UML

The Unified Modeling Language (UML) is a graphical language for OOAD that gives a standard way to write a software system's blueprint. It helps to visualize, specify, construct, and document the artifacts of an object-oriented system. It is used to depict the structures and the relationships in a complex system.

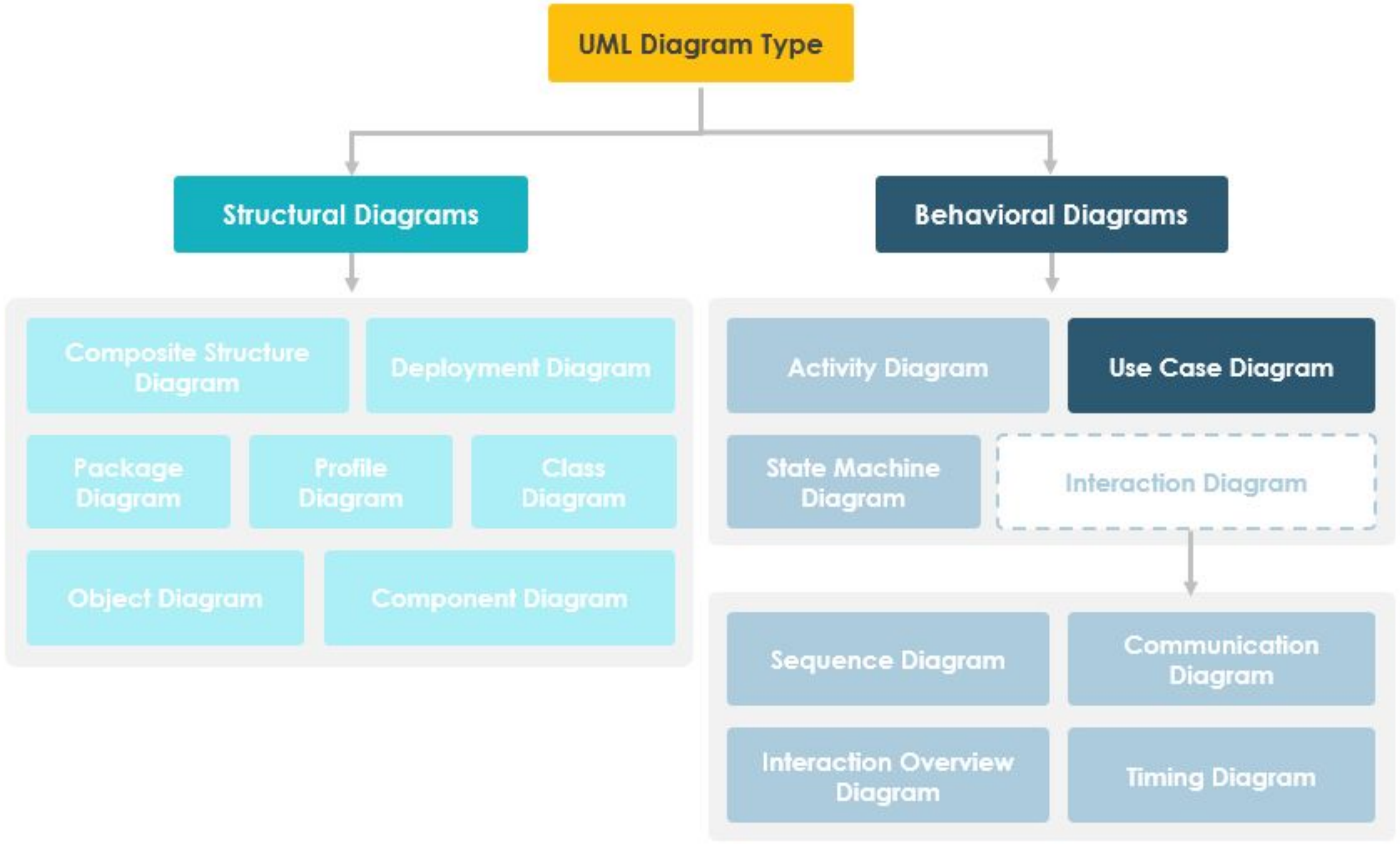
There are 2 basic kinds of UML Diagram, namely −

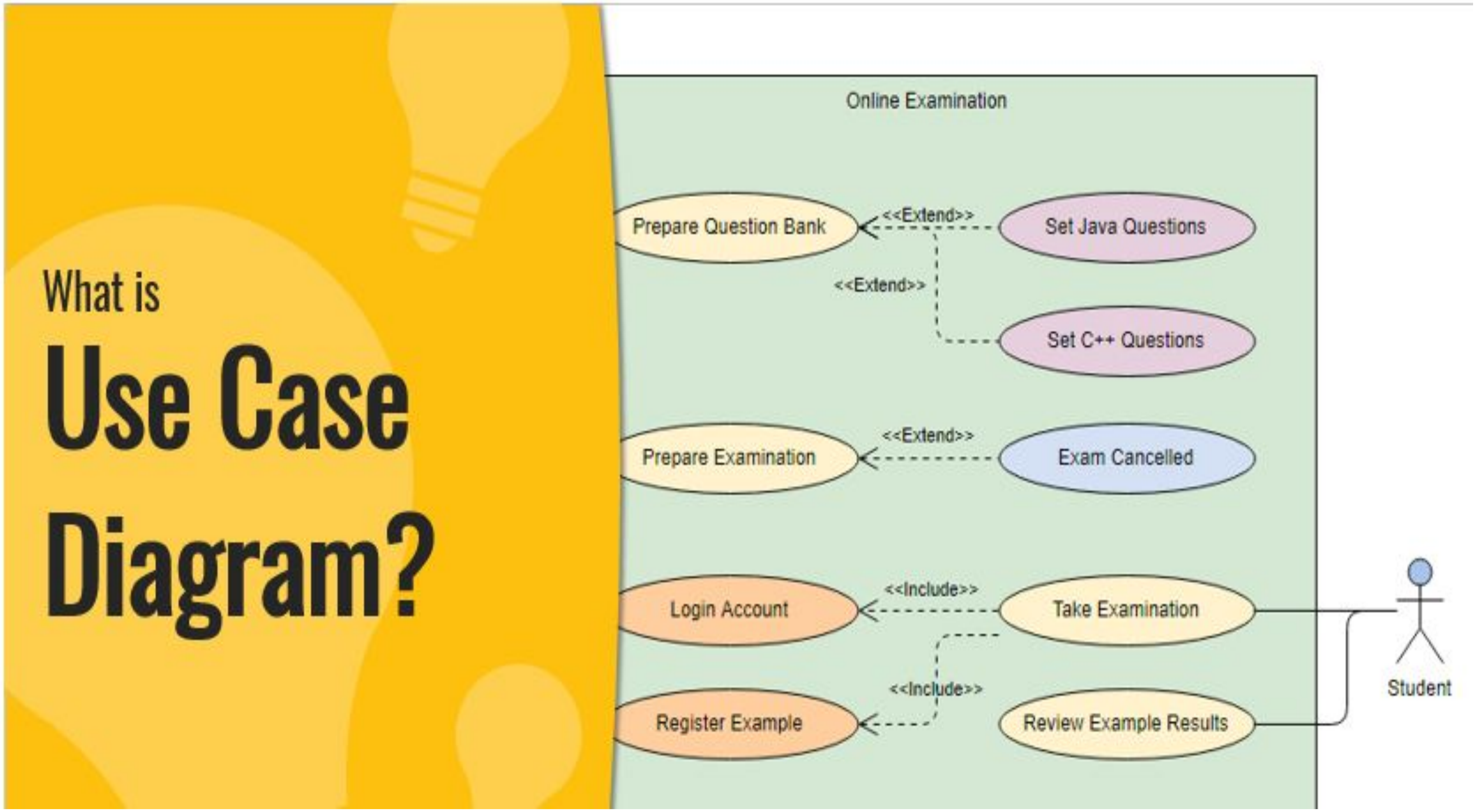
Structure Diagrams(How the system does it /Technical)

- Class Diagram
- Component Diagram
- Deployment Diagram
- Object Diagram
- Package Diagram

Behavioral Diagrams (What the system does/Functional)

- Use Case Diagram
- Activity Diagram
- State Machine Diagram
- Sequence Diagram
- Timing Diagram


UML Diagram Type
Structural Diagrams
Behavioral Diagrams
Composite Structure Diagram
Deployment Diagram
Package Diagram
Profile Diagram
Class Diagram
Object Diagram
Component Diagram
Activity Diagram
Use Case Diagram
State Machine Diagram
Interaction Diagram
Sequence Diagram
Communication Diagram
Interaction Overview Diagram
Timing Diagram


What is
Use Case
Diagram?
Online Examination
Prepare Question Bank
<<Extend>>
Set Java Questions
<<Extend>>
Set C++ Questions
Prepare Examination
<<Extend>>
Exam Cancelled
Login Account
<<Include>>
Take Examination
Student
<<Include>>
Register Example
Review Example Results

What is a use case diagram?
Why Use case diagram?
Why use cases?.

Use cases specify the expected behavior (what), and not the exact method of making it happen (how).
Use cases once specified can be denoted both textual and visual representation (i.e. use case diagram).
A key concept of use case modeling is that it helps us design a system from the end user's perspective.
It is an effective technique for communicating system behavior in the user's terms by specifying all externally visible system behavior.
A use case diagram is usually simple. It does not show the detail of the use cases:
It only summarizes **some of the relationships** between use cases, actors, and systems.
It does **not show the order** in which steps are performed to achieve the goals of each use case.

Origin of Use Case

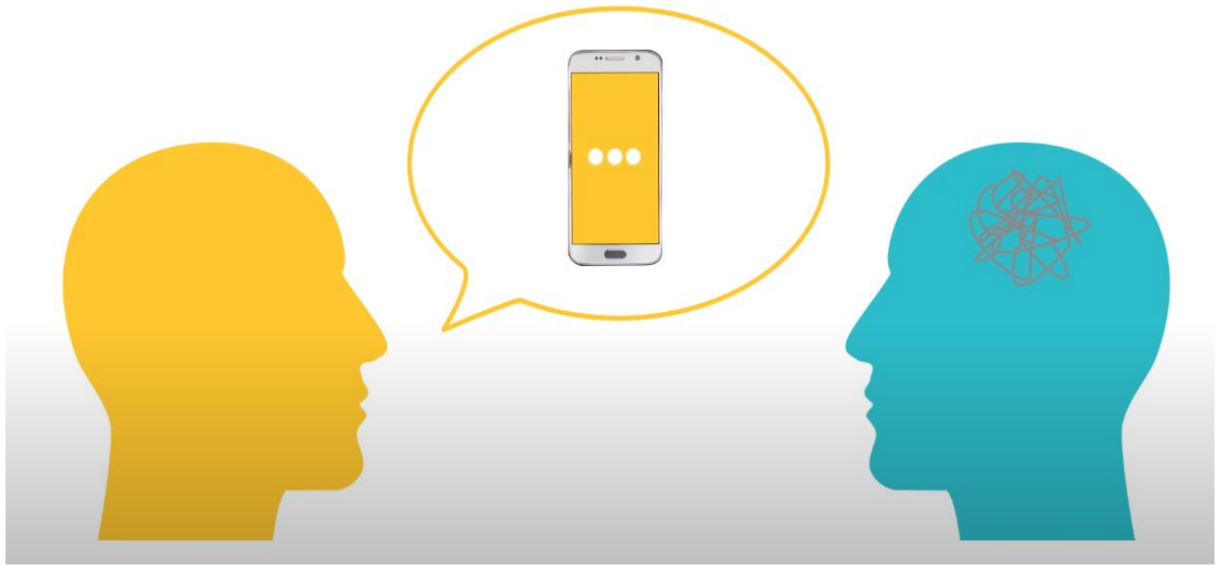
These days use case modeling is often associated with UML, although it has been introduced before UML existed. Its brief history is as follow:

