Software Testing Fundamentals

> IE University · BCSAI · SDD · 2025

1. Why testing matters

- Catches bugs early → cheaper to fix
- Enables safe refactoring and evolution
- Tests as living documentation
- Improves design (modularity, loose coupling)

Key terminology (quick)

- Test case: scenario (inputs, steps, expected outcome)
- Test suite: collection of tests
- Assertion: check that verifies behavior
- Code coverage: lines/branches/functions executed by tests

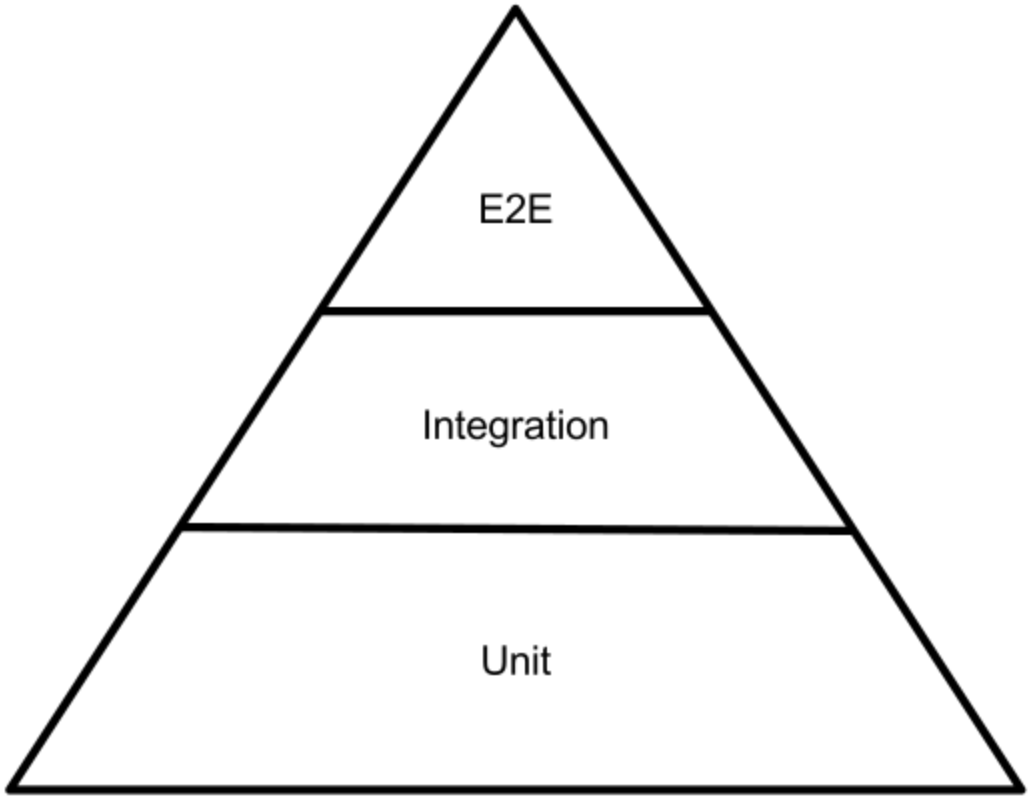
AAA pattern

- Arrange: prepare inputs & preconditions
- Act: execute code under test
- Assert: verify expected outcome

```
# Arrange / Act / Assert
calculator = Calculator()
result = calculator.add(2, 3)
assert result == 5
```

2. Testing Pyramid

- Unit tests: fast, isolated, majority (70-80%)
- Integration tests: component interactions (15-20%)
- E2E tests: full stack, limited set (5-10%)



Unit tests

- Target: single function/method in isolation
- Characteristics: fast, reliable, repeatable
- Use when: business logic, edge cases

Example (pytest):

```
def test_calculate_discount():
    assert calculate_discount(100, 20) == 80
```

Integration tests

- Verify components work together
- May require DB, queues, external services (use test doubles or isolated test env)
- Slower and more fragile than unit tests

Example: testing DB repository

```
import pytest

def test_user_repository(db_session):
    # Arrange
    user = User(email="john@example.com", name="John Doe")
    # Act
    db_session.add(user)
    db_session.commit()
    retrieved = db_session.query(User).filter_by(email="john@example.com").first()
    # Assert
    assert retrieved is not None
    assert retrieved.name == "John Doe"
```

E2E tests

- Simulate real user workflows (UI → backend → DB)
- High confidence but slow and brittle
- Focus on critical happy-path journeys

