UNIT TEST REPORT

GIỚI THIỆU

Trong phát triển phần mềm hiện đại, unit test đóng vai trò quan trọng trong việc đảm bảo chất lượng, ổn định và khả năng bảo trì của hệ thống. Unit test không chỉ giúp phát hiện lỗi sớm trong quá trình phát triển mà còn hỗ trợ việc refactor code mà không sợ làm ảnh hưởng đến các chức năng cốt lõi. Một trong những chỉ số đo lường hiệu quả của unit test là Code Coverage – độ bao phủ kiểm thử mã nguồn. Tuy nhiên, không phải chỉ số này là tiêu chí duy nhất để đánh giá chất lượng của bộ kiểm thử, mà cách triển khai và thực hiện các unit test theo Best Practices mới thực sự quyết định thành công của quy trình kiểm thử.

CÁC LOẠI COVERAGE QUAN TRỌNG

I. Line Coverage

1. Định nghĩa

Line Coverage (độ bao phủ dòng) đo lường tỷ lệ số dòng code được thực thi trong quá trình chạy test so với tổng số dòng code có trong module hoặc chương trình. Nếu một dòng code được thực thi ít nhất một lần, nó được tính là được bao phủ.

2. Ưu điểm và hạn chế

a. Ưu điểm

- Rất trực quan và dễ đo lường.
- Cho phép phát hiện các đoạn code chưa từng được chạy trong quá trình kiểm thử.

b. Hạn chế

- Không phản ánh được logic điều kiện, vì một dòng code có thể chứa nhiều nhánh logic.
- Có thể tạo cảm giác an toàn giả tạo nếu chỉ dựa vào con số này, trong khi một số nhánh quan trọng có thể chưa được test.

c. Áp dụng trong thực tiễn

Để đảm bảo rằng các dòng code quan trọng được kiểm thử, các công

cụ đo lường như Istanbul, JaCoCo hay Coverage.py được sử dụng phổ biến. Tuy nhiên, cần lưu ý rằng chỉ số line coverage cao không đồng nghĩa với việc code không chứa lỗi logic ẩn. Vì vậy, nó cần được kết hợp với các chỉ số khác để đánh giá chất lượng kiểm thử toàn diện.

II. Branch Coverage

1. Định nghĩa

Branch Coverage tập trung vào việc đảm bảo tất cả các nhánh của các cấu trúc điều kiện như if, else, switch, case đều được thực thi ít nhất một lần. Mục tiêu là kiểm tra các đường đi khác nhau trong luồng điều khiển của chương trình.

2. Ưu điểm và hạn chế

a. Ưu điểm:

- Giúp phát hiện các lỗi ở nhánh logic và kiểm tra các điều kiện phức tạp.
- Đảm bảo rằng không có nhánh nào bị bỏ sót, từ đó tăng độ tin cậy của hệ thống.

b. Hạn chế:

- Có thể phức tạp khi xử lý với các biểu thức điều kiện phức tạp.
- Tăng số lượng test case cần thiết để bao phủ đầy đủ tất cả các nhánh, đôi khi dẫn đến việc test case lặp lại hoặc không thực sự cần thiết.

3. Áp dụng trong thực tiễn

Branch coverage thường được sử dụng để đánh giá các tình huống điều kiện trong code. Khi áp dụng branch coverage, người phát triển cần chú ý đến các trường hợp cạnh (edge cases) và kết hợp với kỹ thuật “decision coverage” để đảm bảo toàn bộ các nhánh điều kiện được kiểm thử đúng đắn.

III. Function Coverage

1. Định nghĩa

Function Coverage đo lường tỷ lệ các hàm, phương thức hoặc module được gọi và thực thi ít nhất một lần trong quá trình chạy test. Đây là một cách đánh giá xem liệu toàn bộ các đơn vị chức năng của code có được kích hoạt hay không.

2. **Ưu điểm và hạn chế**

    a. **Ưu điểm:**

        - Đơn giản và dễ theo dõi, đặc biệt đối với các dự án có cấu trúc module rõ ràng.
        - Giúp phát hiện các hàm không được sử dụng hoặc bị bỏ quên.

    b. **Hạn chế:**

        - Không đảm bảo được rằng logic bên trong hàm được kiểm thử toàn diện.
        - Một hàm có thể được gọi nhưng bên trong lại chứa nhiều nhánh logic chưa được test.

3. **Áp dụng trong thực tiễn**

    Function Coverage thường là bước đầu tiên trong việc đánh giá mức độ bao phủ của các unit test. Nó giúp xác định xem có những hàm nào chưa được kiểm thử. Tuy nhiên, để có cái nhìn chính xác hơn về chất lượng kiểm thử, cần kết hợp với các loại coverage khác như branch và line coverage.

IV. Path Coverage

1. **Định nghĩa**

    Path Coverage là mức độ kiểm thử mạnh mẽ nhất, đo lường việc đảm bảo tất cả các đường đi logic có thể có trong chương trình đều được thực hiện trong quá trình kiểm thử. Điều này đòi hỏi phải kiểm tra mọi kết hợp có thể của các nhánh điều kiện.

