BCBM402T

SOFTWARE QUALITY ASSURANCE AND TESTING

DR. C.V. RAMAN UNIVERSITY
Kargi Road, Kota, Bilaspur, (C. G.),
Ph. 07753-253801, 07753-253872
E-mail: info@cvru.ac.in, Website: www.cvru.ac.in

BCBM402T
SOFTWARE QUALITY ASSURANCE AND TESTING

Credit- 6

Subject Expert Team

Dr. Abhinav Shukla
Associate Professor,
Dr. CV Raman University, Bilaspur

Dr. Rajendra Gupta
Professor,
Rabindranath Tagore University, Bhopal

Dr. Ayush Kumar Agrawal
Assistant Professor,
Dr. CV Raman University, Bilaspur

Dr. Swarangini Sinha
Professor,
Rabindranath Tagore University, Bhopal

Course Edited By:

Mr. Vishwas Victor
Assistant Professor, Atal Bihari Vajpai Vishwavidyalaya, Bilaspur

---

Unit Written By:

Dr. Amrita Verma
Associate Professor, Dr. CV Raman University, Bilaspur

Warning: All rights reserved, No part of this publication may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior written permission from the publisher.

Edition: 2026
Published by: **RABINDRANATH TAGORE UNIVERSITY**
Mendua, Post: Bhojpur, Near Bangrasiya chouraha, Bhopal-Chiklod Road,
District: Raisen, (M.P.)
Ph. 0755-2700447, Website: https://rntu.ac.in, E-mail: info@rntu.ac.in

STRUCTURE OF PROGRAM

Title of the Program	**:**	**Bachelor of Computer Application**
Program Type	**:**	**Undergraduate**
Program Credits	**:**	**120**
Program Duration	**:**	**3 years**
Eligibility Criteria	**:**	**12th Pass**

PROGRAMME OBJECTIVE

- To produce employable IT workforce, that will have sound knowledge of IT and business fundamentals that can be applied to develop and customize solutions for Small and Medium Enterprises (SME).
- To develop skilled manpower in the various areas of information technology like. Data base management, Software Development, Computer-Languages, Software engineering, Web based applications etc.
- To Work professionally with positive attitude as an individual or in multidisciplinary teams and communicate effectively.

PROGRAMME OUTCOME

- **PO 1**. Develop competence in basic technical subjects in computer applications like Programming Languages, Data Structures, Databases, Operating Systems, Software Engineering.
- **PO 2.** Identify, analyze, formulate and develop computer applications.
- **PO 3.** Map real life scenarios to various theoretical optimal solutions.
- **PO 4.** Provide simplest automated solutions to various legacy systems.
- **PO 5.** Use modern computing tools and techniques with confidence.

PROGRAMME SPECIFIC OUTCOME

- **PSO 1.** Able to implement computer skill in the areas related to multimedia and website design, Data base, Hardware and networking.
- **PSO 2.** Able to demonstrate basic knowledge in the areas such as, Software Engineering, Data communication and Networking, Data base management, Web Technology and Operating Systems for building IT applications.

- **PSO 3.** Students will be able to understand, analyze and develop computer programs in the areas related to algorithm, system software, web design and networking for efficient design of computer-based system.
- **PSO 4.** Apply standard software engineering practices and strategies in software project development using open-source programming environment to deliver a quality of product for business success.
- **PSO 5.** Student will be able to know various issues, latest trends in technology development and thereby innovate new ideas and solutions to existing problems.

STRUCTURE OF COURSE

Course Title	**:**	**Software Quality Assurance and Testing**
Course Type	**:**	**Core Course**
Course Code	**:**	**BCBM402T**
Course Credits	**:**	**6**
Semester	**:**	**4**

COURSE LEARNING OBJECTIVES:

After completion of the course, the student would be able to

- CO-1 Summarizing the concept Quality of Software.
- CO-2 Classify the Testing Techniques.
- CO-3 Illustrating the unit Testing and its types.
- CO-4 Relating Software Verification and Validation.
- CO-5 Categorizing the Level of Testing

Course Content

BLOCK I	**INTRODUCTION TO QUALITY AND SOFTWARE QUALITY**	**01-88**
Unit 1	Software Quality Assurance	03
Unit 2	Total Quality Management	17
Unit 3	Quality in Different Areas	31
Unit 4	Software Product Quality Assessment	49
Unit 5	Software Development Process	67
BLOCK II	**FUNDAMENTALS OF TESTING**	**89-170**
Unit 6	Software Testing	91
Unit 7	Testing Process	111
Unit 8	Mutation Testing	129
Unit 9	Categories of Defect in Software Testing	**143**
Unit 10	Management Awareness for Testing	157
BLOCK III	**UNIT TESTING: BOUNDARY VALUE TESTING, EQUIVALENCE CLASS TESTING, DECISION TABLE–BASED TESTING, PATH TESTING, DATA FLOW TESTING**	**171-258**
Unit 11	Unit Testing	173
Unit 12	Software Quality Assurance and Testing	189
Unit 13	Decision Table-Based Testing	209
Unit 14	Path Testing	229
Unit 15	Data Flow Testing	243
BLOCK IV	**SOFTWARE VERIFICATION, VALIDATION AND TESTING LIFECYCLE**	**259-330**
Unit 16	Software Verification and Validation	261
Unit 17	Reviews in Testing Lifecycle	**277**
Unit 18	Acceptance Testing	291
Unit 19	V-test Model	303
Unit 20	Testing During Test Planning Phase	315

BLOCK V	**LEVELS OF TESTING, SPECIAL TESTS, INSTALLATION TESTING**	**331-388**
Unit 21	Levels of Testing	333
Unit 22	Special Tests	359
Unit 23	Installation Testing	373
BLOCK VI	**TYPES OF TESTING, OBJECT ORIENTED APPLICATION TESTING**	**389-428**
Unit 24	Types of Testing	391
Unit 25	New Technology Risk and Control Testing	403
Unit 26	Object Oriented Application Testing	415

BLOCK 1	FUNDAMENTALS OF PROGRAMMING AND SOFTWARE DEVELOPMENT	01-88
Unit 1	Programming Concept and Characteristics with Programming Stages	03
Unit 2	Programming Designing and Algorithms	17
Unit 3	Flowcharts Symbols and Programming Techniques	31
Unit 4	Modular Technique and Different Types of Errors	49



This block introduces the fundamental concepts of **quality and software quality**, which are essential for developing reliable, efficient, and user-oriented software systems. It explains the meaning, characteristics, dimensions, and importance of quality in software development. The block also discusses the concept of **software quality**, including quality attributes such as reliability, usability, efficiency, maintainability, portability, security, and functionality. Students will learn how software quality is influenced by requirements, design, coding, testing, and maintenance activities throughout the software development life cycle. The block further highlights the difference between product quality and process quality and explains the role of quality standards and quality assurance in software projects. Understanding these concepts provides a foundation for studying software quality management, software testing, verification and validation, reviews, and other quality-related practices. By completing this block, learners will develop a clear understanding of why quality is important and how systematic quality practices contribute to successful software development.

NOTE

UNIT 1
SOFTWARE QUALITY ASSURANCE

UNIT STRUCTURE

1.1 Introduction
1.2 Objective
1.3 About Software Quality
  1.3.1 Historical Perspective of Quality
  1.3.2 What is Quality? Is it a fact or perception?
  1.3.3 Definitions of Quality
  1.3.4 Core Components of Quality
  1.3.5 Quality View
  1.3.6 Financial Aspect of Quality
  1.3.7 Customers, Suppliers and Processes
1.4 Summary
1.5 Glossary
1.6 Answer to check your progress
1.7 Suggested readings - OER
1.8 Reference
1.9 Exercise

1.1 INTRODUCTION

In the present digital era, software systems have become an essential part of everyday life. From simple mobile applications to complex enterprise systems, software supports communication, business operations, healthcare services, education platforms, transportation systems, and government administration. As society increasingly depends on software-based solutions, the quality of software has emerged as a critical factor for success and reliability. Any failure in software can lead to serious consequences such as financial loss, operational disruption, loss of reputation, and in some cases, threats to human safety.

Software Quality Assurance (SQA) is a structured and systematic approach that ensures software products are developed with the required level of quality. It focuses on establishing processes, standards, and guidelines that help in producing reliable and error-free software. Unlike traditional approaches where quality was checked only at the end of development, SQA integrates quality activities throughout

NOTE

the software development life cycle. This proactive approach helps in identifying potential issues at early stages, thereby reducing rework and development cost.

SQA emphasizes process improvement rather than only product evaluation. It ensures that software development activities such as requirement analysis, design, coding, testing, and maintenance are carried out according to predefined standards. By implementing SQA practices, organizations can deliver software that meets both technical requirements and user expectations. Thus, Software Quality Assurance plays a vital role in developing trustworthy and high-performance software systems.

1.2 OBJECTIVE

After studying this unit, you will be able to:

- To analyse the concept and importance of software quality in modern software systems.
- To explain the historical evolution of quality and its relevance to software development.
- To identify different definitions and perspectives of quality.
- To understand the core components and views of software quality.
- To analyze the financial impact of quality on software development.

1.3 ABOUT SOFTWARE QUALITY

Software quality refers to the degree to which a software product satisfies stated and implied requirements. It is a comprehensive concept that includes functional correctness as well as non-functional attributes such as performance, reliability, usability, maintainability, and security. High-quality software performs its intended functions accurately while providing a positive experience to users.

Quality in software is not determined only by the final product but also by the processes used during development. Well-defined and well-managed development processes lead to better-quality outcomes. Software quality is influenced by factors such as design methodology, coding practices, testing strategies, and documentation standards. In addition, user expectations and perceptions play a significant role in evaluating software quality.

Software quality is often viewed as a combination of objective and subjective factors. Objective factors include measurable attributes such as defect density and response time, while subjective factors include

NOTE

user satisfaction and ease of use. Therefore, software quality is a multidimensional concept that requires both technical excellence and user-oriented design. Understanding software quality is essential for developing dependable and long-lasting software systems.

1.3.1 Historical Perspective of Quality

The idea of quality has its roots in early manufacturing practices, where the primary goal was to produce usable goods rather than defect-free products. In the initial stages of industrial development, quality was ensured mainly through inspection. Finished products were checked, and defective items were separated. This approach was reactive in nature, as defects were identified only after production was completed. Although inspection helped remove faulty products, it did not prevent defects from occurring repeatedly.

As production volumes increased, industries realized that inspection alone was costly and inefficient. This led to the emergence of quality control, where statistical methods were introduced to monitor production processes. Tools such as control charts and sampling techniques were used to identify variations and reduce defects. The focus shifted from detecting defects to controlling the production process.

Later, the concept of quality assurance developed, emphasizing preventive actions rather than corrective ones. Quality assurance focused on defining standardized processes, documentation, and procedures to ensure consistent quality. With the rise of Total Quality Management (TQM), quality became a strategic objective involving the entire organization. Continuous improvement, customer focus, and employee participation became central principles.

In software engineering, quality practices evolved from simple testing activities to structured frameworks such as ISO 9001, CMMI, Six Sigma, and Agile methodologies, which emphasize process maturity, continuous feedback, and early defect prevention.

The concept of "Quality" has evolved significantly over centuries, moving from simple inspection to complex system-wide assurance.

- The Craftsmanship Era (Pre-1900s): Quality was the sole responsibility of the individual craftsman. A shoe-maker, for example, ensured the quality of the shoe personally. Reputation was the primary driver.
- The Industrial Revolution (Early 1900s): With mass production, the "Foreman" took over quality control. However, as factories grew, this became unmanageable, leading to Inspection-based quality where inspectors checked finished goods at the end of the line.

NOTE

- Statistical Quality Control (1930s-1940s): Pioneers like Walter Shewhart introduced statistical methods (control charts) to control the process rather than just inspecting the product. This shifted the focus from detection to prevention.
- Total Quality Management (TQM) (1950s-1980s): W. Edwards Deming and Joseph Juran popularized the idea that quality is everyone's responsibility, not just the quality department's. This era introduced the concept of continuous improvement.

Software Quality Era (1970s - Present): As software became critical to business, these manufacturing principles were adapted for software engineering, leading to standards like ISO 9001, CMMI, and Agile testing methodologies.

1.3.2 What is Quality? Is it a Fact or Perception?

Quality is a complex concept that cannot be defined from a single viewpoint. From a factual perspective, quality is objective and measurable. It is determined by how well software meets documented requirements, technical specifications, and performance standards. Metrics such as defect density, response time, reliability rate, and compliance levels are used to measure quality as a fact.

However, quality is also deeply influenced by perception. Users evaluate software based on their experience, ease of use, reliability, and satisfaction. Even if a software system meets all technical requirements, it may still be perceived as low quality if it is difficult to use or fails to meet user expectations. Factors such as user interface design, response speed, and trustworthiness strongly affect perceived quality.

Therefore, quality is both a fact and a perception. True software quality exists only when objective technical measures align with subjective user satisfaction. Ignoring either aspect can result in an incomplete assessment of quality.

Quality is a multidimensional concept that acts as both a fact and a perception.

1. Quality as a Fact:
    - This is the objective, measurable aspect of quality.
    - It involves compliance with specifications and standards.
    - *Example:* If a software requirement states "The page must load in under 2 seconds," and it loads in 1.8 seconds, the quality is a fact. It can be measured and proven.
2. Quality as a Perception:
    - This is the subjective, emotional aspect of quality based on the user's experience.
    - It relies on expectations, aesthetics, and "look and feel."

NOTE

- *Example:* Two apps may both function correctly (fact), but users might say App A is "higher quality" simply because the color scheme is pleasing or the navigation feels "intuitive" (perception).

Key Takeaway: Delivering high-quality software requires satisfying both the specifications (Fact) and the customer's expectations (Perception).

Activity Title: Identifying Software Quality Assurance Practices

Objective:

To understand the importance of Software Quality Assurance (SQA) and identify quality assurance activities used during software development.

Materials Required:

- Notebook and pen
- Internet or textbook/reference material
- Sample software application

Procedure/Steps:

1. Select any commonly used software application, such as a banking app, educational app, or e-commerce website.
2. Identify at least five quality aspects that should be checked before releasing the software.
3. List the SQA activities that could be performed during its development.
4. Discuss how reviews, standards, documentation, and testing contribute to software quality.
5. Prepare a short table showing the quality aspect and corresponding SQA activity.

Outcome:
The learner will be able to identify major SQA practices and understand their contribution to software quality.

1.3.3 Definitions of Quality

Quality has been defined differently by various experts and standard organizations, reflecting multiple perspectives. The International Organization for Standardization (ISO) defines quality as the degree to which a set of inherent characteristics fulfills requirements. This definition emphasizes conformity to documented needs.

NOTE

The IEEE defines software quality as the degree to which a system, component, or process meets specified requirements and customer expectations. This definition highlights both technical and user-oriented aspects. Philip Crosby defined quality as conformance to requirements, stressing the importance of doing things right the first time. Joseph Juran described quality as fitness for use, focusing on the product's ability to serve its intended purpose. Together, these definitions show that quality is not only about correctness but also about usability, reliability, and customer satisfaction.

Different experts and standards bodies define quality differently. Here are the most critical definitions for a student to know:

- **Philip Crosby:** "Quality is conformance to requirements." (Focus: Do what you said you would do).
- **Joseph Juran:** "Quality is fitness for use." (Focus: Does it solve the customer's problem?).
- **ISO 8402 / ISO 9000:** "The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs."
- **Jerry Weinberg:** "Quality is value to some person." (Focus: Subjective value).

1.3.4 Core Components of Quality

Software quality is determined by several core components that collectively define its effectiveness and acceptance. Correctness ensures that the software performs the required functions accurately according to specifications. Reliability refers to the ability of software to perform consistently without failure over a specified period.

Efficiency measures how well the software utilizes system resources such as memory, processing power, and network bandwidth. Usability focuses on how easily users can understand, learn, and operate the software. Maintainability refers to the ease with which software can be modified to fix defects, improve performance, or adapt to new requirements. Security ensures that the software protects data and resources from unauthorized access and misuse.

A high-quality software system balances all these components rather than optimizing only one aspect. When we discuss "Software Quality," we are usually referring to a specific set of attributes. The ISO 9126 (and later ISO 25010) standard breaks these components down:

1. **Functionality:** Does the software provide the functions needed? (Suitability, Accuracy, Interoperability).
2. **Reliability:** Can the software maintain performance over time? (Maturity, Fault tolerance, Recoverability).
3. **Usability:** Is it easy to understand, learn, and operate?

NOTE

4. **Efficiency:** Does it use appropriate resources (time, memory) relative to its performance?
5. **Maintainability:** How easy is it to modify, fix, or upgrade the software? (Testability, Changeability).
6. **Portability:** Can the software be transferred from one environment to another easily?

1.3.5 Quality View

Quality can be interpreted differently depending on the perspective from which it is viewed. The user view defines quality in terms of user satisfaction, ease of use, and usefulness. If users are satisfied, the software is considered high quality.

The product view focuses on internal attributes such as code structure, design quality, and performance efficiency. The manufacturing view emphasizes adherence to standards, procedures, and specifications during development. The value-based view considers quality in relation to cost, assessing whether the software provides value for money.

Understanding these different views helps organizations make balanced decisions and avoid focusing on only one dimension of quality.

View	Description
1. Transcendental View	Quality is something you instantly recognize but cannot define precisely (e.g., "I know good art when I see it"). In software, this is often the "elegance" of code or design.
2. User View	Quality lies in the eyes of the beholder. If the user is satisfied, the quality is high. This focuses on "fitness for use."
3. Manufacturing View	Quality is conformance to specifications. If the software has zero deviations from the design document, it has high quality, regardless of whether the user likes it.
4. Product View	Quality is viewed as a precise and measurable variable tied to the internal characteristics of the product (e.g., more features = higher quality).
5. Value-based View	Quality is defined in terms of costs and prices. A quality product provides performance at an acceptable price (Value = Benefit / Cost).

1.3.6 Financial Aspect of Quality

NOTE

Quality has a direct and significant impact on the cost of software development and maintenance. The Cost of Quality (CoQ) includes all costs associated with ensuring and maintaining software quality. Prevention costs include expenses related to training, process improvement, and quality planning. Appraisal costs include testing, inspections, and audits.

Failure costs arise when defects are not prevented. Internal failure costs occur when defects are detected before delivery, while external failure costs occur after delivery and may lead to customer dissatisfaction, loss of reputation, legal penalties, and maintenance expenses. Investing in prevention reduces overall cost and improves long-term profitability.

Quality is not free, but *poor* quality costs even more. The **Cost of Quality (CoQ)** is a financial model used to evaluate the costs associated with ensuring quality and the costs resulting from poor quality.

CoQ is divided into two categories:

1. Cost of Control (Good Quality Costs):

- Prevention Costs: Money spent to prevent defects from occurring in the first place.
    - *Examples:* Training, design reviews, requirements analysis, quality planning.
- Appraisal Costs: Money spent to detect defects before shipping.
    - *Examples:* Testing, code inspections, audits.

2. Cost of Failure of Control (Poor Quality Costs):

- Internal Failure Costs: Defects found *before* the product reaches the customer.
    - *Examples:* Rework, bug fixing, re-testing, scrapped code.
- External Failure Costs: Defects found *after* the product reaches the customer. These are the most expensive.
    - *Examples:* Customer support, warranty claims, lawsuits, loss of brand reputation.

Rule of Thumb: It is significantly cheaper to spend money on Prevention than on External Failure.

1.3.7 Customers, Suppliers, and Processes

Customers are the ultimate judges of software quality, as they define requirements and evaluate satisfaction. Their feedback is essential for continuous improvement. Suppliers include developers, testers, vendors, and third-party service providers who contribute to software development.

Processes define how software is developed, tested, delivered, and maintained. Well-documented and standardized processes ensure

NOTE

consistency, reduce errors, and improve quality. Effective coordination among customers, suppliers, and processes is essential for delivering high-quality software products that meet expectations and business goals.

Quality Assurance operates within a specific ecosystem often described by the **SIPOC** model (Supplier, Input, Process, Output, Customer). In software development:

1. The Process:

This is the set of activities that transforms inputs into outputs. In software, this includes coding, designing, and testing.

2. The Supplier:

The entity that provides the input to the process.

- *Internal Supplier:* A business analyst providing requirements to the developer.
- *External Supplier:* A third-party vendor providing an API or database service.

3. The Customer:

The entity that receives the output of the process.

- *Internal Customer:* The QA team is the "customer" of the development team (receiving builds to test).
- *External Customer:* The end-user who buys the software.

Quality Chain Relationship:

Quality is a chain reaction. If a Supplier provides poor inputs (e.g., ambiguous requirements), the Process will be flawed, and the Customer will receive a poor Output. QA acts as the gatekeeper in this chain to ensure that inputs and outputs meet the required standards at every stage.

NOTE

Check Your Progress

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. Which of the following best defines Software Quality Assurance (SQA)?
A) Writing program code faster
B) A process to ensure software meets required quality standards
C) Debugging errors after software release
D) Designing user interface screens

_____

_____

Q2. SQA is only performed after the software development is completed.

_____

Q3. SQA focuses mainly on improving the _____ and preventing defects.

_____

Q4. Write any two objectives of Software Quality Assurance.

Answer (Any two):

1. To ensure software meets customer requirements and standards

2. To prevent defects by improving development processes

3. To ensure proper documentation and compliance

4. To improve reliability and maintainability of software

_____

_____

Q5. Which one is NOT an SQA activity?
A) Quality audits
B) Process monitoring
C) Code debugging only after final release
D) Reviews and inspections

Fill in the blanks:

Software Quality Assurance focuses on _____ defects.

Quality as perception depends on _____ satisfaction.

True/False:

Quality is only a technical concept.

Prevention costs reduce overall software development cost.

NOTE

1.4 SUMMARY

This unit provided an introduction to the fundamental concepts of Software Quality Assurance and software quality. It highlighted the growing importance of quality in software systems due to increased dependency on digital technologies. Software Quality Assurance was presented as a systematic and preventive approach that integrates quality practices throughout the software development life cycle.

The objectives of SQA were discussed, emphasizing defect prevention, standard compliance, cost reduction, and customer satisfaction. The concept of software quality was explained as a multidimensional attribute that includes both technical correctness and user perception. The unit stressed that quality is influenced by development processes as well as the final software product.

Overall, this unit establishes a strong foundation for understanding the principles of Software Quality Assurance. The concepts discussed here prepare learners to study advanced quality models, standards, and techniques used in professional software development environments.

1.5 GLOSSARY

- **Software Quality Assurance (SQA)**: Planned activities to ensure software quality.
- **Defect**: An error or flaw in software.
- **Reliability**: Ability of software to function without failure.
- **Usability**: Ease of use of a software system.
- **Cost of Quality (CoQ)**: Total cost associated with ensuring quality.

1.6 ANSWER TO CHECK YOUR PROGRESS-POSSIBLE ONE

Q1. Answer: B

Explanation: SQA is a planned and systematic set of activities to ensure the software product and process meet specified quality requirements.

Q2. Answer: False

Explanation: SQA is performed throughout the software development life cycle (SDLC), not only at the end. It includes reviews, audits, standards, and process control.

Q3. Answer: Process

NOTE

Explanation: SQA is process-oriented. It ensures the development process is correct so that defects are prevented before they occur.

Q4. Explanation: The main goal of SQA is to deliver quality software by controlling and improving the process used for development.

Q5. Answer: C

Explanation: Debugging only after final release is not a proper SQA activity. SQA emphasizes early defect prevention using reviews, audits, and process monitoring.

Fill in the blanks:

Answer: preventing

Answer: user

True/False:

Answer: False

Answer: True

1.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3780	

1.8 REFERENCE

- Pressman, R. S., *Software Engineering: A Practitioner's Approach*
- Sommerville, I., *Software Engineering*, Pearson
- ISO 9001 Quality Management Standards

1.9 EXERCISE

Part–1

A. Multiple Choice Questions (MCQs)

Q1. Software Quality Assurance (SQA) mainly focuses on:

A) Detecting defects after delivery
B) Preventing defects by improving processes

NOTE

C) Increasing the software size
D) Reducing documentation

Answer: B
Explanation: SQA is process-oriented and aims to prevent defects by using standards, audits, reviews, and proper procedures.

Q2. Which of the following is an SQA activity?

A) Coding only
B) Quality audits and reviews
C) Running the software without test cases
D) Increasing hardware speed

Answer: B
Explanation: SQA includes reviews, inspections, audits, and process monitoring to ensure quality in every SDLC phase.

Q3. Quality in software is best defined as:

A) Number of lines of code
B) Ability of software to satisfy stated and implied needs
C) Amount of money spent on development
D) Number of developers in the team

Answer: B
Explanation: Software quality is the degree to which the product meets customer requirements and expectations.

Q4. The cost of fixing defects is usually:

A) Lowest during maintenance
B) Lowest after release
C) Lowest during early stages of SDLC
D) Same in all phases

Answer: C
Explanation: Defects are cheaper to fix during requirements and design stages. Late defect fixing increases cost significantly.

Q5. Which one is NOT a core component of quality?

A) Customer satisfaction
B) Process improvement
C) Defect prevention
D) Increasing software complexity

Answer: D
Explanation: Increasing complexity reduces maintainability and increases defect probability, so it is not part of quality.

B. Small Answer Type Questions

1. Define Software Quality Assurance (SQA).

NOTE

2. Write any two objectives of SQA.
3. What is the difference between Quality Assurance and Quality Control?
4. What is meant by “quality as perception”?
5. Write any two financial aspects of quality in software development.

Part–2

C. Long Answer Type Questions (200 words each)

1. Explain the importance of Software Quality Assurance in the Software Development Life Cycle (SDLC).
2. Discuss the historical perspective of quality and how it evolved in software engineering.
3. Explain different views of quality and their relevance in software development.
4. Describe the core components of software quality with suitable examples.
5. Discuss the relationship between customers, suppliers, and processes in achieving software quality.

Part–3

D. Long Answer Type Questions (300 words each)

1. Explain the concept of software quality in detail. Discuss whether quality is a fact or perception with examples.
2. Describe various definitions of quality given by quality experts and explain their relevance in software quality assurance.
3. Discuss the financial aspect of quality in software projects. Explain cost of quality and impact of poor quality.
4. Explain how Software Quality Assurance activities help in defect prevention and process improvement throughout SDLC.
5. Discuss the role of customers and stakeholders in determining software quality and ensuring quality assurance.

UNIT-2
TOTAL QUALITY MANAGEMENT

UNIT STRUCTURE

2.1 Introduction
2.2 Objective
2.3 Total Quality Management (TQM)
  2.3.1 Quality Principles of Total Quality Management
  2.3.2 Quality Management Through Statistical Process Control
  2.3.3 Quality Management Through Cultural Changes
  2.3.4 Continual (Continuous) Improvement Cycle
2.4 Summary
2.5 Glossary
2.6 Answer to check your progress
2.7 Suggested readings - OER
2.8 Reference
2.9 Exercise

2.1 INTRODUCTION

In software engineering, quality is a critical success factor because software systems are often complex, widely used, and highly dependent on correctness and reliability. Software defects can lead to system failures, data breaches, financial losses, and damage to organizational reputation. As software products evolve continuously through updates and enhancements, maintaining consistent quality becomes a major challenge. This challenge has led to the adoption of Total Quality Management (TQM) principles in Software Quality Assurance (SQA) and software testing.

Total Quality Management is a holistic management philosophy that emphasizes quality at every stage of the software development life cycle (SDLC). In the context of software quality assurance, TQM ensures that quality is built into requirements analysis, design, coding, testing, deployment, and maintenance. Unlike traditional approaches that rely heavily on testing at the end of development, TQM promotes defect prevention, process improvement, and continuous feedback.

In software testing, TQM encourages early involvement of testers, systematic test planning, data-driven decision-making, and continuous

NOTE

improvement of testing processes. By integrating TQM with SQA and testing, organizations can reduce rework, improve software reliability, and enhance customer satisfaction.

Total Quality Management (TQM) is a comprehensive management philosophy that seeks to integrate all organizational functions (marketing, finance, design, engineering, and customer service) to focus on meeting customer needs and organizational objectives.

Unlike traditional quality control, which focuses on inspecting the final product, TQM focuses on process improvement and employee involvement at every level. In the context of software, TQM means that quality is not just the responsibility of the "Tester"; it is the responsibility of the requirement analyst, the developer, the project manager, and the support staff.

Key Characteristics of TQM:

1. Customer-Defined Quality: The customer determines the level of quality.
2. Total Employee Involvement: All employees participate in working toward common goals.
3. Integrated System: All processes are interconnected.

2.2 OBJECTIVE

After studying this unit, you will be able to:

- **Define** Total Quality Management (TQM) and its key concepts.
- **Explain** the principles and objectives of TQM.
- **Identify** major TQM practices applicable to software development.
- **Differentiate** between traditional quality management and TQM.

2.3 TOTAL QUALITY MANAGEMENT (TQM) IN SOFTWARE QUALITY ASSURANCE

In software quality assurance, Total Quality Management is an organization-wide approach that integrates quality principles into all software engineering activities. TQM treats software development and testing as interconnected processes rather than isolated phases. It emphasizes continuous monitoring, feedback, and improvement of these processes.

NOTE

TQM in SQA ensures that quality standards, coding guidelines, testing strategies, and review mechanisms are consistently applied across projects. It encourages collaboration between development and testing teams, ensuring early detection of issues and alignment with customer requirements. Through TQM, quality becomes a shared responsibility rather than the sole responsibility of testers.

2.3.1 Quality Principles of Total Quality Management in SQA and Testing

The principles of Total Quality Management play a vital role in improving software quality assurance and testing practices. The principle of customer focus ensures that software requirements are clearly understood, validated, and tested against real user needs. Test cases are derived not only from specifications but also from user scenarios and acceptance criteria.

Leadership commitment is essential for establishing a quality-driven software development environment. Management must support quality initiatives by allocating resources for testing tools, training, and process improvement. Without management support, SQA activities often receive low priority and inadequate time.

Employee involvement ensures that developers, testers, analysts, and project managers actively participate in quality improvement. Testers are encouraged to provide feedback on design and code quality, while developers collaborate with testers to resolve defects efficiently.

The process-oriented approach focuses on improving software development and testing processes rather than blaming individuals for defects. Standardized testing processes, test documentation, code reviews, and defect tracking systems are used to achieve consistency. Continuous improvement ensures that lessons learned from previous projects are used to improve future testing and quality assurance activities.

TQM is built on a foundation of core principles (often aligned with ISO 9000 standards). These principles guide organizational management.

1. **Customer Focus:** The primary focus of quality management is to meet customer requirements and strive to exceed customer expectations. In software, this means User Acceptance Testing (UAT) and constant feedback loops are critical.
2. **Leadership:** Leaders at all levels establish unity of purpose and direction. They create conditions in which people are engaged in achieving the organization's quality objectives.

NOTE

3. **Engagement of People:** Competent, empowered, and engaged people at all levels are essential to enhance the organization's capability to create and deliver value.
4. **Process Approach:** Consistent and predictable results are achieved more effectively when activities are understood and managed as interrelated processes.
5. **Improvement:** Successful organizations have an ongoing focus on improvement (Kaizen).
6. **Evidence-based Decision Making:** Decisions based on the analysis and evaluation of data and information are more likely to produce desired results. (e.g., using defect metrics to decide if a product is ready for release).

Relationship Management: For sustained success, an organization manages its relationships with interested parties, such as suppliers and partners.

Activity Title: Applying TQM Principles to Software Development

Objective:

To understand and apply the basic principles of Total Quality Management (TQM) to software development.

Materials Required:

- Notebook and pen
- Textbook/reference material

Procedure/Steps:

1. Identify a software project such as a college management system.
2. List the major stakeholders involved in the project.
3. Identify at least five quality requirements.
4. Explain how customer focus, continuous improvement, teamwork, and employee involvement can be applied.
5. Prepare a brief TQM improvement plan for the selected project.

Outcome:

The learner will understand how TQM principles can be applied to achieve continuous improvement in software quality.

2.3.2 Quality Management Through Statistical Process Control in Software Testing

NOTE

Statistical Process Control (SPC) is widely used in software quality assurance to monitor and improve testing and development processes using quantitative data. In software testing, SPC helps track metrics such as defect density, test coverage, test execution time, defect leakage, and rework effort.

Control charts are used to observe trends in defect discovery and resolution rates. If abnormal variations are detected, root cause analysis is performed to identify issues such as inadequate test cases, poor code quality, or unclear requirements. SPC helps distinguish between normal process variations and serious quality problems.

By applying SPC, SQA teams can make data-driven decisions, predict potential risks, and improve test planning. Statistical analysis supports continuous improvement by identifying weak areas in testing processes and enabling timely corrective actions.

Statistical Process Control (SPC) involves using statistical techniques to measure and analyze the variation in a process. The goal is to determine if a process is stable ("in control") or unstable ("out of control").

Software processes (like coding and testing) produce data: number of defects found per hour, lines of code written, time to fix a bug, etc. By analyzing this data, we can predict future performance.

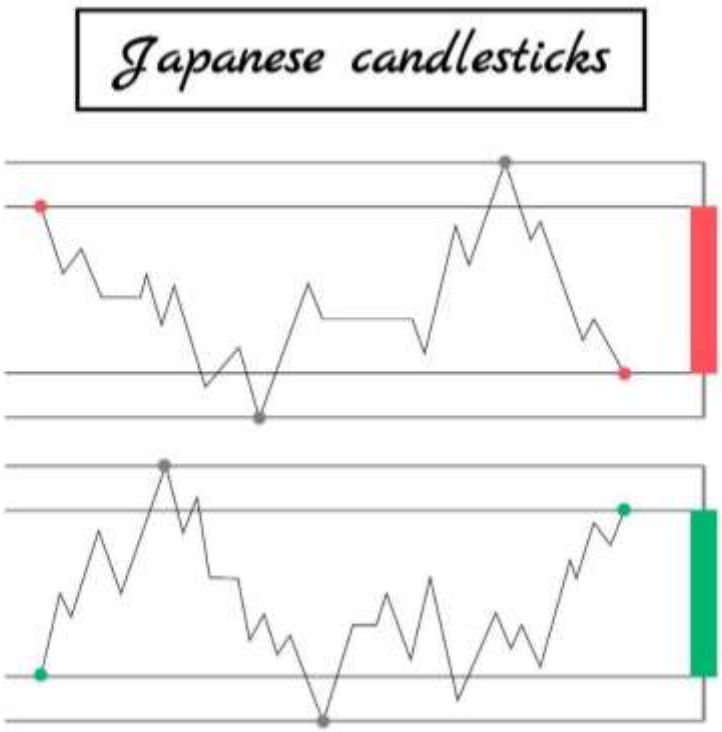


Fig.2.1 : An Example of Japanese Candlesticks

Key Concept: Variation

NOTE

- Common Cause Variation: Natural variations inherent in the process (e.g., minor fluctuations in coding speed).
- Special Cause Variation: Unusual events that disrupt the process (e.g., server crash, a flu outbreak taking out the dev team).

The Control Chart:

The primary tool of SPC is the Control Chart. It plots data over time with three key lines:

1. Center Line (CL): The average performance.
2. Upper Control Limit (UCL): The maximum acceptable variation.

Lower Control Limit (LCL): The minimum acceptable variation.

2.3.3 Quality Management Through Cultural Changes in Software Organizations

Successful implementation of TQM in software quality assurance requires a strong shift in organizational culture. Traditionally, testing is viewed as a separate phase or a fault-finding activity. TQM promotes a culture where testing is seen as a value-adding process that improves software quality.

A quality-oriented software culture encourages collaboration between developers and testers, open communication, and early involvement of QA teams. Employees are encouraged to report defects without fear of blame, promoting transparency and trust. Continuous learning through training in testing tools, quality standards, and automation techniques strengthens this culture.

Implementing TQM is less about tools and more about culture. A shift from a "Detection Culture" to a "Prevention Culture" is required.

1. From "Police" to "Partners"

- *Old Culture:* QA is the "police" force that catches developers making mistakes. Developers hide bugs to avoid blame.
- *TQM Culture:* QA and Development are partners. Finding a bug is a collaborative success, not a personal failure.

2. Fear vs. Empowerment

- *Old Culture:* Management by fear. "If you release a bug, you are fired."
- *TQM Culture:* W. Edwards Deming, a father of TQM, famously said: "Drive out fear." Employees must feel safe to report problems early. They must be empowered to stop the production line (or the release pipeline) if they see a quality issue.

NOTE

3. Silos vs. Cross-Functional Teams

- *Old Culture:* Developers write code and "throw it over the wall" to testers.

TQM Culture: Cross-functional teams (like Agile squads) work together from Day 1. Quality is discussed during requirements gathering, not just at the end.

Management plays a crucial role in fostering a quality culture by recognizing quality achievements, supporting continuous improvement initiatives, and encouraging innovation in testing practices. A positive quality culture ensures that SQA and testing activities are integrated naturally into daily development work.

2.3.4 Continual (Continuous) Improvement Cycle in Software Quality Assurance

Continuous improvement is central to Total Quality Management in software quality assurance and testing. The Plan–Do–Check–Act (PDCA) cycle is widely used to improve testing and QA processes systematically. In the Plan phase, quality goals such as reducing defect leakage or improving test coverage are defined**.**

The Do phase involves implementing process changes, such as introducing automated testing or improving test case design. In the Check phase, test metrics and performance data are analyzed to evaluate effectiveness. In the Act phase, successful improvements are standardized and applied across projects.

Continuous improvement ensures that SQA and testing processes evolve with changing technologies, tools, and customer expectations. It helps organizations reduce defects, improve delivery timelines, and enhance overall software quality.

Continuous improvement is the heartbeat of TQM. It is based on the belief that processes can always be optimized. The standard model for this is the PDCA Cycle (also known as the Deming Cycle or Shewhart Cycle).

NOTE



Fig. 2.2 : PDCA Cycle in Software Quality

The Four Stages:

1. **Plan:**
    - Identify the problem or opportunity.
    - Analyze the current situation (root cause analysis).
    - Develop a plan for improvement.
    - *Software Example:* "Our regression testing takes 3 days. We plan to automate 50% of it to reduce time to 1 day."
2. **Do:**
    - Implement the plan on a small scale (pilot).
    - Execute the process changes.
    - *Software Example:* "Write automation scripts for the login module first and run them."
3. **Check (or Study):**
    - Measure the results of the pilot against the expected results.
    - Did the plan work? Were there side effects?
    - *Software Example:* "The login tests ran in 10 minutes, but they were flaky and failed 20% of the time."
4. **Act:**
    - If successful, standardize the change across the organization.
    - If unsuccessful, adjust the plan and repeat the cycle.

NOTE

- *Software Example:* "Fix the flakiness in the scripts (Act), then expand automation to the rest of the suite (New Plan)."

The Japanese term *Kaizen* ("Change for the better") is often used here. It implies small, incremental, daily improvements rather than waiting for one massive technological breakthrough.

Check Your Progress 2.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. Total Quality Management (TQM) mainly focuses on:

A) Only product inspection
B) Only testing after development
C) Continuous improvement in all organizational activities
D) Increasing software size

_____

Q2. TQM is the responsibility of only the Quality Department.

_____

_____

Q3. The continuous improvement cycle in TQM is commonly known as the _____ cycle.

_____

_____

Q4. Write any two key principles of Total Quality Management.

Answer (Any two):

Customer focus

Continuous improvement

Employee involvement

Process-centered approach

Strong leadership commitment

_____

Q5. Statistical Process Control (SPC) in TQM is used to:

A) Increase project cost
B) Monitor and control process variation
C) Replace software testing
D) Remove documentation work

_____

Fill in the Blanks

TQM in software emphasizes _____ prevention rather than detection.

NOTE

SPC uses _____ data for process improvement

True/False

Testing alone is sufficient to ensure software quality.

Continuous improvement is a key principle of TQM in SQA.

2.4 SUMMARY

Total Quality Management provides a strong framework for improving software quality assurance and testing practices. By emphasizing customer focus, process improvement, data-driven decision-making, and continuous improvement, TQM helps organizations build quality into software products from the beginning. Integration of TQM with SQA and testing reduces defects, improves reliability, and ensures long-term software quality.

2.5 GLOSSARY

- **TQM**: Organization-wide approach to quality improvement.
- **SQA**: Planned activities to ensure software quality.
- **SPC**: Statistical techniques used to monitor processes.
- **PDCA Cycle**: Continuous improvement model used in quality management.
- **Defect Leakage**: Defects that escape testing and reach production.

2.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/User/UnitUrls?unitId=3781	

2.6 ANSWERS TO CHECK YOUR PROGRESS – POSSIBLE ONE

Q1. Answer: C

NOTE

Explanation: TQM is a management approach that focuses on continuous improvement in every process to achieve long-term customer satisfaction.

Q2. Answer: False

Explanation: In TQM, quality is everyone's responsibility — management, developers, testers, and support staff.

Q3. Answer: PDCA (Plan–Do–Check–Act)

Explanation: PDCA is a widely used method for continuous improvement and quality management.

Q4. Explanation: TQM principles ensure that the organization maintains quality by focusing on customer satisfaction and improving processes continuously.

Q5. Answer: B

Explanation: SPC uses statistical techniques to monitor processes and reduce variations, which improves overall quality.

Fill in the Blanks

Answer: defect

Answer: statistical

True/False

Answer: False

Answer: True

2.8 REFERENCE

- Pressman, R. S., *Software Engineering: A Practitioner's Approach*
- Sommerville, I., *Software Engineering*
- Deming, W. E., *Out of the Crisis*
- Juran, J. M., *Quality Planning and Analysis*

2.9 EXERCISE

Part–1

A. Multiple Choice Questions (MCQs)

Q1. Total Quality Management (TQM) mainly focuses on:

A) Only product inspection
B) Continuous improvement in all organizational activities
C) Testing only after coding
D) Increasing project cost

NOTE

Answer: B

Explanation: TQM is a management approach aimed at continuous improvement of processes, people, and products to achieve customer satisfaction.

Q2. The PDCA cycle stands for:

A) Plan–Do–Check–Act
B) Prepare–Develop–Control–Apply
C) Plan–Design–Code–Analyze
D) Process–Develop–Check–Assess

Answer: A

Explanation: PDCA is a continuous improvement cycle used in quality management for planning, implementing, evaluating, and improving processes.

Q3. In TQM, quality is the responsibility of:

A) Only QA department
B) Only developers
C) Only management
D) Everyone in the organization

Answer: D

Explanation: TQM emphasizes organization-wide involvement, meaning every employee contributes to quality.

Q4. Statistical Process Control (SPC) is used to:

A) Increase software size
B) Monitor and control process variation
C) Remove documentation
D) Avoid testing

Answer: B

Explanation: SPC uses statistical methods to monitor process performance and reduce variation, improving quality.

Q5. Which of the following is NOT a principle of TQM?

A) Customer focus
B) Continuous improvement
C) Employee involvement
D) Ignoring process standards

Answer: D

Explanation: TQM strongly supports following standards and improving processes, not ignoring them.

B. Small Answer Type Questions (5 Questions)

(Answer in 2–4 lines)

NOTE

1. Define Total Quality Management (TQM).
2. Write any two principles of TQM.
3. What is the role of leadership in TQM?
4. What is meant by continuous improvement?
5. Write any two benefits of implementing TQM in an organization.

Part–2

C. Long Answer Type Questions (200 words each)

1. Explain the concept of Total Quality Management and its importance in organizations.
2. Discuss the quality principles of TQM and explain how they improve organizational performance.
3. Explain the role of Statistical Process Control (SPC) in quality management.
4. Describe how cultural changes support successful implementation of TQM.
5. Explain the continuous improvement cycle (PDCA) and its application in software organizations.

Part–3

D. Long Answer Type Questions (300 words each)

1. Discuss TQM in detail and explain how it can be applied in software development and IT services.
2. Explain how statistical quality management methods help in monitoring and improving software processes.
3. Describe the importance of organizational culture in achieving quality and customer satisfaction through TQM.
4. Explain the PDCA cycle in detail with suitable examples from software project management.
5. Discuss the challenges in implementing TQM in modern organizations and suggest suitable solutions.

UNIT-3
QUALITY IN DIFFERENT AREAS

UNIT STRUCTURE

3.1 Introduction
3.2 Objective
3.3 Quality in Different Areas
  3.3.1 Benchmarking and Metrics
  3.3.2 Problem Solving Techniques
  3.3.3 Problem Solving Software Tools
3.4 Summary
3.5 Glossary
3.6 Answer to check your progress
3.7 Suggested readings - OER
3.8 Reference
3.9 Exercise

3.1 INTRODUCTION

In software engineering, quality is not limited to the final product alone. It extends across various areas such as processes, people, tools, documentation, and management practices. Each of these areas plays a crucial role in determining the overall quality of software systems. Poor quality in any one area may lead to defects, delays, increased costs, and dissatisfied users.

Software Quality Assurance (SQA) focuses on ensuring that quality is maintained consistently across all stages of the Software Development Life Cycle (SDLC). To achieve this, organizations adopt structured approaches such as benchmarking, measurement through metrics, systematic problem-solving techniques, and the use of specialized software tools. These approaches help in identifying weaknesses, improving performance, and maintaining continuous quality improvement.

Software Quality Assurance and Testing is not limited to checking whether a program runs correctly. In real software projects, quality is a broad concept that must be ensured in every area of software development and management. This unit focuses on understanding how quality is achieved and evaluated in different areas of software work, especially through benchmarking, metrics, and structured

NOTE

problem-solving approaches. The central idea of this unit is that quality is not accidental; it is the result of disciplined processes, measurement, and continuous improvement.

Quality in software is often misunderstood as only “bug-free software.” In practice, quality includes correctness, reliability, usability, efficiency, maintainability, portability, security, and performance. A software product may work correctly but still be considered low-quality if it is difficult to use, slow, insecure, or hard to modify. Therefore, modern software engineering defines quality as the ability of the software to satisfy stated and implied requirements under specified conditions. This includes both functional requirements, which describe what the system should do, and non-functional requirements, which describe how well the system should perform.

Another important point is that quality must be ensured in different areas of the organization and the project. Quality is not only the responsibility of testers. Developers, analysts, designers, project managers, and customers all influence software quality. For example, unclear requirements create defects even before coding begins. Poor design choices may cause performance and scalability problems. Weak coding practices may introduce security vulnerabilities. Inadequate testing may allow defects to reach users. Similarly, poor project management may lead to rushed delivery, reduced verification activities, and compromised product quality.

To ensure quality in different areas, organizations adopt quality frameworks and improvement models such as ISO standards, CMMI, Six Sigma, and Total Quality Management. These approaches encourage organizations to define processes, measure outcomes, and improve continuously. However, implementing such models requires practical tools and techniques. Two major elements that support quality improvement are benchmarking and metrics. Benchmarking helps an organization compare its performance against industry standards or best practices. Metrics provide measurable data about software processes and products. Together, benchmarking and metrics make quality measurable and manageable.

Along with measurement, problem-solving plays a major role in quality assurance. Whenever defects, failures, or process issues arise, organizations must identify the root cause and eliminate it. Problem solving is not a random activity; it requires structured techniques such as root cause analysis, cause-and-effect diagrams, Pareto analysis, and brainstorming. These techniques help teams systematically analyze problems rather than relying on assumptions. Today, many software tools also support problem-solving activities. These tools help in defect tracking, reporting, data analysis, statistical evaluation, and process improvement.

NOTE

The unit introduces quality in different areas of software work by explaining benchmarking and metrics, problem-solving techniques, and problem-solving software tools. It also highlights how these concepts are connected and how they contribute to achieving consistent and high-quality software products.

This unit discusses how quality is managed in different areas of software development and testing, with special emphasis on benchmarking, metrics, problem-solving techniques, and software tools used for quality improvement.

3.2 OBJECTIVE

After studying this unit, you will be able to:

- **Define** quality in the context of software development.
- **Identify** quality requirements in different software development areas.
- **Explain** quality considerations in requirements, design, coding, testing, and maintenance.
- **Differentiate** between quality measures used in different development areas.
- **Analyse** the impact of quality practices on the overall software product.

3.3 QUALITY IN DIFFERENT AREAS

Quality in software engineering must be addressed in multiple areas, including requirements engineering, design, coding, testing, and maintenance. To manage quality effectively in these areas, organizations rely on measurement, analysis, and systematic problem-solving approaches. Benchmarking, metrics, and quality tools form the foundation of quality management in software projects.

Quality in software engineering must be addressed in multiple areas because software systems are complex and involve many stages. Each stage contributes to the final quality of the product. Therefore, quality must be ensured in requirements, design, coding, testing, deployment, and maintenance. At the same time, quality must also be ensured in project management and organizational processes. If quality is ensured only at the end of development, defects become expensive and difficult to fix. Hence, quality must be built into the product from the beginning.

Quality in requirements means that requirements should be complete, correct, consistent, unambiguous, and verifiable. If requirements are not clear, the development team may build the wrong product, and no

NOTE

amount of testing can fully correct that mistake. Quality in design means that the system architecture should be robust, modular, scalable, and maintainable. Design quality also ensures that security, performance, and usability concerns are addressed early. Quality in coding means that code should follow standards, be readable, maintainable, and efficient. It should also be free from common errors such as memory leaks, concurrency issues, and vulnerabilities. Quality in testing means that test cases should be systematic, complete, and aligned with requirements. Testing must include functional testing as well as non-functional testing such as performance testing, security testing, usability testing, and reliability testing.

Quality in deployment and maintenance ensures that the software works correctly in real environments and continues to deliver value over time. Maintenance quality includes quick defect fixes, efficient upgrades, and minimal disruption to users. Quality in project management ensures that schedules, resources, and risks are handled properly. Poor management may lead to rushed delivery and reduced quality activities. Quality in organizational processes ensures that best practices are followed consistently across projects.

To manage quality across these different areas, organizations rely on measurement and improvement practices. Benchmarking and metrics provide measurable standards for quality. Problem-solving techniques and tools support defect prevention and continuous improvement. The following subsections explain these important components in detail.

3.3.1 Benchmarking and Metrics

Benchmarking is the process of comparing an organization's software processes, practices, and performance with industry standards or best-performing organizations. In Software Quality Assurance, benchmarking helps organizations understand their current quality level and identify areas for improvement. For example, defect density, test coverage, and release cycle time can be compared with industry benchmarks to evaluate performance.

Benchmarking may be internal, where different projects within the same organization are compared, or external, where comparisons are made with industry standards such as ISO, CMMI, or industry reports. Through benchmarking, software organizations can adopt best practices and improve their quality processes.

Metrics are quantitative measures used to assess software quality and process effectiveness. In software testing, common metrics include number of defects found, defect severity, defect leakage, test case effectiveness, and code coverage. These metrics help SQA teams monitor progress, identify trends, and make informed decisions.

NOTE

Metrics must be well-defined, measurable, and relevant to quality objectives. Improper or excessive use of metrics may lead to incorrect conclusions. When used correctly, benchmarking and metrics provide valuable insights for continuous quality improvement.

Benchmarking and metrics are essential for managing quality because they convert abstract quality concepts into measurable and comparable values. Without measurement, quality improvement becomes difficult because there is no objective way to determine whether the organization is improving or declining. Benchmarking and metrics provide the foundation for informed decision-making and continuous process improvement.

Benchmarking is the process of comparing the performance of an organization, project, or product against a standard or against other organizations considered leaders in the industry. The goal of benchmarking is not only to compare but also to identify gaps and adopt best practices. Benchmarking helps an organization understand where it stands in terms of quality and what improvements are needed to reach a desired level. It is widely used in quality management because it promotes learning from successful organizations and helps establish realistic improvement targets.

There are different types of benchmarking used in software quality assurance. Internal benchmarking compares different projects or teams within the same organization. This helps identify high-performing teams and learn from them. Competitive benchmarking compares the organization's performance with direct competitors. This can provide insights into how the organization performs in the market. Functional benchmarking compares processes or performance with organizations that may not be direct competitors but are leaders in a particular function. Generic benchmarking compares general processes such as project management or customer support across industries.

In software engineering, benchmarking can be applied to many areas. For example, an organization can benchmark defect density, productivity, release frequency, customer satisfaction, mean time to repair defects, test coverage, or performance response time. Benchmarking is useful because it motivates improvement and helps organizations align their performance with industry expectations. It also supports strategic planning because it provides data-driven direction for investment in tools, training, and process improvement.

Metrics are quantitative measures used to evaluate the quality of software processes, products, and projects. Metrics are important because software quality cannot be ensured only through subjective judgment. Metrics provide objective data that can be analyzed to identify trends, predict risks, and evaluate performance. In software

NOTE

quality assurance, metrics help track defects, measure reliability, evaluate testing effectiveness, and monitor project progress.

Software metrics can be classified into three main categories: product metrics, process metrics, and project metrics. Product metrics measure the characteristics of the software product itself. Examples include size, complexity, performance, reliability, and maintainability. Common product metrics include lines of code, cyclomatic complexity, defect density, and code coverage. Process metrics measure the effectiveness and efficiency of software development and testing processes. Examples include defect removal efficiency, rework effort, review effectiveness, and test execution rate. Project metrics measure project management aspects such as schedule variance, cost variance, resource utilization, and productivity.

A. Benchmarking

Benchmarking is the process of comparing your organization's business processes and performance metrics to industry bests or best practices from other companies. In software quality, it helps answer the question: *"How good are we compared to the competition?"*

Types of Benchmarking:

1. Internal Benchmarking: Comparing one department or project team against another within the same company (e.g., Team A vs. Team B).
2. Competitive Benchmarking: Directly comparing products or processes with direct competitors (e.g., Chrome vs. Firefox performance).
3. Process Benchmarking: Focusing on discrete work processes and systems, such as the "time to fix critical bugs," regardless of industry.
4. Strategic Benchmarking: Examining long-term strategies of high-performing companies to align your own strategic goals.

B. Software Quality Metrics

Metrics are quantitative measures used to estimate the quality of software. "You cannot control what you cannot measure."

Key Metrics to Track:

Metric Category	Metric Name	Description
Product Metrics	Defect Density	Number of defects per KLOC (Thousand Lines of Code). Formula: $Total Defects / Size of Module$.

NOTE

	Mean Time To Failure (MTTF)	The average time a system operates without failing.
Process Metrics	Mean Time To Repair (MTTR)	The average time taken to fix a defect after it is found.
	Defect Removal Efficiency (DRE)	Percentage of bugs found internally vs. bugs found by customers.
Project Metrics	**Code Coverage**	The percentage of code executed during automated testing.

Activity Title: Quality Across Software Development Areas

Objective:
To identify how quality is maintained across different areas of software development.

Materials Required:

- Notebook and pen
- Software development case study

Procedure/Steps:

1. Select a software project.
2. Identify areas such as requirements, design, coding, testing, documentation, and maintenance.
3. For each area, identify two possible quality problems.
4. Suggest suitable quality measures for preventing or controlling those problems.
5. Present the findings in a table.

Outcome:
The learner will be able to recognise quality requirements and quality measures across different software development areas.

3.3.2 Problem Solving Techniques

Problem-solving is a critical activity in Software Quality Assurance, as defects and process issues are inevitable in software projects. Effective problem-solving techniques help identify the root cause of issues rather than merely addressing symptoms.

One widely used technique is root cause analysis, which focuses on understanding why a defect occurred. Techniques such as the Fishbone (Cause-and-Effect) Diagram help categorize possible causes related to people, processes, tools, and environment. Another technique is the 5

NOTE

Whys method, where repeated questioning is used to reach the underlying cause of a problem.

Problem solving is a critical part of software quality assurance because defects and failures are inevitable in software projects. The key objective of quality assurance is not only to detect defects but also to prevent them from recurring. This can be achieved only by systematic problem solving. Problem solving techniques help organizations identify the root causes of defects and process issues, implement corrective actions, and improve future performance.

In software projects, problems may occur in different forms. A problem may be a software defect, a test failure, a customer complaint, a performance issue, a security vulnerability, or a process delay. Sometimes, problems are visible, such as a system crash. Sometimes, problems are hidden, such as slow performance under high load. In both cases, solving the problem requires careful analysis. Random guessing or quick fixes may temporarily solve the issue but may not eliminate the root cause. Therefore, structured problem solving is essential.

One widely used technique is Root Cause Analysis. Root cause analysis focuses on identifying the fundamental cause of a defect rather than only fixing the symptom. For example, if a software module frequently fails due to incorrect input handling, the root cause may be poor input validation design or missing requirement specifications. Root cause analysis involves collecting evidence, analyzing defect patterns, and identifying process weaknesses. It often uses methods such as the “5 Whys,” where the team repeatedly asks why a defect occurred until the underlying cause is identified.

Another common technique is the Cause-and-Effect Diagram, also known as the Fishbone Diagram or Ishikawa Diagram. This technique helps identify possible causes of a problem by categorizing them into groups. In software projects, categories may include people, process, tools, environment, requirements, design, and testing. For example, if a project experiences a high defect rate, possible causes may include insufficient training, unclear requirements, lack of code review, poor test coverage, or unstable development environment. The fishbone diagram helps teams visualize the relationships between causes and effects and supports structured discussion.

Pareto Analysis is another important technique. It is based on the Pareto principle, which states that 80% of problems are often caused by 20% of causes. In software quality, Pareto analysis helps identify the most significant defect types or modules responsible for the majority of defects. For example, if most defects occur in a few critical modules, focusing improvement efforts on those modules can produce the

NOTE

greatest quality improvement. Pareto charts are commonly used to visualize defect frequency and prioritize corrective actions.

Brainstorming is a widely used technique in software teams for generating possible solutions or identifying possible causes. Brainstorming encourages team members to contribute ideas freely without criticism. This technique is useful when problems are complex or when multiple perspectives are needed. Brainstorming can be structured or unstructured. In structured brainstorming, participants contribute ideas in turns. In unstructured brainstorming, ideas are shared freely. The key advantage is that it encourages creativity and collaboration.

Another technique is Failure Mode and Effects Analysis. This method is commonly used in high-reliability systems. It involves identifying possible failure modes, analyzing their effects, and determining their severity and likelihood. In software, failure modes may include incorrect calculations, missing input validation, data corruption, security breaches, or system crashes. By analyzing failure modes early, teams can implement preventive controls and reduce risks.

Pareto analysis is used to identify the most significant problems by applying the 80/20 rule, which states that a small number of causes often lead to most defects. By focusing on these critical issues, quality improvement efforts become more effective. These problem-solving techniques help software teams reduce defect recurrence, improve processes, and enhance overall product quality.

When quality issues arise, simply fixing the symptom is not enough. You must find the root cause. The following techniques are standard in Total Quality Management (TQM).

1. The 5 Whys

A simple but powerful technique developed by Sakichi Toyoda (Toyota). You ask "Why?" five times to peel away layers of symptoms and find the root cause.

Example:

- Problem: The server crashed.
    1. Why? It ran out of memory.
    2. Why? A specific process consumed 100% RAM.
    3. Why? The process entered an infinite loop.
    4. Why? The loop condition was never met.
    5. Why? The input data validation was missing. (Root Cause)

NOTE

2. Ishikawa (Fishbone) Diagram

Also known as the Cause-and-Effect diagram. It helps visualizes all possible causes of a problem. The head of the fish is the "Effect" (Problem), and the bones are the "Causes," typically categorized into the 6 Ms: Man, Machine, Material, Method, Measurement, and Mother Nature (Environment).

3. Pareto Analysis (The 80/20 Rule)

Vilfredo Pareto observed that 80% of effects come from 20% of causes. In software testing, 80% of the defects are usually found in 20% of the code modules.

- Technique: Plot the frequency of defect types on a bar chart. Focus your fixing efforts on the tallest bars first to get the maximum quality improvement for the least effort.

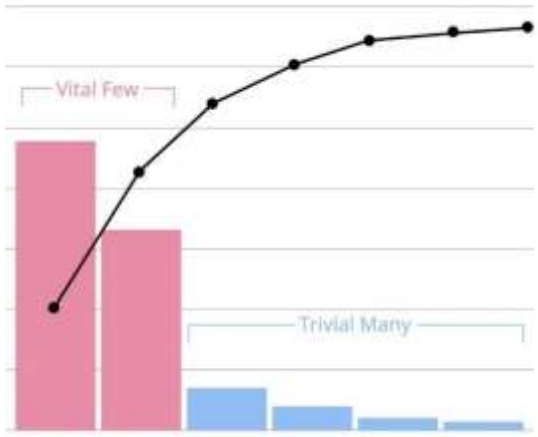


Fig. 3.1 : Depcition of Quality Areas as Vital Few

4. Root Cause Analysis (RCA)

RCA is a collective term for a wide range of approaches, tools, and techniques used to uncover causes of problems. It often combines the 5 Whys and Fishbone diagrams to ensure the problem does not recur.

3.3.3 Problem Solving Software Tools

Modern software projects rely heavily on automated tools to support quality assurance and problem-solving activities. These tools help manage complexity, improve accuracy, and save time. In software testing, test management tools such as test case repositories and execution tracking systems help ensure systematic testing.

Defect tracking tools are widely used to record, track, and analyze software defects throughout the SDLC. These tools provide visibility into

NOTE

defect trends, severity distribution, and resolution time, supporting data-driven decision-making.

Static analysis tools are used to identify code-level issues such as syntax errors, coding standard violations, and potential security vulnerabilities. Continuous integration and testing tools help detect defects early by automatically running tests whenever code changes occur.

In modern software development, problem solving is supported heavily by software tools. These tools help teams record problems, analyze data, track defects, manage corrective actions, and improve processes. Problem-solving tools are important because software projects generate a large amount of data, and manual handling becomes inefficient and error-prone. Tools ensure accuracy, traceability, collaboration, and systematic improvement.

One of the most important categories of problem-solving tools is defect tracking and issue management tools. These tools allow teams to record defects, assign them to developers, track their status, and verify their resolution. Defect tracking tools support the complete lifecycle of a defect, from identification to closure. They also store valuable historical data that can be analyzed to identify defect patterns and root causes. Popular examples include Jira, Bugzilla, Redmine, Mantis, and Azure DevOps. These tools provide features such as defect reporting, priority assignment, severity classification, workflow management, and reporting dashboards.

Issue management tools also support collaboration among team members. When a defect is reported, the tool ensures that it is visible to all relevant stakeholders. Developers can add comments, attach evidence, link related issues, and update progress. Testers can verify fixes and reopen issues if necessary. Project managers can monitor defect trends and release readiness. This improves communication and reduces misunderstandings.

Another important category is test management tools. These tools help organize test cases, test plans, test execution results, and defect links. Test management tools support systematic testing and provide traceability between requirements, test cases, and defects. They help teams identify which requirements have been tested and which remain untested. They also support reporting and metrics generation. Examples include TestRail, Zephyr, HP ALM, and qTest. These tools assist problem solving by providing structured testing data that can highlight problem areas.

Static analysis tools are also widely used for problem solving in software quality assurance. These tools analyze code without executing it and detect potential issues such as coding standard violations, complexity

NOTE

problems, security vulnerabilities, and code smells. Static analysis helps prevent defects early in development. Tools such as SonarQube, ESLint, PMD, Checkstyle, and Fortify help identify quality problems before they become failures. By using static analysis, organizations reduce debugging time and improve maintainability.

Another important tool category is debugging and profiling tools. Debuggers help developers identify the cause of runtime errors by inspecting program execution step-by-step. Profilers help identify performance bottlenecks by measuring CPU usage, memory usage, and execution time. These tools are critical for solving complex issues such as memory leaks, deadlocks, and slow performance. Examples include Visual Studio Debugger, Eclipse Debugger, GDB, JProfiler, and Chrome DevTools.

Configuration management and version control tools also support problem solving. Many software defects occur due to incorrect versions, missing updates, or configuration mismatches. Tools like Git, SVN, and Mercurial ensure that changes are tracked and controlled. They allow teams to identify which change introduced a defect, revert faulty code, and manage multiple releases. Version control tools also support collaboration and reduce conflicts.

Modern Quality Assurance relies on software tools to track issues, analyze code, and manage workflows.

1. Defect Tracking & Management Tools

These tools record the "Problem." They track a bug's lifecycle from "New" to "Fixed" to "Closed."

- Jira: The industry standard for agile project management and bug tracking. It allows teams to link bugs to user stories and track MTTR.
- Bugzilla: An open-source bug tracker originally developed by Mozilla.
- MantisBT: A simple, web-based bug tracking system.

2. Static Analysis Tools (Automated Problem Finding)

These tools "read" the code without running it to find potential problems (logic errors, security vulnerabilities, memory leaks) early.

- SonarQube: Checks code quality and security vulnerabilities; provides a "Technical Debt" metric.
- ESLint: Identifies problematic patterns in JavaScript code.

3. Visualization and Diagramming Tools

Used to create Fishbone diagrams, flowcharts, and process maps during brainstorming sessions.

NOTE

- Lucidchart / Miro: Collaborative whiteboards used for remote teams to perform Root Cause Analysis together.
- Microsoft Visio: A traditional tool for creating detailed process flow diagrams.

By using these software tools, organizations can improve consistency, traceability, and effectiveness in quality management and problem-solving.

Check Your Progress 3.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. Benchmarking in quality management means:
A) Writing more code in less time
B) Comparing performance with best practices or leading organizations
C) Testing software only at the end
D) Removing defects without analysis

_____

_____

Q2. Metrics are used only for measuring product quality, not process quality.

_____

Q3. A measurable value used to assess the performance of a process or product is called a _____.

_____

Q4. Write any two problem-solving techniques used in quality improvement.

Answer (Any two):

Brainstorming

Root Cause Analysis (RCA)

Pareto Analysis

Fishbone (Ishikawa) Diagram

5-Why Technique

Q5. Which of the following is a commonly used software tool for problem solving and quality analysis?
A) Text editor
B) Spreadsheet (Excel) for charts and analysis

NOTE

C) Video player
D) Music player

Fill in the Blanks

Benchmarking helps compare software performance with _____ standards.

The 80/20 rule is applied in _____ analysis.

True/False

Metrics are useful for monitoring software quality trends.

Problem-solving techniques focus only on fixing defects, not their causes.

_____

3.4 SUMMARY

This unit explained the concept of quality in different areas of software development and management. Quality is not limited to testing but must be ensured throughout requirements, design, coding, testing, deployment, and maintenance. Benchmarking and metrics were discussed as essential methods for measuring and improving quality. Benchmarking helps compare performance with standards or best practices, while metrics provide quantitative measures for product, process, and project evaluation. The unit also explained problem solving techniques such as root cause analysis, fishbone diagrams, Pareto analysis, brainstorming, and continuous improvement cycles. Finally, the unit described problem solving software tools such as defect tracking tools, test management tools, static analysis tools, debugging tools, version control systems, and reporting tools. Together, these concepts support continuous improvement and help organizations achieve reliable, efficient, and high-quality software products.

Quality in software engineering must be managed across different areas using structured and systematic approaches. Benchmarking and metrics help measure and compare quality performance, while problem-solving techniques assist in identifying and eliminating root causes of defects. Software tools further enhance quality management by automating testing, defect tracking, and analysis activities. Together, these approaches ensure continuous improvement in Software Quality Assurance and testing with following points :

- Benchmarking allows you to compare your quality performance against competitors or internal standards.

NOTE

- Metrics like Defect Density and MTTR turn subjective quality into objective numbers.
- Techniques like the Fishbone Diagram and Pareto Analysis help teams focus on the most critical causes of defects.
- Tools like Jira and SonarQube automate the tracking and detection of quality issues.

3.5 GLOSSARY

- **Benchmarking**: Comparing processes and performance with standards or best practices.
- **Metrics**: Quantitative measures used to assess quality and performance.
- **Root Cause Analysis**: Technique used to identify the underlying cause of defects.
- **Pareto Analysis**: Method used to identify major causes of problems.
- **Defect Tracking Tool**: Software used to record and manage defects.

3.6 ANSWERS TO CHECK YOUR PROGRESS

Q1. Answer: B

Explanation: Benchmarking is a technique of comparing processes, performance, and quality standards with industry leaders to improve quality.

Q2. Answer: False

Explanation: Metrics can be used for both product quality (defect density, reliability) and process quality (efficiency, productivity, cycle time).

Q3. Answer: Metric

Explanation: Metrics provide numerical data that help in analyzing and improving quality in different areas.

Q4. Explanation: These techniques help identify the real causes of defects or issues so that permanent corrective actions can be taken.

Q5. Answer: B

Explanation: Tools like spreadsheets, statistical tools, defect tracking systems, and quality analysis software help in quality measurement and problem solving.

Fill in the Blanks

Answer: industry

NOTE

Answer: Pareto

True/False

Answer: True

Answer: False

3.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3782	

3.8 REFERENCE

- Pressman, R. S., *Software Engineering: A Practitioner's Approach*
- Sommerville, I., *Software Engineering*
- IEEE Software Engineering Standards
- ISO/IEC 25010 Software Quality Model

3.9 EXERCISE

Part–1

A. Multiple Choice Questions (MCQs)

Q1. Quality in software is mainly measured in terms of:

A) Number of pages in documentation
B) Customer satisfaction and meeting requirements
C) Number of developers in the project
D) Software size only

Q2. Reliability in software quality refers to:

A) Attractive interface design
B) Ability to perform correctly for a specified time
C) High speed of typing
D) High cost of development

Q3. Usability is related to:

A) Ease of use and user satisfaction
B) Memory consumption only

NOTE

C) Coding standards
D) Number of test cases

Q4. Maintainability means:

A) Ability to increase software price
B) Ease of modifying and fixing the software
C) Ability to reduce software features
D) Ability to avoid documentation

Q5. Portability in software quality refers to:

A) Running software on different platforms
B) Making software heavier
C) Increasing software size
D) Reducing customer needs

Answers with Explanations:

1. Answer: B
Explanation: Software quality is judged by how well the software meets user needs, requirements, and expectations.

2. Answer: B
Explanation: Reliability means the software works without failure under specified conditions for a defined period.

3. Answer: A
Explanation: Usability ensures the software is easy to learn, operate, and understand for users.

4. Answer: B
Explanation: Maintainability ensures that the software can be corrected, improved, or adapted easily.

5. Answer: A
Explanation: Portability is the ability of software to be transferred and executed across different environments.

B. Small Answer Type Questions (5 Questions)

1. What is meant by software reliability?
2. Write any two quality attributes of software.
3. Define usability in software quality.
4. What is the meaning of portability?
5. Write any two differences between product quality and process quality.

