📘 **TESTING COURSE — LEVEL 1**

QUALITY ASSURANCE (QA) — COMPLETE NOTES

---

1. What QA actually means

QA (Quality Assurance) = ownership of product quality.

QA is responsible for answering one question:

"Is this product safe and acceptable to release to real users?"

QA is **not**:

- only testing
- only finding bugs
- only paperwork

QA **owns the outcome**, not just activities.

---

2. QA vs Testing (core differentiation)

Aspect	**QA**	**Testing**
Purpose	Ensure overall quality	Find defects
Nature	Preventive + evaluative	Detective
Timing	Before, during, after dev	Mostly during/after dev
Scope	Whole product	Product behavior
Output	Release confidence	Bugs, evidence

Key truth

Testing ⊂ QA
QA ≠ Testing

Testing is a **tool**.
QA is the **owner of quality decisions**.

---

3. Purpose / objective of QA

QA does **not** aim for perfection.

QA exists to:

- reduce critical risks
- prevent severe failures
- make releases predictable
- avoid surprises after delivery
- protect users, business, and team

One-line objective:

QA makes failures controlled and acceptable.

---

4. When QA is involved (lifecycle)

QA is involved end-to-end.

Before development

- Understand requirements
- Question unclear logic
- Identify risks early

During development

- Guide testing depth
- Validate features incrementally
- Catch issues early (cheap fixes)

After development

- Execute testing
- Evaluate risks
- Decide release readiness

If QA appears only at the end → **quality is already compromised**.

---

5. Is QA thinking or practical work?

Both — but thinking comes first.

QA thinking includes:

- risk analysis
- assumption checking
- impact evaluation
- “what can go wrong?” mindset

QA practical work includes:

- reviewing features
- testing
- validating fixes
- release sign-off

Without thinking → execution becomes random.

---

6. How QA thinking improves

QA thinking improves by:

- asking better questions
- analyzing failures
- learning from escaped bugs

Key questions QA asks repeatedly:

- What assumption exists here?
- What if this fails?
- Who is affected?
- Is failure visible or silent?
- Is recovery possible?

QA thinking grows from **experience with failures**, not theory alone.

---

7. Work & responsibilities of QA

Real QA responsibilities:

- Understand product purpose
- Identify risky areas
- Decide testing depth
- Ensure critical flows are safe
- Guide testers
- Validate bug fixes
- Define “done”
- Recommend release readiness
- Communicate risks clearly

QA supports **decisions**, not just defect reporting.

---

8. Negative QA practices (what to avoid)

Bad QA behaviors:

- Blind checklist execution
- Measuring success by bug count
- Blocking releases without reasoning
- Ignoring business context
- Over-documenting, under-thinking
- “Not my responsibility” mindset

Worst mindset:

“I reported the bug, my job is done.”

That is **not QA ownership**.

---

9. QA vs Tester (role clarity)

- **Tester** → executes tests, finds & validates defects
- **QA** → owns quality, uses testing evidence to decide readiness

Tester provides **data**.
QA provides **judgment**.

---

10. QA vs Project Manager / Team Lead

QA is **NOT**:

- Project Manager (timeline, coordination)
- Team Lead (people management)

QA authority is over:

- **quality**
- **risk**
- **release readiness**

Sometimes one person wears multiple hats — but roles are conceptually different.

---

11. Release decision truth

- QA **recommends** release readiness
- Business **decides** release

In healthy teams:

- QA approval is a strong gate

In unhealthy teams:

- QA is overridden (often leads to failures)

QA is a **risk authority**, not a dictator.

---

12. One-line mental model (lock this)

QA owns the question: “Is this safe to release?”
Testing provides the evidence.

LINK : https://chatgpt.com/share/694e7e12-f1dc-8001-a0a6-6824dfb5f06f

📘 **TESTING COURSE — LEVEL 2**

TESTING STRATEGY (COMPLETE NOTES)

---

1. What is Testing Strategy?

Testing Strategy = a high-level plan that decides how testing will be done.

In simple terms:

Testing strategy is a roadmap for testing.

It defines:

- where testing starts
- what areas to test
- how deep to test
- what to skip
- when to stop testing

Without a strategy:

- testing becomes random
- effort is wasted
- confidence is low

---

2. Purpose of Testing Strategy

Testing strategy exists because:

- time is limited
- people are limited
- money is limited

So strategy helps to:

- focus on high-risk areas
- avoid over-testing low-impact areas
- avoid under-testing critical areas
- make testing effective, not exhaustive

Key truth

You cannot test everything.
Strategy decides what is “enough”.

---

3. Testing Strategy as a Roadmap (Your Understanding)

Your understanding is **correct**.

