Part I: Testing Methodology – A Comprehensive Guide

Why testing matters

Software is now a critical part of everyday life and powers medical devices, aircraft, banking systems and more. Faulty software can lead to serious economic loss, safety issues and brand damage. Therefore **testing** is a discipline separate from software development; it is about systematically uncovering defects before a product is released. The first part of Naresh Chauhan's *Software Testing: Principles and Practices* introduces the foundational concepts that anyone learning to test software should understand. This guide summarises and expands on that material using examples, diagrams and additional resources.

Evolution of software testing

Software testing has evolved from ad-hoc debugging to a mature discipline with formal processes. Chauhan identifies six historical phases:

Phase	Characteristics
Debugging-oriented (before 1957)	Testing was not separate; developers debugged their code after writing it.
Demonstration-oriented (1957-1978)	Goal was to demonstrate that programs worked; testers focused on showing that software produced correct answers for selected inputs.
Destruction-oriented (1979-1982)	Testers began designing tests intended to break the software and reveal failures.
Evaluation-oriented (1983-1988)	Emphasis moved to measuring software quality using test results and reliability estimates.
Prevention-oriented (1989-1995)	Testing activities shifted left; testers participated in requirements and design reviews to prevent defects.
Process-oriented (1996-present)	Testing is recognised as a process integrated across the Software Development Life Cycle (SDLC) rather than a single phase. It emphasises planning, documentation and continuous improvement.

Myths vs. facts

Common misconceptions harm the testing effort. Chauhan dispels several myths:

- **Myth:** Testing is just executing programs. **Fact:** Testing encompasses planning, analysis, design, execution and evaluation; it starts long before code exists.
- **Myth:** Complete testing is possible. **Fact:** It is impossible to test all inputs, paths and states of non-trivial software【746099003829693†L1577-L1718】. Effective testing chooses representative cases to expose likely defects.
- **Myth:** Testing aims only to show software works. **Fact:** The objective of testing is to **find defects**; testers adopt a destructive mindset to reveal failures.
- **Myth:** Anyone can test. **Fact:** Good testers require domain knowledge, creativity, scepticism and communication skills. Testing is a professional discipline.

Goals and psychology of testing

Testing serves both immediate and long-term goals:

- **Immediate goals:** uncover bugs and **prevent** future bugs by feeding back lessons learned to developers.
- **Long-term goals:** improve software reliability and quality, enhance customer satisfaction, reduce risk and support maintainability.
- **Post-implementation goals:** reduce maintenance cost and continually improve the testing process.

Psychology of testing

A tester should adopt a mindset of **trying to break the software**, not prove it correct. Dijkstra observed that testing can show the **presence** of bugs but never their **absence**【746099003829693†L1577-L1718】. Good testers question assumptions, explore edge cases and view specifications critically. Their intent is constructive: by deliberately uncovering weaknesses, they contribute to a more robust product.

Key definitions

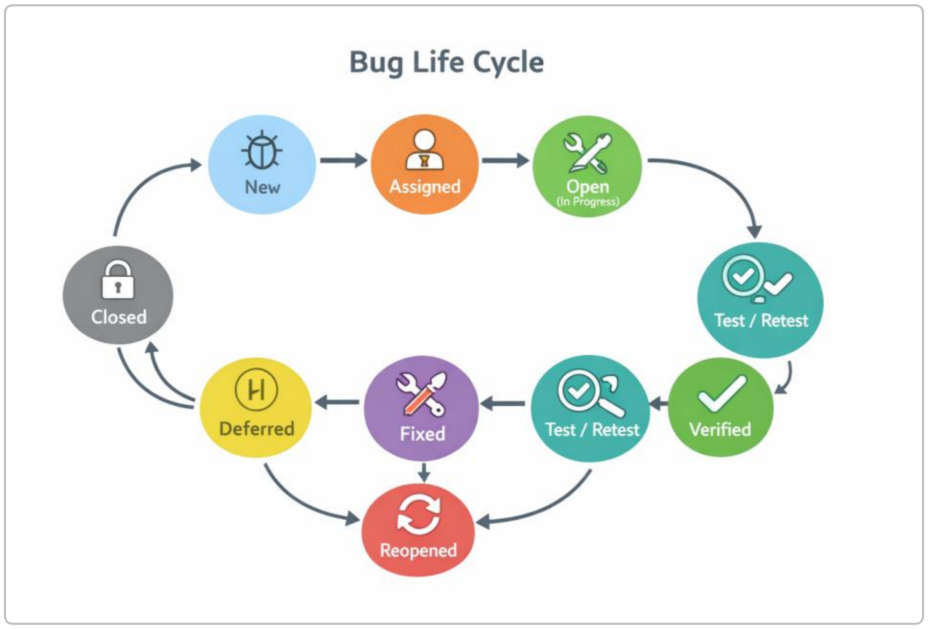
Understanding the terminology of testing prevents confusion:

Term	Meaning
Failure	A mismatch between actual and expected behaviour observed during execution.
Fault / Defect / Bug	A flaw in a component or system that can cause a failure when executed. Faults originate from human errors.
Error	A human mistake (misunderstanding requirement, faulty design, etc.) that introduces a defect into the software.
Test case	A documented procedure with an identifier, purpose, prerequisites, input data and expected outputs used to test a particular aspect.

Term	Meaning
Test suite	A collection of test cases designed to be executed together. *Testware* is a broader term covering all artefacts produced during testing (plans, scripts, environments, etc.).
Incident	A symptom associated with a failure; it becomes a defect report if traced to a specific fault.
Test oracle	A mechanism (algorithm, model, separate program or human expert) that determines whether the outputs of a test are correct.

Bug life-cycle and classification

A defect progresses through several states from discovery to closure. Figure 1 illustrates a typical bug life cycle with states and flows. When a tester finds a defect, its status is **New**. The defect is **Assigned** to a developer and becomes **Open (In Progress)** while being fixed. Once the developer implements a fix, the status changes to **Fixed** and returns to the tester for **Test/Retest**. If the defect persists, it is **Reopened** and re-enters the cycle; otherwise, once verified it is **Closed**. In some organizations a defect may be **Deferred** (postponed) when fixing it is not currently worthwhile.



During testing, many factors influence bugs:

- Bugs arise due to human errors, miscommunication of requirements, unhandled boundary conditions, changes in requirements and propagation of earlier defects.
- Fixing bugs late in the lifecycle is costly. Studies show that removing a defect after delivery may cost 10-100 times more than if it had been removed during design.
- Bugs are classified by **severity** (critical, major, medium, minor) and by origin (requirements, design, coding, interface/integration, system and testing bugs).

Principles of effective testing

Chauhan summarises key principles:

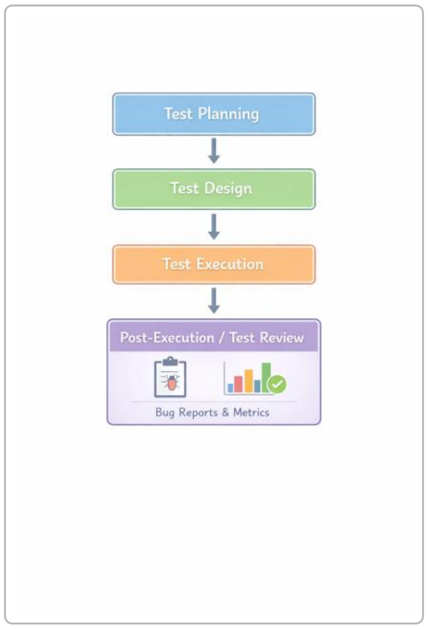
1. **Effective rather than exhaustive:** exhaustive testing is impossible; target high-risk areas and representative inputs.
2. **Testing should begin early:** verification activities start at the requirements and design phases to catch defects early and reduce cost.
3. **Testing is context-dependent:** different projects require different testing approaches; testers should adapt techniques to the context.
4. **Defects cluster together:** a small number of modules often contain the majority of defects; focusing on these modules yields better results.
5. **Testing progresses from units to systems:** start with unit tests, then integration tests, functional tests, system tests and acceptance tests.
6. **Independent testing:** independent testers bring objectivity, though collaboration with developers is essential.
7. **Document everything:** well-documented test plans, cases and results facilitate reproducibility and improvement.
8. **Test both expected and unexpected behaviour:** include invalid and boundary inputs.
9. **Participate in reviews:** testers contribute to requirements and design reviews to prevent defects.