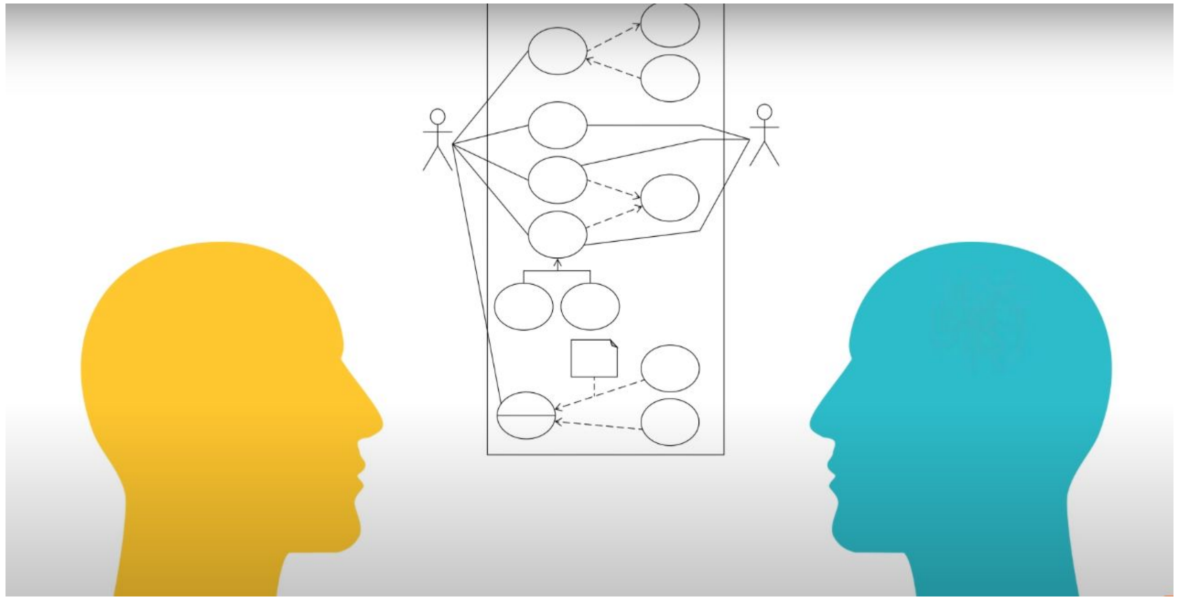
- In 1986, **Ivar Jacobson** first formulated **textual** and **visual modeling** techniques for specifying use cases.
- In 1992 his co-authored book **Object-Oriented Software Engineering - A Use Case Driven Approach** helped to popularize the technique for capturing functional requirements, especially in software development.

Use case diagram

To model a system, the most important aspect is to capture the dynamic behavior. Dynamic behavior means the behavior of the system when it is running/operating.

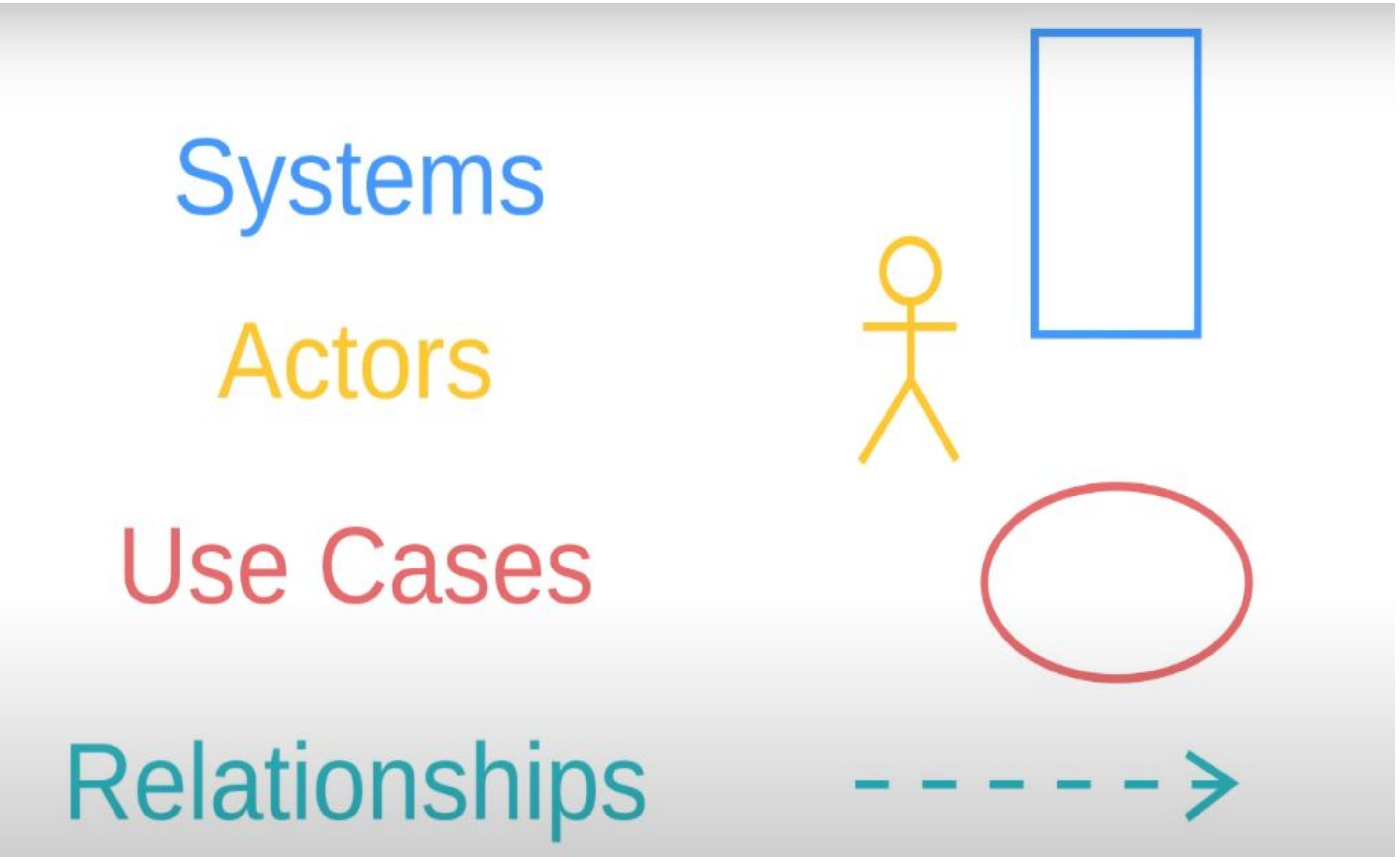
Use case diagrams are considered for high level requirement analysis of a system. When the requirements of a system are analyzed, the functionalities are captured in use cases

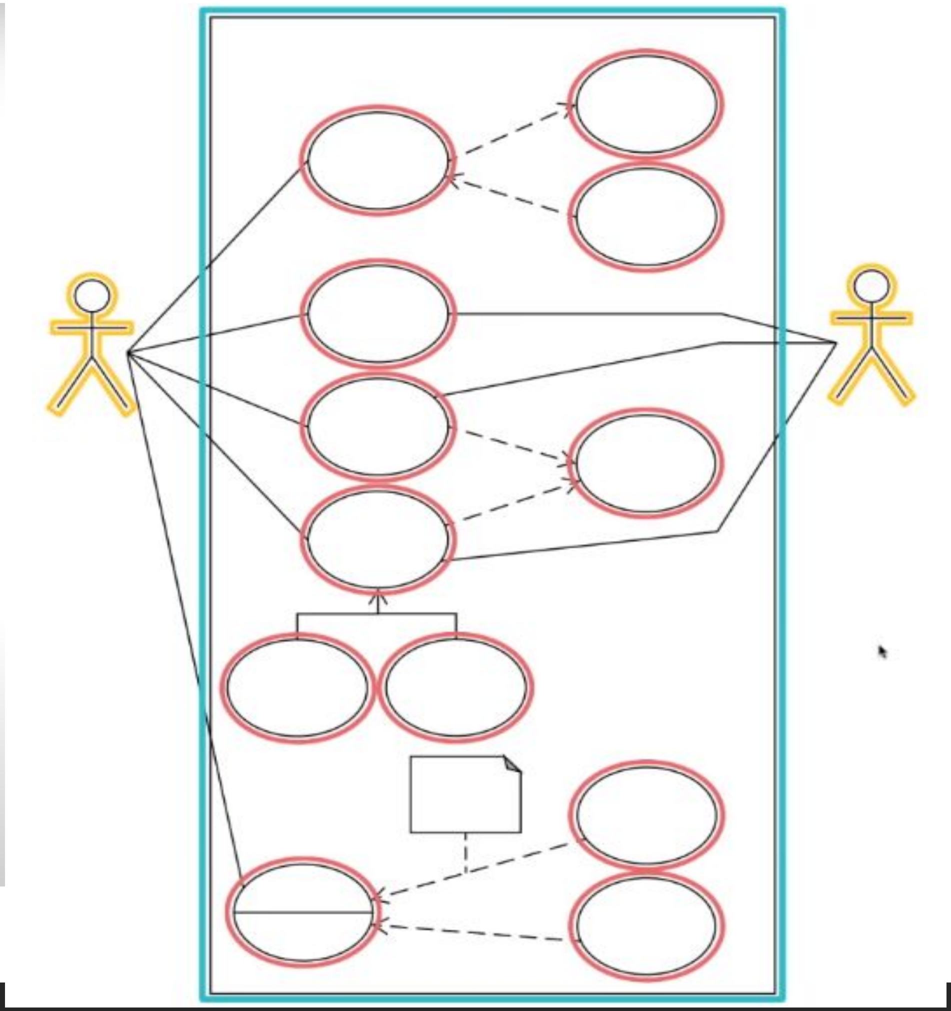




Use case Diagram

- Components of Use case diagram.





Systems

Website

Software Component

Business Process

Application

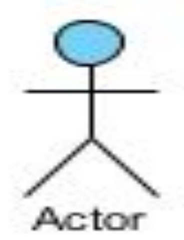
Etc.

- System is represented by a rectangle and name of system is put at the top
- Anything within this rectangle happens within the Banking App.
- Anything outside this rectangle doesn’t happen in the Banking App.

Banking App

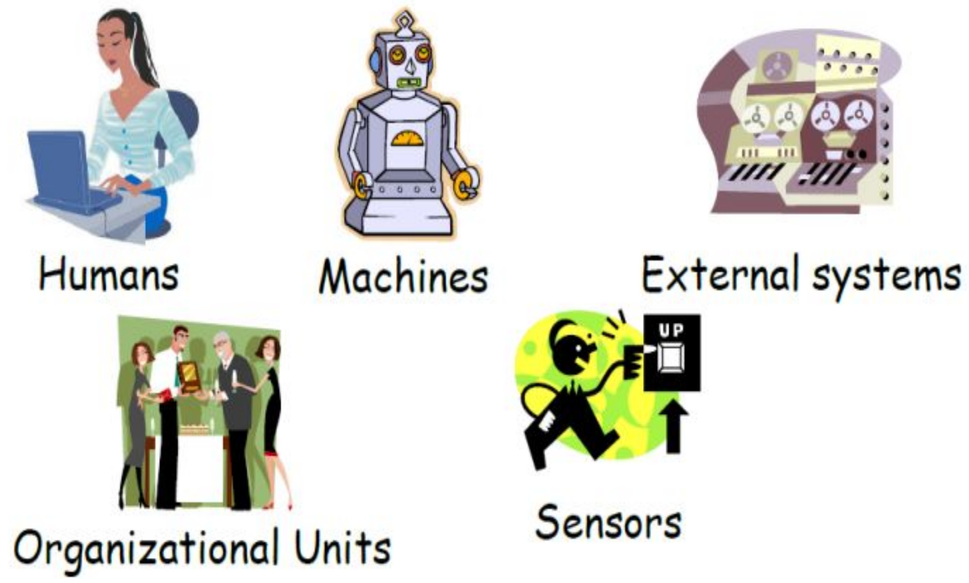
Actor

- Someone interacts with use case (system function).
- Named by noun.
- Actor plays a role in the business
- Similar to the concept of user, but a user can play different roles
- Actor triggers use case(s).
- Actor has a responsibility toward the system (inputs), and Actor has expectations from the system (outputs).
- Actors are always outside the system and interact directly with it by initiating a use case, provide input to it, and/or receive outputs from it.