Testing strategy acts like:

- navigation for testers
- guidance on order, scope, and depth
- alignment for the whole team

It answers:

- From where do we start?
- What must be tested first?
- What is optional?
- Where do we stop?

---

4. Risk-Based Testing (Core of Strategy)

Testing strategy is **risk-based**, not feature-based.

Risk definition

Risk = Probability of failure × Impact of failure

This is a **thinking tool**, not a math formula.

---

5. How to Identify Risk Areas

High-risk areas (test deeply)

- Core business functionality
- Features used by most users
- Authentication, authorization, data handling
- New or recently changed code
- Complex logic
- Areas with past defects

Medium-risk areas

- Supporting features
- Moderately complex logic
- Features with limited users

Low-risk areas

- Static pages
- Cosmetic UI elements
- Rarely used features
- Well-tested third-party components

---

6. Probability of Failure (How likely it is to break)

Probability is judged qualitatively:

- **High**: new, complex, frequently changed features
- **Medium**: stable but moderately complex features
- **Low**: simple, stable, unchanged features

No numbers are calculated.
Only **experience-based judgment**.

---

7. Impact of Failure (How bad it is if it breaks)

Impact is also judged qualitatively:

- **High**: data loss, security issues, system unusable
- **Medium**: partial feature failure, user inconvenience
- **Low**: cosmetic issues, minor UI problems

---

8. How to React to Risk Combinations

Probability	Impact	Action
High	High	Test very deeply
High	Low	Test reasonably
Low	High	Test carefully
Low	Low	Minimal testing

This is where **testing effort is allocated**.

---

9. Over-testing vs Under-testing

Over-testing signs

- Re-testing simple features repeatedly
- Focusing too much on cosmetic issues
- Spending too much time on low-impact areas

Under-testing signs

- Skipping core workflows
- No negative testing
- No role/permission testing
- Ignoring data validation

Testing strategy balances both.

---

10. Entry Criteria (When Testing Starts)

Testing should start only when:

- feature is implemented
- application runs
- environment is ready
- basic smoke test passes

If entry criteria are not met:

- testing is inefficient
- bugs are misleading

---

11. Exit Criteria (When Testing Stops)

Testing can stop when:

- critical test cases pass
- no blocking issues remain
- known issues are documented
- risk is acceptable
- release decision can be made

Exit criteria prevent:

- endless testing
- uncertainty
- last-minute panic

---

12. Manual vs Automation Testing (Strategy View)

This decision is made **at strategy level**, not execution level.

Manual testing is preferred when:

- feature is new
- UI/UX judgment is needed
- logic is changing
- exploratory testing is required

Automation testing is preferred when:

- feature is stable
- tests are repetitive
- regression is needed

- speed and consistency matter

Golden rule

Manual testing comes first.
Automation follows stability.

Manual → stabilize → automate

---

13. Using Manual and Automation Together

Most real projects use **both**.

Correct order:

1. Manual testing (discover bugs)
2. Stabilize feature
3. Automation testing (prevent regressions)

Never automate unstable features.

---

14. How Manual and Automation Tests Are Done (High-Level)

Manual testing uses:

- human thinking
- application UI
- test scenarios
- observation and judgment

Automation testing uses:

- scripts
- tools (later)
- code that simulates user behavior

At Level 2:

- **thinking > tools**
- tools come later

---

15. Why Decide "What Not to Test"?

Purpose:

- save time
- save money
- save people's energy
- focus on what truly matters

Not testing something is a **conscious decision**, not negligence.

---

16. Who Creates Testing Strategy?

Depending on company size:

- QA
- QA Lead
- Senior Tester
- Sometimes PM or Tech Lead (small teams)

Even if undocumented, **strategy always exists**.

---

17. How Testing Strategy Is Created

It is created using:

- product understanding
- risk analysis
- past experience
- constraints (time, people, budget)

It is **designed**, not copied from templates.

---

18. Key Takeaways (Write These)

- Testing strategy controls testing effort
- Strategy is risk-based
- Not everything should be tested
- Manual vs automation is a strategic decision
- Entry & exit criteria define boundaries
- Strategy answers “what is enough testing”

Cosmetic issues = problems that affect only appearance, not behavior or functionality.

Aspect	**Authentication**	**Authorization**
Purpose	Verify identity	Grant permissions
Happens when	First	After authentication
Answers	“Who are you?”	“What can you do?”
Based on	Credentials	Roles, policies, rules
Example	Login	Access control

LINK: https://chatgpt.com/share/694e7e12-f1dc-8001-a0a6-6824dfb5f06f

📘 **TESTING COURSE — LEVEL 3**

MANUAL TESTING vs AUTOMATION TESTING

(COMPLETE & FINAL NOTES)

---

1. What is MANUAL TESTING?

Manual testing = testing performed by a human using thinking, judgment, and observation.

