SOFTWARE ENGINEERING

UNIT III - SOFTWARE DESIGN

Prof. Rahul Pawar
Assistant Professor,
School of CS&IT,
MCA Department,
Knowledge Campus Jayanagar

Design Diagrams: UML

- Unified Modeling Language
- UML is a way of visualizing a software program using a collection of diagrams.
- UML is accepted by the Object Management Group (OMG) as the standard for modeling software development.
- Types of UML Diagrams
    - Structural diagrams and behavioral or interaction diagrams.

- Structural UML diagrams
  - Class diagram
  - Package diagram
  - Object diagram
  - Component diagram
  - Composite structure diagram
  - Deployment diagram
- Behavioral UML diagrams
  - Activity diagram
  - Sequence diagram
  - Use case diagram
  - State diagram
  - Communication diagram
  - Interaction diagram
  - Timing diagram

Use Case Diagrams

- **Scenario** - a specific sequence of actions and interactions between actors and the system
- **Use case** - a collection of related success and failure scenarios, describing actors using the system to support a goal.
- **Actors** - something with a behavior or role, e.g., a person, another system, organization.

Purposes of use case diagrams

- Used to gather the requirements of a system.
- Used to get an outside view of a system.
- Identify the external and internal factors influencing the system.

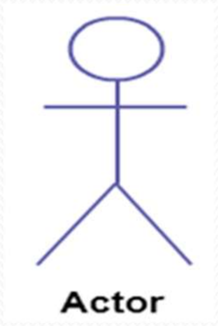
- How to Draw a Use Case Diagram?
  - Show the interaction among the requirements.
  - Actors can be a human user, some internal applications, or may be some external applications. To draw a use case diagram, the following items should be identified.
    - Functionalities to be represented as use case
    - Actors
    - Relationships among the use cases and actors.

Guidelines to be followed to draw an efficient use case diagram

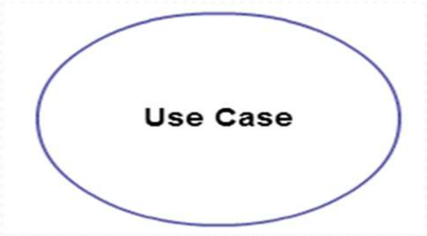
- Use case diagrams are drawn to capture the functional requirements of a system.
  - The name of a use case is very important. The name should be chosen in such a way so that it can identify the functionalities performed.
  - Give a suitable name for actors.
  - Show relationships and dependencies clearly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

Symbols and Notations

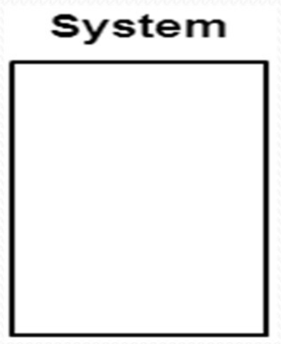
- ***Actor :*** any entity that performs a role in one given system. This could be a person, organization or an external system and usually drawn like skeleton shown below.



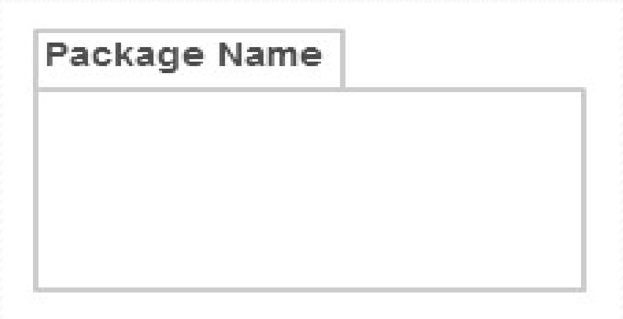
- ***Use Case :*** represents a function or an action within the system. It's drawn as an oval and named with the function.



- ***System:*** The system is used to define the scope of the use case and drawn as a rectangle. This an optional element but useful when you're visualizing large systems.



- ***Package :*** The package is another optional element that is extremely useful in complex diagrams. Similar to class diagrams, packages are used to group together use cases. They are drawn like the image shown below.



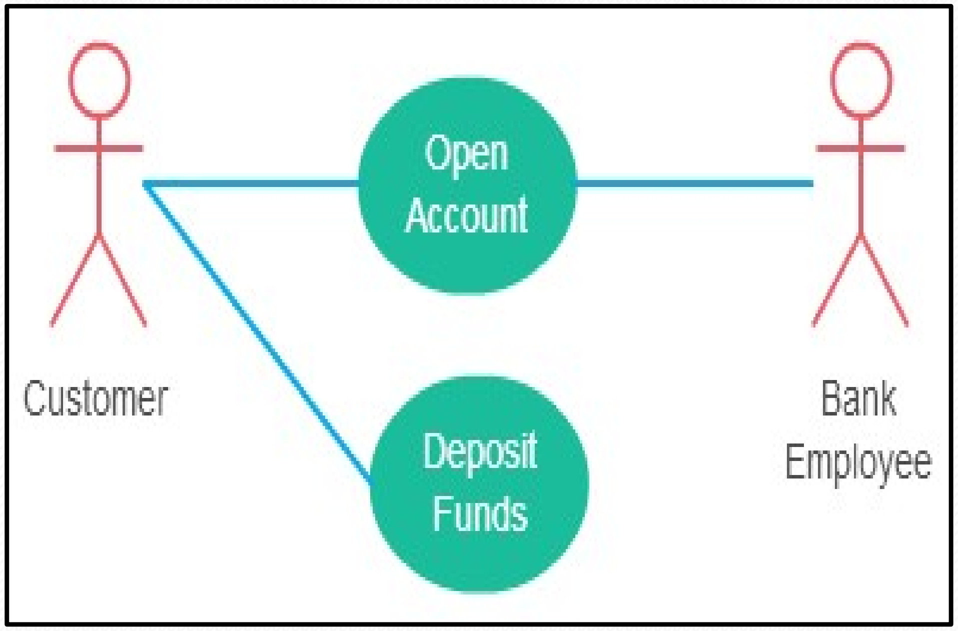
- There can be 5 relationship types in a use case diagram.
  - Association between actor and use case
  - Generalization of an actor
  - Extend between two use cases
  - Include between two use cases
  - Generalization of a use case

Relationships

- Arrow points to the base use case when using <<extend>>
- <<extend>> can have optional extension conditions
- Arrow points to the included use case when using <<include>>
- Both <<extend>> and <<include>> are shown as dashed arrows.
- Actor and use case relationship don’t show arrows.

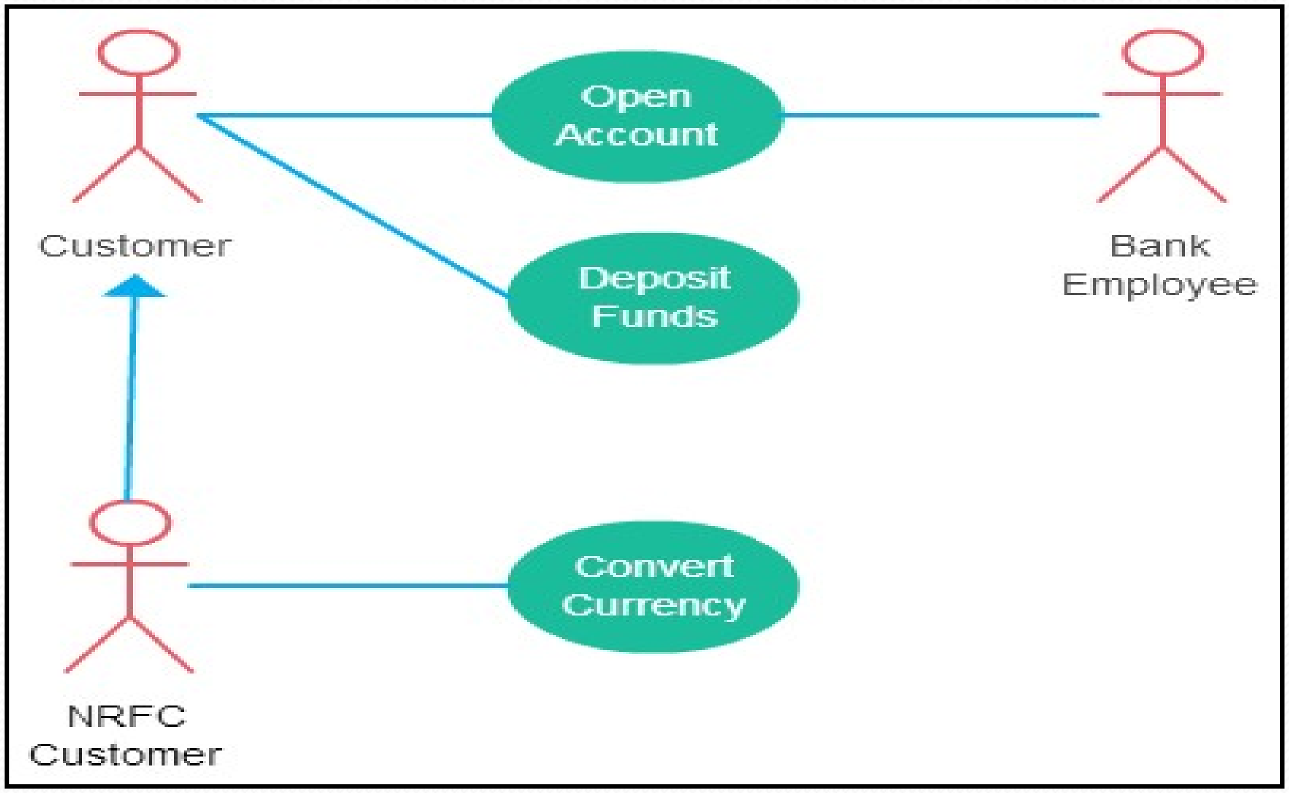
Association Between Actor and Use Case

- An actor must be associated with at least one use case.
- An actor can be associated with multiple use cases.
- Multiple actors can be associated with a single use case.



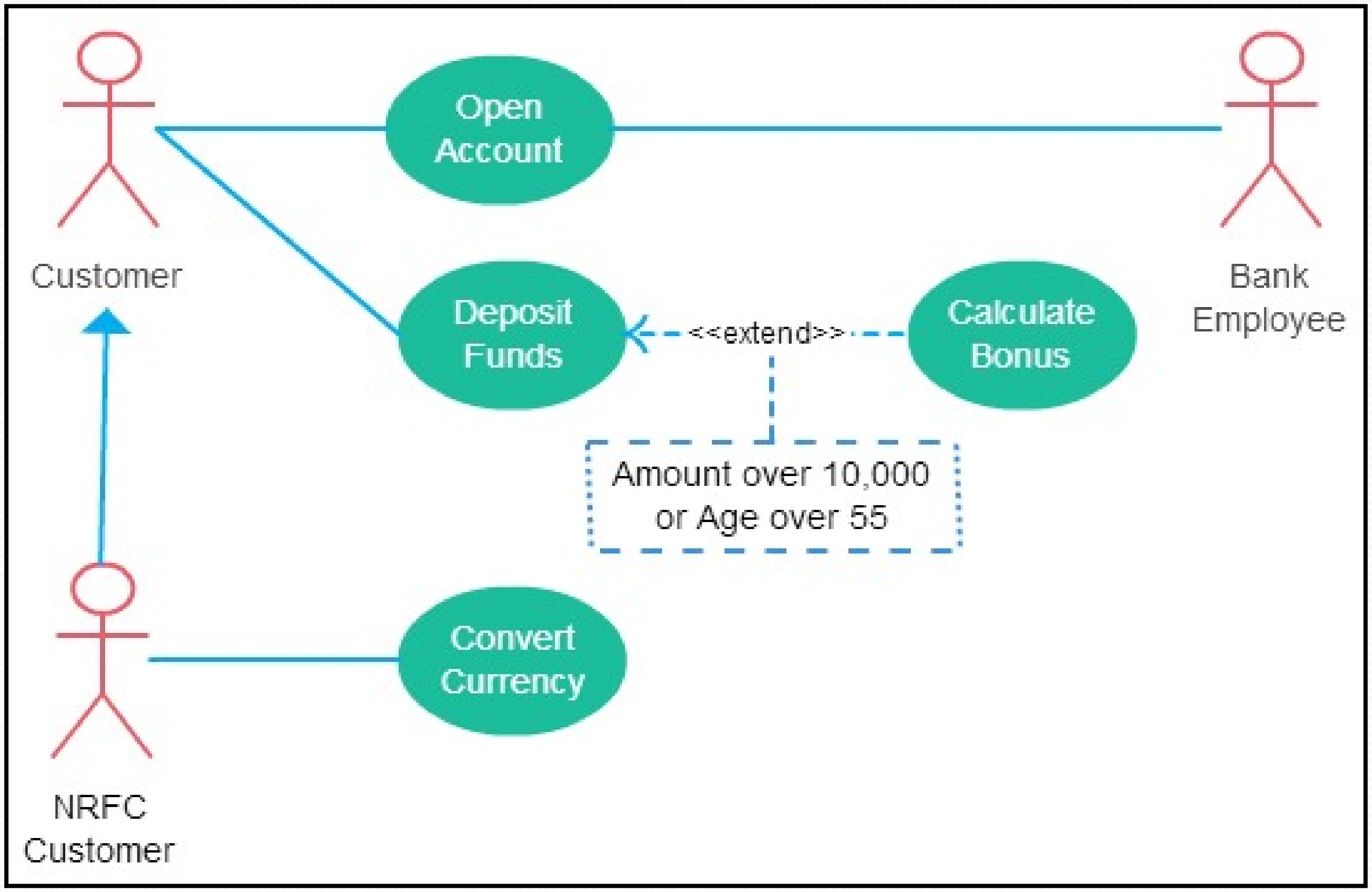
Generalization of an Actor

- Generalization -means that one actor can inherit the role of the other actor.
- The descendant has one or more use cases that are specific to that role.