Part–2

C. Long Answer Type Questions (200 words each)

NOTE

1. Explain the concept of quality in different areas of software development.
2. Discuss the importance of reliability, usability, and maintainability in software systems.
3. Explain the role of software quality attributes in determining overall product quality.
4. Describe how quality differs in various software domains such as web, mobile, and embedded systems.
5. Discuss how quality is measured and improved in different stages of SDLC.

Part–3

D. Long Answer Type Questions (300 words each)

1. Explain software quality attributes in detail. Discuss how each attribute contributes to overall software quality.
2. Discuss how quality requirements differ for different software applications such as banking, healthcare, e-commerce, and education systems.
3. Explain the challenges in achieving quality in different areas of software engineering and suggest solutions.
4. Discuss the relationship between software quality, user satisfaction, and business success.
5. Describe how software organizations ensure quality across different domains using standards, processes, and testing strategies.

UNIT-4
SOFTWARE PRODUCT QUALITY ASSESSMENT

UNIT STRUCTURE

4.1 Introduction of Software Quality
4.2 Objective
4.3 Constraints of Software Product Quality Assessment
  4.3.1 Customer is a King
  4.3.2 Quality and Productivity Relationship
  4.3.2 Requirements of a Product
  4.3.3 Organisation Culture
  4.3.4 Characteristics of Software
4.4 Summary
4.5 Glossary
5.6 Answer to check your progress
4.7 Suggested readings - OER
4.8 Reference
4.9 Exercise

4.1 INTRODUCTION OF SOFTWARE QUALITY

Software quality refers to the degree to which a software product satisfies stated and implied requirements. In today's technology-driven environment, software products are expected to be reliable, secure, efficient, user-friendly, and adaptable to change. Poor-quality software may lead to system failures, security risks, financial losses, and user dissatisfaction. Therefore, assessing software product quality is a critical activity in Software Quality Assurance (SQA).

Software product quality assessment involves evaluating software against predefined quality attributes such as functionality, reliability, usability, performance, maintainability, and portability. Unlike physical products, software is intangible, highly complex, and continuously evolving, which makes quality assessment challenging. Software testing, reviews, audits, and metrics are commonly used to assess and improve software quality.

This unit focuses on understanding the constraints and influencing factors involved in assessing software product quality, with emphasis

NOTE

on customer expectations, productivity, requirements, organizational culture, and software characteristics. Software Quality is not an absolute; it is a relative concept that depends on the specific context of the project. While ideally, every software team wants to deliver a "perfect" product, the reality of software engineering is that we operate within limits.

Software quality is one of the most important concerns in software engineering because software has become an essential part of almost every field such as education, banking, healthcare, transportation, defence, e-commerce, and communication. As software systems grow larger and more complex, the expectations from users and organisations also increase. Users demand software that is correct, reliable, secure, fast, easy to use, and able to work continuously without failure. Therefore, software quality is not only a technical requirement but also a business necessity.

Software quality can be defined as the degree to which a software product satisfies stated requirements, implied needs, and user expectations. It means the software must perform the functions it is intended for, and it must do so efficiently and safely. In modern software engineering, quality is not viewed as “absence of defects only.” A software system may have very few defects but may still fail in the market if it is slow, difficult to use, insecure, or not compatible with user environments. Hence, software quality includes functional as well as non-functional aspects.

Functional quality ensures that the software performs all required tasks correctly. For example, in a banking system, functional quality ensures that deposits, withdrawals, transfers, and account management features work correctly. Non-functional quality includes attributes such as performance, usability, reliability, security, maintainability, portability, scalability, and compatibility. For example, a banking application must not only process transactions correctly but must also protect user data, provide quick response, and remain available even under heavy load.

Software quality is closely linked with Software Quality Assurance (SQA) and software testing. SQA is a planned and systematic approach that ensures software processes and products meet quality standards. Testing is a key activity that detects defects, verifies requirements, and validates that the software works correctly in real conditions. However, software quality is not achieved only through testing at the end. Quality must be built into the product at every stage of development such as requirement analysis, design, coding, integration, testing, deployment, and maintenance.

NOTE

The importance of software quality can also be understood from the cost perspective. Fixing a defect after the software is released is far more expensive than fixing it during requirement or design stage. A defect that reaches customers can damage reputation, cause financial losses, create legal problems, and reduce user trust. Therefore, software quality is essential not only for customer satisfaction but also for business success.

In addition, software quality is not absolute. It is relative and depends on the expectations of users and the environment in which the software is used. For example, a simple mobile app may tolerate minor issues, but a medical system or aircraft control software cannot tolerate even small defects. Similarly, software quality is influenced by time and competition. A product that is considered high quality today may become outdated tomorrow if competitors introduce better features, faster performance, or stronger security.

Thus, software quality is a multi-dimensional concept that includes correctness, efficiency, usability, security, and maintainability. Ensuring software quality is the responsibility of the entire development team and the organisation. The next sections focus on product quality assessment and the constraints faced while assessing software quality.

4.2 OBJECTIVE

After studying this unit, you will be able to:

- **Define** software product quality and its major characteristics.
- **Explain** different software quality attributes and assessment criteria.
- **Identify** suitable measures for evaluating software quality.
- **Evaluate** a software product using appropriate quality characteristics.
- **Recommend** improvements based on software quality assessment results.

4.3 CONSTRAINTS OF SOFTWARE PRODUCT QUALITY ASSESSMENT

Software product quality assessment refers to evaluating the quality level of a software product using different techniques, standards, and metrics. It involves checking whether the product meets functional requirements, non-functional expectations, and quality standards. Quality assessment is essential because it helps determine whether the software is ready for release and whether it will satisfy customers.

NOTE

However, assessing software quality is not simple. It is affected by many constraints that make quality measurement and evaluation difficult.

Unlike physical products, software is intangible and does not wear out physically. Its behaviour depends on logic, environment, user inputs, and external systems. Quality assessment becomes more challenging because software quality includes many attributes such as reliability, performance, security, usability, and maintainability. Some of these attributes are measurable, while others are subjective. For example, performance can be measured using response time, but usability depends on user perception. Hence, software quality assessment faces practical limitations.

Another major constraint is that software quality depends heavily on requirements. If requirements are unclear or incomplete, quality assessment cannot be accurate. Quality assessment also depends on customer satisfaction, and customer expectations may change with time. In addition, organisational culture and productivity pressure can influence how quality is measured and prioritised. Many organisations focus on faster delivery, which may reduce quality assurance efforts. Similarly, the unique characteristics of software such as complexity, frequent changes, and dependency on environment also create constraints.

Assessing software product quality is a complex process influenced by several constraints. These constraints arise due to human factors, organizational practices, technological limitations, and the nature of software itself. Understanding these constraints helps quality assurance teams design effective testing and evaluation strategies.

Assessing and achieving software quality is often limited by several constraints. These are not necessarily "problems" but rather the "boundary conditions" within which a project must operate.

4.3.1 Customer is a King

One of the most significant constraints in software quality assessment is the role of the customer. In software development, customers are considered the most important stakeholders because they are the final users of the product and their satisfaction determines the success of the software. The phrase “Customer is a King” means that customer needs, expectations, and satisfaction are the top priority. While this is a positive principle, it also creates challenges in quality assessment.

The main difficulty is that customers may have different and sometimes unrealistic expectations about quality. Customers often expect software to be perfect, error-free, and available all the time. They may expect fast performance, attractive interface, high security, and continuous upgrades. However, achieving all these expectations within limited

NOTE

budget, time, and resources is difficult. Therefore, quality assessment becomes constrained by customer perception.

Another issue is that customer expectations are not always clearly expressed. Many customers cannot fully define their requirements at the beginning. They may describe their needs in general terms and expect the development team to interpret them correctly. This creates ambiguity. When requirements are unclear, it becomes difficult to assess whether the software meets customer expectations. Even if the product meets documented requirements, customers may still be dissatisfied because their implied expectations were not fulfilled.

Customers may also change their requirements frequently. In modern software development, change is common because market conditions, competition, and business strategies evolve. When requirements change, quality assessment becomes a moving target. The team must re-evaluate quality based on updated expectations. This increases the workload and may lead to confusion about what quality means for the product.

In software product quality assessment, the principle “Customer is King” emphasizes that customer satisfaction is the ultimate measure of quality. A software product may meet all technical specifications, but if it fails to satisfy user needs, it is considered low quality. Customer expectations are often subjective and may vary over time, making quality assessment challenging.

Customers may prioritize usability, performance, security, or reliability depending on the application domain. In software testing, customer requirements form the basis for acceptance testing and user validation. Frequent changes in customer expectations can lead to scope changes, increasing the difficulty of maintaining consistent quality.

The first and most significant constraint is the customer (or end-user).

- **Subjectivity:** "Quality" is often defined by the customer's perception. Even if the code is perfect (fact), if the customer finds the workflow confusing (perception), the quality is considered low.
- **Changing Needs:** Customers rarely know exactly what they want at the start. As they see the product evolve, their requirements change. This "Scope Creep" makes it difficult to assess quality against a fixed standard because the standard keeps moving.
- **Constraint:** You cannot assess quality based solely on technical specifications; you must assess it against *customer satisfaction*, which is variable and subjective.

Therefore, quality assessment must continuously align with customer feedback, usage patterns, and real-world requirements. This customer-

NOTE

centric approach ensures that software quality is evaluated from the user's perspective rather than only technical criteria.

4.3.2 Quality and Productivity Relationship

Another major constraint in software product quality assessment is the relationship between quality and productivity. Productivity in software development generally refers to how quickly software can be developed and delivered using available resources. It may be measured in terms of features delivered, lines of code, function points, or completed tasks per time. Quality, on the other hand, refers to correctness, reliability, and overall satisfaction.

In many organisations, there is constant pressure to increase productivity. Management often demands faster development, shorter release cycles, and quick delivery. This pressure is mainly due to market competition, customer demand, and business goals. However, increased productivity pressure often conflicts with quality goals. When teams are forced to deliver quickly, they may reduce testing time, skip reviews, or ignore documentation. This results in lower product quality.

The relationship between quality and productivity is complex. Some people assume that improving quality reduces productivity because it requires additional time for reviews, testing, and quality assurance activities. For example, performing code reviews, writing automated tests, and conducting security testing requires time. Therefore, teams may believe that focusing on quality will delay delivery.

However, in the long term, high quality actually improves productivity. This is because poor quality leads to frequent defects, rework, and maintenance problems. Fixing defects after release consumes significant time and resources. If software is released with many defects, the team must spend more time on bug fixing rather than developing new features. This reduces productivity and increases cost. Thus, quality and productivity are not always opposite; they can support each other when managed properly.

The constraint arises because organisations often focus on short-term productivity rather than long-term productivity. They may measure success by how quickly the product is delivered rather than how well it performs. This creates a situation where quality assessment is compromised. Quality metrics may be ignored, and release decisions may be based on deadlines rather than readiness.

Another issue is that productivity is easier to measure than quality. Managers can easily track tasks completed, code written, or deadlines met. But quality includes multiple attributes and requires deeper analysis. Therefore, productivity often dominates decision-making.

NOTE

Quality and productivity are closely related in software development, but they often appear to be conflicting goals. High productivity aims to deliver software quickly, while high quality requires careful design, testing, and review. In reality, improving quality often leads to improved productivity in the long run.

There is a common misconception that high quality reduces productivity (because testing takes time). However, the relationship is more complex.

- **The Short-Term View (Constraint):** In the short term, enforcing strict quality standards (rigorous reviews, heavy testing) *does* slow down productivity. It takes longer to write code if you are constantly checking it. This creates pressure on teams to "cut corners" to meet deadlines.
- **The Long-Term View (Reality):** In the long run, high quality *increases* productivity. Fixing a bug after release costs 100x more than fixing it during design.
- **The Dilemma:** Managers often focus on short-term delivery speed (Productivity) over long-term stability (Quality), acting as a major constraint on QA efforts.

Poor-quality software results in frequent defects, rework, and maintenance efforts, reducing overall productivity. Software quality assessment helps identify defects early, reducing costly rework and improving development efficiency. Practices such as automated testing, code reviews, and continuous integration support both quality and productivity. Thus, quality should not be viewed as an obstacle to productivity but as a contributor to sustainable and efficient software development.

4.3.3 Requirements of a Product

Requirements are the foundation of software development and quality assessment. A software product is considered high quality if it meets its requirements. Therefore, one of the biggest constraints in quality assessment is the nature of product requirements. If requirements are incomplete, inconsistent, unclear, or changing, then quality assessment becomes difficult and unreliable.

Requirements may be functional or non-functional. Functional requirements describe what the software should do. Non-functional requirements describe how the software should perform, such as performance, security, usability, and reliability. In many projects, functional requirements are documented more clearly, while non-functional requirements are often ignored or vaguely stated. For example, a requirement may say “the system should be fast” without

NOTE

defining response time limits. In such cases, quality assessment becomes subjective.

Another issue is that requirements may be conflicting. For example, a customer may want high security and also want very simple login access. These goals may conflict because stronger security often requires additional verification steps. Similarly, customers may want high performance and also want many complex features. Such conflicts make it difficult to define measurable quality goals.

Requirements may also change during development. This is common in software because customers discover new needs and market demands evolve. When requirements change, the quality criteria also change. The team must update test cases, metrics, and validation procedures. This creates additional effort and may delay delivery. It also makes it difficult to compare quality results over time.

Sometimes, requirements are not properly communicated to all team members. Miscommunication between customers, analysts, developers, and testers can lead to misunderstanding. This results in defects and quality issues. Quality assessment may show that software meets certain documented requirements, but it may fail to meet actual customer needs.

Another important point is that requirements must be testable. If a requirement cannot be tested, it cannot be verified. For example, a requirement such as “the system should be user-friendly” is difficult to measure without defining usability criteria. Therefore, quality assessment becomes constrained when requirements are not measurable.

Clear and well-defined requirements are the foundation of software product quality. Ambiguous, incomplete, or frequently changing requirements make quality assessment difficult. Software quality assurance relies on requirements to define test cases, acceptance criteria, and validation activities.

Quality is defined as "Conformance to Requirements." Therefore, if the requirements are flawed, the quality assessment will fail.

- **Incomplete Requirements:** If a requirement is missing, the developer won't build it, and the tester won't test it. The product will be "bug-free" but "useless."
- **Ambiguous Requirements:** If a requirement says "The system should be fast," it is impossible to assess. Does "fast" mean 1 second or 10 seconds?
- **Constraint:** The quality of the software can never exceed the quality of the requirements. "Garbage In, Garbage Out."

NOTE

If requirements are poorly documented, testers may misinterpret expected behavior, leading to ineffective testing. Requirement traceability ensures that all functional and non-functional requirements are covered during quality assessment. Changes in requirements must be carefully managed to maintain quality consistency.

Organisational culture plays a very important role in software quality assessment. Organisation culture refers to the values, beliefs, practices, and working environment that influence how employees perform their tasks. In software development, culture determines whether quality is treated as a priority or as an obstacle. Even if an organisation has quality standards and testing tools, poor culture can prevent effective quality assessment.

In some organisations, quality is seen as the responsibility of testers only. Developers may focus only on coding and assume testers will find defects. This mindset reduces overall quality because quality must be built into the product from the beginning. When quality is treated as a separate activity, defects increase and quality assessment becomes difficult.

Another cultural issue is blame culture. In some workplaces, defects are treated as failures of individuals, and employees may be punished for mistakes. This discourages honest reporting of issues. Developers and testers may hide defects or avoid discussing problems openly. This reduces transparency and makes quality assessment inaccurate. A healthy quality culture encourages learning and improvement rather than blame.

Management commitment is also part of organisational culture. If top management prioritizes deadlines and cost over quality, then quality assessment activities may be reduced. Testing time may be cut, reviews may be skipped, and quality metrics may be ignored. In such cases, quality assessment becomes superficial.

Organisational culture also affects training and skill development. Quality assessment requires skilled testers, analysts, and developers who understand testing methods, quality standards, and tools. If the organisation does not invest in training, the team may lack the ability to perform effective quality assessment. Poor skill levels lead to weak testing, incomplete reviews, and incorrect quality measurement.

Therefore, effective requirement management is essential for accurate and meaningful software quality assessment.

4.3.4 Organisation Culture

The unique characteristics of software itself create significant constraints in quality assessment. Unlike hardware products, software

NOTE

is intangible and behaves differently. Software does not wear out physically, but it can become outdated due to changes in environment, user needs, and technology. These characteristics make software quality assessment more challenging.

One important characteristic is complexity. Software systems can contain millions of lines of code and multiple interacting components. A small change in one module may affect other modules. Because of this complexity, it is difficult to assess quality completely. Testing every possible input and execution path is practically impossible. Therefore, quality assessment is always limited by time and resources.

Another characteristic is that software is highly changeable. Software products are frequently updated to add new features, fix defects, and adapt to new platforms. Each change introduces the risk of new defects. Therefore, quality assessment is not a one-time activity. It must be continuous. However, continuous assessment requires significant effort and strong testing processes. In many cases, due to deadlines, assessment may not be fully performed after every change.

Software also depends on its environment. A software product may behave differently on different hardware, operating systems, browsers, or network conditions. For example, a web application may work perfectly in one browser but fail in another. Similarly, performance may vary depending on server load and network speed. This environmental dependency makes quality assessment difficult because it requires testing in many different configurations.

Another characteristic is that software is invisible. Unlike physical products, software cannot be visually inspected for defects. Defects are hidden in logic and behaviour. Therefore, quality assessment depends on systematic testing and analysis. It is not possible to detect defects simply by looking at the software.

Software quality also involves non-functional aspects such as usability and security. These aspects are harder to measure compared to functional correctness. Security assessment requires specialized tools and expertise. Usability assessment requires user studies and feedback. Therefore, assessing these quality attributes becomes constrained.

Organizational culture significantly influences software product quality assessment. A quality-oriented culture promotes accountability, collaboration, and continuous improvement. In such organizations, quality is considered a shared responsibility among developers, testers, and managers.

The internal culture of the company is a massive "soft" constraint.

- **Blame Culture:** If developers are punished for bugs, they will hide them. This makes accurate quality assessment impossible.

NOTE

- **"Us vs. Them":** In some organizations, Developers and Testers are seen as enemies. Developers want to ship; Testers want to block the ship. This friction slows down quality processes.
- **Management Support:** If management views QA as just a "cost center" rather than a value adder, they will not allocate enough budget or time for proper testing tools and environments.

Conversely, organizations that prioritize speed over quality may compromise testing and quality assurance activities. Lack of management support, inadequate training, and poor communication negatively affect quality assessment efforts. A positive organizational culture encourages early testing, open defect reporting, and learning from past mistakes. This culture plays a vital role in overcoming quality assessment constraints and achieving high-quality software products.

Activity Title: Assessing the Quality of a Software Product

Objective:
To assess a software product using basic software quality characteristics.

Materials Required:

- Computer/mobile device
- Any software application
- Assessment checklist

Procedure/Steps:

1. Select a software application.
2. Evaluate its usability, reliability, functionality, efficiency, security, and maintainability.
3. Give each characteristic a rating from 1 to 5.
4. Identify the strongest and weakest quality areas.
5. Suggest improvements for the identified weaknesses.

Outcome:
The learner will be able to assess software product quality systematically and recommend appropriate improvements.

4.3.5 Characteristics of Software

The inherent characteristics of software pose unique challenges to quality assessment. Software is intangible, meaning it cannot be physically inspected like hardware. It is also highly complex, with numerous interdependencies and logical paths.

NOTE

Software is subject to frequent changes, which may introduce new defects while fixing existing ones. Unlike physical products, software does not wear out but may degrade due to changes and updates. Additionally, software behavior depends heavily on the environment in which it operates.

These characteristics make software quality assessment an ongoing and dynamic process that requires continuous testing, monitoring, and improvement.

Software has unique characteristics that distinguish it from hardware, making quality assessment uniquely difficult.

- Intangibility: You cannot touch or see software. You cannot inspect it for "cracks" like a physical bridge. You can only assess it by executing it.
- Complexity: Software systems often have millions of states. It is theoretically impossible to test every possible input combination (Exhaustive Testing is impossible).
- Software Doesn't "Wear Out": Hardware follows a "Bathtub curve" (fails early, stabilizes, then wears out). Software does not wear out; it deteriorates due to *change*. Every time you patch software, you risk introducing new bugs (Regression).
- Constraint: Because we cannot test everything (complexity) and we cannot see the product (intangibility), quality assessment is always based on *sampling* and *probability*, not absolute certainty.

Check Your Progress 4.1

Notes: a) Space is given below for your answer.
b) Check your answer with the one given at the end of this unit.

Q1. Software product quality assessment mainly helps in:
A) Increasing the size of software
B) Measuring and evaluating the quality of software product
C) Removing hardware problems
D) Reducing the number of users

_____

Q2. Customer satisfaction is an important constraint in software product quality assessment.

_____

Q3. The relationship between quality and productivity is generally _____ in nature because better quality reduces rework.

_____

NOTE

Q4. Write any two characteristics of software that affect quality assessment.

Answer (Any two):
Software is intangible (cannot be physically measured)
Software does not wear out like hardware
Software is complex and difficult to fully test
Changes in software are frequent and continuous

Q5. Which factor is MOST related to organization culture in software quality?

A) Team values and working practices
B) Screen resolution
C) Processor speed
D) Internet connection

4.4 SUMMARY

Software product quality assessment is a critical aspect of Software Quality Assurance. It involves evaluating software against customer expectations, requirements, and quality attributes while considering various constraints. Factors such as customer focus, productivity demands, requirement clarity, organizational culture, and software characteristics significantly influence quality assessment. Understanding these constraints helps organizations design effective quality assurance and testing strategies.

This unit introduced the concept of software quality and explained why software quality is essential for successful software products. Software quality includes functional correctness as well as non-functional attributes such as reliability, usability, security, and maintainability. The unit also discussed the major constraints of software product quality assessment. These constraints include customer expectations and satisfaction, the complex relationship between quality and productivity, the challenges of defining clear requirements, the influence of organisational culture, and the unique characteristics of software such as complexity and frequent changes. Understanding these constraints helps organisations plan better quality assurance strategies and improve the accuracy of software quality assessment.

4.5 GLOSSARY

- **Software Quality**: Degree to which software meets requirements and user expectations.

NOTE

- **Quality Assessment**: Evaluation of software against quality criteria.
- **Customer Satisfaction**: Measure of how well software meets user needs.
- **Productivity**: Efficiency in software development and delivery.
- **Organizational Culture**: Shared values and practices within an organization.

4.6 ANSWERS TO CHECK YOUR PROGRESS

Q1. Answer: B

Explanation: Quality assessment evaluates whether the software meets required standards, user expectations, and quality attributes.

Q2. Answer: True

Explanation: The customer is considered “king” because software quality is finally judged by customer satisfaction and usability.

Q3. Answer: Positive]

Explanation: High quality reduces defects, rework, and maintenance effort, which improves productivity.

Q4. Explanation: Software has unique properties such as complexity and frequent changes, making quality assessment more challenging.

Q5. Answer: A

Explanation: Organization culture influences quality through employee attitude, discipline, process follow-up, and commitment to standards.

Fill in the Blanks

Answer: quality

Answer: testing

True/False

Answer: False

Answer: True

NOTE

4.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3783	

4.8 REFERENCE

- Pressman, R. S., *Software Engineering: A Practitioner's Approach*
- Sommerville, I., *Software Engineering*
- ISO/IEC 25010: Systems and Software Quality Models
- IEEE Software Engineering Standards

4.9 EXERCISE

Part–1

A. Multiple Choice Questions (MCQs)

Q1. Software product quality assessment mainly refers to:

A) Increasing the software size
B) Evaluating software quality characteristics and performance
C) Reducing testing activities
D) Writing code without documentation

Q2. ISO/IEC 25010 standard is related to:

A) Database security only
B) Software product quality model
C) Network speed measurement
D) Hardware maintenance

Q3. Functional suitability in quality assessment means:

A) Software runs only on Windows
B) Software provides correct and complete functions
C) Software has more lines of code
D) Software has a complex interface

Q4. Performance efficiency mainly measures:

NOTE

A) Cost of development
B) Response time, throughput, and resource usage
C) Number of pages in SRS
D) Color of the GUI

Q5. Which of the following is NOT a quality characteristic in ISO/IEC 25010?

A) Security
B) Reliability
C) Maintainability
D) Advertising

Answers with explanation

1. Answer: B
Explanation: Product quality assessment evaluates whether software meets required quality characteristics such as reliability, usability, efficiency, etc.

2. Answer: B
Explanation: ISO/IEC 25010 provides a widely used model for evaluating software product quality and quality-in-use.

3. Answer: B
Explanation: Functional suitability checks whether the system provides accurate and complete functionality as required.

4. Answer: B
Explanation: Performance efficiency focuses on how effectively the system uses resources and how fast it performs tasks.

5. Answer: D
Explanation: Advertising is not a software quality characteristic; ISO 25010 includes security, reliability, maintainability, etc.

B. Small Answer Type Questions (5 Questions)

1. Define software product quality assessment.
2. Write any two quality characteristics given in ISO/IEC 25010.
3. What is meant by functional suitability?
4. What is the importance of reliability in product quality?
5. Write any two tools/techniques used for product quality assessment.

Part–2

C. Long Answer Type Questions (200 words each)

1. Explain the concept of software product quality assessment and its importance.

NOTE

2. Discuss ISO/IEC 25010 software product quality characteristics.
3. Explain how performance efficiency and usability are assessed in software products.
4. Describe different methods used to measure and evaluate software product quality.
5. Discuss the role of quality metrics in assessing software product quality.

Part–3

D. Long Answer Type Questions (300 words each)

1. Explain software product quality models in detail and describe how they are used for quality assessment.
2. Discuss ISO/IEC 25010 in detail and explain each product quality characteristic with examples.
3. Explain the process of evaluating software product quality using metrics and measurement techniques.
4. Discuss the challenges in assessing software quality in real software projects and suggest solutions.
5. Describe how software organizations ensure product quality through standards, audits, and quality assurance practices.

UNIT-5
SOFTWARE DEVELOPMENT PROCESS

UNIT STRUCTURE

5.1 Introduction
5.2 Objective
5.3 Software Development Process
- 5.3.1 Types of Products
- 5.3.2 Schemes of Criticality Definitions
- 5.3.3 Problematic Areas of Software Development Life Cycle
- 5.3.4 Software Quality Management
- 5.3.5 Why Software Has Defects? Processes Related to Software Quality
- 5.3.6 Quality Management System Structure
- 5.3.7 Pillars of Quality Management System
- 5.3.8 Important Aspects of Quality Management

5.4 Summary
5.5 Glossary
5.6 Answer to check your progress
5.7 Suggested readings - OER
5.8 Reference
5.9 Exercise

5.1 INTRODUCTION

The software development process refers to a structured set of activities required to develop a software product from initial idea to final deployment and maintenance. As software systems become larger, more complex, and more critical to business and society, the need for a disciplined and well-managed development process has become essential. A systematic software development process ensures that software is developed in a predictable, efficient, and high-quality manner.

Software development is not only about writing code and delivering an application. It is a complete process that begins with understanding user requirements and ends with maintaining the software after deployment. In every stage of software development, quality plays a

NOTE

critical role because even a small defect can lead to serious failures, financial losses, customer dissatisfaction, and damage to the organisation's reputation. Therefore, modern software engineering focuses not only on development speed but also on ensuring that the software is correct, reliable, secure, and maintainable.

The Software Development Process is a structured approach that defines how software is planned, designed, developed, tested, delivered, and maintained. This process is commonly represented through the Software Development Life Cycle (SDLC). SDLC provides a systematic method to manage complexity and reduce risk. It helps organisations deliver software in a controlled manner while ensuring that quality is maintained at every step. The process includes phases such as requirement analysis, design, implementation, testing, deployment, and maintenance. Each phase contributes to the final quality of the product.

In real-world projects, software is developed for different types of products such as commercial software, embedded systems, safety-critical systems, web applications, and mobile applications. Each type of product has different quality requirements and risk levels. For example, defects in a mobile game may be acceptable to some extent, but defects in aircraft control software can lead to disasters. Therefore, software quality assurance must consider product type and criticality.

Criticality refers to the level of importance of the software system and the impact of failure. Software can be categorized based on its criticality, such as low critical, business critical, and safety critical. The more critical the system, the more strict the quality standards must be. This is why criticality definitions and schemes are important in software quality management.

Even though organisations follow SDLC models and apply testing, software defects still occur frequently. This is because software is created by humans, and humans make mistakes. Defects can be introduced in requirements, design, coding, and even during testing. Some SDLC phases are more problematic than others, such as requirement gathering and design, because defects introduced early are difficult and expensive to fix later. Therefore, identifying problematic areas in SDLC is essential for improving quality.

To handle these challenges, organisations use Software Quality Management. Software Quality Management is a set of activities that ensure the software development process and the final product meet quality standards. It includes quality planning, quality assurance, quality control, continuous improvement, audits, reviews, and the use of metrics. Quality management is not limited to testing. It involves managing processes, people, tools, and organisational culture to build quality into the product.

NOTE

A Quality Management System (QMS) provides a structured framework for implementing quality management. It defines policies, procedures, responsibilities, standards, and documentation practices. A strong QMS ensures consistency and continuous improvement. It also helps organisations meet international standards such as ISO 9001, ISO/IEC 25010, and CMMI models.

Software Quality Assurance (SQA) and software testing are closely integrated with the software development process. Quality cannot be added at the end of development; it must be built into every phase of the Software Development Life Cycle (SDLC). Errors introduced during requirements analysis or design can propagate to later stages and become costly to fix. Therefore, understanding the development process is fundamental to managing software quality.

This unit explains the software development process with special emphasis on product types, criticality, problematic areas of SDLC, software quality management, defect causes, and quality management systems.

The Software Development Process is not just about writing code; it is a structured lifecycle aimed at producing high-quality software that meets customer needs. As software systems grow in complexity, managing their quality becomes a critical challenge. This unit explores how different types of products, criticality levels, and management systems interact to ensure the final delivery is reliable and effective. It bridges the gap between the *process* of making software and the *management* of its quality.

5.2 OBJECTIVE

After studying this unit, you will be able to:

- **Define** the software development process and its major stages.
- **Explain** the major activities involved in the Software Development Life Cycle (SDLC).
- **Identify** quality activities associated with different SDLC stages.
- **Differentiate** between various software development process activities.
- **Analyse** the importance of integrating quality assurance throughout the development process.
- Categorize different types of software products and their specific quality needs.
- Define criticality schemes to prioritize testing efforts.

NOTE

- Describe the structure and pillars of a robust Quality Management System (QMS).

5.3 SOFTWARE DEVELOPMENT PROCESS

The software development process consists of a series of structured phases such as requirements analysis, design, implementation, testing, deployment, and maintenance. Each phase produces specific deliverables and involves different stakeholders. A well-defined process ensures better planning, coordination, and quality control.

From a quality perspective, each phase of the SDLC must be carefully managed to prevent defects and ensure compliance with standards. Software testing and reviews are carried out throughout the development process to verify and validate software quality continuously.

The software development process is a disciplined and organised set of activities used to create a software product. It defines the steps required to transform an idea or requirement into a working system. A well-defined development process ensures that software is built systematically, risks are controlled, and quality is maintained throughout development.

The process typically includes requirement gathering, analysis, system design, implementation, testing, deployment, and maintenance. Different organisations may follow different SDLC models such as Waterfall, Spiral, Incremental, Agile, V-Model, and DevOps. Regardless of the model, the goal remains the same: to deliver software that meets customer needs while ensuring quality, reliability, and efficiency.

A strong development process provides clear responsibilities, documentation, communication, and checkpoints. It ensures that defects are detected early and that changes are managed properly. The development process also includes supporting activities such as configuration management, version control, reviews, audits, and process improvement.

Software quality assurance is deeply connected with the development process because quality is not something added at the end. It must be built into every phase. If the process is weak, defects will increase. If the process is disciplined and monitored, defects will reduce and software quality will improve.

The following subsections explain important concepts related to the software development process and software quality management.

Activity Title: Mapping Quality Activities to SDLC

NOTE

Objective:
To understand how quality activities are integrated into different stages of the Software Development Life Cycle (SDLC).

Materials Required:

- Chart paper/notebook
- Pen/pencil
- Reference material

Procedure/Steps:

1. Draw the major stages of the SDLC.
2. Identify quality activities associated with each stage.
3. Map activities such as requirement review, design review, code review, testing, and maintenance.
4. Explain how defects can be prevented at each stage.
5. Prepare a flowchart showing the relationship between SDLC stages and quality activities.

Outcome:
The learner will understand

5.3.1 Types of Products

Software products can be classified into different types based on their purpose, usage, and environment. Each type of product has unique quality requirements. Understanding product types is important because the approach to development, testing, and quality assurance changes according to the product category.

One common type is commercial software, which is developed for general users and sold in the market. Examples include office applications, media players, and accounting software. In such products, usability, performance, and compatibility are key quality attributes. Customers expect the product to be easy to use and reliable, and competition in the market is high.

Another type is custom software, which is developed for a specific organisation or client. Examples include university management systems, hospital management systems, and enterprise resource planning systems. Custom software quality is highly dependent on client requirements. Here, requirement clarity and customer communication play a major role. Quality is assessed based on how well the product meets the client's needs.

Embedded software is another important category. Embedded software runs inside hardware devices such as washing machines, cars, medical

NOTE

devices, and smart TVs. In embedded systems, reliability, efficiency, and safety are extremely important because software failure can damage the device or cause serious harm. Testing embedded software is also challenging because it depends on hardware interaction.

Software products can be classified into different types based on their purpose, complexity, and usage. Common categories include system software, application software, embedded software, web-based software, and safety-critical software.

System software such as operating systems requires high reliability and performance. Application software focuses on usability and functionality. Embedded and safety-critical software, such as software used in medical devices or aviation systems, requires extremely high quality due to the risk involved.

5.3.1 Types of Products

Not all software is created equal. The approach to quality assurance depends heavily on the type of product being developed.

1. **COTS (Commercial Off-The-Shelf):**
    - *Description:* Pre-packaged software sold to the mass market (e.g., MS Office, Antivirus).
    - *Quality Focus:* Usability and compatibility with diverse hardware. Since patches are expensive to distribute (reputation-wise), release quality must be high.
2. **Custom/Bespoke Software:**
    - *Description:* Software developed for a specific client to solve a specific problem (e.g., a banking system for a specific bank).
    - *Quality Focus:* Strict adherence to specific client requirements and business logic.
3. **Embedded Systems:**
    - *Description:* Software running inside hardware devices (e.g., washing machines, cars, pacemakers).
    - *Quality Focus:* Reliability and safety are paramount. Updates are difficult or impossible, so "zero defects" is the goal.
4. **Web/Mobile Applications:**
    - *Description:* Dynamic applications accessed via browsers or phones.
    - *Quality Focus:* Performance (load times), security, and cross-browser/device compatibility.

NOTE

The type of product determines the level of quality assurance, testing rigor, documentation, and compliance required. Understanding product types helps organizations apply appropriate quality management and testing strategies.

5.3.2 Schemes of Criticality Definitions

Criticality refers to how important a software system is and how severe the consequences of its failure can be. Criticality definitions are used to classify software products based on risk, impact, and required quality standards. Criticality schemes are important because they determine how much effort should be applied in development, testing, documentation, and certification.

One common scheme classifies software into low-critical, medium-critical, and high-critical categories. Low-critical software includes applications where failure causes minimal inconvenience, such as simple games, personal utilities, or small productivity tools. Medium-critical software includes business applications where failure may cause financial loss or operational disruption, such as inventory management or customer support systems. High-critical software includes safety-critical and mission-critical systems where failure may cause injury, death, or major disasters.

Another classification scheme divides software into safety-critical, mission-critical, and business-critical. Safety-critical software is the most strict category. Mission-critical software refers to systems essential for national security, military operations, or space missions. Business-critical software refers to systems essential for organisational operations such as banking, airline booking, and payment processing.

Criticality can also be defined based on failure impact. If failure results in human harm, it is safety-critical. If failure results in significant financial loss, it is business-critical. If failure results in loss of data, privacy breach, or legal issues, it is security-critical. If failure results in service disruption, it is availability-critical.

Criticality definitions also influence quality standards and testing depth. For low-critical systems, basic testing may be sufficient. For high-critical systems, extensive testing, formal verification, code reviews, static analysis, and certification may be required. Critical systems often require compliance with standards such as DO-178C for avionics, IEC 62304 for medical software, and ISO 26262 for automotive systems.

Software criticality refers to the level of risk associated with software failure. Criticality schemes classify software based on the consequences of failure, such as financial loss, system downtime, or risk to human life.

NOTE

Low-criticality software may tolerate minor defects, while high-criticality software demands rigorous testing, formal verification, and strict quality controls. Safety-critical systems require compliance with standards and extensive validation.

To manage quality effectively, we must prioritize. We cannot test everything with the same intensity. We use "Criticality Schemes" to classify features or modules.

- **Safety-Critical:** Failure could result in loss of life or physical injury (e.g., Flight control software, medical devices). *Requires:* 100% test coverage, formal verification.
- **Mission-Critical:** Failure causes the failure of the business activity (e.g., Stock exchange transaction system). *Requires:* High redundancy and rigorous testing.
- **Business-Critical:** Failure reduces productivity or customer trust but doesn't stop operations (e.g., Internal HR portal).
- **Non-Critical:** Failure is an inconvenience (e.g., A glitch in the 'About Us' page animation).

Criticality definitions guide decisions related to test coverage, documentation, review intensity, and release criteria. Higher criticality demands stronger SQA and testing processes.

5.3.3 Problematic Areas of Software Development Life Cycle

Even though SDLC provides a structured approach, some phases are more prone to defects than others. These phases are called problematic areas because mistakes introduced here often become costly and difficult to correct later. Understanding problematic areas helps improve quality and reduce rework.

Requirement analysis is considered one of the most problematic phases. This is because requirements are often incomplete, ambiguous, inconsistent, or changing. Customers may not clearly express what they want. Analysts may misunderstand customer needs. Requirements may be written in vague language. When such issues occur, the development team may build incorrect features, and defects become embedded from the start. Fixing requirement defects later is expensive because it affects design, code, and tests.

Design is another problematic area. Poor design decisions can lead to performance issues, scalability problems, security vulnerabilities, and maintainability difficulties. If architecture is weak, the software may become difficult to modify and test. Design defects often remain hidden until later stages, and fixing them may require major rework.

Implementation or coding is also a major source of defects. Developers may introduce logic errors, boundary condition issues, memory leaks,

NOTE

concurrency problems, or incorrect handling of exceptions. Coding defects can be reduced through coding standards, code reviews, and static analysis tools, but they cannot be eliminated completely.

Integration is another problematic stage because different modules are combined. Even if each module works correctly alone, integration can introduce issues due to interface mismatches, data inconsistencies, and dependency conflicts. Integration defects are common in large systems and require careful testing.

Testing is also a challenging area. Many organisations treat testing as the final stage and allocate limited time. This leads to incomplete testing, poor test coverage, and undetected defects. Another issue is that testers may not have enough resources, tools, or knowledge of requirements. Therefore, testing may fail to detect critical defects. Certain phases of the SDLC are more prone to quality issues. Requirements analysis is often problematic due to unclear, incomplete, or changing requirements. Poor requirements lead to incorrect design and ineffective testing.

Design issues arise when architectural decisions are poorly documented or misunderstood. During implementation, coding errors, lack of standards, and insufficient reviews introduce defects. Testing may become problematic if insufficient time or resources are allocated.

Defects can be introduced at any stage, but some areas are more prone to errors than others.

1. **Requirements Phase:** The most problematic area. Ambiguous, incomplete, or changing requirements lead to "building the wrong product correctly."
2. **Design Phase:** Architectural flaws (e.g., poor scalability) are hard to fix later.
3. **Coding Phase:** Logic errors, syntax errors, and lack of commenting.
4. **Integration Phase:** Individual modules work fine, but fail when connected (interface errors).
5. **Maintenance Phase:** "Fixing" one bug often introduces two new ones (Regression).

Maintenance is another critical area, as frequent changes may introduce new defects. Identifying and managing these problematic areas helps improve overall software quality.

5.3.4 Software Quality Management

Software Quality Management refers to the systematic activities used to ensure that software meets quality requirements. It includes quality

NOTE

planning, quality assurance, quality control, and continuous improvement.

Quality planning defines standards, processes, and metrics. Quality assurance focuses on process compliance and prevention of defects. Quality control involves testing, inspections, and reviews to detect defects. Software Quality Management is the set of activities that ensures software processes and products meet quality standards. It focuses on managing quality systematically rather than treating it as a separate or final step. Software quality management includes planning, assurance, control, and improvement activities.

Quality planning involves defining quality goals, standards, procedures, responsibilities, and resources. It ensures that quality requirements are clear and measurable. Quality planning also defines what metrics will be used and what testing methods will be applied.

Quality assurance focuses on ensuring that processes are followed correctly. It involves audits, reviews, process evaluations, and adherence to standards. The goal is defect prevention. Quality assurance ensures that the team follows proper methods during requirement analysis, design, coding, and testing.

Quality control focuses on defect detection. It includes testing, inspections, verification, validation, and defect tracking. Quality control ensures that the product meets quality standards before release.

Continuous improvement is also an important part of quality management. Organisations analyse defects, identify root causes, and improve processes to prevent similar issues in the future. Techniques such as PDCA cycle, Six Sigma, and CMMI maturity models support continuous improvement.

Software quality management is essential because it ensures consistent quality, reduces rework, increases customer satisfaction, and improves long-term productivity.

SQM is the umbrella term for all activities that ensure the software meets its quality goals. It is generally divided into three layers:

1. **Quality Assurance (QA):** Process-oriented. Setting up the right processes to prevent bugs.
2. **Quality Control (QC):** Product-oriented. Testing the product to find bugs.
3. **Quality Improvement (QI):** Using data from QA and QC to improve the process for the next project.

Effective software quality management integrates quality activities into the SDLC, ensuring consistency, traceability, and continuous improvement.

NOTE

5.3.5 Why Software Has Defects? Processes Related to Software Quality

Software defects occur because software development is a human activity and involves complex logic. Defects can be introduced in any stage of SDLC. Understanding why software has defects helps organisations improve processes and reduce defect rates. One major reason is unclear requirements. If requirements are ambiguous, developers may implement incorrect functionality. Another reason is frequent changes. Software requirements and environments change continuously, and each change can introduce new defects. Human errors in design and coding also cause defects. Developers may misunderstand logic, make incorrect assumptions, or overlook boundary cases. Complexity also increases defects because large systems have many interactions and dependencies.

Time pressure is another major reason. When deadlines are strict, teams may skip reviews, reduce testing, and compromise quality. Lack of skills and training can also increase defects. If developers and testers are not trained in best practices, they may introduce or miss defects.

Poor communication between stakeholders also contributes to defects. Misunderstanding between customers, analysts, developers, and testers results in incorrect development and weak testing.

Processes related to software quality include requirement reviews, design reviews, code inspections, test planning, test execution, defect tracking, configuration management, and audits. These processes aim to prevent defects, detect defects early, and ensure that quality standards are maintained.

Software defects arise due to human errors, complex requirements, inadequate processes, and environmental changes. Common causes include misunderstanding requirements, design flaws, coding mistakes, and insufficient testing.

Defects are inevitable because software is built by humans. The primary causes include:

- **Miscommunication:** Between the client and the analyst, or the analyst and the developer.
- **Complexity:** Modern systems have too many moving parts for one person to understand fully.
- **Programming Errors:** Simple typos or logic mistakes.
- **Changing Requirements:** Code designed for "Function A" is hacked to perform "Function B" halfway through the project.
- **Time Pressure:** Rushing to meet a deadline often leads to skipping code reviews or testing.

NOTE

- **Poor Documentation:** Developers guessing how a legacy module works.

Defects are also introduced due to time pressure, lack of skills, poor communication, and inadequate tools. Weak quality processes increase the likelihood of defect introduction. Processes related to software quality such as reviews, audits, testing, configuration management, and change management help detect and prevent defects. Strong processes reduce defect rates and improve reliability.

5.3.6 Quality Management System Structure

A Quality Management System (QMS) provides a structured framework for managing quality within an organization. It defines policies, procedures, roles, responsibilities, and documentation required to achieve quality objectives.

A Quality Management System (QMS) is a structured framework used to manage and improve quality. In software development, a QMS defines how quality policies are implemented and how quality activities are performed. It provides guidelines, procedures, documentation, and responsibilities.

The structure of a QMS usually includes quality policy, quality objectives, standard operating procedures, process documentation, work instructions, templates, and records. It defines roles and responsibilities such as quality manager, project manager, developers, testers, and auditors.

A QMS also includes mechanisms for monitoring and measurement. This includes metrics, audits, reviews, and reporting systems. It ensures that processes are followed consistently and that improvements are made based on evidence. The QMS structure supports standardization and consistency across projects. It ensures that every project follows defined practices rather than relying on individual approaches. This improves predictability and reduces quality variation.

In software development, a QMS includes process definitions, standards, templates, review mechanisms, and measurement systems. It ensures consistency across projects and supports compliance with quality standards such as ISO or CMMI.

A QMS is the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management. A standard QMS (like ISO 9001) usually includes:

- **Quality Manual:** The high-level document stating the company's policy.
- **Procedures:** documented ways of performing tasks (e.g., "How to perform a Code Review").

NOTE

- **Work Instructions:** Step-by-step guides for specific tools.
- **Records:** Evidence that the procedures were followed (e.g., Test Reports, Audit Logs)

A well-structured QMS enables organizations to control quality activities and improve software development processes continuously.

5.3.7 Pillars of Quality Management System

The Quality Management System is supported by several pillars. These include leadership commitment, defined processes, trained personnel, measurement and analysis, and continuous improvement. A Quality Management System is built on certain pillars that ensure its effectiveness. These pillars represent the key elements that support quality in an organisation.

The first pillar is management commitment. Without strong management support, quality initiatives fail. Management must allocate resources, provide training, and enforce quality standards.

The second pillar is process definition and standardization. A QMS must define clear processes for development, testing, reviews, audits, and maintenance. Standard processes ensure consistency.

The third pillar is documentation and traceability. Quality requires proper documentation of requirements, design, code changes, test cases, defects, and decisions. Traceability ensures that every requirement is verified and validated.

The fourth pillar is continuous improvement. A QMS must not be static. It must evolve based on feedback, defect analysis, audits, and lessons learned. Continuous improvement ensures that the organisation becomes more mature over time. The fifth pillar is customer focus. Quality must be aligned with customer needs. A QMS must ensure that customer requirements are understood and satisfied. Thus, the pillars of QMS provide stability, consistency, and improvement capability for software quality management.

Leadership provides direction and resources. Processes ensure consistency and repeatability. Skilled personnel execute processes effectively. Measurement helps monitor performance. Continuous improvement ensures long-term quality enhancement. Together, these pillars form a strong foundation for effective software quality management.

A successful QMS rests on three major pillars:

1. **Process:** The defined steps we follow. A good process is repeatable and measurable. If the process is good, the product should be good.

NOTE

2. **People:** The skills, training, and motivation of the team. Even the best process fails if the people don't understand it or refuse to follow it.
3. **Technology:** The tools used to support the people and process (e.g., Automation tools, Bug trackers, IDEs).

5.3.8 Important Aspects of Quality Management

Quality management in software involves several important aspects that must be considered for effective implementation. One important aspect is quality planning, which ensures that quality goals, standards, and metrics are defined clearly. Another aspect is quality assurance, which focuses on process compliance and defect prevention. Quality control is also important because it focuses on detecting defects through testing and inspections.

Measurement and metrics are essential aspects of quality management. Without measurement, improvement cannot be tracked. Defect metrics, test coverage metrics, reliability metrics, and performance metrics help evaluate quality objectively.

Training and skill development is another key aspect. Quality depends on people, and skilled professionals produce better software. Organisations must invest in training developers and testers.

Tool support is also important. Tools such as defect tracking systems, test automation tools, static analysis tools, and continuous integration tools improve efficiency and accuracy.

Risk management is another important aspect. Critical systems require strict quality assurance, and risks must be identified and controlled early. Important aspects of software quality management include customer focus, risk management, documentation, compliance with standards, and continuous improvement. Customer requirements guide quality objectives and acceptance criteria.

To make SQM effective, organizations must focus on:

- **Management Commitment:** Top management must support quality initiatives with budget and authority.
- **Vendor Management:** Ensuring third-party libraries or outsourced teams also meet quality standards.
- **Risk Management:** Proactively identifying what could go wrong (e.g., "What if the server crashes?") and planning for it.
- **Metric Analysis:** regularly reviewing data (Defect Density, Customer Satisfaction) to drive decisions.

NOTE

Risk management helps identify and mitigate potential quality issues early. Documentation ensures traceability and knowledge sharing. Compliance ensures adherence to industry standards and regulations.

Continuous improvement ensures that lessons learned are applied to future projects, enhancing quality maturity.

Check Your Progress 5.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. Software Development Process mainly refers to:
A) Only coding activity
B) A structured set of activities to develop and maintain software
C) Only software installation
D) Only system testing

_____

Q2. Software defects are caused only due to programming (coding) errors.

_____

Q3. A structured system that defines processes, procedures, and responsibilities for achieving quality is called a _____.

_____

Q4. Write any two problematic areas of the Software Development Life Cycle (SDLC).

Answer (Any two):

Incomplete or unclear requirements

Frequent requirement changes

Poor design decisions

Lack of proper testing

Weak documentation and communication

_____

Q5. Which of the following is considered a pillar of Quality Management System?
A) Ignoring standards
B) Continuous improvement
C) Increasing complexity
D) Avoiding documentation

_____

NOTE

Fill in the Blanks

Quality must be built into every phase of the _____.

High-criticality software requires _____ testing.

True/False

Software defects are unavoidable but can be minimized.

Quality management is only required during testing.

5.4 SUMMARY

The software development process plays a vital role in achieving high-quality software products. Understanding product types, criticality, problematic SDLC areas, and defect causes helps improve quality planning and control. Software quality management and quality management systems provide a structured approach to ensuring consistent and reliable software quality.

This unit explained the software development process and its connection with software quality management. It discussed the types of software products and how quality requirements vary across different product categories. It also explained schemes of criticality definitions, which classify software systems based on failure impact and risk. The unit highlighted problematic areas of SDLC where defects are commonly introduced, such as requirements, design, integration, and testing. It discussed software quality management and explained why software defects occur. The structure of a Quality Management System and its pillars were described, along with important aspects of quality management such as planning, assurance, control, metrics, tools, training, and continuous improvement. Overall, the unit emphasized that software quality must be built into the development process through systematic management and disciplined practices.

5.5 GLOSSARY

- **SDLC**: Software Development Life Cycle
- **Criticality**: Level of risk associated with software failure
- **QMS**: Quality Management System
- **SQA**: Software Quality Assurance
- **Defect**: An error in software that causes incorrect behavior

NOTE

5.6 ANSWERS TO CHECK YOUR PROGRESS

Q1. Answer: B

Explanation: The software development process includes requirement analysis, design, coding, testing, deployment, and maintenance.

Q2. Answer: False

Explanation: Defects can occur at any stage such as requirement misunderstanding, poor design, incorrect coding, or improper testing.

Q3. Answer: Quality Management System (QMS)

Explanation: QMS ensures quality through defined standards, policies, documentation, audits, and continuous improvement.

Q4. Explanation: These areas commonly create defects and delays, reducing overall software quality.

Q5. Answer: B

Explanation: Continuous improvement is a key pillar of quality management, ensuring processes and product quality improve over time.

5.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3784	

5.8 REFERENCE

- Pressman, R. S., *Software Engineering: A Practitioner's Approach*
- Sommerville, I., *Software Engineering*
- ISO 9001 Quality Management Systems
- IEEE Software Engineering Standards

NOTE

5.9 EXERCISE

Part–1

A. Multiple Choice Questions (MCQs)

Q1. SDLC stands for:

A) Software Design Level Control
B) System Development Life Cycle
C) Software Development Life Cycle
D) Software Documentation Level Cycle

Q2. Which SDLC model is best suited when requirements are clearly defined and stable?

A) Spiral model
B) Waterfall model
C) Agile model
D) RAD model

Q3. The main advantage of the Agile model is:

A) No customer involvement
B) Continuous delivery with frequent feedback
C) Works only for small projects
D) No documentation is needed

.

Q4. In which phase of SDLC are system requirements gathered?

A) Coding
B) Maintenance
C) Requirement analysis
D) Testing

Q5. The V-Model is mainly based on:

A) Parallel development and testing
B) Verification and Validation activities
C) Only coding and debugging
D) Hardware development process

. **Answers with explanation**

1. Answer: C
Explanation: SDLC is a structured process used for developing software systematically from planning to maintenance.

2. Answer: B
Explanation: The Waterfall model works well when requirements are fixed and clearly understood from the beginning.

NOTE

3. Answer: B

Explanation: Agile promotes iterative development and frequent customer feedback for continuous improvement

4. Answer: C

Explanation: Requirement analysis is the first major phase where user needs and system requirements are collected and documented.

5. Answer: B

Explanation: The V-Model emphasizes verification (reviews) and validation (testing) at each stage of development

B. Small Answer Type Questions (5 Questions)

1. Define SDLC and mention its importance.
2. Write any two phases of SDLC.
3. What is the difference between Waterfall and Agile model?
4. What is the role of design phase in SDLC?
5. Write any two advantages of using a software development process.

Part–2

C. Long Answer Type Questions (200 words each)

1. Explain the phases of Software Development Life Cycle (SDLC).
2. Discuss Waterfall model with its advantages and disadvantages.
3. Explain the Agile model and its key principles.
4. Describe Spiral model and explain why it is considered risk-driven.
5. Explain the V-Model and its importance in verification and validation.

Part–3

D. Long Answer Type Questions (300 words each)

1. Explain SDLC in detail and discuss how each phase contributes to software quality.
2. Compare Waterfall, Spiral, V-Model, and Agile models with suitable examples.
3. Discuss how risk management is handled in Spiral model and why it is useful for large projects.
4. Explain Agile development process and discuss its role in improving software quality and customer satisfaction.

NOTE

Rabindranath TAGORE UNIVERSITY

Approved by : AICTE, NCTE, BCI, INC, MP PARAMEDICAL COUNCIL, DEB (UGC) | Recognized by : UGC | Member of : AIU, ACU

RABINDRANATH TAGORE UNIVERSITY (M.P.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org / for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

_____

_____

_____

***

BLOCK 2	FUNDAMENTALS OF TESTING	89-170
Unit 6	Software Testing	91
Unit 7	Testing Process	111
Unit 8	Mutation Testing	129
Unit 9	Categories of Defect in Software Testing	143
Unit 10	Management Awareness for Testing	157



Block II introduces the fundamental concepts, processes, techniques, and management aspects of software testing. It develops an understanding of how testing helps identify defects, improve software quality, and ensure that software systems meet specified requirements. The block begins with the basic concepts, objectives, principles, and importance of Software Testing, followed by a detailed study of the Testing Process, including test planning, test design, execution, defect reporting, and test closure. Mutation Testing is discussed as an advanced technique for evaluating the effectiveness of test cases. The block also examines different Categories of Defects in Software Testing, their causes, classification, and impact on software quality. Finally, Management Awareness for Testing focuses on test management, resource planning, risk management, documentation, team coordination, and quality-related responsibilities. Overall, this block provides learners with a systematic foundation for understanding and applying software testing practices in real-world software development.

UNIT-6
SOFTWARE TESTING

UNIT STRUCTURE

6.1 Introduction
6.2 Objective
6.3 Software Testing
- 6.3.1 Necessity of testing
- 6.3.2 What is testing?
- 6.3.3 Fundamental test process
- 6.3.4 The psychology of testing
- 6.3.5 Historical Perspective of Testing
- 6.3.6 Definitions of Testing
- 6.3.7 Approaches to Testing
- 6.3.8 Testing During Development Life Cycle
- 6.3.9 Requirement Traceability Matrix
- 6.3.10 Essentials of Software Testing, Workbench

6.4 Summary
6.5 Glossary
6.6 Answer to check your progress
6.7 Suggested readings - OER
6.8 Reference
6.9 Exercise

6.1 INTRODUCTION

Software testing is a fundamental activity within Software Quality Assurance that ensures a software system functions correctly, reliably, and efficiently according to specified requirements. As software systems increasingly support critical applications such as banking, healthcare, aviation, education, and governance, even small defects can lead to serious consequences. Hence, testing has evolved from a simple debugging activity into a disciplined engineering process.

Testing provides confidence that the software behaves as expected under normal and exceptional conditions. It helps identify functional errors, performance bottlenecks, security vulnerabilities, and usability issues before the software reaches end users. In modern development practices, testing is no longer a final-phase activity but is integrated throughout the Software Development Life Cycle (SDLC). Continuous

NOTE

testing ensures early defect detection, reduces cost of rework, and improves overall software quality.

Software testing is a critical phase in the software development life cycle. It acts as the final gatekeeper before a product reaches the customer. While often mistaken as merely "bug hunting," testing is a systematic discipline aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. In this unit, we explore the fundamental concepts, processes, and psychological aspects that underpin successful software testing.

This unit presents a comprehensive understanding of software testing fundamentals, including its necessity, processes, human aspects, evolution, approaches, integration with SDLC, and essential supporting artifacts.

Software testing is one of the most essential activities in software engineering because it directly influences the reliability, correctness, and acceptance of a software product. In the modern world, software systems are used in almost every field such as banking, healthcare, education, transportation, defence, business, and communication. A failure in software may cause financial loss, customer dissatisfaction, data leakage, business disruption, or even threats to human life in safety-critical systems. Therefore, testing is not optional; it is a necessary process for building trustworthy and high-quality software.

Testing is the process of evaluating a software system to find defects and to ensure that it meets the specified requirements. It verifies whether the software behaves as expected under different conditions. Testing also validates that the product is suitable for user needs and real-world use. Software testing is not only about finding bugs; it is also about reducing risk, improving confidence, and ensuring that software performs correctly and efficiently.

Software development is a complex process that involves multiple stages such as requirements analysis, design, coding, integration, and maintenance. Errors can occur at any stage. If these errors are not detected early, they become expensive and difficult to fix later. Testing helps detect these errors before the product is delivered. It ensures that defects are identified and corrected in time. Testing also ensures that changes in software do not introduce new defects. This is why regression testing is important in maintenance and continuous delivery.

In addition, software quality is not limited to functional correctness. Quality also includes non-functional attributes such as performance, usability, security, reliability, maintainability, and compatibility. Testing must evaluate all these aspects. For example, performance testing checks response time and load capacity, security testing checks vulnerability and data protection, and usability testing checks ease of

NOTE

use. Therefore, testing is a broad discipline with many types, methods, and techniques.

Testing also plays an important role in customer satisfaction. Customers expect software to work correctly, respond quickly, and provide a smooth user experience. Even small defects can damage customer trust. In competitive markets, quality becomes a major factor for success. A product with poor quality may fail even if it has good features. Hence, software testing supports business goals as well as technical goals.

6.2 OBJECTIVE

After studying this unit, you should be able to:

- Analyse the necessity and core definitions of software testing.
- Describe the fundamental test process from planning to closure.
- Analyze the psychological differences between testing and developing.
- Trace the history of testing from debugging to quality assurance.
- Explain the V-Model and how testing aligns with development phases.
- Create and understand the importance of a Requirement Traceability Matrix (RTM).

6.3 SOFTWARE TESTING

Software testing is a systematic process of evaluating software by executing it under specified conditions to detect defects and verify compliance with requirements. It involves observing the behavior of software, comparing actual outcomes with expected results, and identifying deviations. Testing ensures both verification (building the product correctly) and validation (building the correct product).

Testing does not guarantee the absence of defects but significantly reduces the risk of failure. It provides stakeholders with information about software quality, readiness for release, and potential risks.

Software testing is a systematic process used to evaluate a software product to ensure that it satisfies requirements and works correctly. It is a critical part of Software Quality Assurance because it provides confidence that the software is reliable and ready for use. Testing is performed to identify defects, measure quality, and reduce the risk of software failure.

NOTE

Testing is required because software systems are complex and developed by humans. Humans make mistakes, and these mistakes become defects. Defects may appear in requirements, design, code, or interfaces. Testing helps detect these defects before the product is delivered. Testing also ensures that the software works correctly under normal and abnormal conditions. It checks both expected behaviour and unexpected scenarios.

Software testing involves different levels such as unit testing, integration testing, system testing, and acceptance testing. It also includes different types such as functional testing, performance testing, security testing, usability testing, and regression testing. Testing can be manual or automated. Manual testing is performed by human testers, while automated testing uses tools and scripts. Both are important and used depending on project needs.

Testing is also connected with verification and validation. Verification ensures that the product is built correctly according to specifications. Validation ensures that the correct product is built according to customer needs. Testing supports both verification and validation. For example, unit testing verifies code logic, and acceptance testing validates user requirements.

In modern software development, testing is not only a quality activity but also a risk management activity. It helps organizations avoid costly failures, improve customer satisfaction, and ensure business success. The following subsections explain important aspects of software testing in detail.

6.3.1 Necessity of Testing

Testing is necessary because software failures can cause serious consequences. The main purpose of testing is to detect defects and ensure that software meets requirements. Without testing, software may contain hidden defects that appear after release. Such defects can cause system crashes, data loss, security breaches, and customer dissatisfaction.

One major reason testing is necessary is that software is developed by humans and humans make mistakes. Even experienced developers can write incorrect code, misunderstand requirements, or overlook special conditions. Testing provides an independent way to detect these mistakes.

Testing is also necessary because software systems are complex. A system may have many modules, interfaces, and external dependencies. Even if each module works correctly, integration issues may occur when modules interact. Testing ensures that the complete system works correctly.

NOTE

Another reason is that requirements may not be fully correct or complete. Customers may change their needs during development. Testing helps confirm whether implemented features match updated requirements. It also helps identify missing requirements and inconsistencies.

Testing is necessary for ensuring reliability. Many software systems must run continuously without failure, such as banking systems and telecom networks. Testing checks system stability under long execution and heavy load. Performance testing ensures that software works efficiently.

Security is another important reason. Software is often attacked by hackers and malicious users. Security testing helps identify vulnerabilities such as SQL injection, authentication flaws, and data leakage. Without security testing, software may become unsafe for users.

Testing is necessary because software systems are developed by humans, and human errors are inevitable. Mistakes can occur during requirement analysis, design, coding, configuration, or integration. These mistakes manifest as defects, which may cause software to behave incorrectly or fail completely.

Software failures can be expensive, and sometimes dangerous. Why do we test?

1. **Ensuring Quality:** Testing verifies that the software meets the specified requirements.
2. **Risk Reduction:** It reduces the risk of software failure in operation.
3. **Cost Effectiveness:** Finding defects early (in requirements or design) is significantly cheaper than fixing them in production.
4. **Legal & Regulatory Compliance:** In industries like healthcare or finance, testing is required by law to audit trails and ensure data integrity.

Testing helps detect these defects early, reducing the cost and effort required to fix them. Defects found after deployment are much more expensive to correct and may damage customer trust. Testing also ensures compliance with legal, regulatory, and safety requirements.

Furthermore, testing improves software reliability, performance, security, and usability. Without testing, software failures can result in data loss, financial damage, legal penalties, and safety hazards. Therefore, testing is an essential activity for risk reduction and quality assurance.

NOTE

6.3.2 What is Testing?

Software testing is the process of executing or evaluating a software system with the intention of finding defects and verifying that the system meets requirements. It involves checking software behaviour under different conditions and comparing actual results with expected results. Testing can be performed manually or automatically.

Testing is not only about executing test cases. It also includes activities such as planning tests, designing test cases, preparing test data, setting up test environments, executing tests, recording results, reporting defects, and verifying fixes. Testing is a complete process that supports quality assurance.

Testing focuses on both functional and non-functional aspects. Functional testing checks whether software performs required functions correctly. For example, in an e-commerce system, functional testing checks whether users can register, login, search products, add to cart, and complete payment. Non-functional testing checks quality attributes such as performance, security, usability, and reliability.

Testing can be performed at different levels. Unit testing checks individual modules or functions. Integration testing checks interaction between modules. System testing checks the complete system. Acceptance testing checks whether the system meets customer expectations and is ready for release.

Testing also supports verification and validation. Verification ensures that the software is built according to specifications. Validation ensures that the software satisfies user needs. Both are important for quality.

Software testing is defined as the process of executing a program with the intention of finding defects. It involves providing inputs to the software, observing outputs, and comparing them with expected results. Testing aims to reveal failures rather than prove correctness. Testing also includes activities such as reviewing requirements, design documents, and code. These non-execution-based activities are known as static testing. Together, static and dynamic testing help ensure software quality.

Testing is the process of executing a program with the intent of finding errors. However, modern definitions have expanded this scope.

- It is not just "execution" (dynamic testing) but also "checking documents" (static testing).
- It serves two purposes:
    1. **Validation:** "Are we building the right product?" (Does it meet user needs?)

NOTE

2. **Verification:** "Are we building the product right?" (Does it meet the technical specs?)

Testing is not limited to technical correctness; it also evaluates whether the software meets business objectives, user expectations, and operational needs. Effective testing builds confidence in the software product and supports informed decision-making.

6.3.3 Fundamental Test Process

The fundamental test process is a structured sequence of activities used in software testing. It ensures that testing is systematic and controlled rather than random. A well-defined test process improves efficiency, reduces mistakes, and ensures proper coverage.

The first stage of the fundamental test process is test planning and control. In this stage, the testing objectives, scope, strategy, resources, schedule, and risks are defined. Test planning includes selecting testing techniques, defining test levels, and deciding the tools to be used. Test control involves monitoring progress and adjusting plans based on results.

The second stage is test analysis and design. In this stage, test conditions are identified based on requirements and design documents. Test cases are designed to cover these conditions. Test cases include inputs, expected outputs, and execution steps. Test design also includes selection of boundary values, decision tables, and path coverage.

The third stage is test implementation and execution. In this stage, test cases are prepared for execution. Test scripts may be written for automation. Test data is created. Test environment is set up. Then, test cases are executed and actual results are recorded. Any mismatch between expected and actual results is reported as a defect.

The fourth stage is evaluating exit criteria and reporting. In this stage, test results are compared with predefined exit criteria. Exit criteria may include minimum pass rate, defect density limits, or coverage requirements. A test summary report is prepared, describing testing activities, results, defects found, and readiness for release.

The final stage is test closure activities. In this stage, testing is formally completed. Test documentation is finalized, lessons learned are recorded, and test artifacts are archived for future use. Closure activities also include evaluating the effectiveness of testing and improving future test processes.

The fundamental test process provides a structured framework for planning, executing, and controlling testing activities. It ensures systematic coverage and effective defect management. The process begins with test planning, where test objectives, scope, strategies,

NOTE

schedules, resources, and risks are defined. This phase ensures alignment with project goals.

In test analysis and design, requirements are analyzed to identify test conditions, and detailed test cases are designed. This ensures traceability between requirements and tests.

Testing is not a single activity but a lifecycle. The ISTQB (International Software Testing Qualifications Board) defines a standard fundamental test process:

1. **Planning and Control:** Defining the scope, objectives, and strategy (what to test, who will test).
2. **Analysis and Design:** Reviewing requirements and creating test cases (steps, input data, expected results).
3. **Implementation and Execution:** Setting up the test environment and running the tests.
4. **Evaluating Exit Criteria and Reporting:** Comparing actual results with expected results and logging bugs.
5. **Test Closure Activities:** Archiving testware, analyzing lessons learned, and handing over the test report.

Test implementation involves preparing test cases, test scripts, and test data. In test execution, test cases are executed, results are recorded, and defects are reported.

During test evaluation and reporting, test results are analyzed to assess quality and coverage. Finally, test closure involves documenting lessons learned, test summaries, and archiving test artifacts.

6.3.4 The Psychology of Testing

The psychology of testing refers to the human and behavioural aspects involved in software testing. Testing is not only a technical activity; it is also influenced by attitudes, emotions, communication, and organisational culture. Understanding psychology is important because testing often involves conflict between developers and testers.

One key psychological aspect is that developers and testers have different mindsets. Developers focus on building software and proving that it works. They often feel emotionally attached to their code. Testers focus on finding defects and proving that the software may fail. This difference can create tension. Developers may feel criticized when testers report defects, while testers may feel ignored if defects are not taken seriously.

Another psychological issue is that people generally dislike finding faults in their own work. Developers may unintentionally overlook defects because they assume their logic is correct. This is why

NOTE

independent testing is important. Independent testers can evaluate software without bias.

Communication is another important factor. Defect reporting must be clear, professional, and based on evidence. If testers report defects in a blaming manner, it can create conflict. If developers respond defensively, quality suffers. A positive quality culture encourages collaboration rather than blame.

Motivation also affects testing. Testing can be repetitive and time-consuming. Testers may lose motivation if they are under pressure or if their work is not valued. Organisations must recognize the importance of testing and support testers with proper tools and training.

The psychology of testing focuses on human attitudes and behaviors toward testing activities. Developers may unconsciously avoid finding faults in their own code, while testers must actively seek defects. This difference can create conflict if not managed properly.

Effective testing requires a mindset that views defect detection as a positive contribution to quality rather than criticism. Testers must be objective, detail-oriented, and persistent.

Testing requires a different mindset than development.

- **Constructive vs. Destructive:** Development is creative (building something new). Testing is often perceived as destructive (breaking what was built).
- **Confirmation Bias:** A developer testing their own code tends to test "for success" (to prove it works). A tester tests "for failure" (to prove it breaks).
- **Independence:** Independent testing (by a separate team) is more effective because it avoids the bias of the author.

A good tester needs "Professional Pessimism"—assuming there are bugs to be found—but also "Diplomacy" to report them without blaming the developer.

Developers and testers should collaborate constructively, sharing responsibility for quality.

A supportive organizational culture encourages open communication, mutual respect, and early involvement of testers. Understanding the psychology of testing helps reduce conflict and improves overall testing effectiveness.

6.3.5 Historical Perspective of Testing

In the early days of software development, testing was informal and limited to debugging after coding. As software systems grew in

NOTE

complexity, failures became more costly, leading to the development of structured testing methods.

The historical development of software testing shows how testing evolved as software became more complex. In early days of computing, software was small and developed by individual programmers. Testing was informal and often done by the same programmer who wrote the code. As software systems grew, failures became more costly, and testing became a separate discipline.