It means:

- running the application (local / staging / test env)
- acting like a real user
- validating behavior
- finding bugs
- exploring unknown scenarios

Manual testing is **not just clicking**.
It is **thinking-driven testing**.

---

2. Objective of Manual Testing

The main objectives are:

- discover new bugs
- understand system behavior
- test unstable or changing features
- validate logic & workflows
- test negative and edge cases
- judge UI/UX and usability

Key truth

Manual testing is for discovery, understanding, and exploration.

---

3. What Manual Testing is BEST for

Manual testing is mandatory when:

- features are new
- requirements are unclear
- logic changes frequently
- UI/UX judgment is needed
- exploratory testing is required
- failures are unknown

---

4. Exploratory Testing (Important)

Definition:
Exploratory testing is a testing approach where test design, execution, and learning happen at the same time, without predefined test cases.

Objective:
To discover unknown defects and risks by intentionally exploring the system.

Nature:

- Not random
- Intention-driven
- Tester-thinking based

How it is done:

- Start with a clear test intention (charter)
- Explore different flows, edge cases, and user behaviors
- Adapt testing based on observations

Focus:

- Unexpected scenarios
- Edge cases
- Real user behavior

Key line to remember:

- Scripted testing verifies, exploratory testing discovers.
- it is a part of the manual testing
- it comes under manual testing

📌 Exploratory testing is:

- 100% manual
- impossible to automate

---

5. Correct relationship between Manual Testing and UI

- Manual testing ≠ only UI testing
- Manual testing = testing done by a human

UI is just one interface used during manual testing

---

6. What is AUTOMATION TESTING?

Automation testing = testing performed by code/scripts that execute predefined steps automatically.

Automation:

- simulates user actions

- checks expected results
- returns pass/fail

Automation is **execution**, not thinking.

---

7. What is meant by "Simulation"?

Simulation = mimicking user actions using code.

Examples:

- entering username/password via script
- clicking buttons via script
- submitting forms automatically
- validating outputs automatically

The system behaves **as if a user acted**, but no human is involved.

Simulation is the process of creating a controlled imitation of a real-world system that models how the system works and behaves, often displaying those actions visually, in order to study, test, or predict outcomes without using the real system.

---

8. What Automation Testing is GOOD at

Automation is best for:

- regression testing
- repetitive tests
- stable features
- frequent re-testing
- speed and consistency

Automation answers:

"Did this already-working behavior break again?"

---

9. What Automation Testing is NOT Good at

Automation is bad at:

- discovering new bugs
- exploratory testing
- UI/UX judgment
- unstable features
- changing requirements
- cosmetic checks

Trying to automate these causes **waste and maintenance problems**.

---

10. Discovery vs Verification (Core Difference)

- **Manual testing** → discovery
- **Automation testing** → verification

Key lock-in

Automation is for reconfirmation of known things, not for discovering new things.

---

11. Does Automation Only Test What Is Coded?

YES.

Automation tests:

- only what is written in code
- only predefined steps
- only expected outcomes

If something is **not coded**, it is **not tested**.

---

12. Expected Outcomes in Automation

Automation:

- checks expected behavior
- compares actual vs expected
- returns pass/fail

It does **not**:

- think
- explore
- adapt

---

13. Is Automation Done for the Whole Product?

NO. Never.

Automation is done for:

- selected parts only
- critical and stable workflows

Trying to automate the entire product is a **mistake**.

---

14. Manual → Automation Flow (Correct Order)

Correct industry flow:

1. Manual testing (first time)
2. Bugs found & fixed
3. Feature becomes stable
4. Automation added
5. Automation re-tests repeatedly

Golden rule

Manual → Stabilize → Automate

---

15. Regression Scenario (Your Understanding — Correct)

When:

- bugs are fixed
- code changes are made

Then:

- unchanged stable parts → automation
- changed or new parts → manual testing

This reduces:

- time
- effort
- human repetition

---

16. Why Automation Prevents Regression

Automation:

- runs the same tests every time
- immediately detects behavior changes
- “locks” stable behavior

So:

- old bugs don’t return silently
- regressions are caught early

---

17. What Should NEVER Be Automated

Do NOT automate:

- unstable features
- unclear flows
- frequently changing UI

- cosmetic checks
- exploratory testing
- one-time features
- vague requirements

Automation **can** test them technically,
but **should not** test them practically.

---

18. What Should Be Automated First

Good automation candidates:

- login flows
- core stable workflows
- critical regression paths
- repetitive validations

Automation is used **instead of repeated manual testing**, not instead of thinking.

---

19. Manual vs Automation — Core Comparison

Aspect	**Manual Testing**	**Automation Testing**
Thinking	Yes	No
Discovers new bugs	Yes	No
Repetition	Slow	Fast
Handles change	Good	Poor
Scope	Broad	Limited
Purpose	Discovery	Verification

They **complement**, not replace, each other.

