Structured Programming

Lecture9

Dr. Obead Alhadreti

Outline

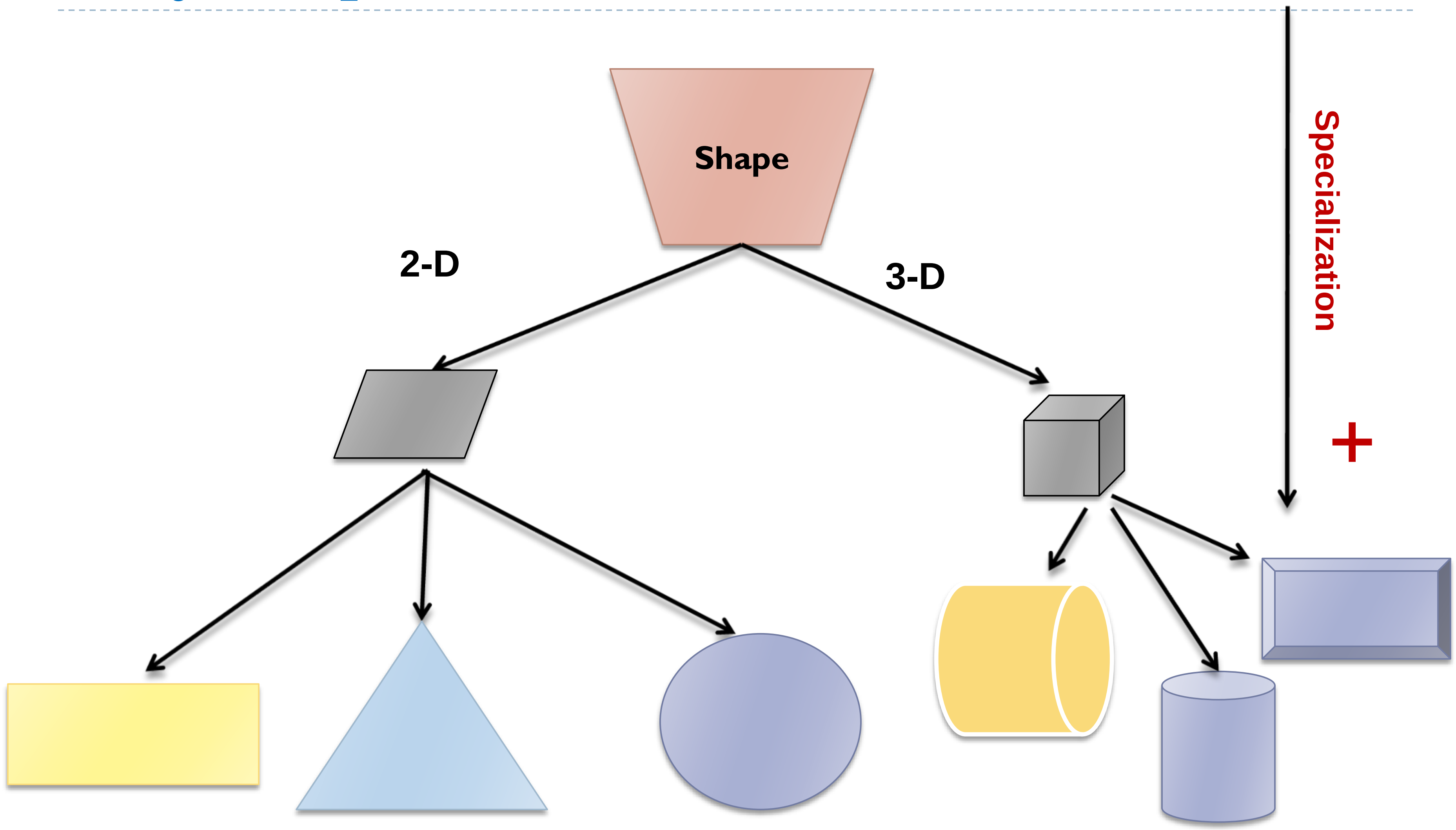
- **Polymorphism**
- **UML**

Polymorphism

Polymorphism

- Polymorphism comes from Greek meaning "many forms."
- In jave, polymorphism means the capability of a method to do different things based on the object that it is acting upon.
- Polymorphism is extensively used in implementing inheritance.
- In particular, polymorphism enables us to write programs that process objects that share the same superclass in a class hierarchy as if they are all objects of the superclass.

Polymorphism



Polymorphism in java

- Different techniques to achieve polymorphism in java.
  1. Method Overloading
  2. Method Overriding
  3. Abstract class
  4. Interfaces

Types of polymorphism in java

- There are two types of polymorphism in java:
  1) Compile time polymorphism (static polymorphism).

  2) Runtime polymorphism (Dynamic polymorphism).

  - In simple terms, static binding means when the type of object which is invoking the method is determined at compile time by the compiler (For example, method overloading). While Dynamic binding means when the type of object which is invoking the method is determined at run time by the compiler (For example, method overriding).

