INDEX

SNO	PRACTICAL	PRACTICAL DATE	SUBMISSION DATE	SIGNATURE
1.	To Prepare Test Plan for the implemented system under test. The Test Plan shall be based onSystem Requirement Specification. The Test plan consists of following issues. a. Purpose of the test. /Location and schedule of the test. b. Test descriptions. /Pass and Fail Criteria.			
2.	To identify and narrate Test cases, Test scripts/procedures and Test incident report identifier for the system under test. Refer Use case analysis document to prepare mentioned/ identified test documents.			
3.	To perform Unit testing especially indicating the traced Independent data paths, control paths and Error handling paths. Prepare control flow graphs for the unit under test. Compute the cyclomatic complexity of the unit			
4.	To perform Data Flow testing for the Program Segments by identifying the Definition-Use chain and type of data flow anomaly			
5.	To perform Mutation Analysis of the Program Segments along with mutant history, mutation score and type of mutation by using any Code analysis Tool / Mutation Testing Tool.			
6.	To perform Black-Box Testing for all the units contained in the architectural segments using Equivalence Partitioning, Boundary Value Analysis and Orthogonal Array testing methods. Study exploratory Testing for the Module under Test and merits/demerits of this technique.			
7.	To perform Regression Testing of the System under construction with Unit and Integration profiles by using any Functional Testing Tool.			

<LINUX GUI INSTALLER WITH BUG FINDER>

TEST PLAN

Version *<1.0>*

<09/01/2024>

VERSION HISTORY

Version #	Implemented By	Revision Date	Approved By	Approval Date	Reason
1.0	*<Kajal Yadav>*	*<09/01/2024 >*	*<Supreet Kaur>*	*<mm/dd/yy>*	Test Plan draft
1.0	*<Chandan Goyal>*	*<09/01/2024 >*	*<Supreet Kaur>*	*<mm/dd/yy>*	Test Plan draft
1.0	*<Ritik Kashyap>*	*<09/01/2024 >*	*<Supreet Kaur>*	*<mm/dd/yy>*	Test Plan draft

UP Template Version: 12/31/07

TABLE OF CONTENTS

INTRODUCTION

1.1 PURPOSE OF THE TEST PLAN DOCUMENT

The document proposes the development of a GUI Application Installer for Linux with an integrated Bug Finder mechanism. The main goals are to create a user-friendly graphical interface for installing software on Linux, and to incorporate bug detection tools to enhance system stability. It discusses the rationale, objectives, feasibility, methodology, and expected outcomes of this project. The installer aims to simplify the software installation process for Linux users while proactively identifying and reporting potential bugs or vulnerabilities. Overall, it seeks to improve the user experience and reliability of software installations in the Linux ecosystem.

1.2 OBJECTIVES AND TASKS

1. FUNCTIONAL TESTING

- verify that the graphical user interface (gui) functions as expected, including proper display of elements, navigation, and user interactions.
- Test the installation process for various applications across different linux distributions and package formats.
- Validate the bug-finding mechanism's ability to detect and report bugs or vulnerabilities during the installation process.

2. USABILITY TESTING

- Evaluate the user-friendliness and intuitiveness of the gui installer.
- Assess the clarity of instructions, error messages, and feedback provided to users.
- Conduct user acceptance testing (uat) with representative users to gather feedback on the overall experience.

3. SECURITY TESTING

- Test the secure installation practices implemented to mitigate potential security risks.
- Verify proper authentication and authorization mechanisms for users and administrators.
- Evaluate the installer's handling of dependencies and potential conflicts during installation.

4. COMPATIBILITY TESTING

- Test the installer's compatibility with various linux distributions, package formats, and system configurations.
- Ensure cross-distribution compatibility and seamless operation across different environments.

5. INTEGRATION TESTING

- Verify the seamless integration of the bug-finding mechanism with the installer.

- Test the interaction between the installer and external bug-finding tools or libraries.

6. PERFORMANCE TESTING

- Assess the installer's performance under different load conditions and with varying numbers of applications.
- Measure installation time, resource utilization, and responsiveness of the gui.

1.3 SCOPE

The scope of testing should encompass the entire gui application installer in linux with bug finder application, including the graphical user interface, installation process, bug-finding mechanism, security features, compatibility, and integration with external components.

1.4 TESTING STRATEGY

The testing strategy should involve a combination of manual and automated testing approaches:

1. **Manual testing:**
   - Functional testing, usability testing, and user acceptance testing will primarily involve manual testing by dedicated testers and representative users.
   - Exploratory testing to identify edge cases and potential issues not covered by predefined test cases.
2. **Automated testing:**
   - Develop automated test scripts and frameworks for regression testing, compatibility testing across distributions, and performance testing.
   - Leverage existing bug-finding tools or develop custom scripts to automate the testing of the bug-finding mechanism.

1.5 HARDWARE REQUIREMENTS

The hardware requirements for testing should align with the target deployment environments and include:

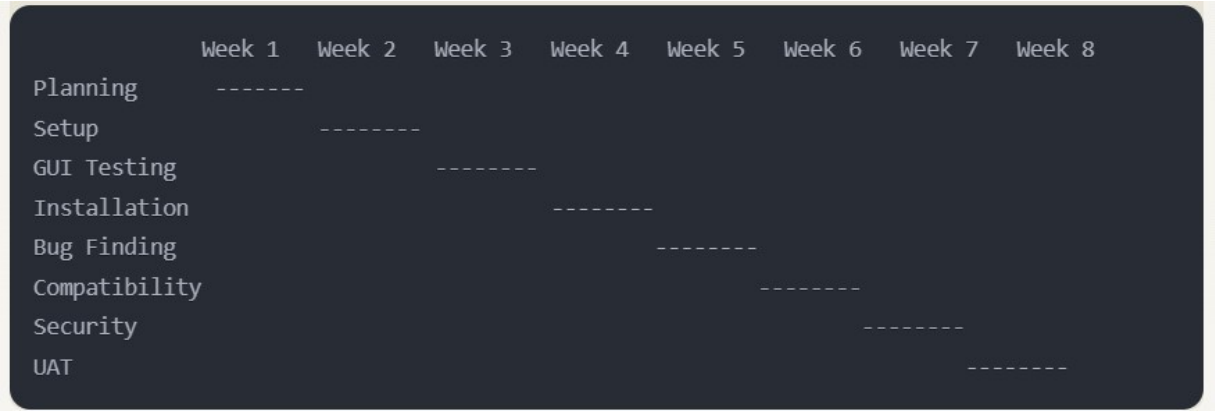
- Linux systems with various distributions (e.g., Ubuntu, Fedora, CentOS) and hardware configurations.
- Dedicated testing machines or virtual environments for isolated testing and bug reproduction.
- Sufficient storage space and RAM to accommodate the installation of multiple applications and the testing framework.

1.6 ENVIRONMENT REQUIREMENTS

The testing environment should closely replicate the production environment to ensure accurate and reliable results. This may include:

- Setting up testing environments with different Linux distributions and package formats.
- Configuring test environments with varying system configurations, dependencies, and software stacks.
- Implementing test data management and version control for test cases, scripts, and bug reports.
- Establishing a centralized issue tracking system for bug reporting and management.

1.7 GANTT CHART



2 COMPATIBILITY TESTING

1.8 TEST RISKS / ISSUES

One of the key issues in compatibility testing would be ensuring the installer works seamlessly across different package management systems and configurations used by different Linux distributions.

