Chapter - Five

UML in Analysis and Design

Unified Modelling Language (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.

What UML build under System Analysis and Design?

- **What UML model to build under analysis.**

These are:

- Use case Diagram
- Sequence diagram
- CRC(Class Responsibility and Collaboration)
- Class diagram
- Activity Diagram

It is possible to use supplementary diagrams such as prototype, diagram etc…

- **What UML model to build under Design.**

- **These are:**
  - Refined sequence Diagram
  - Collaboration Diagram
  - State chart diagram
  - Component Diagram
  - Deployment Diagram
  - Refined class diagram
  - Physical Data Modelling (PDM)
  - Graphical User Interface

UML Types

Any system can have two aspects, **static and dynamic**. So, a model is considered as complete when both the aspects are fully covered.

Structural Modeling/Diagrams

Structural modeling captures the static features of a system. They consist of the following −

- Classes diagrams
- Objects diagrams
- Deployment diagrams
- Component diagram
- Package Diagram
- Profile Diagram
- Composite Structure Diagram

Structural model represents the framework for the system and this framework is the place where all other components exist. Hence, the class diagram, component diagram and deployment diagrams are part of structural modeling. They all represent the elements and the mechanism to assemble them. The structural model never describes the dynamic behavior of the system. Class diagram is the most widely used structural diagram. The structural diagrams represent the static aspect of the system. These static aspects represent those parts of a diagram, which forms the main structure and are therefore stable.

Class Diagram

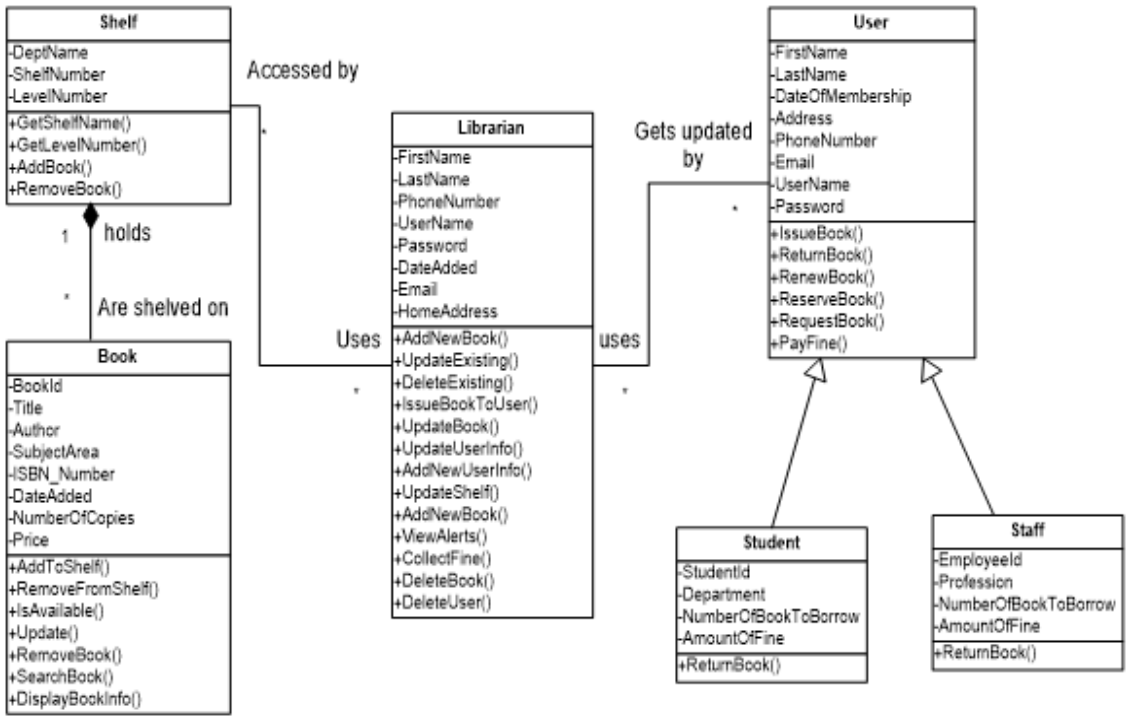
Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, relationships, and collaboration. Class diagrams basically represent the object-oriented view of a system, which is static in nature. A class is a set of objects that share a common structure and common behavior (the same attributes, operations and semantics). A class is an abstraction of real-world items.

- Classes are depicted as **boxes with three sections**:
  - The top one indicates **the name of the class**,
  - The middle one lists the **attributes of the class**, and

- The third one lists the **methods.**

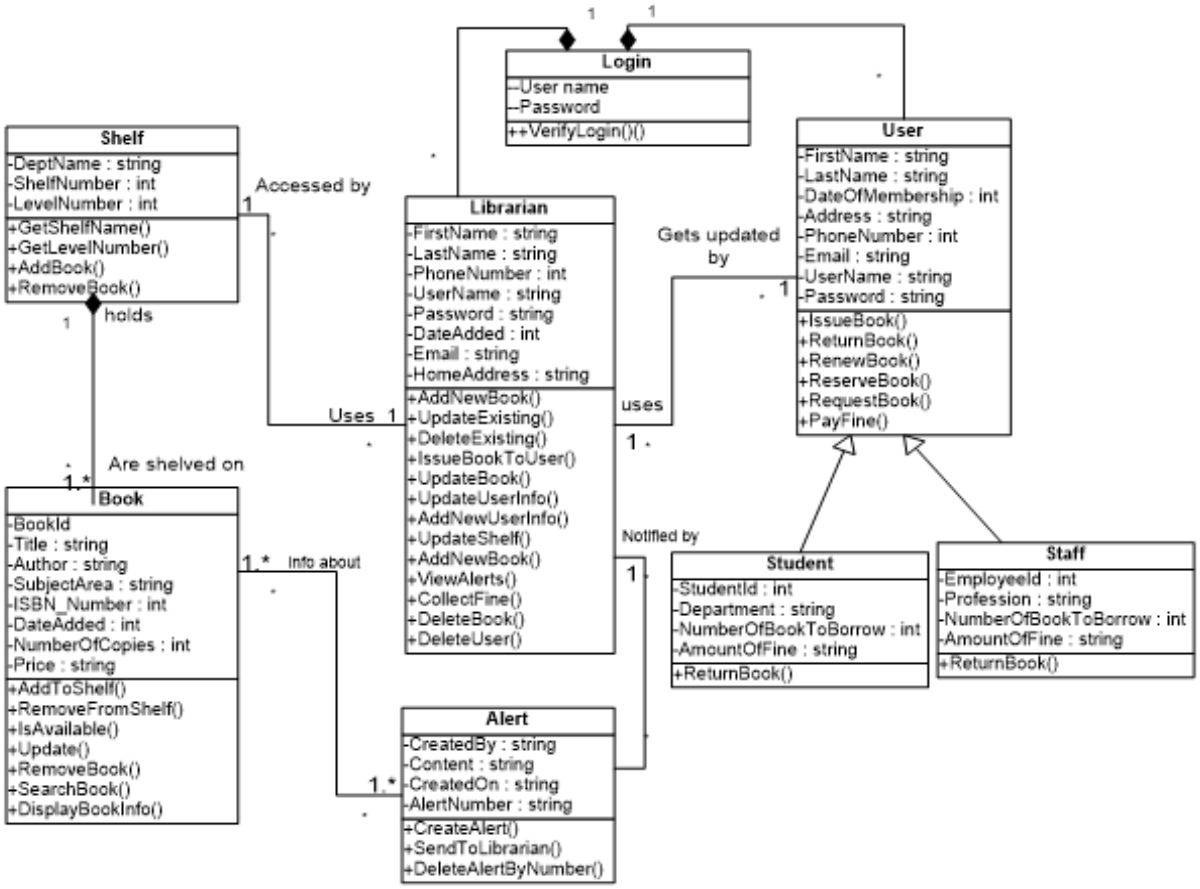
- Or a class can **have two sections**,
  - one for the **name** and
  - One for the **responsibilities**.

Analysis Class Diagram of Library Management System



- To create and evolve a class model, you need to model these items:
  - Classes;
  - Responsibilities(Attributes and methods);
  - Associations;
  - Dependencies;
  - Inheritance relationships;
  - Composition associations; and
  - Association classes.

Design Class Diagram



Design class diagram of Library Management System

Object Diagram

Object diagrams can be described as an instance of class diagram. Thus, these diagrams are more close to real-life scenarios where we implement a system. Object diagrams are a set of objects and their relationship is just like class diagrams. They also represent the static view of the system. The usage of object diagrams is similar to class diagrams but they are used to build prototype of a system from a practical perspective. They show how a system will look like at a given time.

Because there is data available in the objects, they are used to explain complex relationships between objects.

