Standard Chartered Scope International

TEKNOTURF

Top Gun Learning

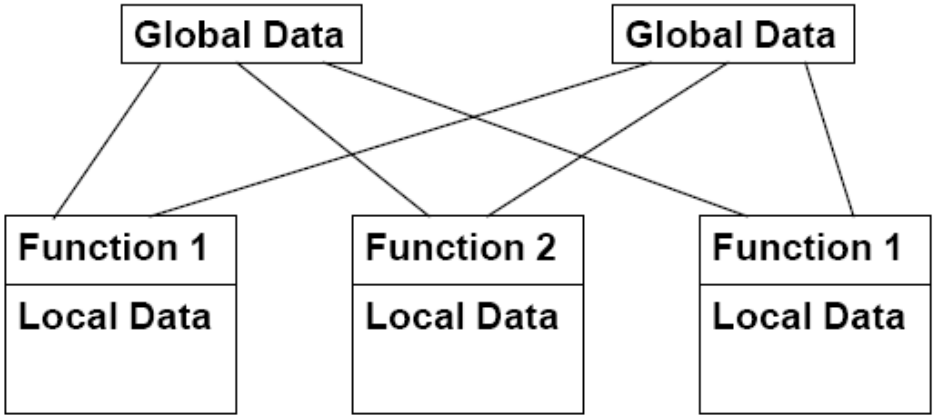
OBJECT ORIENTED PROGRAMMING
&
UML

Abin

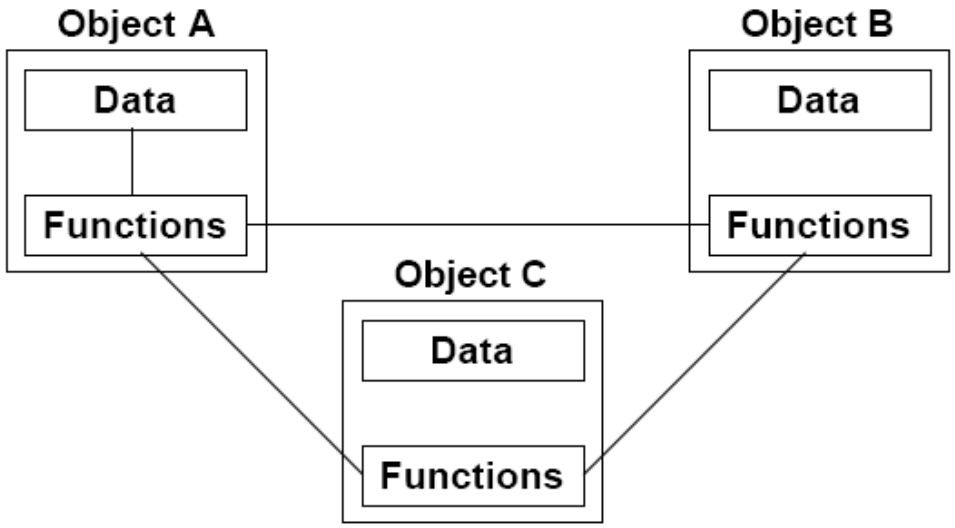
6/8/12

- Understanding different programming styles – Structured, OO, Event
- Major and Minor themes of Object Orientation
  - Encapsulation
  - Abstraction
  - Polymorphism
  - Hierarchy - Inheritance
  - Modularity - Packages
  - Typing
  - Persistence
- Class and Objects
- Object communication – Messages
- Visibility – Access Specifiers

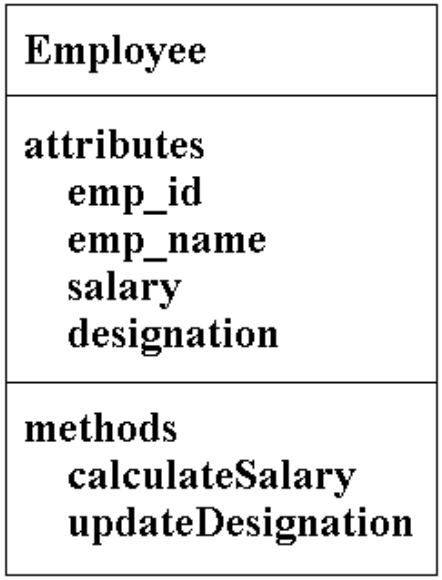
- Emphasis is on doing things (algorithms)
- Large programs are divided into smaller programs (functions)
- Most of the functions share global data
- Data move openly around the system from function to function
- Functions transform data from one form to another
- Employs top-down approach in program design



- Emphasis is on *data* rather than procedure
- Programs are divided into *objects* (abstracted into *classes*)
- Data is *encapsulated* in the objects
- Objects communicate with each other through functions (called *methods*)
- Follows *bottom-up* approach in program design



- A class is a template(specification, blueprint) for a collection of objects that share a common set of attributes and operations.
- Example:
  - Employee (shares common attributes and operation of all employees in a company)



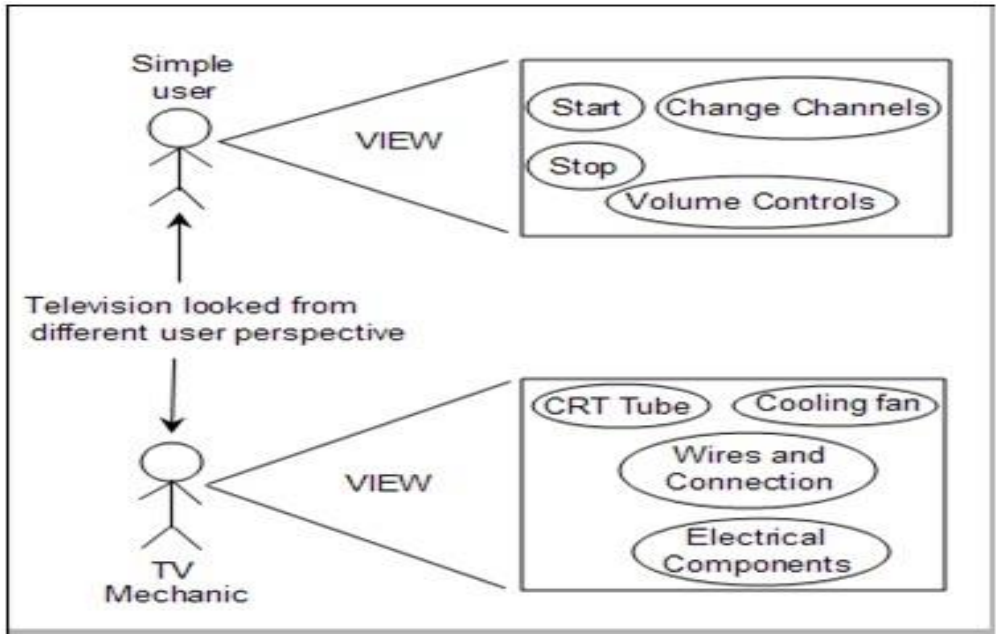
Object

- An instance is an executable copy of a class
- Objects can be:
  - P*hysical* or *conceptual*
  - *Tangible* or *intangible*
- *Example:*
  - *Jerry, Sam, Tom are all objects of Employee*
- *Characteristics of Object*
  - *Identity - Identity is that property of an object which distinguishes it from all others.*
  - *State - The state of an object consists of a set of data fields (its properties) with their current values.*
  - *Behaviour - Behavior is how an object acts and reacts in terms of its state changes and message passing.*

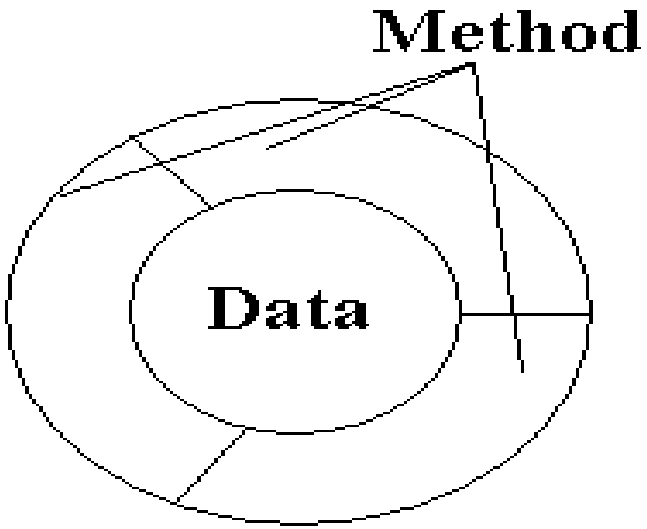
- Software objects communicate and interact with each other in two ways:
  - by calling (or invoking) each other's methods
  - by directly accessing their variables.
- Message Passing
  - Object A calls a method implemented by object B to have it to perform some behavior or return some value.
  - There are three parts to calling a method:
    - The object you are calling that implements the method (e.g., the CD app object)
    - The name of the method to perform (e.g., play)
    - Any arguments needed by the called object (e.g., the CD track #)

- Visibility
  - Visibility is the scope of the attribute.
  - It could take the following values:
    - -: for private - are variables that are visible only to the class to which they belong.
    - + : for public - are variables that are visible to all classes.
    - # : for protected - are variables that are visible only to the class to which they belong, and any subclasses.

