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

BÁO CÁO

Chủ đề : Unit Test
Môn học: Thiết kế phần mềm
Lớp: 22_3

Sinh viên thực hiện: 22120185 – Võ Văn Lĩnh
22120201 – Huỳnh Mẫn
22120208 – Hoàng Hồ Nhật Minh
22120247 – Nguyễn Hữu Khánh Nhân

Hồ Chí Minh, ngày 03 tháng 04 năm 2025

MỤC LỤC

I. Các loại Code Coverage quan trọng:

1. Line Coverage:

Độ bao phủ dòng, còn được gọi là độ bao phủ câu lệnh, là một số liệu cơ bản đo lường tỷ lệ các dòng mã thực thi đã được thực thi ít nhất một lần bởi bộ kiểm thử. Nó được tính bằng cách chia số dòng thực thi đã được thực thi cho tổng số dòng thực thi trong cơ sở mã. Số liệu này mang lại lợi thế là cung cấp một cái nhìn tổng quan đơn giản về mức độ mã đã được kiểm thử, giúp dễ hiểu và hình dung độ bao phủ. Các nhà phát triển có thể dễ dàng nắm bắt tỷ lệ phần trăm các dòng mã phần mềm của họ được bao phủ bởi unit test, đặc biệt khi kết hợp với các công cụ làm nổi bật các dòng đã thực thi và chưa thực thi trực tiếp trong mã nguồn.

⇨ *Đảm bảo mọi dòng mã thực thi đều được kiểm thử*

Tuy nhiên, độ bao phủ dòng có những hạn chế cố hữu. Nó dễ bị ảnh hưởng bởi những thay đổi trong định dạng mã; một thao tác định dạng lại mã đơn giản có thể thay đổi tỷ lệ phần trăm độ bao phủ mà không có bất kỳ thay đổi thực tế nào về chức năng hoặc hiệu quả của các bài kiểm thử. Hơn nữa, độ bao phủ dòng không tính đến logic phân nhánh trong mã. Ví dụ, một hàm chứa câu lệnh if có thể đạt được độ bao phủ dòng 100% nếu chỉ nhánh then được thực thi, trong khi nhánh else hoặc điều kiện mà câu lệnh if đánh giá là sai vẫn chưa được kiểm thử. Sự hời hợt này có nghĩa là ngay cả với độ bao phủ dòng cao, các đường dẫn thực thi quan trọng có thể không được kiểm thử đầy đủ. Do đó, đối với các ứng dụng quan trọng về an toàn, các tiêu chuẩn thường khuyến nghị các số liệu độ bao phủ mạnh mẽ hơn như độ bao phủ nhánh hoặc Độ bao phủ điều kiện/quyết định đã sửa đổi (MC/DC).

2. Branch Coverage:

Độ bao phủ nhánh là một số liệu nâng cao hơn, giải quyết những hạn chế của độ bao phủ dòng bằng cách tập trung vào việc thực thi tất cả các nhánh hoặc điểm quyết định có thể có trong mã. Các điểm quyết định này thường bao gồm các câu lệnh if, điều kiện else, trường hợp switch và các vòng lặp. Độ bao phủ nhánh đảm bảo rằng mọi kết quả có thể có (đúng hoặc sai) của mỗi điểm quyết định đều được thực hiện ít nhất một lần trong quá trình kiểm thử. Điều này giúp xác định các đường dẫn chưa được kiểm thử trong mã và cải thiện chất lượng mã tổng thể bằng cách phát hiện các lỗi logic mà độ bao phủ dòng có thể bỏ sót.

