TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN, ĐHQG-HCM

KHOA CÔNG NGHỆ THÔNG TIN

----o0o----

THIẾT KẾ PHẦN MỀM

Báo cáo: Unit Test

Group: Fullstackoverflowrestling

TÊN	MSSV	EMAIL
Vòng Sau Hùng	22120118	22120118@student.hcmus.edu.vn
Lê Nguyễn Hồng Ngọc	22120232	22120232@student.hcmus.edu.vn
Phan Tấn Phát	22120264	22120264@student.hcmus.edu.vn

Thành phố Hồ Chí Minh, tháng 4 năm 2025

LỜI NÓI ĐẦU

Em xin gửi lời cảm ơn chân thành đến thầy Trần Duy Thảo, thầy Ngô Ngọc Đăng Khoa và thầy Nguyễn Đức Huy đã chủ nhiệm em trong môn học Thiết kế phần mềm – CQ2022/3. Mặc dù em đã cố gắng trình bày đầy đủ nội dung, song bài báo cáo của em có thể không tránh khỏi thiếu sót, kính mong nhận được sự góp ý cũng như đánh giá của các thầy để em có thể rút kinh nghiệm và hoàn thiện hơn trong những lần sau.

MỤC LỤC TỰ ĐỘNG

1. Giới thiệu 4
   1.1. Unit Test là gì? 4
   1.2. Tại sao cần phải Unit Test? 4
   1.3. Các đặc điểm của một Unit Test tốt 5
2. Unit Test Coverage 5
   2.1. Giới thiệu 5
   2.2. Các loại Coverage quan trọng 5
   2.3. Line Coverage – Độ bao phủ dòng code 6
      2.3.1. Định nghĩa: 6
      2.3.2. Mục tiêu: 6
      2.3.3. Khi nào cần Line Coverage cao? 7
      2.3.4. Ví dụ về Line Coverage trong NestJs: 8
      2.3.5. Kết Quả Coverage 8
   2.4. Branch Coverage - Độ bao phủ nhánh 8
      2.4.1. Định nghĩa: 8
      2.4.2. Mục tiêu: 8
      2.4.3. Khi Nào Cần Branch Coverage cao? 9
      2.4.4. Ví dụ Branch Coverage trong NestJs: 9
      2.4.5. Kết quả Coverage 10
   2.5. Function Coverage - Độ bao phủ hàm 10
      2.5.1. Định nghĩa: 10
      2.5.2. Mục tiêu: 10
      2.5.3. Khi nào cần Function Coverage cao?: 11
      2.5.4. Ví dụ Function Coverage trong NestJs: 11
   2.6. Path Coverage - Độ bao phủ đường đi 12
      2.6.1. Định nghĩa: 12
      2.6.2. Mục tiêu: 12
      2.6.3. Khi nào cần Path Coverage cao? 13

2.6.4. Ví dụ Path Coverage trong NestJs: 13
3. Tiêu chuẩn Industry 14
3.1. Tiêu chuẩn viết Unit Test 14
3.1.1. FIRST Principles: 14
3.1.2. AAA Pattern (Arrange – Act – Assert) 14
3.1.3. Single Responsibility for Tests (mỗi test chỉ kiểm tra một thứ) 14
3.1.4. Không Dùng Logic Trong Test: 14
3.2. Tiêu chuẩn Coverage 15
3.3. Tiêu chuẩn Mock Data và Test Case 15
3.3.1. Khi Nào Cần Mock? 15
3.3.2. Tiêu Chuẩn Viết Test Case Hiệu Quả 15

1. Giới thiệu

1.1. Unit Test là gì?

- **Unit Test (kiểm thử đơn vị)** là một kỹ thuật kiểm thử phần mềm nhằm kiểm tra tính chính xác của từng thành phần nhỏ nhất trong một hệ thống phần mềm. Thành phần này có thể là một function, method, class hoặc module, được kiểm tra riêng biệt mà không phụ thuộc vào các phần khác của hệ thống.
- **Unit Test** thường được viết bởi các lập trình viên trong giai đoạn phát triển phần mềm, trước khi tích hợp toàn bộ hệ thống. Điều này giúp phát hiện sớm các lỗi và đảm bảo rằng từng thành phần hoạt động đúng với yêu cầu thiết kế.

