

# Requirement Engineering

## UNIT II

# Syllabus – Unit II

## **REQUIREMENT ENGINEERING**

Software Requirements, User requirements, System requirements, Software Requirements Specification, Requirement Engineering Process, Requirements validation

## **SOFTWARE DESIGN**

Abstraction, Modularity, Cohesion & Coupling, Scenario based modeling, Introduction to Unified Modeling Language (UML) and Data Flow Diagram ( DFDs).

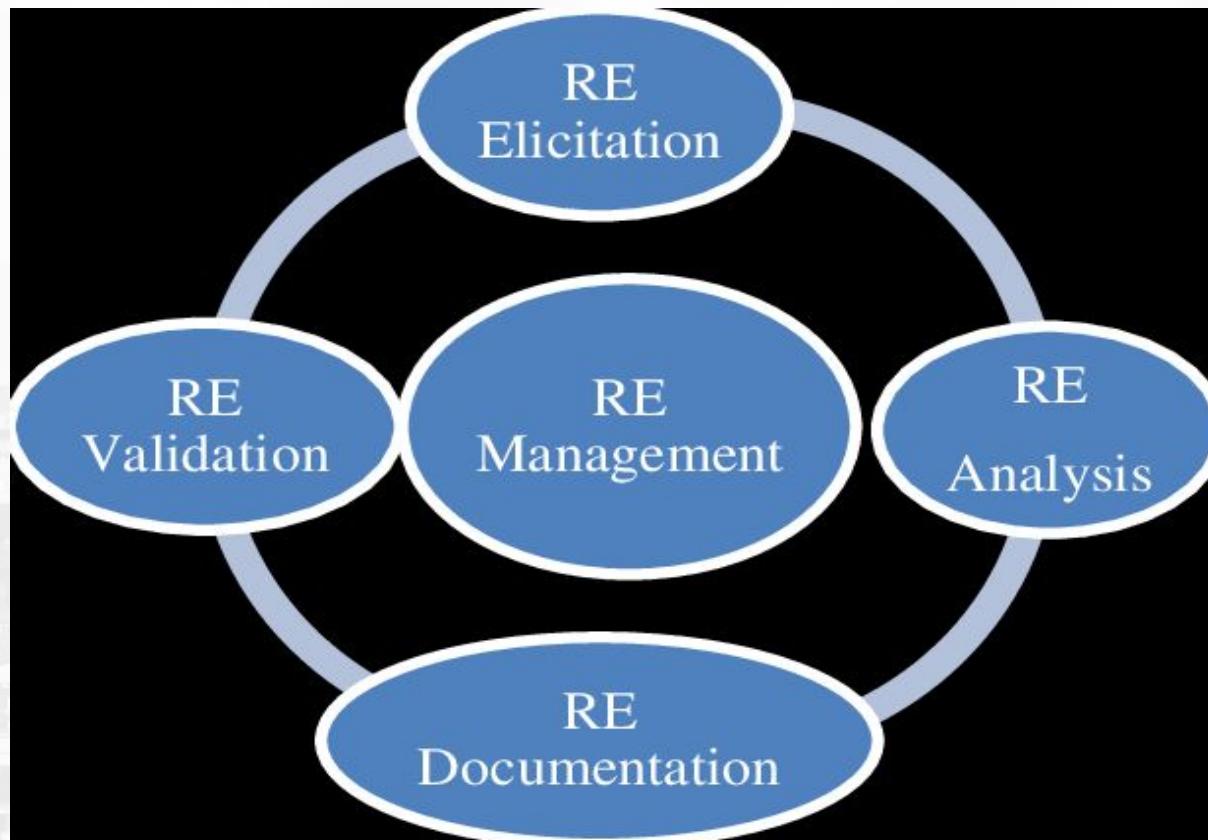
Case Studies.

# Software Development Life Cycle



# The Requirement Engineering

Requirements engineering is a process of gathering and defining of what the services should be provided by the system.



# What is Requirements Engineering?

- Requirements are **description of features and functionalities** of the system and convey the **expectations** of users from the software product.
- Requirements engineering refers to the process of **defining, documenting and maintaining requirements** in the engineering design process.
- It is a common role in systems engineering and software engineering.
- The requirements can be obvious or hidden, known or unknown, expected or unexpected from client's point of view.

# Requirement Definitions and Specifications

## Example

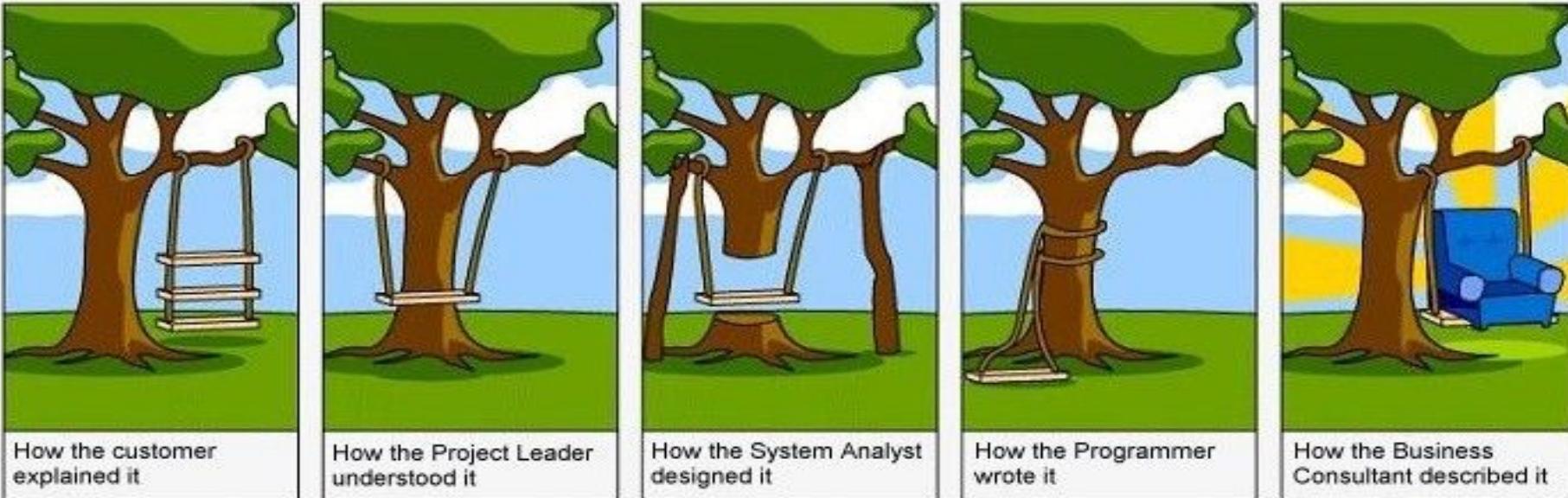
### Requirements definition

1. The software must provide a means of representing and accessing external files created by other tools.

### Requirements specification

- 1.1 The user should be provided with facilities to define the type of external files.
- 1.2 Each external file type may have an associated tool which may be applied to the file.
- 1.3 Each external file type may be represented as a specific icon on the user's display.
- 1.4 Facilities should be provided for the icon representing an external file type to be defined by the user.
- 1.5 When a user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file represented by the selected icon.

# Ugly Face of Requirement Engineering



# Requirement Engineering Steps

- RE has overlapping steps :
  - **Inception** - in which the nature and scope of the system is defined.
  - **Elicitation** - in which the requirements for the software are initially gathered.
  - **Elaboration** - in which the gathered requirements are refined.
  - **Negotiation** -in which the priorities of each requirement is determined, the essential requirements are noted, and, conflicts between requirements are resolved.
  - **Specification** - in which the requirements are gathered into a single product, being the result of the requirements engineering.
  - **Validation** - in which the quality of the **requirements** (i.e., are they **unambiguous, consistent, complete, etc.**), and the developer's interpretation of them, are assessed.
  - **Management** - in which the changes that the requirements must undergo during the project's lifetime are managed.

# Requirements Engineering Steps

- Inception— Ask a set of questions that establish ...
  - Basic understanding of the problem
  - The nature of the solution that is desired
  - The effectiveness of preliminary communication and collaboration between the customer and the developer
  - The people who want a solution -Identify Stakeholders : Stakeholders can affect or be affected by the application's actions, objectives and policies. **End Users, Build Team and Authorities**
  - Recognize multiple points of view

# Requirements Engineering- Steps

- **Elicitation**

- Elicit requirements , identify problems from all stakeholders
- Propose elements of the solution
- The scope and negotiate different approaches,,
- Understanding of the problem and volatility (requirements change over time )
- Specify a preliminary set of solution requirements

# Requirements Engineering- Steps

- Elaboration

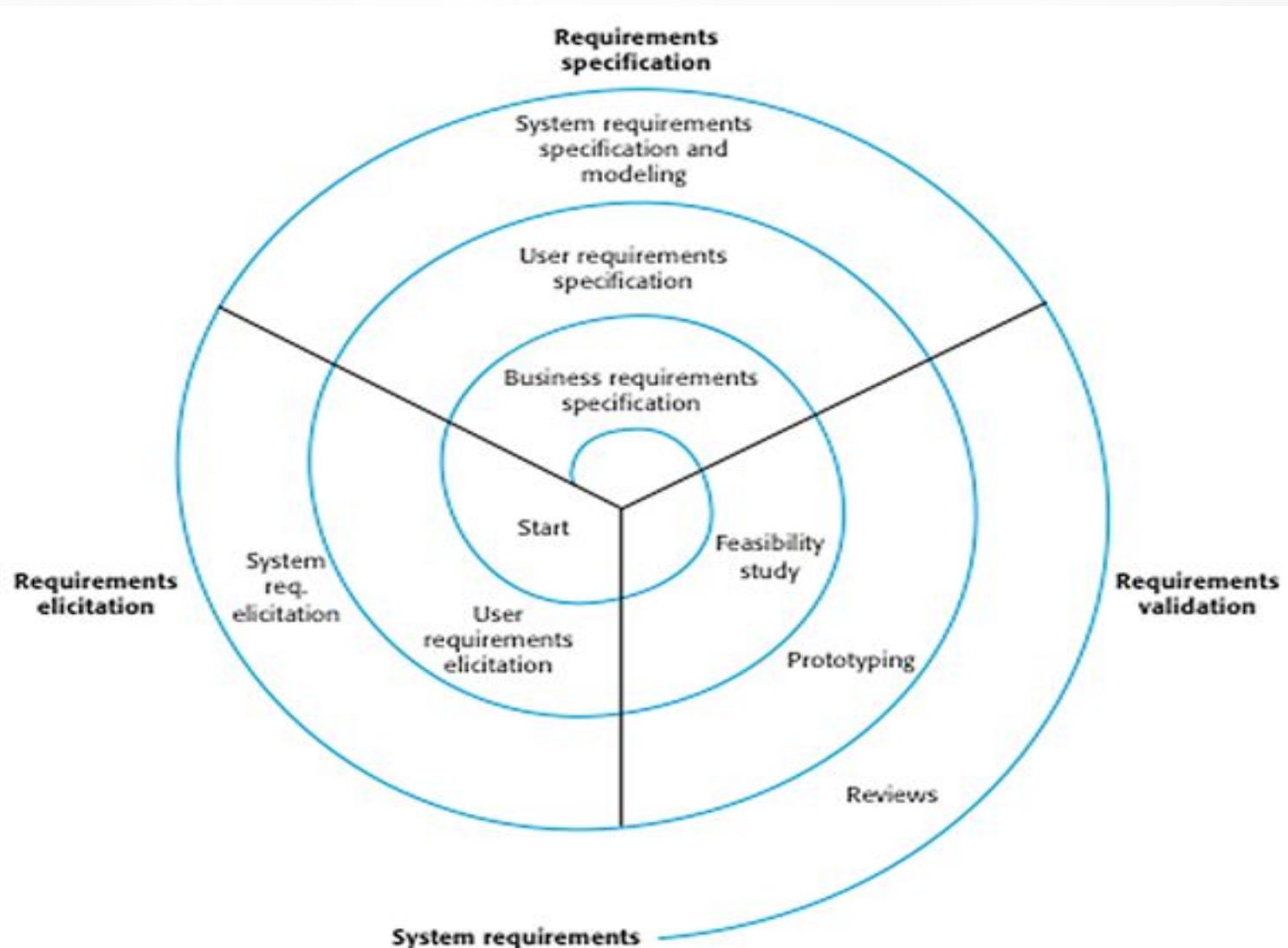
- Create an analysis model that identifies data, function and behavioral requirements.
- It is driven by the creation and refinement of user scenarios that describe how the end-user will act with the system.

# Requirements Engineering- Steps

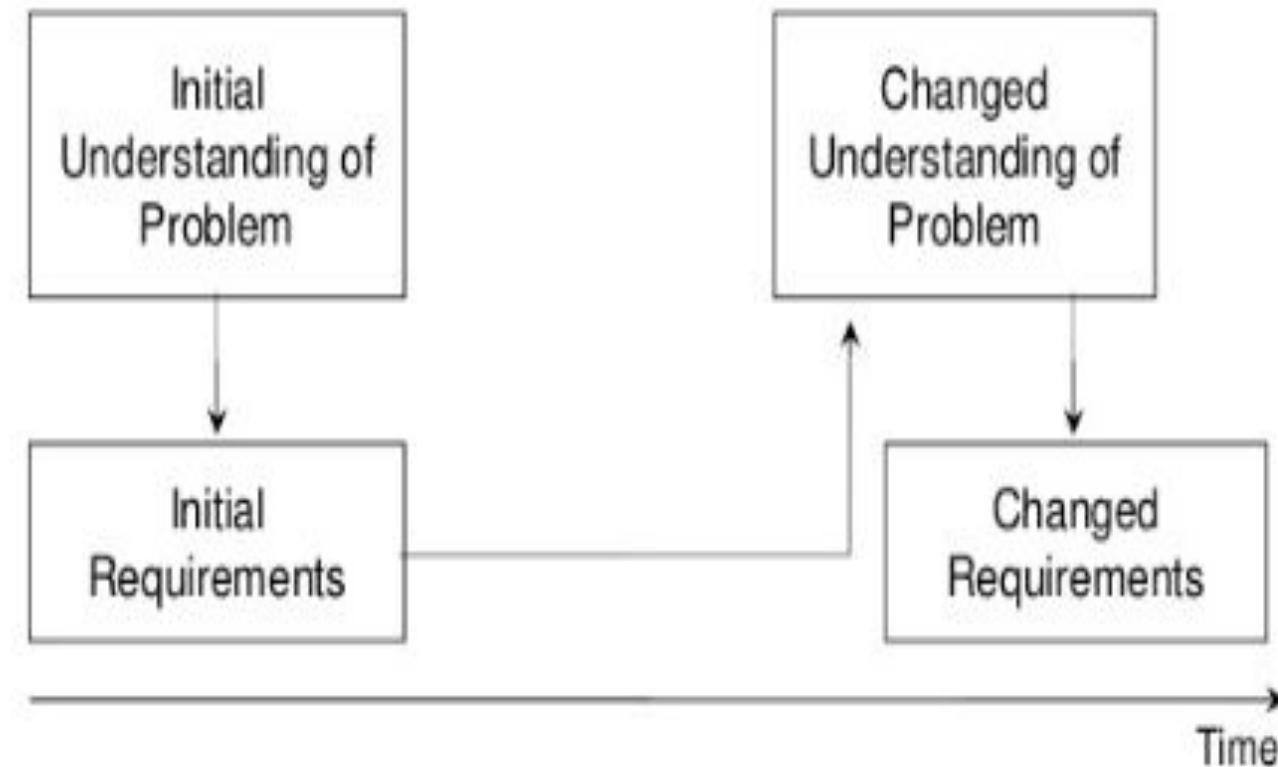
- Negotiation
  - Identify Conflicting Requirements and reconcile these conflicts through a process of *negotiation*.
  - Stakeholders are asked to rank requirements and then discuss conflicts in priority.
  - Risks in each requirement are identified and analyzed.
  - Agree on a deliverable system that is realistic for developers and customers.
- Specification : SRS or a prototype
  - It is the final work produced by the RE. It serves as the foundation for subsequent S.E. activities.

- **Validation**
  - A review mechanism that looks for
    - Errors in content or interpretation
    - Areas where clarification may be required
    - Missing information
    - Inconsistencies (a major problem when large products or systems are engineered)
    - Conflicting or unrealistic (unachievable) requirements.
- **Management**
  - Set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds.

# The process of requirements engineering

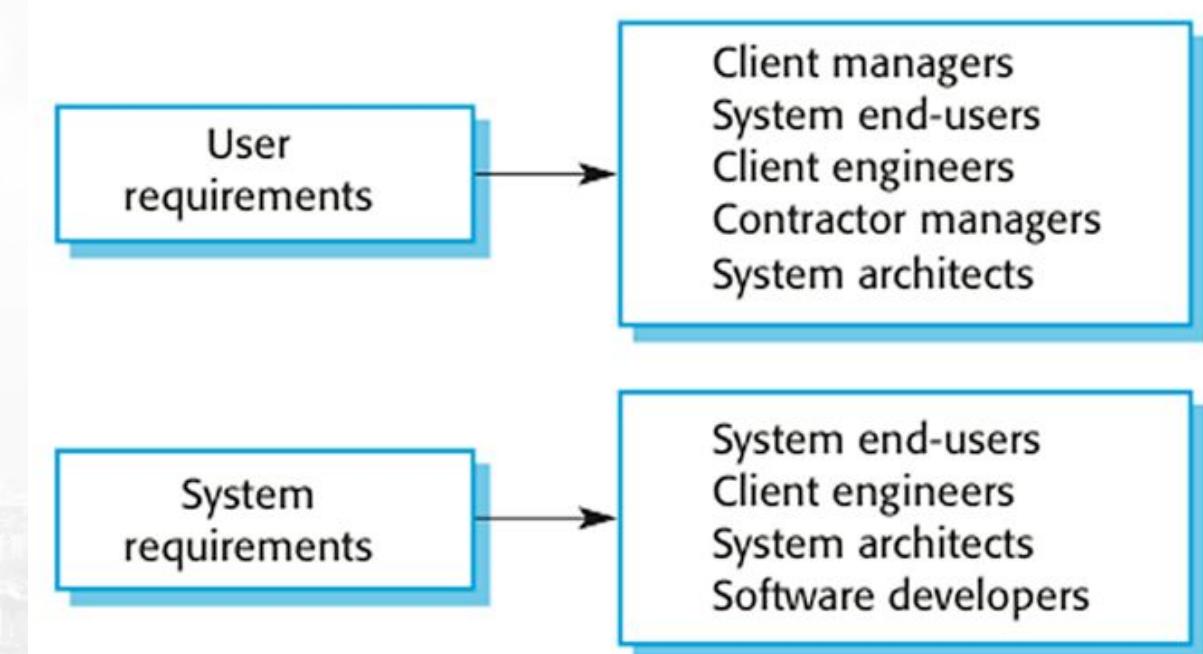


# Requirements Evolution



# User and System Requirements

- User need a high-level statements of the requirements, while system developers need a more detailed system specification.
- Having different level of details is useful because it communicates information about the system being developed for different types of readers.



# User and System Requirements

- User Requirements : describes the services that the system should provide and the constraints under which it must operate.
  - more of generic requirements and Readable by everybody
  - Services and constraints of the system
  - In natural language or diagrams
  - Serve business objectives
- System Requirements : gives a more detailed description of the system services and the operational constraints (how system will be used, programming languages etc)
  - This level of detail is needed by those who are involved in the system development
  - Useful for the design and development
  - Precise and cover all cases
  - Structured presentation

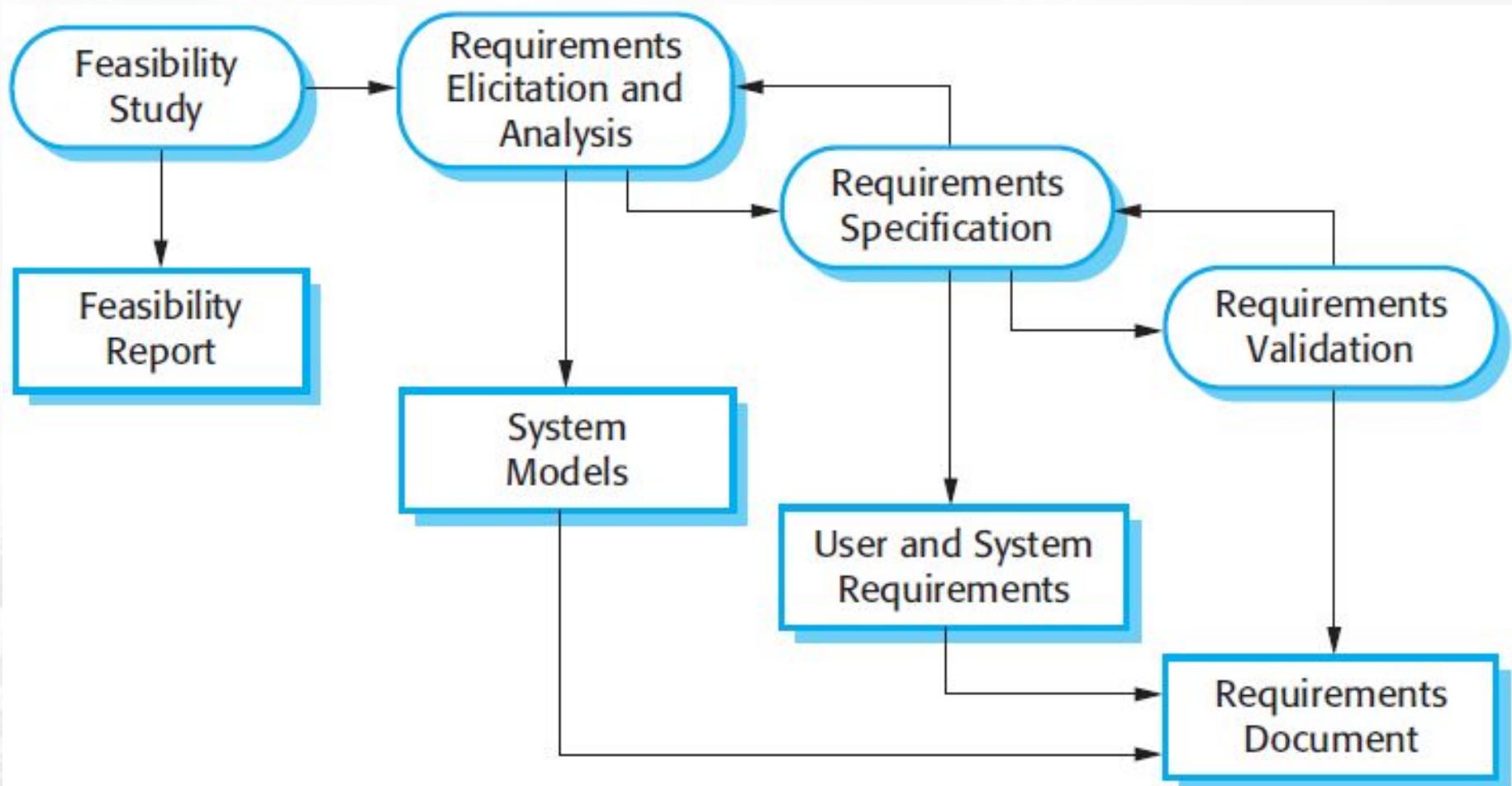
# Example

- **User requirement:** The library system should provide a way to allow a patron to borrow a book from the library.
- **System requirement:** The library system should provide a withdraw interaction that allows a patron to withdraw a book given the isbn and copy number of the book to be withdrawn. The interaction fails if: the book is already withdrawn, the book is not in the library's collection, the patron has already withdrawn 5 books, the book is on hold by someone else.
- **Software Specification:** A detailed software description which can serve as a basis for a design or implementation. Written for developers

# RE Process

- Requirement Engineering is the process of defining, documenting and maintaining the requirements.
- A process of gathering and defining service provided by the system.
  - It ensures your software will meet the user expectations, and ending up with a high quality software.
  - It's a critical stage of the software process as errors at this stage will reflect later on the next stages, which definitely will cause you a higher costs.
- At the end of this stage an SRS is produced and validated with the stakeholders.

