SOFTWARE REQUIREMENT SPECIFICATION

A **Software requirements specification** (SRS), a requirements specification for a software system, is a complete description of the behavior of a system to be developed and may include a set of use cases that describe interactions the users will have with the software

Software Requirements is a field within Software Engineering that deals with establishing the needs of stakeholders that are to be solved by software. The IEEE Standard Glossary of Software Engineering Technology defines a software requirement as:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
3. A documented representation of a condition or capability as in 1 or 2.

IEEE Standard SRS Template

1. Introduction

1.1. Purpose

1.2. Scope

1.3. Definitions, acronyms & abbreviations

1.4. References

1.5. Overview

2. Overall description

2.1. Product perspective

2.1.1. System interfaces

2.1.2. User interfaces

2.1.3. Hardware interfaces

2.1.4. Software interfaces

2.1.5. Communications interfaces

2.1.6. Memory constraints

2.1.7. Operations

2.1.8. Site adaptation requirements

2.2. Product functions

2.3. User characteristics

1. Introduction

The following subsections of the SRS should provide an overview of the entire SRS.

1.1 Purpose

Identify the purpose of this SRS and its intended audience.

1.2 Scope.

(1) Identify the software product(s) to be produced by name

(2) Explain what the software product(s) will, and, if necessary, will not do

(3) Describe the application of the software being specified. As a portion of this, it should:

(a) Describe the relevant benefits, objectives, and goals as precisely as possible

(b) Be consistent with similar statements in higher-level specifications if they exist.

1.3 Definitions, Acronyms, and Abbreviations

Provide the definitions of all terms, acronyms, and abbreviations required to properly interpret the SRS. This information may be provided by reference to an appendix or other document(s).

1.4 References

In this subsection:

(1) Provide a complete list of all documents referenced elsewhere in the SRS.

(2) Identify each document by title, report number (if applicable), date, and publishing organization.

(3) Specify the sources from which the references can be obtained.

1.5 Overview

Describe the rest of the SRS and how it is organized.

2. THE GENERAL DESCRIPTION

Describe the general factors that affect the product and its requirements. This section usually consists of the five subsections that follow. This section does not state specific requirements; each of its subsections makes those requirements easier to understand; they do not specify design or express specific requirements.

2.1 Product Perspective

This subsection of the SRS relates the product to other products or projects.

(1) If the product is independent and totally self-contained, it should be stated here.

(2) If the SRS defines a product that is a component of a larger system or project:

- (a) Describe the functions of each component of the larger system or project, and identify interfaces
- (b) Identify the principal external interfaces of this software product (not a detailed description)
- (c) Describe the computer hardware and peripheral equipment to be used (overview only)

A block diagram showing the major components of the larger system or project, interconnections, and external interfaces can be very helpful.

2.2 Product Functions

Provide a summary of the functions that the software will perform. Sometimes the function summary that is necessary for this part can be taken directly from the section of the higher-level specification (if one exists) that allocates particular functions to the software product. The functions should be organized in a way that makes the list of functions understandable to the customer or to anyone else reading the document for the first time. Block diagrams showing the different functions and their relationships can be helpful. Such a diagram is not a requirement on the design of a product itself; it is simply an effective explanatory tool.

2.3 User Characteristics

Describe those general characteristics of the eventual users of the product that will affect the specific requirements.

Many people interact with a system during the operation and maintenance phase of the software life cycle. Some of these people are users, operators, and maintenance and systems personnel. Certain characteristics of these people, such as educational level, experience, and technical expertise impose important constraints on the system's operating environment.

3. BOOK BANK MANAGEMENT SYSTEM

3.1 AIM:

The aim of this project is to create book managing software to aid the workflow of the book bank management system.

3.2 Scope:

The book bank holds an online interface with its member for maintaining all kinds of transaction details. Each member is provided with a unique user id at the time of registering as a member.

3.3 OVERVIEW:

Typical users, such as students who want to use the software for various book related activities.

Advanced / professional users, such as teachers, book authors, researchers, etc..

Programmers who are interested in working in working on the project by further developing or fixing existing bugs.

3.4 REQUIREMENTS:

A) **FUNCTIONAL REQUIREMENTS:**

A functional requirements defines a function of a software system on its component. A function is described as a set of input, the behavior and output.

I. A main menu including a brief help section.
II. Login
III. Displaying Details
IV. Maintain and update book details.
V. Logout.

1) **REQUIREMENTS:**
The system should have the requirements of the project. The developer should prepare the requirements of the project. They should prepare the requirements which are need for software.

2) **ANALYSIS:**
Analyze the requirements whether it provides proper operations / output and performs the task.

3) **DESIGN:**
Project manager should design the layout of the project before going to implement time allocation, cost allocation, cost allocation and staff allocation will come under design process.

4) **IMPLEMENT:**

After encompassing all the diagrams, we have to generate code for each and every diagram i.e., form use case to development.

5) **TESTING:**
After implementing the diagram with domain language, we have to test the particular projects.

6) **MAINTAINENCE:**
The system should be easily updated. The system should utilize the interchangeable plugins software developed should maintain the cost and time schedule of the project.

B) **NON-FUNCTIONAL REQUIREMENTS:**

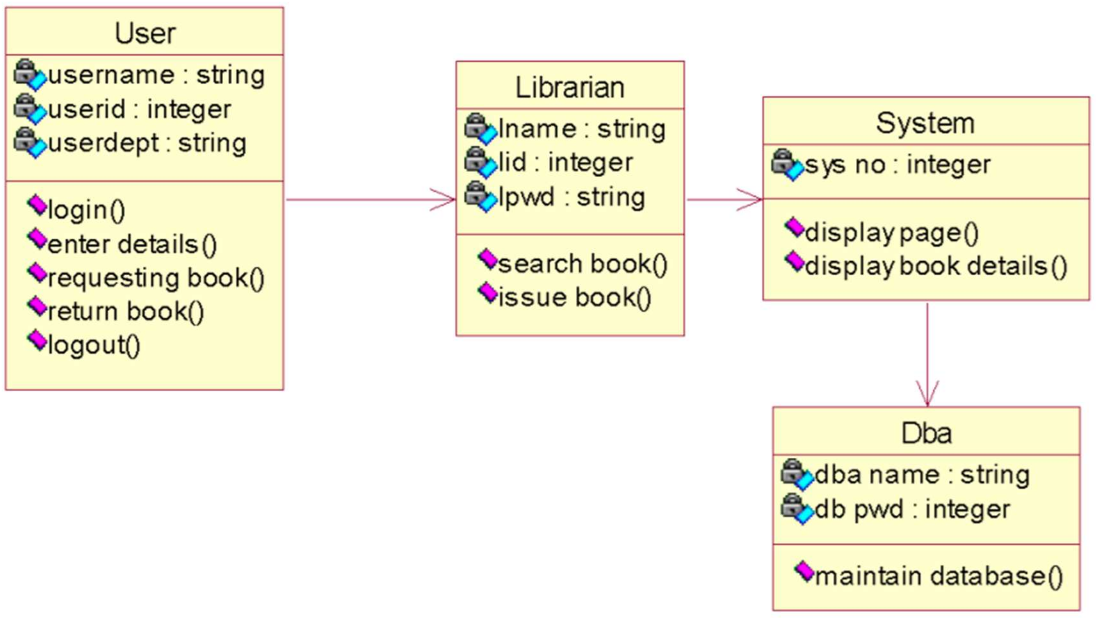
Nonfunctional requirements define the needs in terms if performance, logical database requirements, design constraints standard compliance, reliability, availability, security, maintainability and portability.

i. **PERFORMANCE REQUIREMENTS:**
Performance requirements decline acceptable response times for system functionality.
The total time for user interface screen will take no longer than two seconds.
The login information shall be verified within the seconds.
Queries shall result within five seconds.

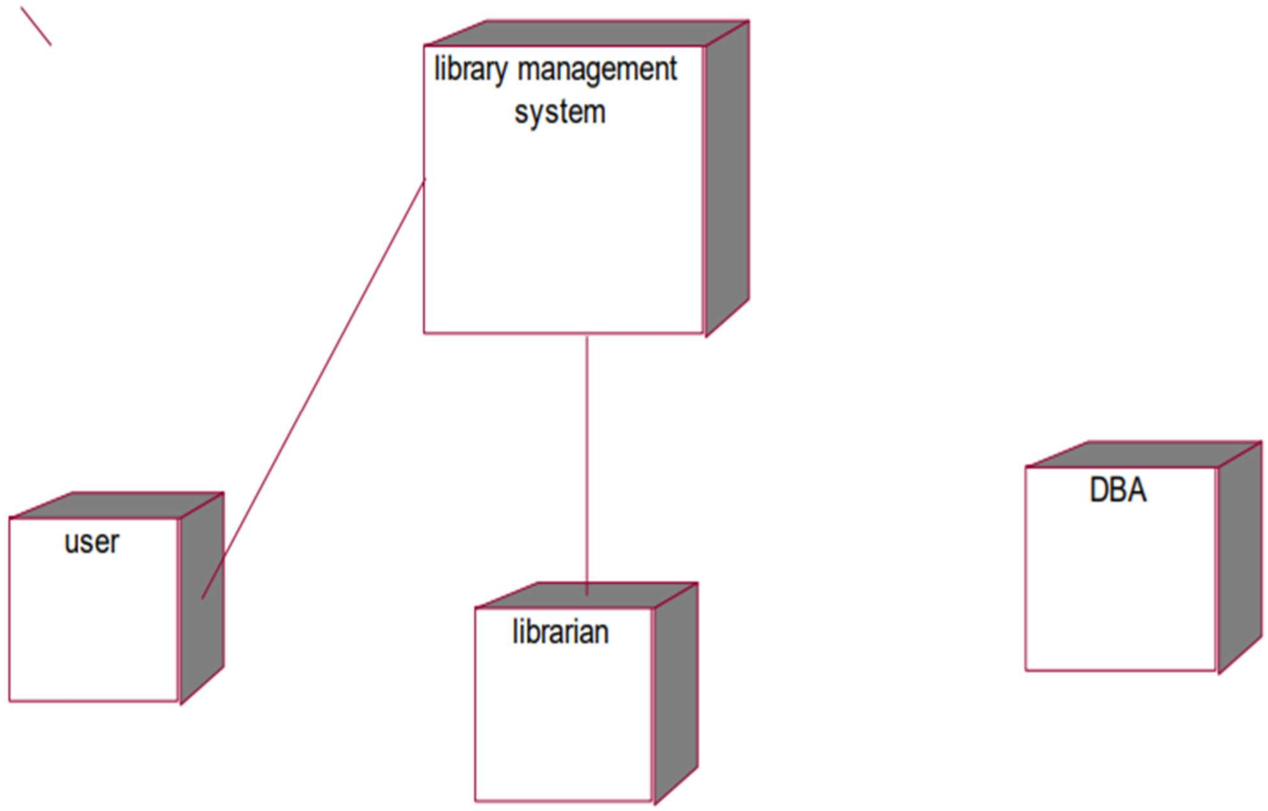
ii. **SAFTEY REQUIREMENTS:**
The details should be made available in the database and must be updated every time a book is used or returned or some kind of payment place to prevent errors.

iii. **SECURITY REQUIREMENTS:**
The members can only access certain details from the database. Hershel should not be able to modify the database nor has any of its information corrupted. Only the DBA must be bestowed with the privilege of handling any kind of modification.

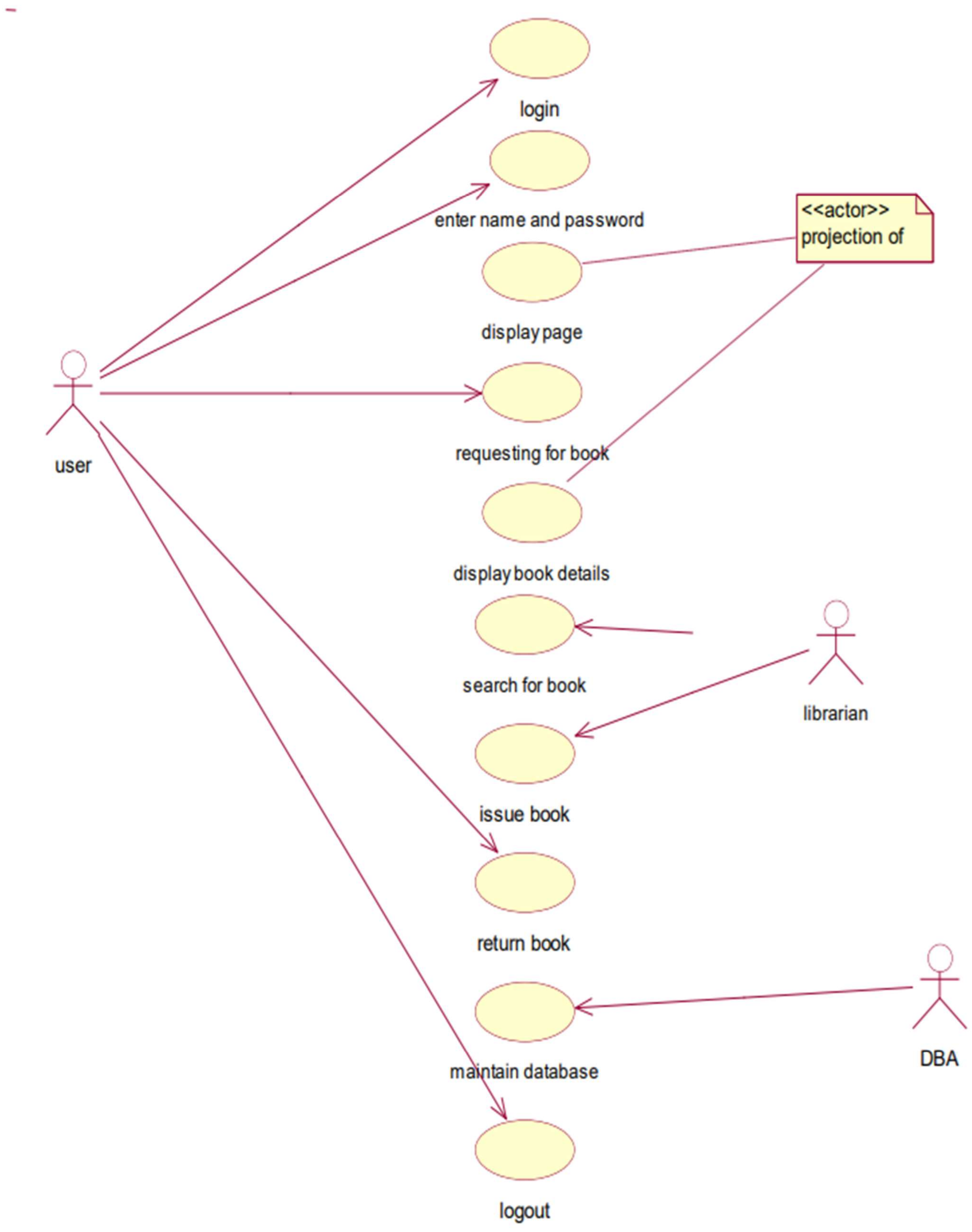
CLASS DIAGRAM:



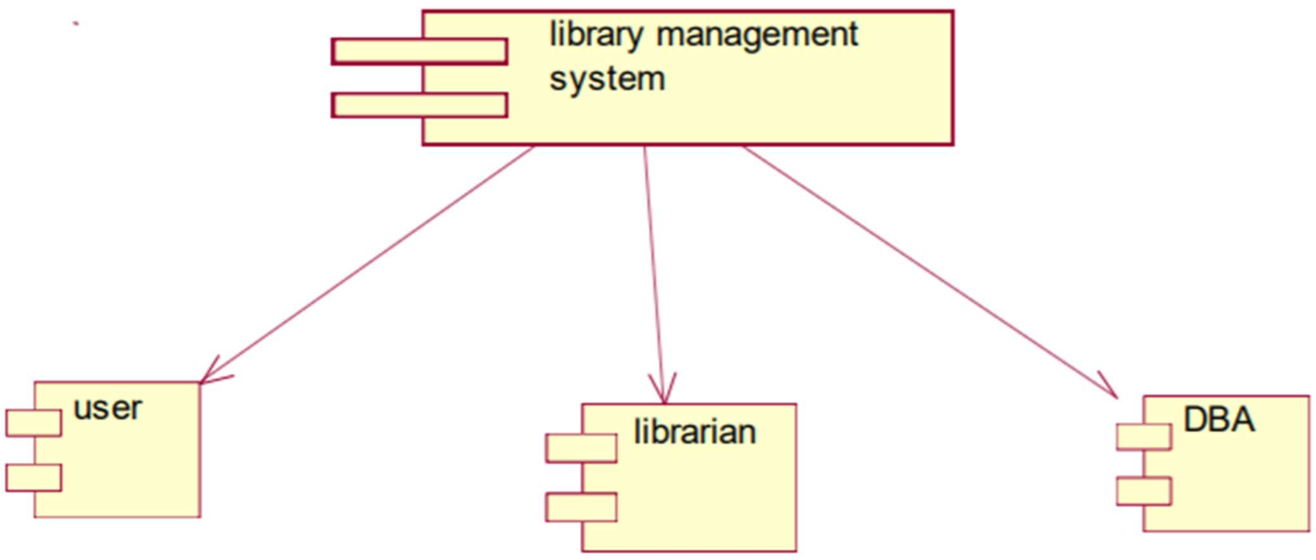
DEPLOYMENT DIAGRAM：



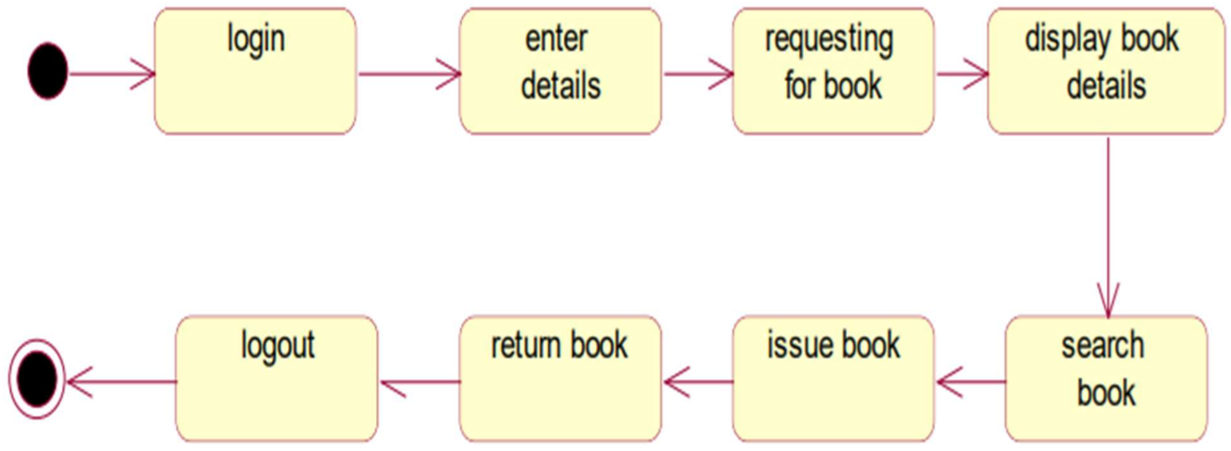
USE CASE DIAGRAM :



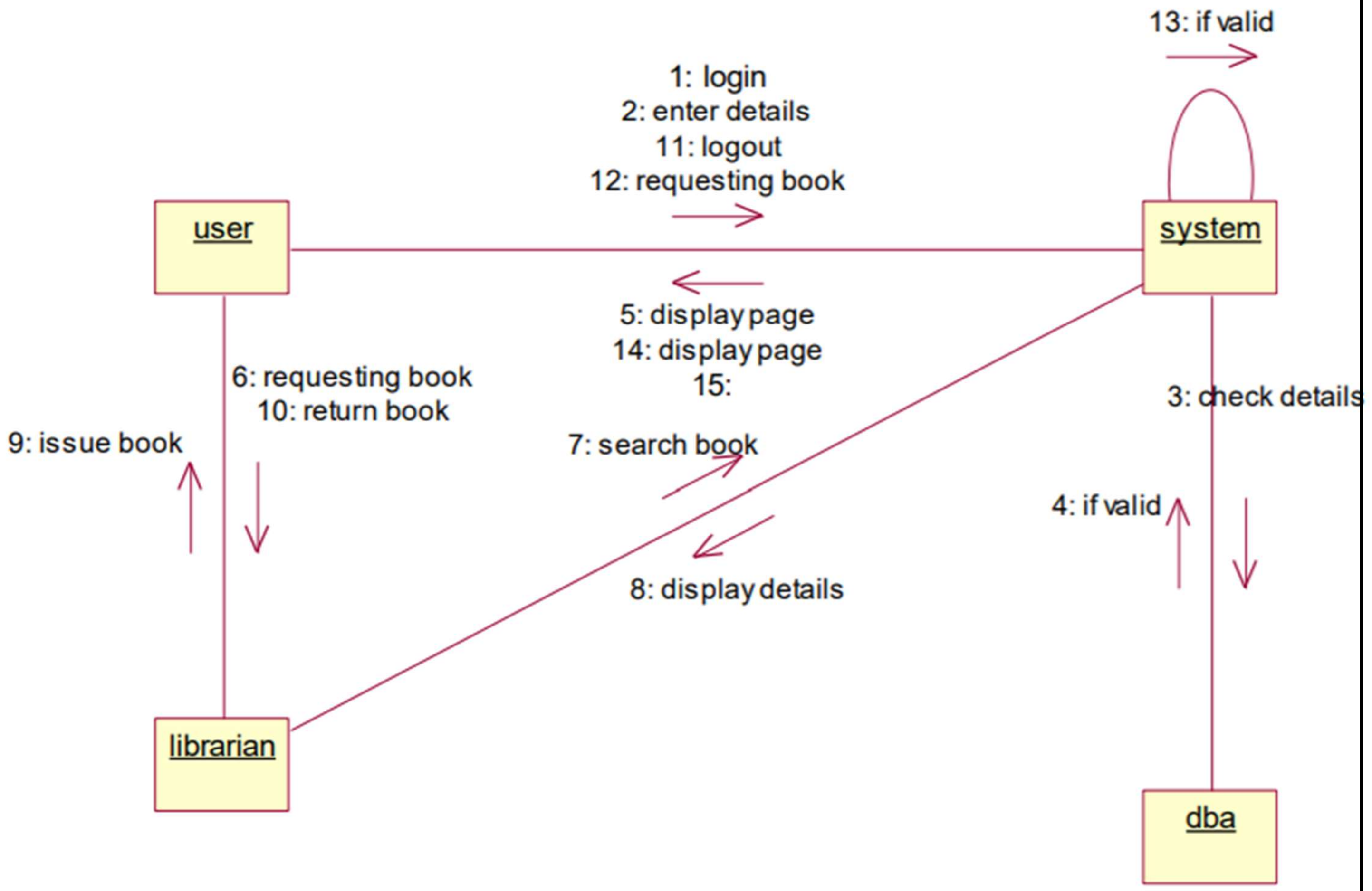
COMPONENT DIAGRAM :



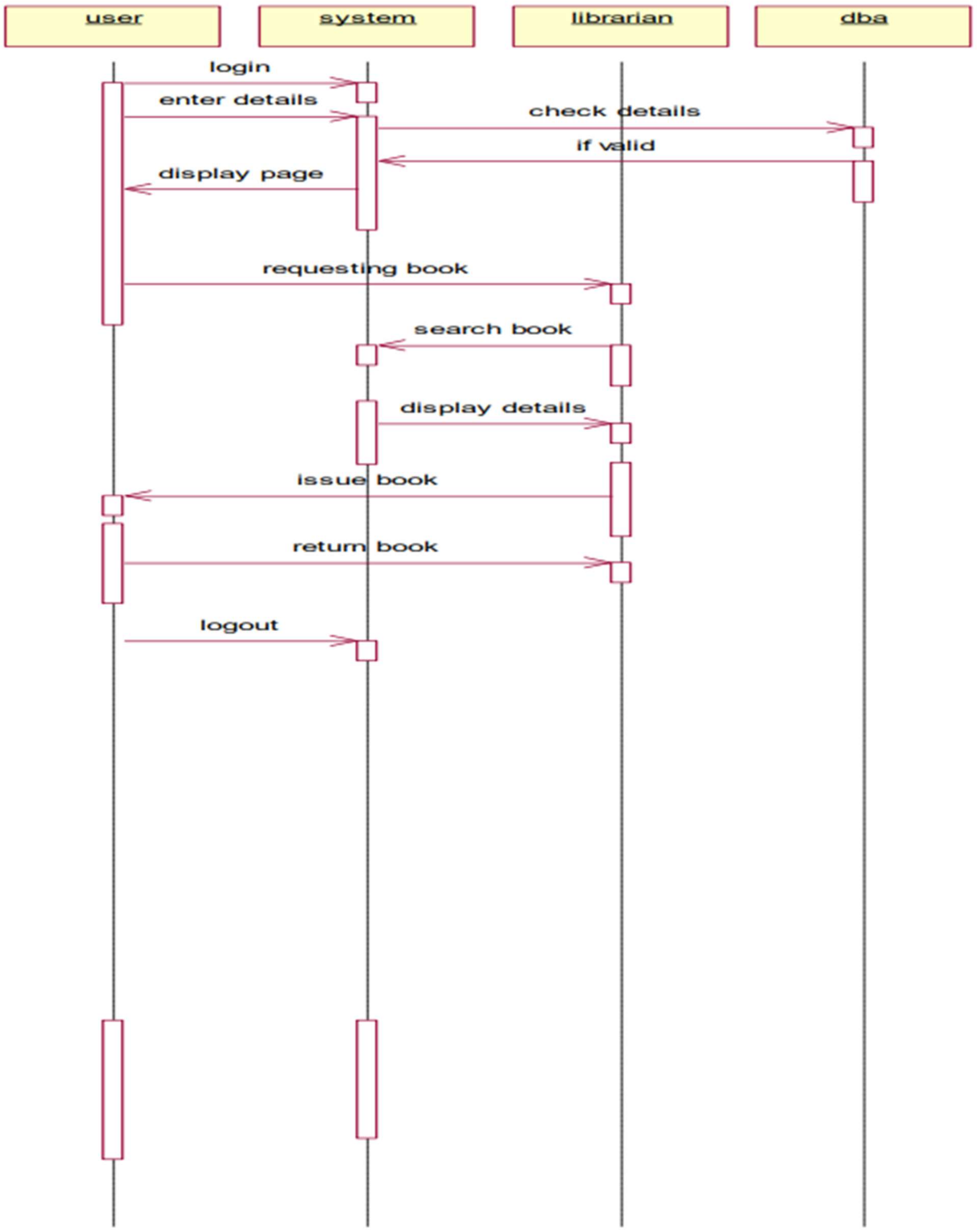
STATECHART DIAGRAM :



