1

INTRODUCTION

Chapter Structure

1.0: Introduction
1.1: What is Quality?
1.1.1: Factors of Quality
1.2: Customer's View of Quality
1.3: Supplier's View of Quality
1.3.1 Expectations
1.4: Financial Aspects of Quality
1.5: Organisation Culture
1.5.1: Difference between Quality culture in 'Q' & Quality Culture in 'q'
1.6: Software Development Model
1.6.1: List of Software Development model
1.6.1.1.: Waterfall Development model
1.6.1.2: Iterative Development model
1.6.1.3: Incremental Development model
1.6.1.4: Spiral Development model
1.6.1.5: Prototyping Development model
1.6.1.6: Rapid Application Development model
1.6.1.7: Agile Development model
1.6.1.8: Maintenance Development model
Questions
Further Reading

1. Introduction:

In early days, Product quality was governed by the Human's skills which used to differ from person to person as the process of creating that product differs. Every product was considered as a separate project & every instance of the manufacturing process let to products of different quality attributes. Due to increase in demand, product specialization & mass production came into picture which leaded to technological development.

In the new era, where mass production needs specialized domain knowledge & manufacturing skills for product creation in huge quantities. The market is growing day by day, and to sustain it is important to maintain the individual product identity in terms of quality. There are one or more similar products exists in the market which fulfils similar needs. They may differ from one another to some extent based on their cost, delivery schedule to acquire it, features & functionalities. They are called as attributes of quality of a product.

1.1 What is Quality?

For us quality means, an idea to shortlist a product or select the best from the given range of similar products. We may not aware of the complete idea behind choosing a product over the other. This is the major issue faced by the people working in quality field even today as it is very difficult to decide what contributes customer loyalty or first-time sale and subsequent repeat sale. The term 'quality' may differ from person to person. For some people quality means a defect-less product, some might think it as a product which matches his/her expectations, matches his/her concept of cost & delivery schedule along with the services offered. In short, we can say **'Quality Is fitness for use'.**

1.1.1 Factors of Quality or Core components of Quality

1. **Quality is based on Customer Satisfaction:**
    a. The effect of quality product delivered & used by a customer, on his satisfaction and delight is the most important factor in determining whether quality has been achieved or not.
    b. It talks about the ability of a product or service to satisfy a customer by fulfilling his/her needs.
2. **The organisation must define Quality parameters before it can be achieved:**
    a. It is difficult for the manufacturer to achieve the quality of product without knowing what customer is looking for.
    b. So, each organisation should decide some parameters through which they can achieve quality. Following are the parameters the organisation should look for,

   c. Define: Defining the product in terms of features, functionalities, attributes & characteristics of a product.
   d. Measure: The quantitative measures must be defined as an attribute of quality of a product. Measurement also gives a gap between what is expected by a customer & what is delivered to him when the product is sold.
   e. Monitor: There must be a mechanism used by the manufacturer to monitor the development, testing and delivering of a product process.
   f. Control: Control gives the ability to provide desired results & avoid the undesired things going to a customer.
   g. Improve: Continuous improvement are necessary to maintain ongoing customer satisfaction & overcome the possible competition.
3. **Management must lead the organisation through improvement efforts:**
   a. Quality must be taken into consideration to achieve the customer satisfaction.
   b. Management should decide on the vision, mission, policies, objectives, goals etc. to improve the Quality improvement program.
   c. And the same must be followed by the employees which is called as 'cultural change brought in by management'.
4. **Continuous process:**
   a. Earlier it was expected that the customer should inspect the product and report to the manufacturer for any kind of defects.
   b. Further it is the responsibility of the manufacturer to work on the defects and bear the cost of fixing those defects.
   c. This added to further loss to the manufacturer as well as the right of getting a good product is withdrawn form customer.
   d. For improving the quality and to have a win-win situation for both manufacturer & customer, quality must be produced at the first time and must be improved continuously.

1.2 Customer's View of Quality:

When customer pays some cost to get a product, he/she expects of getting a better product at a defined schedule, cost & with adequate service along with required features & functionalities. He finds value of the product in all such acquisitions.

1.2.1 Following are the factors affecting the Customer's view towards quality:

1. **Delivering a right product:**
   The product delivered to the Customer must satisfy all their needs and expectations. Even during the requirements gathering phase of Software Development Lifecycle (SDLC), business analyst or system designer must consider views of the customer to decide the validity of the product requirements.
2. **Satisfying Customer's Needs:**

A product may or may not be the best product as per the given constraints & requirements by the customer from the manufacturer's point of view. But the basic concept is, "product must be capable of satisfying customer needs". They may be part of processes for doing requirement analysis and selection of approach for designing based on decision analysis & resolution which comes under product development and delivery.

3. **Meeting Customer Expectations:**
   A customer has two types of expectations – expressed expectations and implied expectations. Expressed expectations are documented and given formally whereas implied expectations are those which are neither documented nor forms a part of requirement specifications. It is the duty of Developing Organisation to note as many implied expectations as possible to expressed definitions.
4. **Treating every Customer with Integrity, Courtesy, and Respect:**
   We should remember that the customer is the owner of the requirements and Organisation must understand that Customer understands what is required by him/her. The major responsibility of the Developing Organisation lies in fulfilling these requirements. Manufacturer cannot push his requirements on the customer. Customer queries over calls or mails must be clarified timely and with courtesy. The information thus provided should be accurate. Organisation must understand the Customer is the purpose of the business.

1.3 Supplier's view of Quality:

Supplier's needs must be satisfied by producing a product & selling it to Customer. The needs can vary from profitability, reputation in the market, customer satisfaction etc.

1.3.1These expectations can be fulfilled in following ways,

1. **Doing the Right Things:**
   Supplier is intended to do things right for the first time. Repetition of work, waste, scrap etc. involves more cost. Changes in the requirements phase causes rework of design, development, testing etc. This adds to the cost of development but if the budget remains the same then it's a huge loss to the company.
2. **Doing it the Right way:**
   Every supplier has their own strategy or process to develop a product. Sometimes, Customer may impose their development process to manufacture a product. The development organisation should capable enough of accommodating the new process to produce the product as required by the customer. The process definition must be the outcome of quality standards or models or business models.
3. **Doing it Right the First time:**

Doing right things at the first time may avoid frustration, scrap, rework etc. & improve profitability, reduce cost & improve customer satisfaction for the supplier. Doing right things at the first time improves performance, productivity & efficiency of a manufacturing process.

4. **Doing it on Time:**
   The resources required to develop a product are scarce and involves cost. The value of money changes with time. If the customer is expected to pay on each milestone, the producer must deliver milestones on time to realise money on time.

1.4 Financial Aspect of Quality

Earlier, people believed more price of a product represents better quality as it involves more inspection, testing, sorting etc. and ensures that only good parts are supplied to the customer. The sales price was defined as,

Sales Price=Cost of manufacturing + Cost of Quality + Profit

If we consider the monopoly way of life, this approach may be considered good since the price is decided by the manufacturer depending upon three factors described above. If any product enjoys higher profitability, more number of producers would enter competition.

Thus, in a competitive environment, the equation changes to,

Profit=Sales Price- [Cost of manufacturing + Cost of Quality]

1. **Cost of Manufacturing**: It is a cost required for developing the right product by right method at the first time. All the money involved in resources like material, people, licenses etc., forms a cost of manufacturing. The cost of manufacturing remains constant over the time span for the given technology & it has a direct relationship with the efforts.
2. **Cost of Quality:**
   a. Cost of Prevention: An organisation may define processes, guidelines, standards of development, testing so they also impart training programs to all the people involved in development & testing. This may represent cost of Prevention. For example, creation & use of formats, templates, various process models & standards etc.
   b. Cost of Appraisal: An organisation may perform various levels of reviews and testing to appraise the quality of the product and the process followed for developing the product.
   c. Cost of Failure: Cost of failure starts when a defect or violation is detected at any stage of development including post-delivery efforts spent on defect fixing. For example, any extent of rework, retesting, sorting, scrapping, regression testing, late payments, sales under concession etc.

2.Organisation Culture:

Organisation culture is based on their philosophy of existence, management perception & employee involvement in defining future. Quality improvements are required to bring in the change in Organisation culture. ‘Q’ organisations are more quality conscious than ‘q’ organisations. The difference between both is elaborated with the following table,

Quality Culture is ‘Q’	Quality Culture is ‘q’
1. Organisations under this category listens to Customer to understand their requirements.	1. Organisations under this category assumes that they know the customer requirements.
2. They are into finding the cost of Quality & also behind minimising the cost of failure.	2. They overlook cost of poor quality & hidden factory effect.
3. Doing things right for the first time & which is the motto of their success.	3. Doing things again and again to make it right, is their way of working.
4. It concentrates on continuous/continual process improvement to eliminate waste & get better output.	4. They work on finding & fixing the problem when it is found.
5. They take ownership of processes & defects at all levels.	5. They try to assign responsibility of defects to someone else.
6. They demonstrate leadership & commitment to quality & customer satisfaction.	6. They believe in assigning responsibility for quality to others.

3.Software Development Model:

It defines how the software is being built. It is also called as Software Development Life Cycle (SDLC). There are many approaches used for software development.

3.1 Following is the list of different models used for Software development,

1. Waterfall Development Model
2. Iterative Development Model
3. Incremental Development Model
4. Spiral Development Model
5. Prototyping Development Model
6. Rapid Application Development Model (RAD)
7. Agile Development Model
8. Maintenance Development Approach/Model

3.1.1 Waterfall Development Model:

a. It is the simplest of all.
b. Though it is always desirable to use waterfall model but it is not always feasible to use it.

c. It has a natural timeline where tasks are executed in a sequential fashion.
d. We start at the top of the waterfall with a feasibility study and flow down through the various project tasks finishing with implementation into the live environment.
e. Design flows through into development, which in turn flows into build, and finally on into test.
f. Typical waterfall model looks like this,

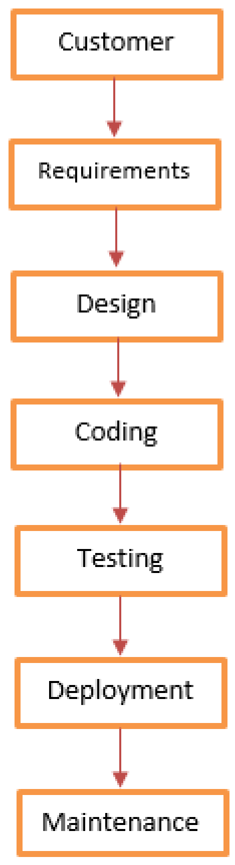


Fig 1.1: Waterfall Model

g. Arrows in waterfall model are unidirectional.
h. It assumes that the requirements are finalised and conveyed to the development team in one go.
i. These requirements are converted to High Level & Low-Level designs and are implemented using coding.
j. Code is integrated, tested and final output is deployed at Customer's premises. Furthermore, maintenance takes place.

3.1.2 Iterative Development Model:

a. It is more practical as compared to waterfall model.
b. Not all life cycles are sequential.
c. There are also iterative or incremental life cycles where, instead of one large development time line from beginning to end, we cycle through several smaller self-contained life cycle phases for the same project.
d. It does not assume that the requirements are finalised in one go.
e. Changes may have cascading effects where one change may initiate a chain of reaction of changes.
f. Limitations are: It takes many cycles of waterfall model. Due to number of changes the design becomes fragile.

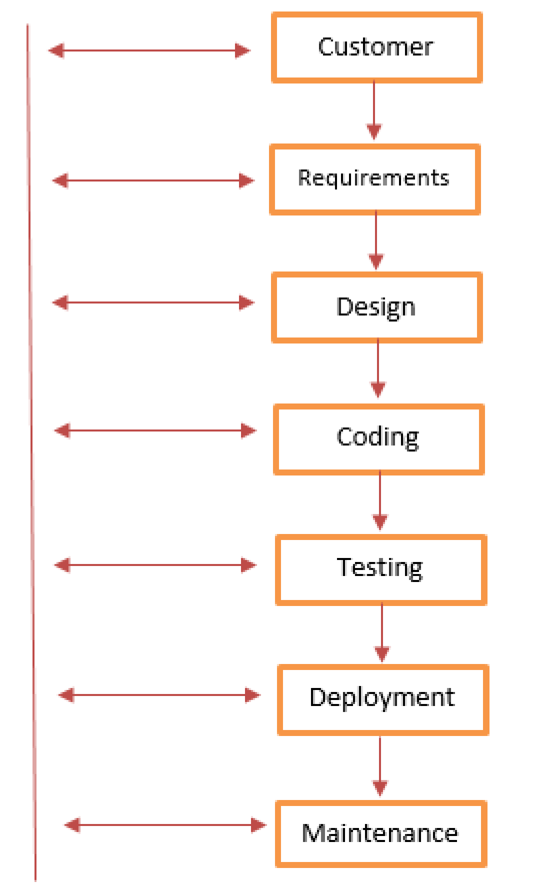


Fig. 1.2 Iterative Model

3.1.3 Incremental Development Model:

a. This model is suitable for huge systems, where these systems are divided in small subsystems and development takes place.
b. Further, these sub systems can be developed using waterfall model or iterative development model.
c. These subsystems may be connected to each other externally, either directly or indirectly.
d. A directly interconnected system allows the subsystems to talk with each other while indirectly interconnected system has some interconnecting application between two subsystems.

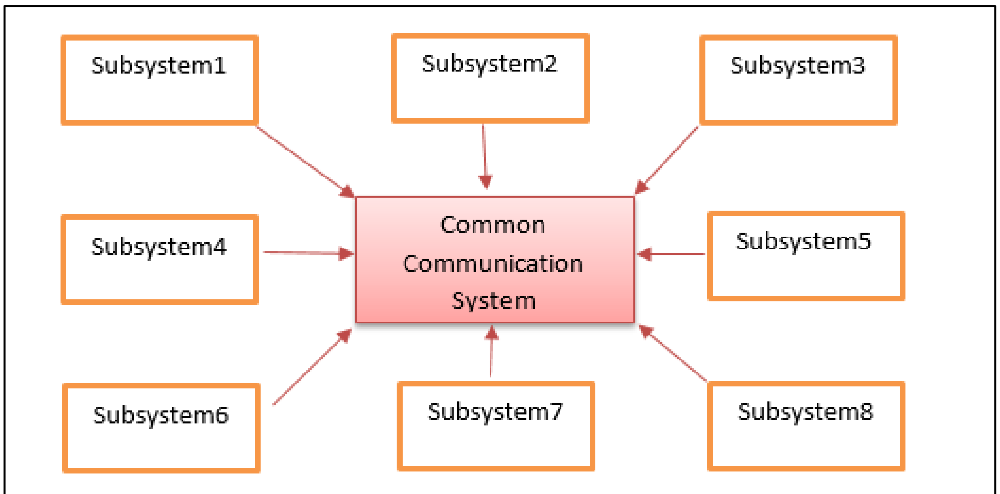


Fig. 1.3 Incremental Model

Limitations of the Incremental model are, system developed by different vendors has an issue of integration as parameter passing between different systems may be difficult. When a system is incremented with new subsystems, it changes the architecture of that system.

3.1.4 Spiral Development Model:

a. It assumes that customer requirements are obtained in multiple iterations & development also works in iterations.
b. First some functionalities are added, then product is created and released to customer.
c. By looking at the first iteration of implementation, the customer may add another chunk of requirements to the existing one.

