Trường Đại học Khoa học tự nhiên - ĐHQG TPHCM

Khoa Công Nghệ Thông Tin

Tìm Hiểu về Unit testing và Best practices

Thiết kế phần mềm - 22KTPM3

Giảng viên hướng dẫn

ThS. Trần Duy Thảo

ThS. Hồ Tuấn Thanh

ThS. Nguyễn Lê Hoàng Dũng

Sinh viên thực hiện

Nguyễn Duy Hoàng - 22127126

Nguyễn Phúc Khang - 22127180

Nguyễn Trung Quân - 22127346

Nguyễn Minh Toàn - 22127419

Phan Thị Tường Vi - 22127451

Ngày 28 tháng 2 năm 2025

Mục lục

1 Bảng phân công công việc

Sinh viên	**Công việc**
Nguyễn Duy Hoàng	Tìm hiểu về mức coverage tối thiểu
Nguyễn Phúc Khang	Tìm hiểu về Path Coverage và Function Coverage
Nguyễn Trung Quân	Best Practices khi viết Unit Test
Nguyễn Minh Toàn	Tìm hiểu về Line Coverage và Branch Coverage
Phan Thị Tường Vi	Tìm hiểu chung về Unit Testig và Unit Testing Coverage

2 Giới thiệu chung về Unit Testing và Unit Testing Coverage

2.1 Unit Testing

2.1.1 Định nghĩa

Unit test là mức độ kiểm thử nhỏ nhất trong quy trình kiểm thử phần mềm. Unit test kiểm thử các đơn vị nhỏ nhất trong mã nguồn như method, class, module...Do đó Unit test nhằm kiểm tra mã nguồn của các chương trình, các chức năng riêng rẽ hoạt động đúng hay không.

Unit test có nhiều định nghĩa khác nhau, nhưng tất cả đều tập trung vào ba đặc điểm cốt lõi:

- Xác minh một phần nhỏ của mã nguồn (còn gọi là một đơn vị).
- Thực hiện nhanh chóng để đảm bảo quá trình kiểm thử không ảnh hưởng đến hiệu suất phát triển.
- Chạy trong môi trường độc lập, không bị ảnh hưởng bởi các thành phần bên ngoài.

Hai đặc điểm đầu tiên khá rõ ràng và ít gây tranh cãi. Mặc dù tốc độ của một kiểm thử đơn vị có thể là một khái niệm mang tính chủ quan, nhưng nếu thời gian chạy của bộ kiểm thử đáp ứng yêu cầu của nhóm phát triển, thì có thể coi đó là đủ nhanh.

Tuy nhiên, đặc điểm thứ ba – tính độc lập – là điểm gây nhiều tranh luận nhất. Chính sự khác biệt trong cách hiểu về tính độc lập đã tạo ra hai trường phái kiểm thử đơn vị: trường phái cổ điển và trường phái London. Những khác biệt khác giữa hai trường phái này đều xuất phát từ quan điểm khác nhau về mức độ cô lập cần thiết trong kiểm thử.

2.1.2 Phân loại

- Phương pháp cổ điển (Classical School): Kiểm thử đơn vị có thể tương tác với các thành phần khác mà không cần giả lập.
- Phương pháp London (Mockist School): Sử dụng các mock object để cô lập từng đơn vị kiểm thử, tránh phụ thuộc vào các thành phần bên ngoài.

2.1.3 Ưu điểm

- Giảm thiểu rủi ro phát sinh lỗi trong giai đoạn triển khai.
- Đảm bảo từng phần của hệ thống hoạt động đúng như mong đợi.
- Khi có thay đổi trong mã nguồn, kiểm thử đơn vị giúp phát hiện lỗi sớm.
- Hỗ trợ phát triển theo hướng TDD (Test-Driven Development)

2.1.4 Nhược điểm

- Unit testing chỉ kiểm tra từng đơn vị nhỏ, không thể đảm bảo toàn bộ hệ thống hoạt động đúng.
- Đòi hỏi lập trình viên viết nhiều kiểm thử đơn vị, đặc biệt khi hệ thống phức tạp.
- Khi mã nguồn thay đổi, kiểm thử đơn vị cũng cần được điều chỉnh để phù hợp.