⇨ *Kiểm thử tất cả các đường dẫn quyết định*

Công thức tính độ bao phủ nhánh là số nhánh đã thực thi chia cho tổng số nhánh trong mã, nhân với 100. Hãy xem xét một hàm Python check_number(num) làm ví dụ:

```
def check_number(num):
    if num > 0:
        return “Positive”
    elif num < 0:
        return “Negative”
    else:
        return “Zero”
```

Hàm này có ba nhánh: `if num > 0`, `elif num < 0` và điều kiện `else`. Để đạt được độ bao phủ nhánh 100%, các trường hợp kiểm thử phải được viết để bao phủ tất cả các khả năng này: một số dương, một số âm và số không. Các trường hợp kiểm thử như `check_number(5)`, `check_number(-3)`và `check_number(0)` sẽ đảm bảo rằng tất cả các nhánh đều được thực thi. Bằng cách đảm bảo rằng tất cả các đường dẫn quyết định đều được kiểm thử, độ bao phủ nhánh cung cấp một đánh giá kỹ lưỡng hơn về tính đầy đủ của việc kiểm thử so với việc chỉ kiểm tra xem mỗi dòng mã đã được thực thi hay chưa.

3. Function Coverage:

Độ bao phủ hàm là một số liệu đánh giá xem tất cả các hàm hoặc phương thức được định nghĩa trong một chương trình đã được gọi và thực thi ít nhất một lần bởi bộ kiểm thử hay chưa. Nó đo lường tỷ lệ phần trăm các hàm trong cơ sở mã được gọi trong quá trình kiểm thử. Mục đích chính của độ bao phủ hàm là đảm bảo rằng các khía cạnh chức năng của mã đang được kiểm thử và không có hàm nào hoàn toàn chưa được kiểm thử. Ví dụ, nếu một ứng dụng chứa 50 hàm khác nhau và bộ kiểm thử thực thi 45 trong số đó, thì độ bao phủ hàm sẽ là 90%.

⇨ *Xác minh việc thực thi tất cả các hàm*

Điều quan trọng là phải phân biệt độ bao phủ hàm với độ bao phủ cuộc gọi. Độ bao phủ hàm tập trung vào việc liệu mỗi hàm trong chương trình đã được thực thi ít nhất một lần hay chưa, trong khi độ bao phủ cuộc gọi đo lường xem tất cả các vị trí gọi có thể có (vị trí mà một hàm được gọi) đã được đạt đến trong quá trình kiểm thử hay chưa. Đạt được độ bao phủ hàm 100% không nhất thiết ngụ ý độ bao phủ cuộc gọi 100%, vì một số hàm có thể được gọi từ nhiều vị trí và không phải tất cả các vị trí này đều được thực hiện bởi các bài kiểm thử. Độ bao phủ hàm cung cấp một cái nhìn tổng quan về các đơn vị mô-đun của mã đang được kiểm thử, đảm bảo rằng tất cả các khối mã được định nghĩa ít nhất đều được gọi trong quá trình thực thi kiểm thử.

4. Path Coverage:

Độ bao phủ đường dẫn là số liệu bao phủ cấu trúc toàn diện nhất, nhằm đảm bảo rằng tất cả các đường dẫn thực thi có thể có trong một chương trình phần mềm đều được kiểm thử. Một đường dẫn biểu thị một chuỗi duy nhất các câu lệnh và nhánh đã thực thi từ điểm vào đến điểm ra của một chương trình. Các đường dẫn khác nhau phát sinh do các lựa chọn được thực hiện trong các câu lệnh điều kiện như if-then-else hoặc các vòng lặp như do-while. Mục tiêu của độ bao phủ đường dẫn là kiểm thử mọi tuyến đường có thể có thông qua logic của ứng dụng.

⇨ *Khám phá tất cả các luồng thực thi có thể có*

Khái niệm về độ phức tạp Cyclomatic, được phát triển bởi McCabe, thường được sử dụng trong độ bao phủ đường dẫn để xác định số lượng đường dẫn độc lập thông qua một chương trình. Số liệu này cung cấp một chỉ số về độ phức tạp của mã và số lượng tối thiểu các trường hợp kiểm thử cần thiết để đạt được độ bao phủ đường dẫn đầy đủ.

