TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN – ĐHQG HCM

KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO TÌM HIỂU VỀ UNIT TESTING

MÔN THIẾT KẾ PHẦN MỀM

LỚP 22_3

NHÓM ARCEUS:

22120271 - Dương Hoàng Hồng Phúc

22120286 - Lê Võ Minh Phương

22120287 - Nguyễn Mạnh Phương

22120326 - Nguyễn Trường Tân

CHƯƠNG TRÌNH CHÍNH QUY - KHÓA 2022

MỤC LỤC

I. Unit Testing

1. Giới thiệu về Unit Testing

Unit testing là quá trình kiểm thử các đơn vị nhỏ nhất của mã nguồn, thường là các hàm hoặc phương thức, để đảm bảo rằng chúng hoạt động đúng như mong đợi. Mục tiêu chính của *unit testing* là phát hiện lỗi trong giai đoạn phát triển, trước khi mã được triển khai vào môi trường sản xuất.

2. Tầm quan trọng của Unit Testing

Phát hiện lỗi sớm: Unit testing giúp phát hiện lỗi ngay từ giai đoạn phát triển, giảm thiểu chi phí sửa chữa sau này. Khi lỗi được phát hiện sớm, lập trình viên có thể dễ dàng xác định nguyên nhân và khắc phục mà không cần phải xem xét toàn bộ mã nguồn.

Cải thiện chất lượng mã nguồn: Việc viết unit tests khuyến khích lập trình viên viết mã rõ ràng và dễ bảo trì hơn. Khi có các bài kiểm thử, lập trình viên sẽ có xu hướng viết mã theo cách dễ hiểu và dễ kiểm thử hơn.

Tạo tài liệu cho mã nguồn: Các bài kiểm thử có thể được xem như tài liệu mô tả cách sử dụng mã. Khi một lập trình viên mới tham gia vào dự án, họ có thể tham khảo các bài kiểm thử để hiểu cách mà mã hoạt động.

Giảm thiểu rủi ro: Unit testing giúp giảm thiểu rủi ro khi thay đổi mã. Khi có các bài kiểm thử, lập trình viên có thể tự tin thực hiện các thay đổi mà không lo ngại về việc làm hỏng các chức năng đã hoạt động tốt trước đó.

3. Mục Tiêu của Báo cáo

Mục tiêu của báo cáo này là tổng hợp và phân tích các tiêu chuẩn unit test coverage trong ngành công nghiệp, cũng như các best practices trong việc viết unit tests. Báo cáo sẽ cung cấp cái nhìn tổng quan về các loại coverage, mức độ coverage tối thiểu cần thiết, và các công cụ hỗ trợ kiểm thử.

4. Tổng quan về Code coverage

Code coverage là một chỉ số đo lường mức độ mà mã nguồn của bạn được kiểm thử bởi các bài kiểm thử. Các loại coverage chính bao gồm:

1. *Line Coverage*: Tỷ lệ phần trăm các dòng mã được thực thi trong quá trình kiểm thử. Line coverage giúp xác định xem có bao nhiêu phần của mã đã được kiểm thử. Tuy nhiên, chỉ số này không đảm bảo rằng tất cả các logic trong mã đều được kiểm thử.

2. *Branch Coverage*: Đảm bảo rằng tất cả các nhánh (if, else, switch) trong mã đều được kiểm thử. Branch coverage cung cấp cái nhìn sâu hơn về cách mà mã xử lý các điều kiện khác nhau. Điều này rất quan trọng để đảm bảo rằng tất cả các tình huống có thể xảy ra đều được kiểm thử.

3. *Function Coverage*: Xác minh rằng tất cả các hàm/method được gọi ít nhất một lần. Function coverage giúp đảm bảo rằng tất cả các phần của mã đều được kiểm thử, nhưng không đảm bảo rằng tất cả các đường đi trong hàm đều được kiểm thử.

4. *Path Coverage*: Đảm bảo rằng tất cả các đường đi logic quan trọng trong mã đều được kiểm thử. Path coverage là chỉ số mạnh mẽ nhất, nhưng cũng khó đạt được do số lượng đường đi có thể rất lớn trong các hàm phức tạp.