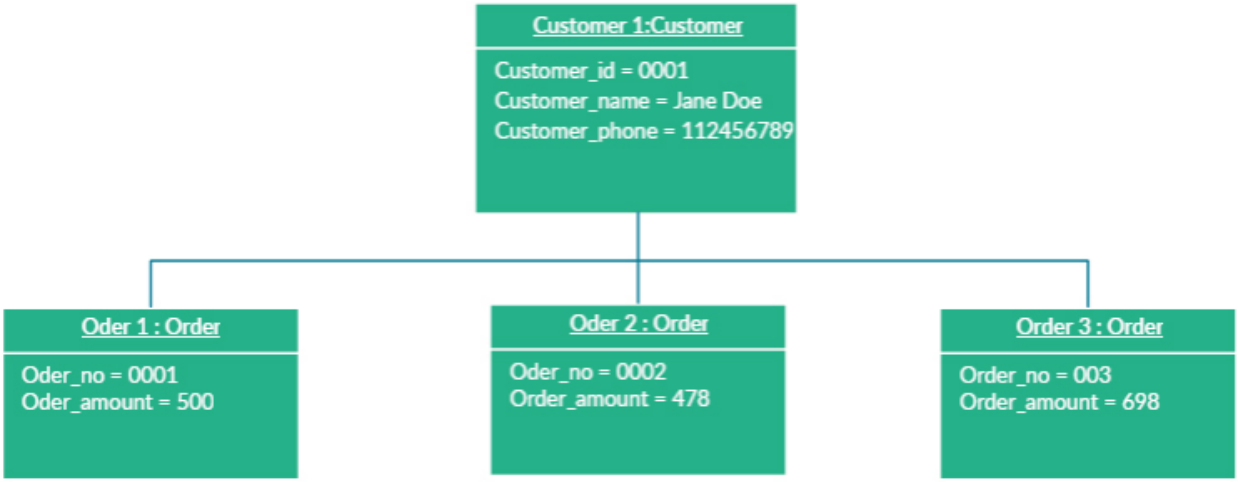
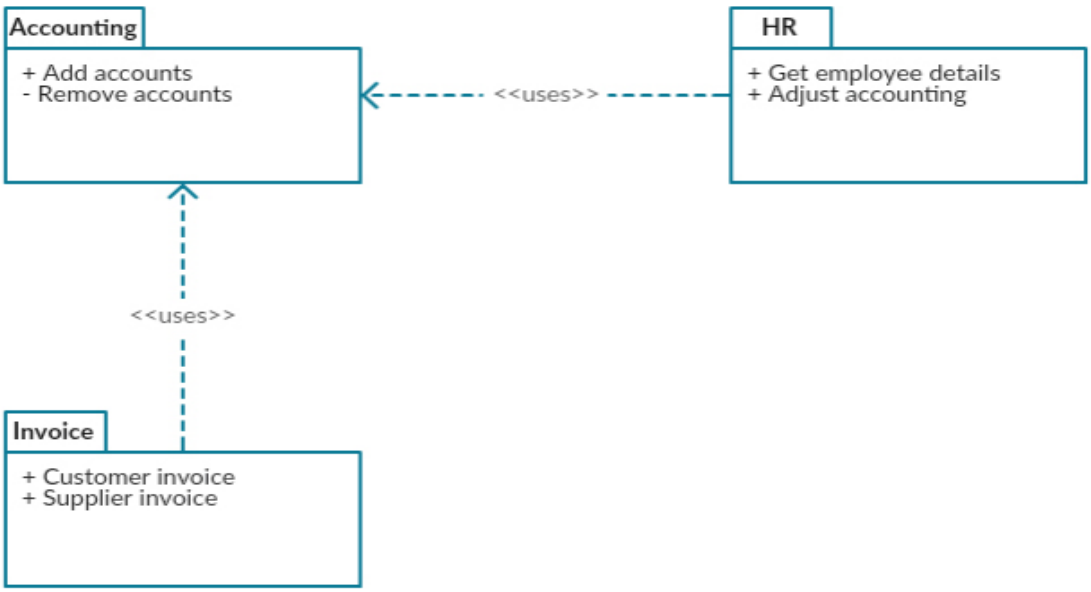


Figure of Object Diagram

Package Diagram

As the name suggests, a package diagram shows the dependencies between different packages in a system.

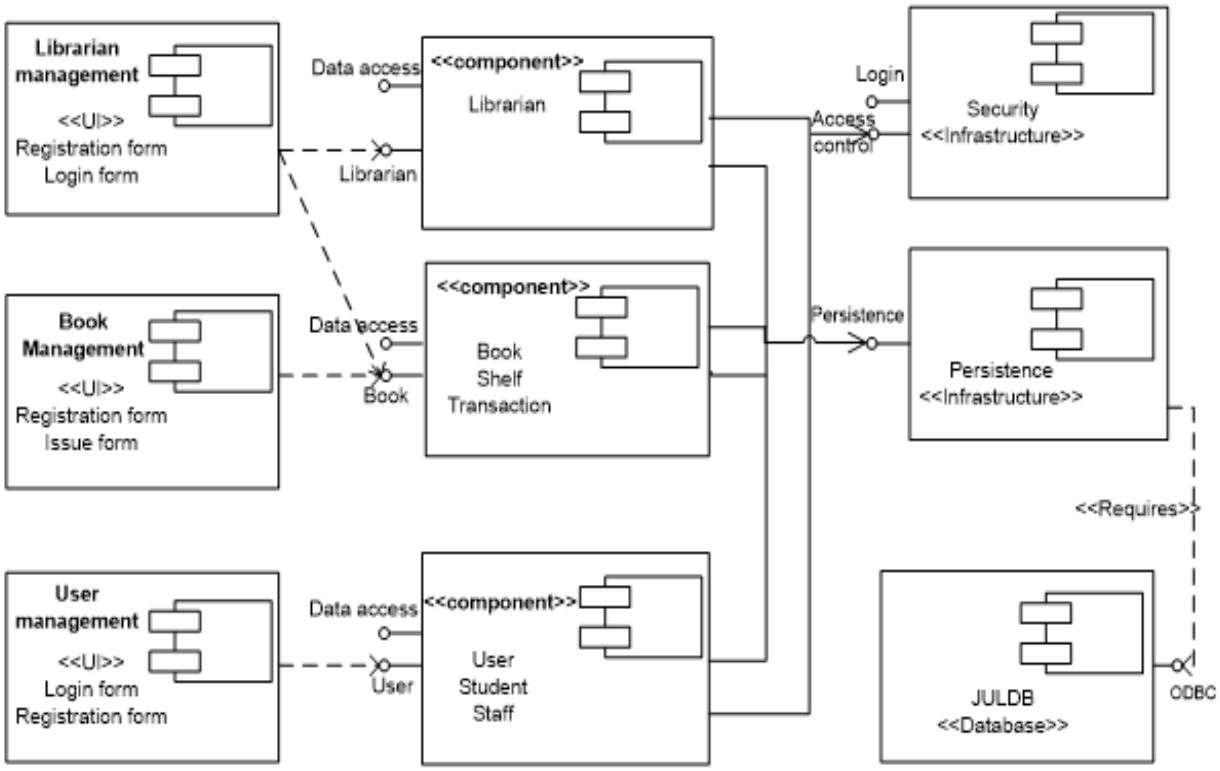


Component diagram

Component diagrams represent a set of components and their relationships. It is used to model physical aspects of a system. These components consist of classes, interfaces, or collaborations. Component diagrams represent the implementation view of a system. During the design phase, software artifacts (classes, interfaces, etc.) of a system are arranged in different groups depending upon their relationship. Now, these groups are known as components.

- These diagrams also show the externally visible behavior of the component by displaying the interfaces of the components.
- Components
- Interfaces
- Dependency relationship.
- Component diagrams
    - Models business and technical software architecture
    - Uses components defined in the composite structure diagrams, in particular their ports and interfaces

Component diagram for Library Management system



Deployment Diagrams

Deployment diagrams are a set of nodes and their relationships. These nodes are physical entities where the components are deployed. Used to visualize the topology of the physical components of a system where the software components are deployed. A deployment diagram shows processors, devices and connections. Each model contains a single deployment diagram which shows the connections between its processors and devices and its processes to processors.

