**VIẾT PHẦN MỀM CHẤT LƯỢNG**

Khi một hệ thống tiến hóa, sự phức tạp của nó tăng lên trừ khi công việc được thực hiện để duy trì hoặc giảm bớt.

-Prof. Manny Lehman

Không ai thích nợ, nó có thể khó nghe, nhưng một số mức nợ trong cuộc sống là không thể tránh khỏi. Nợ kỹ thuật trong phần mềm không phải là ngoại lệ. Tóm lại, nợ kỹ thuật là một loại thế chấp được đưa ra trên mã của bạn. Bạn đưa ra một khoản thế chấp mỗi khi bạn làm nhanh và sai trong mã của bạn. Giống như thế chấp trong cuộc sống, nợ kỹ thuật có thể cho phép bạn đạt được kết quả tức thì

(Ví dụ, đáp ứng một thời hạn), nhưng đó đi kèm với chi phí lãi suất mà cần phải được trả lại. (Ví dụ, chi phí cao hơn để tiếp tục bảo trì và tái cấu trúc).

Trong Chương 2, "Thiết kế cho sự thành công", chúng tôi xác định nguyên nhân phổ biến nhất của nợ kỹ thuật. Nó là một danh sách dài và bao gồm thời hạn gần đây, thời hạn quá sớm, và sự hiểu biết hạn chế các yêu cầu. Đôi khi nó cũng bao gồm các kỹ năng hạn chế, thiếu sự hợp tác, và lập kế hoạch không hiệu quả

Nhiệm vụ. Ngoài ra, thiếu nhận thức về bất kỳ điểm nào trong những điểm này trì hoãn thực hiện đúng hành động đối với mã của bạn và tiến hành thủ tục sửa lỗi, do đó cho vay kỹ thuật lớn hơn.

Trở lại những năm 1970, Giáo sư Manny Lehman đã xây dựng một vài luật để giải thích chu trình sống của phần mềm chương trình. Tại nền tảng của lý thuyết Lehman, có quan sát thấy rằng bất kỳ phần mềm nào của hệ thống đang phát triển xấu đi theo thời gian khi sự phức tạp của nó phát triển. 40 năm sau, rất nhiều các kiến ​​trúc sư phần mềm và các nhà phát triển đã phát hiện ra rằng phần mềm xấu đi thực sự là một lực bê tông và phải được giải quyết bằng những hành động cụ thể để tồn tại các dự án phần mềm.

Các luật và khái niệm của Lehman, như Big Ball of Bùn và nợ Kỹ thuật, tất cả đều đề cập đến cùng một cơ học có thể dẫn các dự án phần mềm xuống dốc trượt. Lehman đã xây dựng pháp luật một thời gian dài trước đây và chủ yếu là từ một quan điểm toán học. Bong bóng lớn của nợ bùn và kỹ thuật nổi lên như những vết cắn thực sự của cuộc sống trong các hào.

Cuối cùng, chúng tôi đã nhận ra rằng tất cả các phần mềm cần phải được duy trì tất cả thời gian nó được sử dụng. Sau đó, một phần mềm tốt là phần mềm cho vay chính nó để được refactored một cách hiệu quả. Cuối cùng, tính hiệu quả của tái cấu trúc phụ thuộc vào ba yếu tố - khả năng kiểm tra, tính mở rộng

Khả năng đọc và một số lượng lớn các bài kiểm tra có liên quan để bắt bất kỳ hình thức hồi quy có thể.

**Nghệ thuật viết mã kiểm chứng**

Trong nhiều năm, các nhà phát triển .NET chỉ dựa trên các công cụ gỡ lỗi trong Microsoft Visual Studio để đảm bảo chất lượng mã của họ. Hiệu quả không thể phủ nhận của các công cụ này làm cho nó đáng giá để kết hợp hai hành động khác biệt logic: kiểm tra mã nguồn thủ công, và làm việc xung quanh các lỗi để irst sao chép và sau đó họ fix.

Trong nhiều năm, cách tiếp cận này hoạt động khá độc đáo.

Tuy nhiên, trong thập kỷ qua, một sự thay đổi lớn diễn ra dưới mắt .NET của chúng tôi: các nhóm phát triển trở nên chú ý nhiều hơn đến việc kiểm tra tự động. Bằng cách nào đó các nhà phát triển đã đánh lừa rằng việc tự động thử nghiệm là một cách tốt hơn để tính ra nhanh chóng và đáng tin cậy những gì có thể là sai và, quan trọng hơn, cho dù các tính năng nhất định vẫn đang hoạt động sau khi một số thay đổi đã được thực hiện. Trình gỡ rối vẫn chức năng và sử dụng, nhưng chủ yếu là chúng được sử dụng để tiến hành từng bước trong các phần đặc biệt của mã để điều tra những gì sai.

Tự động thử nghiệm thêm một chiều hướng hoàn toàn mới cho phát triển phần mềm.

Nói chung, chúng tôi nghĩ rằng đó chỉ là một sự thay đổi tự nhiên do một bản năng con người để thích nghi. Được công nhận sự cần thiết của phần mềm thử nghiệm một cách tự động - chúng ta có thể gọi nó là sự cần thiết của việc áp dụng RAD mô hình thử nghiệm - đưa ra một điểm quan trọng khác: viết phần mềm dễ kiểm tra.

Đây là nơi testability phù hợp của nó.

