Tribhuvan University
Institute of Engineering
Pulchowk Campus
Department of Electronics and Computer Engineering

OBJECT ORIENTED SOFTWARE ENGINEERING

Chapter Three

Structural behavioral and architectural modeling

by
Santosh Giri
Lecturer, IOE, Pulchowk Campus.

Conceptual model of UML

Conceptual model of UML

- To understand the UML, we need to form a conceptual model of the language, and this requires learning three major elements:
  - ✓ The UML's basic building blocks
  - ✓ The rules that dictate how those building blocks may be put together and
  - ✓ Some common mechanisms that apply throughout the UML.

Basic Building Blocks of the UML

The vocabulary of the UML encompasses three kinds of building blocks:

1. Things

2. Relationships

3. Diagrams

Note:

Things *are the abstractions that are first-class citizens in a model*

Relationships *tie these things together*

Diagrams *group interesting collections of things.*

Things in the UML

There are four kinds of things in the UML:

1. ***Structural Things***

- These are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical.
- Collectively, the structural things are called classifiers.
- *Class*, *Interface*, *collaboration*, *use case*, *component*, *node* etc. are example of Structural things in UML.

2. Behavioral Things

- These are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space.
- In all, there are three primary kinds of behavioral things : *interaction*, *state machine and Activity*.

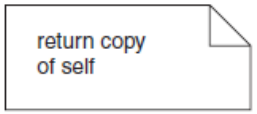
Things in the UML

3. Grouping Things

- These are the organizational parts of UML models.
- These are the boxes into which a model can be decomposed.
- There is one primary kind of grouping thing called *packages*.

4. Annotational Things

- These are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of annotational thing, called a *note*.
- A note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.
- Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment, as in Figure below



Relationship in the UML

Relationships in the UML There are four kinds of relationships in the UML:

1. *Dependency*
2. *Association*
3. *Generalization*
4. *Realization*

Details on later slides

Diagrams in the UML

- A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and paths (relationships).
- We draw diagrams to visualize a system from different perspectives, so a diagram is a projection into a system. The UML includes thirteen kinds of diagrams:
  1. Class diagram
  2. Object diagram
  3. Component diagram
  4. Composite structure diagram
  5. Use case diagram
  6. Sequence diagram
  7. Communication diagram
  8. State diagram
  9. Activity diagram
  10. Deployment diagram
  11. Package diagram
  12. Timing diagram
  13. Interaction overview diagram

UML

Use Case Diagram

UML Use Case Diagram

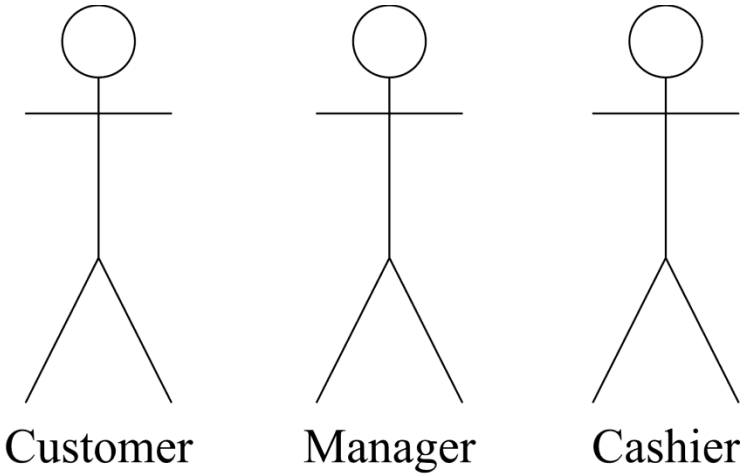
- Use case is used to model the system or subsystem of an application.
- A single use case diagram captures a particular functionality of a system.
- The purpose of use case diagrams can be as follows:
  - Used to gather requirements of a system
  - Used to get an outside view of a system
  - Identify external factors influencing the system
  - Show the interaction among the requirements also called use cases.

- To draw a use case diagram we should have following items identified or components of use case diagrams are:
  - Actor
  - Use case
  - System boundary
  - Relationship

Use Case Diagram

Actor

- An actor is someone or something that must interact with the system under development.
- An UML notation of Actor is represented as:



- Actors are not part of the system they represents anyone or anything that must interact with the system.
- An single actor may perform more than one use cases (functionality)

Use Case Diagram

- An actor may be:
  - Input information to the system
  - Receive information from the system
  - Input to and out from the system.
- How do we find the actors?

  Ask the following questions:
  - Who uses the system?
  - Who install the system?
  - Who starts up the system?
  - What other system use this system?
  - Who gets the information to the system?

Note: *An actor is always external to the system.*

Use Case Diagram

Categories of Actor:

- **Principle**

  who uses the main system functions

- **Secondary**

  who takes care of administration and maintenance

- **External h/w**

  The h/w devices which are part of application domain and must be used

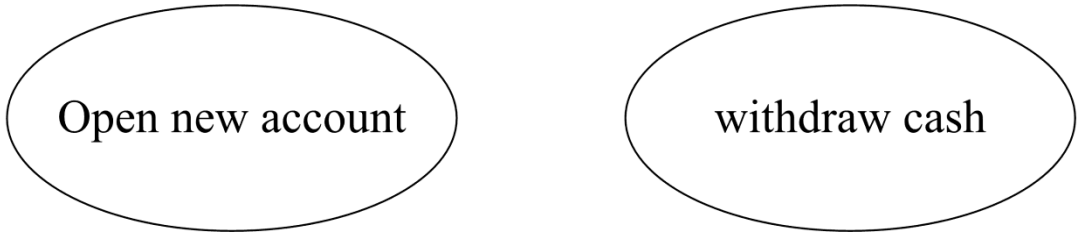
- **Other system**

  The other system with which the system must interact.

Use Case Diagram

Use cases:

- Use cases represents functionality of a system
- which are the specific roles played by the actors within and around the system



How do we find the use cases?

- What functions will the actor want from the system?
- Does the system store information? If yes then which actors will create, read, update or delete that information?
- Does the system need to notify an actor about changes in its internal state?

Use Case Diagram

Generic format for documenting an use case

Pre condition:	if any
Use case:	name of the use case
Actors:	list of actors, indicating who initiates the use case
Purpose:	intention of the use case.
Overview:	Description.
Type:	primary/secondary.
Post condition:	if any

Use Case Diagram

Example:

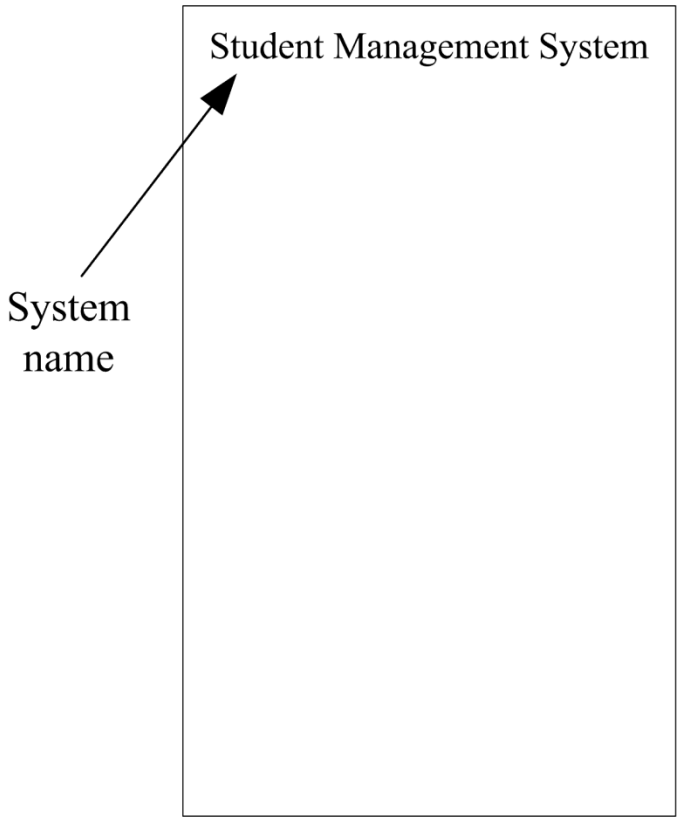
Description of opening a new account in the bank

Use case	Open new account
Actors	Customer, Cashier, Manager
Purpose	Like to have new saving account.
Description	A customer arrives in the bank to open the new account. Customer requests for the new account form, fill the form and submits, along with the minimal deposit. At the end of complete successful process customer receives the passbook.

Use Case Diagram

System boundary:

- It helps to identify what is an external verse internal.
- External environment is represented only by actors.
- Represented as a rectangle.



UML Use Case Diagram

Relationship:

Relationship between use case and actor

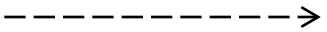
- © communicates

Relationship between two use cases

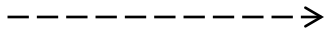
- © Include
- © extend

Notation used to show the include and extend relationship

«include»



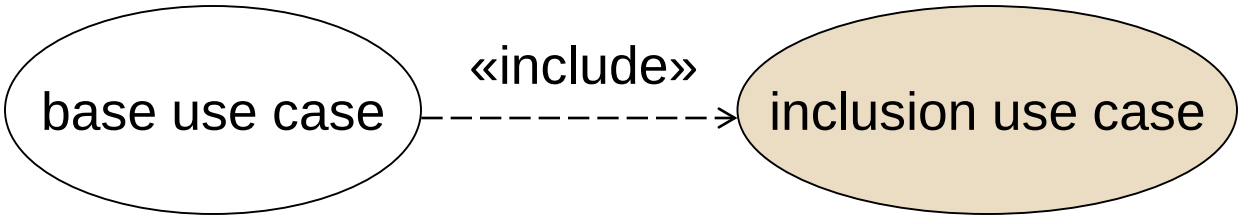
«extend»



Use Case Diagram

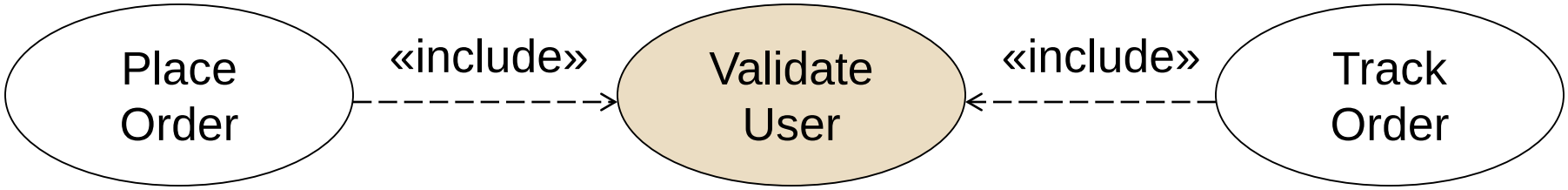
«include» Relationship:

- A use cases may contain the functionality of another use case.
- It is used to show how the system can use a pre existing components
- An include relationship is a relationship in which one use case (the base use case) includes or uses the functionality of another use case (the inclusion use case).
- Represented as a dashed line with an open arrow pointing from the base use case to the inclusion use case. The keyword **«include»** is attached to the connector.



Use Case Diagram

«include» relationship example:

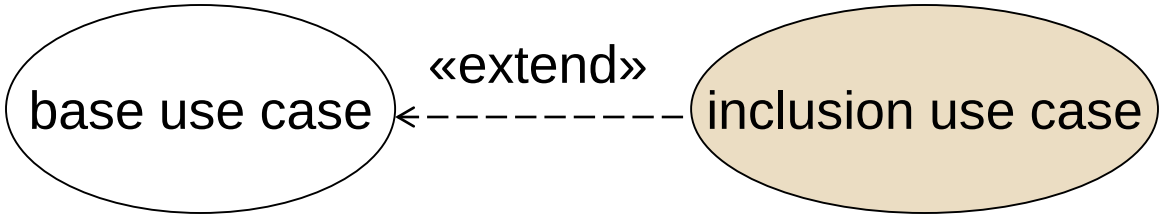


- The following figure illustrates an restaurant order management system that provides customers with the option of placing orders as well as tracking orders. This behavior is modeled with two base use cases called *PlaceOrder* & *TrackOrder* that has an inclusion use case called *ValidateUser*. The *ValidateUser* use case is a separate inclusion use case because it contains behaviors that several other use cases in the system use. That include relationship points from the *PlaceOrder & TrackOrder* use cases to the *ValidateUser* use case indicate that, the *PlaceOrder & TrackOrder* use cases always includes the behaviors of the *ValidateUser* use case.

UML Use Case Diagram

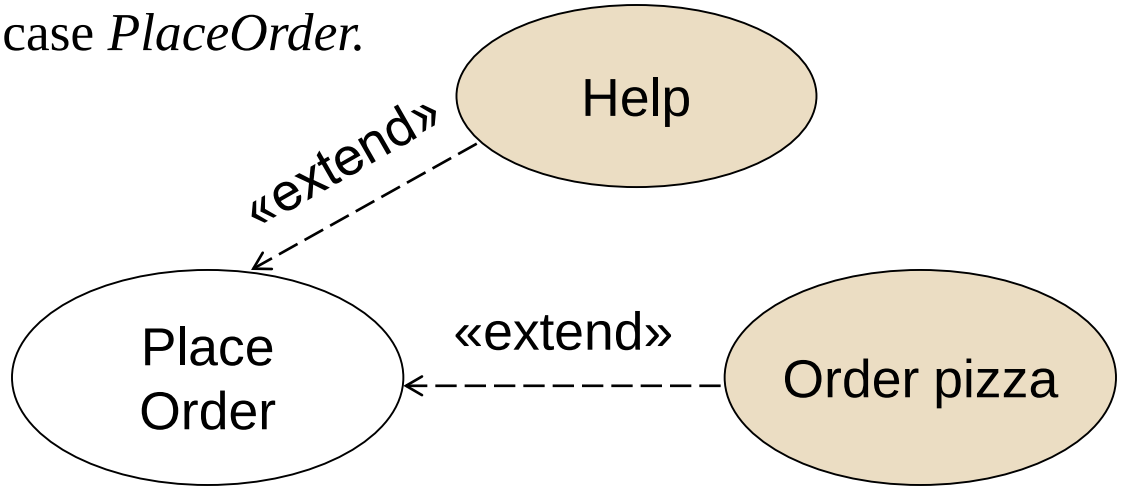
«extend» relationship

- Used to show optional behavior, which is required only under certain condition. This is typically used in exceptional circumstances.
- Represented as dotted line labeled **«extend»** with an arrow toward the base case.



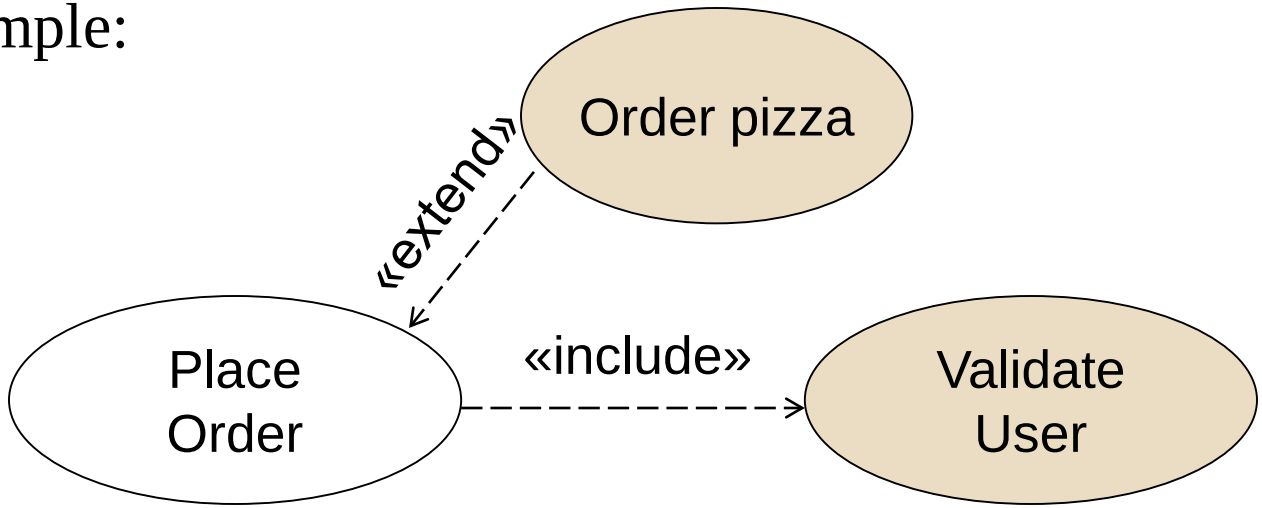
Example:

Here in following example the *OrderPizza & Help* use cases are optional to base use case *PlaceOrder.*



Includes vs. Extend

Example:



Key Points:

	include	extend
Is this use case optional?	*No*	*Yes*
Is the base use case complete without this use case?	*No*	*Yes*
Is the execution of this use case conditional?	*No*	*Yes*
Does this use case change the behavior of the base use case?	*No*	*Yes*

Description: Withdraw money from ATM.

Use case Scenario name: Withdraw money from ATM.	
Participating actors:	Customer ATM Machine Bank
Preconditions:	Network connection is active ATM has available cash

Flow of events:

Bank customer inserts ATM card and enters PIN.

Customer is validated.

ATM displays actions available on ATM unit. Customer selects Withdraw Cash.

ATM prompts account.

Customer selects account.

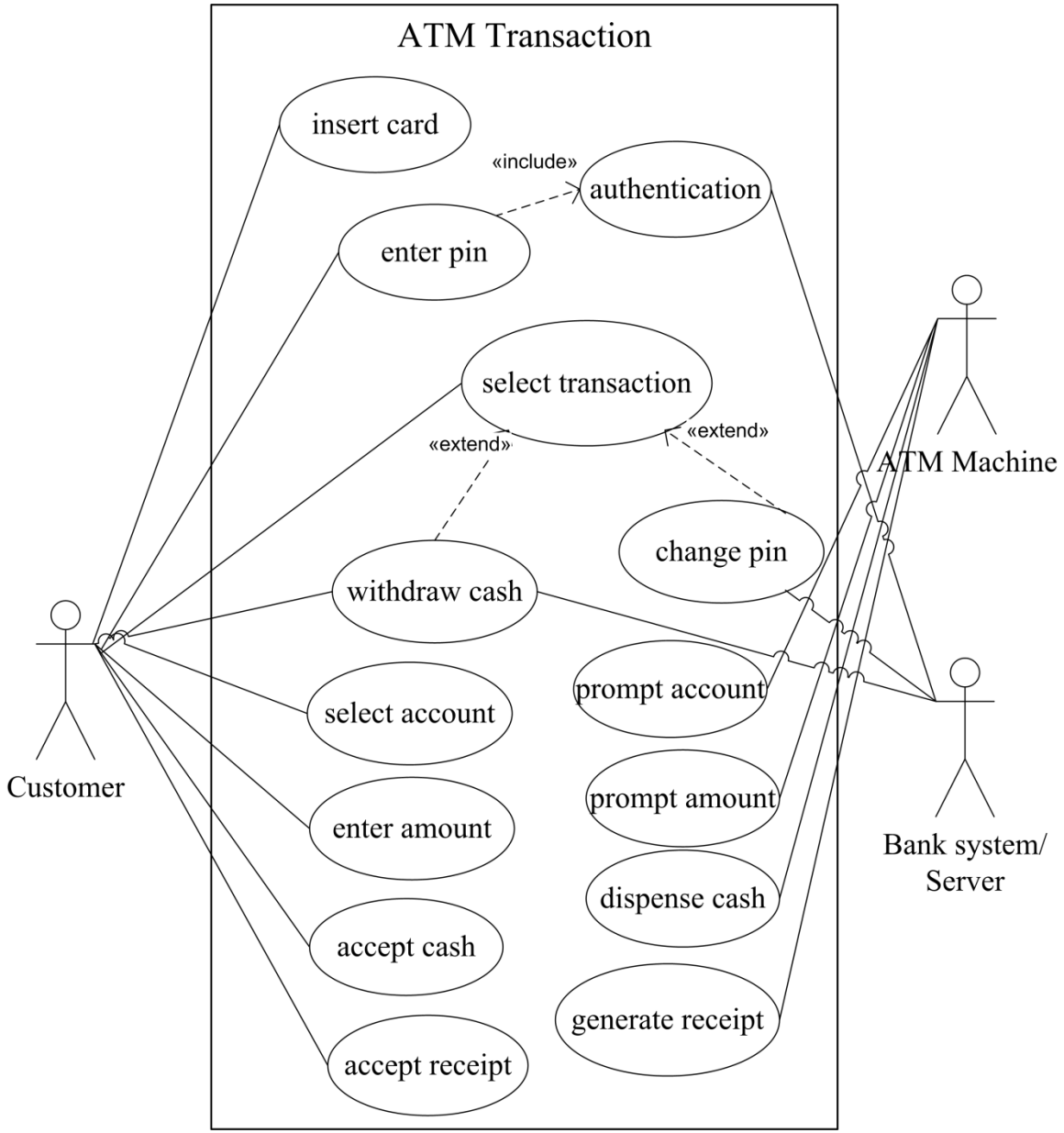
ATM prompts amount.

