***Báo Cáo Tìm Hiểu Công Cụ Unit Test***

**Lớp: Công nghệ phần mềm (4)**

**Giáo viên: Lê Văn Minh**

**Đề Tài : Phần Mềm Quản Lý Thư Viện**

***Nhóm : PHP***

***Tên thành viên :***

***Nguyễn Văn Phàn - 15si***

***Trương Thanh Hưng - 15si***

***Lâm Hồ Phương - 15si***

***Đặng Lê Sơn Tùng – 15si***

## Cách kiểm tra :<https://youtu.be/BsrMZ_K-92A>

## <https://youtu.be/oKj5aDF3EwA>

## Unit test không phải dùng để tìm bug

Hiện giờ tôi mạnh mẽ ủng hộ việc sử dụng unit test, nhưng chỉ khi bạn hiểu được vai trò của unit test trong quy trình Test Driven Development (TDD), và bỏ đi bất kỳ quan niệm sai lầm rằng các unit test là để kiểm thử và tìm kiếm bug.

Theo kinh nghiệm của tôi, các unit test không phải là cách hiệu quả để tìm kiếm bug hoặc phát hiện hồi quy (detect regression). Theo định nghĩa, thì các unit test kiểm tra mỗi đơn vị (unit) code của bạn một cách riêng biệt. Nhưng khi ứng dụng của bạn chạy trong thực tế, tất cả những đơn vị phải làm việc cùng nhau, toàn bộ sẽ trở nên phức tạp và tinh tế hơn so với tổng số các phần của nó đã được kiểm thử độc lập. Việc chứng minh rằng các thành phần X và Y có thể làm việc độc lập không chứng tỏ rằng chúng tương thích với nhau hoặc được cấu hình một cách chính xác. Ngoài ra, các khuyết điểm trong một thành phần riêng biệt có thể chẳng liên quan đến các triệu chứng mà người dùng sẽ trải nghiệm và báo lại. Khi bạn thiết kế các điều kiện tiên quyết cho các unit test của mình, chúng sẽ không bao giờ phát hiện các vấn đề gây ra bởi những điều kiện tiên quyết mà bạn không lường trước được (ví dụ, như một số can thiệp không mong muốn của IHttpModule trong các request gửi đến).

Vì vậy, nếu bạn đang cố gắng tìm kiếm bug, thì có một cách hiệu quả hơn nhiều đó là hãy chạy toàn bộ ứng dụng với nhau như nó sẽ chạy trong môi trường thực tế, giống như bạn làm công việc kiểm tra thủ công một cách tự nhiên vậy. Nếu bạn tự động hóa kiểu test này để phát hiện những sai sót khi chúng xảy ra trong tương lai, đây được gọi là integration testing và thường sử dụng các kỹ thuật và công nghệ khác hơn là unit test. Bạn có muốn sử dụng các công cụ thích hợp nhất cho mỗi công việc không?

|  |  |
| --- | --- |
| **Mục tiêu** | **Kỹ thuật mạnh nhất** |
| **Tìm kiếm bug** (những thứ không làm việc như bạn muốn) | **Kiểm thử thủ công** (đôi khi cả kiểm thử tích hợp tự động) |
| **Phát hiện hồi quy** (những thứ đang làm việc tốt nhưng đã bất ngờ ngừng hoạt động) | **Các kiểm thử tích hợp** tự động (đôi khi cũng bao gồm cả kiểm thử thủ công, mặc dù khá tốn thời gian) |
| **Thiết kế các thành phần trong phần mềm** mạnh mẽ | **Unit testing** (trong quy trình TDD) |

(Lưu ý: có một ngoại lệ, nơi các unit test rất hiệu quả trong việc phát hiện bug. Đó là khi bạn đang tái cấu trúc code của một đơn vị (unit) nhưng không có nghĩa là để thay đổi hành vi của nó. Trong trường hợp này, các unit test có thể cho bạn biết nếu hành vi của đơn vị (unit) đó có thay đổi hay không.)