- Deployment diagrams
  - Models the physical software architecture, including issues such as the hardware, the software installed on it and the middleware
  - Gives a static view of the run-time configuration of processing nodes and the components that run on those nodes

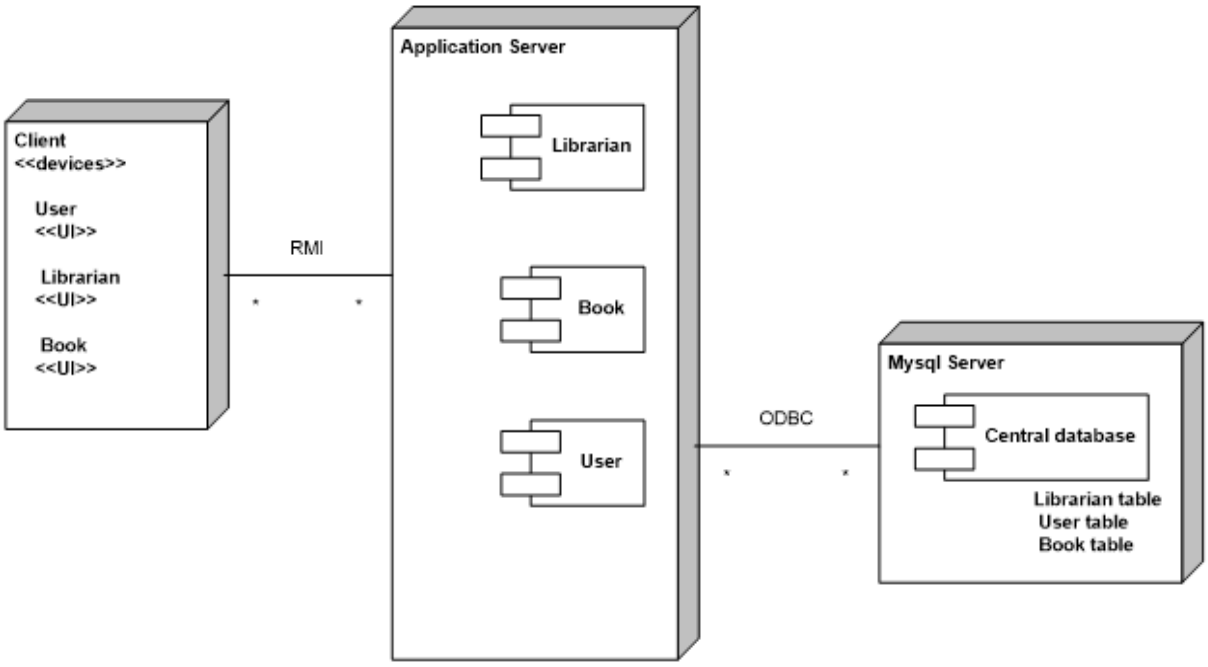


Figure: **Deployment Diagrams for Library Management System**

Behavioral Modelling

Behavioral diagrams basically capture the dynamic aspect of a system. It represents the interaction among the structural diagrams. Dynamic aspect can be further described as the changing/moving part of the system

- All the above show the dynamic sequence of flow in a system.
    - Use case diagrams
    - Sequence Diagram
    - Collaboration Diagram (communication)
    - State chart Diagram
    - Activity diagrams
    - Communication Diagram
    - Interaction Overview Diagram
    - Timing Diagram

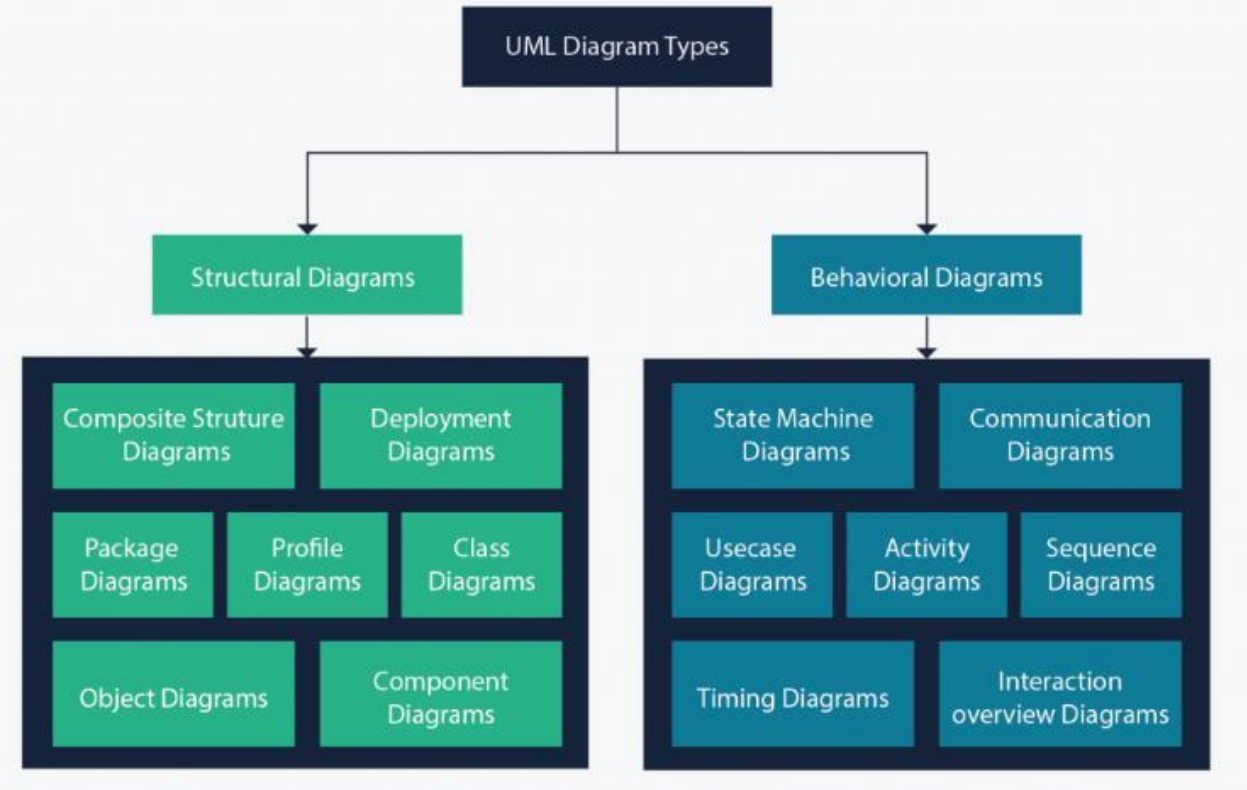


Figure of UML types

Use Case Diagram

Use case diagrams are a set of use cases, actors, and their relationships. They represent the use case view of a system.

- **Use case diagram designed under analysis**

- **For use case we analysis:**

  - Use case ID: example UC1: Login
  - Pre-condition
  - Post-condition
  - Main course of action
  - Alternative course of action
  - Include and Extend

Finally, it can be said component diagrams are used to visualize the implementation.

Include and Extend

- The relationship between Use cases can be reused and extended in two different fashions: **extends and include.**
- In the cases of **"include"** relationship, we define that one use case invokes the steps defined in another use case during the course of its own execution. Hence this defines a relationship that is similar to a relationship between two functions where one makes a call to the other function.
- The **"extends"** relationship is kind of a generalization-specialization relationship. In this case a special instance of an already existing use case is created. The new use case inherits all the properties of the existing use case, including its actors.

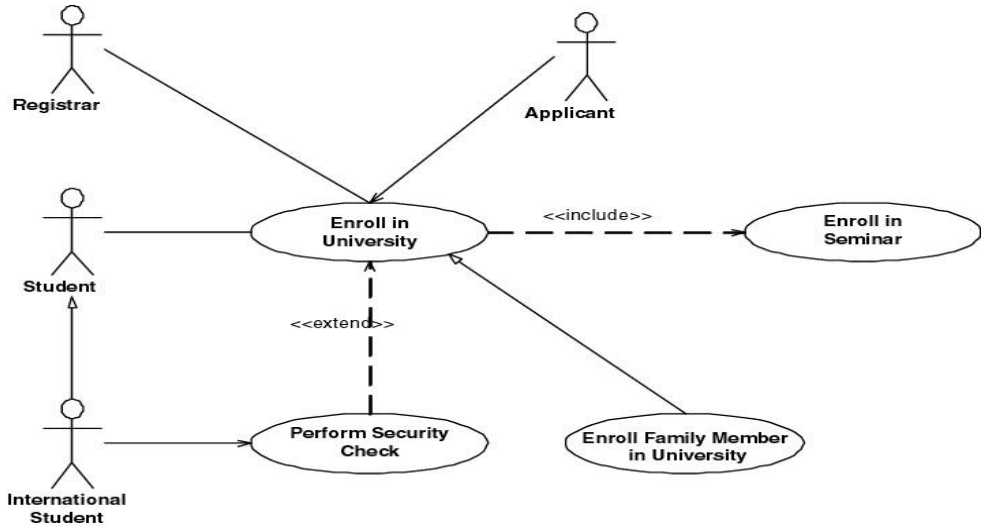
Extend :<<extend>>

- When the extending use case activity sequence is completed, the base use case continues.
- An extending use case is, effectively, an alternate course of the base use case.
- Ex. *Enroll in University* is the base use case and *Perform Security Check* is the extending use case. Step 7

Include: <<include>>

- formerly known as a uses relationship in UML v1.2 and earlier
- is a generalization relationship denoting the inclusion of the behavior described by another use case
- is the invocation of a use case by another one
- Used whenever one use case needs the behavior of another

- Ex. If a student is enrolled in a university, he/she can enroll in a seminar. Step 11



Inheritance between use cases

- Use cases can inherit from other use case. It is not as common as the use of either extend or include associations, but it is still possible
- The inheriting use case would completely replace one or more of the courses of action of the inherited use case
- **Ex. that *Enroll Family Member in University* inherits from the *Enroll in University* use case** (in this case, to reflect that new business rules are applied when the family member of a professor is enrolling at the university)
- It should have a name, description, and identifier, and it should indicate from which use case it inherits (If something is not replaced, then leave that section blank, assuming it is inherited from the parent use case)

Inheritance between actors

- an actor on a use case diagram can inherit from another actor
- **Ex. *International Student* actor inherits from *Student*.**

- International student is subjected to different rules and policies (pay more and undergo a security check during registration process)

Example:

- **Login**
- ID: UC1
- Actor: librarian
- Pre-condition: the librarian/user should have their user name and password to login.
- Post-condition: the main page of the system should display on the screen.
- Include: user database.