---

20. Statement Evaluation (Final Corrected View)

Statement 1

Corrected version:

Automation testing is used to repeatedly test the **stable behavior** that was first tested and understood through manual testing.

✅ Correct with clarification.

Statement 2

What automation cannot do is done via manual testing.

✅ Correct.

Statement 3

At the execution-method level, only manual and automation testing exist.

✅ Correct.

All other terms are **testing intents**, not execution types.

---

21. Should You Learn Automation Tools Now?

NO. Clear NO.

Right now:

- focus on thinking
- focus on manual testing skill
- focus on strategy & testing types

Tools come **after Level 4 or 5**.

Learning tools early:

- creates false confidence
- weak fundamentals
- shallow testing ability

---

22. Final Lock-in Summary (Write This)

- Manual testing = discovery & thinking
- Automation testing = verification & repetition
- Automation does not think
- Automation tests only what is coded
- Manual comes before automation
- Automation is added only after stability
- Both are required, not competitors
- Manual testing is to test the parctical scenarios
- Automation testing is to test the technical scenarios

LINK: https://chatgpt.com/share/694f6f6a-4430-8001-b798-e5dc74ad8cbd

📘 **TESTING COURSE — LEVEL 4**

TESTING TYPES & INTENT

(WHY we test)

---

1. What Level 4 is about

Level 4 answers one question:

WHY am I testing right now?

This level is **not** about:

- tools
- manual vs automation
- execution steps

It is about **INTENT / PURPOSE**.

---

2. Two dimensions of testing (very important)

Testing always has **two independent dimensions**:

Dimension 1 — HOW we test

- Manual testing
- Automation testing

Dimension 2 — WHY we test

- Functional
- Exploratory
- Regression
- Smoke
- Sanity
- Negative
- Security
- Performance

👉 These two dimensions combine.

That is why we write things like:

- **Manual + Functional**
- **Manual + Exploratory**
- **Automation + Regression**

Meaning of the "+" symbol

- **Left side** → HOW the test is executed
- **Right side** → WHY the test is executed

There is **no mixing of concepts**.

---

3. Smoke Testing

Purpose

Check whether the application is stable enough to start testing.

What it answers

- Does the app open?
- Does it run on local / test server?
- Can we log in?
- Can core screen load?

Characteristics

- Very shallow
- Very quick
- Done first
- Mostly manual

If smoke fails → **STOP testing**.

---

4. Sanity Testing

Purpose

Quick confidence check after a small change or bug fix.

What it checks

- Fixed bug works
- Closely related areas are not broken

Characteristics

- Narrow scope
- Fast
- Focused
- Not full testing

Smoke ≠ Sanity
Smoke = "Can we test?"
Sanity = "Does the fix make sense?"

---

5. Functional Testing

Purpose

Verify that the system behaves as per requirements.

What is tested

- Buttons
- Forms
- Workflows
- Business logic
- Expected behavior

One-line definition

Functional testing checks what the product is supposed to do.

---

6. Negative Testing

Purpose

Check how the system behaves when users do wrong or unexpected things.

Examples

- Wrong login credentials
- Empty mandatory fields
- Invalid formats
- Skipping steps

Key idea

Negative testing validates behavior outside normal usage.

---

7. Exploratory Testing

Purpose

Discover unknown issues by testing while learning the system.

Why “learning” is always mentioned

Because:

- tester does not fully know system behavior beforehand
- understanding grows during testing
- next test depends on what was just learned

Characteristics

- No fixed scripts
- Intuition-driven

- Adaptive
- 100% manual
- Cannot be automated

If there is no learning → it is **not exploratory testing**.

---

8. Regression Testing

Purpose

Ensure that previously working behavior still works after changes.

What it answers

- Did something that worked earlier break now?

Characteristics

- Re-testing known behavior
- Done after bug fixes or new features
- Often automated after stability

Core idea

Regression testing protects the past.

---

9. Security Testing (basic intent)

Purpose

Ensure obvious security rules are enforced.

What it covers (at this level)

- Authentication (who you are)
- Authorization (what you can do)
- Role-based access
- No sensitive data exposure

This is **logic-based**, not hacking.

---

10. Performance Testing (basic awareness)

Purpose

Check system behavior under load or stress.

At your current stage:

- knowing *what it is*
- knowing *why it matters*
- knowing *when it is needed*

is **enough**.

Deep performance testing is a **specialization**, not a foundation.

---

11. Same Feature Tested with Different Intent (Important Concept)

A single feature can be tested with multiple intents.