During the 1950s and 1960s, testing was mainly focused on debugging. Developers wrote code and fixed errors as they appeared. There was no structured testing process. In the 1970s, software engineering emerged as a discipline, and testing became more systematic. Researchers introduced concepts such as test case design, coverage criteria, and structured testing.

In the 1980s and 1990s, testing became more organized. Software companies began establishing testing teams and adopting quality assurance practices. Testing standards and methodologies such as black-box testing, white-box testing, and V-model testing became popular. Automation tools also started to appear.

In the 2000s, Agile development changed testing practices. Testing became continuous and integrated into development. Concepts such as test-driven development and continuous integration emphasized early testing. Automated regression testing became essential.

In recent years, DevOps and continuous delivery further expanded testing. Testing is now part of automated pipelines. Performance testing, security testing, and reliability engineering have become important due to increasing cyber threats and large-scale systems.

The introduction of formal SDLC models emphasized planned testing phases. Later, automated testing tools and standardized testing techniques improved efficiency and coverage. Modern methodologies such as Agile and DevOps integrate testing continuously, emphasizing early testing and rapid feedback.

1950s (Debugging Era): Testing was indistinguishable from debugging. It was just "getting the program to run."

1970s (Demonstration Era): The goal was to show that the software works (Positive testing).

1980s (Destruction Era): Glenford Myers defined testing as "executing a program with the intent of finding errors." The focus shifted to breaking the code.

1990s (Evaluation Era): Testing expanded to include measurement of quality and process improvement.

NOTE

2000s+ (Prevention Era): Testing begins early (Shift Left) to prevent defects in requirements before code is even written.

This evolution reflects the growing recognition of testing as a vital engineering discipline.

6.3.6 Definitions of Testing

Software testing has been defined in many ways by experts and standards. These definitions reflect the goals of testing such as defect detection, verification, validation, and confidence building. One common definition is that testing is the process of executing a program with the intent of finding errors. This definition highlights defect detection as the main purpose.

Another definition states that testing is the process of evaluating a system or its components to determine whether it satisfies specified requirements. This definition emphasizes verification. Testing is also defined as the process of validating that the software meets user needs and expectations. This definition focuses on validation and customer satisfaction.

Modern definitions also include risk reduction. Testing is considered a process that provides information about software quality and reduces the risk of failure.

Various definitions highlight different aspects of software testing:

- Testing is the execution of software to find defects.
- Testing is the process of verifying and validating software against requirements.
- Testing evaluates software quality and readiness for release.

These definitions emphasize that testing is both a defect detection and quality evaluation activity. Glenford Myers: "The process of executing a program with the intent of finding errors."

IEEE/ANSI: "The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component."

Dijkstra's Law**:** "Program testing can be used to show the presence of bugs, but never to show their absence." (You can never prove a software is 100% bug-free).

6.3.7 Approaches to Testing

Software testing can be performed using different approaches. The most common approaches are black-box testing, white-box testing, and grey-box testing. Black-box testing focuses on external behaviour of software without considering internal code structure. Testers design test cases

NOTE

based on requirements and inputs/outputs. This approach is useful for functional testing and acceptance testing.

White-box testing focuses on internal code structure. Testers examine logic, paths, and conditions. Techniques such as statement coverage, branch coverage, and path testing are used. This approach is useful for unit testing and code-level verification.

Grey-box testing is a combination of black-box and white-box approaches. Testers have partial knowledge of internal structure. They design tests based on both requirements and internal understanding. This approach is useful for integration testing and security testing. Another classification includes manual testing and automated testing. Manual testing is performed by humans. Automated testing uses scripts and tools. Automation is useful for regression testing and repeated test execution.

Testing approaches determine how testing is planned and executed. Black-box testing focuses on external behavior without considering internal code. White-box testing examines internal logic, control flow, and data structures. Grey-box testing combines both approaches.

There are two broad approaches to how we look at the software under test:

1. **Black Box Testing (Functional):**

   The tester does *not* know the internal code structure.

   Focus is on inputs and outputs.

   Analogy: Using a car (steering, pedals) without knowing how the engine works.

2. **White Box Testing (Structural):**

   The tester knows the internal logic and code.

   Focus is on code coverage, branches, and logic paths.

   Analogy: A mechanic repairing the engine.

3. **Grey Box Testing:**

   A combination of both (e.g., testing a database while knowing the schema but not the stored procedures).

Other approaches include manual testing, automated testing, exploratory testing, and risk-based testing. Each approach has advantages and limitations. Selecting the appropriate approach depends on project complexity, risk level, and resource availability.

6.3.8 Testing During Development Life Cycle

NOTE

Testing must be integrated throughout the software development life cycle. If testing is performed only at the end, defects introduced earlier become expensive to fix. Therefore, testing activities must begin from the requirement stage and continue until maintenance.

During requirement phase, testing includes requirement reviews and validation. Testers check requirements for completeness, clarity, and testability. During design phase, testing includes design reviews and creation of test strategy. Testers identify test conditions and prepare high-level test cases.

During coding phase, unit testing is performed. Developers test individual modules. During integration phase, integration testing checks interaction between modules. During system testing phase, the complete system is tested against requirements. During acceptance phase, customers perform acceptance testing to ensure software meets their needs. After deployment, maintenance testing and regression testing ensure that updates do not break existing functionality.

Testing is integrated into every phase of the SDLC. During requirements, test cases are derived to validate clarity and completeness. During design, test scenarios help evaluate architecture and interfaces. Testing is not a phase that happens *after* coding; it runs parallel to development. This is best represented by the V-Model (Verification and Validation Model).

- **Requirement Analysis** $\leftrightarrow$ **User Acceptance Testing (UAT):** The plan for UAT is created when requirements are gathered.
- **System Design** $\leftrightarrow$ **System Testing:** System test plans are created during the design phase.
- **Architecture Design** $\leftrightarrow$ **Integration Testing:** Interface tests are planned here.
- **Module Design** $\leftrightarrow$ **Unit Testing:** Unit tests are written by developers for individual modules.

During implementation, unit testing verifies individual components. Integration testing ensures correct interaction between modules. System testing validates the complete system, and acceptance testing confirms readiness for deployment. Early and continuous testing reduces defect propagation and improves quality.

NOTE

Activity Title: Finding Defects in a Software Application

Objective:
To understand the purpose of software testing by identifying defects in a simple application.

Materials Required:

- Computer/mobile phone
- Any simple application or website
- Notebook

Procedure/Steps:

1. Select an application or website.
2. Perform common operations such as login, navigation, data entry, and search.
3. Observe unexpected behaviour or incorrect results.
4. Record each identified defect.
5. Classify the defects as functional, usability, performance, or interface-related.

Outcome:
The learner will understand the purpose of software testing and gain practical experience in identifying defects.

6.3.9 Requirement Traceability Matrix

The Requirement Traceability Matrix (RTM) maps requirements to test cases, ensuring complete coverage. It helps track requirement changes and assess their impact on testing. RTM supports audits, quality reviews, and compliance verification. It ensures that no requirement is left untested and that testing remains aligned with business objectives. The RTM is a document (usually a spreadsheet) that maps and traces user requirements with test cases.

A Requirement Traceability Matrix (RTM) is a document that maps requirements to test cases. It ensures that every requirement is tested and verified. RTM is important for ensuring complete coverage and avoiding missing requirements.

RTM provides traceability between requirement IDs and test case IDs. It helps identify which requirements have been tested and which are pending. It also helps in impact analysis when requirements change. If a requirement changes, RTM shows which test cases must be updated. RTM also supports quality audits and compliance. In many projects,

NOTE

especially critical systems, traceability is mandatory. RTM ensures transparency and accountability.

Purpose:

1. Forward Traceability: Ensures every requirement has at least one test case covering it. (Nothing is missed).
2. Backward Traceability: Ensures every test case traces back to a valid requirement. (We aren't testing features nobody asked for).
3. Change Management: If a requirement changes, we know exactly which test cases need to be updated.

Req ID	Requirement Description	Test Case ID	Test Status
REQ-01	User Login	TC-001	Pass
REQ-01	User Login (Invalid Pass)	TC-002	Pass
REQ-02	Upload Photo	TC-003	Fail

6.3.10 Essentials of Software Testing, Workbench

Essentials of software testing include test plans, test cases, test data, defect reports, and metrics. A testing workbench provides tools and environments to support testing activities. Testing workbenches may include test management tools, automation frameworks, defect tracking systems, and continuous integration platforms. These tools improve efficiency, accuracy, and traceability.

The essentials of software testing refer to the basic elements required to perform testing effectively. These include test planning, test case design, test data preparation, test environment setup, execution, defect reporting, and result analysis. A testing workbench is a structured framework that defines how testing activities are performed. It includes guidelines, templates, checklists, tools, and processes. A workbench ensures that testing is consistent and repeatable.

The workbench includes components such as test plan document, test case templates, defect report formats, test execution logs, and summary reports. It also includes tools such as defect tracking systems, automation tools, and reporting dashboards. A well-designed workbench improves testing productivity and quality. It reduces confusion, supports documentation, and ensures that testing follows standards.

The "Test Workbench" is a concept that describes a standard way of performing any testing task. It consists of:

1. Input: The criteria to start work (e.g., Specs, Code).

NOTE

2. Process (The Workbench): The procedures to follow (e.g., Create Test cases, Run Tests).
3. Check Procedures: Quality control on the testing itself (e.g., Peer review of test cases).
4. Output: The deliverable (e.g., Test Report, Defect Log).

Check Your Progress 6.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. The main purpose of software testing is to:
A) Prove the program has no errors
B) Find defects and ensure the software meets requirements
C) Increase coding speed
D) Remove documentation

_____

Q2. Testing can show the presence of defects, but it cannot prove the absence of defects.

_____

Q3. A document that maps requirements to test cases is called _____.

_____

Q4. Write any two reasons why software testing is necessary.

_____

Q5. Which of the following is a part of the fundamental test process?
A) Requirement gathering only
B) Test planning and control
C) Only coding
D) Hardware installation

_____

Fill in the Blanks

Testing aims to _____ defects.

RTM ensures _____ coverage.

True/False

Testing guarantees defect-free software.

Early testing reduces cost of defect correction.

6.4 SUMMARY

This unit introduced software testing and explained its importance in software quality assurance. It discussed the necessity of testing, the meaning and definitions of testing, and the fundamental test process. The unit explained the psychology of testing and how human factors influence testing effectiveness. It also presented the historical evolution of software testing from debugging to modern continuous testing practices. Different testing approaches such as black-box, white-box, and grey-box testing were discussed. The unit explained how testing is integrated throughout the software development life cycle. Requirement Traceability Matrix was described as an important tool for ensuring requirement coverage. Finally, the essentials of software testing and the concept of a testing workbench were explained as the foundation for systematic testing and quality improvement.

Software testing is a vital component of Software Quality Assurance that ensures software reliability, correctness, and user satisfaction. Understanding testing concepts, processes, approaches, and human factors enables effective defect detection and quality improvement throughout the SDLC.

6.5 GLOSSARY

- **Testing** – Process of evaluating software to find defects
- **Defect** – An error that causes incorrect behavior
- **RTM** – Requirement Traceability Matrix
- **Verification** – Ensuring software is built correctly
- **Validation** – Ensuring correct software is built

6.6 ANSWERS TO CHECK YOUR PROGRESS

Q1. Answer: B

Explanation: Testing is performed to detect defects and verify that the software satisfies specified requirements and user needs.

Q2. Answer: True

Explanation: Testing can reveal defects, but it is impossible to test all combinations; therefore, it cannot guarantee a defect-free product.

Q3. Answer: Requirement Traceability Matrix (RTM)

Explanation: RTM ensures that every requirement is tested and no requirement is missed.

NOTE

Q4. Explanation: Early testing reduces defect cost and ensures quality, reliability, and user confidence.

Q5. Answer: B

Explanation: The fundamental test process includes test planning, test analysis, test design, execution, and evaluating results.

Fill in the Blanks

Testing aims to _____ defects.

RTM ensures _____ coverage.

True/False

Testing guarantees defect-free software.

Early testing reduces cost of defect correction.

6.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3785	

6.8 REFERENCE

- Myers, G. J., *The Art of Software Testing*
- Pressman, R. S., *Software Engineering*
- Sommerville, I., *Software Engineering*
- IEEE Software Testing Standards

6.9 EXERCISE

Part–1

A. Multiple Choice Questions (MCQs)

Q1. Software testing is mainly performed to:

A) Increase software size
B) Detect defects and ensure quality

NOTE

C) Reduce documentation
D) Increase coding speed

Q2. Which of the following is a principle of software testing?

A) Testing shows presence of defects
B) Testing proves software is defect-free
C) Testing removes all defects permanently
D) Testing is not required for small projects

Q3. Which testing is performed without executing the code?

A) Dynamic testing
B) Black-box testing
C) Static testing
D) Load testing

Q4. Regression testing is performed to:

A) Test new features only
B) Ensure changes have not affected existing features
C) Increase performance
D) Reduce project cost

Q5. Which of the following is NOT a level of testing?

A) Unit testing
B) Integration testing
C) System testing
D) Marketing testing

Answers with explanation

1. **Answer: B**
   Explanation: Testing is done to identify defects and confirm that the software meets requirements and quality standards.
2. **Answer: A**
   Explanation: Testing can reveal defects but cannot guarantee that software is completely defect-free.
3. **Answer: C**
   Explanation: Static testing includes reviews, inspections, and walkthroughs where code is not executed.
4. **Answer: B**
   Explanation: Regression testing checks that recent changes do not introduce defects in previously working functionality
5. **Answer: D**
   Explanation: Marketing testing is not a software testing level; unit, integration, system, and acceptance are valid levels.

NOTE

B. Small Answer Type Questions (5 Questions)

1. Define software testing.
2. Write any two objectives of testing.
3. What is the difference between verification and validation?
4. Define regression testing.
5. Write any two testing principles.

Part–2

C. Long Answer Type Questions (200 words each)

1. Explain the importance of software testing in software development.
2. Discuss different levels of testing with suitable examples.
3. Explain the principles of software testing in detail.
4. Describe verification and validation with examples.
5. Discuss the role of regression testing in software maintenance.

Part–3

D. Long Answer Type Questions (300 words each)

1. Explain software testing in detail and discuss its objectives and benefits in software projects.
2. Discuss different levels of testing (Unit, Integration, System, Acceptance) with real-world examples.
3. Explain verification and validation in detail and compare them with suitable examples.
4. Describe software testing principles and explain how they help in effective defect detection.
5. Discuss regression testing in detail. Explain why it is important in Agile and DevOps environments.

UNIT–7
TESTING PROCESS

UNIT STRUCTURE

7.1 Introduction
7.2 Objective
7.3 Important Features of Testing Process
  7.3.1 Misconceptions About Testing
  7.3.2 Principles of Software Testing
  7.3.3 Salient Features of Good Testing
  7.3.4 Test Policy
  7.3.5 Test Strategy or Test Approach
  7.3.6 Test Planning
  7.3.7 Testing Process and Number of Defects Found in Testing
  7.3.8 Test Team Efficiency
7.4 Summary
7.5 Glossary
7.6 Answer to check your progress
7.7 Suggested readings - OER
7.8 Reference
7.9 Exercise

7.1 INTRODUCTION

Software Quality Assurance (SQA) and Software Testing are critical activities in the software development life cycle (SDLC) that ensure a software product meets specified requirements, is free from defects, and satisfies user expectations. In today's highly competitive and technology-driven environment, software systems are expected to be reliable, secure, efficient, and user-friendly. Any failure in software can result in financial loss, damage to reputation, legal consequences, and even risks to human life in safety-critical systems. Therefore, a systematic and well-defined testing process is essential to assure software quality.

The testing process refers to a structured set of activities carried out to evaluate software artifacts and identify defects. It is not merely the act of executing test cases but a comprehensive process that begins at the early stages of software development and continues throughout the project lifecycle. Modern testing philosophies emphasize "testing early

NOTE

and testing continuously" to prevent defect leakage into later stages where the cost of fixing defects becomes significantly higher.

Software testing plays a dual role: verification and validation. Verification ensures that the software is built correctly according to specifications, while validation ensures that the right product is built to meet user needs. The testing process integrates both these aspects through planned activities such as test planning, test design, test execution, defect reporting, and test closure.

The testing process is not a random activity of "banging on the keyboard" to see what breaks. It is a disciplined, mathematical, and managerial process designed to reduce risk. This unit explores the core principles that govern effective testing, the hierarchy of testing documentation (Policy, Strategy, Plan), and the metrics used to measure the efficiency of a test team.

In the context of Software Quality Assurance, testing acts as a quality control mechanism. While SQA focuses on process-oriented activities such as standards, audits, and process improvement, testing focuses on product-oriented activities that reveal defects. A robust testing process bridges the gap between development and quality assurance, ensuring that quality is built into the product rather than inspected at the end.

The importance of a well-defined testing process has increased with the adoption of agile, DevOps, and continuous delivery models. These models demand faster releases without compromising quality. As a result, testing processes have evolved to include automation, continuous integration testing, regression testing, and risk-based testing. Despite technological advancements, the fundamental principles of the testing process remain unchanged: systematic planning, objective evaluation, and continuous improvement.

This unit introduces the testing process as a core component of Software Quality Assurance. It explains the objectives, important features, misconceptions, principles, planning activities, and factors influencing test effectiveness and efficiency. Understanding the testing process equips learners with the knowledge required to design effective tests, manage testing activities, and contribute to the development of high-quality software systems.

7.2 OBJECTIVE

After studying this unit, you should be able to:

- Identify and refute common myths and misconceptions about software testing.

NOTE

- Explain the seven fundamental principles of software testing.
- Distinguish between Test Policy, Test Strategy, and Test Plan.
- Analyse the relationship between the testing process and defect discovery.
- Evaluate the efficiency of a test team using standard metrics.

Another important objective is verification of requirements. The testing process ensures that all specified requirements have been implemented correctly and completely. By mapping test cases to requirements through traceability matrices, testers confirm that no requirement is left untested and no unnecessary functionality is introduced.

Validation of user expectations is also a key objective. Software that meets technical specifications but fails to satisfy user needs cannot be considered successful. Through acceptance testing and user-centric test scenarios, the testing process validates that the software performs as intended in real-world conditions.

7.3 IMPORTANT FEATURES OF TESTING PROCESS

The testing process is characterized by several important features that distinguish it from ad-hoc testing activities. These features ensure that testing is systematic, repeatable, and effective.

A key feature of the testing process is its structured nature. Testing activities follow a defined sequence, starting from requirement analysis and ending with test closure. Each phase has clear inputs, outputs, and entry-exit criteria, ensuring transparency and control.

Another important feature is early involvement. Testing is no longer confined to the final stages of development. Testers participate in requirement reviews and design discussions to identify ambiguities and inconsistencies early. This proactive approach helps prevent defects rather than merely detecting them.

Documentation is also a critical feature. Test plans, test cases, test scripts, defect reports, and test summary reports provide a formal record of testing activities. These documents support traceability, auditability, and knowledge transfer.

The testing process is objective and evidence-based. Test results are based on observed behavior rather than assumptions or opinions. Defects are reported with clear reproduction steps, expected results, and actual results, enabling developers to fix issues efficiently.

Another significant feature is adaptability. The testing process can be tailored based on project size, complexity, risk level, and development methodology. Agile projects may adopt lightweight documentation and

NOTE

continuous testing, while safety-critical systems may require rigorous testing and formal approvals.

Measurement and metrics form an integral part of the testing process. Metrics such as defect density, test coverage, defect leakage, and test execution progress help assess test effectiveness and quality levels. These metrics support continuous monitoring and improvement.

Finally, collaboration is a defining feature of the testing process. Effective testing requires close coordination among testers, developers, business analysts, and stakeholders. Communication and collaboration ensure shared understanding of requirements, priorities, and quality goals.

7.3.1 Misconceptions About Testing

Software testing is often misunderstood, leading to unrealistic expectations and ineffective practices. One of the most common misconceptions is that testing is only about finding bugs. While defect detection is an important goal, testing is equally focused on *preventing defects*, evaluating quality, and providing confidence to stakeholders. Through activities such as requirement reviews, design reviews, and early test involvement, testing helps prevent errors from being introduced in later stages.

Another widespread misconception is that testing can ensure a completely defect-free product. In reality, exhaustive testing is practically impossible due to constraints of time, cost, and resources. Software systems can have an infinite number of input combinations and execution paths. Testing can reduce risk and improve confidence, but it cannot prove the absence of defects. The realistic goal of testing is to reduce the probability of failure to an acceptable level.

A third misconception is that testing should start only after coding is complete. This late-start approach often results in higher defect-fixing costs and project delays. Modern software engineering emphasizes early testing, where testers are involved from the requirement and design phases. Early testing helps identify ambiguities, inconsistencies, and missing requirements before they propagate into code.

Many stakeholders believe that testing is an easy task that anyone can do. This assumption undermines the skill and expertise required for effective testing. Testing requires analytical thinking, domain knowledge, understanding of user behavior, and familiarity with testing techniques and tools. Poorly designed tests can miss critical defects, giving a false sense of quality.

Another misconception is that developers should not test their own code. While independent testing is important for objectivity, developer testing such as unit testing is essential for early defect detection.

Testing should be a shared responsibility, with developers, testers, and users contributing at different levels.

There is also a belief that automation can replace manual testing completely. Automation is a powerful tool, but it is best suited for repetitive, regression, and performance tests. Exploratory testing, usability testing, and ad-hoc testing still require human judgment and creativity.

Before understanding what testing *is*, it is crucial to understand what it *is not*. Several myths persist in the industry that can derail a project.

1. "Testing is expensive."

   Reality: Testing costs money, but *not* testing costs more. The cost of fixing a critical bug in production (e.g., a security breach) is exponentially higher than the cost of a tester finding it early.

2. "Testing is easy / Anyone can test."

   Reality: Testing requires logical thinking, domain knowledge, technical skills (SQL, scripting), and a destructive mindset. It is a specialized engineering discipline.

3. "Testing ensures 100% bug-free software."

   Reality: No software is ever 100% bug-free. Testing only reduces the probability of undiscovered defects remaining.

4. "Testing starts after coding is finished."

   Reality: In modern Quality Assurance (Agile/DevOps), testing starts immediately during the requirements phase (Shift Left).

Finally, many organizations consider testing as a cost center rather than a value-adding activity. This misconception leads to inadequate testing budgets and schedules. In reality, effective testing reduces long-term costs by preventing expensive failures in production and improving customer satisfaction.

Understanding and addressing these misconceptions is essential for building a mature testing culture. When testing is viewed as a strategic quality activity rather than a mere defect-finding exercise, organizations are better positioned to deliver reliable and high-quality software products.

7.3.2 Principles of Software Testing

Software testing is guided by a set of fundamental principles that improve its effectiveness and efficiency. These principles provide a conceptual foundation for planning, designing, and executing tests in a systematic manner.

NOTE

The first principle states that testing shows the presence of defects, not their absence. Testing can reveal errors that exist in the software, but it cannot guarantee that no defects remain. This principle sets realistic expectations and highlights the importance of risk-based decision-making.

The second principle emphasizes that exhaustive testing is impossible. Since it is impractical to test all possible combinations of inputs and conditions, testing efforts must be prioritized. Critical functionalities, high-risk areas, and frequently used features should receive greater attention.

Another important principle is early testing. Testing activities should begin as early as possible in the SDLC. Reviewing requirements and design documents helps detect defects at a stage where they are cheaper and easier to fix. Early testing also reduces rework and improves overall project efficiency.

The International Software Testing Qualifications Board (ISTQB) defines seven standard principles that apply to all software testing.

1. Testing shows the presence of defects: Testing can prove that bugs exist, but it cannot prove that they *don't* exist.
2. Exhaustive testing is impossible: You cannot test every possible input combination (e.g., a calculator app has infinite input possibilities). We use risk analysis to prioritize.
3. Early testing: Testing should start as early as possible in the SDLC. Fixing a requirement defect is cheap; fixing a production defect is expensive.
4. Defect clustering: A small number of modules usually contain most of the defects (Pareto Principle: 80% of bugs are in 20% of the code).
5. Pesticide paradox: If you repeat the same tests over and over, eventually they will stop finding new bugs (like insects building immunity to pesticide). You must regularly update your test cases.
6. Testing is context-dependent: You test a banking app differently than you test a mobile game. The approach changes based on the context.
7. Absence-of-errors fallacy: Finding and fixing bugs doesn't help if you built a system that users don't want. A "bug-free" system is useless if it doesn't meet business needs.

The defect clustering principle suggests that a small number of modules usually contain the majority of defects. By analyzing defect data, testers can focus efforts on error-prone areas, improving test effectiveness. The

NOTE

pesticide paradox principle warns that repeated execution of the same tests will eventually stop finding new defects. Test cases must be regularly reviewed and updated to address new risks and changes in the software.

Testing is also context-dependent, meaning that the approach and techniques used depend on the nature of the software. Safety-critical systems require rigorous testing, while simple applications may require lighter approaches. Finally, the absence-of-errors fallacy states that defect-free software is not useful if it does not meet user needs. Testing must validate not only technical correctness but also business and user requirements.

By applying these principles, testers can design meaningful tests, manage stakeholder expectations, and contribute effectively to software quality assurance.

7.3.3 Salient Features of Good Testing

Good testing is characterized by a combination of systematic planning, technical rigor, and user-focused evaluation. One of the most important features of good testing is that it is well-planned and structured. Testing activities are guided by a test plan that defines scope, objectives, resources, schedules, and responsibilities.

Another key feature is traceability. Each test case should be linked to specific requirements, ensuring that all requirements are adequately tested. Traceability helps identify gaps in coverage and supports impact analysis when requirements change. Good testing is also objective and evidence-based. Test results are derived from actual execution and observation rather than assumptions. Defects are reported with clear details such as steps to reproduce, expected results, actual results, and severity levels.

What differentiates "good" testing from "bad" testing?

- Independence: Tests are more effective when designed by someone other than the author of the code.
- Traceability: Every test case is linked to a requirement (RTM).
- Repeatability: If a test fails today, it should fail tomorrow under the same conditions.
- Coverage: It covers both Positive scenarios (happy path) and Negative scenarios (error handling).

A balance between positive and negative testing is another important feature. Positive tests verify that the system works as expected under valid conditions, while negative tests check how the system behaves under invalid or unexpected inputs.

NOTE

Effective testing includes both functional and non-functional testing. While functional testing validates features, non-functional testing evaluates performance, security, usability, reliability, and compatibility.

Good testing is risk-driven and prioritized. Limited resources are allocated to areas with higher business or technical risk. This ensures optimal use of time and effort.

The use of appropriate tools and automation enhances testing efficiency. Automation is particularly valuable for regression and performance testing. However, good testing also recognizes the importance of manual and exploratory testing. A good testing involves continuous feedback and improvement. Lessons learned from defects and test results are used to refine test strategies and development processes.

The testing process also aims to assess software quality attributes such as reliability, performance, security, usability, and compatibility. Non-functional testing ensures that the system behaves consistently under varying conditions and meets defined quality benchmarks. Risk mitigation is another critical objective. Testing helps identify high-risk areas of the system and evaluates their impact on overall functionality. Risk-based testing prioritizes critical features, ensuring that potential failures with severe consequences are addressed first.

Additionally, the testing process supports decision-making. Test reports, defect metrics, and quality indicators provide stakeholders with objective information about the readiness of the software for release. Management uses this information to decide whether to proceed with deployment or invest additional effort in quality improvement.

Finally, the testing process contributes to continuous improvement. By analyzing defect trends, test effectiveness, and process gaps, organizations can refine their development and testing practices. Lessons learned during testing are fed back into the SDLC, leading to better processes and higher-quality products in future projects.

7.3.4 Test Policy

A test policy is a high-level organizational document that defines the overall vision, objectives, and principles governing software testing activities. It provides a common framework for all projects and ensures consistency in testing practices across the organization.

The primary purpose of a test policy is to align testing activities with organizational quality goals. It defines why testing is performed, what level of quality is expected, and how testing contributes to business success. A test policy typically outlines roles and responsibilities, compliance requirements, and adherence to standards.

The Test Policy is the highest-level document.

NOTE

- Definition: A document describing *why* the organization tests. It outlines the organization's philosophy toward quality.
- Audience: Senior Management / Stakeholders.
- Content: "We will achieve ISO 9001 certification," or "We prioritize customer data safety above all else."

A well-defined test policy promotes standardization. It ensures that testing follows approved methodologies, documentation standards, and reporting practices. This consistency improves transparency and facilitates audits and assessments.

The test policy also supports governance and accountability. By clearly defining ownership and responsibilities, it ensures that testing activities are planned, executed, and monitored effectively.

Another important aspect of the test policy is risk management. It defines how risks are identified, prioritized, and mitigated through testing. This helps organizations focus on critical areas and reduce potential failures.

The test policy is not static; it evolves with changes in technology, business needs, and regulatory requirements. Regular review and updates ensure its continued relevance. Overall, the test policy acts as a guiding document that establishes a quality-focused testing culture within the organization.

Activity Title: Preparing a Basic Testing Process

Objective:
To understand and practise the major steps involved in a software testing process.

Materials Required:

- Sample software requirement
- Notebook
- Testing template

Procedure/Steps:

1. Select a simple requirement, such as a login facility.
2. Identify test objectives.
3. Prepare test conditions and test cases.
4. Execute the test cases manually.
5. Record expected and actual results.
6. Report any failures or defects.

NOTE

Outcome:
The learner will be able to understand and perform the basic stages of a software testing process.

7.3.5 Test Strategy or Test Approach

The test strategy, also known as the test approach, describes how testing will be carried out for a specific project or program. It translates the high-level test policy into actionable plans and decisions.

A test strategy defines the scope of testing, including test levels such as unit, integration, system, and acceptance testing. It also specifies the types of testing to be performed, such as functional, performance, security, and usability testing.

Another key element of the test strategy is resource planning. It identifies required tools, environments, and human resources. Decisions regarding test automation and tool selection are also part of the strategy.

The Test Strategy bridges the gap between policy and actual planning.

- Definition: A document describing *how* the testing will be done across the organization or program. It is often generic and reusable across projects.
- Content:

  Tools to be used (e.g., Selenium for web, Appium for mobile).

  Types of testing (e.g., "We will do 100% regression testing").

  Defect reporting standards (Severity vs. Priority).

  Entry and Exit criteria.

The strategy includes risk assessment and mitigation plans. High-risk areas are prioritized, and contingency plans are developed to address potential issues. The test strategy ensures efficient coordination among stakeholders by defining communication mechanisms, reporting formats, and review processes. A well-defined test strategy improves predictability, optimizes resource utilization, and enhances test effectiveness.

7.3.6 Test Planning

Test planning is a critical phase of the testing process that involves defining objectives, scope, resources, schedules, and deliverables. A test plan acts as a roadmap that guides testing activities throughout the project lifecycle.

NOTE

The test plan identifies what will be tested and what will not be tested, ensuring clarity and focus. It includes test objectives, test items, and acceptance criteria. Resource planning is a key component of test planning. It involves estimating effort, allocating personnel, and scheduling activities. Accurate planning helps avoid delays and resource conflicts.

The Test Plan is specific to a single project or release.

- Definition: A document describing *who, what, when, and where* for a specific project.
- Content:

  Scope: What modules are in scope and out of scope.

  Schedule: Testing dates (Start/End).

  Resources: Who is the test lead? Who are the testers?

  Environment: Hardware/Software requirements (e.g., "We need 3 servers with Windows Server 2019").

  Risks: "If the server arrives late, testing will be delayed."

Risk identification and mitigation are integral to test planning. Potential risks such as environment unavailability or skill gaps are analyzed and addressed proactively.

The test plan also defines entry and exit criteria for each test phase, ensuring disciplined progression.

Effective test planning improves coordination, reduces uncertainty, and enhances overall test efficiency.

7.3.7 Testing Process and Number of Defects Found in Testing

The number and nature of defects found during testing provide valuable insights into both product quality and process maturity. Defect metrics help evaluate the effectiveness of the testing process.

Early detection of defects indicates a proactive and mature testing approach. Defects found in later stages or production suggest gaps in earlier testing phases. Analyzing defect trends helps identify root causes and areas for improvement. Metrics such as defect density, severity distribution, and defect leakage are commonly used.

There is a correlation between the rigor of the testing process and the defect discovery rate.

- **The S-Curve:** Typically, defect discovery follows an S-Curve.

  Early Phase: Slow discovery (setup time, learning curve).

  Middle Phase: Rapid discovery (peak testing).

  Late Phase: Discovery slows down (saturation/stability).

NOTE

- **Decision to Stop:** Managers often use this curve to decide when to stop testing. If the "Defect Arrival Rate" drops below a certain threshold (e.g., 0 critical bugs in the last 3 days), the product is considered stable enough for release.

A healthy testing process focuses on defect prevention as well as detection. Continuous monitoring and analysis support informed decision-making.

7.3.8 Test Team Efficiency

Test team efficiency refers to the ability of the testing team to achieve quality objectives within given constraints of time and resources. It depends on skills, tools, processes, and collaboration.

Skilled testers with domain and technical knowledge improve defect detection effectiveness. Training and knowledge sharing enhance team capability.

1. Defect Removal Efficiency (DRE):

This measures how many bugs the test team found vs. how many the users found.

$$DRE = \frac{\text{Defects found during testing}}{\text{Defects found during testing} + \text{Defects found by customers}} \times 100$$

- *Example:* Testers found 90 bugs. Users found 10 bugs. DRE = 90/100 = 90%. High efficiency.

2. Test Execution Productivity:

- Number of test cases executed per hour/day per tester.

The use of automation and tools increases productivity, especially for regression testing. Clear communication and collaboration with development teams further improve efficiency. Metrics such as test execution rate and defect detection effectiveness help assess team performance.

3. Defect Rejection Ratio:

- If testers report 100 bugs, but developers reject 40 of them (as "not a bug" or "works as designed"), the test team's efficiency is low because they are misunderstanding the requirements.

Check Your Progress 7.1

Notes:
(a) Space is given below for your answer.
(b) Check your answer with the one given at the end of this unit.

Fill-in-the-Blank Questions:

The primary objective of software testing is to _____ defects in the software.

Testing that focuses on validating user requirements is known as _____ testing.

A document that defines the overall approach to testing in an organization is called _____.

True/False Questions:

- Exhaustive testing of a software system is always possible.
- Early testing helps reduce the cost of defect correction.
- Test automation can completely replace manual testing.

Q1. Which of the following is a common misconception about testing?
A) Testing improves quality
B) Testing can remove all defects completely
C) Testing helps reduce risk
D) Testing supports verification

_____

Q2. One principle of testing states that exhaustive testing is impossible.

_____

Q3. A formal document that defines how testing will be carried out is called a Test _____.

_____

Q4. Write any two salient features of good testing.

Answer (Any two):

Well-planned and systematic testing
Early testing in SDLC
Clear test cases with expected results
Proper defect reporting and tracking
Repeatable and measurable testing process

Q5. Test strategy mainly defines:
A) Salary of testers
B) Overall approach and methods of testing
C) Only coding rules
D) Only user interface design

_____
_____

NOTE

7.4 SUMMARY

This unit provided a comprehensive understanding of the testing process within Software Quality Assurance. It discussed objectives, misconceptions, principles, planning, and efficiency factors. A systematic testing process ensures delivery of reliable, high-quality software and supports continuous improvement. Misconceptions like "testing is easy" must be corrected to build a strong QA culture.

- The 7 Principles (including Pesticide Paradox and Defect Clustering) guide the testing strategy.
- Documentation Hierarchy: Policy (Why) $\rightarrow$ Strategy (How) $\rightarrow$ Plan (When/Who).
- Efficiency: A good test team is measured not just by the volume of bugs found, but by the percentage of total bugs caught before the customer sees them (DRE).

7.5 GLOSSARY

• Software Testing: Process of evaluating software to identify defects.

• Test Policy: Organizational guidelines for testing.

• Test Strategy: Project-level approach to testing.

• Defect: Deviation from expected behavior.

• Test Plan: Document defining testing activities.

7.6 ANSWER TO CHECK YOUR PROGRESS – POSSIBLE ANSWERS

Fill-in-the-Blank Questions:

Answer: identify

Answer: acceptance

Answer: test policy

True/False Questions:

Answer: False

Answer: True

Answer: False

Q1. Answer: B
Explanation: A common misconception is that testing can remove all

NOTE

defects. In reality, testing reduces risk but cannot guarantee zero defects.

Q2. Answer: True
Explanation: It is not possible to test every input combination in real systems due to time and cost limitations.

Q3. Answer: Plan
Explanation: A test plan includes scope, objectives, resources, schedule, tools, and testing strategy.

Q4. Explanation: Good testing is structured, measurable, and ensures defects are identified early with proper documentation.

Q5. Answer: B
Explanation: Test strategy defines the high-level approach of testing, including test levels, types, tools, and techniques.

7.7SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3786	

7.8 REFERENCE

• Pressman, R. S., & Maxim, B. R. (2019). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill.

• Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education.

• Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing* (3rd ed.). Wiley.

• IEEE Computer Society. (2014). *Guide to the Software Engineering Body of Knowledge (SWEBOK).*

7.9 EXERCISE

Part–1

A. Multiple Choice Questions (MCQs)

NOTE

1. Which of the following best describes software testing?

   A) Writing program code
   B) Executing software to find defects
   C) Designing software architecture
   D) Installing software

2. Which principle states that testing can show the presence of defects but not their absence?

   A) Defect clustering
   B) Pesticide paradox
   C) Testing shows presence of defects
   D) Early testing

3. Which document defines the project-level testing approach?

   A) Test policy
   B) Test strategy
   C) Requirement specification
   D) User manual

4. **The first step in the software testing process is:**

   A) Test execution
   B) Test planning
   C) Defect reporting
   D) Test closure

5. **Test cases are prepared mainly during:**

   A) Test planning
   B) Test design
   C) Test closure
   D) Maintenance

Answers with explanation

1. **Answer:** B) Executing software to find defects
2. **Answer:** C) Testing shows presence of defects
3. **Answer:** B) Test strategy
4. **Answer: B**
   Explanation: Test planning is the first stage where scope, objectives, resources, and schedule are defined.
5. **Answer: B**
   Explanation: Test design includes creating test cases, test data, and test scenarios.

B. Small Answer Type Questions (5 Ques)

1. What is meant by testing process?

NOTE

2. Write any two activities in test planning.
3. What is the purpose of test design?
4. Define defect tracking.
5. What is test closure?

Part–2

C. Long Answer Type Questions (200 words)

1. Explain the phases of software testing process.
2. Discuss the importance of test planning in software testing.
3. Explain test design and test case preparation process.
4. Describe defect life cycle and defect tracking process.
5. Explain test closure activities and their importance.

Part–3

D. Long Answer Type Questions (300 words)

1. Explain the complete testing process in detail with suitable examples from software projects.
2. Discuss the role of test planning and test strategy in ensuring successful testing.
3. Explain test case design, test data preparation, and test execution with examples.
4. Describe defect management process in detail including defect reporting, tracking, and closure.
5. Explain test closure activities in detail and discuss the importance of test summary report and lessons learned.

UNIT–8
MUTATION TESTING

UNIT STRUCTURE

8.1 Introduction
8.2 Objective
8.3 Mutation Testing
  8.3.1 Challenges in Testing
  8.3.2 Test Team Approach
  8.3.3 Process Problems Faced by Testing
  8.3.4 Cost Aspect of Testing
  8.3.5 Establishing Testing Policy
  8.3.6 Methods, Structured Approach to Testing
8.4 Summary
8.5 Glossary
8.6 Answer to check your progress
8.7 Suggested readings - OER
8.8 Reference
8.9 Exercise

8.1 INTRODUCTION

Mutation Testing is an advanced and powerful software testing technique used to evaluate the quality and effectiveness of test cases. Unlike conventional testing methods that focus primarily on detecting defects in the software, mutation testing focuses on assessing the strength of the test suite itself. It answers a critical question in Software Quality Assurance:

Are the existing test cases capable of detecting faults if they occur?

In mutation testing, small, deliberate changes called mutations are introduced into the source code. These modified versions of the program are known as mutants. Each mutant represents a potential fault that a developer might accidentally introduce during coding. The test cases are then executed against these mutants. If a test case detects the change and causes the mutant to fail, the mutant is said to be killed. If the mutant passes all test cases, it is said to have survived, indicating a weakness in the test suite.

As software systems become more critical, standard testing methods (checking if the code works) are sometimes not enough. We also need to

NOTE

check if our *tests* are working. This unit introduces Mutation Testing, a powerful technique to test the quality of your test cases. Additionally, we will explore the real-world management challenges facing test teams, including process bottlenecks, cost implications, and the necessity of establishing a firm testing policy.

Mutation testing is rooted in two fundamental hypotheses the Competent Programmer Hypothesis, which assumes that programmers generally write code that is close to correct, and the Coupling Effect, which suggests that complex faults are combinations of simple faults. By testing small changes, mutation testing indirectly verifies the detection of more complex errors.

This technique plays a vital role in improving software reliability, especially in safety-critical systems such as healthcare, aviation, banking, and defense applications. Although mutation testing is computationally expensive, advancements in automation tools and computing power have made it increasingly practical.

Within Software Quality Assurance, mutation testing complements other testing techniques by providing quantitative measures such as the mutation score, which reflects test effectiveness. This unit introduces mutation testing, its objectives, challenges, team approaches, cost considerations, testing policies, and structured methods for effective implementation.

8.2 OBJECTIVE

After studying this unit, you should be able to:

- Define Mutation Testing and explain how it evaluates test suite quality.
- Identify the common technical and organizational challenges in testing.
- Analyze the "Process Problems" that slow down testing efforts.
- Calculate and justify the Cost of Quality (CoQ) in software projects.
- Outline the steps to establish a robust Testing Policy.
- Differentiate between various structured approaches to testing.

8.3 MUTATION TESTING

Mutation testing is a fault-based testing technique that systematically modifies program statements to create mutants. These mutations typically involve small syntactic changes such as replacing arithmetic

NOTE

operators, altering relational conditions, or modifying logical expressions.

The mutation testing process begins with the creation of mutants using predefined mutation operators. Each operator represents a common programming error, such as replacing "+" with "-" or changing ">" to "<". The original program is referred to as the base program, while the modified versions are mutants.

nce mutants are generated, the existing test cases are executed against each mutant. If a test case produces different output for the mutant compared to the original program, the mutant is killed. The effectiveness of the test suite is measured using the mutation score, calculated as the ratio of killed mutants to the total number of non-equivalent mutants.

Mutation testing is a fault-based testing technique used to evaluate the quality of existing software tests. It involves modifying a program in small ways (creating "mutants") and running the test suite against the mutated code.

- The Concept: If a test suite is good, it should fail when run against broken code. If your tests still pass even after you broke the code, your tests are weak.
- The Process:
    1. Create Mutants: The tool automatically changes the source code.
        - *Example:* Change if (a > b) to if (a < b).
    2. Execute Tests: Run the test suite against the mutant.
    3. Kill the Mutant: If the test fails, the mutant is "killed" (Success).
    4. Mutant Survives: If the test passes, the mutant "survived" (Failure of the test case).

Key Metric:

Mutation Score = (Killed Mutants / Total Mutants) $\times$ 100

A high score indicates a robust test suite.

Mutation testing is highly effective in identifying weak test cases, but it also faces challenges such as equivalent mutants, high computational cost, and tool complexity. Despite these challenges, it remains one of the most rigorous methods for test quality evaluation.

8.3.1 Challenges in Testing

Software testing, including mutation testing, faces several challenges that affect its efficiency and effectiveness. One major challenge is the

NOTE

complexity of modern software systems. Applications often involve distributed architectures, multiple platforms, third-party integrations, and rapidly changing requirements, making comprehensive testing difficult.

Testing is rarely straightforward. Teams face multiple hurdles:

1. **Impossibility of Exhaustive Testing:** With millions of input combinations, you can never test everything. Deciding *what* to skip is a major challenge.
2. **Lack of Domain Knowledge:** Testers often understand the software technically but lack the business knowledge (e.g., banking rules) to test effectively.
3. **Unstable Environment:** If the test server keeps crashing or the database is inconsistent, testing results become unreliable.
4. **Regression Testing Overhead:** As the product grows, the number of test cases grows. Running all old tests for every new feature becomes time-consuming.

In mutation testing specifically, the large number of generated mutants poses a significant challenge. Even small programs can result in thousands of mutants, leading to increased execution time and resource consumption. Managing and analyzing these mutants requires robust tools and automation support.

Another challenge is the presence of equivalent mutants. These mutants behave identically to the original program for all possible inputs, making them impossible to kill. Identifying equivalent mutants is undecidable in general and often requires manual analysis, which increases effort and cost.

Time and budget constraints further limit the extent of testing. Organizations often prioritize delivery speed over thorough testing, leading to reduced test coverage. Additionally, lack of skilled testers and inadequate training can hinder effective testing.

Despite these challenges, structured approaches, automation, and risk-based prioritization can significantly improve testing outcomes.

8.3.2 Test Team Approach

An effective test team approach is essential for successful mutation testing. Testing should be treated as a collaborative activity involving testers, developers, and quality assurance professionals.

The test team should include members with strong analytical skills, domain knowledge, and familiarity with testing tools. Developers play a crucial role by writing unit tests and supporting mutation analysis.

NOTE

Testers focus on designing comprehensive test cases and interpreting mutation results.

The mindset and structure of the team determine success.

- **Destructive Approach:** The team must adopt a "break the system" attitude. If they are too gentle, the customers will be the ones finding the bugs.
- **Collaborative Approach:** While "destructive" in testing, they must be "constructive" in communication. The goal is to improve quality, not to humiliate developers.
- **Risk-Based Approach:** Since we cannot test everything, the team must prioritize tests based on Risk (Probability of Failure $\times$ Impact of Failure).

Clear role definition and communication are vital. Regular reviews, defect triage meetings, and feedback loops help align testing objectives with project goals. A collaborative test team approach fosters shared ownership of quality and improves overall testing effectiveness.

8.3.3 Process Problems Faced by Testing

Testing processes often face issues such as inadequate planning, unclear requirements, and insufficient resources. Poor requirement specifications lead to ambiguous test cases and missed defects. In mutation testing, lack of integration with existing test workflows can cause resistance from teams. Tool complexity and limited automation further complicate adoption.

Even with a good team, the *process* can fail.

- **"Throw it over the wall":** Developers finish coding and immediately start a new task, leaving testers with no support for the code currently being tested.
- **Late Involvement:** Testers are brought in only days before the deadline.
- **Poor Defect Reporting:** Testers report "It's not working," without providing logs, steps to reproduce, or screenshots, leading to a "Ping-Pong" game between Dev and QA.
- **Lack of Version Control:** Testing the wrong version of the build (e.g., testing v1.0 when v1.1 was already fixed).

Addressing these process problems requires management support, standardized procedures, and continuous improvement initiatives.

8.3.4 Cost Aspect of Testing

NOTE

Testing incurs direct and indirect costs, including tools, infrastructure, and personnel. Mutation testing is perceived as expensive due to computational demands. Testing is an investment.

- Cost of Conformance: The money spent on testing tools, salaries, and training.
- Cost of Non-Conformance: The money lost due to bugs (Customer support, refunds, brand damage).

The Rule of Ten: The cost to fix a defect increases by a factor of 10 at each stage of the lifecycle.

- Requirements: $1
- Coding: $10
- Testing: $100
- Production: $1,000

However, early defect detection reduces long-term costs by preventing failures in production. Cost-benefit analysis helps justify investment in advanced testing techniques.

8.3.5 Establishing Testing Policy

A testing policy defines organizational commitment to quality. It establishes objectives, standards, and responsibilities for testing activities, including advanced techniques like mutation testing. A well-defined policy ensures consistency, compliance, and continuous improvement across projects. A Testing Policy is the "Constitution" of the QA department. To establish one:

1. Define Objectives: What is the goal? (e.g., "Zero critical defects in production").
2. Define Scope: What do we test? (Functional, Security, Performance). What do we *not* test?
3. Define Standards: Which tools? Which bug tracking format? Which naming conventions?
4. Get Management Buy-in: Without executive support, the policy is just a piece of paper.
5. Communicate: Every developer and stakeholder must read and understand the policy.

Activity Title: Creating and Detecting Mutants

NOTE

Objective:
To understand the concept of mutation testing and evaluate the effectiveness of test cases.

Materials Required:

- Simple program/code
- Computer
- Testing notes

Procedure/Steps:

1. Select a simple program containing conditions or calculations.
2. Create small changes in the program, such as changing a relational operator.
3. Treat each modified version as a mutant.
4. Run the existing test cases against the mutants.
5. Identify which mutants are detected and which survive.
6. Discuss why additional test cases may be required for surviving mutants.

Outcome:
The learner will understand mutation testing and its role in evaluating the effectiveness of test cases.

8.3.6 Methods, Structured Approach to Testing

Structured testing approaches include black-box, white-box, and gray-box testing. Mutation testing complements these methods by strengthening test adequacy. Combining structured methods with automation and metrics-driven evaluation leads to higher software quality. Testing should not be ad-hoc. Common structured methods include:

1. Static Testing (Verification):

   Reviews: One person checks another's work.

   Walkthroughs: The author explains the code to the team.

   Inspections: Formal, rigorous review with a moderator and checklist.

2. Dynamic Testing (Validation):

   Structure-Based (White Box): Statement coverage, Branch coverage.

NOTE

Specification-Based (Black Box): Equivalence Partitioning, Boundary Value Analysis.

Experience-Based: Exploratory testing based on the tester's intuition.

Mutation Testing checks the testers, not just the code. It ensures test cases are capable of finding real bugs. Challenges include time constraints, unstable environments, and the sheer volume of regression tests. Process Problems often stem from poor communication or late involvement of the QA team. Cost: Testing saves money in the long run. The earlier a bug is found, the cheaper it is to fix. Policy: A clear policy aligns the entire organization on quality goals. Structured Testing: Combining Static (Reviews) and Dynamic (Execution) testing provides the best coverage.

Check Your Progress 8.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. Mutation Testing is mainly used to:
A) Reduce software size
B) Check the quality of test cases by introducing small changes in the program
C) Replace integration testing
D) Improve user interface design

_____

_____

Q2. In mutation testing, a good test case should “kill” the mutant.

_____

Q3. A modified version of the original program created in mutation testing is called a _____.

_____

Q4. Write any two challenges faced in mutation testing.

Answer (Any two):

High computational cost (too many mutants generated)

Time-consuming execution of test cases

Difficulty in handling equivalent mutants

Requires automation tools for practical use

_____

_____

Q5. Which of the following is a major cost aspect of mutation testing?
A) Mutant generation and execution time
B) Printing of test cases
C) Increasing number of computers in office
D) Reducing user training

_____

Fill-in-the-Blank Questions:

1. A modified version of a program used in mutation testing is called a _____.

2. The effectiveness of mutation testing is measured using the _____ score.

True/False Questions:

1. Mutation testing evaluates the quality of test cases.

2. Equivalent mutants can always be easily identified.

8.4 SUMMARY

This unit has provided an in-depth understanding of Mutation Testing as an advanced and rigorous technique within Software Quality Assurance. Unlike conventional testing methods that primarily focus on identifying defects in software, mutation testing emphasizes evaluating the effectiveness and adequacy of test cases. By deliberately introducing small faults, known as mutations, into the program and observing whether existing test cases can detect them, mutation testing offers a quantitative and objective measure of test quality.

The unit began by introducing the fundamental concepts of mutation testing, including mutants, mutation operators, mutation score, and equivalent mutants. It explained how mutation testing is grounded in the competent programmer hypothesis and the coupling effect, making it a reliable method for assessing the robustness of test suites. The process of generating mutants, executing test cases, and analyzing results was discussed as a systematic approach to strengthening testing practices.

Various challenges in testing, particularly in mutation testing, were examined, such as high computational cost, large numbers of mutants, identification of equivalent mutants, and limited time and resources. Despite these challenges, the unit highlighted that automation, tool support, and selective mutation techniques can significantly improve feasibility.

NOTE

The importance of a collaborative test team approach was emphasized, showing how coordination among testers, developers, and quality assurance professionals leads to better defect detection and improved software quality. Process-related problems faced during testing, including unclear requirements, inadequate planning, and resistance to advanced techniques, were also analyzed.

The unit further discussed the cost aspects of testing, explaining that although mutation testing may appear expensive initially, it ultimately reduces long-term costs by preventing failures in production and enhancing system reliability. Establishing a well-defined testing policy and adopting structured testing methods were presented as essential steps for integrating mutation testing into organizational practices. Overall, this unit demonstrates that mutation testing plays a crucial role in improving test effectiveness, strengthening software quality assurance processes, and increasing confidence in software reliability, particularly for complex and safety-critical systems.

8.5 GLOSSARY

• Mutation Testing: A fault-based software testing technique that evaluates the quality of test cases by introducing small changes (mutations) into the program code.

• Mutation: A deliberate syntactic change made to a program statement to simulate a potential programming fault.

• Mutant: A modified version of the original program created by applying a mutation operator.

• Mutation Operator: A predefined rule used to introduce specific types of changes in the program, such as replacing arithmetic or logical operators.

• Mutation Score: A quantitative measure of test effectiveness, calculated as the ratio of killed mutants to the total number of non-equivalent mutants.

• Killed Mutant: A mutant that produces different output from the original program when executed with a test case.

• Surviving Mutant: A mutant that is not detected by any test case, indicating weaknesses in the test suite.

• Equivalent Mutant: A mutant that behaves identically to the original program for all possible inputs and cannot be killed by any test case.

• Competent Programmer Hypothesis: The assumption that programmers usually write programs that are nearly correct, and faults are small deviations.

NOTE

• Coupling Effect: The concept that complex faults are combinations of simpler faults and can be detected by tests designed for simple faults.

8.6 ANSWER TO CHECK YOUR PROGRESS – POSSIBLE ANSWERS

Q1. Answer: B

Explanation: Mutation testing evaluates the effectiveness of test cases by creating "mutants" (small code changes) and checking whether tests detect them.

Q2. Answer: True

Explanation: If a test case detects the change and fails, it means the mutant is killed, showing the test case is strong.

Q3. Answer: Mutant

Explanation: Mutants are generated by applying small syntactic changes such as replacing operators or variables.

Q4. Explanation: Mutation testing is powerful but expensive because it creates many program variants and needs repeated execution.

Q5. Answer: A

Explanation: Mutation testing increases cost due to large numbers of mutants and repeated execution of test suites.

8.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3787	

8.8 REFERENCE

- Pressman, R. S., & Maxim, B. R. (2019). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill Education.
- Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education.

NOTE

- Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing* (3rd ed.). Wiley India.
- Ammann, P., & Offutt, J. (2016). *Introduction to Software Testing* (2nd ed.). Cambridge University Press.
- IEEE Computer Society. (2014). *Guide to the Software Engineering Body of Knowledge (SWEBOK)*.
- Zhu, H., Hall, P. A. V., & May, J. H. R. (1997). Software unit test coverage and adequacy. *ACM Computing Surveys*.

8.9 EXERCISE

Part–1

A. Multiple Choice Questions (MCQs)

(Five MCQs with Answers and Explanations after questions)

Q1. Mutation testing is mainly used to:

A) Improve hardware performance
B) Evaluate the effectiveness of test cases
C) Reduce documentation
D) Increase software size

Q2. A mutant is created by:

A) Removing all test cases
B) Introducing small changes in the source code
C) Increasing memory
D) Changing software requirements

Q3. If a test case detects a mutant, the mutant is said to be:

A) Alive
B) Dead
C) Valid
D) Ignored

Q4. The mutation score is calculated as:

A) Total mutants / killed mutants
B) Killed mutants / total mutants
C) Number of test cases / mutants
D) Mutants / test plan

Q5. Mutation testing is considered expensive because:

A) It reduces software quality
B) It requires many mutants and repeated test executions
C) It avoids automation
D) It cannot be applied to code

NOTE

Answers with explanation

1. **Answer: B**
   Explanation: Mutation testing checks whether test cases can detect small changes (mutants) introduced in the program.
2. **Answer: B**
   Explanation: Mutation testing generates mutants by making small syntactic changes in the program to simulate faults.
3. **Answer: B**
   Explanation: A mutant is "killed" when test cases detect the change and cause the test to fail.
4. **Answer: B**
   Explanation: Mutation score shows test suite strength and is the ratio of killed mutants to total mutants.
5. **Answer: B**
   Explanation: Mutation testing generates many mutants and requires running the test suite multiple times, increasing time and cost.

B. Small Answer Type Questions (5 Questions)

1. Define mutation testing.
2. What is meant by a mutant?
3. What does it mean to "kill a mutant"?
4. Define mutation score.
5. Write any two advantages of mutation testing.

Part–2

C. Long Answer Type Questions (200 words)

1. Explain mutation testing and its importance in software testing.
2. Describe the process of generating mutants and performing mutation testing.
3. Explain mutation score and discuss how it measures test effectiveness.
4. Discuss the advantages and limitations of mutation testing.
5. Explain how mutation testing helps in improving test case quality.

Part–3

D. Long Answer Type Questions (300 words)

1. Explain mutation testing in detail. Discuss its role in evaluating the strength of test cases.

NOTE

2. Describe the complete mutation testing process including mutant generation, execution, and analysis.
3. Discuss mutation operators and explain how they are used to create mutants in a program.
4. Explain the concept of mutation score in detail. Discuss its importance and interpretation.
5. Discuss the challenges of mutation testing and suggest methods to reduce its cost and improve its practical usability.

NOTE

UNIT–9
CATEGORIES OF DEFECT IN SOFTWARE TESTING

UNIT STRUCTURE

9.1 Introduction
9.2 Objective
9.3 Categories of Defect
  9.3.1 Defect, Error or Mistake in Software
  9.3.2 Developing Test Strategy
  9.3.3 Developing Testing Methodologies (Test Plan)
  9.3.4 Testing Process
  9.3.5 Attitude Towards Testing (Common People Issues)
  9.3.6 Test Methodologies/Approaches
  9.3.7 People Challenges in Software Testing
9.4 Summary
9.5 Glossary
9.6 Answer to check your progress
9.7 Suggested readings - OER
9.8 Reference
9.9 Exercise

9.1 INTRODUCTION

Software defects are an inevitable part of software development due to the complexity of systems, human involvement, evolving requirements, and technological constraints. A defect in software refers to any deviation of the software product from its expected behavior, specified requirements, or user expectations. Understanding the categories of defects is fundamental to effective software testing and Software Quality Assurance (SQA). It enables testers, developers, and managers to identify root causes, design appropriate test strategies, and improve overall software quality.

Defects can originate at any stage of the Software Development Life Cycle (SDLC), including requirements analysis, design, coding, integration, testing, deployment, and maintenance. A defect introduced early, if not detected promptly, may propagate into later stages, becoming more complex and costly to fix. Therefore, categorizing defects

NOTE

helps in early detection, efficient defect management, and prevention of recurrence.

In software testing, defects are not limited to coding errors alone. They may arise due to ambiguous requirements, incorrect assumptions, poor design decisions, communication gaps, inadequate testing practices, or human and organizational factors. Modern software systems involve multiple stakeholders, distributed teams, and rapid release cycles, which further increase the likelihood of defects.

Defect categorization provides a structured way to analyze failures and improve testing effectiveness. By classifying defects based on their nature, origin, and impact, testing teams can prioritize testing efforts, focus on high-risk areas, and allocate resources more efficiently. It also supports metrics-based quality assessment, enabling organizations to track defect trends and process maturity.

This unit discusses the various categories of defects in software testing and related aspects such as test strategy development, testing methodologies, testing processes, human attitudes towards testing, and people-related challenges. A comprehensive understanding of these concepts helps learners appreciate the multidimensional nature of software defects and equips them with the knowledge to manage quality effectively in real-world projects.

In the world of software engineering, "perfection" is an impossible goal. Software is built by humans, and humans make mistakes. This unit delves into the anatomy of these mistakes—defining exactly what constitutes an error, a defect, and a failure. Beyond the technical definitions, we will explore the strategic planning required to find these defects (Test Strategy and Planning) and the often-overlooked "human element" of testing: the attitudes, psychological challenges, and interpersonal dynamics that can make or break a QA team.

9.2 OBJECTIVE

After studying this unit, you should be able to:

- Distinguish clearly between an Error, a Defect, and a Failure.
- Categorize defects based on Severity and Priority.
- distinct between a high-level Test Strategy and a project-level Test Plan.
- Outline the core steps of the Testing Process.
- Analyze the psychological barriers and "people challenges" that arise between developers and testers.

NOTE

9.3 CATEGORIES OF DEFECT

In software testing, defects can be categorized to better understand their origin, impact, and preventive measures. Categorization of defects helps testing teams identify weak areas in the software development process and apply corrective actions systematically. Defects may arise during any phase of the Software Development Life Cycle (SDLC) and may be functional or non-functional in nature.

One major category is requirement defects, which occur due to unclear, ambiguous, incomplete, or incorrect requirements. These defects are particularly dangerous because they are introduced at an early stage and can propagate to later phases, increasing correction cost. Closely related are design defects, which occur when system architecture, data flow, algorithms, or interface designs are flawed. Poor design decisions often lead to performance issues, scalability problems, or maintainability challenges.

Coding defects are among the most visible and commonly identified defects. These include syntax errors, logical errors, incorrect calculations, improper data handling, and failure to handle boundary conditions. Interface defects occur when different modules, systems, or external interfaces fail to communicate correctly due to mismatched data formats, protocols, or timing issues.

Non-functional defects include performance defects, such as slow response time or poor throughput, and security defects, such as vulnerabilities that allow unauthorized access or data leakage. Usability defects affect user experience, making software difficult to understand or operate. Categorizing defects helps testers design targeted test cases and supports management in prioritizing defect resolution.

9.3.1 Defect, Error or Mistake in Software

In software engineering, the terms error, defect, and mistake are often used interchangeably, but they have distinct meanings. A mistake is a human action that produces an incorrect result. For example, a developer misunderstanding a requirement or a tester misinterpreting expected behavior commits a mistake.

An error is the incorrect internal state of the software that results from a mistake. Errors occur during activities such as requirement specification, design, or coding. For instance, an incorrect formula in the code represents an error.

While often used interchangeably in casual conversation, these terms have distinct technical meanings in Software Quality Assurance (according to IEEE and ISTQB standards).

1. **Error (Mistake):**

NOTE

Definition: A human action that produces an incorrect result.

Context: This happens in the developer's mind or action.

Example: A developer misunderstands a logic rule and types x > 10 instead of x >= 10.

2. **Defect (Bug/Fault):**

Definition: An imperfection or deficiency in a work product where it does not meet its requirements or specifications. It is the result of an error.

Context: This exists inside the code. It is dormant until executed.

Example: The line of code if (x > 10) exists in the software.

3. **Failure:**

Definition: The event in which a system or component does not perform a required function within specified limits.

Context: This is what the user sees. The defect is "executed" and causes a crash or wrong output.

Example: A user enters "10", expects the loop to run, but nothing happens.

Defect Categorization (Severity vs. Priority): Defects are usually categorized to determine the order of fixing.

- **Severity:** Impact on the system (Technical). "How bad is the crash?"
- **Priority:** Urgency of the fix (Business). "How soon do we need it fixed?"

High Severity, Low Priority: A crash on a legacy page nobody visits.

Low Severity, High Priority: A typo in the company logo on the homepage (doesn't break code, but looks unprofessional).

A defect (also known as a bug or fault) is the observable manifestation of an error in the software product. When the software behaves incorrectly during execution, the defect becomes visible.

Understanding this distinction is important for root cause analysis and defect prevention. By addressing mistakes and errors early, organizations can reduce the number of defects reaching later stages or production.

9.3.2 Developing Test Strategy

NOTE

Developing a test strategy is a critical managerial and technical activity in software testing. A test strategy defines the overall approach, principles, and framework under which testing activities are planned and executed. It acts as a guiding document that aligns testing objectives with business goals and quality expectations.

A comprehensive test strategy outlines the scope of testing, including what will be tested and what will not be tested. It specifies levels of testing such as unit testing, integration testing, system testing, and acceptance testing. It also defines the types of testing to be performed, such as functional, regression, performance, security, and usability testing.

Risk-based thinking is an essential element of test strategy development. By identifying high-risk areas, testers can prioritize testing efforts where defects are most likely to occur or have the greatest impact. The strategy also addresses test environments, test tools, automation approach, defect management procedures, and reporting mechanisms.

A Test Strategy is a high-level document, often creating the standard for the entire organization (or a large program). It does not change frequently.

- Goal: To define the "Approach" to testing.
- Key Components:

  Scope: What types of testing (Functional, Load, Security) apply to our products?

  Tools: Standardizing tools (e.g., "We will use Selenium for all web apps").

  Environment: Defining the standard test labs (e.g., "All testing must occur on Staging environments that mirror Production").

  Risk Analysis: How we determine what to test first.

A well-defined test strategy improves coordination among stakeholders, ensures consistent testing practices, and enhances defect detection efficiency. It also serves as a reference point throughout the project lifecycle, supporting quality assurance and continuous improvement.

Activity Title: Defect Classification Exercise

Objective:
To identify and classify different types of software defects.

Materials Required:

- Sample defect reports
- Notebook

NOTE

- Defect classification table

Procedure/Steps:

1. Collect or create at least ten examples of software defects.
2. Read each defect description carefully.
3. Classify defects into categories such as requirement, design, coding, interface, performance, and documentation defects.
4. Record the reason for each classification.
5. Discuss which defects could have been prevented earlier.

Outcome:
The learner will be able to classify software defects and understand their possible sources.

9.3.3 Developing Testing Methodologies (Test Plan)

Testing methodologies describe the structured and systematic techniques used to perform testing activities. A test plan is the central document that captures these methodologies and translates the test strategy into actionable steps. It provides detailed information on how testing will be conducted.

A test plan typically includes test objectives, test scope, test items, features to be tested, testing techniques, test cases, schedules, resources, roles and responsibilities, and entry and exit criteria. It also defines defect reporting and tracking mechanisms.

While the Strategy is static, the Test Plan is dynamic and specific to a project. It translates the strategy into action.

- Goal: To manage the testing activities for a specific release.
- Key Steps in Development:
    1. Analyze the Product: Review the SRS (Software Requirement Specification).
    2. Define Strategy for this Project: Will we use Automation or Manual testing for this specific release?
    3. Resource Planning: Who is available? Do they need training?
    4. Schedule Estimation: How many days do we need?
    5. Deliverables: What will we give the client? (Test Cases, Defect Reports, Test Summary Report).

Developing a robust test plan ensures that testing is organized, repeatable, and measurable. It enhances communication among team

NOTE

members and stakeholders and reduces ambiguity in testing activities. A well-prepared test plan contributes to early defect detection and better control over testing progress and quality.

9.3.4 Testing Process

The testing process consists of a sequence of activities such as requirement analysis, test design, test execution, defect reporting, and test closure. Each stage plays a critical role in identifying and managing defects. The standard testing process (STLC - Software Testing Life Cycle) follows a logical flow:

1. Requirement Analysis: Understanding what to test.
2. Test Planning: Defining the "How" and "When."
3. Test Case Development: Writing the step-by-step instructions.
4. Test Environment Setup: Preparing the servers and data.
5. Test Execution: Running the tests and logging bugs.
6. Test Cycle Closure: creating the final report and analyzing metrics.

A systematic testing process improves defect detection efficiency and reduces defect leakage into production. Metrics collected during the process help assess quality and process maturity.

9.3.5 Attitude Towards Testing (Common People Issues)

Human attitude towards testing significantly influences defect detection. Testing is sometimes perceived as a secondary or negative activity focused on finding faults. Such attitudes reduce the effectiveness of testing. A positive attitude that views testing as a quality-enabling activity encourages collaboration, early involvement, and proactive defect prevention. Testing is as much about psychology as it is about technology. A tester's attitude determines their success.

- **The "Constructive vs. Destructive" Conflict:**

  Developers are "Builders" (Constructive). They are emotionally attached to their code.

  Testers are "Breakers" (Destructive). Their job is to find flaws in that code.

  Issue: This natural opposition can lead to hostility if not managed well.

- **Confirmation Bias:**

  Developers testing their own code subconsciously avoid "dangerous" inputs because they want the code to work.

NOTE

Testers must have a "Professional Pessimism" attitude—assuming the system *will* fail and trying to prove it.

9.3.6 Test Methodologies / Approaches

Various test methodologies are used to detect defects, including black-box testing, white-box testing, gray-box testing, exploratory testing, and risk-based testing. Each approach addresses different defect categories. Different development models require different testing methodologies.

1. **Waterfall Model:**
    - Testing happens at the end (after coding).
    - *Pros:* Well documented.
    - *Cons:* Defects found late are expensive to fix.
2. **Agile / Scrum:**
    - Testing is continuous. Testers work alongside developers in 2-week sprints.
    - *Pros:* rapid feedback, early bug detection.
    - *Cons:* Documentation is often light; regression testing burden grows quickly.
3. **V-Model (Verification & Validation):**
    - For every development phase, there is a parallel testing phase (e.g., Requirements - Acceptance Testing).
    - *Pros:* Very disciplined.

Selecting appropriate methodologies improves coverage and defect detection effectiveness.

9.3.7 People Challenges in Software Testing

People-related challenges such as skill gaps, communication issues, time pressure, and resistance to change significantly affect testing outcomes. Managing these challenges requires training, clear roles, and organizational support. Beyond attitude, there are structural challenges involving people.

- Communication Gaps: Testers often report "It doesn't work" without technical details. Developers reply "It works on my machine." This "Ping-Pong" wastes time.
- Lack of Respect: In some organizations, testing is seen as a "lower" job than developing. This leads to low morale and high turnover in QA teams.
- Time Pressure: When projects run late, "Testing" is usually the phase that gets cut. Testers are forced to work weekends to make up for development delays.

NOTE

- Skill Gaps: Modern testing requires coding skills (for automation). "Manual-only" testers often struggle to keep up with agile teams.

Check Your Progress 9.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. A defect in software is also commonly called:
A) Patch
B) Bug
C) Compiler
D) Framework

Q2. An error is the human action that produces an incorrect result, while a defect is the result in the software.

Q3. A structured approach for executing testing activities is called a test _____.

Q4. Write any two common categories of defects found in software testing.

Answer (Any two):

Functional defects

Performance defects

Security defects

Interface defects

Usability defects

Compatibility defects

Q5. Which factor is MOST related to “people challenges” in testing?
A) Poor communication between developers and testers
B) Low RAM in system
C) Slow internet speed
D) Printer not working

Fill-in-the-Blank Questions

1. A human action that leads to an incorrect result is called a …….
2. A visible deviation of software behavior from expected results is known as a …….
3. A high-level document that defines the overall testing approach is called a …….