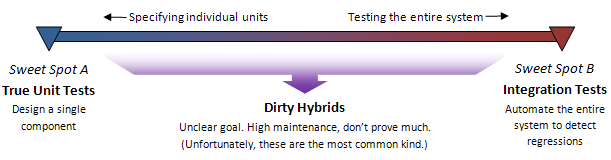
## Vậy nếu unit test không phải để tìm bug, thì nó để làm gì?

, nhưng vì quan niệm sai lầm cố hữu trong kiểm thử cứ treo lơ lửng trong tâm trí của các nhà phát triển phần mềm, tôi sẽ nói lại cái nguyên tắc của nó. Một bậc thầy về TDD đã nói rằng, "TDD là một quá trình thiết kế, chứ không phải là một quá trình kiểm thử". Hãy để tôi giải thích thêm: **TDD là một cách mạnh mẽ trong việc thiết kế các thành phần phần mềm ("units") tương tác để hành vi của chúng được xác định thông qua các unit test.** Tất cả chỉ có vậy!

## Những unit test tốt vs dở

TDD sẽ giúp bạn cung cấp các thành phần phần mềm hoạt động theo thiết kế của bạn. **Một bộ unit test tốt là vô cùng có giá trị:** nó là tài liệu thiết kế của bạn, khiến cho việc tái cấu trúc và mở rộng code của bạn được dễ dàng hơn trong khi giữ lại một cái nhìn tổng quan rõ ràng về hành vi của mỗi thành phần. Tuy nhiên, **một bộ unit test tồi là vô cùng đau thương:** nó chẳng chứng tỏ bất cứ điều gì rõ ràng cả, và có thể làm cản trở nghiêm trọng khả năng tái cấu trúc hoặc thay đổi code của bạn theo bất kỳ cách nào.

Các kiểm thử của bạn nằm ở đâu trên thang mức độ sau?



Các unit test được tạo ra thông qua quy trình TDD dĩ nhiên nằm ở cực trái của thang mức độ ở trên. Chúng chứa rất nhiều kiến ​​thức về hành vi của một đơn vị code riêng lẻ. Nếu hành vi của đơn vị đó thay đổi, vì vậy phải thực hiện unit test cho đơn vị đó, và ngược lại. Nhưng chúng không có bất kỳ kiến ​​thức hoặc giả định về các phần khác trong codebase của bạn, nên **những thay đổi ở các bộ phận khác trong codebase của bạn không làm chúng bị lỗi** (và nếu chúng bị lỗi, thì điều đó cho thấy chúng không phải là những unit test đúng). Do đó việc duy trì chúng khá dễ dàng, và như là một kỹ thuật phát triển, TDD có thể áp dụng cho bất kỳ kích thước của dự án nào.

Ở đầu kia của thang mức độ, các kiểm thử tích hợp (integration test) không biết về cách codebase của bạn được chia nhỏ thành các đơn vị (unit), nhưng thay vào đó nó cho thấy cách toàn bộ hệ thống ứng xử đối với người dùng bên ngoài như thế nào. Chúng có chi phí hợp lý để duy trì (vì không quan trọng việc bạn tái cấu trúc hoạt động nội bộ trong hệ thống của bạn như thế nào, nó sẽ không  ảnh hưởng đến hành vi bên ngoài) và chúng chứng tỏ được rất nhiều về những tính năng thực sự làm việc hiện nay.

