ABSTRACTION

A model that includes most important aspects of a given system while ignoring less important details

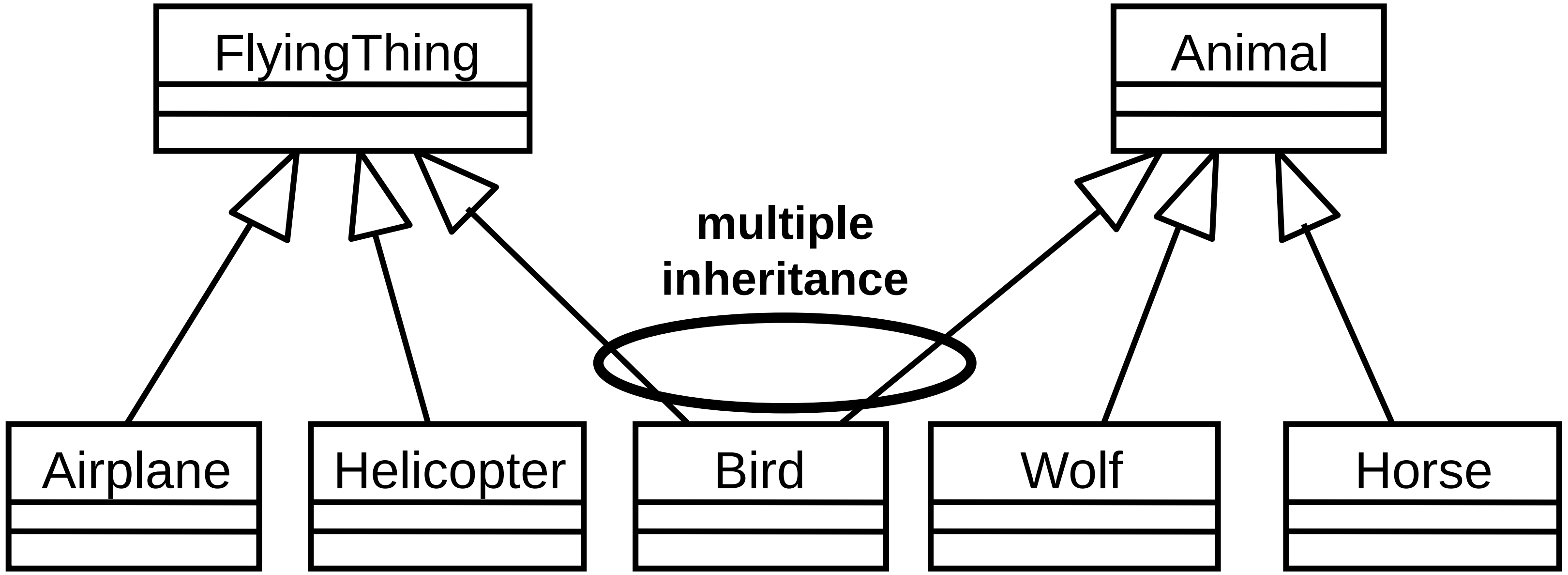
Abstraction allows us to manage complexity by concentrating on essential characteristics that makes an entity different from others.

Not saying Which salesman – just a salesman in general!!!

Example of an order processing abstraction.

MULTIPLE INHERITANCE

A class can inherit from several other classes



Use multiple inheritance only when needed, and always with caution!!!

What is modeling

- ✓ Modeling consists of building an abstraction of reality.
- ✓ Abstractions are simplifications because:
  - ➢ They ignore irrelevant details and
  - ➢ They only represent the relevant details.
- ✓ What is relevant or irrelevant depends on the purpose of the model.

Why Modeling SW

- ✓ Software is getting increasingly more complex
  - ➢ Windows XP > 40 million lines of code
  - ➢ A single programmer may not manage this amount of code in his entire coding life.
- ✓ Code is not easily understandable by developers who did not write it
- ✓ We need simpler representations for complex systems
  - ➢ Modeling is a means for dealing with complexity

What is UML?

Unified Modeling Language

- OMG Standard Object Management Group

UML is a modeling language to express and design documents, software

- Particularly useful for OO design
- Independent of implementation language

Why use UML

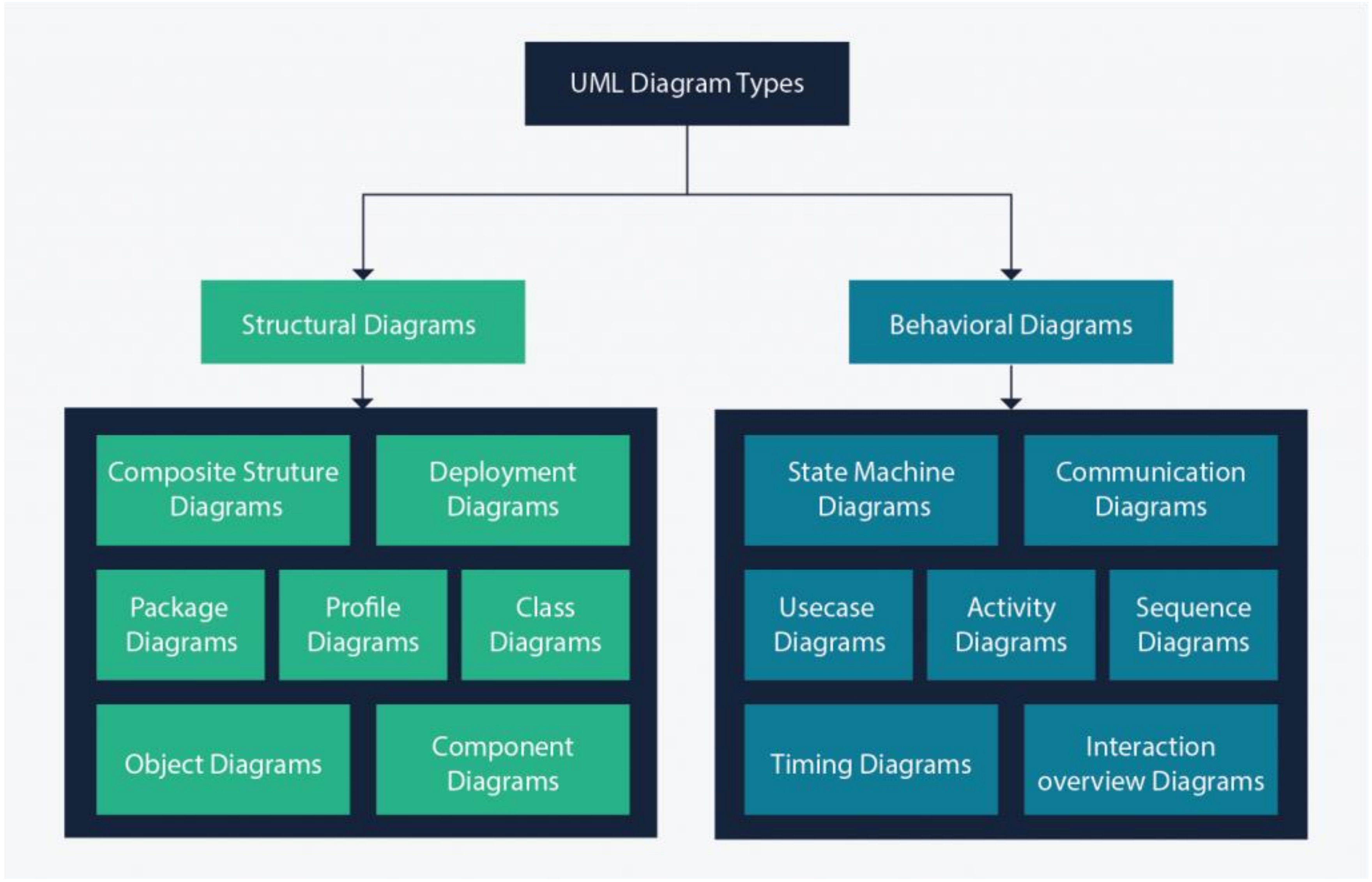
- ✓ The industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market.
- ✓ Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale.

The Unified Modeling Language (UML) was designed to respond to these needs.

- ✓ Open Standard, Graphical notation for
  - ✓ Specifying, visualizing, constructing, and documenting software systems
- ✓ Increase understanding/communication of product to customers and developers
- ✓ Support for diverse application areas
- ✓ Support for UML in many software packages today (e.g. Rational, plugins for popular IDE's like NetBeans, Eclipse)

UML Diagram Types

There are two main categories: **structure diagrams** and **behavioral diagrams**.



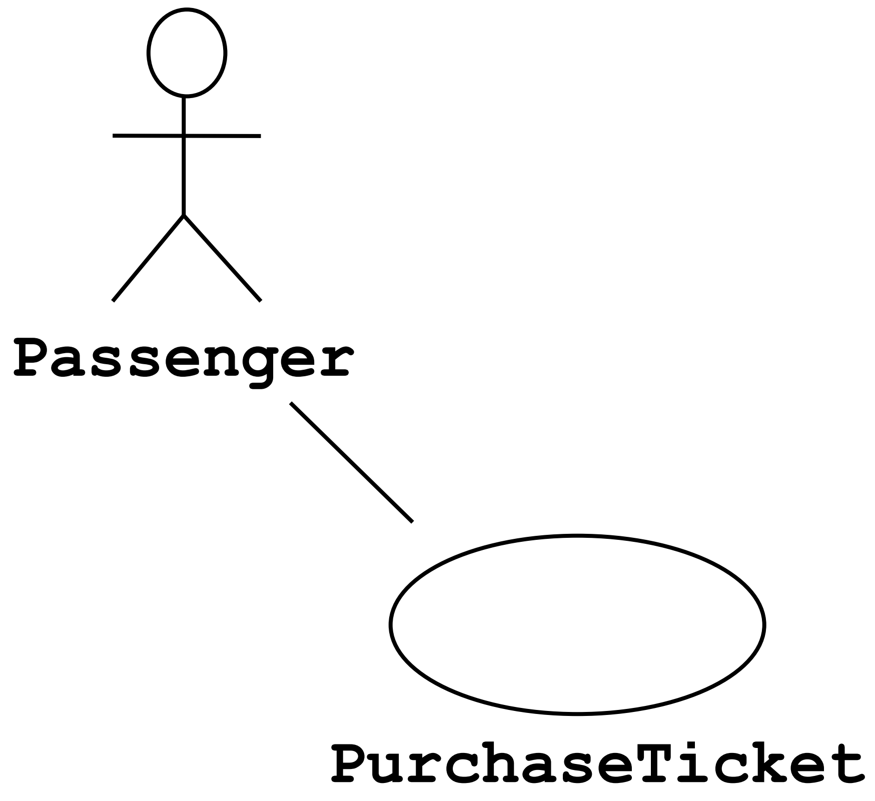
USE CASE DIAGRAMS

Used during requirements elicitation to represent external behavior

Actors represent roles, that is, a type of user of the system

Use cases represent a sequence of interaction for a type of functionality; summary of scenarios

Use case diagrams give a graphic overview of the **actors** involved in a system, **different functions** needed by those **actors** and how these different functions interact



USE CASE DIAGRAMS (CONT'D)

ACTOR:

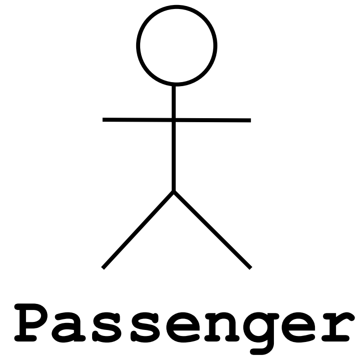
✓ An actor models an external entity which communicates with the system:

- ➢ User
- ➢ External system
- ➢ Physical environment

✓ An actor has a unique name and an optional description.

✓ Examples:

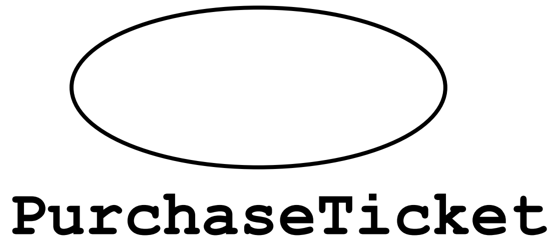
- ➢ Passenger: A person in the train
- ➢ GPS satellite: Provides the system with GPS coordinates



USE CASE DIAGRAMS (CONT'D)

USE CASE:

- ✓A use case represents a class of functionality provided by the system as an event flow.
  - ✓A use case consists of:
  - ✓Unique name
  - ✓Participating actors
  - ✓Entry conditions
  - ✓Flow of events
  - ✓Exit conditions
  - ✓Special requirements



USE CASE DIAGRAMS (CONT'D)

USE CASE : EXAMPLE

Name: `Purchase ticket`

Participating actor: `Passenger`

Entry condition:

- `Passenger` standing in front of ticket distributor.
- `Passenger` has sufficient money to purchase ticket.

Exit condition:

- `Passenger` has ticket.

Event flow:

1. `Passenger` selects the number of zones to be traveled.
2. Distributor displays the amount due.
3. `Passenger` inserts money, of at least the amount due.
4. Distributor returns change.
5. Distributor issues ticket.

