Graphic Era HILL UNIVERSITY
Established by an Act of the State Legislature of Uttarakhand (Adhiniyam Sankhya 12 of 2011)

Term Work on

Software Engineering Lab

(PCS-611)

(2022-2023)

Submitted To:

Mrs. Richa Gupta

Assistant Professor

GEHU, D. Dun

Submitted By:

Pranjali Kothari

Student ID: 20012810

CSE-B-VI Sem

Session: 2022-2023

ACKNOWLEDGEMENT

I would like to thank particularly our Faculty Mrs. Richa Gupta for her patience, support, and encouragement throughout the completion of this project and having faith in us.
I also acknowledge them who help us in developing thepractical file.
I wish to thank our parents for their continuing support and encouragement.
I also wish to thank them for providing us with the opportunity to reach this far in our studies.
At last, but not the least, I am greatly indebted to all other persons who directly or indirectly helped us during this work.

Graphic Era
HILL UNIVERSITY
Established by an Act of the State Legislature of Uttarakhand (Adhiniyam Sankhya 12 of 2011)

Table of Contents

Practical No. 01

OBJECTIVE: Identifying the Requirements from a Problem Statement.

THEORY:

Requirements identification is the first step of any software development project. Until the requirements of a client have been clearly identified, and verified, no other task (design, coding, testing) could begin.

Characteristics of Requirements:

1. **Unambiguity:** There should not be any ambiguity about what a system to be developed should do.
2. **Consistency:** The Requirement must be consistent for all users.
3. **Completeness:** A particular requirement for a system should specify what the system should do and what it should not.

Categories of Requirements:

1. **Based on Target Audience or Subject Matter:** Requirements are of 2 types:
   - **a. User requirements:** They are written in natural language so that both customers can verify their requirements have been correctly identified.
   - **b. System requirements:** They are written involving technical terms and/or specifications and are meant for the development or testing teams.

2. **Based on What they describe:** Requirements are of 2 types:
   - **a. Functional requirements (FRs):** These describe the functionality of a system -- how a system should react to a particular set of inputs and what should be the corresponding output.
   - **b. Non-functional requirements (NFRs):** They are not directly related to what functionalities are expected from the system. However, NFRs could typically define how the system should behave under certain situations.

     Non-functional requirements could be further classified into different types like:
     - **Product requirements:** For example, a specification that the web application should use only plain HTML, and no frames.
     - **Performance requirements:** For example, the system should remain available 24x7.
     - **Organizational requirements:** The development process should comply to SEI CMM level 4.

IDENTIFICATION OF FUNCTIONAL REQUIREMENTS:

Given a problem statement, the functional requirements could be identified by focusing on the following points:

- Identify the high-level functional requirements simply from the conceptual understanding of the problem.
- Identify the cases where an end user gets some meaningful work done by using the system.
- If we consider the system as a black box, there would be some inputs to it, and some output in return. This black box defines the functionalities of the system.
- Any high-level requirement identified could have different sub-requirements.

Case Study:

There's a Library Information System (LIS) for the benefit of students and employees of the institute. LIS will enable the members to borrow a book (or return it) with ease while sitting at his desk/chamber. The system also enables a member to extend the date of his borrowing if no other booking for that book has been made. For the library staff, this system aids them to easily handle day-to-day book transactions.
The librarian, who has administrative privileges and complete control over the system, can enter a new record into the system when a new book has been purchased, or remove a record in case any book is taken off the shelf. Any non-member is free to use this system to browse/search books online. However, issuing or returning books is restricted to valid users (members) of LIS only.
The final deliverable would a web application (using the recent HTML 5), which should run only within the institute LAN. Although this reduces security risk of the software to a large extent, care should be taken no confidential information (e.g., passwords) is stored in plain text.

- **Functional Requirements are as Follows:**
  a. **New user registration:** Any member of the institute who wishes to avail himself of the facilities of the library must register himself with the Library Information System. On successful registration, a user ID and password would be provided to the member. He must use these credentials for any future transaction in LIS**.**
  b. **Search book:** Any member of LIS can avail this facility to check whether any book is present in the institute's library. A book could be searched by its Title, Author's name or Publisher's Name.
  c. **User login:** A registered user of LIS can login to the system by providing his employee ID and password as set by him while registering. After successful login, the "Home" page for the user is shown from where he can access the different functionalities of LIS: search book, issue book, return book, reissue book. Any employee ID not registered with LIS cannot access the "Home" page -- a login failure message would be shown to him, and the login dialog would appear again.
  d. **Issue book:** Any member of LIS can issue a book against his account provided that the book is available in the library, no other member has currently issued a book, or the current user has not issued his/her maximum number of books.
  e. **Return book:** A book is issued for a finite time, which we assume to be a period of 20 days.
  f. **Reissue book:** Any member who has issued a book might find that his requirement is not over by 20 days. In that case, he might choose to reissue the book, and get permission to keep it for another 20 days. However, a member can reissue any book at most twice, after which he must return it.

- **Non-Functional Requirements are as Follows:**
  a. **Performance Requirements:** The system must remain available 24x7 and at least 50 users should be able to access it simultaneously.
  b. **Security Requirements:** The database of LIS should not store any password in plain text -- a hashed value must be stored.
  c. **Software Quality Attributes and Database Requirements.**
  d. **Design Constraints:** The LIS has to be developed as a web application using html 5 compatible for Firefox, Chrome, Safari, Opera.

Practical No. 02

OBJECTIVE: Estimation of Project Metrics

THEORY:

A software project is not just about writing a few hundred lines of source code to achieve a particular objective. The scope of a software project is comparatively *quite large*, and such a project could take several years to complete. However, the phrase "quite large" could only give some (possibly vague) qualitative information.

Some important project parameters that are estimated include:

- **Project size:** What would be the size of the code written say, in number of lines, files, modules?
- **Cost:** How much would it cost to develop a software? A software may be just pieces of code, but one has to pay to the managers, developers, and other project personnel.
- **Duration:** How long would it be before the software is delivered to the clients?
- **Effort:** How much effort from the team members would be required to create the software?

COCOMO:

COCOMO (Constructive Cost Model) was proposed by Boehm. According to him, there could be three categories of software projects: organic, semidetached, and embedded. The concept of organic, semidetached, and embedded systems is described below.

- **Organic:** A development project is said to be of organic type, if
  - The project deals with developing a well understood application.
  - The development team is small.
  - The team members have prior experience in working with similar types of projects.
- **Semidetached:** A development project can be categorized as semidetached type, if
  - The team consists of some experienced as well as inexperienced staff.
  - Team members may have some experience on the type of system to be developed.
- **Embedded:** Embedded type of development project are those, which
  - Aims to develop a software strongly related to machine hardware.
  - Team size is usually large.

Basic Cocomo Model:

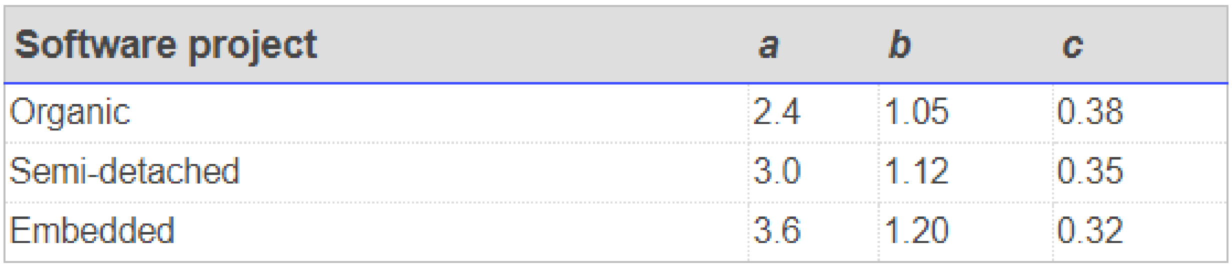
The basic COCOMO model helps to obtain a rough estimate of the project parameters. It estimates effort and time required for development in the following way:

Effort = a * (KDSI)^b PM Tdev = 2.5 * (Effort)^c Months where

- KDSI is the estimated size of the software expressed in Kilo Delivered Source Instructions
- a, b, c are constants determined by the category of software project.
- Effort denotes the total effort required for the software development, expressed in person months (PMs)

- Tdev denotes the estimated time required to develop the software (expressed in months)

The value of the constants a, b, c are given below:

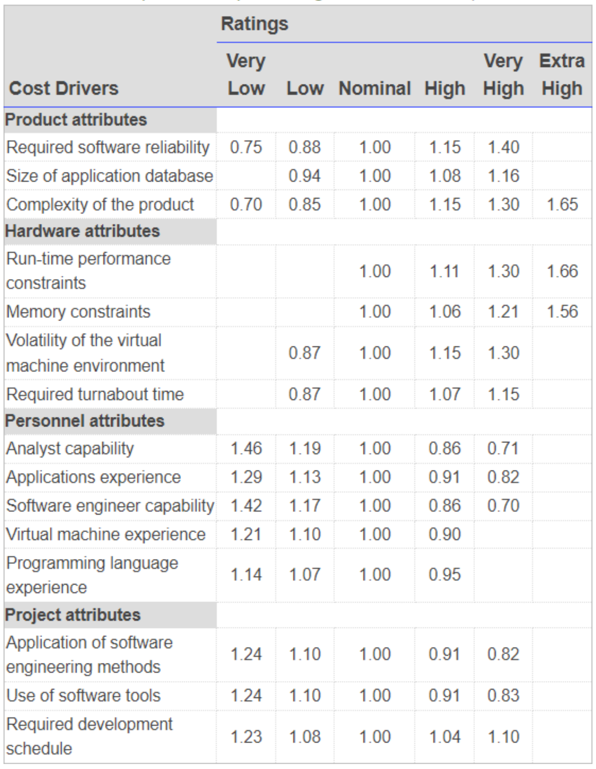


Software project	a	b	c
Organic	2.4	1.05	0.38
Semi-detached	3.0	1.12	0.35
Embedded	3.6	1.20	0.32

Intermediate Cocomo Model:

The basic COCOMO model considers that effort and development time depends only on the size ofthe software. However, in real life there are many other project parameters that influence the development process. The intermediate COCOMO take those other factors into consideration by defining a set of 15 cost drivers (multipliers) as shown in the table below [i]. Thus, any project thatmakes use of modern programming practices would have lower estimates in terms of effort and cost. Each of the 15 such attributes can be rated on a six-point scale ranging from "very low" to "extra high" in their relative order of importance. Each attribute has an effort multiplier fixed as per the rating. The product of effort multipliers of all the 15 attributes gives the **Effort Adjustment Factor (E** EAF is used to refine the estimates obtained by basic COCOMO as follows:

Cost drivers for INtermediate COCOMO (Source: http://en.wikipedia.org/wiki/COCOMO)



Cost Drivers	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
Required software reliability	0.75	0.88	1.00	1.15	1.40	
Size of application database		0.94	1.00	1.08	1.16	
Complexity of the product	0.70	0.85	1.00	1.15	1.30	1.65
Hardware attributes						
Run-time performance constraints			1.00	1.11	1.30	1.66
Memory constraints			1.00	1.06	1.21	1.56
Volatility of the virtual machine environment		0.87	1.00	1.15	1.30	
Required turnabout time		0.87	1.00	1.07	1.15	
Personnel attributes						
Analyst capability	1.46	1.19	1.00	0.86	0.71	
Applications experience	1.29	1.13	1.00	0.91	0.82	
Software engineer capability	1.42	1.17	1.00	0.86	0.70	
Virtual machine experience	1.21	1.10	1.00	0.90		
Programming language experience	1.14	1.07	1.00	0.95		
Project attributes						
Application of software engineering methods	1.24	1.10	1.00	0.91	0.82	
Use of software tools	1.24	1.10	1.00	0.91	0.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

$$\boldsymbol{Effort_{corrected} = Effort * EAF}$$

$$\boldsymbol{Tdev|_{corrected} = 2.5 * (Effort|_{corrected})^{c}}$$

AF

Complete Cocomo Model:

Both the basic and intermediate COCOMO models consider a software to be a single homogeneous entity -- an assumption, which is rarely true. In fact, many real-life applications are made up of several smaller sub-systems. (One might not even develop all the sub-systems -- just use the available services). The complete COCOMO model takes these factors into account to provide a far more accurate estimate of project metrics.

