## Khái niệm về Unit Test

### Unit Test là gì?

**Unit Test** là một loại kiểm thử phần mềm trong đó các đơn vị hay thành phần riêng lẻ của phần mềm được kiểm thử. Unit Test được thực hiện trong quá trình phát triển ứng dụng. Mục tiêu của Unit Test là cô lập một phần code và xác minh tính chính xác của đơn vị đó.

### Unit là gì?

Một Unit là một thành phần PM nhỏ nhất mà ta có thể kiểm tra được như các hàm (Function), thủ tục (Procedure), lớp (Class), hoặc các phương thức (Method).

Vì Unit được chọn để kiểm tra thường có kích thước nhỏ và chức năng hoạt động đơn giản, chúng ta không khó khăn gì trong việc tổ chức, kiểm tra, ghi nhận và phân tích kết quả kiểm tra nên việc phát hiện lỗi sẽ dễ dàng xác định nguyên nhân và khắc phục cũng tương đối dễ dàng vì chỉ khoanh vùng trong một Unit đang kiểm tra.

Mỗi Unit Test sẽ gửi đi một thông điệp và kiểm tra câu trả lời nhận được đúng hay không, bao gồm:

* Các kết quả trả về mong muốn
* Các lỗi ngoại lệ mong muốn

Các đoạn code Unit Test hoạt động liên tục hoặc định kỳ để thăm dò và phát hiện các lỗi kỹ thuật trong suốt quá trình phát triển, do đó Unit Test còn được gọi là kỹ thuật kiểm nghiệm tự động. Unit Test có các đặc điểm sau:

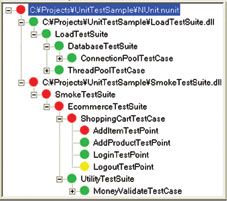
* Đóng vai trò như những người sử dụng đầu tiên của hệ thống.
* Chỉ có giá trị khi chúng có thể phát hiện các vấn đề tiềm ẩn hoặc lỗi kỹ thuật.

### Khi làm Unit test chúng ta thường thấy các khái niệm sau:

1. **Assertion**: Là một phát biểu mô tả các công việc kiểm tra cần tiến hành, thí dụ: AreEqual(), IsTrue(), IsNotNull()… Mỗi một Unit Test gồm nhiều assertion kiểm tra dữ liệu đầu ra, tính chính xác của các lỗi ngoại lệ ra và các vấn đề phức tạp khác như: – Sự tồn tại của một đối tượng – Điều kiện biên: Các giá trị có vượt ra ngoài giới hạn hay không – Thứ tự thực hiện của các luồng dữ liệu…
2. **Test Point**: Là một đơn vị kiểm tra nhỏ nhất, chỉ chứa đơn giản một assertion nhằm khẳng định tính đúng đắn của một chi tiết code nào đó. Mọi thành viên dự án đều có thể viết một test point. Test Case: Là một tập hợp các test point nhằm kiểm tra một đặc điểm chức năng cụ thể, thí dụ toàn bộ giai đoạn người dùng nhập dữ liệu cho đến khi thông tin được nhập vào cơ sở dữ liệu. Trong nhiều trường hợp kiểm tra đặc biệt và khẩn cấp có thể không cần đến test case.
3. **Test Suite**: Là một tập hợp các test case định nghĩa cho từng module hoặc hệ thống con.
4. **Regression Testing (hoặc Automated Testing)**: Là phương pháp kiểm nghiệm tự động sử dụng một phần mềm đặc biệt. Cùng một loại dữ liệu kiểm tra giống nhau nhưng được tiến hành nhiều lần lặp lại tự động nhằm ngăn chặn các lỗi cũ phát sinh trở lại. Kết hợp Regression Testing với Unit Testing sẽ đảm bảo các đoạn code mới vẫn đáp ứng yêu cầu thay đổi và các đoạn code cũ sẽ không bị ảnh hưởng bởi các hoạt động bảo trì.
5. **Production Code**: Phần code chính của ứng dụng được chuyển giao cho khách hàng.
6. **Unit Testing Code**: Phần code phụ để kiểm tra mã ứng dụng chính, không được chuyển giao cho khách hàng.
7. **Vòng đời Unit Test**

UT có 3 trạng thái cơ bản:

* Fail (trạng thái lỗi)
* Ignore (tạm ngừng thực hiện)
* Pass (trạng thái làm việc)
* Toàn bộ UNIT TEST được vận hành trong một hệ thống tách biệt. Có rất nhiều PM hỗ trợ thực thi UNIT TEST với giao diện trực quan. Thông thường, trạng thái của UNIT TEST được biểu hiện bằng các màu khác nhau: màu xanh (pass), màu vàng (ignore) và màu đỏ (fail)



UNIT TEST chỉ thực sự đem lại hiệu quả khi:

* Được vận hành lặp lại nhiều lần
* Tự động hoàn toàn
* Độc lập với các UNIT TEST khác.