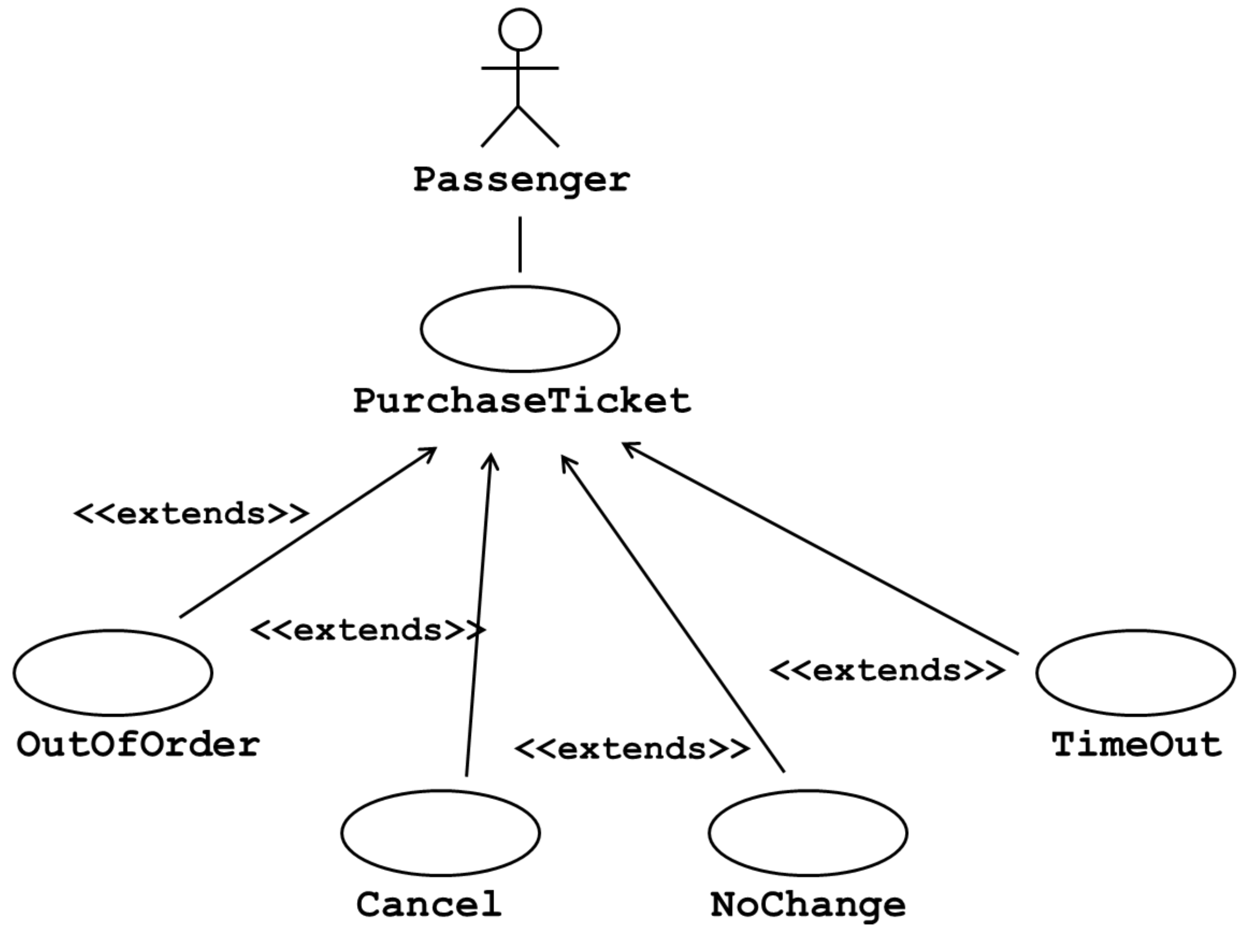
Anything missing?

Exceptional cases!

Use Case Diagrams (cont'd)

The <<*extends*>> Relationship

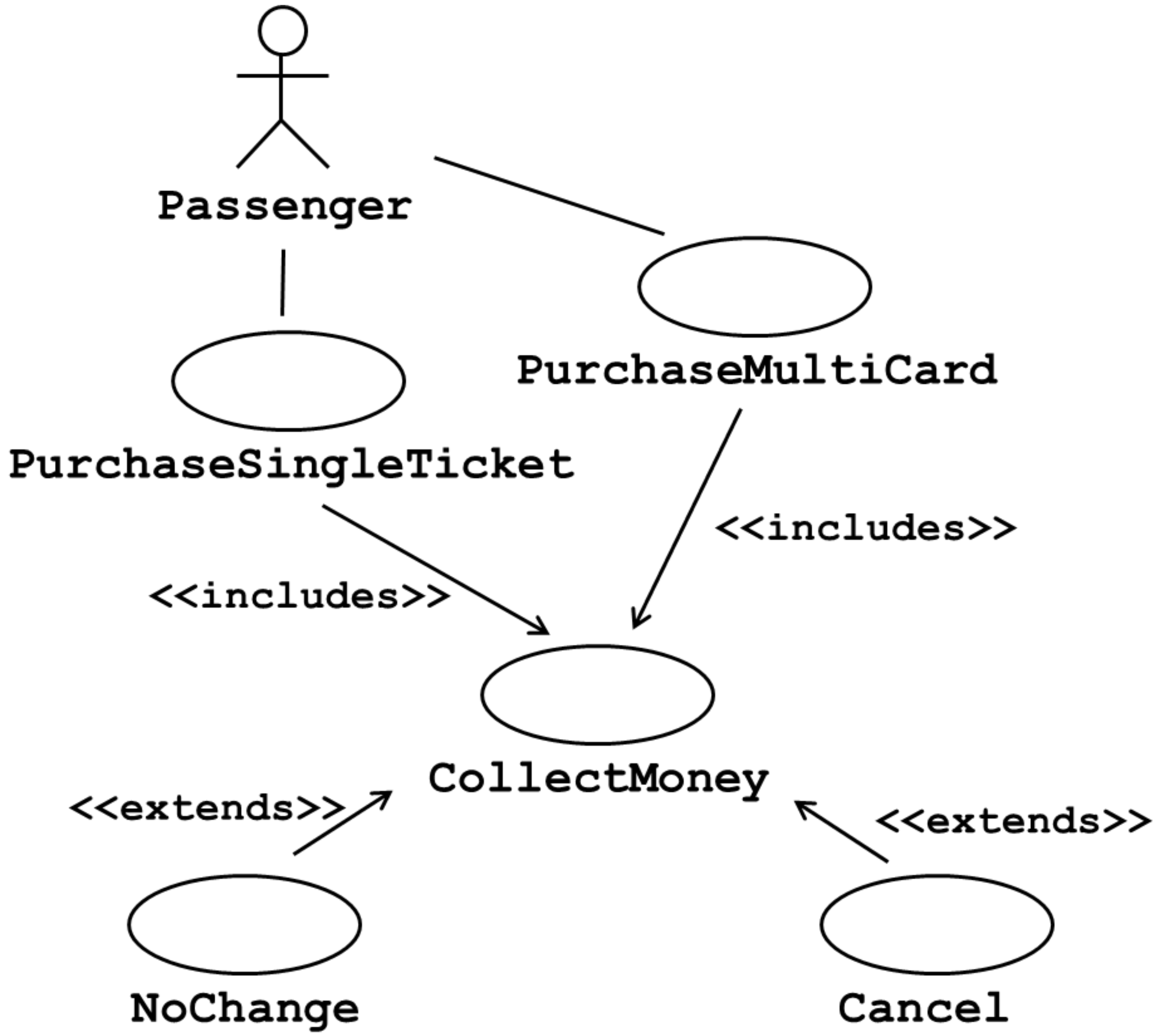
- `<<extends>>` relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a `<<extends>>` relationship is to the extended use case



USE CASE DIAGRAMS (CONT'D)

THE <<INCLUDES>> RELATIONSHIP

- <<includes>> relationship represents behavior that is factored out of the use case.
- <<includes>> behavior is factored out for reuse, not because it is an exception.
- is used to extract use case fragments that are duplicated in multiple use cases.
- The included use case cannot stand alone and the original use case is not complete without the included one.



USE CASES ARE USEFUL TO...

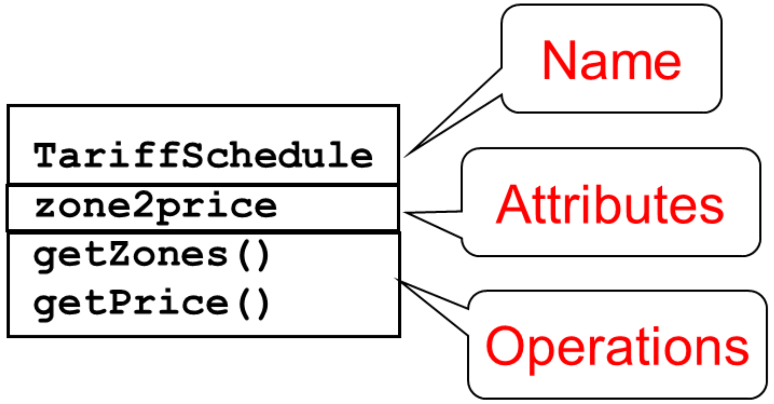
- ✓ Determining requirements
  - ➢ New use cases often generate new requirements as the system is analyzed and the design takes shape.
- ✓ Communicating with clients
  - ➢ Their notational simplicity makes use case diagrams a good way for developers to communicate with clients.
- ✓ Generating test cases
  - ➢ The collection of scenarios for a use case may suggest a suite of test cases for those scenarios.

Class Diagrams

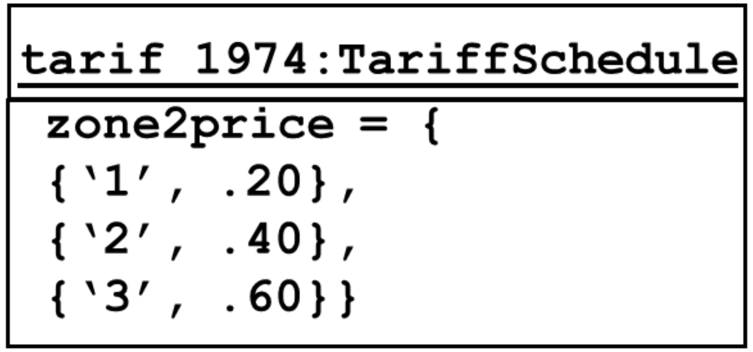
- ✓Gives an overview of a system by showing its classes and the relationships among them.
  - ➢Class diagrams are static
  - ➢they display what interacts but not what happens when they do interact
- ✓Also shows attributes and operations of each class
- ✓Good way to describe the overall architecture of system components

CLASSES AND INSTANCES

- ✓A ***class*** represent a concept
- ✓A class encapsulates state ***(attributes)*** and behavior ***(operations).***
- ✓Each attribute has a ***type***.
- ✓Each operation has a ***signature***.
- ✓The class name is the only mandatory information.

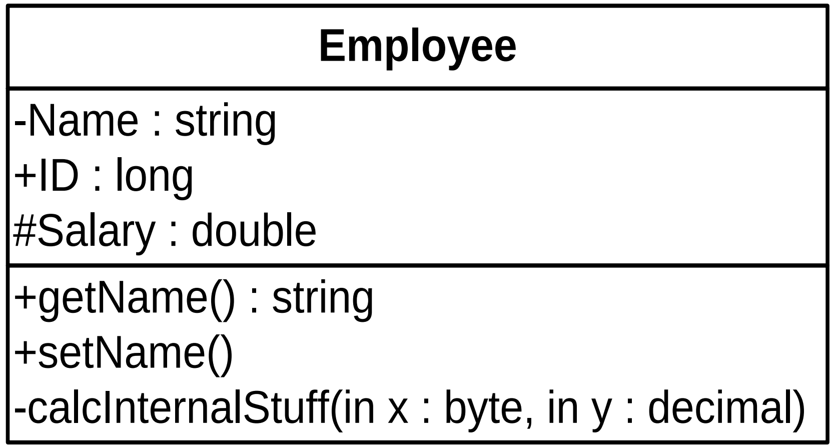


- ✓An ***instance*** represents a phenomenon.
- ✓The name of an instance is <u>underlined</u> and can contain the class of the instance.
- ✓The attributes are represented with their ***values***.



UML Class Notation

- A class is a rectangle divided into three parts
  - Class name
  - Class attributes (i.e. data members, variables)
  - Class operations (i.e. methods)
- Modifiers
  - Private: -
  - Public: +
  - Protected: #
  - Static: Underlined (i.e. shared among all members of the class)
- Abstract class: Name in italics



RELATIONSHIP

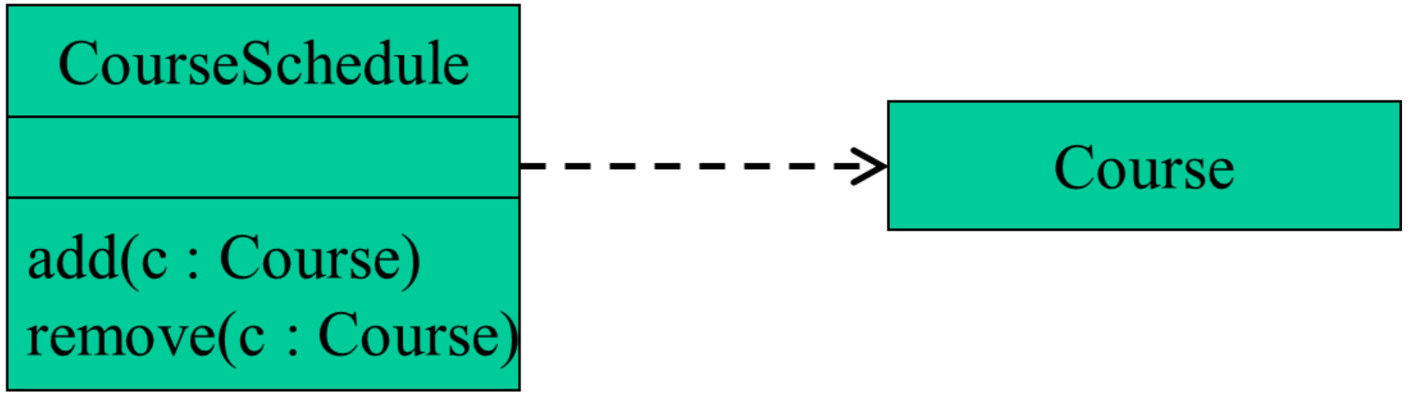
In UML, object interconnections (logical or physical), are modeled as relationships.

There are three kinds of relationships in UML:

- dependencies
- generalizations
- associations

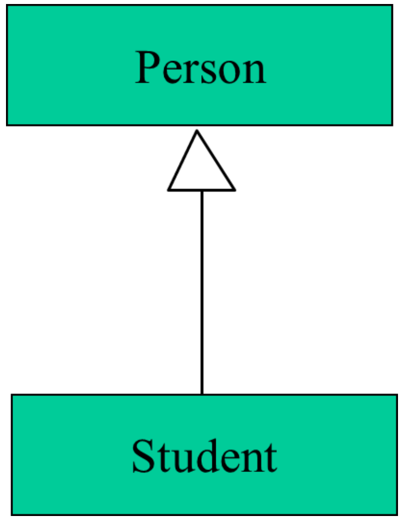
Relationship - Depednency

A *dependency* 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*.

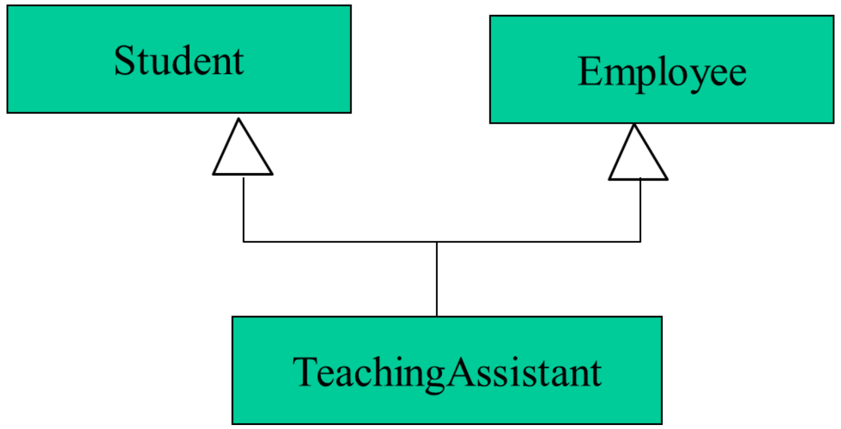


RELATIONSHIP - GENERALIZATION

- ✓ A *generalization* 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.