Halstead's Complexity Metrics:

Halstead took a linguistic approach to determine the complexity of a program. According to him, a computer program consists of a collection of different operands and operators. The definition of operands and operators could, however, vary from one person to another and one programming language to other. Any given program has the following four parameters:

- **n1**: Number of unique operators used in the program.
- **n2**: Number of unique operands used in the program.
- **N1**: Total number of operators used in the program.
- **N2**: Total number of operands used in the program.

Using the above parameters one compute the following metrics:

- **Program Length**: $N = N1 + N2$
- **Program Vocabulary**: $n = n1 + n2$
- **Volume**: $V = N * \lg n$
- **Difficulty**: $D = (n1 * N2) / (2 * n2)$
- **Effort**: $E = D * V$
- **Time to Implement**: $T = E / 18$ (in seconds)

Case Study:

The SE VLabs Institute has been recently setup to provide state-of-the-art research facilities in the field of Software Engineering. Apart from research scholars (students) and professors, it also includes quite many employees who work on different projects undertaken by the institution.
As the size and capacity of the institute is increasing with the time, it has been proposed to develop a Library Information System (LIS) for the benefit of students and employees of the institute. LIS will enable the members to borrow a book (or return it) with ease while sitting at his desk/chamber. The system also enables a member to extend the date of his borrowing if no other booking for that book has been made. For the library staff, this system aids them to easily handle day-to-day book transactions. The librarian, who has administrative privileges and complete control over the system, can enter a new record into the system when a new book has been purchased, or remove a record in case any book is taken off the shelf. Any non-member is free to use this system to browse/search books online. However, issuing or returning books is restricted to valid users (members) of LIS only.
The final deliverable would a web application (using the recent HTML 5), which should run only within the institute LAN. Although this reduces security risk of the software to a large extent, care should be taken no confidential information (e.g., passwords) is stored in plain text.

For organic category of project, the values of a, b, c are 2.4, 1.05, 0.38 respectively. So, the projected effort required for this project becomes:

Effort = $2.4 * (10)^{1.05}$ PM

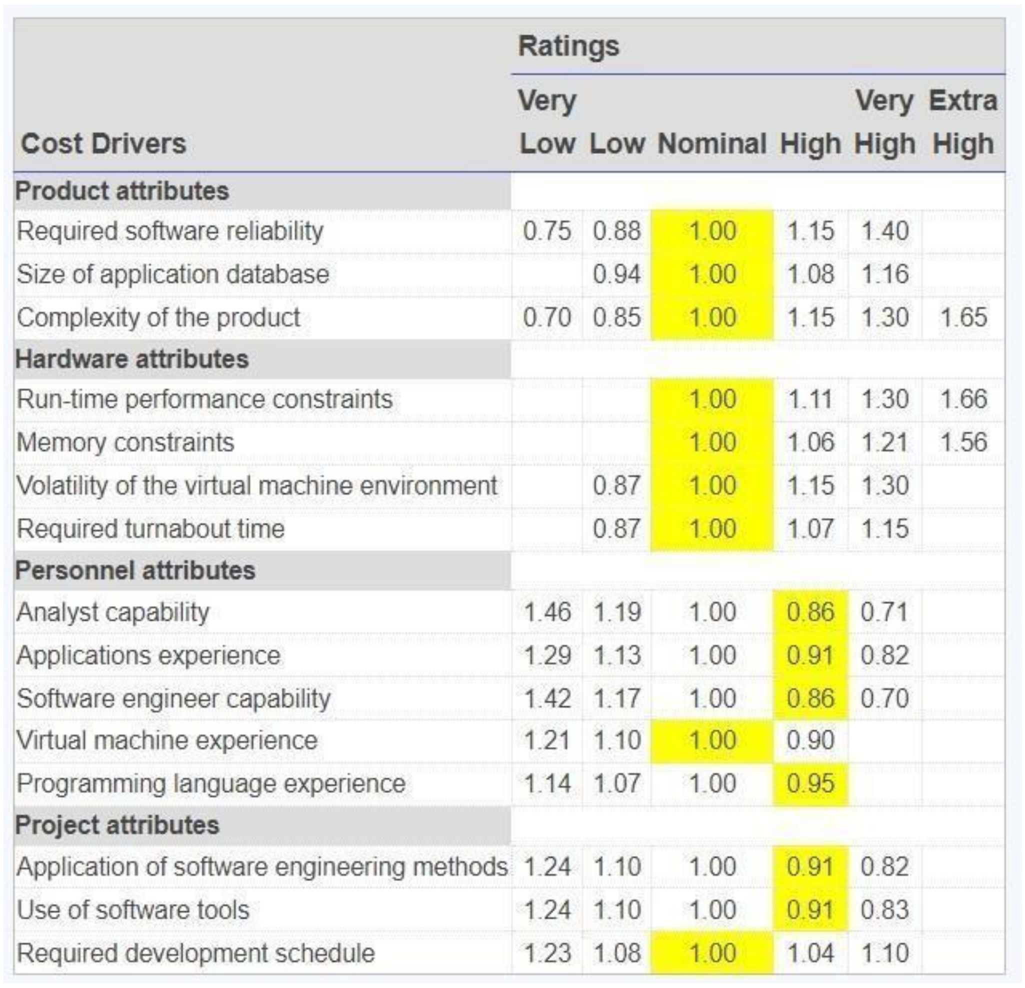
= 27 PM (approx.)

So, around 27 person-months are required to complete this project. With this calculated value for effort, we can also approximate the development time required:

Tdev = $2.5 * (27)^{0.38}$ Months

= 8.7 Months (approx.)

Intermediate Cocomo Model:



Cost Drivers	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
Required software reliability	0.75	0.88	1.00	1.15	1.40	
Size of application database		0.94	1.00	1.08	1.16	
Complexity of the product	0.70	0.85	1.00	1.15	1.30	1.65
Hardware attributes						
Run-time performance constraints			1.00	1.11	1.30	1.66
Memory constraints			1.00	1.06	1.21	1.56
Volatility of the virtual machine environment		0.87	1.00	1.15	1.30	
Required turnabout time		0.87	1.00	1.07	1.15	
Personnel attributes						
Analyst capability	1.46	1.19	1.00	0.86	0.71	
Applications experience	1.29	1.13	1.00	0.91	0.82	
Software engineer capability	1.42	1.17	1.00	0.86	0.70	
Virtual machine experience	1.21	1.10	1.00	0.90		
Programming language experience	1.14	1.07	1.00	0.95		
Project attributes						
Application of software engineering methods	1.24	1.10	1.00	0.91	0.82	
Use of software tools	1.24	1.10	1.00	0.91	0.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

The cells with yellow backgrounds highlight our choice of weight for each of the cost drivers. EAF is determined by multiplying all the chosen weights. So, we get:

EAF = 0.53 (approx.)

Using this EAF value we refine our estimates from basic COCOMO as shown below:

$\text{Effort}_{\text{corrected}}$ = Effort * EAF

= 27 * 0.53

= 15 PM (approx.)

$\text{Tdev}|_{\text{corrected}} = 2.5 * (\text{Effort}_{\text{corrected}})^{c}$

= $2.5 * (15)^{0.38}$

= 7 months (approx.)

Practical No. 03

OBJECTIVE: Modeling UML Use Case Diagrams and Capturing Use Case Scenarios.

THEORY:

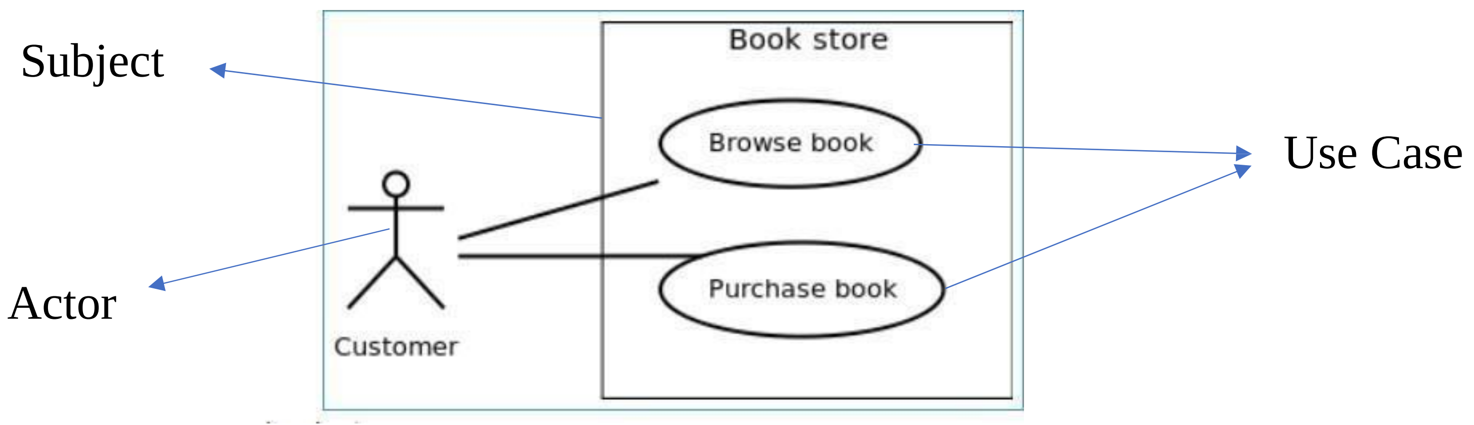
Use case diagrams belong to the category of behavioral diagram of UML diagrams. Use case diagrams aim to present a graphical overview of the functionality provided by the system.

Components of Use Case Diagrams:

1. **Actors:** An actor can be defined as an object or set of objects, external to the system, which interacts with the system to get some meaningful work done. Actors could be human, devices, or even other systems. Actors can be classified into 2 types:
   a. **Primary Actor:** They are principal users of the system, who fulfill their goal by availing some service from the system.
   b. **Supporting Actor:** They render some kind of service to the system.

2. **Use Case:** A use case is simply a functionality provided by a system.

3. **Subject:** Subject is simply the system under consideration. Use cases apply to a subject.

