

Software Engineering

Chapter 3: Testing Strategies & Methods

Study Notes for Engineering Students

Poly Notes Hub

Contents

- 1 3.1 Software Testing Fundamentals
 - 2 3.2 A Strategic Approach to Software Testing
 - 3 3.3 Test Strategies for Conventional Software: Unit Testing, Integration Testing, Regression Testing, Smoke Testing
 - 4 3.4 Validation Testing (Alpha & Beta Testing) and System Testing (Recovery, Security, Stress & Performance Testing)
 - 5 3.5 Black Box & White Box Testing
 - 6 3.6 Debugging Process and Strategies
 - 7 Practice Section: 15 MCQs with Answer Key and 10 Broad Questions
-

Poly Notes Hub

3.1 Software Testing Fundamentals

3.1.1 What is Software Testing?

Software testing is the process of executing a program with the intent of finding errors. It is a set of activities that can be planned in advance and conducted systematically to check whether the software does what it is supposed to do, and does not do what it is not supposed to do. Testing typically consumes **30 to 40 percent** of the total project effort, and more for safety-critical systems such as aircraft controls or medical devices.

Objectives of testing (Glen Myers)

- Testing is a process of executing a program with the intent of **finding an error**.
- A **good test case** is one that has a high probability of finding an as-yet-undiscovered error.
- A **successful test** is one that uncovers an as-yet-undiscovered error.

Testing cannot show the absence of defects; it can only show that defects are present. As Dijkstra observed, testing shows the presence, not the absence, of bugs.

3.1.2 Important Terms

Term	Meaning
Error (mistake)	A human action that produces an incorrect result, such as a wrong design or coding decision
Fault (defect, bug)	The manifestation of an error in the software; an incorrect step, process or data definition in the code
Failure	The inability of the system to perform its required function when the fault is executed
Test case	A set of inputs, execution conditions and expected results developed for a particular objective
Test suite	A collection of test cases grouped for a specific purpose
Test oracle	A mechanism (document, tool or expert) used to decide the expected result for a test
Test driver / stub	Supporting code written to call a module under test (driver) or to stand in for modules it calls (stub)

3.1.3 Verification and Validation

Verification asks "Are we building the product right?" It checks that the software correctly implements a specific function, and uses reviews, inspections and walkthroughs. **Validation** asks "Are we building the right product?" It checks that the software is traceable to customer requirements, and is mainly done by testing the executing software.

Basis	Verification	Validation
Question	Are we building the product right?	Are we building the right product?
Checks	Conformance to the specification	Conformance to the user's needs
Activities	Reviews, walkthroughs, inspections	Testing, user acceptance
Performed on	Documents, design, code (static)	The running software (dynamic)

Basis	Verification	Validation
Timing	Throughout development	Mostly after the product or a build is ready
Done by	Developers, QA team	Testing team, customer

Together, verification and validation (V&V) are part of the broader umbrella of **software quality assurance**.

3.1.4 Principles of Testing (Davis)

- All tests should be **traceable to customer requirements**; the most severe defects are those that make the program fail to meet requirements.
- Tests should be **planned long before testing begins**; planning can start as soon as the requirements model is complete.
- The **Pareto principle** applies: about 80 percent of the errors are likely to be traced to about 20 percent of the program components.
- Testing should begin **"in the small"** (individual components) and progress toward testing **"in the large"** (the whole system).
- **Exhaustive testing is not possible**. Even a small program has an enormous number of path permutations, so testing must be selective and risk-driven.
- To be most effective, testing should be conducted by an **independent third party**, since developers are psychologically attached to their own code.

3.1.5 Testability

Testability is how easily a computer program can be tested. James Bach lists these characteristics of testable software:

Characteristic	Meaning
Operability	"The better it works, the more efficiently it can be tested"; few bugs block the test run
Observability	"What you see is what you test"; outputs, states and internal variables are visible
Controllability	"The better we can control the software, the more testing can be automated and optimised"
Decomposability	Controlling the scope of testing; independent modules can be tested separately
Simplicity	Less to test: functional, structural and code simplicity
Stability	Fewer changes means fewer disruptions to testing
Understandability	The more information available (design, documents), the smarter the testing

3.1.6 Attributes of a Good Test

- A good test has a **high probability of finding an error**.
- A good test is **not redundant**; every test should have a different purpose.

