HNDIT3012 - OBJECT ORIENTED PROGRAMMING

UNIFIED MODELING LANGUAGE (UML) | AN INTRODUCTION

UNIFIED MODELING LANGUAGE (UML)

- It is a general purpose modelling language.
- The main aim of UML is to define a standard way to **visualize** the way a system has been designed.
- It is quite similar to blueprints used in other fields of engineering.
- UML is We use UML diagrams to portray the **behavior and structure** of a system.
- UML helps software engineers, businessmen and system architects with modelling, design and analysis.
- The Object Management Group (OMG) adopted Unified Modelling Language as a standard in 1997.
- International Organization for Standardization (ISO) published UML as an approved standard in 2005.

DO WE REALLY NEED UML?

- Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them.
- UML becomes essential to communicate with non programmers essential requirements, functionalities and processes of the system.
- A lot of time is saved down the line when teams are able to visualize processes, user interactions and static structure of the system.

UML MODEL

- UML is linked with **object oriented** design and analysis.
- UML makes the use of elements and forms associations between them to form diagrams.
- Diagrams in UML can be broadly classified as:
- **Structural Diagrams**
  - Capture static aspects or structure of a system.
  - Structural Diagrams include: Component Diagrams, Object Diagrams, Class Diagrams and Deployment Diagrams.
- **Behavior Diagrams**
  - Capture dynamic aspects or behavior of the system.
  - Behavior diagrams include: Use Case Diagrams, State Diagrams, Activity Diagrams and Interaction Diagrams.

UML DIAGRAMS

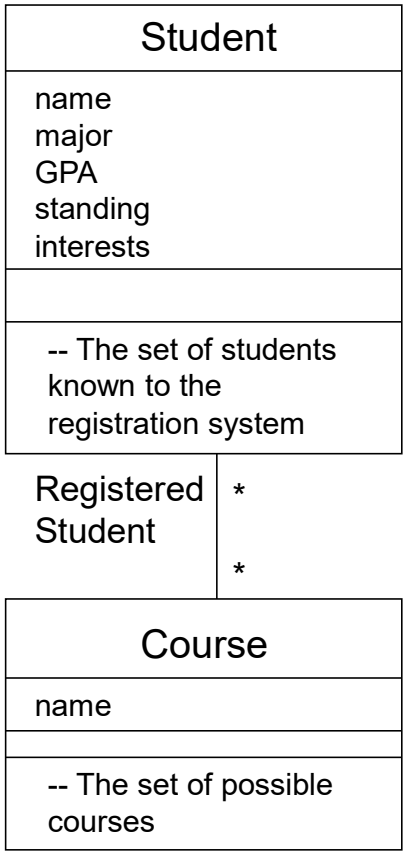
- Use Case Diagrams
- Class Diagrams
- Object Diagrams
- Interaction Diagrams
  - Sequence Diagrams
  - Communication Diagrams
- State Charts (enhanced State Machines)
- Component Diagrams
- Deployment Diagrams

OBJECT ORIENTED CONCEPTS USED IN UML

- **Class –** A class defines the blue print i.e. structure and functions of an object.
- **Objects –** An object is the fundamental unit (building block) of a system which is used to depict an entity.
- **Inheritance –** Inheritance is a mechanism by which child classes inherit the properties of their parent classes.
- **Abstraction –** Abstraction in UML refers to the process of emphasizing the essential aspects of a system or object while disregarding irrelevant details. Abstraction facilitates a clearer understanding and communication among stakeholders.
- **Encapsulation –** Binding data together and protecting it from the outer world is referred to as encapsulation.
- **Polymorphism –** Mechanism by which functions or entities are able to exist in different forms.