# The RE process



# RE Process

- **Feasibility study:**
  - Check is made if the identified can be achieved using the given conditions
  - This step should be cheap and quick;
- **Requirements elicitation and analysis:**
  - Deriving the system requirements through observation of existing systems, discussions with stakeholders, etc.
  - Can involve development of a system models and prototypes to understand the specified system
- **Requirements specification:**
  - It's the activity of writing down the information gathered during the elicitation and analysis activity into a document that defines a set of requirements.
  - Two types of requirements may be included in this document; [user and system requirements](#).
- **Requirements validation:**
  - It's the process of checking the requirements for realism, consistency and completeness.
  - Goal is to discover errors in the requirements document.
  - When errors are found, it must be modified to correct these problems.

# Types of Requirements

- **Functional requirements**
  - Services the system should provide
  - What the system should do or not in reaction to particular situations
- **Non-functional requirements**
  - Constraints on the services or functions offered by the system
  - Examples: Timing constraints, constraints on the development process (CASE, language, development method...), standards etc
- **Domain requirements**
  - From the application domain of the system
  - May be functional or non-functional
  - Examples: Medicine, library, physics, chemistry

# Bad Requirements

- **Missing Requirements** – A functionality that is totally missing from the documentation.
  - “Error messaging” in case of data validation
  - not detailing the need for a certain link available as per the access rules on an application.
- **Conflicting Requirements** – When two or more requirements expect the system to do different things that can't possibly be done at the same time.
  - the business stakeholder might want to retrieve a 1000 records at a time in real-time whereas the technology stakeholder knows this is practically impossible and not feasible.
- **Incomplete/Unclear Requirements** – Requirements that lack all the necessary information constitute this lot.
  - “The system should have the capability to filter search results”. The details around filter criteria is not been provided and thus raises unnecessary queries.
- **Ambiguous Requirements** – A requirement statement that can be interpreted in different ways by different people.

# Ambiguous Requirements

- A client needed to be able to upload "large files." A solution was found where the platform could handle the "large files." Client was assured a solution was found and testing began. Client gave a feedback that the solution was not meeting their needs, was freezing and full of bugs.
  - To the client "large" meant 20-50GB and to our team and the platform provider "large" meant up to 5GB.
- Trigger an automatic log out when a user attempts to download up to 5 times. Display is up to large screen size.
  - Is this 5 times including the 5th time or 4 times?
  - the page has a document that is in 5 parts
  - For the screen size, 720px may be large for some and 1280 may be large for some.

# Requirements Checking

- **Validity.** Does the system provide the functions which best support the customer's needs?
- **Consistency.** Are there any requirements conflicts?
- **Completeness.** Are all functions required by the customer included?
- **Realism.** Can the requirements be implemented given available budget and technology
- **Verifiability.** Can the requirements be checked?

# 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.

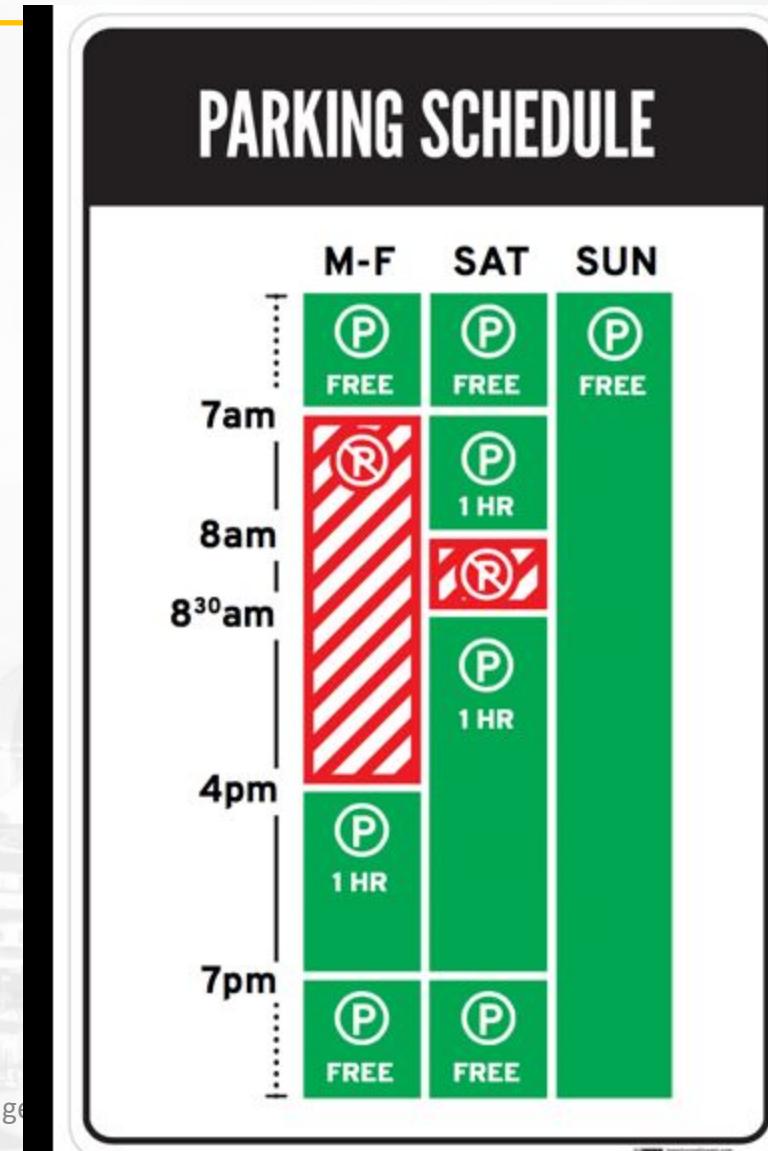
## 3. **Detailed Design-**

- 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

1. Meets all technical requirements
2. Works all the time
3. Meets cost requirements
4. Requires little or no maintenance
5. Is safe
6. Creates no ethical dilemma

## Bad Design

1. Meets only some technical requirements
2. Works initially but stops working after a short time
3. Costs more than it should
4. Requires frequent maintenance
5. Poses a hazard to users
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.

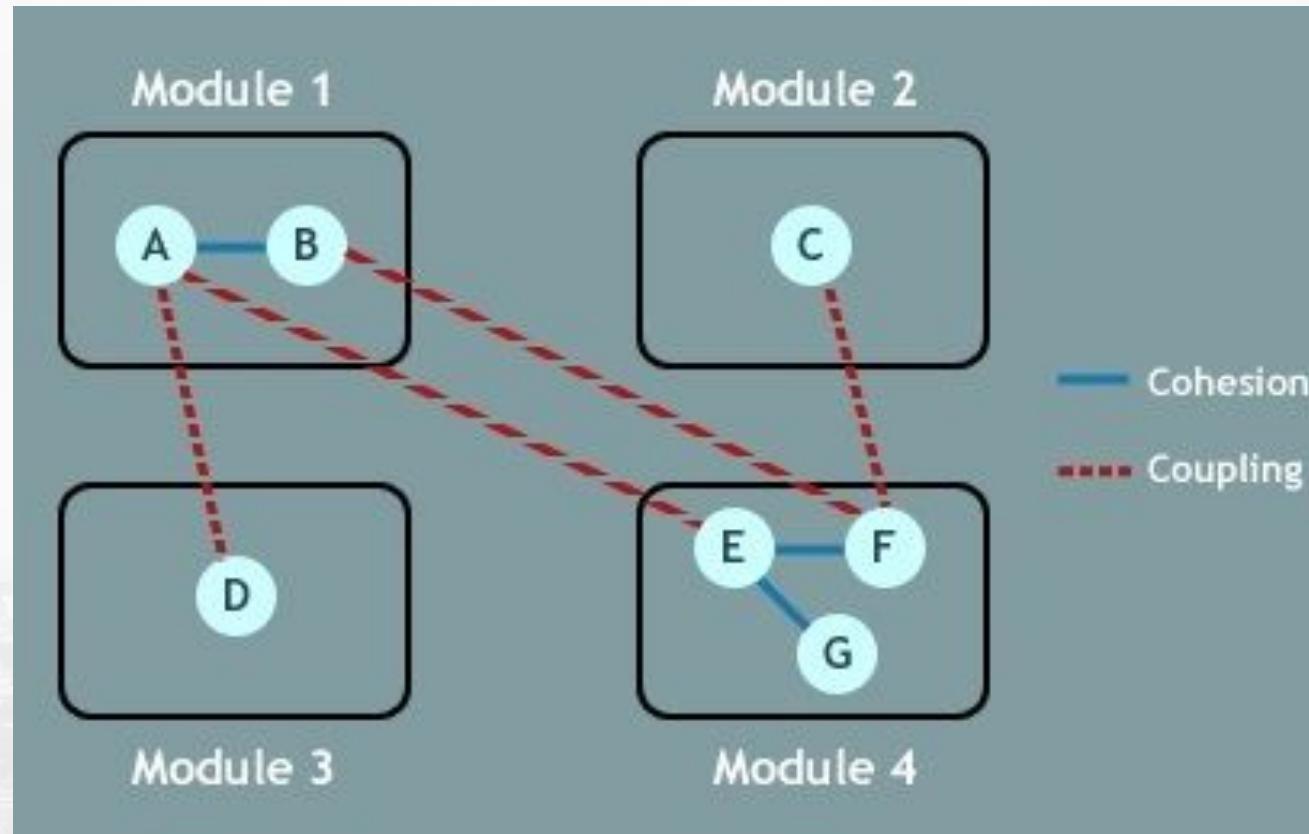
# Modularity

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

[Elliot Chikofsky and James Cross, Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software 7(1):13-17, 1990.]

# Reverse Engineering

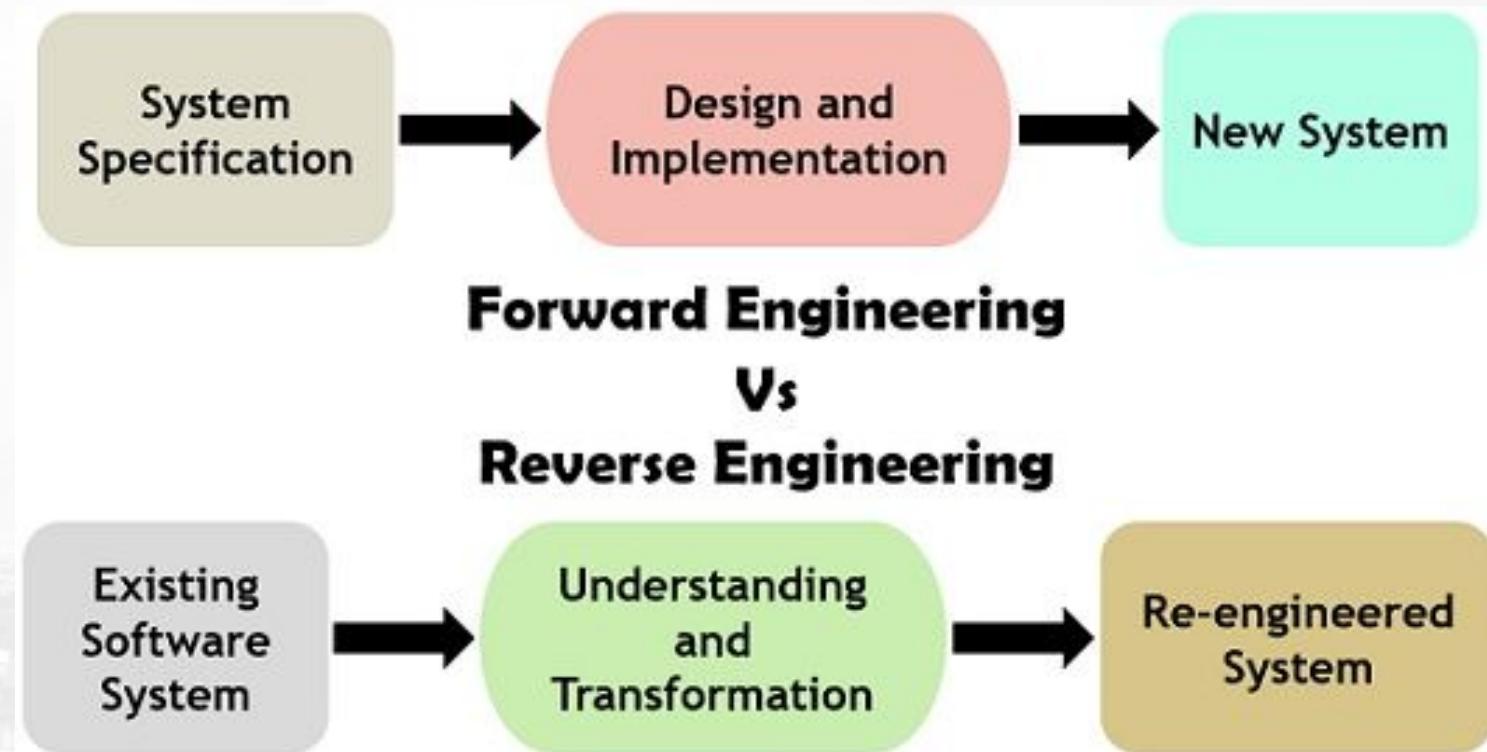
- Analyzing software with a view to understanding its design and specification
- May be part of a re-engineering process but may also be used to re-specify a system for re-implementation
- Builds a program data base and generates information from this
- Program understanding tools (browsers, cross-reference 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,
  2. 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

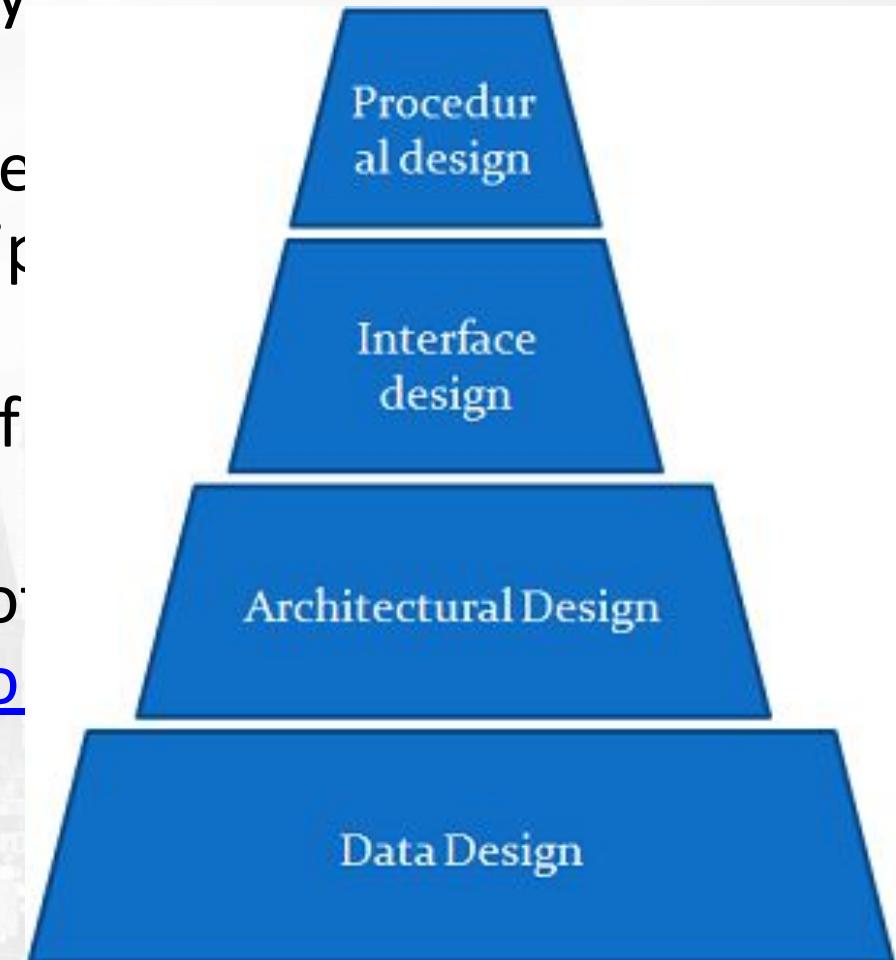
- **Traditional Approach:** SSAD(System Structured Analysis and Design)
  - Structured Analysis (SA), resulting in a logical design, drawn as a set of **data flow diagrams**
  - Structured Design (SD) transforming the logical design into a program structure drawn as a set of **structure charts**
- **OOAD**(Object 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 relationships between the modules of the system
- **Interface design** defines interaction of people with the system
- **Procedural design** transforms the modules of the system architecture in to Pseudo-code or Flow Chart



# Control Flow Diagrams

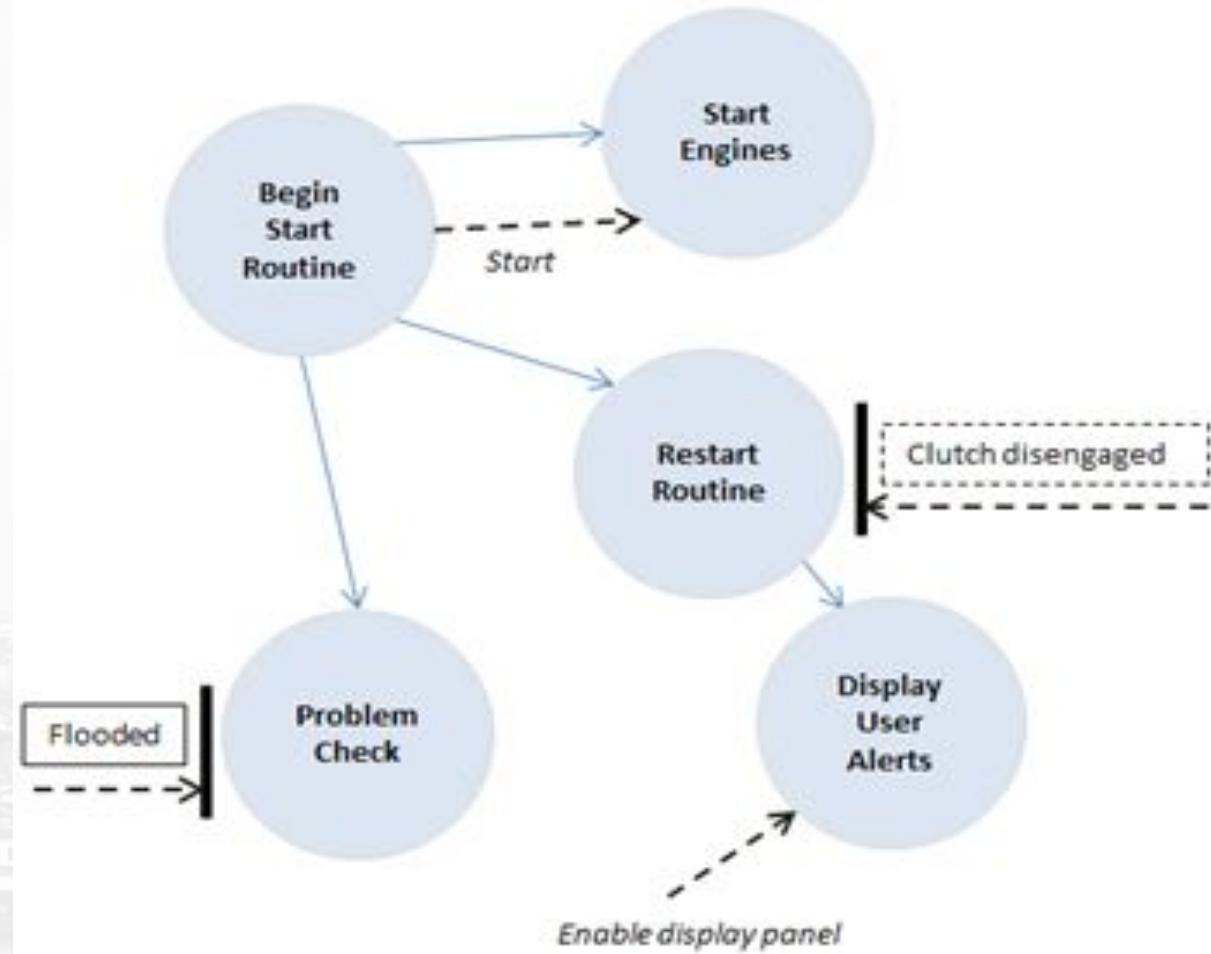
- A control flow diagram helps to understand the detail of a process.
- It shows us where control starts and ends and where it may branch off in another direction, given certain situations.
- *Let's say you are working on software to start a machine. What happens if the engine is flooded, or a spark 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

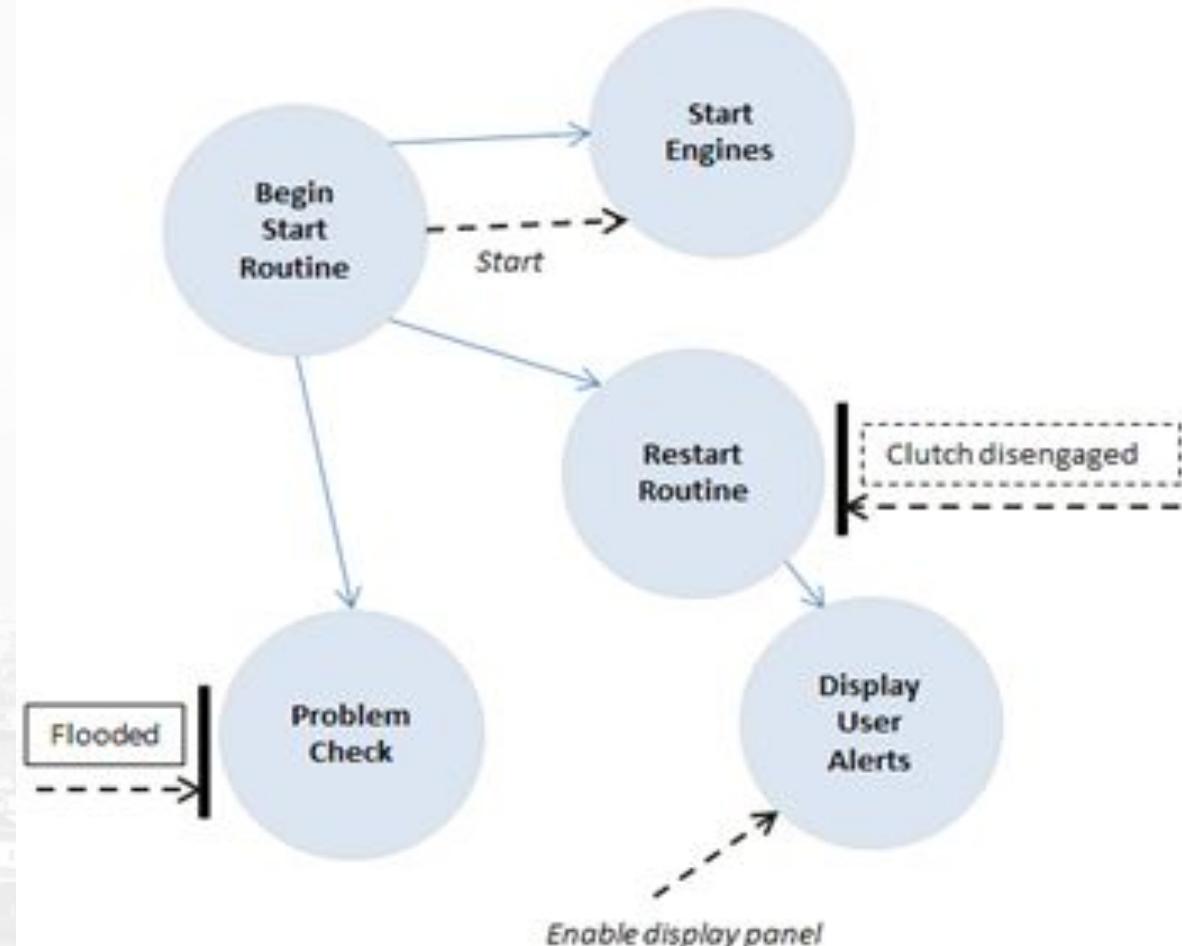
- Keeping with the idea of an engine-start program, the following shows flow through the software:
- **Vertical Bar**
  - The vertical bar indicates **input or output** from the current control specification.
  - Think of the bar as a window INTO or OUT OF the entire specification/flow. Additional notes,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



# Control Flow Diagram

- **Dashed Arrow**

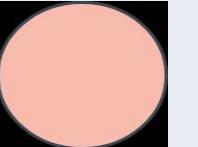
- The input to the problem check step is *Flooded*
- The input into the restart routine is *clutch disengaged*



