2.1	Introduction to OO Methodologies :Booch,Rambaug and Jacobson
2.2	Requirements Engineering Tasks, Requirement Elicitation Techniques, Software Requirements: Functional, Non-Functional, Domain **Bernd Bruegge Chapter 4**
2.3	Requirements Characteristics, Requirement qualities, Requirement Specification, Requirement Traceability, System Analysis Model Generation, Requirement Prioritization, Documentation : Use Case Diagram,Acitvity Diagram **Bernd Bruegge Chp 4 & Chp 2**
2.4	Categorizing classes: entity, boundary and control ,Modeling associations and collections-Class Diagram
2.5	Dynamic Analysis - Identifying Interaction – Sequence and Collaboration diagrams, State chart diagram

OBJECT ORIENTED PROCESS

- The Object Oriented Methodology of Building Systems takes the **objects as the basis**.

- First the system to be developed is **observed and analyzed and the requirements are defined** as in any other method of system development.

- Objects are identified.

  - Example: Banking System: customer is an object.

- Object Modeling is based on identifying the objects in a system and their interrelationships.

- The basic steps of system designing using Object Modeling may be listed as:

  - System Analysis

  - System Design

  - Object Design

  - Implementation

Advantages

1. Object Oriented Methodology **closely represents the problem domain**.

   Because of this, it is easier to produce and understand designs.

2. The objects in the system are **immune to requirement changes**.

   Therefore, allows changes more easily

3. Object Oriented Methodology designs encourage **more re-use**.

New applications can use the existing modules, thereby reduces the development cost and cycle time.

4. The systems designed using this approach are **closer to the real world** as the real world functioning of the system is directly mapped into the system designed using this approach

Class and Object

- The concepts of **objects and classes** are **intrinsically linked** with each other and form the foundation of object–oriented paradigm.

- An object is a **real-world element in an object–oriented environment** that may have a physical or a conceptual existence.

  - Object is an entity that has identity , state and behavior.

Class and Object

- Objects can be **modeled according to the needs of the application.**
  - An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

  - A class represents a collection of objects having same characteristic properties that exhibit common behavior.

- Creation of an object as a member of a class is called instantiation. Thus, object is an **instance of a class.**

THE RAMBAUGH OMT

- The **object-modeling technique** (**OMT**) is an object modeling language for software modeling and designing.

- The object modeling technique(OMT) presented by Jim Rambaugh and his coworkers describes a method for analysis, design, and **implementation of a system using an object –oriented technique.**

- OMT is a **fast and intuitive(easy or simple) approach** for identifying and modeling all the objects making up a system.

THE RAMBAUGH OMT

- The **purposes** of modeling according to Rambaugh are:
  - testing physical entities before building them (simulation),
  - communication with customers,
  - visualization (alternative presentation of information), and
  - reduction of complexity.

THE RAMBAUGH OMT

The Rambaugh OMT has proposed three main types of models:

- ***Object model*** : Main concepts are **classes and associations, with attributes and operations**. **Aggregation and generalization** are predefined relationships.

- ***Dynamic model*** : The dynamic model represents a **state/transition view** on the model. Main concepts are states, transitions between states, and events to trigger transitions. Actions can be modeled as occurring within states.

THE RAMBAUGH OMT

- ***<u>Functional model</u>*** : The functional model handles the **process of the model**, corresponding roughly to **data flow diagrams**. Main concepts are process, data store, data flow, and actors.

OMT is a predecessor of the Unified Modeling Language (UML).

THE BOOCH METHODOLOGY

- The Booch methodology is a widely used **object-oriented method** that helps you **design your system using the object paradigm**.

- It covers **analysis and design phases** of an object-oriented system.

ANALYSIS PHASE

- **Customer's Requirements**

- **Domain analysis**
  - The domain analysis is done by **defining object classes**; their attributes, inheritance, and methods. State diagrams for the objects are then established.

- The analysis phase is completed with a **validation step**.

- The analysis phase iterates between the customer's requirements step, the domain analysis step, and the validation step until consistency is reached.

DESIGN PHASE

- The design phase is **iterative**.

- A **logic design is mapped to a physical design** like processes, performance, data types, data structures, visibility are defined.

- A prototype is created and **tested**.
  - The process iterates between the logical design, physical design, prototypes, and testing.

- The Booch software engineering methodology is **sequential in the sense that the analysis phase is completed and then the design phase** is completed.

THE BOOCH METHODOLOGY

- The Booch method prescribes a macro development and a micro development process.

- **Macro process is high level process** describing activities of development team as whole.

- **Micro process is low level process** describing technical activities of development team.

THE BOOCH METHODOLOGY

- The Booch methodology concentrates on the analysis and design phase and **does not consider the implementation or the testing** phase in much detail.

JACOBSON OOSE

- Object-Oriented Software Engineering (OOSE) is a software design technique that is used in software design in object-oriented programming.

- OOSE is the first object-oriented design methodology that employs **use cases in software design**.

- It includes requirements, an analysis, a design, an implementation and a testing model.

Requirement Elicitation (Gathering)

- **A requirement** is a **feature** that the system must have or a **CONSTRAINT** that it must satisfy to be **accepted by the client.**

- **Requirements engineering** aims at **defining** the **requirements** of the "**SYSTEM UNDER CONSTRUCTION**"

- **Requirements engineering includes two main activities:**

    - requirements elicitation: specification of system that client understands.

    - Analysis: analysis model that developers can interpret.

Requirement Elicitation

- **Requirements elicitation** is about **communication** among **developers** and **users** to define a new system.

- **Failure to communicate** and understand each other - **system fails**

- **Errors** introduced during **requirements elicitation** are "**expensive" to correct**, as they are usually discovered late in the process, often as late as delivery.

Requirement Elicitation

- Requirements elicitation methods **aim at** **improving** **communication** among developers, clients, and users.

- **Requirement Specification**

  - The client, the developers, and the users **identify a problem area** and **define a system** that **addresses the problem**.
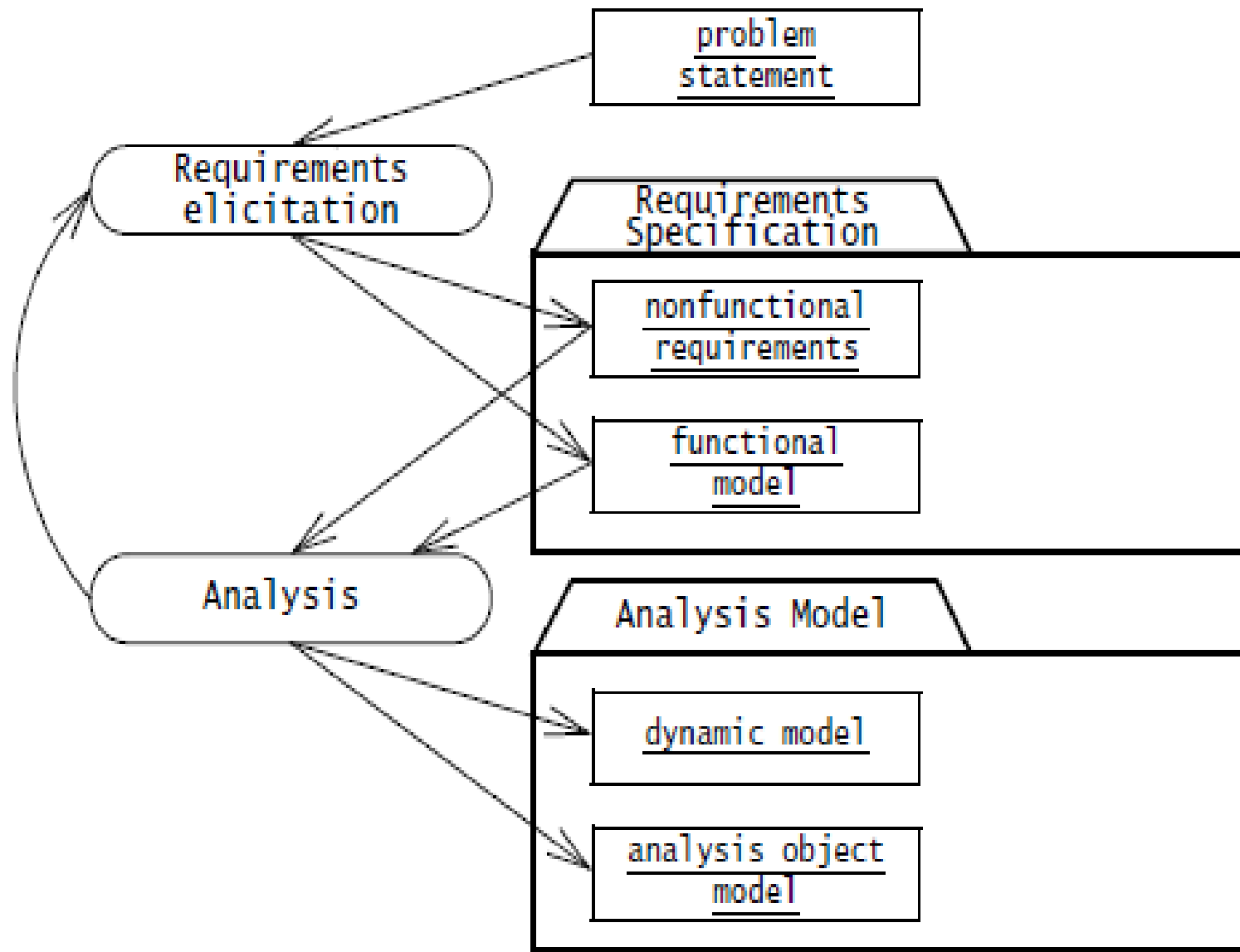
Figure 4-1 Products of requirements elicitation and analysis (UML activity diagram).

Requirements Elicitation Activities

1. IDENTIFYING ACTORS: developers **identify the different types of users** the future system will support.

2. IDENTIFYING SCENARIOS:

- developers **observe users** and develop a set of detailed scenarios (**concrete examples of future systems**) for typical <u>**functionality provided by the future system**</u>.

- Scenarios used for understanding application domain in detail.

Requirements Elicitation Activities

3. IDENTIFYING USE CASES:

- Once developers and users agree on a set of scenarios.

- Set of use cases are derived from scenarios that represent future systems.

- Use cases are a**bstractions** describing all possible cases of the system

Requirements Elicitation Activities

4. REFINING USE CASES:

- **Requirements specification is complete** by detailing each use case.

- Describing the **behavior of the system** in the presence of **errors** and exceptional conditions.

Requirements Elicitation Activities

5. IDENTIFYING RELATIONSHIPS AMONG USE CASES:

- identify **dependencies** among use cases.

- Consolidate(combine) the use case model by factoring out common functionality.

Requirements Elicitation Activities

6. IDENTIFYING NONFUNCTIONAL REQUIREMENTS:

- developers, users, and clients agree on aspects that are visible to the user, **but not directly related to functionality.**

- These include
  - constraints on the performance of the system,
  - its documentation,
  - the resources it consumes,
  - its security, and
  - its quality.

REQUIREMENT ELICITATION TECHNIQUES

- **Questionnaires:** Asking the end user a list of pre-selected questions

- **Task Analysis:** Observing end users in their operational environment

- **Scenarios:** Describe the use of the system as a series of interactions between a concrete end user and the system

- **Use cases:** Abstractions that describe a class of scenarios.

Satellite GPS Timekeeping
Technology with Synchronized
Time Adjustment Available in
40 Time Zones

Luminous Hands
and Markers

Power Reserve
Indicator

Day

Anti-Reflective
Sapphire Crystal

World's Fastest* Timekeeping
Signal Reception Speed From
GPS Navigation Satellites
– as Quick as 3 Seconds**

1/20-Second Chronograph
Measures up to 24 Hours

UTC Display

Alarm

Light Level
Indicator

Perpetual Calendar

Dual Time

Super Titanium™
5x Harder and 40% Lighter
Than Stainless Steel

CITIZEN
Eco-Drive
SATELLITE WAVE GPS

*As of March 2015 according to Citizen Watch Co., LTD.
**Speeds as quickly as 30 seconds when receiving position information.

Quality requirements for SatWatch

- Any user who knows how to read a digital watch and understands international time zone abbreviations should be able to use SatWatch without the user manual. [Usability requirement]
- As the SatWatch has no buttons, no software faults requiring the resetting of the watch should occur. [Reliability requirement]
- SatWatch should display the correct time zone within 5 minutes of the end of a GPS blackout period. [Performance requirement]
- SatWatch should measure time within 1/100th second over 5 years. [Performance requirement]
- SatWatch should display time correctly in all 24 time zones. [Performance requirement]
- SatWatch should accept upgrades to its onboard via the Webify Watch serial interface. [Supportability requirement]

Constraints for SatWatch

- All related software associated with SatWatch, including the onboard software, will be written using Java, to comply with current company policy. [Implementation requirement]
- SatWatch complies with the physical, electrical, and software interfaces defined by WebifyWatch API 2.0. [Interface requirement]

Non-Functional Requirements

- **Nonfunctional requirements** describe aspects of the system that are **not** directly **related** to the **functional behavior** of the system.

Non-Functional Requirements (Types)

1. Usability is the ease with which a user can learn to operate or use system.

– Usability requirements: online help or user level documentation , manual etc.

– Clients can set usability issues for developer for example – color scheme for GUI

Non-Functional Requirements

2. Reliability is the ability of a system or component to **perform its required functions** **under stated conditions** for a **specified period** of time.

- Reliability is replaced with **dependability** which includes other features like **robustness and safety**

Non-Functional Requirements (Types)

3. Performance requirements are concerned with quantifiable attributes of the system such as:

- **Response Time**

time taken for a person or system to react to a given stimulus or event.

- **Throughput**

how many units of information are processed in a given time

- **Availability**

The quality of being able to be used or obtained.

- **Accuracy**

 quality or state of being correct or precise.

Non-Functional Requirements (Types)

4. **Supportability** requirements are concerned with the <u>**ease of changes to the system after deployment**</u>.

- **Adaptability**: the ability to change the system to deal with **additional application domain concepts**.

- **Maintainability**: the ability to **change the system** to deal with new technology or to **fix defects**.

- **Internationalization:** the ability to **change the system** to deal with **additional international conventions**, such as languages, units, and number formats

Additional Non-functional Requirements

- **Implementation requirements**
  - constraints on the **implementation** of the system.
  - specific tools, programming languages, or hardware platforms.

- **Interface requirements**
  - are constraints **imposed by external systems**
  - interchange formats

Additional Non-functional Requirements

- **Operations requirements**
  - are constraints on the **administration** and **management** of the system.

- **Packaging requirements**
  - are constraints on the **actual delivery** of the system
  - Installation media for software setting

Additional Non-functional Requirements

- **Legal requirements**
  - are concerned with licensing, regulation, and certification issues.

- **Quality requirements**

Requirements Characteristics

➢ Requirements are continuously validated with the client and the user.

➢ Requirement validation involves checking that the specification is
  ➢ complete,
  ➢ consistent,
  ➢ unambiguous, and
  ➢ correct.

Requirements Characteristics

1. **COMPLETENESS**: It is complete if all possible **scenarios through the system are described**, including **exceptional behavior**.

    – Complete—All **features of interest** are **described** by requirements.

    – **Example of incompleteness**: Satwatch's limitation of state boundary.

    – **Solution**: addition of the boundary feature.

Requirements Characteristics

2. CONSISTENT: The requirements specification is consistent if it **does not contradict itself**.

- **Consistent**—No two requirements of the specification contradict each other.

- **Example of inconsistency**: Software fault and upgrading mechanisms for new versions.

- **Solution**: Revise one of the conflicting requirements

Requirements Characteristics

3. UNAMBIGUOUS — The requirements specification is unambiguous if **exactly one system is defined**.

– A requirement **cannot be interpreted in two mutually exclusive** (either) ways.

– **Example of ambiguity**: time zones and political boundaries.
   Does SatWatch deal with daylight saving time or not?

– **Solution**: add a requirement that SatWatch should deal with "**daylight saving time**"

Requirements Characteristics

4. CORRECT: represents accurately the system that the client needs and that the developers intend to build.

– Correct—The requirements **describe the features of the system and environment of interest** to the client and the developer, but do **not describe other unintended features**.

Requirement Specification

Three more **desirable properties** (qualities) of a requirements specification are that it be **realistic, verifiable, and traceable.**

- **REALISTIC**: if the system can be **implemented within constraints**.

- **VERIFIABLE**: if once the system is built, **repeatable tests** designed to demonstrate that the system **fulfills the requirements** specification.

Requirement Specification

- **TRACEABILITY**: if each requirement can be traced throughout the **software development to its system functions**, and vice-versa.

- Traceability includes also the **ability to track** the dependencies among requirements, system functions, and the intermediate designs.

USE CASE DIAGRAM

The purpose of a use case diagram in UML is to

demonstrate the different ways that a user might

interact with a system

- **Summarize** the details of your system's users (also known as actors) and their interactions with the system.

- Set of **specialized symbols and connectors.**

- An effective use case diagram can help your team discuss and represent:

  - **Scenarios** in which your system or application interacts with people, organizations, or external systems

  - **Goals** that your system or application helps those entities (known as actors) achieve

  - The **scope** of your system

UML use case diagrams are ideal for:

- Representing the **goals** of system-user interactions

- Defining and organizing **functional requirements** in a system

- Specifying the **context and requirements** of a system

- Modeling the **basic flow** of events in a use case

Identifying Actors

– Actors **represent external entities** that interact with the system.

– An actor can be **human or an external system**.
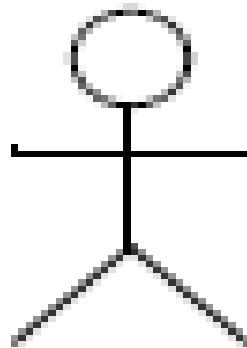
Use Case Diagrams

- **Give meaningful business relevant names for actors**.

- **Primary actors should be to the left side of the diagram** –highlight the important roles in the system.

- **Actors model roles (not positions)** – In a hotel both the front office executive and shift manager can make reservations. "Reservation Agent" actor name to highlight the role.

- **External systems are actors** – If your use case is send email and if interacts with the email management software then the software is an actor to that particular use case.

- Actors don't interact with other actors

- **Place inheriting actors below the parent actor**

Use Case Diagram

- Once the **actors are identified**, the next step in the requirements elicitation activity is to **determine the functionality** that will be accessible to each actor.

- **Use cases are external (user) view of a system**

- **Intended for modeling dialogue between user and system**

Identifying Scenarios

– A scenario is a **narrative description** of what people do and experience as they try to make use of computer systems and applications.

– Scenario is **description** of the system from **single actor's** or user's **point of view.**

– Scenarios may be **defined for current , future , evaluation or training phases** of the software development process.

Identifying Use Cases

– Use cases represent what the actors want your system to do for them.

– A use case represents a complete flow of events through the system.

– A Use Case captures some actor-visible function

– - Achieves some discrete (business-level) goal for that actor

– - May be read, write, or read-modify-write in nature

Use Case Diagram

- **Names begin with a verb** – An use case models an action so the name should begin with a verb.

- **Make the name descriptive** – This is to give more information for others who are looking at the diagram. For example "Print Invoice" is better than "Print".

- **Highlight the logical order** – For example if you're analyzing a bank customer typical use cases include open account, deposit and withdraw. Showing them in the logical order makes more sense.

- **Place included use cases to the right of the invoking use case** – This is done to improve readability and add clarity.

- **Place inheriting use case below parent use case** – Again this is done to improve the readability of the diagram.
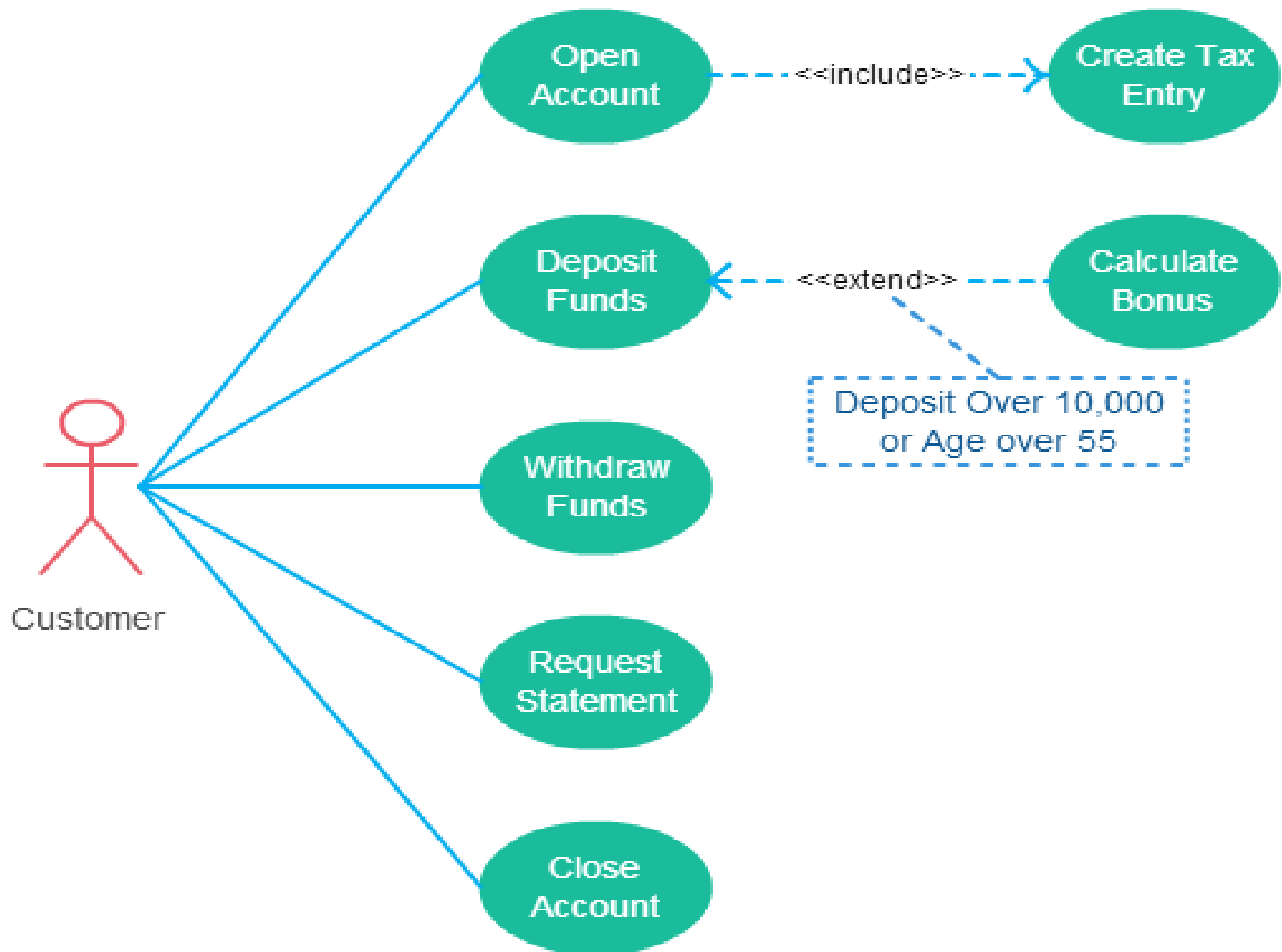
Elements of use case diagram: Actor

Actor
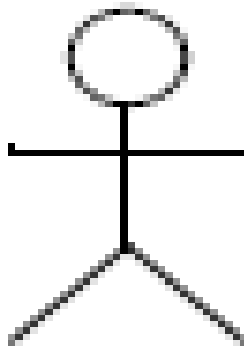
Elements of use case diagram: Use Case

• System function (process – automated or manual).

Do Something
(Operation)
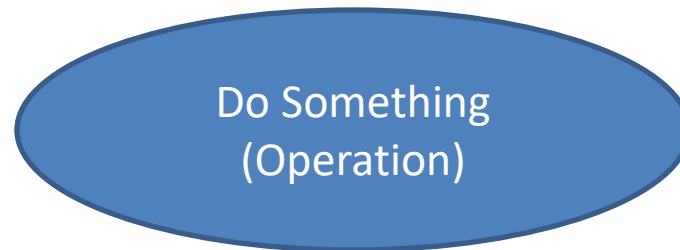
Use Case

Elements of use case diagram

——————— Connection between Actor and Use Case
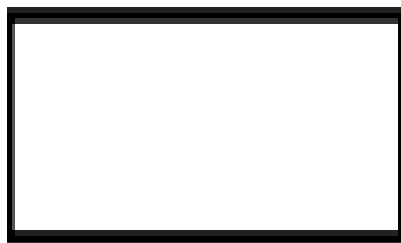
Boundary of system

There can be 5 relationship types in a use case diagram.

- Association between actor and use case
- Generalization of an actor
- Extend between two use cases
- Include between two use cases
- Generalization of a use case

Association Between Actor and Use Case

- This one is straightforward and present in every use case diagram.