Example: Login Feature

- **Functional** → correct credentials allow login
- **Negative** → wrong credentials show error
- **Exploratory** → try refresh, back, rapid attempts
- **Regression** → login still works after changes
- **Security** → user cannot access unauthorized pages

👉 Same feature, **different intent lenses**.

---

12. Manual / Automation vs Testing Types (Final Clarity)

- **Manual / Automation** → HOW the test is executed
- **Testing types** → WHY the test is executed

That's why combinations exist and are correct.

---

13. Key Takeaways (Write These)

- Testing types define **why** we test
- Execution method defines **how** we test
- Same test can serve multiple intents
- Smoke comes first, regression comes often
- Exploratory testing always involves learning
- Security & performance are awareness-level here

LINK: https://chatgpt.com/share/694e7e12-f1dc-8001-a0a6-6824dfb5f06f

📘 **TESTING COURSE — LEVEL 5**

TESTING LEVELS (WHERE testing happens)

---

1. What "Testing Levels" actually mean

Testing levels classify testing based on HOW MUCH of the system is under test at once.

They answer:

"At what depth / scope am I testing the system?"

Levels are based on **system size under test**, not on testing intent or execution method.

---

2. The Four Core Testing Levels

1. **Unit Testing**
2. **Integration Testing**
3. **System Testing**
4. **User Acceptance Testing (UAT)**

These apply to the **same product**, but at **different scopes**.

---

3. Are these levels independent or linear?

Conceptually

- They are **independent layers**
- Each has a distinct purpose

Practically

- They are followed in a **logical order**
- Bugs can create **loops back to earlier levels**

Typical real flow

Unit → Integration → System → UAT

↑ ↓

Fixes ←————

So your understanding is correct:
✓ **Logically linear**
✓ **Practically iterative**

---

4. UNIT TESTING (deep clarity)

What it is

Unit testing = testing individual code units (functions, methods, classes) in isolation.

Scope:

- Smallest pieces of logic
- No UI
- No full system

Who does unit testing?

✓ **Developers only**

Testers / QA:
❌ do not write unit tests
❌ do not execute unit tests

When is unit testing done?

- During development
- While writing code
- Before integration/system testing

How unit tests are written

- Developers write **test code**
- Each test checks:
    - input
    - logic
    - expected output
- Tests run automatically

Where does unit test code live?

✓ In the **same source code repository**
✓ In **separate files/folders**

Typical structure:

project/

├– src/ → product code

├– tests/ → unit test code

Is unit test code removed later?

❌ **NO. Never.**

✓ Unit test code:

- stays permanently
- is not deleted
- protects future changes
- is not part of runtime production execution

Is unit test code delivered to client?

✓ Often yes (as part of repository)
✓ Sometimes excluded from production build

But:

- tests are **never deleted**
- tests are **not exposed to users**

Do developers document unit tests separately?

❌ **NO (in real industry).**

Reality:

- unit tests **are code**
- code = proof
- no manual test-case documents

Sometimes only:

- coverage %
- pass/fail status (CI)

Role of tester in unit testing (important)

Testers:

- do NOT write unit tests
- do NOT execute unit tests

Testers **care indirectly** by:

- asking developers if unit tests exist
- noticing repeated basic bugs
- using unit testing quality as a **signal**

Strong unit tests → fewer silly bugs
Weak unit tests → many basic bugs for testers

Bugs that DO need a unit test ✅

Add a unit test when the bug is:

- Business logic error
- Validation logic mistake
- Permission / role issue
- Security-related behavior
- Conditional or edge-case failure
- Previously fixed bug that might reappear

Your login validation bug is a perfect example.

Bugs that DO NOT need a unit test ❌

Do **not** add a unit test when the bug is:

- UI alignment issue
- CSS / styling problem
- Copy/text change
- JavaScript animation glitch
- Browser-specific layout issue
- Third-party service outage
- Data issue fixed by migration only

Unit tests cannot protect these meaningfully.

Bug fixed

↓

Is it logic / behavior?

↓

Yes → Add a unit test

No → Do not add a unit test

---

5. INTEGRATION TESTING (deep clarity)

What it is

Integration testing = testing interaction between components.

Examples:

- frontend ↔ backend
- API ↔ database
- service ↔ service

- third-party integrations

Who does integration testing?

- Mainly developers
- Sometimes testers (manual / limited automation)

When is integration testing done?

- After unit tests pass
- During development
- Before or alongside system testing

How integration testing is conducted (tester perspective)

Testers:

- validate data flow
- check responses between systems
- verify component communication
- catch mismatches

Testers do NOT:

- write deep integration frameworks
- mock services at code level

Conceptual understanding is enough at this stage.

Tools (awareness only)

- API testing tools
- logs
- test environments

(No need to learn tools now.)

---

6. SYSTEM TESTING (core tester responsibility)

What it is

System testing = testing the complete application as a whole.

