Experiment No: 8

Aim: To design test cases for white box testing. (Basisi path testing)

Objective:

After implementing this experiment you will be able to:

- Understand different types of testing to uncovered errors ensuring bug free software for release.
- Create difference test cases.

Outcome:

- Able to test software for identification and removal of errors and noise present in coding and operation of software application.
- Ability to generate test cases.

Software Used: MS-Word, Notepad

Theory:

Software Testing

Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Software Testing is the process of executing a program or system with the intent of finding errors.

```
PROCEDURE average;

    *  This procedure computes the average of 100 or fewer
       numbers that lie between bounding values; it also computes the
       sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
       minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
    i = 1;
    total.input = total.valid = 0;
    sum = 0;
    DO WHILE value[i] <> -999 AND total.input < 100
       increment total.input by 1;
       IF value[i] > = minimum AND value[i] < = maximum
          THEN increment total.valid by 1;
                  sum = s sum + value[i]
          ELSE skip
       ENDIF
       increment i by 1;
    ENDDO
    IF total.valid > 0
       THEN average = sum / total.valid;
       ELSE average = -999;
    ENDIF
END average
```

Software testing can also be stated as the process of validating and verifying that a software program/application/product:

- meets the business and technical requirements that guided its design and development;
- works as expected; and
- Can be implemented with the same characteristics.

Software testing, depending on the testing method employed, can be implemented at any time in the development process. However, most of the test effort occurs after the requirements have been defined and the coding process has been completed. As such, the methodology of the test is governed by the software development methodology adopted.

Testing Objectives

Testing Objectives are

1. To find Uncovered Errors based on Requirement.

2. Ensure the Product is Bug Free before shipment/release.

3. 'Quality is ensured'

Testing Principles

- A necessary part of a test case is a definition of the expected output or result.
- A programmer should avoid attempting to test his or her own program.
- A programming organization should not test its own programs.
- Thoroughly inspect the results of each test.
- Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.
- Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.
- Avoid throwaway test cases unless the program is truly a throwaway program.
- Do not plan a testing effort under the tacit assumption that no errors will be found.
- The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
- Software Testing is an extremely creative and intellectually challenging task.

Types of Testing

1. **Regression Testing**

Regression testing focuses on finding defects after a major code change has occurred. Specifically, it seeks to uncover software regressions, or old bugs that have come back. Such regressions occur whenever software functionality that was previously working correctly stops working as intended.

2. **Alpha testing**

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing

3. **Beta testing**

Beta testing comes after alpha testing and can be considered a form of external user acceptance testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team. The software is released to groups

of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users

4. **Unit Testing**

Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.

White Box Testing

White-box testing is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality. In white-box testing an internal perspective of the system, as well as programming skills, are required and used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs

While white-box testing can be applied at the unit, integration and system levels of the software testing process, it is usually done at the unit level.

It can test paths within a unit, paths between units during integration, and between subsystems during a system level test.

Though this method of test design can uncover many errors or problems, it might not detect unimplemented parts of the specification or missing requirements.

White-box test design techniques include:

* Control flow testing

* Data flow testing

* Branch testing

* Path testing

Why we do White Box Testing?

To ensure:

* That all independent paths within a module have been exercised at least once.

* All logical decisions verified on their true and false values.

* All loops executed at their boundaries and within their operational bounds internal data structures validity.

Need of the White Box Testing ?

To discover the following types of bugs:

* Logical error tend to creep into our work when we design and implement functions, conditions or controls that are out of the program

* The design errors due to difference between logical flow of the program and the actual implementation

* Typographical errors and syntax checking

Black Box Testing

Black-box testing is a method of software testing that tests the functionality of an application as opposed to its internal structures or workings (see white-box testing). Specific knowledge of the application's code/internal structure and programming knowledge in general is not required.

Main focus in black box testing is on functionality of the system as a whole. The term 'behavioral testing' is also used for black box testing and white box testing is also sometimes called 'structural testing'. Behavioral test design is slightly different from black-box test design because the use of internal knowledge isn't strictly forbidden, but it's still discouraged.

Each testing method has its own advantages and disadvantages. There are some bugs that cannot be found using only black box or only white box. Majority of the application are tested by black box testing method. We need to cover majority of test cases so that most of the bugs will get discovered by black box testing.

Black box testing occurs throughout the software development and Testing life cycle i.e in Unit, Integration, System, Acceptance and regression testing stages. Test cases are built around specifications and requirements, i.e., what the application is supposed to do. It uses external descriptions of the software, including specifications, requirements, and designs to derive test cases.

These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid inputs and determines the correct output. There is no knowledge of the test object's internal structure.

This method of test can be applied to all levels of software testing: unit, integration, functional, system and acceptance. It typically comprises most if not all testing at higher

levels, but can also dominate unit testing as well.

Advantages of Black Box Testing

I. Tester can be non-technical.

II. Used to verify contradictions in actual system and the specifications.

III. Test cases can be designed as soon as the functional specifications are complete

Disadvantages of Black Box Testing

I. The test inputs needs to be from large sample space.

II. It is difficult to identify all possible inputs in limited testing time. So writing test cases is slow and difficult

III. Chances of having unidentified paths during this testing

5. Integration Testing

Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design. Software components may be integrated in an iterative way or all together.