Bất cứ nơi nào ở giữa thang mức độ trên, nó không rõ ràng về những giả định mà bạn đang làm và những gì bạn đang cố gắng chứng minh. Việc tái cấu trúc có thể làm hỏng những kiểm thử này, hoặc cũng có thể không, bất kể trải nghiệm của người dùng vẫn hoạt động. Việc thay đổi các dịch vụ bên ngoài mà bạn sử dụng (chẳng hạn như việc nâng cấp cơ sở dữ liệu) có thể làm hỏng những kiểm thử này, hoặc cũng có thể không, bất kể trải nghiệm của người dùng vẫn hoạt động. Bất kỳ thay đổi nhỏ trong các hoạt động nội bộ của một đơn vị riêng lẻ có thể buộc bạn phải sửa lại hàng trăm hybrid test mà dường như chẳng liên quan gì cả, vì vậy chúng có xu hướng tiêu tốn một lượng lớn thời gian để bảo trì - đôi khi nó gấp 10 lần số lượng thời gian mà bạn dùng để bảo trì phần code thực sự của ứng dụng đó. Và nó cũng gây nản lòng vì bạn biết rằng việc bổ sung thêm nhiều điều kiện tiên quyết để thực hiện các hybrid test chuyển sang màu xanh thì không thực sự chứng tỏ được điều gì cả.

## Bí quyết để viết những unit test tuyệt vời

Vừa rồi là những thảo luận khá mơ hồ - và đây là lúc để đưa ra một số lời khuyên thiết thực. Dưới đây là một số hướng dẫn để viết các unit test mà luôn nằm ở vị trí Sweet Spot A của thang mức độ ở trên, và cũng có những giá trị theo cách khác nữa.

**Hãy làm cho mỗi test độc lập với tất cả những phần khác**

Bất kỳ hành vi nhất định nào cũng nên được xác định tại một và chỉ duy nhất một test. Nếu không, sau này khi bạn thay đổi hành vi đó, bạn sẽ phải thay đổi rất nhiều test. Các hệ quả của nguyên tắc này bao gồm:

* Đừng làm những assertion không cần thiết

Những hành vi cụ thể nào mà bạn đang kiểm thử? Nó sẽ gây phản tác dụng nếu cứ Assert() bất cứ thứ gì mà cũng đã được assert bằng test khác: nó chỉ làm tăng tần suất của những thất bại vô nghĩa mà chẳng cải thiện unit test một chút nào cả. Điều này cũng áp dụng cho việc gọi Verify() không cần thiết - nếu nó không phải là hành vi cốt lõi dưới test đó, thì hãy dừng việc quan sát về nó! Đôi khi, cộng đồng TDD thường diễn đạt điều này bằng cách nói rằng "chỉ có duy nhất một assertion hợp lý trên mỗi test".

Hãy nhớ rằng, các unit test là một thiết kế chỉ rõ một hành vi nào đó sẽ làm việc như thế nào, không phải là một danh sách các quan sát của tất cả mọi thứ mà phần code đó thực hiện.

* Kiểm thử chỉ một unit code tại một thời điểm

Kiến trúc của bạn phải hỗ trợ việc testing units (tức là, các class hoặc mọi nhóm rất nhỏ của các class) độc lập, chứ không phải là tất cả chúng kết lại với nhau. Nếu không, bạn sẽ có rất nhiều chồng chéo giữa các test, vì vậy việc thay đổi một unit ảnh hưởng ra bên ngoài và gây ra thất bại ở khắp mọi nơi.

Nếu bạn không thể làm được điều này, thì kiến trúc của bạn đang làm hạn chế chất lượng công việc của bạn - hãy xem xét sử dụng Inversion of Control.

* Giả lập tất cả những dịch vụ và trạng thái bên ngoài

Nếu không, hành vi trong những dịch vụ bên ngoài sẽ chồng chéo lên nhiều test, và trạng thái dữ liệu khác nhau trong mỗi unit test có thể ảnh hưởng đến kết quả của nhau.

Bạn chắc chắn phải nhận một kết quả sai nếu chạy các test của mình theo một thứ tự xác định, hoặc nếu chúng chỉ làm việc khi cơ sở dữ liệu hoặc kết nối mạng của bạn đang hoạt động.