Customer enters desired amount.

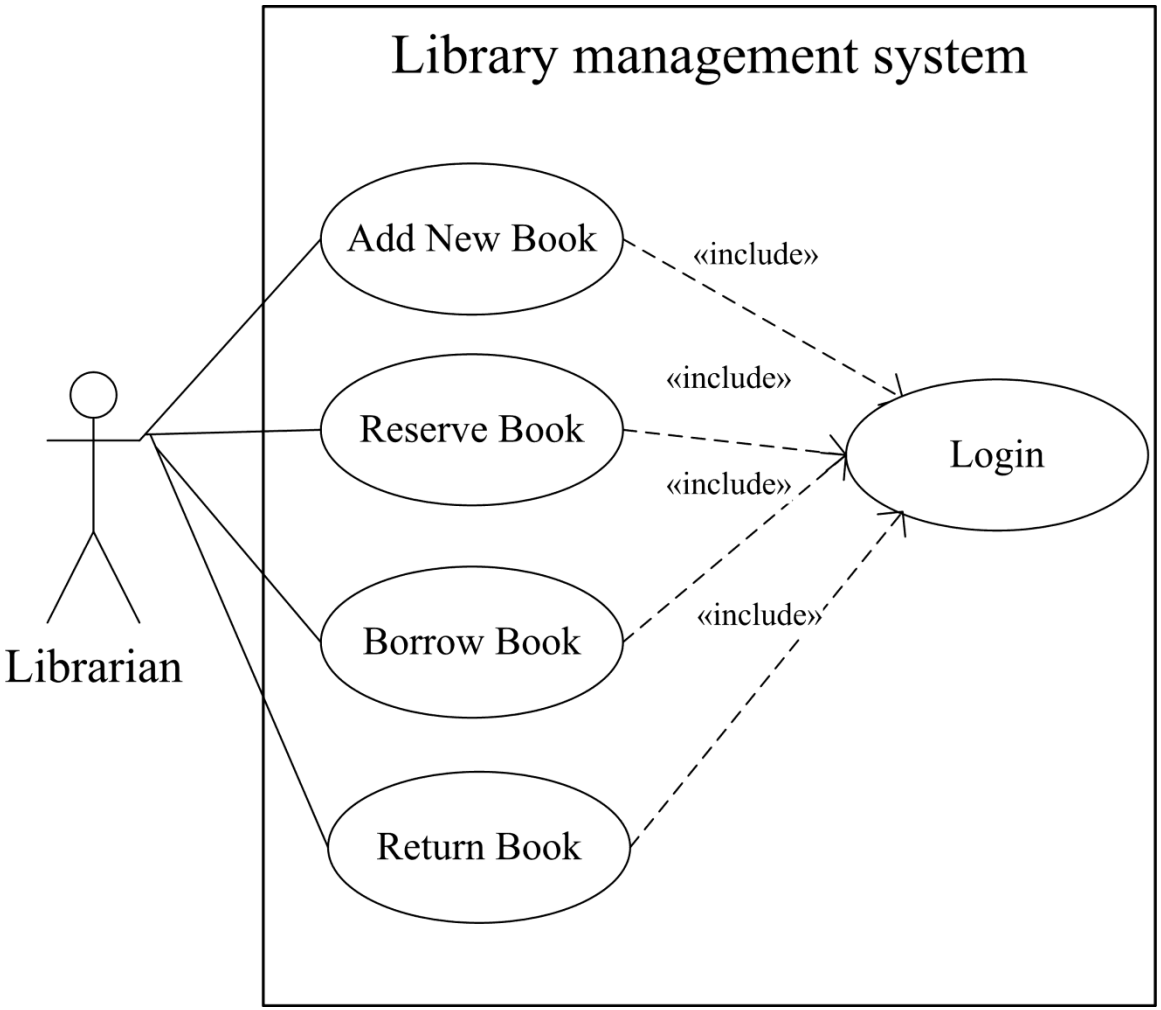
Information sent to Bank, inquiring if sufficient funds/allowable withdrawal limit.

Money is dispensed and receipt prints.

ATM Transaction Example



Library Management System



Example

A coffee vending machine dispenses coffee to **customers**. Customers order coffee by selecting a recipe from a set of recipes. Customers pay using coins. Change is given back if any to the customer. The **service staff** loads ingredients into machine. The service staff can also add a recipe by indicating name of coffee, units of coffee powder, milk, sugar, water and chocolate to be added as well as the cost of coffee.

Actors:

Customer, Service staff

Use Cases:

Dispense coffee

Order

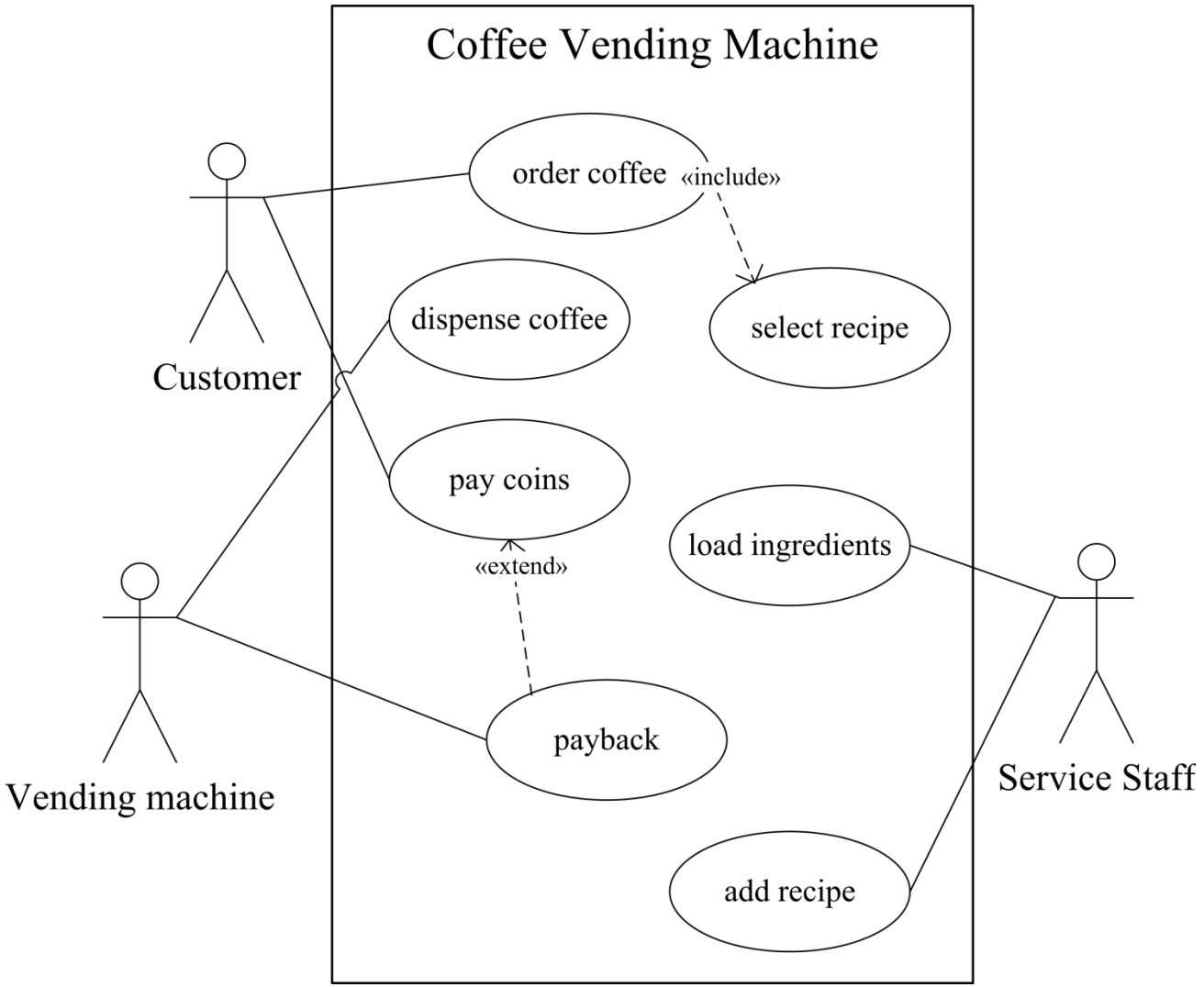
Pay coins

Payback

Load ingredients

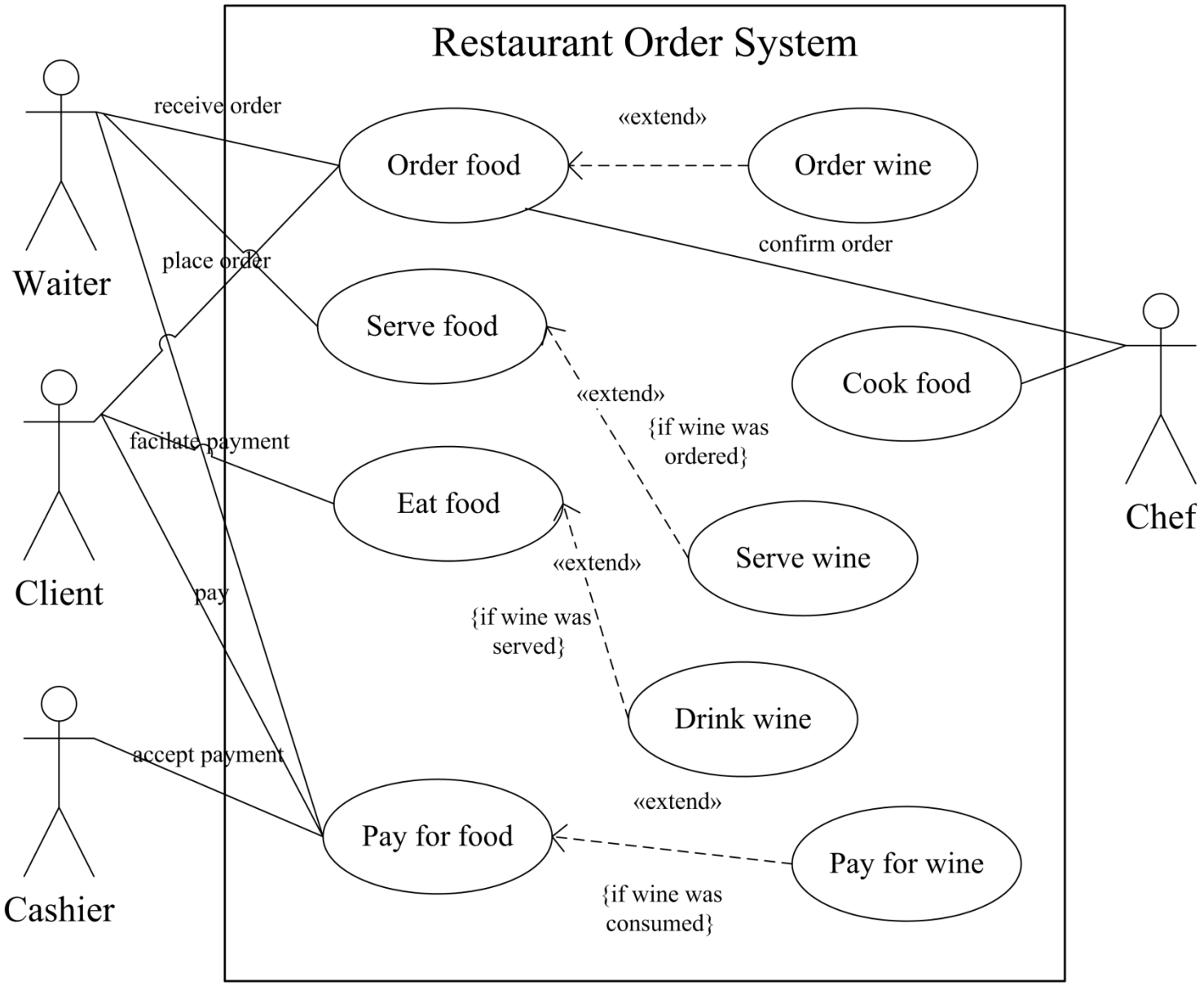
Add recipe

Use Case Diagram Example



Restaurant order System (Class Work)

Restaurant order System (Class Work)



UML

Interaction Diagrams

- *Sequence Diagram*
- *Collaboration (Comm^{n}) Diagram*

UML

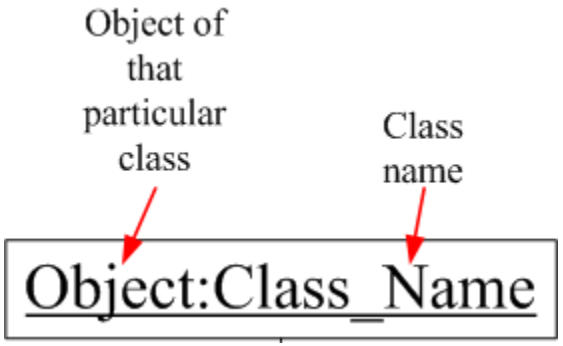
Sequence Diagram

Sequence Diagram

- Sequence diagram is interaction diagram that shows message exchanged or interaction between objects in the system.
- It mainly emphases on time ordering of messages between objects.
- It is used to illustrate the dynamic view of the system

Object or participants:

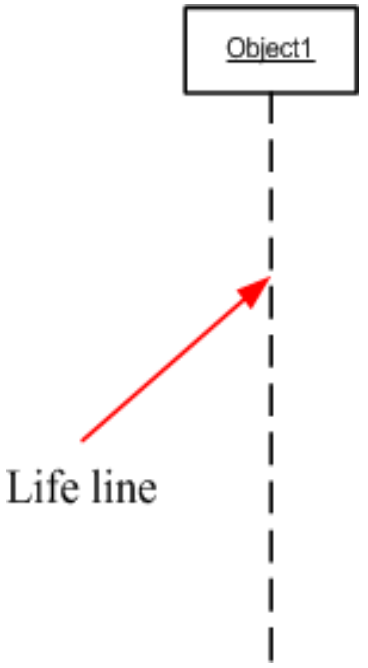
- ✓ The sequence diagram is made up of collection of **participants or objects**. Participants are system parts that interact each other during sequence diagram.
- ✓ The participants interact with each other by sending and receiving message.
- ✓ The object is represented by as:



Sequence Diagram

Lifeline:

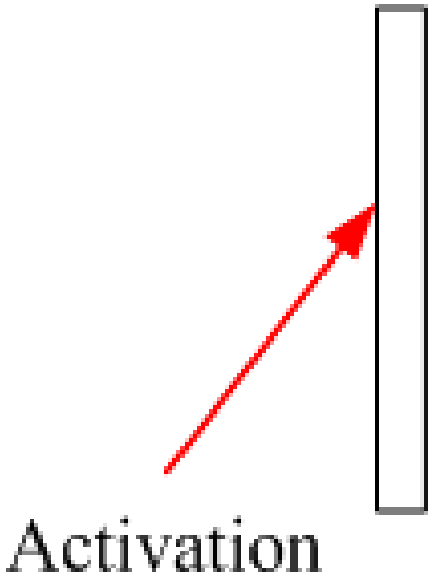
- Lifeline represents the existence of an object over a period of time.
- It is represented by vertical dashed line
- Most objects that appeared in ‘Interaction diagram’ will be in existence for the duration of an interaction. So, these objects are aligned at top of diagram with their lifeline from top to bottom of diagram.



Sequence Diagram

Activation bar:

- It is represented by tall thin rectangle.
- The top of rectangle is aligned with start of the action.
- The bottom is aligned with its completion and can be marked by a written message
- It is also called as focus of control. It shows the period of time during which an object is performing an action or operation.



Sequence Diagram

Messages:

- Messages can be flow in whatever direction required for interaction from left to right and right to left.
- The messages on sequence diagram are specifies using an arrow from participant that wants to pass the messages to the participant that receive the messages.
- The interaction in a sequence diagram between the objects can be shown by using messages.

Message types:

1) Asynchronous messages:-

- ✓ It is a message where the sender is not blocked and can continue executing.
- ✓ Representing by solid line with open arrowhead.

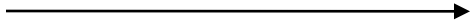
Asynchronous message

Sequence Diagram

2) Synchronous messages

- It is a message where the sender is blocked and waits until the receiver has finished processing of message.
- It is invoked the caller waits for the receiver to return from the message invocation.
- It is represented by solid line with filled arrowhead.

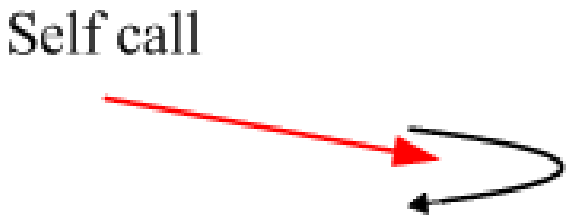
Synchronous message



Sequence Diagram

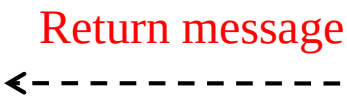
3) Reflexive messages/self message:-

- If the object sends the message to itself then it is called as ‘Reflexive message’.
- It is represented by solid line with loops the lifeline of object.



4) Return messages:-

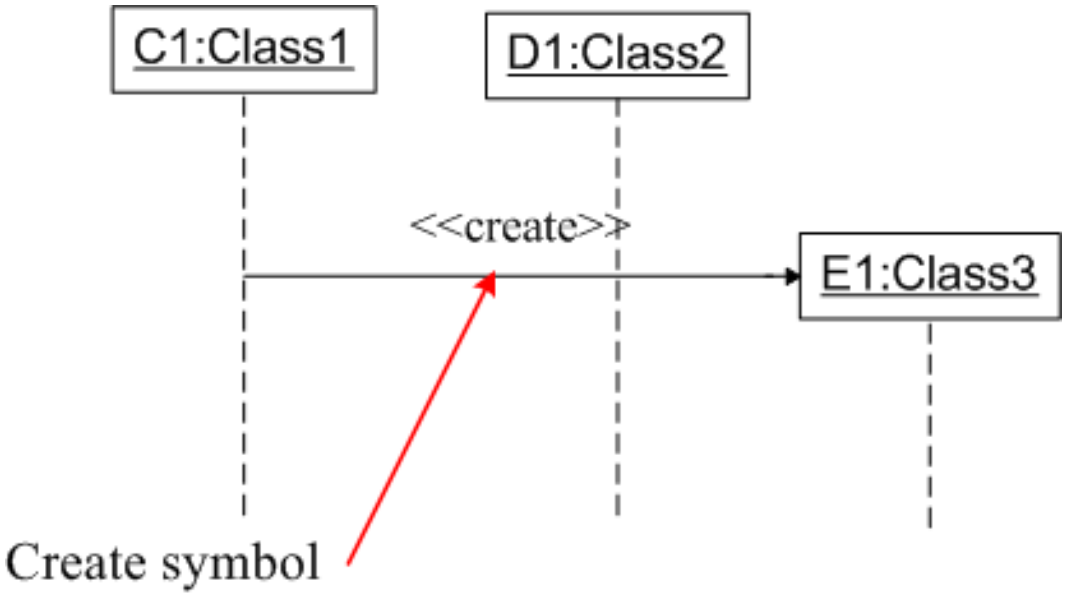
- It can be used at the end of activation bar to show that control flow of activation returns to the participant that pass the original message.
- It is represented by dashed line from sender to receiver.