True or False

NOTE

All defects originate only during coding. –

Defect categorization helps improve testing effectiveness.

9.4 SUMMARY

This unit has provided an extensive discussion on the categories of defects in software testing and the various technical and human factors influencing defect occurrence and detection. The unit emphasized that defects are not limited to coding errors but can arise from requirements, design decisions, testing practices, and organizational behavior.

The concept of defect categorization was introduced as a structured approach to understanding and managing software quality. Different types of defects such as requirement, design, coding, interface, performance, security, and usability defects were explained to highlight their impact on software reliability and user satisfaction.

The unit also discussed the importance of developing effective test strategies and testing methodologies, which serve as foundational elements for systematic testing. A well-defined testing process ensures early defect detection, minimizes defect leakage, and supports continuous quality improvement.

Human aspects such as attitude towards testing and people-related challenges were recognized as critical contributors to testing success. The unit concluded that improving software quality requires not only technical competence but also positive collaboration, communication, and organizational support for testing activities.

This unit equips learners with conceptual clarity and practical insights necessary for effective defect management and quality assurance in real-world software projects.

9.5 GLOSSARY

• Defect: Any deviation of the software from its expected behavior, requirements, or user expectations.

• Error: An incorrect internal state of the software caused by a human mistake.

• Mistake: A human action that leads to an incorrect result during software development or testing.

• Test Strategy: A high-level document defining the overall approach and scope of testing.

• Test Plan: A detailed document describing testing objectives, methods, schedules, and responsibilities.

NOTE

• Defect Leakage: Defects that escape detection during testing and are found in later stages or production.

9.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3788	

9.8 REFERENCE

• Pressman, R. S., & Maxim, B. R. (2019). *Software Engineering: A Practitioner's Approach*. McGraw-Hill. • Sommerville, I. (2016). *Software Engineering*. Pearson Education.

• Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing*. Wiley.

• IEEE Computer Society. (2014). *SWEBOK Guide*.

• Ammann, P., & Offutt, J. (2016). *Introduction to Software Testing*. Cambridge University Press.

9.6 ANSWER TO CHECK YOUR PROGRESS – POSSIBLE ANSWERS

Q1. Answer: B
Explanation: A defect is commonly referred to as a bug, fault, or error that causes the software to behave incorrectly.

Q2. Answer: True
Explanation: An error/mistake is made by a person, and that mistake leads to a defect/fault in the software product.

Q3. Answer: Methodology
Explanation: A testing methodology includes test planning, designing, execution, reporting, and defect tracking.

Q4. Explanation: Defects can appear in different forms depending on the software behavior, performance, and user expectations.

Q5. Answer: A
Explanation: Many testing problems are caused by people issues such

NOTE

as poor communication, lack of cooperation, and negative attitude toward testing.

Fill-in-the-Blank Questions

mistake

defect

test strategy

True or False

False

True

9.9 EXERCISE

Part–1

A. Multiple Choice Questions (MCQs)

Q1. A defect is best defined as:

A) A new feature request
B) A deviation from expected behavior
C) A software license
D) A user manual

Q2. A defect introduced due to misunderstanding of requirements is called:

A) Coding defect
B) Requirement defect
C) Documentation defect
D) Performance defect

Q3. Errors related to user interface design come under:

A) GUI defects
B) Logical defects
C) Database defects
D) Network defects

Q4. Which defect category relates to slow response time?

A) Security defect
B) Performance defect
C) Documentation defect
D) Compatibility defect

Q5. Defects related to system crashes or failures are categorized as:

A) Reliability defects
B) Typing defects
C) Usability defects
D) Cosmetic defects

Answers with Explanation:

1. **Answer: B**
   Explanation: A defect is any difference between actual output and expected output of the software.
2. **Answer: B**
   Explanation: Requirement defects occur when requirements are unclear, incomplete, or wrongly interpreted.
3. **Answer: A**
   Explanation: GUI defects include problems in layout, buttons, navigation, fonts, and screen consistency.
4. **Answer: B**
   Explanation: Performance defects occur when the system fails to meet required speed, throughput, or response time.
5. **Answer: A**
   Explanation: Reliability defects affect stability and cause failures, crashes, or frequent breakdowns

B. Small Answer Type Questions (5 Questions)

(Answer in 2–4 lines)

1. Define defect in software testing.
2. Write any two categories of defects.
3. What is meant by requirement defect?
4. What is a GUI defect? Give one example.
5. What is the difference between a defect and a failure?

Part–2

C. Long Answer Type Questions (200 words each)

1. Explain the concept of defects in software testing and their impact on software quality.
2. Discuss different categories of defects in software testing with examples.
3. Explain requirement defects and design defects with suitable examples.
4. Discuss performance defects and reliability defects in software systems.

NOTE

5. Explain how defect categorization helps in improving software quality.

Part–3

D. Long Answer Type Questions (300 words each)

1. Explain categories of defects in software testing in detail with real-life software examples.
2. Discuss the causes of defects in different stages of SDLC and how they can be minimized.
3. Explain defect classification in detail and describe its importance in defect prevention and process improvement.
4. Discuss how different defect categories affect customer satisfaction and business reputation.
5. Explain the role of defect tracking and defect analysis in identifying root causes and improving software quality.

NOTE

UNIT–10
MANAGEMENT AWARENESS FOR TESTING

UNIT STRUCTURE

10.1 Introduction
10.2 Objective
10.3 Raising Management Awareness for Testing
  10.3.1 Skills Required by Tester
  10.3.2 Testing throughout the software life cycle
  10.3.3 Software development models
  10.3.4 Test levels, Test types,
  10.3.5 The targets of testing,
  10.3.6 Maintenance testing
10.4 Summary
10.5 Glossary
10.6 Answer to check your progress
10.7 Suggested readings - OER
10.8 Reference
10.9 Exercise

10.1 INTRODUCTION

Management awareness for testing refers to the level of understanding, commitment, and proactive involvement of management in software testing activities. In contemporary software development environments, testing is no longer an isolated technical activity performed at the end of development; instead, it is an integral component of Software Quality Assurance (SQA) that must be planned, supported, and monitored by management throughout the Software Development Life Cycle (SDLC).

Many software project failures can be traced not to a lack of technical capability, but to inadequate management attention to testing. Unrealistic schedules, insufficient allocation of resources, lack of trained testers, and absence of a testing culture often result in inadequate testing. Management awareness ensures that testing is viewed as a value-adding activity that reduces risk, improves reliability, and enhances customer satisfaction.

NOTE

From a strategic perspective, management is responsible for defining testing policies, approving test strategies, allocating budgets, selecting appropriate tools, and fostering coordination between development and testing teams. When management understands the cost of poor quality and the benefits of early and continuous testing, it becomes easier to integrate testing into planning and decision-making processes.

This unit focuses on raising management awareness for testing by explaining the skills required by testers, the importance of testing throughout the software life cycle, the impact of different software development models on testing, various test levels and types, testing targets, and the role of maintenance testing. The unit emphasizes the managerial viewpoint of testing and its contribution to achieving high-quality software systems

Software testing is often misunderstood by management as a simple "check" at the end of development. In reality, it is a complex, lifecycle-wide activity that requires specialized skills and strategic integration with development models. This unit focuses on how to raise management awareness regarding the value of testing, the specific skills required by modern testers, and how testing is structured across different levels and lifecycle phases (including maintenance).

10.2 OBJECTIVE

After studying this unit, learners will be able to:

- Explain the role of management in providing resources, policies, and direction for testing.
- Identify the skills required by software testers and the managerial responsibility in skill development.
- Assist the need for testing throughout the software development life cycle.
- Explain different software development models from a testing perspective.
- Distinguish between test levels and test types.
- Identify the targets of testing and their alignment with business goals.
- Analyse the significance of maintenance testing in long-term software quality.

NOTE

10.3 RAISING MANAGEMENT AWARENESS FOR TESTING

Raising management awareness for testing involves educating decision-makers about the importance, scope, and benefits of systematic testing. Managers often focus on cost, schedule, and delivery, while underestimating the risks associated with inadequate testing. Awareness can be raised through quality metrics, defect reports, case studies of software failures, audits, and regular communication between testing teams and management.

Management awareness ensures that testing is planned early, adequately resourced, and aligned with organizational objectives. It supports realistic scheduling, proper staffing, and investment in testing tools and environments. When management reviews testing metrics such as defect density, defect leakage, test coverage, and pass/fail rates, it can make informed decisions about release readiness and risk acceptance.

Management often views testing as a "Cost Center" a necessary evil that consumes time and budget. To change this perspective, we must frame testing as "Risk Management."

- **Value Proposition:** Testing provides information about the quality of the software, allowing management to make informed decisions (e.g., "Is this product safe to release?").
- **Cost of Quality:** Demonstrating that finding bugs early saves money (The 1-10-100 rule).
- **Reputation:** One major production bug can destroy brand trust (e.g., banking app failures).

A strong management commitment also helps create a quality-oriented culture in which testing is viewed as a preventive activity rather than a corrective one. This cultural shift encourages collaboration, accountability, and continuous improvement in testing practices.

10.3.1 Skills Required by Tester

Software testing requires a combination of technical, analytical, and interpersonal skills. From a management perspective, understanding these skill requirements is essential for building and sustaining effective testing teams. Technical skills include knowledge of programming languages, databases, operating systems, networks, and testing tools. Testers must also understand software architecture, requirements, and business processes.

Modern testing is a highly technical profession. The days of "just clicking around" are over.

NOTE

1. Technical Skills (Hard Skills):

- Database/SQL: Ability to query databases to verify if the data is saved correctly.
- Automation: Knowledge of scripting languages (Python, Java) and tools (Selenium, Appium).
- Linux/UNIX: Basic command-line skills to check server logs.
- API Testing: Understanding JSON, XML, and tools like Postman.

2. Soft Skills:

- Analytical Thinking: The ability to break down complex systems to find logical flaws.
- Communication: Clearly reporting bugs (steps to reproduce) without blaming developers.
- Curiosity: The desire to explore "what happens if I do this?"
- Domain Knowledge: Understanding the business rules (e.g., Banking, Healthcare) is often more valuable than coding skills.

Analytical skills are crucial for designing effective test cases, identifying edge conditions, and anticipating potential failure scenarios. Testers must be detail-oriented, logical, and capable of critical thinking. In addition, communication skills are vital for documenting defects clearly and interacting constructively with developers, managers, and stakeholders.

Management plays a significant role in ensuring continuous skill development through training programs, certifications, workshops, and mentoring. Investment in tester skill development directly improves defect detection capability and overall software quality.

10.3.2 Testing throughout the Software Life Cycle

Testing should be an integral activity throughout the software development life cycle rather than a phase performed after coding. Early testing activities such as requirement reviews, design inspections, and static analysis help detect defects before they propagate to later stages. From a management perspective, lifecycle-based testing reduces rework cost, improves schedule predictability, and enhances product quality. Continuous testing ensures that quality is built into the product from the beginning and aligns with modern development approaches such as Agile and DevOps.

Testing is not a phase; it is an activity that runs parallel to the entire lifecycle.

- Early Testing (Static): Reviews of requirements and design documents. This prevents defects before code is written.

NOTE

- Development Testing (Dynamic): Unit testing and Component testing during coding.
- System Testing (Dynamic): Verifying the full application.
- Release Testing (Dynamic): User Acceptance Testing (UAT).

Management support is essential for enabling early tester involvement, allocating time for reviews, and encouraging cross-functional collaboration. Lifecycle testing significantly reduces the cost of defect correction and increases customer confidence.

10.3.3 Software Development Models

Different software development models have different implications for testing. In the Waterfall model, testing is typically performed after development, increasing the risk of late defect discovery. The V-Model emphasizes the relationship between development phases and corresponding testing activities, highlighting the importance of early test planning.

The way testing is performed depends on the development model chosen.

1. Waterfall Model:

   Testing is a separate phase at the end.

   Risk: If critical bugs are found, it is too late/expensive to fix design flaws.

2. V-Model (Verification & Validation):

   Testing is integrated into every phase.

   Verification: "Are we building the product right?" (Reviews, Inspections).

   Validation: "Are we building the right product?" (Dynamic Testing).

3. Agile / Iterative Models:

   Testing is continuous. "Test-Driven Development" (TDD) may be used where tests are written *before* code.

   Feedback is immediate.

Iterative and Agile models promote continuous testing, frequent feedback, and close collaboration between developers and testers. From a management perspective, selecting an appropriate development model is critical for effective testing and risk management.

Activity Title: Preparing a Testing Awareness Plan

NOTE

Objective:
To understand the importance of management support and awareness in successful software testing.

Materials Required:

- Notebook
- Sample software project scenario

Procedure/Steps:

1. Consider a software project that has frequent defects and delayed releases.
2. Identify testing-related problems faced by the project team.
3. Identify the resources required for effective testing.
4. Prepare recommendations for management regarding time, budget, tools, and skilled testers.
5. Prepare a one-page testing awareness plan.

Outcome:
The learner will understand the role of management in supporting effective testing activities.

10.3.4 Test Levels and Test Types

Test levels represent different stages of testing, such as unit testing, integration testing, system testing, and acceptance testing. Test types refer to categories of testing performed to evaluate specific quality attributes, including functional, performance, security, usability, and regression testing.

It is crucial to distinguish between *Levels* (Scope of testing) and *Types* (Nature of testing).

A. Test Levels (The "Where"):

1. Unit Testing: Testing individual components (e.g., a single function or class). Done by Developers.
2. Integration Testing: Testing the interface between units (e.g., Module A Calling Module B). Done by Developers or Testers.
3. System Testing: Testing the complete, integrated system against requirements. Done by the Test Team.
4. Acceptance Testing (UAT): Testing by the end-user to verify business needs.

B. Test Types (The "What"):

NOTE

1. Functional Testing: "What the system *does*." (e.g., Login, Deposit Money).
2. Non-Functional Testing: "How the system *behaves*." (e.g., Performance, Security, Usability).
3. Structural Testing (White Box): Testing based on code logic coverage.
4. Change-Related Testing:
    - *Confirmation Testing (Re-testing):* Checking if a specific bug fix works.
    - *Regression Testing:* Checking if the fix broke anything else.

Management awareness of test levels and types helps in allocating responsibilities, resources, and schedules appropriately. Each level and type of testing contributes uniquely to software quality and risk reduction.

10.3.5 Targets of Testing

The targets of testing define what aspects of the software are to be evaluated. These targets include functional correctness, reliability, performance efficiency, security, usability, and compliance with standards. Clear definition of testing targets helps align testing activities with business objectives and risk priorities. What exactly are we testing? The target changes depending on the level.

- Code: Tested during Unit Testing.
- Architecture/Design: Tested during Integration.
- Requirements: Tested during System/Acceptance Testing.
- Documentation: User manuals and help guides are also targets of testing.

Management involvement is essential in prioritizing testing targets based on customer needs, regulatory requirements, and organizational goals.

10.3.6 Maintenance Testing

Maintenance testing is performed after software deployment to ensure that modifications, enhancements, or defect fixes do not adversely affect existing functionality. It includes regression testing, confirmation testing, and testing of new features. Testing doesn't stop after the first release. Software enters the "Maintenance Phase."

- Triggers for Maintenance Testing:
    1. Modifications: Planned enhancements (new features) or corrective changes (bug fixes).

NOTE

2. Migration: Moving software to a new platform (e.g., On-premise to Cloud).
3. Retirement: Archiving data when shutting down a system.

- Impact Analysis: The key challenge in maintenance is determining exactly what to test so you don't have to re-test the entire system (which is too expensive).

Management often underestimates the effort required for maintenance testing, despite the fact that maintenance accounts for a significant portion of software lifecycle costs. Proper planning, tool support, and resource allocation are essential for effective maintenance testing.

Check Your Progress

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. Management awareness in testing is important mainly because:
A) Testing is optional
B) Management allocates resources, budget, and schedule for testing
C) Testing is done only by customers
D) Testing is done only after deployment

_____

Q2. Testing should be performed throughout the software life cycle.

_____

Q3. The testing stages generally include unit testing, integration testing, system testing, and _____ testing.

_____

Q4. Write any two skills required for a good software tester.

_____

Q5. Maintenance testing is mainly performed when:
A) No changes are made in software
B) Changes, bug fixes, or enhancements are made after release
C) Software is only in design phase
D) Users stop using the system

_____

Fill-in-the-Blank Questions

1. Management awareness helps ensure adequate _____ and resources for testing.
2. Testing performed after deployment is called _____ testing.

True/False Questions

1. Testing is only a technical activity with no managerial involvement. –
2. Early testing reduces defect correction cost. –

NOTE

10.4 SUMMARY

This unit emphasized the importance of management awareness for software testing as a key factor in achieving high-quality software systems. It highlighted that effective testing requires not only technical expertise but also strong managerial support, informed decision-making, and a quality-oriented organizational culture.

The unit discussed tester skills, lifecycle-based testing, software development models, test levels and types, testing targets, and maintenance testing from a managerial perspective. By understanding and supporting these aspects, management can significantly enhance testing effectiveness and reduce software risks.

10.5 GLOSSARY

• **Management Awareness**: Understanding and support of testing activities by organizational leadership.
• **Test Level**: A specific stage of testing such as unit or system testing.
• **Test Type**: A category of testing focused on particular quality attributes.
• **Maintenance Testing**: Testing performed after software delivery to ensure continued correctness.
• **Quality Culture**: An organizational environment that prioritizes quality.

10.6 ANSWER TO CHECK YOUR PROGRESS – POSSIBLE ANSWERS

Q1. Answer: B

Explanation: Management support is essential for providing time, manpower, tools, and proper planning for effective testing.

Q2. Answer: True

Explanation: Testing is not a single phase activity; it should start from requirements and continue until maintenance.

Q3. Answer: Acceptance

Explanation: Acceptance testing is performed to confirm that the software is ready for customer use and meets business requirements.

Q4. Explanation: A tester must think critically, communicate clearly, and apply appropriate techniques to identify defects effectively.

Q5. Answer: B

Explanation: Maintenance testing ensures that modifications do not introduce new defects and the system continues to work properly.

NOTE

Fill-in-the-Blank Questions

1. Answer: planning
2. Answer: maintenance

True/False Questions

1. False
2. True

10.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3789	

10.8 REFERENCE

• Pressman, R. S., & Maxim, B. R. (2019). *Software Engineering: A Practitioner's Approach*. McGraw-Hill. • Sommerville, I. (2016). *Software Engineering*. Pearson Education.

• Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing*. Wiley.

• IEEE Computer Society. *SWEBOK Guide*.

10.9 EXERCISE

Part–1

A. Multiple Choice Questions (MCQs)

Q1. Defect Life Cycle mainly refers to:

A) Development of software only
B) Stages a defect passes through from detection to closure
C) Execution of test cases only
D) Designing user interface

Q2. The first stage in a typical defect life cycle is:

A) Closed
B) Assigned

NOTE

C) New / Logged
D) Verified

Q3. A defect is marked as "Rejected" when:

A) It is fixed successfully
B) It is not considered a valid defect
C) It is tested again
D) It is reopened after closure

Q4. A defect is marked "Reopened" when:

A) It is not tested
B) It fails again after being marked fixed
C) It is removed from the project
D) It is documented

Q5. Which of the following is NOT a defect status?

A) New
B) Assigned
C) Fixed
D) Compiled

Answers with Explanation:

1. Answer: B
Explanation: Defect Life Cycle is the sequence of states a defect goes through from identification to final closure.

2. Answer: C
Explanation: A defect is first identified and logged in the defect tracking tool as "New".

3. Answer: B
Explanation: Rejected defects occur when developers or managers decide the issue is not a bug or is not reproducible.

4. Answer: B
Explanation: If the defect still exists after a fix, testers reopen it for further correction.

5. Answer: D
Explanation: "Compiled" is not a defect status. Defect statuses include New, Assigned, Open, Fixed, Verified, Closed, etc.

B. Small Answer Type Questions (5 Questions)

(Answer in 2–4 lines)

1. Define defect life cycle.
2. Write any two defect statuses.
3. What is meant by "Verified" status?

NOTE

4. When is a defect marked “Closed”?
5. What is the meaning of “Reopened” defect?

Part–2

C. Long Answer Type Questions (200 words each)

1. Explain the defect life cycle and its importance in software testing.
2. Discuss various defect statuses in a typical defect life cycle.
3. Explain the process of defect logging, tracking, and reporting.
4. Describe the role of testers and developers in defect management.
5. Explain how defect life cycle helps in improving software quality.

Part–3

D. Long Answer Type Questions (300 words each)

1. Explain defect life cycle in detail with all major defect states and transitions.
2. Discuss the complete defect management process including defect reporting, tracking, verification, and closure.
3. Explain different defect statuses such as New, Open, Assigned, Fixed, Verified, Closed, Rejected, Deferred, and Reopened with examples.
4. Discuss the importance of defect life cycle in project management and software quality improvement.
5. Explain common challenges in defect tracking and suggest best practices for effective defect life cycle management.

Rabindranath TAGORE UNIVERSITY

Approved by : AICTE, NCTE, BCI, INC, MP PARAMEDICAL COUNCIL, DEB (UGC) | Recognized by : UGC | Member of : AIU, ACU

RABINDRANATH TAGORE UNIVERSITY (M.P.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org / for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

_____

_____

_____

***

BLOCK 3	**UNIT TESTING: BOUNDARY VALUE TESTING, EQUIVALENCE CLASS TESTING, DECISION TABLE–BASED TESTING, PATH TESTING, DATA FLOW TESTING**	**171-258**
Unit 11	Unit Testing	173
Unit 12	Software Quality Assurance and Testing	189
Unit 13	Decision Table-Based Testing	209
Unit 14	Path Testing	229
Unit 15	Data Flow Testing	243



Block III focuses on the systematic testing of individual software components to identify defects at an early stage of software development. It introduces important unit testing techniques used to assess the correctness, reliability, and quality of software. The block covers Boundary Value Testing and Equivalence Class Testing, which help testers select effective test cases from input conditions. It also explains the role of Software Quality Assurance and Testing in maintaining software standards. Decision Table-Based Testing is discussed for testing complex business rules and combinations of conditions, while Path Testing focuses on examining different execution paths in a program. Finally, Data Flow Testing helps identify errors related to the definition and use of data variables. Together, these techniques provide a structured approach to designing effective test cases and improving software quality.

NOTE

UNIT-11
UNIT TESTING

UNIT STRUCTURE

11.1 Introduction
11.2 Objective
11.3 Unit Testing: Boundary Value Testing
  11.3.1 Normal Boundary Value Testing
  11.3.2 Robust Boundary Value Testing
  11.3.3 Worst-Case Boundary Value Testing
  11.3.4 Special Value Testing
  11.3.5 Examples of Random Testing
  11.3.6 Guidelines for Boundary Value Testing
11.4 Summary
11.5 Glossary
11.6 Answer to check your progress
11.7 Suggested readings - OER
11.8 Reference
11.9 Exercise

11.1 INTRODUCTION

Software Quality Assurance (SQA) and Software Testing are critical activities in the software development life cycle. Among the various levels of testing, Unit Testing is the most fundamental and earliest testing activity performed on software. Unit testing focuses on verifying the correctness of individual units or components of a software system. A unit is the smallest testable part of a software program, such as a function, method, class, or module. The primary purpose of unit testing is to ensure that each unit performs exactly as intended according to its design and requirements.

In modern software development, systems are becoming increasingly complex, and software failures can lead to severe financial losses, safety risks, and reputational damage. Therefore, ensuring quality at the earliest stage is essential. Unit testing helps developers identify and fix defects early, reducing the cost and effort required to correct errors at later stages such as integration testing or system testing. Studies have shown that defects detected early are significantly cheaper to fix than those discovered after deployment.

NOTE

Unit Testing is the first level of dynamic testing, where individual components or modules of source code are tested to determine if they are fit for use. While developers often write these tests, the techniques used are rigorous. One of the most effective techniques for Unit Testing is Boundary Value Analysis (BVA).

History and experience show that software errors frequently occur at the "boundaries" of input domains rather than in the "center." For example, if a program accepts numbers between 1 and 100, it is much more likely to fail at 1, 100, 0, or 101 than it is to fail at 50. This unit explores the mathematical variations of this technique to ensure robust unit testing.

Unit testing is typically performed by developers themselves during the coding phase. It involves testing the internal logic of the code rather than its interaction with other modules. This level of testing is generally white-box in nature, meaning that the tester has full knowledge of the internal structure and implementation of the software. Techniques such as control flow testing, data flow testing, and boundary value testing are commonly used during unit testing.

One of the most important aspects of unit testing is the selection of appropriate test cases. Among various test design techniques, Boundary Value Testing (BVT) plays a crucial role. Many software defects occur at the boundaries of input ranges rather than in the middle. Boundary value testing focuses on testing the extreme ends of input values, such as minimum, maximum, and just beyond the valid range. This approach significantly increases the chances of detecting errors related to incorrect conditions, loops, and validations.

Unit testing also supports modern development practices such as Test-Driven Development (TDD) and Continuous Integration (CI). In TDD, developers write unit tests before writing the actual code, ensuring that the code meets the expected behavior from the beginning. Automated unit tests are integrated into CI pipelines to ensure that new changes do not break existing functionality.

The unit testing is the foundation of software quality. It improves code reliability, simplifies debugging, enhances maintainability, and builds confidence in the software product. Understanding unit testing and techniques like boundary value testing is essential for students, developers, and quality assurance professionals.

11.2 OBJECTIVE

After studying this unit, you should be able to:

- Define Boundary Value Analysis (BVA) and explain why boundaries are error-prone.

NOTE

- Differentiate between Normal, Robust, and Worst-Case boundary testing.
- Calculate the number of test cases required for each BVA variation.
- Apply Special Value Testing based on domain knowledge.
- Formulate guidelines for applying BVA effectively in unit tests.
- **Explain** the process and importance of unit testing.
- **Identify** suitable test conditions for individual software units.
- **Develop** basic test cases for testing individual modules or functions.
- **Evaluate** unit test results to identify defects.

11.3 UNIT TESTING: BOUNDARY VALUE TESTING

Boundary Value Testing (BVT) is a widely used test case design technique under unit testing. It is based on the observation that errors frequently occur at the boundaries of input ranges rather than within the valid range. Boundary value testing focuses on testing input values at their minimum, maximum, and just outside the allowed range.

In many software applications, inputs are restricted to specific ranges. For example, an age field may accept values between 18 and 60, or a password length may be limited between 8 and 16 characters. Developers often make mistakes while implementing conditions, loops, or validations related to these limits. Boundary value testing helps detect such errors effectively.

The core idea behind boundary value testing is that if a system works correctly at the boundaries, it is likely to work correctly for values within the range. Therefore, instead of testing all possible inputs, testers select a small but powerful set of test cases that include boundary values.

Boundary value testing is especially useful in:

- Numerical computations
- Loop conditions
- Input validation
- Array indexing
- Date and time calculations

This technique reduces the number of test cases while maintaining high defect detection capability. It is commonly used in combination with equivalence class partitioning. Boundary value testing can be classified

NOTE

into several types, such as normal boundary value testing, robust boundary value testing, worst-case boundary value testing, and special value testing. Each type focuses on different aspects of boundary conditions.

Boundary Value Testing is a data selection technique in which test data is chosen from the "edges" of the input equivalence classes.

The Core Variables:

For any input range $[min, max]$, we identify five standard values:

1. Min: The minimum valid value.
2. Min+: Just above the minimum (Min + 1).
3. Nom: A nominal (average) value in the middle.
4. Max-: Just below the maximum (Max - 1).
5. Max: The maximum valid value.

11.3.1 Normal Boundary Value Testing

Normal Boundary Value Testing is the most basic form of boundary value testing. It involves testing only the valid boundary values of the input domain. In this approach, test cases are designed using minimum, maximum, and just-inside boundary values, while excluding invalid inputs.

Normal BVA focuses on valid boundary values. It assumes the "Single Fault Assumption": failures are rarely caused by two or more variables being at their extreme positions simultaneously. Therefore, we hold all variables at their Nominal (Nom) value and vary only one variable at a time to its boundaries. For example, if an input field accepts values from 1 to 100, normal boundary value testing would include:

- Minimum value: 1
- Just above minimum: 2
- Maximum value: 100
- Just below maximum: 99
- A nominal (middle) value: 50

The objective of normal boundary value testing is to verify that the software behaves correctly for valid boundary conditions. It ensures that the system properly handles edge cases within the acceptable range.

This type of testing is simple to design and execute, making it suitable for early testing phases. It helps identify common programming errors such as incorrect comparison operators, off-by-one errors, and improper loop termination conditions.

NOTE

Normal boundary value testing is most effective when:

- Input ranges are clearly defined
- Invalid inputs are handled elsewhere
- The focus is on functional correctness

However, this approach does not test how the system handles invalid or out-of-range inputs. Therefore, it may not detect errors related to missing validations. Despite its limitations, normal boundary value testing is a powerful technique for improving software reliability and is widely used in unit testing practices.

11.3.2 Robust Boundary Value Testing

Robust Boundary Value Testing extends the concept of normal boundary value testing by including invalid boundary values. In addition to testing valid boundary values, this approach tests inputs just outside the valid range to evaluate the robustness of the software. Robust BVA extends the Normal approach by testing *invalid* values just outside the boundaries. The goal is to see if the system handles errors gracefully (Robustness).

For an input range of 1 to 100, robust boundary value testing includes:

- Minimum – 1 (0)
- Maximum + 1 (101)
- All normal boundary values

The purpose of robust boundary value testing is to ensure that the software can handle invalid inputs gracefully without crashing or producing incorrect results. It verifies error handling, validation logic, and exception management mechanisms.

This type of testing is particularly important in systems where incorrect input can lead to serious consequences, such as financial systems, medical software, or safety-critical applications. Robust boundary value testing helps identify:

- Missing validation checks
- Improper error messages
- Unhandled exceptions
- Security vulnerabilities

Compared to normal boundary value testing, robust testing provides better coverage and improves system reliability. However, it requires additional test cases and slightly more effort.

11.3.3 Worst-Case Boundary Value Testing

NOTE

Worst-Case Boundary Value Testing is an advanced form of boundary value testing that focuses on testing all possible combinations of boundary values for multiple input variables. While normal and robust boundary value testing usually consider one input variable at a time, worst-case boundary value testing considers interactions between multiple inputs, especially at their boundary limits.

In real-world software systems, functions often take more than one input. Errors may not only occur due to a single input reaching a boundary but also due to combinations of boundary values across different inputs. Worst-case boundary value testing addresses this issue by systematically testing all combinations of minimum, maximum, and nominal values for each input variable.

For example, consider a function that calculates the total bill amount based on:

- Number of items (1 to 10)
- Price per item (₹10 to ₹100)

In normal boundary value testing, each input would be tested independently. However, worst-case boundary value testing would include combinations such as:

- Minimum items with minimum price
- Minimum items with maximum price
- Maximum items with minimum price
- Maximum items with maximum price
- Nominal values combined with boundary values

The objective of worst-case boundary value testing is to identify defects caused by the interaction of multiple boundary conditions. This type of testing is particularly useful in complex systems where calculations depend on multiple parameters, such as banking software, scientific applications, and embedded systems.

Normal BVA assumes the "Single Fault Assumption" (only one variable is at a boundary at a time). However, in complex logic, bugs often appear when *multiple* variables interact at their extremes (e.g., low memory AND high CPU usage).

Worst-Case BVA rejects the single fault assumption. It tests the Cartesian product of the 5 standard values (Min, Min+, Nom, Max-, Max) for every variable.

- Test Cases Generated
- This number grows exponentially. If you have 5 variables, you need $5^5 = 3125$ test cases, which may be impractical.

NOTE

Example:

Function Area(Length, Width) where both are 1 to 10.

We test:

- (Min Length, Min Width)
- (Max Length, Min Width)
- (Max Length, Max Width)
- ...and every other combination.

Worst-case boundary value testing significantly increases the number of test cases. If a system has n input variables, and each variable has 5 boundary values (minimum, just above minimum, nominal, just below maximum, maximum), then the total number of test cases becomes 5^n. This can lead to a very large test suite.

Despite the increased effort, worst-case boundary value testing provides:

- High defect detection capability
- Better confidence in system reliability
- Thorough validation of boundary interactions

However, due to its complexity, it is usually applied selectively to critical modules rather than the entire system. Automation tools are often used to manage the large number of test cases. The, worst-case boundary value testing is a powerful technique that ensures software behaves correctly under extreme and combined boundary conditions. It is especially important in safety-critical and calculation-intensive applications.

11.3.4 Special Value Testing

Special Value Testing focuses on testing specific input values that are known to cause problems or have special significance in software systems. These values may not necessarily be boundary values but are often related to known risk areas in software development.

Special values include:

- Zero (0)
- Null or empty values
- Negative numbers
- Very large numbers
- Special characters
- Dates such as leap years (29 February)

NOTE

Many software failures occur due to improper handling of these special values. For example, division by zero can cause system crashes, null values can lead to runtime exceptions, and incorrect date handling can cause serious errors in financial or scheduling systems.

Sometimes, the mathematical boundaries are not enough. "Special Value Testing" relies on the tester's intuition and domain knowledge to test specific values that are known to be problematic for the specific business logic.

Examples:

- Zero (0): Often causes "Division by Zero" errors.
- Null / Empty: Passing a null string or empty list often breaks code.
- Leap Years: Testing Feb 29th.
- Currency: Testing negative numbers in financial fields.
- Powers of 2: In low-level systems, testing 255, 256, 65535, 65536.

The purpose of special value testing is to ensure that the software can handle unusual or unexpected inputs gracefully. It verifies that the system:

- Does not crash
- Provides meaningful error messages
- Maintains data integrity

Special value testing is especially important in:

- Financial systems
- Database applications
- Input validation modules
- Security-sensitive applications

For instance, a login system should correctly handle empty username and password fields. A banking application should correctly process zero balance accounts. A date-processing system should correctly handle leap years and month-end dates.

Special value testing complements boundary value testing by covering scenarios that boundary testing alone may miss. It is often based on tester experience, historical defect data, and domain knowledge. The special value testing enhances software robustness by ensuring that the system behaves correctly under exceptional and uncommon input conditions. It plays a vital role in improving overall software quality.

11.3.5 Examples of Random Testing

NOTE

Random Testing is a testing technique where test inputs are selected randomly from the input domain rather than systematically. Unlike boundary value testing, which focuses on specific values, random testing aims to simulate real-world usage by generating unpredictable input combinations.

The primary objective of random testing is to discover defects that structured testing techniques might overlook. It is based on the assumption that random inputs can expose unexpected behavior, especially in complex systems.

Random Testing is the opposite of BVA. Instead of selecting specific points at the edges, we select input values arbitrarily from the input domain.

- When to use: It is useful for reliability estimates and stress testing (e.g., "Fuzz Testing").
- Limitation: It is statistically unlikely to hit the error-prone boundaries. If the range is 1 to 1,000,000, a random generator will likely pick 584,231 (Nominal) and miss the bug at 1,000,001.

For example:

- Entering random strings in input fields
- Providing random numerical values within and outside valid ranges
- Randomly clicking buttons in a graphical user interface

Random testing is particularly useful for:

- Stress testing
- Performance testing
- Security testing

However, random testing has limitations. Since inputs are not carefully selected, some critical boundary cases may not be tested. Therefore, random testing is often used in combination with boundary value testing and equivalence class partitioning.

Advantages of random testing include:

- Simplicity
- Low cost
- Ability to detect unusual defects

Disadvantages include:

- Lack of systematic coverage
- Difficulty in reproducing defects

NOTE

- Inefficient test case selection

Despite its limitations, random testing is valuable when used appropriately, especially in large systems where exhaustive testing is impractical.

Activity Title: Developing Test Cases for a Unit

Objective:
To understand unit testing by preparing and executing test cases for an individual software component.

Materials Required:

- Simple program/module
- Computer
- Test-case template

Procedure/Steps:

1. Select a simple function, such as calculating the average of marks.
2. Identify inputs and expected outputs.
3. Prepare normal, boundary, and invalid input test cases.
4. Execute the test cases.
5. Compare actual results with expected results.
6. Record any failures.

Outcome:
The learner will be able to prepare and execute basic unit test cases.

11.3.6 Guidelines for Boundary Value Testing

Effective boundary value testing requires careful planning and systematic execution. To apply BVA effectively in Unit Testing:

1. Identify Variables: Clearly define the input variables and their ranges.
2. Apply to Output: Don't just look at Inputs. If a function returns a list of size 1 to 10, try to write a test that forces it to return 1 item and 10 items.
3. Assume Independence: Start by treating variables as independent (Normal BVA). Only move to Worst-Case BVA if the variables are known to interact heavily.
4. Floating Point: For floating-point numbers, boundaries are tricky. If Min = 5.0, then Min+ is not 6.0; it might be 5.0001 (depending on precision).

NOTE

5. Check Internal Boundaries: Look for "hidden" boundaries inside the code (e.g., loops that iterate i < 100 or array size limits).

The following guidelines help testers design effective boundary value test cases:

1. Clearly identify input ranges and constraints before designing test cases.
2. Test both minimum and maximum boundary values.
3. Include values just inside and just outside the valid range.
4. Consider multiple inputs and their interactions.
5. Combine boundary value testing with equivalence class partitioning.
6. Include special values such as zero, null, and negative numbers.
7. Automate boundary tests where possible.

NOTE

Check Your Progress 11.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. Boundary Value Testing (BVT) mainly focuses on:
A) Testing random values only
B) Testing values at the edges of input ranges
C) Testing database security
D) Testing only user interface

_____

Q2. Normal Boundary Value Testing includes testing values just below and just above the boundary.

_____

Q3. Testing input values outside the valid range is called _____ Boundary Value Testing.

_____

Q4. Write any two types of Boundary Value Testing.

_____

Answer (Any two):

Normal Boundary Value Testing

Robust Boundary Value Testing

Worst-Case Boundary Value Testing

Special Value Testing

Q5. Worst-case boundary value testing is mainly used to:
A) Test only one variable at a time
B) Test combinations of boundary values for multiple inputs
C) Avoid testing boundary conditions
D) Test only GUI

_____

11.4 SUMMARY

Unit testing is the foundation of software quality assurance, and boundary value testing plays a crucial role in detecting defects early in the development process. Various forms of boundary value testing, including normal, robust, worst-case, and special value testing, address different aspects of input validation and system behavior. By applying these techniques, developers can significantly improve software

reliability, reduce development costs, and ensure better user satisfaction.

11.5 GLOSSARY

- Unit Testing: Testing individual software components independently.
- Boundary Value Testing: Testing input values at their limits.
- Robust Testing: Testing system behavior with invalid inputs.
- Worst-Case Testing: Testing combinations of boundary values.
- Special Value: Inputs with unique significance such as zero or null.

11.6 ANSWERS TO CHECK YOUR PROGRESS

Q1. Answer: B

Explanation: Boundary Value Testing is based on the idea that defects often occur at boundary limits (minimum, maximum, just inside/outside).

Q2. Answer: False

Explanation: Normal BVT tests boundary values and values just inside the boundary. Testing outside the range is done in Robust BVT.

Q3. Answer: Robust

Explanation: Robust BVT checks how the system behaves for invalid values beyond the boundary limits.

Q4. Explanation: Different types of BVT are used depending on whether we test only valid boundaries or include extreme and invalid values.

Q5. Answer: B

Explanation: Worst-case BVT checks combinations of boundary values for multiple variables, increasing the chance of detecting complex defects.

11.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/User/UnitUrls?unitId=3790	

NOTE

11.8 REFERENCE

- IEEE Std 829 – Software Test Documentation
- ISTQB Foundation Level Syllabus
- Pressman, R. S., *Software Engineering*

11.9 EXERCISE

Part–1

A. Multiple Choice Questions (MCQs)

Q1. Defect reporting mainly involves:

A) Writing program code
B) Recording and communicating defects to developers
C) Increasing project cost
D) Preparing user manuals

Q2. A good defect report must include:

A) Only defect title
B) Only screenshot
C) Steps to reproduce and expected vs actual result
D) Developer name only

Q3. Severity in defect reporting indicates:

A) Who will fix the defect
B) How serious the defect is on the system
C) When defect was found
D) Number of test cases executed

Q4. Priority in defect reporting indicates:

A) The technical complexity of defect
B) The urgency to fix the defect
C) The number of modules
D) The number of developers

Q5. A defect is considered "Critical" when:

A) It affects font size
B) It causes system crash or major functionality failure
C) It affects spelling in documentation
D) It affects color theme only

Answers with Explanation:

1. **Answer: B**
   Explanation: Defect reporting is the process of documenting defects and communicating them clearly for fixing.

NOTE

2. **Answer: C**
   Explanation: Steps to reproduce, expected output, actual output, and evidence make defect reports clear and actionable.
3. **Answer: B**
   Explanation: Severity describes the impact of the defect on system functionality.
4. **Answer: B**
   Explanation: Priority defines how quickly a defect should be fixed based on business need.
5. **Answer: B**
   Explanation: Critical defects stop major functionality or cause system crashes, affecting usability and business operations.

B. Small Answer Type Questions (5 Questions)

1. Define defect reporting.
2. Write any two important fields in a defect report.
3. What is the difference between severity and priority?
4. What is meant by reproducibility in defect reporting?
5. Write any two characteristics of a good defect report.

Part–2

C. Long Answer Type Questions (200 words)

1. Explain the defect reporting process in software testing.
2. Discuss the key components of a defect report with suitable examples.
3. Explain severity levels and priority levels used in defect reporting.
4. Describe best practices for writing effective defect reports.
5. Discuss the role of defect reporting in improving software quality.

Part–3

D. Long Answer Type Questions (300 words)

1. Explain defect reporting in detail. Discuss how proper defect reporting improves communication between testers and developers.
2. Describe the structure of a defect report. Explain each component such as defect ID, title, environment, steps, expected result, actual result, and attachments.
3. Discuss severity and priority in detail. Explain how they influence defect fixing and project planning.
4. Explain common problems in defect reporting and suggest methods to overcome them.

NOTE

5. Discuss defect reporting tools and explain how defect reports are managed in modern Agile and DevOps environments.

NOTE

UNIT – 12
SOFTWARE QUALITY ASSURANCE AND TESTING

UNIT STRUCTURE

12.1 Introduction
12.2 Objective
12.3 Equivalence Class Testing
  12.3.1 Equivalence Classes
  12.3.2 Traditional Equivalence Class Testing
  12.3.3 Improved Equivalence Class Testing
  12.3.4 Edge Testing
  12.3.5 Guidelines and Observations.
12.4 Summary
12.5 Glossary
12.6 Answer to check your progress
12.7 Suggested readings - OER
12.8 Reference
12.9 Exercise

12.1 INTRODUCTION

Equivalence Class Testing (ECT) is one of the most important and widely used black-box test case design techniques in software testing. It is based on the idea that it is neither practical nor necessary to test every possible input value of a software system. Instead, inputs are divided into groups or classes that are expected to behave similarly. These groups are known as equivalence classes, and testing just one representative value from each class is considered sufficient.

In real-world software applications, the number of possible inputs can be extremely large or even infinite. For example, a text field may accept thousands of possible character combinations, or a numeric field may accept a wide range of values. Testing every input individually would require enormous time, effort, and cost. Equivalence class testing addresses this challenge by reducing the number of test cases while still maintaining good test coverage.

The basic assumption behind equivalence class testing is that if one value in a class causes an error, all other values in that class are likely

NOTE

to cause the same error, and if one value works correctly, the rest of the values in that class will also work correctly. This makes equivalence class testing an efficient and systematic testing technique.

Testing every possible input for a software program is usually impossible. For example, if a text field accepts numbers from 1 to 1,000, you cannot realistically write 1,000 test cases. To solve this, we rely on Equivalence Class Partitioning (ECP). This technique divides the input data into groups (or "classes") where the system is expected to behave similarly for all values in that group. If one value in the group works, we assume all values in that group work. This drastically reduces the number of test cases while maintaining high coverage.

Equivalence class testing is usually applied during unit testing, integration testing, and system testing, especially when the tester does not have access to the internal code. It is particularly useful for validating input conditions, business rules, and functional requirements.

This technique works very well when combined with boundary value testing, because while equivalence class testing focuses on general input ranges, boundary value testing focuses on edge values. Together, they provide strong coverage with fewer test cases.

In educational and academic contexts, equivalence class testing helps students understand:

- How to analyze input domains
- How to identify valid and invalid inputs
- How to design effective test cases logically

The equivalence class testing is a powerful, efficient, and easy-to-apply testing technique that improves software quality while reducing testing effort.

Software testing is a crucial activity in Software Quality Assurance because it ensures that a software system performs correctly and satisfies user requirements. However, complete testing of a software system is practically impossible. This is because the number of possible inputs, combinations, execution paths, and user scenarios can be extremely large. For example, even a simple input field that accepts values from 1 to 10,000 has 10,000 possible test inputs. If a system contains multiple such inputs, the number of combinations becomes very large. Testing every possible input and combination would require huge time, effort, and cost. Therefore, software testing requires intelligent techniques that reduce the number of test cases while still providing high defect detection capability.

NOTE

Equivalence Class Testing is one such important black-box test design technique. It is used to reduce the number of test cases by dividing the input domain into groups of values called equivalence classes. The basic idea is that values within the same equivalence class are expected to behave similarly and produce similar results. Therefore, instead of testing every value, testers select a representative value from each class. This approach makes testing more efficient without significantly reducing coverage.

Equivalence class testing is mainly used in functional testing, where the tester focuses on the input-output behaviour of the system rather than internal code. It is particularly useful for input validation, boundary checking, form validation, and business rules. For example, if a system accepts an age value between 18 and 60, then the input domain can be divided into valid and invalid classes. A tester can then select representative values such as 25 (valid), 17 (invalid), and 61 (invalid). This provides coverage of different conditions with fewer test cases.

The technique is based on the assumption that if one value in an equivalence class works correctly, then all other values in that class will also work correctly. Similarly, if one value fails, then all values in that class are likely to fail. Although this assumption is not always perfectly true, it is highly useful in practice because it reduces testing effort while maintaining strong fault detection. Equivalence class testing is often combined with other techniques such as Boundary Value Analysis and Decision Table Testing to achieve better coverage.

12.2 OBJECTIVE

After studying this unit, you should be able to:

- Define Equivalence Class Partitioning and its role in reducing test effort.
- Distinguish between Valid and Invalid Equivalence Classes.
- Apply Traditional and Improved approaches to designing test cases.
- Analyse the concept of Edge Testing in relation to equivalence classes.
- Formulate guidelines for identifying classes in various data scenarios (ranges, lists, booleans).

12.3 EQUIVALENCE CLASS TESTING

Equivalence Class Testing is a black-box testing technique used to reduce the number of test cases while maintaining adequate coverage.

NOTE

The technique divides the input domain into groups of values called equivalence classes. Each class represents a set of inputs that are expected to behave in the same way. A representative value is selected from each class to create test cases.

The purpose of equivalence class testing is to achieve maximum defect detection with minimum effort. Since testing every input is impossible, testers need a strategy to select test inputs intelligently. Equivalence class testing provides such a strategy by focusing on input groups rather than individual values.

This technique is widely used in input validation testing. Many defects occur because software does not handle invalid inputs correctly. Equivalence class testing ensures that both valid and invalid input conditions are tested. It also helps identify missing validation rules and incorrect handling of boundary conditions.

Equivalence class testing is based on the principle that the software should treat all values within a valid class the same way. Similarly, all values in an invalid class should be rejected or handled consistently. For example, if a password field requires at least 8 characters, then passwords of length 8 to maximum allowed form a valid class, and passwords shorter than 8 form an invalid class. Testing one value from each class provides coverage of both acceptance and rejection conditions.

In real systems, equivalence classes may be defined based on ranges, sets, formats, conditions, and business rules. For example, a date input may have classes such as valid dates, invalid dates, leap-year dates, and out-of-range dates. A numeric input may have classes such as positive numbers, negative numbers, zero, and non-numeric inputs. A dropdown field may have classes such as valid selections and invalid selections. Thus, equivalence classes help cover different types of inputs systematically.

Equivalence Class Testing is a test design technique in which the input domain of a program is divided into mutually exclusive and collectively exhaustive groups called equivalence classes. Each equivalence class represents a set of inputs that are treated the same by the software.

These classes are divided into:

- **Valid equivalence classes** – inputs that the system should accept
- **Invalid equivalence classes** – inputs that the system should reject

For example, if an application accepts marks between 0 and 100:

- Valid class: 0 to 100

NOTE

- Invalid classes: less than 0, greater than 100

Instead of testing all values from 0 to 100, a tester can select just one value, such as 50, to represent the valid class. Equivalence class testing is particularly effective for:

- Input validation
- Form processing
- Business rule verification
- Error handling

This technique helps testers identify missing validations, incorrect assumptions, and logic errors. It is simple to apply and can be used early in the testing process.

12.3.1 Equivalence Classes

An equivalence class is a group of input values that are expected to be treated the same way by the software. The main goal of defining equivalence classes is to reduce the number of test cases by selecting representative values from each group. Equivalence classes are usually divided into valid and invalid classes. A valid equivalence class contains inputs that satisfy the specified requirements and should be accepted by the system. An invalid equivalence class contains inputs that violate requirements and should be rejected or handled properly.

For example, consider an input field “Age” that accepts values from 18 to 60. The equivalence classes can be defined as follows. The valid class is age values between 18 and 60 inclusive. The invalid classes include values less than 18 and values greater than 60. In this case, a tester may choose representative values such as 25 for the valid class, 17 for the invalid class below range, and 61 for the invalid class above range. These values represent the classes and reduce test cases significantly.

Equivalence classes can be formed based on different types of input conditions. Range-based classes are common for numeric inputs. Set-based classes are used when inputs must belong to a specific set of values, such as selecting a country from a list. Format-based classes are used when inputs must follow a pattern, such as email format or phone number format. Condition-based classes are used when inputs depend on logical conditions, such as password strength rules.

Sometimes, equivalence classes may overlap or be complex. For example, a password may have multiple rules such as minimum length, uppercase requirement, special character requirement, and no spaces. Each rule can create valid and invalid classes. In such cases, testers must carefully define classes to cover all important conditions.

NOTE

Equivalence classes must be defined based on requirements. If requirements are unclear, equivalence class definition becomes difficult. Therefore, understanding requirements is essential for applying this technique effectively. An equivalence class is a set of input values that are expected to produce the same output or behavior from the software system. The idea is to partition the input domain into classes such that testing one value from the class is sufficient.

Equivalence classes are broadly categorized into:

1. Valid equivalence classes
2. Invalid equivalence classes

For example, consider a password field that requires 8–16 characters:

- Valid class: length between 8 and 16
- Invalid class 1: length less than 8
- Invalid class 2: length greater than 16

Each class represents a unique behavior of the system. Identifying equivalence classes requires careful analysis of:

- Functional requirements
- Input constraints
- Business rules

Equivalence classes must be:

- Disjoint (no overlap)
- Complete (cover all possible inputs)

Proper identification of equivalence classes leads to effective test case design and better defect detection.

An Equivalence Class is a subset of all possible inputs that leads to the same behavior in the software.

- Valid Equivalence Class: Represents valid inputs that the system should accept and process correctly.
- Invalid Equivalence Class: Represents invalid inputs that the system should reject (error handling).

Example: Consider a field Age that accepts values between 18 and 60.

1. Valid Class: 18 to 60 (System accepts). *We only need to test one number, e.g., 30.*
2. Invalid Class 1: Less than 18 (System rejects). *We test one number, e.g., 10.*

NOTE

3. Invalid Class 2: Greater than 60 (System rejects). *We test one number, e.g., 75.*

Instead of testing 42 valid numbers (18-60) and infinite invalid numbers, we now have just 3 test cases.

12.3.2 Traditional Equivalence Class Testing

Traditional equivalence class testing is the standard method used to apply equivalence class partitioning. It focuses on dividing input conditions into valid and invalid equivalence classes and selecting representative test cases from each class. In traditional equivalence class testing, the tester first identifies input conditions from requirements. Each input condition is analyzed to determine its valid range or set of acceptable values. Then, equivalence classes are formed for valid and invalid values. After that, test cases are designed by selecting one representative value from each class.

For example, if a field accepts values from 1 to 100, then the valid class is 1 to 100. Invalid classes include values less than 1 and values greater than 100. Traditional testing would select one value from each class such as 50, 0, and 101.

Traditional equivalence class testing works well when inputs are independent. It reduces test cases significantly and ensures that both valid and invalid inputs are tested. It is simple and widely applicable. It is also effective in detecting defects related to missing validation and incorrect handling of invalid inputs.

However, traditional equivalence class testing has limitations. It may not cover interactions between multiple input fields. For example, in a registration form, one field may depend on another. Traditional testing may test each field separately but may miss defects caused by combinations of inputs.

Another limitation is that selecting only one representative value from each class may miss defects within the class. The assumption that all values behave the same may not always hold. For example, a defect may occur only for certain values within the valid range due to internal calculations. Traditional equivalence class testing may not detect such defects if the chosen representative value does not trigger the issue.

Traditional equivalence class testing involves selecting one test case from each equivalence class. This approach assumes that all values within a class are treated identically.

The steps involved:

1. Identify input conditions
2. Divide inputs into valid and invalid equivalence classes

NOTE

3. Select one representative value from each class

For example, for age input (18–60):

- Valid class: 18–60 → test value: 30
- Invalid class: <18 → test value: 15
- Invalid class: >60 → test value: 65

Traditional equivalence class testing is simple and efficient but may miss defects that occur near boundaries. Therefore, it is often combined with boundary value testing. The traditional approach focuses on ensuring that every class is covered at least once.

Steps:

1. Identify Inputs: Identify the input variables (e.g., username, password).
2. Identify Classes: Identify the valid and invalid classes for each variable.
3. Assign IDs: Assign a unique ID to each class.
4. Write Test Cases:

   For Valid Classes: Create a new test case that covers as many *uncovered* valid classes as possible. (You can combine valid inputs for multiple variables in one test).

   For Invalid Classes: Create a new test case that covers one and only one invalid class. (Do not combine invalid inputs, or one error might mask another).

Despite its limitations, traditional equivalence class testing is widely used due to its simplicity and effectiveness in early testing stages.

Activity Title: Connecting SQA with Testing

Objective:
To understand the relationship between Software Quality Assurance and software testing.

Materials Required:

- Notebook
- Software project case study
- Reference material

Procedure/Steps:

1. Select a software development case study.
2. List its major SQA activities.

NOTE

3. Identify the testing activities performed at different stages.
4. Distinguish between defect prevention and defect detection activities.
5. Prepare a diagram showing how SQA and testing work together.

Outcome:
The learner will understand the complementary roles of SQA and software testing in achieving quality.

12.3.3 Improved Equivalence Class Testing

Improved equivalence class testing is an enhanced approach that addresses limitations of traditional equivalence class testing. It recognizes that modern software systems often involve multiple input conditions, complex constraints, and interactions between fields. Therefore, improved methods focus on better class selection and combination coverage.

One improvement is the use of multiple representative values from a class instead of only one. In traditional testing, one value is selected from each class. In improved testing, testers may select additional values within the same class, especially when the class is large or when risk is high. For example, if a range is 1 to 10,000, selecting only one value may not be sufficient. Selecting multiple values such as 100, 5000, and 9999 provides better coverage.

Another improvement is the combination of equivalence classes across multiple inputs. When a system has multiple input fields, each field has its own equivalence classes. Testing all combinations would be huge. Improved equivalence class testing uses techniques such as pairwise testing to cover combinations efficiently. Pairwise testing ensures that every pair of input conditions is tested at least once. This reduces the number of test cases while still covering interactions.

Improved testing also considers special cases and business rules. For example, in a discount calculation system, different customer types may have different discount rules. Equivalence classes must be defined based on business logic rather than only input ranges. Improved testing ensures that classes represent meaningful functional behaviour.

Improved equivalence class testing extends the traditional approach by selecting multiple representative values from each equivalence class instead of just one. This increases test coverage while still keeping the number of test cases manageable. Sometimes, the traditional method is too simple because it assumes all values in a class are *truly* equal.

However, specific values might still trigger unique bugs. Improved (or "Weak/Strong") forms of equivalence testing address this.

NOTE

1. Weak Normal Equivalence Class Testing:

   Based on the "Single Fault Assumption."

   We select one value from each valid class.

   If we have variables A (3 classes) and B (3 classes), we might only need 3 test cases total if we combine them efficiently.

2. Strong Normal Equivalence Class Testing:

   Based on the "Multiple Fault Assumption."

   We test the Cartesian product of all valid classes.

   Every class of variable A is combined with every class of variable B.

   Ensures that no specific combination of valid inputs causes a failure.

In this approach:

- More than one value is selected from large or critical classes
- Risk-prone classes receive more attention

For example, for marks input (0–100):

- Traditional: test 50
- Improved: test 10, 50, and 90

This method is particularly useful in:

- Safety-critical systems
- Financial applications
- High-risk modules

Improved equivalence class testing balances efficiency and thoroughness, making it suitable for complex systems.

12.3.4 Edge Testing

Edge testing focuses on values close to the boundaries of equivalence classes. Although similar to boundary value testing, edge testing emphasizes transitions between equivalence classes. Edge Testing is often a bridge between Equivalence Partitioning and Boundary Value Analysis.

Edge testing refers to testing values at or near the edges of equivalence classes. Many defects occur at the boundaries of input ranges because developers often make mistakes such as incorrect comparison operators, off-by-one errors, and incorrect handling of extreme values.

For example, consider a system that accepts input values from 1 to 100. The edges of the valid class are 1 and 100. The edges of invalid classes

NOTE

are 0 and 101. Edge testing would focus on values such as 0, 1, 2, 99, 100, and 101. These values help detect boundary-related defects.

Edge testing is important because boundaries separate valid and invalid classes. If the software incorrectly includes or excludes boundary values, it leads to defects. For example, if the requirement says age must be between 18 and 60 inclusive, but the developer mistakenly uses a condition greater than 18 instead of greater than or equal to 18, then age 18 will be rejected incorrectly. Edge testing would detect this defect.

Edge testing is also important for performance and stress conditions. For example, testing maximum allowed file size, maximum number of users, maximum transaction limit, and maximum input length helps identify system limitations. Many failures occur when systems are pushed near limits.

Edge testing is often considered part of boundary value analysis. However, in equivalence class testing, edge testing focuses on selecting values at class boundaries to represent class behaviour more effectively. It strengthens equivalence class testing by covering the most error-prone points.

While Equivalence Partitioning says "pick any value in the middle," Edge Testing argues that the "values at the edge" of the class are more significant.

- In pure Equivalence Class Testing, for range [1-100], choosing '50' is sufficient.
- In Edge Testing, even though 1 and 100 are technically in the same equivalence class as 50, we treat them as special "edge" cases because they are boundary definitions.
- Relevance: Many defects occur because the programmer defined the class incorrectly (e.g., using > instead of >=). Edge testing verifies the *boundaries* of the equivalence classes.

For example:

- Valid class: 1–100
- Invalid class: <1 and >100

Edge values include:

- 0, 1, 2
- 99, 100, 101

Edge testing helps detect off-by-one errors, incorrect comparisons, and validation issues. It is especially useful when equivalence classes are large.

12.3.5 Guidelines and Observations

NOTE

To apply equivalence class testing effectively, testers must follow certain guidelines and observations. These guidelines help ensure that equivalence classes are correctly identified and that test cases provide good coverage.

The first guideline is to understand requirements clearly. Equivalence classes must be based on requirements. If requirements are ambiguous, testers should clarify them before designing classes. Clear requirements lead to accurate class definition.

The second guideline is to identify both valid and invalid classes. Many testers focus only on valid inputs, but invalid inputs are equally important. Software must handle invalid inputs correctly by rejecting them or showing error messages. Testing invalid classes helps detect missing validation.

The third guideline is to define classes for each input condition. Each field or parameter should be analyzed separately. For numeric inputs, define range-based classes. For selection inputs, define set-based classes. For format inputs, define pattern-based classes.

Another important observation is that not all defects occur at boundaries. Some defects occur inside the range due to internal logic. Therefore, selecting additional values from large classes can be useful. This is especially important for critical systems.

Testers should also consider combinations of inputs. In many systems, defects occur due to interaction between fields. Therefore, improved equivalence class testing should include combination coverage such as pairwise testing.

Another guideline is to include edge values and boundary values. Many defects occur at boundaries, so selecting boundary values strengthens test design.

How do you identify the classes? Use these heuristic rules:

1. If input is a Range (e.g., 1-100):

   1 Valid Class (1 to 100).

   2 Invalid Classes (<1 and >100).

2. If input is a Specific Set (e.g., "Car", "Bus", "Truck"):

   1 Valid Class for each item (if handled differently) or 1 Valid Class for the group.

   1 Invalid Class (anything else, e.g., "Bike").

3. If input is a Boolean (True/False):

   1 Valid Class (True).

   1 Valid Class (False).

NOTE

4. If input is a Rule (e.g., "First character must be a letter"):

   1 Valid Class (Starts with letter).

   1 Invalid Class (Starts with number/symbol).

5. Don't combine Invalid Inputs: When testing invalid classes, test one at a time. If you input "Age: 10" (Invalid) and "Name: (Empty)" (Invalid), the system might throw the Age error and never check the Name. You won't know if the Name validation works.

Guidelines for effective equivalence class testing include:

- Identify all input conditions
- Define valid and invalid classes clearly
- Avoid overlapping classes
- Combine with boundary value testing
- Prioritize high-risk classes

Observations:

- ECT reduces test cases significantly
- Works best for input-driven systems
- Not suitable for complex internal logic alone

Check Your Progress 12.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. Equivalence Class Testing is mainly used to:
A) Test every possible input value
B) Divide input data into groups where test cases are treated as equivalent
C) Increase number of test cases
D) Test only output formatting

_____

Q2. In equivalence class testing, one test case from each class is usually sufficient.

_____

Q3. Equivalence classes are generally divided into _____ and invalid classes.

_____

Q4. Write any two advantages of Equivalence Class Testing.

NOTE

Answer (Any two):

Reduces the number of test cases

Saves time and cost in testing

Provides systematic coverage of input domain

Helps identify missing validation checks

Q5. Edge Testing in equivalence class testing is closely related to:
A) Boundary Value Testing
B) Mutation testing
C) Installation testing
D) Regression testing

12.4 SUMMARY

Equivalence class testing is an efficient black-box testing technique that reduces testing effort while maintaining adequate coverage. By grouping inputs into equivalence classes, testers can detect defects early and improve software quality. When combined with boundary value and edge testing, equivalence class testing becomes even more powerful.

This unit explained Equivalence Class Testing as an important black-box test design technique used to reduce test cases while maintaining strong coverage. The concept of equivalence classes was discussed, including valid and invalid classes. Traditional equivalence class testing was explained as a method that selects representative values from each class. Improved equivalence class testing was discussed as an enhanced approach that supports complex systems by using multiple representatives, combination testing, and risk-based selection. Edge testing was explained as a method that focuses on boundary values where defects commonly occur. Finally, guidelines and observations were presented to help apply equivalence class testing effectively in real software projects. Overall, equivalence class testing is essential for efficient and systematic software testing.

12.5 GLOSSARY

- Equivalence Class: Group of inputs with similar behavior
- Valid Input: Acceptable input as per requirements
- Invalid Input: Input outside acceptable range
- Black-Box Testing: Testing without knowledge of internal code

NOTE

12.6 ANSWERS TO CHECK YOUR PROGRESS

12.1

Q1. Answer: B

Explanation: Equivalence class testing reduces test cases by grouping inputs that are expected to behave similarly.

Q2. Answer: True

Explanation: Since values in the same class are assumed to behave similarly, one representative value can detect defects for that class.

Q3. Answer: Valid

Explanation: Valid classes contain acceptable inputs, while invalid classes contain unacceptable inputs that should be rejected.

Q4. Explanation: This technique improves efficiency by covering input ranges without testing every value.

Q5. Answer: A

Explanation: Edge testing focuses on boundary values, which is the same idea used in Boundary Value Testing.

12.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3791	

12.8 REFERENCE

- Pressman, R. S., *Software Engineering*
- IEEE Software Testing Standards
- Sommerville, I., *Software Engineering*

12.9 EXERCISE

A. Multiple Choice Questions (MCQs)

NOTE

1. Equivalence Class Testing is mainly a technique of:
   A) White-box testing
   B) Black-box testing
   C) Grey-box testing
   D) Structural testing

2. The main purpose of Equivalence Class Testing is to:
   A) Increase the number of test cases
   B) Test internal program logic
   C) Reduce the number of test cases while maintaining coverage
   D) Replace boundary value testing

3. Which of the following best defines an equivalence class?
   A) A single test case
   B) A group of test cases that test boundaries
   C) A set of input values expected to behave similarly
   D) A group of invalid inputs only

4. Which of the following is a valid equivalence class for an age field accepting values 18–60?
   A) Age = 10
   B) Age = 65
   C) Age = 30
   D) Age = –5

5. In Equivalence Class Testing, invalid equivalence classes are those that:
   A) Always produce correct output
   B) Represent acceptable input
   C) Lie outside the specified input range
   D) Are tested first

Explanations of MCQs

1. Equivalence Class Testing is mainly a technique of: Answer: B) Black-box testing

Explanation:
Equivalence Class Testing is a **black-box testing technique** because the tester focuses on the expected input and output of the software without examining its internal code or program logic. The input data is divided into different groups, called equivalence classes, and representative values from each group are tested.

2. The main purpose of Equivalence Class Testing is to: Answer: C) Reduce the number of test cases while maintaining coverage

Explanation:
Equivalence Class Testing helps reduce the number of test cases required for testing. Instead of testing every possible input value, similar

NOTE

inputs are grouped into equivalence classes. A representative value from each class is then tested, providing reasonable test coverage with fewer test cases.

3. Which of the following best defines an equivalence class?
Answer: C) A set of input values expected to behave similarly

Explanation:
An equivalence class is a **group of input values that are expected to produce similar results or follow the same behavior** in the software. If one representative value from the class works correctly, other values in the same class are generally expected to behave similarly.

4. Which of the following is a valid equivalence class for an age field accepting values 18–60?
Answer: C) Age = 30

Explanation:
If an age field accepts values from **18 to 60**, all values within this range are considered valid inputs. Since **30** lies within the specified range, it represents a valid equivalence class. Values such as 10, 65, and –5 are outside the accepted range and therefore belong to invalid equivalence classes.

5. In Equivalence Class Testing, invalid equivalence classes are those that:
Answer: C) Lie outside the specified input range

Explanation:
Invalid equivalence classes contain input values that **do not satisfy the specified requirements** of the software. For example, if an age field accepts values from 18 to 60, values below 18 or above 60 form invalid equivalence classes. Testing these classes helps determine whether the software properly handles incorrect or unacceptable inputs.

B. Short Answer Questions

1. Define Equivalence Class Testing.
2. What is an equivalence class?
3. Name the two types of equivalence classes.
4. Is Equivalence Class Testing a black-box or white-box technique?
5. What is the main benefit of equivalence class testing?

C. Long Answer Questions (200 words)

1. Explain valid and invalid equivalence classes with an example.
2. Why is equivalence class testing preferred over exhaustive testing?
3. Describe traditional equivalence class testing.

NOTE

4. What is edge testing? How is it related to equivalence class testing?
5. State any four guidelines for effective equivalence class testing.

D. Long Answer Questions (300 words)

1. Explain Equivalence Class Testing in detail with suitable examples.
2. Discuss the steps involved in identifying equivalence classes.
3. Compare traditional and improved equivalence class testing.
4. Explain the role of Equivalence Class Testing in Software Quality Assurance.
5. How does Equivalence Class Testing help in reducing testing cost and effort?

UNIT – 13
DECISION TABLE-BASED TESTING

UNIT STRUCTURE

13.1 Introduction
13.2 Objective
13.3 Decision Table–Based Testing
13.3.1 Decision Tables
13.3.2 Decision Table Techniques
13.3.3 Cause-and-Effect Graphing
13.3.4 Guidelines and Observations
13.4 Summary
13.5 Glossary
13.6 Answer to check your progress
13.7 Suggested readings - OER
13.8 Reference
13.9 Exercise

13.1 INTRODUCTION

In software testing, many systems do not behave based on a single input or a single condition. Instead, the system's output or action depends on a combination of conditions that work together. These conditions are usually derived from business rules, policies, regulations, or logical requirements. When the number of conditions increases, testing becomes complex and error-prone if it is done without a structured method. Decision Table-Based Testing is a powerful black-box testing technique that addresses this complexity in a systematic manner.

So far, we have looked at testing individual inputs (Boundary Value Analysis, Equivalence Partitioning). But what happens when inputs interact? For example, a bank might say: "If the customer is new AND the balance is < $1000, charge a fee. But if the customer is loyal OR the balance is > $5000, waive the fee."

Testing logic like this using simple partitions is difficult and prone to missing combinations. Decision Table-Based Testing is a systematic technique used to capture complex business logic where the system's

NOTE

output depends on a combination of different inputs (conditions). It ensures that every possible combination of inputs is considered.

Decision Table-Based Testing is especially useful for applications where decisions are made based on multiple conditions. For example, in a banking system, loan approval may depend on income, credit score, employment status, and repayment history. In an online admission system, eligibility may depend on marks, category, age, and document verification. In such cases, testing all possible combinations manually is difficult and may result in missing important scenarios. Decision tables provide a clear and organized way to represent all these combinations.

A decision table is a tabular representation of conditions and actions. It helps testers understand how different combinations of input conditions lead to specific system behaviors. Each column in a decision table represents a rule, and each rule corresponds to a test case. This ensures that all logical paths defined by the requirements are tested at least once.

Decision Table-Based Testing is classified as a black-box testing technique because it focuses on system behavior rather than internal code structure. Testers do not need to know how the program is written internally. They only need to understand the functional requirements and business logic of the system.

This technique is widely used in:

- Financial applications
- Insurance systems
- Payroll and taxation software
- E-commerce platforms
- University and examination management systems

Decision table testing improves test coverage, reduces ambiguity, and helps detect missing or inconsistent requirements. It also improves communication between testers, developers, and stakeholders because decision tables present logic in a simple and visual format.

In modern software development, where systems are becoming increasingly rule-driven and data-dependent, decision table-based testing plays a vital role in ensuring software correctness and reliability.