# 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	<b>External Entity</b>	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	<b>Process</b>	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	<b>Data store</b>	Files or repositories that hold information for later use, such as a database table or a membership form.		
4	<b>Data flow</b>	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

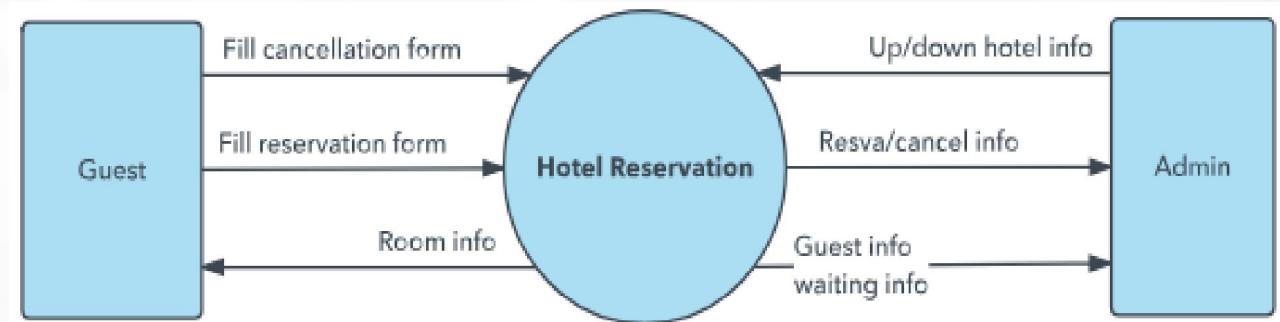
- A data flow diagram can dive into progressively more detail by using levels and layers, zeroing in on a particular piece. DFD levels are numbered 0, 1 or 2, and occasionally go to even Level 3 or beyond.
  - Level 0
  - Level 1
  - Level 2

# 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

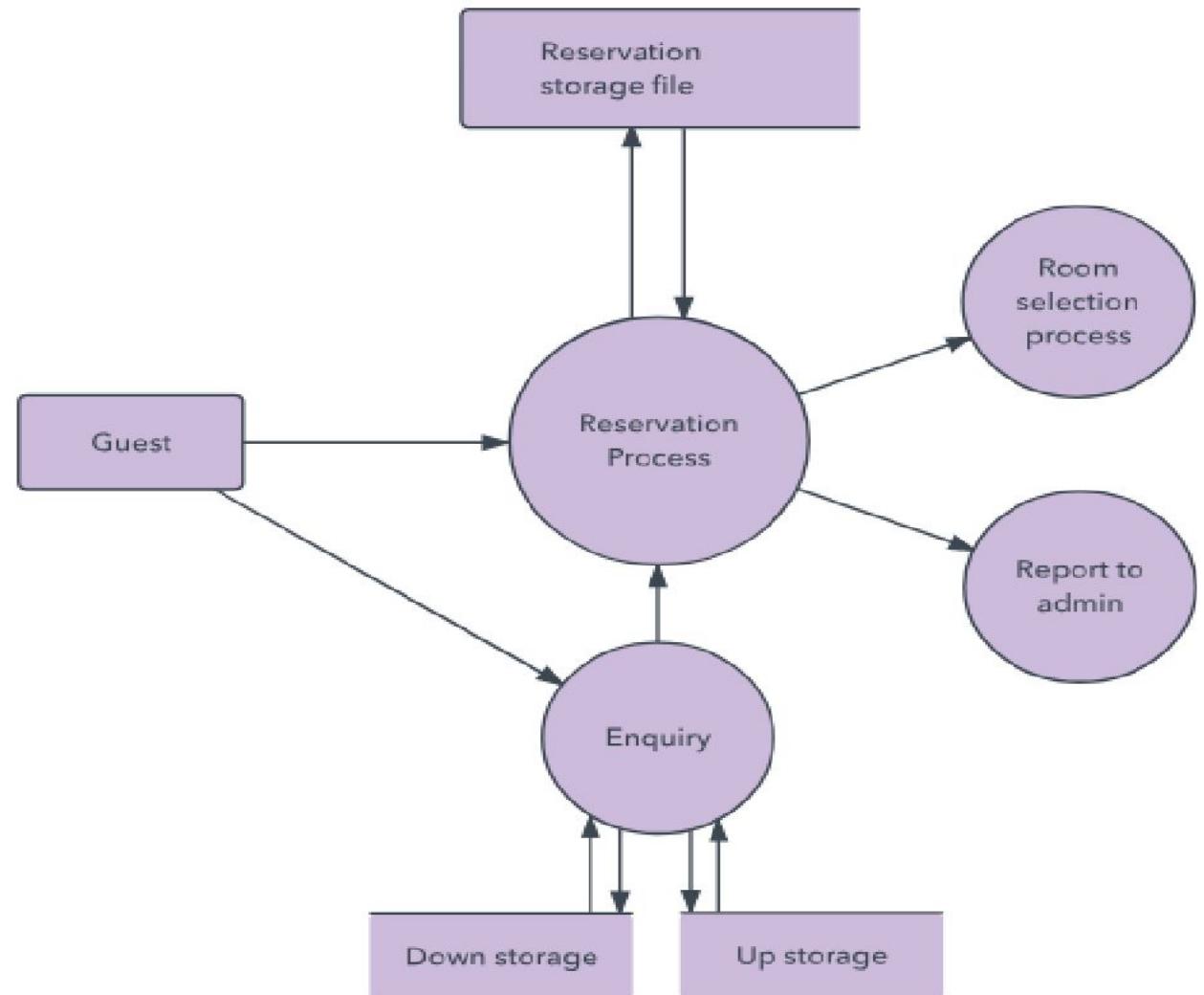
# DFD- Level 0

- 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.



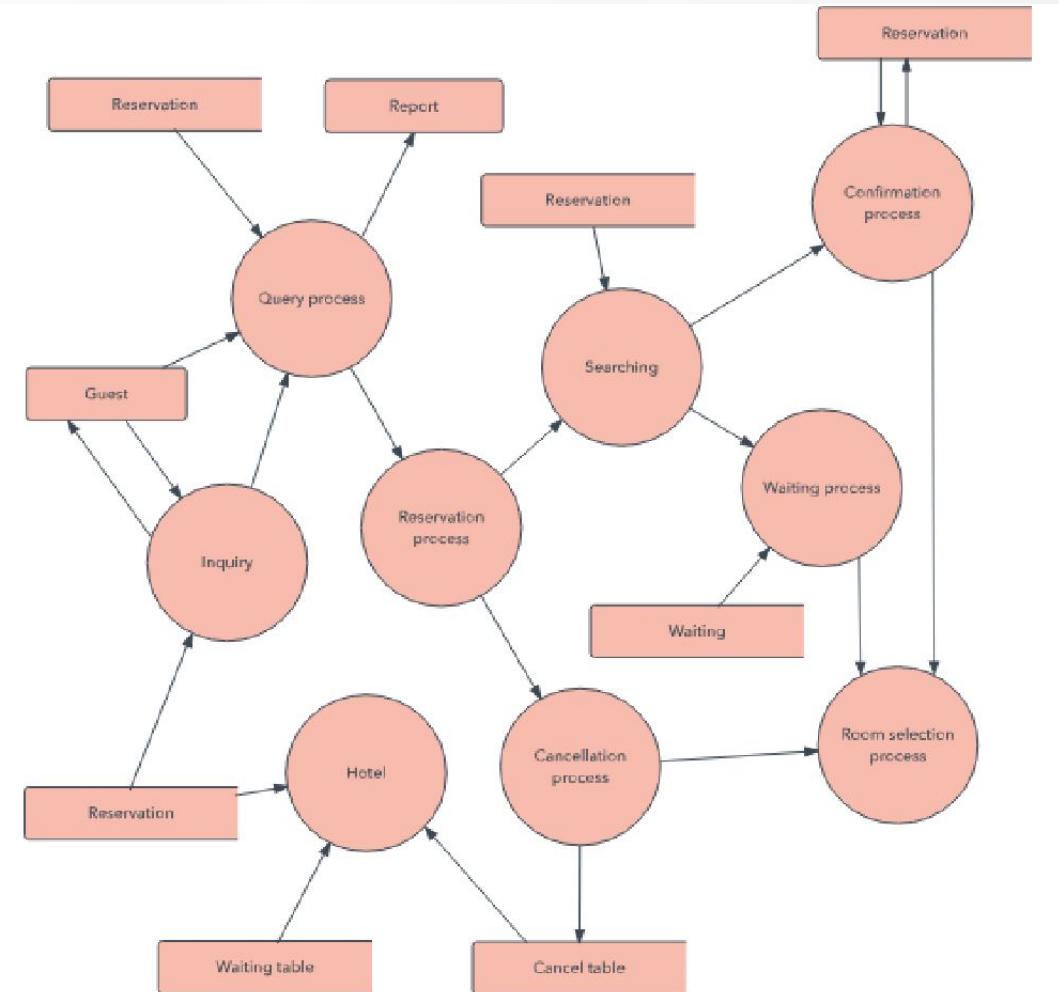
# DFD- Level 1

- 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)

### Context Level DFD



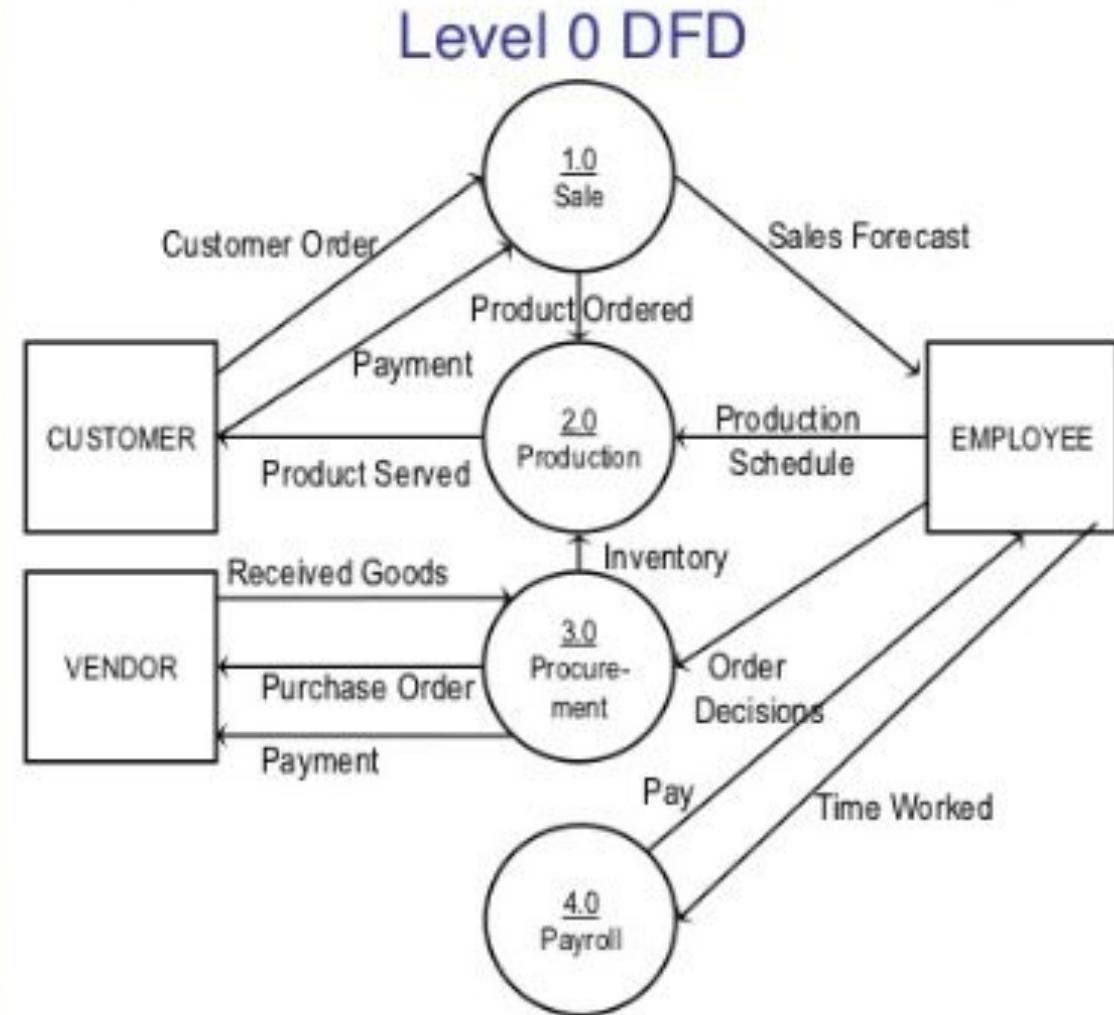
# Creating Data Flow Diagrams

## Example

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



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



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

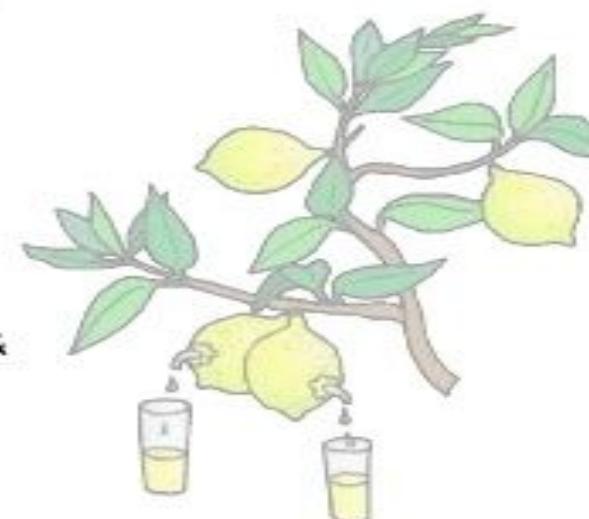
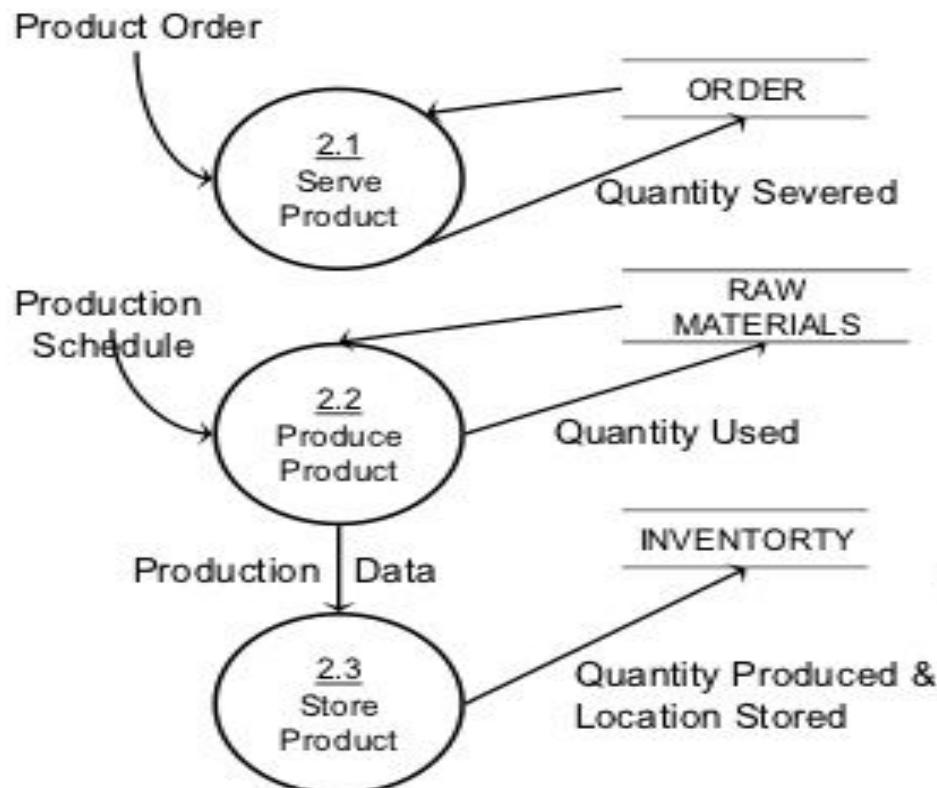
#### Level 1 DFD



# Level 1

## 4. Construct Level 1 (continued)

Level 1 DFD

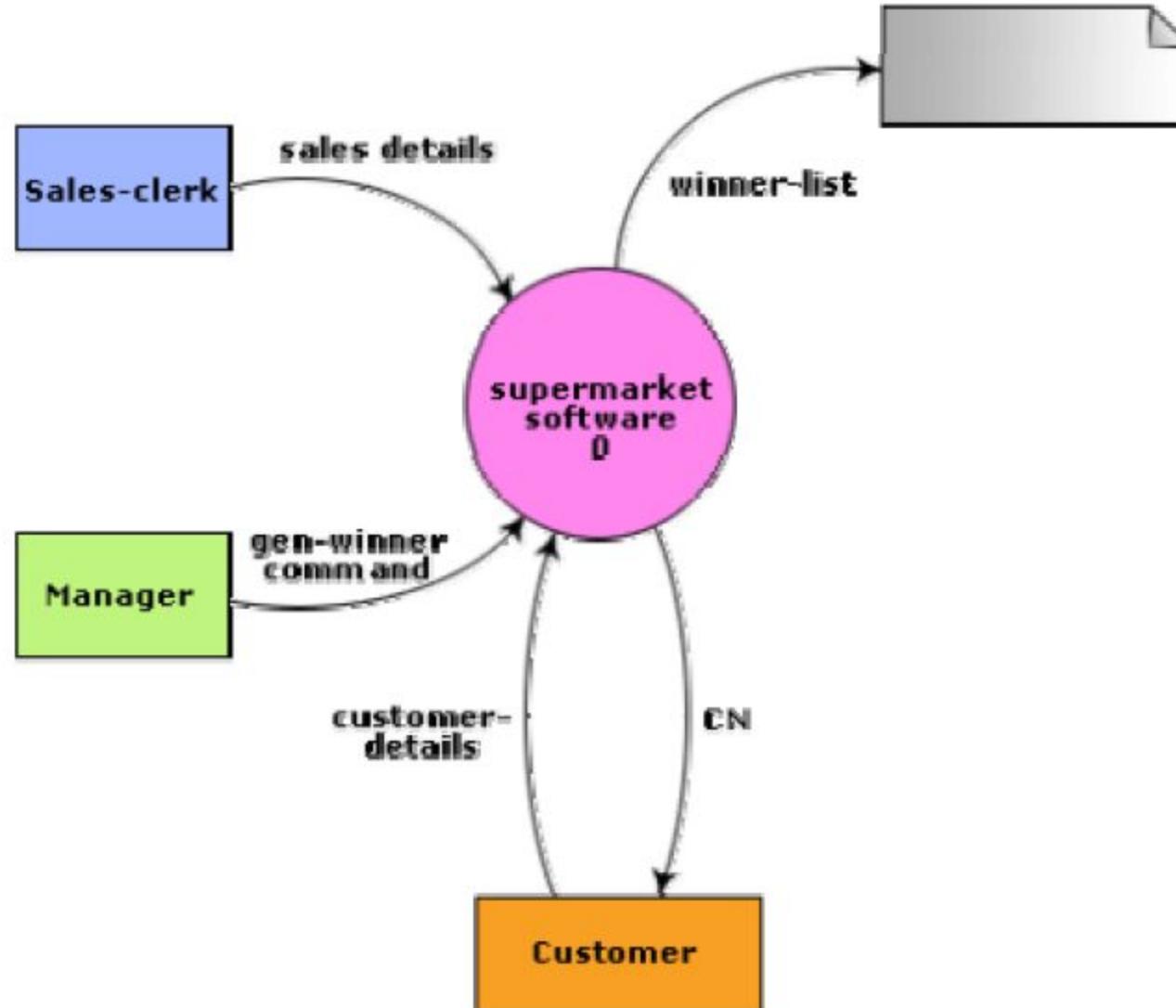




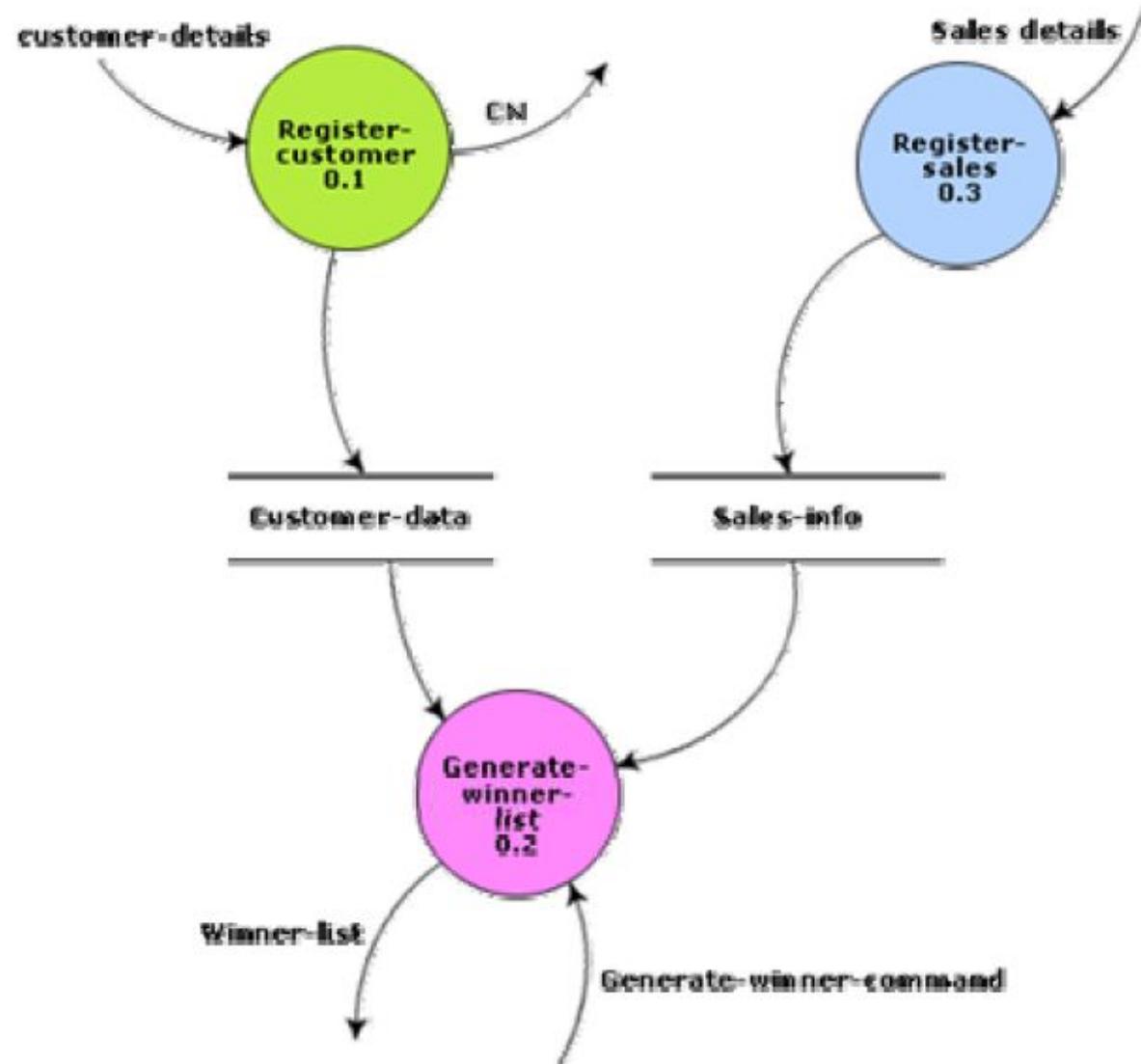
## Example 3

- 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.