Mặc dù độ bao phủ đường dẫn cung cấp sự kiểm tra mã kỹ lưỡng nhất, nhưng việc đạt được độ bao phủ 100% có thể không thực tế, đặc biệt trong các phần mềm phức tạp với nhiều câu lệnh điều kiện và vòng lặp, dẫn đến sự bùng nổ tổ hợp các đường dẫn có thể có. Nỗ lực cần thiết để tạo và duy trì các trường hợp kiểm thử cho mọi đường dẫn có thể rất lớn. Do đó, độ bao phủ đường dẫn thường được ưu tiên cho các mô-đun phần mềm có rủi ro cao, nơi việc kiểm thử toàn diện là rất quan trọng.

II. Mức Coverage:

1. Các khuyến nghị và tiêu chuẩn chung:

Bất chấp bản chất phụ thuộc vào ngữ cảnh của các mục tiêu về độ bao phủ, một mục tiêu thường được chấp nhận trong ngành công nghiệp phần mềm là khoảng 80% độ bao phủ mã. Con số này thường được coi là sự cân bằng hợp lý giữa nỗ lực cần thiết để viết các bài kiểm thử và sự tự tin có được về độ tin cậy của mã. Ví dụ, Google coi 60% độ bao phủ là "chấp nhận được", 75% là "đáng khen ngợi" và 90% là "gương mẫu". Cũng cần lưu ý rằng các nỗ lực unit testing thường nhắm đến tỷ lệ phần trăm độ bao phủ cao hơn một chút so với kiểm thử mức hệ thống. Tuy nhiên, điều quan trọng cần nhắc lại là trọng tâm chính nên là chất lượng và tính phù hợp của các bài kiểm thử hơn là chỉ đạt được một tỷ lệ phần trăm cụ thể. Việc vội vàng để đạt được mục tiêu về độ bao phủ có thể dẫn đến việc tạo ra các bài kiểm thử chỉ đơn giản là chạm vào mọi dòng mã mà không thực sự xác thực các yêu cầu nghiệp vụ cơ bản.

2. Yêu cầu về độ bao phủ đối với các hệ thống quan trọng:

Trong quá trình phát triển các hệ thống quan trọng, chẳng hạn như trong lĩnh vực hàng không vũ trụ, y tế và tài chính, nơi chi phí cho sự cố có thể cực kỳ cao, việc đạt được độ bao phủ mã 100% hoặc rất cao thường là một điều cần thiết. Các ngành công nghiệp này cũng có thể yêu cầu sử dụng các số liệu độ bao phủ cụ thể như Độ bao phủ điều kiện/quyết định đã sửa đổi (MC/DC), cung cấp một đánh giá nghiêm ngặt hơn về việc kiểm thử các biểu thức boolean. Các tiêu chuẩn như DO-178B trong ngành hàng không yêu cầu 100% độ bao phủ điều kiện/quyết định đã sửa đổi và 100% độ bao phủ câu lệnh cho các hệ thống quan trọng nhất về an toàn. Tương tự, tiêu chuẩn IEC 61508 về an toàn chức năng khuyến nghị độ bao phủ 100% cho một số số liệu, với mức độ nghiêm ngặt của khuyến nghị tùy thuộc vào mức độ quan trọng của hệ thống. Những yêu cầu nghiêm ngặt này nhấn mạnh tầm quan trọng của việc điều chỉnh các mục tiêu về độ bao phủ mã cho phù hợp với các rủi ro cụ thể liên quan đến phần mềm đang được phát triển.