- Abstraction is a fundamental principle of modelling.
- Abstraction, as a process, denotes the extracting of the essential details about an item, or a group of items, while ignoring the inessential details.
- Hiding details so as not to confuse the bigger picture

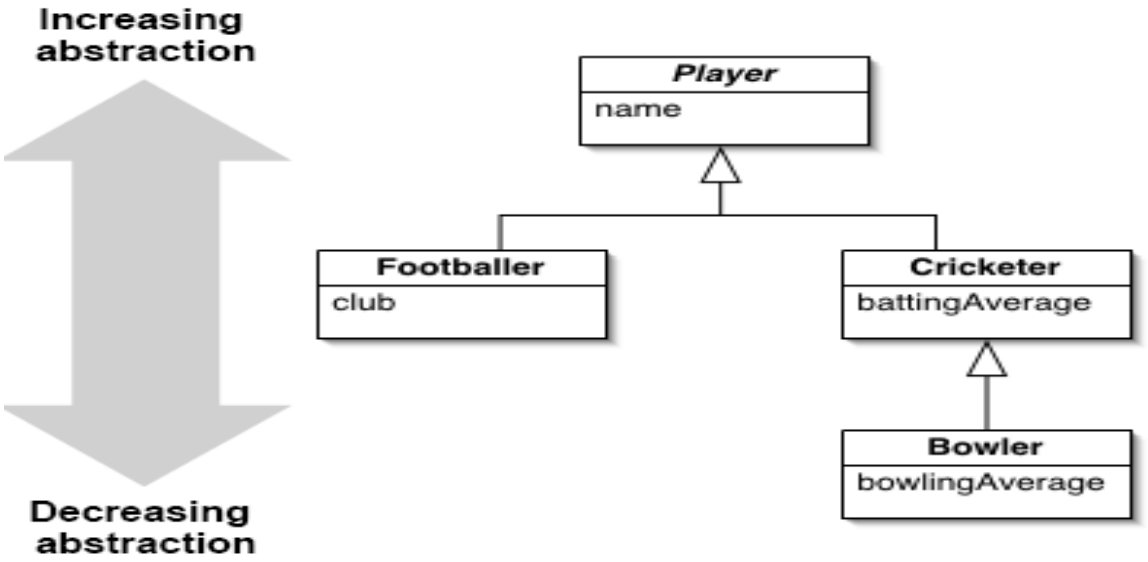


- Encapsulation allows the programmer to **group** data and the subroutines that operate on them together **in one place**, and to **hide irrelevant details** from the user (just like a capsule).
- Each objects methods manage it's own attributes.
- Maximizes maintainability, reuse, extension , coherence



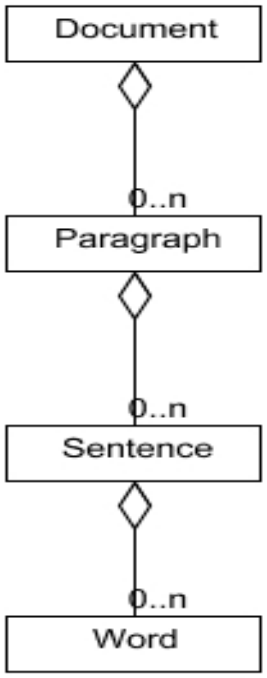
- Polymorphism means having multiple forms
- It is the characteristic of being able to assign a different meaning or usage to something in different contexts specifically, to allow an entity such as a function, or an object to have more than one form
- Two ways of implementing Polymorphism – Compile time polymorphism and Run time polymorphism
- Example:
    - Person
        - Student
        - Employee
        - Son
        - Daughter

- Hierarchy is a ranking or ordering of abstractions.
- Types
    - Inheritance or class structure hierarchy
        - Represents “is a”or “is a type of”relationship between two classes. Features that are common to many classes are migrated to a base class

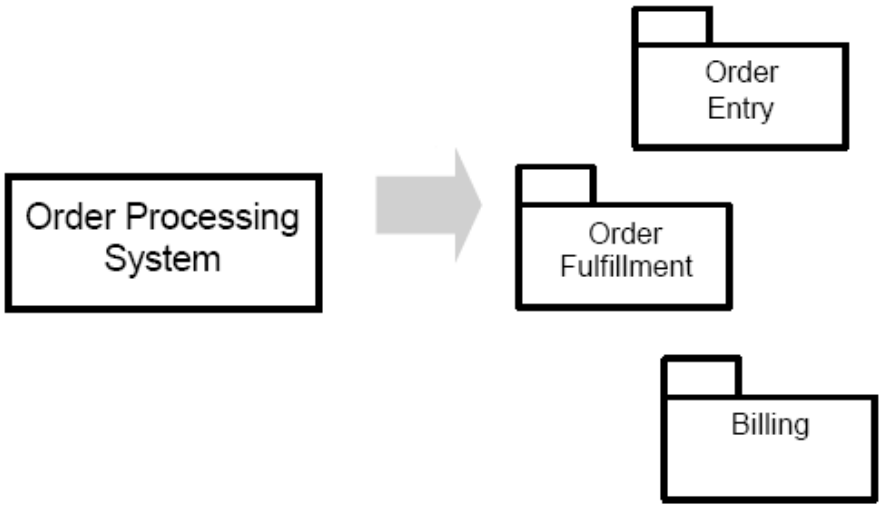


- Aggregation or object structure hierarchy
  - Represents the “is a part of”or “whole-part”relationship where one object becomes a part of other object.

Aggregation:
The document cosists of paragraphs, which themselves consist of sentences, that are built with words.



- Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.
- Modularity can be defined as the process of breaking up of a complex system into small, self contained Pieces that can be managed easily
- A collection of related classes called as packages.



- **Typing**
  - Is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in restricted ways.
  - Examples of Typing: Strongly Typed and Weakly typed
- **Persistence:**
  - Is the property of an object through which its existence transcends in time.(i.e. Object continues to exist after its Creator ceases to exist) and/or space.(i.e Object's locationmoves from the address space in which it was created.)

UML

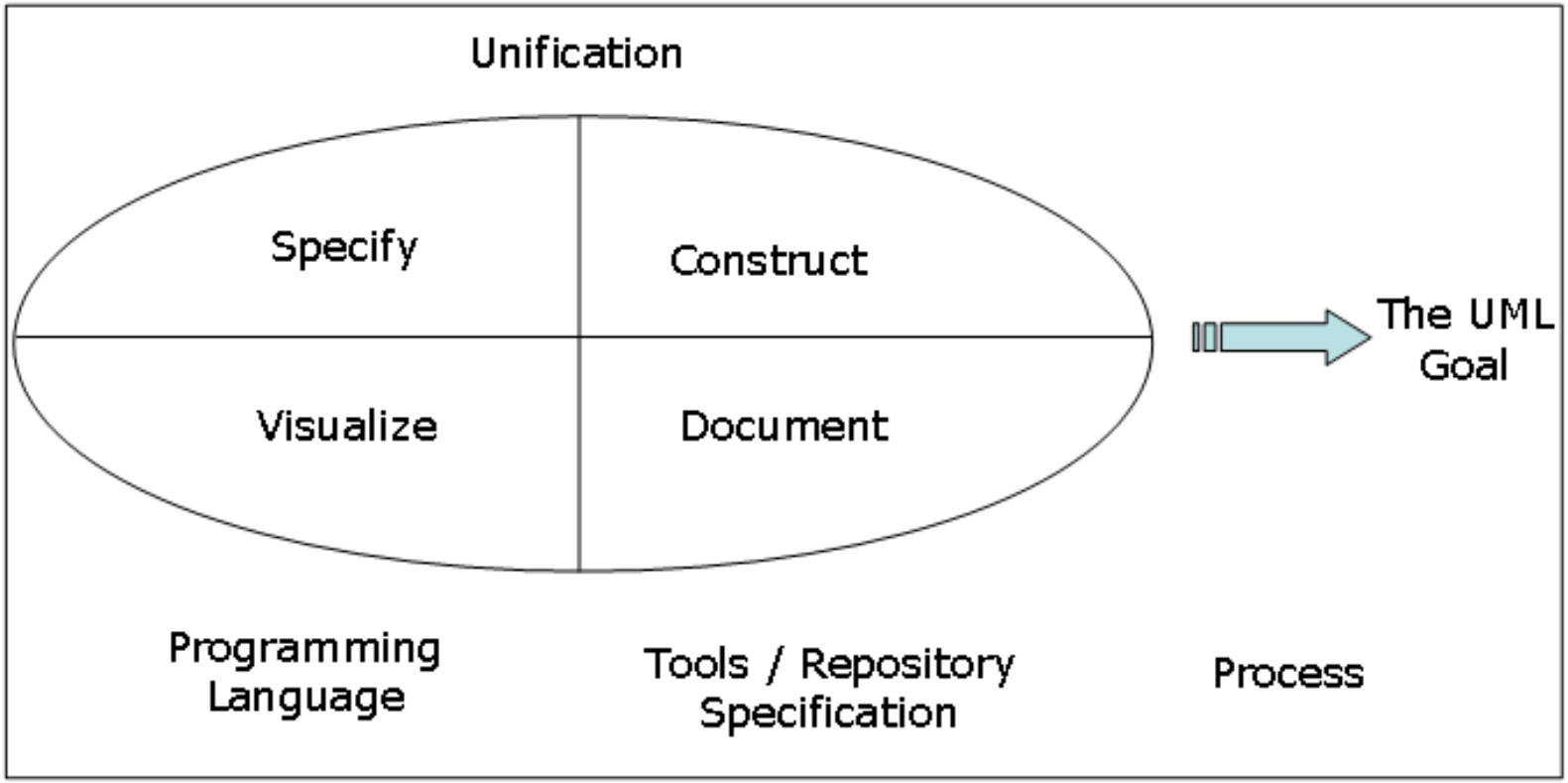
- Modeling
- UML Introduction
- UML view
- UML Diagram
  - Usecase Diagram
  - Class Diagram
  - Interaction Diagram
  - Deployment Diagram
  - Component Diagram
  - Package Diagram
  - Activity Diagram
  - StateChart Diagram

Modeling

- Modeling is a way of thinking about the problems using models organized around the real world ideas.
- A modeling method comprises a language and also a procedure for using the language to construct models.
- modeling is the only way to visualize your design and check it against requirements before your crew starts to code.