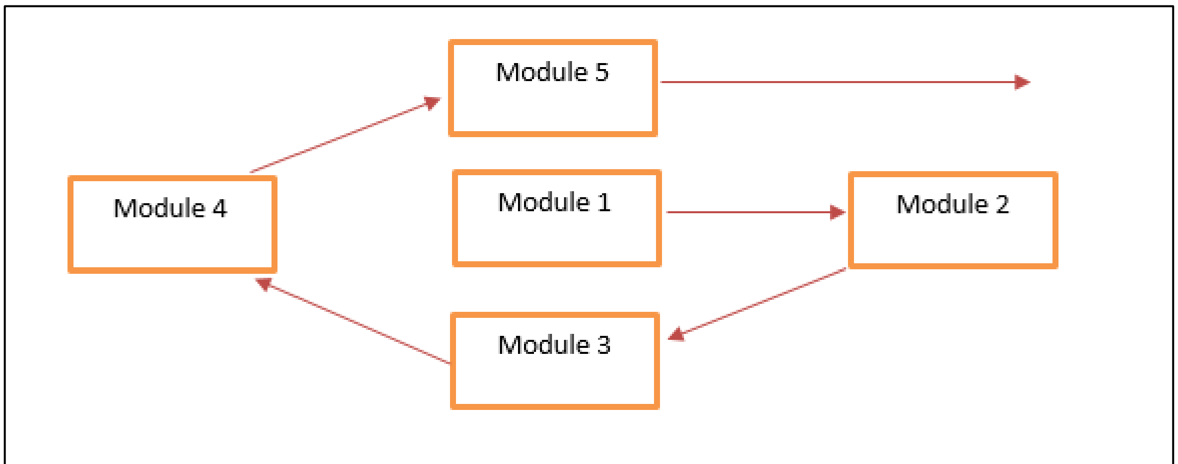


Fig.1.4 Spiral Development Method

d. Further, additional requirements tend to increase the size of the software spiral.

e. Limitations are as follows, Spiral model represents requirement induction as the software is being developed. Sometimes, it may lead to refactoring & changes in approach where initial structures become non-usable. It also needs huge regression testing.

3.1.5 Prototyping Development Model:

a. It has a top-bottom approach of development. It is also called as 'Reverse Integration Process'.
b. Major challenge in software development is to of procuring and understanding the customer requirements for the product.
c. Prototyping is one of the solutions for it.
d. In prototyping, initially a prototype model is created just like a cardboard model and it is shown to the Customer for further feedback.
e. This gives an idea to the customer of this product is going to be and can expect given set of requirements.
f. It helps the development team to understand the look & feel of the project.
g. Limitations are as follows, Customer might feel by looking at the prototype that the system is ready and may pressurise the development team to deliver it immediately.

3.1.6 Rapid Application Development Model (RAD):

a. Rapid Application Development doesn't mean developing the software product rapidly.
b. It means that, during the first meeting of requirements gathering, development team gets only few requirements say about 5 to 6.
c. Development team designs the system, code it, test it and deploy it at the customers place.
d. Once the product is released customer gets the better understanding of his/her expectations.
e. Though the process is like Spiral model but it's different.
f. Limitations are as follows, change in approach & refactoring are the major constraints of RAD. It also involves huge cycles of retesting and regression testing.

3.1.7 Agile Development Model:

a. They are used widely these days due to their dynamic nature & easy adaptability to the situation.
b. User can keep on adding the requirements at any stage of development lifecycle and subsequently development team has accepted those changes.
c. It gives complete freedom to the user to add requirements at any stage of development.
d. The focus is to provide a 'working software' rather than achieving requirements definition.
e. Agile involves many development methodologies. Some of them are as follows,
    a. Scrum
    b. Extreme Programming

        c. Feature driven Development
        d. Test Driven Development

    f. Following are the principles on which Agile works,
        a. Individuals & interactions are more important for ‘fitness of use’.
        b. Releasing ‘working software’ at each stage rather than focussing on customer deliverables.
        c. To deliver working software, customer collaboration is utmost important.
        d. Quick response to the changes required by the customer at any point of time is the key to success for Agile.

3.1.8 Maintenance Development Approach/Model:

a. Major cost of development or may be amount of success of the software development lies in Maintenance phase.
b. New functionalities may be required due to changing business needs.
c. Maintenance activities forms 4 different groups,
    a. Bug fixing: Defects present in the current software are fixed using retesting & regression testing.
    b. Enhancement: New functionalities are added as per the business needs or change in user requirements.
    c. Porting: Old software is brought into new software. This process involves only change in code and not a change in functionalities.
    d. Reengineering: Change in the logic or algorithm, due to changes in the Business environment.

4.Software Quality Control Approach:

As organisations shifts from ‘q’ to ‘Q’, a product is subjected to heavy inspection, rework, sorting, scrapping etc. to ensure that no defects are present in final deliverables to the customer.

‘Q’ organisations concentrate on defect prevention through process improvements. It targets for first-time right. Following diagram shows an improvement process where focus of quality changes gradually,

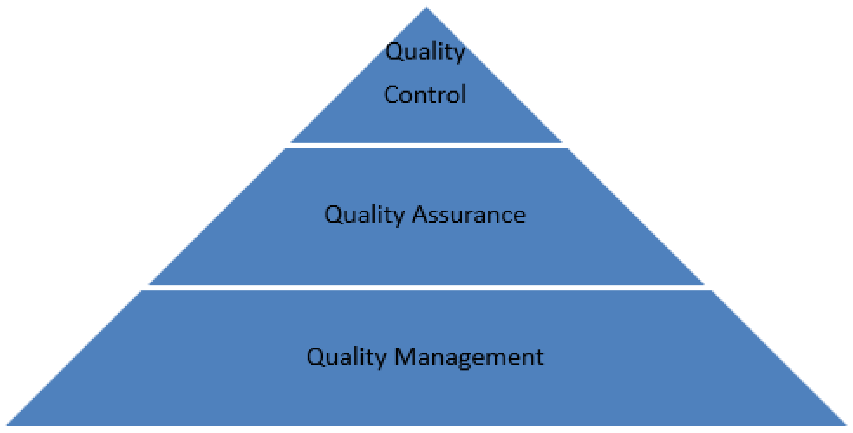


Fig 1.5: Shift in focus from Quality control to Quality Management

1. Quality Control:
    a. This is the oldest approach in engineering, where the product is tested until all the defects are fixed.
    b. This process includes rework, defect fixing, scrap, rework, segregation etc.
    c. All the 'q' organisation follows this approach, and thinks the product is good if it has no defects which goes to the extent of quality improvement until the product delivery has been made & it is delivered to the customer.
2. Quality Assurance Approach:
    a. It is the next stage of improvement of Quality control where the focus shifts from testing & fixing the defects to first-time right.
    b. An organisation becomes a learning organisation as it shifts its approach from 'Quality Control' to 'Quality Assurance'.
    c. They also involve 'root cause analysis' to shift their focus from corrections to corrective actions.
3. Quality Management Approach:
    a. There are 3 kinds of systems in the Universe, they are
        i) <u>Completely closed Systems</u>: It represents that nothing can enter inside the system and nothing can go out of the system.
        ii) <u>Completely Open Systems</u>: It represents a direct influence of universe on the system & vice-versa.
        iii) <u>Systems with semipermeable boundaries</u>: They are the realities as Completely Closed system or completely open systems doesn't exist. Systems with semipermeable boundaries are the realities, which allow the system to get impacted from external changes & also have some effects on external environment.

b. Organisations try to assess the impact of the changes on the system & try to adapt to the changes in the environment to get the benefits.
c. They are highly matured when they implement Quality Management as a management approach.
d. Organisation starts working on defect prevention mechanism for continuous improvement plans.
e. Organisation defines methods, processes & techniques for future technologies & training programs for process improvements.
f. It also involves mentoring, coaching, & guiding people to do better work to achieve organisational objectives.

Questions:

1. What are the factors of Quality?
2. Define Quality. Define Customer's view of Quality.
3. Differentiate between 'Q' organisation & 'q' organisation.
4. Differentiate between Software Quality Assurance and Software Quality Control.
5. Describe any one Software development model.

Further Read:

1. Software Testing Principles, Techniques and Tools, M.G. Limaye, TMH
2. Software testing by Yogesh Singh. Cambridge University Press, 2012
3. Introduction to Software Testing, Paul Ammann, Jeff Offutt, Cambridge University Press.
4. Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, Rex Black, Wiley

2

SOFTWARE QUALITY

2.1: Software Quality Control
- 2.1.1: Software Quality Management
- 2.1.2: Total Quality Management
  - 2.1.2.1: Quality Principles
- 2.1.3: Relationship between Quality & Productivity

2.2: Software Defects
- 2.2.1: Software Testing
  - 2.2.1: Principles
  - 2.2.2: Challenges
  - 2.2.3: Verification
  - 2.2.4: Validation
- 2.2.2: Approaches to Testing
- 2.2.3: Big Bang Approach
- 2.2.4: VV Model

2.3. Questions

2.4. Further Reading

2.1 Software Quality Control

2.1.1 Software Quality Management:

1. Correction: This is the stage where defect is found and is corrected at the same time.
2. Corrective actions: Every defect raised needs be solved using some root cause analysis. Root-cause analysis is initiated to remove root causes so that these kinds of errors does not occur in future.
3. Preventive actions: Since root cause of the problems, other potential weak areas are identified. Preventive action means that there are potential weak areas where defect has not been found till that point, but there exists a probability of finding defect.

2.1.2 Total Quality Management:

This principle intends to view internal and external customers as well as internal and external suppliers for each process, project and for entire organisation. Each supplier eventually becomes a customer at some moment and vice versa. If one can take of his/her customer with an intention to satisfy him, it may result into customer satisfaction & continual improvement for the organisation.

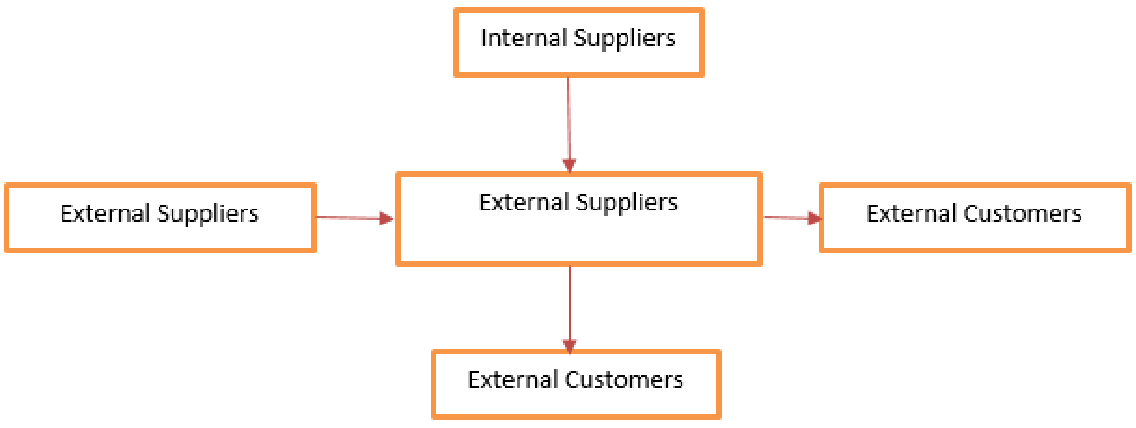


Fig. shows Supply Chain relationship between suppliers & customers

Dr Edward Deming implemented quality management system driven by ‘Total Quality Management’ and ‘Continual Improvement’ in Japanese environment. It resulted into repetitive, cost effective processes with an intention to satisfy customer requirements and achieve customer satisfaction.

2.2.1 Quality principles of ‘Total Quality Management’:

1. Develop constancy of purpose of definition & Deployment of various initiatives: Suppliers & organisation must have an intent to become competitive in the world, to stay in business & to provide jobs to the people & welfare of the society.

2. Adapting to new philosophy of managing people/stakeholders by building confidence and relationships: Management must adopt to the new philosophies of doing work & getting the work done from its people and suppliers.
3. Declare freedom from mass inspection of incoming/produced output: It was believed that one must check everything to ensure that no defect goes to customer.
4. Stop awarding of lowest price tag contracts to suppliers: Organisations must end the practice of comparing unit purchase price as a criterion of awarding contracts.
5. Improve every process used for development and testing of product: Every process of planning, production & service to the customer & other support processes constantly.
6. Institutionalise training across the organisation for all people: It must institute modern methods of training, self-study etc.,
7. Institutionalise leadership throughout organisation at each level: An organisation must adopt and institute leadership at all levels' with the aim of helping people to do their jobs in a better way.
8. Drive out fear of failure from employees: An organisation must encourage effective two-way communication and other means to drive out fear of failure from the minds of all the employees.
9. Break down barriers between functions/departments: People in different areas must work as a team to tackle problems that may be encountered with products and customer satisfaction.
10. Eliminate exhortations by numbers, goals, targets: The organisation shall have methods to demonstrate that the targets can be achieved with smart work.
11. Eliminate arbitrary numerical targets which are not supported by processes: Eliminate work standards that prescribe quotas for the work force and numerical goals for managers to be achieved.
12. Permit pride of workmanship for employees: Remove the barriers that take away the pride of workmanship for workers and management.
13. Encourage Education of new skills and techniques: Advances in competitive position will have their roots in knowledge gained by people during such trainings.
14. Top management commitment and action to improve continually: It is not sufficient that the top management commits for quality and productivity, but employees must also see and perceive their commitment.

2.2.3 Relationship between Quality & Productivity:

Many people feel that more inspection cycles mean finding more defects, fixing defects mean better quality which ultimately gives good quality product. If the processes of development & testing are good, a bad product will not be manufactured in the first place. Thus, quality must improve productivity by reducing wastage.

1. Improvement in Quality directly leads to improved Productivity: All products are the outcome of processes, and good processes must be capable of producing good product at the first instance.
2. The hidden factory producing scrap, rework, sorting, repair and customer complaint is closed: Problems in products can be linked to faulty development processes.
3. Effective way of improving productivity by improving processes: Productivity improvement means improving several good parts produces per unit time and not the part produced per unit time.
4. Quality improvements lead to cost reduction: Quality improves productivity, efficiency and reduces scrap, rework etc.
5. Employee involvement in quality improvement: Management leadership and employee contribution can make an organisation quality conscious while lack of either of the two can create major problems.
6. Proper communication between management & employees is essential: There are huge losses in communication and distortions leading to miscommunication & wrong interpretation.
7. Employees participate and contribute in improvement process: Every employee needs to play a part in implementation of ‘Total Quality Management’ in respective areas of working.
8. Employees share responsibility for innovation & quality improvement: Management provides support, guidance, leadership etc., and employees contribute their part to convert organisations into performing teams.

2.2 Software Defects:

After taking so much of precautions while defining & implementing the processes, doing verification and validation of each artefact during SDLC, yet nobody can claim that the product is free of any defects.

Factors responsible for its success/failure:

1) Miscommunication between different entities such as requirements understanding which leads to defects.
2) Development team is confident about their capabilities and is not ready to accept the mistake.
3) Requirement changes are dynamic.
4) Technologies are responsible for introducing few defects.
5) Customer may not be aware of all requirements.

2.2.1 Software Testing:

Testing is defined as ‘execution of a work product with intent to find a defect’. The primary role of testing is not to demonstrate the correctness of software product, but to expose hidden defects so that they can be fixed. This approach assumes that any amount of testing cannot show that software product is defect free. If there is

no defect found during testing, it can only show that the scenario & test cases used for testing did not discover any defect.

2.2.1 Principles of Software Testing

The basic principles of Software Testing are as follows:

1. Define the expected output or result for each test case executed, to understand if expected & actual output matches or not. Any mismatches may indicate possible defects. Defects can be in the product or test case or test plan.
2. Developers must not test their own programs. No defects can be found in such type of testing as approach-related defects will be difficult to find. Development teams must not test their own products.
3. Inspect the results of each test completely and carefully. It would help in root cause analysis and can be used to find weak processes. This will help in building right processes and improving their capability.
4. Include test cases for invalid or unexpected conditions which are feasible during production.
5. Test the program to see if it does what it is supposed to do or not.
6. Avoid disposable test cases unless the program itself is disposable. Reusability of test case is important for regression.

2.2.2 Challenges of Software Testing

Testing is a challenging job. Challenges are different on different fronts. Major challenges faced by the testing team are as follows:

1. Requirements are not clear: Requirements are not clearly mentioned, sometimes they are incomplete, inconsistent, immeasurable and untestable. All these create problems in defining the test cases and test scenarios.
2. Wrong documentation: Requirements may be wrongly documented by the Business Analyst and wrongly interpreted by System Analyst. They should understand the workflow thoroughly from the customer.
3. Code logic: It may be difficult to capture, often testers are not able to understand the code due to lack of technical knowledge.
4. Error handling: It may be difficult to capture. There are many combinations of errors, and various error messages & controls are required.

