MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

Software Design

UNIT II
S.Y. Semester III

Syllabus – Unit II

SOFTWARE DESIGN

Abstraction, Modularity, Cohesion & Coupling, Scenario based modeling, SSAD (ER diagram, Data Flow Diagram DFD), OOAD (Unified Modeling Language UML). .

Static modeling

Class diagrams- Finding analysis and Design Classes, Object diagrams, Composite structure diagrams, Package diagrams, Interfaces and Components, Deployment Diagram,

Dynamic Modeling

Use case diagram, Activity diagram- Interaction & Interaction overview diagram, sequence diagram, Timing diagram, Communication diagram, Advance state machine diagram

SOFTWARE DESIGN

SOFTWARE DESIGN

- Abstraction, Modularity, Cohesion & Coupling,
- Scenario based modeling : it identifies the primary use cases for the proposed software system or application, to which later stages of requirements **modeling** will refer.
- SSAD (ER diagram, Control Flow Diagram CFD , Data Flow Diagram DFD),
- OOAD (Unified Modeling Language UML)- Modeling Language (UML)- Introduction to all Diagrams.

Software Design

- Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.
- Software design usually involves problem solving and planning a software solution. This includes both a low-level component and algorithm design and a high-level, architecture design.
- It’s the next step after RE

Software Design Levels

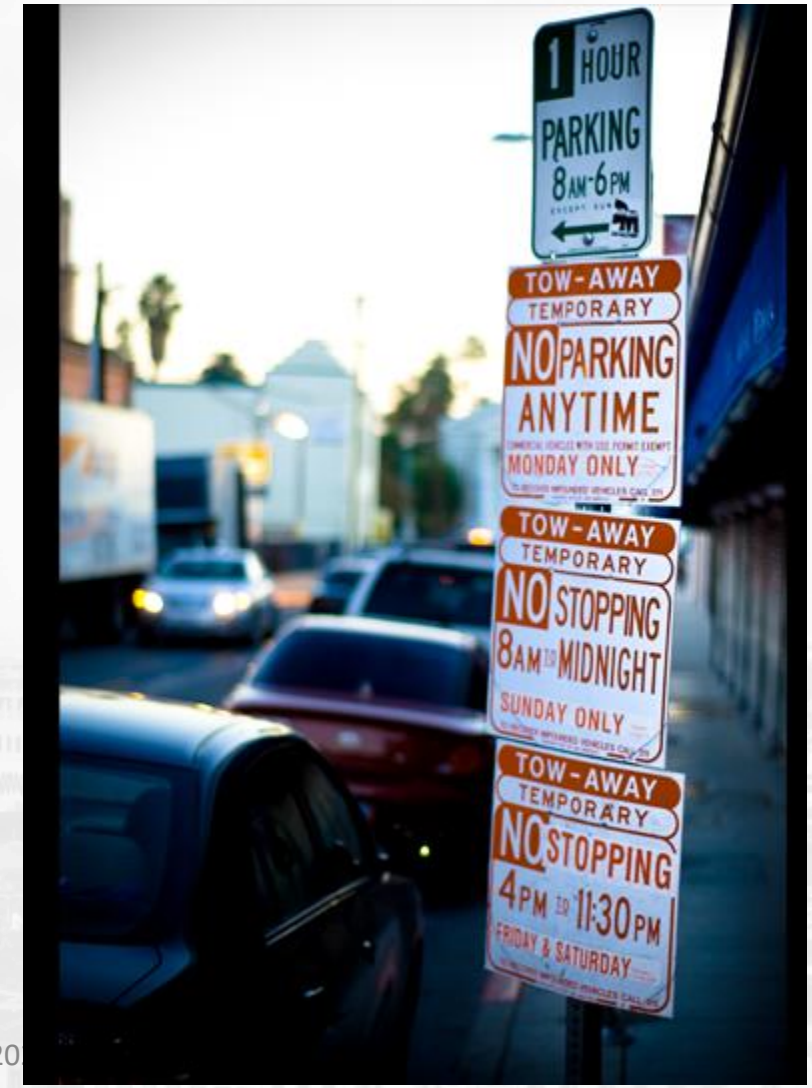
- Software design yields **three** levels of results:

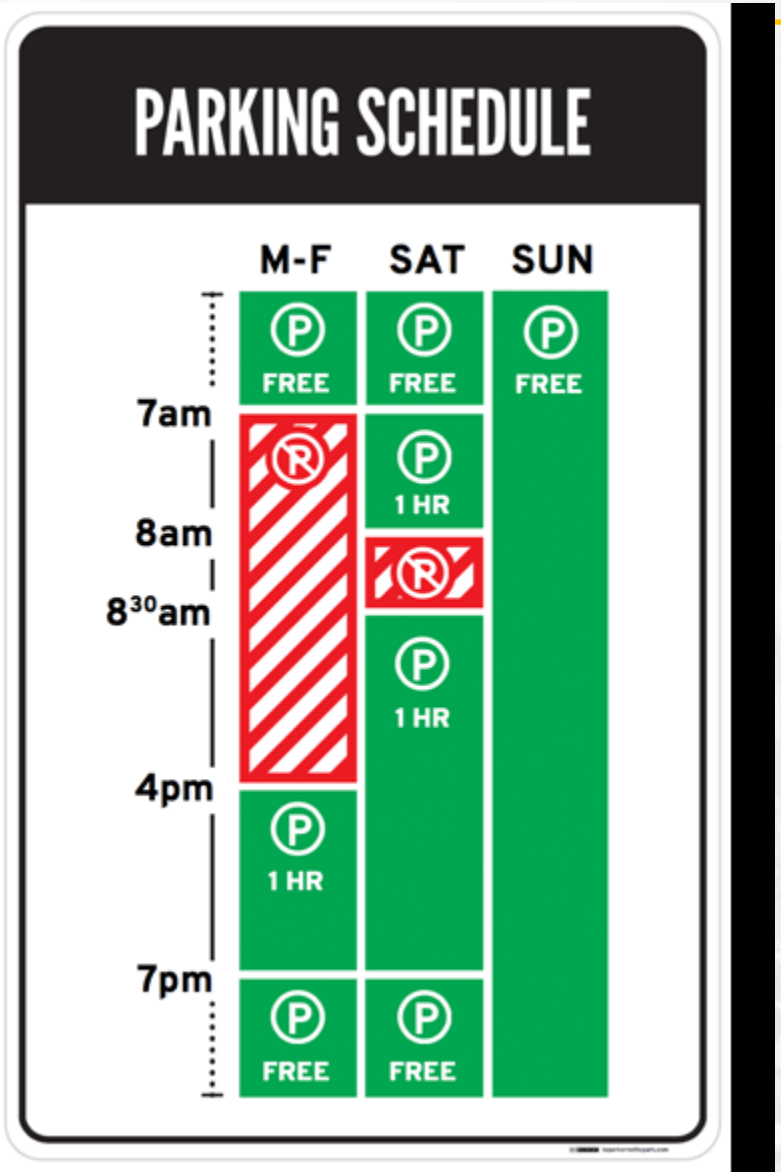
1. **Architectural Design -**
   i. is the highest abstract version of the system and identifies the software as a system with many components interacting with each other.
   ii. "the process of defining a collection of **hardware and software components** and their interfaces to establish the framework for the development of a computer system."
2. **High-level Design-**
   i. Breaks the Architectural Design **into less-abstracted view of sub-systems and modules** and depicts their **interaction** with each other.
3. **Detailed Design-**
   i. deals with the **implementation** part of what is seen as a system and its sub-systems
   ii. It defines **logical structure** of each module and their interfaces to communicate with other modules.

Software Design Levels

1. **Architectural Design**
2. **High-level Design-**
3. **Detailed Design-**

Good design and Bad Design





Good Design vrs Bad Design

Good Design	Bad Design
1. Meets all technical requirements	1. Meets only some technical requirements
2. Works all the time	2. Works initially but stops working after a short time
3. Meets cost requirements	3. Costs more than it should
4. Requires little or no maintenance	4. Requires frequent maintenance
5. Is safe	5. Poses a hazard to users
6. Creates no ethical dilemma	6. Raises ethical questions

What is a good design?

- A **good design** is focused.
- A **good design** is effective and efficient in fulfilling its purpose.
- It relies on as few external factors and inputs as possible, and these are easy to measure and manipulate to achieve an expected other output.
- A **good design** is always the simplest possible working solution.

Abstraction

- The principle of abstraction requires:
  - lower-level modules do not invoke functions of higher level modules.
  - Also known as layered design.
- High Level Design: High-level design maps functions into modules such that:
  - Each module has high cohesion
  - Coupling among modules is as low as possible
  - Modules are organized in a neat hierarchy

Modularity

- Modularity is a fundamental attributes of any good design.
  - Decomposition of a problem cleanly into modules:
  - Modules are almost independent of each other
  - Divide and conquer principle.
- If modules are independent:
  - Modules can be understood separately,
  - Reduces the complexity greatly.
  - To understand why this is so,
  - *Remember that it is very difficult to break a bunch of sticks but very easy to break the sticks individually.*

Modularity

- A module ideally should have **high cohesion** and **low coupling**:
- **Functionally independent** of other modules to have minimal interaction with other modules.
- Improves understandability and good design:
- Complexity of design is reduced as different modules are understood in isolation:
- A functionally independent module:
  - Can be easily taken out and reused in a different program.
  - Each module does some well-defined and precise function
  - The interfaces of a module with other modules is simple and minimal.

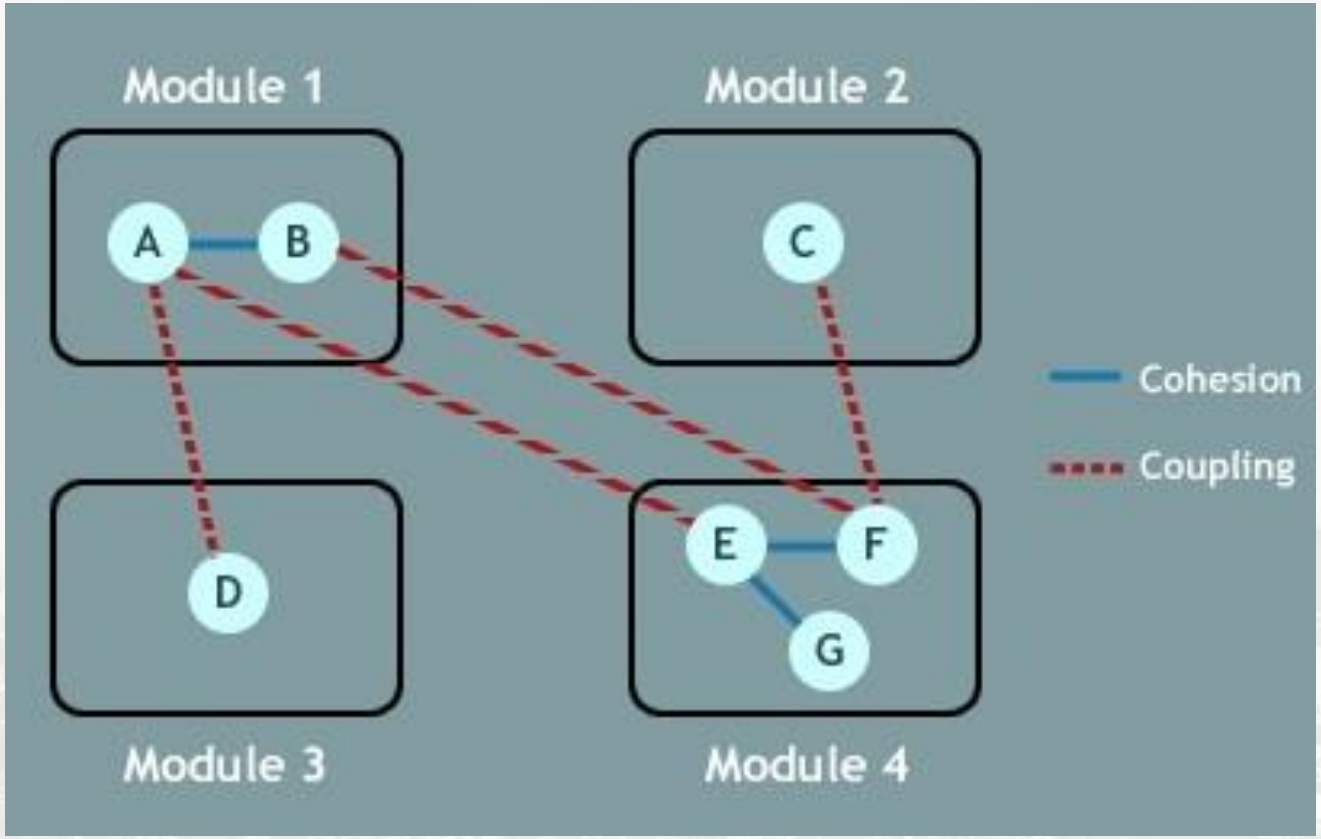
Modularity

- In technical terms, modules should display:
  - high cohesion
  - low coupling.
- Cohesion is a measure of:
  - Functional strength of a module.
  - A cohesive module performs a single task or function.
- Coupling between two modules:
  - A measure of the degree of interdependence or interaction between the two modules.

Cohesion and Coupling

Cohesion	Coupling
Cohesion is the indication of the relationship within **module**.	**Coupling** is the indication of the relationships between modules.
Cohesion shows the module's relative **functional** strength.	Coupling shows the relative **independence** among the modules.
Cohesion is a degree (quality) to which a component / module focuses on the **single** thing	Coupling is a degree to which a component / module is connected to the **other** modules.

Cohesion and Coupling



Ideally good Software Design will promote Tight cohesion and Low coupling
- Changes in the system

Transition from Analysis to Design

- **Analysis:** A process of extracting and organizing **user requirements** and establishing an **accurate model of the problem domain**.(WHAT)

- **Design:** Process of mapping requirements to a **system implementation** that conforms to desired cost, performance, and quality parameters.(HOW)

Transition from Analysis to Design

- Blurred line between analysis & design
- Process of design leads to better understanding of requirements
- Can be performed iteratively
- No direct & obvious mapping exists between structured analysis and structured design.

Forward Engineering

“Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system."

[ElliotChikofsky and JamesCross, Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software 7(1):13-17, 1990.]

Reverse Engineering

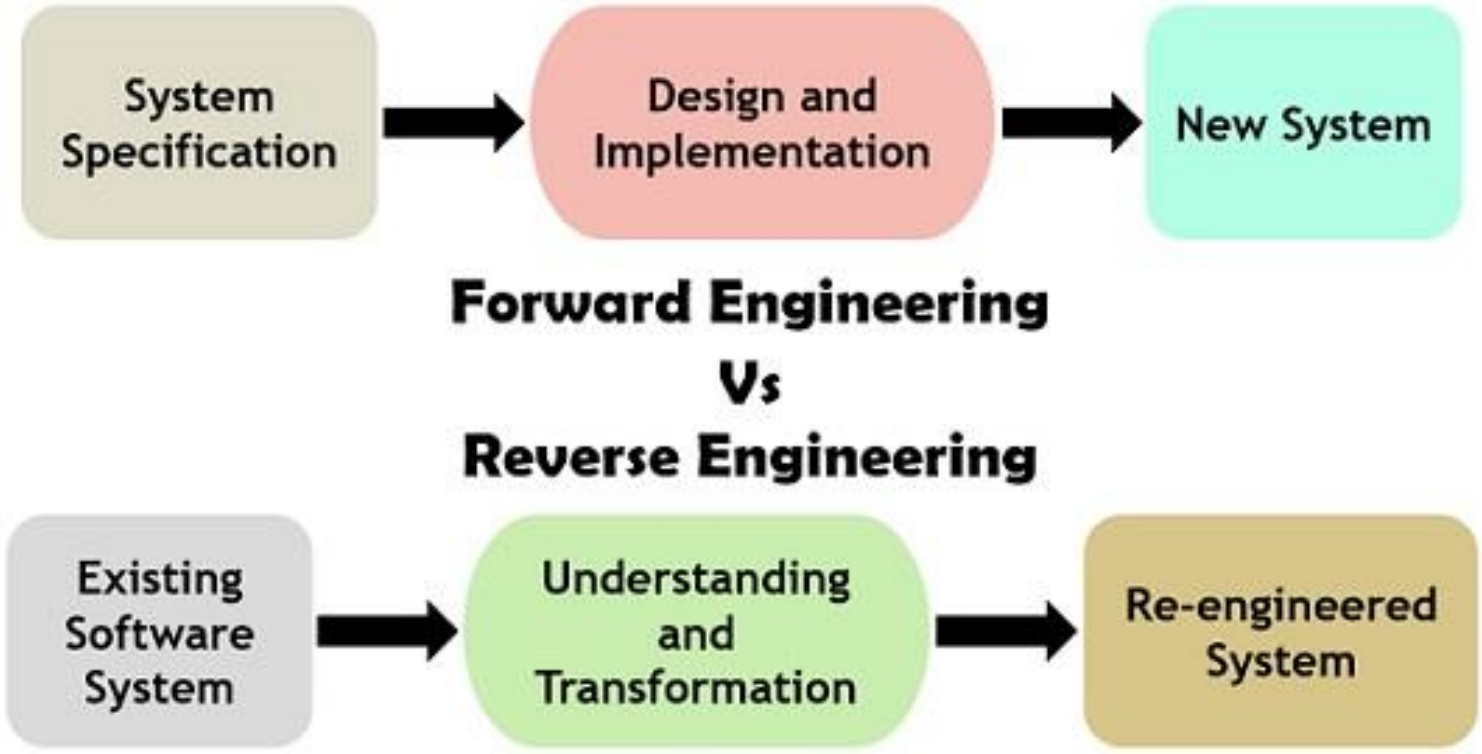
- Analyzing software with a view to understanding its design and specification
- May be part of a re-engineering process but may also be used to re-specify a system for re-implementation
- Builds a program data base and generates information from this
- Program understanding tools (browsers, cross-reference generators, etc.) may be used in this process

Reverse Engineering

- **“Reverse engineering”** is the process of analyzing a subject system with two goals in mind:
    1. To identify the system's components and their interrelationships; and,
    1. To create representations of the system in another form or at a higher level of abstraction."

[ElliotChikofsky and JamesCross, Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software 7(1):13-17, 1990.]

Forward vrs Reverse Engineering



Differences Between Forward Engineering and Reverse Engineering

- Forward engineering begins with the system specification and includes the design and implementation of the developing system.
  - Initial step in Reverse engineering starts with the existing system and the development technique for the replacement is based on interpretation.
- Forward engineering will generate a by-product
  - In reverse engineering, several ideas are generated about the requirement not necessarily generate a product.
- Forward engineering is prescriptive in nature where the developers need to follow particular rules for the proper results.
  - Reverse engineering is adaptive where the engineer has to discover what the developer actually did.
- Forward engineering consumes more time as compared to the reverse engineering.
- The final product of forward engineering must be complete and exact.
  - Reverse engineering model can be imperfect, retrieved partial information is still useful.

Two Approaches for Analysis & Design

- Traditional Approach: SSAD(System Structured Analysis and Design)
  - Structured Analysis (SA), resulting in a logical design, drawn as a set of **data flow diagrams**
  - Structured Design (SD) transforming the logical design into a program structure drawn as a set of **structure charts**

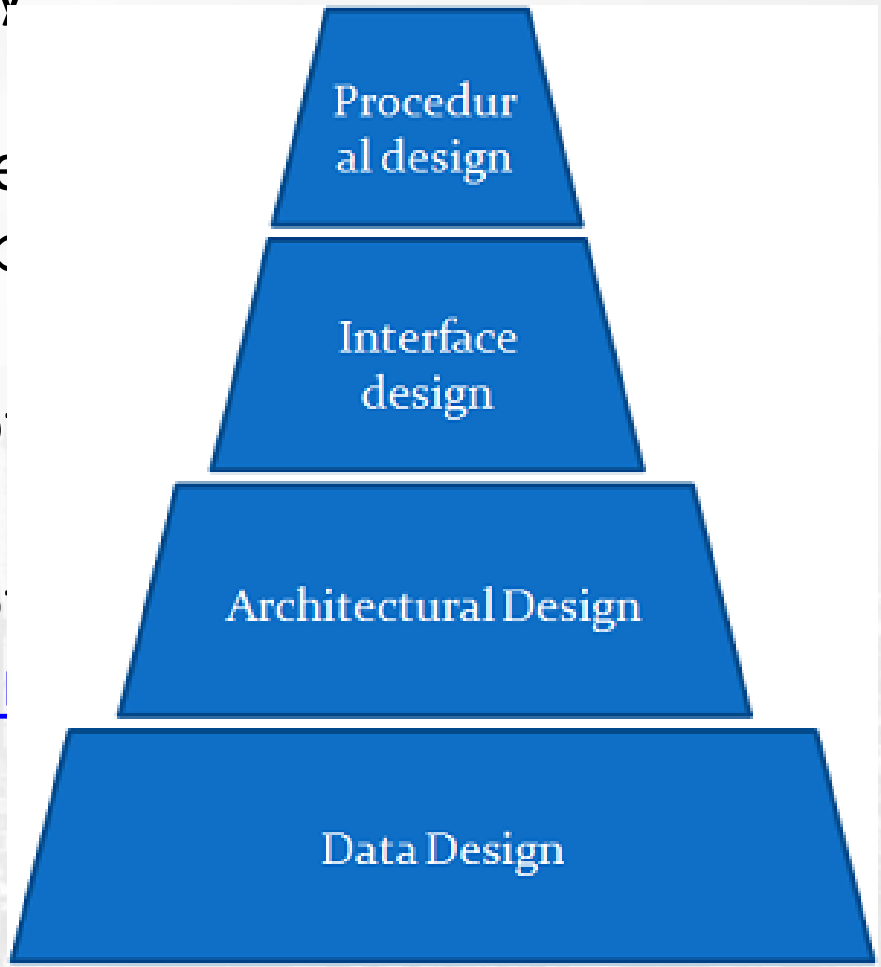
- OOAD(Object Oriented Analysis and Design)
  - Designing systems using self-contained objects and object classes

Structured Analysis

- Is a development method to understand the system and its activities in a logical way.
- It is a systematic approach that analyze and refine objectives of an system
- It has following attributes –
  - It is graphic which specifies the presentation of application.
  - It divides the processes so that it gives a clear picture of system flow.
  - It is logical rather than physical
  - It is an approach that works from high-level overviews to lower-level details.
- Structured Analysis Tools and techniques used for system development:
  - **Data Flow Diagrams**
  - **Entity Relation Diagram**
  - Data Dictionary
  - Decision Trees
  - Structured English
  - Pseudocode

Structured Analysis

- **Data design** transforms ERD (Entity Relationship Diagram) in to data structures
- **Architectural design** - higher-level structure of the system that depicts the relationship between the modules of the system
- **Interface design** defines interaction o people with the system
- **Procedural design** transforms the modules o the system architecture in to Pseudo-code o Flow Chart



Control Flow Diagrams

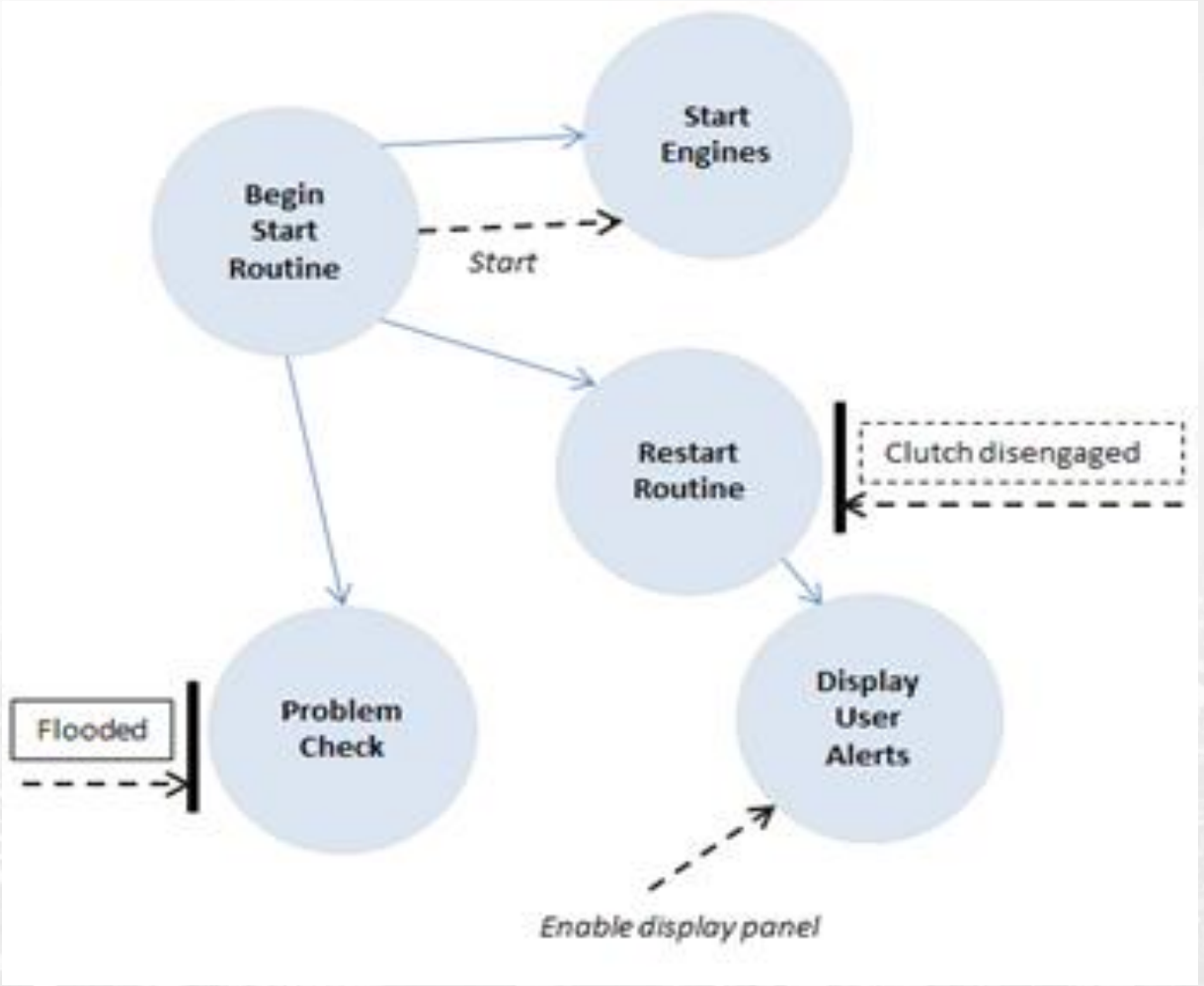
- A control flow diagram helps to understand the detail of a process.
- It shows us where control starts and ends and where it may branch off in another direction, given certain situations.
- *Let's say you are working on software to start a machine. What happens if the engine is flooded, or a spark plug is broken? Control then changes the flow to other parts of the software.*
- We can represent these branches with a diagram. The flow diagram is helpful because it can be understood by both stakeholders and systems professionals.
- Although some of the symbols might not be fully understood by the layperson, they can still grasp the general concept.

Control Flow Diagrams

- Even to the non-IT person, it is fairly clear that the software will attempt to start engines.
- If errors occur, then it will branch off into different directions.
- The general flow should make sense
  - Try to start the engines
  - Run a problem check
  - Display alerts if it can't start
  - Disengage the clutch
  - Try restarting
  - But what about the vertical lines and dashed arrows?

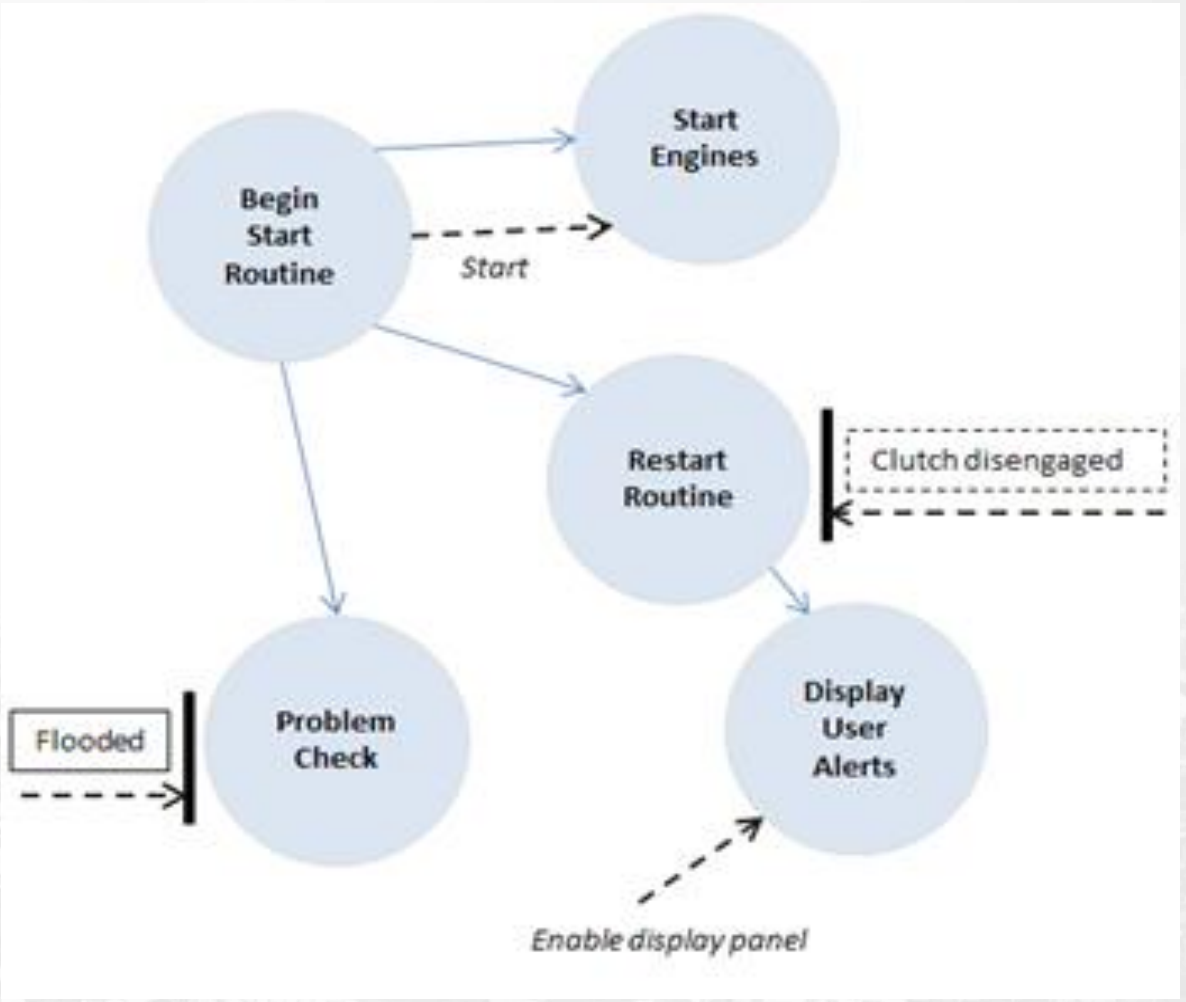
Control Flow Diagram

- Keeping with the idea of an engine-start program, the following shows flow through the software:
- **Vertical Bar**
  - The vertical bar indicates input or output from the current control specification.
  - Think of the bar as a window INTO or OUT OF the entire specification/flow. Additional notes, including an arrow, indicate further detail of the flow through that window.



Control Flow Diagram

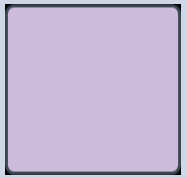
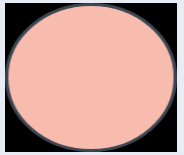
- **Dashed Arrow**
  - The input to the problem check step is *Flooded*
  - The input into the restart routine is *clutch disengaged*



Data Flow Diagrams

- The DFD (also known as a bubble chart) is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among these functions.
- Each function is considered as a processing station (or process) that consumes some input data and produces some output data.
- **The system is represented in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system.**

DFD Notations

S.No	Name	Description	Notation Yourdon and Coad	Gane and Sarson
1	**External Entity**	An outside system that sends or receives data, communicating with the system being diagrammed. They are the sources and destinations of information entering or leaving the system eg. an outside organization or person, a computer system or a business system. They are also known as terminators, sources and sinks or actors. They are typically drawn on the edges of the diagram.		
2	**Process**	Any process that changes the data, producing an output. It might perform computations, or sort data based on logic, or direct the data flow based on business rules		
3	**Data store**	Files or repositories that hold information for later use, such as a database table or a membership form.		
4	**Data flow**	The route that data takes between the external entities, processes and data stores.		

DFD Rules and Tips

- Each process should have at least one input and an output.
- Each data store should have at least one data flow in and one data flow out.
- Data stored in a system must go through a process.
- All processes in a DFD go to another process or a data store.

DFD

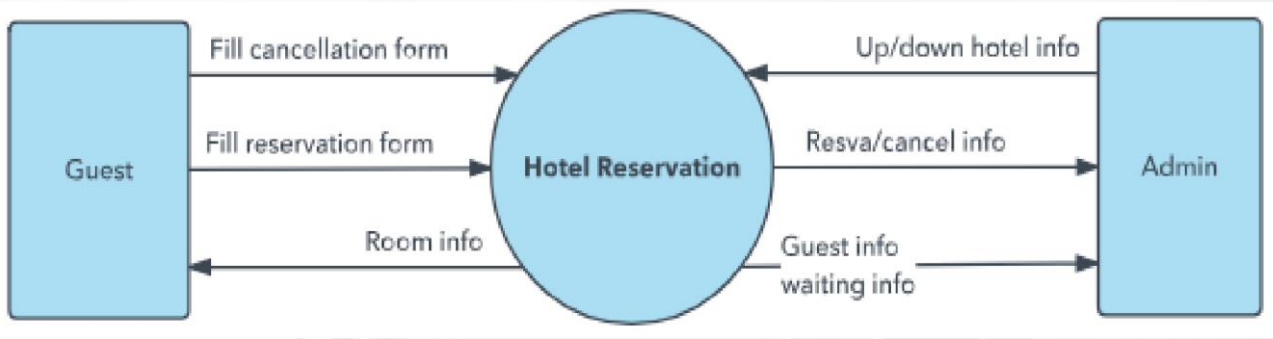
- A data flow diagram can dive into progressively more detail by using levels and layers, zeroing in on a particular piece. DFD levels are numbered 0, 1 or 2, and occasionally go to even Level 3 or beyond.
  - Level 0
  - Level 1
  - Level 2

Data-processing models

- Data flow diagrams are used to model the system's data processing
- These show the processing steps as data flows through a system
- Intrinsic part of many analysis methods
- Simple and intuitive notation that customers can understand
- Show end-to-end processing of data

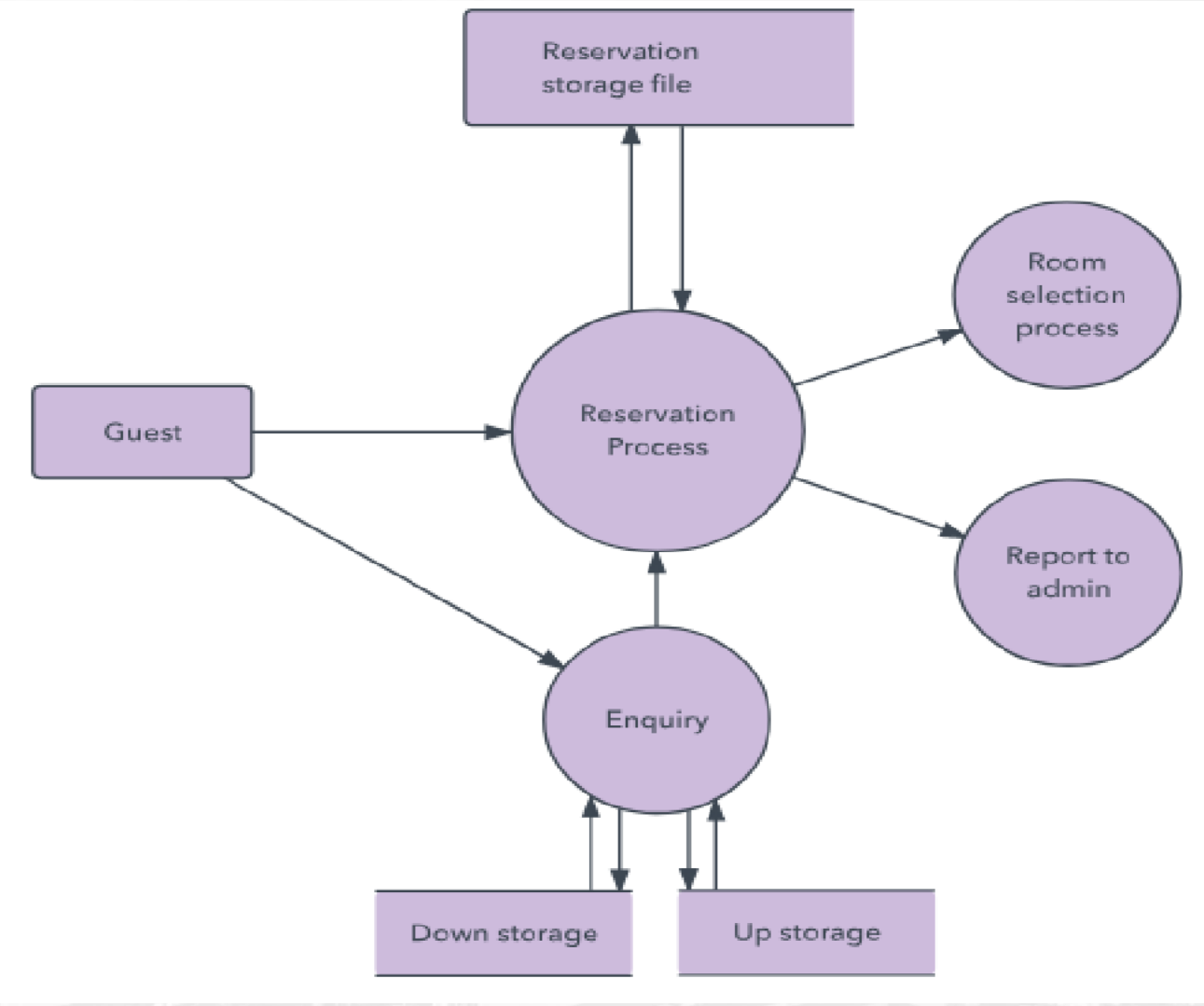
DFD- Level 0

- DFD Level 0 is also called a Context Diagram.
- It's a basic overview of the whole system or process being analyzed or modeled.
- It's designed to be an at-a-glance view, showing the system as a single high-level process, with its relationship to external entities.
- It should be easily understood by a wide audience, including stakeholders, business analysts, data analysts and developers.



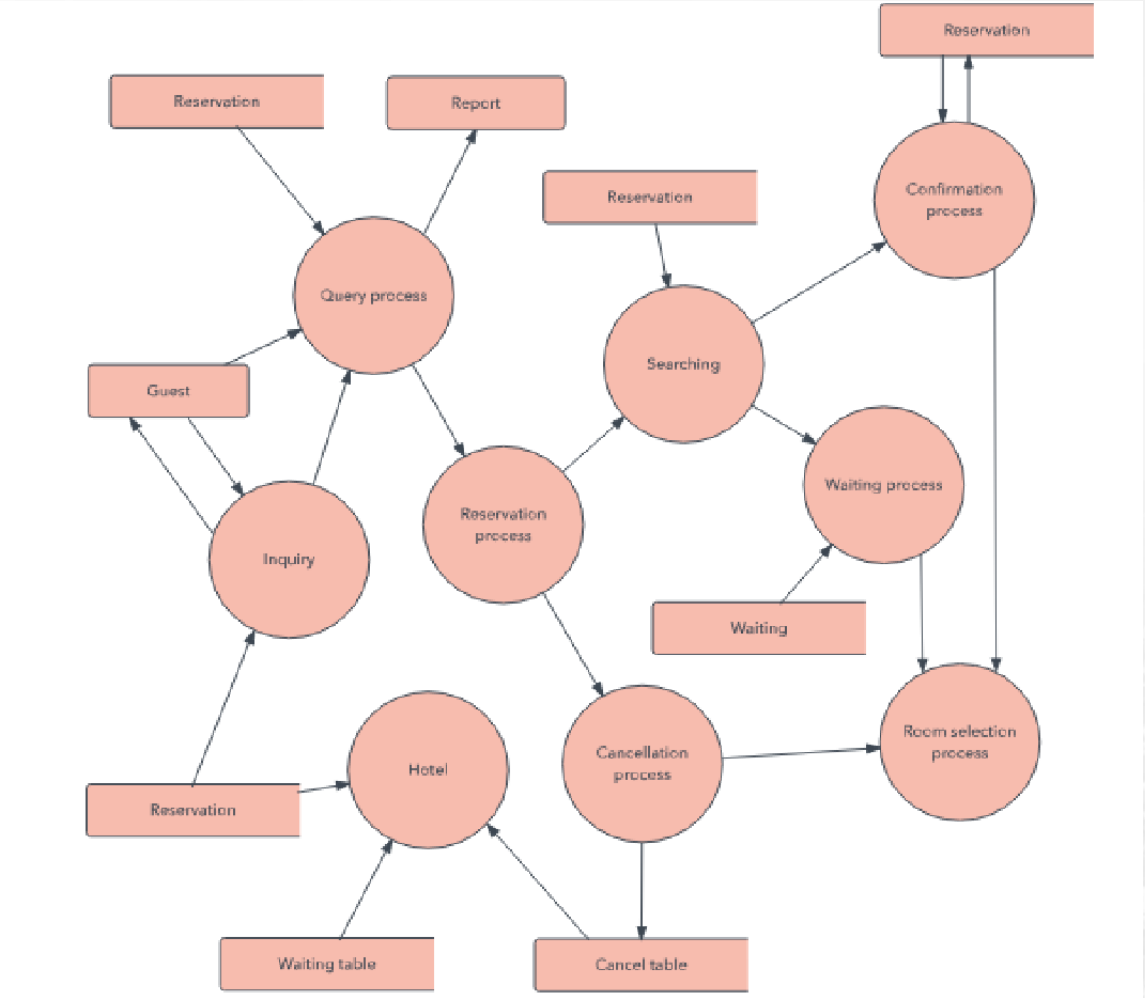
DFD- Level 1

- DFD Level 1 provides a more detailed breakout of pieces of the Context Level Diagram.
- You will highlight the main functions carried out by the system, as you break down the high-level process of the Context Diagram into its subprocesses.



DFD- Level 2

- DFD Level 2 then goes one step deeper into parts of Level 1. It may require more text to reach the necessary level of detail about the system's functioning.



Creating Data Flow Diagrams

Steps:

1. Create a list of activities
2. Construct Context Level DFD
   (identifies external entities and processes)
3. Construct Level 0 DFD
   (identifies manageable sub process)
4. Construct Level 1- n DFD
   (identifies actual data flows and data stores)
5. Check against rules of DFD

Example 2 : Lemonade Stand

Creating Data Flow Diagrams

Example

Group these activities in some logical fashion, possibly functional areas.

1. Create a list of activities

Customer Order
Serve Product
Collect Payment

Produce Product
Store Product

Order Raw Materials
Pay for Raw Materials

Pay for Labor

2. Construct Context Level DFD (identifies sources and sink)

Creating Data Flow Diagrams

Example

Create a level 0 diagram identifying the logical subsystems that may exist.

Customer Order
Serve Product
Collect Payment

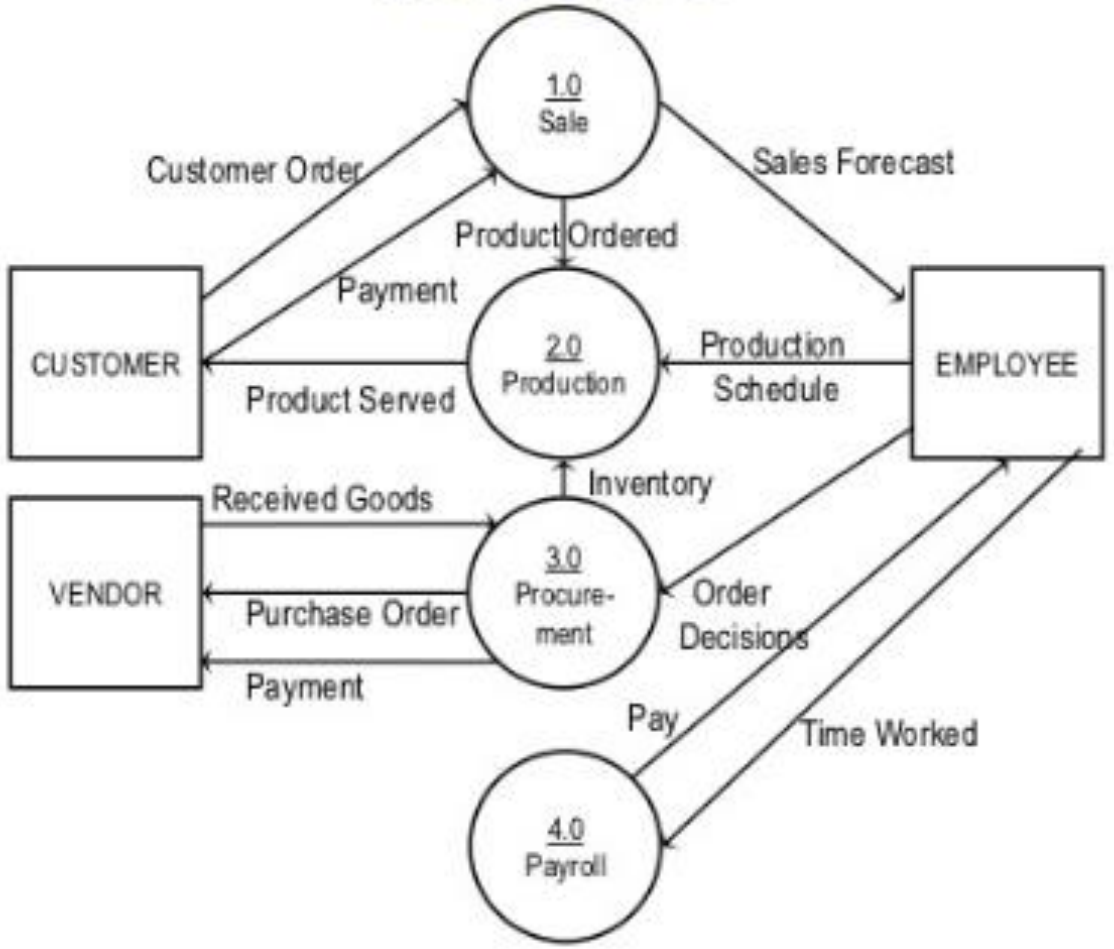
Produce Product
Store Product

Order Raw Materials
Pay for Raw Materials

Pay for Labor

3. Construct Level 0 DFD
(identifies manageable sub processes)

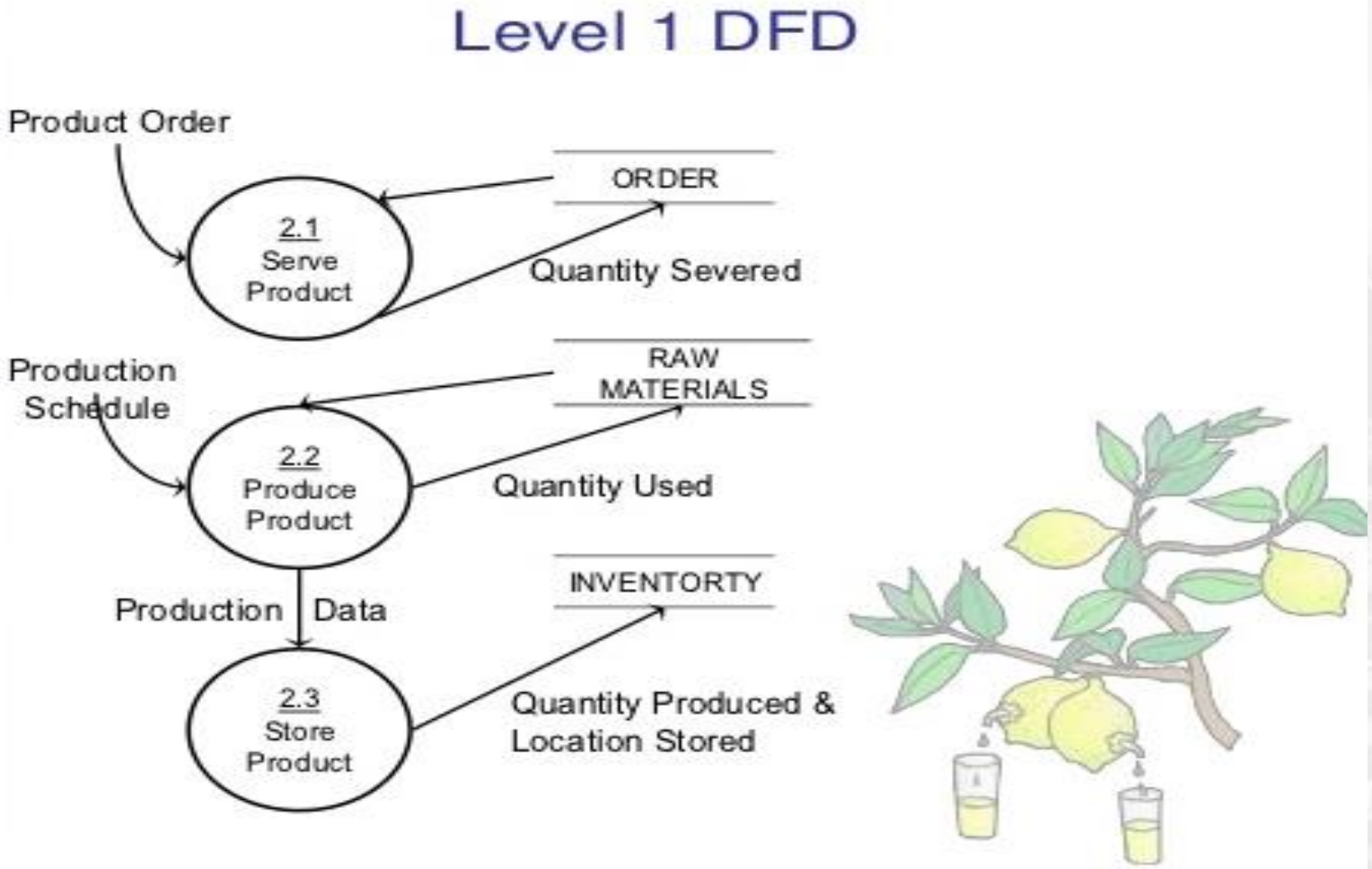
Level 0 DFD



4. Construct Level 1- n DFD
(identifies actual data flows and data stores)

Level 1

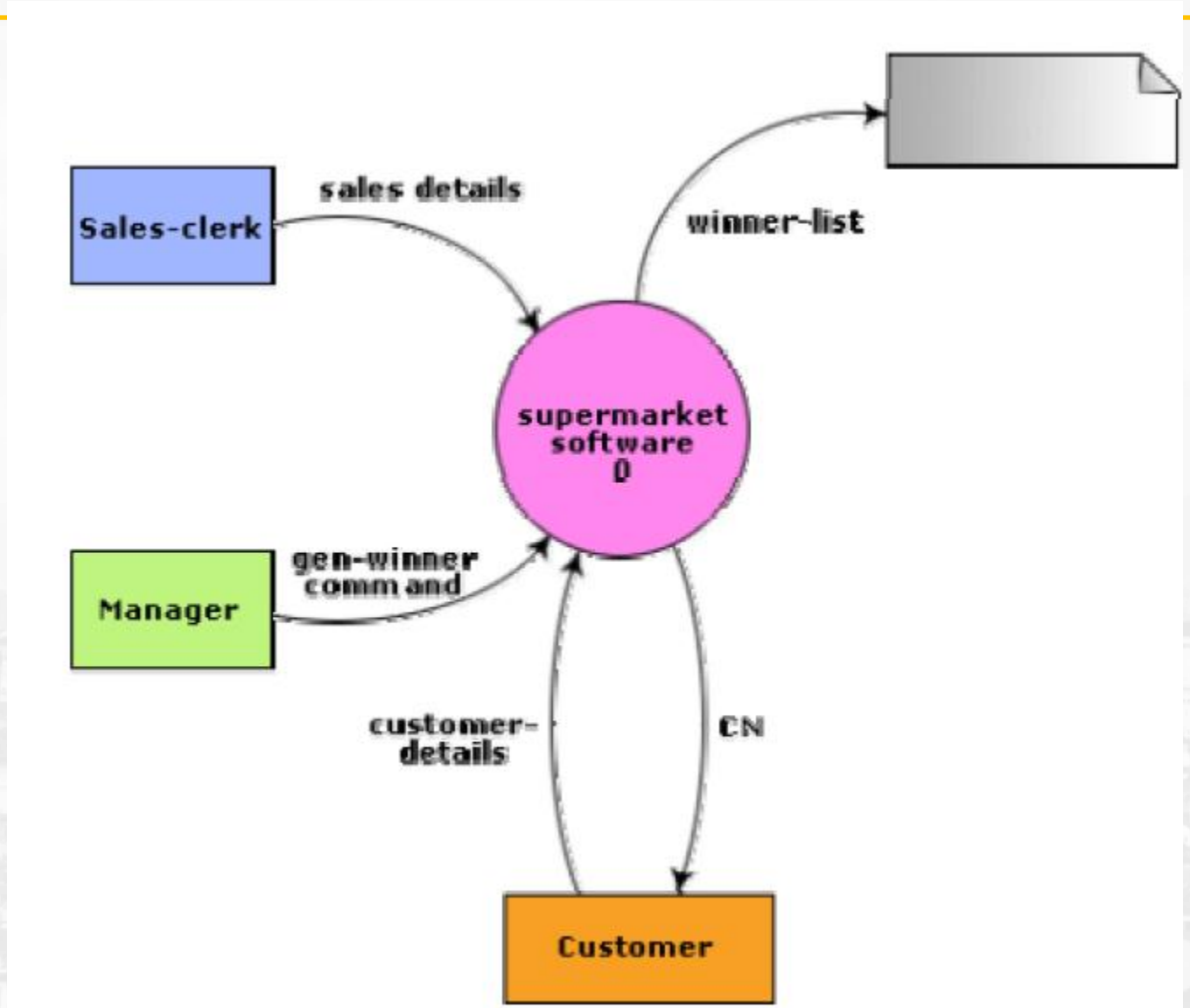
4. Construct Level 1 (continued)



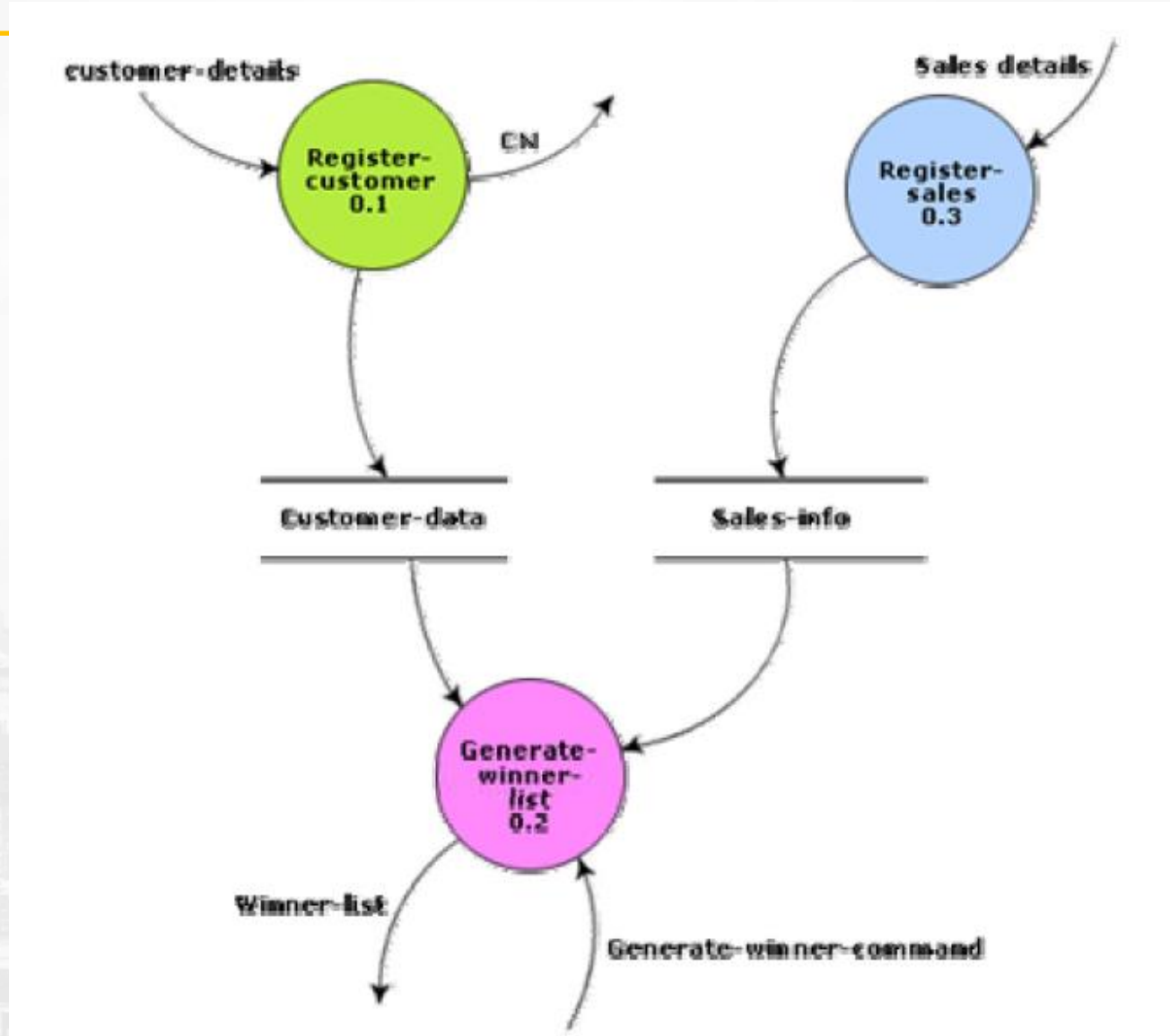
Example 3

- A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. A customer can present his CN to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded Rs.10,000. The entries against the CN are the reset on the day of every year after the prize winners' lists are generated.

