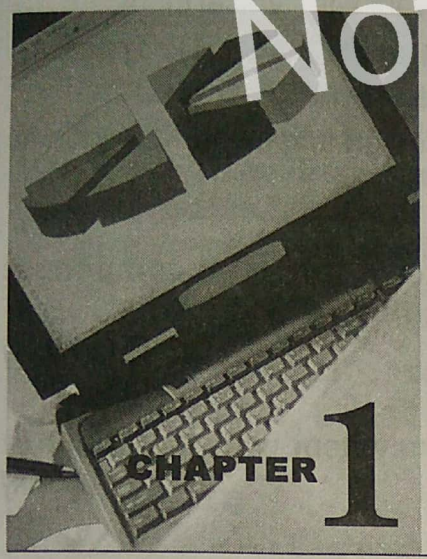


CHAPTER 1

INTRODUCTION TO SOFTWARE TESTING

1.0. INTRODUCTION

Testing is the process of executing the program with the intent of finding faults. Who should do this testing and when should it start are very important questions that are answered in this text. As we know that software testing is the fourth phase of Software Development Life Cycle (SDLC). About 70% of development time is spent on testing. We explore this and many other interesting concepts in this chapter.

INSIDE THIS CHAPTER

1.1. THE TESTING PROCESS

Testing is different from debugging. Removing errors from your programs is known as **debugging** but testing aims to locate an as yet undiscovered errors. We test our program with both valid and invalid inputs and then compare our expected outputs as well as the observed outputs (after execution of software). **Please note that testing starts from requirements analysis phase only and goes till the last maintenance phase.** During requirement analysis and designing we do **static testing** wherein the SRS is tested to check whether it is as per user requirements or not. We use techniques of code reviews, code inspections, walkthroughs and Software Technical Review (STRs) to do static testing. **Dynamic testing** starts when the code is ready or even a unit (or module) is ready. It is dynamic testing as now the code is tested. We use various techniques for dynamic testing like Black-box, Gray-box and white-box testing. We shall be studying these in the subsequent chapters.

1.2. What is Software Testing ?

The concept of software testing has evolved from simple program "check-out" to a broad set of activities that cover the entire software life-cycle.

There are five distinct levels of testing that are given below :

(a) **Debug :** It is defined as the successful correction of a failure.

(b) **Demonstrate :** The process of showing that major features work with typical input.

(c) **Verify :** The process of finding as many faults in the application under test (AUT) as possible.

(d) **Validate :** The process of finding as many faults in requirements, design and AUT.

(e) **Prevent :** To avoid errors in development of requirements, design and implementation by self-checking techniques, including "test before design".

There are various definitions of testing that are given below :

"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".

[IEEE 83a]

OR

"Software testing is the process of executing a program or system with the intent of finding errors." ***[Myers]***

OR

"It involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results." ***[Hetzel]***

Testing is NOT :

(a) The process of demonstrating that errors are not present.

(b) The process to show that a program perform its intended functions correctly.

(c) The process of establishing confidence that a program does what it is supposed to do.

So, all these definitions are incorrect. Because, with these as guidelines, one would tend to operate the system in a normal manner to see if it works. One would unconsciously choose such normal/correct test data as would prevent the system from failing. Besides, it is anyway not possible to certify that a system has no errors—simply because it is almost impossible to detect all errors.

So, simply stated : **"Testing is basically a task of locating errors."**

It may be :

(a) **Positive Testing :** Operate application it should be operated. Does it behave normally. Use proper variety of legal test data, including data values at the boundaries to test if it fails. Check actual test results with the expected. Are results correct ? Does the application function correctly ?

(b) **Negative Testing :** Test for abnormal operations. Does the system fail/crash ? Test with illegal or abnormal data. Intentionally, attempt to make things go wrong and to discover/detect—"Does the program do what it should not ? Does it fail to do what it should ?"

(c) **Positive View of Negative Testing :** The job of testing is to discover errors before the user does. A good tester is one who is successful in making the system fail. Mentality of the tester has to be destructive—opposite to that of the creator/author, which should be constructive.

One very popular equation of software testing is :

Software Testing = Software Verification + Software Validation

As per IEEE definition(s) :

Software Verification : ***"It is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase."***

OR

"It is the process of evaluating, reviewing, inspecting and doing desk checks of work products such as requirement specifications, design specifications and code".

OR

"It is a human testing activity as it involves looking at the documents on paper."

Whereas **Software Validation** : "It is defined as the **process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements. It involves executing the actual software. It is a computer based testing process."**

Both verification and validation (V&V) are complementary to each other.

As mentioned earlier, good testing expects more than just running a program. We consider a leap-year function working on MS SQL (Server Data Base) :

```
CREATE FUNCTION  f_is_leap_year (@ ai_year small int)
RETURNS small int
AS
BEGIN
    --if year is illegal (null or -ve), return -1
    IF (@ ai_year IS NULL) or
         (@ ai_year <= 0) RETURN -1
    IF ((((@ ai_year % 4) = 0) AND
         ((@ ai_year % 100)< > 0)) OR
         ((@ ai_year % 400) = 0)
         RETURN 1  --leap year
    RETURN 0  --Not a leap year
END
```

We execute above program with number of inputs :

TABLE 1.1. Database Table : Test_leap_year.

Serial No.	Year (year to test)	Expected result	Observed result	Match
1.	–1	–1	–1	Yes
2.	–400	–1	–1	Yes
3.	100	0	0	Yes
4.	1000	0	0	Yes
5.	1800	0	0	Yes
6.	1900	0	0	Yes
7.	2010	0	0	Yes
8.	400	1	1	Yes
9.	1600	1	1	Yes
10.	2000	1	1	Yes
11.	2400	1	1	Yes
12.	4	1	1	Yes
13.	1204	1	1	Yes
14.	1996	1	1	Yes
15.	2004	1	1	Yes

In this database table given above there are 15 test cases. But these are not sufficient as we have not tried with all possible inputs. We have not considered the trouble spots like :

(i) Removing statement (@ ai_year % 400 = 0) would result in Y2K problem.

(ii) Entering year in float format like 2010.11.

(iii) Entering year as a character or as a string.

(iv) Entering year as NULL or zero (0).

This list can grow further also. These are our **trouble spots** or **critical areas.** We wish to locate these areas and fix these problems at our earliest before our customer does.

1.3. WHY SHOULD WE TEST ? WHAT IS THE PURPOSE ?

Testing is necessary. Why is this so ?

1. The Technical Case :

(a) Competent developers are not inflallible.

(b) The implications of requirements are not always forseeable.

(c) The behaviour of a system is not necessarily predictable from its components.

(d) Languages, databases, user interfaces and operating systems have bugs that can cause application failures.

(e) Reusable classes and objects must be trustworthy.

2. The Business Case :

(a) If you don't find bugs, your customers or users will.

(b) Post-release debugging is the most expensive form of development.

(c) Buggy software hurts operations, sales and reputation.

(d) Buggy software can be hazardous to life and property.

3. The Professional Case :

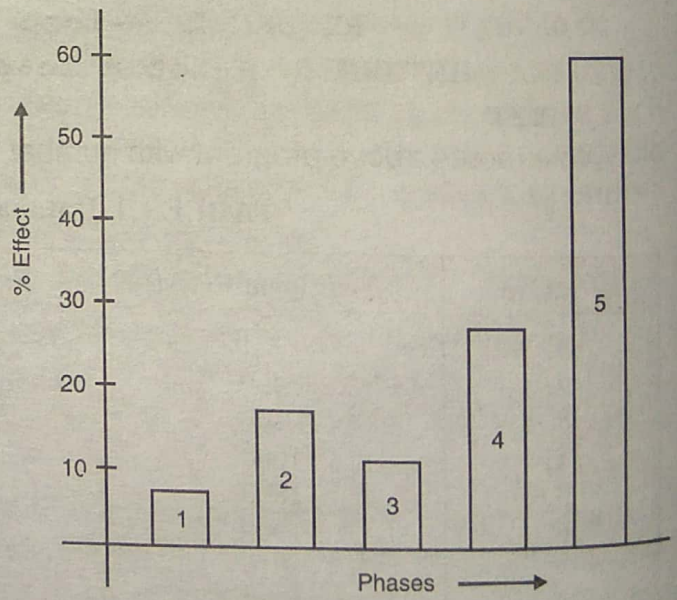
(a) Test case design is a challenging and rewarding task.

