Introduction to Software Testing Fundamentals

- Software testing discovers defects and evaluates quality, not just running the app
- Verification: Are we building the product right? Validation: Are we building the right product?
- Bugs found in production can cost 100x more than those caught during requirements
- Testing reduces risk, it does not prove perfection

Key Testing Principles Every Tester Should Know

01 

- Testing shows defect presence, not absence—passing tests don't prove bug-free software

02 

- Exhaustive testing is impossible: use risk-based focus and test design techniques

03 

- Early testing saves time and money—shift left to requirements reviews

04 

- Defects cluster: ~80% of bugs live in ~20% of modules (Pareto principle)

05 

- Pesticide paradox: stale tests stop finding new bugs; update suites regularly

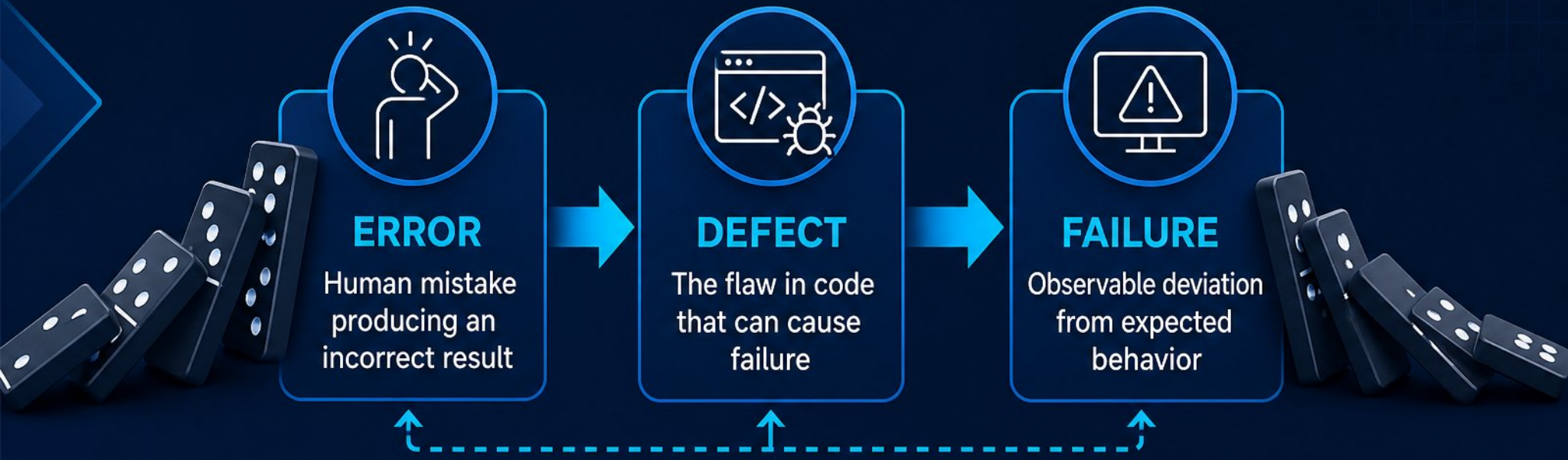
06 

- Testing is context-dependent: a banking app differs from a mobile game