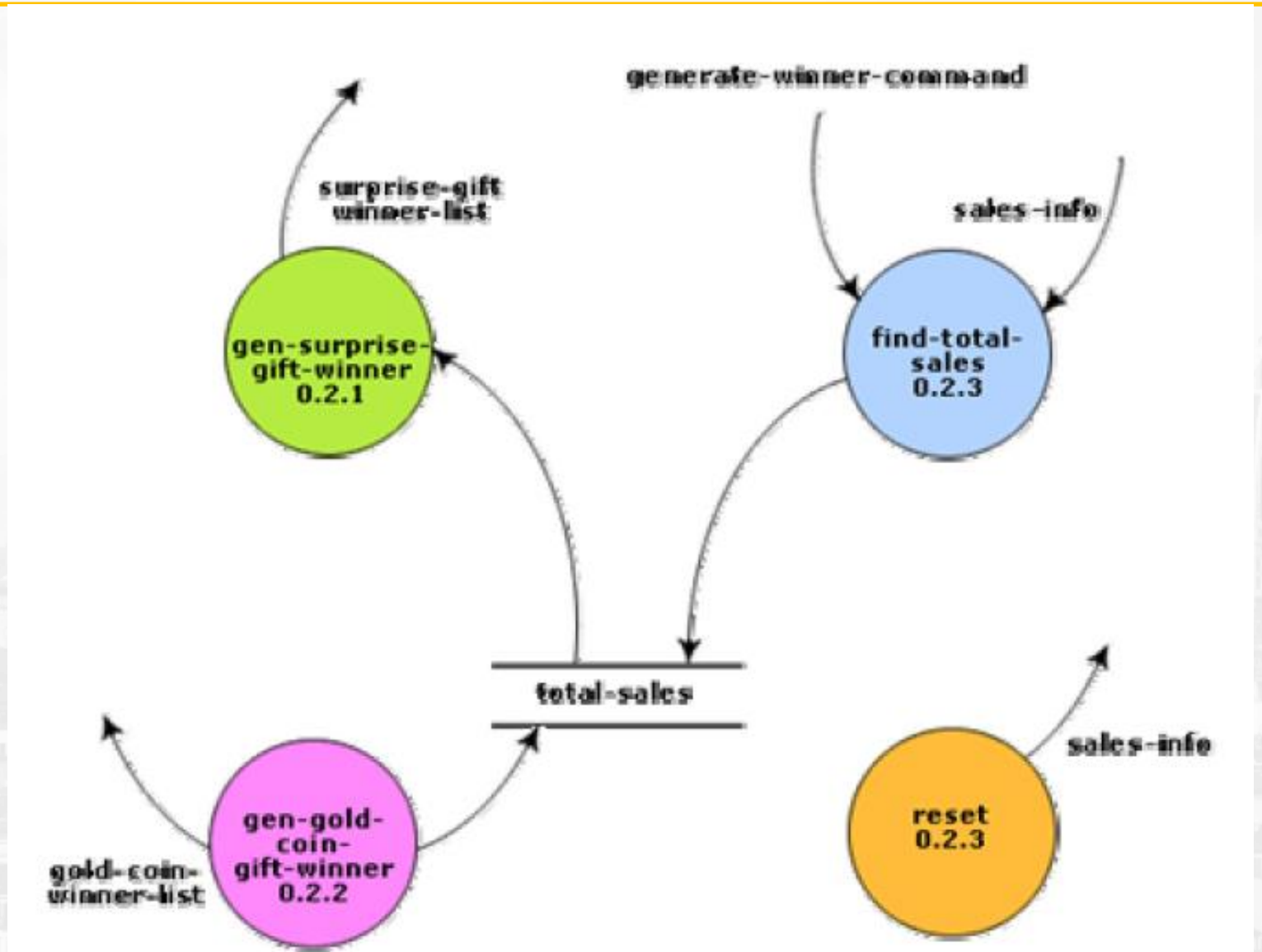
Level 0


Sales-clerk
sales details
winner-list
supermarket
software
0
Manager
gen-winner
command
customer-
details
CN
Customer

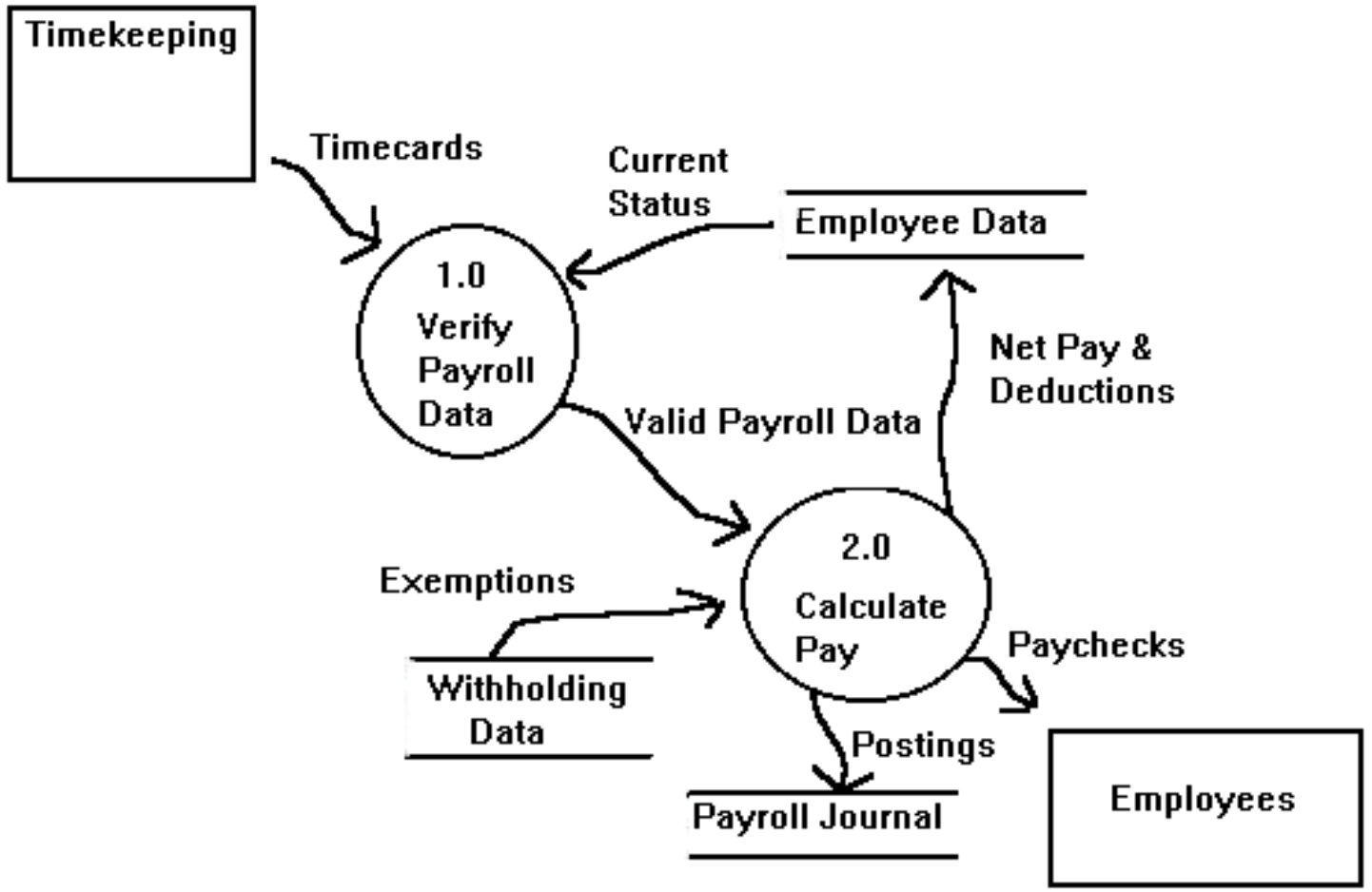
Level 1


customer-details
Register-customer 0.1
CN
Sales details
Register-sales 0.3
Customer-data
Sales-info
Generate-winner-list 0.2
Winner-list
Generate-winner-command

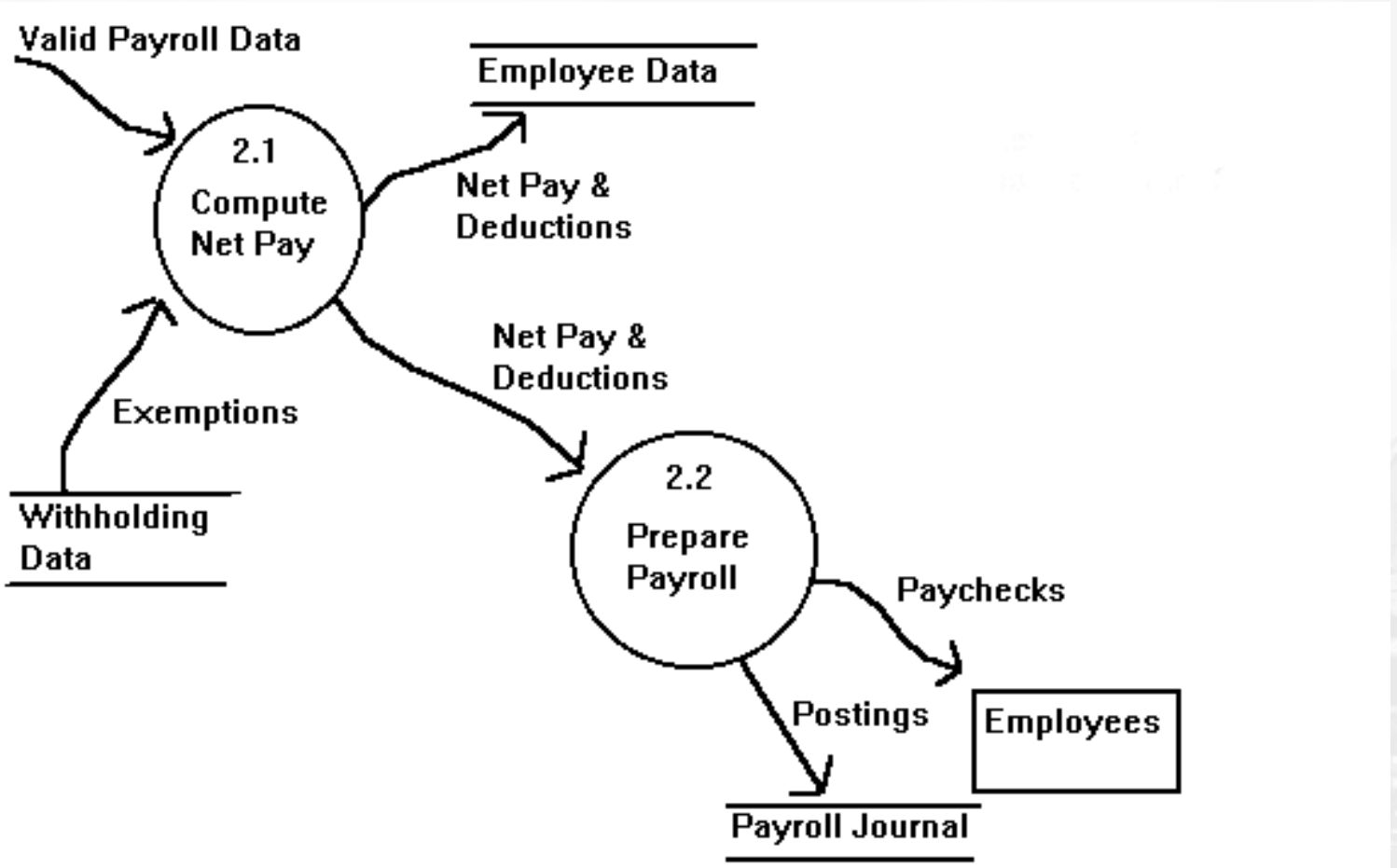
Level 2


generate-winner-command
surprise-gift winner-list
sales-info
gen-surprise-gift-winner 0.2.1
find-total-sales 0.2.3
total-sales
gold-coin-winner-list
gen-gold-coin-gift-winner 0.2.2
reset 0.2.3
sales-info

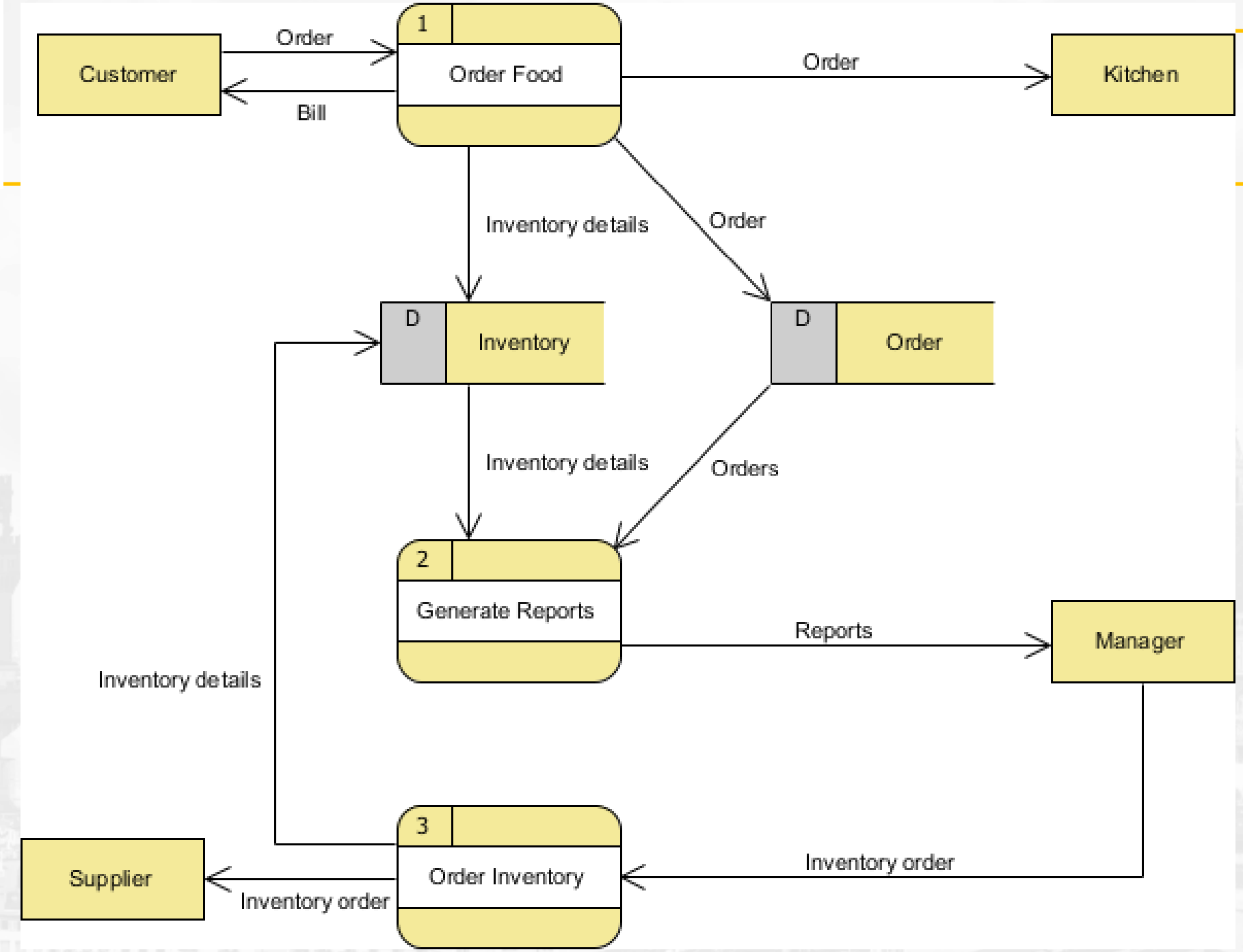
Level 1 :Payroll System



Level 2 :Payroll System



Level 1


Customer
Order
Bill
1
Order Food
Order
Kitchen
Inventory details
Order
D
Inventory
D
Order
Inventory details
Orders
2
Generate Reports
Reports
Manager
Inventory details
Supplier
3
Order Inventory
Inventory order
Inventory order

Entity Relationship Diagrams

- An ERD shows the relationships of entity sets stored in a database.
  - An entity in this context is an object, a component of data.
  - An entity set is a collection of similar entities.
  - These entities can have attributes that define its properties.
- ER diagram illustrates the logical structure of databases.
- ER diagrams are used to sketch out the design of a database.

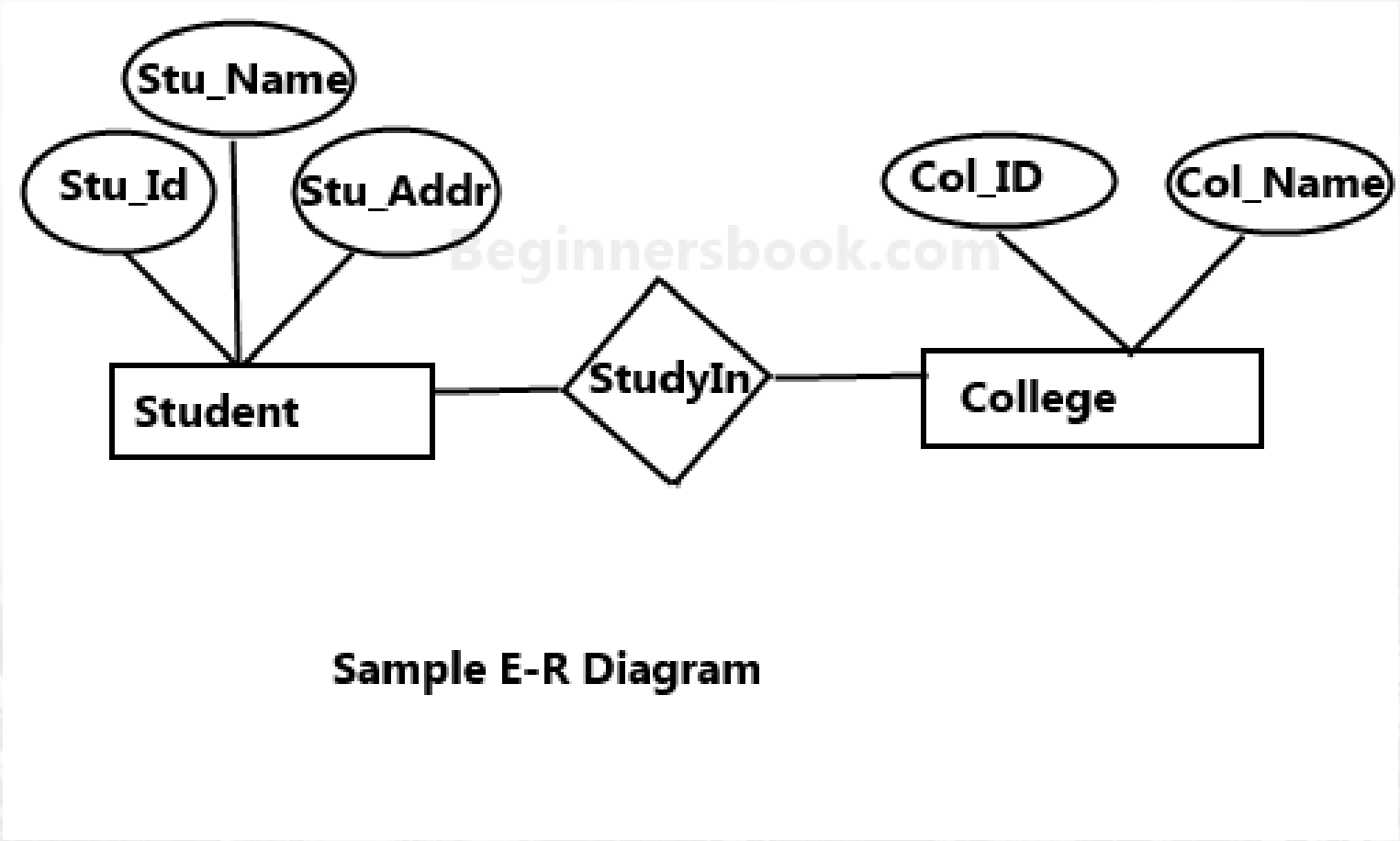
ER Diagram Notations

- Entities : is an object or concept about which you want to store information.

- Actions, which are represented by diamond shapes, show how two entities share information in the database.

- Attributes, which are represented by ovals. A key attribute is the unique, distinguishing characteristic of the entity.
  - A multivalued attribute can have more than one value. For example, an employee entity can have multiple skill values.
  - A derived attribute is based on another attribute. For example, an employee's monthly salary is based on the employee's annual salary.

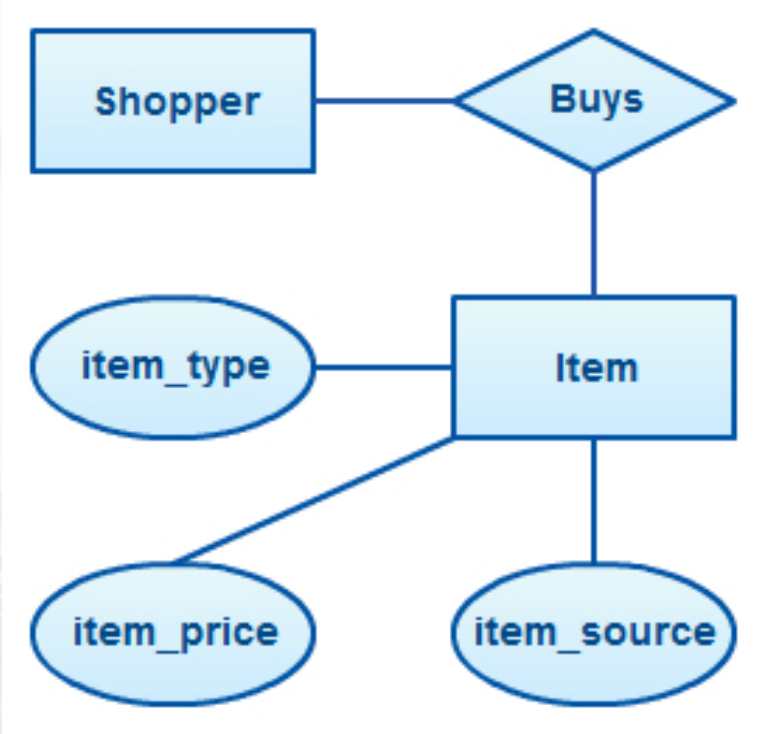
- Connecting lines, solid lines that connect attributes to show the relationships of entities in the diagram.



Sample E-R Diagram

Example 1

- Draw an ER diagram to show that shopper buys items.

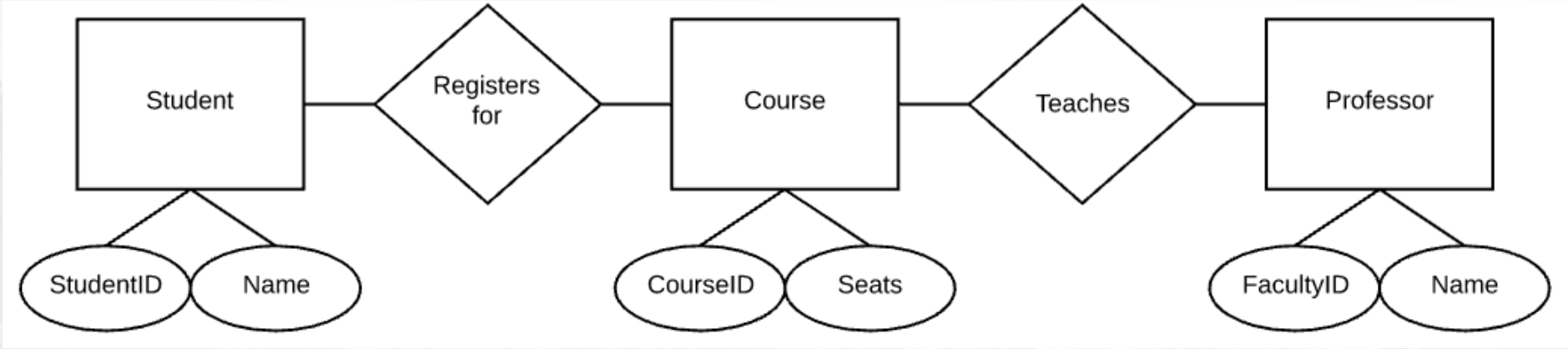


Example 2

- Draw an ER diagram to show that a student registers for a course. The course is taught by an Professor.

Example 2

- Draw an ER diagram to show that a student registers for a course. The course is taught by an Professor.

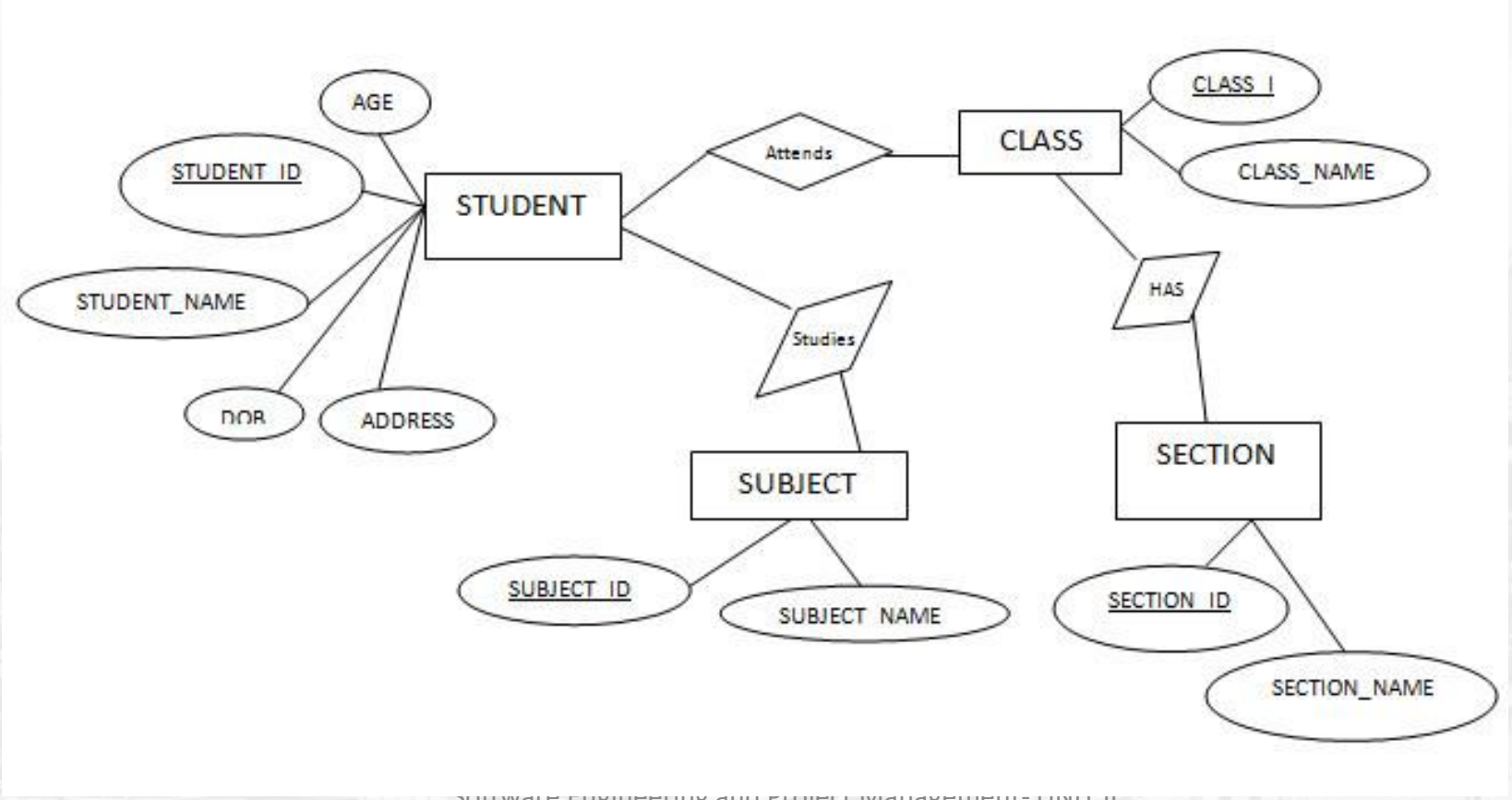


Example 3

- Draw ER diagram for : A student attends classes. Each class has many sections. The student takes up a subject.

Example 3

- Draw ER diagram for : A student attends classes. Each class has many sections. The student takes up a subject.



Object Oriented Analysis and design

- **Object-oriented analysis** is a process that groups items that interact with one another, typically by class, data or behavior, to create a model that accurately represents the system .
- The focus is on capturing the structure and behavior of systems into small modules that combines both data and process.

OOAD : Object Oriented Analysis and Design

- Object-Oriented Analysis : Is the procedure of identifying requirements and developing specifications by using interacting objects.
- The primary tasks in object-oriented analysis (OOA) are –
  - Identifying objects
  - Organizing the objects by creating object model diagram
  - Defining the internals of the objects, or object attributes
  - Defining the behavior of the objects, i.e., object actions
  - Describing how the objects interact
- **The common models used in OOA are use cases and object models.**

OOAD

- Object-Oriented Design: Involves implementation of the conceptual model produced during object-oriented analysis
- The implementation details generally include –
  - Restructuring the class data (if necessary),
  - Implementation of methods, i.e., internal data structures and algorithms,
  - Implementation of control, and
  - Implementation of associations.

Benefits of OOAD

Advantages	Disadvantages
Focuses on data rather than the procedures as in Structured Analysis.	Functionality is restricted within **objects**. This may pose a problem for systems which are intrinsically procedural or computational in nature.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	It cannot identify which objects would generate an optimal system design.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	The object-oriented models do not easily show the communications between the objects in the system.
It allows effective management of software complexity by the virtue of modularity.	All the interfaces between the objects cannot be represented in a single diagram.
It can be upgraded from small to large systems	

OO concept : Class and Object

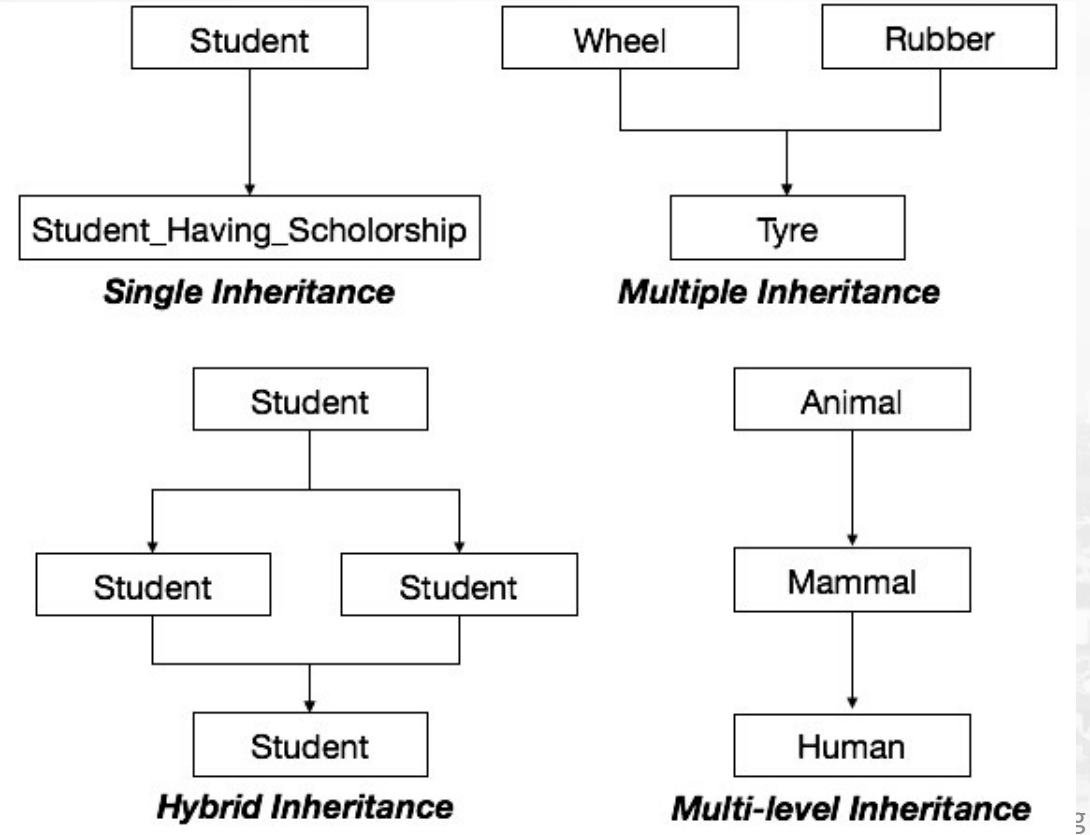
- Consider a class - Circle, that represents the geometrical figure circle in a two–dimensional space.
- The attributes of this class can be identified as follows −
  - x–coord, to denote x–coordinate of the center
  - y–coord, to denote y–coordinate of the center
  - a, to denote the radius of the circle
- Some of its operations can be defined as follows −
  - findArea(), method to calculate area
  - findCircumference(), method to calculate circumference
- During instantiation, - we create an object my_circle, we can assign values like x-coord : 2, y-coord : 3, and a : 4 to depict its state.
- And operations can be performed

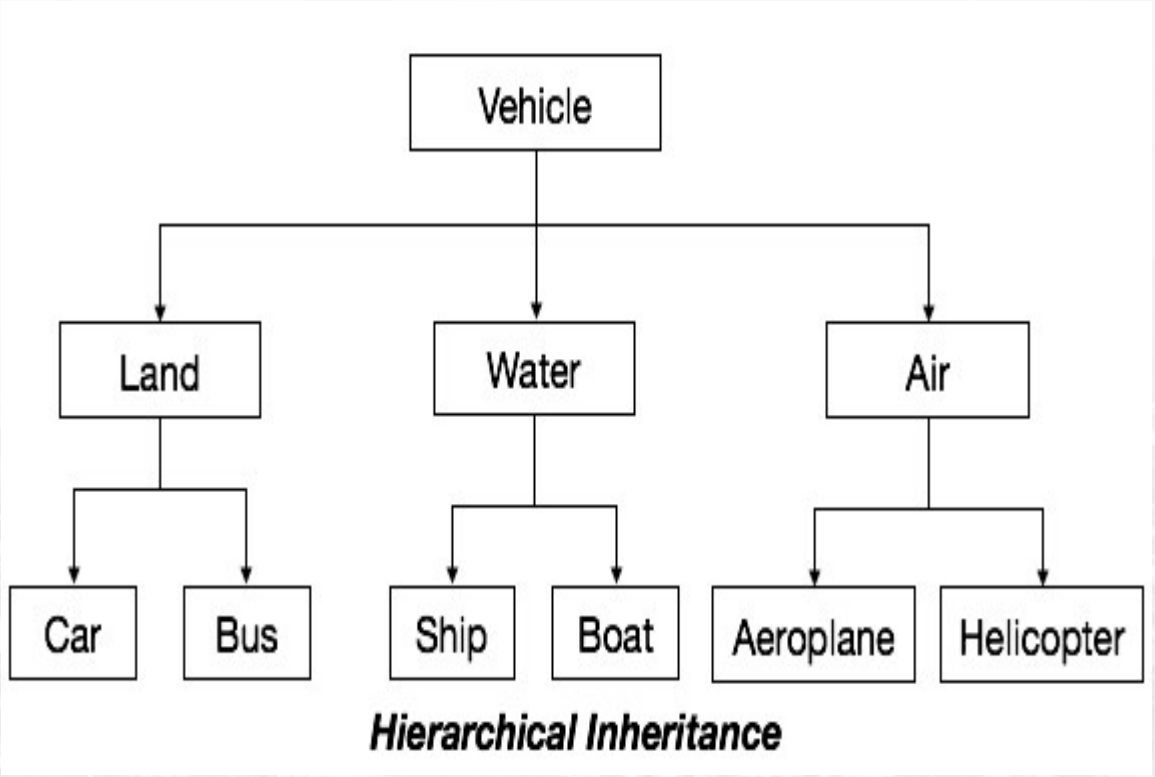
Message Passing

- Any application requires a number of objects interacting in a harmonious manner. Objects in a system may communicate with each other using message passing.
- The features of message passing are –
  - Message passing between two objects is generally unidirectional.
  - Message passing enables all interactions between objects.
  - Message passing essentially involves invoking class methods.
  - Objects in different processes can be involved in message passing.

Inheritance

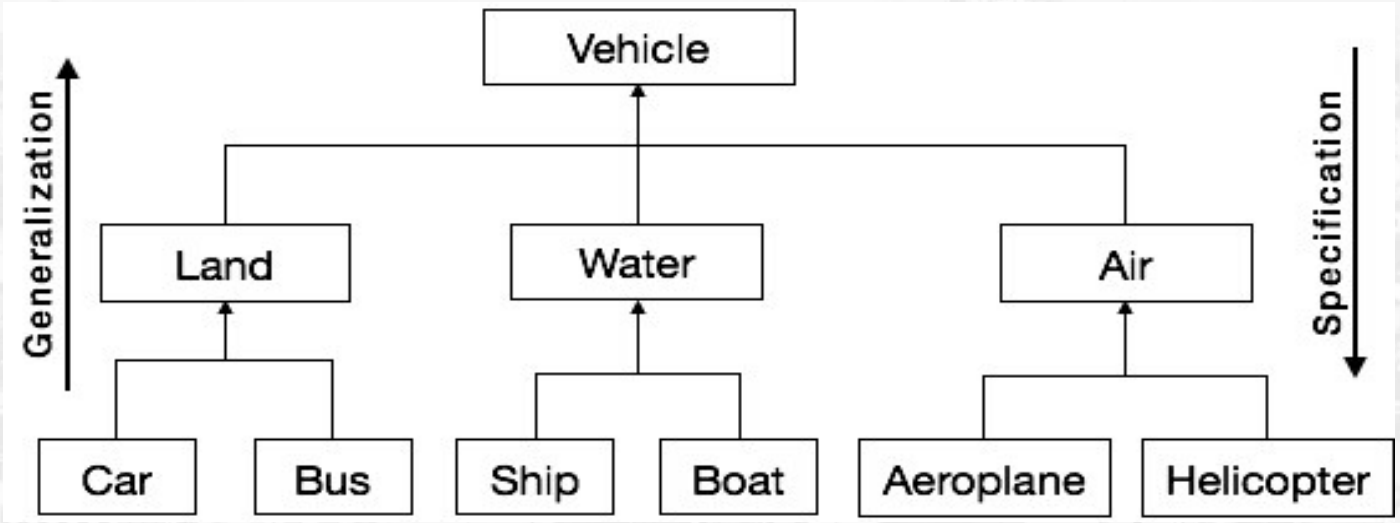
- Is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities.





Generalization and Specification

- Generalization : In the generalization process, the common characteristics of classes are combined to form a class in a higher level of hierarchy, i.e., subclasses are combined to form a generalized super-class
- Specialization :Specialization is the reverse process of generalization. Here, the distinguishing features of groups of objects are used to form specialized classes from existing classes.



Aggregation and Composition

- Aggregation or composition is a relationship among classes by which a class can be made up of any combination of objects of other classes. It allows objects to be placed directly within the body of other classes.
- Aggregation is referred as a “part–of” or “has–a” relationship, with the ability to navigate from the whole to its parts. An aggregate object is an object that is composed of one or more other objects.

Dynamic Modelling

- After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined. This is the purpose of dynamic modelling.
- Dynamic Modelling can be defined as “a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world”.

Functional Modelling

- The functional model shows the processes that are performed within an object and how the data changes as it moves between methods. It specifies the meaning of the operations of object modelling and the actions of dynamic modelling.
- The process of functional modelling can be visualized in the following steps –
  - Identify all the inputs and outputs
  - Construct data flow diagrams showing functional dependencies
  - State the purpose of each function
  - Identify constraints
  - Specify optimization criteria

Unified Modeling Language (UML)

- **UML** (Unified Modeling Language) is a modeling language **used** by software developers.
- **UML** can be **used** to develop diagrams and provide users with ready-to-**use**, expressive modeling examples.
- Some **UML** tools generate program language code from **UML**.
- It's a rich language to model software solutions, application structures, system behavior and business processes
- **UML** can be **used** for modeling a system independent of a platform language.

What is UML?

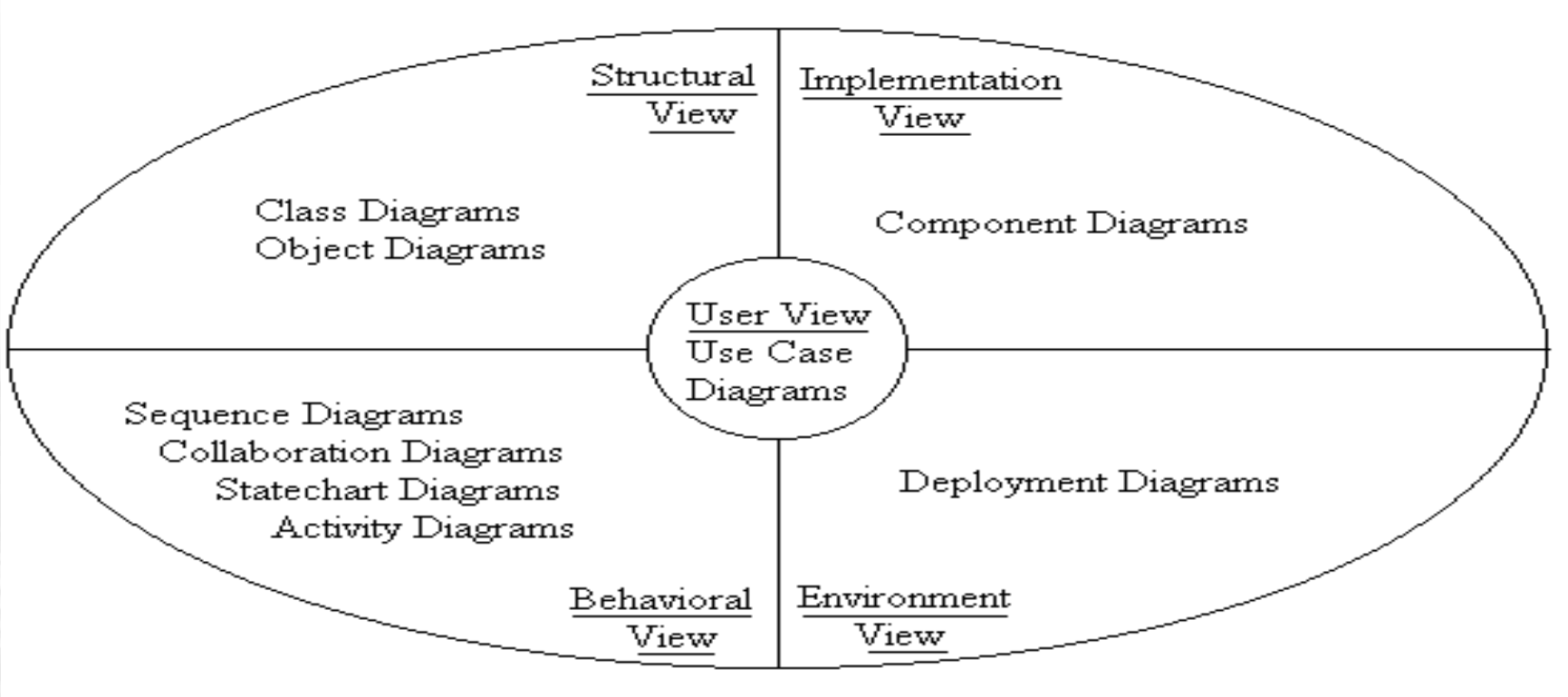
- UML (Unified Modeling Language)
  - An emerging standard for modeling object-oriented software.
  - Resulted from the convergence of notations from three leading object-oriented methods:
    - OMT (James Rumbaugh)
    - OOSE (Ivar Jacobson)
    - Booch (Grady Booch)
- Reference: “The Unified Modeling Language User Guide”, Addison Wesley, 1999.
- Supported by several CASE tools
    - Rational ROSE
    - TogetherJ

UML Concepts

- UML can be used to support your entire life cycle
  - UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.
  - The interaction of your application with the outside world (use case diagram)
  - Visualize object interaction (sequence & collaboration diagrams)
  - The structure of your system (class diagram)
  - View the system architecture by looking at the defined package.
  - The components in your system (component diagram)

UML supported diagrams

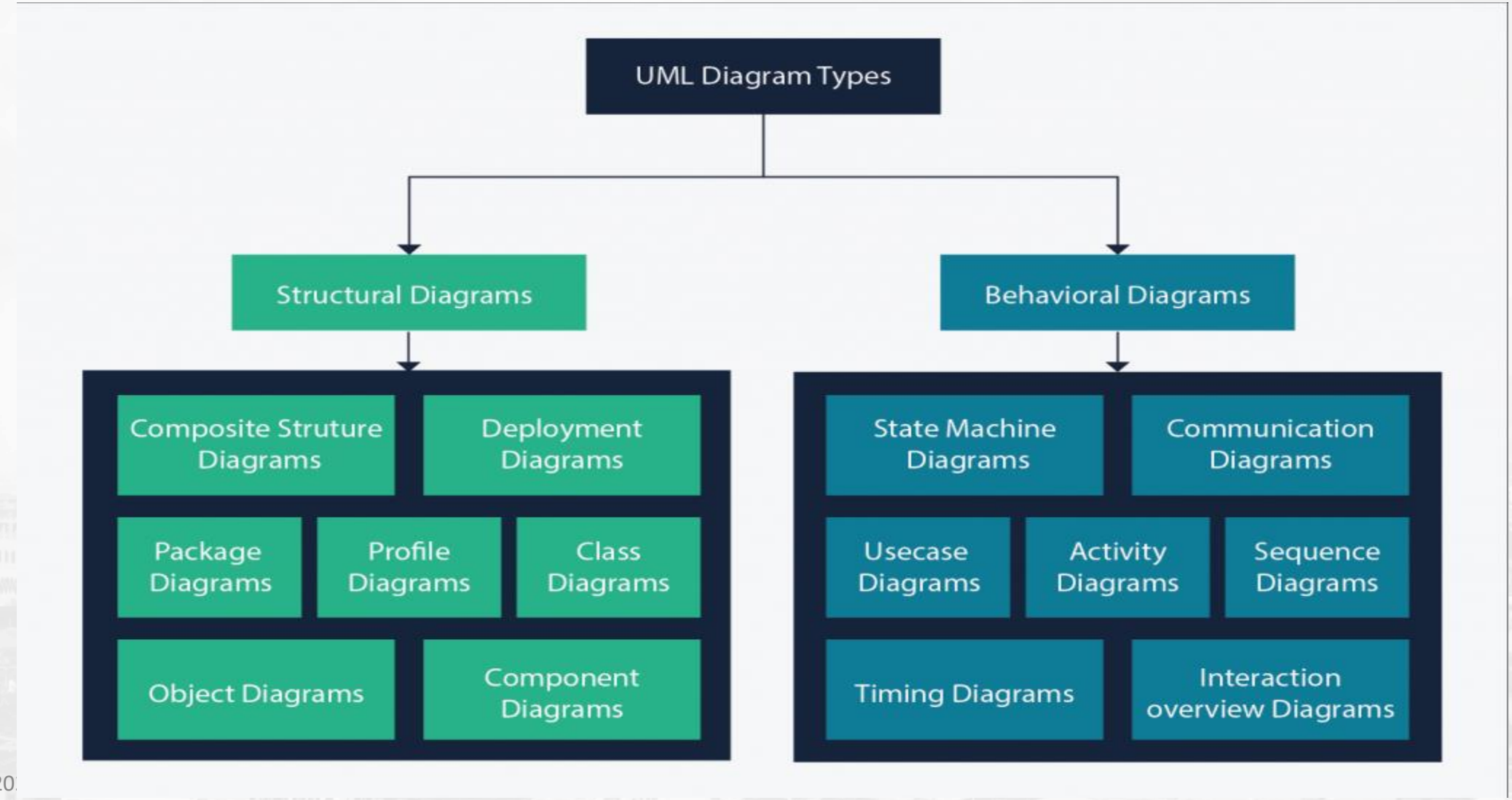
- UML Meta Model: A diagram that defines the notation to be used in the modeling language



Model Terminology

- User model view - problem and solution from the perspective of the users
- Structural model view - static or structural aspects of a problem and solution
- Behavioral model view - dynamic or behavioral aspects; interactions or collaborations among problem and solution elements
- Implementation model view - structural and behavioral aspects of the solution's realization
- Environment model view - structural and behavioral aspects of the domain in which a solution must be realized

UML Diagrams



Brief Overview

- **Class** - Set of classes, interfaces, collaboration, relationships
- **Object** - Set of objects and their relationships
- **Use case** - Description of functionality provided by system along with actors and their connection to the use case
- **Interaction** - Set of objects and their relationships including messages
- **State/statechart** - A state machine showing states, transitions, events, and activities
- **Activity** - Statechart sequential flow of activities
- **Component** - physical structure of code in terms of code components
- **Deployment** - physical architecture of hardware and software in the system
- **Package** - shows packages of classes and dependencies among them
  - Grouping mechanism
  - Form of class diagram
  - Also called subsystem

Use Case Diagram

- A use case diagram describes how a system interacts with outside actors
- It is a graphical representation of the interaction among the elements and system
- Each use case represents a piece of functionality that a system provides
- The use case diagram allows for the specification of higher level user goals
- A use case diagram contains four components
  - The boundary
  - The actor
  - The use case
  - The relationship between and among the actors and the use cases

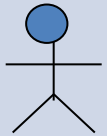
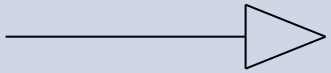
Use Case Diagram

- Use case is one way of representing system functionality.
- Use case refers to
  - A system's behavior (functionality)
  - A set of activities that produce some output.
- Think in terms of main processes happening in the system.

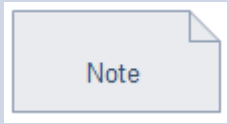
In the simplest form, use case is a list of functions for a user;

- Use verbs for naming use cases.

Use Case Diagram

S.No	Name	Description	Notation
1	System Boundary	The scope of a system can be represented by a system boundary	
2	Use case	A sequences of actions (it must be a verb)	
3	Actor	User (or) someone / something outside the system that interacts with the system (it must be a noun)	
4	Association	It corresponds to a sequence of actions between the actor and use case	
5	Generalization	Inheritance relationship between model elements of same type	

Use Case Diagram

S.No	Name	Description	Notation
6	Include	It specifies how the behavior of the inclusion use case is inserted into the behavior defined for the base use case	<<include>> ----------▸
7	Extend	How the behavior of the extension use case can be inserted into the behavior defined for the base use case	<<extend>> ----------▸
9	Note	Note is generally used to write comment in use case diagram	

Actors

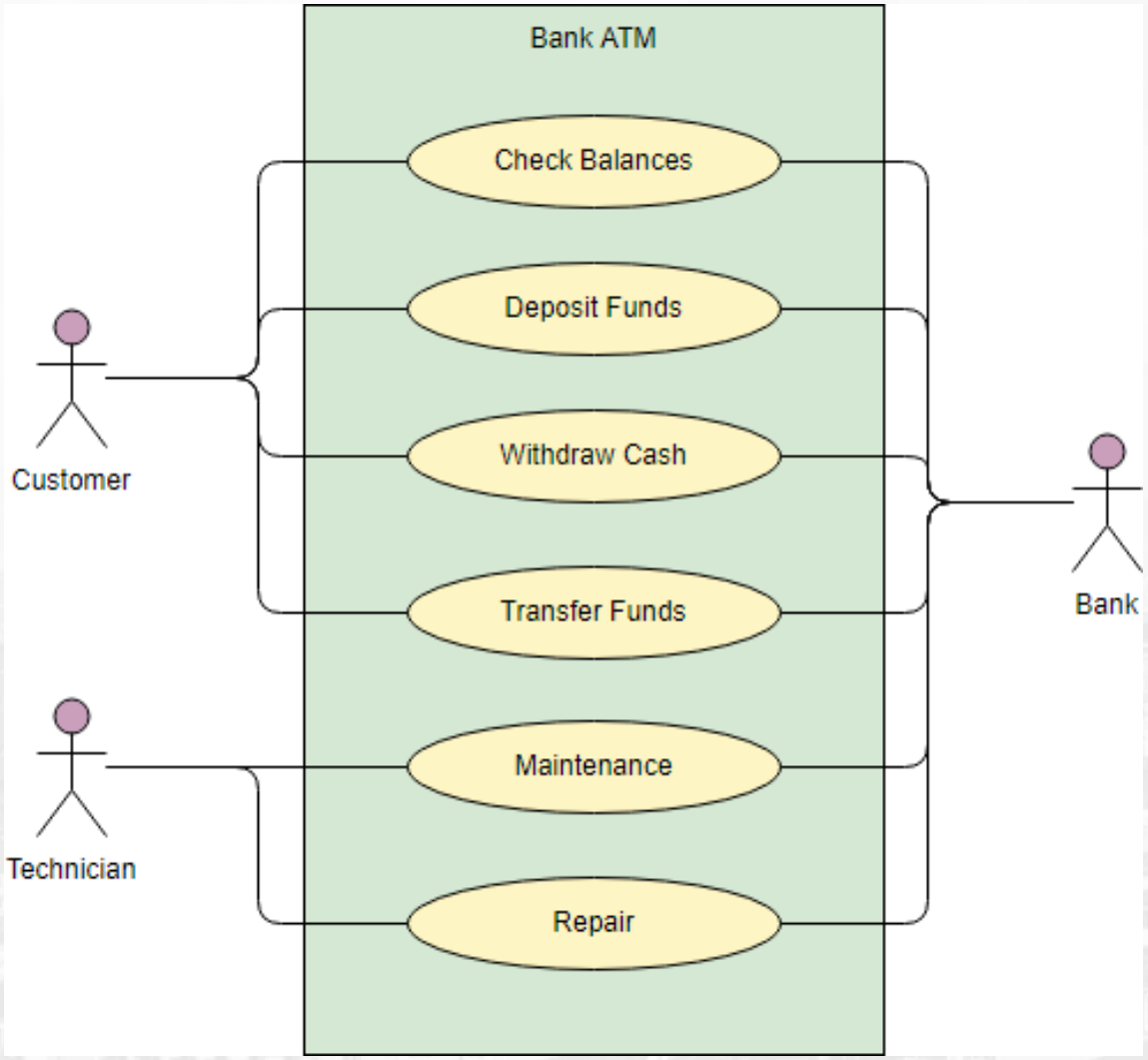
- Actor *triggers* use case.