1.9 ITEMS TO BE TESTED / NOT TESTED

the key items/features to be tested within the scope of this test plan include the GUI interface, package installation process across distributions, bug finder integration, user authentication/authorization, installation logs, and installation rollback mechanisms. Testing will be performed by relevant teams (UI, compatibility, QA, security, etc.) following established quality standards.

Item to Test	Test Description	Test Date	Responsibility

GUI Interface	Verify the usability, layout, and functionality of the graphical user interface for software installation. Test the process flow, input validations, and error handling.		
Package Installation	Test the installation process for various software packages across different Linux distributions. Ensure compatibility with package managers like APT, YUM, etc. Validate successful installations and dependency management.		Distribution Compatibility Testing Team
Authentication/Authorization	Test user authentication and authorization mechanisms to ensure secure access control and permission management for software installations.		Security Testing Team
Installation Logs	Verify the generation of detailed installation logs for debugging purposes, including error logging and traceability.		Logging & Monitoring Team
Installation Rollback	Test the ability to rollback or uninstall software in case of errors or issues during the installation process. Validate data integrity and system state after rollback.		Recovery & Rollback Testing Team

1.10 TEST APPROACH(S)

The overall test approach will involve functional testing of the installation process, GUI, and bug finder integration, as well as cross-distribution compatibility testing, security testing of authentication/authorization mechanisms, and usability testing of the GUI interface. Automated testing frameworks may be utilized for comprehensive coverage.

1.11 TEST REGULATORY / MANDATE CRITERIA

No test regulatory or mandate criteria are described for this system in the given document.

1.12 TEST PASS / FAIL CRITERIA

The pass/fail criteria likely involve verifying correct software installation, bug detection, security, logging, and rollback mechanisms across Linux distributions.

1.13 TEST ENTRY / EXIT CRITERIA

Test entry criteria could involve development completion and test readiness, while exit criteria may depend on meeting coverage goals, resolving defects, and user acceptance.

1.14 TEST DELIVERABLES

The test deliverables will include test reports, bug reports, and test coverage reports.

1.15 TEST SUSPENSION / RESUMPTION CRITERIA

The document does not mention any test suspension or resumption criteria for this project.

1.16 TEST ENVIRONMENTAL / STAFFING / TRAINING NEEDS

The document mentions the hardware and software requirements needed for the proposed work, which would include the testing environment needs.

CONFORMANCE TESTING

1.17 TEST RISKS / ISSUES

The project aims to create a user-friendly GUI installer for Linux with integrated bug-finding capabilities to simplify software installation and ensure system stability.

1.18 ITEMS TO BE TESTED / NOT TESTED

[Describe the items/features/functions to be tested that are within the scope of this test plan. Include a description of how they will be tested, when, by whom, and to what quality standards. Also include a description of those items agreed not to be tested.]

Item to Test	Test Description	Test Date	Responsibility
Frontend Graphical User Interface (GUI)	Test the graphical user interface, layout, usability, and functionality of the frontend application for installing apps.		

1.19 TEST APPROACH(S)

[Describe the overall testing approach to be used to test the project's product. Provide an outline of any planned tests.]

1.20 TEST REGULATORY / MANDATE CRITERIA

[Describe any regulations or mandates that the system must be tested against.]

1.21 TEST PASS / FAIL CRITERIA

[Describe the criteria used to determine if a test item has passed or failed its test.]

1.22 TEST ENTRY / EXIT CRITERIA

[Describe the entry and exit criteria used to start testing and determine when to stop testing.]

1.23 TEST DELIVERABLES

[Describe the deliverables that will result from the testing process (documents, reports, charts, etc.).]

1.24 TEST SUSPENSION / RESUMPTION CRITERIA

[Describe the suspension criteria that may be used to suspend all or portions of testing. Also describe the resumption criteria that may be used to resume testing.]

1.25 TEST ENVIRONMENTAL / STAFFING / TRAINING NEEDS

[Describe any specific requirements needed for the testing to be performed (hardware/software, staffing, skills training, etc).)]

FUNCTIONAL TESTING

1.26 TEST RISKS / ISSUES

[Describe the risks associated with product testing or provide a reference to a document location where it is stored. Also outline appropriate mitigation strategies and contingency plans.]

1.27 ITEMS TO BE TESTED / NOT TESTED

[Describe the items/features/functions to be tested that are within the scope of this test plan. Include a description of how they will be tested, when, by whom, and to what quality standards. Also include a description of those items agreed not to be tested.]

Item to Test	Test Description	Test Date	Responsibility

1.28 TEST APPROACH(S)

[Describe the overall testing approach to be used to test the project's product. Provide an outline of any planned tests.]

1.29 TEST REGULATORY / MANDATE CRITERIA

[Describe any regulations or mandates that the system must be tested against.]

1.30 TEST PASS / FAIL CRITERIA

[Describe the criteria used to determine if a test item has passed or failed its test.]

1.31 TEST ENTRY / EXIT CRITERIA

[Describe the entry and exit criteria used to start testing and determine when to

stop testing.]

1.32 TEST DELIVERABLES

[Describe the deliverables that will result from the testing process (documents, reports, charts, etc.).]

1.33 TEST SUSPENSION / RESUMPTION CRITERIA

[Describe the suspension criteria that may be used to suspend all or portions of testing. Also describe the resumption criteria that may be used to resume testing.]

1.34 TEST ENVIRONMENTAL / STAFFING / TRAINING NEEDS

[Describe any specific requirements needed for the testing to be performed (hardware/software, staffing, skills training, etc).)]

LOAD TESTING

1.35 TEST RISKS / ISSUES

[Describe the risks associated with product testing or provide a reference to a document location where it is stored. Also outline appropriate mitigation strategies and contingency plans.]

1.36 ITEMS TO BE TESTED / NOT TESTED

[Describe the items/features/functions to be tested that are within the scope of this test plan. Include a description of how they will be tested, when, by whom, and to what quality standards. Also include a description of those items agreed not to be tested.]

Item to Test	Test Description	Test Date	Responsibility

1.37 TEST APPROACH(S)

[Describe the overall testing approach to be used to test the project's product. Provide an outline of any planned tests.]

1.38 TEST REGULATORY / MANDATE CRITERIA

[Describe any regulations or mandates that the system must be tested against.]

1.39 TEST PASS / FAIL CRITERIA

[Describe the criteria used to determine if a test item has passed or failed its test.]

1.40 TEST ENTRY / EXIT CRITERIA

[Describe the entry and exit criteria used to start testing and determine when to stop testing.]

1.41 TEST DELIVERABLES

[Describe the deliverables that will result from the testing process (documents, reports, charts, etc.).]

1.42 TEST SUSPENSION / RESUMPTION CRITERIA

[Describe the suspension criteria that may be used to suspend all or portions of testing. Also describe the resumption criteria that may be used to resume testing.]

1.43 TEST ENVIRONMENTAL / STAFFING / TRAINING NEEDS

[Describe any specific requirements needed for the testing to be performed (hardware/software, staffing, skills training, etc).)]

PERFORMANCE TESTING

1.44 TEST RISKS / ISSUES

[Describe the risks associated with product testing or provide a reference to a document location where it is stored. Also outline appropriate mitigation strategies and contingency plans.]

