TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN TP.HCM
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO

Thiết Kế Phần Mềm

Tìm hiểu về Unit Testing

Nhóm 18 – Mewing

Thành viên nhóm:

22120046	**Nguyễn Ngọc Đăng**
22120181	**Nguyễn Duy Lâm**
22120183	**Nguyễn Đặng Minh Lân**

Tp. Hồ Chí Minh, ngày 04 tháng 04 năm 2025

MỤC LỤC

1. Các loại coverage quan trọng

1.1. Line Coverage

- Định nghĩa: Đo lường tỷ lệ dòng code được thực thi trong quá trình kiểm thử.
- Mục tiêu: Đảm bảo mỗi dòng code đều được kiểm tra.
- Ưu điểm: Dễ đo lường, cung cấp cái nhìn chi tiết về từng dòng mã
- Nhược điểm: Không đảm bảo kiểm tra hết các nhánh logic
- Ví dụ:
    - Mã nguồn

```
def func(a):
     if a == 0:
          a = a + 1
     return a
```

    - Kiểm thử
        - **func(1)**: ⅔ nhánh được thực thi
        - **func(0)**: 100% nhánh được thực thi

1.2. Branch Coverage

- Định nghĩa: Đo lường tỷ lệ các nhánh (if, else, switch) được thực thi trong quá trình kiểm thử
- Mục tiêu: Đảm bảo mỗi nhánh đều được kiểm tra
- Ưu điểm: Kiểm tra được các nhánh logic, phát hiện lỗi trong các trường hợp hiếm gặp
- Nhược điểm: Không bao quát hết tất cả các tổ hợp logic
- Ví dụ:
    - Mã nguồn

```
def func(a):
     if a == 0:
          a = a + 1
     else:
          a = a + 2
```

    - Kiểm thử
        - Gọi **func(1)**: ½ nhánh được kiểm thử
        - Gọi **func(1)** và **func(0)**: 100% nhánh được kiểm thử

1.3. Function Coverage

- Định nghĩa: Đo lường tỷ lệ các hàm được thực thi trong quá trình kiểm thử.
- Mục tiêu: Đảm bảo các hàm trong mã nguồn đều được gọi ít nhất một lần
- Ưu điểm: Đảm bảo mọi hàm đều được kiểm tra
- Nhược điểm: Không chú trọng đến logic bên trong hàm
- Ví dụ:
    - Mã nguồn

```
def func1(a):
     return a + 1
def func2(a):
```

```
        return a + 2
```

- Kiểm thử
  - Gọi **func1(0)**: ½ hàm được gọi
  - Gọi **func1(0)** và **func2(0)**: 100% hàm được gọi

1.4. Path Coverage

- Định nghĩa: Đo lường tỷ lệ tất cả các nhánh thực thi (paths) trong mã được kiểm tra. Một con đường là một chuỗi đầy đủ các nhánh từ đầu đến cuối chương trình.
- Mục tiêu: Kiểm tra mọi tổ hợp logic có thể xảy ra của mã nguồn
- Ưu điểm: Kiểm tra toàn diện mã nguồn
- Nhược điểm: Số lượng nhánh logic rất lớn trong các mã nguồn phức tạp, khó đạt được 100%
- Ví dụ:
  - Mã nguồn

```
def func(a, b):
    rs = 0
    if a > 0:
        rs += a
    if b > 0:
        rs += b
    return rs
```

  - Kiểm thử
    - Có 4 path:

```
1. x > 0, y > 0
2. x > 0, y < 0
3. x < 0, y > 0
4. x < 0, y < 0
```

    - **func(1, 0)**: ¼ nhánh được thực thi
    - Cần 4 test case để kiểm tra toàn bộ con đường logic của mã nguồn

2. Mức Coverage tối thiểu

Mức độ bao phủ mã nguồn (code coverage) là một chỉ số quan trọng trong kiểm thử phần mềm, phản ánh tỷ lệ phần trăm của mã nguồn được thực thi thông qua các bài kiểm thử tự động. Việc xác định mức coverage tối thiểu phù hợp phụ thuộc vào nhiều yếu tố, bao gồm loại hệ thống, mức độ quan trọng và yêu cầu chất lượng.