**Testability là gì, còn gì nữa?**

Trong bối cảnh của kiến ​​trúc phần mềm, một sự khôn ngoan được chấp nhận rộng rãi đối với khả năng kiểm tra mô tả nó như dễ thực hiện kiểm tra mã. Và mã kiểm tra chỉ là quá trình kiểm tra phần mềm để đảm bảo rằng nó hoạt động như mong đợi, không chứa lỗi, và yêu cầu satisies. Một khẩu hiệu phổ biến giải quyết tầm quan trọng của việc kiểm thử phần mềm đến từ Bruce Eckel và đọc như sau:

Nếu nó không được kiểm tra, nó sẽ bị hỏng.

Trên bề mặt, tuyên bố đó hơi có khiêu khích, nhưng nó phục vụ mục đích của cuộc gọi

Sự chú ý của mọi người về khả năng xác định một cách rõ ràng, tự động, và liên tục cho dù có hay không một số mã hoạt động như mong đợi.

Tất cả trong tất cả, chúng tôi tin rằng không có sự khác biệt giữa mã kiểm tra hoạt động và mã không thể kiểm tra mà làm việc. Chúng tôi đã sử dụng cách diễn đạt tương tự trong Chương 2 trong khi đề cập đến mã kém được viết. Vâng, tin nó hay không, có một mối quan hệ chặt chẽ giữa mã được viết tốt và mã kiểm chứng được: chúng gần như cùng một điều. Một chất lượng cơ bản của mã tốt, trên thực tế, đó là phải kiểm chứng được. Và thuộc tính Testability là tốt để có bất kể bạn thực sự viết bài kiểm tra hay không.

Vào cuối ngày, mã kiểm tra được luân phiên cùng mã sử dụng các nguyên tắc SOLID (Đơn giản,

Trách nhiệm, Mở / đóng, các nguyên tắc của Liskov, tách biệt giao diện, và sự nghịch đảo phụ thuộc)

Rộng rãi và tránh những sai lầm phổ biến của định hướng đối tượng, như đã thảo luận trong Chương 3, "Nguyên tắc của thiết kế phần mềm". Cụ thể, nguyên tắc Đảo ngược Thuộc tính ở một trong hai Lavor-Dependency Injection and Service Locator- là nhãn hiệu của mã kiểm tra được. Google Dịch dành cho doanh nghiệp:Bộ công cụ DịchTrình biên dịch Trang webGlobal Market Finder

**Nguyên lý của khả năng kiểm tra**

Kiểm tra phần mềm đơn giản là khái niệm: chỉ cần lực lượng chương trình để làm việc trên chính xác, không chính xác, thiếu, hoặc dữ liệu chưa đầy đủ, và xác minh xem kết quả có phù hợp với bất cứ kỳ vọng nào được đặt ra. Sẽ như thế nào

Bạn buộc chương trình làm việc với dữ liệu đầu vào của bạn? Bạn đo lường sự chính xác của

các kết quả? Và trong trường hợp thất bại, làm thế nào bạn sẽ theo dõi các mô-đun speciic mà không thành công?

Những câu hỏi này là nền tảng của một mô hình được gọi là Thiết kế cho Khả năng Chẩn đoán (DfT). Bất kì phần mềm được xây dựng với sự tôn trọng đầy đủ các nguyên tắc DfT là vốn có thể được kiểm chứng được và, như là một hiệu ứng phụ dễ chịu, đó là cũng dễ đọc, hiểu, và sau đó duy trì. DfT đã được phát triển như một khái niệm tổng quát

Vài thập kỷ trước trong một lĩnh vực mà không phải là phần mềm. Trên thực tế, mục tiêu của DfT là cải tiến qui trình xây dựng các mạch mức thấp trong các bảng và các chip. DfT mô tả ba thuộc tính mà bất kỳ đơn vị nào

Phần mềm phải được dễ dàng kiểm chứng được:

* **Kiểm soát** Thuộc tính kiểm soát liên quan đến mức độ mà người kiểm tra có thể áp dụng dữ liệu đầu vào fixed vào phần mềm đang được kiểm tra. Bất kỳ phần mềm nào phải được viết bằng một cách làm cho nó rõ ràng những gì các thông số được yêu cầu và những gì trở lại các giá trị được tạo ra. Ngoài ra, bất kỳ phần mềm nào nên trừu tượng các phụ thuộc của nó-cả hai tham số và các mô-đun cấp thấp và cung cấp đường cho người gọi bên ngoài tiêm chích theo ý muốn.
* **Khả năng hiển thị** Thuộc tính của khả năng hiển thị được xác định là khả năng quan sát hiện trạng phần mềm đang được thử nghiệm và bất kỳ sản phẩm nào mà nó có thể sản xuất. Tầm nhìn là tất cả về khía cạnh này- Postconditions được kiểm tra qua quá trình thực hiện một phương pháp.
* **Tính đơn giản** Các thành phần đơn giản và cực kỳ gắn kết được ưa thích hơn để kiểm tra bởi vì càng ít bạn phải kiểm tra, càng tin cậy và nhanh chóng bạn có thể làm điều đó.

Nhìn chung, tính đơn giản luôn là một thuộc tính tích cực cho bất kỳ hệ thống và trong mọi bối cảnh. Thử nghiệm là rõ ràng không có ngoại lệ.

