Subject: Software Engineering & Design Lab Manual

Subject Code: 21CS44

Semester: IV

Syllabus

Unit No.	No.of experiments	Topic(s) related to Experiment
1	1	Software Processes & process flow diagram using online open source design tool.
2	2	Requirements Engineering: Requirement collection, listing of important functions and analysis. Tools Used for Story Card Preparation and estimation of task.
3	3	Software Design & Development listing the actors with relevance and listing of use-cases summarizing the purpose. Design sequence diagram any one of the function identified with all suitable constructs. Draw an Activity diagram for any software design tools.
4	3	Software Testing-Unit Testing with example & Prepare software Test Document compare test results. Testing based on system testing, Integration tests & automation using the tool.
5	1	Project work: use case of any Common Software Application listing all the functional & non-functional requirements, Show the suitable

Termwork 1:

Software Processes & process flow diagram using online open source design tool.

Introduction

A software process is set of related activities that leads to the production of a software product

Many different software processes but all involve:

- Specification – The functionality of the software and constraints on its operation must be defined.
- Design and implementation – defining the organization of the system and implementing the system.
- Validation – checking that it does what the customer wants;
- Evolution – changing the system in response to changing customer needs.

A software process model is a simplified representation of a process. It presents a description of a process from some particular perspective.

Process Flow Diagram

A process flow diagram in Software Engineering is a graphical representation that illustrates the sequence of activities, tasks, and interactions involved in the software development process. It provides a visual overview of how the software is planned, designed, developed, tested, and deployed, helping stakeholders understand the workflow and the relationships between different phases.

Objectives

To investigate, analyze, and understand the different software processes utilized in software development and to create a comprehensive process flow diagram using an online open-source design tool.

The experiment aims to achieve the following specific objectives:

- Explore Software Processes
- Compare Software Process Models
- Understand Process Flow
- Identify Best Practices
- Hands-On Process Flow Diagram Creation
- Enhance Visualization Skills
- Real-World Application
- Identify Areas of Improvement
- Collaboration and Communication
- Reflection and Learning

Tools

Dia Diagram Editor: The Dia Diagram Editor is a popular open-source diagramming tool used to create a wide variety of diagrams, including process flow diagrams, flowcharts, network diagrams, UML diagrams, and more. It is available for multiple platforms, including Windows, macOS, and Linux, making it accessible to a broad user base.

Website: http://dia-installer.de/

- Draw.io Website: https://www.draw.io/
- Creately Website: https://creately.com/
- PlantUML Website: http://plantuml.com/
- Cacoo Website: https://cacoo.com/
- Graphviz Website:https://www.graphviz.org/
- Pencil Project Website: https://pencil.evolus.vn/

Outcomes

- Enhanced Understanding of Software Processes
- Comparison of Software Process Models
- Proficient in Process Flow Representation
- Insight into Best Practices
- Real-World Application
- Identified Areas of Improvement:
- Improved Visualization and Collaboration Skills
- Informed Decision Making:
- Reflective Learning Experience
- Useful Artifact

Instructions to Students for the record writing

- Problem Statement: Specify the problem statement and the same should be executed using any of the above-mentioned tool.
- Write down the following description related to your project
- Introduction
- Objectives:
- Tools used:
- Output: Attach the print out of the software process flow diagram drawn for the above problem statement specified.

Termwork 2:

Requirements Engineering: Requirement collection, listing of important functions and analysis.

Introduction:

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. Usually business analysts having domain knowledge on the subject matter discuss with clients and decide what features are to be implemented.

In this experiment we will learn how to identify functional and non-functional requirements from a given problem statement. Functional and non-functional requirements are the primary components of a Software Requirements Specification.

Objectives

After completing this experiment, you will be able to:

- Identify ambiguities, inconsistencies and incompleteness from a requirements specification
- Identify and state functional requirements
- Identify and state non-functional requirements

Requirements

Sommerville defines "requirement" as a specification of what should be implemented. Requirements specify how the target system should behave. It specifies what to do, but not how to do. Requirements engineering refers to the process of understanding what a customer expects from the system to be developed, and to document them in a standard and easily readable and understandable format. This documentation will serve as reference for the subsequent design, implementation and verification of the system.

It is necessary and important that before we start planning, design and implementation of the software system for our client, we are clear about it's requirements. If we don't have a clear vision of what is to be developed and what all features are expected, there would be serious problems, and customer dissatisfaction as well.

Characteristics of Requirements

Requirements gathered for any new system to be developed should exhibit the following three properties:

- ***Unambiguity:*** There should not be any ambiguity what a system to be developed should do. For example, consider you are developing a web application for your client. The client requires that enough number of people should be able to access the application simultaneously. What's the "enough number of people"? That could mean 10 to you, but, perhaps, 100 to the client. There's an ambiguity.

- ***Consistency:*** To illustrate this, consider the automation of a nuclear plant. Suppose one of the clients say that it the radiation level inside the plant exceeds R1, all reactors should be shut down. However, another person from the client side suggests that the threshold radiation level should be R2. Thus, there is an inconsistency between the two end users regarding what they consider as threshold level of radiation.
- ***Completeness:*** A particular requirement for a system should specify what the system should do and also what it should not. For example, consider a software to be developed for ATM. If a customer enters an amount greater than the maximum permissible withdrawal amount, the ATM should display an error message, and it should not dispense any cash.

Categorization of Requirements

Based on the target audience or subject matter, requirements can be classified into different types, as stated below:

- User requirements: They are written in natural language so that both customers can verify their requirements have been correctly identified
- System requirements: They are written involving technical terms and/or specifications, and are meant for the development or testing teams

Requirements can be classified into two groups based on what they describe:

- **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.
- **Non-functional requirements (NFRs):** They are not directly related what functionalities are expected from the system. However, NFRs could typically define how the system should behave under certain situations. For example, a NFR could say that the system should work with 128MB RAM. Under such condition, a NFR could be more critical than a FR.

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

Functional Requirements

Identifying 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. For example, a Library Management System, apart from anything else, should be able to issue and return books.
- Identify the cases where an end user gets some meaningful work done by using the system. For example, in a digital library a user might use the "Search Book" functionality to obtain information about the books of his interest.
- 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. For example, to search for a book, user gives title of the book as input and get the book details and location as the output.
- Any high-level requirement identified could have different sub-requirements. For example, "Issue Book" module could behave differently for different class of users, or for a particular user who has issued the book thrice consecutively.

Preparing Software Requirements Specifications

Once all possible FRs and non-FRs have been identified, which are complete, consistent, and non-ambiguous, the Software Requirements Specification (SRS) is to be prepared. IEEE provides a template [iv], also available here, which could be used for this purpose. The SRS is prepared by the service provider, and verified by its client. This document serves as a legal agreement between the client and the service provider. Once the concerned system has been developed and deployed, and a proposed feature was not found to be present in the system, the client can point this out from the SRS. Also, if after delivery, the client says a new feature is required, which was not mentioned in the SRS, the service provider can again point to the SRS. The scope of the current experiment, however, doesn't cover writing a SRS.

Example:

Functional And Non Functional Requirements For Online Banking System Application

Functional Requirements for an Online Banking System Application:

- ***User Registration:*** The system shall allow customers to register for an online banking account by providing their personal information, such as name, address, contact details, and identification documents.
- ***Account Management:*** The system shall enable customers to view their account balances, transaction history, and manage their accounts, including transferring funds between accounts and setting up automatic bill payments.
- ***Fund Transfers:*** The system shall facilitate electronic fund transfers between the customer's accounts and to external accounts, including other bank accounts and third-party payment platforms.

- ***Bill Payment:*** The system shall provide a feature for customers to pay their bills online, allowing them to add and manage payees, schedule recurring payments, and view payment history.
- ***Customer Support:*** The system shall offer customer support features, such as a secure messaging system or a helpline, to address customer inquiries, report issues, and request assistance.

Non-functional Requirements for an Online Banking System Application:

- ***Security:*** The application shall implement robust security measures, including encryption, secure authentication mechanisms, and transaction verification protocols, to protect customer data and prevent unauthorized access.
- ***Performance:*** The application shall have a fast response time, with minimal latency, ensuring smooth user experience even during peak usage periods.
- ***Availability:*** The application shall be available 24/7, with minimal downtime for scheduled maintenance or system updates.
- ***Usability:*** The application shall have an intuitive user interface, with clear navigation, user-friendly controls, and accessibility features to accommodate users with disabilities.
- ***Compliance:*** The application shall adhere to relevant banking regulations and industry standards, such as data protection laws and financial transaction security requirements.