- Each Actor must be linked to a use case, while some use cases may not be linked to actors.

- Actor—Use Case: Actor has responsibility toward the system (inputs), and Actor have expectations from the system (outputs).

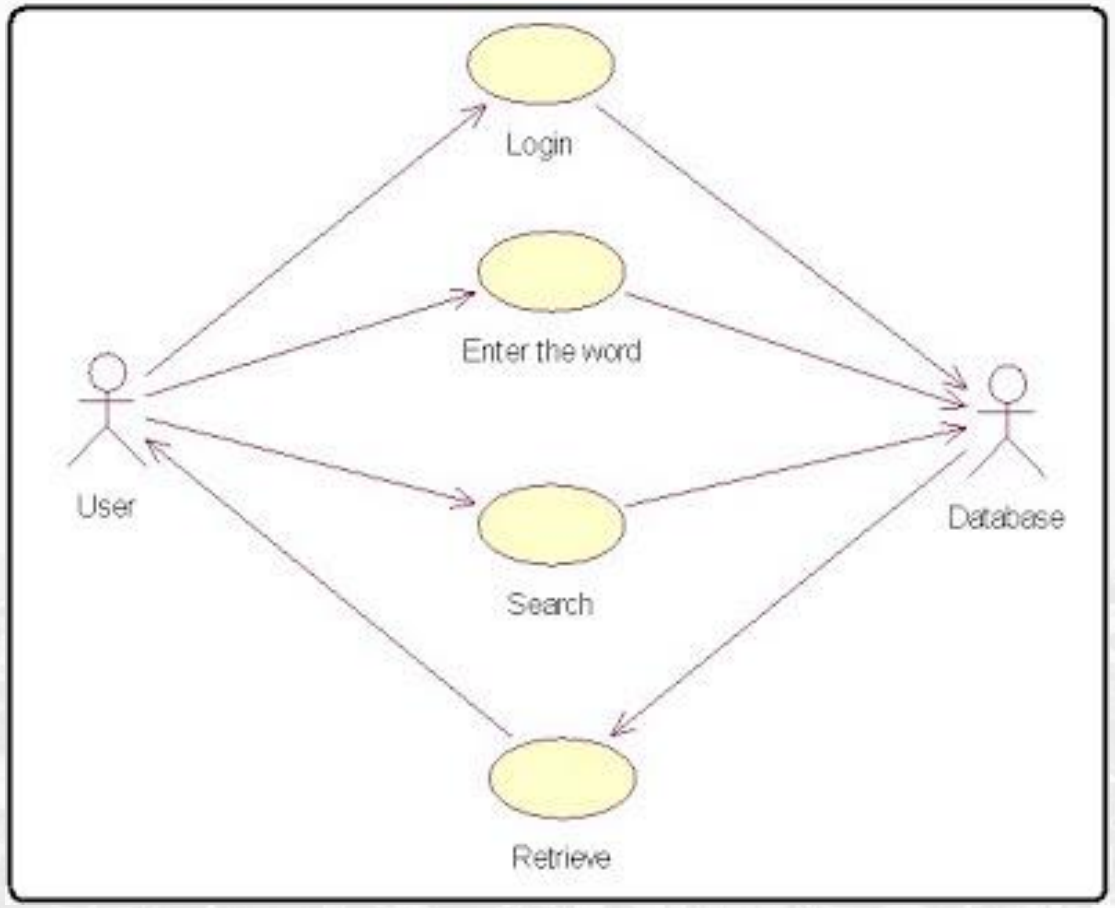
Careful: Lines between Actors and Use Cases are NOT data flows!

Use Case Diagram for ATM



Example 2: **Use case Diagram for** a dictionary system

- In this system, the user first login to the system with user name and password. user name of every user is unique.
- The user enters the word to the system for finding meaning of the word.
- The system searches the meaning for the given word.
- If the word is available on the database then the system retrieve and display the meaning for the word otherwise it show the message box that contains please enter correct word.


Login
Enter the word
User
Database
Search
Retrieve

Extend Relation between Use Cases

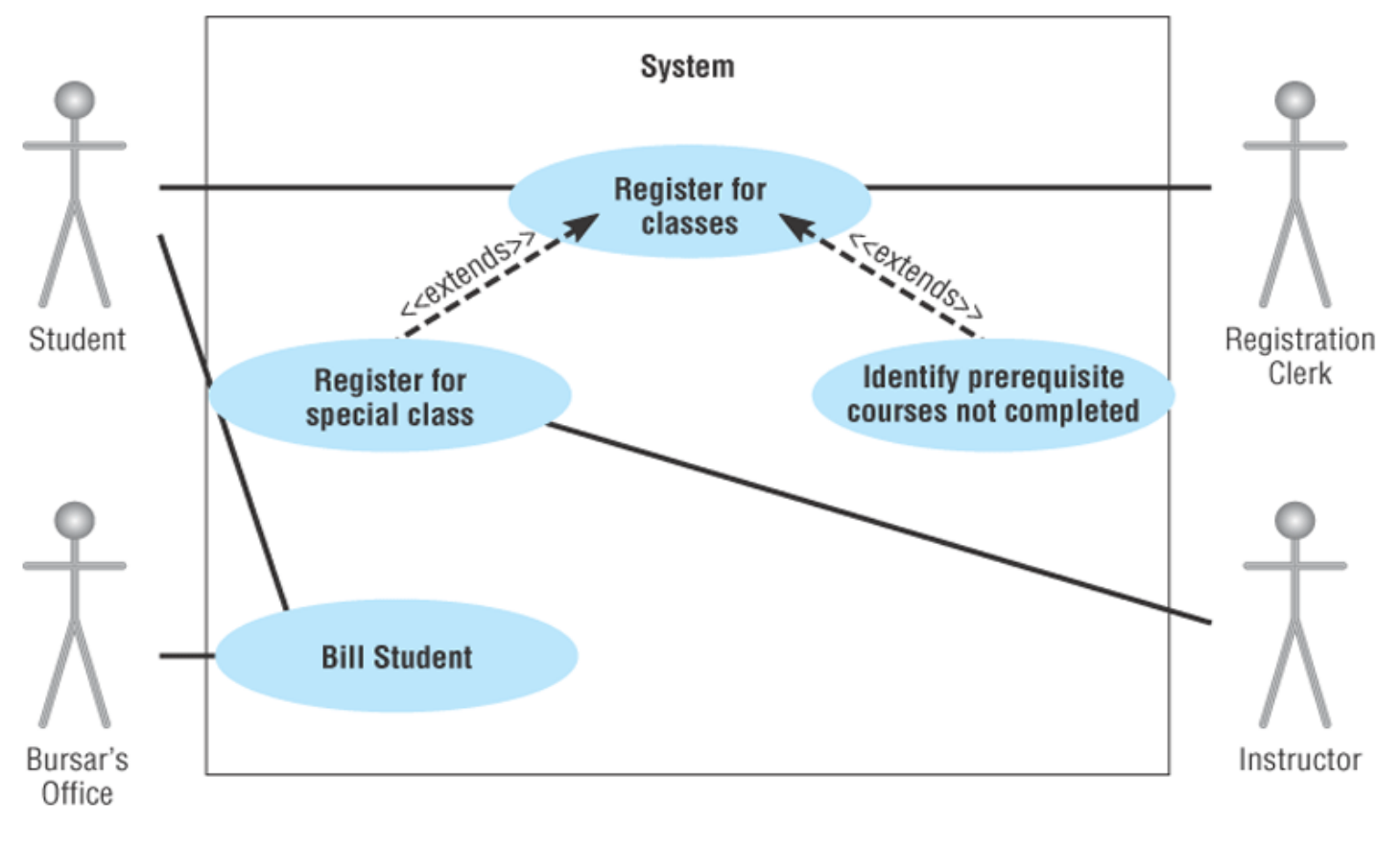
- Extend relationship –
- linking an ***optional*** use case to a standard use case.

- Example: *Register Course* (standard use case) may have
- *Register for Special Class* (extend use case).

- Standard use case can execute without the extend case.
- ☐ loose coupling

Extend Relation between Use Cases

- University Registration System



Careful: Arrows between Use Cases are NOT data flows!

Include Relation between Use Cases

- Include relationship –

  a standard case linked to an ***mandatory*** use case.

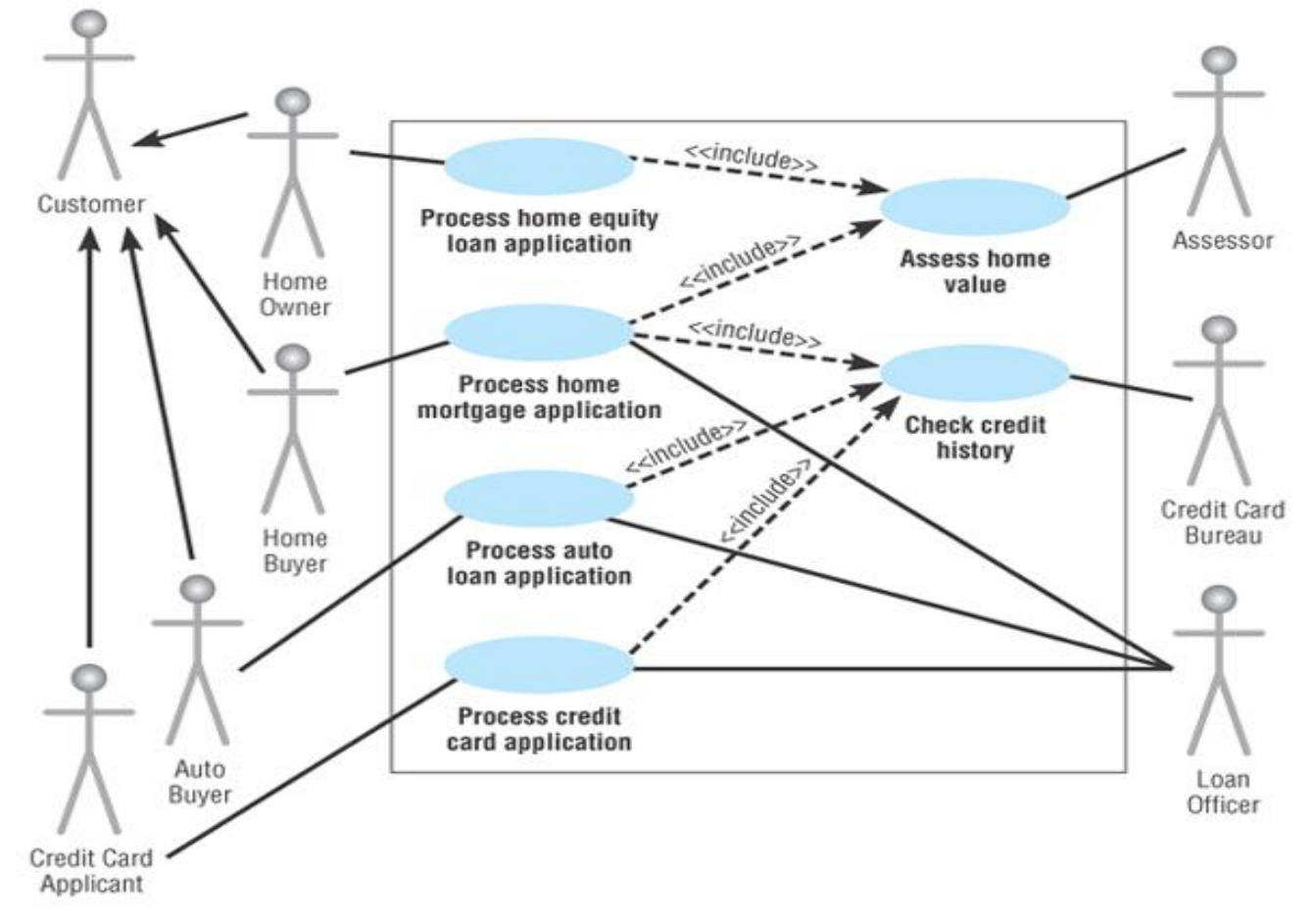
  Example: to *Authorize Car Loan* (standard use case), a clerk must run *Check Client's Credit History* (include use case).

  ☐ tight coupling

  - Standard use case can NOT execute without the include case.

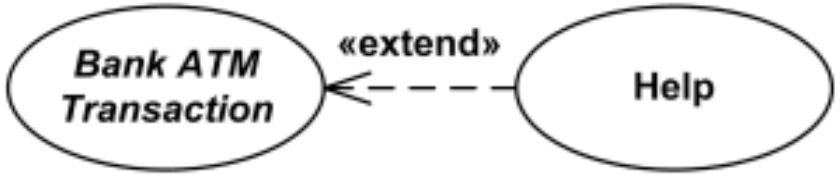
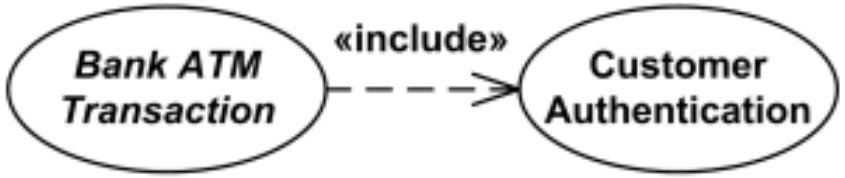
Extend Relation between Use Cases

- Credit Card Processing

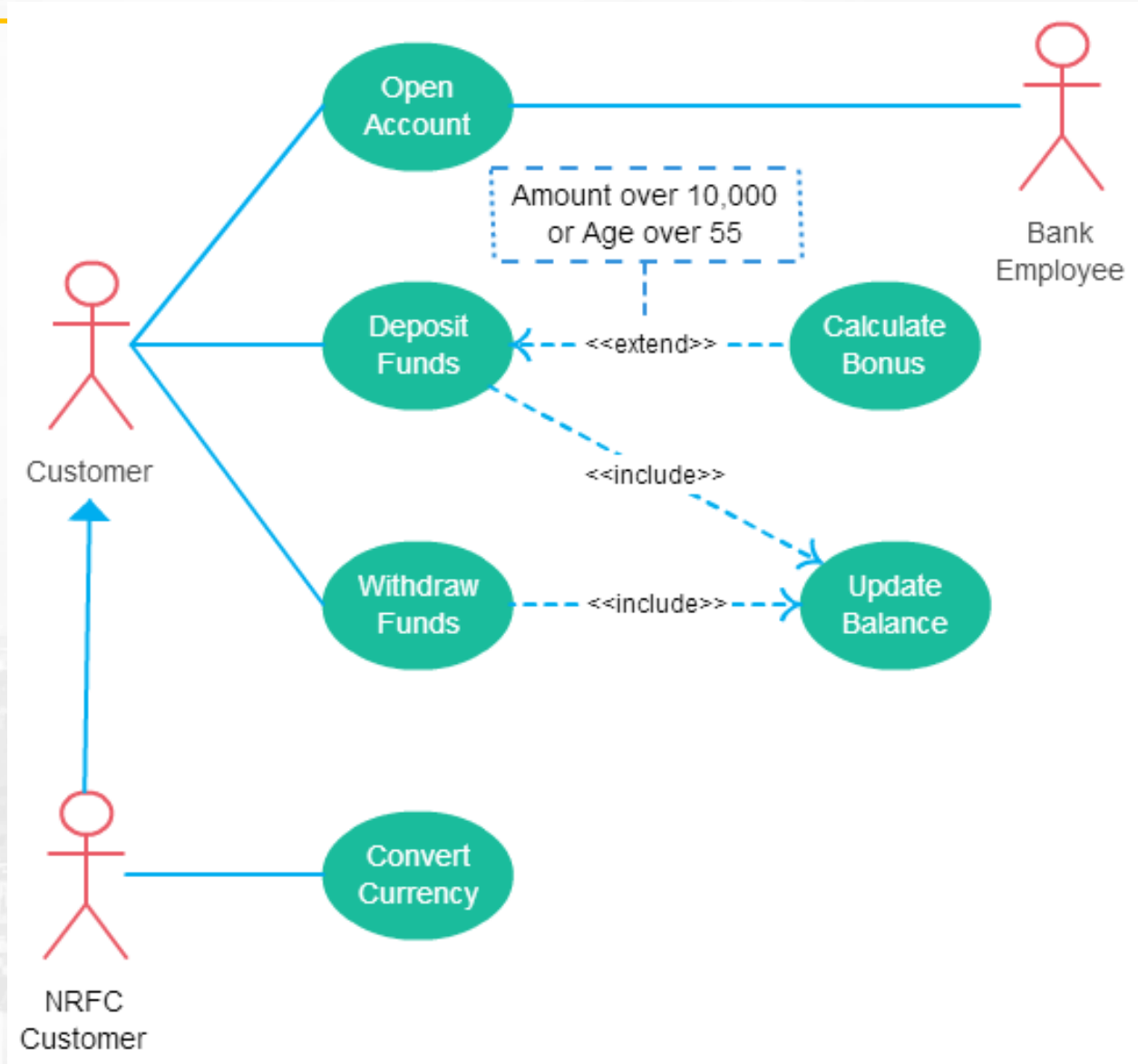


Careful: Arrows between Use Cases are NOT data flows!

Extend and Include

	Extend	Include
		
	Base use case is complete (concrete) by itself, defined independently.	Base use case is incomplete (**abstract use case**).
	Extending use case is optional, supplementary.	Included use case required, not optional.
	Has at least one explicit extension location.	No explicit inclusion location but is included at some location.
	Could have optional extension condition.	No explicit inclusion condition.

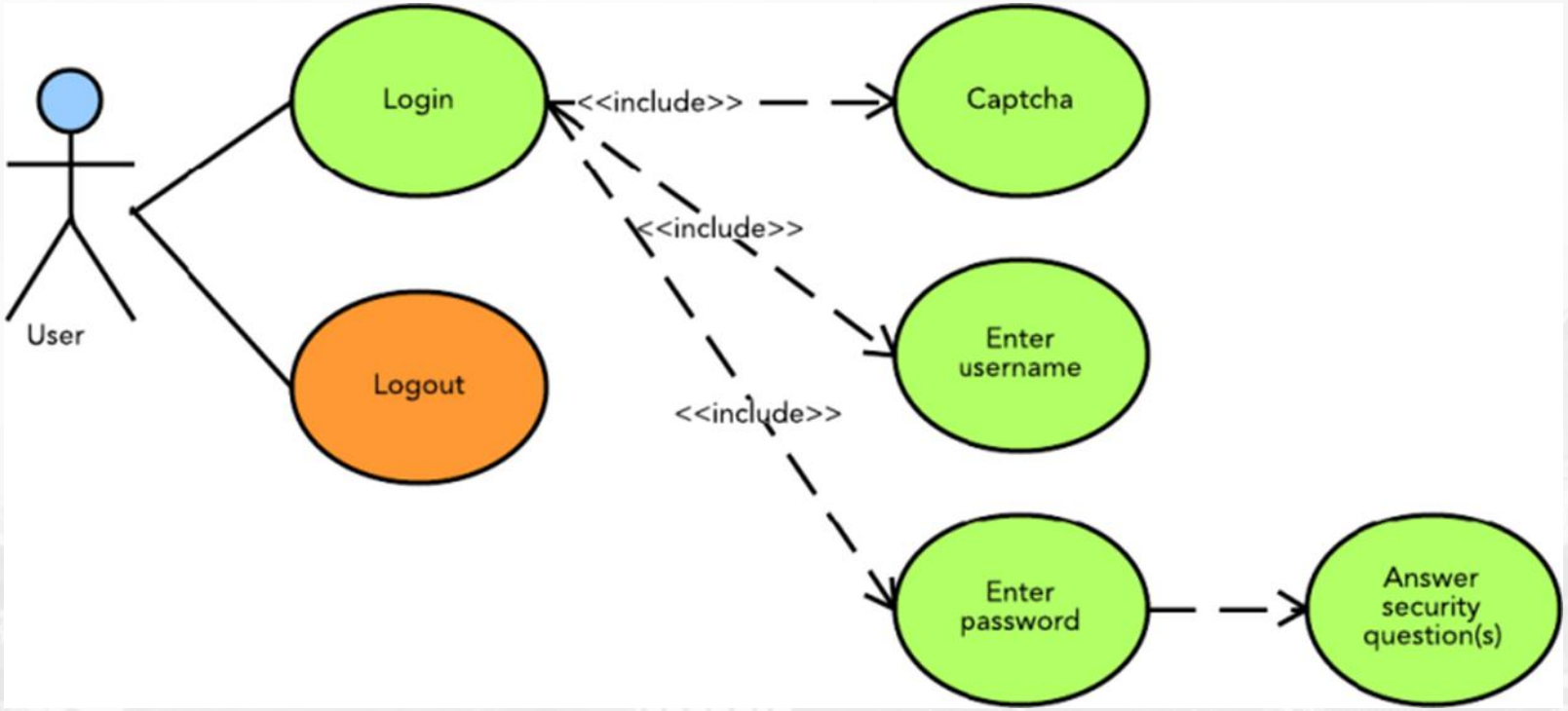
Another example



Use Case Example

- Login:
  - Captcha
  - Password
  - Answer Secret Questions
- Log out

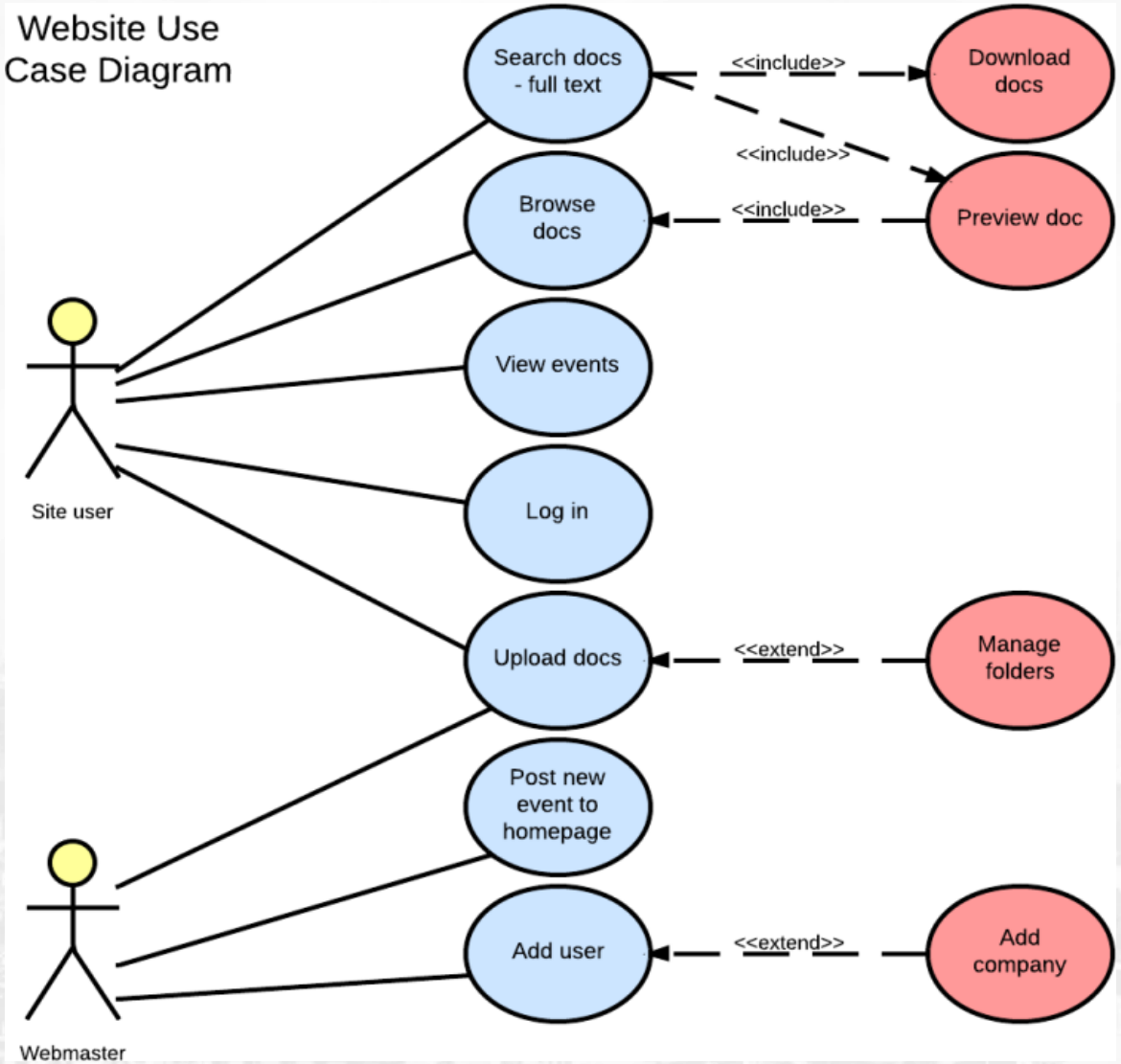
Example : Login



Use Case Diagram for Website Use Case

- Consider the User and Webmaster
- User can login, browse and search document.
  - Download is also possible
  - User can view event only if he is logged in
- Webmaster is responsible for adding user, uploading documents and posting events .
  - Uploaded documents have to be put in folders
  - When a User is added , the Company is also added

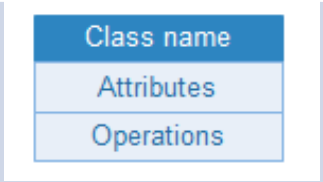
Use Case Diagram for Website Use Case



Class Diagram

- A class diagram depicts classes and their interrelationships
- Used for describing structure and behavior in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- Detailed class diagrams are useful for software developers

Class Diagram

S.No	Name	Description	Notation
1	Classes and interface	They are used to show the different objects in a system, their attributes, their operations and the relationships among them.	
2	Object	An object is an instance or occurrence of a class	
3	Aggregation	An aggregation describes a group of objects and how you interact with them.	
4	Composition	Composition represents whole-part relationships and is a form of aggregation.	
5	Dependency	Dependency relationship is a relationship in which one element, the client, uses or depends on another element, the supplier.	

Class Diagram

S.No	Name	Description	Notation
3	Generalization	Generalization is a relationship in which one model element (the child) is based on another model element (the parent).	
4	Association	Association is a relationship between two classifiers, such as classes or use cases, that describes the reasons for the relationship and the rules that govern the relationship.	
5	Multiplicity		**Multiplicity** Symbol — Meaning 1 — One and only one 0..1 — Zero or one M..N — From M to N (natural language) * — From zero to any positive integer 0..* — From zero to any positive integer 1..* — From one to any positive integer

Drawing a Class Diagram ?

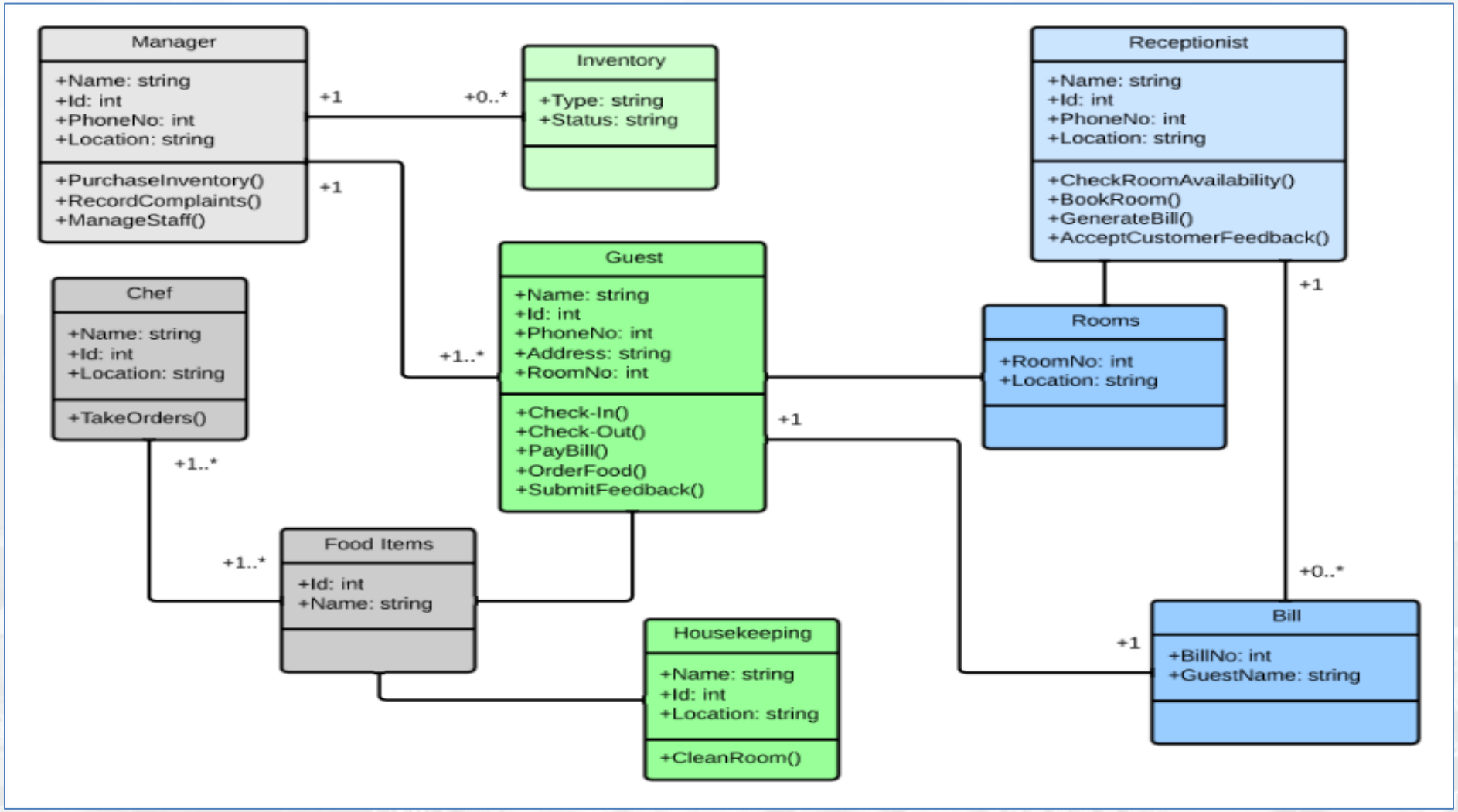
- Identify and model classes—Which classes do we need?
- Identify and model associations—How are the classes connected?
- Define attributes—What do we want to know about the objects?

A Single Class

Rectangle
- width: int - height: int / area: double
+ Rectangle(w: int, h: int) + distance(r: Rectangle): double

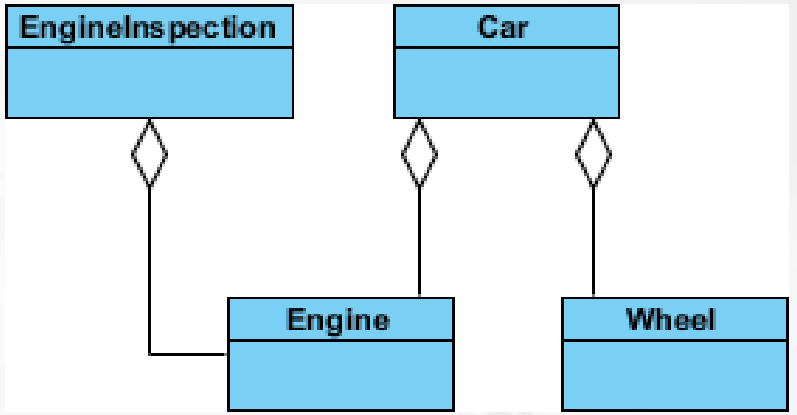
Student
- name: String - id: int - totalStudents: int
getID(): int ~ getEmail(): String

Class Diagram : Hotel Management System

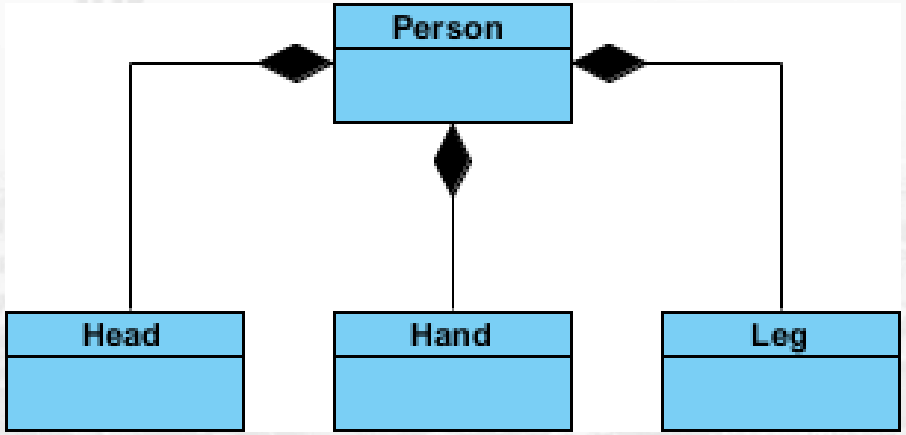


Types of Relations in Class Diagram

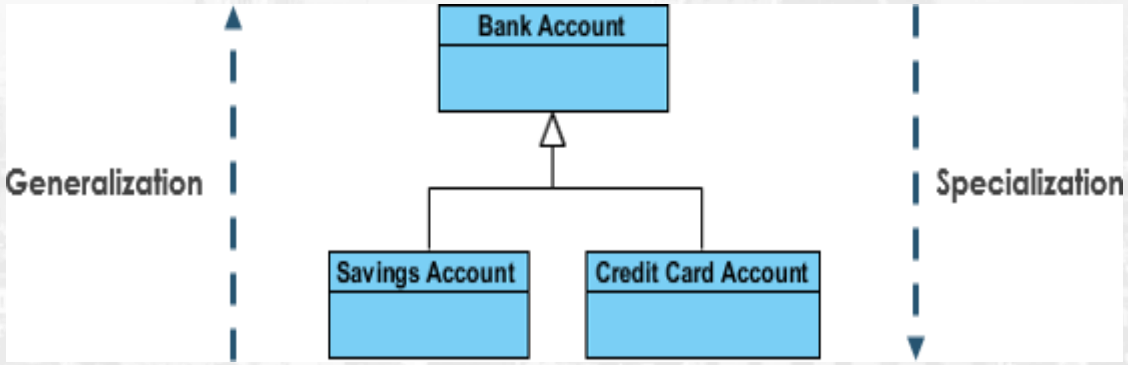
Association



Aggregation (exist alone)

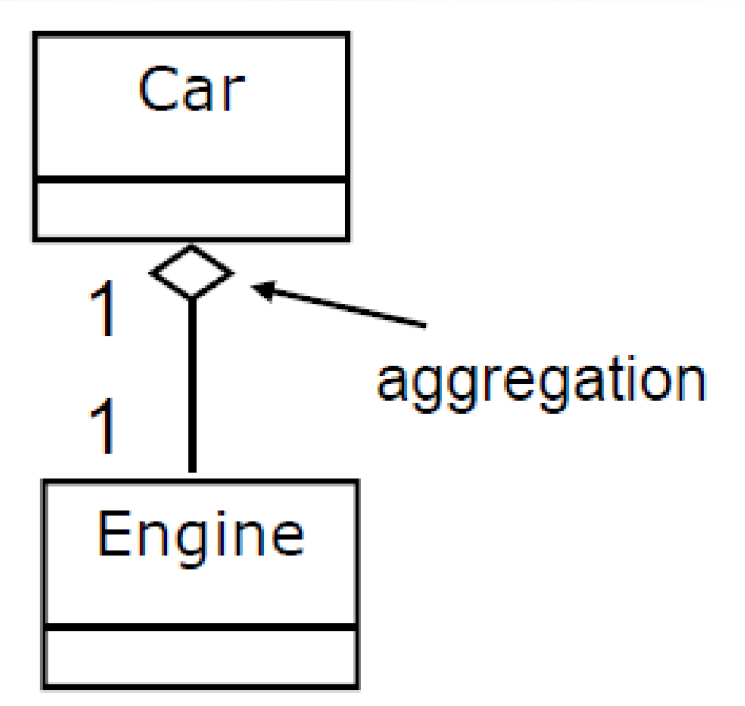


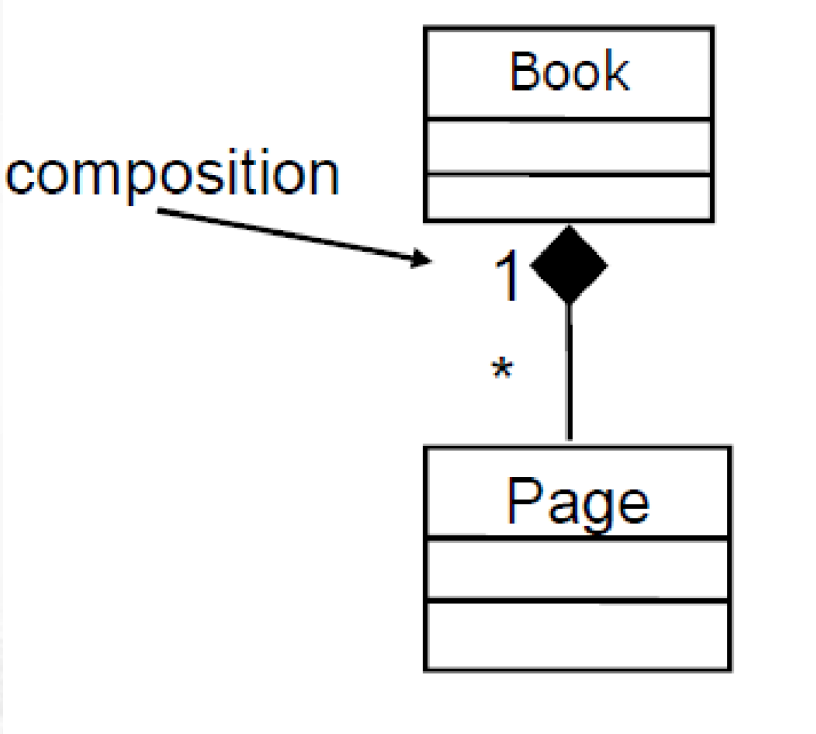
Composition



Generalization

Another Example





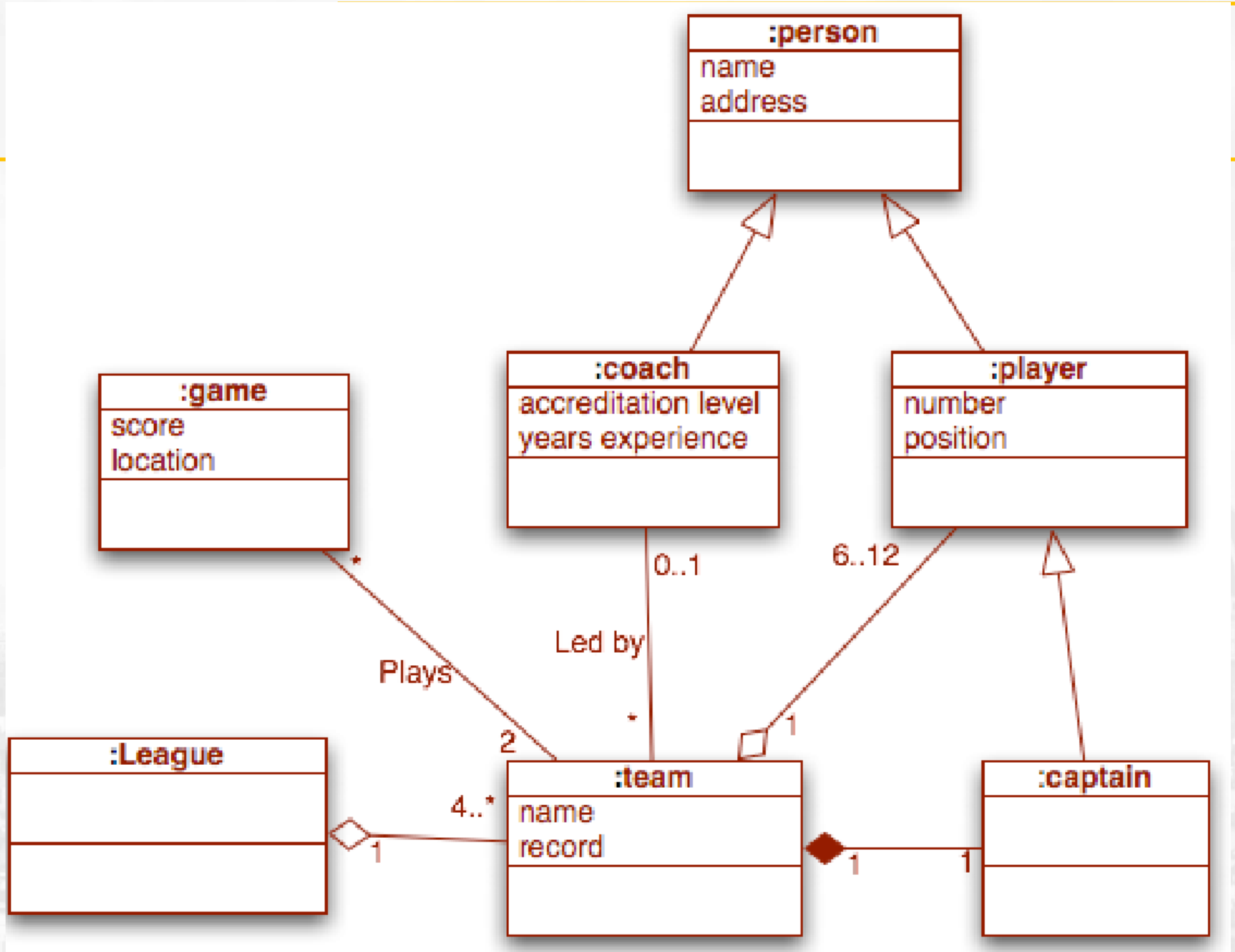
Example

- A stock exchange lists many companies.
- Each company is uniquely identified by a ticker symbol A company can be listed on more than one stock exchange, using the same ticker symbol.

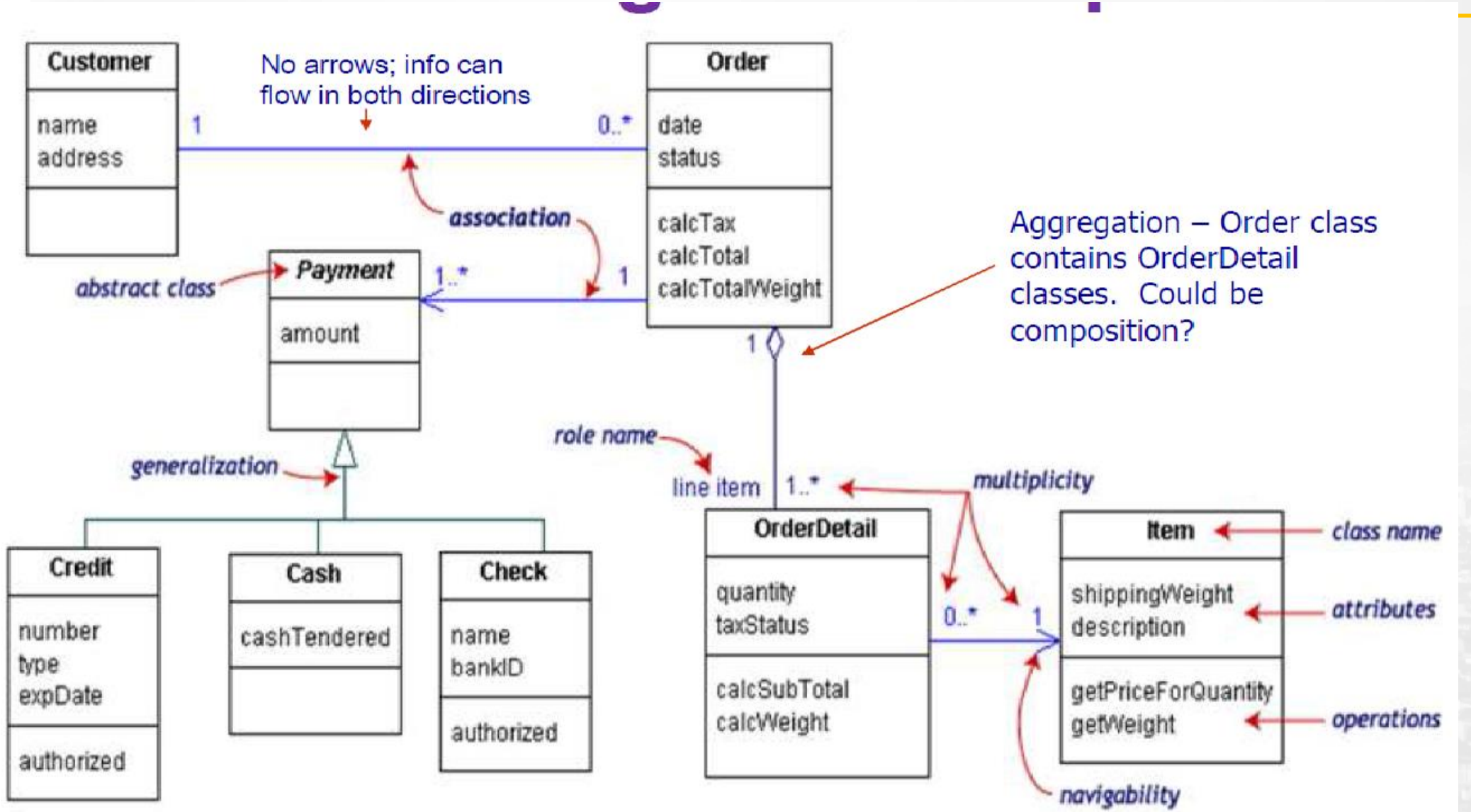

StockExchange
*
*
Company
Lists
tickerSymbol

Example 2

- Draw a UML Class Diagram representing the following elements from the problem domain for a hockey league.
- A hockey league is made up of at least four hockey teams.
- Each hockey team is composed of six to twelve players, and one player captains the team.
- A team has a name and a record. Players have a number and a position. Hockey teams play games against each other. Each game has a score and a location. Teams are sometimes lead by a coach.
- A coach has a level of accreditation and a number of years of experience, and can coach multiple teams. Coaches and players are people, and people have names and addresses.


:person
name
address
:coach
accreditation level
years experience
:player
number
position
:game
score
location
:League
:team
name
record
:captain
Plays
Led by
0..1
6..12
*
2
*
1
4..*
1
1
1

Example : Order Details



Benefits of class diagrams

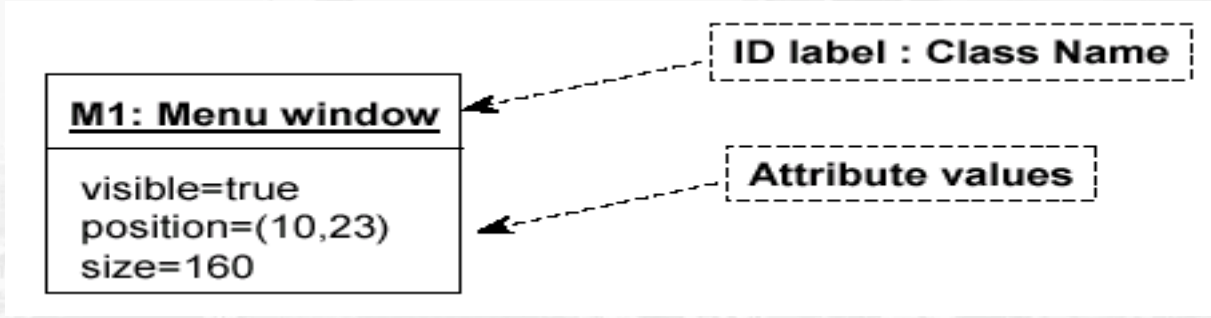
- To understand the general overview of plan of an application.
- Illustrate data models for information systems, no matter how simple or complex.
- Visually express any specific needs of a system
- Create detailed charts that highlight any specific code needed to be programmed and implemented to the described structure.
- Provide an implementation-independent description of types used in a system that are later passed between its components.

Object Diagram

- Derived from class diagram
- Shows a set of objects & their relationship
- Represents a static view of the system
- An object diagram is a diagram that shows a set of objects and their relationships at a **point in time.**
- Object diagrams help you capture the logical view of your model

Object Diagrams

- An object is shown with a rectangle and the title is underlined
- Format is
  - Instance name : Class name
  - Attributes and Values

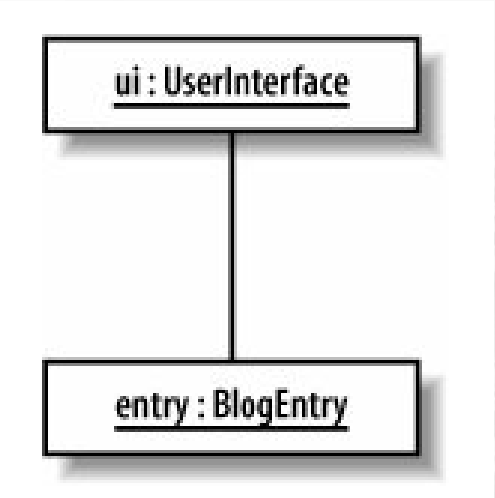


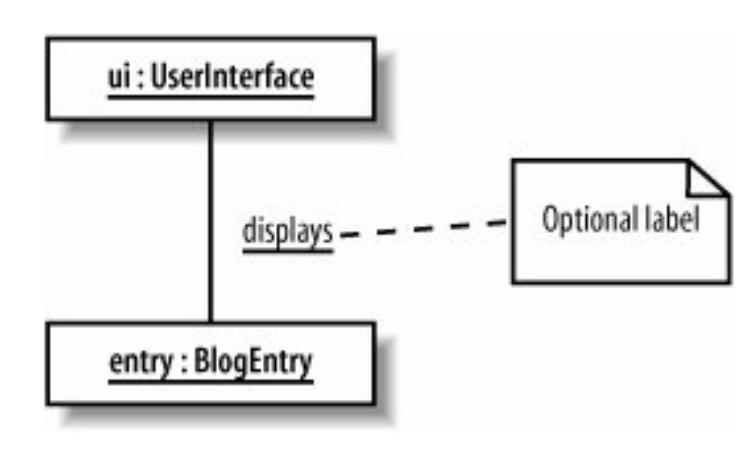
instantiated object

entry object is an instance of the BlogEntry class

Links

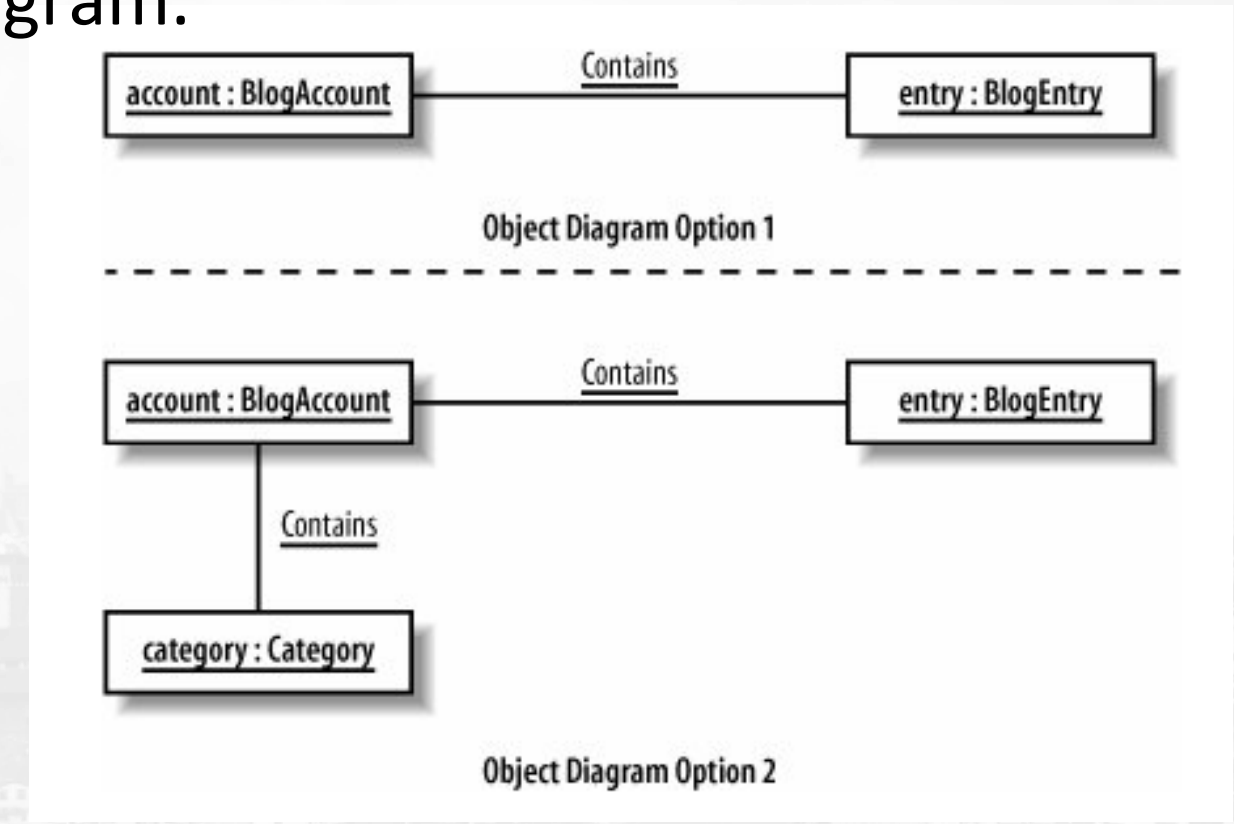
- To show how objects work together, links shows that two objects can communicate with each other
- There must be corresponding association between the classes in the class diagram
- Can add label that indicates the purpose of the link,



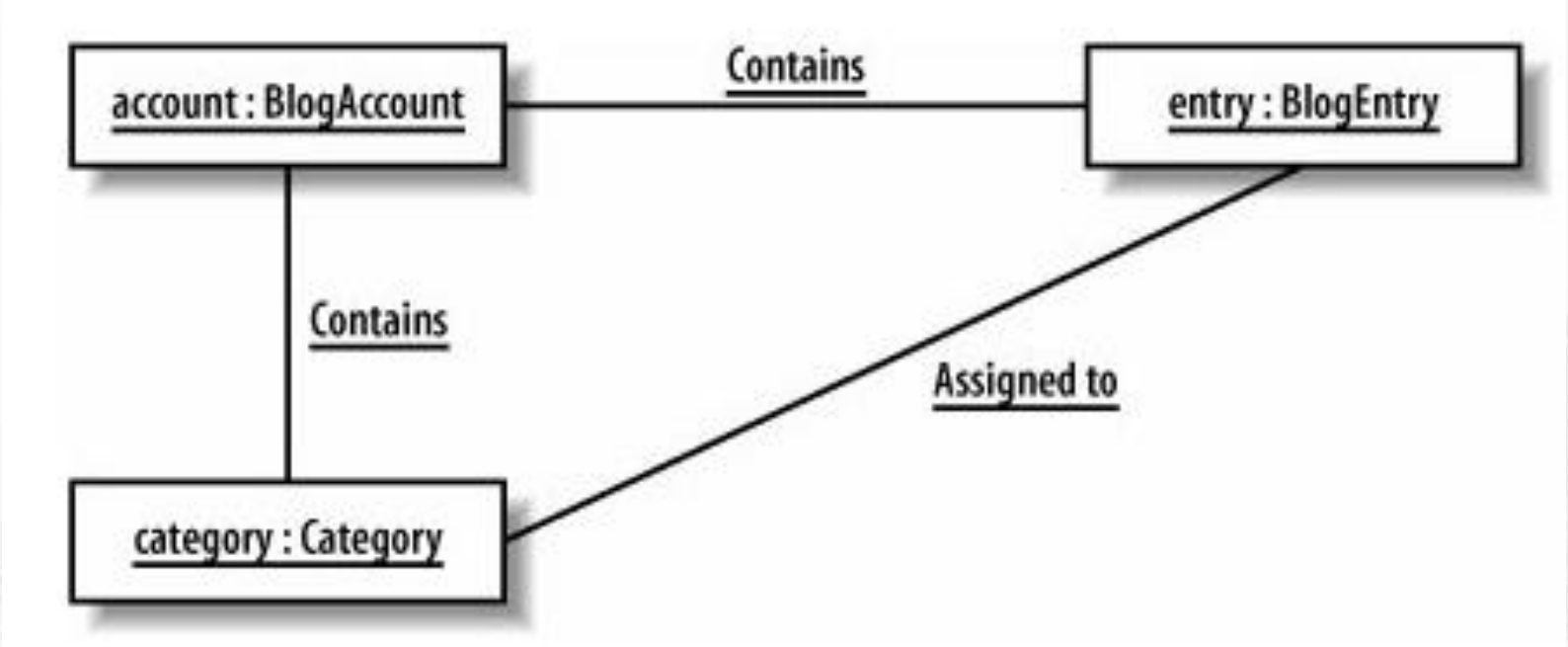


Links and Constraints

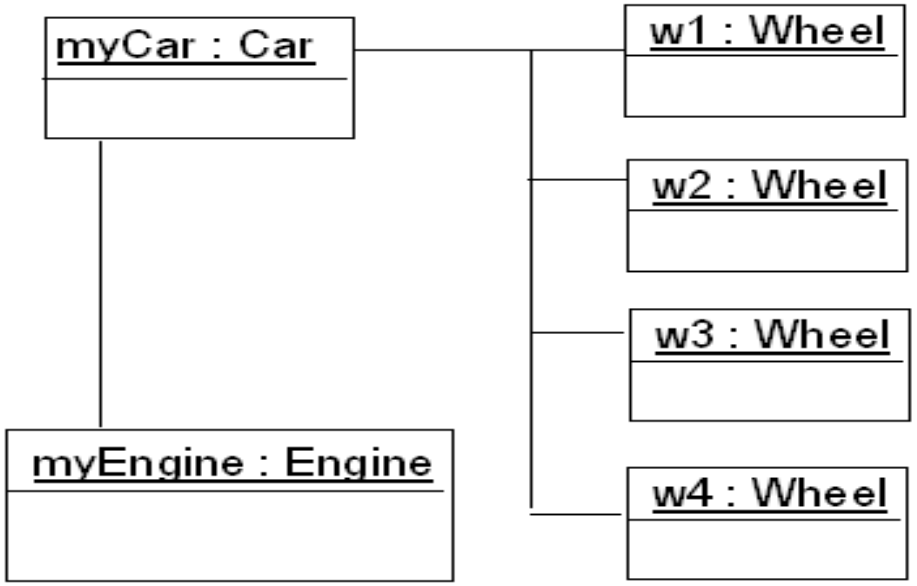
- Links must keep to the rules (constraints) given in class diagram.