  - An actor must be **associated with at least one use case.**

  - An actor can be **associated with multiple use cases.**

  - **Multiple** actors can be **associated with a single use case.**

Association Between Actor and Use Case
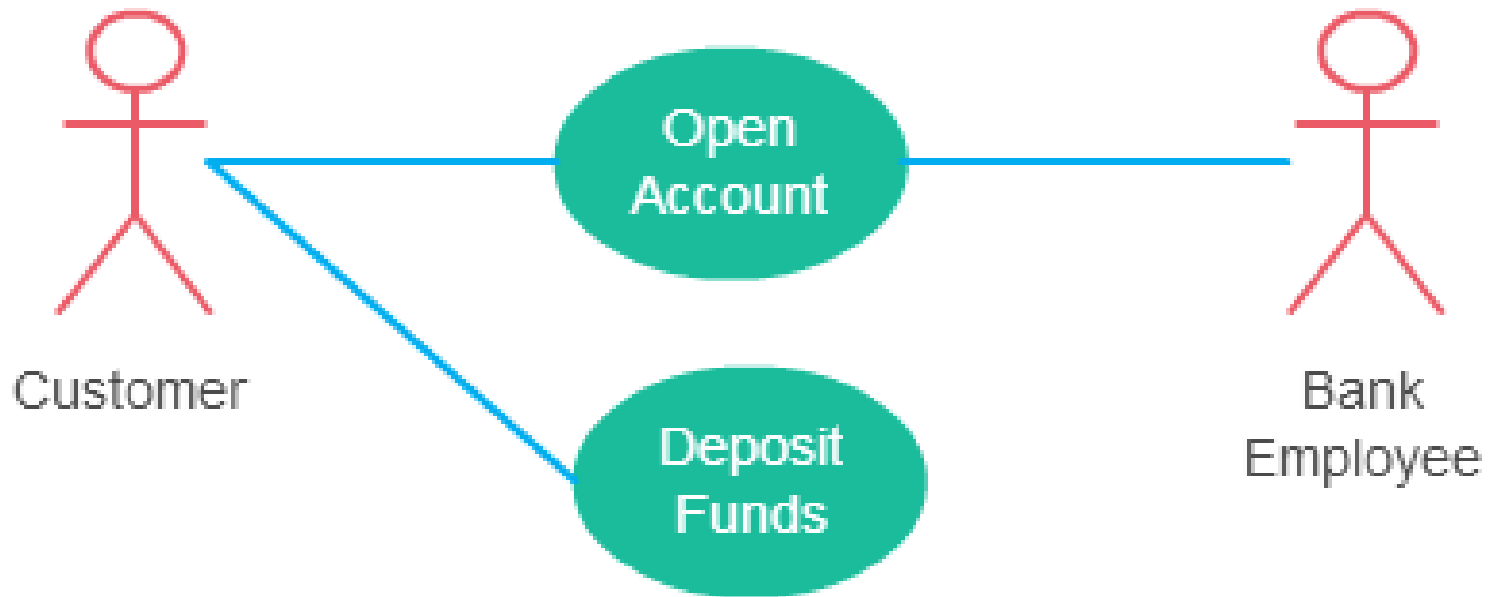
Generalization of an Actor

- Generalization of an actor means that one actor can inherit the role of an other actor.
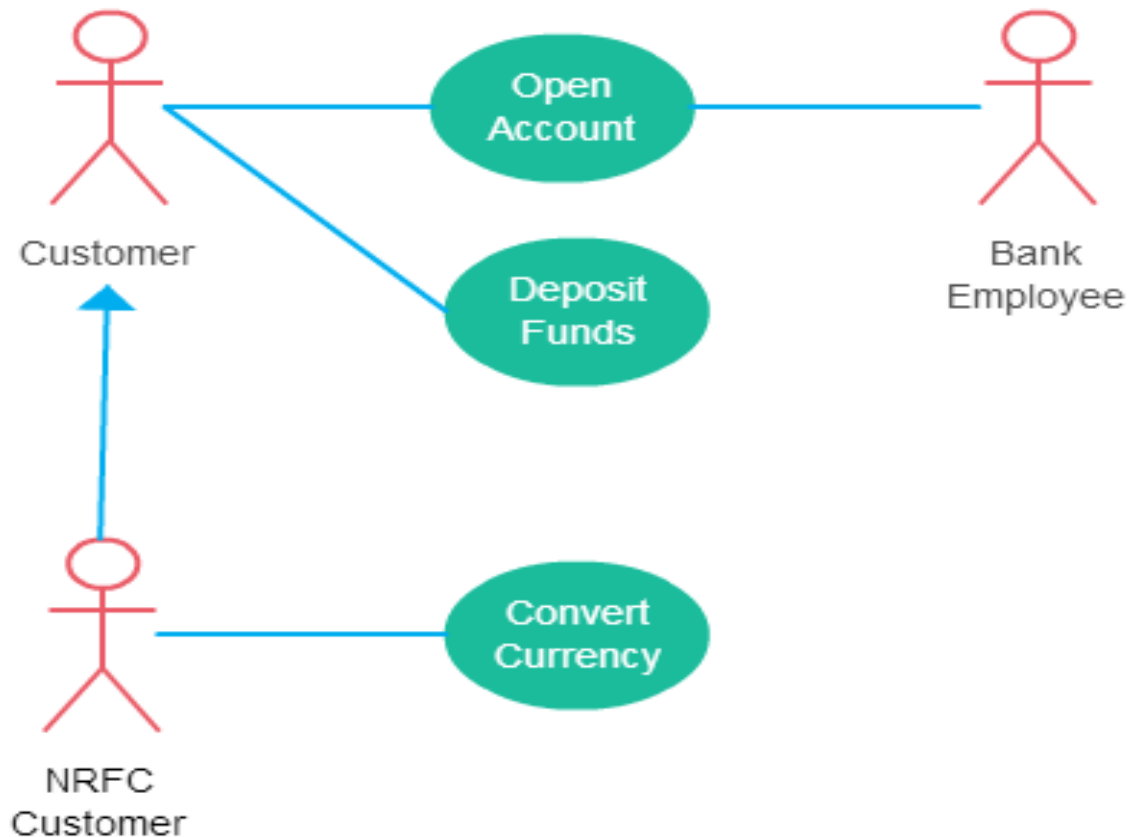
Extend Relationship Between Two Use Cases

- It extends the base use case and adds more functionality to the system.

- **The extending use case is dependent on the extended (base) use case**.

- **The extending use case is usually optional and can be triggered conditionally.**

- **The extended (base) use case must be meaningful on its own**.

- Arrow points to the base use case when using <<extend>>

- Extending use case is optional
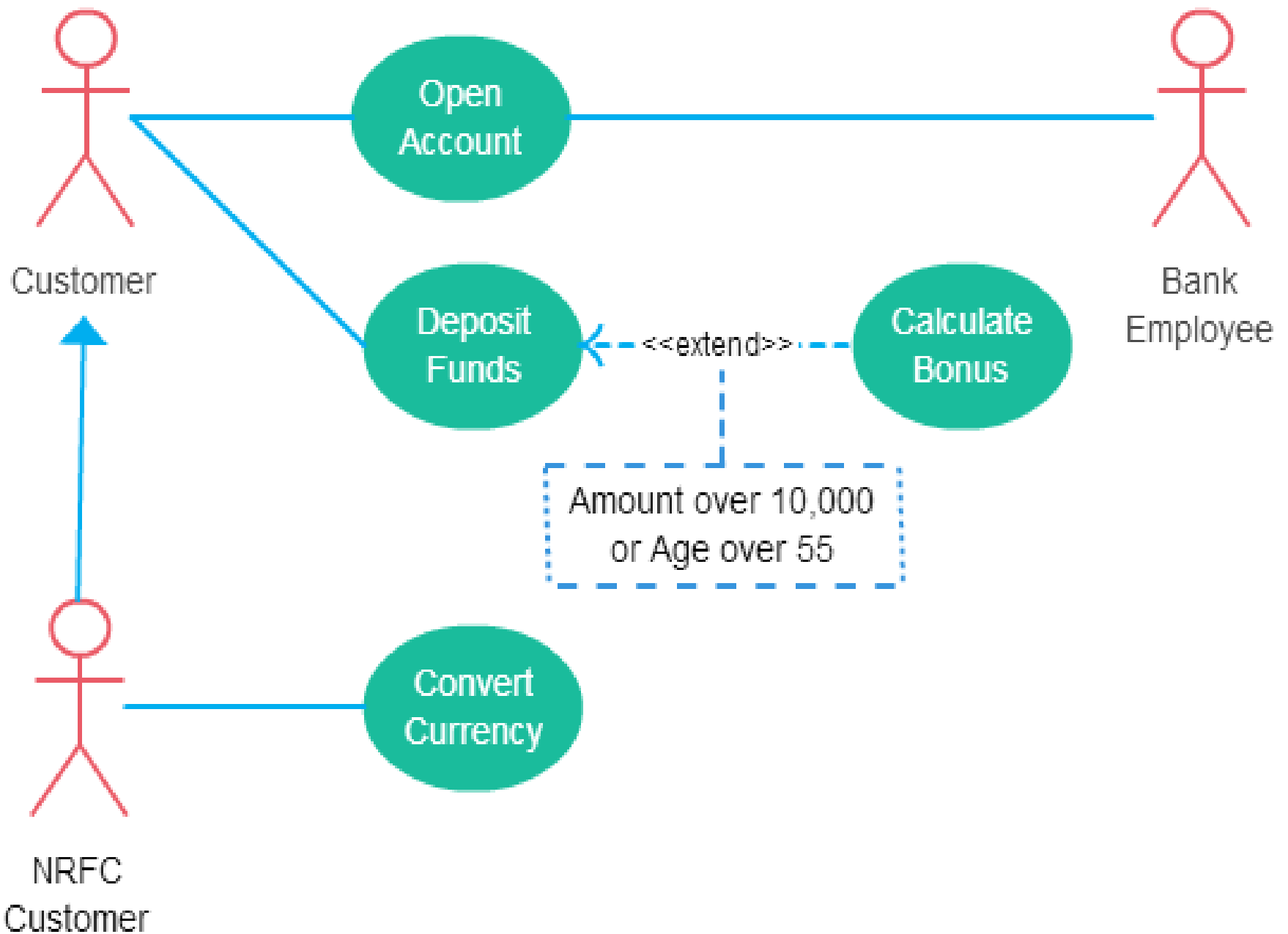
Include Relationship Between Two Use Cases

- Include relationship show that the behavior of the included use case is part of the including (base) use case.

- The base use case is incomplete without the included use case.

- The included use case is mandatory and not optional.

- Like a "use case subroutine"
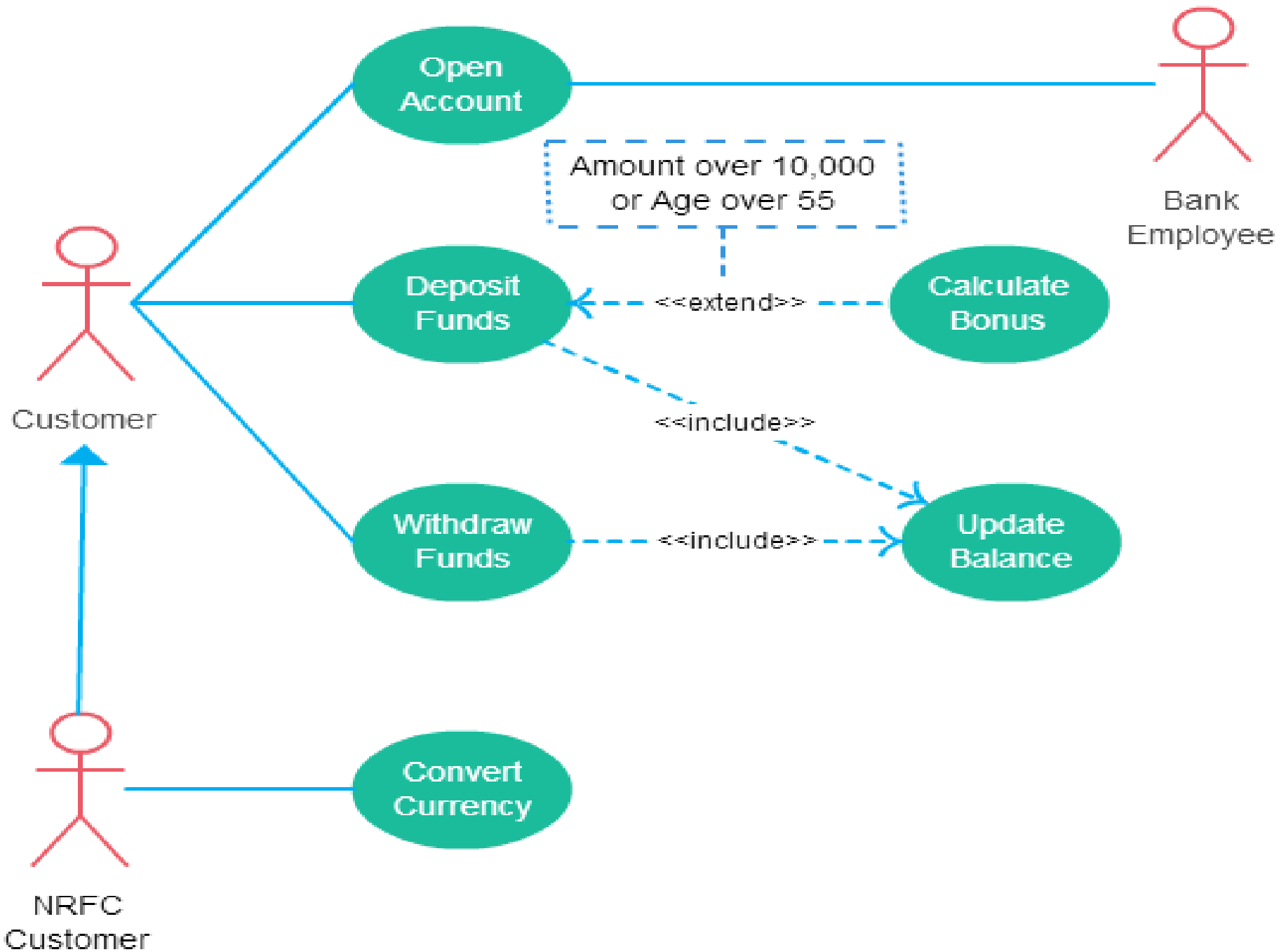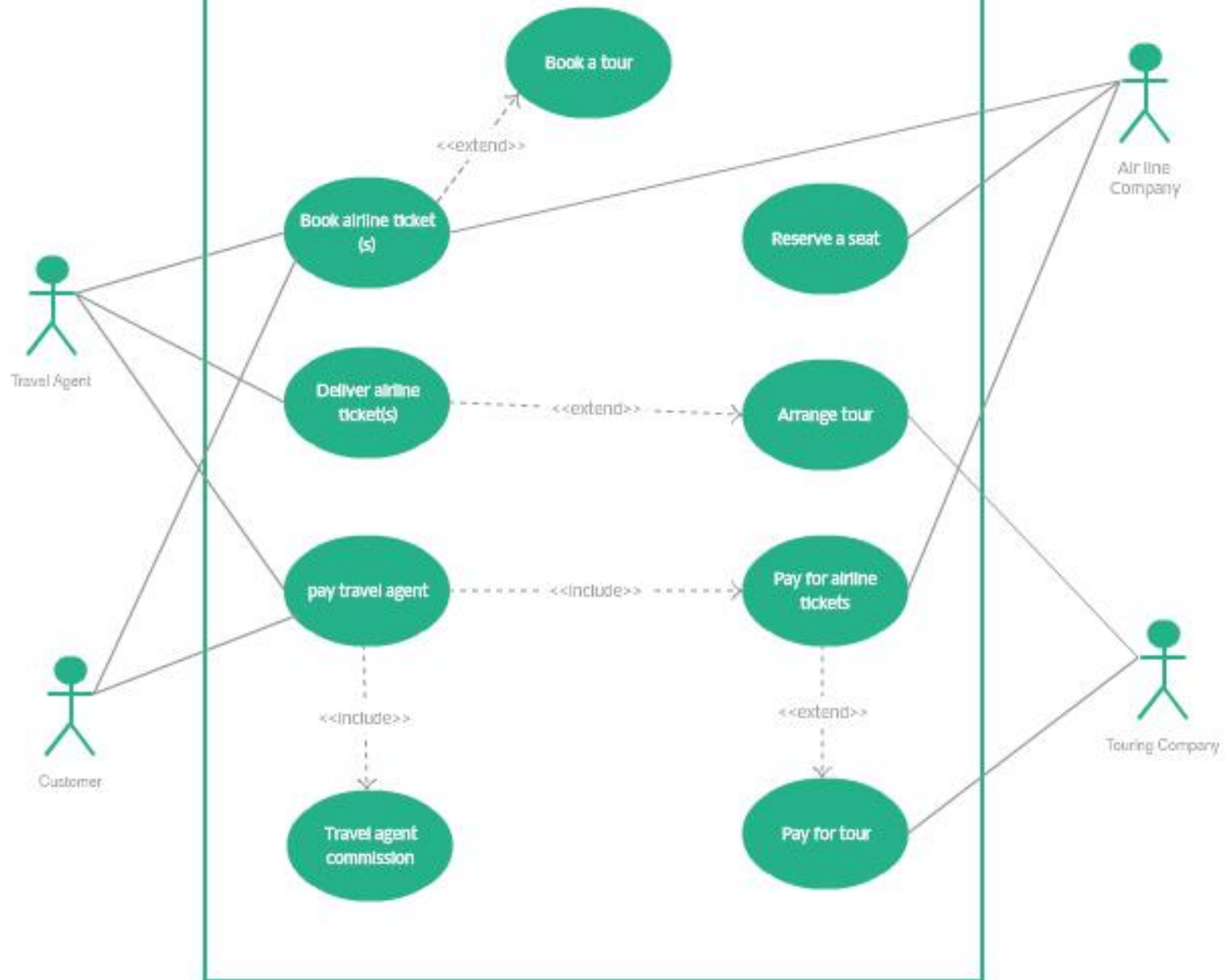
Generalization of a Use Case

- This is similar to the generalization of an actor.
- This is used when there are common behavior between two use cases and also specialized behavior specific to each use case.
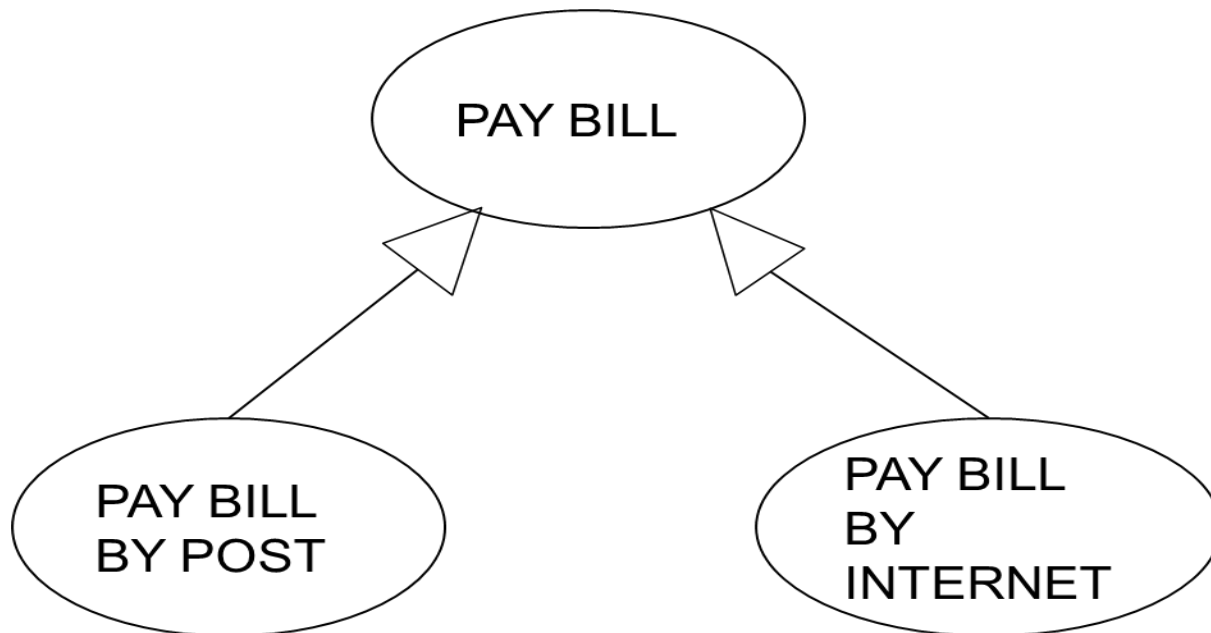
PROBLEM DEFINTION-VENDING MACHINE (write it down)

A vending machine accepts coins for a variety of products. The user selects the drink from products available through the selection panel. If the drink is available the price of the product is displayed. The user the deposits the coins depending on the price of the product. Coin collector collects the coins. After stipulated time, the controller will compare the deposited coins with the price, If the amount deposited is less than the price then error message will be displayed and all deposited coins will be dispensed by coin dispenser else the drink will be dispensed by the product dispenser. Draw use case diagram.

Use Case Diagram for Vending Machine

- Actor- User
- Processes involved in the system
  - Select product
  - Deposit coins
  - Verify the deposited amount
  - Dispense product
  - Dispense coins

Select Product

<<include>>

Deposit Coins

Verify deposited amount

Dispense product

<<extends>>

Dispense Coins

User
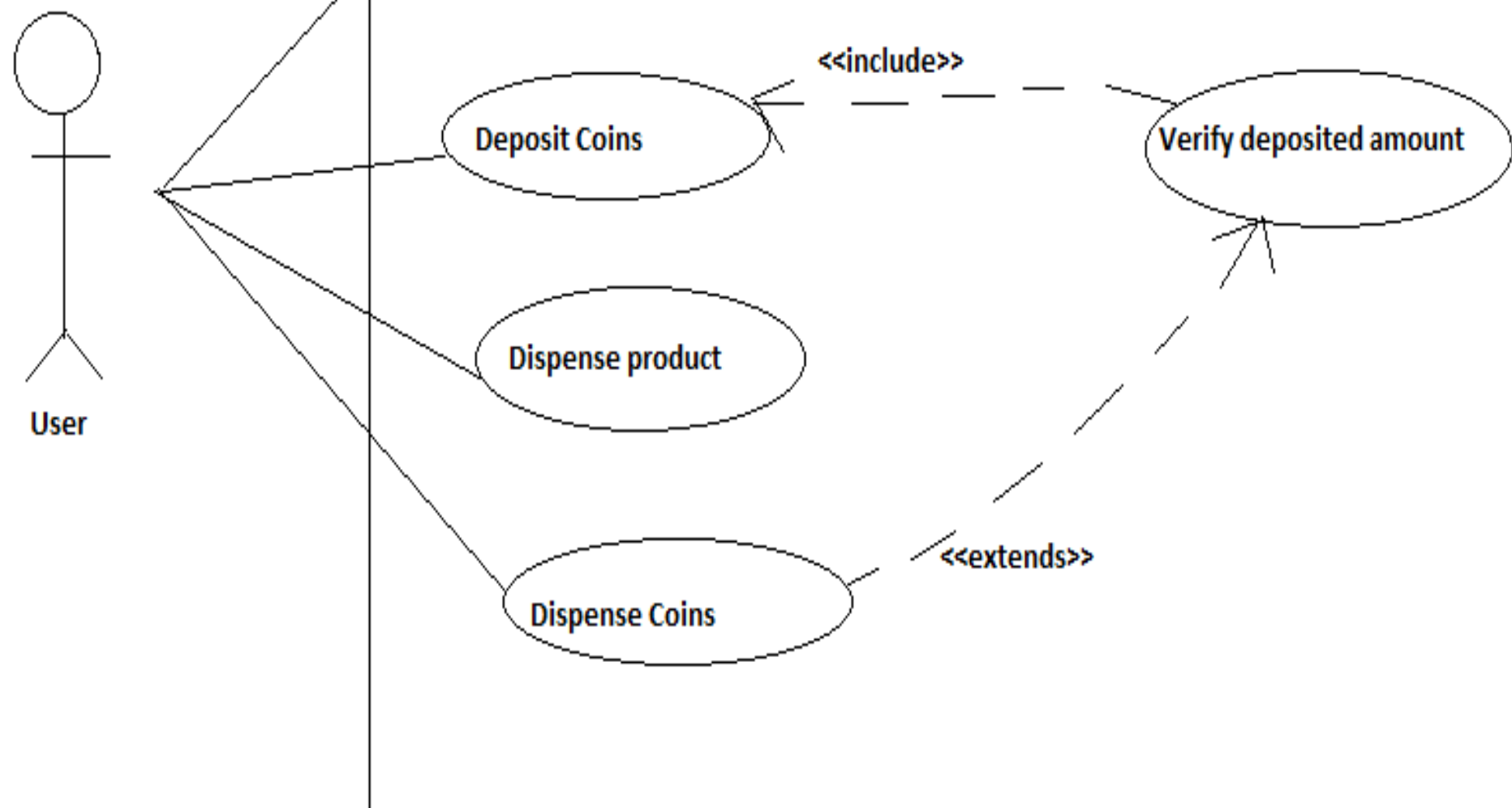
Use Case Example

- Customer (actor) uses bank ATM to Check Balances of his/her bank accounts, Deposit Funds, Withdraw Cash and/or Transfer Funds (use cases). ATM Technician provides Maintenance and Repairs. All these use cases also involve Bank actor whether it is related to customer transactions or to the ATM servicing.