13.2 OBJECTIVE

The primary objective of Decision Table-Based Testing is to ensure that software systems behave correctly for all valid and invalid combinations

NOTE

of input conditions. This technique aims to systematically identify and test every logical decision defined in the system requirements.

After studying this unit, you should be able to:

- Explain the structure and components of a Decision Table.
- Construct a Decision Table to test complex logical requirements.
- Apply techniques to simplify decision tables (reducing redundant rules).
- Describe Cause-and-Effect Graphing and its relationship to decision tables.
- Identify scenarios where Decision Table testing is most effective.

One of the key objectives is to handle complexity effectively. When multiple conditions influence system behavior, it becomes difficult to test every combination using ad-hoc methods. Decision table-based testing provides a structured approach that ensures no important scenario is missed.

Another important objective is to improve test coverage. By representing conditions and actions in a table format, testers can easily verify whether all possible combinations have been considered. This leads to higher functional coverage and fewer defects escaping into later stages of development.

Decision table-based testing also aims to reduce ambiguity in requirements. Many times, requirements are written in natural language and may be unclear or incomplete. While creating a decision table, testers often discover missing rules, conflicting conditions, or undefined behaviors. This helps improve requirement quality early in the development process.

From a learning perspective, the objectives include:

- Developing analytical and logical thinking
- Understanding real-world business rule testing
- Learning systematic test case design

Another objective is to optimize testing effort. Instead of testing random combinations, decision tables help select meaningful and necessary test cases. This reduces redundancy while maintaining adequate coverage.

Decision table-based testing also supports traceability, as each test case can be directly traced back to a specific rule in the decision table. This is particularly useful during audits, reviews, and quality assurance processes.

NOTE

Overall, the objective is to ensure that decision-driven software systems behave reliably, accurately, and consistently under all defined conditions.

13.3 DECISION TABLE-BASED TESTING

Decision Table–Based Testing is an important black-box test design technique used to test systems where the output depends on different combinations of input conditions. In many real-world software applications, the behaviour of the system is not controlled by a single input value but by multiple conditions acting together. For example, in an online shopping system, the final payable amount may depend on whether the customer is a premium member, whether a discount coupon is applied, whether the cart value is above a certain limit, and whether the payment mode is cash or online. In such situations, testing each condition separately is not enough. The software must be tested for all meaningful combinations of conditions.

Decision table–based testing provides a systematic and structured way to represent complex business rules and to generate test cases from them. A decision table is essentially a tabular representation of conditions and corresponding actions. It helps testers ensure that no important rule is missed and that the software is tested for all valid combinations. This technique is highly useful when requirements are expressed in terms of rules such as “if–then–else” logic. It is especially applicable to domains such as banking, insurance, taxation, admission systems, billing systems, payroll systems, and access control systems.

The main strength of decision table–based testing is that it converts complicated logic into an organized table that is easy to understand and verify. It also helps in identifying missing requirements, contradictions in rules, and redundant conditions. For example, if a requirement says “a student is eligible for scholarship if marks are above 80 and family income is below 2 lakhs,” and another rule says “a student is eligible if marks are above 85 regardless of income,” decision tables help detect overlaps and priority conflicts.

Decision table testing also improves test coverage. Instead of randomly selecting test cases, testers derive test cases directly from the table, ensuring that each rule is tested at least once. This reduces the risk of leaving untested combinations, which is a common cause of defects in rule-based systems.

Another important advantage is that decision tables are not only useful for testers. They are equally helpful for business analysts, developers, and quality managers. They improve communication among stakeholders by representing rules clearly. In many cases, decision

NOTE

tables can also be used as documentation of business logic and as a basis for automated test generation.

Decision Table-Based Testing is a structured testing approach used to validate software systems where outputs depend on combinations of inputs. It is particularly effective when the system's logic is governed by multiple conditions that interact with each other.

In this technique, testers identify all relevant conditions that affect system behavior and list all possible actions the system can perform. These conditions and actions are then organized into a decision table. Each unique combination of conditions forms a rule, and each rule represents a test case.

A decision table generally consists of four main parts:

1. Condition Stub
2. Condition Entries
3. Action Stub
4. Action Entries

Decision tables can be simple or complex depending on the number of conditions involved. When conditions increase, the number of possible combinations also increases. Decision table-based testing ensures that these combinations are handled systematically rather than randomly.

This technique is widely used because:

- It ensures completeness
- It improves clarity
- It reduces testing gaps

Decision table-based testing is often used alongside other black-box techniques such as equivalence class testing and boundary value testing to provide comprehensive coverage.

13.3.1 Decision Tables

A decision table is a structured way of representing logic that depends on multiple conditions. It is typically presented in a tabular form consisting of two major parts: the condition part and the action part. The condition part lists all possible input conditions that affect decision-making. The action part lists all possible actions or outputs that the system may produce.

The table is divided into columns called rules. Each rule represents a specific combination of condition values and the corresponding action(s) to be taken. For each rule, the condition entries may be marked as true/false, yes/no, present/absent, or sometimes using specific values.

NOTE

The action entries indicate what action should occur when the conditions match that rule.

The basic structure of a decision table includes four quadrants. The top-left quadrant contains condition stubs, which describe the conditions. The top-right quadrant contains condition entries, which show possible values for each condition under each rule. The bottom-left quadrant contains action stubs, which describe possible actions. The bottom-right quadrant contains action entries, which show which action should be performed under each rule.

Decision tables are useful because they help handle complex logical conditions systematically. Many systems have requirements written in natural language, which may be ambiguous or incomplete. Decision tables help remove ambiguity by forcing the logic into a precise format.

For example, consider a simple rule-based system for loan approval. Conditions may include whether the applicant has a stable income, whether the applicant has good credit history, and whether the loan amount is within the allowed limit. Actions may include approve loan, reject loan, or request additional documents. Without a decision table, it is easy to miss certain combinations such as stable income but poor credit history. Decision tables ensure that all meaningful combinations are considered.

Decision tables also help in identifying missing rules. If a combination of conditions does not have an action specified, it indicates incomplete requirements. They also help detect contradictions. If two rules represent the same condition combination but different actions, then there is a conflict that must be resolved.

A decision table is a tabular representation of conditions and corresponding actions. It provides a clear and organized way to represent complex decision logic that may otherwise be difficult to understand. A Decision Table (also called a Cause-Effect Table) is a tabular representation of logic that links conditions (inputs) to actions (outputs). It is split into four quadrants.

	Rule 1	**Rule 2**	**Rule 3**	**...**
Conditions (Stub)	T	T	F	...
(Inputs)	T	F	T	...
Actions (Stub)	X		X	

- Condition Stub (Upper Left): Lists all the inputs or conditions (e.g., "Is User Logged In?", "Is Cart Empty?").

NOTE

- Condition Entry (Upper Right): The combinations of True/False (or specific values) for the conditions. Each column represents a unique "Rule."
- Action Stub (Lower Left): Lists all possible system actions (e.g., "Show Error", "Process Payment").
- Action Entry (Lower Right): Marks which action should occur for each specific rule (usually marked with an 'X').

The condition stub lists all conditions that influence system behavior. Each condition represents a factor that affects the decision. For example, “User is registered” or “Payment is successful”.

The condition entries specify the possible values of each condition, such as Yes/No or True/False.

The action stub lists all possible actions the system can perform. Actions represent outcomes such as “Approve request”, “Reject request”, or “Display error message”.

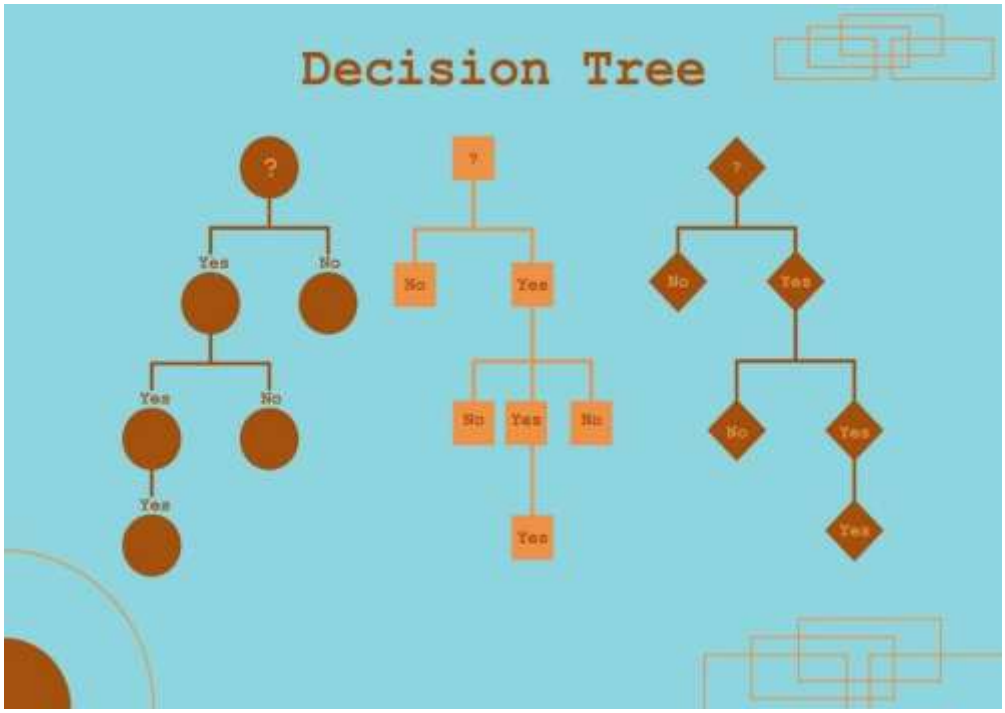


Fig. 13.1 : The action entries as Decision Tree

Each column of the decision table represents a rule, and each rule corresponds to a test case. This ensures systematic test design.

Decision tables help testers:

- Identify missing rules
- Detect contradictory requirements
- Simplify complex logic

NOTE

They are especially useful in business rule validation and compliance testing.

13.3.2 Decision Table Techniques

Decision table techniques refer to the systematic steps used to create decision tables and generate test cases from them. The technique ensures that testers cover all significant combinations of conditions and actions without missing critical cases. The process typically includes identifying conditions, identifying actions, constructing the table, simplifying it, and deriving test cases.

The first step is to identify all relevant conditions from requirements. Conditions are the factors that influence the decision or output. For example, in an ATM withdrawal system, conditions may include whether the card is valid, whether the PIN is correct, whether the account has sufficient balance, and whether the withdrawal amount is within daily limit.

The second step is to identify all possible actions. Actions represent the system outputs or results. In the ATM example, actions may include dispense cash, show error message, retain card, or block account.

The third step is to construct the decision table by listing conditions and actions. Each possible combination of condition values forms a rule. For a table with n conditions, if each condition has two values (yes/no), the maximum number of rules is 2^n. For example, with 3 conditions, there can be up to 8 rules. With 5 conditions, up to 32 rules. This shows that decision tables can become large quickly. Therefore, the technique includes simplification and reduction.

The fourth step is to fill condition entries and action entries. Each column represents a rule. The tester fills each rule with condition values and determines the action that should occur. This step requires careful interpretation of requirements and may involve consultation with stakeholders.

The fifth step is to check completeness and consistency. Completeness means every meaningful combination of conditions has a corresponding action. Consistency means there are no contradictions. This is one of the strongest benefits of decision table techniques because it reveals missing or conflicting requirements early.

The sixth step is to simplify the decision table. In many cases, some combinations are impossible or irrelevant. For example, a rule may represent “card invalid” and “PIN correct,” but if the card is invalid, the PIN may not even be checked. Such combinations can be marked as “don’t care” or removed. Simplification reduces test cases and improves clarity.

NOTE

The seventh step is to derive test cases. Each rule in the decision table becomes at least one test case. The test case includes input values corresponding to the condition combination and expected output corresponding to the action. Testers ensure that each rule is tested. If some rules are high risk, multiple test cases may be designed for that rule. Decision table techniques involve a step-by-step process of identifying conditions, defining actions, and constructing decision tables. The first step is to analyze requirements thoroughly to identify all relevant conditions and actions.

Next, possible values for each condition are determined. Conditions may be binary or multi-valued. Based on this, the tester decides whether to use a limited-entry or extended-entry decision table. Once the table is constructed, impossible or invalid combinations are eliminated. The remaining combinations are treated as valid rules and converted into test cases.

Building a decision table is a structured process.

Step 1: Identify Conditions

Determine how many input conditions (n) influence the decision.

- *Example:* A login screen has 2 conditions: (1) Valid Username, (2) Valid Password.

Step 2: Calculate Rules

The number of rules (columns) is typically 2^n (where n is the number of boolean conditions).

- *Example:* 2 conditions = 2^2 = 4 rules (TT, TF, FT, FF).

Step 3: Fill the Condition Matrix

Fill the upper-right quadrant with every possible combination of True (T) and False (F).

- Rule 1: T, T
- Rule 2: T, F
- Rule 3: F, T
- Rule 4: F, F

Step 4: Determine Actions

For each column (Rule), ask: "If these conditions are met, what should the system do?" Mark the corresponding Action row with an X.

Step 5: Simplify the Table (Rule Reduction)

Look for columns where the action is the same, and the value of one condition does not matter.

NOTE

- *Example:* If "Action A" happens regardless of whether Condition 2 is True or False (as long as Condition 1 is True), you can combine those two columns into one and mark Condition 2 with a dash ("-"), meaning "Don't Care."

Decision table techniques help:

- Reduce ambiguity
- Improve requirement understanding
- Optimize test coverage

They are especially effective in systems governed by policies, regulations, or decision logic.

Activity Title: Creating a Decision Table

Objective:
To develop a decision table and derive test cases from different conditions.

Materials Required:

- Notebook
- Pen
- Sample business rule

Procedure/Steps:

1. Consider a simple example such as an online shopping discount system.
2. Identify the relevant conditions.
3. Identify possible actions or outcomes.
4. Create a decision table showing different combinations of conditions.
5. Derive at least one test case for each important rule.
6. Check whether all significant combinations are covered.

Outcome:
The learner will be able to create decision tables and derive test cases systematically.

13.3.3 Cause-and-Effect Graphing

Cause-and-Effect Graphing is a technique used to visually represent the relationship between input conditions (causes) and outputs (effects). It is often used as a precursor to decision table-based testing. Cause-

NOTE

and-effect graphing is a test design technique closely related to decision table testing. It is used to represent the logical relationships between input conditions (causes) and outputs or system actions (effects). The technique helps in identifying combinations of causes that produce specific effects and then converting them into decision tables for generating test cases.

In cause-and-effect graphing, causes represent input conditions such as user actions, system states, or input values. Effects represent outputs, actions, or results produced by the system. The relationship between causes and effects is represented using a graph. The graph uses logical operators such as AND, OR, and NOT to show how causes combine to produce effects.

For example, consider a login system. Causes may include "username entered," "password entered," and "credentials valid." Effects may include "login success" or "error message displayed." The graph can represent that login success occurs only when all causes are true. Similarly, an error occurs if credentials are invalid.

Cause-and-effect graphing is useful because many requirements involve complex logical conditions. When requirements are written in plain language, they may be difficult to interpret. Cause-and-effect graphs help convert requirements into a clear logical structure. This reduces ambiguity and ensures completeness.

Another advantage is that cause-and-effect graphs help detect missing conditions. If an effect occurs without any cause, it indicates missing logic. If a cause is defined but does not lead to any effect, it indicates unused conditions. This helps in improving requirement quality.

In this technique, causes and effects are identified from requirements and connected using logical operators such as AND, OR, and NOT. The resulting graph is then converted into a decision table.

This approach helps identify:

- Missing conditions
- Redundant logic
- Logical inconsistencies

Cause-and-effect graphing is particularly useful when requirements are complex and interdependent. Cause-and-Effect (C-E) Graphing is a technique often used *before* creating the decision table to help visualize the logic. It maps inputs (Causes) to outputs (Effects) using boolean logic diagramming.

Components:

1. Nodes: Represent causes and effects.

NOTE

2. Links: Relationships using standard logic gates:
    - Identity: If A is 1, B is 1.
    - NOT (~): If A is 1, B is 0.
    - OR (V): If A or B or C is 1, Effect is 1.
    - AND (^): If both A and B are 1, Effect is 1.

Constraints:

Sometimes certain combinations are impossible (e.g., a person cannot be both "Male" and "Female" simultaneously). C-E graphing uses symbols (like 'E' for Exclusive) to mark these constraints so you don't generate impossible test cases.

Process:

1. Identify Causes (Inputs) and assign IDs (c_1, c_2.....).
2. Identify Effects (Outputs) and assign IDs (e_1, e_2.....).
3. Draw the graph linking causes to effects using logic gates.
4. Convert the graph into a Decision Table.

13.3.4 Guidelines and Observations

Decision table–based testing is highly effective, but its success depends on how correctly decision tables are constructed and applied. Therefore, testers must follow certain guidelines and observations to ensure good results.

The first guideline is to clearly identify conditions and actions from requirements. Conditions must represent factors that truly affect system behaviour. Actions must represent observable outputs or system responses. If conditions are missing or incorrect, the decision table will be incomplete and test cases will not provide adequate coverage.

The second guideline is to keep decision tables manageable. Decision tables grow exponentially as conditions increase. Therefore, testers should avoid including unnecessary conditions. Only conditions that influence decisions should be included. If too many conditions exist, testers should break the system into smaller parts and create separate decision tables.

The third guideline is to ensure completeness. Every meaningful combination of conditions should have a defined action. If a combination does not have an action, it indicates missing requirements. Decision tables are excellent for identifying such gaps.

he fourth guideline is to ensure consistency. There should be no contradictory rules. If two rules represent the same condition

NOTE

combination but different actions, then requirements conflict and must be resolved.

Another important observation is the use of “don’t care” conditions. Sometimes, an action depends only on certain conditions and other conditions do not matter. In such cases, testers can mark those conditions as “don’t care.” This helps reduce rules and simplifies the table.

Testers should also identify impossible combinations. Some combinations of conditions may not occur in real life. These should be removed from the decision table. For example, if a system checks PIN only after validating card, then combinations involving invalid card and correct PIN are irrelevant.

Some important guidelines for decision table-based testing include identifying all relevant conditions, keeping tables simple, eliminating impossible combinations, and validating tables with stakeholders. Observations show that decision table-based testing is highly effective for business rule testing but may become complex for systems with a very large number of conditions. Therefore, it should be combined with other techniques when necessary.

When to use Decision Tables:

- When the system behavior depends on complex rules or combinations of inputs.
- When there are logical dependencies (e.g., "If A is True, then B must be False").
- When you suspect the specifications are missing cases (Decision tables make it obvious if a column is missing).

Guidelines:

1. Completeness: Ensure all 2^n combinations are initially listed to avoid missing "silent" rules.
2. Impossible Rules: If a combination is physically impossible (e.g., OS is Mac AND OS is Windows), mark the Action as "Impossible" or "N/A" rather than deleting the column immediately.
3. Large Tables: If you have 10 conditions, you will have $2^{10} = 1024$ rules. This is unmanageable. Break large tables into smaller, linked tables.
4. Order: The order of conditions in the list usually doesn't matter, but grouping related conditions helps readability.

Check Your Progress 13.1

NOTE

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Question 1: What is Decision Table-Based Testing?

_____

_____

Question 2: Why is decision table-based testing important?

_____

_____

Question 3: What are the main components of a decision table?

_____

_____

Question 4: What is a rule in a decision table?

_____

_____

Question 5: What is cause-and-effect graphing?

_____

_____

Q1. Decision Table–Based Testing is most useful when:
A) There is only one input condition
B) System behavior depends on combinations of conditions
C) No rules are involved
D) Only GUI testing is required

Q2. A decision table represents conditions and actions in a tabular form.

_____

Q3. In a decision table, the part that lists possible outputs is called the _____ section.

Q4. Write any two advantages of decision table testing (answer any two).

Ensures coverage of all meaningful condition combinations

Helps detect missing rules or incorrect business logic

Easy to understand and review

Useful for requirement validation

NOTE

Q5. Cause-and-Effect Graphing is mainly used to:
A) Draw program flowcharts
B) Convert logical conditions into a graph and derive test cases
C) Replace coding
D) Improve computer speed

13.4 SUMMARY

Decision Table-Based Testing provides a structured approach to testing complex decision logic. By representing conditions and actions systematically, it ensures complete coverage, reduces ambiguity, and improves software quality. It is especially valuable in rule-based systems and works best when combined with other black-box testing techniques.

Decision Table–Based Testing is a black-box testing technique used for systems where outputs depend on combinations of multiple conditions. It provides a structured way to represent business rules and derive test cases systematically. Decision tables were explained as tabular representations of conditions and actions, ensuring clarity, completeness, and consistency. Decision table techniques were discussed as step-by-step methods to identify conditions, actions, build tables, simplify them, and derive test cases. Cause-and-effect graphing was explained as a related technique that models logical relationships between causes and effects and helps in generating decision tables. Guidelines and observations were presented to ensure effective application, including managing table size, ensuring completeness and consistency, using don't care conditions, eliminating impossible combinations, and prioritizing high-risk rules. Overall, decision table–based testing is essential for testing complex rule-driven systems efficiently and effectively.

13.5 GLOSSARY

- **Decision Table :** A decision table is a tabular representation of conditions and actions. It shows how different combinations of input conditions lead to different outputs or system actions. Decision tables help testers systematically identify and test all possible decision rules in a system.
- **Decision Table-Based Testing :** Decision Table-Based Testing is a black-box testing technique used to test software systems that depend on multiple conditions. Test cases are derived from

NOTE

decision tables to ensure that all combinations of conditions are tested.

- **Condition :** A condition is an input or a factor that affects system behavior. Examples include user age, account balance, eligibility status, or payment confirmation.
- **Condition Stub :** The condition stub is the part of the decision table that lists all conditions influencing the decision. It appears on the left side of the table.
- **Condition Entry :** Condition entries specify possible values of conditions, such as Yes/No, True/False, or specific ranges.
- **Action :** An action is the output or response of the system based on conditions. Examples include approving a request, rejecting an application, or displaying an error message.
- **Action Stub :** The action stub lists all possible actions that the system can perform.
- **Action Entry :** Action entries indicate whether a particular action should be executed for a specific rule.
- **Rule :** A rule is a unique combination of condition values that results in one or more actions. Each column of a decision table represents a rule and corresponds to a test case.
- **Cause :** A cause is an input condition that leads to an effect. Causes are identified during cause-and-effect graphing.
- **Effect :** An effect is the output or action produced by the system as a result of causes.
- **Cause-and-Effect Graphing :** Cause-and-effect graphing is a technique used to identify logical relationships between causes and effects. It helps convert complex requirements into decision tables.
- **Black-Box Testing :** Black-box testing is a testing technique where the internal structure of the program is not known. Testing is based on inputs and expected outputs.
- **Limited-Entry Decision Table :** A limited-entry decision table uses conditions that have only two possible values, such as Yes or No.
- **Extended-Entry Decision Table :** An extended-entry decision table is used when conditions can have more than two possible values.

NOTE

13.6 ANSWERS TO CHECK YOUR PROGRESS

Answer 1 : Decision Table-Based Testing is a black-box testing technique used to test software systems where outputs depend on multiple input conditions. It uses a decision table to represent all combinations of conditions and corresponding actions. Each combination is treated as a rule and converted into a test case. This technique ensures complete coverage of decision logic.

Answer 2 : Decision table-based testing is important because it helps test complex business rules systematically. It reduces ambiguity, improves test coverage, and ensures that no logical combinations are missed. It is especially useful in applications such as banking, insurance, and admission systems.

Answer 3 : The main components are condition stub, condition entries, action stub, and action entries. Together, these components represent all decision rules in a structured manner.

Answer 4 : A rule is a unique combination of condition values that leads to a specific action or set of actions. Each rule corresponds to a test case.

Answer 5 : Cause-and-effect graphing is a technique used to identify relationships between input conditions (causes) and outputs (effects). It helps derive decision tables from complex requirements.

These answers help learners evaluate their progress and identify areas that require further study.

Q1. Answer: B

Explanation: Decision tables are ideal for testing complex logic where output depends on multiple combinations of conditions.

Q2. Answer: True

Explanation: Decision tables clearly show all conditions, possible combinations, and corresponding actions.

Q3. Answer: Action

Explanation: Decision tables have two main parts: condition entries and action entries. Actions describe the system response.

Q4. Answer (Any two):

Explanation: Decision table testing improves accuracy in rule-based systems by checking all combinations systematically.

Q5. Answer: B

NOTE

Explanation: Cause-and-effect graphing models logical relationships between inputs (causes) and outputs (effects), helping generate effective test cases.

13.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3792	

13.8 REFERENCE

- R. S., "Software Engineering: A Practitioner's Approach", which explains software testing techniques in detail.
- Another important reference is Sommerville, I., "Software Engineering", which discusses black-box testing and requirement-based testing approaches.
- Myers, G. J., "The Art of Software Testing" is a classic text that introduces decision table-based testing and its importance in detecting logical errors.
- IEEE standards and publications also provide guidelines and best practices related to software testing and quality assurance.

13.9 EXERCISE

Part–1

A. Multiple Choice Questions (MCQs)

(Five MCQs with Answers and Explanations after questions)

Q1. Static testing is performed:

A) By executing the program
B) Without executing the program
C) Only after deployment
D) Only during maintenance

Q2. Which of the following is an example of static testing?

NOTE

A) Unit testing
B) System testing
C) Code review
D) Load testing

.Q3. The main objective of static testing is to:

A) Find defects early in SDLC
B) Increase program speed
C) Reduce documentation
D) Replace dynamic testing

Q4. Walkthrough is conducted mainly by:

A) Only customers
B) Author of the document with team members
C) Only testers
D) Only project manager

.Q5. Inspection is considered:

A) Informal review
B) Formal review technique
C) Dynamic testing method
D) Load testing method

Answers with Explanation

1. **Answer: B**
   Explanation: Static testing evaluates documents, code, and requirements without running the software.
2. **Answer: C**
   Explanation: Code review is a static testing method where code is checked without execution
3. **Answer: A**
   Explanation: Static testing helps detect defects in early phases, reducing cost and improving quality.
4. **Answer: B**
   Explanation: Walkthrough is an informal review led by the author, with peers providing feedback
5. **Answer: B**
   Explanation: Inspection is a structured and formal review process with defined roles and procedures.

B. Small Answer Type Questions (5 Questions)

1. Define static testing.
2. Write any two advantages of static testing.
3. What is a walkthrough?

NOTE

4. Define inspection.
5. Write any two differences between static testing and dynamic testing.

Part–2

C. Long Answer Type Questions (200 words each)

1. Explain static testing and discuss its importance in software quality assurance.
2. Describe different types of static testing techniques.
3. Explain walkthrough and inspection with suitable examples.
4. Discuss the role of reviews in defect prevention.
5. Explain the benefits of static testing in reducing software development cost.

Part–3

D. Long Answer Type Questions (300 words each)

1. Explain static testing in detail. Discuss how it helps in improving software quality and defect prevention.
2. Discuss different review techniques such as informal review, walkthrough, technical review, and inspection.
3. Explain inspection process in detail including roles, steps, and benefits.
4. Discuss static testing artifacts such as requirement documents, design documents, and source code.
5. Explain how static testing supports quality assurance throughout SDLC and compare it with dynamic testing.

UNIT–14
PATH TESTING

UNIT STRUCTURE

14.1 Introduction
14.2 Objective
14.3 Path Testing
14.3.1 Program Graphs
14.3.2 DD-Paths
14.3.3 Test Coverage Metrics
14.3.4 Basis Path Testing
14.3.5 Guidelines and Observations
14.4 Summary
14.5 Glossary
14.6 Answer to check your progress
14.7 Suggested readings - OER
14.8 Reference
14.9 Exercise

14.1 INTRODUCTION

Path Testing is a fundamental white-box testing technique that focuses on examining the internal structure, logic, and control flow of a software program. Unlike black-box testing, which evaluates software based solely on inputs and outputs without considering internal implementation, path testing requires a detailed understanding of how a program executes internally. The primary objective of path testing is to ensure that all logically independent execution paths within a program are tested at least once, thereby increasing confidence in the correctness, robustness, and reliability of the software system.

Software testing is a critical phase in the software development lifecycle, aiming to identify defects and ensure that the software performs as intended. Broadly, testing methodologies are classified into black-box testing and white-box testing. While black-box testing focuses on the functional requirements without considering the internal structure, white-box testing (or structural testing) delves into the internal logic, code structure, and control flow of the application.

Path testing is a specific structural testing technique that is essential for verifying the logical integrity of a program. It is based on the concept

NOTE

that a program is not fully tested until every possible path of execution has been exercised. In many complex applications, simply testing the input and output is insufficient because specific bugs may only trigger under rare combinations of logical decisions (e.g., a specific "else" condition inside a nested loop).

Path testing provides a systematic approach to selecting test cases. By modeling the software as a graph of nodes and edges, testers can mathematically determine the minimum number of test cases required to cover the code's logic. This unit explores the mechanics of converting code into program graphs, calculating complexity metrics, and identifying the "basis paths" that form the foundation of a rigorous testing strategy.

In modern software systems, programs often contain complex decision-making logic implemented through conditional statements, loops, nested structures, and exception-handling mechanisms. Each of these constructs introduces multiple possible execution paths. An execution path represents a specific sequence of statements executed during runtime, depending on input values and program state. Defects may exist only on specific paths, especially those involving rare conditions or boundary scenarios, and such defects can remain undetected if those paths are never executed during testing.

Path testing addresses this challenge by systematically identifying execution paths and designing test cases to traverse them. It is particularly effective in detecting logical errors such as incorrect conditional expressions, missing branches, improper loop termination, unreachable code, and flawed control flow. These types of defects are difficult to identify through surface-level testing but can have serious consequences in real-world applications, especially in safety-critical, financial, or mission-critical systems.

From a Software Quality Assurance perspective, path testing contributes significantly to improving code quality and reducing the risk of post-release failures. By enforcing disciplined examination of control flow, it encourages developers and testers to write clearer, more maintainable code. Path testing also complements other white-box techniques such as statement testing and branch testing, providing deeper insight into program behavior.

However, path testing also presents challenges. As program size and complexity increase, the number of possible execution paths grows exponentially, a phenomenon known as path explosion. Testing all possible paths quickly becomes infeasible for large systems. Therefore, practical approaches such as basis path testing and the use of coverage metrics are employed to identify a manageable subset of paths that still provide high fault-detection capability.

NOTE

In this unit, path testing is discussed in detail, beginning with its conceptual foundation and progressing through program graphs, DD-paths, coverage metrics, basis path testing, and practical guidelines. The objective is to equip learners with both theoretical understanding and practical insight into applying path testing effectively within the software testing lifecycle.

14.2 OBJECTIVE

After studying this unit, you will be able to:

- **Define** path testing and its purpose.
- **Explain** the basic concepts of control flow and program paths.
- **Identify** different execution paths in a program.
- **Construct** a control flow graph for a simple program.
- **Develop** test cases to cover independent execution paths.

14.3 PATH TESTING

Path testing is a structural or white-box testing technique that focuses on examining the internal control flow of a software program by executing different possible paths through its logic. A path is defined as a sequence of statements or basic blocks that are executed in a specific order from the entry point of a program or module to its exit point. Each path represents a unique way in which the program can behave at runtime depending on input values, decision outcomes, and loop iterations.

The fundamental rationale behind path testing is that software defects often hide in specific combinations of conditions and control flow sequences rather than in individual statements. While statement coverage ensures that every statement is executed at least once, it does not guarantee that all logical combinations of decisions have been tested. Path testing goes a step further by attempting to execute different combinations of decisions and loops, thereby increasing the likelihood of uncovering subtle logical errors.

The process of path testing begins with a thorough understanding of the program under test. Testers analyze the source code to identify decision points such as if–else statements, switch cases, loops (for, while, do–while), and exception-handling constructs. Based on this analysis, a control flow or program graph is constructed. This graph serves as a visual and analytical tool for identifying possible execution paths.

NOTE

Once the program graph is available, testers identify paths that should be executed during testing. Ideally, every possible path would be tested; however, in real-world programs, especially those with loops and nested conditions, the number of paths can be extremely large or even infinite. This phenomenon, known as path explosion, makes exhaustive path testing impractical for most non-trivial systems.

To address this limitation, path testing is often combined with structured techniques such as basis path testing, which focuses on a finite set of independent paths. Independent paths are those that introduce at least one new edge or decision outcome not included in previously selected paths. Executing all independent paths ensures that every statement and decision in the program is exercised at least once.

Path testing is particularly effective for detecting logical defects such as missing conditions, incorrect Boolean expressions, improper nesting of conditions, and faulty loop termination. It also helps identify unreachable code and redundant paths that may indicate poor design or incomplete implementation. These defects are often difficult to detect through black-box testing because they depend on internal program logic rather than external behavior alone.

From a software quality assurance perspective, path testing contributes to higher code reliability and maintainability. By encouraging testers and developers to analyze control flow carefully, it promotes clearer program structure and better documentation. Path testing is commonly applied during unit testing and integration testing phases, where access to source code is available and detailed analysis is feasible.

Despite its advantages, path testing must be applied judiciously. Overemphasis on path testing can lead to excessive test cases with diminishing returns. Therefore, testers must balance thoroughness with practicality by selecting meaningful paths based on risk, complexity, and criticality. When applied systematically and in conjunction with other testing techniques, path testing plays a vital role in ensuring the logical correctness and robustness of software systems.

Path testing is a white-box testing method designed to ensure that all logical paths through a program are traversed at least once. The ultimate goal is to uncover logical errors, incorrect assumptions in decision branches, and potential infinite loops. Since a program with loops can theoretically have an infinite number of execution paths, "complete" path testing is often infeasible. To solve this, software engineers use Basis Path Testing, a technique that selects a finite, "basis" set of paths. If these basis paths are tested, the tester can be confident that every statement and branch in the program has been executed.

14.3.1 Program Graphs

NOTE

A Program Graph, often called a Control Flow Graph (CFG), is the fundamental model used in path testing. It translates textual source code into a visual representation that highlights the flow of control rather than the computational details.

- Nodes: In a CFG, a node represents a computational step or a block of sequential statements. A node is usually drawn as a circle. If a sequence of three assignment statements appears in the code without any branching (e.g., a=1; b=2; c=a+b;), they can be grouped into a single node because if the first statement executes, the others necessarily follow.
- Edges: An edge is a directed line (an arrow) connecting two nodes. It represents the flow of control. An edge from Node A to Node B implies that after executing Node A, the program control moves to Node B.
- Regions: A region is an area bounded by edges and nodes. When calculating complexity, the area *outside* the graph is also counted as a region.

Mapping Code to Graphs:

1. Sequence: A linear block of code maps to a single node.
2. If-Then-Else: A decision node is created where the condition is checked. Two edges originate from this node: one for the "True" path and one for the "False" path. These paths eventually merge back at a common junction node.
3. While Loop: A decision node checks the loop condition. If true, an edge goes to the loop body; if false, an edge bypasses the loop. The loop body eventually circles back to the decision node.

14.3.2 DD-Paths

Real-world software contains thousands of lines of code. Drawing a CFG with a node for every single statement would result in a massive, unreadable graph. To handle this complexity, we use Decision-to-Decision Paths (DD-Paths).

A DD-Path represents a chain of execution that can be treated as a single unit. Formally, a DD-Path is a path in the program graph such that:

- Its first node is either the entry node of the routine or a decision node (a node with multiple outgoing edges).
- Its last node is either the exit node of the routine or a decision node.
- There are no other decision nodes inside the path.

Significance:

NOTE

By identifying DD-Paths, we can "collapse" the graph. A sequence of 50 linear statements that contains no branches or loops is logically equivalent to a single line connecting two decisions. Testing one statement in that chain effectively tests them all. This simplification allows testers to focus on the complexity of the *logic* (the branches and loops) rather than the volume of the code.

14.3.3 Test Coverage Metrics

Test coverage metrics are quantitative measures used to track how much of the software has been exercised by the test suite. In path testing, three specific metrics are paramount:

1. Statement Coverage:

   This is the most basic metric. It calculates the percentage of executable lines of code that have been run.

   Limitation: Statement coverage is weak because it ignores logical branches. A test might execute a line inside an if block but fail to test the scenario where the if condition is false (the else case).

2. Branch Coverage (Decision Coverage:

   This metric requires that every decision point (e.g., if, case, while) evaluates to both True and False at least once. Branch coverage is stronger than statement coverage because it ensures that all code blocks *and* all alternative flows are triggered.

3. Path Coverage:

   This is the most comprehensive metric. It measures the percentage of all possible execution paths taken. As noted, loops make 100% path coverage impossible in many cases. Therefore, we aim for "Basis Path Coverage," which covers all independent linear paths.

14.3.4 Basis Path Testing

Basis Path Testing is a structured technique derived by Tom McCabe. It allows a tester to calculate the logical complexity of a design and defines the maximum number of test cases needed to achieve branch coverage.

Cyclomatic Complexity:

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. It specifically counts the number of "independent paths" through the code. An independent path is any path that introduces at least one new set of processing statements or a new condition.

NOTE

There are three ways to calculate the Cyclomatic Complexity, $V(G)$, for a graph G:

1. The Region Method:

   Simply count the number of enclosed regions in the planar graph. Do not forget to add 1 for the region surrounding the graph.

2. The Edge-Node Method:

   This uses the number of edges (E) and nodes (N) in the graph.

3. The Predicate Node Method:

   Count the number of predicate nodes (P). A predicate node is a node containing a condition (it has two or more outgoing edges).

Procedure for Basis Path Testing:

1. Draw the Flow Graph: Convert the code design into a CFG.
2. Calculate V(G): Use one of the formulas above (e.g., E - N + 2). Let's say V(G) = 4.
3. Identify Basis Paths: You need to find 4 unique paths.

   Path 1: The straightforward main path.

   Path 2: Diverge at the first decision.

   Path 3: Diverge at the second decision, etc.

   Note: The paths are not unique; the *set* of paths is what matters. The key is that the paths are linearly independent.
4. Create Test Cases: Design input data that forces the execution of each of the 4 paths.

14.3.5 Guidelines and Observations

While path testing is a powerful mathematical approach to testing, it requires careful application. Here are key guidelines for practitioners:

- **Complexity Limit:** A Cyclomatic Complexity ($V(G)$) greater than 10 suggests that the module is too complex and potentially error-prone. Such modules should be refactored (broken down) into smaller sub-modules.
- **Loop Testing:** Since loops introduce infinite paths, specific strategies are used.

  Skip loop: Execute the loop 0 times (condition false immediately).

  One pass: Execute the loop body exactly once.

NOTE

Two passes: Execute the loop body twice to check re-initialization logic.

M passes: Execute the loop a typical number of times.

- **Data vs. Control:** Path testing focuses on *control* flow. It might miss errors related to data flow (e.g., using a variable before it is initialized) unless the logic explicitly checks for it.
- **The "Correctness" Fallacy:** Even if you test all paths, the program might still be wrong if the specification was misunderstood. Path testing verifies that the code *does what it was written to do*, not necessarily what the user *wants* it to do.

Activity Title: Identifying Independent Paths

Objective:
To understand path testing by identifying execution paths in a simple program.

Materials Required:

- Simple program/flowchart
- Notebook
- Pen

Procedure/Steps:

1. Select a small program containing conditional statements.
2. Draw its control flow graph.
3. Identify the different execution paths.
4. Determine the independent paths.
5. Prepare test cases to execute the identified paths.
6. Record the expected results.

Outcome:
The learner will be able to represent program flow and identify test cases for different execution paths.

Check Your Progress14.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. Path Testing is mainly a part of:
A) Black-box testing
B) White-box testing

C) Acceptance testing
D) Compatibility testing

_____

Q2. A program graph is used to represent the flow of control in a program.

_____

Q3. In basis path testing, the number of independent paths is determined using _____ complexity.

_____

Q4. Write any two test coverage metrics used in path testing.

_____

Q5. DD-Paths in path testing are mainly related to:
A) Data definition and usage relationships
B) Database design
C) Documentation writing
D) Debugging tools only

_____

14.4 SUMMARY

Path testing serves as a rigorous structural testing technique that moves beyond simple functional checks. By converting source code into Program Graphs, testers visualize the complex web of logical decisions within a software application. The use of DD-Paths helps simplify these graphs, making analysis feasible even for larger systems.

Central to this unit is the concept of Cyclomatic Complexity, a metric that quantifies logical complexity and defines the number of Basis Paths. These basis paths represent the minimum set of test cases required to ensure that every decision and statement is executed. While calculating metrics like Statement and Branch Coverage provides a scorecard for testing progress, the ultimate goal of path testing is to uncover logic errors, broken loops, and unreachable code, thereby significantly improving software reliability.

Check Your Progress

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

NOTE

Fill-in-the-Blank Questions:

1. _____ is a graphical representation of the control flow of a program, using nodes and edges.
2. If a graph has 10 Edges and 8 Nodes, the Cyclomatic Complexity using the formula E - N + 2 is _____.
3. A node in a CFG that contains a condition (like an IF statement) and splits the flow is called a _____ node.

True/False Questions:

1. Path testing is a type of black-box testing that ignores internal code structure.
2. Cyclomatic complexity provides the upper bound for the number of independent paths in a program's basis set.
3. Statement coverage is a stronger metric than branch coverage because it tests more logic.

14.5 GLOSSARY

- White-box Testing: A testing strategy that uses knowledge of the internal code, structure, and logic of the software to design test cases.
- Control Flow Graph (CFG): A diagrammatic representation of a program where nodes represent processing steps and edges represent the flow of control.
- Cyclomatic Complexity ($V(G)$): A software metric derived from the CFG that measures the complexity of the program's logic; it indicates the number of independent paths.
- Predicate Node: A node in a graph representing a decision point (like an if or while statement) characterized by having two or more outgoing edges.
- Basis Set: A set of linearly independent paths through the code. Executing all paths in the basis set guarantees that every edge and node has been traversed.
- DD-Path (Decision-to-Decision Path): A sequence of execution between two decision nodes that contains no internal branches, used to simplify graph analysis.

14.6 ANSWER TO CHECK YOUR PROGRESS

Q1. Answer: B

Explanation: Path testing is a white-box technique because it requires knowledge of internal program structure and control flow.

Q2. Answer: True

Explanation: Program graphs (control flow graphs) show nodes and edges representing program statements and flow of execution.

Q3. Answer: Cyclomatic

Explanation: Cyclomatic complexity gives the number of linearly independent paths and helps decide the minimum number of test cases.

Q4. Explanation: Coverage metrics measure how much of the code structure has been executed during testing.

Q5. Answer: A

Explanation: DD-paths (Decision-to-Decision paths) represent sequences of statements between decision points, useful for analyzing program flow.

14.7 SUGGESTED READINGS - OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3793	

14.8 REFERENCE

- Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill Education.
- Sommerville, I. (2015). *Software Engineering* (10th ed.). Pearson.
- Mathur, A. P. (2008). *Foundations of Software Testing*. Pearson Education India.
- McCabe, T. J. (1976). "A Complexity Measure". *IEEE Transactions on Software Engineering*.

NOTE

14.9 EXERCISE

Part A

Multiple Choice Questions:

1. Cyclomatic complexity is primarily a measure of:

   A) Data complexity

   B) Logical complexity

   C) User Interface complexity

   D) Storage complexity

2. Which formula correctly calculates Cyclomatic Complexity ($V(G)$)?

   A) $V(G) = \text{Edges} + \text{Nodes} + 2$

   B) $V(G) = \text{Edges} - \text{Nodes} + 2$

   C) $V(G) = \text{Nodes} - \text{Edges} + 2$

   D) $V(G) = \text{Predicate Nodes} - 1$

3. If a program contains 3 IF conditions executed sequentially (not nested), the minimum number of paths required to test all branches is:

   A) 1 B) 3 C) 4 D) 8

4. Statement coverage ensures that:

   A) Every path has been executed.

   B) Every decision has been taken.

   C) Every line of executable code has been executed.

   D) Every loop is tested for infinite cycles.

5. The "Basis Path Testing" technique was proposed by:

   A) Tom McCabe B) Roger Pressman C) Barry Boehm D) Glen Myers

1. **Answer: B) Logical complexity**

   Explanation: Cyclomatic complexity measures the **logical complexity of a program** by analyzing its control flow, decisions, and independent execution paths. It is useful for determining the complexity of code and the minimum number of test cases needed for basis path testing.

2. **Answer: B) V(G)=Edges−Nodes+2V(G) = \text{Edges} - \text{Nodes} + 2**

NOTE

Explanation: For a connected control flow graph, Cyclomatic Complexity is calculated using the formula **V(G)=E−N+2V(G) = E - N + 2**, where **E** represents the number of edges and **N** represents the number of nodes. It indicates the number of linearly independent paths in the program.

3. **Answer: C) 4**

   Explanation:
   Each sequential IF condition creates two possible branches. For **3 independent IF conditions**, the minimum number of paths required to cover all branches is **3 + 1 = 4**. Thus, four independent paths are sufficient for branch coverage.

4. **Answer: C) Every line of executable code has been executed.**

   Explanation:
   Statement coverage measures whether **each executable statement or line of code** has been executed at least once during testing. It helps identify portions of the program that have not been exercised by the test cases.

5. **Answer: A) Tom McCabe**

   Explanation: Tom McCabe proposed Basis Path Testing, a white-box testing technique based on Cyclomatic Complexity. It identifies a set of **linearly independent paths** through a program so that these paths can be systematically tested.

B. Short Answer Type Questions

1. Differentiate between Statement Coverage and Branch Coverage.
2. Define "Independent Path" in the context of Basis Path Testing.

C. Long Answer Type Questions

1. Draw a Control Flow Graph for a simple program that determines the largest of three numbers (A, B, C). Calculate its Cyclomatic Complexity.

   Hint: The graph will have decision nodes comparing A>B, then A>C or B>C.

   Calculation: Count the number of decision diamonds (P) and add 1.

2. Explain the steps involved in Basis Path Testing. Why is the "Basis Set" of paths significant?

   Hint: Discuss deriving the CFG, calculating V(G), selecting the paths, and designing test cases. Explain that the basis set guarantees coverage without testing infinite loop combinations.

NOTE

NOTE

UNIT 15
DATA FLOW TESTING

UNIT STRUCTURE

15.1 Introduction
15.2 Objective
15.3 Data Flow Testing
15.3.1 Define/Use Testing
15.3.2 Slice-Based Testing
15.3.3 Program Slicing Tools
15.4 Summary
15.5 Glossary
15.6 Answer to check your progress
15.7 Suggested readings - OER
15.8 Reference
15.9 Exercise

15.1 INTRODUCTION

Software testing has traditionally been dominated by Control Flow Testing (as discussed in Unit 14), which focuses on the order in which statements are executed. In Control Flow Testing, we worry about whether the "If" conditions are working, whether the "While" loops terminate, and whether the program reaches the correct exit point. However, extensive research and industrial experience have shown that a significant percentage of software failures are not caused by "wrong logic" in the control flow, but rather by the "mishandling of data."

While Control Flow Testing (Unit 14) focuses on the "paths" or roads through the program, Data Flow Testing focuses on the "traffic" (variables) on those roads. A common source of errors in programming involves the improper handling of data values: variables being used before they are initialized, or variables being assigned a value that is never used. Data Flow Testing is a white-box technique that detects these anomalies by tracking the lifecycle of variables from their declaration to their final usage. It bridges the gap between path testing and data integrity.

Data Flow Testing is a powerful white-box testing technique that shifts the analytical focus from the control structure to the lifecycle of data variables. It is based on the premise that mere execution of a line of

NOTE

code is not enough; we must ensure that the data being processed on that line is valid, initialized, and consistent. In the Von Neumann architecture, which underpins almost all modern computing, data and instructions are distinct. While control flow testing verifies the instructions, data flow testing verifies the data state.

Consider a scenario in a banking application where a variable balance is declared. Control flow testing ensures that the code enters the withdraw() function. However, if the balance variable was defined but never initialized with a value from the database before the withdrawal calculation occurs, the control flow is perfect, but the outcome is catastrophic. This is where Data Flow Testing intervenes. It tracks a variable from its creation (Definition) to its consumption (Use) and its eventual removal (Kill).

One of the primary motivations for Data Flow Testing is the detection of Data Flow Anomalies. An anomaly is a pattern of variable usage that suggests a bug. Common anomalies include:

1. **Defined but never Used:** A variable is assigned a value (e.g., x = 10), but x is never used in any subsequent calculation or output. This suggests that the code is redundant, or the programmer intended to use x but used a different variable by mistake.
2. **Used but never Defined:** This is a critical error where the program attempts to read a variable that has no value. In languages like C++, this leads to reading "garbage" memory; in Java, it might throw a NullPointer Exception.
3. **Defined twice continuously:** A variable is given a value, and then immediately given a new value without the first one ever being read. This indicates that the first assignment was useless and potentially incorrect.

Data Flow Testing bridges the gap between the syntax of the code and the semantics of the business logic. It forces the tester to ask: "I see that we are calculating the tax here, but where did the 'income' data come from, and is it still valid at this specific line of code?" By tracing these "definition-use chains," Data Flow Testing provides a rigorous method for validating the integrity of the information moving through the software system.

15.2 OBJECTIVE

After studying this unit, you should be able to:

- Define the concepts of "Definition" and "Use" in the context of program variables.
- Identify common data flow anomalies (e.g., Use before Define).

NOTE

- Distinguish between "P-use" (Predicate use) and "C-use" (Computational use).
- Explain the concept of Program Slicing and its utility in debugging.
- List tools used for automated program slicing.

15.3 DATA FLOW TESTING

Data Flow Testing involves identifying paths in the program that traverse from the point where a variable is assigned a value to the point where that value is used. To understand this deep testing methodology, we must first formalize the states of data. In any procedural programming language (like C, Java, Python), a variable can be in one of several states during execution.

The Data Flow Model:

We analyze the program using a Control Flow Graph (CFG), but we annotate each node in the graph with information about the variables. The actions performed on a variable v in a statement s are categorized as:

- **Definition (d):** This occurs when the variable is stored or written to. Examples include x = 5, read(x), or passing x as an output parameter to a function. In a graph, if node n assigns a value to v, we say v is in the set $DEF(n)$.
- **Use (u):** This occurs when the variable is read or referenced. Examples include y = x + 1 (where x is used), if (x > 0) (where x is used), or passing x as an input parameter. If node n reads v, we say v is in the set $USE(n)$.
- **Kill (k) or Undefine:** This occurs when a variable's value is invalidated. This happens when the variable goes out of scope (e.g., exiting a function), or when the variable is redefined (the old value is "killed" by the new definition).

Data Flow Anomalies:

The core of Data Flow Testing is the search for sequences of actions that make no logical sense. These are often denoted by two-letter codes representing the sequence of events:

- ~d (Undefined): The variable is used without ever being defined. This is a fatal error in most languages.
- du (Define-Use): The normal, healthy sequence. A variable is defined, and then it is used.
- dd (Define-Define): A variable is defined and then defined again without being used in between. This is a warning sign. For

NOTE

example: x = 5; x = 10;. The first assignment was wasted. It might indicate a missing line of code that was supposed to use the 5.

- dk (Define-Kill): A variable is defined and then immediately killed (goes out of scope) without being used. This usually means the variable is unnecessary and is cluttering the code.
- kk (Kill-Kill): Attempting to kill or undefine a variable that is already undefined. This often happens in pointer management (double-free errors).

Static vs. Dynamic Data Flow Testing:

Data flow testing can be performed in two ways:

1. Static Data Flow Analysis: This is done by a tool or code review without running the program. The analyzer scans the source code to find patterns like "variable defined but never used." This is very fast and covers the whole code but can produce "false positives" (reporting errors that can never actually happen during execution).
2. Dynamic Data Flow Testing: This involves actually running the program with test cases designed to force execution along specific "Define-to-Use" paths. This confirms that not only does the path exist, but the data values remain correct along that path.

15.3.1 Define/Use Testing

Define/Use (DU) Testing is the practical implementation of data flow theory. It is a systematic method for selecting test paths based on the dependencies between data definitions and data uses. To practice DU testing, we must distinguish between the *types* of uses, as they affect the flow of the program differently.

1. Types of Data Use:

- **C-Use (Computational Use):**

A C-Use occurs when a variable appears on the right-hand side of an assignment statement, in an output statement, or as a parameter in a function call. It directly affects the *computation* of new results.

Example: In the statement total = price * tax_rate;, both price and tax_rate have a C-Use. The flow of control simply passes through this node to the next sequential node.

- **P-Use (Predicate Use):**

A P-Use occurs when a variable appears in a conditional expression that determines the flow of control (branching).

NOTE

Example: In the statement if (balance < 0), the variable balance has a P-Use. This use is critical because it dictates whether the program executes the "True" branch or the "False" branch.

2. The Definition-Use (DU) Chain:

A DU-Chain connects a definition of a variable to a use of that variable. Formally, a path from node i to node j is a **Definition-Clear Path** with respect to variable x if:

1. Variable x is defined at node i.
2. Variable x is used at node j.
3. Variable x is *not* redefined (or killed) at any node between i and j.

If the path is not definition-clear, it means the value from node i has been overwritten, so node j is actually using a *different* definition of x. DU testing focuses exclusively on Definition-Clear paths.

3. Coverage Criteria (The Hierarchy of Testing):

Just as we have statement and branch coverage, Data Flow Testing has a hierarchy of thoroughness. A tester must choose the appropriate level based on the risk associated with the software.

- **All-Defs Strategy:**

This is the weakest acceptable strategy. It requires that for every definition of every variable, we identify and test *at least one* path to *some* use of that variable.

Critique: It ensures no dead code exists (every definition is used), but it might miss errors where a definition works for a calculation (C-Use) but fails when used in a logic check (P-Use).

- **All-Uses Strategy:**

This is a stronger strategy. For every definition of a variable, the tester must test a path to *every* possible use of that definition.

Detail: If variable x is defined in line 1, and used in line 10 (calculation) and line 20 (IF condition), the All-Uses strategy demands we test paths to both line 10 and line 20. This ensures the data is valid for all its intended purposes.

- **All-DU-Paths Strategy:**

This is the strongest and most expensive strategy. A single definition might reach a single use via multiple different paths (e.g., through different branches of a loop or nested IFs). All-DU-Paths requires testing *every* definition-clear path from every definition to every use.

NOTE

Use Case: This is typically reserved for safety-critical systems (avionics, medical devices) where data corruption on any specific trajectory could be fatal.

Activity Title: Tracking Data Flow

Objective:
To understand how data flow testing examines the definition and use of variables in a program.

Materials Required:

- Simple program
- Notebook
- Pen

Procedure/Steps:

1. Select a simple program containing several variables.
2. Identify where each variable is defined.
3. Identify where each variable is used.
4. Trace the movement of important variables through the program.
5. Identify possible definition-use problems.
6. Prepare test cases to cover important data-flow paths.

Outcome:
The learner will understand the relationship between variable definitions and uses and how data flow testing can detect defects.

15.3.2 Slice-Based Testing

Slice-Based Testing is a technique derived from the concept of Program Slicing, introduced by Mark Weiser in his PhD thesis in the early 1980s. It addresses a fundamental problem in software maintenance and testing: Cognitive Overload. When a tester or developer looks at a 10,000-line program and sees an error in the output of variable totalscore, it is incredibly difficult to mentally filter out the 9,500 lines of code that have nothing to do with total_score.

The Concept of a Slice:

A Program Slice is a subset of the original program that preserves the behavior of the program with respect to a specific "Slicing Criterion." A Slicing Criterion is usually a pair (s, v), where s is a specific statement number and v is a variable used at that statement.

NOTE

If we request a slice for (Statement 50, *z*), the slicing algorithm looks backward from line 50. It asks: "What defined *z*? Maybe line 40. What defined the variables used in line 40? Maybe line 30." It recursively follows the data and control dependencies backward. The resulting "slice" is an executable program containing only the lines relevant to computing *z*.

Static vs. Dynamic Slicing:

- **Static Slicing:**

This computes the slice based solely on the source code, considering all *possible* inputs. If a variable could arguably depend on an IF statement, the static slice includes both the True and False blocks to be safe.

  - *Pros:* It guarantees that no dependency is missed.
  - *Cons:* The slices can be very large (imprecise) because they include paths that might never actually execute.

- **Dynamic Slicing:**

This computes the slice based on a *specific execution trace* (a specific test case). If the test case executed the "True" branch of an IF statement, the dynamic slice completely ignores the code in the "False" branch.

  - *Pros:* The slice is much smaller and precise. It shows exactly what caused the value of the variable *in this specific failed test*.
  - *Cons:* It is only valid for that specific input.

Applications in Testing:

1. **Debugging:** If a test fails for a specific output, a dynamic slice of that output isolates the bug location to a fraction of the code.
2. **Regression Testing:** If we change a line of code, we can compute the "Forward Slice" of that variable to see what other parts of the program are affected. We only need to re-test the parts of the program that fall inside the forward slice, saving massive amounts of testing time.
3. **Code Comprehension:** New developers can use slicing to understand how a complex calculation works without reading the entire codebase.

15.3.3 Program Slicing Tools

While the concept of slicing is intuitive, performing it manually is practically impossible for modern software. This is due to complex language features like:

NOTE

- **Pointers and Aliasing:** In C/C++, if p points to x, modifying *p modifies x. A human reader often misses this connection, but a tool tracks memory addresses to catch it.
- **Function Calls:** Data flows across function boundaries. A variable passed into a function might be modified globally. Tracking this across hundreds of files requires automation.
- **Polymorphism:** In Object-Oriented languages, it's often not clear which version of a function is called until runtime.

How Slicing Tools Work:

Automated tools work by constructing a System Dependence Graph (SDG).

1. The tool parses the source code.
2. It builds a Control Flow Graph (CFG) for every function.
3. It adds "Data Dependence Edges" connecting definitions to uses.
4. It adds "Control Dependence Edges" connecting decisions (IF/WHILE) to the statements they control.
5. It connects the graphs of different functions (Interprocedural analysis).

When a user selects a variable, the tool simply performs a "Graph Reachability" algorithm (like Depth-First Search) on the SDG to find all connected nodes.

Categories of Tools:

- Text-Based Tools: These tools take source code as input and output a list of line numbers or a new source file containing only the slice. These are useful for automated pipelines.
- Interactive/Graphical Tools: These are often plugins for Integrated Development Environments (IDEs) like Eclipse, Visual Studio, or specialized tools like CodeSurfer. They allow the developer to right-click a variable and select "Show Backward Slice." The IDE then grays out all irrelevant code, leaving the slice highlighted in bold or color. This visual feedback is incredibly powerful for debugging.

Limitations:

Even the best tools face challenges. "Unreachable code" or obscure pointer arithmetic can sometimes confuse static analysis, leading to slices that are slightly larger than necessary. However, they are still significantly more efficient than manual inspection.

NOTE

Check Your Progress

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. Data Flow Testing mainly focuses on:
A) User interface colors
B) Definition and use of variables in the program
C) Only system performance
D) Hardware compatibility

_____

Q2. Data flow testing is useful for detecting uninitialized variables and incorrect variable usage.

_____

Q3. Define/Use testing checks the relationship between variable _____ and variable use.

_____

Q4. Write any two types of data flow anomalies.

_____

Q5. Program slicing in data flow testing is mainly used to:
A) Reduce program size permanently
B) Extract relevant statements affecting a variable or output
C) Convert code into flowchart
D) Replace unit testing

_____

15.4 SUMMARY

In this unit, we have explored Data Flow Testing as a crucial complement to Control Flow Testing. While control flow ensures that the logic travels the correct path, data flow ensures that the payload (the data) traveling that path is valid. We learned that variables have a lifecycle comprising Definition, Use, and Kill. Errors in this lifecycle result in Data Flow Anomalies, such as using undefined variables or defining variables that are never used.

We examined Define/Use (DU) Testing, which formalizes the process of checking C-Uses (computational) and P-Uses (predicates). We discussed the hierarchy of coverage criteria, noting that All-DU-Paths provides the most rigorous verification by testing every logical path between a data definition and its use.

NOTE

We delved into Slice-Based Testing, a technique that decomposes large programs into smaller, relevant chunks based on variable dependencies. We distinguished between Static Slicing (analyzing all possibilities) and Dynamic Slicing (analyzing specific executions) and highlighted the necessity of Automated Slicing Tools to handle the complexities of modern programming languages. By mastering these techniques, a software tester can uncover deep, hidden bugs that functional testing often misses, leading to more robust and reliable software systems.

15.5 GLOSSARY

- Data Flow Anomaly: An error pattern in the usage of variables, such as a variable being used before it is defined (~d) or defined and then immediately killed (dk).
- Definition (def): The action of assigning a value to a variable. This can happen via direct assignment, input statements (cin, scanf), or as an output parameter of a function.
- Use (u): The action of reading the value of a variable.
- C-Use (Computational Use): A specific type of variable use where the variable is employed in a calculation, output, or to define another variable. It does not directly affect the control flow.
- P-Use (Predicate Use): A specific type of variable use where the variable appears in a condition (Boolean expression) that controls the flow of execution (e.g., inside an if or while statement).
- DU-Path (Definition-Use Path): A path in the control flow graph from a definition of a variable to a use of that variable, with the constraint that the variable is not redefined along the path.
- Definition-Clear Path: A path between two points where a specific variable's value is preserved (i.e., it is not re-assigned or killed).
- Program Slicing: A decomposition technique that extracts all statements relevant to the computation of a specific variable at a specific point in the program.
- System Dependence Graph (SDG): A complex graph structure used by automated tools to represent both data and control dependencies across the entire software system, facilitating program slicing.
- Static Analysis: The analysis of computer software that is performed without actually executing programs, often used to find data flow anomalies.

NOTE

15.6 ANSWER TO CHECK YOUR PROGRESS

Q1. Answer: B

Explanation: Data flow testing is a white-box technique that checks how variables are defined, used, and whether incorrect usage causes defects.

Q2. Answer: True

Explanation: This technique identifies defects such as using a variable before assigning a value or defining it but never using it.

Q3. Answer: Definition

Explanation: Define/Use testing ensures that every variable definition properly reaches its usage without errors.

Q4. Explanation: These anomalies indicate improper variable handling, which may lead to incorrect output or runtime errors.

Q5. Answer: B

Explanation: Program slicing helps identify the set of statements that influence a particular variable or output, making debugging and testing easier.

15.7 SUGGESTED READINGS - OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3794	

15.8 REFERENCE

- Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill Education.
- Pezzè, M., & Young, M. (2008). *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons.
- Frankl, P. G., & Weyuker, E. J. (1988). "An Applicable Family of Data Flow Testing Criteria". *IEEE Transactions on Software Engineering*.

• Roper, M. (1994). *Software Testing*. McGraw-Hill.

NOTE

15.9 EXERCISE

Part–1

A. Multiple Choice Questions:

Q1. Domain testing mainly focuses on:

A) GUI layout

B) Input values and their ranges

C) Hardware faults

D) Database backup

Q2. Testing values at the edge of input ranges is called:

A) Stress testing

B) Boundary value testing

C) Recovery testing

D) Mutation testing

Q3. A domain is:

A) Set of test tools

B) Range of valid input values

C) Set of outputs only

D) Set of documents

Q4. Domain errors mostly occur due to:

A) Incorrect input validation

B) Better documentation

C) Faster CPU

D) More memory

Q5. Domain testing is closely related to:

A) White-box testing

B) Black-box testing

C) Database testing only

D) Security testing only

Explanations of MCQs

Answer: B) Input values and their ranges

NOTE

Explanation: Domain testing focuses on the range of values that can be provided as input to a software system. It checks whether the software correctly handles **valid, invalid, minimum, maximum, and boundary input values**. This helps identify errors in input conditions and validation.

Answer: B) Boundary value testing

Explanation: Boundary Value Testing examines values at the **edges or limits of an input range**, such as the minimum and maximum allowed values. Errors frequently occur at these points, so testing boundary values helps detect defects that may not be found through ordinary input testing.

Answer: B) Range of valid input values

Explanation: In software testing, a domain refers to the **set or range of input values** that a variable or system is expected to accept. For example, if a system accepts an age from 18 to 60, this range represents the valid input domain for the age field.

Answer: A) Incorrect input validation

Explanation: Domain errors commonly occur when a software application **does not correctly validate input values**. If invalid values are accepted or valid values are incorrectly rejected, the system may produce incorrect results or unexpected behaviour. Proper input validation helps prevent such errors.

Answer: B) Black-box testing

Explanation: Domain testing is generally considered a **black-box testing technique** because it focuses on the relationship between inputs and expected outputs without requiring knowledge of the program's internal code. Testers select values from different input domains to verify whether the software behaves correctly.

B. Small Answer Questions

1. Define domain testing.
2. What is meant by input domain?
3. Write any two benefits of domain testing.
4. What is boundary value testing?
5. Write any two examples of domain errors.

Part–2

Long Answer Type Questions (200 words)

1. Explain domain testing and its importance.

NOTE

2. Discuss valid and invalid input domains with examples.
3. Explain boundary testing in relation to domain testing.
4. Describe the process of selecting test cases using domain analysis.
5. Discuss common domain faults and how to detect them.

Part–3

Long Answer Type Questions (300 words)

1. Explain domain testing in detail with suitable examples.
2. Discuss domain analysis and test case design for input ranges.
3. Explain boundary value analysis and its role in domain testing.
4. Discuss domain testing for multi-variable inputs.
5. Explain advantages, limitations, and applications of domain testing.

RNTU Rabindranath TAGORE UNIVERSITY

Approved by : AICTE, NCTE, BCI, INC, MP PARAMEDICAL COUNCIL, DEB (UGC) | Recognized by : UGC | Member of : AIU, ACU

RABINDRANATH TAGORE UNIVERSITY (M.P.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org / for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

_____

_____

_____

***

BLOCK 4	SOFTWARE VERIFICATION, VALIDATION AND TESTING LIFECYCLE	259-330
Unit 16	Software Verification and Validation	261
Unit 17	Reviews in Testing Lifecycle	277
Unit 18	Acceptance Testing	291
Unit 19	V-test Model	303
Unit 20	Testing During Test Planning Phase	315



Block IV focuses on the systematic approaches used to ensure that software meets its specified requirements and satisfies user expectations. It introduces the concepts of Software Verification and Validation (V&V) and explains how reviews contribute to improving software quality throughout the testing lifecycle. The block also discusses acceptance testing, which determines whether a software product is ready for delivery and use by the customer. The V-Model is explored as a structured software development and testing model that establishes a relationship between development activities and corresponding testing activities. Finally, the block examines testing during the test planning phase, emphasizing the importance of defining objectives, scope, resources, schedules, test strategies, risks, and responsibilities before actual testing begins. Together, these units provide a comprehensive understanding of quality assurance, review practices, acceptance criteria, and systematic test planning.

NOTE

UNIT 16
SOFTWARE VERIFICATION AND VALIDATION

UNIT STRUCTURE

16.1 Introduction
16.2 Objective
16.3 Software Verification and Validation
16.3.1 Introduction
16.3.2 Verification
16.3.3 Verification Workbench
16.3.4 Methods of Verification
16.3.5 Types of reviews on the basis of Stage Phase
16.3.6 Entities involved in verification
16.4 Summary
16.5 Glossary
16.6 Answer to check your progress
16.7 Suggested readings - OER
16.8 Reference
16.9 Exercise

16.1 INTRODUCTION

The historical evolution of software engineering is marked by a transition from ad-hoc programming to disciplined quality assurance. In the early decades of computing, testing was often viewed as a distinct phase performed only after the code was written. This "code-and-fix" approach proved economically disastrous for large-scale systems because defects introduced during the requirements or design phases remained undetected until the very end of the lifecycle. By that time, rectifying these fundamental errors required dismantling the underlying architecture, leading to massive cost overruns and project failures.

To address this systemic issue, the industry adopted the twin disciplines of Software Verification and Validation (V&V). These are not synonymous terms; rather, they represent two different angles of quality assurance that run continuously throughout the project. Verification is the theoretical and engineering discipline of ensuring that the product is being built according to the specifications and standards derived from

NOTE

the previous phase. It is a static process that relies on human intellect and mathematical rigor to analyze artifacts without execution. Validation, conversely, is the dynamic process of executing the software to ensure it satisfies the user's operational needs.

In the software development life cycle, ensuring quality is not a single event that happens at the end of the project. It is a continuous process of checking and re-checking. This process is broadly categorized into two distinct but related activities: Verification and Validation (often abbreviated as V&V). While these terms are frequently used interchangeably in casual conversation, they hold very specific and distinct meanings in software engineering. Understanding the difference between checking if the logic is correct (Verification) versus checking if the user is happy (Validation) is the foundation of a robust Quality Assurance strategy. This unit explores the static aspect of this duo Verification in depth.

This unit focuses on the verification methodology, which serves as the primary defect prevention mechanism in the Software Development Life Cycle (SDLC). By treating requirements, design documents, and source code as scientific artifacts subject to rigorous peer review and analysis, organizations can detect and remove the majority of defects before a single test case is ever executed.

16.2 OBJECTIVE

After studying this unit, you should be able to:

- Clearly distinguish between the concepts of Verification and Validation.
- Describe the components and flow of a Verification Workbench.
- Detailed the various methods of verification including Peer Reviews, Walkthroughs, and Inspections.
- Analyze the different types of reviews conducted at various phases of the software life cycle.
- Identify the roles and responsibilities of the different entities (people) involved in the verification process.

16.3 SOFTWARE VERIFICATION AND VALIDATION

16.3.1 Introduction to V&V

Software Verification and Validation is a systems engineering discipline that helps the development organization build quality into the software during every step of the lifecycle. The theoretical relationship between these two concepts is best illustrated by the V-Model of software

NOTE

development. In this model, the left side of the 'V' represents the decomposition of the system (Requirements, Architecture, Detailed Design), while the right side represents the integration and testing of those components. Verification is the dominant activity on the left side, ensuring that each step logically follows from the previous one, while Validation dominates the right side, confirming that the reassembled system works in the real world.

The core distinction lies in the question each discipline answers. Verification addresses the engineering question: "Are we building the product right?" It confirms that the software adheres to the technical specifications, coding standards, and architectural blueprints defined at the start of the phase. Validation addresses the customer-centric question: "Are we building the right product?" It confirms that the software, regardless of how well it follows the spec, actually solves the user's business problem. A piece of software can be completely verified (zero bugs, perfect code structure) but fail validation (it solves the wrong problem).

