# Unit testing C#

### Unit test là gì?

Trong quy trình phát triển phần mềm, Kiểm thử Đơn vị (Unit Test) về cơ bản kiểm tra các phần riêng lẻ (còn được gọi là Đơn vị) của mã nguồn (thường là các phương thức) và đảm bảo chúng động của chúng đúng với dự định của lập trình viên. Một Kiểm thử Đơn vị là một đoạn mã được viết bởi bất kỳ lập trình viên nào nhằm kiểm tra các phần chức năng nhỏ của chương trình lớn. Việc thực hiện kiểm thử đơn vị luôn được thiết kế để đơn giản hóa.

Một "ĐƠN VỊ" trong ngữ cảnh này là thành phần nhỏ nhất trong một đoạn mã lớn mà việc kiểm tra mang lại ý nghĩa, thường là một phương thức nằm trong một lớp nào đó. Nói chung, các trường hợp kiểm tra được viết dưới dạng hàm để đánh giá và xác định xem giá trị trả về sau khi thực hiện Kiểm thử Đơn vị có bằng giá trị mà bạn kỳ vọng khi viết hàm không. Mục tiêu chính của kiểm thử đơn vị là cô lập một phần đơn vị của mã và xác thực độ chính xác và tin cậy của nó.

## Tại sao phải cần Unit Test?

### Giảm thời gian thực hiện các kiểm tra chức năng

* Các kiểm tra chức năng rất tốn kém. Chúng thường liên quan đến việc mở ứng dụng và thực hiện một loạt các bước mà bạn (hoặc người khác) phải tuân theo để xác nhận hoạt động đúng như mong đợi. Những bước này có thể không phải lúc nào cũng được người kiểm tra biết đến. Họ sẽ phải liên hệ với một người am hiểu hơn trong lĩnh vực đó để thực hiện kiểm tra. Bản thân việc kiểm tra có thể mất vài giây đối với những thay đổi nhỏ, hoặc vài phút đối với các thay đổi lớn hơn. Cuối cùng, quy trình này phải được lặp lại cho mọi thay đổi mà bạn thực hiện trong hệ thống.
* Mặt khác, Unit Test (các kiểm tra đơn vị) chỉ mất vài mili giây, có thể được chạy chỉ bằng một nút nhấn và không nhất thiết đòi hỏi bất kỳ kiến thức nào về hệ thống nói chung. Việc kiểm tra có vượt qua hay thất bại phụ thuộc vào trình chạy thử nghiệm (test runner), không phải người thực hiện.

### Chống lại các lỗi hồi quy (regression)

* Lỗi hồi quy là những lỗi được đưa vào khi có thay đổi được thực hiện trong ứng dụng. Người kiểm thử thường không chỉ kiểm tra tính năng mới của họ mà còn kiểm tra cả những tính năng đã tồn tại trước đó để xác minh rằng các tính năng được triển khai trước đó vẫn hoạt động như mong đợi.
* Với Unit Test, có thể chạy lại toàn bộ bộ kiểm tra của bạn sau mỗi lần xây dựng (build) hoặc thậm chí sau khi bạn thay đổi một dòng mã. Điều này mang lại cho bạn sự tự tin để biết rằng mã mới của mình không làm hỏng chức năng hiện có.

### Tài liệu hướng dẫn có thể thực thi

* Có thể không phải lúc nào cũng hiển nhiên việc một phương thức cụ thể làm gì hoặc nó xử lý ra sao với một đầu vào nhất định. Bạn có thể tự hỏi: Phương thức này hoạt động như thế nào nếu tôi truyền cho nó một chuỗi trống? Null?
* Khi bạn có một bộ unit test được đặt tên tốt, mỗi kiểm tra phải có khả năng giải thích rõ ràng đầu ra mong đợi cho một đầu vào nhất định. Ngoài ra, nó phải có khả năng xác minh rằng nó thực sự hoạt động.