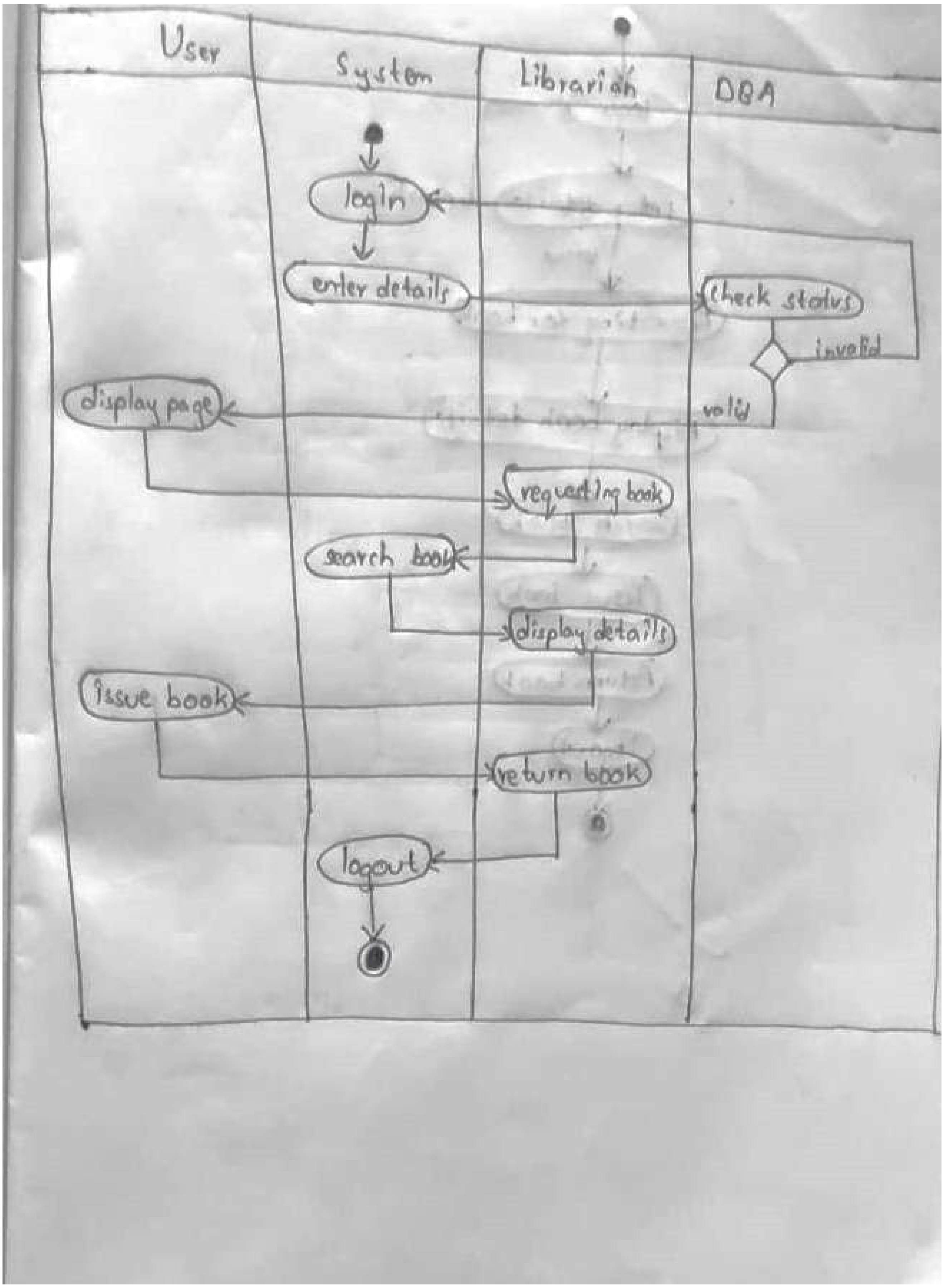
COLLABORATION DIAGRAM :



SEQUENCE DIAGRAM :



ACTIVITY DIAGRAM



4. Visual modeling and Rational Rose Tool

Visual modeling is the graphic representation of objects and systems of interest using graphical languages. Visual modeling languages may be General-Purpose Modeling languages or Domain-Specific Modeling languages.

Rational Rose is an object-oriented Unified Modeling Language (UML) software design tool intended for visual modeling and component construction of enterprise-level software applications. In much the same way a theatrical director blocks out a play, a software designer uses Rational Rose to visually create (model) the framework for an application by blocking out classes with actors (stick figures), use case elements (ovals), objects (rectangles) and messages/relationships (arrows) in a sequence diagram using drag-and-drop symbols. Rational Rose documents the diagram as it is being constructed and then generates code in the designer's choice of C++, Visual Basic, Java, Oracle8, Cobra or Data Definition Language.

Two popular features of Rational Rose are its ability to provide iterative development and round-trip engineering. Rational Rose allows designers to take advantage of iterative development (sometimes called evolutionary development) because the new application can be created in stages with the output of one iteration becoming the input to the next. (This is in contrast to waterfall development where the whole project is completed from start to finish before a user gets to try it out.) Then, as the developer begins to understand how the components interact and makes modifications in the design, Rational Rose can perform what is called "round-trip engineering" by going back and updating the rest of the model to ensure the code remains consistent.

5. UML INTRODUCTION

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

- UML was created by Object Management Group (OMG) and UML
- Specification draft was proposed to the OMG in January 1997.
- OMG is continuously putting effort to make a truly industry standard.
- UML stands for Unified Modeling Language.
- UML is different from the other common programming languages like C++, Java, and COBOL etc.
- UML is a pictorial language used to make software blue prints.

GOALS OF UML

- A picture is worth a thousand words, this absolutely fits while discussing about UML.
- UML diagrams are not only made for developers but also for business users, common people and anybody interested to understand the system.
- The system can be a software or non-software. So, it must be clear that.
- UML is not a development method rather it accompanies with processes to make a successful system.

CONCEPTUAL MODEL OF UML

➢ A conceptual model can be defined as a model which is made of Concepts and their relationships.

➢ A conceptual model is the first step before drawing a UML diagram. It helps to understand the entities in the real world and how they interact with each other. Conceptual model of UML can be mastered by learning.

The following three major elements:

➢ UML building blocks.
➢ Rules to connect the building blocks.
➢ Common mechanisms of UML.

OO Analysis and Design

The purpose of OO analysis and design can described as:

1.Identifying the objects of a system.

2.Identify their relationships.

3.Make a design which can be converted to executables using OO languages.

OO Analysis --> OO Design --> OO implementation using OO languages.

- During object-oriented analysis the most important purpose is to identify objects and describing them in a proper way. If these objects are identified efficiently then the next job of design is easy. The objects should be identified with responsibilities. Responsibilities are the functions performed by the object. Each and every object has some type of responsibilities to be performed. When these responsibilities are collaborated the purpose of the system is fulfilled.

- The second phase is object-oriented design. During this phase emphasis is given upon the requirements and their fulfillment. In this stage the objects are collaborated according to their intended association. After the association is complete the design is also complete.

- The third phase is object-oriented implementation. In this phase the design is implemented using object-oriented languages like Java, C++ etc.

BUILDING BLOCKS

The building blocks of UML can be defined as:

1. Things
2. Relationships
3. Diagrams

(1) Things

Things are the most important building blocks of UML. Things can be:

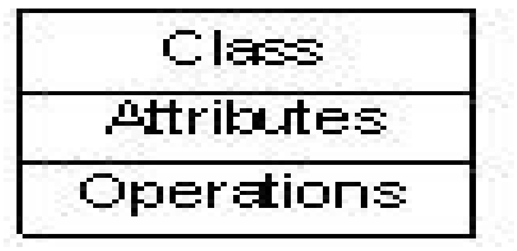
- Structural
- Behavioral
- Grouping
- A notational

Structural things

The **Structural things** define the static part of the model. They represent physical and conceptual elements. Following are the brief descriptions of the structural things.

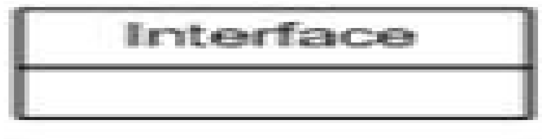
Class

Class represents set of objects having similar responsibilities.



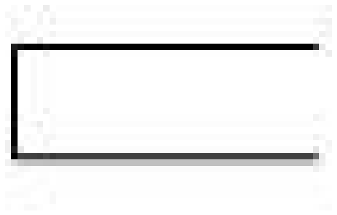
Interface

Interface defines a set of operations which specify the responsibility of a class.



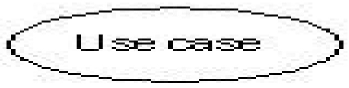
Collaboration

Collaboration defines interaction between elements



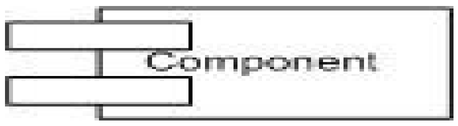
Use case

Use case represents a set of actions performed by a system for a specific goal.



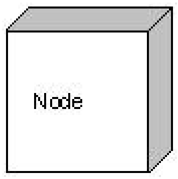
Component

Component describes physical part of a system.



Node

A node can be defined as a physical element that exists at run time.

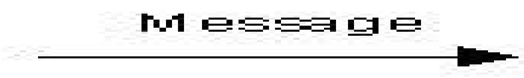


Behavioral things

A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things:

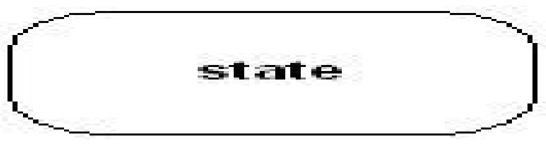
Interaction

Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.



State machine

State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change.

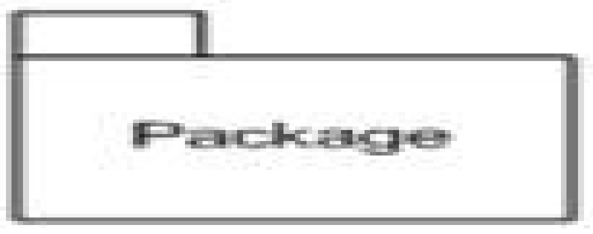


Grouping things

Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available

Package

Package is the only one grouping thing available for gathering structural **and behavioral things**

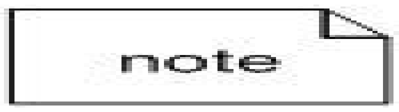


A notational thing

A notational thing can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note** is the only one A notational thing available

Note:

A note is used to render comments, constraints etc. of an UML element.



(2) Relationships

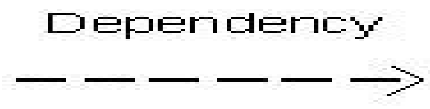
Relationships are another most important building block of UML. It shows how elements are associated with each other and this association describes the functionality of an application.

There are four kinds of relationships available.

- Dependency
- Association
- Generalization.
- Realization

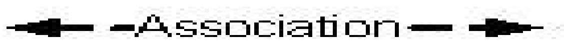
Dependency

Dependency is a relationship between two things in which change in one element also affects the other one.



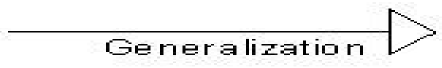
Association

Association is basically a set of links that connects elements of an UML model. It also describes how many objects are taking part in that relationship.



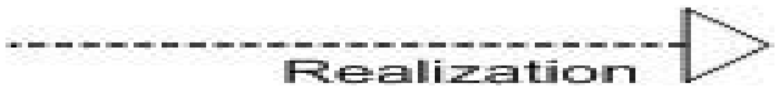
Generalization

Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes inheritance relationship in the world of objects.



Realization

Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility which is not implemented and the other one implements them. This relationship exists in case of interfaces



(3) Diagrams

Each UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction. UML diagrams commonly created in visual modeling tools include

Use Case Diagram displays the relationship among actors and use cases

Class Diagram models class structure and contents using design elements such as classes, packages and objects. It also displays relationships such as containment, inheritance, associations and others.

Interaction Diagrams

- **Sequence Diagram** displays the time sequence of the objects participating in the interaction. This consists of the vertical dimension (time) and horizontal dimension (different objects).
- **Collaboration Diagram** displays an interaction organized around the objects and their links to one another. Numbers are used to show the sequence of messages.

State Diagram displays the sequences of states that an object of an interaction goes through during its life in response to received stimuli, together with its responses and actions.

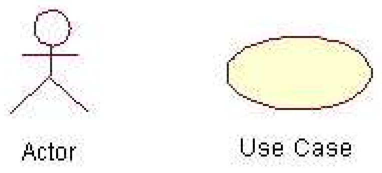
Activity Diagram displays a special state diagram where most of the states are action states and most of the transitions are triggered by completion of the actions in the source states. This diagram focuses on flows driven by internal processing.

Physical Diagrams

- **Component Diagram** displays the high-level packaged structure of the code itself. Dependencies among components are shown, including source code components, binary code components, and executable components. Some components exist at compile time, at link time, at run times well as at more than one time.
- **Deployment Diagram** displays the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of code units.

Use Case Diagrams

A use case is a set of scenarios that describing an interaction between a user and a system. A use case diagram displays the relationship among actors and use cases. The two main components of a use case diagram are use cases and actors.



An actor is representing a user or another system that will interact with the system you are modeling. A use case is an external view of the system that represents some action the user might perform in order to complete a task.

When to Use: Use Cases Diagrams

Use cases are used in almost every project. These are helpful in exposing requirements and planning the project. During the initial stage of a project most use cases should be defined, but as the project continues more might become visible.

Modeling steps for Use case Diagram

1. Draw the lines around the system and actors lie outside the system.
2. Identify the actors which are interacting with the system.
3. Separate the generalized and specialized actors.
4. Identify the functionality the way of interacting actors with system and specify the behavior of actor.
5. Functionality or behavior of actors is considered as use cases.
6. Specify the generalized and specialized use cases.