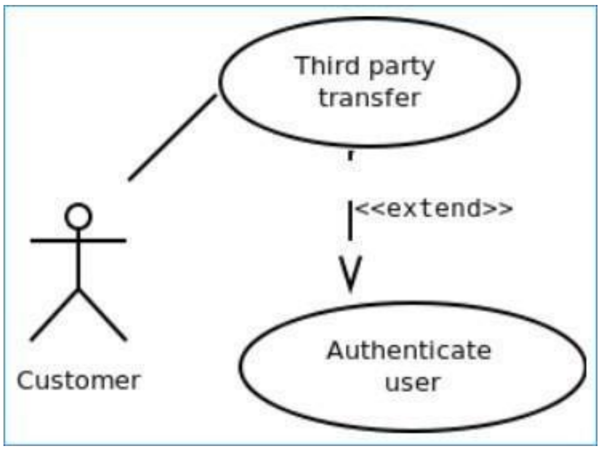


E.g.: A Use Case Diagram for a Bookstore.

Use Case Relationships:

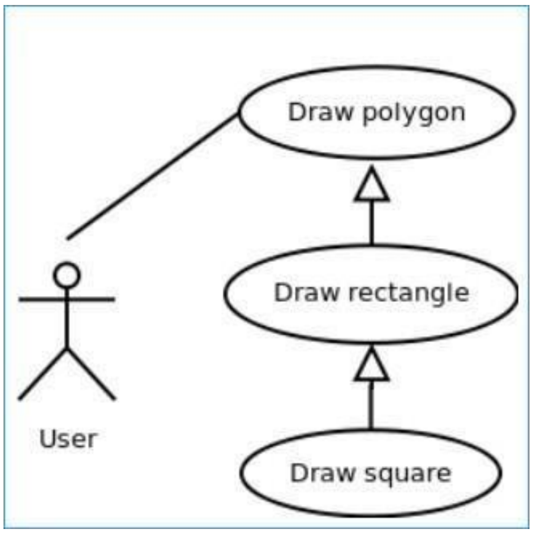
1. **Include Relationship:** Include relationships are used to depict common behaviors that are shared by multiple use cases.
   Include relationship is depicted by a dashed arrow with a «include» stereotype from the including use case to the included use case.

2. **Extend Relationship:** Use case extensions are used to depict any variation to an existing use case. They are used to specify the changes required when any assumption made by the existing use case becomes false.
   Extend relationship is depicted by a dashed arrow with a «extend» stereotype from the extending use case to the extended use case.



E.g.: A Use case Diagram for authentication with Extend relationship

3. **Generalization Relationship**: Generalization relationships are used to represent the inheritance between use cases. A derived use case specializes in some functionality it has already inherited from the base use case.
   Generalization relationship is depicted by a solid arrow from the specialized (derived) use case to the more generalized (base) use case.



E.g.: A Use case Diagram for drawing shapes with Generalization relationship

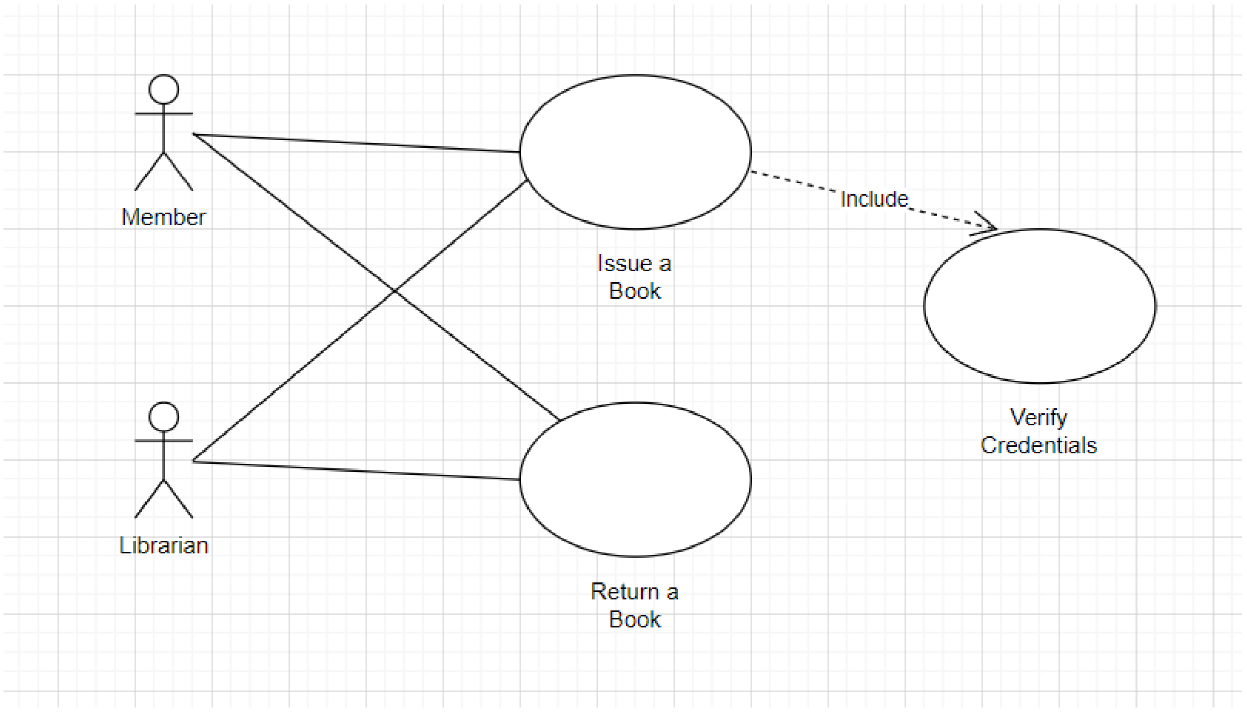
Case Study:

Consider a library, where a member can perform two operations: issue a book and return it. A book is issued to a member only after verifying his credentials. Draw a use case diagram for the problem.

- **Identification of Actors:**
  Considering the above problem statement, there can be 2 actors in the use case diagram:
  a. Member b. Librarian

- **Identification of Use Cases:**
  There are 3 use cases in the above problem statement:
  a. Issue a Book b. Return a Book c. Verify Credentials.

Thus, the final Use Case Diagram for the problem statement is as follows:



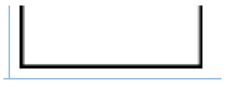
Practical No. 04

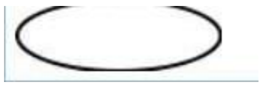
OBJECTIVE: Draw E-R(Entity Relationship) Model from the given Problem Statement.

THEORY:

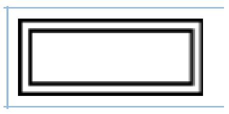
Entity-Relationship model is used to represent a logical design of a database to be created. In ER model, real world objects (or concepts) are abstracted as entities, and different possible associations among them are modeled as relationships.

Components of E-R Model:

1. **Entity Set and Relationship Set:** An Entity set is a collection of all similar entities, and a Relationship set is a set of similar relationships.
   Notation:
   
   The name of the entity is written inside the rectangle.

2. **Attributes:** Attributes are the characteristics describing any entity belonging to an entity set. Any entity in a set can be described by zero or more attributes.
   Notation:
   
   Name of the attribute is written inside the ellipse

3. **Keys:** The attributes in an entity set can be defined with the help of the following keys:
   a. **Primary Key:** It states that the attribute must be unique and can't be left un-assigned. The attribute which is to be made primary key is **underlined**.
   b. **Super Key:** One or more attributes, which when taken together, helps to uniquely identify an entity in an entity set.
   c. **Candidate key:** It is a minimal subset of a super key**.**
   d. **Prime Attribute:** Any attribute taking part in Super Key.

4. **Weak Entity:** An entity set is said to be weak if it is dependent upon another entity set. A weak entity can't be uniquely identified only by it's attributes. In other words, it doesn't have a super key.
   Notation:
   

5. **Cardinality and it's Mapping:** Cardinality represents the number of times an entity of an entity set participates in a relationship set. Based on the number of entities in E1 and E2 are associated with, we can have the following four type of mappings:
   a. **One to One:** An entity in E1 is related to at most a single entity in E2, and vice versa.
   b. **One to Many:** An entity in E1 could be related to zero or more entities in E2. Any entity in E2 could be related to at most a single entity in E1.
   c. **Many to One:** Zero or more number of entities in E1 could be associated to a single entity in E2. However, an entity in E2 could be related to at most one entity in E1.
   d. **Many to Many:** Any number of entities could be related to any number of entities in E2, including zero, and vice versa.

Case Study:

SE VLabs Inc. is a young company with a few departments spread across the country. As of now, the company has a strength of 200+ employees.
Each employee works in a department. While joining, a person must provide a lot of personal and professional details including name, address, phone Number, mail address, date of birth, and so on. Once all these information are furnished, a unique ID is generated for each employee. He is then assigned a department in which he will work.
There are around ten departments in the company. Unfortunately, two departments were given same names. However, departments too have ID's, which are unique.

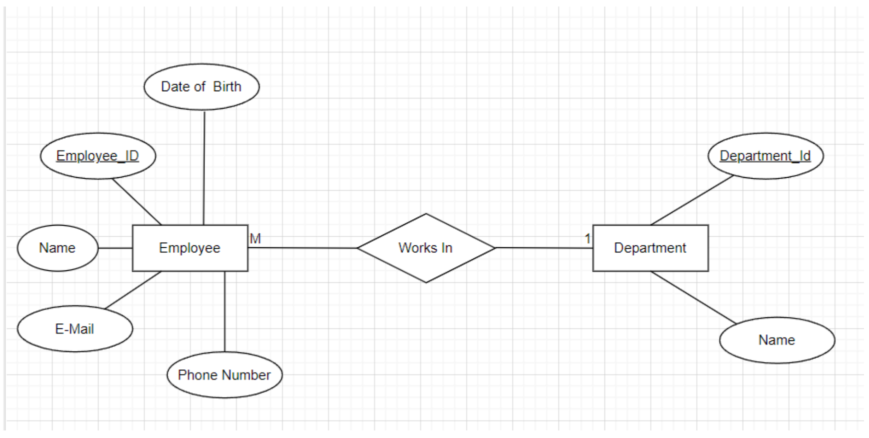
- **Identification of Entity Sets:**
  Considering the above problem statement, there can be 2 Entity Sets in the E-R diagrams:
  a. Employee b. Department

- **Identification of Attributes:**
  a. Employee: The Employee Entity set has 5 Attributes:
    - ➢ Employee_Id (Primary Key)
    - ➢ Name
    - ➢ Phone umber
    - ➢ E-mail Address
    - ➢ Date of birth

  b. Department: The Department Entity set has 2 Attributes:
    - ➢ Department_Id (Primary Key)
    - ➢ Name

The Employee and Department Entity Set are related to each other by a Relationship set with many to one Relation.

Thus, the final E-R Diagram for the problem statement is as follows:



Practical No. 05

OBJECTIVE: Identifying Domain Classes from the Problem Statements.

THEORY:

Domain Model, as a conceptual model, gives proper understanding of problem description through its highly effective component – the Domain Classes. Domain classes are the abstraction of key entities, concepts or ideas presented in the problem statement. Domain classes are used for representing business activities during the analysis phase.