1. **Thiết kế Unit test**

Mỗi UNIT TEST đều được tiết kế theo trình tự sau:

* Thiết lập các điều kiện cần thiết: khởi tạo các đối tượng, xác định tài nguyên cần thiết, xây dựng các dữ liệu giả…
* Triệu gọi các phương thức cần kiểm tra.
* Kiểm tra sự hoạt động đúng đắn của các phương thức.
* Dọn dẹp tài nguyên sau khi kết thúc kiểm tra.

1. **Ứng dụng Unit test**

* Kiểm tra mọi đơn vị nhỏ nhất là các thuộc tính, sự kiện, thủ tục và hàm.
* Kiểm tra các trạng thái và ràng buộc của đối tượng ở các mức sâu hơn mà thông thường chúng ta không thể truy cập được.
* Kiểm tra các quy trình (process) và mở rộng hơn là các khung làm việc(workflow – tập hợp của nhiều quy trình)

1. **Lợi ích của việc áp dụng Unit test**

Có nhiều lý do giải thích [vì sao bạn nên sử dụng unit test](https://comdy.vn/unit-test/unit-test-la-gi/#tai-sao-nen-su-dung-unit-test-2). Dưới đây là một trong những lý do đó.

Bởi vì bạn sẽ thường xuyên thay đổi code của mình. Và khi bạn làm vậy, bạn có thể gây ra một kết quả mà bạn không lường trước được.

Các bài kiểm tra unit test cung cấp cho các developer một loại thông tin chi tiết về một thay đổi gây ra một số kết quả không mong muốn. Khi kiểm tra không thành công, chúng tôi có thể kiểm tra xem có vấn đề nào sau đây không:

* Bản thân bài kiểm tra unit test đã sai, và cần được thay đổi.
* Kiểm tra dựa trên các phụ thuộc được phân tách không đúng cách.
* Code mới thay đổi bị sai và cần được sửa lại.
* Code được thay đổi gây ảnh hưởng không mong muốn tới các chức năng khác.

Và đó chỉ là một vài trong số những nguyên nhân có thể xảy ra. Nhưng nếu không có các bài kiểm tra unit test, chúng tôi sẽ không nhận được *bất kỳ* thông báo *nào* rằng đã xảy ra sự cố, chúng tôi sẽ tiếp tục đi trên một con đường xấu

Thời gian đầu, người ta thường do dự khi phải viết UNIT TEST thay vì tập trung vào code cho các chức năng nghiệp vụ. Công việc viết Unit Test có thể mất nhiều thời gian hơn code rất nhiều nhưng lại có lợi ích sau:

* Tạo ra môi trường lý tưởng để kiểm tra bất kỳ đoạn code nào, có khả năng thăm dò và phát hiện lỗi chính xác, duy trì sự ổn định của toàn bộ PM và giúp tiết kiệm thời gian so với công việc gỡ rối truyền thống.
* Phát hiện các thuật toán thực thi không hiệu quả, các thủ tục chạy vượt quá giới hạn thời gian.
* Phát hiện các vấn đề về thiết kế, xử lý hệ thống, thậm chí các mô hình thiết kế.
* Phát hiện các lỗi nghiêm trọng có thể xảy ra trong những tình huống rất hẹp.
* Tạo hàng rào an toàn cho các code block: Bất kỳ sự thay đổi nào cũng có thể tác động đến hàng rào này và thông báo những nguy hiểm tiềm tàng.

Trong môi trường làm việc Unit Test còn có tác dụng rất lớn đến năng suất làm việc:

* Giải phóng chuyên viên QA khỏi các công việc kiểm tra phức tạp.
* Tăng sự tự tin khi hoàn thành một công việc. Chúng ta thường có cảm giác không chắc chắn về các đoạn code của mình như liệu các lỗi có quay lại không, hoạt động của module hiện hành có bị tác động không, hoặc liệu công việc hiệu chỉnh mã có gây hư hỏng đâu đó…
* Là công cụ đánh giá năng lực của bạn. Số lượng các tình huống kiểm tra (test case) chuyển trạng thái “pass” sẽ thể hiện tốc độ làm việc, năng suất của bạn.