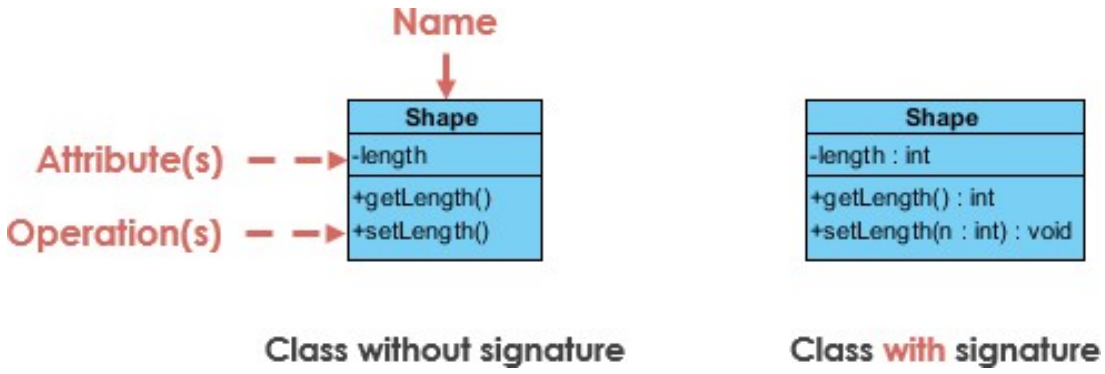
ANALYSIS PERSPECTIVE

- Classes are sets of objects
- Classes may include attributes and operations, but more importantly their intents are defined by responsibilities
- Relationships are set of links between objects
- Components relate to the problem domain