1.2. Tại sao cần phải Unit Test?

- **Phát hiện lỗi sớm:** Unit Test giúp tìm ra lỗi ngay từ giai đoạn đầu của quá trình phát triển phần mềm, trước khi hệ thống được tích hợp hoặc đưa vào sử dụng. Việc phát hiện lỗi sớm sẽ giúp tiết kiệm chi phí và thời gian so với việc sửa lỗi khi phần mềm đã được triển khai.
- **Giảm chi phí sửa lỗi:** Sửa lỗi trong giai đoạn phát triển sẽ ít tốn kém hơn so với việc sửa lỗi ở môi trường production. Khi lỗi được tìm thấy sớm, các lập trình viên có thể sửa ngay lập tức mà không làm ảnh hưởng đến các thành phần khác của hệ thống.
- **Cải thiện chất lượng code:** Khi lập trình viên viết Unit Test, họ sẽ có xu hướng viết code rõ ràng hơn, dễ bảo trì hơn. Unit Test giúp phát hiện những đoạn code dư thừa hoặc không cần thiết, từ đó tối ưu hóa mã nguồn và cải thiện cấu trúc phần mềm.
- **Tăng cường khả năng Refactor Code:** Khi có Unit Test, lập trình viên có thể dễ dàng thay đổi hoặc cải thiện code mà không lo ảnh hưởng đến các chức năng khác. Nếu một thay đổi làm hỏng hệ thống, Unit Test sẽ phát hiện ngay và giúp lập trình viên điều chỉnh kịp thời.
- **Hỗ trợ quá trình tích hợp và triển khai (CI/CD):** Unit Test có thể được tích hợp vào quy trình CI/CD (Continuous Integration/Continuous Deployment) để tự động kiểm tra phần mềm mỗi khi có thay đổi. Điều này giúp đảm bảo rằng mọi cập nhật trong mã nguồn đều được kiểm tra trước khi triển khai.
- **Giảm rủi ro khi phát triển phần mềm:** Việc sử dụng Unit Test giúp giảm thiểu rủi ro về lỗi phần mềm khi triển khai, đặc biệt là với những hệ thống lớn và phức tạp. Khi các đơn vị nhỏ trong hệ thống đã được kiểm thử kỹ lưỡng, khả năng xảy ra lỗi trong quá trình chạy thực tế sẽ giảm đi đáng kể.

1.3. Các đặc điểm của một Unit Test tốt

- **Độc lập:** Mỗi Unit Test nên kiểm tra một thành phần nhỏ của hệ thống mà không phụ thuộc vào các phần khác. Điều này giúp đảm bảo rằng nếu có lỗi xảy ra, chúng ta có thể dễ dàng xác định được nguồn gốc của vấn đề.
- **Dễ hiểu và dễ bảo trì:** Một Unit Test tốt cần có mã nguồn dễ đọc, rõ ràng và dễ bảo trì. Tên của test case nên mô tả được mục đích của bài kiểm thử.
- **Tái sử dụng được:** Test case nên được viết theo cách có thể tái sử dụng, giúp giảm thiểu công sức khi mở rộng hoặc sửa đổi hệ thống.
- **Chạy nhanh:** Test case nên được viết theo cách có thể tái sử dụng, giúp giảm thiểu công sức khi mở rộng hoặc sửa đổi hệ thống.
- **Đưa ra kết quả chính xác:** Unit Test nên đưa ra kết quả rõ ràng và chính xác. Nếu một test case thất bại, lập trình viên cần có đủ thông tin để xác định nguyên nhân lỗi và cách khắc phục.

