UNIT TEST COVERGAGE VÀ BEST PRACTICES

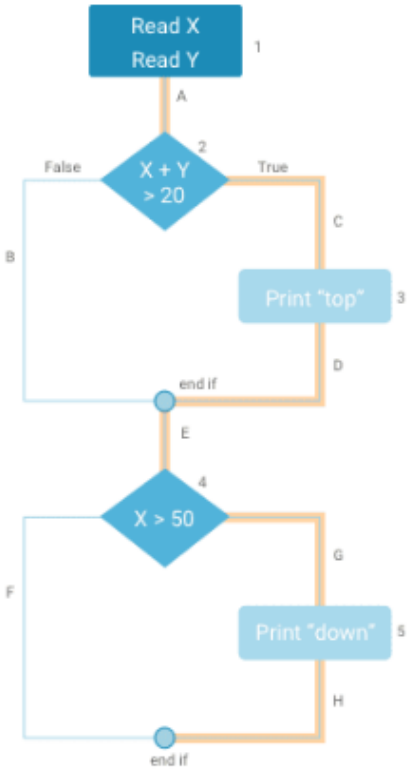
1. **Test Coverage là gì?**

- Test coverage được định nghĩa là một kỹ thuật xác định xem các trường hợp thử nghiệm có thực sự bao trùm mã ứng dụng hay không và bao nhiêu mã được thực hiện khi chạy các trường hợp thử nghiệm đó.
- Nếu có 10 yêu cầu và 100 thử nghiệm được tạo và nếu 90 thử nghiệm được thực hiện thì phạm vi thử nghiệm là 90%. Bây giờ, dựa trên số liệu này, người kiểm tra có thể tạo các trường hợp kiểm tra bổ sung cho các kiểm tra còn lại.
- Dưới đây là một số lợi thế của test coverage.
    - Bạn có thể xác định các lỗ hổng trong yêu cầu, trường hợp kiểm tra và lỗi ở cấp độ sớm và cấp mã.
    - Bạn có thể ngăn ngừa rò rỉ lỗi không mong muốn bằng cách sử dụng phân tích test coverage.
    - Test coverage cũng giúp kiểm tra hồi quy, ưu tiên trường hợp kiểm thử, tăng cường bộ kiểm thử và tối thiểu hóa bộ kiểm thử.

2. **Các kỹ thuật Test Coverage**

2.1 Line/Statement Coverage

- Statement Coverage đảm bảo rằng tất cả các dòng lệnh trong mã nguồn đã được kiểm tra ít nhất một lần. Nó cung cấp các chi tiết của cả hai khối mã được thực thi và thất bại trong tổng số các khối mã.
- Hãy để hiểu nó với ví dụ về sơ đồ sau. Trong ví dụ đã cho, đường dẫn 1A-2C-3D-E-4G-5H này bao gồm tất cả các câu lệnh và do đó nó chỉ yêu cầu trên một trường hợp thử nghiệm để đáp ứng tất cả các yêu cầu. Một trường hợp thử nghiệm có nghĩa là một Statement Coverage...



Path that covers all the statements in the flowchart:

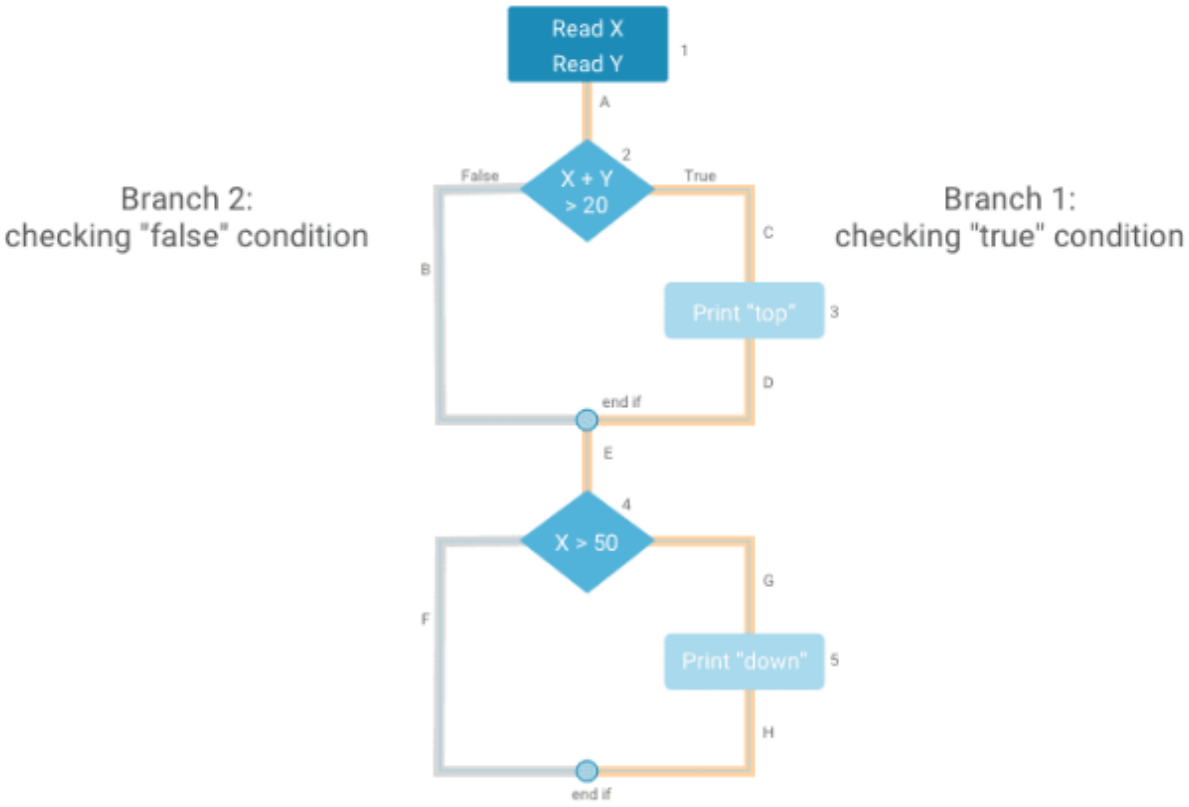
1A - 2C - 3D - E - 4G - 5H

- Trong mã nguồn phức tạp, một đường dẫn không đủ để bao gồm tất cả các câu lệnh. Trong trường hợp đó, bạn cần viết nhiều trường hợp kiểm tra để bao quát tất cả các câu lệnh.
- Ưu điểm:
    - Nó có thể được áp dụng trực tiếp vào mã đối tượng và không yêu cầu xử lý mã nguồn.
    - Nó xác minh những gì mã nguồn viết được dự kiến sẽ thực thi và không thực thi
- Nhược điểm:
    - Nó chỉ bao gồm các điều kiện “true” của mã nguồn.
    - Statement Coverage hoàn toàn không quan tâm với các toán tử logic (|| và &&)

2.2 Decision/Branch coverage

- Các nhà phát triển không thể viết mã trong một chế độ liên tục, tại bất kỳ điểm nào họ cần phân nhánh mã để đáp ứng các yêu cầu chức năng. Sự phân nhánh trong mã thực sự là một bước nhảy từ điểm quyết định này sang điểm khác.

Branch coverage kiểm tra mọi đường dẫn có thể hoặc chi nhánh trong mã được kiểm thử.



- Branch coverage có thể được tính bằng cách tìm số đường dẫn tối thiểu để đảm bảo rằng tất cả các cạnh đã được che phủ. Trong ví dụ đã cho, không có đường dẫn duy nhất đảm bảo vùng phủ sóng của tất cả các cạnh cùng một lúc.
- Ví dụ: nếu bạn đi theo đường dẫn 1A-2C-3D-E-4G-5H này bao gồm số cạnh tối đa (A, C, D, E, G và H), bạn vẫn sẽ bỏ lỡ hai cạnh B và F. Bạn cần đi theo một đường dẫn khác 1A-2B-E-4F để che hai cạnh còn lại. Bằng cách kết hợp hai con đường trên, bạn có thể đảm bảo đi qua tất cả các con đường. Đối với ví dụ này, phạm vi kiểm thử chi nhánh của chúng tôi là 2 vì chúng tôi đang theo hai đường dẫn và nó yêu cầu hai trường hợp thử nghiệm để đáp ứng các yêu cầu.
- Ưu điểm:
    - Nó bao gồm cả các điều kiện đúng và sai không có khả năng được gọi trong statement coverage.
    - Nó đảm bảo tất cả các nhánh được kiểm thử.
- Nhược điểm:

- Nó bỏ qua các nhánh trong các biểu thức Boolean xảy ra do các toán tử ngắn mạch.