1.45 ITEMS TO BE TESTED / NOT TESTED

[Describe the items/features/functions to be tested that are within the scope of this test plan. Include a description of how they will be tested, when, by whom, and to what quality standards. Also include a description of those items agreed not to be tested.]

Item to Test	Test Description	Test Date	Responsibility

1.46 TEST APPROACH(S)

[Describe the overall testing approach to be used to test the project's product. Provide an outline of any planned tests.]

1.47 TEST REGULATORY / MANDATE CRITERIA

[Describe any regulations or mandates that the system must be tested against.]

1.48 TEST PASS / FAIL CRITERIA

[Describe the criteria used to determine if a test item has passed or failed its test.]

1.49 TEST ENTRY / EXIT CRITERIA

[Describe the entry and exit criteria used to start testing and determine when to stop testing.]

1.50 TEST DELIVERABLES

[Describe the deliverables that will result from the testing process (documents,

reports, charts, etc.).]

1.51 TEST SUSPENSION / RESUMPTION CRITERIA

[Describe the suspension criteria that may be used to suspend all or portions of testing. Also describe the resumption criteria that may be used to resume testing.]

1.52 TEST ENVIRONMENTAL / STAFFING / TRAINING NEEDS

[Describe any specific requirements needed for the testing to be performed (hardware/software, staffing, skills training, etc).)]

REGRESSION TESTING

1.53 TEST RISKS / ISSUES

[Describe the risks associated with product testing or provide a reference to a document location where it is stored. Also outline appropriate mitigation strategies and contingency plans.]

1.54 ITEMS TO BE TESTED / NOT TESTED

[Describe the items/features/functions to be tested that are within the scope of this test plan. Include a description of how they will be tested, when, by whom, and to what quality standards. Also include a description of those items agreed not to be tested.]

Item to Test	Test Description	Test Date	Responsibility

1.55 TEST APPROACH(S)

[Describe the overall testing approach to be used to test the project's product. Provide an outline of any planned tests.]

1.56 TEST REGULATORY / MANDATE CRITERIA

[Describe any regulations or mandates that the system must be tested against.]

1.57 TEST PASS / FAIL CRITERIA

[Describe the criteria used to determine if a test item has passed or failed its test.]

1.58 TEST ENTRY / EXIT CRITERIA

[Describe the entry and exit criteria used to start testing and determine when to stop testing.]

1.59 TEST DELIVERABLES

[Describe the deliverables that will result from the testing process (documents, reports, charts, etc.).]

1.60 TEST SUSPENSION / RESUMPTION CRITERIA

[Describe the suspension criteria that may be used to suspend all or portions of testing. Also describe the resumption criteria that may be used to resume testing.]

1.61 TEST ENVIRONMENTAL / STAFFING / TRAINING NEEDS

[Describe any specific requirements needed for the testing to be performed (hardware/software, staffing, skills training, etc).)]

STRESS TESTING

1.62 TEST RISKS / ISSUES

[Describe the risks associated with product testing or provide a reference to a document location where it is stored. Also outline appropriate mitigation strategies and contingency plans.]

1.63 ITEMS TO BE TESTED / NOT TESTED

[Describe the items/features/functions to be tested that are within the scope of this test plan. Include a description of how they will be tested, when, by whom, and to what quality standards. Also include a description of those items agreed not to be tested.]

Item to Test	Test Description	Test Date	Responsibility

1.64 TEST APPROACH(S)

[Describe the overall testing approach to be used to test the project's product. Provide an outline of any planned tests.]

1.65 TEST REGULATORY / MANDATE CRITERIA

[Describe any regulations or mandates that the system must be tested against.]

1.66 TEST PASS / FAIL CRITERIA

[Describe the criteria used to determine if a test item has passed or failed its test.]

1.67 TEST ENTRY / EXIT CRITERIA

[Describe the entry and exit criteria used to start testing and determine when to stop testing.]

1.68 TEST DELIVERABLES

[Describe the deliverables that will result from the testing process (documents, reports, charts, etc.).]

1.69 TEST SUSPENSION / RESUMPTION CRITERIA

[Describe the suspension criteria that may be used to suspend all or portions of

testing. Also describe the resumption criteria that may be used to resume testing.]

1.70 TEST ENVIRONMENTAL / STAFFING / TRAINING NEEDS

[Describe any specific requirements needed for the testing to be performed (hardware/software, staffing, skills training, etc).)]

SYSTEM TESTING

1.71 TEST RISKS / ISSUES

[Describe the risks associated with product testing or provide a reference to a document location where it is stored. Also outline appropriate mitigation strategies and contingency plans.]

1.72 ITEMS TO BE TESTED / NOT TESTED

[Describe the items/features/functions to be tested that are within the scope of this test plan. Include a description of how they will be tested, when, by whom, and to what quality standards. Also include a description of those items agreed not to be tested.]

Item to Test	Test Description	Test Date	Responsibility

1.73 TEST APPROACH(S)

[Describe the overall testing approach to be used to test the project's product. Provide an outline of any planned tests.]

1.74 TEST REGULATORY / MANDATE CRITERIA

[Describe any regulations or mandates that the system must be tested against.]

1.75 TEST PASS / FAIL CRITERIA

[Describe the criteria used to determine if a test item has passed or failed its test.]

1.76 TEST ENTRY / EXIT CRITERIA

[Describe the entry and exit criteria used to start testing and determine when to stop testing.]

1.77 TEST DELIVERABLES

[Describe the deliverables that will result from the testing process (documents, reports, charts, etc.).]

1.78 TEST SUSPENSION / RESUMPTION CRITERIA

[Describe the suspension criteria that may be used to suspend all or portions of testing. Also describe the resumption criteria that may be used to resume testing.]

1.79 TEST ENVIRONMENTAL / STAFFING / TRAINING NEEDS

[Describe any specific requirements needed for the testing to be performed (hardware/software, staffing, skills training, etc).)]

UNIT TESTING

1.80 TEST RISKS / ISSUES

[Describe the risks associated with product testing or provide a reference to a document location where it is stored. Also outline appropriate mitigation strategies and contingency plans.]

1.81 ITEMS TO BE TESTED / NOT TESTED

[Describe the items/features/functions to be tested that are within the scope of this test plan. Include a description of how they will be tested, when, by whom, and to what quality standards. Also include a description of those items agreed not to be tested.]

Item to Test	Test Description	Test Date	Responsibility

1.82 TEST APPROACH(S)

[Describe the overall testing approach to be used to test the project's product. Provide an outline of any planned tests.]

1.83 TEST REGULATORY / MANDATE CRITERIA

[Describe any regulations or mandates that the system must be tested against.]

1.84 TEST PASS / FAIL CRITERIA

[Describe the criteria used to determine if a test item has passed or failed its test.]

1.85 TEST ENTRY / EXIT CRITERIA

[Describe the entry and exit criteria used to start testing and determine when to stop testing.]

1.86 TEST DELIVERABLES

[Describe the deliverables that will result from the testing process (documents, reports, charts, etc.).]

1.87 TEST SUSPENSION / RESUMPTION CRITERIA

[Describe the suspension criteria that may be used to suspend all or portions of testing. Also describe the resumption criteria that may be used to resume testing.]