2.1. Khuyến nghị chung về mức độ bao phủ mã nguồn:

- **Ứng dụng thương mại:** mức độ bao phủ mã nguồn được đề xuất trong ngành phần mềm thương mại thường dao động từ **80% đến 90%**, đặc biệt đối với các ứng dụng không yêu cầu độ an toàn cực cao (như phần mềm quản lý doanh nghiệp hoặc giáo dục). Mức này được xem là điểm cân bằng giữa việc đảm bảo chất lượng mã nguồn và tránh lãng phí tài nguyên vào việc kiểm thử các phần mã không quan trọng.
- **Phần mềm điều khiển hàng không:** Cần phải tuân thủ theo tiêu chuẩn hàng không **DO-178B** yêu cầu 100% coverage cho các hệ thống quan trọng về an toàn như hệ thống xử lý trong các trường hợp tai nạn máy bay, hành khách bị chấn thương,...

- **Các hệ thống điện tử:** Tuân theo tiêu chuẩn **IEC 61508** khuyến nghị mức coverage là 100% của một số chỉ số quan trọng nhưng mức độ nghiêm ngặt còn tùy vào tính quan trọng của chỉ số.
- **Chức năng của xe:** Không có phần trăm Coverage cụ thể cho các chức năng của xe nhưng chúng phải được kiểm thử kỹ càng và hầu hết đảm bảo ở mức coverage 100% tùy tính quan trọng của từng chức năng.
- **Ứng dụng quản lý sinh viên:** Mức coverage khoảng **85%** được coi là đủ để đảm bảo chức năng cốt lõi. Các chức năng như tính điểm trung bình (GPA) hoặc kiểm tra tính hợp lệ của dữ liệu đầu vào cần được kiểm thử kỹ lưỡng, trong khi các phần mã giao diện người dùng đơn giản có thể không cần đạt 100% coverage.

2.2. Lưu ý quan trọng về mức độ bao phủ mã nguồn:

- **Chất lượng hơn số lượng:** Mặc dù mức coverage cao có thể cho thấy phạm vi kiểm thử rộng, nhưng không đảm bảo chất lượng kiểm thử. Việc tập trung vào các phần quan trọng của ứng dụng và đảm bảo kiểm thử hiệu quả quan trọng hơn việc đạt được một con số coverage cụ thể.

- **Tránh theo đuổi 100% coverage:** Mức coverage 100% không phải lúc nào cũng khả thi hoặc cần thiết. Việc cố gắng đạt được 100% có thể dẫn đến lãng phí tài nguyên và không phản ánh đúng chất lượng kiểm thử.

- **Sử dụng coverage như một công cụ, không phải mục tiêu:** Code coverage nên được xem là một công cụ để xác định các phần mã chưa được kiểm thử, hơn là một mục tiêu cần đạt được.

Tóm lại, mức độ bao phủ mã nguồn tối thiểu nên được xác định dựa trên đặc thù của dự án và yêu cầu chất lượng cụ thể, thay vì chỉ dựa vào một con số cố định.