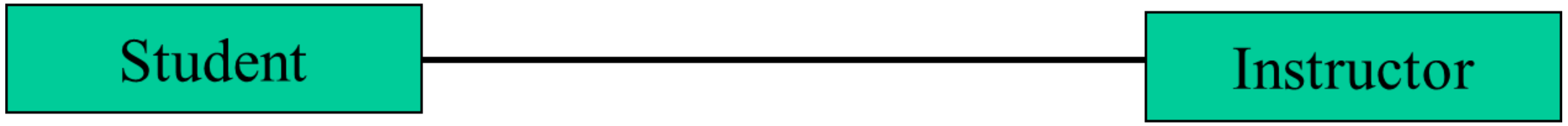


UML permits a class to inherit from multiple superclasses, although some programming languages may not permit multiple inheritance.

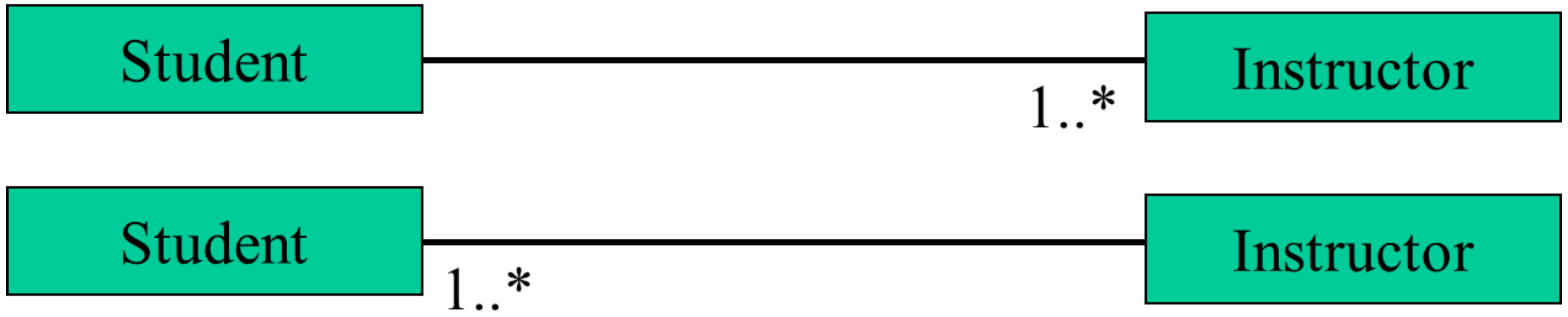


Relationship - Association

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

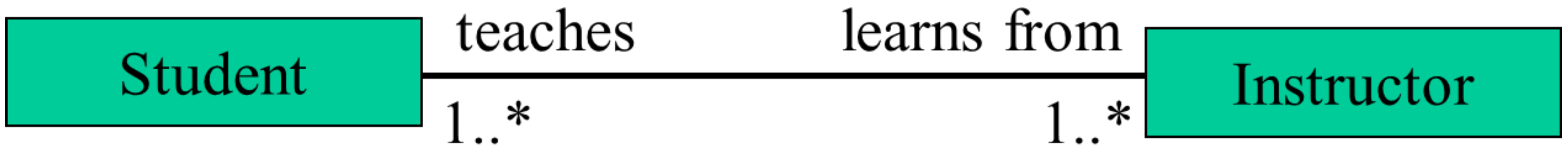


We can indicate the multiplicity of an association by adding multiplicity adornments to the line denoting the association.
below example indicates that a Student has one or more Instructors and next one indicates that every Instructor has one or more Students: :

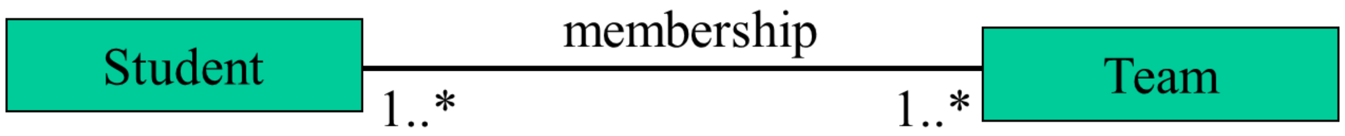


Relationship – Association (Cont'd)

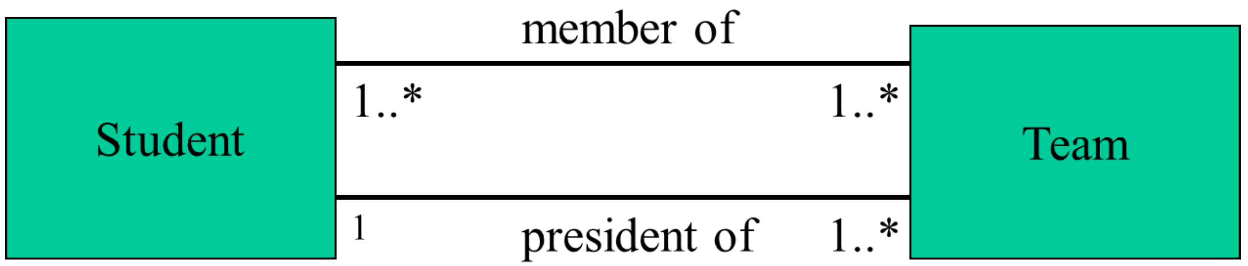
We can also indicate the behavior of an object in an association (*i.e.,* the *role* of an object) using *rolenames.*



We can also name the association.



We can specify dual associations.

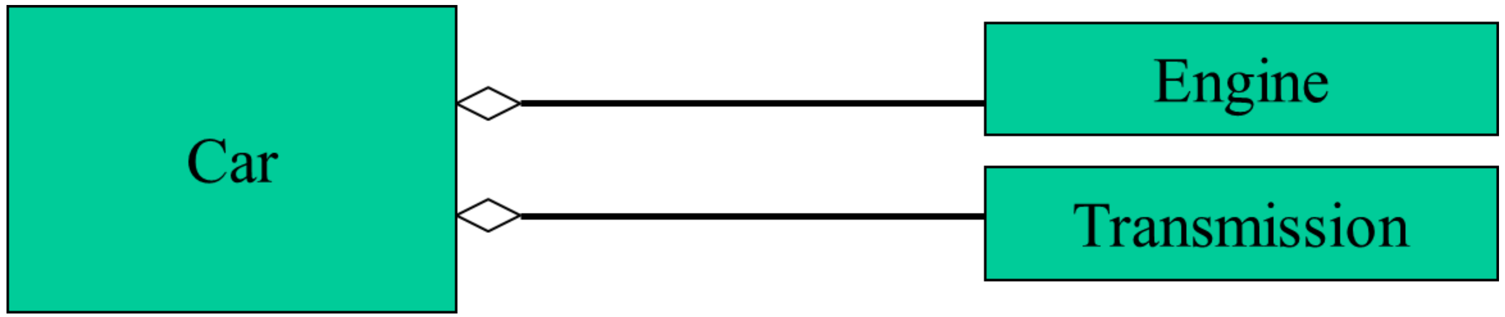


Relationship – Association (Cont'd)

We can model objects that contain other objects by way of special associations called **aggregations** and **compositions**.

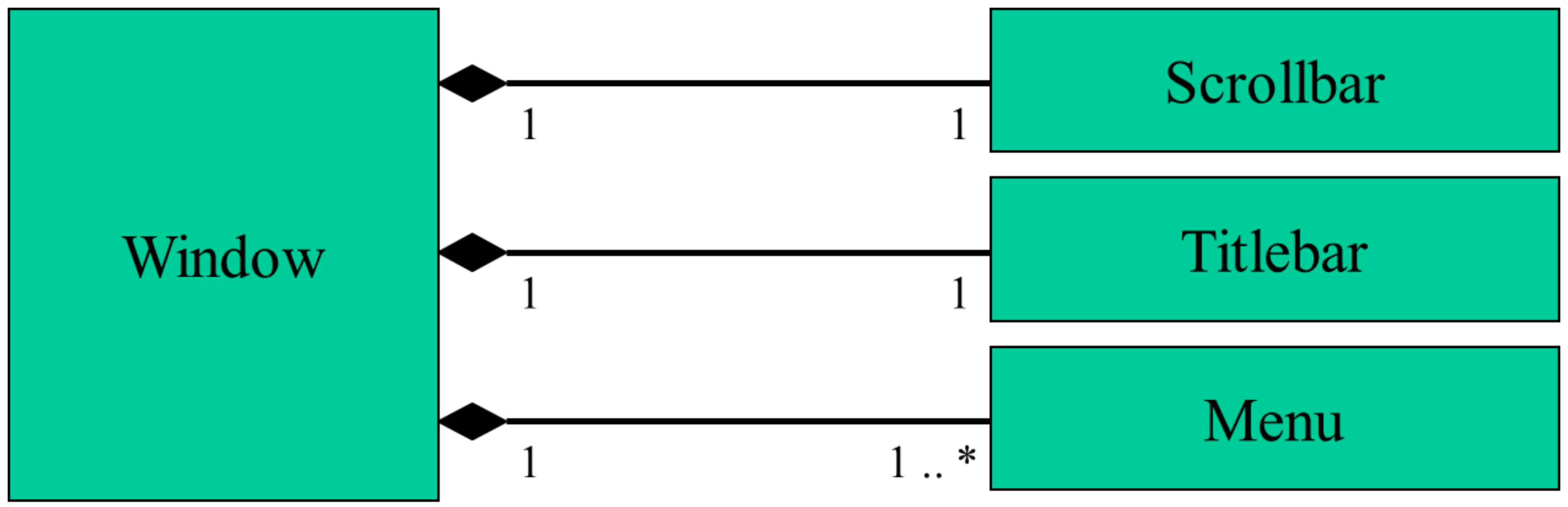
An **aggregation** specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a **hollow-diamond** decoration on the association.

.



Relationship – Association (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.

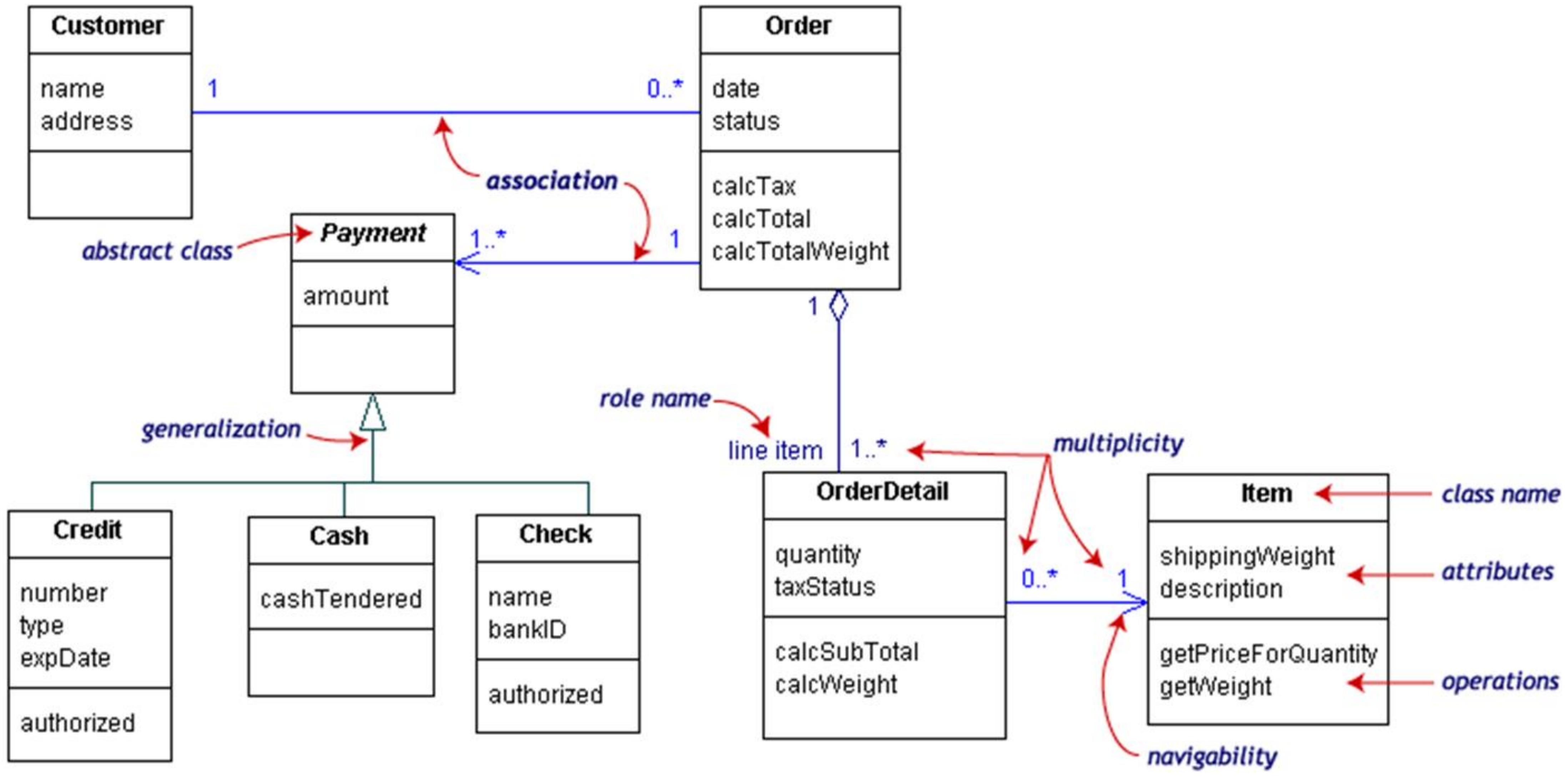


UML MULTIPLICITIES

Links on associations to specify more details about the relationship

Multiplicities	**Meaning**
0..1	zero or one instance. The notation ***n . . m*** indicates ***n*** to ***m*** instances.
0..* *or* *****	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