UML

UML

- The Unified Modeling Language (UML) is a way of visualizing a software program using a collection of diagrams.
- UML can be applied to diverse application domains (e.g., banking, finance, internet, aerospace, healthcare, etc.)
- UML provides a pictorial or graphical notation for documenting the artefacts such as classes, objects and packages that make up an object-oriented system. UML diagrams can be divided into three categories:
  1. Structural diagrams
  2. Behavioual diagrams

Structural diagrams

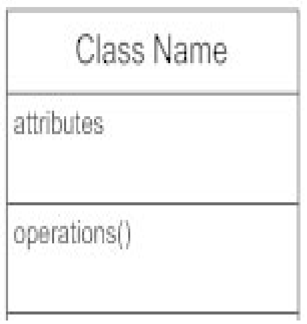
- **Structure diagrams** show the things in the modeled system. In a more technical term, they show different objects in a system. **Behavioral diagrams** show what should happen in a system. They describe how the objects interact with each other to create a functioning system.

- The four structural diagrams are:

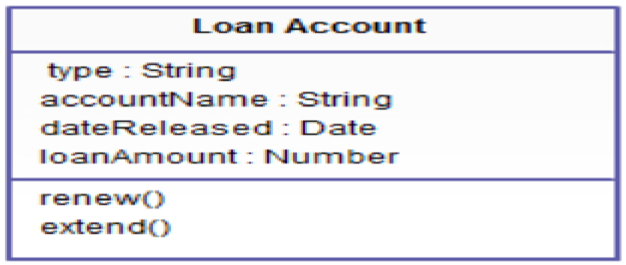
1. Class diagram
2. Object diagram
3. Component diagram
4. Deployment diagram

Class diagram

- In most modeling tools, a class has three parts. Name at the top, attributes in the middle and operations or methods at the bottom. In a large system with many related classes, classes are grouped together to create class diagrams. It shows the classes in a system, attributes, and operations of each class and the relationship between each class. Different relationships between classes are shown by different types of arrows.



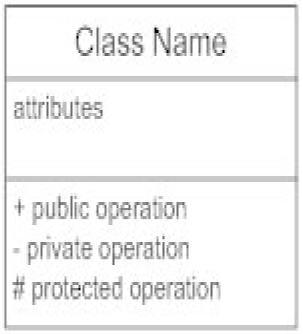
Class



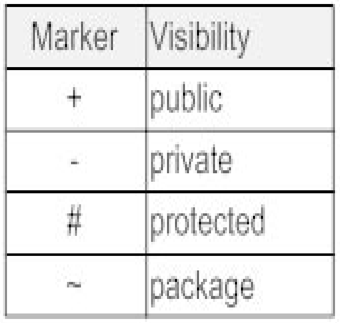
Simple class diagram with attributes and operations

Class diagram

- **Visibility:** Use visibility markers to signify who can access the information contained within a class. Private visibility, denoted with a - sign, hides information from anything outside the class partition. Public visibility, denoted with a + sign, allows all other classes to view the marked information. Protected visibility, denoted with a # sign, allows child classes to access information they inherited from a parent class.