Techniques to Identify Domain Classes:

1. **Grammatical Approach using Nouns:** This technique involves grammatical analysis of the problem statement to identify list of potential classes. The logical steps are:
   - Obtain the user requirements (problem statement) as a simple, descriptive English text.
   - Identify and mark the nouns, pronouns, and noun phrases from the above problem statements.
   - List of potential classes is obtained based on the category of the nouns.

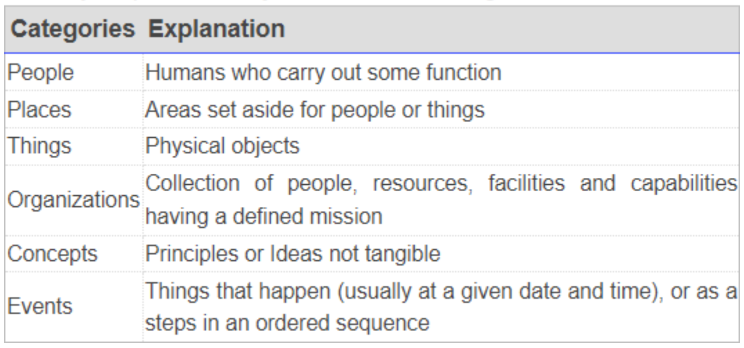
Advantages: This is one of the simplest approaches that could be easily understood and applied by a larger section of the user base.
Disadvantages: The problem statement may not always help towards correct identification of a class. At times it could give us redundant classes.

2. **Using Generalization:** In this approach, all potential objects are classified into different groups based on some common behavior. Classes are derived from these groups.

3. **Using Subclass:** Here, instead of identifying objects one goes for identification of classes based on some similar characteristics. These are the specialized classes. Common characteristics are taken from them to form the higher-level generalized classes.

Steps to Identify Domain Classes from Problem Statement:

a. Make a list of potential objects by finding out the nouns and noun phrases from narrative problem statement.
b. Apply subject matter expertise (or domain knowledge) to identify additional classes.
c. Filter out the redundant or irrelevant classes.
d. Classify all potential objects based on categories as per the following table:



Categories	Explanation
People	Humans who carry out some function
Places	Areas set aside for people or things
Things	Physical objects
Organizations	Collection of people, resources, facilities and capabilities having a defined mission
Concepts	Principles or Ideas not tangible
Events	Things that happen (usually at a given date and time), or as a steps in an ordered sequence

e. Group the objects based on similar attributes**.**
f. Give related names to each group to generate the final list of top-level classes.
g. Iterate over to refine the list of classes

Case Study:

Mr. Bose is the boss of this agency. Cabs are solely owned by the agency. They hire drivers to drive the cabs. Most of the cabs are without AC. However, a few comes with AC.
The agency provides service from 8 AM to 8 PM. Presently the service is limited only within Kolkata. Whenever any passenger books a cab, an available cab is allocated for him. A booking receipt is given to the passenger. He is then dropped to his home, office, or wherever he wants to go. In case the place is in too interior, the passenger is dropped at the nearest landmark.
Payments are made to the drivers by cheque drawn at the local branch of At Your Risk Bank. All kind of finances required for the business are dealt with this bank.
Recently Mr. Roy, neighbour of Mr. Bose, has given a proposal to book one of the cab in the morning every day to drop his son to school, and drop him back to home later. Few other persons in the locality have also found the plan a good one. Hence, Mr. Bose is planning to introduce this "Drop to school" plan also very soon.

- **Identification of noun and Noun Phrases:**
  From the given problem statement, we can identify the following nouns and noun phrases:

Mr Bose, boss, agency, Cabs, drivers, AC, service, Kolkata, passenger, booking receipt, home, office, place, landmark, Payments, cheque, branch, At Your Risk, Bank, finances, business, bank, Mr Roy, morning, every day, son, school, persons, locality, plan, drop to school, neighbour.

We can put these in the following classes:

People	**Places**	**Things**	**Organizations**	**Concepts**	**Events**
Mr. Bose	Kolkata	Cabs	Agency	Service	Morning
Boss	Home	AC	At Your Own Risk	Payments	Everyday
Drivers	Office	Booking-	-Bank	Finances	Drop to School
Passengers	Place	Receipt	Bank	Business	
Mr. Roy	Landmark	Cheque	School	Plan	
Son	Branch				
Persons	Locality				
Neighbor					

Using the above Data, we can make out these classes with their associated properties:

Person	**Employee**	**Customer**	**Place**	**Cab**	**Booking**	**Bank**
Address	Address	Address	Address	Color	Time	Address
Height	Height	Height		Cost	Customer	Branch
Name	Name	Name		Has AC	Starting Place	Name
Weight	Weight	Weight		Weight		
	Emp_ID	Customer_ID				

Practical No. 06

OBJECTIVE: State chart and Activity Modeling from the given problem Statements.

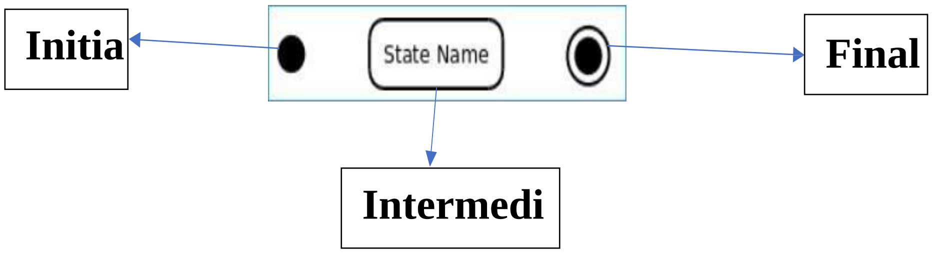
THEORY:

State Diagrams:

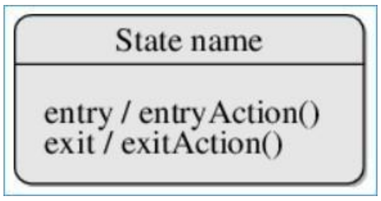
In the case of Object-Oriented Analysis and Design, a system is often abstracted by one or more classes with some well-defined behavior and states. A state chart diagram is a pictorial representation of such a system, with all its states, and different events that lead transition from one state to another.

Components of State Chart Diagram:

1. **State:** A state is any "distinct" stage that an object (system) passes through in its lifetime. An object remains in each state for a finite time until "something" happens, which makes it move to another state. All states are broadly classified into 3 types:
   a) **Initial:** The state in which an object remains when created.
   b) **Final:** The state from which an object do not move to any other state.
   c) **Intermediate:** Any state, which is neither initial, nor final.

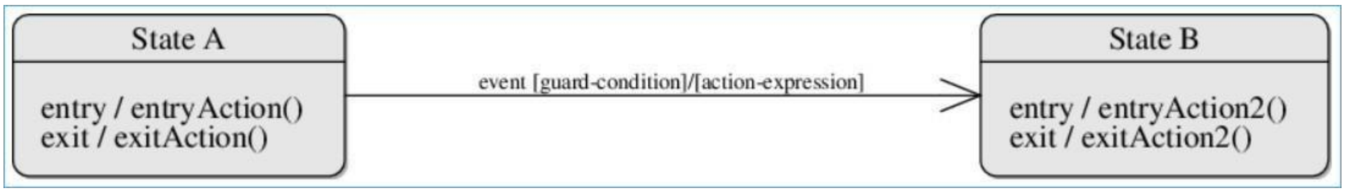


The Intermediate State has 2 compartments namely, **A Name Compartment** and **A Internal transitions compartment.**
The internal activities are indicated using the following syntax: **action-label / action-expression. Action Label are 4 conditions:**
   a) **Entry:** Indicates activity performed when the system enters this state.
   b) **Exit:** Indicates activity performed when the system exits this state.
   c) **Do:** Indicate any activity that is performed while the system remain in this state or until the action expression results in a completed computation.
   d) **Include:** Indicates invocation of a sub-machine.



2. **Transition:** Transition is movement from one state to another state in response to an external stimulus. A transition is represented by a solid arrow from the current state to the next state. It is labeled by: **event [guard-condition]/[action-expression],** where:
   a) **Event:** is the what is causing the concerned transition (mandatory**) .**
   b) **Guard-Condition** is (are) precondition(s), which must be true for the transition to happen [optional].
   c) **Action-Expression: i**ndicate action(s) to be performed as a result of the transition

3. **Action:** Actions represent behaviour of the system. While the system is performing any action for the current event, it doesn't accept or process any new event. The order in which different actions are executed, is given below:
    a) **Exit actions of the present state.**
    b) **Actions specified for the transition.**
    c) **Entry actions of the next state.**

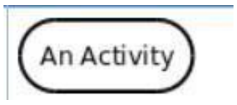


Activity Diagrams:

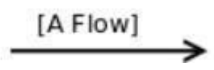
Activity diagrams fall under the category of behavioral diagrams in Unified Modeling Language. It is a high-level diagram used to visually represent the flow of control in a system. It has similarities with traditional flow charts. However, it is more powerful than a simple flow chart since it can represent various other concepts like concurrent activities, their joining, and so on.

Components of Activity Diagram:

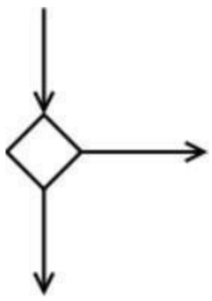
1. **Activity:** An activity denotes a particular action taken in the logical flow of control. This could simply be invocation of a mathematical function, alter an object's properties and so on.
   An activity is represented with a rounded rectangle.



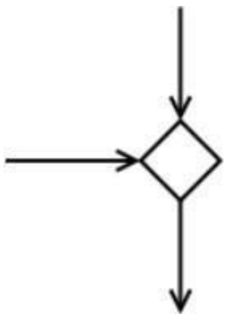
2. **Flow:** A flow is represented with a directed arrow. This is used to depict transfer of control from one activity to another, or to other types of components, as we will see below. A flow is often accompanied by a label, called the guard condition, indicating the necessary condition for the transition to happen. The syntax to depict it is **[guard condition].**



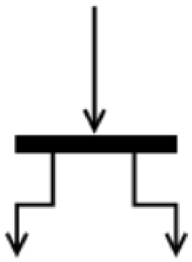
3. **Decision:** A decision node, represented with a diamond, is a point where a single flow enters, and two or more flows leave. The control flow can follow only one of the outgoing paths. The outgoing edges often have guard conditions indicating true-false or if-then-else conditions. However, they can be omitted in obvious cases.



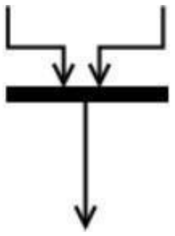
4. **Merge:** This is represented with a diamond shape, with two or more flows entering, and a single flow leaving out. A merger node represents the point where at least a single control should reach before further processing could continue.



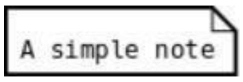
5. **Fork:** Fork is a point where parallel activities begin. A fork is graphically depicted with a black bar, with a single flow entering and multiple flows leaving out.



6. **Join:** A join is depicted with a black bar, with multiple input flows, but a single output flow. Physically it represents the synchronization of all concurrent activities. Unlike a merge, in case of a join all the incoming controls must be completed before any further progress could be made.