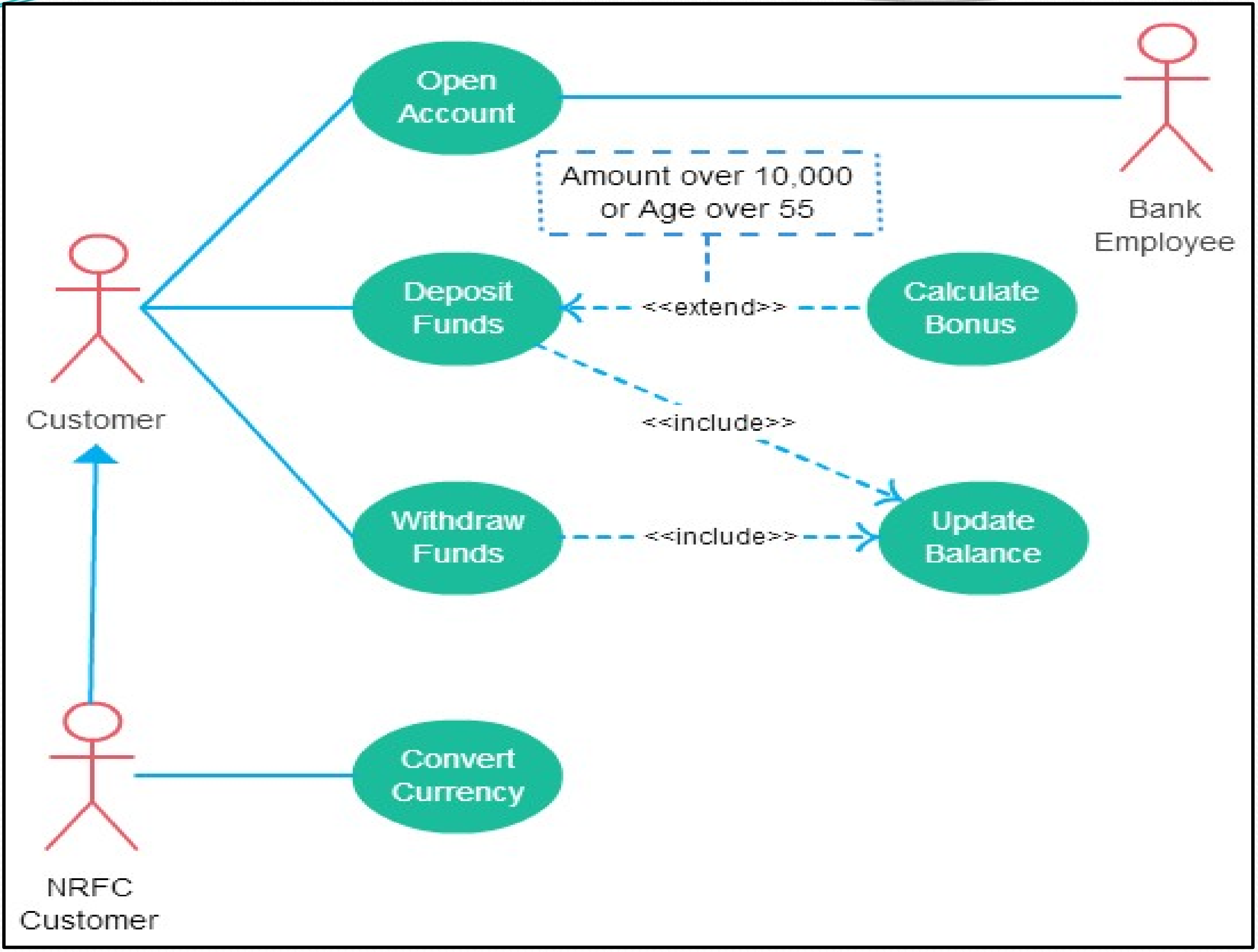
Extend Relationship Between Two Use Cases

- As the name implies it extends the base use case and adds more functionality to the system.
- The extending use case is dependent on the extended (base) use case. In the example diagram the "Calculate Bonus" use case doesn't make much sense without the "Deposit Funds" use case.
- The extending use case is usually optional and can be triggered conditionally. In the diagram, the extending use case is triggered only for deposits over 10,000 or when the age is over 55.
- The extended (base) use case must be meaningful on its own. This means it should be independent and must not rely on the behavior of the extending use case.


Customer
Open Account
Bank Employee
Deposit Funds
<<extend>>
Calculate Bonus
Amount over 10,000 or Age over 55
NRFC Customer
Convert Currency

Include Relationship Between Two Use Cases

- Shows that the behavior of the included use case is part of the including (base) use case.
- The main reason for this is to reuse the common actions across multiple use cases.
- In some situations, this is done to simplify complex behaviors.
- Few things to consider when using the <<include>> relationship.
- The base use case is incomplete without the included use case.
- The included use case is mandatory and not optional.


Open Account
Bank Employee
Amount over 10,000 or Age over 55
Customer
Deposit Funds
<<extend>>
Calculate Bonus
<<include>>
Withdraw Funds
<<include>>
Update Balance
NRFC Customer
Convert Currency

Class Diagrams

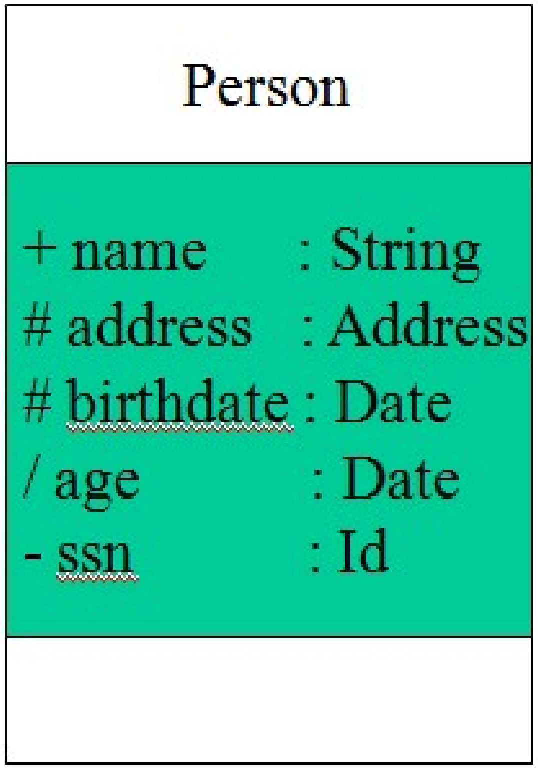
- A type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.
- The purpose of the class diagram can be summarized as follows -
  - Analysis and design of the static view of an application.
  - Describe responsibilities of a system.
  - Base for component and deployment diagrams.
  - To perform Forward and reverse engineering.
  - Construction of software using object oriented languages

The standard class diagram is composed of three sections:

- **Upper section:** Contains the name of the class. This section is always required
- **Middle section:** Contains the attributes of the class. This is only required when describing a specific instance of a class.
- **Bottom section:** Includes class operations (methods). The operations describe how a class interacts with data.

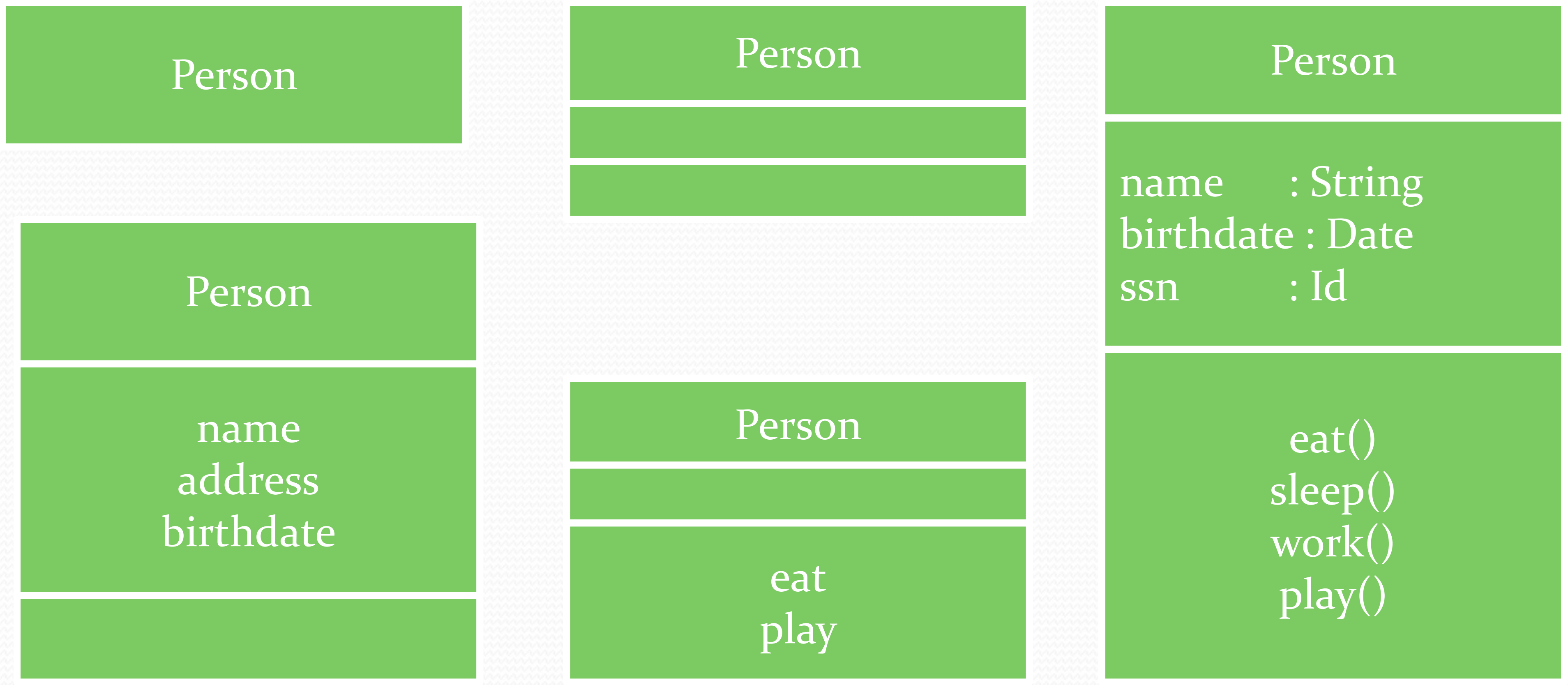
Member access modifiers

- All classes have different access levels depending on the access modifier (visibility). Here are the access levels with their corresponding symbols:
  - Public (+)
  - Private (-)
  - Protected (#)
  - Package (~)
  - Derived (/)
  - Static (underlined)



Depicting Classes

When drawing a class, you needn't show attributes and operation in every diagram.