Example: Playwright snippet for login

3. Test automation & frameworks

- Automate tests for speed, consistency, CI/CD
- Popular Python tools: pytest, unittest, pytest-mock
- For browser: Playwright, Selenium

Benefits:

- Fast feedback, consistent execution, CI integration

pytest basics

- Simple function-based tests with plain assert
- Fixtures for setup/teardown
- Plugins (pytest-mock) for mocking

Example:

```
# sum.py
def sum(a, b): return a + b

# test_sum.py
def test_adds_positive_numbers():
    assert sum(1, 2) == 3
```

Mocking dependencies

- Replace slow/side-effecting deps (DB, network) with mocks
- pytest-mock provides `mocker` fixture
- Configure return values and side effects

Example:

```
mock_db = mocker.Mock()
mock_db.find_by_id.return_value = {'id':1,'name':'Alice'}
service = UserService(mock_db)
user = service.get_user(1)
assert user['name'] == 'Alice'
```

4. Fixtures (pytest)

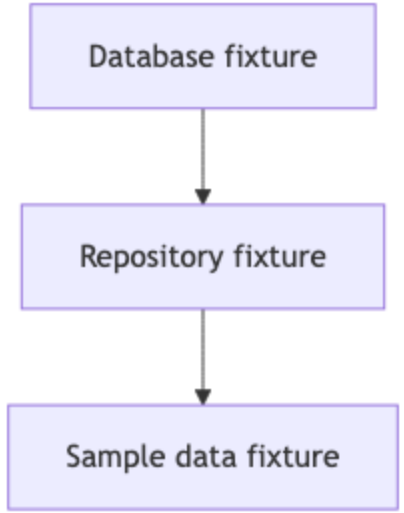
- Reusable setup/teardown via `@pytest.fixture`
- Scopes: function (default), class, module, session
- Use `yield` to separate setup/teardown

Example:

```
@pytest.fixture(scope='module')
def database_connection():
    conn = create_db_connection()
    yield conn
    conn.close()
```

Fixture composition & dependencies

- Fixtures can depend on other fixtures
- Keeps setups small and reusable



Mocking vs Stubs vs Fakes

- Mock: records interactions, verify calls
- Stub: returns fixed values (no verification)
- Fake: simplified working implementation (e.g., in-memory DB)

When to use each:

- Unit tests: mocks/stubs
- Integration-like tests: fakes

Patching & monkeypatch

- Use `monkeypatch` (pytest) or `mocker.patch` to replace objects
- Great for time, env vars, or internal creation of deps