2.3 Path Coverage

- Path Coverage là một phương pháp kiểm tra cấu trúc liên quan đến việc sử dụng mã nguồn của chương trình để tìm mọi đường dẫn thực thi có thể.
- Path Coverage đảm bảo phạm vi của tất cả các đường dẫn từ đầu đến cuối.

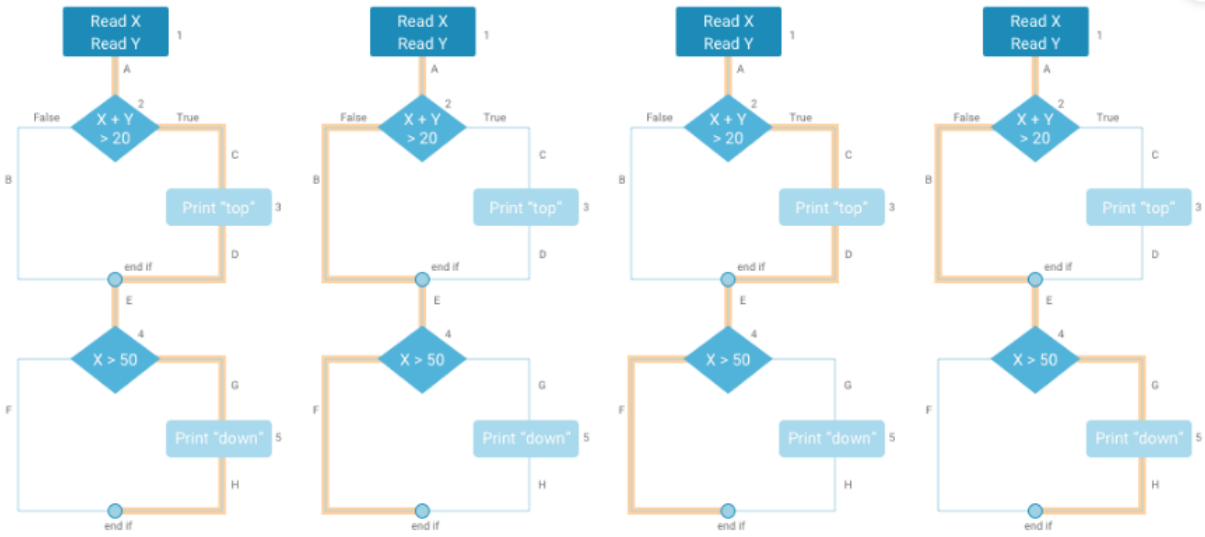
Trong ví dụ này, có bốn đường dẫn có thể kiểm thử:

1A-2B-E-4F

1A-2B-E-4G-5H

1A-2C-3D-E-4G-5H

1A-2C-3D-E-4F



- Ưu điểm:
  - Nó giúp giảm các test thừa.
  - Cung cấp phạm vi kiểm tra cao vì nó bao gồm tất cả các câu lệnh và các nhánh trong mã.
- Nhược điểm:

- Đánh giá mỗi đường dẫn là một thách thức cũng như tốn thời gian vì một số đường dẫn theo cấp số nhân của số nhánh. Ví dụ, một hàm chứa 10 câu lệnh if có 1024 đường dẫn để kiểm tra.
- Đôi khi nhiều đường dẫn không thể thực hiện do mối quan hệ của dữ liệu.

2.4 Condition Coverage

- Condition Coverage sẽ kiểm tra phạm vi điều kiện nếu cả hai kết quả (có nghĩa là "true" hay "fail") của mọi điều kiện đã được thực hiện. Kết quả của điểm quyết định chỉ liên quan để kiểm tra các điều kiện. Nó đòi hỏi hai trường hợp thử nghiệm cho mỗi điều kiện cho hai kết quả.