1.88 TEST ENVIRONMENTAL / STAFFING / TRAINING NEEDS

[Describe any specific requirements needed for the testing to be performed

(hardware/software, staffing, skills training, etc).)]

USER ACCEPTANCE TESTING

1.89 TEST RISKS / ISSUES

[Describe the risks associated with product testing or provide a reference to a document location where it is stored. Also outline appropriate mitigation strategies and contingency plans.]

1.90 ITEMS TO BE TESTED / NOT TESTED

[Describe the items/features/functions to be tested that are within the scope of this test plan. Include a description of how they will be tested, when, by whom, and to what quality standards. Also include a description of those items agreed not to be tested.]

Item to Test	Test Description	Test Date	Responsibility

1.91 TEST APPROACH(S)

[Describe the overall testing approach to be used to test the project's product. Provide an outline of any planned tests.]

1.92 TEST REGULATORY / MANDATE CRITERIA

[Describe any regulations or mandates that the system must be tested against.]

1.93 TEST PASS / FAIL CRITERIA

[Describe the criteria used to determine if a test item has passed or failed its test.]

1.94 TEST ENTRY / EXIT CRITERIA

[Describe the entry and exit criteria used to start testing and determine when to stop testing.]

1.95 TEST DELIVERABLES

[Describe the deliverables that will result from the testing process (documents, reports, charts, etc.).]

1.96 TEST SUSPENSION / RESUMPTION CRITERIA

[Describe the suspension criteria that may be used to suspend all or portions of testing. Also describe the resumption criteria that may be used to resume testing.]

1.97 TEST ENVIRONMENTAL / STAFFING / TRAINING NEEDS

[Describe any specific requirements needed for the testing to be performed (hardware/software, staffing, skills training, etc).)]

TEST PLAN APPROVAL

The undersigned acknowledge they have reviewed the *<Project Name>* **Test Plan** document and agree with the approach it presents. Any changes to this Requirements Definition will be coordinated with and approved by the undersigned or their designated representatives.

[List the individuals whose signatures are required. Examples of such individuals are Business Steward, Technical Steward, and Project Manager. Add additional signature lines as necessary.]

Signature: _____ Date: _____

Print Name: _____

Title: _____

Role: _____

Signature: _____ Date: _____

Print Name: _____

Title: _____

Role: _____

Signature: _____ Date: _____

Print Name: _____

Title: _____

Role: _____

Appendix A: References

[Insert the name, version number, description, and physical location of any documents referenced in this document. Add rows to the table as necessary.]

The following table summarizes the documents referenced in this document.

Document Name and Version	Description	Location
<Document Name and Version Number>	*[Provide description of the document]*	*<URL or Network path where document is located>*

Appendix B: Key Terms

[Insert terms and definitions used in this document. Add rows to the table as necessary. Follow the link below to for definitions of project management terms and acronyms used in this and other documents.

http://www2.cdc.gov/cdcup/library/other/help.htm

The following table provides definitions for terms relevant to this document.

Term	Definition
[Insert Term]	*[Provide definition of the term used in this document.]*
[Insert Term]	*[Provide definition of the term used in this document.]*
[Insert Term]	*[Provide definition of the term used in this document.]*

PRACTICAL - 2

Aim :- To identify and narrate Test cases, Test scripts/procedures and Test incident report identifier for the system under test. Refer Use case analysis document to prepare mentioned/ identified test documents.

Program :

```
#include <iostream>
#include <cmath>
using namespace std;

double calculate_telephone_bill(int num_calls) {
    double bill = 0;
    if (num_calls <= 120) {
        bill = 300;
    } else {
        bill = 300; // Minimum bill
        if (num_calls <= 190) {
            bill += (num_calls - 120) * 1;
        } else {
            bill += 70 * 1; // For first 70 calls after 120
            if (num_calls <= 240) {
                bill += (num_calls - 190) * 0.8;
            } else {
                bill += 50 * 0.8; // For first 50 calls after 190
                bill += (num_calls - 240) * 0.4;
            }
        }
    }
    return bill;
}
```

Testcases for telephone bill:

	A	B	C	D	E	F	G	H
1	Name- Kajal Yadav , URN - 2203585							
2	Test Case ID	Test Case Description	Input Data	Expected Outpu	Actual Output	Pass/Fail	Remarks	
3	1	Zero calls	0	300	300	Pass		
4	2	Calls within initial range	199	300	300	Pass		
5	3	Boundary case: 120 calls	120	300	300	Pass		
6	4	Invalid: Negative calls	-10	Error Message	N/A	Pass		
7	5	Invalid: Non-integer input	125.5	Error Message	N/A	Pass		
8	6	Calls in range (121-190)	150	320	320	Pass		
9	7	Boundary case: 190 calls	190	360	360	Pass		
10	8	Calls in range (191-240)	200	368	368	Pass		
11	9	Boundary case: 240 calls	240	400	402	Fail		
12	10	Calls beyond 240	250	408	408	Pass		
13	11	Large input	1000	860	860	Pass		
14	12	Failing case: Incorrect bill calculation	150	320	325	Fail	Billing logic error	
15	13	Failing case: Boundary case	191	360.8	361	Fail	Rounding error	
16	14	Failing case: Large input	10000	4260	4300	Fail	Overflow error	
17	15	Failing case: Invalid input	"abc"	Error Message	N/A	Pass		
18	16	Edge case: Maximum possible calls	2147483647	858993459.8	N/A	Pass		
19	17	Calls in range (121-190)	130	310	310	Pass		
20	18	Calls in range (191-240)	210	376	376	Pass		
21	19	Calls beyond 240	300	460	460	Pass		
22	20	Boundary case: 121 calls	121	301	301	Pass		
23	21	Failing case: Incorrect bill calculation	180	350	360	Fail	Billing logic error	
24	22	Failing case: Boundary case	241	400.1	401	Fail	Rounding error	
25	23	Failing case: Large input	5000	2260	2300	Fail	Overflow error	
26	24	Failing case: Invalid input	"xyz"	Error Message	N/A	Pass		
27	25	Edge case: Minimum possible calls	-2.147E+09	Error Message	N/A	Pass		
28	26	Calls in range (121-190)	170	340	340	Pass		
29	27	Calls in range (191-240)	220	384	384	Pass		
30	28	Call Beyound 240	350	500	500	Pass		
31	29	Boundary case: 189 calls	189	359	359	Pass		
32	30	Boundary case: 239 calls	239	399.2	399	Pass		
33	31	Failing case: Incorrect bill calculation	200	368	370	Fail	Billing logic error	
34	32	Failing case: Boundary case	190	360	361	Fail	Rounding error	
35	33	Failing case: Large input	2000	8260	8300	Fail	Overflow error	
36	34	Failing case: Invalid input	"123abc"	Error Message	N/A	Pass		
37	35	Edge case: Zero calls	0	300	300	Pass		
38	36	Calls in range (121-190)	140	320	320	Pass		
39	37	Calls in range (191-240)	230	392	392	Pass		
40	38	Calls beyond 240	400	540	540	Pass		
41	39	Boundary case: 122 call	122	302	302	Pass		
42	40	Boundary case: 238 calls	238	398.4	398	Pass		
43	41	Failing case: Incorrect bill calculation	250	408	410	Fail	Billing logic error	
44	42	Failing case: Boundary case	241	400.4	401	Fail	Rounding error	
45	43	Failing case: Large input	3000	12260	12300	Fail	Overflow error	
46	44	Failing case: Invalid input	"1.5"	Error Message	N/A	Pass		
47	45	Edge case: Maximum possible calls	2147483647	858993459.8	N/A	Pass		
48	46	Calls in range (121-190)	160	330	330	Pass		
49	47	Calls in range (191-240)	215	379	379	Pass		
50	48	Calls beyond 240	450	560	560	Pass		
51	49	Boundary case: 123 calls	123	303	303	Pass		
52	50	Boundary case: 237 calls	237	397.6	397	Pass		

PRACTICAL :- 3

Aim : To perform Unit testing especially indicating the traced Independent data paths, control paths and Error handling paths. Prepare control flow graphs for the unit under test. Compute the cyclomatic complexity of the unit.