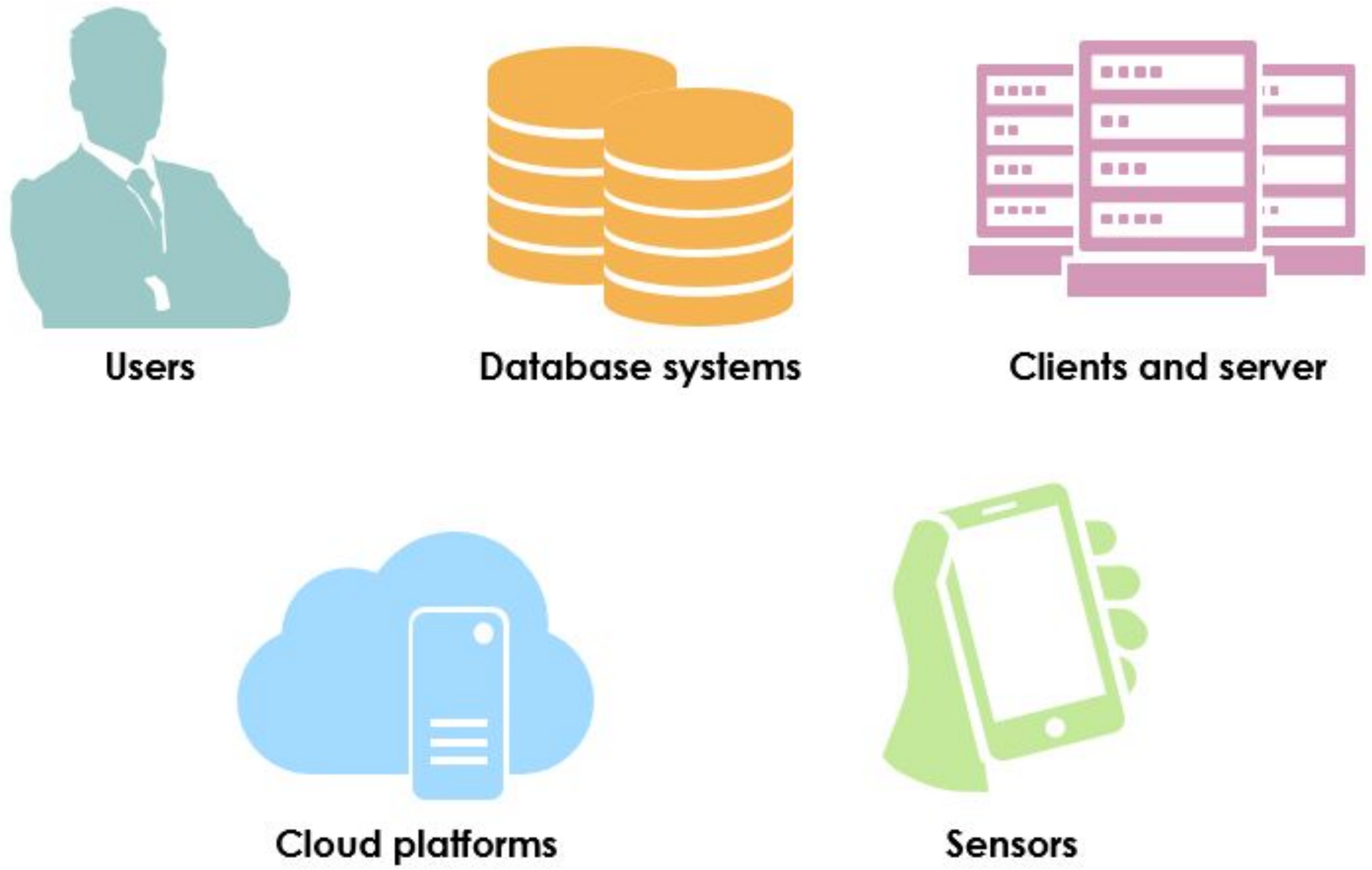
Actor

- Include all user roles that interact with the system
- Include system components only if they responsible for initiating/triggering a use case.
  - For example, a timer that triggers sending of an e-mail reminder
- ***primary*** - a user whose goals are fulfilled by the system
  - importance: define user goals
- ***supporting*** - provides a service (e.g., info) to the system
  - importance: clarify external interfaces and protocols
- ***offstage*** - has an interest in the behavior but is not primary or supporting, e.g., government
  - importance: ensure all interests (even subtle) are identified and satisfied

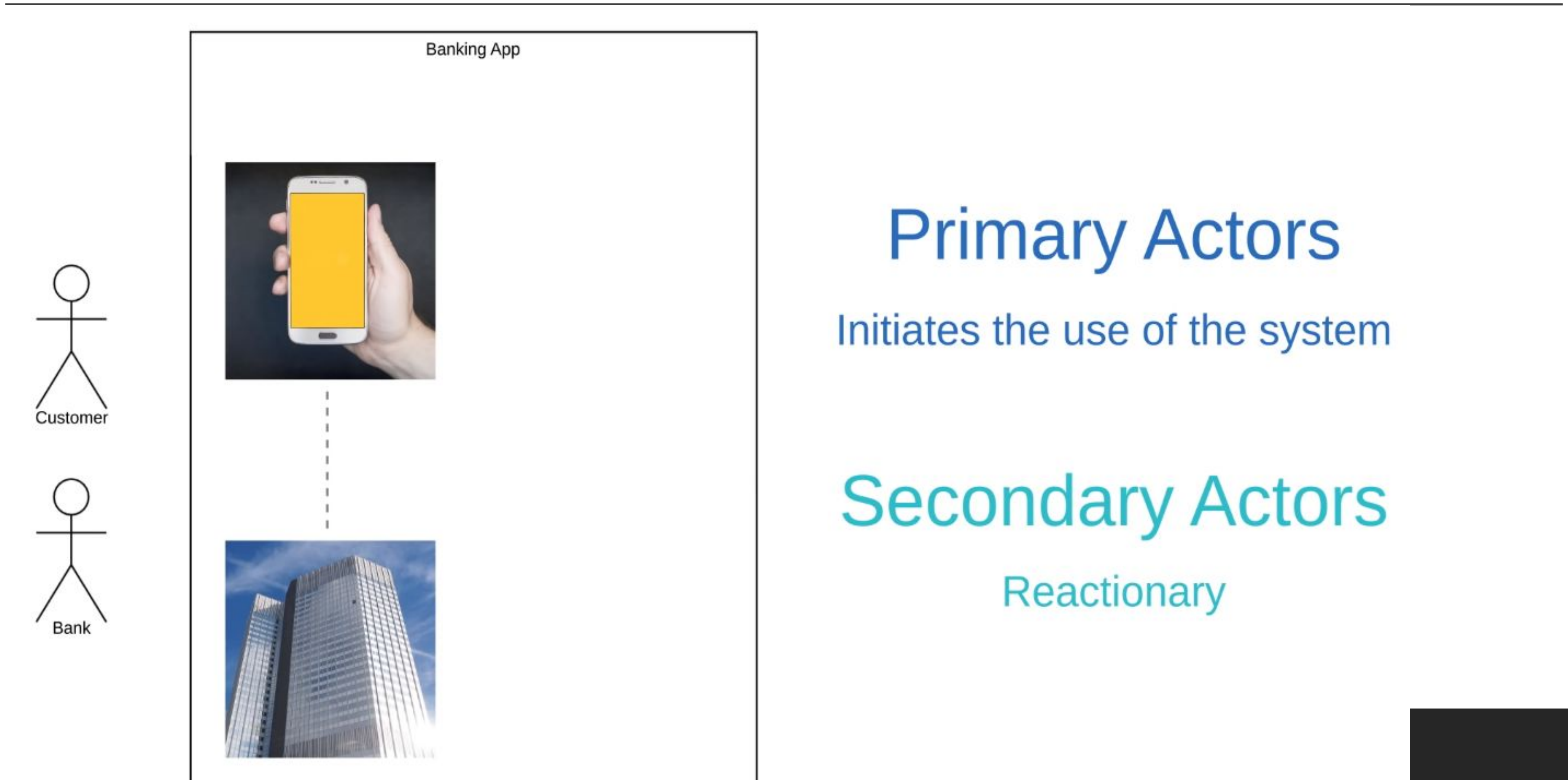


Types of actors include:

- Users
- database systems
- clients and servers
- cloud platforms
- devices



- First, it’s important to note that these actors are external objects. They always need to be placed outside of our system.
- Keep things categorical
- Right now we’re saying that both Customers and Banks are going to use our app.



Primary Actors

Initiates the use of the system

Secondary Actors

Reactionary

Identify the actors

A user clicks the search button on an application’s user interface. The application sends an SQL query to a database system. The database system responds with a result set. The application formats and displays the result set to the user.

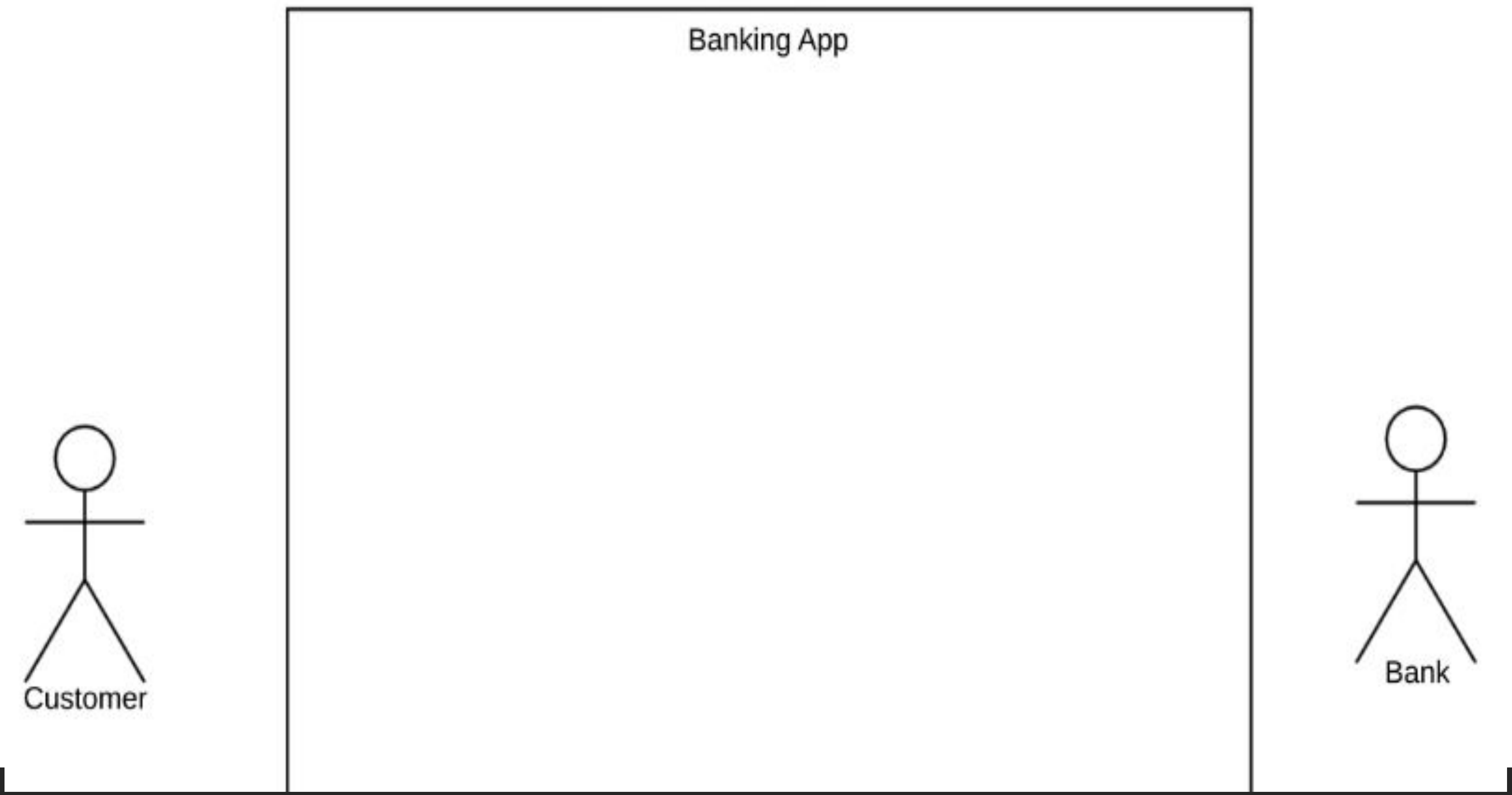
In this scenario:

- The user is a **primary acto**r because he initiates the interaction with the system (application).
- The database system is a **secondary actor** because the application initiates the interaction by sending an SQL query.