2.2.3 Verification: It is a disciplined approach to evaluate whether a software product fulfils the requirements or conditions imposed on them by the standards or processes. It is done to ensure that the processes and procedures defined by the customer and/or organisation for development & testing.

Following are the techniques of Verification:

1. Self-Review: It may not be considered as an official way of review, because it assumes that everybody does a self-check before giving work product for further verification.
2. Peer Review: It is the most informal type of review where an author & a peer are involved. Review records are maintained.
3. Walkthrough: It is a semi-formal type of review as it involves larger teams along with the author reviewing a work product.
4. Inspection: It is a formal review where people external to the team may be involved as inspectors. They are the ‘Subject Matter Experts’ who review the work product.
5. Audit: It is a formal review based on samples. Audits are conducted by the auditors who may or may not be the experts in the given work product.

2.2.4 Validation: It is used to evaluate whether the final built software product fulfils its specific intended use. It is also called as ‘Dynamic Testing’. It must be done by the independent users, functional experts, and black box testers to ensure independence of testing from development activities. It helps in analysing whether the software product meets the requirements as specified in requirement statement.

Following are the levels of validation:

1. Unit Testing
2. Integration Testing
3. Interface Testing
4. System Testing
5. Cause & Effect Graphing
6. Path Expression & Regular Expression

2.2.2 Approaches to Testing

There are many approaches to software testing defined by the experts in software quality and testing. They differ as per the customer requirements, type of the system etc.,

1. **Big-Bang Approach:**
    a. It means testing the software after the development work is completed.
    b. It is also called as ‘System Testing’ or final testing which is done just before the release.
    c. This is the last part of software development as per waterfall model.
    d. It emphasised on black box testing to ensure the requirements as defined and documented in requirement specifications and design specifications are met successfully.

e. In case of Big-Bang approach, software is tested before delivery using the executable or final product.
f. It may not be able to detect all the defects as all possibilities cannot be tested.
g. Sometimes, defects found may not be fixed correctly as analysis and defect fixing can be a problem.

VV Model:

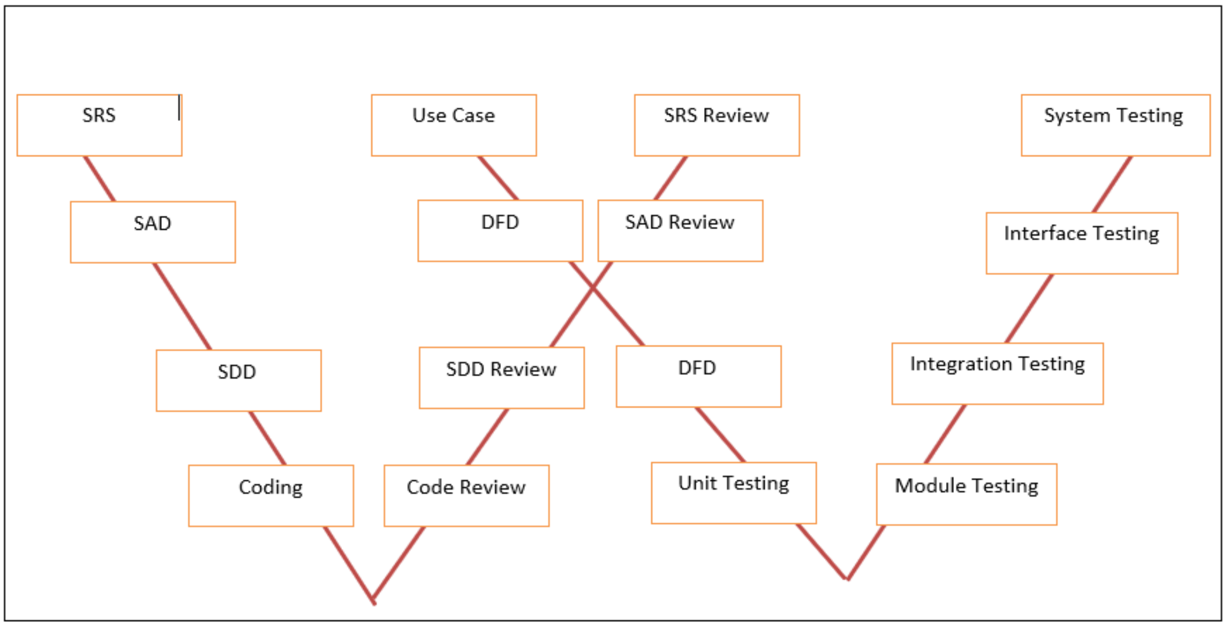


Fig. shows VV Model (Verification & Validation Model)

'V model' is also termed as 'Validation Model' or 'Test Model' as it mainly considers only validation activities associated with software development which are popularly known as 'Testing Activities'.

'VV Model' talks about verification & validation activities associated with software development during entire life cycle.

1. <u>Requirements:</u>
   a. <u>Requirement Verification</u>: It is done through inspection of requirement specification document using checklist, standards or guidelines. Experts in domain, customer representative, and other stakeholders may be involved in conducting such an inspection.
   b. <u>Requirement Validation</u>: It takes place at two or more stages during software development. The first stage of validation involves writing complete use cases by referring to requirement statement. The second stage is system testing. It also done by having 'Review'.
2. <u>Design</u>:
   a. Design Verification: It may be walkthrough of design document by design experts, team members and stakeholders of the project. Project

team along with architect/designer may walkthrough the design to find the completeness & give documents.

  b. Design Validation: It can happen at two or more stages during software development life cycle. The first stage of validation happens when data flow diagrams can be created by referring to the design document. The second stage includes integration testing and interface testing.

3. Coding:
   a. Code verification: As coding is done, it undergoes a code review. Peer review helps in identification of errors with respect to coding standards, indenting standards, commenting standards, and variable declaration issues.
   b. Code validation: It happens through unit testing where individual units are tested separately. The developer may have to write special programs so that individual units can be tested.

Questions:

1. What are the factors of Quality?
2. What is defect? Why defect arises?
3. What are the principles of Software Testing?
4. Define Testing. Why is it necessary?
5. Explain VV Model for testing with a diagram.
6. Explain the Big-Bang approach of testing.
7. State the methods or techniques of Verification.

Further Read:

1. Software Testing Principles, Techniques and Tools, M.G. Limaye, TMH
2. Software testing by Yogesh Singh. Cambridge University Press, 2012
3. Introduction to Software Testing, Paul Ammann, Jeff Offutt, Cambridge University Press.
4. Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, Rex Black, Wiley

2

Functional Testing

Unit Structure

1.0 : Objectives
1.1: Functional Testing
1.1.1: Advantages of functional testing
1.1.2: Disadvantage of functional testing
1.2: Boundary Value Analysis
1.2.1 Limitations of Boundary Value
1.3: Robustness Testing
1.4: Worst Case Testing
1.5: Equivalence Class Testing
1.5.1 Weak Normal Equivalence Class Testing
1.5.2 Strong Normal Equivalence Class Testing
1.5.3 Weak Robust Equivalence Class Testing
1.5.4 Weak Robust Equivalence Class Testing

1.0 OBJECTIVES

After completing this chapter, you will be able to:

- ❖ Understand the concept of Functional testing.
- ❖ Understand the concept of Boundary value analysis and Equivalence partition.
- ❖ Importance and benefits of different testing techniques

1.1 Functional Testing

Testing based on an analysis of the specification of the functionality of a component or
system.
Functional testing includes all kind of tests which verify a system's input–output behavior.
To design functional test cases the black box testing methods are used, and the test bases are the functional requirements.
Functional requirements specify the behavior of the system; they describe "what" the

system must be able to do.
Characteristics of functionality, according to [ISO 9126], are suitability, accuracy, interoperability, and security.
Testing functionality can be done from two perspectives:
requirements-based business-process-based.

Requirements-based testing uses a specification of the functional requirements for the system as the basis for designing tests.
The table of contents of the requirements specification are used as an initial test inventory or list of items to test (or not to test).
The requirements are prioritize based on risk criteria which can be used to prioritize the test cases.
This will ensure that the most important and most critical tests are included in the testing effort.
Business-process-based testing uses knowledge of the business processes.
Business processes describe the scenarios involved in the day-to-day business use of the system.
For example, a personnel and payroll system may have a business employee joins or leaves the company.
Use cases originate from object-oriented development.
They also take the business processes as a starting point, although they start from tasks to be performed by users.
Use cases are a very useful basis for test cases from a business perspective

1.1.1: Advantages of functional testing

Functional testing is an essential step in analysing the performance of a software before it is delivered to the end user.
The shortcomings which can lead to serious consequences can be realized before the software is placed in the real world for use.
To check if the system meets the compliance requirement and regulatory guidelines

1.1.2: Disadvantage of functional testing

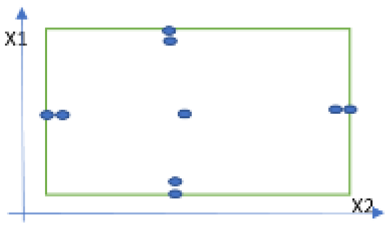
Functional testing focuses on the input and output behaviour of the systems.
It doesn’t includes other performance issues that aren’t directly related to its functions
It doesn’t include functionality to detect logical errors in software.
Possibility of redundant testing.

1.2 Boundary Value Analysis

The BVA depends on the concept that errors tend to occur near the extremities of the input variables
Strongly typed languages like Ada and Pascal is not suitable for Boundary value testing as it requires associated datatypes.

COBOL, Fortran and C is ideal for BVA since they are weakly typed language
Boundary value analysis (BVA) is based on testing at the boundaries between partitions.
Range checking is an example of using the boundary value analysis technique.
BVA concentrated more on the boundary of the input space to identify the test cases
Most of the programs can be viewed as a function F
The input variables of F will have some possibly boundaries where a,b and c,d are the range of x1 and x2 respectively:

$a \leq x1 \leq b$
$c \leq x2 \leq d$
The basic idea for BVA is to use input variable values at their minimum, just above minimum, a nominal value, just below their maximum, ad their maximum
(min, min+, nom, max-, max)
A commercially available tool called as T for BVA



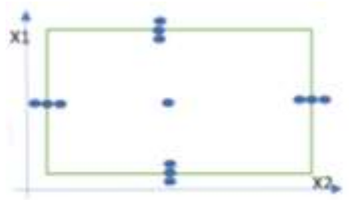
1.2.1 Limitations of Boundary Value

Boundary Value Analysis works efficiently only when the Program to be tested is a “function
of several independent variables that represent bounded physical quantities” [1].
When the required conditions are met BVA works without the problem but when they are not deficiencies in the output can be noticed.
For example the date generator problem would need more attention towards the end of February or on leap years because same test cases would be applied on all months.
The Boundary Value Analysis cannot take into consideration the nature of a function or the dependencies between its variables. This lack of understanding for the variable nature means that BVA can be seen as quite rudimentary

1.3 Robustness Testing

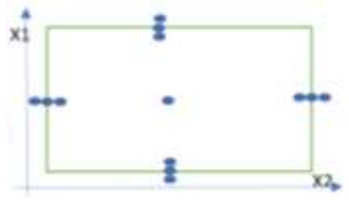
Robustness testing can be seen as an extension of Boundary Value Analysis.

The idea behind Robustness testing is to test for variables that lie in the legitimate input range and variables that fall just outside this input domain.
Two more values for each variable (min-, max+) are added such that it fall just outside of input range
In addition to 5 testing values (min, min+, nom, max-, max), two more values (min-, max+) are added to fall just outside of the input range



1.4 Robustness Testing

Robustness testing can be seen as an extension of Boundary Value Analysis.
The idea behind Robustness testing is to test for variables that lie in the legitimate input range and variables that fall just outside this input domain.
Two more values for each variable (min-, max+) are added such that it fall just outside of input range
In addition to 5 testing values (min, min+, nom, max-, max), two more values (min-, max+) are added to fall just outside of the input range



1.5 Worst Case Testing

Worst cast testing is based on the assumption that if more than one variable has an extreme value.
To generate test cases consider the original 5-tuple set (min, min+, nom, max-, max) and
perform the Cartesian product of these values.

EXAMPLE
Consider a program to classify a triangle. Its inputs is a triple of positive integers (say x, y, z) and the data type for input parameters ensures that these will be integers greater than 0 and less than or equal to 100. The program output may be one of the following words:
[Scalene; Isosceles; Equilateral; Not a triangle]
Design the Boundary value test cases.

Standard boundary value
min =1
min+ = 2
nom = 50
max- = 99
max = 100

Boundary Value Analysis Test Cases				
Case	x	y	z	Expected Output
1	50	50	1	Isosceles
2	50	50	2	Isosceles
3	50	50	50	Equilateral
4	50	50	99	Isosceles
5	50	50	100	Not a triangle
6	50	1	50	Isosceles
7	50	2	50	Isosceles
8	50	99	50	Isosceles
9	50	100	50	Not a triangle
10	1	50	50	Isosceles
11	2	50	50	Isosceles
12	99	50	50	Isosceles
13	100	50	50	Not a triangle

1.6 Equivalence Class Testing

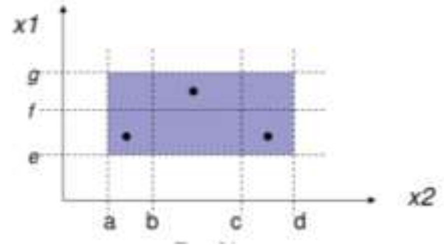
Equivalence class testing is based on creating partitions. It removes the redundancy gaps that appears in the boundary value analysis
Input or output data is grouped or partitioned into sets of data that is expected to behave similarly using an Equivalence relation. An equivalence relation describes how data is going to be processed when it enters a function.
The equivalence class testing requires to test only one condition from each partition. This is because all the conditions in one partition will be treated in the same way by the software.
If one condition in a partition works, then all the conditions in that partition will work, and so there is little point in testing any of these others.
Conversely, if one of the conditions in a partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition
Consider a function of two variables x1,x2 having the following boundaries and intervals within the boundaries:

$a \le x1 \le d$ with interval [a,b)[b,c)[c,d)
$c \le x2 \le d$ with interval [e,f)[f,g)

where closed and open intervals are denoted by square and round parentheses

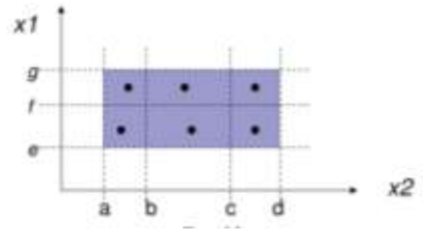
1.6.1 WEAK NORMAL EQUIVALENCE CLASS TESTING

Weak equivalence class testing is based on the single fault assumption. It states that the testing is accomplished by using only one variable from each equivalence partition



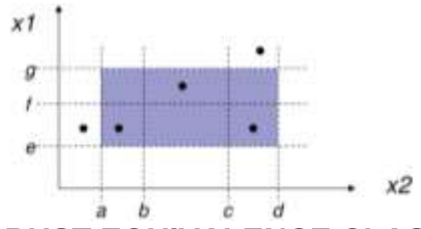
1.6.2 STRONG NORMAL EQUIVALENCE CLASS TESTING

Strong equivalence class testing is based on the multiple fault assumption. The assumption is based on the cartesian product equivalence partitions therefore strong equivalence class testing tests every combination of elements formed as a result of the Cartesian product of the Equivalence relation



1.6.3 WEAK ROBUST EQUIVALENCE CLASS TESTING

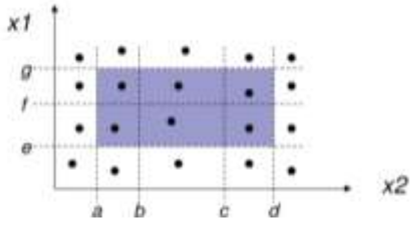
Weak Robust equivalence class testing is also known as the traditional equivalence class testing. The weak part refers to the single fault assumption and robust refers to the consideration of invalid values. Test case will have one invalid value and the remaining values will all be valid from each partition. Problems with weak robust equivalence class testing is that the expected output for the invalid input is not specified



1.6.4 STRONG ROBUST EQUIVALENCE CLASS TESTING

The strong part refers to the multiple fault assumption and robust refers to the consideration of invalid values. This form of Equivalence Class testing produces test cases for all valid and

invalid elements of the Cartesian product of all the equivalence classes.
The main problem with strong robust equivalence is the massive redundancy.
Testers use a specification which tells them what the expected outputs should be when entering test cases into a system. More often, the specification will not identify expected outputs for invalid test cases. This means that testers have to spend excessive time defining expected outputs for invalid test cases



EXAMPLE
Consider a program to classify a triangle. Its inputs is a triple of positive integers (say x, y, z) and the data type for input parameters ensures that these will be integers greater than 0 and less than or equal to 100. The program output may be one of the following words:
[Scalene; Isosceles; Equilateral; Not a triangle]
Design the Equivalence class test cases.
Four possible outputs: Not a Triangle, Isosceles, Equilateral, Scalene
Equivalence partitions
R1 = {x, y, z : 0 .. 100 | equilateral_triangle (<x,y,z>) }
R2 = {x, y, z : 0 .. 100 | isoceles_triangle (<x,y,z>) }
R3 = {x, y, z : 0 .. 100 | scalene_triangle (<x,y,z>) }
R4 = {x, y, z : 0 .. 100 | not_a_triangle (<x,y,z>) }
Condition for triangle : sum of two sides should be greater than the third side.
Weak Normal Test Cases

Test case	x	y	z	Expected output
1	40	40	40	Equilateral
2	40	40	50	Isosceles
3	50	40	30	Scalene
4	20	10	50	Not a triangle

Weak Robust Test Cases

Test case	x	y	z	Expected output
1	-1	5	5	Invalid range
2	5	-1	5	Invalid range
3	5	5	-1	Invalid range
4	201	5	5	Invalid range
5	5	201	5	Invalid range

6	5	5	201	Invalid range

1.7 QUESTIONS

1. Explain equivalence class testing. How does weak normal differ from strong normal form of equivalence class testing?
2. What is functional Testing? Explain its advantages and disadvantages.
3. Differentiate between strong equivalence class testing and weak equivalence class testing.
4. What is boundary value testing? List all the limitations of boundary value testing.