**Tại sao testability là mong muốn?**

Như chúng ta thấy mọi thứ, testability quan trọng hơn nhiều so với bước thực sự của thử nghiệm. Khả năng kiểm tra là một thuộc tính của phần mềm đại diện cho một tuyên bố (tuyệt vời) về chất lượng của nó. Thử nghiệm là một quá trình nhằm khi kiểm tra xem mã có đáp ứng được kỳ vọng hay không. Áp dụng testability (ví dụ, làm cho mã của bạn dễ dàng kiểm chứng được) cũng giống như học cách câu cá; viết một bài kiểm tra đơn giản giống như được cho ăn một ish. Việc đưa ra một ish giải quyết vấn đề; Học câu cá là một điều khác biệt vì nó cho biết thêm các kỹ năng mới, làm cho bạn mạnh mẽ hơn, và cung cấp một cách để giải quyết cùng một vấn đề bất cứ khi nào nó xảy ra.

Khi DfT được áp dụng thành công, mã của bạn nói chung có chất lượng tốt, thích hợp với bảo trì và tái cấu trúc, và có thể được hiểu dễ dàng hơn bởi bất kỳ nhà phát triển những người xảy ra gặp nó. Trong các điều kiện như vậy, viết bài kiểm tra đơn vị có hiệu quả cao và dễ dàng hơn tổng thể.

**ROI của khả năng kiểm tra**

 Lợi tức đầu tư (ROI) của khả năng kiểm tra là tất cả trong chất lượng cải tiến của mã bạn nhận được. Viết các lớp học với mục đích làm cho chúng có thể kiểm chứng dẫn bạn đến sự đơn giản và tiến hành một bước nhỏ một thời gian. Bạn có thể nhanh chóng bắt gặp khi một lớp được cho là trở nên cồng kềnh, tại chỗ bạn cần phải tiêm phụ thuộc, và xác định được phụ thuộc thực tế bạn cần phải đưa vào tài khoản. Bạn chắc chắn có thể sản xuất mã kiểm tra mà không thực sự viết tất cả các bài kiểm tra nhiều cho từng lớp học và thành phần. Nhưng viết bài kiểm tra và các lớp với nhau giúp bạn hiểu ROI. Tuy nhiên, mục tiêu nội bộ là có mã tốt, chứ không phải các bài kiểm tra hay. Nếu bạn cần chứng minh cho chính mình, hoặc người quản lý của bạn, ROI của khả năng kiểm tra, chúng tôi khuyên bạn nên thử nghiệm với việc viết các lớp và các bài kiểm tra cùng nhau. Nó chỉ ra rằng các bài kiểm tra kết quả là một công cụ hồi quy và cung cấp bằng chứng rằng trong tất cả các điều kiện được kiểm tra (bao gồm các trường hợp phổ biến và cạnh) mã của bạn hoạt động. Các bài kiểm tra cũng cải thiện thiết kế tổng thể của các lớp, bởi vì để viết bài kiểm tra, bạn kết thúc làm cho các giao diện công cộng của các lớp học dễ dàng hơn để sử dụng. Kiểm tra chỉ là nhiều mã để viết và duy trì, và đây là một chi phí thêm. Điều này cho biết, nó chỉ ra rằng khả năng kiểm chứng là một hình ảnh cá nhân mà mỗi nhà phát triển và nhóm các nhà phát triển cuối cùng sẽ trải qua, nhưng có lẽ không phải cho đến khi thời gian phù hợp với họ.

**Chú ý** Mã thuật ngữ thường được sử dụng để chỉ một khía cạnh khó chịu của một số mã có thể chỉ ra một vấn đề nghiêm trọng hơn. Một mùi mã không phải là một lỗi cũng không phải là một vấn đề per se; Nó vẫn đề cập đến mã mà các công trình hoàn hảo. Tuy nhiên, nó đề cập đến một thực tiễn lập trình tồi hoặc việc thực hiện ít hơn lý tưởng có thể có tác động sâu sắc hơn đến phần còn lại của mã. Mã mùi làm cho mã yếu hơn. Tìm và loại bỏ mùi mã là mục tiêu chính của tái cấu trúc. Ở một mức độ nào đó, mùi mã là chủ quan và thay đổi theo ngôn ngữ và mô hình.

**Thử nghiệm phần mềm**

Thử nghiệm phần mềm của bạn xảy ra ở các cấp độ khác nhau. Bạn có các bài kiểm tra đơn vị để xác định xem các thành phần riêng lẻ của phần mềm có đáp ứng yêu cầu chức năng hay không. Bạn có các bài kiểm tra tích hợp để xác định xem phần mềm của nó trong môi trường và cơ sở hạ tầng và liệu hai hoặc nhiều thành phần hoạt động tốt với nhau. Cuối cùng, bạn có các bài kiểm tra chấp nhận để xác định xem hệ thống đã hoàn thành có đáp ứng các yêu cầu của khách hàng hay không.

**Các bài kiểm tra đơn vị**

Kiểm tra đơn vị bao gồm viết và chạy một chương trình nhỏ (gọi là dây an toàn kiểm tra) để tạo các lớp học và kích hoạt tất cả các phương pháp đã được xử lý một cách tự động. Thân của mỗi phương pháp khởi tạo các lớp học để kiểm tra và thực hiện một số hành động đối với chúng và sau đó kiểm tra kết quả.