Outcomes

- Comprehensive Requirement Collection Process
- Effective Stakeholder Communication
- Requirement Prioritization & Requirements Documentation
- Functional and Non-Functional Requirements
- Requirements Analysis Techniques
- Problem Identification and Resolution
- Requirement Change Management
- Collaboration and Teamwork & Risk Mitigation
- Improved Software Development Process

<u>Instructions to Students for the record writing</u>

- Problem Statement: Specify the problem statement and identify the functional & non-functional requirements for the specified problem statement.
- Write down the following description related to your project
- Introduction & Related Theory
- Objectives & Outcomes

Termwork 3:

Tools Used for Story Card Preparation and estimation of task in Software Engineering

Introduction:

In Software Engineering, Story Card preparation and task estimation are critical activities in Agile development methodologies like Scrum. These activities help teams break down project requirements into manageable tasks and estimate the effort required for each task. Several tools can be used to facilitate Story Card preparation and task estimation. Some popular ones include:

- **Jira:** Jira is a widely used project management and issue tracking tool that supports Agile methodologies like Scrum and Kanban. It allows teams to create Story Cards, manage tasks, and estimate work using story points or other estimation techniques.
- **Trello:** Trello is a flexible and user-friendly project management tool that is based on Kanban boards. Teams can use Trello boards to create and organize Story Cards and perform task estimation using custom labels or plugins.
- **Azure DevOps (formerly Visual Studio Team Services):** Azure DevOps provides a comprehensive set of tools for Agile project management, including support for Story Cards, task tracking, and estimation using effort points or story points.
- **VersionOne:** VersionOne is an Agile project management tool that offers features tailored for Scrum, such as Story Card creation, backlog management, and task estimation.
- **Pivotal Tracker:** Pivotal Tracker is specifically designed for Agile software development and offers a simple interface for creating, prioritizing, and estimating Story Cards.
- **Clubhouse:** Clubhouse is an Agile project management tool that supports Story Card creation, task tracking, and estimation using points or other custom units.
- **Targetprocess:** Targetprocess is a visual project management tool that allows teams to create Story Cards, track tasks, and perform estimation using various techniques.
- **Rally (formerly CA Agile Central):** Rally is an Agile project management tool that provides features for Story Card management and task estimation based on points.
- **Yodiz:** Yodiz is a project management tool tailored for Agile and Scrum, offering capabilities for Story Card creation, backlog management, and task estimation.
- **Monday.com:** Monday.com is a versatile project management tool that can be customized to support Agile workflows, including Story Card creation and task estimation.

It's essential to choose a tool that aligns with your team's specific needs, preferences, and existing workflows. Some tools offer integrations with other development and collaboration tools, making it easier to manage the entire software development process seamlessly.

Outcomes

leads to a more organized, collaborative, and data-driven approach to project management. These outcomes can result in increased productivity, reduced development time, better resource allocation, and overall higher quality software products.

Instructions to Students for the record writing

- Introduction & Related Theory
- Objectives
- Problem Statement: Specify the problem statement and prepare the following
  (a) Story Card Preparation with respect to problem statement chosen.
  (b) estimation of task for the whole project execution.
- Outcomes.

Termwork No.4

Software Design & Development listing the actors with relevance and listing of use-cases summarizing the purpose.

Introduction

Use case diagram is a platform that can provide a common understanding for the end-users, developers and the domain experts. It is used to capture the basic functionality i.e. use cases, and the users of those available functionality, i.e. actors, from a given problem statement.

In this experiment, we will learn how use cases and actors can be captured and how different use cases are related in a system.

Objectives

After completing this experiment, you will be able to:

- How to identify different actors and use cases from a given problem statement
- How to associate use cases with different types of relationships
- How to draw a use-case diagram

Use case diagrams

Use case diagrams belong to the category of behavioural diagram of UML diagrams. Use case diagrams aim to present a graphical overview of the functionality provided by the system. It consists of a set of actions (referred to as use cases) that the concerned system can perform, one or more actors, and dependencies among them.

Actor

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.

For example, consider the case where a customer withdraws cash from an ATM. Here, customer is a human actor.

Actors can be classified as below [2], [i]:

- **Primary actor:** They are principal users of the system, who fulfill their goal by availing some service from the system. For example, a customer uses an ATM to withdraw cash when he needs it. A customer is the primary actor here.
- **Supporting actor:** They render some kind of service to the system. "Bank representatives", who replenishes the stock of cash, is such an example. It may be noted that replenishing stock of cash in an ATM is not the prime functionality of an ATM.

In a use case diagram primary actor are usually drawn on the top left side of the diagram.

Use Case

- A use case is simply [1] a functionality provided by a system.
- Continuing with the example of the ATM, withdraw cash is a functionality that the ATM provides. Therefore, this is a use case. Other possible use cases includes, check balance, change PIN, and so on.
- Use cases include both successful and unsuccessful scenarios of user interactions with the system. For example, authentication of a customer by the ATM would fail if he enters wrong PIN. In such case, an error message is displayed on the screen of the ATM.

Subject

- Subject is simply [iii] the system under consideration. Use cases apply to a subject. For example, an ATM is a subject, having multiple use cases, and multiple actors interact with it. However, one should be careful of external systems interacting with the subject as actors.

Graphical Representation

An actor is represented by a stick figure and name of the actor is written below it. A use case is depicted by an ellipse and name of the use case is written inside it. The subject is shown by drawing a rectangle. Label for the system could be put inside it. Use cases are drawn inside the rectangle, and actors are drawn outside the rectangle, as shown in figure - 01.

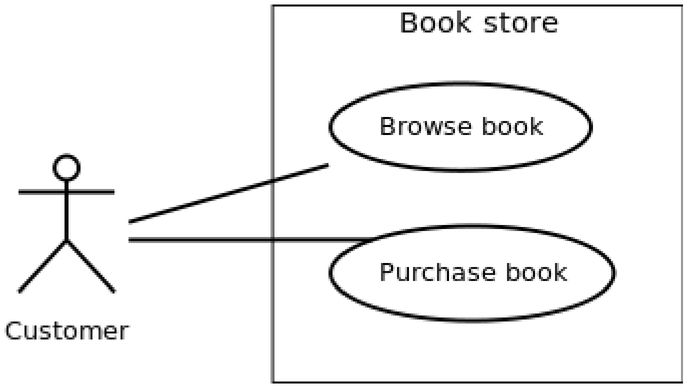


Figure - 01: A use case diagram for a book store

Association between Actors and Use Cases

- A use case is triggered by an actor. Actors and use cases are connected through binary associations indicating that the two communicates through message passing.
- An actor must be associated with at least one-use case. Similarly, a given use case must be associated with at least one actor. Association among the actors are usually not shown. However, one can depict the class hierarchy among actors.

Use Case Relationships

Three types of relationships exist among use cases:

- Include relationship
- Extend relationship
- Use case generalization

Include Relationship

Include relationships are used to depict common behaviour that are shared by multiple use cases. This could be considered analogous to writing functions in a program in order to avoid repetition of writing the same code. Such a function would be called from different points within the program.

Example

For example, consider an email application. A user can send a new mail, reply to an email he has received, or forward an email. However, in each of these three cases, the user must be logged in to perform those actions. Thus, we could have a login use case, which is included by compose mail, reply, and forward email use cases. The relationship is shown in figure - 02.

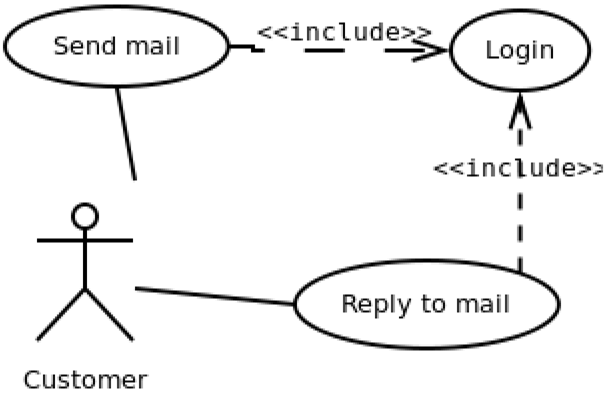


Figure - 02: Include relationship between use cases

Notation

Include relationship is depicted by a dashed arrow with a «include» stereotype from the including use case to the included use case.

Extend Relationship

Use case extensions are used to depict any variation to an existing use case. They are used to the specify the changes required when any assumption made by the existing use case becomes false.

Example

Let's consider an online bookstore. The system allows an authenticated user to buy selected book(s). While the order is being placed, the system also allows to specify any special shipping instructions, for example, call the customer before delivery. This Shipping Instructions step is optional, and not a part of the main Place Order use case. Figure - 03 depicts such relationship.

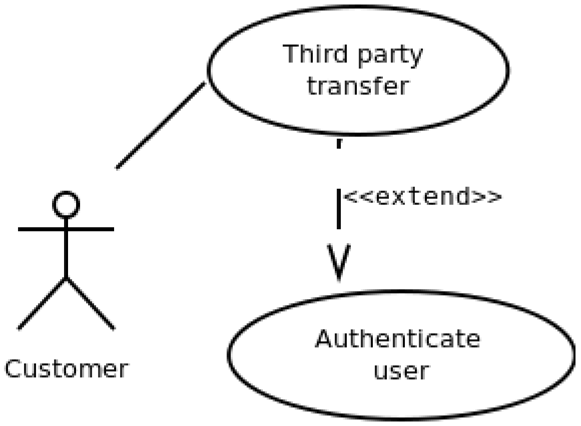


Figure - 03: Extend relationship between use cases

Notation

Extend relationship is depicted by a dashed arrow with a «extend» stereotype from the extending use case to the extended use case.

Generalization Relationship

Generalization relationship are used to represent the inheritance between use cases. A derived use case specializes some functionality it has already inherited from the base use case.

Example

To illustrate this, consider a graphical application that allows users to draw polygons. We could have a use case draw polygon. Now, rectangle is a particular instance of polygon having four sides at right angles to each other. So, the use case draw rectangle inherits the properties of the use case draw polygon and overrides it's drawing method. This is an example of generalization relationship. Similarly, a generalization relationship exists between draw rectangle and draw square use cases. The relationship has been illustrated in figure - 04.

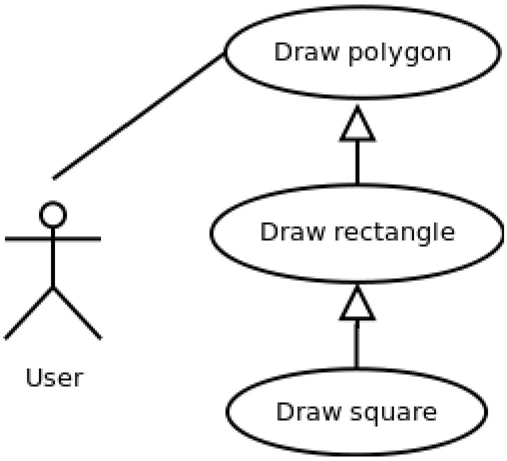


Figure - 04: Generalization relationship among use cases

Notation

Generalization relationship is depicted by a solid arrow from the specialized (derived) use case to the more generalized (base) use case.