Sequence Diagram

5) Create messages:

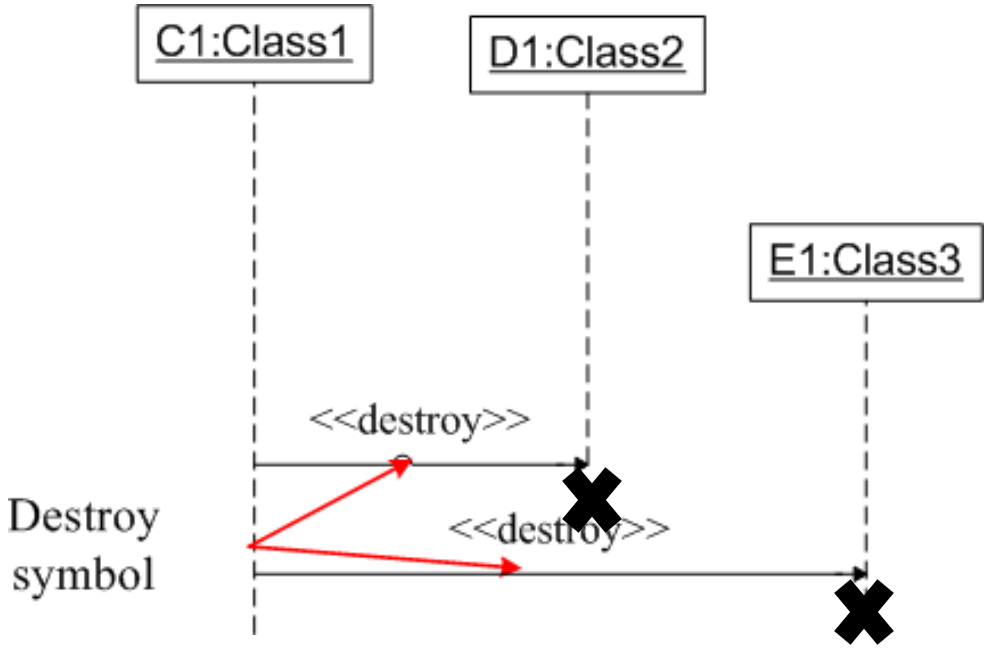
- It is used to create object during interaction.
- The object can be created by using <<create>> to indicate the timing of creation.
- Creating message can be shown as below:



Sequence Diagram

6) Destroy messages:

- It is used to destroy the objects during interaction.
- The objects can be terminated using <<destroy>> which points to an “x”.
- It indicates that object named message is terminated.

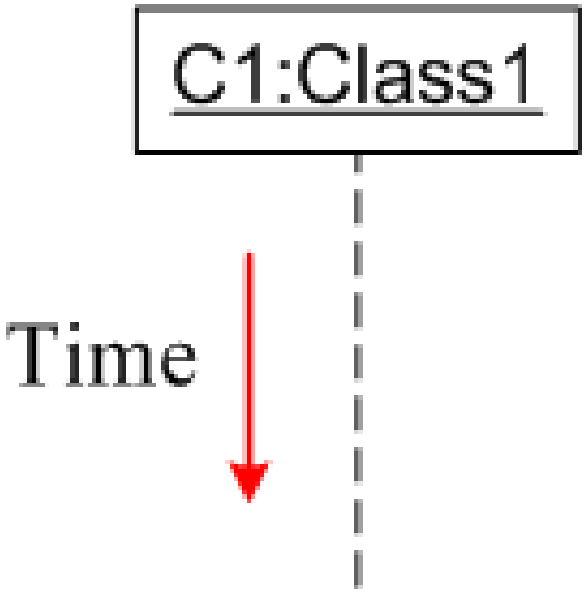


Note: Avoid modeling object destruction unless memory management is critical.

Sequence Diagram

Time:

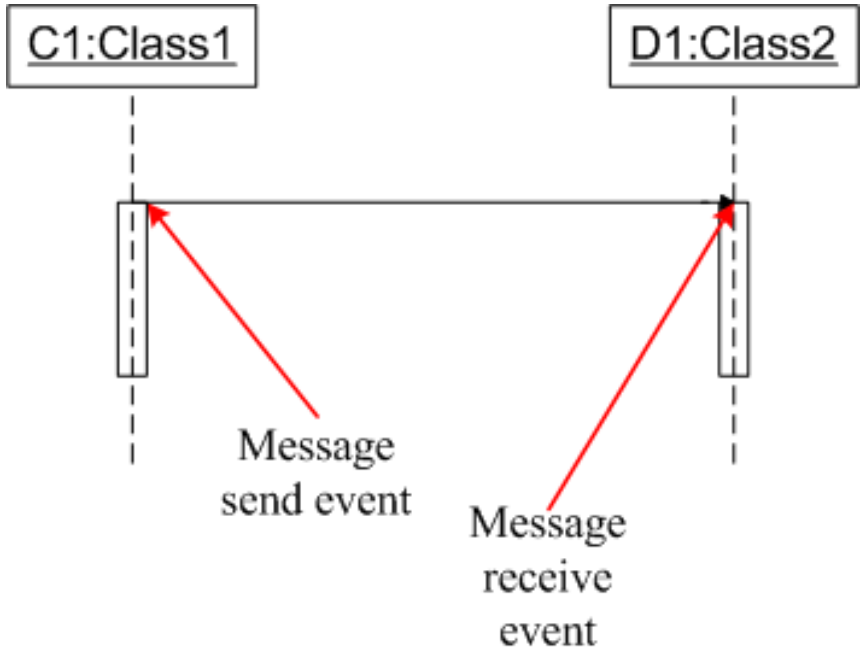
- Time is all about ordering but not duration.
- So time in an important factor.
- The time on sequence diagram starts at top of the page just below the object and then progress down the page.
- The sequence diagram describes the order in which interaction takes place.



Sequence Diagram

Event:

- Event is created while sending and receiving message.
- When interaction takes place, Events are called as build in blocks for messages and signals.
- It can be referred as smallest part of an interaction and event can occur of at any given point in a Time.



Sequence Diagram: Control Information

Condition

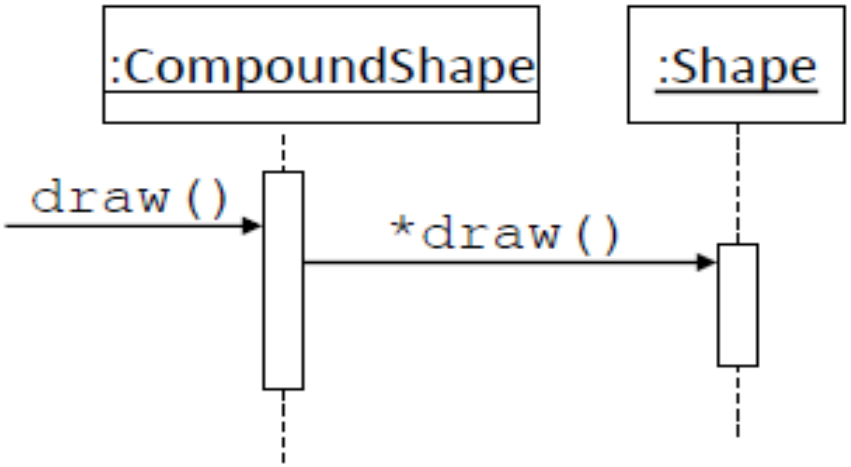
- **syntax:** [expression or condition] message-label
- The message is sent only if the condition is true
- example: [user valid= “true”] give access

Iteration

- **syntax:** * [expression] message-label or *message-label

Note: * [expression] message-label is not standard syntax

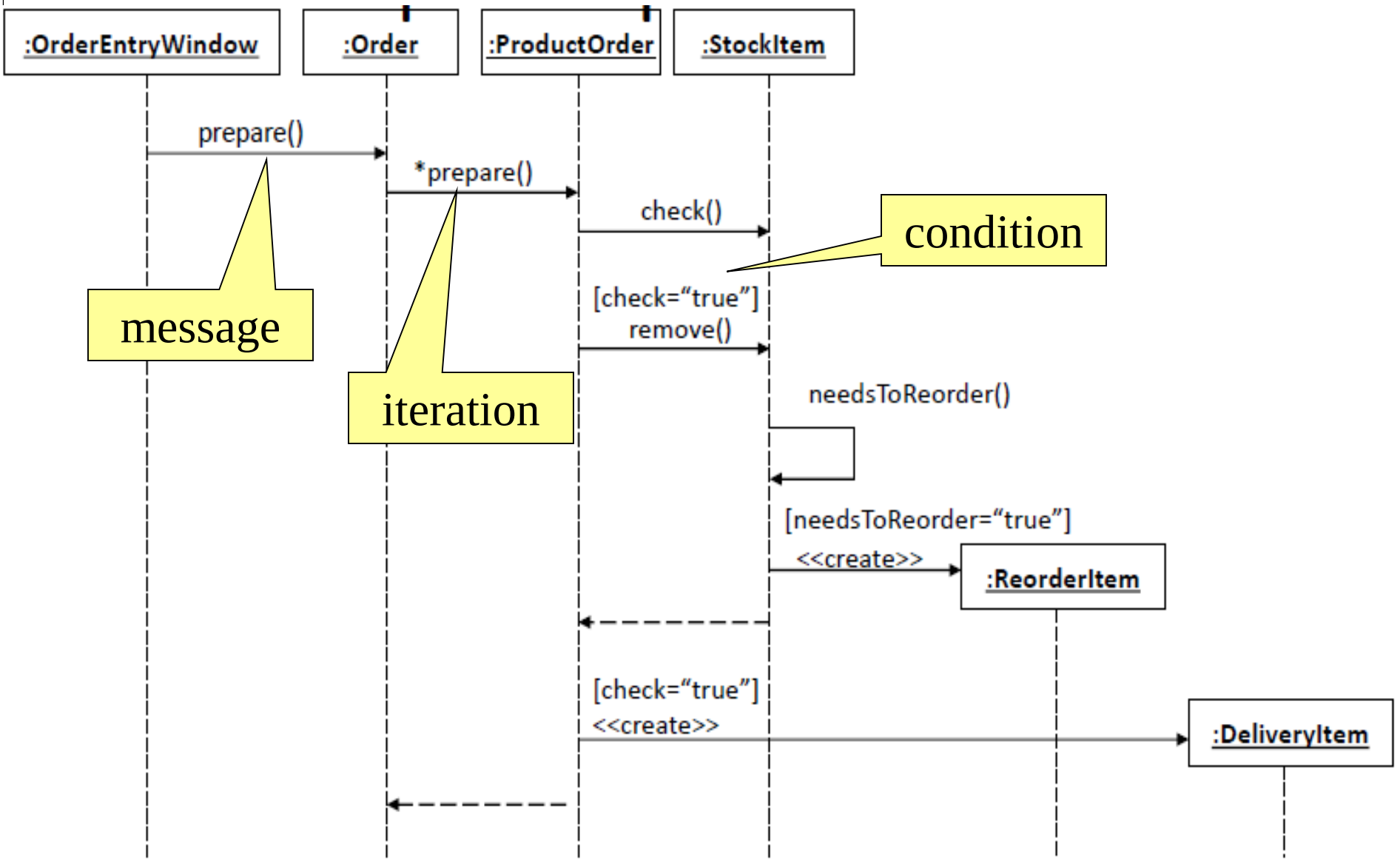
- The message is sent many times to possibly multiple receiver objects.



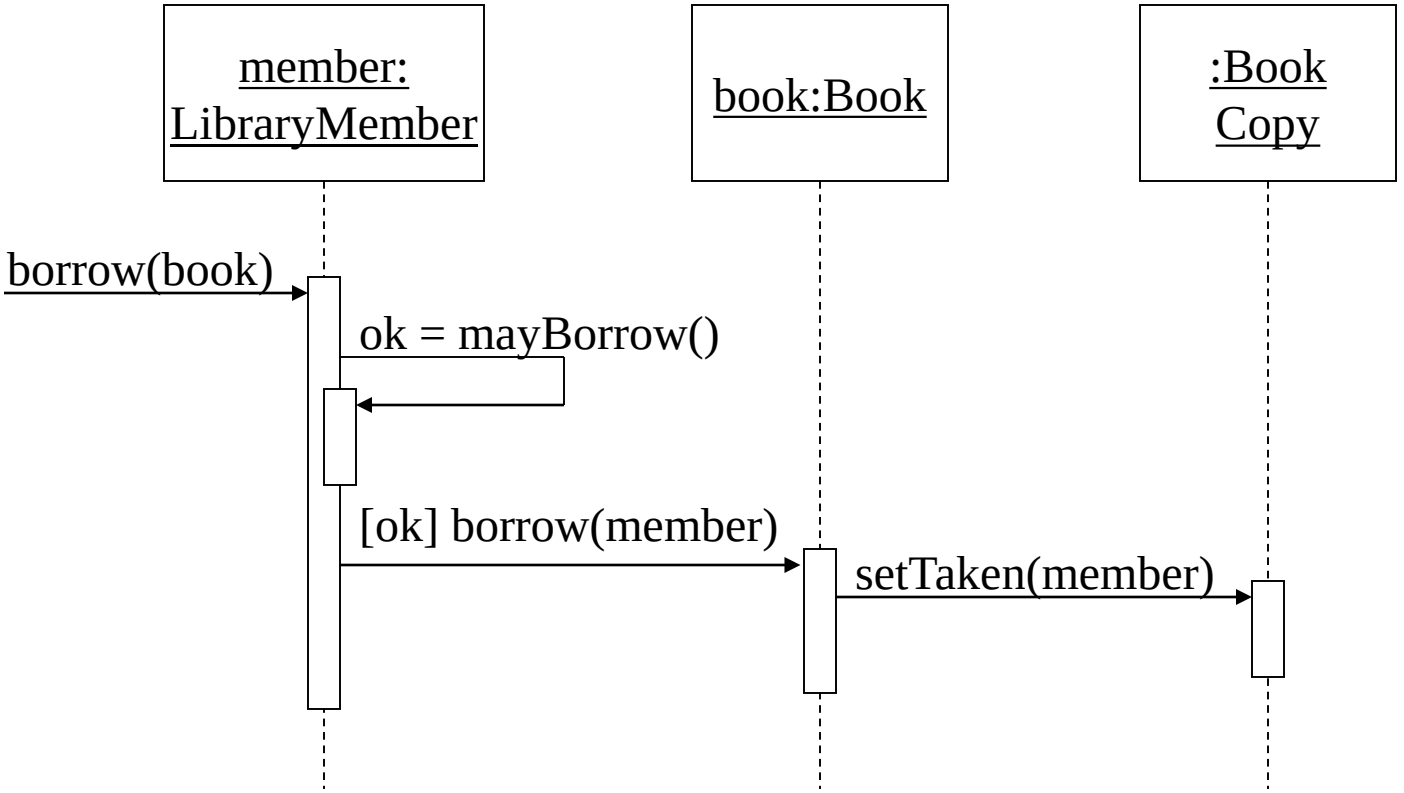
Sequence Diagram

- The control mechanisms of sequence diagrams suffice only for modeling simple alternatives.
- Consider drawing several diagrams for modeling complex scenarios.
- Don’t use sequence diagrams for detailed modeling of algorithms (this is better done using *activity diagrams, pseudo-code or state-charts).*