This is the **main testing phase**.

Who does system testing?

✓ **Testers / QA**

When is it done?

- After development builds are ready
- In testing environment
- Before UAT

What happens in system testing?

This is where most of your previous levels apply:

- Functional testing
- Negative testing
- Exploratory testing
- Regression testing
- Smoke & sanity
- Basic security & performance checks

System testing validates:

“Does the whole system work end-to-end?”

---

7. USER ACCEPTANCE TESTING (UAT)

What it is

UAT = testing to confirm the product is acceptable for real business use.

Who does UAT?

✓ **Client / business users / end users**

QA/testers:
❌ do not own acceptance

When is UAT done?

- After system testing
- When product is stable
- Before production release

How UAT is done (real example)

Example: College management system

- Admin creates departments
- Staff raises tickets
- Reports are generated
- Permissions validated

Users ask:

“Can we actually run our organization using this?”

No scripts.
No automation.
Just real workflows.

QA/Testers’ role in UAT

QA/testers:

- prepare UAT scenarios
- guide users
- clarify expected behavior
- reproduce issues
- convert feedback into bugs

QA acts as a **bridge**, not decision-maker.

---

8. Relationship between all levels (big picture)

- All 4 levels apply to the **same product**
- Scope increases at each level
- Earlier levels catch bugs cheaper
- Later levels validate real usage

Weak earlier levels → pain later.

---

9. Final Responsibility Map (LOCK THIS)

Level	**Who owns it**	**Tester role**
Unit	Developer	Awareness only
Integration	Developer / Tester	Partial
System	Tester / QA	Full ownership
UAT	Client	Support only

---

10. Key truths you clarified (important)

- Strong unit testing → fewer bugs later
- Weak unit testing → many basic bugs
- Testers confirm unit testing via communication & signals
- Unit tests stay in codebase permanently
- Testers do not write unit test code
- Integration testing overlaps roles
- System testing is tester's core
- UAT is business acceptance

---

11. Level 5 — FINAL TAKEAWAYS

Write these exactly:

- Testing levels define **where testing happens**
- Levels are based on **scope size**
- Unit & integration happen during development
- System testing is QA's main responsibility
- UAT is client-owned
- Testers care about unit tests indirectly
- Unit test code stays in the repository
- Levels are layered and interconnected

LINK: https://chatgpt.com/share/694e7e12-f1dc-8001-a0a6-6824dfb5f06f

📘 **TESTING COURSE — LEVEL 6**

EXECUTION & PROFESSIONAL PRACTICE

(HOW testing is done in real IT projects)

---

1. What Level 6 Covers

Level 6 explains:

- how testing runs day-to-day
- how testers work practically

- how bugs are handled
- how communication happens
- how decisions are made

This is **execution-level knowledge**, not theory.

---

2. End-to-End Testing Workflow (Real Life)

Typical real project flow:

1. Feature / build is ready
2. Build is given to testing
3. Smoke testing
4. Test execution (system testing)
5. Bugs identified & logged
6. Developers fix bugs
7. Re-testing
8. Regression testing
9. Release readiness
10. UAT
11. Production release

This cycle **repeats multiple times**.

---

3. What a Tester Does Practically

Daily tester activities:

- understand features
- execute test scenarios
- perform exploratory testing
- identify defects
- log bugs
- re-test fixes
- perform regression
- communicate risks & status

Testing is a **continuous cycle**, not a one-time action.

---

4. Test Scenario vs Test Case (IMPORTANT)

Test Scenario

- High-level idea of **what to test**
- Focuses on **coverage**

Example:

“Verify user login works correctly”

Test Case

- Detailed steps + expected result
- Focuses on **execution**

Example:

1. Enter valid username
2. Enter valid password
3. Click Login
   Expected: User logged in successfully

Key difference

- 👉 **Scenarios = what to test**
- 👉 **Test cases = how to test**

Good testers **think in scenarios**, not just steps.

---

5. When a Bug Is Found — Stop or Continue?

Critical / Blocker Bug

Examples:

- app crashes
- login not working at all
- system unusable

Action:

- log bug immediately
- inform team
- stop testing dependent areas

Normal / Minor Bug

Examples:

- validation missing
- UI issue

- single feature issue

Action:

- log bug
- continue testing other areas

Rule:

Critical bug → stop & escalate
Normal bug → log & continue

---

6. Bug / Defect Lifecycle (VERY IMPORTANT)

Every bug follows a lifecycle:

1. **New** – Tester logs the bug
2. **Assigned** – Given to developer
3. **Open / In Progress** – Developer works on it
4. **Fixed** – Developer claims fix done
5. **Re-test** – Tester verifies fix
6. **Closed** – Bug resolved
7. **Reopened** – If fix fails

This lifecycle ensures **traceability and accountability**.

