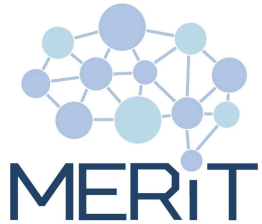
Testing 2

Test Types by Scope / Level

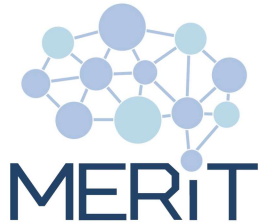
Unit Testing

Definition



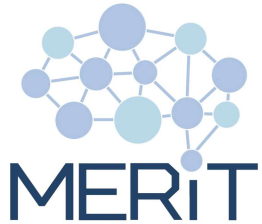
- Unit testing involves **verifying individual units** of source code in isolation to **ensure they behave correctly**.
- A “unit” typically refers to the **smallest testable part** of an application, usually a function, method, or class.

Purpose



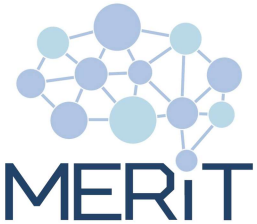
- Detect **bugs early** in the development process.
- Ensure **code correctness** before integration.
- Enable safe **refactoring**.
- Serve as a live, executable **specification** for a unit's behavior.

Characteristics



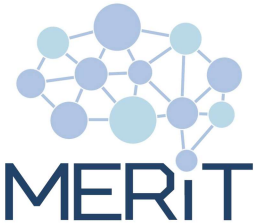
- **Fast**: Executes in milliseconds.
- **Deterministic**: Always produces the same result.
- **Isolated**: No database, file system, network, or external dependencies.
- **Automated**: Run frequently as part of a CI/CD pipeline.

When to Write Unit Tests



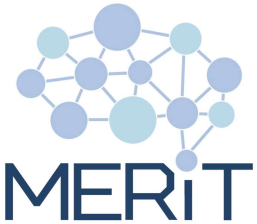
- When developing **core business logic**.
- For functions with **complex conditional flows**.
- For public methods that are **reused** or **critical** to correctness.

When NOT to Write Unit Tests



- For **trivial** getters/setters or one-liners with **no logic**.
- For private methods that are **indirectly tested** through public methods.
- For code that's **likely** to be **deleted** or **rewritten** soon.

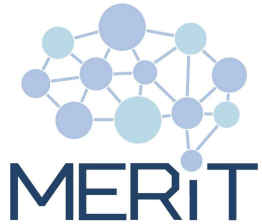
Mocking, Stubbing, and Faking



Used to **isolate** the unit and eliminate external influences.

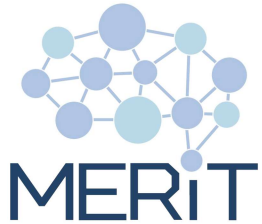
- **Mock**: A fake object that verifies it was used in a certain way.
- **Stub**: A replacement that returns predefined data.
- **Fake**: A lightweight, working implementation (e.g., in-memory DB).

Best Practices



- Test **one behavior** at a time.
- Use clear **naming conventions** (e.g., should_throw_error_when_password_invalid).
- Keep tests **independent** and **idempotent**.
- Mock/stub **external dependencies** (e.g., I/O, services).
- Follow the **AAA** pattern: Arrange, Act, Assert.

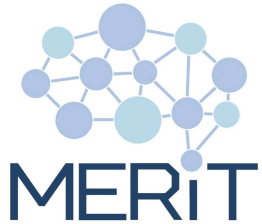
Code Coverage



- **Statement** coverage: All lines executed?
- **Branch** coverage: All if/else conditions evaluated?
- **Path** coverage: All code paths tested?

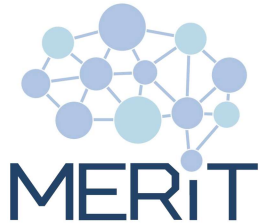
High coverage is useful, but not sufficient. Good tests also assert correct behavior, not just code execution.

Common Pitfalls



- **Overuse** of mocking → tightly coupled tests.
- Testing **implementation details** instead of behavior.
- Not **refactoring** tests → test debt.
- Writing unit tests for code that should be tested at a **higher level**.

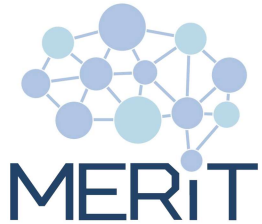
Industry Practice: TDD



- Test-Driven Development
- Write the **test first** (which fails).
- Write the **minimal code to pass** the test.
- **Refactor** the code and keep the test passing.

