BỘ GIÁO DỤC VÀ ĐÀO TẠO
ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

***

BÁO CÁO

Môn: Thiết kế phần mềm

Unit Test Coverage & Best Practice

GVHD: Thầy Trần Duy Thảo
SVTH: Nguyễn Nhật Tân -22120325
Trần Nhật Tân -22120328
Lương Thị Diệu Thảo - 22120337

Thủ Đức, ngày 02 tháng 04 năm 2025

Mục Lục

A. TÌM HIỂU VỀ UNIT TEST COVERAGE

Unit Test Coverage là một chỉ số quan trọng để đánh giá mức độ kiểm thử của mã nguồn. Nó giúp xác định những phần của mã nguồn chưa được kiểm thử và đảm bảo rằng các logic quan trọng của ứng dụng được kiểm tra đầy đủ.

a) Line Coverage

Line Coverage là một trong những loại coverage cơ bản nhất trong unit testing. Nó kiểm tra tỷ lệ phần trăm số dòng mã trong chương trình được thực thi trong quá trình kiểm thử. Line Coverage đánh giá mức độ bao phủ của mã nguồn qua việc xác định xem các dòng mã đã được thực thi hay chưa trong suốt quá trình kiểm thử.

1. Cách hoạt động của Line Coverage:

Mỗi dòng mã trong một chương trình được đánh dấu là "đã được kiểm thử" nếu dòng mã đó được thực thi ít nhất một lần trong quá trình kiểm thử. Điều này có nghĩa là nếu một dòng mã chỉ được thực thi khi một điều kiện cụ thể xảy ra (ví dụ: một câu lệnh if có thể không được thực thi nếu điều kiện không đúng), thì dòng mã đó không được tính vào coverage.

Ví dụ:

```
1  public void checkNumber(int number) {
2      if (number > 0) {
3          System.out.println("Positive");
4      } else {
5          System.out.println("Negative");
6      }
7  }
```

Trong trường hợp kiểm thử với number = 10, chỉ có dòng mã trong nhánh if được thực thi, dòng trong nhánh else không được kiểm thử.

Line Coverage sẽ tính toán tỷ lệ phần trăm của các dòng mã được thực thi so với tổng số dòng trong phương thức.

2. Ưu và nhược điểm của Line Coverage:

   Ưu điểm: Line Coverage đơn giản và dễ sử dụng, giúp xác định nhanh chóng các dòng mã chưa được kiểm thử.

   Nhược điểm: Line Coverage không đảm bảo rằng tất cả các nhánh hoặc tình huống của chương trình đã được kiểm thử. Dòng mã có thể được thực thi nhưng không có nghĩa là tất cả các logic quan trọng đã được kiểm tra.

b) Branch Coverage

Branch Coverage là một mức độ kiểm thử cao hơn Line Coverage, bởi vì nó không chỉ kiểm tra xem một dòng mã có được thực thi hay không mà còn kiểm tra xem tất cả các nhánh (branch) của chương trình có được kiểm thử đầy đủ hay không. Mỗi câu lệnh điều kiện (if, else, switch) trong mã tạo ra ít nhất một nhánh, và Branch Coverage yêu cầu rằng tất cả các nhánh này đều phải được kiểm tra trong quá trình kiểm thử.

1. Cách hoạt động của Branch Coverage:

   Branch Coverage kiểm tra tất cả các nhánh của chương trình bằng cách đảm bảo rằng mỗi điều kiện logic có thể có tất cả các giá trị (thường là đúng và sai trong trường hợp các câu lệnh if, else). Điều này giúp phát hiện các lỗi logic mà có thể không xuất hiện trong Line Coverage.

   Ví dụ:

```
1  public void checkEvenOdd(int number) {
2      if (number % 2 == 0) {
3          System.out.println("Even");
4      } else {
5          System.out.println("Odd");
6      }
7  }
```

   Để đảm bảo Branch Coverage, bài kiểm thử cần kiểm tra cả hai nhánh của câu lệnh if. Nếu chỉ kiểm tra một nhánh (ví dụ chỉ thử với number = 2), thì Branch Coverage sẽ không đạt 100%.

Trong trường hợp này, cần ít nhất hai bài kiểm thử: một với số chẵn (number = 2) và một với số lẻ (number = 3).

2. Ưu và nhược điểm của Branch Coverage:

   Ưu điểm: Branch Coverage đảm bảo rằng tất cả các nhánh của chương trình đều được kiểm thử. Điều này giúp phát hiện các lỗi trong các điều kiện của câu lệnh điều kiện mà chỉ kiểm thử Line Coverage không thể kiểm tra.

   Nhược điểm: Branch Coverage không thể phát hiện các lỗi liên quan đến các kết hợp phức tạp của các điều kiện (như các điều kiện kết hợp trong if-else).

c) Function Coverage

Function Coverage (hoặc Method Coverage) là loại coverage đảm bảo rằng tất cả các hàm hoặc phương thức trong chương trình đều được gọi và kiểm thử ít nhất một lần. Điều này giúp đảm bảo rằng các phương thức không bị bỏ sót trong quá trình kiểm thử.