3. Best practices khi viết unit tests

3.1 Viết Unit Test Dễ Bảo Trì và Đọc Hiểu

- Unit test không chỉ dành cho việc kiểm tra mà còn là tài liệu sống động mô tả cách ứng dụng hoạt động. Vì vậy, hãy đặt tên test case rõ ràng, phản ánh đúng mục đích kiểm tra. Ví dụ: thay vì đặt tên mơ hồ như test1, hãy dùng testAddStudentWithValidDataReturnsSuccess để người đọc hiểu ngay test này kiểm tra việc thêm sinh viên với dữ liệu hợp lệ.
- Giữ code trong test đơn giản, tránh nhồi nhét quá nhiều logic phức tạp. Một test case lý tưởng chỉ nên tập trung kiểm tra một khía cạnh cụ thể của chức năng. Nếu code test quá dài hoặc khó hiểu, hãy xem xét chia nhỏ thành nhiều test riêng biệt.
- Áp dụng cấu trúc **AAA (Arrange-Act-Assert)** một cách nhất quán:
    - **Arrange**: Chuẩn bị dữ liệu đầu vào, mock các dependency nếu cần (ví dụ: tạo một đối tượng sinh viên giả lập).
    - **Act**: Gọi hàm hoặc phương thức cần kiểm tra (ví dụ: gọi hàm addStudent()).
    - **Assert**: Xác nhận kết quả đúng như kỳ vọng (ví dụ: kiểm tra danh sách sinh viên có thêm phần tử mới).
- Điều này giúp test dễ đọc, dễ bảo trì và giảm thiểu lỗi trong quá trình phát triển.

3.2 Không Phụ Thuộc Vào Database hoặc Persistent Storage

- Unit test nên kiểm tra logic của code một cách cô lập, không dựa vào các hệ thống bên ngoài như database, API, hoặc file system. Nếu phụ thuộc vào database thực, test sẽ chậm hơn và dễ bị ảnh hưởng bởi dữ liệu hiện có hoặc trạng thái hệ thống.
- Thay vào đó, sử dụng kỹ thuật **mocking** (giả lập) hoặc **stubbing** để thay thế các dependency. Ví dụ: trong ứng dụng quản lý sinh viên, thay vì kết nối thật tới database để lấy danh sách sinh viên, bạn có thể mock một đối tượng StudentRepository trả về danh sách giả lập như List<Student>.
- Lợi ích: Test chạy nhanh hơn (thường chỉ mất vài mili giây), không cần thiết lập môi trường phức tạp, và kết quả không bị ảnh hưởng bởi lỗi từ hệ thống bên ngoài. Công cụ phổ biến hỗ trợ mocking bao gồm Mockito (Java), unittest.mock (Python), hoặc Moq (C#).

3.3 Kiểm Thử Cả Happy Case và Edge Case

- **Happy Case (Trường hợp lý tưởng)**: Đảm bảo chức năng hoạt động đúng với dữ liệu đầu vào hợp lệ. Ví dụ: kiểm tra xem hàm addStudent() có thêm được sinh viên với tên, tuổi, và mã sinh viên hợp lệ vào danh sách hay không. Đây là bước cơ bản để xác nhận logic chính hoạt động như mong đợi.
- **Edge Case (Trường hợp biên)**: Kiểm tra các tình huống bất thường hoặc dữ liệu ở ranh giới. Ví dụ: thử thêm sinh viên với tuổi âm (-5), mã sinh viên trùng lặp, hoặc tên để trống (""). Edge case giúp phát hiện lỗ hổng mà happy case không bao phủ.
- **Error Case (Trường hợp lỗi)**: Xác minh ứng dụng xử lý tốt các lỗi, như ném ngoại lệ (exception) khi cần thiết. Ví dụ: kiểm tra xem hàm deleteStudent() có báo lỗi khi mã sinh viên không tồn tại hay không.
- Việc kiểm tra đầy đủ các trường hợp này đảm bảo ứng dụng không chỉ hoạt động đúng trong điều kiện lý tưởng mà còn đủ mạnh mẽ để xử lý các tình huống thực tế.