The concepts of Verification and Validation are derived from the need to ensure that software systems are reliable, safe, and perform as intended. The IEEE Standard Glossary of Software Engineering Terminology defines these terms separately to highlight their unique focuses.

Validation is a dynamic process that involves executing the software to check if it satisfies the customer's needs. It is often summarized by the question: *"Are we building the right product?"*

In contrast, Verification is a static process. It involves evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. It is summarized by the question: *"Are we building the product right?"*

Both processes are complementary. You can verify a product perfectly (the code matches the spec perfectly), but if the spec was wrong to begin with, the validation will fail. Conversely, a product might pass validation (the customer likes it), but fail verification because the internal code is messy and unmaintainable.

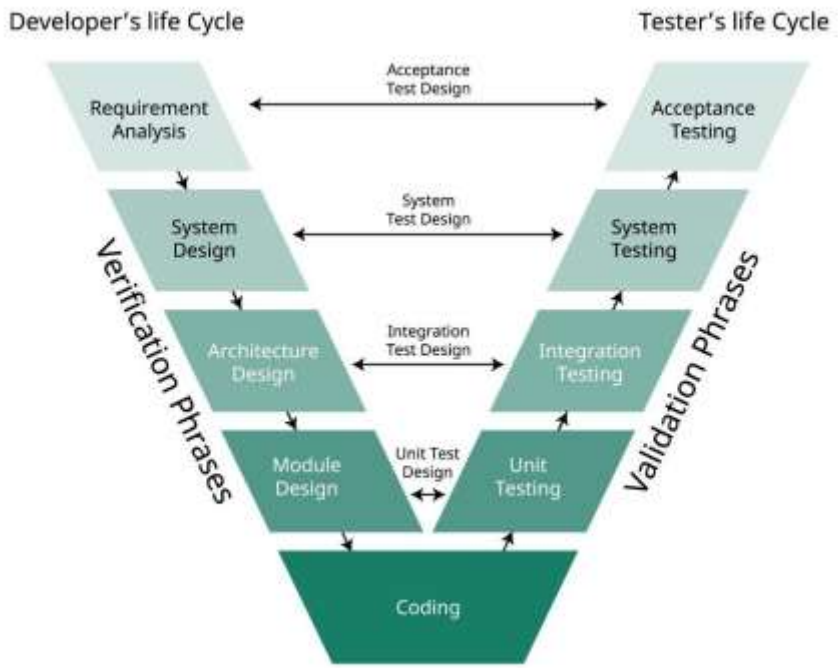


Fig. 16.1 : Development Life Cycle in 'V' shape

The economic justification for robust verification is found in the "Cost of Change Curve" proposed by Barry Boehm. Theoretical models and empirical data show that the cost to fix a defect rises exponentially as the project moves through the SDLC phases. A logic error identified during a design verification session might require only minutes to fix by erasing a line on a whiteboard. However, if that same error propagates into the code, passes through unit testing, and is only discovered during acceptance testing, it necessitates re-coding, re-compiling, re-testing, and documentation updates, costing orders of magnitude more. Thus, verification is essentially a cost-containment strategy.

16.3.2 Verification

Verification is often classified as "Static Testing" because it does not involve the execution of the program. Instead, the software artifacts whether they are requirement documents, design models, or source code files are treated as static data to be analyzed. This analysis relies on the premise that human peers and experts can identify logic errors, missing conditions, and potential failures by tracing the mental model of the software.

The process is fundamentally preventative. While dynamic testing proves the presence of failure, verification works to prove the correctness of the logic. It covers a broader scope than just the code; verification activities are applied to the requirements specification to check for ambiguity, to the design documents to check for structural

NOTE

integrity, and to the test plans themselves to ensure traceability. A key theoretical benefit of verification is its ability to uncover the root cause of a defect immediately. When a reviewer spots an infinite loop condition in a code snippet, the error is located instantly. In contrast, if that loop causes a system crash during dynamic testing, the developer must spend significant time debugging to trace the crash back to the specific line of code.

Verification is primarily a human-centric, manual activity that does not involve running the code. It is often referred to as "Static Testing." The goal of verification is to ensure that the work products generated at any stage of development—whether they are requirement documents, design architectures, or source code—adhere to the defined standards and specifications.

The primary advantage of verification is early defect detection. Since verification starts during the requirements phase, errors can be caught before a single line of code is written. Fixing an ambiguity in a requirement document during a review costs pennies compared to the thousands of dollars it might cost to fix the resulting bug after the software has been deployed to production. Verification acts as a filter, removing defects at the point of creation rather than letting them flow downstream into the testing phase.

Activity Title: Verification or Validation?

Objective:
To distinguish between software verification and validation activities.

Materials Required:

- Notebook
- List of software quality activities
- Reference material

Procedure/Steps:

1. Prepare a list of activities such as requirement review, code inspection, unit testing, system testing, and user acceptance testing.
2. Classify each activity as primarily verification or validation.
3. Give reasons for each classification.
4. Discuss how both activities contribute to software quality.
5. Prepare a comparison table.

NOTE

Outcome:
The learner will be able to distinguish between verification and validation and understand their importance.

16.3.3 Verification Workbench

The concept of the Verification Workbench elevates the review process from a casual activity to a structured industrial process. It serves as a theoretical model describing the environment necessary for effective verification. The workbench functions as a transformation system where specific inputs are processed through defined procedures to produce quality-assured outputs.

The inputs to the workbench typically include the item to be verified (such as a source code module) and the reference specification (such as the design document) against which the item is checked. Crucially, the workbench relies on "Standards and Guidelines" and "Checklists." The checklist acts as a cognitive aid for the reviewer, ensuring that common error patterns—such as uninitialized variables or weak error handling are actively sought out rather than relying on the reviewer's memory.

The process component of the workbench defines the rigor of the review, dictating whether it is a simple desk check or a formal team inspection. The output of the workbench is not merely a "pass/fail" grade but a comprehensive set of artifacts, including a formal defect log, a verification report summarizing the quality metrics, and a decision regarding the release of the work product to the next phase.

The concept of a "Workbench" is used to describe the structured environment in which verification takes place. Just as a carpenter has a workbench with inputs (wood), tools (saws), and outputs (furniture), a software verifier has a process workbench.

The Verification Workbench consists of three primary stages. First is the Input stage, where the item to be verified (such as a Requirement Specification or a Design Document) is brought into the environment alongside the standards and checklists against which it will be checked.

The second stage is the Process itself. This is where the verification activity such as a review or inspection takes place. During this stage, the verifier compares the input document against the standards. If discrepancies are found, they are logged as defects. Rework is then performed by the author to correct these defects.

The final stage is the Check and Output stage. Once the rework is complete, a follow-up check ensures that the corrections were made successfully. If the document passes this check, it is marked as "Verified" and becomes the output of the workbench. This output then

NOTE

serves as the approved input for the next phase of the software development life cycle.

16.3.4 Methods of Verification

Verification methodologies exist on a continuum of formality, ranging from informal peer discussions to strictly regulated inspections. The choice of method depends on the criticality of the software and the organizational culture.

Walkthroughs

A Walkthrough is a semi-informal, author-centric presentation. In this method, the author of the document or code leads a team of peers through the technical product, simulating the execution of the logic and explaining the intent behind the design. The primary theoretical goal of a walkthrough is knowledge transfer and consensus building rather than rigorous defect detection. Because the author controls the pace and the narrative, there is a risk that they may inadvertently gloss over logical gaps or guide the reviewers away from problematic areas. It is an educational tool that helps the team understand the system but is less effective at filtering out deep defects.

Technical Reviews

Moving up the formality scale, Technical Reviews are structured meetings focused on the technical suitability of the product. Unlike walkthroughs, these are not necessarily led by the author. A team of qualified personnel examines the software to determine if it meets the specifications and adheres to standards. The focus here is often on the approach and architecture—evaluating whether the chosen algorithm is the most efficient or if the class structure is maintainable. The outcome is a technical consensus on the integrity of the solution.

Formal Inspections (Fagan Inspections)

The most rigorous form of verification is the Inspection, formally developed by Michael Fagan. This method treats software verification as a formal engineering process with strict entry and exit criteria. An inspection cannot begin until the documents are proven to be ready, and it cannot conclude until all identified defects are verified as fixed. The theoretical underpinning of an inspection is the removal of the author's ego from the process. The meeting is led by a trained Moderator, not the author, and follows a disciplined script where code is often read aloud line-by-line. Data is collected on defect density (errors per page) to predict software quality quantitatively.

16.3.5 Types of reviews on the basis of Stage Phase

Verification activities adapt to the specific nature of the artifacts produced at each stage of the SDLC. During the Requirements

NOTE

Verification stage, the analysis focuses on semantics and logic. Reviewers scrutinize the requirements specification for ambiguity, incompleteness, and inconsistency. A critical aspect of this phase is verifying "testability" ensuring that every requirement is written in a way that can be definitively proven true or false.

As the project moves to Design Verification, the focus shifts to architectural integrity. Preliminary design reviews examine the allocation of requirements to hardware and software components, while critical design reviews analyze the detailed data structures and algorithms. Reviewers look for high coupling (dependencies) between modules, which is a theoretical indicator of poor maintainability.

Code Verification occurs during the implementation phase. Here, the focus is on syntactical correctness, adherence to coding standards, and algorithmic logic. Reviewers check for security vulnerabilities (like buffer overflows), memory leaks, and dead code that will never be executed.

Verification reviews happen continuously throughout the Software Development Life Cycle (SDLC). The nature of the review changes depending on the phase.

Requirement Reviews : This review occurs at the very beginning of the project. Stakeholders, business analysts, and testers review the Software Requirement Specification (SRS). The focus is to find ambiguous, incomplete, or contradictory requirements. For example, checking if a requirement states "The system should be fast" (which is ambiguous) versus "The system should load in under 2 seconds" (which is verifiable).

Design Reviews : Once requirements are finalized, the system architecture is verified. In High-Level Design reviews, the team checks if the proposed architecture meets the scalability and security requirements. In Low-Level Design reviews, the logic of specific modules, database schemas, and interface definitions are scrutinized to ensure they align with the requirements.

Code Reviews : This is the most common form of verification. Developers review source code to check for syntax errors, logic flaws, memory leaks, and adherence to coding standards (like naming conventions). Code reviews also ensure that the code is readable and maintainable for future developers.

Test Plan Reviews : Verification also applies to the testing team itself. The Test Plan and Test Cases are reviewed to ensure coverage. A Test Plan review checks if the testing strategy covers all requirements and if the estimated resources and schedules are realistic.

NOTE

Finally, Test Plan Verification ensures that the validation strategy itself is sound. Reviewers trace every test case back to a specific requirement to ensure 100% coverage, verifying that the testing effort will be sufficient to validate the system.

16.3.6 Entities involved in verification

Formal verification relies on a specific sociological structure to function effectively. The team roles are designed to create a system of checks and balances that maximizes defect detection while minimizing interpersonal conflict.

The Moderator is the central authority of the inspection. Their role is process management rather than technical critique. They schedule the inspection, ensure all participants are prepared, and facilitate the meeting to keep it focused on defect detection rather than solution design. Crucially, the Moderator must be impartial and independent of the author.

The Author is the creator of the work product. In a formal inspection, the author plays a passive role during the meeting to prevent defensiveness. They are present to answer questions about intent but are not allowed to steer the inspection. Their primary work occurs after the meeting during the rework phase, where they correct the identified defects.

The Scribe (or Recorder) acts as the official historian of the meeting. Their responsibility is to accurately record every defect found, noting its location and severity. This role is vital because it separates the act of finding defects from the administrative burden of documenting them, allowing the rest of the team to maintain their analytical flow.

Formal verification, particularly Inspections, involves a specific set of roles (Entities), each with a defined responsibility. It is not just a group of people sitting in a room; it is a structured team.

The Moderator : The Moderator is the leader of the verification process. They are responsible for planning the inspection, scheduling the meeting, and ensuring the team sticks to the agenda. Crucially, the Moderator must be a neutral party, not the boss or the author, to ensure an unbiased environment. They mediate disagreements and ensure the focus remains on finding defects, not attacking the author.

The Author : The Author is the person who created the work product being verified. Their role is primarily passive during the inspection; they are there to hear the feedback and answer clarification questions. They should not try to defend their work aggressively, as the goal is to improve the product. After the meeting, the Author is responsible for performing the rework to fix the identified defects.

NOTE

The Scribe (or Recorder) : The Scribe documents the findings of the meeting. As reviewers call out defects, the Scribe writes them down in the defect log, noting the location and severity. This allows the other participants to focus on the discussion without worrying about taking notes.

The Reviewer (or Inspector) : The Reviewers are the subject matter experts responsible for finding defects. They spend time before the meeting reading the document (Preparation) and attend the meeting to raise issues. A reviewer might be a peer developer, a tester, or a business analyst, depending on the document being reviewed. Inspectors (or Reviewers) are the subject matter experts responsible for finding errors. Theoretical best practice dictates that inspectors must prepare *before* the meeting by reviewing the documents alone; the meeting itself is simply for logging the issues they found.

The Reader : In some formal inspections, a "Reader" is designated to read the document aloud to the group, paraphrasing the logic. This forces the team to move at a specific pace and ensures no section is skipped.

Check Your Progress

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. Verification mainly answers the question:
A) Are we building the right product?
B) Are we building the product right?
C) Is the product profitable?
D) Is the software installed?

_____

Q2. Validation ensures that the final product meets customer requirements.

_____

Q3. Verification activities include reviews, inspections, and _____.

_____

Q4. Write any two methods of verification.

_____

Q5. Verification workbench mainly refers to:
A) A place to install software
B) A structured set of tools and techniques used for verification
C) A testing lab for hardware
D) A database server

_____

NOTE

16.4 SUMMARY

Software Verification and Validation represent the comprehensive quality strategy required for modern software engineering. We have established that Verification is the static, preventative process of "building the product right," utilizing human intellect to analyze artifacts against specifications. This is distinct from Validation, the dynamic process of "building the right product" through execution. The unit explored the Verification Workbench as a system of inputs, checklists, and procedures designed to standardize quality control. We contrasted the educational nature of Walkthroughs with the rigorous, defect-finding focus of Fagan Inspections. Furthermore, we examined how verification evolves from checking semantic ambiguity in requirements to analyzing syntax in code, and how the assignment of specific roles like the Moderator and Scribe ensures the objectivity and efficiency of the review process.

Check Your Progress

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Fill-in-the-Blank Questions:

1. The systems engineering discipline that asks "Are we building the product right?" is called _____.
2. The formal review process that involves entry criteria, exit criteria, and data collection is known as a _____.
3. In the V-Model of software development, verification activities occur _____ to the coding and integration phases.

True/False Questions:

1. Verification involves the dynamic execution of the software to find runtime errors.

2. A Walkthrough is a formal inspection method led by a trained Moderator.

3. The cost of fixing a software defect increases as the project moves from requirements to maintenance.

NOTE

16.5 GLOSSARY

- Verification: The static process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
- Validation: The dynamic process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.
- Verification Workbench: A conceptual environment comprising the tools, input products, checklists, and standards required to perform effective verification.
- Fagan Inspection: A highly structured, formal peer review process defined by Michael Fagan, characterized by defined roles, separate preparation and meeting phases, and quantitative quality assessment.
- Static Analysis: The analysis of software artifacts, such as documentation and source code, without the actual execution of the program.
- Moderator: The individual responsible for leading a formal review, ensuring process adherence, and managing the team dynamics, distinct from the author of the work.

16.6 ANSWER TO CHECK YOUR PROGRESS

Q1. Answer: B

Explanation: Verification ensures that the software is developed correctly according to specifications and standards.

Q2. Answer: True

Explanation: Validation checks whether the software satisfies user needs and intended use, usually through execution and user-level testing.

Q3. Answer: Audits

Explanation: Verification is mostly a static process, including reviews, audits, walkthroughs, and inspections without executing the code.

Q4. Explanation: Verification methods check correctness of documents, design, and code to prevent defects early.

Q5. Answer: B

Explanation: A verification workbench supports verification tasks such as document reviews, inspections, and compliance checking.

NOTE

16.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3795	

16.8 REFERENCE

- Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill Education.
- Sommerville, I. (2011). *Software Engineering* (9th ed.). Addison-Wesley.
- IEEE Computer Society. (2014). *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide)*.
- Burnstein, I. (2003). *Practical Software Testing*. Springer.

16.9 EXERCISE

Part–1

A. Multiple Choice Questions

Q1. Requirement testing ensures that:
A) Code is optimized
B) Requirements are correct, complete, and testable
C) GUI is attractive
D) Database is encrypted

Q2. Requirement defects mostly occur due to:
A) Clear requirements
B) Ambiguity and incompleteness
C) Good reviews
D) Good tools

Q3. A testable requirement should be:
A) Vague
B) Measurable
C) Hidden

D) Unnecessary

Q4. Requirement review is an example of:
A) Dynamic testing
B) Static testing
C) Stress testing
D) Volume testing

Q5. Traceability means:
A) Removing defects
B) Linking requirements to test cases
C) Writing code faster
D) Increasing documentation

Answers with Explanation:

1. **Answer:** B
   Requirement testing checks requirement quality before development.
2. **Answer:** B
   Ambiguous requirements cause misunderstandings and defects.
3. **Answer:** B
   Measurable requirements can be verified by test cases.
4. **Answer:** B
   Requirement review is done without executing software.
5. **Answer:** B
   Traceability ensures each requirement is tested.

B. Small Answer Questions

1. Define requirement testing.
2. What is a requirement defect?
3. What is requirement traceability?
4. Write any two qualities of good requirements.
5. What is the purpose of SRS review?

Part–2

C. Long Answer Type Questions (200 words)

1. Explain requirement testing and its role in defect prevention.
2. Discuss characteristics of good software requirements.
3. Explain requirement traceability matrix (RTM).

NOTE

4. Describe requirement review techniques.
5. Discuss how requirement testing improves software quality.

Part–3

D. Long Answer Type Questions (300 words)

1. Explain requirement testing in detail with suitable examples.
2. Discuss requirement validation and verification methods.
3. Explain RTM preparation and its importance in testing.
4. Discuss requirement defects and their impact on software projects.
5. Explain best practices for requirement testing in Agile projects.

NOTE

UNIT 17
REVIEWS IN TESTING LIFECYCLE

UNIT STRUCTURE

17.1 Introduction
17.2 Objective
17.3 Reviews in testing lifecycle
- 17.3.1 Coverage in Verification
- 17.3.2 Concerns of Verification
- 17.3.3 Validation
- 17.3.4 Validation Workbench
- 17.3.5 Levels of Validation
- 17.3.6 Coverage in Validation

17.4 Summary
17.5 Glossary
17.6 Answer to check your progress
17.7 Suggested readings - OER
17.8 Reference
17.9 Exercise

17.1 INTRODUCTION

In the broad landscape of Software Quality Assurance (SQA), the testing lifecycle is a continuum that begins long before the first line of code is compiled. While the previous unit introduced the concepts of Verification and Validation, Unit 17 drills deeper into the mechanics of these processes, specifically focusing on how we measure their effectiveness and how they are structured across different levels of the system. The title "Reviews in Testing Lifecycle" implies a holistic view where "Review" is not just a meeting but a rigorous assessment mechanism applied to both static documents and dynamic test results.

The software crisis of the late 20th century taught engineers that simply "testing" (executing code) is insufficient because it happens too late. By the time a system is ready for validation, the architectural concrete has already set. Therefore, the lifecycle introduces Reviews as a verification tool to ensure coverage before execution, and Validation levels to ensure correctness during execution. This unit explores the metrics used to

NOTE

determine "how much" verification is enough (coverage), the specific concerns addressed during these reviews, and the structured "Workbench" used to validate the software once it is executable. We will transition from the theoretical inspection of design documents to the practical, multi-tiered execution of code in Unit, Integration, and System testing.

In the previous unit, we explored Verification—the static process of checking documents. In this unit, we transition into the broader lifecycle of reviews and the dynamic world of Validation. While verification ensures the logic is sound, validation ensures the software actually works for the user. This unit connects the static reviews to the dynamic execution levels, providing a complete picture of Quality Assurance.

17.2 OBJECTIVE

By the end of this unit, you should be able to:

- Explain how coverage is measured in the static phase using techniques like Requirements Traceability.
- Identify the specific quality attributes (consistency, completeness, maintainability) looked for during reviews.
- Articulate the shift from building the product *right* to building the *right* product.
- Describe the inputs, tools, and outputs comprising the Validation Workbench.
- Compare and contrast the scope and goals of Unit, Integration, System, and Acceptance testing.
- Apply metrics to assess the thoroughness of dynamic testing efforts.

17.3 REVIEWS IN TESTING LIFECYCLE

The testing lifecycle is often visualized as a "V," but in practice, it is a series of feedback loops. Reviews act as the "gates" in these loops. A review in the testing lifecycle is a formal or informal assessment of a project artifact. However, unlike a simple proofread, these reviews are driven by specific coverage criteria and quality concerns. They serve as the static counterpart to dynamic testing. Reviews are not isolated events; they are checkpoints embedded throughout the Software Development Life Cycle (SDLC). A "Review" is any process where a work product is examined by someone other than the author to identify defects.

NOTE

In the testing lifecycle, reviews occur at three critical junctures:

1. Requirements Phase: Reviewing the Software Requirement Specification (SRS) ensures that testers understand what to test before they write a single test case.
2. Design Phase: Reviewing the System Architecture prevents structural defects that could make the system untestable later.
3. Test Design Phase: Reviewing the Test Plan and Test Cases. This is crucial. If the test cases are wrong, the testing will be useless. A peer review of test cases ensures that the tests actually cover the requirements.

17.3.1 Coverage in Verification

When we speak of "Coverage" in the context of coding (Validation), we usually mean "how many lines of code ran." However, Coverage in Verification is a more abstract but equally critical concept. It addresses the question: "Have we reviewed everything that needs to be reviewed, and does the design cover all requirements?"

Verification coverage is primarily measured through Traceability. The central tool for this is the Requirements Traceability Matrix (RTM). In a verification review, the reviewers do not simply read the Design Document; they map it back to the Requirements Specification. Coverage is achieved when every single requirement is linked to a specific design element. If Requirement R-101 states "Users must be able to reset passwords," and the Design Document has no module or database field corresponding to password reset, then Verification Coverage is incomplete.

Furthermore, verification coverage extends to the Document Structure itself. Reviewers verify that all necessary sections of a standard document are present. For instance, in an Interface Design review, coverage implies checking that every external system interface has a defined protocol, data format, and error-handling mechanism. If the design mentions three external APIs but only defines the data format for two, the verification coverage is partial.

"Coverage" generally asks the question: *"How much have we checked?"* In the context of Verification (Static Testing), coverage refers to the extent to which the documents and code have been reviewed.

Since we cannot run the code during verification, coverage is measured by:

- Document Coverage: Have we reviewed all chapters of the Requirement Specification?

NOTE

- Checklist Coverage: Have we checked every item on the Inspection Checklist (e.g., Security standards, Naming conventions)?
- Traceability Coverage: Does every requirement in the SRS have a corresponding section in the Design Document?
- Code Verification Coverage: What percentage of the source code modules have undergone a peer review or static analysis?

Key Difference in Verification, coverage means "Reviewed." In Validation, coverage means "Executed."

Another aspect of verification coverage is Risk Coverage. Not all parts of a system are equal. A payment gateway module requires 100% rigorous verification coverage (formal inspections), whereas a "Change Color Theme" module might only require a walkthrough. Coverage strategies often dictate that high-risk areas receive higher-density reviews. Verification coverage ensures that the intellectual effort of the review team is distributed effectively across the system's architecture to catch the most expensive defects early.

17.3.2 Concerns of Verification

While "Coverage" tells us *what* to review, "Concerns" tell us *what to look for*. The concerns of verification are the specific quality attributes that reviewers verify against. These are often mnemonicized or listed in checklists to ensure the review is not just subjective opinion but an objective engineering assessment.

Consistency is a primary concern. Reviewers check for internal contradictions. For example, does the Data Dictionary define CustomerID as an 8-digit integer, while the Database Schema defines it as a 10-character string? Does the logic in Module A assume Module B returns a generic error, while Module B is designed to throw specific exceptions? These inconsistencies are fatal if they reach the coding phase.

Completeness is the concern that nothing is missing. This is arguably the hardest concern to verify because it requires the reviewer to see what is *not* there. For example, does the design handle the "Happy Path" (successful transaction) but fail to specify what happens when the network drops? Are all edge cases defined? Verification reviews specifically hunt for "To Be Determined" (TBD) tags or vague statements like "Handle errors appropriately," which indicate incomplete design.

Correctness involves checking the logic against domain rules. If a banking application calculates interest, the verification review must mathematically prove that the formula in the design document matches the financial regulations. This often involves "Desk Checking," where the reviewer manually traces the logic with sample data.

NOTE

Maintainability and Clarity are also major concerns. Code is read more often than it is written. Verification reviews assess whether the design or code is understandable. If a reviewer cannot understand the logic flow of a proposed algorithm during the design phase, it is a defect. Complexity is a concern; if a design is overly convoluted, verification aims to simplify it before it becomes technical debt.

When we perform verification (reviews/inspections), we are concerned with specific attributes of the work product. We are asking "Is it built *right*?"

The three primary concerns are:

1. Consistency: Does the document contradict itself? (e.g., Requirement A says "User must be 18," Requirement B says "User must be 21").
2. Completeness: Is anything missing? (e.g., The design handles the "Success" scenario, but is the "Error" scenario described?).
3. Correctness: Does it follow the standards? (e.g., Does the code follow the Java Naming Conventions? Is the logic mathematically sound?).

17.3.3 Validation

Once the artifacts have passed verification reviews and the code has been written, the lifecycle shifts to Validation. Validation is the dynamic process of executing the software to ensure it meets the user's needs. It is often summarized as: "Are we building the right product?"

Validation differs from verification in that it requires a running environment. It is empirical rather than analytical. In verification, we *predict* the software will work based on logic; in validation, we *observe* it working. Validation is the moment of truth where the software meets the chaotic reality of the real world memory constraints, network latency, unpredictable user inputs, and hardware interrupts.

Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

- The Question: "Are we building the *right* product?"
- The Method: Dynamic Testing (Executing the software).
- The Goal: To demonstrate that the software meets the user's needs in a real-world environment.

Validation is what most people think of when they hear "Software Testing." It involves running the app, clicking buttons, and checking if the output matches the expectation.

NOTE

The scope of validation is broad. It encompasses not just functional correctness (does the button work?) but also non-functional attributes (is it fast enough? is it secure? is it usable?). Validation is the bridge between the development team's technical output and the customer's business value. It is strictly strictly performed against the Requirements Specification (or User Stories). If the software does something "cool" that was not in the requirements, it is technically not validated behavior; it might even be considered 'gold-plating' or a defect.

17.3.4 Validation Workbench

To perform validation effectively, we cannot simply "play around" with the software. We need a structured environment known as the Validation Workbench. This concept parallels the Verification Workbench but deals with executable artifacts.

Inputs:

The primary input is the Source Code or the Executable Build. However, code alone is insufficient. The workbench requires Test Data—specific, carefully crafted data sets designed to trigger both success and failure paths. It also requires Test Cases (the steps to execute) and, crucially, the Expected Results (the Oracle). Without an expected result, testing is meaningless because you cannot determine pass or failure.

The Process (Test Execution):

The core of the workbench is the execution engine. This might be a human tester manually clicking through screens, or it might be an automated framework (like Selenium, JUnit, or Appium) running scripts. The process involves setting up the Pre-conditions (e.g., creating a user in the database), performing the Action (e.g., clicking 'Login'), and observing the Post-conditions (e.g., redirect to dashboard).

Tools:

The validation workbench relies heavily on tools. Test Management Tools (like Jira or TestRail) store the test cases. Automation Tools execute the scripts. Defect Tracking Tools record the failures. Environment Management Tools (like Docker) ensure the test bed is consistent.

Outputs:

The output of the validation workbench is a Test Log or Execution Report. This document details what passed, what failed, and which tests were skipped. The most critical output is the Defect Report (or Bug Report), which provides the developer with the steps to reproduce the failure. Ultimately, the workbench produces a "Release/No-Release" recommendation based on the stability of the build.

17.3.5 Levels of Validation

NOTE

Validation is not a monolithic activity. It is stratified into distinct levels, each targeting a different scope of the system. This is often referred to as the "Testing Pyramid."

1. Unit Testing:

This is the lowest level of validation, typically performed by the developers themselves. The goal is to validate the smallest testable part of an application—usually a single function, method, or class in isolation. Unit tests verify the internal logic of a component. They use "Mock Objects" or "Stubs" to simulate the behavior of dependencies (like databases or APIs) so that the unit can be tested purely on its own logic.

2. Integration Testing:

Once individual units are verified, they must be combined. Integration testing validates the *interfaces* and *interactions* between modules. The concern here is not "does the loop work?" (Unit test) but "does Module A pass the correct data to Module B?" and "does Module B return the expected response?" Approaches include Big Bang (combine everything at once) or Incremental (Top-down or Bottom-up). Most defects found here relate to data format mismatches or API communication errors.

3. System Testing:

This is the first level where the entire application is tested as a whole. System testing validates the fully integrated product against the functional and non-functional requirements. It is performed in an environment that closely mirrors production. This level includes functional testing (business logic), performance testing (load/stress), security testing, and reliability testing. It ignores the internal code structure (Black-box) and focuses on user behaviors.

4. Acceptance Testing (UAT):

The final level is Acceptance Testing, often performed by the client or end-users. The goal is not to find bugs (System testing should have caught them) but to establish confidence. Alpha Testing happens at the developer's site, while Beta Testing happens at the client's site with real data. Success here triggers the contractual acceptance of the software and payment.

17.3.6 Coverage in Validation

Just as we measure coverage in verification, we must measure it in validation. Test Coverage is a metric that quantifies the completeness of the testing effort. It answers: "How much of the system did we actually test?"

Structural Coverage (Code Coverage):

NOTE

This is a white-box metric measured by tools. Statement Coverage measures the percentage of executable lines of code that were run. Branch Coverage is more rigorous; it measures the percentage of "decisions" (True/False branches in IF/WHILE statements) taken. A suite with 100% statement coverage might still miss a branch (e.g., an IF statement with no ELSE). Path Coverage checks all possible paths, though this is often impossible in loops.

Functional Coverage (Requirements Coverage):

This is a black-box metric. It measures the percentage of business requirements that have valid test cases executed against them. If you have 100 requirements and you have only tested 90 of them, your functional coverage is 90%. This is often tracked using the Requirements Traceability Matrix.

Data Coverage:

This ensures that the input fields have been tested with a sufficient variety of data. It includes Boundary Value Analysis (testing edges like 0, -1, 100, 101) and Equivalence Partitioning (testing one representative value from each data class).

It is important to note that 100% coverage does not mean 100% bug-free. Coverage metrics are strictly heuristics; they tell you what you *haven't* tested, but they don't guarantee that what you *have* tested works perfectly under all timing and environmental conditions.

Validation happens in layers, moving from the smallest piece of code to the full system. This hierarchy is often mapped to the V-Model.

1. Unit Validation (Component Testing):

   What: Testing individual functions or methods.

   Who: Developers.

   Focus: Logic, loops, variables.

2. Integration Validation:

   What: Testing the interfaces between units.

   Who: Developers or Testers.

   Focus: Data flow between modules (e.g., Does the Login module correctly send the User ID to the Dashboard module?).

3. System Validation:

   What: Testing the complete, integrated system against the Requirements.

   Who: The QA Team.

NOTE

Focus: Functional and Non-Functional requirements (Performance, Security).

4. User Acceptance Validation (UAT):

   What: The final check by the client.

   Who: The End-Users / Customer.

   Focus: Business process fitness. "Can I use this to do my job?"

Activity Title: Conducting a Peer Review

Objective:
To understand the importance of reviews in identifying defects before testing.

Materials Required:

- Sample requirement/code/document
- Review checklist
- Notebook

Procedure/Steps:

1. Select a short software requirement or program segment.
2. Form a small peer-review group.
3. Examine the document for errors, omissions, ambiguity, and inconsistencies.
4. Record the identified issues.
5. Discuss corrective actions.
6. Prepare a short review report.

Outcome:
The learner will gain practical understanding of peer reviews and their role in early defect detection.

Check Your Progress

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. Reviews in testing lifecycle are mainly conducted to:
A) Increase program size
B) Detect defects early in documents and code

NOTE

C) Replace customer feedback
D) Avoid documentation

_____

Q2. Reviews are a part of static testing techniques.

_____

Q3. Validation mainly answers the question: "Are we building the _____ product?"

_____

Q4. Write any two levels of validation.

_____

Q5. Coverage in verification mainly refers to:
A) How many users installed the software
B) Extent to which verification activities cover requirements, design, and code
C) Internet network coverage
D) Amount of money spent

17.4 SUMMARY

The unit has detailed the critical role of Reviews and Validation in the testing lifecycle. We established that Verification utilizes reviews to ensure coverage through traceability and addresses concerns like consistency and completeness before code is written. We then transitioned to Validation, the dynamic execution of software. We explored the Validation Workbench, a structured environment of inputs (test cases), tools, and outputs (bug reports). We broke down validation into hierarchical levels: Unit Testing (logic), Integration Testing (interfaces), System Testing (end-to-end), and Acceptance Testing (user confidence). Finally, we discussed Coverage in Validation, emphasizing that while metrics like Statement and Branch coverage are vital indicators of thoroughness, they must be combined with functional requirements coverage to ensure a truly robust software product.

Check Your Progress

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Fill-in-the-Blank Questions:

NOTE

1. The document that maps requirements to design elements to ensure verification coverage is called the _____.
2. Testing individual components or modules in isolation is known as _____ testing.
3. _____ coverage measures the percentage of decision points (True/False) executed in the code.

True/False Questions:

1. Validation is primarily concerned with "Are we building the product right?"
2. Integration testing focuses on the interfaces and data flow between modules.
3. If Statement Coverage is 100%, the software is guaranteed to be bug-free.

17.5 GLOSSARY

- Traceability Matrix: A document, usually a table, that correlates any two baselined documents that require a many-to-many relationship to determine the completeness of the relationship (e.g., Requirements to Test Cases).
- Validation Workbench: The integrated set of tools, environment, inputs, and processes used to perform dynamic software testing.
- Unit Testing: The testing of individual software components or modules, typically done by the developer using white-box techniques.
- Integration Testing: Testing performed to expose defects in the interfaces and interaction between integrated components or systems.
- System Testing: Testing the complete, integrated system to evaluate the system's compliance with its specified requirements.
- Branch Coverage: A testing metric that indicates whether every branch (decision outcome) of the control structure has been executed at least once.

NOTE

17.6 ANSWER TO CHECK YOUR PROGRESS

Q1. Answer: B
Explanation: Reviews help identify defects early in requirements, design, and code before testing begins, reducing cost and effort.

Q2. Answer: True
Explanation: Reviews are performed without executing the program and are considered static testing methods.

Q3. Answer: Right
Explanation: Validation ensures the software meets customer expectations and real-world needs.

Q4. Explanation: Validation is performed at different levels to ensure the product works correctly as per user requirements.

Q5. Answer: B
Explanation: Coverage ensures that verification activities are performed on all important artifacts such as requirements, design documents, and code modules.

17.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3796	

17.8 REFERENCE

- Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill Education.
- Sommerville, I. (2015). *Software Engineering* (10th ed.). Pearson.
- Myers, G. J. (2011). *The Art of Software Testing* (3rd ed.). Wiley.
- Beizer, B. (1990). *Software Testing Techniques*. Van Nostrand Reinhold.

17.9 EXERCISE

NOTE

A. Objective Type Questions

Q1. Design testing mainly focuses on:
A) Execution speed only
B) Checking correctness of software design
C) Hardware purchase
D) Marketing plan

Q2. Design reviews are part of:
A) Static testing
B) Dynamic testing
C) Stress testing
D) Security testing

Q3. High coupling in design results in:
A) Better maintainability
B) Poor maintainability
C) Faster execution always
D) No defects

Q4. A design defect may lead to:
A) Only GUI issues
B) Major system failures
C) No impact
D) Better performance

Q5. Cohesion refers to:
A) Interdependence between modules
B) Strength of relationship inside a module
C) Number of developers
D) Testing tools

1. **Answer:** B
   Design testing ensures design meets requirements and quality.
2. **Answer:** A
   Design is evaluated without executing code.
3. **Answer:** B
   High coupling increases complexity and defects.
4. **Answer:** B
   Poor design causes reliability and performance issues.
5. **Answer:** B
   High cohesion is desirable.

B. Small Answer Questions

NOTE

1. Define design testing.
2. What is a design review?
3. Write two design quality attributes.
4. Define coupling and cohesion.
5. Why is design testing important?

Part–2

C. Long Answer type Questions (200 words)

1. Explain design testing and its importance.
2. Discuss design defects and their consequences.
3. Explain design review process.
4. Discuss coupling and cohesion in design quality.
5. Explain how design testing supports SQA.

Part–3

D. Long Answer type Questions (200 words)

1. Explain design testing in detail with examples.
2. Discuss methods used for design verification and validation.
3. Explain how design testing improves maintainability and reliability.
4. Discuss common design issues and how to detect them.
5. Explain design metrics and their role in quality assessment.

NOTE

UNIT 18
ACCEPTANCE TESTING

UNIT STRUCTURE

18.1 Introduction
18.2 Objective
18.3 Acceptance Testing
18.3.1 Management of Verification and Validation
18.3.2 Software development verification and validation activities
18.4 Summary
18.5 Glossary
18.6 Answer to check your progress
18.7 Suggested readings - OER
18.8 Reference
18.9 Exercise

18.1 INTRODUCTION

In the vast ecosystem of software engineering, Acceptance Testing represents the final hurdle before a system is deployed to the production environment or delivered to the customer. While previous testing phases such as Unit, Integration, and System Testing—are primarily technical exercises conducted by the development and quality assurance teams, Acceptance Testing is distinct in its audience and intent. It is the phase where the focus shifts from "detecting defects" to "establishing confidence."

Acceptance Testing is a formal testing process conducted to determine whether a system satisfies its acceptance criteria and to enable the user, customer, or other authorized entity to determine whether to accept the system. It is the ultimate validation activity. The success of this phase, however, is not accidental. It is the result of a rigorously managed Verification and Validation (V&V) process that spans the entire Software Development Life Cycle (SDLC). If the V&V activities in the earlier phases are mismanaged, Acceptance Testing becomes a bottleneck of failures rather than a gateway to release.

In the final stages of the software development lifecycle, the focus shifts from finding bugs (defect detection) to demonstrating that the system is ready for use (validation). This unit covers Acceptance Testing, the crucial "green light" phase where the customer decides if the software is fit for purpose. Furthermore, we will step back to look at the

NOTE

Management of Verification and Validation (V&V) how these two massive parallel activities are coordinated, managed, and audited to ensure the project stays on track.

This unit explores the theoretical underpinnings of Acceptance Testing, not just as an isolated event, but as the culmination of a managed V&V strategy. We will delve into how V&V is organized, planned, and executed across different development stages to ensure that when the software finally reaches the user for acceptance, it is truly ready for business use.

18.2 OBJECTIVE

After studying this unit, you should be able to:

- Define Acceptance Testing and distinguish it from System Testing.
- Explain the role of the customer in the Acceptance Testing process.
- Describe the management strategies required to coordinate V&V activities.
- Outline the specific V&V activities that occur during each phase of software development.
- Visualize the interaction between V&V management and the technical development process.

18.3 ACCEPTANCE TESTING

Acceptance Testing, often referred to as User Acceptance Testing (UAT), is the formal process of validating the software against the business requirements in a realistic environment. Unlike system testing, which focuses on finding bugs, acceptance testing focuses on verifying that the software is "fit for purpose." It answers the question: "Can this system support the day-to-day business processes of the user?"

There are several forms of acceptance testing. User Acceptance Testing (UAT) involves the actual end-users testing the software to ensure it handles real-world tasks. Operational Acceptance Testing (OAT) focuses on the readiness of the system from an IT perspective, checking backups, disaster recovery, and maintenance procedures. Contract Acceptance Testing is performed against the specific criteria defined in a legal contract between the vendor and the client.

Acceptance Testing is a level of software testing where a system is tested for acceptability. The purpose of this test is to evaluate the system's

NOTE

compliance with the business requirements and assess whether it is acceptable for delivery.

Unlike System Testing, which is performed by the internal QA team to find technical faults, Acceptance Testing is typically performed by the Client or End-Users. It is not about "crashing" the software; it is about "using" the software to conduct business.

Types of Acceptance Testing:

1. User Acceptance Testing (UAT): The end-users test the software in a real-world scenario to ensure it handles day-to-day tasks.
2. Operational Acceptance Testing (OAT): The IT admins test the backups, disaster recovery, and maintenance procedures.
3. Contract Acceptance Testing: Verification that specific criteria defined in the legal contract have been met.
4. Alpha and Beta Testing:

   Alpha: Performed by users at the developer's site (controlled environment).

   Beta: Performed by users at their own sites (uncontrolled, real-world environment).

Two common strategies for acceptance testing are Alpha and Beta testing. Alpha Testing takes place at the developer's site. It involves a select group of users performing tasks in a controlled environment, often with the developers looking over their shoulders to observe errors. Beta Testing, conversely, takes place at the customer's site. The software is released to a limited audience who use it in an uncontrolled, real-world setting and report feedback. This helps uncover environmental issues and usability problems that the internal team could not predict.

18.3.1 Management of Verification and Validation

Acceptance testing is the final step of Validation. However, ensuring a successful acceptance phase requires the careful management of V&V activities from day one. Management of V&V is the discipline of planning, organizing, and controlling the quality assurance efforts to ensuring the software is built correctly and performs the right functions.

Managing V&V is not just about scheduling tests. It involves planning, reporting, and monitoring the quality process across the entire lifecycle. The V&V Manager is responsible for ensuring that the V&V activities are independent, adequately funded, and rigorous.

Key Management Responsibilities:

NOTE

1. Software Verification and Validation Plan (SVVP): Creating a master document that defines *what* will be verified/validated, *how* (methods like inspection or testing), and *when*.
2. Independence: Ensuring the V&V team has technical and managerial independence from the development team. They should not report to the Lead Developer (conflict of interest).
3. Metrics and Reporting: Collecting data (e.g., Defect Density, Review Efficiency) and reporting the "health" of the software to stakeholders.
4. Defect Management: overseeing the lifecycle of defects from discovery to closure, ensuring no critical bug is ignored.

The Software Verification and Validation Plan (SVVP)

The cornerstone of V&V management is the Software Verification and Validation Plan (SVVP). This is a formal document (often following IEEE Standard 1012) that outlines the V&V tasks for the project. The SVVP is not merely a schedule; it is a strategic blueprint. It defines the *organization* of the V&V effort, specifying who is responsible for what. Crucially, it establishes the degree of independence required. For critical software (like avionics or medical devices), Independent Verification and Validation (IV&V) is often mandated, where the V&V team is administratively and financially separate from the development team to prevent conflict of interest.

Resource and Schedule Management

Managing V&V involves allocating resources—both human and computational. The manager must decide which tools (static analyzers, test automation frameworks) are needed and ensure the team is trained. Scheduling is equally vital; V&V activities must be synchronized with development milestones. For instance, the "Requirements Review" (a verification activity) must be scheduled immediately after the requirements are drafted and *before* design begins. If this management fails, errors propagate downstream, making the final Acceptance Testing a disaster.

Metrics and Reporting

Effective management relies on data. V&V managers track metrics such as defect density, requirement coverage, and review efficiency. These metrics provide an early warning system. If the "Code Verification" phase yields an unusually high number of logic errors, the manager knows that the subsequent "System Testing" and "Acceptance Testing" phases are at risk. This allows for corrective action, such as re-training developers or revising the design, long before the software reaches the client.

NOTE

18.3.2 Software development verification and validation activities

To ensure that Acceptance Testing is successful, V&V activities must be embedded into every phase of the software lifecycle. These activities form a "chain of quality" that leads to the final deliverable.

1. Concept and Requirements Phase Activities

The journey to acceptance begins here. The primary V&V activity is the Requirements Review. This is a verification task where the requirements are analyzed for ambiguity, consistency, and testability. If a requirement is not testable, it cannot pass Acceptance Testing. Management ensures that a "Traceability Matrix" is initiated here, linking every requirement to future test cases. This ensures that when UAT starts, no business rule is forgotten.

2. Design Phase Activities

During the design phase, verification activities focus on the architecture. Design Inspections are conducted to ensure the proposed system structure can handle the load and security requirements defined earlier. For example, if the acceptance criteria state the system must support 10,000 concurrent users, the V&V team verifies the architectural diagrams to ensure the database and server configuration can theoretically support that load. This is "static validation" of the design.

3. Implementation (Coding) Phase Activities

As code is written, V&V activities become more granular. Code Walkthroughs and Peer Reviews are performed to catch syntax and logic errors. Unit Testing is the first layer of dynamic validation. Management ensures that unit tests cover all critical paths. This prevents small, technical bugs from blocking the users during Acceptance Testing. If the software crashes every time a user logs in, they cannot perform UAT; adequate implementation phase V&V prevents this embarrassment.

4. Test Phase Activities

This phase precedes acceptance and includes Integration and System Testing. Here, the distinct modules are combined, and the system is tested as a whole. The V&V activity focuses on "System Validation" ensuring the system meets the functional specifications. This is the rehearsal for Acceptance Testing. Management monitors the "Defect fix rate" here; the software cannot move to the Acceptance phase until the severity of open bugs drops below a defined threshold (e.g., "Zero Critical Defects").

5. Installation and Checkout Phase (Acceptance)

NOTE

Finally, the lifecycle reaches the Acceptance phase. The V&V activity here is the User Acceptance Test (UAT) itself. The V&V team facilitates the process, providing users with test scripts and recording the results. They also perform the Configuration Audit, verifying that the software installed in the production environment is exactly the same version that was tested and approved. This prevents "version creep" where untested code accidentally slips into the live system.

Activity Title: Designing Acceptance Test Cases

Objective:
To understand acceptance testing from the perspective of end-user requirements.

Materials Required:

- Sample application
- User requirements
- Test-case template

Procedure/Steps:

1. Select an application such as a college admission system.
2. Identify major user requirements.
3. Convert the requirements into acceptance test cases.
4. Define expected outcomes for each test case.
5. Execute or simulate the test cases.
6. Determine whether the system satisfies the stated requirements.

Outcome:
The learner will be able to develop basic acceptance test cases based on user requirements.

Check Your Progress

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. Acceptance testing is mainly performed to:
A) Check code formatting
B) Confirm the system meets business and user requirements
C) Test database indexing only
D) Increase software complexity

NOTE

_____

Q2. Acceptance testing is generally performed by end users or customers.

_____

Q3. Acceptance testing is performed before the software is finally _____ to the customer.

_____

Q4. Write any two types of acceptance testing.

Answer (Any two):

User Acceptance Testing (UAT)

Alpha Testing

Beta Testing

Contract Acceptance Testing

Regulatory Acceptance Testing

_____

Q5. Management of verification and validation activities mainly ensures:
A) No testing is done
B) Proper planning, control, and reporting of V&V tasks
C) Only coding is performed
D) Only UI is improved

_____

_____

18.4 SUMMARY

Acceptance Testing is the final, decisive phase in the software delivery process, determining whether the product is ready for deployment. Unlike earlier testing phases that focus on technical correctness, Acceptance Testing focuses on business suitability and user confidence. We explored how this phase is not an isolated event but the successful conclusion of a managed Verification and Validation (V&V) process.

We examined the Management of V&V, highlighting the importance of the SVVP (Software Verification and Validation Plan) and the need for independence (IV&V) to ensure objectivity. Furthermore, we traced the V&V Activities across the lifecycle. We learned that rigorous requirements reviews, design inspections, and unit testing are essentially "pre-acceptance" activities. They filter out defects early,

NOTE

ensuring that when the users finally touch the system during UAT, they are validating a mature, stable product rather than debugging a broken one.

Check Your Progress

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Fill-in-the-Blank Questions:

1. Testing performed at the developer's site by a select group of users is called _____ testing.
2. The document that outlines the organization, schedule, and resources for V&V activities is called the _____.
3. _____ Acceptance Testing focuses on the readiness of the system from an IT perspective, such as backups and maintenance.

True/False Questions:

1. Acceptance Testing is primarily meant to find syntax errors in the code.
2. IV&V stands for Internal Verification and Validation.
3. V&V activities should only begin after the coding phase is complete.

18.5 GLOSSARY

- Acceptance Testing: Formal testing with respect to user needs, requirements, and business processes, conducted to determine whether or not a system satisfies the acceptance criteria.
- Alpha Testing: Operational testing by potential users/customers or an independent test team at the developers' site.
- Beta Testing: Installation and use of the software in the real-world environment by a selected group of customers to gather feedback before full release.
- SVVP (Software Verification and Validation Plan): A comprehensive document that describes the work products to be

NOTE

verified and validated, the methods to be used, and the resources required.

- IV&V (Independent Verification and Validation): V&V processes performed by an organization that is technically, managerially, and financially independent of the development organization.
- V&V Management: The planning, control, and monitoring of verification and validation tasks throughout the SDLC.

18.6 ANSWER TO CHECK YOUR PROGRESS

Q1. Answer: B

Explanation: Acceptance testing ensures the software is ready for delivery and satisfies customer needs.

Q2. Answer: True

Explanation: Acceptance testing is usually done by customers or user representatives to verify that the system is suitable for real use.

Q3. Answer: Delivered

Explanation: This testing ensures that the product is acceptable before final release or delivery.

Q4. Explanation: Acceptance testing may vary depending on customer agreement, user involvement, and legal/regulatory requirements.

Q5. Answer: B

Explanation: Managing V&V ensures that verification and validation activities are systematic, planned, and properly executed.

18.7 SUGGESTED READINGS - OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/User/UnitUrls?unitId=3797	

NOTE

18.8 REFERENCE

- Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill Education.
- Sommerville, I. (2015). *Software Engineering* (10th ed.). Pearson.
- Hambling, B., & van Goethem, P. (2013). *User Acceptance Testing: A Step-by-Step Guide*. BCS, The Chartered Institute for IT.
- IEEE Computer Society. (2014). *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide)*.

18.9 EXERCISE

Part–1

A. Objective Type Questions

Q1. Code review is performed to:
A) Increase code size
B) Detect defects early in code
C) Replace testing
D) Avoid documentation

Q2. Code review is an example of:
A) Dynamic testing
B) Static testing
C) Stress testing
D) Volume testing

Q3. Peer review means:
A) Customer reviews code
B) Team members review code
C) Only manager reviews code
D) Only testers review code

Q4. One key benefit of code review is:
A) Higher defect leakage
B) Improved maintainability
C) Increased cost
D) Reduced documentation

Q5. Code review checklist is used to:
A) Write code faster

NOTE

B) Ensure systematic review
C) Avoid defects
D) Replace unit testing

Answers with Explanation:

1. **Answer:** B
   Code review finds bugs before execution.
2. **Answer:** B
   Code is reviewed without execution.
3. **Answer:** B
   Peer review is performed by colleagues
4. **Answer:** B
   Review improves readability and maintainability.
5. **Answer:** B
   Checklist helps reviewers cover key points.

B. Small Answer Questions

1. Define code review.
2. Write any two benefits of code review.
3. What is peer review?
4. What is a review checklist?
5. Why is code review important?

Part–2

C. Long Answer Type Questions (200 words)

1. Explain code review and its objectives.
2. Discuss different types of code reviews.
3. Explain the code review process.
4. Discuss the role of checklists in code review.
5. Explain how code review improves software quality.

Part–3

D. Long Answer Type Questions (200 words)

1. Explain code review in detail with examples.
2. Discuss formal and informal review techniques.
3. Explain best practices for effective code review.
4. Discuss common coding defects detected by reviews.

5. Explain how code review supports defect prevention and quality assurance.

NOTE

UNIT 19
V-TEST MODEL

UNIT STRUCTURE

19.1 Introduction
19.2 Objective
19.3 V-test Model
19.3.1 Introduction
19.3.2 V-model for software
19.3.3 Testing during Proposal stage
19.3.4 Testing during requirement stage
19.4 Summary
19.5 Glossary
19.6 Answer to check your progress
19.7 Suggested readings - OER
19.8 Reference
19.9 Exercise

19.1 INTRODUCTION

In the historical progression of software engineering methodologies, the V-Model (often referred to as the V-test Model) emerged as a direct response to the limitations of the traditional Waterfall Model. In the Waterfall approach, testing was frequently relegated to a distinct phase at the very end of the development lifecycle. This "testing-last" philosophy often resulted in a catastrophic discovery of fundamental design or requirement errors only days before the deadline, leading to cost overruns and delayed releases. The industry realized that quality cannot be "tested in" at the end; it must be integrated throughout the process.

In traditional software development, testing was often treated as an "afterthought"—something that happened only after the code was written. This approach proved to be expensive and risky, as fundamental design errors were discovered too late. The V-Model (Verification and Validation Model) revolutionized this by demonstrating that testing is not a single phase but a continuous activity that mirrors every stage of development. This unit explores the V-Model structure and emphasizes the often-overlooked importance of testing during the very earliest stages: the Proposal and Requirement phases.

NOTE

The V-Model, which stands for Verification and Validation Model, fundamentally reshapes the lifecycle by visualizing the software development process as a "V" shape rather than a linear cascade. It demonstrates the relationships between each phase of the development life cycle and its associated phase of testing. Ideally, for every distinct phase of development on the left-hand side of the "V" (Requirements, Design, Code), there is a corresponding testing phase on the right-hand side (Acceptance, System, Unit). This symmetry enforces the discipline that testing is not an afterthought but a parallel activity that begins the moment a project is conceived.

This unit explores the theoretical structure of the V-Model, dissecting its sequential execution of verification and validation. Furthermore, we will delve into the often-overlooked "early testing" activities that occur during the proposal and requirements stages—critical periods where the cost of finding a defect is virtually zero compared to finding it in production.

19.2 OBJECTIVE

After studying this unit, you should be able to:

- Explain the structure and philosophy of the V-Model.
- Correlate development phases (left side of the V) with their corresponding testing phases (right side of the V).
- **Explain** the relationship between development and testing phases in the V-Model.
- **Identify** the testing activities associated with different development stages.
- **Differentiate** the V-Model from other software development approaches.
- **Analyse** the advantages and limitations of the V-Model.Apply verification techniques to the Requirement stage to prevent defects early.

19.3 V-TEST MODEL

19.3.1 Introduction

The V-Model is an SDLC model where execution of processes happens in a sequential manner in a V-shape. It is also known as the Verification and Validation model. It is based on the association of a testing phase for each corresponding development stage. Development of each step is directly associated with the testing phase. The next phase starts only

NOTE

after completion of the previous phase, for example, requirements to design, design to coding.

The fundamental principle of the V-Model is that the "Left Side" of the V represents the decomposition of requirements and the creation of system specifications (Verification), while the "Right Side" of the V represents the integration of parts and their verification against specifications (Validation). The bottom of the V represents the coding phase. By connecting these two sides, the model emphasizes that the tests for a specific phase are derived directly from the artifacts produced in the corresponding development phase. For instance, System Test cases are written based on the System Requirements Specification, not just on the code.

19.3.2 V-model for software

The architecture of the V-Model is strictly hierarchical. It ensures that testing addresses multiple levels of granularity, from the smallest unit of code to the overarching business logic.

The Left Side: The Specification Stream

The left side of the model descends from the abstract to the concrete.

1. User Requirement Specification (URS): This is the highest level, capturing the user's business needs.
2. System Requirement Specification (SRS): The technical translation of business needs into system functions.
3. High-Level Design (HLD): The architecture of the system, defining modules, interfaces, and data flow.
4. Low-Level Design (LLD): The internal logic of each module (algorithms, database tables).

The Right Side: The Testing Stream

The right side ascends from the concrete back to the abstract, verifying the work done on the left.

1. Unit Testing: Corresponds to Low-Level Design. It verifies that individual snippets of code function correctly according to the logic defined in the LLD.
2. Integration Testing: Corresponds to High-Level Design. It verifies that the modules communicate correctly as defined in the system architecture.
3. System Testing: Corresponds to the System Requirement Specification (SRS). It validates that the fully integrated system meets the functional and non-functional requirements.

NOTE

4. Acceptance Testing: Corresponds to the User Requirement Specification (URS). It validates that the system solves the business problem originally stated by the user.

The "V" Connection

The most important theoretical aspect of this model is the horizontal lines connecting the two sides. These lines represent Early Test Design. When the requirements are written (top left), the Acceptance Tests are designed *immediately*. We do not wait for the code to be finished. By designing the tests early, we validate the requirements themselves. If you cannot write a test case for a requirement, the requirement is likely ambiguous or flawed. This process of "testing the requirements" prevents defects from ever entering the design phase.

19.3.3 Testing during Proposal stage

While standard literature often starts the V-Model at "Requirements," in a comprehensive quality assurance lifecycle, testing begins even earlier: at the Proposal or Pre-project stage. A software project usually starts with a Request for Proposal (RFP) or a business case. If the underlying premise of the project is flawed, no amount of coding excellence can save it.

Testing during the proposal stage involves Static Verification of the project concept. The "testing team" (which at this stage may comprise senior analysts and QA managers) reviews the proposal documents. The primary objective is to verify Feasibility and Risk.

One critical activity is reviewing the Budget and Schedule estimations. Historical data is used to "test" whether the proposed timeline is realistic. If a proposal promises to deliver a complex banking system in two months, the "test" fails, and the proposal is flagged as high-risk. Accepting such a proposal without verification sets the project up for failure before it begins.

Most people assume testing starts with requirements. However, in a vendor-client relationship, testing begins at the Proposal / Contract Stage.

Why test a proposal? If a software company promises unrealistic features or timelines in a proposal to win a contract, the project is doomed before it begins.

Testing Activities:

1. Feasibility Analysis: Can we actually build what the sales team is promising? (e.g., "AI-based prediction with 100% accuracy" is technically impossible; QA must flag this).

NOTE

2. Schedule Verification: Is the timeline realistic? If the proposal says "Delivery in 3 months" but historical data suggests 6 months, QA raises a risk flag.
3. Resource Check: Do we have the skilled staff required? (e.g., If the proposal requires Blockchain experts, and we have none, this is a defect in the proposal).

Outcome: The "Defect" found here is a modification to the contract or proposal text to make it realistic and achievable.

Another aspect is Contract Review. The acceptance criteria defined in the proposal (the contract) act as the ultimate test plan. Reviewers check these criteria for clarity. For example, if a contract says "The system must handle high traffic," the reviewer flags this as a defect because "high traffic" is undefined. The proposal must be corrected to say "The system must handle 10,000 concurrent users." This is testing applied to legal and business documents.

19.3.4 Testing during requirement stage

The Requirement Stage is the most critical phase in the V-Model for defect prevention. Studies show that a defect introduced in the requirements phase (e.g., a misunderstood business rule) is 100 times more expensive to fix if found in production than if found immediately. Therefore, "Testing" during this stage is rigorous and formal.

Static Testing of Requirements

Since there is no executable code yet, testing takes the form of Reviews, Walkthroughs, and Inspections (as detailed in previous units). The QA team analyzes the Software Requirement Specification (SRS) for specific attributes:

1. **Ambiguity:** Are there requirements with words like "fast," "user-friendly," or "efficient"? These are untestable and must be rejected.
2. **Completeness:** are there missing scenarios? For example, the SRS defines how to create a user, but does it define how to delete one?
3. **Contradiction:** Does Requirement A (allow unlimited uploads) contradict Requirement B (server storage limit is 5GB)?

Generation of Acceptance Test Cases

The most powerful technique for testing requirements is the act of writing User Acceptance Test (UAT) cases. When a tester attempts to write a step-by-step test case for a requirement, logical gaps become immediately apparent.

NOTE

For instance, if a requirement states "The system shall calculate tax based on regional laws," the tester will ask, "Which regions? What are the specific rates for this year? What happens if the region is undefined?" These questions, raised *during* the requirements phase, force the business analysts to clarify the specifications. In this way, the act of preparing for testing actually debugs the requirements document itself.

Activity Title: Mapping the V-Model

Objective:
To understand the relationship between development stages and corresponding testing stages in the V-Model.

Materials Required:

- Chart paper/notebook
- Pen/pencil
- Reference material

Procedure/Steps:

1. Draw a V-shaped diagram representing the V-Model.
2. Write the development phases on the left side.
3. Write the corresponding testing phases on the right side.
4. Connect each development phase with its corresponding testing activity.
5. Explain the purpose of each connection.
6. Discuss the advantages and limitations of the V-Model.

Outcome:
The learner will be able to explain and visually represent the V-Model and its testing relationships.

Check Your Progress 19.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. The V-test model mainly shows the relationship between:
A) Coding and marketing
B) Development phases and corresponding testing phases
C) Hardware and software
D) Database and network

_____

Q2. In the V-model, testing activities start only after coding is completed.

_____

Q3. In the V-model, requirement stage is associated with _____ testing.

_____

Q4. Write any two advantages of the V-test model.

Answer (Any two):

Testing is planned early in SDLC

Clear mapping between development and testing phases

Early defect detection

Suitable for small and medium projects with stable requirements

_____

Q5. Testing during the requirement stage mainly involves:
A) Unit testing
B) Reviewing and validating requirements
C) GUI testing
D) Performance testing

_____

_____

Fill-in-the-Blank Questions:

In the V-Model, the _____ phase is validated by System Testing.

The V-Model is also known as the _____ Model.

Testing performed during the proposal stage to check if the project can be done within budget and time is called a _____ study.

True/False Questions:

In the V-Model, test cases for acceptance testing are written after the coding is complete.

The left side of the V-Model represents the Verification stream.

Ambiguity in requirements can be detected by attempting to design test cases early.

NOTE

19.4 SUMMARY

The V-Model (V-test Model) represents a disciplined, structured approach to software development that elevates testing to a status equal to that of coding. Unlike the linear Waterfall model, the V-Model emphasizes the parallel execution of verification and validation. We explored the structure of the model, noting how the Left Side (Requirements, Design) corresponds directly to the Right Side (Acceptance, Unit Testing).

We delved into the nuances of early-stage testing. We learned that testing begins during the Proposal Stage, where feasibility and contractual criteria are verified to ensure the project is viable. We also examined the Requirement Stage, identifying it as the most high-leverage point in the lifecycle. By performing static analysis on the SRS and designing acceptance tests immediately, teams can identify and resolve logical contradictions and ambiguities. This "Shift Left" approach moving testing earlier in the timeline—is the central theoretical contribution of the V-Model to modern software engineering.

19.5 CHECK YOUR PROGRESS

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

19.5 GLOSSARY

- **V-Model:** A software development lifecycle model where execution of processes happens in a sequential manner in a V-shape, emphasizing the relationship between development phases and testing phases.
- **Verification:** The process of evaluating work-products (like documents) of a development phase to ensure they meet the specified requirements of that phase (The Left Side of the V).
- **Validation:** The process of evaluating the final software system to ensure it meets the user's requirements (The Right Side of the V).
- **SRS (Software Requirement Specification):** A document that describes what the software will do and how it will be expected to perform.
- **Proposal Stage Testing:** The evaluation of project feasibility, risk, and contractual definitions before the official project kickoff.

NOTE

- **Static Testing:** Testing that involves reviewing documents and models without executing code.

19.6 ANSWER TO CHECK YOUR PROGRESS

Q1. Answer: B

Explanation: The V-model connects each development stage (requirements, design, coding) with a corresponding testing stage.

Q2. Answer: False

Explanation: In the V-model, test planning begins early and is linked to development phases, even before coding is finished.

Q3. Answer: Acceptance

Explanation: Requirements are validated through acceptance testing to ensure the final system meets customer needs.

Q4. Explanation: The V-model improves quality by integrating testing with each development stage.

Q5. Answer: B

Explanation: Requirement stage testing focuses on verifying correctness, completeness, and clarity of requirements through reviews and analysis.

19.7 SUGGESTED READINGS - OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3798	

19.8 REFERENCE

- Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill Education.
- Sommerville, I. (2015). *Software Engineering* (10th ed.). Pearson.
- Mathur, A. P. (2008). *Foundations of Software Testing*. Pearson Education India.

NOTE

- Burnstein, I. (2003). *Practical Software Testing*. Springer.

19.9 EXERCISE

Part–1

A. Objective Type Questions

Q1. Unit testing is performed to test:
A) Complete system
B) Individual modules/functions
C) Database only
D) Network only

Q2. Unit testing is usually done by:
A) Customers
B) Developers
C) End users
D) Auditors

Q3. Unit testing mainly detects:
A) Coding defects
B) Requirement defects only
C) Hardware faults
D) Marketing issues

Q4. A stub is used in:
A) System testing
B) Unit/Integration testing
C) Recovery testing
D) Security testing

Q5. A driver is used to:
A) Replace database
B) Call the module under test
C) Remove test cases
D) Encrypt code

1. **Answer:** B
   Explanation: Unit testing verifies small components.
2. **Answer:** B
   Developers test code units before integration.
3. **Answer:** A
   Unit testing finds logic and coding errors.
4. **Answer:** B
   Stubs simulate called modules.

NOTE

5. **Answer:** B
   Drivers simulate calling modules.

B. Small Answer Questions

1. Define unit testing.
2. Write two objectives of unit testing.
3. What is a stub?
4. What is a driver?
5. Write any two unit testing techniques.

Part–2

C. Long Answer Type Questions (300 words)

1. Explain unit testing and its importance.
2. Discuss unit test case design.
3. Explain stubs and drivers with examples.
4. Discuss unit testing tools and automation.
5. Explain advantages and limitations of unit testing.

Part–3

D. Long Answer Type Questions (300 words)

1. Explain unit testing in detail with suitable examples.
2. Discuss unit testing strategies and best practices.
3. Explain the role of unit testing in defect prevention.
4. Discuss unit testing in Agile and DevOps environments.
5. Explain how unit testing improves maintainability and reliability.

NOTE

UNIT 20
TESTING DURING TEST PLANNING PHASE

UNIT STRUCTURE

20.1 Introduction
20.2 Objective
20.3 Testing during test planning phase
  20.3.1 Testing during design phase
  20.3.2 Testing during coding
  20.3.3 VV Model
  20.3.4 Critical Roles and Responsibilities
20.4 Summary
20.5 Glossary
20.6 Answer to check your progress
20.7 Suggested readings - OER
20.8 Reference
20.9 Exercise

20.1 INTRODUCTION

In traditional software development views, "Test Planning" and "Testing" were often seen as sequential steps: first you plan, then you code, and finally, you test. However, modern Software Quality Assurance (SQA) recognizes that this linear approach is inefficient and risky. A significant portion of "testing activity" actually occurs parallel to the planning, design, and coding phases. This concept is often referred to as "Shift Left" moving quality assurance activities earlier in the timeline.

In the comprehensive study of software quality assurance, it is vital to understand that "testing" is not a solitary activity that occurs after the code is written. Instead, it is a pervasive discipline that permeates every stage of the software development life cycle. From the moment a test plan is drafted to the final line of code written, verification activities are taking place to ensure alignment with requirements. This unit examines the specific testing activities that occur during the planning, design, and coding phases. Furthermore, it revisits the V-Model to solidify the relationship between development and verification, and finally, it delineates the critical human roles required to manage and execute these complex tasks effectively.

NOTE

The phrase "Testing during test planning phase" might sound contradictory, but it refers to the Verification activities and the Test Design efforts that take place while the project is still being planned and constructed. While the project manager is planning the schedule and the architects are planning the design, the test team is not idle. They are actively "testing" the plans and designs through reviews and analysis. They are also preparing the test infrastructure. If testing only begins after the code is delivered, the project is already behind schedule. This unit explores the critical quality assurance activities that run concurrently with the early stages of development, ensuring that when the actual test execution begins, the strategy is sound and the major structural defects have already been identified.

20.2 OBJECTIVE

After studying this unit, you should be able to:

- Analyze the verification activities performed during the Test Planning phase.
- Explain the importance of testing design documents (HLD and LLD) before implementation begins.
- Describe the static and dynamic testing methods applied during the coding phase.
- Articulate the structure and philosophy of the VV (Verification and Validation) Model.
- Define the critical roles and responsibilities within a testing team, including the Test Manager, Test Lead, and Test Engineer.

20.3 TESTING DURING TEST PLANNING PHASE

The core philosophy of this unit is that Quality Assurance is a continuous process, not a distinct phase. When the development team is in the "Planning" or "Construction" mode, the QA team is simultaneously in the "Verification" mode. This section breaks down these activities by phase.

The Test Planning phase is often mistaken for a purely administrative task, yet it is one of the first major quality gates in a project. Testing during this phase does not involve executing code, as the code often does not exist yet. Instead, the testing activity here takes the form of rigorous review and verification of the Test Plan document itself.

The primary goal during this phase is to ensure the test strategy is feasible and complete. Stakeholders, including the Project Manager, Architects, and QA Leads, gather to review the Test Plan. They

NOTE

scrutinize the scope to ensure no critical modules have been accidentally excluded. They evaluate the resource estimates to determine if the proposed timeline is realistic or if it sets the team up for failure. For instance, if the plan allocates two days for a task that historically takes five, "testing the plan" identifies this defect immediately. Furthermore, the entry and exit criteria are validated to ensure they are measurable and attainable. By catching these planning defects early, the organization avoids the costly chaos of running out of time or budget later in the project lifecycle.

20.3.1 Testing during design phase

The Design Phase is where the requirements are translated into a technical blueprint. This phase is usually divided into High-Level Design (HLD) (Architecture) and Low-Level Design (LLD) (Component logic). Testing during this phase is purely static (verification), but it is arguably the most high-leverage activity in the lifecycle.

1. Reviewing the Design Architecture:

The primary "test" here is the Design Review. Architects and Senior Testers examine the proposed architecture for flaws.

- **Completeness:** Does the architecture cover all requirements? If the requirement is "secure login," does the design specify encryption protocols?
- **Testability:** This is a crucial check. A design that is tightly coupled (spaghetti code) is hard to test. Testers "test the design" by asking: "How would I mock this database connection?" or "How can I inject a failure here?" If the design doesn't allow for easy testing, it is flagged as a defect.
- **Scalability and Performance:** The design is analyzed mathematically. "Will this database schema support 1 million records?" This is a theoretical test performed before a single table is created.

2. Designing Integration Test Cases:

While developers are designing the modules, testers begin designing the Integration Test Plan.

