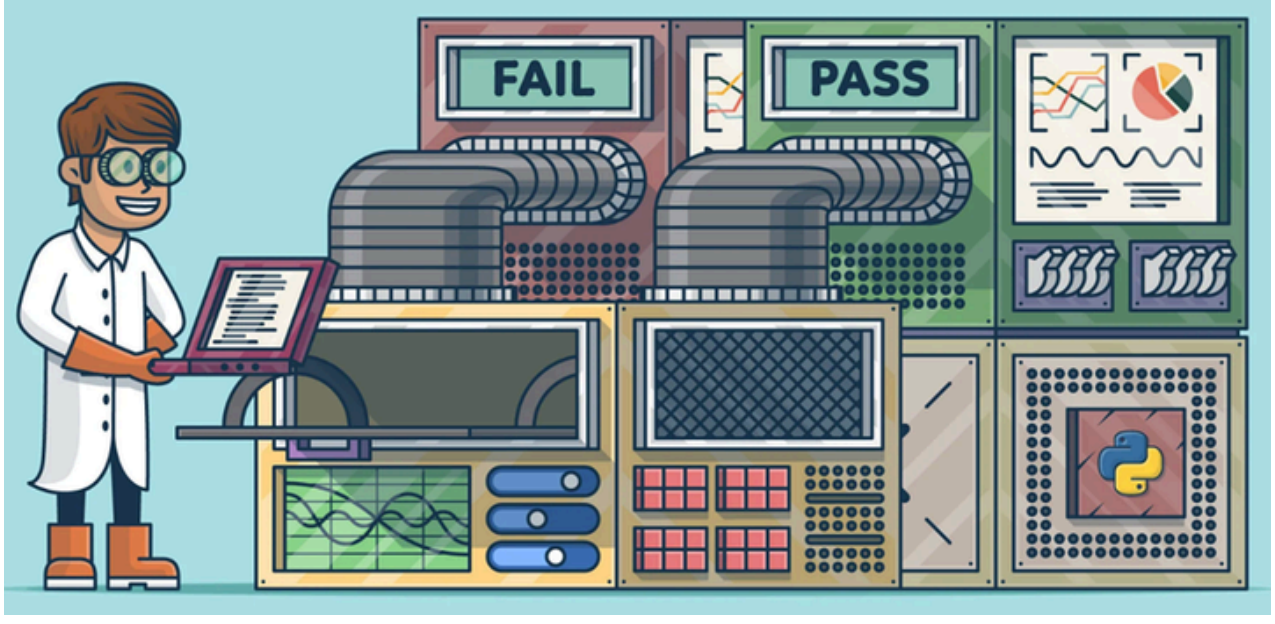
Unit Testing - Kiểm thử đơn vị

Giới thiệu về kiểm thử đơn vị trong phát triển phần mềm



Line Coverage (Bao phủ theo dòng lệnh)

🧠 Định nghĩa:

Line Coverage đo tỷ lệ các dòng code được thực thi khi chạy unit test.

💡 Ví dụ minh họa:

```
function greet(name) {
  if (name) {
    console.log("Hello, " + name);
  }
}
```

✅ Test chạy dòng: greet("An")
⛔ Dòng chưa chạy: dòng trong if sẽ không được chạy nếu test gọi greet(null)

🔍 Tại sao quan trọng?

Giúp đảm bảo rằng không có dòng nào bị "bỏ quên" – tránh lỗi tiềm ẩn.

Branch Coverage (Bao phủ nhánh điều kiện)

🧠 Định nghĩa:

Branch Coverage kiểm tra tất cả các nhánh điều kiện (if, else, switch) có được test hay không.

💡 Ví dụ minh họa:

```
function checkAge(age) {
  if (age >= 18) {
    return "Adult";
  } else {
    return "Minor";
  }
}
```

✅ Cần test:

- checkAge(20) → đi nhánh if
- checkAge(16) → đi nhánh else

🔍 Tại sao quan trọng?

Đảm bảo mọi logic phân nhánh đều được kiểm thử – tránh lỗi khi logic bị bỏ sót.

Function Coverage (Bao phủ theo hàm)

🧠 Định nghĩa:

Function Coverage kiểm tra xem tất cả các hàm/method đã được gọi ít nhất một lần chưa.

💡 Ví dụ minh họa:

```
function add(a, b) { return a + b; }
function subtract(a, b) { return a - b; }
```

✅ Nếu chỉ test add(1,2) thì subtract không được test → Coverage thấp

🔍 Tại sao quan trọng?

Giúp phát hiện các hàm chưa bao giờ được sử dụng hoặc kiểm thử.