Software testing as a process

Testing is a structured process executed in parallel with development. The **Software Testing Life Cycle (STLC)** comprises four broad phases:

Phase	Key activities
Test planning	Determine test scope and objectives; select testing methodology; assess resources and risks; schedule activities; define entry/exit criteria and metrics.
Test design	Identify what needs to be tested; map requirements to test cases; design detailed test cases with clear inputs and expected results; prepare test data.
Test execution	Set up test environment; execute test cases; log results; report defects with severity and priority; retest fixes; perform regression tests as needed.
Post-execution / Test review	Analyse defect reports; calculate metrics (e.g., defect density, test coverage); identify root causes; document lessons learned and update test process.

The STLC is shown diagrammatically in Figure 2. Each phase feeds into the next, and feedback loops encourage continuous improvement.



Test strategy and methodology

A **test strategy** outlines *what* to test and *how* to test it across the SDLC. Factors such as risk, regulations, environment and resource constraints influence the choice of testing methods. The **test strategy matrix** is a tool for mapping risk areas against test levels to determine priorities. A **testing methodology** combines strategy (objectives, coverage, priorities) with tactics (specific techniques, tools, scheduling). For example, a risk-based methodology might allocate more test effort to modules that are mission-critical or have complex logic.

Verification and validation: building the product right vs. building the right product

Chauhan distinguishes **verification** and **validation** as complementary activities:

- **Verification** checks that work products (requirements, designs, code) conform to specifications and standards through reviews, analyses and static testing. It answers the question, **"Are we building the product right?"** and does not necessarily execute the software.
- **Validation** checks that the final product fulfils user needs by executing the software in realistic scenarios. It answers, **"Are we building the right product?"**[1].

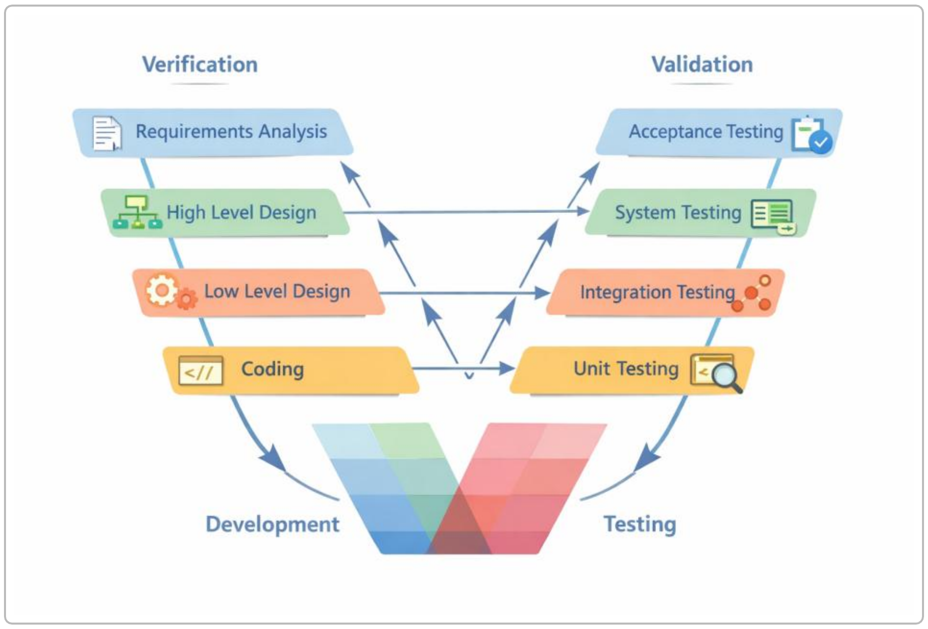
Table 1 contrasts these activities using additional sources [2].

Aspect	Verification	Validation
Objective	Ensure work products meet specifications and standards	Ensure the product meets user and stakeholder needs
Nature	Static analysis; reviews, inspections, walkthroughs	Dynamic execution and evaluation

Aspect	Verification	Validation
Timing	Performed throughout the development lifecycle	Performed when software or components are executable
Activities	Requirements reviews, design reviews, code inspections, static analysis tools	Unit, integration, system and acceptance testing, usability tests
Error focus	Prevent defects by ensuring conformance	Detect defects and ensure correct behaviour

The V-Model

The **V-Model** formalises the relationship between development phases and corresponding testing activities. Verification activities occupy the left side of the V, while validation activities occupy the right. Each development phase produces artifacts that are verified and then validated by a matching test phase. Figure 3 illustrates the V-Model. During requirements analysis, acceptance test plans are prepared; high-level design corresponds to system test planning; low-level design corresponds to integration test planning; coding corresponds to unit test planning.