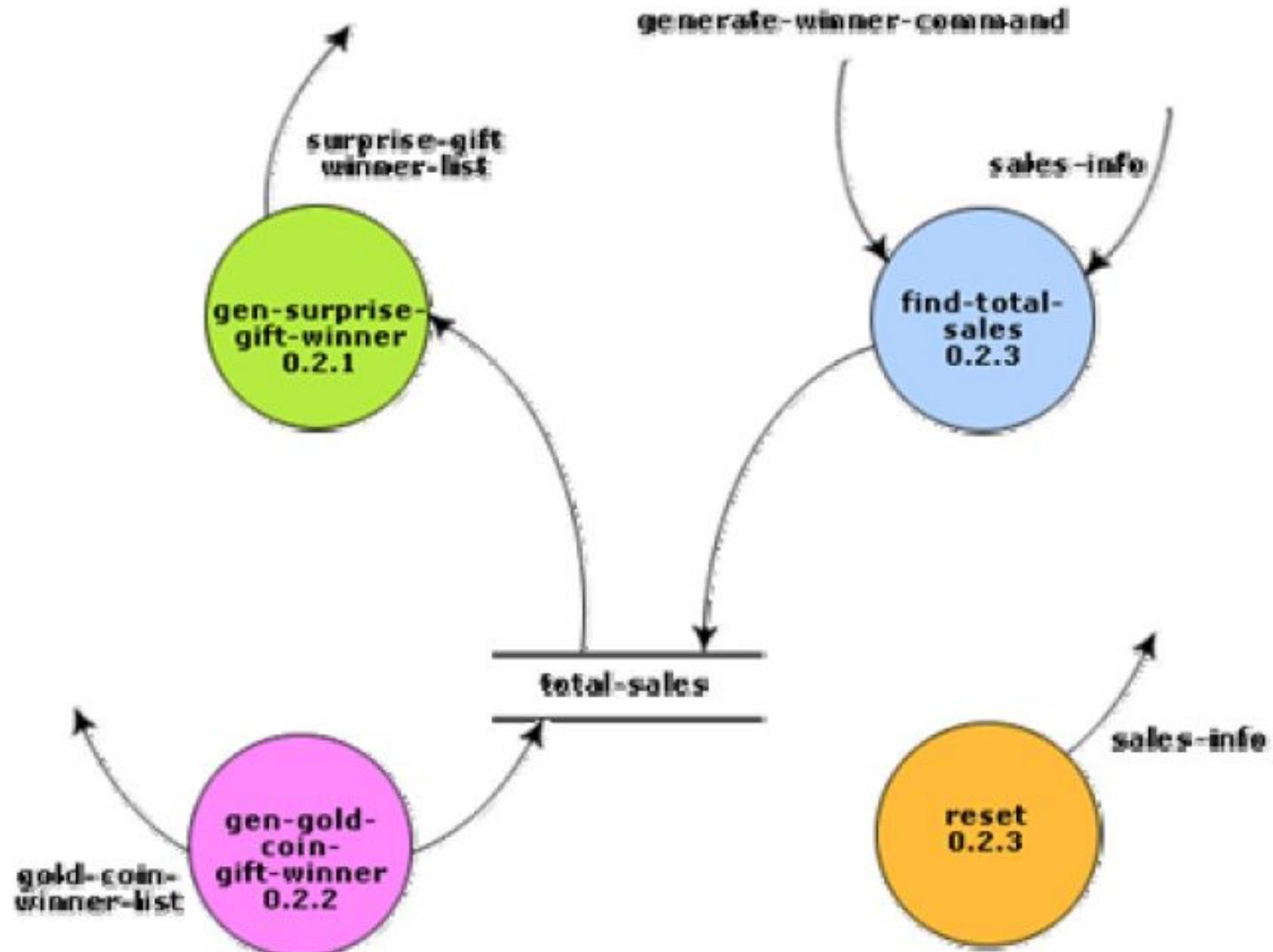
# Level 0



# Level 1

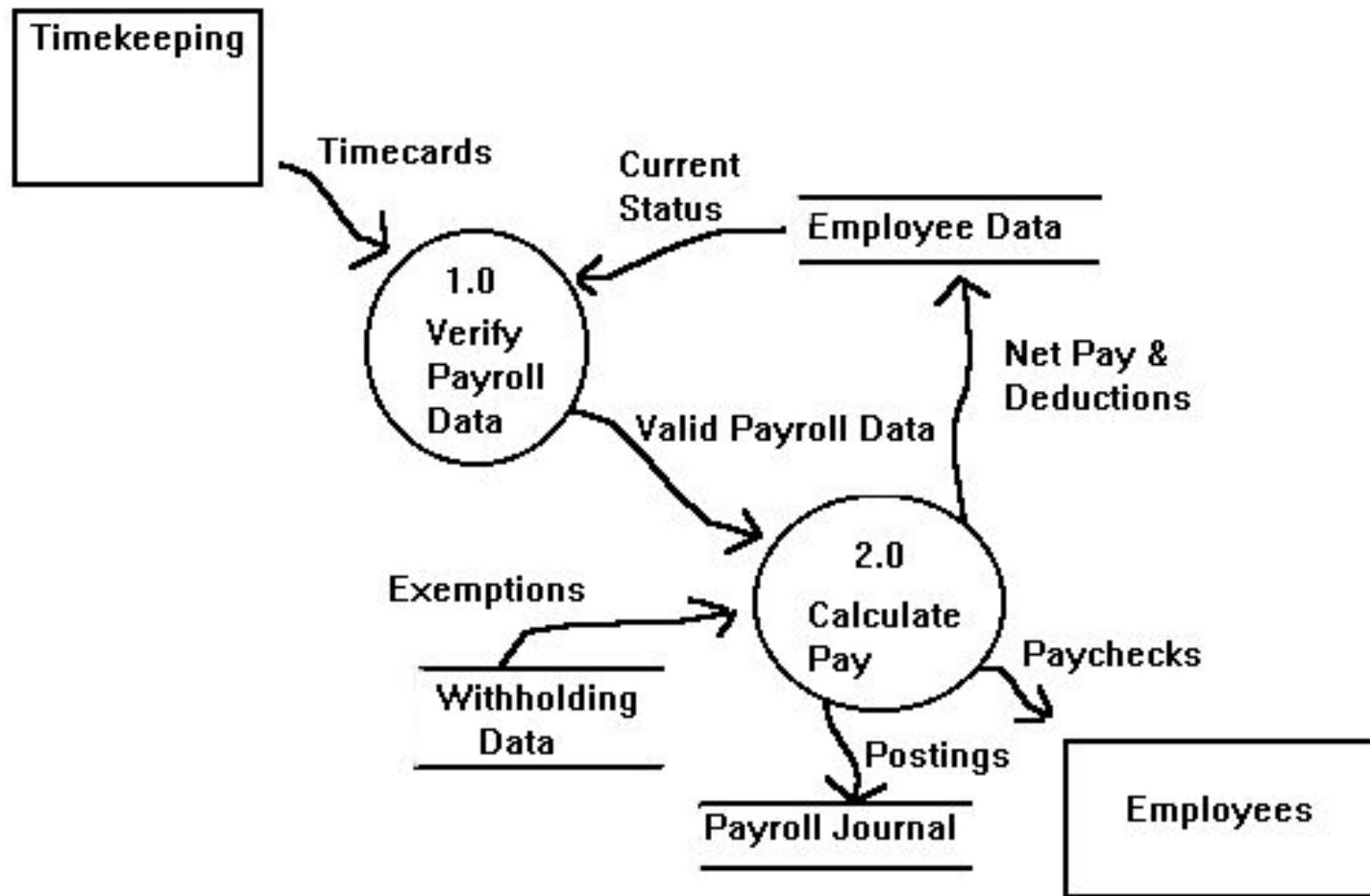


# Level 2

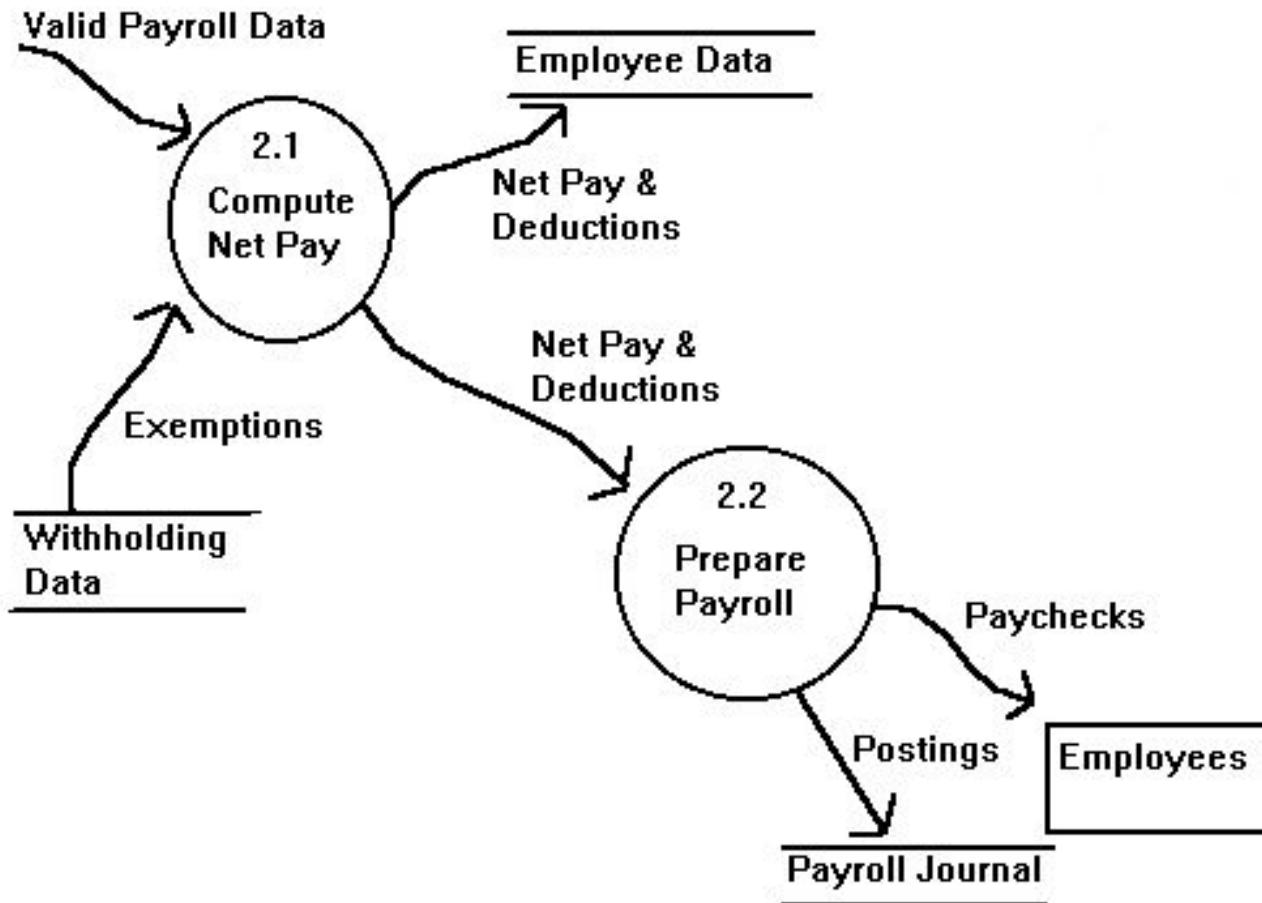




# Level 1 :Payroll System

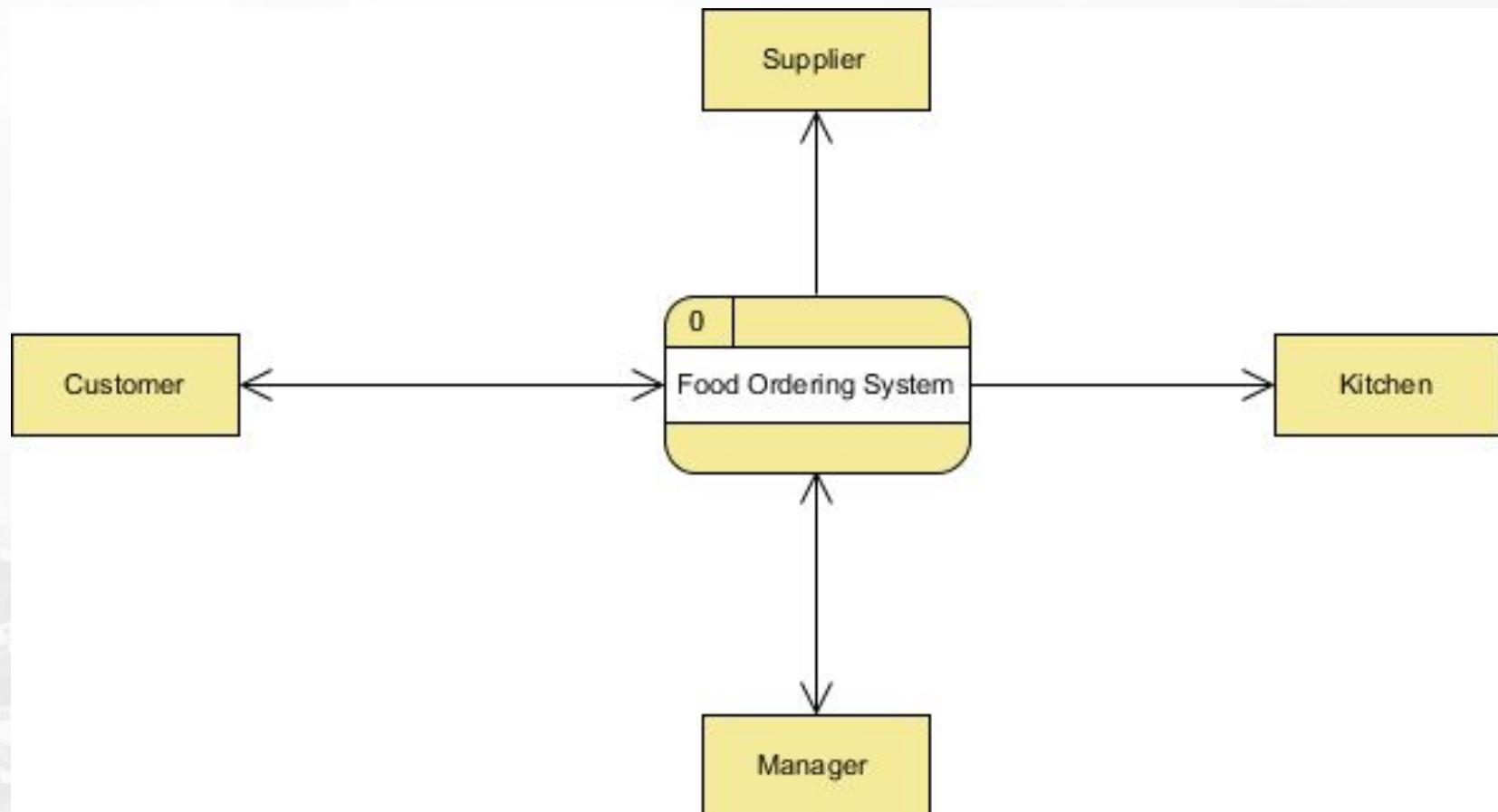


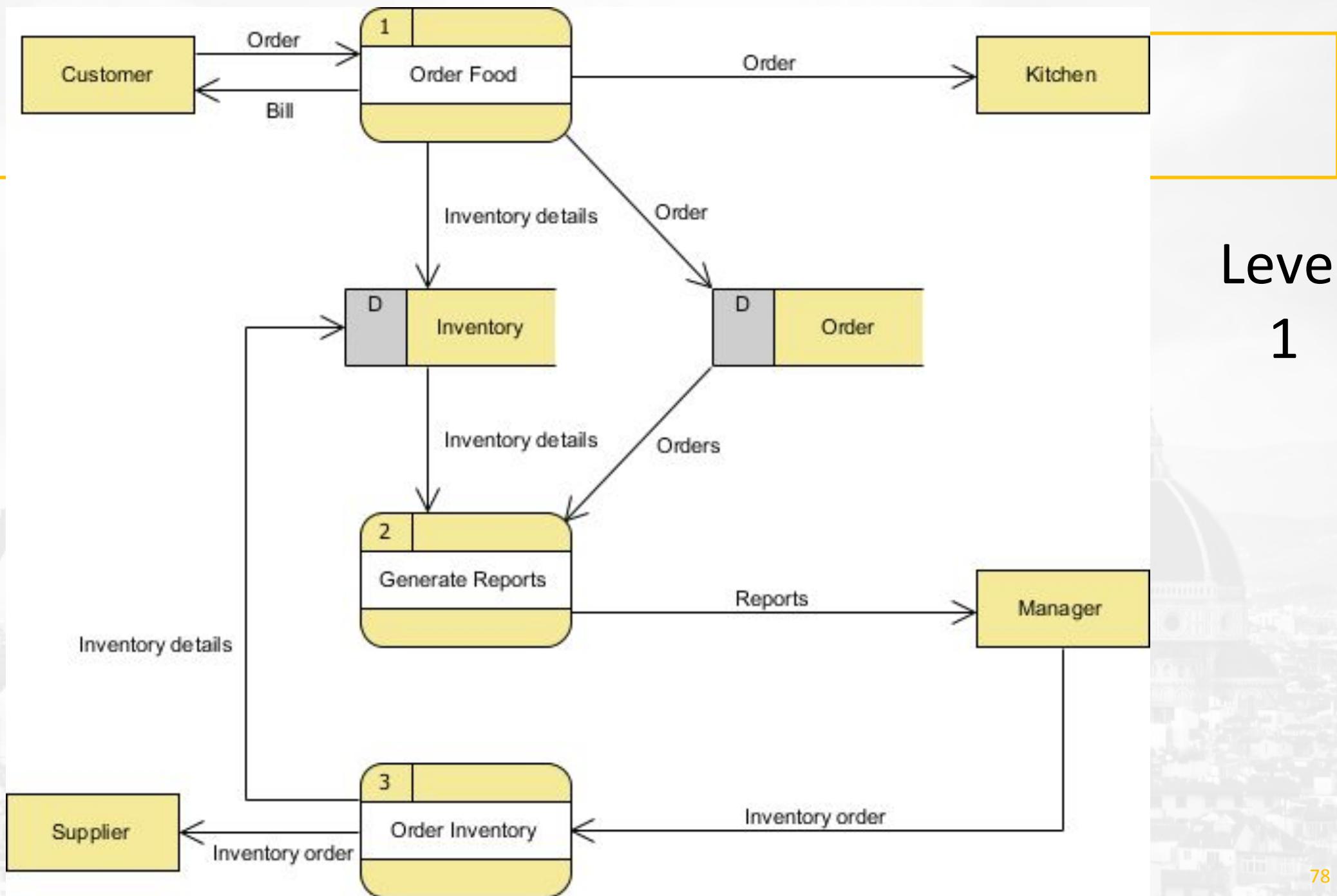
# Level 2 :Payroll System





# The Food Ordering System Example







# 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.

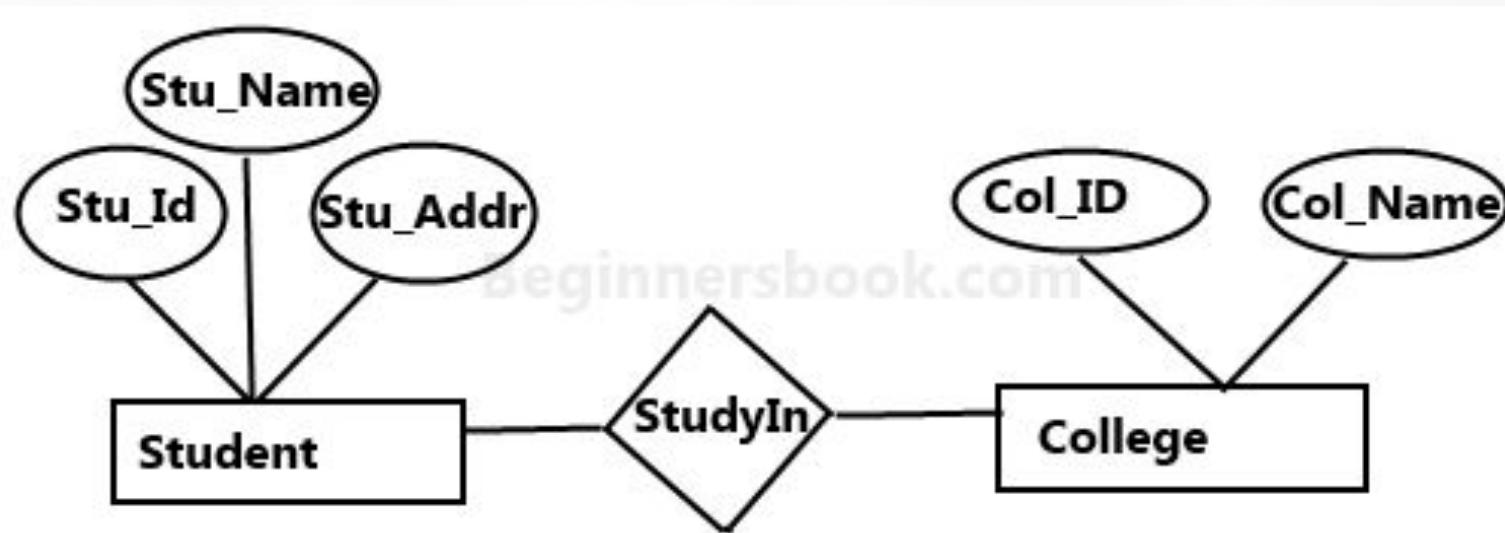
# ER Diagram Notations

- **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.