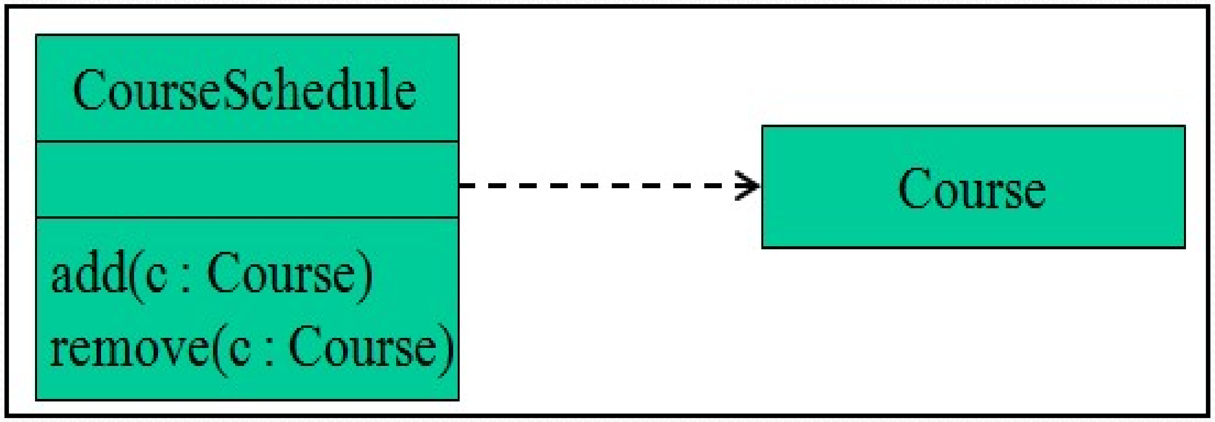


Relationships

- In UML, object interconnections (logical or physical), are modeled as relationships.
- There are three kinds of relationships in UML:
  - Dependencies
  - Generalizations
  - Associations

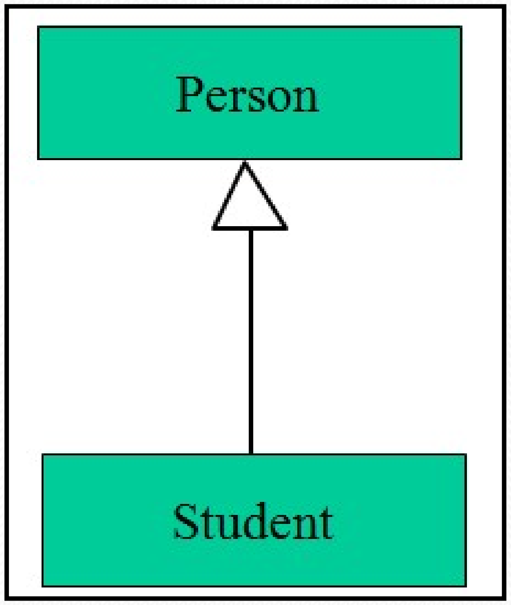
Dependency Relationships

- Indicates a semantic relationship between two or more elements.
- The dependency from *CourseSchedule* to *Course* exists because *Course* is used in both the **add** and **remove** operations of *CourseSchedule*.

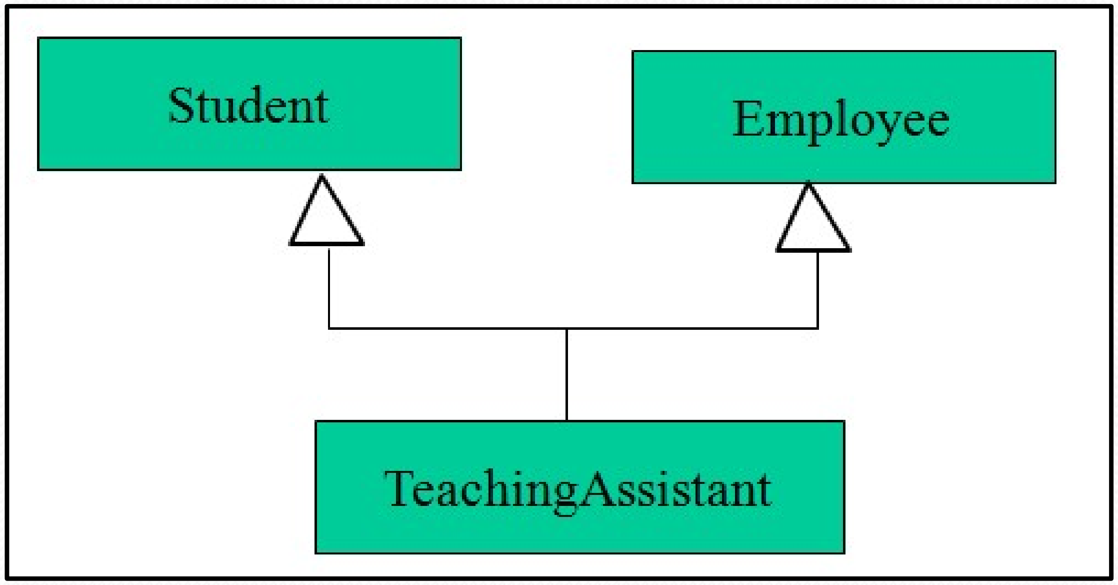


Generalization Relationships

- It connects a subclass to its superclass. It denotes an inheritance of attributes and behavior
- From the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.

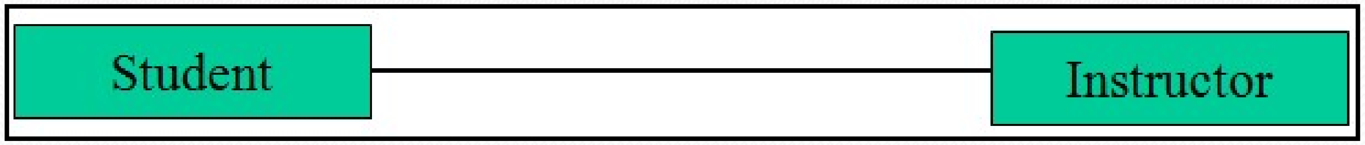


- UML permits a class to inherit from multiple super classes, although some programming languages (*e.g.*, Java) do not permit multiple inheritance.

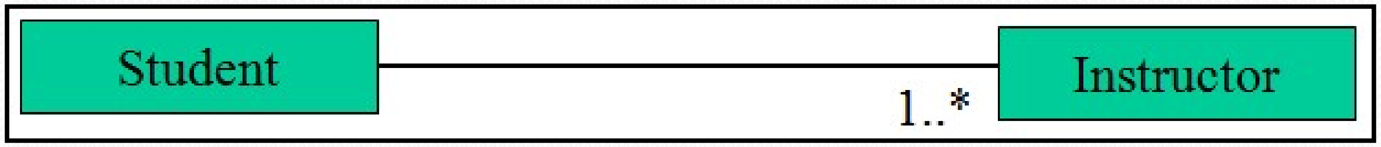


Association Relationships

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

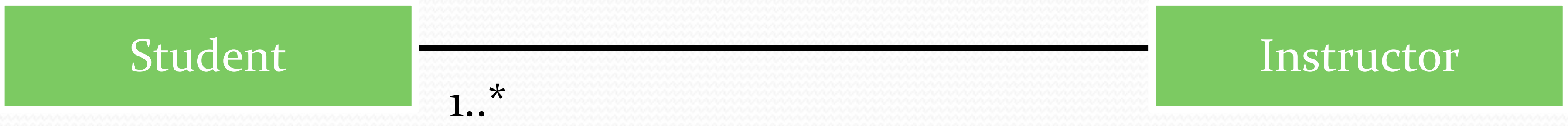


- We can indicate the *multiplicity* of an association
- The example indicates that a *Student* has one or more *Instructors*:

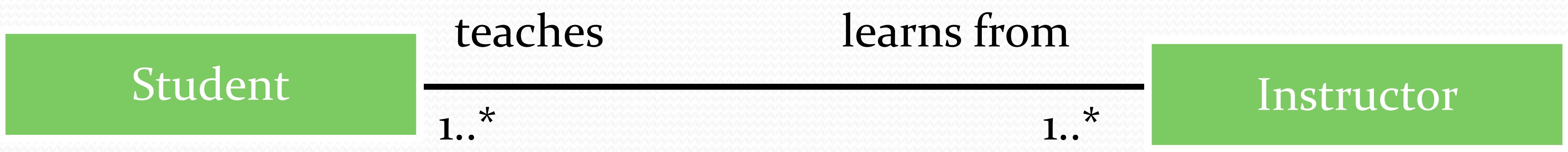


Association Relationships (Cont'd)

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

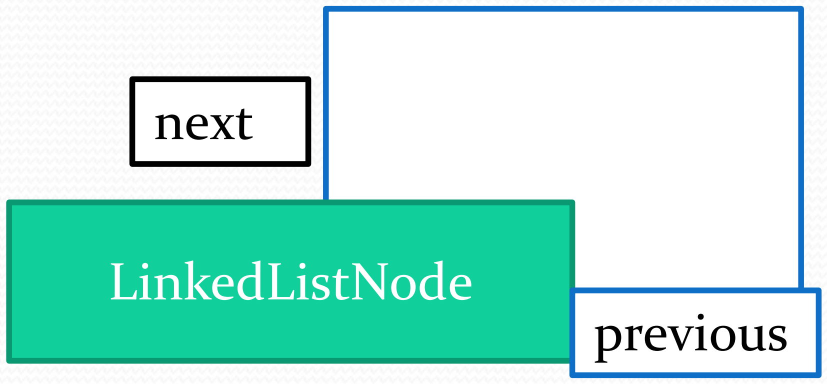


indicate the behavior of an object in an association (*i.e.,* the *role* of an object) using *rolenames.*



Association Relationships (Cont'd)

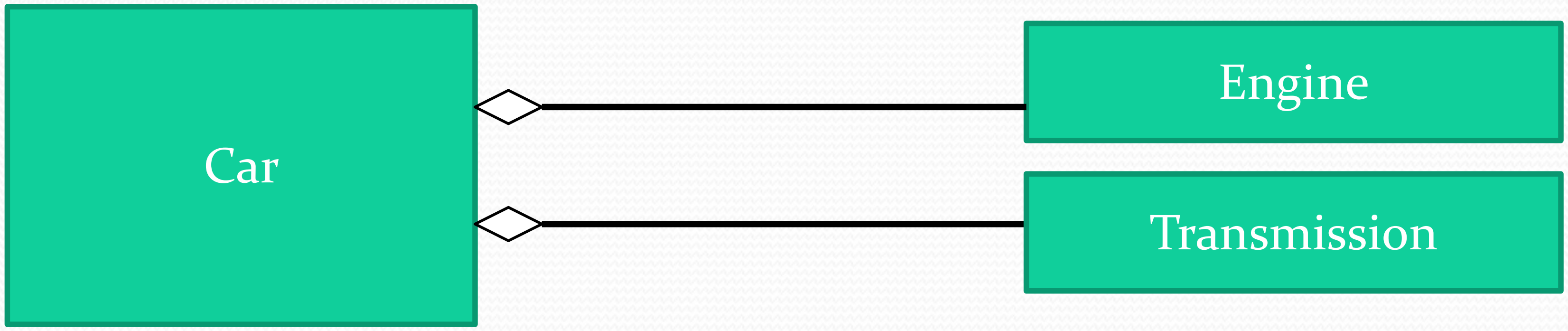
A class can have a *self association*.



Association Relationships (Cont'd)

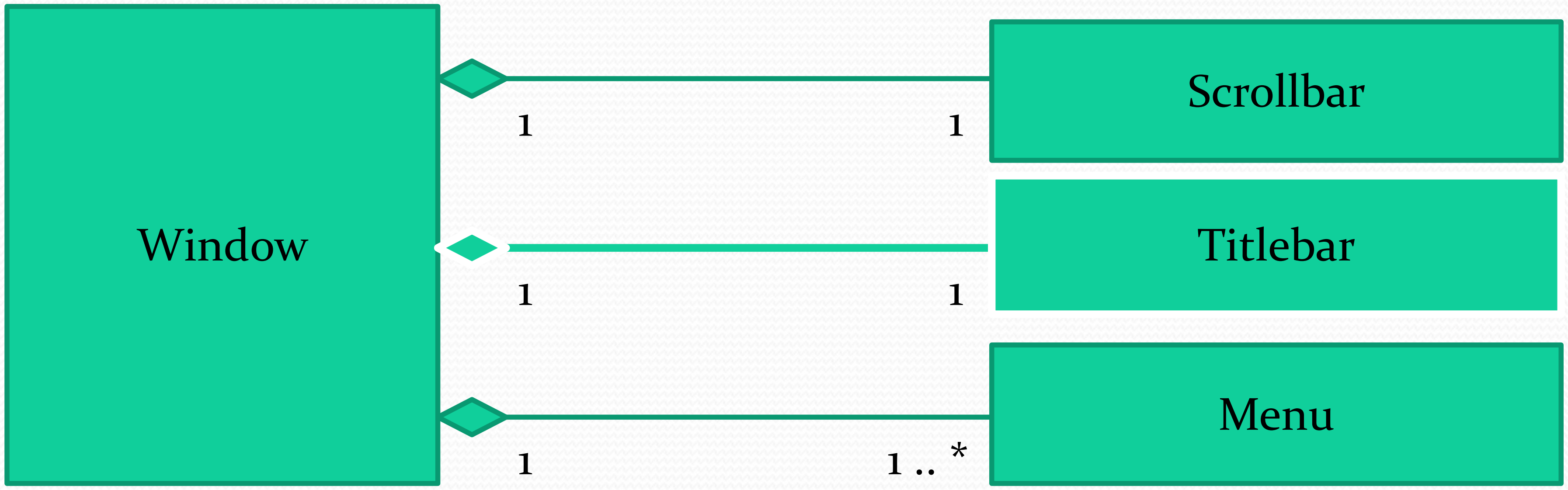
We can model objects that contain other objects by way of special associations called ***Aggregations And Compositions***.