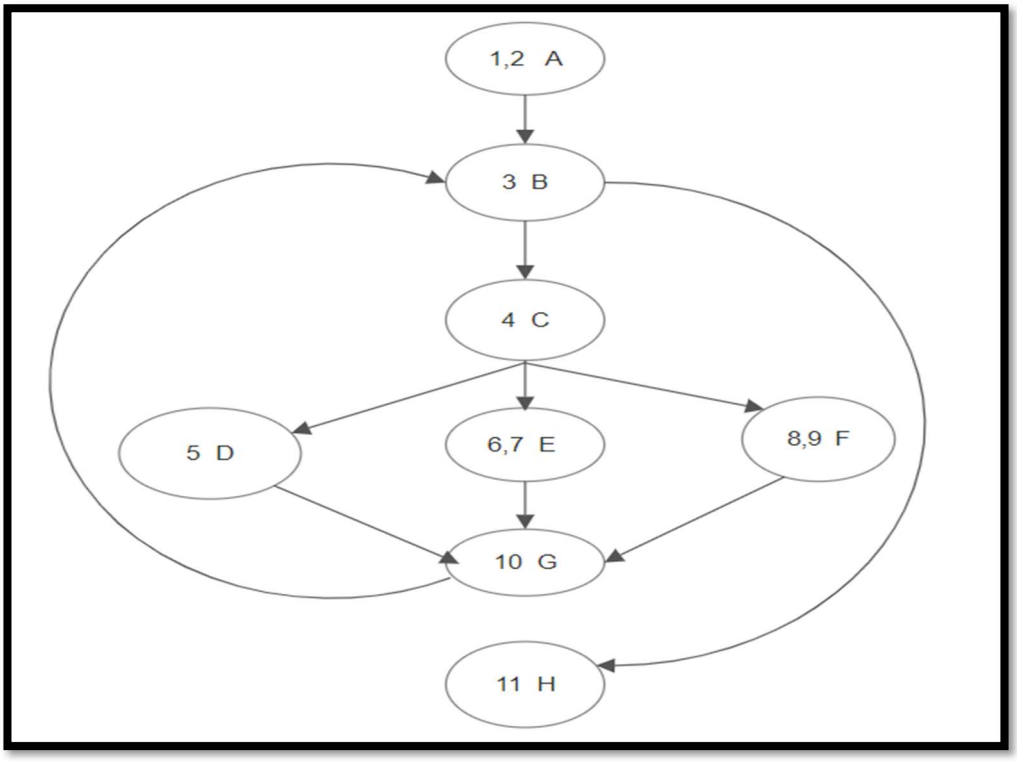
Problem :-

```
main() {
   char string[80];
   int index;

1.  printf("Enter a string: ");
2. scanf("%s", string);

3. for (index = 0; string[index] != '\0'; ++index)
4.   if (string[index] >= '0' && string[index] <= '9')
5.     printf("%c is a digit\n", string[index]);
6.    else if ((string[index] >= 'A' && string[index] <= 'Z') || (string[index] >= 'a' &&
   string[index] <= 'z')) {
7.     printf("%c is an alphabet\n", string[index]);
8.   else
9.     printf("%c is a special character\n", string[index]);
10.  }
11. }
```

DD graph :



Cyclomatic Complexity :

1. V(G) = e-n+2*P
   = 10-8+2
   = 4

2. V(G) = Number of Predicate nodes + 1
   = 3(Nodes B,C)+1
   = 4
   Node C is a multiple IF THEN ELSE; so for finding out the number predicate nodes for this case, follow this formula:

   Number of predicate nodes = Number of links out of main node – 1
   = 3-1
   = 2 (For node C)

3. V(G) = Number of regions
   = 4

Independent Path:

1. A-B-H
2. A-B-C-D-G-B-H
3. A-B-C-E-G-B-H
4. A-B-C-F-G-B-H

Testcases for Independent path :

	A	B	C	D
4	Test Case ID	Input Line	Expected Output	Independent Paths Coveredby Test Case
5	1	0987	0 is a digit	A-B-C-D-G-B-H
6			9 is a digit	A-B-H
7			8 is a digit	
8			7 is a digit	
9	2	AzxG	A is a alphabet	A-B-C-E-G-B-H
10			Z is a alphabet	A-B-H
11			X is a alphabet	
12			G is a alphabet	
13	3	@ #	@ is a special character	A-B-C-F-G-B-H
14			# is a special character	A-B-H
15				
16				
17	Name- Kajal Yadav		URN- 2203585	

PRACTICAL:- 4

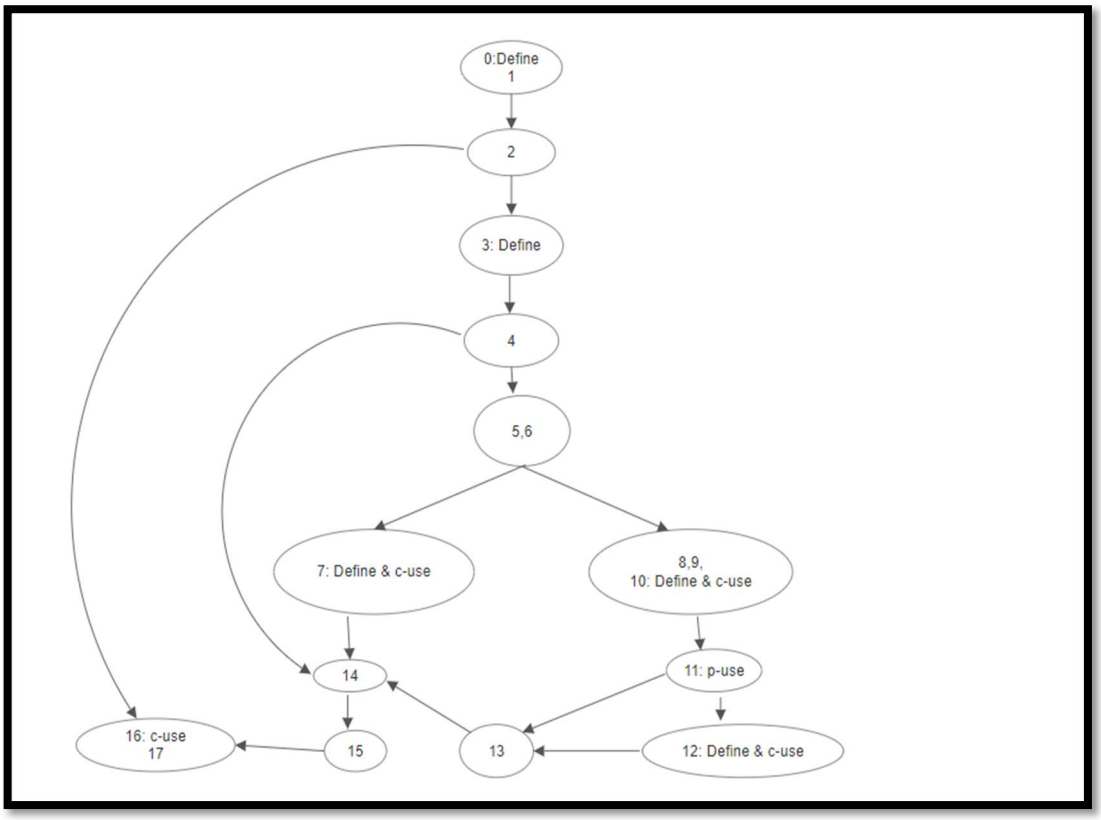
Aim: To perform Data Flow testing for the Program Segments by identifying the Definition-Use chain and type of data flow anomaly.

