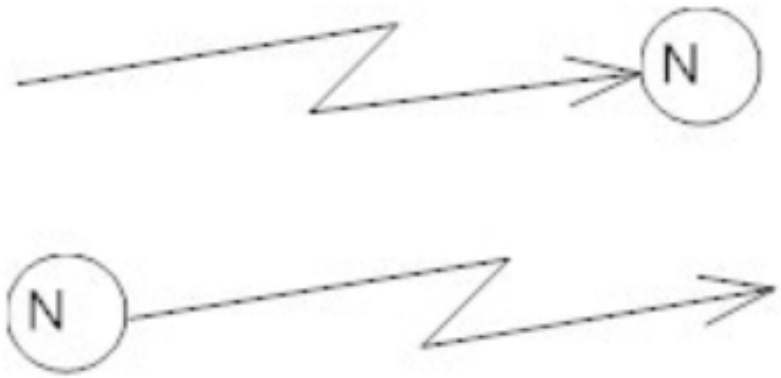
Sent and Received Signals

- Signals represent how activities can be modified from outside the system
- They usually appear in pairs of sent and received signals, because the state can't change until a response is received
- For example, an authorization of payment is needed before an order can be completed

Activity Diagram: Notations and Symbols

Interrupting Edge:

An event, such as a cancellation, that interrupts the flow denoted with a lightning bolt.

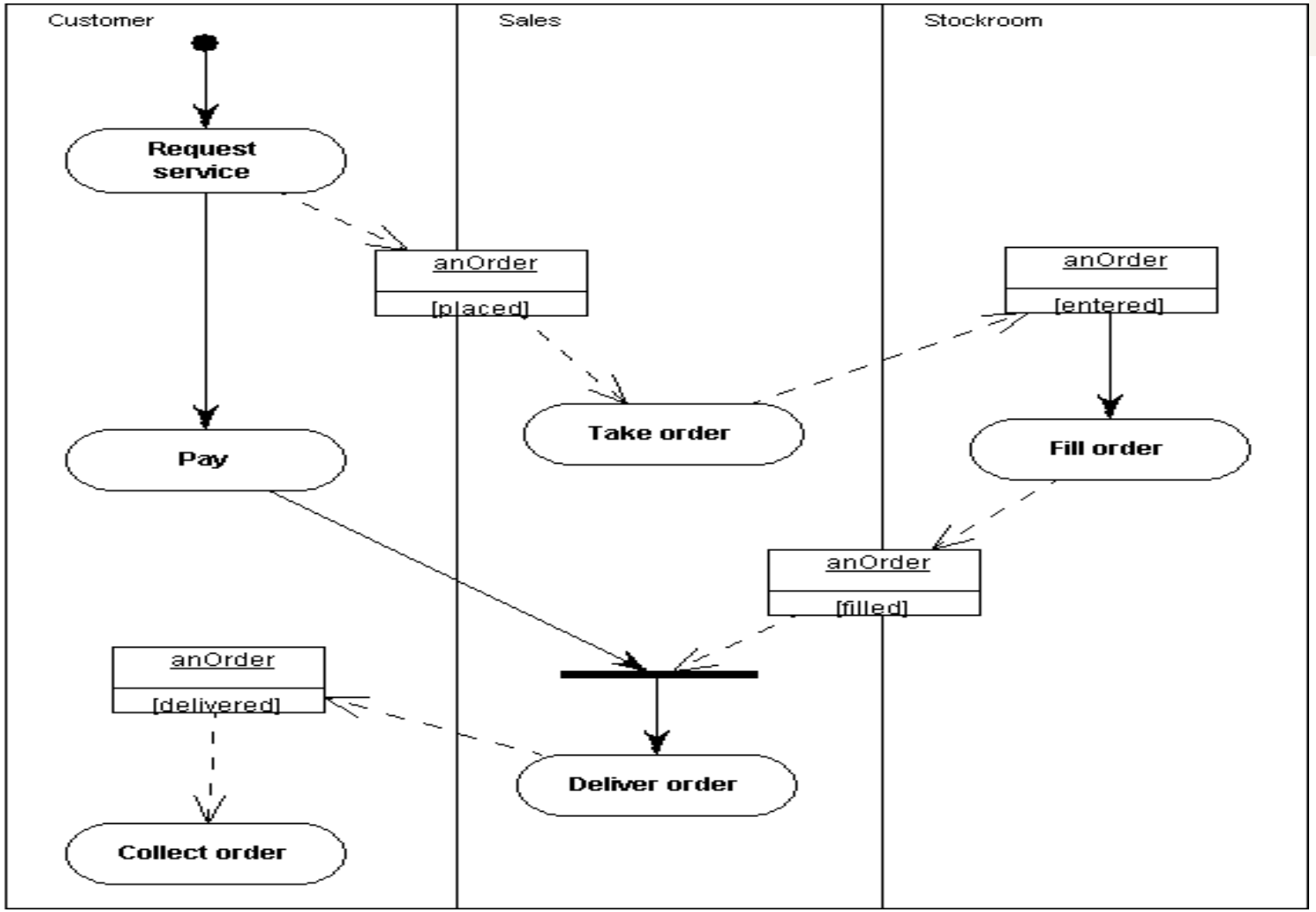


Interrupting Edge Symbols

Activity Diagram: Notations and Symbols

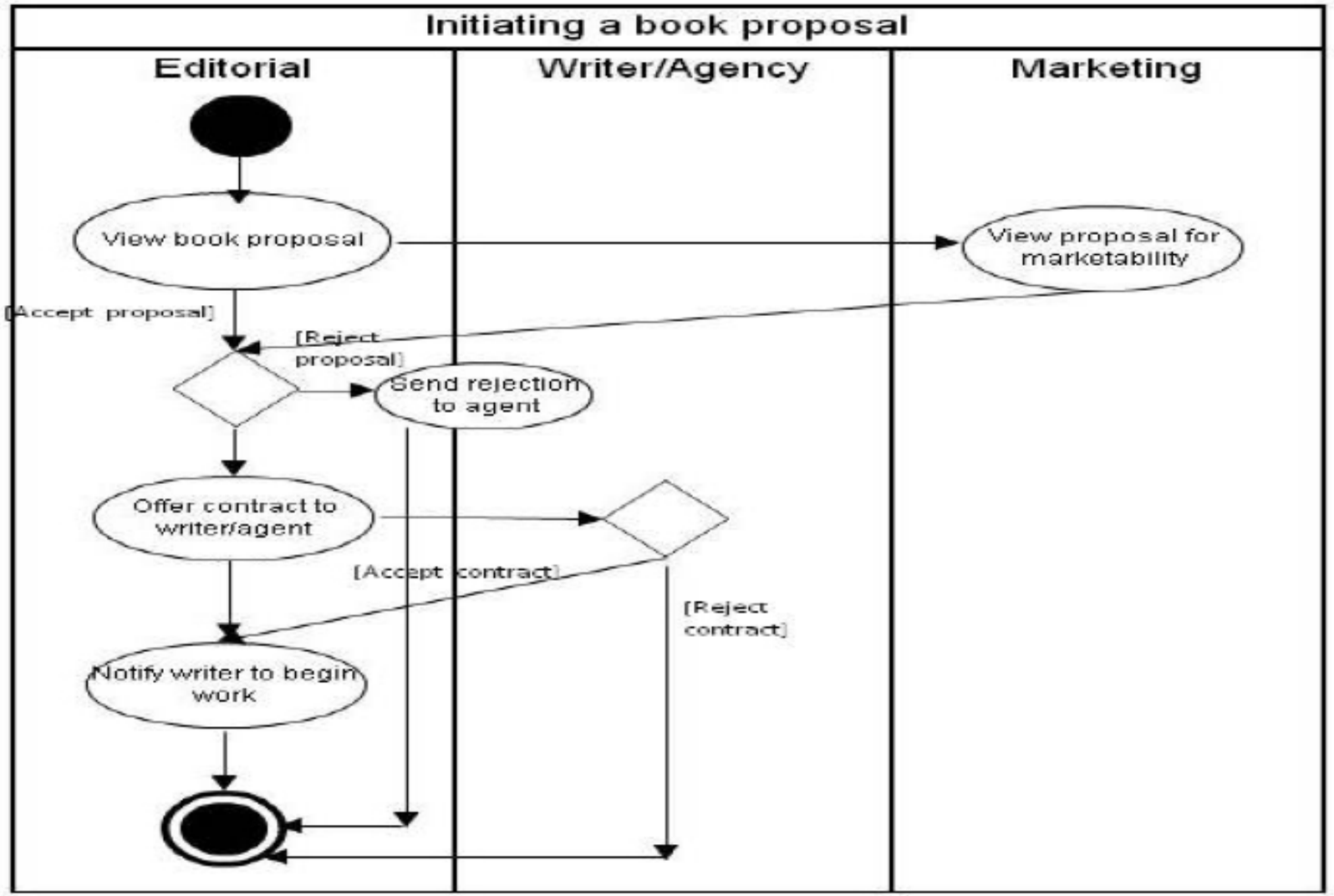
Swimlanes:

Swimlanes group related activities into one column



Activity Diagram: Notations and Symbols

Swimlanes:



Activity Diagram: Notations and Symbols

Final State or End Point

An arrow pointing to a filled circle nested inside another circle represents the final action state.

End Point Symbol

Steps to Construct an Activity Diagram

Step 1: Figure out the action steps from the use case

Here you need to identify the various activities and actions your business process or system is made up of.

Step 2: Identify the actors who are involved

If you already have figured out who the actors are, then it's easier to discern each action they are responsible for.

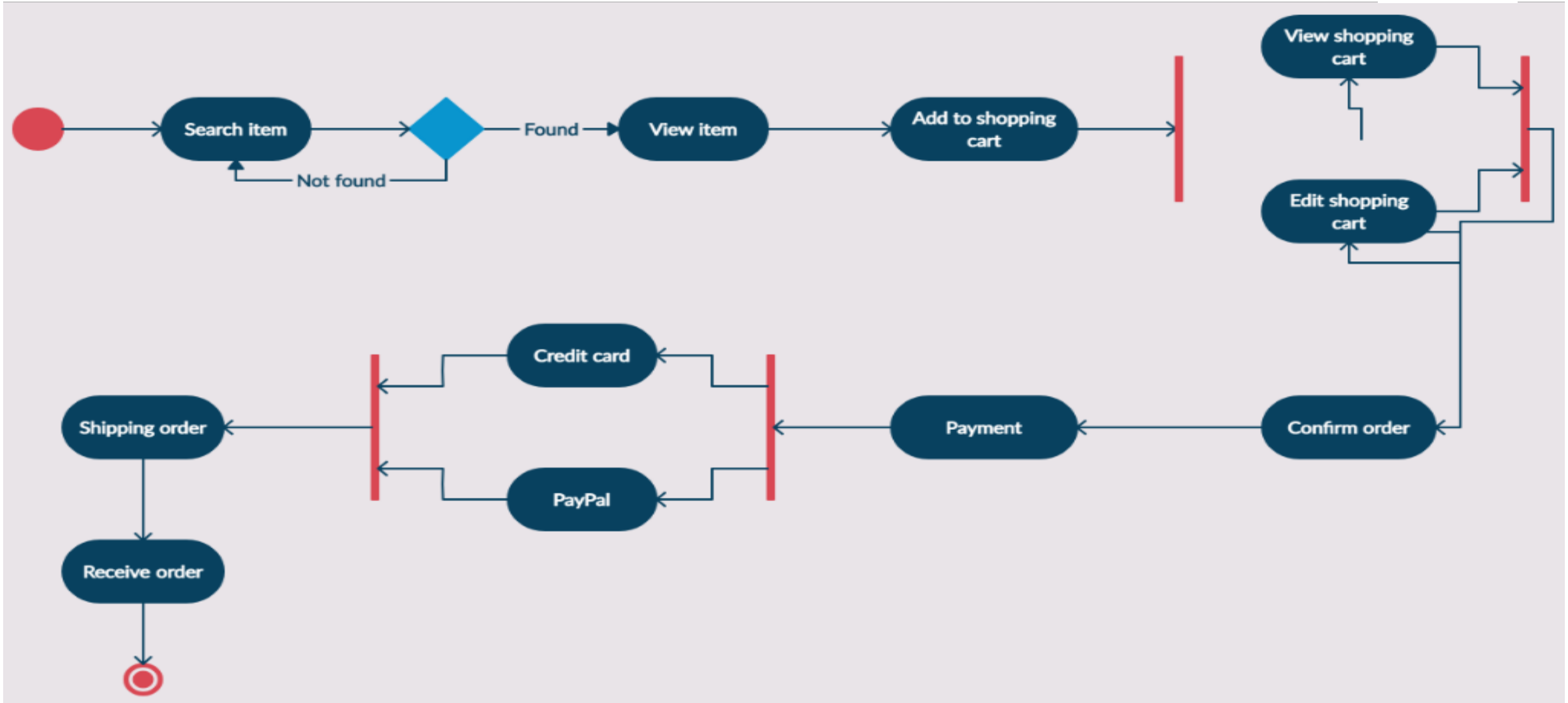
Step 3: Find a flow among the activities

Figure out in which order the actions are processed. Mark down the conditions that have to be met in order to carry out certain processes (which actions occur at the same time, whether you need to add any branches in the diagram, do you have to complete some actions before you can proceed to others?).

Step 4: Add swimlanes

Figure out who is responsible for each action. And assign them to a swimlanes by grouping each action the objects are responsible for, under them.