Hình thức đơn giản nhất của nó là một chương trình viết bằng văn bản đọc các giá trị đầu vào của các trường hợp thử nghiệm và các kết quả dự kiến tương ứng từ một số iles bên ngoài. Sau đó, khai thác kiểm tra các cuộc gọi các phương pháp trên các lớp học để kiểm tra, sử dụng các giá trị đầu vào, và so sánh các kết quả với các giá trị mong đợi. Rõ ràng, viết như một kiểm tra khai thác hoàn toàn từ đầu là, ở mức tối thiểu, tốn nhiều thời gian và dễ bị lỗi. Quan trọng hơn, nó là hạn chế về khả năng thử nghiệm mà bạn có thể tận dụng. Phương pháp hiệu quả và phổ biến nhất để tiến hành kiểm tra đơn vị là sử dụng một khuôn khổ kiểm tra tự động như là MSTest (tích hợp trong Visual Studio), NUnit, hoặc xUnit. Dưới đây là một ví dụ ngắn về mã bạn cần để viết để kiểm tra các lớp của bạn với MSTest trong Visual Studio:

using Microsoft.VisualStudio.TestTools.UnitTesting;

using MyMath.Model;

namespace MyMath.Model.Tests

{

[TestClass] public class MyMathTests

{

[TestMethod] public void TestIfFactorialIsCorrect()

{

var factorial = new Factorial();

var result = factorial.Compute(5);

Assert.AreEqual(result, 120);

}

}

}

The test harness framework processes the class shown in the preceding code, instantiates it, and runs each of the methods labeled as TestMethod. Each test method performs some work on the classes under test. In the example, the test is conducted on the class Factorial to see if the method Compute produces the expected result when it is invoked with a given argument. Note that unit tests run repeatedly most commonly as part of the build process. A test passes if the behavior of the tested class is the one you assert. For example, the following is another legitimate test that passes if the sample BasketMatch class throws an argument exception, but it fails otherwise:

[TestMethod]

[ExpectedException(typeof(System.ArgumentException))]

public void TestIfBasketMatchThrowsIfNullIsPassed()

{

var match = new BasketMatch("12345", null, null);

return;

}

Ví dụ này, tuy nhiên, là thú vị vì nó dẫn đến một điểm nhỏ hơn. Giả sử mã vượt qua bài kiểm tra; Là nó đảm bảo rằng nó đã qua bởi vì các tham số thứ hai hoặc thứ ba là null? Theo lý thuyết, bạn có thể có một lỗi trong constructor mà ném một ngoại lệ đối số, nếu, nói, các đối số đầu tiên chỉ là 12345. Trong trường hợp này, thử nghiệm đi nhưng bạn bị mất một lỗi. Để chính xác hơn, bạn nên tăng gấp đôi phép thử và kiểm tra độc lập liệu các tham số thứ hai và thứ ba có giá trị không. Thậm chí tốt hơn thế, bạn nên kiểm tra tên tham số của ngoại lệ được ném ra. Tính năng này không được hỗ trợ bởi MSTest mặc dù một số phần mở rộng tồn tại hỗ trợ tính năng này. Một là SharpTestex, bạn có thể xem tại http://sharptestex.codeplex.com. Sử dụng cú pháp này, bạn có thể viết lại bài kiểm tra như được hiển thị ở đây:

[TestMethod]

public void TestIfBasketMatchThrowsWhenTeam1IsNull()

{

Executing.This(() => new BasketMatch("12345", null, "Visitors")) .Should() .Throw() .And .ValueOf .ParamName .Should() .Be .EqualTo("Team1");

}

Có thể có nhiều lý do khác nhau khiến một bài kiểm tra không vượt qua. Ở nơi irst, nó chỉ có thể là một lỗi trong thử nghiệm. Có nhiều khả năng, đó là một vấn đề trong lớp bạn kiểm tra. Trong trường hợp này, bạn đưa tay vào trình gỡ lỗi, giết lỗi và mã ix. Cuối cùng, nó có thể là vấn đề phụ thuộc.

**Quan trọng** Bao nhiêu khẳng định bạn nên có cho mỗi thử nghiệm? Nếu bạn buộc mình phải có một khẳng định cho mỗi bài kiểm tra trong sự tôn trọng hoàn toàn nguyên tắc của các bài kiểm tra phạm vi hẹp? Đó là một điểm gây tranh cãi. Đôi khi, nhu cầu khẳng định nhiều thường giấu một thực tế là bạn đang thử nghiệm nhiều tính năng trong một lần kiểm tra đơn. Tuy nhiên, nếu bạn đang kiểm tra trạng thái của một đối tượng sau một thao tác nhất định, bạn có thể cần phải kiểm tra nhiều giá trị và cần nhiều khẳng định. Bạn chắc chắn có thể ind một cách để thể hiện điều này thông qua một loạt các bài kiểm tra, mỗi với một sự khẳng định duy nhất, nhưng nó có lẽ sẽ chỉ là một rất nhiều refactoring cho chút lợi. Chúng tôi không phiền nếu có nhiều khẳng định cho mỗi bài kiểm tra miễn là mã trong bài kiểm tra chỉ đang kiểm tra một hành vi vĩ mô.

**Xử lý các phụ thuộc**

Khi bạn kiểm tra một phương pháp, bạn chỉ muốn tập trung vào mã trong phương thức đó. Tất cả những gì bạn muốn biết là liệu mã đó có cung cấp kết quả mong đợi trong các tình huống được kiểm tra. Để có được điều này, bạn cần phải thoát khỏi tất cả các phụ thuộc phương pháp có thể có. Ví dụ, nếu phương pháp gọi một lớp khác, bạn giả sử lớp được gọi ra sẽ luôn trả về kết quả chính xác. Bằng cách này, bạn loại bỏ ở gốc nguy cơ rằng phương pháp không thành công theo kiểm tra bởi vì một sự thất bại xảy ra xuống ngăn xếp cuộc gọi. Nếu bạn thử nghiệm phương pháp A và nó không thành công, lý do đã được tìm thấy độc quyền trong mã nguồn của phương pháp A và không có trong bất kỳ phụ thuộc của nó. Nó rất cao nên bạn cô lập các lớp đang được thử nghiệm từ các phụ thuộc của nó. Tuy nhiên, lưu ý rằng điều này chỉ có thể xảy ra nếu lớp học được thiết kế theo cách lỏng lẻo - ví dụ: thông qua tiêm phụ thuộc hoặc trình định vị dịch vụ. Trong một kịch bản hướng đối tượng, lớp A phụ thuộc vào lớp B khi một trong các điều kiện sau được kiểm tra:

■ Lớp A bắt nguồn từ lớp B.