Order processing → POS

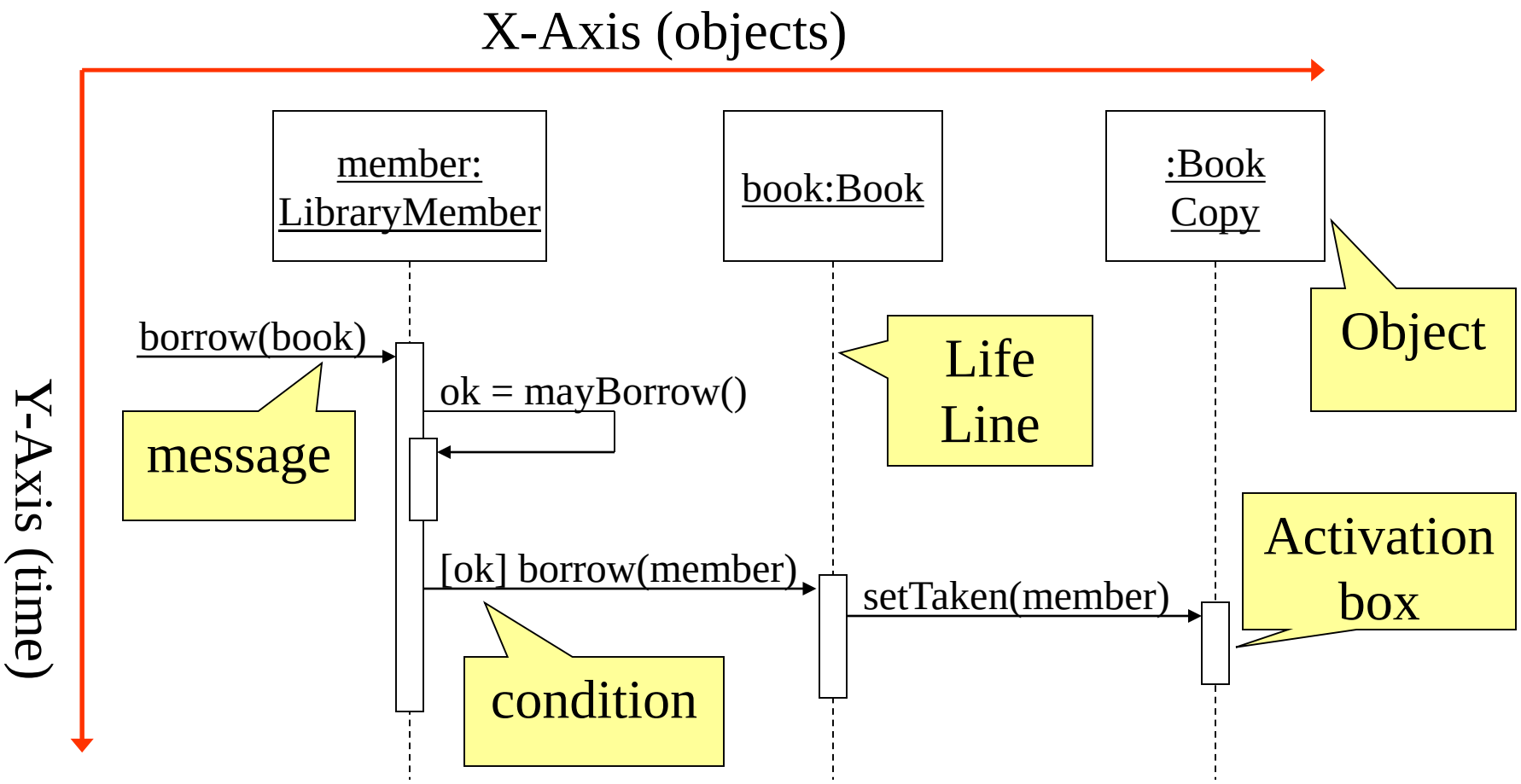


Borrowing book from library



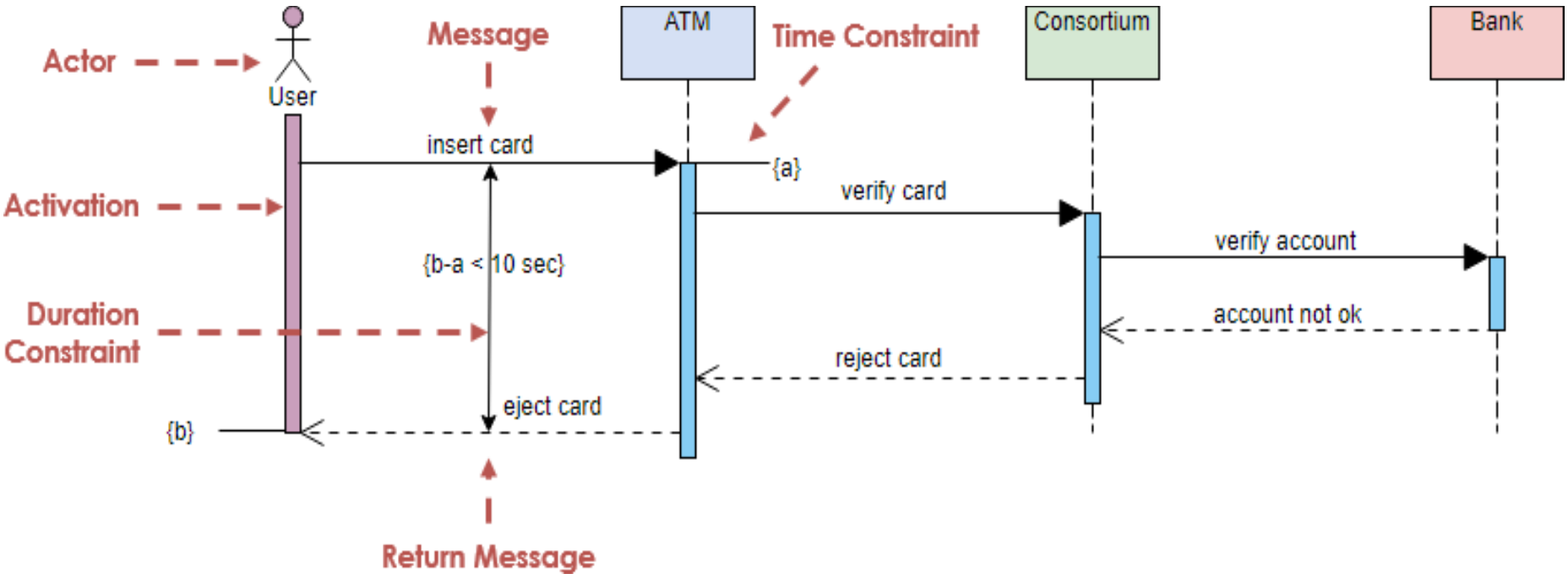
Sequence Diagrams

Borrowing book from library

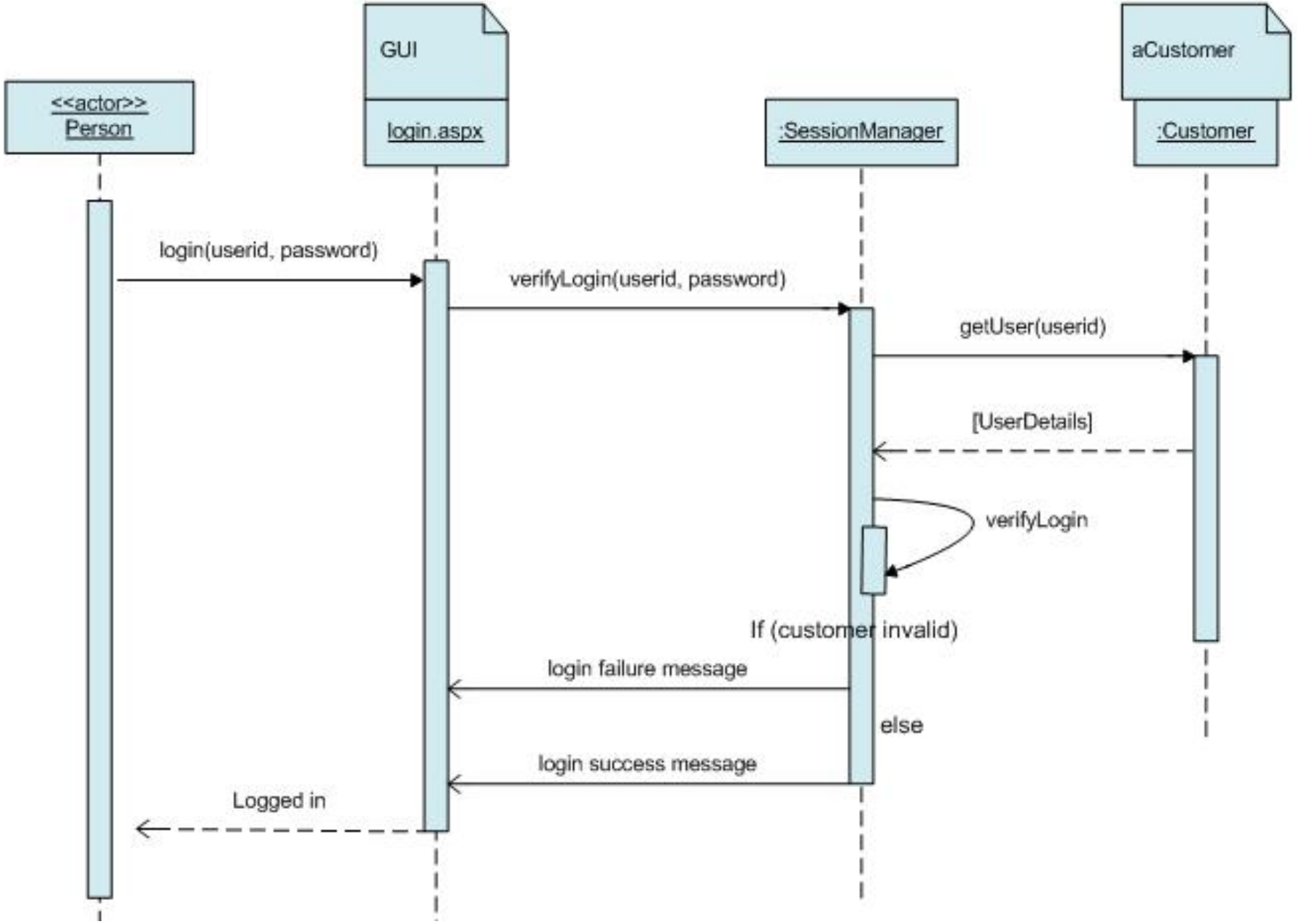


Sequence Diagrams

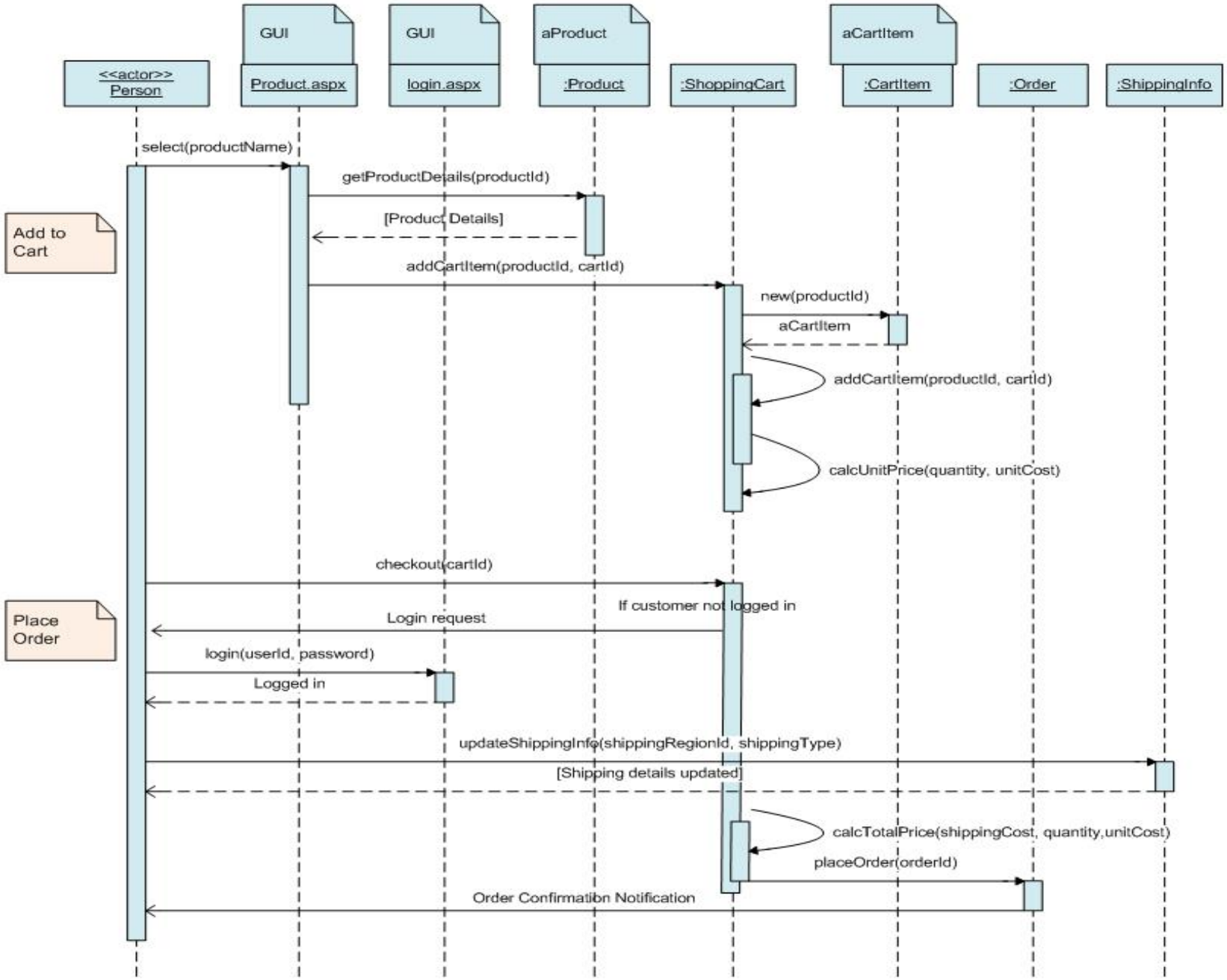
ATM (Invalid Pin)



Login


<<actor>>
Person
GUI
login.aspx
:SessionManager
aCustomer
:Customer
login(userid, password)
verifyLogin(userid, password)
getUser(userid)
[UserDetails]
verifyLogin
If (customer invalid)
login failure message
else
login success message
Logged in

Online shopping: Explanation [Assignment]

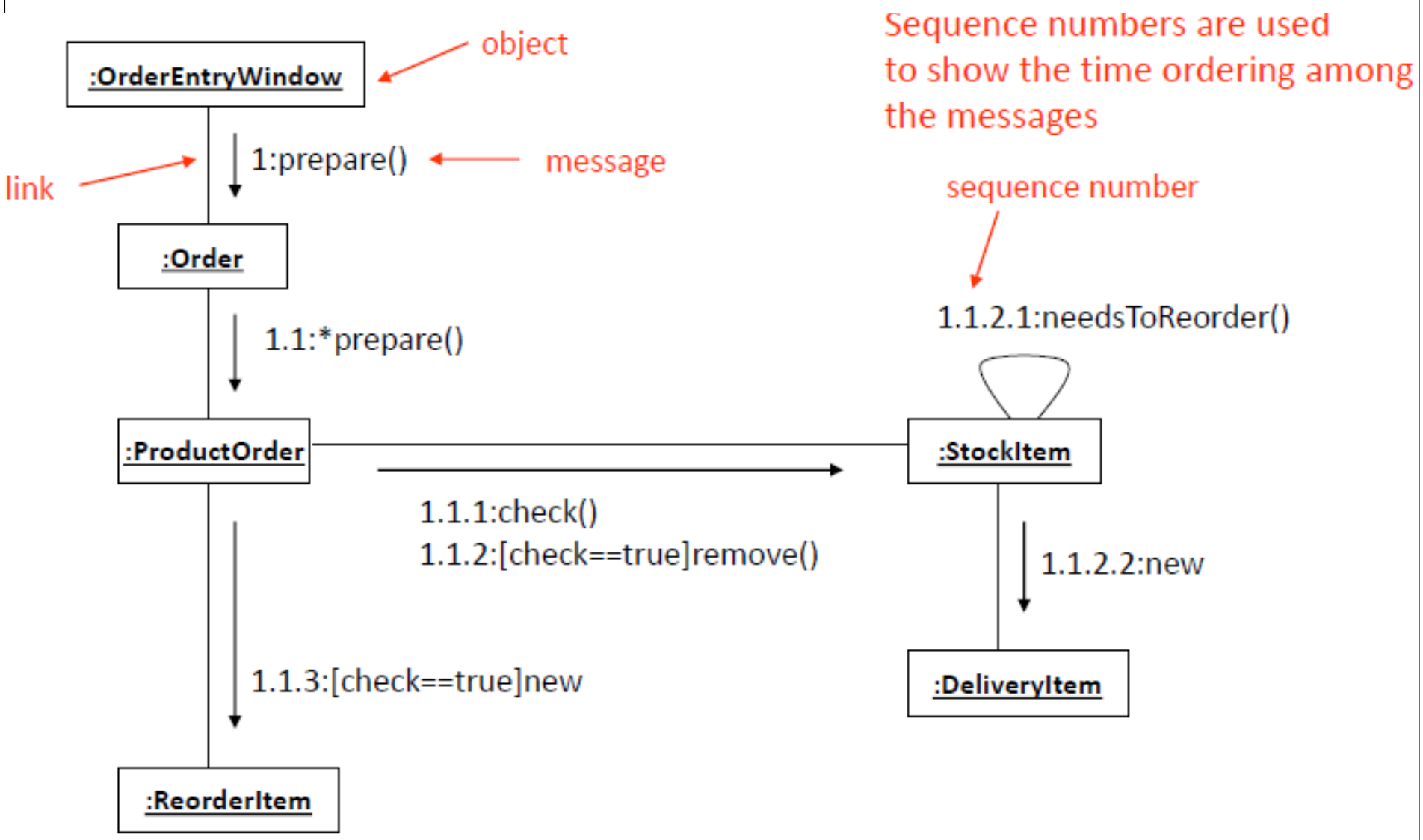


UML Collaboration/ Comm^{n} Diagram

Collaboration (Communication) Diagrams

- Collaboration diagrams (also called Communication diagrams) show a particular sequence of messages exchanged between a number of objects
  - ✓ This is what sequence diagrams do too!
- Sequence diagram highlight more the temporal aspect of the system i.e. it shows object interaction in timely manner(so no need of numbering the messages).
- In Collaboration diagram, the temporal aspect can be shown here too, by numbering the interactions with sequential labels. (so need to numbering the messages).
- So Sequence numbers are used to show the time ordering among the messages.

Collaboration (Communication) Diagrams



UML

Activity Diagram

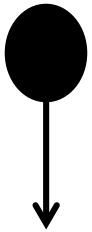
Activity Diagram

- An activity diagram visually represents a series of actions or flow of control in a system similar to a flowchart.
- Activities modeled can be sequential and concurrent.
- In both cases an activity diagram will have a beginning (an initial state) and an end (a final state) and in between them series of actions to be performed by the system.

Symbols in Activity Diagram

Initial State or Start Point

- A small filled circle followed by an arrow represents the initial action state or the start point for any activity diagram.



Symbols in Activity Diagram

Activity or Action State

- An activity represents execution of an action or performing some operation.
- It is represented using a rectangle with rounded corners.

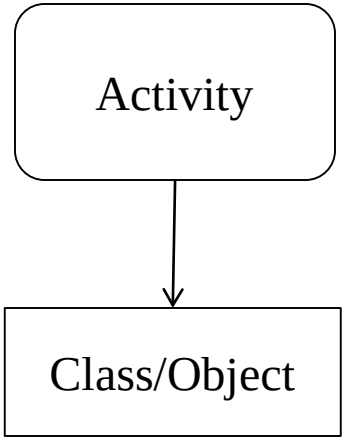
Action Flow

- Action flows, also called edges and paths, illustrate the transitions from one action state to another.
- They are usually drawn with an arrowed line.

Symbols in Activity Diagram

Object Flow

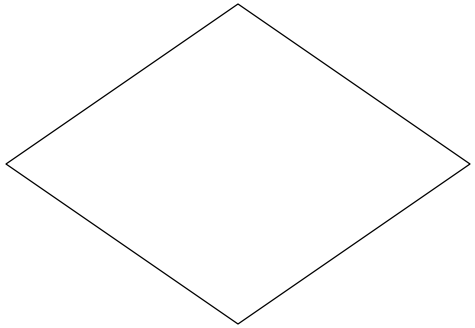
- Object flow refers to the creation and modification of objects by activities.
- An object flow arrow from an action to an object means that the action creates or influences the object.
- An object flow arrow from an object to an action indicates that the action state uses the object.



Symbols in Activity Diagram

Decisions and Branching

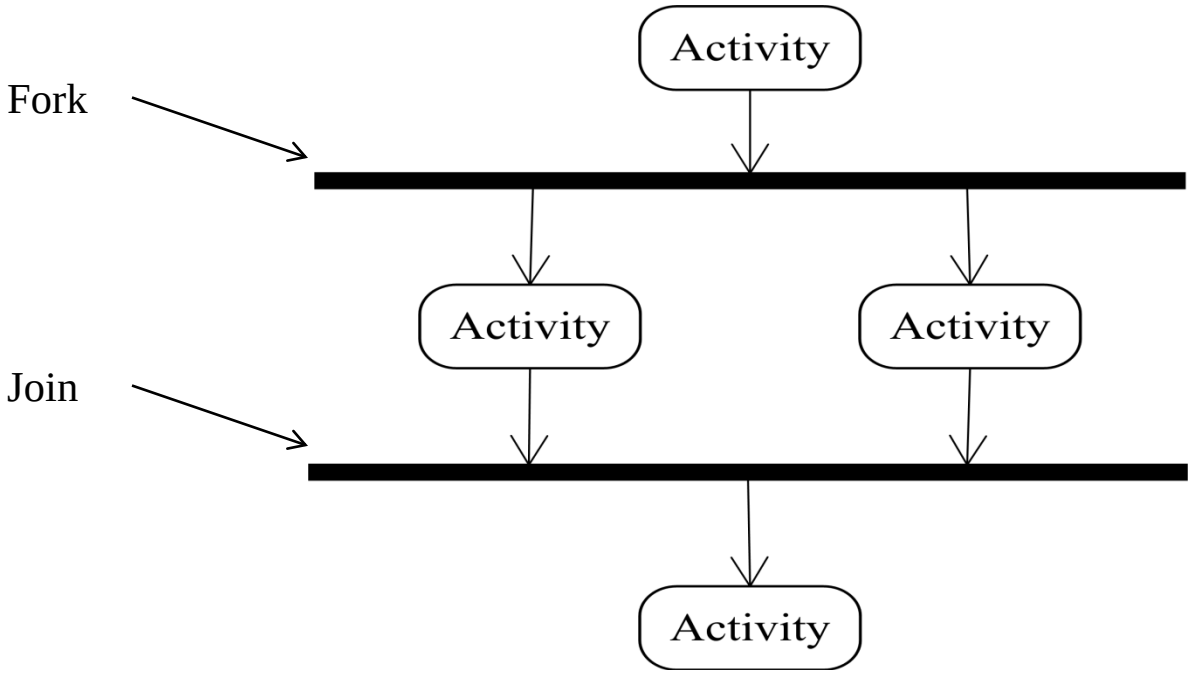
- A diamond represents a decision with alternate paths.
- When an activity requires a decision prior to moving on to the next activity, add a diamond between the two activities.
- The outgoing alternates should be labeled with a condition or guard expression. You can also label one of the paths "else."



Symbols in Activity Diagram

Synchronization

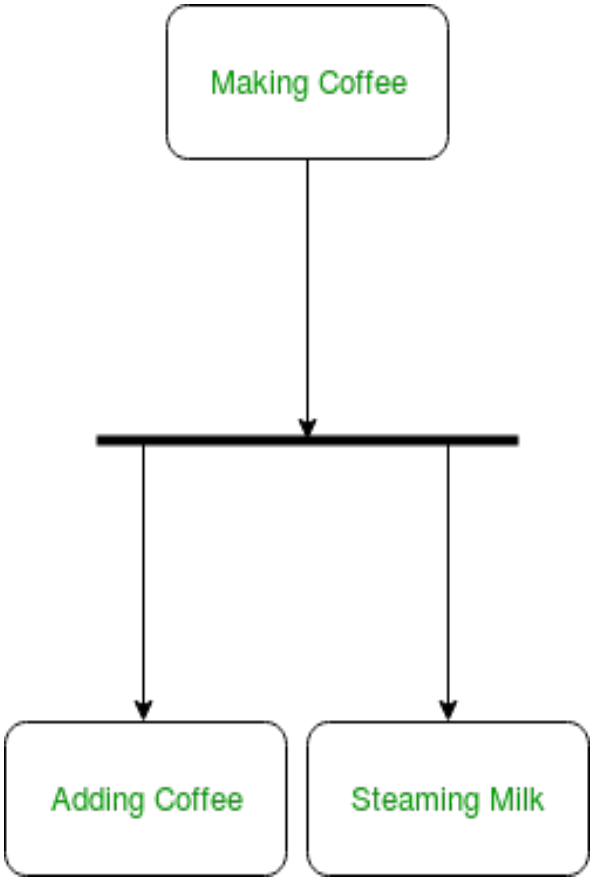
- A fork node is used to split a single incoming flow into multiple concurrent flows. It is represented as a straight, slightly thicker line in an activity diagram.
- A join node joins multiple concurrent flows back into a single outgoing flow.
- A fork and join mode used together are often referred to as synchronization.



