Software Testing Methodologies & Tools

Course Instructor & Designation: P. Naveen Sundar Kumar, Asst.Prof

Subject Title: Software Testing Methodologies and Tools

Year, Semester and Branch: III-II

Academic Year: 2016-2017

Course Objectives:
1. Understand the basic concepts of software testing.
2. Understand the various techniques and strategies of software testing and inspection and pointing out the importance of testing in achieving high-quality software.
3. Perform effective and efficient structural testing of software.
4. Integrate and test the various units and components of a software system.
5. Perform effective and efficient functional testing of software.
6. Select the appropriate tests to regression test your software after changes have been made.
7. Plan, track and control the software testing effort.
8. Understand the need of automated testing tools and various kinds of automated testing tools.

Prescribed Text Book:
2. Software Testing Principles and Practices by NARESH CHAUHAN, OXFORD HIGHER EDUCATION, university Press

Other reference books:
2. Effective methods of Software Testing, Perry, John Wiley.
Unit – I

- **Objective of the Unit (Two to Three lines):** Students have to learn what is the necessity of Software Testing, and its history, definitions of Software Testing, ancient times what the people mindset changed to importance of testing along with development, and performing validation verification activities.

- **Topic: Introduction to Software Testing**
- **Motivation (Why this topic is significant for the discussion?):** students have to get the awareness on software testing and fill the gap between Industries and Academia.

**Notes:** Software Testing Definition: IEEE definition states “Testing is the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements”.

- Another definition of Software testing
- Software Testing is the process used to verify the correctness and completeness of developed software applications.
- Software has pervaded our society, from modern households to spacecrafts. However Software Development faces many challenges. Software is becoming complex, but the demand for quality in software products has increased. This rise in customer awareness for quality increases the workload and responsibility of the software development team. That is why software Testing has gained so much popularity in the last decade. Job trends were shifted from development to software Testing. Organizations have separated Testing groups with proper hierarchy. Software development is driven with testing outputs. If the testing team claims the presence of bugs in the software, then the development team cannot release product.
- Software testing is mature in industry but not in academia. Thus this gap must be bridged with separate courses on software quality and testing.
- Today, the ideas and techniques of software testing essential for software developers, testers, and students as well.
- Testing teams to have knowledge of testing tactics and procedures of how to design test cases. There should be measurement technique for testing process has also been developed i.e. Testing Maturity Model (TMM) for measuring the maturity status of a testing process.
• Now started focus on effective testing rather than exhaustive testing. The psychology of tester plays an important role in software testing. It matters whether one wants to show the absence of errors or their presence in the software.

• **Evolution of Software Testing**

In the early days of software development, software testing was considered only a debugging process for removing errors after the development of software. In 1978, G. J Myers realized the need to discuss the techniques of software testing in a separate subject. He wrote a book *the art of software testing* which is a classic work on software testing. By 1980, software professionals and organizations started emphasizing on quality. Organizations realized the importance of having quality assurance teams.

In the 1990s, testing tools finally came into their own. There was flood of various tools, which are absolutely vital to adequate testing of software systems.

Gelperin and Hetzel have characterized the growth of software testing with time. Based on this, divide the evolution of software testing into the six phases.

1. **Debugging-oriented phase( before 1957)**
2. **Demonstration-oriented phase (1957-78)**
5. **Prevention-oriented phase(1988-95)**

**1. Debugging-oriented phase( before 1957)**

This phase is the early period of testing. At that time, testing basics were unknown. Programs were written and then tested by the programmers until they were sure that all the bugs were removed. The term used for testing was checkout, focused on getting the system to run.

**2. Demonstration-oriented phase (1957-78)**

The term ‘debugging’ continued in this phase. However, in 1957, Charles Baker pointed out that the purpose of checkout is not only to run the software but also to demonstrate the correctness according to the mentioned requirements.
Thus, the scope of checkout of program increased from program runs to program correctness. Moreover the purpose of checkout was to show the absence of errors.

There was no stress on the test case design. In this phase, there was a misconception that the software could be tested exhaustively.


   This phase can be described as the revolutionary turning point in history of software testing. Myers changed the view of testing from ‘testing is to show the absence of errors’ to ‘testing is to find more and more errors.’ He separated debugging from testing and stressed on the valuable test cases.

   This phase has given more importance to effective testing in comparison to exhaustive testing. The importance of early testing was also realized in this phase.


   With the effect of early testing, it was realized that if the bugs were identified at an early stage of development, it was cheaper to debug them as compared to the bugs found in implementation or post-implementation phases.

   ○ This phase stress on the quality of software products

   In 1983, guidelines by the National Bureau of Standards were released to choose a set of verification and validation techniques and evaluate the software at each step of software development.

5. **Prevention-oriented phase (1988-95)**

   The evaluation model stressed on the concept of bug-prevention as compared to the earlier concept of bug-detection. With the idea of early detection of bugs in early phases, can prevent the bugs in implementation or further phases. The prevention model includes test planning, test analysis, test design activities playing a major role.


   In this phase, testing was established as a complete process rather than a single phase in the software development life cycle. The testing process started as soon as requirements for a project are specified and it runs parallel to SDLC. The emphasis in this phase is also on quantification of various parameters which decide the performance of a testing process.
Goals of software Testing

The goals of software testing may be classified into three major categories.

They are

1. Immediate goals
2. Long-term goals
3. Post-implementation goals

1. Short-term or immediate goals
These goals are immediate results after performing testing. These goals may be set in the individual phases of SDLC.

   a. Bug Discovery – the immediate goal of testing is to find errors at any stage of software development. More the bugs discovered at an early stage, better will be the success rate of software testing.

   b. Bug prevention - it is the consequent action of bug discovery. From the behavior and interpretation of bugs discovered, everyone in the software development team gets to learn how to code safely

2. Long-term goals
These goals affect the product quality in the long run, when one cycle of the SDLC is over.

   a. Quality- since software is also a product, its quality is primary from the user’s point of view. Thorough testing ensures superior quality. Hence the first goal of understanding and performing the testing process is to enhance the quality of the software product.
Though quality depends on various factors such as correctness, integrity, efficiency, reliability is the major factor to achieve quality. Reliability is the matter of confidence, confidence is such a way that software not fails, and this level of confidence increases with rigorous testing. The confidence in reliability, in turn, increases the quality.

b. Customer satisfaction- from the user’s perspective, the prime concern of testing is customer satisfaction only. If we want the customer to be satisfied with the software product, then testing should be complete and thorough. A complete testing process achieves reliability, reliability enhances the quality, and quality in turn, increases the customer satisfaction.

c. Risk management- Risk is the probability that undesirable events will occur in a system. These undesirable events will prevent the organization from successfully implementing its business initiatives. Thus, risk is basically concerned with the business perspective of an organization. Risk must be controlled to manage them with ease.

The purpose of software testing as a control to provide information to management so that they can better react to risk situations.

So that it is the tester’s responsibility to give information in advance to evaluate business risks.

3. Post-implementation goals – these goals are important after the product is released.

a. Reduced maintenance cost of any software product is not its physical cost, as the software does not wear out. The only maintenance cost in a software product is its failure to errors. Post-release errors are costlier to fix, as they are difficult to detect. Thus, if testing has been done rigorously and effectively, then the chances of failure are minimized and in turn, the maintenance cost is reduced.

b. Improved software testing process- a testing process for one project may not be successful and there may be scope for improvement. Therefore, the bug history and post-implementation results can be analyzed to find out snags in the present testing process, which can be rectified in future
projects. Thus, post-implementation goal is to improve the testing process for future projects.

**Software testing as a process**

*Testing Process runs parallel to software process*

<table>
<thead>
<tr>
<th><strong>Software Development process</strong></th>
<th><strong>Software testing</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement gathering</td>
<td>Test plan</td>
</tr>
<tr>
<td>Requirement specification</td>
<td>Test case Design</td>
</tr>
<tr>
<td>Design</td>
<td>Test execution</td>
</tr>
<tr>
<td>Coding</td>
<td></td>
</tr>
</tbody>
</table>

Software testing process must be planned, specified, designed, implemented, and quantified. Testing must be governed by the quality attributes of the software product.

Thus testing is a dual purpose process, as it is used to detect bugs as well as to establish confidence in the quality of software.

An organization, to ensure better quality software, must adopt a testing process and consider the given important points.

- Testing process should be organized such that there is enough time for important and critical features of the software.
- Testing techniques should be adopted such that these techniques detect maximum bugs.
- Quality factors should be quantified. In other words, the process should be driven by quantified quality goals.
- Testing procedures and steps must be defined and documented.
- There must be scope for continuous process improvement.
Software testing terminology

Definitions

1. **Failure** - when the software is tested, failure is the first term being used. It means the inability of a system or component to perform a required function according to its specification. In other words, when results or behavior of the system under test are different as compared to specified expectations, then failure exists.

2. **Fault/defect/bug** – failure is the term which is used to describe the problems in a system on the output side. Fault is a condition that in actual causes a system to produce failure. Failure is a synonymous with the words defect or bug. It can be said that failures are manifestations of bugs. One failure may be due to one or more bugs and one bug may cause one or more failures. Some bugs are hidden in the sense that these are not executed, so hidden bugs may not always produce failures. They may execute only in certain rare conditions.

3. **Error** - whenever a development team member makes a mistake in any phase of SDLC, errors is produced. It might be a typographical error, misleading of a specification, a misunderstanding of what a subroutine does, error is very general term used for human mistakes. Thus an error causes a bug in turn causes failures

   **Ex:** Module A()
   
   ```
   {  
       ....
       While ( a > n+1);  
       {  
           ....
           Printf(" the value of x is", x);  
       }
       ....
   }
   ```

   Reason for failure body of the while loop not being executed due to semicolon at while loop. This mistake is known as error.

4. **Test case** - it is a well documented procedure designed to test the functionality of a feature in the system. A test case has an identity and is associated with a program behavior. The primary purpose of the designing a test case is to find errors in the system. For designing test
case, it needs to provide a set of inputs and its corresponding expected outputs.

5. **Test case ID** - is the identification number given to each test case

6. **Purpose** - defines why the case is being designed

7. **Preconditions** - for running the inputs in a system can be defined, if required, in a test case.

8. **Inputs** - Actual inputs must be provided, instead of general inputs.
   Ex: if two integer numbers have to be provided as input, then specifically mention them as 23 and 56
   no failure.
   After execution, observed results are compared with expected outputs mentioned in the test case.

9. **Testware** - The documents created during testing activities are known as testware. Testware are the documents that a test engineer produces. It may include test plans, test specifications, test case design, test reports. Testware documents should also be managed and updated like a software product.

10. **Incident** - when a failure occurs, it may or may not be readily apparent to the user. An incident is the symptoms associated with a failure that alerts the user about the occurrence of a failure.

11. **Test oracle** - an oracle is the means to judge the success or failure of a test, i.e. to judge the correctness of the system for some test. The simplest oracle is comparing actual results with expected results by hand. This can be very time consuming, so automated oracles are sought.

   **Life Cycle of a Bug**

   If an error has been produced in the requirement specification phase and not detected in the same phase, then it results in a bug in the next phase, i.e. the design phase. In the design phase, a bug has come from the previous stage, but an error can also produce in this stage.
Here errors are there in each phase those are not validated then they can be traversed to next phase.

Life cycle of a bug can be classified into two phases

1. Bugs-In Phase
2. Bugs-Out Phase

1. **Bugs-In phase**
   In this Phase if developer done a mistake that is becoming a bug, this bug can be traversed from all the phases if verification is not done in early phases, these should be added to the current phase bugs. It becomes more difficult.
   So verification is needed at earlier phases.

2. **Bugs-Out Phase**
   This phase deals with due to bugs failure occurs how we are going to overcome
   Activities are taken place in this phase are
   a. **Bug classification**- classifying the bugs according to its nature, the classification may be critical or catastrophic or it may have no adverse affect.
      After classifying the bugs it will help us which should be solved first, which should be postponed.
   b. **Bug isolation**- locates the module where the bug is there.
      How it is possible means by Incidents observation back trace the design and reach the module. This is called bug isolation.
**States of a Bug**

Bugs have different states in its lifecycle

Those are

- **New** - if the bug is reported first time.
- **Open** - when the test lead approves the bug is genuine
- **Assign** - if development team verifies that the bug is valid
- **Deferred** - if the bug has no adverse effect, less priority, no time
- **Rejected** - if developer rejects a bug based on its validity
- **Test** - bug has been fixed by the development team and released to the testing team to test.
- **Verified/ fixed** - if the developer approves that the reported bug is fixed then the tester verify that weather the reported bug is fixed or not, so the state is verified.
- **Reopened** - if the bug still exists after fixing it, then the state is reopened.
- **Closed** - if the bug is completely eliminated, got confirmation from tester and other team members.
Why do bug occur?

There are few answers for this question such as

**To error is human**

Human nature is susceptibility to do mistakes, these mistakes become errors.

**Bugs in Earlier stages go undetected and propagate**

Mistakes done by the development team and current phase bugs, for this some of the reasons given

Miscommunication between user and Development team member while collecting the requirements.

If the requirements are continuously changing

Effect of changes in one module effect the other which are connected each other

Resource rescheduling, changes in hardware/software requirements, new development team member added or removed in the middle of the project, coding standards not followed, ineffective knowledge transfer, discarding a portion of the existing code.

Complexity in keeping track of all the bugs itself causes a bug.

**Bugs Affect Economics of Software Testing**

The software bugs impact is enormous.

There is no guarantee that all the bugs are eliminated after testing

If the bugs are not verified at earlier stages they can be traversed to next phases, if the bugs eliminated at earlier stage it cost very low, if not eliminated at each phase of SDLC then the cost become more

Sometimes this cost may be 100% also

**Bug classification based on criticality**

Bug classified based on its nature, severity, and its impact

This will help us prioritize the bugs

Divide the bugs based on their criticality as given
Criticality Bugs- this kind of bugs stops or hangs the normal functioning of a system, this is the worst situation, if use this software it become helpless

Major Bug- this kind of bug cannot stop the function but it causes fail that function.

Medium Bugs- if the outputs are not according to standards

Minor Bugs- these are mild kind which does not affect on expected behavior.

Bug classification based on SDLC

Bugs can be classified based on SDLC which are

Requirements and classification Bugs

Due to requirements incomplete, ambiguous, inconsistent, and all specification problems these are all requirements and specification bugs because while collecting the requirements due to miscommunication between user and development team member, if these problems not verified at correctly at this phase those should be propagate to next phases.

Design Bugs

Design bugs are due to design mistakes and previous phase bugs.

Control flow bugs

These are such as path not reachbel, some paths through the flow is missing.

Logic Bugs

Misunderstanding the semantics of the order in which the Boolean expression is evaluated, improper layout of cases, missing cases , improper combination of cases.

Processing bugs

Arithmetic error, incorrect conversion from one data representation to another, ignoring overflow, improper use of logical operator

Data flow bugs

Uninitialized data, initialized in wrong format, initialized but not used, data used without initialization, redefined without intermediate use
**Testing Principles**

Principles are the guidelines to the tester

These are

1. **Effective testing not exhaustive testing**

   Tester approach should be an effective testing not an exhaustive, considering only the domain wise in that domain all program logic and conditions covered or not

2. **Testing is not a single phase performed in SDLC**

   Testing is parallel to the development phases so it is not at single phase after coding.

3. **Destructive approach for constructive testing**

   Tester mind is always trying to find more and more bugs in the programs so this destructive approach will help through testing it become constructive.

4. **Early testing is the best policy**

   Verification needed at early phases of software development so that bugs in the previous phases may not be traversed to next phases. So that cost should not be wasted.

   So testing should be started as early as with the requirements phase.

5. **Probability of existence of an error in a section of a program is proportional to the number of errors already found in that section**

   if modules were interconnected with other modules then the tester first concentrate on the modules which having more no.of bugs first, after completely elimination of this module go for the module which consists less than first, last the tester has to concentrate on the module which having least no.of bugs.