### Mã nguồn ít phụ thuộc (less coupled)

* Khi mã được ghép nối chặt chẽ, có thể khó thực hiện unit test. Nếu không tạo unit test cho mã bạn đang viết, sự ghép nối có thể ít rõ ràng hơn.
* Viết các kiểm thử cho mã của bạn sẽ tự nhiên tách rời mã nguồn, bởi nó sẽ khó kiểm tra hơn theo cách khác.

## Đặc điểm của một unit test tốt

* **Nhanh:** Không hiếm khi các dự án lớn có hàng nghìn unit test. Các unit test phải mất ít thời gian để chạy, chỉ khoảng vài mili giây.
* **Cô lập:** Các unit test độc lập, có thể được chạy một cách biệt lập và không có sự phụ thuộc vào bất kỳ yếu tố bên ngoài nào như hệ thống tệp hoặc cơ sở dữ liệu.
* **Có thể lặp lại:** Chạy một unit test sẽ có kết quả ổn định, nghĩa là, nó luôn trả về cùng một kết quả nếu bạn không thay đổi bất cứ điều gì giữa các lần chạy.
* **Tự kiểm tra:** Kiểm tra phải có khả năng tự động phát hiện nó đã vượt qua hay thất bại mà không cần bất kỳ sự tương tác nào của con người.
* **Kịp thời:** Một unit test không nên mất một khoảng thời gian quá mức lâu để viết so với mã đang được kiểm tra. Nếuếu bạn thấy việc test mã mất quá nhiều thời gian so với việc viết mã, hãy cân nhắc thiết kế lại theo cách dễ test hơn.

## Độ bao phủ mã (Code coverage)

* Tỷ lệ phần trăm độ bao phủ mã cao thường được liên kết với chât lượng mã nguồn cao hơn. Tuy nhiên, bản thân phép đo lường này không thể xác định chất lượng mã nguồn. Việc đặt một mục tiêu tỷ lệ phần trăm độ bao phủ mã quá tham vọng có thể phản tác dụng. Hãy tưởng tượng một dự án phức tạp với hàng nghìn nhánh điều kiện và bạn đặt mục tiêu 95% độ bao phủ mã. Hiện tại dự án đang duy trì 90% độ bao phủ mã. Khoảng thời gian cần để tính đến tất cả các trường hợp phụ (edge case) trong 5% còn lại có thể là một nhiệm vụ to lớn.
* Tỷ lệ phần trăm độ bao phủ mã không phải là một thước đo mã nguồn, và nó cũng không hàm ý chât lượng của mã nguồn cao. Nó chỉ đại diện cho lượng mã nguồn được phủ bởi các unit test.

### Các công cụ hỗ trợ Unit Test:

Trong bài báo cáo này, chúng ta sẽ sử dụng ngôn ngữ lập trình C# (.NET 8.0), có thể lựa chọn các Test Framework như NUnit, XUnit, MSTest. Hoặc có thể lựa chọn sử dụng “Microsoft.Testing.Platform”, một mã nguồn mỡ có thể được tích hợp trực tiếp vào dự án kiểm thử của bạn, và không có ứng dụng phụ thuộc nào, như vstest.console hay dotnet.test để chạy ca kiểm thử.

### Các vấn đề cần hiểu rõ:

Thuật ngữ "mock" (mô phỏng) thường bị hiểu sai trong lĩnh vực kiểm thử đơn vị (unit testing). Các định nghĩa sau đây sẽ giúp phân biệt các dạng "fake" (hư cấu) thông dụng nhất khi viết unit test:

* **Fake (Hư cấu):** Fake là thuật ngữ bao quát để chỉ các đối tượng giả mạo, bao gồm cả stub và mock object. Xác định chính xác nó là stub hay mock phụ thuộc vào ngữ cảnh sử dụng. Nói cách khác, một fake có thể đóng vai trò như stub hoặc mock.
* **Mock object (Đối tượng mô phỏng):** Mock object là một đối tượng giả mạo trong hệ thống, có trách nhiệm đánh giá kết quả của unit test (thành công hay thất bại). Ban đầu, mock object hoạt động như một Fake cho đến khi các hành vi của nó được kiểm chứng (assertion) dựa trên các điều kiện nhất định.
* **Stub (Cùi):** Stub là sự thay thế được kiểm soát cho một phụ thuộc (dependency) hoặc đối tác cộng tác (collaborator) có sẵn trong hệ thống. Bằng cách sử dụng stub, bạn có thể kiểm thử mã nguồn của mình mà không cần tương tác trực tiếp với phần phụ thuộc đó. Theo mặc định, một stub cũng bắt đầu hoạt động như một Fake.