Identifying Actors

Given a problem statement, the actors could be identified by asking the following questions :

- Who gets most of the benefits from the system? (The answer would lead to the identification of the primary actor)
- Who keeps the system working? (This will help to identify a list of potential users)
- What other software / hardware does the system interact with?
- Any interface (interaction) between the concerned system and any other system?

Identifying Use cases

Once the primary and secondary actors have been identified, we have to find out their goals i.e. what are the functionality they can obtain from the system. Any use case name should start with a verb like, "Check balance".

Guidelines for drawing Use Case diagrams

Following general guidelines could be kept in mind while trying to draw a use case diagram [1]:

- Determine the system boundary
- Ensure that individual actors have well-defined purpose
- Use cases identified should let some meaningful work done by the actors
- Associate the actors and use cases -- there shouldn't be any actor or use case floating without any connection
- Use include relationship to encapsulate common behaviour among use cases , if any

Instructions to Students for the record writing

- Introduction & Related Theory
- Objectives
- Problem Statement: Specify the problem statement
- Write down the following description relating to your project
  (a) list of actors: List all the actors and give one-line description for the same.
  (b) use cases: List all the use cases and give description for the same.
  (C) Mapping functional requirements to use cases
  (d) Draw the Use case diagram for one/more specific use cases
- Outcomes

Termwork No.5: Design sequence diagram any one of the functions identified with all suitable constructs

Sequence diagram

It represents the behavioural 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

Sequence diagram contains the objects of a system and their life-line bar and the messages passing between them.

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. Also, we may use only class name or only instance name.

Objects which are created at the time of execution of use case and are involved in message passing, are appear in diagram, at the point of their creation.

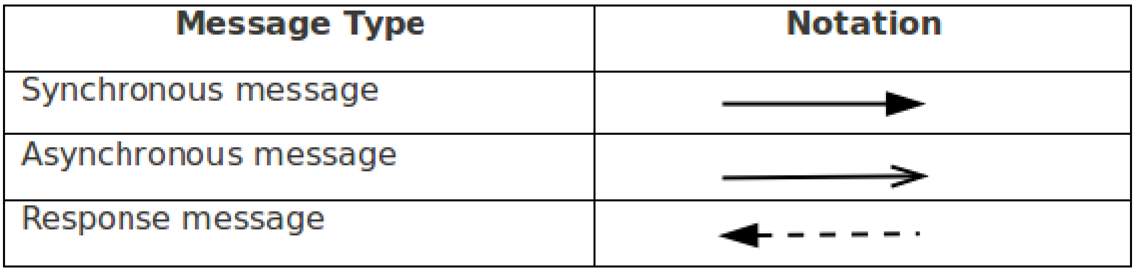
Life-line bar

A down-ward vertical line from object-box is shown as the life-line of the object. A rectangle bar on life-line indicates that it is active at that point of time.

Messages

Messages are shown as an arrow from the life-line of sender object to the life-line of receiver object and labeled with the message name. Chronological order of the messages passing throughout the objects' life-line show the sequence in which they occur. There may exist some different types of messages:

- **Synchronous messages**: Receiver start processing the message after receiving it and sender needs to wait until it is made. A straight arrow with close and fill arrow-head from sender life-line bar to receiver end, represent a synchronous message.
- **Asynchronous messages**: For asynchronous message sender needs not to 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 arrow-head from sender life-line bar to receiver end, represent an asynchronous message.
- **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 arrow-head from sender life-line bar to receiver end, represent that 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 object.



Message Type	Notation
Synchronous message	——▶
Asynchronous message	——>
Response message	◀- - - - -

Figure-08

<u>Instructions to Students for the record writing</u>

- Introduction & Related Theory
- Objectives
- Problem Statement: Specify the problem statement
- Write down the following description relating to your project

  (a) Identify the logical sequence of activities undergoing in a system (Identify the objects and the actors)

  (b) Draw the Sequence Diagram for the said problem statement
- Outcomes

Termwork No.6 : Draw an Activity diagram for any software design tools

Activity Diagrams

Activity diagrams fall under the category of behavioural 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.

Activity diagrams, however, cannot depict the message passing among related objects. As such, it can't be directly translated into code. These kinds of diagrams are suitable for confirming the logic to be implemented with the business users. These diagrams are typically used when the business logic is complex. In simple scenarios it can be avoided entirely.

Components of an Activity Diagram

Below we describe the building blocks of an activity diagram.

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, as shown in table-01. A label inside the rectangle identifies the corresponding activity.
- There are two special type of activity nodes: initial and final. They are represented with a filled circle, and a filled in circle with a border respectively (table-01). Initial node represents the starting point of a flow in an activity diagram. There could be multiple initial nodes, which means that invoking that particular activity diagram would initiate multiple flows.
- A final node represents the end point of all activities. Like an initial node, there could be multiple final nodes. Any transition reaching a final node would stop all activities.

Flow

A flow (also termed as edge, or transition) 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 with a label, called the guard condition, indicating the necessary condition for the transition to happen. The syntax to depict it is [guard condition].

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. The input edge could also have guard conditions. Alternately, a note can be attached to the decision node indicating the condition to be tested.

Merge

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

Fork

Fork is a point where parallel activities begin. For example, when a student has been registered with a college, he can in parallel apply for student ID card and library card. A fork is graphically depicted with a black bar, with a single flow entering and multiple flows leaving out.

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 of the incoming controls must be completed before any further progress could be made. For example, a sales order is closed only when the customer has receive the product, and the sales company has received it's payment.

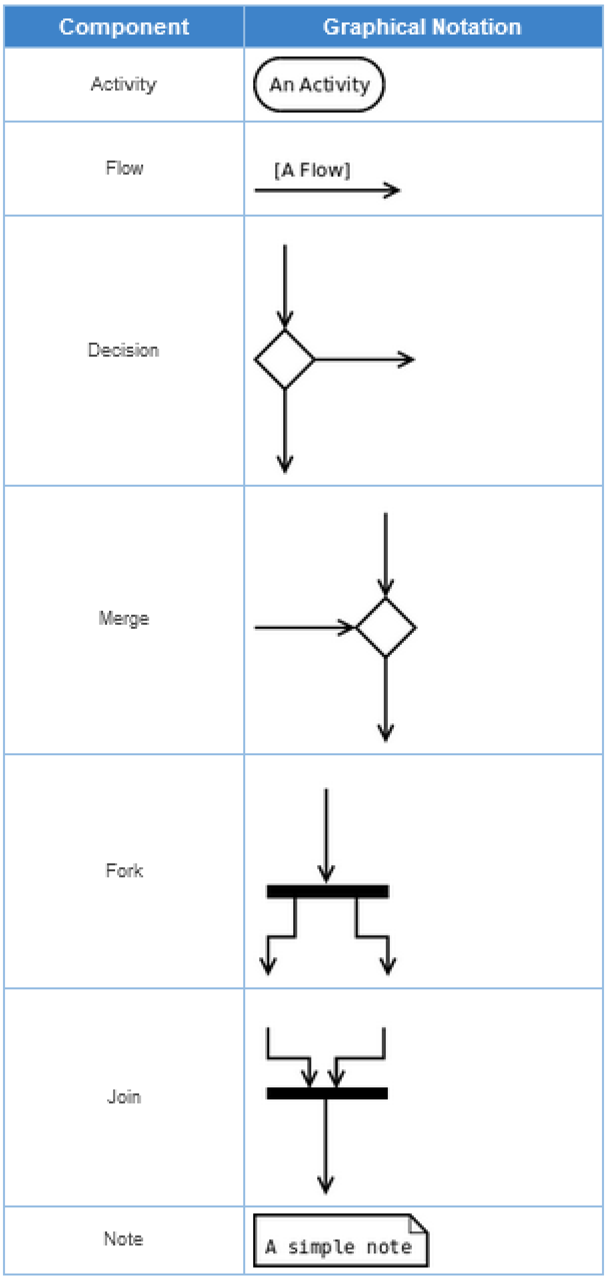
Note

UML allows attaching a note to different components of a diagram to present some textual information. The information could simply be a comment or may be some constraint. A note can be attached to a decision point, for example, to indicate the branching criteria.

Partition

Different components of an activity diagram can be logically grouped into different areas, called partitions or swimlanes. They often correspond to different units of an organization or different actors. The drawing area can be partitioned into multiple compartments using vertical (or horizontal) parallel lines. Partitions in an activity diagram are not mandatory.

The following table shows commonly used components with a typical activity diagram.



Component	Graphical Notation
Activity	An Activity
Flow	[A Flow]
Decision	
Merge	
Fork	
Join	
Note	A simple note

Table-01: Typical components used in an activity diagram

Guidelines for drawing an Activity Diagram

The following general guidelines could be followed to pictorially represent a complex logic.

- Identify tiny pieces of work being performed by the system
- Identify the next logical activity that should be performed
- Think about all those conditions that should be made, and all those constraints that should be satisfied, before one can move to the next activity
- Put non-trivial guard conditions on the edges to avoid confusion

Problem Statement: Specify the problem statement

Write down the following description wrt your project

(A)Identify activities representing basic units of work, and represent their flow using activity diagram

Instructions to Students for the record writing

- Introduction & Related Theory
- Objectives
- Problem Statement: Specify the problem statement
- Write down the following description wrt your project
  (a) Identify activities representing basic units of work, and represent their flow using activity diagram
- Outcomes.