Symbols in Activity Diagram

Fork

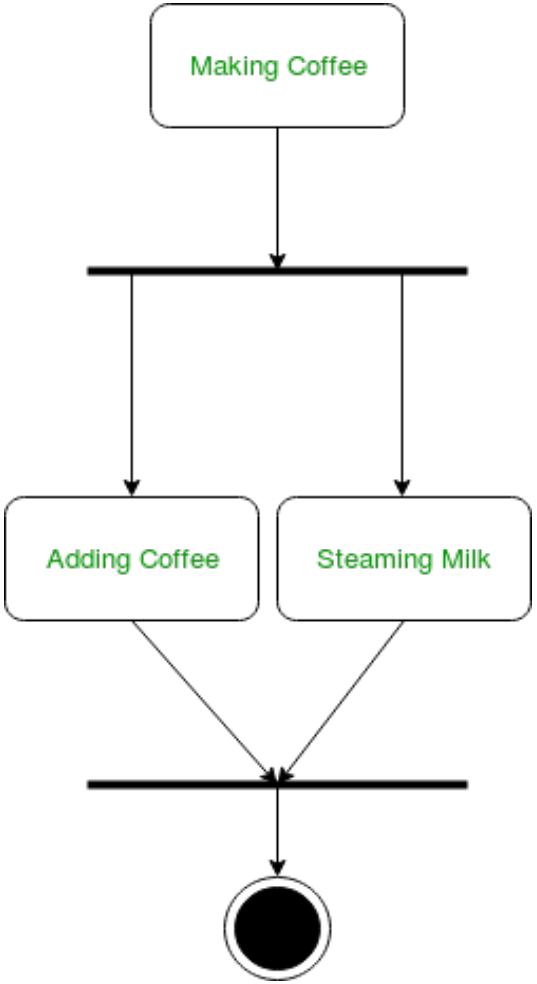
- A fork node is used to split a single incoming flow into multiple concurrent flows.
- It is represented as a straight, slightly thicker line in an activity diagram.



Symbols in Activity Diagram

Join

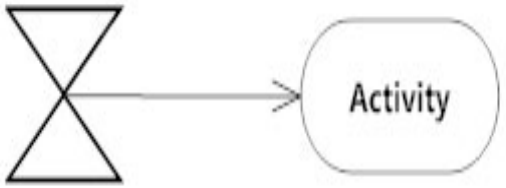
- A join node joins multiple concurrent flows back into a single outgoing flow.



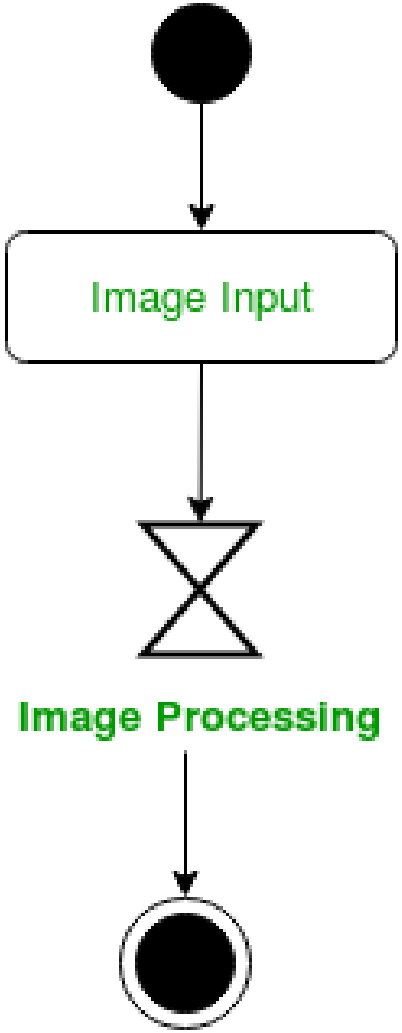
Symbols in Activity Diagram

Time Event

- This refers to an event that stops the flow for some amount of time.
- It is represented by a hourglass.



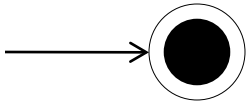
Time Event



Symbols in Activity Diagram

Final State or End Point

- An arrow pointing to a filled circle nested inside another circle represents the final action state.



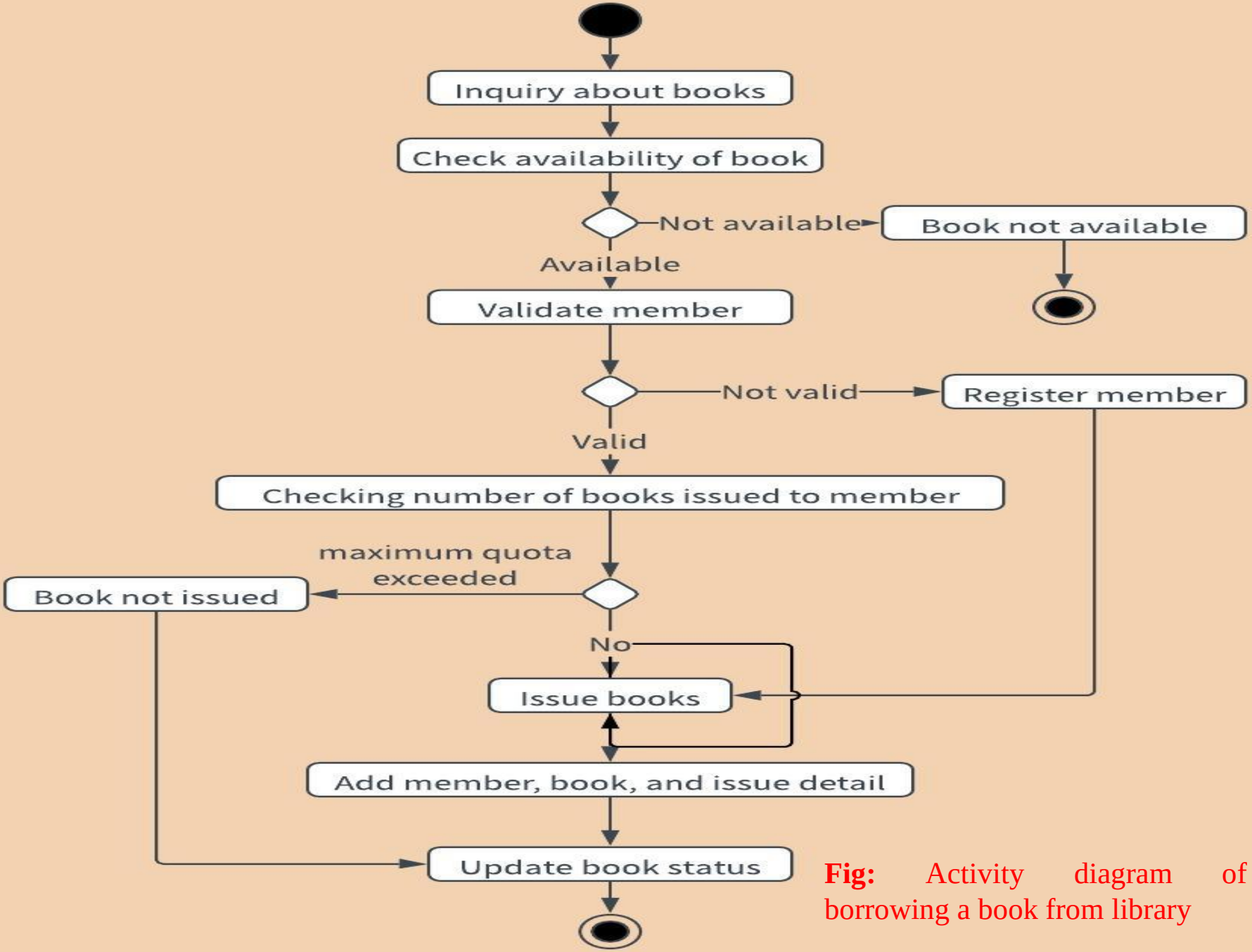


Fig: Activity diagram of borrowing a book from library

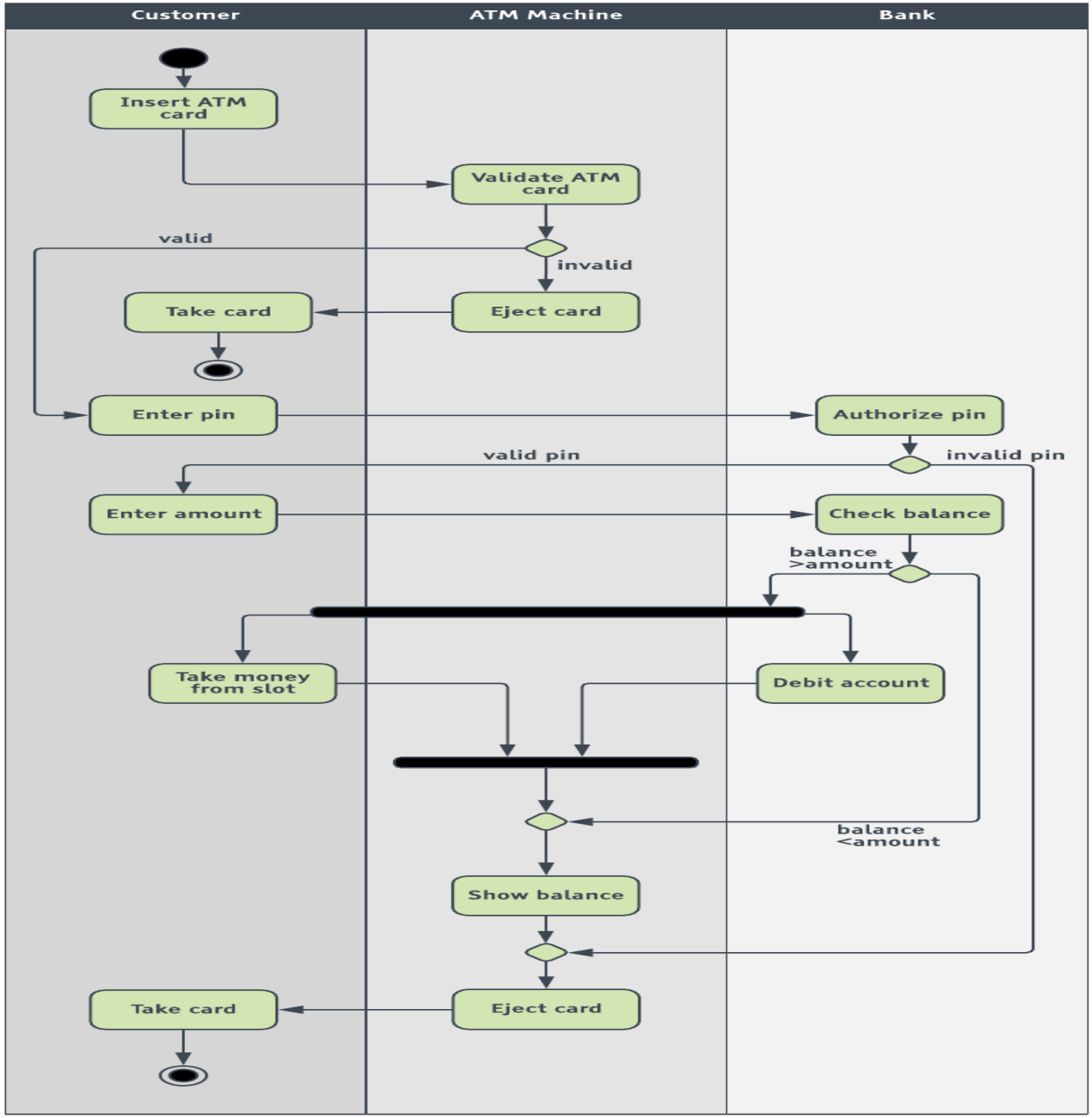
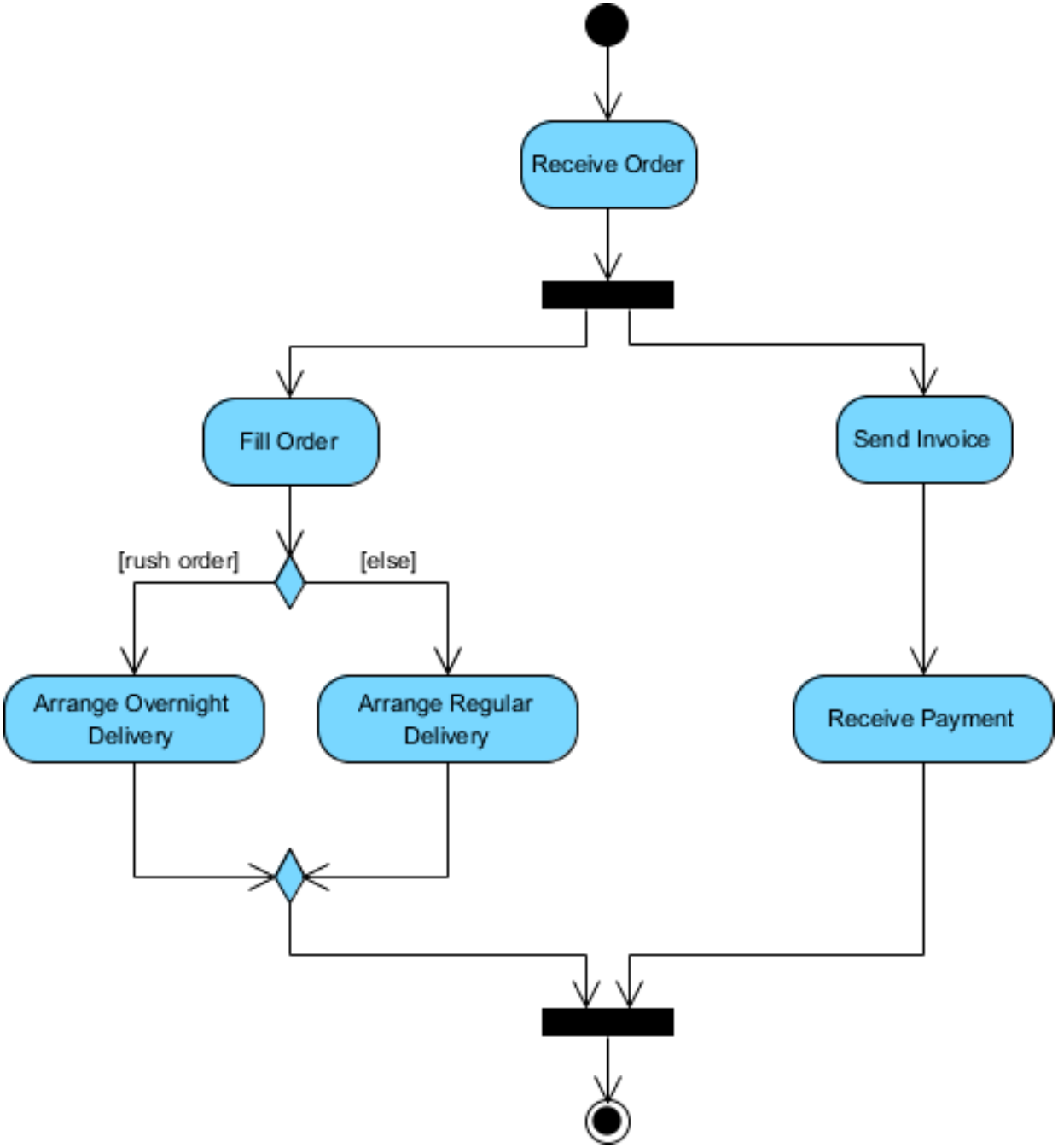


Fig: Swimlane diagram of ATM transaction

Process Order - Problem Description (Class Work)

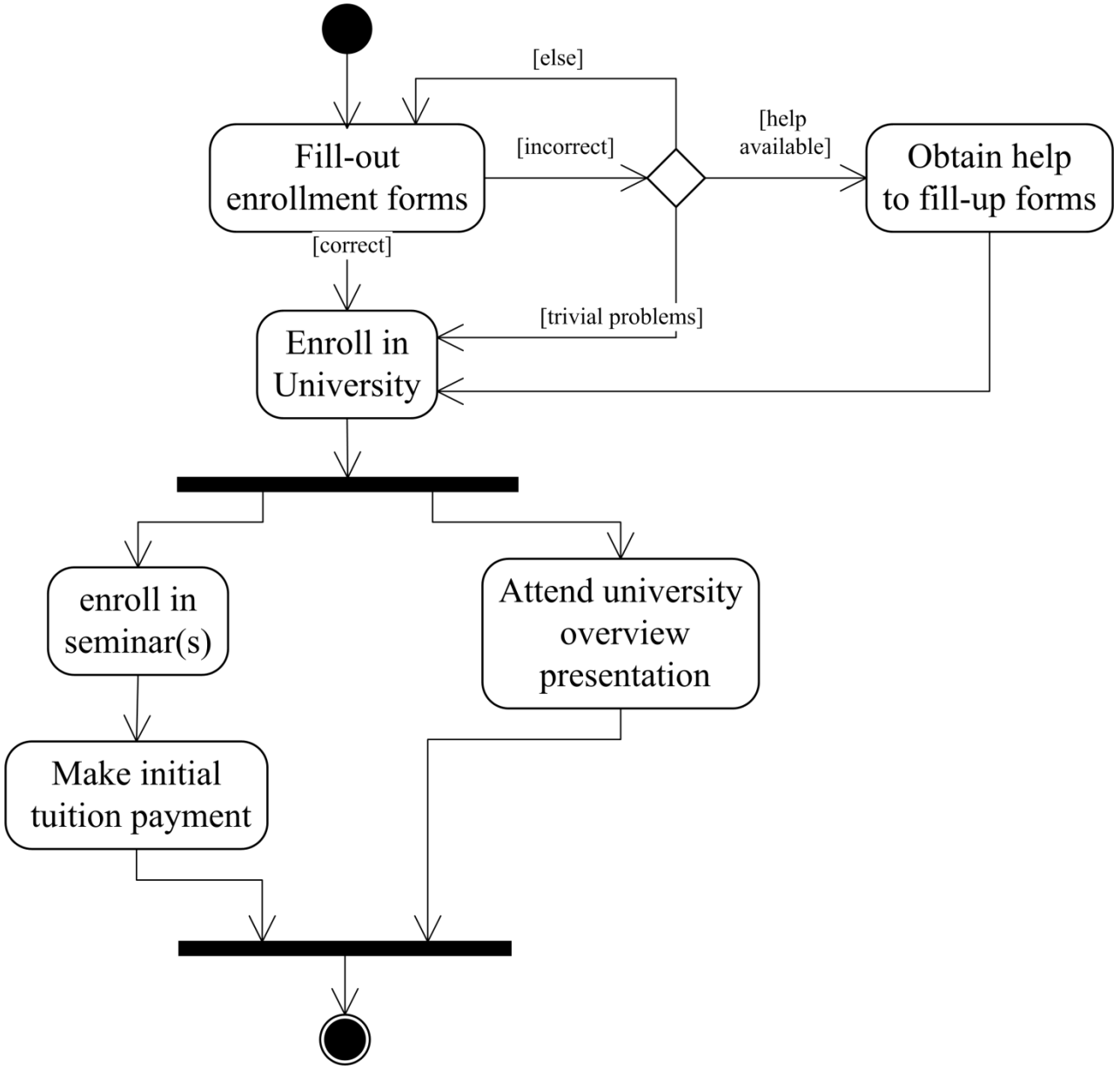
- Once the order is received, the activities split into two parallel sets of activities. One side fills and sends the order while the other handles the billing.
- On the Fill Order side, the method of delivery is decided conditionally. Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed.
- Finally the parallel activities combine to close the order.


Receive Order
Fill Order
Send Invoice
[rush order]
[else]
Arrange Overnight Delivery
Arrange Regular Delivery
Receive Payment

Activity Diagram

Activity Diagram Example - Student Enrollment (CW)

- An applicant wants to enroll in the university.
- The applicant hands a filled out copy of Enrollment Form.
- The registrar inspects the forms.
- The registrar determines that the forms have been filled out properly.
- The registrar informs student to attend in university overview presentation.
- The registrar helps the student to enroll in seminars
- The registrar asks the student to pay for the initial tuition.