2. **Ưu điểm và hạn chế**

    a. **Ưu điểm:**

        - Cung cấp mức độ đảm bảo cao nhất, vì mọi đường đi logic được kiểm thử.
        - Phát hiện được những lỗi phức tạp thường không được phát

hiện qua các loại coverage khác.

b. Hạn chế:

- Khó thực hiện và thường không khả thi đối với các ứng dụng lớn, do số lượng đường đi có thể là vô hạn hoặc quá lớn.
- Yêu cầu số lượng test case vượt trội, dẫn đến tốn kém thời gian và công sức bảo trì.

3. Áp dụng trong thực tiễn

Trong thực tế, việc đạt được 100% path coverage thường là mục tiêu không thực tế đối với các hệ thống phức tạp. Tuy nhiên, các trường hợp quan trọng, đặc biệt là những phần logic phức tạp hoặc những điểm nóng có nguy cơ cao về lỗi, cần được chú trọng kiểm thử theo hướng path coverage. Các kỹ thuật như phân tích luồng điều khiển và sử dụng các công cụ hỗ trợ tự động hoá có thể giúp tối ưu hóa quá trình này.

MỨC COVERAGE TỐI THIỂU

I. Yếu tố xác định mức coverage phù hợp

A. Loại hình ứng dụng:

1. **Ứng dụng web/di động:** Các ứng dụng này thường có nhiều tương tác với người dùng, do đó việc kiểm thử các tình huống thao tác và phản hồi của hệ thống là rất quan trọng. Một mức coverage từ 70% đến 80% có thể là phù hợp nếu các logic cốt lõi được kiểm tra chi tiết.
2. **Hệ thống nhúng và hệ thống an toàn:** Những hệ thống này đòi hỏi mức độ kiểm thử cao hơn, vì lỗi có thể dẫn đến hậu quả nghiêm trọng. Ở đây, mức coverage từ 90% trở lên là cần thiết.

B. Mức độ phức tạp của code:

1. **Code phức tạp với nhiều điều kiện logic:** Khi hệ thống có nhiều nhánh điều kiện, branch coverage và path coverage trở nên quan trọng. Do đó, mức coverage tổng thể cần cao hơn để đảm bảo không bỏ sót bất kỳ trường hợp nào.
2. **Code đơn giản và ổn định:** Ở các module đơn giản, việc đạt mức line coverage cao có thể đã đủ. Tuy nhiên, vẫn cần đảm bảo các tình huống bất thường (edge case) được kiểm thử.

C. Mục tiêu kiểm thử của dự án

1. **Kiểm thử chức năng vs kiểm thử phi chức năng:** Nếu dự án ưu tiên kiểm thử chức năng, mức coverage tập trung vào các hàm và nhánh điều kiện sẽ được ưu tiên. Ngược lại, các hệ thống yêu cầu kiểm thử phi chức năng (ví dụ: hiệu năng, bảo mật) có thể tập trung nhiều hơn vào việc xác định các điểm yếu trong hệ thống thay vì chỉ số coverage.

II. Đề xuất mức coverage tối thiểu

Trong ngành phần mềm hiện nay, nhiều tổ chức đã chia sẻ kinh nghiệm và tiêu chuẩn nội bộ về mức coverage tối thiểu. Một số đề xuất cụ thể bao gồm:

- **Dự án startup hoặc dự án nhỏ:**
  - **Line Coverage:** Tối thiểu 70%
  - **Branch Coverage:** Tối thiểu 60%
  - **Function Coverage:** Đảm bảo tất cả các hàm đều được gọi ít nhất một lần
  - **Path Coverage:** Áp dụng cho các logic quan trọng, không nhất thiết đạt 100% đối với toàn bộ hệ thống
- **Dự án doanh nghiệp hoặc hệ thống có độ quan trọng cao:**
  - **Line Coverage:** Tối thiểu 80% trở lên
  - **Branch Coverage:** Tối thiểu 70% trở lên
  - **Function Coverage:** Đảm bảo không có hàm nào bỏ sót, đặc biệt các hàm xử lý nghiệp vụ quan trọng
  - **Path Coverage:** Tập trung kiểm thử các đường đi logic có nguy cơ cao và phức tạp

III. Đảm bảo kiểm thử các logic quan trọng

Mặc dù mức coverage không cần phải quá cao, nhưng cần đảm bảo kiểm thử đầy đủ các **logic quan trọng**. Cụ thể:

- Kiểm thử tất cả các **business rules** quan trọng.
- Đảm bảo kiểm thử đầy đủ các **điều kiện bất thường** và lỗi ngoại lệ.
- Tập trung vào các **module có mức độ rủi ro cao** thay vì chỉ chạy theo tỷ lệ coverage.