Teamwork's based on Software Testing:

Software Testing-Unit Testing with example & Prepare software Test Document compare test results. Testing based on system testing, Integration tests & automation using the tool.

Termwork No. 7:

Design and develop a program in a language of your choice to solve the triangle problem defined as follows: Accept three integers which are supposed to be the three sides of a triangle and determine if the three values represent an equilateral triangle, isosceles triangle, scalene triangle, or they do not form a triangle at all. Derive test cases for your program based on decision-table approach, execute the test cases and discuss the results.

REQUIREMENTS:

R1. The system should accept 3 positive integer numbers (a, b, c) which represents 3 sides of the triangle. Based on the input it should determine if a triangle can be formed or not.

R2. If the requirement R1 is satisfied then the system should determine the type of the triangle, which can be

- Equilateral (i.e. all the three sides are equal)
- Isosceles (i.e Two sides are equal)
- Scalene (i.e All the three sides are unequal)

else suitable error message should be displayed. Here we assume that user gives three positive integer numbers as input.

DESIGN:

Form the given requirements we can draw the following conditions:

C1: $a<b+c$?

C2: $b<a+c$?

C3: $c<a+b$?

C4: $a=b$?

C5: $a=c$?

C6: $b=c$?

According to the property of the triangle, if any one of the three conditions C1,

C2 and C3 are not satisfied then triangle cannot be constructed. So only when C1, C2 and C3 are true the triangle can be formed, then depending on conditions C4, C5 and C6 we can decide what type of triangle will be formed. (i.e requirement R2).

ALGORITHM:

```
Step 1: Input a, b & c i.e three integer values which represent three sides of the triangle.
Step 2: if (a < (b + c)) and (b < (a + c)) and (c < (a + b) then
        do step 3
        Else
        print not a triangle. do step 6.
Step 3: if (a=b) and (b=c) then
Print triangle formed is equilateral. do step 6. Step 4: if (a ≠ b) and (a ≠ c) and (b ≠ c) then
Print triangle formed is scalene. do step 6.
Step 5: Print triangle formed is Isosceles.
Step 6: stop
```

PROGRAM CODE:

```
#include<stdio.h>
#include<ctype.h>
#include<conio.h>
#include<process.h>
int main()
{
int a, b, c;
clrscr();
printf("Enter three sides of the triangle");
scanf("%d%d%d", &a, &b, &c); if((a<b+c)&&(b<a+c)&&(c<a+b))
{
if((a==b)&&(b==c))
{
printf("Equilateral triangle");
}
else if((a!=b)&&(a!=c)&&(b!=c))
{
```

```
printf("Scalene triangle");
}
else
        printf("Isosceles triangle");


}
else
{
        printf("triangle cannot be formed");
}
getch();
 return 0;
}
```

TESTING:

Technique Used: Decision Table Approach

Decision Table-Based Testing has been around since the early 1960's; it is used to depict complex logical relationships between input data. A Decision Table is the method used to build a complete set of test cases without using the internal structure of the program in question. In order to create test cases we use a table to contain the input and output values of a program.

The decision table is as given below:

Conditions	**Condition Entries (Rules)**										
	R 1	R 2	R 3	R 4	R 5	R 6	R 7	R 8	R 9	R 10	R 11
C1: a<b+c?	F	T	T	T	T	T	T	T	T	T	T
C2: b<a+c?	--	F	T	T	T	T	T	T	T	T	T
C3: c<a+b?	--	--	F	T	T	T	T	T	T	T	T
C4: a=b?	--	--	--	F	T	T	T	F	F	F	T
C5: a=c?	--	--	--	T	F	T	F	T	F	F	T
C6: b=c?	--	--	--	T	T	F	F	F	T	F	T
Actions	**Action Entries**										
a1: Not a Triangle	X	X	X								
a2: Scalene										X	
a3: Isosceles							X	X	X		

a4: Equilateral											X
a5: Impossible				X	X	X					

The “--“symbol in the table indicates don’t care values. The table shows the six conditions and 5 actions. All the conditions in the decision table are binary; hence, it is called as “Limited Entry decision table”.

Each column of the decision table represents a test case. That is, the table is read as follows:

1. When condition C1 is false we can say that with the given ‘a’ ‘b’ and ‘c’ values, it’s Not a triangle.

   Action: Not a Triangle

2. Similarly condition C2 and C3, if any one of them are false, we can say that with the given ‘a’ ‘b’ and ‘c’ values it’s Not a triangle.

   Action: Impossible

3. When conditions C1, C2, C3 are true and two conditions among C4, C5, C6 is true, there is no chance of one conditions among C4, C5, C6 failing. So we can neglect these rules.

   Example: if condition C4: a=b is true and C5: a=c is true

   Then it is impossible, that condition C6: b=c will fail, so the action is Impossible.

   Action: Isosceles

4. When conditions C1, C2, C3 are true and any one condition among C4, C5 and C6 is true with remaining two conditions false then action is Isosceles triangle.

   Example: If condition C4: a=b is true and C5: a=c and C6: b=c are false, it means two sides are equal. So the action will be Isosceles triangle.

   Action: Equilateral

5. When conditions C1, C2, C3 are true and also conditions C4, C5 and C6 are true then, the action is Equilateral triangle.

   Action: Scalene

6. When conditions C1, C2, C3 are true and conditions C4, C5 and C6 are false i.e sides a, b and c are different, then action is Scalene triangle.

   Number of Test Cases = Number of Rules.

   Using the decision table we obtain 11 functional test cases: 3 impossible cases.

3 ways of failing the triangle property, 1 way to get an equilateral triangle, 1 way to get a scalene triangle, and 3 ways to get an isosceles triangle.

Deriving test cases using

Decision Table Approach:

Test Cases:

TC ID	Test Case Description	a	B	c	Expected Output	Actual Output	Status
1	Testing for Requirement 1	4	1	2	Not a Triangle		
2	Testing for Requirement 1	1	4	2	Not a Triangle		
3	Testing for Requirement 1	1	2	4	Not a Triangle		
4	Testing for Requirement 2	5	5	5	Equilateral		
5	Testing for Requirement 2	2	2	3	Isosceles		
6	Testing for Requirement 2	2	3	2	Isosceles		
7	Testing for Requirement 2	3	2	2	Isosceles		
8	Testing for Requirement 2	3	4	5	Scalene		

EXECUTION & RESULT DISCUSION

Execute the program against the designed test cases and complete the table for Actual output column and status column.

Test Report:

1. No of TC’s Executed: 08

2. No of Defects Raised:

3. No of TC’s Pass:

4. No of TC’s Failed:

The decision table technique is indicated for applications characterised by any of the following:

Prominent if-then-else logic

Logical relationships among input variables

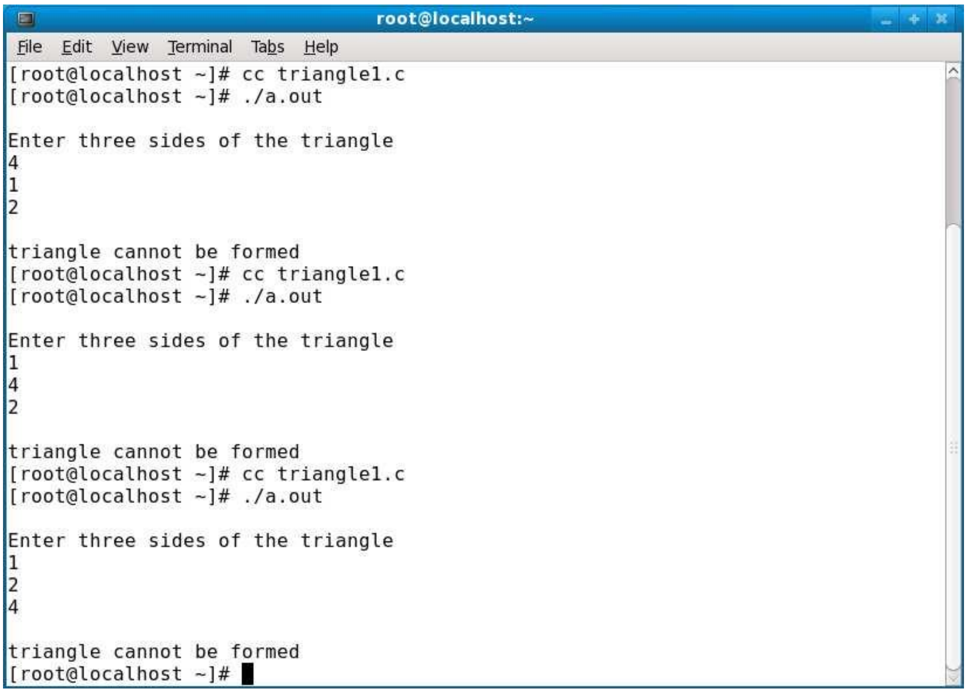
Calculations involving subsets of the input variables

Cause-and-effect relationship between inputs and outputs

The decision table-based testing works well for triangle problem because a lot of decision making i.e if-then-else logic takes place.