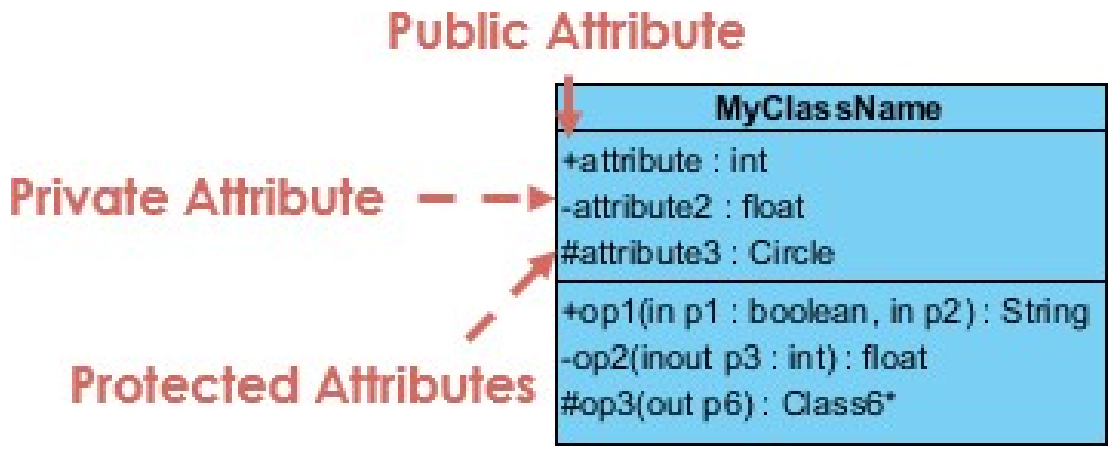
UML CLASS NOTATION

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

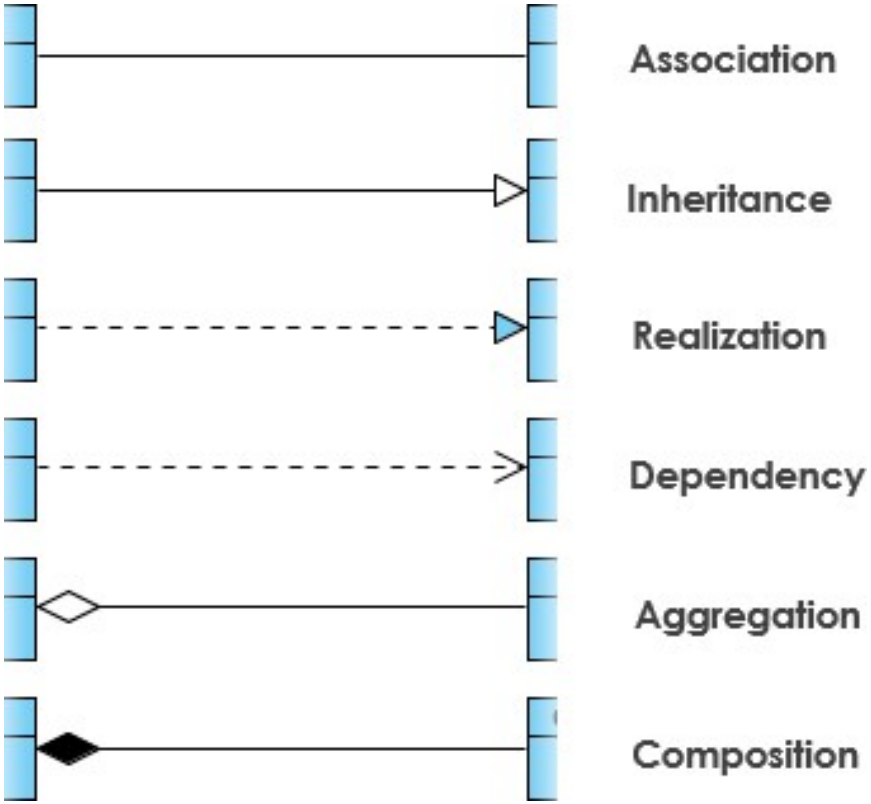


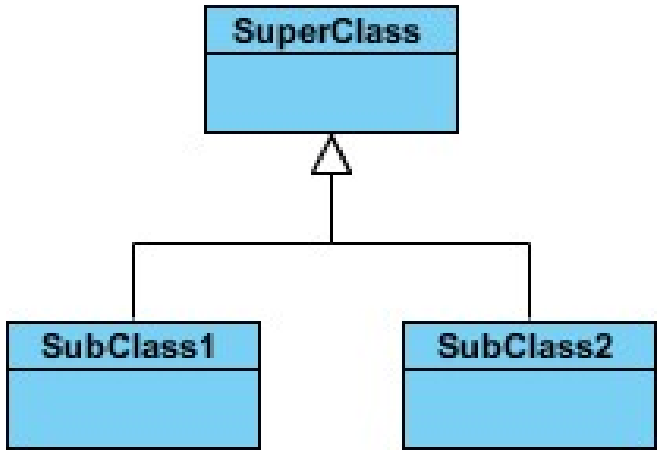
CLASS VISIBILITY

- The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.