UML Class Diagram Example

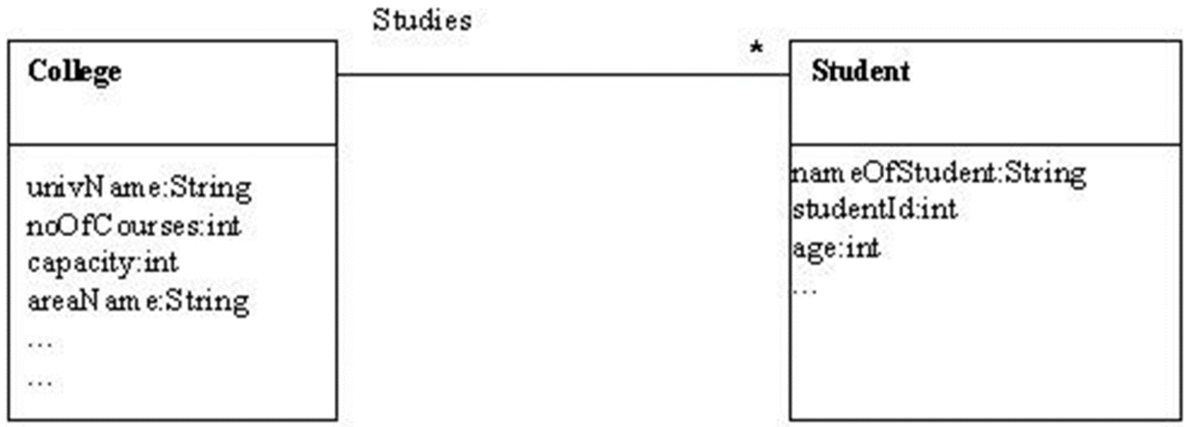


Object Diagram

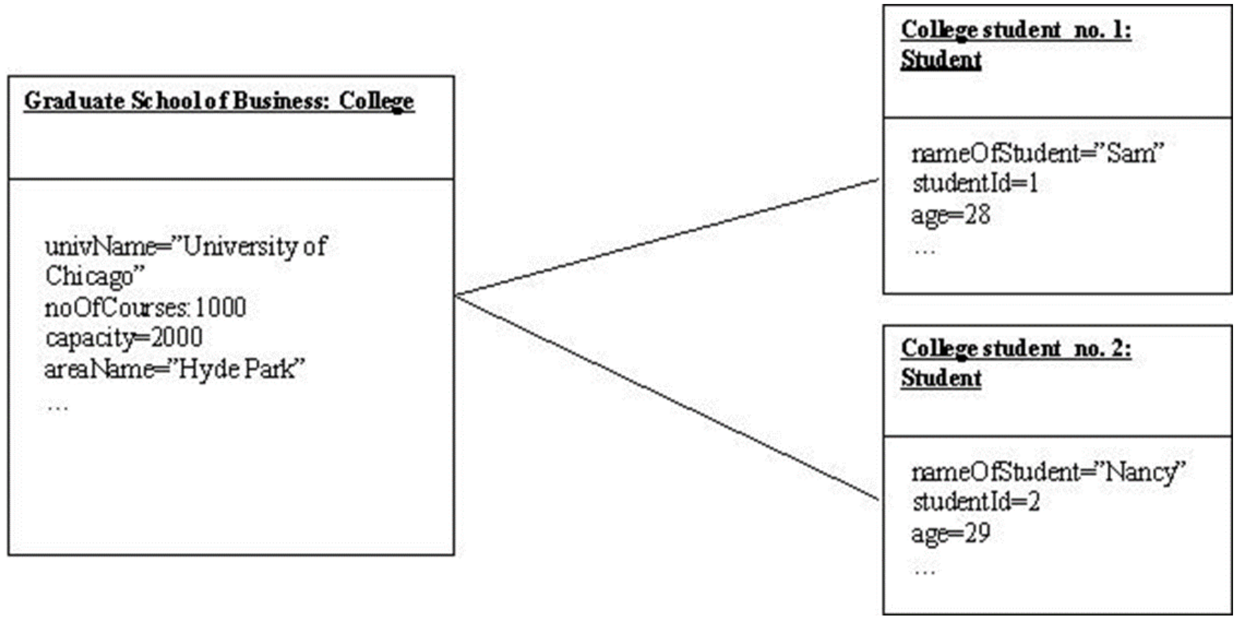
- In a live application classes are not directly used, but instances or objects of these classes are used.
- A pictorial representation of the relationships between these instantiated classes at any point of time (called objects) is called an "**Object diagram**."
- It looks very similar to a class diagram, and uses the similar notations to denote relationships.
- It reflects the picture of how classes interact with each others at runtime. and in the actual system, how the objects created at runtime are related to the classes.
- shows this relation between the instantiated classes and the defined class, and the relation between these objects.

Object Diagram (cont'd)

The minor difference between class diagram and the object diagram is that, the class diagram shows a class with attributes and methods declared. However, in an object diagram, these attributes and method parameters are allocated values.



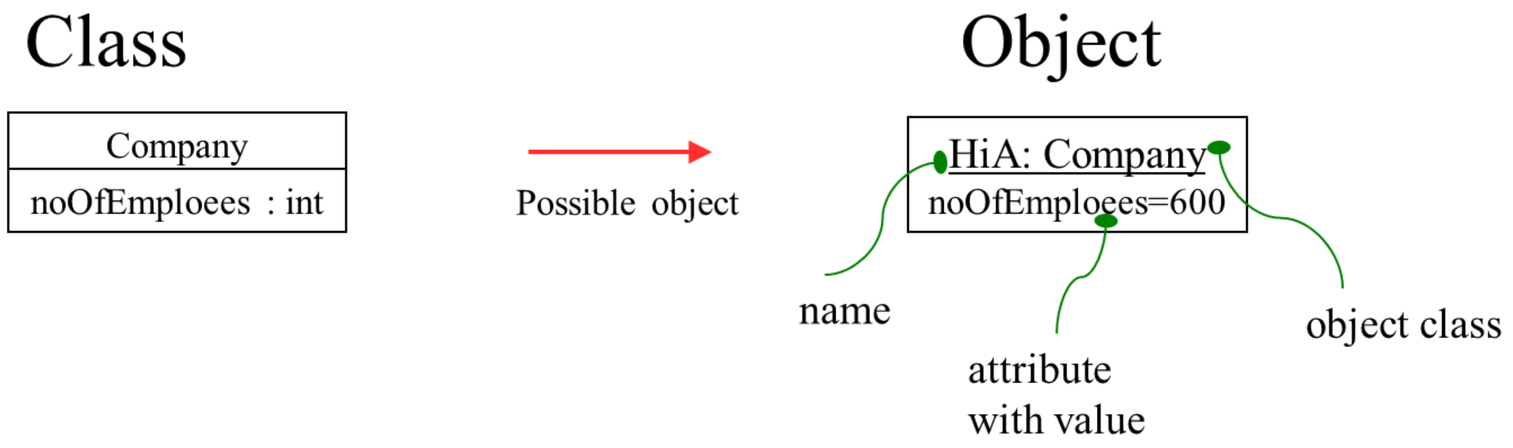
Now, when an application with the class diagram as shown above is run, instances of College and Student class will be created, with values of the attributes initialized. The object diagram for such a scenario will be represented as shown below:



Graduate School of Business: College
univName="University of Chicago"
noOfCourses:1000
capacity=2000
areaName="Hyde Park"
…
College student no. 1: Student
nameOfStudent="Sam"
studentId=1
age=28
…
College student no. 2: Student
nameOfStudent="Nancy"
studentId=2
age=29
…

Object Diagram (cont'd)

The object diagram shows the name of the instantiated object, separated from the class name by a ":", and underlined, to show an instantiation.



When to use object diagrams?

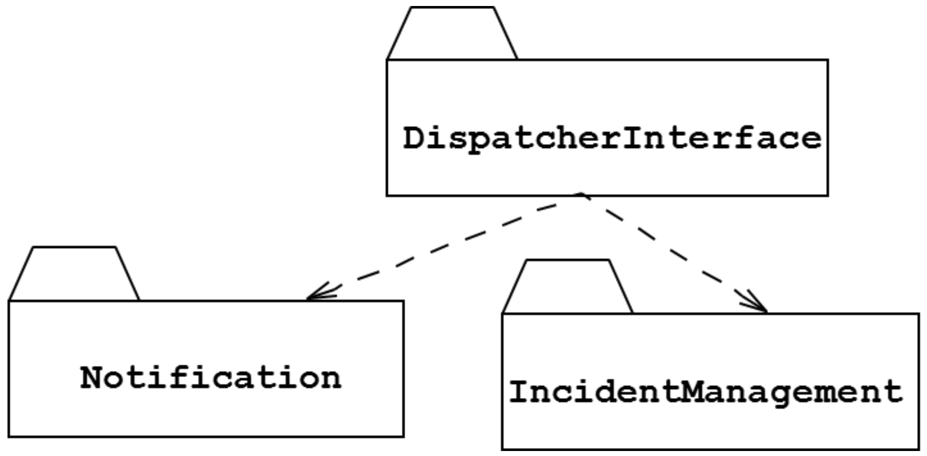
✓ Use the object diagram as a means of debugging the functionality of your system.

✓ Check whether the system has been designed as per the requirements, and behaves how the business functionality needs the system to respond.

Be careful !!
Avoid representing all the objects of your system in an object diagram → complex → unreadable.

PACKAGE DIAGRAM

- To organize complex class diagrams, you can group classes into packages.
- A package is a collection of logically related UML elements
- Notation
  - 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.
  - Packages are the basic grouping construct with which you may organize UML models to increase their readability

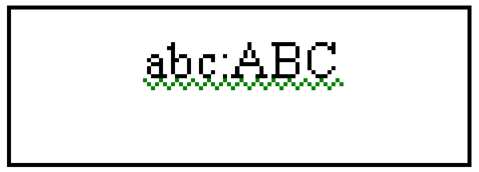


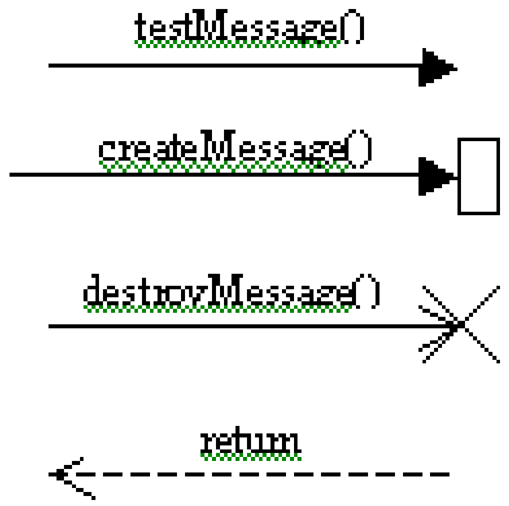
SEQUENCE DIAGRAMS

- ✓ A sequence diagram captures the behavior of a single scenario. The diagram shows a number of example objects and the messages that are passed between these objects within the use case.
- ✓ A Sequence diagram depicts the sequence of actions that occur in a system.
- ✓ A Sequence diagram is two-dimensional in nature. On the horizontal axis, it shows the life of the object that it represents, while on the vertical axis, it shows the sequence of the creation or invocation of these objects.

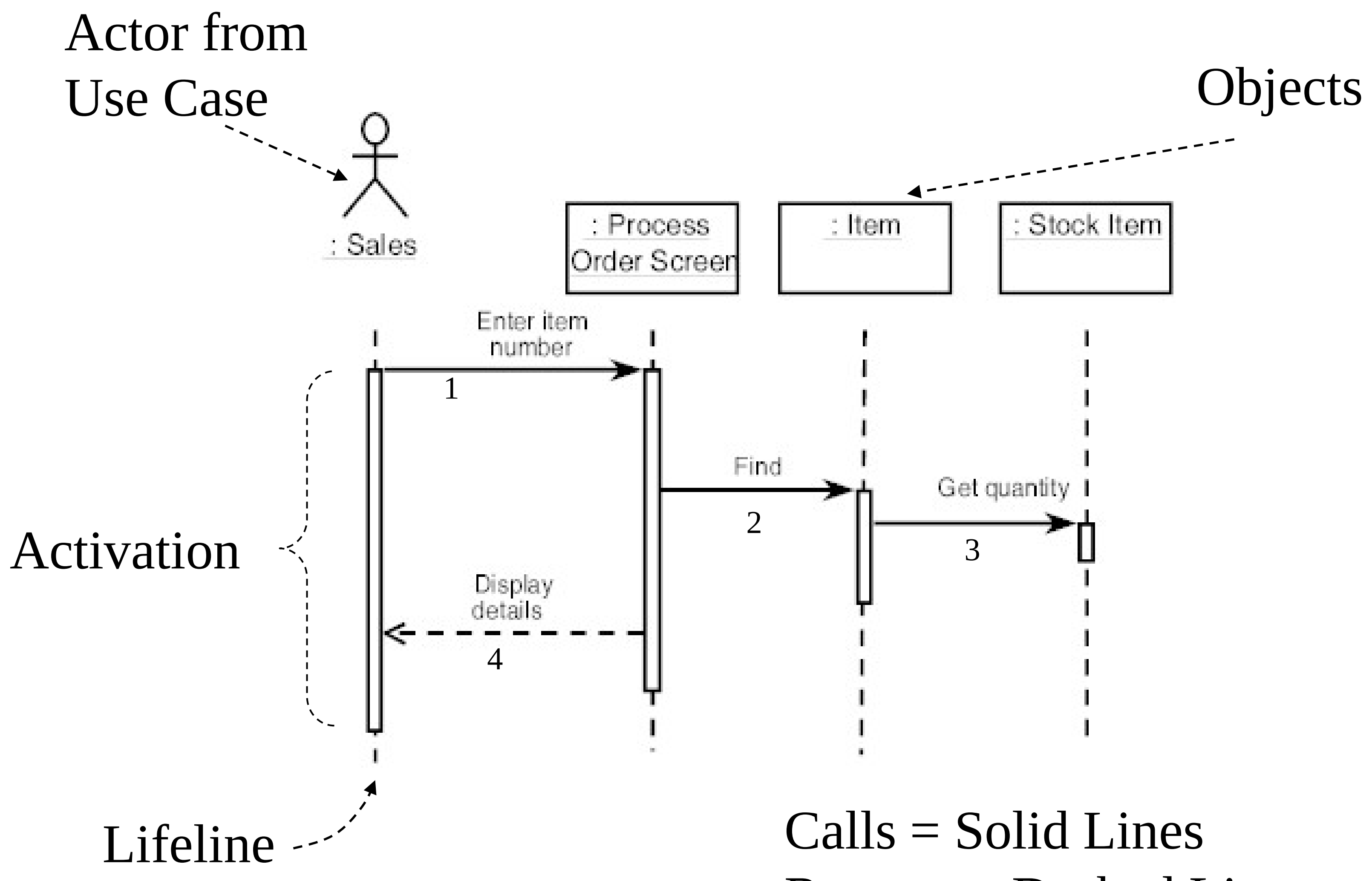
SEQUENCE DIAGRAMS (CONT'D)

Elements of a Sequence Diagram
A Sequence diagram consists of the following behavioral elements:

Object: **The primary element involved in a sequence diagram is an Object. A Sequence diagram consists of sequences of interaction among different objects over a period of time.**	

Message: **The interaction between different objects in a sequence diagram is represented as messages. A messages is represented by a directed arrow.**	 testMessage()
	createMessage()
	destroyMessage()
	return