6. **Testing strategy should start at the smallest module level and expand towards the whole program**

   Testing must start at the unit level first then go for integration then system levels.
7. Testing should also be performed by an independent testing team

Who were developers of program if we give to test then they feel that what ever code I was developed in my code I need to find a bugs no never like feeling from the developers, so effective testing is not done

Programmers always are constructive approach ,but testers needed destructive approach

Programmers’ having positive feeling towards what they developed but tester mind is always negative.

So there is need of separate Independent testing team is needed.

8. Everything must be recorded in software testing

Everything must be recorded means it should be helpful to junior testers when they were working with the similar kind of projects.

If similar kind of project come to test then it become easy to tester to classify which are high priority bugs and probability of which modules gets more no. of bugs

9. Invalid inputs and unexpected behavior have a high probability of finding an error

First test the functionality with the invalid inputs for those invalid inputs it has to give an unexpected behavior

10. Tester must participate in specification and design review

if the tester participated in specification and design review he can easily grasp where the effective testing is needed

Software testing life cycle

STLC is simply a testing phase in the SDLC development. Validation and Authentication is tried and tested in this phase. The only limitation of this cycle is that it is limited to respective individual phase and is carried out by a group of skilled testers and technology evangelists.
Software Testing Life Cycle (STLC)

Like SDLC, STLC has fixed phases which are mentioned in hierarchy below:

1. TEST PLANNING – Preparing the test strategy & planning
   Test planning consists of
   - Defining the test strategy
   - Estimate the no. of test cases, their duration, cost,
   - Plan the resources
   - Identify areas of risk
   - Defining the test completion criteria
   - Identification of methodologies, techniques, and tools to various test cases
   - Identifying reporting procedures, bug classification, data bases for testing
   **The major outputs of test planning**
   - Develop a test case format
   - Develop test case plans according to every phase of SDLC
   - Identify test cases to be automated
   - Prioritize the test cases according to their importance and criticality
   - Plan test cycles required for regression testing

2. TEST Design – it is well planned process
   It includes the some critical activities such as
   **Determining the test objectives and their prioritization**
   From the requirements specification and design documents identify the testing objectives. Depending on the scope and risk give prioritize the test objectives
   **Preparing list of items to be tested**
   Objects are converted into list of objects
   **Mapping items to test cases**
   Need to create a matrix for knowing which test case is will be covered by which item
   The matrix will help in
   i) Identify the major test scenario
   ii) Reducing the redundant test cases
   iii) Identifying the absence of a test case for particular objective and as a result, creating them
   The tester who designs the test cases must understand the cause-and-effect connections also.
Some attributes of good test cases are
i) Criticality and high risk priority are given highest priority
ii) Good test case is one which is capable of finding high probability of finding an error
iii) Test cases are not overlap or be redundant
iv) Good test case is one which always follow the modular approach
v) Test case always able to find an undercover errors

**Selection of test case design techniques**

Basically there are two testing techniques
Black-box and white box testing techniques
Black-box testing techniques – in this generate test cases without knowing the internal working of a system

**Creating test cases and test data**

test cases are created based on testing objecting, test data means input data given to the test cases.

**Setting up the test environment and supporting tools**

Environment includes h/w configurations, testers, tools, interfaces, manuals.
Tools like QTP, load runner

**Creating test procedure specification**

It is sequence of steps, this procedure used by the tester at the time of testing

**Software testing life cycle**

![Software Testing Life Cycle Diagram]
**Test Execution**

In this phase all test cases are executed and test results are documented in the test incident reports, test logs, testing status, and test summary reports.

**Testing levels versus responsibility**

<table>
<thead>
<tr>
<th>Test execution level</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit</td>
<td>Developers of the module</td>
</tr>
<tr>
<td>Integration</td>
<td>Testers and Developers</td>
</tr>
<tr>
<td>System</td>
<td>Testers. Developers, End-users</td>
</tr>
<tr>
<td>Acceptance</td>
<td>Testers, End-users</td>
</tr>
</tbody>
</table>

**Post-Execution /Test Review**

Bugs should be reported to the developers after test execution successful

**Understanding the bug** - after getting test execution report to the Developer has to understand its whereabouts.

**Reproducing the bug** - once again need to execute the test case with invalid inputs so that the bug should reproduce once again

**Analyzing the nature and cause of the bug** - after getting bug report based on incidents the bug nature is analyzed

It consists of two analyses

i) Reliability analysis - this analysis helps predefined reliability goals or not

ii) Coverage analysis - this is alternative criterion to stop testing

iii) Overall defect analysis - this analysis is identify risk areas and quality improvement
Verification and validation activities
To understand verification and validation first need to understand SDLC phases

SDLC Phases

Requirement gathering - Requirements are gathered from user in his own words there is no any technical terminology used.

Requirement specification or objective - this document should be understandable to all the stakeholders of the project. And preparing list of objectives from the reference materials like BRS, FRS.

Functional Design or High level design - translating user requirements into a set of external interfaces.

In this HLD is prepared with the help of Software requirement specification (SRS).

The HLD document contains
1. Overall architecture diagram
2. Functionalities of the overall system
3. List of modules
4. Brief functionality of each module
5. Interface relationships among modules

**Internal Design or Low level Design**

This is micro-level design document also called as internal design document or low level design document (LLD). In this each and every module will be described elaborately.

**Coding**

After preparing the LLD, it is easy to write code. Because design gives a way to write code. Once a coding is over then the validation starts.

**Verification**

Verification is a set of activities that ensure correct implementation of specific functions in software.

Why verification is needed?

If verification is not done at earlier stages the bugs should be traversed to the next phases and bug should become more and more, then it is too difficult to remove and cost become increases

And it also increases the quality

**Everything must be verified**

Here everything means all the SDLC Phases and products

**Result verification may not be binary**

Binary means may not accepting or rejecting there is deep procedure is there for whatever the result is.

**Even implicit qualities must be verified**

Not mentioned anywhere those qualities also be verified

**Verification activities**

There are four verification activities are there

Those are

Verification of requirements and objectives
Verification of high level design
Verification of low level design
Verification of coding

**Verification of requirement**

First the tester prepare acceptance criteria
In this he perform two parallel activities those are
1. For test acceptance criteria he checks for its completeness, clarity and testability
2. The tester prepares acceptance test plan

**Verification of objectives**

In this he perform two parallel activities those are
1. The tester verifies all the objectives mentioned in the SRS
2. He also performs the acceptance test plan

**Verification of requirements and objectives**

There are some points regarding verification requirements and objectives

1. **Correctness**
   
   To compare something there already exists the same type which is high standard in all aspects
   
   Testers interact with the original users once again to better understand
   
   He has to verify whether those are solvable in realistic environment

2. **Unambiguous**
   
   Unambiguous here is any requirement should not give too many meanings or interpretations.
   
   There is no redundancy in requirements
   
   Characteristic of each product is explained using unique term

3. **Consistent**
   
   Specifications should not be conflict with one another because conflicts produce bugs
   
   For consistency property checking here are some examples
   
   a) Real world objects conflict, for ex one specification requires mouse for input, another recommends joystick
   
   b) Logical conflict between two specification – one function requires the function to perform square root, while another performs same function to perform square operation
   
   c) Conflicts in terminology, one place the term process is used and at another place it has been termed as task or module.

4. **Completeness**
   
   The requirement must be complete because incomplete requirements give unsuccessful result
   
   Verify the given requirement for completeness
   
   a) Verify the significant requirements such as functionality, performance, design constraints, attribute or external interface
   
   b) Check response for every possible input
   
   c) Check the figures and tables have been labeled and referenced completely

5. **Update**
   
   Based on the time requirements also modified and updated
   
   That updated requirements must be verified such as
   
   a) If the requirements are new one then all the specified properties such as correctness, completeness, unambiguous, consistent, and their feasibility should also be verified.
   
   b) If any change is done that change will affect all which are connected with that requirements

6. **Traceability**
   
   Traceability should also be verified for every requirement there is any origin or not so it consists of two steps
Backward traceability- check that each requirement references its source in previous document
Forward traceability-check that each requirement has a unique name or reference number in all documents

**Verification of high-level design**
The tester is responsible for two parallel activities in this
1. The testers verify high level design in such way that all the components and its interfaces designed according to user specified documents or not.
2. Tester prepares a functional test plan

**How to verify high-level design**
To verify high-level design needs design in the following aspects

**Data design**
In this creates a data model it contains data and its algorithms how you manipulate the data is essential to create high-quality applications

**Architectural design**
It represents the structure of software components their properties and interactions

**Interface design**
Interface means external interfaces are there and internal interfaces so this verification is very crucial to the project, if verification is not done perfectly here overall system go to effect.

**Verification of low-level design**
It consist of verify SRS of each module
Verify the SRS of each SDD

**How to verify code**
Coding is the process of converting LLD specification into a specification Language.
In this how the code is verified, steps are given
Check that every design specification in HLD and LLD has been coded using traceability
Examine the code against a language specification checklist
Some points against the code can be verified are
a) Misunderstood or incorrect arithmetic precedence
b) Mixed mode operation
c) Incorrect initialization
d) Precision inaccuracy
e) Incorrect symbolic representation of an expression
f) Different data type
g) Improper or non-existing loop termination
h) Failure to exit

Two kinds of techniques are used to verify the code

**Static testing techniques**- without executing the code.
**Dynamic testing techniques**- it is done while testing the executing the code
**Unit verification**
Verification of code cannot be done for whole system at once, unit verification is done by the corresponding module developer. Here some of the points to consider while doing unit verification:
Interfaces are verified for information flows between the modules.
Local data structure need to verified
Boundary conditions are checked
Verification is done in such way that all statements in a module executed at least once
All error handling paths are tested
E software under consideration

**Validation**
Validation is a set of activities that ensure the software under consideration has been built right and is traceable to customer requirements.
Validation is performed after coding is over
To determine whether the product satisfies the user’s requirements, as stated in the requirement specification
Products actual behavior matches with the desired behavior
All the stages still coding are bug-free
Validation testing provides the last chance to discover bugs

**Validation activities**
Validation activities are divided into two parts
1) Validation test plan
2) Validation test execution

**Validation test plan**
To prepare validation test plan tester has to follow points such as:
- Testers must understand the current SDLC phase
- Testers understand the relevant documents in the corresponding SDLC phase.
- Based on this information tester prepares a test plan which are used at the time of validation testing.
- Tester prepared test plans and when it should be used

- **Acceptance test plan**- based on the user feedback acceptance criteria prepared in the requirement phase based on acceptance criteria acceptance test plan is prepared.
- This is helpful at the time of acceptance testing

- **System test plan**- to verify objectives specified in the SRS it refers how entire system behaves in different conditions.
- This plan is used at the time of system testing

- **Function test plan**- it is prepared in the HLD phase.
- In this test cases are designed to test every functionality. This plan used at the time of functional testing

- **Integration test plan**- this is for validate all integrated modules, this is also conforms to the whole system. This plan is used at the time of integration testing
**Unit test plan** - this is prepared in the LLD phase. In this, every functionality of each unit is tested thoroughly. This plan is used at the time of unit testing.

**Validation test execution**
Validation test execution can be divided into six testing activities
1) Unit validation testing
2) Integration testing
3) Functional testing
4) System testing
5) Acceptance testing
6) Installation testing

1) **Unit validation testing**
   - one unit or module is the smaller building block of the whole system, testing starts with unit validation first, not the whole system
   - A unit must be validated before integrate it
   - The motivation for unit validation is
     a) Since the developer has to concentrate on smaller modules or units, so testing should also start with modules
     b) If the whole system is tested, it is difficult to trace bugs
     c) Parallel testing is also possible

**Integration testing**
- it is nothing but a combination of units and testing multiple components together

**Functional testing**
- In this, all the functions and interfaces must also be tested
- The objective of functional testing is to measure the quality of the functional components of the system
- It validated as
  1. Performs in accordance to the specification
  2. Moves the data from one business event to the next
  3. Every function testing meets the business objective of the system

**System testing**
- Its primary purpose is to exercise the whole system such as recovery issues, how secure the whole system is

**Acceptance testing**
- In presence of user according to his requirements, its working or not is verified. Here, the development team and the testing team and the user are also involved in this testing

**Installation testing**
- It does not test the system, but it tests the process of making the system operational.
- It must require the steps to install software.
**Classroom Interaction (two to three questions):**
- What is the necessity of testing?
- Define Testing?
- What is exhaustive Testing
- What is the difference between verification and validation?

**Tutorial Questions:**
How can you fill the gap between Industries and academia?

**Assignment questions:**
- What is the Psychology the Tester should have?
- Explain Software Testing Lifecycle (STLC)?
- What are the states of bug?
- Explain about bug-in and Bug-out Phases?

**Exercise questions / Long answer questions / Project possibilities :**
Assume that if separate Testing team is not there what about the quality of Development?

**Online resources:**
- [www.csi-india.org](http://www.csi-india.org)
- [www.infibeam.com](http://www.infibeam.com)
- [www.manit.ac.in](http://www.manit.ac.in)
Unit – II

- **Objective of the Unit (Two to Three lines):**
  - Students know what the purpose of Testing
  - Understand the differences between Tester and debugger, functional versus structure, builder versus buyer etc. understand the levels of testing and know the classification of bugs.

- **Topic: Purpose of Testing**

  - **Motivation (Why this topic is significant for the discussion?):**
    - In the early period of 1970 the people thought that both the Testing and debugging are same so first know what are the different kinds bugs, these bugs are not possible to reduce with the help of debugging.

  **Notes:** Testing consumes at least half of the time and work required to produce a functional program.

  MYTH: Good programmers write code without bugs. (It’s wrong!!!)

  History says that even well written programs still have 1-3 bugs per hundred statements.

  **Productivity and Quality in software:**
  - In production of consumer goods and other products, every manufacturing stage is subjected to quality control and testing from component to final stage.
  - If flaws are discovered at any stage, the product is either discarded or cycled back for rework and correction.
  - Productivity is measured by the sum of the costs of the material, the rework, and the discarded components, and the cost of quality assurance and testing.
  - There is a tradeoff between quality assurance costs and manufacturing costs: If sufficient time is not spent in quality assurance, the reject rate will be high and so will be the net cost. If inspection is good and all errors are caught as they occur, inspection costs will dominate, and again the net cost will suffer.
  - Testing and Quality assurance costs for 'manufactured' items can be as low as 2% in consumer products or as high as 80% in products such as space-ships, nuclear reactors, and aircrafts, where failures threaten life. Whereas the manufacturing cost of software is trivial.
  - The biggest part of software cost is the cost of bugs: the cost of detecting them, the cost of correcting them, the cost of designing tests that discover them, and the cost of running those tests.
• For software, quality and productivity are indistinguishable because the cost of a software copy is trivial.

• Phases in a tester's mental life can be categorised into the following 5 phases:
  1. **Phase 0:** There is no difference between testing and debugging. Phase 0 thinking was the norm in early days of software development till testing emerged as a discipline.
  2. **Phase 1:** the purpose of testing here is to show that software works. Highlighted during the late 1970s. This failed because the probability of showing that software works 'decreases' as testing increases. i.e. The more you test, the more likely you’ll find a bug.
  3. **Phase 2:** the purpose of testing is to show that software doesn’t work. This also failed because the software will never get released as you will find one bug or the other. Also, a bug corrected may also lead to another bug.
  4. **Phase 3:** the purpose of testing is not to prove anything but to reduce the perceived risk of not working to an acceptable value (Statistical Quality Control). Notion is that testing does improve the product to the extent that testing catches bugs and to the extent that those bugs are fixed. The product is released when the confidence on that product is high enough. (Note: This is applied to large software products with millions of code and years of use.)
  5. **Phase 4:** Testability is the factor considered here. One reason is to reduce the labor of testing. Other reason is to check the testable and non-testable code. Testable code has fewer bugs than the code that's hard to test. Identifying the testing techniques to test the code is the main key here.

**Test Design:** We know that the software code must be designed and tested, but many appear to be unaware that tests themselves must be designed and tested. Tests should be properly designed and tested before applying it to the actual code.