Once you have identified your actors and their goals, you have now created your initial list of high-level use cases. Remember, effective use cases must have understandable actors and goals.

Primary actors should be to the left of the system, and secondary actors should be to on the right.

This just visually reinforces the fact that Customer engages with the Banking App and then the Bank reacts

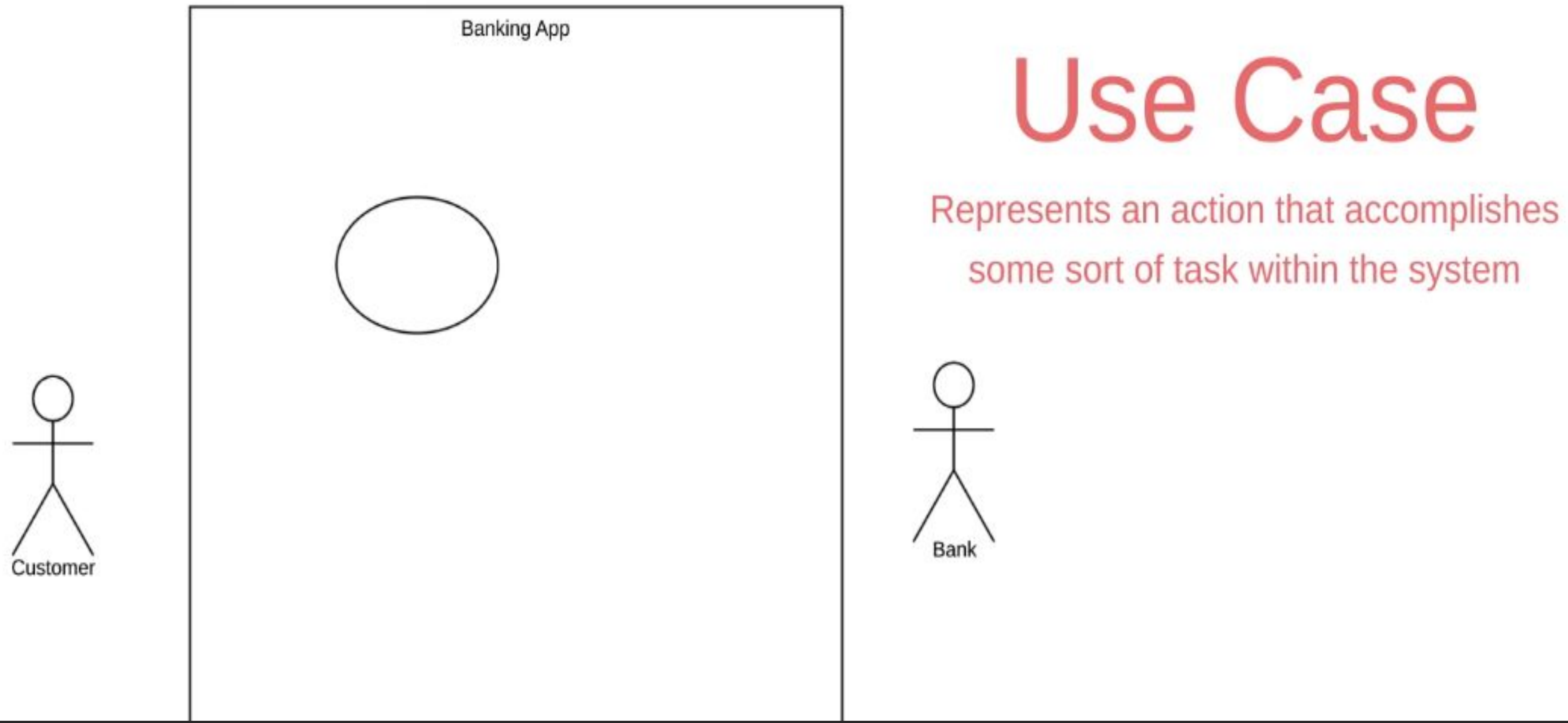


Use Case

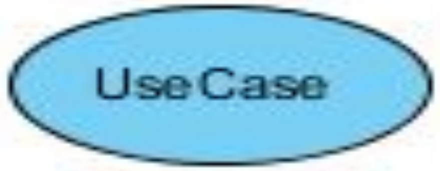
The next element is a Use Case and this is where you really start to describe what your system does.

A Use Case is depicted with this oval shape and it represents an action that accomplishes some sort of task within the system

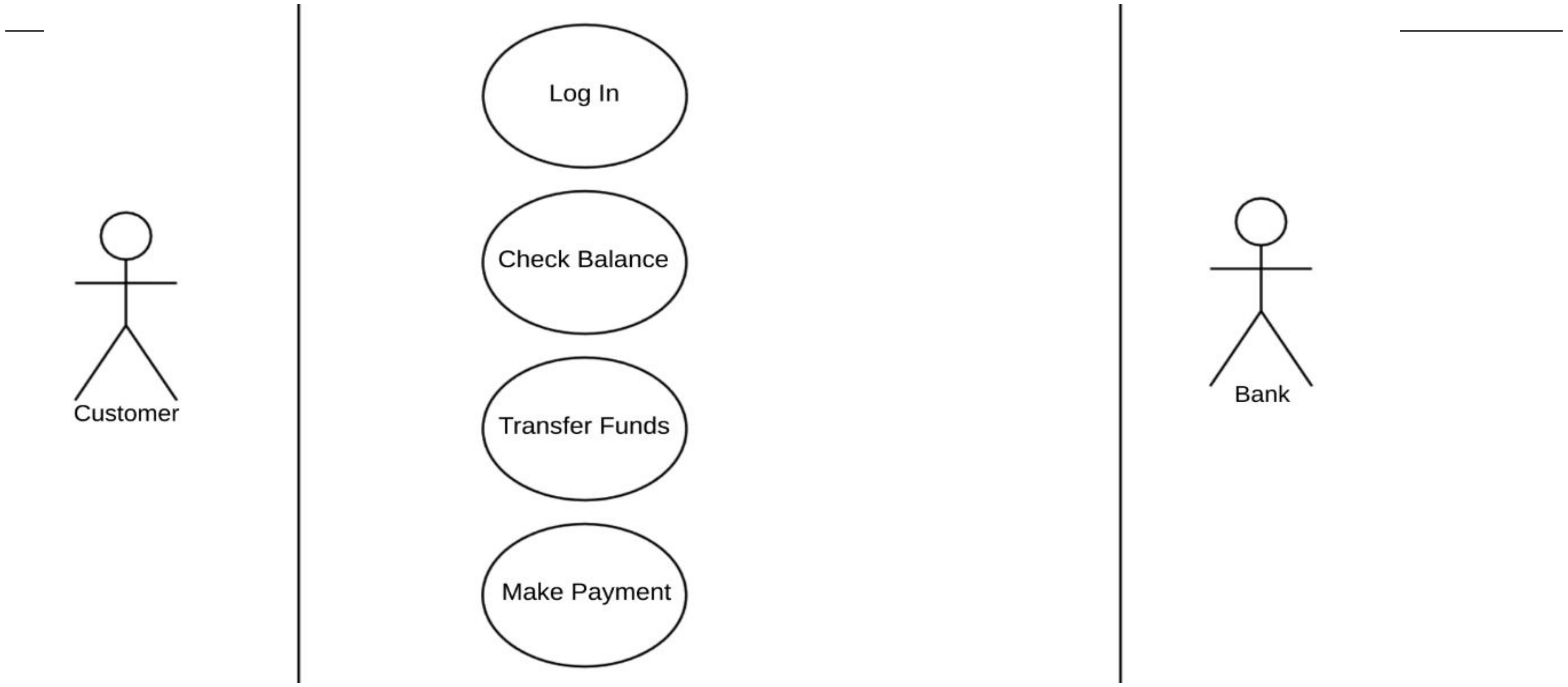
They're going to be placed within the rectangle because they're actions that occur within the Banking App.


Banking App
Use Case
Represents an action that accomplishes some sort of task within the system
Customer
Bank

- System function (process - automated or manual)
- Named by verb + Noun (or Noun Phrase).
- i.e. Do something
- Each Actor must be linked to a use case, while some use cases may not be linked to actors.



It’s good practice to put your Use Cases in a logical order when possible.

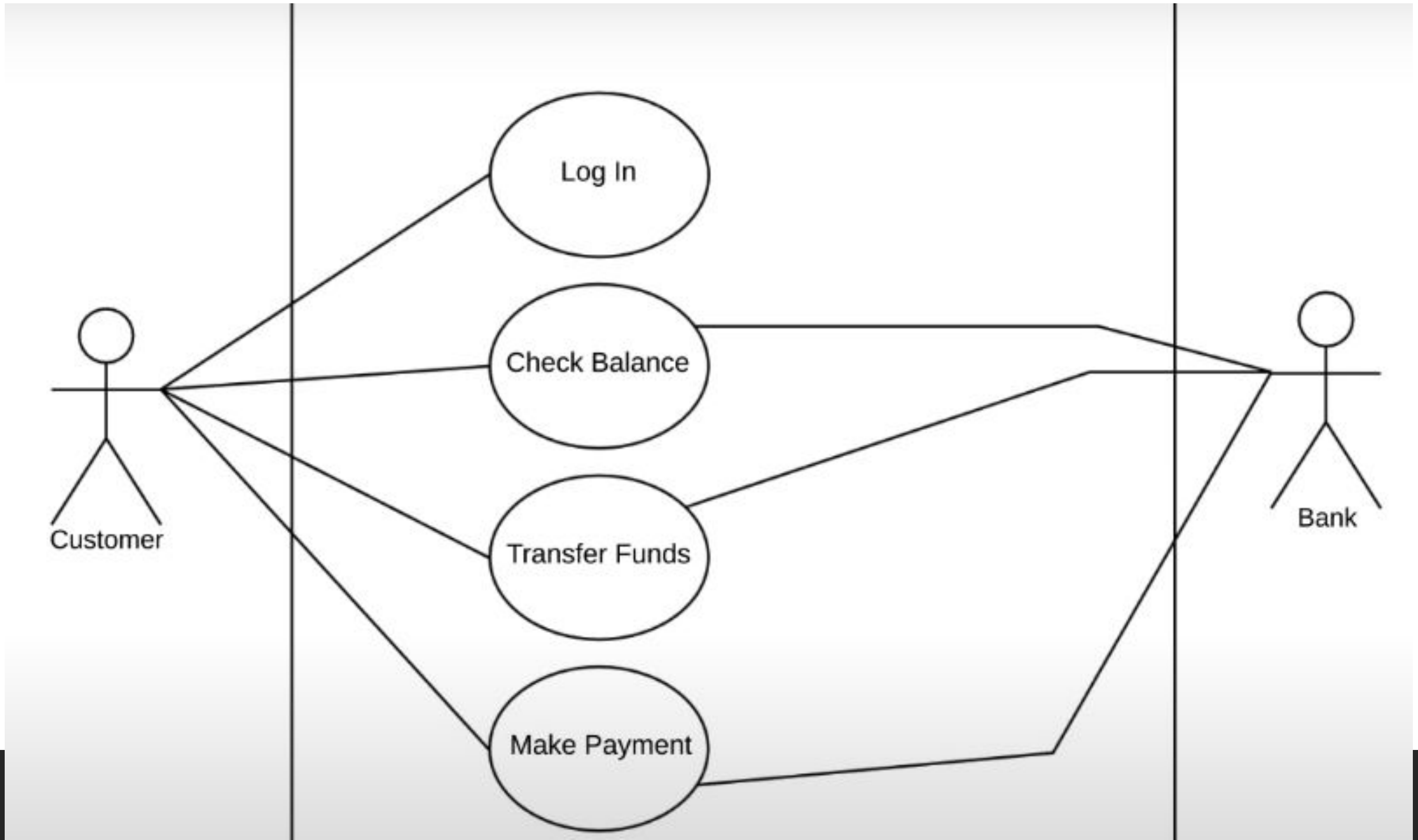


Relationship

In this example, a Customer is going to Log In to our Banking App.