SEQUENCE DIAGRAM FORMAT

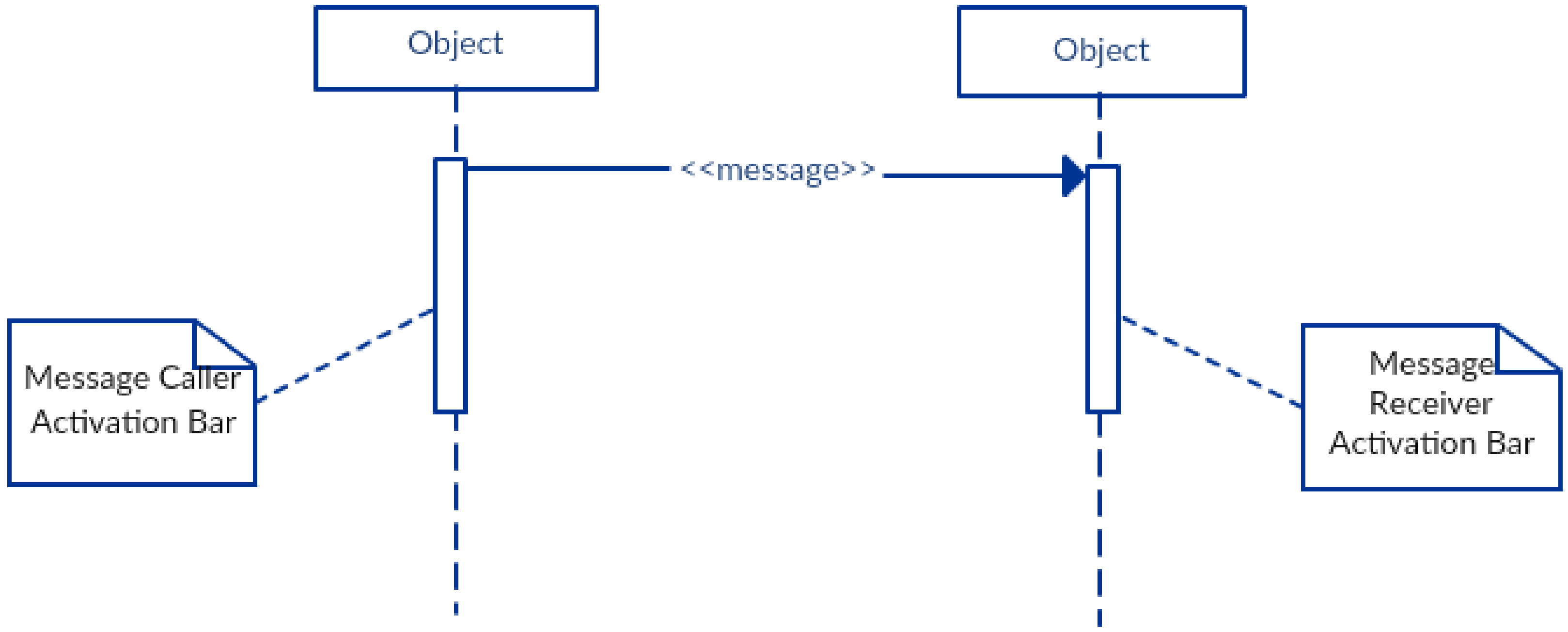


Calls = Solid Lines
Returns = Dashed Lines

SEQUENCE DIAGRAMS (CONT'D)

Activation Bars

Activation bar is the box placed on the lifeline. It is used to indicate that an object is active (or instantiated) during an interaction between two objects. The length of the rectangle indicates the duration of the objects staying active.


Object
Object
<<message>>
Message Caller Activation Bar
Message Receiver Activation Bar

Sequence Diagrams (cont'd)

Message Arrow

An arrow from the Message Caller to the Message Receiver specifies a message in a sequence diagram.   A message can flow in any direction; from left to right, right to left or back to the Message Caller itself.

The message arrow comes with a description, which is known as a message signature, on it. The format for this message signature is below. All parts except the message_name are optional.

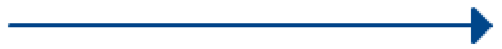
attribute = message_name (arguments): return_type

Sequence Diagrams (cont'd)

Message Arrow

•***Synchronous message***

a synchronous message is used when the sender waits for the receiver to process the message and return before carrying on with another message. The arrowhead used to indicate this type of message is a solid one, like the one below.

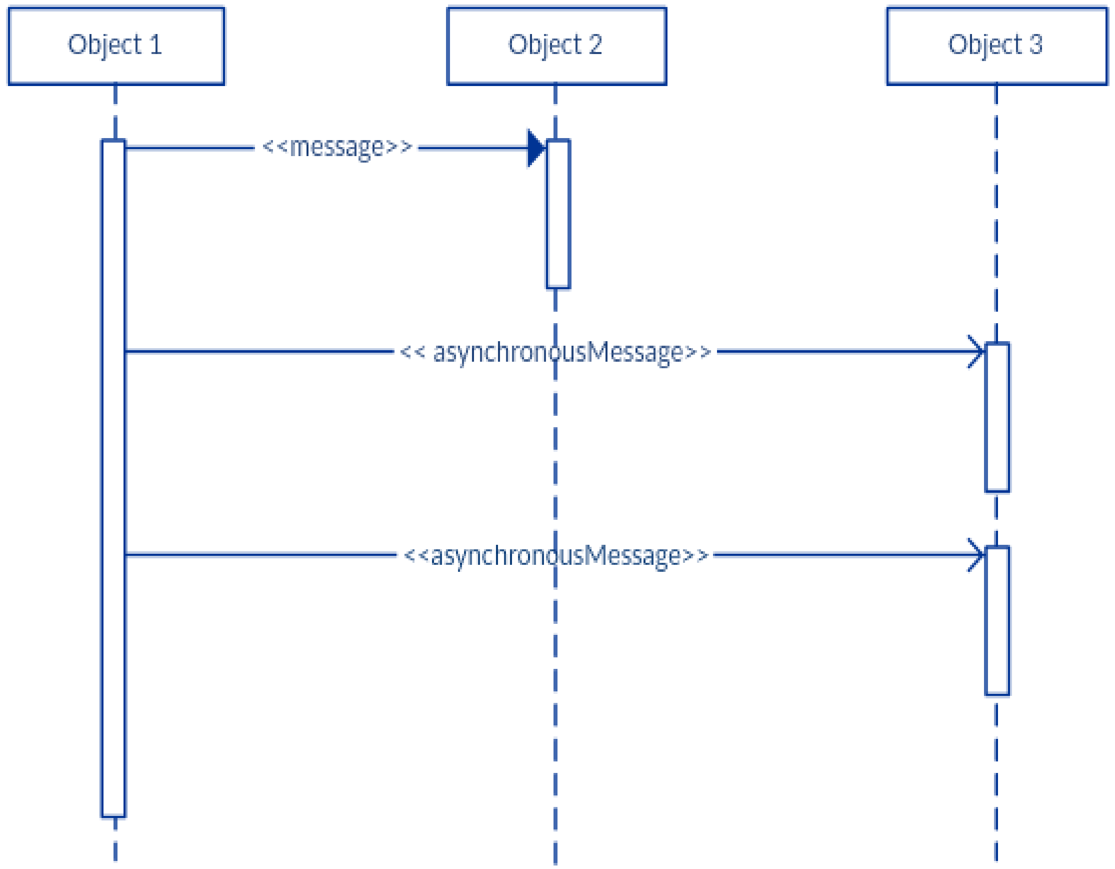


SEQUENCE DIAGRAMS (CONT'D)

Message Arrow

- ***Asynchronous message***

  An asynchronous message is used when the message caller does not wait for the receiver to process the message and return before sending other messages to other objects within the system. The arrowhead used to show this type of message is a line arrow like shown in the example below.

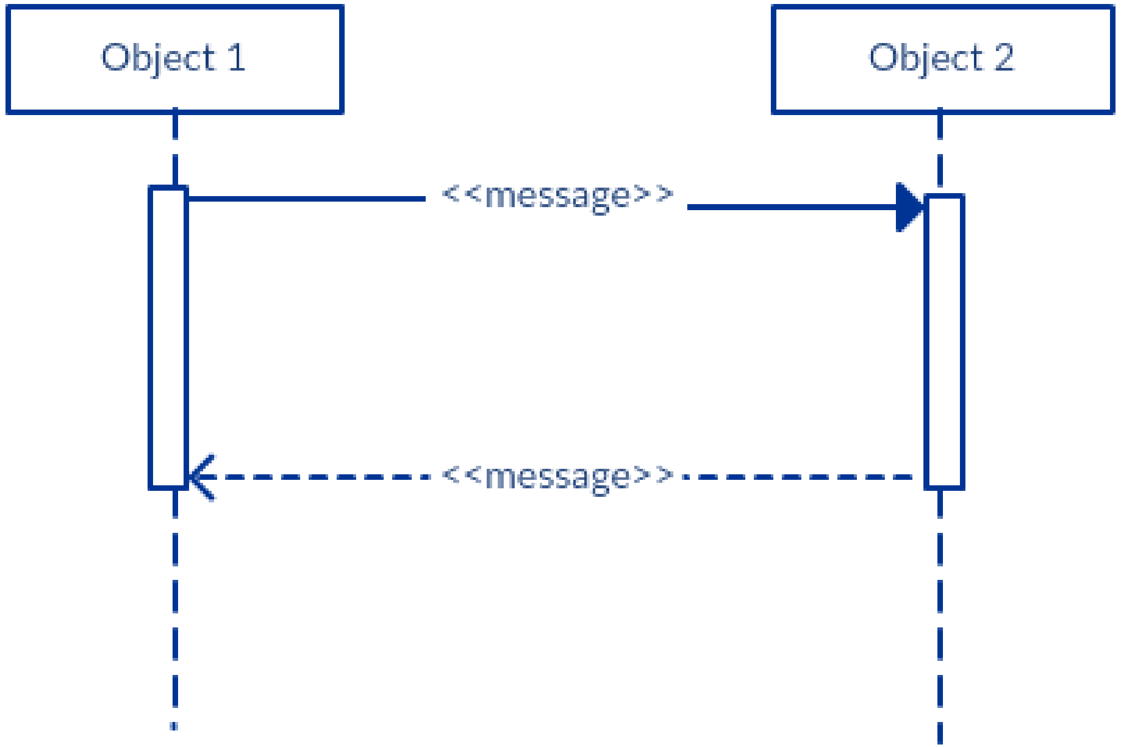


Sequence Diagrams (cont’d)

Message Arrow

- ***Return message***

  A return message is used to indicate that the message receiver is done processing the message and is returning control over to the message caller.

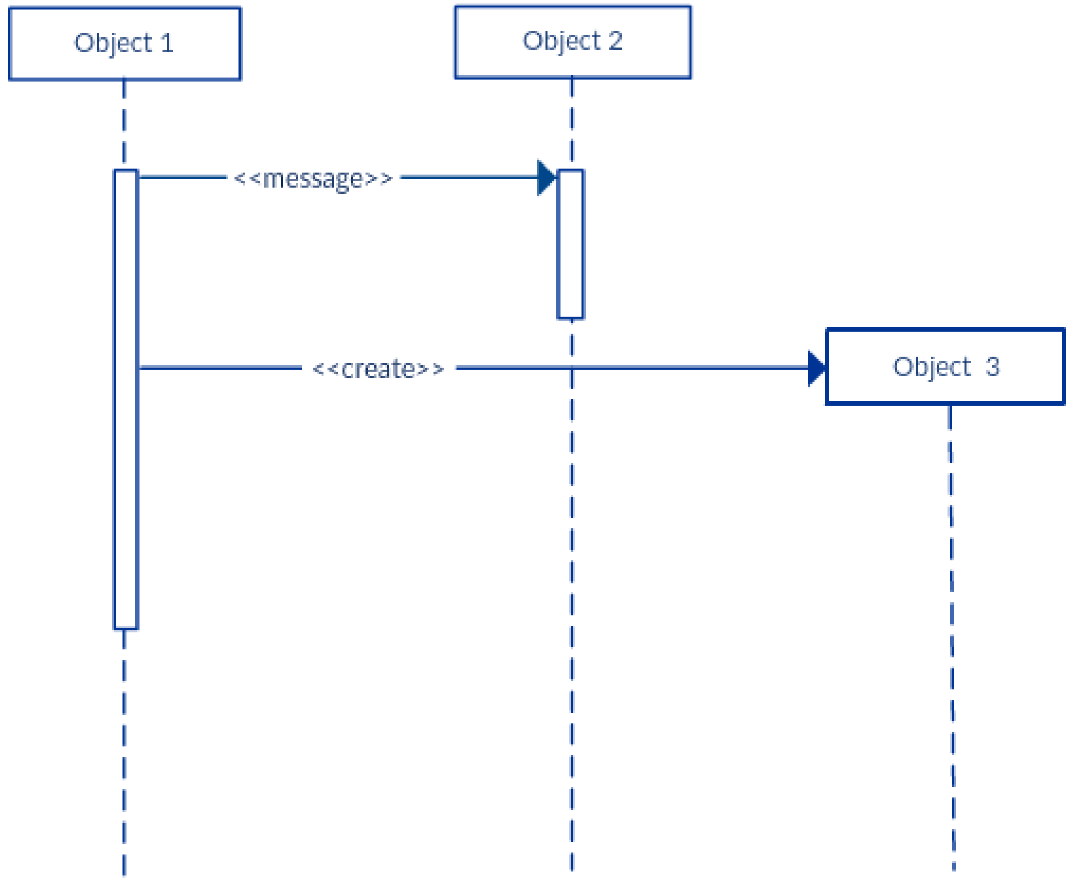


Sequence Diagrams (cont'd)

Message Arrow

- ***Participant creation message***

  Objects do not necessarily live for the entire duration of the sequence of events. Objects or participants can be created according to the message that is being sent.