II. Code coverage

1. Hiểu về Code coverage

Trong testing, *code coverage* là một tiêu chuẩn đo lường nhằm đánh giá tỷ lệ source code đã được test. Tiêu chuẩn này giúp đánh giá chất lượng của các *test suite* - tập hợp nhiều test case để test một unit. Có nhiều loại *code coverage* khác nhau và chúng phụ thuộc vào tiêu chí mà chúng đánh giá.

Để tham khảo, trong bài viết này, ta lấy ví dụ về việc chuẩn bị thành phần cho coffee như sau:

```
/* coffee.js */
function calcCoffeeIngredient(coffeeName, cup = 1) {
  let espresso, water;

  if (coffeeName === 'espresso') {
    espresso = 30 * cup;
    return { espresso };
  }

  if (coffeeName === 'americano') {
    espresso = 30 * cup; water = 70 * cup;
    return { espresso, water };
  }

  return {};
}

function isValidCoffee(name) {
  return ['espresso', 'americano', 'mocha'].includes(name);
}

module.exports = {
  calcCoffeeIngredient,
  isValidCoffee
}
```

Sau đó, sử dụng **Jest** - một test framework cho NodeJS, ta viết *test suit* để đánh giá hàm `calcCoffeeIngredient()`:

```
/*coffee.test.js*/
const {calcCoffeeIngredient} = require("./coffee.js");

describe('Coffee', () => {
```

```
  test('Should have espresso', () => {
    const result = calcCoffeeIngredient('espresso', 2);
    expect(result).toEqual({espresso: 60});
  });
  test('Should have nothing', () => {
    const result = calcCoffeeIngredient('unknow');
    expect(result).toEqual({});
  })
})
```

Sau đây là các loại *code coverage* phổ biến:

- Function coverage
- Line coverage
- Stament coverage
- Branch coverage
- Path coverage

Sử dụng công cụ phân tích có sẵn của **Jest**, ta có kết quả *code coverage* như sau:

File	**% Stmts**	**% Branch**	**% Funcs**	**% Lines**	**Uncovered Line #s**
All files	60	80	50	66.66	
coffee.js	60	80	50	66.66	10-11,18

Ta sẽ lần lượt tìm hiểu các *code coverage* và xem xét kết quả trên.

2. Các loại Code Coverage

2.1 Function coverage

Đây là loại đơn giản nhất. Function coverage đánh giá số *function* được gọi bởi *test suite*.

Trong ví dụ, ta thấy `coffee.test.js` có 1 *test suite* `Coffee`, với 2 *test cases*, và chỉ gọi một hàm `-calcCoffeeIngredient()` - trong hai hàm `calcCoffeeIngredient()` và `isValidCoffee()` của file `coffee.js`

Do đó, *function coverage* trên file `coffee.js` là 50%

2.2 Line coverage

Line coverage đánh giá tỷ lệ phần trăm số dòng code *thực thi được - excutable code lines* - mà *test suite* đã chạy qua.

Các *excutable code lines* không bao gồm các câu lệnh khai báo - *declaration statements*, như các câu lệnh khai báo biến hay hàm. Ví dụ: `let espresso, water;`, `function calcCoffeeIngredient(coffeeName, cup = 1)`

Một câu lệnh (*statement*) có thể viết trên nhiều dòng, ví dụ như dòng `module.exports` ở cuối file `coffee.js`. Tuy nhiên, miễn là nó là một câu lệnh thì nó vẫn tính như là một dòng *excutable code line*. Trong ví dụ, các *executable code lines* trong file `coffee.js` được đánh dấu như dưới đây

```
/* coffee.js */
function calcCoffeeIngredient(coffeeName, cup = 1) {
  let espresso, water;

  if (coffeeName === 'espresso') { // This
    espresso = 30 * cup; // This
    return { espresso }; // This
  }

  if (coffeeName === 'americano') { // This
    espresso = 30 * cup; water = 70 * cup; // This
    return { espresso, water }; // This
  }

  return {}; // This
}

function isValidCoffee(name) {
  return ['espresso', 'americano', 'mocha'].includes(name); // This
}

module.exports = {
  calcCoffeeIngredient,
  isValidCoffee
} // This
```