1.8 FUTHER READING

- ❖ **Software testing a Craftsman's Approach** by Paul C Jorgensen
- ❖ **Software Testing** by Yogesh Singh University Press 2012
- ❖ http://www.cs.swan.ac.uk/
- ❖ http://softwaretestingfundamentals.com/functional-testing/
- ❖ https://dzone.com/articles/what-is-functional-testing-types-tips-limitations

❖❖❖❖

4

Decision Table

Unit Structure

1.0 : Objectives
1.1: Decision Table
1.1.1: Advantages of Decision base testing
1.1.2: Decision Table creation using triangle problem
1.2: Retrospective on Functional testing
1.2.1 Test Efforts
1.2.2 Test Efficiency
1.2.3 Test Effectiveness

1.0 OBJECTIVES

After completing this chapter, you will be able to:

- ❖ Understand the concept of Decision based testing.
- ❖ Retrospect functional testing.

1.1 DECISION TABLE – BASED TESTING

The techniques of equivalence partitioning and boundary value analysis are often applied to specific situations or inputs.
A decision table is a good way to deal with combinations of things (e.g. inputs).
This technique is sometimes also referred to as a 'cause-effect' table.
If different combinations of inputs result in different actions being taken, this can be more difficult to show using equivalence partitioning and boundary value analysis, which tend to be more focused on the user interface.
Decision tables – based testing is more focused on business logic or business rules
Decision tables provide a systematic way of stating complex business rules, which is useful for developers as well as for testers.

1.1.1: Advantages of Decision based testing

Decision tables can be used in test design whether or not they are used in specifications, as they help testers explore the effects of combinations of different inputs and other software states that must correctly implement business rules.
Helping the developers do a better job can also lead to better relationships with them
Decision tables aid the systematic selection of effective test cases and can have the beneficial side-effect of finding problems and ambiguities in the specification.
It is a technique that works well in conjunction with equivalence partitioning.

COMPONENT OF DECISION – BASED TESTING:

STUB PORTION	CONDITION	ENTRY PORTION
	ACTION	

Stub portion: the left most column
Entry: the right most column
Condition: noted by c’s
Action: noted by a’s, a column in the entry portion is the rule
Thus there are condition stubs, condition entries, action stubs and action entries.
A column in the entry portion is called rule, Rules indicate which actions
In figure given the inputs in this given table derive the outputs depending on what conditions these inputs meet. The table consist of ‘Don’t care entries’ normally viewed as being false values which don’t require the value to define the output. Tables using binary conditions are known as limited entry decision tables and the one using multiple conditions are known as extended entry decision tables.

Condition	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Condition1	T	T	F	F
Condition1	T	F	T	F
Action1	✓	✓	✗	✗
Action2	✗	✓	✓	✓

1.1.2 Decision table creation using Triangle Problem

Decision table stub conditions and the type of decision table are important while creating a Decision table. Limited Entry decision tables are easier to create than extended entry tables. Step One – List All Stub Conditions
Three inputs are taken
Conditional checks are performed to calculate if it’s a triangle

If so then what type of triangle it is.
The first condition must check whether all 3 sides constitute a triangle, no other checks to be performed if the answer is false.
Then the remainder of the conditions will check whether the sides of the triangles are equal or not. As there are only three sides to a triangle means that we have three conditions when checking all of the sides.
So the condition stubs for the table would be:

- a, b, c form a triangle?
- a = b?
- a = c?
- a = c?

Step Two – Calculate the Number of Possible Combinations (Rules)
Use the following formula:
Number of Rules = 2Number of Condition Stubs
Number of Rules = 24 = 16
So there are 16 possible combinations in the decision table.

Step Three – Place all of the Combinations into the Table

C1: a,b,c forms a triangle?	**N**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**
C2: a=b?	-	Y	Y	Y	Y	N	N	N	N
C3: a=c?	-	Y	Y	N	N	Y	Y	N	N
C4: b=c?	✓	✓	X	X	X	X	X	X	X

Step Four – Check Covered Combinations
Precautionary step to check for errors and redundant and inconsistent rules.

Step Five – Fill the Table with the Actions
For the final step of creating a decision table fill the Action Stub and Entry sections of the table.
After completing the decision table and adding the actions.
Each action stub is exercised once, the “impossible” action is also added into the table

C1: a,b,c forms a triangle?	**N**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**
C2: a=b?	-	Y	Y	Y	Y	N	N	N	N
C3: a=c?	-	Y	Y	N	N	Y	Y	N	N
C4: b=c?	-	Y	N	Y	N	Y	N	Y	N
A1: Not a triangle	X								

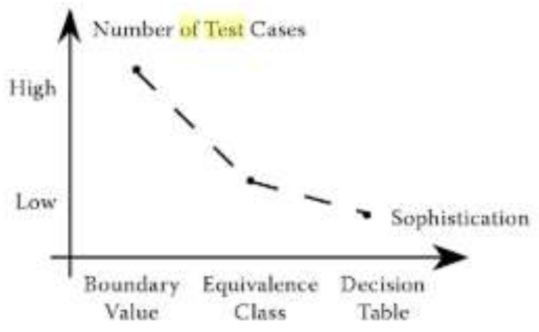
A2: Scalene									X
A3: Isosceles					X		X	X	
A4: Equilateral		X							
A5: Impossible			X	X		X			

1.2 Retrospective on Functional Testing

Retrospective means “looking back or dealing with past events or situations”.
Basic purpose of retrospective is to find out: What went well, What didn’t went well, Improvement Areas
Retrospective of function Testing is useful if there is an important issue that the tester need to understand correctly, or if the test cases has behaved in a way that the tester cannot understand.

1.2.1Test efforts

Boundary value technique have no recognition of data or logical dependencies. It is very mechanical in the way it generate test cases, because of this they are very easy to automate
The equivalence class techniques pay attention to data dependencies and to the function itself
More thought process is required to identify the equivalence class after that the process is also mechanical
The decision table technique is the most sophisticated because it required to concentrate on both data and logical dependencies



1.2.2 Testing efficiency

The intuitive notion is that a set of test cases is right that is there is no gaps and no redundancy
When several test cases with same purpose is considered, redundancy is obersved
Detecting gaps are harder if only functional testing been used

The best method will be to perform two methods of testing. In general more sophisticated methods will help to recognise gap with respect to the specification

1.2.3 Testing effectiveness

The easy choice is to be dogmatic: mandate a method, use generate test cases and then run the test cases. This is an absolute method and conformity is measurable

There exist twin possibilities of gaps of untested functionality and redundant tests

1.3 QUESTIONS

1. A rectangle program accepts four integers as lengths for four sides of length from 1 to 100, inclusively. The output of the program is to determine whether the inputted numbers can form a rectangle, square or neither of these. Create a decision table for the above problem with any five rules. Provide a test case for each given rule.With the help of an illustrative example, explain how decision table can be used for testing.
2. Explain decision tables with an example.
3. Discuss the advantages and disadvantages of decision table
4. What is retrospection? Explain with reference to functional testing.

1.4 FUTHER READING

❖ **Software testing a Craftsman's Approach** by Paul C Jorgensen

❖ **Software Testing** by Yogesh Singh University Press 2012

❖ http://www.cs.swan.ac.uk/

❖ http://softwaretestingfundamentals.com/functional-testing/

❖ https://dzone.com/articles/what-is-functional-testing-types-tips-limitations

❖❖❖❖

5

Path Testing

Unit Structure

1.0 : Objectives
1.1 : Introduction
1.2 : Definition
1. 3: Program graph construction
    1.3.1: Unstructured Graph
1.4: DD-Path
1.5: Test Coverage Metrics
    1.5.1: Metric-Based Testing
    1.5.2: Test Coverage Analyzers
1.6: Basic Path Testing
    1.6.1: McCabe’s Basic Path Method
1.7: Essential Complexity
1.8 Questions
1.9 Further Reading

1.0 OBJECTIVES

After completing this chapter, you will be able to:

- ❖ Understand the concept of path testing.
- ❖ Understand the concept of coverage metrics for testing .

1.1 Introduction

The characteristics of structural programming method is that, it is based on source code of the program and not on the specification
Path testing is a structural testing method that involves using the source code of a program to attempt to find every possible executable path.

The idea is to test each individual path in as many ways as possible in order to maximise the coverage of each test case.
This gives the best possible chance of discovering all faults within a piece of code.
The fact that path testing is based upon the source code of a program means that it is a white box testing method.

1.2 Definition

Program graph is a directed graph in which nodes are statement fragments and edges represent flow of control

1.3 Program graph construction

- Program graphs are a graphical representation of a program's source code.
- The nodes of the program graph represent the statement fragments of the code, and the edges represent the program's flow of control.
- Figure 1.1 shows pseudocode for a simple program that simply subtracts two integers and outputs the result to the terminal.
- The number subtracted depends on which is the larger of the two; this stops a negative number from being output.

```
1. Program 'Simple Subtraction'
2. Input (x, y)
3. Output (x)
4. Output (y)
5. If x > y then DO
6. x – y = z
7. Else y – x = z
8. EndIf
9. Output (z)
10. Output "End Program"
```

Figure 1.1 Pseudocode for the simple subtraction program.

The construction of a program graph for this simple code is a basic task.
Each line number is used to enumerate the relevant nodes of the graph.
It is not necessary to include basic declarations and module titles in the program graph, and so line 1 of the pseudocode in Figure 1.1 will be ignored.
For a path to be executable it must start at line 2 of the pseudocode, and end at line 10. These nodes are known as source and sink node respectively.
The importance of the program graph is that program executions correspond to paths from source to the sink nodes

It depicts the relationship between the test case and the part of the program it exercises.
The figure 1.2 is the graph of the simple program
Starting at the source node and ending at the sink node, there exist two possible paths.
The first path would be the result of the If-Then clause being taken, and the second would be the result of the Else clause being taken.
Nodes 2 through to 4 and nodes 9 to 10 are sequences. This means that these nodes represent simple statements such as variable declarations, expressions or basic input/output commands. Nodes 5 through to 8 are a representation of an if then-else construct,

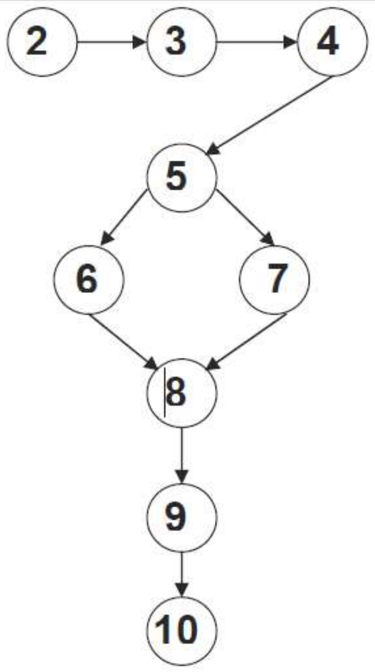


Figure 1.2 A program graph of the simple subtraction program

1.3.1 Unstructured Graph

Program graph would be much more complex to test, solely because if it is unstructured.
The reason behind this lack of structure is due to the program graph containing a loop construct in which there exists internal branching.
As a result, if the loop from node G to node A had 18 repetitions, it would see the number of distinct possible execution paths rise to 4.77 trillion [Jorgensen, 2002].
This demonstrates how an unstructured program can lead to difficulties in even finding every possible path, while testing each path would be an infeasible task.

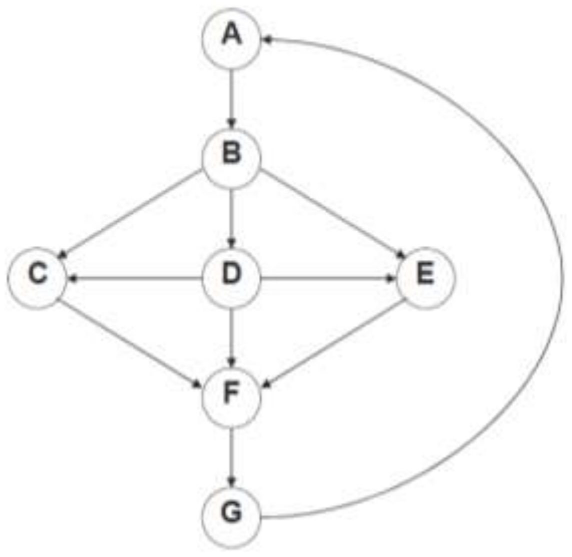


Figure 1.3 A simple yet unstructured graph

1.4 DD-Path

The best known form of structural testing is based on construct known as decision to decision path

The reason that program graphs play such an important role in structural testing is because it forms the basis of a number of testing methods, including one based on a construct known as decision-to-decision paths (DD-Paths).

The idea is to use DD-Paths to create a condensation graph of a piece of software's program graph, in which a number of constructs are collapsed into single nodes known as DD-Paths.

DD-Paths are chains of nodes in a directed graph that adhere to certain definitions.

Each chain can be broken down into a different type of DD-Path, the result of which ends up as being a graph of DD-Paths. The length of a chain corresponds to the number of edges that the chain contains.

The definitions of each different type of DD-Path that a chain can be reduced to are given as follows:

Type 1: A single node with an in-degree = 0.
Type 2: A single node with an out-degree = 0.
Type 3: A single node with in-degree >= 2 or out-degree >= 2.
Type 4: A single node with in-degree = 1 and out-degree = 1.
Type 5: The chain is of a maximal length >=1.

All programs must have an entry and an exit and so every program graph must have a source and sink node.

Type 1 and Type 2 are needed to provide us with the capability of defining these key nodes as initial and final DD-Paths.

Type 3 deals with slightly more complex structured constructs that often appear in a program graph such as If-Then-Else statements and Case statements.

Type 4 allows for basic nodes such as expressions and declarations to be defined as DD-Paths.

Type 5 is used to take chains of these nodes and condense them into a single node. The definition of a Type 5 DD-Path must examine the maximal length of the chain.