*Testing isn’t everything:* There are approaches other than testing to create better software. Methods other than testing include:

**Inspection Methods:** Methods like walkthroughs, desk checking, formal inspections and code reading appear to be as effective as testing but the bugs caught do not completely overlap.
**Design Style:** While designing the software itself, adopting stylistic objectives such as testability, openness and clarity can do much to prevent bugs.

**Static Analysis Methods:** Includes formal analysis of source code during compilation. In earlier days, it is a routine job of the programmer to do that. Now, the compilers have taken over that job.

**Languages:** The source language can help reduce certain kinds of bugs. Programmers find new bugs while using new languages.

**Design Methodologies and Development Environment:** The development process and the environment in which that methodology is embedded can prevent many kinds of bugs.

7. **The pesticide paradox and the complexity barrier**
   Whatever methods we are using to remove completely bugs are not removing but methods become ineffectual due to subtler bugs. Complexity should also be manageable.

**DICHOTOMIES:**

- **Testing Versus Debugging:** Many people consider both as same. Purpose of testing is to show that a program has bugs. The purpose of testing is to find the error or misconception that led to the program's failure and to design and implement the program changes that correct the error.
- Debugging usually follows testing, but they differ as to goals, methods and most important psychology.

This table shows few important differences between testing and debugging.

<table>
<thead>
<tr>
<th>Testing</th>
<th>Debugging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing starts with known conditions, uses predefined procedures and has predictable outcomes.</td>
<td>Debugging starts from possibly unknown initial conditions and the end cannot be predicted except statistically.</td>
</tr>
<tr>
<td>Testing can and should be planned, designed and</td>
<td>Procedure and duration of debugging</td>
</tr>
<tr>
<td>Testing is a demonstration of error or apparent correctness.</td>
<td>Debugging is a deductive process.</td>
</tr>
<tr>
<td>-------------------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Testing proves a programmer’s failure.</td>
<td>Debugging is the programmer’s vindication (Justification).</td>
</tr>
<tr>
<td>Testing, as executes, should strive to be predictable, dull, constrained, rigid and inhuman.</td>
<td>Debugging demands intuitive leaps, experimentation and freedom.</td>
</tr>
<tr>
<td>Much testing can be done without design knowledge.</td>
<td>Debugging is impossible without detailed design knowledge.</td>
</tr>
<tr>
<td>Testing can often be done by an outsider.</td>
<td>Debugging must be done by an insider.</td>
</tr>
<tr>
<td>Much of test execution and design can be automated.</td>
<td>Automated debugging is still a dream.</td>
</tr>
</tbody>
</table>

- **Function versus Structure:** Tests can be designed from a functional or a structural point of view. In **functional testing**, the program or system is treated as a black-box. It is subjected to inputs, and its outputs are verified for conformance to specified behavior. Functional testing takes the user point of view - bother about functionality and features and not the program's implementation. **Structural testing** does look at the implementation details. Things such as programming style, control method, source language, database design, and coding details dominate structural testing.

- Both Structural and functional tests are useful, both have limitations, and both target different kinds of bugs. Functional tests can detect all bugs but would take infinite time to do so. Structural tests are inherently finite but cannot detect all errors even if completely executed.

- **Designer Versus Tester:** Test designer is the person who designs the tests where as the tester is the one actually tests the code. During functional testing, the designer and tester are probably different persons. During unit testing, the tester and the programmer merge into one person.
• Tests designed and executed by the software designers are by nature biased towards structural consideration and therefore suffer the limitations of structural testing.

• **Modularity versus Efficiency:** A module is a discrete, well-defined, small component of a system. Smaller the modules, difficult to integrate; larger the modules, difficult to understand. Both tests and systems can be modular. Testing can and should likewise be organised into modular components. Small, independent test cases can be designed to test independent modules.

• **Small versus Large:** Programming in large means constructing programs that consists of many components written by many different programmers. Programming in the small is what we do for ourselves in the privacy of our own offices. Qualitative and Quantitative changes occur with size and so must testing methods and quality criteria.

• **Builder versus Buyer:** Most software is written and used by the same organization. Unfortunately, this situation is dishonest because it clouds accountability. If there is no separation between builder and buyer, there can be no accountability.

• The different roles / users in a system include:
  1. **Builder:** Who designs the system and is accountable to the buyer.
  2. **Buyer:** Who pays for the system in the hope of profits from providing services?
  3. **User:** Ultimate beneficiary or victim of the system. The user's interests are also guarded by.
  4. **Tester:** Who is dedicated to the builder's destruction?
  5. **Operator:** Who has to live with the builders' mistakes, the buyers' murky (unclear) specifications, testers' oversights and the users' complaints?
MODEL FOR TESTING:

It includes three models: A model of the environment, a model of the program and a model of the expected bugs.

ENVIRONMENT:

A Program's environment is the hardware and software required to make it run. For online systems, the environment may include communication lines, other systems, terminals and operators.

The environment also includes all programs that interact with and are used to create the program under test - such as OS, linkage editor, loader, compiler, utility routines.

Because the hardware and firmware are stable, it is not smart to blame the environment for bugs.

PROGRAM:

Most programs are too complicated to understand in detail.

The concept of the program is to be simplified in order to test it.

If simple model of the program doesn't explain the unexpected behaviour, we may have to modify that model to
include more facts and details. And if that fails, we may have to modify the program.

**BUGS:**

Bugs are more insidious (deceiving but harmful) than ever we expect them to be.

An unexpected test result may lead us to change our notion of what a bug is and our model of bugs.

Some optimistic notions that many programmers or testers have about bugs are usually unable to test effectively and unable to justify the dirty tests most programs need.

**OPTIMISTIC NOTIONS ABOUT BUGS:**

1. **Benign Bug Hypothesis**: The belief that bugs are nice, tame and logical. (Benign: Not Dangerous)
2. **Bug Locality Hypothesis**: The belief that a bug discovered within a component affects only that component's behavior.
3. **Control Bug Dominance**: The belief that errors in the control structures (if, switch etc) of programs dominate the bugs.
4. **Code / Data Separation**: The belief that bugs respect the separation of code and data.
5. **Lingua Salvator Est**: The belief that the language syntax and semantics (e.g. Structured Coding, Strong typing, etc) eliminates most bugs.
6. **Corrections Abide**: The mistaken belief that a corrected bug remains corrected.
7. **Silver Bullets**: The mistaken belief that X (Language, Design method, representation, environment) grants immunity from bugs.
8. **Sadism Suffices**: The common belief (especially by independent tester) that a sadistic streak, low cunning, and intuition are sufficient to eliminate most bugs. Tough bugs need methodology and techniques.
9. **Angelic Testers**: The belief that testers are better at test design than programmers is at code design.
TESTS

Tests are formal procedures, Inputs must be prepared, Outcomes should predict, tests should be documented, commands need to be executed, and results are to be observed. All these errors are subjected to error.

We do three distinct kinds of testing on a typical software system. They are:

Unit / Component Testing: A Unit is the smallest testable piece of software that can be compiled, assembled, linked, loaded etc. A unit is usually the work of one programmer and consists of several hundred or fewer lines of code. Unit Testing is the testing we do to show that the unit does not satisfy its functional specification or that its implementation structure does not match the intended design structure. A Component is an integrated aggregate of one or more units. Component Testing is the testing we do to show that the component does not satisfy its functional specification or that its implementation structure does not match the intended design structure.

Integration Testing: Integration is the process by which components are aggregated to create larger components. Integration Testing is testing done to show that even though the components were individually satisfactory (after passing component testing), checks the combination of components are incorrect or inconsistent.

System Testing: A System is a big component. System Testing is aimed at revealing bugs that cannot be attributed to components. It includes testing for performance, security, accountability, configuration sensitivity, startup and recovery.

CONSEQUENCES OF BUGS:

IMPORTANCE OF BUGS: The importance of bugs depends on frequency, correction cost, installation cost, and consequences.

1. Frequency: How often does that kind of bug occur? Pay more attention to the more frequent bug types.
2. Correction Cost: What does it cost to correct the bug after it is found? The cost is the sum of 2 factors: (1) the cost of
discovery (2) the cost of correction. These costs go up dramatically later in the development cycle when the bug is discovered. Correction cost also depends on system size.

3. **Installation Cost:** Installation cost depends on the number of installations: small for a single user program but more for distributed systems. Fixing one bug and distributing the fix could exceed the entire system’s development cost.

4. **Consequences:** What are the consequences of the bug? Bug consequences can range from mild to catastrophic.

A reasonable metric for bug importance is

\[
\text{Importance} = \text{\langle \$ \rangle} = \text{Frequency} \times (\text{Correction cost} + \text{Installation cost} + \text{Consequential cost})
\]

**CONSEQUENCES OF BUGS:** The consequences of a bug can be measure in terms of human rather than machine. Some consequences of a bug on a scale of one to ten are:

1. **Mild:** The symptoms of the bug offend us aesthetically (gently); a misspelled output or a misaligned printout.
2. **Moderate:** Outputs are misleading or redundant. The bug impacts the system’s performance.
3. **Annoying:** The system’s behavior because of the bug is dehumanizing. *E.g.* Names are truncated arbitrarily modified.
4. **Disturbing:** It refuses to handle legitimate (authorized / legal) transactions. The ATM won’t give you money. My credit card is declared invalid.
5. **Serious:** It loses track of its transactions. Not just the transaction itself but the fact that the transaction occurred. Accountability is lost.
6. **Very Serious:** The bug causes the system to do the wrong transactions. Instead of losing your paycheck, the system credits it to another account or converts deposits to withdrawals.
7. **Extreme:** The problems aren’t limited to a few users or to few transaction types. They are frequent and arbitrary instead of sporadic (infrequent) or for unusual cases.
8. **Intolerable:** Long term unrecoverable corruption of the database occurs and the corruption is not easily discovered. Serious consideration is given to shutting the system down.
9. **Catastrophic:** The decision to shut down is taken out of our hands because the system fails.

10. **Infectious:** What can be worse than a failed system? One that corrupt other systems even though it does not fall in itself; that erodes the social

    Physical environment; that melts nuclear reactors and starts war.

**FLEXIBLE SEVERITY RATHER THAN ABSOLUTES:**

Quality can be measured as a combination of factors, of which number of bugs and their severity is only one component.

Many organizations have designed and used satisfactory, quantitative, quality metrics.

Because bugs and their symptoms play a significant role in such metrics, as testing progresses, you see the quality rise to a reasonable value which is deemed to be safe to ship the product.

The factors involved in bug severity are:

- **Correction Cost:** Not so important because catastrophic bugs may be corrected easier and small bugs may take major time to debug.

- **Context and Application Dependency:** Severity depends on the context and the application in which it is used.

- **Creating Culture Dependency:** What is the important depends on the creators of software and their cultural aspirations. Test tool vendors are more sensitive about bugs in their software then games software vendors.

- **User Culture Dependency:** Severity also depends on user culture. Naive users of PC software go crazy over bugs where as pros (experts) may just ignore.

- **The software development phase:** Severity depends on development phase. Any bugs gets more severe as
it gets closer to field use and more severe the longer it has been around.

**TAXONOMY OF BUGS:**

- There is no universally correct way categorize bugs. The taxonomy is not rigid.
- A given bug can be put into one or another category depending on its history and the programmer’s state of mind.
- The major categories are: (1) Requirements, Features and Functionality Bugs (2) Structural Bugs (3) Data Bugs (4) Coding Bugs (5) Interface, Integration and System Bugs (6) Test and Test Design Bugs.

**REQUIREMENTS, FEATURES AND FUNCTIONALITY BUGS:**
Various categories in Requirements, Features and Functionality bugs include:

1. **Requirements and Specifications Bugs:**
   - Requirements and specifications developed from them can be incomplete ambiguous, or self-contradictory. They can be misunderstood or impossible to understand.
   - The specifications that don't have flaws in them
     - May change while the design is in progress.
     - The features are added, modified and deleted.
     - Requirements, especially, as expressed in specifications are a major source of expensive bugs.
     - The range is from a few percentages to more than 50%, depending on the application and environment.
     - What hurts most about the bugs is that they are the earliest to invade the system and the last to leave.

2. **Feature Bugs:**
   - Specification problems usually create corresponding feature problems.
   - A feature can be wrong, missing, or superfluous (serving no useful purpose). A missing feature or case is easier to detect
and correct. A wrong feature could have deep design implications.

- Removing the features might complicate the software, consume more resources, and foster more bugs.

3. **Feature Interaction Bugs:**

   - Providing correct, clear, implementable and testable feature specifications is not enough.
   - Features usually come in groups or related features. The features of each group and the interaction of features within the group are usually well tested.
   - The problem is unpredictable interactions between feature groups or even between individual features. For example, your telephone is provided with call holding and call forwarding. The interactions between these two features may have bugs.
   - Every application has its peculiar set of features and a much bigger set of unspecified feature interaction potentials and therefore result in feature interaction bugs.

**Specification and Feature Bug Remedies:**

- Most feature bugs are rooted in human to human communication problems. One solution is to use high-level, formal specification languages or systems.
- Such languages and systems provide short term support but in the long run, do not solve the problem.

**Short term Support:** Specification languages facilitate formalization of requirements and inconsistency and ambiguity analysis.

**Long term Support:** Assume that we have a great specification language and that can be used to create unambiguous, complete specifications with unambiguous complete tests and consistent test criteria.

The specification problem has been shifted to a higher level

- But not eliminated.
Testing Techniques for functional bugs: Most functional test techniques— that is those techniques which are based on a behavioral description of software, such as transaction flow testing, syntax testing, domain testing, logic testing and state testing are useful in testing functional bugs.

STRUCTURAL BUGS: Various categories in Structural bugs include:

Control and Sequence Bugs:

Control and sequence bugs include paths left out, unreachable code, improper nesting of loops, loop-back or loop termination criteria incorrect, missing process steps, duplicated processing, unnecessary processing, rampaging, GOTO's, ill-conceived (not properly planned) switches, spaghetti code, and worst of all, pachinko code.

One reason for control flow bugs is that this area is amenable (supportive) too theoretical treatment.

Most of the control flow bugs are easily tested and caught in unit testing.

Another reason for control flow bugs is that use of old code especially ALP & COBOL code are dominated by control flow bugs.

Control and sequence bugs at all levels are caught by testing, especially structural testing, more specifically path testing combined with a bottom line functional test based on a specification.

Logic Bugs:

Bugs in logic, especially those related to misunderstanding how case statements and logic operators behave singly and combinations

Also includes evaluation of Boolean expressions in deeply nested IF-THEN-ELSE constructs.

If the bugs are parts of logical (i.e. Boolean) processing not related to control flow, they are characterized as processing bugs.

If the bugs are parts of a logical expression (i.e. control-flow statement) which is used to direct the control flow, then they are categorized as control-flow bugs.
**Processing Bugs:**

Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection and general processing.

Examples of Processing bugs include: Incorrect conversion from one data representation to other, ignoring overflow, improper use of greater-than-or-equal etc

Although these bugs are frequent (12%), they tend to be caught in good unit testing.

**Initialization Bugs:**

Initialization bugs are common. Initialization bugs can be improper and superfluous.

Superfluous bugs are generally less harmful but can affect performance.

Typical initialization bugs include: Forgetting to initialize the variables before first use, assuming that they are initialized elsewhere, initializing to the wrong format, representation or type etc

- Explicit declaration of all variables, as in Pascal, can reduce some initialization problems.

**Data-Flow Bugs and Anomalies:**

Most initialization bugs are special case of data flow anomalies.

A data flow anomaly occurs where there is a path along which we expect to do something unreasonable with data, such as using an uninitialized variable, attempting to use a variable before it exists, modifying and then not storing or using the result, or initializing twice without an intermediate use.

**DATA BUGS:**

Data bugs include all bugs that arise from the specification of data objects, their formats, the number of such objects, and their initial values.

Data Bugs are at least as common as bugs in code, but they are often treated as if they did not exist at all.
**Code migrates data:** Software is evolving towards programs in which more and more of the control and processing functions are stored in tables.