Online Shopping System



THANK YOU

Dr. Kamatchi Priya L

Department of Computer Science and Engineering

priyal@pes.edu

Object Oriented Analysis and Design with Java

Dr. Kamatchi Priya L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Object Oriented Analysis and Design with Java

Behaviour Modelling: Sequence Diagram

Dr. Kamatchi Priya L

Department of Computer Science and Engineering

Behaviour Modelling

- **UML Behavioral Diagrams Overview:**
  Illustrate elements in a system dependent on time, conveying dynamic concepts and their interrelationships.

- **Nature of Elements:**
  Elements in these diagrams resemble verbs in natural language, representing dynamic actions.

- **Temporal Relationships:**
  Relationships among elements typically signify the passage of time.

- **Example Scenario:**
  A behavioral diagram of a vehicle reservation system might contain elements such as Make a Reservation, Rent a Car, and Provide Credit Card Details

Sequence Diagram

A Sequence diagram is a structured representation of behavior as a series of sequential steps over time. You can use it to:

- Depict workflow, Message passing and how elements in general cooperate over time to achieve a result
- Capture the flow of information and responsibility throughout the system, early in analysis; Messages between elements eventually become method calls in the Class model
- Make explanatory models for Use Case scenarios; by creating a Sequence diagram with an Actor and elements involved in the Use Case, you can model the sequence of steps the user and the system undertake to complete the required tasks

Sequence Diagram Construction

- Sequence elements are arranged in a horizontal sequence, with Messages passing back and forward between elements
- Messages on a Sequence diagram can be of several types; the Messages can also be configured to reflect the operations and properties of the source and target elements (see the Notes in the *Message* Help topic)
- An Actor element can be used to represent the user initiating the flow of events
- Stereotyped elements, such as Boundary, Control and Entity, can be used to illustrate screens, controllers and database items, respectively
- Each element has a dashed stem called a Lifeline, where that element exists and potentially takes part in the interactions

Sequence Diagram: Notations and Symbols

Actor:

- An Actor is a user of the system; user can mean a human user, a machine, or even another system or subsystem in the model. Anything that interacts with the system from the outside or system boundary is termed an Actor. Actors are typically associated with Use Cases.

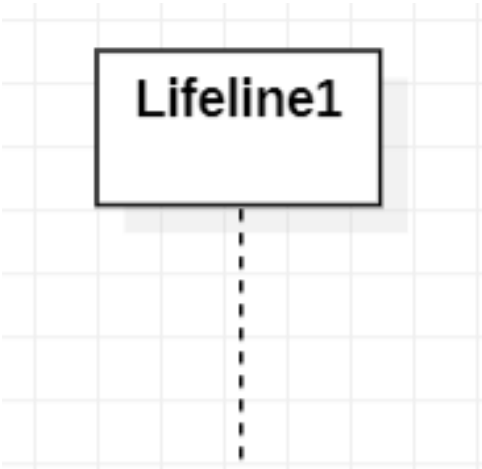
- Actors can use the system through a graphical user interface, through a batch interface or through some other media. An Actor's interaction with a Use Case is documented in a Use Case scenario, which details the functions a system must provide to satisfy the user requirements.

- Actors also represent the role of a user in Sequence diagrams, where you can display them using rectangle notation. Enterprise Architect supports a stereotyped Actor element for business modeling. The business modeling elements also represent Actors as stereotyped Objects.

Sequence Diagram: Notations and Symbols

Lifeline:

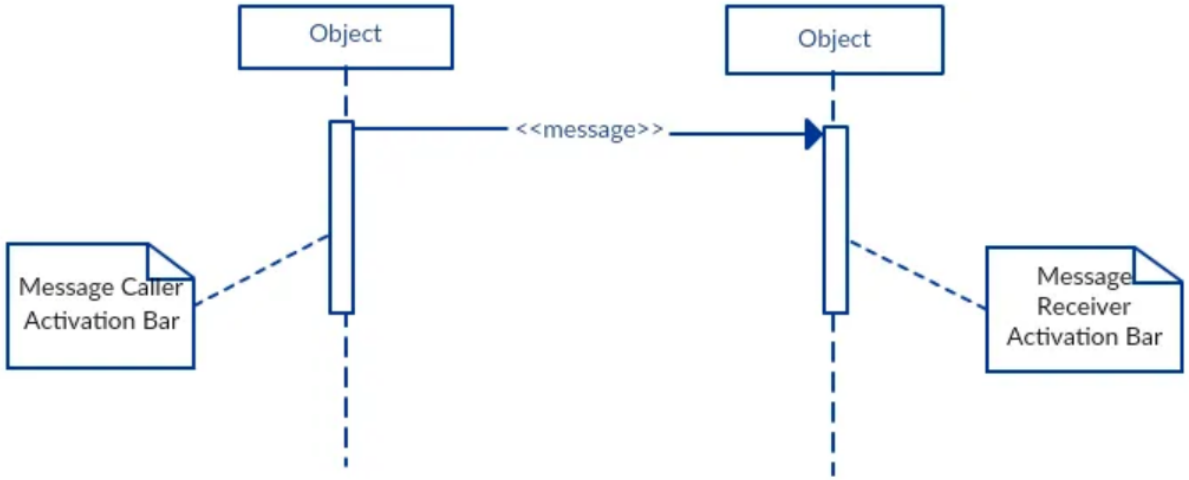
A Lifeline is an individual participant in an interaction (that is, Lifelines cannot have multiplicity).
A Lifeline represents a distinct connectable element.
To specify that representation within Enterprise Architect, right-click on the Lifeline and select the 'Advanced | Instance Classifier' option.
The 'Select <Item>' dialog displays, which you use to locate the required project classifiers.
Lifelines are available in Sequence diagrams.
There are different Lifeline elements for Timing diagrams (State Lifeline and Value Lifeline); however, although the representation differs between the two diagram types, the meaning of the Lifeline is the same.



Sequence Diagram: Notations and Symbols

Activation Bar (Execution Occurrence):

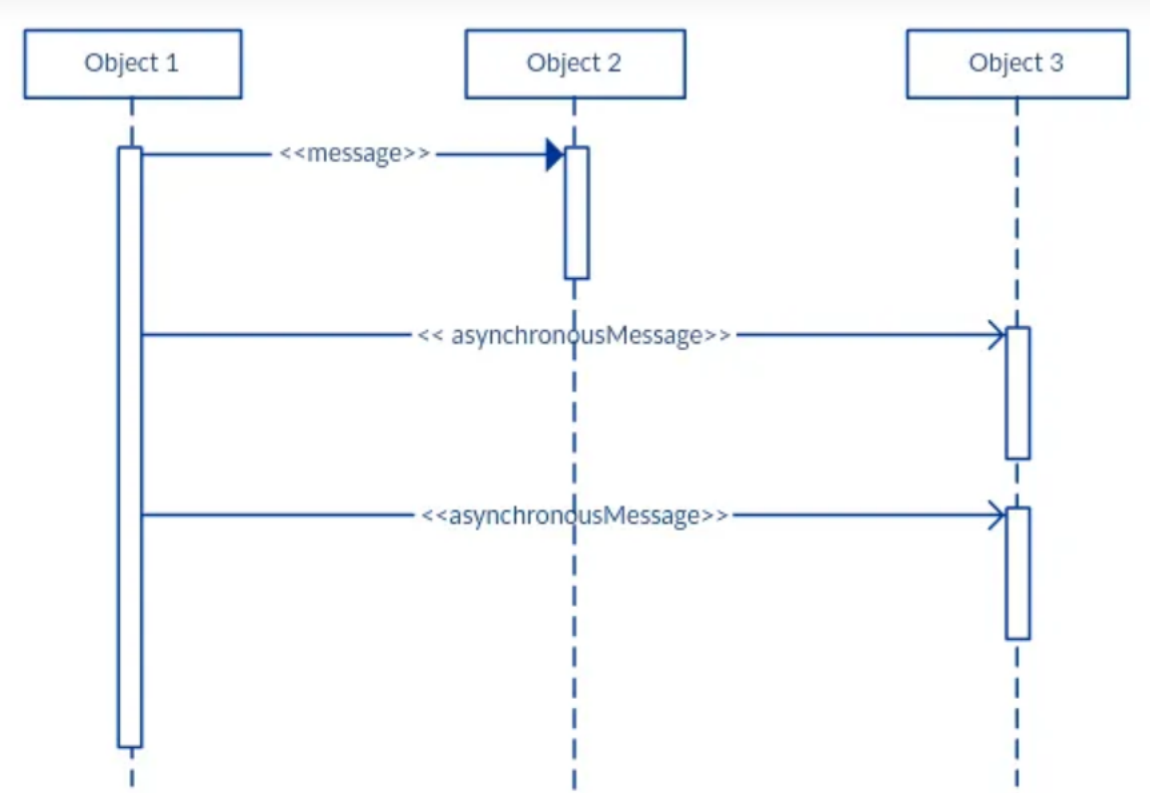
- Indicates the period during which an object is active or performing some action.
- It shows when messages are being sent and received.



Sequence Diagram: Notations and Symbols

Message:

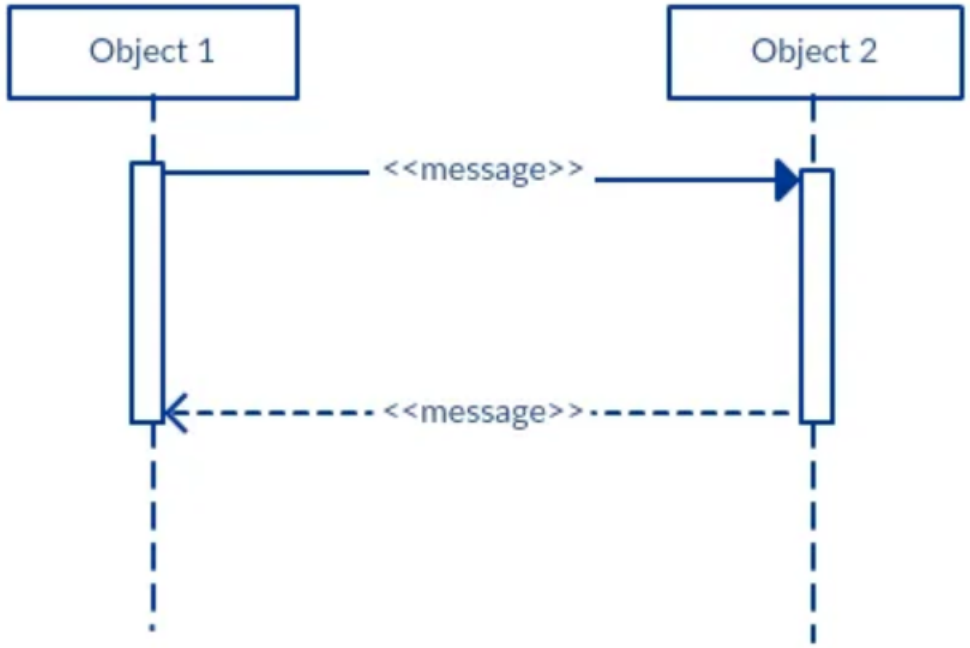
- Represents communication or interaction between objects.
- Messages can be synchronous (solid line with filled arrowhead), asynchronous (dashed line with open arrowhead), or a return message (dotted line).



Sequence Diagram: Notations and Symbols

Message:

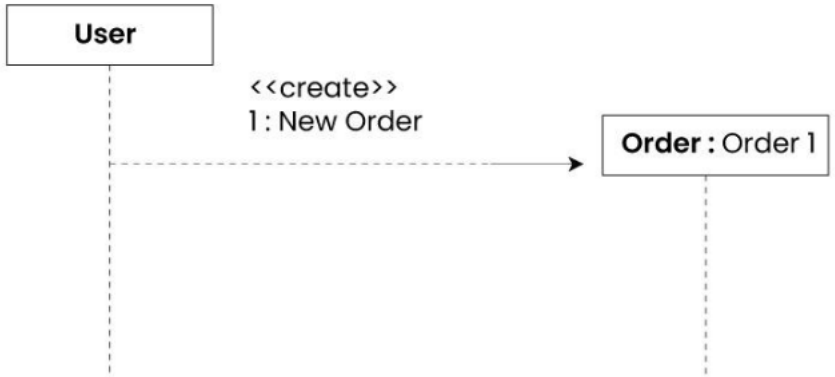
- Represents communication or interaction between objects.
- Messages can be synchronous (solid line with filled arrowhead), asynchronous (dashed line with open arrowhead), or a return message (dotted line).



Sequence Diagram: Notations and Symbols

Create Message:

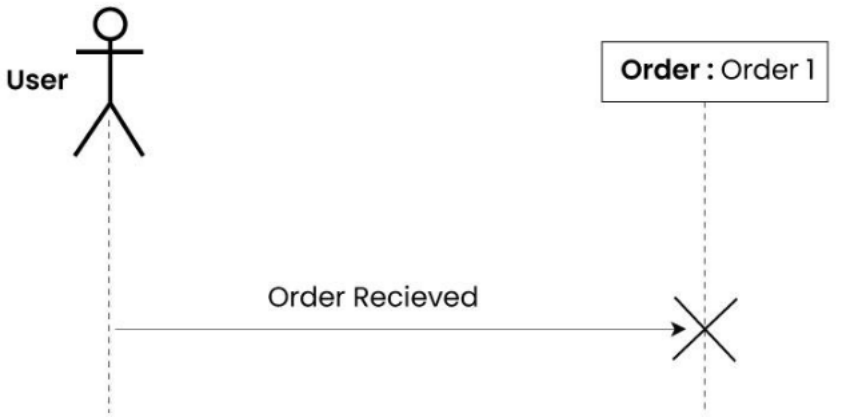
- Create message to instantiate a new object in the sequence diagram.
- There are situations when a particular message call requires the creation of an object.
- It is represented with a dotted arrow and create word labelled on it to specify that it is the create Message symbol.