Entity

Relationship

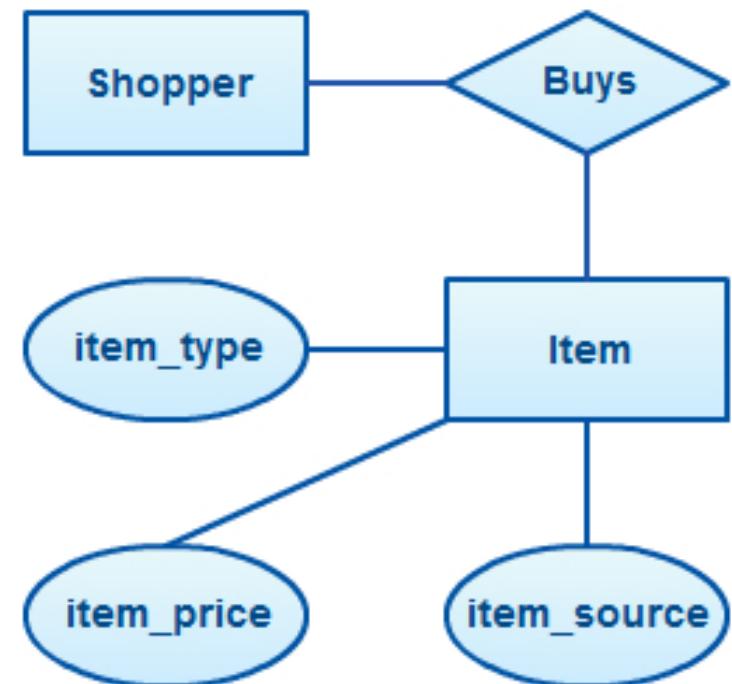
Attribute



**Sample E-R Diagram**

# Example 1

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

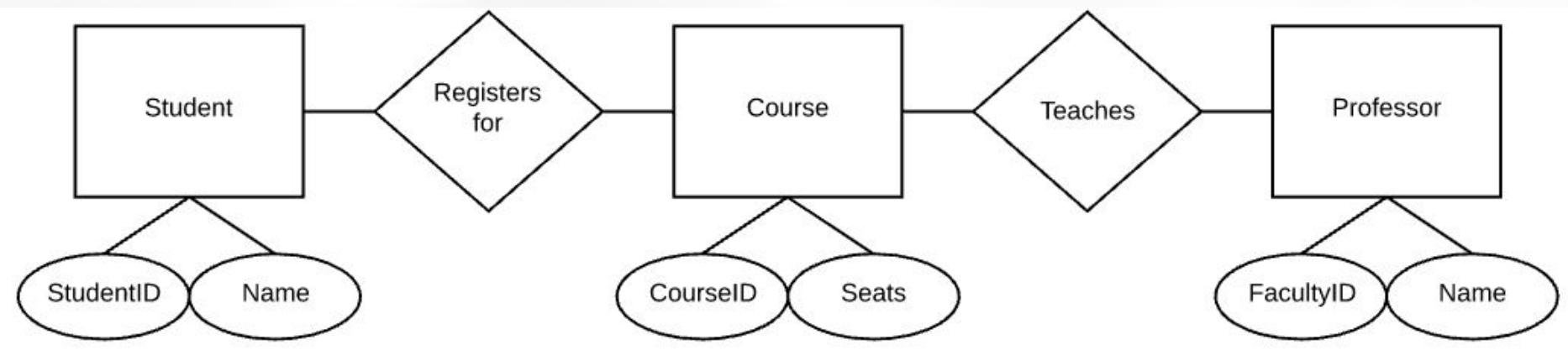


## Example 2

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

## Example 2

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

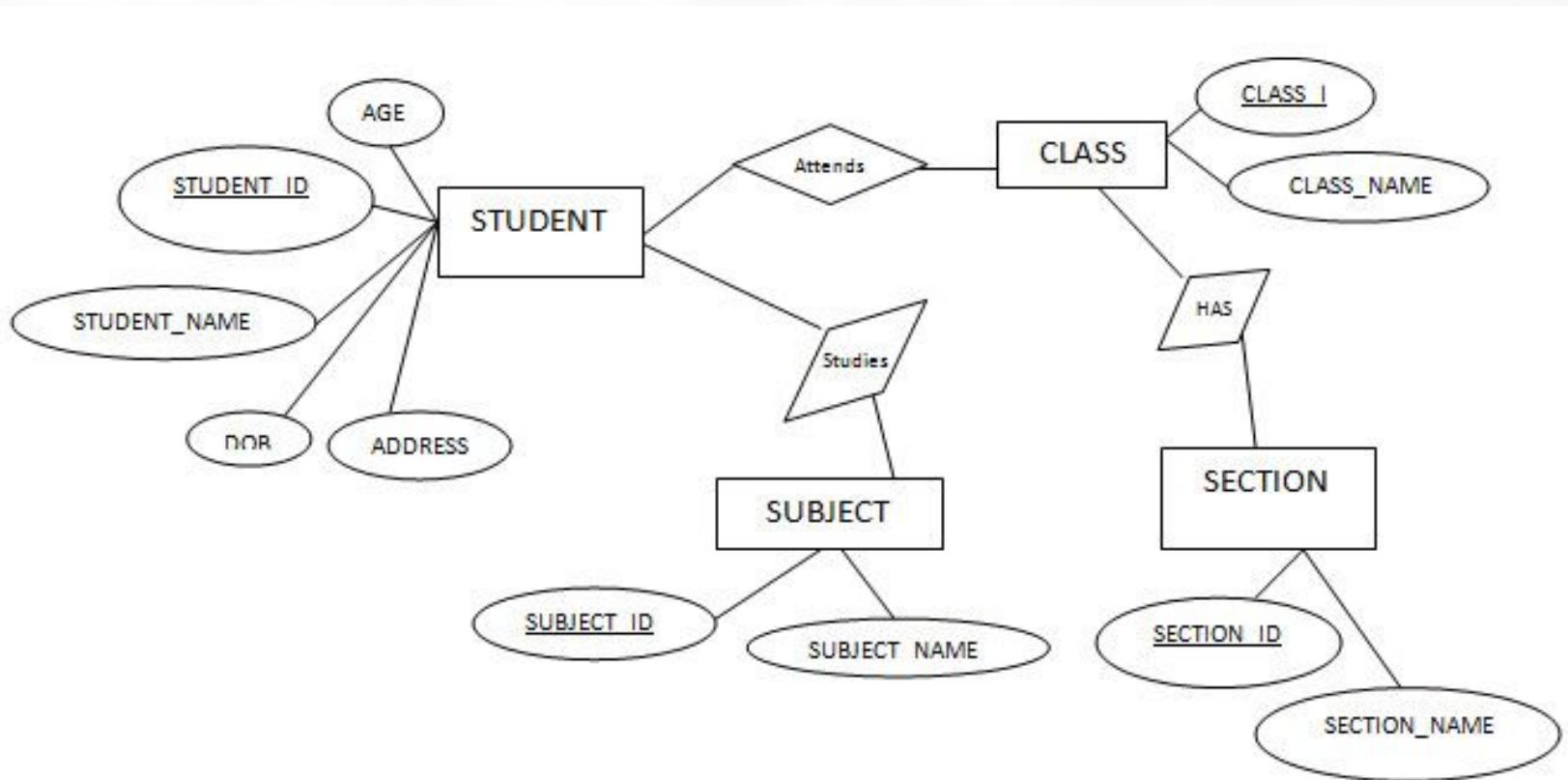


# Example 3

- 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 <b>data</b> rather than the procedures as in Structured Analysis.	Functionality is restricted within <b>objects</b> . 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

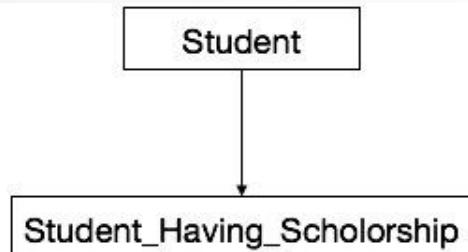
- Consider a class - Circle, that represents the geometrical figure circle in a two-dimensional space.
- The attributes of this class can be identified as follows –
  - x-coord, to denote x-coordinate of the center
  - y-coord, to denote y-coordinate of the center
  - a, to denote the radius of the circle
- 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

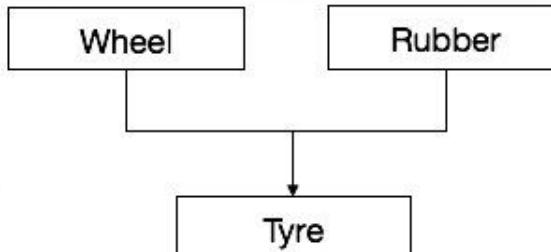
- Any application requires a number of objects interacting in a harmonious manner. Objects in a system may communicate with each other using message passing.
- The features of message passing are –
  - Message passing between two objects is generally unidirectional.
  - Message passing enables all interactions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

# Inheritance

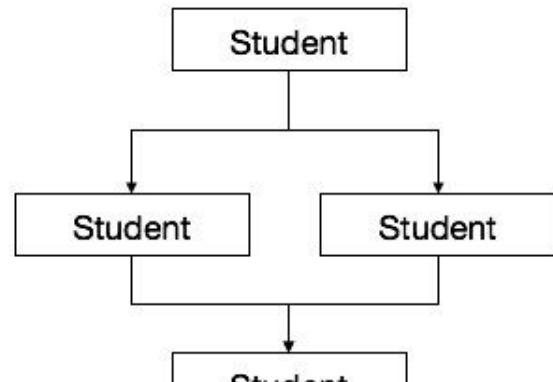
- Is the mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities.



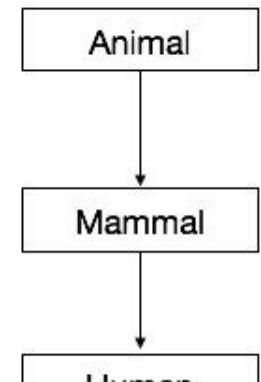
**Single Inheritance**



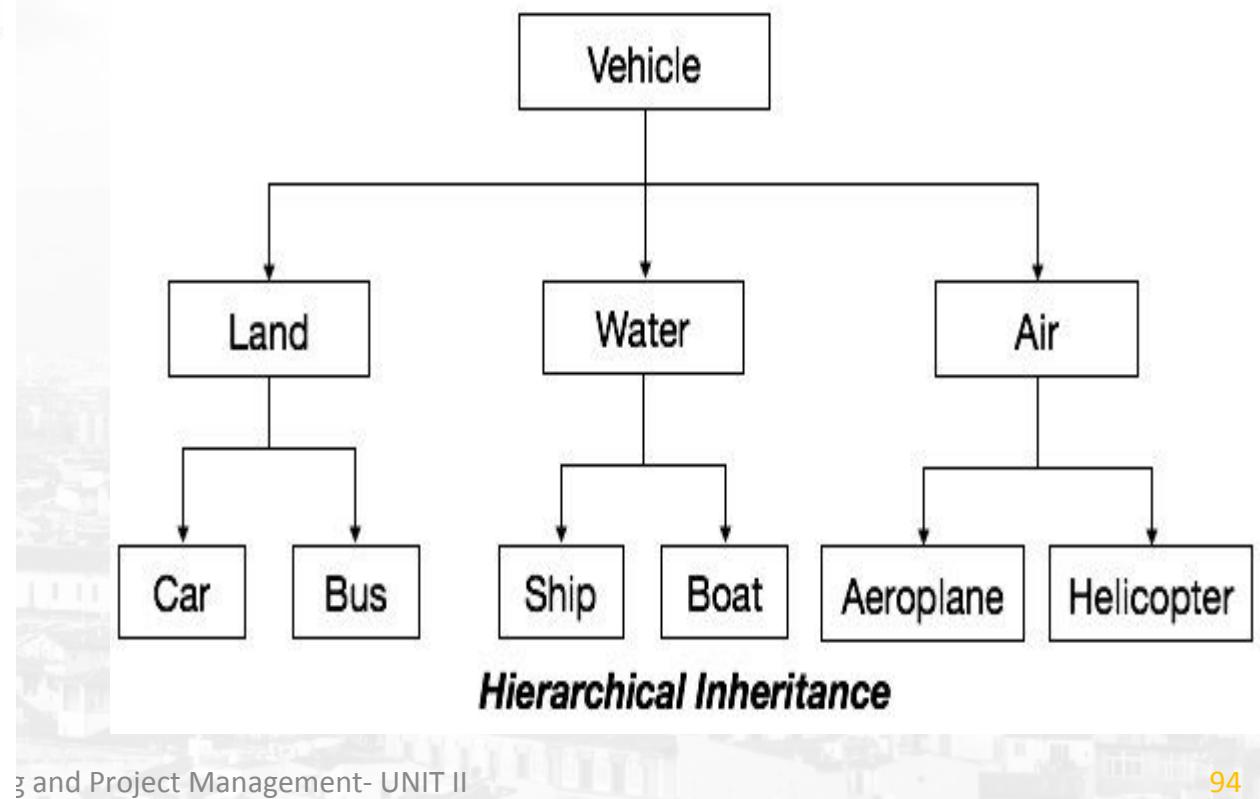
**Multiple Inheritance**



**Hybrid Inheritance**

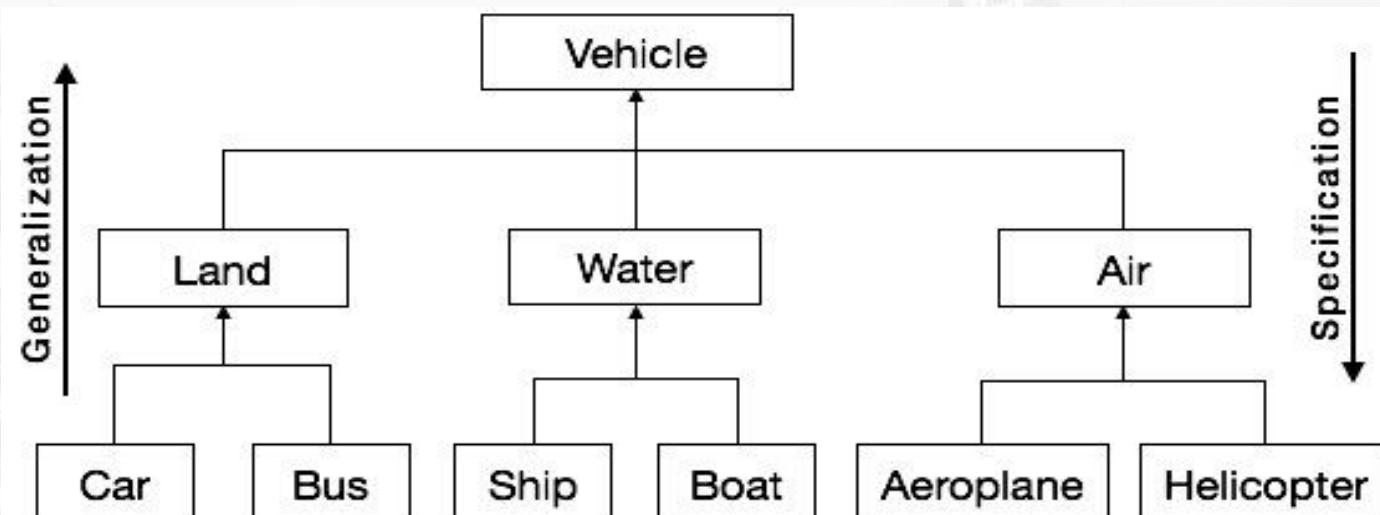


**Multi-level Inheritance**



# 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?

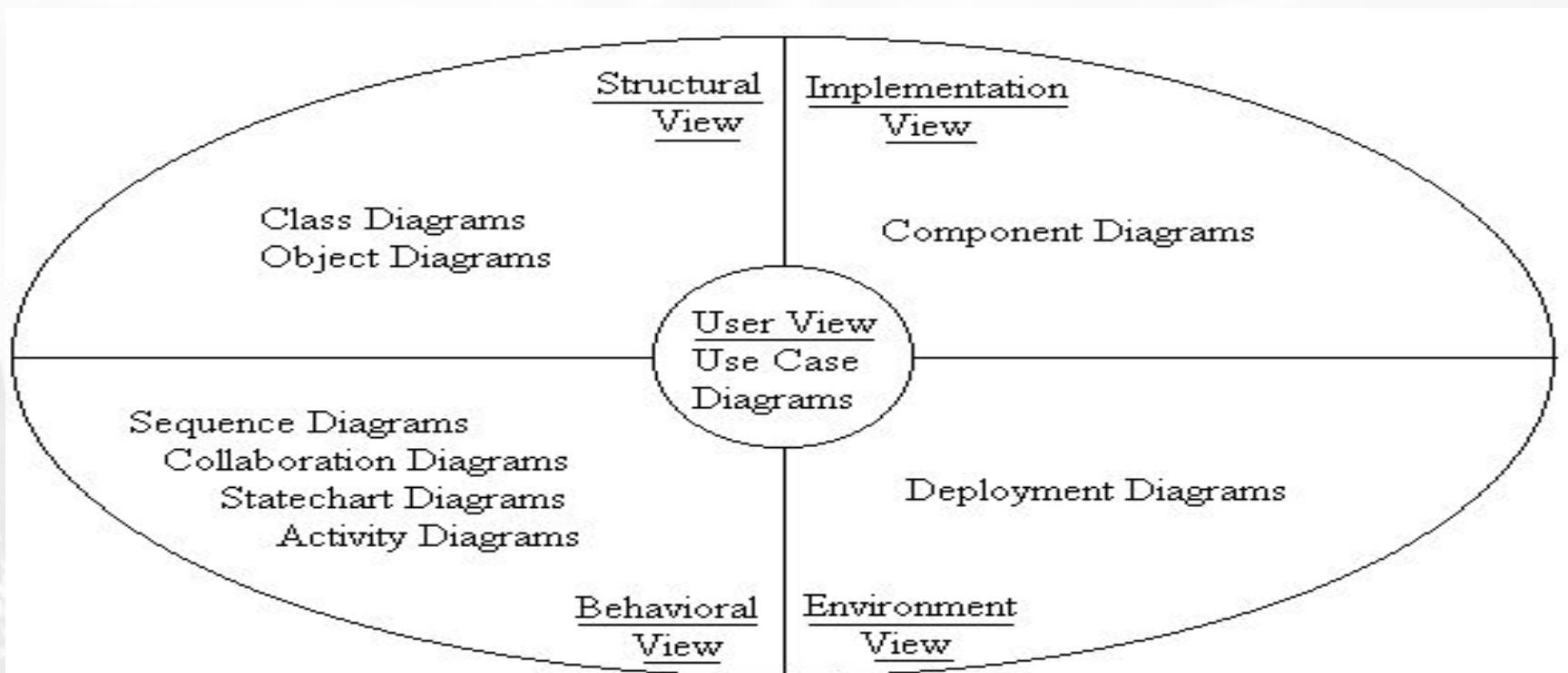
- UML (Unified Modeling Language)
  - An emerging standard for modeling object-oriented software.
  - Resulted from the convergence of notations from three leading object-oriented methods:
    - OMT (James Rumbaugh)
    - OOSE (Ivar Jacobson)
    - Booch (Grady Booch)
- Reference: “The Unified Modeling 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

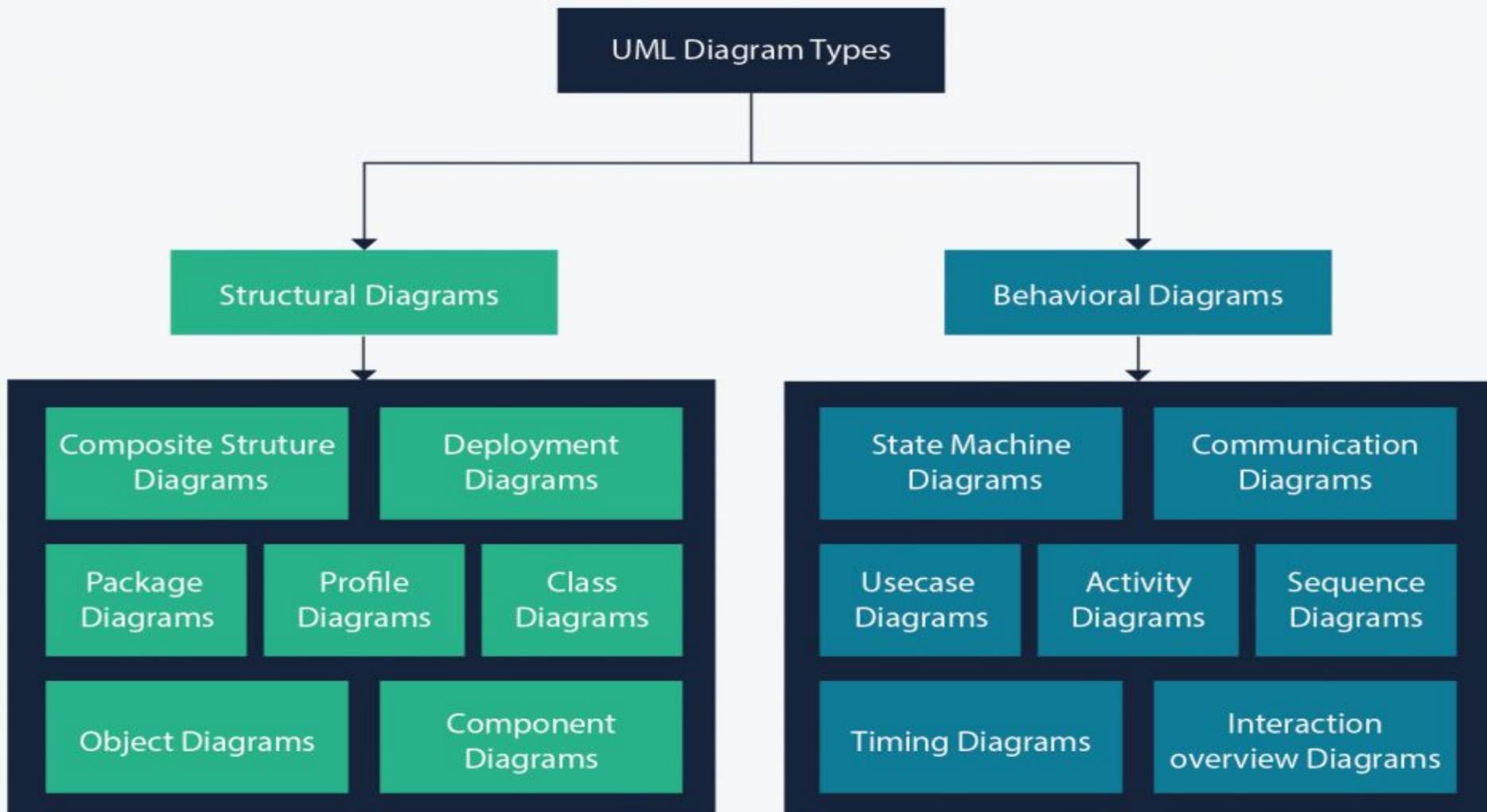
- UML Meta Model: A diagram that defines the notation to be used in the modeling language



# 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

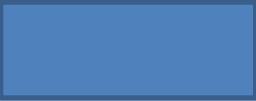
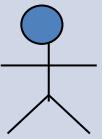
# Use Case Diagram

- 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

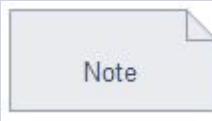
# Use Case Diagram

- 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

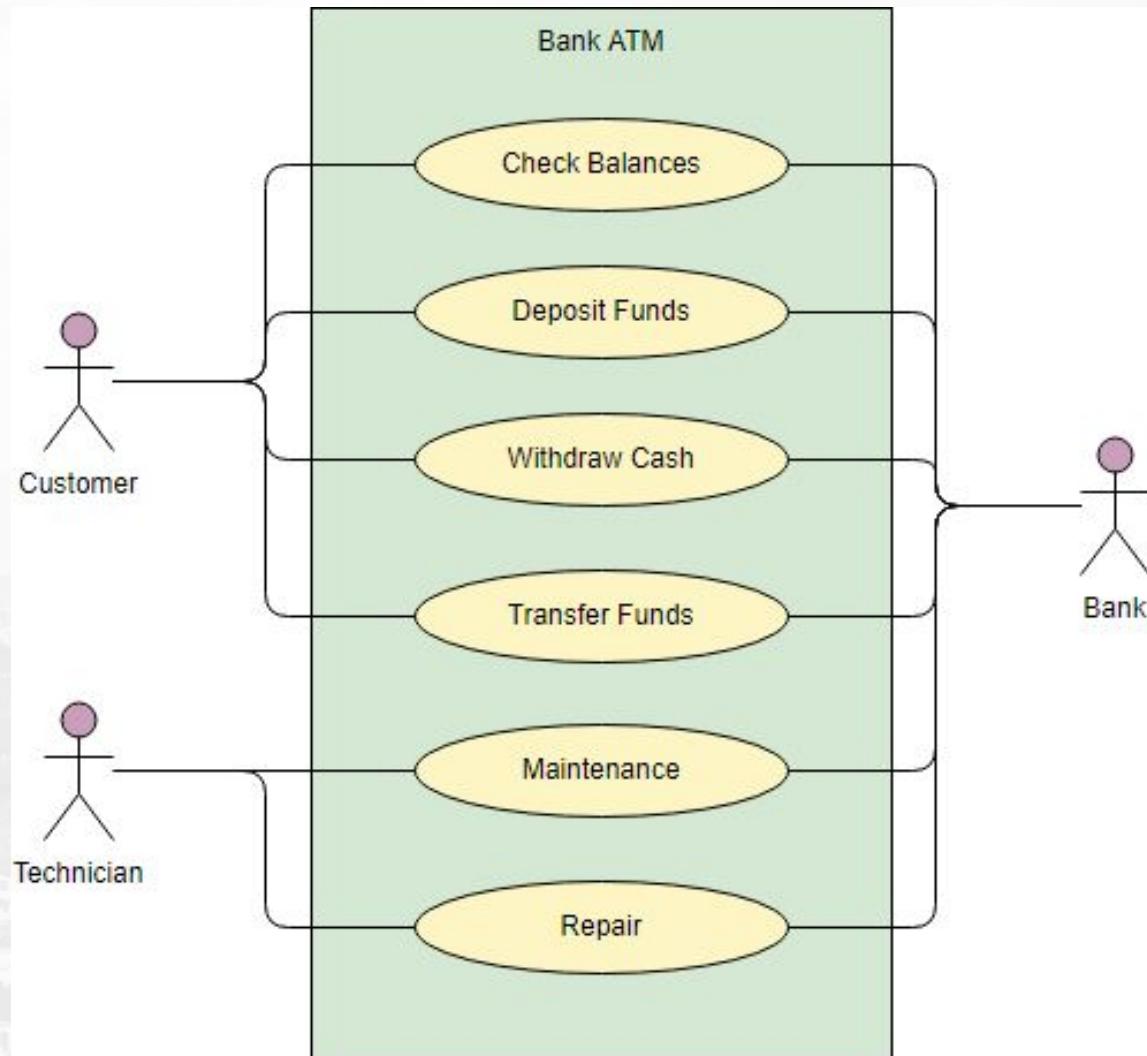
S.No	Name	Description	Notation
6	Include	It specifies how the behavior of the inclusion use case is inserted into the behavior defined for the base use case	<<include>> ----->
7	Extend	How the behavior of the extension use case can be inserted into the behavior defined for the base use case	<<extend>> ----->
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).