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

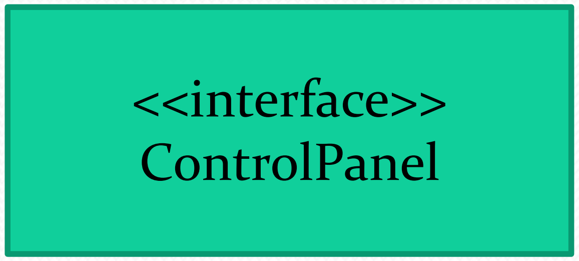


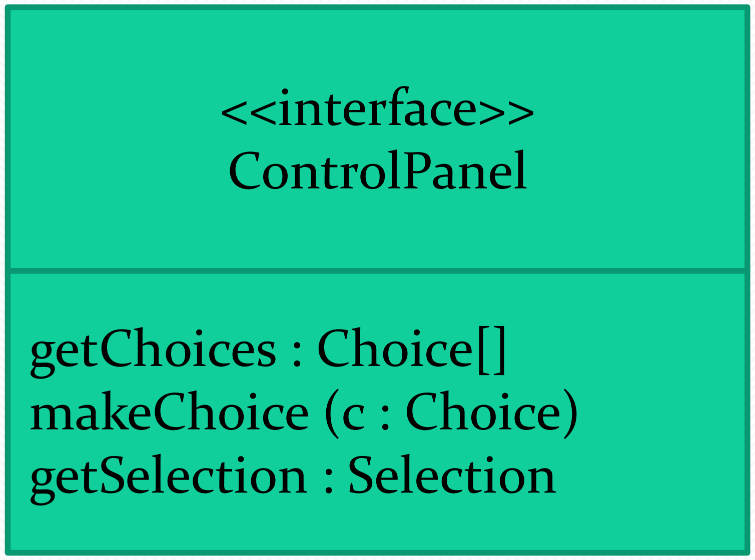
Association Relationships (Cont'd)

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



Interfaces

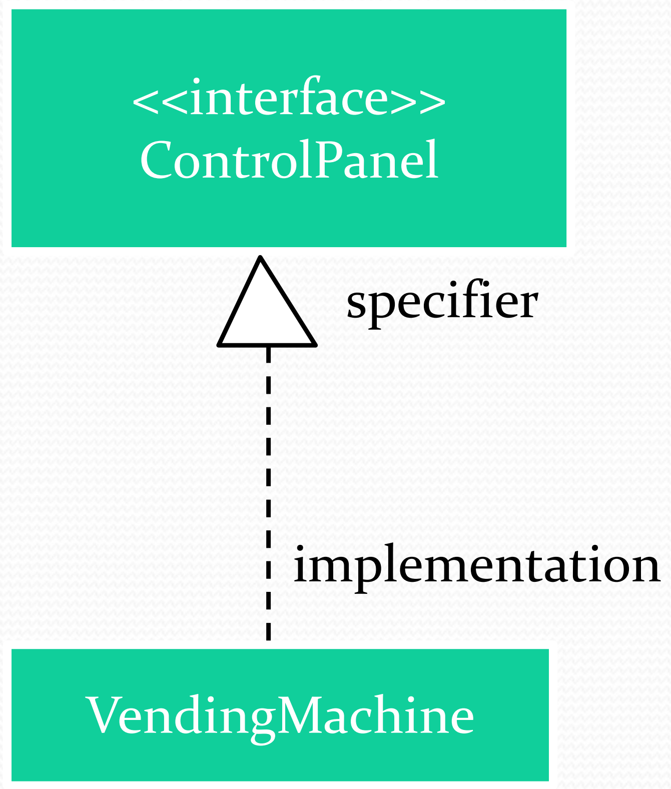




An *interface* is a named set of operations that specifies the behavior of objects without showing their inner structure. It can be rendered in the model by a one- or two-compartment rectangle, with the *stereotype* <<interface>> above the interface name.

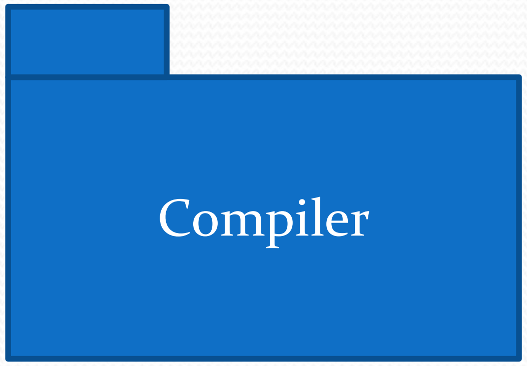
Interfaces do not get instantiated. They have no attributes or state. Rather, they specify the services offered by a related class.

Interface Realization Relationship



A *realization* relationship connects a class with an interface that supplies its behavioral specification. It is rendered by a dashed line with a hollow triangle towards the specifier.

Packages



A *package* is a container-like element for organizing other elements into groups.

A package can contain classes and other packages and diagrams.

Packages can be used to provide controlled access between classes in different packages.

Interaction Diagrams

Two Types of Interaction diagrams defined in UML

- Sequence Diagram
  - Emphasizes the time ordering of those messages
- Collaboration Diagram
  - Emphasizes the structural organization of objects that send and receive messages via the method invocation

The purpose of interaction diagram is –

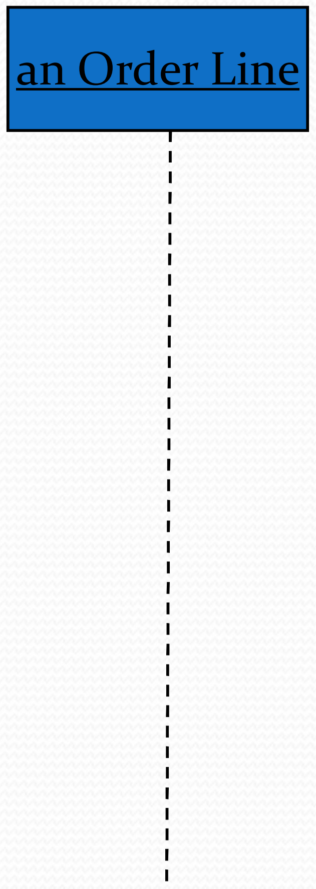
- To capture the dynamic behavior of a system.
- To describe the message flow in the system.
- To describe the structural organization of the objects.
- To describe the interaction among objects.

Sequence Diagram

A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. It shows a set of objects and the messages sent and received by those objects.

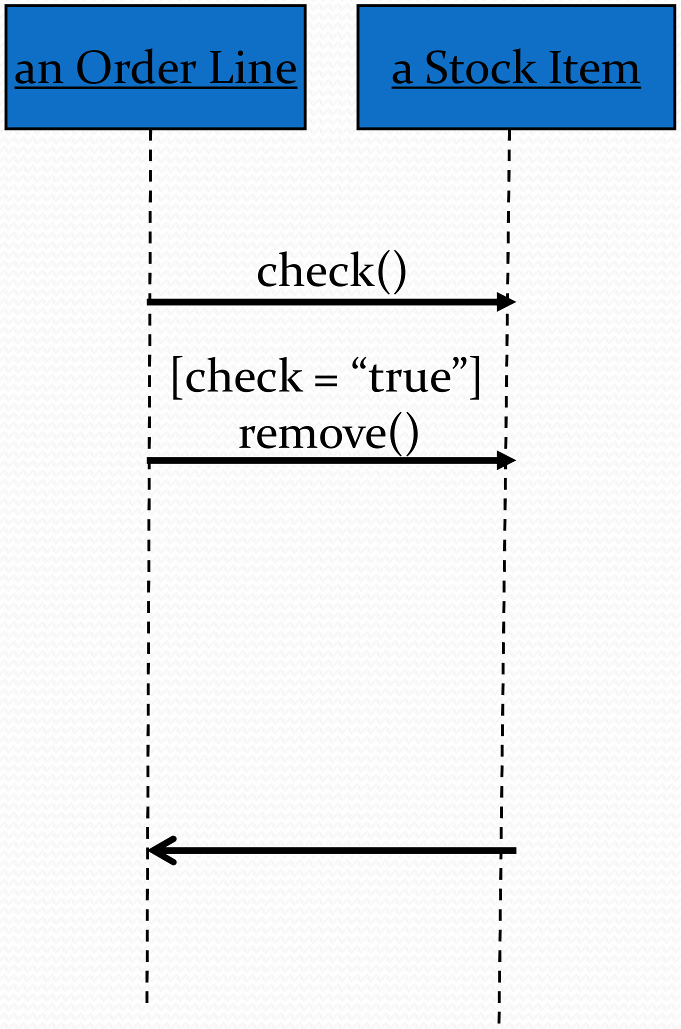
Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis.

Sequence Diagram



An object in a sequence diagram is rendered as a box with a dashed line descending from it. The line is called the *object lifeline*, and it represents the existence of an object over a period of time.

Sequence Diagram

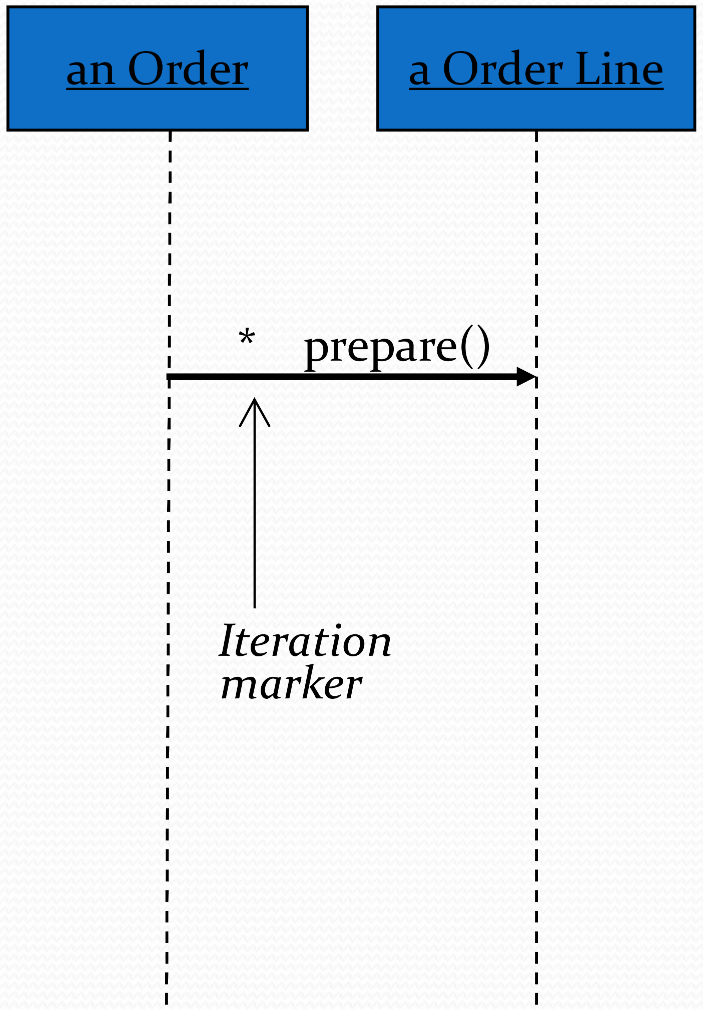


Messages are rendered as horizontal arrows being passed from object to object as time advances down the object lifelines. Conditions (such as [check = “true”]) indicate when a message gets passed.