So we draw a solid line between the Actor and the Use Case to show this relationship.

This type of relationship is called an association and it just signifies a basic communication or interaction



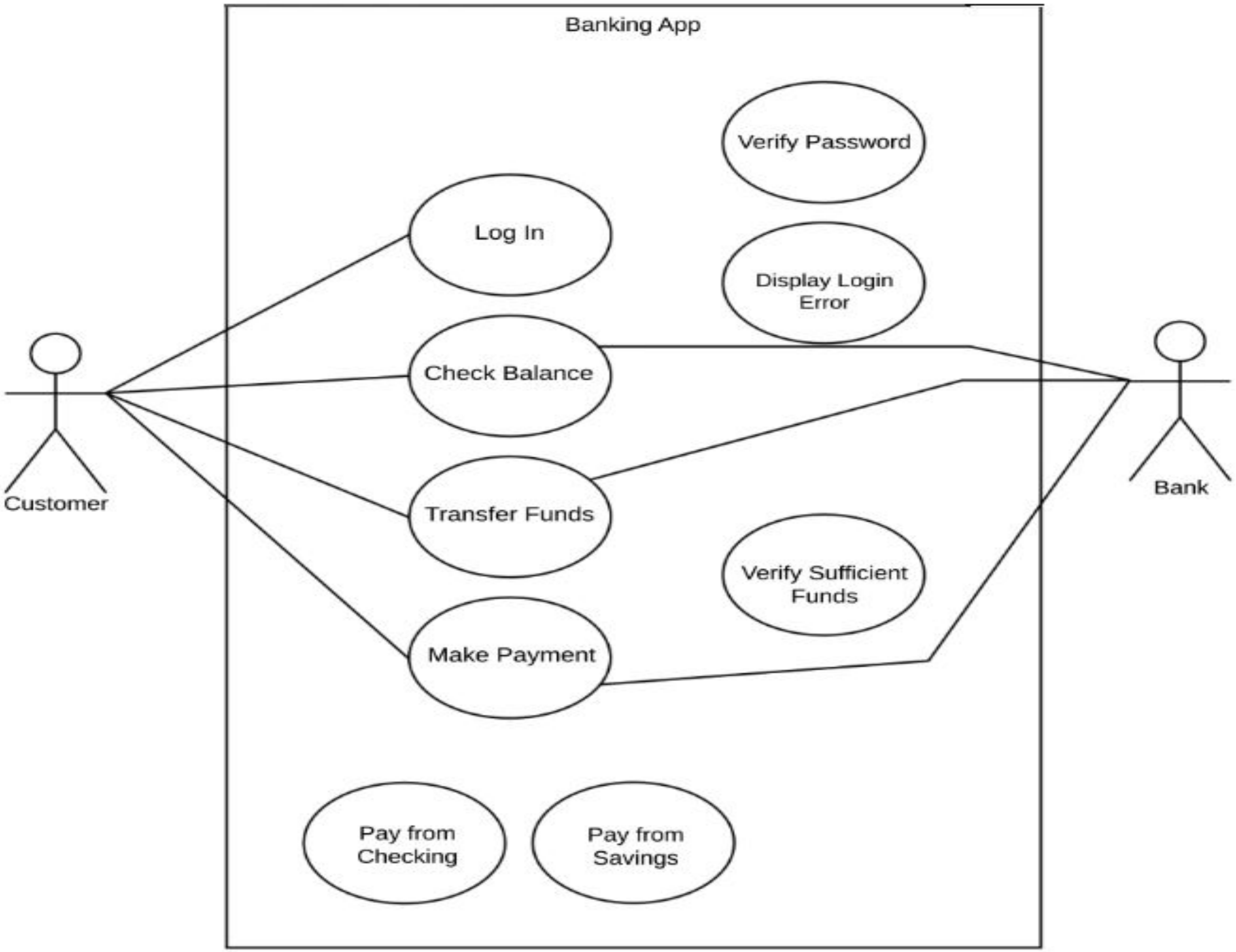
Relationships

Association

Include

Extend

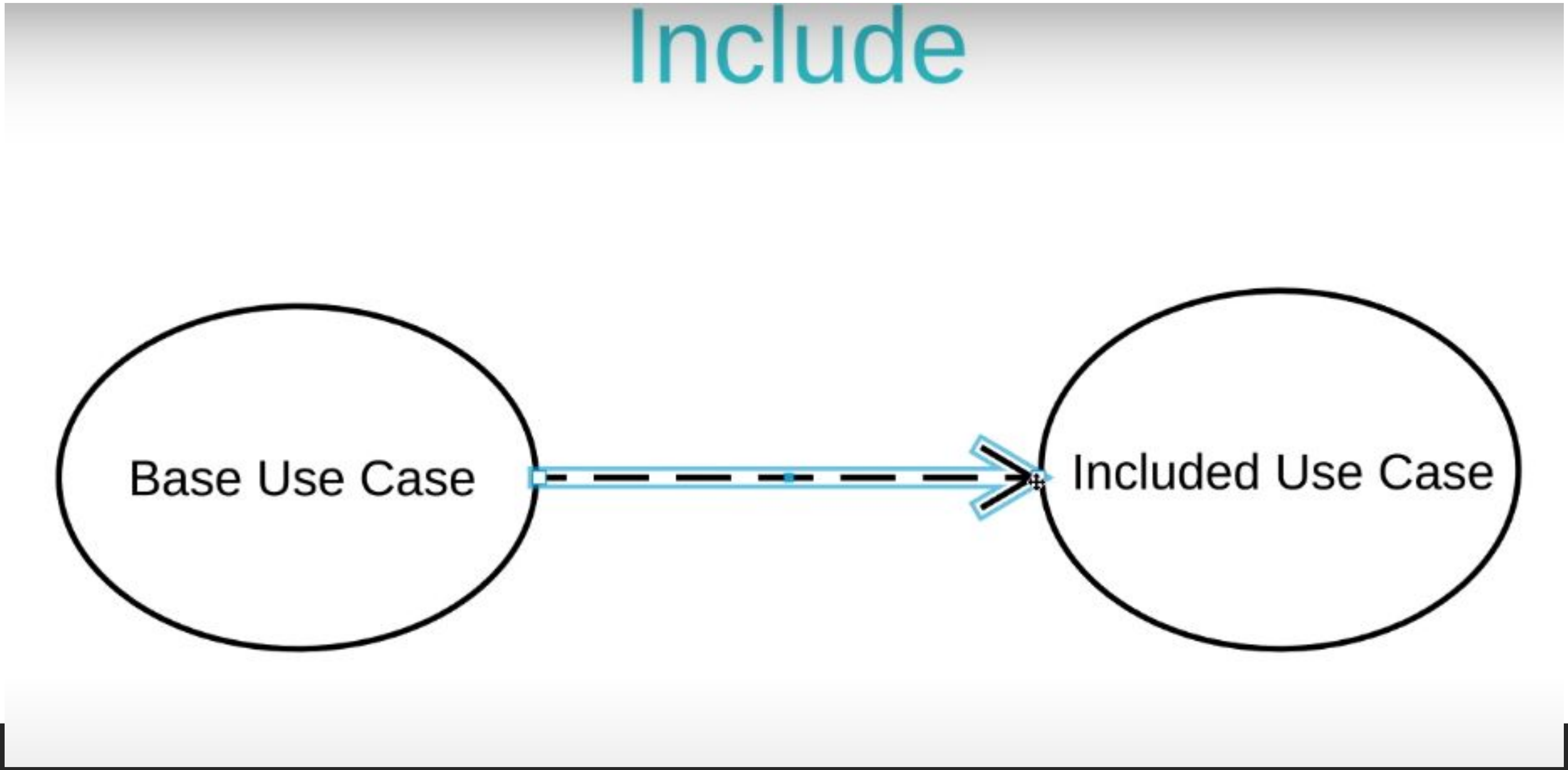
Generalization

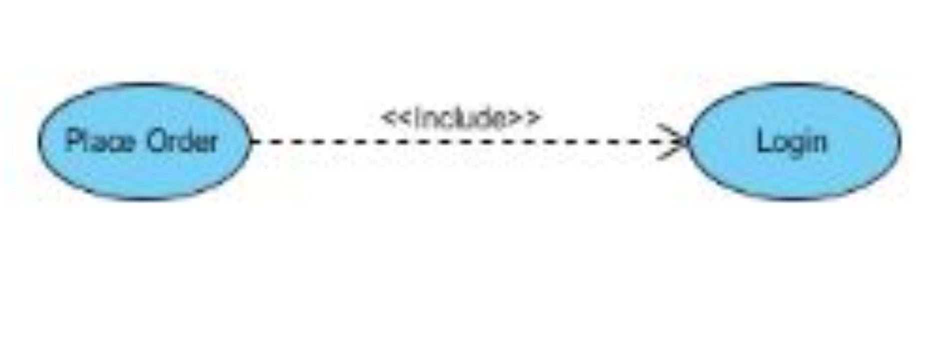

Banking App
Verify Password
Log In
Display Login Error
Check Balance
Transfer Funds
Verify Sufficient Funds
Make Payment
Pay from Checking
Pay from Savings
Customer
Bank

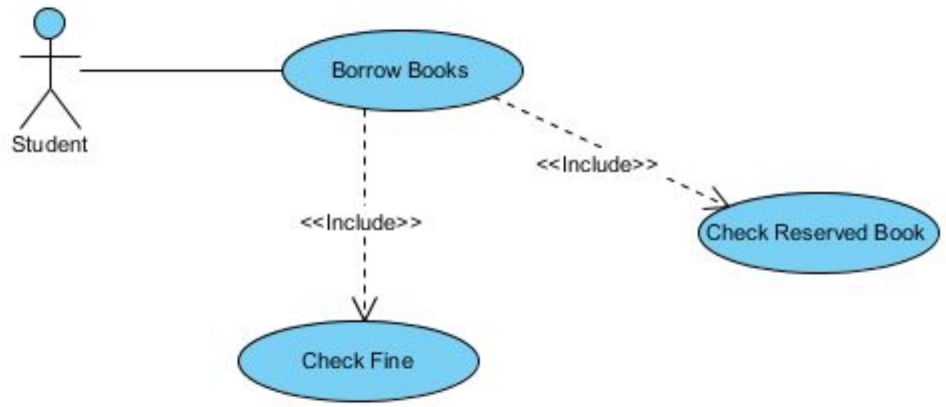
An Include relationship shows dependency between a base use case and an included use case.