---

7. Bug Report — Core Concept

Correct definition

A bug report is a structured, reproducible record of a defect that allows developers to understand, reproduce, and fix the issue without confusion.

Bug reports are **not casual messages**.

---

8. Basic Structure of a Bug Report (Tool-Independent)

Every good bug report contains **7 core parts**:

1. Title / Summary

- short
- clear
- problem-focused

Example:

“Login fails with valid credentials”

---

2. Environment

Where the bug occurred:

- OS
- Browser
- App version
- Test environment

Why?
Because bugs can be **environment-specific**.

---

3. Preconditions

What must already be true:

Example:

- User account exists
- User is logged out

---

4. Steps to Reproduce

- step-by-step actions
- anyone should reproduce it

Example:

1. Open login page
2. Enter valid username
3. Enter valid password
4. Click Login

---

5. Actual Result

What **actually happened**

Example:

“Error message shown and login fails”

---

6. Expected Result

What **should have happened**

Example:

“User should be logged in successfully”

---

7. Severity & Priority

- Impact (severity)
- Urgency (priority)

Optional but Helpful

- Screenshot / video
- Logs
- Notes

---

9. One Bug = One Bug Report

Industry rule:

1 bug = 1 report

Never combine multiple bugs into one report.

---

10. Severity vs Priority (CLEAR DIFFERENCE)

Severity = Impact

“How badly does this bug affect the system?”

Typical scale:

- **Blocker** – system unusable
- **Critical** – core function broken
- **Major** – important feature broken
- **Minor** – small issue
- **Cosmetic** – appearance only

Severity is usually decided by **tester / QA**.

Priority = Urgency

“How soon must this bug be fixed?”

Typical scale:

- **High** – must fix immediately
- **Medium** – fix soon
- **Low** – can wait

Priority is often decided by **QA + PM + business**.

Examples

- Login button color wrong
  → Severity: Low
  → Priority: High (branding)
- Rare admin report failure
  → Severity: High
  → Priority: Low (used once a month)

---

11. Error vs Bug vs Failure (LOCK THIS)

Error

- Human mistake (developer misunderstanding or wrong logic)

Bug / Defect

- Observable problem in software caused by error

Failure

- User experiences the bug in real usage

Chain (remember this):

Error (human mistake)

↓

Bug / Defect (software issue)

↓

Failure (user impact)

---

12. Logs — Practical Understanding

Common Log Types

- **Application logs** – internal app behavior
- **Error logs** – exceptions & failures
- **Server logs** – request/response activity
- **Database logs** – DB operations/issues
- **Security logs** – login attempts, access violations

What matters for a tester

- You **do not create logs**
- You **may attach logs** if available
- You use logs to **support a bug**, not debug deeply

---

13. Attaching Logs vs Screenshots

- **Screenshot / video** → what the user sees

- **Logs** → what the system records internally

Both are **evidence**, but from different angles.

---

14. Where Testers See Logs (Reality)

- Usually developers check logs
- Sometimes testers:
    - see browser console logs
    - get logs from developers
    - attach auto-generated logs

Reading logs is not mandatory for testers.

---

15. Bug Tracking / Tracking System

What it is

A centralized online system to record, track, and manage bugs.

Bug reports are written in **tracking tools**, not:

- Word documents
- Excel sheets
- Emails

Tools change, **concept stays the same**.

---

16. Communication — Silent Skill

A professional tester must:

- communicate clearly
- avoid blame language
- explain risk, not emotions
- be precise, not dramatic

Good QA = good communicator.

---

17. What a Tester Is (Professional Identity)

A tester is:

- **risk identifier**
- **quality advocate**
- **user representative**
- **system thinker**

Not just a button clicker.

---

18. Key Level 6 Takeaways (WRITE THESE)

- Testing is iterative, not linear
- Scenarios define coverage; test cases define execution
- Critical bugs stop testing; normal bugs don't
- Bug reports are structured communication
- 1 bug = 1 report
- Severity ≠ Priority
- Error ≠ Bug ≠ Failure
- Logs support bugs; screenshots show symptoms
- Communication is part of QA work

LINK: https://chatgpt.com/share/694e7e12-f1dc-8001-a0a6-6824dfb5f06f

MASTER PROMPT (TO CONTINUE THE JOURNEY) :

I have already completed a full, structured learning journey in SOFTWARE TESTING / QA from beginner to execution level**.**

What I have done so far:

- Learned testing as a structured domain in 6 levels, end-to-end.

- Level 1: QA fundamentals – quality ownership, QA vs tester, responsibility boundaries.

- Level 2: Testing strategy – risk-based thinking, scope, entry/exit criteria, what to test vs what not to test.