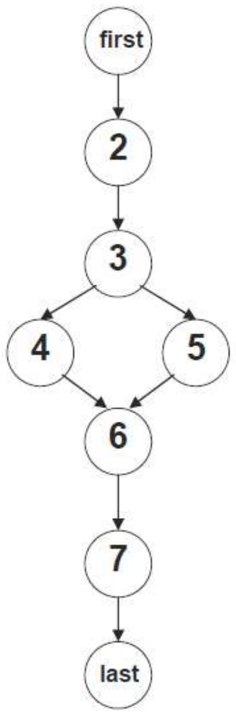


Figure 1.4 A DD-Path graph of the simple subtraction program

The differences in the program graph of Figure 1.1 and its DD-Path graph of figure 1.4 can be easily identified.

The source and sink nodes of the graph have been replaced by the words ‘first’ and ‘last’ in order to identify the nodes that conform to Type 1 and Type 2 DD-Paths.

Perhaps more interestingly, there exists one less node. This is due to the fact that nodes 3 and 4 in the original program graph were a chain of maximal length >=1, and so they have been condensed into a single node in the DD-Path graph

There also exist similarities between the two graphs. Node 7 remains unchanged while the If-Then-Else construct is still visible. Nodes 3 and 6 obey the Type 3 definition, while nodes 4 and 6 are simply chains of length 1 and so are defined as Type 4 DD-Paths.

DD-Path graph presents testers with all possible linear code sequences. Test cases can be set up to execute each of these sequences, i.e all paths within the DD-Path graph of the program can be tested. As a result, DD-Paths can be used as a test coverage metric

1.5 Test Coverage Metrics

Functional testing has the problem of gaps. This is to say that functional methods such as boundary value testing only checks maximum, minimum and nominal values, thus leaving areas of code untested for faults.
Structural testing methods attempt to prevent this by having a number of different test coverage metrics which shows the tester degree to which a program has
been tested.
Structured Test Coverage Metrics

C_0	Every Statement
C_1	Every DD path(predicate outcome)
C_{1p}	Every predicate to each outcome
C_2	C1 coverage + loop coverage
C_d	C1 coverage +every dependent pair of DD path
C_{MCC}	Multiple condition coverage
C_{ik}	Every program path that contains upto k repetition of a loop
C_{stat}	Statistically significant fraction of path
C_{∞}	All possible execution path

1.5.1 Metric-Based Testing

DD-Path Testing
When every DD-path is traversed (the C1 metric), each predicate outcome has been executed; this amounts to traversing every edge in the DD-path graph.
For if–then and if–then–else statements, this means that both the true and the false branches are covered (C1p coverage).

Dependent Pairs of DD-Paths
Identification of dependencies must be made at the code level..
The most common dependency among pairs of DD-paths is the define/reference relationship, in which a variable is defined

(receives a value) in one DD-path and is referenced in another DD-path. The importance of these dependencies is that they are closely related to the problem of infeasible paths.

Multiple Condition Coverage
Instead of simply traversing such predicates to their true and false outcomes, investigating the different ways that each outcome can occur.
One possibility is to make a decision table; a compound condition of three simple conditions will have eight rules yielding eight test cases.
Another possibility is to reprogram compound predicates into nested simple if–then–else logic, which will result in more DD-paths to cover.
Multiple condition coverage assures that this complexity is not swept under the DD-path coverage

Loop Coverage
Loop coverage has been the highly fault prone portion of source code. Concatenated loops are simply a sequence of disjoint loops while nested loops are on contained in another. Loop testing involves a decision and both outcomes of the decisions are supposed to be tested. While traversing the loop and while exiting or entering the loop. Once the loop has been tested, the tester condenses it to single node. If loops are nested, the process is repeated starting with the innermost loop and working outward.

1.5.2 Test Coverage Analyzers

Coverage analyzers are a class of test tools that offer automated support for this approach to testing management. The tester runs a set of test cases on a program that has been instrumented by the coverage analyzer. This information is used for generating reports.

1.6 Basic Path Testing

The basis is always define in terms of vector space, which is a set of element as well as operations that corresponds to multiplication and addition defined for the vectors.

The basis of a vector space contains a set of vectors that are independent of one another, and have a spanning property; this means that everything within the vector space can be expressed in terms of the elements within the basis.

1.6.1 McCabe’s Basic Path Method

In the 1970's, Thomas McCabe came up with the idea of using a vector space to carry out path testing. What McCabe noticed was that if a basis could be provided for a program graph, this basis could be subjected to rigorous testing; if proven to be without fault, it could be assumed that those paths expressed in terms of that basis are also correct.

The method devised by McCabe to carry out basis path testing has four steps. These are:

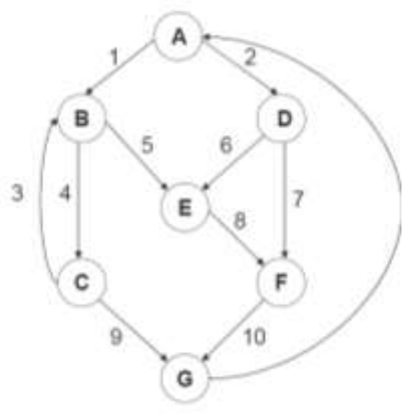
1. Compute the program graph.
2. Calculate the cyclomatic complexity.

3. Select a basis set of paths.

4. Generate test cases for each of these paths

Figure given below shows an example taken from [McCabe, 1982]. This is a commonly used example, as it demonstrates how the basis of a graph containing a loop is computed.

It should be noted that the graph is strongly connected; that is, there exists an edge from the sink node to the source node.



McCabe's Strongly connected program graph

The cyclomatic complexity of a strongly connected graph is provided by the formula V(G) = e – n + p.

The number of edges is represented by e, the number of nodes by n and the number of connected areas by p. If we apply this formula to the graph given, the number of linearly independent circuits is:

V(G) = e – n + p

= 11 – 7 + 1 = 5

An independent path is any path through the software that introduces at least one new set of processing statements or a new condition [Pressman, 2001].

To find these paths, McCabe developed a procedure known as the baseline method [McCabe, 1996].

The procedure works by starting at the source node. From here, the leftmost path is followed until the sink node is reached.

This provides us with the path A, B, C, G. We then repeatedly retrace this path from the source node, but change our decisions at every node with out-degree >= 2, starting with the decision node lowest in the path.

For example, the next path would be A, B, C, B, C, G, as the decision at node C would be 'flipped'.

The third path would then be A, B, E, F, G, as the next lowest decision node is B. Two important points should be made here.

Firstly, if there is a loop, it only has to be traversed once, or else the basis will contain redundant paths.

Secondly, it is possible for there to be more than one basis; the property of uniqueness is one not required.

The five linearly independent paths of our graph are as follows:

Path 1: A, B, C, G.
Path 2: A, B, C, B, C, G.
Path 3: A, B, E, F, G.
Path 4: A, D, E, F, G.
Path 5: A, D, F, G.

1.7 Essential Complexity

When carrying out his work on the basis path testing method, McCabe developed the notion of what is now known as essential complexity.

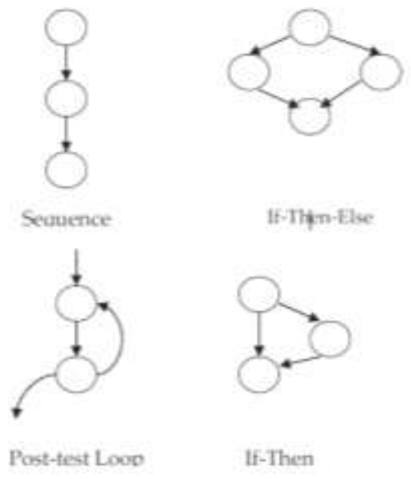
This is the term given for using the cyclomatic complexity to produce a condensation graph;the result is a graph that can be used to assist in both programming and the testing procedure.

The concept behind essential complexity is that the program graph of a piece of software is traversed until a structured programming construct is discovered;

Once located, the structured programming construct is collapsed into a single node and the graph traversal continues.

The desired outcome of this procedure is to end up

with a graph of V(G) = 1, that is, a program made up of one node. This will mean that the entire program is composed of structured programming constructs. If a graph cannot be reduced to one in which there is a cyclomatic complexity of 1, then it means that the program must contain an unstructured programming construct.



McCabe’s Structured Constructs

1.8 QUESTIONS

1. Write short note on metric based testing
2. What is basic path testing? Explain Mc’Cabe cyclomatic complexity
3. Explain DD path and DD path graph
4. Write short note on DU path test coverage metric

5. Write short note on Path testing coverage metrics

1.9 FUTHER READING

❖ **Software testing a Craftsman's Approach** by Paul C Jorgensen

❖ **Software Testing** by Yogesh Singh University Press 2012

❖ http://www.cs.swan.ac.uk/

❖ http://softwaretestingfundamentals.com/functional-testing/

❖ https://dzone.com/articles/what-is-functional-testing-types-tips-limitations

❖❖❖❖

6

Dataflow Testing

Unit Structure

1.0 : Objectives
1.1 : Introduction
1.2 : Define/Use testing
1. 3: Slice based testing
1.4 Questions
1.5 Further Reading

1.0 OBJECTIVES

After completing this chapter, you will be able to:

- ❖ Understand the concept of Data and Information.
- ❖ Differentiate between the Analog Verses digital Signals.
- ❖ Deal with the different number system in arithmetic.
- ❖ Understand the number system conversions.
- ❖ Solve the Arithmetic examples based on Binary arithmetic.

1.1 Introduction

Data flow testing can be considered to be a form of structural testing: in contrast to functional testing, where the program can be tested without any knowledge of its internal structures, structural testing techniques require the tester to have access to details of the program's structure. Data flow testing focuses on the variables used within a program.

Variables are defined and used at different points within the program; data flow testing allows the tester to chart the changing values of variables within the program. It does this by utilising the concept of a program graph: in this respect, it is closely related to path testing, however the paths are selected on variables.

Variables have been seen as the main areas where a program can be tested structurally. Early methods of data testing involved static

analysis: the compiler produces a list of lines at which variables are defined or used.
The term static analysis refers to the fact that the tester does not have to run the program to analyse it.
Static analysis allows the tester, according to Jorgensen, to focus on three "define/reference anomalies" [1]:
"A variable that is defined but never used (referenced).
A variable that is used but never defined.
A variable that is defined twice before it is used."

1.2 Define/Use testing

Define/Use testing uses paths of the program graph, linked to particular nodes of the graph that relate to variables, to generate test cases.
The term "Define/Use" refers to the two main aspects of a variable: it is either defined (a value is assigned to it) or used (the value assigned to the variable is used elsewhere – maybe when defining another variable).
Define/use testing is meant for use with structured programs. The program is referred to as P, and its graph as G(P). The program graph has single entry and exit nodes, and there are no edges from a node to itself.
The set of variables within the program is called V, and the set of all the paths within the program graph P(G) is PATHS(P).

Defining nodes, referred to as DEF(v, n): Node n in the program graph of P is a defining node of a variable v in the set V if and only if at n, v is defined. For example, with respect to a variable x, nodes containing statements such as "input x" and "x = 2" would both be defining nodes.

Usage nodes, referred to as USE(v, n): Node n in the program graph of P is a usage node of a variable v in the set V if and only if at n, v is used. For example, with respect to a variable x, nodes containing statements such as "print x" and "a = 2 + x" would both be usage nodes.

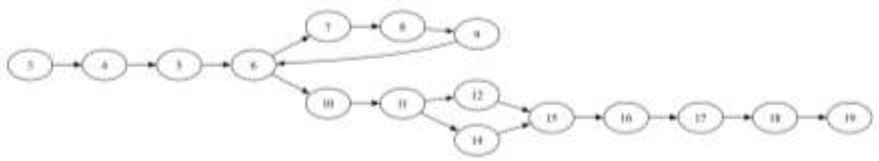
Usage nodes can be split into a number of types
The two major types of usage node are:
• **P-use**: predicate use – the variable is used when making a decision (e.g.
if b > 6).
• **C-use**: computation use – the variable is used in a computation (for
example, b = 3 + d – with respect to the variable d).

Definition-use (du) paths: A path in the set of all paths in P(G) is a du-path for some variable v (in the set V of all variables in the program) if and only if there exist DEF(v, m) and USE(v, n) nodes such that m is the first node of the path, and n is the last node.

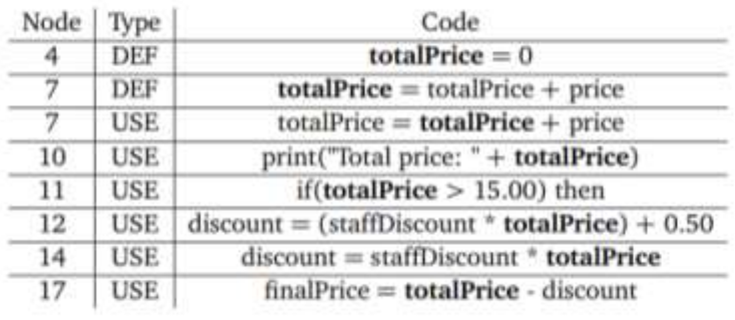
Example

The source code, written in pseudocode, for a program which has been written to perform the task

```
1 program Example()
2 var staffDiscount, totalPrice, finalPrice, discount, price
3 staffDiscount = 0.1
4 totalPrice = 0
5 input(price)
6 while(price != -1) do
7 totalPrice = totalPrice + price
8 input(price)
9 od
10 print("Total price: " + totalPrice)
11 if(totalPrice > 15.00) then
12 discount = (staffDiscount * totalPrice) + 0.50
13 else
14 discount = staffDiscount * totalPrice
15 fi
16 print("Discount: " + discount)
17 finalPrice = totalPrice – discount
18 print("Final price: " + finalPrice)
19 endprogram
```



Program graph for the code



Node	Type	Code
4	DEF	**totalPrice** = 0
7	DEF	**totalPrice** = totalPrice + price
7	USE	totalPrice = **totalPrice** + price
10	USE	print("Total price: " + **totalPrice**)
11	USE	if(**totalPrice** > 15.00) then
12	USE	discount = (staffDiscount * **totalPrice**) + 0.50
14	USE	discount = staffDiscount * **totalPrice**
17	USE	finalPrice = **totalPrice** - discount

The defining and usage nodes for the variable totalPrice

• Defining nodes:
– DEF(price, 5)
– DEF(price, 8)
• Usage nodes:
– USE(price, 6)
– USE(price, 7)

Therefore, there are four du-paths:
• <5, 6>
• <5, 6, 7>
• <8, 9, 6>
• <8, 9, 6, 7>
All of these paths are definition-clear, so they are all dc-paths

1.3 Sliced Based Testing

Part of the utility and versatility of program slices is due to the natural, intuitively clear intent of the concept.
Program slice is a set of program statements that contributes to, or affects the value of, a variable at some point in a program. This notion of slice corresponds to other disciplines as well.
A program P that has a program graph G(P) and a set of program variables V.
Definition:
Given a program P and a set V of variables in P, a slice on the variable set V at statement n, written S(V, n), is the set of all statement fragments in P that contribute to the values of variables in V at node n.

Program slices use the notation S(V, n), where S indicates that it is a program slice, V is the set of variables of the slice and n refers to the statement number (i.e. the node number with respect to the program graph) of the slice.
So, for example, with respect to the price variable given in the example in section 2, the following are slices for each use of the variable:
• S(price, 5) = {5}
• S(price, 6) = {5, 6, 8, 9}
• S(price, 7) = {5, 6, 8, 9}
• S(price, 8) = {8}
To generate the slice S(price, 7), the following steps were taken:

• Lines 1 to 4 have no bearing on the value of the variable at line 7 (and,for that matter, for no other variable at any point), so they are not added to the slice.
• Line 5 contains a defining node of the variable price that can affect the value at line 7, so 5 is added to the slice.
• Line 6 can affect the value of the variable as it can affect the flow of control of the program. Therefore, 6 is added to the slice.
Line 7 is not added to the slice, as it cannot affect the value of the variable at line 7 in any way.
• Line 8 is added to the slice – even though it comes after line 7 in the program listing. This is because of the loop: after the first iteration of the loop, line 8 will be executed before the next

execution of line 7. The program graph in figure 1 shows this in a clear way.

• Line 9 signifies the end of the loop structure. This affects the flow of control (as shown in figure 1, the flow of control goes back to node 6).This indirectly affects the value of price at line 7, as the value stored in the variable will have almost certainly been changed at line 8. Therefore, 9 is added to the slice.