(b) Good testing allows confidence in your work.

(c) Systematic testing allows you to be most effective.

(d) Increases your credibility, allows pride your efforts.

4. The Economics Case : Practically speaking, defects get introduced in every phase of SDLC. **Pressman** has described a '**Defect Amplification Model**' wherein he says that errors get amplified by a certain factor if that error is not removed in that phase only. This may increase cost of defect removal. This principle of detecting errors as close to its point of introduction as possible is known as **Phase containment of errors.**



1. Requirements Analysis; 2. Design; 3. Implementation; 4. Testing; 5. Maintenance

Fig. 1.1. Efforts During SDLC

5. To Improve Quality : As computers and software are used in critical applications, the outcome of a bug can be severe. Bugs can cause huge losses. Bugs in critical systems have caused airplane

crashes, allowed space shuttle systems to go away, halted trading on stock market. Bugs can kill. Bugs can cause disasters.

In a computerized embedded world, the quality and reliability of software is a matter of life and death. This can be achieved only if thorough testing is done.

6. For Verification and Validation (V&V) : Testing can serve as metrics. It is heavily used as a tool in the V&V process. We can compare the quality among different products under the same specification, based on results from the same test.

Good testing can provide measures for all relevant quality factors.

7. For Reliability Estimation : Software reliability has important relations with many aspects of software, including the structure and the amount of testing it has been subjected to. Based on an operational profile (an estimate of the relative frequency of use) of various inputs to the program, testing can serve as a statistical sampling method to gain failure data for reliability estimation.

Recent Software Failures

(a) 31st May, 2012, HT reports the failure of air traffic management software-Auto Trac-III at Delhi Airport. The system is unreliable. This ATC software was installed in 2010 as Auto Trac-II (its older version). Since then it has faced many problems due to inadequate testing. Some of the snags were:

1. May 28, 2011, snag hits radar system of ATC.
2. Feb. 22, 2011, ATC goes blind for 10 minutes with no data about the arriving or departing flight.
3. July 28, 2010, radar screens at ATC go blank 25 mins. after the system displaying flight data crashes.
4. Feb. 10, 2010, one of the radar scopes stops working at ATC.
5. Jan. 27, 2010, A screen goes blank at ATC due to a technical glitch.
6. Jan. 15, 2010, radar system collapses due to software glitch. ATC officials manually handles the aircraft.

(b) The case of 2010 Toyota Prins, had a software bug that caused braking problems on bumpy roads.

(c) In another case of Therac-25, 6 cancer patients were given overdose.

(d) A breach on play station network caused a loss of $ 170 mn to Sony Corp.

Why it happened?

As we know that Software Testing constitutes about 40% of overall effort and 25% of the overall software budget. Software defects are introduced during SDLC due to poor quality requirements, design and code. Sometimes due to lack of time and inadequate testing, some of the defects are left behind, only to be found later by users. Software is a ubiquitous product, 90% of the people use software in their everyday life. Software has highest failure rates due to poor qualify of software.

Smaller companies which donot have deep pockets can get wiped out because they did not pay enough attention to software quality and conduct the right amount of testing.

Sometimes due to lack of time and inadequate testing some defects escape out – to be found later by customers. Locating a defect is like finding a pin in a haystack.

1.4. Who Should Do Testing ?

As mentioned earlier also, testing starts right from the very beginning. This implies that testing is everyone's responsibility. By 'everyone', I mean all project team members. So, we cannot rely on one person only. Naturally, it is a **team effort.** We cannot only designate tester responsible. Even developers are responsible. They build the code but do not indicate any errors as they have written their own code.

1.5. How Much Should We Test ?

Consider that there is a while_loop that has three paths. If this loop is executed twice, we have (3 * 3) paths and so on. So, the total number of paths through such a code will be :

$$= 1 + 3 + (3 * 3) + (3 * 3 * 3) + \ldots$$

$$= 1 + \Sigma 3^n \qquad \text{(where } n > 0\text{)}$$

This means an infinite number of test case. Thus, testing is not 100% exhaustive.

1.6. Selection of Good Test Cases

Designing good test case is a complex art. It is complex because :

(a) Different types of test cases are needed for different classes of information.

(b) All test cases within a test suite will not be good. Test cases may be good in variety of ways.

(c) People create test cases according to certain testing styles like domain testing or risk-based testing. And good domain tests are different from good risk-based tests.

Brian Marick coins a new term to a lightly documented test case—**The test idea.** According to Brian, **"A test idea is a brief statement of something that should be tested."** For example, if we are testing a square-root function, one **test idea** would be—**"test a number less than zero."** The idea here is again to check if the code handles an error case.

Cem Kaner said—"The best test cases are the ones that find bugs". Our efforts should be on the test cases that finds issues. Do broad or deep coverage testing on the trouble spots.

A test case is a question that you ask of the program. The point of running the test is to gain information like whether the program will pass or fail the test.

1.7. Measurement of Testing

There is no single scale that is available to measure the testing progress. A good project manager (PM) wants that worse conditions should occur in the very beginning of the project only than in the later phases. If errors are large in numbers, we can say either testing was not done thoroughly or it was done so thoroughly that all errors were covered. So, there is no standard way to measure our testing process. But metrics can be computed at the organizational, process, project and product levels. Each set of these measurements has its value in monitoring, planning and control.

NOTES

Metrics is assisted by four core components—schedule, quality, resources and size.

1.8. Incremental Testing Approach

To be effective, a software tester should be knowledgeable in two key areas :

1. Software testing techniques.
2. The application under test (AUT).

For each new testing assignment, a tester must invest time in learning about the application. A tester with no experience must also learn testing techniques, including general testing concepts and how to define test cases. Our goal is to define a suitable list of tests to perform within a tight deadline. There are 8 stages for this approach :

Stage 1 : Exploration.

Purpose : To gain familiarity with the application.

Stage 2 : Baseline test.

Purpose : To devise and execute a simple test case.

Stage 3 : Trends analysis.

Purpose : To evaluate whether the application performs as expected when actual output cannot be predetermined.

Stage 4 : Inventory.

Purpose : To identify the different categories of data and create a test for each category item.

Stage 5 : Inventory combinations.

Purpose : To combine different input data.

Stage 6 : Push the boundaries.

Purpose : To evaluate application behaviour at data boundaries.

Stage 7 : Devious data.

Purpose : To evaluate system response when specifying bad data.

Stage 8 : Stress the environment.

Purpose : To attempt to break the system.

The schedule is tight, so we may not be able to perform all of the stages. The time permitted by the delivery schedule determines how many stages one person can perform. After executing the baseline test, later stages could be performed in parallel if more testers are available.

1.9. Basic Terminology Related to Software Testing

We must define the following terminologies one by one :

1. Error (or Mistake or Bugs) : People make errors. A good synonym is mistake. **When people make mistakes while coding, we call these mistakes bugs.** Errors tend to propagate. A requirements errors may be magnified during design and still amplified during coding. So, error is mistake during SDLC.

2. Fault (or Defect) : A missing or incorrect statement(s) in a program resulting from an error is a fault. So, fault is the representation of an error. Representation here means the mode of expression, such as a narrative text, data flow diagrams, hierarchy charts etc. Defect is a good synonym for fault. Faults can be elusive. It requires a fix.

3. Failure : A failure occurs when a fault executes. The manifested inability of a system or component to perform a required function within specified limits is known as a failure. A failure is evidenced by incorrect output, abnormal termination, or unmet time and space constraints. It is dynamic process.

So, Error (or mistake or bug) ⟶ Fault (or defect) ⟶ Failure. For example

Error (e.g., * replaced by /) ⟶ Defect (e.g., C = A/B) ⟶ (e.g., C = 2 instead of 8)

4. Incident : When a failure occurs, it may or may not be readily apparent to the user. **An incident is the symptom associated with a failure that alerts the user to the occurrence of a failure.** It is an unexpected occurrence that requires further investigation. It may not need to be fixed.

5. Test : Testing is concerned with errors, faults, failures and incidents. **A test is the act of exercising software with test cases.** A test has two distinct goals—to find failures or to demonstrate correct execution.

6. Test Case : A test case has an identity and is associated with program behaviour. A test case also has a set of inputs and a list of expected outputs. **The essence of software testing is to determine a set of test cases for the item to be tested.**

The test case template is discussed next

```
Test Case ID
Purpose
Preconditions
Inputs
Expected Outputs
Postconditions
Execution History
Date              Result          Version              Run By
```

Fig. 1.2. Test Case Template.