III. Best Practices khi viết Unit Test:

1. Viết các bài kiểm thử dễ bảo trì và dễ đọc

Các unit test dễ bảo trì và dễ đọc là điều cần thiết cho sự phát triển lâu dài của bất kỳ dự án phần mềm nào. Bước đầu tiên là viết mã có thể kiểm thử, thường đạt được bằng cách tuân theo các nguyên tắc như Nguyên tắc trách nhiệm duy nhất, đảm bảo rằng mỗi hàm hoặc phương thức thực hiện một nhiệm vụ duy nhất, được xác định rõ ràng. Cấu trúc các bài kiểm thử theo mẫu Arrange-Act-Assert (AAA) giúp tăng cường sự rõ ràng bằng cách tách biệt việc thiết lập các điều kiện tiên quyết và đầu vào (Arrange), việc thực thi mã đang được kiểm thử (Act) và việc xác minh kết quả (Assert). Sử dụng tên rõ ràng và mô tả cho các trường hợp kiểm thử giúp dễ dàng hiểu mục đích của mỗi bài kiểm thử và nhanh chóng xác định các kịch bản thất bại. Giữ cho các bài kiểm thử nhỏ, tập trung và đơn giản, tránh logic phức tạp trong chính mã kiểm thử, góp phần vào khả năng đọc và giảm khả năng đưa ra lỗi trong các bài kiểm thử. Hơn nữa, đảm bảo rằng các bài kiểm thử là xác định, tạo ra cùng một kết quả mỗi khi chúng được chạy bất kể môi trường hoặc thứ tự thực hiện, là rất quan trọng đối với độ tin cậy.

2. Quản lý phụ thuộc hiệu quả thông qua Mocking

Để kiểm thử hiệu quả một đơn vị mã một cách độc lập, thường cần thiết phải quản lý các phụ thuộc của nó vào các phần khác của hệ thống, chẳng hạn như cơ sở dữ liệu, dịch vụ bên ngoài hoặc các mô-đun khác. Điều này thường đạt được thông qua việc sử dụng mocking và stubbing. Các đối tượng mock mô phỏng hành vi của các phụ thuộc này, cho phép đơn vị đang được kiểm thử được xác minh mà

không cần tương tác thực tế với các phụ thuộc thực. Dependency injection là một mẫu thiết kế tạo điều kiện thuận lợi cho khả năng kiểm thử bằng cách truyền các phụ thuộc vào một lớp thay vì khởi tạo chúng bên trong, giúp dễ dàng thay thế các phụ thuộc thực bằng các mock hoặc stub trong quá trình kiểm thử. Trong khi các mock chủ yếu được sử dụng để xác minh hành vi (kiểm tra xem các tương tác với các phụ thuộc có xảy ra như mong đợi hay không), thì các stub cung cấp các đầu vào được kiểm soát cho đơn vị đang được kiểm thử. Mặt khác, các fake là các triển khai nhẹ của các phụ thuộc có thể có một số hành vi thực hạn chế. Điều quan trọng là phải đạt được sự cân bằng trong việc sử dụng mocking; mặc dù nó rất cần thiết để cô lập các bài kiểm thử, nhưng việc mock quá nhiều chi tiết triển khai có thể dẫn đến các bài kiểm thử gắn chặt với các hoạt động bên trong của mã và trở nên dễ vỡ khi tái cấu trúc xảy ra.

3. Kiểm thử toàn diện các trường hợp thành công và các trường hợp đặc biệt

Việc kiểm thử đơn vị kỹ lưỡng không chỉ bao gồm việc xác minh hành vi dự kiến của mã trong điều kiện bình thường (trường hợp "happy path") mà còn đảm bảo rằng nó xử lý các điều kiện bất ngờ hoặc giới hạn một cách duyên dáng (các "edge case"). Các trường hợp đặc biệt bao gồm các kịch bản với đầu vào không hợp lệ, các giá trị nằm ngoài phạm vi hoặc các tập dữ liệu lớn. Việc kiểm thử các kịch bản ít phổ biến này là rất quan trọng để ngăn chặn các lỗi bất ngờ và tăng độ tin cậy tổng thể của mã. Ngoài ra, điều quan trọng là phải xem xét "sad path" hoặc kiểm thử tiêu cực, bao gồm việc cố ý đưa ra các điều kiện lỗi để đảm bảo rằng hệ thống xử lý các lỗi và ngoại lệ một cách thích hợp, cung cấp các thông báo mang tính thông tin mà không bị sập. Một cách tiếp cận cân bằng để kiểm thử cả các kịch bản điển hình và không điển hình dẫn đến phần mềm mạnh mẽ và đáng tin cậy hơn.