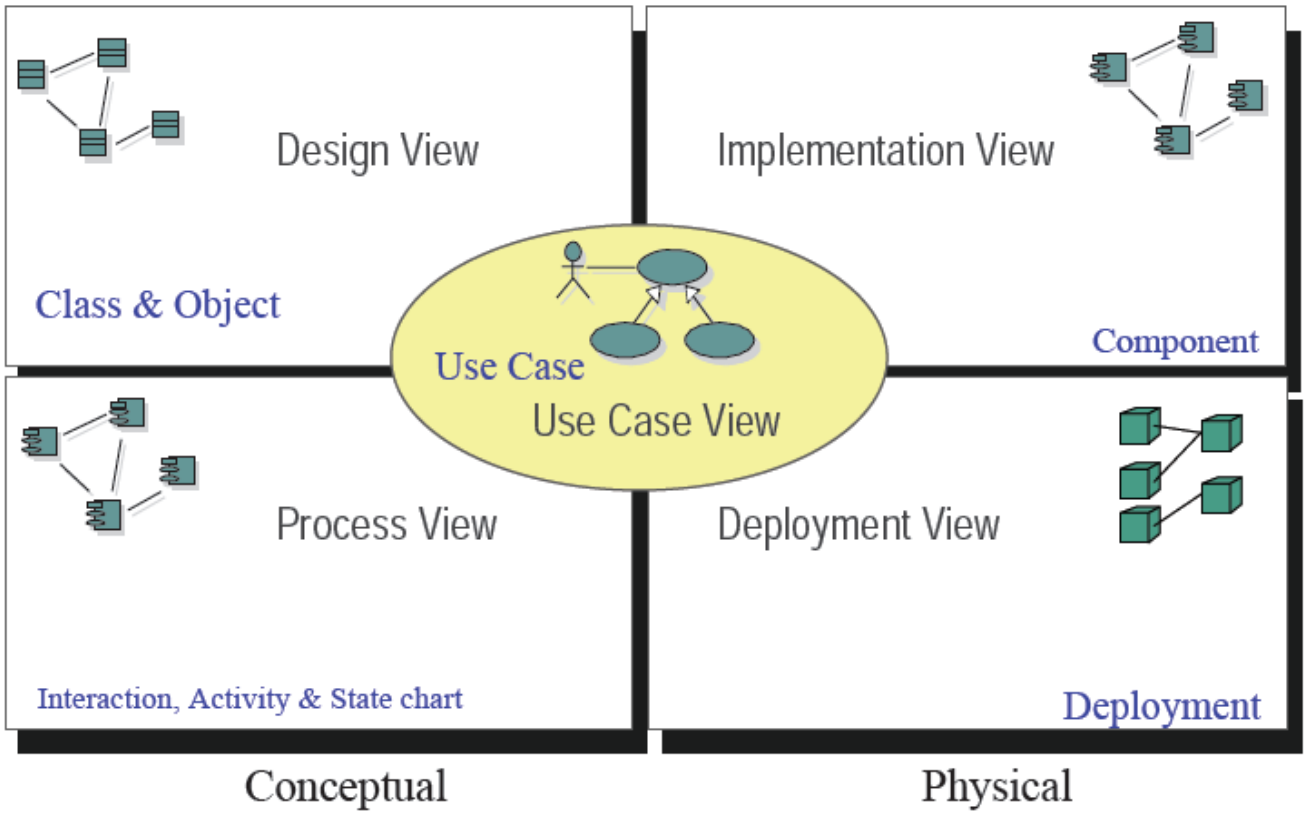
UML

- UML – Unified Modeling Language is a Standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, business modeling and other non-software systems
- UML is a pictorial language used to make software blue prints.
- It is not simply a notation for drawing diagrams, but a complete language for capturing knowledge (semantics) about a subject and expressing knowledge (syntax) regarding the subject for the purpose of communication.
- Applies to modeling and systems. Modeling involves a focus on understanding a subject (system) and capturing and being able to communicated in this knowledge.

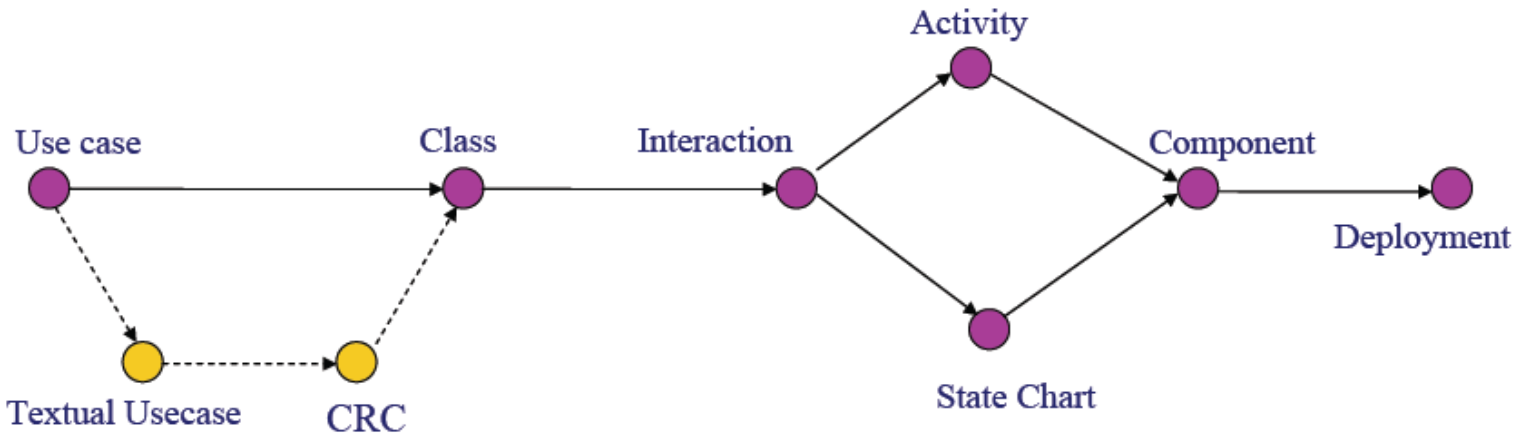

Unification
Specify
Construct
Visualize
Document
The UML Goal
Programming Language
Tools / Repository Specification
Process

- **User View**
  - Models which define a solution to a problem as understood by the client or stakeholders.
  - Use Case Diagram
- **Structural View**
  - Models which provide the static, structural dimensions and properties of the modelled system.
  - Class diagram, Object diagram
- **Behavioural View**
  - Models describe the behavioural and dynamic features and methods of the modelled system.
  - Interaction diagram(sequence diagram, Collaboration diagram) , Activity diagram, state chart diagram

- **Implementation View**
  - View combines the structural and behavioral dimensions of the solution realization or implementation.
  - Component diagram, package diagram
- **Environment view**
  - Models describe both structural and behavioural dimensions of the domain or environment in which the solution is implemented.
  - Deployment diagram


Design View
Implementation View
Class & Object
Use Case
Use Case View
Component
Process View
Deployment View
Interaction, Activity & State chart
Deployment
Conceptual
Physical

UML



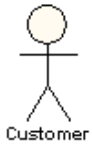
- UML Diagrams
- Not Part Of UML

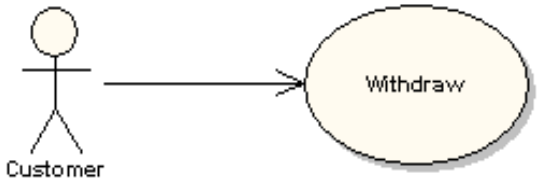
- depicts different users of the system and the functions of the system.
- Use case diagram expose the requirements of the system.
- Actors interact with a use case and the functions are called as use cases.
- Use cases diagrams describe the interaction between the actor and the use case.

Use Case Diagram

Components

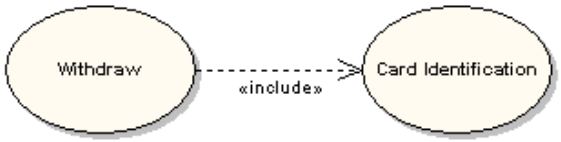
- **Actor** : which represent users of a system, including human users and other systems.
- **Usecase** : which represent functionality or services provided by a system to users.
- **Association** :Associations between actors and use cases are indicated by solid lines. An association exists whenever an actor is involved with an interaction described by a use case.

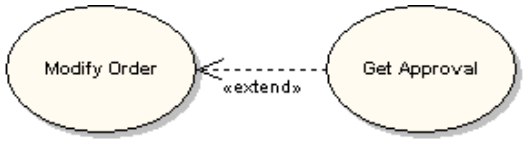


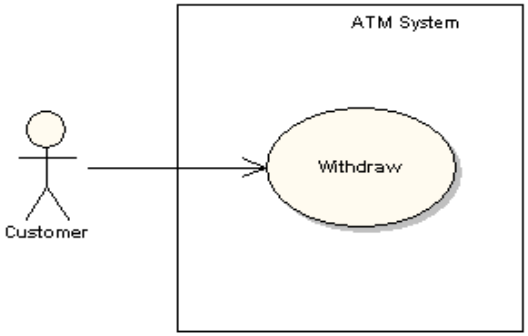


Use Case Diagram

- **Includes** : Use cases may contain the functionality of another use case as part of their normal processing. In general it is assumed that any included use case will be called every time the basic path is run.
- **Extends** : One use case may be used to extend the behavior of another; this is typically used in exceptional circumstances.
- **System Boundary** :It is usual to display use cases as being inside the system and actors as being outside the system.







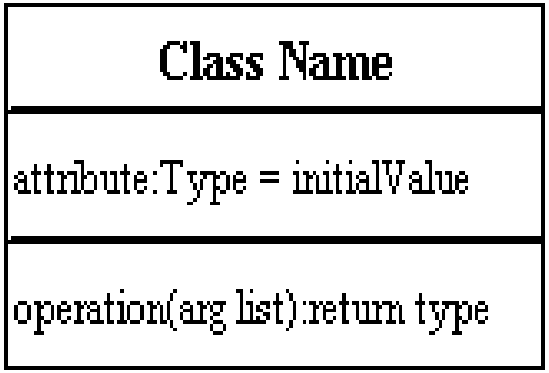
Use Case Specification

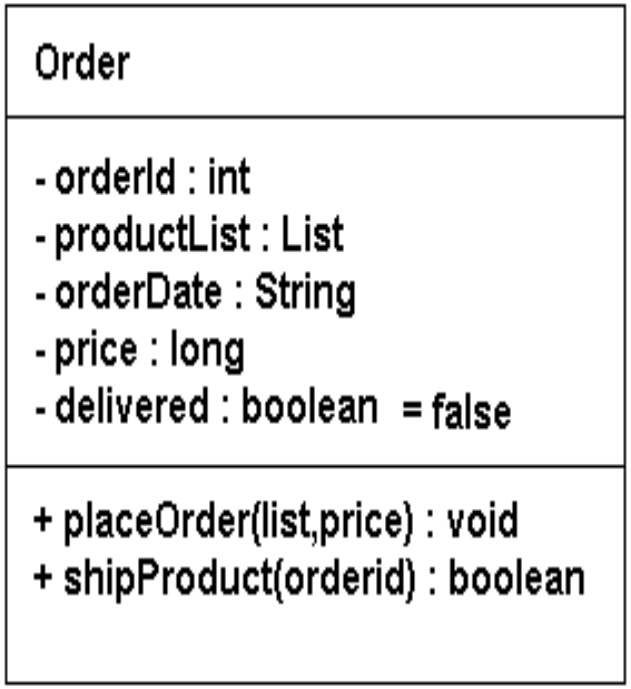
- Use case name - States the use case name.
- Use case actor - Actor interacting with the use case
- Preconditions - A state of the system that must be present before a use case is performed.
- Basic flow of events - Describes the ideal, primary behavior of the system.
- Alternative flow of events - Describes exceptions or deviations from the basic flow,
- Post conditions - A list of possible states for the system immediately after a use case is finished.
- Extension points - A point in the use-case flow of events at which another use case is referenced

- Class Diagrams describe the static structure of a system, or how it is structured rather than how it behaves. These diagrams contain the following elements.
  - Classes, which represent entities with common characteristics or features. These features include attributes, operations and associations.
  - Associations, which represent relationships that relate two or more other classes where the relationships have common characteristics or features. These attributes and operations.