1. **Cách code hiệu quả với Unit Test**

Phân tích các tình huống có thể xảy ra đối với code. Đừng bỏ qua các tình huống tồi tệ nhất có thể xảy ra, thí dụ dữ liệu nhập làm một kết nối database thất bại, ứng dụng bị treo vì một phép toán chia cho không, các thủ tục đưa ra lỗi ngoại lệ sai có thể phá hỏng ứng dụng một cách bí ẩn…

Mọi UNIT TEST phải bắt đầu với trạng thái “fail” và chuyển trạng thái “pass” sau một số thay đổi hợp lý đối với mã chính.

Mỗi khi viết một đoạn code quan trọng, hãy viết các UNIT TEST tương ứng cho đến khi bạn không thể nghĩ thêm tình huống nào nữa.

Nhập một số lượng đủ lớn các giá trị đầu vào để phát hiện điểm yếu của mã theo nguyên tắc:

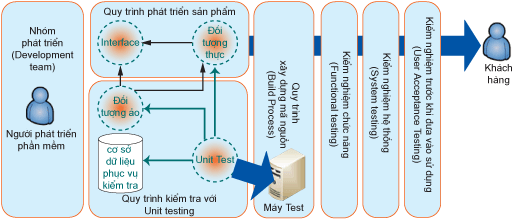
* Nếu nhập giá trị đầu vào hợp lệ thì kết quả trả về cũng phải hợp lệ
* Nếu nhập giá trị đầu vào không hợp lệ thì kết quả trả về phải không hợp lệ
* Sớm nhận biết các đoạn code không ổn định và có nguy cơ gây lỗi cao, viết UNIT TEST tương ứng để khống chế.

Ứng với mỗi đối tượng nghiệp vụ (business object) hoặc đối tượng truy cập dữ liệu (data access object), nên tạo ra một lớp kiểm tra riêng vì những lỗi nghiêm trọng có thể phát sinh từ các đối tượng này.

Để ngăn chặn các lỗi có thể phát sinh trở lại thực thi tự động tất cả UNIT TEST mỗi khi có một sự thay đổi quan trọng, hãy làm công việc này mỗi ngày. Các UNIT TEST lỗi cho chúng ta biết thay đổi nào là nguyên nhân gây lỗi.

Để tăng hiệu quả và giảm rủi ro khi viết các UNIT TEST, cần sử dụng nhiều phương thức kiểm tra khác nhau. Hãy viết càng đơn giản càng tốt.

Cuối cùng, viết UNIT TEST cũng đòi hỏi sự nỗ lực, kinh nghiệm và sự sáng tạo như viết PM.



Trước khi kết thúc phần này, tôi có một lời khuyên là viết UNIT TEST cũng tương tự như viết code một chương trình, điều bạn cần làm là không ngừng thực hành. Hãy nhớ UNIT TEST chỉ thực sự mang lại lợi ích nếu chúng ta đặt vấn đề chất lượng phần mềm lên hàng đầu hơn là chỉ nhằm kết thúc công việc đúng thời hạn.

1. **Có nên viết unit test cho mọi thứ không?**

Câu trả lời ngắn gọn là **KHÔNG**.

Không phải tất cả mọi thứ đều cần phải được kiểm tra bằng unit test. Các đoạn code có thể không bao giờ thay đổi, hoặc không quan trọng đối với chức năng chính của hệ thống, hoặc rất khó để viết unit test hoặc đơn giản là quá lãng phí khi viết unit test, ...

Ngoài ra, chính xác khi nào và tại sao một thứ gì đó cần được viết unit test. Không có một kế hoạch kiểm thử nào phù hợp với tất cả. Mọi hệ thống đều khác nhau, mọi nhu cầu đều khác nhau, và bạn phải đánh giá phạm vi kiểm thử cần thiết dựa trên thiết kế và độ phức tạp của hệ thống bạn đang xây dựng, chưa kể thời gian thực hiện các kiểm tra đó.

### Vậy chúng ta nên viết unit test cho cái gì?

Chúng ta nên viết unit test cho bất cứ điều gì là:

* Quan trọng đối với toàn bộ chức năng của ứng dụng.
* Đã biết hoặc có khả năng hỏng hóc cao.
* Có khả năng thay đổi trong tương lai.

Nói cách khác: bạn nên kiểm tra cosde nào quan trọng, dễ hỏng hoặc có khả năng thay đổi.