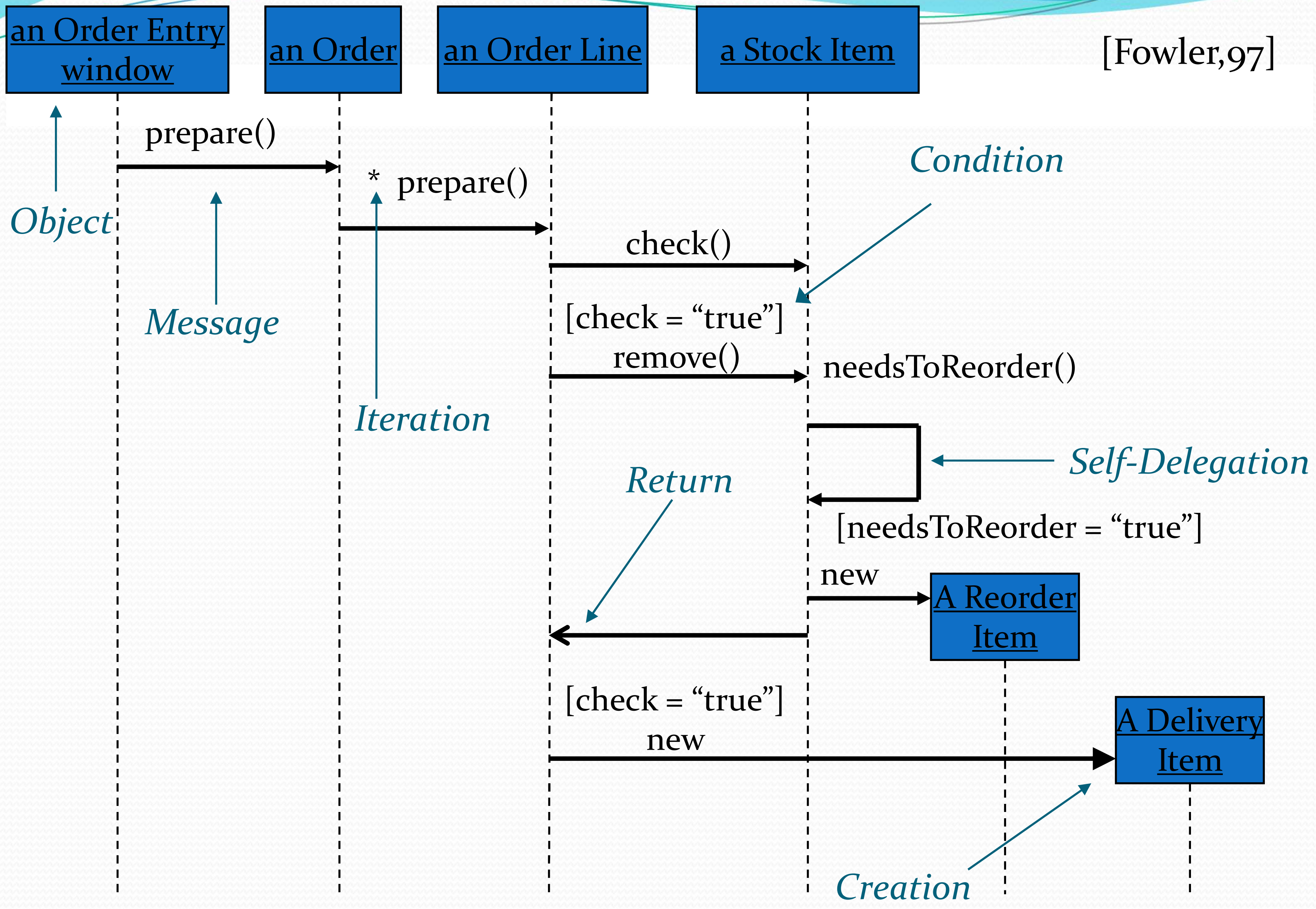
Notice that the bottom arrow is different. The arrow head is not solid, and there is no accompanying message.

This arrow indicates a **return** from a previous message, not a new message.

Sequence Diagram



An iteration marker, such as * (as shown), or *[i = 1..n] , indicates that a message will be repeated as indicated.


an Order Entry window
an Order
an Order Line
a Stock Item
[Fowler,97]
prepare()
Object
Message
* prepare()
Iteration
check()
Condition
[check = "true"] remove()
needsToReorder()
Self-Delegation
Return
[needsToReorder = "true"] new
A Reorder Item
[check = "true"] new
A Delivery Item
Creation

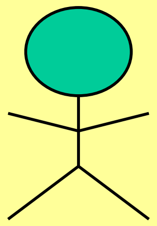
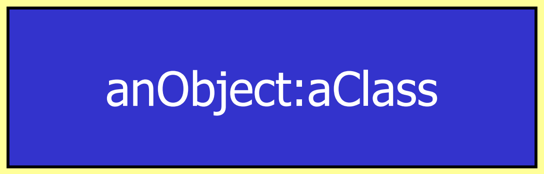
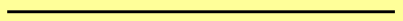
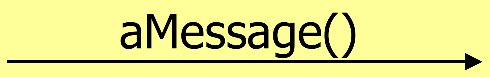
Collaboration Diagram

- It shows the object organization .
- In the collaboration diagram, the method call sequence is indicated by some numbering technique.
- The number indicates how the methods are called one after another.
- To choose between these two diagrams, emphasis is placed on the type of requirement.
- If the time sequence is important, then the sequence diagram is used. If object organization is required, then collaboration diagram is used.

Collaboration Diagram Elements

- There are three primary elements of a collaboration diagram:
  - Objects
  - Links
  - Messages

Collaboration Diagram notations

AN ACTOR	
AN OBJECT	
AN ASSOCIATION	
A MESSAGE	

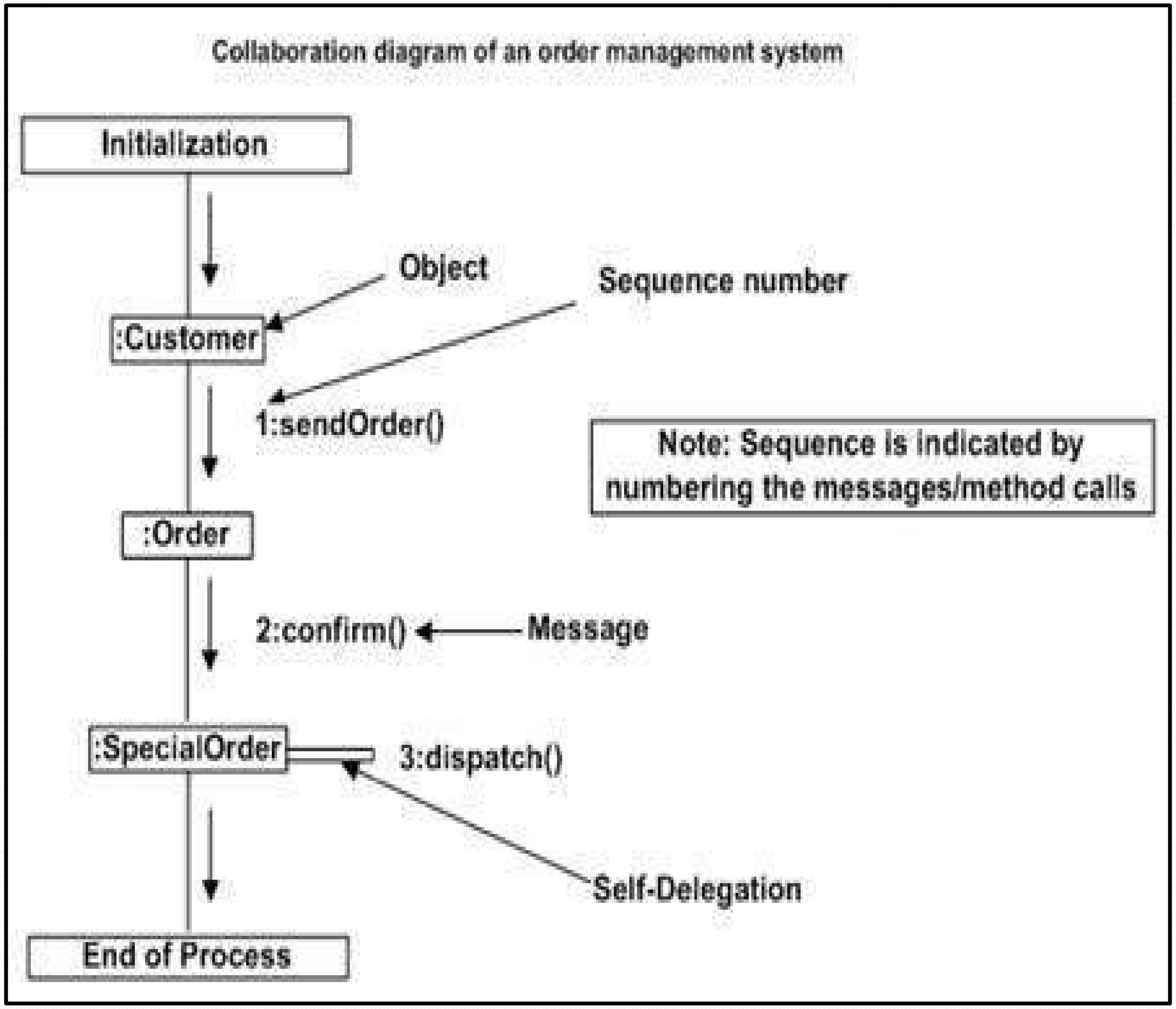
Objects

- **Objects** rectangles containing the object signature
- object signature:
  - **object name : object Class**
  - object name (optional) - starts with lowercase letter
  - class name (mandatory) - starts with uppercase letter
- Objects connected by lines and actor can appear

Iterating Messages

- Collaboration diagrams use syntax similar to sequence diagrams to indicate that either a message iterates (is run multiple times) or is run conditionally
  - An asterisk (*) indicates that a message runs more than once
  - Or the number of times a message is repeated can be shown by numbers (for example, 1..5)

Collaboration diagram for Order Management system



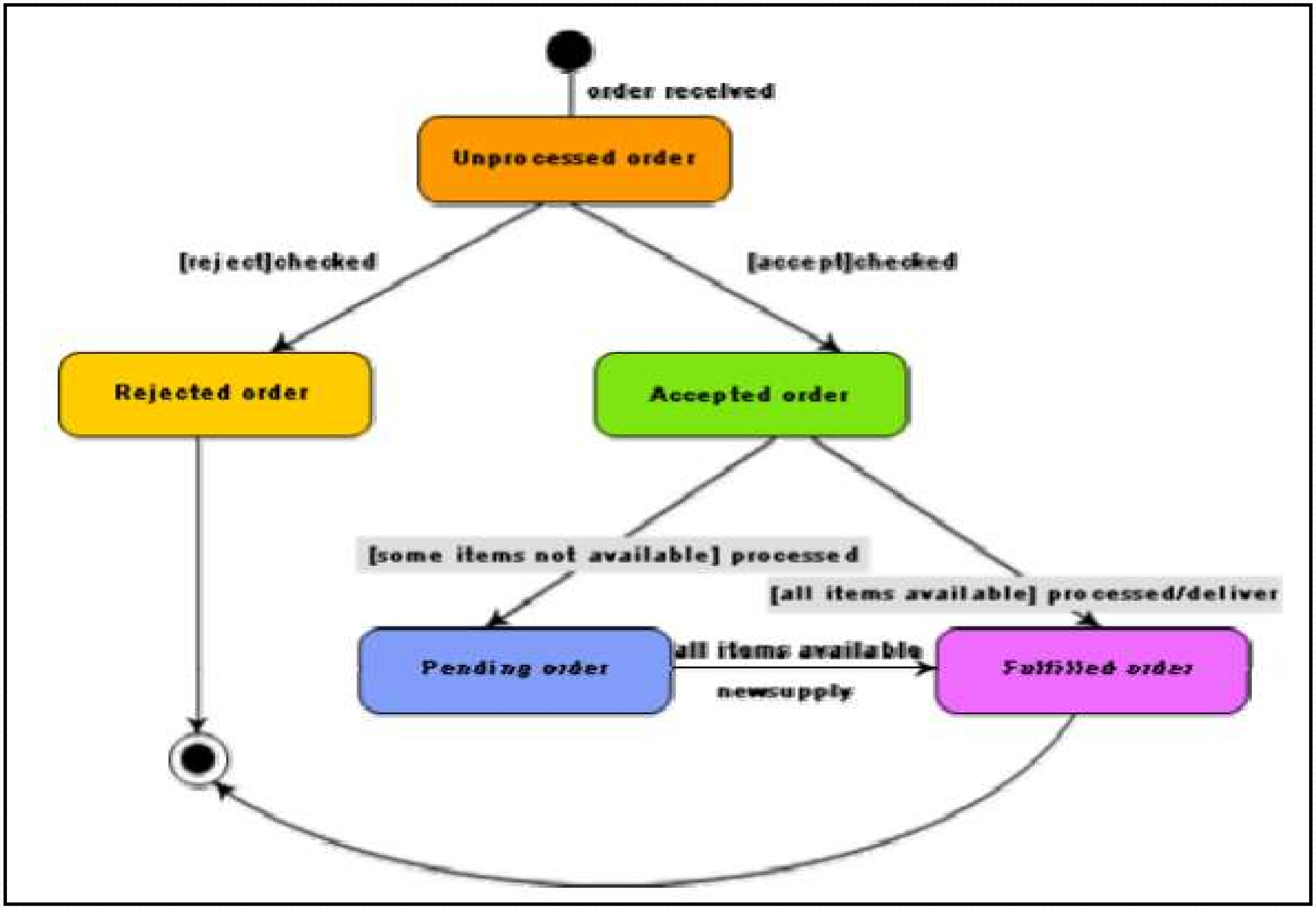
State Chart Diagrams

- Describes different states of a component/object in a system.
- A Statechart diagram describes a state machine. State machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events.
- Purposes of using Statechart diagrams –
  - To model the dynamic aspect of a system.
  - To model the life time of a reactive system.
  - To describe different states of an object during its life time.
  - Define a state machine to model the states of an object.

The basic elements of the state chart diagram are as follows:

- • Initial state. This is represented as a filled circle.
- • Final state. This is represented by a filled circle inside a larger circle.
- • State. These are represented by rectangles with rounded corners.
- • Transition. A transition is shown as an arrow between two states. Normally, the name of the event which causes the transition is places along side the arrow. A guard to the transition can also be assigned. A guard is a Boolean logic condition. The transition can take place only if the grade evaluates to true. The syntax for the label of the transition is shown in 3 parts: event[guard]/action.