- Level 3: Manual testing vs automation testing – purpose, differences, when to use which, stability concept, regression locking.

- Level 4: Testing types & intent – smoke, sanity, functional, negative, exploratory, regression, security (basic), performance (basic).

- Level 5: Testing levels – unit, integration, system, UAT; who owns each level; tester's indirect role in unit & integration testing; unit test code staying in repo but not runtime.

- Level 6: Execution & professional practice – real IT workflow, test scenarios vs test cases, bug lifecycle, bug reporting structure, severity vs priority, error vs bug vs failure, logs vs screenshots, bug tracking systems (conceptual), tester communication & mindset.

Depth achieved:

- I do NOT have only surface-level definitions.

- I clarified edge cases, role boundaries, real industry practices, misconceptions.

- I understand WHY things are done, not just WHAT they are.

- I converted all learning into structured written notes and saved them.

- I intentionally did NOT learn tools yet (JIRA, automation tools, etc.) to avoid premature complexity.

What I have NOT done yet:

- I have NOT practiced on a live system.

- I have NOT written real test cases.

- I have NOT logged real bug reports.

- I have NOT executed a full testing cycle.

- I have NOT used testing tools yet.

Current status:

- I am at the transition point between THEORY → APPLICATION.

- My foundation is complete and industry-correct.

- I am ready to apply this knowledge practically.

- I want guidance that assumes all the above knowledge is already present.

- I do NOT want repeated explanations of basics again.

How I want to continue:

- Continue from this exact point forward.

- Focus on APPLICATION, SIMULATION, and REAL PRACTICE.

- Help me convert knowledge into behavior.

- Guide me step-by-step like a real QA mentor.

- Avoid re-teaching completed levels unless I explicitly ask.

Important constraints:

- Keep explanations clear, structured, and no sugarcoating.

- Do not overload with tools unless necessary.

- Prioritize practical execution over theory.

ADDITIONAL TAKEAWAYS:

STEP 1: Freeze Your Role (Mindset Shift)

Write this down:

“From now on, I am acting as the QA / Tester for this project.”

This means:

- you are NOT fixing code
- you are NOT assuming correctness

- you are validating risk & quality

This mindset shift is **mandatory**.

STEP 2: Create a VERY SIMPLE testing skeleton (today)

Before touching the application, create **only this** (no heavy docs):

A. Feature list (high-level)

Example:

- Login
- User roles
- Ticket creation
- Ticket status update
- Admin actions
- Notifications
- Reports

👉 Just list features. No testing yet.

Best Practice

- You **can write in any order**
- You **should start from core features** (recommended)

Example Structure

- Authentication
- Core business workflows
- Supporting features
- Admin features
- Reports
- Settings

Best Practice

- You **can write in any order**
- You **should start from core features** (recommended)

Example Structure

- Authentication
- Core business workflows
- Supporting features
- Admin features
- Reports
- Settings

STEP 2B: Testing Intent Awareness

For each feature, mentally note (do not execute yet):

- Functional
- Exploratory
- Negative
- Regression (later)

This step ensures **coverage awareness**, not execution.

STEP 3: Decide FEATURE TESTING ORDER

Feature Testing Order Rule

Test features in the order of dependency and business criticality.

Correct Feature Order

1. Entry / Access features (login, auth)
2. Core business features (why product exists)
3. Supporting / dependent features
4. Admin / control features
5. Reports & analytics
6. Settings / low-risk features

Rule:

Test what blocks everything else first.

STEP 4: Decide ROLE TESTING ORDER (If Multiple Roles Exist)

Role Testing Rule

Test roles from lowest privilege → highest privilege.

Correct Role Order

1. Lowest-privilege / primary user
   (creates data, uses system most)
2. Other operational users
3. Admin users
4. Super-admin (if any)

Rule:

Always test the role that CREATES data before the role that CONTROLS data.

STEP 5: Decide TESTING TYPE ORDER (Execution Order)

For **each role + feature**, follow this order:

1. Smoke testing
   → “Is it testable at all?”

2. Core functional testing (happy paths)
   → “Does it basically work?”
3. Exploratory testing
   → “What breaks when humans use it?”
4. Negative testing
   → “How does it fail?”
5. Regression testing (after fixes)

Never mix all at once.

LINK: https://chatgpt.com/share/694e7e12-f1dc-8001-a0a6-6824dfb5f06f

✅ **CORRECT ORDER IN WHICH BUGS ARE FIXED**

Bugs are NOT fixed in the order they are found.

They are fixed based on **impact, urgency, and dependency**.

The real fixing order is:

1️⃣ Blocker bugs

2️⃣ Critical bugs

3️⃣ High-priority Major bugs

4️⃣ Normal / Medium bugs

5️⃣ Low-priority / Cosmetic bugs