2.5 Đo lường Coverage Chức Năng (Function Coverage)

- Phương pháp này kiểm tra mức độ bao phủ của các hàm, đảm bảo rằng mỗi hàm trong mã nguồn đều được gọi ít nhất một lần. Phương pháp này hữu ích trong việc phát hiện các hàm không hoạt động hoặc không cần thiết.

3. Mức Coverage Tối Thiểu Theo Lĩnh Vực

- Hệ thống quan trọng (Fintech, Healthcare): Thường yêu cầu mức coverage từ 80% đến 90% để đảm bảo độ tin cậy và an toàn.
- Ứng dụng web, mobile thông thường: Mức coverage từ 60% đến 80% có thể chấp nhận được, tùy thuộc vào yêu cầu cụ thể của dự án.
- Coverage dưới 50%: Được xem là thấp và có thể cho thấy hệ thống chưa được kiểm thử đầy đủ.

➔ Tuy nhiên, cần lưu ý rằng việc đạt 100% coverage không phải lúc nào cũng khả thi hoặc cần thiết, và không đảm bảo rằng phần mềm không có lỗi.

4. Best Practices Khi Viết Unit Test

4.1 Lập kế hoạch kiểm thử

- Mọi dự án nên bắt đầu bằng việc lập kế hoạch, và kiểm thử đơn vị cũng không ngoại lệ. Do tài nguyên có hạn, chúng ta không thể thực hiện kiểm thử đơn vị nhiều lần hoặc dành đều tài nguyên cho tất cả các đơn vị. Vì vậy, cần lập kế hoạch và phân bổ tài nguyên như ngân sách, thời gian và nhân lực cho mỗi kiểm thử đơn vị, đảm bảo quá trình phát triển diễn ra suôn sẻ.
- **Tại sao?**
  - Xác định ngân sách cần thiết và cách sử dụng hiệu quả
  - Tạo ra quy trình làm việc, đặc biệt quan trọng trong công việc nhóm.
  - Sử dụng tài nguyên hợp lý, giảm lãng phí, giúp phát triển thuận lợi hơn.

4.2 Viết kiểm thử sạch và dễ đọc

- Khi viết mã kiểm thử, hãy nhớ rằng đây cũng là tài liệu của phần mềm bạn đang kiểm thử. Không chỉ bạn, mà cả những người khác cũng có thể phải đọc và hiểu nó. Đặt tên kiểm thử rõ ràng để phản ánh chính xác chức năng và mục đích của nó.
- **Tại sao?**
    - Khi kiểm thử thất bại, dễ hiểu ngay mà không cần debug mã.
    - Dễ bảo trì khi mã sản phẩm thay đổi.
    - Giảm hiểu lầm giữa các lập trình viên, tiết kiệm thời gian và công sức.

4.3 Sử dụng mẫu AAA (Arrange, Act, Assert)

- Để đảm bảo kiểm thử dễ đọc và tổ chức tốt, nên sử dụng mẫu **AAA**:
    - **Arrange**: Chuẩn bị dữ liệu đầu vào.
    - **Act**: Thực thi hành động kiểm thử.
    - **Assert**: Kiểm tra kết quả mong đợi.
- **Tại sao?**
    - Mục tiêu kiểm thử rõ ràng trong phần **Arrange**.
    - Quá trình kiểm thử mượt mà trong **Act**. Kết quả kiểm thử rõ ràng, không gây nhầm lẫn trong **Assert**.

4.4 Kiểm thử xác định (Deterministic Tests)

- Cần đảm bảo kiểm thử luôn **xác định**, tức là kiểm thử chỉ có thể **thành công hoặc thất bại**, và kết quả không thay đổi nếu không có sự thay đổi trong mã nguồn. Kiểm thử **không xác định** (nondeterministic tests) có thể gây nhầm lẫn vì kết quả không nhất quán
- **Tại sao?**
    - Kết quả rõ ràng giúp dễ dàng sửa lỗi.
    - Tránh sự hiểu lầm về kết quả kiểm thử.
    - Kiểm thử xác định đáng tin cậy hơn và được chấp nhận rộng rãi hơn.