Inputs are of two types :

(a) **Preconditions :** Circumstances that hold prior to test case execution.

(b) **Actual inputs :** That were identified by some testing method.

Expected outputs, are again of two types :

(a) Post conditions (b) Actual outputs.

The act of testing entails establishing the necessary preconditions, providing the test case inputs, observing the outputs and then comparing these with the expected outputs to determine whether the test passed.

The remaining information in a test case primarily supports testing team. Test cases should have an identity and a reason for being. It is also useful to record the execution history of a test case, including when and by whom it was run, the pass/fail result of each execution and the version of the software on which it was run. This makes us clear that test cases are valuable, at least as valuable as source code. Test cases need to be developed, reviewed, used, managed and saved. So, we can say that test cases occupy a central position in testing.

Test cases for ATM Machine:

Preconditions: System is started.

Test case ID	Test case name	Test case descritpion	Test steps			Test status (P/F)	Test priority
			Step	Epected result	Actual result		
Session 01	Verify Card	To verify whether system reads a customer's ATM card.	Insert a readable card	Card is accepted; System asks for entry of PIN			High
			Insert an unreadable card	Card is ejected; System displays an error screen; System is ready to start a new session			High
	Validate PIN	To verify whether system accepts customer's PIN	Enter valid PIN	System displays a menu of transaction types			High
			Enter invalid PIN	Customer is asked to re-enter PIN			High
			Enter incorrect PIN the first time, then correct PIN the second time	System displays a menu of transaction types.			High
			Enter incorrect PIN the first time and second times, then correct PIN the third time	System displays a menu of transaction types.			High
			Enter incorrect PIN three times	An appropriate message is displayed; Card is retained by machine; Session is terminated.			High
Session 02	Validate User session	To verify whether system allows customer to perform a transaction	Perform a transaction	System asks whether customer wants another transaction			High
		To verify whether system allows multiple transactions in one session	When system asks for another transaction, Answer yes	System displays a menu of transaction types			High
			When system asks for another transaction, Answer no	System ejects card and is ready to start a new session			High

(Contd...)

Test case ID	Test case name	Test case descritpion	Test steps			Test status (P/F)	Test priority
			Step	Epected result	Actual result		
	Verify transac-tions	To verify whether the system allows to withdraw.	Choose With-drawal transaction	System displays a menu of possible withdrawal amounts			High
		To verify whether system performs a legal withdrawal transaction properly.	Choose an amount that the system currently has and which is not greater than the account balance	System dispenses this amount of cash; System prints a correct receipt showing amount and correct updated balance.			High
			Choose an amount greater than what the system currently has.	System displays an appropriate message and asks customer to choose a different amount.			High
			Press "cancel" key.	System displays an appropriate message and offers the cus-tomer the option of choosing to operate another transaction.			Medium
Deposit 01	Verify deposit Transaction	To verify whether system allows to deposit.	Choose deposit transaction	System displays a request for the customer to type the amount.			High
		To verify whether the system performs a legal deposit transac-tion properly.	Enter a legitimate amount	System requests that the customer insert an enve-lope.			High
			Insert an envelope.	System accepts envelope; System prints a correct receipt showing the amount and correct updated balance			High
			Press "cancel" key.	System displays an appropriate message and offers the cus-tomer the option of choosing to operate another transaction.			Medium

7. Test Suite : A collection of test scripts or test cases that is used for validating bug fixes (or finding new bugs) within a logical or physical area of a product. For example, an acceptance test suite contains all the test cases that used to verify that software has met certain predefined acceptance criteria.

8. Test Script : The step-by-step instructions that describe how a test case is to be executed. It may contain one or more test cases.

9. Test Ware : It includes all testing documentation created during testing process. For example, test specification, test scripts, test cases, test data, environment specification.

10. Test Oracle : Any means used to predict the outcome of a test.

11. Test Log : A chronological record of all relevant details about the execution of a test.

12. Test Report : A document describing the conduct and results of testing carried out for a system.

1.10. Testing Life Cycle

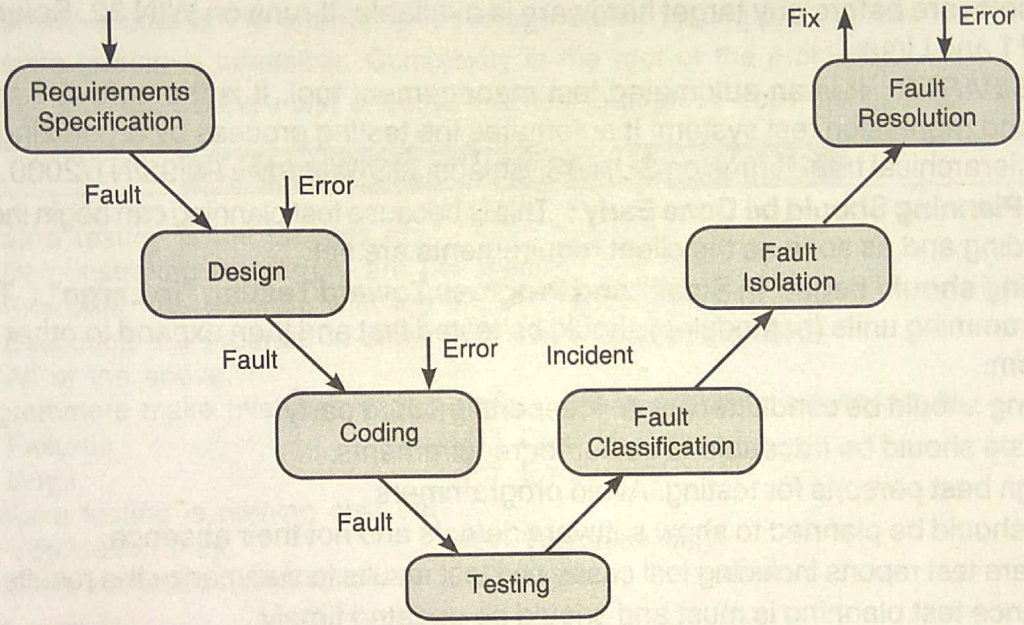


Fig. 1.3. A Testing Life Cycle.

In the development phase, three opportunities arise for errors to be made, resulting in faults that propagate through the remainder of the development process. The first three phases are putting bugs IN, the testing phase is finding bugs and the last three phases are Getting Bugs OUT. The Fault Resolution Step is another opportunity for errors and new faults. When a fix causes formerly correct software to misbehave, the fix is deficient.

1.11. When to Stop Testing ?

Testing is potentially endless. We cannot test till all defects are unearthed and removed. It is simply impossible. At some point, we have to stop testing and ship the software. The question is when ?

Realistically, testing is a trade-off between budget, time and quality. It is driven by profit models.

The **pessimistic approach** to stop testing is whenever some or any of the allocated resources—time, budget or test cases are exhausted.

The **optimistic stopping rule** is to stop testing when either reliability meets the requirement, or the benefit from continuing testing cannot justify the testing cost. ***[Yang]***

1.12. Principles of Testing

To make software testing effective and efficient we follow certain principles. These principles are stated below.

1. **Testing should be Based on User Requirements :** This is in order to uncover any defects that might cause the program or system to fail to meet the client's requirements.
2. **Testing Time and Resources are Limited :** Avoid redundant tests.
3. **Exhaustive Testing is Impossible :** As stated by Myer, it is impossible to test everything due to huge data space and large number of paths that a program flow might take.
4. **Use Effective Resources to Test :** This represents use of the most suitable tools, procedures and individuals to conduct the tests. The test team should use the tools like :
   (a) *Deja Gnu :* It is a testing frame work for interactive or batch oriented applications. It is designed for regression and embedded system testing. It runs on UNIX platform.
   (b) *E-SIM :* It is a native software simulator for embedded software. It is used to test software before any target hardware is available. It runs on WIN 32, Solaris 5, HP-UX 11 and Linux.
   (c) *SMARTS :* It is an automated test management tool. It is the software maintenance and regression test system. It automates the testing process by organizing tests into a hierarchical tree. It runs on SUN 0S, Solaris, MS-Windows 95/98/NT/2000.
5. **Test Planning Should be Done Early :** This is because test planning can begin independently of coding and as soon as the client requirements are set.
6. **Testing should begin "in Small" and Progress Toward Testing "in Large" :** The smallest programming units (or modules) should be tested first and then expand to other parts of the system.
7. Testing should be conducted by an independent third party.
8. All tests should be traceable to customer requirements.
9. Assign best persons for testing. Avoid programmers.
10. Test should be planned to show software defects and not their absence.
11. Prepare test reports including test cases and test results to summarize the results of testing.
12. Advance test planning is must and should be updated timely.