«subsystem»
Bank ATM

Customer

Check Balances

Deposit Funds

Withdraw Cash

Transfer Funds

Maintenance

Repair

ATM Technician

Bank

© uml-diagrams.org
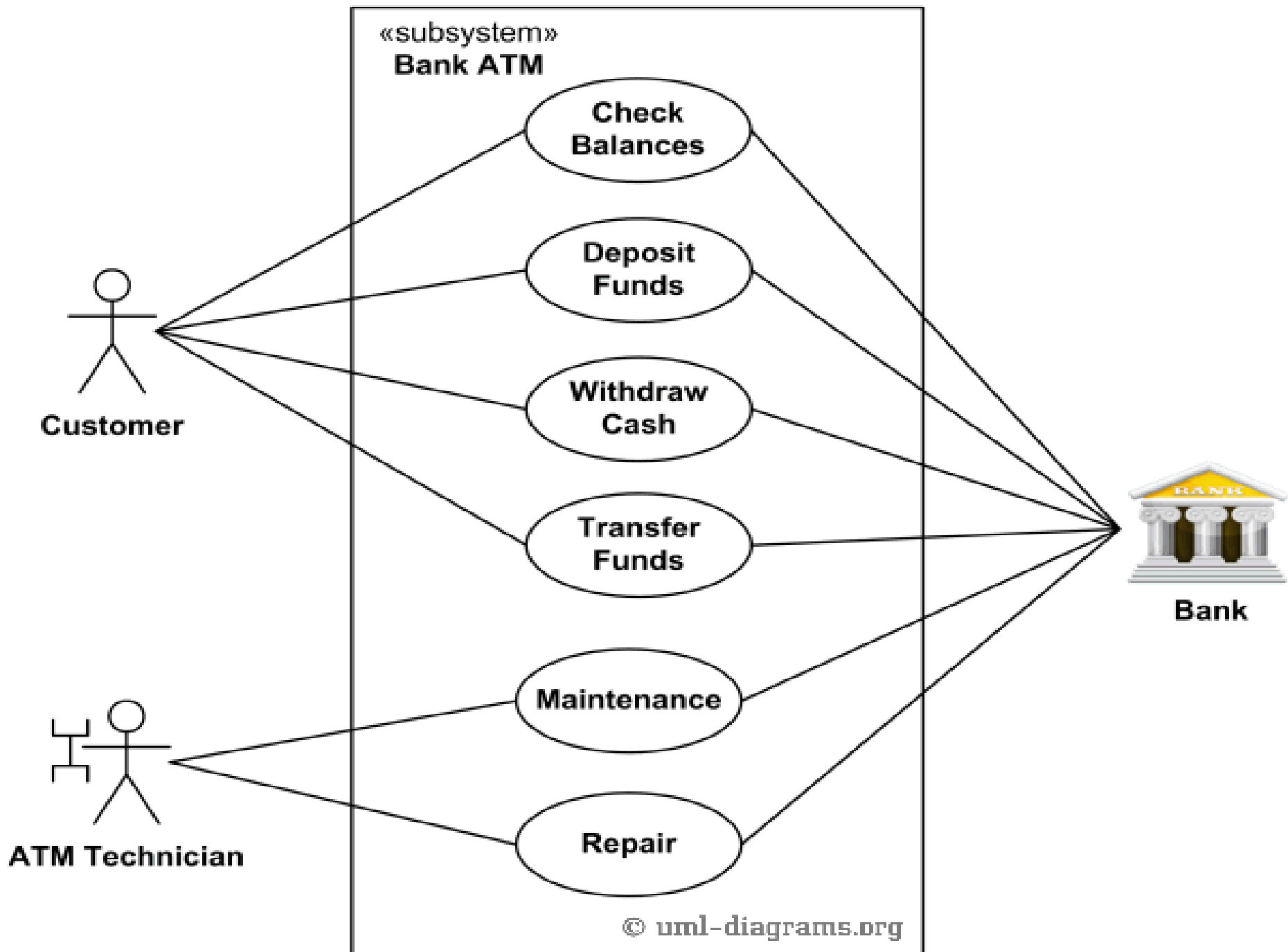
Use Case Example

- On most bank ATMs, the customer is authenticated by inserting a plastic ATM card and entering a personal identification number (PIN). Customer Authentication use case is required for every ATM transaction so we show it as include relationship. Including this use case as well as transaction generalizations make the ATM Transaction an abstract use case. Customer may need some help from the ATM. ATM Transaction use case is extended via extension point called menu by the ATM Help use case whenever ATM Transaction is at the location specified by the menu and the bank customer requests help, e.g. by selecting Help menu item.
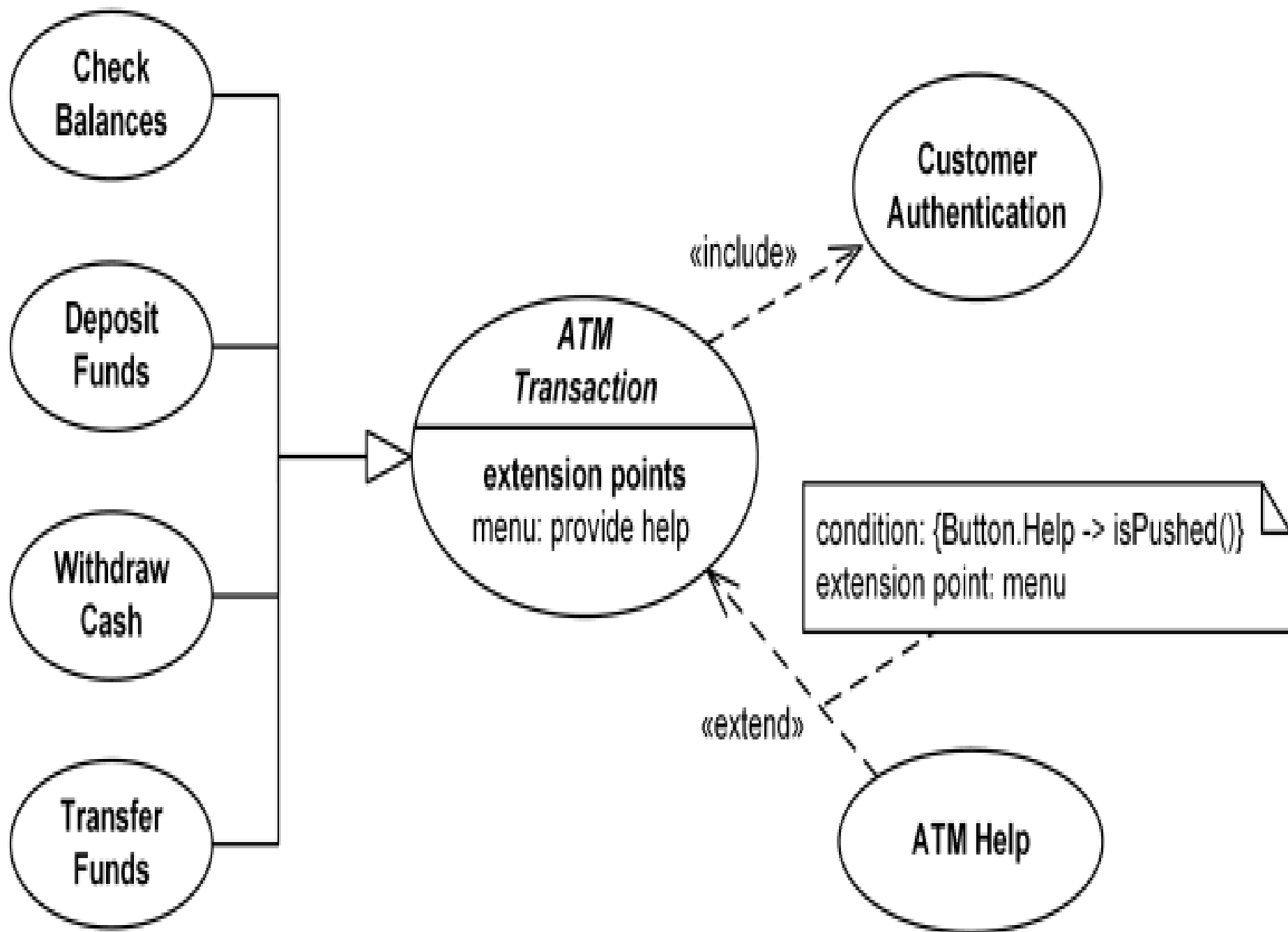
Use Case Example

- ATM Technician maintains or repairs Bank ATM. Maintenance use case includes Replenishing ATM with cash, ink or printer paper, Upgrades of hardware, firmware or software, and remote or on-site Diagnostics. Diagnostics is also included in (shared with) Repair use case.
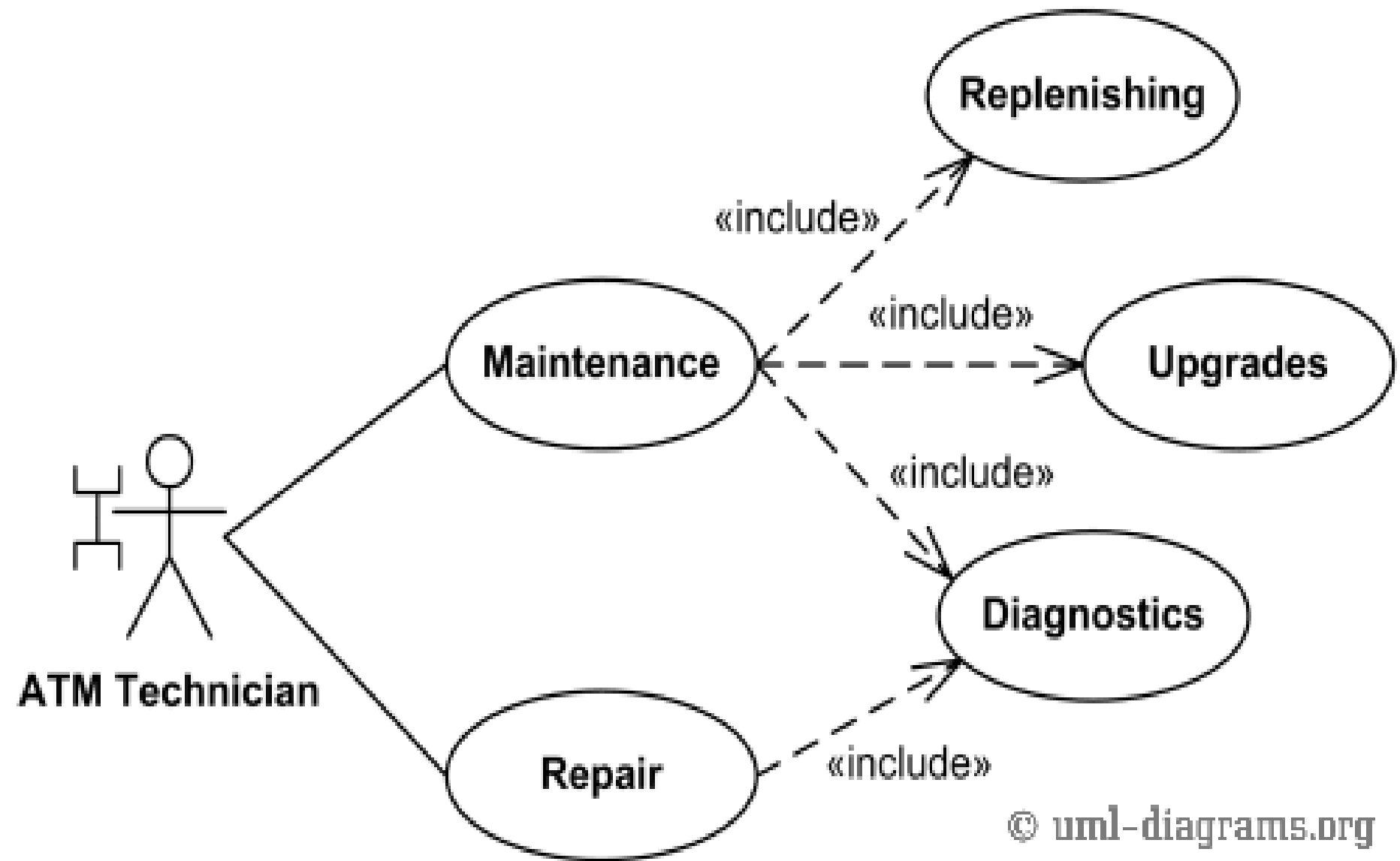
Bank ATM Maintenance, Repair, Diagnostics Use Cases Example.

1. A leading TRAVEL AGENCY has decided to develop application package to help its **20**
   customer in planning tours. The agency provides services like tour, air, railway, luxury
   coach, hotel booking etc. Many a times customers do not have idea of availability of
   transport services to a particular destination. The agency also gives advice regarding
   economical planning of vacation/tour. Given the tour constraints like number of days,
   affordable cost and places to visit the software should present alternative tour plans.
   Alternatively the software may be just used for querying to know availability of transport
   services, hotels etc. Besides this main objective of this software should also have facilities
   for billing and accounting for the agency. You are appointed as a consultant to develop
   implementation strategy for Automated Tourist System. Draw use case and class diagram.

Customer

Agent

Make Enquiry

Plan Trip

Select Trip And Finalize It

Make Payment

<<include>>

Receive Ticket

Cancel Trip

<<include>>

Get Refund

Provide Enquiry Details

Give Different Trip Details

Receive Payment

<<include>>

Book Ticket

Cancel Tickets

<<extend>>

Give Discount If Any

<<include>>

Refund Money

System

http://umldiagramtutorial.blogspot.in/
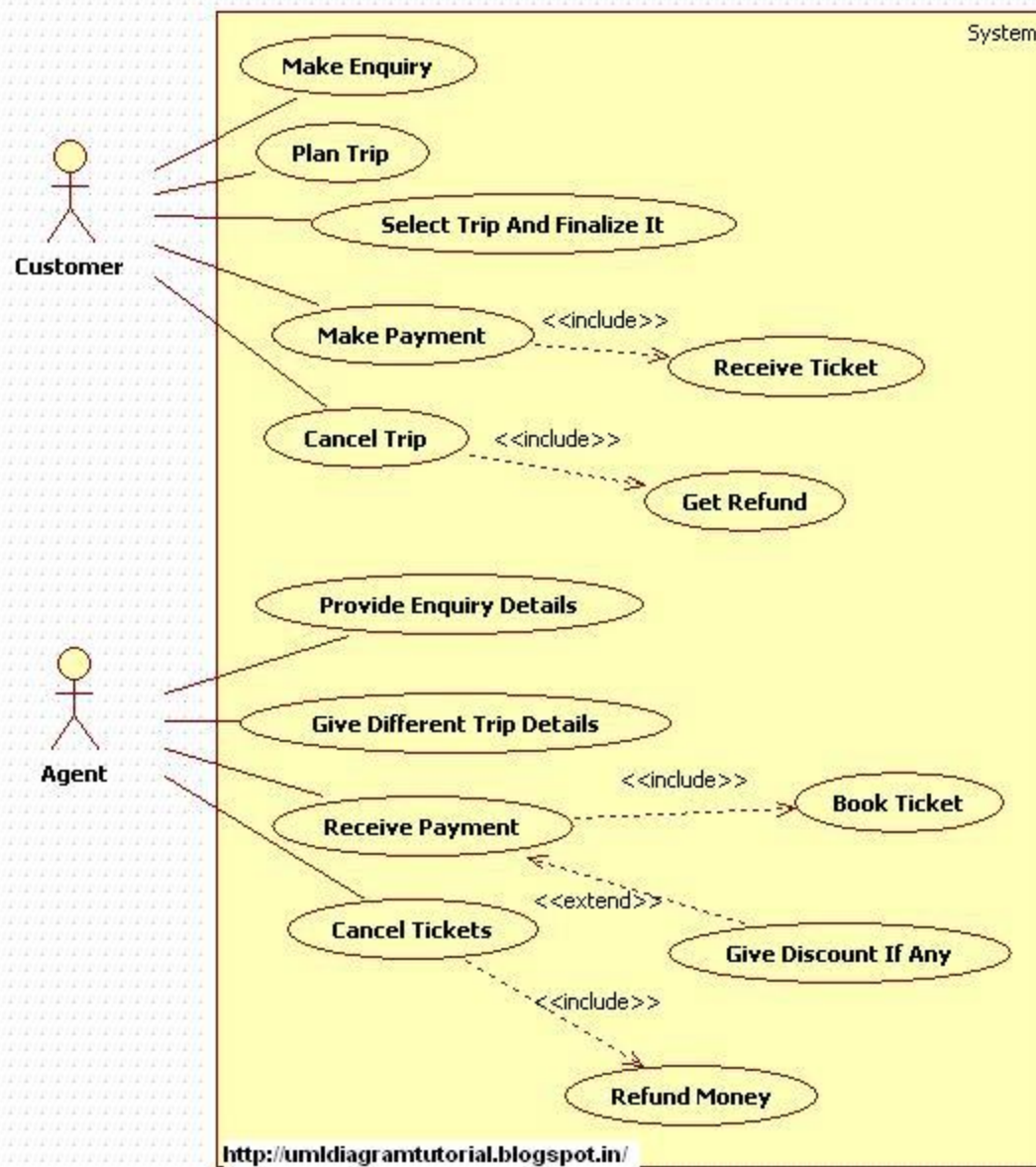
USE CASE

- **Online Airline Reservation System**
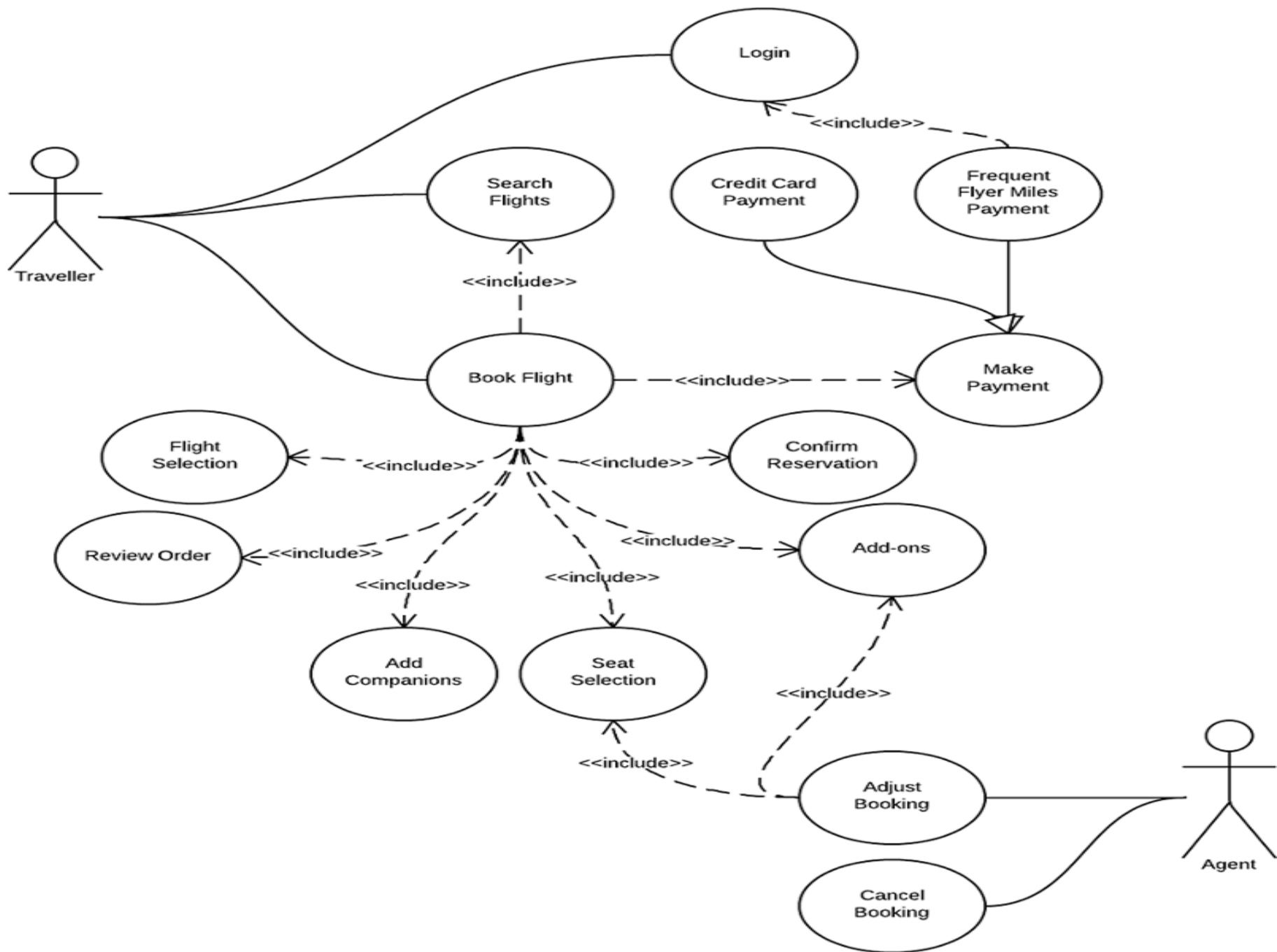
Activity Diagram

- An activity diagram describes the **behavior of a system in terms of activities.**

- **Activity diagram** is basically a **flow chart** to represent the flow form **one activity to another activity.**

- Activity diagram: **operation of a system**.

- This flow can be **sequential, branched or concurrent.**

Activity Diagram (basic notations)

- **Initial node**.
- ✓ The filled in circle is the starting point of the diagram.
- ✓ It is shown as filled circle.

Initial Activity

Activity Diagram (basic notations)

- **Activity final node**.

✓ The filled circle with a border is the ending point.

✓ The final activity is optional.
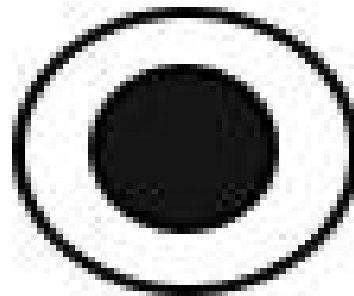
Final Activity

Activity Diagram (basic notations)

- **Activity**. The rounded rectangles represent activities that occur.
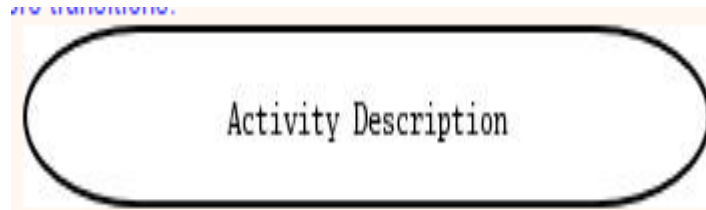
Activity Diagram (basic notations)

- **Flow/edge (Control Flow)**

✓ The arrows on the diagram. Within the control flow an incoming arrow starts a single step of an activity; after the step is completed the flow continues along the outgoing arrow.

Activity Diagram (basic notations)

Decision.

- **Decisions** are branches in the control flow.

- They denote alternatives based on a condition of the state of an object or a set of objects.

- Decisions are depicted by a diamond with one or more incoming open arrows and two or more outgoing arrows.

Activity Diagram (basic notations)

- The outgoing edges are labeled with the conditions that select a branch in the control flow.

- Decisions are depicted by a diamond with one or more incoming open arrows and two or more outgoing arrows.

- The set of all outgoing edges from a decision represents the set of all possible outcomes.

Activity Diagram (basic notations)
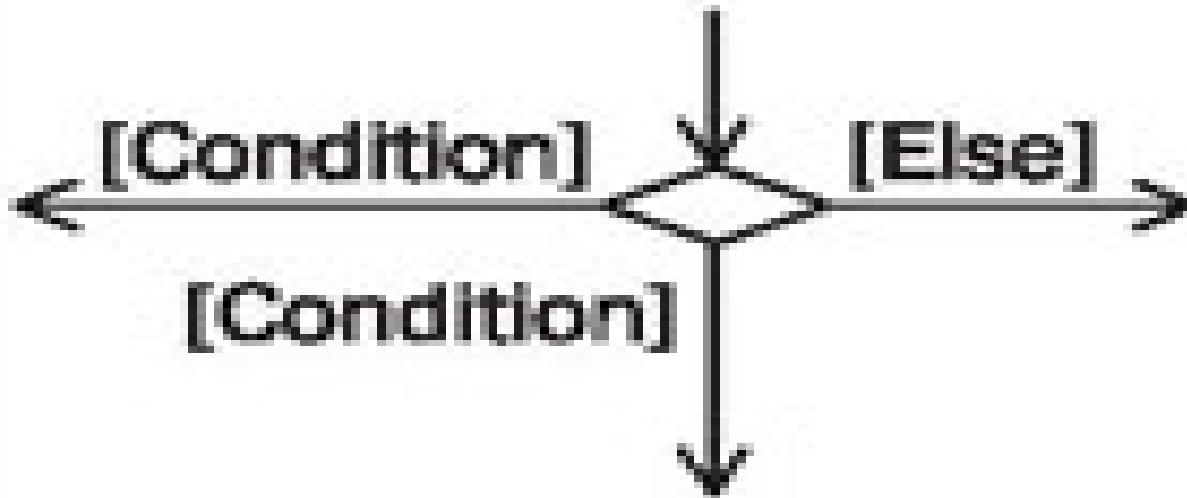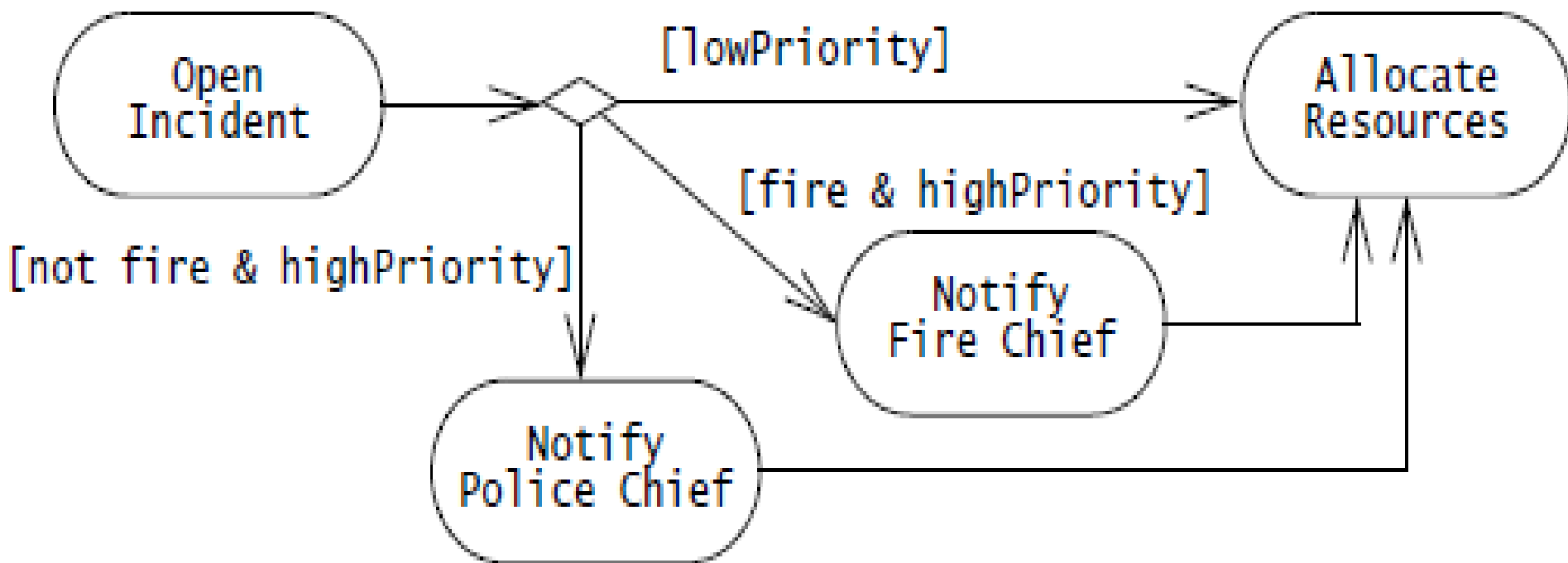
Activity Diagram (basic notations)

- Some **activities** occur simultaneously or in **parallel**. Such activities are called **concurrent activities.**

- **Fork nodes** and **join nodes** represent concurrency.

- **Fork nodes** denote the **splitting of the flow** of control into multiple threads.