■ Lớp A bao gồm một thành viên của lớp B.

■ Một trong những phương pháp của lớp A gọi phương pháp của lớp B.

■ Một trong những phương pháp của lớp A nhận hoặc trả về một tham số của lớp B.

■ Lớp A phụ thuộc vào lớp mà đến lượt nó phụ thuộc vào lớp B.

Làm thế nào bạn có thể vô hiệu hóa các phụ thuộc khi kiểm tra một phương pháp? Bạn phải thử hai lần.

Fakes and mocks Một thử nghiệm tăng gấp đôi là một đối tượng bạn sử dụng thay cho một người khác. Một thử nghiệm tăng gấp đôi là một đối tượng mà giả vờ là một thực tế mong đợi trong một kịch bản nhất định. Một lớp được viết để tiêu thụ, nói, một đối tượng triển khai giao diện ILogger có thể chấp nhận một đối tượng logger thực sự đăng nhập vào Internet Information Services (IIS) hoặc một bảng cơ sở dữ liệu. Đồng thời, nó cũng có thể chấp nhận một đối tượng mà giả vờ là một logger nhưng chỉ không làm gì cả. Có hai loại chính của kiểm tra đôi: giả mạo và. Tùy chọn đơn giản nhất là sử dụng vật giả. Một đối tượng giả là một bản sao tương đối đơn giản của một đối tượng cung cấp giao diện tương tự như đối tượng gốc nhưng trả lại giá trị mã hoá cứng hoặc các giá trị được xác định theo chương trình. Dưới đây là một đối tượng giả mẫu cho kiểu ILogger:

public class FakeLogger : ILogger

{

public void Log(String message)

{

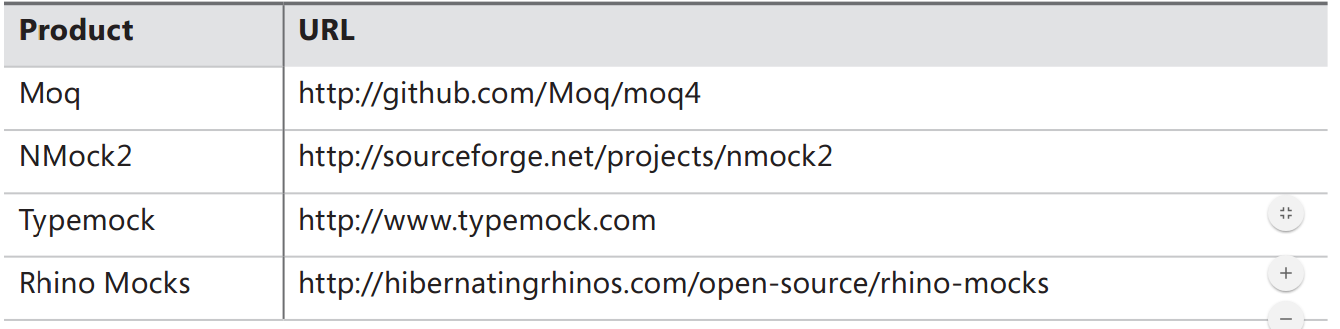
return;

}

}

Như bạn thấy, hành vi của một đối tượng giả mạo được mã hoá cứng; Đối tượng giả mạo không có trạng thái và không có hành vi đáng kể. Từ quan điểm của đối tượng giả mạo, nó không tạo ra sự khác biệt bao nhiêu lần bạn gọi một phương pháp giả mạo và khi ở mức thấp các cuộc gọi xảy ra. Thông thường, bạn sử dụng hàng giả khi bạn chỉ muốn bỏ qua một phụ thuộc. Một lựa chọn phức tạp hơn là sử dụng các đối tượng giả. Một đối tượng giả dối hiện tất cả những gì mà một giả làm, cộng với một cái gì đó nhiều hơn nữa. Theo một cách nào đó, một người giả là một đối tượng có tính cách bắt chước hành vi và giao diện của một đối tượng khác. Còn gì nhiều hơn một mô hình cung cấp cho người kiểm tra? Về cơ bản, giả mạo có thể chấp nhận veriication của ngữ cảnh của cuộc gọi phương thức. Với một mô hình giả, bạn có thể xác minh rằng một cuộc gọi phương pháp sẽ xảy ra với các điều kiện tiên quyết phải và theo thứ tự đúng với các phương pháp khác trong lớp. Viết một giả bằng tay thường không phải là một vấn đề lớn - phần lớn; Tất cả các logic bạn cần là đơn giản và không cần phải thay đổi thường xuyên. Khi bạn sử dụng hàng giả, bạn chủ yếu quan tâm đến trạng thái mà một đối tượng giả mạo có thể đại diện; Bạn không quan tâm đến tương tác với nó. Ngược lại, sử dụng mô hình giả khi bạn cần tương tác với các đối tượng phụ thuộc trong quá trình kiểm tra. Ví dụ, bạn có thể muốn biết liệu các mô hình đã được gọi, và bạn có thể quyết định trong vòng thử nghiệm các đối tượng giả phải trả lại cho một phương pháp nhất định. Viết mocks thủ công chắc chắn là một khả năng, nhưng nó là hiếm khi một lựa chọn mà bạn muốn xem xét nghiêm túc. Đối với mức độ lexibility bạn mong đợi từ một giả, bạn cần một mô hình mocking ad hoc. Bảng 4-1 liệt kê một vài mocking framework phổ biến.

