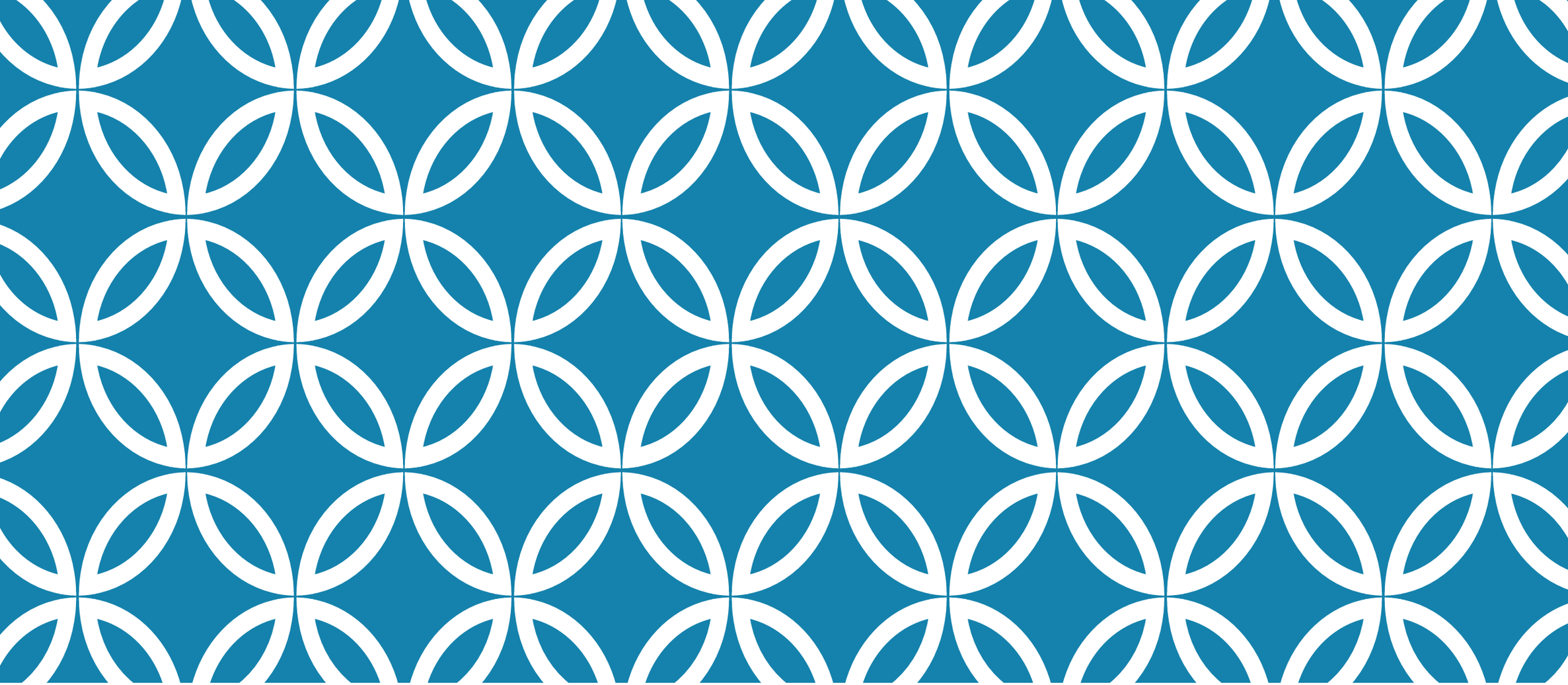


CHAPTER SIX

System Design using UML

INTRODUCTION TO OBJECT ORIENTED DESIGN

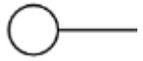
- Analysis results in the requirements model described by the following products:
  - a set of nonfunctional requirements and constraints
  - a use case model, describing the system functionality from the actors' point of view
  - an object model, describing the entities manipulated by the system
  - a sequence diagram for each use case, showing the sequence of interactions among objects participating in the use case.

- The analysis model describes the system completely from the actors' point of view.
- It serves as the basis of communication between the client and the developers.
- The analysis model, however, does not contain information about the internal structure of the system, its hardware configuration, or more generally, how the system should be realized.
- System design is the first step in this direction.

- System design results in the following products:
  - **design goals**, describing the qualities of the system that developers should optimize. They are derived from the nonfunctional requirements.
  - **software architecture**, describing the subsystem decomposition in terms of subsystem responsibilities, dependencies among subsystems, subsystem mapping to hardware, and major policy decisions such as control flow, access control, and data storage
  - **boundary use cases**, describing the system configuration, startup, shutdown, and exception handling issues.

SYSTEM DESIGN CONCEPTS

- A **subsystem is a** replaceable part of the system with well-defined interfaces that encapsulates the state and behavior of its contained classes.
- A subsystem is characterized by the services it provides to other subsystems.
- A **service is a set** of related operations that share a common purpose.
  - Example: A subsystem providing a notification service defines operations to send notices, look up notification channels, and subscribe and unsubscribe to a channel.
- The set of operations of a subsystem that are available to other subsystems form the **subsystem interface.**

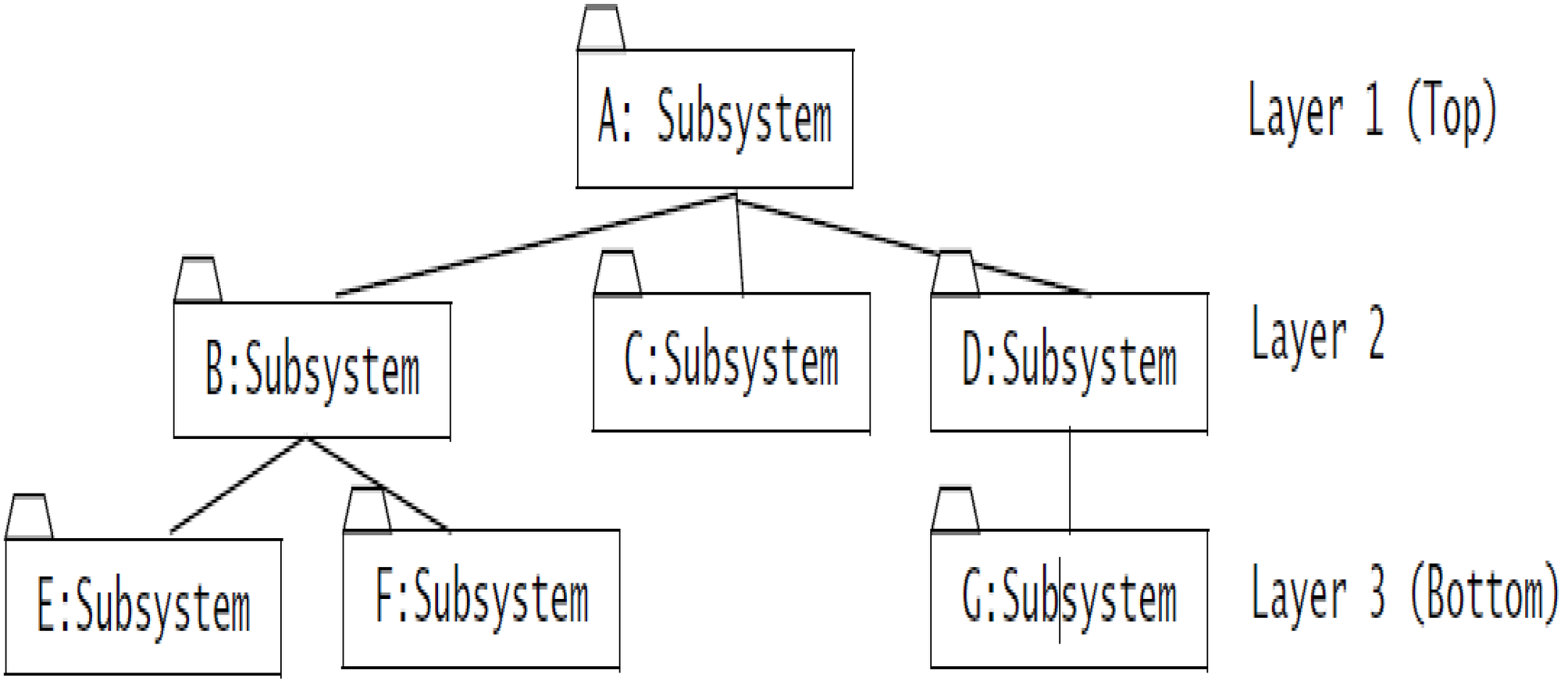
- Provided and required interfaces can be depicted in UML with **assembly connectors,** also called **ball-and-socket connectors.**
- The **provided interface** is shown as **a ball icon (also called lollipop)** with its name next to it. 
- A **required interface** is shown as **a socket icon.** 
- The dependency between two subsystems is shown by connecting the corresponding ball and socket in the component diagram.

COUPLING AND COHESION

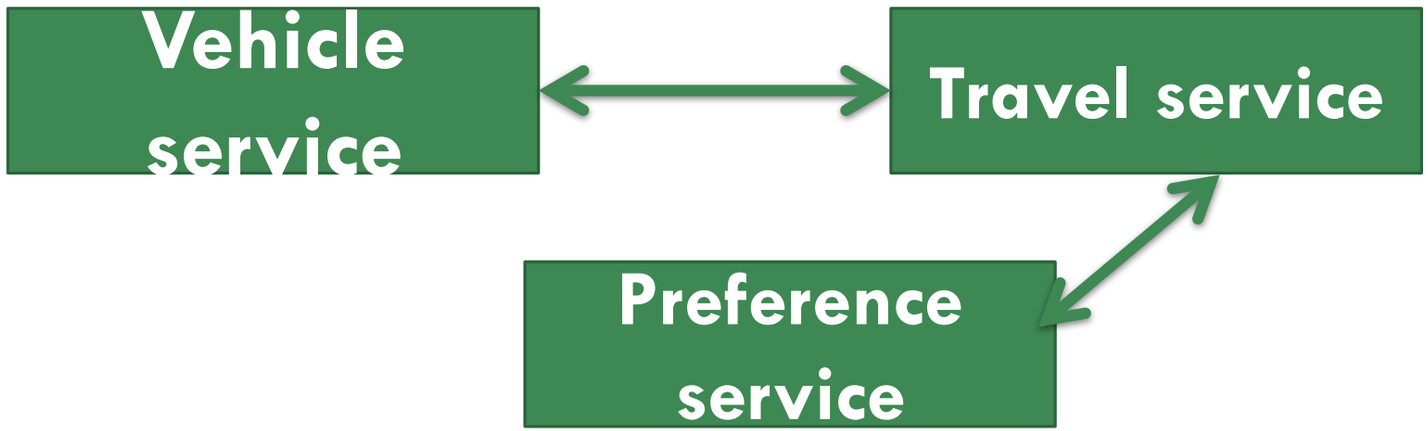
- **Coupling** measures the dependencies between two subsystems,
- **cohesion** measures the dependencies among classes within a subsystem.
- Ideal subsystem decomposition should minimize coupling and maximize cohesion.

LAYERS AND PARTITIONS

- A hierarchical decomposition of a system yields an ordered set of layers.
- A layer is a grouping of subsystems providing related services, possibly realized using services from another layer.
- Layers are ordered in that each layer can depend only on lower level layers and has no knowledge of the layers above it.
- The layer that does not depend on any other layer is called the **bottom layer**, and the layer that is not used by any other is called the **top layer**.
- In a **closed architecture**, each layer can access only the layer immediately below it.
  - Example: Reference Model of Open Systems Interconnection (the OSI model)
- In an **open architecture**, a layer can also access layers at deeper levels.


A: Subsystem
B:Subsystem
C:Subsystem
D:Subsystem
E:Subsystem
F:Subsystem
G:Subsystem
Layer 1 (Top)
Layer 2
Layer 3 (Bottom)

- Another approach to dealing with complexity is to **partition** the system into peer subsystems, each responsible for a different class of services.
  - **For example,** an onboard system for a car could be decomposed into a travel service that provides real-time directions to the driver, an individual preferences service that remembers a driver's seat position and favorite radio station, and vehicle service that tracks the car's gas consumption, repairs, and scheduled maintenance.
  - Each subsystem depends loosely on the others, but can often operate in isolation.



- In general, a subsystem decomposition is the result of both partitioning and layering.
- We first partition the system into top-level subsystems, which are responsible for specific functionality or run on a specific hardware node.
- Each of the resulting subsystems are, if complexity justifies it, decomposed into lower- and lower-level layers until they are simple enough to be implemented by a single developer.
- Each subsystem adds a certain processing overhead because of its interface with other systems.
- Excessive partitioning or layering can increase complexity.

OBJECT-ORIENTED DESIGN WITH UML
A) COMPONENT DIAGRAMS

- A component diagram shows the relationship between different components in a system.
- A component diagram in UML gives a bird's-eye view of your software system.
- The term "component" refers to a module of classes that represent independent systems or subsystems with the ability to interface with the rest of the system.
- Component diagrams can describe software systems that are implemented in any programming language or style.

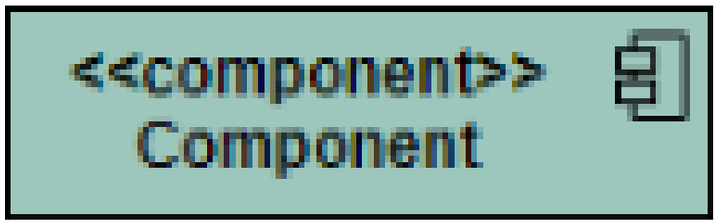
BENEFITS OF COMPONENT DIAGRAMS

- Imagine the system's physical structure.
- Pay attention to the system's components and how they relate.
- Emphasize the service behavior as it relates to the interface

COMPONENT DIAGRAM NOTATION

1. Component

A component is a logical unit block of the system, a slightly higher abstraction than classes. It is represented as a rectangle with a smaller rectangle in the upper right corner with tabs or the word written above the name of the component to help distinguish it from a class.

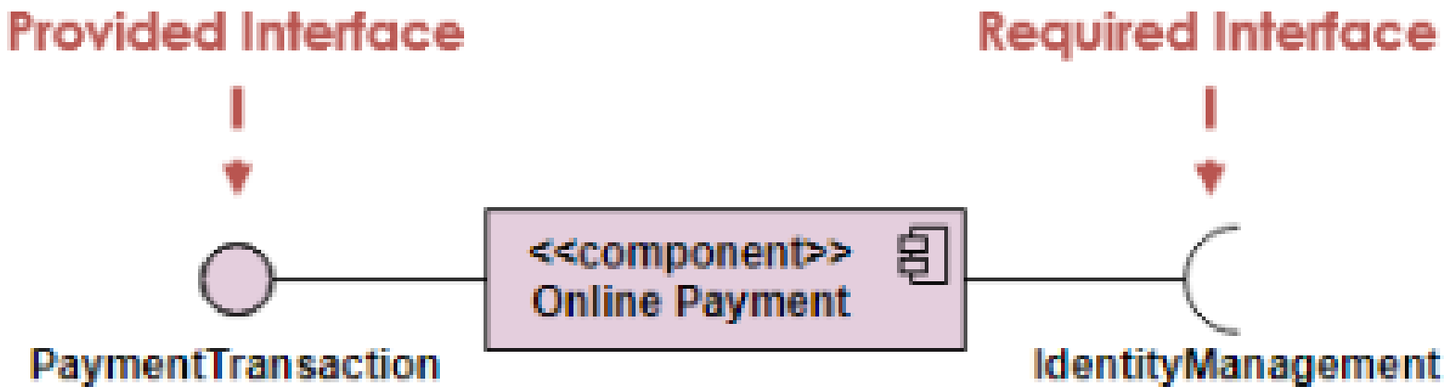


2. Interface

An interface (small circle or semi-circle on a stick) describes a group of operations used (required) or created (provided) by components.

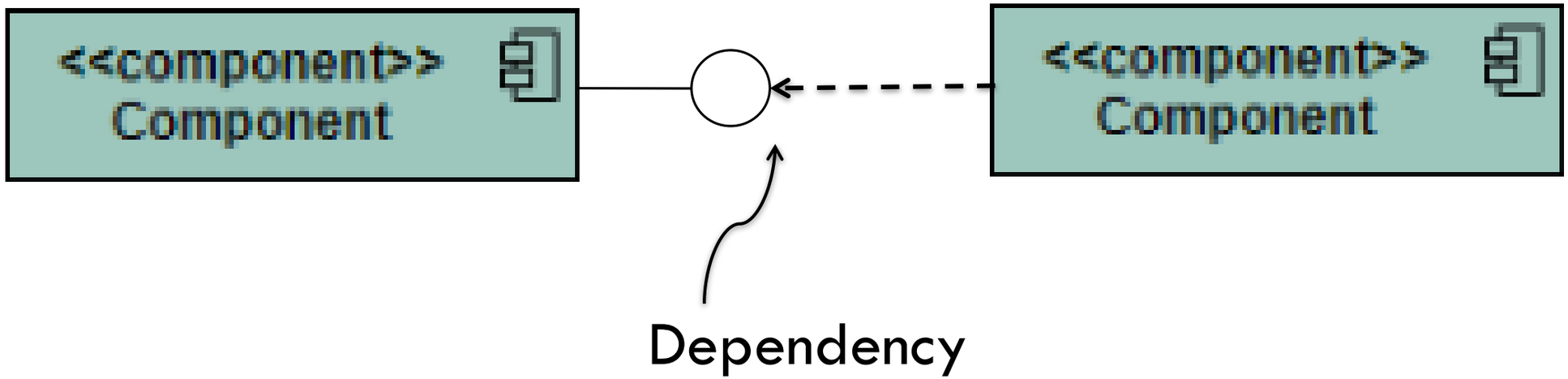
A full circle represents an interface created or provided by the component.