2. Unit Test Coverage

2.1. Giới thiệu

- **Unit Test Coverage (độ bao phủ kiểm thử đơn vị)** là thước đo thể hiện mức độ mã nguồn (source code) được kiểm thử thông qua các Unit Test. Nó giúp đánh giá phạm vi kiểm thử và phát hiện các phần chưa được test trong hệ thống.
- **Tại sao cần quan tâm đến Code Coverage:**
  - Xác minh tính hiệu quả của Unit Test: Kiểm tra xem các test case có thực sự kiểm thử đủ các phần quan trọng của ứng dụng không.
  - Phát hiện các đoạn code chưa được test: Giúp tránh lỗi chưa được phát hiện trong quá trình phát triển.
  - Tăng cường độ tin cậy của phần mềm: Mức độ bao phủ cao giúp đảm bảo phần mềm hoạt động ổn định hơn.
  - Hỗ trợ quá trình Refactoring: Nếu code coverage cao, lập trình viên có thể tự tin thay đổi code mà không lo phá vỡ tính năng hiện tại.
  - Giảm chi phí sửa lỗi: Phát hiện lỗi sớm giúp giảm chi phí và thời gian sửa chữa khi phần mềm đã được triển khai.

2.2. Các loại Coverage quan trọng

Loại Coverage	Định nghĩa	Dùng khi nào?
Line Coverage	Kiểm tra tỷ lệ dòng code được thực thi.	Khi cần đảm bảo mọi dòng code quan trọng đều được chạy.
Branch Coverage	Kiểm tra tất cả các nhánh (if, else, switch).	Khi có nhiều nhánh logic, cần test đầy đủ.
Function Coverage	Kiểm tra tất cả các hàm có được gọi ít nhất một lần.	Khi có nhiều function cần xác minh rằng chúng đã được test.
Path Coverage	Kiểm tra tất cả các đường đi logic có thể xảy ra.	Khi có nhiều điều kiện lồng nhau, cần test mọi kịch bản có thể.

2.3. Line Coverage – Độ bao phủ dòng code

2.3.1. Định nghĩa:

Line Coverage đo tỷ lệ phần trăm dòng code đã được chạy qua bởi Unit Test. Nếu một dòng code không được chạy qua trong bất kỳ test case nào: Chứa lỗi chưa được phát hiện, không thực sự cần thiết.

2.3.2. Mục tiêu:

- Xác định code chưa được test
  - Nếu một dòng code chưa bao giờ được thực thi, có thể nó không cần thiết hoặc có lỗi logic mà chưa được phát hiện.
  - Điều này giúp lập trình viên tối ưu code và bổ sung test case để đảm bảo chất lượng.
- Tăng độ tin cậy của phần mềm
  - Khi tất cả dòng code quan trọng đều được kiểm thử, nguy cơ lỗi khi triển khai sẽ giảm đáng kể.
  - Đặc biệt quan trọng trong hệ thống lớn hoặc hệ thống quan trọng như tài chính, y tế.
- Hỗ trợ refactoring an toàn
  - Nếu cần thay đổi code, có thể kiểm tra các dòng nào có test coverage, từ đó giảm rủi ro khi chỉnh sửa.
  - Nếu coverage thấp, cần bổ sung test trước khi refactor để tránh lỗi ngoài ý muốn.

- Đánh giá chất lượng Unit Test
  - Nếu Line Coverage quá thấp (dưới 50%), có thể Unit Test chưa đầy đủ.
  - Nếu quá cao (trên 90%) nhưng vẫn có bug, có thể test case chưa thực sự kiểm tra tính đúng đắn của logic.

2.3.3. Khi nào cần Line Coverage cao?

- Khi viết các hàm xử lý dữ liệu cần đảm bảo mọi dòng đều thực thi đúng.
- Khi debug lỗi trong quá trình phát triển.
- Khi muốn đánh giá phần code nào đang bị bỏ sót.
- Khi áp dụng CI/CD, vì coverage thấp có thể làm pipeline bị fail nếu có quy định coverage tối thiểu.
- Khi phát triển hệ thống lớn, cần xác minh các phần quan trọng đã được kiểm thử.
- Khi làm việc với code mới hoặc refactor – Đảm bảo test đã bao phủ toàn bộ logic mới.