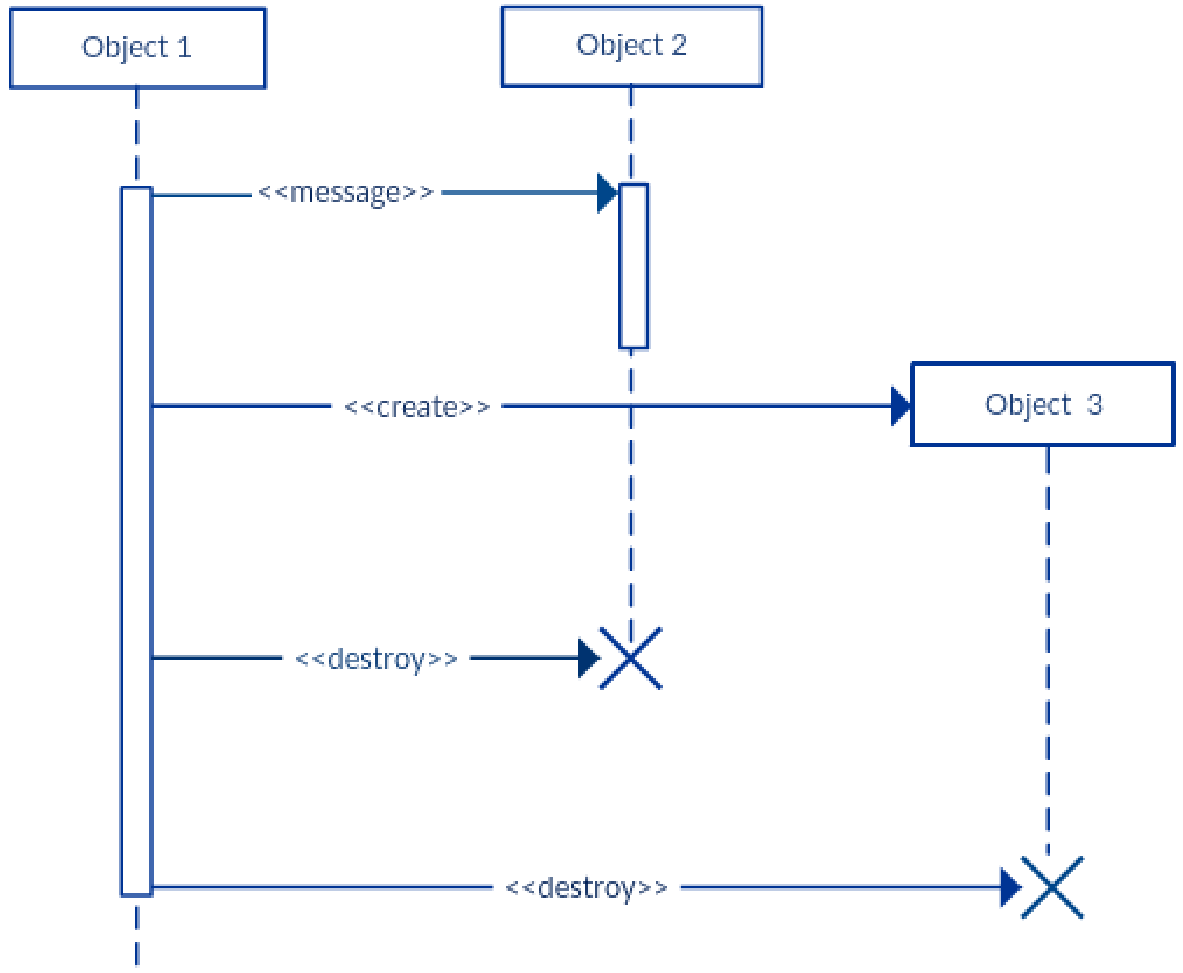


Sequence Diagrams (cont'd)

Message Arrow

- ***Participant destruction message***

  Likewise, participants when no longer needed can also be deleted from a sequence diagram. This is done by adding an 'X' at the end of the lifeline of the said participant.

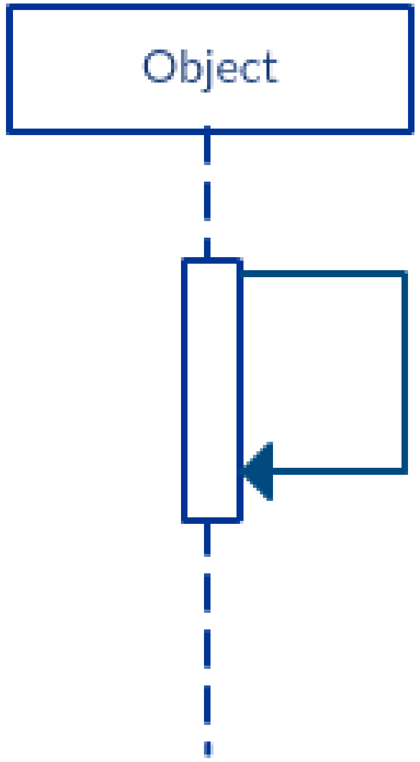


Sequence Diagrams (cont'd)

Message Arrow

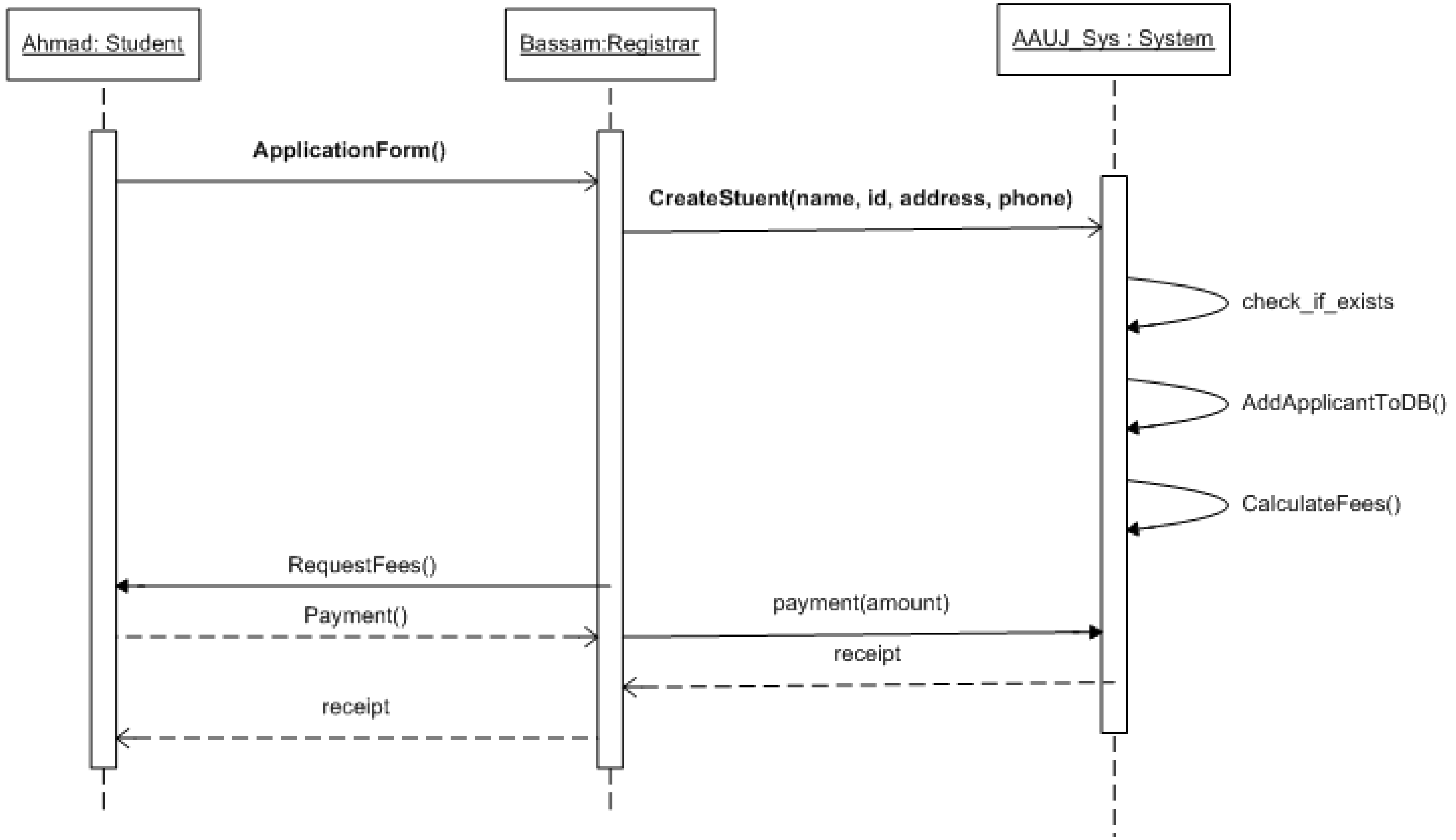
- ***Reflexive message***

  When an object sends a message to itself, it is called a reflexive message. It is indicated with a message arrow that starts and ends at the same lifeline as shown in the example.



SEQUENCE DIAGRAMS (CONT'D)

Sequence Diagram that for the Enroll in University Use Case



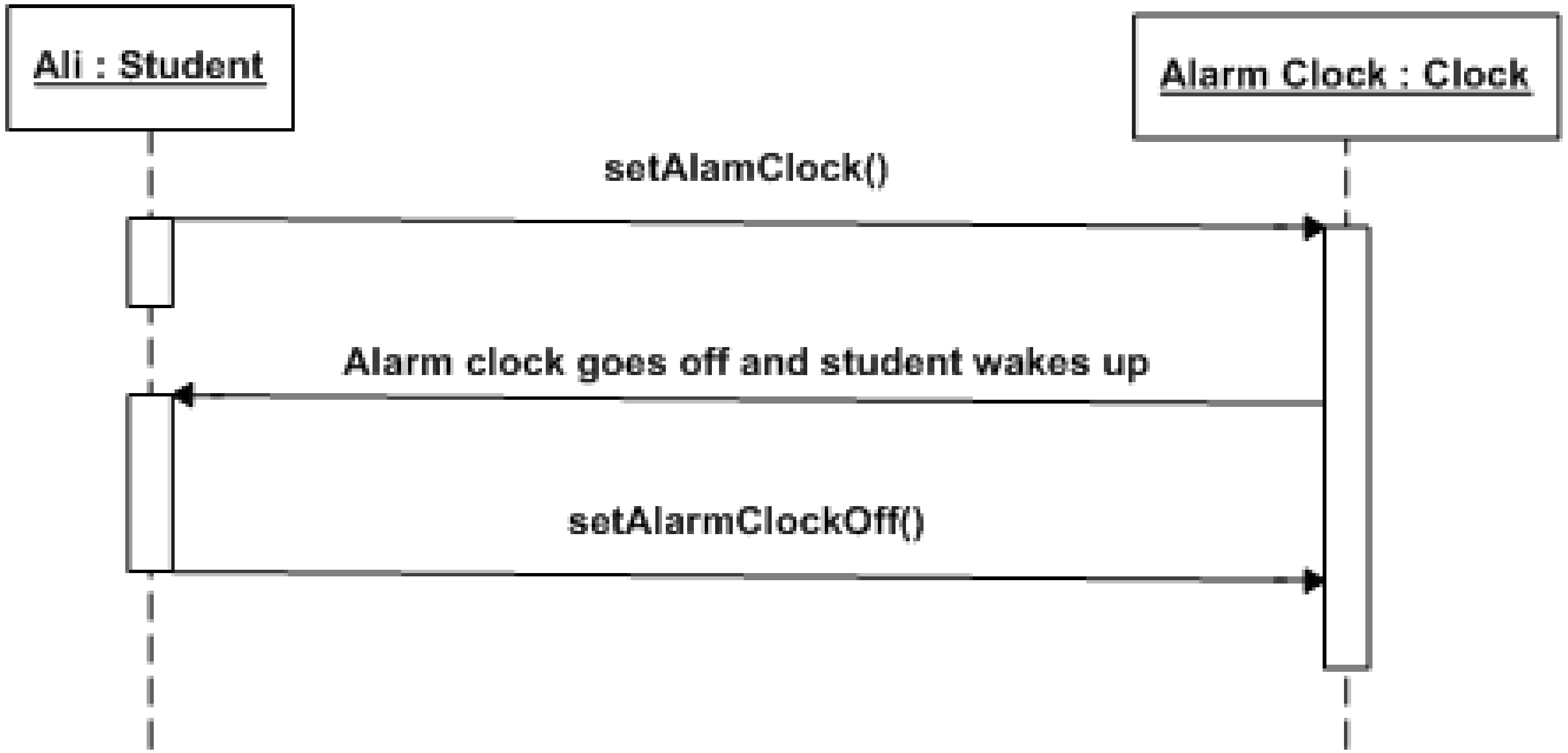
SEQUENCE DIAGRAMS (CONT'D)

Draw the sequence diagram for the following scenario

Student sets electric alarm clock to wake up time.

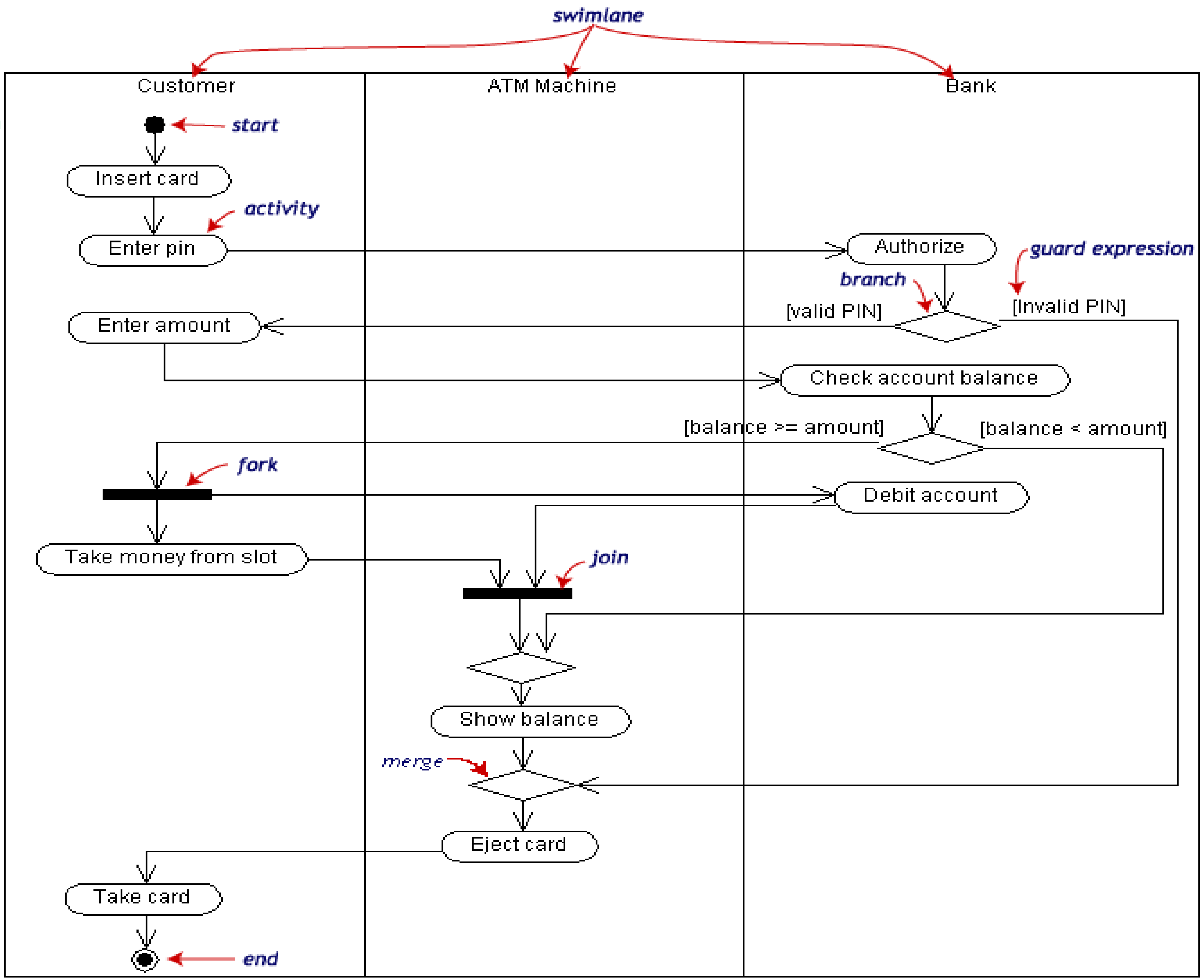
Alarm clock goes off and student wakes up.