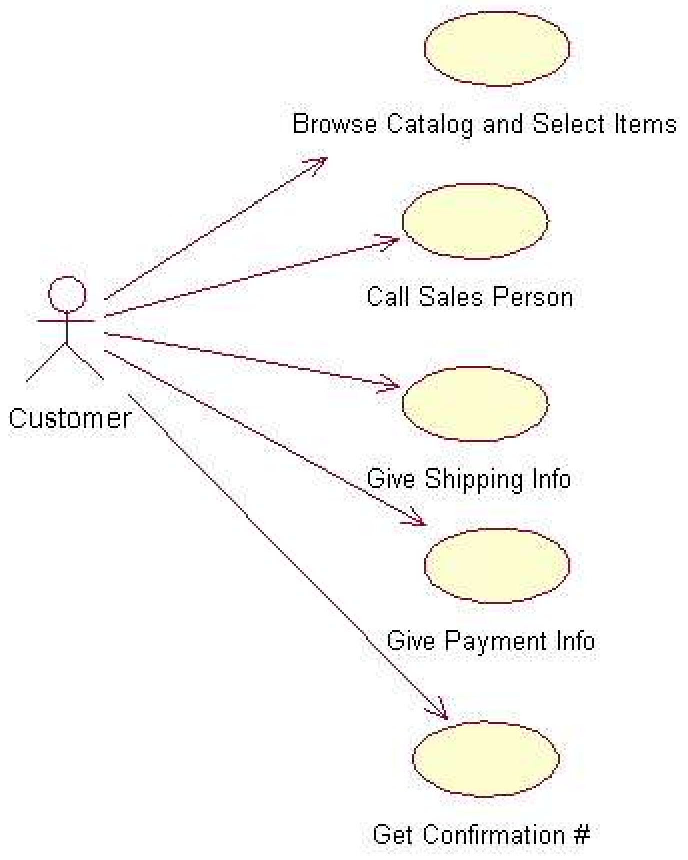
7. Se the relationship among the use cases and in between actor and use cases.
8. Adorn with constraints and notes.
9. If necessary, use collaborations to realize use cases.

How to Draw: Use Cases Diagrams

Use cases are a relatively easy UML diagram to draw, but this is a very simplified example. This example is only meant as an introduction to the UML and use cases. Start by listing a sequence of steps a user might take in order to complete an action. For example, a user placing an order with a sales company might follow these steps.

1. Browse catalog and select items.
2. Call sales representative.
3. Supply shipping information.
4. Supply payment information.
5. Receive conformation number from salesperson.

These steps would generate this simple use case diagram:



This example shows the customer as an actor because the customer is using the ordering system. The diagram takes the simple steps listed above and shows them as actions the customer might perform. The salesperson could also be included in this use case diagram because the salesperson is also interacting with the ordering system.

From this simple diagram the requirements of the ordering system can easily be derived. The system will need to be able to perform actions for all of the use cases listed. As the project progresses other use cases might appear. The customer might have a need to add an item to an order that has already been placed. This diagram can easily be expanded until a complete description of the ordering system is derived capturing all of the requirements that the system will need to perform.

The <<*extends*>> Relationship

- <<Extends>> relationships represent exceptional or seldom invoked cases.
- The exceptional event flows are factored out of the main event flow for clarity.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extends>> relationship is to the extended use case

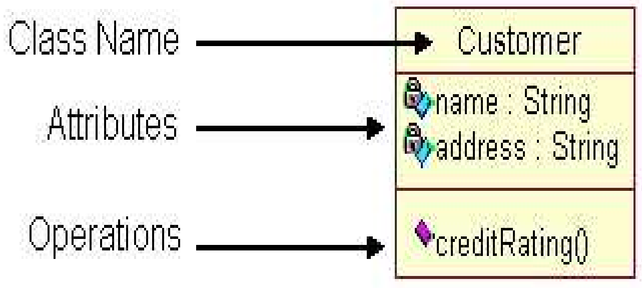
The <<*includes*>> Relationship

- <<Includes>> relationship represents behavior that is factored out of the use case.
- <<Includes>> behavior is factored out for reuse, not because it is an exception.
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships).

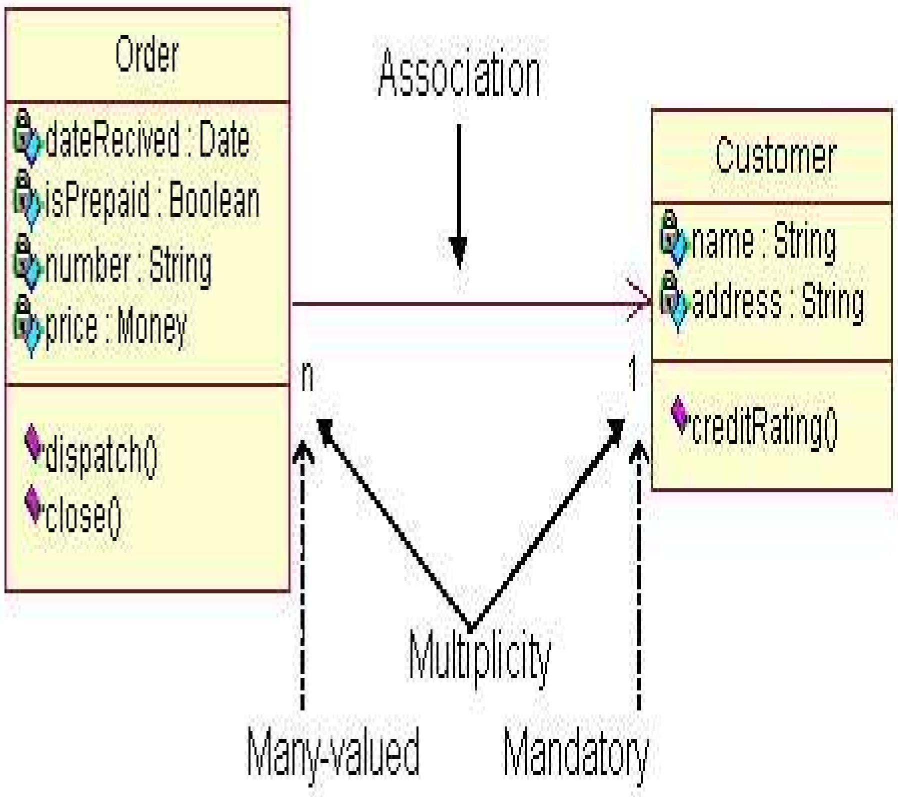
Class Diagrams

Class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system, conceptual, specification, and implementation. These perspectives become evident as the diagram is created and help solidify the design. This example is only meant as an introduction to the UML and class diagrams. Classes are composed of three things: a name, attributes, and operations.

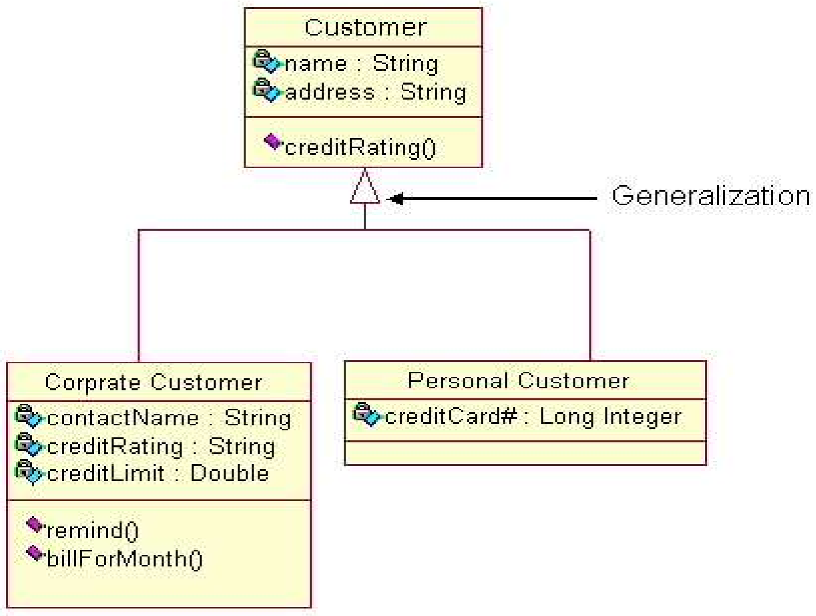
Below is an example of a class.



Class diagrams also display relationships such as containment, inheritance, associations and others. Below is an example of an associative relationship:



The association relationship is the most common relationship in a class diagram. The association shows the relationship between instances of classes. For example, the class Order is associated with the class Customer. The multiplicity of the association denotes the number of objects that can participate in the relationship. For example, an Order object can be associated to only one customer, but a customer can be associated to many orders. Another common relationship in class diagrams is a generalization. A generalization is used when two classes are similar, but have some differences.



In this example the classes Corporate Customer and Personal Customer have some similarities such as name and address, but each class has some of its own attributes and operations. The class Customer is a general form of both the Corporate Customer and Personal Customer classes. This allows the designers to just use the Customer class for modules and do not require in-depth representation of each type of customer.

When to Use: Class Diagrams

Class diagrams are used in nearly all Object-Oriented software designs. Use them to describe the Classes of the system and their relationships to each other.

Modeling steps for Class Diagrams

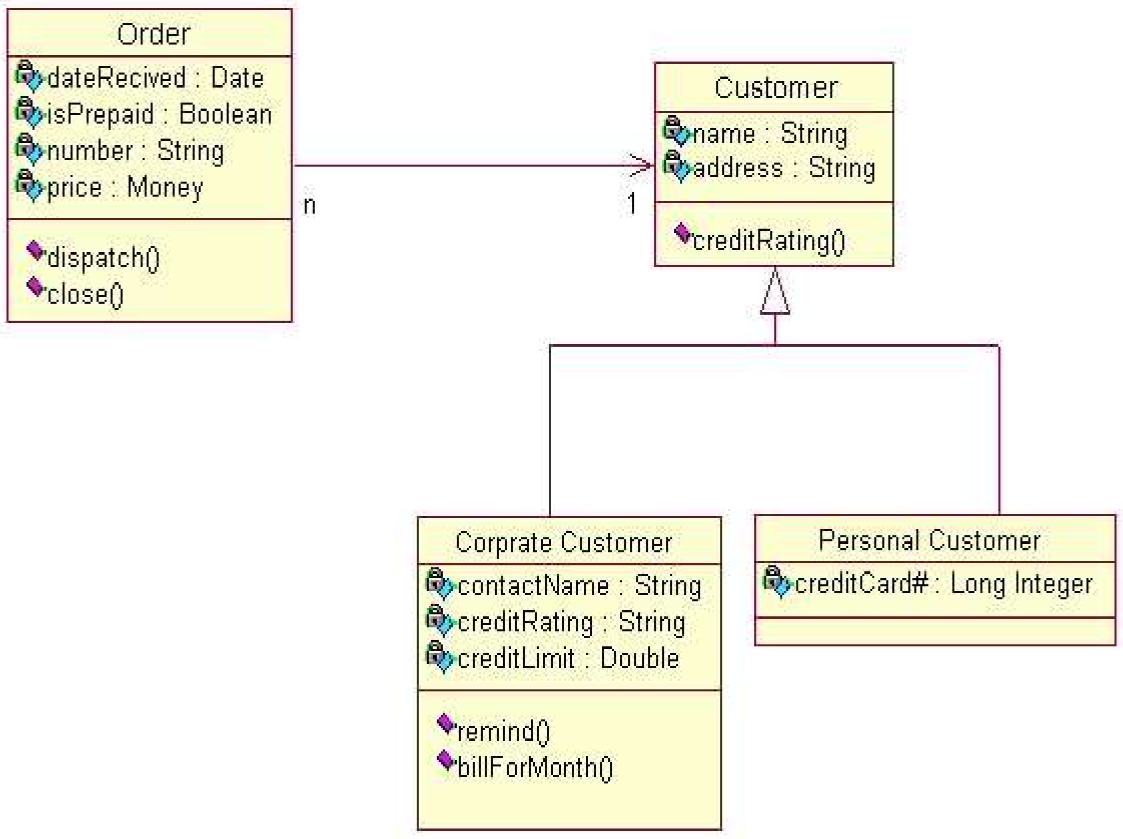
1. Identity the things that are interacting with class diagram.
2. Set the attributes and operations.
3. Set the responsibilities.
4. Identify the generalization and specification classes.
5. Set the relationship among all the things.
6. Adorn with tagged values, constraints and notes.

How to Draw: Class Diagrams

Class diagrams are some of the most difficult UML diagrams to draw. To draw detailed and useful diagrams a person would have to study UML and Object-Oriented principles for a long time. Therefore, this page will give a very high-level overview of the process.

Before drawing a class diagram consider the three different perspectives of the system the diagram will present; conceptual, specification, and implementation. Try not to focus on one perspective and try see how they all work together.

When designing classes consider what attributes and operations it will have. Then try to determine how instances of the classes will interact with each other. These are the very first steps of many in developing a class diagram. However, using just these basic techniques one can develop a complete view of the software system.



Interaction Diagrams

Interaction diagrams model the behavior of use cases by describing the way groups of objects interact to complete the task. The two kinds of interaction diagrams are **sequence** and **collaboration** diagrams. This example is only meant as an introduction to the UML and interaction diagrams.

When to Use: Interaction Diagrams

Interaction diagrams are used when you want to model the behavior of several objects in a use case. They demonstrate how the objects collaborate for the behavior. Interaction diagrams do not give a in depth representation of the behavior. If you want to see what a specific object is doing for several use cases use a state diagram. To see a particular behavior over many use cases or threads use an activity diagrams.