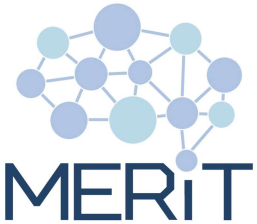
Integration Testing

Definition



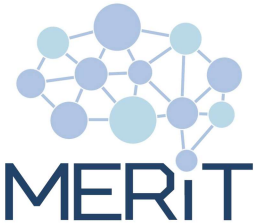
- Integration testing verifies that **multiple components work together** correctly.
- It checks whether **interfaces**, **data flows**, and **interactions between units** behave as expected.
- Unlike unit tests, integration tests **don’t isolate** components; instead, they test how they **collaborate**.

Purpose



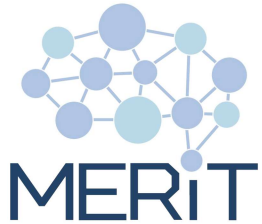
- Validate the correctness of **inter-component communication**.
- Detect issues such as **mismatched data formats**, **broken interfaces**, or **failed contracts**.
- Uncover bugs in **module boundaries** (often missed by unit tests).
- Build confidence that subsystems interact correctly **before full system testing**.

Types of Integration Testing



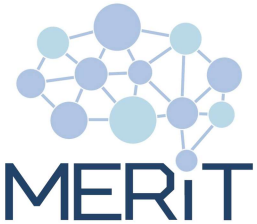
- **Big Bang** Integration: All modules are integrated together and tested as a whole.
- **Incremental** Integration:
  - ✔ **Top-Down**: Higher-level modules are tested first with stubs for lower-level modules.
  - ✔ **Bottom-Up**: Lower-level modules are tested first, using drivers for upper layers.
  - ✔ **Sandwich / Hybrid**: Combines both approaches.

Types of Integration Testing



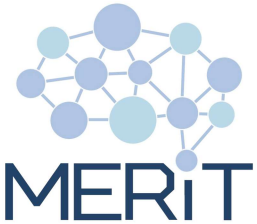
- **Continuous Integration** Testing: Triggered automatically in CI/CD pipelines as modules are developed and integrated.

Examples



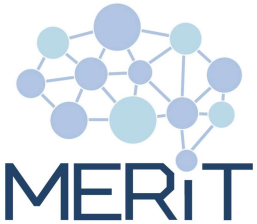
- A REST API that calls a service layer, which queries a database.
- A microservice that consumes another via HTTP.
- A frontend that calls backend endpoints.

Tools



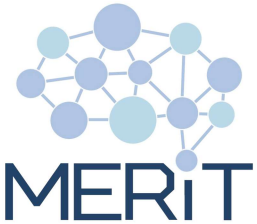
- **Postman**, **Newman**: API test scripting and automation
- **Supertest** (Node.js), **Spring Test** (Java): HTTP endpoint testing
- **TestContainers** (Java, Python): Spin up actual DBs/services for realistic integration tests
- **Docker Compose**: For setting up integration environments locally

Common Patterns



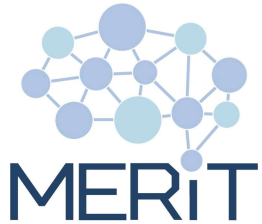
- **Postman**, **Newman**: API test scripting and automation
- **Supertest** (Node.js), **Spring Test** (Java): HTTP endpoint testing
- **TestContainers** (Java, Python): Spin up actual DBs/services for realistic integration tests
- **Docker Compose**: For setting up integration environments locally

Best Practices



- Don’t test everything in integration tests; instead, **focus on risky or critical interactions**.
- Maintain **realistic environments** (schema, endpoints, auth).
- Use **unique test data** to avoid false positives and negatives.
- **Clean up** test artifacts (data, files, resources).

Common Pitfalls



- **Brittle tests** due to tight coupling or environment assumptions.
- **Slow execution times** occur if too many integration tests or heavy resources are used.
- Tests **relying on external services** → flaky or unreliable builds.
- Lack of **proper test data** setup → hard to reproduce bugs.

System Testing

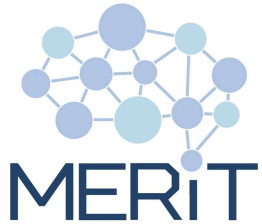
Definition

- System testing is the process of validating a **complete, integrated system** to ensure it meets its functional and non-functional requirements.
- It **simulates real-world usage** to verify that the system behaves correctly as a whole.

Purpose

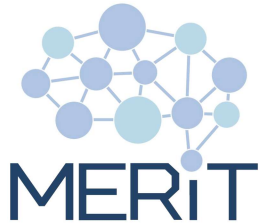
- Verify the system's overall **end-to-end behavior** from the user or stakeholder perspective.
- Confirm that **all** integrated components **work together** as expected.
- Ensure that **functional requirements** (features) and **non-functional requirements** (performance, usability, security) are fulfilled.