1.13. Limitations of Testing

1. Testing can show presence of errors—not their absence.
2. No matter how hard you try, you would never find the last bug in an application.
3. The domain of possible inputs is too large to test.
4. There are too many possible paths through the program to test.
5. In short, maximum coverage through minimum test-cases. That is the challenge of testing.
6. Various testing techniques are complementary in nature and it is only through their combined use that one can hope to detect most errors.

1.14. Available Testing Tools, Techniques and Metrics

There are an abundance of software testing tools that exist. Some of them are listed below :

(a) **Mothora :** It is an automated mutation testing tool-set developed at Purdue university. Using Mothora, the tester can create and execute test cases, measure test case adequacy, determine input-output correctness, locate and remove faults or bugs and control and document the test.

(b) **NuMega's Bounds Checker, Rational's Purify :** They are run-time checking and debugging aids. They can both check and protect against memory leaks and pointer problems.

(c) **Ballista COTS Software Robustness Testing Harness [Ballista] :** It is a full-scale automated robustness testing tool. It gives quantitative measures of robustness comparisons across operating systems. The goal is to automatically test and harden commercial off-the-shalf (COTS) software against robustness failures.

SUMMARY

1. Software testing is an art. Most of the testing methods and practices are not very different from 20 years ago. It is no where near maturity, although there are many tools and techniques available to use. Good testing also requires a tester's creativity, experience and intuition, together with proper techniques.
2. Testing is more than just debugging. It is not only used to locate errors and correct them. It is also used in validation, verification process and reliability measurement.
3. Testing is expensive Automation is a good way to act down cost and time. Testing efficiency and effectiveness is the criteria for coverage based testing techniques.
4. Complete testing is infeasible. Complexity is the root of the problem.
5. Testing may not be the most effective method to improve software quality.

MULTIPLE CHOICE QUESTIONS

1. Software testing is the process of
   (a) Demonstrating that errors are not present
   (b) Executing the program with the intent of finding errors
   (c) Executing the program to show that it executes as per SRS
   (d) All of the above.
2. Programmers make mistakes during coding. These mistakes are known as
   (a) Failures (b) Defects
   (c) Bugs (d) Errors.
3. Software testing is nothing else but
   (a) Verification only (b) Validation only
   (c) Both verification and validation (d) None of the above.
4. Test suite is
   (a) Set of test cases (b) Set of inputs
   (c) Set of outputs (d) None of the above.
5. Which one is not the verification activity ?
   (a) Reviews (b) Path testing
   (c) Walkthroughs (d) Acceptance testing.
6. A break in the working of a system called
   (a) Defect (b) Failure
   (c) Fault (d) Error.
7. One fault may lead to
   (a) One failure (b) No failure
   (c) Many failures (d) All of the above.
8. Verification is
   (a) Checking product with respect to customer's expectations
   (b) Checking product with respect to SRS
   (c) Checking product with respect to the constraints of the project
   (d) All of the above.

CHAPTER 2

SOFTWARE VERIFICATION AND VALIDATION

2.0. INTRODUCTION

The evolution of software that satisfies its user expectations is a necessary goal of a successful software development organization. To achieve this goal, software engineering practices must be applied throughout the evolution of the software product. Most of these practices attempt to create and modify software in a manner that maximizes the probability of satisfying its user expectations.

INSIDE THIS CHAPTER

2.1. DIFFERENCES BETWEEN VERIFICATION AND VALIDATION

Software verification and validation (V&V) is a technical discipline of systems engineering. According to Stauffer and Fuji (1986), Software V&V is "**a systems engineering process employing a rigorous methodology for evaluating the correctness and quality of software product through the software life cycle.**"

According to Dr. Berry Boehm (1981), Software V&V is performed in parallel with the software development and not at the conclusion of the software development. However, both verification and validation are different. Let us tabulate the differences between them.

TABLE 2.1.

Verification	Validation
1. It is a static process of verifying documents, design and code.	1. It is a dynamic process of validating/testing the actual product.
2. It does not involve executing the code.	2. It involves executing the code.
3. It is human based checking of documents/files.	3. It is computer based execution of program.
4. Target is requirements specification, application architecture, high level and detailed design, database design.	4. Target is actual product—a unit, a module, a set of integrated modules, final product.
5. It uses methods like inspections, walk throughs, Desk-checking etc.	5. It uses methods like black box, gray box, white box testing etc.
6. It, generally, comes first—done before validation.	6. It generally follows verification.
7. It answers to the question—Are we building the product right ?	7. It answers to the question—Are we building the right product ?
8. It can catch errors that validation cannot catch.	8. It can catch errors that verification cannot catch.

Both of these are **essential** and **complementary.** Each provides its own sets of **Error Filters.** Each has its own way of finding out the errors in the software.

2.2. Differences between QA and QC ?

Quality Assurance : The planned and systematic activities implemented in a quality system so that quality requirements for a product or service will be fulfilled, is known as **quality assurance.**

Quality Control : The observation techniques and activities used to fulfill requirements for quality is known as a **quality control.**

Both are, however, different to each other. We tabulate the differences between them.

Quality Assurance (QA)	Quality Control (QC)
1. It is process related.	1. It is product related.
2. It focuses on the process used to develop a product.	2. It focuses on testing of a product developed or a product under development.
3. It involves the quality of the processes.	3. It involves the quality of the products.
4. It is a preventive control.	4. It is a detective control.
5. Allegiance is to development.	5. Allegiance is not to development.

2.3. Evolving Nature of Area

As the complexity and diversity of software products continue to increase, the challenge to develop new and more effective V&V strategies continues. The V&V approaches that were reasonably effective on small batch—oriented products are not sufficient for concurrent, distributed or embedded products. Thus, this area will continue to evolve as new research results emerge in response to new V&V challenges.

2.4. V&V Limitations

The overall objective of software V&V approaches is to insure that the product is free from failures and meets its user's expectations. There are several theoretical and practical limitations that make this objective impossible to obtain for many products. They are discussed below :

1. Theoretical Foundations

Howden claims the most important theoretical result in program testing and analysis is that no general purpose testing or analysis procedure can be used to prove program correctness.

2. Impracticality of Testing all Data

For most programs, it is impractical to attempt to test the program with all possible inputs, due to a combinational explosion. For those inputs selected, a testing oracle is needed to determine the correctness of the output for a particular test input.

3. Impracticality of Testing All Paths

For most programs, it is impractical to attempt to test all execution paths through the product, due to a combinational explosion. It is also not possible to develop an algorithm for generating test data for paths in an arbitrary product, due to the mobility to determine path feasibility.

4. No Absolute Proof of Correctness

Howden claims that there is no such thing as an absolute proof of correctness. Instead, he suggests that there are proofs of equivalency i.e., proofs that one description of a product is equivalent to another description. Hence, unless a formal specification can be shown to be correct and, indeed, reflects exactly the user's expectations, no claims of product correctness can be made.