- A good test should be "**best of breed**": among similar tests, the one with the highest likelihood of uncovering a whole class of errors.
- A good test should be **neither too simple nor too complex**; combining too many checks can mask errors.

3.1.7 Two Ways of Looking at Testing

- **Black box (functional) view**: knowing the specified function of the product, tests show that each function is fully operational.
 - **White box (structural) view**: knowing the internal workings, tests show that internal operations are performed according to specification and all internal components are exercised.
-

3.2 A Strategic Approach to Software Testing

3.2.1 What is a Testing Strategy?

A **testing strategy** is a template for software testing: a set of steps into which we can place specific test case design techniques and testing methods. It states what tests will be done, in what order, by whom, how much effort will be spent, and how to know when enough testing is done. A strategy must accommodate **low-level tests** (to verify a small source code segment) as well as **high-level tests** (to validate major system functions against customer requirements).

Generic characteristics of a testing strategy

- Testing begins at the **component level** and works "outward" toward the integration of the entire system.
- Different testing techniques are appropriate for **different software engineering approaches** and at different points in time.
- Testing is conducted by the **developer of the software** and, for large projects, an **independent test group**.
- Testing and **debugging** are different activities, but debugging must be accommodated in any testing strategy.

3.2.2 Organising for Software Testing

There is a natural conflict of interest: the developer wants to demonstrate that the program works, while a tester wants to break it. Therefore:

- The **software developer** is responsible for testing the individual units (components) of the program, ensuring each performs the function it was designed for. Developers often also conduct integration testing, which leads to the complete software architecture.
- The **Independent Test Group (ITG)** removes the inherent problems of letting the builder test the thing that has been built. The ITG works with the developers throughout the project to ensure that thorough testing is conducted. It becomes involved in planning and specifying test procedures.
- The developer and ITG must work closely together: the developer must remain available to correct errors uncovered, and the ITG reports to the quality assurance side, not to the project manager.

3.2.3 The Testing Strategy: Spiral View

The testing strategy can be viewed as a spiral that moves outward from the centre:

Step	Focus	Description
1. Unit testing	Source code	Tests each component or module; spiral starts at the vortex with the code
2. Integration testing	Design and architecture	Focuses on assembling components and the issues of verification and program construction
3. Validation testing	Requirements	Validates the software against requirements established during requirements analysis
4. System testing	System engineering	Software and other system elements (hardware, people, databases) are tested as a whole

Moving outward along the spiral, the scope of testing widens: from a single unit to the whole system in its operating environment.

3.2.4 Criteria for Completion of Testing

A classic question is "When are we done testing?" No definitive answer exists, but practical responses are:

- **"You are never done testing;** the burden simply shifts from you to the customer." Every time the customer uses the program, a test is being conducted.
- **"You are done testing when you run out of time or run out of money."**
- A more rigorous approach uses **statistical quality assurance and reliability models**, for example a logarithmic Poisson execution-time model, to predict the failure rate and decide when the software reaches an acceptable level of reliability.

3.2.5 Strategic Issues (Tom Gilb)

For a testing strategy to succeed, the following issues must be addressed:

- Specify product requirements in a **quantifiable manner** long before testing commences.
- State testing objectives **explicitly** (for example, test effectiveness, test coverage, mean time to failure, cost to find and fix defects).
- Understand the **users** of the software and develop a profile for each user category; use cases help.
- Develop a testing plan that emphasises **"rapid cycle testing"**.
- Build **"robust" software** that is designed to test itself (self-diagnosis, anti-bugging).
- Use **effective technical reviews** as a filter before testing.
- Conduct technical reviews to assess the **test strategy and test cases** themselves.
- Develop a **continuous improvement approach** for the testing process, using metrics.

3.2.6 The Test Plan

A **test plan** documents the strategy. A typical test plan contains: scope and objectives, items to be tested and not tested, approach, schedule and milestones, resources and responsibilities, test environment, entry and exit criteria, risks and contingencies, and deliverables (test cases, test reports, defect logs).