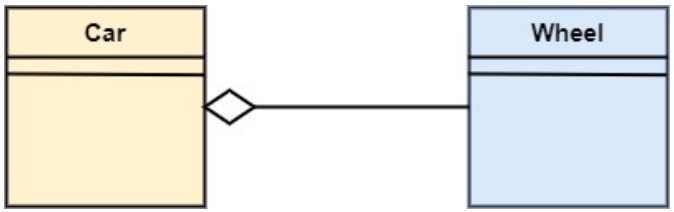


RELATIONSHIPS BETWEEN CLASSES

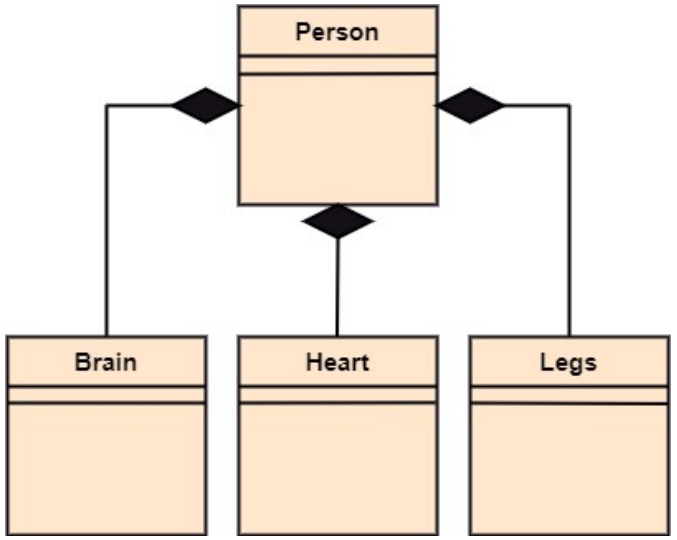




ASSOCIATION VS AGGREGATION

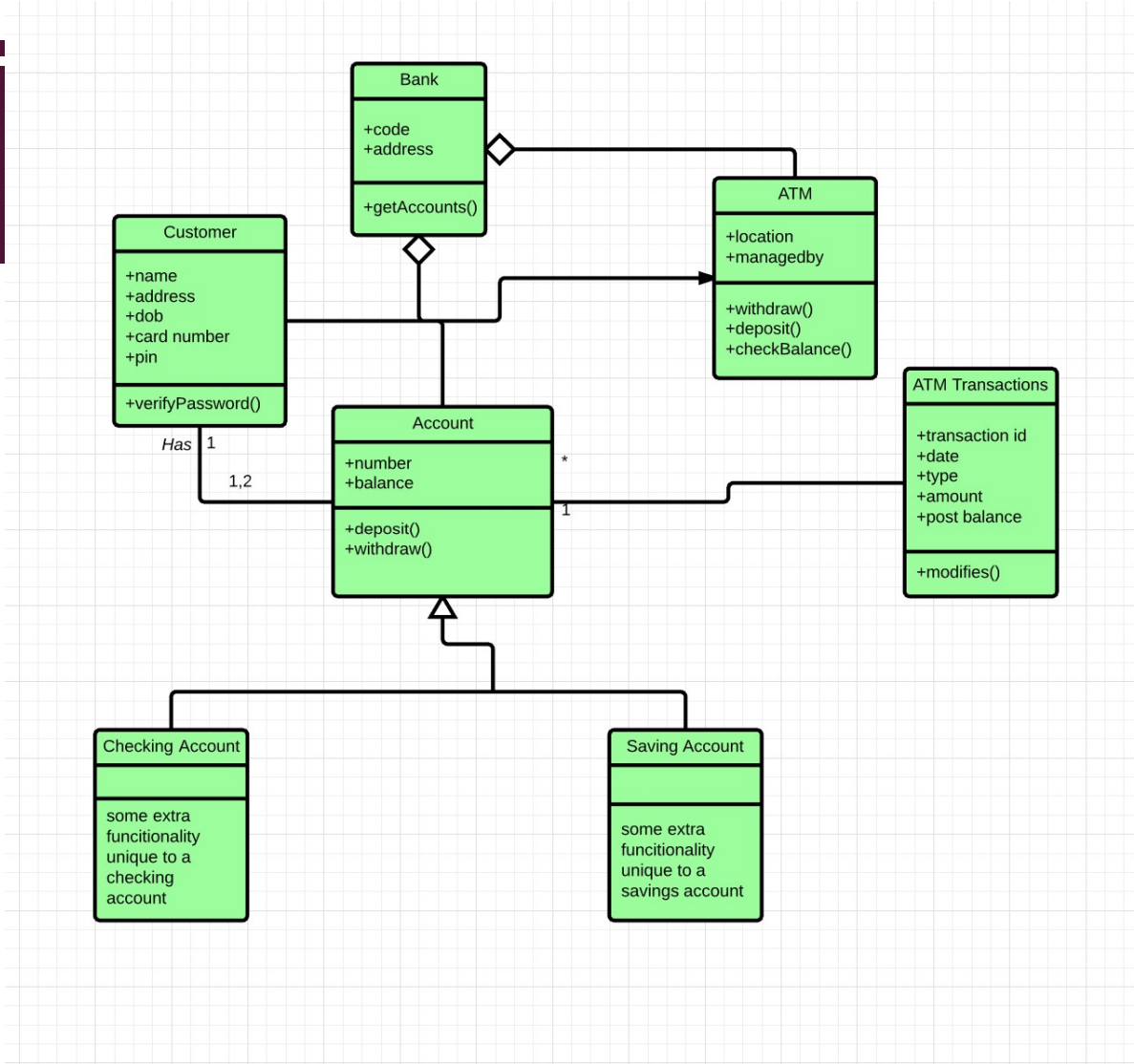


The wheel object can exist without the car object, which proves to be an aggregation relationship.



If the person is destroyed, the brain, heart, and legs will also get discarded.