2.3.4. Ví dụ về Line Coverage trong NestJs:

```
@Injectable()
export class MathService {
  add(a: number, b: number): number {
    return a + b;  // Dòng này cần được test
  }
}
```

```
describe('MathService', () => {
  let service: MathService;

  beforeEach(() => {
    service = new MathService();
  });

  it('should add two numbers correctly', () => {
    expect(service.add(3, 5)).toBe(8);
  });
});
```

2.3.5. Kết Quả Coverage

- Nếu không có test nào gọi service.add(), Line Coverage sẽ thấp.
- Khi test chạy thành công, Line Coverage đạt 100% cho phương thức add().

2.4. Branch Coverage - Độ bao phủ nhánh

2.4.1. Định nghĩa:

- Branch Coverage kiểm tra tất cả các nhánh (if, else, switch) có được test hay chưa. Nếu một nhánh không được kiểm thử, có thể nó chứa lỗi logic nghiêm trọng.

2.4.2. Mục tiêu:

- Đảm bảo mọi nhánh điều kiện đều được kiểm thử
    - Nếu chỉ có Line Coverage, ta chỉ biết code có chạy hay không, nhưng không biết tất cả các nhánh logic có được test không.

    - Branch Coverage đảm bảo mọi kịch bản điều kiện đều hoạt động chính xác.
- Phát hiện bug trong điều kiện chưa được test
    - Nếu một nhánh (else) không bao giờ được chạy, có thể nó chứa lỗi mà ta chưa phát hiện.
    - Tránh trường hợp một số điều kiện hiếm gặp gây lỗi trong runtime.
- Cải thiện chất lượng Unit Test
    - Chạy tất cả nhánh giúp kiểm thử kỹ lưỡng hơn, đặc biệt quan trọng khi function có nhiều điều kiện lồng nhau.
- Tăng độ tin cậy của hệ thống
    - Hữu ích với các hệ thống quan trọng (tài chính, y tế, giao dịch ngân hàng, API bảo mật, ...), nơi một nhánh sai có thể gây lỗi nghiêm trọng.

2.4.3. Khi Nào Cần Branch Coverage cao?

- Khi có nhiều điều kiện rẽ nhánh (if, else).
- Khi xử lý nhiều trường hợp khác nhau trong một function.
- Khi test API hoặc rules engine, nơi mỗi nhánh có thể ảnh hưởng kết quả cuối cùng.

2.4.4. Ví dụ Branch Coverage trong NestJs:

```
@Injectable()
export class AuthService {
  isAdmin(userRole: string): boolean {
    if (userRole === 'admin') {
      return true;  // Nhánh 1
    }
    return false;   // Nhánh 2
  }
}
```

```
describe('AuthService', () => {
  let service: AuthService;

  beforeEach(() => {
    service = new AuthService();
  });

  it('should return true for admin', () => {
    expect(service.isAdmin('admin')).toBe(true);  // Kiểm tra Nhánh 1
  });

  it('should return false for non-admin', () => {
    expect(service.isAdmin('user')).toBe(false);  // Kiểm tra Nhánh 2
  });
});
```

2.4.5. Kết quả Coverage

- Nếu chỉ test isAdmin('admin'), Nhánh 2 chưa được kiểm tra.
- Test isAdmin('user') giúp đạt 100% Branch Coverage.

2.5. Function Coverage - Độ bao phủ hàm

2.5.1. Định nghĩa:

- Function Coverage kiểm tra xem tất cả các hàm có được gọi ít nhất một lần hay không.