2.5. Categorizing V&V Techniques

Various V&V techniques are categorized below :

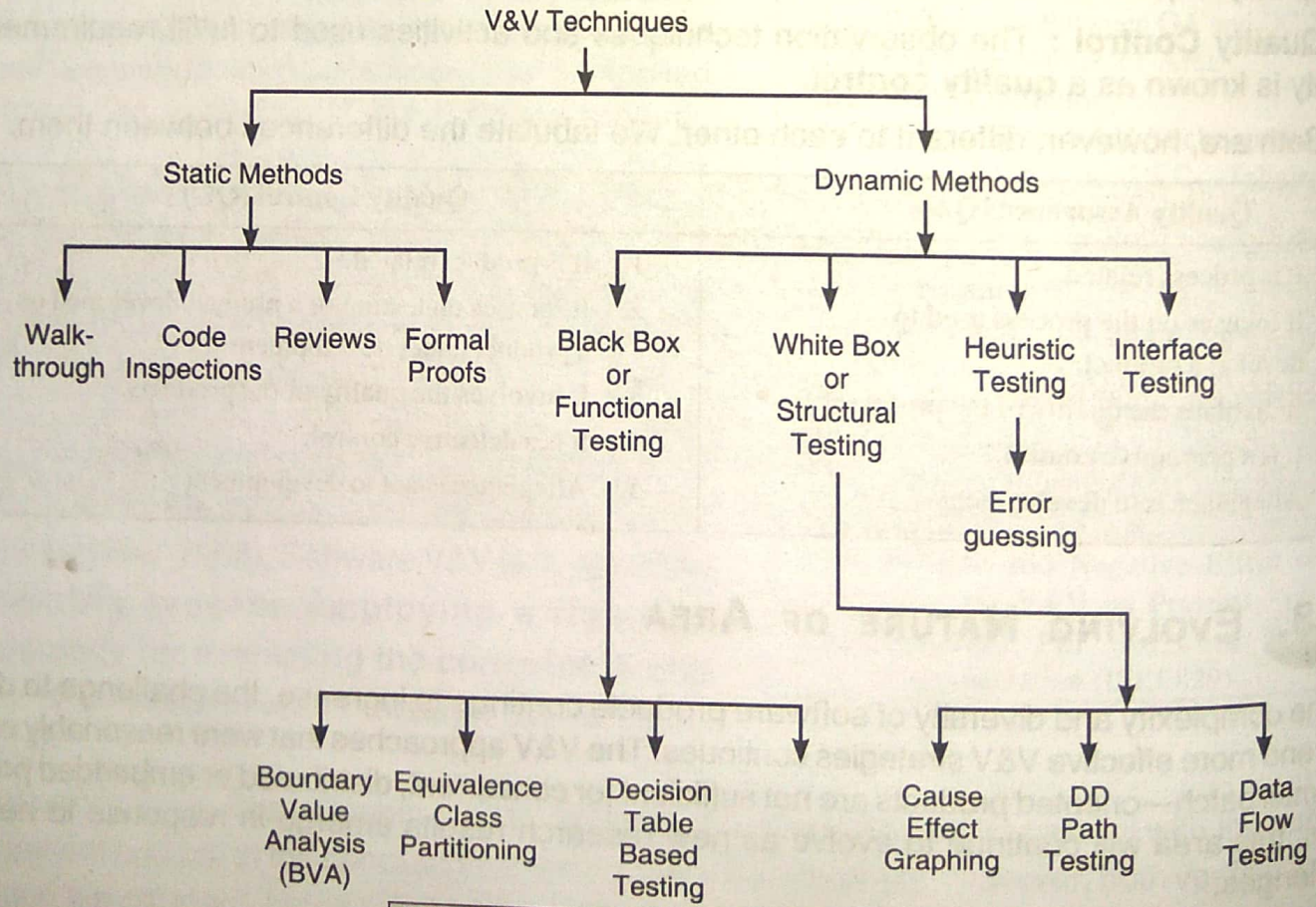


Fig. 2.1. Types of V&V Techniques.

The static methods of V&V involves basically, the review processes. Whereas dynamic methods like black box testing can be applied at all levels, even at system level. While the principle of white box testing is that it checks in for interface errors at the module level. Black box testing can also be done at module level by testing boundary conditions and low level functions like correct displaying of error messages.

2.6. Role of V&V in SDLC—Tabular Form *[IEEE Std. 1012]*

Traceability Analysis

It traces each software requirement back to the system requirements established in the concept activity. This is to ensure that each requirement correctly satisfies the system requirements and that no extraneous software requirements are added. In this technique, we also determine whether any derived requirements are consistent with the original objectives, physical laws and the technologies described in system document.

Interface Analysis

It is the detailed examination of the interface requirements specifications. The evaluation criteria is same as that for requirements specification. The main focus is on the interfaces between software, hardware, user and external software.

Criticality Analysis

Criticality is assigned to each software requirement. When requirements are combined into functions, the combined criticality of requirements form the cricality for the aggregate function. Criticality analysis is updated periodically as requirement changes are introduced. This is because such changes can cause an increase or decrease in a functions criticality which depends on how the revised requirement impacts system criticality.

Criticality analysis is a method used to locate and reduce high risk problems and is performed at the beginning of the project. It identifies the functions and modules that are required to implement critical program functions or quality requirements like safety, security etc.

Criticality analysis involves the following steps :

Step 1 : Construct a block diagram or control flow diagram (CFD) of the system and its elements. Each block will represent one software function (or module) only.

Step 2 : Trace each critical function or quality requirement through CFD.

Step 3 : Classify all traced software functions as critical to :

(a) Proper execution of critical software functions.

(b) Proper execution of critical quality requirements.

Step 4 : Focus additional analysis on these traced critical software functions.

Step 5 : Repeat critality analysis for each life cycle process to determine whether the implementation details shift the emphasis of the criticality.

Hazard and Risk Analysis

It is done during the requirements definition activity. Now hazards or risks are identified by further refining of the system requirements into detailed software requirements. These risks are assessed for their impact on the system.

Let us summarize these activities in a tabular form given below :

V&V Activity	V&V Tasks	Key Issues
1. Requirements V&V	➤ Traceability analysis. ➤ Software requirements evaluation. ➤ Interface analysis. ➤ Criticality analysis. ➤ System V&V test plan generation. ➤ Acceptance V&V test plan generation. ➤ Configuration management assessment. ➤ Hazard analysis. ➤ Risk analysis.	➤ Evaluates the correctness, completeness, accuracy, consistency, testability and readability of software requirements. ➤ Evaluates the software interfaces. ➤ Identifies the criticality of each software function. ➤ Initiates the V&V test planning for V&V system test. ➤ Initiates the V&V test planning for V&V acceptance test. ➤ Ensures completeness and adequacy of SCM process. ➤ Identifies potential hazards, based on the product data during the specified development activity. ➤ Identifies potential risks, based on the product data during the specified development activity.
2. Design V&V	➤ Traceability analysis. ➤ Software design evaluation. ➤ Interface analysis. ➤ Criticality analysis. ➤ Component V&V test plan generation and verification. ➤ Integration V&V test plan generation and verification. ➤ Hazard analysis ➤ Risk analysis.	➤ Evaluates software design modules for correctness, completeness, accuracy, consistency, testability and readability. ➤ Initiates the V&V test planning for V&V component test. ➤ Initiates the V&V test planning for V&V integration test.
3. Implementation V&V	➤ Traceability analysis. ➤ Source code and source code documentation evaluation. ➤ Interface analysis. ➤ Criticality analysis. ➤ V&V test case generation and verification ➤ V&V test procedure generation and verification. ➤ Component V&V test execution and verification. ➤ Hazard analysis. ➤ Risk analysis.	➤ Verifies the correctness, completeness, consistency, accuracy, testability and readability of source code.

V&V Activity	V&V Tasks	Key Issues
4. Test V&V	➤ Traceability analysis. ➤ Acceptance V&V test procedure generation and verification. ➤ Integration V&V test execution and verification. ➤ System V&V test execution and verification. ➤ Acceptance V&V test execution and verification.	
5. Maintenance V&V	➤ SVVP (Software verification and validation plan) revision. ➤ Proposed change assessment. ➤ Anomaly evaluation. ➤ Criticality analysis. ➤ Migration assessment. ➤ Retirement assessment. ➤ Hazard analysis. ➤ Risk analysis.	➤ Modifies the SVVP. ➤ Evaluates the effect on software of operation anomalies. ➤ Verifies the correctness of software when migrated to a different operational environment. ➤ Ensures that the existing system continues to function correctly when specific software elements are retired.

2.7. Proof of Correctness (Formal Verification)

A proof of correctness is a mathematical proof that a computer program or a part thereof will, when executed, yield correct results i.e., results fulfilling specific requirements. Before proving a program correct, the theorem to be proved must, of course, be formulated.

Hypothesis : The hypothesis of such a correctness theorem is typically a condition that the relevant program variables must satisfy immediately 'before' the program is executed. This condition is called the **'precondition'.**

Thesis : The thesis of the correctness theorem is typically a condition that the relevant program variables must satisfy immediately 'after' execution of the program. This latter condition is called the **'postcondition'.**

So, the correctness theorem is stated as follows :

"If the condition, V, is true before execution of the program, S, then the condition, P, will be true after execution of S".

where V is precondition and P is postcondition.

Notation : Such a correctness theorem is usually written as {V} S {P}, where V, S and P have been explained above.