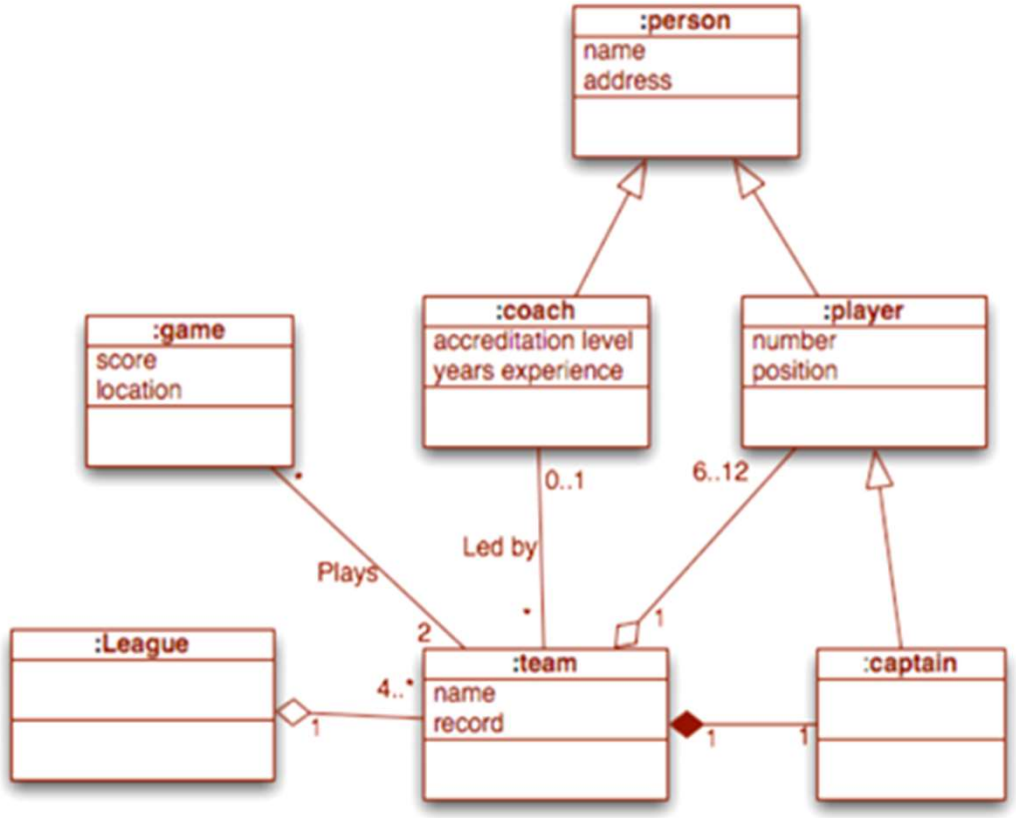
EXAMPLE


Bank
+code
+address
+getAccounts()
ATM
+location
+managedby
+withdraw()
+deposit()
+checkBalance()
Customer
+name
+address
+dob
+card number
+pin
+verifyPassword()
Has
1
1,2
Account
+number
+balance
+deposit()
+withdraw()
*
1
ATM Transactions
+transaction id
+date
+type
+amount
+post balance
+modifies()
Checking Account
some extra funcitionality unique to a checking account
Saving Account
some extra funcitionality unique to a savings account

EXAMPLE

- Consider the following scenario, draw a class diagram for this information, and be sure to label all associations with appropriate multiplicities
- A hockey league is made up of at least four hockey teams. Each hockey team is composed of six to twelve players, and one player captains the team. A team has a name and a record. Players have a number and a position. Hockey teams play games against each other. Each game has a score and a location. Teams are sometimes lead by a coach. A coach has a level of accreditation and a number of years of experience, and can coach multiple teams. Coaches and players are people, and people have names and addresses.

ANSWER

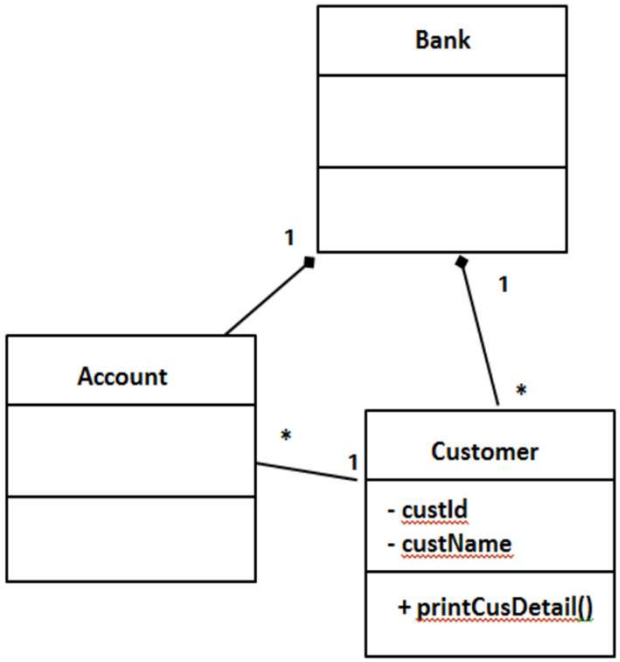

:person
name
address
:coach
accreditation level
years experience
:player
number
position
:game
score
location
:League
:team
name
record
:captain
0..1
6..12
Led by
Plays
2
4..*
1
1
1
1