A semi-circle represents a required interface, like a person's input.



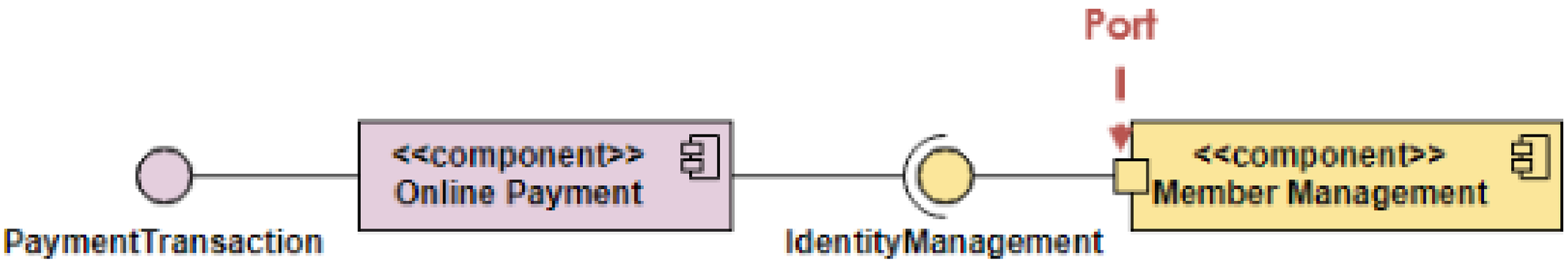
3. Dependencies

Draw dependencies among components using dashed arrows.

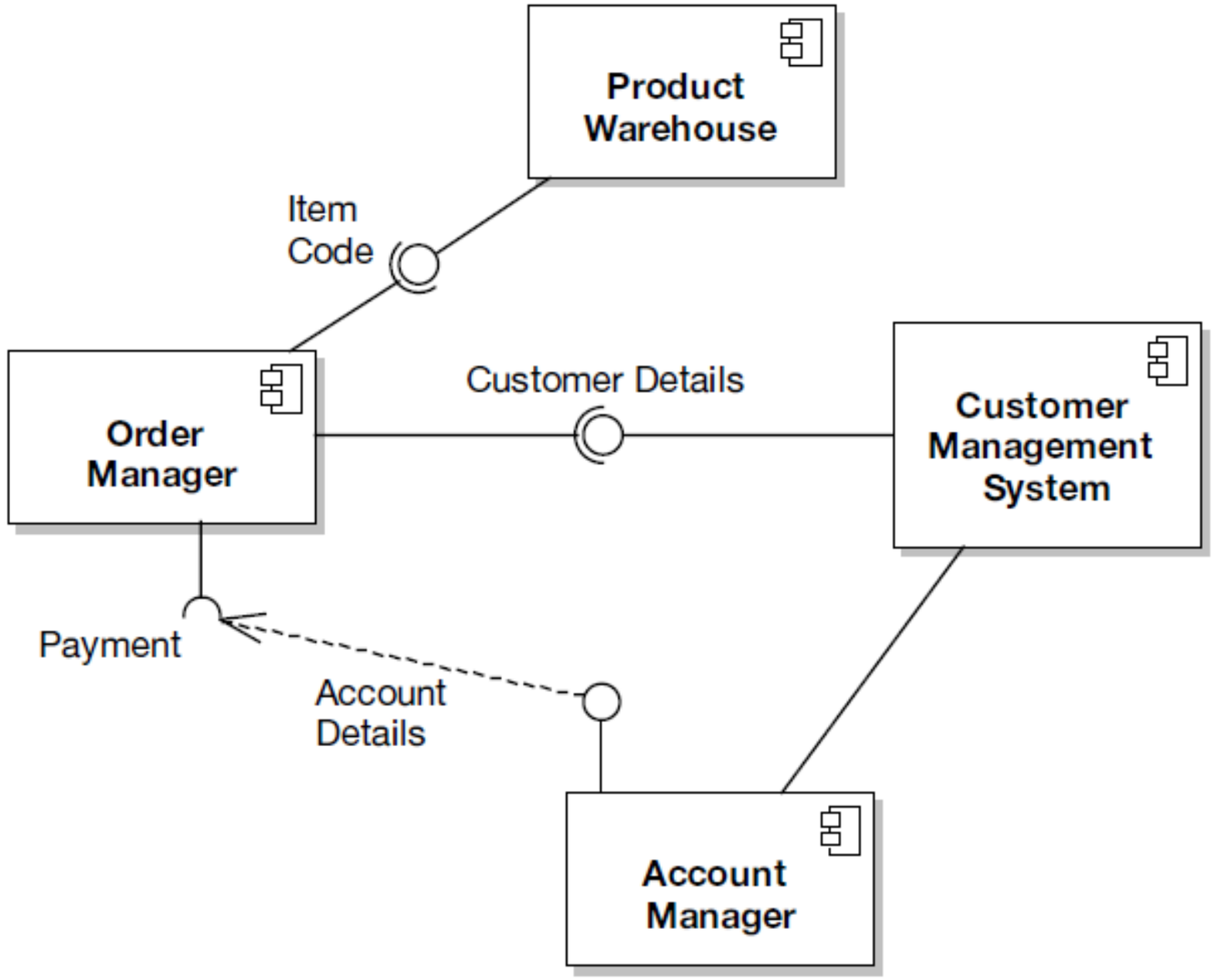


4. Port

A port (definition) indicates that the component itself does not provide the required interfaces (e.g., required or provided). Instead, the component delegates the interface(s) to an internal class.



ORDER MANAGEMENT SYSTEM EXAMPLE



COLLABORATION DIAGRAM

- Collaboration diagrams (**known as Communication Diagram in UML 2.x**) are used to show how objects interact to perform the behavior of a particular use case, or a part of a use case.
  - **OR** it shows interactions between objects and/or **parts** (represented as **lifelines**) using sequenced messages in a free-form arrangement.
- Along with sequence diagrams, collaboration are used by designers to define and clarify the roles of the objects that perform a particular flow of events of a use case.
- They are the primary source of information used to determining class responsibilities and interfaces.

COMMUNICATION DIAGRAM VS. SEQUENCE DIAGRAM

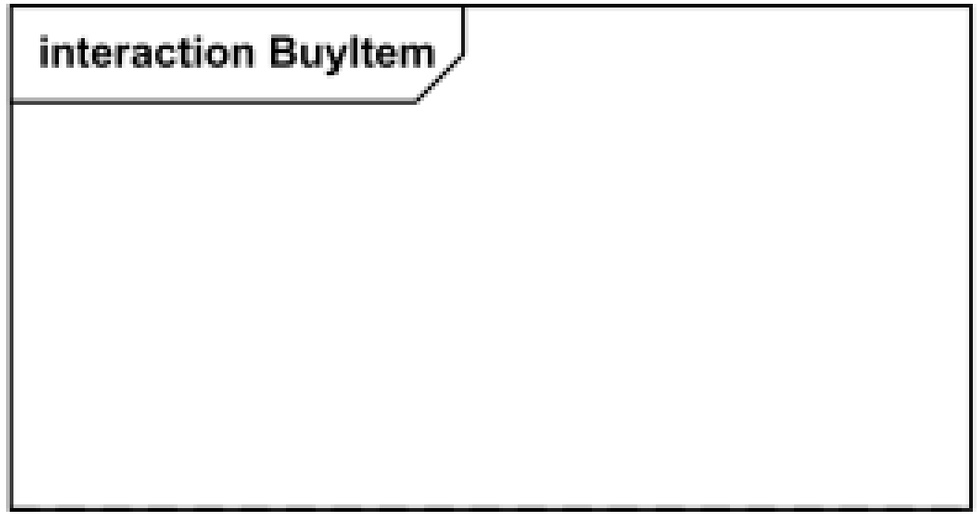
- The communication diagram and the sequence diagram are semantically equivalent, that is, they present the same information, and you can turn a communication to a sequence diagram and vice versa.
- The main distinction between them is that the communication diagram arranged elements according to space, the sequence diagram is according to time.

NOTATIONS

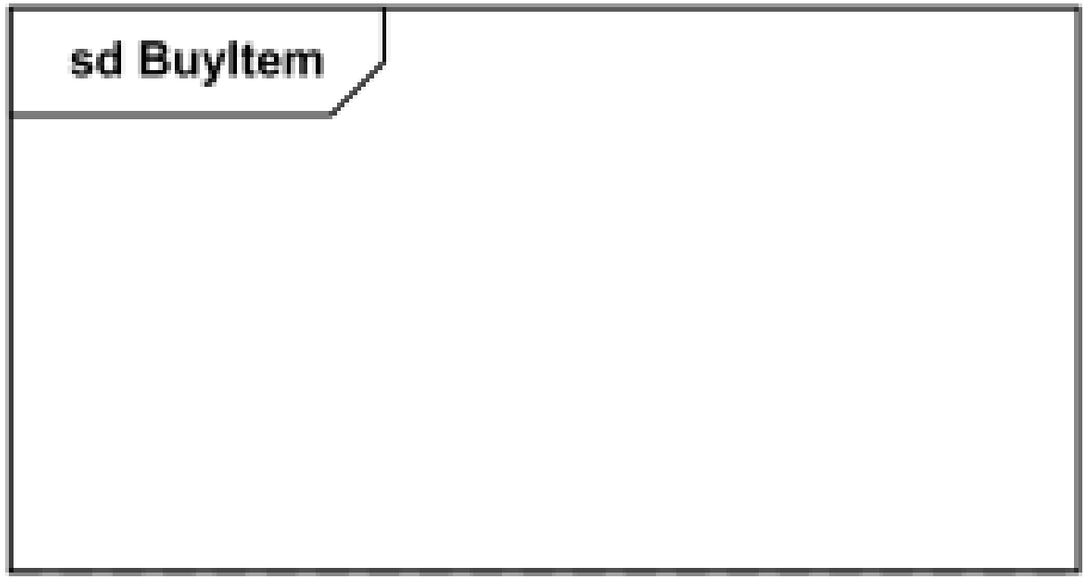
- Frame

Communication diagrams could be shown within a rectangular **frame** with the **name** in a compartment in the upper left corner.

There is no specific long form name for communication diagrams heading types. The long form name **interaction** (used for **interaction diagrams** in general) could be used.



- There is also no specific short form name for **communication diagrams**. Short form name **sd** (which is used for **interaction diagrams** in general) could be used.
  - ✓This **sd** is bit confusing as it looks like abbreviation of **s**equence **d**iagram.



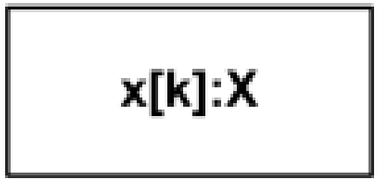
LIFELINE

- **Lifeline** is a specialization of **named element** which represents an **individual participant** in the interaction.
- While parts and structural features may have multiplicity greater than 1, lifelines represent **only one** interacting entity.
- If the referenced connectable element is multi-valued (i.e, has a multiplicity > 1), then the lifeline may have an expression (**selector**) that specifies which particular part is represented by this lifeline.
- If the selector is omitted, this means that an **arbitrary representative** of the multi-valued connectable element is chosen.
- A **Lifeline** is shown as a rectangle (corresponding to the "head" in sequence diagrams).
- Lifeline in sequence diagrams does have "tail" representing the **line of life** whereas "lifeline" in **communication diagram** has no line, just "head".

- Information identifying the lifeline is displayed inside the rectangle in the following format:

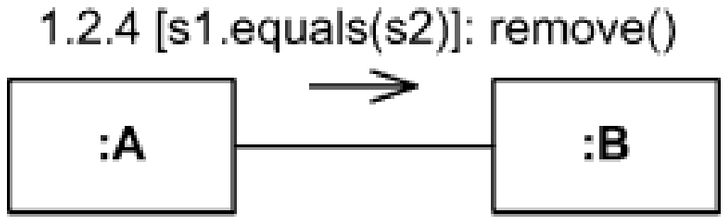
Anonymous lifeline of class User.

***Lifeline** "data" of class Stock*



MESSAGE

- **Message** in **communication diagram** is shown as a line with **sequence expression** and **arrow** above the line.
- The arrow indicates direction of the communication.



Instance of class A sends remove() message to instance of B if s1 is equal to s2

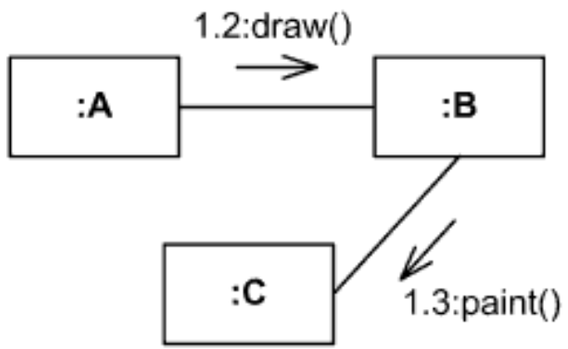
SEQUENCE EXPRESSION

- The **sequence expression** is a dot separated list of **sequence terms** followed by a colon (":") and message name after that:
- ***sequence-expression*** ::= ***sequence-term*** '.' . . . ':' ***message-name***
- For example,

  3b.2.2:m5

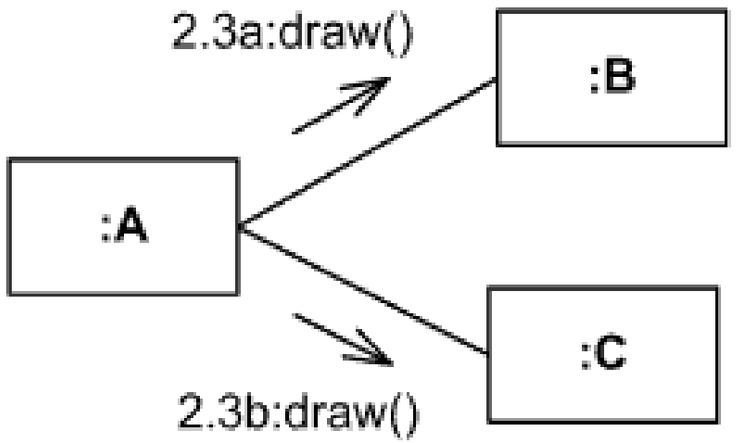
contains sequence expression **3b.2.2** and message name **m5**.

- The **integer** represents the **sequential order** of the message within the next higher level of procedural calling (activation).
- Messages that differ in one integer term are sequential at that level of nesting.
- For example,
  - ✓message with sequence 2 follows message with sequence 1,
  - ✓2.1 follows 2
  - ✓5.3 follows 5.2 within activation 5
  - ✓1.2.4 follows message 1.2.3 within activation 1.2.



Instance of A sends draw() message to instance of B, and after that B sends paint() to C

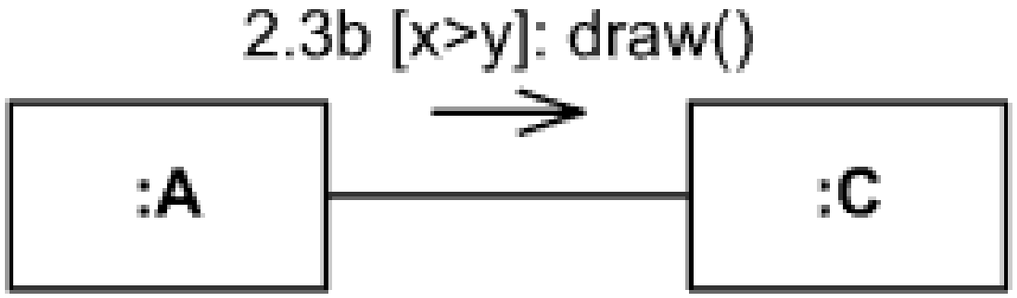
- The **name** represents a **concurrent thread** of control. Messages that differ in the final name are concurrent at that level of nesting.
- For example,
  - ✓ messages 2.3a and 2.3b are concurrent within activation 2.3,
  - ✓ 1.1 follows 1a and 1b,
  - ✓ 3a.2.1 and 3b.2.1 follow 3.2.



Instance of A sends draw() messages concurrently to instance of B and to instance of C

- A **guard** specifies condition for the message to be sent (executed) at the given nesting depth. UML does not specify guard syntax, so it could be expressed in pseudocode, some programming language, or something else.

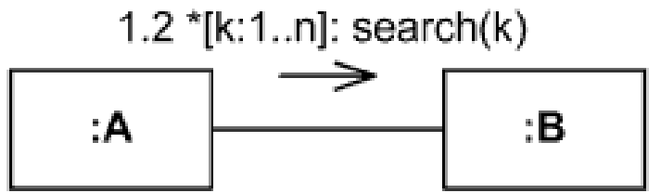
- For example,
  - ✓ **2.3b [x>y]: draw()** - message draw() will be executed if x is greater than y,
  - ✓ **1.1.1 [s1.equals(s2)]: remove()** - message remove() will be executed if s1 equals s2.



