**1. Introduction to Software Testing**

* **Definition:** Process of evaluating software to find defects and ensure it meets requirements.
* **Analogy:** Like a chef tasting dishes before serving customers to make sure they’re perfect.
* **Need of Software Testing:** To ensure software works as expected and meets quality standards.
* **Analogy:** Like checking a bridge before letting cars drive on it — safety first.

**2. Error, Failure, Defect**

* **Error:** Human mistake in coding.
* **Defect/Bug:** The manifestation of an error in the software.
* **Failure:** When software behaves incorrectly during execution.
* **Analogy:**
  + Error = cook forgets salt
  + Defect = soup is under-salted
  + Failure = customer tastes and complains

**3. Causes of Software Defects**

* Ambiguous requirements
* Design mistakes
* Coding errors
* Environment issues
* **Analogy:** Like building a house: wrong blueprint, poor materials, or sloppy construction leads to defects.

**4. Cost of Software Defects**

* Early defects = cheap to fix
* Late defects = expensive
* **Analogy:** Fixing a crack in a wall while building costs less than after the house is finished.

**5. What Testing Reveals**

* Reveals **presence of defects**
* **Does not guarantee absence of defects**
* **Analogy:** Like inspecting fruit — you can spot rotten ones, but can’t guarantee all are perfect.

**6. Importance of Testing Early in SDLC**

* **Find defects early → lower cost, better quality**
* **Analogy:** Like checking ingredients before cooking instead of after the dish is served.

**7. Testing and Quality**

* **Testing ensures software meets requirements and is reliable**
* **Quality perception:** Users judge quality based on software behavior, performance, and usability.
* **Analogy:** Like a restaurant judged by taste, cleanliness, and service — not just by the chef’s skill.

**8. Seven Testing Principles**

| **Principle** | **Explanation** | **Analogy** | **Extra Analogy** |
| --- | --- | --- | --- |
| Exhaustive testing impossible | Can’t test all inputs | Weeding a garden — focus on most important areas | Sampling grains from a sack of rice — impossible to taste all |
| Early testing | Test as early as possible | Catch typos while writing | Fix small roof leak during rain rather than after collapse |
| Defect clustering | Most defects in few modules | Garden weeds cluster | Most potholes appear on one road stretch |
| Pesticide paradox | Same tests stop finding new bugs | Same weed killer stops working | Same medicine repeatedly loses effect |
| Testing shows presence of defects | Can find defects, cannot prove absence | Finding cracks in wall | Hidden cracks may exist even if none visible |
| Context-dependent testing | Testing depends on software type | Clothing choice depends on weather | Winter coat fails in summer |
| Absence-of-errors fallacy | Bug-free ≠ meets requirements | Perfectly cooked dish can taste bad | Looks perfect but fails expectations |

**9. Economics of Testing**

* Balances **cost, effort, and quality**
* **Analogy:** Like investing in safety gear — upfront cost saves huge accidents later.

**10. How Testing is Conducted**

* Test planning, designing, executing, reporting, and closing
* **Analogy:** Like organizing a treasure hunt: plan clues, hide them, test run, and finalize.

**11. Software Testing — Past vs. Present**

* **Then:** Manual, slow, less coverage
* **Now:** Automated, faster, integrated into SDLC, CI/CD pipelines
* **Analogy:** Horse carriage → Cars → Self-driving cars

**12. Scope of Software Testing**

* Testing every aspect is unrealistic — focus on **high-risk, critical areas**
* **Analogy:** Like fire safety inspection — check main exits and alarms first

**13. Risk-Based Testing**

* **Project Risks:** Delays, budget overrun
* **Product Risks:** Critical features failing
* **Analogy:** Like checking lifeboats first on a ship — focus on high-risk areas

**14. Need for Independent Testing**

* Provides **unbiased view** of software quality
* **Analogy:** Like hiring an external food critic instead of tasting your own cooking

**15. Activities in Fundamental Test Process**

1. Test planning
2. Test design
3. Test execution
4. Defect reporting
5. Test closure

* **Analogy:** Like planning a movie production: script → shooting → editing → review → release

**16. Attributes of a Good Tester**

* Curious, detail-oriented, analytical, persistent, ethical
* **Analogy:** Like a detective solving a mystery, noticing every small clue