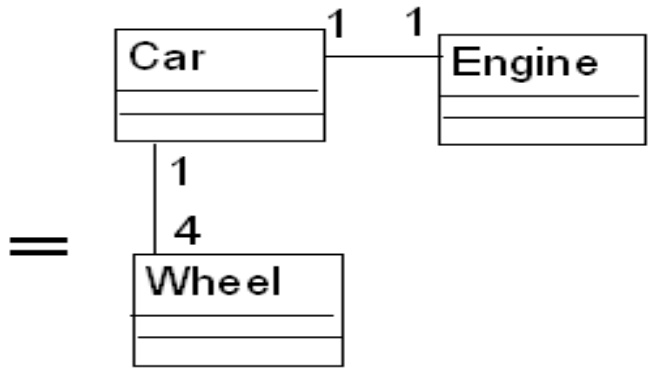


Both diagrams are valid.


account : BlogAccount
Contains
entry : BlogEntry
Contains
Assigned to
category : Category

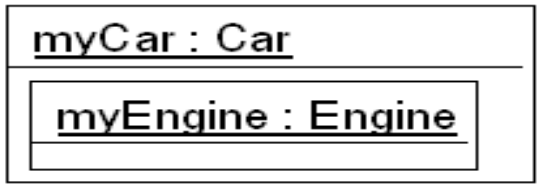
An Object Diagram example





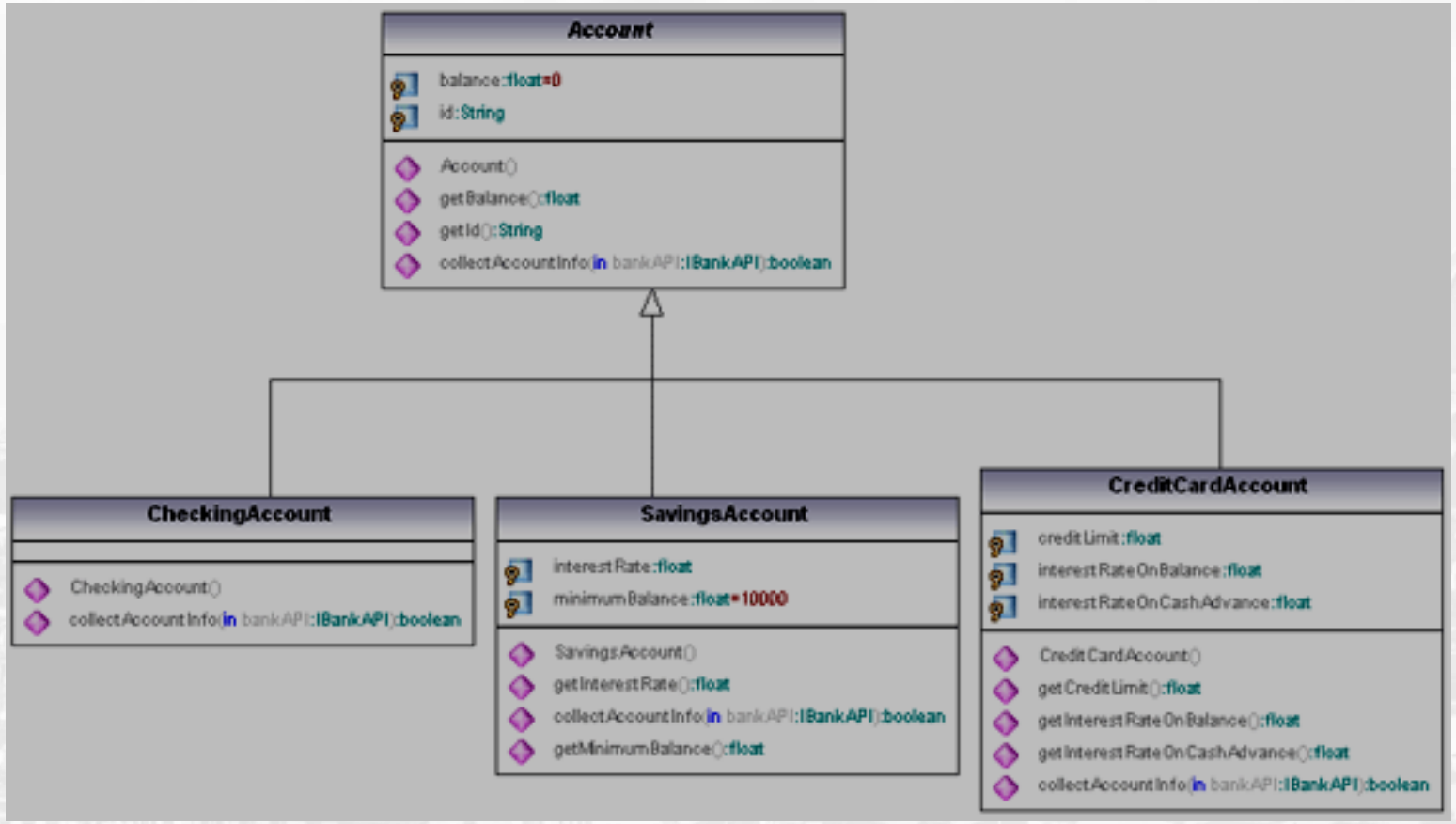
Composition with :

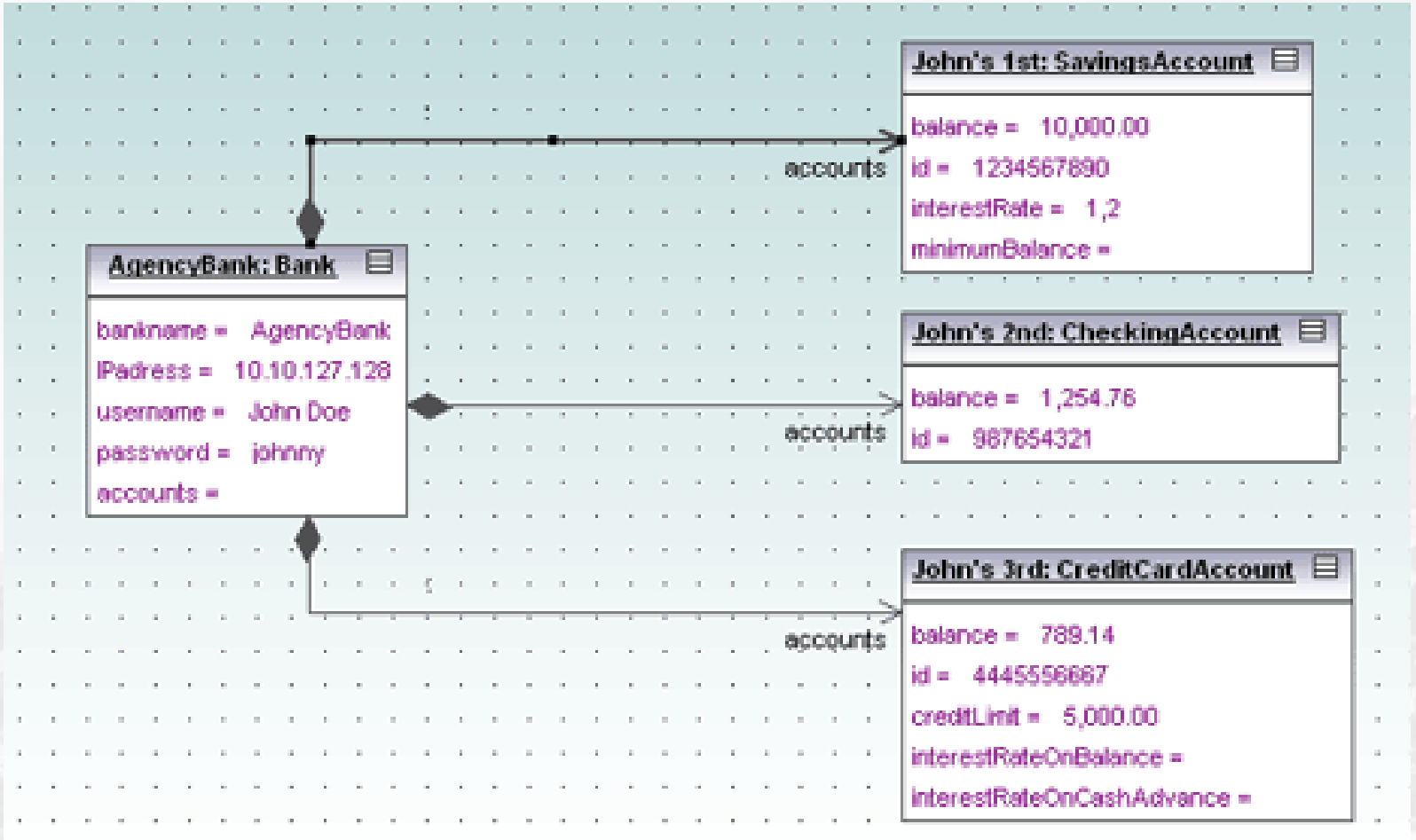
Or containment :



Eg.1

- From the given class diagram draw an object diagram for John's accounts

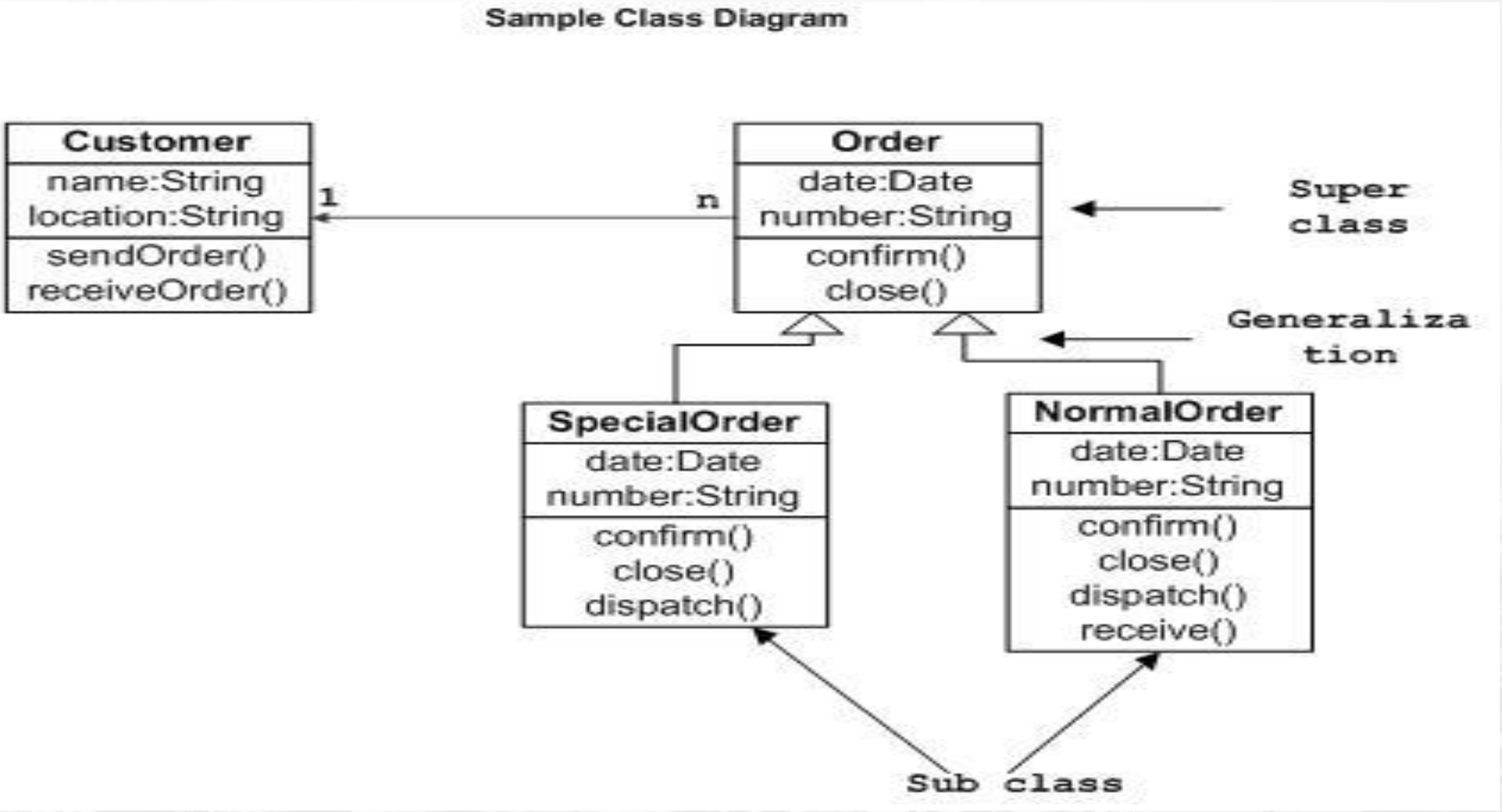



John's 1st: SavingsAccount
balance = 10,000.00
id = 1234567890
interestRate = 1.2
minimumBalance =
accounts
AgencyBank: Bank
bankname = AgencyBank
IPadress = 10.10.127.128
username = John Doe
password = johnny
accounts =
John's 2nd: CheckingAccount
balance = 1,254.76
id = 987654321
accounts
John's 3rd: CreditCardAccount
balance = 789.14
id = 4445556667
creditLimit = 5,000.00
interestRateOnBalance =
interestRateOnCashAdvance =
accounts

Eg. 2

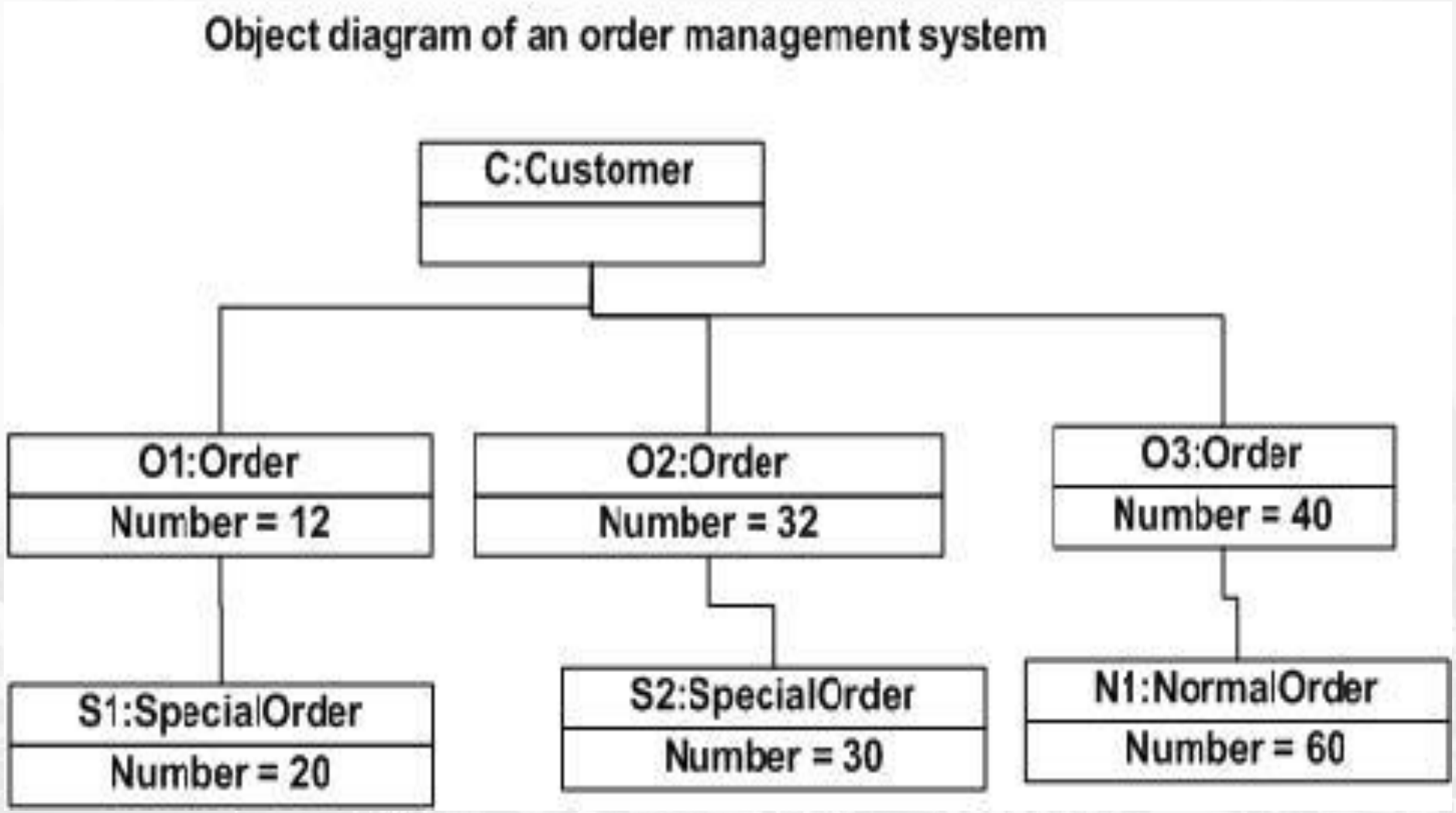
- Draw Object diagram for the *Order management system* . It has the following objects
  - Customer
  - Order
  - SpecialOrder
  - NormalOrder
- Customer object (C) is associated with three order objects (O1, O2 and O3). These order objects are associated with special order and normal order objects (S1, S2 and N1). The customer is having the following three orders with different numbers (12, 32 and 40) for the particular time considered.

Sample Class Diagram

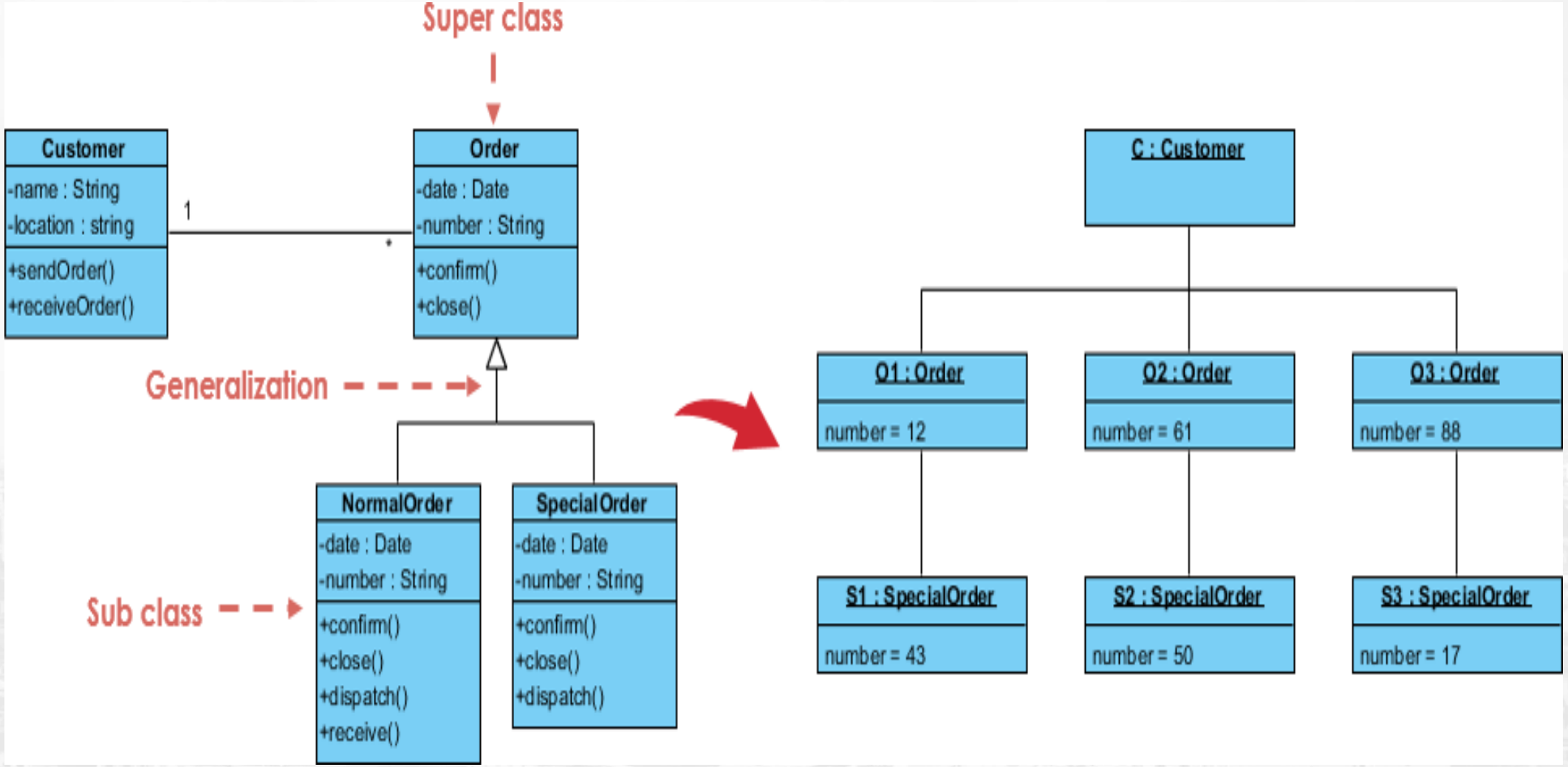


MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

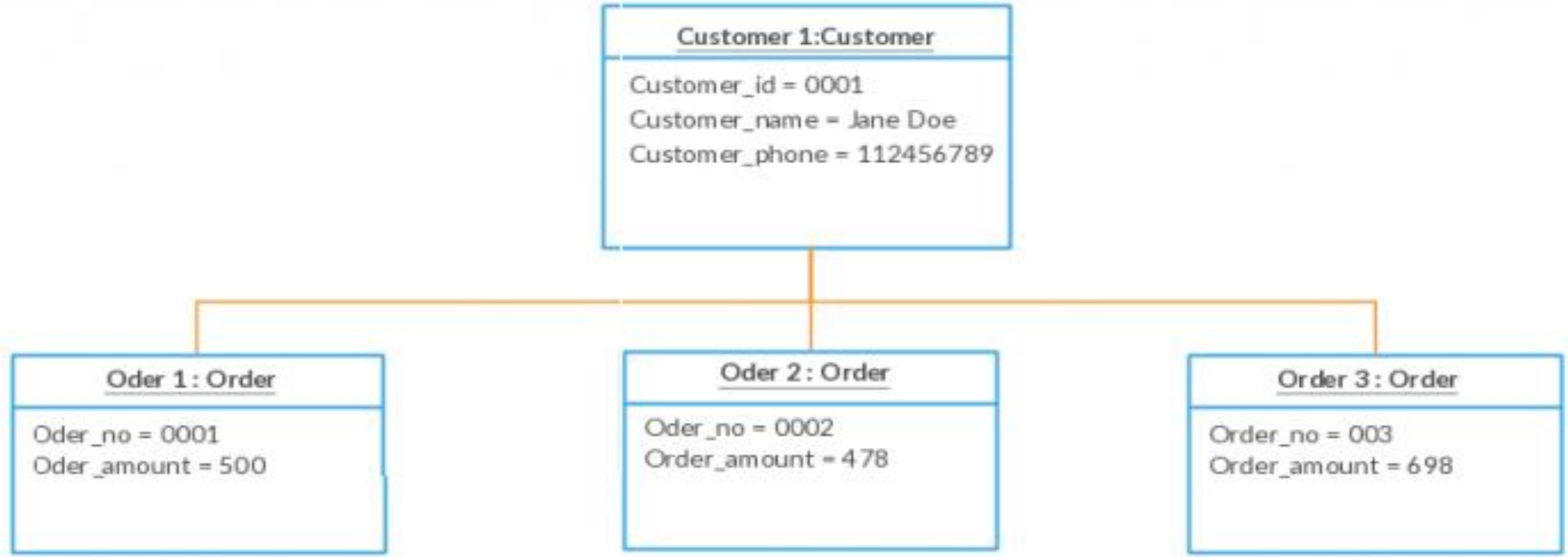
Object diagram of an order management system



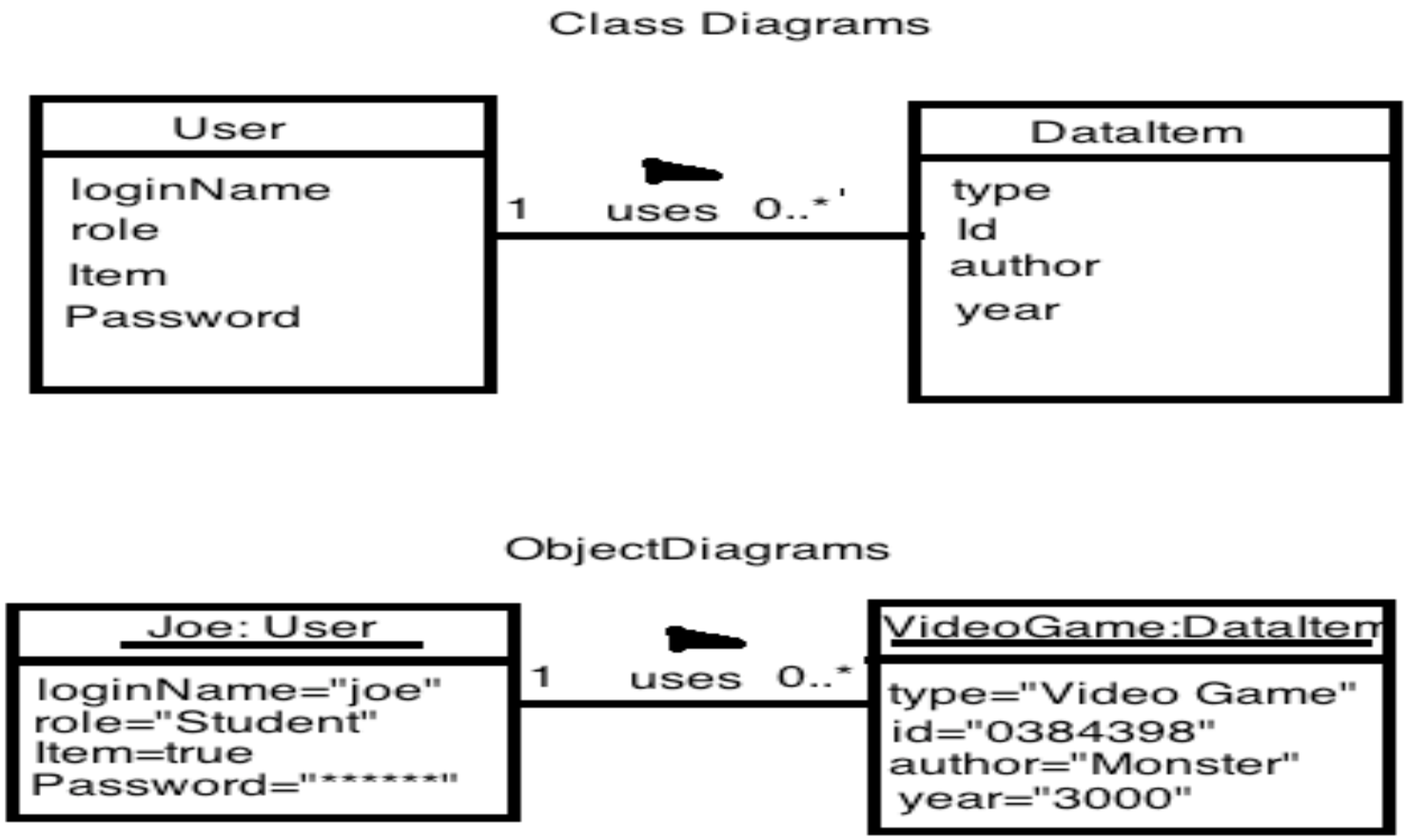
Class to Object Diagram Example - Order System



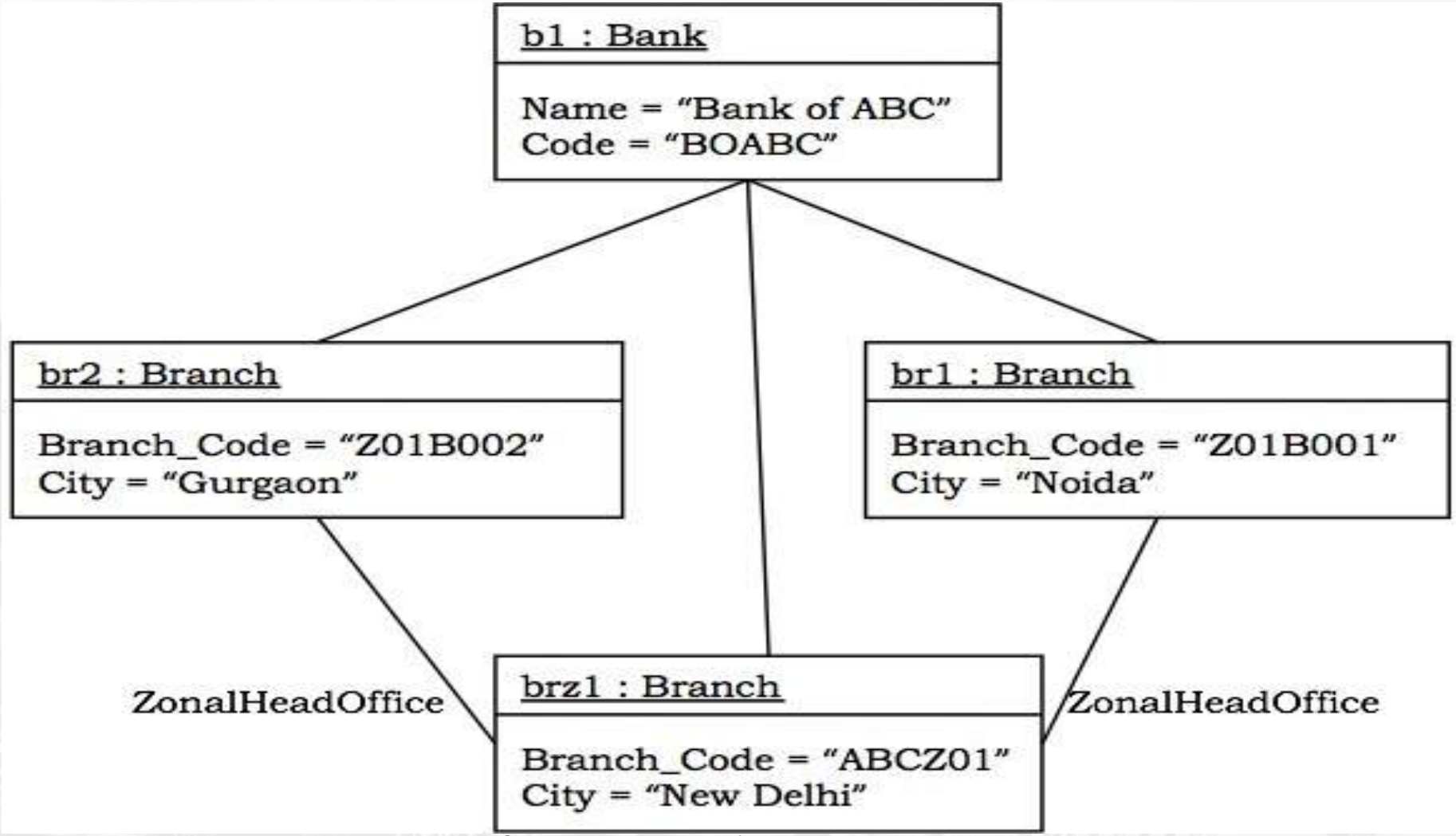
Example contd..



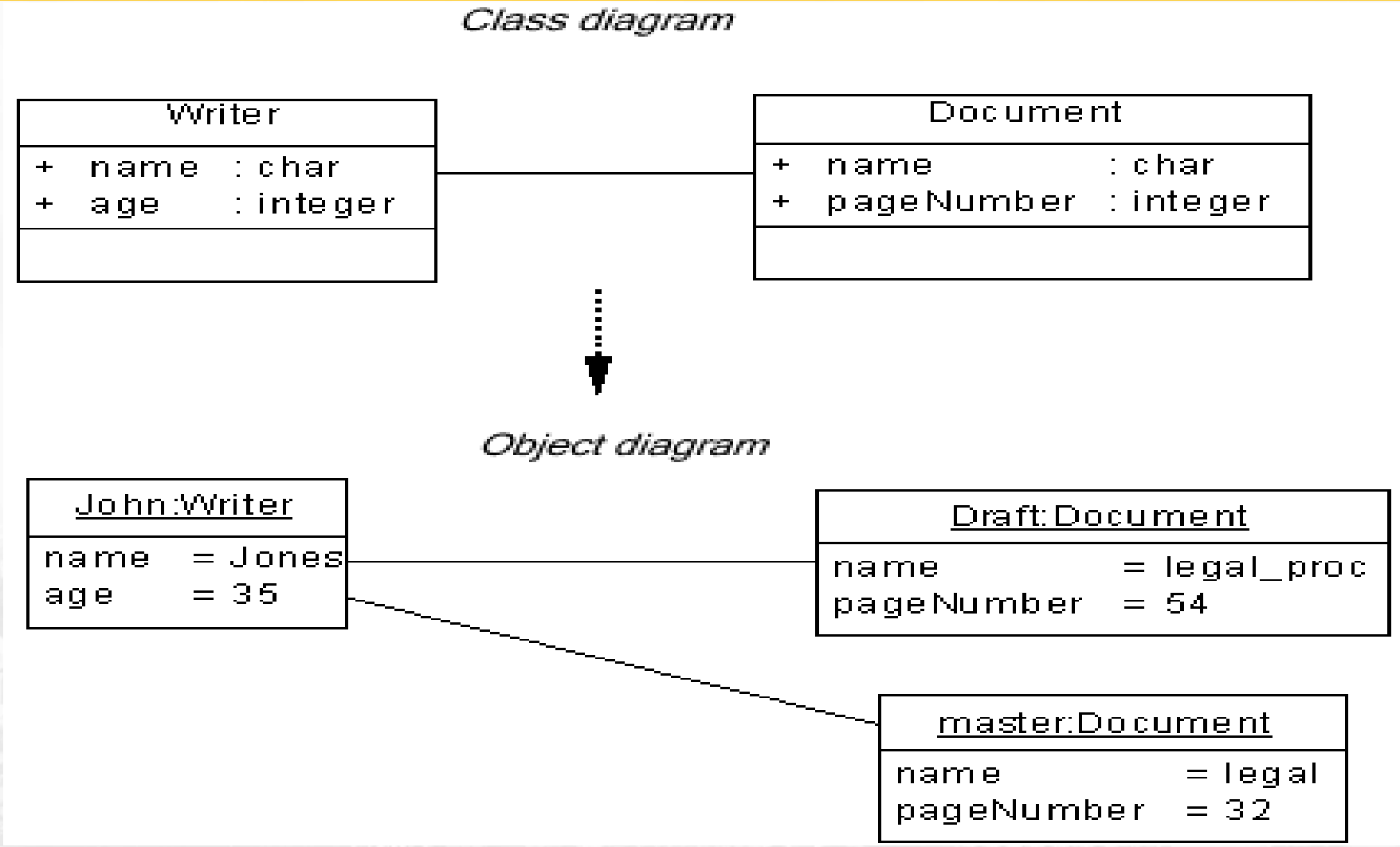
Example contd..



Example (contd..)



Example contd..

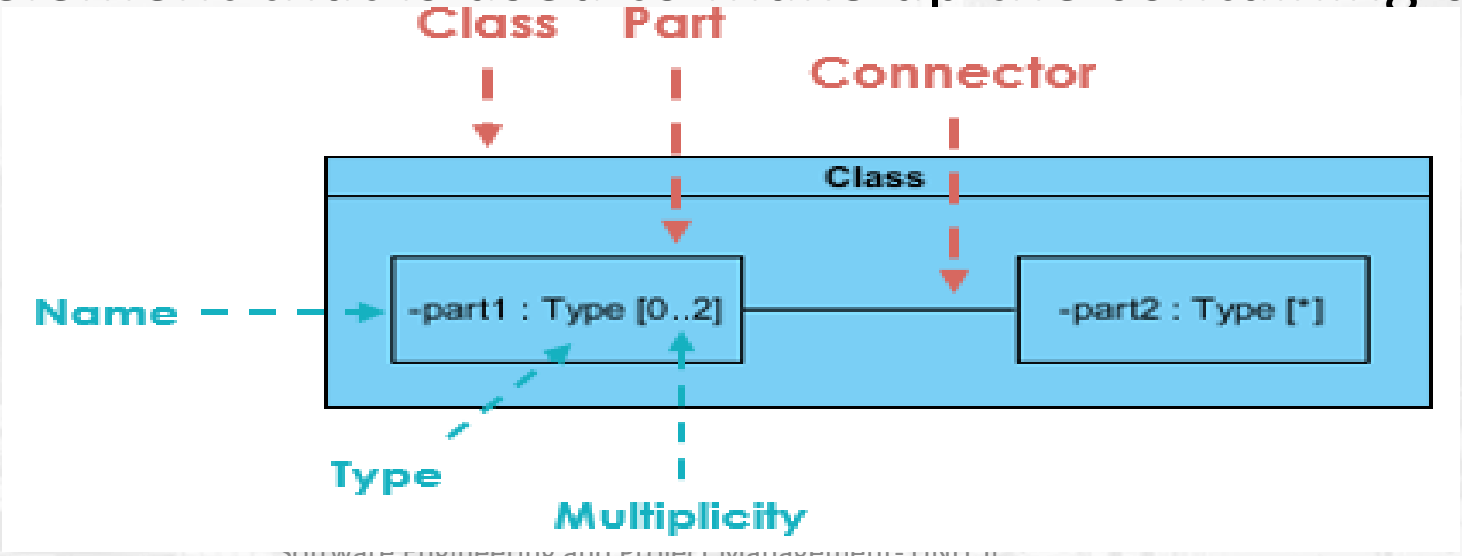


Composite Structure Diagrams

- A composite structure diagram is a UML structural diagram that contains classes, interfaces, packages, and their relationships, and that provides a logical view of all, or part of a software system.
- It shows the internal structure (including parts and connectors) of a structured classifier or collaboration.
- A composite structure diagram performs a similar role to a class diagram, but allows you to go into further detail in describing the internal structure of multiple classes and showing the interactions between them.

Representation

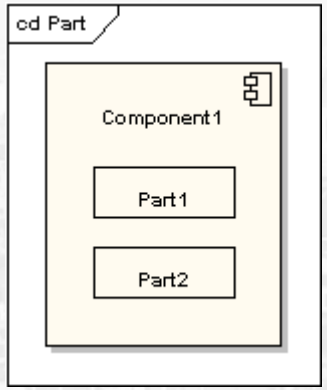
- Composite Structure Diagrams show the internal parts of a class.
- Parts are named: partName:partType[multiplicity]
- Aggregated classes are parts of a class but parts are not necessarily classes
- A part is any element that is used to make up the containing class.

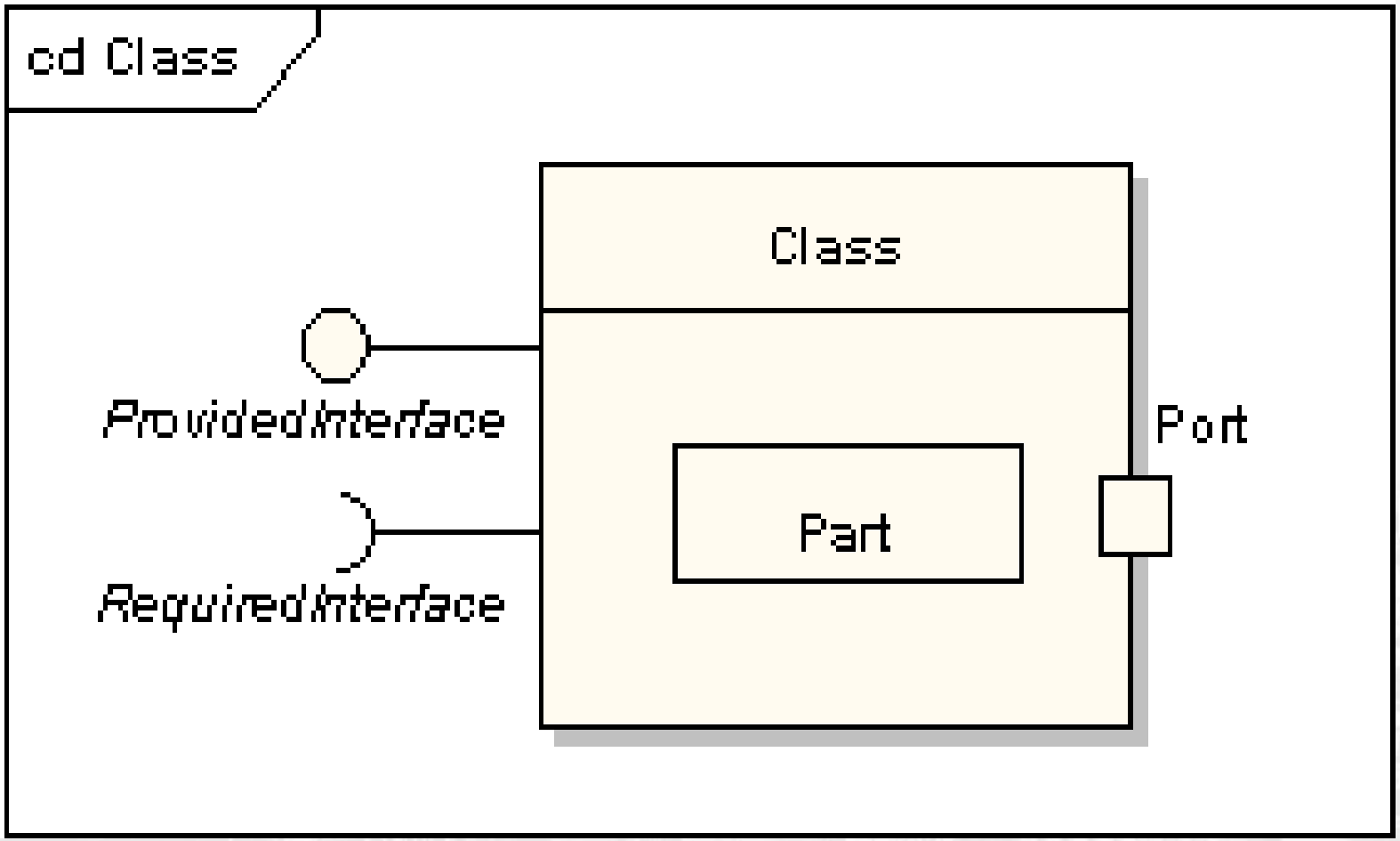


Part

- A part is an element that represents a set of one or more instances which are owned by a containing classifier instance.
- If a diagram instance owned a set of graphical elements, then the graphical elements could be represented as parts.

- A part is shown as an unadorned rectangle contained within the body of a class or component element.

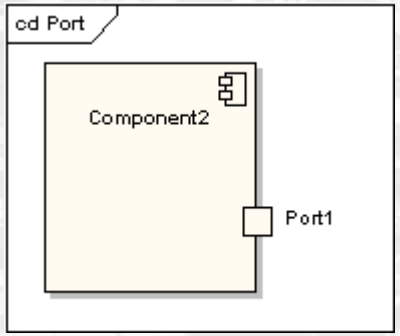



cd Class
Class
ProvidedInterface
RequiredInterface
Part
Port

Port

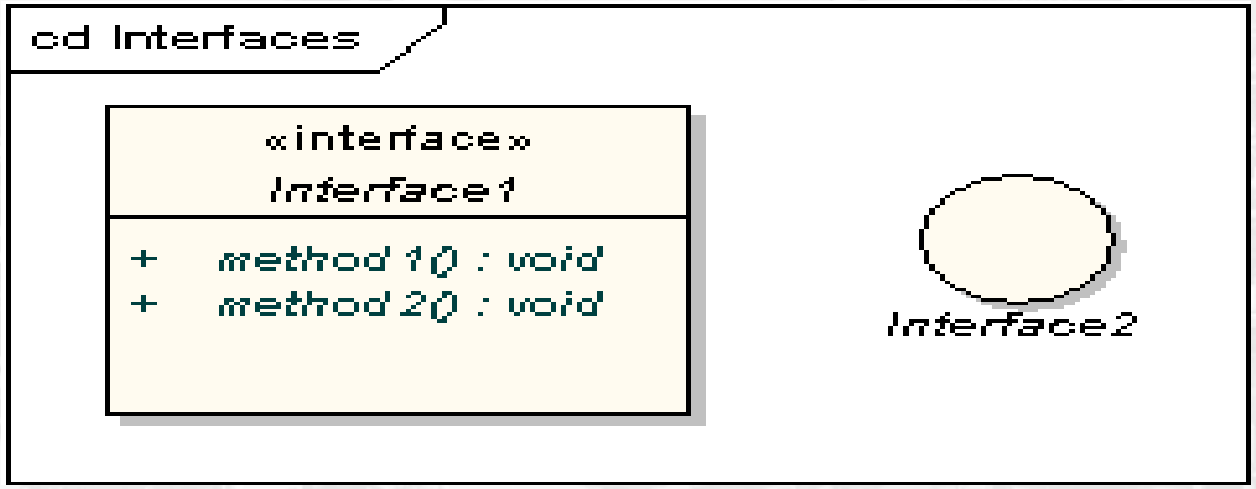
- A port is a typed element that represents an externally visible part of a containing classifier instance.
- Ports define the interaction between a classifier and its environment.
- A port can appear on the boundary of a contained part, a class or a composite structure.
- A port may specify the services a classifier provides as well as the services that it requires of its environment.

- A port is shown as a named rectangle on the boundary edge of its owning classifier.



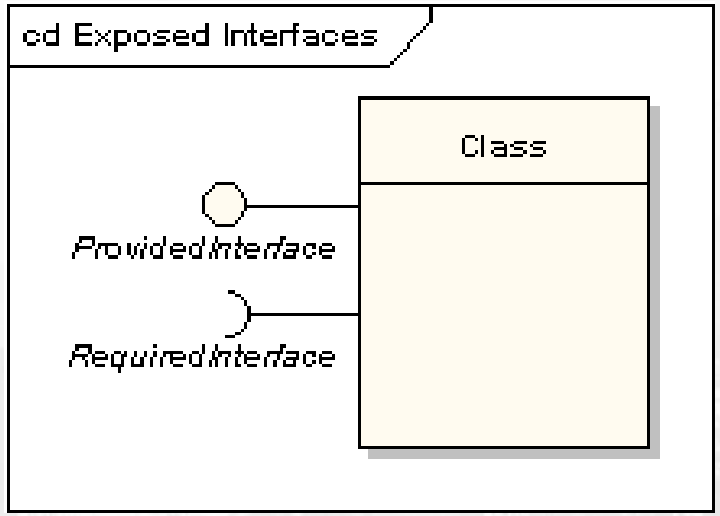
Interfaces

- All interface operations are public and abstract, and do not provide any default implementation.
- All interface attributes must be constants.
- An interface, when standing alone in a diagram, is either shown as a class element rectangle with the «interface» keyword and with its name italicized to denote it is abstract, or it is shown as a circle.



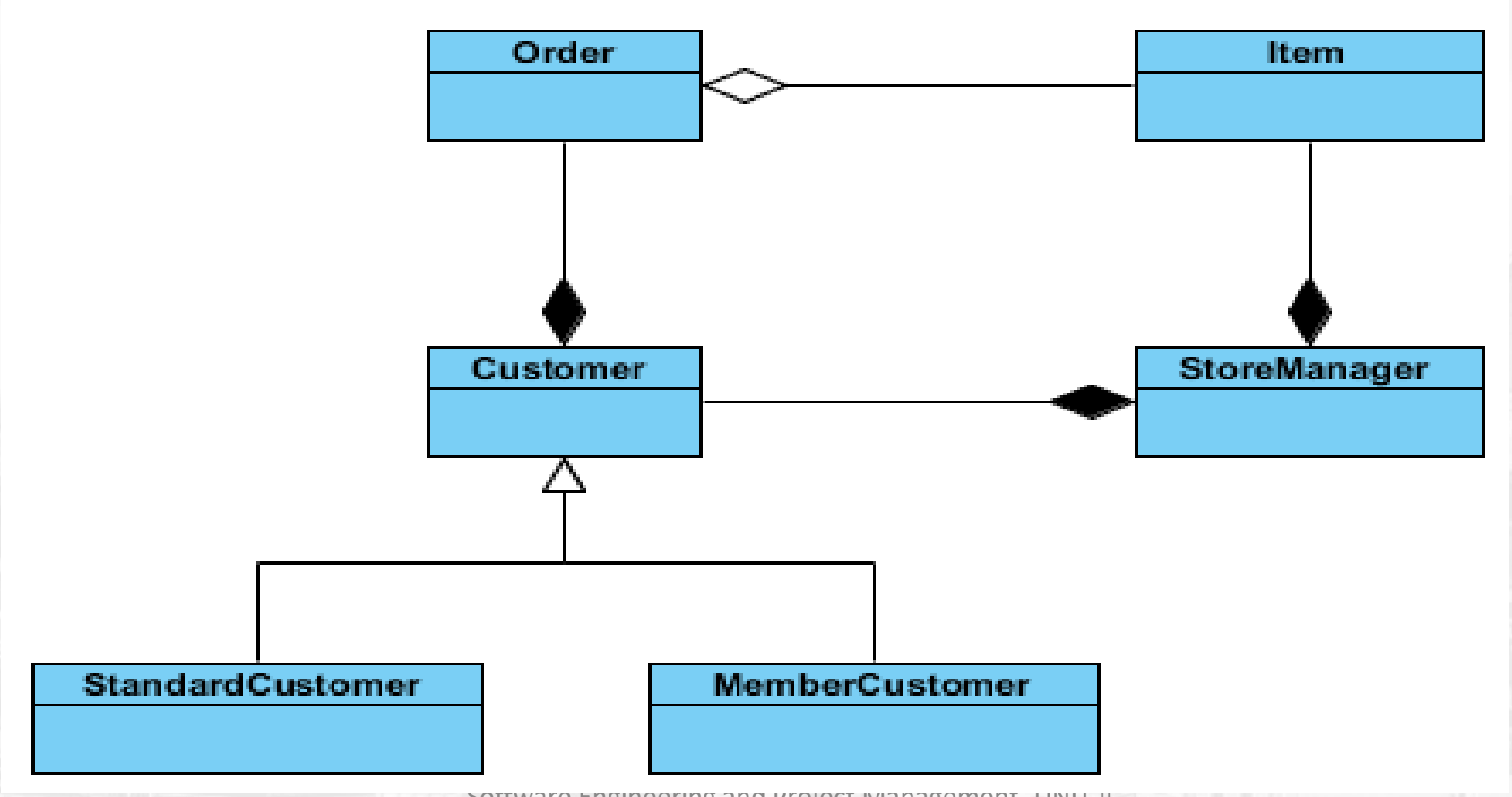
- An interface can be defined as either provided or required.

- A provided interface is shown as a "ball on a stick" attached to the edge of a classifier element.
- A required interface is shown as a "cup on a stick" attached to the edge of a classifier element.