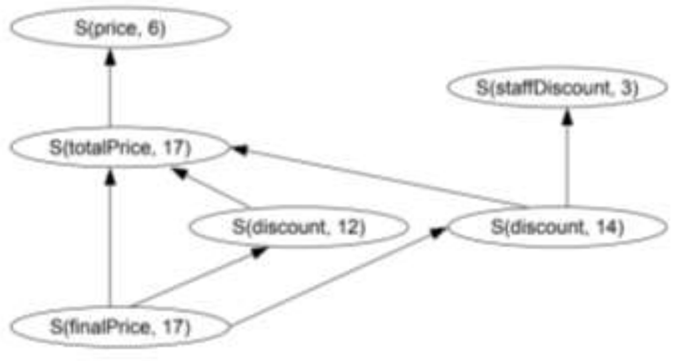
• No other line of the program can be executed before line 7, and so cannot affect the value of the variable at that point. Therefore, no other line is added to the slice.

he program slice, as already mentioned, allows the programmer to focus specifically on the code that is relevant to a particular variable at a certain point. However, the program slice concept also allows the programmer to generate a lattice of slices: that is, a graph showing the subset relationship between the different slices. For instance, looking at the previous example for the variable price, the slices S(price, 5) and S(price, 8) are subsets of S(price,7)

With respect to a program as a whole, certain variables may be related to the values of other variables: for instance, a variable that contains a value that is to be returned at the end of the execution may use the values of other variables in the program. For instance, in the main example in this document,the finalPrice variable uses the totalPrice variable, which itself uses the price variable. The finalPrice variable also uses the discount variable, which uses the staffDiscount and totalPrice variables – and so on.

Therefore, the slices of the totalPrice and discount variables are a subset of the slice of the finalPrice variable at lines 17 and 18, as they both contribute to the value. This subset relationship ‘ripples down’ to the other variables, according to the use-relationship described.This is shown visually in the following example:

• S(staffDiscount, 3) = {3}
• S(totalPrice, 4) = {4}
• S(totalPrice, 7) = {4, 5, 6, 7, 8}



• S(totalPrice, 11) = {4, 5, 6, 7, 8}
• S(discount, 12) = {3, 4, 5, 6, 7, 8, 11, 12}
• S(discount, 14) = {3, 4, 5, 6, 7, 8, 13, 14}
• S(finalPrice, 17) = {3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 17}

1.4 QUESTIONS

1. Write short note on Dataflow testing and how to find DD path?
2. Explain the significance of dataflow testing
3. Explain DD path and DD path graph
4. Why and how the dataflow testing is carried out?
5. Explain Slice based testing

1.5 FUTHER READING

❖ **Software testing a Craftsman's Approach** by Paul C Jorgensen

❖ **Software Testing** by Yogesh Singh University Press 2012

❖ http://www.cs.swan.ac.uk/

❖ http://softwaretestingfundamentals.com/functional-testing/

❖ https://dzone.com/articles/what-is-functional-testing-types-tips-limitations

❖❖❖❖

7

Object Oriented Testing Part 1

Unit Structure

1.0: Objectives
1.1: Units for Object Oriented Testing
1. 2: Implications of Composition and Encapsulation
1.3: Implications of Inheritance
1.4: Levels of Object Oriented Testing
1.5. GUI Testing
1.6: Class Testing
1.7: Questions
1.8 Further Reading

1.0 OBJECTIVES

After completing this chapter, you will be able to:

- ❖ Understand the concept of Object oriented testing.
- ❖ Differentiate between the levels of testing .
- ❖ Understand the concept of class testing and GUI testing.

1.1 Units for Object Oriented testing

Definition:

A unit is a smallest software component that can be complied and executed.

A unit is a software component that would never be assigned to more than one designer to develop.

An object oriented unit is a work of one person which likely ends up as a subset of a class operation

1.2 Implications of Composition and Encapsulation

Composition:

Is the central design strategy in object oriented software development. With the goal of reuse, composition creates the need for very string unit testing.

Encapsulation:
Has potential to resolve this concern but only if the units are highly cohesive and very loosely coupled

1.3 Implications of Inheritance

Inheritance is one of the foundations of the object orientated paradigm, the basics for this idea is one class is able to inherit the functionality of another class.
This gives rise to the ideas of a superclass as well as a subclass, where a superclass refers to the class which another class is inheriting from, and where this inheriting class a subclass
A flattened class is an original class expanded to include all the attributes and operations it inherits
Unit testing on flattened class solves the problem of inheritance.
Flattened class will not be the part of final system so some uncertainty remains.
Figure 1.3.1 shows a UML inheritance diagram of Simple Automated Teller Machine (SATM) system; Both checking and savings accounts have account numbers and balances, and these can be accessed and changed. Checking accounts have a per-check processing charge that must be deducted from the account balance. Savings accounts draw interest that must be calculated and posted on some periodic basis.
If the checkingAccount and savingsAccount classes did not flatten, the access to the balance attributes would not have been possible, and we would not be able to access or change the balances. This is clearly unacceptable for unit testing. The “flattened” checkingAccount and savingsAccount classes is shown in the next figure 1.3.1. These are clearly stand-alone units that are sensible to test.

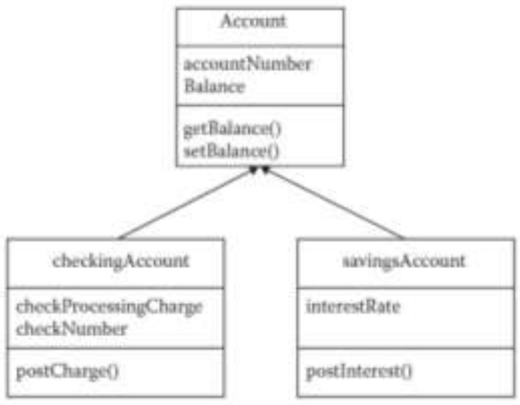


Figure 1.3.1 UML inheritance

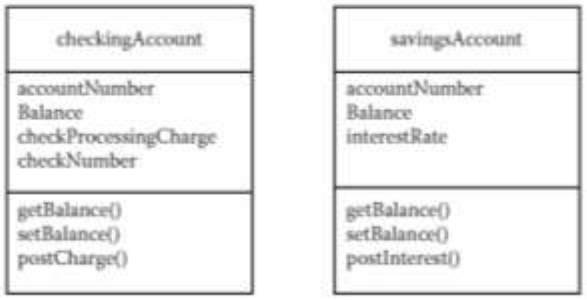


Figure 1.3.2 Flattened classes

1.4 Levels of Object Oriented Testing

If individual operations are or methods are considered to be units, there are four levels : operations/method, class, integration and system testing.

Method
In this level the individual methods are tested to ensure that they are working as they are intended to by the specification.
Although the intended testing is just for one method, it may be that other methods and variables may have to be included in the testing, to facilitate the testing of current method.
This might be the case for instance if you are testing a method that calls one or more other methods within the same class,
This then brings up the need for stub, which are replacements for the real methods/variables which are often static in nature and simply return values that are known to be true by the tester.
This then negates any problems with the testing of a method which calls untested methods, but of course then puts more of the burden on a detailed specification and the accuracy of the stub methods created by the tester.

Class Level
Class is tested as a whole, with each of the methods being tested to ensure when a class is instantiated that the methods work correctly with the given inputs.
Other classes might also be include during the testing of a class, since classes are dependent upon another class. It also requires stub classes.

Integration level
The focus of the testing at this level is in finding errors in classes. Faults which are caused during communication between different classes are identified. It also ensure that there is a minimum chance of these faults in system testing.

System Level

One of the main aims is to find errors that can only be found when dealing with the application as a whole. Another aim of system testing is to ensure that the application that has finally been developed have all the functionality required as per the specification from the view point of a potential end user, which essentially comes down to the question whether the application is finished and can be handed over to the end user

1.5 GUI Testing

Graphical user interface has become closely associated with object oriented software. GUI is an event driven system which are vulnerable to the problem of an infinite number of event sequences

1.6 Class Testing

1.6.1 Method as Units

A method is equivalent to a procedure. Unit testing of procedural code requires stuns and driver test program to supply test cases and record results.

1.6.2 Classes as Units:

Treating a class as a unit solves the intra class integration problem, but it creates other problems. One has to do with various views of a class. In the static view, a class exists as source code. This is fine with code reading. The problem with the static view is that inheritance is ignored, but this can be fixed by fully flattened classes. The second view is the compile-time view because this is when the inheritance actually “occurs.” The third view is the execution-time view, when objects of classes are instantiated. Testing really occurs with the third view

1.6.3 Pseudocode for Windshield Wiper Class

When a dial or lever event occurs, the corresponding sense method sends an (internal) message to the setWiperSpeed method, which, in turn, sets its corresponding state variable wiperSpeed. windshieldWiper class has three attributes, get and set operations for each variable, and methods that sense the four physical events on the lever and dial devices.

```
class windshieldWiper
        private wiperSpeed
        private leverPosition
        private dialPosition
        windshieldWiper(wiperSpeed, leverPosition, dialPosition)
        getWiperSpeed()
        setWiperSpeed()
        getLeverPosition()
        setLeverPosition()
        getDialPosition()
        setDialPosition()
        senseLeverUp()
        senseLeverDown()
        senseDialUp(),
        senseDialDown()

End class windshieldWiper
```

Unit Testing for Windshield Wiper Class

Part of the difficulty with the class-as-unit choice is that there are levels of unit testing. In the example, it makes sense to proceed in a bottom–up order beginning with the get/set methods for the state variables (these are only present in case another class needs them). The dial and lever sense methods are all quite similar; pseudocode for the senseLeverUp method is given next

```
senseLeverUp()
Case leverPosition Of
Case 1: Off
leverPosition = Int
Case dialPosition Of
Case 1:1
wiperSpeed = 6
Case 2:2
wiperSpeed = 12
Case 3:3
wiperSpeed = 20
EndCase  'dialPosition
Case 2:Int
leverPosition = Low
wiperSpeed = 30
Case 3: Low
leverPosition = High
wiperSpeed = 60
Case 4: High
(impossible; error condition)
EndCase  'leverPosition

End enseLeverUp
```

1.7 QUESTIONS

1. What are the different levels of Object Oriented Testing
2. What is class testing? Explain the process
3. Compare conventional and Object Oriented testing
4. Explain classes as unit with respect to Object Oriented Testing
5. What is the significance of performance testing with respect to Object Oriented System Testing

6. Explain how UML supports Object Oriented Integration Testing

1.8 FUTHER READING

- **Software testing a Craftsman's Approach** by Paul C Jorgensen
- **Software Testing** by Yogesh Singh University Press 2012
- http://www.cs.swan.ac.uk/
- http://softwaretestingfundamentals.com/functional-testing/
- https://dzone.com/articles/what-is-functional-testing-types-tips-limitations

❖❖❖❖

8

Object Oriented Testing Part 2

Unit Structure

1.0 : Objectives
1.1 : Object Oriented Integration Testing
1. 2: UML Support of Integration Testing
1.3: MM-Paths for Object-Oriented Software
1.4: Object Oriented Testing
1.5: Question
1.6: Further Reading

1.0 OBJECTIVES

After completing this chapter, you will be able to:

- ❖ Understand the concept of Data and Information.
- ❖ Differentiate between the Analog Verses digital Signals.
- ❖ Deal with the different number system in arithmetic.
- ❖ Understand the number system conversions.
- ❖ Solve the Arithmetic examples based on Binary arithmetic.

1.1 Object-Oriented Integration Testing

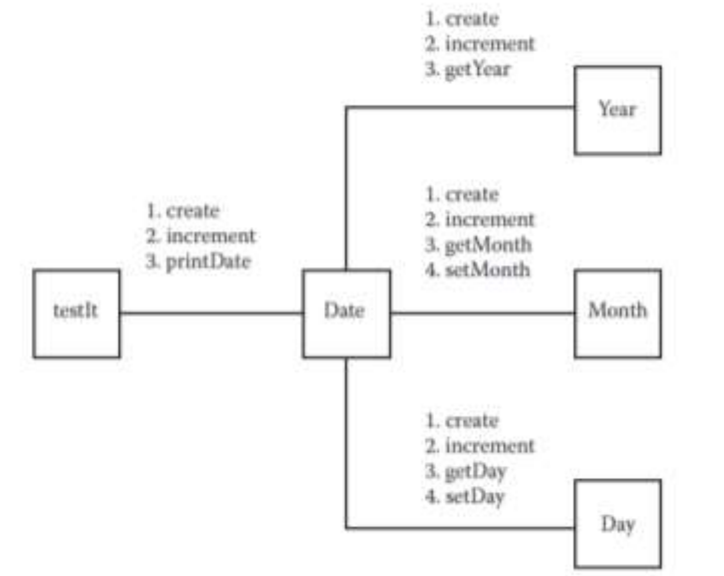
If the operation/method choice is taken, two levels of integration are required: one to integrate operations into a full class, and one to integrate the class with other classes. This should not be dismissed. The whole reason for the operation-as-unit choice is that the classes are very large, and several designers were involved.

Turning to the more common class-as-unit choice, once the unit testing is complete, two steps must occur: (1) if flattened classes were used, the original class hierarchy must be restored, and
(2) if test methods were added, they must be removed.

1.2 UML Support for Integration Testing

In UML-defined, object-oriented software, collaboration and sequence diagrams are the basis for integration testing.
Once this level is defined, integration-level details are added. A collaboration diagram shows the message traffic among classes.
A collaboration diagram is very analogous to the Call Graph. Collaboration diagram supports both the pairwise and neighborhood approaches to integration testing.

With pairwise integration, a unit (class) is tested in terms of separate "adjacent" classes that either send messages to or receive messages from the class being integrated. To the extent that the class sends/receives messages from other classes, the other classes must be expressed as stubs.



Collaboration diagram for o-o calendar

One drawback to basing object-oriented integration testing on collaboration diagrams is that, at the class level, the behavior model of choice in UML is the StateChart.
Neighborhood integration raises some very interesting questions from graph theory. Using the (undirected) graph in Figure 15.9, the neighborhood of Date is the entire graph, while the neighborhood of testIt is just Date. Mathematicians have identified various "centers" of a linear
graph. One of them, for example, is the ultracenter, which minimizes the maximum distances to
the other nodes in the graph. In terms of an integration order, we might picture the circular ripples
caused by tossing a stone in calm water. We start with the ultracenter and the neighborhood of

nodes one edge away, then add the nodes two edges away, and so on. Neighborhood integration of
classes will certainly reduce the stub effort, but this will be at the expense of diagnostic precision.
If a test case fails, we will have to look at more classes to find the fault.

1.3 MM-Paths for Object-Oriented Software

An object-oriented MM-path is a sequence of method executions linked by messages
An MM-path starts with a method and
ends when it reaches a method that does not issue any messages of its own; this is the point of
message quiescence
MM Paths can be very useful for testing as they provide a detail outline of how a program will execute when a given method is called, this precisely greatly improves the efficiency of a tester or developer in finding a Bug if one should occur during the test.
However care must be taken when selecting the paths themselves to ensure that all messages are covered i.e. all lines of
code are executed at least once.
MM Paths do not require stub files, however if a method was called that contained hundreds of other method calls the advantage of precision would be lost and more time would need to be taken for a Bug to be found.
Another disadvantage of this approach is that the paths themselves although easy for a tester to see for a single test, manual or even automated selection of MM Paths for the whole program could be quite computationally expensive especially for larger programs

1.4 Object Oriented System Testing

System testing is (or should be) independent of system implementation. A system tester does not really need to know if the implementation is in procedural or object-oriented code.
UML can be used for generating test cases.
Consist of several levels of use cases including high level, essential, expanded essential, Real. Each use case defines a scenario, which describes the functional requirement of system. Complete GUI design is tested with controls

1.5 QUESTIONS

1. What are the different levels of Object Oriented Testing
2. What is class testing? Explain the process

3. Compare conventional and Object Oriented testing
4. Explain classes as unit with respect to Object Oriented Testing
5. What is the significance of performance testing with respect to Object Oriented System Testing
6. Explain how UML supports Object Oriented Integration Testing

1.6 FUTHER READING

- ❖ **Software testing a Craftsman's Approach** by Paul C Jorgensen
- ❖ **Software Testing** by Yogesh Singh University Press 2012
- ❖ http://www.cs.swan.ac.uk/
- ❖ http://softwaretestingfundamentals.com/functional-testing/
- ❖ https://dzone.com/articles/what-is-functional-testing-types-tips-limitations

❖❖❖❖

8

LEVELS OF TESTING - II

8.1: Integration Testing

    8.1.1: Top-Down Approach

    8.1.2: Bottom-Up Approach

    8.1.3: Modified Top-down Approach

8.2: Big-Bang Approach

8.3: Acceptance Testing

8.4: System Testing

8.5: Thread

8.6: Regression

8.7: Questions

8.8: Further Reading

8.1 Integration Testing:

Integration Testing involves integration of units to make a module/integration of modules to make a system integration of system with environmental variables if required to create a real-life application.