How to Draw: Interaction Diagrams

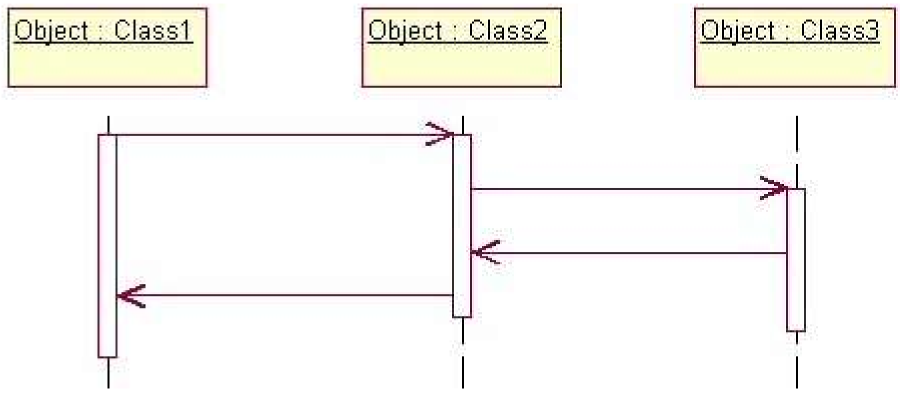
Sequence diagrams, collaboration diagrams, or both diagrams can be used to demonstrate the interaction of objects in a use case. Sequence diagrams generally show the sequence of events that occur. Collaboration diagrams demonstrate how objects are statically connected. Both diagrams are relatively simple to draw and contain similar elements.

Sequence diagrams:

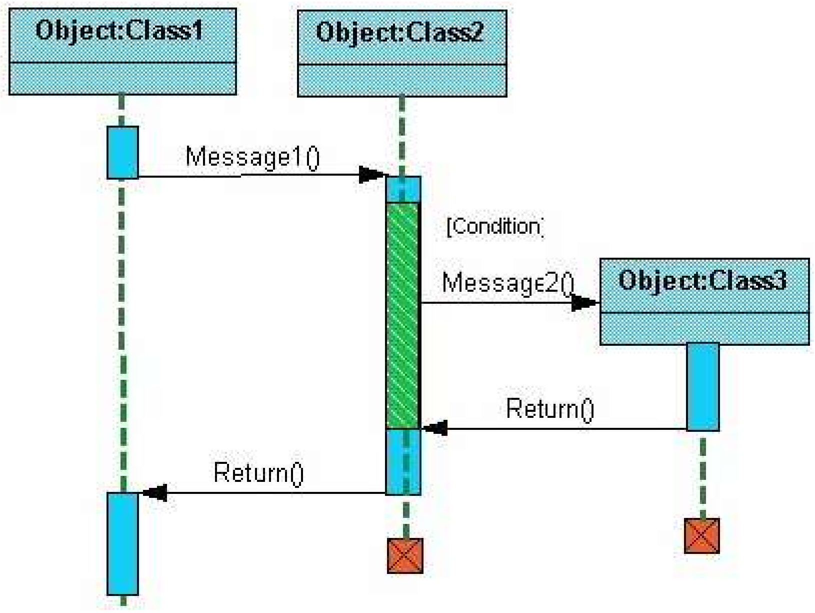
Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass. the diagrams are read left to right and descending. The example below shows an object of class 1 start the behavior by sending a message to an object of class 2. Messages pass between the different objects until the object of class 1 receives the final message

Modeling steps for Sequence Diagrams

1. Set the context for the interactions, system, subsystem, classes, object or use cases.
2. Set the stages for the interactions by identifying objects which are placed as actions in interaction diagrams.
3. Lay them out along the X-axis by placing the important object at the left side and others in the next subsequent.
4. Set the lifelines for each and every object by sending create and destroy messages.
5. Start the message which is initiating interactions and place all other messages in the increasing order of items.
6. Specify the time and space constraints.
7. Set the pre and post conditions.

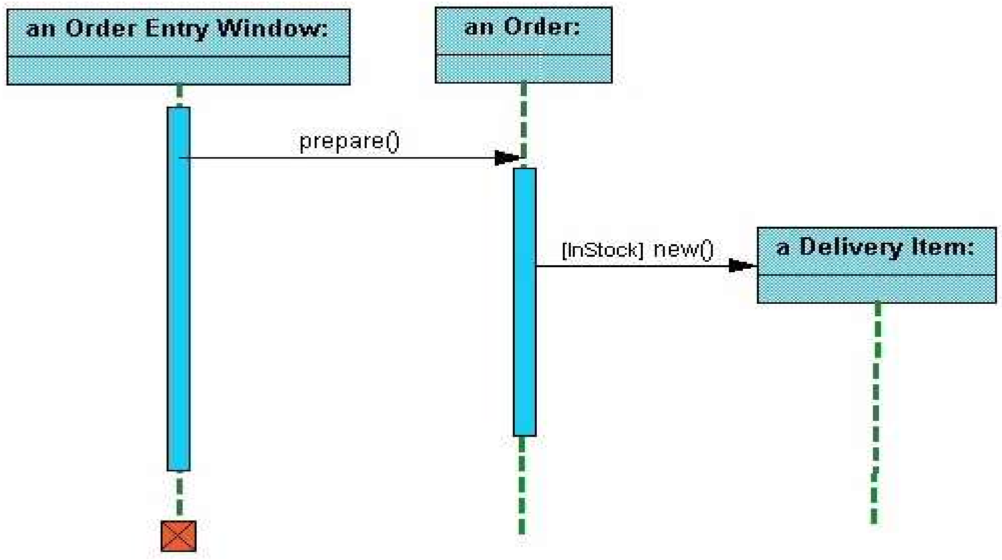


Below is a slightly more complex example. The light blue vertical rectangles the objects activation while the green vertical dashed lines represent the life of the object. The green vertical rectangles represent when a particular object has control. The ☒represents when the object is destroyed. This diagrams also shows conditions for messages to be sent to another object. The condition is listed between brackets next to the message. For example, a [condition] has to be met before the object of class 2 can send a message () to the object of class 3.

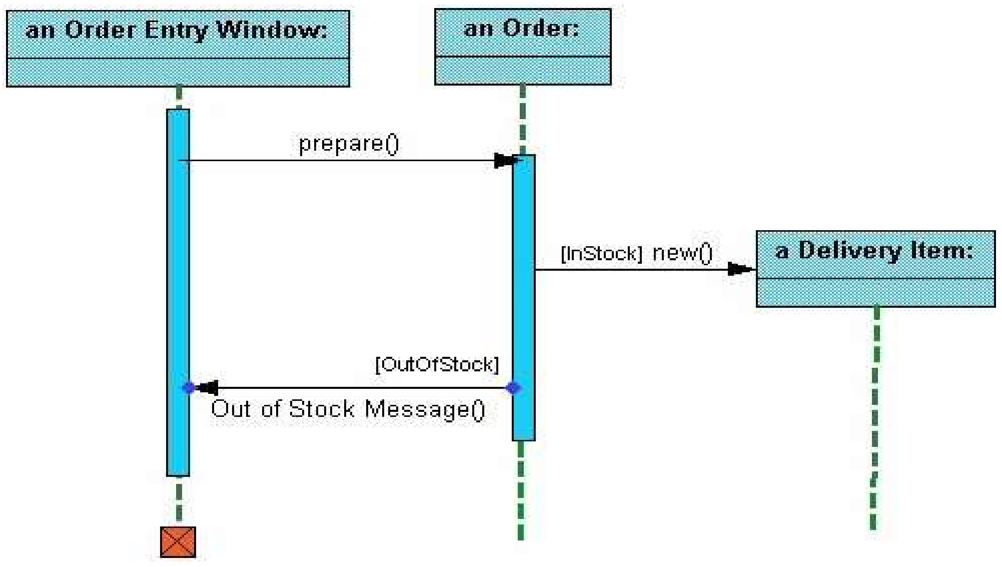


The next diagram shows the beginning of a sequence diagram for placing an order. The object an Order Entry Window is created and sends a message to an Order object to prepare the order. Notice the the names of the objects are followed by a colon. The names of the classes the objects belong to do not have to be listed. However the colon is required to denote that it is the name of an object following the objectName:className naming system.

Next the Order object checks to see if the item is in stock and if the [InStock] condition is met it sends a message to create an new Delivery Item object.



The next diagrams adds another conditional message to the Order object. If the item is [OutOfStock] it sends a message back to the Order Entry Window object stating that the object is out of stack.



This simple diagram shows the sequence that messages are passed between objects to complete a use case for ordering an item.

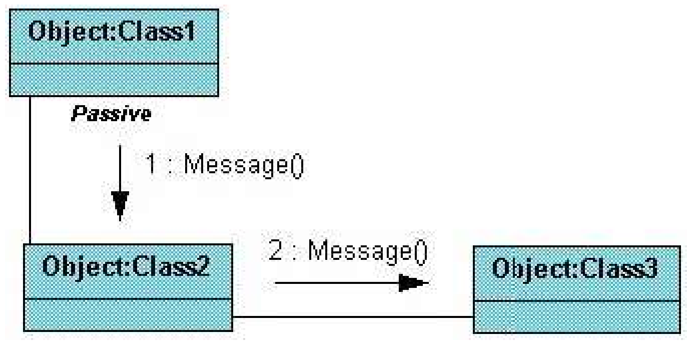
Collaboration diagrams

Collaboration diagrams are also relatively easy to draw. They show the relationship between objects and the order of messages passed between them. The objects are listed as icons and arrows indicate the messages being passed between them. The numbers next to the messages are called sequence numbers. As the name suggests, they show the sequence of the messages as they are passed between the objects. There are many acceptable sequence numbering schemes in UML. A simple 1, 2, 3... format can be used, as the example below shows, or for more detailed and complex diagrams a 1, 1.1 ,1.2, 1.2.1... scheme can be used.

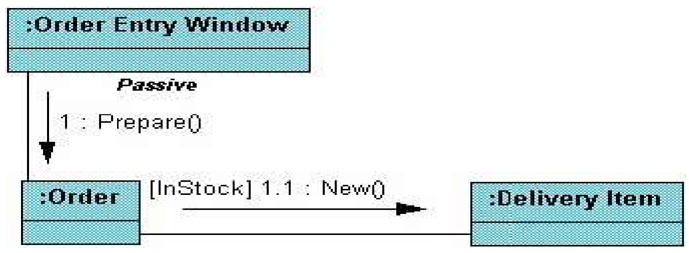
Modeling steps for Collaboration Diagrams

1. Set the context for interaction, whether it is ystem, subsystem, operation or class or one scenario of use case or collaboration.
2. Identify the objects that play a role in the interaction. Lay them as vertices in graph, placing important objects in centre and neigboring objects to outside.
3. Set the initial properties of each of these objects. If the attributes or tagged values of an object changes in significant ways over the interaction, place a duplicate object, update with these new values and connect them by a message stereotyped as become or copy.
4. Specify the links among these objects.Lay the association links first represent structural connection lay out other links and adorn with stereotypes.

5. Starting with the message that initiates this interaction, attach each subsequent message to appropriate link, setting sequence number as appropriate.



The example below shows a simple collaboration diagram for the placing an order use case. This time the names of the objects appear after the colon, such as :Order Entry Window following the objectName:className naming convention. This time the class name is shown to demonstrate that all of objects of that class will behave the same way.



State chart diagram

State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and track the different states of its objects through the system.

Modeling steps for State chart Diagram

1. Choose the context for state machine, whether it is a class ,a use case, or the system as a whole.
2. Choose the initial & final states of the objects.
3. Decide on the stable states of the object by considering the conditions in which the object may exist for some identifiable period of time. Start with the high level states of the objects & only then consider its possible substates.
4. Decide on the meaningful partial ordering of stable states over the lifetime of the object.
5. Decide on the events that may trigger a transition from state to state. Model these events as triggers to transitions that move from one legal ordering of states to another.
6. Attach actions to these transitions and/or to these states.

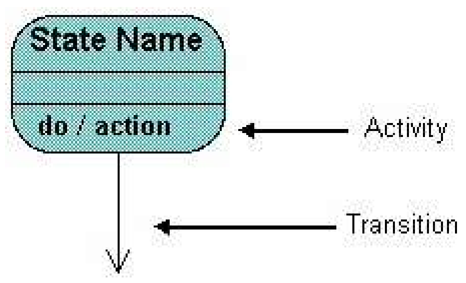
7. Consider ways to simplify your machine by using substates, branches, forks, joins and history states.
8. Check that all states are reachable under some combination of events.
9. Check that no state is a dead from which no combination of events will transition the object out of that state.
10. Trace through the state machine, either manually or by using tools, to check it against expected sequence of events & their responses.

When to Use: State Diagrams

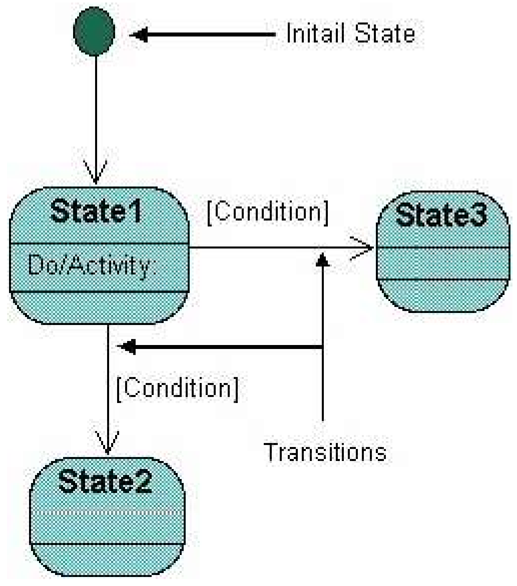
Use state diagrams to demonstrate the behavior of an object through many use cases of the system. Only use state diagrams for classes where it is necessary to understand the behavior of the object through the entire system. Not all classes will require a state diagram and state diagrams are not useful for describing the collaboration of all objects in a use case. State diagrams are other combined with other diagrams such as interaction diagrams and activity diagrams.