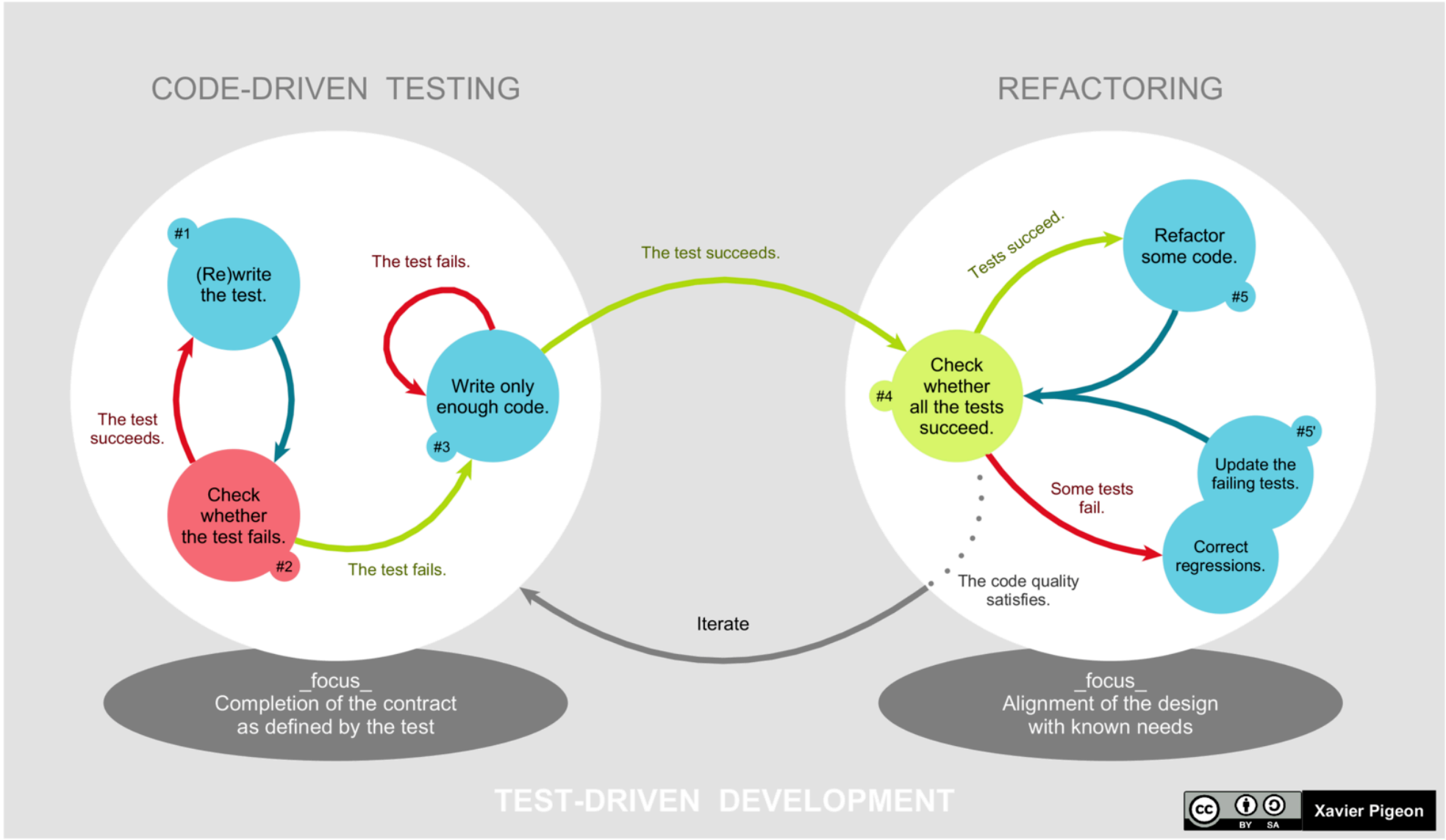
Example:

```
monkeypatch.setenv('API_KEY','test-key')
```

5. Test-Driven Development (TDD)

- Red → Green → Refactor cycle
  - Red: write failing test
  - Green: implement minimum to pass
  - Refactor: improve design while tests stay green

Benefits: designs APIs from user's perspective, builds coverage, supports refactoring


CODE-DRIVEN TESTING
REFACTORING
#1
(Re)write the test.
The test fails.
The test succeeds.
Write only enough code.
#3
The test succeeds.
Check whether the test fails.
#2
The test fails.
The test succeeds.
Tests succeed.
Refactor some code.
#5
Check whether all the tests succeed.
#4
#5'
Update the failing tests.
Some tests fail.
Correct regressions.
The code quality satisfies.
Iterate
focus
Completion of the contract as defined by the test
focus
Alignment of the design with known needs
TEST-DRIVEN DEVELOPMENT
Xavier Pigeon

TDD example: password validator (summary)

- Start with a failing test for minimum length
- Implement minimal code
- Add next test (uppercase required) → expand implementation
- Repeat and refactor

6. Best practices (FIRST)

- Fast: quick to run
- Independent: no shared state
- Repeatable: same result every run
- Self-validating: pass/fail clearly
- Timely: written with / before code

More:

- Descriptive test names
- Prefer behavior tests over implementation details
- One assertion per test when possible

Handling flaky tests

Common causes:

- Time dependencies
- Async operations / race conditions
- External deps
- Test-order dependency
- Random data

Mitigations:

- Mock time, use proper waiting, isolate external services, seed RNG, make tests independent

Real-world example: API endpoint tests

- Use app test client fixture
- Test happy path, validation errors, duplicate resources

Short example outline:

```
def test_creates_user(client):
    resp = client.post('/api/users', json={...})
    assert resp.status_code == 201
```

Code coverage guidance

- Aim: ~70-80% as a practical goal
- Coverage finds untested code; focus on business-critical paths
- 100% coverage ≠ bug-free tests

7. Takeaways

- Testing is essential for quality and confidence
- Invest in unit tests; use integration/E2E sparingly
- TDD improves design and test coverage
- Automate tests in CI/CD
- Tests should focus on behavior, be fast and independent

Q&A / Common questions

- How much testing is enough?
  - enough to deploy confidently
- Test private methods?
  - test via public API; extract if needed
- Testing legacy code?
  - characterization tests, then refactor
- ROI?
  - fewer bugs, faster changes, lower maintenance costs