Test case design for online book store

1: REGISTRATION: In online Book store, for registration the user has to enter the name, emailed, password, contact_no, and address and credit card details. For testing we check whether the user enters only letters and not numbers. For Email-id the testing is done whether it contains @ and dot (.).If not found then error is generated. For Contact_no the number should start from 9/8/7 as they are only valid nos. In this way testing at registration is done.

2: LOGIN: At login stage the user enters the name and password. If the user is registered then only he/she can buy book otherwise the user has to register.

2.1: If the user is registered and enters proper username and password the user can login successfully

2.2: if any one of detail is incorrect then error is generated.

2.3: if user name and password is incorrect then error is generated.

3: BOOK SEARCH: The customer search for the book, by author name, type of book. If book found the n he/she can buy book.

4: BUY BOOK: Once the book is found the user can buy book. For buying book the user has to register him if not registered. Then for carrying transaction the amount in credit card is checked. If the amount in credit card is less then user can't buy book. If amount in credit card is more than according to ACID property transaction is carried out. The credit card amount is also checked before performing any transaction. If the credit card number is wrong than no transaction is carried.

e.g.

1. **Using the design or code as a foundation, draw a corresponding flow graph.** A flow graph is created using the symbols and construction rules pre-

2. **Determine the cyclomatic complexity of the resultant flow graph.** The cyclomatic complexity, $V(G)$, is determined by applying the algorithms described in Section 17.5.2. It should be noted that $V(G)$ can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average*, compound conditions count as two) and adding 1. Referring to Figure 17.5,

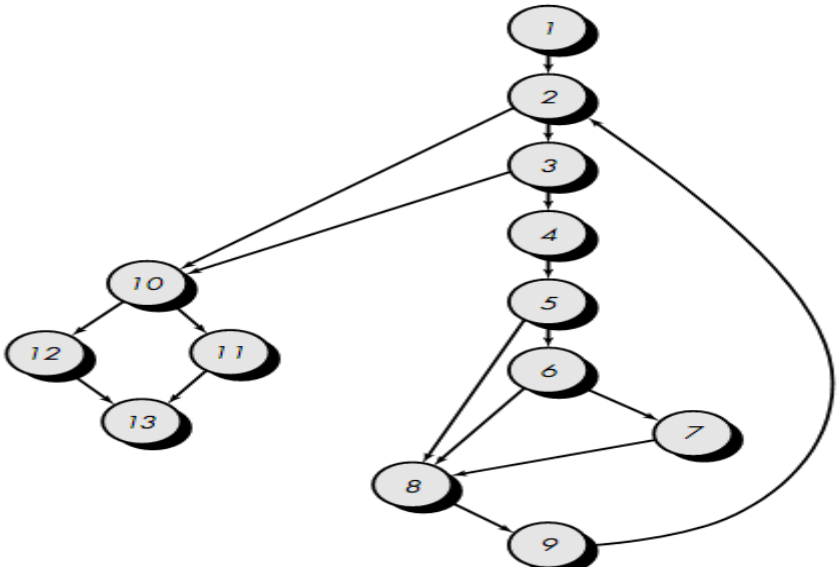
   $V(G)$ = 6 regions
   $V(G)$ = 17 edges − 13 nodes + 2 = 6
   $V(G)$ = 5 predicate nodes + 1 = 6

3. **Determine a basis set of linearly independent paths.** The value of $V(G)$ provides the number of linearly independent paths through the program control structure. In the case of procedure *average*, we expect to specify six paths:

   path 1: 1-2-10-11-13
   path 2: 1-2-10-12-13
   path 3: 1-2-3-10-11-13
   path 4: 1-2-3-4-5-8-9-2-. . .
   path 5: 1-2-3-4-5-6-8-9-2-. . .
   path 6: 1-2-3-4-5-6-7-8-9-2-. . .

   The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.

4. **Prepare test cases that will force execution of each path in the basis set.** Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Test cases that satisfy the basis set just described are



Path 1 test case:

value(*k*) = valid input, where $k < i$ for $2 \le i \le 100$

value(*i*) = −999 where $2 \le i \le 100$

Expected results: Correct average based on *k* values and proper totals.

Note: Path 1 cannot be tested stand-alone but must be tested as part of path 4, 5, and 6 tests.

Path 2 test case:

value(1) = −999

Expected results: Average = −999; other totals at initial values.

Path 3 test case:

Attempt to process 101 or more values.

First 100 values should be valid.

Expected results: Same as test case 1.

Path 4 test case:

value(*i*) = valid input where i < 100

value(*k*) < minimum where $k < i$

Expected results: Correct average based on *k* values and proper totals.

```
PROCEDURE average;

    *  This procedure computes the average of 100 or fewer
       numbers that lie between bounding values; it also computes the
       sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
       minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
    i = 1;
    total.input = total.valid = 0;
    sum = 0;
    DO WHILE value[i] <> -999 AND total.input < 100
       increment total.input by 1;
       IF value[i] > = minimum AND value[i] < = maximum
          THEN increment total.valid by 1;
                  sum = s sum + value[i]
          ELSE skip
       ENDIF
       increment i by 1;
    ENDDO
    IF total.valid > 0
       THEN average = sum / total.valid;
       ELSE average = -999;
    ENDIF
END average
```

Conclusion: Thus we have successfully test driven software development and applied the principles on test cases (write what you learnt).

Experiment No.09

Aim: To prepare Risk Mitigation, Monitoring and Management Plan (RMMM)