7. **Note:** UML allows attaching a note to different components of a diagram to present some textual information. The information could simply be a comment or maybe some constraint.



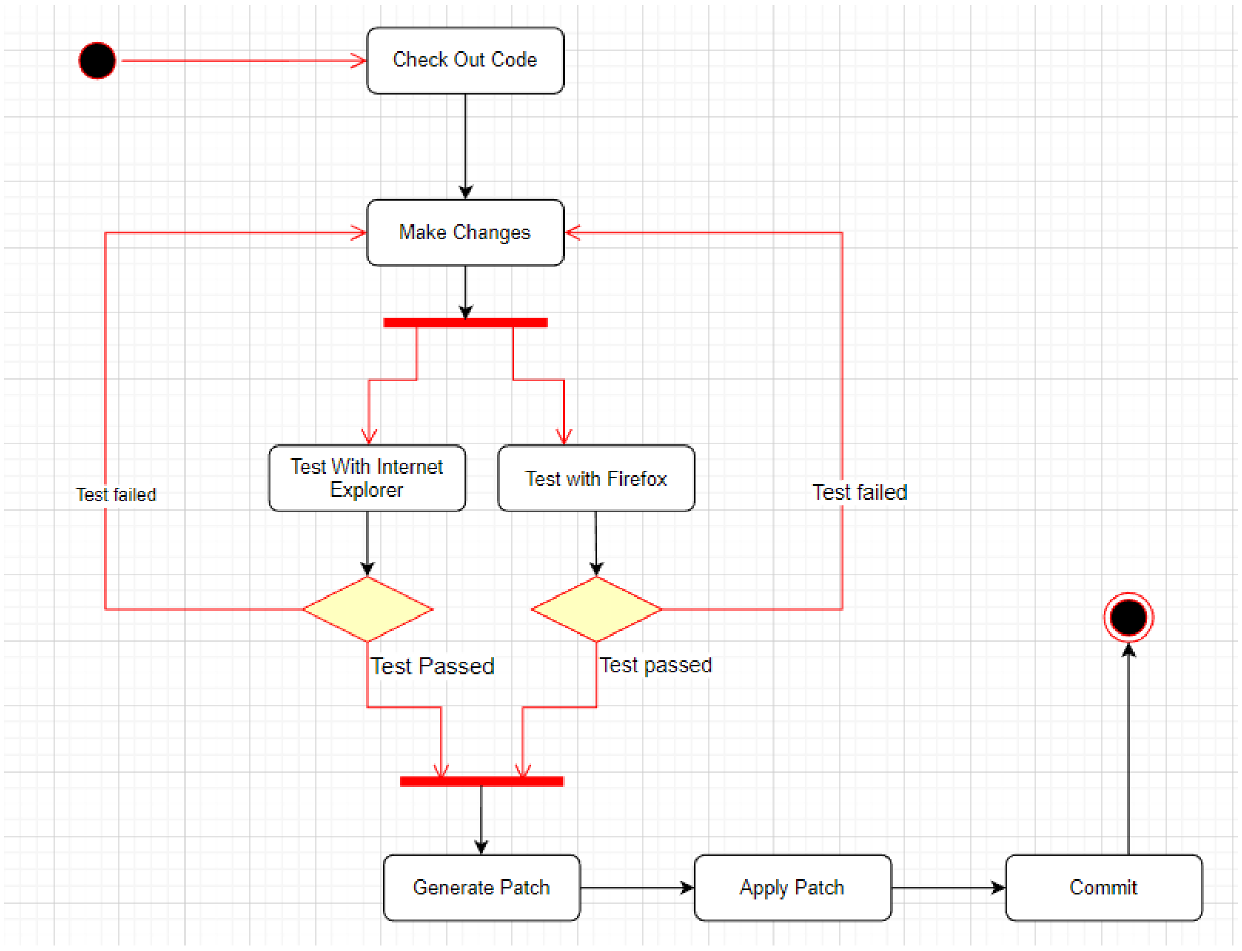
8. **Partition:** Different components of an activity diagram can be logically grouped into different areas, called partitions or swim lanes. They often correspond to different units of an organization or different actors.

<u>Case Study:</u>

Draw an activity diagram to graphically represent the following workflow.
Let us consider the development activities of SE Virtual Labs. The process begins by checking out the code from Subversion repository. Necessary modifications are then made to the checked-out code(local copy). Once the developer is done with his changes, the application must be tested to verify whether the new functionality is working fine. This test must be performed with two of the more popular web browsers: Firefox and Internet Explorer, to support cross-browser accessibility. If testing fails in at least one of the two browsers, developer goes back to his code, and fixes it. Only when all the browsers pass the test, a patch is generated from the local copy, and applied to the production code. The local copy is then committed resulting in update of the SVN repository. Note that, if the local copy is committed before generating a patch file, then local changes would get registered, and one won't be further able to generate the patch file.

There are 6 seven states associated with the Problem statement namely Checkout Code, Make Changes, testing withbrowsers, generate patch, apply Patch and Commit.
Testing with browser will be included inside a fork as per the problem statement, if any of the test with any one of the browsers fails, the developer needs to go back and make changes.

Thus, the Activity diagram of the following problem statement is:



Practical No. 07

OBJECTIVE: Modelling UML Class Diagrams and Sequence diagrams

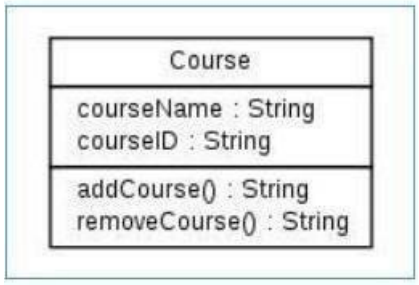
THEORY:

Class Diagrams:

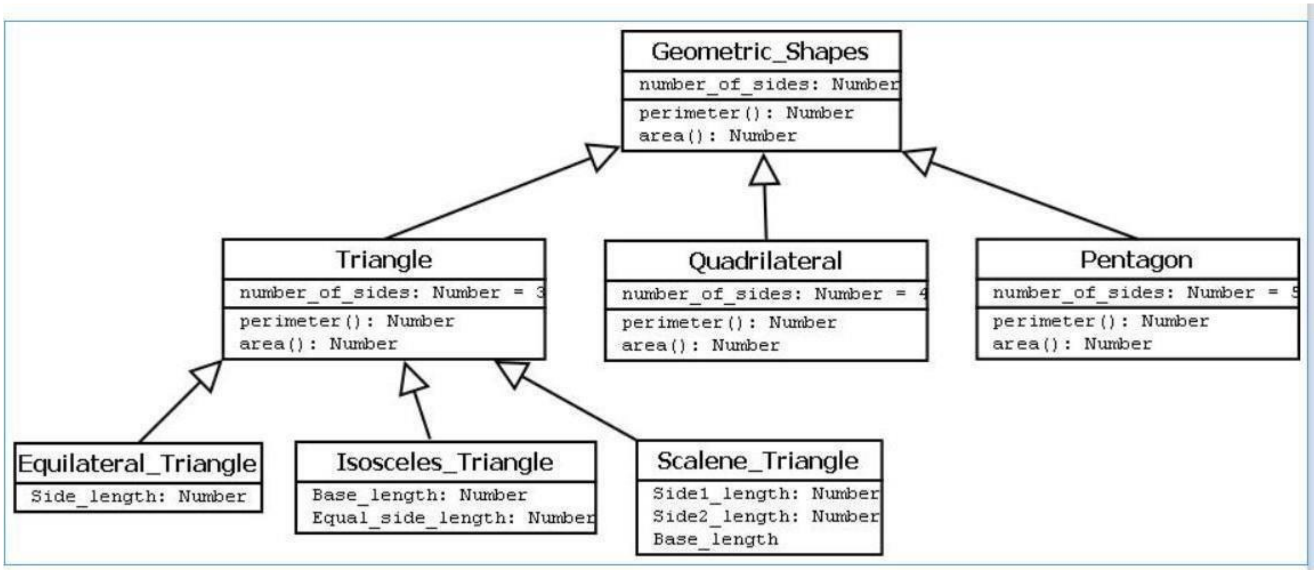
It is a graphical representation for describing a system in context of its static construction.

Elements in Class Diagram:

1. **Class:** A set of objects containing similar data members and member functions is described by a class. In UML syntax, class is identified by solid outline rectangle with three compartments which contain:
   a) **Class Name:** A class is uniquely identified in a system by its name. A textual string is taken as class name. It lies in the first compartment in the class rectangle.
   b) **Attribute:** Property shared by all instances of a class. It lies in the second compartment in the class rectangle.
   c) **Operations:** An execution of an action can be performed for any object of a class. It lies in the last compartment in the class rectangle.

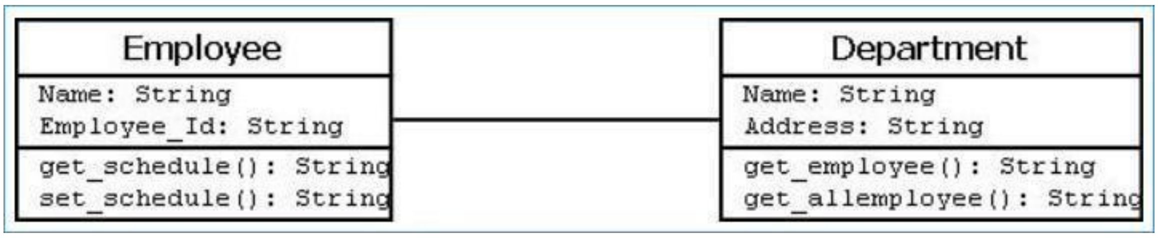


   d) **Generalization/Specialization:** It describes how one class is derived from another class. The derived class inherits the properties of its parent class.

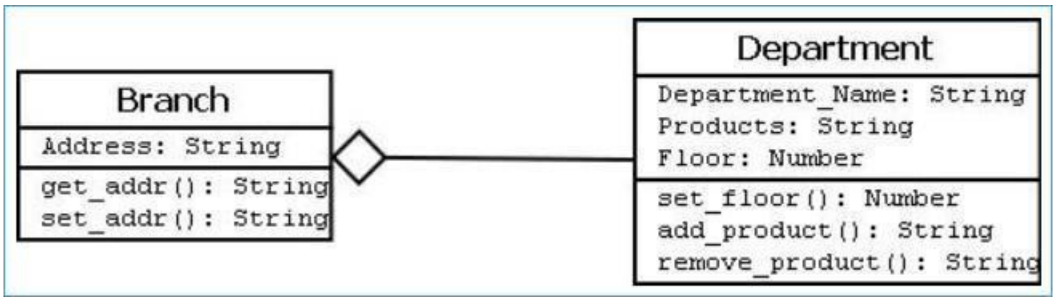


2. Relationships: Existing relationships in a system describe legitimate connections between the classes in that system.

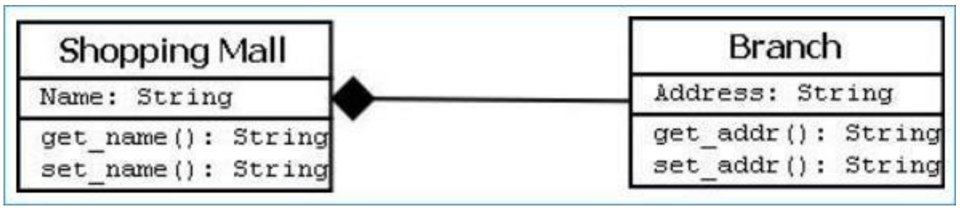
- **Association:** It is an instance level relationship that allows exchanging messages among the objects of both ends of association. A simple straight line connecting two class boxes represent an association.