Class Diagram

- Class Notation



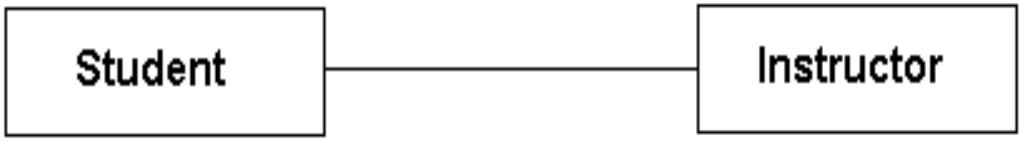


Class Diagram

- Relationship
  - Relationships connect two or more things. In object orientation its connections between two elements.
  - types of relationship among classes
    - Association
    - Inheritance or Generalization
    - Aggregation
    - Composition
    - Dependency
    - Realization

Class Diagram

- The association relationship is a simple relationship. It connects two classes , denoting the structural relationship between them. An association is shown as a line connecting the related classes.

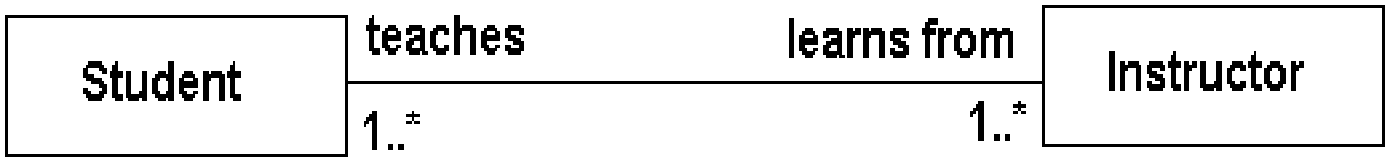


- Multiplicity
  - The number of instances of the class, next to which the multiplicity expression appears, that are referenced by a **single** instance of the class that is at the other end of the association path.
  - Indicates whether or not an association is mandatory.
  - Provides a lower and upper bound on the number of instances.

Class Diagram

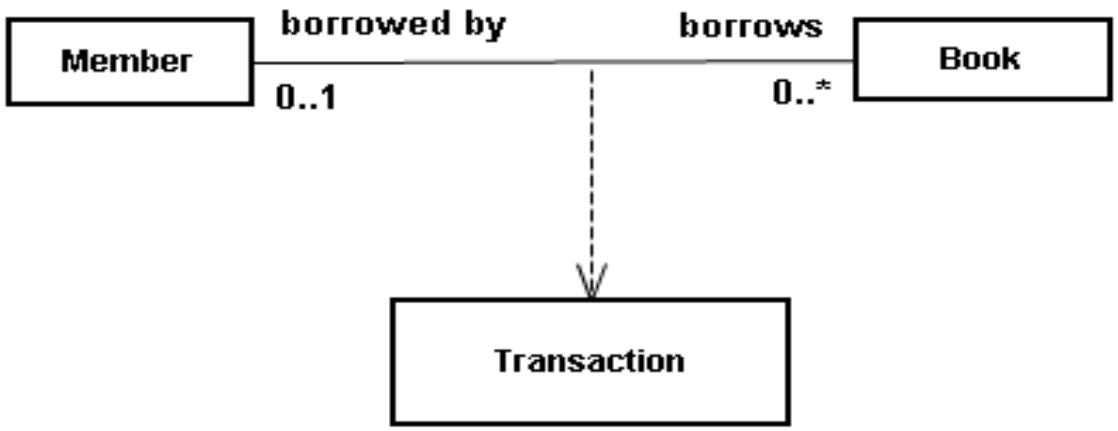
- Multiplicity Indicators

Exactly one	1
Zero or more (unlimited)	* (0..*)
One or more	1..*
Zero or one (optional association)	0..1
Specified range	2..4
Multiple, disjoint ranges	2, 4..6, 8



Class Diagram

- Association class
  - we can also show an association class in an association relationship. An association class is one that makes it possible for the association to exist between two other classes.



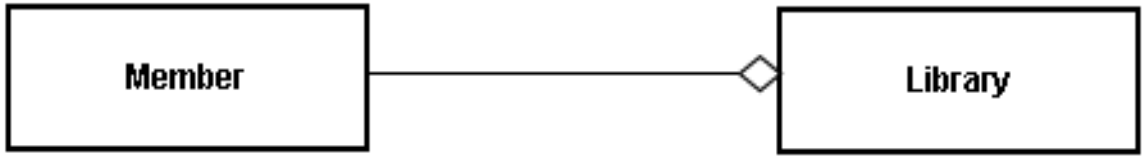
Class Diagram

- Dependency
  - Is the “using”or “uses”relationship where one element is using another element to get its task done. A change in one element affects the another element that uses it. i.e. if the one changes then it affects the other element. The reverse need not be true.
  - The dependency from *CourseSchedule* to *Course* exists because *Course* is used in both the **add** and **remove** operations of *CourseSchedule*.

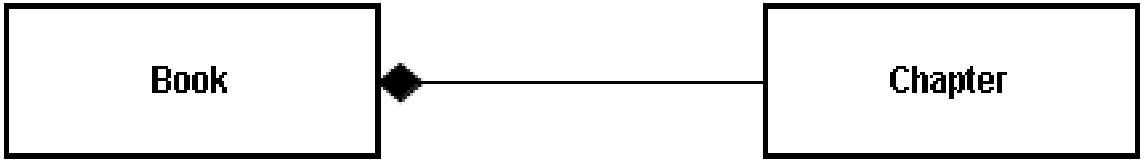
- Aggregation
  - The class at one end of the relationship will contain classes at the other end of the relationship. It is also called a whole part relationship. Signifies the “part of”or “has a”relationship.
  - Examples:
    - A room has a door.
    - Building has rooms.
    - Vehicle has an engine.
    - Company has departments
    - Book has chapters.
- Two forms of Aggregation:
  - Simple Aggregation
  - Composition

Class Diagram

- Aggregation

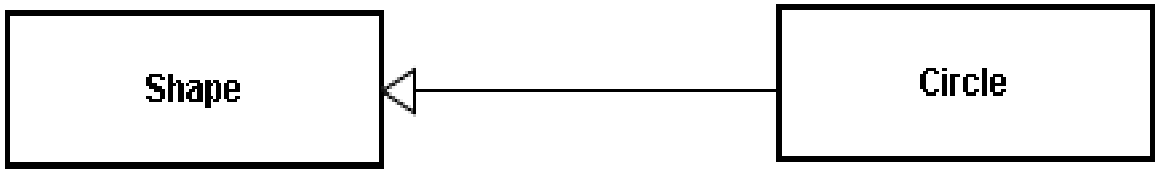


- Composit



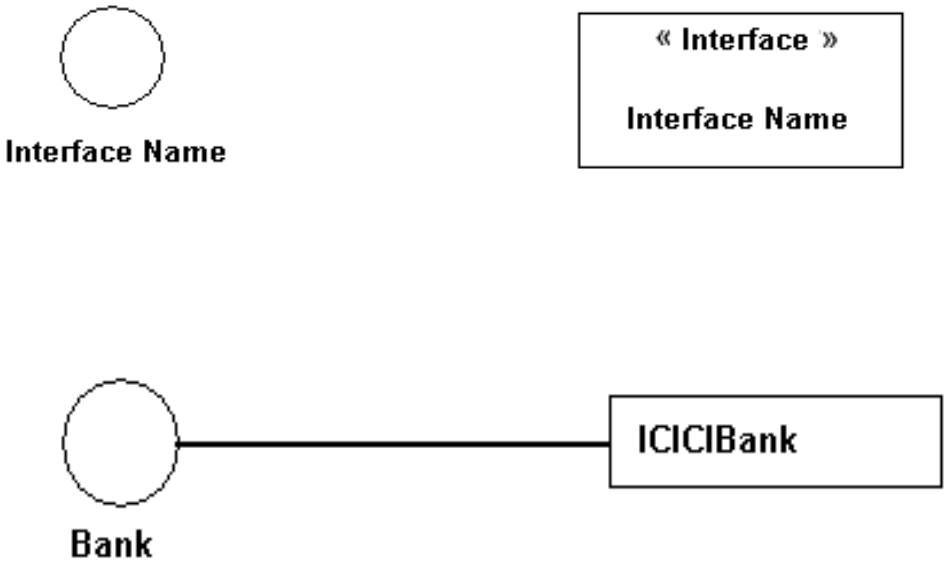
Class Diagram

- Inheritance
  - Generalization consists of the super class and sub-class relationship.
  - The super class represents a general concept while the sub class represents a more specific concept.
  - Also called as “is a type of ”.
  - Examples:
    - Rose is a type of flower.
    - Windows is a kind of operating system.



Class Diagram

- Realization
  - The relationship between the class and the interface. An interface is formally equal an abstract class with no attributes and methods(implementations), only abstract Operations.



Interaction Diagram

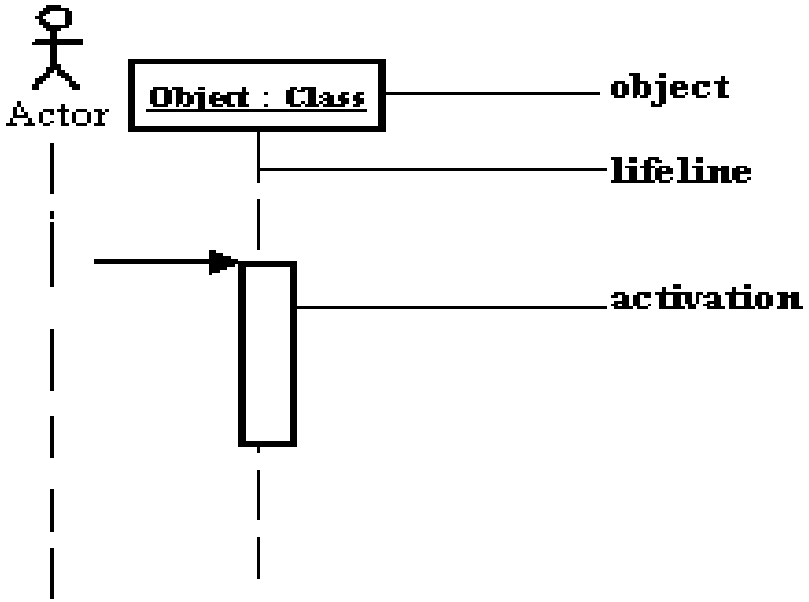
- Visualize the interactive behaviour of the system
- Purpose of interaction diagram
  - To capture dynamic behaviour of a system.
  - To describe the message flow in the system.
  - To describe structural organization of the objects.
  - To describe interaction among objects.
- Interactive behaviour is represented in UML by two diagrams known as Sequence diagram and Collaboration diagram.