***Bảng 4-1 liệt kê một vài mocking framework phổ biến.***



Lưu ý rằng không có mô hình mocking hiện đang được kết hợp trong Visual Studio, nhưng bạn có thể tìm hầu hết trong số họ dễ dàng có sẵn thông qua các gói NuGet. Dưới đây là một ví dụ nhanh về cách sử dụng một khuôn khổ chế nhạo như Moq:

[TestMethod]

public void TestIfMethodWorks()

{

// Arrange

var logger = new Mock();

logger.Setup(l => l.Log(It.IsAny()))

var match = new Match("12345", "Home", "Visitors", logger);

// Act ...

// Assert ...

}

Lớp được kiểm tra-lớp Kết hợp - có một sự phụ thuộc vào một đối tượng thực hiện

Giao diện ILogger:

public interface ILogger

{

void Log(String msg);

}

Kho lưu trữ mô phỏng cung cấp một đối tượng động được tạo ra mà chế nhạo các giao diện cho những gì các bài kiểm tra sẽ sử dụng. Đối tượng mô hình thực hiện phương thức Đăng nhập theo cách mà nó không làm gì cho bất cứ đối số chuỗi nó nhận được. Bạn không thực sự kiểm tra người logger ở đây; Bạn đang tập trung vào lớp Kết hợp và cung cấp mô hình mô phỏng nhanh và chức năng cho thành phần logger mà bộ điều khiển sử dụng nội bộ.

Bạn không cần phải tạo ra một lớp giả mạo; Bạn chỉ cần xác định mã bạn cần một phương thức đã cho để chạy khi được gọi. Đó là sức mạnh của những kẻ chế nhạo (và những khuôn mẫu chế giễu đặc biệt) khi so sánh với giả mạo.

**Lưu ý** Arrange-Act-Assert (AAA) là một mẫu phổ biến để cấu trúc các bài kiểm tra đơn vị. Trước tiên, bạn sắp xếp tất cả các điều kiện tiên quyết cần thiết và thiết lập dữ liệu đầu vào. Tiếp theo, bạn hành động theo phương pháp bạn dự định thử nghiệm và nắm bắt một số kết quả. Cuối cùng, bạn khẳng định rằng các kết quả mong đợi đã được thu được.

**Tích hợp các bài kiểm tra**

Đơn vị kiểm tra được viết bởi các nhà phát triển cho mình. Chúng tồn tại như là một công cụ hồi quy và để làm cho nhóm nghiên cứu về chất lượng đi của phần mềm. Đơn vị kiểm tra cơ bản trả lời các câu hỏi như "Chúng ta đang làm tốt?" Hoặc "Chúng ta đi đúng hướng?" Hoặc "Liệu nó phá vỡ các thành phần khác?" Phạm vi của bài kiểm tra đơn vị bao gồm một đơn vị chức năng duy nhất của hệ thống. Các đơn vị có thể là một lớp, nhưng không phải tất cả các lớp học đạt đến mức hợp lý của được coi là một đơn vị chức năng của hệ thống.

Các bài kiểm tra tích hợp là cấp độ tiếp theo của thử nghiệm-đó là khi bạn thực hiện nhiều đơn vị chức năng và xem họ có thể làm việc cùng nhau hay không. Các thử nghiệm hội nhập sau đó sẽ liên quan đến các phần khác nhau của hệ thống và thử nghiệm chúng trong một kịch bản thực sự gần với thực tế. Mục tiêu của các bài kiểm tra tích hợp là lấy thông tin phản hồi về hiệu năng và độ tin cậy của từng thành phần. Thông thường, kiểm tra tích hợp các lớp và lớp của toàn bộ hệ thống và liên quan đến cơ sở dữ liệu và dịch vụ.

Trong các bài kiểm tra đơn vị, bạn chỉ kiểm tra một đơn vị một lần, nhưng bạn nên thử bao nhiêu thành phần trong một bài kiểm tra tích hợp? Các bài kiểm tra tích hợp mất nhiều thời gian hơn các bài kiểm tra đơn vị được thiết lập và chạy. Và thời gian thiết lập này là như nhau mỗi lần bạn chạy chúng. Ví dụ, hãy tưởng tượng một kịch bản mà bạn cần một thành phần để hoạt động trên cơ sở dữ liệu. Mỗi lần chạy thử, bạn phải đảm bảo rằng cơ sở dữ liệu ở trạng thái chặt chẽ mà bạn phải khôi phục (hoặc kiểm tra ở mức tối thiểu) trước khi bạn có thể chạy thử lại.