2.2 Unit Testing Coverage

2.2.1 Định nghĩa

Unit Test Coverage (độ bao phủ kiểm thử đơn vị) là một chỉ số quan trọng đo lường mức độ mã nguồn được kiểm tra thông qua các bài kiểm thử đơn vị. Nó giúp đánh giá mức độ toàn diện của bộ kiểm thử và xác định các phần của mã chưa được kiểm tra.

2.2.2 Phân loại

- Statement Coverage: Kiểm tra xem có bao nhiêu câu lệnh trong mã nguồn đã được thực thi bởi các bài kiểm thử.
- Branch Coverage: Xác minh rằng tất cả các nhánh trong câu lệnh điều kiện (if, else, switch-case) đều đã được kiểm thử.
- Function Coverage: Đảm bảo rằng tất cả các hàm hoặc phương thức trong mã nguồn đều đã được gọi ít nhất một lần.
- Condition Coverage: Kiểm tra tất cả các điều kiện logic (&&, ||) để đảm bảo mỗi điều kiện đều được kiểm thử với cả giá trị true và false.
- Path Coverage: Đánh giá tất cả các luồng thực thi có thể có của một chương trình.

2.2.3 Ưu điểm

- Giúp phát hiện các lỗi tiềm ẩn trước khi phần mềm được triển khai.
- Đảm bảo rằng tất cả các phần quan trọng của mã đều được kiểm thử.
- Khi có thay đổi trong mã nguồn, độ bao phủ kiểm thử giúp đảm bảo rằng các thay đổi không ảnh hưởng đến các phần khác của hệ thống.
- Nhược điểm các lỗi chưa được kiểm tra có thể gây ra sự cố trong môi trường sản phẩm.

2.2.4 Nhược điểm

- Độ bao phủ cao không có nghĩa là phần mềm hoàn toàn không có lỗi, vì có thể một số trường hợp đặc biệt vẫn chưa được kiểm thử.
- Việc cố gắng đạt tỷ lệ bao phủ cao có thể dẫn đến các bài kiểm thử không có giá trị thực tế.
- Một bài kiểm thử có thể thực thi mã nhưng không kiểm tra đúng logic hoặc kết quả mong đợi.

3 Các loại Code Coverage quan trọng

3.1 Line Coverage

3.1.1 Định nghĩa

Line Coverage là một số liệu trong kiểm thử phần mềm, phản ánh tỷ lệ số dòng mã đã được thực thi trong quá trình chạy thử nghiệm. Mục tiêu chính là đảm bảo các dòng mã quan trọng đã được kiểm thử và không có đoạn mã nào bị bỏ sót.

3.1.2 Cách tính

$$Line\ Coverage = \left(\frac{Số\ dòng\ mã\ đã\ thực\ thi}{Tổng\ số\ dòng\ mã\ có thể\ thực\ thi} \right) \times 100\%$$

3.1.3 Ví dụ

Xét hàm đơn giản sau trong Python:

```
def divide(a, b):
    if b == 0:
        return "undefined"
    return a/b
```

Bài test sau chỉ kiểm tra phép chia với giá trị khác 0:

```
def test_divide():
    assert divide(1, 2) == 0.5
```

Khi chạy bài test này, các dòng sau sẽ được thực thi: def divide(a, b): if b == 0: return a/b Dòng return "undefined"không được thực thi. Do đó, độ bao phủ dòng là 75

3.1.4 Ưu điểm

- Đơn giản: Dễ hiểu và dễ tính toán.
- Dễ đo lường: Các công cụ test có thể tự động theo dõi độ bao phủ dòng.
- Chỉ số chất lượng cơ bản: Độ bao phủ dòng thấp (<30%) là dấu hiệu của việc test chưa đầy đủ.