Việc làm rõ khái niệm trên là rất quan trọng. Nếu bạn gọi stub là "mock" thì các lập trình viên khác có thể hiểu sai ý định của bạn.

Điểm then chốt cần lưu ý để phân biệt mock và stub là mock gần giống stub, tuy nhiên bạn sẽ đánh giá (assert) mock object, còn với stub thì không.

Bạn có thể tìm hiểu them: https://viblo.asia/p/tim-hieu-ve-stub-mock-va-fake-trong-unit-test-mPjxMe1bM4YL

### Quy ước chung:

### Tránh các phụ thuộc cơ sở hạ tầng

Cố gắng không đưa vào các phụ thuộc vào cơ sở hạ tầng khi viết unit tests. Những phụ thuộc này sẽ khiến cho tests chạy chậm và dễ lỗi, tốt nhất chỉ nên dành chúng cho integration tests (kiểm thử tích hợp). Bạn có thể né tránh các phụ thuộc này trong ứng dụng của mình bằng cách tuân theo Nguyên tắc phụ thuộc rõ ràng (Explicit Dependencies Principle) và sử dụng Dependency Injection (lồng ghép phụ thuộc). Bạn cũng có thể để các unit test trong một dự án tách biệt với các integration tests. Cách tiếp cận này sẽ đảm bảo project unit test của bạn không có các tham chiếu hoặc phụ thuộc vào các gói (packages) cơ sở hạ tầng.

Ví dụ:

public class OrderService {

private SqlConnection dbConnection;

public OrderService() {

this.dbConnection = new SqlConnection("Server=localhost;Database=orderdb;Trusted\_Connection=True;"); // Phụ thuộc trực tiếp vào cơ sở dữ liệu

}

}

Cách tiếp cận để tránh phụ thuộc trên:

public class OrderService {

//Interface để giao tiếp với db

private readonly IDbConnection \_DbConnection;

public OrderService(IDbConnection DbConnection) { \_DbConnection = DbConnection;

}

}

### Quy tắc đặt tên:

Tên của một kiểm thử mức đơn vị nên bao gồm ba phần:

* Tên của phương thức được kiểm tra.
* Tình huống mà phương thức được kiểm tra.
* Kết quả mong muốn khi tình huống được áp dụng.

Ví dụ:

[Fact]

public void Add\_SingleNumber\_ReturnsSameNumber()

{

var stringCalculator = new StringCalculator();

var actual = stringCalculator.Add("0");

Assert.Equal(0, actual);

}

### Quy tắc chuẩn bị một ca kiểm thử đơn vị:

Arrange, Act, Assert là một pattern (mẫu thiết kế) phổ biến trong kiểm thử đơn vị (unit testing). Đúng như tên gọi, nó bao gồm ba bước chính:

* **Arrange (Chuẩn bị):** Khởi tạo các đối tượng cần thiết cho việc kiểm thử và thiết lập các giá trị cho chúng.
* **Act (Thực thi):** Gọi phương thức cần được kiểm thử.
* **Assert (Kiểm định):** Đảm bảo rằng kết quả thực thi đáp ứng đúng với những gì mong đợi.

Ví dụ:

[Fact]

public void Add\_EmptyString\_ReturnsZero()

{

// Arrange

var stringCalculator = new StringCalculator();

// Act

var actual = stringCalculator.Add("");

// Assert

Assert.Equal(0, actual);

}

### Giá trị sử dụng để kiểm thử:

Đầu vào sử dụng trong một ca kiểm thử đơn vị nên đơn giản nhất có thể để dễ dàng xác nhận tính chính xác của hàm đang kiểm tra.