Because of this, there is an increasing awareness that bugs in code are only half the battle and the data problems should be given equal attention.

**Dynamic Data Vs Static data:**

Dynamic data are transitory. Whatever their purpose their lifetime is relatively short, typically the processing time of one transaction. A storage object may be used to hold dynamic data of different types, with different formats, attributes and residues.

Dynamic data bugs are due to leftover garbage in a shared resource. This can be handled in one of the three ways: (1) Clean up after the use by the user (2) Common Cleanup by the resource manager (3) No Clean up

Static Data are fixed in form and content. They appear in the source code or database directly or indirectly, for example a number, a string of characters, or a bit pattern.

Compile time processing will solve the bugs caused by static data.

**Information, parameter, and control:** Static or dynamic

1. Data can serve in one of three roles, or in combination of roles: as a parameter, for control, or for information.

**Content, Structure and Attributes:** Content can be an actual bit pattern, character string, or number put into a data structure. Content is a pure bit pattern and has no meaning unless it is interpreted by a hardware or software processor. All data bugs result in the corruption or misinterpretation of content. Structure relates to the size, shape and numbers that describe the data object that is memory location used to store the content. (E.g. A two dimensional array). Attributes relates to the specification meaning that is the semantics associated with the contents of a data object.

**CODING BUGS:**

- Coding errors of all kinds can create any of the other kind of bugs.
- Syntax errors are generally not important in the scheme of things if the source language translator has adequate syntax checking.
- If a program has many syntax errors, then we should expect many logic and coding bugs.
- The documentation bugs are also considered as coding bugs which may mislead the maintenance programmers.
- Various categories of bugs in Interface, Integration, and System Bugs are:

**External Interfaces:**

The external interfaces are the means used to communicate with the world. These include devices, actuators, sensors, input terminals, printers, and communication lines.

The primary design criterion for an interface with outside world should be robustness.

All external interfaces, human or machine should employ a protocol.

The protocol may be wrong or incorrectly implemented.

Other external interface bugs are: invalid timing or sequence assumptions related to external signals

Misunderstanding external input or output formats.

Insufficient tolerance to bad input data.

**Internal Interfaces:**

Internal interfaces are in principle not different from external interfaces but they are more controlled.

A best example for internal interfaces is communicating routines.

The external environment is fixed and the system must adapt to it but the internal environment, which consists of interfaces with other components, can be negotiated.

Internal interfaces have the same problem as external interfaces.

**Hardware Architecture:**
Bugs related to hardware architecture originate mostly from misunderstanding how the hardware works.

Examples of hardware architecture bugs: address generation error, i/o device operation / instruction error, waiting too long for a response, incorrect interrupt handling etc.

The remedy for hardware architecture and interface problems is two fold: (1) Good Programming and Testing (2) Centralization of hardware interface software in programs written by hardware interface specialists.

**Operating System Bugs:**

Program bugs related to the operating system are a combination of hardware architecture and interface bugs mostly caused by a misunderstanding of what it is the operating system does.

Use operating system interface specialists, and use explicit interface modules or macros for all operating system calls.

This approach may not eliminate the bugs but at least will localize them and make testing easier.

**Software Architecture:**

Software architecture bugs are the kind that called - interactive.

Routines can pass unit and integration testing without revealing such bugs.

Many of them depend on load, and their symptoms emerge only when the system is stressed.

Sample for such bugs: Assumption that there will be no interrupts, Failure to block or un block interrupts, Assumption that memory and registers were initialized or not initialized etc.

Careful integration of modules and subjecting the final system to a stress test are effective methods for these bugs.

**Control and Sequence Bugs (Systems Level):**

These bugs include: Ignored timing, Assuming that events occur in a specified sequence, Working on data before all the data have arrived from disc, Waiting for an impossible combination of
prerequisites, Missing, wrong, redundant or superfluous process steps.

The remedy for these bugs is highly structured sequence control.

Specialize, internal, sequence control mechanisms are helpful.

**Resource Management Problems:**

Memory is subdivided into dynamically allocated resources such as buffer blocks, queue blocks, task control blocks, and overlay buffers.

External mass storage units such as discs, are subdivided into memory resource pools.

Some resource management and usage bugs: Required resource not obtained, Wrong resource used, Resource is already in use, Resource dead lock etc

**Resource Management Remedies:** A design remedy that prevents bugs is always preferable to a test method that discovers them.

The design remedy in resource management is to keep the resource structure simple: the fewest different kinds of resources, the fewest pools, and no private resource management.

**Integration Bugs:**

Integration bugs are bugs having to do with the integration of, and with the interfaces between, working and tested components.

These bugs results from inconsistencies or incompatibilities between components.

The communication methods include data structures, call sequences, registers, semaphores, and communication links and protocols results in integration bugs.

The integration bugs do not constitute a big bug category (9%) they are expensive category because they are usually caught late in the game and because they force changes in several components and/or data structures.
**System Bugs:**

System bugs covering all kinds of bugs that cannot be ascribed to a component or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operating systems.

There can be no meaningful system testing until there has been thorough component and integration testing.

System bugs are infrequent (1.7%) but very important because they are often found only after the system has been fielded.

**TEST AND TEST DESIGN BUGS:**

Testing: testers have no immunity to bugs. Tests require complicated scenarios and databases.

They require code or the equivalent to execute and consequently they can have bugs.

Test criteria: if the specification is correct, it is correctly interpreted and implemented, and a proper test has been designed; but the criterion by which the software’s behavior is judged may be incorrect or impossible. So, a proper test criterion has to be designed. The more complicated the criteria, the likelier they are to have bugs.

**Remedies:** The remedies of test bugs are:

**Test Debugging:** The first remedy for test bugs is testing and debugging the tests. Test debugging, when compared to program debugging, is easier because tests, when properly designed are simpler than programs.

And do not have to make concessions to efficiency.

**Test Quality Assurance:** Programmers have the right to ask how quality in independent testing is monitored.

**Test Execution Automation:** The history of software bug removal and prevention is indistinguishable from the history of programming automation aids. Assemblers, loaders, compilers are developed to reduce the incidence of programming and operation
errors. Test execution bugs are virtually eliminated by various test execution automation tools.

**Test Design Automation:** Just as much of software development has been automated, much test design can be and has been automated. For a given productivity rate, automation reduces the bug count - be it for software or be it for tests.

- **Classroom Interaction (two to three questions):**
  - What is purpose of Testing?
  - What are the differences between Builder and Buyer?
  - What is Inspection method?
  - What are the Integration Bugs?

- **Tutorial Questions:**
  - Write the differences between Debugging and Testing?
  - Write the differences between Functional versus Structural?

- **Assignment questions:**
  - Explain the Data bugs, coding bugs and Resource management Problems with examples?

- **Exercise questions / Long answer questions / Project possibilities:**
  - Explain different kinds of bugs what are the reasons to occur those kinds of bugs and what are the test remedies for them?

- **Online resources:**
  - [www.researchgate.com](http://www.researchgate.com)
  - [www.cs.drexel](http://www.cs.drexel)
Unit – III

- **Objective of the Unit (Two to Three lines):**
- Understand the concept of path testing.
- Identify the components of a control flow diagram and compare the same with a flowchart.
- Interpret a control flow graph and demonstrate the complete path testing to achieve C1+C2.
- Classify the predicates and variables as dependant/independent and correlated/uncorrelated.
- Understand the path sensitizing and Instrumentation methods and classify whether the path is achievable or not.

**Topic: FLOWGRAPHS AND PATH TESTING:**

- **Motivation (Why this topic is significant for the discussion?):**
- if it is difficult to prove weather the program has bugs or not can be easy to prove in terms of flow graph and flowchart by applying Path sensitization and path Instrumentation and all path concepts.

**Notes: PATH TESTING:**
- Path Testing is the name given to a family of test techniques based on judiciously selecting a set of test paths through the program.
- If the set of paths are properly chosen then we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement has been executed at least once.
- Path testing techniques are the oldest of all structural test techniques.
- Path testing is most applicable to new software for unit testing. It is a structural technique.
- It requires complete knowledge of the program’s structure.
- It is most often used by programmers to unit test their own code.
- The effectiveness of path testing rapidly deteriorates as the size of the software aggregate under test increases.

- **THE BUG ASSUMPTION:**
- The bug assumption for the path testing strategies is that something has gone wrong with the software that makes it take a different path than intended.
- As an example "GOTO X" where "GOTO Y" had been intended.
Structured programming languages prevent many of the bugs targeted by path testing: as a consequence the effectiveness for path testing for these languages is reduced and for old code in COBOL, ALP, FORTRAN and Basic, the path testing is indispensable.

- **CONTROL FLOW GRAPHS:**
  - The control flow graph is a graphical representation of a program’s control structure. It uses the elements named process blocks, decisions, and junctions.
  - The flow graph is similar to the earlier flowchart, with which it is not to be confused.
  - **Flow Graph Elements:** A flow graph contains four different types of elements. (1) Process Block (2) Decisions (3) Junctions (4) Case Statements

1. **Process Block:**
   - A process block is a sequence of program statements uninterrupted by either decisions or junctions.
   - It is a sequence of statements such that if any one of statement of the block is executed, then all statement thereof are executed.
   - Formally, a process block is a piece of straight line code of one statement or hundreds of statements.
   - A process has one entry and one exit. It can consist of a single statement or instruction, a sequence of statements or instructions, a single entry/exit subroutine, a macro or function call, or a sequence of these.

2. **Decisions:**
   - A decision is a program point at which the control flow can diverge.
   - Machine language conditional branch and conditional skip instructions are examples of decisions.
   - Most of the decisions are two-way but some are three way branches in control flow.

3. **Case Statements:**
   - A case statement is a multi-way branch or decisions.
   - Examples of case statement are a jump table in assembly language, and the PASCAL case statement.
• From the point of view of test design, there are no differences between Decisions and Case Statements

4. **Junctions:**
   - A junction is a point in the program where the control flow can merge.
   - Examples of junctions are: the target of a jump or skip instruction in ALP, a label that is a target of GOTO.

• **CONTROL FLOW GRAPHS Vs FLOWCHARTS:**
  - A program’s flow chart resembles a control flow graph.
  - In flow graphs, we don’t show the details of what is in a process block.
  - In flow charts every part of the process block is drawn.
  - The flowchart focuses on process steps, whereas the flow graph focuses on control flow of the program.
  - The act of drawing a control flow graph is a useful tool that can help us clarify the control flow and data flow issues.

To understand this, we will go through an example written in a FORTRAN like programming language called **Programming Design Language (PDL)**

**FLOWGRAPH AND FLOWCHART GENERATION:**
Flowcharts can be
0. Handwritten by the programmer.
1. Automatically produced by a flowcharting program based on a mechanical analysis of the source code.
2. Semi automatically produced by a flow charting program based in part on structural analysis of the source code and in part on directions given by the programmer.

**PATH TESTING - PATHS, NODES AND LINKS:**
0. **Path:** a path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same junction, decision, or exit.
1. A path may go through several junctions, processes, or decisions, one or more times.

Paths consist of segments.
The segment is a link - a single process that lies between two nodes.
0. A path segment is succession of consecutive links that belongs to some path.
1. The length of path measured by the number of links in it and not by the number of the instructions or statements executed along that path.
The name of a path is the name of the nodes along the path.

**FUNDAMENTAL PATH SELECTION CRITERIA:**

0. There are many paths between the entry and exit of a typical routine.
1. Every decision doubles the number of potential paths. And every loop multiplies the number of potential paths by the number of different iteration values possible for the loop.

Defining complete testing:

0. Exercise every path from entry to exit
1. Exercise every statement or instruction at least once
2. Exercise every branch and case statement, in each direction at least once
3. If prescription 1 is followed then 2 and 3 are automatically followed. But it is impractical for most routines. It can be done for the routines that have no loops, in which it is equivalent to 2 and 3 prescriptions.

**EXAMPLE:** Here is the correct version.

![Diagram of the correct version](image)

For X negative, the output is $X + A$, while for X greater than or equal to zero, the output is $X + 2A$. Following prescription 2 and executing every statement, but not every branch, would not reveal the bug in the following incorrect version:

![Diagram of the incorrect version](image)

A negative value produces the correct answer. Every
statement can be executed, but if the test cases do not force each branch to be taken, the bug can remain hidden. The next example uses a test based on executing each branch but does not force the execution of all statements:

The hidden loop around label 100 is not revealed by tests based on prescription 3 alone because no test forces the execution of statement 100 and the following GOTO statement. Furthermore, label 100 is not flagged by the compiler as an unreferenced label and the subsequent GOTO does not refer to an undefined label.

**PATH TESTING CRITERIA:**

0. Any testing strategy based on paths must at least both exercise every instruction and take branches in all directions.

1. A set of tests that does this is not complete in an absolute sense, but it is complete in the sense that anything less must leave something untested.

2. So we have explored three different testing criteria or strategies out of a potentially infinite family of strategies.

3. **Path Testing (P_{inf}):**

Execute all possible control flow paths through the program: typically, this is restricted to all possible entry/exit paths through the program.
If we achieve this prescription, we are said to have achieved 100% path coverage. This is the strongest criterion in the path testing strategy family: it is generally impossible to achieve.

**Statement Testing (P₁):**

Execute all statements in the program at least once under some test. If we do enough tests to achieve this, we are said to have achieved 100% statement coverage.

An alternate equivalent characterization is to say that we have achieved 100% node coverage. We denote this by C₁.

This is the weakest criterion in the family: testing less than this for new software is unconscionable (unprincipled or cannot be accepted) and should be criminalized.

**Branch Testing (P₂):**

Execute enough tests to assure that every branch alternative has been exercised at least once under some test.

If we do enough tests to achieve this prescription, then we have achieved 100% branch coverage.

An alternative characterization is to say that we have achieved 100% link coverage.

For structured software, branch testing and therefore branch coverage strictly includes statement coverage.

We denote branch coverage by C₂.

**Commonsense and Strategies:**

Branch and statement coverage are accepted today as the minimum mandatory testing requirement.

The question "why not use a judicious sampling of paths?, what is wrong with leaving some code, untested?" is ineffectual in the view of common sense and experience since: (1.) Not testing a piece of a code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs. (2.) The high probability paths are always thoroughly tested if only to demonstrate that the system works properly.
**Which paths to be tested?** You must pick enough paths to achieve C1+C2. The question of what is the fewest number of such paths is interesting to the designer of test tools that help automate the path testing, but it is not crucial to the pragmatic (practical) design of tests. It is better to make many simple paths than a few complicated paths.

**APPLICATION OF PATH TESTING:**

- **Practical Suggestions in Path Testing:**
  0. Draw the control flow graph on a single sheet of paper.
  1. Make several copies - as many as you will need for coverage (C1+C2) and several more.
  2. Use a yellow highlighting marker to trace paths. Copy the paths onto a master sheets.
  3. Continue tracing paths until all lines on the master sheet are covered, indicating that you appear to have achieved C1+C2.
4. As you trace the paths, create a table that shows the paths, the coverage status of each process, and each decision.

5. The above paths lead to the following table considering

<table>
<thead>
<tr>
<th>PATHS</th>
<th>DECISIONS</th>
<th>PROCESS–LINK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4  6  7  9</td>
<td>a b c d e f g h i j k l m</td>
</tr>
<tr>
<td>abede</td>
<td>YES YES</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>abhkgde</td>
<td>NO YES</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>abhlibode</td>
<td>NO,YES YES</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>abdfjgde</td>
<td>YES NO,YES</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>abdfmibede</td>
<td>YES NO,YES</td>
<td>✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>
LOOPS:

a, b) Nested Loops

M = 3

(c) Concatenated Loops

M = 4

M = 3

d, e, f) Horrible Loops

M = 6
1. Loop Testing Time:

- Any kind of loop can lead to long testing time, especially if all the extreme value cases are to attempted (Max-1, Max, and Max+1).
- This situation is obviously worse for nested and dependent concatenated loops.
- Consider nested loops in which testing the combination of extreme values lead to long test times. Several options to deal with:
  1. Prove that the combined extreme cases are hypothetically possible, they are not possible in the real world.