Verification activities

Chauhan lists the following verification tasks:

1. **Requirements verification:** Ensure the **Software Requirements Specification (SRS)** is correct, unambiguous, consistent, complete and traceable. Testers should derive **acceptance test cases** from acceptance criteria. Ambiguities must be clarified with stakeholders.
2. **High-level design (HLD) verification:** Check that all functional requirements are addressed in the architectural design and that interfaces between modules are consistent. Prepare the **system test plan** and **integration test plan**.

3. **Low-level design (LLD) verification:** Verify that module algorithms and data structures are consistent with the HLD and that each requirement is traceable to module functions. Prepare **unit test plans**.
4. **Code verification:** Perform static analysis, code reviews, walkthroughs and inspections to ensure coding standards, error handling and boundary conditions are implemented correctly. Use tools to detect potential defects (e.g., dead code, memory leaks). Dynamic techniques such as desk checking and dry runs complement static reviews.

Validation activities

Validation involves executing tests to ensure the product behaves as intended [3]. Key validation levels are:

1. **Unit testing:** Tests individual modules or functions in isolation to verify their logic. It checks local data structures, control flow and error handling. Unit tests are typically written by developers and derived from LLD.
2. **Integration testing:** Combines modules and verifies their interfaces and interactions. Integration tests can follow approaches such as big-bang, incremental (top-down or bottom-up) or sandwich. They uncover interface errors like mismatched data types or incorrect protocol sequencing.
3. **System testing:** Tests the complete integrated system against the SRS. It includes functional testing as well as non-functional testing (performance, security, reliability, etc.).
4. **Acceptance testing:** Carried out in a realistic environment by the customer or independent testers to ensure the software meets user needs before deployment. Acceptance tests derive from acceptance test plans prepared during requirements analysis.

Schools of software testing

Chauhan categorises perspectives on testing into five “schools”:

- **Analytical school:** views testing as proving the correctness of programs using logic and formal methods.
- **Standard school:** focuses on managed processes, standards and metrics (e.g., ISO/IEC 9126) to ensure quality.
- **Quality school:** integrates testing into quality assurance; testers prevent defects by participating throughout SDLC.
- **Context-driven school:** argues that testing practices should be adapted to context; no one-size-fits-all methodology exists.
- **Agile school:** emphasises continuous integration, test-driven development and automation to support rapid delivery.

Understanding these perspectives helps testers choose appropriate techniques and challenge assumptions.

Example: classifying bugs by phase and severity

Suppose a web application occasionally crashes when an uploaded file exceeds a size limit. Investigating reveals that the requirement specified a maximum upload size, but the high-level design neglected to enforce this limit. As a result, the code allocates insufficient memory and triggers an exception. This defect can be classified as:

- **Phase:** Requirements (inadequate requirement clarifying limit) and design (missing check for file size), propagating into the coding phase.
- **Severity: Major** because it causes the application to crash, affecting user experience.
- **State progression:** New → Assigned → Open → Fixed → Test/Retest → Closed (if fixed) or Reopened (if persists).

Practice questions and answers

The following questions test your understanding of Part I concepts. Try answering them yourself before reviewing the solutions.

Short-answer questions

1. **Why can't we test software exhaustively?**
Answer: The input domain of most programs is extremely large, programmes may respond differently under different environments and states, and there are too many execution paths to evaluate. Complete testing is therefore infeasible; testers select representative test cases to maximise defect discovery【746099003829693†L1577-L1718】.

2. **Differentiate between error, fault and failure.**
Answer: An **error** is a human mistake made during specification, design or coding. A **fault** (bug/defect) is the manifestation of an error in the code or artefact. A **failure** is the observable misbehaviour (deviation from expected output) when a fault is executed.

3. **List the four phases of the Software Testing Life Cycle (STLC).**
Answer: Test planning, test design, test execution and post-execution/test review.

4. **What is the difference between verification and validation?**
Answer: Verification ensures work products (requirements, design, code) conform to specifications through static reviews and analyses; validation ensures the developed product meets user needs by executing it in realistic scenarios [1].

5. **What is meant by independent testing and why is it recommended?**
Answer: Independent testing means testers are organisationally separate from developers. It reduces bias because testers have no vested interest in proving the software works. Independent testers can challenge assumptions and design effective tests.

