Unit Test Coverage và Best Practices

Báo cáo nghiên cứu

Ngày 28 tháng 2 năm 2025

Mục lục

1 Giới thiệu

Unit Testing đóng vai trò quan trọng trong quá trình phát triển phần mềm, đảm bảo tính ổn định và độ tin cậy của code. Báo cáo này trình bày các tiêu chuẩn unit test coverage trong industry, bao gồm các loại code coverage, mức coverage tối thiểu, và các best practices khi viết unit test.

2 Các loại Code Coverage

Code coverage là thước đo đánh giá mức độ mà mã nguồn của ứng dụng được kiểm tra bởi các unit test. Dưới đây là các loại code coverage quan trọng được sử dụng rộng rãi trong industry:

2.1 Line Coverage

Line Coverage (hay còn gọi là Statement Coverage) đo lường tỷ lệ phần trăm các dòng code được thực thi trong quá trình chạy unit test.

Ví dụ về Line Coverage

```
public int calculateDiscount(int amount) {
    int discount = 0;                        // Line 1
    if (amount > 1000) {                     // Line 2
        discount = amount * 10 / 100;        // Line 3
    }
    return discount;                         // Line 4
}
```

Nếu unit test chỉ kiểm tra trường hợp `amount <= 1000`, thì line 3 không được thực thi, và line coverage sẽ là 75% (3/4 dòng được kiểm tra).

Ưu điểm:

- Dễ hiểu và đo lường
- Cung cấp cái nhìn tổng quan về mức độ kiểm thử

Hạn chế:

- Không đảm bảo kiểm tra tất cả các luồng logic
- Có thể bỏ qua các trường hợp ngoại lệ hoặc biên

2.2 Branch Coverage

Branch Coverage đo lường tỷ lệ các nhánh điều kiện logic (như `if`, `else`, `switch-case`) được thực thi. Branch coverage cung cấp cái nhìn sâu hơn so với line coverage vì nó đảm bảo tất cả các luồng thực thi có thể xảy ra đều được kiểm tra.

Ví dụ về Branch Coverage

```
public String evaluateGrade(int score) {
    if (score >= 90) {                  // Branch 1: score >= 90
        return "A";                     // Line 1
    } else if (score >= 80) {           // Branch 2: 80 <=
    score < 90
        return "B";                     // Line 2
    } else if (score >= 70) {           // Branch 3: 70 <=
    score < 80
        return "C";                     // Line 3
    } else {                           // Branch 4: score < 70
        return "F";                     // Line 4
    }
}
```

Để đạt 100% branch coverage, cần phải có unit test kiểm tra tất cả 4 nhánh điều kiện.

Ưu điểm:

- Đảm bảo tất cả các nhánh điều kiện được thực thi
- Phát hiện các lỗi liên quan đến logic điều kiện

Hạn chế:

- Không đảm bảo tất cả các tổ hợp điều kiện phức tạp được kiểm tra
- Không xem xét thứ tự thực thi của các nhánh

2.3 Function Coverage

Function Coverage đo lường tỷ lệ các hàm và phương thức trong chương trình được gọi ít nhất một lần trong quá trình thực thi unit test.

Ví dụ về Function Coverage

```
public class Calculator {
    public int add(int a, int b) {          // Function 1
        return a + b;
    }

    public int subtract(int a, int b) {     // Function 2
        return a - b;
    }

    public int multiply(int a, int b) {     // Function 3
        return a * b;
    }

    public int divide(int a, int b) {       // Function 4
        if (b == 0) {
            throw new
                ArithmeticException("Division by zero");
        }
        return a / b;
    }
}
```

Nếu các unit test chỉ gọi các hàm `add` và `subtract`, function coverage sẽ là 50% (2/4 hàm được kiểm tra).

Ưu điểm:

- Dễ dàng nhận biết các chức năng chưa được kiểm thử
- Giúp phát hiện "dead code"(mã không được sử dụng)

Hạn chế:

- Không cung cấp thông tin về chất lượng kiểm thử bên trong mỗi hàm
- Không đảm bảo tất cả các đường đi logic trong hàm được kiểm tra

2.4 Path Coverage

Path Coverage là loại coverage toàn diện nhất, đo lường tỷ lệ các đường đi thực thi có thể xảy ra trong chương trình. Một đường đi là một chuỗi cụ thể các câu lệnh từ điểm bắt đầu đến điểm kết thúc.