Nói một cách đơn giản và dễ hiểu nhất thì nên viết unit test cho:

* Các lớp thư viện, lớp hạ tầng dùng chung cho nhiều project.
* Các lớp tầng business logic.
* Các lớp controller của Web UI và Web API.

### Làm thế nào để chúng tôi viết unit test?

Tôi rất vui vì bạn đã hỏi điều đó. Chính xác thì chúng ta phải viết unit test như thế nào?

Có rất nhiều thư viện cho unit test có sẵn trong thế giới .NET như [MSTest](https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-mstest), [NUnit](https://nunit.org/), [XUnit](https://xunit.net/) và các thư viện khác.

Đây là những thứ phổ biến nhất và chúng hoạt động theo những cách tương tự nhau. Tuy nhiên, theo ý kiến ​​của tôi xUnit mang lại sự dễ sử dụng và khả năng tái sử dụng tốt nhất.

1. **Thông tin cơ bản về XUnit**

XUnit là một framework unit test cho .NET cung cấp một cách dễ dàng để viết code, chạy và debug các bài kiểm tra unit test.

Trong xUnit, một bài kiểm tra cơ bản là một phương thức trong lớp public, không có tham số hoặc giá trị trả về, được gắn nhãn bằng attribute Fact, như sau:

public class TestCases

{

    [Fact]

    public void ClassName\_MethodName\_ExpectedResult()

    {

        // Arrange

        // Fact

        // Assert

    }

}

Tại thời điểm này, bài kiểm tra trên chính xác là không có gì. Để điền vào những gì bài kiểm tra nên làm, chúng ta có thể làm theo phương pháp "Arrange, Act, Assert."

*LƯU Ý: Bạn có thể thiết lập xUnit để chạy các bài kiểm tra với đầu vào và đầu ra bằng Attribute [Theory].*

### Arrange, Act, Assert (AAA)

Cụm từ "Arrange, Act, Assert" hay AAA là một cách hay để ghi nhớ cấu trúc của một bài kiểm tra unit test.

* **Arrange** có nghĩa là để thiết lập môi trường thử nghiệm cần thiết và các phụ thuộc. Điều này có thể có nghĩa là tạo một tập hợp dữ liệu thử nghiệm tốt đã biết hoặc tạo mock của một phần phụ thuộc mà chúng ta không muốn kiểm tra. Tóm lại, "Arrange" có nghĩa là "tạo ra những gì bài kiểm tra cần để chạy."
* **Act** có nghĩa là thực thi mã cần được kiểm tra, đã được thiết lập trong bước Arrange.
* **Assert** có nghĩa là kiểm tra kết quả và đầu ra và xác nhận rằng chúng là những gì chúng ta mong đợi.

Ví dụ dưới đây là một bài kiểm tra unit test có sử dụng mock:

[Fact]

public void PlayerService\_GetAllPlayers\_InvalidLeague()

{

    //Arrange

    var mockLeagueRepo = new MockLeagueRepository().MockIsValid(false);

    var playerService = new PlayerService(new MockPlayerRepository().Object,

        new MockTeamRepository().Object,

        mockLeagueRepo.Object);

    //Act

    var allPlayers = playerService.GetForLeague(1);

    //Assert

    Assert.Empty(allPlayers);

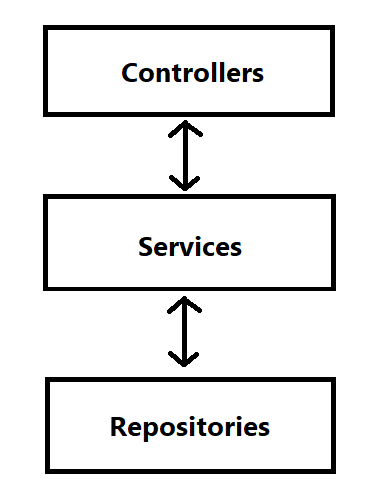
    mockLeagueRepo.VerifyIsValid(Times.Once());

}

Đừng lo lắng về đoạn mã này; bây giờ, tất cả những gì bạn cần nhớ là mô hình "Arrange, Act, Assert".

### Mock

Đây là kiến ​​trúc cơ bản của ứng dụng:



Ở tầng thấp nhất, chúng ta có **Repositories**. Các lớp này giao tiếp trực tiếp với cơ sở dữ liệu.

Ở tầng tiếp theo là **Services**. Các lớp này xử lý các nghiệp vụ (business logic) của ứng dụng. Nó gọi các Repositories để lấy dữ liệu từ database hoặc lưu trữ dữ liệu vào database.