07 

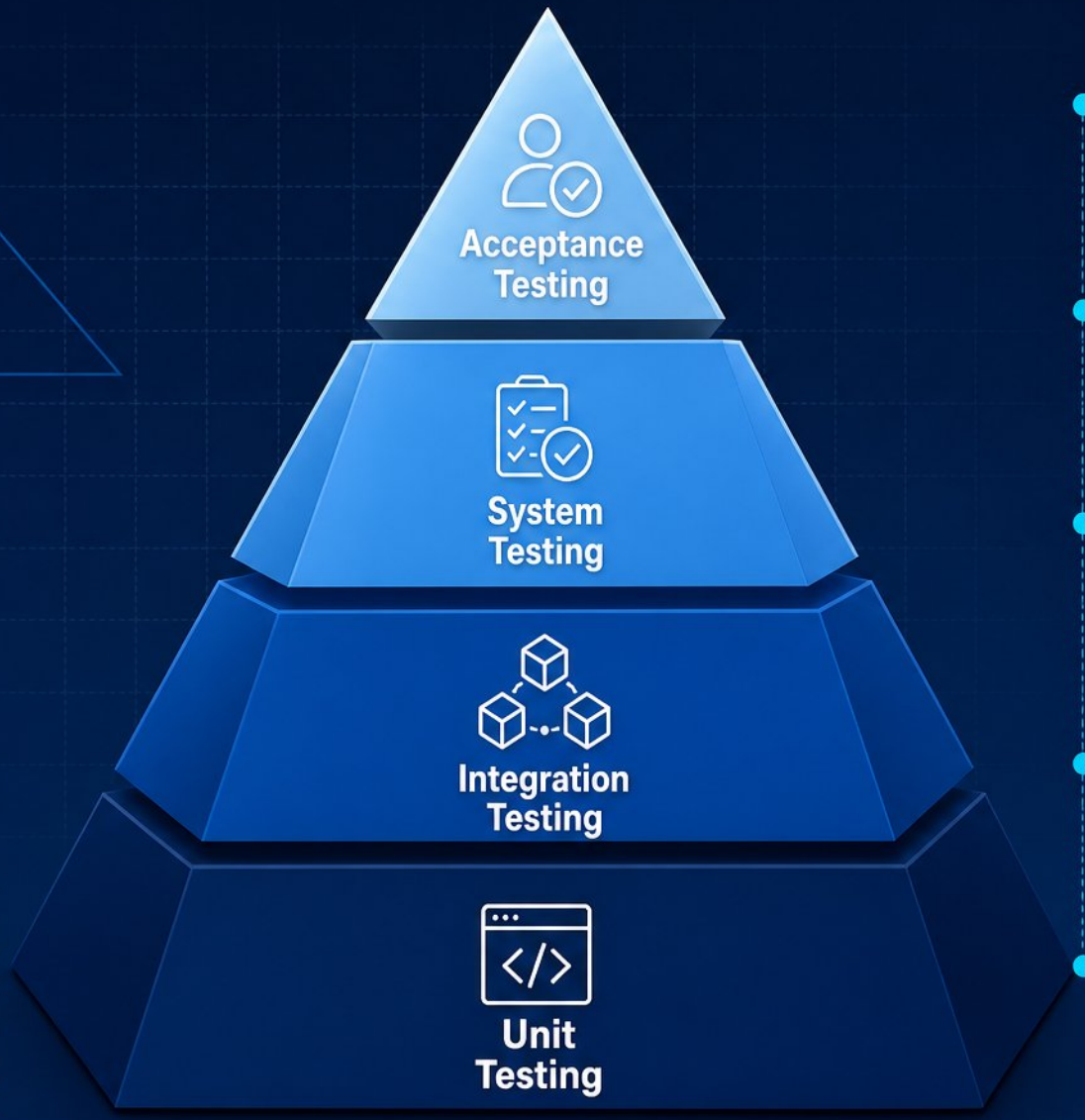
- Absence-of-errors fallacy: bug-free systems still fail if they don't meet user needs

Understanding Defects, Errors, and Failures



- **Error:** human mistake producing an incorrect result
- **Defect:** the flaw in code that can cause failure
- **Failure:** observable deviation from expected behavior
- **Chain:** error → defect → failure; not instant
- **Root cause:** fix the process to prevent recurrence

Testing Levels: From Unit to Acceptance



Unit Testing: validates isolated functions/methods; fastest, cheapest, developer-owned

Integration Testing: verifies component interactions, interfaces, and data flow

System Testing: evaluates the complete system against requirements in a production-like environment

Acceptance Testing: confirms the system meets business/user needs (UAT, OAT)

Testing Pyramid: many unit tests, fewer integration tests, even fewer system/acceptance tests