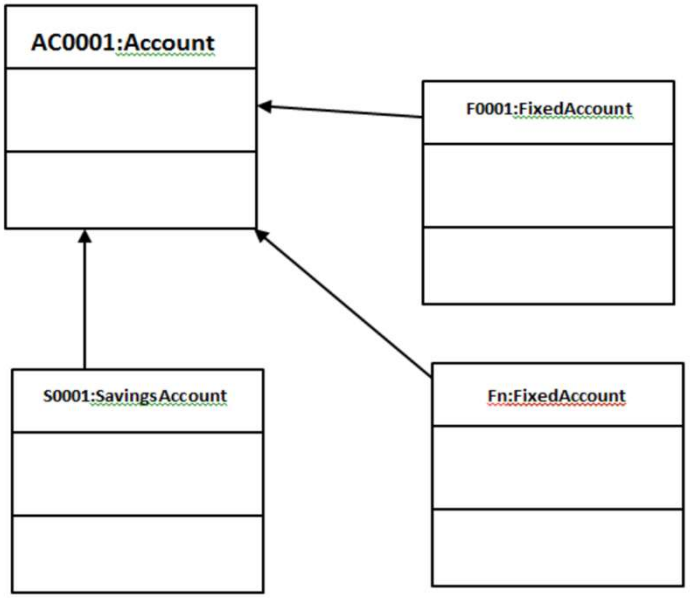
EXAMPLE 03

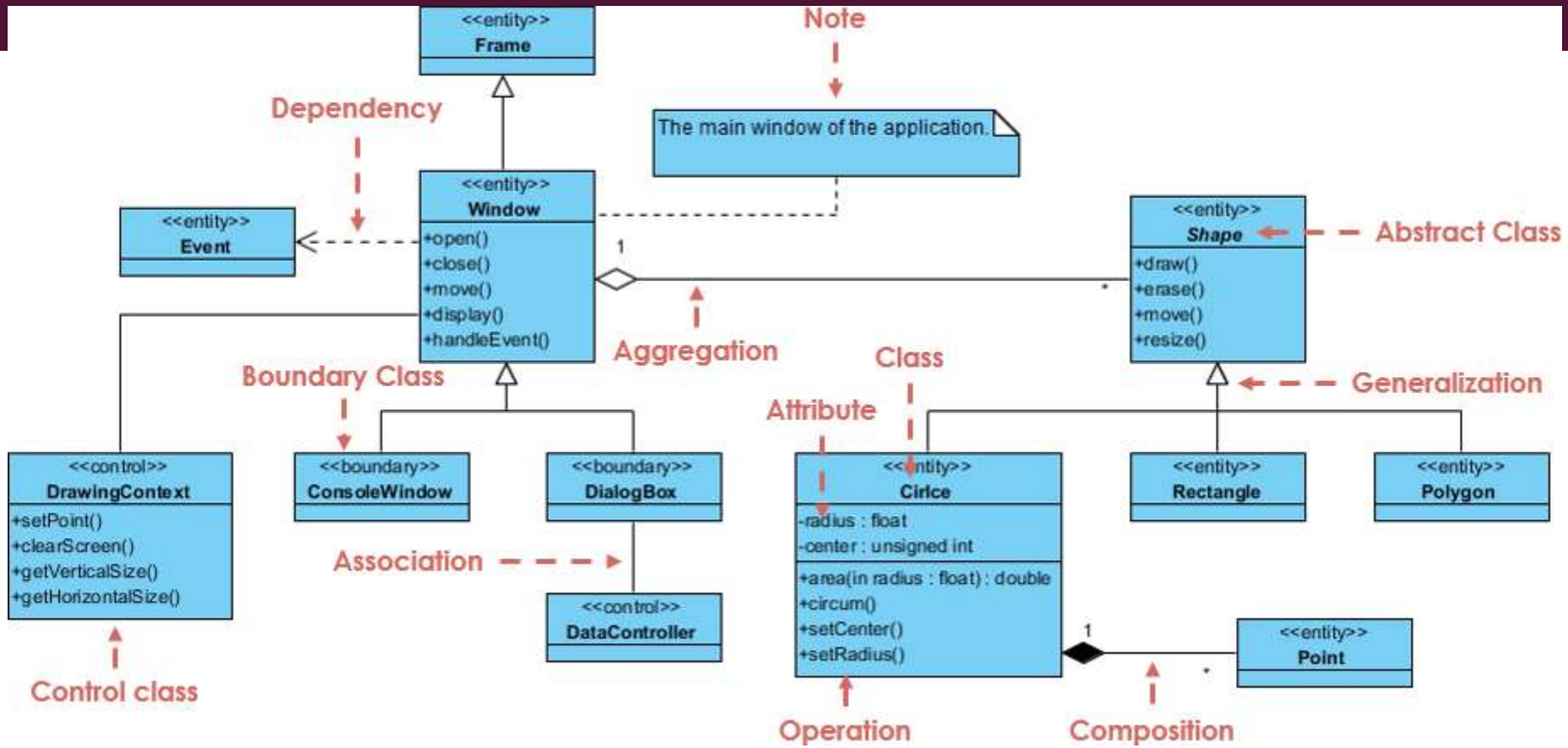
Consider the given scenario for a bank.