3.3 Test Strategies for Conventional Software

A conventional software testing strategy proceeds from unit testing, through integration testing, to higher-order tests (validation and system testing). Regression and smoke testing are applied repeatedly along the way.

3.3.1 Unit Testing

Unit testing focuses verification effort on the smallest unit of software design: the **component or module**. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. Because the scope is limited, unit testing is a **white-box oriented** activity and can be conducted in parallel for multiple components.

Unit test considerations (what is tested)

- **Module interface:** tests ensure that information properly flows into and out of the unit (number and order of parameters, data types, units, global variables, file attributes and I/O).
- **Local data structures:** tests ensure that data stored temporarily maintains its integrity during execution (improper typing, incorrect initialisation, inconsistent data types, overflow, underflow).
- **Boundary conditions:** tests ensure the module operates properly at the boundaries (maximum, minimum, just above and just below limits, loop boundaries).
- **Independent paths:** all independent paths through the control structure are exercised at least once.
- **Error-handling paths:** tests verify that error conditions are anticipated, messages are meaningful and handling is correct.

Common errors found during unit testing: misunderstood or incorrect arithmetic precedence, mixed-mode operations, incorrect initialisation, precision inaccuracy, incorrect symbolic representation of an expression, comparison of different data types, incorrect logical operators, improper loop termination, failure to exit when divergent iteration is encountered, and improperly modified loop variables.

Unit test procedures: drivers and stubs

Since a component is not a stand-alone program, supporting code is needed:

- A **driver** is a "main program" that accepts test case data, passes it to the component under test and prints the results.
- A **stub** (dummy subprogram) replaces modules that are called by the component being tested. It uses the subordinate module's interface, does minimal data manipulation, prints verification of entry and returns control.

Drivers and stubs are **overhead**: code that must be written but is not delivered with the final product. Unit testing is simplified when a component with high cohesion is designed, because the number of test cases reduces and errors are easily predicted.

3.3.2 Integration Testing

Once individual units pass their tests, the question arises: "If they all work individually, why do you doubt that they will work when put together?" The problem is interfacing: data can be lost across an interface, one component can have an adverse effect on another, sub-functions combined may not produce the desired major function, and global data structures can present problems. **Integration testing** is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.

Non-incremental approach: Big bang. All components are combined in advance and the entire program is tested as a whole. The result is usually chaos: a set of errors is encountered, and correction is difficult because the isolation of causes is complicated.

Incremental integration. The program is constructed and tested in small increments, where errors are easier to isolate and correct and interfaces are more likely to be tested completely.

Approach	How it works	Advantages	Disadvantages
Top-down	Start from the main control module; integrate subordinate modules downward, either depth-first or breadth-first ; stubs replace lower modules	Major control and decision points tested early; early demonstration of a working skeleton	Needs many stubs; low-level processing needed to test upper levels may not be available early
Bottom-up	Start with the lowest-level (atomic) modules; group them into clusters (builds) ; use drivers; drivers are removed as integration moves upward	No stubs needed; easier test case design and observation of results	The program as an entity does not exist until the last module is added; control flaws found late
Sandwich (hybrid)	Combines top-down for upper layers and bottom-up for lower layers, meeting in the middle	Uses advantages of both; good for large systems	More complex planning; middle layer may need extra testing
Big bang	All modules integrated at once	Simple, little planning	Difficult to isolate errors; risky for large systems

Steps in bottom-up integration: (1) low-level components are combined into clusters that perform a specific software sub-function; (2) a driver is written to coordinate test case input and output; (3) the cluster is tested; (4) drivers are removed and clusters are combined moving upward in the program structure.

Critical modules. Modules that address several requirements, have a high level of control, are complex or error-prone, or have definite performance requirements should be identified and tested as early as possible.

3.3.3 Regression Testing

Each time a new module is added as part of integration testing, the software changes: new data flow paths are established, new I/O may occur and new control logic is invoked. These changes may cause problems with functions that previously worked. **Regression testing** is the re-execution of a subset of tests that have already been conducted to ensure that changes have not propagated **unintended side effects**.