- Join nodes denotes the **synchronization of multiple threads** and their **merging of the flow** of control into a **single thread**.
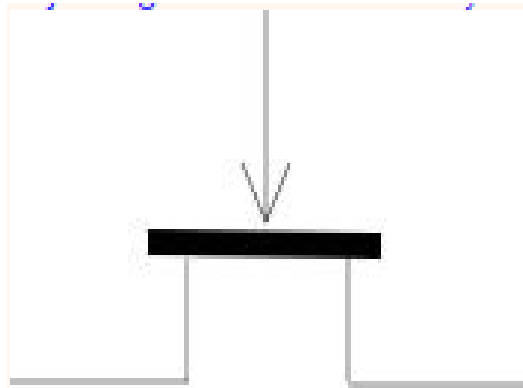
Synchronization Bar (Fork)
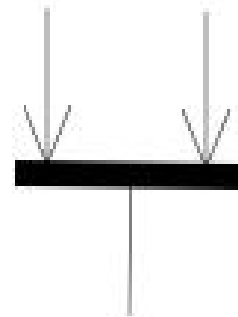
Synchronization Bar (Join)
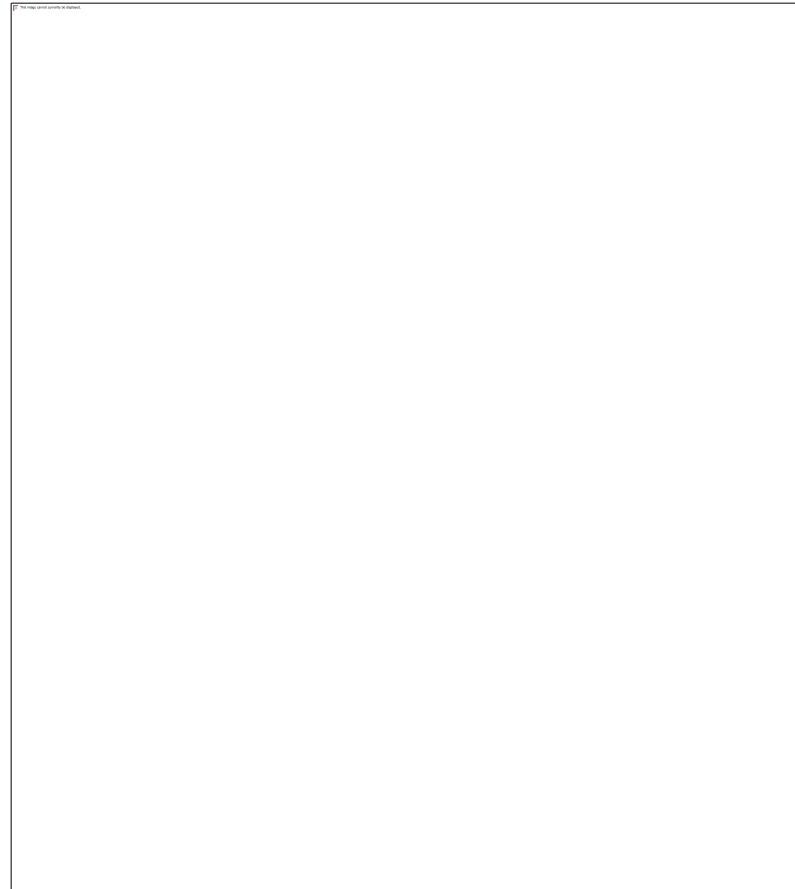
An example of fork and join nodes in a UML activity diagram.
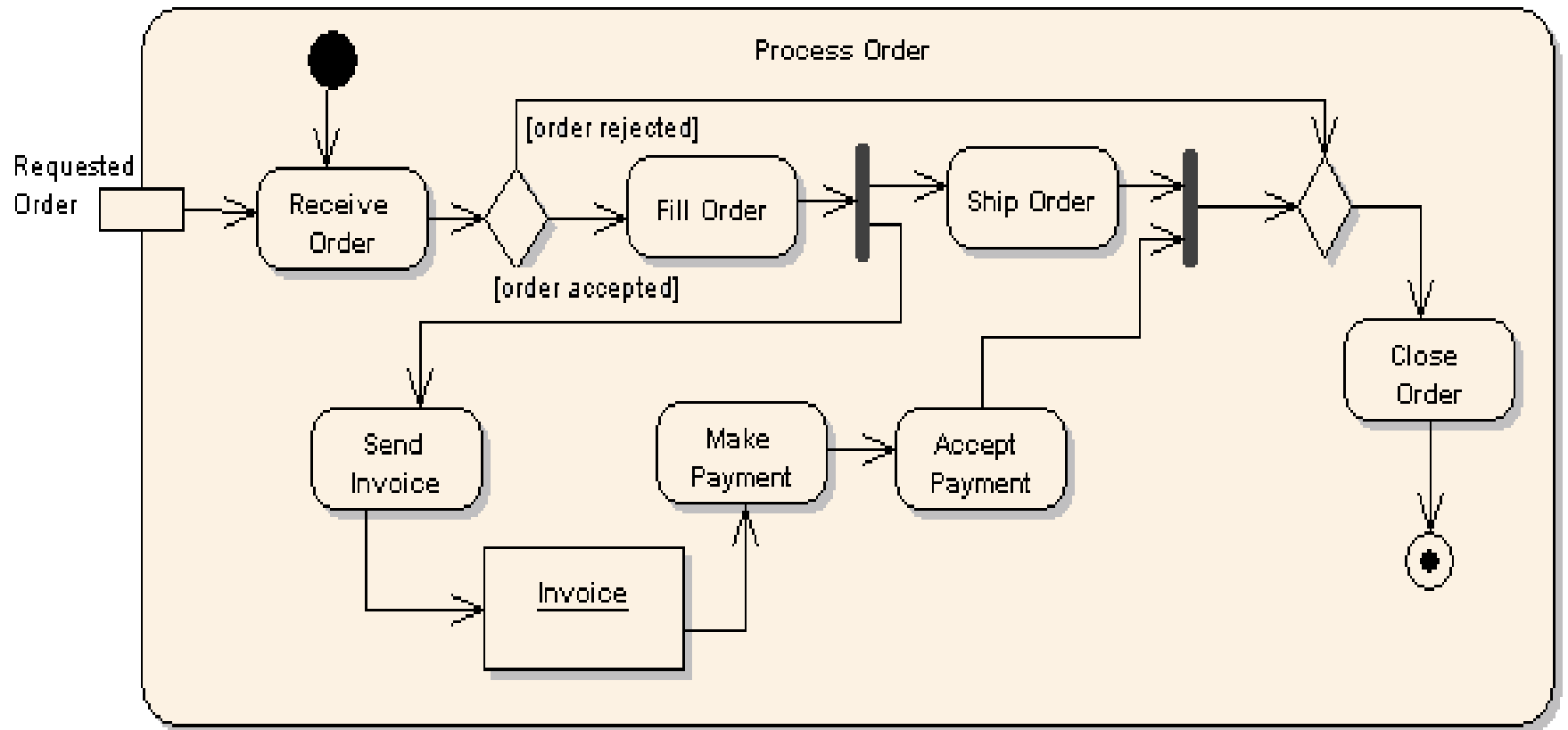
Activity Diagram (basic notations)

- **Swim Lanes**
  - The contents of an activity diagram may be organized into *partitions* (swimlanes) using solid vertical lines.

  - A swim lane (or **swimlane diagram**) is a visual element used in process flow **diagrams**, or flowcharts, that visually distinguishes job sharing and responsibilities for sub-processes of a business process.

Activity Diagram (basic notations)

- Order Swimlanes in a Logical Manner.

- Apply Swim Lanes To Linear Processes.

- Have Less Than Five Swimlanes.

- Consider Swim areas For Complex Diagrams.
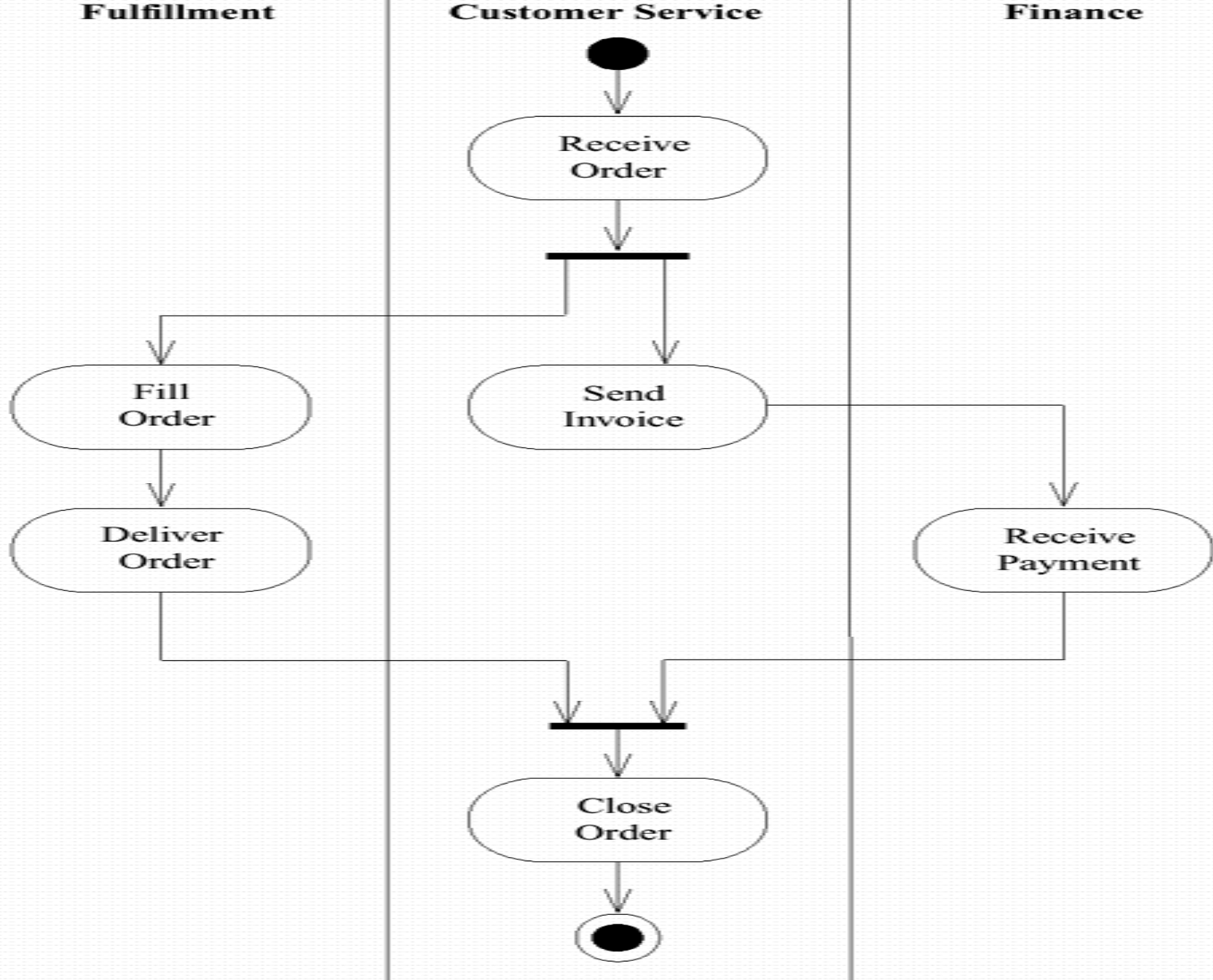
Problem

- **Activity Diagram for Ticket Vending Machine**

- **Problem Definition:** Draw Activity Diagram for Login Procedure of a system.

Problem Definition

- Activity is started by Commuter actor who needs to buy a ticket. Ticket vending machine will request trip information from Commuter. This information will include number and type of tickets, e.g. whether it is a monthly pass, one way or round ticket, route number, destination or zone number, etc. Based on the provided trip info ticket vending machine will calculate payment due and request payment options. Those options include payment by cash, or by credit or debit card. If payment by card was selected by Commuter, another actor, Bank will participate in the activity by authorizing the payment. After payment is complete, ticket is dispensed to the Commuter. Cash payment might result in some change due, so the change is dispensed to the Commuter in this case. Ticket vending machine will show some "Thank You" screen at the end of the activity.
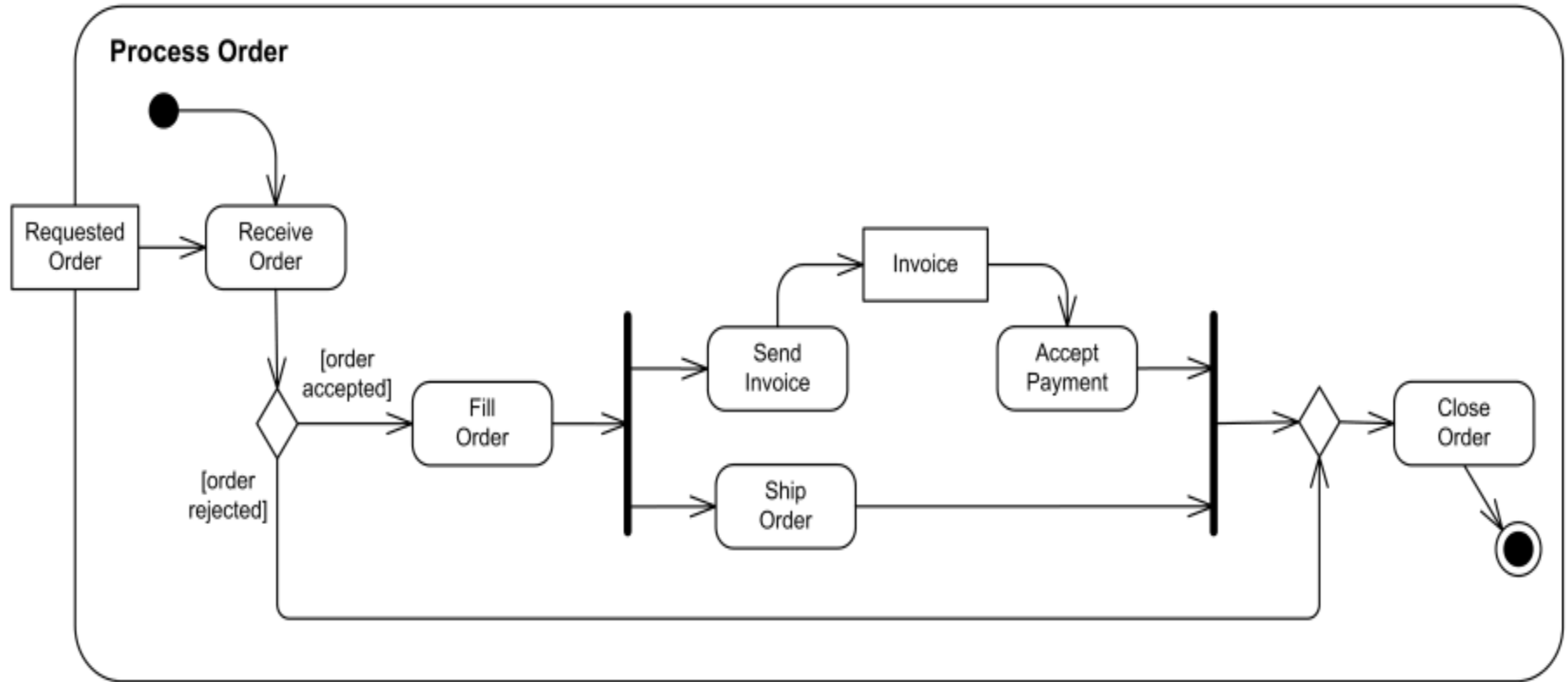
- *An example of business flow activity to process purchase order.*

Commuter	Ticket vending machine	Bank

Start Session

Request Trip Info

Provide Trip Info

Process Trip Info

Provide Payment Info

Request Payment

Process Payment

[pay with card]

[pay with cash]

Authorize Card Payment

Dispense Ticket

Ticket

Get Ticket

[paid with cash & with change]

Dispense Change

Change

Get Change

Show Thank You

© uml-diagrams.org

CLASS DIAGRAM

Student
Name:
Address:
Birthdate:

Student
Name:
Address:
Birthdate:
Age:

GENERALIZATION

DEPENDENCY

ASSOCIATION

CLASS DIAGRAM

- A Class Diagram is a diagram **describing the structure of a system.**

- Class diagrams are the **main building blocks** of every object oriented methods.

  - The class diagram can be used to show the classes, relationships, interface, association, and collaboration

- shows the system's

✓ classes

✓ Attributes

✓ operations (or methods),

✓ Relationships among the classes.

Essential Elements of a
Class Diagram

- **CLASS**: A class represents the blueprint of its objects.
- Attributes
- Operations
- Relationships
  - Associations
  - Generalization
  - Realization
  - Dependency

Class

- Describes a set of objects having similar:
  - Attributes (status)
  - Operations (behavior)
  - Relationships with other classes
- Attributes and operations may
  - have their visibility marked:
  - "+" for *public*
  - "#" for *protected*
  - "–" for *private*

Class Name

Attributes

Operations

Window
size: Size visibility: boolean
display() hide()
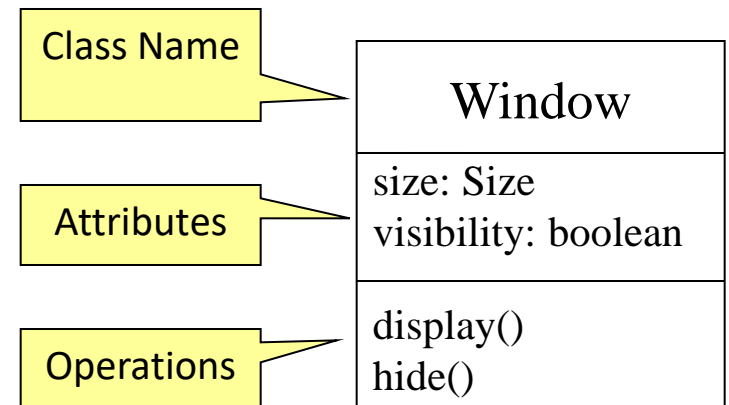
Associations

- An association between two classes indicates that objects at one end of an association "recognize" objects at the other end and may send messages to them.

- Example: "An Employee works for a Company"

| Employee | Company |
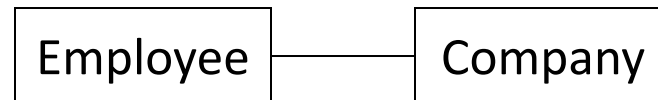
Directed Association

- Refers to a **directional relationship** represented by a line with an arrowhead. The arrowhead depicts a **container-contained** directional flow.
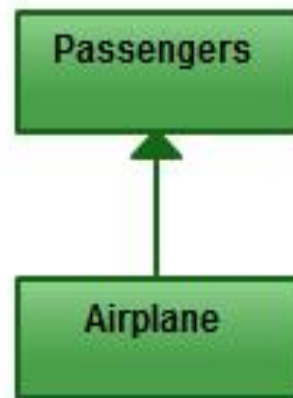
Reflexive Association

- A reflexive association is a relationship from one class back to itself.
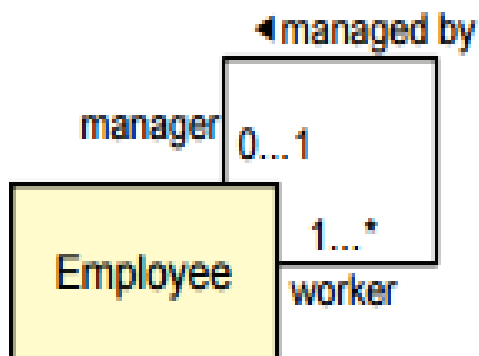
Figure 2.17

Reflexive relationship. A manager is an employee who manages other workers. Notice how the labels explain the purpose of the relationship.
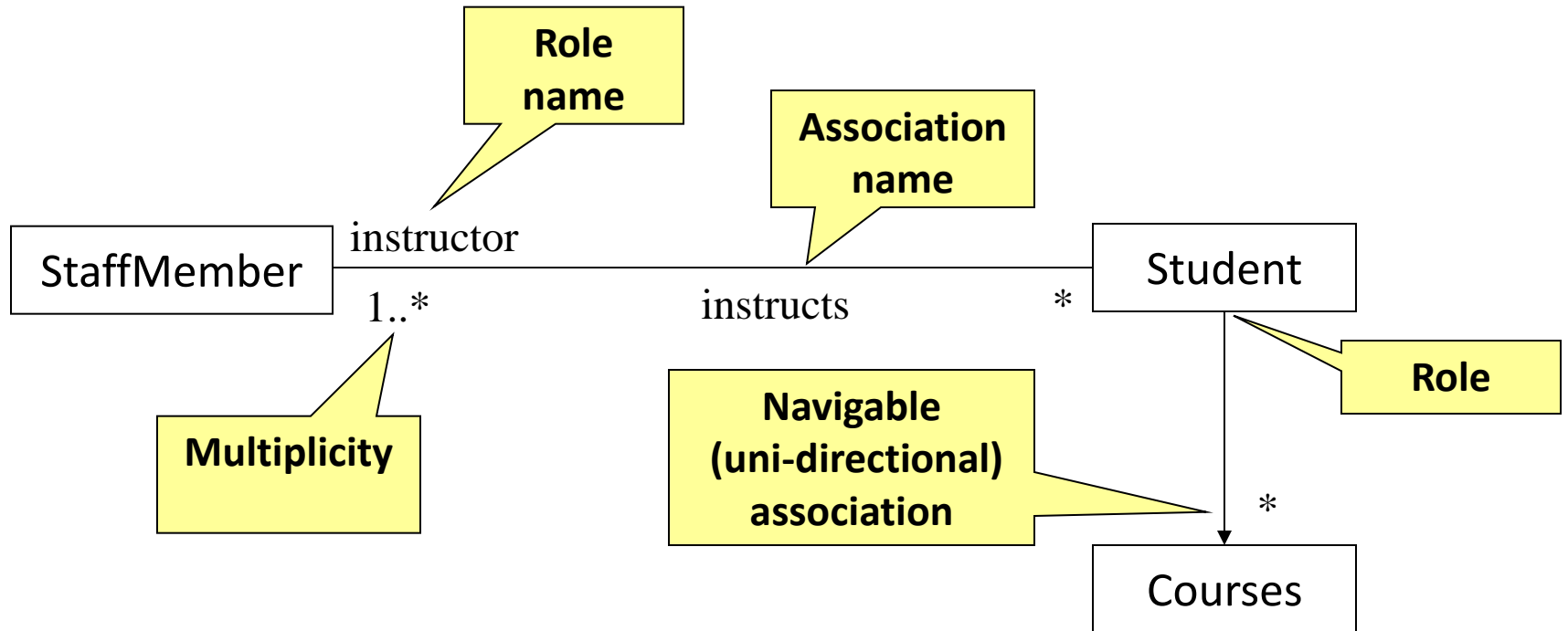
Association Class

- In modeling an association, there are times when you need to **include another class** **because it includes valuable information** about the relationship.

- use an *association class* **that you tie to the primary association.**

- An association class is represented like a normal class.

- The difference is that the **association line** between the primary classes intersects a **dotted line connected** to the association class.

```
                                    0..                          1..
                                    *                            *
┌─────────────────────┐                                      ┌──────────────────┐
│      Student        │──────────────────┬───────────────────│     Course       │
├─────────────────────┤                  │                   └──────────────────┘
│ Name                │                  │
│ Student #           │                  │
│ ...                 │                  │
├─────────────────────┤                  │
│                     │                  │
│ Constraints         │                  │
│ {must be 18 years   │        ┌─────────────────────┐
│ of age or older}    │        │     Enrollment      │
└─────────────────────┘        ├─────────────────────┤
                               │ Enrollment Date     │
                               ├─────────────────────┤
                               │ Enroll              │
                               │ Drop                │
                               │ Cancel              │
                               └─────────────────────┘
```
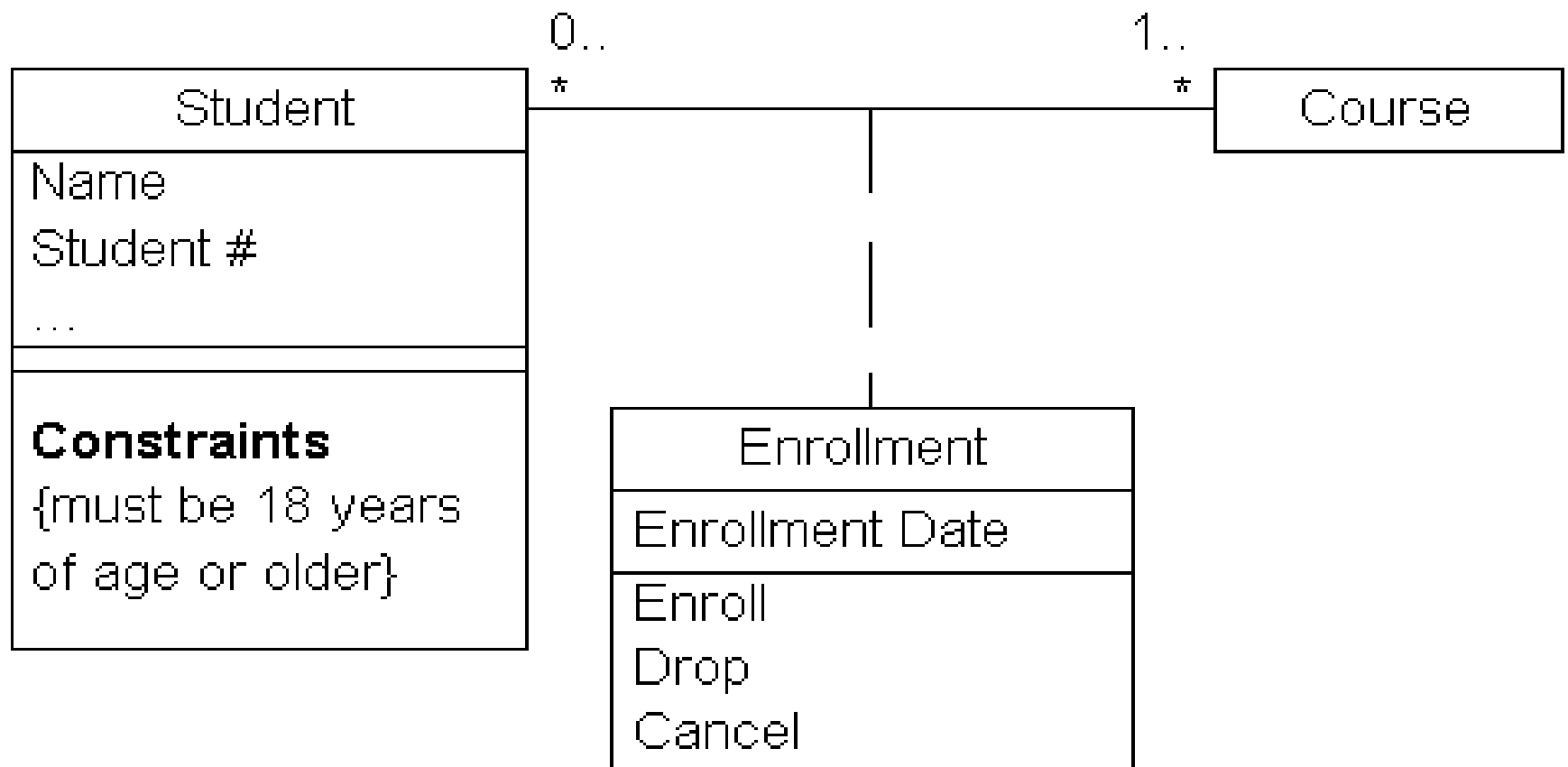
Multiplicity

- the number of objects that participate in the association.