Bank has large number of accounts and customers. Customer can have any number of accounts. Assume **custId**, **custName** are private data of the customer and **PrintCusDetail** is a public method of the customer. The accounts can be divided in to two groups, as saving account and fixed account. Each savings account is created with a serial number starting ‘s’ , and Each Fixed account is created with a serial number starting ‘F’. Draw the class diagram to illustrate design specifications for the bank

ANSWER


Bank
1
1
Account
*
*
1
Customer
- custId
- custName
+ printCusDetail()


AC0001:Account
F0001:FixedAccount
S0001:SavingsAccount
Fn:FixedAccount


<<entity>>
Frame
Note
Dependency
The main window of the application.
<<entity>>
Window
+open()
+close()
+move()
+display()
+handleEvent()
<<entity>>
Event
1
*
<<entity>>
Shape
+draw()
+erase()
+move()
+resize()
Abstract Class
Aggregation
Boundary Class
Class
Attribute
Generalization
<<control>>
DrawingContext
+setPoint()
+clearScreen()
+getVerticalSize()
+getHorizontalSize()
<<boundary>>
ConsoleWindow
<<boundary>>
DialogBox
<<entity>>
Cirlce
-radius : float
-center : unsigned int
+area(in radius : float) : double
+circum()
+setCenter()
+setRadius()
<<entity>>
Rectangle
<<entity>>
Polygon
Association
<<control>>
DataController
1
*
<<entity>>
Point
Control class
Operation
Composition

WHAT IS A USE CASE DIAGRAM?

- A use case diagram is used to represent the dynamic behavior of a system.
- It encapsulates the system's functionality by incorporating use cases, actors, and their relationships.
- It models the tasks, services, and functions required by a system/subsystem of an application.
- It depicts the high-level functionality of a system and also tells how the user handles a systemScenarios in which your system or application interacts with people, organizations, or external systems