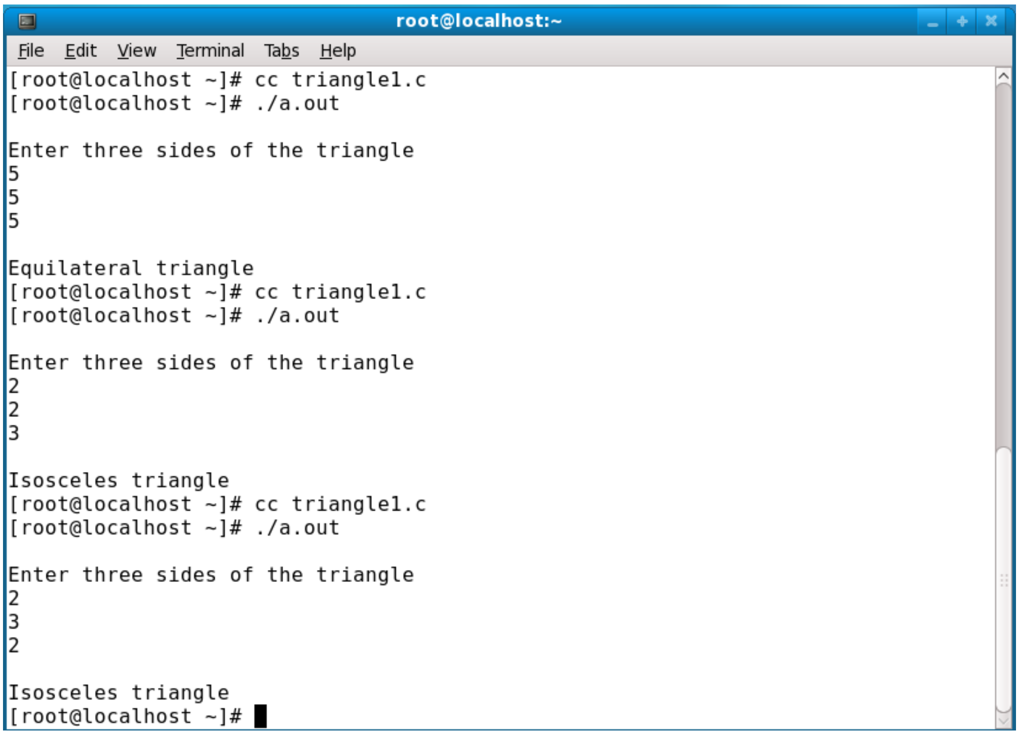
Snapshots

1. Output screen of Triangle cannot be formed



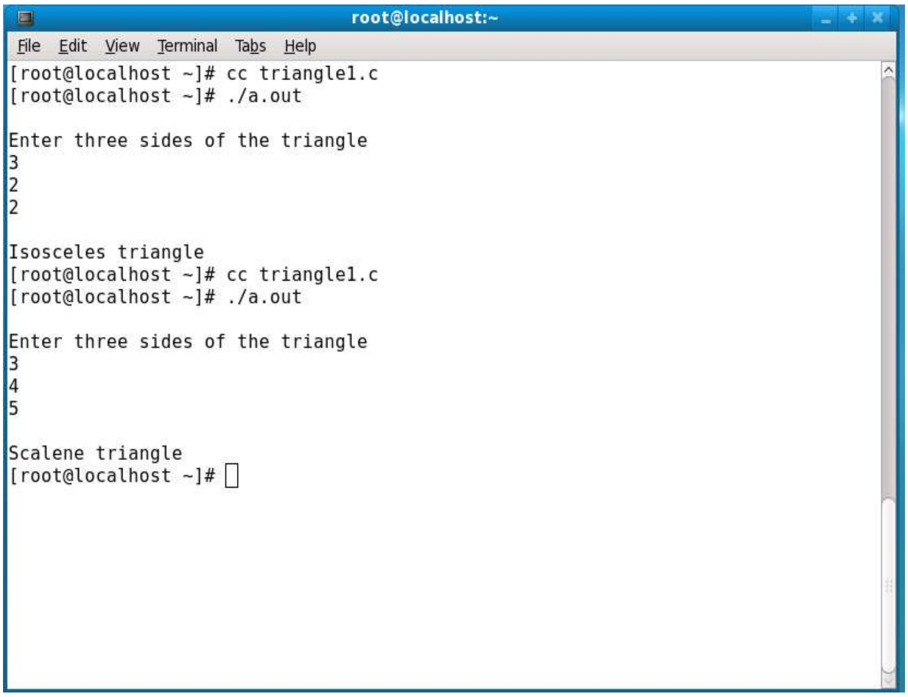
```
[root@localhost ~]# cc triangle1.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
4
1
2

triangle cannot be formed
[root@localhost ~]# cc triangle1.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
1
4
2

triangle cannot be formed
[root@localhost ~]# cc triangle1.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
1
2
4

triangle cannot be formed
[root@localhost ~]# 
```

2. Output screen of Equilateral and Isosceles Triangle.



```
[root@localhost ~]# cc triangle1.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
5
5
5

Equilateral triangle
[root@localhost ~]# cc triangle1.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
2
2
3

Isosceles triangle
[root@localhost ~]# cc triangle1.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
2
3
2

Isosceles triangle
[root@localhost ~]#
```

3. Output screen for Scalene Triangle



```
[root@localhost ~]# cc triangle1.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
3
2
2

Isosceles triangle
[root@localhost ~]# cc triangle1.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
3
4
5

Scalene triangle
[root@localhost ~]#
```

Termwork No: 8

Design and develop a program in a language of your choice to solve the triangle problem defined as follows: Accept three integers which are supposed to be the three sides of a triangle and determine if the three values represent an equilateral triangle, isosceles triangle, scalene triangle, or they do not form a triangle at all. Assume that the upper limit for the size of any side is 10. Derive test cases for your program based on **boundary-value analysis**, execute the test cases and discuss the results.

REQUIREMENTS

R1. The system should accept 3 positive integer numbers (a, b, c) which represents 3 sides of the triangle.

R2. Based on the input should determine if a triangle can be formed or

not.

R3. If the requirement R2 is satisfied then the system should determine the type of the triangle, which can be

- Equilateral (i.e. all the three sides are equal)
- Isosceles (i.e Two sides are equal)
- Scalene (i.e All the three sides are unequal)

R4. Upper Limit for the size of any side is 10

DESIGN

ALGORITHM:

```
Step 1: Input a, b & c i.e three integer values which represent three sides of the triangle.
Step 2: if $(a < (b + c))$ and $(b < (a + c))$ and $(c < (a + b)$ then
do step 3
else
print not a triangle. do step 6.
Step 3: if $(a=b)$ and $(b=c)$ then
Print triangle formed is equilateral. do step 6.
Step 4: if $(a \neq b)$ and $(a \neq c)$ and $(b \neq c)$ then
Print triangle formed is scalene. do step 6
Step 5: Print triangle formed is Isosceles.
Step 6: stop
```

PROGRAM CODE:

```
#include<stdio.h>
#include<ctype.h>
#include<conio.h>
#include<process.h>
int main()
{
int a, b, c;
clrscr();
printf("Enter three sides of the triangle");
scanf("%d%d%d", &a, &b, &c);
if((a > 10) || (b > 10) || (c > 10))
{
printf("Out of range");
getch();
exit(0);
}

if((a<b+c)&&(b<a+c)&&(c<a+b))
{
	if((a==b)&&(b==c))
	{
		printf("Equilateral triangle");
	}

	else if((a!=b)&&(a!=c)&&(b!=c))
	{
		printf("Scalene triangle");
	}
	else
```

```
                printf("Isosceles triangle");
}
else
{
        printf("triangle cannot be formed");
}
getch();
return 0;
}
```

TESTING

1. Technique used: Boundary value analysis
2. Test Case design

For BVA problem the test cases can be generation depends on the output and the constraints on the output. Here we least worried on the constraints on Input domain.

The Triangle problem takes 3 sides as input and checks it for validity, hence n = 3. Since BVA yields (4n + 1) test cases according to single fault assumption theory, hence we can say that the total number of test cases will be (4*3+1) =12+1=13.

The maximum limit of each sides a, b, and c of the triangle is 10 units according to requirement R4. So a, b and c lies between

$0 \leq a \leq 10$

$0 \leq b \leq 10$

$0 \leq c \leq 10$

Equivalence classes for a:

E1: Values less than 1. E2: Values in the range.

E3: Values greater than 10.

Equivalence classes for b:

E4: Values less than 1

E5: Values in the range.

E6: Values greater than 10.

Equivalence classes for c:

E7: Values less than 1. E8: Values in the range.

E9: Values greater than 10.

From the above equivalence classes we can derive the following test cases using boundary value analysis approach.

TC Id	**Test Case Description**	**Input Data**			**Expected Output**	**Actual Output**	**Status**
		A	**b**	**C**			
1	For A input is not given	X	3	6	Not a Triangle		
2	For B input is not given	5	X	4	Not a Triangle		
3	For C input is not given	4	7	X	Not a Triangle		
4	Input of C is in negative(-)	5	5	-1	Not a Triangle		
5	Two sides are same one side is given different input	5	5	1	Isosceles		
6	All Sides of inputs are equal	5	5	5	Equilateral		
7	Two sides are same one side is given different input	5	5	9	Isosceles		
8	The input of C is out of range (i.e., range is <10)	5	5	10	Not a Triangle		
9	Two sides are same one side is given different input (i.e., A & C are 5, B=1)	5	1	5	Isosceles		
10	Two sides are same one side is given different input (i.e., A & C are 5, B=2)	5	2	5	Isosceles		

11	Two sides are same one side is given different input (i.e., A & C are 5, B=9)	5	9	5	Isosceles		
12	Two sides are same one side is given different input (i.e., A & C are 5, B=10 so, it is out of given range)	5	10	5	Not a Triangle		
13	Two sides are same one side is given different input (i.e., B & C are 5, A=1)	1	5	5	Isosceles		
14	Two sides are same one side is given different input (i.e., B & C are 5, A=2)	2	5	5	Isosceles		
15	Two sides are same one side is given different input (i.e., B & C are 5, A=9)	9	5	5	Isosceles		
16	Two sides are same one side is given different input (i.e., B & C are 5, A=10, so the given input of A is out of range)	10	5	5	Not a Triangle		