➢ **Main course of action: step by step**

1. An authorized librarian/user want to login to the system.
2. The librarian/user click on login form.
3. The system asks user name and password to the login to the system.
4. Then enter his/her username and password to login form and click on login.
5. The system checks and authenticates the librarian/user.
6. The database accepts the username and password of the librarian/user.
7. Then the main page will be displayed to the users.

Best practices in writing Alternate Courses of Action

- **Briefly describe the condition:** description of the condition that must be met to invoke the alternate course
- **Indicate that the alternate course exists within the basic course of action. *(Specify the alternate course name)***
- **Set a consistent step numbering strategy*:*** each step starts with the letter of the ***alternate course***, followed by the number of the step in the ***basic course of the use case*** it replaces
- **You need an identification scheme:** identify the first alternate course as *A*, the second as *B*, and so on. Also, notice the numbering scheme for the steps of the alternate course.

- **End the alternate course:** indicate either that the use case ends or the use case continues at another step

➢ **Alternative course of action**

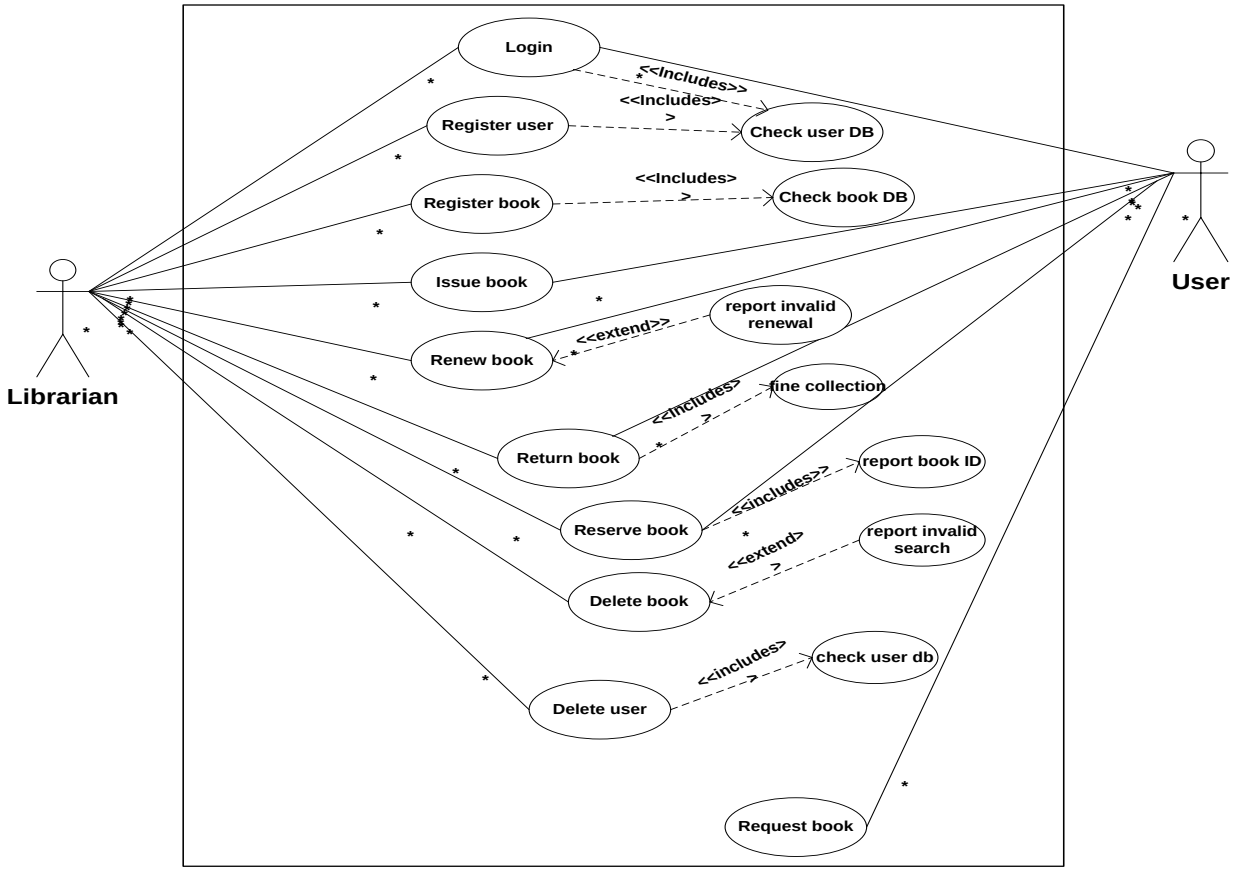
A.5 If the user type wrong password or username.

A.6.The login controller initiates login.

A.7.The database authenticate the librarian/user.

A.8.The system display login form to the librarian/user continuously.

Software
Use case diagram



Interaction Diagram

Interaction diagram is part of behavioral diagram. Interaction is basically a message exchange between two UML components. The following diagram represents different notations used in an interaction.

Interaction overview diagrams are very similar to activity diagrams. While activity diagrams show a sequence of processes, Interaction overview diagrams show a sequence of interaction diagrams.

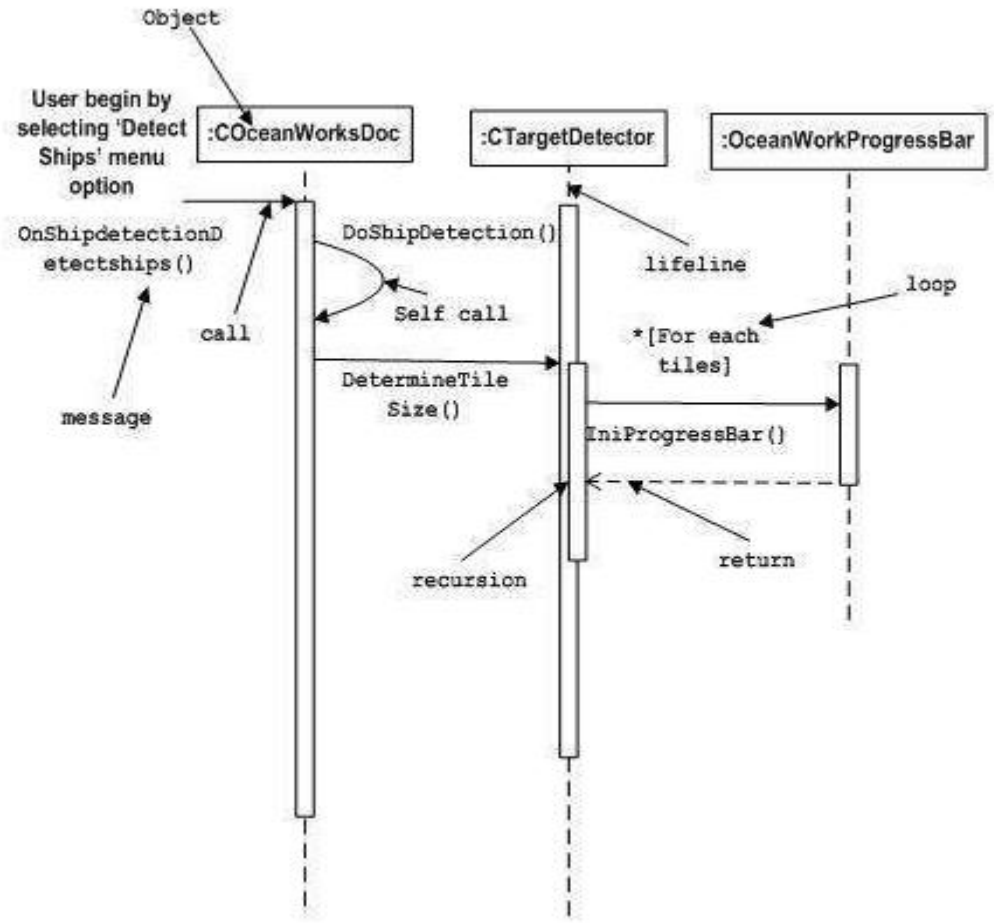
- Interactions can be of two types −

- **Sequential** (Represented by sequence diagram)
- **Collaborative** (Represented by collaboration diagram)
- ***Sequence Diagrams:*** A sequence diagram is a graphical view of a scenario that shows object interaction in **a time based sequence**, what happens first, what happens next.
- Sequence diagrams establish the roles of objects and help provide essential information to determine class responsibilities and interfaces.
- A sequence diagrams in design phases are a refined diagrams with additional details of activities.

