OOS Assignment - 3

Q1). what is Use Case Methodology ? What must be a Use-Case contain ? why it is so important in unified process Prepare a Use Case diagram for Bank ATM system.

⇒ Use case methodology - It is description of set of sequences of actions, that system perform to achieve desirable result to the user. It describe the behaviour of system, in UML it is most useful methodology to show overall behaviour of system.

⇒ Use case contain following things -

1. Subject
2. Use-Cases
3. Actor
4. Various relationship such as generalization, include and extend.

Subject - The subject denotes the purpose of the use case diagram. It is written as a text within a rectangular boundary. This boundary is known as system boundary.

Use-Case - The set of ellipse within which some behaviour is specified.

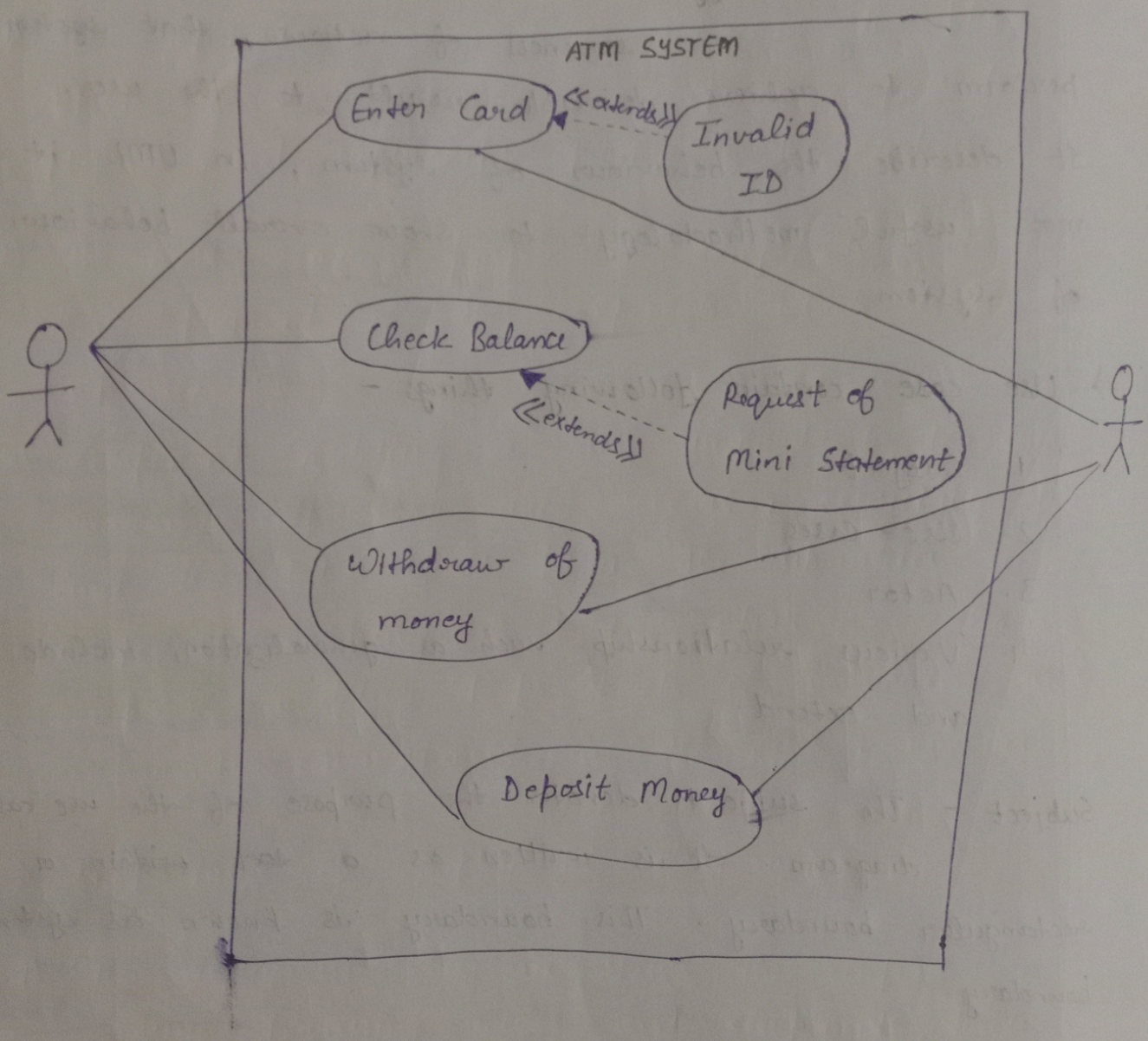
Actors - Actor represents a role when users interact with the use case.