Ở tầng cao nhất là **Controller**. Đây là các lớp Controller của MVC, nó gọi các Services để lấy dữ liệu, xử lý nghiệp vụ, ..., cũng như xử lý view và những thứ khác mà controller thường làm.

Ví dụ mô hình dữ liệu của dự án như sau:

* **Leagues** là giải đấu thể thao, bao gồm nhiều team.
* **Teams** là các đội chơi bao gồm nhiều player.
* **Players** là những người chơi cho một team cụ thể.

#### Kiến thức cơ bản về Mocking

Một trong những mục tiêu mà bộ thử nghiệm nên có để kiểm tra một ứng dụng như ứng dụng này là kiểm tra từng lớp độc lập với những lớp khác. Điều đó có nghĩa là để kiểm tra lớp Service, chúng ta cần chạy các bài kiểm tra độc lập với các chức năng trong lớp Repository, nơi mà lớp Service phụ thuộc vào.

Để làm điều này, chúng ta sẽ "mocking" lớp Repository, có nghĩa là chúng ta thiết lập một lớp "giả" trả về một giá trị đã biết cho các lệnh gọi nhất định. Chúng ta thực hiện điều này đối với mỗi lệnh gọi mà lớp Service sẽ thực hiện đối với lớp Repository, hoặc tổng quát hơn là đối với từng phụ thuộc mà lớp đang được kiểm tra có.

Thư viện phổ biến nhất để thực hiện mocking trong ASP.NET là [Moq](https://github.com/Moq/moq4/wiki/Quickstart).

#### Sử dụng Moq

Moq là [NuGet package](https://www.nuget.org/packages/Moq/), vì vậy trước khi có thể sử dụng nó, chúng ta cần thêm nó vào dự án của mình thông qua NuGet.

Cách đầu tiên chúng tôi sử dụng Moq là thiết lập một thể hiện "giả" của lớp, như sau:

var mockTeamRepository = new Mock<ITeamRepository>();

Đối tượng mockTeamRepository được tạo sau đó có thể được đưa vào các lớp cần nó, như sau:

var teamService = new TeamService(mockTeamRepository.Object);

Tuy nhiên, cho đến lúc này, lớp mock của chúng ta không thực sự làm bất cứ điều gì; chúng ta cần thiết lập cho phương thức được gọi hoặc trả về giá trị. Để làm được điều đó, chúng ta có thể làm như sau:

mockTeamRepository.Setup(x => x.GetByID(It.IsAny<int>()))

                  .Returns(new Team());

Giải thích đoạn mã trên:

1. Phương thức .Setup() được sử dụng để chỉ định phương thức cần mocking (trong trường hợp này là GetByID()).
2. Phương thức GetByID() có một tham số kiểu int. Đoạn mã It.IsAny<int>() chỉ định rằng phương thức GetByID() sẽ được gọi với bất kỳ tham số kiểu int nào. Các ví dụ khác có trên trang [Moq GitHub](https://github.com/Moq/moq4/wiki/Quickstart#matching-arguments).
3. Phương thức .Returns() được sử dụng để chỉ định dữ liệu trả về cho phương thức GetByID(). Trong ví dụ này sẽ trả về new Team().

Vì vậy, với bất kỳ số nguyên nào, lớp được mock sẽ trả về new Team().

Đôi khi, bạn có thể muốn có một phương thức ném một ngoại lệ thay vì trả về một đối tượng. Bạn có thể thực hiện việc này bằng phương thức Throws() như sau:

mockTeamRepository.Setup(x => x.GetByID(It.IsAny<int>()))

                  .Throws(new Exception());

#### Xây dựng một Fluent Mock

Tất cả những điều này đều ổn và tuyệt vời, bởi vì nó cho phép chúng ta dễ dàng mô phỏng các lớp của mình để chúng ta có thể kiểm tra chúng một cách độc lập.

Nhưng trong các bài kiểm tra unit test thực tế, chúng ta thường phải sử dụng các mô phỏng đó trong nhiều thử nghiệm.

Một trong những cách mà nhóm của tôi và tôi đã khám phá ra để hạn chế số lượng mã lặp lại trong các bài kiểm tra của chúng tôi là xây dựng một lớp "fluent mock".