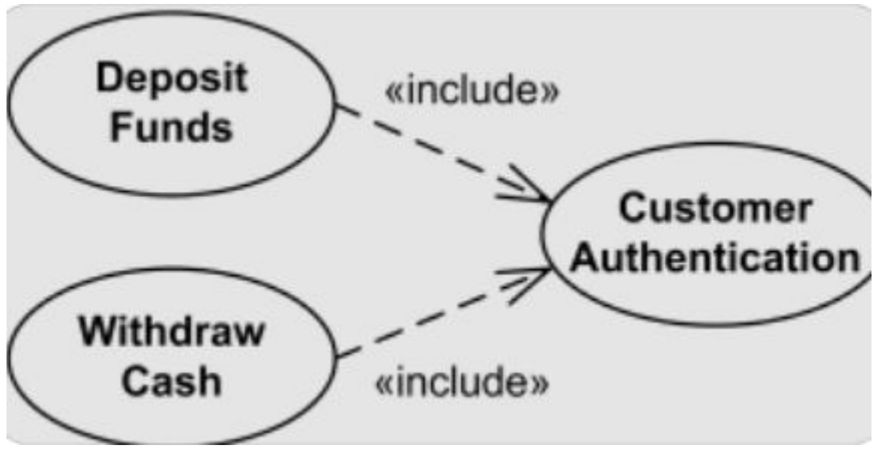
Every time the base use case is executed, the included use case is executed as well.

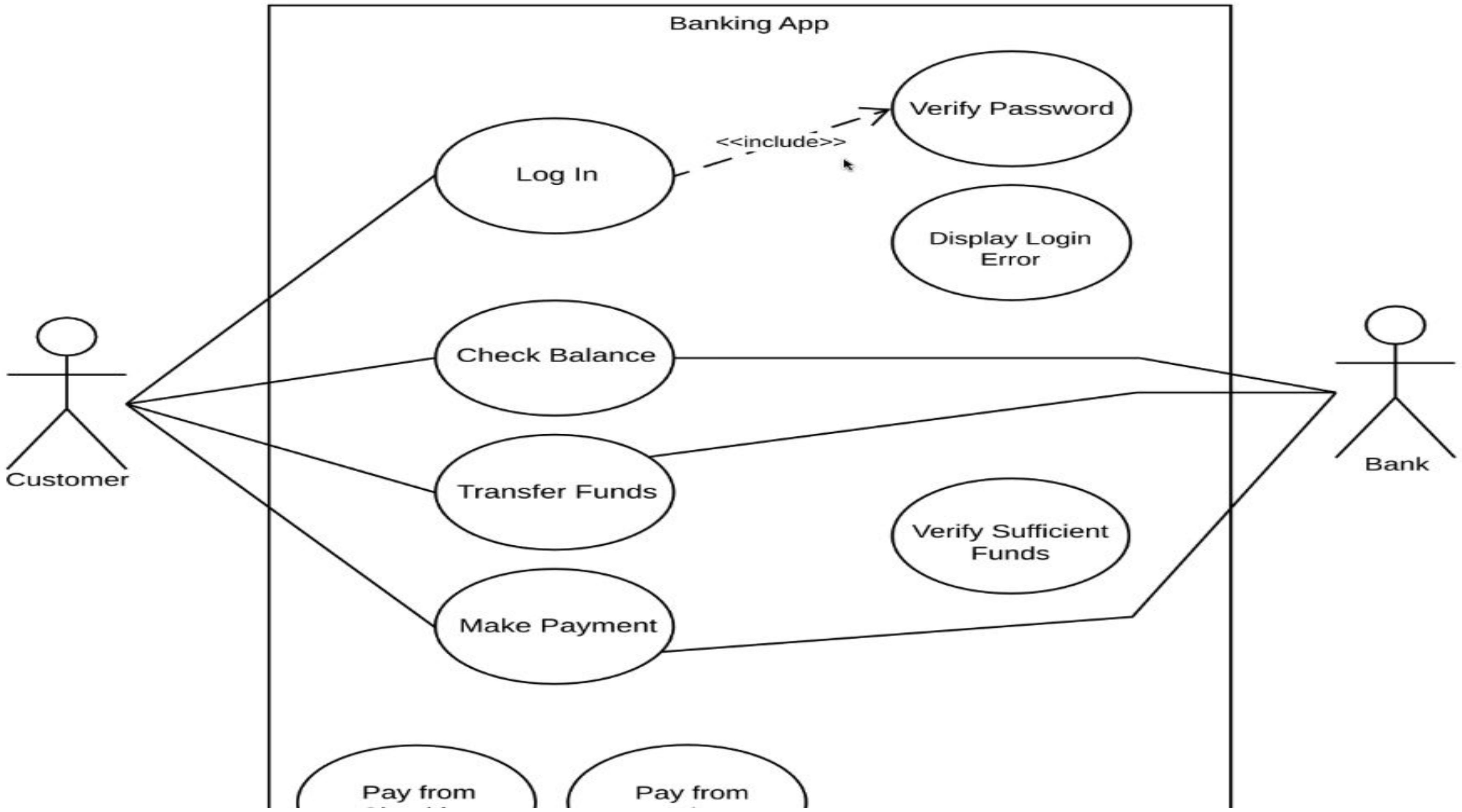
Another way to think of it is that the base use case requires an included use case in order to be complete.




Place Order
<<Include>>
Login


Student
Borrow Books
<<Include>>
<<Include>>
Check Reserved Book
Check Fine


Deposit Funds
«include»
Customer Authentication
Withdraw Cash
«include»

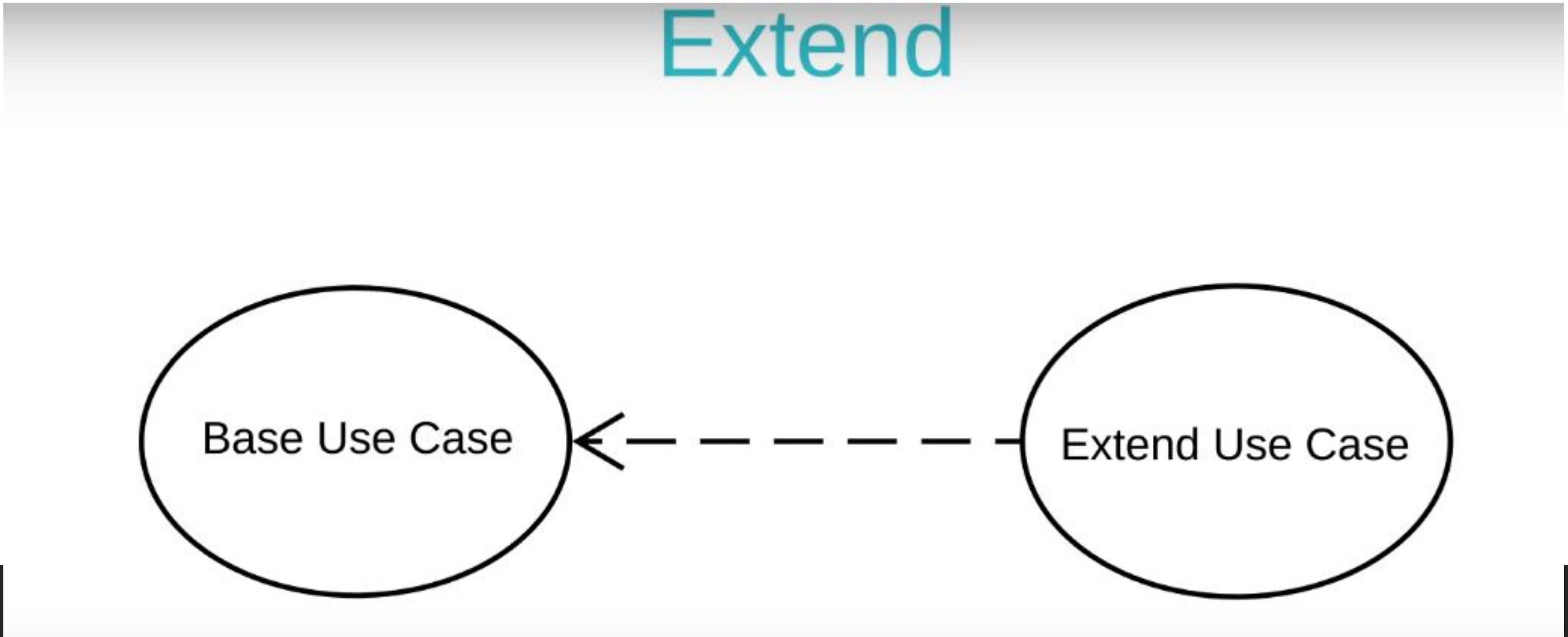

Banking App
Verify Password
<<include>>
Log In
Display Login Error
Check Balance
Customer
Bank
Transfer Funds
Verify Sufficient Funds
Make Payment
Pay from
Pay from

An extend relationship has a base use case and an extend use case.

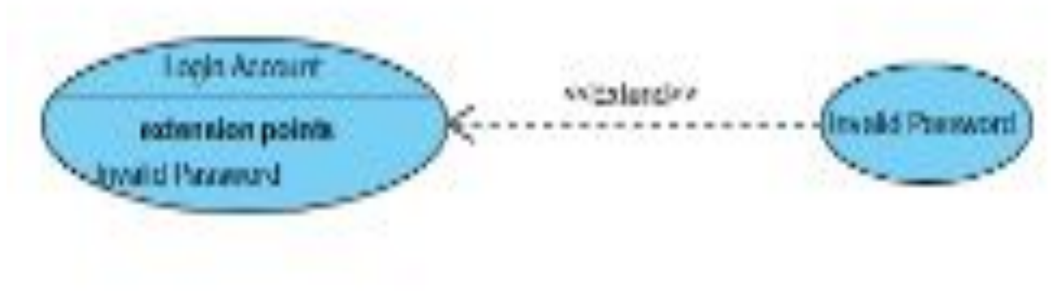
When the base use case is executed, the extend use case will happen sometimes but not every time.

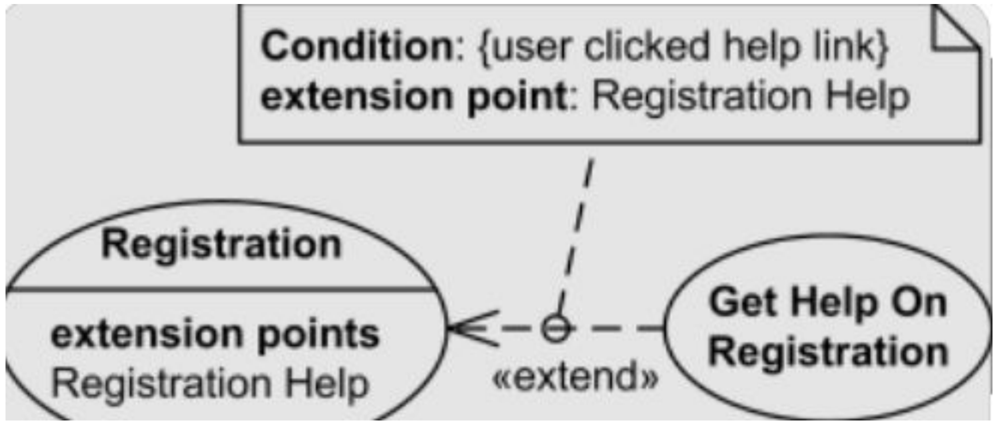
The extend relationships are important because they show optional functionality or system behavior.