Để tiết kiệm thời gian trong các thử nghiệm tích hợp, đôi khi bạn đi theo lối tắt của mô hình Big Bang. Trong kịch bản này, tất cả hoặc hầu hết các thành phần được kết nối và thử nghiệm, tất cả cùng một lúc. Bạn cấu hình hệ thống trong một môi trường giống như người sử dụng và đẩy một khối lượng công việc điển hình vào nó để xem làm thế nào nó đi. Bằng cách này, bạn chắc chắn sẽ tiết kiệm được rất nhiều thời gian, nhưng nếu có vấn đề gì đó, bạn có thể không biết gì khi điều tra vấn đề. Thực tế, các bài kiểm tra tích hợp có thể cho biết các bộ phận của hệ thống có hoạt động tốt không, nhưng đôi khi, nếu có nhiều phần tham gia, các bài kiểm tra cũng không cho bạn biết mô đun cụ thể nào thực sự thất bại. Rõ ràng nó có thể âm thanh, kế hoạch hội nhập kiểm tra không phải là dễ dàng. Trước tiên, bạn không thể bắt đầu cho đến khi tất cả các mô-đun có liên quan được phát hành, và các mô-đun sẽ không được phát hành cho đến khi chúng được coi là làm mất hiệu lực và làm việc của chủ sở hữu tương ứng. Thông thường, với thử nghiệm tích hợp, bạn có nguy cơ bắt đầu khá muộn, vào thời điểm mà các nhà quản lý đang bị áp lực và muốn vội vàng, có thể chỉ hy vọng rằng nó sẽ ổn thôi. Một chiến lược tốt hơn (nếu bạn được phép theo dõi nó) đang thực hiện thử nghiệm tích hợp từ dưới lên, về cơ bản bao gồm việc tích hợp các mô đun cấp thấp irst và càng sớm càng tốt. Mỗi mô đun có thể chỉ là một chút ít hơn một hộp rỗng vào lúc này, nhưng với sự giúp đỡ của một số lớp học sơ cấp, bạn sẽ sớm thấy bức tranh lớn. Các vấn đề có thể được báo cáo sớm, và chúng có thể ngăn ngừa những vấn đề tương tự khác xuất hiện sau này.

**Mẹo** Việc triển khai sơ bộ ở đây cũng có nghĩa là có các lớp có giao diện phù hợp nhưng không có logic chỉ trả về dữ liệu đã được lưu. Điều thú vị là bạn phải chú ý làm thế nào để phù hợp với một trong các bài thuốc TDD: bắt đầu phát triển các lớp học của bạn, viết mã đủ để vượt qua bài kiểm tra irst (unit) test. Nếu bạn làm như vậy, bạn cũng có thể vị trí chính mình cho các bài kiểm tra tích hợp. Chúng tôi sẽ quay lại TDD chỉ trong chốc lát.

**Chú ý** Một trong những điểm đau của việc khởi sự vấn đề của health.gov là thiếu các bài kiểm tra tích hợp đầy đủ, rất quan trọng trong hệ thống lớn, đa dạng và phân tán. Bạn vừa chứng kiến cách tiếp cận Big Bang để kiểm tra tích hợp một trong những nguyên nhân phổ biến nhất gây ra những rắc rối ban đầu. Bạn có thể đọc một bản phân tích thú vị và ngắn về câu chuyện health.gov ở đây: <http://bit.ly/1bWPgZS>.

**Thử nghiệm chấp nhận**

Thử nghiệm chấp nhận là các bài kiểm tra theo hợp đồng mà bạn chạy để xác định liệu hệ thống được triển khai đầy đủ có đáp ứng được các yêu cầu của bạn hay không. Đây là thực tế chắc chắn duy nhất chúng ta biết về các bài kiểm tra chấp nhận. Ai viết và chạy thử nghiệm thay đổi; Do đó, việc xây dựng các thử nghiệm như vậy cũng khác nhau. Đó là một ý tưởng chung - hầu hết được hấp thụ từ Lập trình cực đoan - các bên liên quan và nhân viên bảo đảm chất lượng viết bài kiểm tra chấp nhận lại với nhau, bắt đầu từ những câu chuyện của người dùng và chọn các tình huống có liên quan để kiểm tra. Mỗi câu chuyện của người dùng có thể có một bài kiểm tra chấp nhận hoặc nhiều bài kiểm tra, nếu thích hợp. Các bên liên quan quan tâm đến khía cạnh tích cực; QA chăm sóc các trường hợp cạnh. Cuối cùng, các nhà phát triển xem xét các bài kiểm tra để đảm bảo rằng các hành động thực tế có ý nghĩa trong hệ thống triển khai. Một bài kiểm tra chấp nhận là một bài kiểm tra hộp đen. Tất cả những vấn đề đó là kết quả thực tế bạn nhận được; Không có kiến ​​thức về internals. Các bài kiểm tra chấp nhận cũng nên làm việc như các bài kiểm tra hồi quy trước khi phát hành ứng viên và để phát hiện các thay đổi vi phạm khi một phiên bản mới hơn được phát hành. Âm thanh đơn giản? Vâng, có lẽ vì có một vài điểm chúng tôi để lại không được nêu ra. Cụ thể, làm thế nào để bạn viết các bài kiểm tra chấp nhận? Trong hình thức đơn giản nhất, một bài kiểm tra nghiệm thu là tập hợp các hướng dẫn từng bước được lặp lại với hệ thống được điều chỉnh đầy đủ để quan sát các đầu ra. Để chạy thử nghiệm chấp nhận thường xuyên, bạn nên ind một cách để tự động hoá chúng. Điều này có nghĩa là các bài kiểm tra chấp nhận phải được mã hoá dưới dạng các chương trình khi có thể.