Instance of class A will send message draw() to the instance of C, if x > y

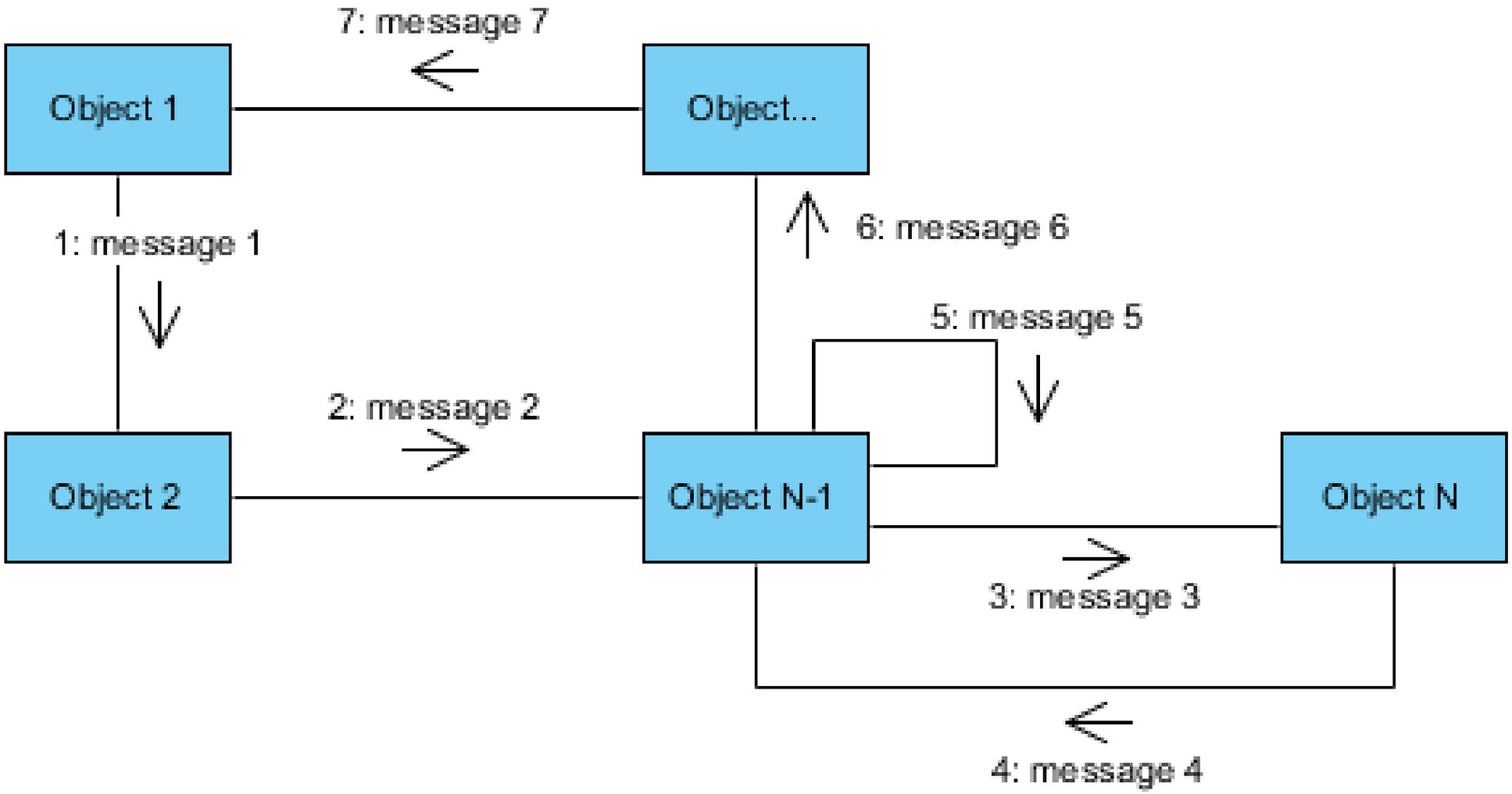
- An **iteration** specifies a sequence of messages at the given nesting depth.
- UML does not specify **iteration-clause** syntax, so it could be expressed in pseudocode, some programming language, or something else.
- Iteration clause may be omitted, in which case the iteration conditions are unspecified.
- The * iteration notation specifies that the messages in the iteration will be executed **sequentially**.
- The *|| (star followed by a double vertical line) iteration notation specifies **concurrent** (parallel) execution of messages.

- For example,
  - ✓ **4.2c** *[i=1..12]: **search(t[i])** - search() will be executed 12 times, one after another
  - ✓ **4.2c** *||[i=1..12]: **search(t[i])** - 12 search() messages will be sent concurrently,
  - ✓ **2.2** *: **notify()** - message notify() will be repeated some unspecified number of times.

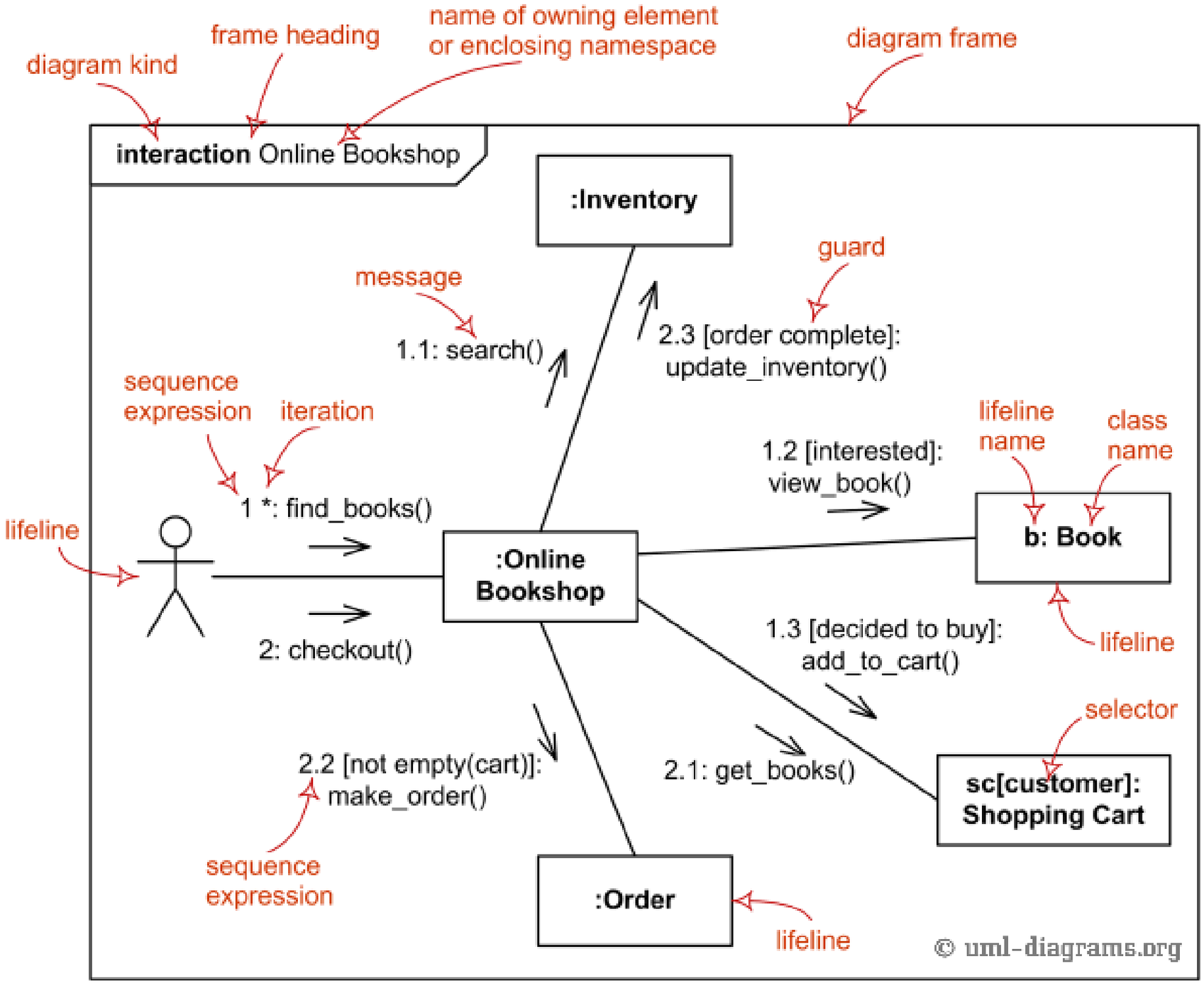


Instance of class A will send search() message to instance of B n times, one by one

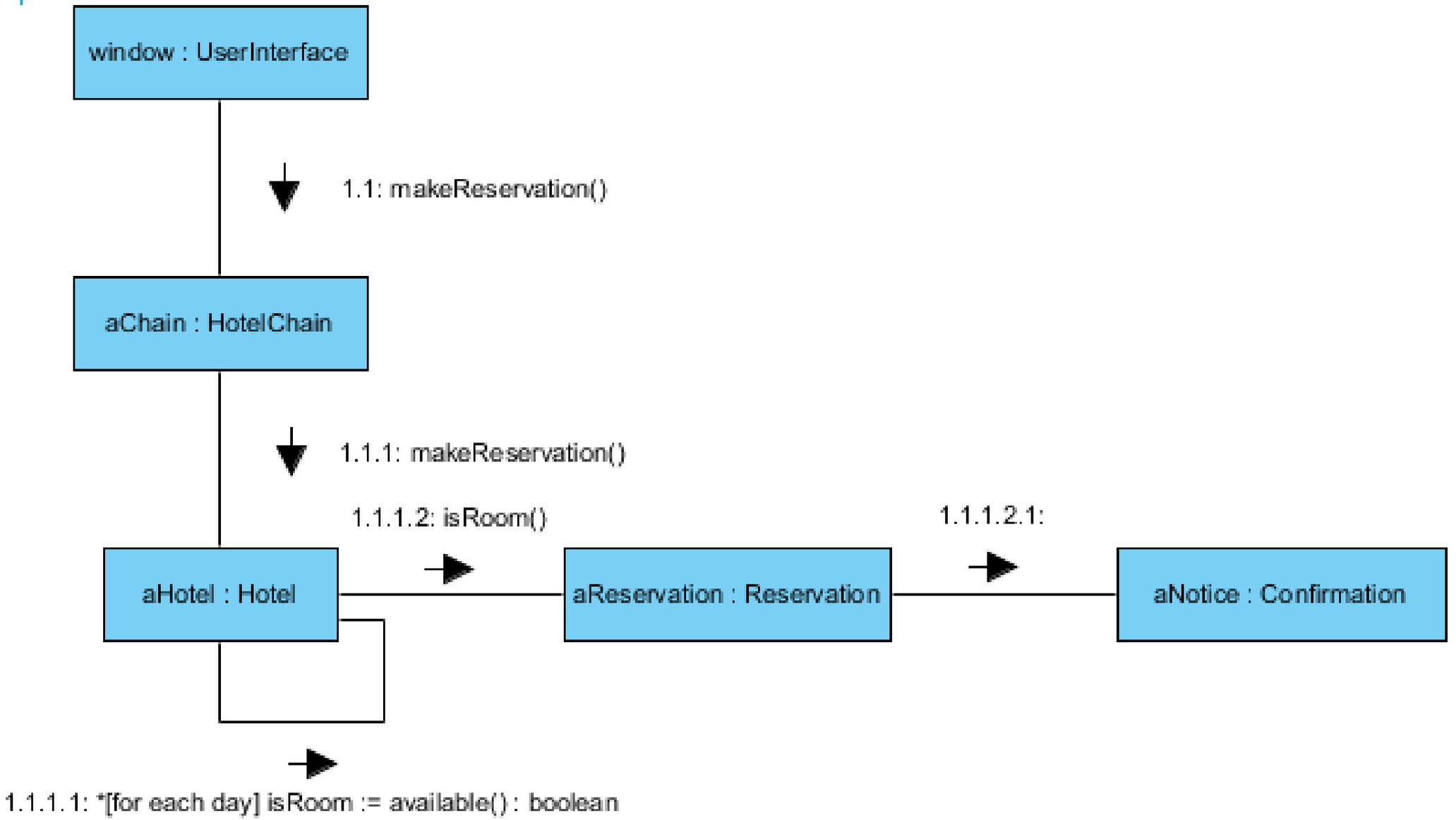
Instance of class A will send n concurrent search() messages to instance of B


7: message 7
Object 1
Object...
1: message 1
6: message 6
5: message 5
2: message 2
Object 2
Object N-1
Object N
3: message 3
4: message 4

EXAMPLE – ONLINE BOOKSHOP



EXAMPLE - HOTEL RESERVATION

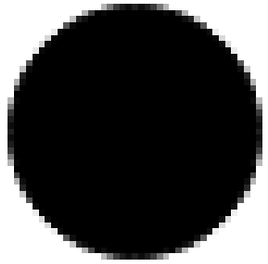


ACTIVITY DIAGRAM

- **Activity Diagrams** and advanced flowchart to illustrate the flow of control in a system and refer to the steps involved in the execution of a use case.
- We model sequential and concurrent activities using activity diagrams.
- So, we basically depict workflows visually using an activity diagram.
- An activity diagram focuses on condition of flow and the sequence in which it happens.

NOTATIONS

- **Initial State –** The starting state before an activity takes place is depicted using the initial state.

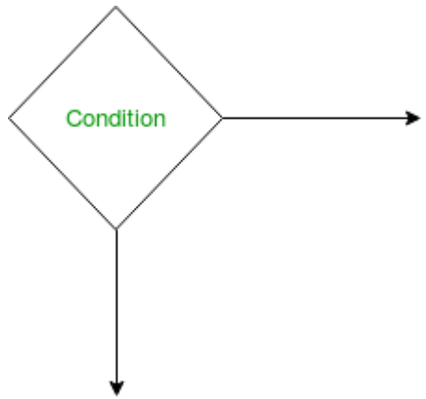


- A process can have only one initial state unless we are depicting nested activities.
- We use a black filled circle to depict the initial state of a system.

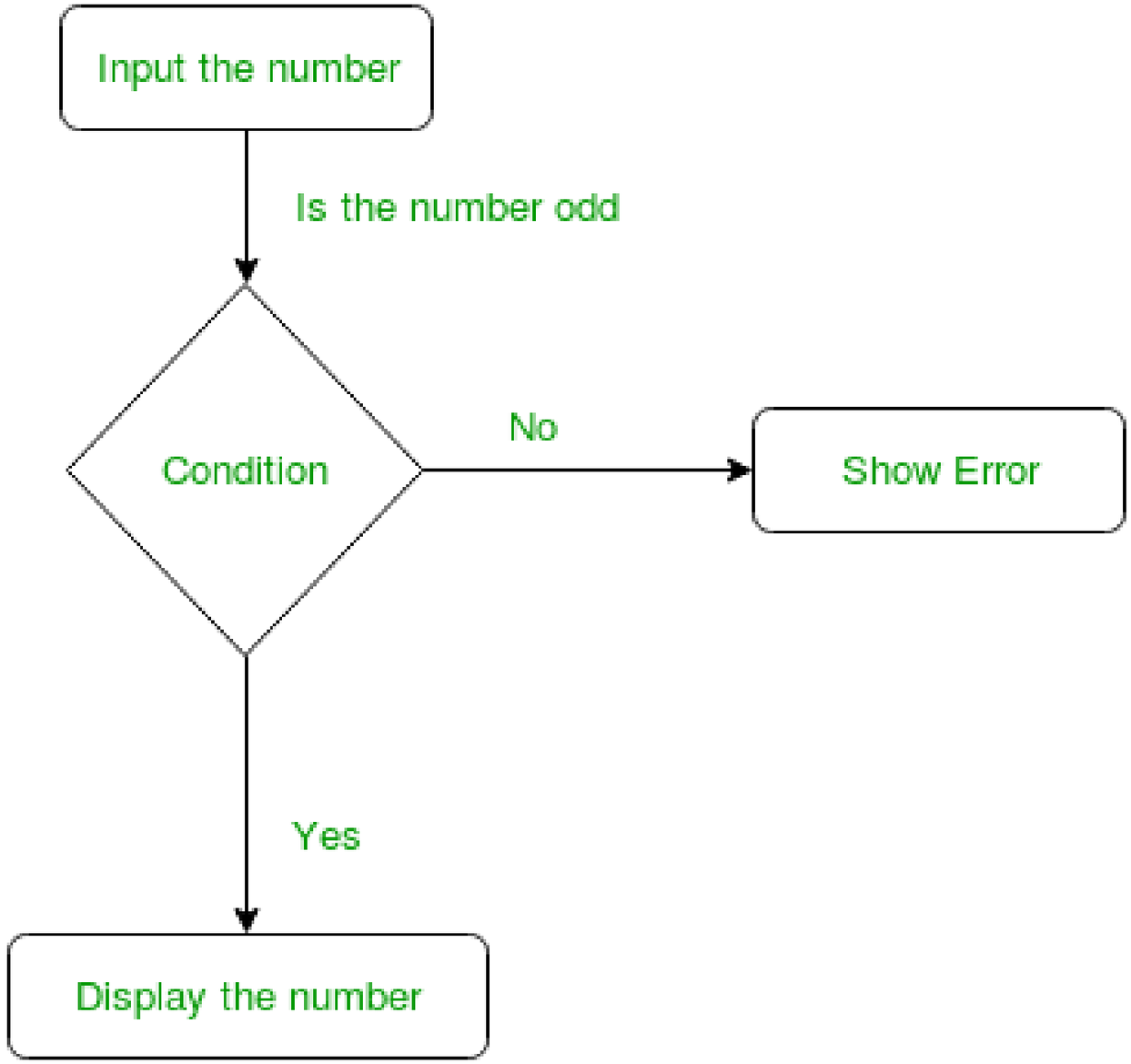
- **Action or Activity State –** An activity represents execution of an action on objects or by objects.
- We represent an activity using a rectangle with rounded corners.
- Basically any action or event that takes place is represented using an activity.

