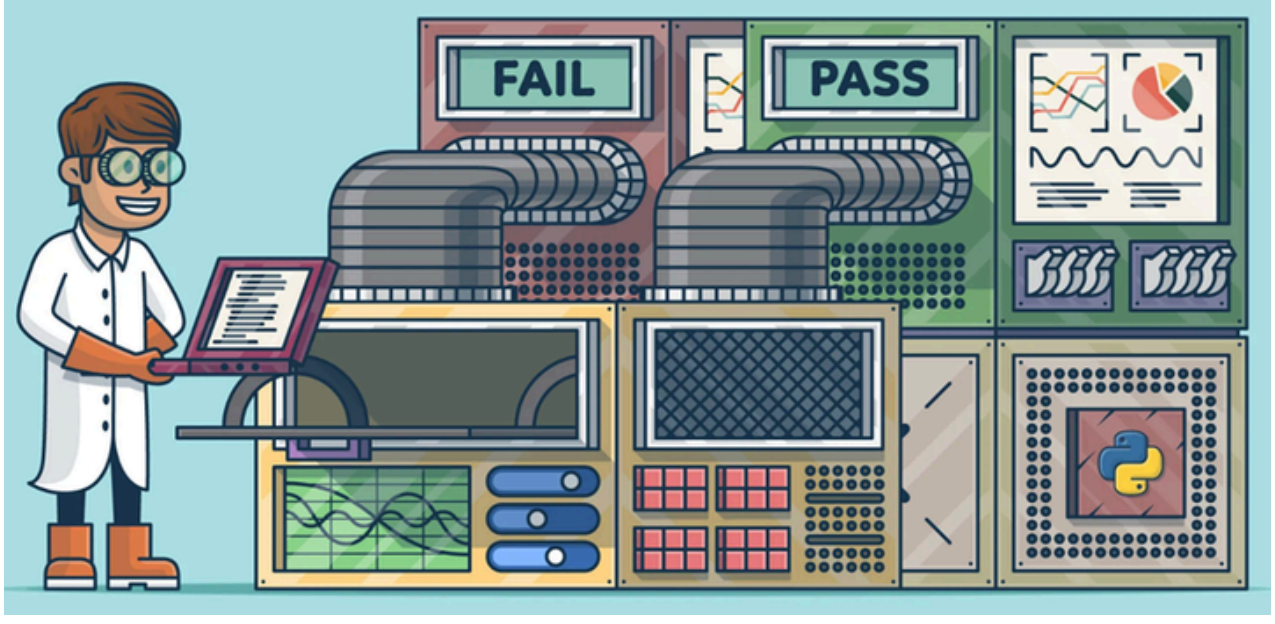
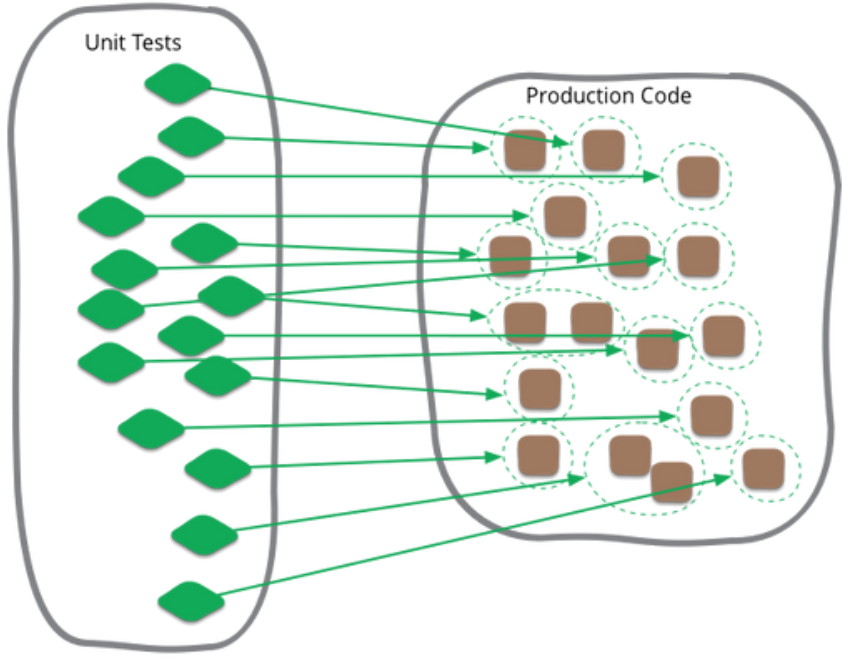
Unit Testing - Kiểm thử đơn vị

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



Unit Testing là gì ?

- Kiểm thử đơn vị là quá trình kiểm thử từng thành phần nhỏ nhất của phần mềm.
- Đảm bảo từng hàm, module hoạt động đúng như mong đợi.
- Thường được viết bởi lập trình viên trong giai đoạn phát triển.



Lợi ích của Unit Testing

- Phát hiện lỗi sớm trong quá trình phát triển.
- Giúp tái cấu trúc code dễ dàng hơn.
- Tăng cường tính ổn định của phần mềm.
- Hỗ trợ quá trình CI/CD (Tích hợp và triển khai liên tục).

Đặc điểm của một Unit Test tốt

- Nhanh chóng: Chạy nhanh và hiệu quả.
- Độc lập: Không phụ thuộc vào test khác.
- Có thể tái sử dụng và bảo trì.
- Xác định rõ đầu vào và đầu ra mong đợi.

Chu kỳ phát triển với Unit Testing

1. Viết test case trước (TDD - Test Driven Development).
2. Viết code để pass test.
3. Chạy test, kiểm tra kết quả.
4. Sửa lỗi nếu có, tối ưu code.
5. Lặp lại chu kỳ.

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 @DisplayName để giải thích rõ ràng.

- Để mô tả rõ hơn về test case, ta sử dụng @DisplayName

- Ví dụ:

```
@Test
@DisplayName("Should return student when given valid input")
void createStudent_ValidInput_ReturnsStudent() {
    // Test logic
}
```

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

3. 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);
    });
}
```

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

6. Đảm bảo Test có tốc độ nhanh và độc lập.

- Không phụ thuộc vào external API, Database, Persistent storage.
- Mỗi test case phải chạy độc lập, không bị phụ thuộc vào test case khác.

7. Kiểm tra code coverage & tối

- Sử dụng JaCoCo để tính toán code coverage.
- Kiểm tra line code, branch code, tránh viết test case mà chỉ để đạt tỉ lệ coverage cao mà không thực sự kiểm tra tính logic.

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.