3.1.5 Nhược điểm

- Ảnh hưởng khi tái cấu trúc mã: Việc rút gọn mã có thể tăng % Line Coverage một cách giả tạo mà không đảm bảo test tốt hơn.

```
def divide(a, b):
  return "undefined" if b == 0 else a/b
```

- Mặc dù cách viết này ngắn gọn hơn, nhưng trong những đoạn mã phức tạp hơn, nếu lập trình viên chỉ tập trung tối ưu hóa Line Coverage, họ có thể vô tình làm cho mã nguồn trở nên khó đọc và khó bảo trì.
- Không đảm bảo chất lượng: 100% Line Coverage không đồng nghĩa với việc tất cả trường hợp quan trọng đã được kiểm thử.
- Kiểm thử hời hợt: Chạy tất cả các dòng mà không có assert xác thực kết quả thì không có giá trị. Nếu đặt mục tiêu coverage cao, lập trình viên có thể viết test vô nghĩa chỉ để tăng % mà không cải thiện kiểm thử.
- Không tính đến thư viện bên ngoài: Ví dụ, nếu gọi sum([1, 2, 3]), Line Coverage không thể kiểm tra logic bên trong của sum().

3.1.6 Công cụ hỗ trợ

- Python: coverage.py
- Java: JaCoCo, Cobertura
- JavaScript: Istanbul (nyc), Jest
- C#: dotCover, Visual Studio Coverage

3.2 Branch Coverage

3.2.1 Định nghĩa

Branch Coverage đo lường tỷ lệ các nhánh điều kiện (if-else, switch-case, vòng lặp có điều kiện) đã được thực thi. Nó đảm bảo rằng tất cả các nhánh logic của chương trình đều được kiểm thử.

3.2.2 Cách tính

$$Branch\ Coverage = \left(\frac{Số\ nhánh\ đã\ thực\ thi}{Tổng\ số\ nhánh\ cóthể\ thực\ thi} \right) \times 100\%$$

3.2.3 Ví dụ

```
def divide(a, b):
```

```
    if b == 0:
        return "undefined"
    return a/b
```

Hàm này có hai nhánh:

- Khi b == 0
- Khi b != 0

Bài test sau chỉ kiểm thử khi b != 0:

```
def test_divide():
    assert divide(1, 2) == 0.5
```

Lúc này, độ bao phủ nhánh là 50% vì chưa kiểm thử trường hợp b == 0. Để đạt 100%, cần thêm test:

```
def test_divide_by_zero():
    assert divide(1, 0) == "undefined"
```

Bây giờ cả hai nhánh đều được test, dẫn đến 100% độ bao phủ nhánh.

3.2.4 Ưu điểm

- Chính xác hơn Line Coverage: Ví dụ:

  ```
  def divide(a, b):
      return "undefined" if b == 0 else a/b
  ```

  - Nếu chỉ test với b != 0, Line Coverage đạt 100%, nhưng Branch Coverage chỉ 50
  - Điều này cho thấy Branch Coverage phản ánh tốt hơn mức độ kiểm thử.

- Xác định logic chưa được test: Giúp phát hiện nhánh chưa được kiểm tra. Ví dụ:

  ```
  def complex_function(a, b, c):
      if a and (b or c):
          return True
      return False
  ```

  - Hàm này có nhiều nhánh tiềm năng dựa trên các kết hợp giá trị khác nhau của a, b và c. Độ bao phủ nhánh giúp đảm bảo chúng ta test kỹ lưỡng các kết hợp này.

- Ít bị ảnh hưởng bởi tái cấu trúc: Ít phụ thuộc vào cách viết mã hơn Line Coverage.
- Bao phủ các trường hợp biên: Khuyến khích kiểm thử các điều kiện biên.

3.2.5 Nhược điểm

- Không đảm bảo kiểm thử đầy đủ: 100% Branch Coverage không có nghĩa tất cả các trường hợp biên đã được kiểm tra.
- Không đảm bảo tính đúng đắn: Giống Line Coverage, Branch
- Coverage không có assert vẫn không xác minh được kết quả và không kiểm tra nhánh trong thư viện bên ngoài.
- Khó hiểu hơn Line Coverage: Phức tạp hơn khi phân tích và viết test.

3.2.6 Công cụ hỗ trợ

Tương tự như Line Coverage, các công cụ hỗ trợ Line Coverage đều hỗ trợ Branch Coverage

3.3 Function Coverage

3.3.1 Định nghĩa

Function Coverage đo lường xem tất cả các function trong code có được gọi ít nhất một lần trong bộ unit test hay không.