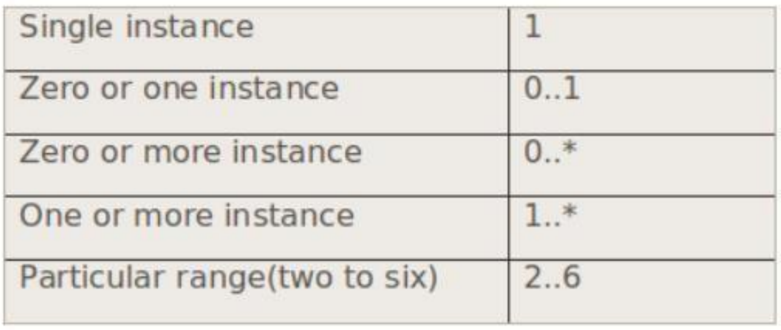
- **Aggregation:** It is a special form of association which describes a part-whole relationship between a pair of classes. It means, in a relationship, when a class holds some instances of related class, then that relationship can be designed as an aggregation.



- **Composition:** It is a strong form of aggregation which describes that whole is completely owns its part. The life cycle of the part depends overall.



- **Multiplicity:** It describes how many numbers of instances of one class are related to the number of instances of another class in an association.



Single instance	1
Zero or one instance	0..1
Zero or more instance	0..*
One or more instance	1..*
Particular range(two to six)	2..6

Sequence Diagrams:

It represents the behavioral aspects of a system. Sequence diagram shows the interactions between the objects by means of passing messages from one object to another with respect to time in a system.

Elements in Sequence Diagram:

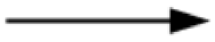
1. **Object:** Objects appear at the top portion of sequence diagram. Object is shown in a rectangle box. Name of object precedes a colon ':' and the class name, from which the object is instantiated. The whole string is underlined and appears in a rectangle box.

2. **Life-Line Bar:** A downward vertical line from object-box is shown as the lifeline of the object. A rectangle bar on lifeline indicates that it is active at that point of time.

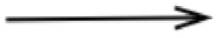
3. **Messages:** Messages are shown as an arrow from the lifeline of sender object to the lifeline of receiver object and labeled with the message name. The chronological order of the messages passing throughout the objects' lifeline shows the sequence in which they occur.

   There are different types of Messages:

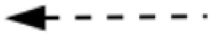
   - **Synchronous Message:** Receiver starts processing the message after receiving it and sender needs to wait until it is made. A straight arrow with close and fill arrowhead from sender life-line bar to receiver end, represent a synchronous message.



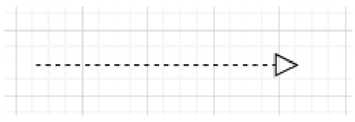
   - **Asynchronous Message:** For asynchronous message sender need not wait for the receiver to process the message. A function call that creates thread can be represented as an asynchronous message in sequence diagram. A straight arrow with open arrowhead from sender life-line bar to receiver end, represent an asynchronous message.



   - **Response Message:** One object can send a message to self. We use this message when we need to show the interaction between the same objects.



   - **Return Message:** For a function call when we need to return a value to the object, from which it was called, then we use return message. But it is optional, and we are using it when we are going to model our system in much detail. A dashed arrow with open arrowhead from sender life-line bar to receiver end, represent that message.



Case Study:

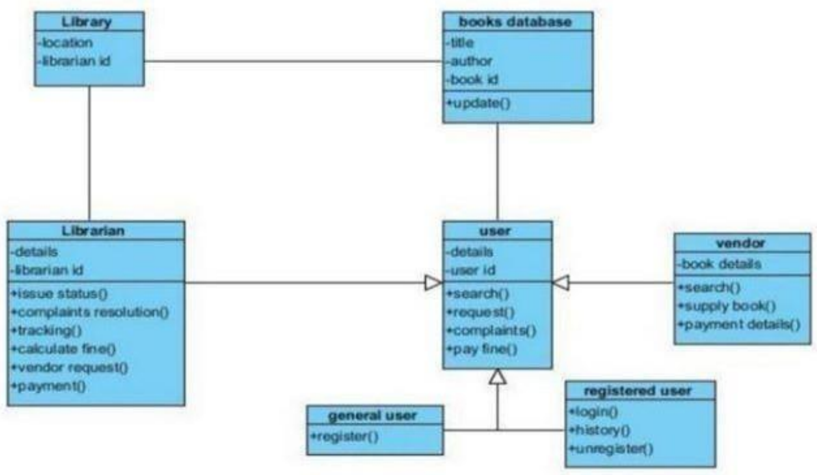
Case Study:

The case study titled Library Management System is library management software for the purpose of monitoring and controlling the transactions in a library. This case study on the library management system gives us the complete information about the library and the daily transactions done in a Library. We need to maintain the record of new sand retrieve the details of books available in the library which mainly focuses on basic operations in a library like adding new member, new books, and up new information, searching books and members and facility to borrow and return books. It features a familiar and well thought-out, an attractive user interface, combined with strong searching, insertion and reporting capabilities. The report generation facility of library system helps to get a good idea of which are then borrowed by the members, makes users possible to generate hard copy.

The following are the brief description on the functions achieved through this case study:

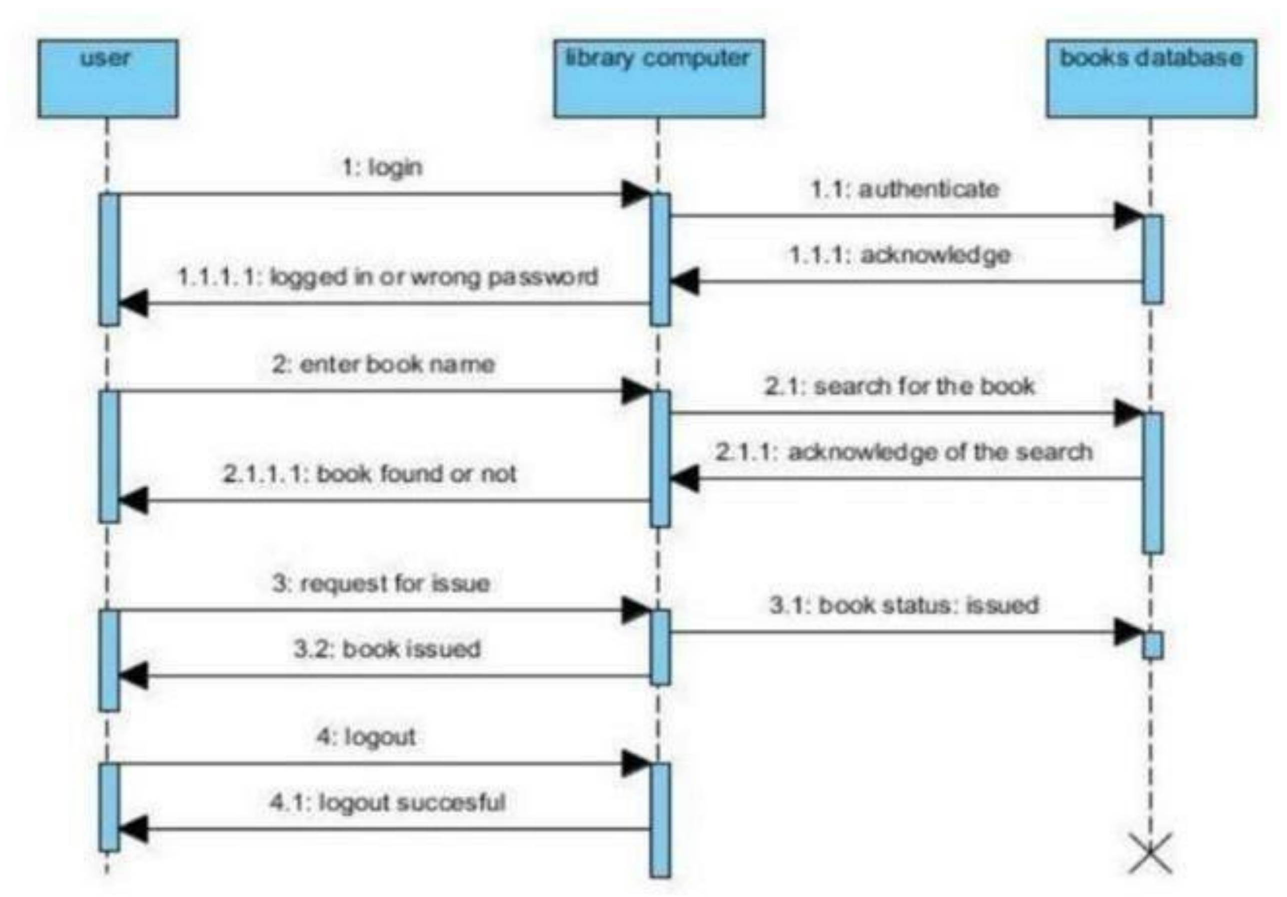
End-Users:
•Librarian: To maintain and update the records and also to cater the needs of the users.
•Reader: Need books to read and also places various requests to the librarian.
•Vendor: To provide and meet the requirement of the prescribed books.

CLASS DIAGRAM:
Classes identified:
- Library
- Librarian
- Books
- Database
- User



SEQUENCE DIAGRAM

Sequence diagram for searching a book and issuing it as per the request by the user from the librarian:



Practical No. 08

OBJECTIVE: Modeling Data Flow Diagrams

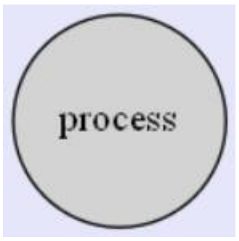
THEORY:

Data Flow Diagrams:

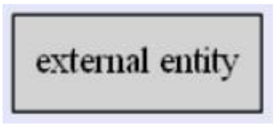
DFD provides the functional overview of a system. The graphical representation easily overcomes any gap between 'user and system analyst' and 'analyst and system designer' in understanding a system. Starting from an overview of the system it explores detailed design of a system through a hierarchy. It also includes the transformations of data flow by the process and the data stores to read or write a data.

Symbols used in DFD:

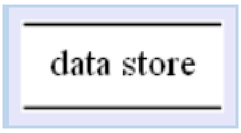
1. **Process:** Processes are represented by circle. The name of the process is written into the circle. The name of the process is usually given in such a way that represents the functionality of the process. More detailed functionalities can be shown in the next Level if it is required.



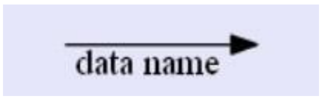
2. **External Entity**: External entities are only appear in context diagram[2]. External entities are represented by a rectangle and the name of the external entity is written into the shape. These send data to be processed and again receive the processed data.



3. **Data store**: Name of the data store is written in between two horizontal lines of the open rectangle. Data stores are used as repositories from which data can be flown in or flown out to or from a process.