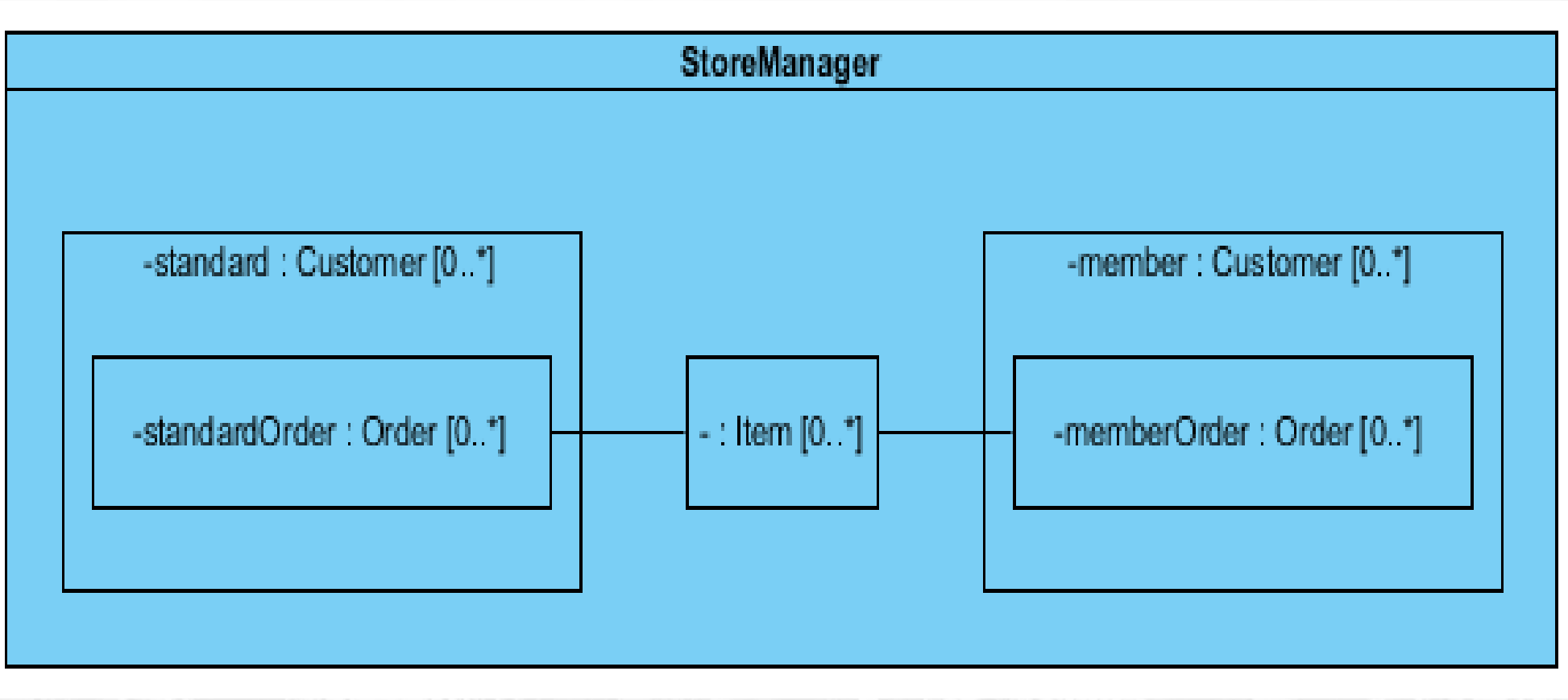
Example

- Online Store – Class Diagram

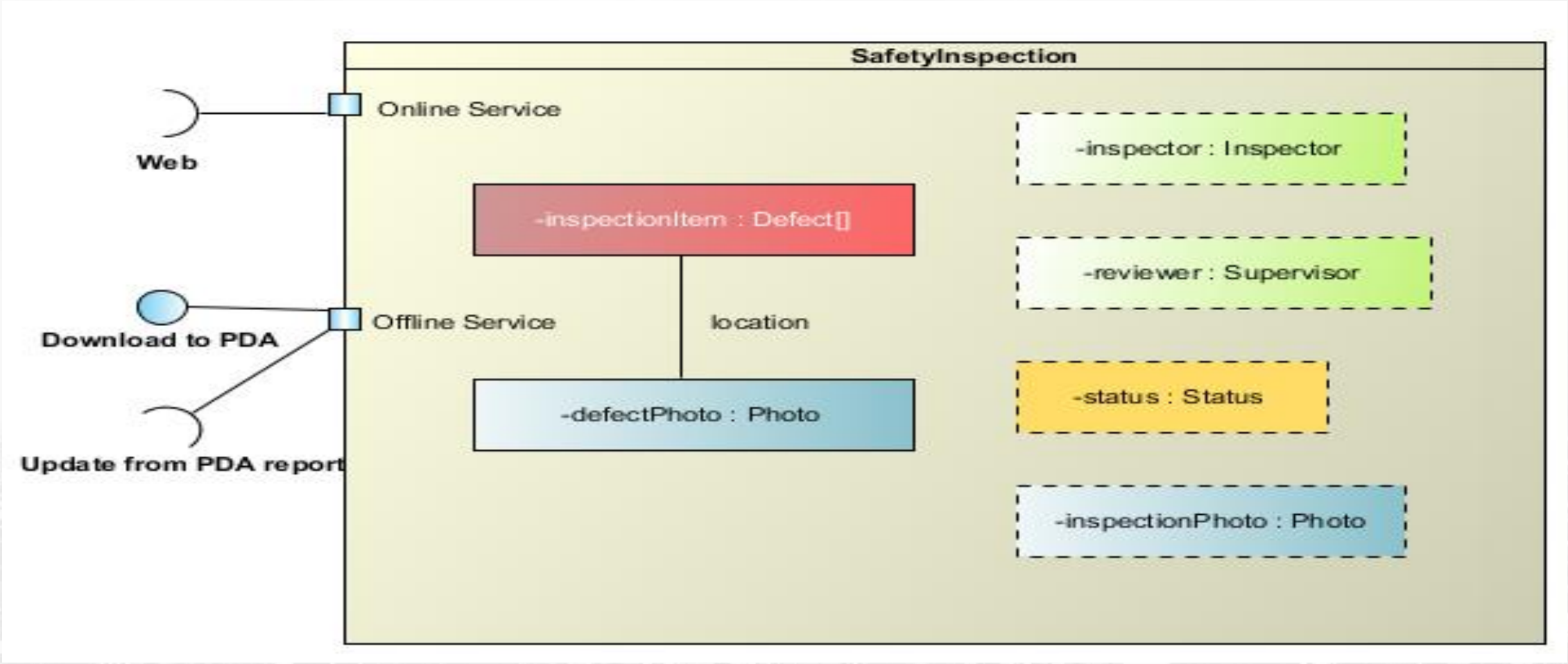


Example (contd..)

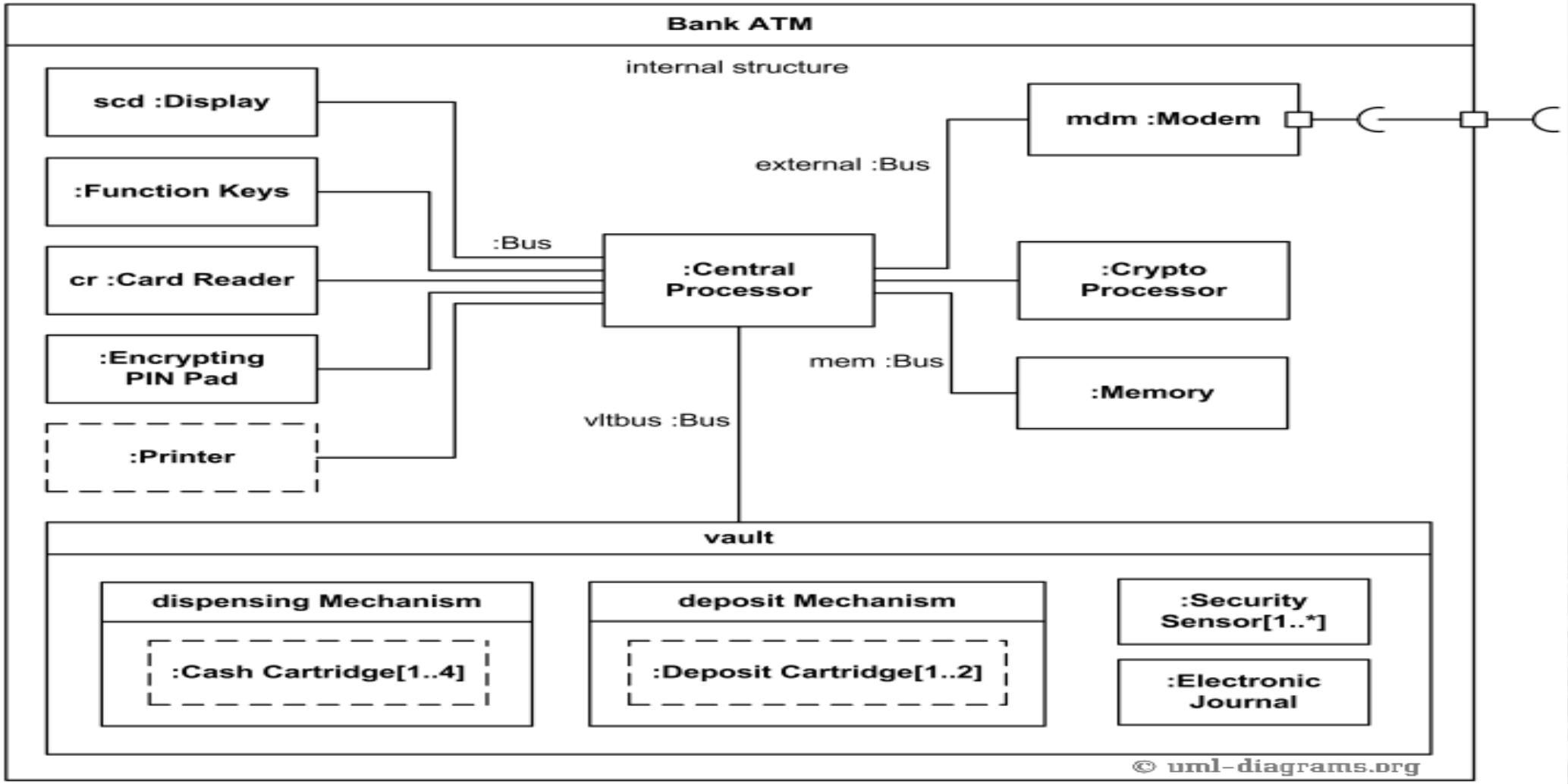
- Composite structure diagram for store manager



Example contd..



Example


Bank ATM
internal structure
scd :Display
:Function Keys
cr :Card Reader
:Encrypting PIN Pad
:Printer
:Bus
external :Bus
vltbus :Bus
mem :Bus
mdm :Modem
:Central Processor
:Crypto Processor
:Memory
vault
dispensing Mechanism
:Cash Cartridge[1..4]
deposit Mechanism
:Deposit Cartridge[1..2]
:Security Sensor[1..*]
:Electronic Journal
© uml-diagrams.org

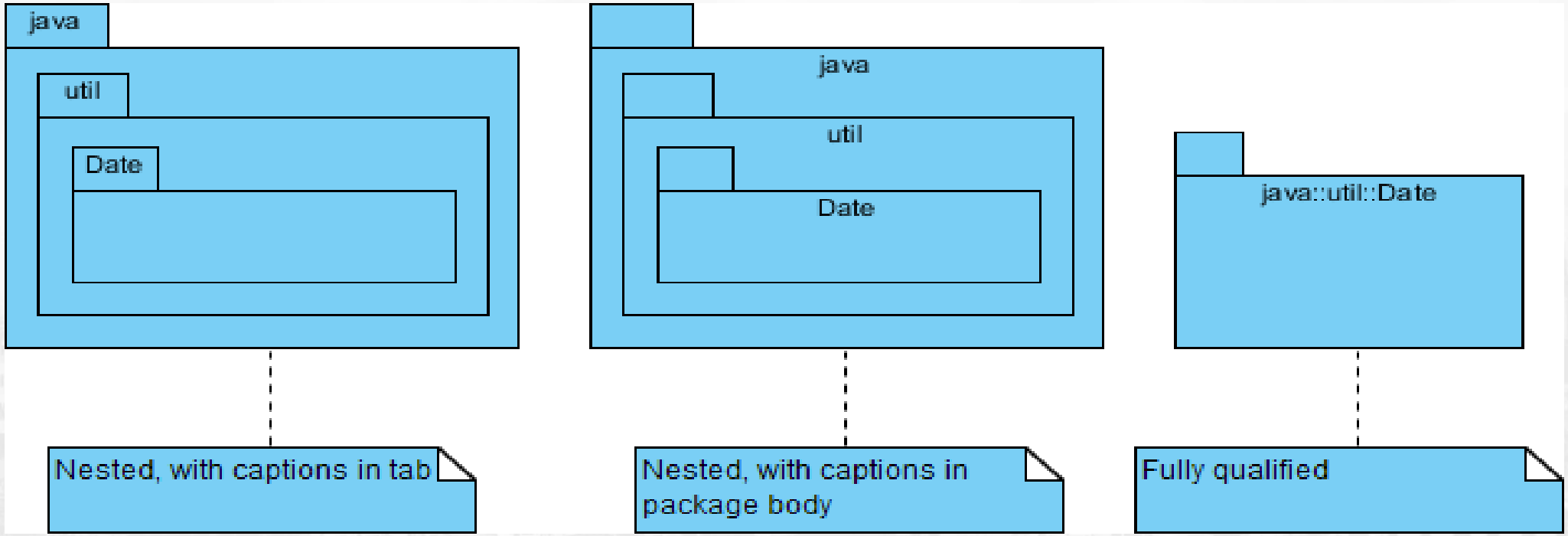
Package diagrams

- Package diagram is used to simplify complex class diagrams, you can group classes into packages. A package is a collection of logically related UML elements.
- Packages appear as rectangles with small tabs at the top.
- The package name is on the tab or inside the rectangle.
- The dotted arrows are dependencies.
- One package depends on another if changes in the other could possibly force changes in the first.

Package diagrams

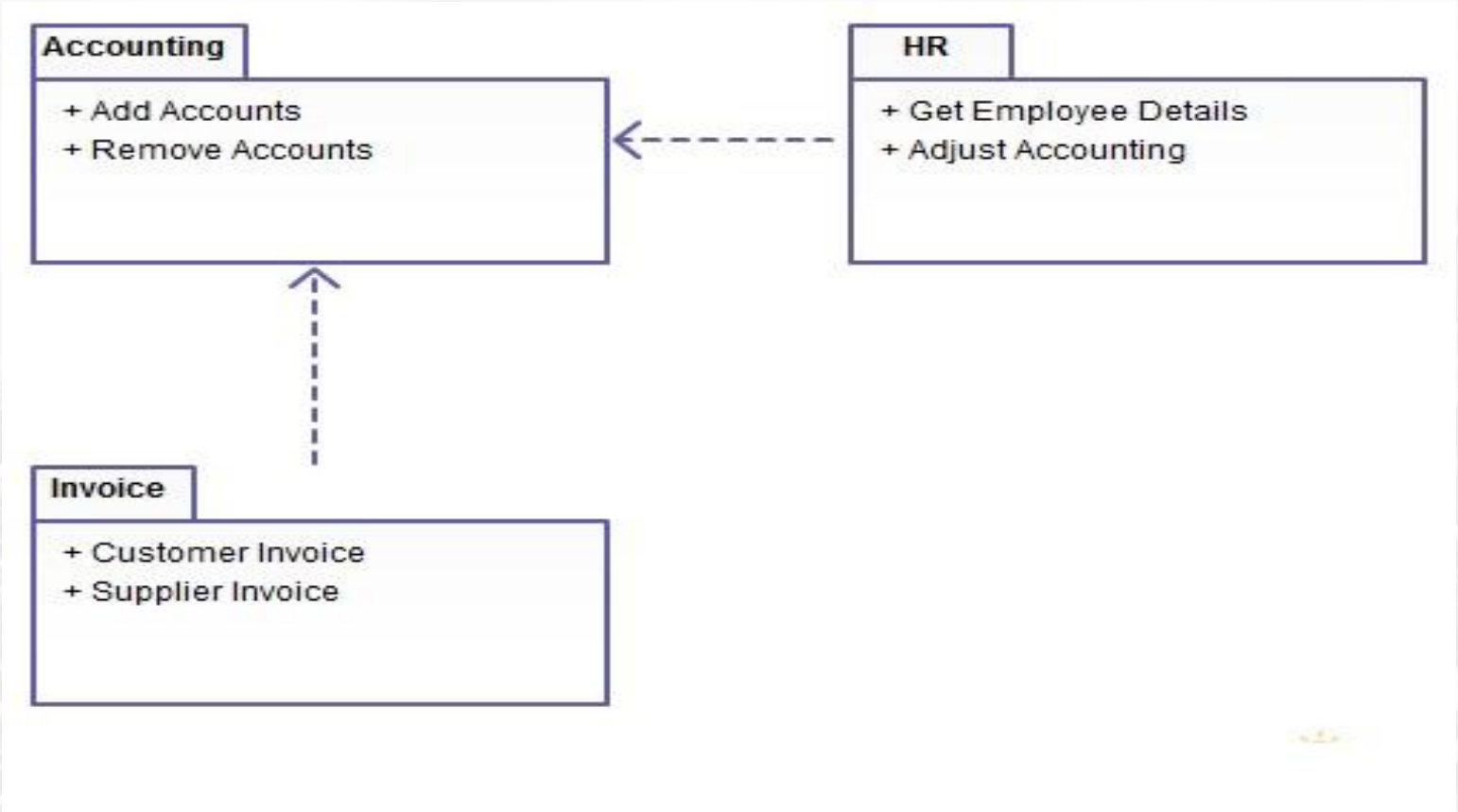
- Purpose of Package Diagrams
- Package diagrams are used to structure high level system elements. Packages are used for organizing large system which contains diagrams, documents and other key deliverables.
- Package Diagram can be used to simplify complex class diagrams, it can group classes into packages.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

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



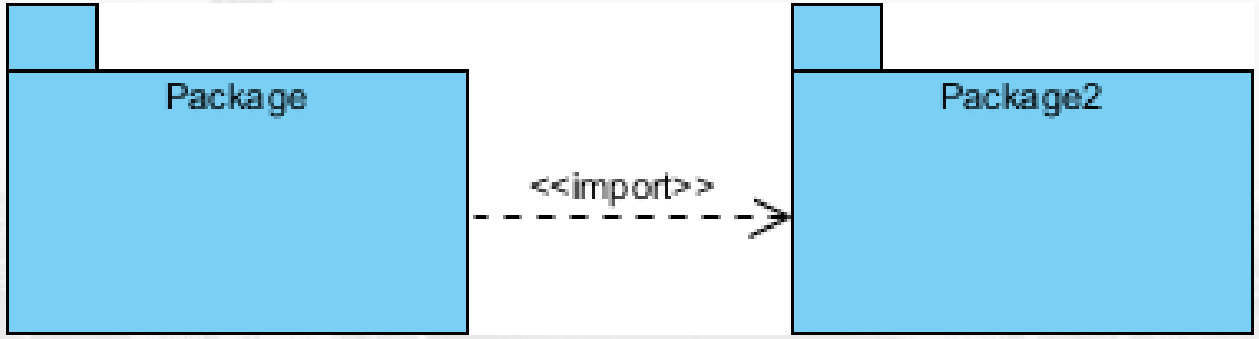
Package Diagram

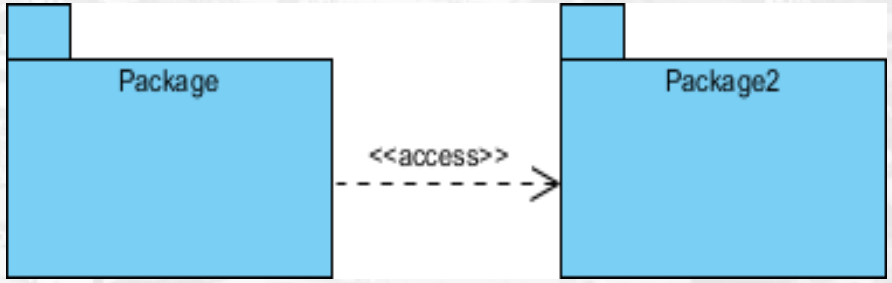
- It shows the decomposition of the model itself into organization units & their dependencies



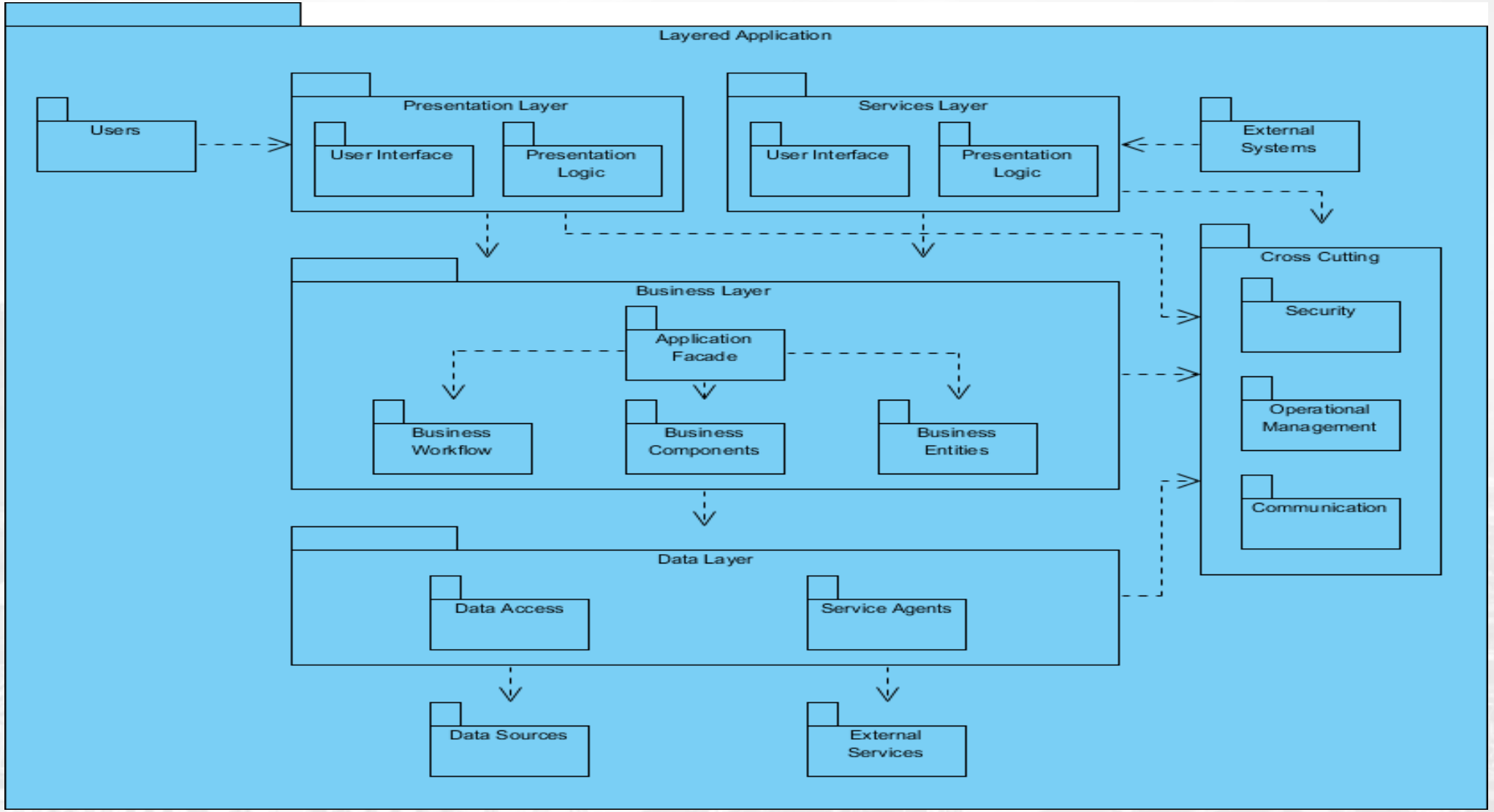
Dependency Relationship

- Two stereotypes are used in dependency: <<import>> & <<access>>.
- **<<import>>** - one package imports the functionality of other package
- **<<access>>** - one package requires help from functions of other package.

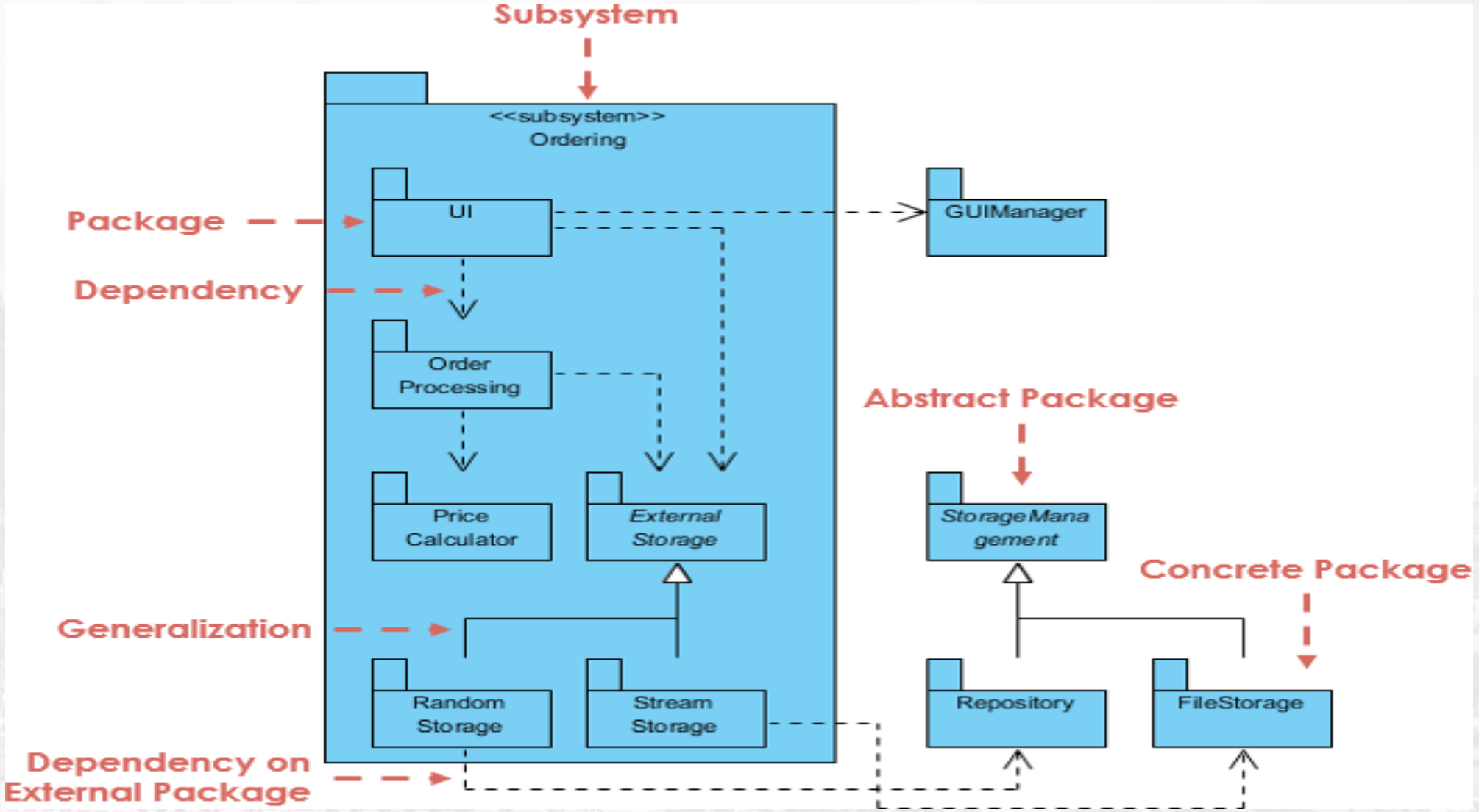




Package Diagram Example - Layered Application



Package Diagram Example - Order Subsystem



Package Diagram Example - Order Subsystem

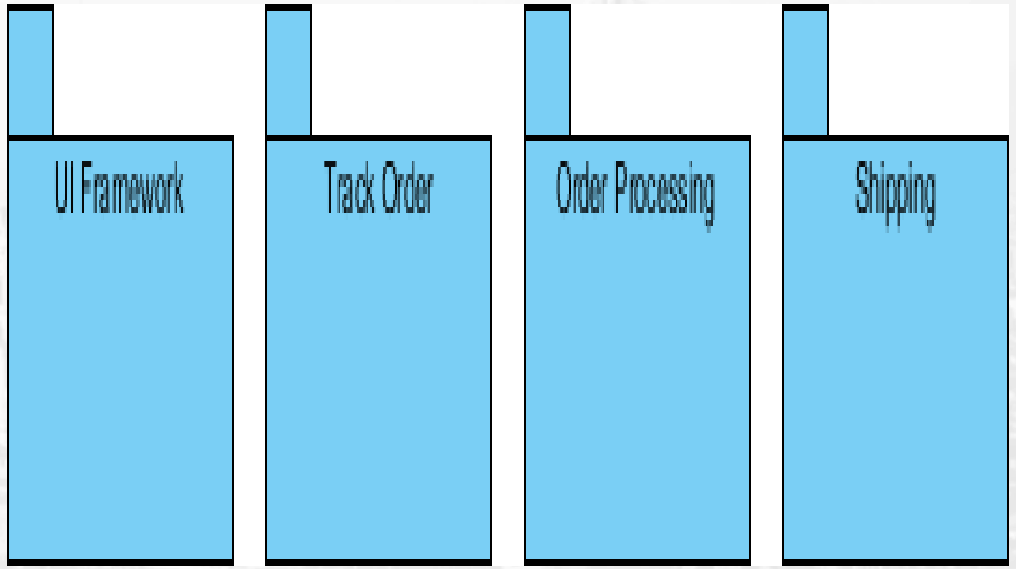
- **Order Processing System - The Problem Description**

We are going to design package diagram for "Track Order" scenario for an online shopping store. Track Order module is responsible for providing tracking information for the products ordered by customers. Customer types in the tracking serial number, Track Order modules refers the system and updates the current shipping status to the customer.

- Based on the project Description we should first identify the packages in the system and then related them together according to the relationship:

Identify the packages of the system

- There is a track order module, it has to talk with other module to know about the order details, let us call it "Order Details".
- Next after fetching Order Details it has to know about shipping details, let us call that as "Shipping".

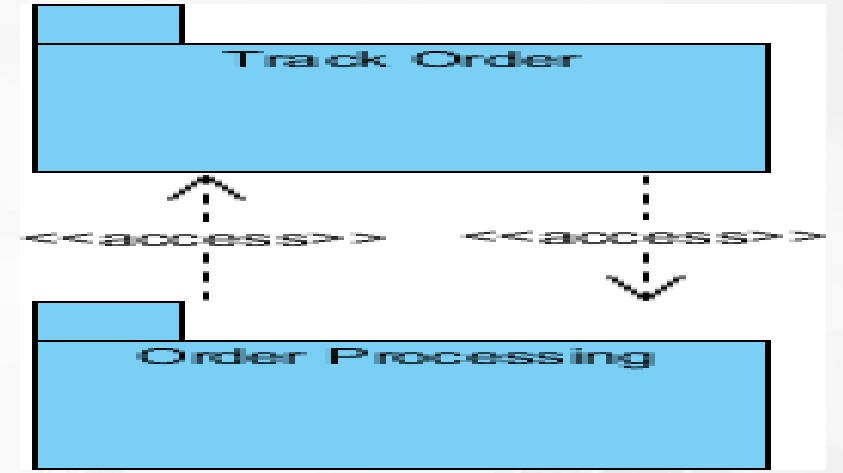


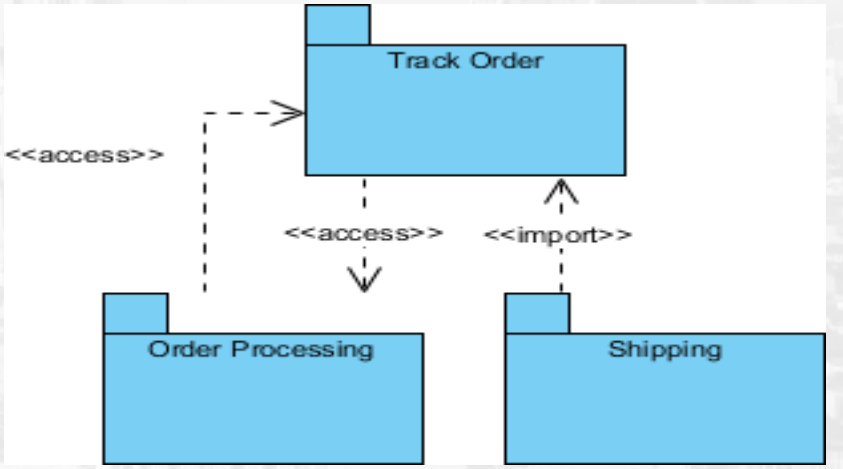
Package Diagram Example - Order Subsystem

Identify the dependencies in the System

- Track order should get order details from "Order Details" and "Order Details" has to know the tracking info given by the customer. Two modules are accessing each other which suffices <<access>> dual dependency

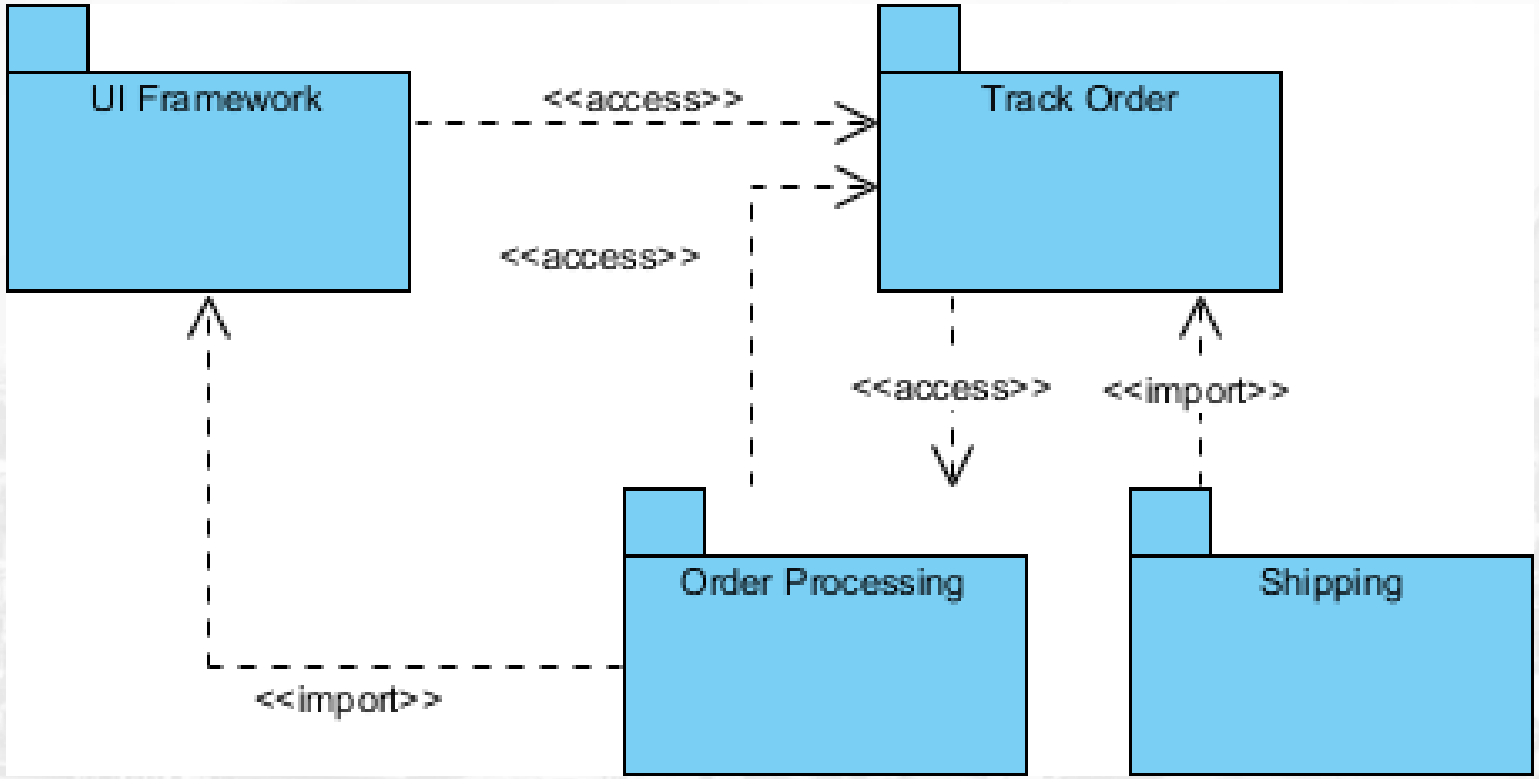
- To know shipping information, "Shipping" can import "Track Order" to make the navigation easier.





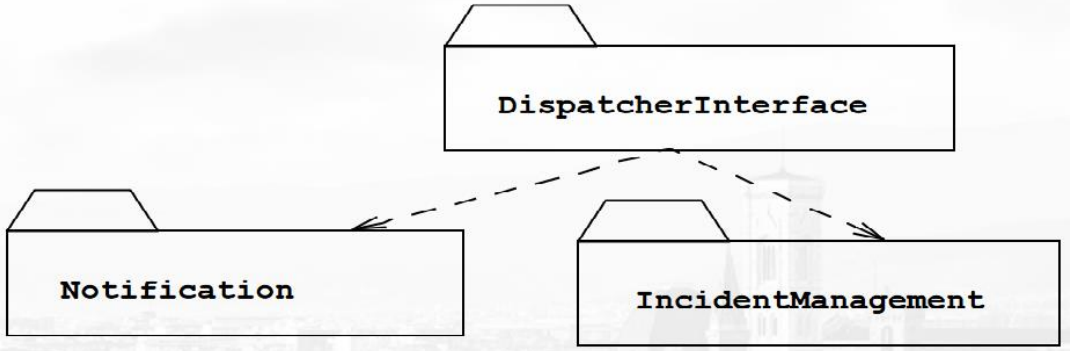
Package Diagram Example - Order Subsystem

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

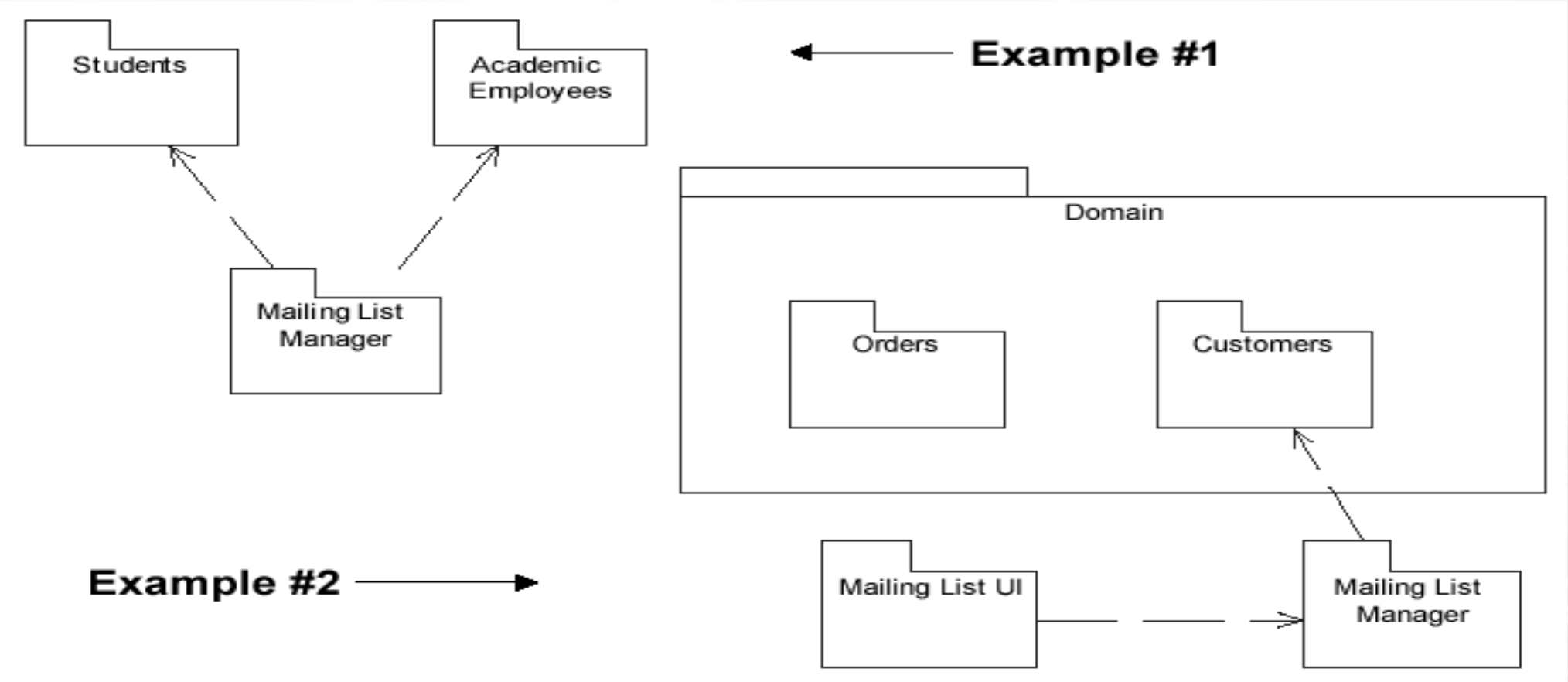


Package Example

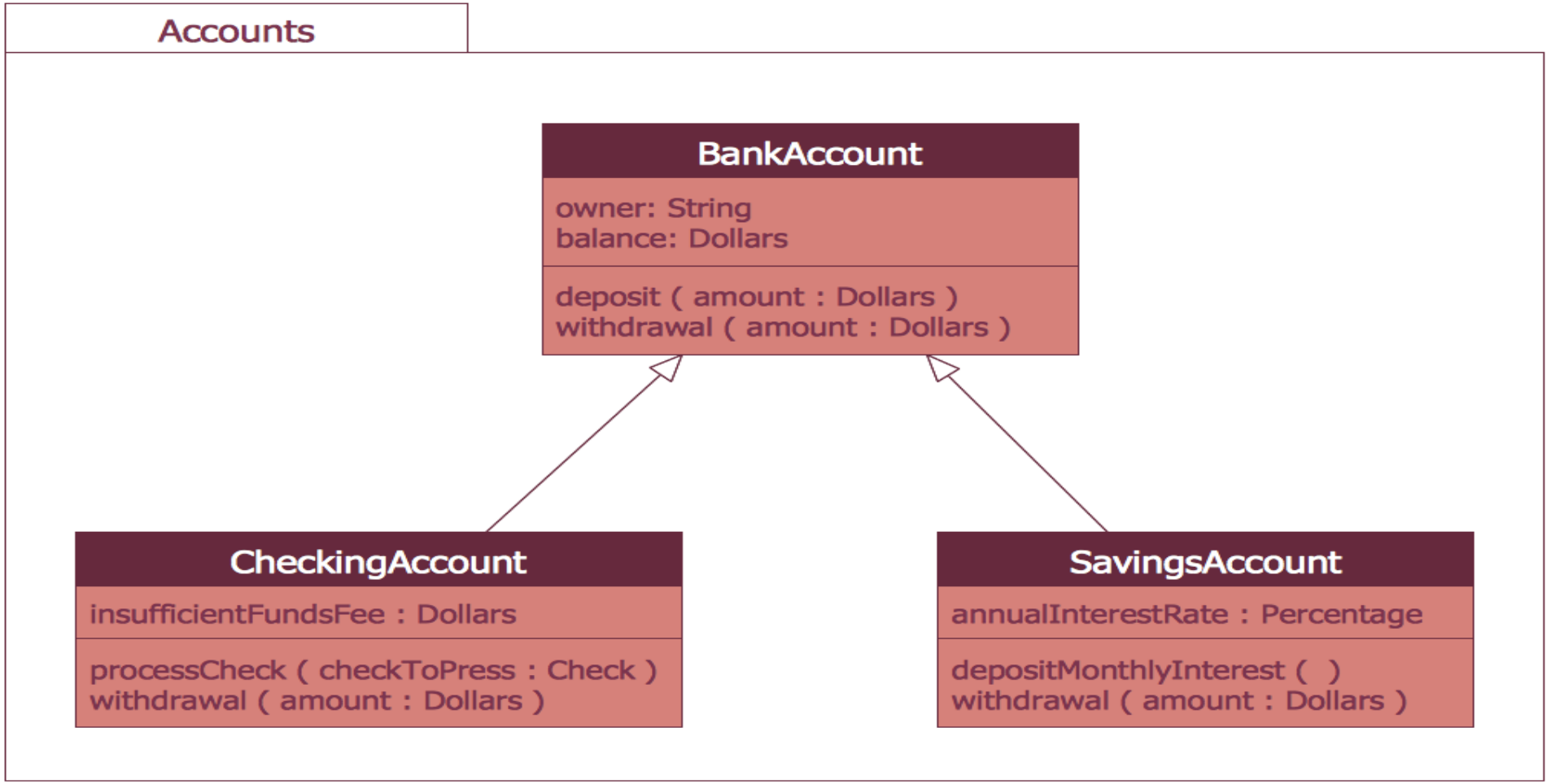
They provide a clear view of the hierarchical structure of the various UML elements within a given system.
These diagrams can simplify complex class diagrams into well-ordered visuals.
They offer valuable high-level visibility into large-scale projects and systems.
Package diagrams can be used to visually clarify a wide variety of projects and systems.
These visuals can be easily updated assystems and projects evolve.
Basic com



More Package Examples



Package Diagram Example - Accounts



Component Diagram

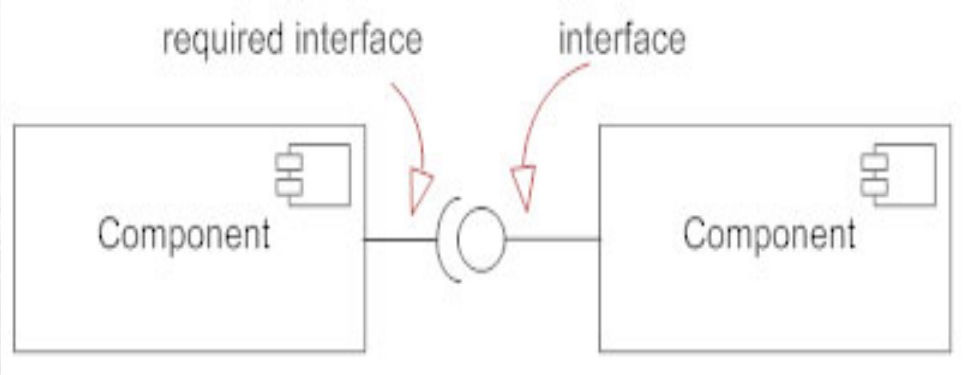
- Component diagrams are used to model the physical aspects of a system.
- Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.
- Component diagrams are used to visualize the organization and relationships among components in a system.
- Component diagrams can also be described as a static implementation view of a system.
- Static implementation represents the organization of the components at a particular moment.
- They show the organization and dependencies between a set of components.
- It represents the software layout of the system

Basic Component Diagram Symbols and Notations

- Component
- A component is a logical unit block of the system, a slightly higher abstraction than classes.

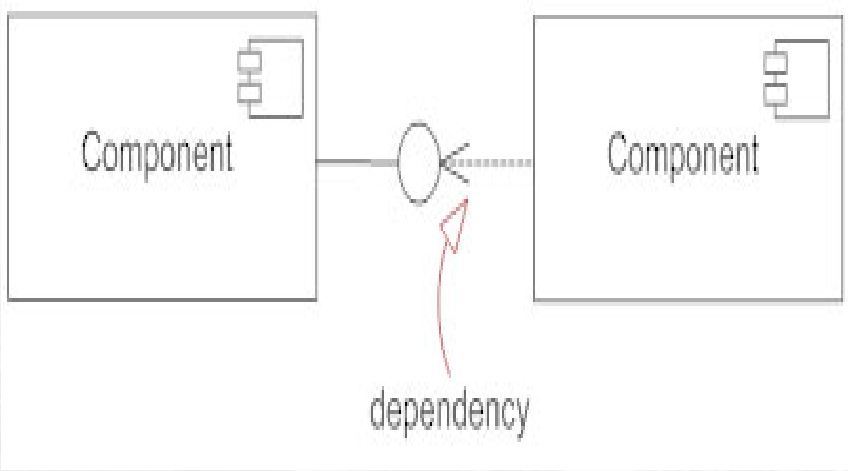
Basic Component Diagram Symbols and Notations

- Interface
- An interface (small circle or semi-circle on a stick) describes a group of operations used (required) or created (provided) by components. A full circle represents an interface created or provided by the component. A semi-circle represents a required interface, like a person's input.



Component diagram notation

- Dependencies



Component Views

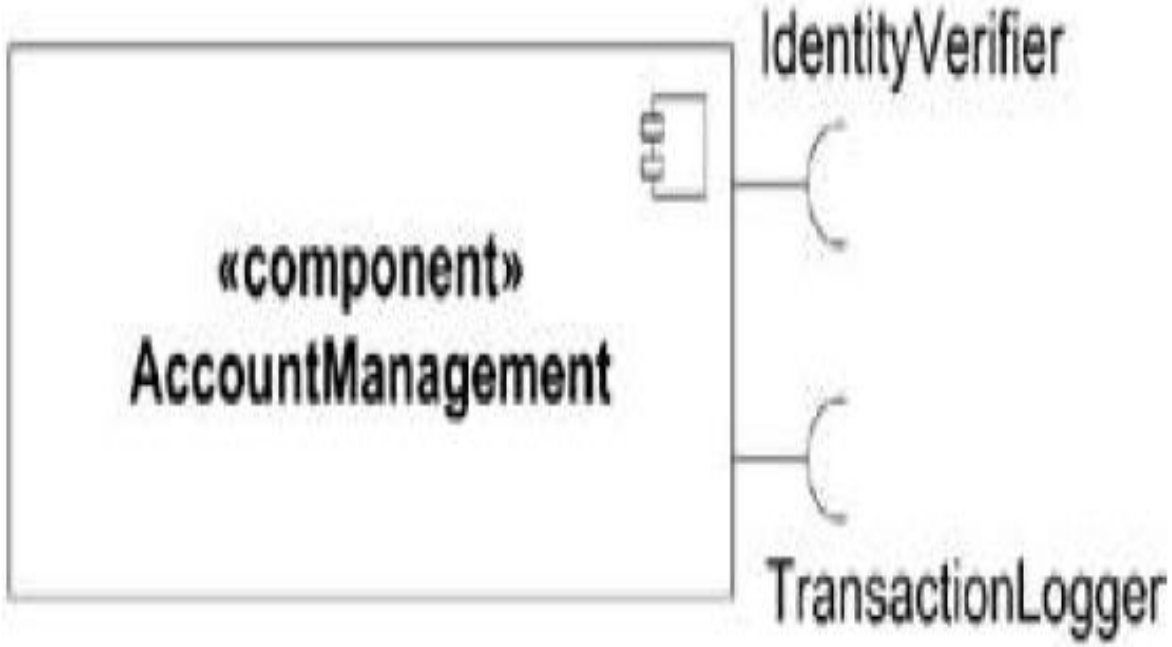
- UML uses two views of components, a **black-box view** and a **white-box view**.
- The black-box view shows a component from an **outside perspective**;
- The white-box view shows how a component **realizes the functionality** specified by its provided interfaces.

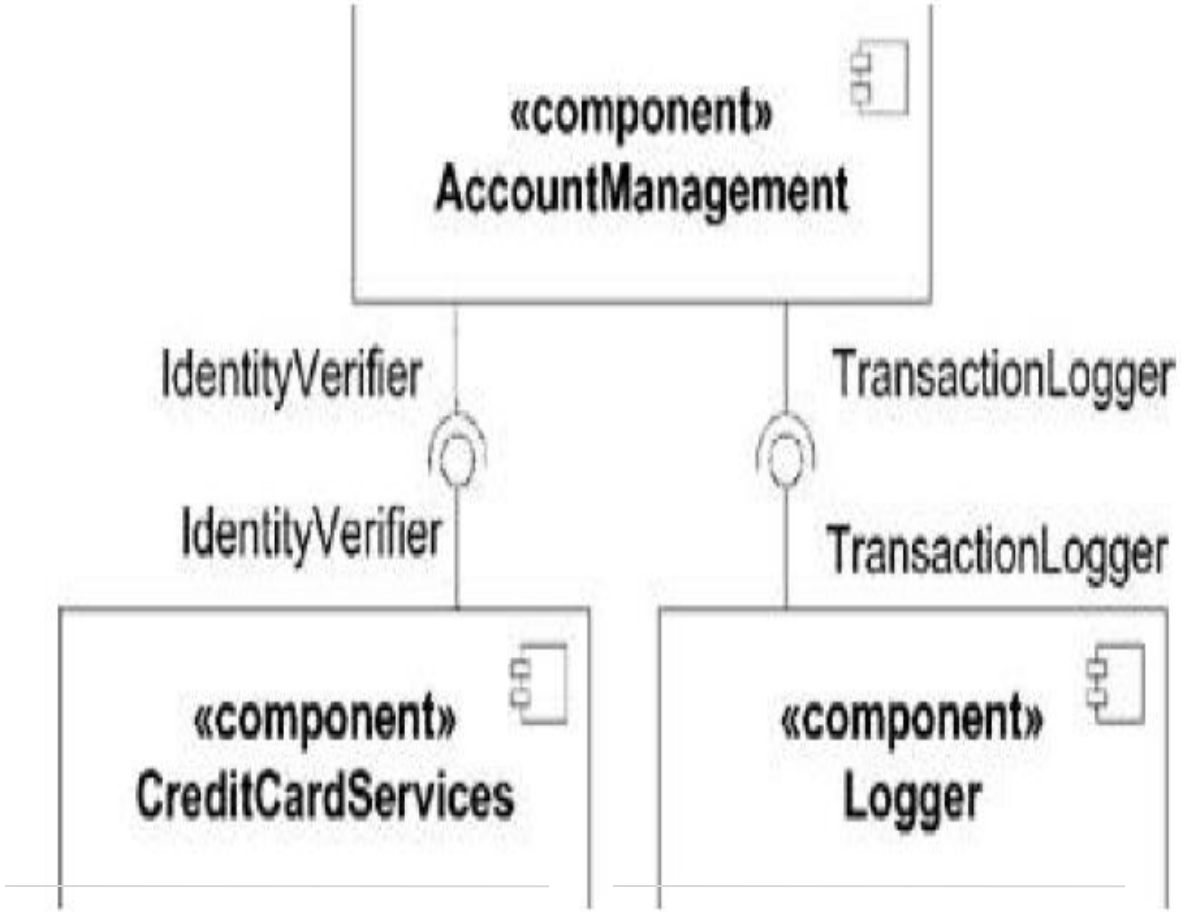
Black-Box View

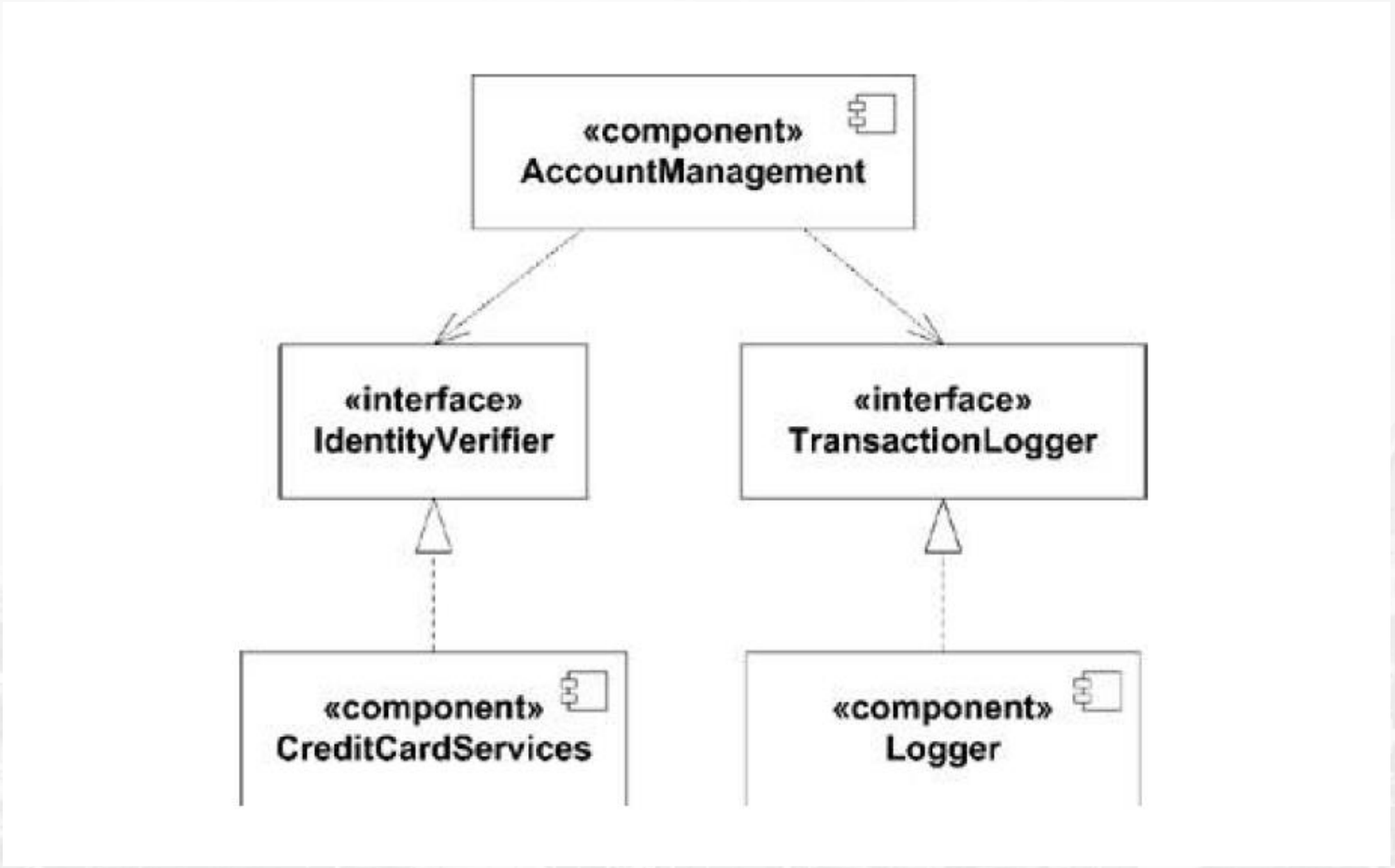
- The black-box view of a component shows the

interfaces the component provides the interfaces it requires

- It does not specify anything about the internal implementation of the component.