3.3.2 Cách tính

$$\text{Function Coverage} = \left(\frac{\text{Số hàm được gọi}}{\text{Tổng số hàm}} \right) \times 100\%$$

3.3.3 Ví dụ

Nếu có 3 hàm mà test chỉ gọi 2 hàm thì Function Coverage = 2/3

3.3.4 Ưu điểm

- Dễ tính toán.
- Đảm bảo không có function nào bị bỏ sót.

3.3.5 Nhược điểm

- Không kiểm tra xem function có được kiểm thử với đủ các trường hợp hay không.
- Không đảm bảo rằng tất cả các dòng code bên trong function đều được thực thi.
- Cần kết hợp với các loại coverage khác như branch và path coverage để đánh giá toàn diện.

3.4 Path Coverage

3.4.1 Định nghĩa

Path Coverage đo lường tất cả các đường đi (execution paths) có thể có trong một function. Có thể hiểu là 1 luồng thực thi, chuỗi các hành động, các logic. bao gồm cả function, loop coverage (đảm bảo các vòng lặp được thực hiện và số lần nó thực hiện), call coverage (1 hàm gọi 1 hàm khác)

3.4.2 Cách tính

$$\text{Path Coverage} = \left(\frac{\text{Số đường đi đã kiểm tra}}{\text{Tổng số đường đi có thể có}} \right) \times 100\%$$

3.4.3 Ví dụ

```
def check_number(n)
   if (n>0)
        print("Positive")
    else:
        print("Negative")
```

Code này có 2 paths khả thi:

- n $>$ 0 $\rightarrow$ print("Positive")
- n $<=$ 0 $\rightarrow$ print("Negative or Zero")
- Nếu unit test chỉ kiểm tra n = 5 (path 1) mà không kiểm tra n = -3, thì Path Coverage chỉ đạt 50

3.4.4 Ưu điểm

- Đảm bảo kiểm thử tất cả các đường đi, trường có thể xảy ra trong chương trình.
- Có lợi thế trong phát hiện các lỗi liên quan đến logic điều kiện ít gặp, lỗi tiềm ẩn.

3.4.5 Nhược điểm

- Do kiểm thử rất chi tiết nên số lượng paths tăng theo cấp số nhân khi có nhiều điều kiện (if-else, loops), gây khó khăn khi kiểm thử.
- Không thực tế để đạt 100% path coverage trong các ứng dụng doanh nghiệp lớn vì số lượng path quá nhiều

4 Mức coverage tối thiểu

Theo Vladimir Khorikov – tác giả của sách **Unit Testing Principles, Practices and Patterns** đã nhấn mạnh nhiều lần trong sách của mình: "Mức coverage là một chỉ số tốt để cảnh báo những thiếu

sót, nhưng lại tệ trong việc phản ánh một chất lượng tốt". Điều này có nghĩa, nếu mức độ coverage thấp, cho là dưới 60%, thì đó là dấu hiệu của các lỗi ẩn và các logic chưa được kiểm tra đầy đủ; trong khi mức độ coverage cao thật sự không mang nhiều ý nghĩa. Nó chỉ cho biết rằng nhiều dòng code đã được thực thi trong quá trình kiểm thử , nhưng không đảm bảo tất cả các trường hợp hay các điều kiện quan trọng của logic đã được kiểm tra một cách sâu sắc.

Hãy tưởng tượng một bệnh nhân trong bệnh viện. Nhiệt độ cao của họ có thể báo hiệu bệnh sốt và đó là một quan sát hữu ích. Tuy nhiên, bệnh viện không nên đặt mục tiêu duy trì nhiệt độ "đúng"của bệnh nhân bằng bất cứ giá nào. Nếu không, bệnh viện có thể sẽ đưa ra giải pháp nhanh chóng và "hiệu quả" là lắp đặt một máy lạnh ngay bên cạnh bệnh nhân và điều chỉnh nhiệt độ của họ bằng cách tăng giảm lượng không khí lạnh thổi vào da. Dĩ nhiên, cách làm này hoàn toàn vô lý.