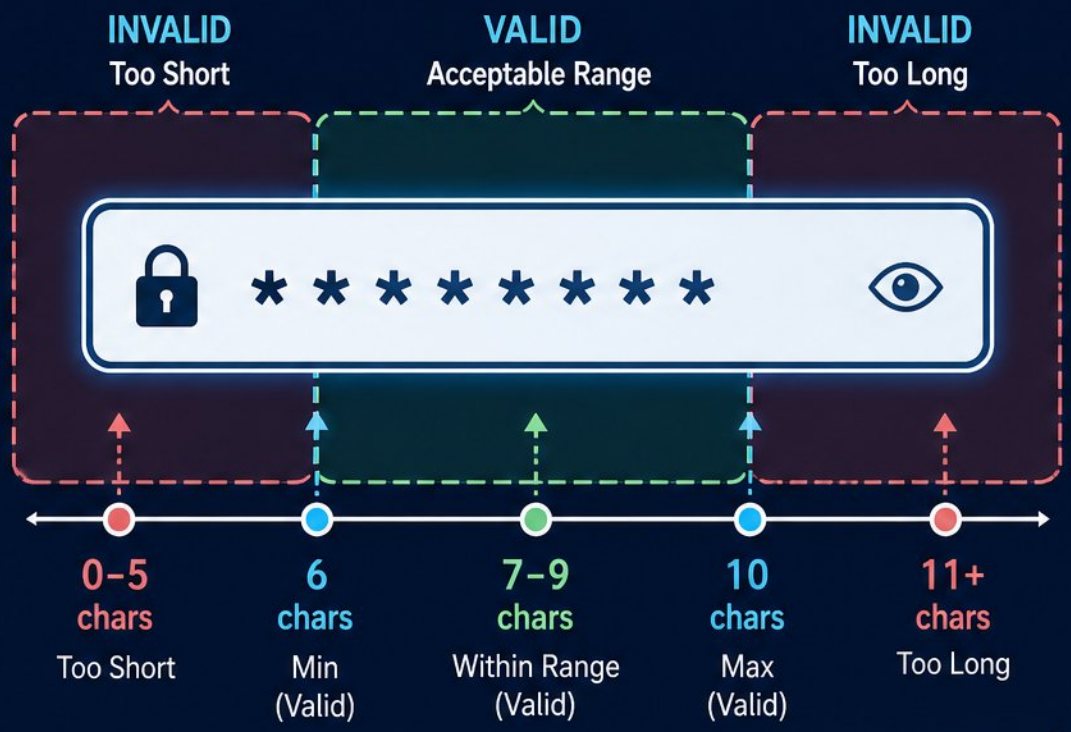
Static vs. Dynamic Testing and Test Types

- **Static testing:** review work products without running code (reviews, static analysis)
- **Dynamic testing:** execute software to validate behavior (functional and non-functional)
- **Functional testing:** checks what the system does (login, search, checkout)
- **Non-functional testing:** checks how the system performs (performance, security, usability)
- **White-box:** tests internal structure; black-box: tests without code knowledge; gray-box: combines both

The Structured Test Process

Designing Effective Test Cases with Black-Box Techniques

- **Equivalence Partitioning:** Group valid and invalid inputs; test one value per class
- **Boundary Value Analysis:** Target edge values (min, max, just inside, just outside)
- **Example: Password field** (6–10 chars) → bound. tests: 0,5,6,7,10,11
- **Decision Tables:** Map condition combos to expected actions for business logic
- **State Transition Testing:** Model states/events; test valid and invalid transitions