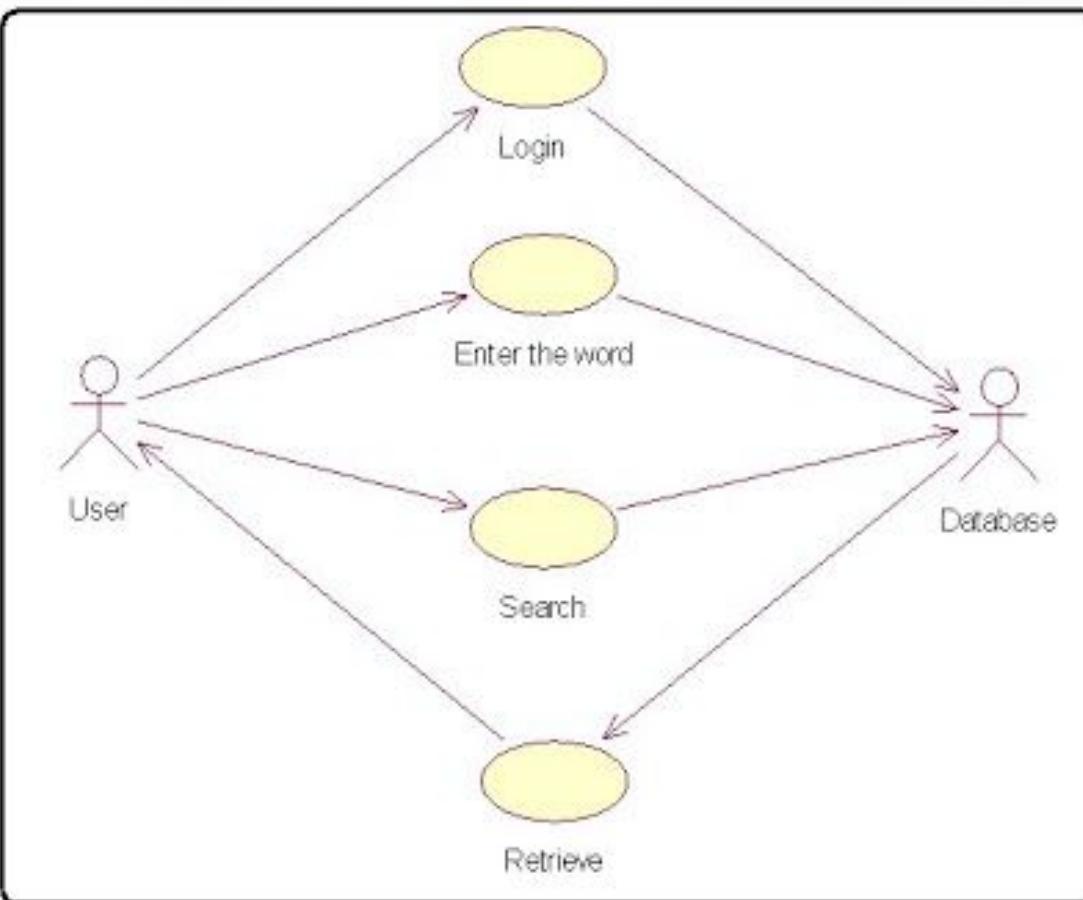
Careful: Lines between Actors and Use Cases are NOT data flows!

# 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.

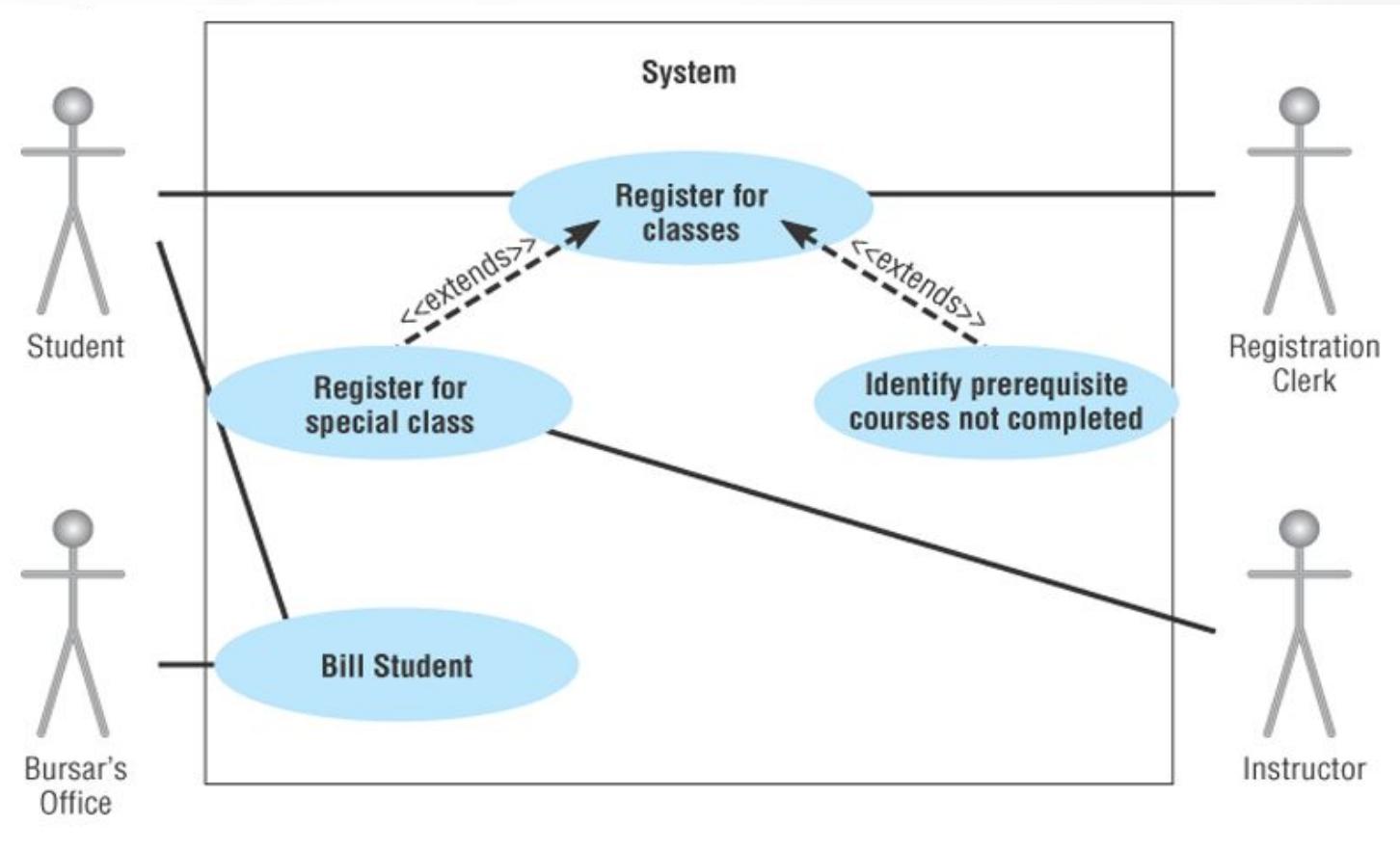


# Extend Relation between Use Cases

- 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



# Include Relation between Use Cases

- Include relationship –

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

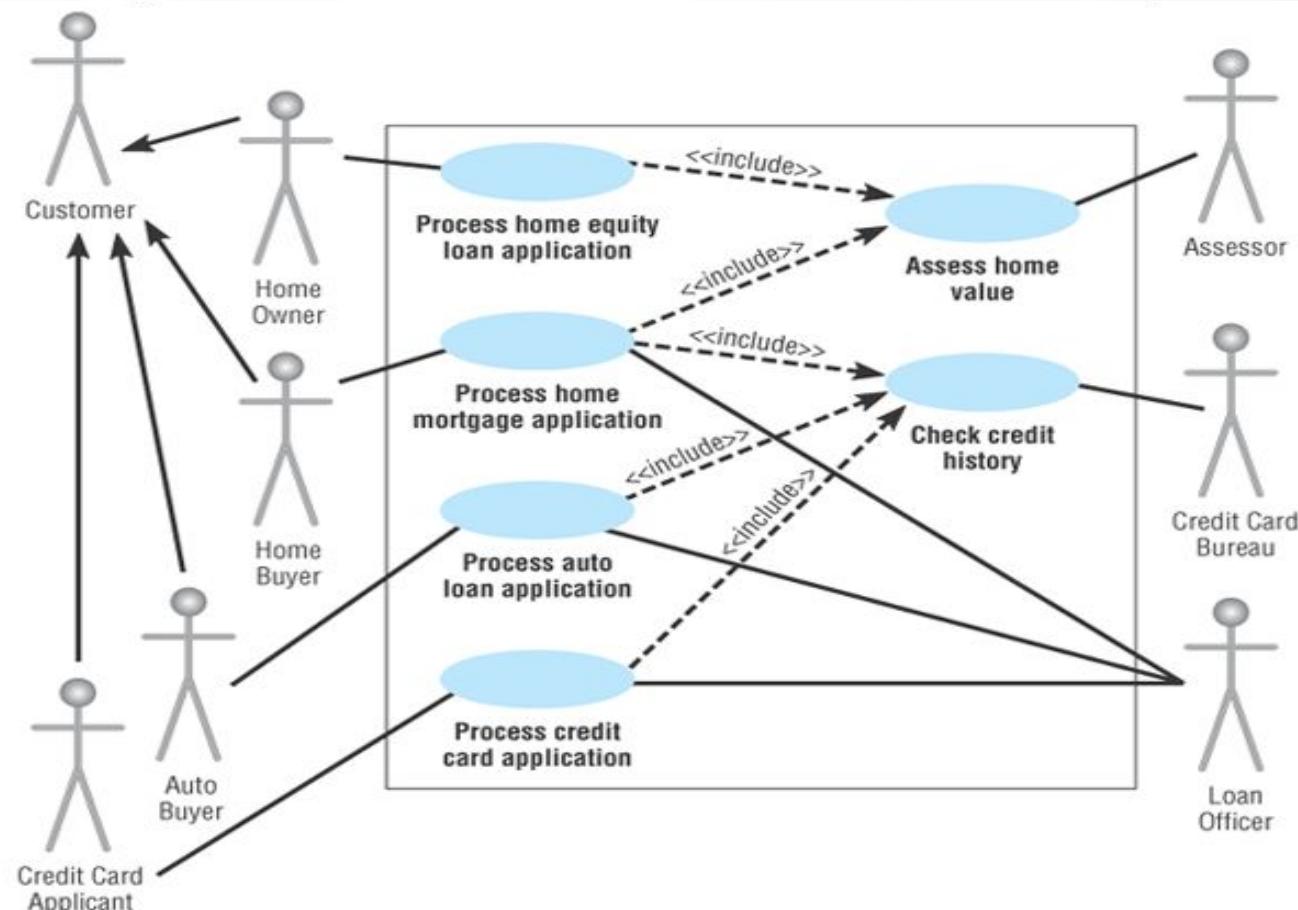
Example: to *Authorize Car Loan* (standard use case), a clerk must run *Check Client's Credit History* (include use case).

tight coupling

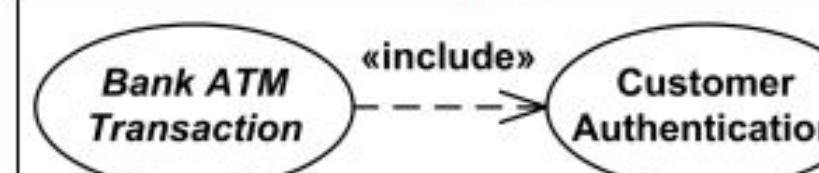
- Standard use case can NOT execute without the include case.

# Extend Relation between Use Cases

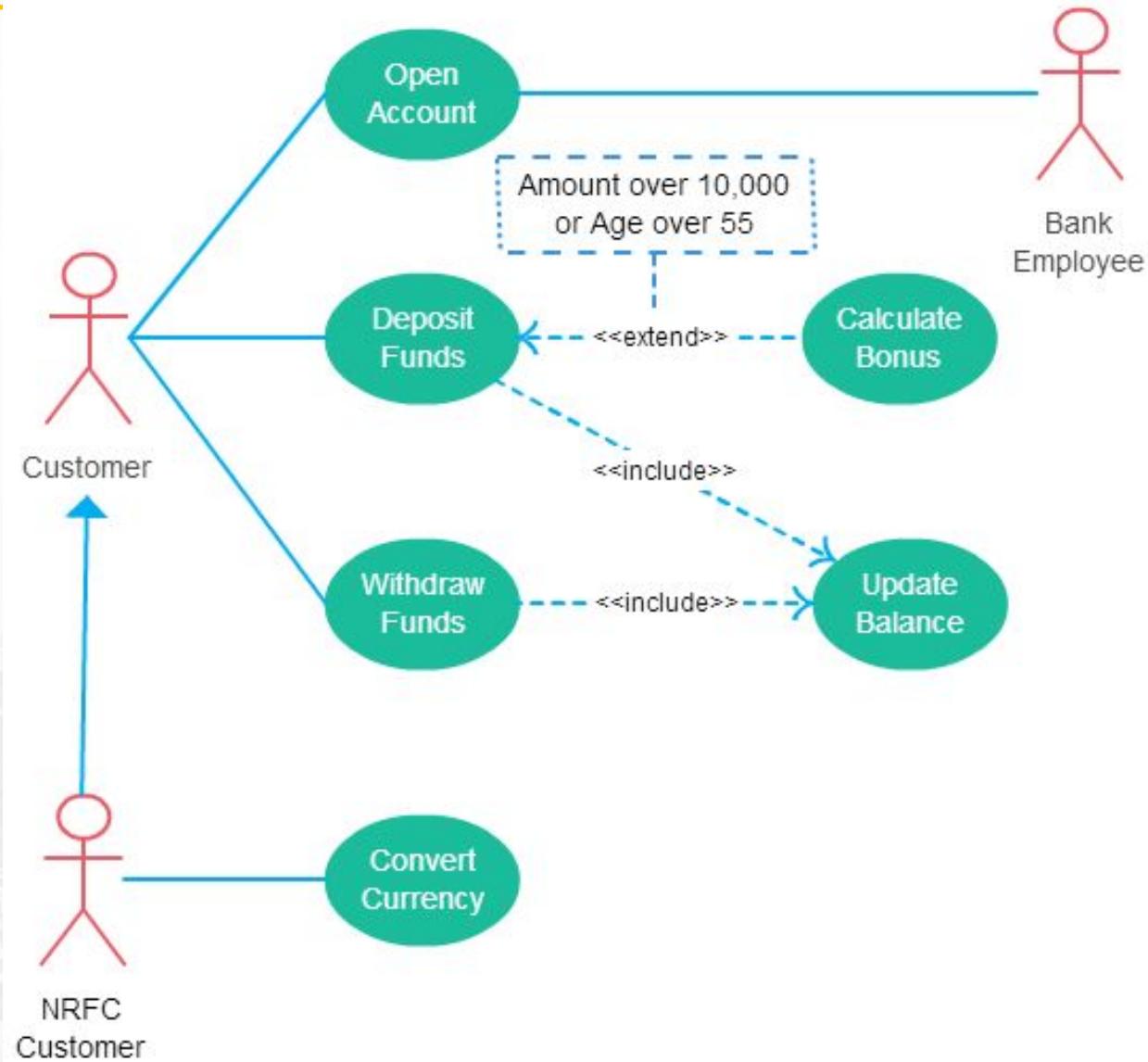
## • Credit Card Processing



# Extend and Include

Extend	Include
	
Base use case is complete (concrete) by itself, defined independently.	Base use case is incomplete ( <b>abstract use case</b> ).
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.

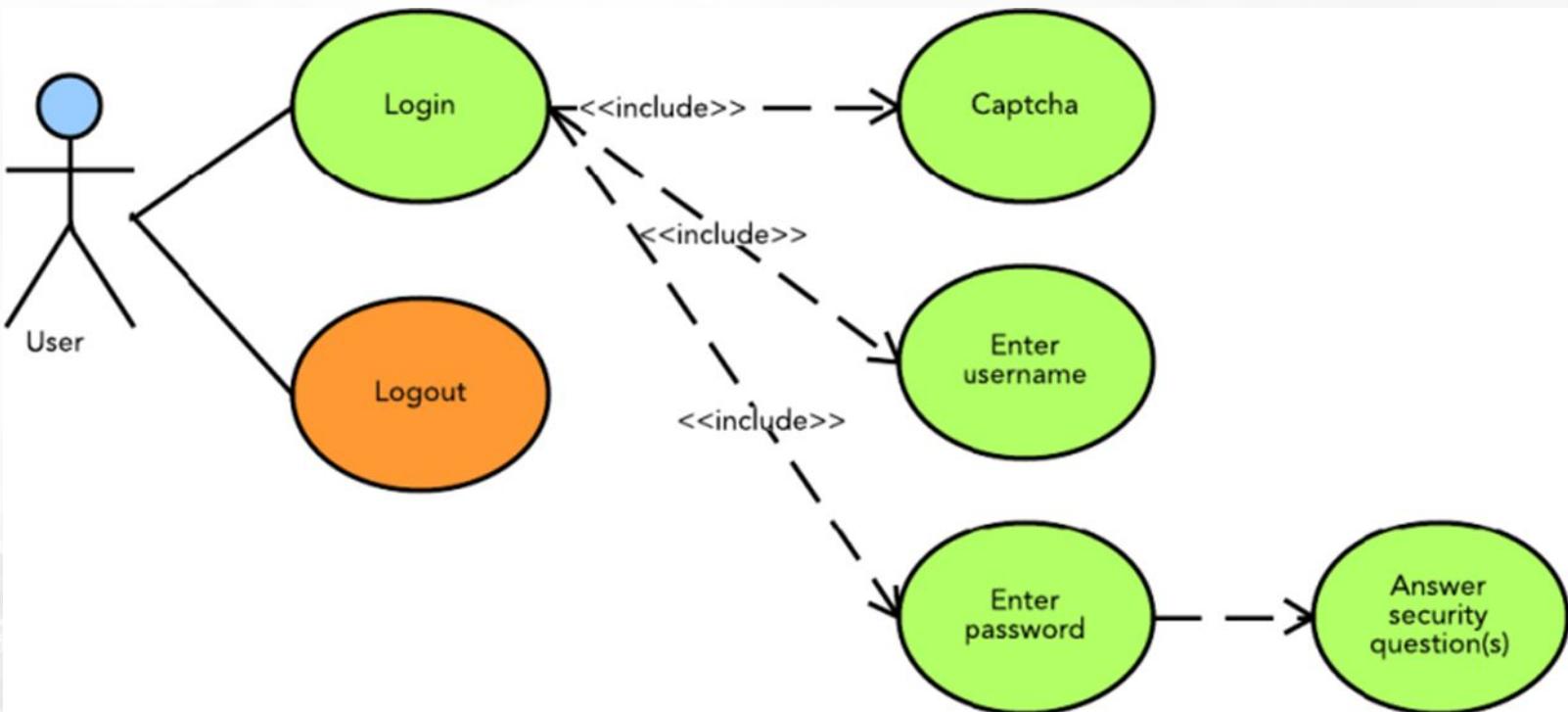
# Another example



# Use Case Example

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

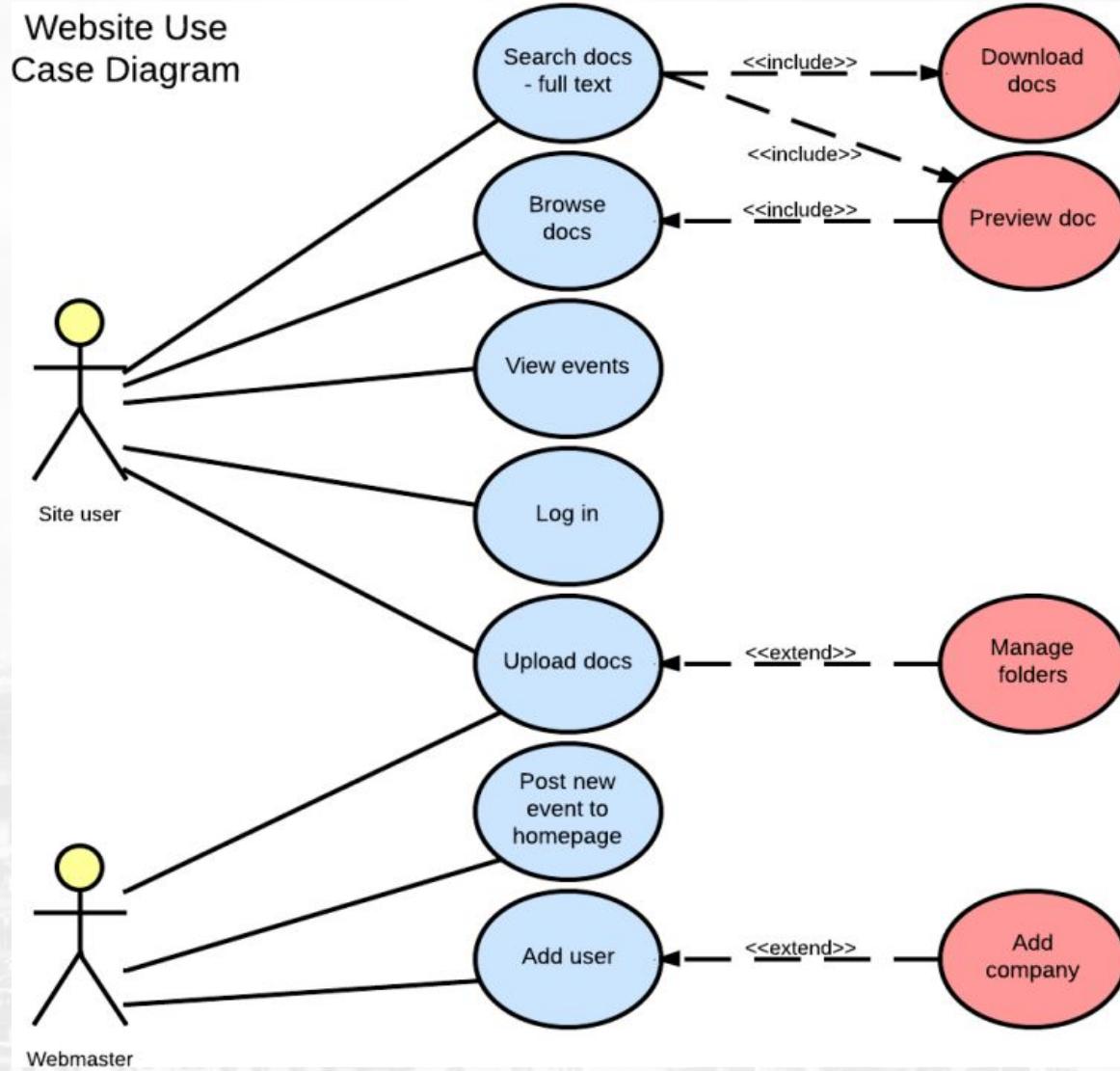
# Example : Login



# 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

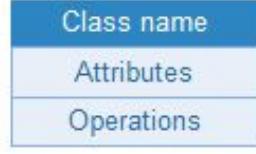
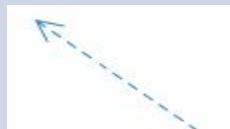
# Use Case Diagram for Website Use Case



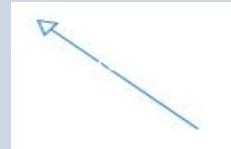
# Class Diagram

- 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	Object: 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		<table border="1"> <thead> <tr> <th colspan="2">Multiplicity</th> </tr> <tr> <th>Symbol</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>One and only one</td> </tr> <tr> <td>0..1</td> <td>Zero or one</td> </tr> <tr> <td>M..N</td> <td>From M to N (natural language)</td> </tr> <tr> <td>*</td> <td>From zero to any positive integer</td> </tr> <tr> <td>0..*</td> <td>From zero to any positive integer</td> </tr> <tr> <td>1..*</td> <td>From one to any positive integer</td> </tr> </tbody> </table>	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
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