Sequence Diagram: Notations and Symbols

Destroy/Delete Message:

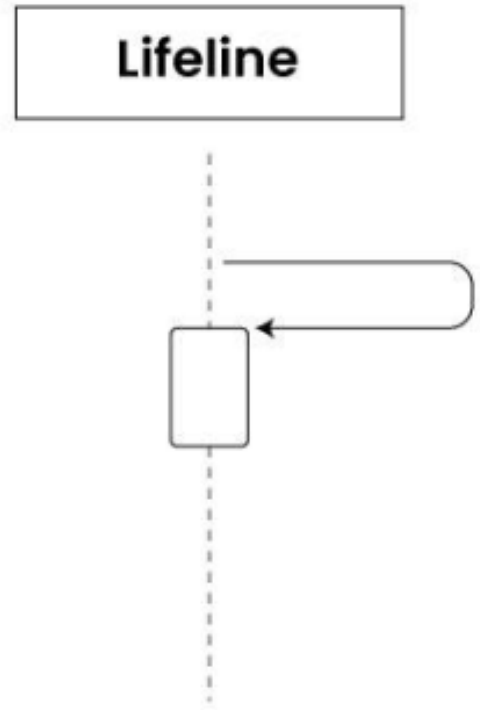
- **Symbol:** X-shaped arrowed line
- **Description:** Represents the termination or destruction of an object. It indicates the end of the object's existence.



Sequence Diagram: Notations and Symbols

Self Message:

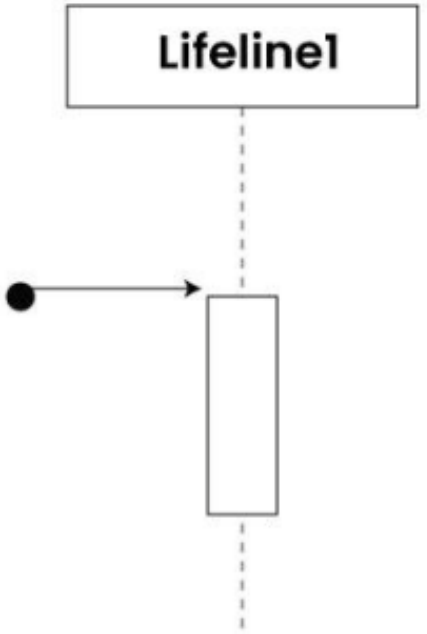
- **Symbol:** Loop arrowed line
- **Description:** Denotes a message sent by an object to itself. It is used to represent actions or method calls within the same object.



Sequence Diagram: Notations and Symbols

Found Message:

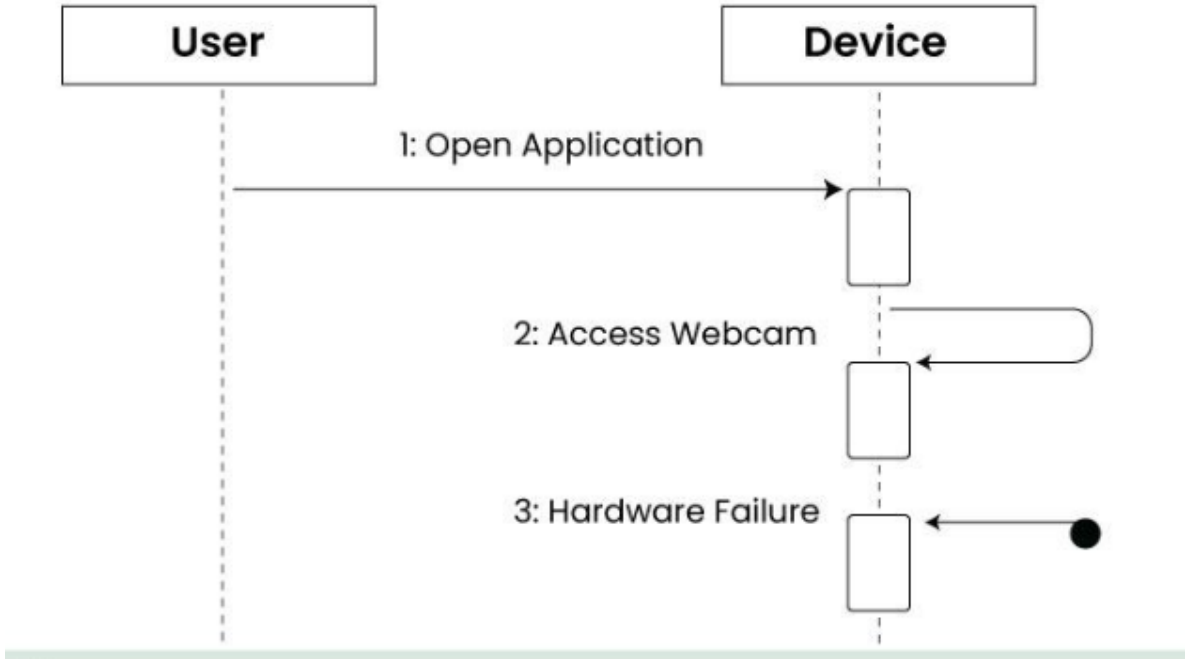
- Represent a scenario where an unknown source sends the message.
- It is represented using an arrow directed towards a lifeline from an end point.



Sequence Diagram: Notations and Symbols

Found Message:

- It can be due to multiple reasons and we are not certain as to what caused the hardware failure.



Steps to Construct an Sequence Diagram

Identify the Use Case:

Clearly define the use case or scenario that you want to represent with the sequence diagram. Understand the purpose of the interaction and the involved actors.

Identify Objects/Participants:

Identify the main objects or participants (actors, classes, components) that are part of the interaction. These will be represented by lifelines in the sequence diagram.

Draw Lifelines:

Draw vertical dashed lines to represent the lifelines of each identified object or participant. Lifelines indicate the existence of an object over time.

Arrange Lifelines:

Steps to Construct an Sequence Diagram

Arrange Lifelines:

Position the lifelines vertically, ensuring they are spaced appropriately. This layout represents the temporal order of the participating objects.

Determine the Order of Execution:

Identify the order in which the objects will execute their actions. This helps in organizing the sequence of messages in the diagram.

Draw Messages:

Use arrows to draw messages between the lifelines to represent interactions or communications between objects. Different types of messages (synchronous, asynchronous, etc.) can be indicated by varying line styles and arrowheads.

Steps to Construct an Sequence Diagram

Label Messages:

- Label each message with the method or operation being called, along with any parameters or return values. This adds clarity to the diagram and helps in understanding the purpose of each message.

Include Create and Destroy Messages:

- If necessary, include create and destroy messages to represent the instantiation and termination of objects. These are particularly relevant for dynamic interactions.

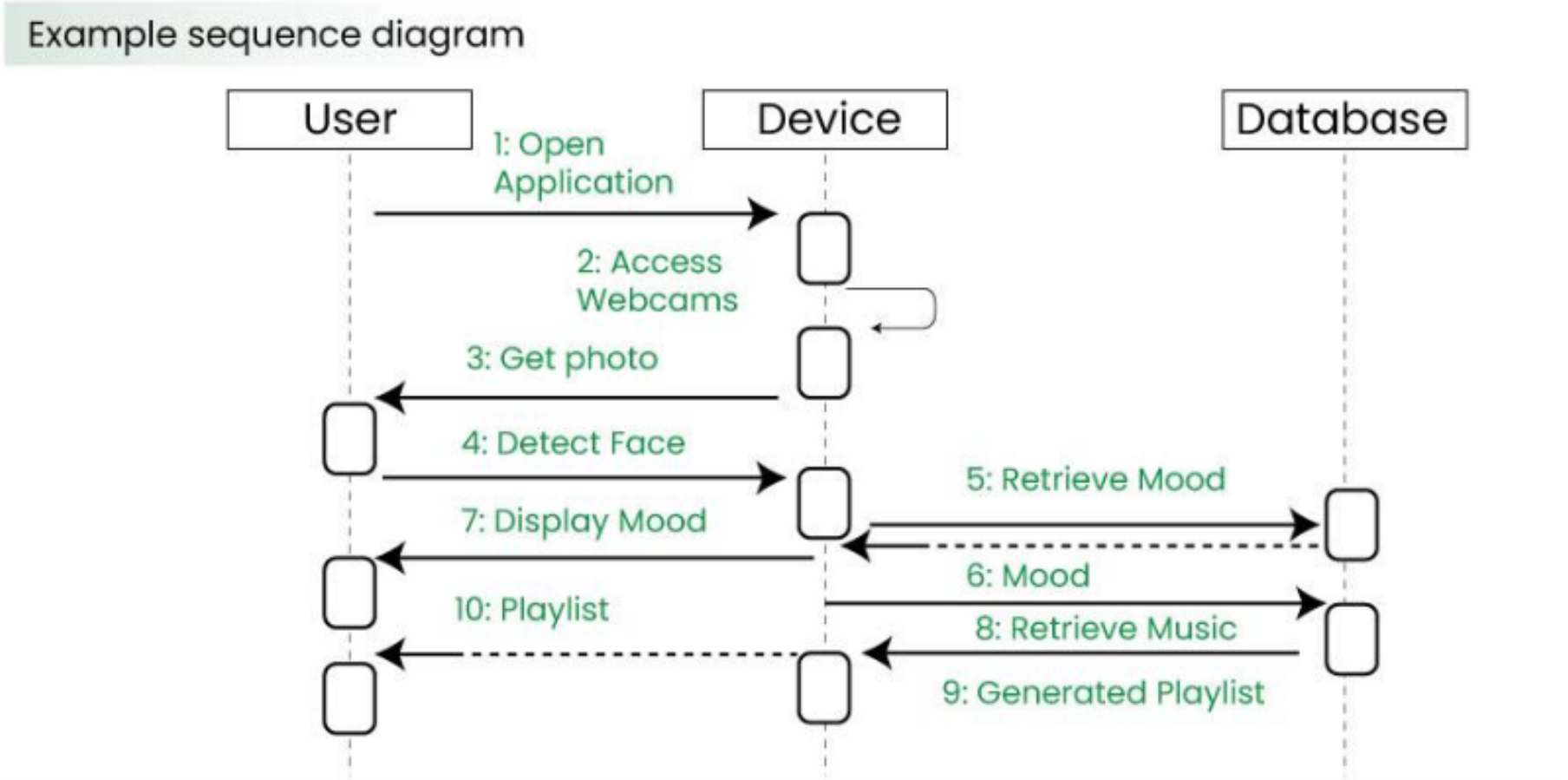
Add Conditions and Loops (Optional):

- If the sequence involves conditions or loops, you can use combined fragments or loop/alt/ option boxes to indicate these structures within the sequence diagram.
- Include Additional Details:
- Add annotations, notes, or comments to provide additional information or context that might be important for understanding the sequence of events.

Review and Refine:

- Review the sequence diagram to ensure it accurately represents the desired interaction. Refine the diagram as needed to improve clarity and readability.
- Consider Time Constraints (Optional):
- If necessary, include duration constraints to specify the minimum and maximum time duration of certain interactions.
- Validate and Iterate:
- Validate the sequence diagram with stakeholders or team members to ensure it accurately captures the intended behavior. Iterate on the diagram based on feedback received.
- Create Additional Diagrams (Optional):
- If the interaction is complex, consider creating separate sequence diagrams for different aspects of the interaction or breaking it down into smaller, more manageable parts.

Emotion based music player



THANK YOU

Dr. Kamatchi Priya L

Department of Computer Science and Engineering

priyal@pes.edu

Object Oriented Analysis and Design using Java

Dr. Kamatchi Priya L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Object Oriented Analysis and Design using Java

Behaviour Modelling:
UML State Machine Diagrams and Models

Dr. Kamatchi Priya L

Department of Computer Science and Engineering

Behavoiur Modeling

- A behavior Modeling is intended to provide clarity, for example, about internal processes, business processes or the interaction of different systems.
- UML Behavioral Diagrams depict the elements of a system that are dependent on time and that convey the dynamic concepts of the system and how they relate to each other.
- The elements in these diagrams resemble the verbs in a natural language and the relationships that connect them typically convey the passage of time.

State Diagram

- State machine diagram typically are used to describe state-dependent behavior for an object.
- An object responds differently to the same event depending on what state it is in.
- The state model is a reductionist view of behaviour that examines each object individually.

State Diagram Vs Flowchart

- A flowchart illustrates processes that are executed in the system that change the state of objects.
- A state diagram shows the actual changes in state, not the processes or commands that created those changes.

How to Draw State Diagram

- Before you begin your drawing find the initial and final state of the object in question.
- Next, think of the states the object might undergo. For example, in e-commerce a product will have a release or available date, a sold out state, a restocked state, placed in cart state, a saved on wish list state, a purchased state, and so on.
- Certain transitions will not be applicable when an object is in a particular state, for example a product can be in a purchased state or a saved in cart state if its previous state is sold out.

State Diagram: Notations and Symbols

States

States represent situations during the life of an object. A state is represented by using a rectangle with rounded corners.

A simple state

A state with internal activities

State Diagram: Notations and Symbols

Transition

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

State Diagram: Notations and Symbols

Initial State

A filled circle followed by an arrow represents the object's initial state.