By 'program variable' we broadly include input and output data, e.g., data entered via a keyboard, displayed on a screen or printed on paper. Any externally observable aspect of the program's execution may be covered by the precondition and postcondition.

2.8. Simulation and Prototyping

Simulation and prototying are techniques for analyzing the expected behaviour of a product. There are many approaches to constructing simulations and prototypes that are well documented in the literature.

For V&V purposes, simulations and prototypes are normally used to analyze requirements and specifications to insure that they reflect the user's needs. Since they are executable, they offer additional insight into the completeness and correctness of these documents.

Simulations and prototypes can also be used to analyze predicted product performance, especially for candidate product designs, to insure that they conform to the requirements.

It is important to note that the utilization of simulation and prototyping as V&V technique requires that the simulations and prototypes themselves be correct. Thus, the utilization of these techniques requires an additional level of V&V activity.

2.9. Requirements Tracing

"It is a technique for insuring that the product, as well as the testing of the product, addresses each of its requirements." The usual approach to performing requirements tracing uses matrices.

(a) **One type of matrix maps requirements to software modules.** Construction and analysis of this matrix can help insure that all requirements are properly addressed by the product and that the product does not have any superfluous capabilities.

System verification diagrams are another way of analyzing requirements/modules traceability.

(b) **Another type of matrix** maps requirements to test cases. Construction and analysis of this matrix can help insure that all requirements are properly tested.

(c) A **third type of matrix** maps requirements to their evaluation approach. The evaluation approaches may consist of various levels of testing, reviews, simulations etc.

The requirements/evaluation matrix insures that all requirements will undergo some form of V&V. Requirements tracing can be applied for all the products of the software evolution process.

2.10. Software V&V Planning (SVVP)

The development of a comprehensive V&V plan is essential to the success of a project. This plan must be developed early in the project. Depending on the development approach followed, multiple levels of test plans may be developed, corresponding to various levels of V&V activities. IEEE 83b has documented the guidelines for the contents of system, software, build and module test plans.

The steps that are followed for SVVP are listed below :

Step 1 : Identification of V&V Goals

V&V goals must be identified from the requirements and specifications. These goals must address those attributes of the product that correspond to its user expectations. These goals must be achievable, taking into account both theoretical and practical limitations.

Step 2 : Selection of V&V Techniques

Once step 1 (above) is finished, we must select specific techniques for each of the product that evolves during SDLC. These are given below :

(a) During Requirements Phase : The applicable techniques for accomplishing the V&V objectives for requirements are—technical reviews, prototyping and simulations. The review process is often called as a **System Requirement Review (SRR).**

(b) During Specifications Phase : The applicable techniques for this phase are technical reviews, requirements tracing, prototyping and simulations. The requirements must be traced to the specifications.

(c) During Design Phase : The techniques for accomplishing the V&V objectives for designs are technical reviews, requirements tracing, prototyping, simulation and proof of correctness. We can go for 2 types of design reviews :

(i) **High level designs** that correspond to an architectural view of the product are often reviewed in a **Preliminary Design Review (PDRs).**

(ii) **Detailed designs** are addressed by a **Critical Design Review (CDRs).**

(d) During Implementation Phase : The applicable techniques for accomplishing V&V objectives for implementation are technical reviews, requirements tracing, testing and proof of correctness. Various code review techniques such as walk throughs and inspections exist.

At the source-code level, several static analysis techniques are available for detecting implementation errors. The requirements tracing activity is here concerned with tracing requirements to source-code modules. The bulk of the V&V activity for source code consists of testing. Multiple levels of testing are usually performed. At the module-level, proof-of-correctness techniques may be applied, if applicable.

(e) During Maintenance Phase : Since changes describe modifications to products, the same techniques used for V&V during development may be applied during modification. Changes to implementation require regression testing.

Step 3 : Organizational Responsibilities

The organizational structure of a project is a key planning consideration for project managers. An important aspect of this structure is delegation of V&V activities to various organizations.

This decision is based upon the size, complexity and criticality of the product. Four types of organizations are addressed. These organizations reflect typical strategies for partitioning tasks to achieve V&V goals for the product. These are :

(a) Developmental Organization : This type of organization has the following responsibilities :

1. To participate in technical reviews for all of the evolution products.
2. To construct prototypes and simulations.
3. To prepare and execute test plans for unit and integration levels of testing. This is called as **'Preliminary Qualification Testing' (PQT).**
4. To construct any applicable proofs of correctness at the module level.

(b) Independent Test Organization (ITO) : This type of organization has the following responsibilities :

1. It enables test activities to occur in parallel with those of development.
2. It participates in all of the product's technical reviews and monitors PQT effort.
3. The primary responsibility of the ITO is the preparation and execution of the product's system test plan. This is sometimes referred to as the **'Formal Qualification Test' (FQT).**

   The plan for this must contain the equivalent of a requirements/evaluation matrix that defines the V&V approach to be applied for each requirement.
4. If the product must be integrated with other products, this integration activity is normally the responsibility of the ITO.

(c) Software Quality Assurance (SQA) Organizations : The intent here is to identify some activities for assuring software quality. Evaluations are the primary avenue for assuring software quality. Some evaluation types are given below :

(i) Internal consistency of product.
(ii) Understandability of product.
(iii) Traceability to indicated documents.
(iv) Consistency with indicated documents.
(v) Appropriate allocation of sizing, timing and resources.
(vi) Adequate test coverage of requirements.
(vii) Completeness of testing.
(viii) Completeness of regression testing.

(d) Independent V&V Contractor : An independent contractor may be selected to do V&V. The scope of activities of this organization varies from that of ITO (discussed above) and SQA organization.

Step 4 : Integrating V&V Approaches

Once a set of V&V objectives has been identified, an overall integrated V&V approach must be determined. This approach involves integration of techniques applicable to various life cycle phases as well as delegation of these tasks among the project's organizations. The planning of this integrated V&V approach is very dependent upon the nature of the product and the process used to develop it. Earlier waterfall approach for testing was used and now incremental approach is used. Regardless of the approach selected, V&V progress must be tracked. Requirements/ evaluation matrices play a key role in this tracking by providing a means of insuring that each requirement of the product is addressed.

Step 5 : Problem Tracking

(i) It involves documenting the problems encountered during V&V effort.
(ii) Routing these problems to appropriate persons for correctness.
(iii) Insuring that corrections have been done.
(iv) Typical information to be collected includes :
  (a) When the problem occurred.
  (b) Where the problem occurred.
  (c) State of the system before occurrence.
  (d) Evidence of the problem.
  (e) Priority for solving problem.

Note : This fifth step is very important, when we go with OS testing.

Step 6 : Tracking Test Activities

SVVP must provide mechanism for tracking testing effort, testing cost and testing quality. To do this the following data is collected :

(a) Number of tests executed.
(b) Number of tests remaining.
(c) Time used.

(d) Resources used.

(e) Number of problems found.

These data can be used to compare actual test progress against scheduled progress.

Step 7 : Assessment

It is important that the software V&V plan provide for the ability to collect data that can be used to assess both the product and the techniques used to develop it. This involves careful collection of error and failure data, as well as analysis and classification of these data.

2.11. Software Technical Reviews (STRs)

A review process can be defined as a critical evaluation of an object. It includes techniques such as walk-throughs, inspections and audits. Most of these approaches involve a group meeting to assess a work product.

Software technical reviews can be used to examine all the products of the software evolution process. In particular, they are especially applicable and necessary for those products not yet in machine-processable form, such as requirements or specifications written in natural language.

2.11.1. Rationale for STRs

The main rationale behind STRs are as follows :

(a) Error Prone Software Development and Maintenance Process : The complexity and error-prone nature of developing and maintaining software should be demonstrated with statistics depicting error frequencies for intermediate software products. These statistics must also convey the message that errors occur throughout the development process and that the later these errors are detected, the higher the cost for their repair.

Summary :

(i) Complexity of software development and maintenance processes.

(ii) Error frequencies for software work products.

(iii) Error distribution throughout development phases.

(iv) Increasing costs for error removal throughout the life cycle.

(b) Inability to Test all Software : It is not possible to test all software. **Clearly exhaustive testing of code is impractical.** Current technology also does not exist for testing a specification or high level design. The idea of testing a software test plan is also bewildering. Testing also does not address quality issues or adherence to standards which are possible with review processes.

Summary :

(i) Exhaustive testing is impossible.