- **Multiplicity Indicators**

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

BASIC AGGREGATION

- Aggregation is a special type of association used to model a **"whole to its parts"** relationship.

- Example: Car as a whole entity and Car Wheel as part of the overall Car.

Car		2..*	Door		1..*	House
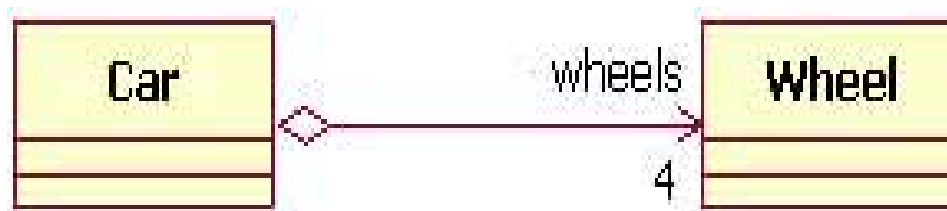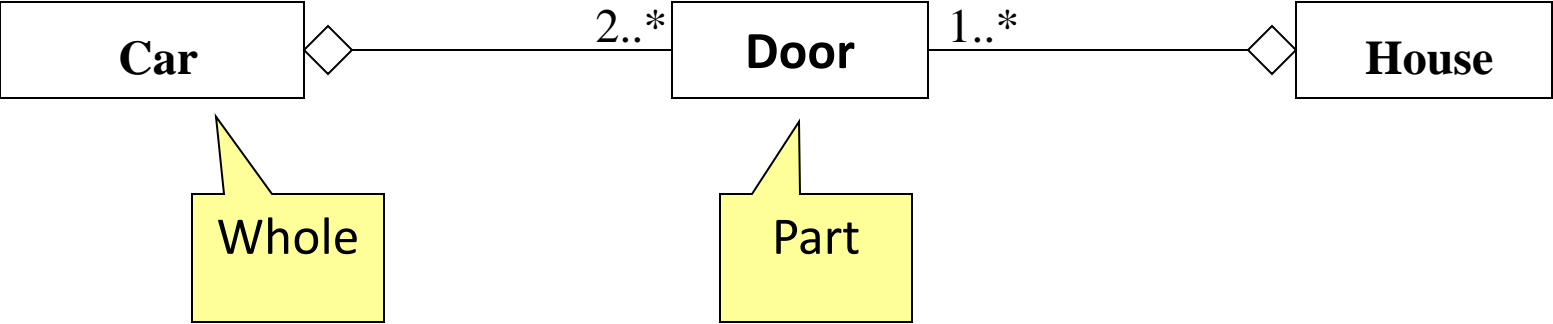
Whole

Part

Composition Aggregation

- **A strong form of aggregation**
  - The whole is the sole owner of its part.
    - The part object may belong to only one whole
  - Multiplicity on the whole side must be zero or one.
  - The life time of the part is dependent upon the whole.
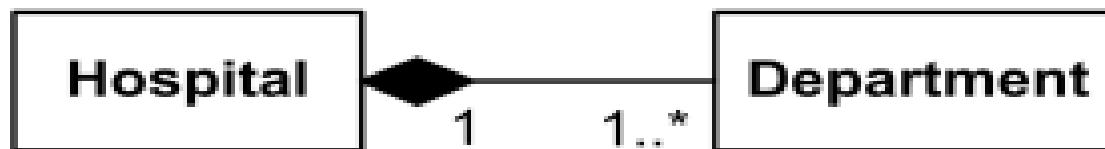
Hospital has 1 or more Departments, and each Department belongs to exactly one Hospital. If Hospital is closed, so are all of its Departments.

Generalization

- The Generalization association ("**is a**") is the **relationship** between the **base class** that is named as "superclass" or "**parent**" and the specific class that is named as "**subclass**" or "**child**".
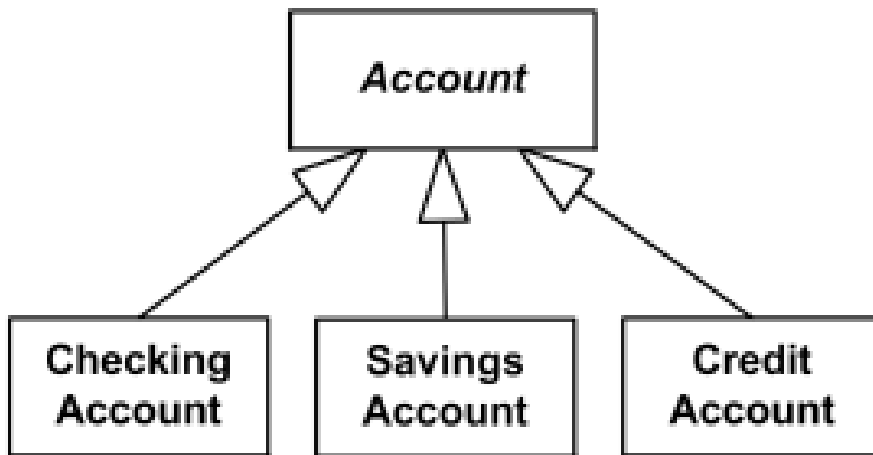
*Checking, Savings, and Credit Accounts are **generalized** by Account*
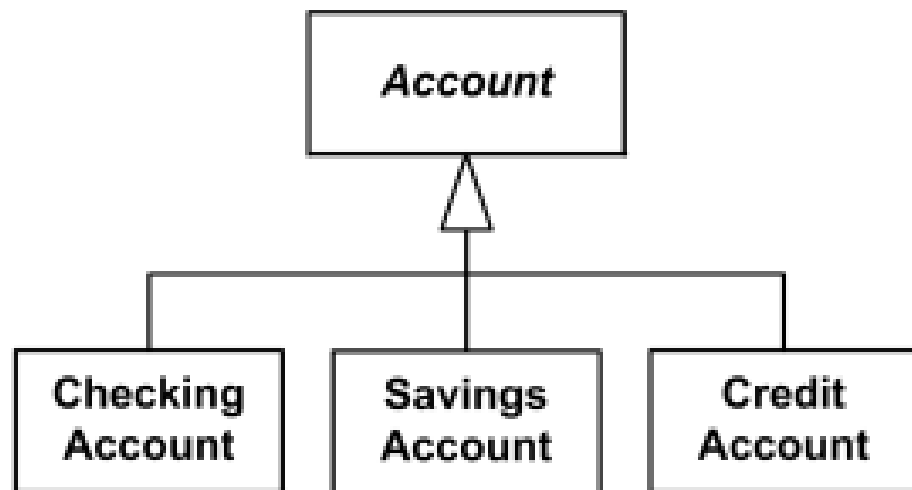
*Checking, Savings, and Credit Accounts are **generalized** by Account*

Realization

- A realization relationship indicates that **one class implements a behavior specified by another class** (an interface or protocol).

- a broken line with an unfilled solid arrowhead is drawn from the class that defines the functionality to the class that implements the function.

Realization
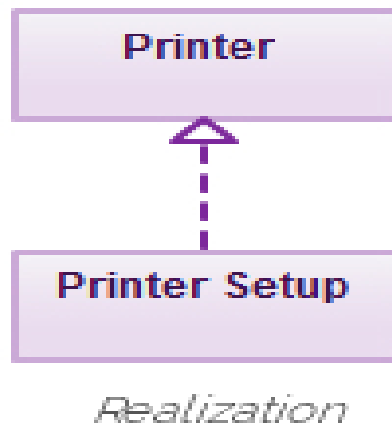
Dependency

- It means that the class at the **source end** of the relationship has some sort of **dependency** on the class at the **target** (arrowhead) end of the relationship.
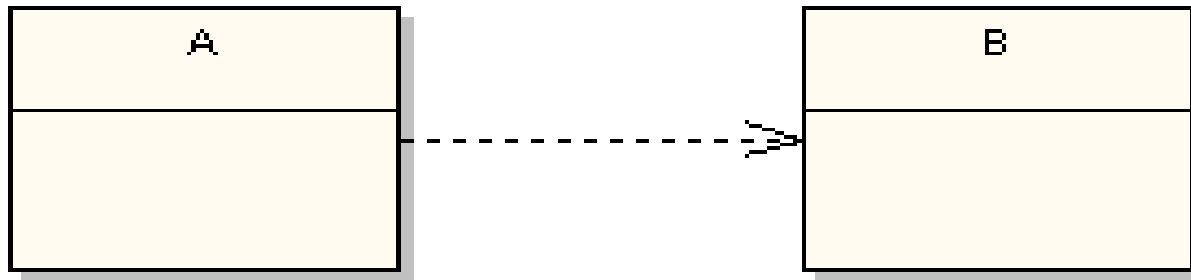
Dependency

- In UML, a dependency relationship is a relationship in which one element, the client, uses or depends on another element, the supplier.

- a **Cart class depends on a Product class** because the Cart class uses the Product class as a parameter for an **add operation.**

- Cart class is, therefore, the client, and the Product class is the supplier.

- Dependency shows that an element is dependent on another element as it fulfils its responsibilities within the system
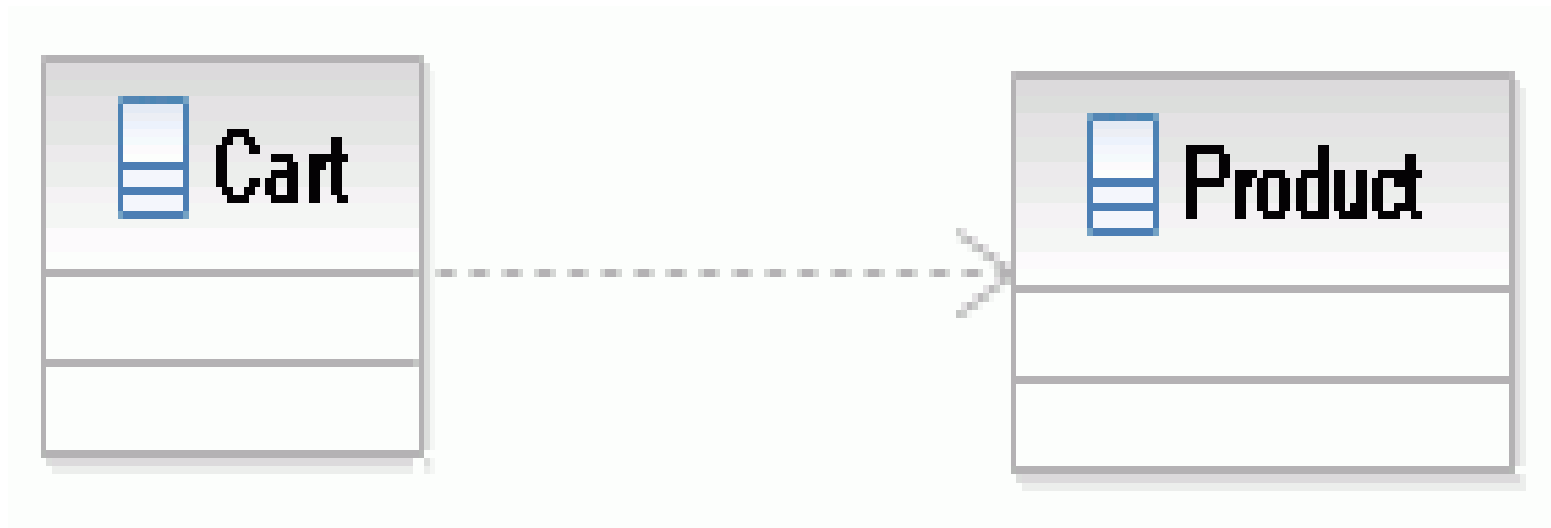
Dependency Relationships

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*.

© SERG
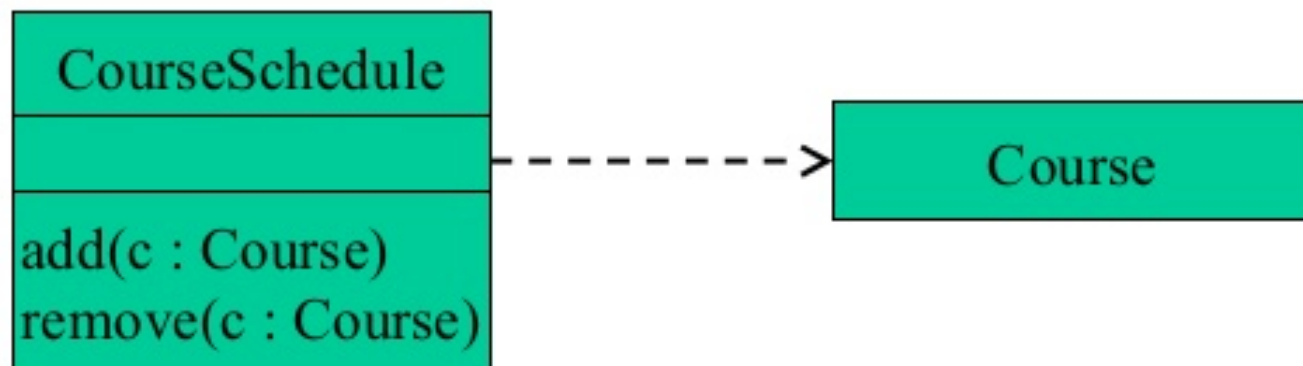
Composition/aggregation example
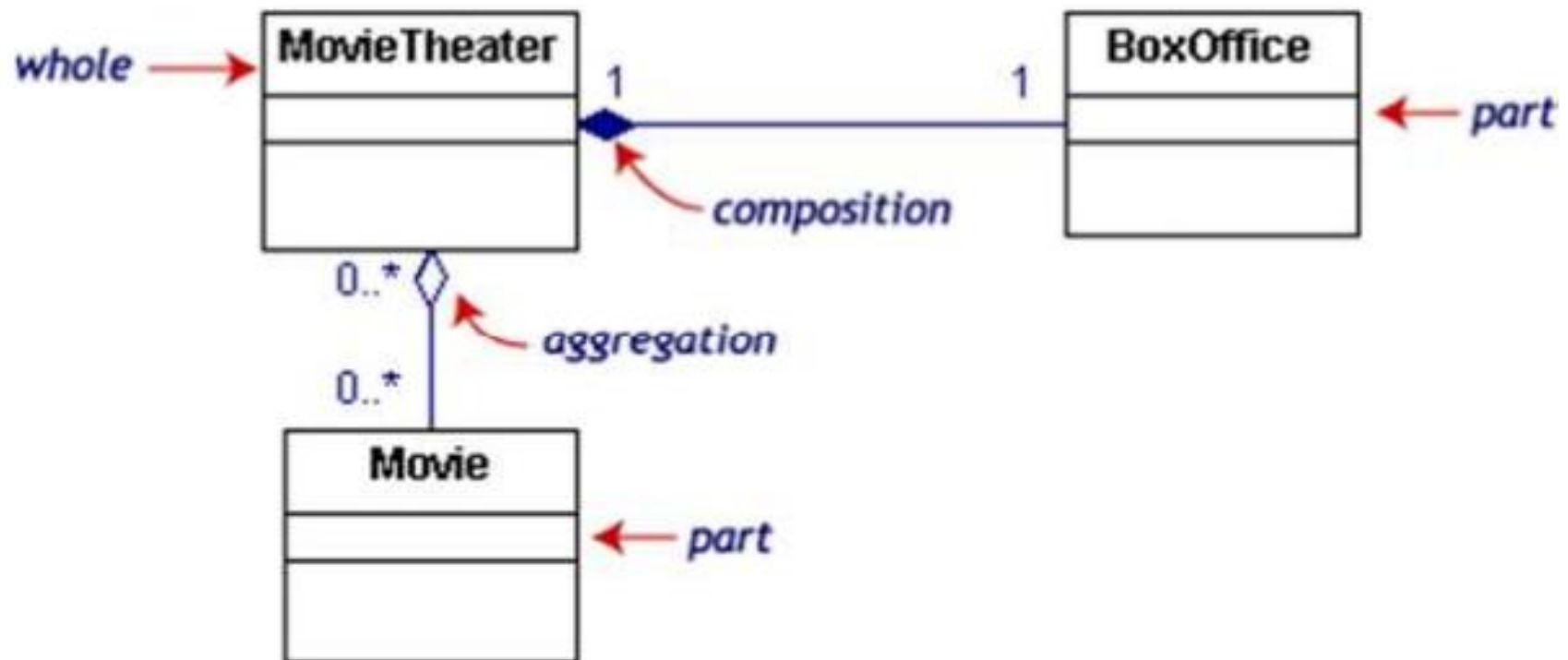
If the movie theater goes away
    so does the box office => composition
    but movies may still exist => aggregation

Class diagram example

Customer
name
address

No arrows; info can flow in both directions

Order
date
status
calcTax
calcTotal
calcTotalWeight

1   0..*

association

Aggregation – Order class contains OrderDetail classes. Could be composition?

abstract class

Payment
amount

1..*   1

generalization

Credit
number
type
expDate
authorized

Cash
cashTendered

Check
name
bankID
authorized

role name

line item   1..*

multiplicity

OrderDetail
quantity
taxStatus
calcSubTotal
calcWeight

0..*   1

Item   class name
shippingWeight
description
getPriceForQuantity
getWeight

attributes

operations

navigability

UML example: people

Let's add the visibility attributes

Sequence Diagram

- The Sequence Diagram **models the collaboration of objects based on a time sequence.**

- **Sequence diagrams** represent the objects participating in the interaction horizontally and time vertically.

Sequence Diagrams

- Depicts **sequence of actions** that occur in a system

- Useful tool to **represent dynamic behavior** of a system

- 2 dimensional :

  – Horizontal axis

  – Vertical axis

Sequence Diagram

- The focus is **less on messages** themselves and more on the **order in which messages.**

- Sequence diagrams are good at showing **which objects communicate** with which other objects; and what messages **trigger those communications.**

Participants in a Sequence Diagram

- A sequence diagram is made up of a **collection of participants.**

- Participants – the system parts that interact each other during the sequence.

- Classes or Objects – each class (object) in the interaction is represented by its named icon along the top of the diagram

Elements of Sequence Diagram

- ACTOR

- OBJECTS

- LIFELINES

- MESSAGES

- ACTIVATION-represents time an object needs to complete task

Object1:C1

Object2:C2

Click Update Button

User

updateStatus( )
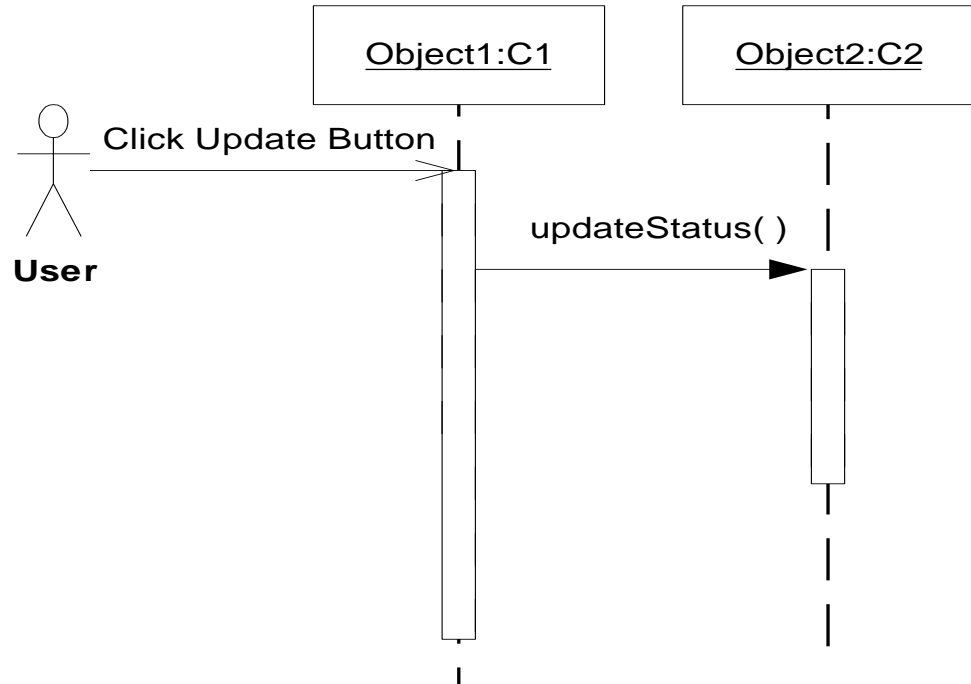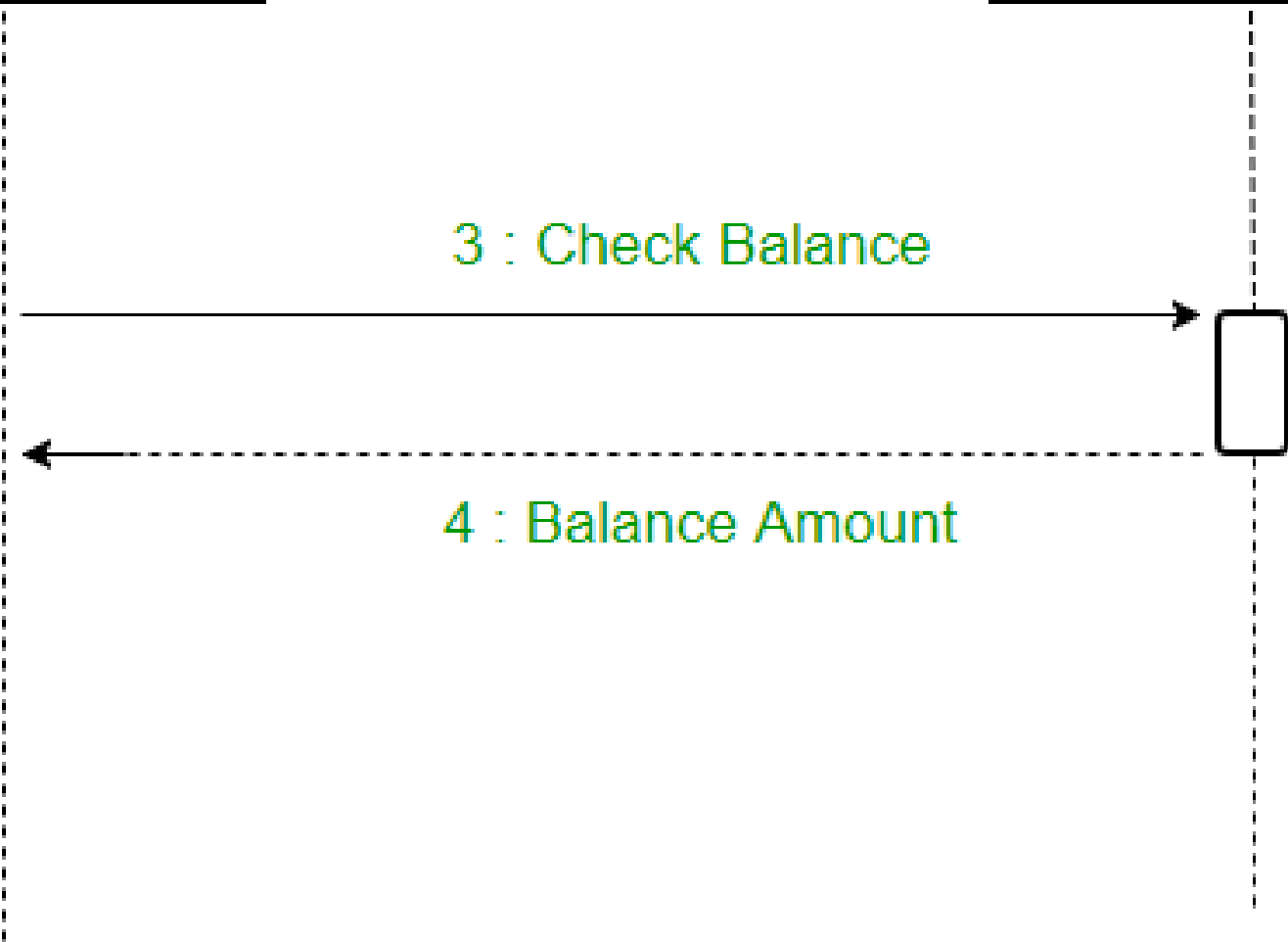
For example – Here the user in seat reservation system is shown as an actor where it exists outside the system and is not a part of the system.
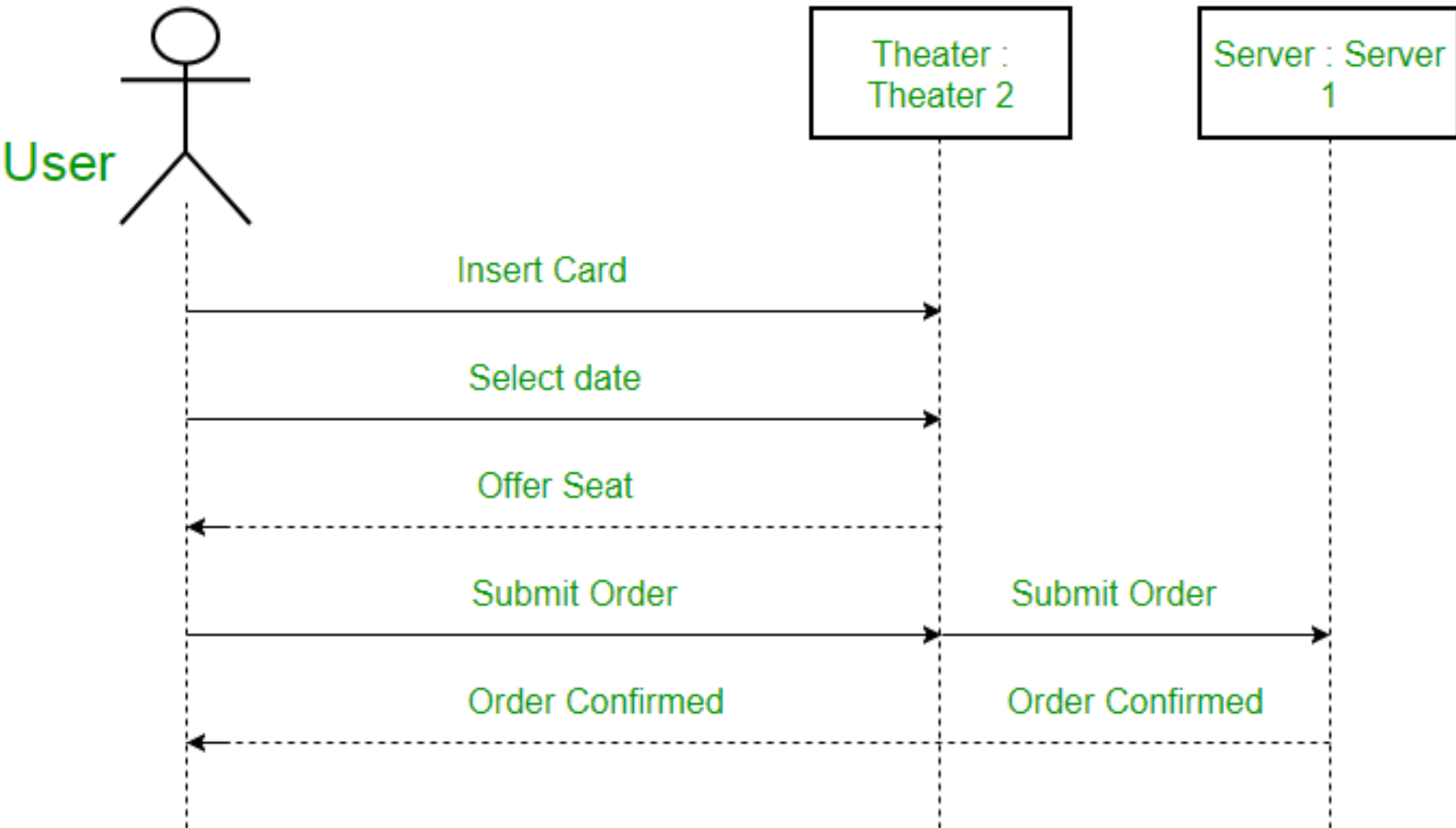
Elements of Sequence Diagram

- **Class Roles or Participants or Objects**: Class roles describe the way an object will behave in context.