- Since the HLD defines the interfaces between modules (e.g., how the Billing Module talks to the User Module), the QA team writes test cases to verify these handshakes.
- By writing these tests *now*, testers often find logical gaps in the interface definitions (e.g., "The Billing module expects a String, but the User module sends an Integer"). Finding this mismatch during the design phase costs minutes to fix; finding it during execution costs days.

NOTE

3. Risk Analysis:

During the design phase, the testing team identifies high-risk areas. If the design involves a complex new algorithm or a third-party API, the test plan is adjusted to allocate more time and rigorous testing techniques (like Fagan Inspection) to those specific areas.

Once the requirements are finalized, the project moves into the design phase. Testing during this stage is purely a verification activity, typically conducted through reviews, inspections, and walkthroughs. The objective is to identify structural flaws in the architecture before they are cemented into code.

The design phase is usually split into High-Level Design (HLD) and Low-Level Design (LLD), and testing occurs in both. During the High-Level Design review, the focus is on the system architecture. Testers and architects verify that the proposed architecture can support the non-functional requirements, such as performance, security, and scalability. They check if the interaction between major components is logically sound and if the proposed technology stack is compatible with the client's infrastructure.

During the Low-Level Design review, the focus narrows to the logic of individual modules, database schemas, and interface definitions. Testers verify that the pseudo-code or logic flow correctly implements the business requirements defined in the SRS. They specifically look for missing error handling, such as what happens if a database connection fails or an invalid input is provided. This phase also relies heavily on the Traceability Matrix, where reviewers ensure that every single requirement has a corresponding design element. If a requirement exists but has no corresponding design, it is a defect that must be fixed immediately.

Activity Title: Preparing a Basic Test Plan

Objective:
To understand how testing activities are planned before test execution.

Materials Required:

- Sample software project
- Test-plan template
- Notebook

Procedure/Steps:

1. Select a simple software project.

NOTE

2. Define testing objectives and scope.
3. Identify features to be tested and not tested.
4. Identify resources, responsibilities, tools, and test environment.
5. Prepare a basic test schedule.
6. Identify major risks and assumptions.

Outcome:
The learner will be able to prepare a basic software test plan and understand the importance of systematic test planning.

20.3.2 Testing during coding

The Coding Phase is where the "Test Planning" meets the "Test Execution" for the first time. While the primary output is source code, the testing activities here are intense and involve both static and dynamic methods.

Testing during the coding phase is a hybrid activity that involves both static analysis (reading the code) and dynamic testing (running the code). This is the phase where the abstract designs finally become concrete executable software.

The first line of defense is Static Code Analysis. This involves using automated tools to scan the source code for syntax errors, security vulnerabilities, and non-compliance with coding standards. For example, a tool might flag that a developer has hard-coded a password or failed to close a database connection. These checks are instant and prevent "technical debt" from accumulating.

Simultaneously, Unit Testing is performed. This is a dynamic testing activity usually carried out by the developers themselves. The goal is to isolate the smallest testable parts of the application individual functions, methods, or classes and verify that they function correctly in isolation. Developers write test harnesses that feed inputs to a function and assert that the output matches the expectation. Effective unit testing relies on techniques like Boundary Value Analysis to ensure the code handles edge cases correctly. Additionally, Code Reviews are conducted where a peer developer reads the code to check for logic errors, readability, and maintainability.

1. Static Code Analysis:

This is "testing without running." Automated tools (like SonarQube, ESLint, or Checkstyle) run constantly as developers write code. They "test" the code against defined standards.

NOTE

- **Syntax and Style:** Ensuring code follows the project's naming conventions and indentation rules.
- **Complexity Metrics:** Tools measure Cyclomatic Complexity. If a function is too complex (too many if/else statements), the tool flags it. This acts as an early warning that the code will be buggy and hard to maintain.

2. Code Reviews and Walkthroughs:

Before code is merged into the main branch, it undergoes peer review. A second developer reads the code to look for logic errors, memory leaks, and security vulnerabilities. This is a manual "test" of the code's logic.

- *Example:* A reviewer might spot that a loop will never terminate under a specific condition, saving the project from an infinite loop bug in production.

3. Unit Testing (White Box Testing):

This is the first dynamic testing. Developers write small scripts to test individual functions.

- Test-Driven Development (TDD): In this approach, the test is actually written *before* the code. The developer writes a failing test, then writes the code to make it pass. This ensures 100% test coverage for the logic.
- Testing during this phase focuses on Statement Coverage and Branch Coverage (as discussed in Unit 17).

20.3.3 VV Model

The VV Model (Verification and Validation Model)—synonymous with the V-Model discussed in Unit 19—is the theoretical framework that mandates this early testing. In the context of "Testing during Test Planning," the VV Model dictates the principle of Correspondence. The VV Model, commonly known as the V-Model, provides the structural framework for all the activities discussed above. It is based on the principle that for every development activity, there is a corresponding testing activity. The model visualizes the software development lifecycle as a "V" shape, with execution moving down the left side and up the right side.

The Principle of Correspondence:

For every planning or development activity on the left side of the 'V', there is a corresponding testing activity that must be planned *at the same time*.

1. Requirement Phase ↔ Acceptance Test Planning:

NOTE

While requirements are being gathered, the Acceptance Test Plan is written. The "testing" activity here is verifying the requirements for ambiguity.

2. System Design Phase ↔ System Test Planning:

   While the system is being designed, the System Test Plan is created. We define the hardware and software environments needed for testing. We create "Stubs" and "Drivers" plans.

3. Module Design Phase ↔ Integration Test Planning:

   As detailed logic is planned, the specific data flows between modules are mapped out in the Integration Test Plan.

Benefits of VV Model in Planning:

- **No Gap Time:** When coding finishes, testing starts immediately because the plans and test cases are already ready.
- **Traceability:** It ensures every requirement has a test case before coding even starts.
- **Early Defect Detection:** It enforces the "Shift Left" philosophy, catching design errors on paper.

The Verification arm (the left side) focuses on the question, "Are we building the product right?" This includes the Requirement Analysis, System Design, and Module Design phases. As we have discussed, "testing" in these phases consists of reviews and inspections of documents. It is a preventative approach aimed at error detection.

The Validation arm (the right side) focuses on the question, "Are we building the right product?" This includes Unit Testing, Integration Testing, System Testing, and User Acceptance Testing. These phases involve executing the code to demonstrate that it meets the user's needs. The strength of the VV Model lies in its coupling of these two arms; the Test Plan for System Testing is derived directly from the System Requirements phase on the left, ensuring that the tests are perfectly aligned with the original specifications.

20.3.4 Critical Roles and Responsibilities

Effective testing during the planning phases requires a clear division of labor. It is not just the "Tester's" job; it involves the entire technical team.

Successful software testing is not just about processes; it is about people. A structured testing team requires specific roles with defined responsibilities to function effectively.

The Test Manager The Test Manager is the strategic leader of the QA department. Their primary responsibility is high-level planning and management. They define the organization's testing policy and strategy.

NOTE

The Test Manager interfaces with upper management to secure budgets and resources. They are responsible for the overall schedule and ensure that the testing effort aligns with the project's business goals. They also handle escalation—if the testing team is blocked by a lack of environment or delayed builds, the Test Manager resolves these issues with the Project Manager.

The Test Lead The Test Lead is the tactical manager who oversees the day-to-day operations of a specific project. They are responsible for writing the detailed Test Plan for that project. The Test Lead assigns tasks to individual testers, monitors their progress, and reviews their work. They act as the technical mentor for the team, helping to resolve difficult testing scenarios. The Test Lead is also responsible for generating the daily or weekly status reports that show defect trends and testing progress to the rest of the project team.

The Test Engineer (Tester) The Test Engineer is the operational core of the team. Their primary responsibility is to analyze requirements and design detailed test cases. They set up the test environment and execute the tests, whether manually or using automation tools. When they find a discrepancy between the actual and expected results, they log a detailed defect report. Beyond just finding bugs, the Test Engineer is responsible for verifying the fixes provided by developers (Re-testing) and running regression suites to ensure no new bugs were introduced.

1. Test Manager:

- Strategy: Defines *what* to test and *how* deep to test. Decides on the tools and automation frameworks.
- Resource Planning: Estimates how many testers are needed and what hardware (servers, mobile devices) is required.
- Risk Management: Identifies risks (e.g., "The payment gateway API is unstable") and plans mitigation strategies.

2. Test Lead:

- Technical Guidance: Mentors the junior testers.
- Monitoring: Tracks the progress of test case creation against the project timeline.
- Review: Verification of the Test Plan and Test Cases created by team members to ensure quality.

3. Software Tester (QA Engineer):

- Test Design: Creates the detailed test cases and scripts during the design/coding phase.
- Data Preparation: Creates the mock data sets (e.g., dummy user profiles, invalid credit card numbers) needed for execution.

NOTE

- Requirement Analysis: strict review of the requirements to ensure they understand what to test.

Check Your Progress

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. The main purpose of test planning is to:
A) Avoid testing
B) Decide scope, strategy, schedule, and resources for testing
C) Increase program size
D) Write user manuals only

_____

Q2. Testing during the design phase helps in detecting design defects early.

_____

Q3. Testing during coding phase includes activities like unit testing and _____ reviews.

_____

Q4. Write any two critical roles and responsibilities in testing.

Answer (Any two):

- Test manager: planning, scheduling, and monitoring testing
- Tester: designing and executing test cases
- Developer: fixing defects and supporting unit testing
- Quality assurance team: audits and process monitoring

_____

_____

Q5. VV model in testing refers to:
A) Video verification model
B) Verification and Validation model
C) Virtual volume model
D) Variable value model

_____

Fill-in-the-Blank Questions:

The practice of moving testing activities earlier in the project timeline is known as _____.

NOTE

_____ analysis involves checking the code for errors without executing it, often using automated tools.

In the VV Model, the Integration Test Plan is derived from the _____ phase.

True/False Questions:

Testers should wait until the code is fully written before writing any test cases.

A primary goal of reviewing the design is to check for "Testability."

Unit testing is typically the responsibility of the Developer, not the QA Tester.

20.4 SUMMARY

Testing is not a phase that happens at the end; it is a discipline that permeates the entire lifecycle. In Unit 20, we explored the concept of Testing during the Test Planning Phase, which refers to the concurrent Verification and Test Preparation activities. We learned that during the Design Phase, testing takes the form of architectural reviews and risk analysis, looking for completeness and testability. During the Coding Phase, testing involves static analysis, peer reviews, and unit testing to ensure the building blocks are solid. We revisited the VV Model to emphasize how test plans are born from development artifacts. Finally, we outlined the Roles, noting that the Test Manager focuses on strategy and risk, while the Test Lead and Testers focus on the tactical creation of test cases and data, ensuring the team is ready to execute the moment code is delivered.

20.5 GLOSSARY

- Shift Left: A practice in software development where testing is performed earlier in the lifecycle (i.e., moved "left" on the project timeline).
- Static Analysis: The analysis of computer software that is performed without actually executing programs (e.g., code reviews, linting).
- Testability: The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.

NOTE

- Cyclomatic Complexity: A software metric used to indicate the complexity of a program. Higher complexity often implies lower testability and higher risk.
- TDD (Test-Driven Development): A software development process where requirements are turned into very specific test cases, then the software is improved so that the tests pass.

20.6 ANSWER TO CHECK YOUR PROGRESS

Q1. Answer: B

Explanation: Test planning defines what will be tested, how it will be tested, who will test, and when testing will be performed.

Q2. Answer: True

Explanation: Reviewing design documents and test design during the design stage helps prevent major defects before coding begins.

Q3. Answer: Code

Explanation: Code reviews and unit tests help detect defects early at the implementation stage.

Q4. Explanation: Testing requires coordination between different roles to ensure quality and timely defect resolution.

Q5. Answer: B

Explanation: VV stands for Verification and Validation, which ensures software is built correctly and meets customer needs.

20.7 SUGGESTED READINGS - OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3799	

20.8 REFERENCE

- Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach*. McGraw-Hill.
- Myers, G. J. (2011). *The Art of Software Testing*. Wiley.

NOTE

- Burnstein, I. (2003). *Practical Software Testing*. Springer.

20.9 EXERCISE

Part–1

A. Multiple Choice Questions

Q1. Module testing focuses on:

A) Testing a group of related units

B) Only database testing

C) Only GUI testing

D) Only performance testing

Q2. Module testing is performed:

A) Before unit testing

B) After unit testing

C) Only after deployment

D) Never required

Q3. Module testing helps in detecting:

A) Integration issues within module

B) Hardware faults

C) License errors

D) Marketing defects

.

Q4. Module testing is also called:

A) Component testing

B) Security testing

C) Alpha testing

D) Beta testing

Q5. Module testing ensures:

A) Only performance

B) Module functions as specified

C) No documentation needed

D) No need of integration testing

Answers with Explanation:

NOTE

1. **Answer:** A

   Explanation: Module testing checks combined units.

2. **Answer:** B

   Module testing comes after unit tests.

3. **Answer:** A

   It identifies internal module integration problems

4. **Answer:** A

   Modules are treated as components.

5. **Answer:** B

   It validates module behavior.

B. Small Answer Questions

1. Define module testing.
2. How is module testing different from unit testing?
3. Write two objectives of module testing.
4. Write two examples of module-level defects.
5. Why is module testing required?

Part–2

Long Answer Type Questions (200 words)

1. Explain module testing and its importance.
2. Discuss module test planning and test design.
3. Explain module integration issues and their detection.
4. Discuss advantages of module testing.
5. Explain module testing in real software projects.

Part–3

Long Answer Type Questions (300 words)

1. Explain module testing in detail with examples.
2. Discuss module testing techniques and strategies.
3. Explain how module testing reduces integration risks.
4. Discuss module testing challenges and solutions.
5. Explain the role of module testing in quality assurance.

RNTU Rabindranath TAGORE UNIVERSITY

Approved by : AICTE, NCTE, BCI, INC, MP PARAMEDICAL COUNCIL, DEB (UGC) | Recognized by : UGC | Member of : AIU, ACU

RABINDRANATH TAGORE UNIVERSITY (M.P.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org / for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

_____

_____

_____

***

BLOCK 4	LEVELS OF TESTING, SPECIAL TESTS, INSTALLATION TESTING	331-388
Unit 21	Levels of Testing	333
Unit 22	Special Tests	359
Unit 23	Installation Testing	373



Block V focuses on important testing activities that are performed at different stages of the software development and deployment process. It introduces the **levels of testing**, beginning with unit testing and progressing through integration, system, and acceptance testing. The block also discusses **special tests** designed to evaluate specific characteristics of software, such as performance, usability, security, reliability, and compatibility. Finally, it explains **installation testing**, which ensures that software can be installed, configured, upgraded, and uninstalled correctly across different environments. Through these units, learners will understand how different testing approaches help identify defects, verify software quality, and ensure that applications are ready for successful deployment and use.

NOTE

UNIT 21
LEVELS OF TESTING

UNIT STRUCTURE

21.1 Introduction
21.2 Objective
21.3 Levels of Testing
21.3.1 About Testing
21.3.2 Proposal Testing
21.3.3 Requirement Testing
21.3.4 Design Testing
21.3.5 Code Review
21.3.6 Unit Testing
21.3.7 Module Testing
21.3.8 Integration Testing
21.3.9 Big-Bang Testing
21.3.10 Sandwich Testing
21.3.11 Critical Path First
21.3.12 Sub System Testing
21.3.13 System Testing
21.3.14 Testing Stages
21.4 Summary
21.5 Glossary
21.6 Answer to check your progress
21.7 Suggested readings - OER
21.8 Reference
21.9 Exercise

21.1 INTRODUCTION

Software testing is one of the most important activities in the field of Software Quality Assurance and Testing. The purpose of testing is not only to find defects, but also to ensure that the software system meets its requirements, satisfies user expectations, and performs reliably under real-world conditions. In modern software development, testing is not treated as a single activity performed at the end of the project. Instead, testing is considered a continuous process that begins from the earliest stage of development and continues until the software is deployed and maintained.

NOTE

Software systems today are complex, large, and highly integrated. They may include web applications, mobile applications, embedded systems, cloud services, database systems, and distributed components. Such systems interact with different environments, different users, and different hardware and software platforms. Because of this complexity, there are many ways in which a system can fail. A failure may occur due to incorrect requirements, poor design decisions, coding errors, integration issues, security vulnerabilities, performance problems, or incorrect assumptions about user behaviour. Testing helps identify these issues before the software reaches the customer.

Another reason why testing is critical is the high cost of defects. A defect found in the early stages, such as during requirement analysis, is much cheaper to fix than a defect found after deployment. Studies in software engineering show that the cost of fixing a defect increases significantly as it moves through the development life cycle. A defect in requirements may require only documentation changes. A defect in design may require redesign. A defect in code may require rewriting and retesting. A defect found after release may lead to customer dissatisfaction, business loss, legal issues, and damage to the organisation's reputation. Therefore, testing at different stages helps reduce cost and improve quality.

Software testing is not limited to executing a program and checking outputs. It also includes verification activities such as reviews, inspections, walkthroughs, and audits. These activities examine documents, designs, and code without executing the program. Such techniques are often more effective at finding defects early. For example, code review can detect logical errors, security issues, and poor coding practices before the program is even compiled. Requirement testing can identify missing or ambiguous requirements that would otherwise cause major defects later.

Different types of testing exist because different testing objectives must be achieved at different stages. Some testing focuses on checking correctness of functionality. Some focuses on checking integration of components. Some focuses on performance and reliability. Some focuses on usability and security. The classification of testing into different types helps testers plan testing activities systematically. It also helps ensure that the software is evaluated from all important quality perspectives.

In the complex discipline of software engineering, "testing" is not a singular activity but a vast spectrum of verification and validation techniques. The quality of a software product is not determined by a single phase of execution but by a continuous series of quality filters applied from the moment a project is conceived until it is retired. This unit explores the taxonomy of testing types, classifying them not just

NOTE

by *how* they are performed (static vs. dynamic) but by *when* they are performed and *what* specific scope of the system they target.

Testing is not a monolithic activity; it is a layered process that evolves as the software grows from a simple idea into a complex system. Just as a building is inspected at the blueprint stage, the foundation stage, and the final walk-through, software undergoes distinctly different types of testing at each stage of its lifecycle. This unit breaks down the spectrum of testing types, starting from the early verification of proposals and requirements, moving through the technical validation of code and modules, and culminating in the strategic integration and system-level testing strategies like Big-Bang and Sandwich testing.

Historically, organizations relied heavily on "System Testing" waiting until the entire application was built before looking for defects. This approach proved disastrously expensive because a bug introduced in the requirements phase might remain undetected for months, burying itself deep in the architectural foundation. Modern Software Quality Assurance (SQA) advocates for a multi-layered testing strategy. This involves "Static Testing" types (like Proposal and Design verification) that prevent defects from entering the code, and a hierarchy of "Dynamic Testing" types (Unit, Integration, System) that catch execution errors at different levels of granularity.

This unit provides a detailed theoretical examination of these types. We will analyze why testing a business proposal is as critical as testing code, how integration strategies like "Sandwich Testing" solve complex dependency issues, and how the progression from Unit to System testing creates a "safety net" that ensures robust, reliable software delivery.

21.2 OBJECTIVE

After studying this unit, you should be able to:

- Define the scope and purpose of testing at the proposal and requirement stages.
- Differentiate between Unit, Module, and Sub-system testing.
- Explain the role of Code Reviews as a static testing mechanism.
- Compare and contrast Integration strategies such as Big-Bang, Sandwich, and Critical Path First.
- Describe the comprehensive nature of System Testing.

NOTE

21.3 LEVELS OF TESTING

Software testing can be classified in many ways, such as based on testing level, testing technique, testing objective, or testing phase. In this unit, the focus is on testing types that occur across the software development life cycle. These testing types begin from early documents such as proposals and requirements, continue through design and code, and extend into integration and system-level validation.

The reason for dividing testing into multiple types is that each stage of development produces different outputs. For example, the requirement stage produces requirement specifications. The design stage produces architectural and detailed design documents. The coding stage produces source code. Integration produces combined components. System testing validates the full software product. Each output must be tested using appropriate methods.

Testing types also differ in their goals. Proposal testing checks feasibility and correctness of project proposals. Requirement testing checks completeness and clarity of requirements. Design testing checks whether design satisfies requirements. Code review checks correctness and quality of code. Unit testing checks individual units. Module and integration testing check combined components. System testing checks the complete system. Testing stages ensure systematic progression and quality assurance.

The following sub-sections explain each type of testing in detail.

21.3.1 About Testing

Testing is the process of evaluating a software product or system to determine whether it meets specified requirements and to identify defects. Testing is not only about finding errors but also about increasing confidence in the software. It ensures that the software behaves correctly under normal and abnormal conditions.

Testing can be divided into two broad categories: verification and validation. Verification means checking whether the software is being built correctly according to specifications. Validation means checking whether the correct software is being built according to user needs. Both are important. Verification activities include reviews, inspections, and walkthroughs. Validation activities include executing the software and checking outputs.

Testing also includes static and dynamic testing. Static testing is performed without executing the code, such as reviewing documents and code. Dynamic testing is performed by executing the program with test inputs. Both types are necessary. Static testing helps detect defects

NOTE

early, while dynamic testing ensures that the software works correctly when executed.

Another important aspect of testing is that it should be planned. Testing requires test plans, test cases, test data, test environments, and expected results. Testing also requires documentation so that defects can be tracked and fixed. Without planning, testing becomes random and ineffective.

Testing is also influenced by the psychology of developers and testers. Developers may unconsciously avoid testing their own code thoroughly because they assume it works. Testers must think creatively and critically to identify weaknesses. Therefore, independent testing teams often provide better results.

Software testing is often categorized into two broad domains: Static Testing and Dynamic Testing. Static testing involves reviewing the work products (documents, code) without executing the software, whereas Dynamic testing involves running the software to validate its behavior. The types of testing discussed in this unit span both domains, moving chronologically from the earliest project phases to the final system release. The goal of using multiple types of testing is to filter out defects at the earliest possible point, as the cost of fixing a bug grows exponentially the later it is found.

Software testing is best understood as a series of sieves or filters, each with a different mesh size.

- Coarse Filters (Early Testing): These catch the "boulders" the massive logic errors, misunderstood business goals, and architectural flaws. If these pass through, they can crush the project.
- Medium Filters (Integration/System): These catch the "stones" the interface mismatches, data flow errors, and performance bottlenecks.
- Fine Filters (Unit/Code): These catch the "sand" the syntax errors, calculation mistakes, and boundary value failures.

A robust testing strategy uses *all* these types. Relying only on one type (e.g., only System Testing) inevitably leads to low-quality software because fine-grained errors slip through coarse filters, and coarse errors are too big to be seen by fine filters. This section details each filter in the sequence of the SDLC.

21.3.2 Proposal Testing

Proposal testing is the very first line of defense in the project lifecycle, often occurring before a formal contract is even signed. It involves a rigorous review of the project proposal by technical and quality

NOTE

assurance leads. The primary objective is to verify the feasibility of the commitments being made to the client. The testing team checks if the proposed timeline is realistic, if the technical solution is viable, and if the resource estimates are accurate. For example, if a proposal promises to deliver a complex AI engine in two weeks with a team of junior developers, "Proposal Testing" would flag this as a critical risk (a defect in the proposal) that must be fixed before the contract is finalized.

Proposal testing refers to the evaluation of the software project proposal before development begins. In many organisations, a software project starts with a proposal document that describes the purpose of the system, expected benefits, scope, major requirements, estimated cost, schedule, and resources. Proposal testing checks whether the proposal is realistic, complete, and feasible.

Proposal testing is important because many project failures occur due to unrealistic expectations and poor planning at the proposal stage. If the proposal underestimates cost or time, the project may suffer delays and budget overruns. If the proposal does not clearly define scope, the project may experience scope creep. Proposal testing helps identify such issues early.

In proposal testing, reviewers evaluate whether the problem statement is clear, whether objectives are measurable, and whether the proposed solution is technically feasible. They also check whether the proposal considers risks such as technology limitations, dependency on external systems, security requirements, and legal constraints. For example, if a project involves handling sensitive customer data, the proposal must include compliance requirements such as data protection laws.

Proposal testing also evaluates resource planning. It checks whether the required skills are available, whether team size is sufficient, and whether infrastructure is adequate. It also verifies whether the proposed schedule is achievable based on complexity.

Often overlooked in technical testing literature, Proposal Testing is the strategic "Gate Zero" of software quality. Before a single requirement is written or a line of code is drafted, a project exists as a Proposal or a Statement of Work (SOW). If this document is flawed, the project is doomed regardless of how good the coding is.

The Feasibility Test:

The primary goal of proposal testing is to verify feasibility. Technical architects review the proposal to "test" the technical claims.

- *Example:* If a proposal claims the system will "Use Machine Learning to predict stock prices with 100% accuracy," a proposal test would flag this as scientifically impossible. Catching this lie here saves millions of dollars in wasted development.

NOTE

The Budget and Schedule Test:

Proposals often contain optimistic schedules to win contracts. Proposal testing involves analyzing these estimates against historical data.

- *Risk Assessment:* If the proposal allocates 2 weeks for a phase that historically takes 2 months, the "test" fails. The QA team flags this as a "Schedule Risk." This allows management to renegotiate the timeline before the contract is signed, preventing inevitable "crunch time" and low-quality code later.

The Acceptance Criteria Test:

The proposal defines what "success" looks like. Proposal testing verifies that these criteria are measurable. If the proposal says, "The system will be delivered when it is stable," the tester flags this as defective because "stable" is subjective. It must be corrected to: "Delivered when critical defects are zero."

21.3.3 Requirement Testing

Requirement testing is a static verification activity where the Software Requirement Specification (SRS) is scrutinized for quality. The goal is to ensure that requirements are clear, complete, consistent, and testable. A requirement that states "The system must be fast" is defective because "fast" is subjective. Requirement testing ensures this is rewritten to something measurable, like "The system must respond within 500ms." By identifying ambiguity or contradictory requirements at this stage, the team prevents fundamental design flaws that would be expensive to correct later. This phase often involves the creation of a Requirement Traceability Matrix (RTM) to ensure every business need is tracked.

Requirement testing is the process of evaluating software requirements to ensure they are correct, complete, consistent, unambiguous, and testable. Requirements are the foundation of software development. If requirements contain defects, the entire system may be built incorrectly.

Requirement testing is usually performed through requirement reviews, inspections, and validation meetings. Testers, developers, business analysts, and customers participate. The goal is to identify missing requirements, unclear statements, contradictions, and unrealistic expectations.

A requirement is testable only if it is measurable. For example, “system should be fast” is not testable. A better requirement is “system should respond within 2 seconds for 95% of requests.” Requirement testing ensures that requirements are expressed clearly and in measurable terms.

NOTE

Requirement testing also ensures traceability. Each requirement should be linked to design elements, code modules, and test cases. This helps ensure that every requirement is implemented and tested. Requirement Traceability Matrix (RTM) is often used for this purpose.

Another important part of requirement testing is checking non-functional requirements. Many systems fail not because of functional issues but because of performance, security, usability, and reliability issues. Requirement testing ensures these quality attributes are defined properly.

Requirement Testing acts as the bridge between the business need and the technical solution. It validates the Software Requirement Specification (SRS). As established by the "Cost of Quality" curve, a defect found here costs $1 to fix, whereas finding it in production costs $100+.

Attributes of Quality Requirements:

Static testing of requirements involves inspecting the SRS for specific quality attributes:

1. Ambiguity: Words like "fast," "easy," "user-friendly," or "robust" are ambiguity defects. Requirement testing forces these to be quantified (e.g., "Page load < 2 seconds").
2. Completeness: The tester looks for missing logic. If the requirement defines the "Happy Path" (user pays with valid card), does it define the "Sad Path" (card declined)? If not, the requirement is incomplete.
3. Consistency: Does Requirement A (Admin sees all data) contradict Requirement B (Privacy laws hide user data)?
4. Testability: This is the ultimate litmus test. If a QA engineer cannot imagine a test case to prove a requirement true or false, the requirement is defective.

The Shift-Left Philosophy:

Requirement testing is the embodiment of "Shift Left"—moving the testing mind set to the very start of the timeline. By writing Acceptance Test Cases during this phase, testers effectively "debug" the requirements.

21.3.4 Design Testing

Design testing refers to evaluating software design documents to ensure that the design correctly implements the requirements and provides a reliable structure for development. Design testing is mainly a verification activity and is usually performed through design reviews, inspections, and walkthroughs.

NOTE

Design includes architectural design and detailed design. Architectural design defines system structure, modules, interfaces, and data flow. Detailed design defines algorithms, data structures, and module-level logic. Design testing checks whether the architecture is scalable, maintainable, secure, and efficient.

One major objective of design testing is to identify design flaws before coding begins. Design defects are expensive to fix later. For example, if the system architecture is not scalable, it may require major restructuring. If security is not considered in design, the system may become vulnerable.

Design testing also checks interface consistency. Many integration defects occur due to mismatched interfaces between modules. Design testing ensures that module interfaces are clearly defined and consistent. Design testing also verifies that error handling is included. Many designs focus only on normal scenarios and ignore failure cases. Design testing ensures that exceptions, invalid inputs, and system failures are handled.

Once requirements are finalized, the project moves to design. Design testing verifies the architectural and detailed design documents. This includes checking the High-Level Design (HLD) for architectural soundness, scalability, and security, and the Low-Level Design (LLD) for logical correctness and data integrity. Reviewers ensure that the design actually covers all the requirements defined in the SRS. If a requirement asks for "User Recovery," but the design document lacks a password reset flow, this gap is identified as a defect. Design testing prevents the team from building a robustly coded system that solves the wrong problem.

Once requirements are baselined, the solution is designed. Design Testing verifies the High-Level Design (HLD) (Architecture) and Low-Level Design (LLD) (Component logic) before construction begins.

Architectural Verification:

Reviewers simulate the flow of data through the proposed architecture to find bottlenecks.

Example: If the design uses a single database instance for a global application, the "Design Test" will calculate the latency for users on the other side of the world. If the latency exceeds requirements, the design fails.

Security and Compliance:

Design testing is crucial for security. Reviewers look at the design to ensure "Security by Design." They check if passwords are encrypted in the database schema, if APIs require authentication, and if sensitive data logs are masked. Finding a security flaw here (e.g., "The design

NOTE

transmits data in clear text") allows for a secure protocol to be selected before the insecure code is written.

Interface Definition:

A major part of design testing is verifying the API Contracts. Do the inputs and outputs of Module A match what Module B expects? Fixing an interface mismatch on a whiteboard is trivial; fixing it after both teams have coded their modules is painful.

21.3.5 Code Review

Code Review is the systematic examination of source code by someone other than the author. It is a form of static testing that finds bugs, security vulnerabilities, and violation of coding standards without running the program. Beyond just finding errors, code reviews facilitate knowledge sharing across the team and ensure the codebase remains maintainable. Peer reviews, walkthroughs, and formal inspections are all methods used here. It checks for issues like memory leaks, inefficient algorithms, and poor variable naming that automated tests might miss.

Code review is a static testing technique in which source code is examined manually to detect defects without executing the program. Code review is one of the most effective methods for early defect detection. It helps identify logical errors, coding standard violations, security vulnerabilities, performance issues, and maintainability problems.

Code reviews can be informal or formal. Informal reviews include peer reviews where developers check each other's code. Formal reviews include inspections with defined roles such as moderator, reviewer, author, and recorder. In formal inspections, defects are documented systematically.

Code review helps detect defects that may not be easily found by testing. For example, a security vulnerability such as SQL injection may not be detected unless specific tests are performed. Code review helps identify such issues by examining how input validation is implemented.

Code review also improves code quality by encouraging best practices. It helps ensure that code is readable, properly commented, and follows standards. It also helps reduce technical debt. Another important benefit is knowledge sharing. Code review allows team members to understand each other's work, improving collaboration and reducing dependency on individual developers.

Code Review is a static testing technique where source code is examined by humans or automated tools without executing the program. It serves as a check for maintainability, readability, and adherence to standards.

Manual Peer Reviews:

NOTE

A developer (the author) submits code, and another developer (the reviewer) reads it. They look for:

- Logic Errors: "This loop has an off-by-one error."
- Optimization: "This database query is inside a loop; it will be slow."
- Readability: "Variable name x is unclear; rename it to customerCount."

Automated Static Analysis:

Tools like SonarQube, ESLint, or Fortify scan the code for patterns known to cause bugs.

- Cyclomatic Complexity: The tool calculates the number of linearly independent paths through a program's source code. If a function has a complexity score > 10 (meaning lots of nested IFs and Loops), the tool flags it as "Too Complex / High Risk."
- Security Scanning: Tools look for known vulnerabilities like SQL Injection or Hardcoded Passwords.

21.3.6 Unit Testing

Unit Testing is the first level of dynamic testing and is typically performed by the developers themselves. A "unit" is the smallest testable part of an application, such as a single function, method, or procedure. The goal is to isolate each part of the program and show that the individual parts are correct. Developers use drivers (simulating a calling function) and stubs (simulating a called function) to test the unit in isolation.

Unit testing is the testing of individual units or components of software. A unit is the smallest testable part of a program, such as a function, method, or class. Unit testing focuses on verifying that each unit works correctly according to its design and requirements.

Unit testing is usually performed by developers during the coding phase. It is a dynamic testing technique because code is executed with test inputs and outputs are checked. Unit testing helps detect coding errors early, such as incorrect logic, wrong calculations, and incorrect handling of conditions.

Unit tests are usually written using testing frameworks such as JUnit, NUnit, or pytest. They may include assertions to compare expected and actual outputs. Unit testing also involves testing boundary conditions, invalid inputs, and exception handling.

Unit testing is important because it ensures that each unit is correct before integration. If units are not tested, integration testing becomes difficult because defects are harder to isolate.

NOTE

Unit Testing marks the beginning of Dynamic Testing. It focuses on the smallest atomic parts of the software typically individual functions, methods, or classes.

Isolation is Key:

The defining characteristic of a Unit Test is Isolation. The unit should be tested in a vacuum. If a function calculateTax() calls a database to get tax rates, a proper Unit Test will not call the real database. Instead, it will use a Mock Object or Stub to simulate the database response.

- *Why?* If the test fails, we must be 100% sure the bug is in the logic of calculateTax(), not in the database connection or the network.

White-Box Nature:

Unit testing is almost always "White-Box." The tester (usually the developer) looks at the internal code structure to ensure that every line of code is executed at least once (Statement Coverage) and every condition is tested as True and False (Branch Coverage).

21.3.7 Module Testing

Module Testing is often synonymous with Unit Testing but is sometimes distinguished as testing a collection of dependent units that form a distinct logical module (e.g., the entire "Login" module comprising the UI, validation logic, and database connector). It ensures that the functions inside a single component work together correctly before being integrated with other large modules.

Module testing is testing of a software module, which may consist of multiple units combined together. A module is larger than a unit and usually represents a functional part of the system. For example, in a banking system, modules may include account management, transaction processing, loan processing, and customer management.

Module testing verifies that the combined units within the module work correctly together. It focuses on internal interactions between units, correct data flow, and correct handling of module-level functionality.

Module testing may involve testing module interfaces and ensuring that the module provides correct outputs for different inputs. It may also include testing of module-level error handling and performance.

Module testing is important because even if individual units work correctly, their integration within a module may fail due to interface mismatches or incorrect assumptions.

While often synonymous with Unit Testing, Module Testing (or Component Testing) typically operates at a slightly higher level of granularity.

NOTE

Scope of a Module:

A "Unit" might be a single function (e.g., validateEmail()). A "Module" is the collection of functions and data structures that represent a feature (e.g., the UserRegistration class, containing validation, database formatting, and error handling methods).

Cohesion Testing:

Module testing verifies Intra-module interaction. It checks if the internal functions of the module work together correctly. For example, does the private variable updated by Function A get correctly read by Function B within the same class? It validates the "cohesion" of the component before it is exposed to the rest of the system.

21.3.8 Integration Testing

Integration Testing focuses on the interactions between different modules or components. Even if Module A and Module B work perfectly in isolation, they might fail when connected due to interface mismatches, data format errors, or timing issues. Integration testing exposes faults in the interaction between integrated units. There are several strategies to approach this, as detailed below.

Once modules are verified to work individually, they must be assembled. Integration Testing is the phase where these combined modules are tested as a group.

The "Interface" Problem:

The most common source of failure in large systems is not the logic *inside* the components, but the communication *between* them. Integration testing specifically targets:

- **Data Mismatches:** Module A sends a Date in "MM-DD-YYYY" format, but Module B expects "YYYY-MM-DD".
- **Protocol Errors:** Module A uses HTTP, but Module B expects HTTPS.
- **Timing Issues:** Module A requests data, but Module B hasn't finished calculating it yet (Race Condition).

Integration testing does *not* re-test the internal business logic (that was Unit Testing). It focuses strictly on the flow of data across the interfaces.

Integration testing is testing performed after units or modules are combined. The objective is to detect defects related to interactions between integrated components. Integration defects occur due to incorrect interfaces, wrong data formats, incorrect sequence of operations, and improper communication between components.

Integration testing ensures that modules work correctly when combined. It verifies data flow, control flow, and communication

NOTE

between components. For example, in a web application, integration testing checks whether the front-end interacts correctly with the back-end, whether APIs return correct responses, and whether database operations work correctly.

Integration testing can be performed using different approaches such as top-down, bottom-up, big-bang, and sandwich testing. Each approach has advantages and limitations. Integration testing is essential because most real systems involve multiple components and defects often occur at integration points.

21.3.9 Big-Bang Testing

In Big-Bang Integration Testing, all the modules are developed independently and then integrated simultaneously in one go. The entire system is effectively built and tested as a massive block. While this approach requires little planning, it is generally discouraged for large systems. If a critical bug is found, it is extremely difficult to isolate the root cause because the error could originate from any of the hundreds of newly connected modules. It is a high-risk strategy usually reserved for very small or throwaway prototypes.

Big-bang testing is an integration testing approach in which all modules are integrated at once and the complete system is tested as a whole. In this method, individual modules may be unit tested, but integration testing begins only after all modules are combined.

The advantage of big-bang testing is that it is simple to implement. There is no need to create stubs or drivers because all modules are available. The system is tested in its final integrated form.

However, big-bang testing has major disadvantages. If defects occur, it becomes difficult to locate the source because many modules are integrated simultaneously. Debugging becomes complex and time-consuming. It also delays integration testing until late in the project, increasing risk.

Big-bang testing is suitable only for small systems where the number of modules is limited. For large systems, incremental integration approaches are preferred.

Big-Bang Testing is an integration strategy where integration testing is not performed until all modules are completed.

The Approach:

1. Develop Module A, B, C, D, E...
2. Wait until *all* are finished.
3. Link them all together simultaneously.
4. Run the system and see what happens.

NOTE

The Disadvantages:

While this requires zero planning overhead, it is catastrophic for large systems. If the system crashes, the root cause could be in any of the 100 modules or any of the 1,000 connections between them. Fault Isolation is nearly impossible. It creates a "bottleneck" at the end of the project where debugging takes unpredictable amounts of time. It is typically only suitable for very small, experimental projects.

21.3.10 Sandwich Testing

Sandwich testing is an integration testing approach that combines both top-down and bottom-up integration. In this method, integration occurs simultaneously from the top level and the bottom level, meeting in the middle. It is called sandwich because integration occurs from both ends like the two layers of a sandwich.

In top-down integration, higher-level modules are tested first, using stubs for lower-level modules. In bottom-up integration, lower-level modules are tested first, using drivers for higher-level modules. Sandwich testing combines these approaches.

The advantage of sandwich testing is that it reduces the disadvantages of purely top-down or purely bottom-up testing. It allows early testing of both high-level logic and low-level utility functions. It also reduces the number of stubs and drivers required.

However, sandwich testing requires careful planning. Integration must be coordinated from both ends. The middle layer becomes critical because it connects top and bottom parts.

Sandwich Testing (or Hybrid Integration) combines the strengths of Top-Down and Bottom-Up integration approaches. In this strategy, the system is viewed as three layers: the top layer (UI/Logic), the middle layer, and the bottom layer (Database/Drivers). Testing proceeds simultaneously from the top down and from the bottom up, meeting in the middle. This allows the team to test the critical user interfaces early (Top-Down benefit) while simultaneously verifying the stability of the low-level utilities (Bottom-Up benefit). It is highly efficient but requires complex coordination.

Sandwich Testing (or Hybrid Integration) serves as a compromise between Top-Down and Bottom-Up integration strategies, addressing the shortcomings of both.

The Context:

- Top-Down integration starts with the UI/Main Menu but requires "Stubs" to simulate lower-level databases/logic.
- Bottom-Up integration starts with the utility/database layers but requires "Drivers" to simulate the user commands.

NOTE

The Sandwich Approach:

In large projects, the "Middle Layer" (Business Logic) is often the most critical. Sandwich testing targets this layer by working from both ends simultaneously:

1. Top Layer: The UI is tested using Stubs to simulate the middle layer.
2. Bottom Layer: The Database/Utility layer is tested using Drivers to simulate the middle layer.
3. Middle Layer: As the middle layer is built, it is "sandwiched" between the working UI and the working Database.

This allows parallel development and testing teams to converge on the center, maximizing efficiency.

21.3.11 Critical Path First

Critical path first is an integration testing strategy where the most important or critical functionality of the system is integrated and tested first. The critical path refers to the sequence of operations that must work correctly for the system to function. For example, in an e-commerce system, the critical path may include product selection, cart update, checkout, payment processing, and order confirmation.

The advantage of critical path first is that it reduces risk. By testing the most important functionality early, major defects are identified early. It also ensures that the core system is working before less important features are integrated.

Critical path first is especially useful in systems where certain features are mission-critical, such as banking transactions, healthcare systems, and airline booking systems.

This strategy prioritizes the integration of modules based on the most critical functional paths of the system. Instead of integrating geographically (all UI first) or structurally (all database first), the team integrates the specific modules required to make the core feature work (e.g., "User Checkout"). This ensures that the most important business functions are tested and stabilized earliest in the lifecycle, reducing the risk of a major failure near the release date.

This is a risk-based integration strategy widely used in Agile and Continuous Delivery environments.

Prioritization by Risk:

Instead of integrating by architectural layer (all databases, then all logic, then all UI), the team integrates by Functional Thread. They identify the "Critical Path"—the core functionality without which the system is useless.

NOTE

- *Example:* In an E-commerce site, the critical path is: Login -> Search -> Add to Cart -> Checkout.
- Modules required for this specific path are prioritized, integrated, and tested first.

Benefits:

- Early Viability: We get a working "skeleton" of the system very early.
- Risk Reduction: If the project runs out of time, we can still ship the core functionality, dropping only non-essential features (like "Wishlist" or "Reviews").

21.3.12 Sub System Testing

Subsystem testing is testing performed on a subsystem, which is a collection of modules that work together to provide a major function. A subsystem is larger than a module but smaller than the complete system. For example, in an enterprise system, subsystems may include inventory subsystem, billing subsystem, and reporting subsystem.

Subsystem testing verifies that all modules within the subsystem interact correctly. It focuses on subsystem-level functionality, performance, and reliability. It may also test subsystem interfaces with other subsystems. Subsystem testing is important because large systems are often built as multiple subsystems. Testing subsystems individually helps isolate defects and reduce complexity.

As systems grow larger, they are often divided into Sub-Systems. For example, an ERP system might have an "Accounting Sub-system" and an "HR Sub-system." Sub-system testing is a form of large-scale integration testing where these massive independent blocks are tested in isolation before being combined into the final full system. It acts as a checkpoint to ensure that major functional areas are stable.

In massive enterprise architectures (like ERP systems, Avionics, or Operating Systems), the leap from "Integration Testing" to "System Testing" is too big. There is an intermediate step called Sub-System Testing.

Decomposition:

A large system is broken down into independent functional areas (Sub-systems), such as "Inventory Management," "Payroll," and "CRM."

- Before these three giants interact, each sub-system is tested as a standalone entity.
- This ensures that the "Payroll" system handles salaries correctly before we worry about whether it can talk to the "Bank Interface" system.

NOTE

- This is essentially "System Testing" applied to a fraction of the total application scope.

21.3.13 System Testing

System Testing is the testing of a complete and fully integrated software product. It is performed by the independent QA team in an environment that mimics production. The scope of System Testing is broad: it verifies that the entire package meets the specified requirements. This includes Functional testing (does the feature work?) and Non-Functional testing (Performance, Security, Reliability, Usability). Unlike Integration testing, which checks technical interfaces, System testing checks business behavior.

System testing is testing of the complete integrated software system. It is performed after integration testing. The objective is to verify that the system meets all functional and non-functional requirements. System testing checks end-to-end functionality. It ensures that the system works correctly in the intended environment. It includes testing of all modules, interfaces, external integrations, and system behaviour under real conditions.

System testing includes various types such as functional testing, performance testing, security testing, usability testing, compatibility testing, and recovery testing. It verifies that the system is reliable, secure, and user-friendly. System testing is usually performed by an independent testing team. It is critical because it validates the system before acceptance testing and deployment.

System Testing is the validation of the complete, fully integrated software product. It is the final testing phase conducted by the technical team before the software is handed over to the user.

Black-Box Perspective:

Unlike Unit or Integration testing, System Testing ignores the internal code. It treats the software as a "Black Box." The tester inputs data via the actual user interface and checks the output.

Scope of System Testing:

It covers the entire requirements specification:

1. Functional Testing: Verifying every feature against the Business Requirements.
2. Performance Testing: Load testing (can it handle 10,000 users?) and Stress testing (when does it break?).
3. Security Testing: Penetration testing to find vulnerabilities.
4. Reliability/Recovery Testing: Pulling the power plug on the server and seeing if data is lost.

NOTE

5. Usability Testing**:** Checking navigation and user experience.

Activity Title: Preparing a Basic Test Plan

Objective:
To understand how testing activities are planned before test execution.

Materials Required:

- Sample software project
- Test-plan template
- Notebook

Procedure/Steps:

7. Select a simple software project.
8. Define testing objectives and scope.
9. Identify features to be tested and not tested.
10. Identify resources, responsibilities, tools, and test environment.
11. Prepare a basic test schedule.
12. Identify major risks and assumptions.

Outcome:
The learner will be able to prepare a basic software test plan and understand the importance of systematic test planning.

21.3.14 Testing Stages

Testing stages refer to the progression of testing activities across the software development life cycle. Testing is performed in stages to ensure systematic quality evaluation at every level.

Testing stages begin with early verification such as proposal testing, requirement testing, and design testing. These stages ensure that documents and plans are correct. Next comes code-level verification through code review and unit testing. After that, module testing and integration testing verify combined components. Subsystem testing verifies major functional groups. Finally, system testing validates the complete product.

Each stage has its own objectives and deliverables. Early stages focus on preventing defects. Later stages focus on detecting defects through execution. Together, these stages provide complete coverage.

NOTE

Testing stages also support defect tracking and quality improvement. Defects found at each stage are recorded, analyzed, and corrected. This helps improve processes and reduce future defects.

The "Testing Stages" represent the chronological lifecycle view of the types discussed above. A mature SQA process enforces Quality Gates between these stages.

1. Unit Stage: Developers verify logic. *Gate: Code must compile and pass all unit tests.*
2. Integration Stage: QA validates interfaces. *Gate: Data must flow correctly between core modules.*
3. System Stage: QA validates requirements. *Gate: No critical defects open.*
4. Acceptance Stage: Users validate business value. *Gate: User sign-off.*

This staged approach ensures that expensive system-level tests are not wasted on buggy code that should have failed a unit test weeks ago.

Check Your Progress 21.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Q1. Testing performed to check individual program units or functions is called:
A) System Testing
B) Unit Testing
C) Acceptance Testing
D) Volume Testing

_____

Q2. Integration testing is performed after unit testing.

_____

Q3. Testing where all modules are integrated at once and tested together is called _____ testing.

_____

Q4. Write any two types of testing performed during SDLC.

Answer (Any two):

Unit testing

Integration testing

System testing

NOTE

Module testing

Sub-system testing

Acceptance testing

Q5. Sandwich testing is a combination of:

A) Unit and system testing

B) Top-down and bottom-up integration testing

C) Manual and automation testing

D) Regression and smoke testing

21.4 SUMMARY

This unit explained the different types of testing performed across the software development life cycle. It began with the concept of testing and its importance in quality assurance. It covered early-stage testing such as proposal testing, requirement testing, and design testing. It explained static testing through code review and dynamic testing through unit testing and module testing. It discussed integration testing and special integration strategies such as big-bang testing, sandwich testing, and critical path first. It also explained subsystem testing and system testing as higher-level testing activities. Finally, it described testing stages as a systematic progression of testing activities. Overall, understanding these types of testing helps in planning effective testing strategies and ensures the development of high-quality software products.

We have conducted a deep exploration of the Types of Testing. We established that testing is a continuous activity that changes form as the project evolves. We began with the Static Testing of Proposals, Requirements, and Designs emphasizing that verification of these documents is the most cost-effective way to improve quality. We then analyzed the Dynamic Testing hierarchy. We saw how Unit Testing ensures the reliability of the smallest building blocks through isolation and mocking. We examined various Integration Strategies, noting that while Big-Bang is risky, Sandwich Testing and Critical Path integration offer robust, parallel workflows for modern development. Finally, we looked at System Testing as the comprehensive validation of the fully assembled product against functional and non-functional requirements. This unit reinforces the concept that quality is constructed layer by layer, with each testing type acting as a safety net for the one before it.

21.5 GLOSSARY

- **Static Testing:** A form of testing where the software is not executed. It involves reviews, walkthroughs, and inspections of documents (requirements, design) or source code.
- **Dynamic Testing:** Testing that involves the execution of the software component or system.
- **Cyclomatic Complexity:** A software metric used to verify the complexity of a program. It directly correlates to the difficulty of testing and maintaining the code.
- **Big-Bang Integration:** An integration testing strategy where all components of the system are integrated simultaneously, and then tested as a whole.
- **Sandwich Testing:** A hybrid integration testing strategy that combines top-down and bottom-up approaches, using both stubs and drivers to test the middle layers.
- **Stub:** A skeletal implementation of a software component used in Top-Down testing to simulate a lower-level module that is not yet developed.
- **Driver:** A test harness used in Bottom-Up testing to call a lower-level module and provide it with inputs, simulating a higher-level calling module.
- **Critical Path:** The sequence of stages determining the minimum time needed for an operation; in testing, it refers to the essential functional flow of the application.

21.6 ANSWER TO CHECK YOUR PROGRESS

Q1. Answer: B

Explanation: Unit testing verifies the smallest testable parts of software such as functions, methods, or modules.

Q2. Answer: True

Explanation: Integration testing checks interaction between combined modules after unit testing verifies individual modules.

Q3. Answer: Big-Bang

Explanation: Big-Bang testing integrates all components at once, which may make defect isolation difficult.

Q4. Explanation: Different testing types ensure quality at different levels, from small components to complete system validation.

NOTE

Q5. Answer: B

Explanation: Sandwich (hybrid) testing combines top-down and bottom-up approaches to gain advantages of both integration methods.

21.7 SUGGESTED READINGS - OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3800	

21.8 REFERENCE

- Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill Education.
- Sommerville, I. (2015). *Software Engineering* (10th ed.). Pearson Education.
- Beizer, B. (1990). *Software Testing Techniques* (2nd ed.). Van Nostrand Reinhold.
- Mathur, A. P. (2013). *Foundations of Software Testing* (2nd ed.). Pearson.

21.9 EXERCISE

Part–1

A. Multiple Choice Questions

Q1. System testing is performed on:
A) Individual function
B) Complete integrated system
C) Only database
D) Only UI

Q2. System testing is mainly:
A) White-box testing
B) Black-box testing
C) Mutation testing

NOTE

D) Code review

Q3. Functional and non-functional tests are part of:
A) Unit testing
B) System testing
C) Code inspection
D) Requirement review

Q4. System testing is performed after:
A) Requirement gathering
B) Integration testing
C) Deployment
D) Maintenance

Q5. Acceptance testing is usually done after:
A) Unit testing
B) System testing
C) Code review
D) Design review

ANSWERS WITH EXPLANATION:

1. **Answer:** B
   Explanation: System testing validates full system behavior.
2. **Answer:** B
   System testing focuses on external behavior.
3. **Answer:** B
   System testing includes both.
4. **Answer:** B
   System testing comes after integration.
5. **Answer:** B
   Acceptance is final before release.

B. Small Answer Questions

1. Define system testing.
2. Write two objectives of system testing.
3. What is functional system testing?
4. What is non-functional system testing?
5. Why is system testing important?

Part–2

Long Answer Type Questions (200 words)

NOTE

1. Explain system testing and its scope.
2. Discuss functional and non-functional system testing.
3. Explain system test planning and test execution.
4. Discuss system testing challenges.
5. Explain system testing deliverables.

Part–3

Long Answer Type Questions (200 words)

1. Explain system testing in detail with examples.
2. Discuss types of system testing in software projects.
3. Explain system testing process and documentation.
4. Discuss system testing in Agile development.
5. Explain how system testing ensures software quality.

NOTE

UNIT–22
SPECIAL TESTS

UNIT STRUCTURE

22.1 Introduction
22.2 Objective
22.3 Special Tests
  22.3.1 About Special Tests
  22.3.2 GUI testing
  22.3.3 Compatibility Testing
  22.3.4 Security Testing
  22.3.5 Performance Testing
  22.3.6 Volume Testing
  22.3.7 Stress Testing
  22.3.8 Recovery Testing
22.4 Summary
22.5 Glossary
22.6 Answer to check your progress
22.7 Suggested readings - OER
22.8 Reference
22.9 Exercise

22.1 INTRODUCTION

Software Quality Assurance and Testing (SQAT) is not limited to verifying functional correctness alone; it also focuses on ensuring that software systems operate reliably, securely, efficiently, and consistently under diverse conditions. As software applications become increasingly complex, distributed, and user-centric, traditional functional testing is no longer sufficient to guarantee overall software quality. This necessity has led to the emergence of Special Tests, which address non-functional and quality-related attributes of software systems.

Special Tests are designed to evaluate specific characteristics of software such as usability, compatibility, performance, security, robustness, and reliability. These tests go beyond verifying whether the software performs its intended functions and instead assess how well the software performs those functions under varying environments, workloads, and stress conditions. In modern software development, especially in domains such as e-commerce, healthcare, banking,

NOTE

defense, and cloud computing, special testing plays a critical role in minimizing risks and ensuring user satisfaction.

Special testing techniques help uncover defects that may not surface during unit, integration, or system testing. For example, a system may function correctly under normal conditions but fail when subjected to heavy user load, malicious attacks, or unexpected failures. Such defects can have severe consequences, including financial losses, data breaches, reputational damage, and system downtime. Therefore, special tests are an essential component of comprehensive software quality assurance strategies.

This unit introduces the concept of Special Tests, discusses their objectives, and explains various types such as GUI testing, compatibility testing, security testing, performance testing, volume testing, stress testing, and recovery testing. Each type of test is explained in detail with its purpose, techniques, challenges, and importance in ensuring high-quality software products.

Software Quality Assurance and Testing is not limited to verifying whether a system works according to its functional requirements. Modern software systems operate in highly dynamic environments, interact with diverse hardware and software platforms, handle large volumes of data, and are exposed to numerous security threats. Under such circumstances, traditional functional testing alone is insufficient to assure the overall quality of the software product. This gap is addressed by a category of testing activities commonly referred to as special tests. Special tests focus on non-functional aspects of software such as usability, performance, reliability, security, compatibility, and recoverability. These tests are critical because they directly influence user satisfaction, operational stability, and organizational trust in the software system.

Special tests are usually conducted after the core functional behavior of the software has been verified, although in practice many of them are planned in parallel with functional testing. The results of special testing often determine whether the software is ready for deployment in real-world conditions. For example, a system may perform all required functions correctly but still fail in the market due to slow response time, inability to handle peak loads, poor user interface design, or vulnerability to security attacks. Therefore, special tests play a decisive role in certifying the readiness, robustness, and fitness of software for production use. This unit introduces the concept of special testing and discusses important types such as GUI testing, compatibility testing, security testing, performance testing, volume testing, stress testing, and recovery testing.

22.2 OBJECTIVE

After studying this unit, you will be able to:

1. Understand the concept and significance of Special Tests in Software Quality Assurance and Testing.
2. Differentiate between functional testing and special (non-functional) testing approaches.
3. Explain the need for GUI testing to ensure usability, consistency, and user satisfaction.
4. Understand compatibility testing and its role in ensuring software operability across diverse platforms and environments.
5. Describe security testing techniques to identify vulnerabilities and protect software systems from threats.
6. Analyze performance-related tests, including performance, volume, and stress testing.
7. Explain recovery testing and its importance in ensuring system resilience and fault tolerance.
8. Develop an overall understanding of how special tests contribute to reliable, secure, and high-performing software systems.

22.3 SPECIAL TESTS

Special tests are a group of testing techniques that focus on validating non-functional requirements of software systems. Unlike functional testing, which verifies what the system does, special testing concentrates on how the system behaves under different conditions. These tests assess characteristics such as user interface quality, system performance, data handling capacity, resistance to security threats, and the ability to recover from failures. Special tests are particularly important in large-scale and mission-critical systems where failure can lead to significant financial loss, reputational damage, or safety risks. Each type of special test addresses a specific quality attribute, and together they provide a holistic view of software quality.

22.3.1 About Special Tests

Special Tests refer to a category of testing techniques that focus on evaluating non-functional aspects of software systems. Unlike functional testing, which validates what the system does, special testing examines how the system behaves under various conditions. These tests assess attributes such as usability, performance, security, scalability, stability, and reliability.

NOTE

Special testing is typically performed after functional correctness has been established, although some types may be executed iteratively throughout the development lifecycle. These tests are particularly important for mission-critical and user-facing applications where failures can lead to serious consequences. The outcomes of special tests help stakeholders make informed decisions about system readiness, deployment, and maintenance.

Activity Title: Exploring Special Testing Requirements

Objective:
To understand the purpose of special tests used for specific software quality requirements.

Materials Required:

- Computer/mobile device
- Sample application
- Testing checklist

Procedure/Steps:

1. Select a web or mobile application.
2. Identify tests related to usability, security, performance, compatibility, recovery, or reliability.
3. Select any three special tests.
4. Design two test scenarios for each selected test.
5. Record the expected results.
6. Discuss why these tests are important for the selected application.

Outcome:
The learner will understand different special testing requirements and their practical applications.

22.3.2 GUI Testing

Graphical User Interface (GUI) Testing focuses on validating the visual and interactive components of a software application. The GUI is the primary point of interaction between users and the system; therefore, its quality significantly influences user experience and acceptance.

GUI testing verifies elements such as buttons, menus, icons, dialog boxes, text fields, labels, images, colors, fonts, and layouts. It ensures that these elements are correctly displayed, aligned, and function as

NOTE

intended across different screen resolutions and devices. GUI testing also evaluates usability aspects such as ease of navigation, accessibility, consistency, and responsiveness.

Key aspects of GUI testing include layout testing, functional testing of UI elements, usability testing, accessibility testing, and cross-browser or cross-device UI testing. Automated GUI testing tools are often used to improve efficiency, though manual testing remains important for subjective usability evaluation.

Graphical User Interface testing is concerned with verifying the visual and interactive elements of a software application. The GUI is the primary point of interaction between the user and the system, and its quality significantly influences user perception and acceptance of the software. GUI testing ensures that screens, windows, menus, buttons, icons, text fields, and other interface components function correctly and appear as intended. It also checks whether the interface follows design specifications, usability standards, and accessibility guidelines.

In GUI testing, testers evaluate aspects such as consistency of layout, alignment of elements, readability of text, correctness of labels, and responsiveness of interactive controls. They also verify navigation flow to ensure that users can move smoothly between screens without confusion or errors. Another important aspect of GUI testing is validation of error messages and system feedback, ensuring that users receive clear and meaningful information when an error occurs. Effective GUI testing improves usability, reduces user errors, and enhances overall satisfaction, making it a crucial component of special testing.

22.3.3 Compatibility Testing

Compatibility Testing ensures that a software application works correctly across different environments, platforms, devices, browsers, operating systems, hardware configurations, and network conditions. With the proliferation of devices and platforms, compatibility testing has become increasingly critical.

This testing type verifies whether the software behaves consistently when executed in varied environments. It identifies issues related to configuration dependencies, platform-specific limitations, and integration inconsistencies. Compatibility testing may include backward compatibility testing, forward compatibility testing, hardware compatibility testing, software compatibility testing, and network compatibility testing.

Effective compatibility testing improves software reliability, reduces user complaints, and enhances market reach by ensuring that the application is accessible to a broad user base. Compatibility testing

NOTE

evaluates how well a software application functions across different environments, platforms, and configurations. In today's heterogeneous computing landscape, users may access the same software using different operating systems, browsers, devices, screen resolutions, and hardware configurations. Compatibility testing ensures that the software behaves consistently and reliably across these variations.

This type of testing involves running the application in multiple environments to identify issues related to display, functionality, or performance. For example, a web application may work correctly in one browser but fail in another due to differences in rendering engines or supported features. Similarly, a mobile application may behave differently on various devices because of differences in hardware capabilities or operating system versions. Compatibility testing helps organizations deliver software that reaches a wider audience without compromising quality, thereby improving market acceptance and reducing post-release support issues.

22.3.4 Security Testing

Security Testing focuses on identifying vulnerabilities, threats, and risks in a software system to prevent unauthorized access, data breaches, and malicious attacks. The objective of security testing is to ensure that data and system resources are protected against potential threats.

Security testing evaluates aspects such as authentication, authorization, confidentiality, integrity, availability, and non-repudiation. Common security testing techniques include vulnerability scanning, penetration testing, risk assessment, security auditing, and ethical hacking. Security testing is essential in applications handling sensitive information such as financial data, personal records, and confidential communications. A robust security testing strategy helps organizations comply with regulatory requirements and build user trust.

Security testing focuses on identifying vulnerabilities and weaknesses that could be exploited by unauthorized users or malicious attacks. The objective of security testing is to ensure that the software protects data, maintains confidentiality, preserves integrity, and enforces proper authentication and authorization mechanisms. With the increasing frequency of cyber attacks and data breaches, security testing has become one of the most critical aspects of Software Quality Assurance.

During security testing, testers examine the system for potential threats such as unauthorized access, data leakage, injection attacks, and improper session management. They also evaluate the effectiveness of encryption, password policies, and access controls. Security testing is not limited to identifying technical flaws; it also assesses whether the

NOTE

system complies with relevant security standards and regulations. By addressing security issues early, organizations can prevent costly breaches, protect user trust, and ensure legal and ethical compliance.

22.3.5 Performance Testing

Performance Testing evaluates how a software system performs under expected workloads. It focuses on attributes such as response time, throughput, resource utilization, and scalability. Performance testing ensures that the system meets performance requirements and provides a satisfactory user experience.

This testing identifies performance bottlenecks, inefficient resource usage, and scalability limitations. Performance testing is especially important for systems with high user traffic, real-time processing requirements, or strict service-level agreements (SLAs). Performance testing results help developers optimize system architecture, improve efficiency, and ensure smooth operation under normal conditions.

Performance testing assesses how a software system performs under expected and varying workloads. The primary goal of performance testing is to ensure that the system meets specified response time, throughput, and resource utilization requirements. Performance is a key quality attribute, especially for systems that support large numbers of users or process time-critical transactions.

In performance testing, the system is evaluated under normal, peak, and sometimes extreme conditions to observe its behavior. Testers measure parameters such as response time, processing speed, memory usage, and CPU utilization. Performance testing helps identify bottlenecks, inefficient algorithms, and resource constraints that could degrade system performance. By optimizing performance based on test results, developers can deliver software that provides a smooth and reliable user experience even under demanding conditions.

22.3.6 Volume Testing

Volume Testing examines the behavior of a software system when subjected to large volumes of data. The objective is to ensure that the system can handle significant amounts of data without performance degradation, data loss, or failure.

Volume testing focuses on database performance, data processing capabilities, storage management, and system stability. It is particularly important for applications involving large datasets, such as enterprise systems, data warehouses, and big data applications. This testing helps identify issues related to memory leaks, data handling inefficiencies, and storage limitations.

NOTE

Volume testing examines the system's ability to handle large volumes of data. Many software applications are required to store, process, and retrieve massive amounts of information, such as databases in banking systems, e-commerce platforms, or enterprise resource planning systems. Volume testing ensures that the system continues to function correctly when the size of data grows significantly.

This type of testing focuses on identifying issues related to data storage, retrieval efficiency, indexing, and data integrity. It also helps detect problems such as slow response times, data corruption, or system crashes caused by excessive data volume. Volume testing is particularly important for applications expected to scale over time, as it provides insight into how the system will behave as data accumulates during long-term use.

22.3.7 Stress Testing

Stress Testing evaluates the behavior of a software system under extreme conditions, beyond its normal operational limits. The goal is to determine the system's breaking point and observe how it fails. Stress testing helps identify weaknesses related to resource exhaustion, concurrency issues, and error handling. It also evaluates system robustness and recovery capabilities. This testing is crucial for mission-critical systems where failure can have severe consequences.

Stress testing evaluates the behavior of a software system under extreme conditions that exceed normal operational limits. The purpose of stress testing is to determine the system's breaking point and observe how it fails. By intentionally subjecting the system to high loads, limited resources, or abnormal conditions, testers can identify weaknesses that may not be apparent under normal usage.

Stress testing helps answer critical questions such as whether the system fails gracefully, whether data is preserved during failure, and how quickly the system can recover once the stress condition is removed. This type of testing is essential for systems that must maintain stability and reliability during peak usage or unexpected surges in demand. Stress testing provides valuable information for risk assessment and capacity planning.

22.3.8 Recovery Testing

Recovery Testing verifies the ability of a software system to recover from crashes, hardware failures, network outages, or other disruptions. The objective is to ensure that the system can restore operations and data with minimal impact. Recovery testing evaluates backup mechanisms, fault tolerance, data recovery procedures, and system restart processes. It ensures that the system meets reliability and availability requirements.

NOTE

Recovery testing focuses on verifying the system's ability to recover from failures such as hardware crashes, software faults, network interruptions, or power outages. The objective of recovery testing is to ensure that the system can restore normal operations within an acceptable time frame and without loss of critical data.

During recovery testing, failures are deliberately introduced to observe how the system responds. Testers evaluate backup mechanisms, data recovery procedures, and fault tolerance features. Recovery testing is especially important for mission-critical systems where downtime or data loss can have severe consequences. By validating recovery capabilities, organizations can ensure business continuity and enhance the overall reliability of their software systems.

Check Your Progress

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Fill-in-the-Blank Questions:

_____ testing focuses on evaluating the graphical elements and user interaction of a software application.

Compatibility testing ensures that software works correctly across different _____, devices, and environments.

_____ testing is performed to identify vulnerabilities and protect software systems from threats.

The primary objective of stress testing is to determine the system's _____ point.

Recovery testing verifies the system's ability to _____ after failure.

True/False Questions:

GUI testing deals only with backend logic and database operations. _____

Performance testing evaluates response time, throughput, and resource utilization. _____

Volume testing focuses on system behavior under extreme workload conditions. _____

Security testing is important only for banking and financial applications. _____

Recovery testing ensures data integrity after system crashes. _____

NOTE

Q1. GUI testing mainly focuses on:
A) Database tables only
B) User interface components like buttons, menus, and screens
C) Hardware device testing
D) Compiler speed

_____

Q2. Security testing ensures that the system protects data from unauthorized access.

_____

Q3. Testing to check system behavior under heavy load is called _____ testing.

_____

Q4. Write any two types of special tests.

Answer (Any two):

Compatibility testing

Security testing

Performance testing

Volume testing

Recovery testing

Stress testing

22.4 SUMMARY

Special Tests play a vital role in ensuring the overall quality of software systems by addressing non-functional requirements. These tests complement functional testing by evaluating usability, compatibility, security, performance, robustness, and reliability. Through GUI, compatibility, security, performance, volume, stress, and recovery testing, organizations can deliver high-quality, dependable, and user-friendly software products.

This unit on Special Tests in Software Quality Assurance and Testing highlights the importance of evaluating software beyond its functional correctness. Modern software systems operate in complex, dynamic, and often unpredictable environments where quality is influenced by factors such as usability, performance, security, compatibility, and reliability. Special tests are designed to assess these non-functional attributes and ensure that software performs effectively under real-world conditions. They complement functional testing by focusing on

NOTE

how the system behaves rather than what it does, thereby providing a more comprehensive assurance of software quality.

The unit explains that special testing is essential for identifying potential risks that may not surface during routine functional testing. Graphical User Interface testing ensures that the user interface is intuitive, consistent, and user-friendly, directly affecting user satisfaction and acceptance. Compatibility testing confirms that the software operates correctly across different platforms, devices, and environments, which is critical in today's heterogeneous computing landscape. Security testing addresses the growing concern of protecting software systems from unauthorized access and cyber threats by validating confidentiality, integrity, and access control mechanisms.

The unit also emphasizes the role of performance-related tests in ensuring system efficiency and stability. Performance testing evaluates system responsiveness and resource utilization under expected workloads, while volume testing focuses on the system's ability to handle large amounts of data without failure. Stress testing examines software behavior under extreme conditions to identify breaking points and assess system robustness. Finally, recovery testing ensures that the software can recover gracefully from failures and resume normal operations with minimal data loss and downtime.

22.5 GLOSSARY

- Special Tests: Testing techniques focused on non-functional attributes of software.
- GUI Testing: Testing of graphical user interface elements.
- Compatibility Testing: Testing software across multiple environments.
- Security Testing: Testing to identify vulnerabilities and protect systems.
- Performance Testing: Testing system performance under expected load.
- Stress Testing: Testing system behavior under extreme conditions.

22.6 ANSWERS TO CHECK YOUR PROGRESS – POSSIBLE ONE

Fill-in-the-Blank:

GUI

platforms

NOTE

Security

breaking

recover

True/False:

False

True

False

False

True

Q1. Answer: B

Explanation: GUI testing checks the correctness, appearance, and functionality of user interface elements.

Q2. Answer: True

Explanation: Security testing checks confidentiality, integrity, authentication, authorization, and data protection.

Q3. Answer: Stress

Explanation: Stress testing evaluates how the system behaves when workload exceeds normal limits.

Q4. Explanation: Special tests focus on non-functional quality attributes such as performance, security, reliability, and compatibility.

Q5. Answer: B

Explanation: Recovery testing ensures that the system can restore data and resume operation after failure.