Relationship - These are various relationship which tell us what is the relation between use case.

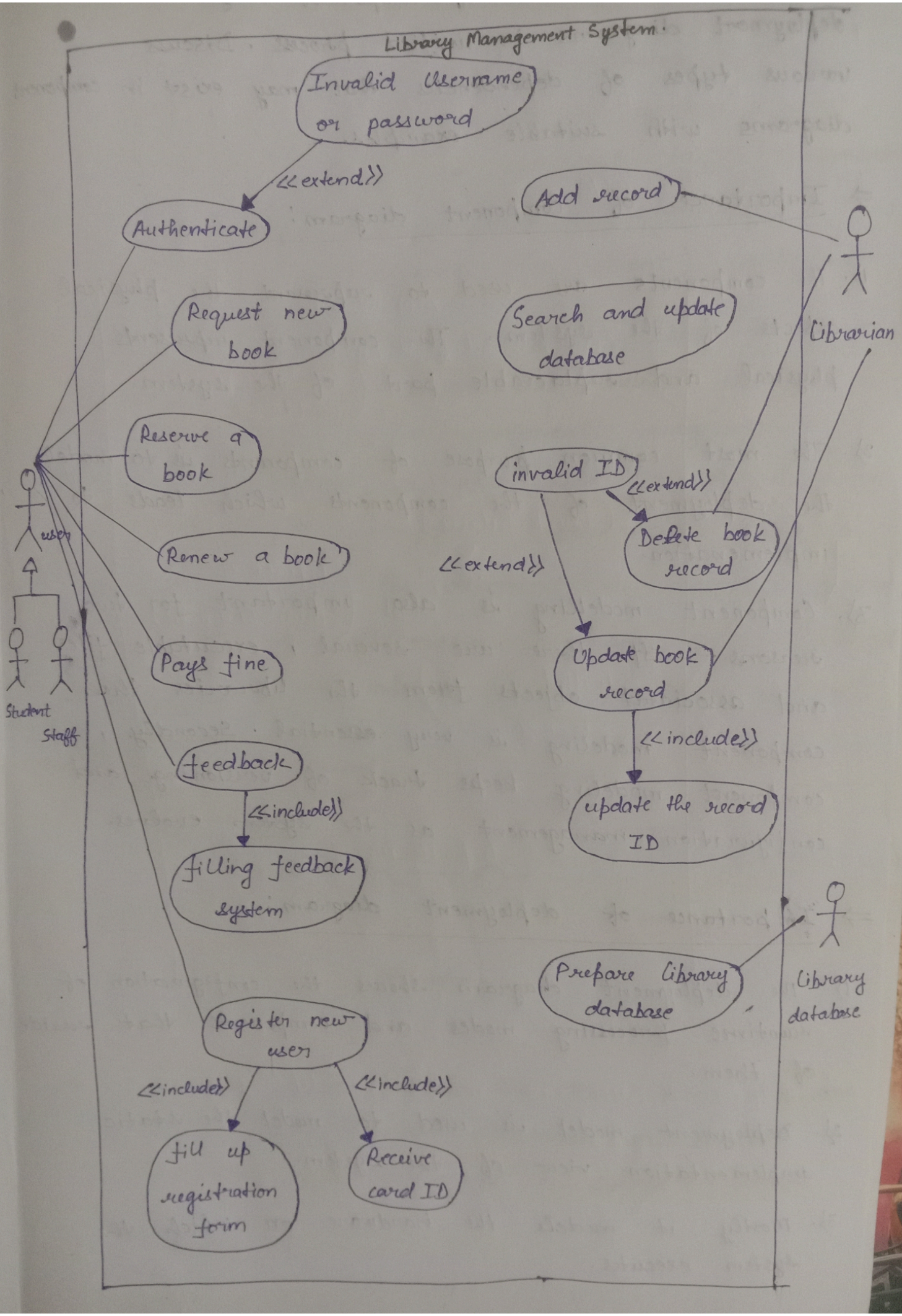
Importance:

1. To model the context of the subject.
2. To model the requirement of the subject.

Use-Case for ATM:



Q12). Prepare a use-case model for library management software system.