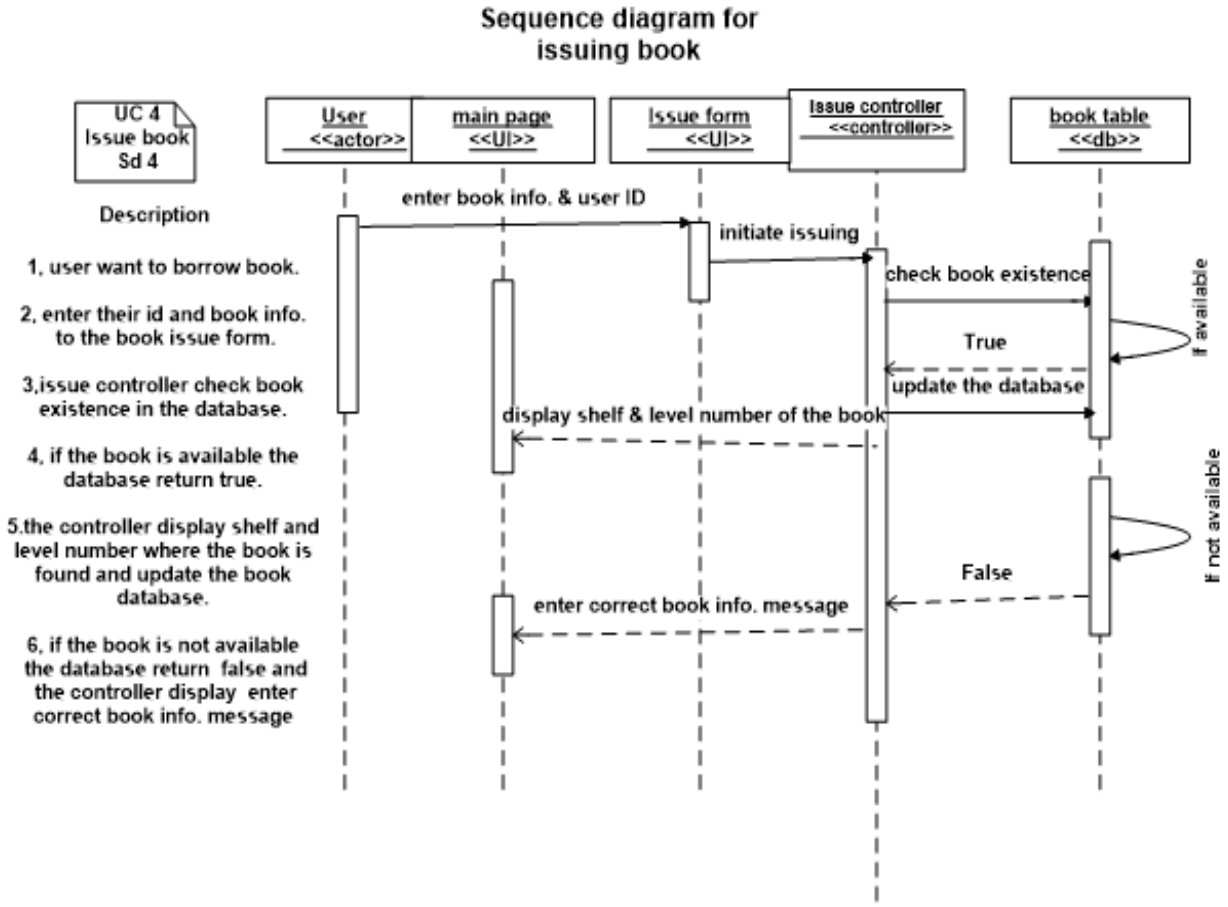
Rules in drawing Sequence Diagram

- the **horizontal arrows** shows the **message** : the first message starts in the top left corner; the next message appears just below that one
- The **boxes across the top of the diagram** represent **classifiers** or their instances; typically use cases, objects, classes, or actors
    - **Objects have labels** in the standard UML format ***name: ClassName*** (name is optional, if object has no name "Anonymous") and ***object labels are underlined***
    - **Classes have labels** in the format ***ClassName***, and
    - **Actors have names** in the format ***Actor Name***.
- The **dashed lines hanging from the boxes** are called **object lifelines**, representing the life span of the object during the scenario being modeled.

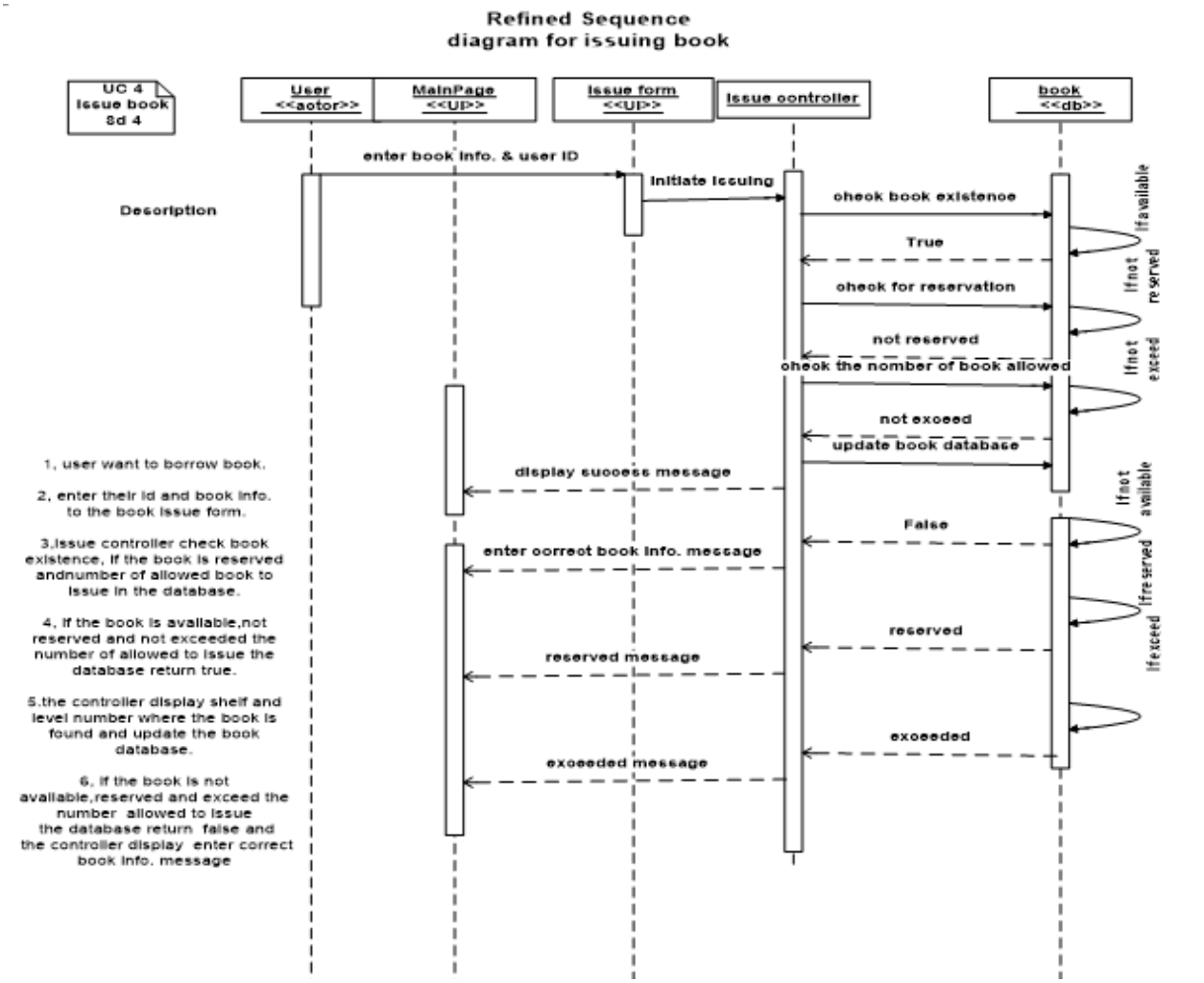
- **The long, thin boxes on the lifelines** are **activation boxes**, also called **method-invocation boxes**, which indicate processing is being performed by the target object/class to fulfill a message.