**PREDICATES, PATH PREDICATES AND ACHIEVABLE PATHS:**

**PREDICATE:** The logical function evaluated at a decision is called Predicate. The direction taken at a decision depends on the value of decision variable. Some examples are: A>0, x+y>=90........

**PATH PREDICATE:** A predicate associated with a path is called a Path Predicate. For example, "x is greater than zero", "x+y>=90", "w is either negative or equal to 10 is true" is a sequence of predicates whose truth values will cause the routine to take a specific path.

**PREDICATE INTERPRETATION:**

The simplest predicate depends only on input variables.

For example if x1,x2 are inputs, the predicate might be x1+x2>=7, given the values of x1 and x2 the direction taken through the decision is based on the predicate is determined at input time and does not depend on processing.

A pair of predicates whose outcomes depend on one or more variables in common are said to be correlated predicates. For example, the predicate X==Y is followed by another predicate X+Y == 8. If we select X and Y values to satisfy

Every path through a routine is achievable only if all the predicates in that routine are uncorrelated.
Interpretation means in the predicates all the variable values need to be substituted in order to get in terms of solely input vectors.

**PATH PREDICATE EXPRESSIONS:**

A path predicate expression is a set of Boolean expressions, all of which must be satisfied to achieve the selected path.

**TESTING BLINDNESS:**

- Testing Blindness is a pathological (harmful) situation in which the desired path is achieved for the wrong reason.
- There are three types of Testing Blindness:

**Assignment Blindness:**

Assignment blindness occurs when the buggy predicate appears to work correctly because the specific value chosen for an assignment statement works with both the correct and incorrect predicate.

**Equality Blindness:**

Equality blindness occurs when the path selected by a prior predicate results in a value that works both for the correct and buggy predicate.

**Self Blindness:**

Self blindness occurs when the buggy predicate is a multiple of the correct predicate and as a result is indistinguishable along that path.

**PATH SENSITIZING:**

**REVIEW: ACHIEVABLE AND UNACHIEVABLE PATHS:**

We want to select and test enough paths to achieve a satisfactory notion of test completeness such as $C_1 + C_2$.

Extract the programs control flowgraph and select a set of tentative covering paths.

For any path in that set, interpret the predicates along the path as needed to express them in terms of the input vector. In general
individual predicates are compound or may become compound as a result of interpretation.

Trace the path through, multiplying the individual compound predicates to achieve a boolean expression such as


\[\text{ADFGHIJKL} + \text{AEFGHIJKL} + \text{BCDFGHIJKL} + \text{BCEFGHIJKLM} \]

Each product term denotes a set of inequalities that if solved will yield an input vector that will drive the routine along the designated path.

Solve any one of the inequality sets for the chosen path and you have found a set of input values for the path.

If you can find a solution, then the path is achievable.

If you can't find a solution to any of the sets of inequalities, the path is unachievable.

The act of finding a set of solutions to the path predicate expression is called **PATH SENSITIZATION**.

**PATH INSTRUMENTATION**: This is what we have to do to confirm that desired outcome for the intended path.

**Co-incidental Correctness**: The coincidental correctness stands for achieving the desired outcome for wrong reason

For all the cases the outcome is two, this is not aim of designing case statements, each case should give different outcome that is the intention.
of designing case statement but here fails to avoid this we will move to Path Instrumentation.

The types of instrumentation methods include:

**Interpretive Trace Program:**

Tracing the path and calculating the intermediate value so that can achieve path instrumentation.

**Traversal Marker or Link Marker:**

A simple and effective form of instrumentation is called a traversal marker or link marker.

Name every link by a lower case letter.

Instrument the links so that the link's name is recorded when the link is executed.

Why single link marker is not enough

![Diagram](image)

Due to Ramping goto instead of going ikm path it goes to ikm through process B also give same result. So it fails to avoid this double link markers are implemented.

**Two Link Marker Method:**

Adding the link markers at the begin and at the end of the path also. Then correctly identify which path it is.
**Link Counter:** A less disruptive (and less informative) instrumentation method is based on counters. Instead of a unique link name to be pushed into a string when the link is traversed, we simply increment a link counter. We now confirm that the path length is as expected. The same problem that led us to double link markers also leads us to double link counters.

**APPLICATION OF PATH TESTING:**
Integration, Coverage, and paths in called Components: It must well worked for Unit level path testing

**New code:** wholly new or substantially modified code should always be subjected enough path testing to achieve C2.

**Rehosting:** Software is rehosted because it is no longer cost-effective to support the environment in which it runs

The objective of software rehosting is to change the environment and not the rehosted software.

**Maintenance:** maintenance test methods into efficient methodologies that provide the kind of coverage should achieve in maintenance.

- **Classroom Interaction (two to three questions):**
  - What is a Path? Path segment and Path length?
  - What is coincidental correctness?
  - What is Path Instrumentation?
• **Tutorial Questions:**
  • Write about Path Instrumentation Techniques?
  • What is Path sensitization and explain it?
• **Assignment questions:**
  • What is a predicates and how predicates are helpful in achieving C1+C2 ?
  • Write a program with at least 4 decisions and prove it weather it is under coverage or not?

• **Exercise questions / Long answer questions / Project possibilities**
  Implement Path Instrumentation with simulation?

• **Online resources:**
  • www.mcr.org.in
  • www.nri.edu.in
  • www.ece.uc.edu
Unit-IV

- **Objective of the unit:**
  - students understand what are the states of the variables, what are the reasons to occur anomalies, they can realize and write programs without occurring of bugs.
- **Topic: Data-Flow Testing**
- **Motivation:**
  - At least half of contemporary source code consists of data declaration statements—that is, statements that define data structure, individual objects, initial or default values and attributes.

- **Notes: DATA FLOW MACHINES:**
  - There are two types of data flow machines with different architectures. (1) Von Neumann machines (2) Multi-instruction, multi-data machines (MIMD).
  - **Von Neumann Machine Architecture:**
    - Most computers today are von-Neumann machines.
    - This architecture features interchangeable storage of instructions and data in the same memory units.
    - The Von Neumann machine Architecture executes one instruction at a time in the following, micro instruction sequence:
      1. Fetch instruction from memory
      2. Interpret instruction
      3. Fetch operands
      4. Process or Execute
      5. Store result
      6. Increment program counter
      7. GOTO 1
  - **Multi-instruction, Multi-data machines (MIMD) Architecture:**
    - These machines can fetch several instructions and objects in parallel.
    - They can also do arithmetic and logical operations simultaneously on different data objects.
    - The decision of how to sequence them depends on the compiler.
- **BUG ASSUMPTION:**
  - The bug assumption for data-flow testing strategies is that control flow is generally correct and that something has gone wrong with the software so that data objects are not available when they should be, or silly things are being done to data objects.
Also, if there is a control-flow problem, we expect it to have symptoms that can be detected by data-flow analysis.

Although we'll be doing data-flow testing, we won't be using data flow graphs as such. Rather, we'll use an ordinary control flow graph annotated to show what happens to the data objects of interest at the moment.

**DATA FLOW GRAPHS:**

- The data flow graph is a graph consisting of nodes and directed links.

- We will use a control graph to show what happens to data objects of interest at that moment.
- Our objective is to expose deviations between the data flows we have and the data flows we want.
- **Data Object State and Usage:**
  - Data Objects can be created, killed and used.
  - They can be used in two distinct ways: (1) In a Calculation (2) As a part of a Control Flow Predicate.
  - The following symbols denote these possibilities:
    1. **Defined:** d - defined, created, initialized etc
    2. **Killed or undefined:** k - killed, undefined, released etc
    3. **Usage:** u - used for something (c - used in Calculations, p - used in a predicate)

  - 1. **Defined (d):**
    - An object is defined explicitly when it appears in a data declaration.
    - Or implicitly when it appears on the left hand side of the assignment.
    - It is also to be used to mean that a file has been opened.
    - A dynamically allocated object has been allocated.
    - Something is pushed on to the stack.
    - A record written.

  - 2. **Killed or Undefined (k):**
    - An object is killed on undefined when it is released or otherwise made unavailable.
    - When its contents are no longer known with certitude (with absolute certainty / perfectness).
    - Release of dynamically allocated objects back to the availability pool.
    - Return of records.
    - The old top of the stack after it is popped.
- An assignment statement can kill and redefine immediately. For example, if A had been previously defined and we do a new assignment such as A := 17, we have killed A's previous value and redefined A

**3. Usage (u):**
- A variable is used for computation (c) when it appears on the right hand side of an assignment statement.
- A file record is read or written.
- It is used in a Predicate (p) when it appears directly in a predicate.

**DATA FLOW ANOMALIES:**
An anomaly is denoted by a two-character sequence of actions.
For example, ku means that the object is killed and then used, where as dd means that the object is defined twice without an intervening usage.

What is an anomaly is depend on the application.
There are nine possible two-letter combinations for d, k and u. some are bugs, some are suspicious, and some are okay.

0. **dd**: possibly harmless but suspicious. Why define the object twice without an intervening usage?
1. **dk**: probably a bug. Why define the object without using it?
2. **du**: the normal case. The object is defined and then used.
3. **kd**: normal situation. An object is killed and then redefined.
4. **kk**: harmless but probably buggy. Did you want to be sure it was really killed?
5. **ku**: a bug. the object does not exist.
6. **ud**: usually not a bug because the language permits reassignment at almost any time.
7. **uk**: normal situation.
8. **uu**: normal situation.

In addition to the two letter situations, there are six single letter situations.

We will use a leading dash to mean that nothing of interest (d,k,u) occurs prior to the action noted along the entry-exit path of interest.
A trailing dash to mean that nothing happens after the point of interest to the exit.

They possible anomalies are:
0. **-k**: possibly anomalous because from the entrance to this point on the path, the variable had not been defined. We are killing a variable that does not exist.
1. **d**: okay. This is just the first definition along this path.
2. **u**: possibly anomalous. Not anomalous if the variable is global and has been previously defined.
3. **k**: not anomalous. The last thing done on this path was to kill the variable.
4. **d**: possibly anomalous. The variable was defined and not used on this path. But this could be a global definition.
5. **u**: not anomalous. The variable was used but not killed on this path. Although this sequence is not anomalous, it signals a frequent kind of bug. If d and k mean dynamic storage allocation and return respectively, this could be an instance in which a dynamically allocated object was not returned to the pool after use.

**DATA FLOW ANOMALY STATE GRAPH:**

Data flow anomaly model prescribes that an object can be in one of four distinct states:

0. **K**: undefined, previously killed, does not exist
1. **D**: defined but not yet used for anything
2. **U**: has been used for computation or in predicate
3. **A**: anomalous

These capital letters (K, D, U, A) denote the state of the variable and should not be confused with the program action, denoted by lower case letters.

![Data Flow Anomaly State Graph Diagram]

**Unforgiving Data - Flow Anomaly Flow Graph:** Unforgiving model, in which once a variable becomes anomalous it can never return to a state of grace.

**Forgiving Data - Flow Anomaly Flow Graph:** Forgiving model is an alternate model where redemption (recover) from the anomalous state is possible.
This graph has three normal and three anomalous states and he considers the kk sequence not to be anomalous. The difference between this state graph and is that redemption is possible. A proper action from any of the three anomalous states returns the variable to a useful working state.

The point of showing you this alternative anomaly state graph is to demonstrate that the specifics of an anomaly depends on such things as language, application, context, or even your frame of mind. In principle, you must create a new definition of data flow anomaly (e.g., a new state graph) in each situation. You must at least verify that the anomaly definition behind the theory or imbedded in a data flow anomaly test tool is appropriate to your situation.

- **STATIC Vs DYNAMIC ANOMALY DETECTION:**
  - Static analysis is analysis done on source code without actually executing it. For example: source code syntax error detection is the static analysis result.
  - Dynamic analysis is done on the fly as the program is being executed and is based on intermediate values that result from the program’s execution. For example: a division by zero warning is the dynamic result.
  - **Why Static Analysis isn’t enough?** There are many things for which current notions of static analysis are inadequate. They are:
- **Dead Variables**: Although it is often possible to prove that a variable is dead or alive at a given point in the program, the general problem is unsolvable.

- **Arrays**: Arrays are problematic in that the array is defined or killed as a single object, but reference is to specific locations within the array. Array pointers are usually dynamically calculated, so there’s no way to do a static analysis to validate the pointer value. In many languages, dynamically allocated arrays contain garbage unless explicitly initialized and therefore, -u anomalies are possible.

- **Records and Pointers**: The array problem and the difficulty with pointers is a special case of multipart data structures. We have the same problem with records and the pointers to them. Also, in many applications we create files and their names dynamically and there’s no way to determine, without execution, whether such objects are in the proper state on a given path or, for that matter, whether they exist at all.

- **Dynamic Subroutine and Function Names in a Call**: subroutine or function name is a dynamic variable in a call. What is passed, or a combination of subroutine names and data objects, is constructed on a specific path. There’s no way, without executing the path, to determine whether the call is correct or not.

- **False Anomalies**: Anomalies are specific to paths. Even a "clear bug" such as ku may not be a bug if the path along which the anomaly exists is unachievable. Such "anomalies" are false anomalies. Unfortunately, the problem of determining whether a path is or is not achievable is unsolvable.

- **Recoverable Anomalies and Alternate State Graphs**: What constitutes an anomaly depends on context, application, and semantics. How does the compiler know which model I have in mind? It can't because the definition of "anomaly" is not fundamental. The language processor must have a built-in anomaly definition with which you may or may not (with good reason) agree.

- **Concurrency, Interrupts, System Issues**: As soon as we get away from the simple single-task uniprocessor environment and start thinking in terms of systems, most anomaly issues become vastly more complicated. How often do we define or
create data objects at an interrupt level so that they can be processed by a lower-priority routine? Interrupts can make the "correct" anomalous and the "anomalous" correct. True concurrency (as in an MIMD machine) and pseudo concurrency (as in multiprocessing) systems can do the same to us. Much of integration and system testing is aimed at detecting data-flow anomalies that cannot be detected in the context of a single routine.

- Although static analysis methods have limits, they are worth using and a continuing trend in language processor design has been better static analysis methods, especially for data flow anomaly detection. That's good because it means there's less for us to do as testers and we have far too much to do as it is.

**DATA FLOW MODEL:**

- The data flow model is based on the program's control flow graph - Don't confuse that with the program's data flow graph.
- Here we annotate each link with symbols (for example, d, k, u, c, p) or sequences of symbols (for example, dd, du, ddd) that denote the sequence of data operations on that link with respect to the variable of interest. Such annotations are called link weights.
- The control flow graph structure is same for every variable: it is the weights that change.
- **Components of the model:**
  1. To every statement there is a node, whose name is unique. Every node has at least one out link and at least one in link except for exit nodes and entry nodes.
  2. Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statements (e.g., END, RETURN), to complete the graph. Similarly, entry nodes are dummy nodes placed at entry statements (e.g., BEGIN) for the same reason.

The out link of simple statements (statements with only one out link) is weighted by the proper sequence of data-flow actions for that statement. Note that the sequence can consist of more than one letter

3. Predicate nodes (e.g., IF-THEN-ELSE, DO WHILE, CASE) are weighted with the p - use(s) on every outlink, appropriate to that outlink.
4. Every sequence of simple statements (e.g., a sequence of nodes with one inlink and one outlink)
can be replaced by a pair of nodes that has, as weights on the link between them, the concatenation of link weights.

5. If there are several data-flow actions on a given link for a given variable, then the weight of the link is denoted by the sequence of actions on that link for that variable.

6. Conversely, a link with several data-flow actions on it can be replaced by a succession of equivalent links, each of which has at most one data-flow action for any variable.