Long-answer questions

1. **Explain the goals of software testing and classify them into immediate, long-term and post-implementation goals.**
Solution: The immediate goal of testing is to **discover defects** and prevent their recurrence. This includes executing test cases to reveal bugs and feeding back lessons learned to developers so similar mistakes can be avoided. Long-term goals relate to **quality and reliability**: thorough testing

increases the dependability of software, enhances user satisfaction, manages risk, and builds confidence in the product. Post-implementation goals focus on **maintenance**—well-tested software incurs lower maintenance costs—and on **process improvement**, using metrics and retrospectives to refine the testing process over time.

2. **Describe the bug life cycle with appropriate states and discuss why managing this cycle efficiently is important.**
   Solution: When a tester discovers a defect, it enters the **New** state. After review, it is **Assigned** to a developer and becomes **Open** while being fixed. Once the developer implements a fix, the status becomes **Fixed** and the defect is sent back to the tester for **Test/Retest**. If the bug persists, it is **Reopened**; otherwise it progresses to **Verified** and finally **Closed**. Some organisations include **Deferred**, **Rejected** or **Duplicate** states. Efficiently managing this lifecycle ensures clear communication, prevents defects from being lost, prioritises fixes based on severity and avoids releasing faulty software [4].

3. **What activities are involved in verification at the requirements, design and code levels?**
   Solution: **Requirements verification** checks the SRS for correctness, completeness, consistency, and traceability. Ambiguous requirements are clarified and acceptance criteria are defined; acceptance test plans are prepared. **High-level design verification** ensures that architectural modules cover all functional requirements, interfaces are well defined and exception handling is considered; system and integration test plans are drafted. **Low-level design verification** checks that module algorithms and data structures implement the HLD and that data sizes, boundaries and control logic are correct; unit test plans are produced. **Code verification** involves static analyses, peer reviews, inspections and adherence to coding standards to catch defects early and ensure maintainability.

4. **Discuss the V-Model and explain how it integrates verification and validation. Include an example showing how test plans are linked to development activities.**
   Solution: The V-Model maps each development phase to a corresponding testing phase. On the left branch of the V, verification occurs: requirements analysis, high-level design, low-level design and coding. On the right branch, validation occurs: unit testing, integration testing, system testing and acceptance testing. Each pair across the V is related; for example, requirements analysis yields acceptance test plans that will be used during acceptance testing, high-level design maps to system test planning, low-level design to integration test planning and coding to unit test planning. When coding is complete, unit tests validate the lowest level modules. Integration tests validate interactions between modules defined in the HLD. System tests validate the assembled system against the SRS, and acceptance tests validate the product in the user environment. This structured approach ensures that test planning is performed concurrently with design, leading to early detection of defects and ensuring traceability between requirements and tests.

5. **How do the different schools of software testing influence the way we approach testing? Give one implication of adopting each school.**
   Solution: The **analytical** school encourages formal verification and mathematical reasoning, implying that testers may use proof techniques and model checking for critical systems. The **standard** school emphasises adherence to documented processes and metrics, so organisations may implement quality standards like ISO 9001 and track defect density. The **quality** school integrates testing into quality assurance, encouraging testers to participate in design and code reviews to prevent defects. The **context-driven** school insists that no single best practice fits all projects; testers tailor their

approach based on project constraints, risks and team skills. The **agile** school promotes continuous testing and automation; testers collaborate closely with developers, write automated regression tests and adapt to fast feedback cycles.

Conclusion

Part I of Chauhan's book lays a strong foundation for understanding software testing. It emphasises that testing is more than running programs: it is a process spanning requirements, design and code, with verification and validation activities tightly coupled to development. Effective testers understand the impossibility of exhaustive testing, adopt a fault-finding mindset, apply established principles and adapt methods to the context. By mastering the concepts introduced here—terminology, bug life cycles, testing processes, the V-Model and verification versus validation—you will be prepared to design sound test strategies and contribute to high-quality software.

---

[1] [2] [3] Verification Vs Validation - GeeksforGeeks
https://www.geeksforgeeks.org/software-engineering/differences-between-verification-and-validation/

[4] Bug Life Cycle in Software Development - GeeksforGeeks
https://www.geeksforgeeks.org/software-engineering/bug-life-cycle-in-software-development/