4. **Data Flow:** Data flows are shown as a directed edge between two components of a Data Flow Diagram. Data can flow from external entity to process, data store to process, in between two processes and vice-versa.



Context Diagrams and Leveling DFD:

Level 0 diagram is known as context diagram of the system. The entire system is shown as single process and the interactions of external entities with the system are represented in context diagram. Further we split the process in next levels into several numbers of processes to represent the detailed functionalities performed by the system. Data stores may appear in higher level DFDs.

- **Numbering of processes:** If process 'p' in context diagram is split into 3 processes 'p1', 'p2'and 'p3' in next level then these are labeled as 0.1, 0.2 and 0.3 in level 1 respectively. Let the process 'p3' is again split into three processes 'p31', 'p32' and 'p33' in level 2, so, these are labeled as 0.3.1, 0.3.2 and 0.3.3 respectively and so on.

- **Balancing DFD:** The data that flows into the process and the data that flow out to the process need to be matched when the process is split into in the next level. This is known as balancing a DFD.

Case Study:

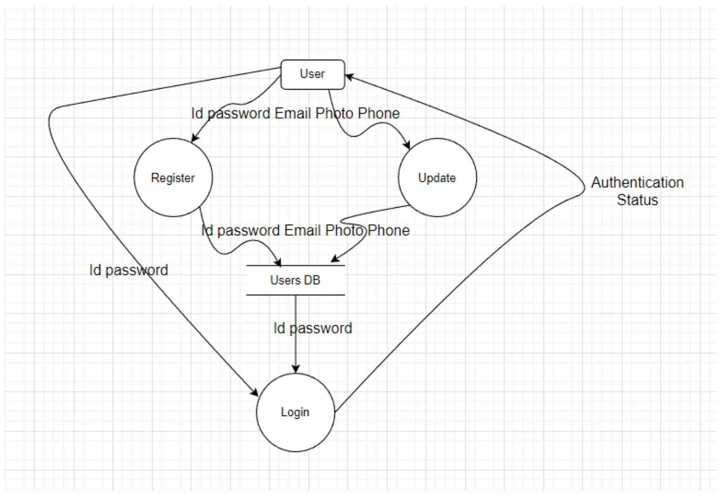
The Absolute Beginners Inc. is planning to launch a revolutionary social networking site, Eye Copy. You have been entrusted with designing a DFD for the proposed application. You have been asked to show the following scenarios:
- **User registration**
- **User login**
- **Profile update**

Draw a Level 1 DFD to depict the above data flow and the corresponding processes.

As per the problem statement, there's only one External Entity Associated which is the USER. Now there are 3 processes involve here, which are as follows:
1. Register User 2. Update User 3. Login

There's also a need of a datastore which will store the updated records i.e. username and password of the registered user.

Thus, the final DFD for the given problem statement is as follows:



Practical No. 09

OBJECTIVE: Estimation of Test Coverage Metrics and Structural Complexity.

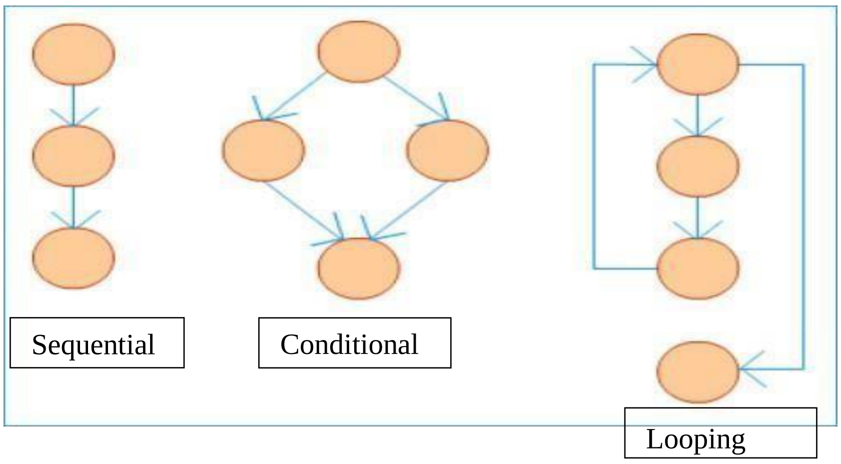
THEORY:

Control Flow Graph:

A control flow graph (CFG) is a directed graph where the nodes represent different instructions of a program, and the edges define the sequence of execution of such instructions. Instructions are Sequential, Conditional, and looping and hence, we have CFG for every type of instructions.

Terminologies used in CFG:

1. **Path:** A path in a CFG is a sequence of nodes and edges that starts from the initial node (or



entry block) and ends at the terminal node. The CFG of a program could have more than one terminal node.
2. **Linearly Independent Path:** A linearly independent path is any path in the CFG of a program such that it includes at least one new edge does not present in any other linearly independent path. A set of linearly independent paths give a clear picture of all possible paths that a program can take during its execution.

McCabe's Cyclomatic Complexity:

Cyclomatic complexity metric, as proposed by McCabe, provides an upper bound for the number of linearly independent paths that could exist through a given program module. Complexity of a module increases as the number of such paths in the module increases. Thus, if Cyclomatic complexity of any program module is 7, there could be up to seven linearly independent paths in the module.

- **Computing Cyclomatic Complexity:**
  Let G be a given CFG. Let E denote the number of edges, and N denote the number of nodes. Let V(G) denote the Cyclomatic complexity for the CFG. V(G) can be obtained in either of the following three ways:

Method 1: **V(G) = E - N + 2**

Method 2: V(G) could be directly computed by a visual inspection of the CFG:

V(G) = Total number of bounded areas + 1

Method 3: If LN be the total number of loops and decision statements in a program, Then:

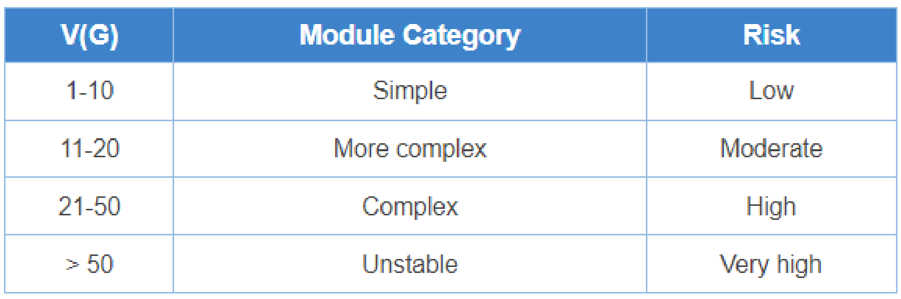
V(G) = LN + 1

Once the complexities of individual modules of a program are known, complexity of the program (or class) could be determined by:

$$V(G) = SUM(V(G_i)) - COUNT(V(G_i)) + 1$$

where $COUNT(V(G_i))$ gives the total number of procedures (methods) in the program (class).

- **Optimum Value of Cyclomatic Complexity:**



V(G)	Module Category	Risk
1-10	Simple	Low
11-20	More complex	Moderate
21-50	Complex	High
> 50	Unstable	Very high

Case Study:

Identification of basic blocks from a program and determining it's cyclomatic complexity
Consider the following simple C program.
Sum of first n natural numbers (not in the best possible way though)

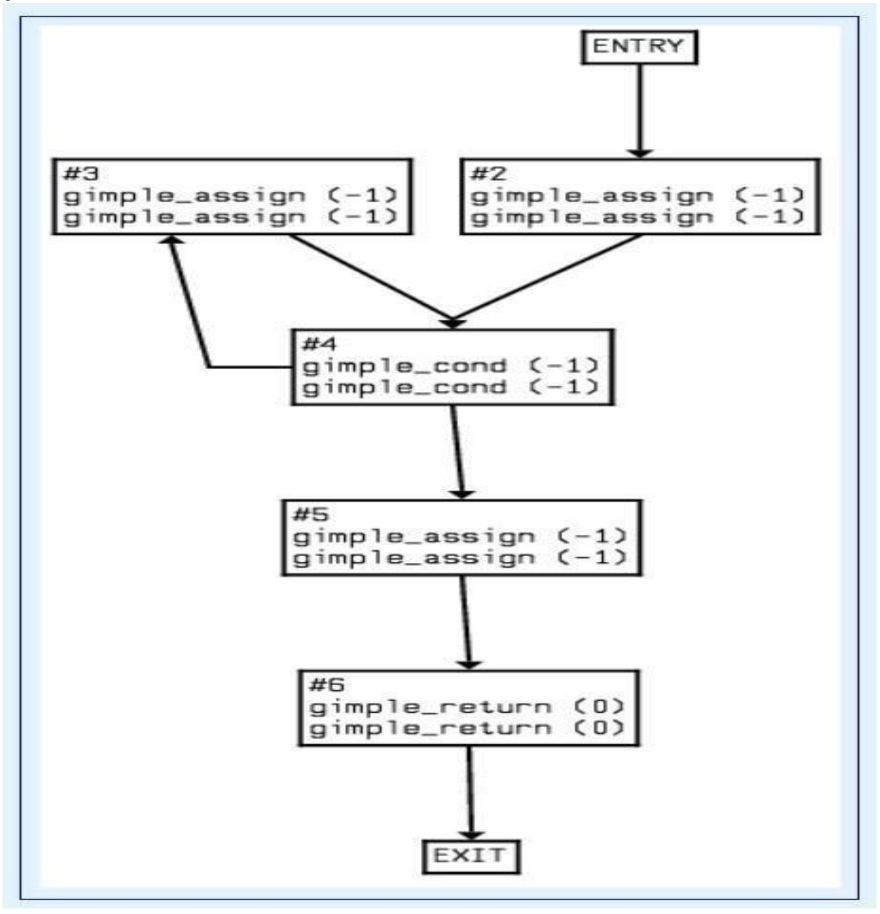
```
#include <stdio.h>
int main(int argc, char **argv)
{
    int i;
    int sum;
    int n = 10;
    sum = 0;
    for (i = 1; i <= n; i++)
        sum += i;
    printf("Sum of first %d natural numbers is: %d\n", n, sum);return
    0;
}
```

Any sequence of instructions in a program could be represented in terms of basic blocks, and a CFG could be drawn using those basic blocks. For the given C program:

1. Identify the basic blocks and verify whether your representation matches withthe output produced after compiling your program.
2. Draw a Control Flow Graph (CFG) using these basic blocks. Again, verify how the CFG generated after compilation relates to the basic blocks identified by the compiler.
3. Calculate McCabe's complexity from the CFG so obtained**.**

```
Function main (main)

Merging blocks 5 and 6


main (int argc, char * * argv)
{
  int  n;
  int sum;
  int i;
  int D.1710;
  const char * restrict D.1709;

<bb 2>:
  n = 10;
  sum = 0;
  i = 1;
  goto <bb 4>
```

```
 <bb 3>:
 sum = sum + i;
 i = i + 1;

<bb 4>:
 if (i ≤ n)
   goto <bb 3>;
 else
   goto <bb 5>;

<bb 5>:
 D.1709 = (const char * restrict) "Sum of first %d natural numbers is: %d\n";
 printf (D.1709, n, sum);
 D.1710 = 0;
 return D.1710;

}
```



Now to calculate the cyclomatic complexity:
Here, Total number of looping statements are (LN): 1
Hence, V(G) = LN + 1

V(G)= 2

Practical No. 10

OBJECTIVE: Designing Test Suites

THEORY:

Software Testing:

Testing software is an important part of the development life cycle of a software. It is an expensive activity. Hence, appropriate testing methods are necessary for ensuring the reliability of a program. According to the ANSI/IEEE 1059 standard, the definition of testing is the process of analyzing a software item, to detect the differences between existing and required conditions i.e., defects/errors/bugs and to evaluate the features of the software item.
The purpose of testing is to verify and validate software and to find the defects present in a software.

a. **Verification:** is the checking or we can say the testing of software for consistency and conformance by evaluating the results against pre-specified requirements.
b. **Validation:** looks at the system's correctness, i.e., the process of checking that what has been specified is what the user wanted**.**
c. **Defect:** is a variance between the expected and actual result. The defect's ultimate source may be traced to a fault introduced in the specification, design, or development (coding) phases.