BEST PRACTICE KHI VIẾT UNIT TEST

Việc xây dựng bộ unit test không chỉ dừng lại ở việc đạt được một con số coverage cao. Các test cần phải có chất lượng, dễ bảo trì và đảm bảo kiểm thử các logic cốt lõi một cách hiệu quả. Dưới đây là một số Best Practices quan trọng:

I. Viết unit test dễ bảo trì

a. Tách biệt rõ ràng giữa test case và code sản xuất

- **Mục tiêu:** Giữ cho các unit test không phụ thuộc vào những thay đổi nhỏ của implementation.
- **Cách thực hiện:**
    - Sử dụng các framework kiểm thử như JUnit, NUnit, hoặc pytest để tạo ra cấu trúc test rõ ràng.
    - Chia nhỏ các test case thành các hàm riêng biệt, mỗi hàm chỉ test một khía cạnh cụ thể của chức năng.

b. Không phụ thuộc vào database hoặc persistent storage

- **Vấn đề:** Các unit test phụ thuộc vào database hay persistent storage thường chậm, không ổn định và khó kiểm soát.
- **Giải pháp:**
    - **Mocking và Stubbing:** Sử dụng các thư viện như Mockito (cho Java), Moq (cho .NET) hoặc unittest.mock (cho Python) để thay thế các dependency thật.
    - **Fake Objects:** Tạo ra các đối tượng giả lập để mô phỏng hành vi của các thành phần bên ngoài.

Việc tách biệt dependency giúp cho unit test nhanh chóng, dễ bảo trì và dễ phát hiện lỗi khi có thay đổi từ phía hệ thống bên ngoài.

II. Kiểm thử cả Happy Case lẫn Edge Case

a. Happy Case

- **Định nghĩa:** Happy Case là những trường hợp mà hệ thống hoạt động theo kịch bản mong đợi khi đầu vào hợp lệ.
- **Lợi ích:**
    - Xác nhận rằng logic nghiệp vụ được thực thi đúng đắn khi mọi thứ đều ổn.
    - Giúp đảm bảo tính ổn định của các chức năng cốt lõi.

b. Edge Case

- **Định nghĩa:** Edge Case bao gồm các trường hợp biên, dữ liệu không hợp lệ, tình huống ngoại lệ hoặc những đầu vào không được mong đợi.
- **Lợi ích:**
    - Phát hiện sớm các lỗi tiềm ẩn khi hệ thống gặp các điều kiện bất

thường.
    - Giúp nâng cao độ tin cậy của hệ thống trong môi trường thực tế.

c. Cách triển khai

- **Phân tích đầu vào:** Xác định các giá trị biên, dữ liệu ngoại lệ và các tình huống đặc biệt có thể xảy ra.

- **Viết test case riêng biệt:** Mỗi trường hợp cần có test case riêng, đảm bảo không lẫn lộn giữa happy case và edge case.

- **Sử dụng kỹ thuật parametrized test:** Giúp chạy nhiều biến thể của đầu vào trong cùng một hàm test, đảm bảo tính toàn diện.

III. Tránh viết test trùng lặp và phụ thuộc vào implementation details

a. Lợi ích của việc giảm thiểu sự phụ thuộc vào chi tiết cài đặt

- **Giữ cho test code không bị phá vỡ khi có refactor:** Khi thay đổi cấu trúc bên trong hàm nhưng không ảnh hưởng đến hành vi bên ngoài, các test vẫn không cần thay đổi.
- **Tăng tính tái sử dụng:** Test code chung chung và không bị ràng buộc vào các chi tiết cụ thể sẽ dễ dàng tái sử dụng và bảo trì.

b. Các nguyên tắc cần tuân thủ

- **Test theo giao diện (interface):** Đảm bảo rằng test chỉ dựa vào các giao diện công khai của đối tượng, không can thiệp vào biến cục bộ hay trạng thái bên trong.
- **Sử dụng kỹ thuật abstraction:** Khi cần thiết, tách riêng logic phức tạp ra thành các module riêng, sau đó kiểm thử từng module độc lập.
- **Giữ cho test code đơn giản và dễ hiểu:** Nếu test code trở nên quá phức tạp, khó đọc, khả năng gây nhầm lẫn và khó bảo trì sẽ tăng cao.

KẾT LUẬN

Việc đạt được một mức coverage tối thiểu không nên được coi là mục tiêu cuối cùng, mà là một chỉ số hỗ trợ để đảm bảo rằng các logic quan trọng được kiểm thử đầy đủ. Mỗi dự án có những đặc thù riêng, do đó mức coverage cần được thiết lập dựa trên nhiều yếu tố như độ phức tạp của hệ thống, loại hình ứng dụng, và yêu cầu nghiệp vụ. Một bộ kiểm thử tốt phải không ngừng được cải thiện, phản ánh đúng trạng thái của hệ thống và giúp các developer có cái nhìn toàn diện về chất lượng phần mềm.

Một bộ unit test hiệu quả không chỉ làm tăng độ tin cậy của sản phẩm mà còn giúp tiết kiệm thời gian và công sức trong việc bảo trì và phát triển thêm các tính năng mới. Với sự hiểu biết sâu sắc về các loại code coverage, mức coverage tối thiểu và áp dụng các best practices, các nhà phát triển có thể xây dựng một hệ thống kiểm thử mạnh mẽ, đáp ứng yêu cầu chất lượng ngày càng khắt khe của ngành phần mềm hiện nay.