:Object	component

Elements of Sequence Diagram

Activation or Execution Occurrence

- **Activation or Execution Occurrence**
  Activation boxes represent the time an object needs to complete a task.

- When an object is busy executing a process or waiting for a reply message, use a thin gray rectangle placed vertically on its lifeline.

Elements of Sequence Diagram

- **Synchronous Message**

- A synchronous message requires a response before the interaction can continue.

- It's usually drawn using a line with a solid arrowhead pointing from one object to another.

Synchronous

Elements of Sequence Diagram

- **Asynchronous Message**
  Asynchronous messages don't need a reply for interaction to continue.

- They are drawn with an arrow connecting two lifelines; however, the arrowhead is usually open and there's no return message depicted.

Simple, also used for asynchronous

Asynchronous

Elements of Sequence Diagram

- **Reply or Return Message**

- A reply message is drawn with a dotted line and an open arrowhead pointing back to the original lifeline.
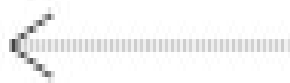
Reply or return message

Elements of Sequence Diagram

- **Self Message**

- A message an object sends to itself, usually shown as a U shaped arrow pointing back to itself.

Self message
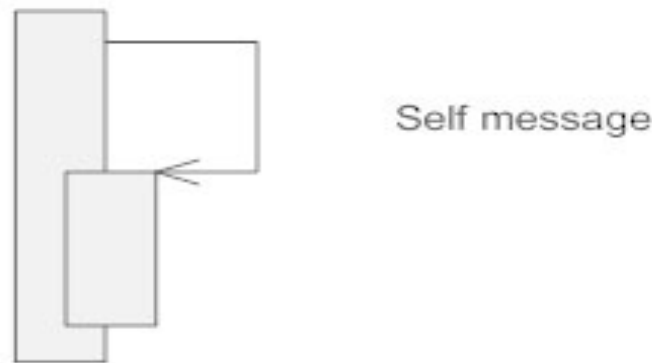
Elements of Sequence Diagram

- Lost: A lost message occurs whet the sender of the message is known but there is no reception of the message.

- Found: A found message indicates that although the receiver of the message is known in the current interaction fragment, the sender of the message is unknown.
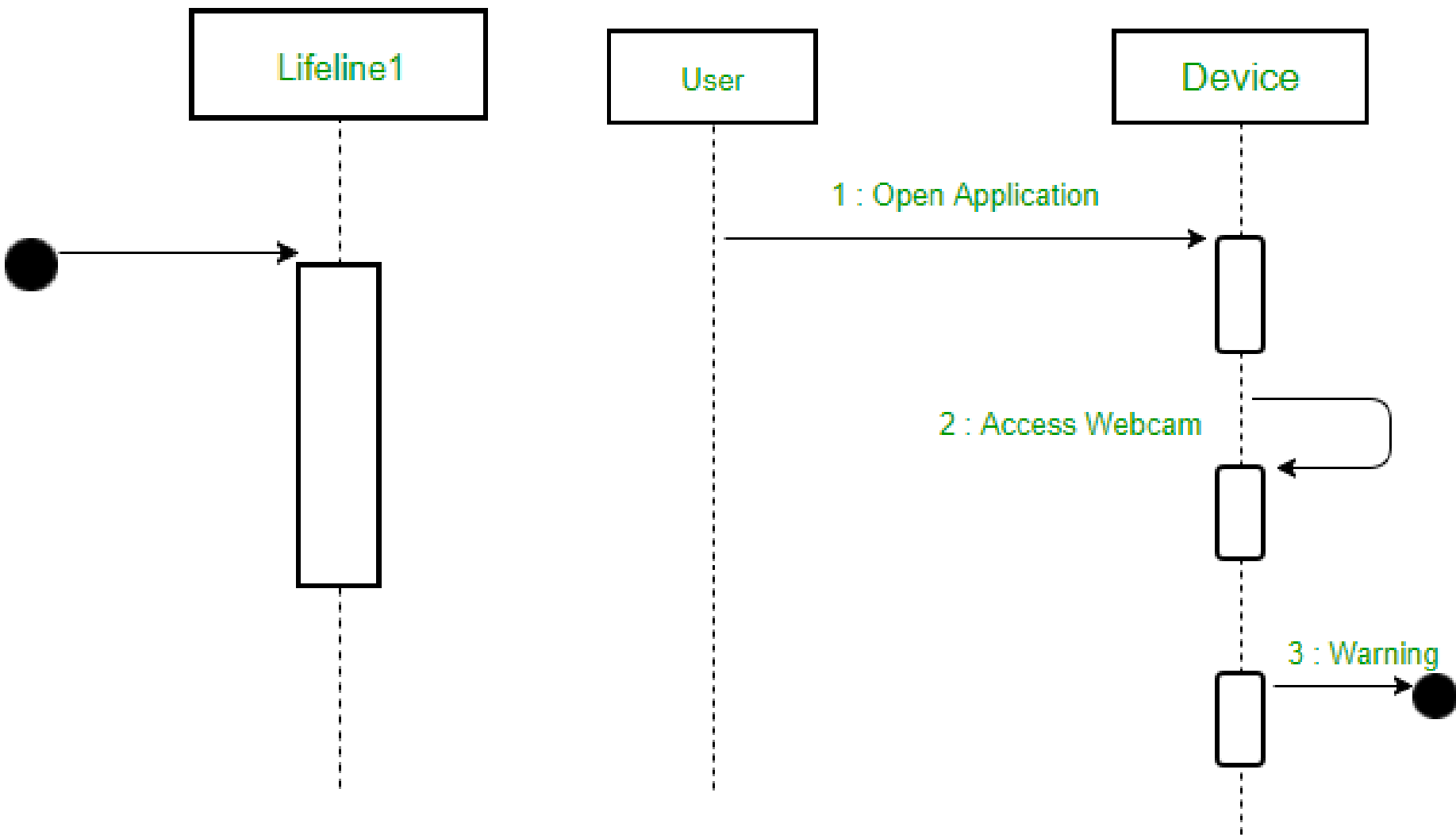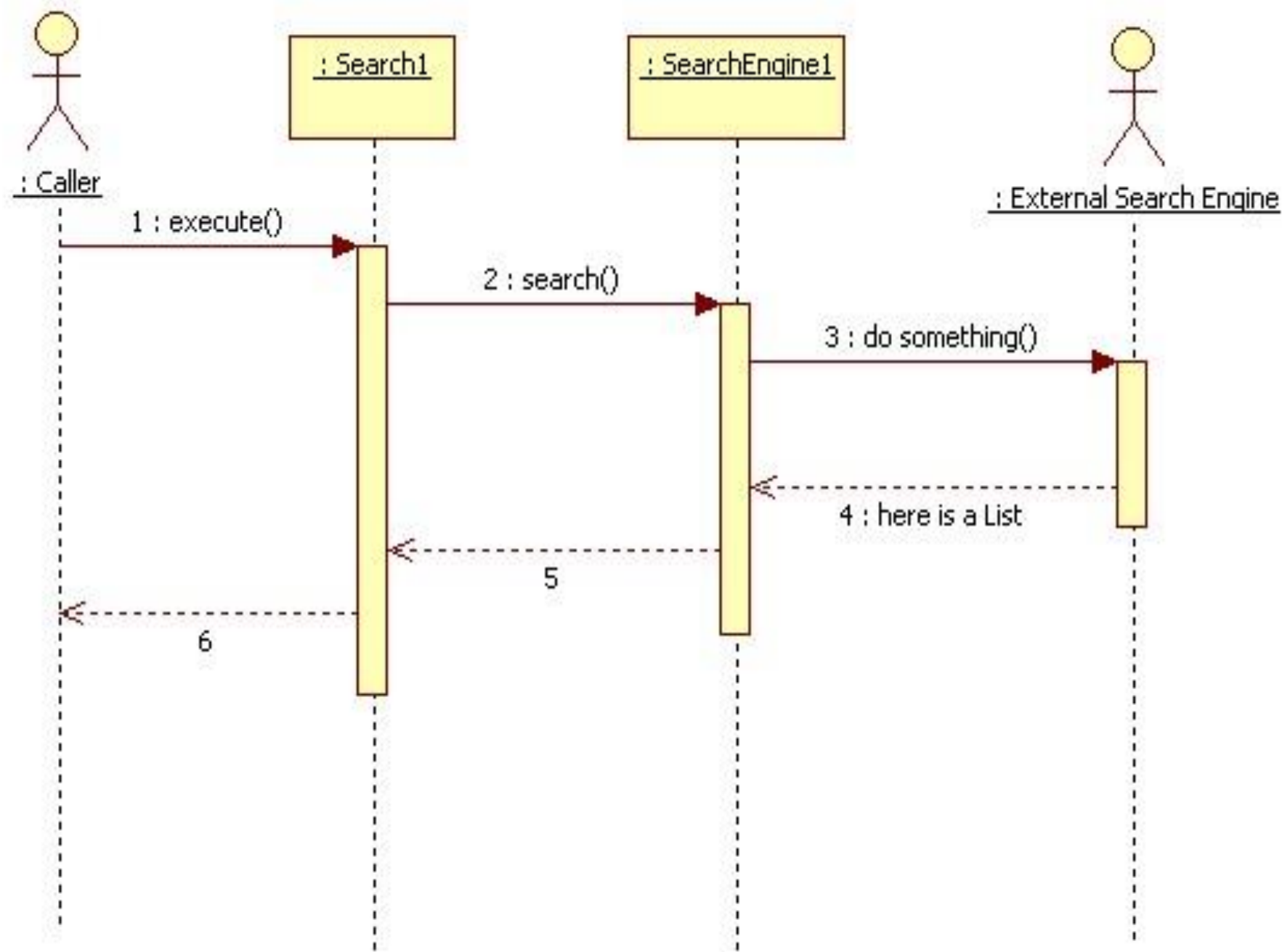
A sequence diagram for an emotion based music player

User | **Device** | **Database**

1 : Open Application

2 : Access Webcam

3 : Get Photo

4 : Detect face

5 : Retrieve Mood

7 : Display Mood

6 : Mood

8 : Retrieve Music

10 : Playlist

9 : Generated Playlist
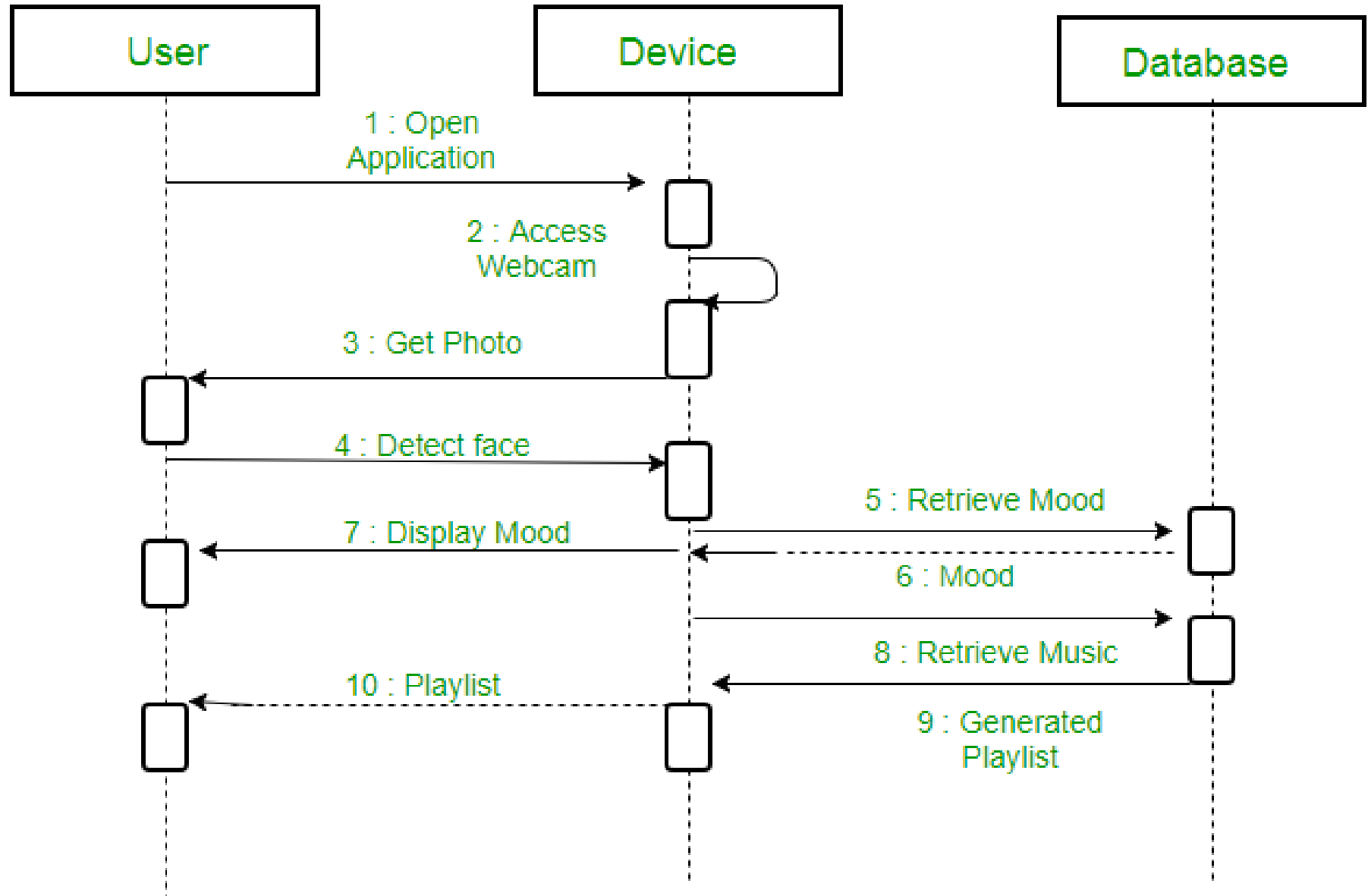
1. Firstly the application is opened by the user.
2. The device then gets access to the web cam.
3. The webcam captures the image of the user.
4. The device uses algorithms to detect the face and predict the mood.
5. It then requests database for dictionary of possible moods.
6. The mood is retrieved from the database.
7. The mood is displayed to the user.
8. The music is requested from the database.
9. The playlist is generated and finally shown to the user.

Uses of sequence diagrams

- Used to model and visualise the logic behind a sophisticated function, operation or procedure.

- They are also used to show details of UML use case diagrams.

- Used to understand the detailed functionality of current or future systems.

- Visualise how messages and tasks move between objects or components in a system.

Sequence diagrams: an example
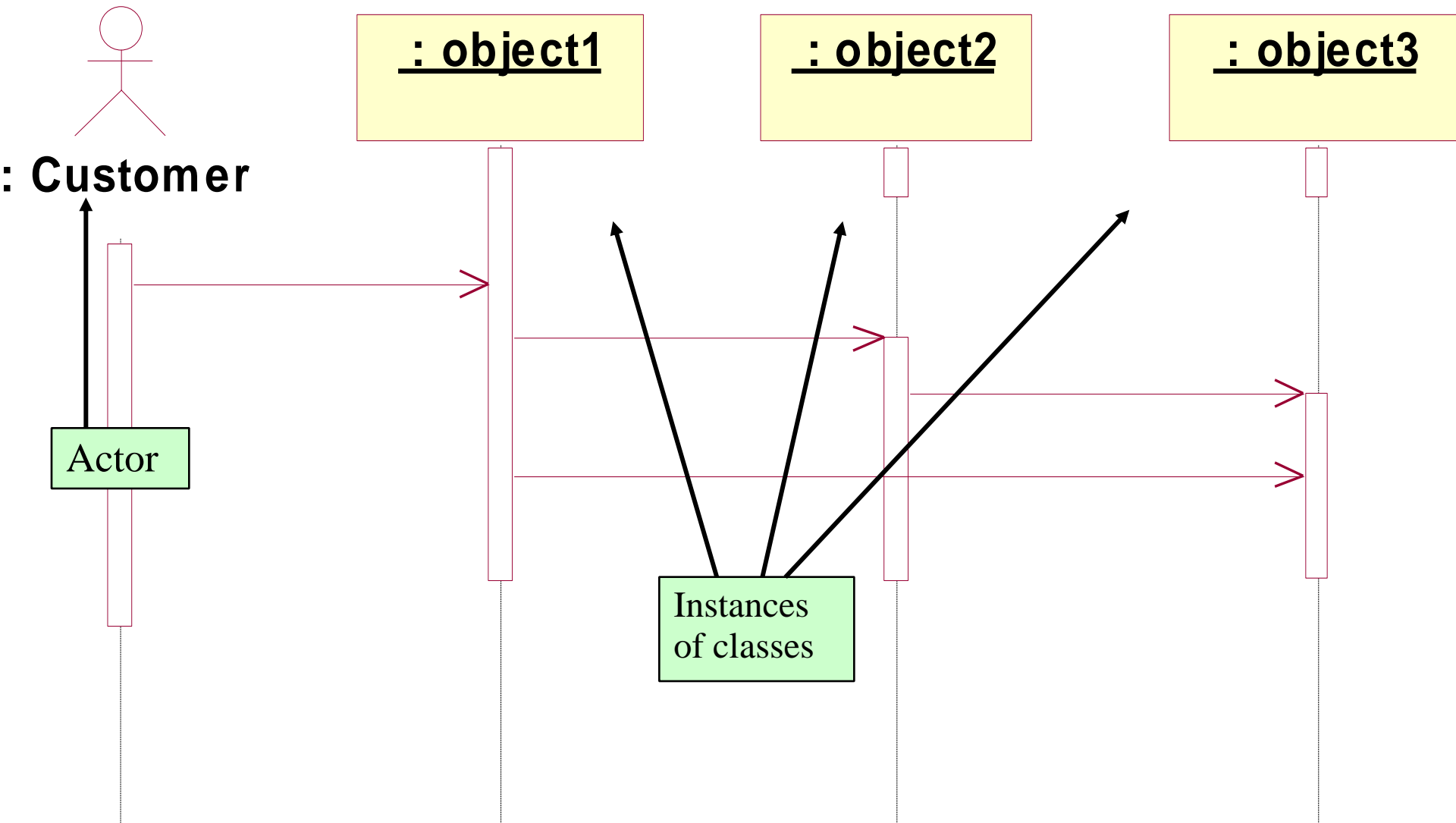
Sequence Diagram: Lifelines

:actor

:object1

:object2

:object3

Time

Each object has a lifeline.

Time flows from top to down.
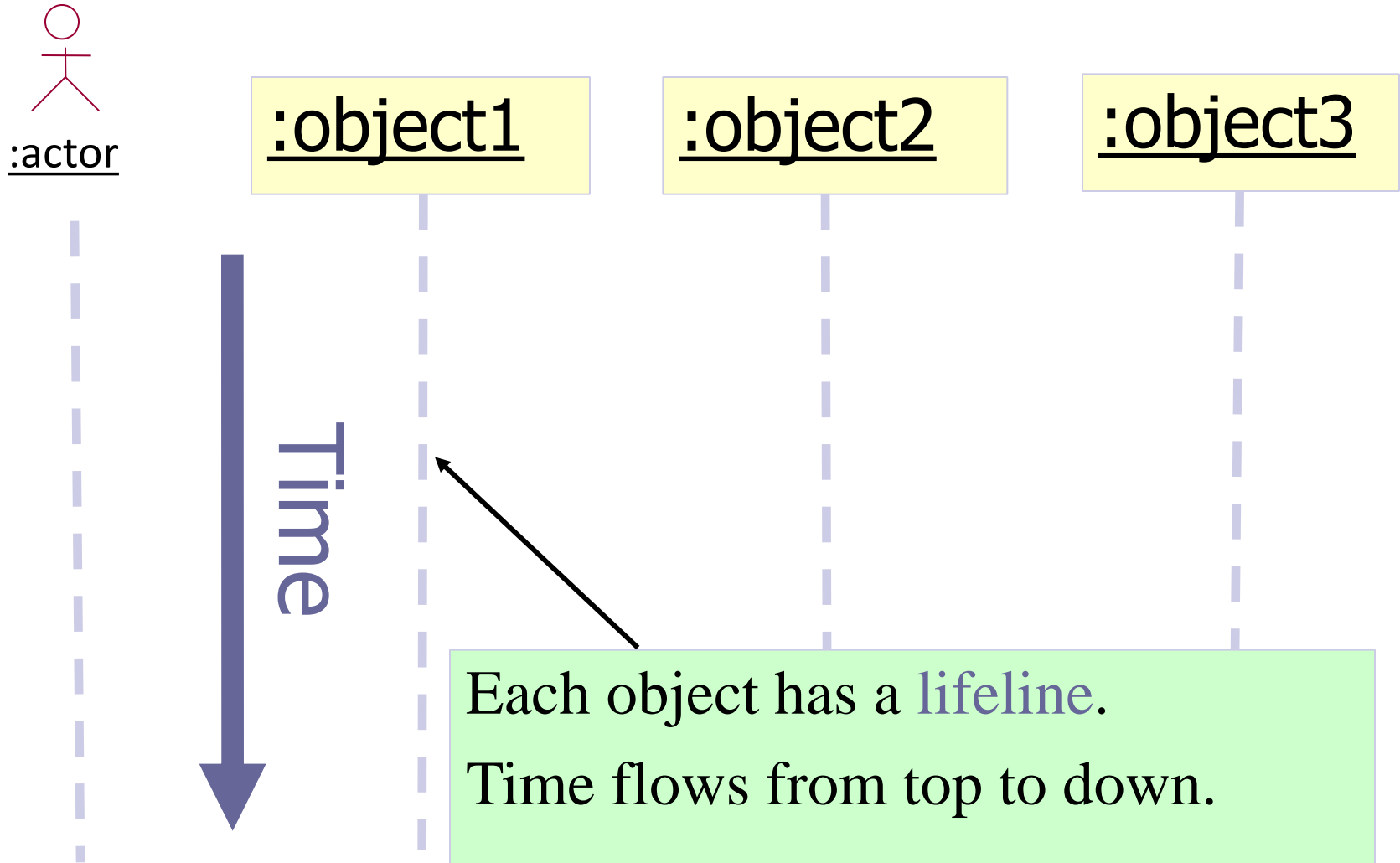
Sequence diagrams: rectangles

:actor

:object1

:object2

:object3

Time

Rectangles refer to events that are related to each other.
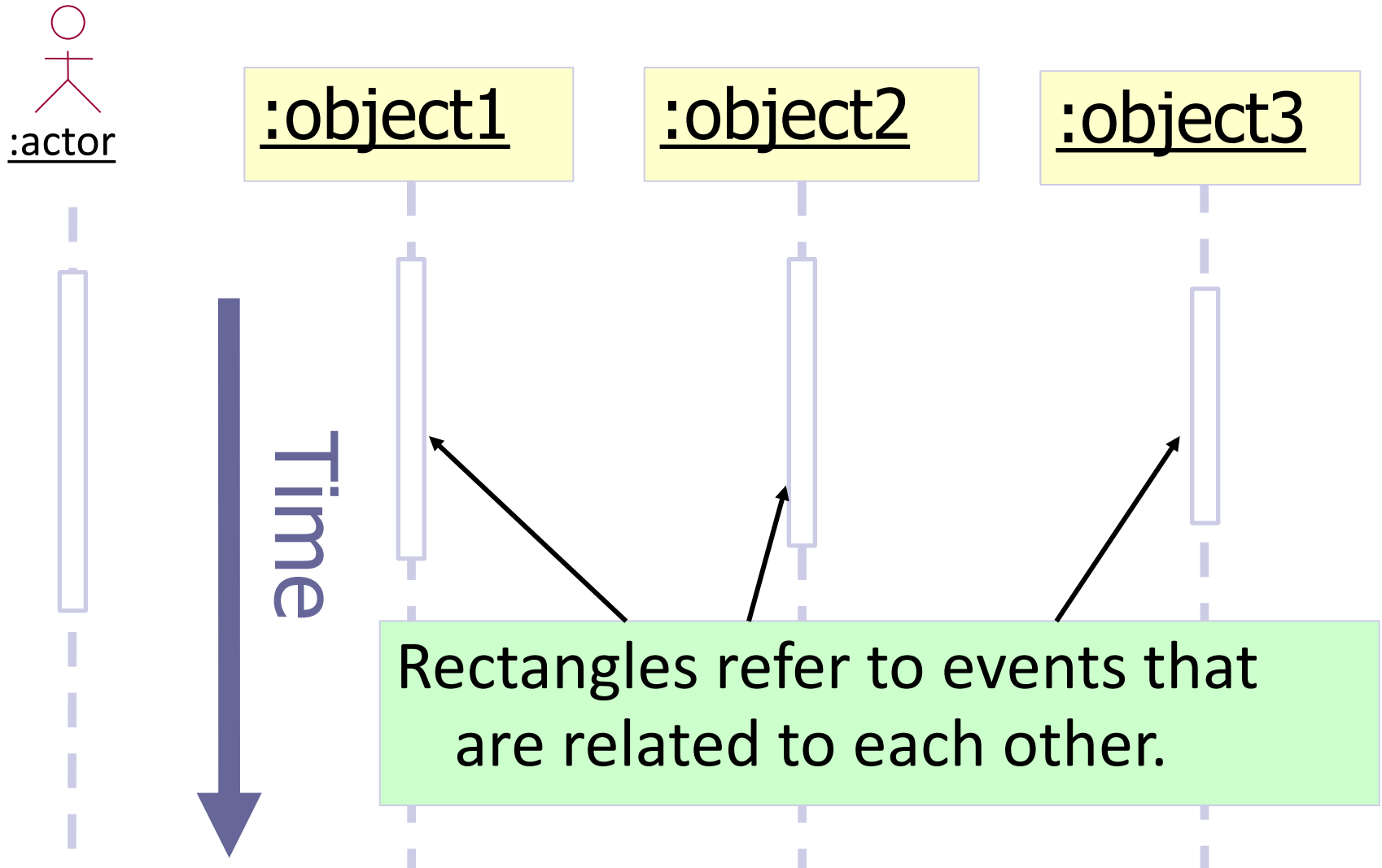
Sequence diagrams: messages