Example: Password field (6–10 chars) → bound. tests: 0,5,6,7,10,11

Writing Your First Tests with pytest

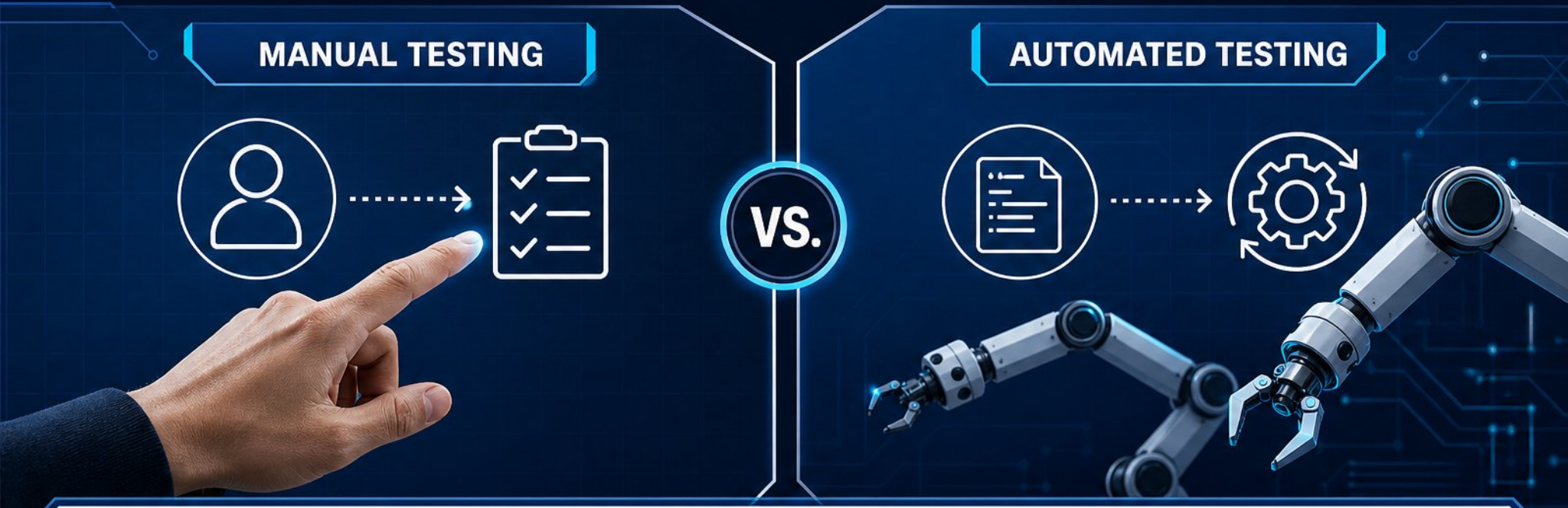
- Install pytest: `pip install pytest` and verify with `pytest --version`
- Test discovery: files named `test_*.py` or `*_test.py`, functions prefixed `test_`
- Write tests with Python's `assert` statement; pytest shows detailed failure messages
- Run tests: `pytest -v` for verbose, `-k` to filter by name, `--lf` for last failed
- Test exceptions: `pytest.raises()` confirms expected exceptions under specific conditions

```
$ pytest -v
========================== test session starts ==========================
collecting ...
test_sample.py::test_addition PASSED
test_sample.py::test_division_by_zero FAILED
============================== FAILURES ==============================
_____________________ test_division_by_zero _____________________
   def test_division_by_zero():
       result = 1 / 0
>      assert result == 0
E      assert (ZeroDivisionError) == 0
E      +  where ZeroDivisionError = 1 / 0
======================== short test summary info ========================
FAILED test_sample.py::test_division_by_zero
========================= 1 failed, 1 passed =========================
```

Structuring Tests with Fixtures and Parametrization

- Fixtures: reusable setup defined with @pytest.fixture() and injected as a parameter
- Fixtures reduce duplication, isolate data, and ensure a clean state per test
- Parametrize: @pytest.mark.parametrize runs one test with many input sets
- Example: add(2,3,5), (-1,-1,-2), (0,0,0) — one function, three test cases
- Best practices: descriptive names, one behavior per test, keep tests readable

Manual vs. Automated Testing: When to Use Each



- Manual testing: human-driven execution; best for exploration, usability, and ad-hoc checks
- Automated testing: script-driven execution; ideal for regression, large data sets, and CI/CD
- Manual strengths: flexibility, intuition, catching subtle UX and visual issues
- Automation strengths: speed, repeatability, consistency, 24/7 execution
- Balanced strategy: automate what is repetitive and stable; manually test what requires human judgment

Defect Reporting and Tracking

- Anatomy of a good bug report: clear title, steps to reproduce, expected vs. actual results, environment details, and attachments (screenshots/logs).
- Severity vs. Priority: severity is system impact (e.g., crash, data loss); priority is business urgency to fix.
- Reproduction is key: a bug that cannot be reproduced is nearly impossible to fix; always include minimal steps to trigger the defect.
- Use issue trackers like Jira, GitHub Issues, or GitLab to log, assign, and track defects through their lifecycle.
- A defect report is a communication tool, not a blame assignment; write clearly and objectively.

The Tester's Mindset and Real-World Collaboration

- Cultivate curiosity: explore edge cases, not just happy paths
- Collaborate early: join requirement reviews and sprint planning
- Handle incomplete requirements: use exploratory testing, ask questions
- Embrace shift-left: integrate testing at every SDLC stage
- Own quality together: testing is a whole-team responsibility in Agile/DevOps

Your Testing Journey: Next Steps and Resources

- **Recap:** fundamentals, principles, test levels, design techniques, and pytest basics
- **Practice:** write tests for a small project, focusing on unit tests and boundary value analysis
- **Explore:** test-driven development (TDD), behavior-driven development (BDD), and API testing
- **Connect:** join testing forums, attend meetups, and consider ISTQB Foundation Level certification
- **Remember:** testing is a skill that grows with practice; keep learning and questioning

PersonWise.

PersonWise • AI digital-human course platform

Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/ea53c44d/public