**17. Psychology of Testing**

* Think like **both user and attacker**
* **Analogy:** Like testing a bank vault — both honest usage and attempts to break in

**18. Code of Ethics for Tester**

* Honest, thorough, impartial, professional
* **Analogy:** Like a referee in a sports match

**19. Limitations of Software Testing**

* Cannot prove absence of defects
* May not cover all scenarios
* **Analogy:** Like testing a car in a track — real-world accidents may differ

**20. Testing Throughout SDLC**

* **Software Development Models:** Waterfall, Agile, V-Model, Spiral
* **Testing in SDLC:** Early, continuous, risk-based
* **Analogy:** Like quality checks at every stage of cake baking — ingredients, batter, oven, decoration

**21. Test Levels**

| **Level** | **Description** | **Analogy** |
| --- | --- | --- |
| Component (Unit) | Individual modules | Inspecting a single Lego block |
| Integration | Modules together | Connecting Lego blocks to form structure |
| System | Complete application | Whole Lego city |
| Acceptance | Meets business/user requirements | Judge inspects completed Lego city |

**1️⃣ Functional Testing**

* **Definition:** Validates that the software behaves according to requirements and performs its intended functions.
* **Example:** Clicking a “Submit” button stores data correctly in a database.
* **Analogy:** Like checking if a car’s engine, brakes, and headlights work as they are supposed to — does it do what it should?

**2️⃣ Non-Functional Testing**

* **Definition:** Checks aspects **other than functionality**, like performance, usability, security, and reliability.
* **Example:** Measuring website load time or testing if the app can handle 1000 users simultaneously.
* **Analogy:** Like judging a car’s smooth ride, fuel efficiency, comfort, and crash safety — not just whether it moves.

**3️⃣ White-box Testing (Structural Testing)**

* **Definition:** Tests the internal code logic, paths, and structure. Testers know the implementation details.
* **Example:** Checking if every condition in an if statement executes at least once.
* **Analogy:** Like a mechanic inspecting the engine, gearbox, and wiring of a car — looks under the hood rather than just driving it.

**4️⃣ Smoke Testing (Build Verification Test)**

* **Definition:** Quick test to verify that the basic and critical functionalities work in a new build.
* **Example:** Open the application, login, check main page loads.
* **Analogy:** Like turning on a new car for the first time to see if the engine starts and wheels move — quick sanity check before detailed inspection.

**5️⃣ Sanity Testing**

* **Definition:** Focused testing on specific functionality or bug fixes after minor changes. Ensures changes didn’t break existing functionality.
* **Example:** After fixing the login bug, only test the login functionality, not the whole app.
* **Analogy:** Like tasting only the soup you just added salt to, rather than the entire meal — a quick check on the affected part.

**6️⃣ Change-Related Testing (Regression Testing)**

* **Definition:** Ensures that recent code changes or bug fixes don’t negatively affect existing functionality.
* **Example:** After updating payment module, verify shopping cart, checkout, and login still work.
* **Analogy:** Like checking a car after replacing the brake pads — test brakes, steering, and suspension to ensure nothing else broke.

✅ **Summary Analogy Table:**

| **Test Type** | **Focus** | **Analogy** |
| --- | --- | --- |
| Functional | Does it work? | Car engine starts, brakes work |
| Non-Functional | How well does it work? | Smooth ride, fuel efficiency, comfort |
| White-box | Internal logic | Mechanic inspecting engine/wiring |
| Smoke | Basic functionality check | Turn car on, check wheels move |
| Sanity | Small focused check | Taste only the soup you added salt to |
| Change-related | Ensure changes don’t break | Test car after brake replacement |

**23. Maintenance Testing**

* Triggered by bug fixes, enhancements, or environment changes
* Impact analysis performed before tests
* **Analogy:** Like servicing a car after minor repairs or upgrades

**24. Test Case Terminologies**

* **Test Case:** Step-by-step instructions to test a scenario
* **Test Data:** Inputs used during testing
* **Expected Result:** What should happen
* **Actual Result:** What actually happened
* **Analogy:** Like a recipe (steps) with ingredients (data) and expected dish (result)

**1️⃣ Static Testing**

* **Definition:** Testing **without executing the code**. Focuses on reviewing documents, code, or design to find defects early.
* **Example:** Reviewing code, requirement docs, or design diagrams for mistakes.
* **Analogy:** Like proofreading an essay or checking a car blueprint before building it — you catch errors without actually driving the car.