How to Draw: State Diagrams

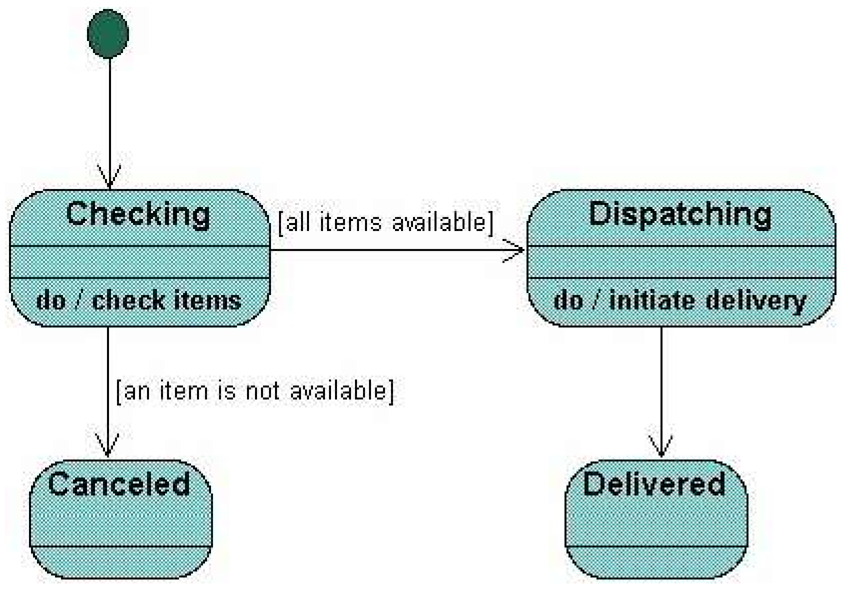
State diagrams have very few elements. The basic elements are rounded boxes representing the state of the object and arrows indicting the transition to the next state. The activity section of the state symbol depicts what activities the object will be doing while it is in that state.



All state diagrams being with an initial state of the object. This is the state of the object when it is created. After the initial state the object begins changing states. Conditions based on the activities can determine what the next state the object transitions to.

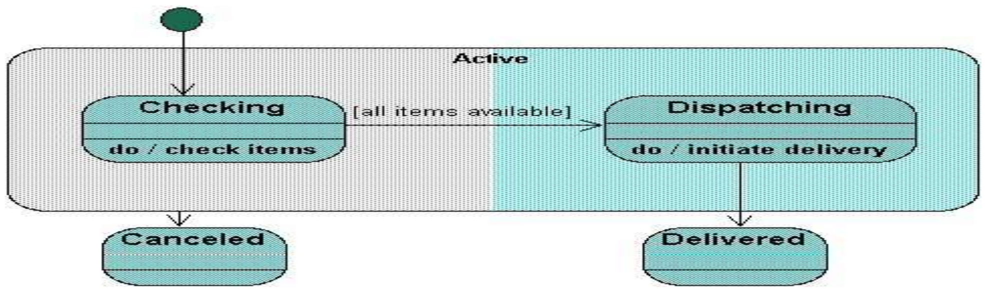


Below is an example of a state diagram might look like for an Order object. When the object enters the Checking state it performs the activity "check items." After the activity is completed the object transitions to the next state based on the conditions [all items available] or [an item is not available]. If an item is not available the order is canceled. If all items are available then the order is dispatched. When the object transitions to the Dispatching state the activity "initiate delivery" is performed. After this activity is complete the object transitions again to the Delivered state.



State diagrams can also show a super-state for the object. A super-state is used when many transitions lead to the certain state. Instead of showing all of the transitions from each state to the redundant state a super-state can be used to show that all of the states inside of the super-state can transition to the redundant state. This helps make the state diagram easier to read.

The diagram below shows a super-state. Both the Checking and Dispatching states can transition into the Canceled state, so a transition is shown from a super-state named Active to the state Cancel. By contrast, the state Dispatching can only transition to the Delivered state, so we show an arrow only from the Dispatching state to the Delivered state.



Activity Diagram

Activity diagrams describe the workflow behavior of a system. Activity diagrams are similar to state diagrams because activities are the state of doing something. The diagrams describe the state of activities by showing the sequence of activities performed. Activity diagrams can show activities that are conditional or parallel.

Modeling steps for Activity Diagrams

1. Select the object that have high level responsibilities.
2. These objects may be real or abstract. In either case, create a swimlane for each important object.
3. Identify the precondition of initial state and post conditions of final state.
4. Beginning at initial state, specify the activities and actions and render them as activity states or action states.
5. For complicated actions, or for a set of actions that appear multiple times, collapse these states and provide separate activity diagram.
6. Render the transitions that connect these activities and action states.
7. Start with sequential flows, consider branching, fork and joining.
8. Adorn with notes tagged values and so on.

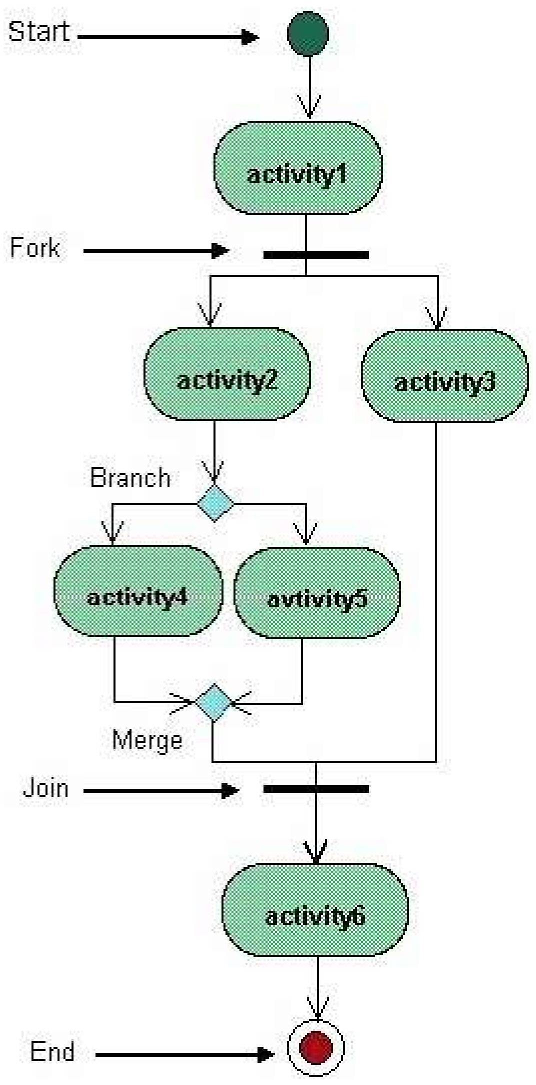
When to Use: Activity Diagrams

Activity diagrams should be used in conjunction with other modeling techniques such as interaction diagrams and state diagrams. The main reason to use activity diagrams is to model the workflow behind the system being designed. Activity Diagrams are also useful for analyzing a use case by describing what actions need to take place and when they should occur; describing a complicated sequential algorithm; and modeling applications with parallel processes.

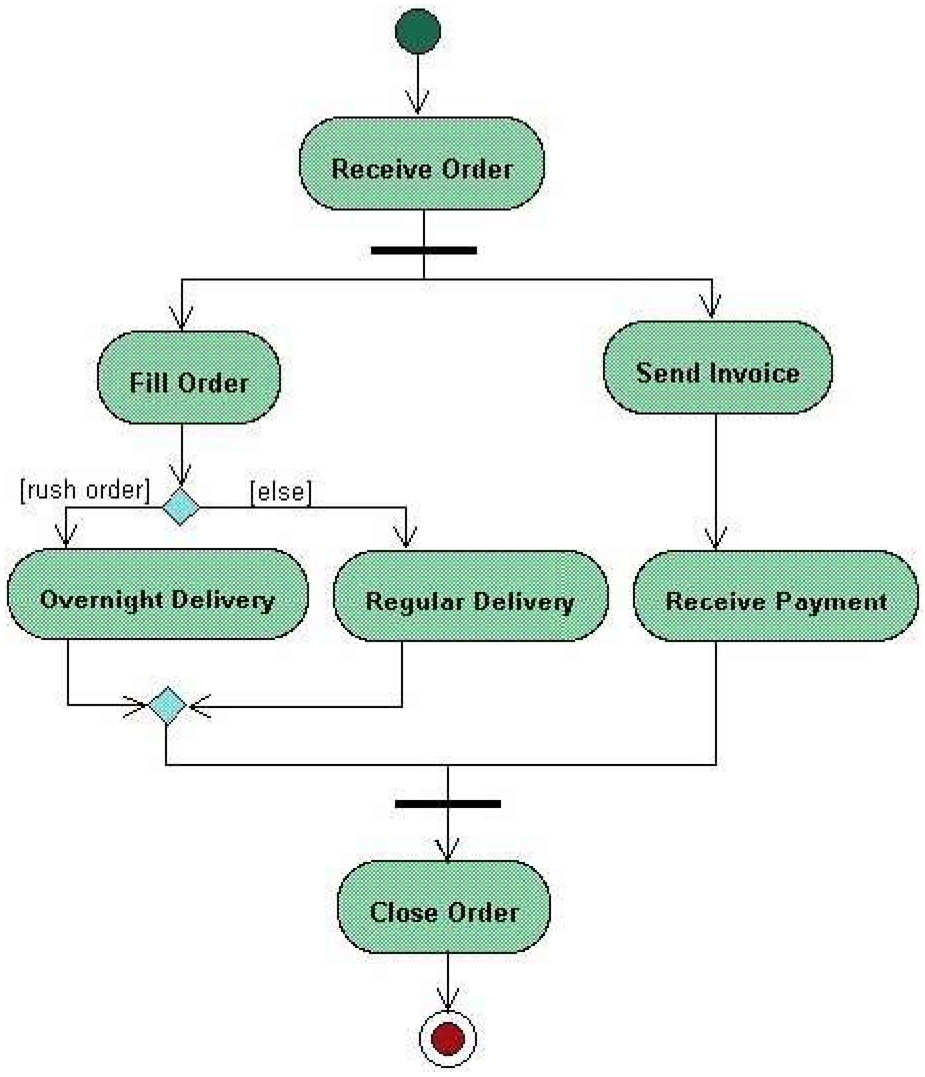
However, activity diagrams should not take the place of interaction diagrams and state diagrams. Activity diagrams do not give detail about how objects behave or how objects collaborate.

How to Draw: Activity Diagrams

Activity diagrams show the flow of activities through the system. Diagrams are read from top to bottom and have branches and forks to describe conditions and parallel activities. A fork is used when multiple activities are occurring at the same time. The diagram below shows a fork after activity1. This indicates that both activity2 and activity3 are occurring at the same time. After activity2 there is a branch. The branch describes what activities will take place based on a set of conditions. All branches at some point are followed by a merge to indicate the end of the conditional behavior started by that branch. After the merge all of the parallel activities must be combined by a join before transitioning into the final activity state.



Below is a possible activity diagram for processing an order. The diagram shows the flow of actions in the system's workflow. Once the order is received the activities split into two parallel sets of activities. One side fills and sends the order while the other handles the billing. On the Fill Order side, the method of delivery is decided conditionally. Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed. Finally the parallel activities combine to close the order.



Physical Diagrams

There are two types of physical diagrams: **deployment diagrams** and **component diagrams.** Deployment diagrams show the physical relationship between hardware and software in a system. Component diagrams show the software components of a system and how they are related to each other. These relationships are called dependencies.

Modeling steps for Component Diagrams

1. Identify the component libraries and executable files which are interacting with the system.
2. Represent this executables and libraries as components.
3. Show the relationships among all the components.
4. Identify the files,tables,documents which are interacting with the system.
5. Represent files,tables,documents as components.
6. Show the existing relationships among them generally dependency.
7. Identify the seams in the model.
8. Identify the interfaces which are interacting with the system.
9. Set attributes and operation signatures for interfaces.
10. Use either import or export relationship in b/w interfaces & components.

11. Identify the source code which is interacting with the system.
12. Set the version of the source code as a constraint to each source code.
13. Represent source code as components.
14. Show the relationships among components.

Modeling steps for Deployment Diagram

1. Identify the processors which represent client & server.
2. Provide the visual cue via stereotype classes.
3. Group all the similar clients into one package.
4. Provide the links among clients & servers.
5. Provide the attributes & operations.
6. Specify the components which are living on nodes.
7. Adorn with nodes & constraints & draw the deployment diagram.

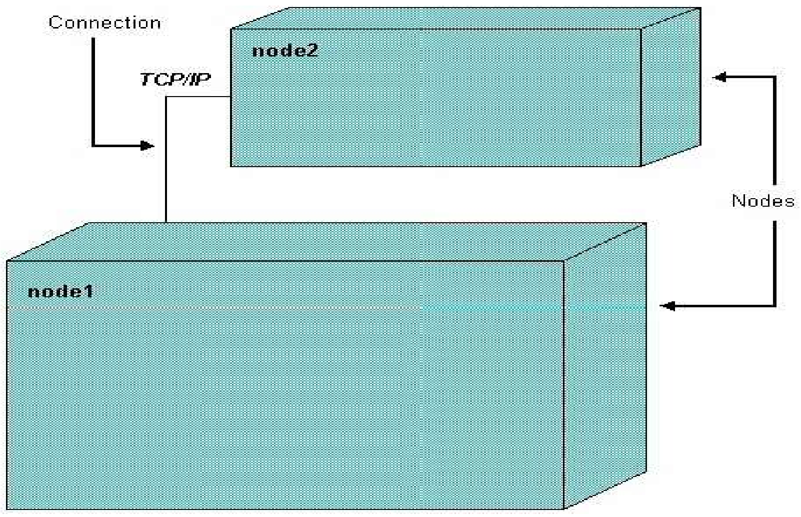
When to Use: Physical Diagrams

Physical diagrams are used when development of the system is complete. Physical diagrams are used to give descriptions of the physical information about a system.