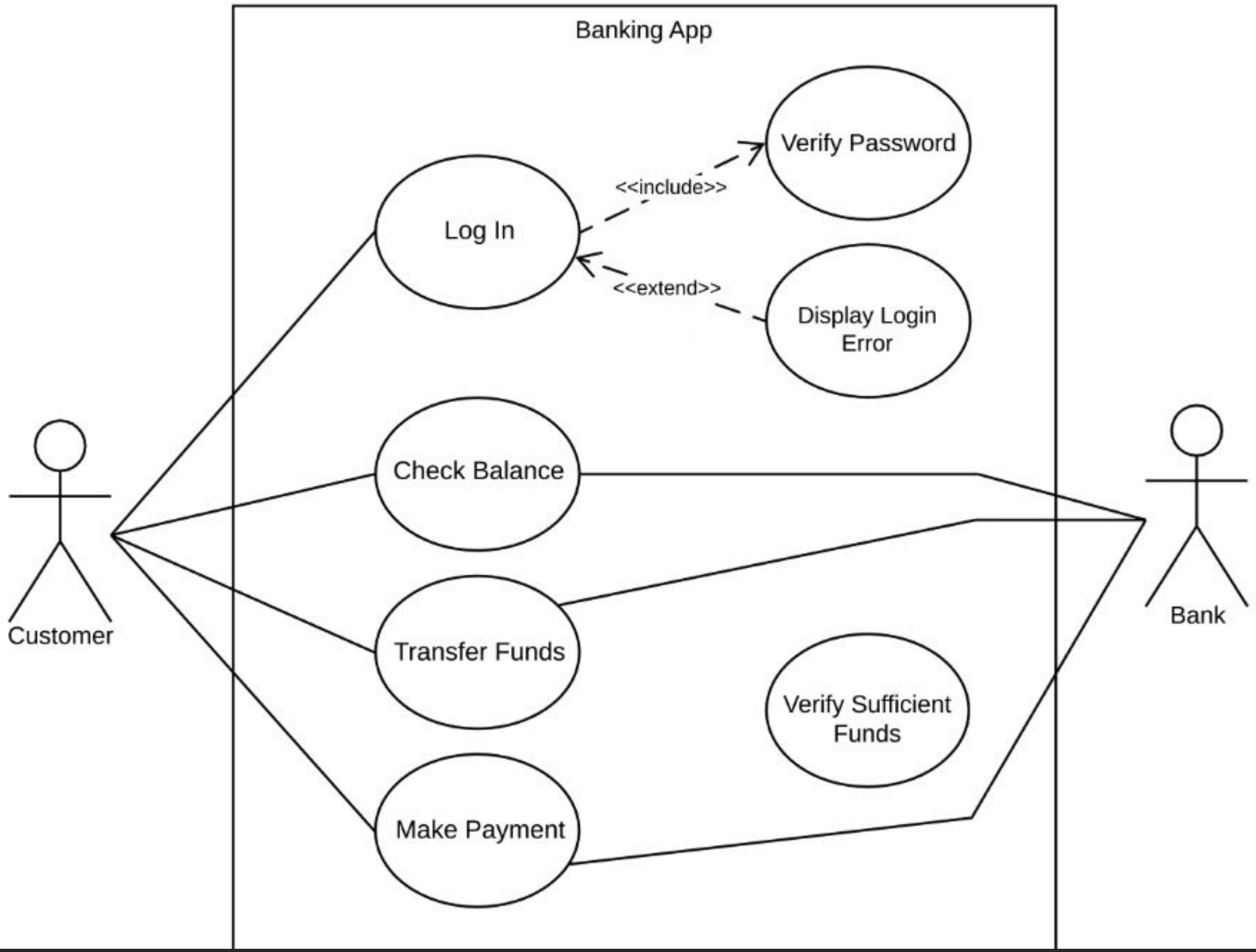
The extend use case will only happen if certain criteria are met.



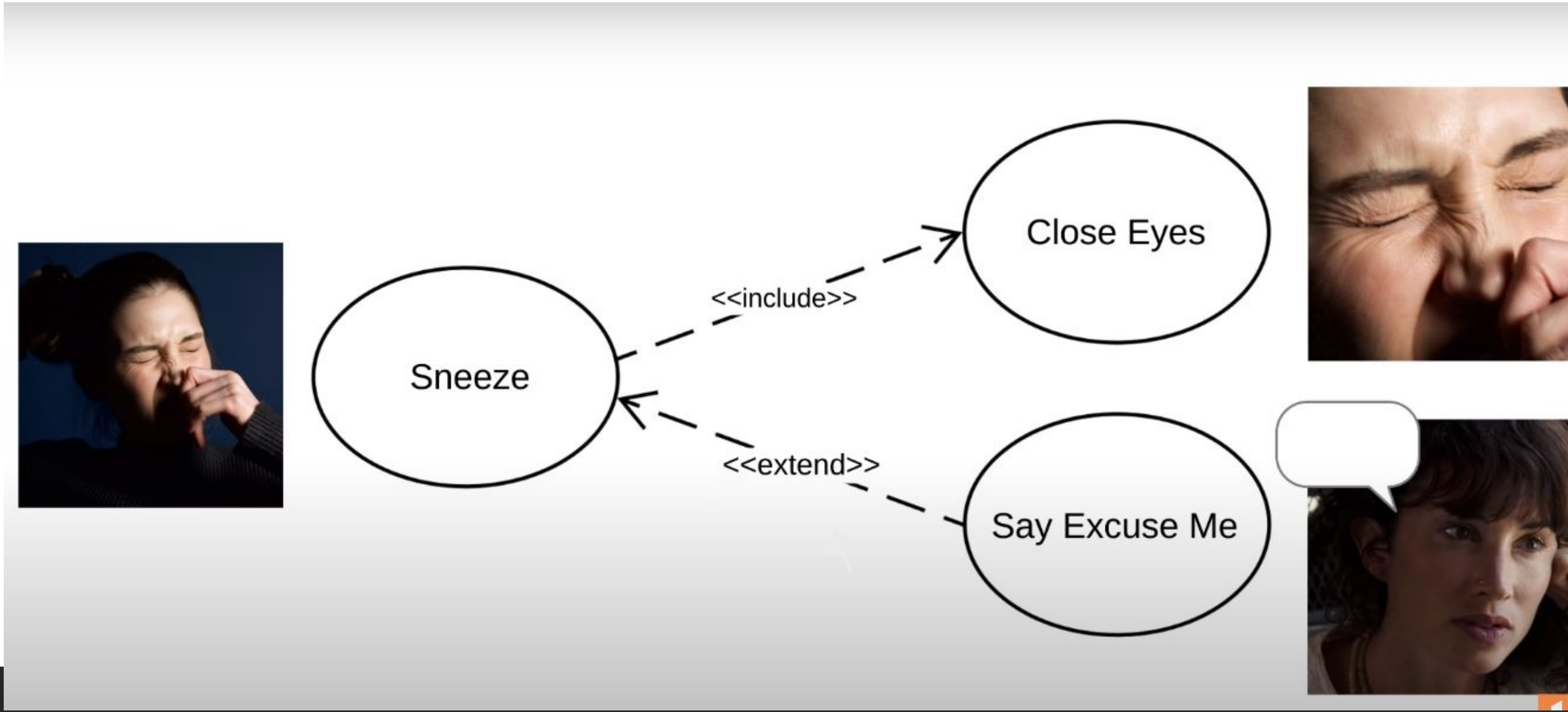
Extends adds functionality usually for abnormal case