- **Action Flow or Control flows –** Action flows or Control flows are also referred to as paths and edges.
- They are used to show the transition from one activity state to another.
- An activity state can have multiple incoming and outgoing action flows.
- We use a line with an arrow head to depict a Control Flow.
- If there is a constraint to be adhered to while making the transition it is mentioned on the arrow

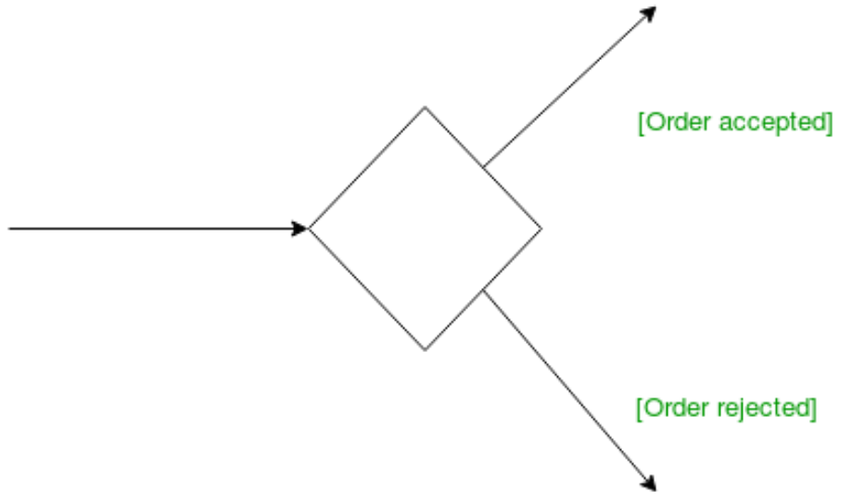
- **Decision node and Branching –** When we need to make a decision before deciding the flow of control, we use the decision node.



- The outgoing arrows from the decision node can be labeled with conditions or guard expressions.
  It always includes two or more output arrows.

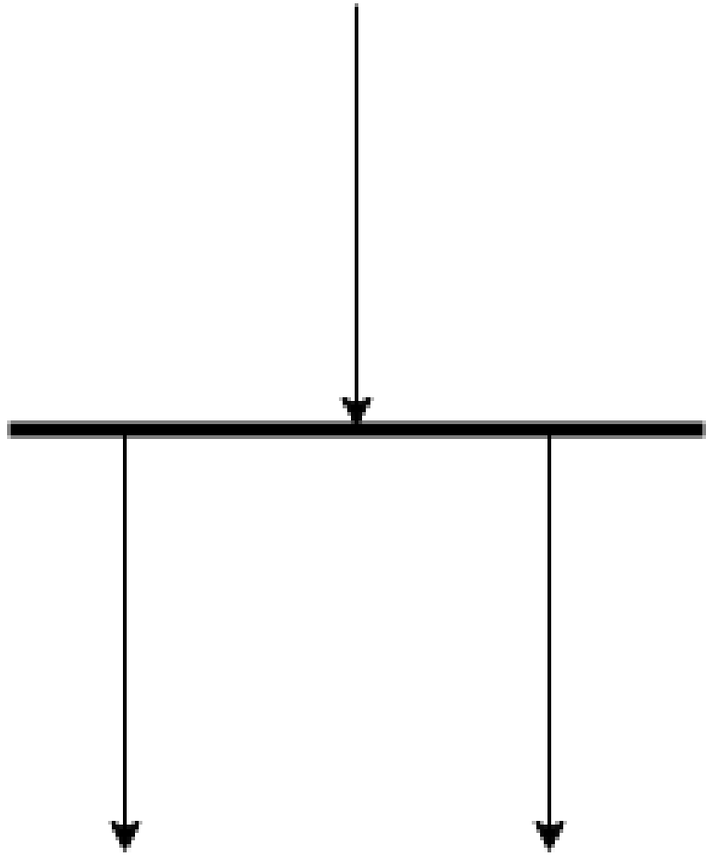

Input the number
Is the number odd
Condition
No
Show Error
Yes
Display the number

- **Guards –** A Guard refers to a statement written next to a decision node on an arrow sometimes within square brackets.

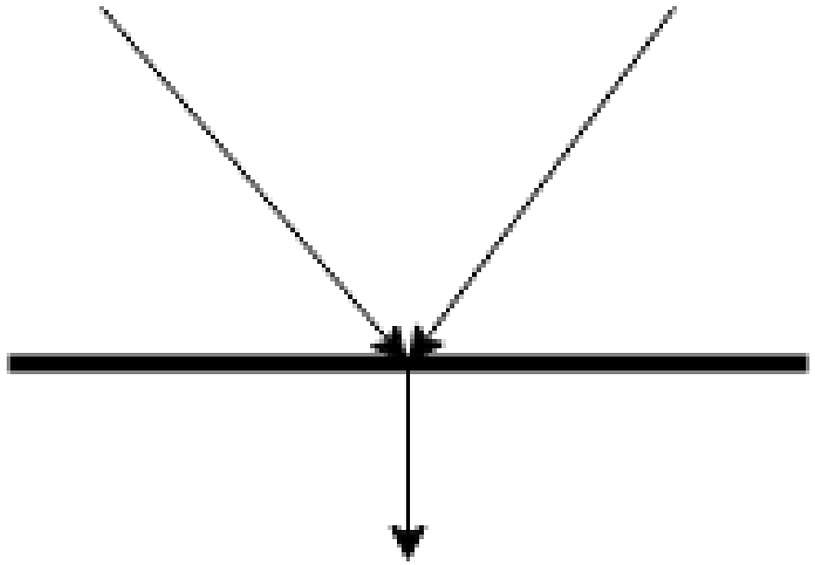


- The statement must be true for the control to shift along a particular direction.
- Guards help us know the constraints and conditions which determine the flow of a process.

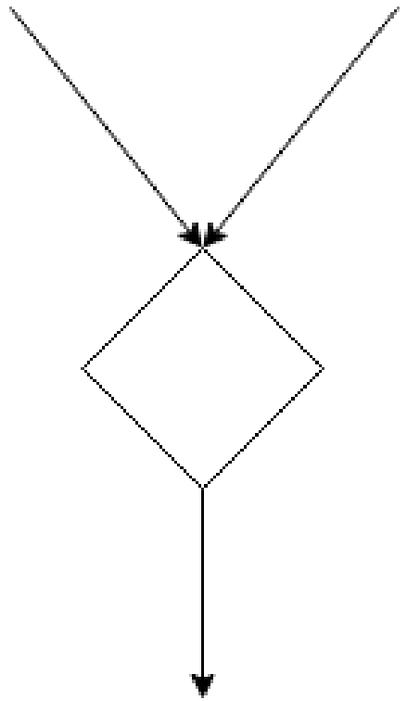
- **Fork –** Fork nodes are used to support concurrent activities.



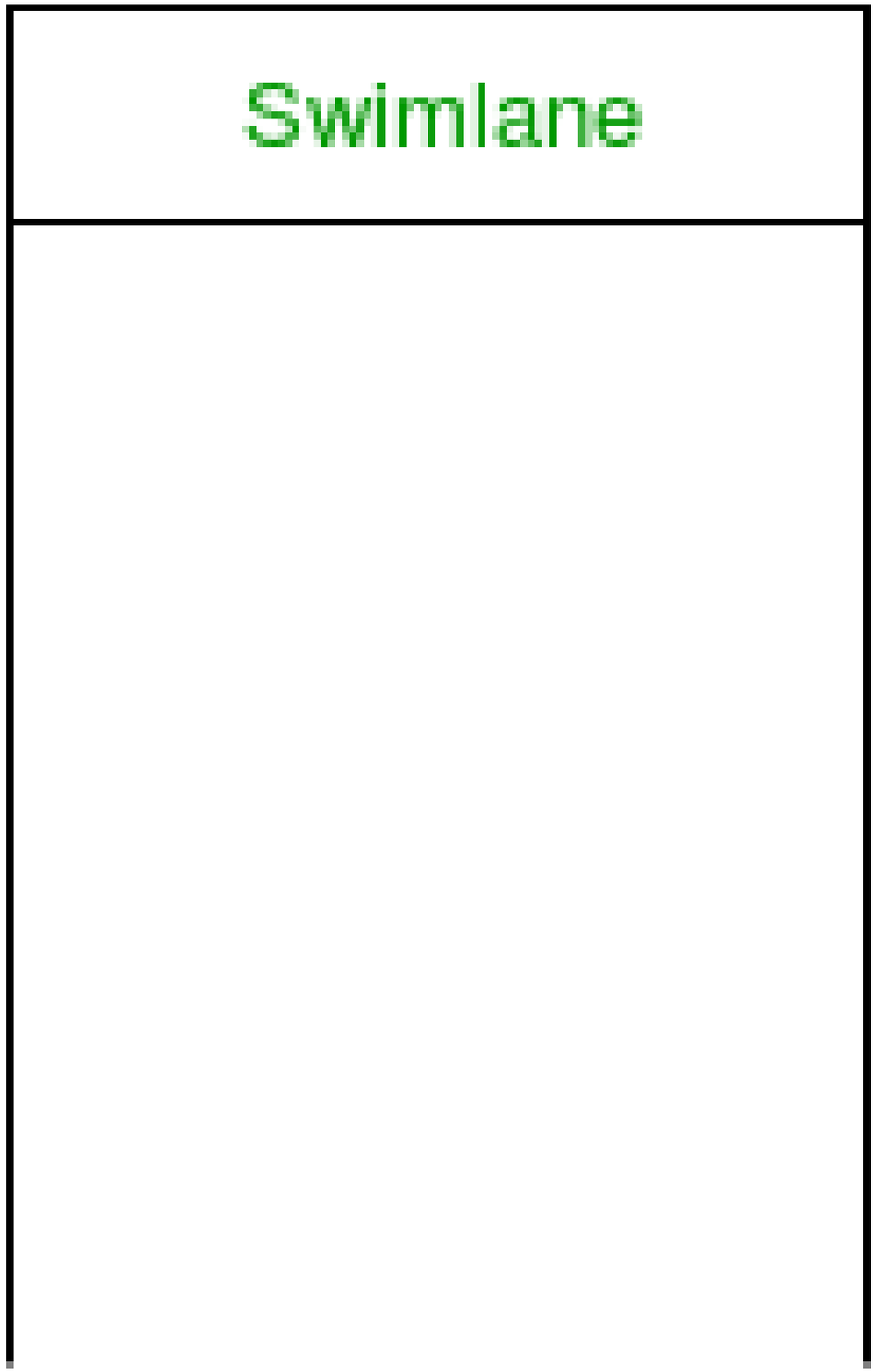
- **Join –** Join nodes are used to support concurrent activities converging into one. For join notations we have two or more incoming edges and one outgoing edge



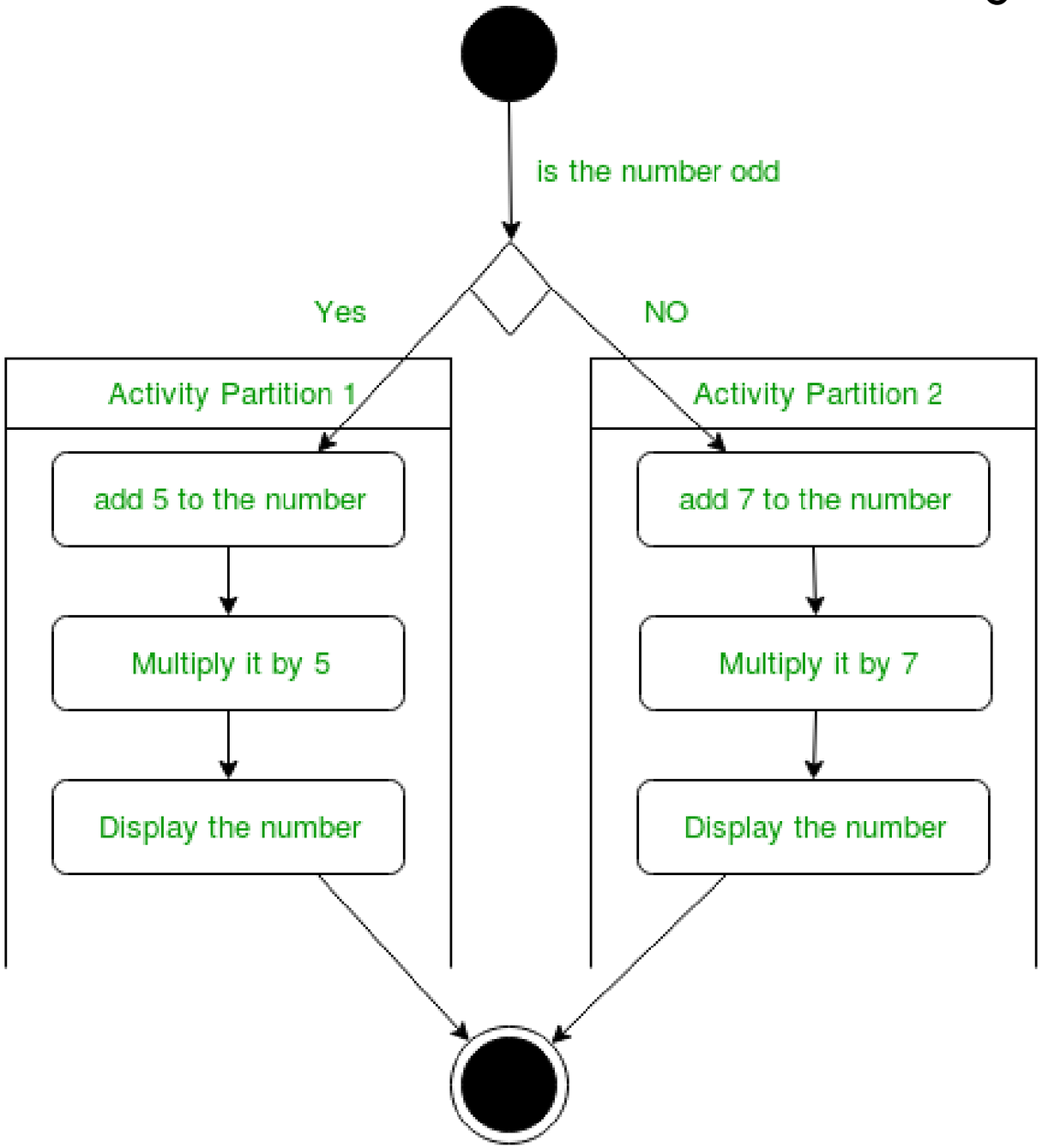
- **Merge or Merge Event –** Scenarios arise when activities which are not being executed concurrently have to be merged.
- We use the merge notation for such scenarios.
- We can merge two or more activities into one if the control proceeds onto the next activity irrespective of the path chosen.



- **Swimlanes –** We use swimlanes for grouping related activities in one column.
- Swimlanes group related activities into one column or one row. Swimlanes can be vertical and horizontal.
- Swimlanes are used to add modularity to the activity diagram. It is not mandatory to use swimlanes.
- They usually give more clarity to the activity diagram. It's similar to creating a function in a program.
- It's not mandatory to do so, but, it is a recommended practice.


Swimlane

For example – Here different set of activities are executed based on if the number is odd or even. These activities are grouped into a swimlane.

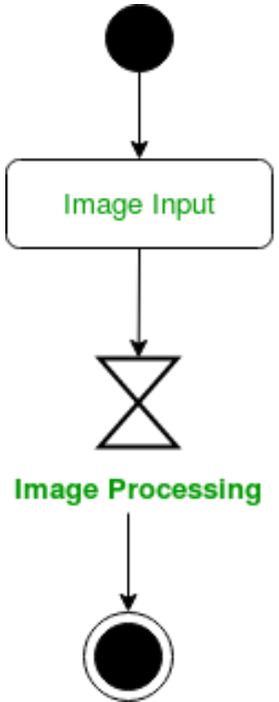


- **Time Event –** We can have a scenario where an event takes some time to complete.