Hãy bắt đầu với một lớp kế thừa từ Mock<>:

public class MockTeamRepository : Mock<ITeamRepository>

{

}

Đối với mỗi phương thức chúng ta muốn giả lập, chúng ta cần gọi phương thức .Setup(), nhưng chúng ta muốn làm như vậy theo cách cho phép chúng ta gọi "chuỗi" các phương thức khi tạo các thể hiện của các lớp mock này. Điều này cho phép một tập hợp các lệnh gọi "fluent".

public class MockTeamRepository : Mock<ITeamRepository>

{

    public MockTeamRepository MockGetByID(Team result)

    {

        Setup(x => x.GetByID(It.IsAny<int>()))

        .Returns(result);

        return this;

    }

    public MockTeamRepository MockGetForLeague(List<Team> results)

    {

        Setup(x => x.GetForLeague(It.IsAny<int>()))

        .Returns(results);

        return this;

    }

}

Sau đó, chúng ta có thể sử dụng lớp "fluent mock" mới này để tạo một đối tượng giả lập như sau:

var mockTeamRepo = new MockTeamRepository()

                       .MockGetByID(new Team())

                       .MockGetForLeague(new List<Team>());

#### Xác minh phương thức được gọi

Moq cũng bao gồm một tính năng "Verify" cho phép chúng ta đảm bảo rằng một phương thức mock được gọi với số lần xác định.

Ví dụ: bạn có thể muốn sử dụng tính năng này khi bạn cần thực hiện một số loại xác minh trước khi gọi đến một lớp mock. Hãy xem một ví dụ về phương thức Search() của lớp TeamRepository.

public List<Team> Search(TeamSearch search)

{

    //If we are searching for an invalid or unknown League...

    var isValidLeague = \_leagueRepo.IsValid(search.LeagueID);

    if (!isValidLeague)

    {

        return new List<Team>(); //Return an empty list.

    }

    //Otherwise get all teams in the specified league...

    var allTeams = \_teamRepo.GetForLeague(search.LeagueID);

    //... and filter them by the specified Founding Date and Direction.

    if (search.Direction == Enums.SearchDateDirection.OlderThan)

    {

        return allTeams.Where(x => x.FoundingDate <= search.FoundingDate).ToList();

    }

    return allTeams.Where(x => x.FoundingDate >= search.FoundingDate).ToList();

}

Trong phương pháp này, có thể phương thức GetForLeague() của lớp TeamRepository hoàn toàn không được gọi; điều này xảy ra khi LeagueID không hợp lệ.

Phương thức Verify sẽ cho phép chúng ta đảm bảo trong quá trình thử nghiệm của mình rằng, khi LeagueID giải đấu không hợp lệ, phương thức GetForLeague() sẽ không bao giờ được gọi.

Sau đó, chúng ta có thể thêm một hàm mới vào lớp MockTeamRepository, như sau:

public class MockTeamRepository : Mock<ITeamRepository>

{

    //...

    public MockTeamRepository VerifyGetForLeague(Times times)

    {

        Verify(x => x.GetForLeague(It.IsAny<int>()), times);

        return this;

    }

}

Lớp Times được định nghĩa bởi Moq để sử dụng trong những tình huống này; nó cho phép chúng ta chỉ định số lần chúng ta mong đợi phương thức được gọi.

Ví dụ: nếu chúng ta mong đợi nó được gọi chính xác một lần, chúng ta có thể sử dụng phương thức này như sau:

//Arrange

var mockTeamRepo = new MockTeamRepository()

                       .MockGetByID(new Team())

                       .MockGetForLeague(new List<Team>());

var teamSearch = new TeamSearch()

{

    LeagueID = 1

}

var teamService = new TeamService(mockTeamRepo.Object);

//Act

var result = teamService.Search(teamSearch);

//Assert

mockTeamRepo.VerifyGetForLeague(Times.Once());//Expect this to be called exactly once.

Có nhiều giá trị cho lớp Times mà chúng ta có thể sử dụng:

mockTeamRepo.VerifyGetForLeague(Times.Once());

mockTeamRepo.VerifyGetForLeague(Times.AtLeastOnce());

mockTeamRepo.VerifyGetForLeague(Times.AtMostOnce());

mockTeamRepo.VerifyGetForLeague(Times.Never());

mockTeamRepo.VerifyGetForLeague(Times.AtMost(3)); //At most three calls

mockTeamRepo.VerifyGetForLeague(Times.Exactly(5)); //Exactly five calls

#### Xây dựng lớp Mock

Chúng ta đã tìm hiểu cách xây dựng một lớp fluent mock, bây giờ hãy thực hành xây dựng chúng.