Sequence Diagram

- Used to show how objects interact in a given situation. An important characteristic of a sequence diagram is that time passes from top to bottom
- Is an interaction diagram that emphasizes the time ordering of messages.
- Graphically a sequence diagram is a table that shows objects arranged along X axis and messages, ordered in increasing time, along the Y axis.
- Can model simple sequential flow, branching, iteration, and concurrency.

Sequence Diagram - Notation

- Object – instance of the class participating in interaction
- Lifeline - The lifeline represents the object's life during the interaction.
- Activation - Activation box represent the time an object needs to complete a task.

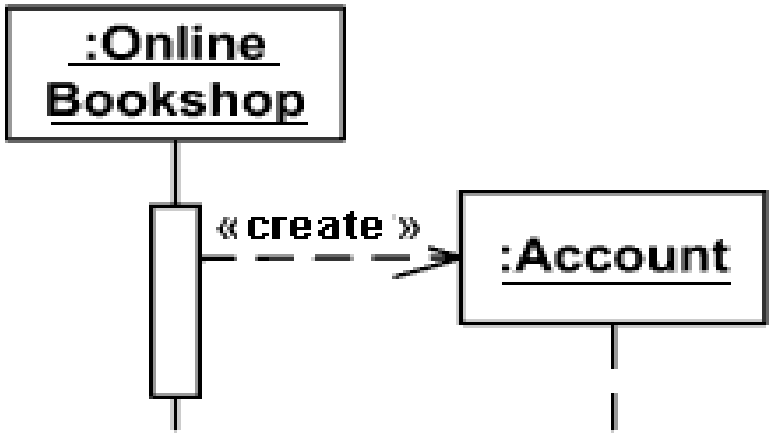


Sequence Diagram - Notation

Messages

- Create

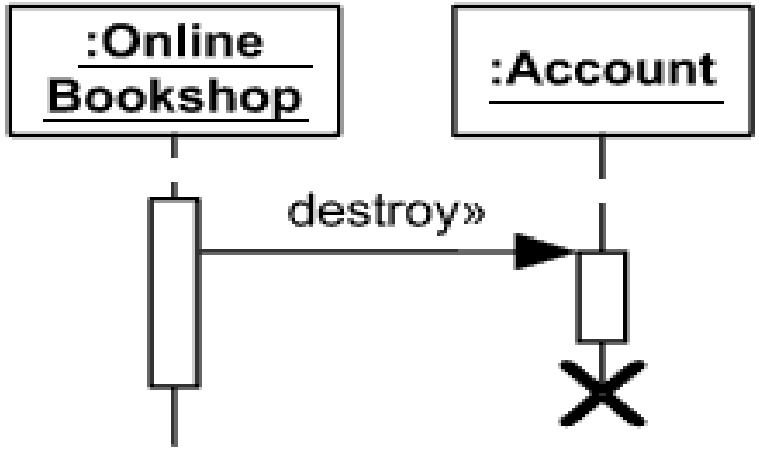
Create message is a kind of message that represents the instantiation of (target) lifeline.



Sequence Diagram - Notation

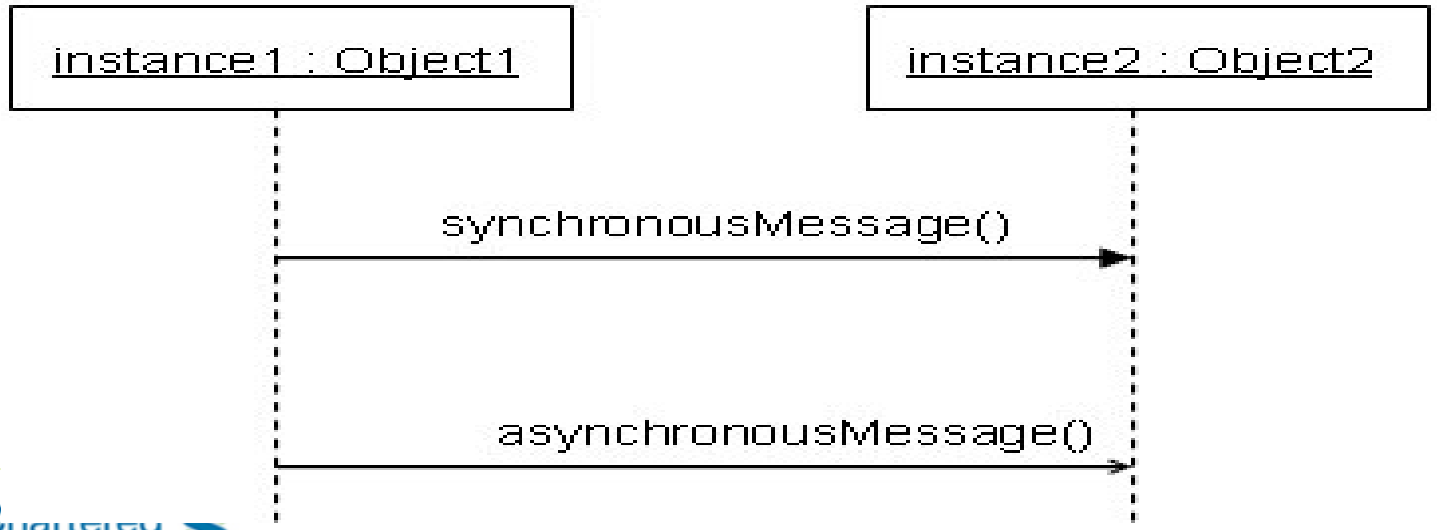
- Destroy

Destroy message is a kind of message that represents the request of destroying the life cycle of target lifeline.



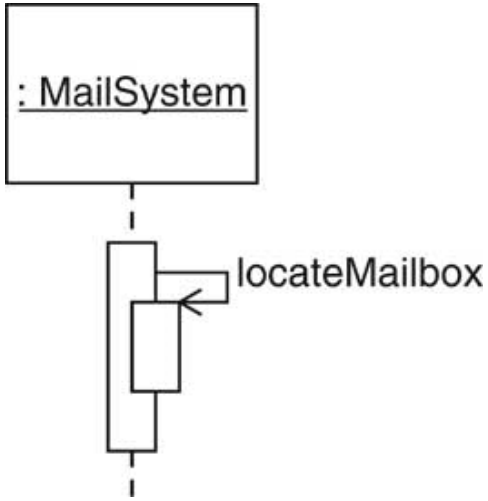
Sequence Diagram - Notation

- Synchronous call
  - It must wait until the message is done, such as invoking a subroutine.
- Asynchronous call
  - It can continue processing and doesn't have to wait for a response.



Sequence Diagram - Notation

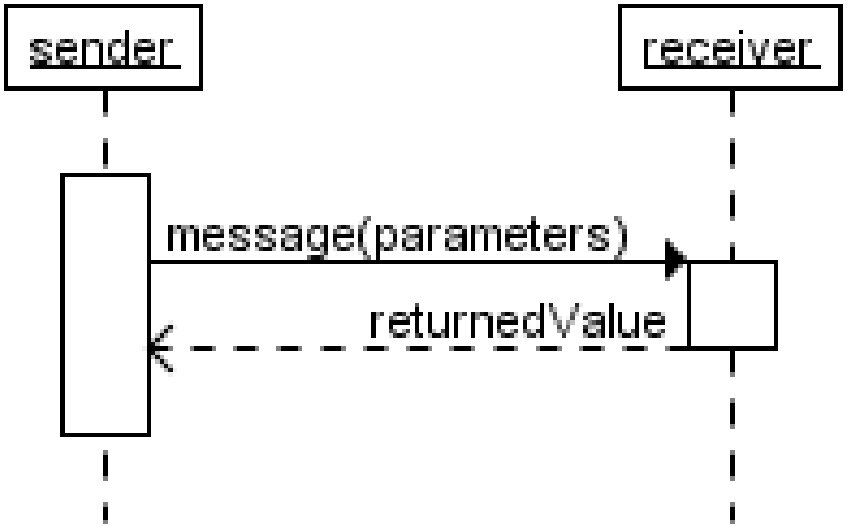
- Self call
  - A self message can represent a recursive call of an operation, or one method calling another method belonging to the same object.



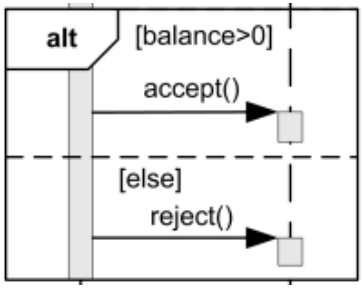
Sequence Diagram - Notation

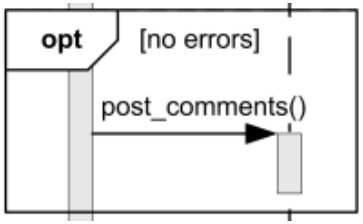
- Return message

Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message.



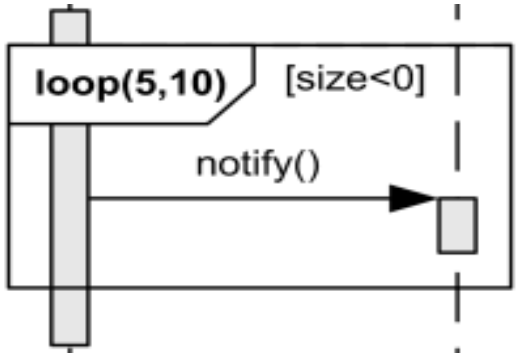
- Alt – shows the true and the false logic flow
- Opt - The interaction operator opt means that the combined fragment represents a choice of behavior where either the (sole) operand happens or nothing happens.