Trong tổng cộng 9 *executable code lines*, 2 *test cases* trong `coffee.test.js` đã chạy qua 7:

```
/*coffee.test.js*/
// Should have espresso test case
if (coffeeName === 'espresso') {
  espresso = 30 * cup;
  return { espresso };
}
// Should have nothing test case
if (coffeeName === 'americano') {
  // Code này đã không được chạy
}
return {};
// Cả hai
module.exports = {
  calcCoffeeIngredient,
  isValidCoffee
}
```

Như vậy Line coverage trên file `coffee.js` là 6/9 = 66,66%

2.3 Statement coverage

Khác với Line coverage - tính theo dòng, Statement coverage tính theo câu lệnh. Điểm khác biệt lớn nhất giữa 2 loại *code coverage* là Line coverage *không phân biệt nhiều câu lệnh trên một dòng*, còn Statement coverage thì *có*.

Trong ví dụ, dòng như vậy là:

```
/*coffee.js*/
espresso = 30 * cup; water = 70 * cup;
```

So với Line coverage, ta có 10 *statements*, `coffee.test.js` vẫn chỉ chạy qua 6 do điều kiện`if (coffeeName === 'americano')` không bao giờ thỏa.

Như vậy Statement coverage trên file `coffee.js` là 6/10 = 60%

2.4 Branch coverage

Branch coverage đánh giá sự bao phủ của các nhánh điều kiện trong mã nguồn, đặc biệt là các câu lệnh điều kiện như `if`, `else`, `switch`, và vòng lặp như `for`, `while`. Một nhánh điều kiện được tính là đã được kiểm tra khi cả hai nhánh của nó (như `true` và `false` trong câu lệnh `if`) đều đã được thực thi ít nhất một lần trong các bài kiểm tra.

Trong ví dụ, hàm `calcCoffeeIngredient` có 2 câu lệnh `if`:

1. `if (coffeeName === 'espresso')`
2. `if (coffeeName === 'americano')`

Cả hai đều có nhánh `true` và `false`. Tổng cộng có 5 nhánh (2 điều kiện `if` mỗi cái có 2 nhánh, và 1 nhánh mặc định).

Các nhánh:

1. `coffeeName === 'espresso'` (true)
2. `coffeeName !== 'espresso'` (false)
3. `coffeeName === 'americano'` (true)
4. `coffeeName !== 'americano'` (false)
5. `return {}` (mặc định, khi cả 2 điều kiện `if` đều false)

Trong các test case hiện tại, chỉ có 4 nhánh được chạy: nhánh `espresso` (true), nhánh `return { espresso }`, và nhánh mặc định `return {}`. Nhánh `americano` không được kiểm tra.

Như vậy Branch coverage trên file *coffee.js* là 4/5 = 80%

2.5 Path coverage

Path coverage là phương pháp khi mà tester phải duyệt qua và đánh từng dòng code. Đây là loại *code coverage* thủ công và đòi hỏi nhiều công sức.

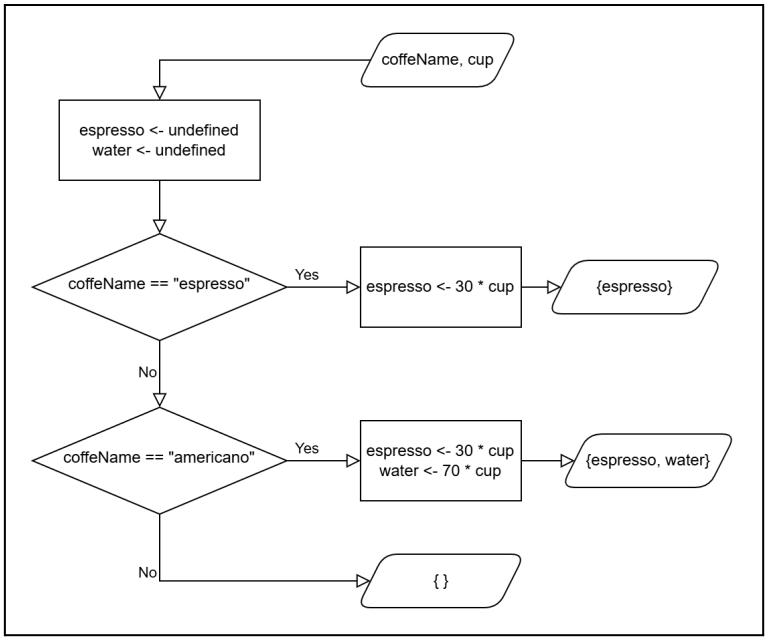
Path coverage đòi hỏi tester phân tích các trường hợp, các nhánh mà unit có thể đi qua, lập biểu đồ đường đi logic của unit và viết *test case* cho từng đường đi như vậy.