Table-1: Test case for Triangle Problem

EXECUTION:

Execute the program and test the test cases in Table-1 against program and complete the table with for Actual output column   and Status column

Test Report:

1. No of TC's Executed:

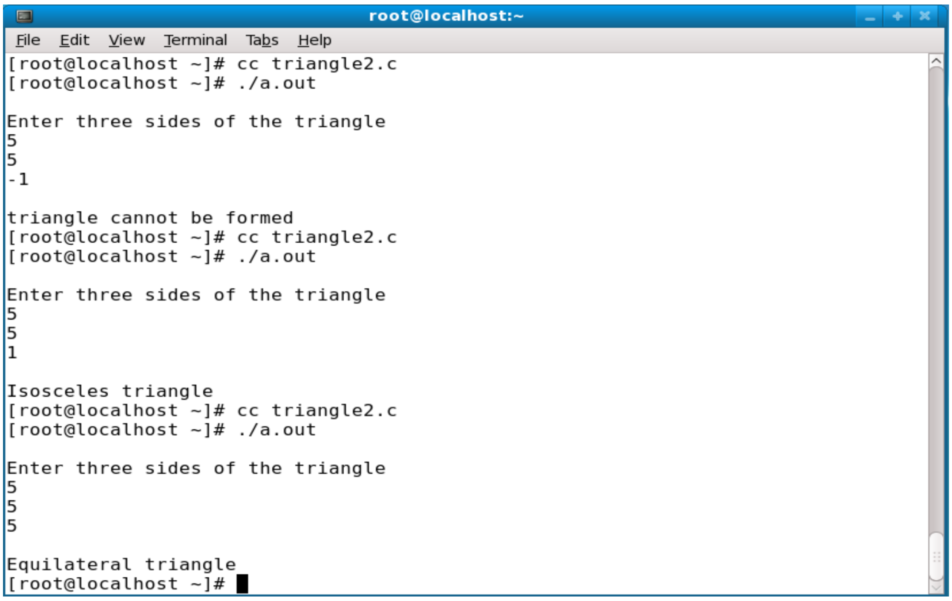
2. No of Defects Raised:

3. No of TC's Pass:

4. No of TC's Failed:

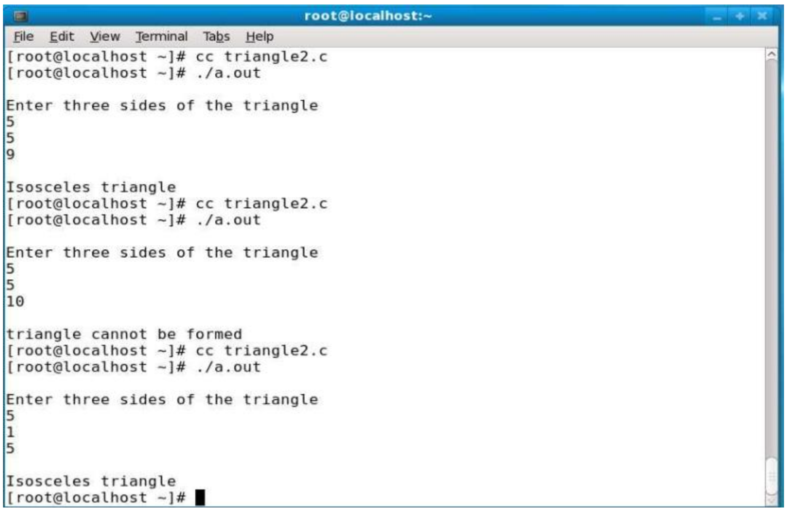
SNAPSHOTS:

1.Snapshot of Isosceles and Equilateral triangle and triangle cannot be formed.



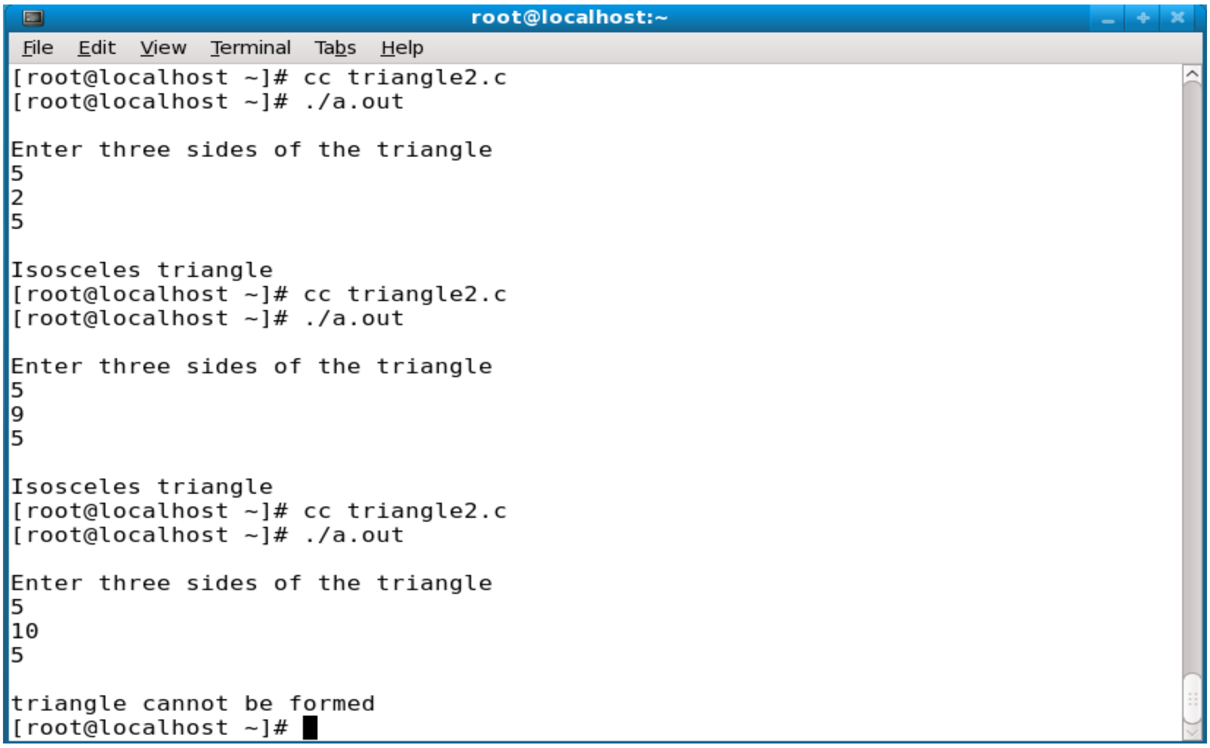
```
[root@localhost ~]# cc triangle2.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
5
5
-1

triangle cannot be formed
[root@localhost ~]# cc triangle2.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
5
5
1

Isosceles triangle
[root@localhost ~]# cc triangle2.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
5
5
5

Equilateral triangle
[root@localhost ~]# 
```

2. Snapshot for Isosceles and triangle cannot be formed

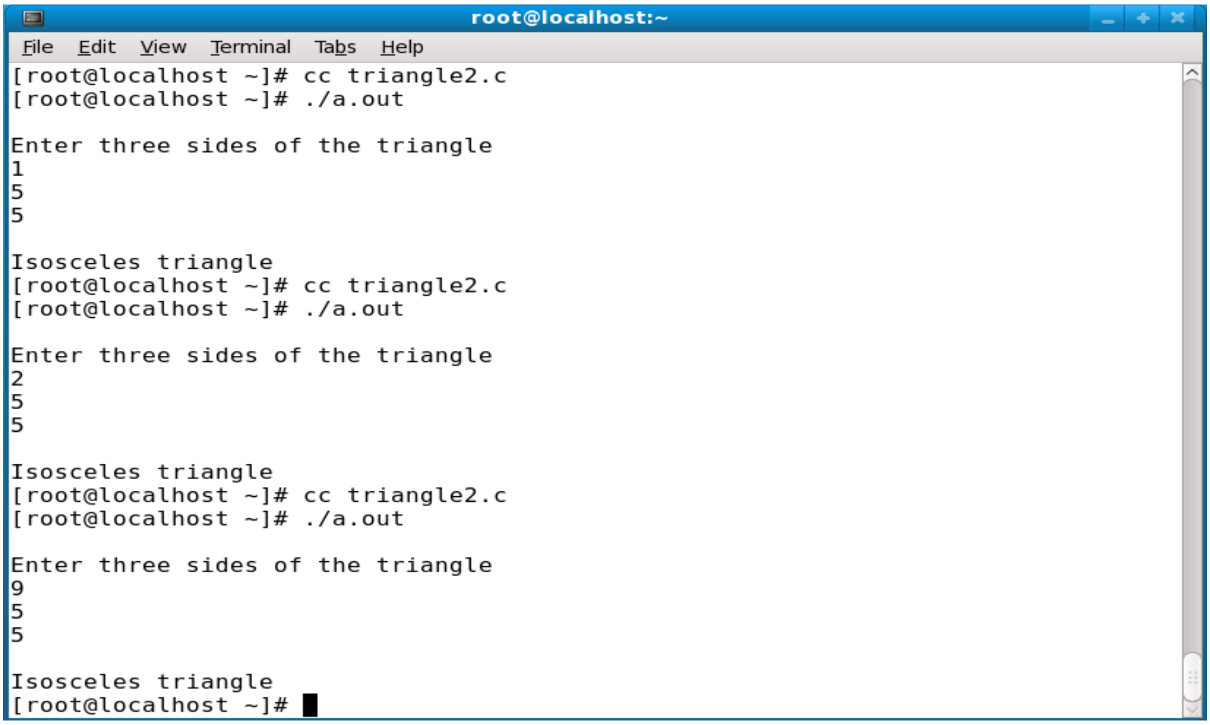


```
[root@localhost ~]# cc triangle2.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
5
5
9

Isosceles triangle
[root@localhost ~]# cc triangle2.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
5
5
10

triangle cannot be formed
[root@localhost ~]# cc triangle2.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
5
1
5

Isosceles triangle
[root@localhost ~]# 
```