Scope



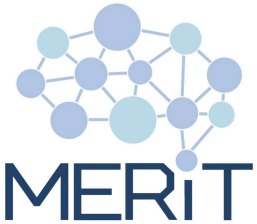
- Full application flow (UI → backend → DB → services → response)
- Realistic environments (or production-like)
- Functional correctness
- Performance under realistic conditions
- Data integrity across modules
- Workflow and state transitions

Tools



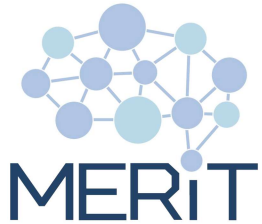
- **UI Automation**: Selenium, Cypress, Playwright
- **API Testing**: Postman/Newman, RestAssured, Karate
- **Load Testing**: JMeter, Locust, k6
- **Security Testing**: OWASP ZAP, Burp Suite
- **Accessibility Testing**: axe-core, Lighthouse

Best Practices



- Mirror the **production environment** as closely as possible.
- Maintain stable **test data** and **reset** routines.
- Use **CI environments** with proper isolation and teardown steps.
- Clearly separate **functional failure** from **infrastructure failure** (e.g., service timeouts vs. app bugs).

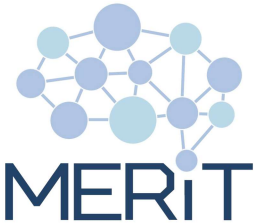
Common Pitfalls



- **Environment drift**: differences between test/staging/production.
- **Flaky tests** due to external dependencies or unstable data.
- Long **setup/teardown times** and poor **reproducibility**.
- False confidence from **shallow tests** (e.g., happy paths only).

Acceptance Testing

Definition

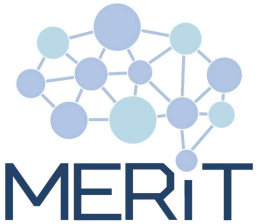


- Acceptance testing determines whether the system **meets the business requirements** and **is ready to be accepted** by the customer, product owner, or end-user.
- This testing ensures that the software **performs as intended**, not just that it works technically.

Purpose

- Validate the system from the **user or business stakeholder's perspective**.
- Ensure all **critical business workflows** are functional.
- Gain **formal sign-off** for product release.
- **Reduce the risk** of customer dissatisfaction, rework, or release delays.

Types of Acceptance Testing



- **User Acceptance Testing (UAT):**
  - ✔ Conducted by business stakeholders or real users.
  - ✔ Focuses on real-world tasks and usability.
  - ✔ Usually manual and scripted, but sometimes exploratory.
  - ✔ Often performed in a UAT/staging environment.

Types of Acceptance Testing

- **Business Acceptance Testing (BAT)**:
  - ✔ Performed by product owners or business analysts.
  - ✔ Verifies alignment with business rules, contracts, and goals.

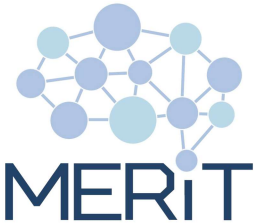
Types of Acceptance Testing

- **Alpha Testing**:
  - ✔ Internal acceptance test done by the development organization.
  - ✔ Performed before the software is released to external users.

Types of Acceptance Testing

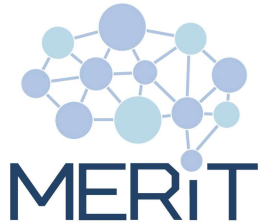
- **Beta Testing**:
  - ✔ Performed by selected external users in real-world conditions.
  - ✔ Feedback is gathered for final fixes and improvements.

Tools



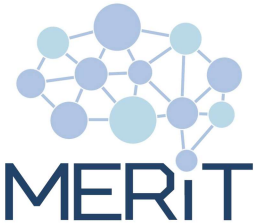
- **Cucumber** (Java, JS, etc.): Supports BDD with Gherkin syntax (Given, When, Then)
- **Robot Framework**: Keyword-driven acceptance tests
- **FitNesse**: Wiki-based collaborative acceptance test tool
- **SpecFlow**, **Behave**: BDD frameworks for .NET and Python

Best Practices



- Write tests in **natural**, **business-readable language**.
- Collaborate with stakeholders to define **clear acceptance criteria**.
- Run acceptance tests on **stable builds** only.
- **Automate** repeatable acceptance scenarios when possible.

Common Pitfalls



- Ambiguous or missing acceptance criteria.
- Stakeholders not involved in writing/approving test cases.
- Acceptance testing done too late → bottlenecks or last-minute rejections.
- Tests that focus on UI quirks instead of core business value.