It may start at module level, where different units and components come together to form a module and go till system level. If module is self-executable, it may be taken for testing by testers. If it needs stubs and drivers, it is tested by developers.

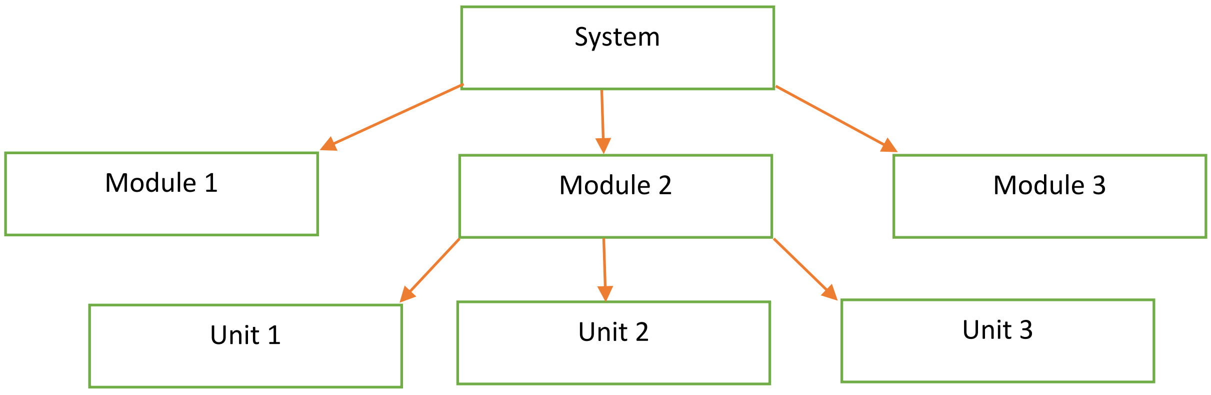


Fig. shows Integration Testing View

There are different approaches of Integration Testing depending upon how the system is integrated. Following are the different approaches:

8.1.1. Top-Down Testing:

In top-down testing approach, the top level of the application is tested first and then it goes downward till it reaches the final component of the system. All top-level components called by tested components are combined one by one and tested in the process. Drivers may not be required as we go downward as earlier phase will act as driver for latter phase while one may have to design stubs to take care of lower-level components which are not available at that time.

Top-level components are the user interfaces which are created first to elicit user requirements or creation of prototype. Agile approaches like prototyping, formal proof of concept, and test-driven development use this approach for testing.

Advantages of Top-Down Approach:

1. Feasibility of an entire program can be determined easily at an early stage as the top most layer is made first.

2. Top-down approach can detect major flaws in system designing by taking inputs from the user.
3. Many times, this approach does not need drivers as the top layers are available first which can work as drivers.

Disadvantages of Top-Down Approach:

1. Units & modules are rarely tested alone before their integration. There may be few problems in individual units/modules which may get compensated in testing.
2. It can create a false belief that software can be coded & tested before design is finished.
3. Stubs are to be written & tested before they can be used in integration testing.

8.1.2. Bottom-Up Testing:

It focuses on testing the bottom part/individual units and modules, and then goes upward by integrating tested and working units and modules for system testing. It is a mirror image of the Top-down approach, with the difference that stubs are replaced by driver modules that emulate units at the next level up in the tree. In Bottom-top integration, we start with the leaves of the decomposition tree and the test them with specially coded drivers.

Advantages of Bottom-Top Approach:

1. Each component & unit is tested first for its correctness. If it found to be working correctly, then only it goes for further integration.
2. It makes a system more robust since individual units are tested & confirmed as working.
3. Incremental integration testing is useful where individual components can be tested in integration.

Disadvantages of Bottom-Top Approach:

1. Top-level components are the most important but tested last, where the pressure of delivery may cause problem of not completing testing. There can be major problems during integration or interface testing, or system-level functioning may be a problem.
2. Objects are combined one at a time, which may need longer time and result into slow testing. Time required for complete testing may be very long and thus, it may disrupt entire delivery schedule.
3. Designing and writing stubs and drivers for testing is waste of work as they do not form a part of final system.

4. Stubs & drivers are to be written and tested before using them in integration testing. One needs to maintain the review and test records of stubs and drivers to ensure that they do not introduce any defect.
5. For initial phases, one may need both stubs & drivers. As one goes on integrating units, original stubs may be used while large number of new drivers may be required.

8.1.3. Modified Top-Down Approach:

This approach tries to combine the better parts of both approaches. It gives advantages of top-down approach & bottom-top approach to some extent at a time and tries to remove disadvantages of both approaches. Tracking and effecting changes is most important as development and testing start at two extreme ends at a time.

Advantages of Modified Top-Down Approach:

1. Important units are tested individually, then combined to form the modules and modules are tested before system is made.
2. The systems tested by modified approach are better in terms of residual defects as bottom-up approach is used for critical components and better for customer-requirement elicitation as top-down approach.
3. It also saves time as all components are not tested individually.

Disadvantages of modified Top-Down Approach:

1. Stubs & drivers are required for testing individual units before they are integrated.
2. Definition of critical units is very important. Criticality of the unit must be defined in design.

8.2 Big-Bang Testing:

This is the most commonly used approach at many places, where the system is tested completely after development is over. There is no testing of individual units/modules & integration sequence.

Advantages of Big-Bang Approach:

1. It gives a feeling that cost can be saved by limiting testing to last phase of development. Testing is done as a last-phase of the development lifecycle in the form of system testing. Time for writing test cases & defining test data at unit level, integration level etc.
2. If an organization has optimized processes, this approach can be used as a validation of development process and not a product validation. If all levels of

testing are completed and all defects are found and closed, then system testing may act as certification testing to see if there are any major issues still left.

3. No stub/driver is required to be designed and coded in this approach. The cost involved is very less as it does not involve much creation of test artifacts.
4. Big-Bang approach is very fast. It may not give adequate confidence to users as all permutations and combinations cannot be tested in this testing.

Disadvantages of Big-Bang Approach:

1. Problems found in this approach are hard to debug. Many times, defects found in random testing cannot be reproduced as one may not remember the steps followed in testing at that particular instance.
2. It is difficult to say that system interfaces are working currently and will work in all cases.
3. Location of defects may not be found easily. In big-bang testing, even if we can reproduce the defects, it can be very difficult to locate the problematic areas for correcting them.
4. Interface faults may not be distinguishable from other defects.
5. Testers conduct testing based on few test cases by heuristic approach and certify whether the system works/does not work.

8.3 Acceptance Testing:

It is considered as the final stage of testing of an application, before it is accepted formally by a customer for operational purpose. It is done by the customer or by somebody on behalf of the customer.

Acceptance Testing validates the following:

1. Whether user needs, as defined in the system requirement specifications, are achieved by the system or not. It must evaluate ‘conformance to requirements’ along with ‘fitness for use’ from customer’s perspective.
2. Whether system performance meets the expectations of customer as defined/documented in system requirement statement. Performance criteria must be defined in measurable terms to avoid any misinterpretation in future.
3. It is a formal testing conducted, generally at the end of software development, to determine whether the application satisfies its acceptance criteria or not.

Acceptance testing characteristics are given below:

1. It is the final opportunity for buyer to examine the software with respect to expectations and to decide about acceptance/modification of software either through enhancement, bug fixing or changing the entire business process to match software.

2. It is conducted in production environment or simulated production environment as defined in the contract.
3. It can be an incremental process of accepting/rejecting software through the software development life cycle where the decision of acceptance starts from the beginning of the development.
4. It occurs at pre-specified times as defined in software development contract & plan, where work products are accepted by customer.
5. Formal final acceptance must occur at the end of software development life cycle when customer decides the outcome of development activities.
6. It consists of tests to determine whether developed system meets predefined criteria of acceptance or not. Outcome of such test may be acceptance/rejection of software or changes proposed in terms of enhancements.

8.4 System Testing:

It represents the final testing done on a system before it is delivered to the customer. It is done on integrated subsystems that make up the entire system, or the final system getting delivered to the customer. System testing validates that the entire system meets its functional/non-functional requirements as defined by the customer in software requirement specification. The criteria involve an entire domain or selected parts depending upon the scope of testing.

System testing goes through the following stages:

1. Functional Testing: It intends to find whether all the functions as per requirement definition are working or not. Any software is intended for doing some functions which must be defined in requirement statement.
2. User Interface Testing: Once the functionalities are finalized, the next step is to set the user interface correct. User interface testing may involve colors, navigations, spellings and fonts. Sometimes, there can be a thin line between functional testing and user interface testing and one may take decision depending upon the situation.

8.5 Thread:

Threads are hard to define; in fact, some published definitions are counterproductive, misleading, or wrong. It is possible to simply treat threads as a primitive concept that needs no formal definition. Threads have distinct levels.

- A Unit-level thread is usefully understood as an execution-time path of source instructions or as a sequence of DD Paths.

- An Integration-level thread is MM path i.e. an alternating sequence of module executions and messages.
- A System-level thread is a sequence of atomic system functions. Because atomic system functions have port events as their inputs & outputs, a sequence of atomic system functions have port events as their inputs & outputs.

Unit Testing tests individual functions; integration testing examines interactions among units; and system testing examiners interactions among atomic system functions.

8.6 Regression Testing:

It is intended to determine whether the changed components have introduced any error in unchanged components of the system. Regression testing may not be considered as special testing in development projects.

Regression testing can be done at,

1. Unit level to identify that changes in the units have not affected its intended purpose, and the other parts of the unit are working properly even after the changes are made in some parts.
2. Module level to identify that the module behaves in a correct way after the individual units are changed.
3. System level to identify that the system is performing all the correct actions that it was doing previously as well as actions intended by requirements after change is made in some parts of the system.

Important Development methodologies where regression testing is very important:

1. Any kind of maintenance activity conducted in a system may need a regression testing cycle.
2. Iterative development methodology needs huge regression testing as there are several iterations of requirement changes followed by design changes and changes in code.
3. Agile development needs huge cycles of regression testing.

8.8 Questions:

1. Explain the advantages and disadvantages of Bottom Up Approach.
2. Efficiency of unit testing affects the effectiveness of testing process. Discuss. Compare the objectives of Integration and Interaction Testing.
3. Explain Acceptance Testing in detail.
4. What is retrospection? Explain retrospection with respect to Unit Testing.
5. What are the levels of testing? Explain.
6. What is integration testing? List all the integration testing strategies.
7. Explain any one. (a) List all advantages and disadvantages of call graph integration testing. (b) Briefly explain the concept of system testing.
8. Define the term ‘Interaction’. Discuss Taxonomy o interactions.
9. Define Threads. What are the distinct levels of Threads.

8.9 Further Read

1. Software Testing Principles, Techniques and Tools, M.G. Limaye, TMH
2. Software testing by Yogesh Singh. Cambridge University Press, 2012.
3. Introduction to Software Testing, Paul Ammann, Jeff Offutt, Cambridge University Press.
4. Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, Rex Black, Wiley

11

Testing Process

Unit Structure

11.0: Planning

11.1: Test Policy

11.1.1: Number Systems

11.2: Test Plan

11.2.1: Advantages of Testing Planning

11.2.2: Benefits of Test Plan

11.3: Test Cases

11.3.1: Characteristics of Good Test Case

11.3.2: How to write a Good Test Case

11.4: Test Reports

11.4.1: Types of Test Reports

11.4.2: Characteristics of Test Reports

11.0 Planning

It is the most important phase in project or product life cycle. Software development, testing, deployment and maintenance are planned activities happening in the life cycle of a software product & they must be planned to ensure effectiveness of life cycle activities. During test planning, we make sure we understand the goals and objectives of the customers, stakeholders, and the project, and the risks which testing is intended to address. This will give us what is sometimes called the mission of testing or the test assignment. Based on this understanding, we set the goals and objectives for the testing itself, and derive an approach and plan for the tests, including specification of test activities. To help us we may have organization or program test policies and a test strategy. Test policy gives rules for testing, e.g. 'we always review the design documents'; test strategy is the overall high-level approach, e.g. 'system testing is carried out by an independent team reporting to the program quality manager.

Test planning has the following major tasks:

1. Determine the scope and risks and identify the objectives of testing: we consider what software, components, systems or other products are in scope for testing; the business, product, project and technical risks which need to be addressed; and whether we are testing primarily to uncover defects, to show that the software meets requirements, to demonstrate that the system is fit for purpose or to measure the qualities and attributes of the software.
2. Determine the test approach (techniques, test items, coverage, identifying and interfacing with the teams involved in testing, testware): we consider how we will carry out the testing, the techniques to use, what needs testing and how extensively (i.e. what extent of coverage). We'll look at who needs to get involved and when (this could include developers, users, IT infrastructure teams); we'll decide what we are going to produce as part of the testing.
3. Implement the test policy and/or the test strategy: we mentioned that there may be an organization or program policy and strategy for testing. If this is the case, during our planning we must ensure that what we plan to do adheres to the policy and strategy or we must have agreed with stakeholders, and documented, a good reason for diverging from it.
4. Determine the required test resources: from the planning we have already done we can now go into detail; we decide on our team make-up and we also set up all the supporting hardware and software we require for the test environment.
5. Schedule test analysis and design tasks, test implementation, execution and evaluation: we will need a schedule of all the tasks and activities, so that we can track them and make sure we can complete the testing on time.
6. Determine the exit criteria: we need to set criteria such as coverage criteria that will help us track whether we are completing the test activities correctly.

They will show us which tasks and checks we must complete for a particular level of testing before we can say that testing is finished.

11.1 Test Policy

Test policy is defined at organisation level by the senior management of the organisation. It defines the intent of senior management with respect to testing activities which is derived from the mission statement of the organisation and the vision of management.

The reason for this difference is as follows:

- In case of product organisation, the entire product is owned by the organisation & it is responsible for any problem faced by final user.
- In case of project organisation, it offers development service to customer but does not own the product.

11.1.1 Content of the Test Policy

It is very important that the entire organisation must know & understand organisational policy for testing.