2.5.2. Mục tiêu:

- **Đảm bảo tất cả function đều hoạt động đúng**
  - Nếu một function không được gọi trong test, có thể nó không quan trọng hoặc chưa có test case phù hợp.
  - Nếu function chưa được test, có thể nó chứa bug mà ta chưa phát hiện.
- **Phát hiện function dư thừa hoặc không cần thiết**
  - Nếu một function không bao giờ được test hoặc không bao giờ được gọi trong runtime, có thể nó không cần thiết và nên được xóa.
  - Giúp tinh gọn codebase, tránh technical debt.
- **Tăng độ tin cậy của Unit Test**

    - Nếu tất cả function đều được gọi và test đầy đủ, Unit Test sẽ có độ tin cậy cao hơn.
    - Đặc biệt quan trọng trong hệ thống lớn, codebase cũ, hoặc code bảo mật.
- **Hữu ích khi refactor hoặc cập nhật code**
    - Nếu một function đã có test coverage đầy đủ, ta có thể chỉnh sửa hoặc tối ưu function đó một cách an toàn, vì nếu có lỗi, test sẽ fail ngay.

2.5.3. Khi nào cần Function Coverage cao?:

- Khi có nhiều function quan trọng, cần đảm bảo tất cả đều được test.
- Khi thực hiện refactoring, giúp xác định function nào có thể sửa mà không gây lỗi.
- Khi làm việc với hệ thống bảo mật hoặc xử lý tài chính, nơi mỗi function cần được kiểm thử cẩn thận.

2.5.4. Ví dụ Function Coverage trong NestJs:

```
@Injectable()
export class StringService {
  toUpperCase(str: string): string {
    return str.toUpperCase();
  }

  toLowerCase(str: string): string {
    return str.toLowerCase();
  }
}
```

```
describe('StringService', () => {
  let service: StringService;

  beforeEach(() => {
    service = new StringService();
  });

  it('should convert string to uppercase', () => {
    expect(service.toUpperCase('hello')).toBe('HELLO');
  });

  it('should convert string to lowercase', () => {
    expect(service.toLowerCase('HELLO')).toBe('hello');
  });
});
```

2.6. Path Coverage - Độ bao phủ đường đi

2.6.1. Định nghĩa:

- Path Coverage kiểm tra tất cả các đường đi logic có thể xảy ra trong một hàm. Khác với Branch Coverage (kiểm tra từng nhánh if, else có chạy không), Path Coverage kiểm tra mọi tổ hợp đường đi có thể có trong function.

2.6.2. Mục tiêu:

- **Kiểm tra tất cả tổ hợp logic có thể xảy ra**
  - Đặc biệt hữu ích trong các function phức tạp có nhiều điều kiện lồng nhau.
  - Nếu chỉ có Branch Coverage, có thể một số tổ hợp điều kiện vẫn chưa được kiểm tra.
- **Phát hiện lỗi trong logic điều kiện kết hợp**

- Một function có thể chạy đúng trong một số trường hợp, nhưng khi kết hợp nhiều điều kiện lại gây lỗi.
- Path Coverage giúp phát hiện bug xảy ra khi các điều kiện tương tác với nhau.

- **Tăng độ tin cậy của hệ thống quan trọng**
  - Trong các hệ thống bảo mật, tài chính, AI, kiểm soát giao thông, mỗi tổ hợp điều kiện có thể dẫn đến kết quả khác nhau.
- **Hữu ích khi tối ưu hóa logic điều kiện**
  - Nếu có quá nhiều đường đi không cần thiết, có thể đơn giản hóa code bằng cách gộp logic điều kiện.

2.6.3. Khi nào cần Path Coverage cao?

- Khi có nhiều điều kiện lồng nhau (if trong if).
- Khi logic nghiệp vụ có nhiều trạng thái (ví dụ: xử lý giao dịch, kiểm tra quyền truy cập).