3.4 Tránh Test Trùng Lặp hoặc Quá Phụ Thuộc Vào Implementation Details

- **Không trùng lặp**: Nếu hai test case kiểm tra cùng một logic theo cách gần giống nhau, hãy gộp lại hoặc loại bỏ một test để giảm chi phí bảo trì. Ví dụ: không cần viết hai test riêng biệt để kiểm tra addStudent() với tên "John" và "Jane" nếu logic xử lý tên là như nhau.
- **Tập trung vào hành vi, không phải triển khai**: Unit test nên kiểm tra kết quả đầu ra (output) hoặc hiệu ứng (side effect) của hàm, chứ không phải cách hàm được viết bên trong. Ví dụ: khi kiểm tra hàm sortStudentsByAge(), chỉ cần xác nhận danh sách trả về được sắp xếp đúng, không cần kiểm tra từng bước trong thuật toán sắp xếp. Nếu sau này thay đổi thuật toán từ QuickSort sang MergeSort, test vẫn sẽ hoạt động mà không cần sửa.
- Điều này giúp test bền vững hơn khi code thay đổi, tránh việc phải viết lại test chỉ vì refactor code.

3.5 Đảm Bảo Test Độc Lập, Có Tính Lặp Lại và Không Phụ Thuộc Thứ Tự

- Mỗi test case phải chạy độc lập, không dựa vào kết quả của test trước đó. Ví dụ: nếu testAddStudent() thêm một sinh viên và testDeleteStudent() xóa sinh viên đó, không được

phép cho rằng testAddStudent() đã chạy trước để cung cấp dữ liệu. Thay vào đó, testDeleteStudent() nên tự chuẩn bị dữ liệu của riêng nó.

- Test phải cho kết quả nhất quán dù chạy bao nhiêu lần hay trong môi trường nào. Tránh sử dụng các giá trị ngẫu nhiên (random) hoặc thời gian hệ thống (system time) trừ khi có cách kiểm soát chúng.
- Đảm bảo các test không bị ảnh hưởng bởi thứ tự chạy trong suite. Công cụ như JUnit hoặc pytest thường chạy test ngẫu nhiên, nên việc giữ tính độc lập là rất quan trọng.

3.6 Tận Dụng Công Cụ Hỗ Trợ và Tích Hợp Tự Động Hóa

- Sử dụng các framework mạnh mẽ như **JUnit** (Java), **pytest** (Python), **NUnit** (C#), hoặc **Jest** (JavaScript) để tổ chức test case, chạy test tự động và báo cáo kết quả.
- Kết hợp với công cụ đo **code coverage** như **JaCoCo** (Java), **Coverage.py** (Python), hoặc **Coveralls** để đánh giá mức độ bao phủ của test, từ đó xác định các phần code chưa được kiểm tra.
- Tích hợp unit test vào quy trình **CI/CD** (Continuous Integration/Continuous Deployment) để tự động chạy test mỗi khi code được đẩy lên repository, đảm bảo chất lượng code liên tục.

3.7 Giữ Test Nhanh và Nhẹ

- Unit test nên chạy nhanh (thường dưới 1 giây cho toàn bộ suite) để nhà phát triển có thể kiểm tra thường xuyên mà không bị gián đoạn. Nếu test chậm, hãy xem xét giảm sự phụ thuộc vào I/O hoặc tối ưu hóa mocking.
- Tránh kiểm tra quá nhiều thứ trong một test case. Ví dụ: thay vì kiểm tra cả thêm, xóa, và cập nhật sinh viên trong cùng một test, hãy tách thành ba test riêng biệt để dễ debug khi có lỗi.

3.8 Viết Test Song Song Với Code Phát Triển (TDD Nếu Có Thể)

- Nếu áp dụng **Test-Driven Development (TDD)**, hãy viết test trước, sau đó viết code để test pass. Điều này buộc bạn phải suy nghĩ về yêu cầu và thiết kế trước khi triển khai.
- Ngay cả khi không dùng TDD, việc viết test sớm giúp phát hiện lỗi ngay từ đầu và giảm công sức sửa chữa sau này.

TÀI LIỆU THAM KHẢO

[1] "Atlassian," [Online]. Available: https://www.atlassian.com/continuous-delivery/software-testing/code-coverage.

[2] "StackOverflow," [Online]. Available: https://stackoverflow.com/questions/90002/what-is-a-reasonable-code-coverage-for-unit-tests-and-why.

[3] "Bulleyes," [Online]. Available: https://www.bullseye.com/minimum.html.