Regression testing may be conducted manually by re-executing a subset of all test cases, or using **automated capture/playback tools**, which enable the tester to record test cases and results for subsequent playback and comparison. The regression test suite contains three classes of test cases:

- A **representative sample of tests** that exercise all software functions.
- **Additional tests** that focus on software functions likely to be affected by the change.
- Tests that focus on the **software components that have been changed**.

As integration testing proceeds, the number of regression tests can grow very large, so the suite should be designed to include only tests that address one or more classes of errors in each major program function.

3.3.4 Smoke Testing

Smoke testing is an integration testing approach commonly used when product software is developed for time-critical projects. It is designed as a **pacing mechanism**: the software is rebuilt (with new components added) and smoke tested every day. The name comes from hardware testing: switch on a new circuit and see if it "smokes".

Activities in smoke testing

- 1 Software components that have been translated into code are integrated into a **build**. A build includes all data files, libraries, reusable modules and engineered components required to implement one or more product functions.
- 2 A series of tests is designed to expose errors that will keep the build from properly performing its function; the intent is to uncover **"show-stopper" errors** that have the highest likelihood of throwing the software project behind schedule.
- 3 The build is integrated with other builds, and the entire product (in its current form) is **smoke tested daily**. The integration approach may be top-down or bottom-up.

Benefits of smoke testing

- **Integration risk is minimised**: incompatibilities and show-stopper errors are uncovered early, because the build is tested daily.
- **The quality of the end product is improved**: it is likely to uncover functional errors as well as architectural and component-level design defects.
- **Error diagnosis and correction are simplified**: errors uncovered are likely to be associated with the "new software increment", i.e., the latest build.
- **Progress is easier to assess**: each day more software has been integrated and more has been demonstrated to work, which boosts team morale.

Basis	Smoke testing	Regression testing
Purpose	Check that the build is stable enough for further testing	Check that changes have not broken existing functionality
Scope	Shallow and wide; critical functions only	Deeper; many previously passed tests
When	On every new build, daily	After changes, fixes or additions
Outcome	Accept or reject the build	Confirm no unintended side effects

3.4 Validation Testing and System Testing

3.4.1 Validation Testing

Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package and interfacing errors have been uncovered and corrected. Validation succeeds when the software **functions in a manner that can be reasonably expected by the customer**. The reasonable expectations are defined in the **Software Requirements Specification (SRS)**, particularly its validation criteria section.

Validation is achieved through a series of tests that demonstrate conformity with requirements. A **test plan** outlines the classes of tests to be conducted and a **test procedure** defines specific test cases. The plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioural characteristics are achieved, all content is accurate, all performance requirements are attained, documentation is correct, and usability and other requirements (portability, compatibility, error recovery, maintainability) are met.

After each validation test case, one of two conditions exists: (1) the function or performance characteristic conforms to specification and is accepted, or (2) a **deviation** from specification is uncovered and a **deficiency list** is created. Deviations found at this stage are rarely corrected before the scheduled delivery, so negotiation with the customer is often needed.

Configuration review (audit). An important element of the validation process is a configuration review to ensure that all elements of the software configuration have been properly developed, catalogued and have the necessary detail to support the support phase.

3.4.2 Alpha and Beta Testing

It is virtually impossible for a software developer to foresee how the customer will really use a program: instructions may be misinterpreted, strange data combinations used, and output that seemed clear to the tester may be confusing to a user. When software is built for many customers, a series of **acceptance tests** is conducted to let end users validate all requirements. The two common forms are alpha and beta testing.

Basis	Alpha testing	Beta testing
Where	At the developer's site	At one or more customer (end-user) sites
Conducted by	A representative group of end users	Actual end users
Environment	Controlled environment	Uncontrolled , real-world environment
Developer presence	Developer "looks over the shoulder" of the user and records errors and usage problems	Developer is generally not present
Reporting	Developer records problems directly	Customer records all problems and reports them to the developer at regular intervals
Timing	Before beta testing	After alpha testing, before general release
Result	Errors fixed; software made more stable	Modifications made; software released to the entire customer base

A variation of beta testing, called **customer acceptance testing**, is sometimes performed when custom software is delivered to a customer under contract. The customer performs a series of specific tests to uncover errors before accepting the software from the developer.