Hãy bắt đầu với lớp TeamRepository. Đây là mã của nó:

public class TeamRepository : ITeamRepository

{

    public Team GetByID(int id)

    {

        throw new NotImplementedException();

    }

    public List<Team> GetForLeague(int leagueID)

    {

        throw new NotImplementedException();

    }

}

Chờ một chút, tại sao không có gì được triển khai trong lớp này?! Để chứng minh một luận điểm: **Các chi tiết triển khai thực tế không quan trọng khi mocking một lớp**. Vì vậy, hãy viết một số quy tắc cho các phương thức của lớp này.

* Phương thức GetByID() nên trả về một đối tượng Team nếu một đối tượng được tìm thấy và ném một ngoại lệ nếu không tìm thấy.
* Phương thức GetForLeague() sẽ trả về một danh sách nếu Id giải đấu hợp lệ và được tìm thấy, và một danh sách trống nếu không tìm thấy.

Với các quy tắc này, chúng ta có thể tạo ra lớp giả lập. Nó như sau:

public class MockTeamRepository : Mock<ITeamRepository>

{

    public MockTeamRepository MockGetByID(Team result)

    {

        Setup(x => x.GetByID(It.IsAny<int>()))

            .Returns(result);

        return this;

    }

    public MockTeamRepository MockGetByIDInvalid()

    {

        Setup(x => x.GetByID(It.IsAny<int>()))

            .Throws(new Exception());

        return this;

    }

    public MockTeamRepository VerifyGetByID(Times times)

    {

        Verify(x => x.GetByID(It.IsAny<int>()), times);

        return this;

    }

    public MockTeamRepository MockGetForLeague(List<Team> results)

    {

        Setup(x => x.GetForLeague(It.IsAny<int>()))

            .Returns(results);

        return this;

    }

    public MockTeamRepository VerifyGetForLeague(Times times)

    {

        Verify(x => x.GetForLeague(It.IsAny<int>()), times);

        return this;

    }

}

Lớp fluent mock này và những lớp khác giống như nó có thể được sử dụng trong các Unit Test của chúng ta như sau:

[Fact]

public void TeamService\_Search\_OlderThan\_Valid()

{

    //Arrange

    var mockTeams = GetMockTeams();

    var mockTeamRepo = new MockTeamRepository().MockGetForLeague(mockTeams);

    var mockLeagueRepo = new MockLeagueRepository().MockIsValid(true);

    var teamService = new TeamService(mockTeamRepo.Object, mockLeagueRepo.Object);

    var searchParams = new TeamSearch()

    {

        LeagueID = 1,

        FoundingDate = new DateTime(2013, 1, 1),

        Direction = SearchDateDirection.OlderThan

    };

    //Act

    var results = teamService.Search(searchParams);

    //Assert

    Assert.NotEmpty(results);

    mockLeagueRepo.VerifyIsValid(Times.Once());

    mockTeamRepo.VerifyGetForLeague(Times.Once());

}

Tương tự là lớp fluent mock cho lớp PlayerRepository và LeagueRepository:

public class MockPlayerRepository : Mock<IPlayerRepository>

{

    public MockPlayerRepository MockGetByID(Player result)

    {

        Setup(x => x.GetByID(It.IsAny<int>()))

            .Returns(result);

        return this;

    }

    public MockPlayerRepository MockGetForTeam(List<Player> results)

    {

        Setup(x => x.GetForTeam(It.IsAny<int>()))

            .Returns(results);

        return this;

    }

    public MockPlayerRepository VerifyGetForTeam(Times times)

    {

        Verify(x => x.GetForTeam(It.IsAny<int>()), times);

        return this;

    }

}

public class MockLeagueRepository : Mock<ILeagueRepository>

{

    public MockLeagueRepository MockIsValid(bool result)

    {

        Setup(x => x.IsValid(It.IsAny<int>()))

            .Returns(result);

        return this;

    }

    public MockLeagueRepository VerifyIsValid(Times times)

    {

        Verify(x => x.IsValid(It.IsAny<int>()), times);

        return this;

    }

}

### Theory trong XUnit

Ta có class Test sau:

using System;

using Xunit;

using Validators.Password;

namespace PasswordValidatorTests

{

    public class ValidityTest

    {

        [Fact]

        public void ValidPassword()

        {

            //Arrange

            var passwordValidator = new PasswordValidator();

            const string password = "Th1sIsapassword!";

            //Act

            bool isValid = passwordValidator.IsValid(password);

            //Assert

            Assert.True(isValid, $"The password {password} is not valid");

        }

        [Fact]

        public void NotValidPassword()

        {

            //Arrange

            var passwordValidator = new PasswordValidator();

            const string password = "thisIsaPassword";

            //Act

            bool isValid = passwordValidator.IsValid(password);

            //Assert

            Assert.False(isValid, $"The password {password} should not be valid!");

        }

    }

}

Hai trường hợp về tính hợp lệ của mật khẩu được kiểm tra bởi các bài kiểm tra unit test ở trên là không đầy đủ.