How to Draw: Physical Diagrams

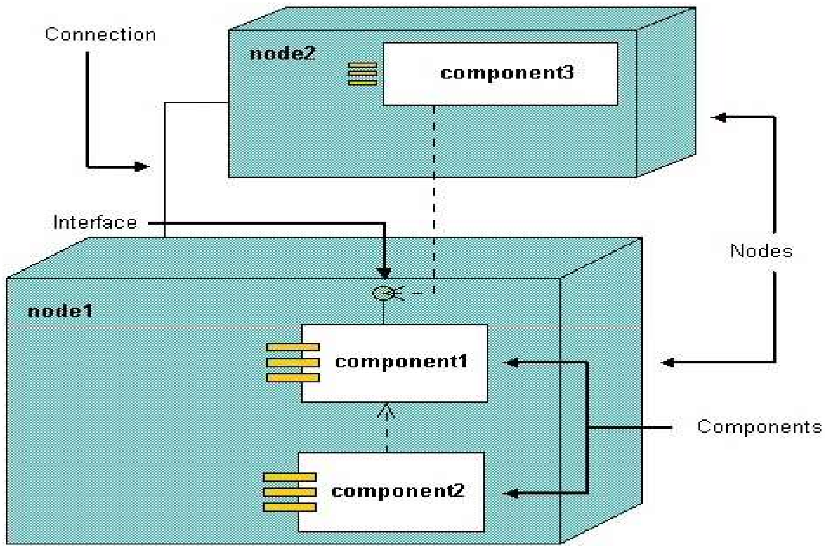
Many times the deployment and component diagrams are combined into one physical diagram. A combined deployment and component diagram combines the features of both diagrams into one diagram.

The deployment diagram contains nodes and connections. A node usually represents a piece of hardware in the system. A connection depicts the communication path used by the hardware to communicate and usually indicates a method such as TCP/IP.



The component diagram contains components and dependencies. Components represent the physical packaging of a module of code. The dependencies between the components show how changes made to one component may affect the other components in the system. Dependencies in a component diagram are represented by a dashed line between two or more components. Component diagrams can also show the interfaces used by the components to communicate to each other.

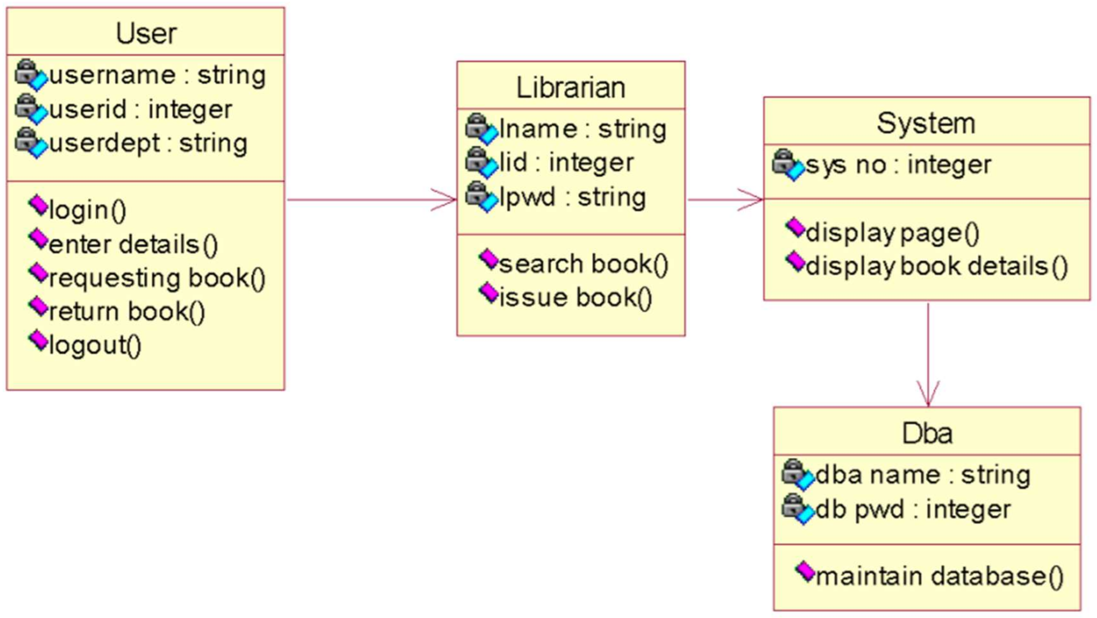
The combined deployment and component diagram below gives a high level physical description of the completed system. The diagram shows two nodes which represent two machines communicating through TCP/IP. Component2 is dependent on component1, so changes to component 2 could affect component1. The diagram also depicts component3 interfacing with component1. This diagram gives the reader a quick overall view of the entire system.



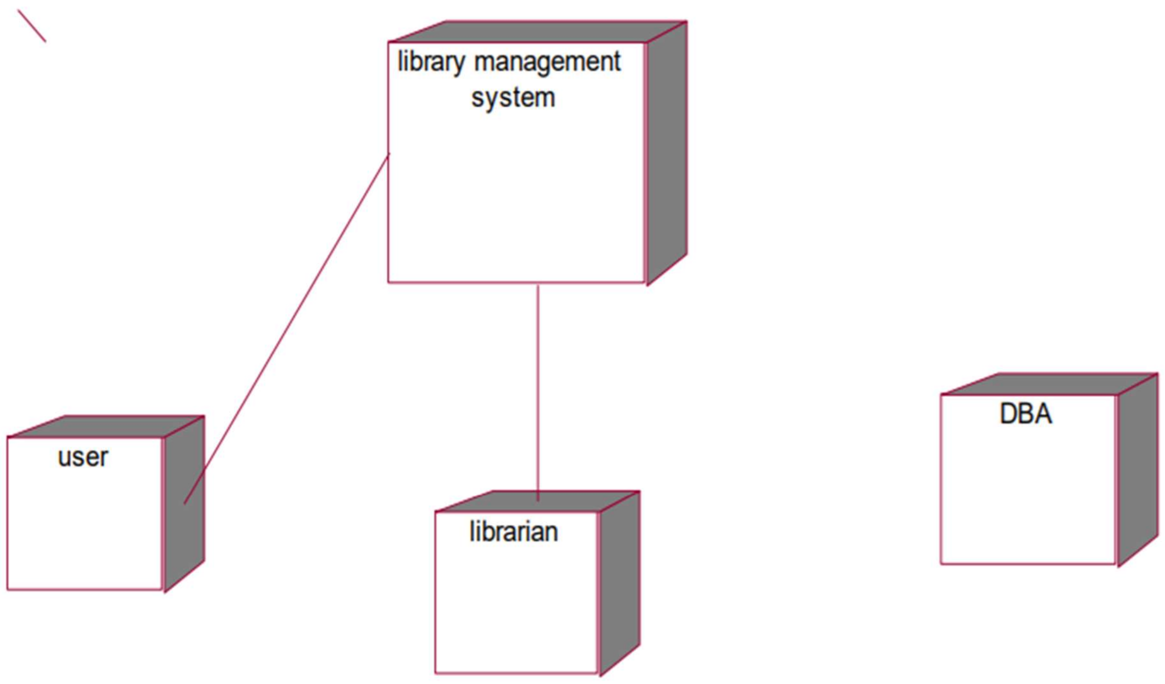
6. APPLICATIONS

BOOK BANK MANAGEMENT SYSTEM

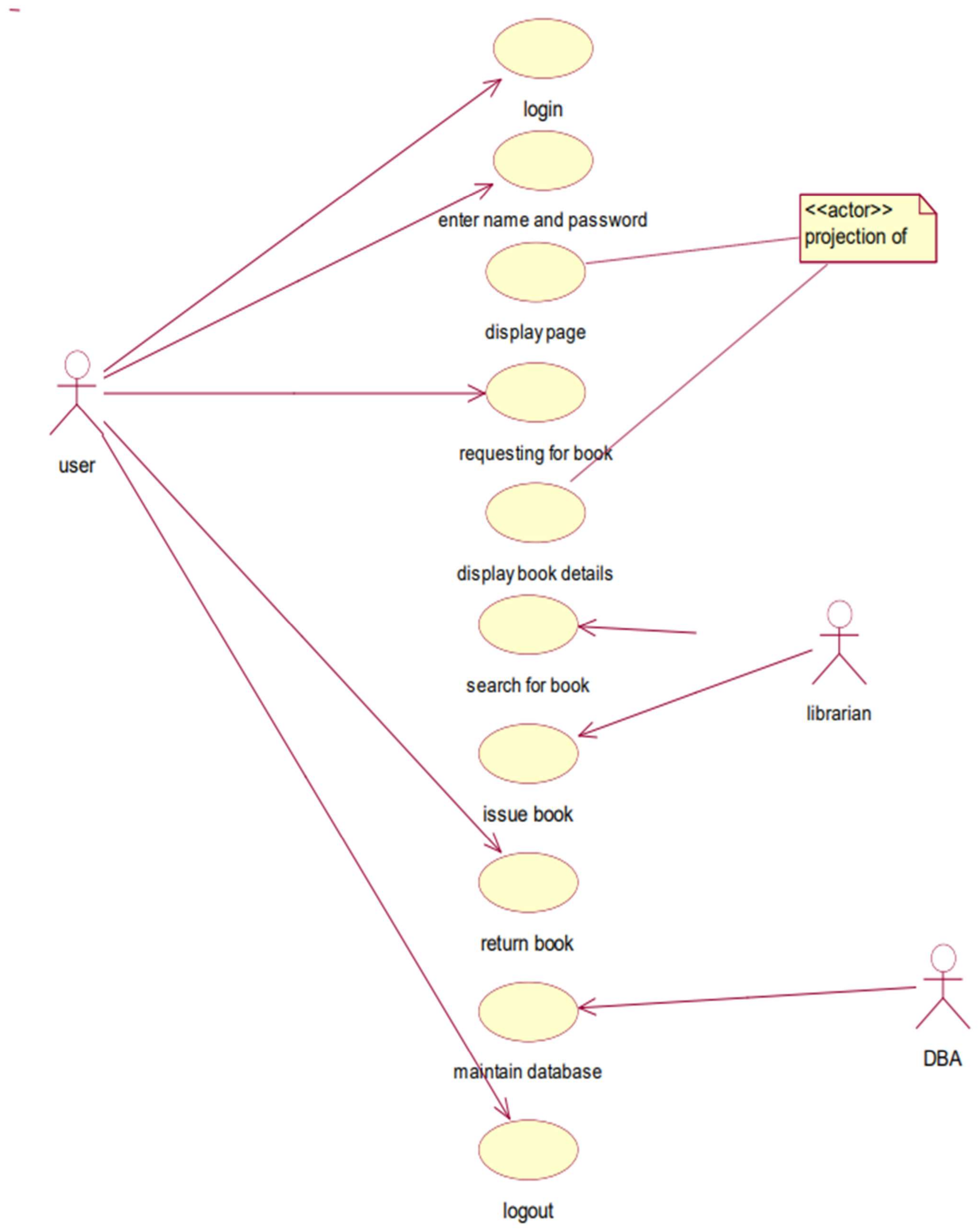
CLASS DIAGRAM:



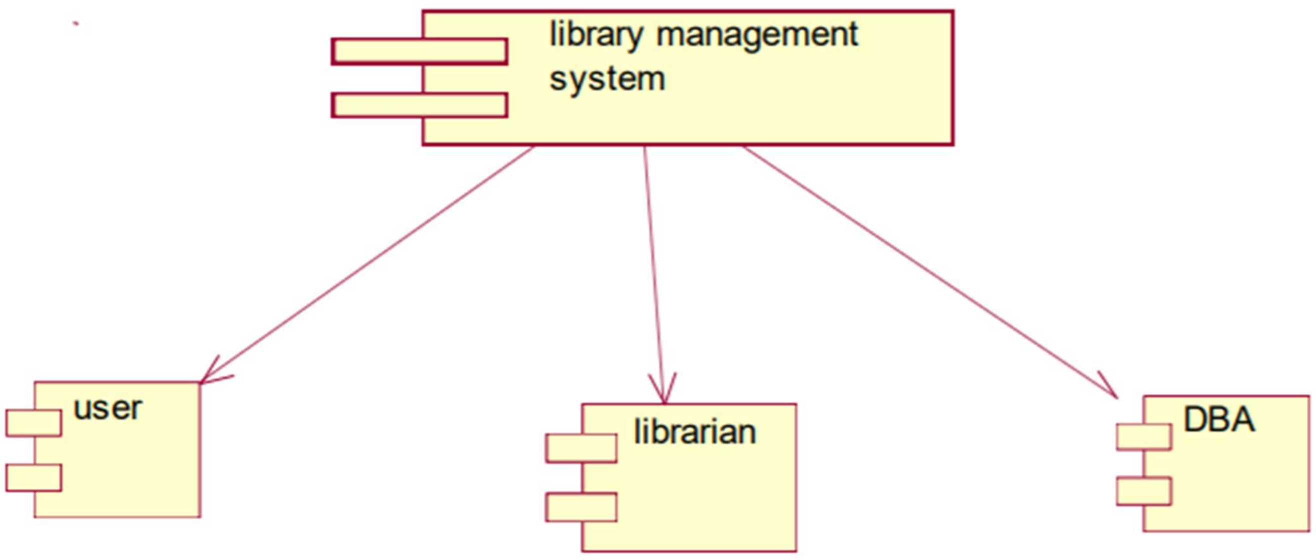
DEPLOYMENT DIAGRAM :