3.4.3 System Testing

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Software is only one element of a larger system (hardware, people, information and so on). System testing verifies that all system elements have been properly integrated and perform allocated functions. The main types are described below.

1. Recovery testing

A system test that forces the software to **fail in a variety of ways** and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), re-initialisation, checkpointing mechanisms, data recovery and restart are evaluated for correctness. If recovery requires human intervention, the **mean time to repair (MTTR)** is evaluated to determine whether it is within acceptable limits. Example: unplugging the network during a transaction and checking that no data is lost on restart.

2. Security testing

Verifies that the **protection mechanisms** built into the system will, in fact, protect it from improper penetration. Any computer-based system that manages sensitive information or causes actions that can harm or benefit individuals is a target. The tester plays the role of an intruder: attempting to acquire passwords, attacking the system with custom software, overwhelming it, causing errors deliberately, browsing insecure data, and so on. Given enough time and resources, good security testing will eventually penetrate a system; the designer's goal is to make the cost of penetration greater than the value of the information obtained. Typical checks cover authentication, authorisation, data encryption, SQL injection and session management.

3. Stress testing

Executes the system in a manner that demands resources in **abnormal quantity, frequency or volume**. Examples: generating ten interrupts per second when one or two is average; increasing input data rates by an order of magnitude; running test cases that require maximum memory; test cases that cause thrashing in a virtual operating system; test cases that cause excessive hunting for disk-resident data. The tester asks: "How high can we crank this up before it fails?" A variation, **sensitivity testing**, tries to uncover data combinations within valid input classes that may cause instability or improper processing.

4. Performance testing

Tests the **run-time performance** of software within the context of an integrated system. It occurs throughout all steps of the testing process, even at the unit level, but true performance is assessed only when all system elements are fully integrated. Performance tests are often coupled with stress testing and usually require **both hardware and software instrumentation** (for example, measuring resource utilisation, response time, throughput and execution intervals).

5. Deployment (configuration) testing

Exercises the software in each environment (operating systems, browsers, devices) in which it is to operate, and examines all installation procedures and installation documentation.

Type	Main question	Example
Recovery	Can the system recover from failure?	Power cut during a database write
Security	Can the system be penetrated improperly?	Trying to bypass login
Stress	What happens beyond normal limits?	Sudden surge to many times the usual users
Performance	Is the system fast and efficient enough?	Response time of a web page under normal load

3.5 Black Box & White Box Testing

Any engineered product can be tested in two ways: knowing the **specified function** it is designed to perform (black box), or knowing the **internal workings** (white box).

3.5.1 Black Box Testing (Behavioural / Functional Testing)

Black box testing treats the software as a "**black box**": the tester focuses on the **functional requirements** (inputs and outputs) without knowledge of the internal structure or code. It is complementary to white box testing and is likely to uncover a different class of errors.

Errors targeted: incorrect or missing functions, interface errors, errors in data structures or external database access, behaviour or performance errors, and initialisation and termination errors. It is applied during **later stages** of testing (validation and system testing).

Black box techniques

(a) Equivalence partitioning. The input domain is divided into **classes of data** (equivalence classes) from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors that might otherwise require many cases. Guidelines:

- If an input condition specifies a **range**, one valid and two invalid equivalence classes are defined.
- If it requires a **specific value**, one valid and two invalid classes are defined.
- If it specifies a **member of a set**, one valid and one invalid class are defined.
- If it is **Boolean**, one valid and one invalid class are defined.

Example: an age field accepting 18 to 60. Classes: valid (18 to 60), invalid (below 18), invalid (above 60). Test inputs: 35, 10, 75.

(b) Boundary Value Analysis (BVA). A greater number of errors occurs at the **boundaries** of the input domain than in the "centre", so test cases are chosen at the edges of the equivalence classes. For range a to b, test **a, b, just above a, just below a, just above b, just below b**. For the age range 18 to 60: test 17, 18, 19, 59, 60, 61. BVA also applies to output conditions.

