# **Functional Testing**

# **Unit, Integration, System & UAT**

## **📖 Introduction**

Functional Testing validates software against business requirements by testing:  
✔ Features (Does it work as specified?)  
✔ User Interactions (Is the workflow correct?)  
✔ Data Processing (Are inputs/outputs accurate?)

Four Key Levels:

1. Unit Testing → Tests individual components.
2. Integration Testing → Checks module interactions.
3. System Testing → Validates the full system.
4. User Acceptance Testing (UAT) → Confirms readiness for release.

## **🧩 Unit Testing**

### **What is it?**

Testing smallest code units (functions, methods, classes) in isolation.

### **Key Aspects**

| **Aspect** | **Details** |
| --- | --- |
| Scope | Single function/method. |
| Ownership | Developers. |
| Tools | JUnit (Java), pytest (Python), NUnit (.NET). |
| Automation | 100% automated. |

### **Techniques**

* White-Box Testing: Covers code paths (branches, statements).
* Mocking: Isolates dependencies (Mockito, Sinon.js).

### **Example**

# Python (pytest)

def add(a, b):

return a + b

def test\_add():

assert add(2, 3) == 5 # Pass

assert add(-1, 1) == 0 # Pass

### **Pros & Cons**

| **Pros** | **Cons** |
| --- | --- |
| ✅ Early bug detection. | ❌ Misses integration issues. |
| ✅ Fast execution. | ❌ Requires developer involvement. |

## **🔗 Integration Testing**

### **What is it?**

Tests interactions between modules/components.

### **Approaches**

1. Big Bang: Test all modules at once (risky).
2. Top-Down: Test from top-level modules first (stubs for lower levels).
3. Bottom-Up: Test lower-level modules first (drivers for upper levels).
4. Sandwich: Hybrid (top + bottom simultaneously).

### **Key Aspects**

| **Aspect** | **Details** |
| --- | --- |
| Scope | Multi-module workflows. |
| Ownership | Developers + QA. |
| Tools | Postman (APIs), Selenium (UI). |

### **Example**

* Scenario: Test login (UI) + auth (API) integration.
* Tool: Postman (API calls) + Cypress (UI validation).

### **Pros & Cons**

| **Pros** | **Cons** |
| --- | --- |
| ✅ Catches interface defects. | ❌ Complex to debug. |
| ✅ Validates data flow. | ❌ Slower than unit tests. |

## **🖥️ System Testing**

### **What is it?**

Tests the entire system end-to-end against requirements.

### **Types**

| **Type** | **Purpose** |
| --- | --- |
| Functional | Validates features. |
| Regression | Ensures new changes don’t break old features. |
| Performance | Tests speed, scalability. |
| Security | Checks vulnerabilities. |

### **Key Aspects**

| **Aspect** | **Details** |
| --- | --- |
| Scope | Full application. |
| Ownership | QA team. |
| Tools | Selenium, JMeter, OWASP ZAP. |

### **Example**

* Test Case: "User adds item to cart → checks out → receives confirmation email."
* Tools: Selenium (UI) + Mailosaur (email testing).

### **Pros & Cons**

| **Pros** | **Cons** |
| --- | --- |
| ✅ Real-world simulation. | ❌ Time-consuming. |
| ✅ Finds environment-specific bugs. | ❌ Expensive to automate. |

## **👥 User Acceptance Testing (UAT)**

### **What is it?**

Final testing by end-users/clients to confirm readiness for production.

### **Types**

* Alpha Testing: Internal team (simulated environment).
* Beta Testing: Real users (production-like environment).

### **Key Aspects**

| **Aspect** | **Details** |
| --- | --- |
| Scope | Business workflows. |
| Ownership | End-users + stakeholders. |
| Tools | JIRA (bug tracking), TestRail (test cases). |

### **Example**

* Scenario: Retail client tests purchase workflow before Black Friday launch.
* Criteria: "All critical transactions must process in <2 seconds."

### **Pros & Cons**

| **Pros** | **Cons** |
| --- | --- |
| ✅ Validates business needs. | ❌ Slow feedback cycle. |
| ✅ Reduces post-launch defects. | ❌ Limited technical depth. |

## **🆚 Comparison Summary**

| **Aspect** | **Unit** | **Integration** | **System** | **UAT** |
| --- | --- | --- | --- | --- |
| Scope | Code Unit | Module Interactions | Full System | Business Needs |
| Tester | Developers | Devs + QA | QA Team | End-Users |
| Environment | Local | Test Env | Staging | Production-like |
| Automation | 100% | Partial | Partial | Rare |

## **🏆 Best Practices**

✔ Unit Testing:

* Aim for 90%+ code coverage.
* Run in CI pipelines (e.g., GitHub Actions).

✔ Integration Testing:

* Use contract testing (Pact) for microservices.
* Test API responses + DB states.

✔ System Testing:

* Prioritize critical user journeys.
* Combine manual + automated checks.

✔ UAT:

* Define clear exit criteria (e.g., "0 Critical bugs").
* Involve real users early (Beta programs).

## **🌍 Real-World Examples**

### **Case 1: Unit Testing Saves PayPal**

* Issue: Currency conversion bug in a utility function.
* Catch: Unit test failed for edge-case values.
* Fix: Patched before integration.

### **Case 2: Facebook API Integration Failure**

* Issue: Graph API returned 500 errors after an update.
* Root Cause: Missing integration tests for rate limits.
* Solution: Added Postman test suites.

### **Case 3: NHS COVID App (UAT Success)**

* Process: 50,000 beta testers validated usability.
* Result: Smooth public rollout with 98% satisfaction.

## **📚 References**

* [ISTQB Functional Testing](https://www.istqb.org/)
* [Martin Fowler on Unit Testing](https://martinfowler.com/bliki/UnitTest.html)
* [Google Testing Blog](https://testing.googleblog.com/)

🔹 Conclusion:

* Unit → Integration → System → UAT = Progressive validation.
* Automate early (unit/integration), validate late (UAT).
* Balance speed and coverage for optimal quality.