Analysis sequence diagram for library management system

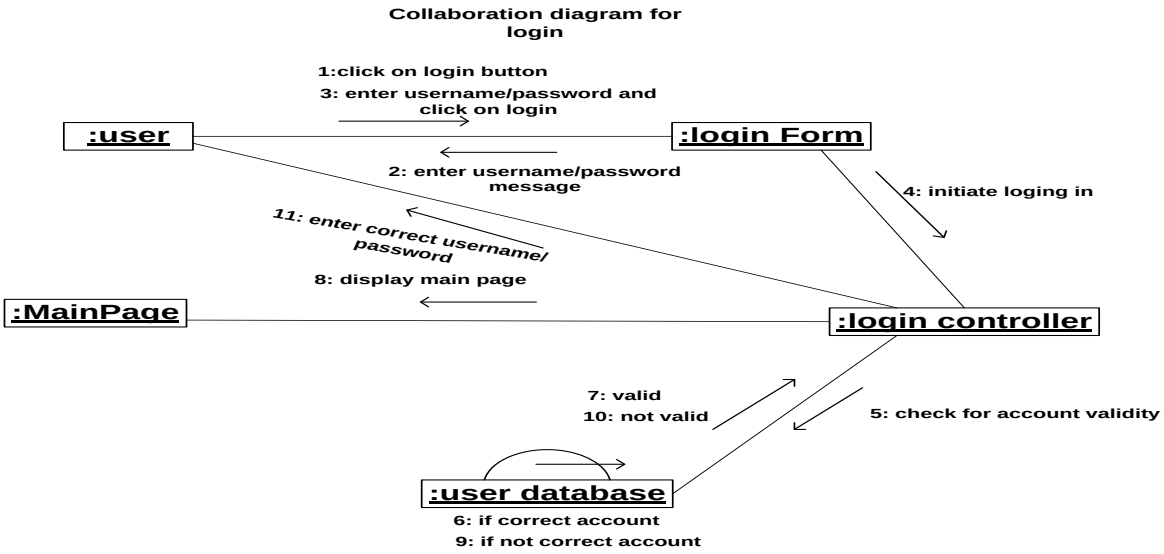


Design sequence diagram for library management system



Collaboration diagram

It represents the structural organization of a system and the messages sent/received. Structural organization consists of objects, links and messages. Order of messages that implement an operation or a transaction. They can also contain simple class instances and class utility instances. The purpose of collaboration diagram is similar to sequence diagram. However, the specific purpose of collaboration diagram is to visualize the organization of objects and their interaction.



Timing Diagram

Timing diagrams are very similar to sequence diagrams. They represent the behavior of objects in a given time frame. If it's only one object, the diagram is straightforward. But, if there is more than one object is involved, a Timing diagram is used to show interactions between objects during that time frame.

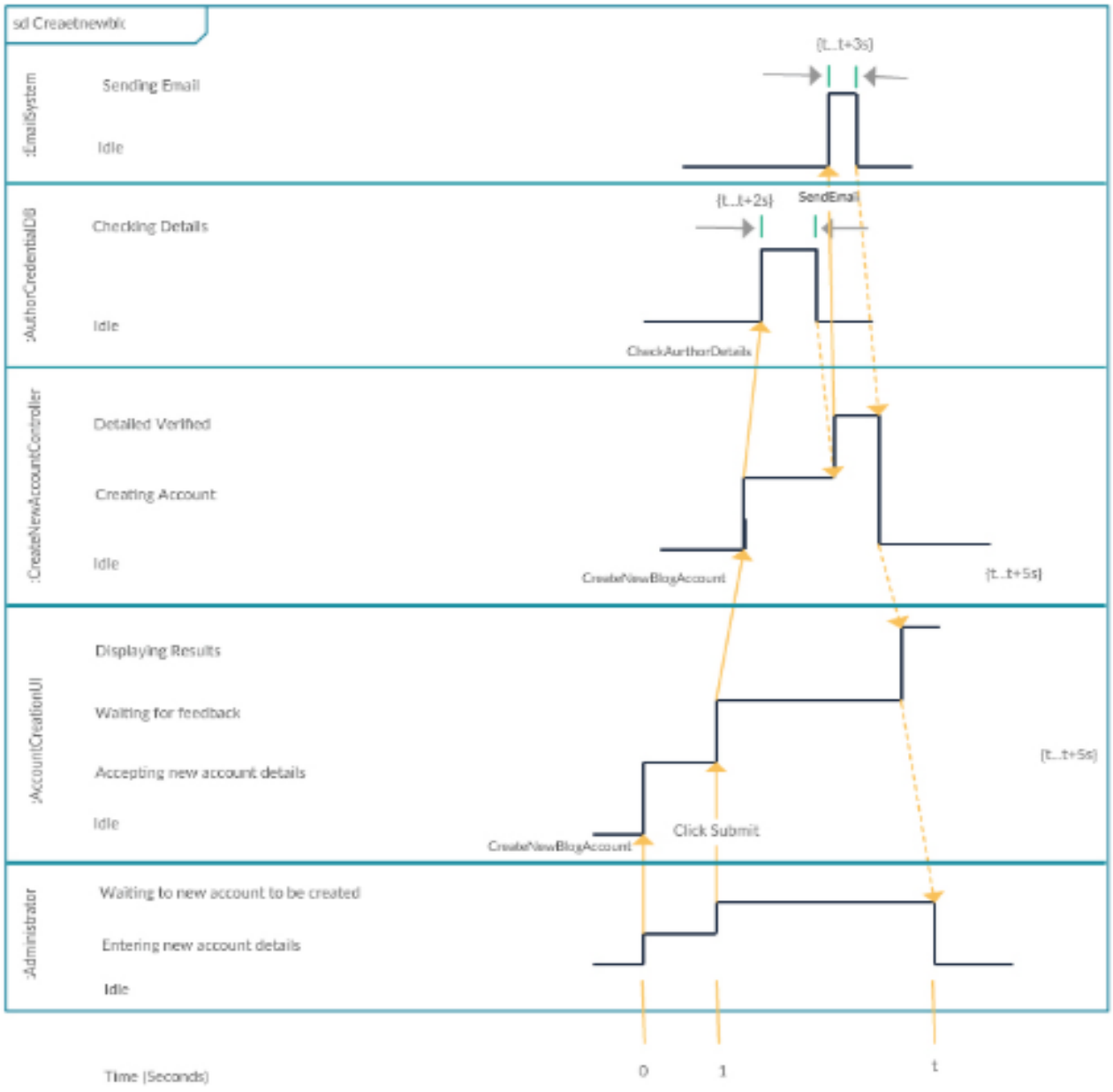


Figure of Timing Diagram

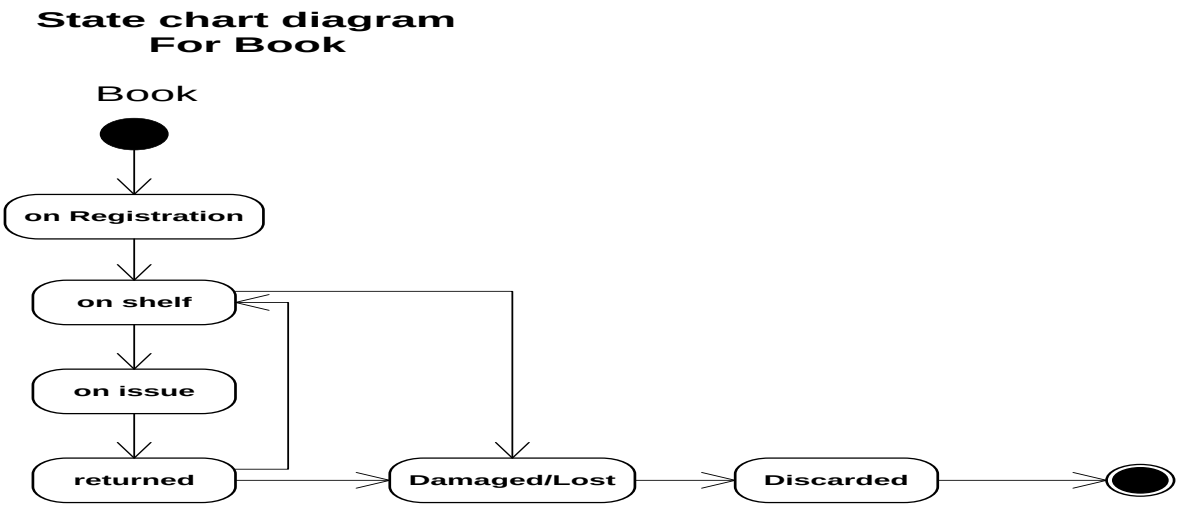
State Machine Diagram/Notation

State machine describes the different states of a component in its life cycle. The notations are described in the following diagram. State machine is used to describe different states of a system component. The state can be active, idle, or any other depending upon the situation

- *State chart diagram*: state chart diagrams model the dynamic behavior of individual classes or any other kind of object. They show the sequence of states that an object goes through the events that cause a transition from one state to another and the actions that result from a state change. A state chart diagram is typically used to model the discrete stages of an objects lifetime.