**2️⃣ Types of Testing Techniques**

**a) Static Testing**

* **Definition:** Reviews, walkthroughs, inspections. No code execution.
* **Analogy:** Proofreading a recipe before cooking.

**b) Dynamic Testing**

* **Definition:** Running the code to find errors.
* **Analogy:** Cooking the dish and tasting it to see if it’s good.

**3️⃣ Differences between Static and Dynamic Testing**

| **Aspect** | **Static Testing** | **Dynamic Testing** | **Analogy** |
| --- | --- | --- | --- |
| Execution | No | Yes | Reading blueprint vs test-driving car |
| Cost | Low | High | Proofreading vs fixing after cooking |
| Timing | Early in SDLC | Later | Before baking vs after baking |
| Tools | Review checklists, code analyzers | Automation tools, test scripts | Red pen vs car test track |

**4️⃣ Static Testing Basics**

* Focuses on **documents, code, and design** to detect errors.
* Techniques include **reviews, walkthroughs, inspections, static analysis tools**.
* **Analogy:** Like checking spelling, grammar, and layout of a book before printing.

**5️⃣ Work Products that Can Be Examined**

* **Examples:**
  + Requirements document
  + Design diagrams
  + Source code
  + Test cases
* **Analogy:** Reviewing blueprints, assembly instructions, or recipe steps before actual work.

**6️⃣ Benefits of Static Testing**

* Detects errors **early** → cheaper to fix.
* Improves **quality of work products**.
* Reduces **later rework**.
* **Analogy:** Catching typos in a manual before it goes to print saves printing cost and complaints.

**7️⃣ Review Process**

* Steps to systematically examine work products:
  1. Planning → Schedule review
  2. Kick-off → Explain objectives
  3. Preparation → Review individually
  4. Review meeting → Discuss issues
  5. Rework → Fix defects
  6. Follow-up → Verify fixes
* **Analogy:** Like a team proofreading a newspaper before printing.

**8️⃣ Work Product Review Process**

* Focused on **specific deliverables**: code, design, documents.
* **Analogy:** Checking a single chapter of a book carefully instead of the whole book.

**9️⃣ Roles in Formal Review**

* **Moderator:** Leads the review → like a teacher guiding a proofreading session.
* **Author:** Person who wrote the document/code → like the student writing the essay.
* **Reviewer/Inspector:** Checks for defects → like classmates marking mistakes.
* **Scribe/Recorder:** Notes all defects → like a secretary noting corrections.

**10️⃣ Review Types**

* **Walkthrough:** Informal step-by-step explanation of document/code.
  + **Analogy:** Reading aloud an essay in class.
* **Technical Review:** Group examines technical content.
  + **Analogy:** Engineers reviewing car design.
* **Inspection:** Formal, rigorous defect detection.
  + **Analogy:** Official editor reviewing a book before publishing.
* **Peer Review:** Colleagues review each other’s work.
  + **Analogy:** Friends proofread your assignment.

**11️⃣ Applying Review Techniques**

* **Tips:**
  + Use **checklists** to avoid missing common mistakes.
  + Focus on **high-risk areas** first.
  + Track **defects found** for improvement.
* **Analogy:** Using a checklist when inspecting a car or home to make sure nothing is overlooked.

✅ **Key Takeaway Analogy:**

* Static Testing = **proofreading or blueprint check** (no execution)
* Dynamic Testing = **actual driving or cooking** (execution needed)

**1️⃣ Categories of Test Techniques**

Test techniques are approaches or methods used to design test cases. They fall into **three main categories**:

1. **Black-box Testing** – Focus on **what the software does**. Ignore internal code.
   * **Analogy:** Like checking if a microwave heats food properly without looking inside.
2. **White-box Testing** – Focus on **how the software works internally** (code, logic, paths).
   * **Analogy:** Like opening a microwave and checking the wiring and circuits.
3. **Experience-based Testing** – Based on tester’s knowledge, intuition, and experience.
   * **Analogy:** Like a chef tasting a dish and guessing which ingredient might be off.

**2️⃣ Choosing Test Techniques**

Factors to consider:

* Type of application
* Risk areas
* Stage of SDLC
* Resources available  
  **Analogy:** Choosing the right tool in a toolbox — hammer for nails, screwdriver for screws.