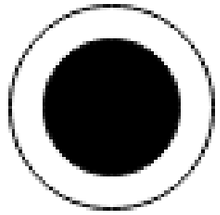
Time Event

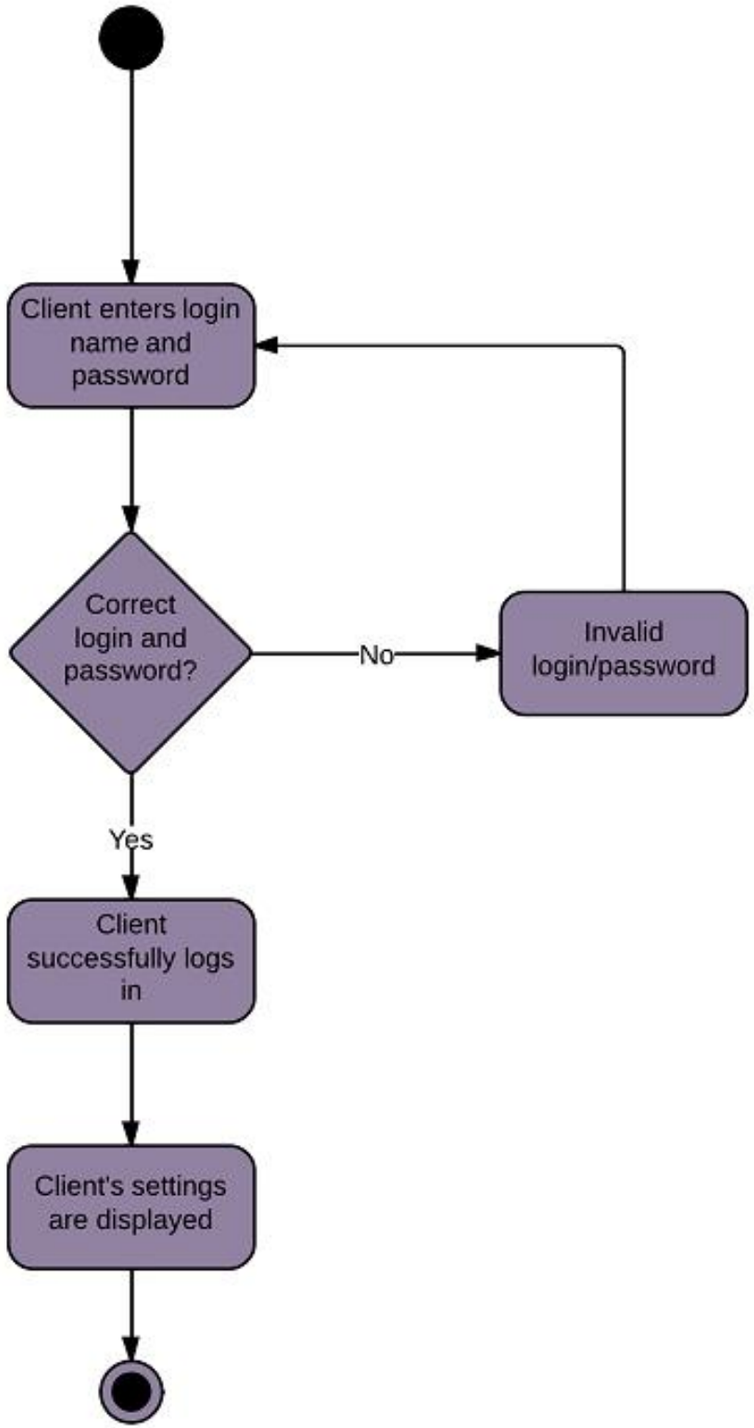
- We use an hourglass to represent a time event.

For example – Let us assume that the processing of an image takes takes a lot of time. Then it can be represented as shown below.

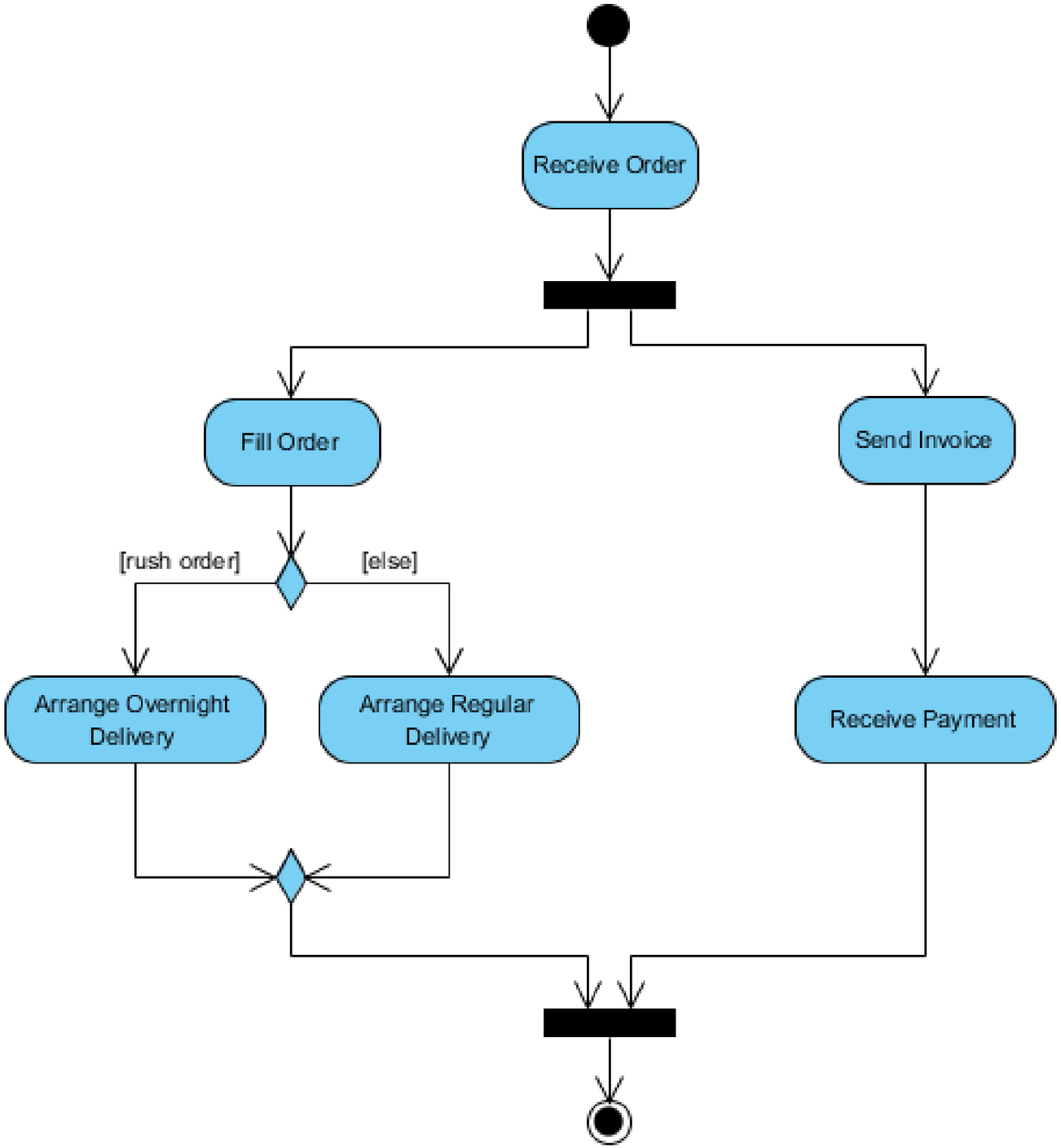


- **Final State or End State –** The state which the system reaches when a particular process or activity ends is known as a Final State or End State.
- We use a filled circle within a circle notation to represent the final state in a state machine diagram.
- A system or a process can have multiple final states.

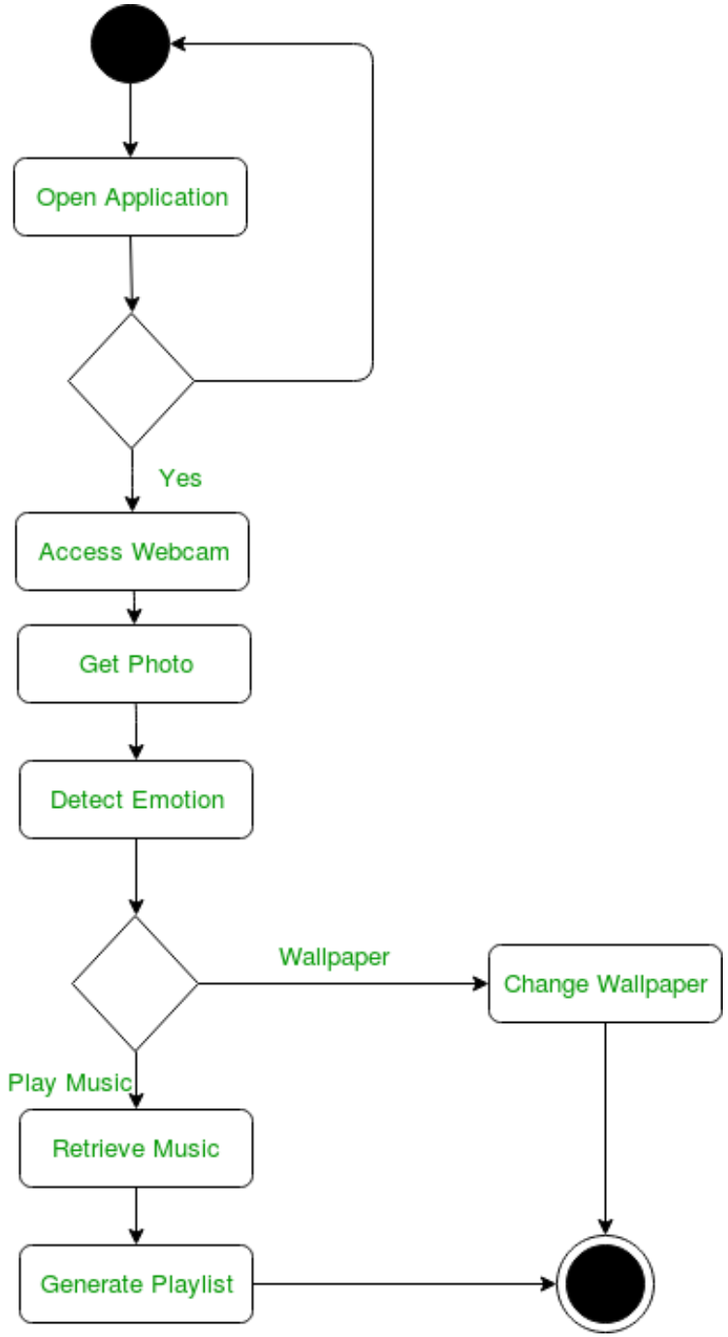



Client enters login name and password
Correct login and password?
No
Invalid login/password
Yes
Client successfully logs in
Client's settings are displayed

EXAMPLE - PROCESS ORDER

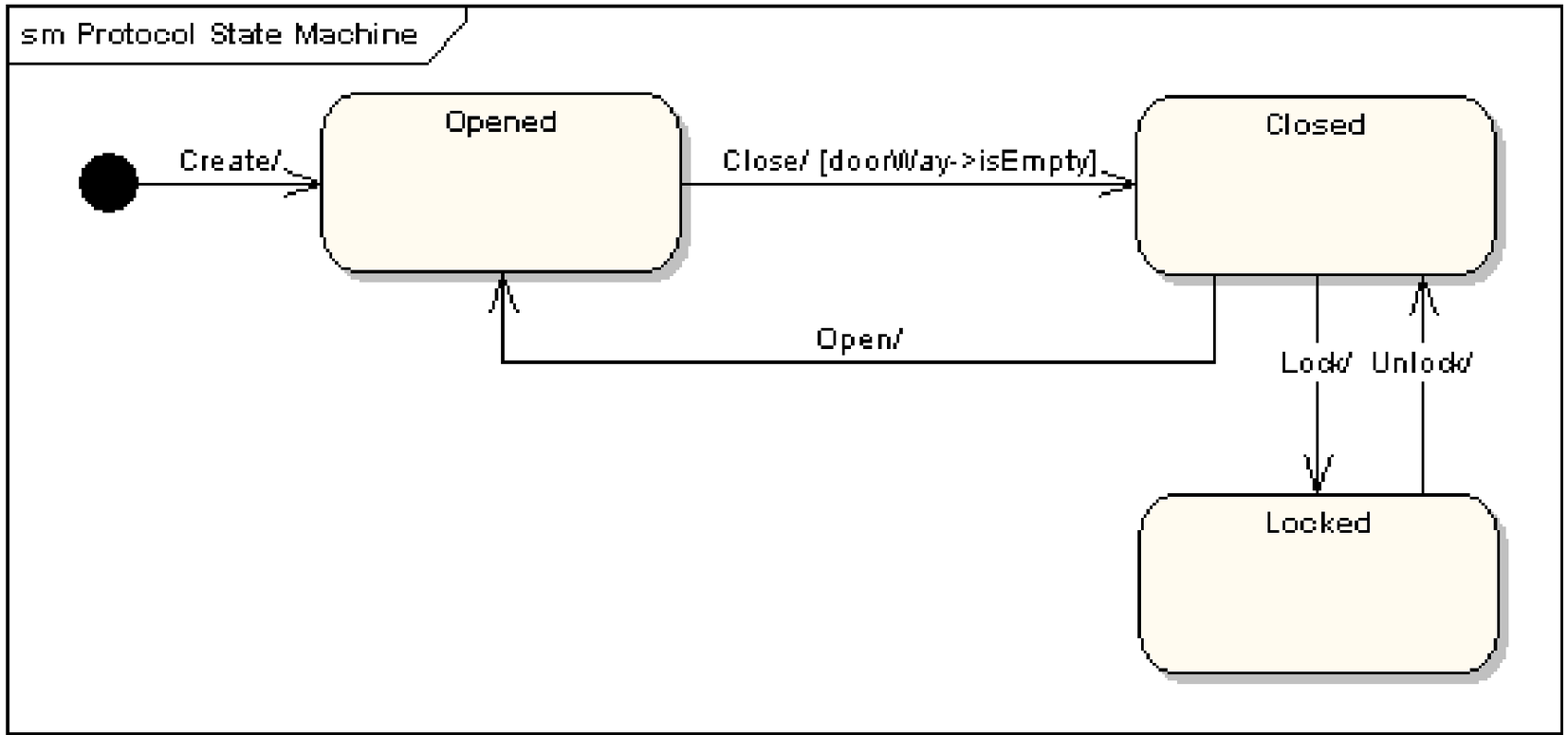


EXAMPLE: AN ACTIVITY DIAGRAM FOR AN EMOTION BASED MUSIC PLAYER



STATE MACHINE DIAGRAM

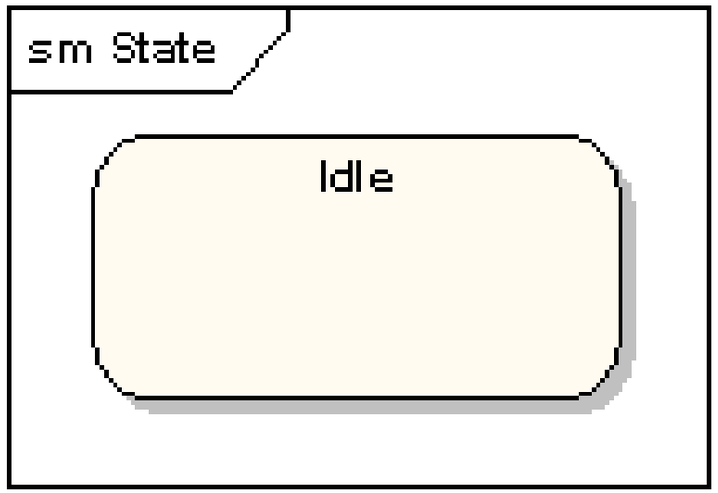
- A state machine diagram models the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.
- As an example, the following state machine diagram shows the states that a door goes through during its lifetime.



- The door can be in one of three states: "Opened", "Closed" or "Locked".
- It can respond to the events Open, Close, Lock and Unlock.
- Notice that not all events are valid in all states; for example, if a door is opened, you cannot lock it until you close it.
- Also notice that a state transition can have a guard condition attached: if the door is Opened, it can only respond to the Close event if the condition doorWay->isEmpty is fulfilled.

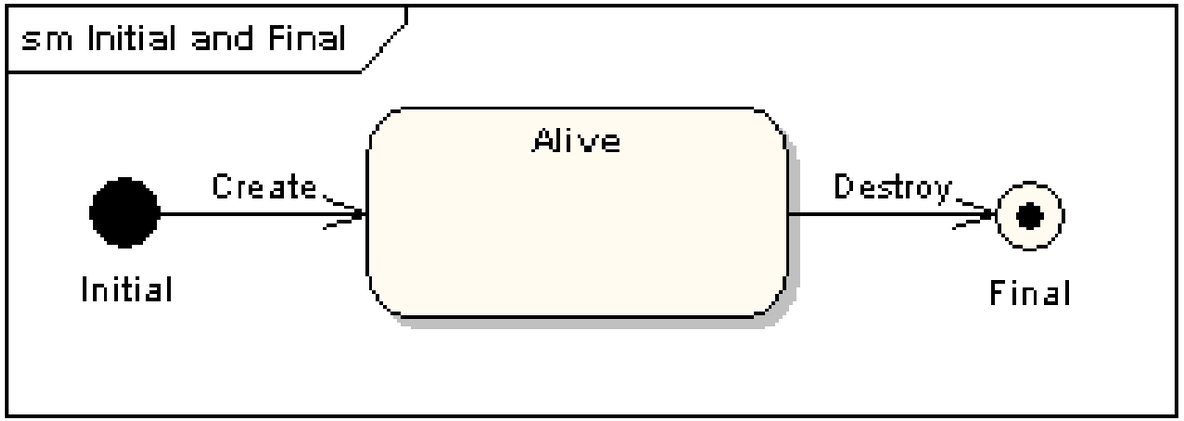
States

- A state is denoted by a round-cornered rectangle with the name of the state written inside it.



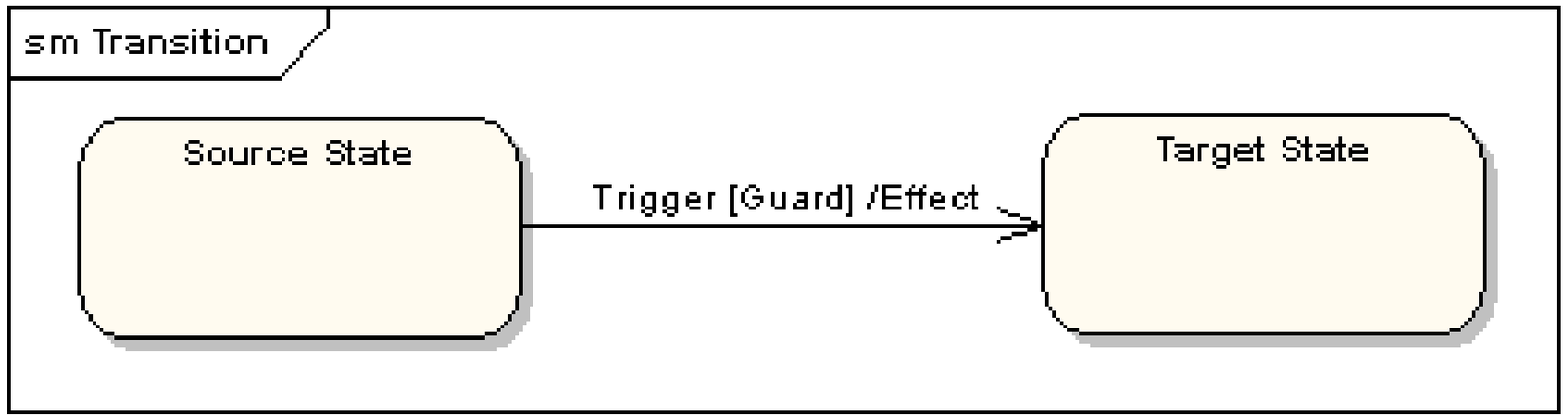
Initial and Final States

- The initial state is denoted by a filled black circle and may be labeled with a name.
- The final state is denoted by a circle with a dot inside and may also be labeled with a name.