An example state chart for the order object of the Trade House Automation software

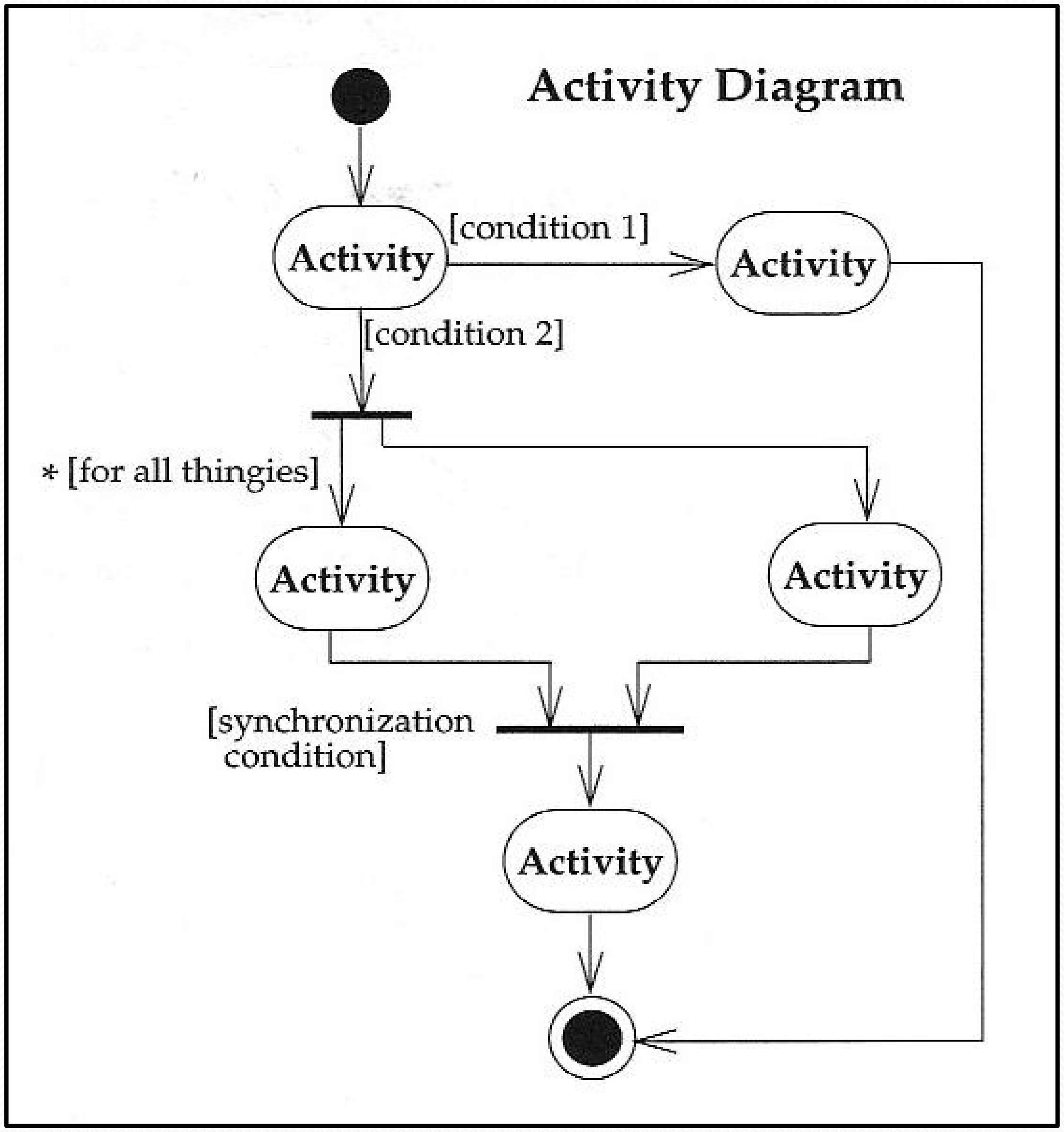


Activity Diagrams

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

- Activity arrangement
  - Sequential – one activity is followed by another
  - Parallel – two or more sets of activities are performed concurrently, and order is irrelevant
    - Interleaving is permitted – we can jump between the parallel flows

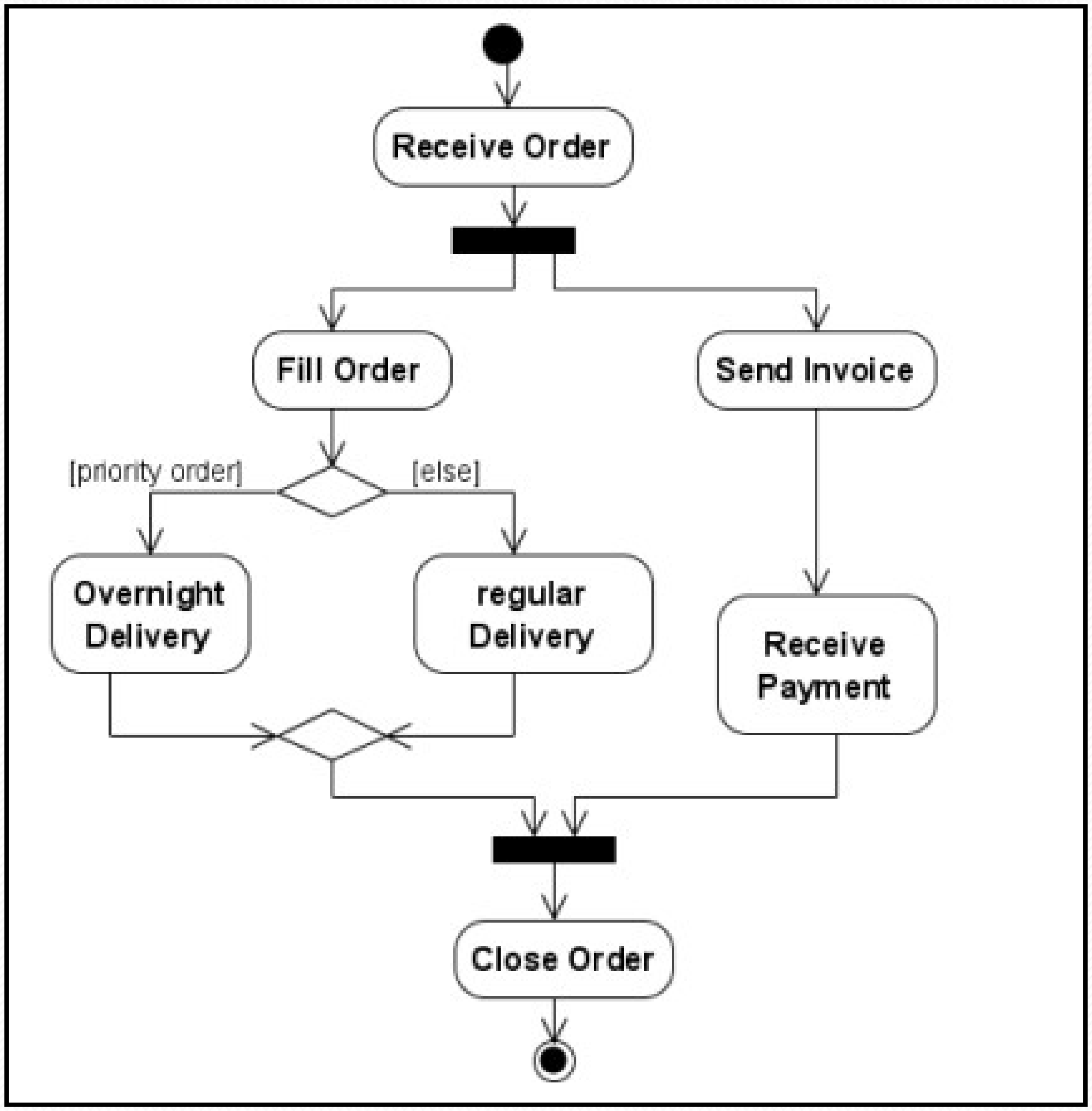
Activity Diagrams - Notation



- Start at the top black circle
- If condition 1 is TRUE, go right; if condition 2 is TRUE, go down
- At first bar (a synchronization bar), break apart to follow 2 parallel paths (Fork)
- At second bar, come together to proceed only when both parallel activities are done (Join)

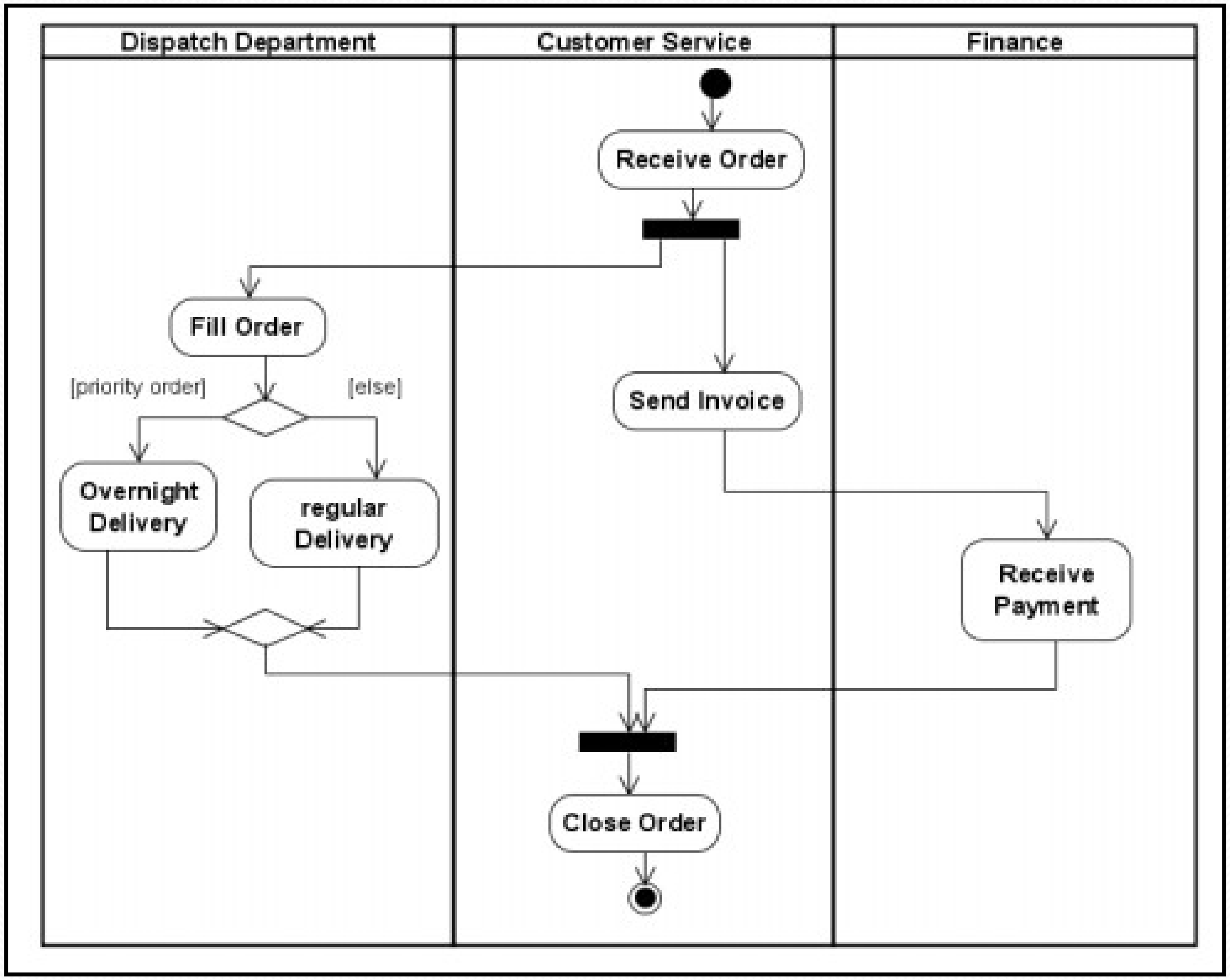
Activity Diagrams – Notation (concluded)

- Activity – an oval
- Trigger – path exiting an activity
- Guard – each trigger has a guard, a logical expression that evaluates to "true" or "false"
- Synchronization Bar – can break a trigger into multiple triggers operating in parallel or can join multiple triggers into one when all are complete
- Decision Diamond – used to describe nested decisions (the first decision is indicated by an activity with multiple triggers coming out of it)


Receive Order
Fill Order
Send Invoice
[priority order]
[else]
Overnight Delivery
regular Delivery
Receive Payment
Close Order