:actor

:object1

:object2

:object3

Time

Arrows indicate messages that are sent from one object to another.
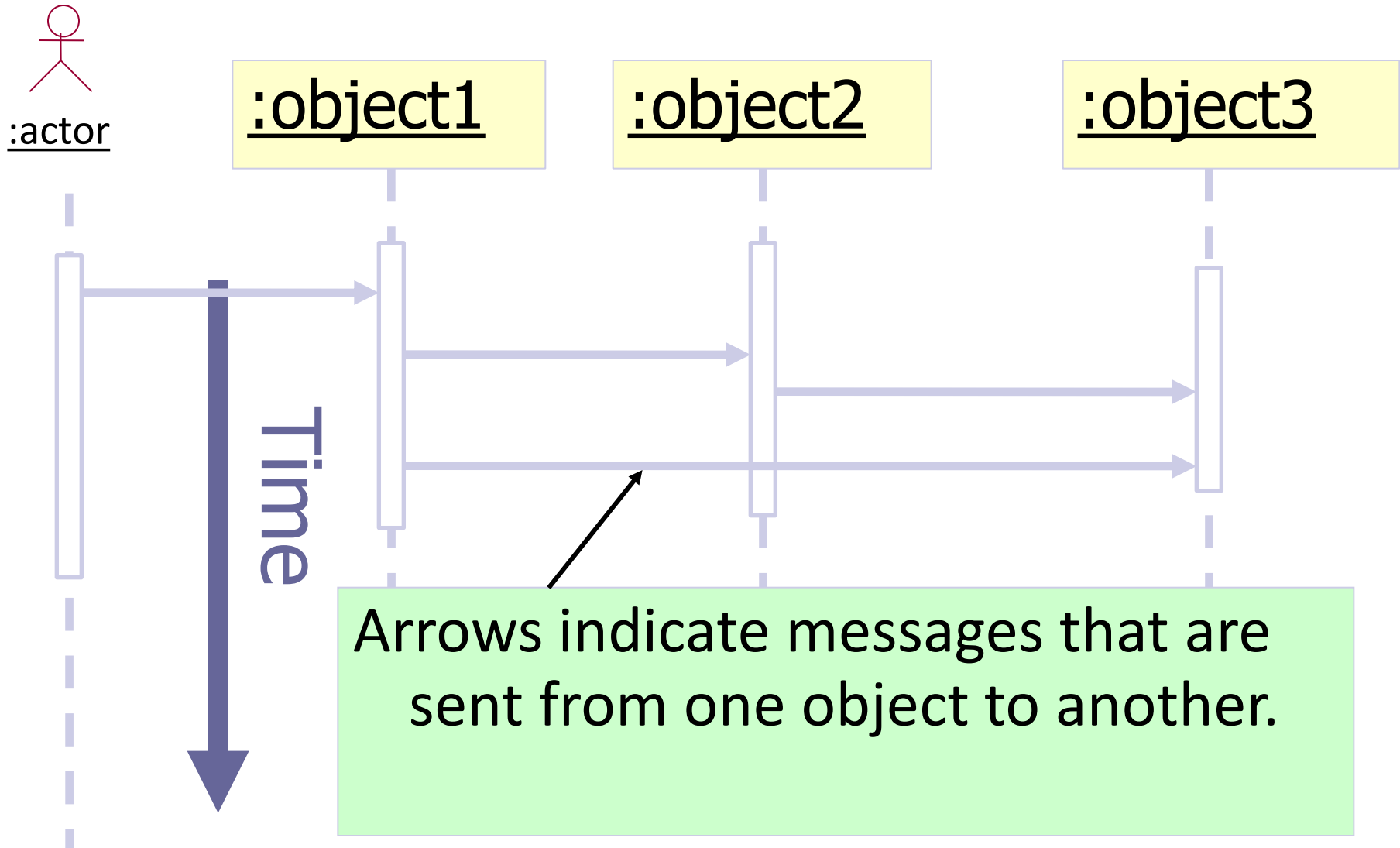
Sequence diagrams - different types of messages

: Customer

: object1

: object2

: object3

request( )

Message

createObject3( )

create( )

An object sends a message to itself.

Feedback arrow shows that a value is returned.
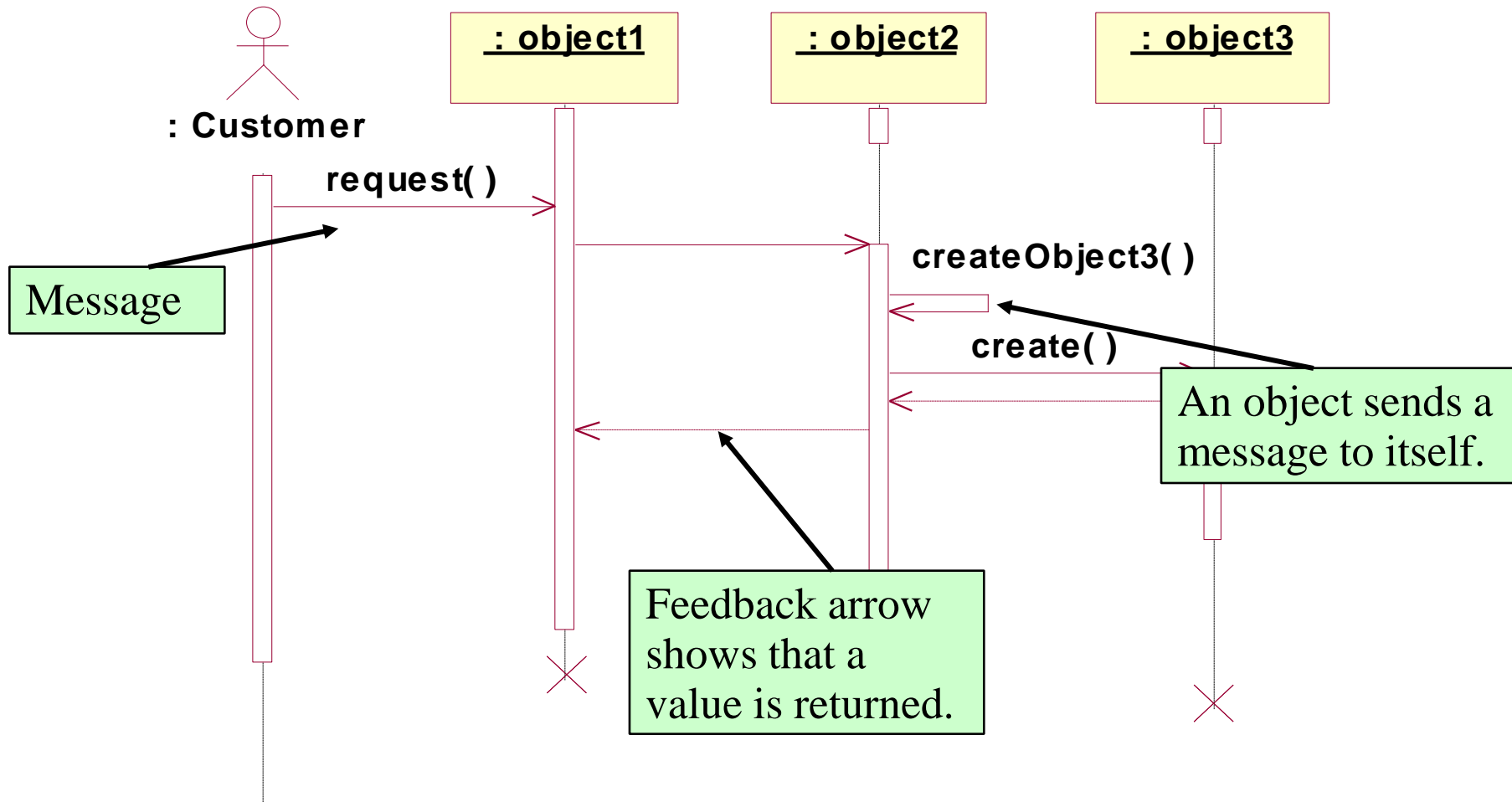
Sequence diagrams: endpoints

: object1

: object2

: object3

: Customer

request( )

createObject3( )

create( )

X at the end of the lifeline shows that the object ceases to exist.
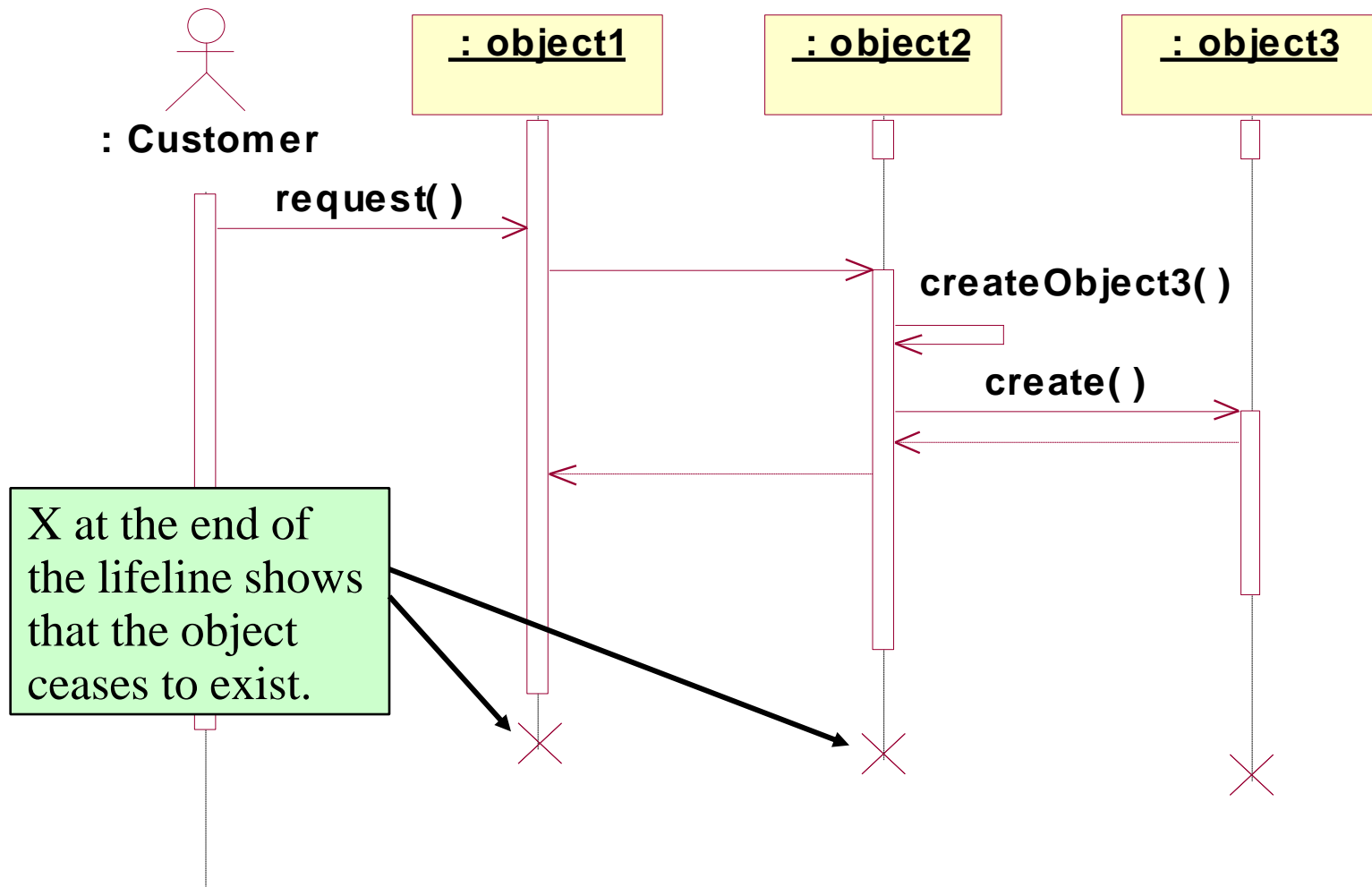
Sequence Diagram

- Draw sequence diagram for login procedure of a system. Include all possible scenarios and also draw activity diagram. (10M)
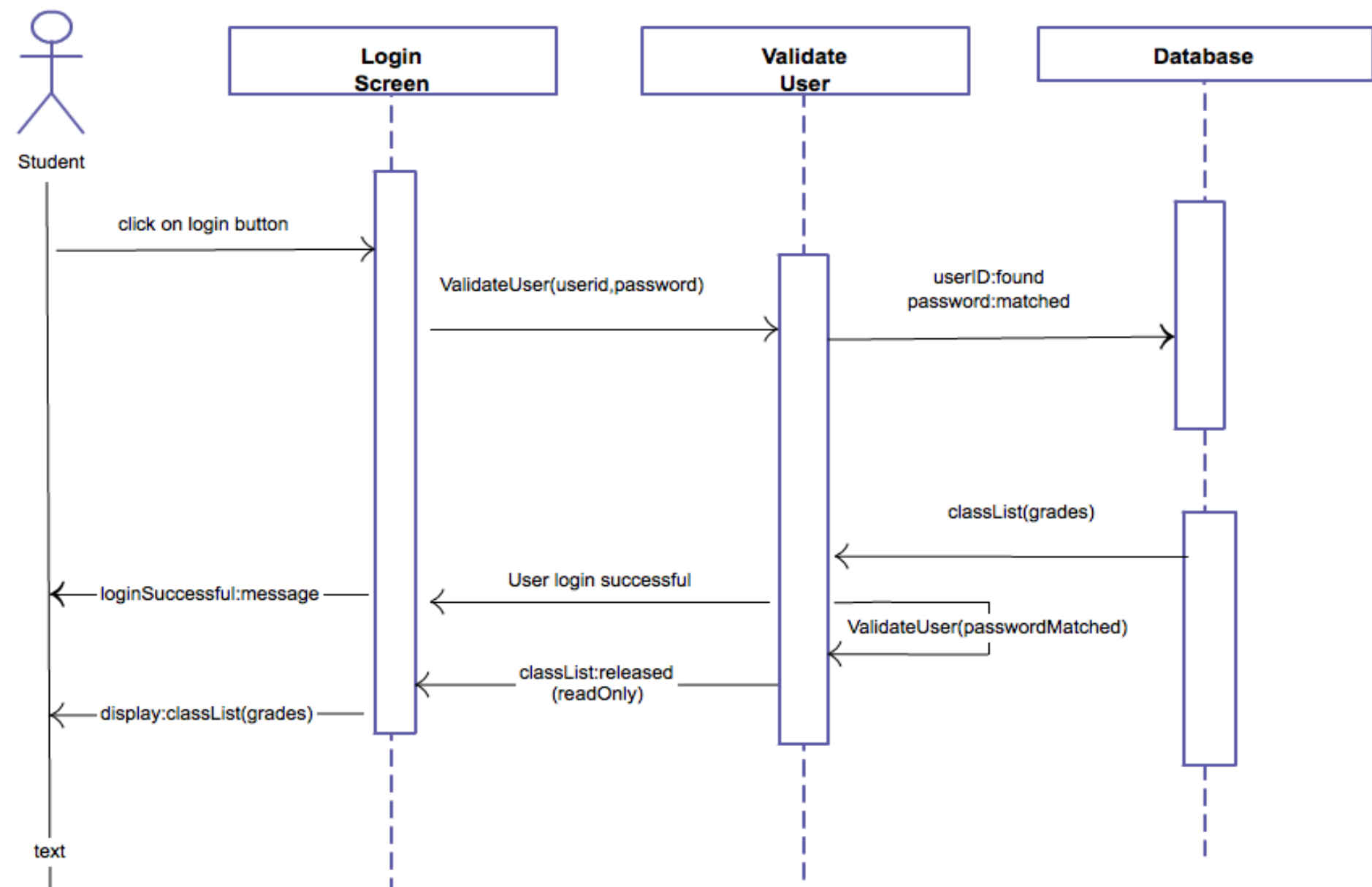
COLLABORATION DIAGRAM

- A **collaboration diagram** is a type of **visual presentation** that shows how various software <u>objects</u> interact with each other within an <u>overall IT architecture</u>

- How users can benefit from this **collaboration**.

- A **collaboration diagram** often comes in the form of a visual **chart** that resembles a flow**chart**.

Ex. Collaboration Diagram
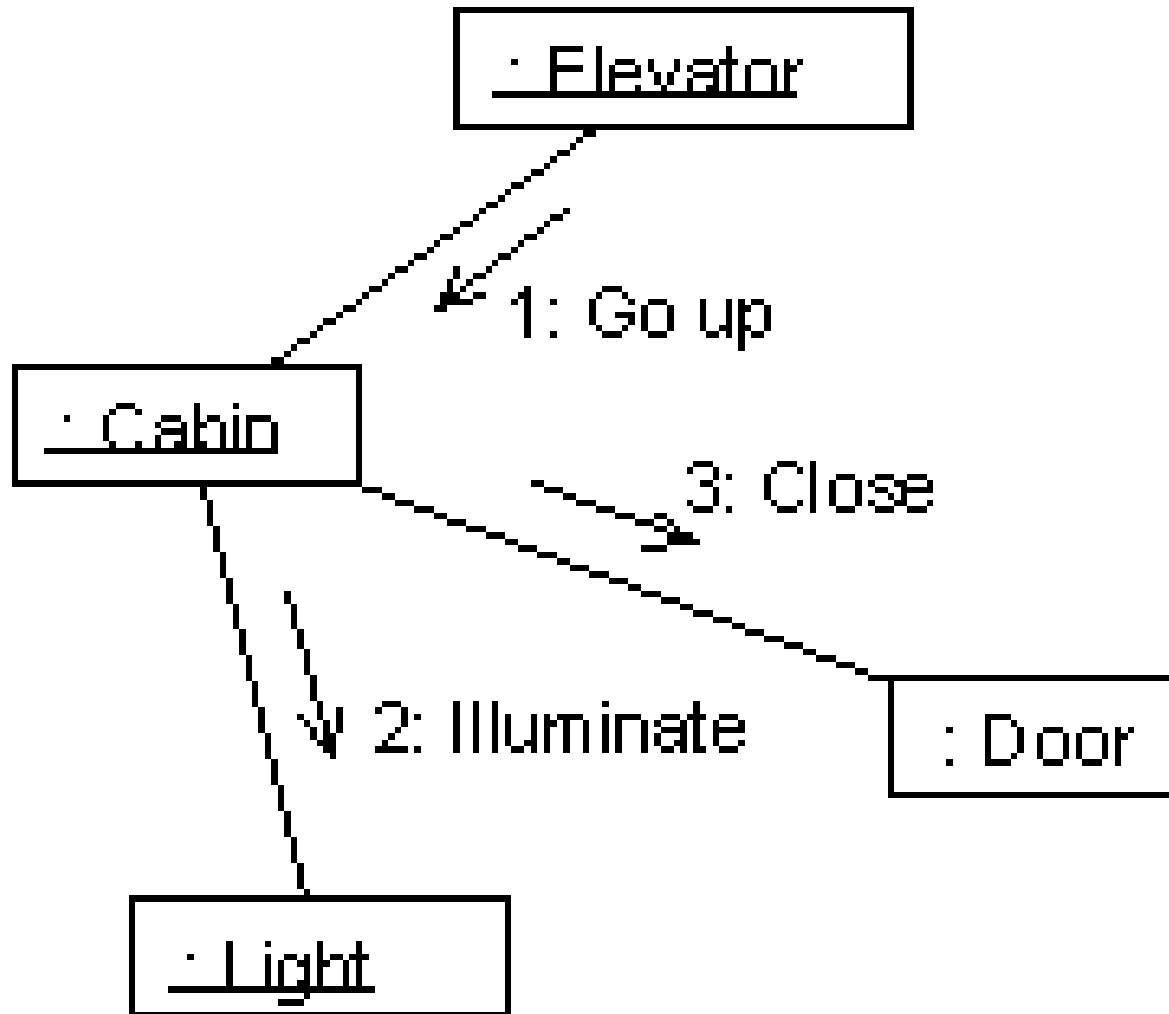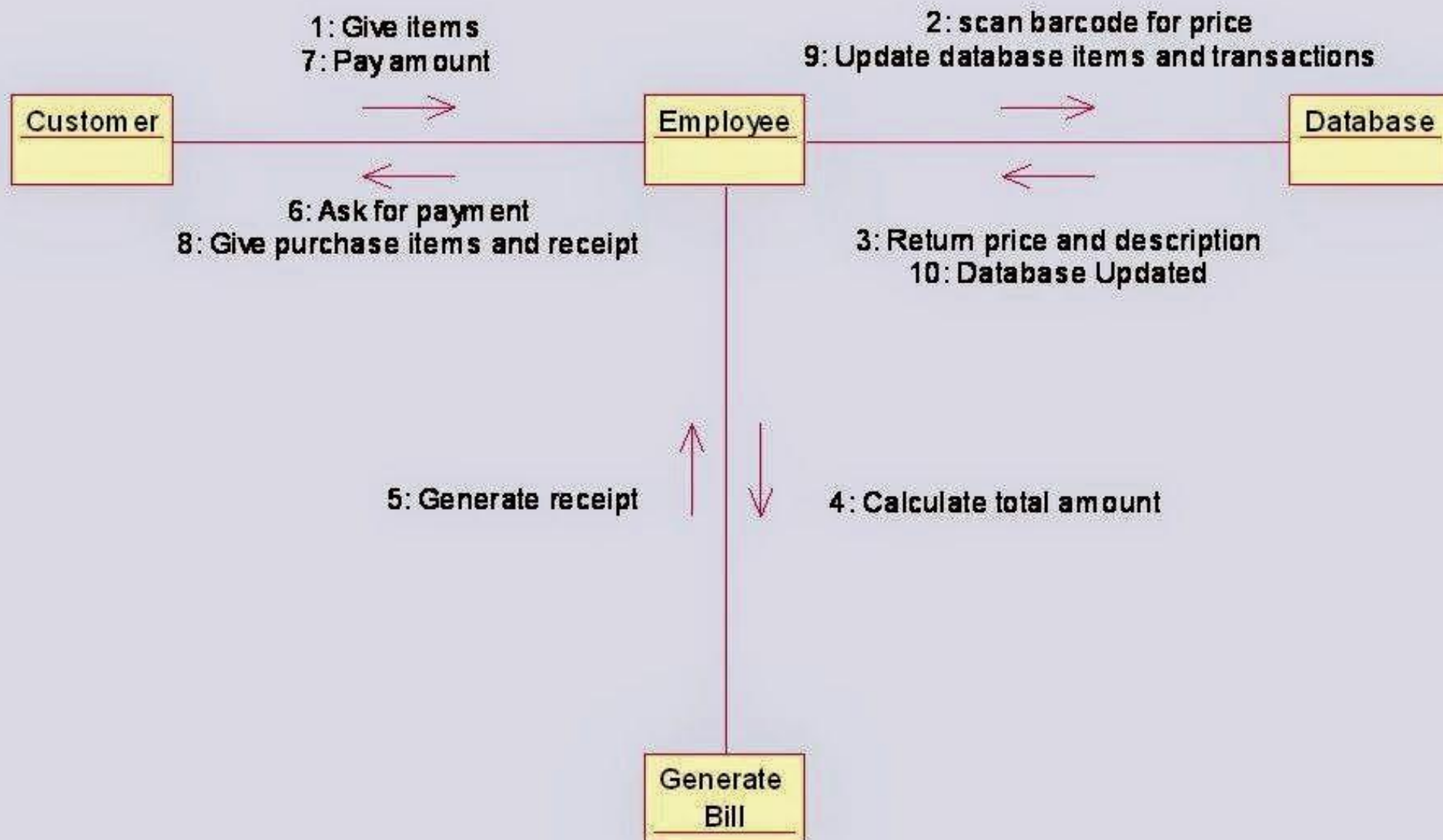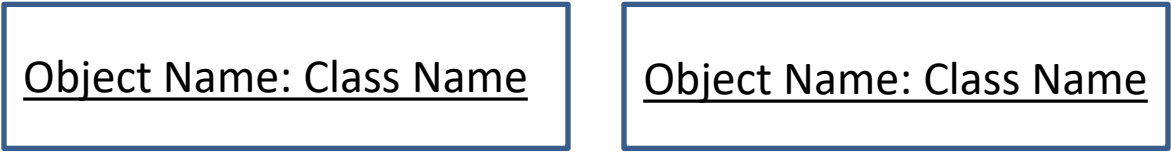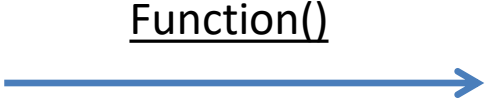
Collaboration Diagram

- **Collaboration diagram**-emphasis is on structural organization of the objects that send/receive messages.