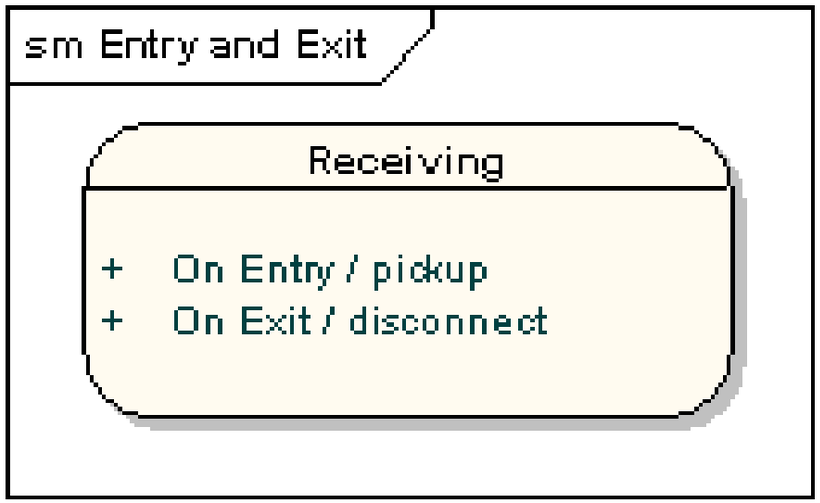
Transitions

- Transitions from one state to the next are denoted by lines with arrowheads.
- A transition may have a trigger, a guard and an effect, as below.
- "Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time.
- "Guard" is a condition which must be true in order for the trigger to cause the transition.
- "Effect" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.



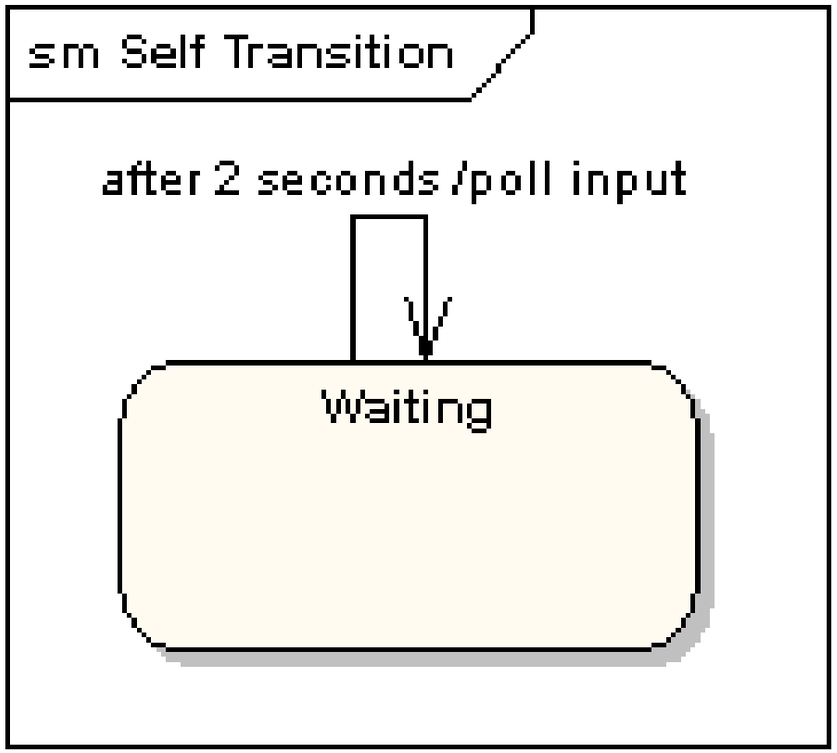
State Actions

- If the target state had many transitions arriving at it, and each transition had the same effect associated with it, it would be better to associate the effect with the target state rather than the transitions.
- This can be done by defining an entry action for the state.
- The diagram below shows a state with an entry action and an exit action.
- It is also possible to define actions that occur on events, or actions that always occur. It is possible to define any number of actions of each type.



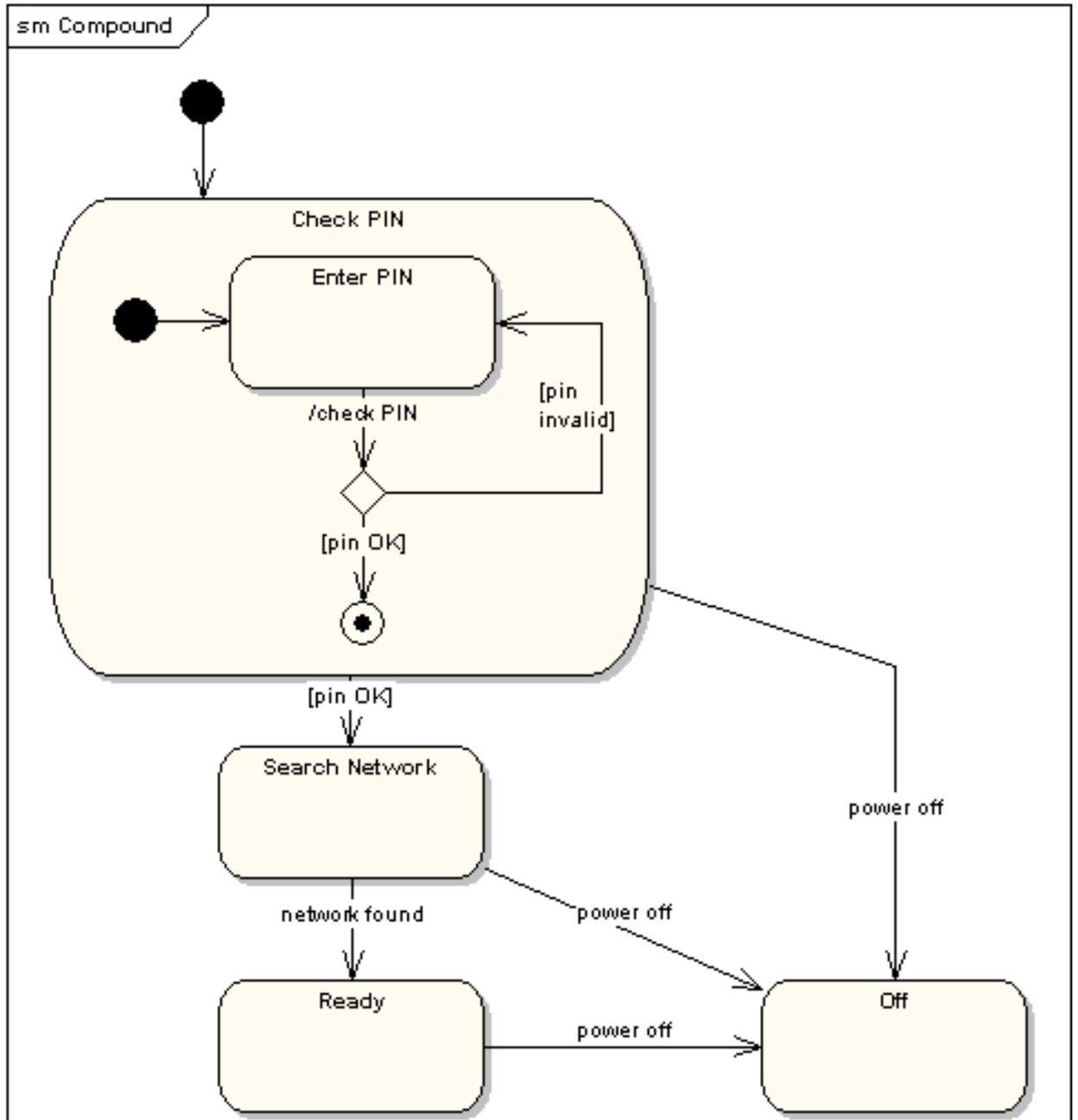
Self-Transitions

- A state can have a transition that returns to itself, as in the following diagram.
- This is most useful when an effect is associated with the transition.



Compound States

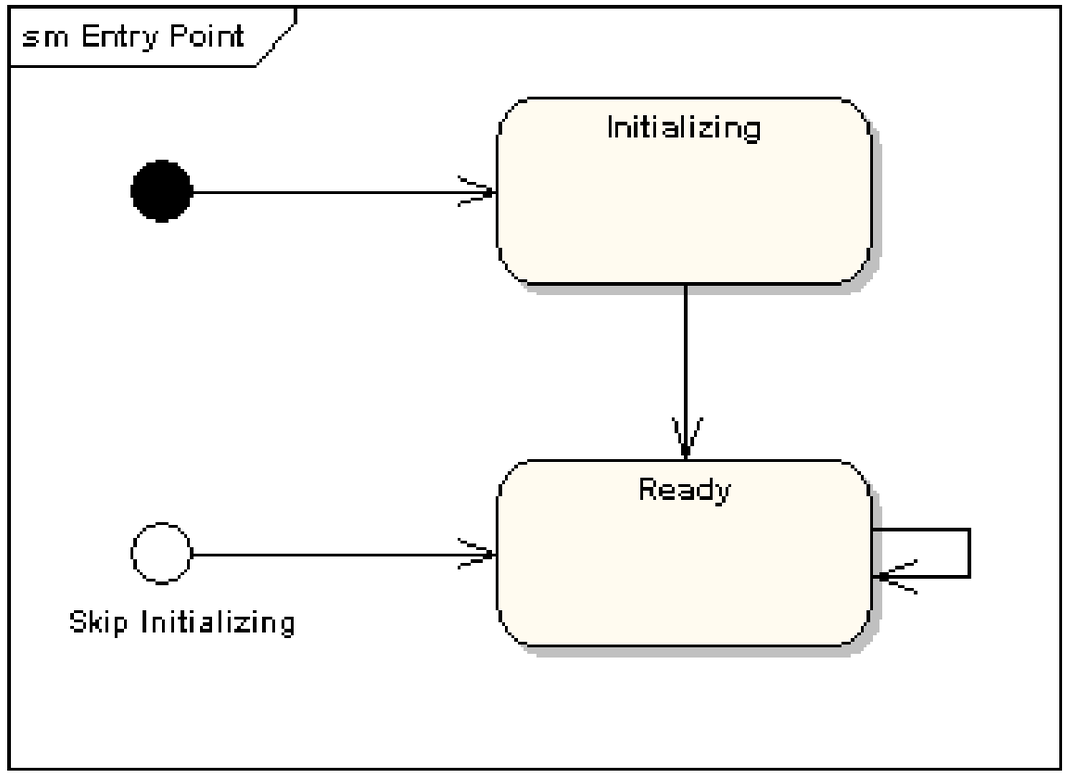
- A state machine diagram may include sub-machine diagrams, as in the example below.



Entry Point

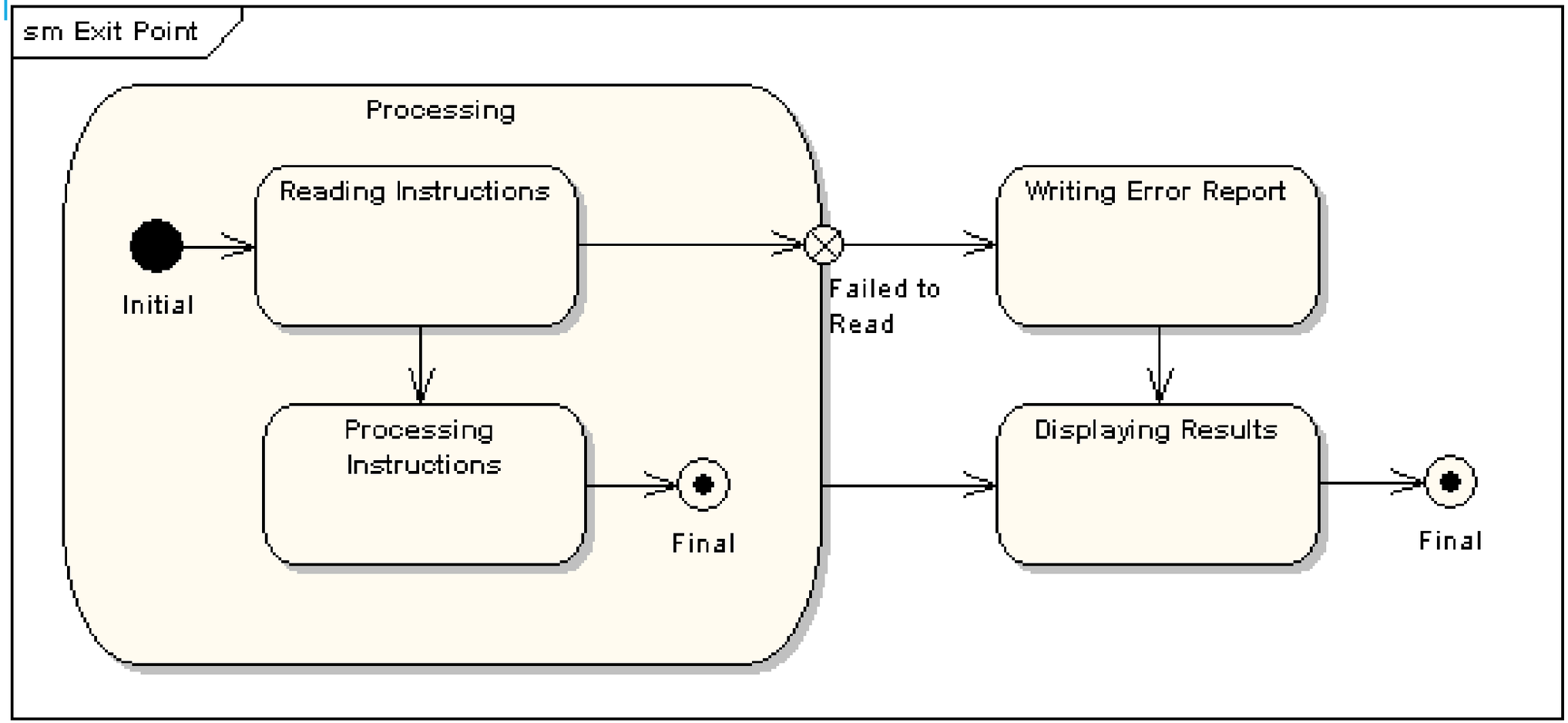
Sometimes you won't want to enter a sub-machine at the normal initial state.

- For example, in the following sub-machine it would be normal to begin in the "Initializing" state, but if for some reason it wasn't necessary to perform the initialization, it would be possible to begin in the "Ready" state by transitioning to the named entry point.