3. Snapshot for Isosceles and triangle cannot be formed

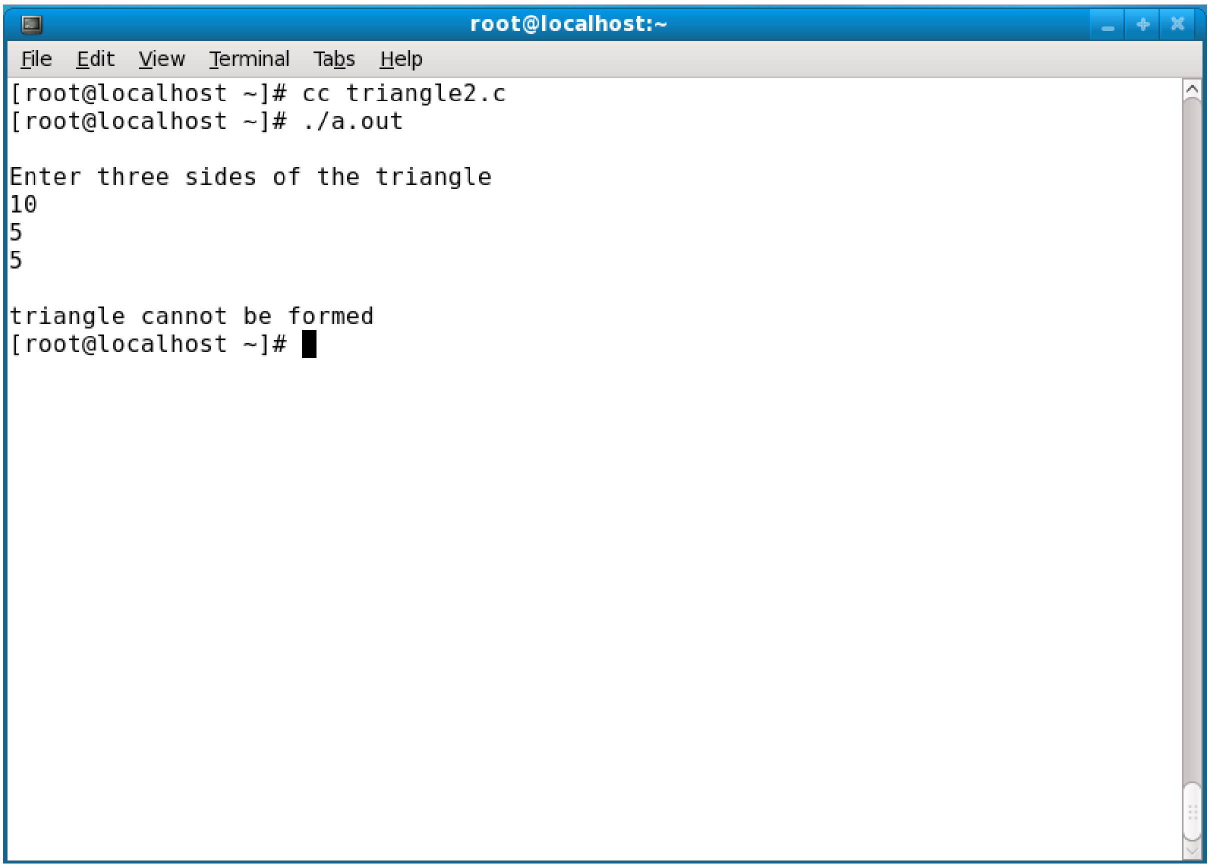


```
[root@localhost ~]# cc triangle2.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
5
2
5

Isosceles triangle
[root@localhost ~]# cc triangle2.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
5
9
5

Isosceles triangle
[root@localhost ~]# cc triangle2.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
5
10
5

triangle cannot be formed
[root@localhost ~]# 
```



```
[root@localhost ~]# cc triangle2.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
1
5
5

Isosceles triangle
[root@localhost ~]# cc triangle2.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
2
5
5

Isosceles triangle
[root@localhost ~]# cc triangle2.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
9
5
5

Isosceles triangle
[root@localhost ~]# 
```

4. Output screen for Triangle cannot be formed



```
[root@localhost ~]# cc triangle2.c
[root@localhost ~]# ./a.out

Enter three sides of the triangle
10
5
5

triangle cannot be formed
[root@localhost ~]# 
```

Termwork No:9

Design, develop, code and run the program in any suitable language to implement the binary search algorithm. Determine the **basis paths** and using them derive different test cases, execute these test cases and discuss the test results.

Introduction

- A white box method
- Proposed by McCabe in 1980's
- A hybrid of path testing and branch testing methods.
- Based on Cyclomatic complexity and uses control flow to establish the path coverage criteria.

Cyclomatic Complexity

- Developed by McCabe in 1976
- Measures the number of linearly independent paths through a program.
- The higher the number the more complex the code

Basic path testing approach

Step 1: Draw a control flow graph.

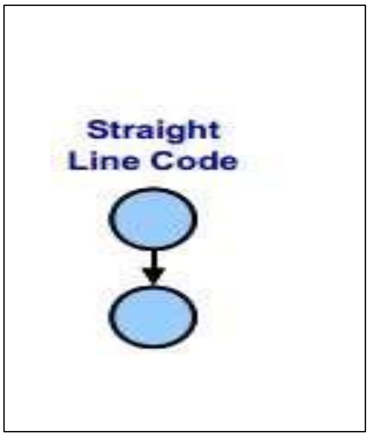
Step 2: Determine Cyclomatic complexity.

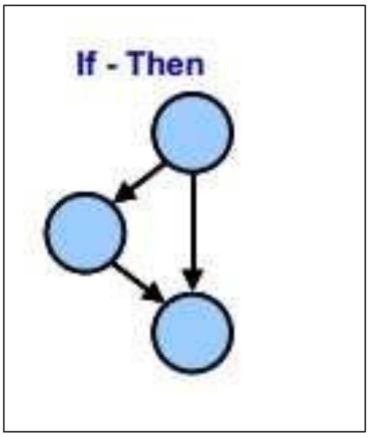
Step 3: Find a basis set of paths.

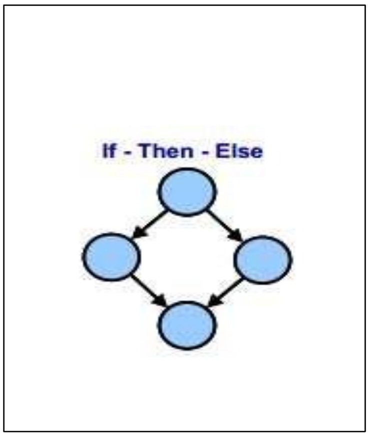
Step 4: Generate test cases for each path

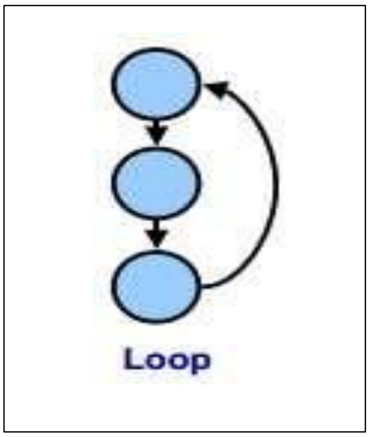
Draw a control flow graph

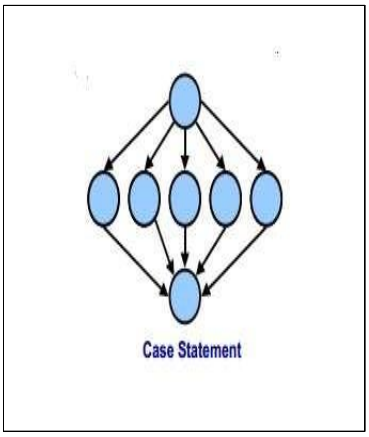
Basic control flow graph structures





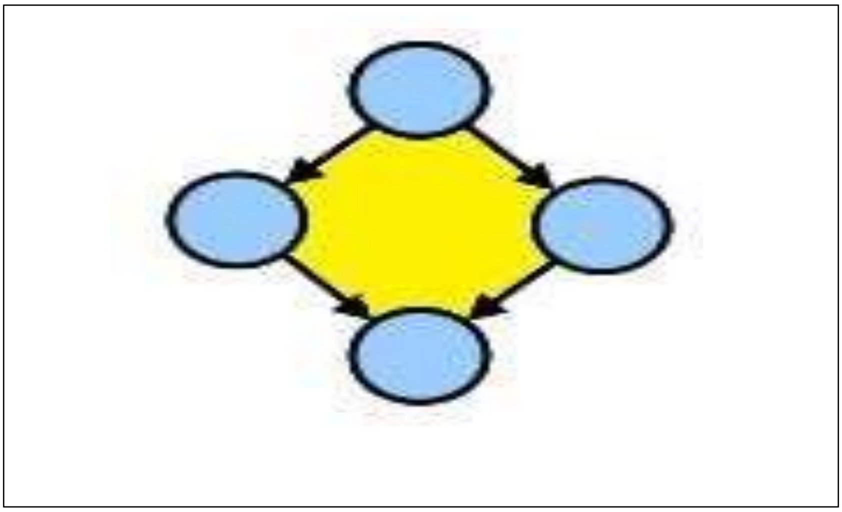




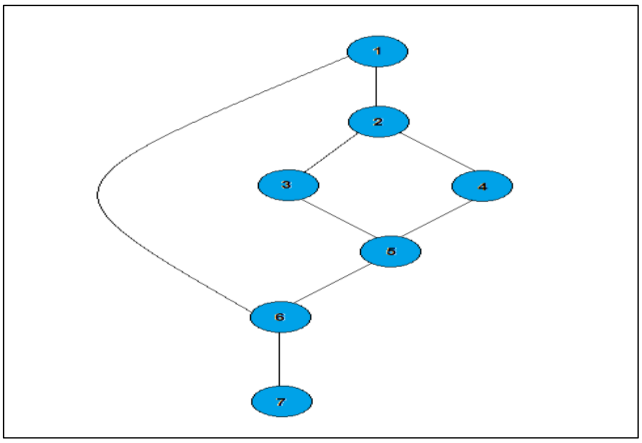


Draw a control flow graph

- Arrowsor edges represent flows of control.
- Circles or nodes represent actions.
- Areas bounded by edges and nodes are called regions.
- A predicate node is a node containing a condition.