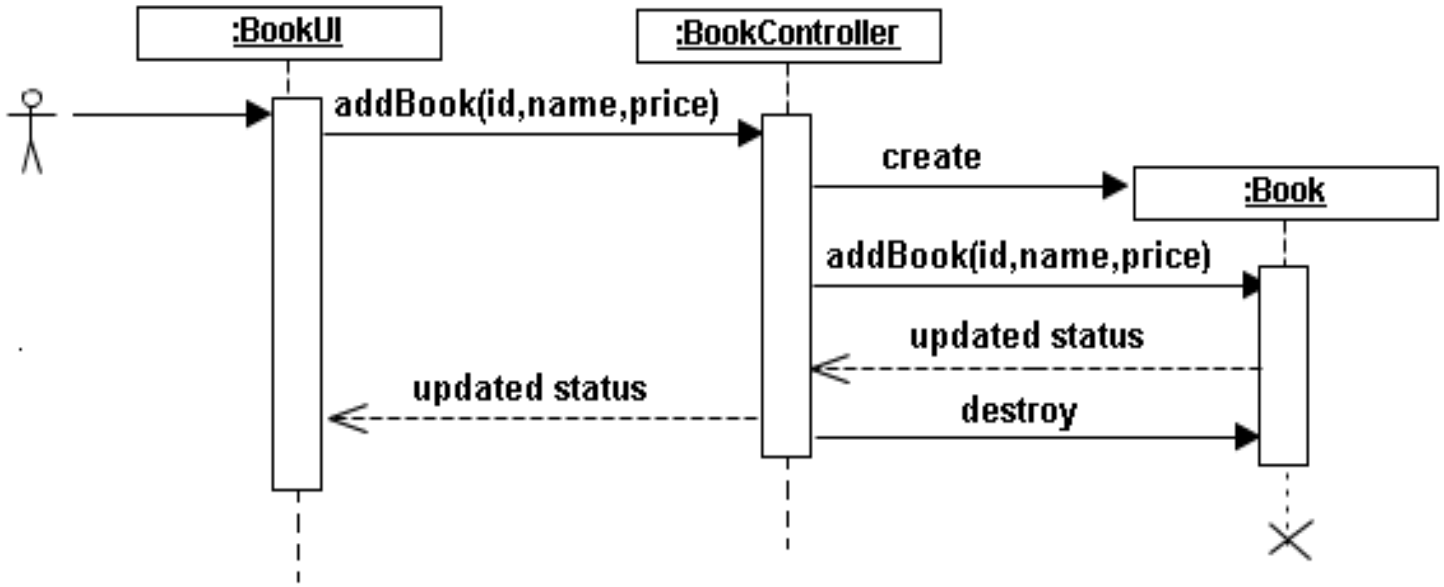


Sequence Diagram - Notation

- The loop is expected to execute minimum 5 times and no more than 10 times. If guard condition [size<0] becomes false loop terminates regardless of the minimum number of iterations specified.

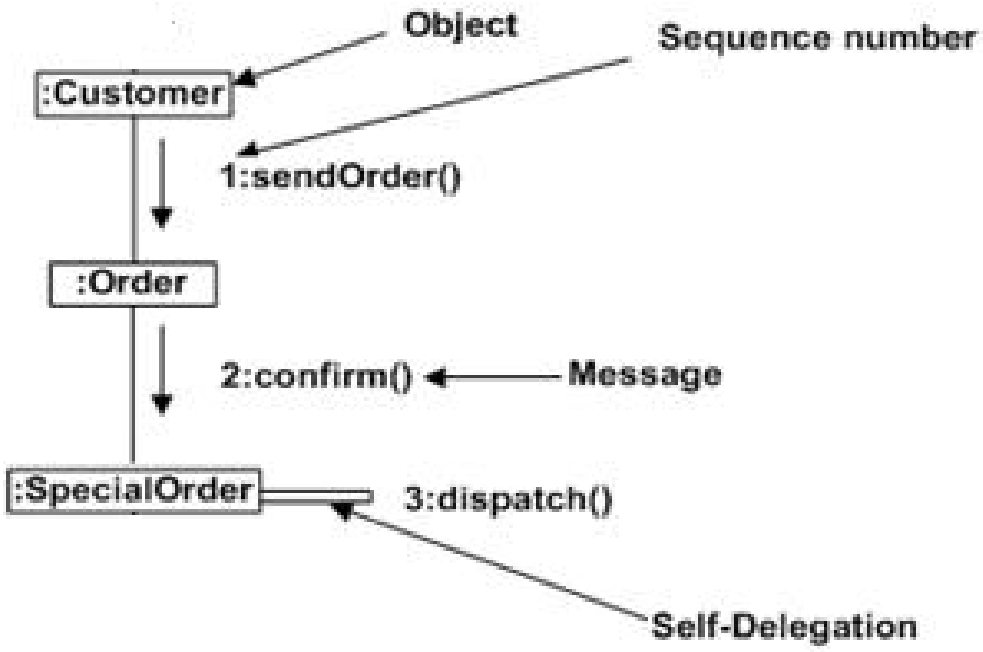


Sample Sequence Diagram



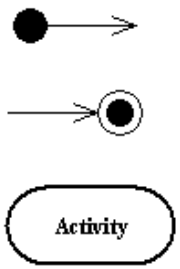
Collaboration Diagram

Collaboration Diagram



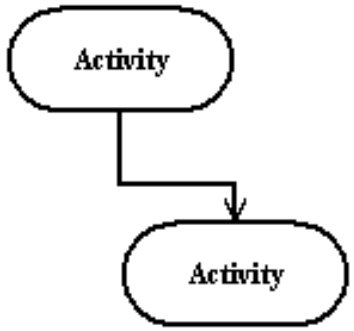
Activity Diagram

- An activity diagram illustrates the dynamic nature of a system by modeling the flow of control from activity to activity.
- An activity represents an operation on some class in the system that results in a change in the state of the system.
- Activity diagrams are used to model workflow or business processes and internal operation
- Notations
  - Initial state
  - Final state
  - Action state

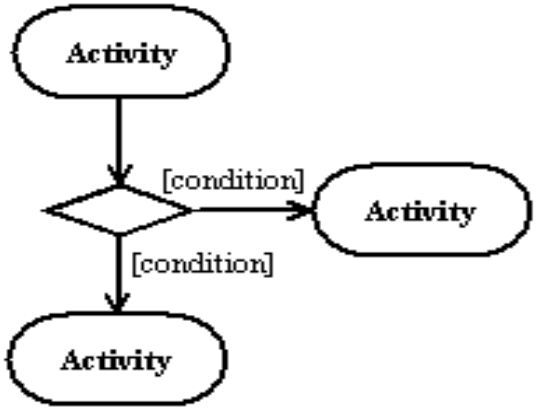


Activity Diagram

- Action flow

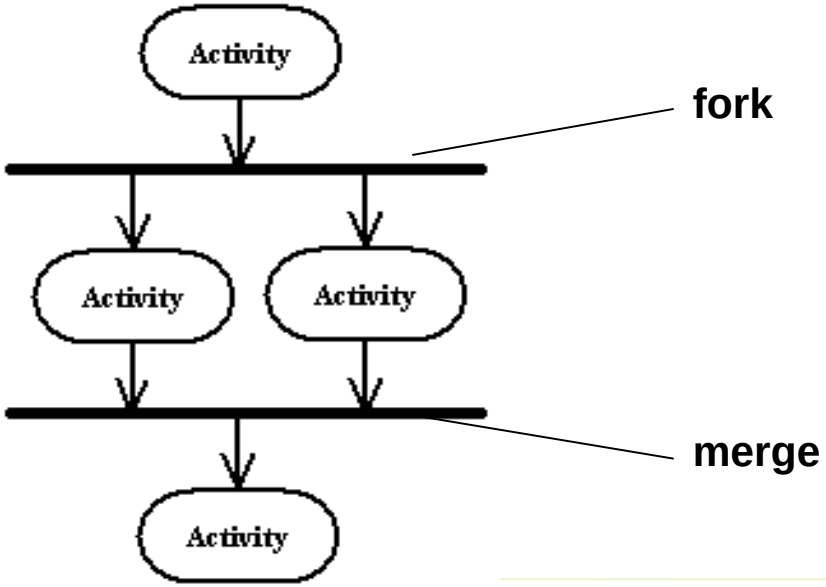


- Branching

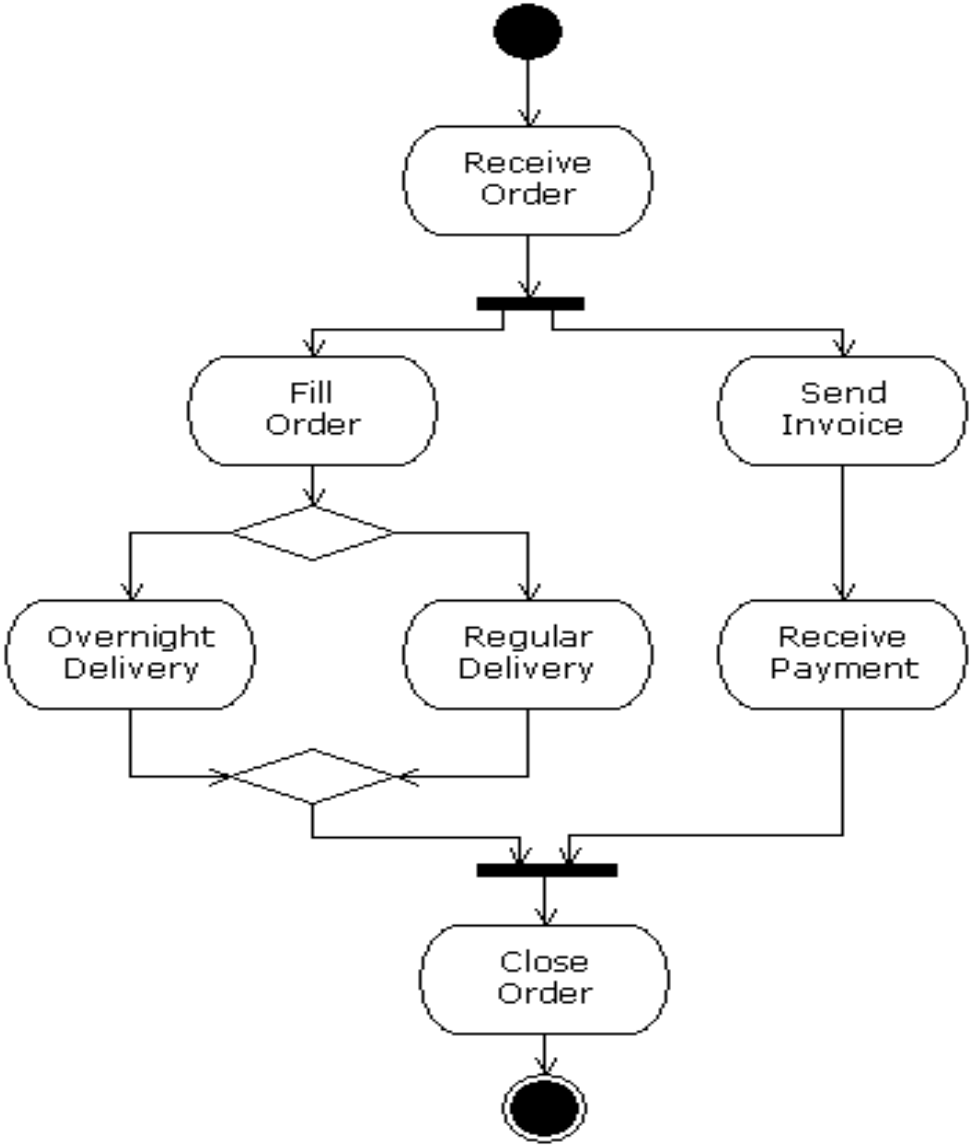


Activity Diagram

- Synchronization
  - A synchronization bar helps illustrate parallel transitions. Synchronization is also called forking and joining.