Các bước của Path coverage bao gồm:

1. Vẽ đồ thị điều khiển (*control graph*) để xác định các đường đi khác nhau trong chương trình. Đồ thị điều khiển là một biểu đồ đồ họa thể hiện mã nguồn của phần mềm.
2. Tính toán độ phức tạp cyclomatic - *cyclomatic complexity* - của chương trình hoặc xác định số lượng đường đi độc lập.
3. Tìm một tập hợp các đường đi để sử dụng làm cơ sở.
4. Tạo các test case để kiểm tra từng đường đi.

Độ phức tạp cyclomatic đề cập đến cách thức tính độ phức tạp của mã nguồn thông qua các thuộc tính của đồ thị (cạnh, đỉnh, các thành phần liên thông). Xem thêm tại đây: Cyclomatic complexity - Wikipedia

Từ ví dụ ta có biểu đồ sau:



Tương tự với Branch coverage, ta thấy có 5 đường đi cần phải kiểm tra, vàt ta còn thiếu một trường hợp của`americano`. Điểm khác biệt là ta không đoán số đường đi mà dựa vào biểu đồ rõ rãng.

Về các ưu điểm của phương thức này, vì path coverage testing kiểm tra mọi dòng mã nguồn của phần mềm, nó có thể:

- **Giảm bớt các bài kiểm tra dư thừa:** Vì bạn đã kiểm tra mỗi dòng mã ít nhất một lần, bạn không cần phải lặp lại các bài kiểm tra khác trên phần mềm chỉ để xem nó có hoạt động đúng không.
- **Tập trung vào logic chương trình:** Người kiểm thử có thể đảm bảo phần mềm tạo ra kết quả hợp lý. Ví dụ, phần mềm không nên trả về hai kết quả cho cùng một đầu vào.
- **Chạy test case ít nhất một lần:** Phần mềm sẽ thực thi tất cả các kịch bản có thể có vì bạn cần phải chạy qua mọi câu lệnh.
- **Đảm bảo thiết kế tốt nhất:** Kiểm thử kiểu này hỗ trợ thiết kế trường hợp phân tích (nghĩa là bạn sẽ có kết quả hợp lý hoặc kết quả có ý nghĩa) thay vì thiết kế trường hợp tùy tiện (liên quan nhiều đến chức năng hơn là độ chính xác).

Tuy nhiên, phương pháp này tốn rất nhiều công sức và thời gian. Nó không phù hợp cho các dự án bị giới hạn thời gian hay khi nhu cầu thay đổi liên tục.

III. Tiêu chuẩn và xu hướng công nghiệp

Không có 1 con số cụ thể để xác định tiêu chuẩn về *code coverage*. Do là vì *code coverage* không đồng nghĩa với chất lượng test, có thể dẫn đến những nhận định nhầm về dự án, và chi phí để làm tăng giá trị coverage có thể không phù hợp với những gì nhận lại được.

Lượng testing cần được thực hiện hoàn toàn phụ thuộc vào các đặc tính của riêng dự án hiện tại. Chúng bao gồm yêu cầu doanh nghiệp, độ phức tạp của dự án/thành phần, yêu cầu của khách hàng hay yêu cầu để làm việc nhóm,...

Tiêu chuẩn còn dựa vào kinh nghiệm và quyết định thống nhất của các thành viên trong đội nhóm.