State Diagram: Notations and Symbols

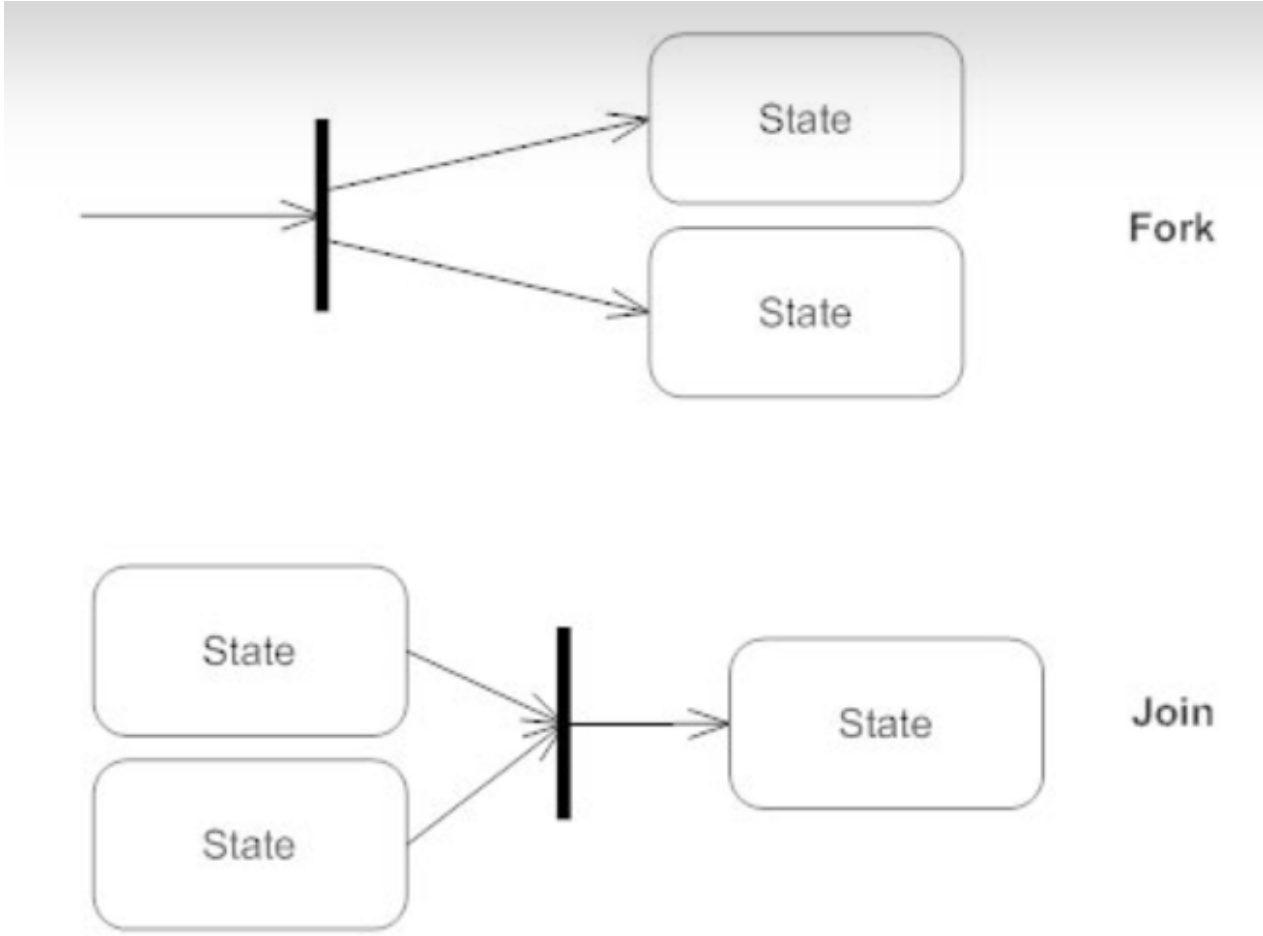
Final State

An arrow pointing to a filled circle nested inside another circle represents the object's final state

State Diagram: Notations and Symbols

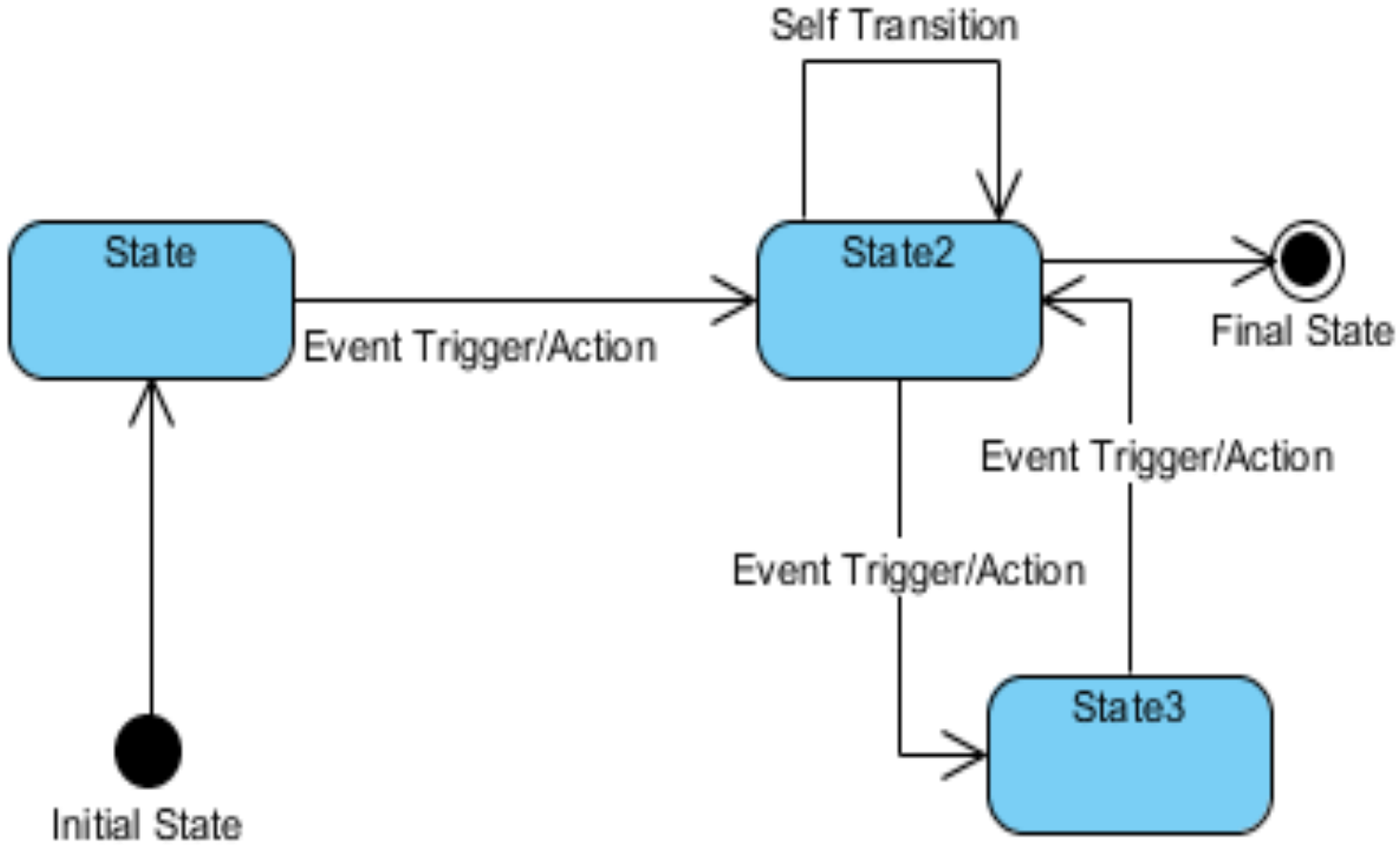
Synchronization and Splitting of Control

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



Triggers Event, Action and Guard Condiotion

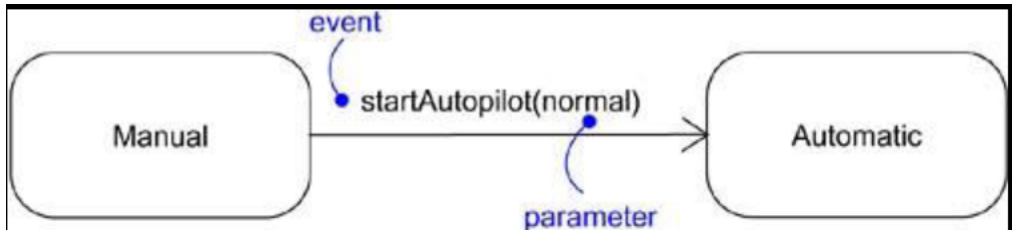
- A trigger is an event that initiates a transition from one state to another.
- A guard condition is a Boolean condition that must be satisfied for a transition to occur.
- An effect is the action or activity that happens when a transition occurs.



Adding triggers to state machine diagrams

Adding call events to state machine diagrams

A call event is an event that represents the receipt of a request to invoke an operation. A transition with a call event initiates when the called operation is invoked.



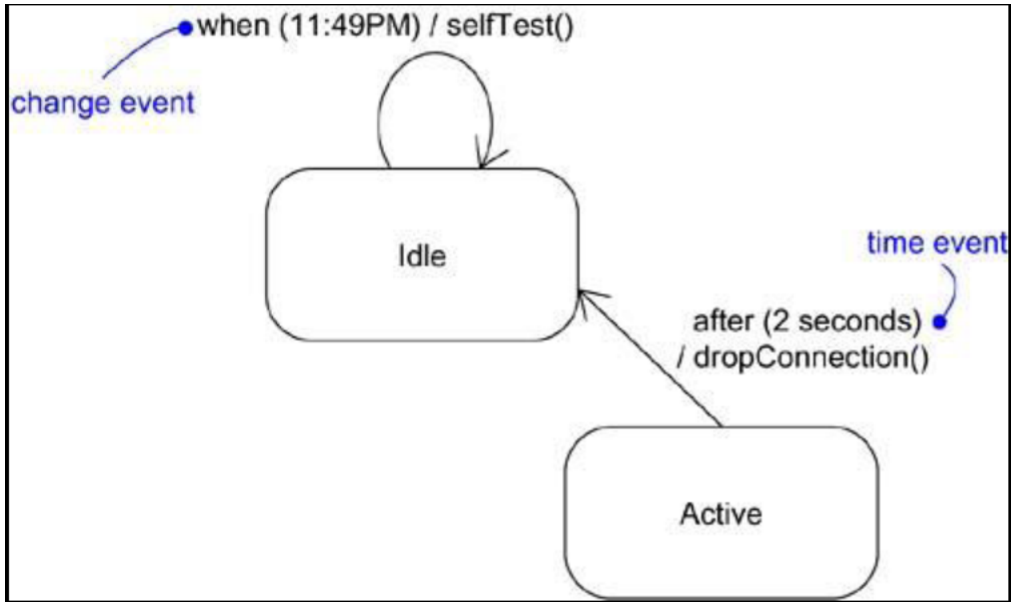
Adding triggers to state machine diagrams

Adding change events to state machine diagrams

A change event is an event that represents a condition. The condition is defined by a Boolean-valued expression that triggers a transition when its value changes from false to true.

Adding time events to state machine diagrams

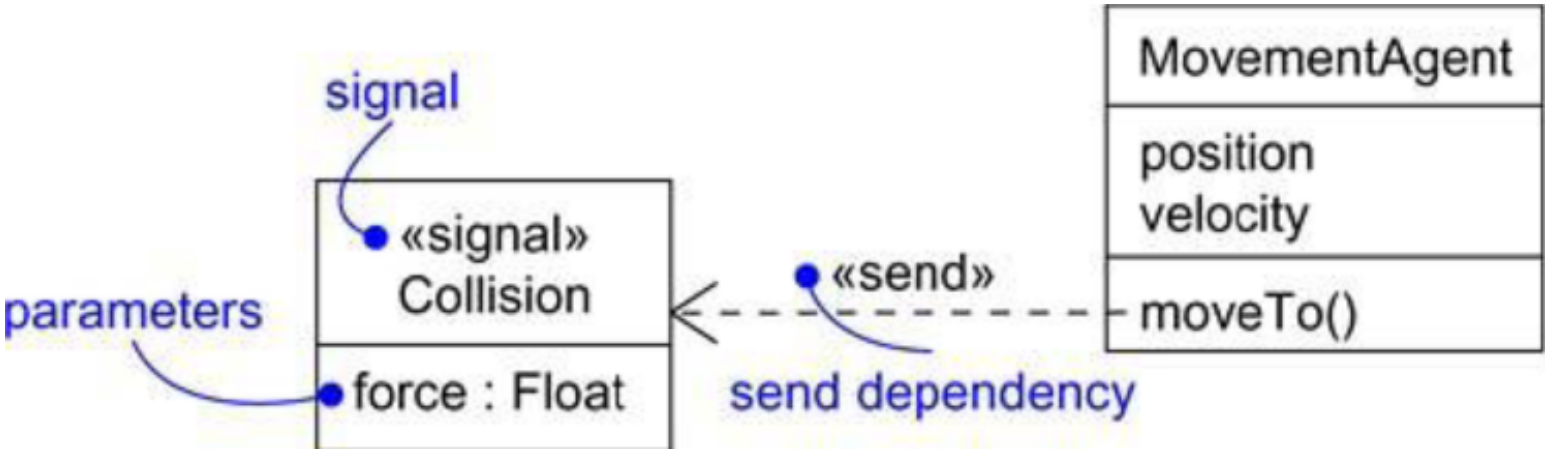
A time event is an event that represents the passage of a defined period of time or an absolute time. A transition with a time event trigger initiates when the time value is satisfied.