Tương tự vậy, việc bắt buộc phải đảm bảo một mức độ coverage cụ thể đề ra đi ngược lại với mục đích của unit testing. Thay vì tập trung vào những phần quan trọng cần được kiểm tra, mọi người lại tìm cách chinh phục mục tiêu nhân tạo này. Việc unit testing đúng cách vốn đã khó khăn rồi, thì bắt buộc đạt được một con số coverage nhất định chỉ khiến mọi người xao nhãng, không tập trung vào việc suy nghĩ kỹ lưỡng về những gì cần kiểm thử, và qua đó làm cho việc kiểm thử đơn vị đúng cách càng trở nên khó khăn hơn.

Trong nhiều dự án và tổ chức, mức unit test coverage từ 70% đến 80% được xem là phù hợp. Con số này cho thấy phần lớn code đã được kiểm thử mà không làm tăng quá nhiều chi phí và thời gian viết test. Đối với các hệ thống có mức độ rủi ro cao (như hệ thống tài chính, y tế), có thể cần tăng coverage cho những module quan trọng. Trong các hệ thống không quá phức tạp, việc tập trung vào các logic chính thay vì toàn bộ codebase có thể là lựa chọn hợp lý.

Hãy nhớ rằng mục tiêu của unit testing là đảm bảo rằng các tính năng và logic quan trọng của hệ thống hoạt động đúng đắn, chứ không phải chỉ đạt được một con số coverage ấn tượng. Các module có tính chất rủi ro cao hoặc chứa các thuật toán phức tạp cần được kiểm thử kỹ càng, có thể đạt coverage gần 100%. Các phần mã đơn giản, tự giải thích hoặc có mức rủi ro thấp không cần thiết phải đạt con số cao nếu đã được kiểm thử đủ các trường hợp cần thiết.

Chỉ số này nên được sử dụng như một công cụ cảnh báo, không phải là mục tiêu cuối cùng của quy trình kiểm thử.

5 Best Practices khi viết Unit Test

"Best Practice" là tập hợp những kỹ thuật và cách làm mà đã được nghiên cứu, chứng minh trong thực tế, khi áp dụng nó sẽ làm cho sản phẩm của chúng ta tốt hơn, tránh được rất nhiều các vấn đề mà người khác đã gặp phải

1. Tách biệt giữa phần test và code

    - Chúng ta cần có ít nhất 2 folder một cho code và một cho test. Ví dụ trong scala + play2 có thể đặt code trong project/app, test trong project/test. Ngoài việc giúp quản lý code dễ dàng hơn, đây còn là yêu cầu của rất nhiều công cụ quản lý source code

2. Tên package giống nhau giữa test code & source code

    - Khi test code được tổ chức giống như source code sẽ giúp tìm test code dễ dàng hơn và ngược lại. Điều này rất có ý nghĩa trong các dự án lớn với hàng trăm file code & test code.

3. Đặt tên class test dựa trên file code

    - Trong các dự án lớn, số lượng các file code rất lớn. Khi đó nếu các file test không được đặt tên theo một quy tắc nào đó sẽ rất khó tìm file test code. Để giải quyết vấn đề này thì một quy tắc hay được dùng là, đặt tên class test bao gồm tên class cần test và hậu tố "Test" hoặc "Spec" ở cuối. Ví dụ ta có class cần được viết test là ApplicationController.scala, class test sẽ là ApplicationControllerTest.scala hoặc ApplicationControllerSpec.Scala

4. Đặt tên phương thức test mô tả đầy đủ ý nghĩa

    - Giúp hiểu đầy đủ ý nghĩa của phương thức test mà không cần xem comment
    - Có rất nhiều cách để đặt tên test method. Nhưng cách được ưa chuộng là đặt tên dùng Given/When/Then trong khái niệm của BDD. Given – đưa ra điều kiện, When – miêu tả hành động, Then – mô tả kết quả mong đợi. Nếu một số test không có điều kiện trước, thì Given có thể bỏ qua

5. Viết test trước khi viết code

    - Bằng cách viết hoặc sửa đổi test code trước khi viết, người phát triển sẽ tập trung hơn vào yêu cầu trước khi bắt đầu code. Điều này là điểm khác biệt lớn nhất so với việc viết test sau khi viết code. Hơn nữa việc viết test trước giúp chúng ta tăng chất lượng của test code, tránh vấn đề viết test một cách qua loa.