Program :

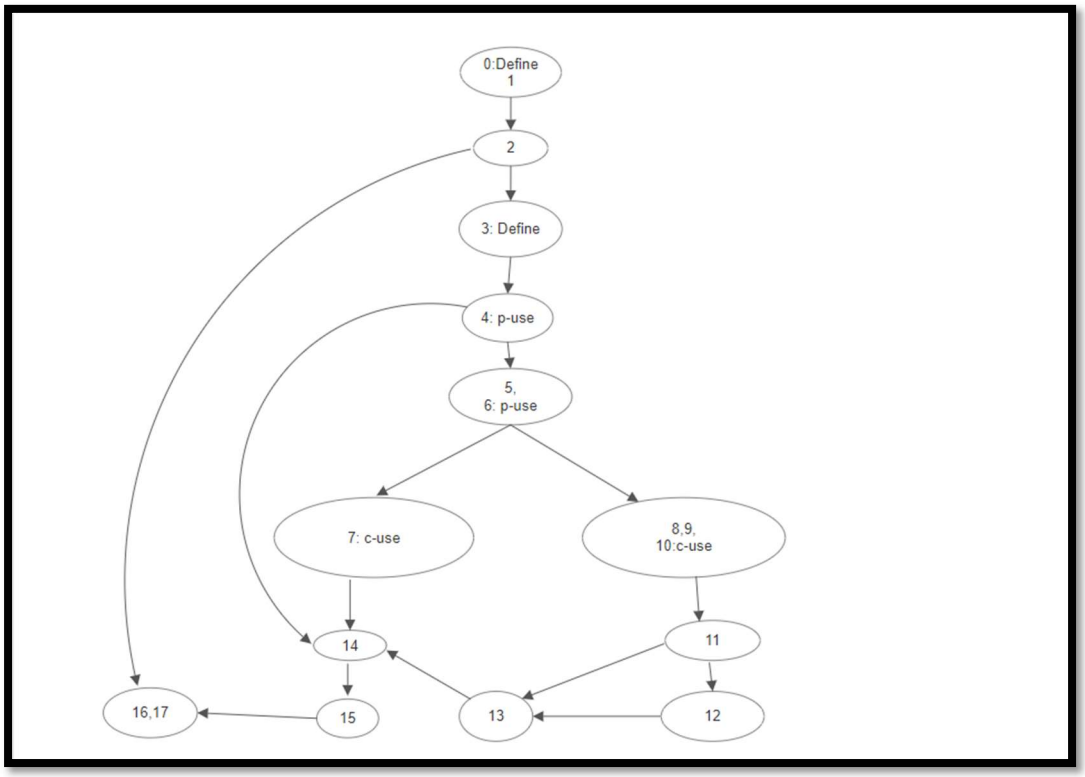
```
   Main()
    {
0.  double payment = 0;
1.  scanf("%d", &work);
2.  if (work > 0) {
3.      payment = 40;
4.      if (work > 20)
5.       {
6.          if (work <= 30)
7.              payment = payment + (work - 25) * 0.5;
8.          else
9.          {
10.              payment = payment + 50 + (work - 30) * 0.1;
11.              if (payment >= 3000) {
12.                  payment = payment * 0.9;
14.           }
14.    }
15.    }
16. printf("Final payment: %.2f", payment);
17.    }
```

Data Flow Graph :

1. Data flow graph for “Payment”



2. Data flow graph for “work”



Testcase on Excelsheet:-

	A	B	C	D	E	F
3						
4	Test CaseID	Input(work)	Expected Output(payment)	Actual Output(payment)	Pass/Fail	Remarks
5	1	-10	0	0	Fail	
6	2	0	0	0	Pass	
7	3	10	40	40	Pass	
8	4	15	40	40	Pass	
9	5	19	40	20	Pass	
10	6	20	40	0	Pass	
11	7	20	40	0	Fail	Boundary case
12	8	21	45	45	Pass	
13	9	25	45	45	Pass	
14	10	30	52.5	52.5	Pass	
15	11	30	52.5	103.5	Pass	Boundary case
16	12	35	103.5	115	Pass	
17	13	50	115	2699.1	Pass	
18	14	2999	2699.1	2700	Pass	
19	15	3000	2700	4500	Pass	Boundary case
20	16	3001	2700.9	4500	Pass	
21	17	5000	4500	9000	Pass	
22	18	10000	9000	2970000	Pass	
23	19	2147483647	858993459.8	858994359.8	Pass	Maximum intvalue
24	20	2147483648	Invalid	Invalid	Pass	Overflow
25						
26	Name- Kajal Yadav		URN-2203585			

PRACTICAL- 5

Aim :- To perform Mutation Analysis of the Program Segments along with mutant history, mutation score and type of mutation by using any Code analysis Tool / Mutation Testing Tool.

1. **Statement Mutation :-**
   a. **Write a program for the greatest of two Numbers.**

```
#include <iostream>
using namespace std;

int main() {
   int y, X;
   cout << "Enter two numbers: ";
   cin >> y >> X;

   if (y > X) {
      cout << "The greatest number is: " << y << endl;
   } else {
      cout << "The greatest number is: " << X << endl;
   }

   return 0;
}
```

   b. **Write its Test Cases.**

	A	B	C	D	E	F	G	H
1	Test Case	Input (y,X)	Expected Output	Mutant 1	Mutant 2	Mutant 3	Mutant 4	Mutant 5
2	1	10,5	The greatest number is: 10	Pass	Fail	Pass	Pass	Pass
3	2	5,10	The greatest number is: 10	Fail	Pass	Pass	Pass	Fail
4	3	10,10	The greatest number is: 10	Pass	Fail	Fail	Fail	Fail
5	4	-5,0	The greatest number is: 10	Pass	Fail	Pass	Pass	Fail
6	5	0,0	The greatest number is: 10	Pass	Fail	Fail	Pass	Fail
7								
8								
9	NAME- KAJAL YADAV							
10	URM- 2203585							

   c. **Create a statement Mutation (Test the mutant for all the test cases , if the mutant test case fails then kill the mutant.)**

Mutant 1: Changing the comparison operator from > to >= This mutant is killed because it fails Test Case 2.

Mutant 2: Changing the comparison operator from > to < This mutant is killed because it fails multiple test cases (Test Cases 1, 3, 4, and 5).