Tuy vậy vẫn có một vài con số thường được áp dụng:

- Gợi ý chung cho các dự án: > 70-80% coverage
- Gợi ý của Google: 60% được coi là "chấp nhận được", 75% được coi là "đáng khen ngợi" và 90% là "Xuất xắc."
- 100% không thường được khuyến khích áp dụng do chi phí thực hiện lớn, giá trị đem lại không cao, và có thể khuyến khích các biện pháp "luồn lách" làm chất lượng test không ổn định.

Coverage theo các hệ thống tương tự:

- Business Logics, Security and Authentication: >80-90%
- API, Data Layer: >70-80%
- UI/UX, Notifications: >50-60%
- Overalls: 70-80%

Nhiều thông số coverage có thể tự động được kiểm tra sử dụng các tool và framework về test:

- Java: JaCoCo, Cobertura, OpenClover
- Python: Coverage.py, pytest-cov
- JavaScript / TypeScript: Istanbul, Jest, Vitest, c8
- .NET: Coverlet, AltCover
- ...

IV. Những thực hành tốt (Best Practices) trong Unit Testing

1. Duy trì tính dễ bảo trì (maintainability) và tránh phụ thuộc vào cơ sở dữ liệu

Unit test cần dễ đọc, cập nhật và bảo trì theo thời gian. Một trong những khía cạnh quan trọng nhất của khả năng bảo trì là *tránh phụ thuộc không cần thiết*, đặc biệt là vào các tài nguyên bên ngoài như cơ sở dữ liệu.

Vậy tại sao cần tránh phụ thuộc vào cơ sở dữ liệu?

- **Tốc độ**: Truy cập vào cơ sở dữ liệu làm chậm quá trình test do độ trễ mạng và truy xuất dữ liệu từ ổ đĩa.

- **Tính độc lập**: Unit test nên tập trung vào một đơn vị mã nguồn duy nhất. Nếu test phụ thuộc vào cơ sở dữ liệu, lỗi có thể đến từ database thay vì bản thân chính logic cần kiểm tra trong mã nguồn.

- **Tính lặp lại**: Trạng thái của cơ sở dữ liệu có thể thay đổi giữa các lần test, dẫn đến kết quả không nhất quán, dẫn đến việc Tester không xác định được lỗi đến từ đâu.

Những phương pháp thay thế tốt hơn

- **Sử dụng Mocks hoặc Fakes**: Thay vì dùng cơ sở dữ liệu thật, sử dụng mock objects hoặc Fakes.Vậy chúng là gì?

    + Mock là một ***đối tượng giả lập hành vi*** của một đối tượng thật nhưng được lập trình để trả về kết quả cố định khi gọi các phương thức nhất định. Mục đích là để kiểm tra logic của code mà không cần chạy toàn bộ hệ thống. Mock có các đặc điểm sau: (1) Đơn giản hơn đối tượng thực nhưng vẫn giữ được sự tương tác với các đối tượng khác, (2) Không lặp lại nội dung đối tượng thực, (3) Cho phép thiết lập các trạng thái riêng trợ giúp cho việc thực hiện unit test.

    + Fake là **một *bản sao đơn giản*** của thành phần thật, có thể hoạt động giống như thật nhưng nhẹ hơn và không tương tác với tài nguyên thật như cơ sở dữ liệu. Ví dụ: một danh sách lưu trong bộ nhớ thay vì table thật trong database.

Tóm lại, Mocks/Fakes giúp giả lập dữ liệu và hành vi mà không cần sử dụng đến hệ thống thực, từ đó giúp test chạy nhanh và ổn định hơn.

- **Dependency Injection**: Đây là kỹ thuật mà các phần phụ thuộc (ví dụ như database, API...) được truyền vào thay vì bị cố định bên trong class hoặc hàm. Nếu ta có thể "tiêm" các đối tượng khác nhau vào một class (ví dụ: database thật khi chạy thật, mock khi test), ta dễ dàng kiểm soát và thay thế chúng trong test.