**3️⃣ Black-box Test Techniques**

**a) Equivalence Partitioning (EP)**

* Divide input data into valid & invalid partitions. Test **one value from each**.
* **Analogy:** If testing shoes sizes 6–10, just test 6, 8, 10 — one from each range.

**b) Boundary Value Analysis (BVA)**

* Test **edges of input ranges** (common defect points).
* **Analogy:** Checking the first and last seats in a theater — defects often happen at boundaries.

**c) Decision Table Testing**

* Test different **combinations of inputs and conditions** using a table.
* **Analogy:** Like a menu table showing all possible toppings combinations for a pizza.

**d) State Transition Testing**

* Test **how software behaves when moving from one state to another**.
* **Analogy:** Elevator: test buttons when elevator is on different floors.

**e) Use Case Testing**

* Test **user scenarios or workflows** end-to-end.
* **Analogy:** Simulating a customer buying a product from login → purchase → payment → logout.

**4️⃣ White-box Test Techniques**

**a) Statement Testing and Coverage**

* Ensure **every line of code is executed at least once**.
* **Analogy:** Reading every line of a recipe to make sure no step is skipped.

**b) Decision Testing and Coverage**

* Ensure **every decision point (if/else) is tested for all outcomes**.
* **Analogy:** Testing both "yes" and "no" answers in a flowchart quiz.

**c) Value of Statement and Decision Testing**

* Helps detect **hidden bugs in logic**.
* **Analogy:** Like double-checking all the wiring in a circuit board to prevent short-circuits.

**5️⃣ Experience-based Test Techniques**

**a) Error Guessing**

* Tester uses experience to **guess where defects may exist**.
* **Analogy:** A seasoned driver checking the most accident-prone curves first.

**b) Exploratory Testing**

* Tester **simultaneously learns, designs, and executes tests**.
* **Analogy:** Exploring a new city without a map — discovering potential problems as you go.

**c) Checklist-based Testing**

* Tester follows a **predefined checklist** to ensure nothing is missed.
* **Analogy:** Pilot pre-flight checklist — ensures all critical steps are covered.

✅ **Key Takeaway Analogy:**

| **Technique Category** | **Focus** | **Analogy** |
| --- | --- | --- |
| Black-box | What software does | Using a microwave without opening it |
| White-box | How software works | Inspecting microwave wiring |
| Experience-based | Tester intuition | Chef tasting a dish |

**1️⃣ Evolution of Requirements**

* Requirements change and evolve over time as the project progresses.
* **Analogy:** Like planning a road trip — initially you plan a route, but you may change it based on weather, traffic, or new destinations discovered along the way.

**2️⃣ Who Provides the Requirements?**

* **Stakeholders:** Customers, end-users, business analysts, managers, developers.
* **Analogy:** Planning a wedding — bride/groom (users), wedding planner (analyst), family (stakeholders) — everyone provides input.

**3️⃣ Challenges in Requirement Gathering**

* Unclear, incomplete, or conflicting requirements.
* Changing needs over time.
* Communication gaps between stakeholders and developers.
* **Analogy:** Asking someone what they want for lunch — they say “something tasty” → too vague!

**4️⃣ Why Do We Need Good Requirements?**

* To build the **right product**, avoid rework, save time and cost.
* **Analogy:** Like building a house — if the blueprint is wrong, the house won’t match expectations.

**5️⃣ Characteristics & Impact of Bad Requirements**

* **Bad Requirements:** Ambiguous, incomplete, contradictory, not testable.
* **Impact:** Project delays, cost overruns, defects, unhappy users.
* **Analogy:** Cooking with a recipe that’s missing ingredients → the dish won’t turn out right.

**6️⃣ Requirement Engineering**

* Systematic process to **gather, analyze, document, and manage requirements**.
* **Analogy:** Architect designing a building: gather needs, draw plans, get approvals, manage changes.