Mutant 3: Changing the comparison operator from > to != This mutant is killed because it fails Test Case 3 and Test Case 5.

Mutant 4: Changing the if statement to an if-else statement This mutant is killed because it fails Test Case 3.

Mutant 5: Removing the else block This mutant is killed because it fails multiple test cases (Test Cases 2, 3, 4, and 5).

In the Excel sheet, the "Pass" and "Fail" values indicate whether the mutant passed or failed the corresponding test case. The mutants that fail any of the test cases are considered "killed".

2. **Value Mutation :-**

   a. **WAP to display all the numbers in an array greater than 10:**

```
#include <iostream>
using namespace std;

int main() {
    int arr[] = {5, 15, 8, 20, 12, 7, 18};
    int size = sizeof(arr) / sizeof(arr[0]);

    cout << "Numbers greater than 10 in the array are: ";
    for (int i = 0; i < size; i++) {
        if (arr[i] > 10) {
            cout << arr[i] << " ";
        }
    }
    cout << endl;

    return 0;
}
```

   b. **Write its Test Cases for Code.**

	A	B	C	D	E
1	Test Case	Input (arr)	Expected Output	Original Code	Mutant
2	1	{5, 15, 8, 20, 12, 7, 18}	15, 20,12,18	Pass	Fail
3	2	{10, 10, 10, 10, 10}	(no output)	Pass	Pass
4	3	{20, 30, 40, 50, 60}	20, 30, 40, 50, 60	Pass	Pass
5	4	{-5, -10,-15, -20}	(no output)	Pass	Pass
6	5	{0, 0, 0, 0, 0}	(no output)	Pass	Pass
7					
8	NAME - KAJAL YADAV				
9	URN - 2203585				

   c. **Create a mutant (change 10 to 20) Fail test case will be Greater than 10 (kill the mutant) if Test Case pass then code is correct.**

```
#include <iostream>
using namespace std;
```

```
int main() {
    int arr[] = {5, 15, 8, 20, 12, 7, 18};
    int size = sizeof(arr) / sizeof(arr[0]);

    cout << "Numbers greater than 20 in the array are: ";
    for (int i = 0; i < size; i++) {
        if (arr[i] > 20) {
            cout << arr[i] << " ";
        }
    }
    cout << endl;

    return 0;
}
```

Test cases-
Test Case 1: Fail (should be "15, 20, 12, 18")
Test Case 2: Pass
Test Case 3: Pass
Test Case 4: Pass
Test Case 5: Pass

This mutant is killed because it fails Test Case 1. The original code is correct because it passes all the test cases, and the mutant code is killed because it fails Test Case 1.

3. **Decision Mutation .**
    a. **Change the condition of 1st statement (greatest of two number) x>y = x<y in the same code.**

```
#include <iostream>
using namespace std;

int main() {
    int y, X;
    cout << "Enter two numbers: ";
    cin >> y >> X;

    if (y < X) {
        cout << "The greatest number is: " << X << endl;
    } else {
        cout << "The greatest number is: " << y << endl;
    }

    return 0;
}
```

- **Mutant 1:** Changing the comparison operator from < to >.

```
if (y > X) {
   cout << "The greatest number is: " << X << endl;
} else {
   cout << "The greatest number is: " << y << endl;
}
```

Test Cases:
Test Case 1: Fail (should be "The greatest number is: 10")
Test Case 2: Pass
Test Case 3: Fail (should be "The greatest number is: 10")
Test Case 4: Fail (should be "The greatest number is: 0")
Test Case 5: Fail (should be "The greatest number is: 0")

This mutant is killed because it fails multiple test cases.

- **Mutant 2:** Changing the comparison operator from < to <=.

```
if (y <= X) {
   cout << "The greatest number is: " << X << endl;
} else {
   cout << "The greatest number is: " << y << endl;
}
```

Test Cases:
Test Case 1: Fail (should be "The greatest number is: 10")
Test Case 2: Pass
Test Case 3: Pass
Test Case 4: Pass
Test Case 5: Pass

This mutant is killed because it fails Test Case 1.

- **Mutant 3:** Changing the comparison operator from < to !=.

```
if (y != X) {
   cout << "The greatest number is: " << X << endl;
} else {
   cout << "The greatest number is: " << y << endl;
}
```

Test Cases:
Test Case 1: Pass
Test Case 2: Pass
Test Case 3: Fail (should be "The greatest number is: 10")
Test Case 4: Pass
Test Case 5: Fail (should be "The greatest number is: 0")

This mutant is killed because it fails Test Case 3 and Test Case 5.

- **Mutant 4:** Changing the if statement to an if-else statement.

```
if (y < X) {
   cout << "The greatest number is: " << X << endl;
}
else {
   cout << "The greatest number is: " << y << endl;
}
```

Test Cases:
Test Case 1: Pass
Test Case 2: Pass
Test Case 3: Fail (should be "The greatest number is: 10")
Test Case 4: Pass
Test Case 5: Pass

This mutant is killed because it fails Test Case 3.

- **Mutant 5:** Removing the else block.

```
if (y < X) {
   cout << "The greatest number is: " << X << endl;
}
```

Test Cases:
Test Case 1: Fail (should be "The greatest number is: 10")
Test Case 2: Pass
Test Case 3: Fail (should be "The greatest number is: 10")
Test Case 4: Fail (should be "The greatest number is: 0")
Test Case 5: Fail (should be "The greatest number is: 0")

This mutant is killed because it fails multiple test cases.

Mutant Analysis:

Kill Mutant = Kill Mutant / Total Number of mutant * 100

Total number of mutants: 5
Number of killed mutants: 5
Percentage of killed mutants: 100%
Percentage of bug-free code: 100%

The original code is 100% bug-free, as all the mutants were successfully killed.

PRACTICAL-6

Aim :- To perform Black-Box Testing for all the units contained in the architectural segments using Equivalence Partitioning, Boundary Value Analysis and Orthogonal Array testing methods. Study exploratory Testing for the Module under Test and merits/demerits of this technique.

Program :

```
#include <iostream>
#include <cmath>
using namespace std;

double calculate_telephone_bill(int num_calls) {
   double bill = 0;
   if (num_calls <= 120) {
      bill = 300;
   } else {
      bill = 300; // Minimum bill
      if (num_calls <= 190) {
         bill += (num_calls - 120) * 1;
      } else {
         bill += 70 * 1; // For first 70 calls after 120
         if (num_calls <= 240) {
            bill += (num_calls - 190) * 0.8;
         } else {
            bill += 50 * 0.8; // For first 50 calls after 190
            bill += (num_calls - 240) * 0.4;
         }
      }
   }
   return bill;
}
```

1. **Equivalence Partitioning:**

   Partitions based on the number of calls:

   0 <= num_calls <= 120
   121 <= num_calls <= 190
   191 <= num_calls <= 240
   num_calls > 240

Test cases :-

	A	B	C	D	E
1					
2					
3					
4	Partition	Test Case	num_calls	Expected Bill	
5	0 <= num_calls <= 120	1	0	300	
6		2	60	300	
7		3	120	300	
8	121 <= num_calls <= 190	4	121	301	
9		5	150	330	
10		6	190	370	
11	191 <= num_calls <= 240	7	191	371	
12		8	220	407	
13		9	240	432	
14	num_calls > 240	10	241	432.4	
15		11	300	492	
16		12	1000	1132	
17					
18					
19					
20	Name - Kajal Yadav				
21	URN - 2203585				
22					