(Bằng cách này, đôi khi kiến trúc của bạn có thể có nghĩa là code của bạn tương tác với các biến static trong quá trình unit test. Hãy tránh điều này nếu có thể, nhưng nếu bạn không thể, ít nhất là đảm bảo mỗi test sẽ reset các biến static liên quan đến một trạng thái được biết đến trước khi nó chạy.)

* Tránh những điều kiện tiên quyết không cần thiết

Tránh việc có các code setup chung chạy vào lúc bắt đầu của rất nhiều các test không liên quan. Nếu không, nó không rõ ràng về những giả định mà mỗi test dựa trên đó, và chỉ ra rằng bạn đang không test chỉ một đơn vị duy nhất.

Một ngoại lệ: Đôi khi tôi thấy nó hữu ích khi có một phương pháp thiết lập chung được chia sẻ bởi một số lượng rất nhỏ các unit test (một nhóm nhỏ) nhưng chỉ khi tất cả những test đó yêu cầu tất cả những điều kiện tiên quyết này. Nó liên quan đến một unit testing pattern là context-specification, nhưng vẫn có rủi ro là nó trở nên không thể bảo trì được nếu bạn cố gắng sử dụng lại cùng một setup code cho một số lượng lớn các test.

(Bằng cách này, tôi sẽ không tính đẩy nhiều điểm dữ liệu thông qua các test tương tự (ví dụ, bằng cách sử dụng các [TestCase] API của NUnit) là vi phạm quy tắc độc lập này. Quá trình test có thể hiển thị nhiều thất bại nếu một cái gì đó thay đổi, nhưng nó vẫn chỉ có một phương pháp test để duy trì, vì vậy điều đó cũng tốt.)

**Đừng sử dụng unit test trong phần thiết lập cấu hình**

Theo định nghĩa, các thiết lập cấu hình của bạn không phải là một phần của bất kỳ unit code nào (đó là lý do tại sao bạn trích xuất các thiết lập ra khỏi code của unit của bạn). Ngay cả khi bạn có thể viết một unit test để kiểm tra cấu hình của mình, nó chỉ đơn thuần buộc bạn phải xác định cấu hình tương tự tại một vị trí dự phòng bổ sung. Xin chúc mừng: nó chứng tỏ rằng bạn có thể copy và paste!

Cá nhân tôi coi việc sử dụng những thứ như các filter trong ASP.NET MVC như là cấu hình. Các filter như [Authorize] hoặc [RequiresSsl] là các cấu hình tùy chọn gắn vào trong code của bạn. Bằng mọi cách viết một kiểm thử tích hợp (integration test) cho các hành vi bên ngoài-quan sát được (externally-observable), nhưng nó là vô nghĩa để thử unit testing cho sự hiện diện của các thuộc tính của filter trong mã nguồn của bạn - một lần nữa nó chỉ chứng tỏ rằng bạn có thể copy và paste. Điều đó không giúp bạn thiết kế bất cứ điều gì, và nó sẽ chẳng bao giờ phát hiện bất kỳ lỗi nào cả.

**Đặt tên các unit test của bạn một cách rõ ràng và nhất quán**

Nếu bạn đang kiểm thử action Purchase của ProductController hành động như thế nào khi hết hàng trong kho (stock is zero), sau đó có thể có một lớp test fixture được gọi là PurchasingTests cùng với một unit test gọi là ProductPurchaseAction\_IfStockIsZero\_RendersOutOfStockView(). Cái tên này mô tả **đối tượng** (action Purchase của ProductController), **kịch bản** (hết hàng trong kho), và **kết quả** (hiển thị trên view là "hết hàng"). Tôi không biết liệu có một cái tên cho mô hình đặt tên này hay không, mặc dù tôi biết nhiều người khác cũng làm theo cách này.

Tránh sử dụng những tên unit test không rõ nghĩa như Purchase() hoặc OutOfStock(). Việc bảo trì sẽ rất khó khăn nếu bạn không biết là mình đang cố duy trì cái gì.