Chúng chỉ là hai ví dụ đơn giản về các trường hợp thành công và thất bại, nhưng tất nhiên, các trường hợp có thể để kiểm tra còn nhiều hơn thế nữa.

Bạn không thể mong đợi có thể kiểm tra hết mọi trường hợp có thể xảy ra, nhưng bạn có thể kiểm tra một tập hợp con đáng kể các trường hợp điển hình.

Điều này giúp bạn có phạm vi mã được kiểm tra lớn hơn cho code của bạn. Trong ví dụ xác thực mật khẩu, điều này có nghĩa là bạn nên xác định một tập mật khẩu hợp lệ và không hợp lệ đại diện. Đối với mỗi mật khẩu trong các tập này, bạn nên áp dụng một trong các bài kiểm tra được triển khai ở trên.

Cách tiếp cận này sẽ đảm bảo sự tin cậy đáng kể vào hoạt động chính xác của phương thức IsValid(). Nhưng nó yêu cầu phải sao chép cùng một mã cho mỗi mật khẩu mẫu để kiểm tra. Bạn biết rằng [sao chép mã không phải là một best practice](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself).

May mắn thay, xUnit có thể giúp bạn vấn đề này bằng cách sử dụng attribute **Theory**. Theory là một bài kiểm tra unit test có tham số cho phép bạn thực hiện một tập hợp các bài kiểm tra đơn vị có cùng cấu trúc.

Theory cho phép bạn triển khai cái được gọi là [kiểm thử theo hướng dữ liệu](https://en.wikipedia.org/wiki/Data-driven_testing), là một phương pháp kiểm thử dựa nhiều vào sự biến đổi dữ liệu đầu vào.

Vì vậy, để dễ hình dung Theory là gì, hãy thay thế nội dung của đoạn code lúc đầu bằng nội dung sau:

using System;

using Xunit;

using Validators.Password;

namespace PasswordValidatorTests

{

    public class ValidityTest

    {

        [Theory]

        [InlineData("Th1sIsapassword!")]

        [InlineData("thisIsapassword123!")]

        [InlineData("Abc$123456")]

        public void ValidPassword(string password)

        {

            //Arrange

            var passwordValidator = new PasswordValidator();

            //Act

            bool isValid = passwordValidator.IsValid(password);

            //Assert

            Assert.True(expectedResult);

        }

        [Theory]

        [InlineData("Th1s!")]

        [InlineData("thisIsAPassword")]

        [InlineData("thisisapassword#")]

        [InlineData("THISISAPASSWORD123!")]

        [InlineData("")]

        public void NotValidPassword(string password)

        {

            //Arrange

            var passwordValidator = new PasswordValidator();

            //Act

            bool isValid = passwordValidator.IsValid(password);

            //Assert

            Assert.False(expectedResult);

        }

    }

}

Đoạn mã trên thay thế attribute Fact bằng attribute Theory, bổ sung thêm các attribute InlineData và tham số password vào 2 bài kiểm tra unit test.

Như bạn có thể thấy, thay vì hard code thông tin mật khẩu thì ở ví dụ trên, chúng ta tạo tham số password để lấy thông tin mật khẩu từ attribute InlineData. Mỗi attribute InlineData sẽ đại diện cho một mật khẩu trong tập mật khẩu chúng ta cần kiểm tra.

Nói cách khác, mỗi attribute InlineData đại diện cho một lệnh gọi của bài kiểm tra. Trên thực tế, nếu bạn khởi chạy lệnh, bạn sẽ nhận được thông báo rằng tất cả 3 bài kiểm tra ValidPassword và 5 bài kiểm tra NotValidPassword đều thành công.

Bên cạnh attribute InlineData, xUnit cung cấp cho bạn các cách khác để định nghĩa dữ liệu cho Theory, chẳng hạn như nguồn dữ liệu ClassData là một lớp triển khai interface IEnumerable và nguồn dữ liệu MemberData là một thuộc tính hoặc một phương thức.