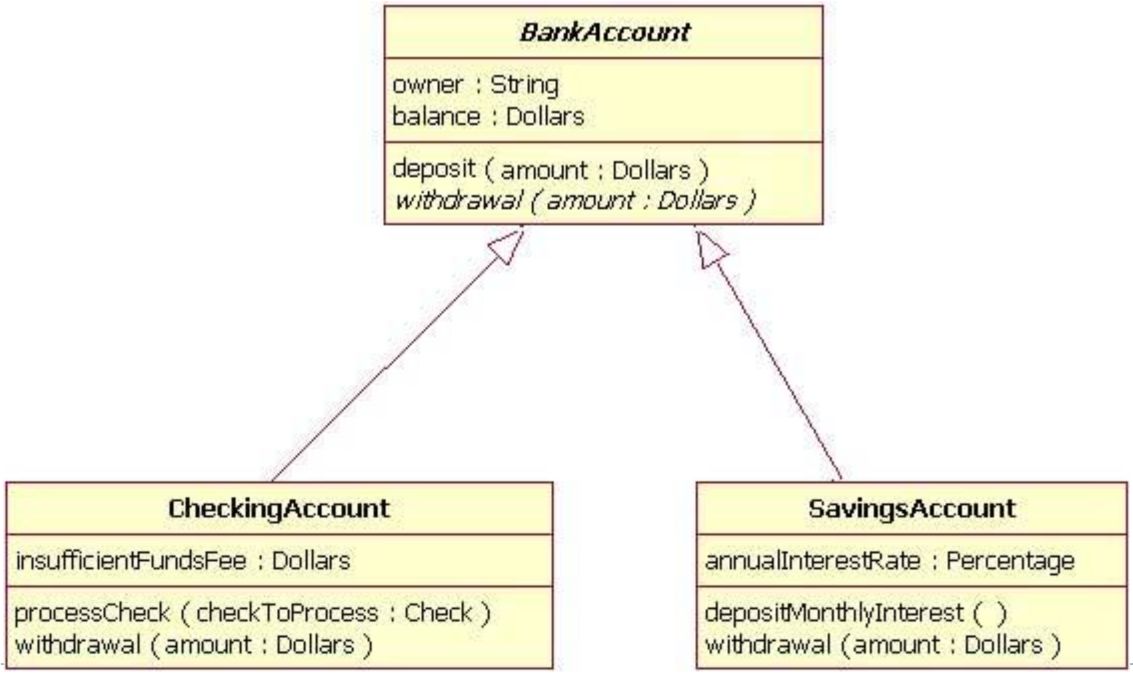
Visibility



Marker	Visibility
+	public
-	private
#	protected
~	package

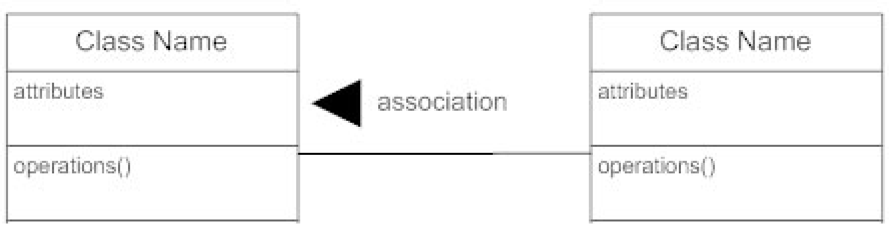
Class diagram

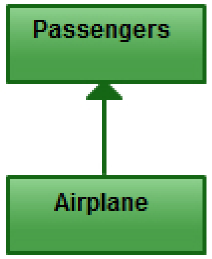
- **Generalization/inheritance:** Generalization is another name for inheritance or an "is a" relationship. It refers to a relationship between two classes where one class is a specialized version of another.


BankAccount
owner : String
balance : Dollars
deposit (amount : Dollars)
withdrawal (amount : Dollars)
CheckingAccount
insufficientFundsFee : Dollars
processCheck (checkToProcess : Check)
withdrawal (amount : Dollars)
SavingsAccount
annualInterestRate : Percentage
depositMonthlyInterest ( )
withdrawal (amount : Dollars)

Class diagram

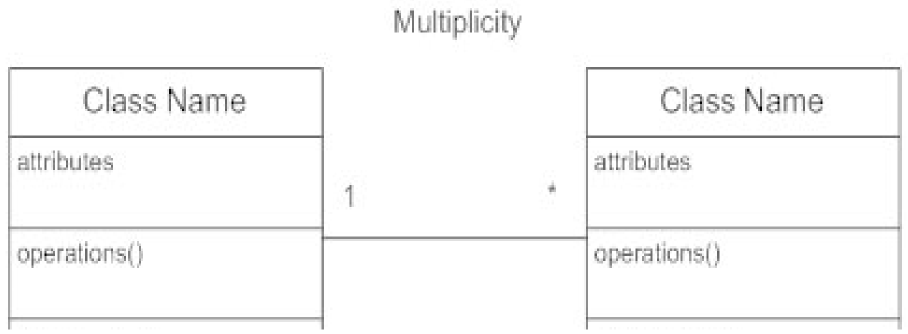
- **Associations:** Associations represent static relationships between classes. Place association names above, on, or below the association line. Use a filled arrow to indicate the direction of the relationship. Place roles near the end of an association. Roles represent the way the two classes see each other. For example, passenger and airline may be linked

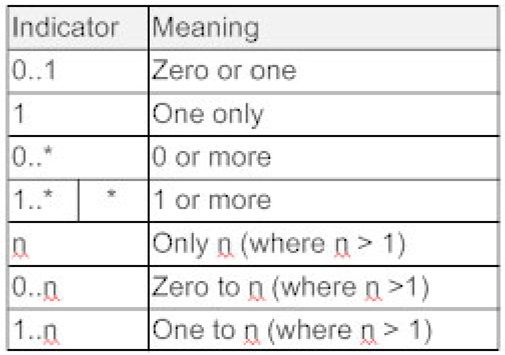




Class diagram

- **Multiplicity (Cardinality):** Place multiplicity notations near the ends of an association. These symbols indicate the number of instances of one class linked to one instance of the other class.

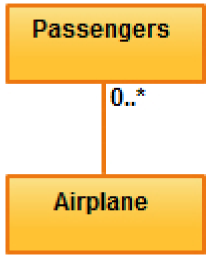




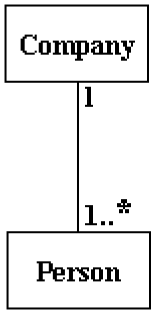
Indicator	Meaning
0..1	Zero or one
1	One only
0..*	0 or more
1..* | *	1 or more
n	Only n (where n > 1)
0..n	Zero to n (where n >1)
1..n	One to n (where n > 1)

Class diagram

- **For example,** one commercial airplane may contain zero to many passengers. The notation 0..* in the diagram means "zero to many".



- **For example,** one company will have one or more employees, but each employee works for one company only.



Behavioral diagrams

- **Behavioral diagrams** show what should happen in a system. They describe how the objects interact with each other to create a functioning system.

