ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

Báo cáo về unit test

Teachers:
Nguyễn Lê Hoàng Dũng

Members:

Names	MSSV
Lê Minh Nhật	22127308
Nguyễn Hồng Phúc	22127333

1. Các loại Code Coverage quan trọng

1.1. Line Coverage

- **Kiểm tra tỷ lệ dòng code được thực thi trong unit test.**
- **Mục tiêu là đảm bảo càng nhiều dòng code càng tốt được chạy trong quá trình kiểm thử.**
- **Không đảm bảo tất cả các nhánh logic được kiểm thử, chỉ đo lường mức độ thực thi của từng dòng code.**
- **Phù hợp để phát hiện phần code chưa được kiểm tra nhưng không đủ để đảm bảo tất cả trường hợp logic đã được test.**

Ví dụ:

```
public int add(int a, int b) {
    return a + b;
}
```

Test case:

```
@Test
public void testAdd() {
    assertEquals(5, add(2, 3));
}
```

Line Coverage sẽ là 100% nếu dòng return a + b; được thực thi.

1.2. Branch Coverage

- **Đảm bảo tất cả nhánh (if, else, switch) được kiểm thử ít nhất một lần.**
- **Đánh giá xem mỗi điều kiện nhánh có được thực hiện cả ở trạng thái đúng và sai không.**
- **Giúp phát hiện các nhánh logic chưa được xử lý trong code, đảm bảo rằng không có lỗi tiềm ẩn ở các nhánh chưa được test.**

Ví dụ:

```
public String checkNumber(int num) {
    if (num > 0) return "Positive";
    else return "Negative";
```

```
}
```

Test cases:

```
@Test
public void testCheckNumber() {
    assertEquals("Positive", checkNumber(5)); // Kiểm tra nhánh num > 0
    assertEquals("Negative", checkNumber(-3)); // Kiểm tra nhánh num <= 0
}
```

Branch Coverage sẽ đạt 100% vì cả hai nhánh đều được kiểm thử.

1.3. Function Coverage

- **Xác minh tất cả hàm/method được gọi ít nhất một lần trong unit test.**
- **Giúp đảm bảo không có hàm bị bỏ qua.**
- **Không đảm bảo tất cả dòng code trong hàm đó được thực thi.**
- **Thích hợp để kiểm tra mức độ sử dụng của từng hàm, nhưng cần kết hợp với Line Coverage để có kết quả đầy đủ hơn.**

Ví dụ:

```
public void printMessage() {
    System.out.println("Hello, World!");
}
```

Test case:

```
@Test
public void testPrintMessage() {
    printMessage();
}
```

Function Coverage sẽ là 100% vì hàm printMessage() đã được gọi.

1.4. Path Coverage

- **Đảm bảo tất cả các đường đi logic quan trọng đều được test.**
- **Xác minh rằng tất cả các đường đi có thể có của một hàm hoặc module đã được kiểm thử.**

- **Cung cấp mức độ bảo đảm cao hơn so với Branch Coverage vì nó kiểm tra tất cả các tổ hợp đường đi có thể xảy ra.**
- **Thường khó đạt được mức độ Path Coverage cao do số lượng đường đi có thể tăng theo cấp số nhân.**

Ví dụ:

```
public String checkEvenOdd(int num) {
    if (num % 2 == 0) {
        if (num > 0) return "Even Positive";
        else return "Even Negative";
    } else {
        if (num > 0) return "Odd Positive";
        else return "Odd Negative";
    }
}
```

Test cases:

```
@Test
public void testCheckEvenOdd() {
    assertEquals("Even Positive", checkEvenOdd(4));
    assertEquals("Even Negative", checkEvenOdd(-2));
    assertEquals("Odd Positive", checkEvenOdd(3));
    assertEquals("Odd Negative", checkEvenOdd(-5));
}
```

Path Coverage sẽ là 100% vì tất cả các đường đi logic đều được kiểm thử.

2. Mức Coverage tối thiểu

Mức độ code coverage tối thiểu phụ thuộc vào loại hệ thống và yêu cầu cụ thể của từng dự án:

2.1. Ứng dụng doanh nghiệp

- **Mức coverage phổ biến: 70-80%.**

- **Không yêu cầu quá cao vì ứng dụng doanh nghiệp thường có logic kinh doanh phức tạp và phụ thuộc vào nhiều hệ thống bên ngoài.**
- **Tập trung kiểm thử các thành phần quan trọng như dịch vụ xử lý nghiệp vụ, API endpoints.**

2.2. Ứng dụng quan trọng (y tế, tài chính)

- **Mức coverage tối thiểu: >90%.**
- **Các hệ thống quan trọng cần đảm bảo mức độ kiểm thử cao để tránh lỗi nghiêm trọng.**
- **Yêu cầu kiểm thử cả các tình huống bất thường, sai sót có thể gây thiệt hại lớn.**