[else]
Fill-out enrollment forms
[incorrect]
[help available]
Obtain help to fill-up forms
[correct]
[trivial problems]
Enroll in University
enroll in seminar(s)
Attend university overview presentation
Make initial tuition payment

UML

State Machine/ State Chart Diagram

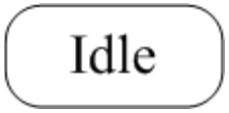
State Machine Diagram

- **An object responds differently to the same event depending on what state it is in**.
- 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.
- State diagrams are used to show possible states a single object can get into
  - ✓ i.e. shows states of an object .
- How object changes state in response to events
  - ✓ shows transitions between states

State Machine Diagram

States

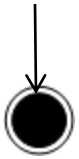
- A state is denoted by a round-cornered rectangle with the name of the state written inside it.



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.

Initial state

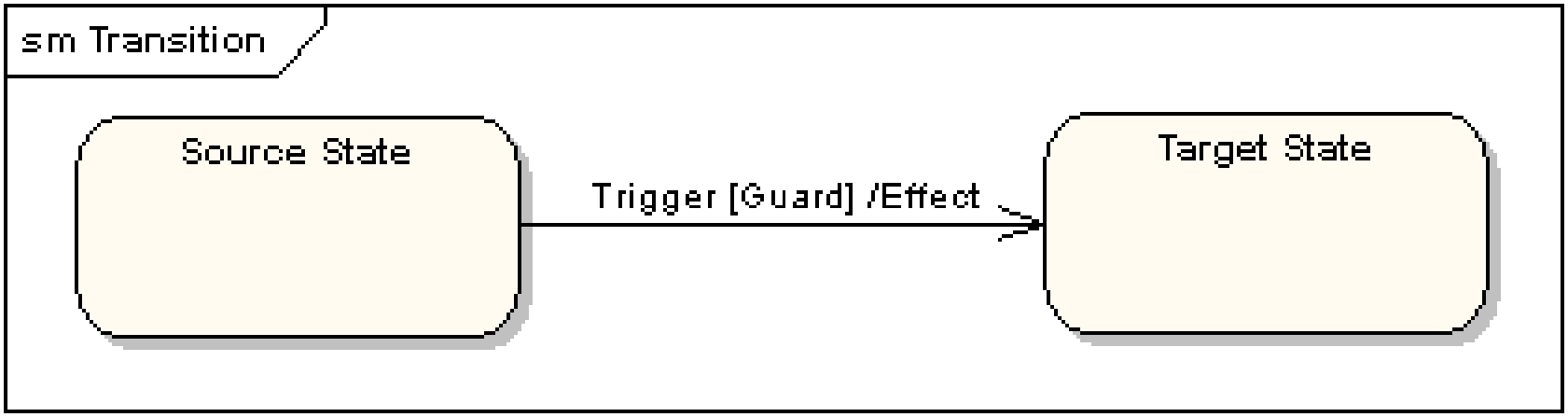


Final state

State Machine Diagram

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.

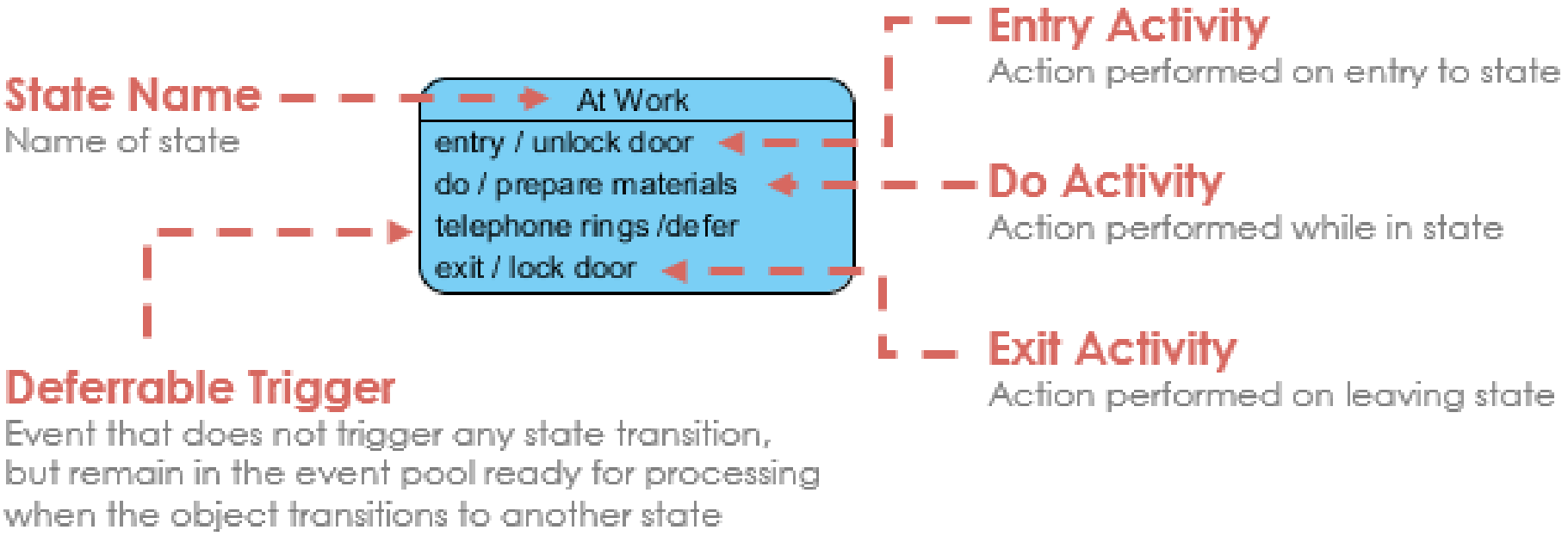


- "Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. "Guard" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

State Machine Diagram

State Actions

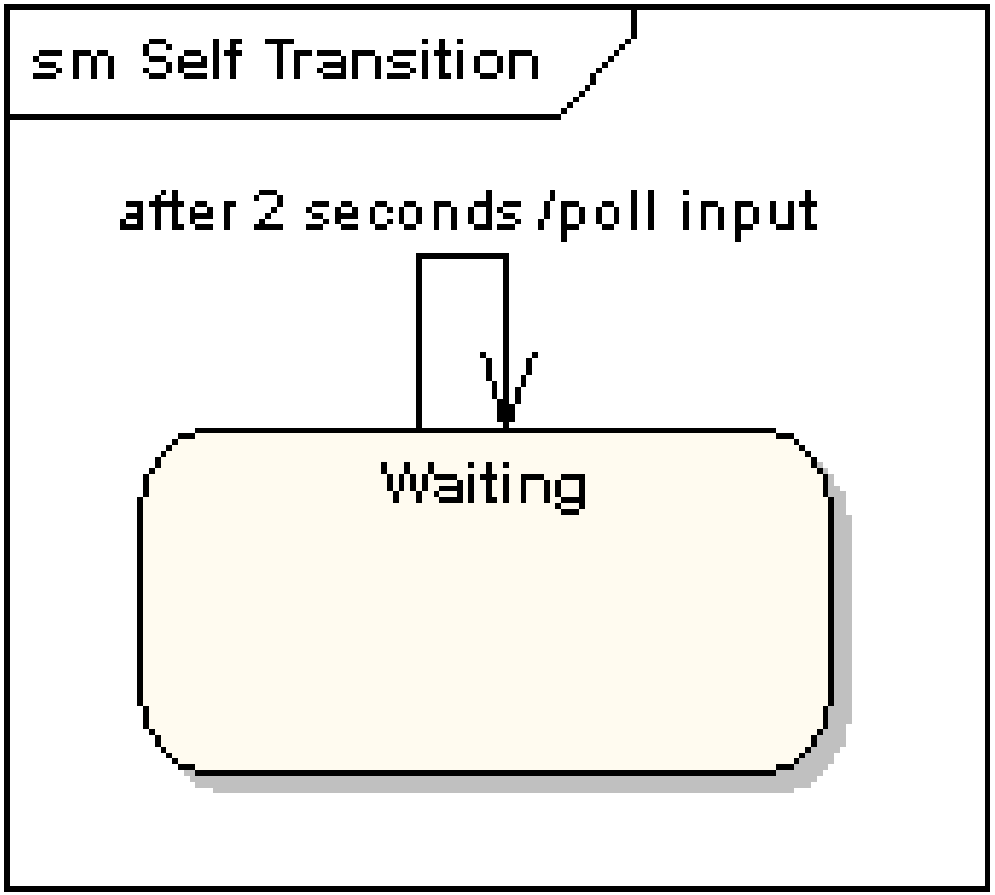
- For each transition, an effect was associated with the transition.
  - ✓ Entry
  - ✓ Exit
  - ✓ Do
  - ✓ Defer



State Machine Diagram

Self-Transitions

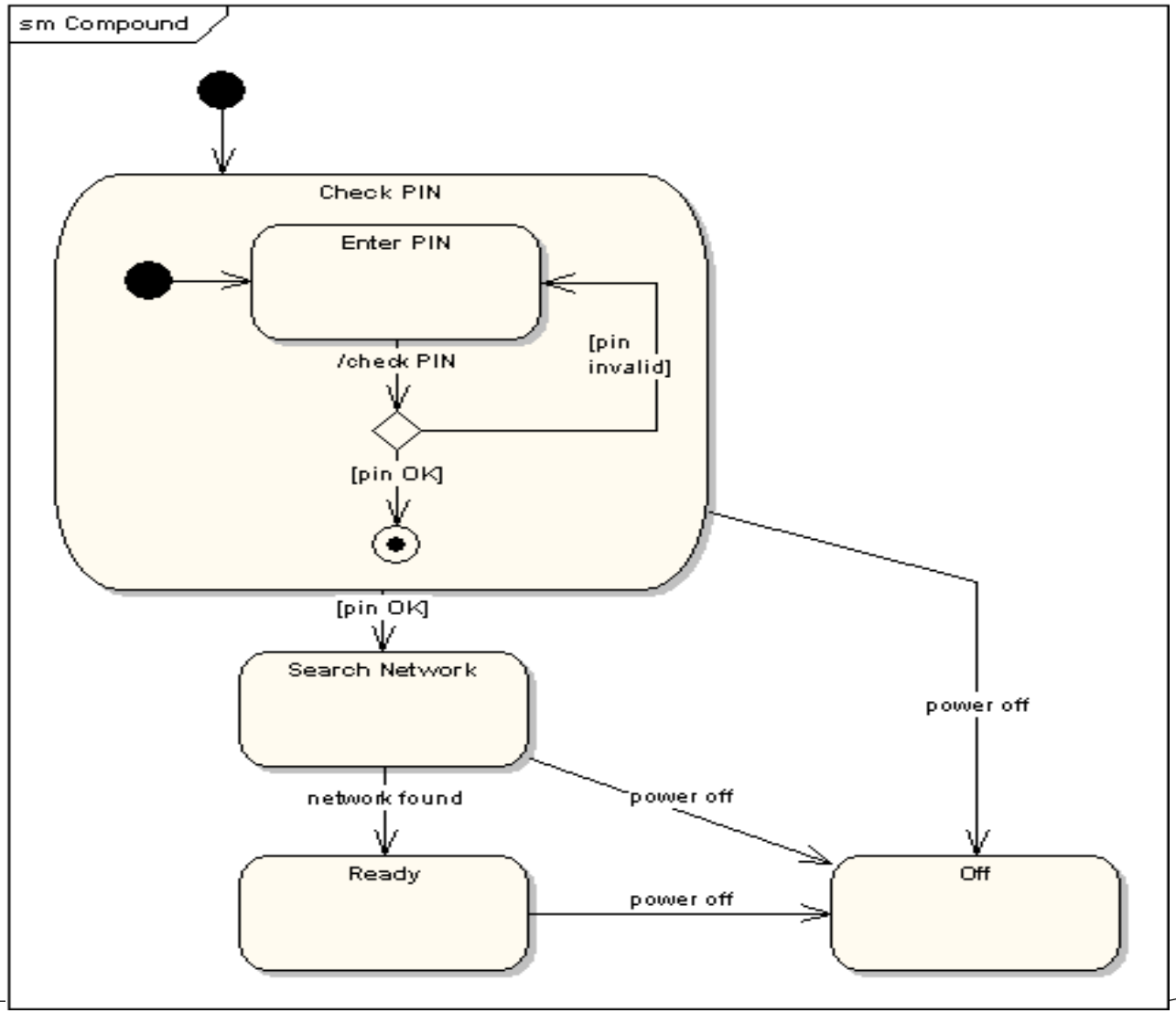
- A state can have a transition that returns to itself, as in the following diagram



State Machine Diagram

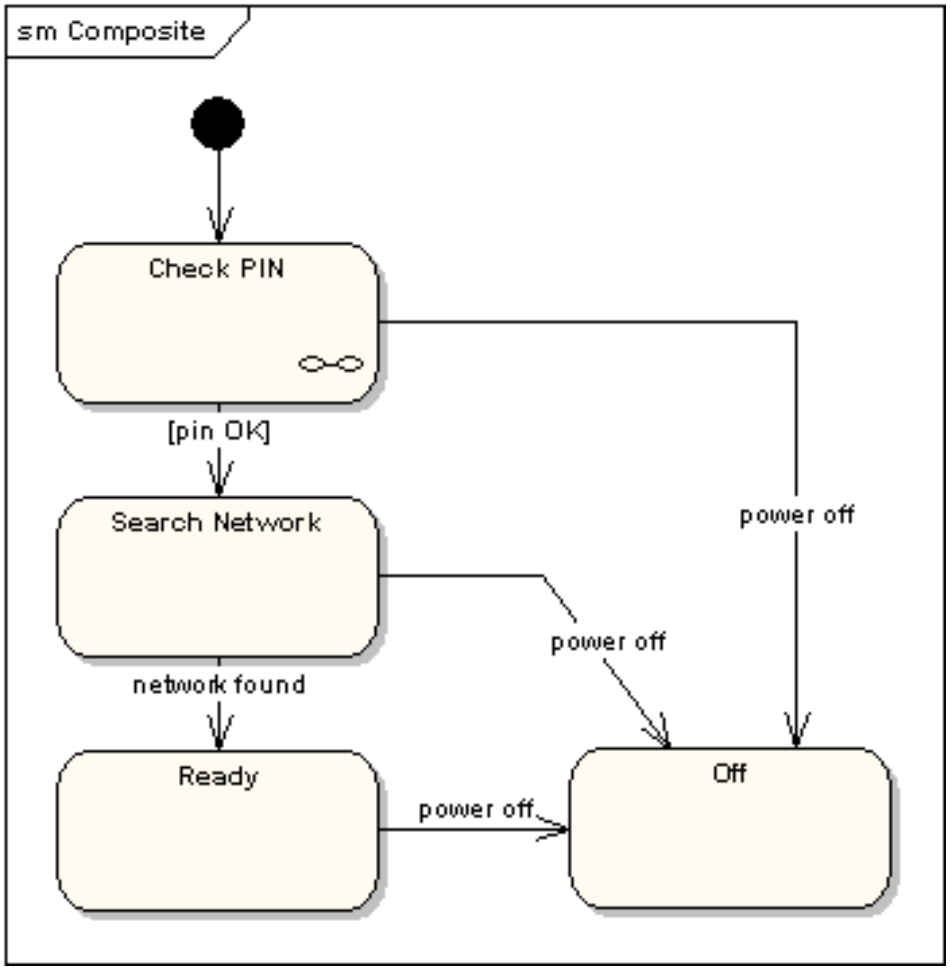
Compound States or Sub States

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



State Machine Diagram

- The alternative way to show the same information is as follows.

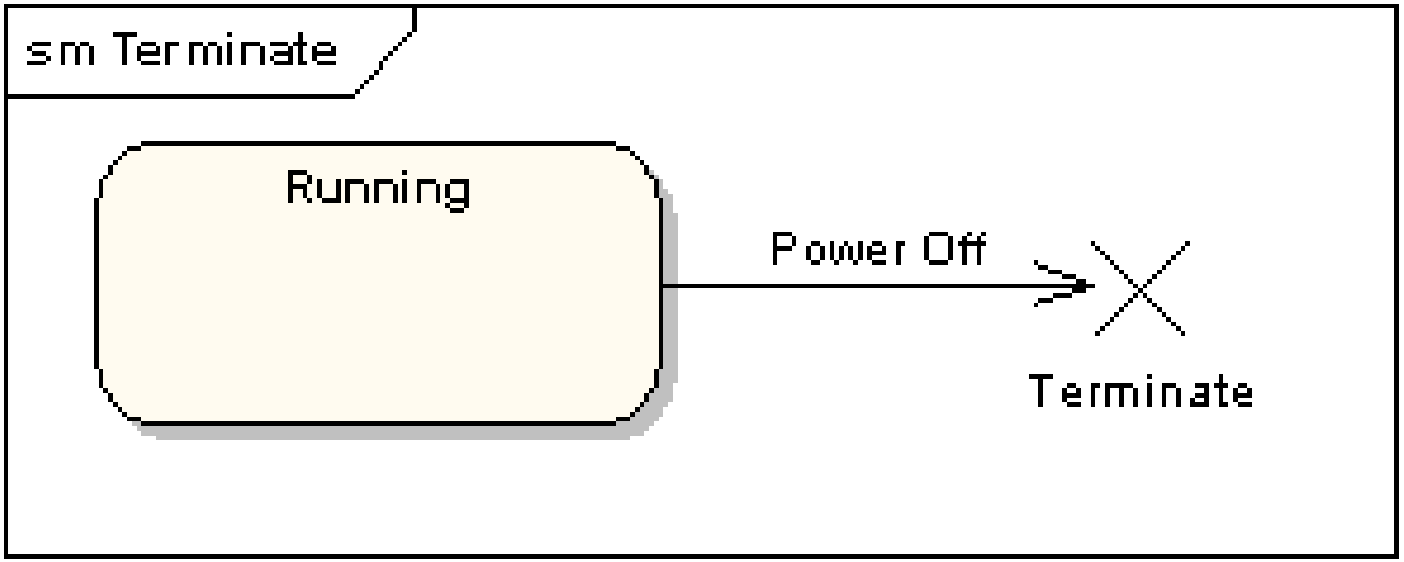


- The notation in above fig. indicates that the details of the Check PIN sub-machine are shown in a separate diagram.