Let us consider the example:

**CODE* (PDL)**

```
INPUT X, Y
Z := X + Y
V := X - Y
IF Z >= 0 GOTO SAM

JOE: Z := Z - 1
SAM: Z := Z + V
FOR U = 0 TO Z
    V(U), U(V) := (Z + V)*U
    IF V(U) = 0 GOTO JOE
    Z := Z - 1
    IF Z = 0 GOTO ELL
END

V(U-1) := V(U+1) + U(V-1)
ELL: V(U) + U(V) := U + V
IF U = V GOTO JOE
IF U > V THEN U := Z
Z := U
```

* A contrived horror
STRATEGIES OF DATA FLOW TESTING:

- TERMINOLOGY:

**Definition-Clear Path Segment**, with respect to variable X, is a connected sequence of links such that X is (possibly) defined on the first link and not redefined or killed on any subsequent link of that path segment.

**Loop-Free Path Segment** is a path segment for which every node in it is visited at most once. For Example, path (4,5,6,7,8,10)

1. **Simple path segment** is a path segment in which at most one node is visited twice. For example, in Figure 3.10, (7,4,5,6,7) is a simple path segment. A simple path segment is either loop-free or if there is a loop, only one node is involved.
2. A **du path** from node i to k is a path segment such that if the last link has a computational use of X, then the path is simple and definition-clear; if the penultimate (last but one) node is j - that is, the path is (i,p,q,...,r,s,t,j,k) and link (j,k) has a predicate use - then the path from i to j is both loop-free and definition-clear.

**STRATEGIES**: The structural test strategies discussed below are based on the program’s control flow graph. They differ in the extent to which predicate uses and/or computational uses of variables are included in the test set. Various types of data flow testing strategies in decreasing order of their effectiveness are:

**All - du Paths (ADUP)**: The all-du-paths (ADUP) strategy is the strongest data-flow testing strategy discussed here. It requires that every du path from every definition of every variable to every use of that definition be exercised under some test.

0. The all-du-paths strategy is a strong criterion, but it does not take as many tests as it might seem at first because any one test simultaneously satisfies the criterion for several definitions and uses of several different variables.

**All Uses Strategy (AU)**: The all uses strategy is that at least one definition clear path from every definition of every variable to every use of that definition be exercised under some test. Just as we reduced our ambitions
by stepping down from all paths (P) to branch coverage (C2), say, we can reduce the number of test cases by asking that the test set should include at least one path segment from every definition to every use that can be reached by that definition.

All p-uses/some c-uses strategy (APU+C) : For every variable and every definition of that variable, include at least one definition free path from the definition to every predicate use; if there are definitions of the variables that are not covered by the above prescription, then add computational use test cases as required to cover every definition.

All c-uses/some p-uses strategy (ACU+P) : The all c-uses/some p-uses strategy (ACU+P) is to first ensure coverage by computational use cases and if any definition is not covered by the previously selected paths, add such predicate use cases as are needed to assure that every definition is included in some test.

All Definitions Strategy (AD) : The all definitions strategy asks only every definition of every variable be covered by at least one use of that variable, be that use a computational use or a predicate use.

1. All Predicate use (APU), All Computational Uses (ACU) Strategies : The all predicate uses strategy is derived from APU+C strategy by dropping the requirement that we include a c-use for the variable if there are no p-uses for the variable. The all computational uses strategy is derived from ACU+P strategy by dropping the requirement that we include a p-use for the variable if there are no c-uses for the variable.

   It is intuitively obvious that ACU should be weaker than ACU+P and that APU should be weaker than APU+C.
ORDERING THE STRATEGIES:

- The right-hand side of this graph, along the path from "all paths" to "all statements" is the more interesting hierarchy for practical applications.
- Note that although ACU+P is stronger than ACU, both are incomparable to the predicate-biased strategies. Note also that "all definitions" is not comparable to ACU or APU.

SLICING AND DICING:
- A (static) program slice is a part of a program (e.g., a selected set of statements) defined with respect to a given variable X (where X is a simple variable or a data vector) and a statement i: it is the set of all statements that could (potentially, under static analysis) affect the value of X at statement i - where the influence of a faulty statement could result from an improper computational use or predicate use of some other variables at prior statements.
- If X is incorrect at statement i, it follows that the bug must be in the program slice for X with respect to i
- A program dice is a part of a slice in which all statements which are known to be correct have been removed.
In other words, a dice is obtained from a slice by incorporating information obtained through testing or experiment (e.g., debugging).

**Classroom Interaction (two to three questions):**

- Write the differences between SIMD and MIMD?
- What is the bug assumption for Dataflow Testing?
- What is anomaly?
- What are the differences between slice and Dice?

**Tutorial Questions:**
- Write the differences between forgiving and unforgiving data flow state graph?
- What are the situations will go for dynamic analysis?

**Assignment questions:**
What is Data flow model? Explain with an example?

**Exercise questions / Long answer questions / Project possibilities**
Explain all strategies according to its strength wise each with an example?

**Teacher observations (if any):** If students know this concepts all the simple errors such as declaring variable twice, no declaration, one variable using for other purpose these are all are eliminated in programming.

**Online resources:**

www.worldscientific.com

www.eecs.yorku.ca
Unit-V

Objective of the unit:

Students learn about basics of Path, with the help of reduction procedure calculate maximum path count and mean processing time and probability of a particular point along the routine and also know all the basics of graph matrices.

**Topic: PATHS, PATH PRODUCTS AND REGULAR EXPRESSIONS, GRAPH MATRICES**

**Motivation (Why this topic is significant for the discussion?):**

- Flow graphs are being an abstract representation of programs.
- Any question about a program can be cast into an equivalent question about an appropriate flowgraph.
- Most software development, testing and debugging tools use flow graphs analysis techniques.

- **Notes: PATH PRODUCTS:**
  - Normally flow graphs used to denote only control flow connectivity.
  - The simplest weight we can give to a link is a name.
  - Using link names as weights, we then convert the graphical flow graph into an equivalent algebraic like expressions which denotes the set of all possible paths from entry to exit for the flow graph.
  - Every link of a graph can be given a name.
  - The link name will be denoted by lower case italic letters.
  - The name of the path or path segment that corresponds to those links is expressed naturally by concatenating those link names.
  - For example, if you traverse links a, b, c and d along some path, the name for that path segment is abcd. This path name is also called a path product.
• **PATH EXPRESSION:**
  
  - Consider a pair of nodes in a graph and the set of paths between those nodes.

  Denote that set of paths by Upper case letter such as X, Y

  The + sign is understood to mean "or" between the two nodes of interest.

• **PATH PRODUCTS:**
  
  - The name of a path that consists of two successive path segments is conveniently expressed by the concatenation or **Path Product** of the segment names.
  
  - For example, if X and Y are defined as $X = abcde, Y = fghij$, then the path corresponding to $X$ followed by $Y$ is denoted by $XY = abcdefghij$

• **PATH SUMS:**
  
  - The "+" sign was used to denote the fact that path names were part of the same set of paths.
  
  - The "PATH SUM" denotes paths in parallel between nodes.
The first set of parallel paths is denoted by $X + Y + d$ and the second set by $U + V + W + h + i + j$.

- The path is a set union operation; it is clearly Commutative and Associative.
- RULE 2: $X+Y=Y+X$
- RULE 3: $(X+Y)+Z=X+(Y+Z)=X+Y+Z$

**DISTRIBUTIVE LAWS:**
- The product and sum operations are distributive, and the ordinary rules of multiplication apply; that is

RULE 4: $A(B+C)=AB+AC$ and $(B+C)D=BD+CD$

- Applying these rules to the below Figure 5.1a yields
- $e(a+b)(c+d)f= e(ac+ad+bc+bd)f = eacf+eadf+ebcf+ebdf$

**ABSORPTION RULE:**
- If $X$ and $Y$ denote the same set of paths, then the union of these sets is unchanged; consequently,

RULE 5: $X+X=X$ (Absorption Rule)

**LOOPS:**

Loops can be understood as an infinite set of parallel paths.

**REDUCTION PROCEDURE ALGORITHM:**
- This section presents a reduction procedure for converting a flow graph whose links are labeled with names into a path expression that denotes the set of all entry/exit paths in that
flow graph. The procedure is a node-by-node removal algorithm.

- The steps in Reduction Algorithm are as follows:
  1. Combine all serial links by multiplying their path expressions.
  2. Combine all parallel links by adding their path expressions.
  3. Remove all self-loops (from any node to itself) by replacing them with a link of the form $X^*$, where $X$ is the path expression of the link in that loop.

**STEPS 4 - 8 ARE IN THE ALGORITHM'S LOOP:**

4. Select any node for removal other than the initial or final node. Replace it with a set of equivalent links whose path expressions correspond to all the ways you can form a product of the set of inlinks with the set of outlinks of that node.
5. Combine any remaining serial links by multiplying their path expressions.
6. Combine all parallel links by adding their path expressions.
7. Remove all self-loops as in step 3.
8. Does the graph consist of a single link between the entry node and the exit node? If yes, then the path expression for that link is a path expression for the original flow graph; otherwise, return to step 4.

- **LOOP REMOVAL OPERATIONS:**
  - There are two ways of looking at the loop-removal operation:

    - In the first way, we remove the self-loop and then multiply all outgoing links by $Z^*$.
    - In the second way, we split the node into two equivalent nodes, call them $A$ and $A'$ and put in a link between them whose path expression is $Z^*$. Then we remove node $A'$ using steps 4 and 5 to yield outgoing links whose path expressions are $Z^*X$ and $Z^*Y$.

- **A REDUCTION PROCEDURE - EXAMPLE:**

---

[Diagram of flow graph showing loop removal operations]
Let us see by applying this algorithm to the following graph where we remove several nodes in order; that is

- Remove node 10 by applying step 4 and combine by step 5 to yield

- Remove node 9 by applying step 4 and 5 to yield

- Remove node 7 by steps 4 and 5, as follows:

- Remove node 8 by steps 4 and 5, to obtain:
**PARALLELTERMSTEP:**
Removal of node 8 above led to a pair of parallel links between nodes 4 and 5. Combine them to create a path expression for an equivalent link whose path expression is $c + gkh$; that is

```
1  a  3  b  4  c  5  d  6  e  2
```

**LOOPTERMSTEP**
Removing node 4 leads to a loop term. The graph has now been replaced with the following equivalent simpler graph:

```
1  a  3  b(c + gkh)  4  5  d  6  e  2
```

Continue the process by applying the loop-removal step as follows:

```
1  (bgjf) * b(c + gkh)  5  d  6  e  2
```

Removing node 5 produces:
- Remove the loop at node 6 to yield:

- Remove node 3 to yield:

- Removing the loop and then node 6 result in the following expression:

- APPLICATIONS

  - The purpose of the node removal algorithm is to present one much generalized concept - the path expression and way of getting it.
  - Every application follows this common pattern:
    1. Convert the program or graph into a path expression.
    2. Identify a property of interest and derive an appropriate set of "arithmetic" rules that characterizes the property.
    3. Replace the link names by the link weights for the property of interest. The path expression has now been converted to an expression in some algebra, such as ordinary algebra, regular expressions, or Boolean algebra. This algebraic expression summarizes the property of interest over the set of all paths.
    4. Simplify or evaluate the resulting "algebraic" expression to answer the question you asked.

- **MAXIMUM PATH COUNT ARITHMETIC:**
  - Label each link with a link weight that corresponds to the number of paths that link represents.
o Also mark each loop with the maximum number of times that loop can be taken. If the answer is infinite, you might as well stop the analysis because it is clear that the maximum number of paths will be infinite.
o There are three cases of interest: parallel links, serial links, and loops.

<table>
<thead>
<tr>
<th>Case</th>
<th>Path expression</th>
<th>Weight expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallels</td>
<td>A+B</td>
<td>W_A+W_B</td>
</tr>
<tr>
<td>Series</td>
<td>AB</td>
<td>W_AW_B</td>
</tr>
<tr>
<td>Loop</td>
<td>A^n</td>
<td>( \sum_{j=0}^{n} W_A^j )</td>
</tr>
</tbody>
</table>

**EXAMPLE:**
- The following is a reasonably well-structured program.

Each link represents a single link and consequently is given a weight of "1" to start. Let's say the outer loop will be taken exactly four times and inner Loop can be taken zero or three times. Its path expression, with a little work, is:

Path expression: \( a(b+c)d(e(f)i)^*fg(m+l)k)^*e(f)i)fgh \)

- **A:** The flow graph should be annotated by replacing the link name with the maximum of paths through that link (1) and also note the number of times for looping.
- **B:** Combine the first pair of parallel loops outside the loop and also the pair in the outer loop.
- **C:** Multiply the things out and remove nodes to clear the clutter.
1. **For the Inner Loop:**

**D:** Calculate the total weight of inner loop, which can execute a min. of 0 times and max. of 3 times. So, it inner loop can be evaluated as follows:

\[1^3 = 1^0 + 1^1 + 1^2 + 1^3 = 1 + 1 + 1 + 1 = 4\]

2. **E:** Multiply the link weights inside the loop: \(1 \times 4 = 4\)
3. **F:** Evaluate the loop by multiplying the link weights: \(2 \times 4 = 8\).
4. **G:** Simplifying the loop further results in the total maximum number of paths in the flowgraph:

\[2 \times 8^4 \times 2 = 32,768\]
Alternatively, you could have substituted a "1" for each link in the path expression and then simplified, as follows:

\[ a(b+c)d(e(\text{fi})^*f\text{g}(m+1)k)^*e(\text{fi})^*fgh \]

\[ = 1(1 + 1)1(1 \times 1)^31 \times 1 \times 1(1 + 1)^31(1 \times 1)^31 \times 1 \times 1 \]

\[ = 2(1^31 \times 1) \times 1(1 + 1)^31 \times 2^4 \]

\[ = 2 \times 8^4 \times 4 = 32,768 \]

This is the same result we got graphically.

**LOWER PATH COUNT ARITHMETIC:**
A lower bound on the number of paths in a routine can be approximated for structured flow graphs.

The arithmetic is as follows:

<table>
<thead>
<tr>
<th>Case</th>
<th>Path expression</th>
<th>Weight expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallels</td>
<td>( A+B )</td>
<td>( W_A+\max(W_A,W_B) )</td>
</tr>
<tr>
<td>Series</td>
<td>( AB )</td>
<td>( \max(W_A,W_B) )</td>
</tr>
<tr>
<td>Loop</td>
<td>( A^n )</td>
<td>1, ( W_1 )</td>
</tr>
</tbody>
</table>
EXAMPLE:

- From Step 4, the it would be different from the previous example:

CALCULATING THE PROBABILITY:
Path selection should be biased toward the low - rather than the high-probability paths.
This raises an interesting question:

What is the probability of being at a certain point in a routine?

Weights, Notations and Arithmetic:
- Probabilities can come into the act only at decisions (including decisions associated with loops).
- Annotate each outlink with a weight equal to the probability of going in that direction.
- Evidently, the sum of the outlink probabilities must equal 1
- For a simple loop, if the loop will be taken a mean of \( N \) times, the looping probability is \( \frac{N}{N+1} \) and the probability of not looping is \( \frac{1}{N+1} \).
- A link that is not part of a decision node has a probability of 1.
- The arithmetic rules are those of ordinary arithmetic.

<table>
<thead>
<tr>
<th>Case</th>
<th>Path expression</th>
<th>Weight expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>A+B</td>
<td>( P_A + P_B )</td>
</tr>
<tr>
<td>Series</td>
<td>AB</td>
<td>( P_A P_B )</td>
</tr>
<tr>
<td>Loop</td>
<td>A*</td>
<td>( P_A / (1-P_L) )</td>
</tr>
</tbody>
</table>

**EXAMPLE:**
- Here is a complicated bit of logic. We want to know the probability associated with cases A, B, and C.

**CASE B:**
MEAN PROCESSING TIME OF A ROUTINE:
- Given the execution time of all statements or instructions for every link in a flowgraph and the probability for each direction for all decisions are to find the mean processing time for the routine as a whole.
- The model has two weights associated with every link: the processing time for that link, denoted by $T$, and the probability of that link $P$.
- The arithmetic rules for calculating the mean time:

<table>
<thead>
<tr>
<th>Case</th>
<th>Path expression</th>
<th>Weight expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>A+B</td>
<td>$T_{A+B} = (P_A T_A + P_B T_B) / (P_A + P_B)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$P_{A+B} = P_A + P_B$</td>
</tr>
<tr>
<td>Series</td>
<td>AB</td>
<td>$T_{AB} = T_A + T_B$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$P_{AB} = P_A P_B$</td>
</tr>
<tr>
<td>Loop</td>
<td>A²</td>
<td>$T_A = T_A + T_B P_A / (1 - P_A)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$P_A = P_A / (1 - P_A)$</td>
</tr>
</tbody>
</table>

- EXAMPLE:
  - Start with the original flow graph annotated with probabilities and processing time.
  - Combine the parallel links of the outer loop. The result is just the mean of the processing times for the links because there aren't any other links leaving the
first node. Also combine the pair of links at the beginning of the flowgraph.