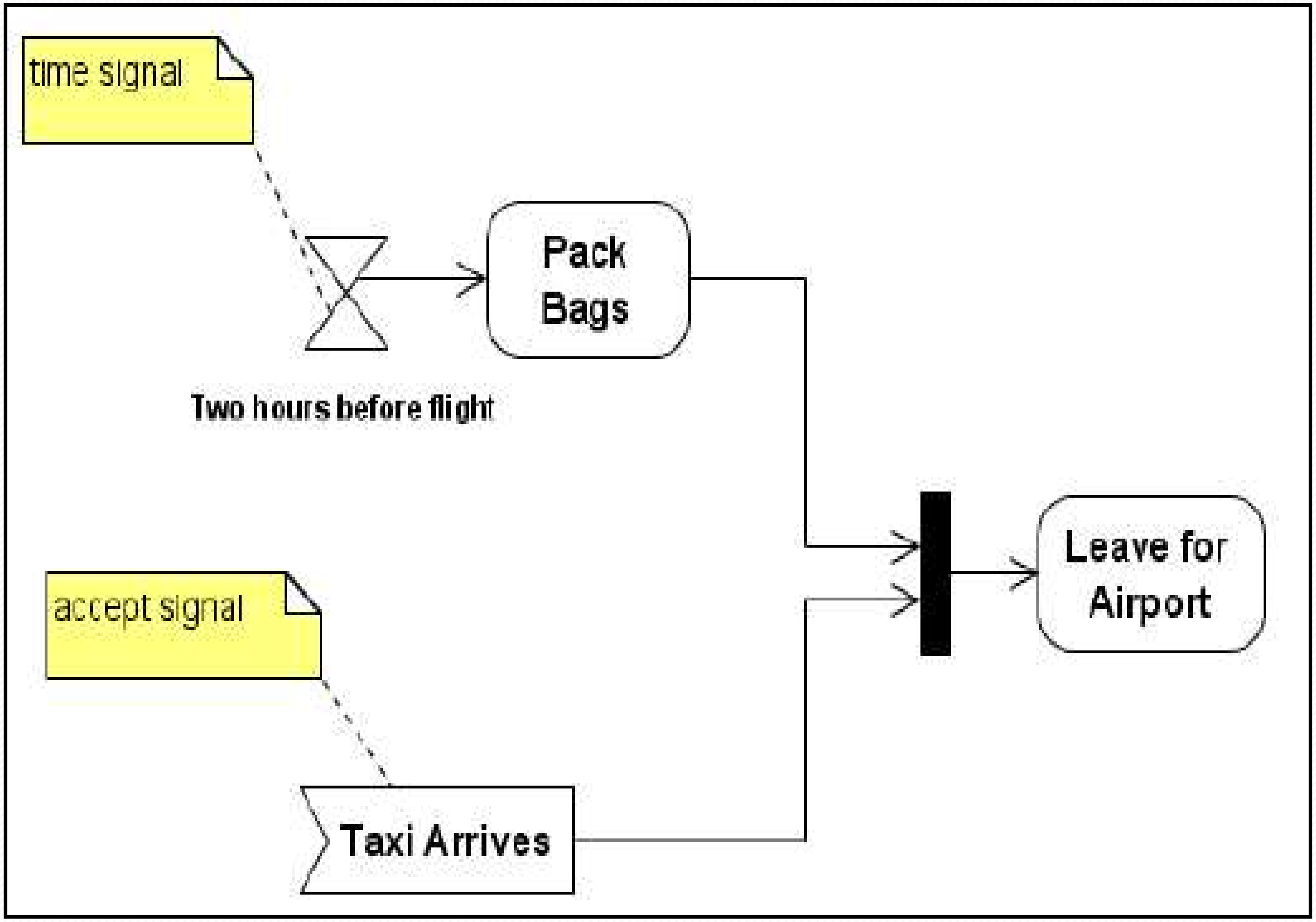
- Swimlanes (or activity partitions) indicate where activities take place.
- Swimlanes can also be used to identify areas at the technology level where activities are carried out
- Swimlanes allow the partition an activity diagram so that parts of it appear in the swimlane relevant to that element in the partition

Swimlanes – Example



Sending and Receiving Signals

- In activity diagrams, signals represent interactions with external participants
- Signals are messages that can be sent or received
- A receive signal has the effect of waking up an action in your activity diagram
- Send signals are signals sent to external participants.


time signal
Pack Bags
Two hours before flight
accept signal
Taxi Arrives
Leave for Airport

Design Process

- Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software.
- Initially, the **blueprint** depicts a holistic view of software, i.e. the design is represented at a high-level of abstraction.
- Throughout the design process, the quality of the evolving design is assessed with a series of formal technique reviews or design walkthroughs.

Software Quality Guidelines and Attributes

- Three characteristics serve as a guide for the evaluation of a good design:
  - The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
  - The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
  - The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality Guidelines

In order to evaluate the quality of a design representation, we must establish technical criteria for good design.

1)A design should exhibit an architecture that:

(a) Has been created using recognizable architectural styles or patterns,

(b) Is composed of components that exhibit good design characteristics,

(c) Can be implemented in an evolutionary fashion

2)A design should be modular; that is, the software should be logically partitioned into elements or subsystems

3) A design should contain distinct representations of data, architecture, interfaces, and components.

4) A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

5)A design should lead to components that exhibit independent functional characteristics.

6) A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.

7) A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

8) A design should be represented using a notation that effectively communicates its meaning.

Quality Attribute

- Hewlett-Packard developed a set of software quality attributes that has been given the acronym FURPS. The FURPS quality attributes represent a target for all software design:
  - ***Functionality*:** is assessed by evaluating the features set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
  - ***Usability*:** is assessed by considering human factors, overall visual, consistency, and documentation.
  - ***Reliability*:** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure, the ability to recover from failure, and the predictability of the program.

- ***Performance***: is measured by processing speed, response time, resource consumption, throughput, and efficiency.
- ***Supportability:*** combines the ability to extend the program extensibility, adaptability, serviceability maintainability. In addition, testability, compatibility, configurability, etc.

Design concepts

- What criteria can be used to partition software into individual components?
- How is function or data structure detail separated from a conceptual representation of the software?
- What uniform criteria define the technical quality of a software design?

1. Abstraction

- Highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- Lower levels of abstraction, a more detailed description of the solution is provided.
- As different levels of abstraction are developed- **procedural** and **data abstractions.**
- A ***procedural abstraction*** refers to a sequence of instructions that have a specific and limited function. An example of a procedural abstraction would be the word ***open*** for a door.
- Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.)

- A ***data abstraction*** *is a named collection of data that describes a data object*.
- *In* the context of the procedural abstraction *open, we can define a* ***data abstraction*** called **door.**
- Like any data object, the data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).

2 Architecture

- *Software architecture shows "the overall structure of the software and the ways* in which that structure provides conceptual integrity for a system".
- Architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.
- The aim of the software design is to obtain an architectural framework of a system.
- The more detailed design activities are conducted from the framework.
- A set of architectural patterns enable a software engineer to reuse design-level concepts.

3. Patterns

- A design pattern describes a design structure and that structure solves a particular design problem in a specified content.
  The intent of each design pattern is to provide a description that enables a designer to determine:
  - Whether the pattern is applicable to the current work,
  - Whether the pattern can be reused, and
  - Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

4 .Separation of Concerns

- *Separation of concerns is a design concept that suggests that any complex* problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
- A *concern is a feature or behavior that is* specified as part of the requirements model for the software.
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

5 Modularity

- A software is separately divided into named and addressable components or modules.
- Modularity is the single attribute of a software that permits a program to be managed easily.
- modularize a design (and the resulting program) so that development can be more easily planned;
  - software increments can be defined and delivered;
  - changes can be more easily accommodated;
  - testing and debugging can be conducted more efficiently,
  - and long-term maintenance can be conducted without serious side effects.

6. Information Hiding

- Modules must be specified and designed so that the information like algorithm and data presented in a module is not accessible for other modules that have no need for such information.
- The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance.
- Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

7. Functional Independence

- is the concept of separation and related to the concept of modularity, abstraction and information hiding.
- The functional independence is accessed using two criteria i.e Cohesion and coupling.
- ***Cohesion*** is an indication of the relative functional strength of a module. A cohesive module performs a single task and it requires a small interaction with the other components in other parts of the program. Strive for high cohesion
- ***Coupling*** *is* an indication of the relative interdependence among modules. Always strive for the lowest possible coupling.

8. Refinement

- Refinement is a top-down design approach.
- It is a process of elaboration.
- A program is established for refining levels of procedural details.
- A hierarchy is established by decomposing a statement of function in a stepwise manner till the programming language statement are reached.
- Refinement helps the designer to reveal low-level details as design progresses.
- Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement "elaboration" occurs.

9. Refactoring

- It is a reorganization technique that simplifies the design of a component without changing its function or behavior.
- When software is re-factored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed data structures, or any other design failures that can be corrected to yield a better design.

10.Aspects

- A mechanism for understanding how global requirements affect design.
- Consider two requirements, A and B. Requirement A crosscuts requirement B “if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account.
- An aspect is a representation of a cross-cutting concern.

Object Oriented Design Concepts

- Object Oriented is a popular design approach for analyzing and designing an application.
- Object-oriented concepts are used in the design methods such as classes, objects, polymorphism, encapsulation, inheritance, dynamic binding, information hiding, interface, constructor, destructor.
- The main advantage of object oriented design is that improving the software development and maintainability.
- Another advantage is that faster and low cost development, and creates a high quality software.
- The disadvantage of the object-oriented design is that larger program size and it is not suitable for all types of program.

Design Classes

- A set of design classes refined the analysis class by providing design details.

There are five different types of design classes and each type represents the layer of the design architecture

1. User interface classes:

- These classes are designed for Human Computer Interaction(HCI).
- These interface classes define all abstraction which is required for Human Computer Interaction(HCI).

2. Business domain classes:

- These classes are commonly refinements of the analysis classes.
- These classes are recognized as attributes and methods which are required to implement the elements of the business domain.

3. Process classes

Implement lower-level business abstractions required to fully manage the business domain classes.

4. Persistence classes

Represent data stores (e.g., a database) that will persist beyond the execution of the software.

5. System classes

Implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

Design class characteristic

- **The characteristic of well formed designed class are as follows:**

1. **Complete and sufficient** :
   A design class must be the total encapsulation of all attributes and methods which are required to exist for the class.

2. **Primitiveness:** The method in the design class should fulfill one service for the class.

- If service is implemented with a method then the class should not provide another way to fulfill same thing.

3. **High cohesion :** A cohesion design class has a small and focused set of responsibilities.

- For implementing the set of responsibilities the design classes are applied single-mindedly to the methods and attribute.

4. **Low-coupling :**All the design classes should collaborate with each other in a design model.

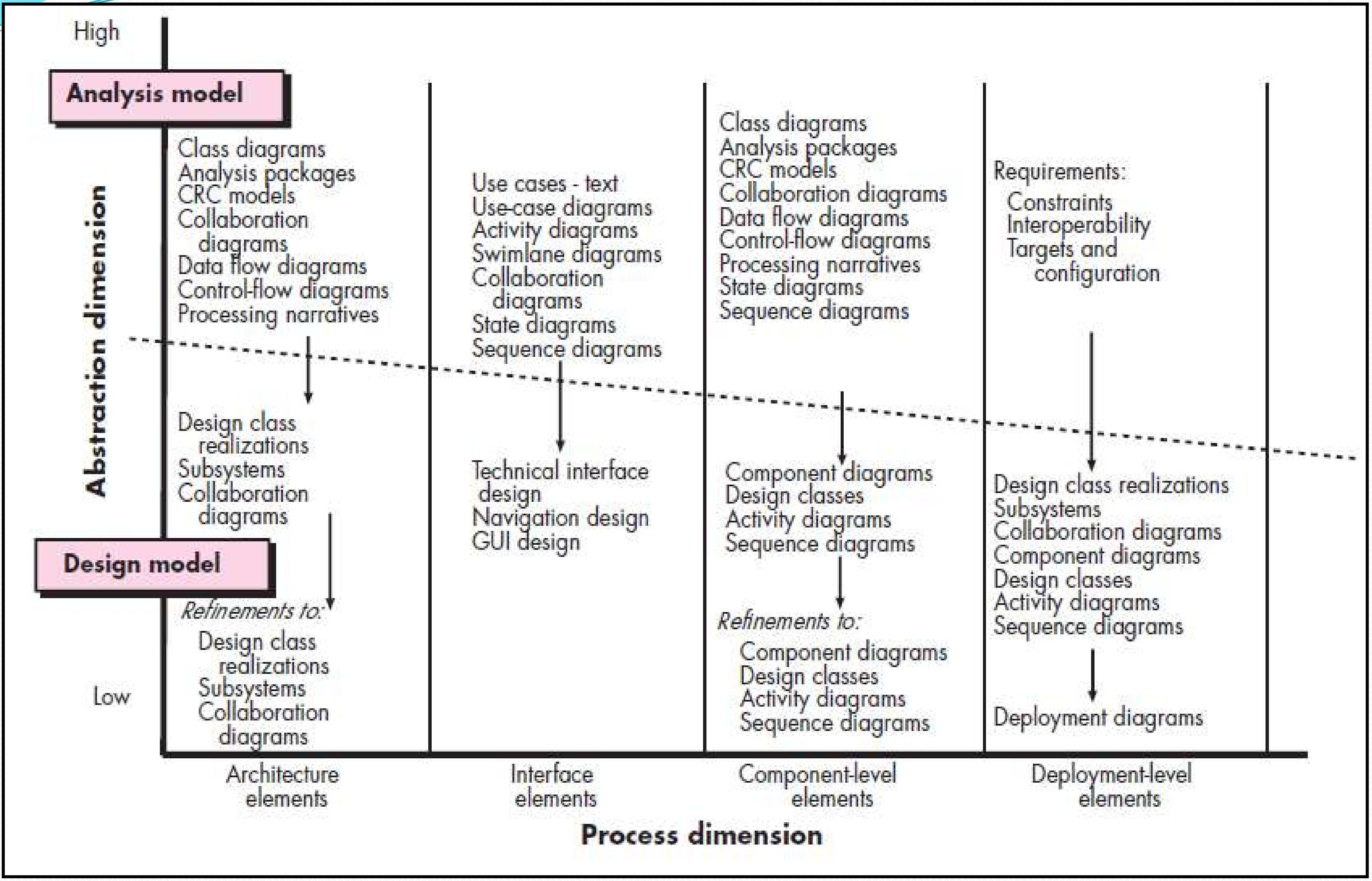
- The minimum acceptable of collaboration must be kept in this model.
- If a design model is highly coupled then the system is difficult to implement, to test and to maintain over time.