Adding triggers to state machine diagrams

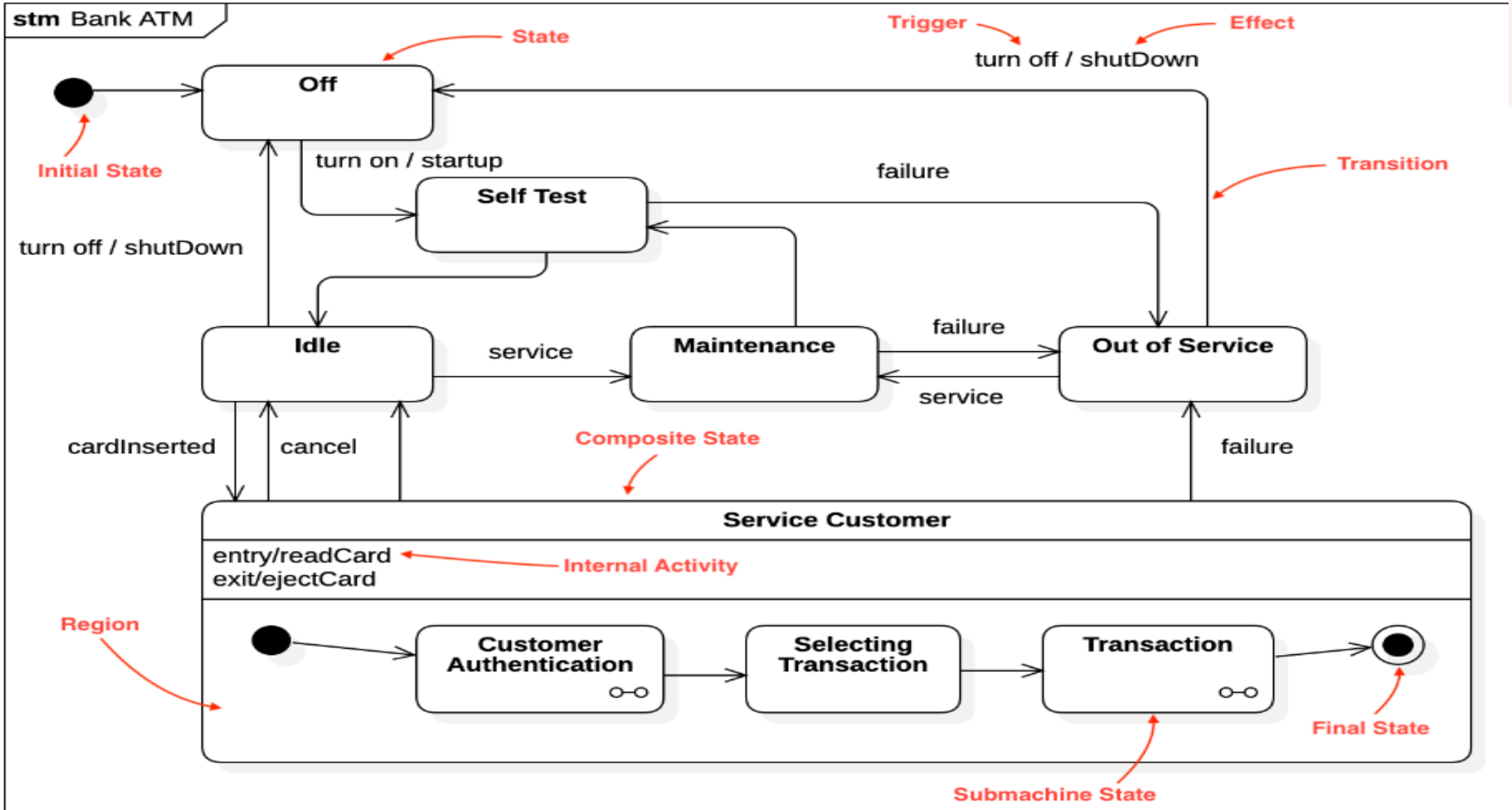
Adding signal events to state machine diagrams

A signal event represents a specific message that initiates a transition when an object receives it.



Object Oriented Analysis and Design with Java

Bank ATM



THANK YOU

Dr. Kamatchi Priya L

Department of Computer Science and Engineering

priyal@pes.edu

Object Oriented Analysis and Design using Java

Prof. Swati Shah
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

PES UNIVERSITY
CELEBRATING 50 YEARS

Object Oriented Analysis and Design using Java

Advanced state Modelling

Prof. Swati Shah

Department of Computer Science and Engineering

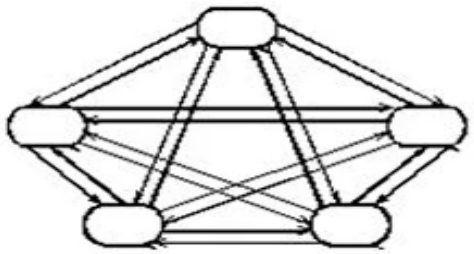
Advanced State Models

- Conventional state diagrams are **sufficient for describing simple systems but need additional power to handle large problems**.

- You can more richly model complex systems by using nest- ed state diagrams, nested states, signal generalization, and concurrency.

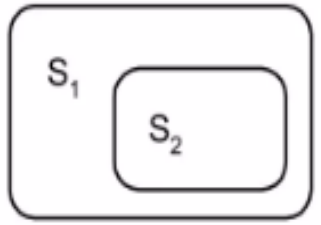
Problems with Flat State Models

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

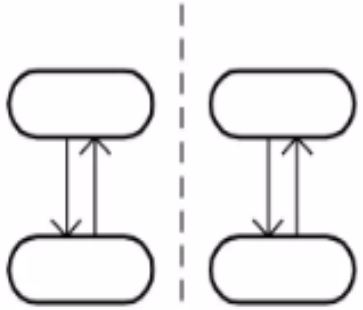


Features of Advanced state Diagram

- Two major features are introduced for controlling complexity and combinatorial explosion in state diagrams
  - **Nested** state diagrams
  - **Concurrent** state diagrams
- Many other features are also added
  - propagated transitions
  - broadcast messages
  - actions on state entry, exit
  - …



Nested State Diagrams



Concurrent State Diagrams

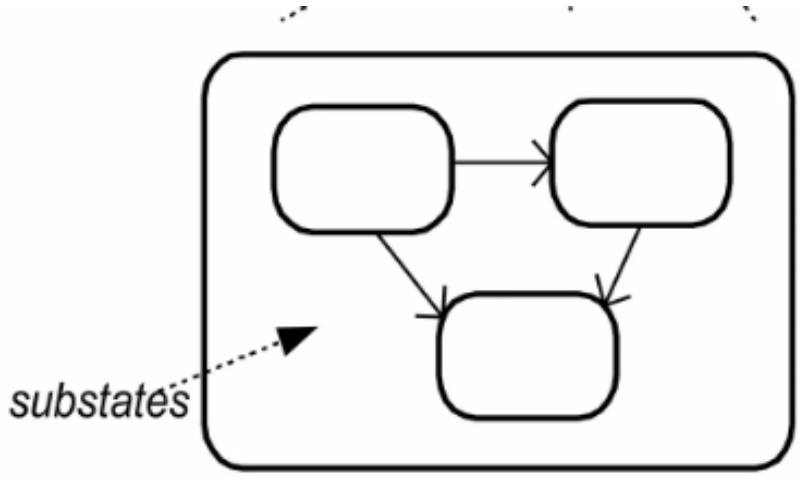
Nested State Diagram

- Activities in states are composite items denoting other lower-level state diagrams

- A lower-level state diagram corresponds to a sequence of lower-level states and events that are invisible in the higher-level diagram.

Super or Substate

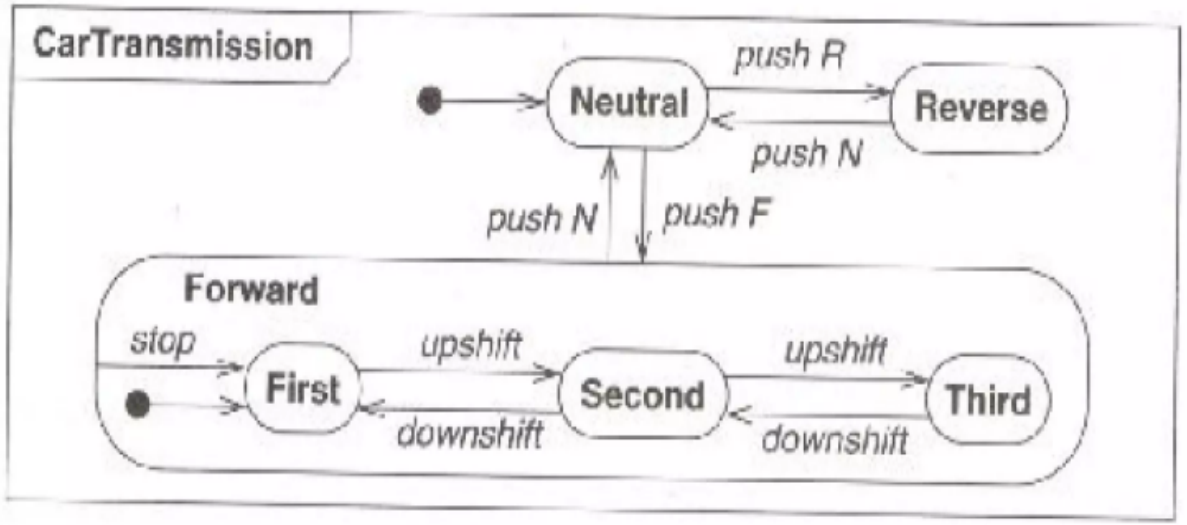
- When one state is complex, you can include substates in it.
  - drawn as nested rounded rectangles within the larger state



Nested state example

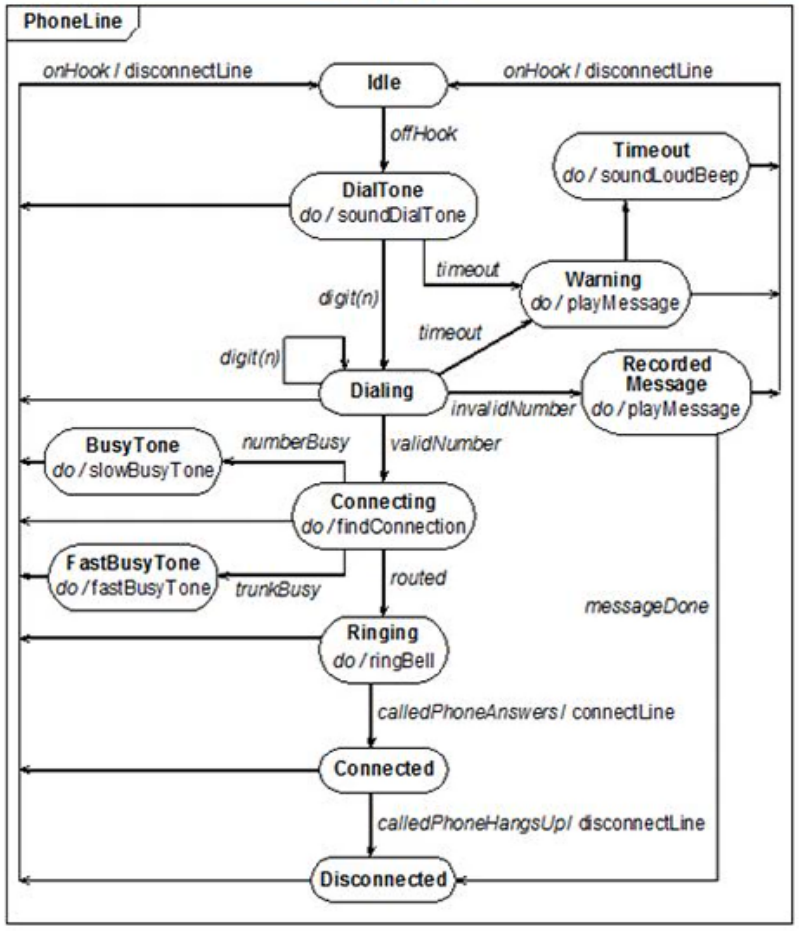
A state may be represented as nested substates.