Ví dụ về Path Coverage

```
public String classifyTriangle(int a, int b, int c) {
    if (a <= 0 || b <= 0 || c <= 0) {
        return "Invalid";
    }

    if (a + b <= c || a + c <= b || b + c <= a) {
        return "Not a triangle";
    }

    if (a == b && b == c) {
        return "Equilateral";
    } else if (a == b || b == c || a == c) {
        return "Isosceles";
    } else {
        return "Scalene";
    }
}
```

Hàm này có nhiều đường đi khác nhau dựa trên các điều kiện. Để đạt được 100% path coverage, cần kiểm tra tất cả các tổ hợp điều kiện có thể.

Ưu điểm:

- Cung cấp coverage toàn diện nhất
- Phát hiện các lỗi phức tạp và tương tác giữa các điều kiện

Hạn chế:

- Khó đạt được 100% path coverage trong các hệ thống phức tạp
- Số lượng đường đi có thể tăng theo cấp số nhân với số lượng nhánh điều kiện

3 Mức Coverage Tối thiểu trong Industry

Mức độ code coverage phù hợp phụ thuộc vào nhiều yếu tố như độ phức tạp của hệ thống, tính chất của ứng dụng, và các yêu cầu về chất lượng. Dưới đây là một số tiêu chuẩn thông dụng trong industry:

3.1 Tiêu chuẩn theo loại ứng dụng

Loại ứng dụng	Line Coverage	Branch Coverage	Function Coverage
Ứng dụng mission-critical	90-100%	80-100%	100%
Ứng dụng enterprise	70-90%	60-80%	80-100%
Ứng dụng thông thường	60-80%	50-70%	70-90%

Bảng 1: Mức coverage tối thiểu theo loại ứng dụng

3.2 Tiêu chuẩn theo tổ chức

Nhiều tổ chức và framework có tiêu chuẩn riêng về code coverage:

- **Google**: Không áp dụng mức coverage cứng nhắc, nhưng khuyến khích khoảng 60% line coverage cho hầu hết các dự án.
- **Microsoft**: Khuyến nghị ít nhất 70% line coverage và 80% branch coverage cho các dự án quan trọng.
- **Sonar**: Đề xuất ngưỡng tối thiểu 80% line coverage cho các dự án mới.
- **CISQ (Consortium for IT Software Quality)**: Khuyến nghị ít nhất 70% line coverage.

3.3 Cân nhắc khi xác định mức coverage phù hợp

- **Chất lượng quan trọng hơn số lượng**: Coverage cao không đảm bảo code không có lỗi. Tập trung vào việc kiểm thử logic quan trọng.
- **Quy tắc 80/20**: 80% effort nên tập trung vào 20% code quan trọng nhất (core functionality, business logic).
- **Cân bằng ROI (Return on Investment)**: Đạt được 100% coverage thường đòi hỏi nỗ lực không tương xứng với lợi ích.
- **Tăng dần theo thời gian**: Đặt mục tiêu coverage tăng dần thay vì áp đặt ngay mức cao.

4 Best Practices khi viết Unit Test

4.1 Phương pháp FIRST

Unit test nên tuân theo nguyên tắc FIRST:

- **Fast**: Test nên chạy nhanh để có thể chạy thường xuyên.
- **Isolated/Independent**: Test không nên phụ thuộc vào nhau hoặc thứ tự chạy.
- **Repeatable**: Test phải cho kết quả nhất quán khi chạy nhiều lần.
- **Self-validating**: Test tự động xác định pass/fail mà không cần kiểm tra thủ công.
- **Timely**: Test được viết đúng thời điểm (trước hoặc cùng lúc với code).

4.2 Độc lập với External Dependencies

- **Sử dụng Mock Objects**: Tạo các đối tượng giả để thay thế các dependency bên ngoài như database, API, hay file system.

Ví dụ sử dụng Mockito trong Java

```
public class UserService {
    private UserRepository repository;

    public UserService(UserRepository repository) {
        this.repository = repository;
    }

    public User findUserById(long id) {
        return repository.findById(id);
    }
}

@Test
public void testFindUserById() {
    UserRepository mockRepo =
    Mockito.mock(UserRepository.class);

    User expectedUser = new User(1L, "John Doe");

    Mockito.when(mockRepo.findById(1L)).thenReturn(expectedUser);

    UserService service = new UserService(mockRepo);
    User actualUser = service.findUserById(1L);

    assertEquals(expectedUser, actualUser);
}
```

- **Sử dụng In-memory Database**: Thay vì kết nối đến database thật, sử dụng database trong bộ nhớ như H2, SQLite cho testing.
- **Dependency Injection**: Thiết kế code để dễ dàng inject các dependency, giúp unit test dễ dàng hơn.