1. Cách hoạt động của Function Coverage:

   Khi một bài kiểm thử gọi một phương thức, phương thức đó được đánh dấu là "được kiểm thử". Điều này giúp đảm bảo rằng tất cả các phương thức trong mã nguồn đều được kiểm tra ít nhất một lần, từ đó giúp phát hiện các lỗi có thể xảy ra trong các hàm chưa được kiểm thử.

   Ví dụ:

1	public class Calculator {
2	public int add(int a, int b) {
3	return a + b;
4	}
5	public int subtract(int a, int b) {
6	return a - b;
7	}
	}

Function Coverage yêu cầu rằng các phương thức add và subtract đều phải được kiểm thử ít nhất một lần. Việc gọi một trong các phương thức này trong bài kiểm thử đảm bảo rằng chúng được bao phủ.

2. Ưu và nhược điểm của Function Coverage::

   Ưu điểm: Function Coverage giúp đảm bảo rằng tất cả các hàm trong chương trình đều được kiểm thử.

   Nhược điểm: Function Coverage không kiểm tra các điều kiện bên trong hàm hoặc các chi tiết của logic phương thức, vì vậy nó có thể không đảm bảo rằng tất cả các logic của phương thức đã được kiểm tra.

d) Path Coverage

Path Coverage là mức độ coverage phức tạp hơn, nó yêu cầu kiểm tra tất cả các đường đi logic có thể có trong một chương trình. Mỗi đường đi là một chuỗi các lệnh được thực thi trong một chương trình từ đầu đến cuối, bao gồm tất cả các nhánh và quyết định điều kiện..

1. Cách hoạt động của Path Coverage:

   Path Coverage kiểm tra tất cả các chuỗi các lệnh có thể xảy ra trong chương trình, bao gồm tất cả các nhánh và quyết định trong mã. Điều này giúp phát hiện lỗi trong các tình huống rất đặc biệt mà các loại coverage khác có thể không bao phủ.

   Ví dụ:

1	public void processOrder(int quantity) {
2	if (quantity > 10) {
3	discount();
4	}
5	if (quantity < 5) {
6	charge();
7	}
	}

Trong trường hợp này, có thể có nhiều đường đi: (1) quantity > 10 và quantity < 5, (2) chỉ có quantity > 10, (3) chỉ có quantity < 5, hoặc (4) không có điều kiện nào được thực thi. Path Coverage yêu cầu tất cả các đường đi này đều phải được kiểm thử.

2. Ưu và nhược điểm của Path Coverage:

   Ưu điểm: Path Coverage cung cấp mức độ kiểm thử rất chi tiết và đảm bảo rằng mọi kết hợp có thể xảy ra trong mã đều được kiểm thử.

   Nhược điểm: Vì mỗi chương trình có thể có số lượng đường đi rất lớn, việc kiểm tra toàn bộ các đường đi có thể rất tốn kém và khó khăn, đặc biệt là đối với các chương trình phức tạp.

B. MỨC COVERAGE TỐI THIỂU

a) Tùy thuộc vào loại ứng dụng mà mức coverage có thể khác nhau, ví dụ:

- Ứng dụng quan trọng như y tế, tài chính, ... thường yêu cầu coverage cao (~80-90%)
- Ứng dụng web/e-commerce có thể duy trì mức coverage khoảng 60-80%
- Game development thường không tập trung vào unit test mà thay vào đó là integration và playtest

b) Tiêu chuẩn của industry

- Backend API: 70-80%, phải đảm bảo kiểm thử các logic quan trọng
- Frontend: 50-65%, do UI khó kiểm thử tự động
- Critical Services (Authentication, Payment, Security): 85-95%

c) Mức coverage cho ứng dụng Quản lý sinh viên

1. Backend:
   - Quản lý sinh viên: 70-85%, các API CRUD cần test đầy đủ để tránh lỗi logic
   - Quản lý môn học & Đăng ký: 75-85%, các quy tắc nghiệp vụ cần được test kỹ càng
   - Xử lý điểm số & Bảng điểm: 80-90%, các số liệu cần được tính toán kỹ lưỡng, tránh sai sót

2. Frontend:

- Quản lý sinh viên: 60-75%, form validation, xử lý lỗi khi nhập liệu sai cũng cần được kiểm thử
- Quản lý khóa học & Đăng ký: 65-75%, kiểm thử giao diện chọn môn học, xử lý lỗi trùng môn
- Hiển thị bảng điểm: 50-65%, UI-heavy, có thể ưu tiên test integration

3. Database & Integration:

Integration test giữa Backend & Database: 70-80%

C. BEST PRACTICE KHI VIẾT UNIT TEST

a) Đặt tên theo tiêu chuẩn

Tên của unit test cần mô tả rõ ràng mục đích kiểm thử. Tên nên gồm 3 thông tin:

- Tên phương thức kiểm thử
- Kịch bản kiểm thử
- Kết quả mong đợi khi kịch bản được gọi

Ví dụ: Add_SingleNumber_ReturnsSameNumber()

b) Dùng cấu trúc AAA

Cấu trúc của unit test nên chia ra làm 3 phần để thuận tiện cho việc đọc:

- Arrange: Chuẩn bị dữ liệu đầu vào và cấu hình môi trường cần thiết.
- Act: Gọi phương thức hoặc function cần test.
- Assert: Kiểm tra kết quả mong đợi và kết quả thực tế. Đây là phần quyết định test fail hoặc pass.