2. Boundary Value Analysis:-

	A	B	C	D
1	Test ID	Num_Call	Expected Call	Boundary Discription
2	1	0	300	Boundary of Partition 1 (0 <= num_calls <= 120)
3	2	120	300	Boundary of Partition 1 and 2 (120 and 121)
4	3	121	301	Boundary of Partition 2 and 1 (121 and 120)
5	4	190	370	Boundary of Partition 2 and 3 (190 and 191)
6	5	191	371	Boundary of Partition 3 and 2 (191 and 190)
7	6	240	432	Boundary of Partition 3 and 4 (240 and 241)
8	7	241	432.4	Boundary of Partition 4 and 3 (241 and 240)
9				
10				
11	Name- Kajal Yadav			
12	URN - 2203585			
13				

3. **Orthogonal Array Testing:**

   Factors:
   Number of calls (4 levels: 0-120, 121-190, 191-240, >240)

	A	B	C	D
1	Test Case	num_calls	Expected Bill	Level (Number of Calls)
2	1	0	300	0-120
3	2	121	301	121-190
4	3	191	371	191-240
5	4	241	432.4	>240
6				
7	Name- Kajal Yadav			
8	Urn- 2203585			
9				

4. **Exploratory Testing:**
   Exploratory testing involves simultaneous learning, test design, and test execution. It is a hands-on approach where the tester dynamically designs and executes tests based on the current state of the software and the information gained during testing.
   For the calculate_telephone_bill function, exploratory testing could involve:
   - Testing with various input values, including boundary cases, extreme values, and edge cases.
   - Checking the function's behavior when invalid or unexpected inputs are provided.
   - Verifying the correctness of the calculated bill for different ranges of input values.
   - Exploring the interaction between the function and other parts of the system, if applicable.
5. **Merits of Exploratory Testing:**
   - Flexibility: Exploratory testing allows testers to adapt and change testing strategies on the fly based on the information gathered during testing.
   - Rapid feedback: Testers can quickly identify and investigate issues as they are encountered.
   - Efficient use of resources: Exploratory testing can be performed with minimal preparation and planning, making it a cost-effective approach.

6. Demerits of Exploratory Testing:
   - Lack of documentation: Exploratory testing relies heavily on the tester's experience and skills, making it challenging to document and reproduce test cases.

- Limited test coverage: Since exploratory testing is unscripted, it may not provide comprehensive test coverage, leaving some areas of the software untested.
- Skill dependency: Exploratory testing requires highly skilled and experienced testers, making it difficult to scale or train new testers.

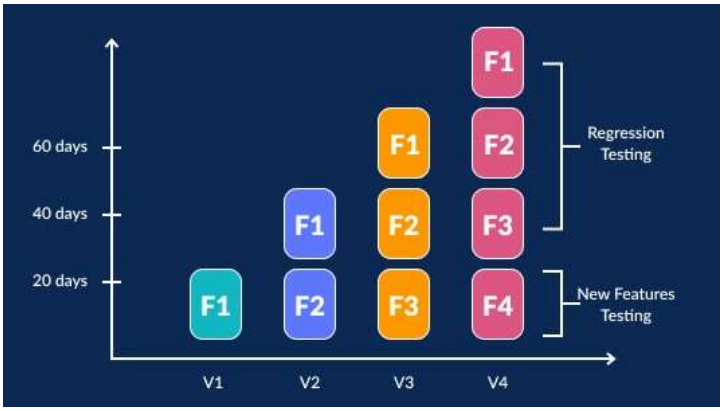
PRACTICAL - 7

Aim :- To perform Regression Testing of the System under construction with Unit and Integration profiles by using any Functional Testing Tool.

What is Regression testing used for?

- Before getting into the details of what it is, it's important to understand why we need it.
- When software developers fix a bug, add new functionality, or modify an existing feature or functionality, they must change the program code. Even a slight change will likely result in a plethora of new bugs. In such a scenario, a Test Engineer can reveal and pinpoint undesirable side effects through Regression tests. A properly executed regression test suite is vital. It is imperative that after a bug fix, the original product does not stop working.

The below graph depicts the importance of the Regression test:



When to perform Regression testing?

- When new features or enhancements are deployed to an existing codebase or application, Regression testing is required. It ensures that any new functionality or update to an existing application works properly without any bugs or defects.

  Developers and testers often struggle to trace every code thread, with significant chances of code incompatibility issues. As a result, executing Regression tests of their codebase (or application) allows them to detect defects earlier and ship applications with fewer risks.
- It can be used when a deployment takes longer than expected. In this case, the tester should run Regression tests daily. Also, it is better to run Regression tests after functional testing for weekly releases.
- When some functionality is overhauled, Regression test becomes even more critical as it may risk the codebase's present functionality. Furthermore, repairing one defect can sometimes lead to another. In this case, you can use a blend of debugging and Regression tests to ensure that everything works as intended.
- Want to make your life of testing easy and fuss-free while debugging? Try LT Debug Chrome extension!

Types of Regression testing

Depending on your Software Development Life Cycle (SDLC) and the new feature or update you aim to deploy, you can implement various types of regression tests. However, it is essential to understand the several regression tests types to choose the right one.

Below are the different types of regression testing –

- Corrective Regression Testing

Corrective Regression testing is one of the simpler forms of Regression tests requiring minimal effort. Corrective Regression testing involves no changes to the existing codebase and adding new functionality to the application. You simply need to test the existing functionality and the test cases that go with it rather than creating new ones.

- Unit Regression Testing

Unit Regression testing is an integral part of Regression tests in which the code is tested in isolation. All other interactions, integration, and dependencies are disabled while performing unit Regression testing, and the emphasis is on single unit code. Typically, this testing is done during low traffic and off-peak hours.

- Selective Regression Testing

Selective Regression testing analyzes the impact of existing code and the effect of both new and existing code. Common elements like variables and functions are incorporated into the application to identify quick results without affecting the process.

- Progressive Regression Testing

Test cases are created based on the requirements of a progressive regression test. When there are only minor product improvements, the new test cases are designed without affecting the existing code of a product.

- Complete Regression Testing

Some minor or significant changes might have a massive impact on the product. Complete Regression testing is used in this instance when there are significant modifications to the current code. It aids in the repair of any modifications made during the testing process.

- Partial Regression Testing

When new code is added to an existing codebase, partial Regression testing is conducted. This aids in the discovery of critical bugs in existing code and allows them to be tested without affecting the system.

- Retest-all Regression Testing

Re-test all Regression testing is the process of re-executing all the test cases to ensure that there are no bugs due to code changes in an application. This type of testing needs immense effort from the QA front.

Differences between Re-testing and Regression testing.

- Re-testing is the continuous process of testing specific test cases to ensure that the bugs are fixed and the web product's functionality works fine in the final execution. The same set of unit tests is repeated in re-testing to verify the code's functionality. In other words, re-testing is executing the same manual or automated tests to validate that the new build is working flawlessly.
    - Regression testing is a technique to verify the new build whenever any code commit takes place. In this process, a tester's job is to verify that no new bugs are included in the code due to software modification and adjustments. Once the Regression test suite is developed, you can automate it using an automation testing tool. However, this does not apply to re-testing.