TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

🙧🕮🙥

A picture containing logo

Description automatically generated

Báo cáo Tìm Hiểu về Unit Test Coverage và Best Practices

Môn: Thiết kế phần mềm

GVHD: Trần Duy Thảo

*Thành phố Hồ Chí Minh, tháng 2, năm 2025*

# 1. Giới Thiệu về Unit Test Coverage

Unit Test Coverage là một trong những tiêu chí quan trọng để đánh giá mức độ kiểm thử của mã nguồn. Mục tiêu chính của Unit Test là phát hiện sớm các lỗi tiềm ẩn, đảm bảo phần mềm hoạt động chính xác và nâng cao chất lượng mã nguồn.

Unit Test Coverage không chỉ giúp xác định tỷ lệ mã nguồn được kiểm thử mà còn đóng vai trò quan trọng trong việc giảm thiểu lỗi khi triển khai phần mềm vào thực tế. Các tiêu chí chính của Unit Test Coverage bao gồm:

* Tăng cường độ tin cậy: Khi một hệ thống có độ phủ kiểm thử cao, khả năng phát hiện lỗi trước khi sản phẩm đến tay người dùng sẽ cao hơn.
* Cải thiện bảo trì mã nguồn: Việc có một bộ kiểm thử đầy đủ giúp dễ dàng phát hiện lỗi khi có thay đổi hoặc nâng cấp hệ thống.
* Giảm thiểu chi phí sửa lỗi: Phát hiện lỗi sớm giúp giảm đáng kể chi phí khắc phục so với việc sửa lỗi sau khi triển khai.
* Tối ưu hóa hiệu suất phát triển phần mềm: Việc có sẵn các unit test giúp tăng tốc độ phát triển và thử nghiệm các tính năng mới mà không làm ảnh hưởng đến hệ thống hiện có.
* Giảm thiểu rủi ro bảo mật: Một số unit test có thể giúp phát hiện sớm các lỗ hổng bảo mật hoặc lỗi logic trong ứng dụng.

# 2. Các Loại Code Coverage Quan Trọng

## 2.1 Line Coverage

* Kiểm tra tỷ lệ dòng code được thực thi khi chạy unit test.
* Nếu một phương thức có 10 dòng lệnh và unit test chạy qua 8 dòng, coverage sẽ là 80%.

## 2.2 Branch Coverage

* Đảm bảo tất cả các nhánh điều kiện (if, else, switch) được kiểm thử.
* Một điều kiện if (a > 10) {} phải có ít nhất 2 test case để kiểm tra cả trường hợp true và false.

## 2.3 Function Coverage

* Xác minh rằng tất cả các phương thức/hàm được gọi ít nhất một lần.
* Đảm bảo không có phương thức nào bị bỏ sót trong quá trình kiểm thử.

## 2.4 Path Coverage

* Kiểm thử tất cả các đường đi logic quan trọng.
* Nếu một hàm có nhiều điều kiện lồng nhau, các trường hợp test phải bao quát tất cả các tổ hợp có thể.

## 2.5 Statement Coverage

* Đo lường tỷ lệ các câu lệnh trong chương trình đã được kiểm thử.
* Đảm bảo mọi câu lệnh trong mã nguồn được chạy ít nhất một lần.

# 3. Mức Coverage Tối Thiểu

* Trong thực tế, các hệ thống phần mềm lớn thường đặt mục tiêu 70%-80% Line Coverage.
* Quan trọng hơn là kiểm thử đủ các logic quan trọng, thay vì chạy theo chỉ số coverage mà bỏ qua các case thực sự có ý nghĩa.
* Đối với các hệ thống quan trọng như tài chính, y tế, mức độ coverage có thể yêu cầu cao hơn để giảm thiểu rủi ro.
* Một số tổ chức đặt mục tiêu 90%+ coverage, nhưng điều này thường đòi hỏi chi phí cao và không phải lúc nào cũng cần thiết.
* Độ phủ càng cao càng tốt, nhưng phải cân nhắc giữa chi phí và lợi ích thực tế.