4. Tránh những cạm bẫy thường gặp trong Unit Testing

Một số sai lầm phổ biến có thể làm suy yếu hiệu quả của các nỗ lực unit testing. Một cạm bẫy thường gặp là kiểm thử các chi tiết triển khai của mã thay vì hành vi của nó. Các bài kiểm thử gắn chặt với cách mã được viết hơn là những gì nó làm, trở nên mong manh và bị hỏng ngay cả khi chức năng cơ bản vẫn giữ nguyên. Một sai lầm phổ biến khác là không cô lập đúng cách các phụ thuộc, dẫn đến các bài kiểm thử chậm, không đáng tin cậy và dễ bị ảnh hưởng bởi những thay đổi trong các hệ thống bên ngoài. Việc kiểm thử không đầy đủ các trường hợp lỗi cũng là một thiếu sót đáng kể; điều quan trọng là phải xác minh cách mã xử lý các lỗi và tình huống bất ngờ. Các cạm bẫy khác bao gồm việc gộp nhiều trường hợp kiểm thử trong một phương thức kiểm thử duy nhất, điều này có thể che khuất các lỗi và ngăn chặn việc thực thi các bài kiểm thử tiếp theo, và có nhiều bước "act"

trong một bài kiểm thử duy nhất, gây khó khăn cho việc xác định nguồn gốc của lỗi. Cuối cùng, việc sử dụng các giá trị được mã hóa cứng hoặc "chuỗi ma thuật" trong các bài kiểm thử có thể làm giảm khả năng đọc và bảo trì. Tránh những sai lầm phổ biến này là điều cần thiết để tạo ra một bộ unit test có giá trị và bền vững.

NGUỒN THAM KHẢO

1. What is Unit Testing? - AWS
   https://aws.amazon.com/what-is/unit-testing/
2. Unit testing - Wikipedia
   https://en.wikipedia.org/wiki/Unit_testing
3. What is Unit Testing? Best Practices to Do it Right - ACCELQ
   https://www.accelq.com/blog/unit-testing/
4. Unit Testing: Definition, Examples, and Critical Best Practices - Bright Security
   https://brightsec.com/blog/unit-testing/
5. The Ultimate Guide to Unit Testing: Benefits, Challenges, and Best Practices - TestDevLab
   https://www.testdevlab.com/blog/the-ultimate-guide-to-unit-testing
6. How to Decide on Unit Test Coverage - SoftTeco
   https://softteco.com/blog/unit-test-coverage
7. Four common types of code coverage | Articles | web.dev
   https://web.dev/articles/ta-code-coverage
8. Comparing Code Coverage Techniques: Line, Property-Based, and Mutation Testing
   https://svenruppert.com/2024/05/31/comparing-code-coverage-techniques-line-property-based-and-mutation-testing/
9. What is Code Coverage? | Atlassian
   https://www.atlassian.com/continuous-delivery/software-testing/code-coverage
10. 5 Benefits of Unit Testing and Why You Should Care - CSW Solutions
    https://cswsolutions.com/blog/posts/2022/december/5-benefits-of-unit-testing-and-why-you-should-care/
11. Unit Testing: Principles, Benefits & 6 Quick Best Practices - Codefresh
    https://codefresh.io/learn/unit-testing/
12. The Measurable Benefits of Unit Testing - STX Next
    https://www.stxnext.com/blog/measurable-benefits-unit-testing
13. Why unit tests are critical to reliable code and reduced costs | Credera
    https://www.credera.com/insights/why-unit-tests-are-critical-to-reliable-code-and-reduced-costs
14. how unit tests can improve your product development?
    https://cxdojo.com/how-unit-tests-can-improve-your-product-development
15. Unit Testing in Software Development: Key Strategies | DashDevs
    https://dashdevs.com/blog/are-unit-tests-a-wasteful-expenditure-or-a-promising-investment/