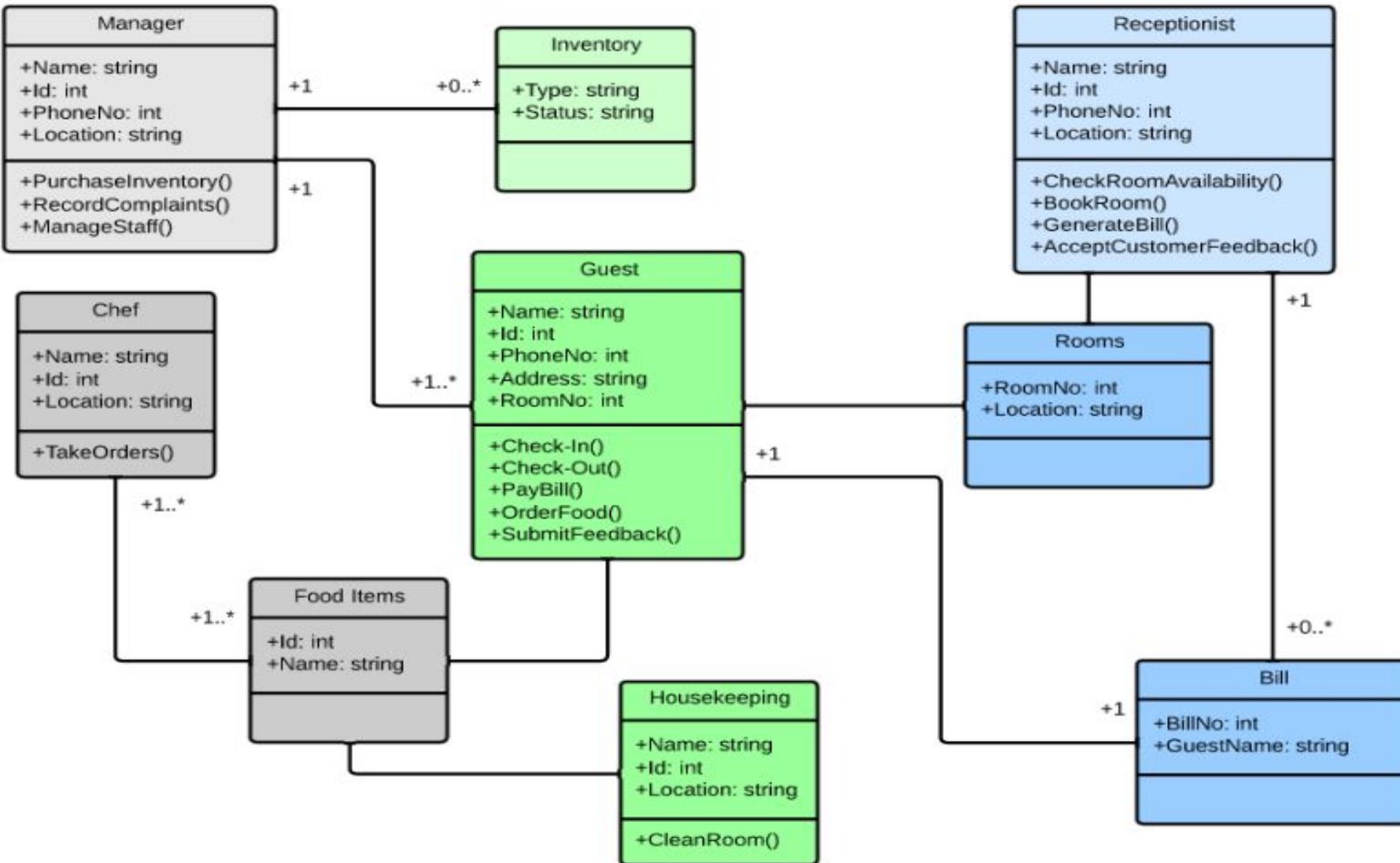
# getID(): int  
~ getEmail(): String

Class name

Attributes

Operations

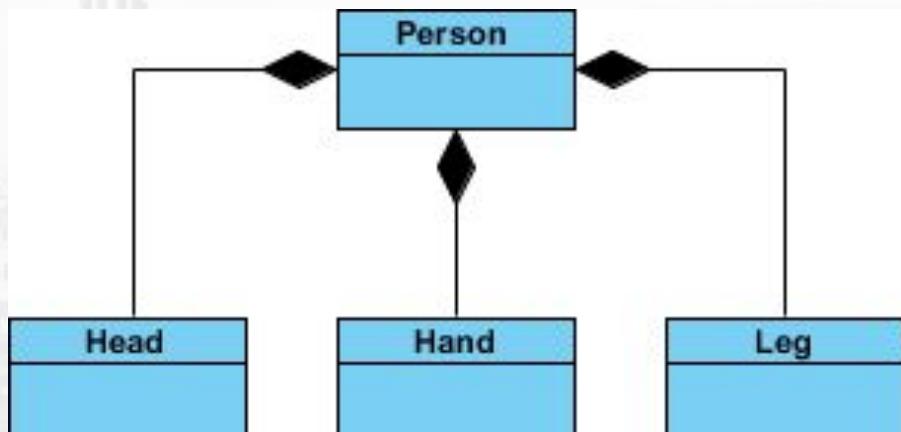
# Class Diagram : Hotel Management System



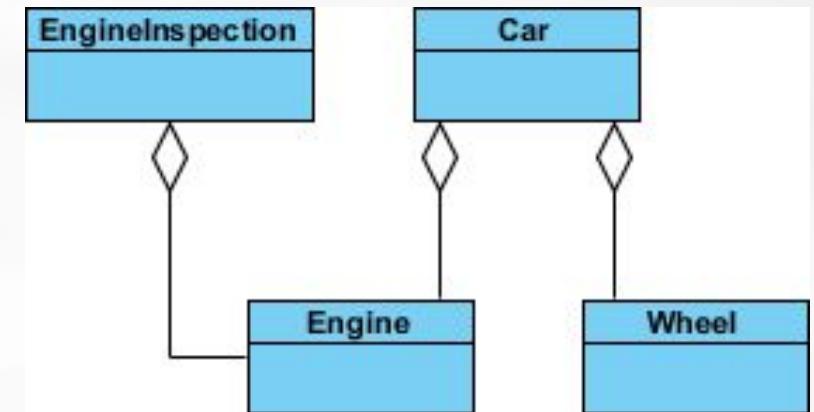
# Types of Relations in Class Diagram



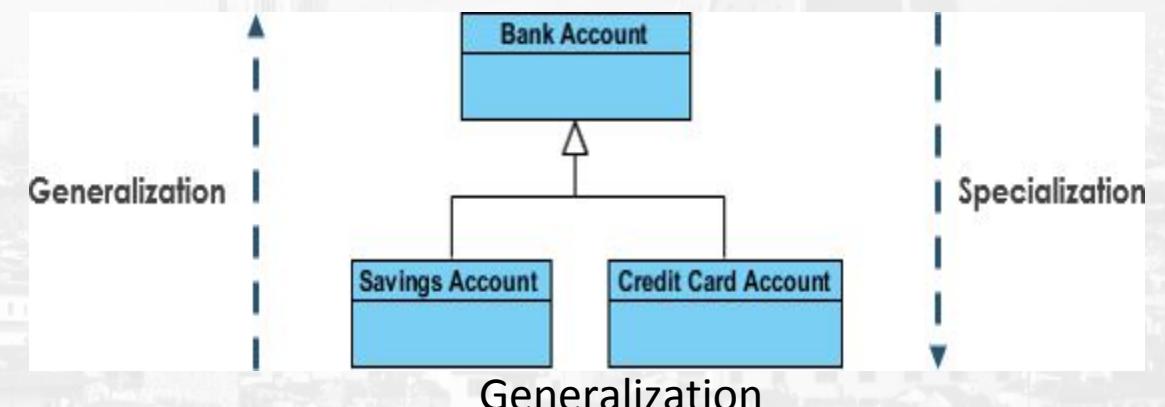
Association



Composition



Aggregation ( exist alone)

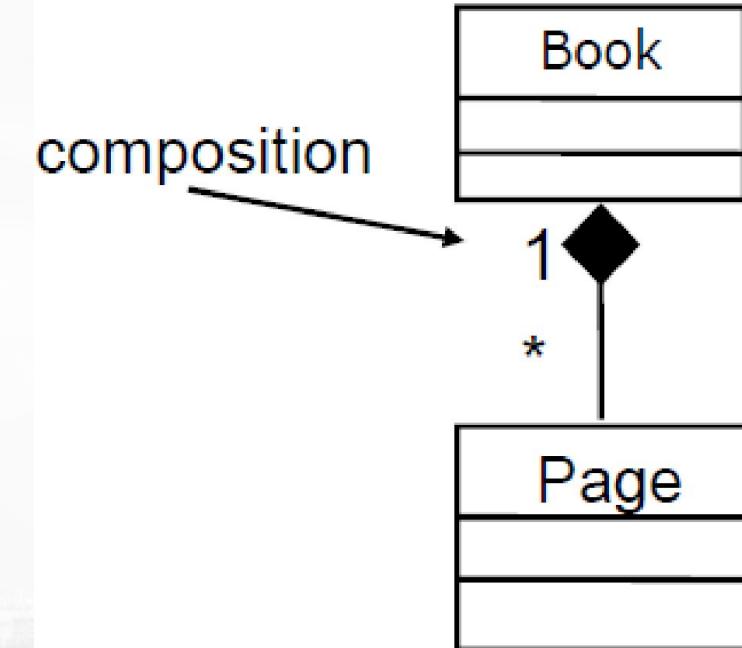
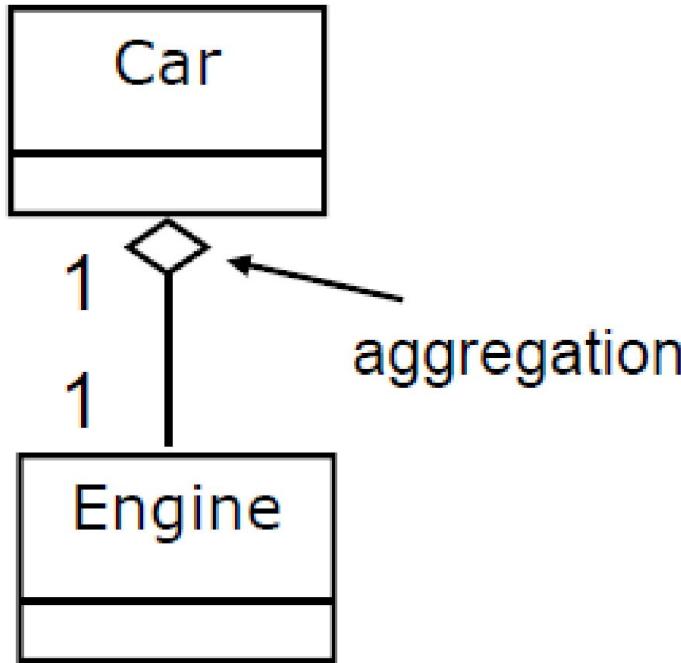


Generalization

Specialization

Generalization

# Another Example



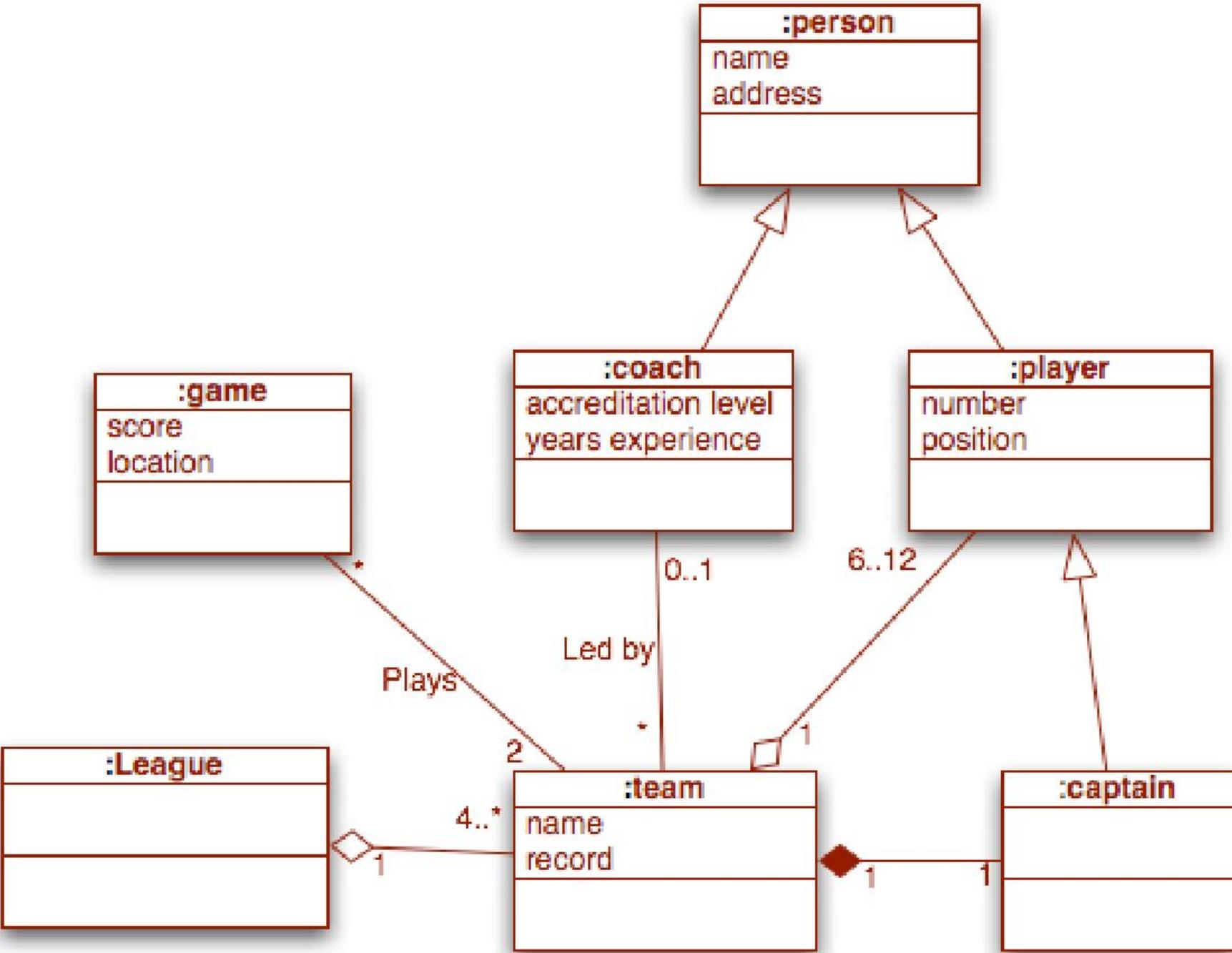
# Example

- 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.

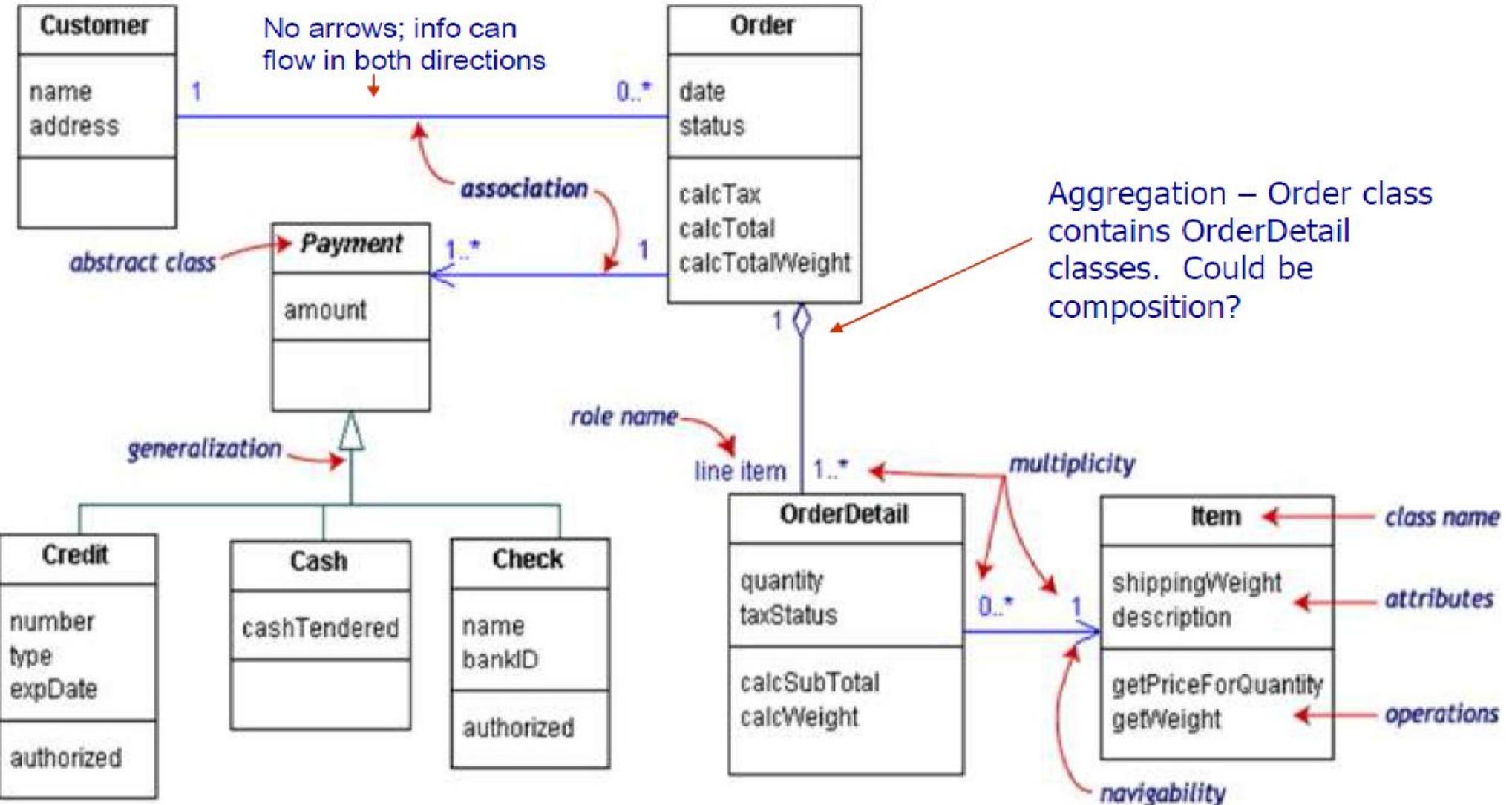


## Example 2

- 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.



# Example : Order Details



# Benefits of class diagrams

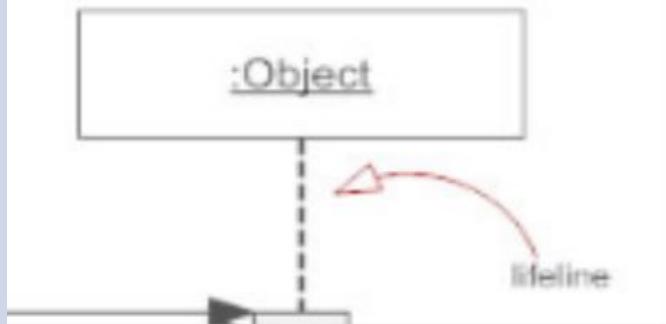
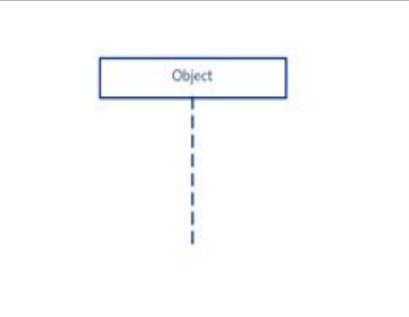
- To understand the general overview of plan of an application.
- Illustrate data models for information systems, no matter how simple or complex.
- Visually express any specific needs of a system
- Create detailed charts that highlight any specific code needed to be programmed and implemented to the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

# UML Sequence Diagrams

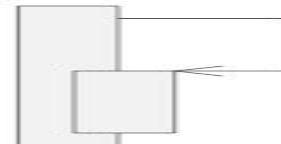
- Sequence diagrams model the dynamic aspects of a software system
- The emphasis is on the “**sequence**” of messages rather than relationship between objects
- Sequence diagrams provide more detail and show the messages exchanged among a set of objects over time.
- The main purpose of this diagram 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	:Object component
2	Activation or Execution Occurrence/Scope	Activation boxes represent the time an object needs to complete a task.	Activation or Execution Occurrence
3	Diagram Boundary		< Diagram's Label > < Diagram's Content Area >

S.No	Name	Description	Notation
3	Messages	Messages are arrows that represent communication between objects.	 <p>A UML sequence diagram element showing a rectangular lifeline labeled ':Object'. A dashed vertical line extends from the bottom of the rectangle. A red curved arrow originates from the right side of the rectangle and points back towards the dashed line, representing a return message.</p>
4	Lifelines	Lifelines represent either roles or object instances that participate in the sequence being modeled.	 <p>A UML sequence diagram element showing a rectangular lifeline labeled 'Object'. A dashed vertical line extends from the bottom of the rectangle, representing the lifetime of an object instance.</p>

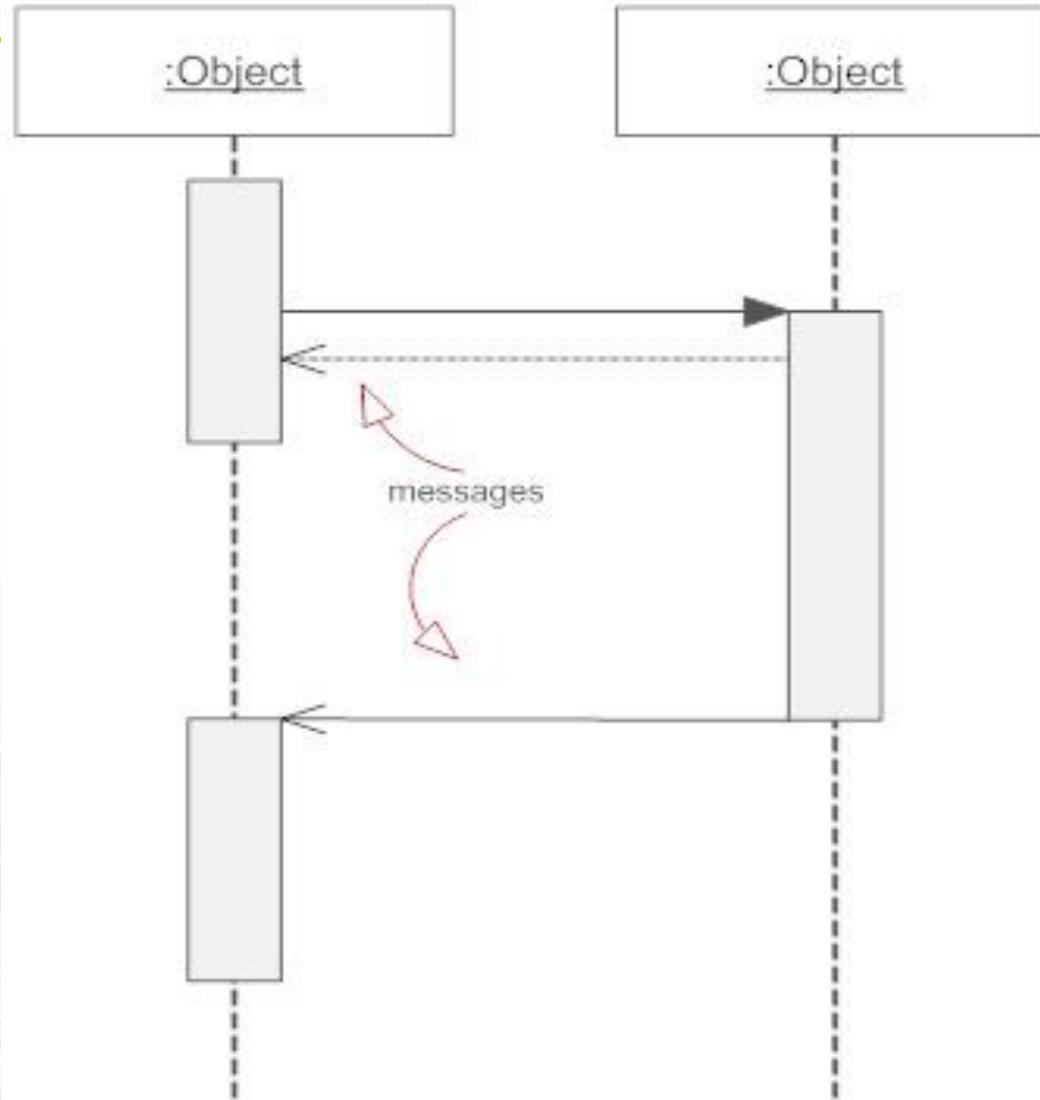
# UML Sequence Diagrams- Types of Messages in Sequence Diagrams