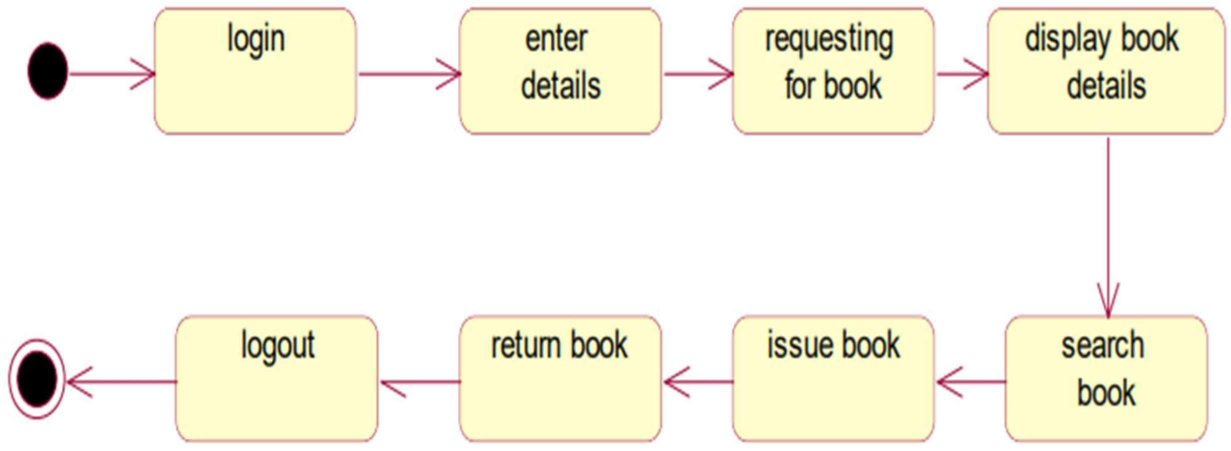
USE CASE DIAGRAM :



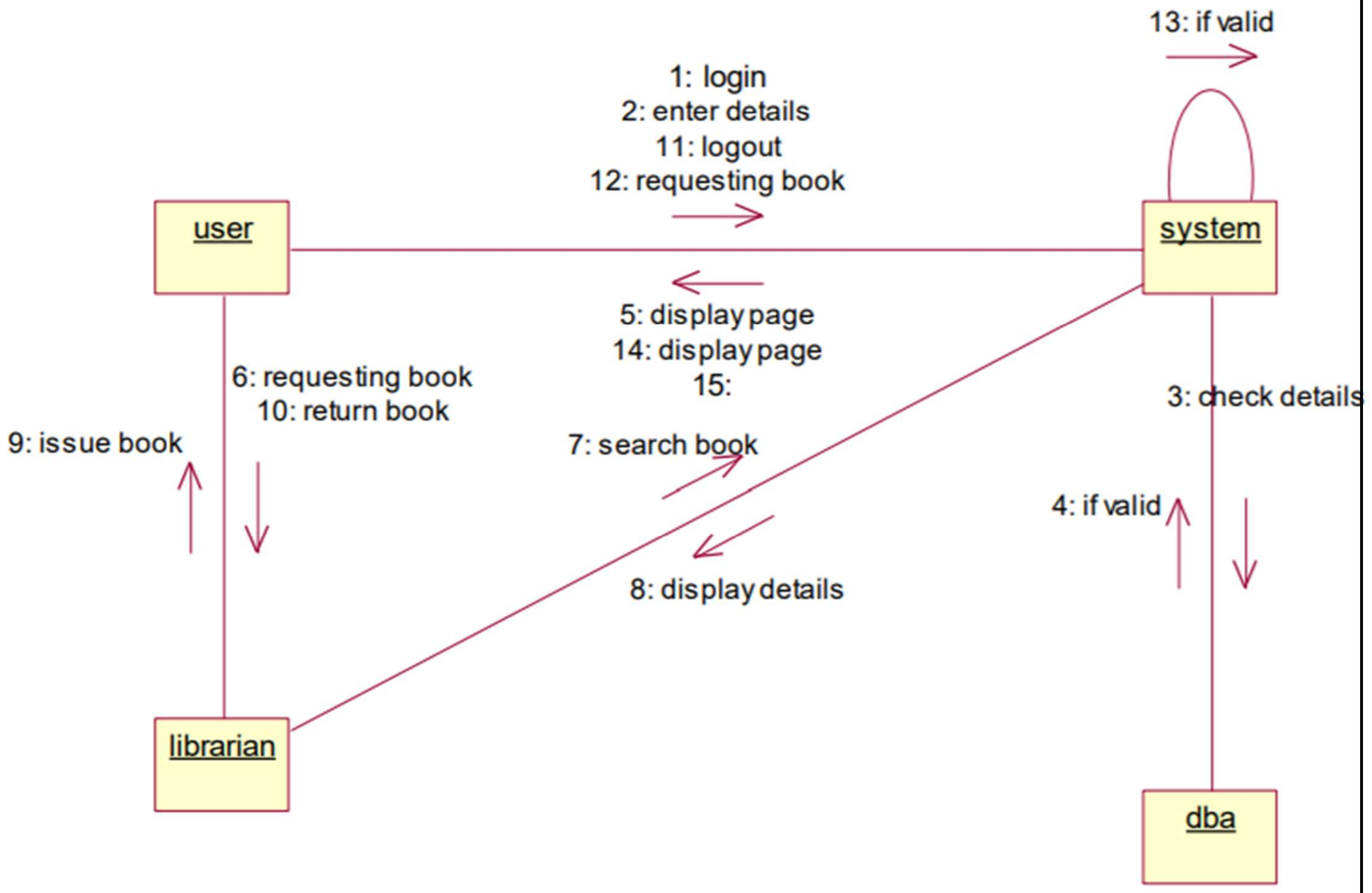
COMPONENT DIAGRAM :



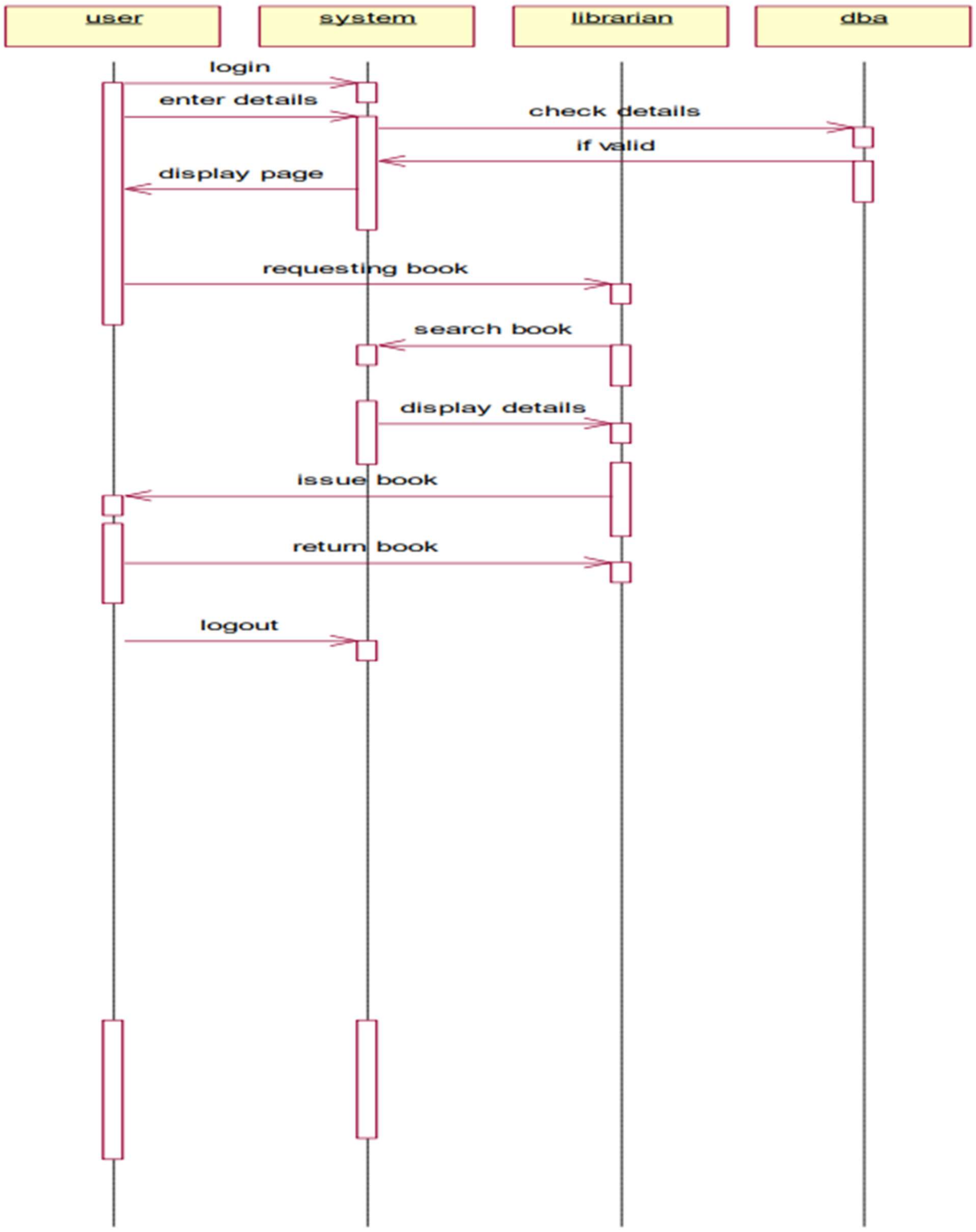
STATECHART DIAGRAM :



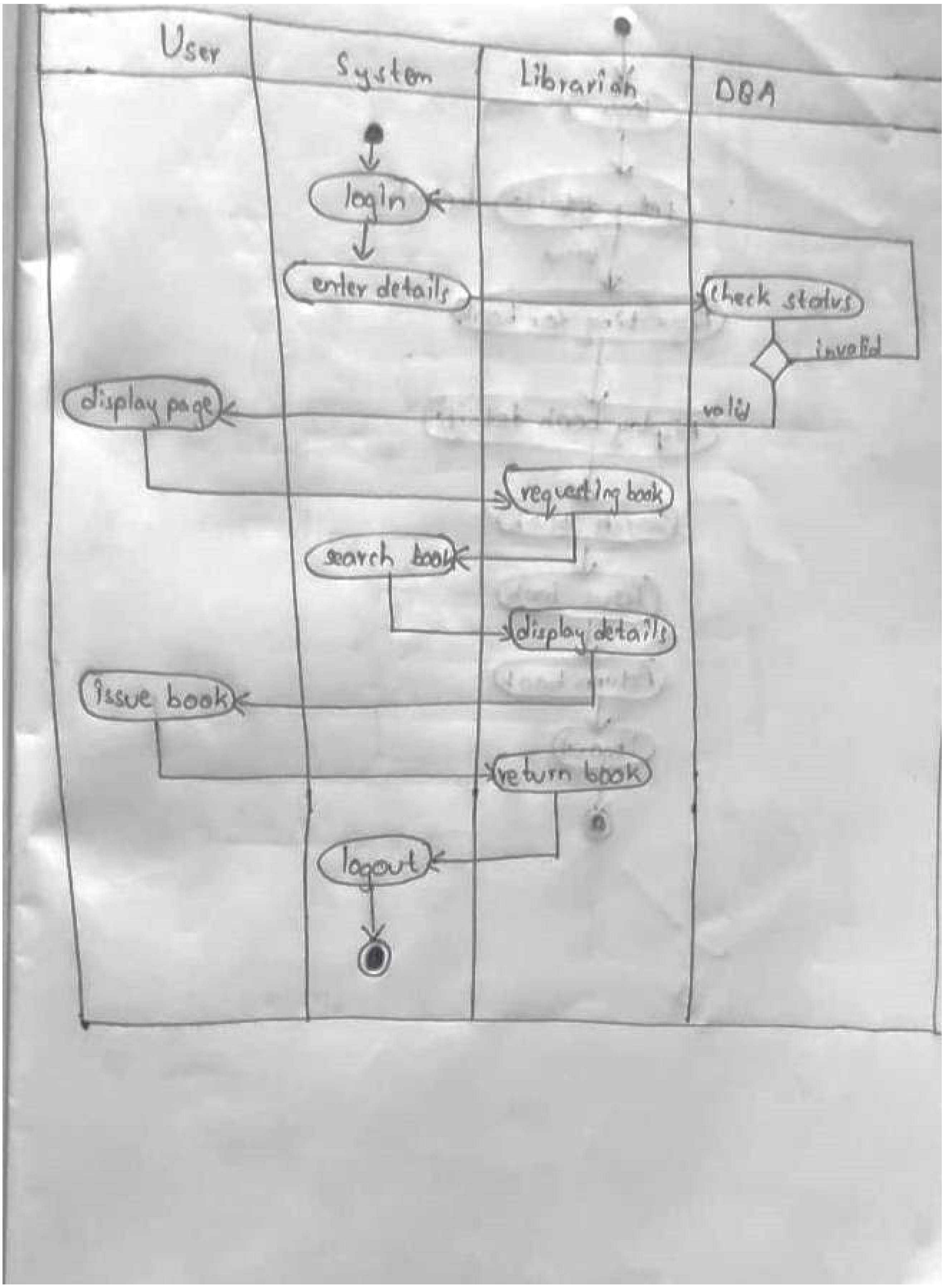
COLLABORATION DIAGRAM :



SEQUENCE DIAGRAM :



ACTIVITY DIAGRAM



User
System
Librarian
DBA
login
enter details
check status
invalid
valid
display page
requesting book
search book
display details
issue book
return book
logout