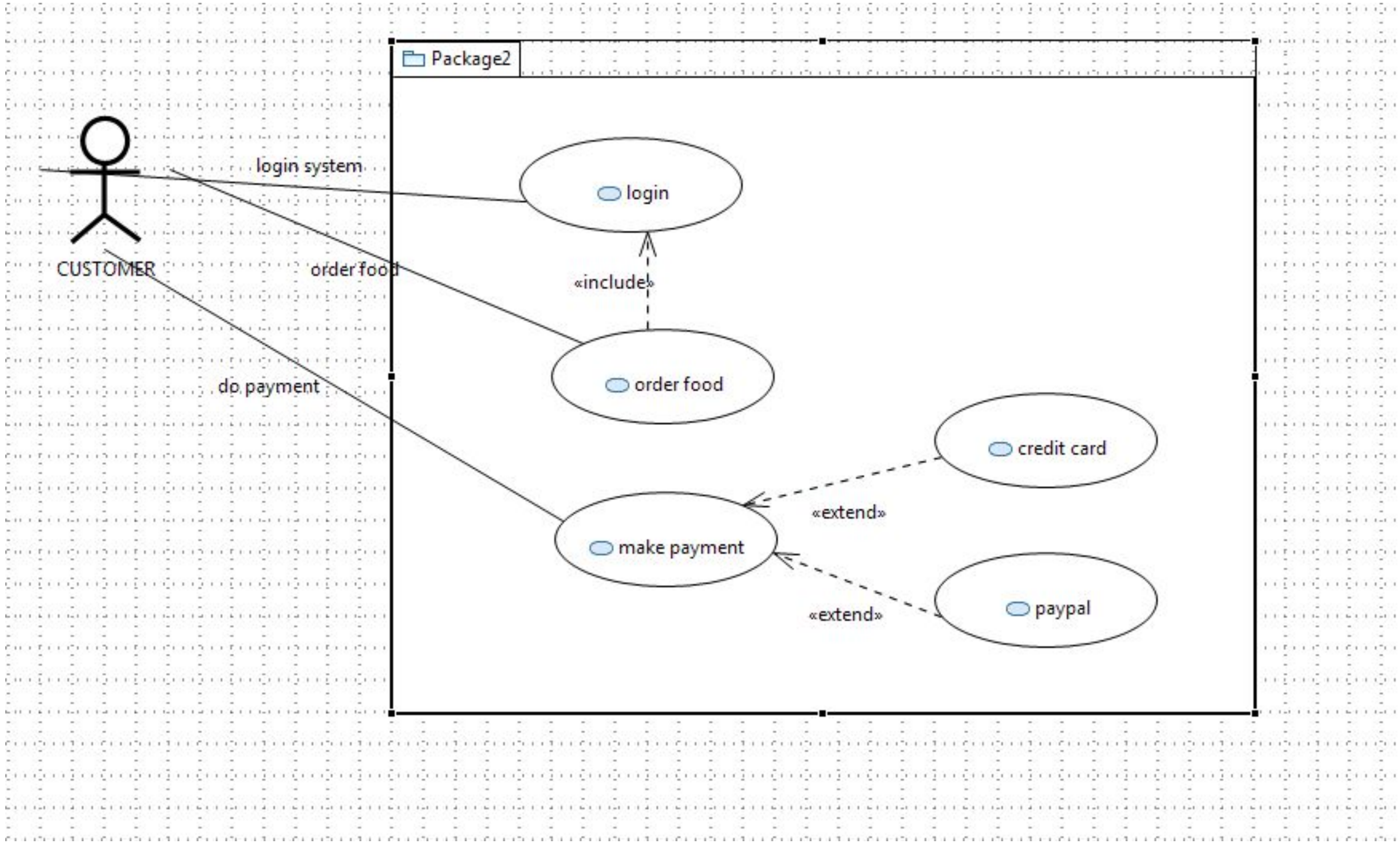


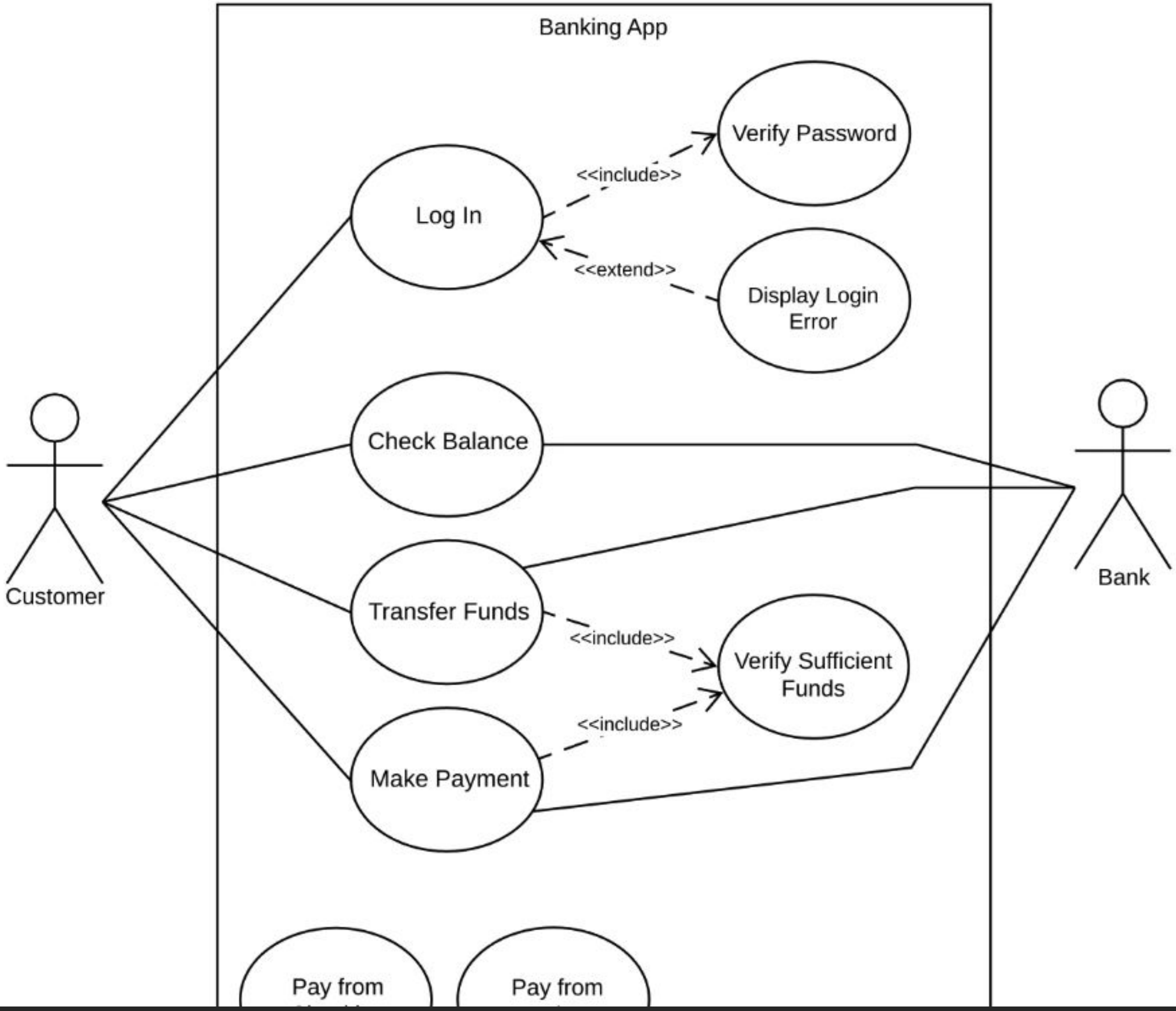



Banking App
Verify Password
<<include>>
Log In
<<extend>>
Display Login Error
Check Balance
Customer
Transfer Funds
Bank
Verify Sufficient Funds
Make Payment

Just remember that include happens every time, extend happens just sometimes,

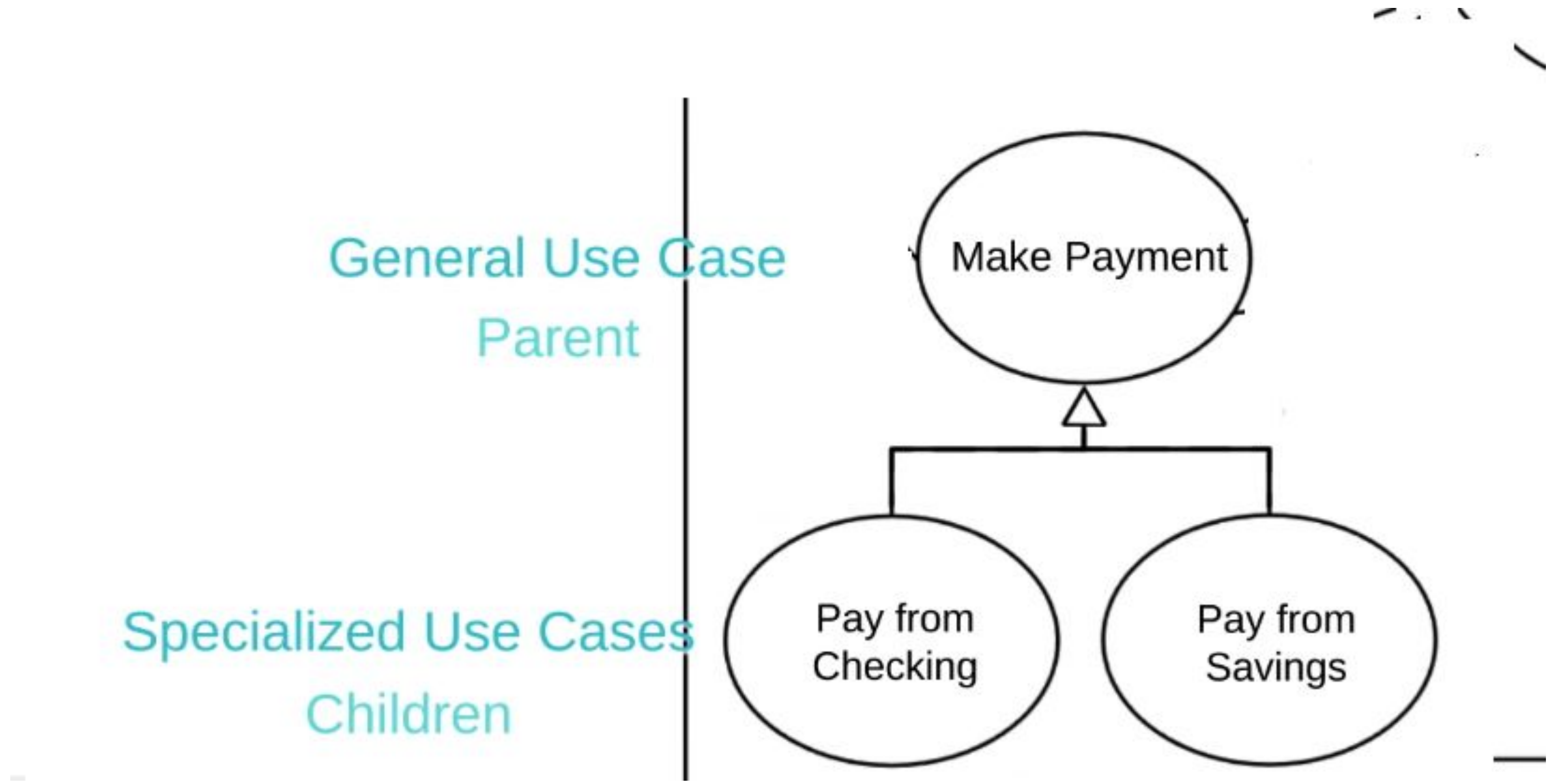



Package2
CUSTOMER
login system
order food
do payment
login
«include»
order food
credit card
«extend»
make payment
«extend»
paypal


Banking App
Verify Password
<<include>>
Log In
<<extend>>
Display Login Error
Check Balance
Customer
Bank
Transfer Funds
<<include>>
Verify Sufficient Funds
<<include>>
Make Payment
Pay from
Pay from

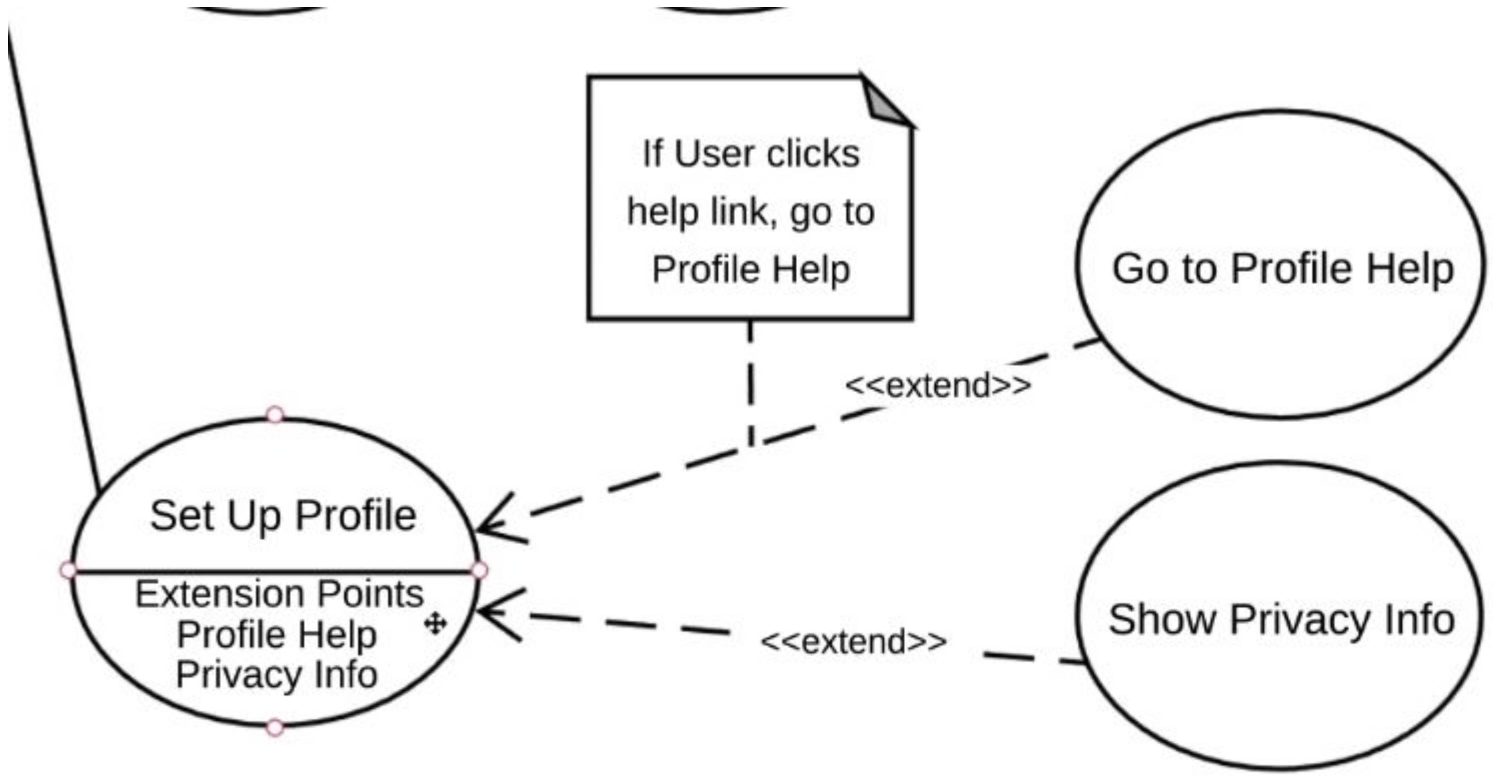
Generalization

One element (child) "is based on" another element (parent)



Extension Points

Extension points are just a detailed version of extend relationships.



Task

Draw a use case for a University website having actors student, teacher and administration. Add all types of relationships

Scenario

Draw a use case for Hospital Receptionist

Hospital Reception subsystem or module supports some of the many job duties of hospital receptionist. Receptionist schedules patient's appointments and admission to the hospital, collects information from patient upon patient's arrival and/or by phone. For the patient that will stay in the hospital ("inpatient") she or he should have a bed allotted in a ward. Receptionists might also receive patient's payments, record them in a database and provide receipts, file insurance claims and medical reports.