- The five structural diagrams are:

1. Use case diagram
2. Sequence diagram
3. Collaboration diagram
4. Statechart diagram
5. Activity diagram

Use Cases

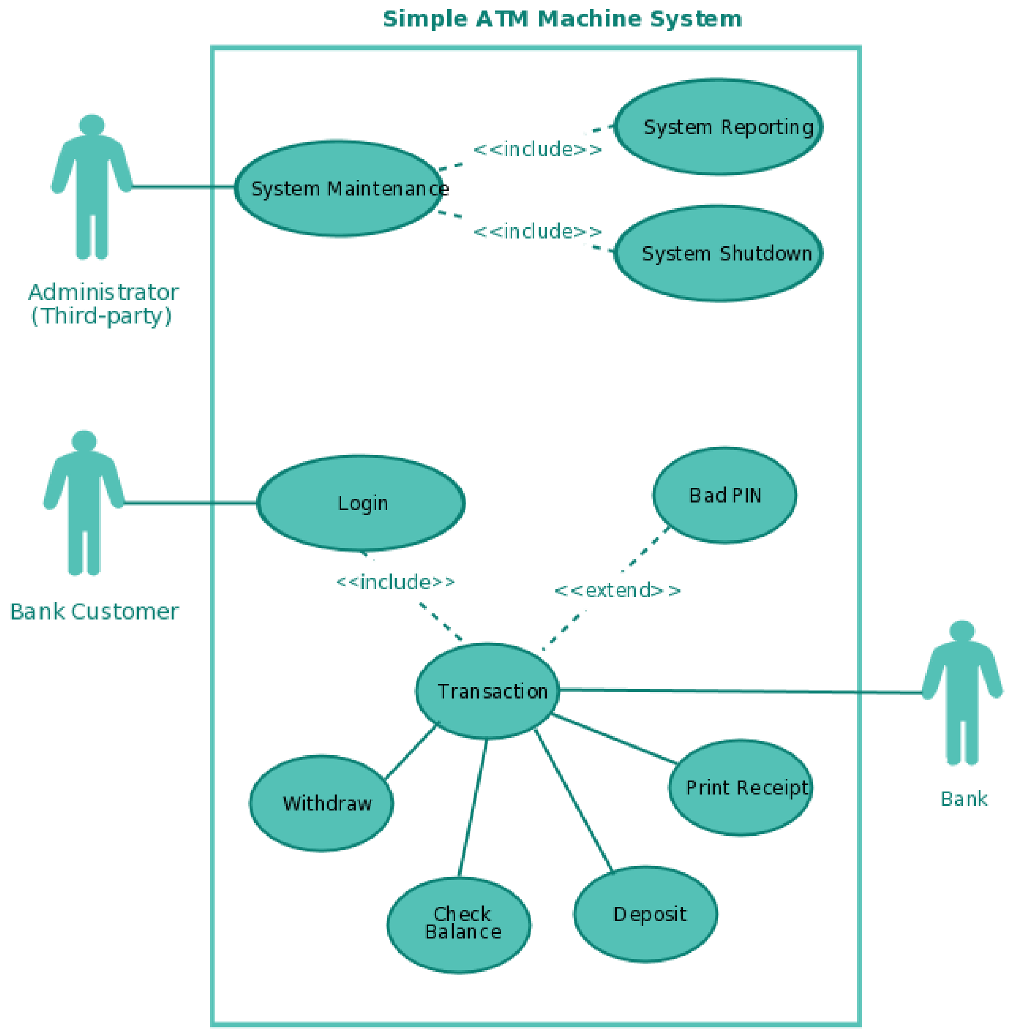
- As the most known diagram type of the behavioral UML diagrams, Use case diagrams give a graphic overview of the actors involved in a system, different functions needed by those actors and how these different functions interact.
- Focus on user-system interaction
- Give an “external view” of the system

Use Cases

- **Use Case Diagrams have 4 major elements:**

The **actors** that the system you are describing interacts with, the **system** boundary (the system itself), the **use cases**, or function, that the system knows how to perform, and the lines that represent **relationships** between these elements.

Use Case Diagrams (UCD)



Use Case Diagrams (UCD)

- **Actors**
  - **Give meaningful relevant names for actors** – For example if your use case interacts with an outside organization its much better to name it with the function rather than the organization name. (Eg: Airline Company is better than PanAir)
  - **Primary actors should be to the left side of the diagram** – This enables you to quickly highlight the important roles in the system.
  - **Actors don’t interact with other actors**

Use Case Diagrams (UCD)

- **System boundary** (also called system or subject) is presented by a rectangle with system's name, associated keywords and stereotypes in the top left corner. Use cases applicable to the system are located inside the rectangle and actors - outside of the system boundary.
- It is an *optional* element.

Use Case Diagrams (UCD)

- **Use Cases**
  - **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.

Use Case Diagrams (UCD)

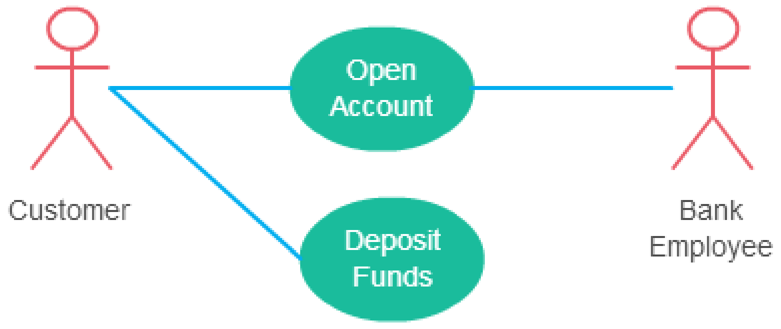
- **Relationships in Use Case Diagrams**
- There are four main types of relationships in a use case diagram. They are

1. Association between an actor and a use case
2. Extend relationship between two use cases
3. Include relationship between two use cases
4. Generalization of a use case

Relationships in Use Case Diagrams

1. Association Between Actor and Use Case

- This one is straightforward and present in every use case diagram. Few things to note.
  - 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.