Testing Frameworks:

These are the different testing frameworks: jUnit, Selenium, HP QC and IBM Rational.

Test Cases and Test Suite:

A test case describes an input description and an expected output description. Input is of two types: preconditions (circumstances that hold prior to test case execution) and the actual inputs that are identified by some testing methods. The set of test cases is called a test suite.
Software testing are mainly of following types:

a. **Unit Testing:** Unit testing is done at the lowest level. It tests the basic unit of software, that is the smallest testable piece of software. The individual component or unit of a program are tested in unit testing. It's of 2 types:
  a) Black Box testing: This is also known as **functional testing**, where the test cases are designed based on input output values only.
  b) White Box testing: It is also known as **structural testing**. In this testing, test cases are designed based on examination of the code. This testing is performed based on the knowledge of how the system is implemented. It includes analyzing data flow, control flow, information flow, coding practices, exception, and error handling within the system, to test the intended and unintended software behavior.

b. **Integration Testing:** Integration testing is performed when two or more tested units are combined into a larger structure. The main objective of this testing is to check whether the different modules of a program interface with each other properly or not.
This testing is also of 2 types:
  a) Top-Down Approach   b) Bottom-up Approach

c. System Testing:

System testing tends to affirm the end-to-end quality of the entire system. System testing is often based on the functional / requirement specification of the system. Non-functional quality attributes, such as reliability, security, and maintainability are also checked**.**
It is of 3 types:

a) Alpha testing: is done by the developers who develop the software. This testing is also done by the client or an outsider with the presence of developer or we can say tester.

b) Beta Testing: is done by very few numbers of end users before the delivery, where the change requests are fixed, if the user gives any feedback or reports any type of defect.

c) User Acceptance testing: s also another level of the system testing process where the system is tested for acceptability. This test evaluates the system's compliance with the client requirements and assesses whether it is acceptable for software delivery.

Regression Testing: The purpose of regression testing is to ensure that bug fixes and new functionality introduced in a software do not adversely affect the unmodified parts of the program. Regression testing is an important activity at both testing and maintenance phases.

Case Study:

The Absolute Beginners Inc. seems to have been fascinated by your work. Recently they have entrusted you with the task of writing web-based mathematical software (using JavaScript). As part of this software, your teammate has written a small module, which computes areas of simple geometric shapes. A portion of the module is shown below.

```
function square(side) { return side * side }

function rectangle(side1, side2) { return side1 * side1; }

function circle(radius) { return Math.PI * radius * radius; }

function right_triangle(base, height) { return 1 / 2 * base * height; }
```

Prepare a test suite that will verify each of the above-mentioned individual function is working correctly.
Your task essentially is to verify whether each of the above function is returning correct values for given inputs. For example, a rectangle with length 10 unit and breadth 5 unit will have an area of 50 sq. unit. This can be verified from the output of the function call:

```
Rectangle(10,5);
```

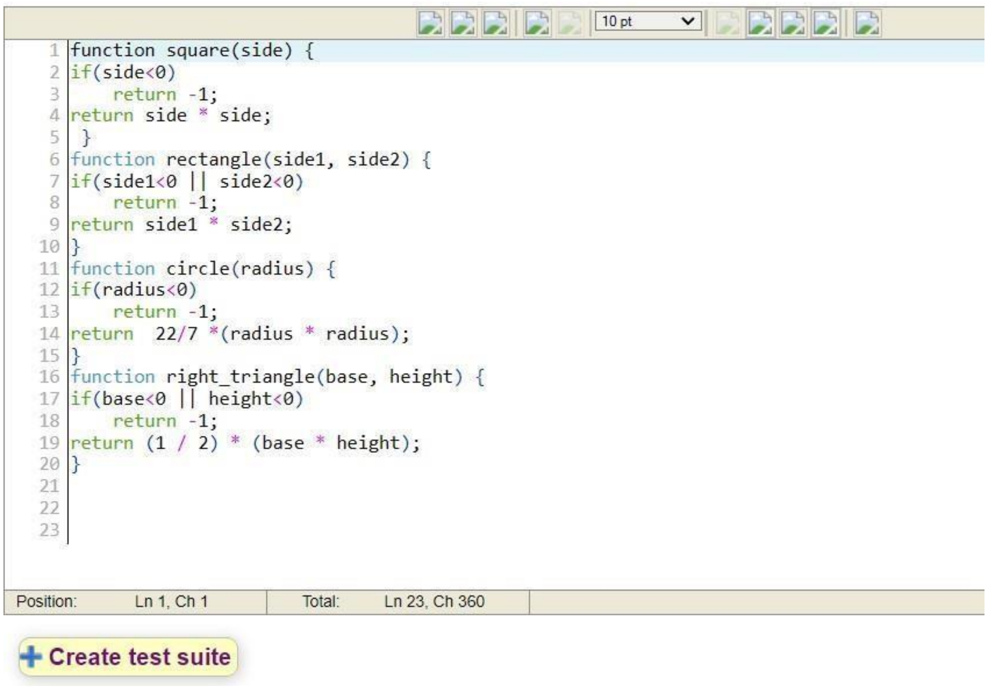
However, testing also attempts to point out possible bugs in the software. How would the above code behave for a call:

```
rectangle(10,-5);
```

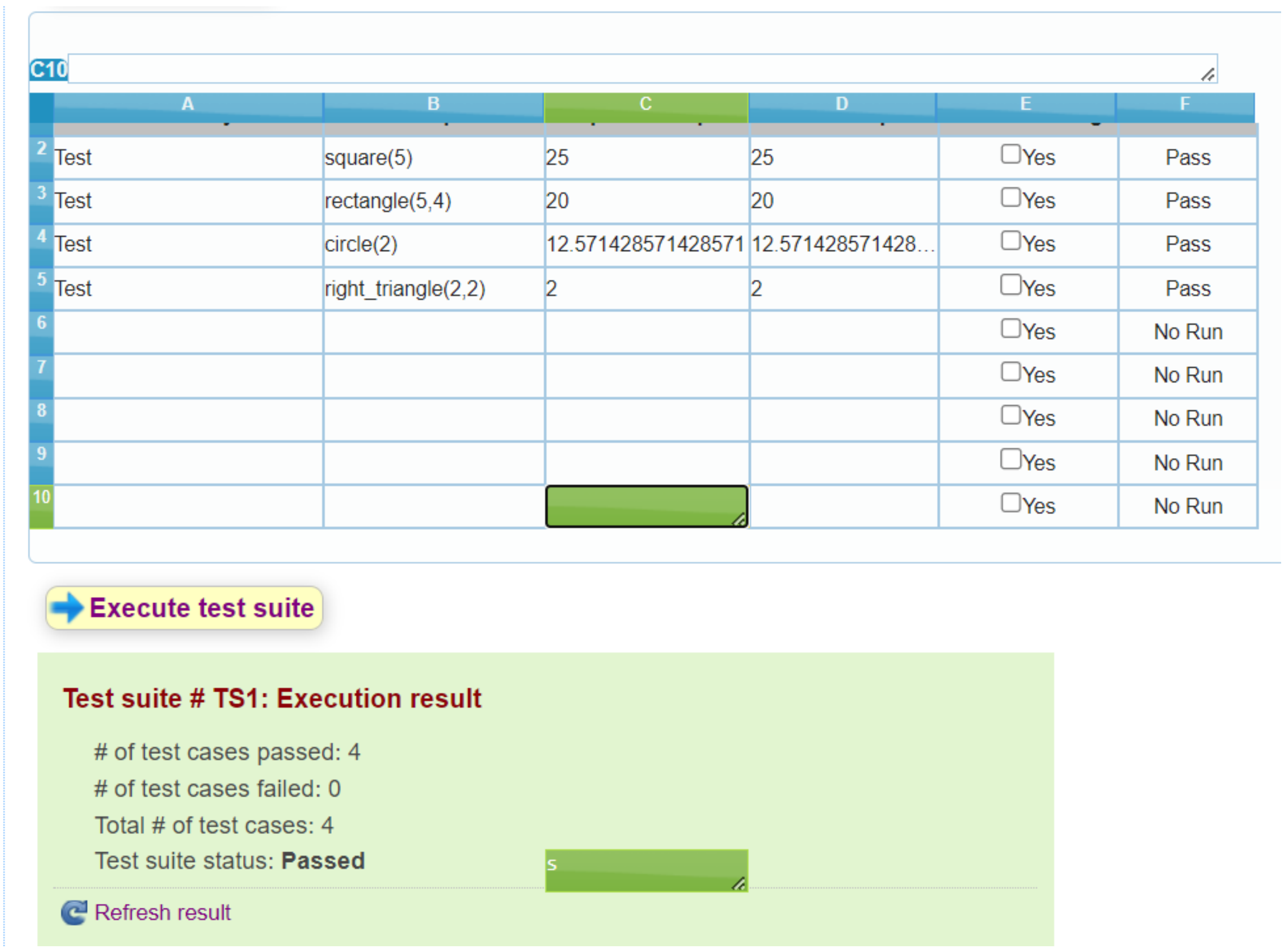
Modify the code to address this defect.

- In each function, return -1 if any given dimension is negative.
- Modify the test suite such that it reflects desired performance for both correct and incorrect input(s)
- The code has another bug -- how would you identify it **from testing results**? Fix the bug and test it again.

Code



```
function square(side) {
if(side<0)
    return -1;
return side * side;
 }
function rectangle(side1, side2) {
if(side1<0 || side2<0)
    return -1;
return side1 * side2;
}
function circle(radius) {
if(radius<0)
    return -1;
return  22/7 *(radius * radius);
}
function right_triangle(base, height) {
if(base<0 || height<0)
    return -1;
return (1 / 2) * (base * height);
}
```



C10

	A	B	C	D	E	F
2	Test	square(5)	25	25	☐Yes	Pass
3	Test	rectangle(5,4)	20	20	☐Yes	Pass
4	Test	circle(2)	12.571428571428571	12.571428571428…	☐Yes	Pass
5	Test	right_triangle(2,2)	2	2	☐Yes	Pass
6					☐Yes	No Run
7					☐Yes	No Run
8					☐Yes	No Run
9					☐Yes	No Run
10					☐Yes	No Run

Test suite # TS1: Execution result

# of test cases passed: 4
# of test cases failed: 0
Total # of test cases: 4
Test suite status: **Passed**