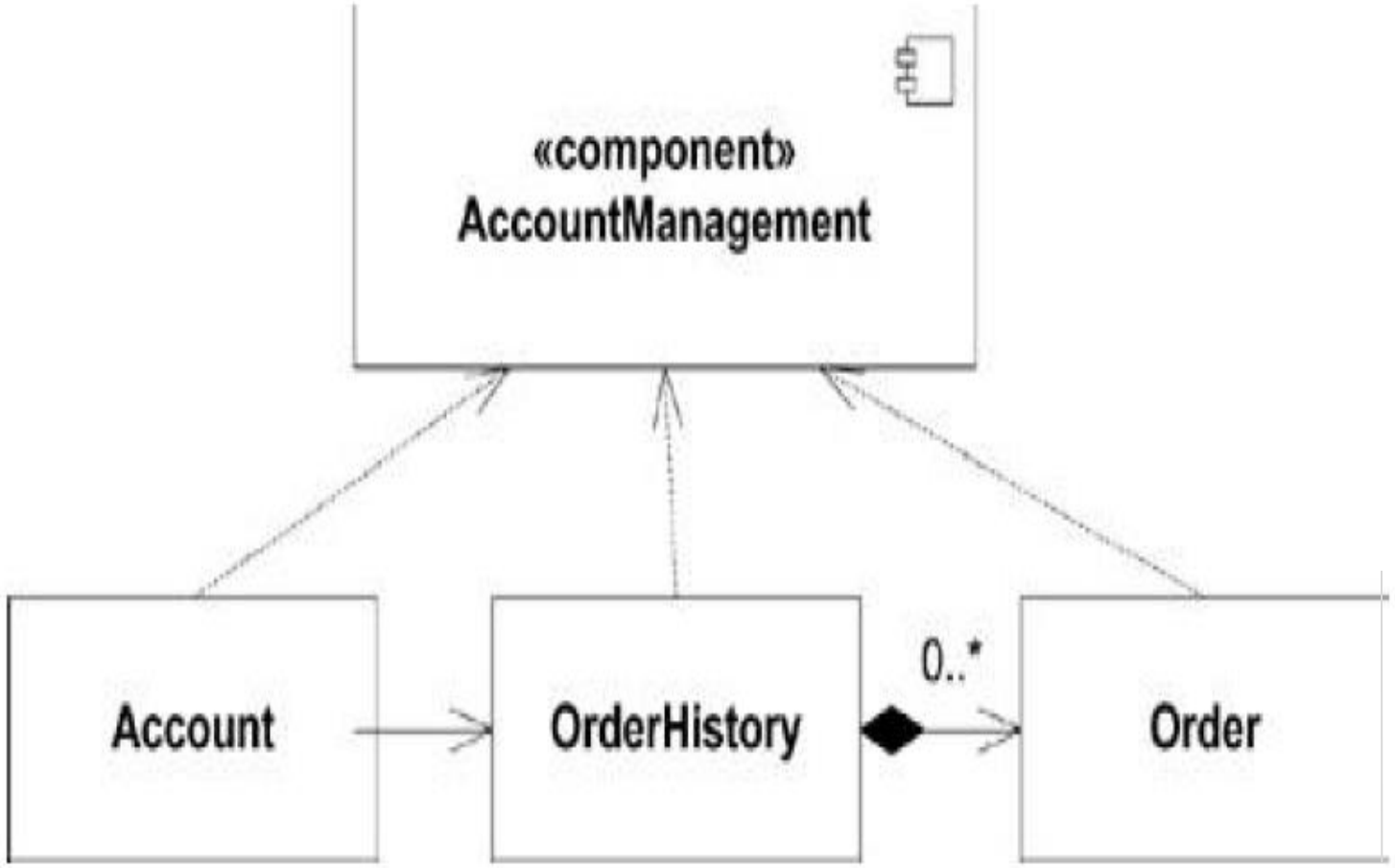

IdentityVerifier
«component»
AccountManagement
TransactionLogger


«component»
AccountManagement
IdentityVerifier
TransactionLogger
IdentityVerifier
TransactionLogger
«component»
CreditCardServices
«component»
Logger


«component»
AccountManagement
«interface»
IdentityVerifier
«interface»
TransactionLogger
«component»
CreditCardServices
«component»
Logger

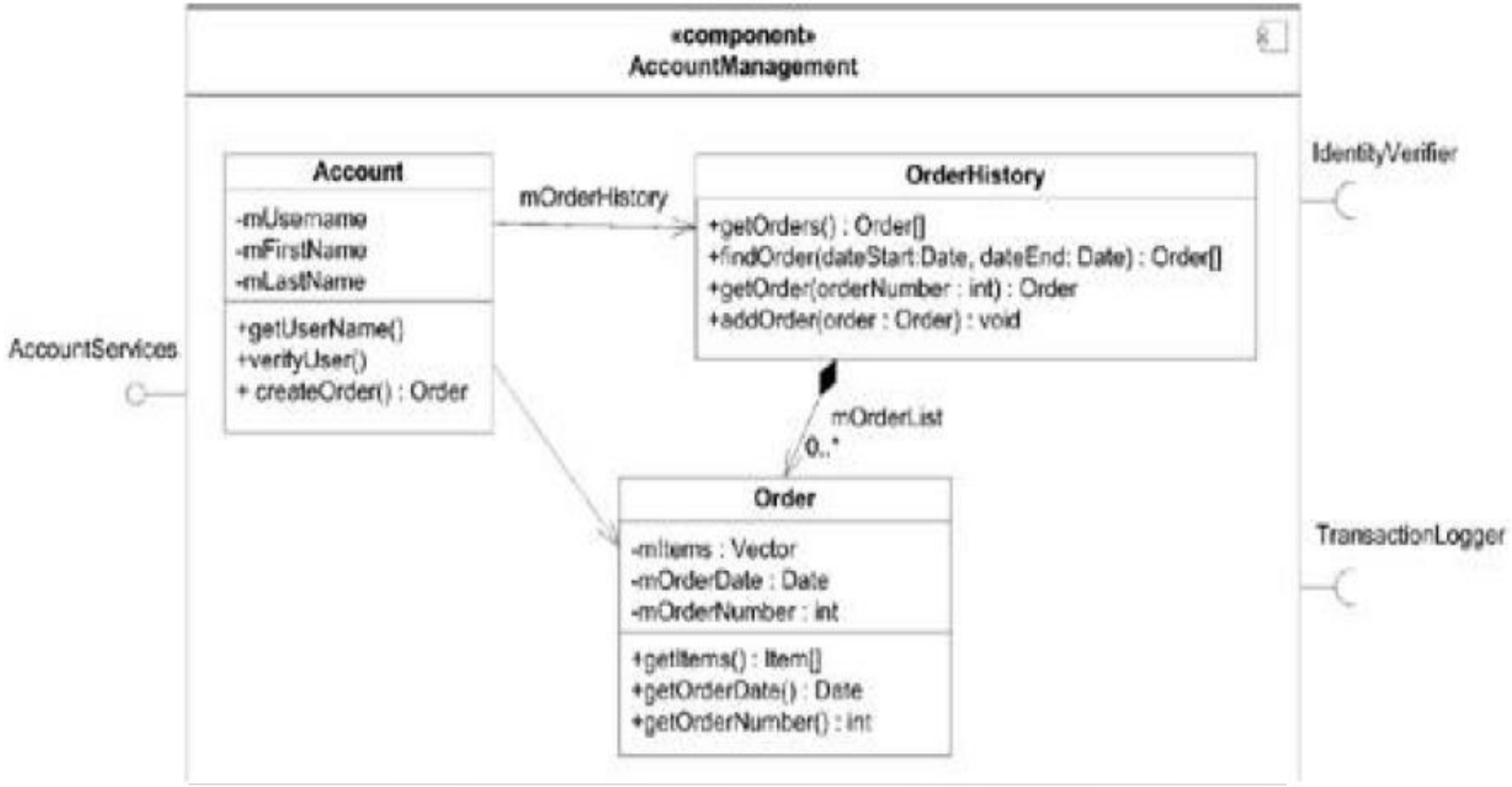
White Box view

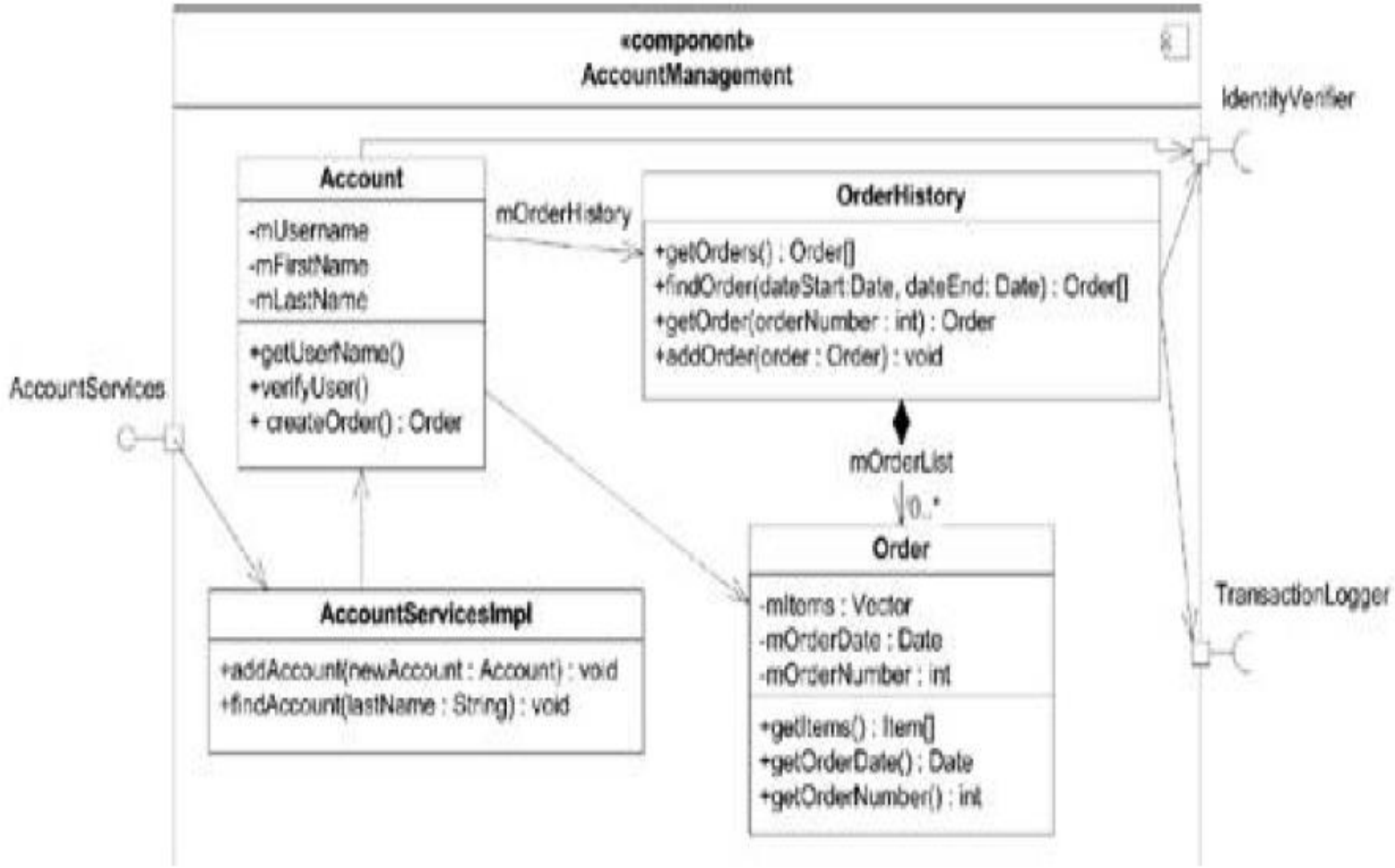
- In order to **provide details about the implementation** of a component, UML defines a white box view.
- The white-box view shows exactly **how a component realizes the interfaces** it provides.
- This is typically done using classes and is illustrated with a class diagram


«component»
AccountManagement
Account
OrderHistory
0..*
Order

Detailed Realization

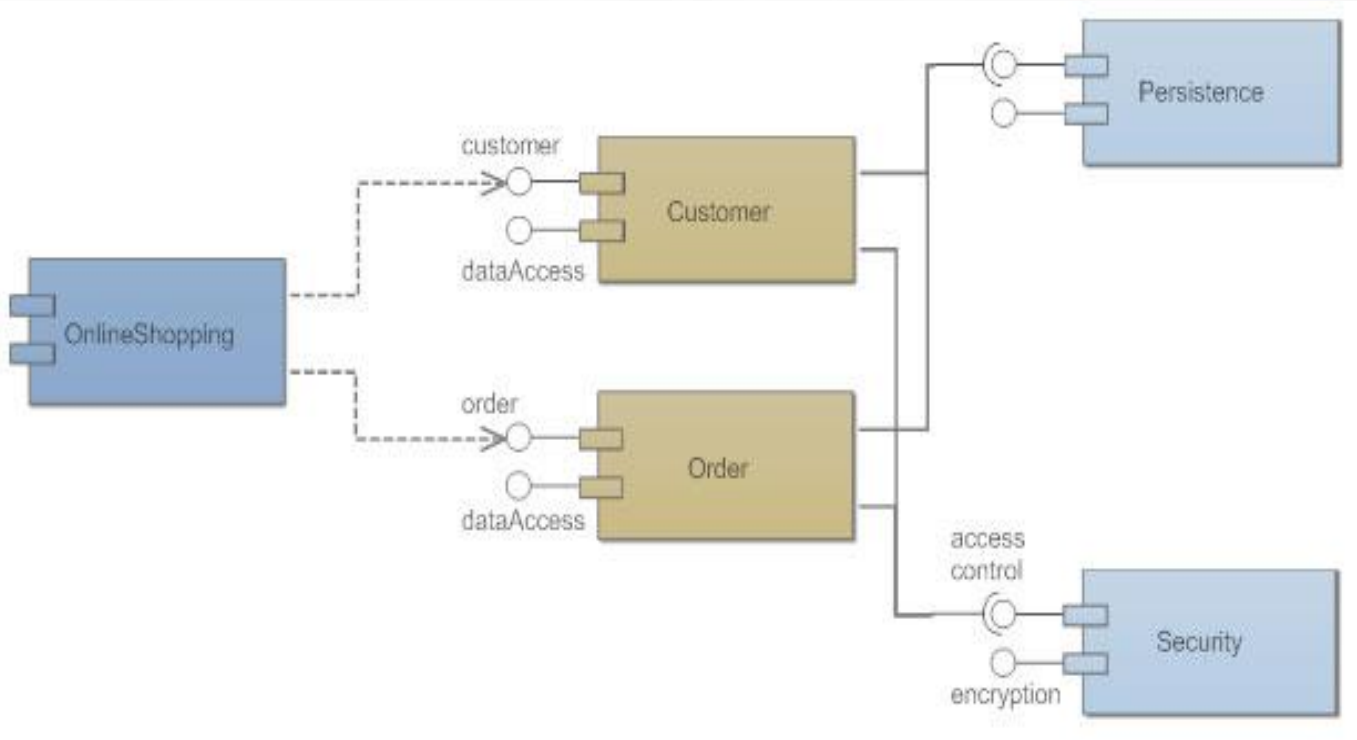
Level 2



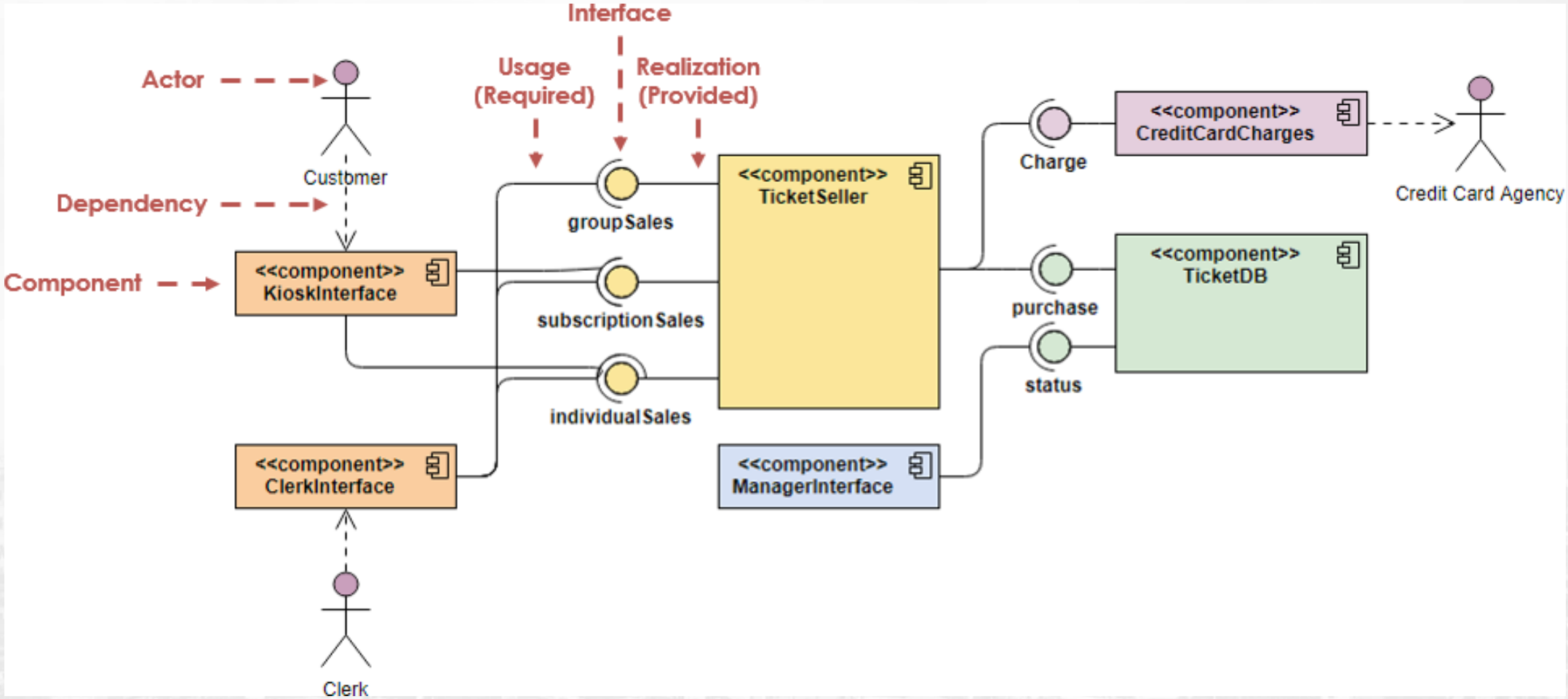

«component»
AccountManagement
IdentityVerifier
Account
-mUsername
-mFirstName
-mLastName
+getUserName()
+verifyUser()
+ createOrder() : Order
mOrderHistory
OrderHistory
+getOrders() : Order[]
+findOrder(dateStart:Date, dateEnd: Date) : Order[]
+getOrder(orderNumber : int) : Order
+addOrder(order : Order) : void
AccountServices
mOrderList
0..*
Order
-mItems : Vector
-mOrderDate : Date
-mOrderNumber : int
+getItems() : Item[]
+getOrderDate() : Date
+getOrderNumber() : int
TransactionLogger
AccountServicesImpl
+addAccount(newAccount : Account) : void
+findAccount(lastName : String) : void

Component diagram example

- Online shopping system – component diagram



Component diagram – Ticket selling system



Deployment Diagram

- Deployment diagrams are used for modeling configuration of ***run-time processing*** nodes and the components that live on them.
- Deployment diagrams are used to model the ***static deployment view*** of a system.
- This involves modeling the topology of the hardware on which the system executes.

Deployment Diagram

- Deployment diagrams Show the physical relationship between hardware and software in a system
- Hardware elements
- Computers (clients, servers)
- Embedded processors
- Devices (sensors, peripherals)
- Are used to show the nodes where software components reside in the run-time system

Deployment diagram

- **Purpose of Deployment Diagrams**

✓ They show the structure of the run-time system

✓ They capture the hardware that will be used to implement the system and the links between different items of hardware.

✓ They model physical hardware elements and the communication paths between them

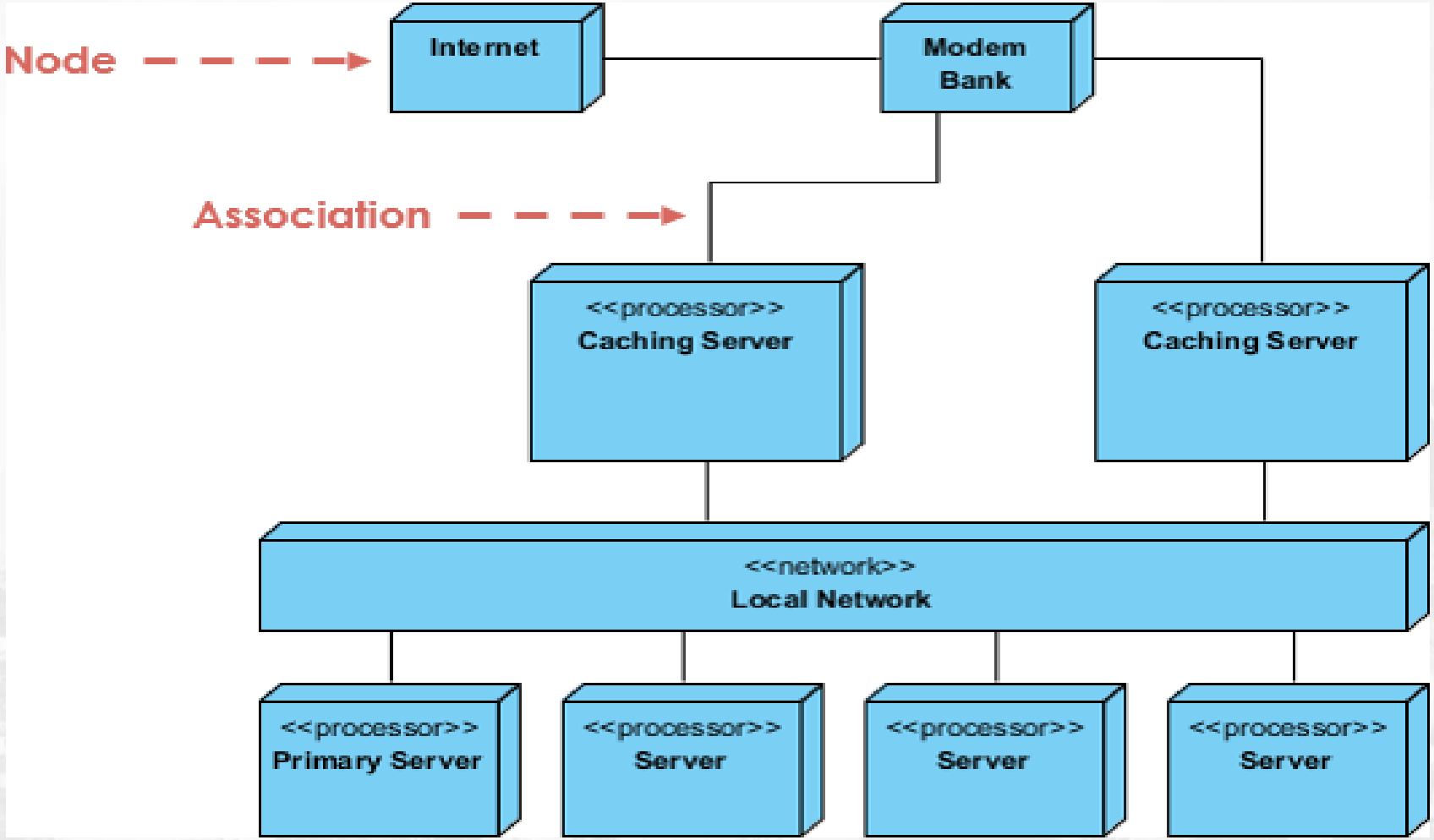
✓ They can be used to plan the architecture of a system.

✓ They are also useful for Document the deployment of software components or nodes

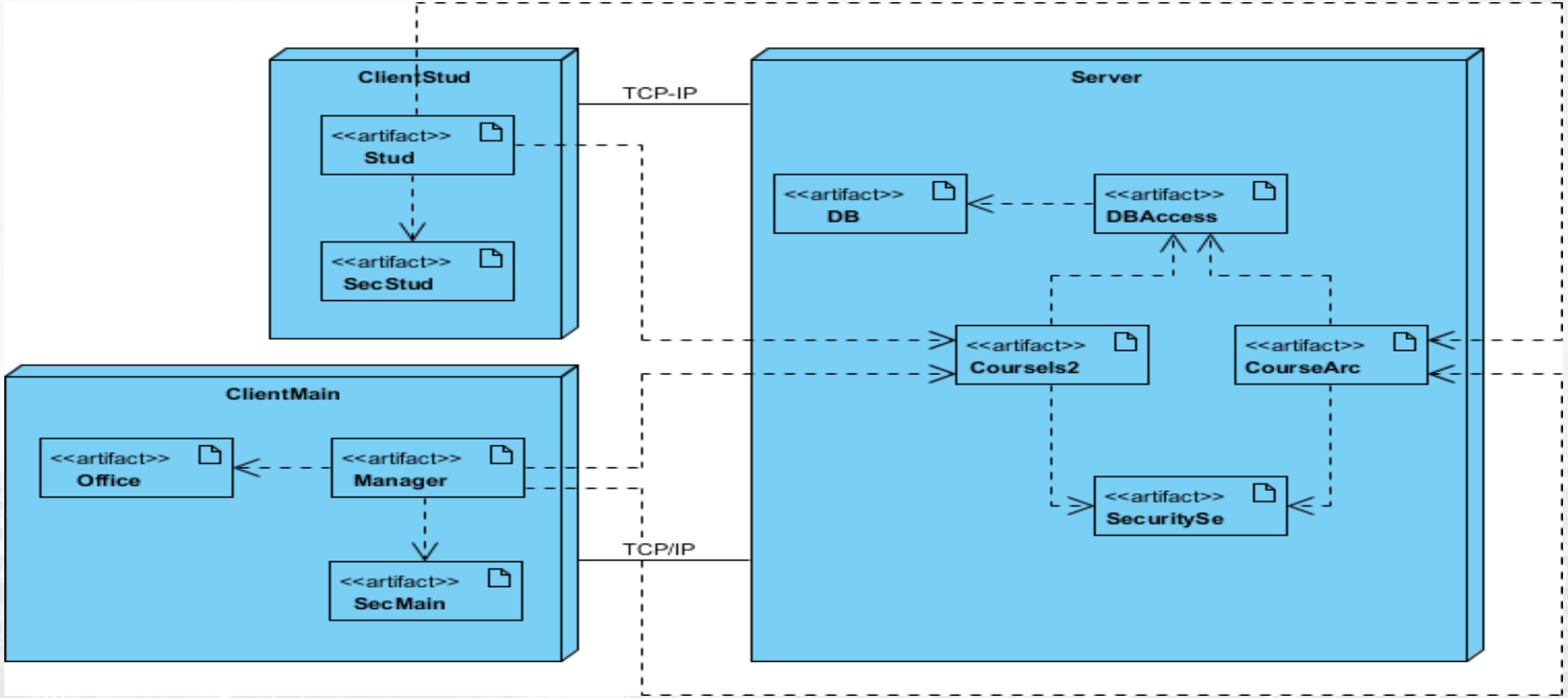
Deployment diagram

- Deployment diagram contains nodes and connections
- A node usually represent a piece of hardware in the system
- A connection depicts the communication path used by the hardware to communicate
- Usually indicates the method such as TCP/IP

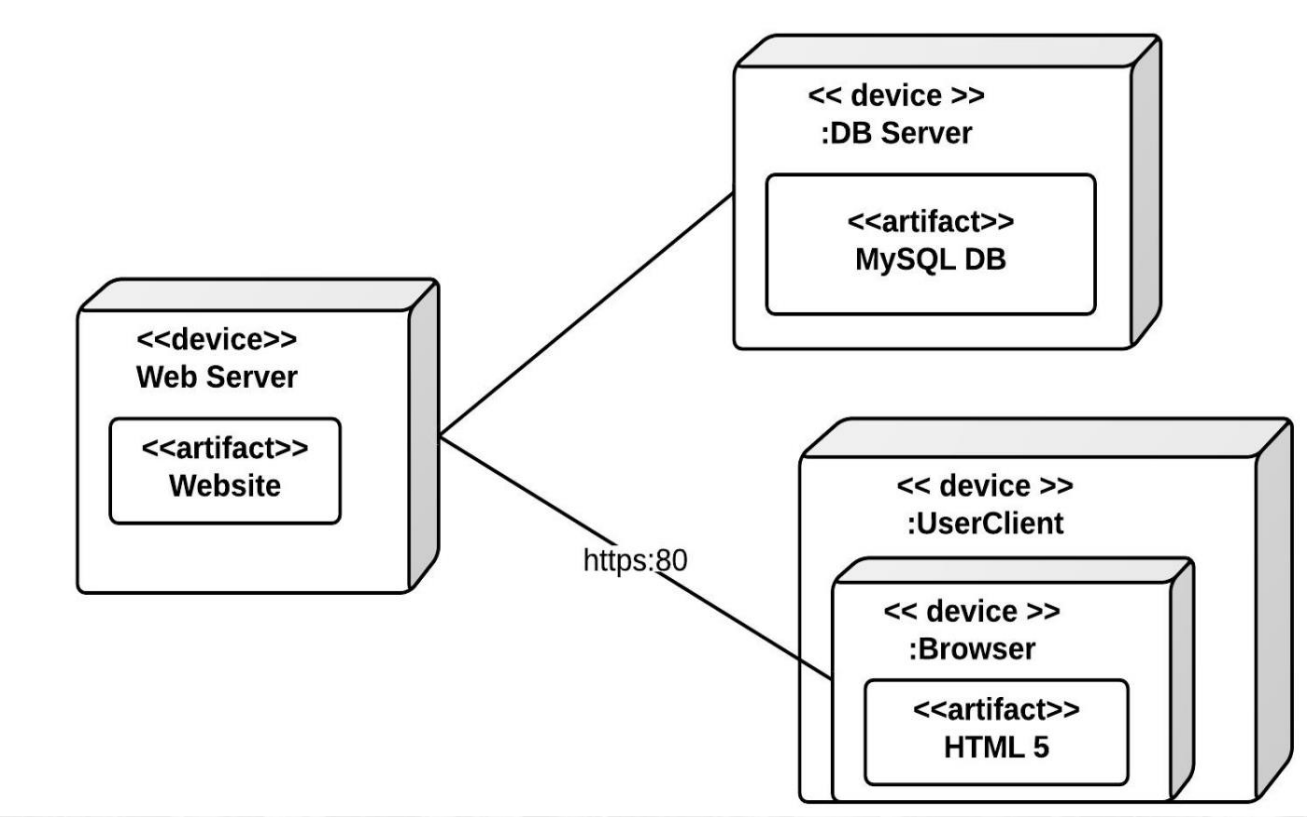
Deployment diagram of embeded system



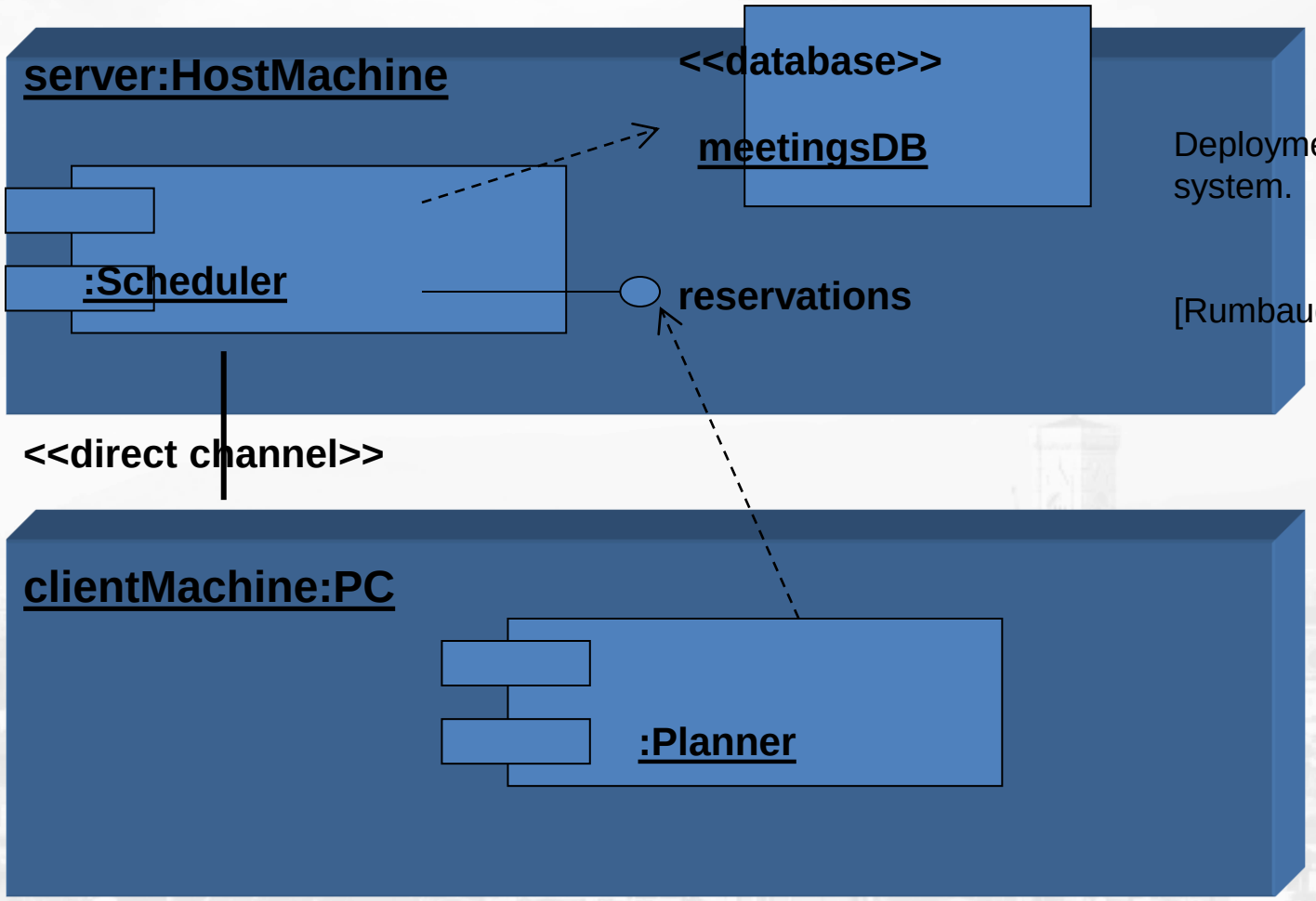
Deployment diagram of TCP/IP



Example:


<< device >>
:DB Server
<<artifact>>
MySQL DB
<<device>>
Web Server
<<artifact>>
Website
https:80
<< device >>
:UserClient
<< device >>
:Browser
<<artifact>>
HTML 5

Deployment Diagram

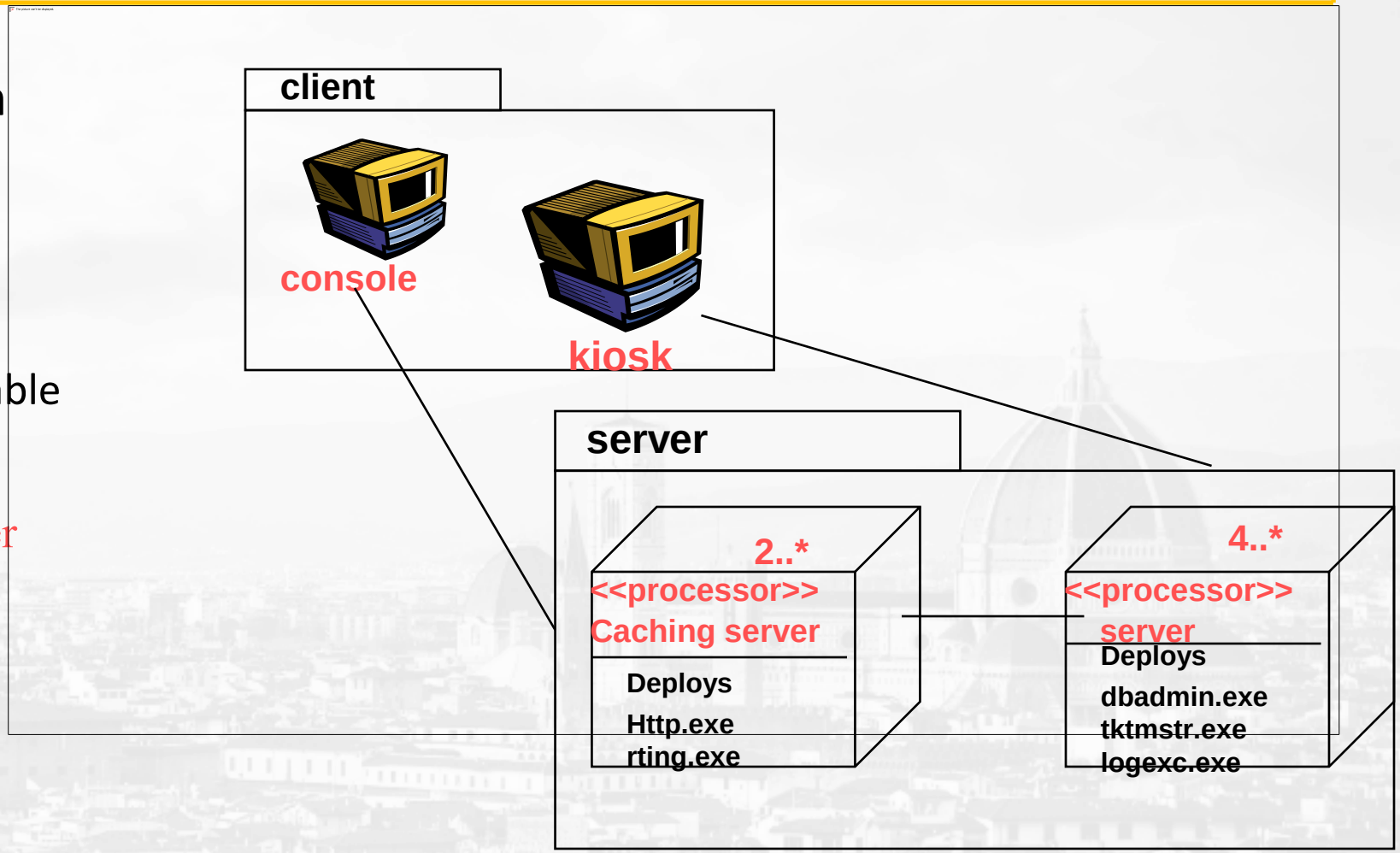


Deployment diagram of a client-server system.

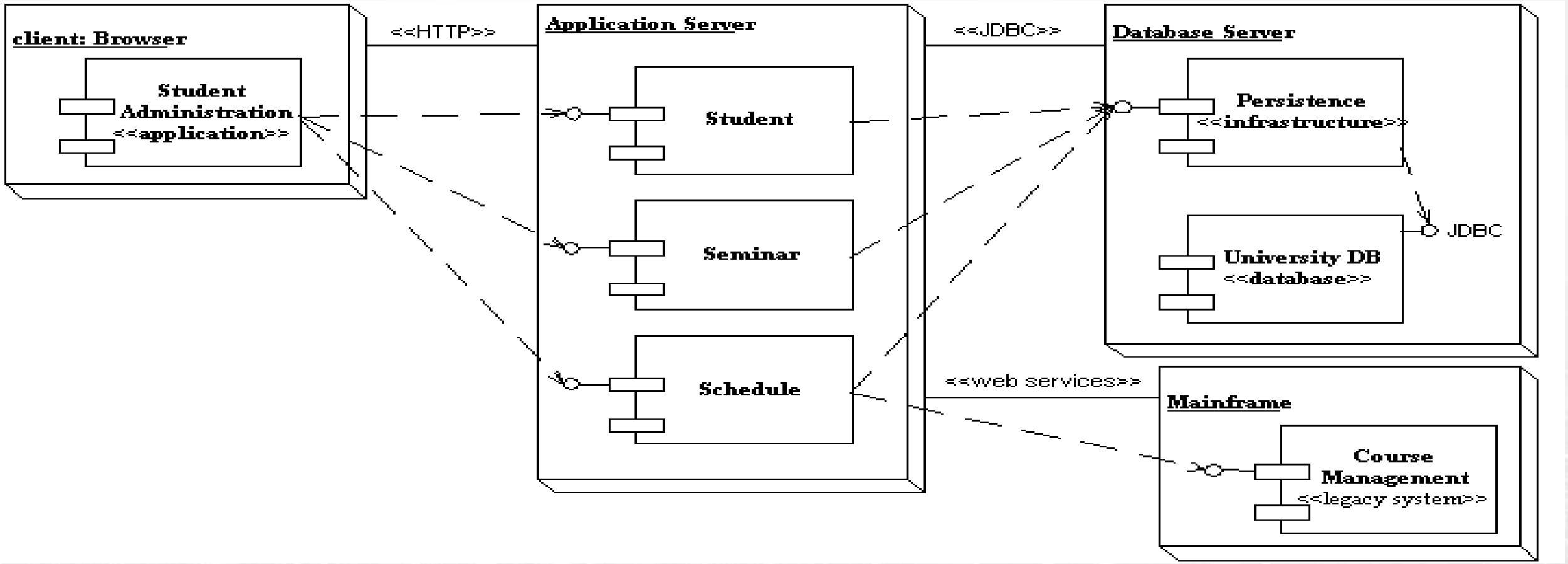
[Rumbaugh,99]

Client-Server System

- Human resource system
- 2 pkgs: client, server
- Client: 2 nodes
  - console and kiosk
  - stereotyped, distinguishable
- Server: 2 nodes
  - caching server and server
  - Multiplicities are used



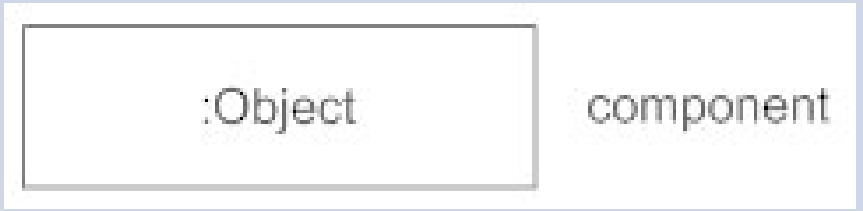
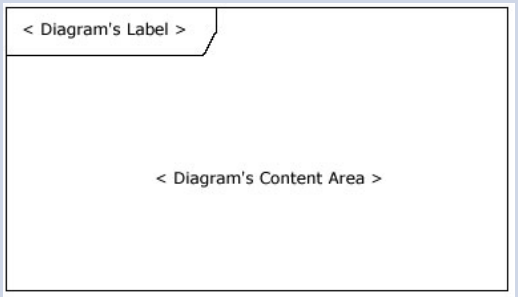
Sample Communication Links

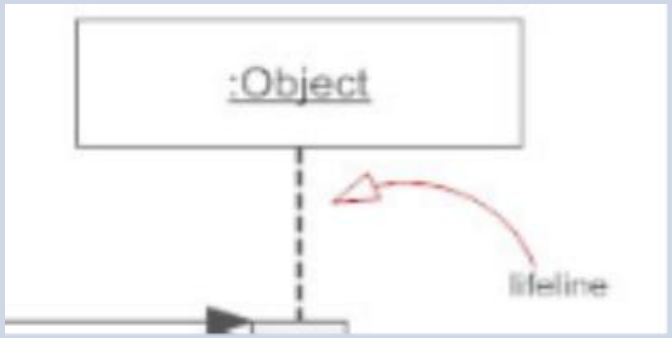
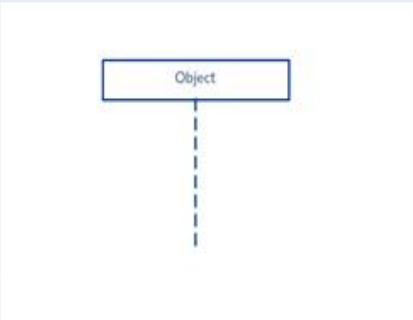


UML Sequence Diagrams

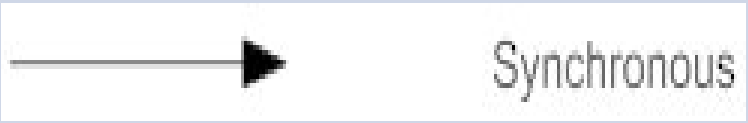
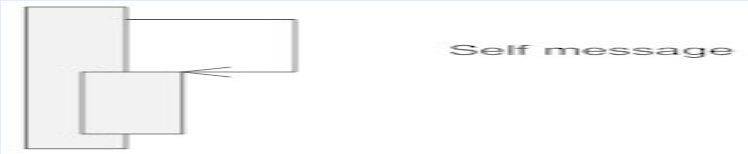
- Sequence diagrams model the dynamic aspects of a software system
- The emphasis is on the "sequence" of messages rather than relationship between objects
- Sequence diagrams provide more detail and show the messages exchanged among a set of objects over time.
- The main purpose of this diagram is to represent how different business object interacts

UML Sequence Diagrams

S.No	Name	Description	Notation
1	Class Roles or Participants	Class roles describe the way an object will behave in context	
2	Activation or Execution Occurrence/ Scope	Activation boxes represent the time an object needs to complete a task.	
3	Diagram Boundry		

S.No	Name	Description	Notation
3	Messages	Messages are arrows that represent communication between objects.	
4	Lifelines	Lifelines represent either roles or object instances that participate in the sequence being modeled.	

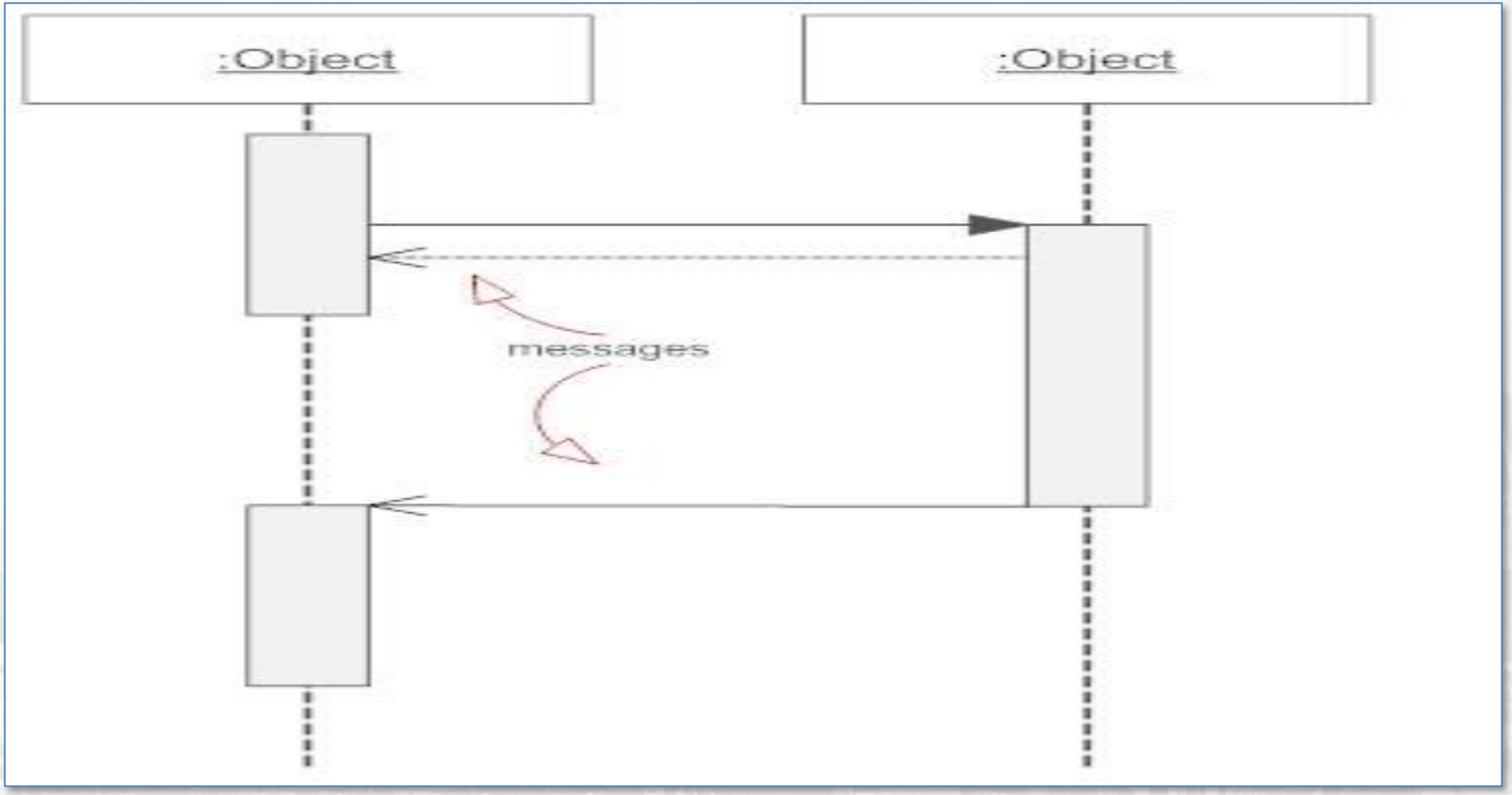
UML Sequence Diagrams- Types of Messages in Sequence Diagrams

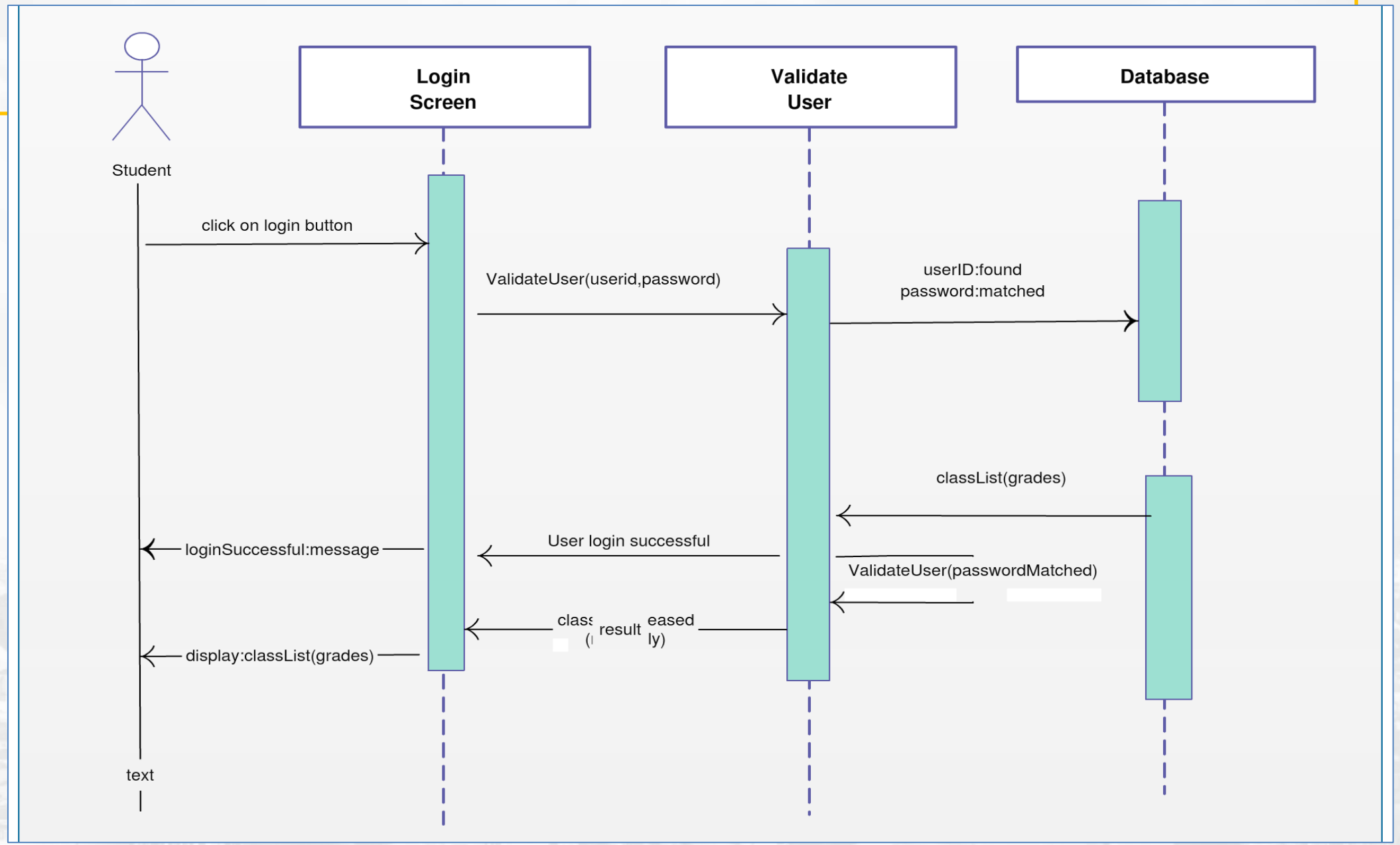
S.No	Name	Description	Notation
1	Synchronous Message	A synchronous message requires a response before the interaction can continue.	
2	Asynchronous Message	Asynchronous messages don't need a reply for interaction to continue.	
3	Reply or Return Message	A reply message is drawn with a dotted line and an open arrowhead pointing back to the original lifeline.	
4	Self Message	A message an object sends to itself, usually shown as a U shaped arrow pointing back to itself.	