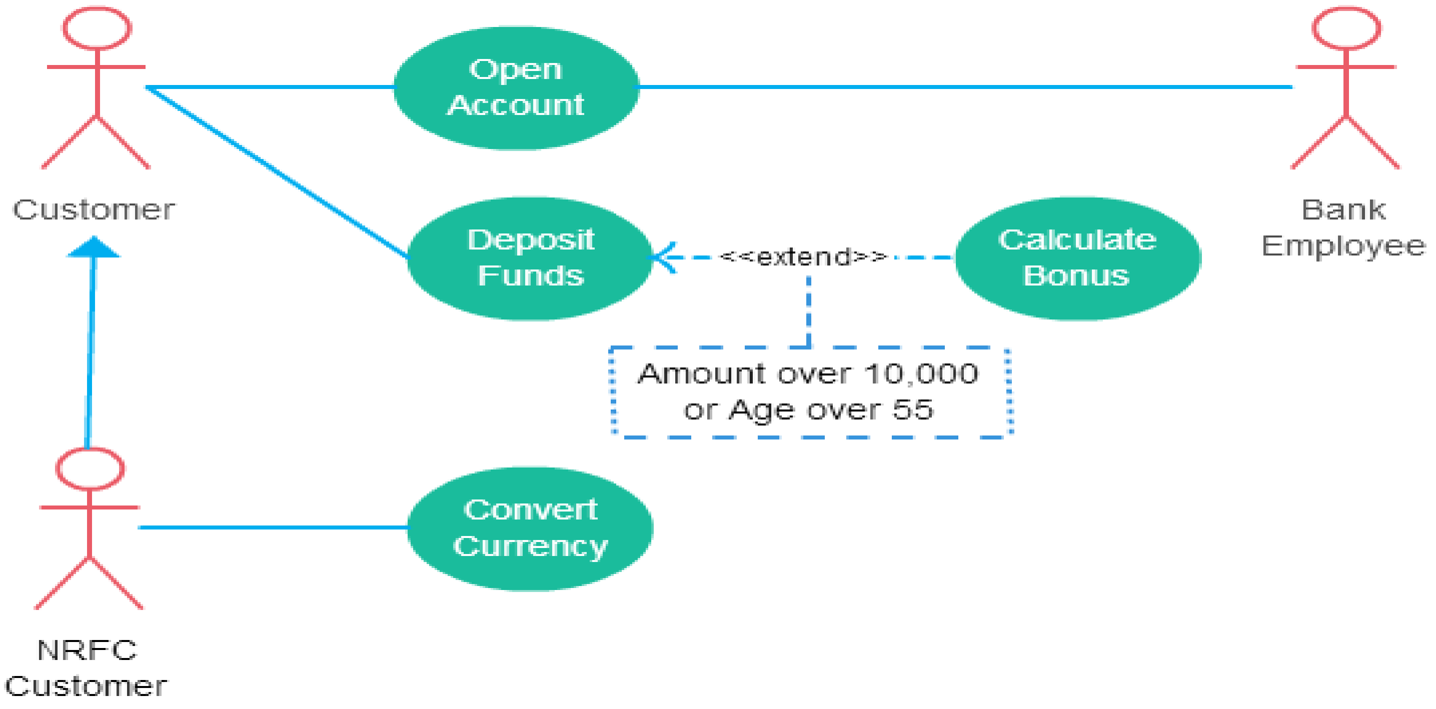
Relationships in Use Case Diagrams

2. Extend Relationship Between Two Use Cases

- As the name implies it extends the base use case and adds more functionality to the system

- Here are few things to consider when using the **<<extend>>** relationship.
  - **The extending use case is dependent on the extended (base) use case**. In the below diagram the "Calculate Bonus" use case doesn't make much sense without the "Deposit Funds" use case.

Relationships in Use Case Diagrams



Relationships in Use Case Diagrams

•The extending use case is usually optional and can be triggered conditionally. In the diagram you can see that the extending use case is triggered only for deposits over 10,000 or when the age is over 55.

•The extended (base) use case must be meaningful on its own. This means it should be independent and must not rely on the behavior of the extending use case.