State Machine Diagram

Terminate Pseudo-State

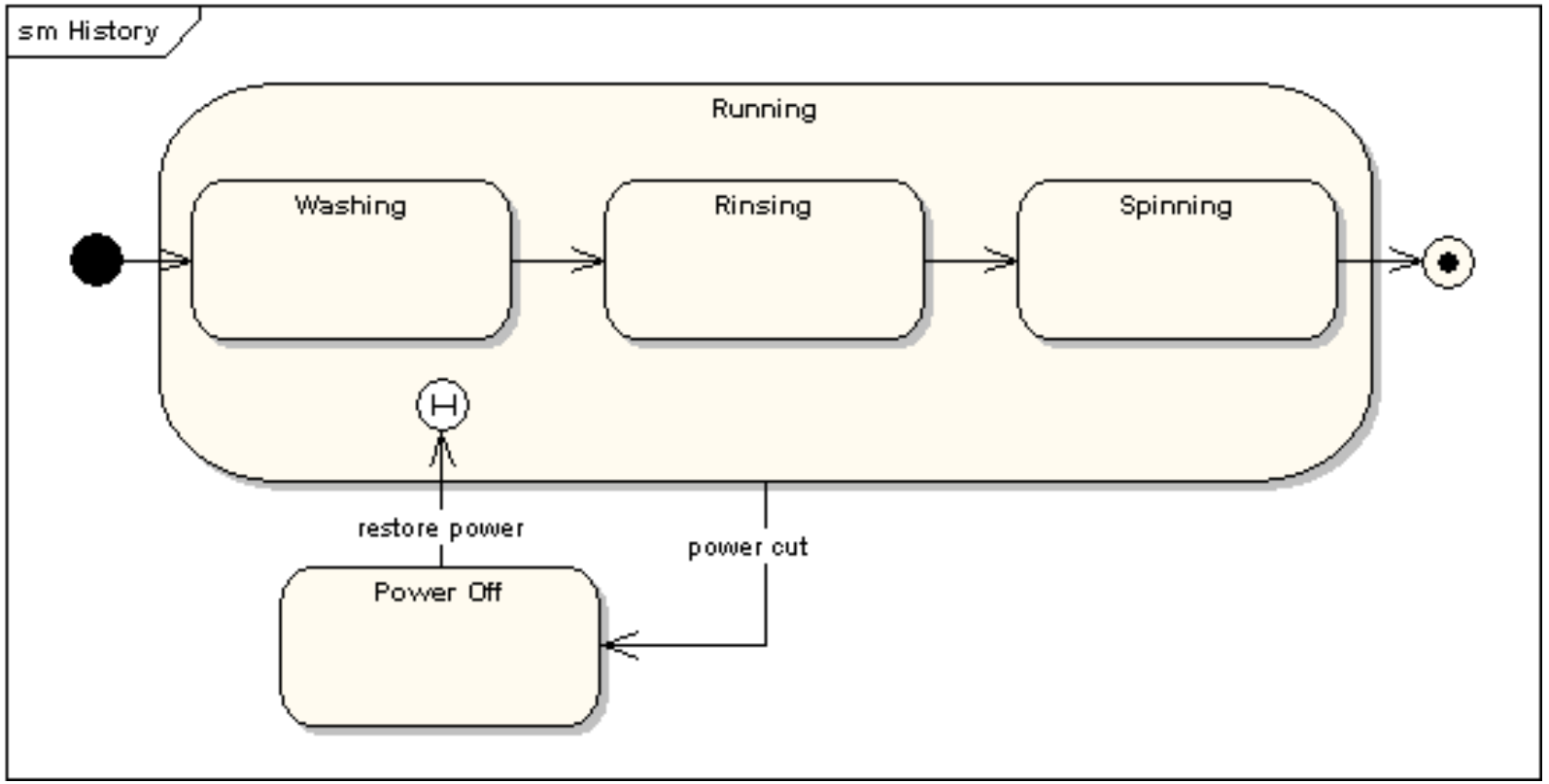
- Entering a terminate pseudo-state indicates that the lifeline of the state machine has ended.
- A terminate pseudo-state is notated as a cross.



State Machine Diagram

History States

- A history state is used to remember the previous state of a state machine when it was interrupted.



- The above diagram illustrates the use of history states. The example is a state machine belonging to a washing machine.

State Machine Diagram

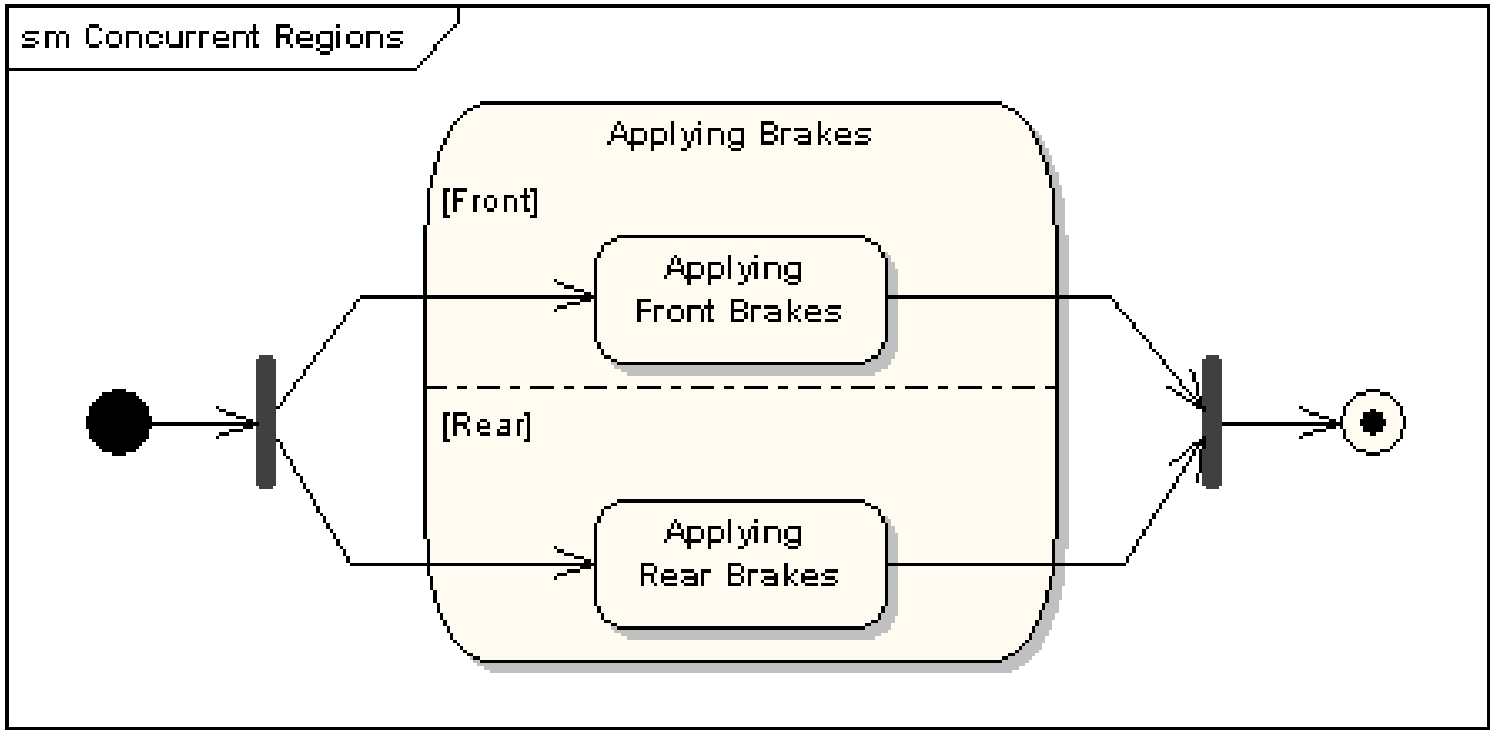
History States

- In this state machine shown in previous slide, when a washing machine is running, it will progress from "Washing" through "Rinsing" to "Spinning".
- If there is a power cut, the washing machine will stop running and will go to the "Power Off" state. Then when the power is restored, the Running state is entered at the "History State" symbol meaning that it should resume where it last left-off.

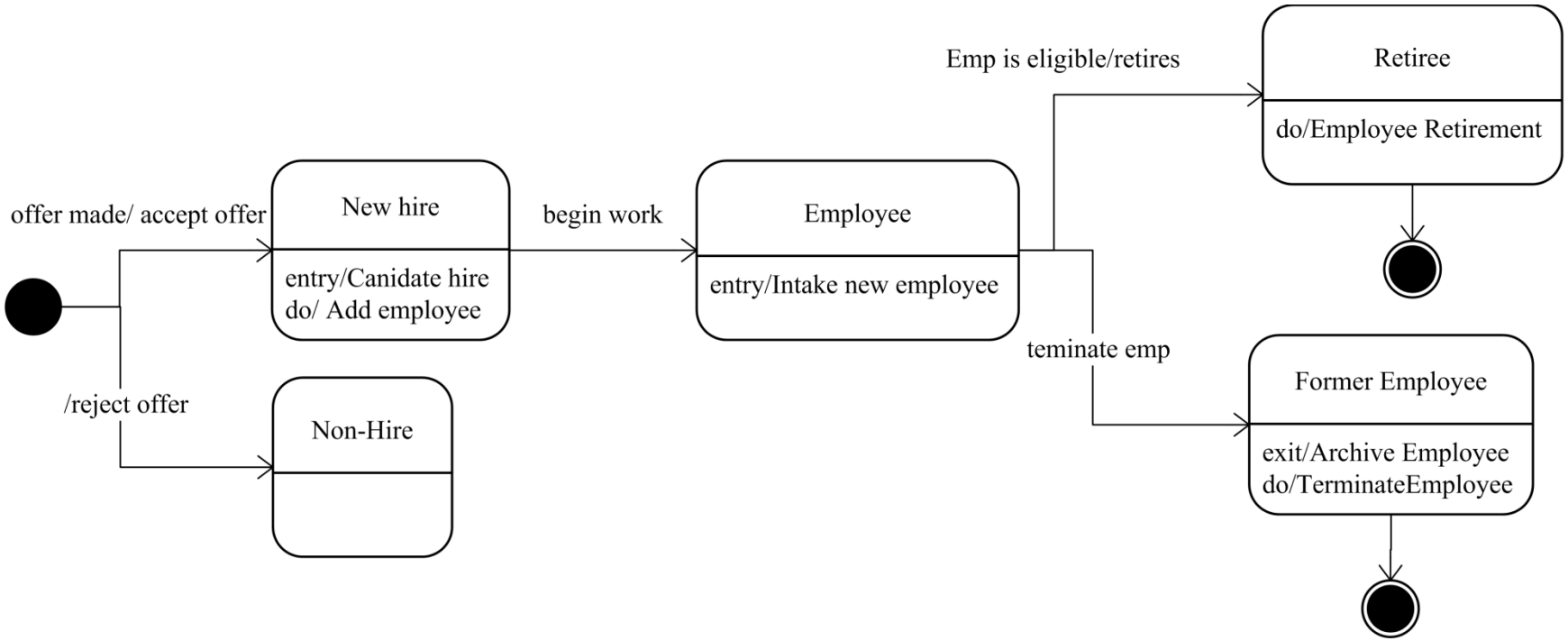
State Machine Diagram

Concurrent state machine

- A state may be divided into regions containing sub-states that exist and execute concurrently. The example below shows that within the state "Applying Brakes", the front and rear brakes will be operating simultaneously and independently. Notice the use of fork and join pseudo-states, rather than choice and merge pseudo-states. These symbols are used to synchronize the concurrent threads.

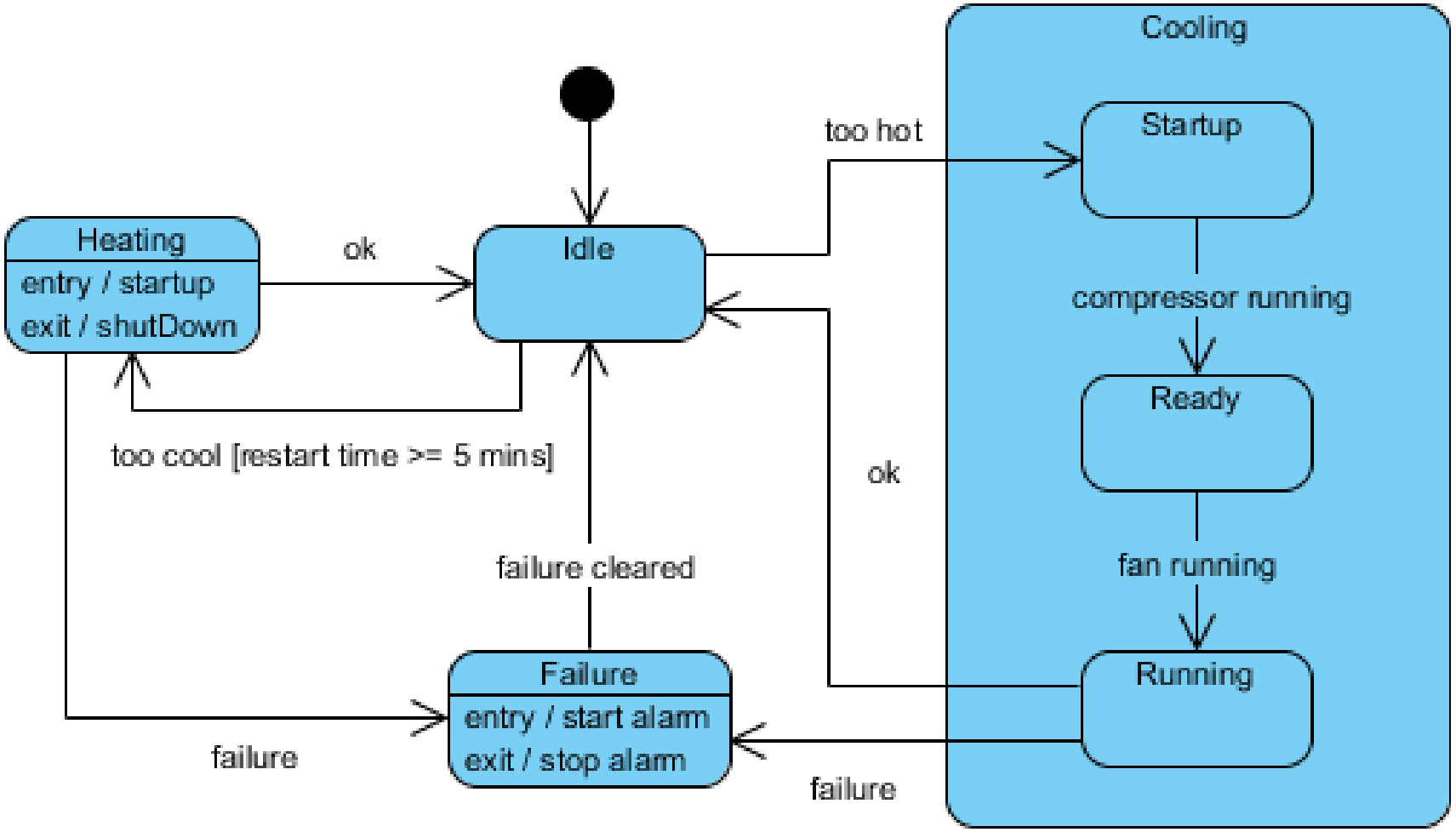


SMD of Recruitment process



State Machine Diagram

- **Sub states or compounded states example**



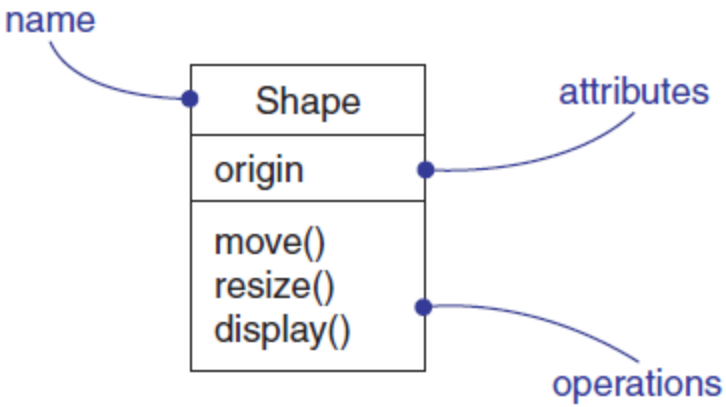
- **Class work (description)** **->Even roll -> Too Hot**

->Odd roll -> Too Cool

Class Diagram

Class Diagram

- A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Graphically, a class is rendered as a rectangle.
- This notation permits you to visualize an abstraction apart from any specific programming language and in a way that lets you emphasize the most important parts of an abstraction: its name, attributes, and operations.



Class Diagram

Class Name:

- Every class must have a name that distinguishes it from other classes. A name is a textual string. That name alone is known as a simple name; a qualified name is the class name prefixed by the name of the package in which that class lives. A class may be drawn showing only its name, as shown below.

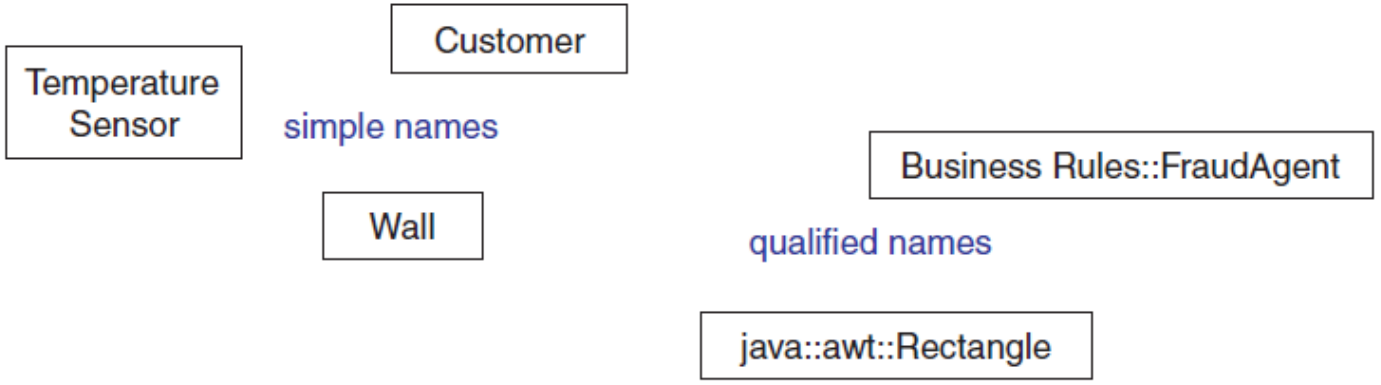


Fig: Simple and Quantified names

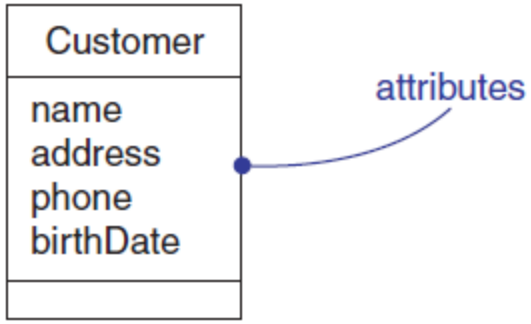
Notes:

A class name may be text consisting of any number of letters, numbers, and certain punctuation marks (except for marks such as the double colon, which is used to separate a class name and the name of its enclosing package) and may continue over several lines. In practice, class names are short nouns or noun phrases drawn from the vocabulary of the system you are modeling. Typically, you capitalize the first letter of every word in a class name, as in Customer or TemperatureSensor.

Class Diagram

Attributes:

- An attribute is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all.
- An attribute represents some property of the thing you are modeling that is shared by all objects of that class.
- For example, every wall has a height, width, and thickness; you might model our customers in such away that each has a name, address, phone number, and date of birth.



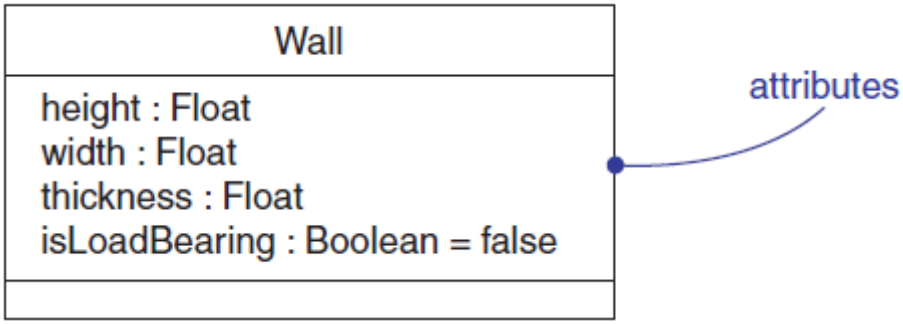
Note:

An attribute name may be text, just like a class name. In practice, an attribute name is a short noun or noun phrase that represents some property of its enclosing class.

Class Diagram

Attributes:

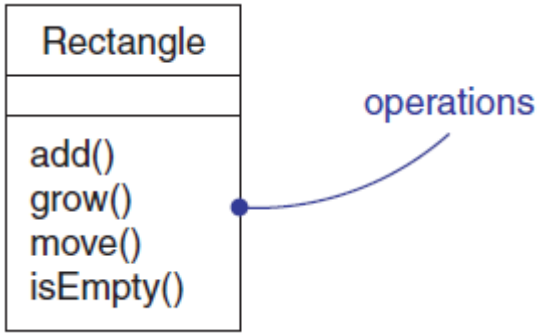
- we can further specify an attribute by stating its type and possibly a default initial value, as shown Figure below.



Class Diagram

Operations

- An operation is the implementation of a service that can be requested from any object of the class to affect behavior.
- In other words, an operation is an abstraction of something you can do to an object that is shared by all objects of that class.
- Operations may be drawn showing only their names, as in Figure below.