4.3 Kiểm thử Happy Path và Edge Cases

- **Happy Path**: Kiểm tra hành vi của code trong điều kiện lý tưởng khi input hợp lệ.
- **Edge Cases**: Kiểm tra các trường hợp đặc biệt và giá trị biên:

– Giá trị null hoặc empty
– Giá trị âm, zero, giá trị rất lớn
– Danh sách rỗng hoặc có một phần tử
– Input không hợp lệ và xử lý ngoại lệ

Ví dụ kiểm thử Happy Path và Edge Cases

```
// Happy Path Test
@Test
public void
    divideNumbers_ValidInput_ReturnsCorrectResult() {
    Calculator calculator = new Calculator();
    double result = calculator.divide(10, 2);
    assertEquals(5.0, result, 0.0001);
}

// Edge Cases Tests
@Test
public void divideNumbers_ZeroDivisor_ThrowsException() {
    Calculator calculator = new Calculator();
    assertThrows(ArithmeticException.class, () -> {
        calculator.divide(10, 0);
    });
}

@Test
public void
    divideNumbers_NegativeNumbers_ReturnsCorrectResult()
    {
    Calculator calculator = new Calculator();
    double result = calculator.divide(-10, -2);
    assertEquals(5.0, result, 0.0001);
}

@Test
public void
    divideNumbers_LargeNumbers_ReturnsCorrectResult() {
    Calculator calculator = new Calculator();
    double result = calculator.divide(Integer.MAX_VALUE,
    1);
    assertEquals(Integer.MAX_VALUE, result, 0.0001);
}
```

4.4 Tránh Anti-patterns trong Unit Testing

- **Test trùng lặp**: Không nên có nhiều test kiểm tra cùng một chức năng.
- **Quá phụ thuộc vào Implementation Details**: Test nên kiểm tra hành vi (behavior) thay vì chi tiết triển khai, giúp code dễ dàng refactor mà không cần thay đổi test.
- **Test quá phức tạp**: Mỗi test nên tập trung kiểm tra một tính năng cụ thể thay vì cố gắng kiểm tra nhiều chức năng cùng lúc.
- **Non-deterministic Tests (Flaky Tests)**: Tránh các test không ổn định, đôi khi pass, đôi khi fail do phụ thuộc vào điều kiện bên ngoài.

4.5 Naming Conventions và Cấu trúc Test

- **Đặt tên Test rõ ràng**: Tên test nên mô tả chính xác chức năng được kiểm tra. Format phổ biến: `[MethodName]_[Scenario]_[ExpectedResult]`
- **Cấu trúc AAA (Arrange-Act-Assert)**: Tổ chức test theo 3 phần:
  - **Arrange**: Chuẩn bị dữ liệu và đối tượng cần thiết
  - **Act**: Thực hiện hành động cần kiểm tra
  - **Assert**: Kiểm tra kết quả thực tế so với kết quả mong đợi

Ví dụ cấu trúc AAA

```
@Test
public void
    calculateTotal_ItemsWithTax_ReturnsSumWithTax() {
    // Arrange
    ShoppingCart cart = new ShoppingCart();
    cart.addItem(new Item("Book", 10.0));
    cart.addItem(new Item("Pen", 5.0));
    double taxRate = 0.1; // 10% tax

    // Act
    double total = cart.calculateTotal(taxRate);

    // Assert
    assertEquals(16.5, total, 0.0001); // (10 + 5) * 1.1
    = 16.5
}
```

5 Công cụ đo lường Code Coverage

Dưới đây là một số công cụ phổ biến để đo lường code coverage:

6 Kết luận

Unit testing với coverage đầy đủ là một phần quan trọng trong quy trình phát triển phần mềm hiện đại. Tuy nhiên, mục tiêu không nên chỉ là đạt được tỷ lệ coverage cao mà còn là viết những unit test có ý nghĩa, phát hiện lỗi và đảm bảo hành vi đúng của hệ thống.

Các best practices như sử dụng mock objects, kiểm thử cả happy path và edge cases, tuân thủ nguyên tắc FIRST, và tránh các anti-patterns sẽ giúp nâng cao chất lượng của unit test.

Việc xác định mức coverage tối thiểu phù hợp nên dựa trên tính chất của dự án, yêu cầu về chất lượng, và cân nhắc về ROI. Thay vì cố gắng đạt được 100% coverage, tập trung vào việc kiểm thử tốt những phần quan trọng của hệ thống.