Student sets alarm clock to off.


Ali : Student
Alarm Clock : Clock
setAlamClock()
Alarm clock goes off and student wakes up
setAlarmClockOff()

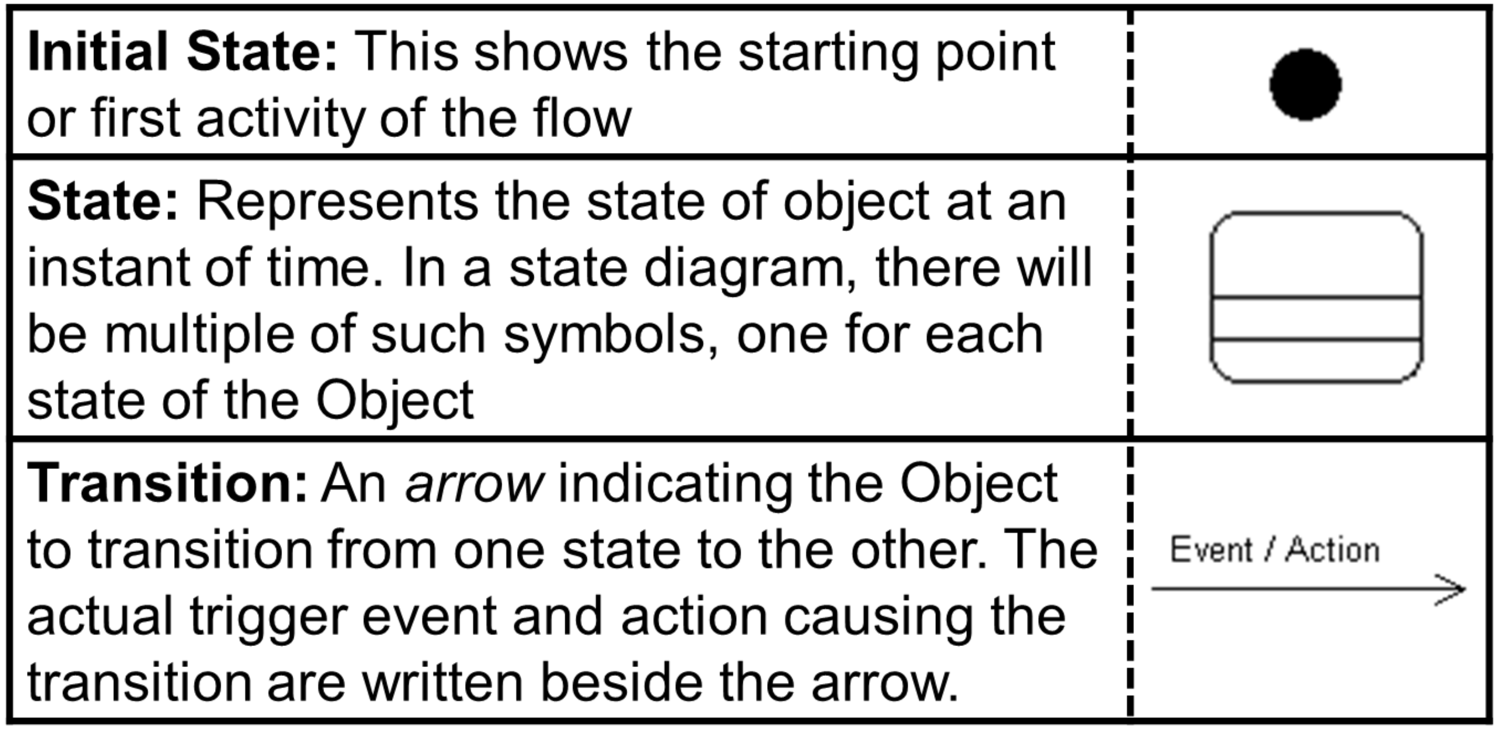
Activity Diagrams

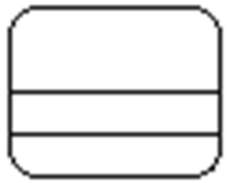
- Fancy flowchart
  - Displays the flow of activities involved in a single process
  - States
    - Describe what is being processed
    - Indicated by boxes with rounded corners
  - Swim lanes
    - Indicates which object is responsible for what activity
  - Branch
    - Transition that branch
    - Indicated by a diamond
  - Fork
    - Transition forking into parallel activities
    - Indicated by solid bars
  - Start and End


swimlane
Customer
ATM Machine
Bank
start
Insert card
activity
Enter pin
Authorize
guard expression
branch
[valid PIN]
[Invalid PIN]
Enter amount
Check account balance
[balance >= amount]
[balance < amount]
fork
Debit account
Take money from slot
join
Show balance
merge
Eject card
Take card
end

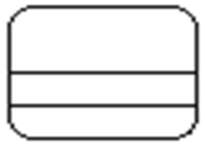
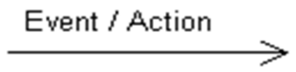
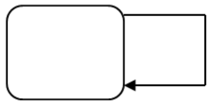
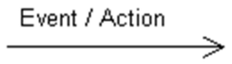
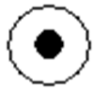
STATE DIAGRAM

- ✓ State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur.
- ✓ It is important to note that having a State diagram for your system is not a mandatory, but must be defined only on a need basis (to understand the behavior of the object through the entire system)



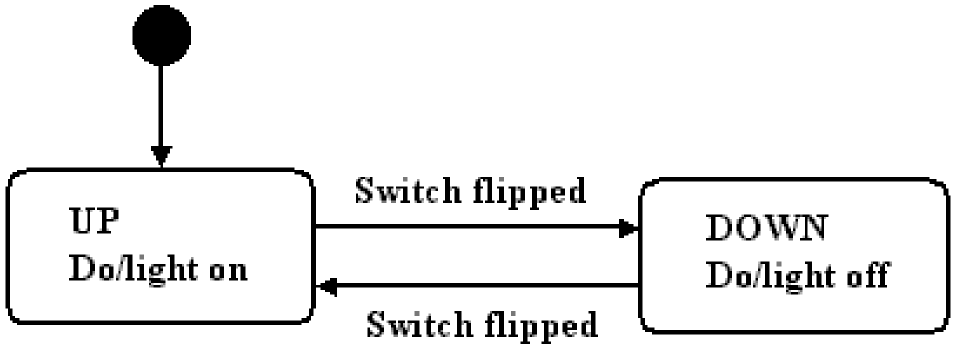
Initial State: This shows the starting point or first activity of the flow	●
State: Represents the state of object at an instant of time. In a state diagram, there will be multiple of such symbols, one for each state of the Object	
Transition: An *arrow* indicating the Object to transition from one state to the other. The actual trigger event and action causing the transition are written beside the arrow.	$\xrightarrow{\text{Event / Action}}$

Elements of a State diagram

Initial State: This shows the starting point or first activity of the flow	●
State: Represents the state of object at an instant of time. In a state diagram, there will be multiple of such symbols, one for each state of the Object	
Transition: An *arrow* indicating the Object to transition from one state to the other. The actual trigger event and action causing the transition are written beside the arrow.	
Self Transitions: Sometimes an object is required to perform some action when it recognizes an event, but it ends up in the same state it started in	
Event and Action: A trigger that causes a transition to occur is called as an event or action.	
Final State: The end of the state diagram is shown by a bull's eye symbol, also called a final state.	

Example: ordinary two-position light switch.

A light switch will have two states: up and down. (We could call them "on" and "off" if we liked.) In a UML state diagram, each state is represented by a rounded rectangle.



STATE DIAGRAM (CONT'D)

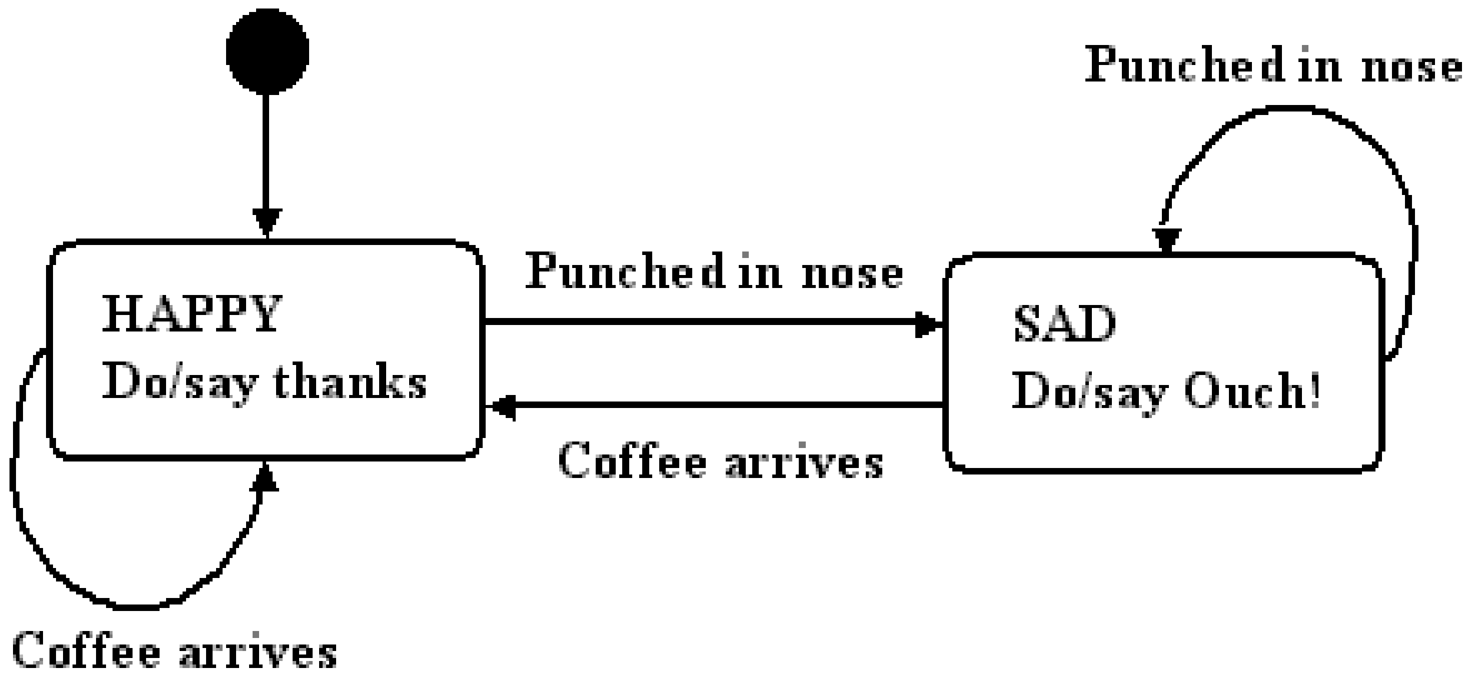
Example: simplistic Teaching Assistant (TA)

TA only has two states:

Happy when getting coffee.

Sad when getting punched in the nose.

Suppose that TA is basically cheerful and starts out happy

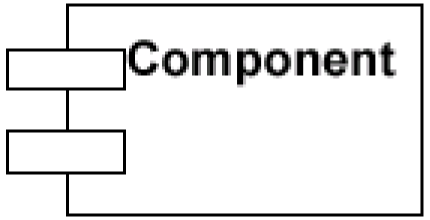


Component Diagram

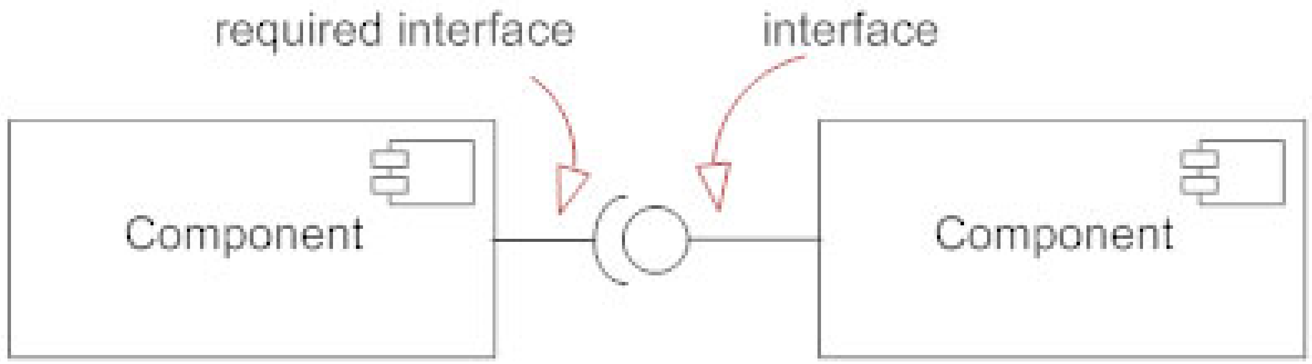
- A component diagram displays the structural relationship of components of a software system.
- These are mostly used when working with complex systems with many components.
- Components communicate with each other using interfaces. The interfaces are linked using connectors.