Relationships in Use Case Diagrams

3. Include Relationship Between Two Use Cases

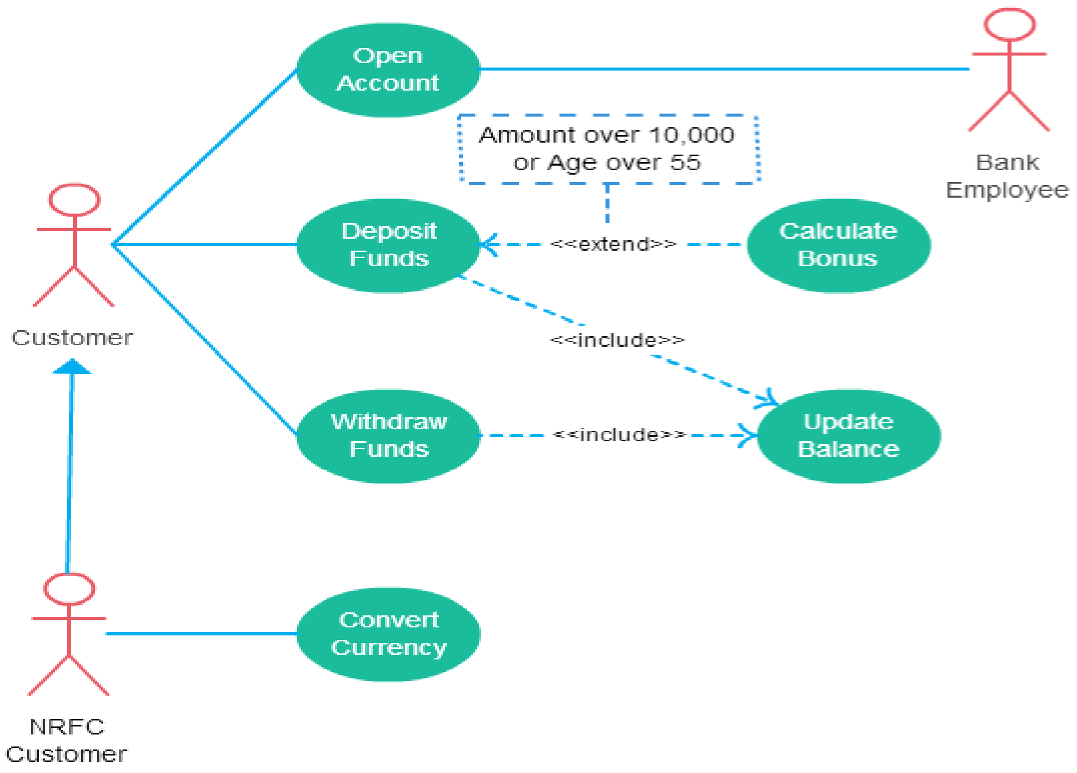
- Include relationship show that the behavior of the included use case is part of the including (base) use case. The main reason for this is to reuse the common actions across multiple use cases. In some situations this is done to simplify complex behaviors. Few things to consider when using the <<include>> relationship.

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

- The included use case is mandatory and not optional.

Relationships in Use Case Diagrams

- Lest expand our banking system use case diagram to show include relationships as well.

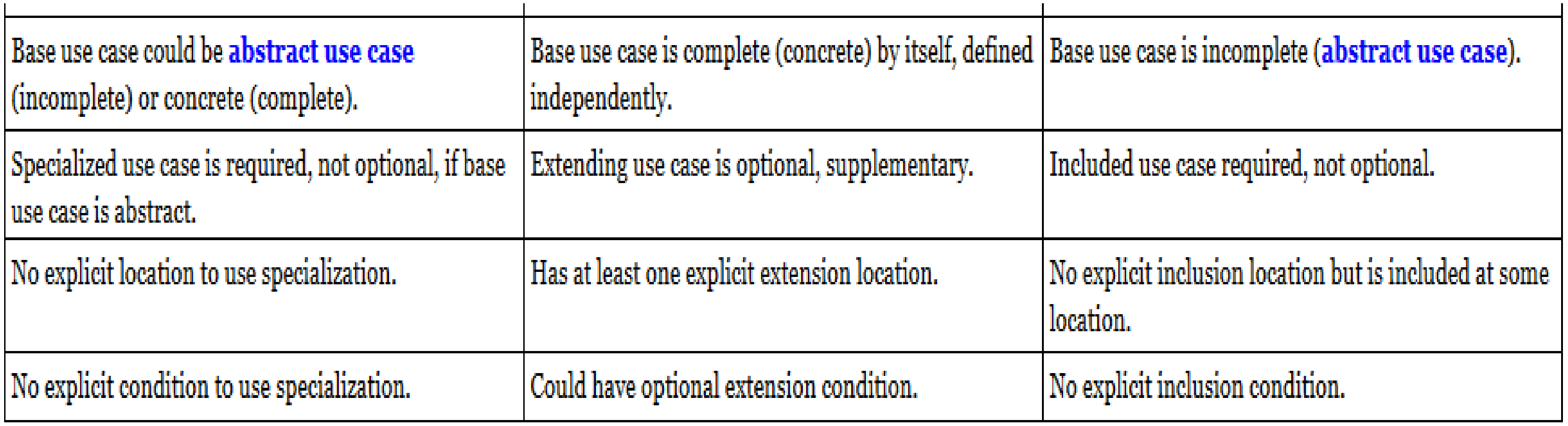


Relationships in Use Case Diagrams

4. Generalization of a Use Case

- The behavior of the ancestor is inherited by the descendant. This is used when there are common behavior between two use cases and also specialized behavior specific to each use case.
- For example in the previous banking example there might be an use case called "Pay Bills". This can be generalized to "Pay by Credit Card", "Pay by Bank Balance" etc.