Path Coverage (Bao phủ đường đi logic)

🧠 Định nghĩa:

Path Coverage kiểm tra tất cả đường đi logic qua chương trình (tổ hợp điều kiện).

```
function login(user, pass) {
  if (user && pass) {
    if (user === "admin") return "Admin Login";
    return "User Login";
  }
  return "Login Failed";
}
```

✅ Cần test:

- login("admin", "123")
- login("user", "123")
- login("", "123")
- login("admin", "")

🔍 Tại sao quan trọng?

Đảm bảo mọi kịch bản logic được test – đặc biệt quan trọng với hệ thống phức tạp.

Đặc điểm của một Unit Test tốt

- Xác định rõ đầu vào và đầu ra mong đợi.
- Nhanh chóng: Chạy nhanh và hiệu quả.
- Độc lập: Không phụ thuộc vào test khác.
- Đảm bảo kiểm thử **happy case** (thành công) và **edge case** (dữ liệu không hợp lệ).
- Có thể tái sử dụng và bảo trì.

Mức coverage tối thiểu

1. Tham khảo chuẩn Industry.

- Mỗi tổ chức hoặc hệ thống sẽ có mức coverage tối thiểu khác nhau như sau:

Loại hệ thống	**Mức coverage khuyến khích**
Hệ thống thông thường	70% - 80%
Hệ thống tài chính/ nhạy cảm	90%+
Library/Open source	> 80% để tăng niềm tin của cộng đồng người sử dụng

Mức coverage tối thiểu

2. Kết luận

- Không cần “100%” cho mọi trường hợp:
  - Mục tiêu không nên đạt là 100% coverage vì:
    - Một số đoạn code không thể test (logging, debug, hoặc các exception rất hiếm khi xảy ra.
    - Test “để tăng coverage” nhưng không kiểm tra login thật sự có thể gây ảo tưởng hệ thống tin cậy, an toàn
- Tập trung vào logic quan trọng:
  - Ưu tiên kiểm thử kỹ càng:
    - Các service, controller, và business logic.
    - Các phần xử lý transaction, luồng chính của user.
  - Có thể giảm độ ưu tiên cho:
    - Getter/ setter đơn giản
    - Code không ảnh hưởng đến logic chính.

Best practice khi viết Unit Test với JUnit (Java & Spring)

1. Tuân thủ cấu trúc đặt tên.

- Đặt tên test theo cú pháp sau để dễ đọc và bảo trì methodName_StateUnderTest_ExpectedBehavior
- Ví dụ:

```
@Test
void createStudent_WithValidData_ShouldReturnSavedStudent() {
    // Arrange
    // Act
    // Assert
}
```

Best practice khi viết Unit Test với JUnit (Java & Spring)

2. Sử dụng @BeforeEach và @AfterEach để chuẩn bị dữ liệu.

- Tránh code lặp lại bằng cách thiết lập dữ liệu chung cho các test case.
- Ví dụ:

```
@BeforeEach
void setUp() {
    studentService = new StudentService(studentRepository);
}
```

Best practice khi viết Unit Test với JUnit (Java & Spring)

4. Sử dụng Mock để tránh phụ thuộc vào Database hoặc Persistent storage.

- Không nên gọi trực tiếp Database trong Unit test, mà thay vào đó sử dụng Mock(Mockito) để mock repository.
- Ví dụ:

```
@Mock
private StudentRepository studentRepository;

@InjectMocks
private StudentService studentService;

@Test
void findStudentById_ExistingId_ShouldReturnStudent() {
    // Arrange
    Student student = new Student(1L, "John Doe");
    when(studentRepository.findById(1L)).thenReturn(Optional.of(student));

    // Act
    Student result = studentService.getStudentById(1L);

    // Assert
    assertNotNull(result);
    assertEquals("John Doe", result.getName());
}
```

Best practice khi viết Unit Test với JUnit (Java & Spring)

5. Đảm bảo kiểm thử Happy case & Edge case.

- Happy case: Dữ liệu hợp lệ, kết quả mong đợi.
- Edge case: Dữ liệu thiếu, không hợp lệ, null, kết quả có thể không mong đợi.
- Ví dụ:

Java

```java
@Test
void getStudentById_NonExistingId_ShouldThrowException() {
    when(studentRepository.findById(anyLong())).thenReturn(Optional.empty());

    assertThrows(StudentNotFoundException.class, () -> {
        studentService.getStudentById(999L);
    });
}
```

Kết luận

- Unit Testing là bước quan trọng giúp nâng cao chất lượng phần mềm.
- Giúp phát hiện lỗi sớm, giảm chi phí sửa lỗi về sau.
- Kết hợp Unit Testing với các phương pháp kiểm thử khác để đảm bảo chất lượng tốt nhất.