4.5 Tránh sử dụng logic trong kiểm thử

- Tránh sử dụng các cấu trúc điều khiển như if, while, for, switch trong kiểm thử vì chúng làm giảm tính dễ đọc và có thể gây lỗi. Kiểm thử nên tập trung vào **kết quả mong đợi**, không phải cách thực hiện kiểm thử. Nếu bắt buộc phải sử dụng logic, hãy chia nhỏ kiểm thử.

- **Tại sao?**
    - Kiểm thử dễ đọc hơn khi không có logic phức tạp.
    - Giảm nguy cơ lỗi trong mã kiểm thử.
    - Giúp kiểm thử rõ ràng và dễ bảo trì hơn.

4.6 Độ bao phủ kiểm thử (Test Coverage)

- 100% độ bao phủ kiểm thử là **không khả thi** vì không thể phát hiện hết tất cả lỗi và tài nguyên cũng có hạn. Tuy nhiên, càng nhiều kiểm thử được thực hiện thì khả năng phát hiện lỗi càng cao.
- **Tại sao?**
    - Nhiều kiểm thử hơn đồng nghĩa với nhiều cơ hội phát hiện lỗi hơn.
    - Phát hiện lỗi sớm giúp giảm chi phí sửa lỗi sau này.
    - Làm phần mềm ổn định hơn trước khi phát hành.

4.7 Kiểm thử tự động (Automated Testing)

- Kiểm thử tự động giúp phát hiện lỗi sớm, cung cấp phản hồi nhanh và theo dõi hiệu suất kiểm thử. Nó nhanh hơn nhiều so với kiểm thử thủ công và có thể chạy liên tục mà không cần can thiệp của con người
- **Tại sao?**
    - Có thể chạy nhiều kiểm thử đồng thời và liên tục.
    - Tiết kiệm thời gian và công sức của lập trình viên.
    - Giúp phát hiện lỗi nhanh chóng trong suốt quá trình phát triển.

4.8 Viết kiểm thử song song với quá trình phát triển

- Kiểm thử đơn vị được thực hiện ở giai đoạn đầu của quá trình phát triển phần mềm. Vì vậy, tốt nhất là viết kiểm thử **ngay khi phát triển sản phẩm**, tránh để đến cuối cùng mới viết, vì khi đó một số đoạn mã có thể trở nên **không kiểm thử được**.
- **Tại sao?**
    - Phát hiện sớm các vấn đề có thể phát sinh trong tương lai.
    - Giúp lập trình viên hiểu rõ hơn về mã nguồn.
    - Tránh tình trạng có những đoạn mã không thể kiểm thử sau này.

4.9 Một trường hợp kiểm thử cho mỗi đơn vị (One Use Case per Unit Test)

- Mỗi kiểm thử đơn vị nên chỉ kiểm thử **một trường hợp cụ thể**, giúp dễ dàng xác định lỗi nếu có vấn đề xảy ra. Dù việc này có vẻ mất thời gian hơn, nhưng về lâu dài, nó giúp tiết kiệm công sức và tăng độ chính xác.
- **Tại sao?**
    - Cho kết quả kiểm thử rõ ràng và dễ hiểu.
    - Giúp nhanh chóng xác định nguyên nhân lỗi.
    - Tiết kiệm thời gian trong quá trình bảo trì và phát triển.

4.10 Lưu trữ tài liệu kiểm thử (Test Documentation)

- Tài liệu kiểm thử rất quan trọng vì kết quả kiểm thử thường xuyên được tham chiếu trong các giai đoạn phát triển sau. Khi sản phẩm triển khai và có lỗi phát sinh, tài liệu kiểm thử sẽ giúp nhanh chóng xác định và giải quyết vấn đề.
- **Tại sao?**
    - Giúp mọi thành viên trong nhóm dễ dàng tra cứu.
    - Đảm bảo kiểm thử có thể lặp lại để xác minh kết quả.
    - Lưu trữ và sử dụng tài liệu kiểm thử cho các dự án sau này.