COMPONENT DIAGRAM (CONT'D)

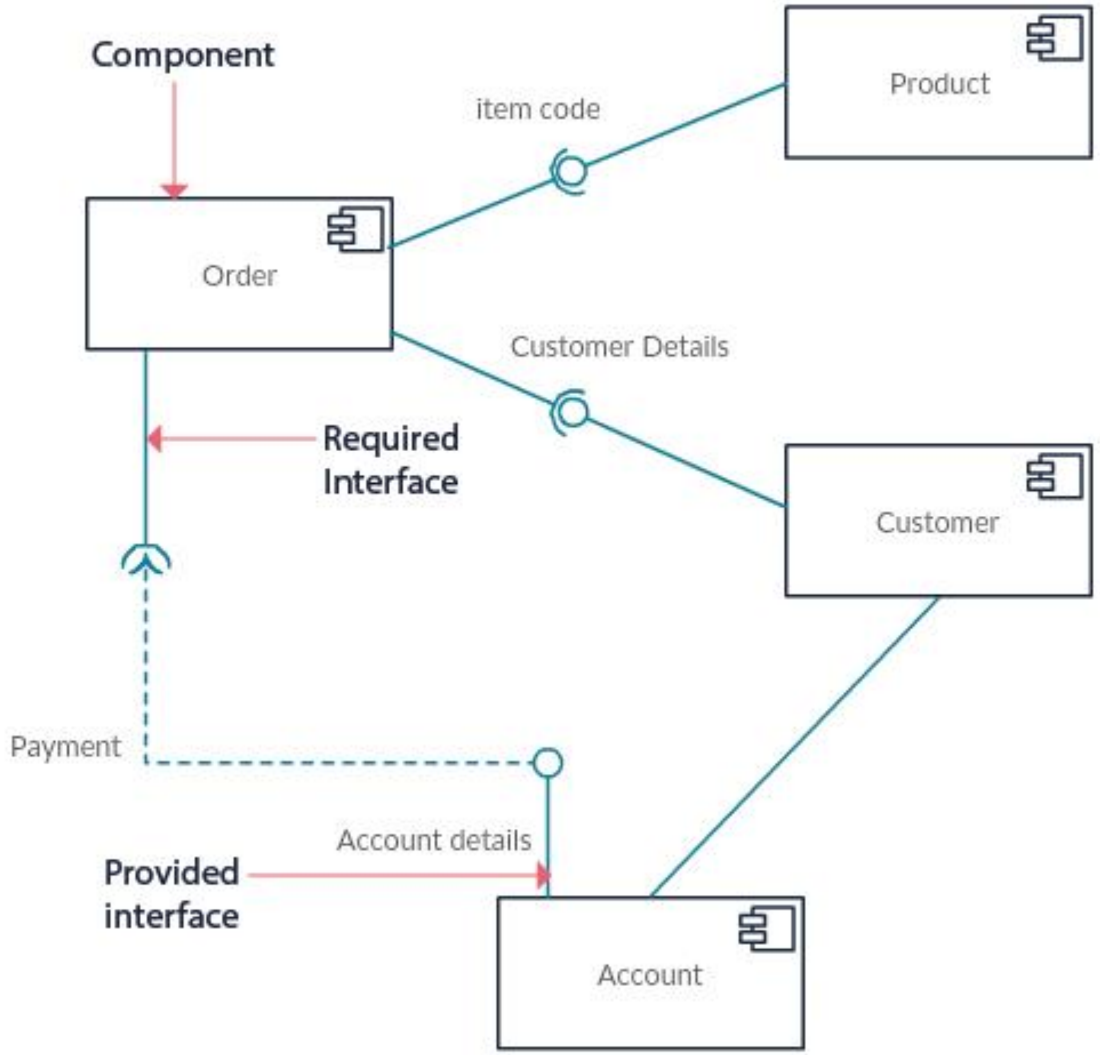
- ✓ Components are shown as rectangles with two tabs at the upper left



- ✓ Dashed arrows indicate dependencies
- ✓ Circle and solid line indicates an interface to the component



COMPONENT DIAGRAM (CONT'D)



ordering system

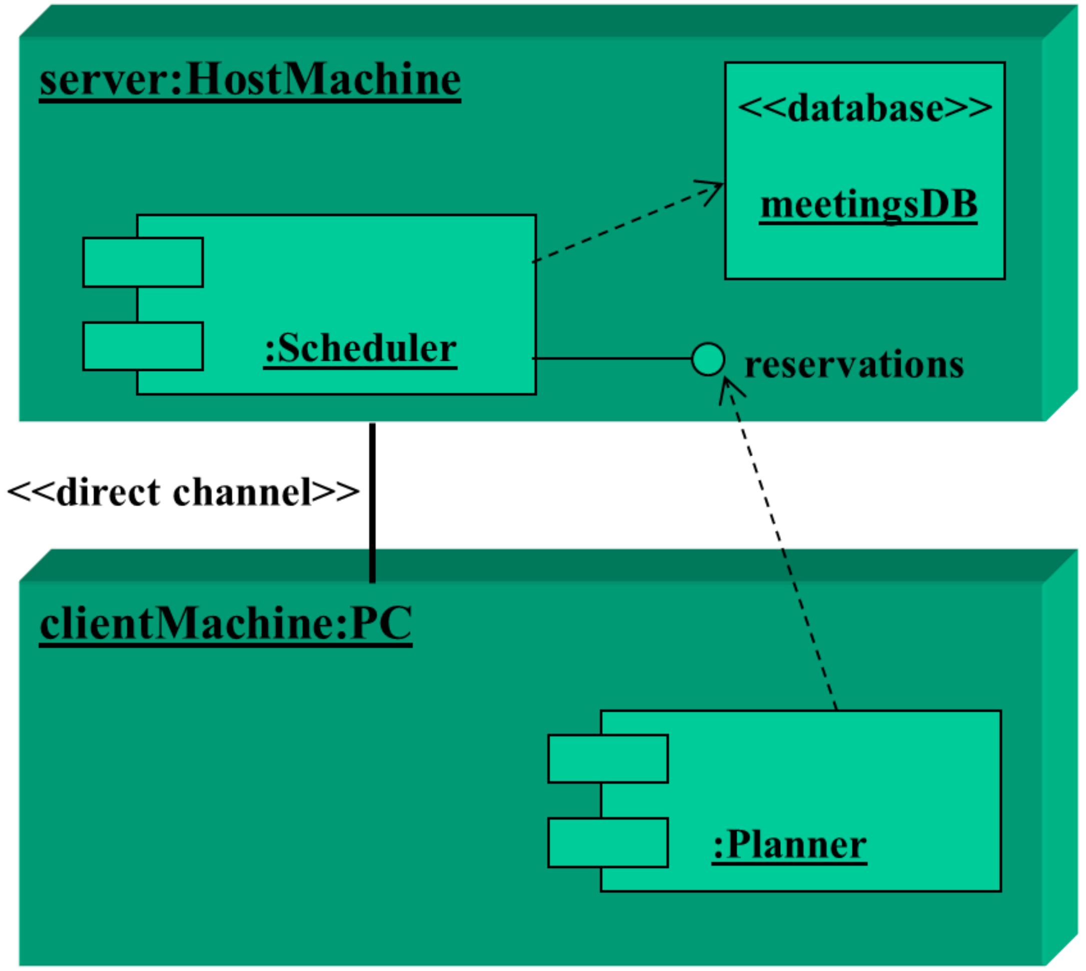
Deployment Diagram

- ✓ Shows the physical architecture of the hardware and software of the deployed system
- ✓ Nodes
  - ➢ Typically contain components or packages
  - ➢ Usually some kind of computational unit; e.g. machine or device (physical or logical)
- ✓ Physical relationships among software and hardware in a delivered systems
  - ➢ Explains how a system interacts with the external environment

Deployment diagrams are useful when your software solution is deployed across multiple machines with each having a unique configuration.

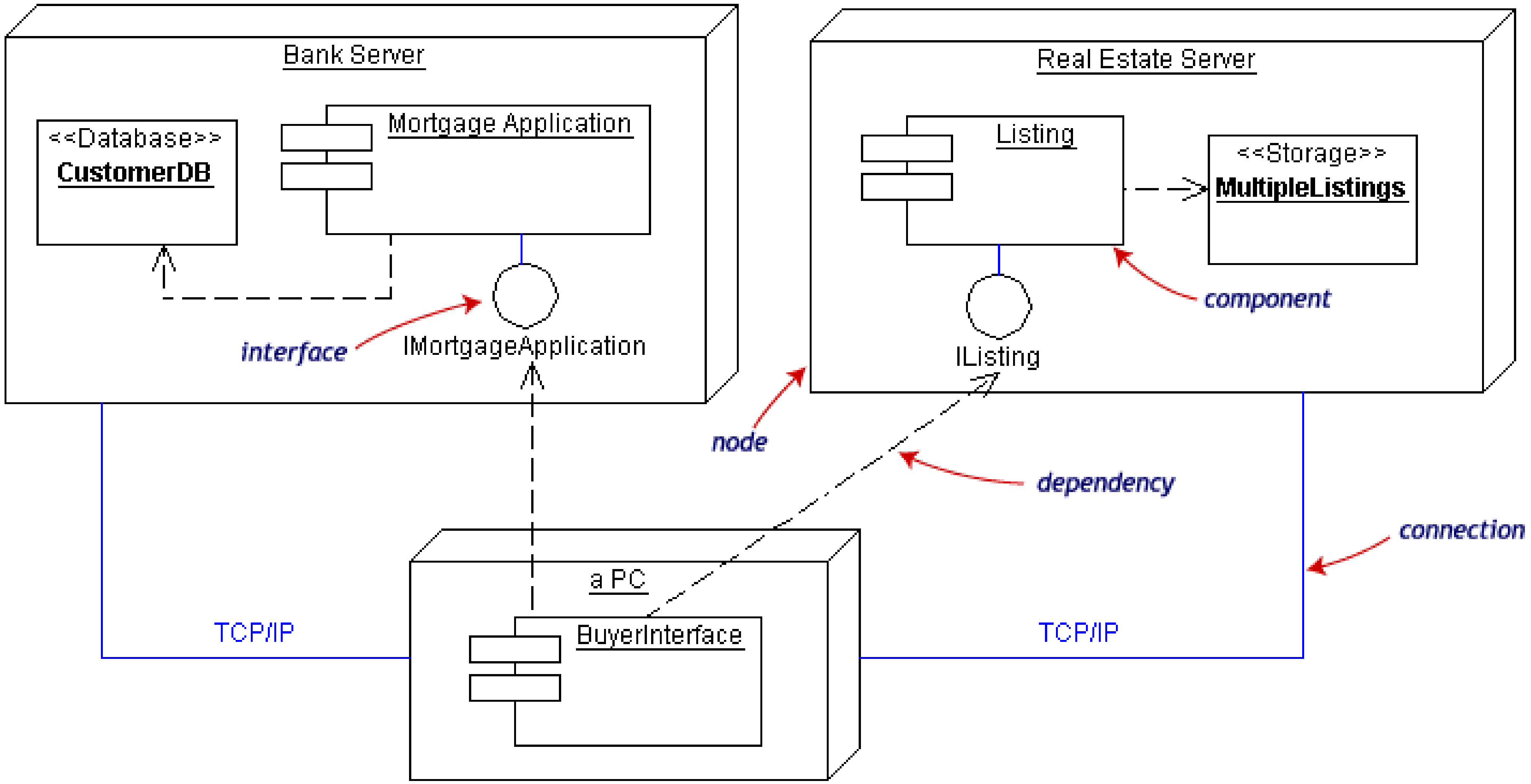
- ✓ A **deployment diagram** in the Unified Modeling Language models the *physical* deployment of artifacts on nodes. To describe a web site, for example, a deployment diagram would show what hardware components ("nodes") exist (e.g., a web server, an application server, and a database server), what software components ("artifacts") run on each node (e.g., web application, database), and how the different pieces are connected (e.g. JDBC, REST).

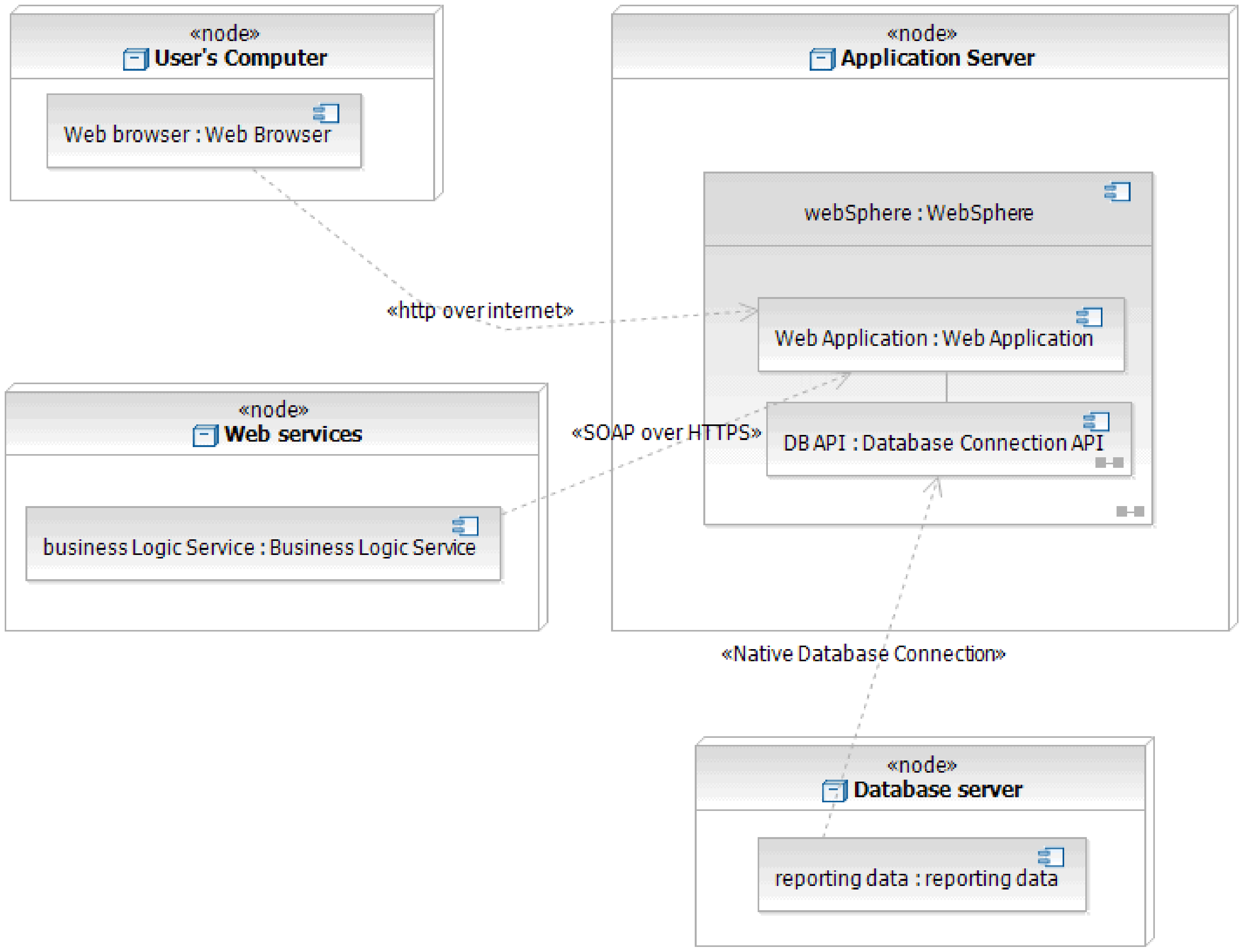
- ✓ The nodes appear as boxes, and the artifacts allocated to each node appear as rectangles within the boxes. Nodes may have subnodes, which appear as nested boxes. A single node in a deployment diagram may conceptually represent multiple physical nodes, such as a cluster of database servers.
- ✓ There are two types of Nodes:
  - ✓ Device Node
  - ✓ Execution Environment Node



Deployment diagram of a client-server system.

DEPLOYMENT EXAMPLE




«node»
User's Computer
Web browser : Web Browser
«node»
Application Server
webSphere : WebSphere
Web Application : Web Application
DB API : Database Connection API
«http over internet»
«SOAP over HTTPS»
«node»
Web services
business Logic Service : Business Logic Service
«Native Database Connection»
«node»
Database server
reporting data : reporting data