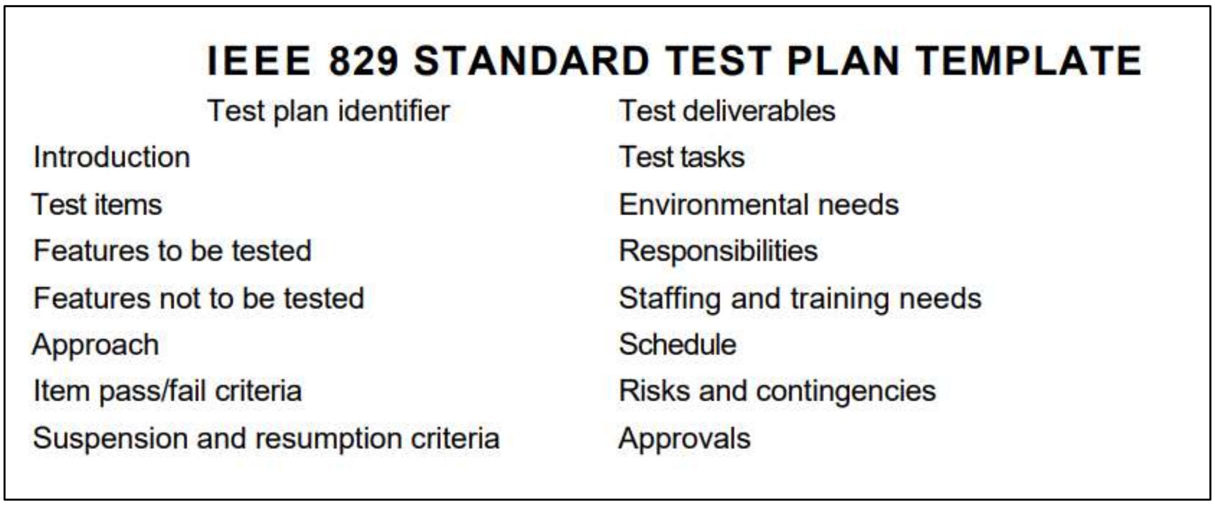
1. <u>Introduction</u>: It defines the background information for establishing the testing policy in the organisation. Testing policy details the views & opinions of senior management about testing activities in the organisation.
2. <u>Scope</u>: Scope of testing policy defines the areas or circumstances where the test policy would be applicable as well as define some cases where it will not be applicable.
3. <u>Testing organisation & reporting</u>: It defines the overall structure of testing as a function and “locating testing function” in the organisation structure. It helps in defining roles and responsibilities of people associated with testing activities and their relationship with development team and other stakeholders.
4. <u>Training</u>: Testers need training to enhance their skills, understand new concepts of testing and face the changing scenarios in the technological/domain fields. Training may include domain-related training which may help testers to identify domain-related issues, if any.

11.2 Test Plan:

Test plan is a contract between test team and development team/customer. It helps in identifying testing as an organisation in entire project & describing the role and responsibility of testing in entire project or product development or maintenance. It describes in detail about testing activities, testing resources, how test scenarios and test cases will be defined. It is not a test design specification, a collection of test

cases or a set of test procedures; in fact, most of our test plans do not address that level of detail. The test plan also helps us manage change. During early phases of the project, as we gather more information, we revise our plans. As the project evolves and situations change, we adapt our plans. Written test plans give us a baseline against which to measure such revisions and changes. Furthermore, updating the plan at major milestones helps keep testing aligned with project needs.

Following is the Test Plan Template:



IEEE 829 STANDARD TEST PLAN TEMPLATE

Test plan identifier	Test deliverables
Introduction	Test tasks
Test items	Environmental needs
Features to be tested	Responsibilities
Features not to be tested	Staffing and training needs
Approach	Schedule
Item pass/fail criteria	Risks and contingencies
Suspension and resumption criteria	Approvals

11.2.1 Advantages of Test Planning:

1. Work involved in test planning and setup pays in the long term.
2. It describes the way in which testing team will show whether software works correctly as per requirements & the acceptance criteria.
3. It addresses various levels of testing such as unit testing, module testing, system testing, integration testing, black box testing as well as white box testing.
4. It explains who does testing, why tests are performed, how tests are conducted and when tests are scheduled.
5. It must contain procedures, environment and tools necessary to implement an orderly, controlled process for test execution, defect tracking, coordination of rework and configuration.

11.2.2 Benefits of Test Plan:

1. Build knowledge of stakeholders about testing: It builds essential knowledge of all the stakeholders in planning for tests and execution of tests.
2. Identify Stakeholders: Test plan identifies the audiences for test plan & sources for writing a test plan, test scenarios, test cases and defining test data.
3. Enhance Visibility: Test plan improves visibility of the upcoming test to management, project team, customer and test team at large.

4. Ensure Understanding: Test planning ensures complete understanding of the test event by test planner and test team.
5. Assist in Team Building: Test planning helps in test building for test team.

11.3 Test Cases

Test cases are derived from the test scenario. Test case describes each transaction and expected result of each transaction included in test scenario. Test case definition is one of the debatable issues as there are many views about what is meant by test cases. Test cases are executed through test data. Test data is defined using different techniques such as equivalence partitioning, boundary value analysis, state transition and error guessing.

11.3.1 Characteristics of Good Test Case:

1. Accurate: Test case execution must test the application for the criteria for which it is designed.
2. Economical: Test case must define the correct steps for testing an application.
3. Repeatable & reusable: Test cases must be documented and used so that they can be repeatable and reusable.
4. Traceable to requirements: Every test case defined and executed must have corresponding requirement relationship through test scenarios.
5. Appropriate: Test case must be appropriate for the system under testing.
6. Self-Standing: Test case must be independent of any tester performing it.
7. Self-cleaning: This is an important parameter in automation testing. Test case must clean the changes it has made in the application, environment & other factors affecting system.

11.3.2 How to write a Good Test Case:

Test case is a very important document from validation of system perspective. They must be written at each stage of validation starting from unit testing till acceptance testing.

- Test case must be testable. Test cases are defined to execute the application and it must do the task.
- Use active voice while writing the test cases so that testers knows what is to be done and when to wait for the system to do it.
- Inform tester what will be displayed by the system on the screen & what will be done by the system at each step.
- While writing test cases, simple conversational language is preferable in place of use of jargons.
- Exact and consistent names of fields must be used in place of generic names.
- Do not explain windows basics. The basic controls like combo boxes, text boxes and radio buttons etc. must be known to testers.
- Order of test cases must follow business scenario.

11.4 Test Reports

Testing reports are created at the end of test cycles. Its purpose is to communicate about the progress achieved by test team & any impediments faced by them to the stakeholders of the project.

11.4.1 Types of Test reports:

1. Unit Test Report: They are generated at the end of unit testing activity. As the unit testing may happen as and when the units are completed, it may not be a formalised test report but some informal communication between various stakeholders.
2. Integration Test Reports: It is generated after integration testing activities are completed, and one can say something about the software being tested when the units are bundled together.
3. System Test Reports: There may be many cycles of system testing as defined by project plan and test plan. System test reports are generated when individual system testing cycle is completed.
4. Acceptance Test Reports: They are generally made in two phases – alpha acceptance test report when testing is done at development site and beta acceptance test report when business pilot is conducted.
5. Various Interim Test Reports: There are many interim test reports as individual steps in testing are completed.

After each phase of testing, test reports are prepared & distributed to relevant stakeholders. Test report may or may not contain a test log. If the number of tests is very large, test report may indicate summary of test activities and status of project and may give pointer to test log for detailing.

11.4.2 Following are the characteristics of Test report:

a. Individual project test reports for the test phases completed or iteration test report for the iterations completed.
b. Integration test report after integration testing is completed.
c. System test report after system testing is completed.
d. Acceptance test report prepared by project test team and customer.

Following is the template for Test Summary Report:

IEEE 829 STANDARD: TEST SUMMARY REPORT TEMPLATE

Test summary report identifier	Evaluation
Summary	Summary of activities
Variances	Approvals
Comprehensive assessment	
Summary of results	

Questions:

1. How to write a good test case?
2. What is the purpose of test reports?
3. Explain benchmarking concept. Why it is required?
4. Differentiate between Qualitative & Quantitative data.
5. What are the phases of Test process improvement model?
6. What are the different types of efforts?

Further Read

1. Software Testing Principles, Techniques and Tools, M.G. Limaye, TMH
2. Software testing by Yogesh Singh. Cambridge University Press, 2012.
3. Introduction to Software Testing, Paul Ammann, Jeff Offutt, Cambridge University Press.
4. Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, Rex Black, Wiley.

12

Testing Process-II

Unit Structure

Benchmarking
12.1: Qualitative & Quantitative Analysis
    6.6.1: Data
    6.6.2: Qualitative Data
    6.6.3: Quantitative Data
12.2: Efforts
    6.7.1: Estimated Efforts
    6.7.2: Budgeted Efforts
    6.7.3: Approved Efforts
    6.7.4: Actual Efforts
12.3: Test Process
    12.3.1: Need for Test Process improvement

12.0 Benchmarking:

It talks about the reference, which may be considered as 'The Best' in the given area or the field as per the opinion of some authority, generally project/organisation's leadership. Benchmark may be taken as a reference when organisation is planning for improvements. For the improvements, there are two options available.

- Invent the wheel again and again which may involve huge efforts in definition of what is the best. It may not be able to define exact achievable target, and people may be running behind something which is not feasible at all.
- Use the processes followed by somebody else for incorporating improvements, where somebody else has invented a wheel already which can be used with little customisation.

Benchmark Partner may be either competitors or a leading organisation which can help indirectly in improving performance of an organisation. The targeted organisation is called a benchmarking partner.

Benchmarking is used,

- To gain better understanding of oneself with respect to somebody who is considered as best in a selected area.
- To gain awareness of definition of 'The best' so that one can define the objectives clearly.
- To understand necessary improvements to close the gap between the current level and best level expected.

12.1 Qualitative & Quantitative Analysis

There are two ways for continual improvement in an organisation, P-D-C-A (Plan-Do-Check-Act) and M-M-C-I (Measure-Monitor-Control-Improve). In reality, both the names have similar approach, where one must plan on the basis of what the current performance is, and where the organisation wishes to react.

Data: Any measurement needs data to be collected from the process/product. Data may represent a process attribute or product attribute accordingly. Data may indicate the status of process or product. Classification of data may be done under various schemes.

Some of them are as follows:

12.1.1 Qualitative Data:

a. It indicates the classes or ranges in which a attribute of a product or process is present or not.
b. It is also called as 'Categorial data'.

c. When the process definition is vague or there are no accurate measurements or precision measuring instruments are available, one may have to select qualitative data to understand the process.
d. Types of Qualitative Data:
    i. Nominal Data: In case of nominal data, attributes measurements are collected without following any specific categorisation such as class definition, class range & unit of measurement.
    ii. Ordinal Data or Class Definition: The measurements can be classified in different strata depending upon the perception of person making such observation.
    iii. Boolean Data: Sometimes data is not exactly a variable. It can take only two forms such as present/absent, and true/false etc.
e. Advantages of Qualitative Data:
    i. Data gathering is very fast as one need not measure actual attributes. It is mainly on perception of an individual about the situation.
    ii. It suffices the purpose when the maturity level of audience is very low.
    iii. It does not need expertise in the area or domain under evaluation. Since it is perception based, very high level of calibration is not involved.
    iv. It is independent of measurement process & measuring instrument.
f. Disadvantages:
    i. It is perception based and may change from person to person and instance to instance.
    ii. It may not be useful for highly matured environment.

12.1.2 Quantitative Data:

a. It is the actual measurement of an attribute of a product or process, expressed in numbers with some units of measurement.
b. It makes comparison between different entities easy, and one may be able to map the variations in process or product easily by using numbers.
c. When the organisational maturity improves, it shifts its measurement practices from qualitative data to quantitative data to help in measurement and data improvement.
d. Types of Quantitative Data:
    i. Absolute data: For example, temperature, size of software, and number of employees can be considered as absolute data.
    ii. Data sets: For example, ‘days of week’ which includes all the days from Sunday to Saturday.
    iii. Relative data: For example, speed and defect density.
    iv. Continuous data: For example, set of relational numbers can take any value between -ve infinity to +ve infinity.

        v. Discrete Data: For example, a set of rational numbers completely divisible by 7.

e. Advantages:
    i. It is accurate than qualitative technique as it is expressed in numbers and units of measurement.
    ii. It gives exact status of a situation or the process in terms of its attributes.
    iii. Statistical analysis of a process is possible when quantitative analysis is used.
f. Disadvantages:
    i. It needs a definition of measurement including unit of measurement and type of instrument.
    ii. It may need some level of automation to collect data and conduct analysis.
    iii. It may consume more time in collecting measurements from the process and preparing data for calculations.
    iv. Amount of error id dependent on the accuracy of the measuring equipment, including human beings in terms of repeatability and reproducibility.

12.2 Efforts

12.2.1 Estimated Efforts:

a. When a test manager is writing a test plan or estimating the efforts required for testing as per the defined approach, he may be making several assumptions about various aspects such as status of application, number of testers, skills of testers, availability of test tools etc.
b. These assumptions are the inputs for estimation and effect estimation directly.

12.2.2 Budgeted Efforts:

c. Despite making a test plan, it may be possible that 10 testers with 5 years of experience are not available.
d. Test manager may have to work with, say 15 people with lesser experience.
e. This will affect the estimated efforts significantly.
f. But this is planned arrangement, and organisation must have done analysis before putting such team for testing.

12.2.3 Approved Efforts:

a. They are guided by market conditions & relationships between customer and organisation.
b. Marketing may be able to sell, say 200-person days to customer, though estimation is 100 person days.
c. This is called ‘approved effort’ which customer is going to pay for.

12.2.4 Actual Efforts:

a. These are the efforts spend in testing for the given project. It may be any value and it is reality. Thus, we get three variances in such case as shown below,

   Estimated effort variance = (Actual efforts – Estimated efforts) x 100/Estimated efforts.
   Budgeted effort variance = (Actual efforts – Budgeted efforts) x 100/Budgeted efforts.
   Approved effort variance = (Actual efforts – Approved efforts) x 100/Approved efforts.

12.3 Test Process:

There are many problems associated with testing as an activity, typically with adhoc testing, or testing as a last stage of development or certification approach where there are no verification activities in previous phases.

Problems are as follows:

1. Though we decide testing must start from proposal, it rarely happens. It is difficult to test the software completely in system testing.
2. Testing is performed by someone who happens to be available or is free.
3. Testing stops when delivery date of an application is reached or when application goes into production.
4. Testing is considered as a certification activity, and if no few defects have been found recently, then it authorises release of a product to the user.
5. Testing is a short of time, people, resources & expertise.

12.3.2 The need for Test Process improvement:

Testing process improvement may be needed for continual improvement.

1. The aim of an improved test process must be to detect defects as close as possible to their source of introduction to minimise correction costs, and to give information about the system quality as early as possible.
2. All activities related to testing must be adjusted to each other for achieving an optimised strategy for developing a good product, by detecting the defects as early as possible so that they can be fixed and process can be improved.
3. Testing must become a highly professional task which requires special testing skills, with functional expertise, testing methodology expertise, and design expertise.
4. Quality of the test process must be measured and the results must be used as an input for further test process improvement.

Following are the phases of Test process improvement model:

1. Process Improvement Planning (Plan Phase):
    a. Understand Organisation’s as well as Business Requirements: Before starting any improvement process, one must understand where they are now and where they wish to reach as a part of test process improvement model. The gap between the two states, viz. expected state and actual state will indicate the direction & quantum of improvement needed.
    b. Conducting assessment of present situation: Once the goals to be achieved by such process improvements are decided, one must analyse the present situation to find where they are now. There must be some methodology defined for such assessment which can interpret the current level of test process and achievement of test process in terms of achieving organisational goals.
    c. Initiate Process improvement actions: once the gap between expected and actual stage is very clear, one knows where they wish to reach and where they are currently. Now, a path for reaching the target must be devised.
2. Process Improvement Action implementation (Do Phase):
    a. Implementation of Action Plan Designed: Actions on paper may look good, but one needs to implement them. There may be lot of constraints while implementing the actions, and one may have to address these constraints. Action plan must declare dependencies, assumptions and risks of implementation.
3. Process Improvement Action Review (Check):
    a. As defined, there must be continuous review of progress made and actions in case of deviations. Feedback loop must be used for monitoring and guiding progress- whether one is in the right direction or not and whether time schedule has been monitored correctly or not.
4. Process Improvement Action Changes (Act):
    a. Learn the lessons of Implementation: As an output of check process, there may be some variations in achievements with respect to planned expectations from the improvement actions.
    b. Gap can be removed either by changing the action plan or revising the action effectiveness.
    c. Sustain Improvements: The process improvements done initially may have a good result for some time as people may find something interesting that may exceed the planned actions. There can be problems with measurements as people may manage measures which are not real.

Questions:

1. How to write a good test case?
2. What is the purpose of test reports?
3. Explain benchmarking concept. Why it is required?
4. Differentiate between Qualitative & Quantitative data.
5. What are the phases of Test process improvement model?
6. What are the different types of efforts?

Further Read

1. Software Testing Principles, Techniques and Tools, M.G. Limaye, TMH
2. Software testing by Yogesh Singh. Cambridge University Press, 2012.
3. Introduction to Software Testing, Paul Ammann, Jeff Offutt, Cambridge University Press.
4. Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, Rex Black, Wiley.