UML Sequence Diagrams

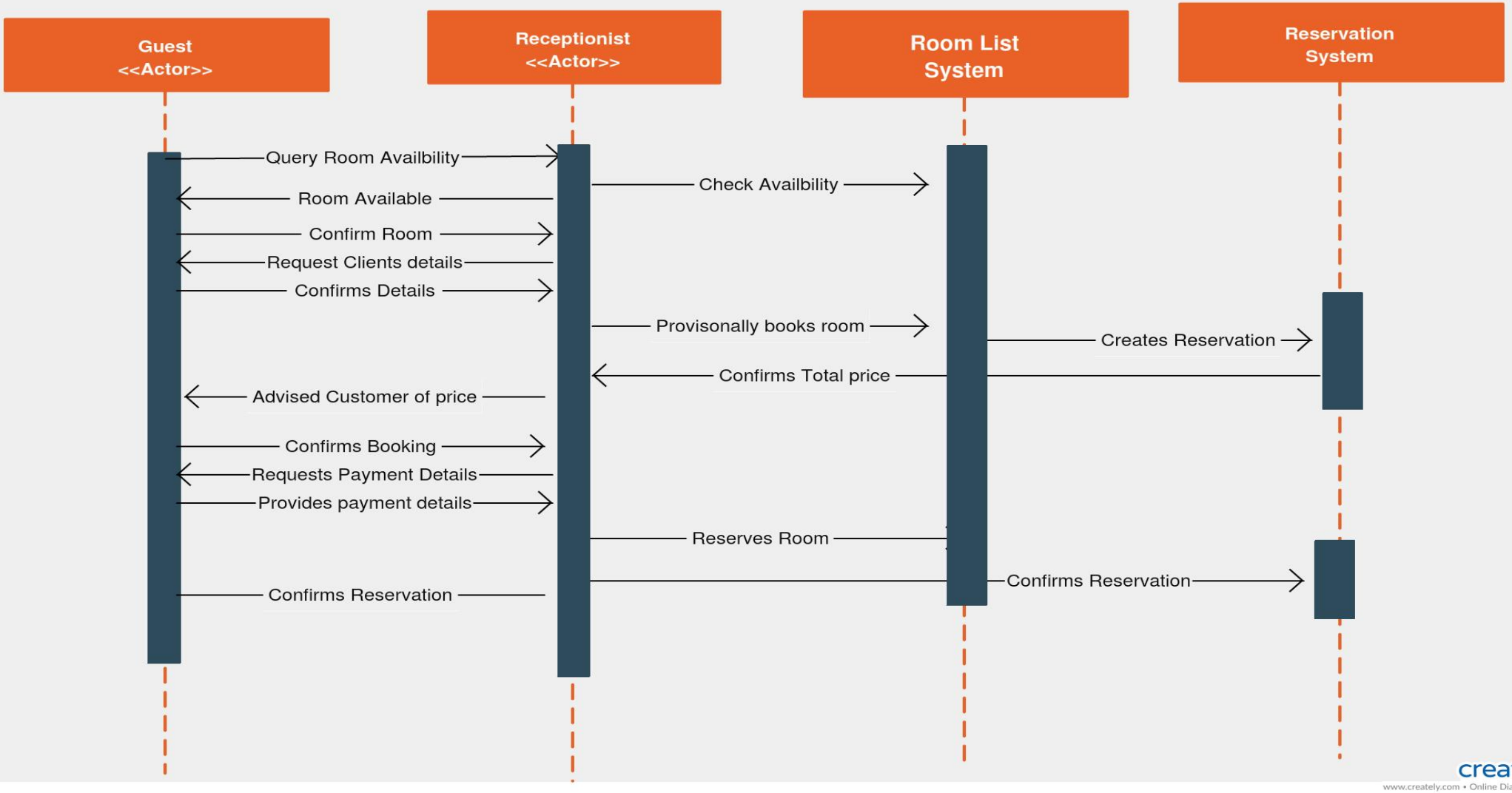
- Used during requirements analysis
  - To refine use case descriptions
  - to find additional objects (“participating objects”)
- Used during system design
  - to refine subsystem interfaces
- **Classes** are represented by columns
- **Messages** are represented by arrows
- **Activations** are represented by narrow rectangles
- **Lifelines** are represented by dashed lines

Drawing a Sequence Diagram




Login Screen
Validate User
Database
Student
click on login button
ValidateUser(userid,password)
userID:found
password:matched
classList(grades)
loginSuccessful:message
User login successful
ValidateUser(passwordMatched)
result
display:classList(grades)
text

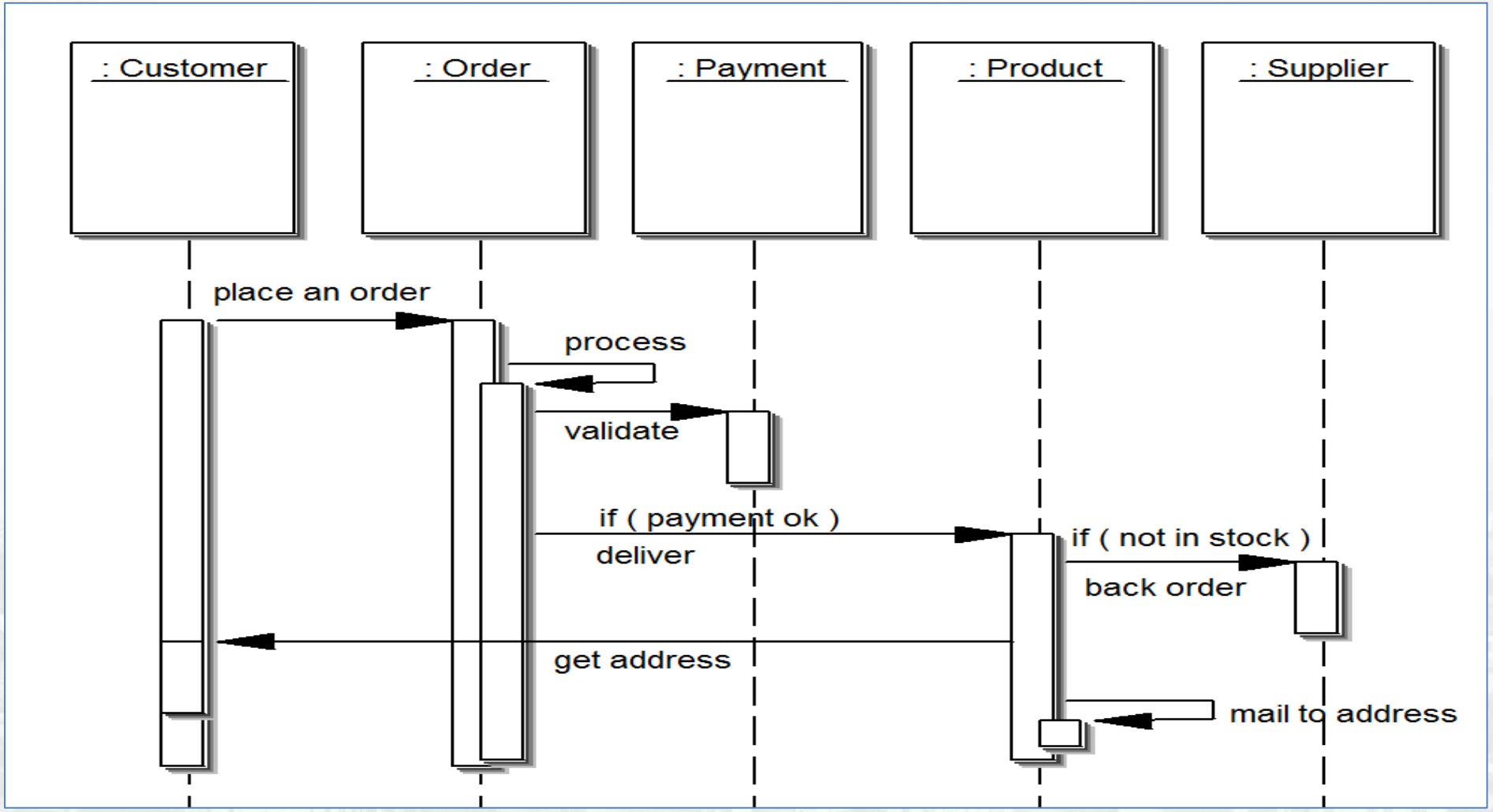
HOTEL RESERVATION SYSTEM



Sequence Diagram

- A sequence diagram is a good way to visualize and validate various runtime scenarios.
- These can help to predict how a system will behave and to discover responsibilities a class may need to have in the process of modeling a new system.

Order System



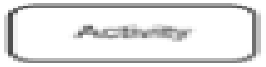
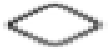
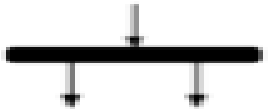
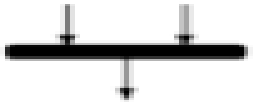
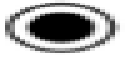
Activity Diagram

- It shows the structure of a process
- Supplements the use-case by providing a diagrammatic representation of procedural flow
- Shows flow of control from activity to activity
- Main Components of activity diagram
  - Action
  - Activity Node
  - Branching
  - Forking & Joining

Activity Diagram

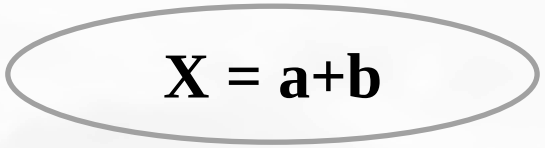
- Describes how activities are coordinated.
- Is particularly useful when you know that an operation has to achieve a number of different things, and you want to model what the essential dependencies between them are, before you decide in what order to do them.
- Records the dependencies between activities, such as which things can happen in parallel and what must be finished before something else can start.
- Represents the workflow of the process.

Activity diagram Notations

Initial Node 	A black circle is the standard notation for an initial state before an activity takes place. It can either stand alone or you can use a note to further elucidate the starting point.
Activity 	The activity symbols are the basic building blocks of an activity diagram and usually have a short description of the activity they represent.
Control Flow	Arrows represent the direction flow of the flow chart. The arrow points in the direction of progressing activities.
Branch 	A marker shaped like a diamond is the standard symbol for a decision. There are always at least two paths coming out of a decision and the condition text lets you know which options are mutually exclusive.
Fork 	A fork splits one activity flow into two concurrent activities
Join 	A join combines two concurrent activities back into a flow where only one activity is happening at a time.
	The final flow marker shows the ending point for a process in a flow. The difference between a final flow node and the end state node is that the latter represents the end of all flows in an activity.
 **Complete Activity Flow**	The black circle that looks like a selected radio button is the UML symbol for the end state of an activity. As shown in two examples above, notes can also be used to explain an end state.
Notes 	The shape used for notes.

Activity Diagram (Contd..)

- **Action:** Executable Computations

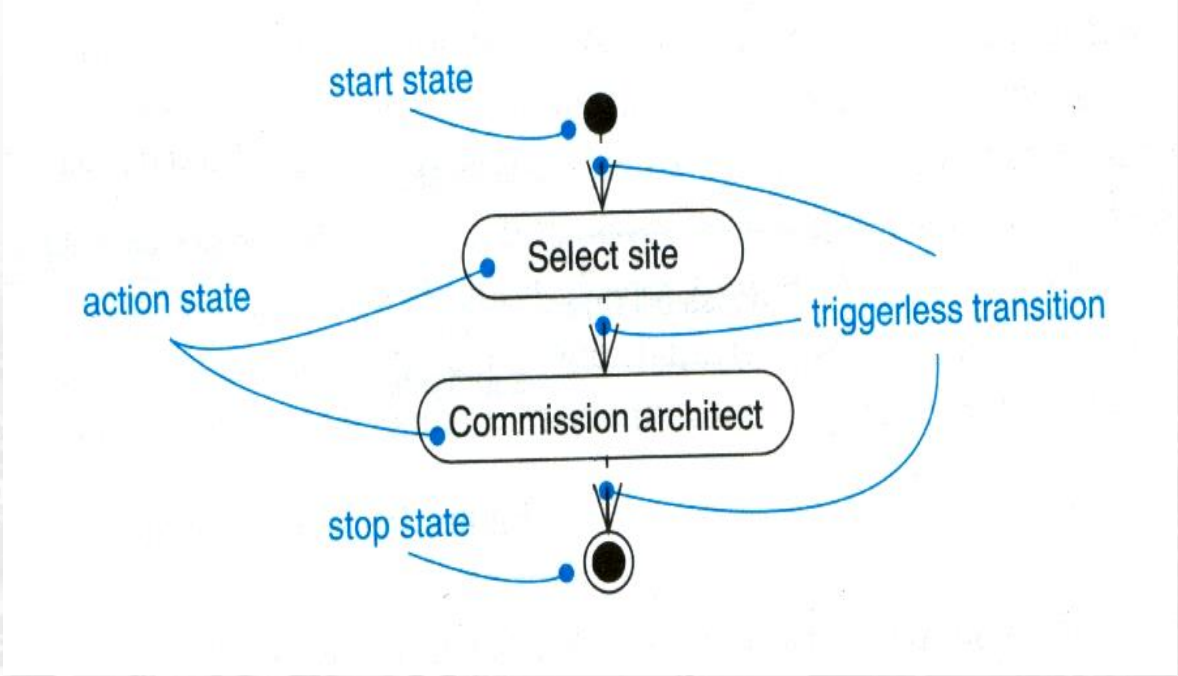


- **Activity Node:** Nested groupings of actions

- **Branching:** Specifies alternate paths taken based on some Boolean expression

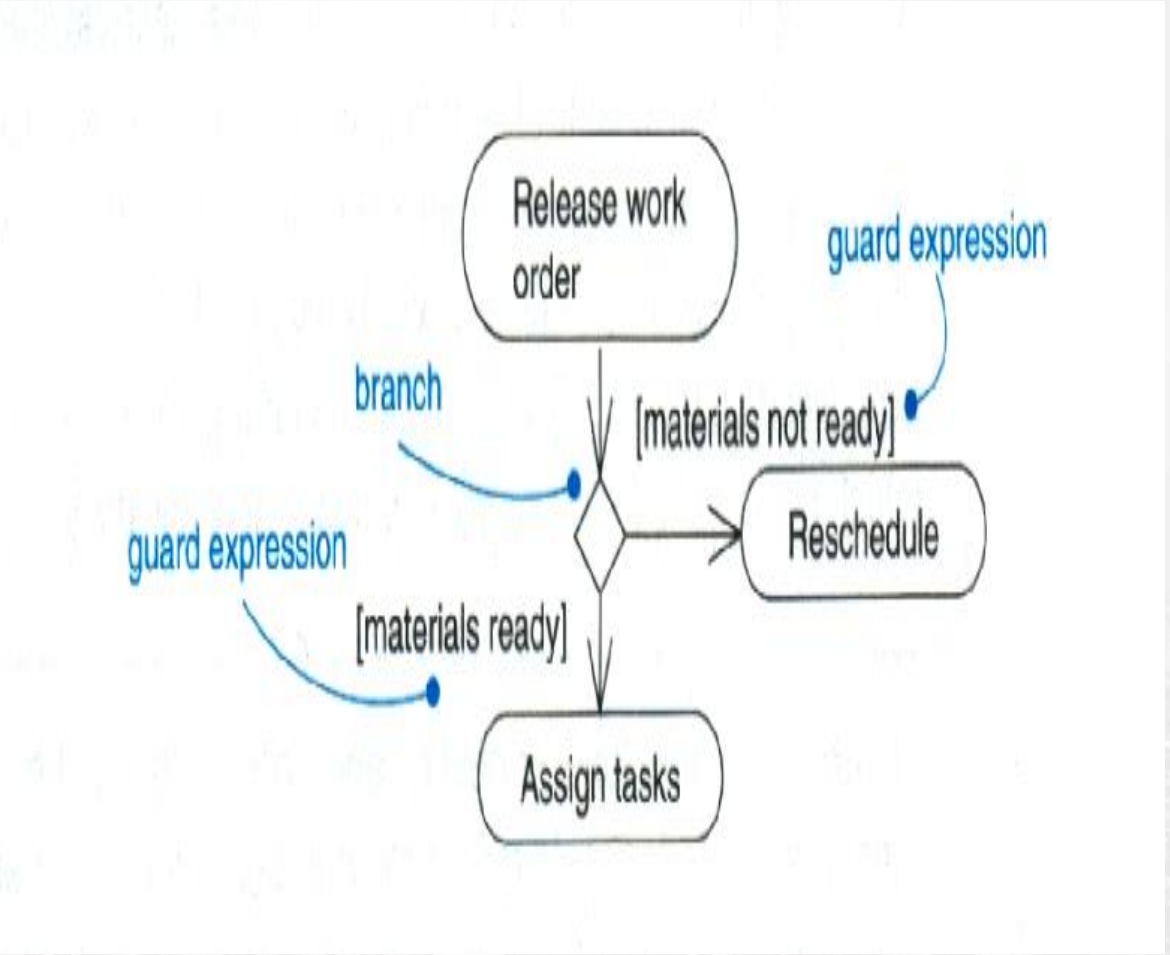
Transitions

- When the action or activity of a state completes, flow of control passes immediately to the next action or activity state
- A flow of control has to start and end someplace
  - initial state -- a solid ball
  - stop state -- a solid ball inside a circle



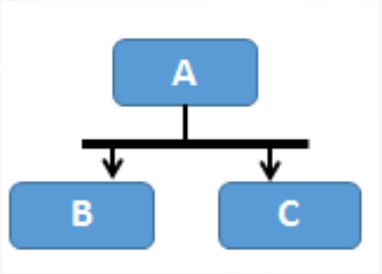
Branching

- A branch specifies alternate paths taken based on some Boolean expression
- A branch may have one incoming transition and two or more outgoing ones

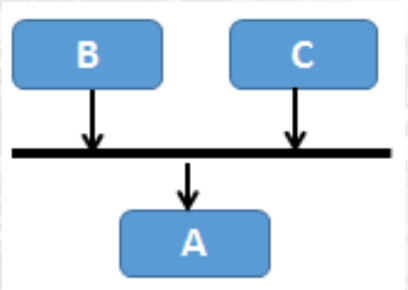


Activity Diagram (Contd..)

- **Forking & Joining:**
- **Fork:** Splitting of single flow of control into 2 or concurrent flow of control

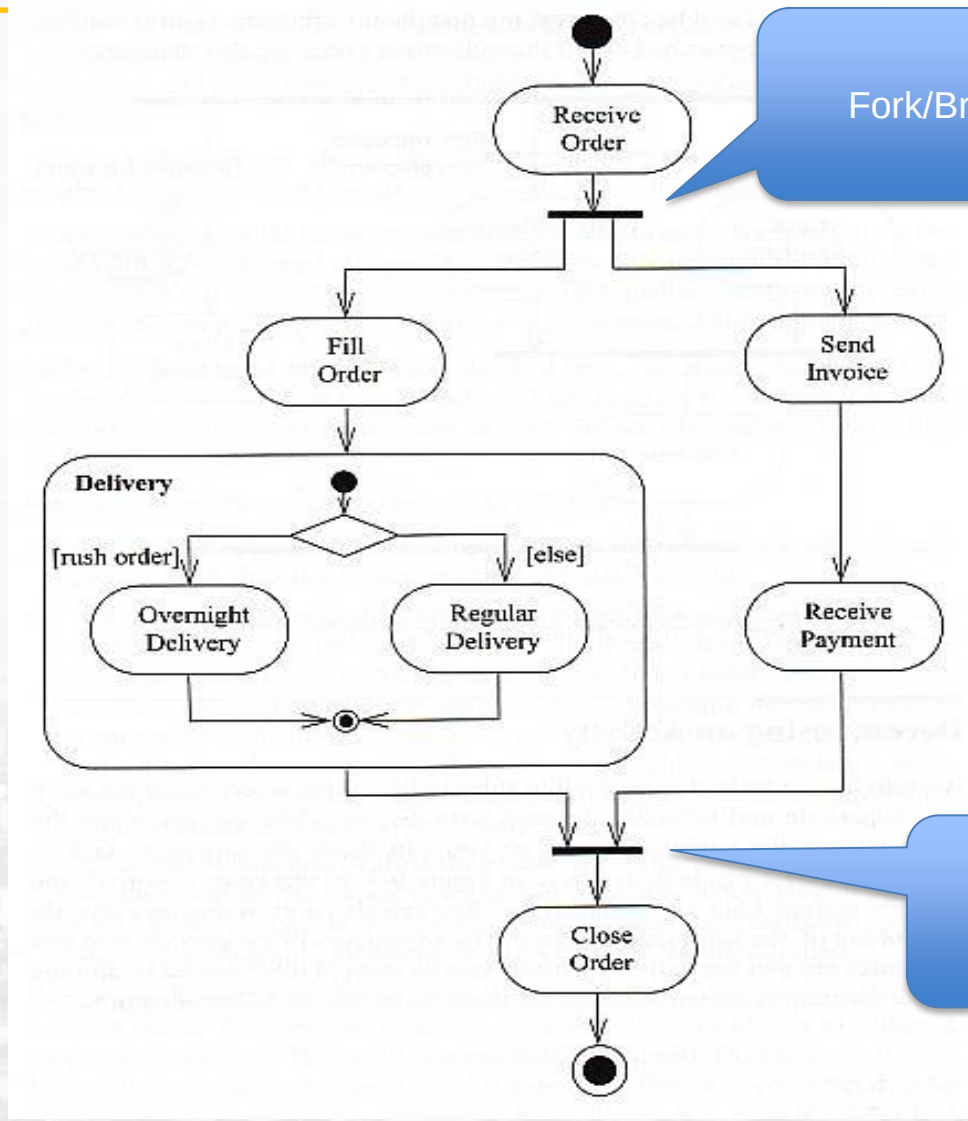


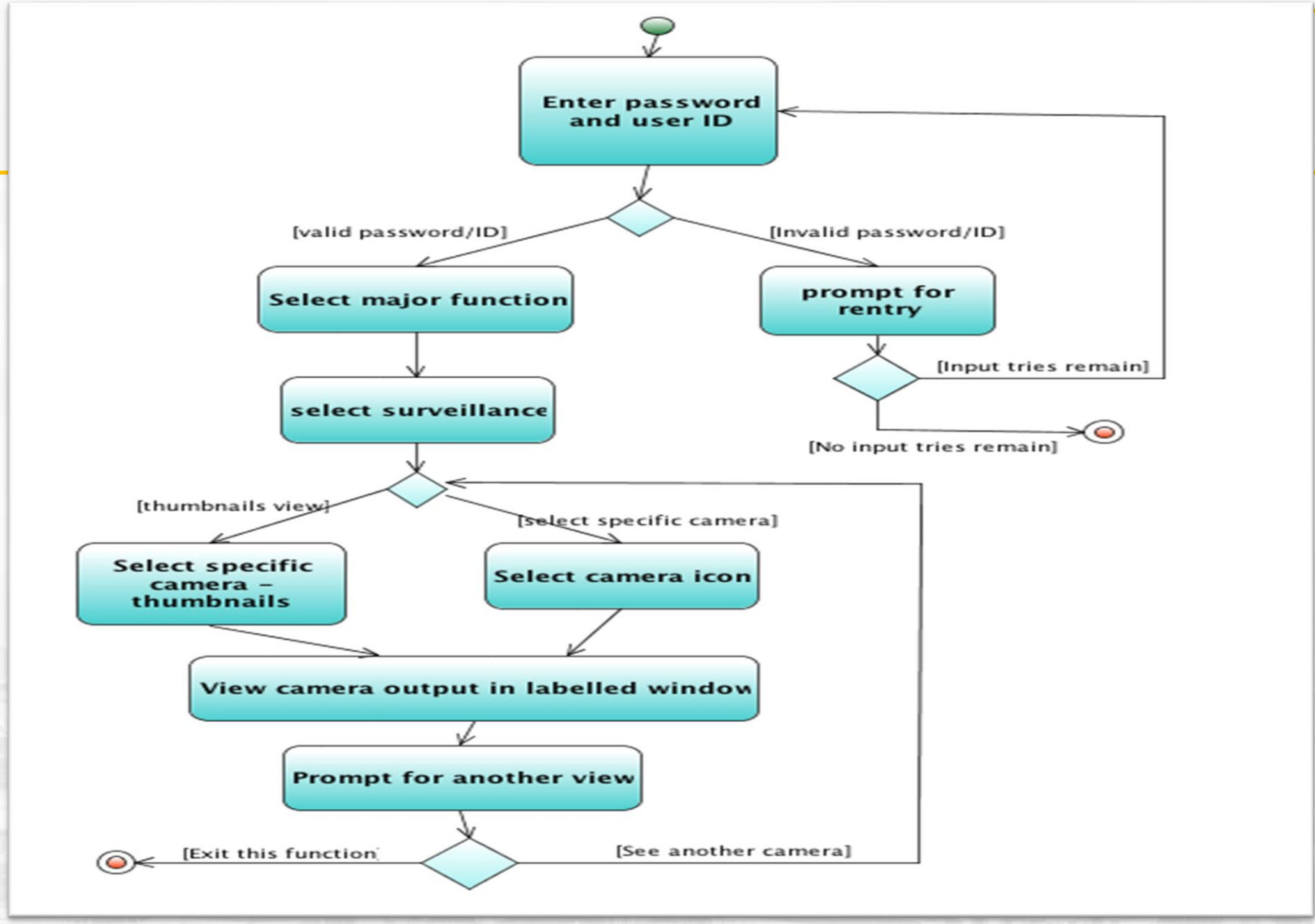
- **Join:** Synchronization of 2 or more concurrent flow of control



Activity Diagram Example

- To show concurrent activity, activity diagrams allow branches and joins.
- You can also reference or include other activity diagrams

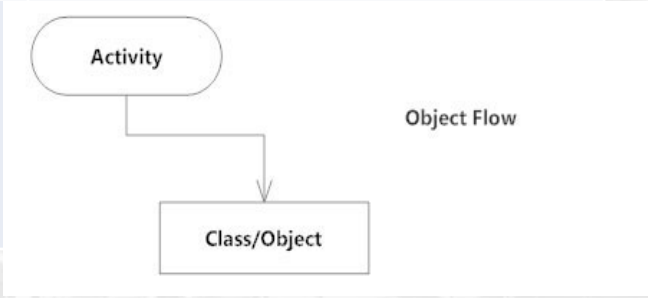



Enter password and user ID
[valid password/ID]
[Invalid password/ID]
Select major function
prompt for rentry
[Input tries remain]
select surveillance
[No input tries remain]
[thumbnails view]
[select specific camera]
Select specific camera – thumbnails
Select camera icon
View camera output in labelled window
Prompt for another view
[Exit this function]
[See another camera]

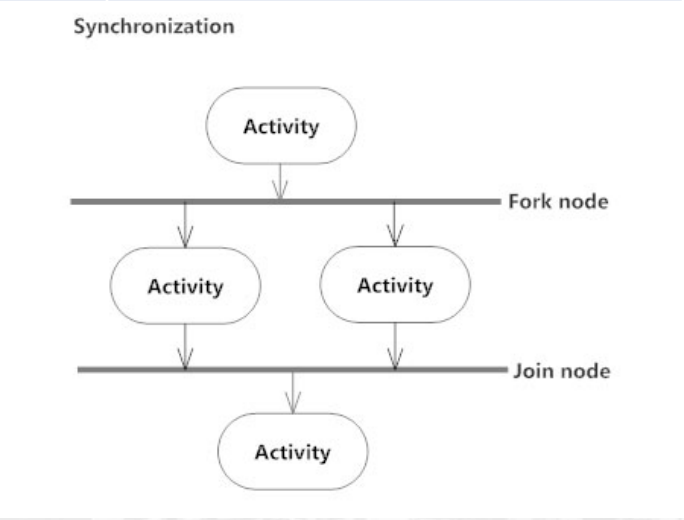
Activity Diagrams

- Normally employed in business process modelling.
- 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).

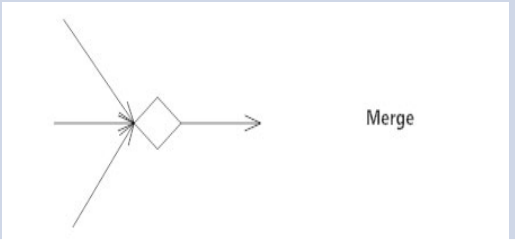
Activity Diagrams

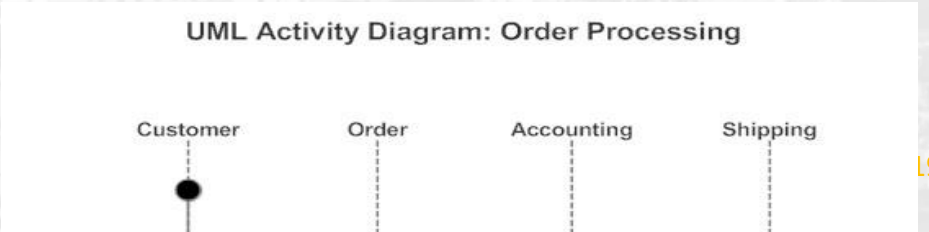
S.No	Name	Description	Notation
1	Initial State or Start Point	Represents the initial action state or the start point for any activity diagram	
2	Activity or Action State	An action state represents the non-interruptible action of objects.	
3	Action Flow	Action flows, also called edges and paths, illustrate the transitions from one action state to another	
4	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.	

Activity Diagrams

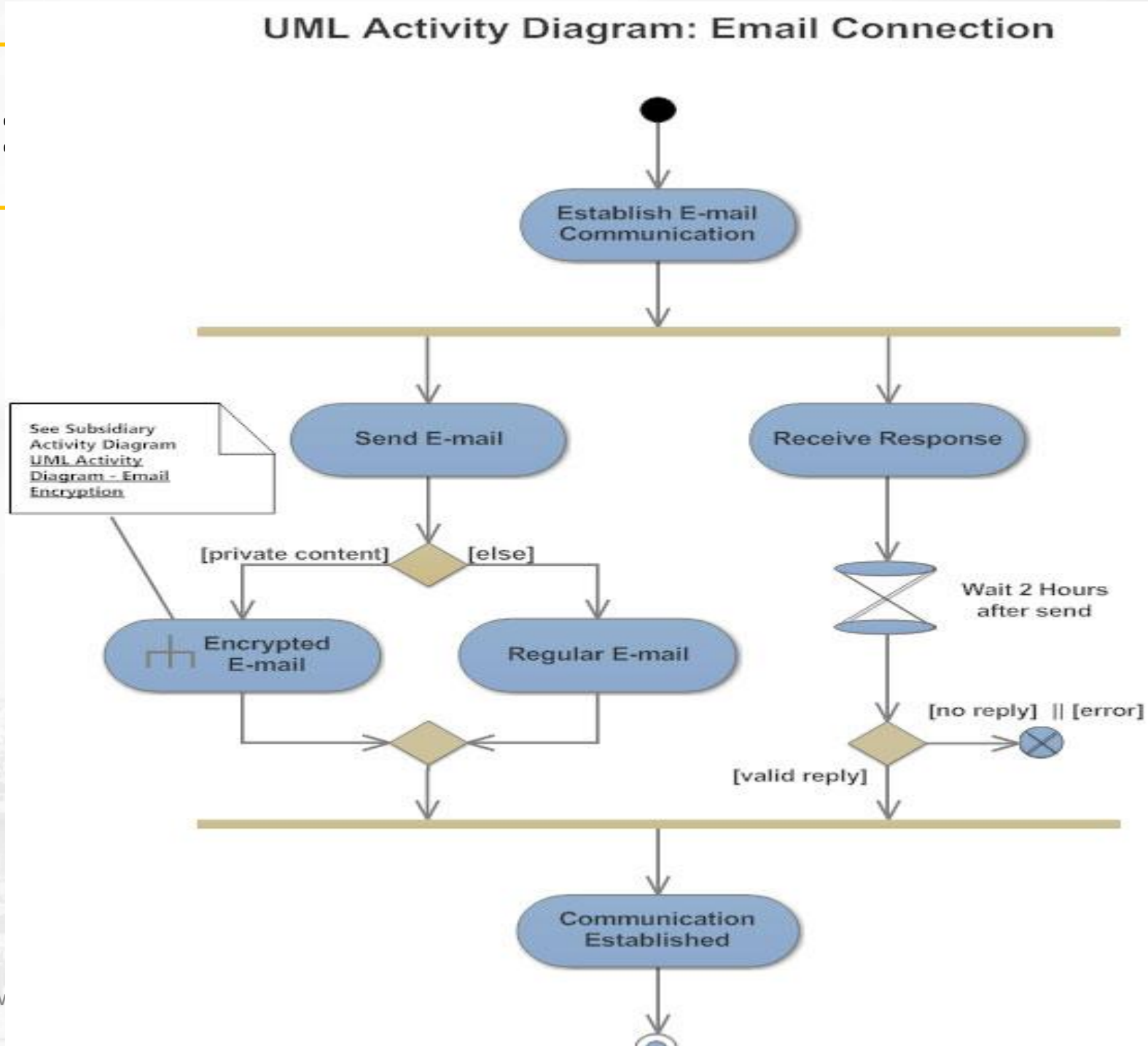
S.No	Name	Description	Notation
5	Decisions and Branching	When an activity requires a decision prior to moving on to the next activity, add a diamond between the two activities.	
6	Guards	In UML, guards are a statement written next to a decision diamond that must be true before moving next to the next activity.	
7	Synchronization	A fork node is used to split a single incoming flow into multiple concurrent flows. A join node joins multiple concurrent flows back into a single outgoing flow.	

Activity Diagrams

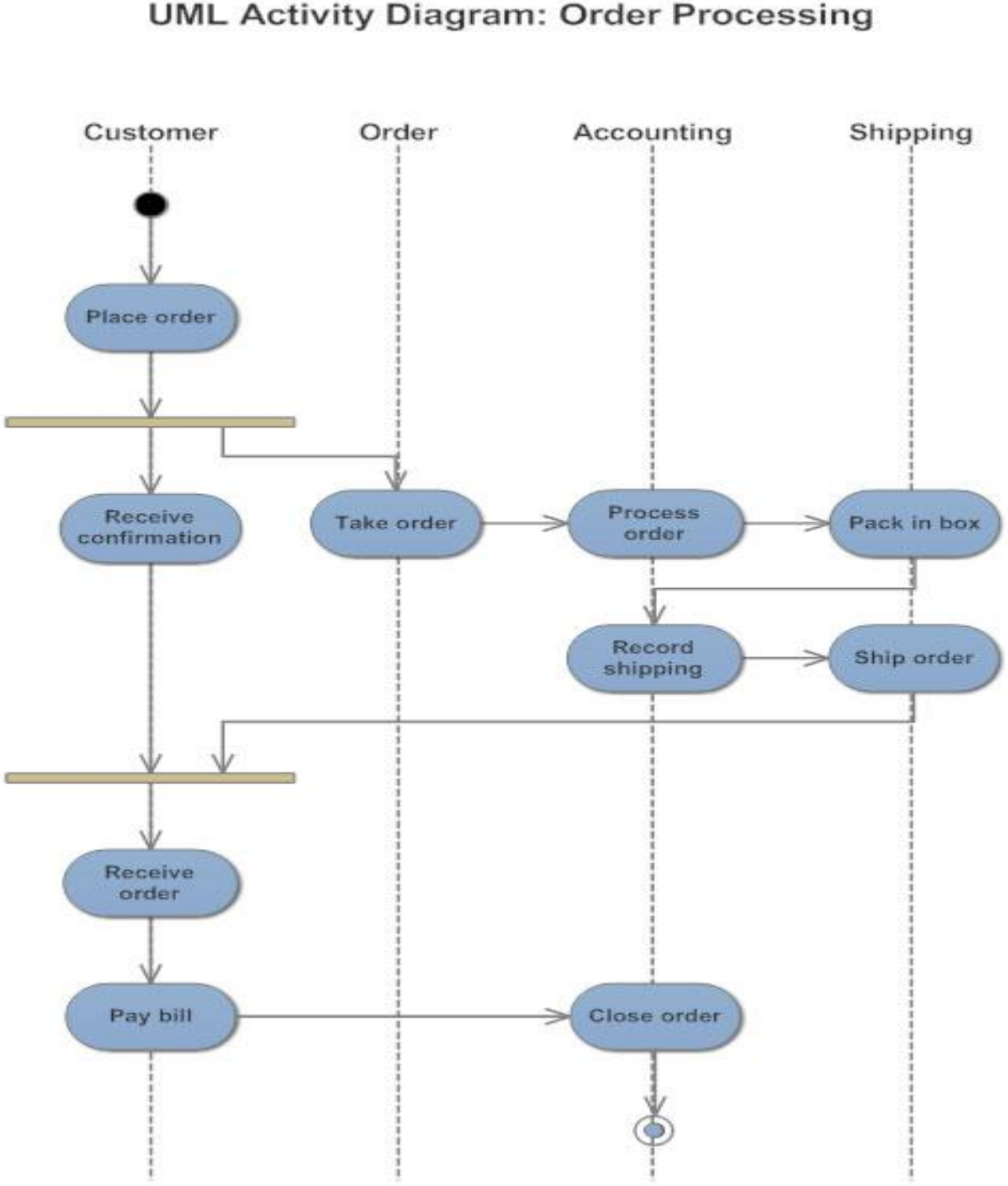
S.No	Name	Description	Notation
8	Merge Event	A merge event brings together multiple flows that are not concurrent.	
9	Final State or End Point	An arrow pointing to a filled circle nested inside another circle represents the final action state	
10	Swimlanes	Swimlanes group related activities into one column.	



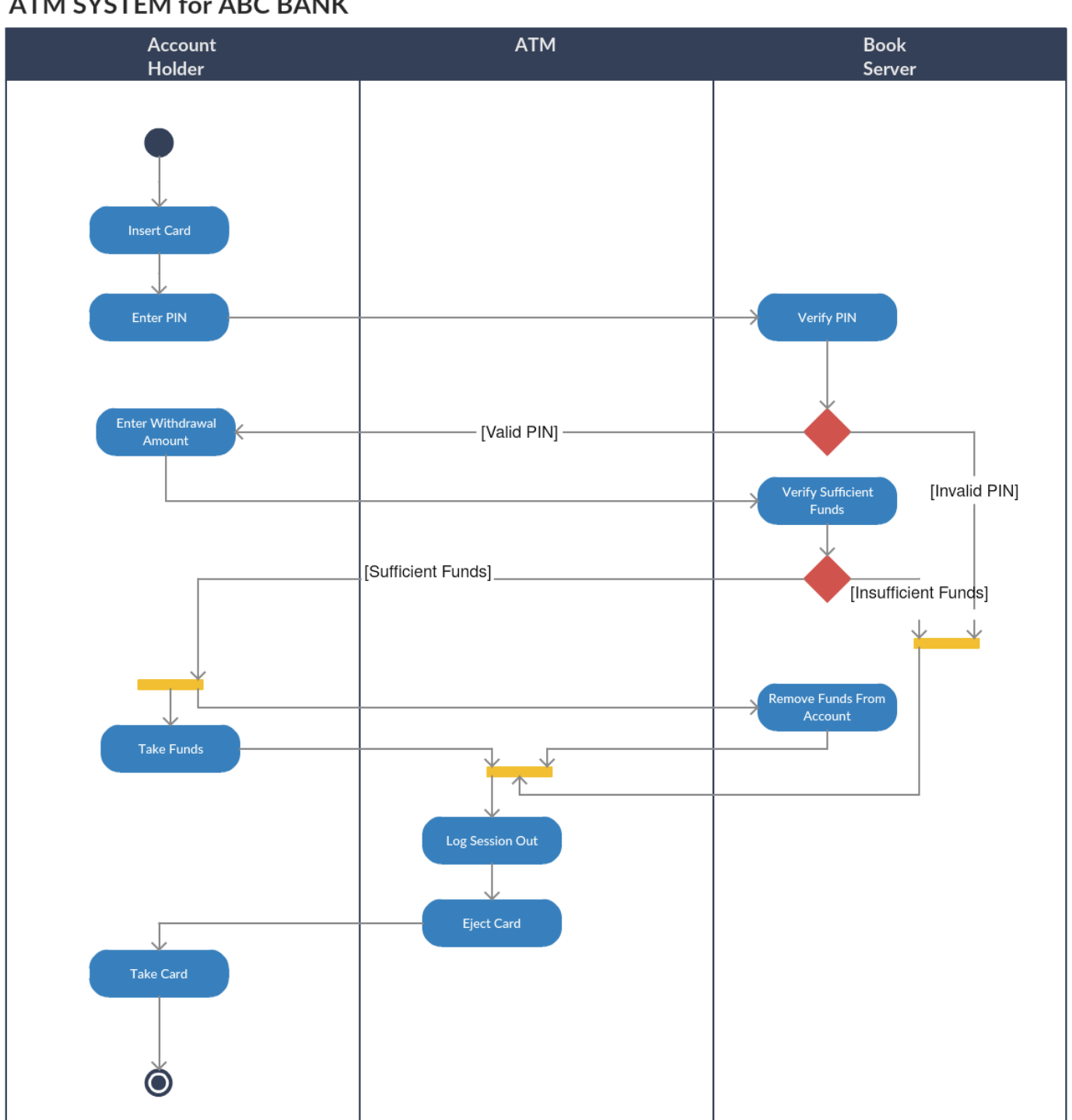
Example 1 :


UML Activity Diagram: Email Connection
Establish E-mail Communication
Send E-mail
Receive Response
See Subsidiary Activity Diagram UML Activity Diagram - Email Encryption
[private content]
[else]
Encrypted E-mail
Regular E-mail
Wait 2 Hours after send
[no reply] || [error]
[valid reply]
Communication Established

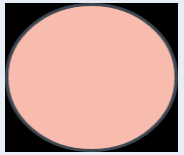
Example 2 : Food Ordering Processing with Swimlanes



Example 3: ABC Bank with swimlanes



Revision : Symbols of CFD

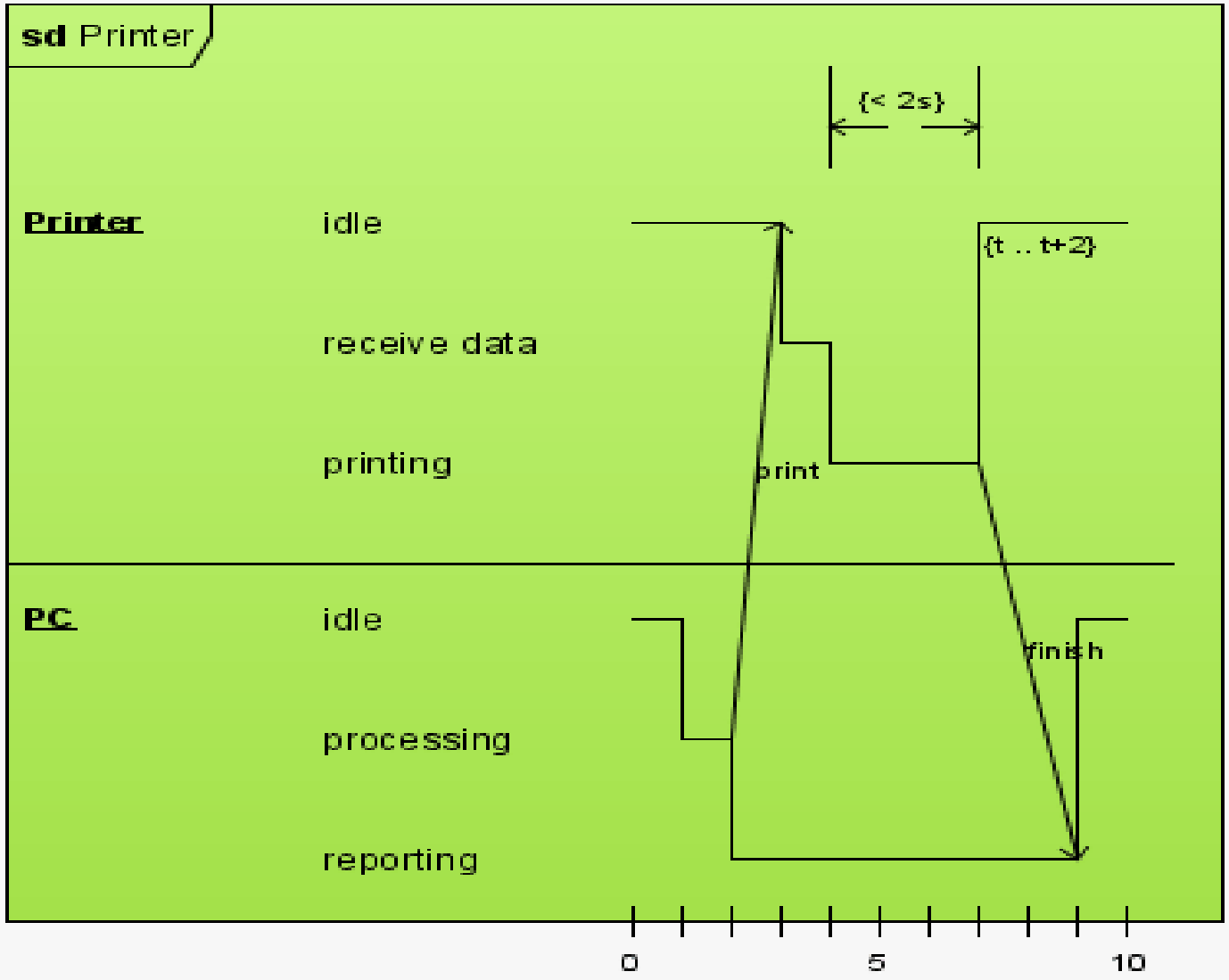
S.No	Name	Description	Notation Yourdon and Coad
1	**External Entity**	An outside system that sends or receives data, communicating with the system being diagrammed. They are the sources and destinations of information entering or leaving the system. They might be an outside organization or person, a computer system or a business system. They are also known as terminators, sources and sinks or actors. They are typically drawn on the edges of the diagram.	
2	**Process**	Any process that changes the data, producing an output. It might perform computations, or sort data based on logic, or direct the data flow based on business rules	
3	**Data store**	Files or repositories that hold information for later use, such as a database table or a membership form.	
4	**Data flow**	The route that data takes between the external entities, processes and data stores.	

Timing diagrams

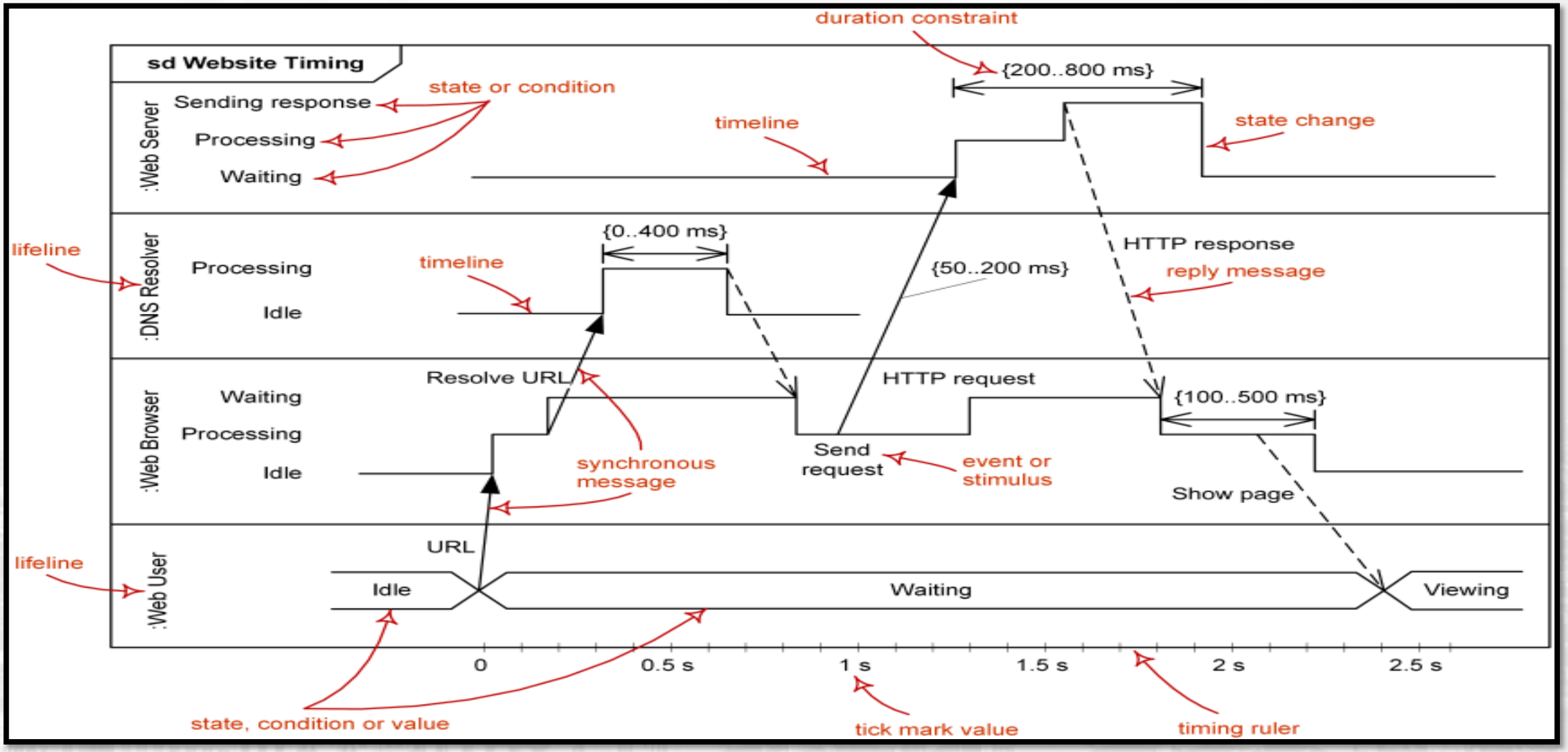
- UML 2 Timing diagrams shows the behavior of the objects in a given period of time. Timing diagram is a special form of a sequence diagram.
- The differences between timing diagram and sequence diagram are that the axes are reversed so that the time are increase from left to right and the lifelines are shown in separate compartments arranged vertically.
- Timing diagrams focus on conditions changing within and among lifelines along a linear time axis.

Timing diagram

- The timing diagram focusing attention on time of occurrence of events causing changes in the modeled conditions of the Lifelines.



Timing diagrams

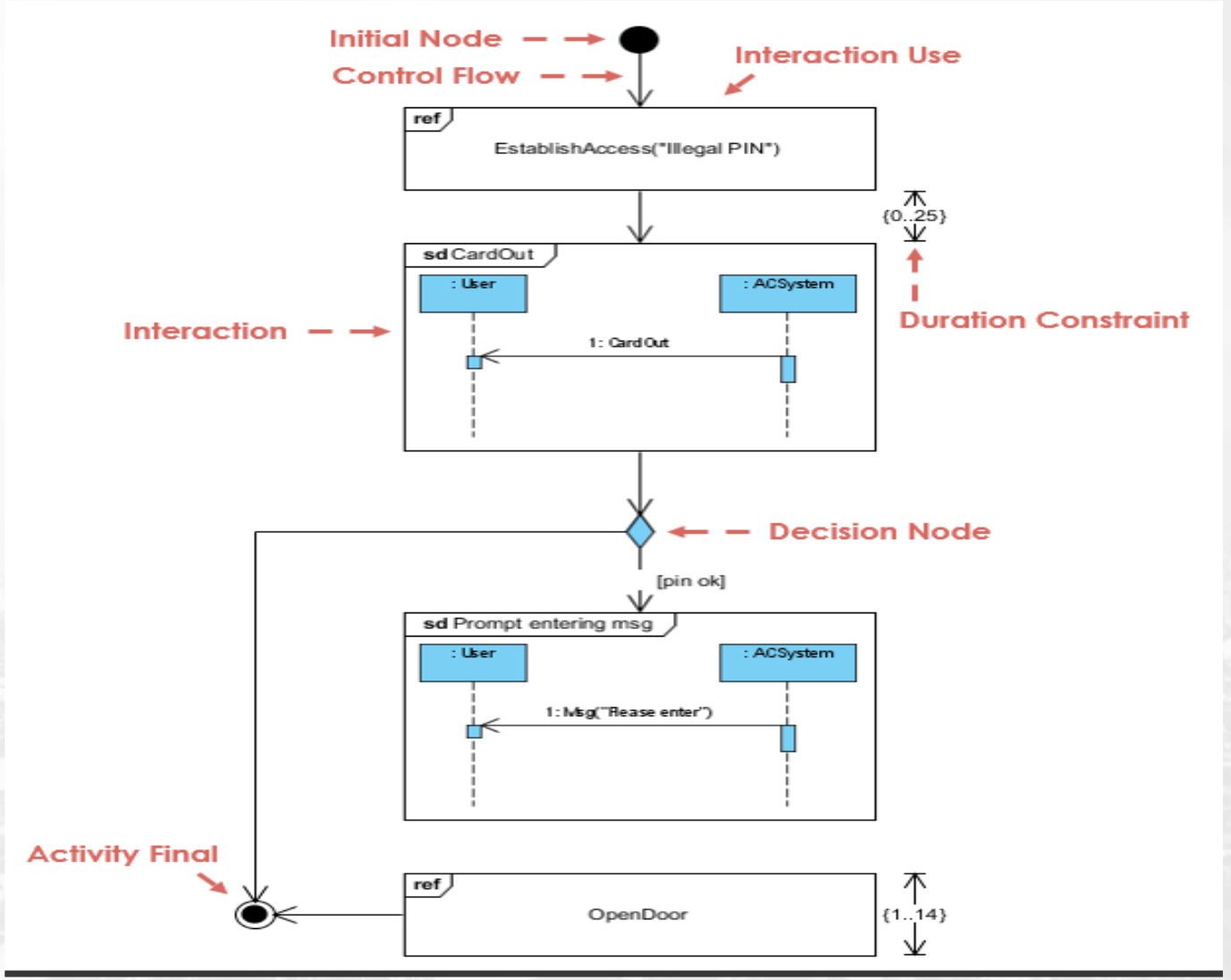


Interaction overview diagram

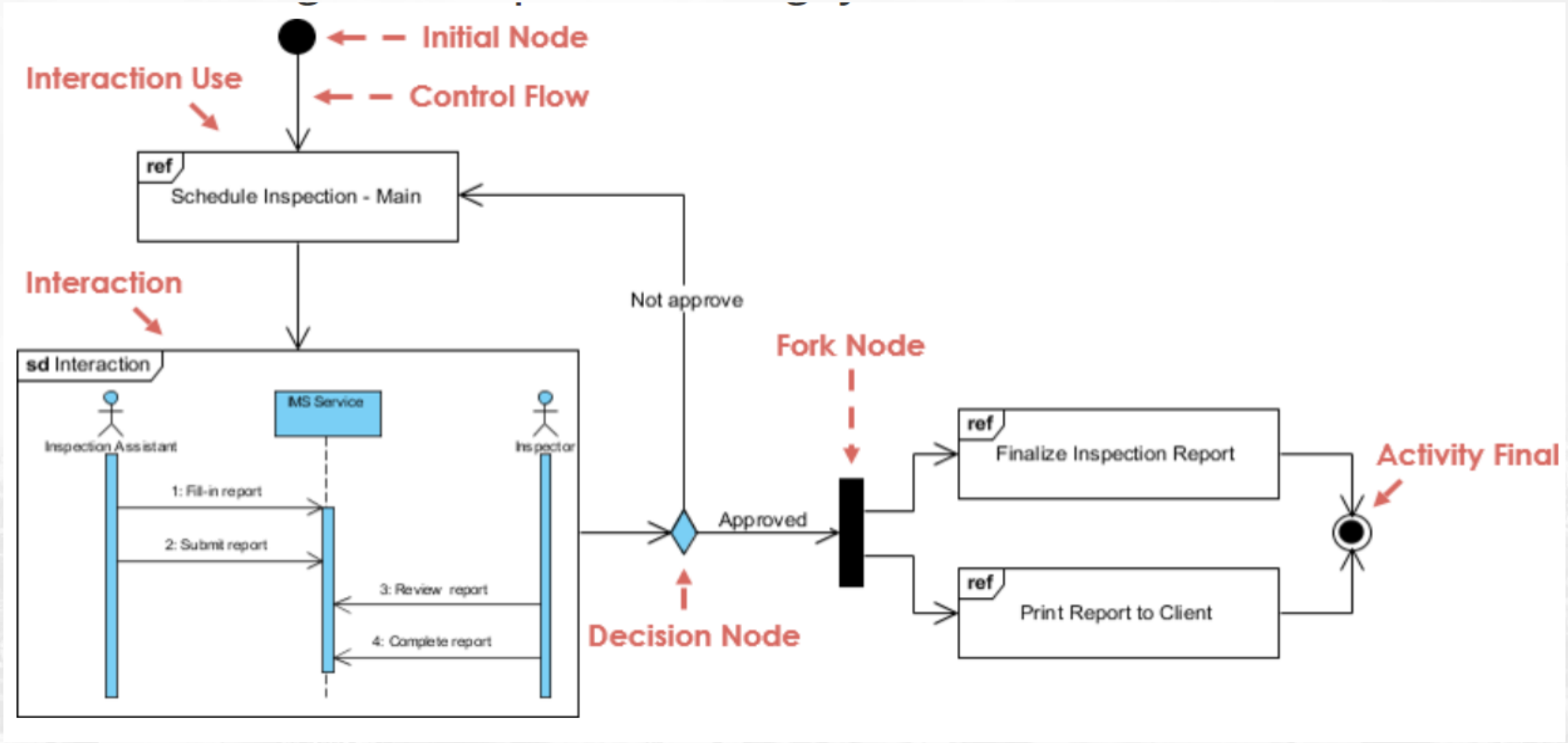
- Interaction overview diagrams focuses on the overview of the flow of control of the interactions.
- It is a variant of the Activity Diagram where the nodes are the interactions or interaction occurrences.
- It describes the interactions where messages and lifelines are hidden.