22.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/User/UnitUrls?unitId=3801	

NOTE

22.8 REFERENCE

- IEEE Std 829-2008: IEEE Standard for Software and System Test Documentation.
- ISO/IEC 25010:2011 – Systems and Software Quality Models.
- Beizer, B. (2003). Software Testing Techniques. Dreamtech Press.
- Kaner, C., Falk, J., & Nguyen, H. Q. (1999). Testing Computer Software. Wiley.

22.9 EXERCISE

Part-1

A. Objective Type Questions:

1. Which of the following testing types evaluates the user interface of an application?
   A) Unit Testing
   B) GUI Testing
   C) Integration Testing
   D) Regression Testing

   Answer: B) GUI Testing

2. Which testing ensures software works across different operating systems and browsers?
   A) Performance Testing
   B) Stress Testing
   C) Compatibility Testing
   D) Recovery Testing

   Answer: C) Compatibility Testing

3. Which testing focuses on identifying system vulnerabilities?
   A) Security Testing
   B) Volume Testing
   C) GUI Testing
   D) Alpha Testing

   Answer: A) Security Testing

4. Stress testing is mainly used to:
   A) Improve user interface design
   B) Test system behavior under extreme load
   C) Verify functional requirements
   D) Test data recovery procedures

   Answer: B) Test system behavior under extreme load

NOTE

5. Recovery testing ensures:
   A) Faster execution speed
   B) Correct GUI alignment
   C) System restoration after failure
   D) Cross-platform compatibility

Answer: C) System restoration after failure

Part-2

B- Short Answer Type Questions:

1. Define special tests in software testing.
2. Explain the importance of GUI testing.
3. What is compatibility testing?
4. Why is security testing essential?
5. What is the purpose of recovery testing?

Part-3

C- Long Answer Type Questions:

1. Discuss the role of special tests in ensuring software quality.
2. Explain GUI testing and compatibility testing in detail.
3. Describe security testing techniques and their importance.
4. Differentiate between performance testing, volume testing, and stress testing.
5. Explain recovery testing with suitable examples.

UNIT–23
INSTALLATION TESTING

UNIT STRUCTURE

23.1 Introduction
23.2 Objective
23.3 Installation Testing
  23.3.1 Requirement Testing
  23.3.2 Regression Testing
  23.3.3 Error Handling Testing
  23.3.4 Manual Support Testing
  23.3.5 Intersystem Testing
  23.3.6 Control Testing
  23.3.7 Smoke Testing
  23.3.8 Adhoc Testing
  23.3.9 Parallel Testing
23.4 Summary
23.5 Glossary
23.6 Answer to check your progress
23.7 Suggested readings - OER
23.8 Reference
23.9 Exercise

23.1 INTRODUCTION

Installation Testing is a critical component of Software Quality Assurance and Testing (SQAT) that focuses on verifying whether a software application is installed correctly and functions as expected in its target environment. In real-world scenarios, software is deployed across diverse hardware configurations, operating systems, network environments, and user settings. Even if software passes unit, integration, and system testing, failures during installation can lead to user dissatisfaction, operational disruptions, and increased support costs.

Installation testing ensures that the complete installation process—starting from setup execution to successful configuration and readiness for use—meets specified requirements. It evaluates installation packages, scripts, configuration files, environment dependencies, and uninstallation procedures. This type of testing is especially important

NOTE

for commercial software, enterprise systems, cloud-based applications, and packaged products where users independently install software without direct developer intervention.

In modern software development, installation testing has gained more importance due to continuous delivery, frequent updates, patches, and version upgrades. Faulty installation processes can cause system crashes, data corruption, incomplete configurations, or security vulnerabilities. Therefore, installation testing helps ensure reliability, usability, and maintainability of software products in real deployment environments.

This unit explains the concept of installation testing, its objectives, and various associated testing techniques such as requirement testing, regression testing, error handling testing, manual support testing, intersystem testing, control testing, smoke testing, ad hoc testing, and parallel testing.

Installation testing is an important phase of Software Quality Assurance that focuses on verifying whether a software product can be successfully installed, configured, upgraded, and uninstalled in its target environment. Even if a software system is functionally correct and performs well, it cannot be considered ready for use unless users are able to install it without difficulty. Installation is the first real interaction between the end user and the software, and any failure at this stage can lead to user dissatisfaction, loss of confidence, and rejection of the product. Therefore, installation testing plays a crucial role in determining the usability and acceptance of software.

Installation testing ensures that the software behaves correctly under different installation scenarios such as fresh installation, upgrade from a previous version, installation on different platforms, and installation with varying system configurations. It also verifies that all required components, dependencies, and configurations are properly handled during the installation process. In modern software systems, where applications are deployed across desktops, servers, mobile devices, and cloud environments, installation testing becomes even more significant. This unit discusses the concept of installation testing and explains various testing activities associated with it, highlighting their role in delivering a reliable and user-friendly software product.

23.2 OBJECTIVE

The primary objective of this unit is to provide a clear understanding of installation testing and its importance in ensuring overall software quality. After studying this unit, learners should be able to understand why installation testing is a critical part of the testing life cycle and how it contributes to user satisfaction and system reliability. The unit aims

NOTE

to explain different types of tests that are performed during installation testing and the specific quality aspects they address.

The objectives of this unit are to:

1. Understand the concept and importance of installation testing in software quality assurance.
2. Identify common issues associated with software installation and deployment.
3. Explain different testing activities involved during installation testing.
4. Understand the role of requirement testing and regression testing during installation.
5. Describe error handling and manual support testing in installation scenarios.
6. Explain intersystem, control, smoke, ad hoc, and parallel testing in the context of installation.
7. Ensure software readiness, reliability, and stability after installation in real environments.

Another objective is to help learners recognize common installation-related problems such as missing files, incorrect configurations, version conflicts, and improper error handling. By understanding these issues, learners can appreciate the need for structured and systematic installation testing. This unit also aims to equip students with conceptual knowledge that can be applied in real-world projects, enabling them to plan and execute installation testing effectively. Overall, the objective is to strengthen the learner's ability to evaluate software readiness from a deployment and operational perspective.

23.3 INSTALLATION TESTING

Installation Testing refers to the systematic evaluation of the installation process of a software application. It verifies that the application installs correctly according to specified requirements and operates properly once installed. Installation testing covers fresh installations, upgrades, patches, rollbacks, and uninstallation procedures.

This testing ensures that the installer correctly checks system prerequisites, handles errors gracefully, configures necessary settings, and provides clear messages to users. Installation testing also validates documentation, licensing mechanisms, and post-installation configuration steps.

NOTE

Installation testing is a type of testing that verifies whether a software application can be installed correctly and operates as expected in its target environment. It checks the complete installation process, including setup programs, configuration settings, dependency management, and system integration. Installation testing also ensures that the software can be safely uninstalled without leaving residual files or corrupting the system. This testing is especially important for commercial and large-scale software products that are distributed to a wide range of users with varying system configurations.

Installation testing covers multiple dimensions of quality, such as correctness, reliability, usability, and compatibility. It often involves testing different installation modes, such as default installation, custom installation, and silent installation. In addition, installation testing may include testing upgrades, patches, and rollbacks. The following subsections describe various testing activities that are commonly associated with installation testing.

23.3.1 Requirement Testing

Requirement Testing in installation testing ensures that all installation-related requirements are correctly implemented. These requirements may include supported operating systems, hardware prerequisites, disk space, memory requirements, dependencies, user permissions, and installation paths. This testing verifies that the installer enforces mandatory requirements and provides appropriate warnings or errors when requirements are not met. It ensures compliance with functional and non-functional installation specifications. Requirement testing helps prevent installation failures and reduces user confusion during deployment.

Requirement testing in the context of installation testing focuses on verifying whether the installation process meets the specified requirements and constraints. These requirements may include supported operating systems, hardware specifications, memory requirements, disk space, and prerequisite software. Requirement testing ensures that the installer correctly checks system compatibility before proceeding with the installation.

This type of testing also verifies that all installation-related requirements documented in the software specification are implemented correctly. For example, if the requirement states that the software must prompt the user when system requirements are not met, requirement testing ensures that such prompts appear correctly. By validating installation requirements, this testing helps prevent installation failures and ensures that users receive clear guidance during the setup process.

NOTE

23.3.2 Regression Testing

Regression Testing during installation ensures that new changes, updates, or patches do not adversely affect previously working installation functionality. Whenever an installer is modified, regression testing verifies that existing installation scenarios continue to function correctly. This testing covers fresh installations, upgrades from previous versions, repair installations, and uninstallation processes. Regression testing helps maintain installation stability across versions and ensures backward compatibility.

This testing is particularly important for software products that undergo frequent updates or patches. Regression testing verifies that upgrades from older versions work smoothly and that existing user settings and data are preserved. By detecting unintended side effects early, regression testing helps maintain consistency and reliability across software releases. Regression testing during installation testing ensures that new changes, updates, or fixes do not negatively impact existing installation functionality. Whenever a new version of software is released or modifications are made to the installer, there is a risk that previously working installation features may fail. Regression testing addresses this risk by re-executing installation test cases that were successful in earlier versions.

23.3.3 Error Handling Testing

Error Handling Testing evaluates how the installation process responds to unexpected situations such as missing files, insufficient permissions, corrupted installation media, network failures, or incompatible environments. This testing ensures that error messages are clear, informative, and user-friendly. It verifies that the system recovers gracefully without leaving the system in an inconsistent state. Proper error handling during installation improves usability and reduces technical support efforts.

Effective error handling testing checks whether the system avoids partial or corrupted installations and whether it can recover or roll back changes when errors occur. Clear and informative error messages improve user experience and reduce support costs. This type of testing is essential for building robust and user-friendly installation processes. Error handling testing evaluates how well the installation process responds to unexpected conditions and failures. These conditions may include insufficient disk space, missing dependencies, interrupted installation, invalid user inputs, or system crashes. The objective of this testing is to ensure that the installer handles errors gracefully and provides meaningful messages to the user.

NOTE

23.3.4 Manual Support Testing

Manual Support Testing evaluates the extent of manual intervention required during installation. It verifies whether installation steps are clearly documented and whether support personnel can assist users effectively. This testing ensures that installation manuals, help files, and support documentation are accurate, complete, and easy to understand. Manual support testing is essential for enterprise software where installations may require administrative or technical assistance.

Manual support testing focuses on verifying the availability and effectiveness of user support during the installation process. This includes testing installation guides, user manuals, help files, and on-screen instructions provided with the software. Manual support testing ensures that users can understand and follow installation steps without requiring extensive technical knowledge.

This testing also evaluates whether troubleshooting information is sufficient to resolve common installation issues. By validating documentation and support materials, manual support testing helps reduce user frustration and dependency on technical support teams, thereby enhancing the overall usability of the software.

23.3.5 Intersystem Testing

Intersystem Testing during installation verifies the interaction between the software being installed and other systems or applications already present in the environment. It ensures compatibility and coexistence with existing software, databases, middleware, and network services. This testing identifies conflicts related to shared resources, ports, libraries, or services. Intersystem testing is especially important for integrated enterprise systems and distributed applications.

Intersystem testing during installation testing examines how the software interacts with other systems and applications present in the target environment. Many software products depend on external systems such as databases, middleware, operating system services, or third-party applications. Intersystem testing ensures that these interactions function correctly after installation.

This testing verifies that the installation process correctly configures interfaces, communication protocols, and data exchange mechanisms between systems. It also checks for conflicts with existing software. Intersystem testing is crucial for enterprise and distributed systems where seamless integration is a key requirement.

23.3.6 Control Testing

Control Testing focuses on verifying administrative and control mechanisms associated with installation. It ensures that access

NOTE

controls, user permissions, licensing mechanisms, and configuration controls are correctly enforced during and after installation.

This testing validates that only authorized users can install, modify, or uninstall the software. Control testing contributes to system security and compliance with organizational policies.

Control testing focuses on verifying access controls and permissions associated with the installation process. It ensures that only authorized users can install, modify, or uninstall the software. Control testing checks whether the installer correctly enforces security policies related to user roles, administrative privileges, and system permissions.

This type of testing is important for protecting systems from unauthorized modifications and ensuring compliance with organizational security standards. Control testing helps maintain system integrity and prevents misuse of installation privileges.

23.3.7 Smoke Testing

Smoke Testing is a preliminary testing activity performed immediately after installation to verify that the software is stable enough for further testing or use. It checks basic functionality and confirms that critical components are operational. Smoke testing during installation helps detect major failures early, saving time and resources. If smoke tests fail, further testing is halted until issues are resolved.

Smoke testing in installation testing is a preliminary testing activity performed to verify that the installation process is stable enough for further testing. It involves executing a basic set of tests to confirm that the software installs successfully and launches without major issues. Smoke testing acts as a quick validation of installation readiness.

If smoke testing fails, it indicates serious problems that must be resolved before proceeding with detailed testing. This approach saves time and effort by identifying critical issues early in the testing process.

Activity Title: Testing Software Installation

Objective:
To understand and evaluate the installation process of a software application.

Materials Required:

- Computer/mobile device
- Software installer or application
- Installation checklist

Procedure/Steps:

1. Select a safe and legitimate software application.

NOTE

2. Review its installation requirements.
3. Install the application following the provided instructions.
4. Observe installation steps, errors, permissions, and configuration.
5. Verify whether the application launches successfully after installation.
6. Record any installation-related issues.

Outcome:
The learner will understand the purpose of installation testing and identify common installation problems.

23.3.8 Ad Hoc Testing

Ad Hoc Testing is an informal testing approach conducted without predefined test cases. During installation testing, ad hoc testing involves exploring installation scenarios based on tester experience and intuition. This testing helps identify unusual or unexpected installation issues that structured testing might miss. Ad hoc testing complements formal installation testing techniques.

Adhoc testing is an informal and exploratory approach applied during installation testing without predefined test cases. Testers rely on their experience and intuition to identify potential installation issues by trying unusual or unexpected scenarios. This may include interrupting the installation process, changing system settings during installation, or installing the software under uncommon configurations.

Although adhoc testing lacks formal structure, it is effective in uncovering hidden defects that structured testing may overlook. It complements other installation testing activities by providing broader coverage of real-world usage conditions.

23.3.9 Parallel Testing

Parallel Testing during installation involves comparing the behavior of the new software version with an existing version under similar installation and operational conditions. Both versions are installed and tested simultaneously. This testing ensures consistency, validates improvements, and helps detect deviations introduced in newer versions. Parallel testing is commonly used during migrations and system upgrades.

Parallel testing involves installing and running the new software alongside an existing system to compare their behavior and outputs. In installation testing, parallel testing ensures that the new version can coexist with the old version without conflicts. This is especially important during system upgrades or migrations.

NOTE

Parallel testing helps verify data consistency, functional equivalence, and system stability during transition phases. It reduces the risk associated with replacing existing systems and builds confidence in the new installation before full deployment.

Check Your Progress

Notes:
(a) Space is given below for your answer.
(b) Check your answer with the one given at the end of this unit.

Fill-in-the-Blank Questions:

1. _____ testing verifies that software installs correctly in the target environment.
2. _____ testing ensures new installation changes do not break existing functionality.
3. Error handling testing checks system response to _____ situations.

True/False Questions:

1. Installation testing is unnecessary if system testing is complete. _____
2. Smoke testing is performed after installation. _____

Q1. Installation testing mainly ensures that:
A) Software is written in correct language
B) Software installs correctly in the target environment
C) Only GUI is attractive
D) Only coding is completed

_____

Q2. Smoke testing is a basic test performed to check whether the build is stable enough for further testing.

_____

Q3. Testing performed to ensure that new changes have not affected existing features is called _____ testing.

_____

Q4. Write any two activities covered under installation testing.

Answer (Any two):

1. Checking installation steps and setup wizard
2. Verifying system requirements and dependencies
3. Testing upgrade/uninstall process

NOTE

4. Checking configuration settings after installation

5. Validating installation on different OS environments

Q5. Adhoc testing is best described as:
A) Testing with a fixed set of documented test cases
B) Informal testing without detailed planning
C) Testing only for security
D) Testing only during design phase

23.4 SUMMARY

This unit highlights the importance of installation testing as a vital component of Software Quality Assurance and Testing. Installation testing ensures that software can be correctly deployed, configured, and integrated into its target environment. The unit explains various testing activities associated with installation testing, including requirement testing, regression testing, error handling testing, manual support testing, intersystem testing, control testing, smoke testing, adhoc testing, and parallel testing. Together, these tests address technical, usability, security, and integration aspects of software installation.

By systematically validating the installation process, organizations can reduce deployment risks, improve user satisfaction, and enhance software reliability. Installation testing ensures that the first interaction between the user and the software is smooth and trouble-free, thereby contributing significantly to the overall success of the software product.

Installation testing ensures that software can be successfully deployed and operated in real-world environments. It validates installation processes, error handling, compatibility, and system readiness. By performing requirement, regression, error handling, manual support, intersystem, control, smoke, ad hoc, and parallel testing, organizations can deliver reliable and user-friendly software products.

23.5 GLOSSARY

- Installation Testing: Testing the process of installing and configuring software.
- Regression Testing: Testing to ensure changes do not affect existing functionality.
- Smoke Testing: Basic testing to verify system stability.
- Intersystem Testing: Testing interactions between multiple systems.
- Control Testing: Testing access and control mechanisms.

NOTE

23.6 ANSWERS TO CHECK YOUR PROGRESS – POSSIBLE ONE

Fill-in-the-Blank:

Installation

Regression

unexpected

True/False:

False

True

Q1. Answer: B

Explanation: Installation testing checks whether the software installs, configures, and runs correctly in the user environment.

Q2. Answer: True

Explanation: Smoke testing verifies basic functionality and ensures the build is not broken before detailed testing starts.

Q3. Answer: Regression

Explanation: Regression testing confirms that bug fixes or new features do not introduce new defects into existing functionality.

Q4. Explanation: Installation testing ensures software works correctly when installed, upgraded, or removed in different environments.

Q5. Answer: B

Explanation: Adhoc testing is an informal method where testers explore the application freely to find defects without strict test cases.

23.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3802	

NOTE

23.8 REFERENCE

- IEEE Std 829 – Software Test Documentation.
- ISO/IEC 25010 – Software Quality Models.
- Myers, G. J. The Art of Software Testing.

23.9 EXERCISE

Part–1

A. Multiple Choice Questions

Q1. Installation testing checks:

A) GUI color

B) Correct installation and configuration

C) Only code quality

D) Only performance

Q2. Installation testing is performed:

A) Before development

B) Before release

C) After retirement

D) Never

Q3. A common installation defect is:

A) Wrong font

B) Missing dependencies

C) Wrong login password

D) Typing error

Q4. Uninstallation testing checks:

A) GUI speed

B) Proper removal of software

C) Database queries

D) Requirement quality

Q5. Installation testing is important for:

A) Customer satisfaction

B) Code formatting

C) Increasing size

D) Reducing requirements

Answers with Explanations

1. Answer: B

   Explanation: It ensures software installs correctly.

2. Answer: B

   It is done before product delivery.

3. Answer: B

   Missing libraries cause install failure.

4. Answer: B

   It ensures clean uninstall.

5. Answer: A

   Smooth installation improves user experience.

B. Small Answer Questions

1. Define installation testing.
2. What is uninstallation testing?
3. Write two objectives of installation testing.
4. Write two installation defects.
5. Why is installation testing important?

Part–2

C. Long Answer Type Questions (200 words)

1. Explain installation testing and its scope.
2. Discuss installation testing steps and checklist.
3. Explain uninstallation testing with examples.
4. Discuss installation defects and their impact.
5. Explain installation testing in real environments.

Part–3

D. Long Answer Type Questions (200 words)

1. Explain installation testing in detail with examples.
2. Discuss installation testing for different operating systems.
3. Explain installation and configuration testing challenges.

NOTE

4. Discuss installation testing tools and best practices.
5. Explain how installation testing improves product quality.

RNTU Rabindranath TAGORE UNIVERSITY

Approved by : AICTE, NCTE, BCI, INC, MP PARAMEDICAL COUNCIL, DEB (UGC) | Recognized by : UGC | Member of : AIU, ACU

RABINDRANATH TAGORE UNIVERSITY (M.P.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org / for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

_____

_____

_____

***

NOTE

BLOCK 6	TYPES OF TESTING, OBJECT ORIENTED APPLICATION TESTING	389-428
Unit 24	TYPES OF TESTING	391
Unit 25	State Graph, Risk Associated with New Technologies, Process Maturity Level of Technology, Testing Adequacy of Control in New Technology Usage	403
Unit 26	OBJECT ORIENTED APPLICATION TESTING	415



This block introduces learners to advanced concepts and techniques used for testing different types of software applications. It begins with an exploration of various types of testing, their purposes, applications, and appropriate usage during the software development life cycle. The block then focuses on state graph-based testing, risks associated with adopting new technologies, and the role of technology process maturity levels in ensuring reliable and controlled software development. It also explains methods for assessing the adequacy of controls when implementing new technologies. Finally, the block discusses Object-Oriented Application Testing, highlighting the unique testing challenges associated with classes, objects, inheritance, polymorphism, encapsulation, and interactions among objects. Overall, this block helps learners develop a systematic understanding of advanced testing approaches and apply suitable testing techniques to improve software quality, reliability, and performance.

NOTE

NOTE

UNIT–24
TYPES OF TESTING

UNIT STRUCTURE

24.1 Introduction
24.2 Objective
24.3 Types of Testing
- 24.3.1 Execution Testing
- 24.3.2 Operations Testing
- 24.3.3 Compliance Testing
- 24.3.4 Usability Testing
- 24.3.5 Decision Table Testing
- 24.3.6 Documentation Testing
- 24.3.7 Training testing
- 24.3.8 Rapid Testing
- 24.3.9 Control flow graph
- 24.3.10 Generating tests on the basis of Combinatorial Designs

24.4 Summary
24.5 Glossary
24.6 Answer to check your progress
24.7 Suggested readings - OER
24.8 Reference
24.9 Exercise

24.1 INTRODUCTION

Software Quality Assurance and Testing (SQAT) encompasses a wide range of testing techniques aimed at ensuring that software systems meet specified requirements and perform reliably in real-world environments. As software systems grow in complexity, it becomes necessary to apply different types of testing to evaluate both functional and non-functional aspects. No single testing technique is sufficient to uncover all defects; therefore, multiple types of testing are employed throughout the software development life cycle.

Types of Testing provide a structured classification of testing activities based on objectives, scope, execution approach, and quality attributes being evaluated. These testing types help testers systematically identify defects related to execution behavior, operational readiness, regulatory

NOTE

compliance, usability, logic, documentation, training adequacy, and system control mechanisms.

Understanding different types of testing enables software engineers and testers to select appropriate techniques at various stages of development. It also helps organizations improve software reliability, user satisfaction, and compliance with standards. This unit discusses important types of testing, including execution testing, operations testing, compliance testing, usability testing, decision table testing, documentation testing, training testing, rapid testing, control flow graph–based testing, and combinatorial design–based test generation.

Software testing is a systematic activity that aims to evaluate and improve the quality of software products. As software systems grow in size, complexity, and usage, testing has evolved into a broad discipline encompassing various techniques and approaches. Different types of testing are required to address different quality attributes such as correctness, reliability, usability, compliance, and maintainability. No single testing method is sufficient to ensure complete software quality; instead, multiple types of testing are applied at different stages of the software development life cycle.

This unit introduces several important types of testing that are commonly used in Software Quality Assurance. These testing types focus on both functional and non-functional aspects of software and help ensure that the system behaves as expected under real-world conditions. By understanding these testing approaches, learners can appreciate how systematic testing reduces risks, improves user confidence, and ensures that software meets organizational and regulatory requirements.

24.2 OBJECTIVE

After studying this unit, you will be able to:

- Explain execution and operations testing in ensuring software readiness.
- Describe compliance testing and its importance in meeting standards and regulations.
- Explain decision table testing as a systematic test design technique.
- Describe documentation and training testing for effective software usage.
- Analyse rapid testing approaches in fast-paced development environments.

NOTE

- Explain control flow graph concepts in test case design.

24.3 TYPES OF TESTING

Types of testing refer to different testing approaches used to evaluate specific characteristics of software systems. Each type addresses particular risks and quality concerns and collectively contributes to delivering a robust and dependable software product.

It refer to various testing approaches designed to validate specific aspects of a software system. Each type of testing focuses on a particular quality characteristic or operational concern. These testing techniques may be applied individually or in combination, depending on project requirements, system complexity, and risk factors. The following subsections describe important types of testing commonly used in software quality assurance.

24.3.1 Execution Testing

Execution Testing focuses on evaluating the behavior of software by executing it under controlled conditions. The primary objective of execution testing is to verify that the software performs its intended functions correctly when run with valid and invalid inputs.

This type of testing observes actual system behavior, output results, error messages, and system responses during execution. Execution testing helps identify runtime errors, logical faults, and unexpected system behavior. It includes activities such as functional testing, system testing, and acceptance testing.

Execution testing is essential because many defects only surface when the software is executed. Static reviews alone cannot reveal issues related to memory usage, performance bottlenecks, or incorrect runtime interactions.

Execution testing focuses on verifying the behavior of software by executing it with valid and invalid inputs. The objective of execution testing is to confirm that the software performs its intended functions correctly when it is run in a real or simulated environment. This type of testing examines actual program execution rather than theoretical analysis.

Execution testing helps identify runtime errors, incorrect logic, and unexpected behavior that may not be detected through static analysis. It ensures that the software produces correct outputs, handles exceptions properly, and interacts correctly with system resources. Execution testing is fundamental to software validation, as it provides direct evidence of how the system behaves during operation.

NOTE

24.3.2 Operations Testing

Operations Testing evaluates the readiness of software for operation in a production environment. It focuses on operational aspects such as system startup, shutdown, backup, recovery, monitoring, and maintenance procedures. This testing ensures that system administrators and operators can manage the software effectively. Operations testing verifies operational documentation, scripts, scheduling jobs, and system logs. It helps identify issues that may arise during daily operations, ensuring smooth and reliable system functioning.

Operations testing evaluates the readiness of software for deployment and use in an operational environment. It focuses on verifying whether operational procedures such as system startup, shutdown, backup, recovery, and maintenance are correctly implemented and documented. Operations testing ensures that the software can be managed efficiently by system administrators and support teams.

This type of testing also examines system behavior during routine operations and abnormal conditions. By validating operational aspects, operations testing helps ensure system stability, availability, and maintainability in real-world usage scenarios.

24.3.3 Compliance Testing

Compliance Testing verifies whether the software conforms to applicable standards, regulations, legal requirements, and organizational policies. These standards may include industry standards, government regulations, security policies, and quality frameworks.

Compliance testing is particularly important in domains such as healthcare, finance, defense, and education. It helps organizations avoid legal penalties, ensure data protection, and maintain trust with stakeholders. Compliance testing checks whether the software adheres to specified standards, regulations, policies, and legal requirements. These may include industry standards, organizational guidelines, government regulations, or contractual obligations. Compliance testing is especially important in domains such as finance, healthcare, and aviation, where regulatory compliance is mandatory.

This type of testing ensures that software development and implementation follow prescribed rules and best practices. Compliance testing reduces legal risks, ensures accountability, and enhances trust among stakeholders by demonstrating that the software meets required standards.

NOTE

24.3.4 Usability Testing

Usability Testing focuses on evaluating the ease of use, learnability, efficiency, and user satisfaction of a software application. It examines how effectively users can interact with the system to achieve their goals. This testing identifies usability issues such as confusing navigation, unclear instructions, poor layout, and accessibility barriers. Usability testing is often performed with real users and helps improve user experience and acceptance.

Usability testing focuses on evaluating how easy and efficient it is for users to interact with the software. The objective of usability testing is to ensure that the system is user-friendly, intuitive, and aligned with user expectations. It examines aspects such as navigation, layout, clarity of instructions, and ease of learning.

Usability testing often involves observing real users as they perform tasks using the software. Feedback obtained from usability testing helps identify design flaws and areas for improvement. By improving usability, organizations can enhance user satisfaction, reduce errors, and increase acceptance of the software product.

24.3.5 Decision Table Testing

Decision Table Testing is a systematic test design technique used to test complex business logic involving multiple conditions and actions. It represents conditions and corresponding actions in a tabular format. This testing ensures that all possible combinations of conditions are tested. Decision table testing improves test coverage and helps detect missing or incorrect logic in decision-making processes.

Decision table testing is a systematic testing technique used to validate complex business logic and decision-making rules. It represents conditions and corresponding actions in a tabular format, making it easier to identify all possible combinations of inputs and expected outcomes.

This type of testing is particularly useful when software behavior depends on multiple conditions. Decision table testing helps ensure complete coverage of logical rules and reduces the risk of missing important test cases. It improves clarity and consistency in testing complex decision logic.

24.3.6 Documentation Testing

Documentation Testing verifies the accuracy, completeness, and usability of software documentation. This includes user manuals, installation guides, online help, and technical documentation.

NOTE

Accurate documentation is essential for effective software usage and maintenance. Documentation testing ensures that instructions match actual system behavior and are easy to understand.

Documentation testing focuses on verifying the accuracy, completeness, and usability of software documentation. This includes user manuals, installation guides, help files, and technical documentation. Documentation testing ensures that written materials correctly describe software functionality and usage.

Accurate documentation is essential for effective software use, maintenance, and support. Documentation testing helps prevent misunderstandings, reduces training effort, and improves overall user experience. It also ensures consistency between the software and its supporting documents.

24.3.7 Training Testing

Training Testing evaluates the effectiveness of training materials and programs provided to users, operators, and support staff. It ensures that training content adequately prepares users to operate the software. This testing verifies training manuals, tutorials, presentations, and hands-on exercises. Effective training reduces errors, support requests, and resistance to system adoption.

Training testing evaluates the effectiveness of training materials and programs provided to users, administrators, or support staff. The objective of training testing is to ensure that training resources adequately prepare users to operate and manage the software.

This type of testing assesses whether training content is clear, relevant, and aligned with actual system functionality. Effective training testing helps reduce operational errors, increases productivity, and ensures smooth adoption of the software.

24.3.8 Rapid Testing

Rapid Testing is an approach that emphasizes speed and adaptability in testing activities. It is commonly used in agile and continuous delivery environments where quick feedback is essential.

Rapid testing relies on tester expertise, automation, and exploratory techniques. It helps identify critical issues quickly and supports fast-paced software releases.

Rapid testing is an approach that emphasizes speed and efficiency in testing activities. It is often used in agile and iterative development environments where software is developed and delivered in short cycles. Rapid testing focuses on quick feedback and continuous improvement.

This testing approach relies on automation, exploratory testing, and close collaboration between testers and developers. Rapid testing helps

identify defects early and supports faster delivery of high-quality software without compromising essential testing objectives.

24.3.9 Control Flow Graph

A Control Flow Graph (CFG) is a graphical representation of all possible execution paths in a program. Nodes represent program statements or blocks, and edges represent control flow between them.

Control flow graph–based testing helps design test cases to achieve coverage criteria such as statement coverage, branch coverage, and path coverage. CFGs are widely used in white-box testing techniques.

A control flow graph is a graphical representation of the logical flow of a program. It illustrates the sequence of execution paths through program statements and decision points. Control flow graphs are used as a basis for structural testing techniques.

By analyzing control flow graphs, testers can design test cases that ensure coverage of different execution paths. This helps identify unreachable code, logical errors, and incomplete test coverage. Control flow graph-based testing improves program reliability and code quality.

24.3.10 Generating Tests on the Basis of Combinatorial Designs

Combinatorial Design–based testing generates test cases by systematically combining input parameters and values. The objective is to detect faults caused by interactions among multiple input factors. This approach reduces the number of test cases while maintaining high defect detection effectiveness. Techniques such as pairwise testing are commonly used in combinatorial testing.

Generating tests based on combinatorial designs is a structured testing approach that focuses on covering combinations of input parameters efficiently. Instead of testing all possible combinations, which may be impractical, this technique selects a representative set of combinations that are most likely to reveal defects.

Combinatorial testing is particularly effective for systems with multiple configuration options and parameters. It helps achieve high coverage with a reduced number of test cases, making testing more efficient and cost-effective. This approach is widely used in configuration testing and complex system validation.

Activity Title: Selecting the Appropriate Testing Type

Objective:
To identify different types of software testing and determine their suitability for specific situations.

Materials Required:

- Testing case studies

NOTE

- Notebook
- Pen

Procedure/Steps:

1. Prepare five different software scenarios.
2. Identify the quality concern in each scenario.
3. Select an appropriate testing type for each situation.
4. Give reasons for selecting the particular testing type.
5. Discuss the possible consequences of not performing the selected test.

Outcome:
The learner will be able to identify appropriate testing types based on software requirements and quality concerns.

Check Your Progress 24.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Fill-in-the-Blank Questions:

_____ testing focuses on software behavior during execution.

Usability testing evaluates user _____ and ease of use.

A control flow graph represents program _____ paths.

True/False Questions:

Compliance testing is important only for legal purposes.

Decision table testing is useful for complex logic.

Q1. Execution testing mainly refers to:

A) Testing by reading documents only
B) Testing by running the software and checking output
C) Testing hardware devices
D) Testing only database design

_____

Q2. Usability testing checks how easy and user-friendly the software is.

_____

Q3. Testing done to ensure software follows laws, standards, and regulations is called _____ testing.

_____

Q4. Write any two types of testing covered in this unit.

Answer (Any two):

1. Operations testing

2. Compliance testing
3. Usability testing
4. Documentation testing
5. Rapid testing
6. Training testing

Q5. Combinatorial design-based testing is mainly used to:

A) Test only one input at a time
B) Generate test cases based on combinations of multiple input parameters
C) Replace installation testing
D) Avoid regression testing

24.4 SUMMARY

Different types of testing address diverse quality aspects of software systems. Execution, operations, compliance, usability, decision table, documentation, training, rapid, control flow–based, and combinatorial testing collectively ensure software reliability, usability, compliance, and maintainability. Selecting appropriate testing types at the right stages is essential for delivering high-quality software.

This unit presents an overview of various types of testing used in Software Quality Assurance and Testing. It explains how different testing approaches address specific quality attributes such as correctness, usability, compliance, and operational readiness. The unit covers execution testing, operations testing, compliance testing, usability testing, decision table testing, documentation testing, training testing, rapid testing, control flow graph analysis, and combinatorial design-based testing.

Together, these testing types provide a comprehensive framework for evaluating software quality from multiple perspectives. By applying appropriate testing techniques at different stages of development, organizations can reduce risks, improve software reliability, and deliver systems that meet user expectations and regulatory requirements.

24.5 GLOSSARY

- Execution Testing: Testing based on actual execution of software.
- Compliance Testing: Testing against standards and regulations.
- Usability Testing: Testing ease of use and user experience.

NOTE

- Control Flow Graph: Graphical representation of program execution paths.
- Combinatorial Testing: Testing based on combinations of input parameters.

24.6 ANSWERS TO CHECK YOUR PROGRESS – POSSIBLE ONE

Fill-in-the-Blank:

Execution

satisfaction

execution

True/False:

False

True

Q1. Answer: B

Explanation: Execution testing is dynamic testing where the program is executed and results are compared with expected output.

Q2. Answer: True

Explanation: Usability testing focuses on user experience, ease of learning, navigation, and overall satisfaction.

Q3. Answer: Compliance

Explanation: Compliance testing verifies whether the system meets required standards such as ISO, legal rules, and organizational policies.

Q4. Explanation: These testing types ensure software quality not only in functionality but also in usability, compliance, and operational readiness.

Q5. Answer: B

Explanation: Combinatorial testing reduces test cases while ensuring good coverage of parameter combinations, especially for complex systems.

24.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/User/UnitUrls?unitId=3803	

NOTE

24.8 REFERENCE

- IEEE Std 829 – Software Test Documentation.
- ISO/IEC 25010 – Software Quality Models.
- Myers, G. J. The Art of Software Testing.

24.9 EXERCISE

Part–1

A. Multiple Choice Questions

Q1. Acceptance testing is performed to:
- A) Check coding style
- B) Validate software against user requirements
- C) Increase complexity
- D) Remove test cases

Q2. Acceptance testing is usually done by:
- A) Developers only
- B) End users/customers
- C) Compiler
- D) Database admin

Q3. Alpha testing is conducted at:
- A) Customer site
- B) Developer site
- C) Production only
- D) Market only

Q4. Beta testing is conducted at:
- A) Developer lab
- B) Customer environment
- C) Requirement phase
- D) Design phase

Q5. UAT stands for:
- A) Unified Application Testing
- B) User Acceptance Testing
- C) Universal Automation Tool
- D) Unit Analysis Test

Answers with Explanation:

NOTE

1. Answer: B
   Explanation: It ensures product meets customer needs.
2. Answer: B
   Explanation: Customers verify readiness.
3. Answer: B
   Explanation: Alpha is internal testing.
4. Answer: B
   Explanation: Beta is external user testing.
5. Answer: B
   Explanation: UAT validates business requirements.

B. Small Answer Questions

1. Define acceptance testing.
2. Differentiate alpha and beta testing.
3. What is UAT?
4. Write two objectives of acceptance testing.
5. Why is acceptance testing important?

Part–2

C. Long Answer Type Questions (200 words)

1. Explain acceptance testing and its purpose.
2. Discuss alpha testing and beta testing.
3. Explain UAT process and test case preparation.
4. Discuss acceptance criteria and exit criteria.
5. Explain challenges in acceptance testing.

Part–3

D. Long Answer Type Questions (300 words)

1. Explain acceptance testing in detail with examples.
2. Discuss UAT planning, execution, and reporting.
3. Explain acceptance criteria and how they are defined.
4. Discuss alpha and beta testing in real projects.
5. Explain how acceptance testing ensures customer satisfaction.

NOTE

UNIT–25
NEW TECHNOLOGY RISK AND CONTROL TESTING

UNIT STRUCTURE

25.1 Introduction
25.2 Objective
25.3 State Graph, Risk Associated with New Technologies, Process maturity level of Technology, Testing Adequacy of Control in New technology usage
  25.3.1 State Graph
  25.3.2 Risk Associated with New Technologies
  25.3.3 Process maturity level of Technology
  25.3.4 Testing Adequacy of Control in New technology usage
25.4 Summary
25.5 Glossary
25.6 Answer to check your progress
25.7 Suggested readings - OER
25.8 Reference
25.9 Exercise

25.1 INTRODUCTION

The apid evolution of software technologies has significantly transformed the way systems are designed, developed, deployed, and maintained. Modern software systems increasingly rely on new and emerging technologies such as cloud computing, artificial intelligence, machine learning, Internet of Things (IoT), blockchain, and mobile platforms. While these technologies offer numerous advantages in terms of efficiency, scalability, and innovation, they also introduce new complexities and risks that must be carefully managed through systematic testing and quality assurance practices.

In Software Quality Assurance and Testing (SQAT), understanding system behavior, technological risks, and maturity levels is essential for ensuring reliable and secure software systems. Concepts such as state graphs help testers model and analyze system behavior under different conditions. At the same time, assessing risks associated with new technologies enables organizations to anticipate failures, security

NOTE

threats, and operational challenges. The maturity level of technology processes provides insights into how well an organization can manage and control technological change. Furthermore, testing the adequacy of controls ensures that safeguards are effective when new technologies are adopted.

This unit focuses on four important and interrelated aspects: state graphs, risks associated with new technologies, process maturity levels of technology, and testing the adequacy of control in new technology usage. Together, these topics provide a comprehensive understanding of how advanced testing techniques and management practices contribute to successful adoption of new technologies.

The rapid evolution of software technologies has significantly increased the complexity of software systems and the challenges associated with ensuring their quality. New technologies such as cloud computing, mobile platforms, artificial intelligence, distributed systems, and emerging development frameworks introduce innovative capabilities but also bring new risks and uncertainties. In Software Quality Assurance and Testing, it is essential to understand how software behavior changes across different states, how risks arise from adopting new technologies, and how process maturity influences the effectiveness of testing activities. This unit focuses on important concepts that help testers and quality professionals manage complexity and uncertainty in modern software systems. It introduces the concept of state graphs as a technique for modeling and analyzing system behavior, discusses the risks associated with new technologies, explains the significance of process maturity levels, and examines the adequacy of testing controls in the usage of new technologies. Together, these topics provide a structured approach to assessing and improving software quality in technologically dynamic environments.

25.2 OBJECTIVE

After studying this unit, you will be able to:

- Analyse the concept and significance of state graphs in software testing.
- Explain how state graphs are used to model system behavior and design test cases.
- Identify and analyze risks associated with the adoption of new technologies.
- Understand the concept of process maturity levels in technology management.

NOTE

- Explain different maturity models used to assess technological processes.
- Understand the importance of testing adequacy of controls when using new technologies.
- Ensure reliability, security, and compliance in systems built using emerging technologies.

25.3 STATE GRAPH, RISK ASSOCIATED WITH NEW TECHNOLOGIES, PROCESS MATURITY LEVEL OF TECHNOLOGY, TESTING ADEQUACY OF CONTROL IN NEW TECHNOLOGY USAGE

The adoption of new technologies requires both technical and managerial preparedness. Software testing plays a crucial role in validating system behavior, mitigating risks, and ensuring adequate control mechanisms. The following subsections discuss key concepts relevant to testing and managing new technology-based systems.

Modern software systems often exhibit complex behavior that depends on their current state, inputs, and interactions with external systems. At the same time, the adoption of new technologies introduces risks related to uncertainty, lack of experience, and evolving standards. To manage these challenges, Software Quality Assurance relies on techniques such as state-based modeling, risk assessment, process maturity evaluation, and control adequacy testing. These approaches help organizations understand system behavior, identify potential risks, and ensure that appropriate testing mechanisms are in place.

25.3.1 State Graph

A State Graph is a graphical representation used to model the dynamic behavior of a system by illustrating its various states and the transitions between those states. Each state represents a specific condition or mode of operation of the system, while transitions represent events or inputs that cause the system to move from one state to another.

State graphs are widely used in software testing, particularly in state-based and model-based testing techniques. They help testers understand system behavior, identify valid and invalid state transitions, and design comprehensive test cases. State graph–based testing ensures that all possible states and transitions are adequately tested, thereby improving test coverage and defect detection.

State graphs are especially useful for testing reactive systems, embedded systems, user interfaces, and protocol-based applications where system behavior depends on previous actions or events. A state

NOTE

graph is a graphical representation of the various states of a software system and the transitions between those states based on events or conditions. Each state represents a specific situation or mode of operation of the system, while transitions represent changes triggered by inputs or actions. State graphs are particularly useful for modeling systems with dynamic behavior, such as communication protocols, embedded systems, user interfaces, and workflow-based applications.

In software testing, state graphs help testers visualize system behavior and identify valid and invalid state transitions. By analyzing a state graph, testers can design test cases that ensure all states and transitions are adequately tested. This approach helps detect errors such as missing transitions, incorrect state changes, or unexpected system behavior. State graph-based testing improves coverage and reliability by ensuring that the system behaves correctly across all possible states.

25.3.2 Risk Associated with New Technologies

The adoption of new technologies introduces various risks that can affect software quality, security, performance, and reliability. These risks arise due to limited experience, lack of standards, evolving tools, and integration challenges.

Common risks associated with new technologies include technical risks, security vulnerabilities, operational risks, compliance risks, and skill-related risks. For example, new technologies may lack mature security mechanisms, leading to potential data breaches. Integration with legacy systems may introduce compatibility issues, while insufficient expertise can result in design and implementation errors.

Risk assessment and mitigation are essential components of software testing in new technology environments. Testing helps identify potential failures early and reduces the impact of risks through preventive and corrective measures.

The adoption of new technologies introduces various risks that can affect software quality. These risks may arise due to limited knowledge, immature tools, lack of standardized practices, or unproven architectures. New technologies often evolve rapidly, leading to frequent changes in frameworks, platforms, and best practices. This uncertainty increases the likelihood of defects, integration issues, and performance problems.

From a testing perspective, risks associated with new technologies include inadequate test coverage, insufficient expertise among testing teams, and difficulty in predicting system behavior under real-world conditions. Security risks may also increase due to newly discovered vulnerabilities or incomplete understanding of threat models.

NOTE

Identifying and assessing these risks early allows organizations to prioritize testing efforts, allocate resources effectively, and implement mitigation strategies to reduce potential failures.

25.3.3 Process Maturity Level of Technology

The Process Maturity Level of Technology refers to the degree to which an organization's technology-related processes are defined, managed, measured, and continuously improved. Maturity models provide a structured framework for evaluating and improving process capability.

Higher maturity levels indicate more predictable, controlled, and optimized processes, leading to better quality outcomes. Well-known maturity models such as the Capability Maturity Model Integration (CMMI) help organizations assess their readiness to adopt and manage new technologies.

Understanding process maturity levels helps organizations identify gaps, improve testing practices, and ensure consistent quality in technology-driven projects.

Process maturity refers to the degree to which an organization's software development and testing processes are defined, managed, measured, and optimized. A higher level of process maturity indicates well-established practices, clear documentation, consistent execution, and continuous improvement. In contrast, low process maturity is characterized by ad hoc activities, inconsistent results, and dependence on individual expertise.

In the context of technology usage, process maturity plays a critical role in managing complexity and risk. Mature processes enable organizations to adopt new technologies more effectively by providing structured guidelines for development, testing, and quality assurance. Process maturity models help organizations assess their current capabilities and identify areas for improvement. As process maturity increases, testing becomes more systematic, predictable, and effective, leading to higher software quality.

25.3.4 Testing Adequacy of Control in New Technology Usage

Testing the adequacy of control in new technology usage involves verifying whether appropriate safeguards, policies, and mechanisms are in place to manage risks and ensure secure and reliable operation. Controls may include access controls, audit mechanisms, error handling, monitoring systems, and compliance checks.

Adequacy testing ensures that controls are correctly implemented, effective, and capable of preventing or detecting failures and misuse.

NOTE

This type of testing is particularly important in environments involving sensitive data, critical infrastructure, and regulatory compliance.

By validating control mechanisms, organizations can ensure that new technologies are used responsibly and safely without compromising system integrity or user trust.

Testing adequacy of control refers to the evaluation of whether appropriate testing controls and mechanisms are in place when using new technologies. Controls include policies, procedures, standards, tools, and review mechanisms that ensure testing activities are thorough and reliable. When new technologies are introduced, existing controls may no longer be sufficient or applicable, creating gaps in quality assurance.

Assessing testing adequacy involves examining whether test strategies cover new risks, whether testers are adequately trained, and whether tools and environments support the new technology. It also includes verifying that security, performance, and compliance requirements are addressed through suitable testing controls. Adequate testing controls help organizations balance innovation with quality, ensuring that the adoption of new technologies does not compromise system reliability or user trust.

Activity Title: Analysing Risks and Testing Controls in New Technology

Objective:
To analyse technology-related risks and determine suitable testing controls using a state graph and risk assessment approach.

Materials Required:

- Notebook
- Pen
- Case study of a new technology or application

Procedure/Steps:

1. Select a new technology-based application, such as a mobile payment, cloud, IoT, or AI-based system.
2. Identify its major operational states and draw a simple state graph.
3. Identify possible risks associated with the technology.
4. Assess the maturity of the technology as low, medium, or high based on available evidence.
5. Identify existing testing and control measures.
6. Suggest additional testing controls for high-risk areas.
7. Prepare a short risk-control matrix.

NOTE

Outcome:
The learner will be able to represent system states, identify technology-related risks, assess maturity, and recommend appropriate testing controls.

Check Your Progress 25.1

Notes: a) Space is given below for your answer.

b) Check your answer with the one given at the end of this unit.

Fill-in-the-Blank Questions:

A _____ graph represents system states and transitions.

Risks associated with new technologies may arise due to lack of _____.

Process maturity models help assess organizational _____.

True/False Questions:

State graphs are useful only for documentation purposes.

Higher process maturity leads to better quality outcomes.

Q1. A state graph is mainly used to represent:

A) Database tables
B) System behavior through states and transitions
C) User salary structure
D) Coding standards

Q2. Risk associated with new technologies can increase testing effort and uncertainty.

Q3. The maturity level of a technology indicates how _____ and stable the technology is.

Q4. Write any two risks associated with using new technologies in software projects.

Answer (Any two):

1. Lack of skilled resources and expertise
2. Higher defect probability due to immature tools
3. Integration issues with existing systems
4. Higher maintenance cost
5. Lack of documentation and community support

Q5. Testing adequacy of control mainly ensures that:

A) Only developers control the project

B) Proper controls exist to reduce failures and ensure safe operation

C) Software is free to download

D) Only GUI is tested

NOTE

25.4 SUMMARY

This unit emphasized the importance of understanding system behavior, technological risks, and process maturity in the context of new technologies. State graphs provide a powerful tool for modeling and testing system behavior, while risk analysis helps anticipate and mitigate challenges associated with emerging technologies. Process maturity levels offer insights into organizational readiness, and testing the adequacy of controls ensures secure and reliable technology usage. Together, these concepts strengthen software quality assurance practices in modern technology-driven environments.

This unit emphasizes the importance of understanding system behavior, technological risk, and process maturity in Software Quality Assurance and Testing. It introduces state graphs as an effective technique for modeling and testing state-dependent system behavior. The unit also discusses the risks associated with adopting new technologies and highlights the need for systematic risk assessment and mitigation through focused testing.

Furthermore, the unit explains the concept of process maturity and its impact on the effectiveness and consistency of testing practices. It underscores the importance of evaluating the adequacy of testing controls when new technologies are used. Together, these concepts provide a comprehensive framework for managing quality in modern software systems, enabling organizations to adopt new technologies confidently while maintaining high standards of reliability and performance.

25.5 GLOSSARY

- State Graph: A graphical model representing system states and transitions.
- Technology Risk: Potential issues arising from adopting new or emerging technologies.
- Process Maturity: The level of capability and control of organizational processes.
- Control Adequacy: The effectiveness of safeguards in managing risks.
- New Technology: Recently developed or rapidly evolving technological solutions.

NOTE

25.6 ANSWERS TO CHECK YOUR PROGRESS – POSSIBLE ONE

Fill-in-the-Blank:

State

experience

capability

True/False:

False

True

Q1. Answer: B

Explanation: A state graph shows different states of a system and how it moves from one state to another based on events or conditions.

Q2. Answer: True

Explanation: New technologies may have unknown defects, limited expertise, and unpredictable behavior, increasing risk and testing needs.

Q3. Answer: Reliable

Explanation: Mature technologies are more reliable, stable, and well-supported compared to newly introduced technologies.

Q4. Explanation: New technologies often introduce uncertainty, which increases project risk and testing complexity.

Q5. Answer: B

Explanation: Adequacy of control testing ensures that the system has proper control mechanisms (validation, security, error handling) to reduce operational risk.

25.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com /User/UnitUrls?unitId=3804	

NOTE

25.8 REFERENCE

- IEEE Std 829 – Standard for Software Test Documentation.
- ISO/IEC 25010 – Systems and Software Quality Models.
- CMMI Product Team. CMMI for Development, SEI.

25.9 EXERCISE

Part–1

A. Multiple Choice Questions

Q1. Regression testing is performed to:
A) Increase code size
B) Ensure changes do not break existing features
C) Reduce requirements
D) Avoid automation

Q2. Regression testing is mostly required after:
A) Requirement gathering
B) Code changes or bug fixes
C) Hiring new staff
D) Writing documentation

Q3. Regression test suite contains:
A) Random tests only
B) Previously executed test cases
C) Only new test cases
D) Only performance tests

Q4. Automation is preferred for regression testing because:
A) It increases manual effort
B) It saves time and ensures repeatability
C) It removes all defects
D) It avoids testing

Q5. Retesting is different from regression because retesting:
A) Tests unchanged features
B) Confirms a specific defect fix
C) Avoids test cases
D) Tests only performance

Answers with explanation:

NOTE

1. **Answer:** B
   Regression checks old functionality.
2. **Answer:** B
   Changes may introduce new defects.
3. **Answer:** B
   Old tests verify stability.
4. **Answer:** B
   Regression is repetitive.
5. **Answer:** B
   Retesting verifies fix; regression checks side effects.

B. Small Answer Questions

1. Define regression testing.
2. What is retesting?
3. Write two advantages of regression testing.
4. Why is automation used in regression testing?
5. Write two examples where regression testing is required.

Part–2

C. Long Answer Type Questions (200 words)

1. Explain regression testing and its importance.
2. Differentiate regression testing and retesting.
3. Discuss regression test suite selection.
4. Explain automation in regression testing.
5. Discuss challenges in regression testing.

Part–3

D. Long Answer Type Questions (300 words)

1. Explain regression testing in detail with examples.
2. Discuss regression testing strategies and best practices.
3. Explain regression testing in Agile and CI/CD pipelines.
4. Discuss how regression testing reduces risk in software releases.
5. Explain regression testing tools and test suite maintenance.

UNIT–26
OBJECT ORIENTED APPLICATION TESTING

UNIT STRUCTURE

26.1 Introduction
26.2 Objective
26.3 Object Oriented Application Testing
  26.3.1 Testing of Internal Controls
  26.3.2 COTS Testing
  26.3.3 Client Server Testing
  26.3.4 Web Application Testing
  26.3.5 Mobile Application Testing
  26.3.6 eBusiness eCommerce Testing
  26.3.7 Agile Development Testing
  26.3.8 Data Warehousing Testing
26.4 Summary
26.5 Glossary
26.6 Answer to check your progress
26.7 Suggested readings - OER
26.8 Reference
26.9 Exercise

26.1 INTRODUCTION

Object Oriented Application Testing is an important area of Software Quality Assurance and Testing (SQAT) that focuses on validating software systems developed using object-oriented paradigms. Object-oriented development (OOD) emphasizes concepts such as objects, classes, inheritance, polymorphism, encapsulation, and abstraction. These concepts enhance software reusability, scalability, and maintainability; however, they also introduce new challenges for testing activities.

Unlike traditional procedural systems, object-oriented applications consist of interacting objects whose behavior depends on state, message passing, and dynamic binding. Defects in such systems may arise due to improper object interactions, incorrect inheritance hierarchies, misuse of polymorphism, or violations of encapsulation. Therefore,

NOTE

testing object-oriented applications requires specialized strategies that address both structural and behavioral aspects of objects.

With the widespread adoption of object-oriented technologies in areas such as client-server systems, web applications, mobile applications, eBusiness platforms, agile development environments, and data warehousing systems, object-oriented testing has become indispensable. This unit discusses the principles of object-oriented application testing and explains various testing contexts where object-oriented techniques are extensively applied.

The advancement of software engineering has led to the widespread adoption of object-oriented technologies for developing complex, scalable, and maintainable software systems. Object-oriented applications are built around concepts such as classes, objects, inheritance, encapsulation, and polymorphism. While these concepts offer significant advantages in terms of reusability and modularity, they also introduce new challenges for Software Quality Assurance and Testing. Traditional testing approaches designed for procedural systems are often insufficient to address the dynamic behavior and interactions present in object-oriented systems.

Object-oriented application testing focuses on validating not only individual components but also the interactions among objects and the behavior of the system as a whole. In addition, modern software systems are deployed across diverse environments such as client-server architectures, web platforms, mobile devices, and enterprise data systems. This unit explores how testing principles are applied to object-oriented applications across different domains, highlighting specialized testing approaches required for contemporary software systems such as web applications, mobile applications, eBusiness platforms, agile projects, and data warehousing systems.

26.2 OBJECTIVE

After studying this unit, you will be able to::

- Analyse the concept and importance of object-oriented application testing.
- Identify testing challenges associated with object-oriented software systems.
- Explain testing of internal controls in object-oriented applications.
- Understand testing approaches for Commercial Off-The-Shelf (COTS) software.
- Explain client-server testing in object-oriented environments.

NOTE

- Describe web, mobile, and eBusiness application testing.
- Explain data warehousing testing in object-oriented systems.

26.3 OBJECT ORIENTED APPLICATION TESTING

Object Oriented Application Testing involves validating the correctness, reliability, and performance of software systems developed using object-oriented principles. It focuses on testing individual classes, object interactions, inheritance structures, and system behavior as a whole.

Testing in object-oriented systems is typically performed at multiple levels, including class testing, integration testing, system testing, and acceptance testing. Special emphasis is placed on testing object interactions, dynamic behavior, and reuse mechanisms.

Object-oriented application testing involves validating software systems developed using object-oriented methodologies. Unlike traditional systems, object-oriented applications emphasize interactions among objects rather than linear control flow. As a result, testing must address object states, message passing, inheritance hierarchies, and dynamic binding. Object-oriented testing typically includes testing at the class level, integration testing of object collaborations, and system-level testing to validate overall behavior.

This form of testing also considers reuse of components, as object-oriented systems often incorporate existing classes or frameworks. Ensuring that reused components function correctly in new contexts is a critical aspect of object-oriented application testing. The following subsections discuss specific testing approaches relevant to different application domains and technologies.

26.3.1 Testing of Internal Controls

Testing of Internal Controls in object-oriented applications ensures that control mechanisms embedded within classes and objects function correctly. Internal controls include access control, validation logic, exception handling, authorization checks, and data integrity mechanisms.

This testing verifies that encapsulation is properly enforced and that object states cannot be manipulated in unauthorized ways. It also ensures that business rules implemented within objects are consistently applied. Effective testing of internal controls enhances system security, reliability, and compliance with organizational policies.

Testing of internal controls focuses on verifying the mechanisms within an application that ensure correct processing, data integrity, and security. In object-oriented systems, internal controls are often implemented through encapsulation, access modifiers, validation logic,

NOTE

and exception handling mechanisms. Testing these controls ensures that objects maintain consistent states and that unauthorized access to internal data is prevented.

This type of testing examines whether business rules are enforced correctly and whether error conditions are handled gracefully. By validating internal controls, testers can ensure that the system operates reliably and securely under normal and abnormal conditions. Effective testing of internal controls is essential for maintaining trust in software systems, especially those handling sensitive or critical data.

26.3.2 COTS Testing

Commercial Off-The-Shelf (COTS) Testing involves evaluating third-party object-oriented software components that are integrated into an application. Since source code is often unavailable, COTS testing relies heavily on black-box testing techniques.

COTS testing verifies functionality, compatibility, performance, security, and interoperability of purchased components. It also ensures that COTS software integrates seamlessly with existing object-oriented systems. Thorough COTS testing reduces dependency risks and ensures system stability.

Commercial Off-The-Shelf software testing focuses on evaluating ready-made software products that are integrated into larger systems. COTS components are widely used to reduce development time and cost, but they present unique challenges for testing because their internal design and source code are often unavailable. As a result, testing relies heavily on functional, integration, and performance testing.

COTS testing ensures that the software meets organizational requirements and integrates smoothly with existing systems. It also verifies vendor claims regarding functionality, compatibility, and reliability. Thorough testing of COTS products helps organizations reduce dependency risks and ensures that third-party software does not compromise overall system quality.

Activity Title: Testing an Object-Oriented Application

Objective:
To understand the testing of classes, objects, methods, inheritance, and interactions in an object-oriented application.

Materials Required:

- Computer
- Simple object-oriented program/application
- Notebook

NOTE

- Testing checklist

Procedure/Steps:

1. Select a simple object-oriented program, such as a student management or banking application.
2. Identify its classes and objects.
3. List important methods associated with each class.
4. Prepare test cases for individual methods.
5. Identify relationships such as inheritance or association, if present.
6. Test interactions between objects and classes.
7. Record expected and actual results.
8. Identify any defects and suggest corrective actions.

Outcome:
The learner will be able to identify important object-oriented components and prepare basic test cases for classes, methods, and interactions.

26.3.3 Client Server Testing

Client-Server Testing focuses on validating applications where processing is distributed between client and server components. Object-oriented technologies are commonly used in designing both client-side and server-side components.

This testing ensures proper communication, data consistency, synchronization, and error handling between client and server objects. It also verifies scalability, performance, and security of distributed object interactions.

Client-server testing focuses on applications where processing is distributed between client devices and centralized servers. In such architectures, objects interact across network boundaries, making testing more complex. Client-server testing verifies correct communication, data exchange, synchronization, and error handling between clients and servers.

This type of testing also evaluates performance under different network conditions and user loads. By testing client-server interactions, organizations can ensure that distributed object-oriented applications remain responsive, reliable, and secure in real-world environments.

26.3.4 Web Application Testing

Web Application Testing evaluates object-oriented web applications that operate in browser-based and networked environments. Such

NOTE

applications often involve multiple layers, including presentation, business logic, and data access layers.

Testing focuses on functionality, usability, security, compatibility, performance, and reliability. Object-oriented testing ensures correct interaction between web components, services, and backend systems.

Web application testing addresses software systems accessed through web browsers and deployed over the internet or intranets. Object-oriented principles are commonly used in web applications through frameworks and server-side components. Web application testing verifies functionality, usability, compatibility, security, and performance across different browsers and platforms.

Testing web applications also involves validating session management, input validation, and integration with backend systems. Given the public exposure of web applications, thorough testing is essential to prevent security vulnerabilities and ensure consistent user experience.

26.3.5 Mobile Application Testing

Mobile Application Testing involves testing object-oriented applications designed for mobile devices such as smartphones and tablets. Mobile environments introduce challenges such as device fragmentation, limited resources, and varying network conditions.

This testing ensures functionality, usability, performance, security, and compatibility across different devices and operating systems. Object-oriented testing techniques help validate modular and reusable mobile application components.

Mobile application testing focuses on applications developed for smartphones and tablets using object-oriented frameworks. Mobile environments introduce additional challenges such as limited resources, varying screen sizes, and diverse operating systems. Mobile application testing ensures that applications function correctly across different devices, operating system versions, and network conditions.

This testing also evaluates usability, performance, battery consumption, and responsiveness. Effective mobile application testing helps deliver reliable and user-friendly applications in a highly competitive and fast-evolving market.

26.3.6 eBusiness eCommerce Testing

Testing eBusiness and eCommerce applications ensures that online business systems function correctly, securely, and efficiently. These applications are typically object-oriented and involve complex workflows, transactions, and integrations.

NOTE

Testing focuses on transaction processing, data security, payment gateways, performance, scalability, and user experience. Robust testing is essential to prevent financial losses and maintain customer trust.

eBusiness and eCommerce testing focuses on applications that support online business transactions, including shopping carts, payment gateways, order processing, and customer management. These systems are often object-oriented and integrate multiple components such as databases, third-party services, and web interfaces.

Testing eBusiness applications ensures accuracy of transactions, data security, system performance during peak loads, and compliance with regulatory standards. Given the financial and reputational risks involved, thorough testing is critical for maintaining customer trust and business continuity.

26.3.7 Agile Development Testing

Agile Development Testing emphasizes continuous testing throughout iterative development cycles. Object-oriented applications developed using agile methodologies require frequent validation of changing requirements and designs.

Testing in agile environments relies on automation, unit testing, integration testing, and collaboration between developers and testers. Object-oriented design supports agile testing through modularity and reusability.

Agile development testing is conducted within agile software development environments that emphasize iterative development, frequent releases, and close collaboration among stakeholders. In agile projects, object-oriented applications evolve continuously, requiring ongoing testing throughout the development process.

Testing in agile environments focuses on automation, continuous integration, and rapid feedback. Agile testing ensures that new features do not break existing functionality and that quality is maintained despite frequent changes. This approach aligns testing closely with development, improving adaptability and responsiveness.

26.3.8 Data Warehousing Testing

Data Warehousing Testing involves validating object-oriented systems used for data storage, analysis, and reporting. These systems integrate data from multiple sources and support decision-making processes.

Testing ensures data accuracy, consistency, completeness, performance, and security. Object-oriented testing techniques help verify data transformation logic, schema design, and reporting mechanisms.

NOTE

Data warehousing testing focuses on systems designed to store, integrate, and analyze large volumes of data. Object-oriented techniques are often used in designing data access layers and analytical components. Data warehousing testing verifies data accuracy, consistency, completeness, and performance.

This testing also examines data extraction, transformation, and loading processes to ensure that data is correctly processed from multiple sources. Effective data warehousing testing supports reliable decision-making by ensuring the integrity and quality of analytical data.

Check Your Progress 26.1:

Notes:
(a) Space is given below for your answer.
(b) Check your answer with the one given at the end of this unit.

Fill-in-the-Blank Questions:

Object-oriented applications are based on the concept of _____.

COTS testing mainly uses _____ testing techniques.

Agile testing emphasizes _____ testing.

True/False Questions:

bject-oriented testing focuses only on individual classes.

Web applications often use object-oriented design principles.

Q1. Object Oriented Application Testing mainly focuses on testing:
A) Only hardware components
B) Objects, classes, inheritance, and interactions between components
C) Only user manuals
D) Only network cables

_____

Q2. Client-server testing ensures correct interaction between client requests and server responses.

_____

Q3. Testing of ready-made commercial software components is called _____ testing.

_____

Q4. Write any two types of application testing included in this unit.

_____

NOTE

Q5. Data warehousing testing mainly checks:
A) Screen colors
B) Data accuracy, ETL process, and reporting correctness
C) Keyboard speed
D) Printer connection

_____

26.4 SUMMARY

Object Oriented Application Testing addresses the unique challenges posed by object-oriented software systems. By testing internal controls, COTS components, client-server interactions, web and mobile applications, eBusiness systems, agile projects, and data warehousing solutions, organizations can ensure high-quality, secure, and reliable software products. Object-oriented testing techniques play a vital role in modern software quality assurance.

This unit provides an overview of object-oriented application testing and its application across various modern software domains. It explains how object-oriented principles influence testing strategies and highlights the importance of validating object interactions, internal controls, and system behavior. The unit discusses testing approaches for COTS software, client-server systems, web applications, mobile applications, eBusiness platforms, agile development environments, and data warehousing systems.

By understanding these testing contexts, learners can appreciate the need for specialized testing strategies tailored to different technologies and architectures. Object-oriented application testing plays a vital role in ensuring the reliability, security, and effectiveness of modern software systems, contributing significantly to overall Software Quality Assurance.

26.5 GLOSSARY

- Object-Oriented Testing: Testing software developed using object-oriented principles.
- Encapsulation: Restricting direct access to object data.
- COTS: Commercial Off-The-Shelf software components.
- Client-Server Architecture: Distributed computing model.
- Data Warehouse: Centralized repository for data analysis.

NOTE

25.6 ANSWERS TO CHECK YOUR PROGRESS – POSSIBLE ONE

Fill-in-the-Blank:

Objects

Black-box

Continuous

True/False Questions:

False

True

Q1. Answer: B) Objects, classes, inheritance, and interactions between components
Explanation: OOP testing checks class behavior, object communication, inheritance effects, and overall interactions.

Q2. Client-server testing verifies that the client sends correct requests, the server processes them correctly, and the response returned is accurate and timely.

Q3. COTS software is purchased and used directly. Testing ensures it meets system requirements and integrates properly.

Q4. These are common application testing types used based on the nature of the software system.

Q5. B) Data accuracy, ETL process, and reporting correctness

26.7 SUGGESTED READINGS – OER

Sr. No.	Title	URL	QR Code
1	Open Educational Resource - 1	http://books.aisectonline.com/User/UnitUrls?unitId=37805	

26.8 REFERENCE

• Myers, G. J. The Art of Software Testing.
• IEEE Std 829 – Software Test Documentation.
• ISO/IEC 25010 – Software Quality Models.

NOTE

26.9 EXERCISE

Part–1

A. Multiple Choice Questions

Q1. Object-oriented testing mainly focuses on:
A) Only database queries
B) Classes, objects, inheritance, and polymorphism
C) Only GUI screens
D) Hardware speed

Q2. Testing of classes is usually performed during:
A) Unit testing
B) Installation testing
C) Acceptance testing
D) Beta testing

Q3. Inheritance may cause:
A) No impact
B) Reuse but also hidden defects
C) Only faster execution
D) Less complexity always

Q4. Polymorphism makes testing:
A) Easier always
B) More complex due to dynamic binding
C) Unnecessary
D) Impossible

Q5. Integration testing in OO systems mainly checks:
A) Class interactions and object collaboration
B) Only variable names
C) Only documentation
D) Compiler output

Answers with Explanation:

1. **Answer:** B
   Explanation: testing checks object behavior and interactions.
2. **Answer:** A
   Classes are treated as units.
3. **Answer:** B
   Inherited methods can introduce unexpected behavior.

NOTE

4. **Answer:** B
Different method versions may execute.
5. **Answer:** A
OO integration tests relationships.

B. Small Answer Questions

1. Define object-oriented testing.
2. Write two challenges of OO testing.
3. What is the impact of inheritance on testing?
4. What is polymorphism in OO systems?
5. Write two OO testing techniques.

Part–2

C. Long Answer Type Questions (200 words)

1. Explain object-oriented application testing and its importance.
2. Discuss testing of classes, objects, and methods.
3. Explain challenges of inheritance and polymorphism in testing.
4. Discuss integration testing in OO systems.
5. Explain how OO testing differs from procedural testing.

Part–3

D. Long Answer Type Questions (300 words)

1. Explain object-oriented application testing in detail with examples.
2. Discuss testing strategies for classes, objects, and collaborations.
3. Explain testing issues related to inheritance, encapsulation, and polymorphism.
4. Discuss OO integration testing approaches and test case design.
5. Explain the role of OO testing in ensuring quality of large-scale object-oriented software systems.

Rabindranath TAGORE UNIVERSITY

Approved by : AICTE, NCTE, BCI, INC, MP PARAMEDICAL COUNCIL, DEB (UGC) | Recognized by : UGC | Member of : AIU, ACU

RABINDRANATH TAGORE UNIVERSITY (M.P.)

SLM FEEDBACK FORM

PROGRAMME : _____

COURSE : _____

BLOCK : _____

UNIT : _____

Dear Student,

While studying the units of this block, you may have found certain portions of the text difficult to comprehend. We wish to know your difficulties and suggestions, in order to improve the course. Therefore, we request you to fill up and send us the following questionnaire, which pertains to this block. If you find the space provided insufficient, kindly use a separate sheet.

Please mail to aisectcontent@aisect.org / for online submission please scan QR code:

Questionnaire

1. **Name** : _____

2. **Enrolment No.** : | | | | | | | | | | | |

3. **How many hours did you need for studying the units?** : | |

4. **Please give your reactions to the following items based on your reading of the block:**

Items	Excellent	Very Good	Good	Poor	Give specific Examples (If poor)
Presentation Quality					
Language and Style					
Illustrations					
Used (diagrams, tables, etc.)					
Conceptual Clarity					
Check Your Progress Questions					
Suggested Readings - OER					

5. **Any other comments:**

_____

_____

_____

_____

***