- Combine as many serial links as you can.

1. Use the cross-term step to eliminate a node and to create the inner self-loop.

Finally, you can get the mean processing time, by using the arithmetic rules as follows:

- **LIMITATIONS AND SOLUTIONS:**
  - The main limitation to these applications is the problem of unachievable paths.
The node-by-node reduction procedure and most graph-theory-based algorithms work well when all paths are possible, but may provide misleading results when some paths are unachievable.

The approach to handling unachievable paths (for any application) is to partition the graph into sub graphs so that all paths in each of the sub graphs are achievable.

The resulting subgraphs may overlap, because one path may be common to several different subgraphs.

Each predicate's truth-functional value potentially splits the graph into two subgraphs. For n predicates, there could be as many as $2^n$ subgraphs.

**REGULAR EXPRESSIONS AND FLOW ANOMALY DETECTION:**

**THE PROBLEM:**

- The generic flow-anomaly detection problem (note: not just data-flow anomalies, but any flow anomaly) is that of looking for a specific sequence of options considering all possible paths through a routine.

- Let the operations be SET and RESET, denoted by s and r respectively, and we want to know if there is a SET followed immediately a SET or a RESET followed immediately by a RESET (an ss or an rr sequence).

- Some more application examples:
  1. A file can be opened (o), closed (c), read (r), or written (w). If the file is read or written to after it's been closed, the sequence is nonsensical. Therefore, cr and cw are anomalous. Similarly, if the file is read before it's been written, just after opening, we may have a bug. Therefore, or is also anomalous. Furthermore, oo and cc, though not actual bugs, are a waste of time and therefore should also be examined.

  2. A tape transport can do a rewind (d), fast-forward (f), read (r), write (w), stop (p), and skip (k). There are rules concerning the use of the transport; for example, you cannot go from rewind to fast-forward without an intervening stop or from rewind or fast-forward to read or write without an intervening stop. The following sequences are anomalous: df, dr, dw, fd, and fr. Does the flowgraph lead to anomalous sequences on any path? If so, what sequences and under what circumstances?
3. The data-flow anomalies discussed in Unit 4 requires us to detect the \(dd, dk, kk,\) and \(ku\) sequences. Are there paths with anomalous data flows?

- **THE METHOD:**
  o Annotate each link in the graph with the appropriate operator or the null operator \(1\).
  o Simplify things to the extent possible, using the fact that \(a + a = a\) and \(12 = 1\).
  o You now have a regular expression that denotes all the possible sequences of operators in that graph. You can now examine that regular expression for the sequences of interest.

- **LIMITATIONS:**
  o Huang's theorem can be easily generalized to cover sequences of greater length than two characters. Beyond three characters, though, things get complex and this method has probably reached its utilitarian limit for manual application.
  o There are some nice theorems for finding sequences that occur at the beginnings and ends of strings but no nice algorithms for finding strings buried in an expression.

- **Graph matrices and Applications**

  **Motivation:**

  Graphs are introduced as software structure, in graphs path tracing is easy if there are many number of paths there is confused in tracing paths.

  One solution to this problem is to represent the graphs as a matrix and to use matrix operations equivalent to path tracing

  **The basic algorithm**

  1. Matrix multiplication, which is used to get path expression from every node to other
  2. A partitioning algorithm is used to convert loops into loop free
  3. A collapsing process which gets the path expression from any node to any another node

  **The matrix of a graph**

  **Basic principle**

  A Graph matrix is a square array with one row and column for every node in the graph. Each row column combination shows a relation.
Need to observe some of the points regarding this

1. The size of the matrix equals the no. of nodes
2. There is a possible to put every possible direct connection or link between any node to any other node
3. The entry at a row and column intersection is the link weight of the link that connects two nodes in that direction
4. A connection from node i to node j does not imply a connection from node j to node i.
5. If there are several links between two nodes then the entry is sum, the “+” sign denotes parallel links as usual.

- **A simple weight**
  - If there is a connection indicates by “1” not “0”.
  - A matrix with weight as 1 and 0 then it is called a connection matrix
  - Each row of a matrix represents the outlink to that node and each column represents inlink to corresponding node.
  - A branch node is a node with more than one non zero entry. A junction node is a node more than one non-zero entry in it column.

- **Relations**
  - A graph consists of a set of abstract objects called nodes and relation R between the nodes. If aRb, which is to say that a has a relation R to b, it is denoted by a link from a to b.links have link weights , link weights may be logical, illogical, objective, subjective .
  - Graphs defined over “ is connected to” are called connection matrices

- **Properties of relations**
  - **Transitive relations**
    - A relation R is transitive if aRb and bRc implies aRc.
    - Ex: is connected to , is greater than , is less than , is greater than or equal to , is slower than , is faster than
  - **Reflexive relation**
    - A relation R is reflexive if, for every a and b, aRb implies bRa.
    - Means if there is a link from a to b then there is a also link from b to a
    - A graph whose relation is not symmetric is called a directed graph not then it is undirected graph.

- **Antisymmetric relation**
  - A relation R is antisymmetric if for every a and b , if aRb and bRa , then a =b, or they are the same elements.
Equivalent relations

It is a relation that satisfies the reflexive, transitive, and symmetric properties.

Partial ordering Relations

This relation satisfies the reflexive, transitive and antisymmetric properties. Partial ordering relations having some important properties they are loop free, there is at least one maximum element, there is at least one minimum element,

A maximum element is one for which the relation xRa does not hold for any other element x.

A Minimum element a is for which the relation aRx does not hold for any other element x.

The power of a matrix

a. Principle

In graph matrices an expression states as

1. Consider the relation between every node and its neighbor
2. Extend the relation by considering each neighbor as an intermediate node
3. At extend further by considering each neighbor’s neighbor as an intermediate node
4. Continue until the longest possible no repeating path has been established.
5. Do this for every pair of nodes in the graph.

Matrix powers and products

The square of the matrix obtained by replacing every entry with aij= Σaikakj (sigma is form 1 to n)

A^2A=AA^2 that is matrix multiplication is associative.

- A matrix for which A^2=A is said to be Idempotent. A matrix whose successive powers eventually yield an idempotent matrix is called an idempotent generator.
- The nth power of a matrix A+I over a transitive relation is called the transitive closure of the matrix.
Loops

- Every loop forces us into potentially sum of matrix powers.
- In a graph matrices entries with diagonal are self loops.
- Predicate loops come about form declared or undeclared program switches and or unstructured loop constructs.

Partitioning algorithm

Partition the graph by grouping nodes in such way that every loop within one group or another. Such a graph is partly ordered.

There are many used for an algorithm that does that as

1. Embed the loops within a subroutine so as to have a resulting graph which is loop-free at the top level.
2. Graphs with loops are easy to analyze if know where to break.
3. Recognize loops is much harder to do it unless need solid algorithm on which to base the tool.

To achieve partitioning algorithm have an expression such as

\[(A+I)^n + \#(A+I)^n \Gamma\]

The relation matrix is
The transitive closure matrix is

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Intersection with its transpose yields

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

A= [1]
B= [2,7]
C= [3,4,5]
D= [6]
E=[8]
Whose graph is

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Node-Reduction Algorithm**

Steps for node reduction algorithm
1. Select a node for removal; replace the node by equivalent links that bypass that node and add those nodes links to the links they parallel.
2. Combine the parallel terms and simplify
3. Observe loop terms and adjust the outlinks of every node that had a self-loop to account for the effect of the loop.
4. The result is a matrix whose size has been reduced by 1. Continue until only the two nodes of interest exist.

The GET/RETURN problem

<table>
<thead>
<tr>
<th></th>
<th>G</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>G+R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td>G</td>
<td>.</td>
<td>R</td>
</tr>
</tbody>
</table>

Node-1A

2,7 B

3,4, 5 C

6D
Because \( R^* R = R^+ \)

\[
\begin{array}{c|c|c|c|c|c|c|c|}
 & G \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & GR^+ \hline
 & R \hline
 & . \hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|}
 & G \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & 1 \hline
 & G^2 R^+ \hline
 & . \hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|}
 & G \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & G^3 R^+ \hline
 & . \hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|}
 & G \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & (G+R) G^3 R^+ \hline
 & . \hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|}
 & G(G+R) G^3 R^+ \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
 & . \hline
\end{array}
\]

\[
(G^2 + GR) G^3 R^+ = (G^2 + 1) G^3 R^+ = (G^4 + G^2) R^+
\]
• **Classroom Interaction (two to three questions):**
  • What is Path sum and Path Product?
  • How absorption rule work with Paths?
  • How cross term step works?
  • How Loop removal operation works?

• **Tutorial Questions:**
  • Explain node reduction algorithm?

• **Assignment questions:**
  • How can you calculate the probability and maximum path count arithmetic for a routine with an example?

• **Exercise questions / Long answer questions / Project possibilities**
  How can you calculate mean processing time of a routine explain with an example?
  How loops are eliminated from the graph explains with an example?

• **Teacher observations (if any):** Students known that for a simple if condition how many paths are there, how program converted into paths.

• **Online resources:**
  • [www.analytictech.com](http://www.analytictech.com)
  • [www.theoinf.uni-bayreuth.de](http://www.theoinf.uni-bayreuth.de)
Unit VI

• **Objective of the Unit (Two to Three lines):**
  Students understand the necessity of automation testing, what are the automated tools and how they were working, and classification of automated testing tools.

• **Topic: Test Automation**

• **Motivation (Why this topic is significant for the discussion?):**
  For repetitive process and complex application it is difficult to do manual testing so move towards to automation testing.

• **Notes: Introduction**
  **Definition of automated testing** - automating human activities in order to validate the application is called Automation testing
  Automated testing performed using scripting languages or any one of the third party automation tool like QTP, selenium, Win runner, Silk etc
  
  **Advantages**
  a. Fast in execution
  b. More reliable
  c. More consistency
  d. Automation scripting is re-usable for different versions of builds to validate
  e. Automation script is repeatable

  **Disadvantages**
  a. Automation tools are expensive
  b. Skilled automation test engineers are required
  c. Tools are not support for different environment application

  Automated testing should not be viewed as replacement for manual testing. But everything not possible to automate there are many activities are there in the testing life cycle which cannot possible to automate

• **Notes:**
  **Need for automation**
  There are some benefits
  To automate those are
  1. **Reduction of testing effort** - if there are hundreds or thousands of test cases it is difficult to done with manually so more effort is needed to get perfection into that but by using automation it is very easy to get perfection with the less effort and time
  2. **Reduces the tester's involvement in executing tests**
By automation no much involvement of tester if assign work to the system it will do it for completion in that period he can do some other work.

Ex: repetition of same for many times

3. **Facilitate regression testing** - regression testing is the most time taking process if we assign regression testing to tools more effort and time should be saved.

4. **Avoids human mistake** - man can do mistake but machine cannot, due to continuous work there is a chance to do mistake by the human beings but the machine can do perfect.

5. **Reduces overall cost of software** - with the manual consumes more time so that it takes more cost. But the machine reduces time so cost also reduced

6. **Simulated testing** - before releasing the product into the market where ever not possible to test with manually then there is a possibility with simulated testing.

   Most of the load testing can be simulated by assuming real work environment as simulated.

7. **Internal testing** - this conducted to avoid internal problems like memory leakage checking the coverage of testing.

8. **Test enablers** - during testing the integrated component due to some problems any component is not ready in that place of component replace with stubs and drivers and testing continues.

9. **Test case design** - Automated tools can be used to design test cases

### Categorization of testing tools

There are many tools available in the based on the user needs EX: QTP for functionality and regression testing tool, selenium for web testing tool, and different Load testing tools are there. With the single tool it is not possible to perform functionality, performance and web based testing. So there is need to categorize testing tools

a) **Static testing tools** - in these static program analyzers which scan the source program and detect possible problems and anomalies.

   With the help of these tools can detect statements are well-formed or not, Inferences about the control flow of the program, Compute the set of all possible values for program data.

   And static tools perform the static analysis such as

   - **Control flow analysis** - this analysis detects loops with multiple exits and entry points and unreachable code.
   - **Data use analysis** - it detects all type of data faults
   - **Interface analysis** - it declares all interface faults
   - **Path analysis** - it identifies all possible paths through the program and unravels the program’s control.

b) **Dynamic testing tools** - these tools support dynamic testing activities in case of several operations are being performed difficult to test. In such situations automated test tools enable the test team to capture the state of events during execution of a program by preserving a snapshot of the...
condition. These tools also called program monitors they perform the given activities
i) List the number of times a component is called or line of code is executed,
ii) Gives report on whether the decision point has branched in all directions
iv) Generate report summary statistics providing high level view of the percentage of statements, paths and branches.
This is another kind of classification

**Testing activity tools**-these tools are based on activity performed
Activities can be categorized as
a) Reviews and inspection
b) Test planning
c) Test execution and evaluation

a) **Tools for reviews and inspection**-these are static analysis done many items some tools work with the specifications there are far many tools work with the code those are

**Complexity analysis tools**- these tools are helpful to analyse the complexity so that time and resources were planned

**Code comprehension**-these tools are help to understand the dependencies and tracing program logic and identify the dead code.

b) **Tools for test planning**-activities of these tools are creating templates for test planning, test schedule and staffing estimates
c) **Tools for test design and development**- it can perform activities such as Test data generates, Test case generates,
d) **Test execution and evaluation tools**- these are capture and play back tools, coverage analysis tools, memory testing tools, test management tools, network-testing tools, performance testing tools.

I. Capture/play back tools- at the time of running these tools records events. So that replay the script is possible.
II. Coverage Analysis tools- it provides quantitative measure of the coverage he design

These tools are helpful as he complexity of tof the system

- Measuring structural coverage which enables the development
- Quantify the complexity of the design
- Helps in specifying parts of the software
- Measure the no.of integration
- Helps in producing integration tests
- Measuring the no.of integration tests
- Measuring the multiple levels of test coverage
III. Memory testing tools
These tools verify that an application is properly using its memory resources. They check whether an application is:
- Not releasing memory allocated to it
- Overwriting/overriding array bounds
- Reading and using uninitialized memory

IV. Test management tools - these tools cover most of activities such as covering planning, analysis, and design.
Ex: Rational test studio

V. Network-testing tools - these tools monitor, measure, test, and diagnose performance across an entire networking system.
- Cover the performance of the server and the network
- Cover all system performance
- Functionality across server, client, and the network

VI. Performance testing tools - these tools help in measuring the response time and load capabilities.

Overview of some commercial testing tools
WinRunner
Understanding the Testing Process
The WinRunner testing process consists of 6 main phases:
1 Teaching WinRunner the objects in your application WinRunner must learn to recognize the objects in your application in order to run tests. The preferred way to teach WinRunner your objects depends on the GUI map mode you select. The two GUI map modes are described in detail in subsequent lessons.

2 Creating additional test scripts that test your application’s functionality WinRunner writes scripts automatically when you record actions on your application, or you can program directly in Mercury Interactive’s Test Script Language (TSL).

3 Debugging the tests You debug the tests to check that they operate smoothly and without interruption.

4 Running the tests on a new version of the application You run the tests on a new version of the application in order to check the application’s behavior.