Draw a control flow graph



1: IF A = 100

2: THEN IF B > C

3: THEN A = B

4: ELSE A= C

5: ENDIF

6: ENDIF

7: Print A

Determine Cyclomatic complexity

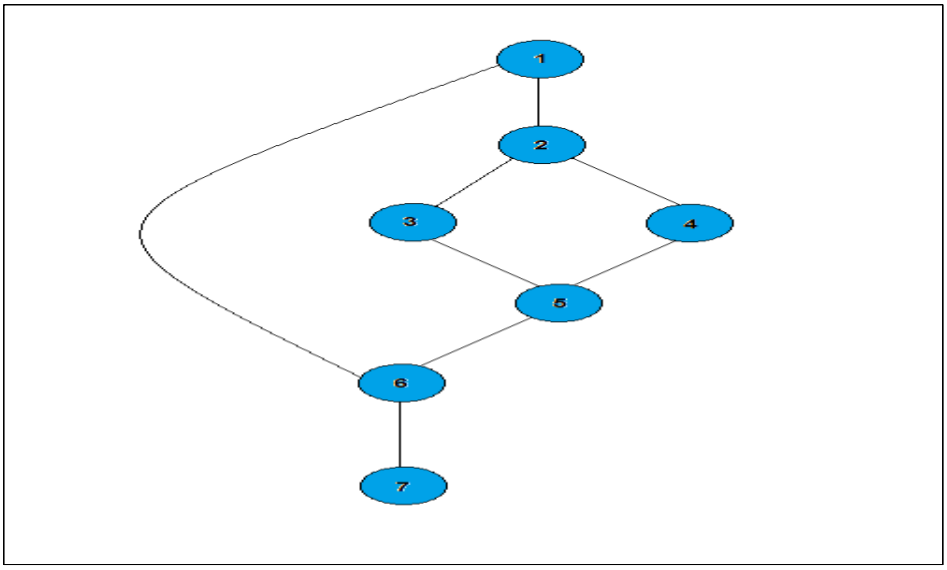
There are several methods:

1. **Cyclomatic complexity = edges - node's+ 2p**

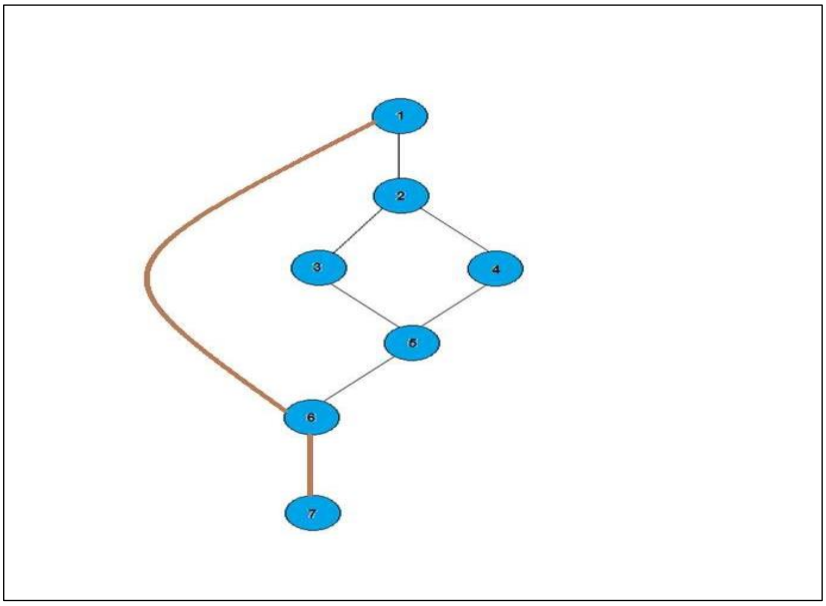
Determine Cyclomatic complexity

Cyclomatic complexity = edges - nodes + 2p

p = number of unconnected parts of the graph.



Cyclomatic complexity = 8-7+ 2*1= 3.



Find a basis set of paths

- Path 1: 1, 2, 3, 5, 6, 7
- Path 2: 1, 2, 4, 5, 6, 7
- Path 3: 1, 6, 7

Generate test cases for each path

We have 3 paths so we need at least one test case to cover each path.

Write test case for these paths .

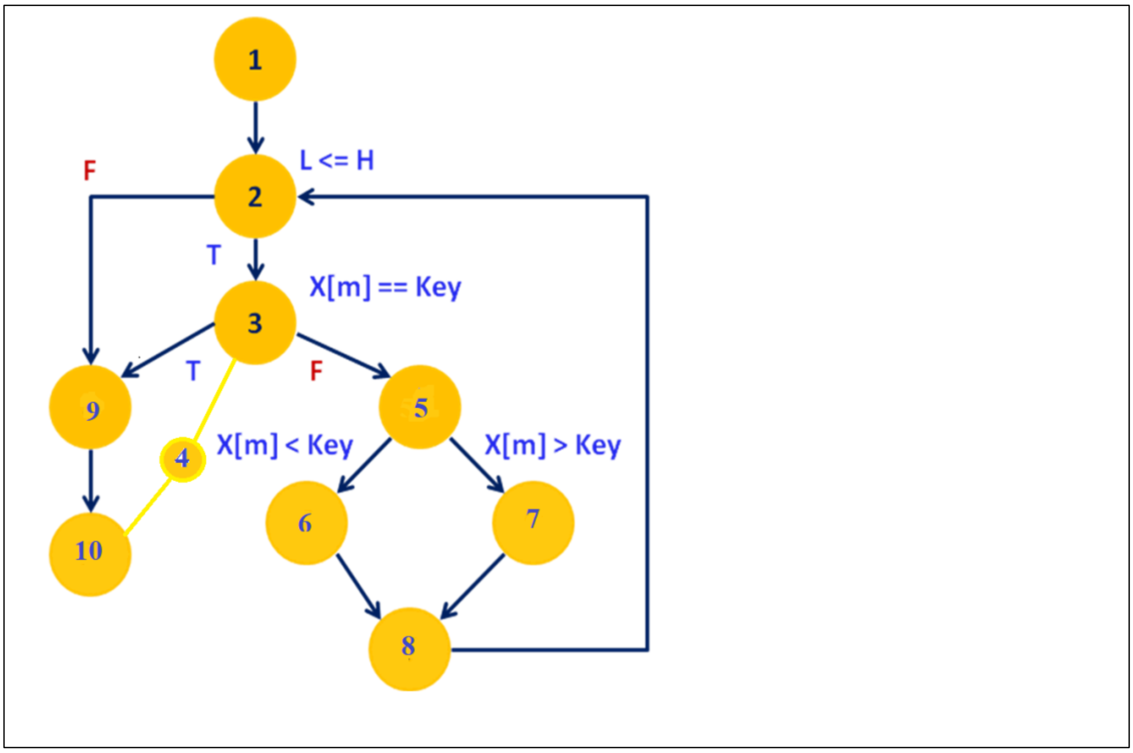
Program Code

```
#include<stdio.h>
int binsrc(int x[],int low, int high, int key)
 {
int mid;
 while(low<=high)
{
 mid=(low+high)/2;
  if(x[mid]==key)
return mid;
 if(x[mid]<key)
  low=mid+1;
  else
  high=mid-1;
 }
return -1;
 }


int main()
 {
 int a[20],key,i,n,succ;
 printf("Enter the n value");
scanf("%d",&n);
 if(n>0)
 {
 printf("enter the elements in ascending order\n");
for(i=0;i<n;i++)
 scanf("%d",&a[i]);
 printf("enter the key element to be searched\n");
scanf("%d",&key);
succ=binsrc(a,0,n-1,key);
 if(succ>=0)
 printf("Element found in position = %d\n",succ+1);
else
 printf("Element not found \n");
}
Else
printf("Number of element should be greater than zero\n");
return 0;
}
```

Binary Search function with line number

```
int binsrc(int x[],int low, int high, int key)
{
int mid;                                       1
while(low<=high)                               2
{
mid=(low+high)/2; if(x[mid]==key)              3
return mid;                                    4
if(x[mid]<key)                                 5
low=mid+1;                                     6
else
high=mid-1;                                    7
}                                              8
return -1;                                     9
}                                              10
```

Program Graph



Test cases

1.Cyclomatic complexity=Edges-nodes+2(P)

12-10+2(1)=4 Test cases

2.Independent paths=

1.1-2-9-10------->(10,20,30,40,50) Empty list

2.1-2-3-4-10------(10,20,30,40,50) Success

3.1-2-3-5-6-8 -----(10,20,30,40,50) Success

4.1-2-3-5-7-8 -----(10,20,30,40,50) Success