Activity Diagram



State Chart Diagram

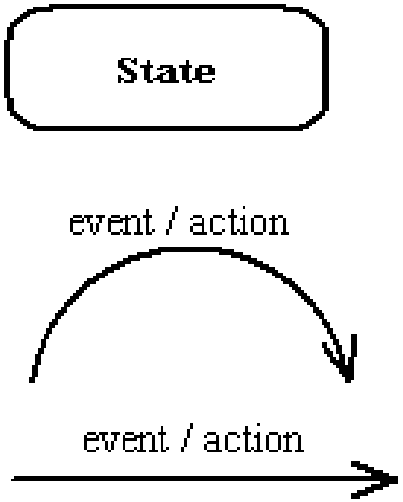
- Used to model dynamic nature of a system.
- They define different states of an object during its lifetime.
- purpose of Statechart diagram:
  - To model dynamic aspect of a system.
  - To model life time of a reactive system.
  - To describe different states of an object during its life time.
  - Define a state machine to model states of an object.

State Chart Diagram

Notation

- States - States represent situations during the life of an object.

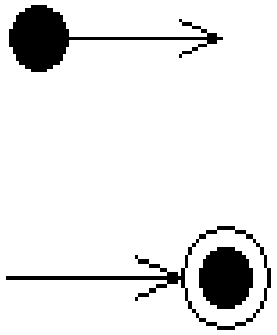
- Transition - A solid arrow represents the path between different states of an object.



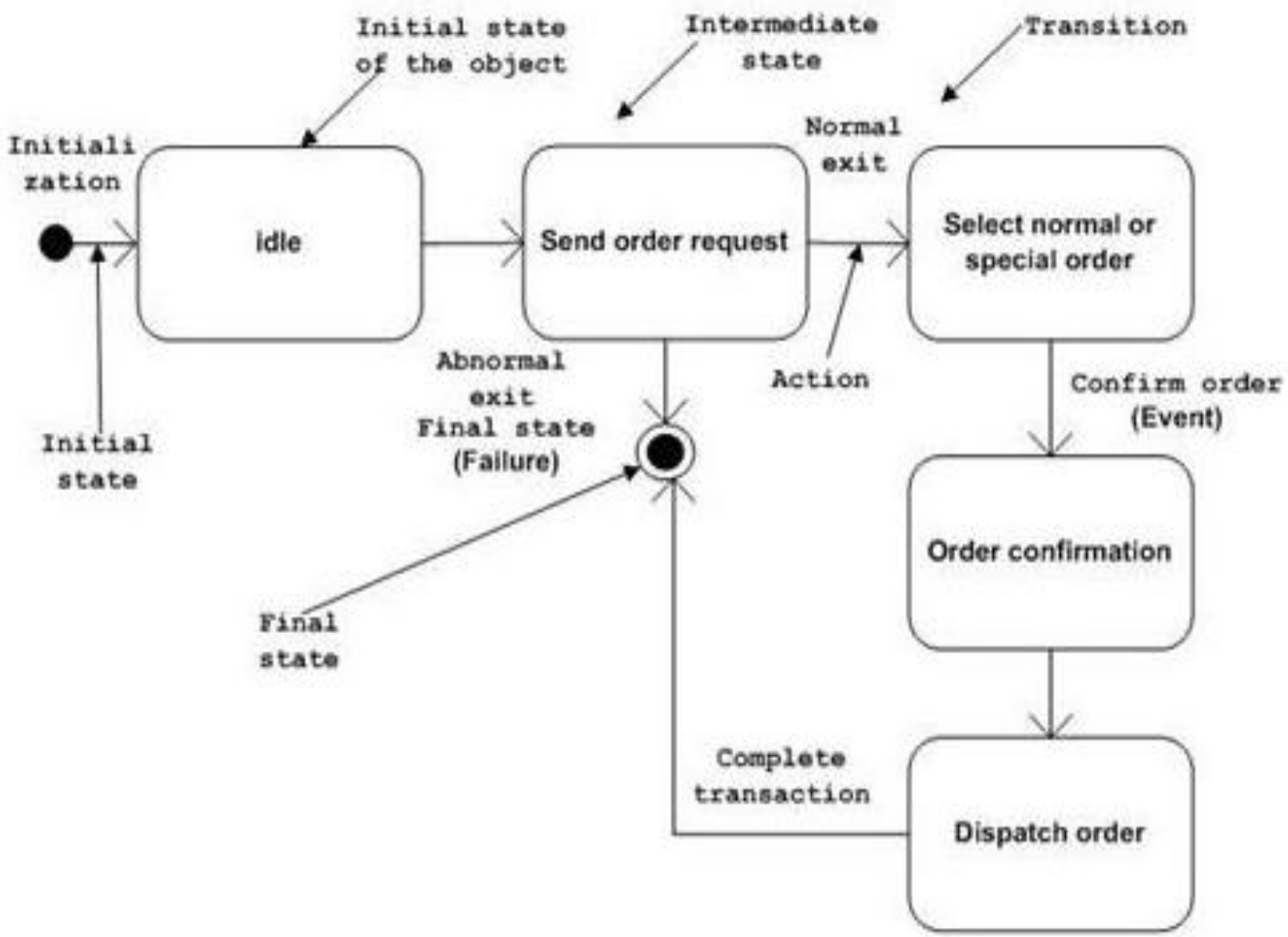
State Chart Diagram

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

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



Statechart diagram of an order management system



Component Diagram

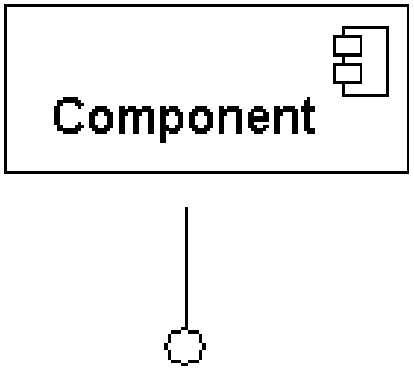
- Component diagrams are used to model physical aspects of a system.
- Physical aspects are the elements like executables, libraries, files, documents etc which resides in a node.
- component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.
- Component diagrams are used during the implementation phase of an application.

Component Diagram

Notations

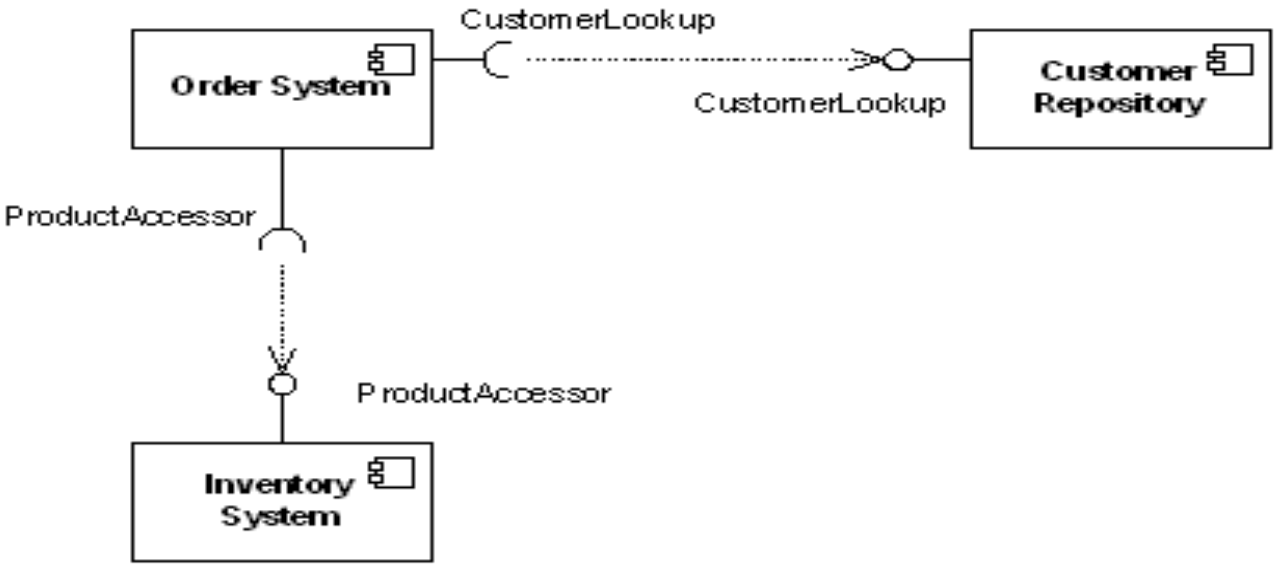
- Component -A component is a physical building block of the system

- Interface -An interface describes a group of operations used or created by components.