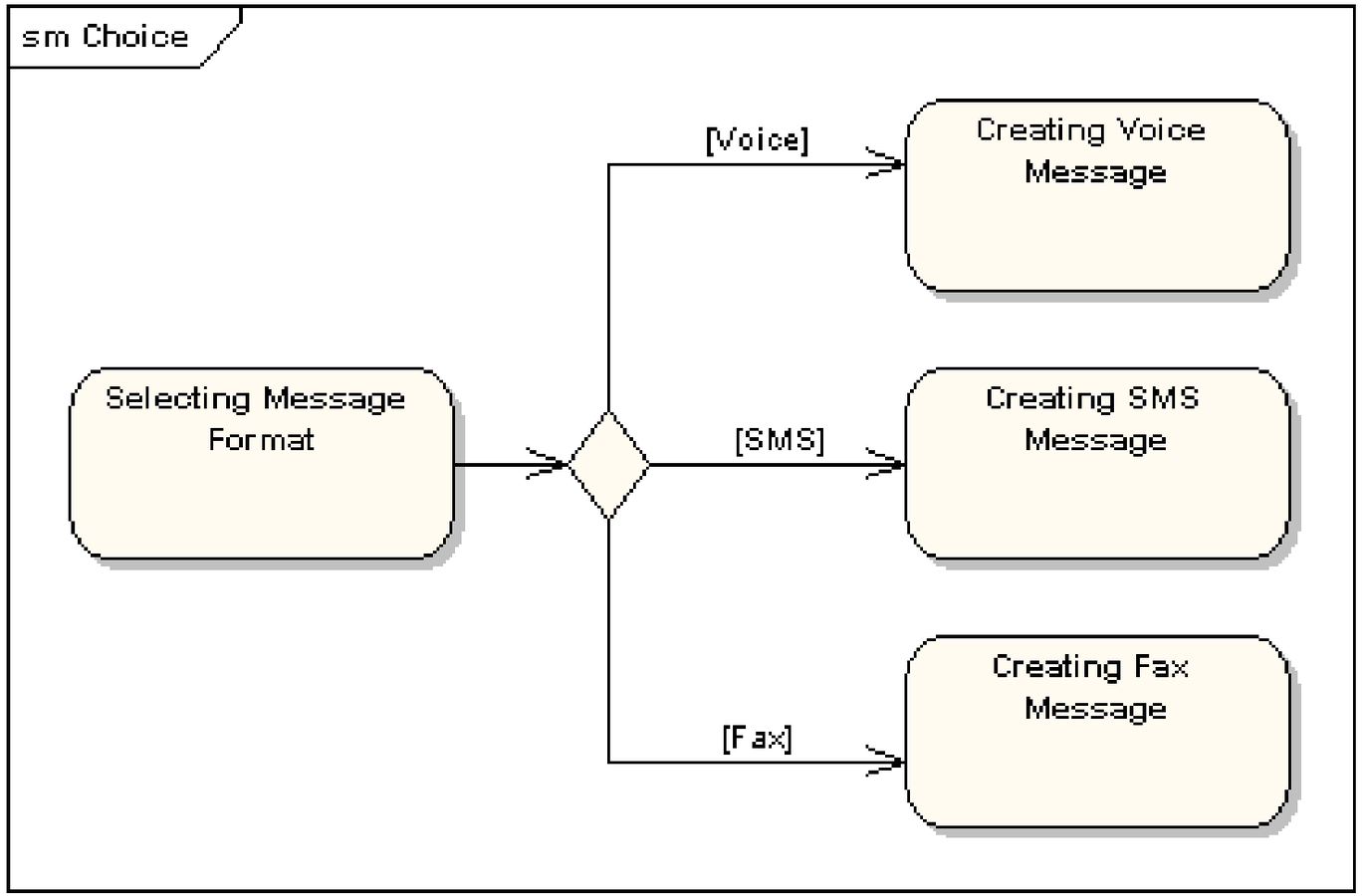
Exit Point

- In a similar manner to entry points, it is possible to have named alternative exit points.
- The following diagram gives an example where the state executed after the main processing state depends on which route is used to transition out of the state.


sm Exit Point
Processing
Reading Instructions
Initial
Processing
Instructions
Final
Failed to
Read
Writing Error Report
Displaying Results
Final

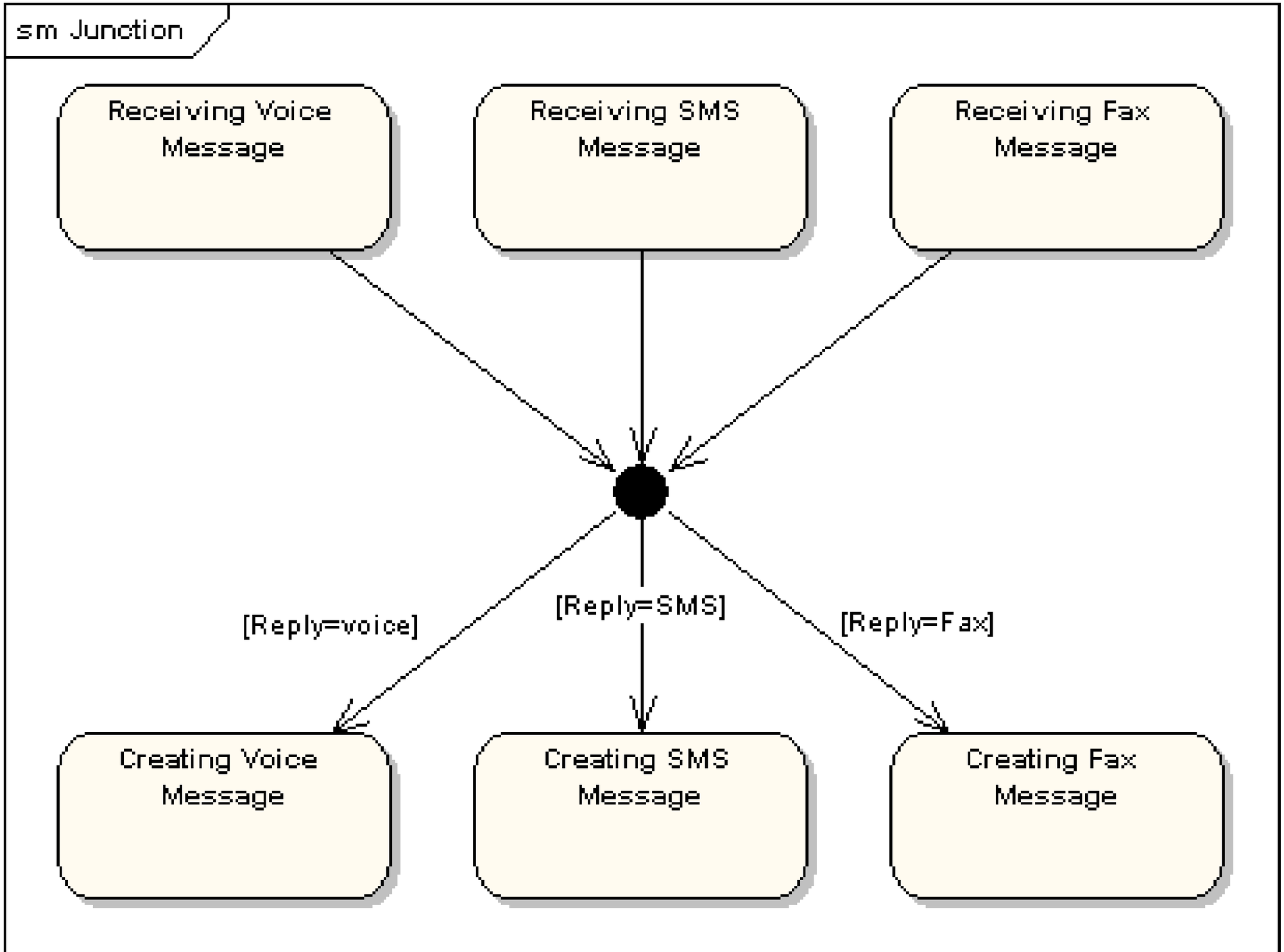
Choice Pseudo-State

- A choice pseudo-state is shown as a diamond with one transition arriving and two or more transitions leaving.
- The following diagram shows that whichever state is arrived at, after the choice pseudo-state, is dependent on the message format selected during execution of the previous state.

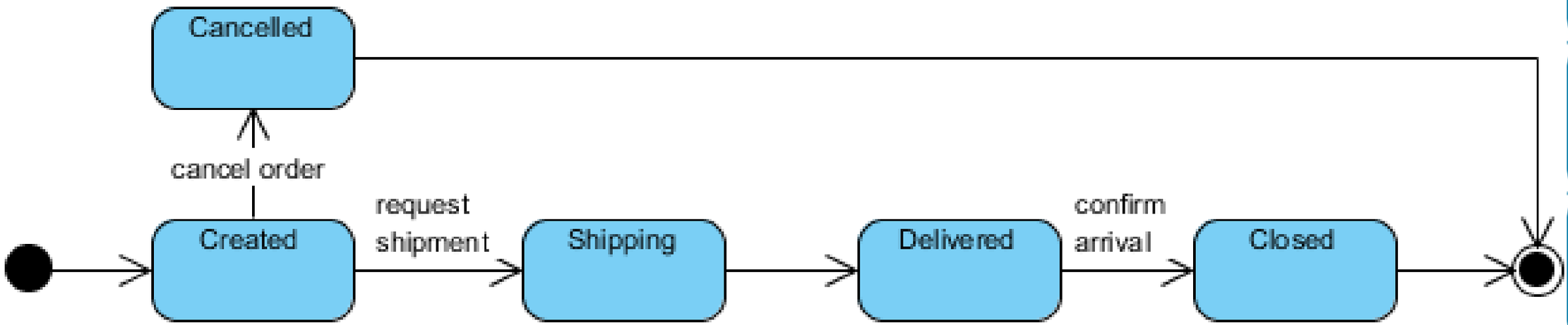


Junction Pseudo-State

- Junction pseudo-states are used to chain together multiple transitions.
- A single junction can have one or more incoming, and one or more outgoing, transitions; a guard can be applied to each transition.
- Junctions are semantic-free.
- A junction which splits an incoming transition into multiple outgoing transitions realizes a static conditional branch, as opposed to a choice pseudo-state which realizes a dynamic conditional branch.


sm Junction
Receiving Voice Message
Receiving SMS Message
Receiving Fax Message
[Reply=voice]
[Reply=SMS]
[Reply=Fax]
Creating Voice Message
Creating SMS Message
Creating Fax Message

ORDER MANAGEMENT STATE DIAGRAM EXAMPLE

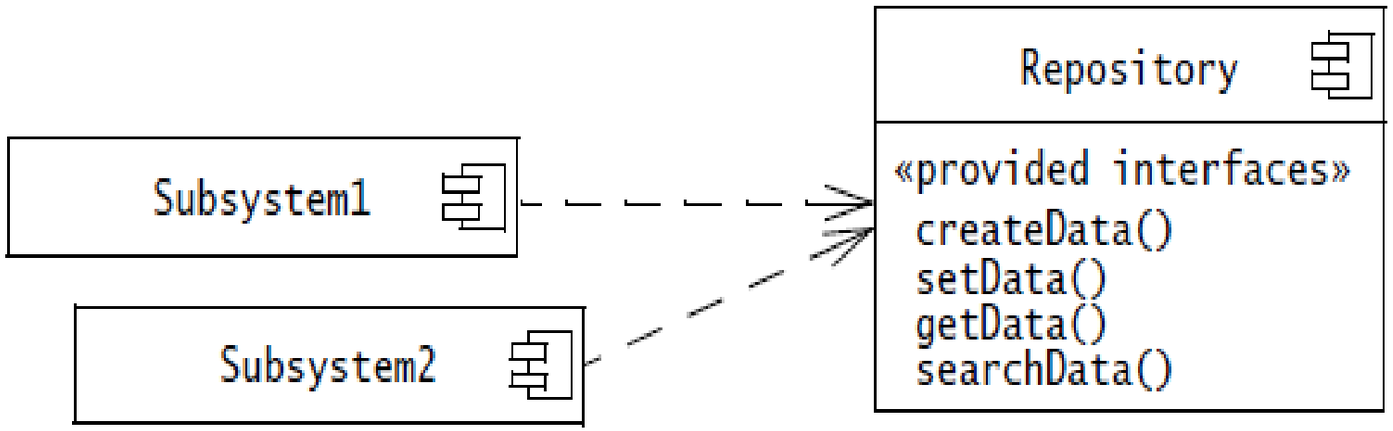


ARCHITECTURAL DESIGN

- Software architecture includes system decomposition, global control flow, handling of boundary conditions, and inter-subsystem communication protocols
- As the complexity of systems increases, the specification of system decomposition is critical.
- It is difficult to modify or correct weak decomposition once development has started, as most subsystem interfaces would have to change.
- There are several architectural styles that can be used as a basis for the architecture of different systems. Some of them are discussed here.

REPOSITORY

- In the **repository architectural style** subsystems access and modify a single data structure called the central **repository**.
- Subsystems are relatively independent and interact only through the repository.
- Control flow can be dictated either by the central repository (e.g., triggers on the data invoke peripheral systems) or by the subsystems (e.g., independent flow of control and synchronization through locks in the repository).



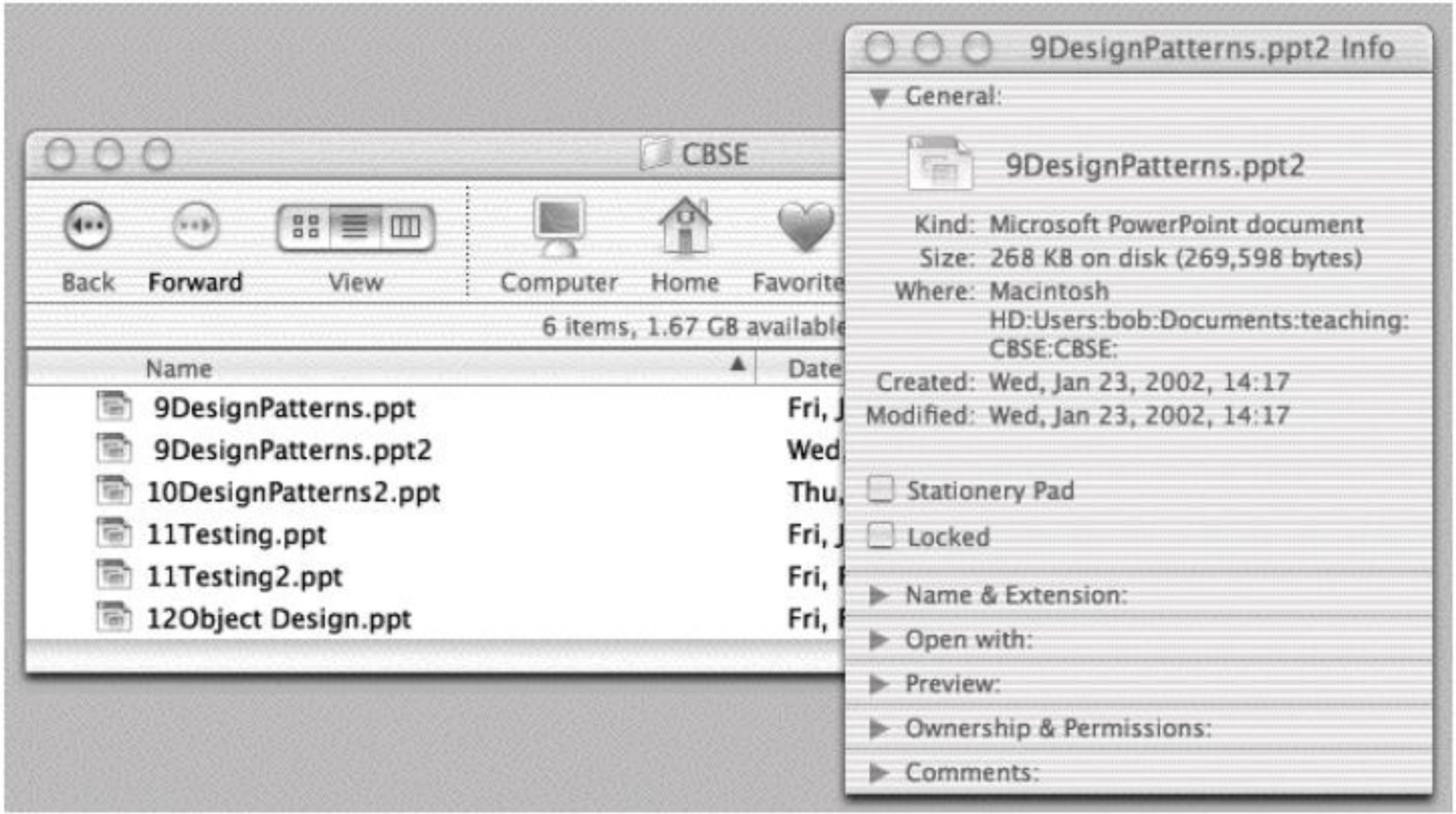
- Repositories are well suited for applications with constantly changing, complex data processing tasks.
- Once a central repository is well defined, we can easily add new services in the form of additional subsystems.
- The main disadvantage of repository systems is that the central repository can quickly become a bottleneck, both from a performance aspect and a modifiability aspect.
- The coupling between each subsystem and the repository is high, thus making it difficult to change the repository without having an impact on all subsystems.

MODEL/VIEW/CONTROLLER

- In the **Model/View/Controller (MVC)** architectural style subsystems are classified into three different types:
  - ✓ **model subsystems : maintain domain knowledge,**
  - ✓ **View subsystems** :display it to the user, and
  - ✓ **controller subsystems :** manage the sequence of interactions with the user.
- The model subsystems are developed such that they do not depend on any view or controller subsystem.
- Changes in their state are propagated to the view subsystem via a subscribe/notify protocol.
- The MVC is a special case of the repository where Model implements the central data structure and control objects dictate the control flow.

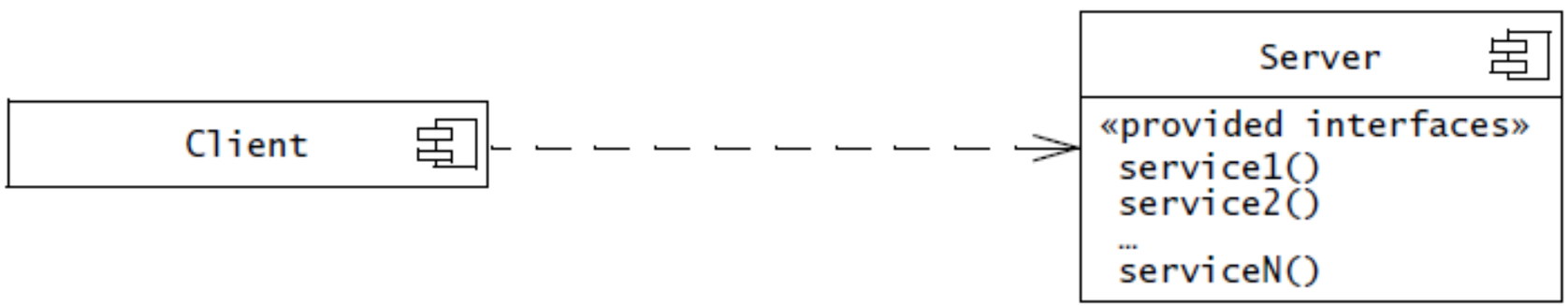

Controller
initiator
*
repository
1
Model
1
notifier
View
subscriber
*

- An example of MVC architectural style.
- The “model” is the filename 9DesignPAtterns2.ppt.
- One “view” is a window titled CBSE, which displays the contents of a folder containing the file 9DesignPatterns2.ppt.
- The other “view” is window called 9DesignPatterns2.ppt Info, which displays information related to the file. If the file name is changed, both views are updated by the “controller.”