- **Repository Pattern**: Đây là một mẫu thiết kế giúp tách biệt logic truy xuất dữ liệu (như SQL hoặc truy vấn database) ra khỏi logic nghiệp vụ. Lập trình viên làm việc với interface (giao diện), không cần quan tâm việc dữ liệu lấy từ đâu. Khi đó, việc thay thế implementation thật bằng mock hoặc fake trong test, giúp code dễ bảo trì và kiểm thử được sẽ trở nên dễ dàng hơn.

Ví dụ

Xem xét ví dụ về một hàm truy xuất thông tin người dùng từ cơ sở dữ liệu:

```
class UserRepository:
    def get_user(self, user_id):
        # Giả lập truy cập cơ sở dữ liệu
        return database.fetch("SELECT * FROM users WHERE id = ?", user_id)
```

Nếu trong test, chúng ta thật sự kết nối đến database thì sẽ: tốn thêm thời gian cho test, có khả năng bị lỗi do database không sẵn sàng hoặc dữ liệu thay đổi, khó chạy tự động trong môi trường CI/CD vì phải phụ thuộc thêm vào database.

Thay vì gọi database thật, chúng ta có thể mock UserRepository trong test:

```
from unittest.mock import MagicMock
def test_get_user():
    mock_repo = UserRepository()
    mock_repo.get_user = MagicMock(return_value={"id": 1, "name": "John Doe"})

    user = mock_repo.get_user(1)
    assert user["name"] == "John Doe"
```

`MagicMock` là một lớp trong thư viện `unittest.mock` của Python, cho phép tạo ra các đối tượng giả lập (mock objects) có thể bắt chước hành vi của các đối tượng thật. Điểm đặc biệt của `MagicMock` là nó **tự động giả lập tất cả các phương thức và thuộc tính**, nên rất linh hoạt và tiện lợi khi test.

Lợi ích của cách làm này:

- **Nhanh**: Không có thao tác với ổ đĩa hay mạng.

- **Ổn định:** Kết quả test luôn giống nhau vì mock trả về dữ liệu cố định.

- **Dễ bảo trì:** Không cần cài đặt database chỉ để chạy test.

- **Tập trung vào logic**: Test chỉ quan tâm đến đầu vào và đầu ra của hàm, không bị nhiễu bởi lỗi từ hệ thống khác.

2. Kiểm tra Happy Case và Edge Case

Một bộ test tốt cần bao gồm:

- **Happy Path Cases**: Các trường hợp đầu vào hợp lệ và đầu ra mong đợi.
- **Edge Cases**: Các giá trị nằm ở biên của giới hạn đầu vào để đảm bảo tính ổn định.

Ví dụ

Xem xét một hàm tính toán chiết khấu (giảm giá):

```
def calculate_discount(price, percentage):
    if percentage < 0 or percentage > 100:
        raise ValueError("Invalid percentage")
    return price * (1 - percentage / 100)
```

Test Happy Case:

```
def test_calculate_discount():
    assert calculate_discount(100, 10) == 90
```

Test Edge Case:

```
def test_calculate_discount_edge_cases():
    assert calculate_discount(100, 0) == 100  # Không giảm giá
    assert calculate_discount(100, 100) == 0  # Giảm giá 100%
```

Test giá trị không hợp lệ:

```
import pytest

def test_calculate_discount_invalid():
    with pytest.raises(ValueError):
        calculate_discount(100, -5)
    with pytest.raises(ValueError):
        calculate_discount(100, 105)
```

3. Tránh test dư thừa và gắn chặt với triển khai cụ thể (Over-Tied Implementation)

Test Dư Thừa là gì?
Test dư thừa là những bài kiểm thử không mang lại giá trị kiểm tra thực sự cho phần mềm. Chúng có thể:
- Kiểm tra lại những thứ đã được kiểm thử ở nơi khác.
- Trùng lặp logic kiểm tra.
- Kiểm thử những phần không phải trách nhiệm của đơn vị mã hiện tại.
Tại sao cần tránh test dư thừa?
- Làm tăng chi phí bảo trì mà không mang lại giá trị thực.

- Làm chậm quá trình chạy test.

- Gây nhầm lẫn khi xác định lỗi.

Test có yếu tố Gắn chặt với triển khai cụ thể (Over-Tied to Implementation)

Là khi một test:

- Phụ thuộc vào cách thức thực hiện (implementation) thay vì kết quả mong đợi (behavior).