6. Chạy tất cả các test mỗi lần thay đổi code

    - Mỗi lần thay đổi bất kỳ phần nào trong code, dù lớn hay nhỏ, ta cũng cần chạy lại tất cả các test. Một cách lý tưởng thì các test có thể chạy nhanh ngay trên máy của người phát triển để họ không phải đợi quá lâu. Mỗi khi code được đưa lên git hoặc svn, cần chạy các test lại để đảm bảo rằng không có vấn đề gây ra do merge code. Điều này đặc biệt quan trọng khi có nhiều người cùng tham gia phát triển code. Có thể setup công việc này một cách tự động dùng các phần mềm như Jenkins hay các công cụ CI khác được cung cấp bởi các nền tảng Git như Github Actions, Gitlab CI/CD, Bitbucket Pipelines.

7. Kiểm tra code(Refactor) chỉ sau khi tất cả test đã thành công

    - Nếu tất cả test chạy thành công thì nó tương đối an toàn để kiểm tra code. Sau khi refactor code thì không cần phải viết thêm test mới, nhưng sẽ cần sửa đổi test code cho những phần đã thay đổi. Điều kiện lý tưởng là, sau khi refactor thì tất cả các test chạy đều thành công

8. Hạn chế sự phụ thuộc giữa các test

    - Mỗi test nên độc lập từ các test khác. Người phát triển nên có thể thực thi bất kỳ phương thức test nào đó, hoặc một tập các test độc lập. Nếu có sự phụ thuộc từ các test thì chúng dễ bị ảnh hưởng khi viết các test mới.

9. Các test nên chạy nhanh

    - Nếu test chạy mất nhiều thời gian, người phát triển sẽ dễ dàng dừng chúng hoặc chỉ chạy một tập nhỏ của các test liên quan tới phần họ sắp thay đổi. Rõ ràng điều này là không tốt vì không thể đảm bảo tất cả các test sẽ thành công khi viết test mới hoặc code mới. Ưu điểm từ chạy nhanh là bên cạnh việc tích kiệm thời gian, nó còn giúp phát hiện các vấn đề sớm và có thể giải quyết vấn đề sớm.

10. Dùng các công cụ để hỗ trợ test

    - Code Coverage: Là công cụ đo phần trăm code được test trong dự án. Có thể dùng một trong các tool như SonarQuebe, Clover. Các công cụ này hỗ trợ hầu hết các ngôn ngữ
    - CI – Continuous integration (CI): Tool để thực hiện các thao tác tự động theo một kế hoạch lập từ trước đó. Có thể dùng các công cụ như: Jenkins, Hudson

11. Dùng setup và tear-down phương thức

    - Phương thức setup cho phép cài đặt dữ liệu trước khi chạy unit test, tear-down để reset dữ liệu sau khi test chạy xong. Điều này đảm bảo các test không bị phụ thuộc lẫn nhau. Dữ liệu sinh ra do test sẽ được reset lại sau khi nó chạy xong.

Tài liệu

[1] Flinters - "Best Practice" cần thiết khi viết TDD https://labs.flinters.vn/none/best-practice-can-thiet-khi-viet-tdd/. Ngày truy cập: 27/02/2024

[2] topdev.vn - Tại sao Test Coverage là một phần quan trọng của Kiểm thử phần mềm? https://topdev.vn/blog/tai-sao-test-coverage-la-mot-phan-quan-trong-cua-kiem-thu-phan-mem/. Ngày truy cập: 26/02/2025

[3] viblo.asia - Statement, Branch and Path Coverage Testing https://viblo.asia/p/statement-branch-and-path-coverage-testing-63vKjwWdZ2R. Ngày truy cập: 27/02/2025

[4] Vladimir Khorikov - Unit Testing Principles, Practices, and Patterns - Manning Publications (2019) https://www.amazon.com/Unit-Testing-Principles-Practices-Patterns/dp/1617296279. Ngày truy cập: 25/02/2025