2.3. Codebase Open-source

- **Mức coverage dao động: 50-70%.**
- **Thường không yêu cầu coverage quá cao vì dự án có nhiều contributor và việc đảm bảo coverage cao có thể khó khăn.**
- **Tập trung kiểm thử các module quan trọng và đảm bảo tính ổn định của thư viện.**

2.4. Phần mềm nhúng và hệ thống an toàn

- **Mức coverage lý tưởng: 95-100%.**
- **Các hệ thống nhúng trong lĩnh vực hàng không, ô tô, y tế phải đảm bảo kiểm thử toàn diện.**
- **Yêu cầu test chặt chẽ các tình huống bất thường, tránh lỗi phần mềm gây hậu quả nghiêm trọng.**

2.5. So sánh mức Coverage

Loại hệ thống	Mức Coverage dề xuất (%)
Ứng dụng doanh nghiệm	70-80
Ứng dụng tài chính, y tế	>90
Open-source projects	50-70
Hệ thống nhúng, an toàn	95-100

- **Coverage cao không đồng nghĩa với chất lượng test cao. Quan trọng nhất là đảm bảo các logic quan trọng được bao phủ.**
- **Mục tiêu không phải đạt 100% coverage, mà là đảm bảo kiểm thử những phần quan trọng nhất.**

3. Best Practices khi viết unit test

3.1. Viết Test dễ bảo trì

- **Tránh phụ thuộc vào database hoặc persistent storage.**
- **Dùng mock dependency thay vì truy xuất trực tiếp tại test.**
- **Sử dụng dependency injection để tách biệt logic kiểm thử khỏi hệ thống thực.**
- **Viết test có thể tái sử dụng bằng cách tận dụng setup/teardown trong test framework.**

3.2. Kiểm thử cả happy case và edge case

- **Happy case: Tình huống chạy thành công với đầu vào hợp lệ.**
- **Edge case:**
  - **Kiểm thử với đầu vào null hoặc rỗng.**
  - **Kiểm thử với giá trị cực đại, cực tiểu.**
  - **Kiểm thử với dữ liệu không hợp lệ hoặc sai định dạng.**
  - **Kiểm thử với dữ liệu lớn để xem hệ thống có xử lý tốt không.**

Ví dụ:

```
@Test
public void testDivide() {
   assertEquals(2, divide(10, 5)); // Happy case
   assertThrows(ArithmeticException.class, () -> divide(10, 0)); // Edge case: Chia cho 0
}
```

3.3. Tránh Test trùng lặp và phụ thuộc vào Implementation Details

- **Viết test tập trung vào hành vi (behavior) của hệ thống, không kiểm tra cách nó được triển khai.**
- **Tránh kiểm thử các method private hoặc nội bộ, thay vào đó kiểm tra thông qua API public.**
- **Không viết test phụ thuộc vào cấu trúc dữ liệu cụ thể hoặc thứ tự thực thi bên trong.**
- **Dùng parameterized tests để giảm thiểu trùng lặp.**

Ví dụ:

```
@ParameterizedTest
@CsvSource({"1,2,3", "4,5,9", "-1,-2,-3"})
public void testAddition(int a, int b, int expected) {
    assertEquals(expected, add(a, b));
}
```

3.4. Đảm bảo Unit Test độc lập và nhanh chóng

- **Unit test không nên phụ thuộc vào trạng thái của các test khác.**
- **Tránh sử dụng file hệ thống, network hoặc database thực trong test.**
- **Đảm bảo test có thể chạy nhanh bằng cách tối ưu logic kiểm thử.**

3.5. Cập nhật Test khi Code thay đổi

- **Khi code thay đổi, test cũng cần được cập nhật tương ứng.**
- **Nếu test bị lỗi sau khi thay đổi code, xem xét lại xem lỗi do test chưa đúng hay do code thực sự có bug.**
- **Dùng Continuous Integration (CI) để tự động chạy test mỗi khi có thay đổi.**

4. Kết luận

- **Code coverage là chỉ số quan trọng, nhưng chế đỉnh coverage cao không đảm bảo chất lượng test.**
- **Các loại coverage quan trọng gồm Line Coverage, Branch Coverage, Function Coverage, Path Coverage.**
- **Mức coverage được khuyến nghị từ 70-80% trong hầu hết các hệ thống.**
- **Nên tập trung vào best practices như viết test độc lập, kiểm thử cả edge case, tránh test implementation details.**

Nguồn tham khảo:

- **"Unit Testing Best Practices" - Martin Fowler**
- **"Effective Unit Testing" - Lasse Koskela**
- **AI**