(ii) Intermediate software products are largely untestable.

(c) Reviews are a form of Testing : The degree of formalism, scheduling and generally positive attitude afforded to testing must exist for software technical reviews if qualify products are to be produced.

Summary :

(i) Objectives

(ii) Human-based versus machine based

(iii) Attitudes and norms.

(d) Reviews are a Way of Tracking a Project : Through identification of deliverables with well defined entry and exit criteria and successful review of these deliverables, progress on a project can be followed and managed more easily [Fagan]. In essence, review processes provide milestones with teeth. This tracking is very beneficial for both project management and customers.

Summary :

(i) Individual developer tracking
(ii) Management tracking
(iii) Customer tracking

(e) Reviews Provide Feedback : The instructor should discuss and provide examples about the value of review processes for providing feedback about software and its development process.

Summary :

(i) Product (ii) Process

(f) Educational Aspects of Reviews : It includes benefits like a better understanding of the software by the review participants that can be obtained by reading the documentation as well as the opportunity of acquiring additional technical skills by observing the work of others.

Summary :

(i) Project understanding (ii) Technical skills

2.11.2. Types of STRs

A variety of STRs are possible on a project depending upon the developmental model followed, the type of software product being produced and the standards which must be adhered to. These developmental modes may be

(i) Waterfall model (ii) Rapid prototyping
(iii) Iterative enhancement (iv) Maintenance activity modeling.

And the current standards may be

(i) Military standards (ii) IEEE standards
(iii) NBS standards

Reviews are classified as formal and Informal reviews. We tabulate the differences between them in tabular form.

Informal reviews	Formal reviews
(i) It is a type of review that typically occurs spontaneously among peers.	(i) It is a planned meeting.
(ii) Reviewers have no responsibility.	(ii) Reviewers are held accountable for their participation in the review.
(iii) No review reports are generated.	(iii) Review reports containing action items is generated and acted upon.

2.11.3. Review Methodologies

There are three approaches to reviews

(a) Walk through (or presentation reviews)
(b) Inspection (or work product reviews), and
(c) Audits

(a) Walkthroughs (or Presentation Reviews) : The first approach to be described is 'walk throughs'. Walk throughs are well defined by Yourdon. Walkthroughs can be viewed as presentation reviews in which a review participant, usually the developer of the software being reviewed, narrates a description of the software and the remainder of the review group provides their feedback throughout the presentation.

They are also known as the **presentation reviews** because the bulk of the feedback usually only occurs for the material actually presented at the level it is presented. Thus, advance preparation on the part of reviewers is often not detectable during a walkthrough. Walkthroughs suffer from these limitations as well as the fact that they may lead to disorganized and uncontrolled reviews.

Walkthroughs may also be stressful if the developer of the software is conducting the walkthrough. This has lead to many variations such as having someone other than the developer perform the walkthrough. It is also possible to combine multiple reviews into a single review such as a combined design and code walkthrough.

(b) Inspections (or Walk Product Reviews) : It is a formal approach. Inspections were first performed by Fagan at IBM. They require a high degree of preparation for the review participants but the benefits include a more systematic review of the software and a more controlled and less stressed meeting.

There are many variations of inspections, but all include some form of a checklist or agenda that guides the preparation of review participants and serves to organize the review meeting itself. Inspections are also characterized by rigorous entry and exit requirements for the work products being inspected.

Another important point to be noted is that an inspection process involves the collections of data that can be used to feedback on the quality of the development and review processes as well as influence future validation techniques on the software itself. Let us compare the two in a tabular form.

Inspection	Walkthroughs
1. It is a five-step process that is well-formalized.	1. It has fewer steps than inspection and is a less formal process.
2. It uses checklists for locating errors.	2. It does not use a checklist.
3. It is used to analyze the quality of the process.	3. It is used to improve the quality of the product.
4. This process takes longer time.	4. It does not take longer time.
5. It focuses on training of junior staff.	5. It focuses on finding defects.

(c) Audits : Audits should also be described as an external type of review process. Audits serve to insure that the software is properly validated and that the process is producing its intended results.

2.12. INDEPENDENT V&V CONTRACTOR (IV&V)

An independent V&V contractor may sometimes be used to insure independent objectivity and evaluation for the customer.

The use of a different organization, other than the software development group, for software V&V is called independent verification and validation (IV&V). Three types of independence are usually required :

I. Technical Independence : It requires that members of the IV&V team (organization or group) may not be personnel involved in the development of the software. This team must have some knowledge about the system design or some engineering background enabling them to understand the system. The IV&V team must not be influenced by the development team when the IV&V team is learning about the system requirements, proposed solutions for building the system and problems encountered. Technical independence is crucial in the team's ability to detect the subtle software requirements, software design and coding errors that escape detection by development testing and SQA reviews.

The technical IV&V team may need to share tools from the computer support environment (e.g., compilers, assemblers, utilities) but should execute qualification tests on these tools to ensure that the common tools themselves do not mask errors in the software being analyzed and tested. The IV&V team uses or develops its own set of test and analysis tools separate from the developer's tools whenever possible.

II. Managerial Independence : It means that the responsibility for IV&V belongs to an organization outside the contractor and program organizations that develop the software. While assurance objectives may be decided by regulations and project requirements, the IV&V team independently decides the areas of the software/system to analyze and test, techniques to conduct the IV&V, schedule of tasks and technical issues to act upon. The IV&V team provides its findings in a timely fashion simultaneously to both the development team and the systems management who acts upon.

III. Financial Independence : It means that control of the IV&V budget is retained in an organization outside the contractor and program organization that develop the software. This indepedence protects against diversion of funds or adverse financial pressures or influences that may cause delay or stopping of IV&V analysis and test tasks and timely reporting of results.

2.13. Positive and Negative Effect of Software V&V on Projects

Software V&V has some **positive effects** on a software project. These are given below :

1. Better quality of software. This includes factors like completeness, consistency, readability and testablity of the software.
2. More stable requirements.
3. More rigorous development planning, at least to interface with the software V&V organization.
4. Better adherence by the development organization to programming language and development standards and configuration management practices.
5. Early error detection and reduced false starts.
6. Better schedule compliance and progress monitoring.
7. Greater project management visibility into interim technical quality and progress.
8. Better criteria and results for decision-making at formal reviews and audits.

Some **negative effects** of software V&V on a software development project include—

1. Additional project cost of software V&V (10–30% extra).
2. Additional interface involving the development team, user and software V&V organization. For example attendance at software V&V status meetings, anomaly resolution meetings.
3. Additional documentation requirements, beyond the deliverable products, if software V&V is receiving incremental program and documentation releases.
4. Need to share computing facilities with and to provide access to, classified data for the software V&V organization.
5. Lower development staff productivity if programmers and engineers spend time explaining the system to software V&V analysts, especially if explanations are not documented.
6. Increased paper work to provide written responses to software V&V error reports and other V&V data requirements. For example, notices of formal review and audit meetings, updates to software release schedule and response to anomaly reports.
7. Productivity of development staff affected adversely in resolving invalid anomaly reports.

Some steps can be taken to minimize the negative effects and to maximize the positive effects of software V&V. To recover much of the software V&V costs, software V&V is started early in the software requirements phase. The interface activities for documentation, data and software deliveries between developer and software V&V groups should be considered as an inherently necessary step required to evaluate intermediate development products.

To offset unnecessary costs, software V&V must organize its activities to focus on critical areas of the software so that it uncovers critical errors for the development group and thereby results in significant cost savings to the development process. To do this, software V&V must use its criticality analysis to identify critical areas and must scrutinize each discrepancy before release to ensure that no false or inaccurate information is released to prevent the development group wasting time on inaccurate or trivial reports.

To eliminate the need to have development personnel train the software V&V staff, it is imperative that software V&V select personnel who are experienced and knowledgeable about the software and its engineering application. When software V&V engineers and computer scientists reconstruct the specific details and idiosyncrasies of the software as a method of reconfirming the correctness of engineering and programming assumptions, they often find subtle errors. They gain detailed insight into the development process and an ability to spot critical errors early. The cost of the development interface is minimal, and at times nonexistent, when the software V&V assessment is independent.

Finally, the discrepancies detected in software and the improvement in documentation quality resulting from error correction suggests that software V&V costs are offset by having more reliable and maintainable software. Many companies rely on their software systems for their daily operations. Failure of the system, loss of data, release of or tampering with sensitive information may cause serious work disruptions and serious financial impact. The costs of software V&V are offset in many application areas by increased reliability during operation and reduced costs of maintenance.