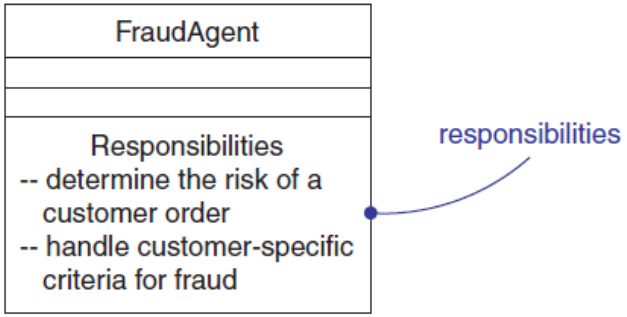
Note:

An operation name may be text, just like a class name. In practice, an operation name is a short verb or verb phrase that represents some behavior of its enclosing class.

Class Diagram

Responsibilities

- A responsibility is a contract or an obligation of a class.
- For example, A Wall class is responsible for knowing about height, width, and thickness; a FraudAgent class, as we might find in a credit card application, is responsible for processing orders and determining if they are legitimate, suspect, or fraudulent; a TemperatureSensor class is responsible for measuring temperature and raising an alarm if the temperature reaches a certain point.
- Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon, as shown below.



Note:

Responsibilities are just free-form text. In practice, a single responsibility is written as a phrase, a sentence, or (at most) a short paragraph.

Advance Class

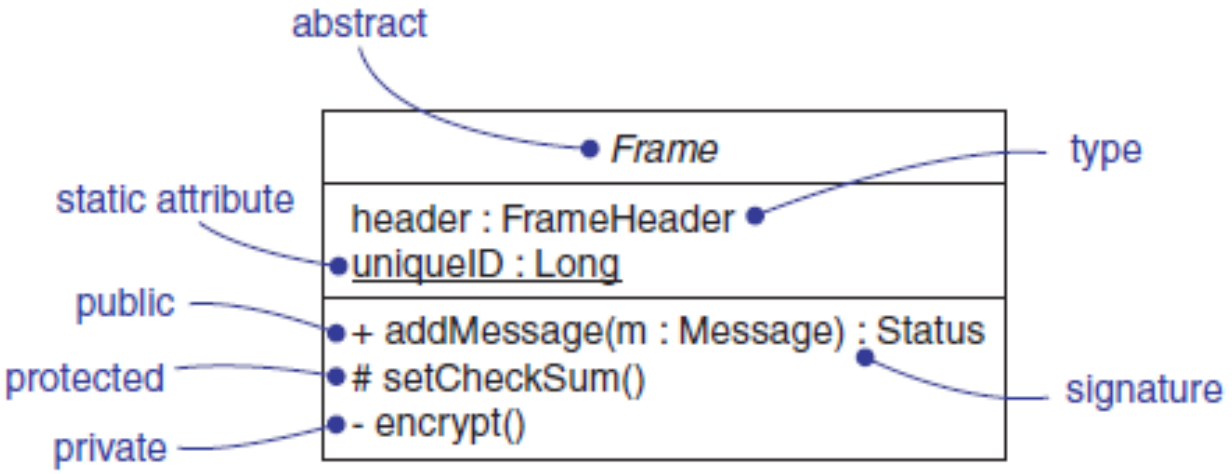
Advance class

Advance class

- Classifiers (and especially classes) have a number of advanced features beyond the simpler properties of attributes and operations described in the previous part: We can model multiplicity, visibility, signatures, polymorphism, and other characteristics.
- In the UML, we can model the semantics of a class so that you can state its meaning to whatever degree of formality you like.
- Early in a project, it's sufficient to say that we'll include a Customer class that carries out certain responsibilities.
- As we refine our architecture and move to construction, we'll have to decide on a structure for the class (its attributes) and a behavior (its operations) that are sufficient and necessary to carry out those responsibilities.
- Finally, as we evolve to the executable system, we'll need to model details, such as the visibility of individual attributes and operations, the concurrency semantics of the class as a whole and its individual operations, and the interfaces the class realizes.

Advance class

- The UML provides a representation for a number of advanced properties, as Figure below shows.
- This notation permits we to visualize, specify, construct, and document a class to any level of detail we wish.



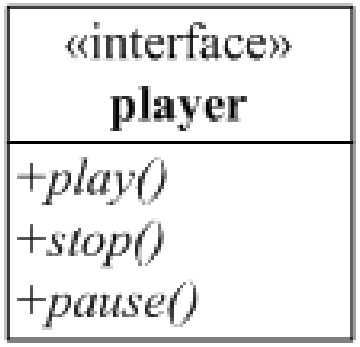
Advance class

Classifier

- A classifier is a mechanism which describes structural and behavioral features. The most important kind of classifier in the UML is the class.
- A classifier represents structural aspects in terms of properties and behavioral aspects in terms of operations. Beyond these basic features, there are several advanced features like multiplicity, visibility, signatures, polymorphism and others.
- *Not only the class, the UML provides a number of other kinds of classifiers to help we model.*

i. Interface

A collection of operations that are used to specify a service of a class or a component



Advance class

ii. Datatype

A type whose values are immutable, including primitive built-in types (such as numbers and strings) as well as enumeration types (such as Boolean).

iii. Association

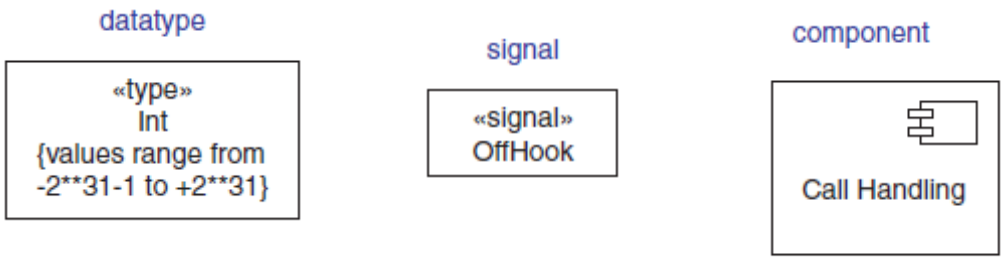
A description of a set of links, each of which relates two or more objects.

iv. Signal

The specification of an asynchronous message communicated between instances.

v. Component

A modular part of a system that hides its implementation behind a set of external interfaces.



Advance class

vi. Node

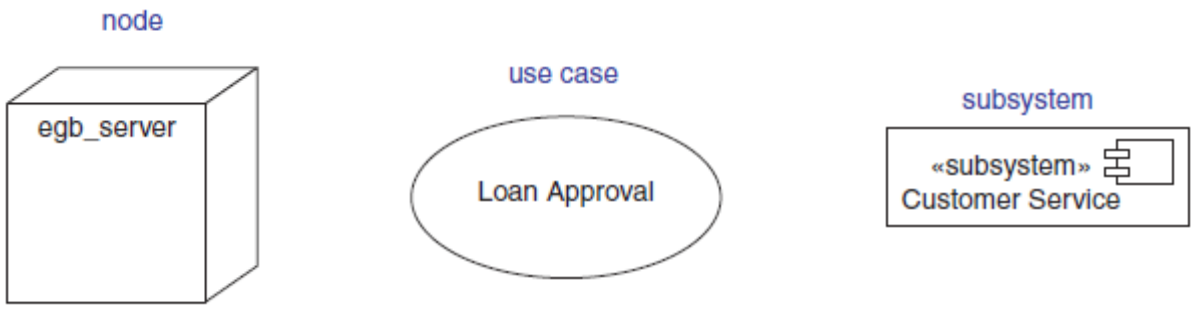
A physical element that exists at run time and that represents a computational resource, generally having at least some memory and often processing capability.

vii. Use case

A description of a set of a sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor

viii. Subsystem

A component that represents a major part of a system



Advance class

Visibility

The visibility of a feature specifies whether it can be used by other classifiers. In the UML, we can specify any of four levels of visibility.

1. Public

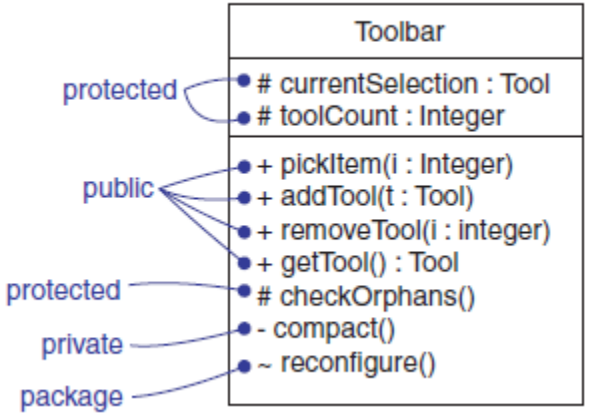
Any outside classifier with visibility to the given classifier can use the feature;

specified by prepending the symbol +.

2. Protected

Any descendant of the classifier can use the feature;

specified by prepending the symbol #.



3. Private

Only the classifier itself can use the feature;

specified by prepending the symbol -.

3. ***Package***

Only classifiers declared in the same package can use the feature;

specified by prepending the symbol ~.

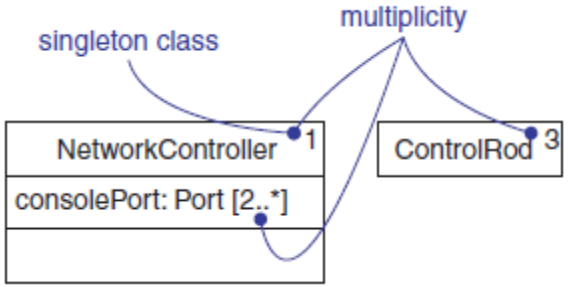
Note:

When we specify the visibility of a classifier's features, we generally want to hide all its implementation details and expose only those features that are necessary to carry out the responsibilities of the abstraction.

Advance class

Multiplicity

- The number of instances a class may have is called its multiplicity.
- In the UML, we can specify the multiplicity of a class by writing a multiplicity expression in the upper-right corner of the class icon
- For example, in Figure below, *NetworkController* is a singleton class (it can have *exactly one instance)*. Similarly, there are exactly three instances of the class *ControlRod* in the system.
- Multiplicity applies to attributes, as well. we can specify the multiplicity of an attribute by writing a suitable expression in brackets just after the attribute name.
- For example, in the figure, there are two or more *consolePort* instances in the instance of *NetworkController*.



Advance class

Attributes:

- At the most abstract level we simply write each *attribute's name*. That's usually enough information for the average reader to understand the intent of our model.
- However, We can also specify the visibility, scope, and multiplicity of each attribute. There's still more. We can also specify the type, initial value, and changeability of each attribute.
- For example, the following are all legal attribute declarations:
  - origin ←Name only
  - + origin ←Visibility and name
  - origin : Point ←Name and type
  - name : String[0..1] ←Name, type, and multiplicity
  - origin : Point = (0,0) ←Name, type, and initial value
  - id: Integer {readonly} ←Name and property

Unless otherwise specified, attributes are always changeable. we can use the *readonly* property to indicate that the attribute's value may not be changed after the object is initialized.

Advance class

Operations:

- At the most abstract level, we will simply write each operation's name. That's usually enough information for the average reader to understand the intent of our model.
- however, we can also specify the visibility and scope of each operation. There's still more: we can also specify the parameters, return type, concurrency semantics, and other properties of each operation. Collectively, the name of an operation plus its parameters (including its return type, if any) is called the operation's signature.

For example, the following are all legal operation declarations:

- display ←Name only
- + display ←Visibility and name
- set(n : Name, s : String) ←Name and parameters
- getID() : Integer ←Name and return type
- restart() {guarded} ←Name and property

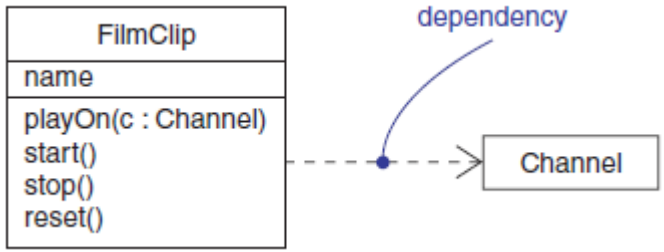
Class Relationships

Class Relationships

A relationship is a connection among things. In object-oriented modeling, the three most important relationships are dependencies, generalizations, and associations.

Dependencies

- A dependency is a relationship that states that one thing (for example, class Window) uses the information and services of another thing (for example, class Event), but not necessarily the reverse.
- Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on.



Class Relationships

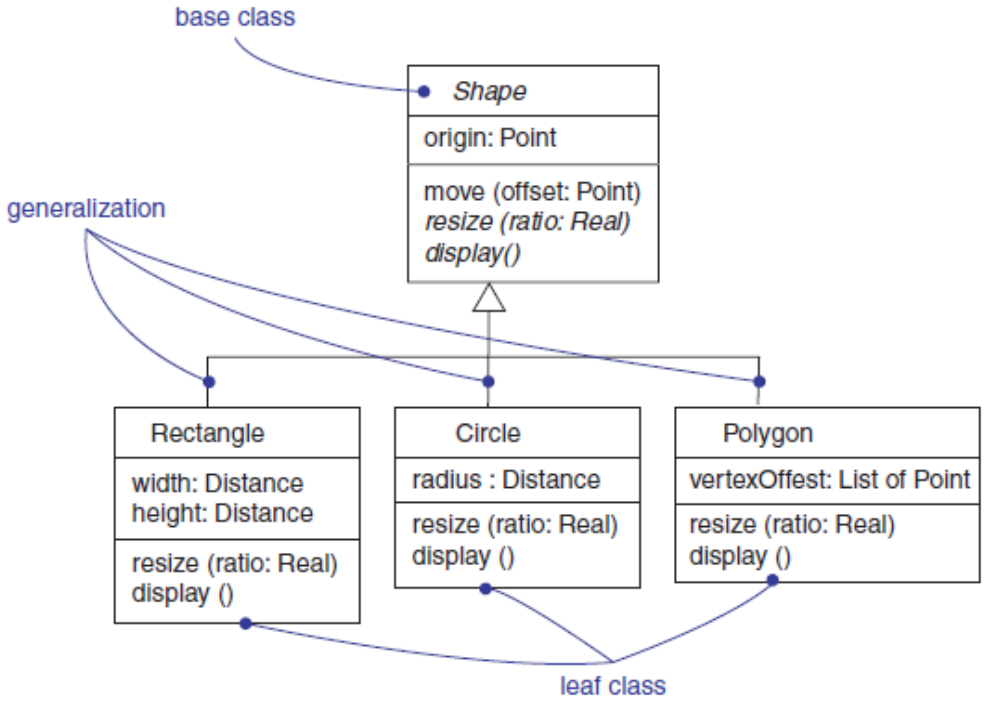
Dependencies

- Most often, we will use dependencies between classes to show that one class uses operations from another class or it uses variables or arguments typed by the other class..
- This is very much a using relationship—if the used class changes, the operation of the other class may be affected as well, because the used class may now present a different interface or behavior.
- In the UML we can also create dependencies among many other things, especially notes and packages.

Class Relationships

Generalizations

- A generalization is a relationship between a general kind of thing (called the superclass or parent) and a more specific kind of thing (called the subclass or child).
- Graphically, generalization is rendered as a solid directed line with a large unfilled triangular arrowhead, pointing to the parent, as shown below.
- Use generalizations when we want to show parent/child relationships.



Class Relationships

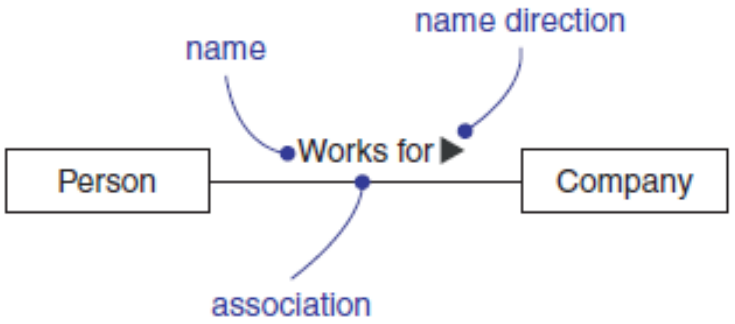
Associations

- An association is a structural relationship that specifies that objects of one thing are connected to objects of another.
- Given an association connecting two classes, we can relate objects of one class to objects of the other class.
- An association that connects exactly two classes is called a *binary* association. Although it's not as common, we can have associations that connect more than two classes; these are called *n-ary* associations.
- Graphically, an association is rendered as a solid line connecting the same or different classes. Use associations when we want to show structural relationships.
- Beyond this basic form, there are four adornments that apply to associations.
  - ✓ Name
  - ✓ Role
  - ✓ Multiplicity
  - ✓ Aggregation

Class Relationships → Associations

Name

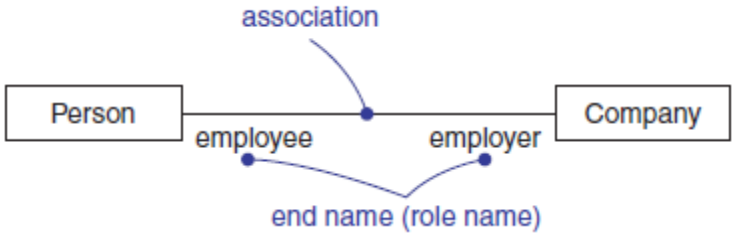
- An association can have a name, and we use that name to describe the nature of the relationship. So that there is no ambiguity about its meaning
- Also we can give a direction to the name by providing a direction triangle that points in the direction we intend to read the name, as shown in Figure below.



Class Relationships → Associations

Role

- When a class participates in an association, it has a specific role that it plays in that relationship; a role is just the face the class at the far end of the association presents to the class at the near end of the association.
- we can explicitly name the role a class plays in an association. The role played by an end of an association is called an end name.
- As in Figure below, *the class Person playing the role of employee is associated with the class Company playing the role of employer.*



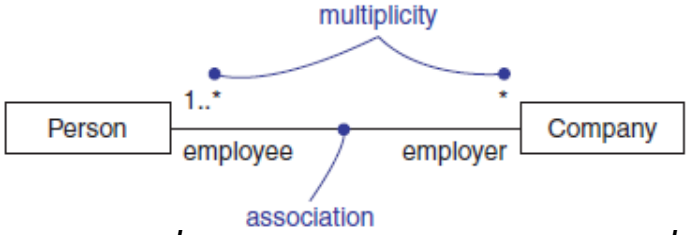
Note:

The same class can play the same or different roles in other associations.

Class Relationships → Associations

Multiplicity

- In many modeling situations, it's important for we to state how many objects may be connected across an instance of an association. This "how many" is called the multiplicity of an association's role.
- It represents a range of integers specifying the possible size of the set of related objects. It is written as an expression with a minimum and maximum value, which may be the same; two dots are used to separate the minimum and maximum values.
- *The number of objects must be in the given range. we can show a multiplicity of as:*
  - ✓ *exactly one (1)*
  - ✓ *zero or one (0..1)*
  - ✓ *many (0..*), or one or more (1..*).*
  - ✓ *integer range (such as 2..5)*
  - ✓ *exact number (for example, 3, which is equivalent to 3..3).*

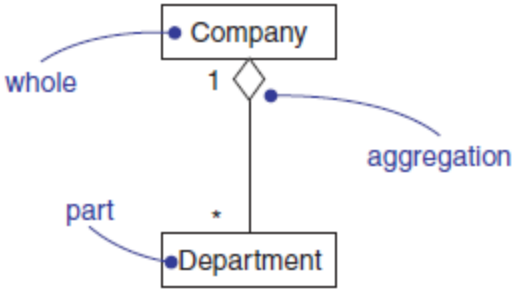


In above figure, each company object has as employee one or more person objects (multiplicity 1..); each person object has as employer zero or more company objects (multiplicity *, which is equivalent to 0..*).*

Class Relationships → Associations

Aggregation

- A plain association between two classes represents a structural relationship between peers, meaning that both classes are conceptually at the same level, no one more important than the other.
- Sometimes we will want to model a “whole/part” relationship, in which one class represents a larger thing (the “whole”), which consists of smaller things (the “parts”). This kind of relationship is called aggregation, which represents a “has-a” relationship, meaning that an object of the whole has objects of the part.
- Aggregation is really just a special kind of association and is specified by adorning a plain association with an unfilled diamond at the whole end, as shown in Figure below.



Advance Relationships

→*Assignment*

→*See book G. Booch, J. Rumbaugh, I Jacobson/ pg. 133*