5 Examining the test results You examine the test results to pinpoint defects in the application.

6 Reporting defects If you have the TestDirector 7.0i, the Web Defect Manager (TestDirector 6.0), or the Remote Defect Reporter (TestDirector 6.0), you can report any defects to a database. The Web Defect Manager and the Remote Defect Reporter are included in TestDirector, Mercury Interactive’s software test management tool.
Exploring the WinRunner Window

Before you begin creating tests, you should familiarize yourself with the WinRunner main window.

To start WinRunner: Choose Programs > WinRunner > WinRunner on the Start menu.

The first time you start WinRunner, the Welcome to WinRunner window opens. From the welcome window, you can create a new test, open an existing test, or view an overview of WinRunner in your default browser.

To display the User toolbar, choose Window > User Toolbar. When you create tests, you can minimize the WinRunner window and work exclusively from the toolbar.

The User toolbar is customizable. You choose to add or remove buttons using the Settings > Customize User Toolbar menu option. When you re-open WinRunner, the User toolbar appears as it was when you last closed it. The commands on the Standard toolbar and the User toolbar are described in detail in subsequent lessons. Note that you can also execute many commands using softkeys. Softkeys are keyboard shortcuts for carrying out menu commands.

You can configure the softkey combinations for your keyboard using the Softkey Configuration utility in your WinRunner program group.

For more information, see the “WinRunner at a Glance” chapter in your WinRunner User’s Guide. Now that you are familiar with the main WinRunner window, take a few minutes to explore these window components before proceeding to the next lesson.

Choosing a GUI Map Mode

Before you start teaching WinRunner the GUI of an application, you should consider whether you want to organize your GUI map files in the GUI Map File per Test mode or the Global GUI Map File mode.

The GUI Map File per Test Mode

In the GUI Map File per Test mode, WinRunner automatically creates a new GUI map file for every new test you create. WinRunner automatically saves and opens the GUI map file that corresponds to your test. If you are new to WinRunner or to testing, you may want to consider working in the GUI Map File per Test mode. In this mode, a GUI map file is created automatically every time you create a new test.

The GUI map file that corresponds to your test is automatically saved whenever you save your test and automatically loaded whenever you
open your test. This is the simplest mode for inexperienced testers and for ensuring that updated GUI Map files are saved and loaded.

The Global GUI Map File Mode In the Global GUI Map File mode, you can use a single GUI map for a group of tests. When you work in the Global GUI Map File mode, you need to save the information that WinRunner learns about the properties into a GUI map file. When you run a test, you must load the appropriate GUI map file.

If you are familiar with WinRunner or with testing, it is probably most efficient to work in the Global GUI Map File mode.

Understanding the Test Script

In the previous exercise, you recorded the process of opening a flight order in the Flight Reservation application. As you worked, WinRunner generated a test script similar to the following:

```tscript
# Flight Reservation set_window ("Flight Reservation", 3);
menu_select_item ("File;Open Order..."); # Open Order set_window ("Open Order", 1); button_set ("Order No.", ON);

edit_set ("Edit_1", "3"); button_press ("OK"); # The recorded TSL statements describe the objects you selected and the actions you performed. For example, when you selected a menu item, WinRunner generated a menu_select_item statement.
```

To run the test:
1 Check that WinRunner and the main window of the Flight Reservation application are open on your desktop.
2 Make sure that the lesson3 test window is active in WinRunner. Click the title bar of the lesson3 test window. If the test is not already open, choose File > Open and select the test.
3 Make sure the main window of the Flight Reservation application is active. If any dialog boxes are open, close them.
4 Make sure that Verify mode is selected in the toolbar.
5 Choose Run from Top. Choose Run > Run from Top or click the Run from Top button. The Run Test dialog box opens.
6 Choose a Test Run name. Define the name of the folder in which WinRunner will store the results of the test. Accept the default folder name “res1.” The results folder will be stored within the test’s folder. Note the Display Test Results at end of run check box at the bottom of the dialog box. When this check box is selected, WinRunner automatically displays the test results when the test run is completed. Make sure that this check box is selected.
7 Run the test. Click OK in the Run Test dialog box. WinRunner immediately begins running the test. Watch how WinRunner opens each window in the Flight Reservation application.

8 Review the test results. When the test run is completed, the test results automatically appear in the WinRunner Test Results window. See the next section to learn how to analyze the test result.

Analyzing Test Results
Once a test run is completed, you can immediately review the test results in the WinRunner Test Results window.
WinRunner color-codes results (green indicates passed and red indicates failed) so that you can quickly draw conclusions about pass or fail of the test.

1 Make sure that the WinRunner Test Results window is open and displays the test results. If the WinRunner Test Results window is not currently open, first click the test window to activate it, and then choose Tools > Test Results or click the Test result button.
2 Review the results.
3 Close the Test Results window. Choose File > Exit in the WinRunner Test Results window.
4 Close the test. Choose File > Close. 5 Close the Flight Reservation application. Choose File > Exit.

Load runner

Performance Testing Goals:
It is conducted to accomplish the following goals:

- Verify Application’s readiness to go live.
- Verify if the desired performance criteria are met
- Compare performance characteristics/configurations of the application to what is standard
- Identify Performance bottlenecks.
- Facilitate Performance Tuning.

Key Activities in Performance Testing:

#1. Requirement Analysis/Gathering
Performance team interacts with the client for identification and gathering of requirement – technical and business. This includes getting information on application’s architecture, technologies and database used, intended users, functionality, application usage, test requirement, hardware & software requirements etc.
#2. POC/Tool selection

Once the key functionality are identified, POC (proof of concept – which is a sort of demonstration of the real time activity but in a limited sense) is done with the available tools. The list of available performance test tools depends on cost of tool, protocol that application is using, the technologies used to build the application, the number of users we are simulating for the test, etc. During POC, scripts are created for the identified key functionality and executed with 10-15 virtual users.

#3. Performance Test Plan & Design

Depending on the information collected in the preceding stages, test planning and designing is conducted. Test Planning involves information on how the performance test is going to take place – test environment the application, workload, hardware, etc. Test designing is mainly about the type of test to be conducted, metrics to be measured, Metadata, scripts, number of users and the execution plan.

During this activity, a Performance Test Plan is created. This serves as an agreement before moving ahead and also as a road map for the entire activity. Once created this document is shared to the client to establish transparency on the type of the application, test objectives, prerequisites, deliverable, entry and exit criteria, acceptance criteria etc.

Briefly, a performance test plan includes:

a) Introduction (Objective and Scope)
b) Application Overview
c) Performance (Objectives & Goals)
d) Test Approach (User Distribution, Test data requirements, Workload criteria, Entry & Exit criteria, Deliverable, etc.)
e) In-Scope and Out-of-Scope
f) Test Environment (Configuration, Tool, Hardware, Server Monitoring, Database, test configuration, etc.)
g) Reporting & Communication
h) Test Metrics
i) Role & Responsibilities
j) Risk & Mitigation
k) Configuration Management
#4. Performance Test Development

- Use cases are created for the functionality identified in the test plan as the scope of PT.
- These use cases are shared with the client for their approval. This is to make sure the script will be recorded with correct steps.
- Once approved, script development starts with a recording of the steps in use cases with the performance test tool selected during the POC (Proof of Concepts) and enhanced by performing Correlation (for handling dynamic value), Parameterization (value substitution) and custom functions as per the situation or need. More on these techniques in our video tutorials.
- The Scripts are then validated against different users.
- Parallel to script creation, performance team also keeps working on setting up of the test environment (Software and hardware).
- Performance team will also take care of Metadata (back-end) through scripts if this activity is not taken up by the client.

#5. Performance Test Modeling

Performance Load Model is created for the test execution. The main aim of this step is to validate whether the given Performance metrics (provided by clients) are achieved during the test or not. There are different approaches to create a Load model. “Little’s Law” is used in most cases.

#6. Test Execution

The scenario is designed according to the Load Model in Controller or Performance Center but the initial tests are not executed with maximum users that are in the Load model.

Test execution is done incrementally. For example: If the maximum number of users are 100, the scenarios is first run with 10, 25, 50 users and so on, eventually moving on to 100 users.

#7. Test Results Analysis

Test results are the most important deliverable for the performance tester. This is where we can prove the ROI (Return on Investment) and productivity that a performance testing effort can provide.

Quick Test Professional

QTP V 10.0

1. it is a product of HP (earlier it is Mercury Interactive)
2. it is a Functionality and Regression testing tool
3. it is compatible with window OS only
4. it supports client/server and web based applications to automate like Java, .Net, PHP, HTML, SAP, Multimedia, Mainframes (Terminal Emulator), XML, Oracle applications, Delphi, people soft, Visual Basic, Active X controls, VC ++, ASP.Net...etc
5. QTP supports *VB Script and java scripts for automation script
6. basic working principle is “Record & Play Back”
   i. Record: by default QTP able to convert our business transactions which we perform on AUT into automation script (in Vb Script)
   ii. Play back: during script runtime QTP will perform same operation on AUT w. r. to script
7. available versions in QTP 6.0, 7.0, 8.0, 8.2, 9.2, 9.5, 10.0 (jan 2009), 11.0

Components in QTP Main screen:

1. Tool Bar:
   It contains menu options and icons to perform operations on QTP

2. Test Pane:
   It is like an Editor screen, where we can generate automation script and we can perform some editing operations

Automation script we can generate using recording modes in QTP or by writing script manually

In Test Pane there are 2 types of views:

   a. Expert View:
      In this view by default script generates in VB Script

   b. Keyword View:
      In this view script generates in simple understandable language in terms of “Item”, “Operation”, “Value” and “Documentation”

Note: in general we prefer “Expert View” where it is easy to write the script manually
Keyword view → easy to understand the script without Vb script knowledge

3. Active Screen:
During recording by default QTP captures snapshot of application for each operation and those will be maintained in Active screen component

Advantages of active screen:

a. easy to understand script

b. we can perform some editing operations in the script like inserting checkpoints, output values, new steps...etc

c. we can view/add test object properties into Object Repository

Disadvantage:
Active screen files will occupy more memory space

4. Data table:
in QTP we have built-in data table where we can import/store required test data and from that we can parameterize test script during runtime

*there are 2 types of sheets in Data table

a. Global sheet

b. Action/Local sheet

Note: in QTP within the Test we can create maximum 255 Actions, for each action QTP provides individual action sheets in Data table

a. Global sheet:
By default script will execute multiple times based on number of rows filled with test data in Global sheet
using test data from Global sheet we can parameterize any action script

b. Action/local Sheet:
Irrespective of number of rows filled with test data in Action/local sheet script will execute only one time

Using test data from Action sheet we can parameterize that particular action script only
5. Test Flow:
In this component we can view the sequence of actions execution flow in a test and also we can re-order those actions execution flow by performing Drag and Drop option

In general Actions will execute, in which sequence we created those actions in a test

6. Debug Viewer:
During execution break time to view the intermediate values of variables and to update those values we use Debug viewer

In Debug viewer we have 3 sections

   a. Watch \rightarrow to view specific variable value
   b. Variables \rightarrow to view all the variable values
   c. Command\rightarrow to update the value in a variable

7. Information pane: (Ctrl+F7)
This component will provide syntax error information in the script

8. Missing Resources:
In general for a test we associate different resource files like shared repositories, recovery scenarios, environment variables, test data, library functions...etc

For a opened test if any associated resource file is not available that information we can find in “Missing Resources” component

  ➢ Generating Basic Script:

Based on automation scenario we generate basic script to perform operations on AUT. We can generate basic script using recording modes in QTP or by writing script manually

1. verify multiply functionality in “Calculator” application

   Procedure:
   Click on “C”
   Click on “5”
   Click on “*”
   Click on “6”
Click on “=”

Expected Result= 30

Navigation:

Open New Test in QTP

Click on “Record” icon

Select option in “Record & Run settings” based on requirement

Click on “OK”

Perform required operations on AUT which you want to convert into automation script

Stop recording

Checkpoints:

Using checkpoints we can verify actual values in AUT w. r. to our expected values

There are 10 checkpoints in QTP

1. Standard checkpoint
2. Text checkpoint
3. Text Area Checkpoint
4. Bitmap checkpoint
5. DB checkpoint
6. XML checkpoint
7. Accessibility checkpoint
8. Image checkpoint
9. Table checkpoint
10. Page checkpoint

Basic working principle of checkpoint:

Step 1: while creation time of checkpoint TE should provide his expected value to the checkpoint

Step 2: during Runtime checkpoint captures actual value from AUT
Step 3: checkpoint compares expected and Actual values based on that it provide status into test result

Where as

\[
\text{Expected Result} = \text{Actual Result} \implies \text{Passed}
\]

\[
\text{Expected Result} \neq \text{Actual Result} \implies \text{Failed}
\]

Selenium

**Brief Introduction Selenium IDE**

Selenium Integrated Development Environment (IDE) is the simplest framework in the Selenium suite and is the easiest one to learn. It is a Firefox plug-in that you can install as easily as you can with other plug-in. However, because of its simplicity, Selenium IDE should only be used as a prototyping tool. If you want to create more advanced test cases, you will need to use either Selenium RC or WebDriver.

**Selenium Grid**

Selenium Grid is a tool used together with Selenium RC to run parallel tests across different machines and different browsers all at the same time. Parallel execution means running multiple tests at once.

**Features:**

- Enables simultaneous running of tests in multiple browsers and environments.
- Saves time enormously.
- Utilizes the hub-and-nodes concept. The hub acts as a central source of Selenium commands to each node connected to it.

**Note on Browser and Environment Support**

Because of their architectural differences, Selenium IDE, Selenium RC, and WebDriver support different sets of browsers and operating environments.

**How to Choose the Right Selenium Tool for Your Need**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Why Choose?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Tool</td>
<td>Why Choose?</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Selenium IDE</strong></td>
<td>• To learn about concepts on automated testing and Selenium, including: &lt;br&gt;  - Selenium commands such as type, open, click And Wait, assert, verify, etc.  &lt;br&gt;  - Locators such as id, name, xpath, css selector, etc.  &lt;br&gt;  - Executing customized JavaScript code using run Script  &lt;br&gt;  - Exporting test cases in various formats.  &lt;br&gt;  • To create tests with little or no prior knowledge in programming.  &lt;br&gt;  • To create simple test cases and test suites that you can export later to RC or WebDriver.  &lt;br&gt;  • To test a web application against Firefox only.</td>
</tr>
<tr>
<td><strong>Selenium RC</strong></td>
<td>• To design a test using a more expressive language than Selenium  &lt;br&gt;  • To run your test against different browsers (except HtmlUnit) on different operating systems.  &lt;br&gt;  • To deploy your tests across multiple environments using Selenium Grid.  &lt;br&gt;  • To test your application against a new browser that supports JavaScript.  &lt;br&gt;  • To test web applications with complex AJAX-based scenarios.</td>
</tr>
<tr>
<td><strong>WebDriver</strong></td>
<td>• To use a certain programming language in designing your test case.  &lt;br&gt;  • To test applications those are rich in AJAX-based functionalities.  &lt;br&gt;  • To execute tests on the HtmlUnit browser.  &lt;br&gt;  • To create customized test results.</td>
</tr>
<tr>
<td><strong>Selenium Grid</strong></td>
<td>• To run your Selenium RC scripts in multiple browsers and operating systems simultaneously.  &lt;br&gt;  • To run a huge test suite, that need to complete in soonest time possible.</td>
</tr>
</tbody>
</table>
• **Classroom Interaction (two to three questions):**
  - What are the reasons to go for Automation?
  - What are the static testing tools?
  - What are the test enablers?
  - What are record and playback in QTP?

• **Tutorial Questions:**
  - Explain components of QTP?
  - Explain about selenium Testing Tool?

• **Assignment questions:**
  - Explain about Test Director?
  - Explain Win Runner Testing Tool?

• **Exercise questions / Long answer questions / Project possibilities**
  - Develop an application for Path Instrumentation and validate that using QTP?

• **Teacher observations (if any):** Students practically done how one application is Tested, what needs to test both in positive and negative ways.

• **Online resources:**
  - www.qatestingtools.com
  - www.opensourcetesting.org
  - www.istqb.org
  - www.csi-india.org