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

# 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





Student

Login  
Screen

Validate  
User

Database

click on login button

ValidateUser(userid,password)

userID:found  
password:matched

classList(grades)

ValidateUser(passwordMatched)

loginSuccessful:message

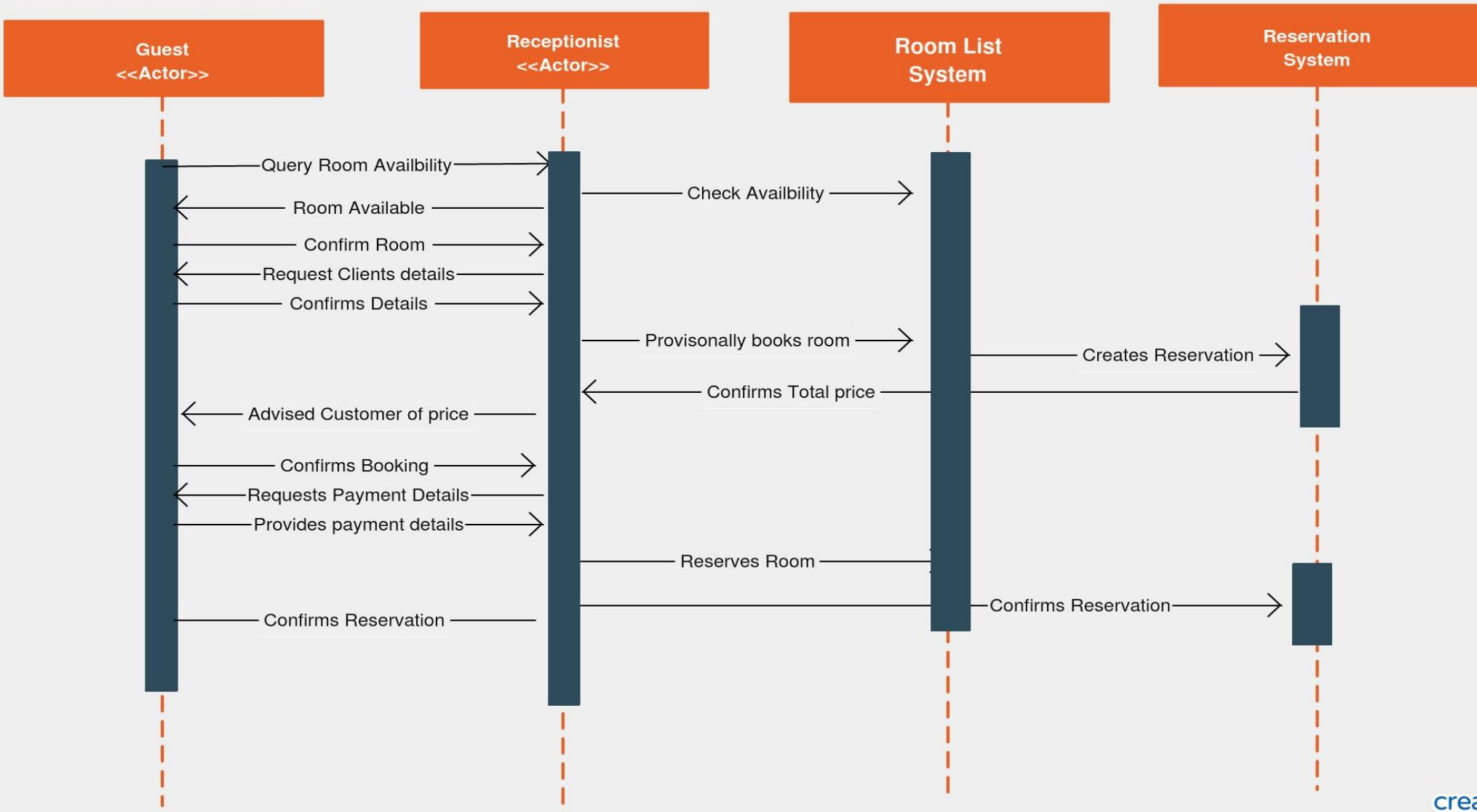
User login successful

class result  
eased  
(| ly)

display:classList(grades)

text

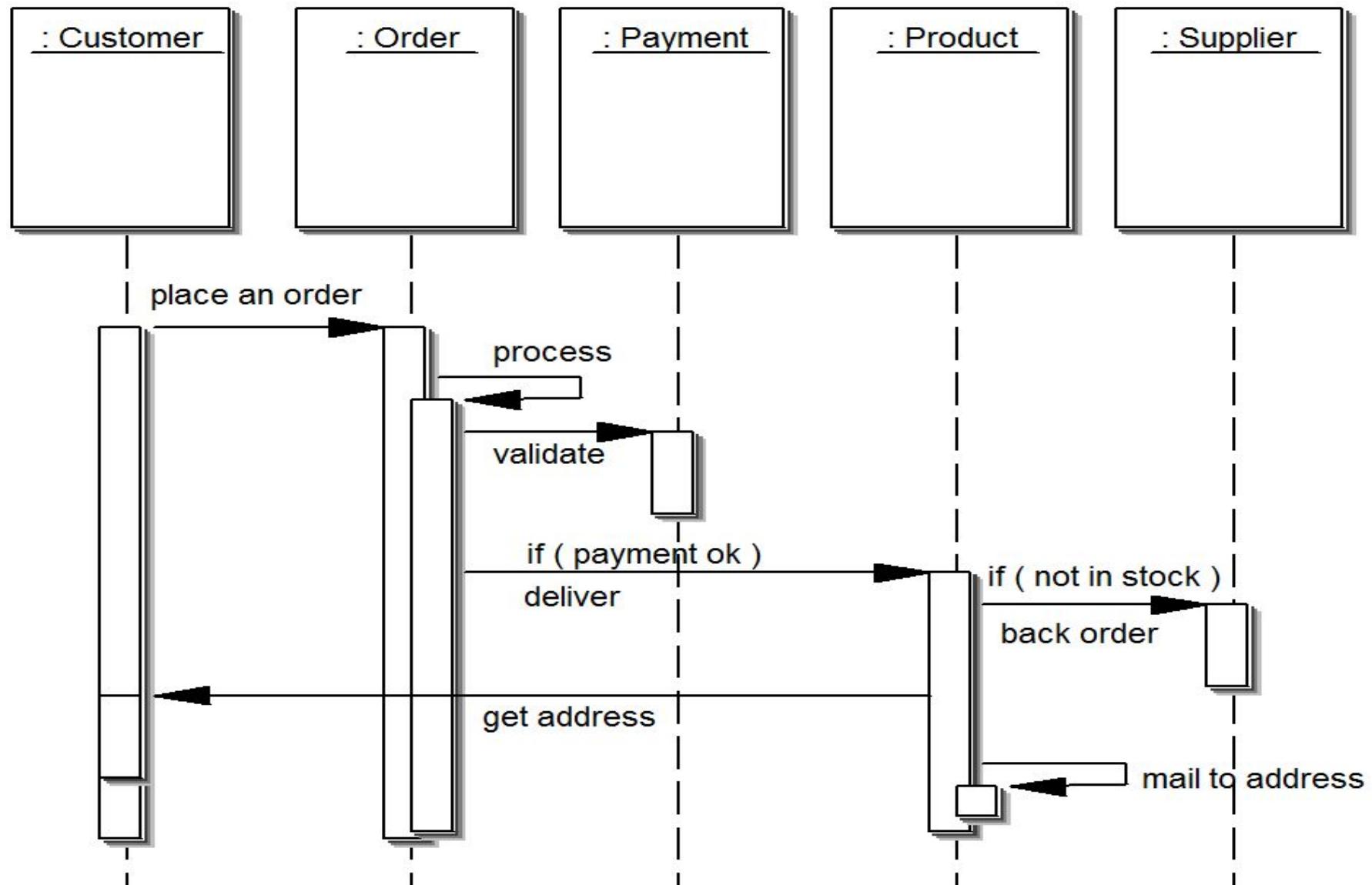
# HOTEL RESERVATION SYSTEM



# 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.

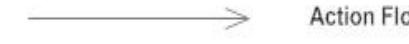
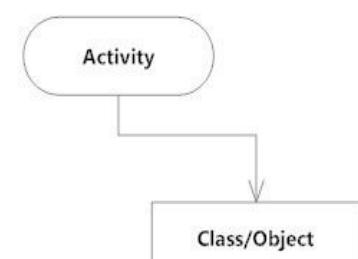
# Order System



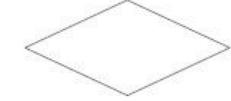
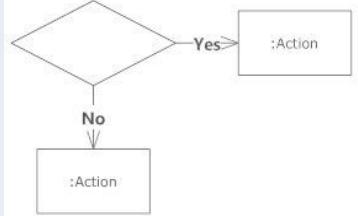
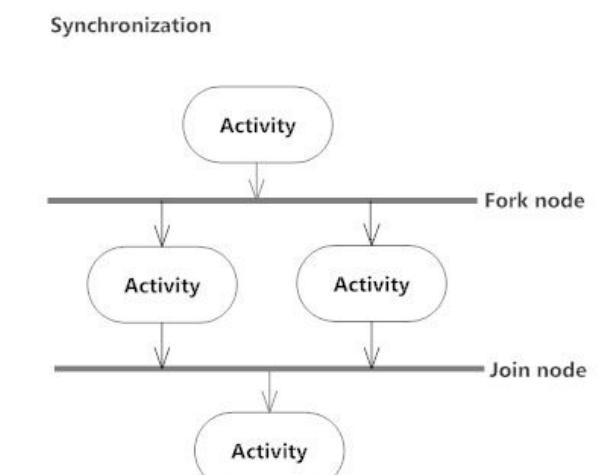
# Activity Diagrams

- 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).

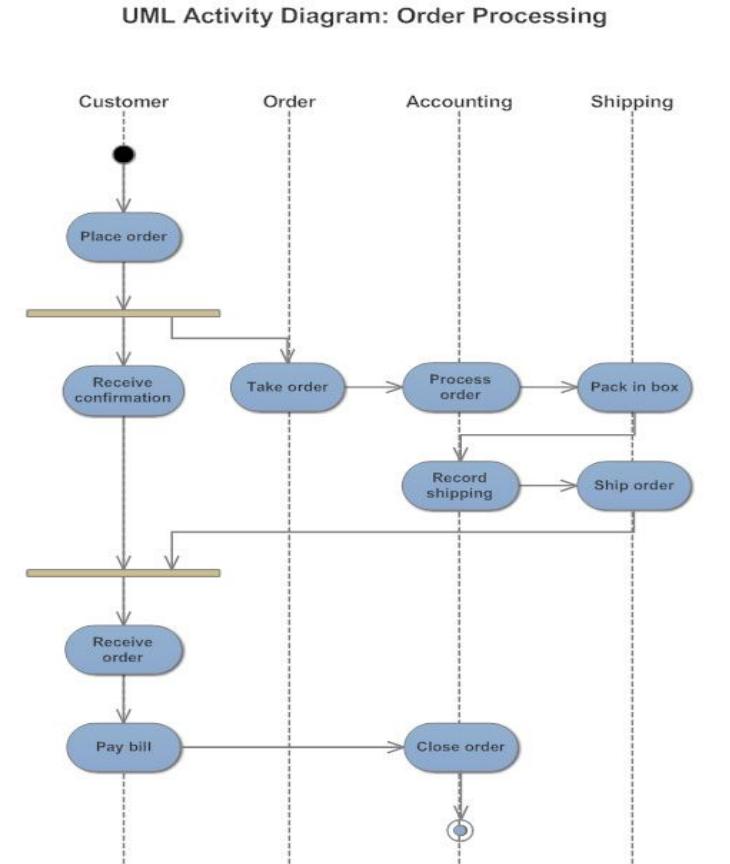
# Activity Diagrams

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

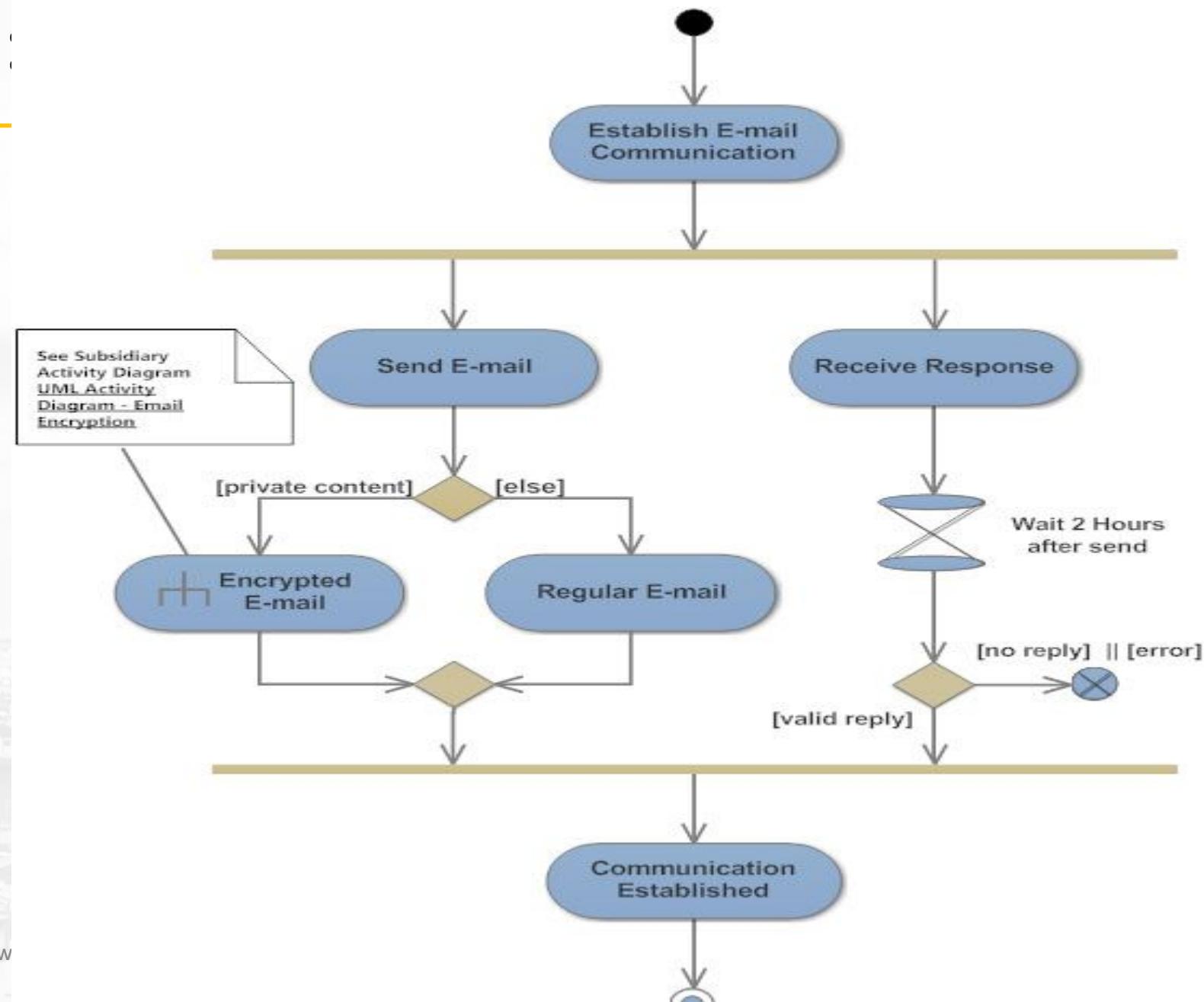
# Activity Diagrams

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.	 Decision Symbol
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.	 Guard Symbols
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.	 Synchronization

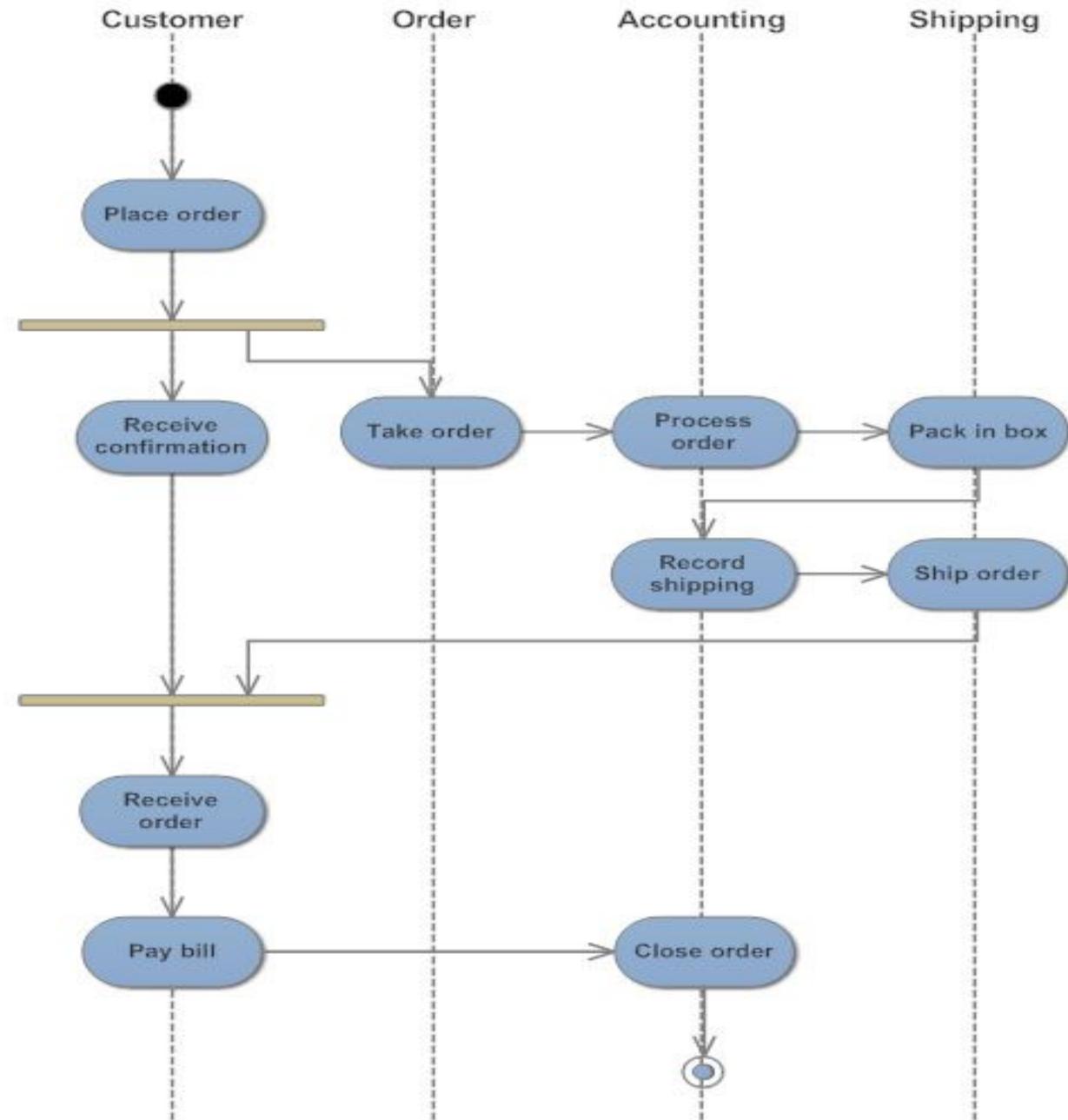
# Activity Diagrams

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 an open circle represents the final action state	
10	Swimlanes	Swimlanes group related activities into one column	

## Example 1



## Example 2 : Food Ordering Processing with Swimlanes

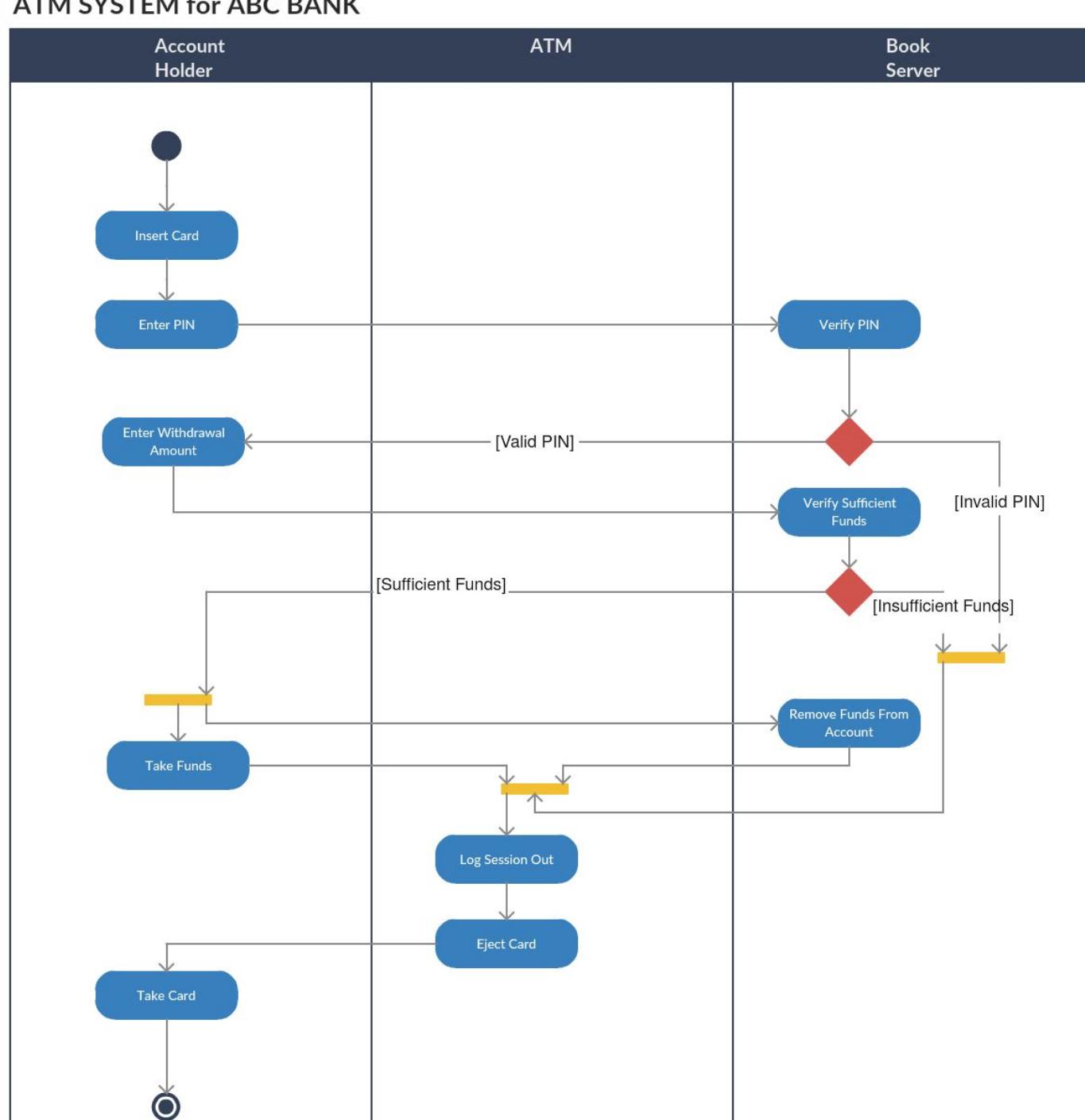




MIT-WPU

॥ विश्वान्तिर्दुर्व ध्रुवा ॥

# Example 3: ABC Bank with swimlanes



# Revision : Symbols of CFD

# Revision : Symbols of CFD

S.No	Name	Description	Notation Yourdon and Coad
1	<b>External Entity</b>	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	<b>Process</b>	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	<b>Data store</b>	Files or repositories that hold information for later use, such as a database table or a membership form.	
4	<b>Data flow</b>	The route that data takes between the external entities, processes and data stores.	

# 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 slide 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.