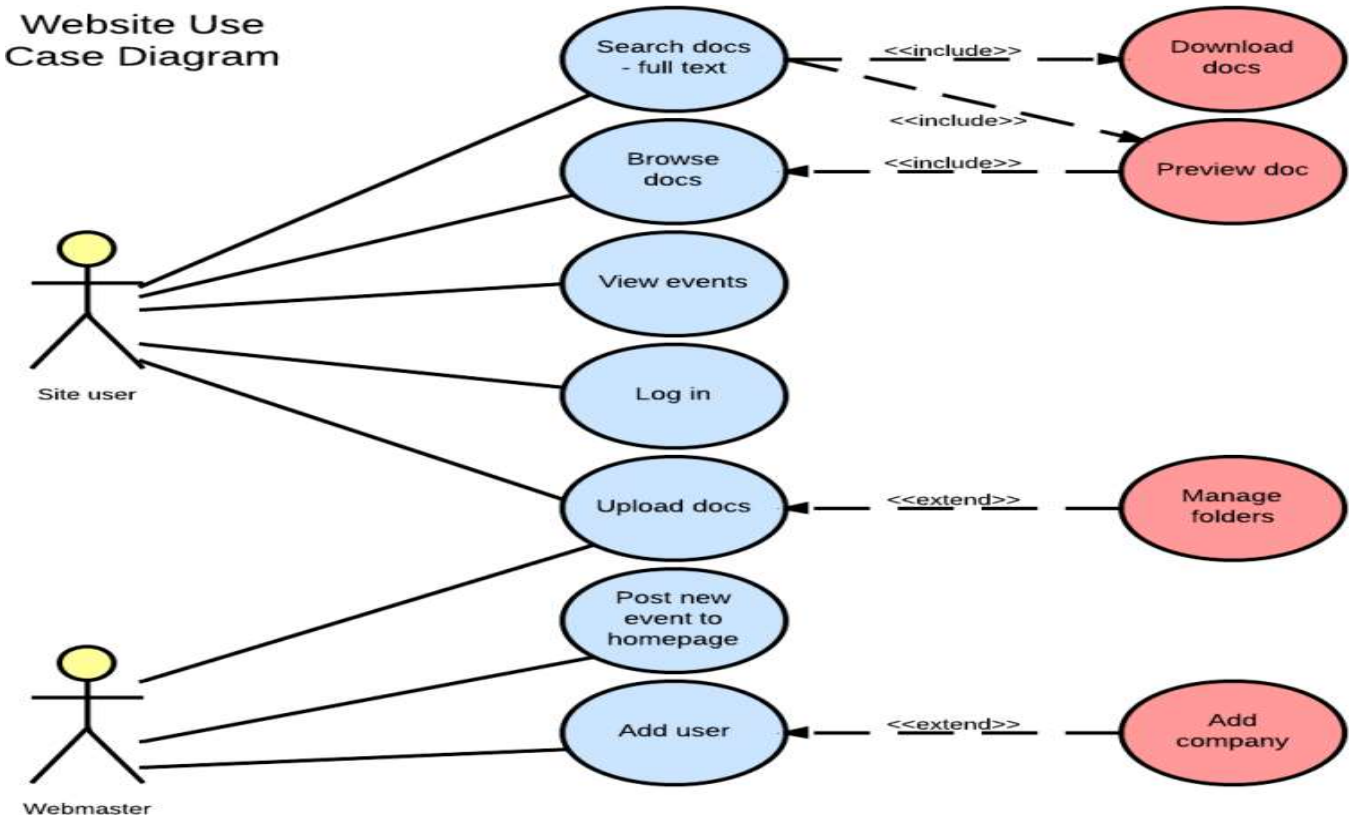
PURPOSE OF USE CASE DIAGRAMS

- It gathers the system's needs.
- It depicts the external view of the system.
- It recognizes the internal as well as external factors that influence the system.
- It represents the interaction between the actors.

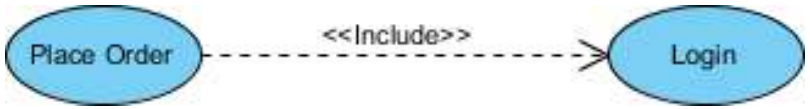
USE CASE DIAGRAM SYMBOLS AND NOTATION

- **Use cases:** Horizontally shaped ovals that represent the different uses that a user might have.
- **Actors:** Stick figures that represent the people actually employing the use cases.
- **Associations:** A line between actors and use cases. In complex diagrams, it is important to know which actors are associated with which use cases.
- **System boundary boxes:** A box that sets a system scope to use cases. All use cases outside the box would be considered outside the scope of that system.
- **Packages:** A UML shape that allows you to put different elements into groups. Just as with component diagrams, these groupings are represented as file folders.

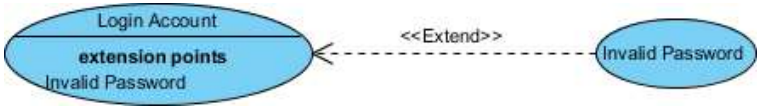
USE CASE EXAMPLE



INCLUDE VS EXTEND



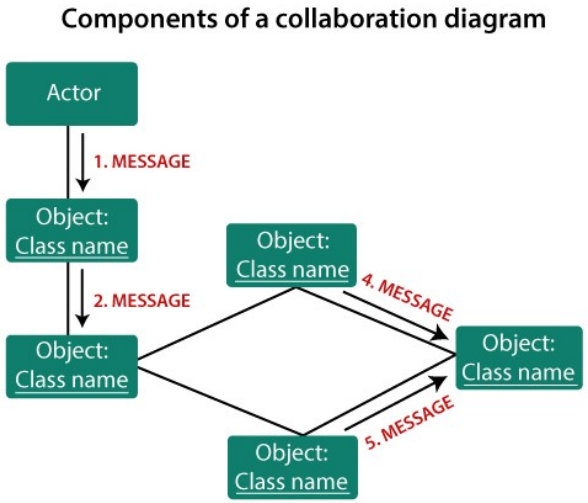
- The <<Include>> relationship is used to include common behavior from an included use case into a base use case in order to support the reuse of common behavior.



- The <<extend>> relationship is used to include optional behavior from an extending use case in an extended use case.

COLLABORATION DIAGRAM

- The collaboration diagram is used to show the relationship between the objects in a system.
- Both the sequence and the collaboration diagrams represent the same information but differently.
- The collaboration diagram, which is also known as a communication diagram, is used to portray the object's architecture in the system.



SEQUENCE DIAGRAMS

- Sequence diagrams describe interactions among classes in terms of an exchange of messages over time.
- They're also called event diagrams.
- A sequence diagram is a good way to visualize and validate various runtime scenarios.
- These can help to predict how a system will behave and to discover responsibilities a class may need to have in the process of modeling a new system.

PURPOSE OF SEQUENCE DIAGRAM

- Model high-level interaction between active objects in a system
- Model the interaction between object instances within a collaboration that realizes a use case
- Model the interaction between objects within a collaboration that realizes an operation
- Either model generic interactions (showing all possible paths through the interaction) or specific instances of a interaction (showing just one path through the interaction)