CBSE
Back
Forward
View
Computer
Home
Favorite
6 items, 1.67 GB available
Name
Date
9DesignPatterns.ppt
9DesignPatterns.ppt2
10DesignPatterns2.ppt
11Testing.ppt
11Testing2.ppt
12Object Design.ppt
9DesignPatterns.ppt2 Info
General:
9DesignPatterns.ppt2
Kind: Microsoft PowerPoint document
Size: 268 KB on disk (269,598 bytes)
Where: Macintosh HD:Users:bob:Documents:teaching: CBSE:CBSE:
Created: Wed, Jan 23, 2002, 14:17
Modified: Wed, Jan 23, 2002, 14:17
Stationery Pad
Locked
Name & Extension:
Open with:
Preview:
Ownership & Permissions:
Comments:

CLIENT/SERVER

- In the **client/server architectural style,** a subsystem, the server, provides services to instances of other subsystems called the clients, which are responsible for interacting with the user.
- The request for a service is usually done via a remote procedure call mechanism or a common object broker (e.g., CORBA, Java RMI, or HTTP).
- Control flow in the clients and the servers is independent except for synchronization to manage requests or to receive results.



- An information system with a central database is an example of a client/server architectural style.
- The clients are responsible for receiving inputs from the user, performing range checks, and initiating database transactions when all necessary data are collected.
- The server is then responsible for performing the transaction and guaranteeing the integrity of the data.
- Client/server systems, however, are not restricted to a single server.
- Client/server architectural styles are well suited for distributed systems that manage large amounts of data.

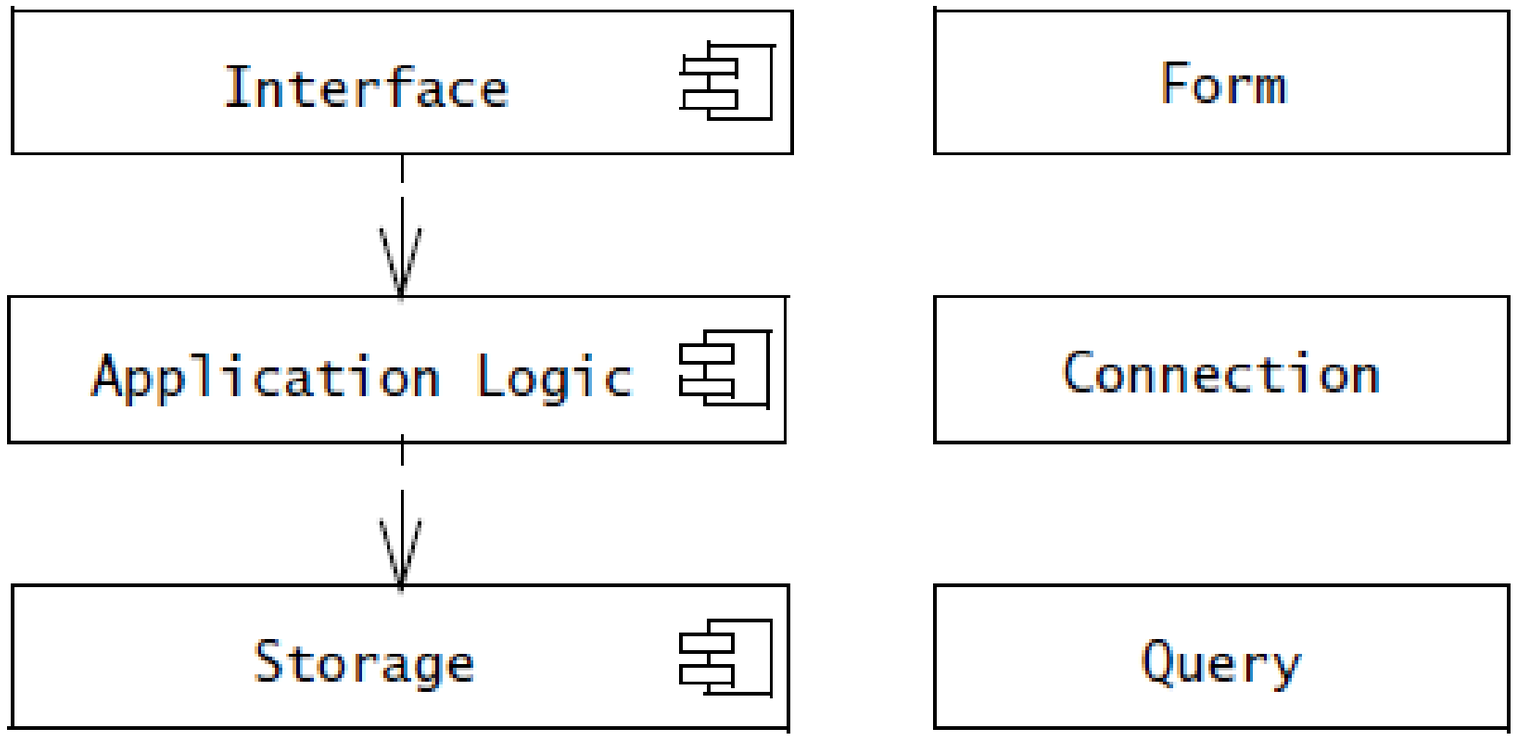
PEER-TO-PEER

- **peer-to-peer architectural style** is a generalization of the client/server architectural style in which subsystems can act both as client or as servers, in the sense that each subsystem can request and provide services.
- The control flow within each subsystem is independent from the others except for synchronizations on requests.

THREE-TIER

The **three-tier architectural style organizes** subsystems into three layers:

- **The *interface layer*** includes all boundary objects that deal with the user, including windows, forms, web pages, and so on.
- **The *application logic layer*** includes all control and entity objects, realizing the processing, rule checking, and notification required by the application.
- **The *storage layer*** realizes the storage, retrieval, and query of persistent objects.

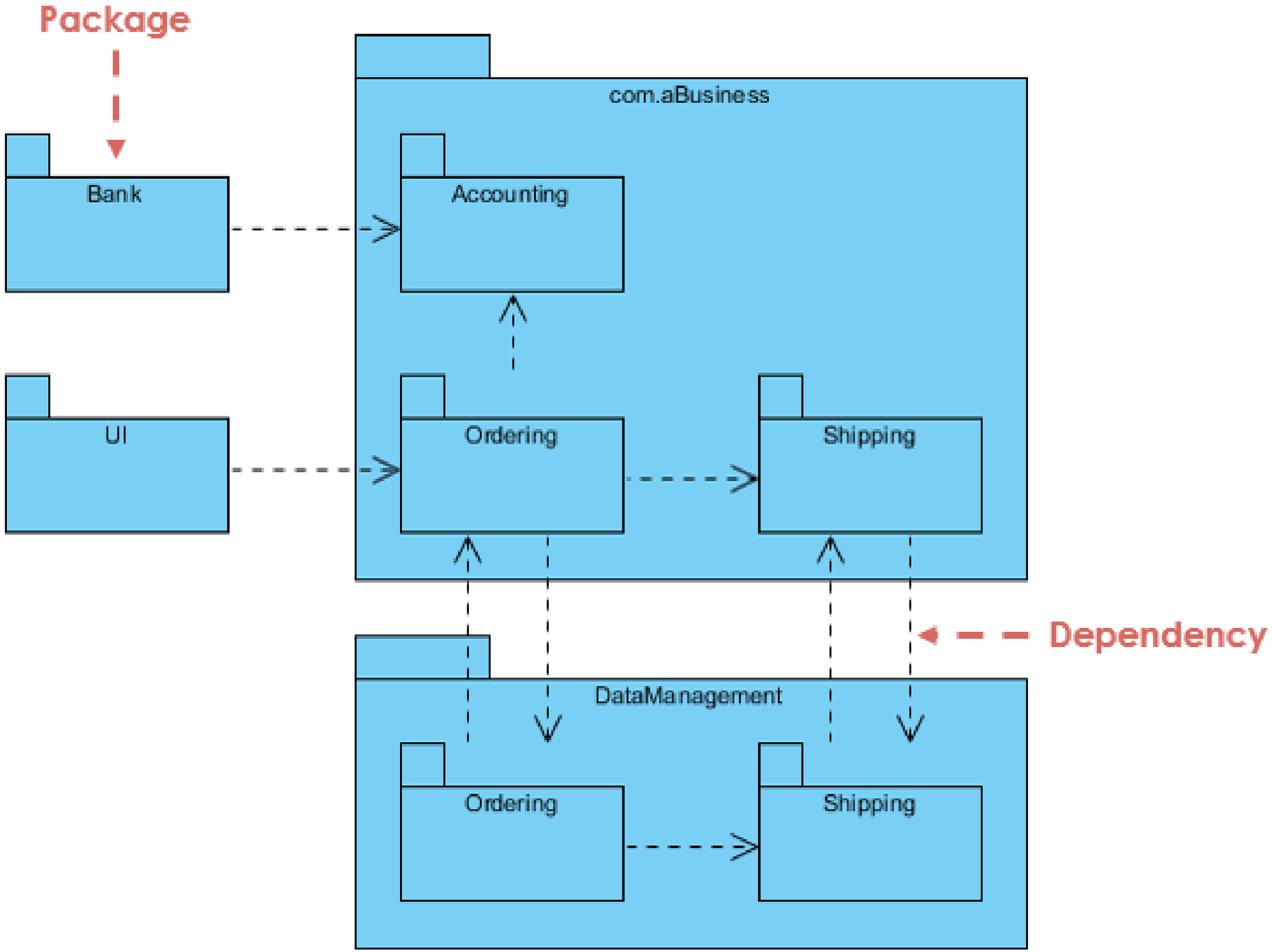

Interface
Application Logic
Storage
Form
Connection
Query

PACKAGE DIAGRAM

Package diagrams are used to structure high level system elements. Packages are used for organizing large system which contains diagrams, documents and other key deliverables.

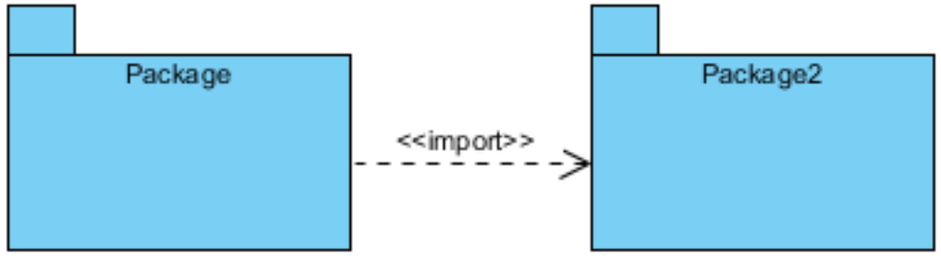
- ✓Package Diagram can be used to simplify complex class diagrams, it can group classes into packages.
- ✓A package is a collection of logically related UML elements.
- ✓Packages are depicted as file folders and can be used on any of the UML diagrams.

- Packages appear as rectangles with small tabs at the top.
- The package name is on the tab or inside the rectangle.
- The dotted arrows are dependencies.
- One package depends on another if changes in the other could possibly force changes in the first.


Package
com.aBusiness
Bank
Accounting
UI
Ordering
Shipping
Dependency
DataManagement
Ordering
Shipping

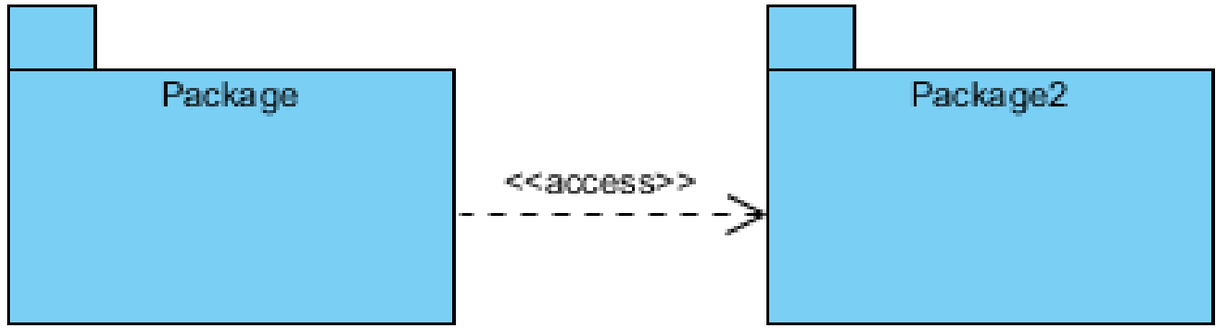
- Package Diagram Example - Import

<<import>> - one package imports the functionality of other package



- Package Diagram Example - Access

<<access>> - one package requires help from functions of other package.



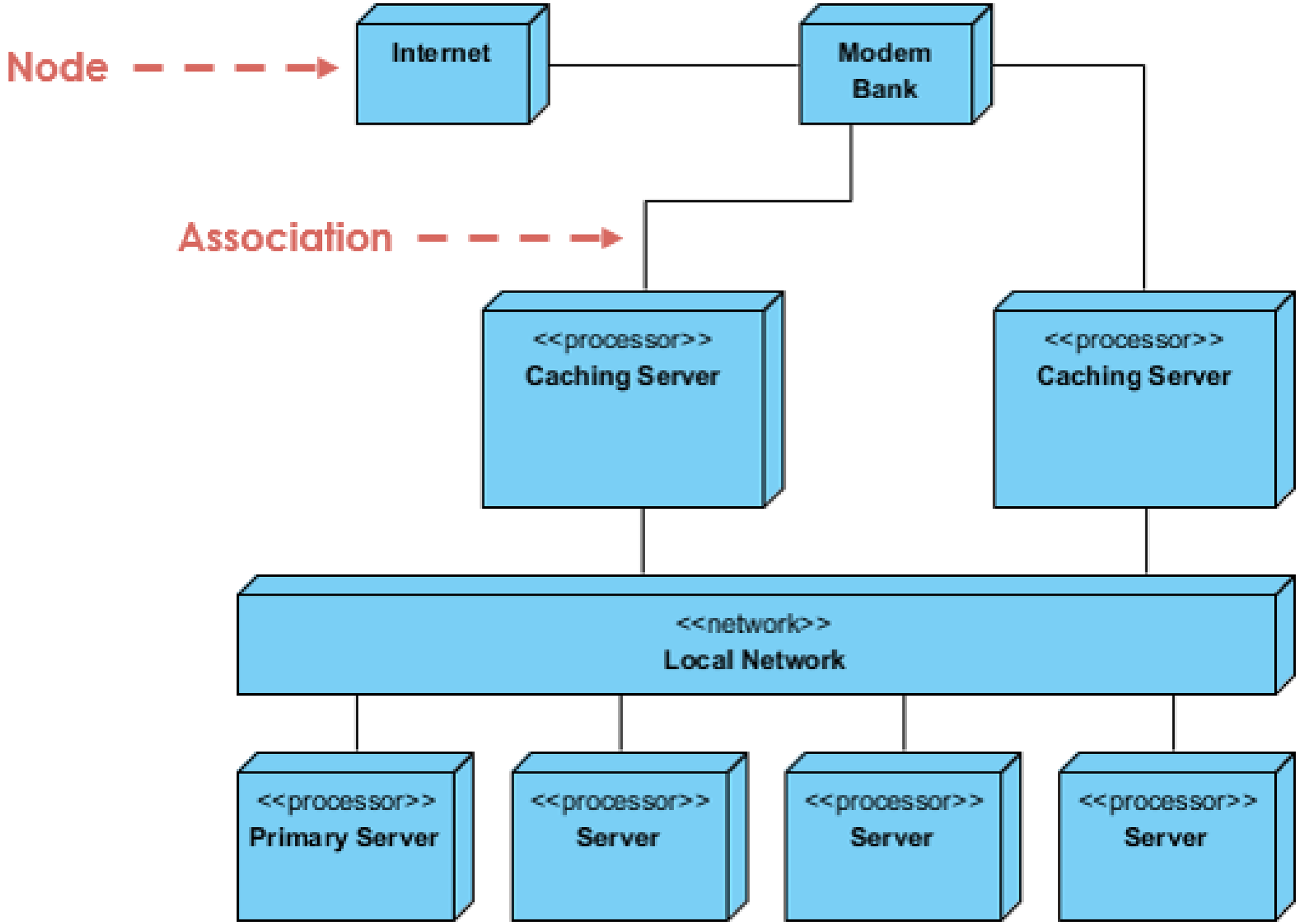
DEPLOYMENT DIAGRAM

- They show the structure of the run-time system
- They capture the hardware that will be used to implement the system and the links between different items of hardware.
- They model physical hardware elements and the communication paths between them
- They can be used to plan the architecture of a system.
- They are also useful for Document the deployment of software components or nodes

- A deployment diagram is just a special kind of class diagram, which focuses on a system's nodes. Graphically, a deployment diagram is a collection of vertices and arcs. Deployment diagrams commonly contain:

- Nodes
  - 3-D box represents a node, either software or hardware
  - HW node can be signified with <<stereotype>>
  - Connections between nodes are represented with a line, with optional <<stereotype>>
  - Nodes can reside within a node

- Other Notations
  - Dependency
  - Association relationships.
  - May also contain notes and constraints.


Node
Association
Internet
Modem Bank
<<processor>>
Caching Server
<<processor>>
Caching Server
<<network>>
Local Network
<<processor>>
Primary Server
<<processor>>
Server
<<processor>>
Server
<<processor>>
Server