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

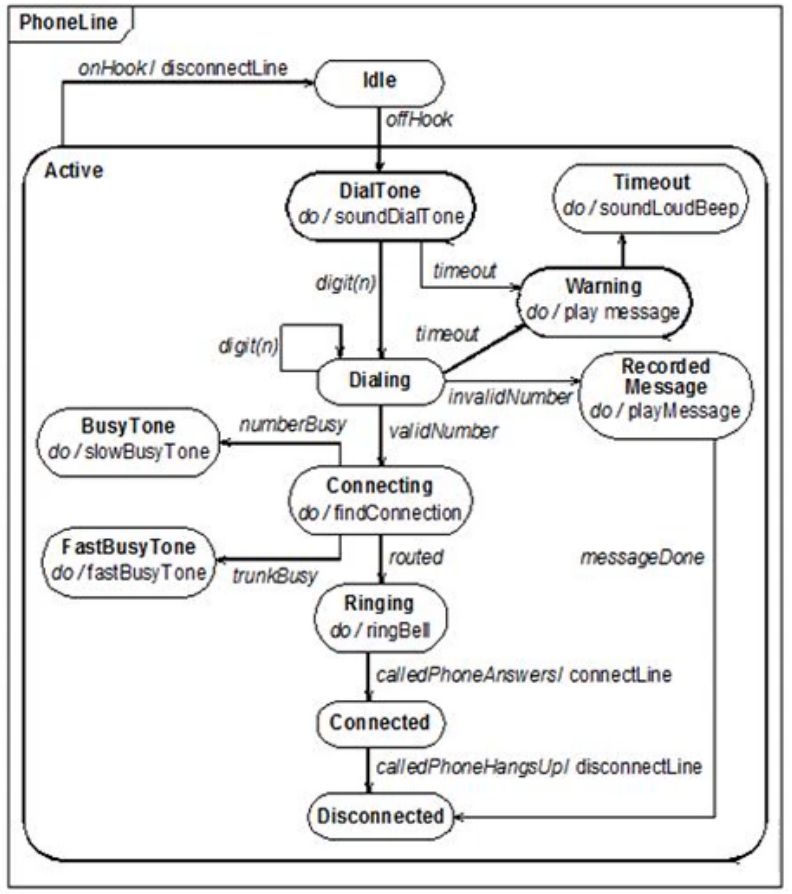


Simple state v/s Nested state

State Diagram for phone line with Activities

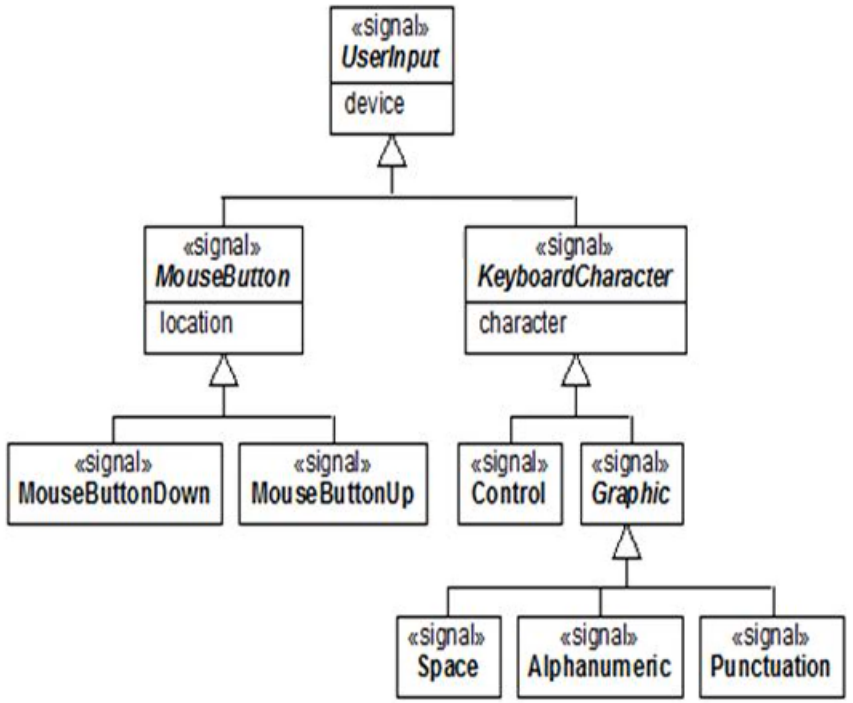


Nested states for a phone line



Signal Concurrency

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



Concurrency

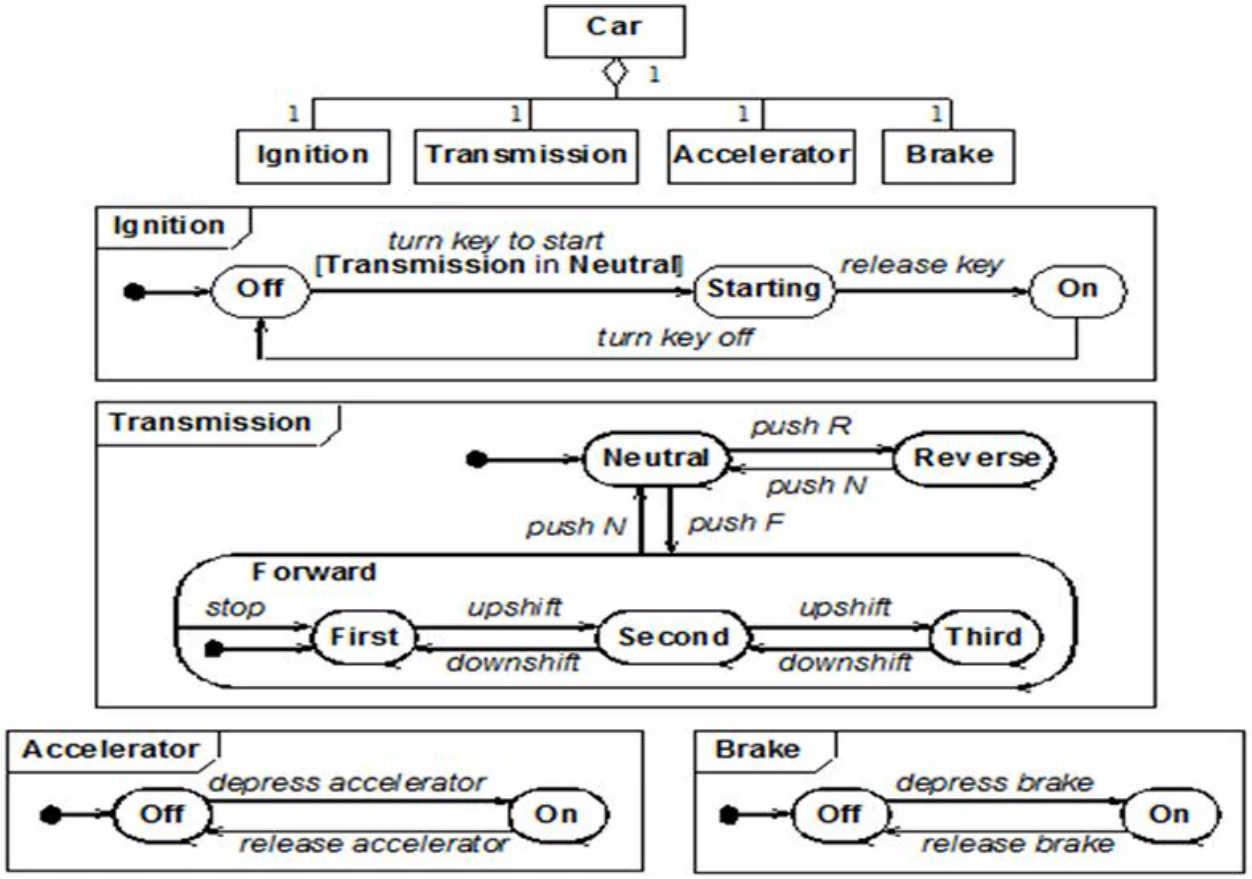
- State Models also Supports concurrency among objects. It supports two types of concurrency-

    1. Aggregation Concurrency
    2. Concurrency within an object

Aggregation Concurrency

- A state diagram for an assembly is a collection of state diagram, one for each part. The aggregate state corresponds to the combined states of all the parts.
- Aggregation is "**and-relationship**".
- Aggregate state is one state from the first diagram, and a state from second diagram and a state from each other diagram. In the more interesting cases, the part states interact.

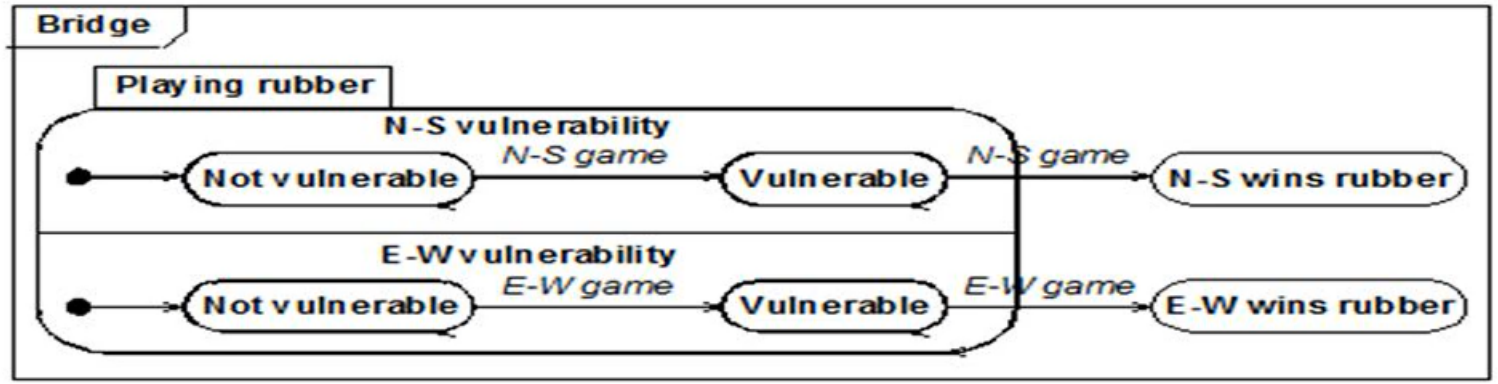
Aggregation Concurrency(Cont'd)



Concurrency within an object

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

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



THANK YOU

Prof Swati Shah

Department of Computer Science and Engineering

swatishahjaiswal@pes.edu

Object Oriented Analysis and Design Using Java

UE21CS352B
Unit:1

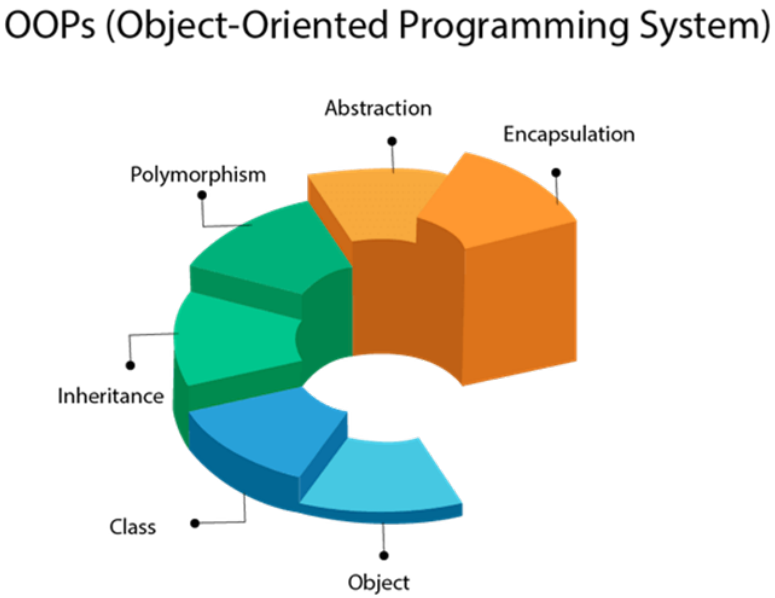
Lecture 19: Object-oriented Programming: JVM

Prof . Shilpa S

Department of Computer Science and Engineering

UE21CS352B: Object Oriented Analysis and Design Using Java

Introduction to Object Oriented Programming



Prof . Shilpa S

Department of Computer Science and Engineering

Agenda

- Introduction to Java
- Features
- JVM
- JRE
- Java First Program

Introduction to Java

- Java programming language was originally developed by Sun Microsystems, by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0)

- The new J2 versions were renamed as Java SE, Java EE and Java ME respectively.

1. Java Platform, Standard Edition (Java SE)
2. Java Platform, Enterprise Edition (Java EE)
3. Java Platform, Micro Edition (Java ME)

- Java is guaranteed to be Write Once, Run Anywhere.

- Java was mainly developed to create software for consumer electronic devices that could be controlled by a remote

Introduction to Java

Why Use Java?

- Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- It is one of the most popular programming languages
- It is easy to learn and simple to use
- It is open-source and free
- It is secure, fast and powerful
- It has huge community support (tens of millions of developers)
- Java is an object-oriented language that gives a clear structure to programs and allows code to be reused, lowering development costs

Applications:

Mobile applications (specially Android apps), Desktop applications, Web applications
Web servers and application servers, Games, Database connection, And much, much more!

Introduction to Java: Features

1. Simple

There is no need for header files, pointer arithmetic, structures, unions, operator overloading, virtual base classes

2. Object-Oriented

Object-oriented design is a programming technique that focuses on the data (= objects) and on the interfaces to that object.

3. Distributed

Java has an extensive library of routines for coping with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across the Net via URLs with the same ease as when accessing a local file system.

4. Robust

Inbuilt exception handling features and memory management features.

Introduction to Java: Features

5. Secure

Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems.

6. Portable

Unlike C and C++, there are no “implementation-dependent” aspects of the specification. The sizes of the primitive data types are specified, as is the behavior of arithmetic on them.

7. Interpreted

The programmer writes code that will be executed by an interpreter, rather than compiled into object code loaded by the OS and executed by CPU directly. An interpreter executes the code line by line.

8. High-Performance

The bytecodes can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on. The just-in-time compiler knows which classes have been loaded. It can use inlining when, based upon the currently loaded collection of classes, a particular function is never overridden, and it can undo that optimization later if necessary.

Introduction to Java: Features

Common terms of Java:

- **Java Virtual Machine(JVM):** This is generally referred to as JVM. There are three execution phases of a program. They are written, compile and run the program. In the Running phase of a program, JVM executes the bytecode generated by the compiler.

- **Java Development Kit(JDK):** It is a complete Java development kit that includes compiler, Java Runtime Environment (JRE), java debuggers, java docs, etc.