Ví dụ:

[Fact]

public void Add\_SingleNumber\_ReturnsSameNumber()

{

var stringCalculator = new StringCalculator();

var actual = stringCalculator.Add("42");// nên sử dụng 0

Assert.Equal(42, actual);

}

### Tránh dung chuỗi ảo: Việc đặt tên biến trong các ca kiểm thử đơn vị rất quan trọng, thậm chí có thể quan trọng hơn đặt tên biến trong mã nguồn sản phẩm thật. Các ca kiểm thử đơn vị không nên chứa các chuỗi ảo. Ví dụ:

[Fact]

public void Add\_BigNumber\_ThrowsException()

{

var stringCalculator = new StringCalculator();

// Action actual = () => stringCalculator.Add("1001"); không nên

const string MAXIMUM\_RESULT = "1001";

Action actual = () => stringCalculator.Add(MAXIMUM\_RESULT);

Assert.Throws<OverflowException>(actual);

}

* + 1. **Tránh việc thực thi logic trong kiểm thử:**  
       Ví dụ:

// Không nên  
[Fact]

public void Add\_MultipleNumbers\_ReturnsCorrectResults()

{

var stringCalculator = new StringCalculator();

var expected = 0;

var testCases = new[]

{

"0,0,0",

"0,1,2",

"1,2,3"

};

foreach (var test in testCases)// xử lý logic

{

Assert.Equal(expected, stringCalculator.Add(test));

expected += 3;

}

}

// Nên

[Theory]

[InlineData("0,0,0", 0)]

[InlineData("0,1,2", 3)]

[InlineData("1,2,3", 6)]

public void Add\_MultipleNumbers\_ReturnsSumOfNumbers(string input, int expected)

{

var stringCalculator = new StringCalculator();

var actual = stringCalculator.Add(input);

Assert.Equal(expected, actual);

}

* + 1. **Tránh thực thi quá nhiều “Acts”:**

Khi viết các bài kiểm thử, hãy cố gắng chỉ sử dụng một hành động (act) cho mỗi bài test. Dưới đây là các cách tiếp cận phổ biến:

* **Tạo một bài test riêng cho mỗi hành động (act).**
* **Sử dụng các bài test được tham số hóa (parameterized tests).**

Mục đích:

* **Khi bài test thất bại, bạn sẽ biết rõ ràng hành động nào gặp lỗi.** Điều này giúp chẩn đoán và sửa lỗi nhanh chóng.
* **Đảm bảo bài test tập trung vào một trường hợp duy nhất.** Giúp bài test đơn giản và dễ hiểu hơn.
* **Cung cấp cái nhìn toàn cảnh hơn về nguyên nhân khiến bài test thất bại.** Bạn có thể nhanh chóng phân tích vấn đề và điều chỉnh cho phù hợp.

Ví dụ:

// Không nên

[Fact]

public void Add\_EmptyEntries\_ShouldBeTreatedAsZero()

{

// Act

var actual1 = stringCalculator.Add("");

var actual2 = stringCalculator.Add(",");

// Assert

Assert.Equal(0, actual1);

Assert.Equal(0, actual2);

}

// Nên

[Theory]

[InlineData("", 0)]

[InlineData(",", 0)]

public void Add\_EmptyEntries\_ShouldBeTreatedAsZero(string input, int expected)

{

// Arrange

var stringCalculator = new StringCalculator();

// Act

var actual = stringCalculator.Add(input);

// Assert

Assert.Equal(expected, actual);

}

* 1. **Tổng kết:**

Báo cáo trên được dịch lại từ bài viết gốc nói về Unit Test C# của Microsoft, link bài viết: <https://learn.microsoft.com/vi-vn/dotnet/core/testing/>

Tham khảo từ các bài viết khác như:

<https://viblo.asia/p/unit-test-la-gi-loi-ich-va-nhung-luu-y-khi-viet-unit-test-YWOZr20NZQ0>

<https://en.wikipedia.org/wiki/Unit_testing>

<https://viblo.asia/p/tim-hieu-ve-stub-mock-va-fake-trong-unit-test-mPjxMe1bM4YL>