Interaction overview diagram

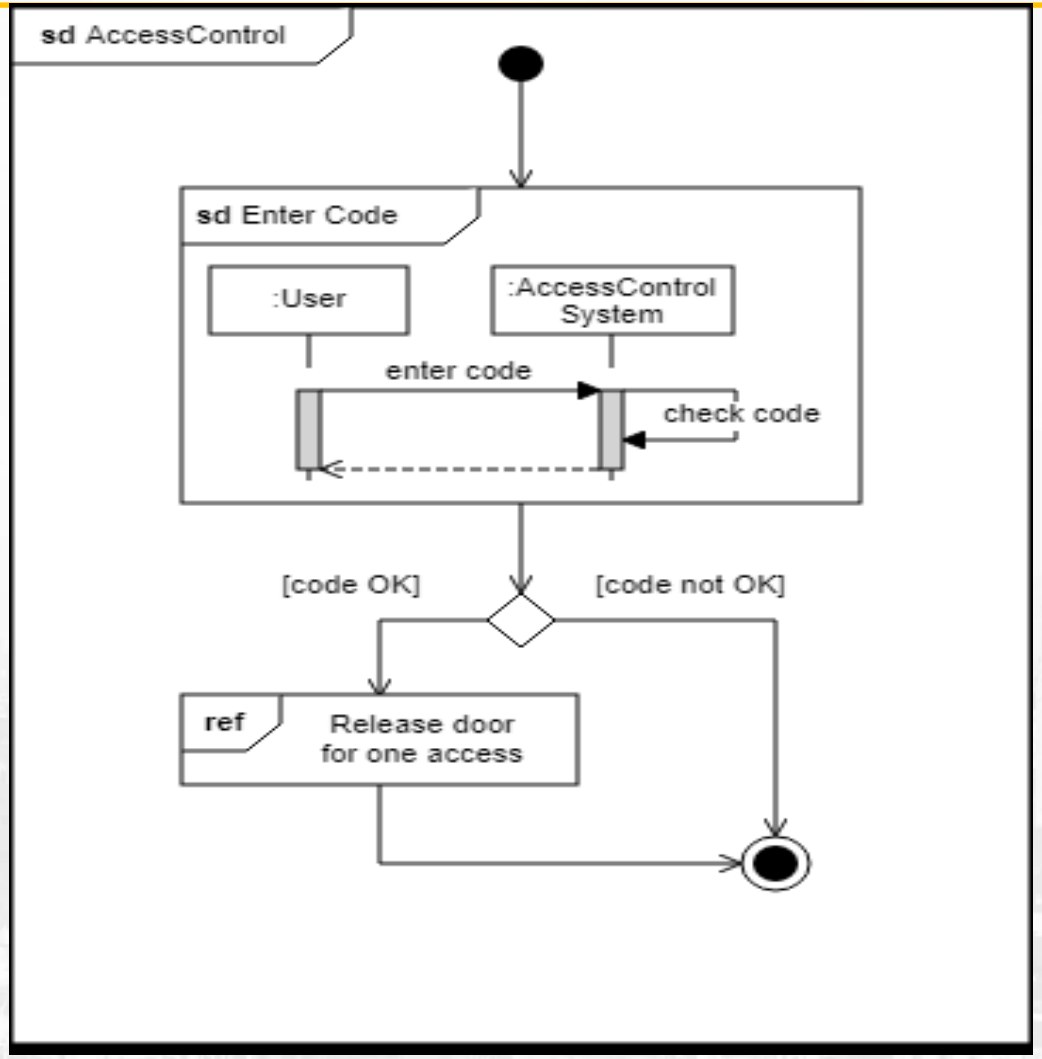
- The Interaction EstablishAccess occurs first with argument"Illegal PIN" followed by an interaction with the message CardOut which is shown in an inline Interaction.
- Then there is an alternative as we find a decision node with an InteractionConstraint on one of the branches.
- Along that control flow we find another inline Interaction and an InteractionUse in the sequence.



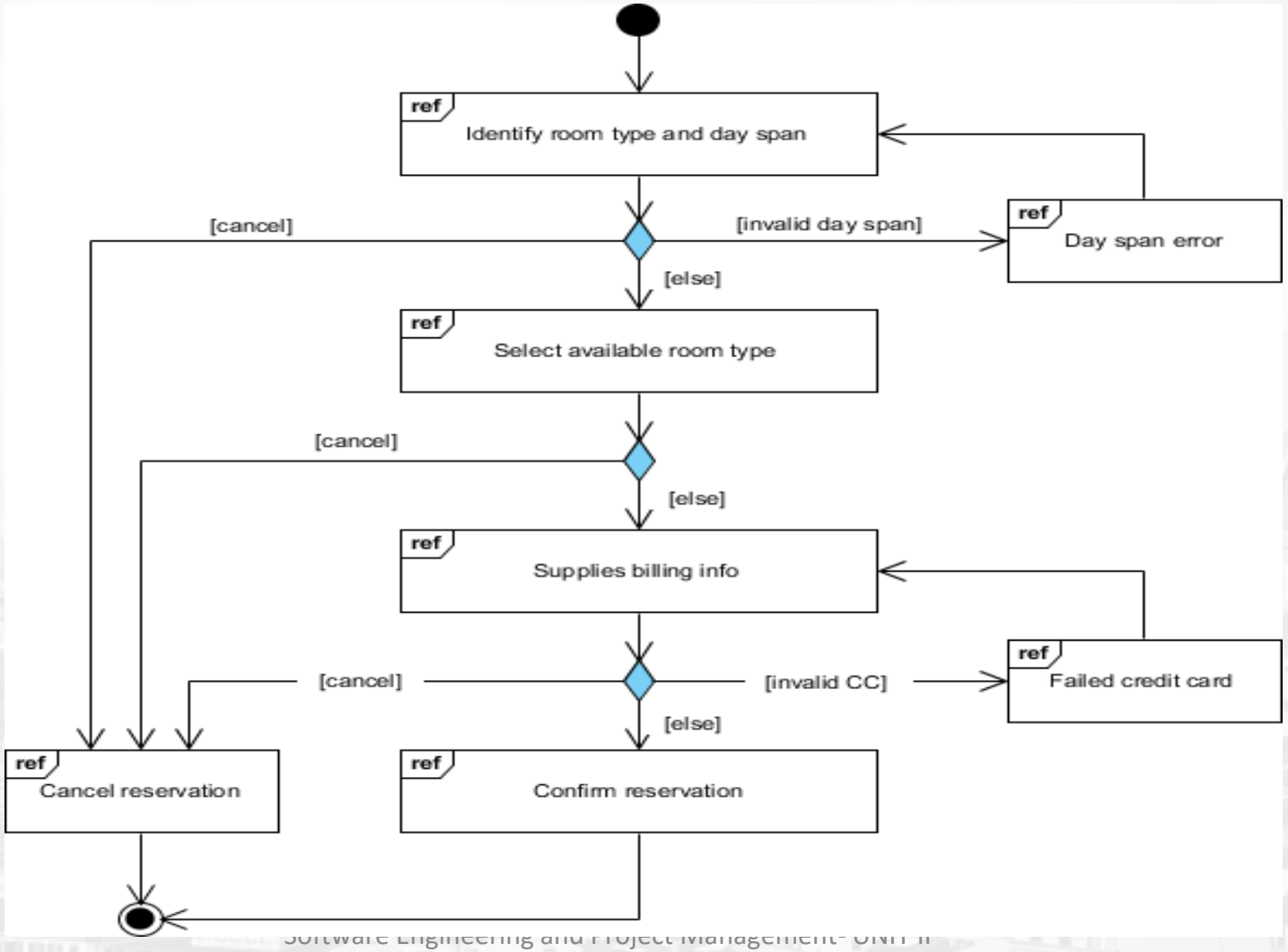
Interaction overview diagram- Scheduling System



Interaction overview diagram



Interaction overview diagram



Communication diagrams

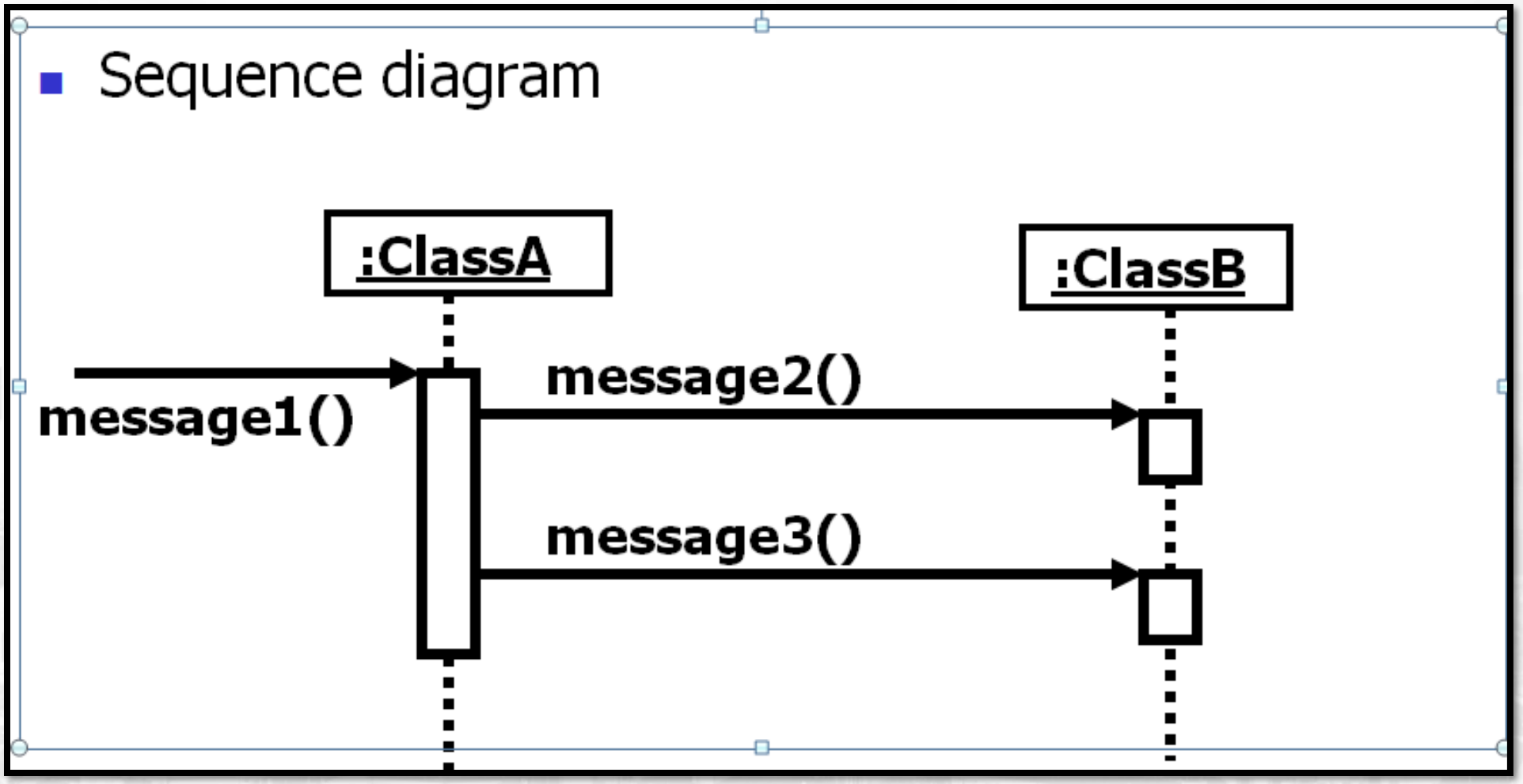
•UML Communication diagrams used to model the dynamic.

•When compare to Sequence Diagram, the Communication Diagram is more focused on showing the **collaboration of objects rather than the time sequence.**

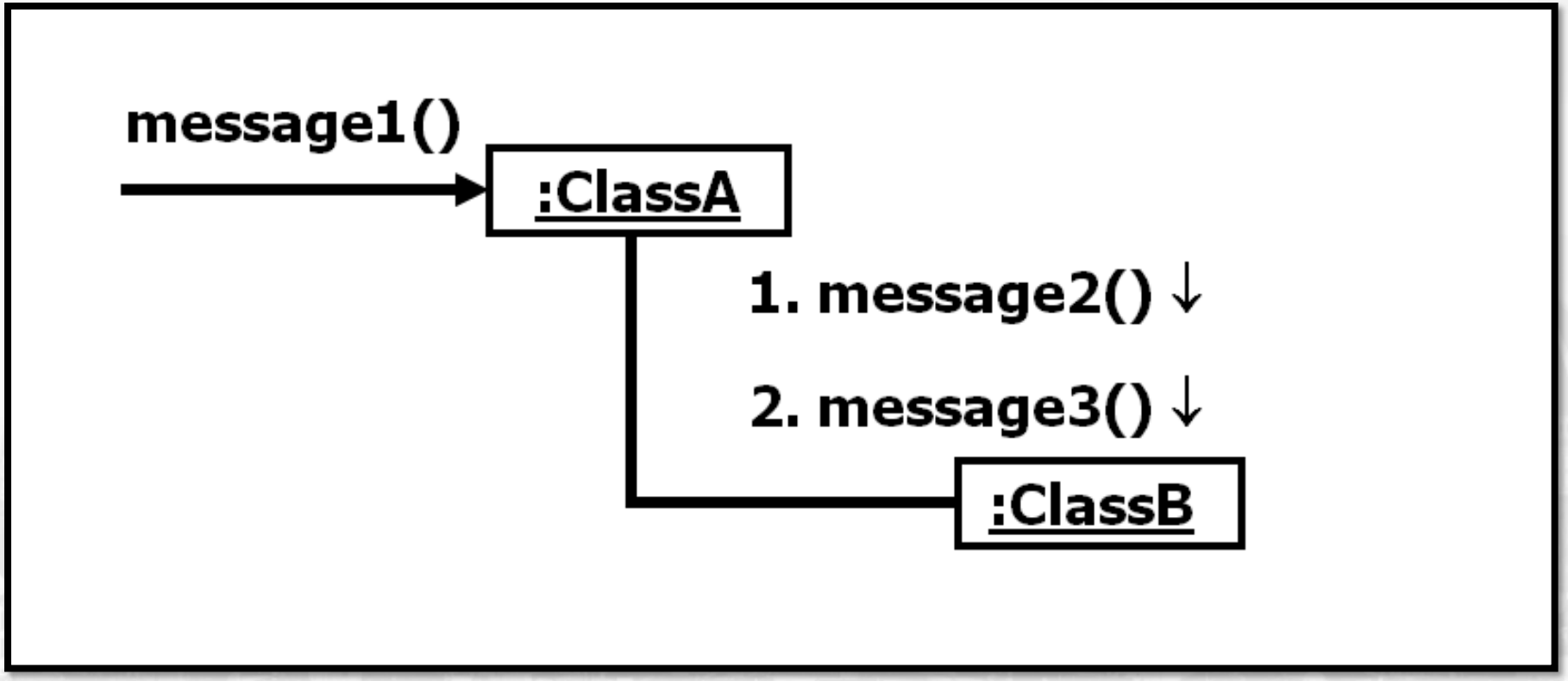
•Components of the **communication process** include a sender, encoding of a message, selecting of a channel of **communication**, receipt of the message by the receiver and decoding of the message.

•It show the network and sequence of messages or communications between objects at run-time during a collaboration instance

Sequence Diagrams vs Communication Diagrams



Sequence Diagrams vs Communication Diagrams



Sequence Diagrams vs Communication Diagrams

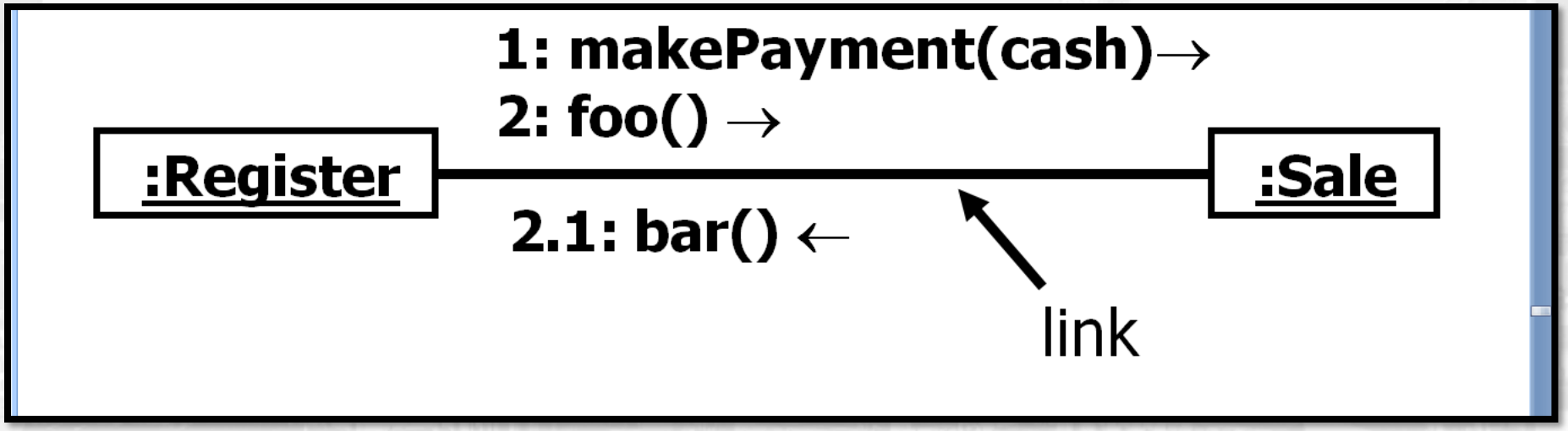
- **Sequence diagrams :**
  - Strength: clearly show sequence or time ordering of events, simple notation
  - Weakness: forced to extend to the right when adding new objects
- **Communication diagrams :**
  - Strength: space economical flexibility to add new objects in two dimensions, better to illustrate complex branching, iteration and concurrent behavior
  - Weakness: difficult to see sequence of messages, more complex notation

CommunicationDiagrams

- **Communication diagrams contain**
  - Classes
  - Associations
  - Message exchanges within a collaboration

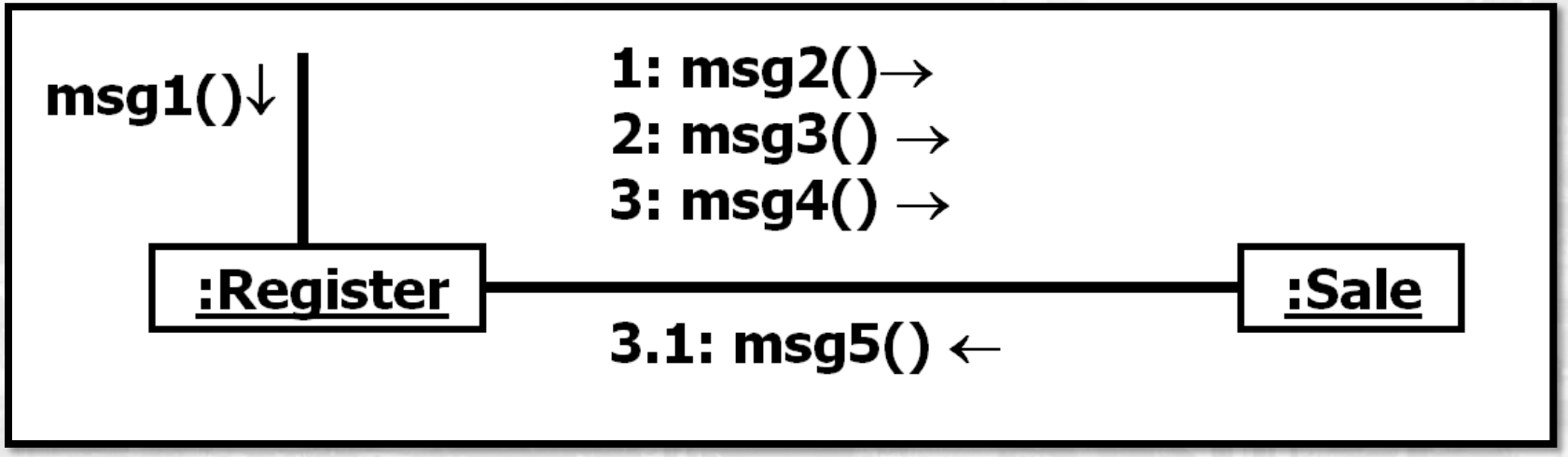
Communication Diagrams

- Link : A link is a connection path between two objects, it indicates some form of navigation and visibility between the objects. A link is an instance of an association.
- Note that multiple messages, and messages both ways can be exchanged along the same link.

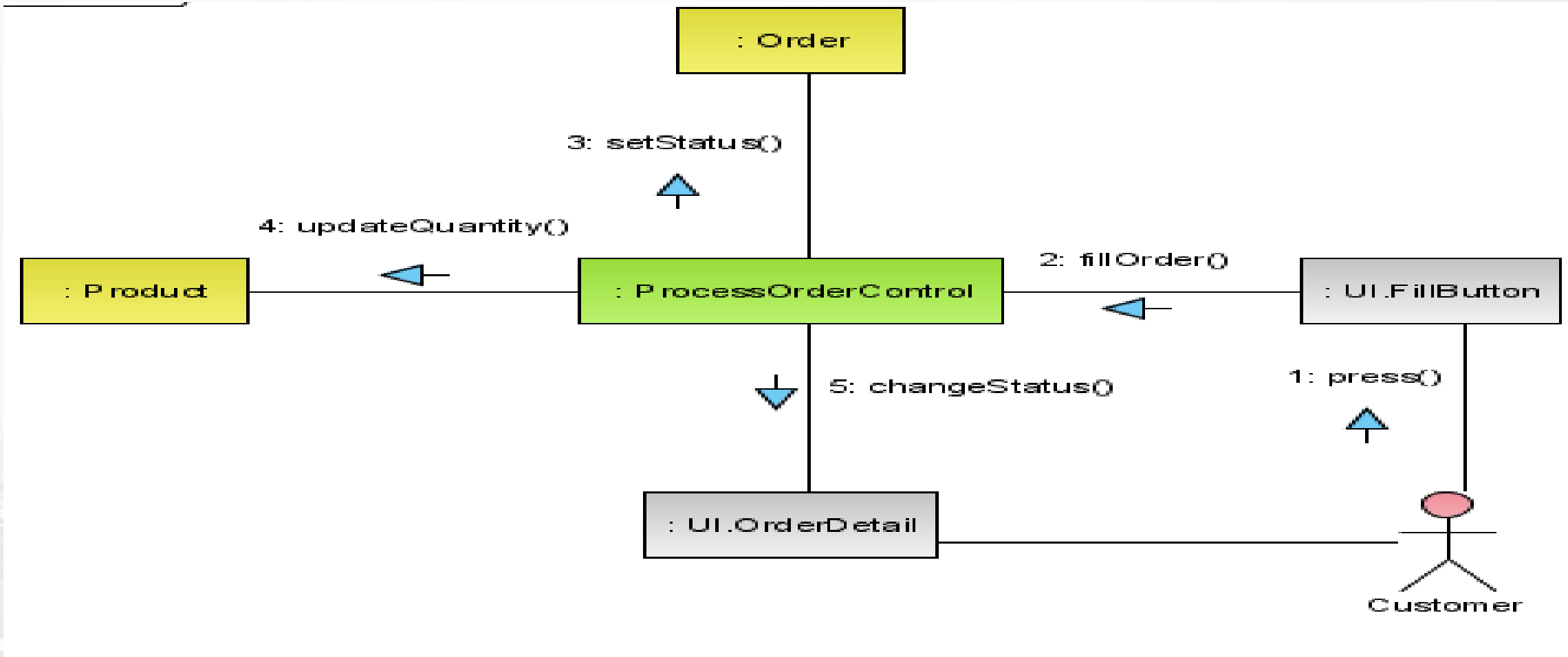


Communication Diagrams

- Messages : each message between objects is represented with a message expression and a small arrow indicating the direction of the message. A sequence number is added to show the sequential order of messages.



Process Order Control Communication diagram



Communication vs. Sequence Diagrams

	Communication Diagrams	Sequence Diagrams
Participants	✓	✓
Links	✓	
Message Signature	✓	✓
Parallel Messages	✓	✓
Asynchronous messages		✓
Message Ordering		✓
Create & Maintain	✓	

State Machine Diagrams…
Terms and Concepts

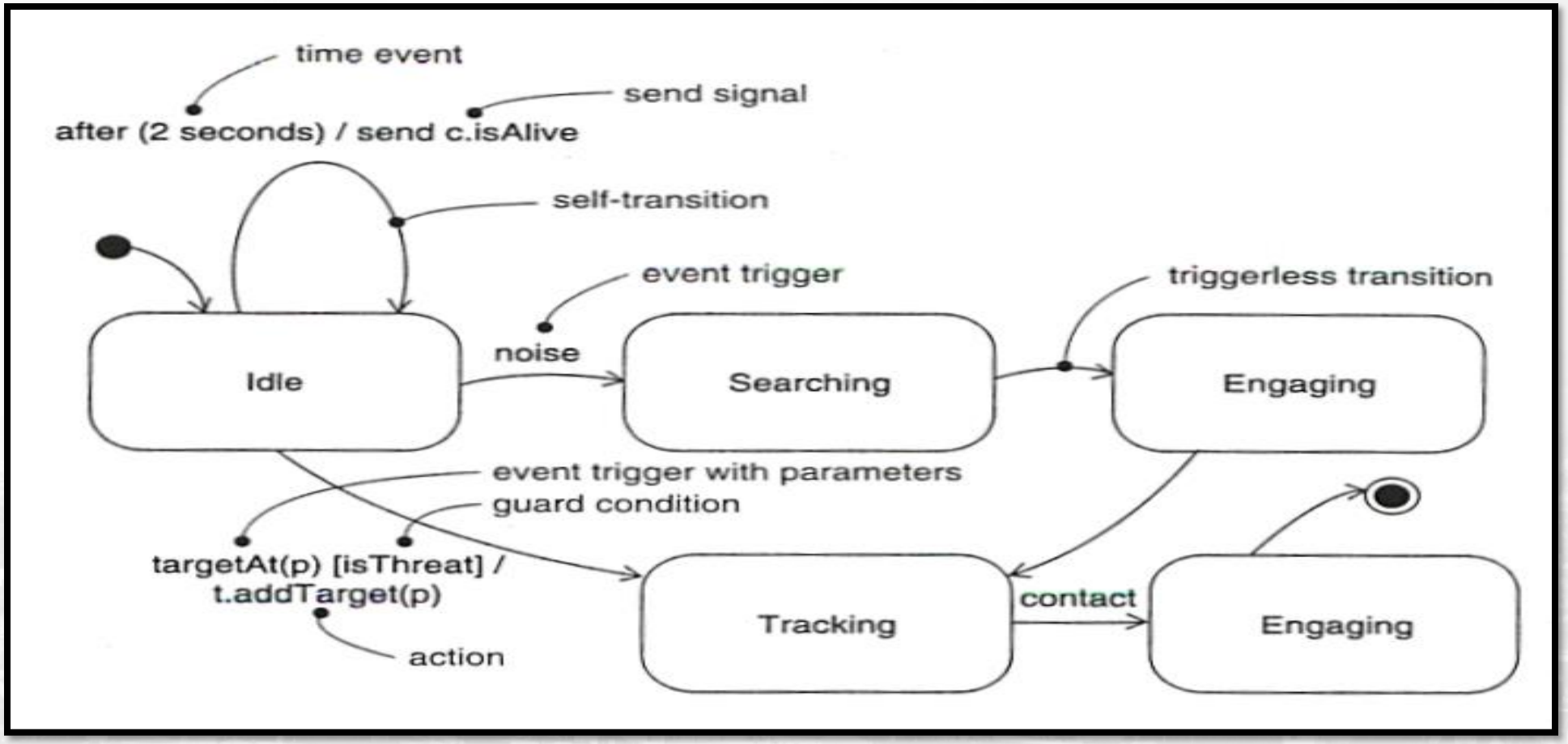
- *State*
  - A state is a condition or situation during the life of an object in which it satisfies some condition, performs some activity, or waits for some event.
  - A state may include …
    - Name
    - Entry/exit actions
    - Internal transitions
    - Activities
    - Substates - may sequential or concurrent
    - Deferred events (infrequently used)

Terms and Concepts

- ***State***
  - Special categories of states
    - *Initial state* - indicates the initial starting state for for the state machine or a substate.
    - *Final state* - indicates the state machine's execution has completed.
      - Real-time state machines frequently do not include a final state.
    - Neither initial or final states contain any of the parts found in traditional states.
- ***Transition***
  - A directed relationship between two states.
  - A flow of control through a statechart diagram.
  - **Contains five parts**
    - Source state - current state before transition fires.
    - Event trigger - external stimulus that has the potential to cause a transition to fire.
    - Guard condition - a condition that must be satisfied before a transition can fire.
    - Action - an executable atomic computation.
    - Target state - new state after transition fires.

State Machine Diagrams

Example of Transition of states



Terms and Concepts

- **Advanced States and Transitions**
  - **Entry action** - Upon each entry to a state, a specified action is automatically executed.
    - Syntax (to be placed inside the state symbol): **entry / action**
  - **Exit action** - Just prior to leaving a state, a specified action is automatically executed.
    - Syntax (to be placed inside the state symbol): **exit / action**
  - **Internal Transitions** - The handling of an event without leaving the current state.
    - Used to avoid a states entry and exit actions.
    - Syntax (to be placed inside the state symbol): **event / action**
  - **Activities -** Ongoing work that an object performs while in a particular state. The work automatically terminates when the state is exited.
    - Syntax (to be placed inside the state symbol): **do / activity**
    - Activities are ongoing operations; actions are instantaneous operations.

Terms and Concepts

- **Advanced States and Transitions**
  - **Deferred Event -** An event whose occurrence is responded to at a later time.
    - Syntax (to be placed inside the state symbol): **event / defer**
    - Once an event has been deferred it remains deferred until a state is entered where that particular type of event
    - listed as deferred.
    - The state diagram then responds
    - to the event as if it had just occurre

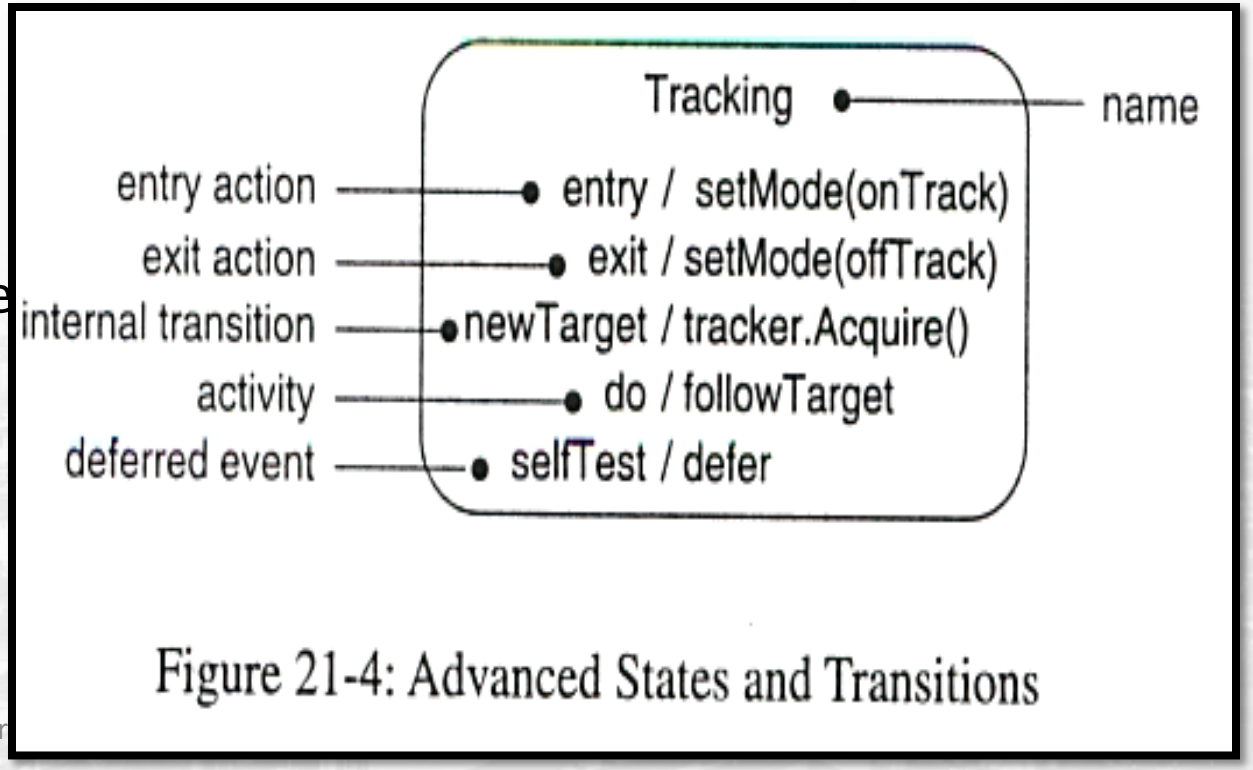
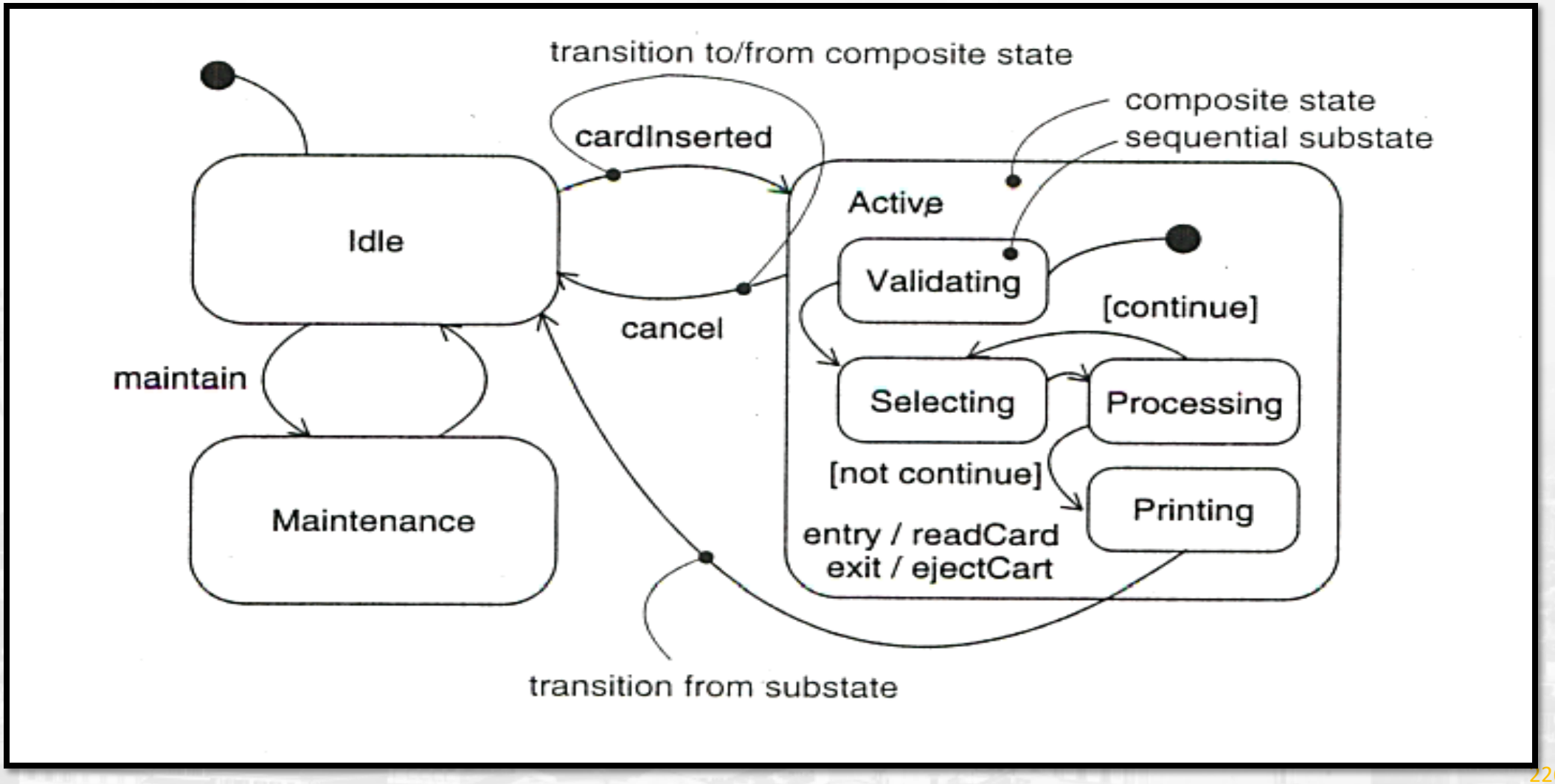


Figure 21-4: Advanced States and Transitions

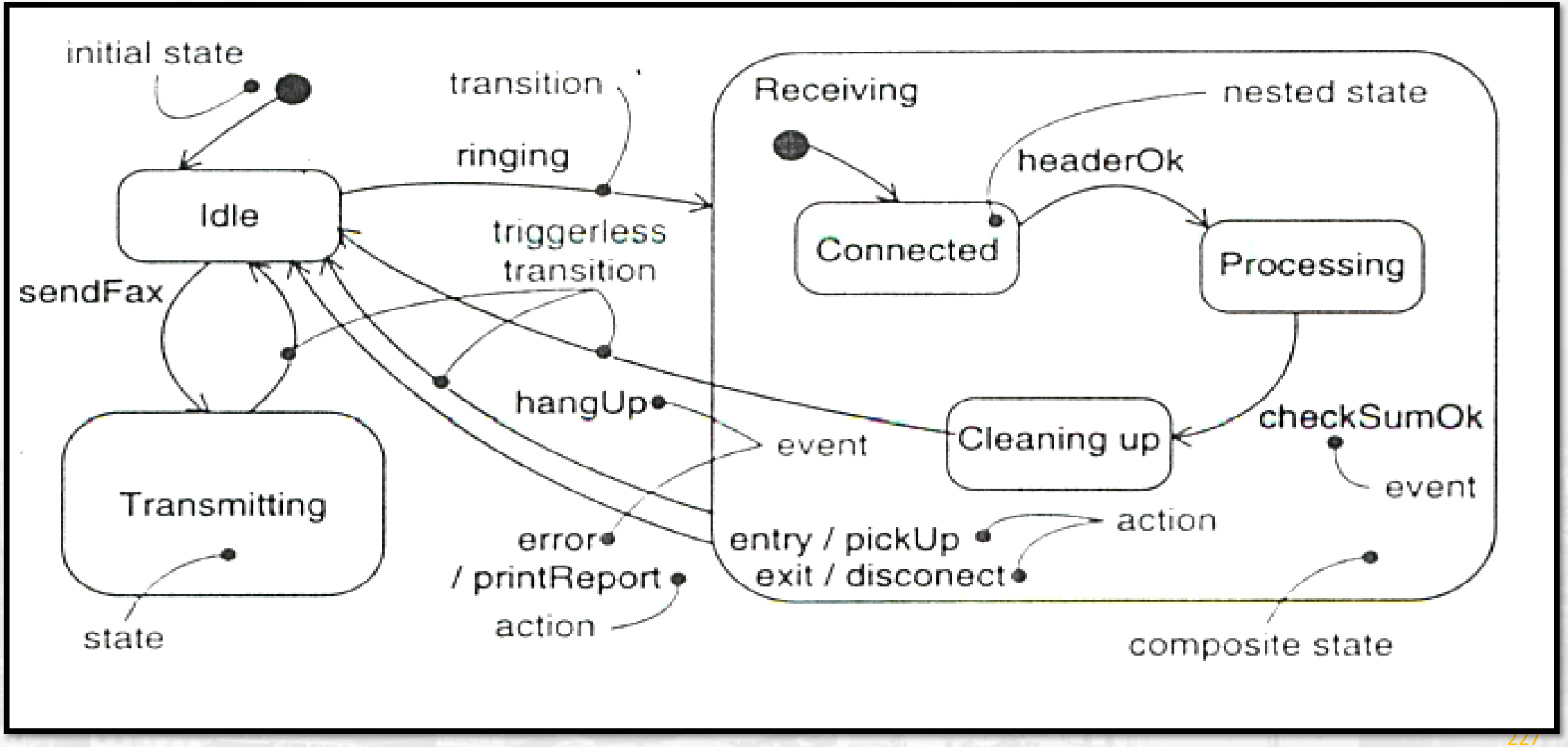
Terms and Concepts

- **Advanced States and Transitions**
  - Simple state - A state that contains no *substates*.
  - Composite state - A state that contains *substates*.
  - Substate - A state that is nested inside another state.
    - Substates allow state diagrams to show different levels of abstraction.
    - Substates may be sequential or concurrent.
    - Substates may be nested to any level.
- **Sequential Substates -** The most common type of substate. Essentially, a state diagram within a single state.
    - The “containing” state becomes an abstract state.
    - The use of substates simplifies state diagrams by reducing the number of transition lines.
    - A nested sequential state diagram may have at most one initial state and one final state.

ATM Machine Composite State Diagram



Phone Call Advance State Diagram



Terms and Concepts

- **Advanced States and Transitions**
  - History States - Allows an object to remember which substate was last active when the containing state was exited.
    - Upon re-entry to the containing state, the substate that was last active will be re-entered directly.

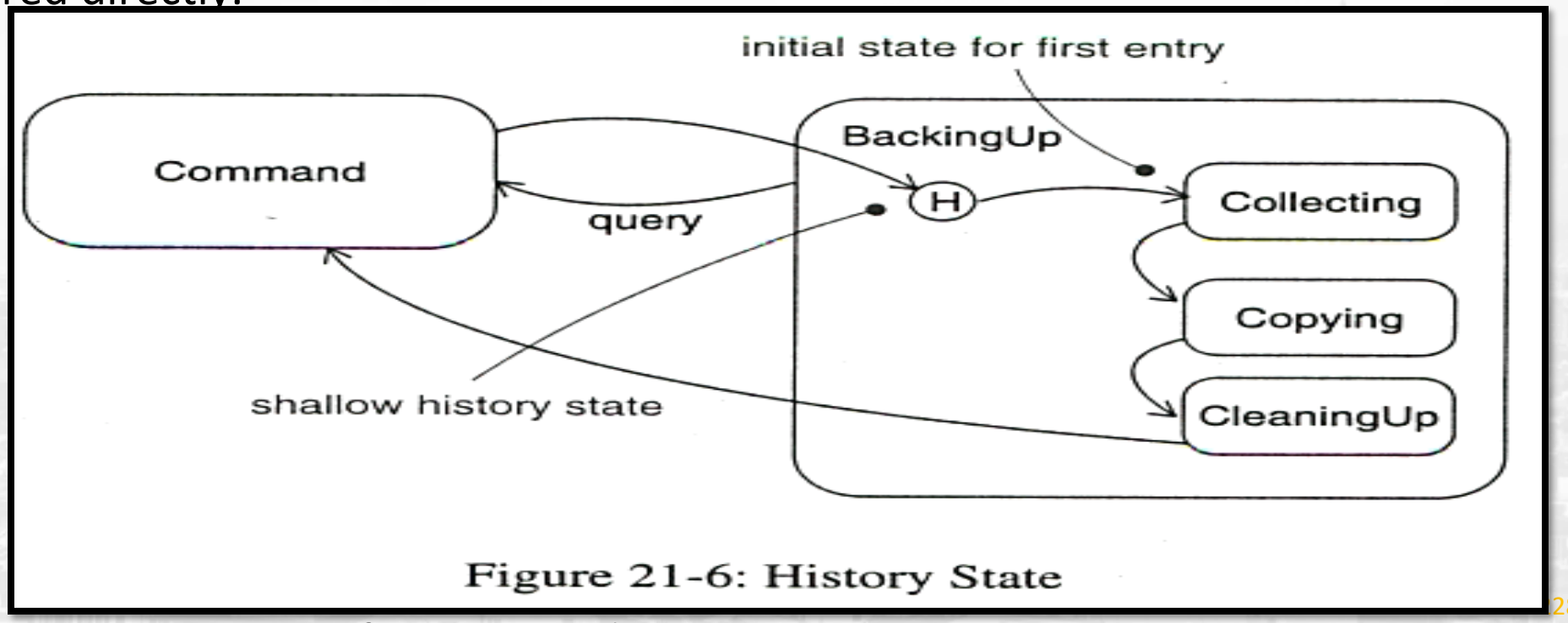


Figure 21-6: History State

Terms and Concepts

- **Advanced States and Transitions**
  - Concurrent Substates - Used when two or more state diagrams are executing concurrently within a single object.
    - Allows an object to be in multiple states simultaneously.
    - The concurrent state diagrams within a “containing” state must begin and end execution simultaneously.
    - If one concurrent state diagram finishes first, it must wait for the others to complete before exiting the containing state.

Concurrent Substrates Example

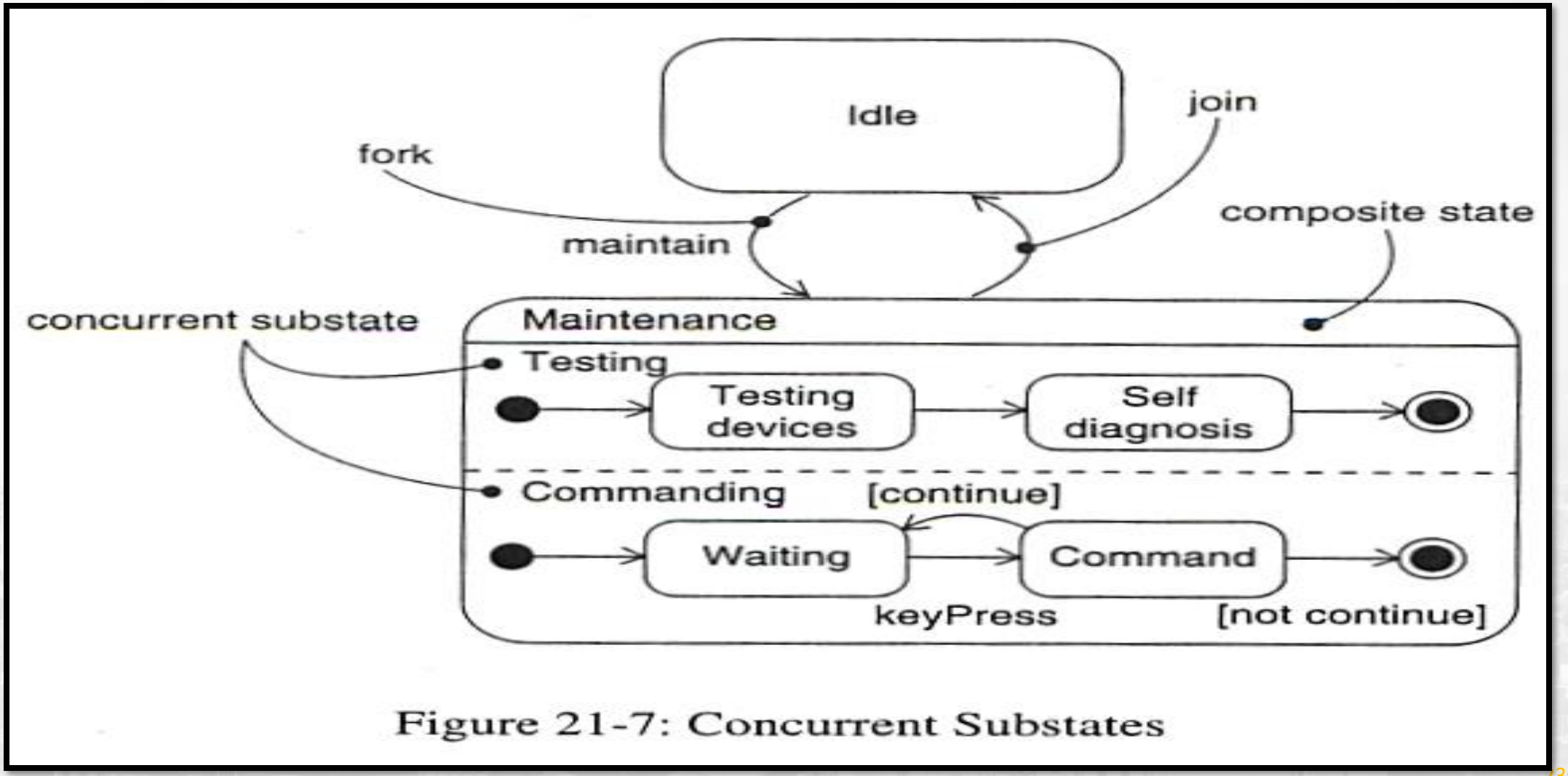


Figure 21-7: Concurrent Substates

State Diagram for a Phone Line

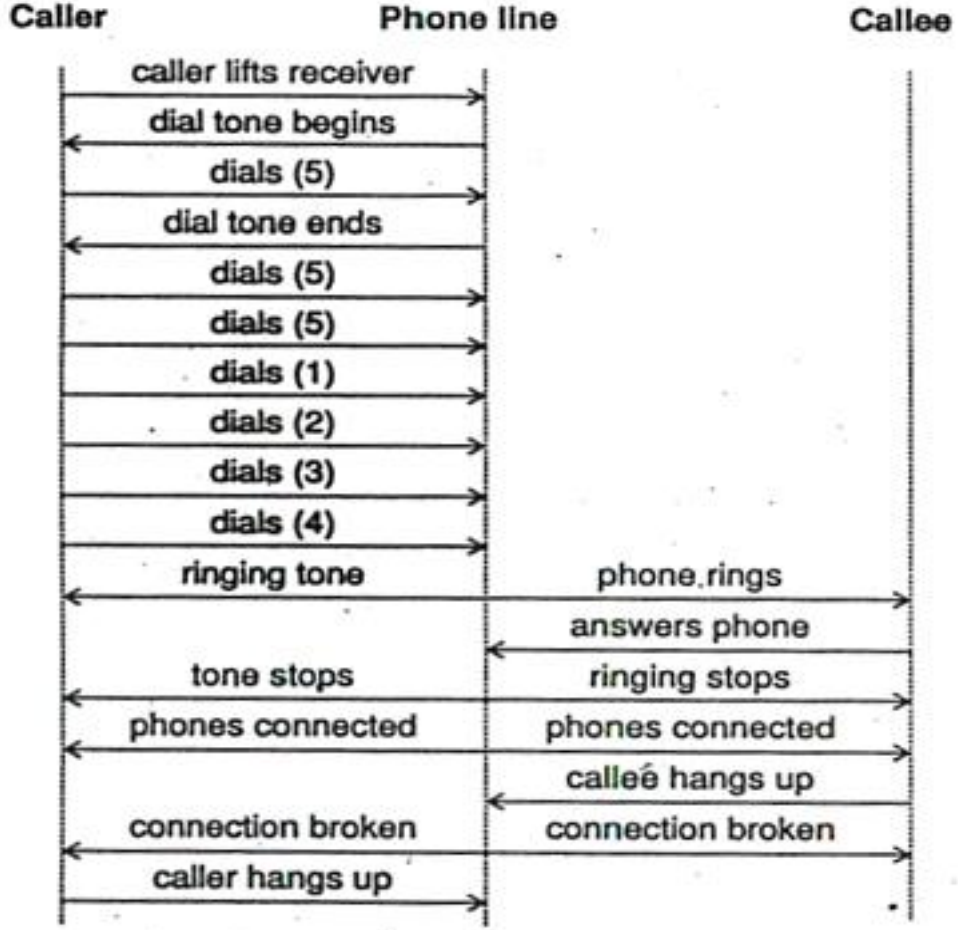


Figure 5.3 Event trace for phone call

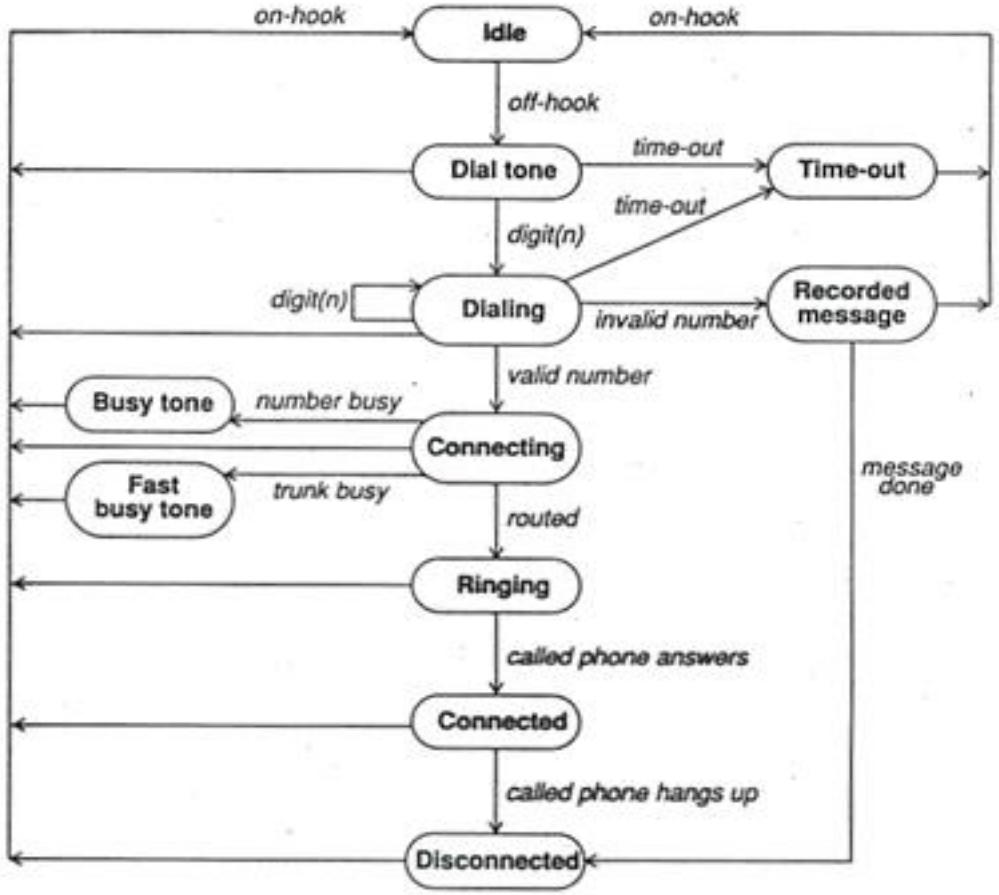


Figure 5.5 State diagram for phone line

References

- Ian Sommerville, — Software Engineering‖, Addison and Wesley. 9th Ed., 2011.
- Roger S Pressman, Software Engineering: A Practitioner's Approach, Mcgraw-Hill, ISBN: 0073375977, Seventh Edition, 2014
- Pankaj Jalote, Software Engineering: A Precise Approach, Wiley India.2010.

Disclaimer:

a. Information included in this slides came from multiple sources. We have tried our best to cite the sources. Please refer to the References to learn about the sources, when applicable.

b. The slides should be used only for academic purposes (e.g., in teaching a class), and should not be used for commercial purposes.