- Thay đổi nhỏ về cách viết code (nhưng không ảnh hưởng kết quả) cũng khiến test bị lỗi.

- Làm giảm tính linh hoạt và khả năng tái cấu trúc mã nguồn.

Tại sao cần tránh test gắn chặt với triển khai?

- Nếu test phụ thuộc vào chi tiết triển khai cụ thể, các thay đổi nhỏ trong mã có thể làm hỏng test không cần thiết.

- Test nên tập trung vào hành vi mong đợi thay vì cách thức thực hiện.

=> Test nên hỏi: “Hành vi này có đúng không?” – chứ không phải: “Code bên trong chạy như tôi đoán không?”

Ví dụ test dư thừa:

```
def add(a, b):
    return a + b
def test_add_1():
    assert add(1, 2) == 3
def test_add_2():
    assert add(2, 1) == 3
```

Vấn đề:
- Cả hai test đều đang kiểm tra một logic đơn giản giống nhau.
- Không có giá trị kiểm tra mới hoặc tình huống đặc biệt.
- Gây nặng test suite, tốn thời gian duy trì khi hàm thay đổi.

Có thể chỉnh sửa lại như sau:

```
def test_add_basic_cases():
    assert add(1, 2) == 3
    assert add(0, 3) == 3
    assert add(-1, 4) == 3  # Thêm edge case: số âm
```

Việc này giúp test:
- Gom nhóm hợp lý các tình huống đơn giản.

- Bổ sung test có giá trị hơn (số âm).
- Dễ đọc, dễ bảo trì, tránh trùng lặp logic.

Ví dụ test gắn chặt vào triển khai cụ thể

```
def test_sort_numbers():
    numbers = [3, 1, 2]
    result = sorted(numbers)
    assert result == [1, 2, 3]  # Quá gắn với triển khai cụ thể
```

Test không kiểm tra logic riêng mà đang kiểm tra chính hàm `sorted()` của Python – vốn là thư viện chuẩn, đã được kiểm thử. Nếu sau này bạn thay `sorted()` bằng một hàm khác, chúng ta phải thay cả test, dù hành vi không đổi.

Viết lại hợp lý hơn:

```
def my_sort_function(nums):
    # Một hàm sắp xếp tự viết (ví dụ)
    return sorted(nums)

def test_sort_numbers():
    numbers = [3, 1, 2]
    result = my_sort_function(numbers)
    assert result == sorted(numbers)  # So sánh với hành vi mong đợi
```

Việc này đã
- So sánh với một hành vi đúng (hàm chuẩn `sorted()`).
- Nếu sau này thay đổi thuật toán trong `my_sort_function`, test vẫn đúng miễn là đầu ra không đổi.
- Giúp đảm bảo **đầu ra đúng**, không quan tâm cách thực hiện bên trong.

Kết luận

Việc tuân theo các best practices trong unit testing giúp đảm bảo độ tin cậy, khả năng bảo trì và hiệu suất. Tránh phụ thuộc vào cơ sở dữ liệu, bao quát cả happy case và edge case, và viết test tập trung vào hành vi giúp mã nguồn trở nên ổn định và dễ mở rộng hơn.

TÀI LIỆU THAM KHẢO

- Nickolay Bakharev (2024), Unit Testing: Definition, Examples, and Critical Best Practices, Bright Security, https://brightsec.com/blog/unit-testing/
- Phan Dang Hai Vu (2020), Các tiêu chí tính độ bao phủ code trong unit test, Viblo https://viblo.asia/p/GrLZD0w2Zk0
- Nhóm tác giả Jecelyn Yeen, Michael Hablich, Rachel Andrew, và Sofia Emelianova, Four common types of code coverage, Web.dev, https://web.dev/articles/ta-code-coverage
- Techslang (2022), What is Path Coverage Testing? A short definition of Path Coverage Testing, Techslang, https://www.techslang.com/definition/what-is-path-coverage-testing/
- Tìm hiểu về Stub, Mock và Fake trong unit test
- https://docs.python.org/3/library/unittest.mock.html