(c) Other techniques

- **Decision table testing:** combinations of conditions and the resulting actions are tabulated and each rule is tested.
- **Cause-effect graphing:** a graphical technique that represents logical relationships between input conditions (causes) and outputs (effects), then converted into a decision table.
- **State transition testing:** tests the behaviour of a system as it moves among states in response to events.

- **Orthogonal array testing:** used for problems with a small input domain but too many combinations for exhaustive testing; provides maximum coverage with fewer test cases.
- **Comparison (back-to-back) testing:** independently developed versions of the same application are run with the same inputs and outputs compared.

3.5.2 White Box Testing (Glass Box / Structural Testing)

White box testing uses the **control structure** described as part of the component-level design to derive test cases. It guarantees that:

- All **independent paths** within a module have been exercised at least once.
- All **logical decisions** have been exercised on their true and false sides.
- All **loops** are executed at their boundaries and within their operational bounds.
- **Internal data structures** have been exercised to ensure their validity.

Why do it, when black box seems enough? Because logical errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed; we often believe a path is unlikely to be executed when in fact it is executed regularly; and typographical errors are random, so a path may contain them regardless of how obscure it is.

(a) Basis path testing (McCabe)

Basis path testing lets the test case designer derive a **logical complexity measure** of a procedural design and use it as a guide for defining a basis set of execution paths.

- **Flow graph notation:** a graph with **nodes** (circles) representing one or more procedural statements, **edges** (links) representing flow of control, and **regions** (areas bounded by edges and nodes, including the outside area). A predicate node is a node with two or more outgoing edges (a decision).
- **Independent path:** any path through the program that introduces at least one new set of processing statements or a new condition (that is, traverses at least one edge not previously traversed).
- **Cyclomatic complexity $V(G)$:** a software metric giving a quantitative measure of the logical complexity of a program. It defines the number of independent paths in the basis set and an upper bound for the number of tests needed to cover every statement at least once.

$V(G)$ can be computed in three equivalent ways:

- $V(G) = \text{number of regions}$ of the flow graph.
- $V(G) = E - N + 2$, where E is the number of edges and N is the number of nodes.
- $V(G) = P + 1$, where P is the number of predicate (decision) nodes.

Example: a flow graph with 11 edges and 9 nodes has $V(G) = 11 - 9 + 2 = 4$, so four independent paths are required. Higher $V(G)$ means more complex, error-prone code; a value above about 10 is usually considered a candidate for redesign.

Steps of basis path testing: (1) draw the flow graph from the design or code; (2) determine the cyclomatic complexity; (3) determine a basis set of linearly independent paths; (4) prepare test cases that force execution of each path.

(b) Control structure testing

- **Condition testing:** exercises the logical conditions in a program module (simple conditions, compound conditions, relational expressions); detects errors in conditions as well as other errors.

- **Data flow testing:** selects test paths according to the locations of **definitions and uses** of variables in the program.
- **Loop testing:** concentrates on the validity of loop constructs.
 - **Simple loops** (n is the maximum number of passes): skip the loop entirely; one pass; two passes; m passes where $m < n$; n-1, n and n+1 passes.
 - **Nested loops:** start with the innermost loop, setting outer loops to minimum values; conduct simple loop tests for the innermost; work outward; continue until all loops are tested.
 - **Concatenated loops:** if independent, test as simple loops; if one loop's counter is used by the next, treat them as nested.
 - **Unstructured loops:** should be redesigned into structured constructs.

(c) Coverage criteria

Coverage	Meaning
Statement coverage	Every statement is executed at least once
Branch (decision) coverage	Every decision takes both true and false outcomes at least once
Condition coverage	Every condition within a decision takes both true and false values
Path coverage	Every possible path through the code is executed

3.5.3 Comparison: Black Box vs White Box

Basis	Black box testing	White box testing
Knowledge of code	Not required	Required
Focus	Functionality (what the software does)	Internal logic and structure (how it does it)
Based on	Requirements, specifications	Design, source code
Performed by	Testers, end users	Developers, specialised testers
Test levels	Validation, system, acceptance	Unit, integration
Techniques	Equivalence partitioning, BVA, decision tables	Basis path, loop, condition, data flow
Also called	Behavioural, functional, opaque-box testing	Structural, glass-box, clear-box testing
Drawback	Cannot find hidden code errors or untested paths	Cannot detect missing requirements; needs skilled staff

Grey box testing combines the two: the tester has partial knowledge of internal structure (for example, database schema or architecture) while designing tests from the user's point of view.