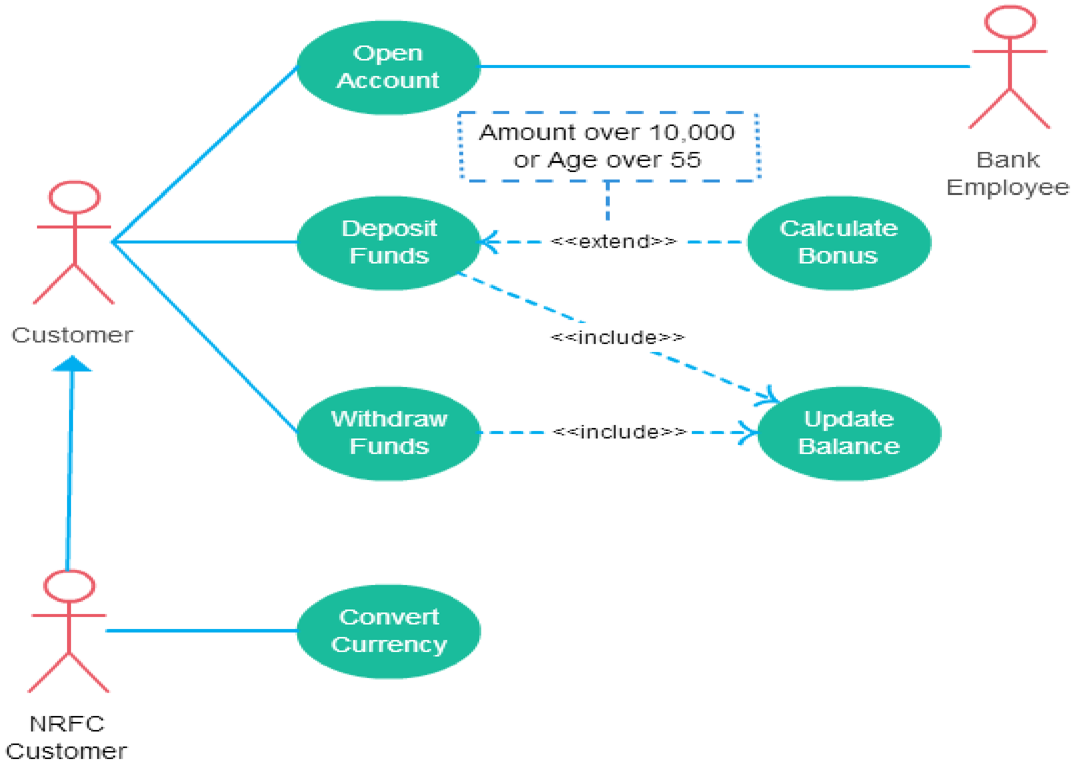
Relationships in Use Case Diagrams



Generalization	Extend	Include
Base use case could be **abstract use case** (incomplete) or concrete (complete).	Base use case is complete (concrete) by itself, defined independently.	Base use case is incomplete (**abstract use case**).
Specialized use case is required, not optional, if base use case is abstract.	Extending use case is optional, supplementary.	Included use case required, not optional.
No explicit location to use specialization.	Has at least one explicit extension location.	No explicit inclusion location but is included at some location.
No explicit condition to use specialization.	Could have optional extension condition.	No explicit inclusion condition.

Relationships in Use Case Diagrams

- Arrow points to the base use case when using <<extend>>
- Arrow points to the included use case when using <<include>>
- Both <<extend>> and <<include>> are shown as dashed arrows.
- Actor and use case relationship doesn’t show arrows.


Open Account
Bank Employee
Amount over 10,000 or Age over 55
Customer
Deposit Funds
<<extend>>
Calculate Bonus
<<include>>
Withdraw Funds
<<include>>
Update Balance
NRFC Customer
Convert Currency