**7️⃣ Functional vs Non-Functional Requirements**

| **Type** | **Meaning** | **Example** | **Analogy** |
| --- | --- | --- | --- |
| Functional | What the system should do | Login, Submit form | Like a car engine: starts and drives |
| Non-Functional | How the system behaves | Performance, Security, Usability | Like a car’s smooth ride, comfort, or safety features |

**8️⃣ Non-Functional Requirements: FURPS+**

* **F**: Functionality
* **U**: Usability
* **R**: Reliability
* **P**: Performance
* **S**: Supportability
* **+**: Security, Scalability, Portability, etc.
* **Analogy:** Features of a hotel room: comfy bed (Usability), power backup (Reliability), fast Wi-Fi (Performance), good service (Supportability).

**9️⃣ Stable and Volatile Requirements**

* **Stable:** Rarely change (core features).
* **Volatile:** Frequently change (nice-to-have, UI tweaks).
* **Analogy:** Foundation of a building = stable, wall color = volatile.

**🔟 Baselining Requirements**

* Freezing requirements at a certain point to avoid uncontrolled changes.
* **Analogy:** Signing a contract for house construction — changes after signing require extra approvals.

**1️⃣1️⃣ Requirements Traceability**

* Linking requirements to **design, code, and tests** to ensure nothing is missed.
* **Analogy:** Tracking luggage at an airport — ensuring each bag reaches the right plane.

**1️⃣2️⃣ Requirement Traceability Matrix (RTM)**

* Table mapping requirements → design → code → test cases.
* **Analogy:** School report card: student (requirement) → subjects (design) → marks (tests).

**1️⃣3️⃣ Maintaining Requirement Traceability**

* Keep the RTM updated with changes.
* **Analogy:** Like updating a family tree when a new member joins.

**1️⃣4️⃣ Requirements Change & Requirement Creep**

* **Change:** Modification to requirement due to new info or need.
* **Creep:** Small, uncontrolled changes accumulating over time.
* **Analogy:** Renovating a house → adding new rooms after construction started → can delay the project.

**15️⃣ Change Management Process**

* Steps to handle requirement changes:
  1. Identify change
  2. Analyze impact
  3. Approve/reject
  4. Implement & update documents
* **Analogy:** Changing a travel itinerary — check costs, feasibility, then update tickets and bookings.

✅ **Key Takeaways Analogy Summary**

| **Concept** | **Analogy** |
| --- | --- |
| Requirement gathering | Asking what someone wants for lunch |
| Good requirements | Correct house blueprint |
| Functional vs Non-functional | Car engine vs comfort/safety |
| Traceability | Tracking luggage at airport |
| Baselining | Signing a contract before building |
| Requirement creep | Adding rooms to a house during construction |

Here’s a simple, analogy-based explanation of **Software Version Guidance**:

**1️⃣ Introduction to Software Versioning**

* Software versioning helps **track changes** in software over time.
* Versions usually have a format like Major.Minor.Revision.Build → 2.3.1.456
* **Analogy:** Like tracking editions of a book: first edition, second edition with minor edits, etc.

**2️⃣ Major Release**

* Significant updates that may include **new features, big changes, or breaking changes**.
* Version example: 1.0 → 2.0
* **Analogy:** Upgrading from a basic phone to a smartphone — major change, new capabilities.

**3️⃣ Minor Release**

* Small updates that **add features or improvements**, but don’t break existing functionality.
* Version example: 2.0 → 2.1
* **Analogy:** Installing a new camera app or widget on your smartphone — enhances experience without changing the core.

**4️⃣ Revision Release (Patch/Update)**

* Fixes **bugs or issues** in existing features.
* Version example: 2.1 → 2.1.1
* **Analogy:** Repairing scratches on a car or updating a recipe to fix a taste issue.

**5️⃣ Build Release**

* Internal version used by developers or testers; **may not be stable**.
* Often automated builds during development.
* **Analogy:** A cake sample from the kitchen — not ready for customers, just for taste testing.

**6️⃣ Beta Version for User Testing**

* Pre-release version shared with users to **get feedback** before official release.
* Version example: 2.2 Beta
* **Analogy:** Movie premiere before official release — audience gives feedback, suggests improvements.

✅ **Quick Analogy Summary Table**

| **Type of Version** | **Purpose** | **Analogy** |
| --- | --- | --- |
| Major | Big changes/new features | Phone → Smartphone |
| Minor | Small features/improvements | Adding apps/widgets |
| Revision | Bug fixes | Repairing scratches on a car |
| Build | Internal/testing version | Cake sample in kitchen |
| Beta | User testing before release | Movie premiere for feedback |