3.6 Debugging Process and Strategies

3.6.1 The Debugging Process

Debugging occurs as a consequence of successful testing: when a test case uncovers an error, debugging is the process that results in the **removal of the error**. It is an art as much as a science. The debugging process begins with the execution of a test case; results are assessed, and a lack of correspondence between expected and actual performance is encountered. The debugging process then attempts to match symptom with cause, thereby leading to error correction.

Two outcomes are possible: (1) the **cause is found and corrected**, or (2) the **cause is not found**; the person performing debugging may suspect a cause, design a further test case to validate the suspicion, and work toward error correction in an iterative fashion.

Steps in debugging: reproduce the failure; locate the fault; understand the cause; fix the fault; re-test and perform regression testing to ensure the fix has not introduced new errors; document the defect and its fix.

3.6.2 Characteristics of Bugs

- The **symptom and the cause may be geographically remote**: the symptom may appear in one part of the program while the cause is located elsewhere. Highly coupled components exaggerate this.
- The symptom may **disappear (temporarily)** when another error is corrected.
- The symptom may actually be caused by **non-errors** (for example, round-off inaccuracies).
- The symptom may be caused by **human error** that is not easily traced.
- The symptom may be a result of **timing problems** rather than processing problems.
- It may be difficult to **accurately reproduce** input conditions (for example, a real-time application where input ordering is indeterminate).
- The symptom may be **intermittent**, which is common in embedded systems that couple hardware and software.
- The symptom may be due to causes that are **distributed across a number of tasks** running on different processors.

3.6.3 Debugging Strategies

Strategy	Description	Remarks
Brute force	Most common and least efficient. Used when all else fails: memory dumps are taken, run-time traces are invoked, and the program is loaded with output (print) statements. The hope is that the clutter will lead to the cause	Time-consuming; may waste effort
Backtracking	Common in small programs. Starting at the site where a symptom has been uncovered, the source code is traced backward (manually) until the site of the cause is found	As the number of source lines increases, the number of potential backward paths becomes unmanageably large
Cause elimination	Based on induction or deduction and introduces the concept of binary partitioning . Data related to the error occurrence are organised to isolate potential causes; a "cause hypothesis" is devised and data are used to prove or disprove it; alternatively a list of all possible causes is developed and tests are conducted to eliminate each	Systematic and efficient

In practice, strategies are also described as: **induction** (moving from specific symptoms to the general cause by locating pertinent data, organising it, devising and proving a hypothesis) and **deduction** (starting with a list of possible causes and eliminating them one by one through tests).

Automated debugging. Each strategy can be supplemented with debugging tools that provide semi-automated support: **debugging compilers, dynamic debugging aids ("tracers"), automatic test case generators and cross-reference mapping tools** (and today, IDE debuggers with breakpoints, watch windows and step execution). Tools are not a substitute for careful evaluation of the design, code and test cases.

3.6.4 Correcting the Error

Once a bug has been found, it must be corrected. Before making the correction, ask three questions:

- **Is the cause of the bug reproduced in another part of the program?** In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere.
- **What "next bug" might be introduced by the fix I am about to make?** Before the correction is made, the source code (or design) should be evaluated to assess coupling of logic and data structures.
- **What could we have done to prevent this bug in the first place?** This question is the first step toward establishing a statistical quality assurance approach; if we correct both the process and the product, the bug will be removed from the current program and may be eliminated from all future programs.