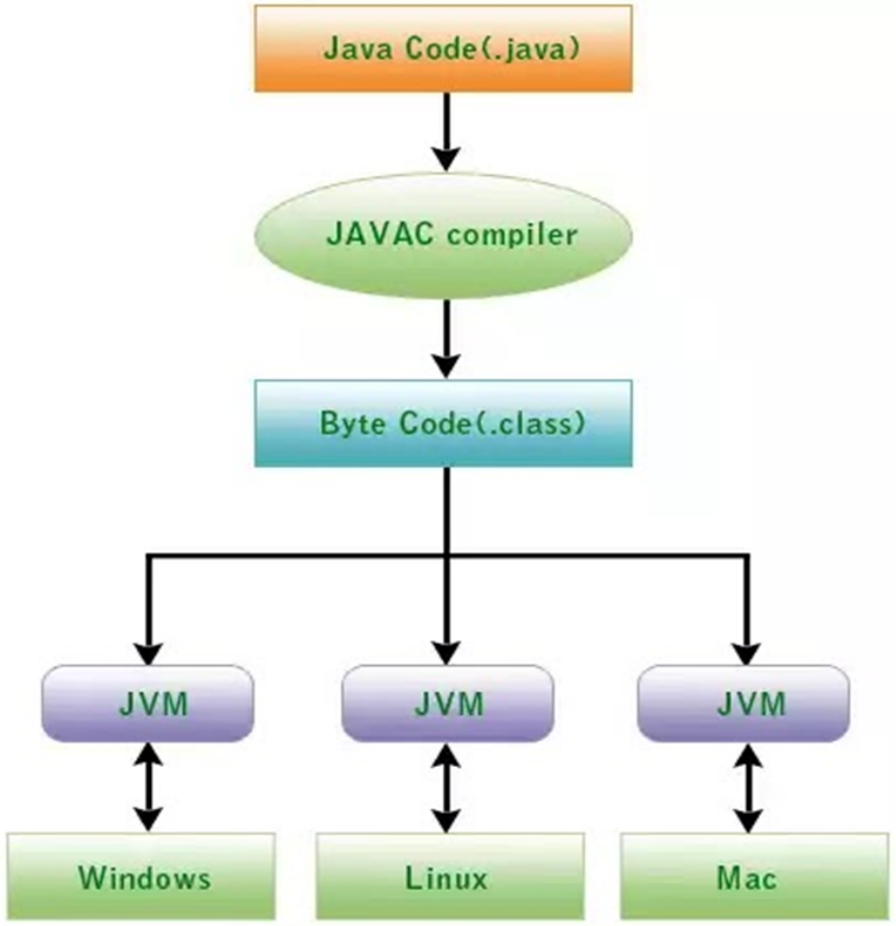
- **Java Runtime Environment (JRE):** JDK includes JRE. JRE installation on our computers allows the java program to run, however, we cannot compile it. JRE includes a browser, JVM, applet support, and plugins. For running the java program, a computer needs JRE.

JAVA Virtual Machine

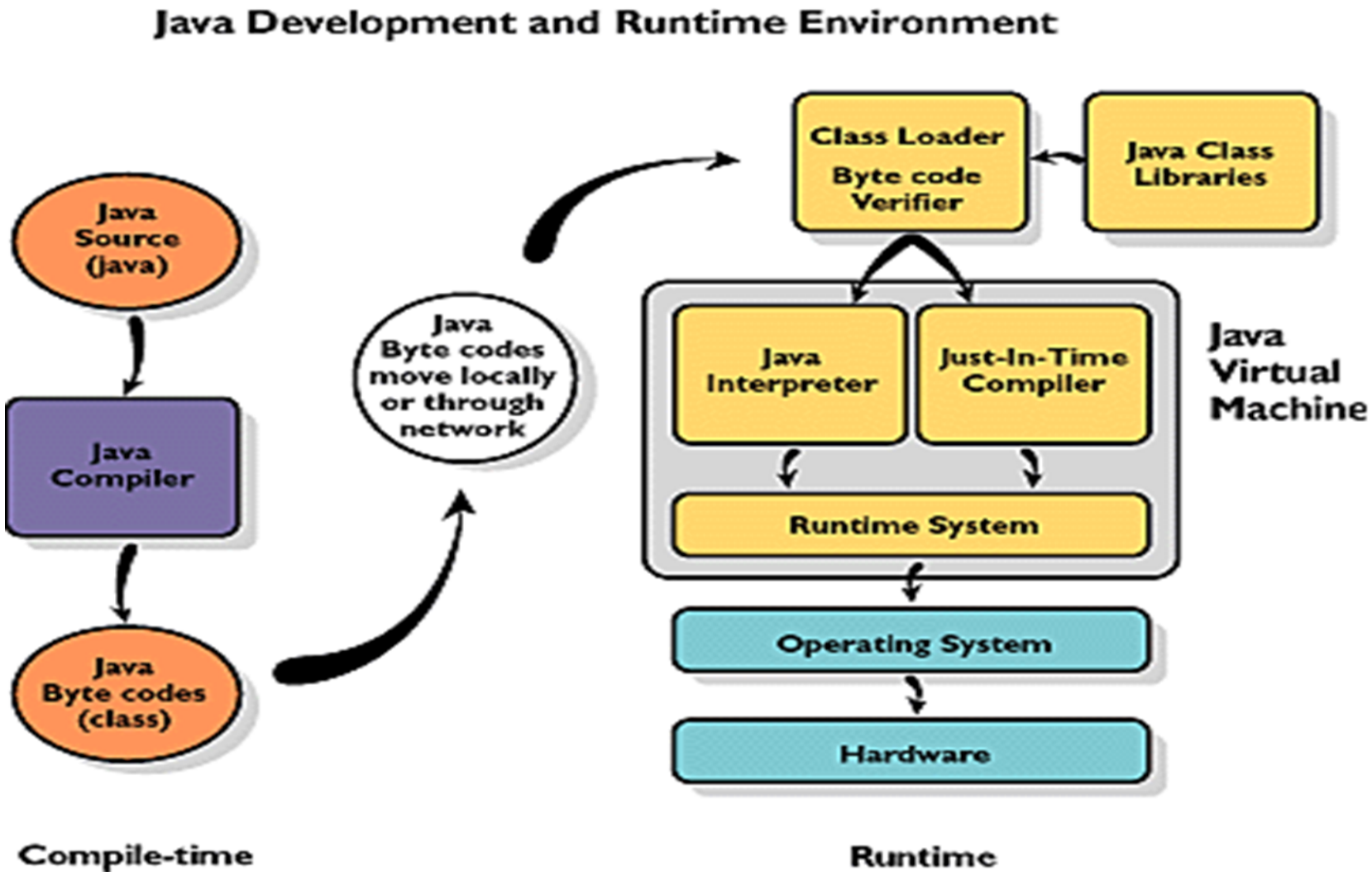
JVM is a platform-independent execution environment that converts Java bytecode(.class file) into machine language and executes it.

It is:

- **A specification** where the working of Java Virtual Machine is specified. But implementation provider is independent in choosing the algorithm. Its implementation has been provided by Oracle and other companies.
- **An implementation** Its implementation is known as JRE (Java Runtime Environment).
- **Runtime Instance** Whenever you write Java command on the command prompt to run the Java class, an instance of JVM is created.



Java Runtime Environment(JRE)



Java First Program

Implementation of a Java application program involves the following steps.

1. Creating the program
2. Compiling the program
3. Running the program

Example:

```
public class Main {
        public static void main(String[] args) {
                System.out.println(“My Java First Program");
        }
}
```

File -> Save -> Test.java

Output:

My Java First Program

THANK YOU

Prof. Shilpa S
Department of Computer Science and Engineering

shilpas@pes.edu

Object Oriented Analysis and Design Using Java

UE21CS352B
Unit:1

Lecture 20: Abstraction, Encapsulation, Composition

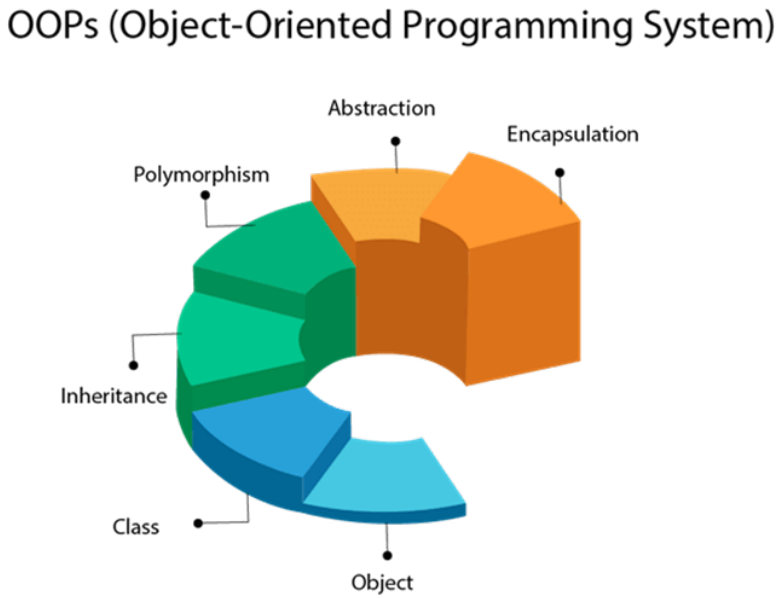
Prof . Shilpa S

Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE21CS352B: Object Oriented Analysis and Design Using Java

Abstraction, Encapsulation, Composition



Prof . Shilpa S

Department of Computer Science and Engineering

Agenda

- Abstraction
- Encapsulation
- Composition

Abstraction

Data abstraction in Object-oriented programming is a process of providing functionality to the users by hiding its implementation details from them. In other words, the user will have just the knowledge of what an entity is doing instead of its internal working.

In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

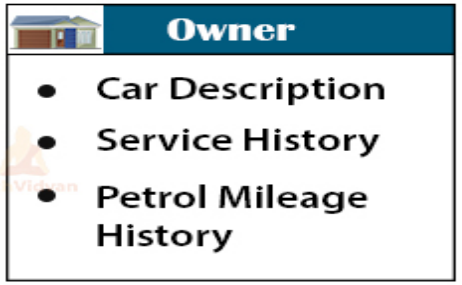
Advantages of Java Abstraction:

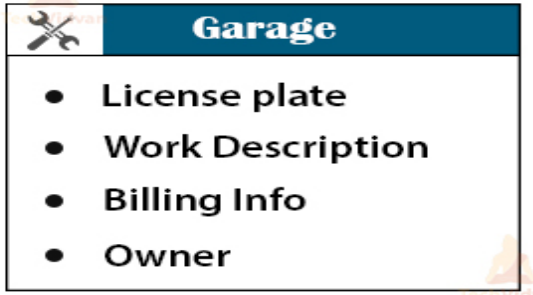
- Reduce Complexity
- Avoid code duplication
- Eases the burden of maintenance
- Increase Security and Confidentiality

Abstraction

Abstraction defines an object in terms of its properties (attributes), behavior (methods), and interfaces (means of communicating with other objects).
Abstraction refers to the act of representing essential features without including the background details or explanations.
Since classes use the concept of data abstraction, they are known as **Abstract Data Types (ADT)**

Example:





Abstraction

Abstract class in Java:

- In Java, we can achieve Data Abstraction using Abstract classes and interfaces.
- Interfaces allow 100% abstraction (complete abstraction).
- An Abstract class is a class whose objects can’t be created.
- An Abstract class is created through the use of the abstract keyword. It is used to represent a concept.
- An abstract class can have abstract methods (methods without body) as well as non-abstract methods or concrete methods (methods with the body). A non-abstract class cannot have abstract methods.

Abstraction

Abstract class in Java:

- The class has to be declared as abstract if it contains at least one abstract method.
- An abstract class does not allow you to create objects of its type. In this case, we can only use the objects of its subclass.
- Using an abstract class, we can achieve 0 to 100% abstraction.
- There is always a default constructor in an abstract class, it can also have a parameterized constructor.
- The abstract class can also contain final and static methods.