Design classes

- Entity classes
- Boundary classes
- Controller classes

- Inheritance—all responsibilities of a superclass is immediately inherited by all subclasses
- Messages—stimulate some behavior to occur in the receiving object
- Polymorphism—a characteristic that greatly reduces the effort required to extend the design

Design Models



Types of design elements:

1. Data design elements

- The data design element produced a model of data that represent a high level of abstraction.
- This model is then more refined into more implementation specific representation which is processed by the computer based system.
- The structure of data is the most important part of the software design.

2. Architectural design elements

- Provides us overall view of the system.
- It is generally represented as a set of interconnected subsystem that are derived from analysis packages in the requirement model.

The architecture model is derived from following sources:

- The information about the application domain to built the software.
- Requirement model elements like data flow diagram or analysis classes, relationship and collaboration between them.
- The architectural style and pattern as per availability.

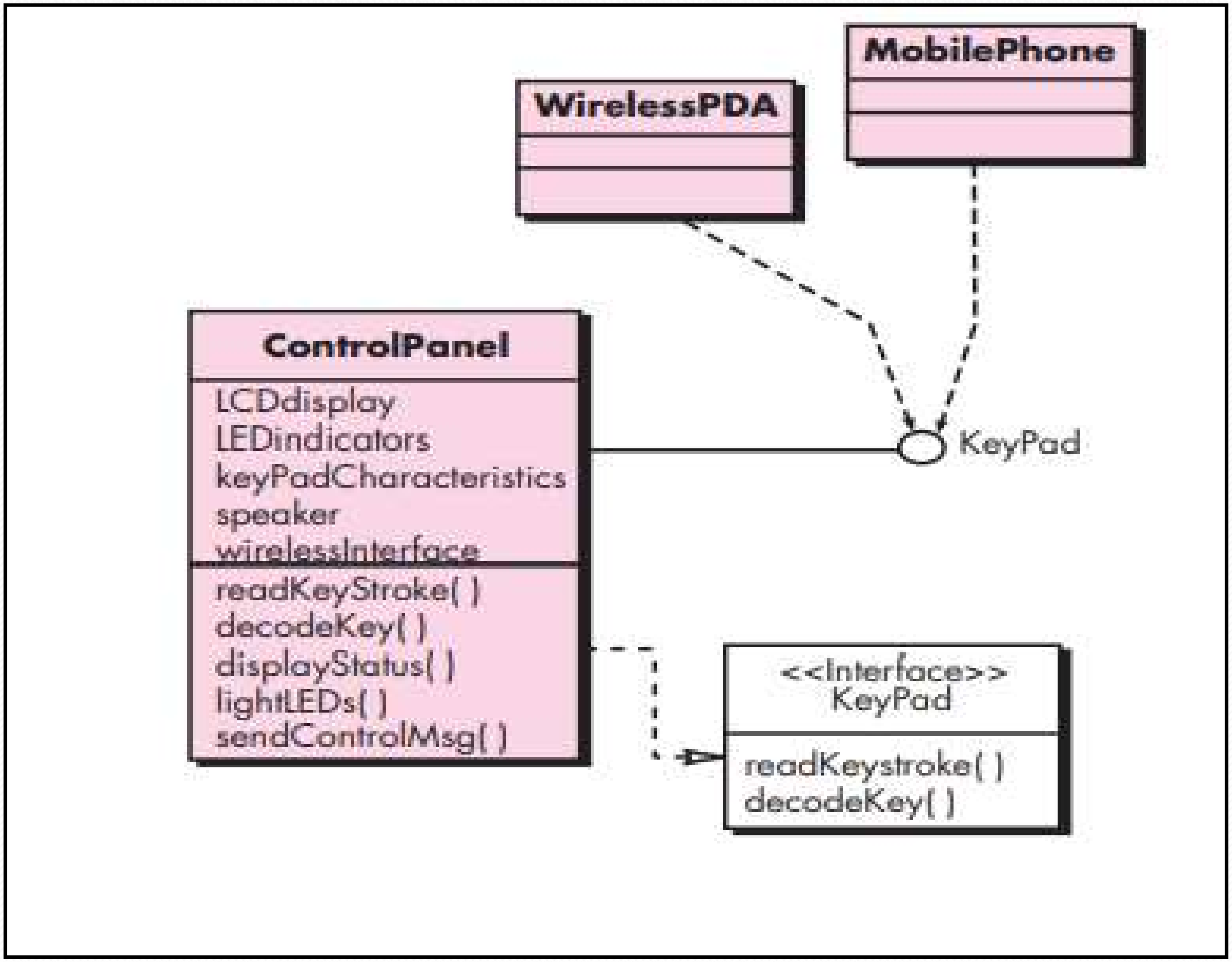
3. Interface design elements

- Represents the information flow within it and out of the system.
- They communicate between the components defined as part of architecture.

Following are the important elements of the interface design:

1. The user interface
2. The external interface to the other systems, networks etc.
3. The internal interface between various components.

Interface



WirelessPDA
MobilePhone
ControlPanel
LCDdisplay
LEDindicators
keyPadCharacteristics
speaker
wirelessInterface
readKeyStroke( )
decodeKey( )
displayStatus( )
lightLEDs( )
sendControlMsg( )
KeyPad
<<Interface>>
KeyPad
readKeystroke( )
decodeKey( )

4. Component level diagram elements

- The component level design for software is similar to the set of detailed specification of each room in a house.
- Describes the internal details of the each software component.
- The processing of data structure occurs in a component and an interface which allows all the component operations.
- In a context of object-oriented software engineering, a component shown in a UML diagram.
- The UML diagram is used to represent the processing logic.

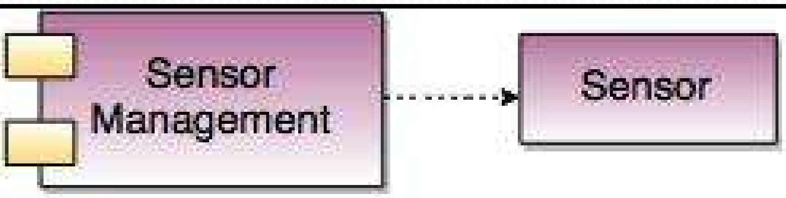
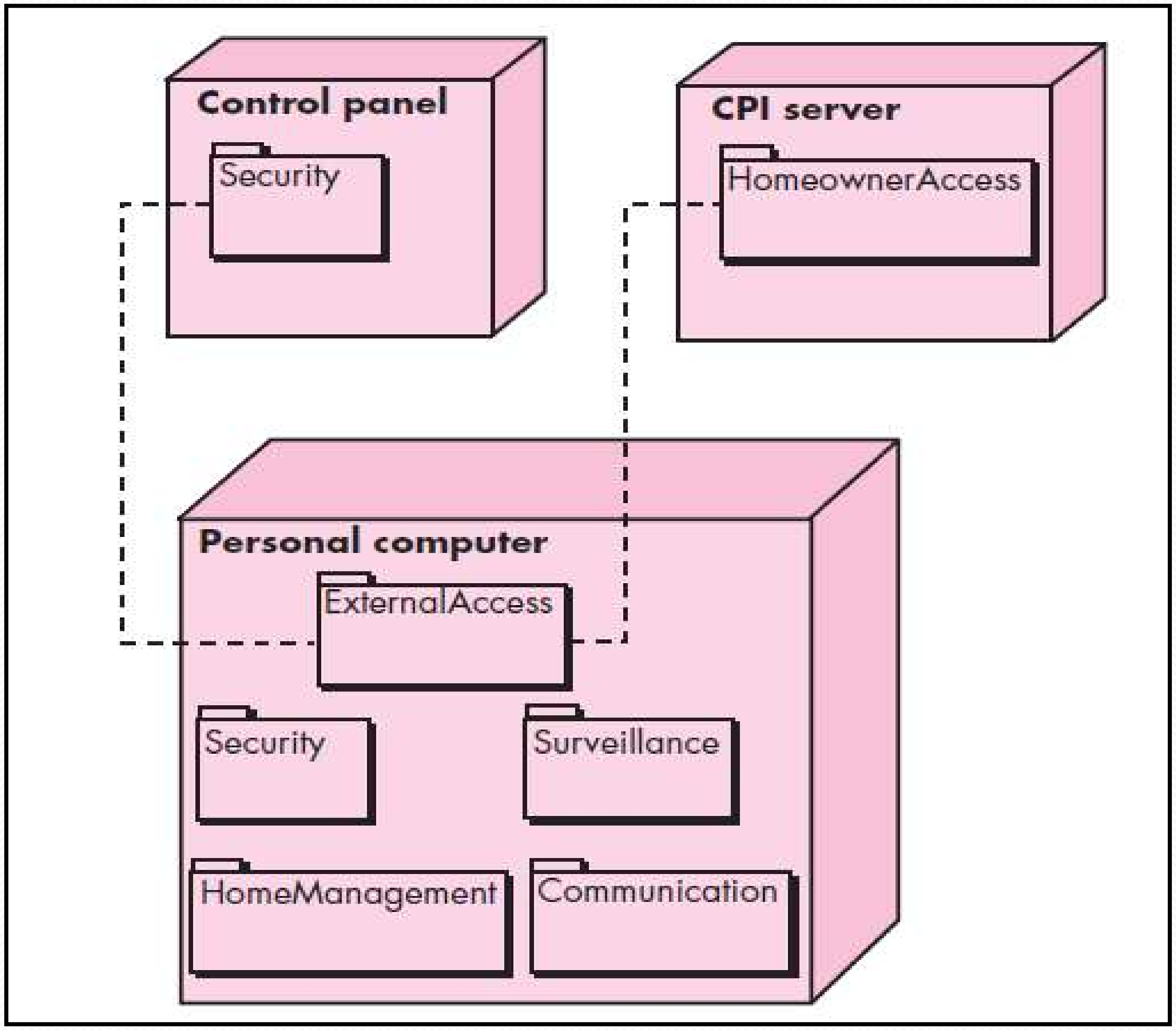


Fig. - UML component diagram for sensor managemnet

5. Deployment level design elements:

- Indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.
- For example, the elements of the *SafeHome product are configured* to operate within three primary computing environments—a home-based PC, the *SafeHome control panel, and a server housed at CPI Corp. (providing Internet-based* access to the system).

A UML deployment diagram



Code Review Analysis

- Code review (sometimes referred to as peer review) is a software quality assurance activity in which one or several humans check a program mainly by viewing and reading parts of its source code, and they do so after implementation or as an interruption of implementation.
- At least one of the humans must not be the code's author.
- The humans performing the checking, excluding the author, are called "reviewers".

Although direct discovery of quality problems is often the main goal, code reviews are usually performed to reach a combination of goals :

- **Better code quality** – improve internal code quality and maintainability (readability, uniformity, understandability, …)
- **Finding defects** – improve quality regarding external aspects, especially correctness, but also find performance problems, security vulnerabilities, injected malware, …
- **Learning/Knowledge transfer** – help in transferring knowledge about the codebase, solution approaches, expectations regarding quality, etc; both to the reviewers as well as to the author
- **Increase sense of mutual responsibility** – increase a sense of collective code ownership and solidarity
- **Finding better solutions** – generate ideas for new and better solutions and ideas that transcend the specific code at hand.
- **Complying to QA guidelines** – Code reviews are mandatory in some contexts, e.g., air traffic software

THANK YOU