3.6.5 Testing versus Debugging

Basis	Testing	Debugging
Purpose	To find errors (show that a defect exists)	To locate and remove the cause of a known error
Nature	Planned, designed and scheduled; can be done without design knowledge	Consequence of testing; needs detailed knowledge of the design and code
Done by	Testers or an independent group	Developers
Automation	Largely automatable	Mostly manual, tool-assisted
Outcome	Test report, list of failures	Corrected code
Timing	Can be predicted and bounded	Time required is unpredictable

Practice Section

Part A: 15 Multiple Choice Questions

Q1. The primary objective of software testing is to:

- (a) Prove that the program has no errors
- (b) Find errors in the program
- (c) Increase the lines of code
- (d) Reduce the documentation

Q2. "Are we building the product right?" is the question answered by:

- (a) Validation
- (b) Verification
- (c) Debugging
- (d) Maintenance

Q3. According to the Pareto principle in testing, about 80 percent of errors are traced to:

- (a) 80 percent of the modules
- (b) 50 percent of the modules
- (c) 20 percent of the modules
- (d) 5 percent of the modules

Q4. In unit testing, a module that calls the component under test and passes test data to it is called a:

- (a) Stub
- (b) Driver
- (c) Oracle
- (d) Proxy

Q5. Which integration approach combines all modules at once and tests the whole program?

- (a) Top-down
- (b) Bottom-up
- (c) Sandwich
- (d) Big bang

Q6. Re-running previously passed tests after a change, to detect unintended side effects, is called:

- (a) Smoke testing
- (b) Regression testing
- (c) Alpha testing
- (d) Stress testing

Q7. Smoke testing is also known as:

- (a) Build verification testing
- (b) Beta testing
- (c) Security testing
- (d) Recovery testing

Q8. Alpha testing is performed:

- (a) At the customer's site without the developer
- (b) At the developer's site in a controlled environment
- (c) Only after release to the market
- (d) By the compiler

Q9. Forcing the software to fail in various ways and checking that it recovers properly is:

- (a) Stress testing
- (b) Security testing
- (c) Recovery testing
- (d) Performance testing

Q10. Executing a system with abnormal quantity, frequency or volume of resources is:

- (a) Stress testing
- (b) Unit testing
- (c) Alpha testing
- (d) Regression testing

Q11. For an input range of 1 to 100, boundary value analysis would select which set of test values?

- (a) 1, 50, 100
- (b) 0, 1, 2, 99, 100, 101
- (c) 10, 20, 30
- (d) 0 and 101 only

Q12. Equivalence partitioning is a type of:

- (a) White box testing
- (b) Black box testing
- (c) Debugging
- (d) Static analysis

Q13. The cyclomatic complexity of a flow graph with 12 edges and 9 nodes is:

- (a) 3
- (b) 4
- (c) 5
- (d) 21

Q14. Which debugging strategy uses binary partitioning and induction or deduction?

- (a) Brute force
- (b) Backtracking
- (c) Cause elimination
- (d) Trial and error

Q15. Which statement about testing and debugging is TRUE?

- (a) They are the same activity
- (b) Debugging finds that errors exist; testing removes them
- (c) Testing finds errors; debugging locates and removes their cause
- (d) Debugging is always done by an independent group

Answer Key

Q	Ans	Q	Ans	Q	Ans
1	(b)	6	(b)	11	(b)
2	(b)	7	(a)	12	(b)
3	(c)	8	(b)	13	(c)
4	(b)	9	(c)	14	(c)
5	(d)	10	(a)	15	(c)

Note for Q13: $V(G) = E - N + 2 = 12 - 9 + 2 = 5$.

Part B: 10 Broad Questions

- 1 Define software testing. Explain its objectives and the principles of testing. What are the attributes of a good test?
- 2 Differentiate between verification and validation. Explain testability and its characteristics.
- 3 Explain a strategic approach to software testing. Describe the spiral view of the testing strategy and the criteria for completion of testing.
- 4 What is unit testing? Explain unit test considerations, and the role of drivers and stubs, with a neat diagram.
- 5 Explain integration testing. Compare top-down, bottom-up, sandwich and big bang integration with advantages and disadvantages.
- 6 What is regression testing? Explain smoke testing with its activities and benefits, and differentiate it from regression testing.
- 7 Explain validation testing. Differentiate between alpha testing and beta testing.
- 8 Describe system testing. Explain recovery, security, stress and performance testing with examples.
- 9 What is black box testing? Explain equivalence partitioning and boundary value analysis with examples. Compare black box and white box testing.
- 10 Explain white box testing with basis path testing and cyclomatic complexity (with a flow graph example). Also describe the debugging process and its strategies.