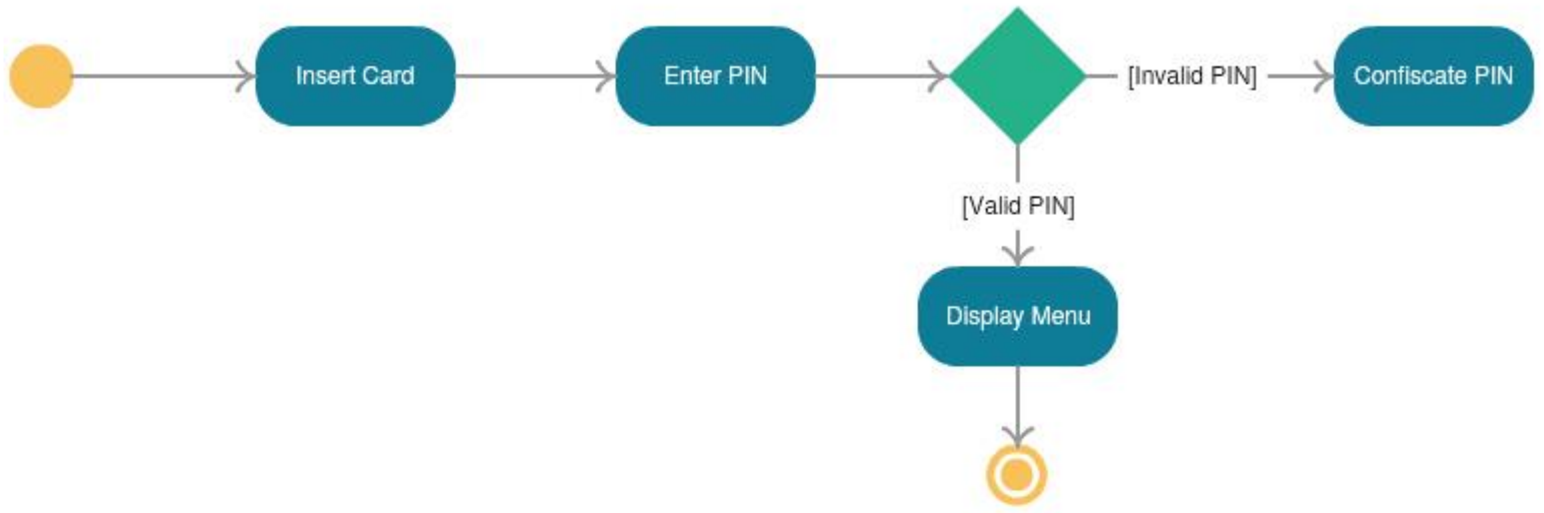
Activity Diagram

- Activity diagrams represent workflows in a graphical way.

- Activity diagrams are constructed from a limited number of shapes, connected with arrows. The most important shape types:

1. rounded rectangles represent actions.
2. diamonds represent decisions.
3. bars represent the start (split) or end (join) of concurrent activities.
4. a black circle represents the start (initial state) of the workflow.
5. an encircled black circle represents the end (final state).

Activity Diagram



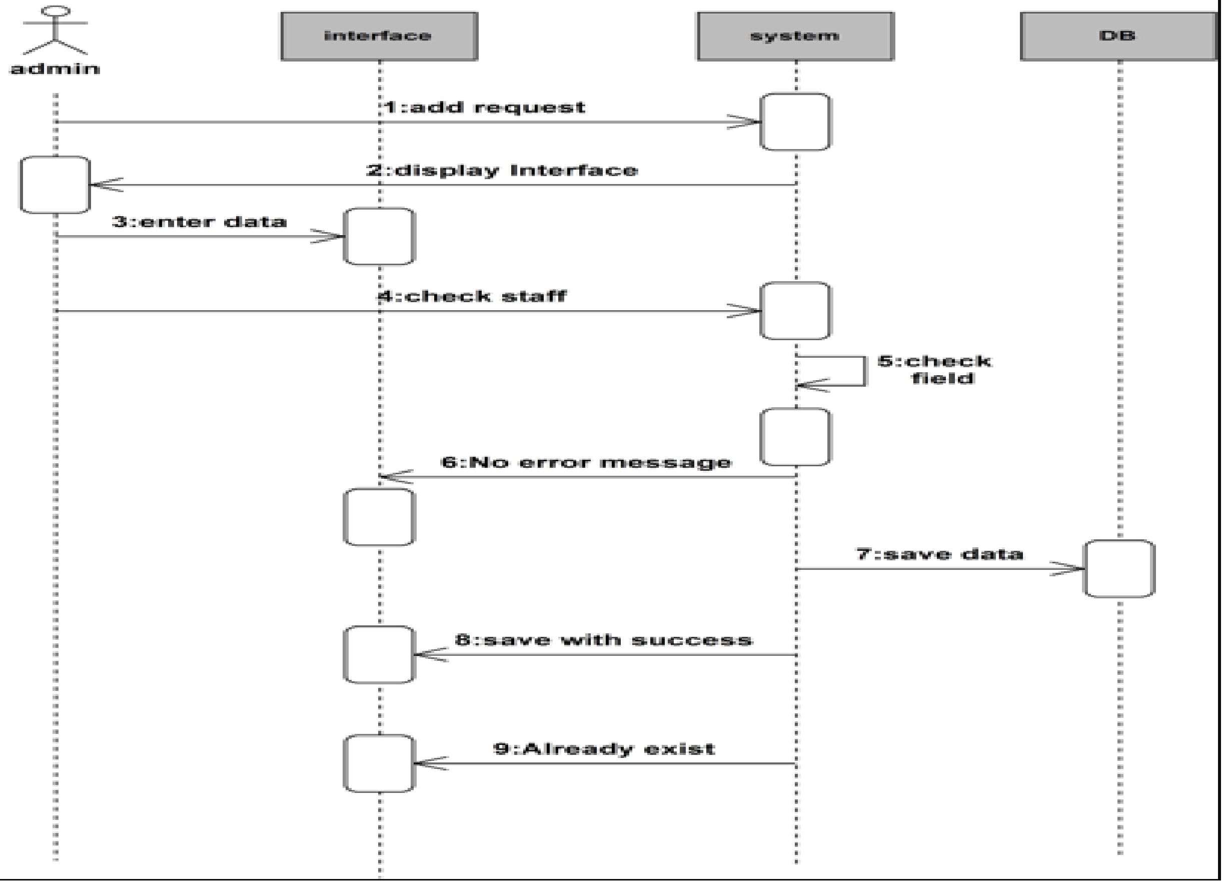
Sequence Diagram

- Sequence diagrams in UML show how objects interact with each other and the order those interactions occur.

- It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario.

- Sequence diagrams are sometimes called event diagrams or event scenarios.

Example: Add staff


admin
interface
system
DB
1:add request
2:display Interface
3:enter data
4:check staff
5:check field
6:No error message
7:save data
8:save with success
9:Already exist