# 4. Best Practices Khi Viết Unit Test

* Độc lập với Database: Tránh test trực tiếp với database, sử dụng mock data.
* Test cả Happy Case và Edge Case: Đảm bảo kiểm thử cả trường hợp dữ liệu hợp lệ và các trường hợp ngoại lệ.
* Không trùng lặp, dễ bảo trì: Tránh viết các test case thừa thãi hoặc phụ thuộc quá nhiều vào chi tiết triển khai.
* Sử dụng Test-Driven Development (TDD): Viết test trước khi viết mã giúp đảm bảo thiết kế tốt hơn.
* Tích hợp Continuous Integration (CI): Tự động chạy unit test trên CI/CD để đảm bảo code không bị lỗi khi merge.
* Đặt tên test rõ ràng: Một test case nên có tên mô tả chính xác chức năng mà nó kiểm tra.
* Tránh hardcode dữ liệu trong test: Dữ liệu nên được tạo động để tránh phụ thuộc vào giá trị cố định.

# 5. Những Thách Thức Khi Đạt Độ Phủ Code Coverage Cao

Mặc dù code coverage cao có lợi ích lớn, nhưng cũng tồn tại một số thách thức:

* Tốn thời gian và tài nguyên: Viết test coverage cao đòi hỏi thời gian phát triển và bảo trì.
* Không đảm bảo chất lượng code: Độ phủ cao không đồng nghĩa với code chất lượng nếu test không kiểm tra đúng logic.
* Khó áp dụng cho hệ thống phức tạp: Một số hệ thống lớn có kiến trúc khó kiểm thử unit test.
* Nguy cơ viết test kém chất lượng: Nếu chạy theo chỉ số coverage, có thể phát sinh các test không có giá trị thực sự.

# 6. Báo Cáo Cải Tiến Kiểm Thử Ứng Dụng StudentManagement

## 6.1. Khó Khăn Khi Viết Unit Test

Trong quá trình viết Unit Test cho ứng dụng StudentManagement, chúng tôi nhận thấy rằng các hàm trong chương trình hiện tại gặp nhiều vấn đề làm cho việc kiểm thử trở nên khó khăn. Dưới đây là một số khó khăn cụ thể:

* Hàm Làm Quá Nhiều Trách Nhiệm

Các hàm như deleteStudent(), addStudent(), editStudent() vừa nhận dữ liệu từ người dùng (Scanner), vừa thực hiện xử lý dữ liệu, khiến chúng khó test.

Vấn đề: Unit Test không thể nhập dữ liệu vào Scanner, nên không thể kiểm tra tự động.

* Phụ Thuộc Vào Input Từ Người Dùng

Chương trình hiện tại yêu cầu nhập dữ liệu từ Scanner, khiến Unit Test không thể kiểm tra logic xử lý mà không cần nhập tay.

Ví dụ, hàm addStudent() yêu cầu nhập nhiều thông tin, nhưng Unit Test không thể kiểm tra tự động.

## 6.2. Đề Xuất Cải Tiến

Tách Phần Nhập Liệu Ra Khỏi Logic Xử Lý

Cách cải tiến: Chia nhỏ các hàm xử lý mỗi chức năng thành:

* Một hàm xử lý chỉ nhận các tham số để Unit Test có thể gọi trực tiếp.
* Một hàm riêng dùng Scanner để lấy dữ liệu từ người dùng.

Lợi ích:

Khi chạy chương trình, vẫn có thể nhập MSSV bình thường.

Unit Test có thể gọi hàm mà không cần nhập tay.

## 6.3. Kết Luận

Vấn đề chính:

* Các hàm làm quá nhiều trách nhiệm, vừa nhập dữ liệu, vừa xử lý logic.
* Phụ thuộc vào Scanner, gây khó khăn cho Unit Test.

Giải pháp: Tách nhập liệu ra khỏi hàm xử lý logic.

Lợi ích sau cải tiến:

Unit Test có thể kiểm tra logic dễ dàng mà không cần nhập dữ liệu tay.

Code rõ ràng, dễ bảo trì hơn.

Giảm lỗi phát sinh khi thay đổi chương trình.