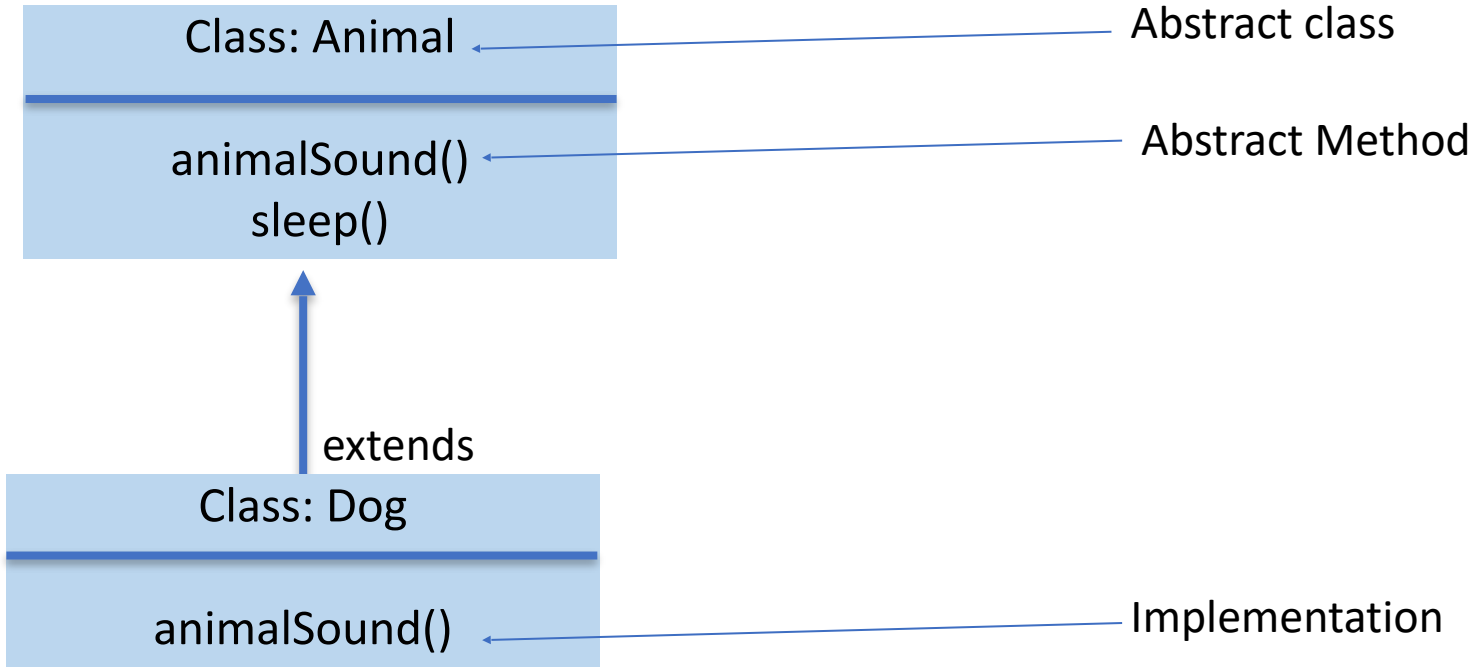
Abstraction : Example

```
abstract class Animal {                    // Abstract class
   public abstract void animalSound();          // Abstract method (does not have a body)
   public void sleep() {                        // Regular method
   System.out.println("Zzz");
  }
}
class Dog extends Animal {              // Subclass (inherit from Animal)
  public void animalSound() {
   System.out.println("The Dog says: bow bow ");        // The body of animalSound() is provided here
  }
}
class Main {
  public static void main(String[] args) {
   Dog mydog = new Dog(); // Create a Dog object
   mydog.animalSound();
   mydog.sleep();
  }
}
```

Output:

```
The Dog says: bow bow
Zzz
```

Abstraction : Example



Encapsulation

- The wrapping up of data and functions into a single unit is known as encapsulation.
- The data is not accessible to the outside world, only those function which are wrapped in can access it.
- These functions provide the interface between the object's data and the program.
- This insulation of the data from direct access by the program is called data hiding or information hiding.

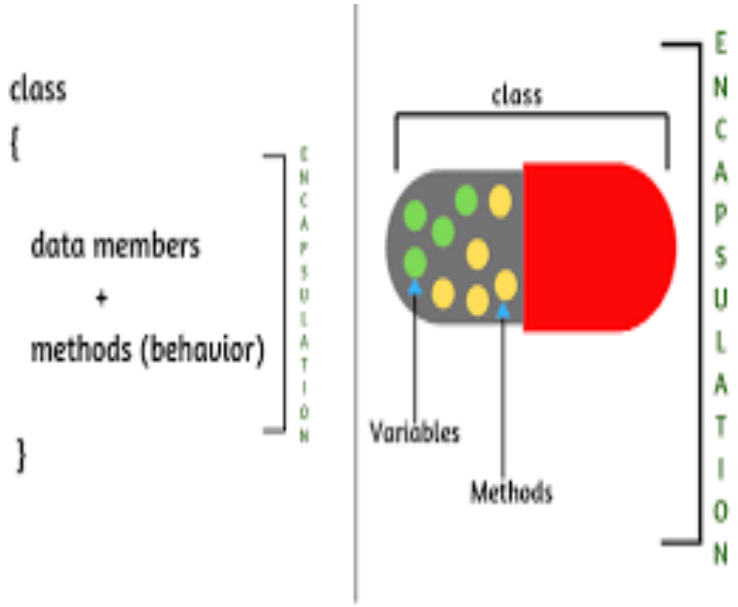


Fig: Encapsulation

Encapsulation

Advantages of Encapsulation

- **Data Hiding**: it is a way of restricting the access of our data members by hiding the implementation details.
- **Increased Flexibility**: We can make the variables of the class read-only or write-only depending on our requirements.
- **Reusability**: Encapsulation also improves the re-usability and is easy to change with new requirements.
- **Testing code is easy**: Encapsulated code is easy to test for unit testing.
- **Freedom to the programmer in implementing the details of the system**: This is one of the major advantages of encapsulation that it gives the programmer freedom in implementing the details of a system.

Encapsulation

Disadvantages of Encapsulation in Java

- Can lead to increased complexity, especially if not used properly.
- Can make it more difficult to understand how the system works.
- May limit the flexibility of the implementation.

Encapsulation: Example

```
class Person {
  private String name;        // private = restricted access

  public String getName() {                    // Getter
    return name;
  }
    public void setName(String newName) {     // Setter
    this.name = newName;
  }
}
class Main{
  public static void main(String[] args) {
    Person myObj = new Person();
    myObj.name = "John";                       // error
    System.out.println(myObj.name);            // error
  }
}
```

Output:

ERROR!

javac/tmp/mNvaXCz9rO/Main.java/tmp/mNvaXCz9rO/Main.java:15: error: name has private access in Person

myObj.name = "John"; //error

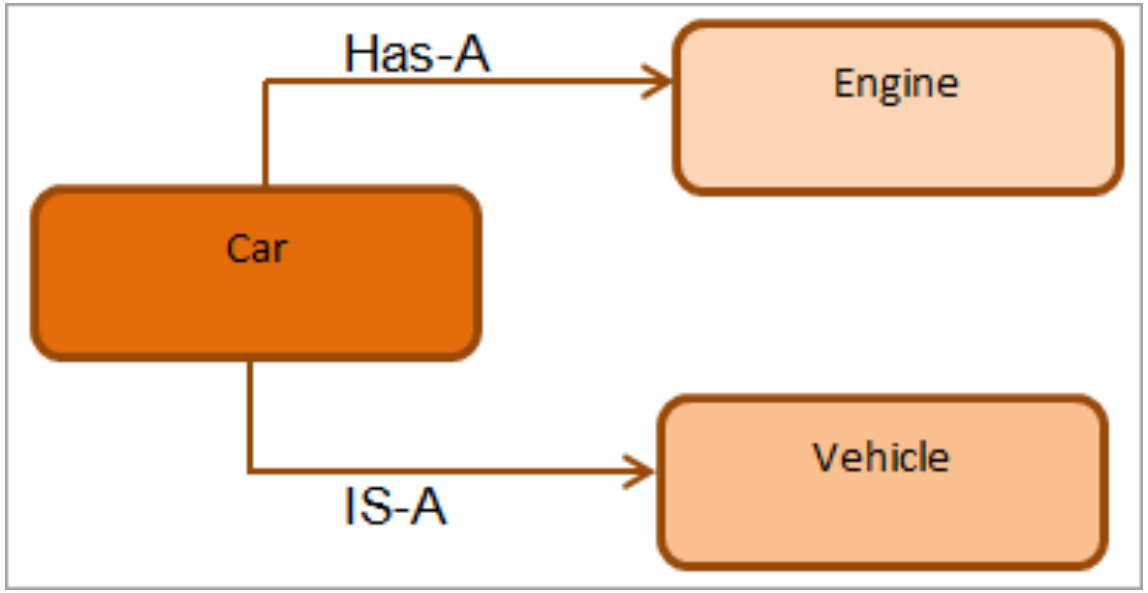
/tmp/mNvaXCz9rO/Main.java:16: error: name has private access in Person

System.out.println(myObj.name); // error

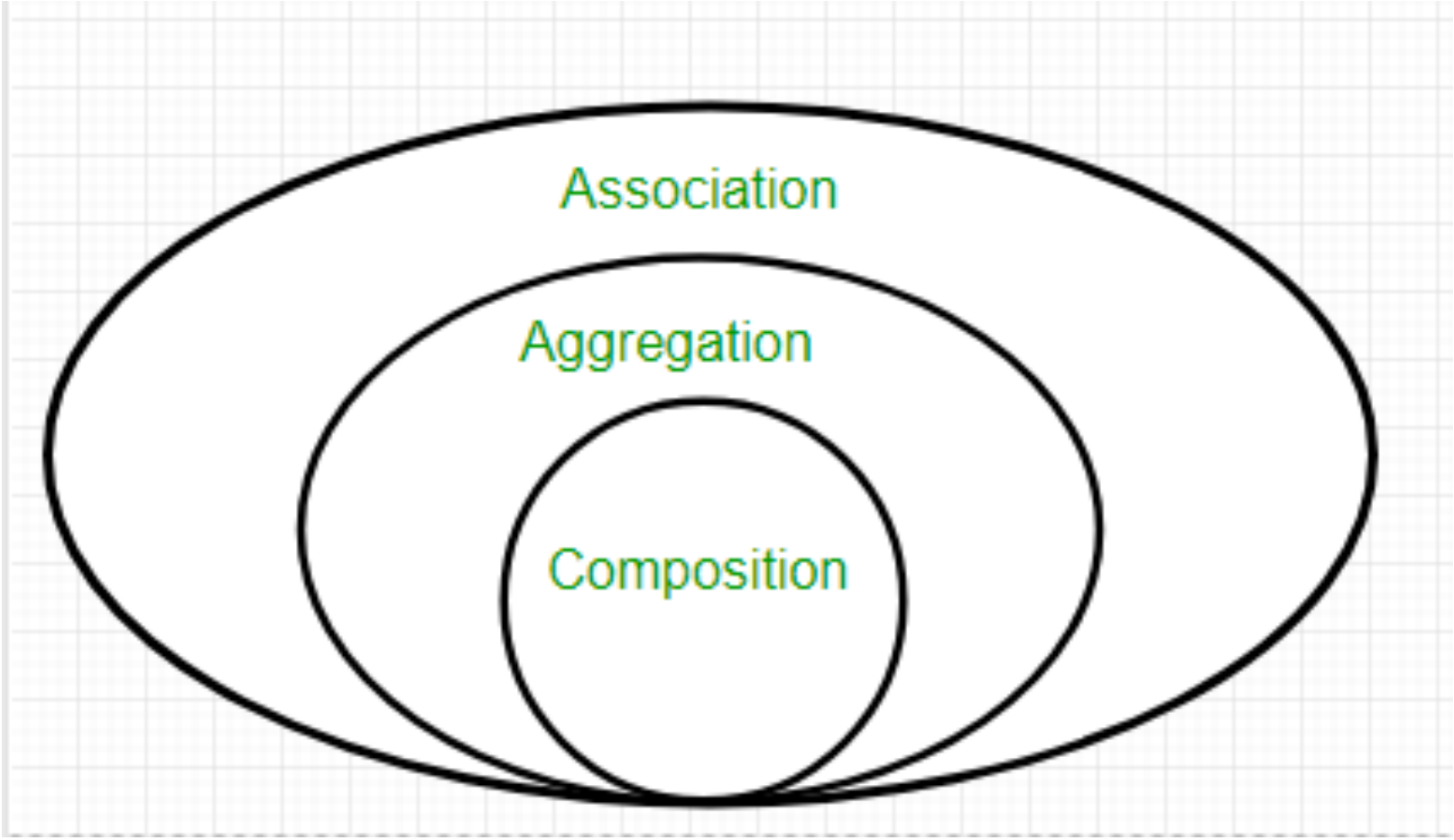
^2 errors

Composition

- The Composition is a way to design or implement the "has-a" relationship.
- Composition and Inheritance both are design techniques.
- The Inheritance is used to implement the "is-a" relationship. The "has-a" relationship is used to ensure the code reusability in our program.
- In Composition, we use an instance variable that refers to another object.



Composition



Composition

- The Composition represents a part-of relationship.
- Both entities are related to each other in the Composition.
- The Composition between two entities is done when an object contains a composed object, and the composed object cannot exist without another entity. For example, if a university HAS-A college-lists, then a college is a whole, and college-lists are parts of that university.
- Favor Composition over Inheritance.
- If a university is deleted, then all corresponding colleges for that university should be deleted.

Composition

Benefits of using Composition:

- Composition allows us to reuse the code.
- In Java, we can use multiple Inheritance by using the composition concept.
- The Composition provides better test-ability of a class.
- Composition allows us to easily replace the composed class implementation with a better and improved version.
- Composition allows us to dynamically change our program's behavior by changing the member objects at run time.

Composition : Example

```
class CarOil {
  public void FillOil() {
    System.out.println("The fuel is full in the car");
  }
  public void EmptyOil() {
    System.out.println("The car has low oil");
  } }
class Car {
  private String colour;
  private int maxi_Speed;
  public void carDetails(){
    System.out.println("Car Colour= "+colour + ", Maximum Speed= " + maxi_Speed);
  }
    public void setColour(String colour) {              //Setting colour of the car
    this.colour = colour;
  }
    public void setMaxiSpeed(int maxi_Speed) {                  //Setting maximum car Speed

    this.maxi_Speed = maxi_Speed;
```

Composition : Example

```
class Ninja extends Car {
  public void NinjaOil() {
    CarOil Ninja_Oil = new CarOil();    //composition
    Ninja_Oil.FillOil();
  }
}

class Main {
  public static void main(String[] args) {
    Ninja NinjaCar = new Ninja();
    NinjaCar.setColour("Orange");
    NinjaCar.setMaxiSpeed(180);
    NinjaCar.carDetails();
    NinjaCar.NinjaOil();
  }
}
```

OUTPUT

Car Colour= Orange, Maximum Speed= 180
The fuel is full in the car

THANK YOU

Prof. Shilpa S
Department of Computer Science and Engineering

shilpas@pes.edu