➤ Elements of a collaboration diagram

- Object

 | Object Name: Class Name | Object Name: Class Name |
  |---|---|

- Relation/Association
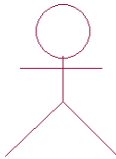
- Messages

  Function()

STEPS TO DRAW COLLABORATION DIAGRAM

- Place objects as vertices in a graph

- Add links to connect these object as arcs of graph

- Write messages on links with sequence numbers for send and receive

Collaboration Diagram Syntax

AN ACTOR	Operator
AN OBJECT	anObject:aClass
AN ASSOCIATION	_____
A MESSAGE	a Message() ⟶
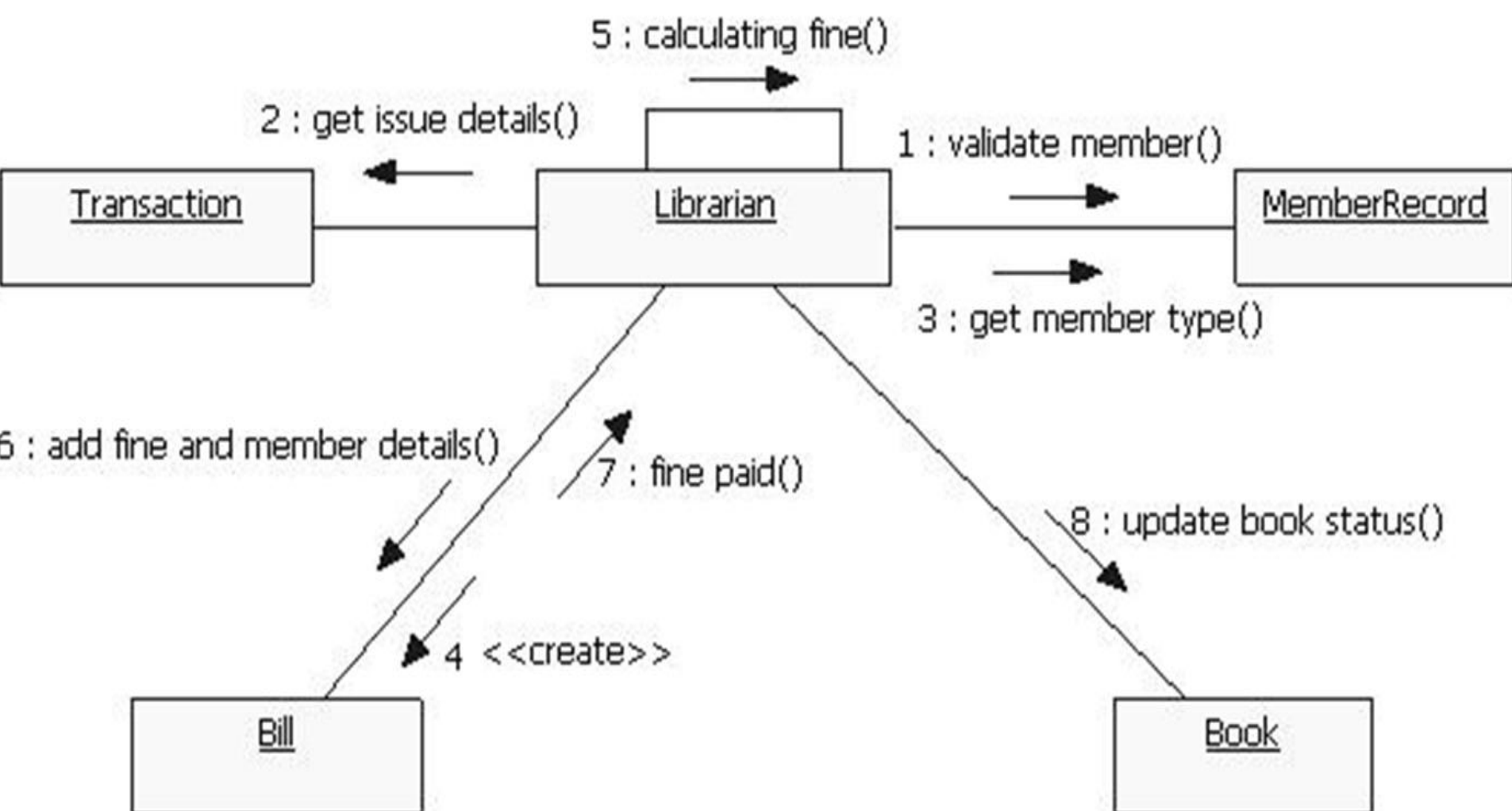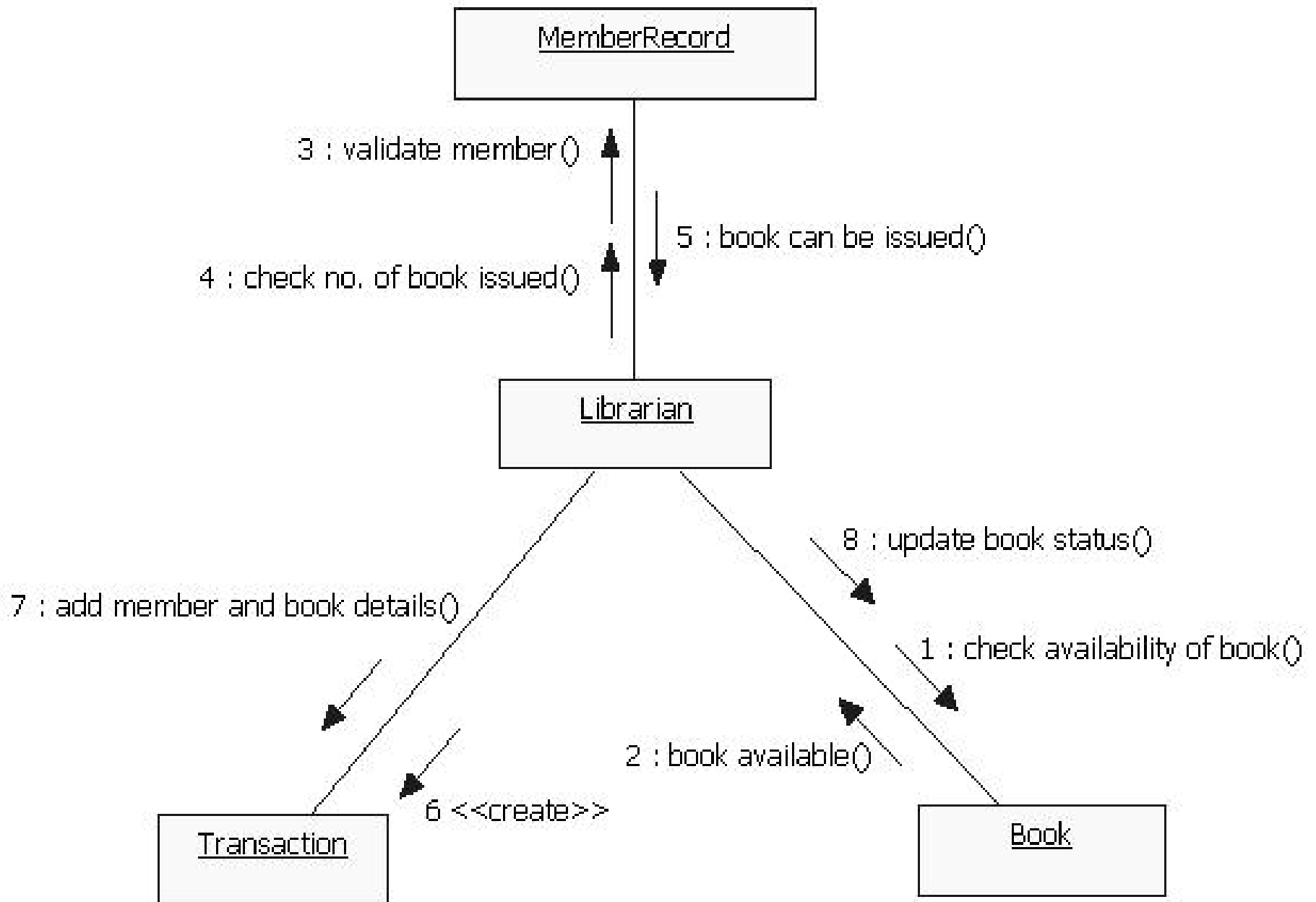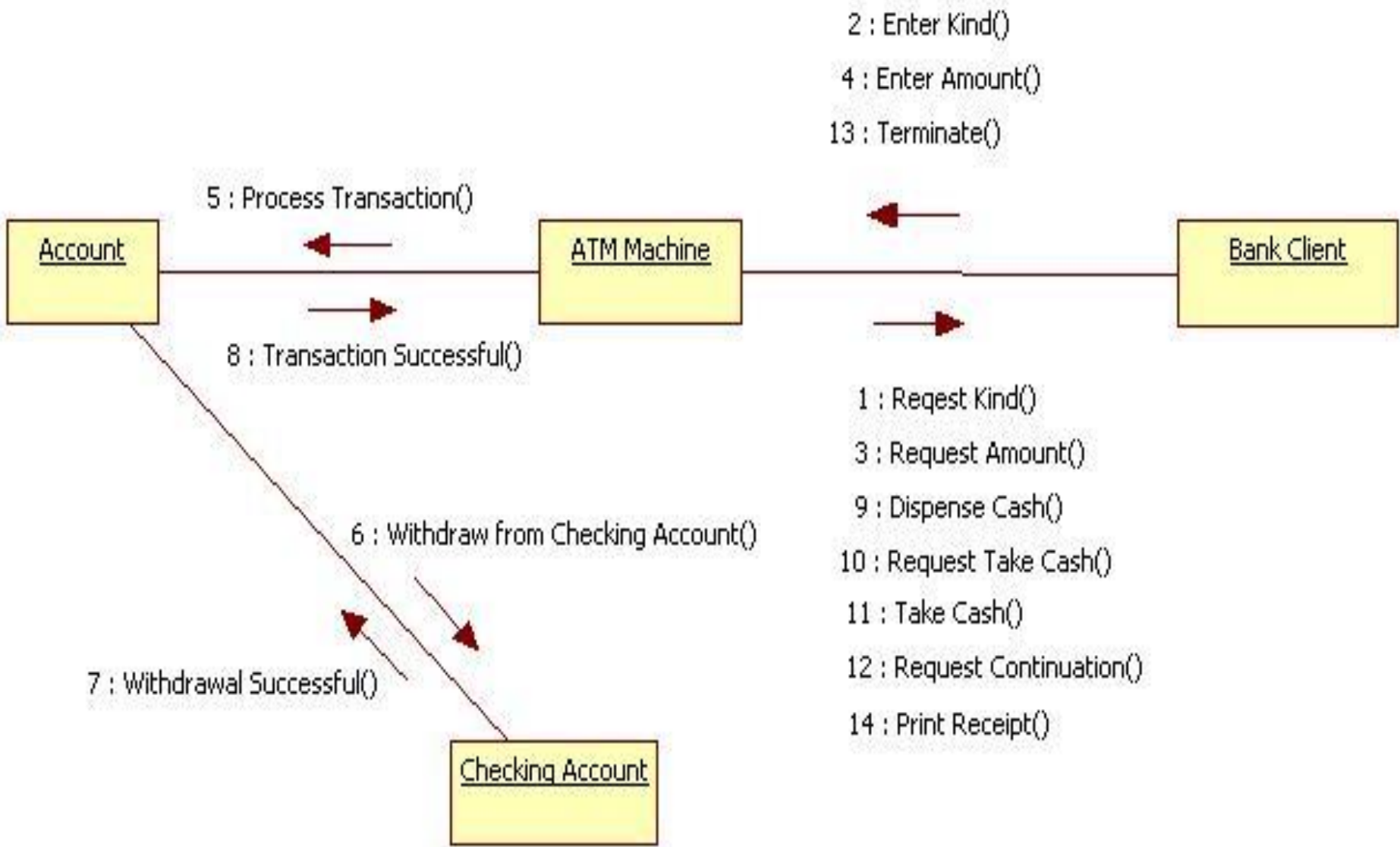
State Chart Diagram

- A **state** in UML is a condition or **situation** an object (in a system) might find itself in during its life time.

- A state chart diagram is normally used to **model how the <u>STATE</u> of an object changes** in its lifetime.

- We **capture** the behavior of the subject object through modeling these various states and transitions between them.

State Chart Diagram

- Thus, a state machine diagram does not necessarily model all possible states, but rather the **critical** ones only.

  - When we say "critical" states, we mean those that act as **stimuli** and **prompt** for **response** in the external world.

State diagram for user verification
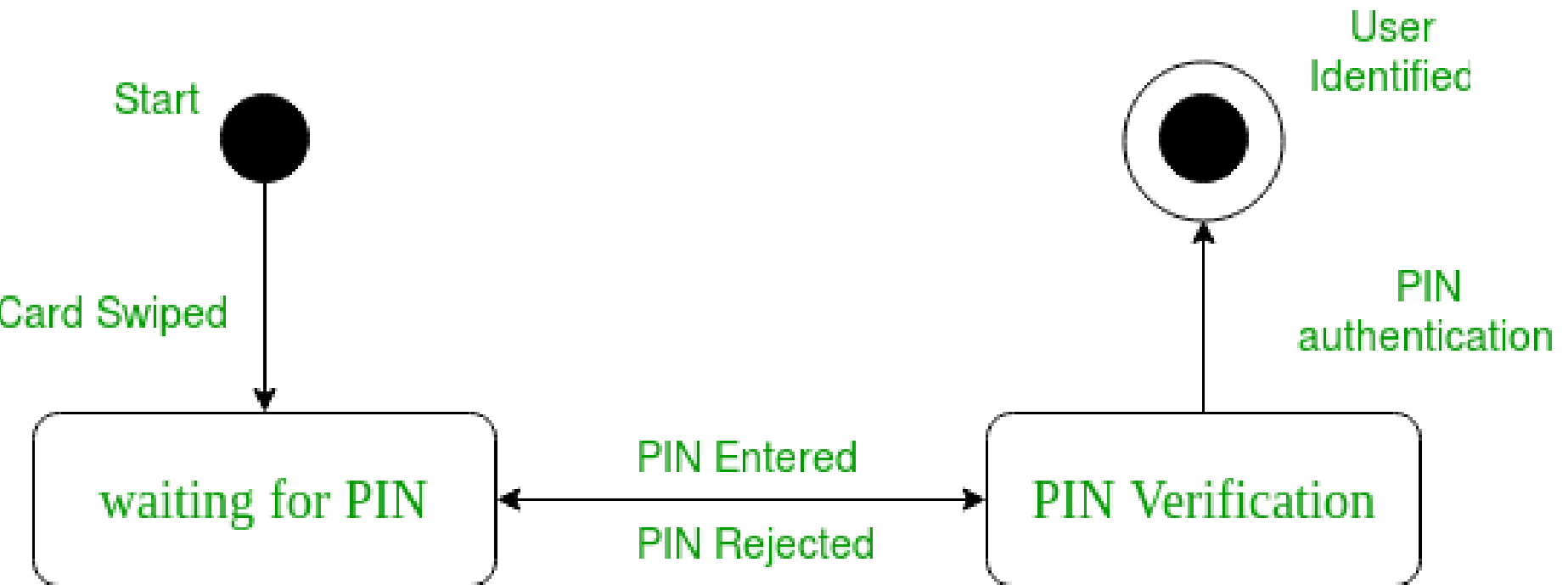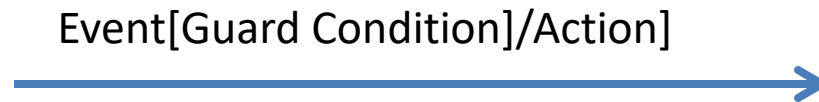
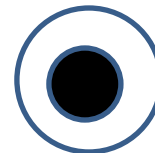# Elements of state diagram

- Initial State

- State

State

- Transition

Event[Guard Condition]/Action]

- Final States

State Chart Diagram

- Initial state: This is represented as a filled circle.
- Final state: This is represented by a filled circle inside a larger circle.

- State: These are represented by rectangles with rounded corners.

- Transition: A transition is shown as an arrow between two states.
  - Normally, the name of the event which causes the transition is places along side the arrow.

State Chart Diagram

- A guard condition is a condition that has to be met in order to enable the transition to which it belongs:
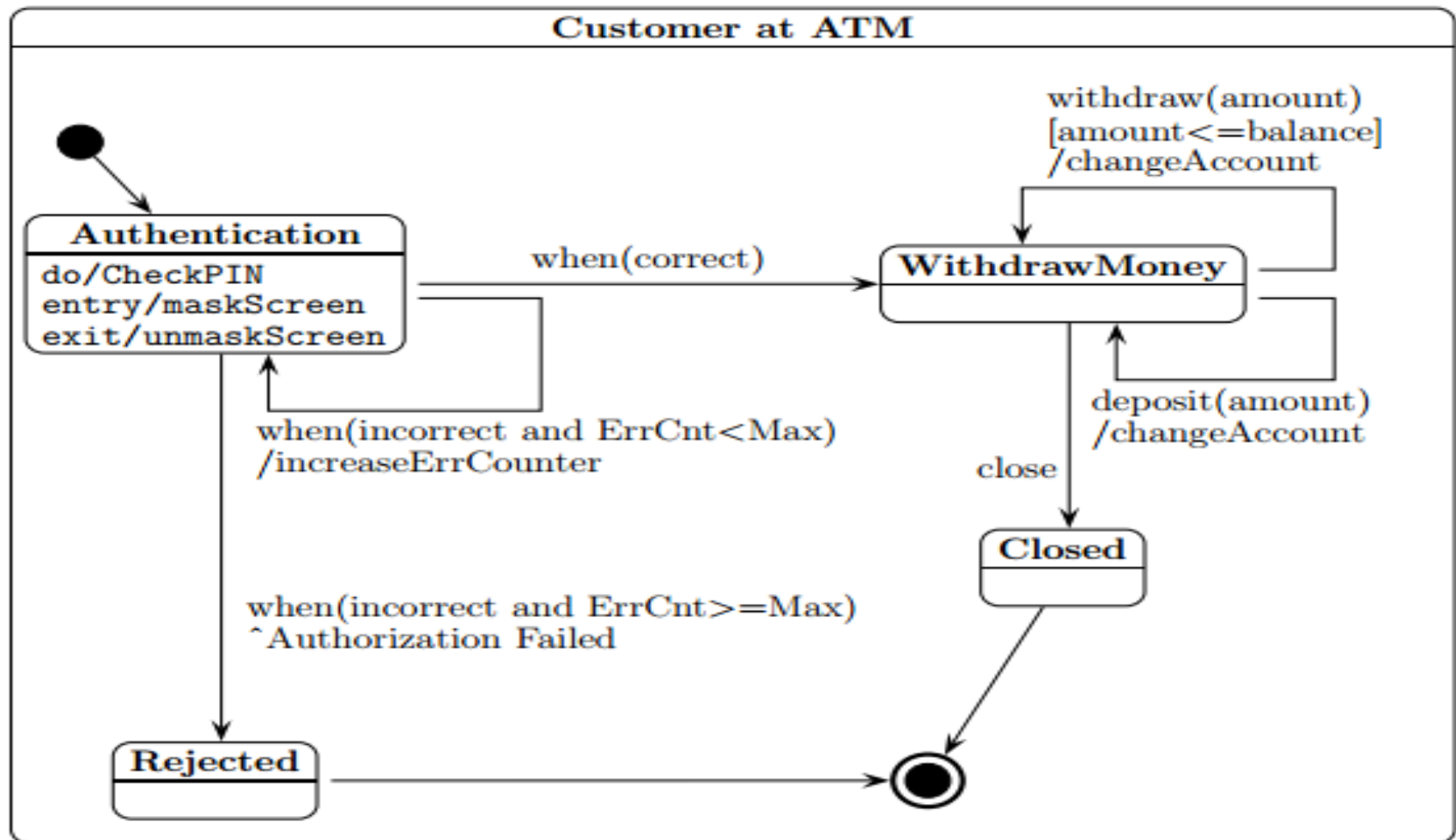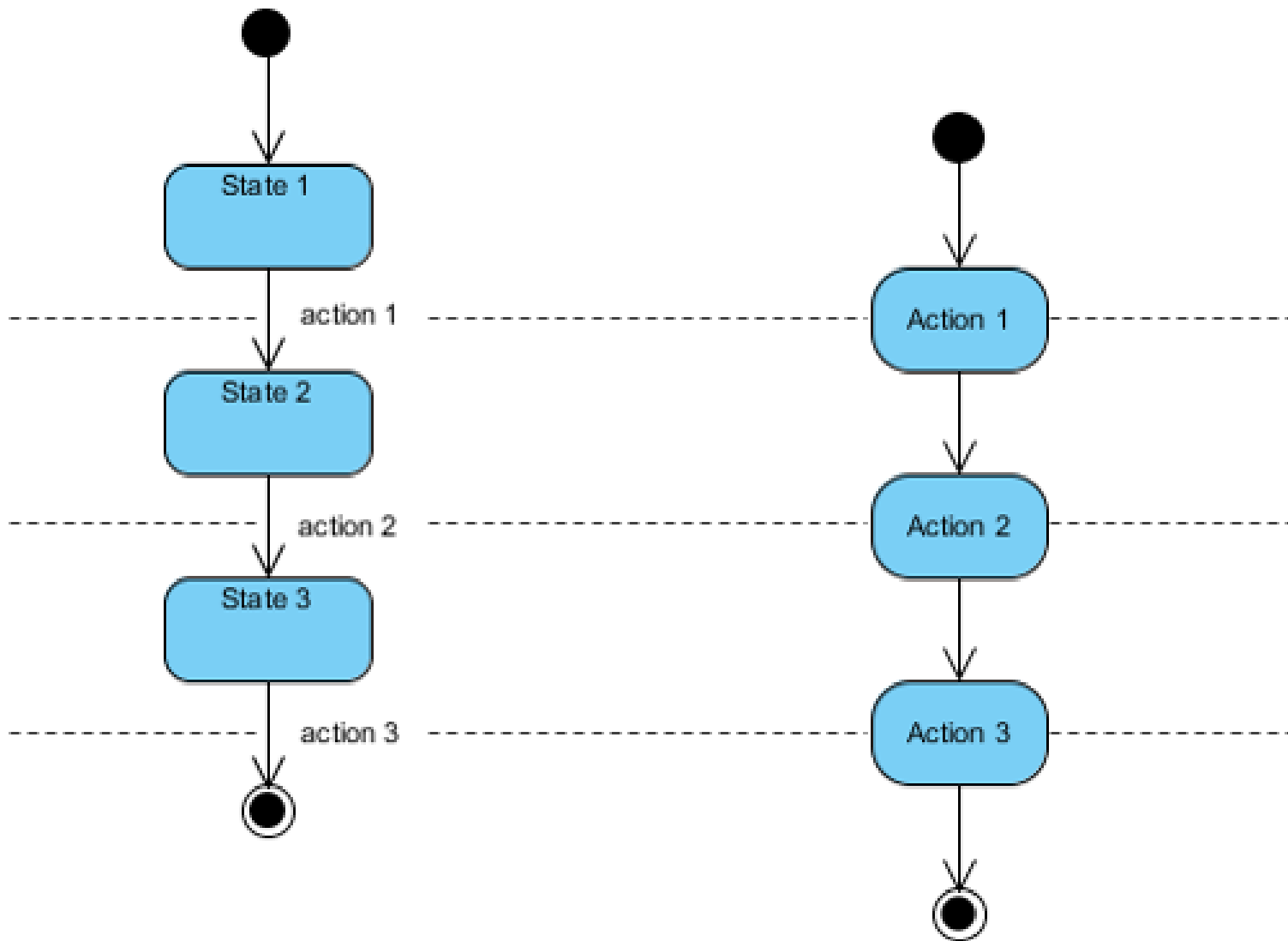
[Guard Condition]

- Guard conditions can be used to document that a certain event, depending on the condition, can lead to different transitions.
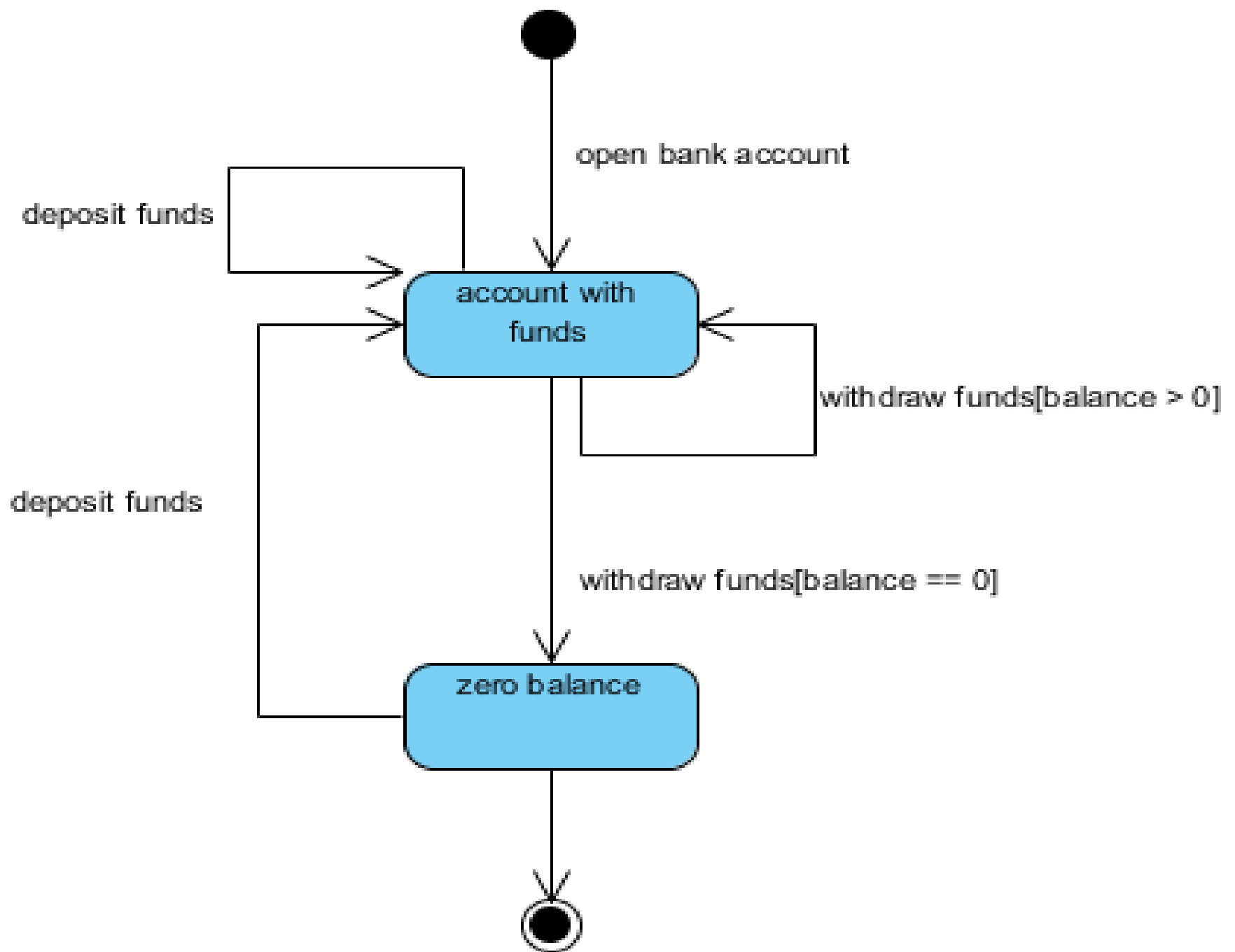
Uses of state chart diagram

- To state the **events responsible for change** in state
  - (we do not show what processes cause those events).
- We use it to **model the dynamic behavior** of the system .
- To **understand the reaction of objects**/classes to internal or external stimuli.

ATM Withdraw

Email Example