Essential elements of state chart diagram

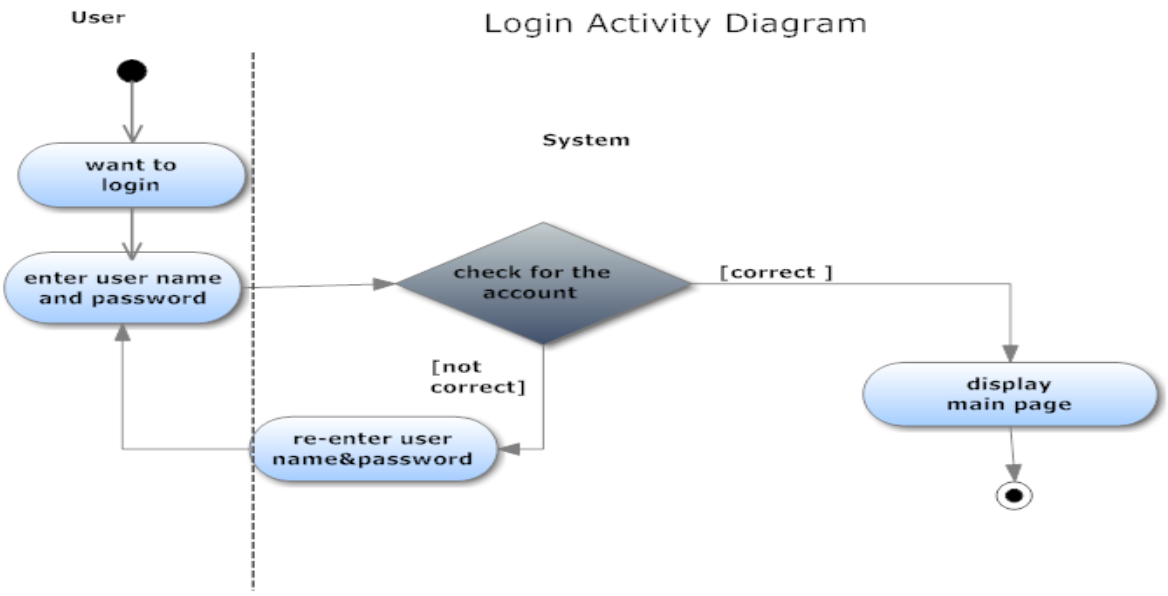
- ***State:*** - A state represents a condition or situation during the life of an object during which it satisfies some condition or waits for an event. Each state represents a cumulative history of its behavior. States can be shared between state machines. Transitions cannot be shared
- ***Start state:*** - A start state (also called an "initial state") explicitly shows the beginning of the execution of the state machine on the state chart diagram or beginning of the workflow on an activity diagram. Normally, one outgoing transition can be placed from the start state.
- ***End state: -*** An end state represents a final or terminal state on an activity or state chart diagram. Transitions can only occur into an end state. The end state icon is a filled circle inside a slightly larger unfilled circle that may contain the name (End process).
- ***State transition: -*** A state transition indicates that an action in the source state will perform certain specified actions and enter the destination state when a specified event occurs or when certain conditions are satisfied.



Activity Diagram

Activity diagram describes the flow of control in a system. It consists of activities and links. The flow can be sequential, concurrent, or branched. Activities are nothing but the functions of a system. Numbers of activity diagrams are prepared to capture the entire flow in a system. Activity diagrams are used to visualize the flow of controls in a system. This is prepared to have an idea of how the system will work when executed.

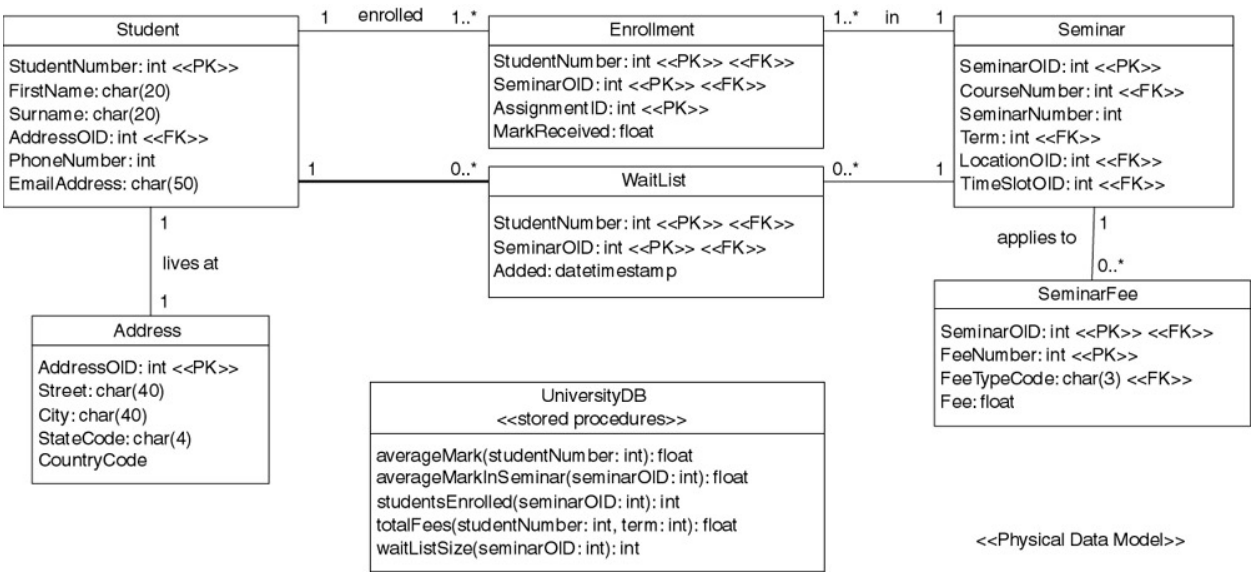
- **Basic notations of Activity Diagram**
  - **Initial node:** filled circle is the starting point of the diagram
  - **final node.** The filled circle with a border is the ending point (0 , >0)
  - **Activity.** The rounded rectangles represent activities that occur (be physical, such as *Inspect Forms*, or electronic, such as *Display Create Student Screen)*
  - **Flow/edge.** The arrows on the diagram
  - **Condition.** Text such as [*Incorrect Form*] on a flow
  - **Decision.** A diamond with one flow entering and several leaving



Physical Data Modelling

Data modeling is the act of exploring data-oriented structures. Physical data modeling is similar to design class modeling, the goal being to design the internal schema of a database, depicting the data tables, the data columns of those tables, and the relationships between the tables.

Figure below presents a partial PDM for the university—you know that it is not complete by the fact that the *Seminar* table includes foreign keys to tables not shown, and quite frankly it is obvious that many domain concepts such as course and professor are clearly not modeled. All but one of the boxes represents tables, the one exception being *UniversityDB*, which lists the stored procedures implemented within the database. Relationships between tables are modeled using standard UML notation; although not shown in the example it would be reasonable to model composition and inheritance relationships between tables. Relationships are implemented via the use of keys.



- Purpose of PDM diagrams:
  - Design the internal schema of a database,
  - Depicting the data tables,
  - The data columns of those tables, and
  - The relationships between the tables. (implemented using keys)

In designing physical data modelling it:

- **Identify tables:** Tables are the database equivalent of classes; data are stored in physical tables. As you can see in Figure below the university has a *Student* table to store student data, a *Course* table to store course data, and so on.

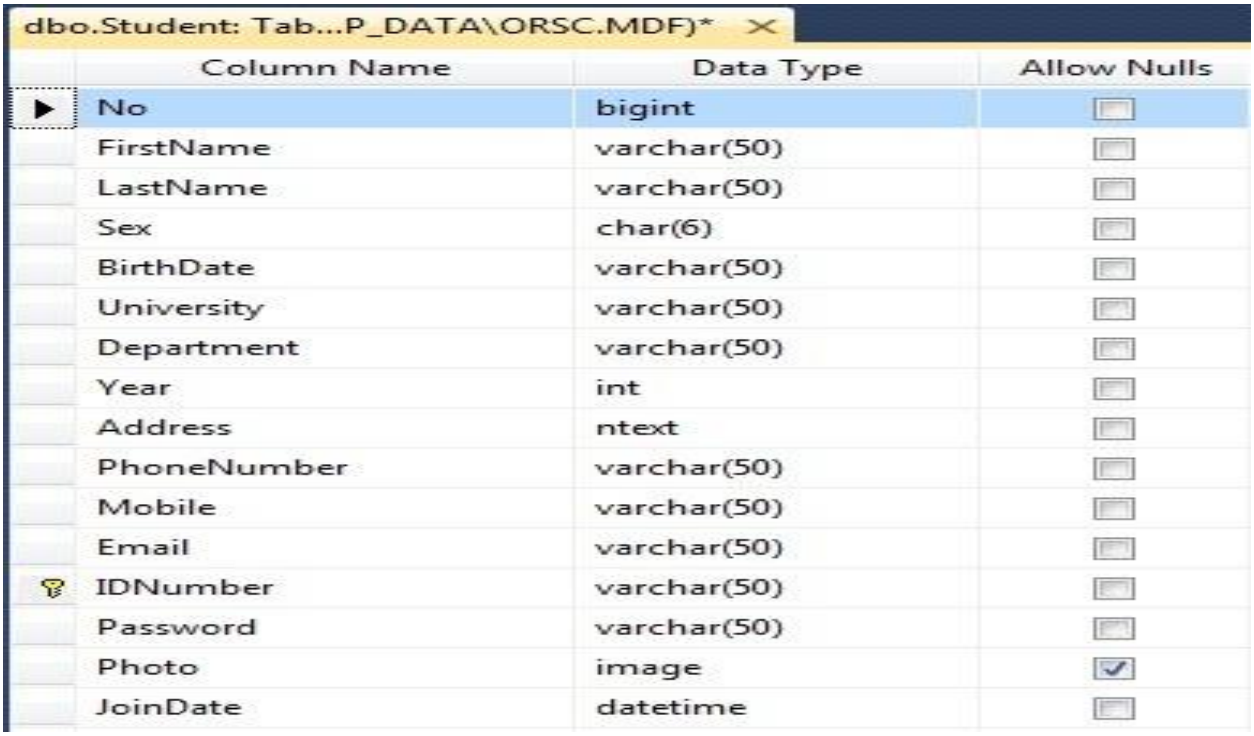
Table of: Data Normalization Rules

Level	**Rule**
First normal form (1NF)	An entity type is in 1NF when it contains no repeating groups of data.
Second normal form (2NF)	An entity type is in 2NF when it is in 1NF and when all of its nonkey attributes are fully dependent on its primary key.
Third normal form (3NF)	An entity type is in 3NF when it is in 2NF and when all of its attributes are directly dependent on the primary key.