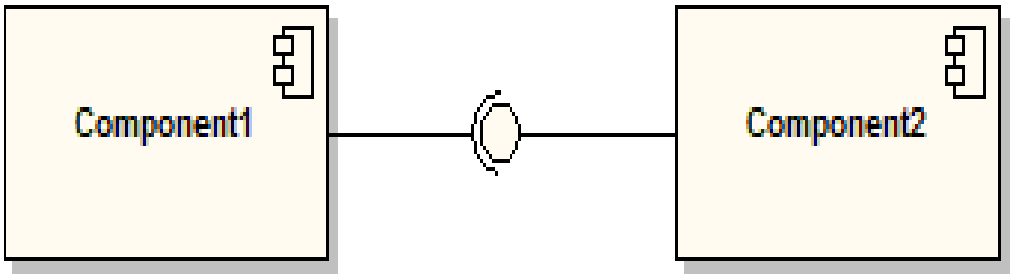
Component Diagram

- Dependencies



Component Diagram

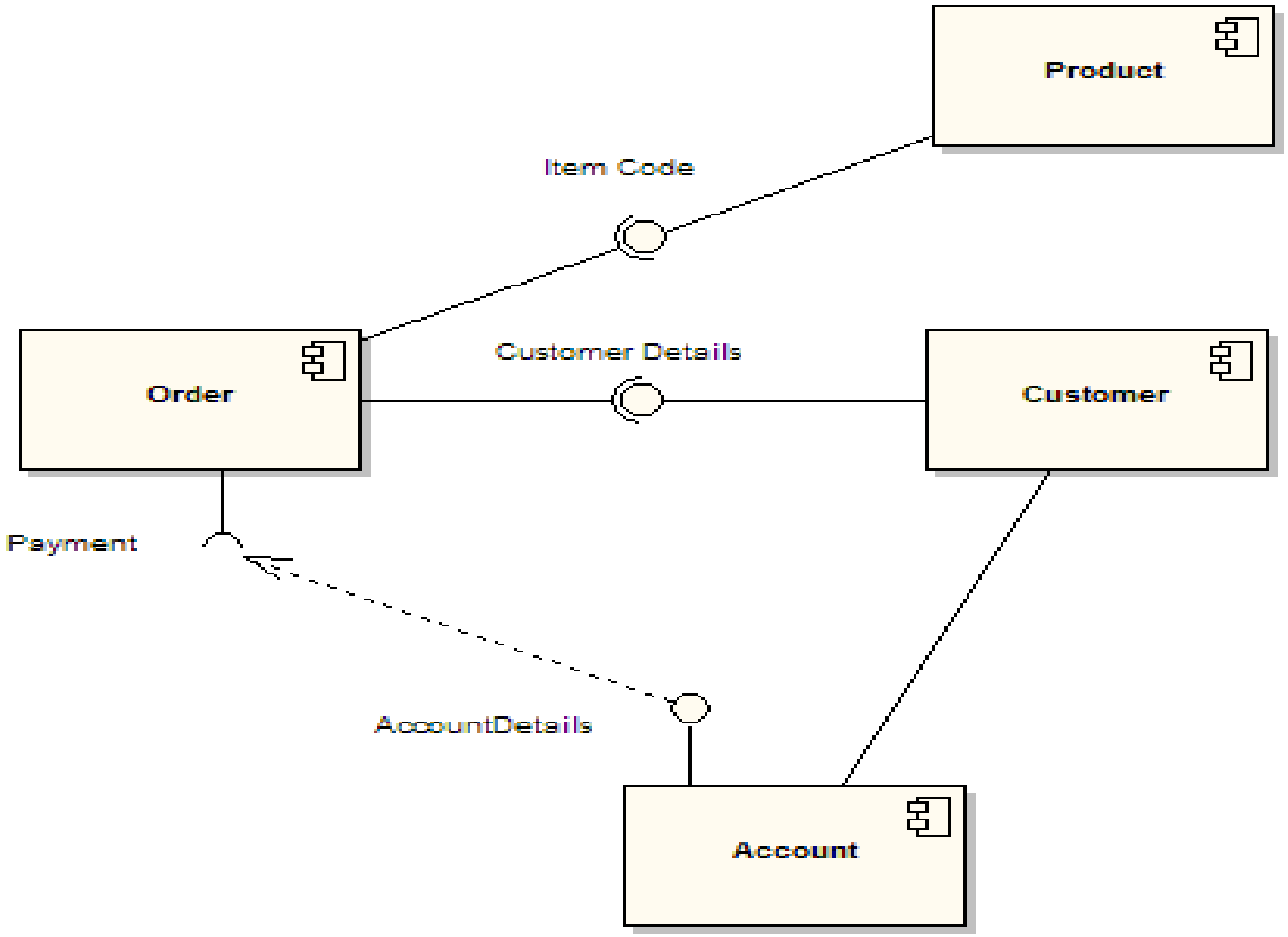
- Assembly Connector - The assembly connector bridges a component's required interface (Component1) with the provided interface of another component (Component2).This allows one component to provide the services that another component requires.



Component Diagram

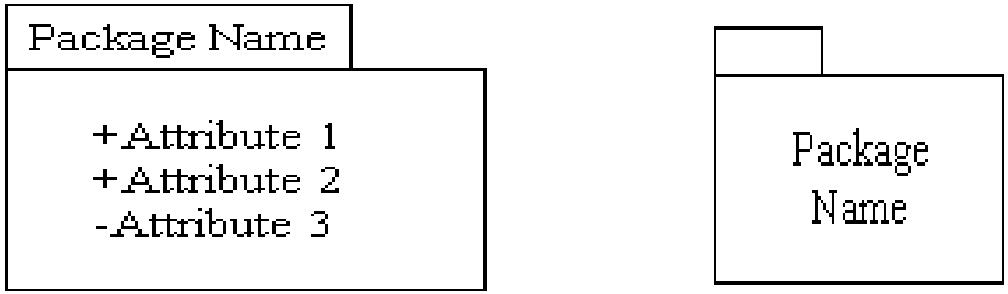
- Components with Ports -
  - the interface symbols with a complete circle at their end(ports) represent an interface that the component provides.
  - Interface symbols with only a half circle at their end(sockets) represent an interface that the component requires
  - the Order component provides two interfaces: OrderEntry and AccountPayable, and the Order component requires the Person interface.

Component Diagram



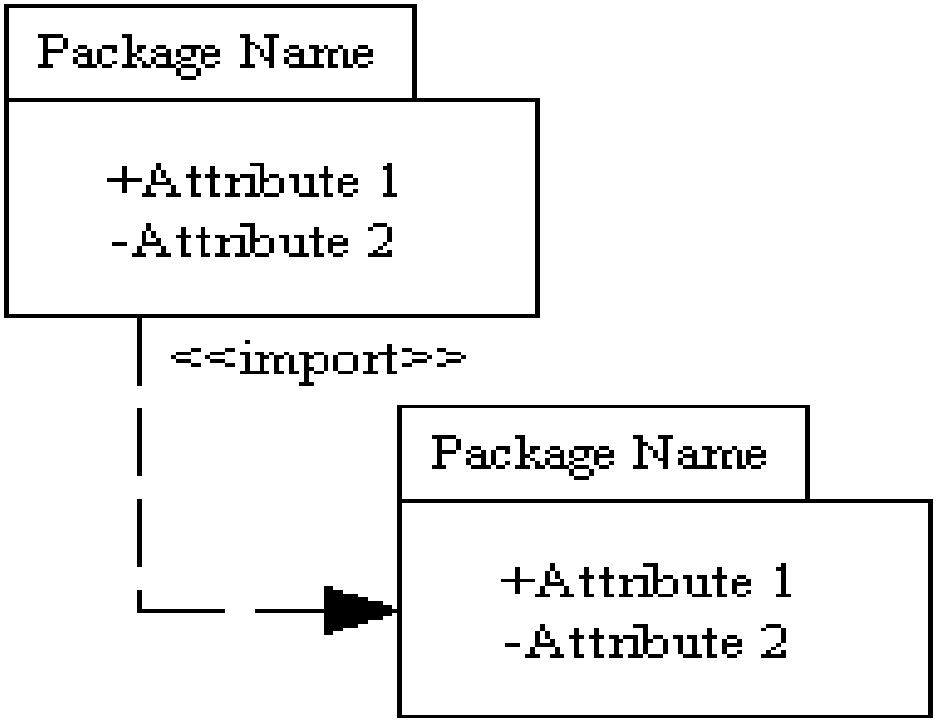
Package Diagram

- Package diagrams organize the elements of a system into related groups to minimize dependencies among them.
- Notations
  - Package

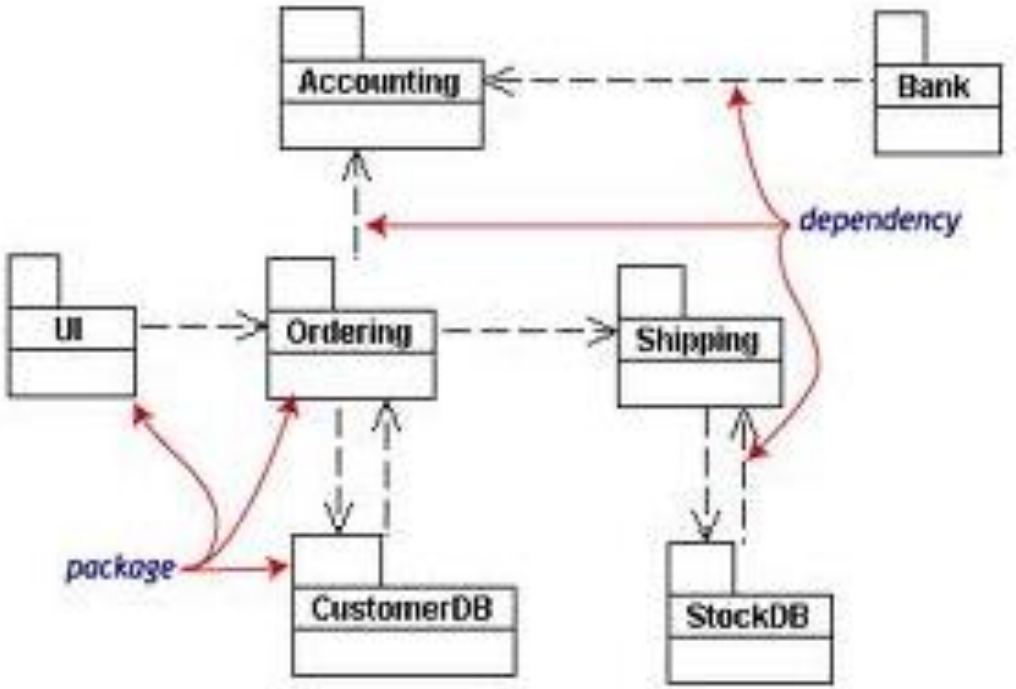


Package Diagram

- Dependency
  - Dependency defines a relationship in which changes to one package will affect another package.



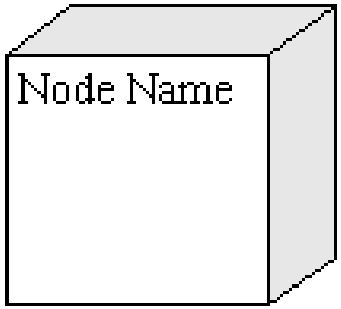
Package Diagram

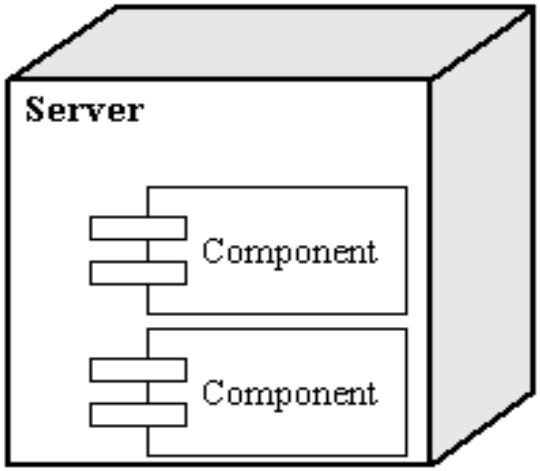


Deployment diagram

- The deployment diagram shows how a system will be physically deployed in the hardware environment. Its purpose is to show where the different components of the system will physically run and how they will communicate with each other.
- Notation

Node - A node is a physical resource that executes code components.





Deployment diagram