Ví dụ:

```
1  public void IsPrime_WhenNumberIsPrime_ReturnsTrue()
2  {
3    // Arrange
4    var primeUtils = new PrimeUtils();
5    int number = 5;
6    bool expected = true;
7    // Act
```

8	var actual = primeUtils.IsPrime(number);
9	// Assert
10	Assert.Equal(expected, actual);
11	}

c) Viết test dễ hiểu, dễ duy trì

- Tránh đặt giá trị cứng (magic string) vào unit test.
- Tránh đặt các hàm tự tạo hoặc lệnh điều kiện (if, else, while,…), logic không cần thiết trong unit test để tránh xảy ra lỗi trong unit test thay vì phát hiện lỗi trong mã nguồn.

d) Viết test độc lập và có thể chạy riêng lẻ

- Mỗi unit test nên kiểm tra một phần nhỏ của code mà không phụ thuộc vào test khác.
- Tránh shared state giữa các test để không ảnh hưởng đến kết quả.
- Không viết test trùng lặp hoặc quá phụ thuộc vào implementation details.
- Khi unit test có sự phụ thuộc vào database, API bên ngoài, hoặc hệ thống khác, nên sử dụng mocking để cô lập logic cần test.

e) Test trường hợp biên (edge cases)

Kiểm tra các trường hợp đặc biệt như:

- Input rỗng, null.
- Giá trị âm, giá trị cực đại/cực tiểu.
- Dữ liệu có định dạng không hợp lệ.

f) Tránh test không ổn định (flaky test)

- Đảm bảo test không bị ảnh hưởng bởi yếu tố bên ngoài như thời gian, database, hoặc kết nối mạng.
- Không để các test phụ thuộc vào thứ tự chạy.

g) Đảm bảo độ bao phủ (code coverage)

- Viết test cho cả trường hợp thành công và thất bại.
- Nên ưu tiên test những phần quan trọng như:
  + Business logic

+ Xử lý lỗi
+ Các method có điều kiện rẽ nhánh

h) Dùng hàm helper để cài đặt

Nếu có một object được sử dụng trong nhiều unit test, cần tạo phương thức helper. Điều này giúp giảm lặp lại những đoạn code giống nhau và cho phép test dễ bảo trì hơn khi có thay đổi.

Ví dụ:

```
1  [Theory]
2  [InlineData(2, 3, 5)]
3  [InlineData(11, 5, 16)]
4
5  public void Add_TwoNumbers_ReturnsSum(int number1, int number2, int expected)
6  {
7      // Arrange
8      var calculator = CreateCalculator();
9
10     // Act
11     var actual = calculator.Add(number1, number2);
12
13     // Assert
14     Assert.Equal(expected, actual);
15 }
16
17 private Calculator CreateCalculator()
18 {
19     return new Calculator();
20 }
```

i) Trong một unit test chỉ nên kiểm tra một vấn đề

- Mỗi unit test chỉ nên có 1 act và assert 1 object
- Trong một unit test, nếu assert nhiều hơn 1 object, unit test có thể đang kiểm tra nhiều hơn 1 vấn đề

Ví dụ:

```
1  [Theory]
2  [InlineData(2, 3, 5)]
3  [InlineData(11, 5, 16)]
4  public void Add_TwoNumbers_ReturnsSum(int number1, int number2, int expected)
5  {
6      // Arrange
7      var calculator = new Calculator();
8      // Act
9      var actual = calculator.Add(number1, number2);
10     // Assert
11     Assert.Equal(expected, actual);
12 }
13 [Theory]
14 [InlineData(2, 3, -1)]
15 [InlineData(11, 5, 6)]
16 public void Subtract_TwoNumbers_ReturnsDifference(int number1, int number2, int expected)
17 {
18     // Arrange
19     var calculator = new Calculator();
20 // Act
21     var actual = calculator.Subtract(number1, number2);
22 // Assert
23 Assert.Equal(expected, actual);
```

24	}

j) Unit test cần nhanh

Unit test nhanh giúp tăng tốc độ phát triển mã nguồn và đảm bảo chạy được thường xuyên. Unit test nhanh cũng giúp phát hiện lỗi và sửa lỗi sớm hơn.

Để unit test nhanh cần:

- Đơn giản
- Độc lập với các test khác
- Sử dụng mocking để cô lập logic cần test