Library Management System
Invalid Username or password
<<extend>>
Authenticate
Add record
Librarian
Request new book
Search and update database
Reserve a book
invalid ID
<<extend>>
Delete book record
<<extend>>
user
Renew a book
Pays fine
Update book record
<<include>>
update the record ID
Student
Staff
feedback
<<include>>
filling feedback system
Prepare Library database
Library database
Register new user
<<include>>
<<include>>
fill up registration form
Receive card ID

Q.3) Explain the importance of component diagram and deployment diagrams in unified process. Discuss various types of dependencies that may exist in component diagrams with suitable examples.

⇒ Importance of component diagram:

1). The components are used to represent the physical aspects of the system. The component represents physical and replaceable part of the system.

2). The most common purpose of components is to model the deployment of the components which leads to implemenation.

3). Component modelling is also important for two reasons - If there are several, executable files and associated objects from the libraries then component modeling is very essential. Secondly, component modeling keeps track of versioning and configuration management as the system evolves.

⇒ Importance of deployment diagram:

1). The deployment diagram shows the configuration of runtime processing modes and components that reside of them.

2). Deployment model is used to model the static implementation view of the system.

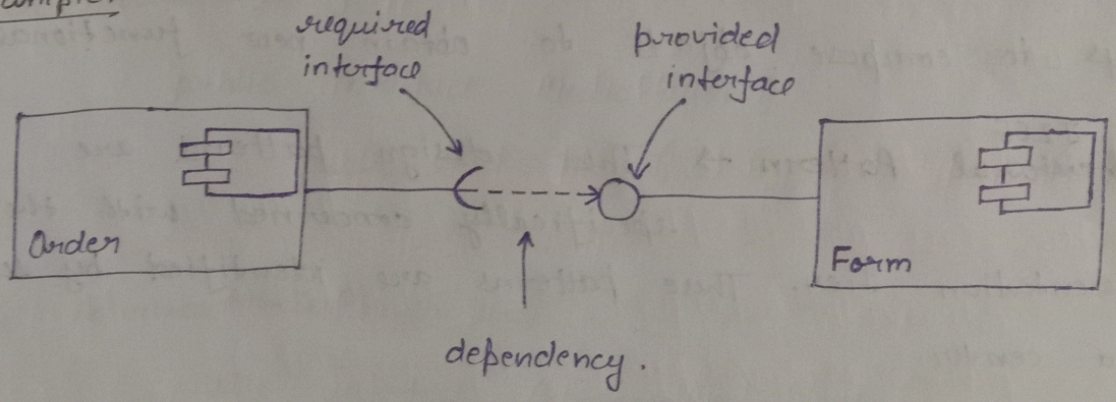
3). Mostly it models the hardware on which the system executes.

⇒ Dependencies in component diagram:

A dependency exists when the functioning of one element depends on the existence of another element. A dependency between two components in a component diagram results in an #include statement in the makefile for the dependent (or client) component.

Dependencies in component diagrams have the same stereotype values as dependencies created in object model diagrams. In a component diagram, a dependency relation is also used in the definition of a component interface. Dependencies appear in the browser under the dependent or client component. Dependency in collaboration diagram gets established between required interface and provided interface.

Example:-



Q4). What are design patterns in UML? How they are useful? Briefly discuss creational or adaptor patterns with description and a suitable example code.

Design patterns represent the best practices used by experienced object - oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. Those solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Types of Design pattern:

1). Creational Patterns → These design patterns provides way to create objects while hiding the creation logic, rather than instantiating objects need to be created for a given use case.

2). Structural Patterns → These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

3). ~~Behavioral~~ J2EE Pattern → These design patterns are specifically concerned with the presentation tier. These patterns are identified by sun Java center.

4). Behavioral Pattern → These design patterns are specifically concerned with communication between objects

⇒ Creational pattern or Adapter pattern:

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes

structural pattern as this pattern combines the capability of two independent interfaces.

These pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugins the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

Lets demonstrate use of adapter pattern via following example in which an audio player device can play mp3 files only and wants to use an advanced audio player vlc and mp4 files.

Step 1: Create interfaces for Media player and Advanced Media player:

(i) MediaPlayer.java

```
public interface MediaPlayer {
    public void play(String audioType, String fileName);
}
```

(ii) AdvancedMediaPlayer.java

```
public interface AdvancedMediaPlayer {
    public void playVlc(String fileName);
    public void playMp4(String fileName);
}
```

Step 2: Create concrete classes implementing the above interface.

⇒ VlcPlayer.java:

```
public class VlcPlayer implements AdvancedMediaPlayer {
    public void playVlc(String fileName) {
        System.out.println("Playing vlc file. Name: "
                + fileName);
    }
    public void playMp4(String fileName) {
    }
}
```

⇒ Mp4Player.java:

```
public class Mp4Player implements AdvancedMediaPlayer {
    public void playVlc(String fileName) {
    }
    public void playMp4(String fileName) {
        System.out.println("Playing mp4 file. Name: "+
                fileName);
    }
}
```

Step 3: Create adapter class implementing the MediaPlayer interface.

⇒ MediaAdapter.java:

```
public class MediaAdapter implements MediaPlayer {
    AdvancedMediaPlayer advancedMusicPlayer;
    public MediaAdapter(String audioType) {
        if(audioType.equalsIgnoreCase("vlc")) {
            advancedMusicPlayer = new VlcPlayer();
        } else if(audioType.equalsIgnoreCase("mp4")) {
            advancedMusicPlayer = new Mp4Player();
        }
    }
```

```
    public void play(String audioType, String fileName){
        if(audioType.equalsIgnoreCase("vlc")){
            advancedMusicPlayer.playVlc(fileName);
        }else if(audioType.equalsIgnoreCase("mp4")){
            advancedMusicPlayer.playMp4(fileName);
        }
    }
}
```

Step 4: Create concrete class implementing the MediaPlayer interface.

AudioPlayer.java:

```
public class AudioPlayer implements MediaPlayer {
    MediaAdapter mediaAdapter;
    public void play(String audioType, String fileName){
        if(audioType.equalsIgnoreCase("mp3")){
            System.out.println("playing mp3 file. Name: "
                    + fileName);
        }
        else if(audioType.equalsIgnoreCase("vlc") ||
                audioType.equalsIgnoreCase("mp4")){
            mediaAdapter = new MediaAdapter(audioType);
            mediaAdapter.play(audioType, fileName);
        }else{
            System.out.println("Invalid media. "+
                    audioType);
        }
    }
}
```

Step 5]. Use the AudioPlayer to play different types of audio formats.

AdapterPatternDemo.java:

```
public class AdapterPatternDemo {
    public static void main(String args[]) {
        AudioPlayer audioPlayer = new AudioPlayer();
        audioPlayer.play("mp3", "beyond the horizon.mp3");
        audioPlayer.play("mp4", "alone.mp4");
        audioPlayer.play("vlc", "far far away.vlc");
        audioPlayer.play("avi", "mind me.avi");
    }
}
```

Q5]. Write short notes on two:

(i) MVC Design patter:

→ MVC stands for Mode view controller.

→ The model contains only the pure application data, It contains no longer logic describing how to present the data to a user.

→ The view presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to amanipulate it.

→ The controller exists between the view and the model. It listens to events triggered by the view (or another external source) and executes the appropriate reaction to these events. In most cases, the reaction is to call a method on the model.

(ii) Features of C# that are not present in C++ and java.

⇒

	C++	C#
(i)	In C++ memory management is performed manually by the programmer.	In C# memory management is performed automatically.
(ii)	C++ code can be run on any platform.	C# is window specific.
(iii)	C++ includes very complex features.	C# is quite easy because it has the well defined hierarchy of classes.
(iv)	In C++, size of binaries is low and light weight.	In C#, size of binaries is high because of overhead libraries.

⇒

	Java	C#
(i)	Java does not support operator overloading	C# support operator overloading.
(ii)	In java there can be only one public class in source code	In C#, we can have many public class in source code.
(iii)	Java does not support pointers.	We can use pointers in C# in unsafe mode
(iv)	Java does not support goto statement	C# supports goto statement.