2.6.4. Ví dụ Path Coverage trong NestJs:

```
@Injectable()
export class WalletService {
  checkDeposit(amount: number, userStatus: string, isVerified: boolean): string {
    if (amount <= 0) {
      return 'Invalid amount';
    }
    if (userStatus !== 'active') {
      return 'User not active';
    }
    if (!isVerified) {
      return 'User not verified';
    }
    return 'Deposit successful';
  }
}
```

3. Tiêu chuẩn Industry

Trong industry, Unit Test không chỉ đơn thuần là kiểm thử từng thành phần nhỏ mà còn phải đáp ứng các tiêu chuẩn nhất định để đảm bảo hiệu quả. Dưới đây là các tiêu chuẩn quan trọng trong Unit Test mà bạn cần biết khi phát triển phần mềm.

3.1. Tiêu chuẩn viết Unit Test

3.1.1. FIRST Principles:

Một Unit Test tốt phải tuân theo nguyên tắc FIRST

Nguyên tắc	Ý nghĩa
Fast (Chạy nhanh)	Unit Test cần chạy nhanh để không làm chậm quá trình phát triển.
Isolated (Độc lập)	Test không nên phụ thuộc vào các test khác, dữ liệu bên ngoài (database, API).
Repeatable (Lặp lại được)	Test phải cho kết quả giống nhau mỗi khi chạy.
Self-validating (Tự kiểm tra)	Kết quả test phải rõ ràng: PASS hoặc FAIL, không cần kiểm tra thủ công.
Timely (Viết đúng lúc)	Test nên được viết trước hoặc cùng lúc với code, không nên viết sau khi code đã hoàn thành.

3.1.2. AAA Pattern (Arrange – Act – Assert)

- Arrange – Chuẩn bị dữ liệu, mock dependency.
- Act – Gọi function cần test.
- Assert – Kiểm tra kết quả đầu ra có đúng không.

3.1.3. Single Responsibility for Tests (mỗi test chỉ kiểm tra một thứ)

- Mỗi test case chỉ nên kiểm tra duy nhất một logic, tránh kiểm tra nhiều thứ cùng lúc.
- Nếu một test fail, ta biết ngay nguyên nhân lỗi mà không cần debug nhiều.

3.1.4. Không Dùng Logic Trong Test:

- Unit Test không nên chứa câu lệnh if, for, hoặc logic phức tạp, vì điều này làm test khó hiểu và khó bảo trì.

3.2. Tiêu chuẩn Coverage

- Mức Code Coverage Tiêu Chuẩn
  - Lưu ý: 100% Coverage không đồng nghĩa với code không có bug. Chất lượng test quan trọng hơn con số.

Loại Coverage	**Ý nghĩa**	**Mức tối thiểu (Industry standard)**
Line Coverage	Kiểm tra số dòng code đã chạy.	**70-80%**
Branch Coverage	Kiểm tra các nhánh if, else, switch đã chạy.	**80-90%**
Function Coverage	Kiểm tra số function đã gọi ít nhất một lần.	**90-100%**
Path Coverage	Kiểm tra mọi tổ hợp đường đi có thể xảy ra trong function.	**95-100%** (nếu logic phức tạp)

- Không Chase 100% Coverage
  - Không cần đạt 100% Coverage nếu:
    - Code có nhiều function helper đơn giản (formatDate(), convertToUpperCase()).
    - Có các phần khó test (ví dụ: logging, third-party API, config).
    - Coverage đạt 100% nhưng test chỉ kiểm tra các trường hợp đơn giản, không test edge case.
  - Nên tập trung vào việc kiểm thử logic quan trọng hơn là đạt 100% Coverage.

3.3. Tiêu chuẩn Mock Data và Test Case

3.3.1. Khi Nào Cần Mock?

- Khi test Service, không nên gọi Repository/Database thật.
- Khi test Controller, không nên gọi Service thật.
- Khi test API, không nên gọi third-party API thật.

3.3.2. Tiêu Chuẩn Viết Test Case Hiệu Quả

- Test case nên bao gồm cả trường hợp thành công và thất bại.

- Nên có test case cho edge case (giá trị cực đại, cực tiểu, null, undefined,...).
- Không chỉ kiểm tra happy case, mà phải test cả trường hợp lỗi, input không hợp lệ.
- Đặt tên test case rõ ràng, dễ hiểu.