- **Normalize tables.** Data normalization is a process in which data attributes within a data model are organized to increase the cohesion of tables and to reduce the coupling between

tables. The fundamental goal is to ensure that data are stored in one and only one place/table. This is an important consideration for application developers because it is incredibly difficult to store objects in a relational database if a data attribute is stored in several places. The first three rules of data normalization are summarized in table given above.

- **Identify columns.** A column is the database equivalent of an attribute, and each table will have one or more columns. For example, in figure the *Student* table has attributes such as *FirstName* and *StudentNumber*. Unlike attributes in classes, which can be either primitive types or other objects, a column may only be a primitive type such as a *char* (a string), an *int* (integer), or a *float*.
- **Identify stored procedures.** A stored procedure is conceptually similar to a global method implemented by the database. In the figure you see that stored procedures such as *averageMark()* and *studentsEnrolled()* are modeled as operations of the class *UniversityDB*. These stored procedures implement code that work with data stored in the database; in this case they calculate the average mark of a student and count the number of students enrolled in a given seminar respectively. Although some of these stored procedures clearly act on data contained in a single table they are not modeled as part of the table (along the lines of methods being part of classes). Instead, because stored procedures are a part of the overall database and not a single table, they are modeled as part of a class with the name of the database.
- **Identify relationships.** There are relationships between tables just like there are relationships between classes.
- **Assign keys.** A key is one or more data attributes that uniquely identify a row in a table. A key that is two or more attributes is called a composite key. A primary key is the preferred key for an entity type, whereas an alternate key (also known as a secondary key) is an alternative way to access rows within a table. In a physical database a key would be formed of one or more table columns whose value(s) uniquely identifies a row within a relational table. In the next figure primary keys are indicated using the <<PK>> stereotype and foreign keys via <<FK>>.
- **Finally show all the tables with their fields/attributes description**



Finally show all the tables with their fields/attributes description

Graphical User Interface

- Help the user to navigate through the system and enter data to the database, access and retrieve data from the database.
- Design class diagram will be converted to tables and tables will be converted to GUI
- Every table that is designed in the physical data model should have corresponding graphical user interface.

Properly designed GUI have the following benefits:

A fundamental reality of application development is that the user interface is the system to the users. What users want is for developers to build applications that meet their needs and that are easy to use. Points out that the reality is that a good user interface allows people who understand the problem domain to work with the application without having to read the manuals or receive training.

After doing your physical data modeling you should design Graphical User Interface (GUI) that used the user to navigate the system and enter data to the database, access and retrieve data from the database. Every table that is designed in the physical data model should have corresponding graphical user interface. Design class diagram will be converted to tables and tables will be converted to GUI

Interface design is important for several reasons. First of all the more intuitive the user interface the easier it is to use. The better the user interface the easier it is to train people to use it, reducing your training costs. The better your user interface the less help people will need to use it, reducing your support costs. The better your user interface the more your users will like to use it, increasing their satisfaction with the work that you have done.

User Interface Design Tips and Techniques

In this section we will cover a series of user interface design tips that will help you to improve the object oriented interfaces that you create.

1. **Consistency.** The most important thing that you can possibly do is make sure that your user interface works consistently. If you can double-click on items in one list and have something happen then you should be able to double-click on items in any other list and have the same sort of thing happen.
2. **Set standards and stick to them**. The only way that you'll be able to ensure consistency within your application is to set design standards and then stick to them. The best approach is to adopt an industry standard and then fill any missing guidelines that are specific to your needs.
3. **Explain the rules**. Your users need to know how to work with the application that you built for them.
4. **Navigation between screens is important**. If it is difficult to get from one screen to another then your users will quickly become frustrated and give up.
5. **Word your messages and labels appropriately**. The text that you display on your screens is a primary source of information for your users. If your text is worded poorly then your interface will be perceived poorly by your users.

6. **Understand your widgets**. You should use the right widget for the right task, helping to increase the consistency in your application and probably making it easier to build the application in the first place.
7. **Look at other applications with a grain of salt**. Unless you know that another application follows the user-interface standards and guidelines of your organization, you must not assume that the application is doing things right.
8. **Use color appropriately**. Color should be used sparingly in your applications, and if you do use it you must also use a secondary indicator.
9. **Follow the contrast rule**. If you are going to use color in your application you need to ensure that your screens are still readable.
10. **Use fonts.** Use fonts that are easy to read, such as serif fonts like Times Roman. Furthermore, use your fonts consistently and sparingly.
11. **Gray things out, do not remove them**. You often find that at certain times it is not applicable to give your users access to all the functionality of an application.
12. **Alignment of fields**. When a screen has more than one editing field you want to organize the fields in a way that is both visually appealing and efficient.
13. **Justify data appropriately**. For columns of data it is common practice to right justify integers, decimal align floating point numbers, and left justifies strings.
14. **Do not create busy screens**. Crowded screens are difficult to understand and hence are difficult to use.
15. **Group things on the screen effectively**. Items that are logically connected should be grouped together on the screen to communicate that they are connected, whereas items that have nothing to do with each other should be separated. You can use whitespace between collections of items to group them and/or you can put boxes around them to accomplish the same thing.

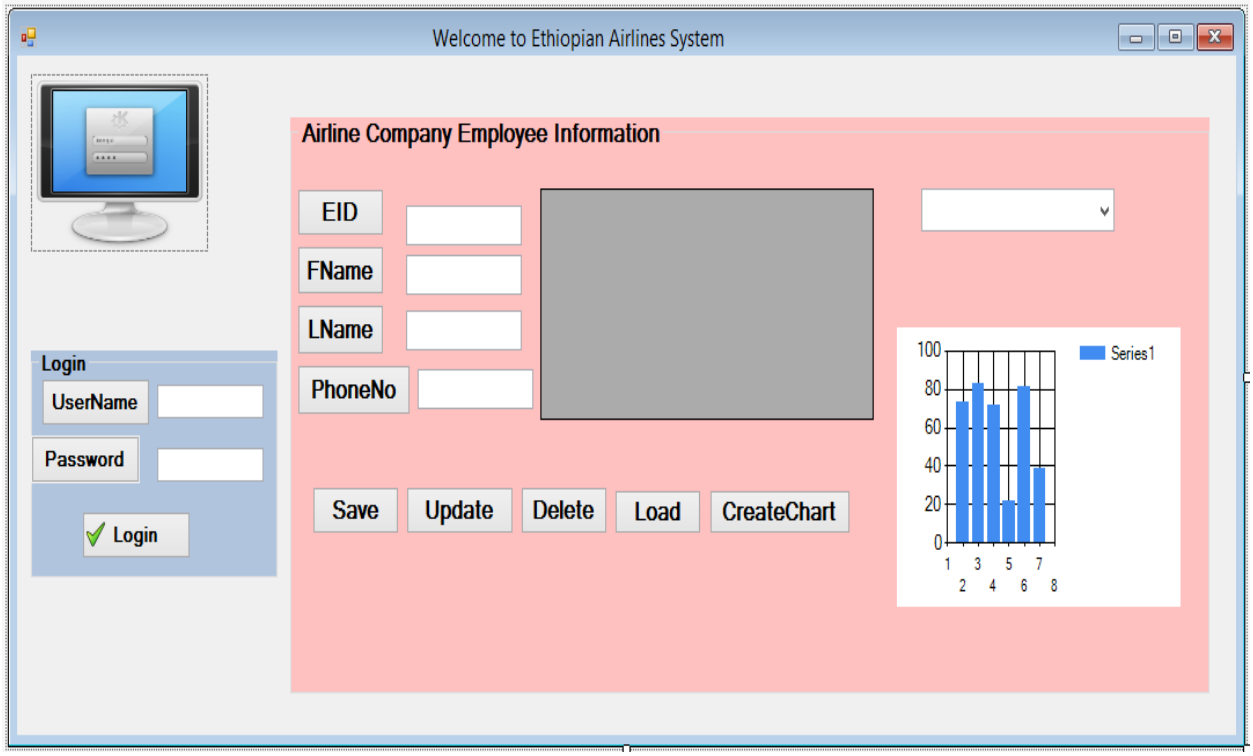


Figure of Graphical User Interface