2.14. Standard For Software Test Documentation (IEEE829)

- The IEEE829 standard for software test documentation describes a set of basic software test documents. It defines the content and form of each test document.
- In this addendum we give a summary of the structure of the most importnat IEEE829 defined test documents.
- This addendum is based on the course materials by Jukka Paakki (and the IEEE829 standard).

Test Plan

1. Test-plan Identifier : Specifies the unique identifier assigned to the test plan.

2. Introduction : Summarizes the software items and software features to be tested, provides references to the documents relevant for testing (overall project plan, quality assurance plan, configuration management plan, applicable standards...)

3. Test Items : Identifies the items to be tested, including their version/revision level; provides references to the relevant item documentation (requirements specification, design specification, user's guide, operations guide, installation guide, ...); also identifies items which are specifically excluded from testing.

4. Features to be Tested : Identifies all software features and their combinations to be tested, identifies the test-design specification associated with each feature and each combination of features.

5. Features not to be Tested : Identifies all features and significant combinations of features which will not be tested, and the reasons for this.

6. Approach : Describes the overall approach to testing (the testing activities and techniques applied, the testing of non-functional requirements such as performance and security, the tools used in testing); specifies completion criteria (for example, error frequency or code coverage); identifies significant constraints such as testing-resource availability and strict deadlines; serves for estimating the testing efforts.

7. Item pass/fail Criteria : Specifies the criteria to be used to determine whether each test item has passed or failed testing.

8. Suspension criteria and resumption : Specifies the criteria used to suspend all or portion of the testing activity on the test items (at the end of working day, due to hardware failure or other external exception, ...), specifies the testing activities which must be repeated when testing is resumed.

9. Test Deliverables : Identifies the deliverable documents, typically test-design specifications, test-case specifications, test-procedure specifications, test-item transmittal reports, test logs, test-incident reports, description of test-input data and test-output data, description of test tools.

10. Testing Tasks : Identifies the set of tasks necessary to prepare and perform testing (description of the main phases in the testing process, design of verification mechanisms, plan for maintenance of the testing environment, ...).

11. Environmental needs : Specifies both the necessary and desired properties of the test environment (hardware, communications and systems software, software libraries, test support tools, level of security for the test facilities, drivers and stubs to be implemented, office or laboratory space, ...).

12. Responsibilities : Identifies the groups of persons responsible for managing, designing, preparing, executing, witnessing, checking and resolving the testing process; identifies the groups responsible for providing the test items (section 3) and the environmental needs (section 11).

13. Staffing and Training Needs : Specifies the number of testers by skill level, and identifies training options for providing necessary skills.

14. Schedule : Includes test milestones (those defined in the overall project plan as well as those identified as the overall project plan as well as those identifies as internal ones in the testing process), estimates the time required to do each testing task, identifies the temporal dependencies between testing tasks, specifies the schedule over calendar time for each task and milestone.

15. Risks and Contingencies : Identifies the high-risk assumptions of the test plan (lack of skilled personnel, possible technilcal problems, ...), specifies contingency plans for each risk (employment of additional testers, increase of night shift, exclusion of some tests of minor importance, ...).

16. Approvals : Specifies the persons who must approve this plan.

Test-Case Specification

1. Test-case Specification identifier : Specifies the unique identifier assigned to this test-case specification.

2. Test Items : Identifies and briefly describes the items and features to be exercised by this test case, supplies references to the relevant item documentation (requirements specification, design specification, user's guide, operations guide, installation guide, ...).

3. Input Specifications : Specifies each input required to execute the test case (by value with tolerance or by name); identifies all appropriate databases, files, terminal messages, memory resident areas, and external values passed by the operating system; specifies all required relationships between inputs (for example, timing).

4. Output Specifications : Specifies all of the outputs and features (for example, response time) required of the test items, provides the exact value (with tolerances where appropriate) for each required output or feature.

5. Environmental Needs : Specifies the hardware and software configuration eeded to execute this test case, as well as other requirements (such as specially trained operators or testers).

6. Special Procedural Requirements : Describes any special constraints on the test procedures which execute this test case (special set-up, operator intervention, ...).

7. Intercase Dependecies : Lists the identifiers of test cases which must be executed prior to this test case, describes the nature of the dependencies.

Test-Incident Report (Bug Report)

1. Bug-Report Identifier : Specifies the unique identifier assigned to this report.

2. Summary : Summarizes the (bug) incident by identifying the test items involved (with version/revision level) and by referencing the relevant documents (test-procedure specification, test-case specification, test log).

3. Bug Description : Provides a description of the incident, so as to correct the bug, repeat the incident or analyze it off-line.

- Inputs
- Expected results
- Actual results
- Date and time
- Test-procedure step
- Environment.
- Repeatability (whether repeated whether occurring always occasionally or just once).
- Testers
- Other observers.
- Additional information that may help to isolate and correct the cause of the incident; for example, the sequence of operational steps or history of user-interface commands that lead to the (bug) incident.

4. Impact : Priority of solving the incident/correcting the bug (urgent, high, medium, low).

Test-Summary Report

1. Test-Summary-Report Identifier : Specifies the unique identifier assigned to this report.

2. Summary : Summarizes the evaluation of the test items, identifies the items tested (including their version/revision level), indicates the environment, in which the testing activities took place, supplies references to the documentation over the testing process (test plan, test-design specifications, test-procedure specifications, test-item transmittal reports, test logs, test-incident reports, ...).

3. Variances : Reports any variances/deviations of the test items from their design specifications, indicates any variances of the actual testing process from the test plan or test procedures, specifies the reason for each variance.

4. Comprehensiveness Assessment : Evaluates the comprehensiveness of the actual testing process against the criteria specified in the test plan, identifies features or feature combinations which were not sufficiently tested and explains the reasons for omission.

5. Summary of Results : Summarizes the success of testing (such as coverage), identifies all resolved and unresolved incidents.

6. Evaluation : Provides an overall evaluation of each test item including its limitations (based upon the test results and the item-level pass/fail criteria).

7. Summary of Activities : Summarizes the major testing activities and events, summarizes resource consumption (total staffing level, total person-hours, total machine time, total elapsed time used for each of the major testing activities, ...).

8. Approvals : Specifies the persons who must approve this report (and the whole testing phase).

Inspection Checklist for Test Plans

1. Have all materials required for a test plan inspection been received ?
2. Are all materials in the proper physical format ?
3. Have all test plan standards been followed ?
4. Has the testing environment been completely specified ?
5. Have all resources been considered, both human and hardware/software ?
6. Have all testing dependicies been addressed (driver function, hardware, etc.) ?
7. Is the test plan complete, i.e., does it verify all of the requirements ? (For unit testing : does the plan test all functional and structural variations from the high-level and detailed design.)
8. Is each script detailed and specific enough to provide the basis for test case generation ?
9. Are all test entrance and exit criteria sufficient and realistic ?
10. Area invalid as well as valid input conditions tested ?
11. Have all pass/fail criteria been defined ?
12. Does the test plan outline the levels of acceptability for pass/fail and exit criteria (e.g., defect tolerance) ?
13. Have all suspension criteria and resumption requirements been identified ?
14. Are all items excluded from testing documented as such ?
15. Have all test deliverables been defined ?
16. Will software development changes invalidate the plan ? (Relevant for unit test plans only).
17. Is the intent of the test plan to show the presence of failures and not merely the absence of failures ?
18. Is the test plan complete, correct and unambiguous ?
19. Are there holes in the plan, is there overlap in the plan ?
20. Does the test plan offer a measure of test completeness and test reliability to be sought ?
21. Are the test strategy and philosophy feasible ?

Inspection Checklist for Test Cases

1. Have all materials required for a test case inspection been received ?
2. Are all materials in the proper physical format ?
3. Have all test case standards been followed ?
4. Area the functional variations exercised by each test case required by the test plan ? (Relevant for unit test case documents only.)
5. Area the functional variations exercised by each test case clearly documentéd in the test case description ? (Relevant for unit test case documents only.)
6. Does each test case include a complete description of the expected input and output or result ?
7. Have all testing execution procedures been defined and documented ?
8. Have all testing dependencies been addressed (driver function, hardware, etc.)?
9. Do the test cases accurately implement the test plan ?
10. Are all data set definitions and setup requirements complete and accurate ?