Nhưng nếu bạn mong đợi các bên liên quan để viết bài kiểm tra chấp nhận, họ nên sử dụng ngôn ngữ nào? Làm thế nào để bạn đảm bảo rằng các bài kiểm tra chấp nhận được đồng bộ với tất cả các chi tiết của việc thực hiện? Các bài kiểm tra chấp nhận phải có trước hay sau khi toàn bộ hệ thống được xây dựng? Tất cả những câu hỏi này không có câu trả lời deinitive, và hầu hết các đội chỉ làm theo khuyến cáo của Extreme Programming và deine các chi tiết thực hiện trên cơ sở mỗi-dụ. Một vài năm trước, Dan North đã đề xuất một phương pháp Agile (BDD), thiết kế theo hướng hành vi (Behavior-Driven Design) (BDD) với mục tiêu là làm sạch sự hỗn loạn xung quanh mọi khía cạnh của việc phát triển dự án, bao gồm các bài kiểm tra chấp nhận tự động.

**Thiết kế dựa trên hành vi**

BDD nhằm mục đích làm rõ các tương tác tuần hoàn giữa bất kỳ bên liên quan nào dẫn đến việc phân phối sản phẩm nội bộ. Một trong những trụ cột của BDD là sản xuất các đầu ra tốt từ mỗi tương tác. Để thực thi hướng dẫn đó, BDD đề nghị bạn làm việc xung quanh một số phát biểu sử dụng một ngôn ngữ tương đối cao ở irst. Các tuyên bố này sau đó được các chuyên gia xác định cụ thể hơn đến mức họ trở thành mã thực và tạo thành một sự thực hiện cụ thể của hệ thống. Các tuyên bố BDD được tập trung vào ba điểm: điều kiện tiên quyết, hành động và đầu ra. Bạn có thể tạo các câu lệnh này theo cách thủ công bằng cách sử dụng bất kỳ trình soạn thảo văn bản nào và sử dụng chúng dưới dạng các bài kiểm tra chấp nhận. Tốt hơn là bạn có thể thực hiện công việc BDD của mình trong một công cụ tích hợp với IDE của bạn để bắt đầu với các câu lệnh BDD và kết thúc bằng mã và tất cả các bài kiểm tra-đơn vị và sự tích hợp-bạn có thể cần trên đường đi.

Một câu lệnh BDD điển hình sẽ như sau:

**Kịch bản:** Người dùng đang theo dõi số điểm của một trận polo phù hợp

**Cho** Người sử dụng trên trang / trận đấu

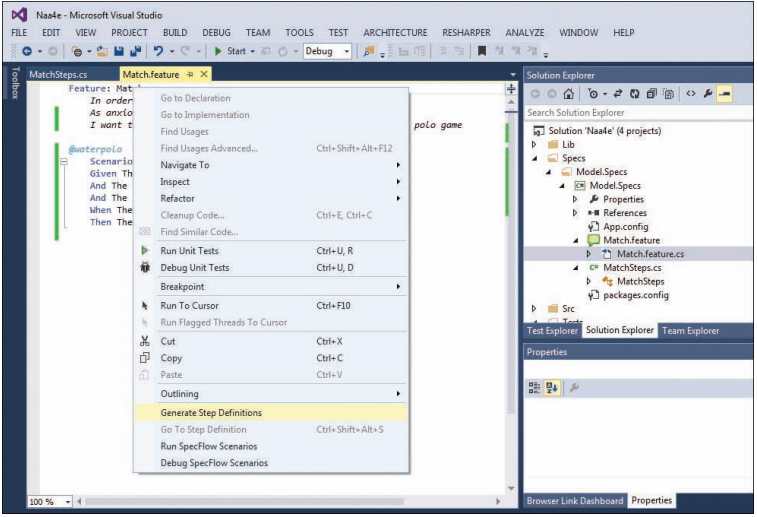
**Và** trận đấu được cấu hình đầy đủ

**Và** trận đấu bắt đầu

**Khi** Người dùng nhấp vào nút Mục tiêu ở bên trái

**Sau đó** Hệ thống cập nhật trận đấu ghi một bàn cho Trang chủ

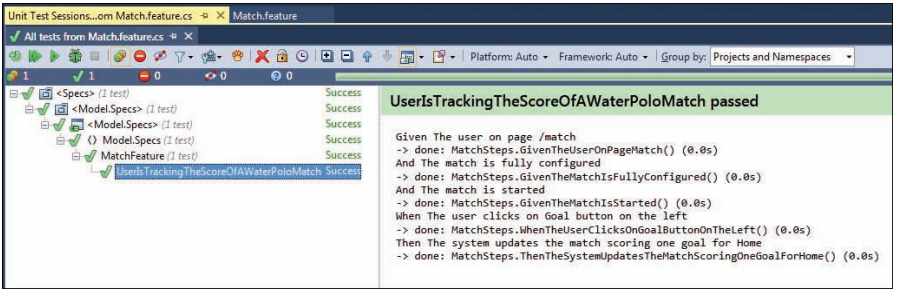
Một khuôn khổ BDD có thể biến đổi một tuyên bố đơn giản thành mã chạy được và các bài kiểm tra. Trong Visual Studio 2013 (và các phiên bản trước đó), bạn có thể sử dụng SpecFlow, có sẵn như một phần mở rộng từ Visual Studio Gallery Extension. Visual Studio cũng đi kèm với một vài gói NuGet bạn có thể sử dụng để cấu hình một dự án thử nghiệm để được BDD nhận thức. Để biết thêm thông tin, hãy kiểm tra <http://www.speclow.org>



Hình 4-1 Các mở rộng SpecFlow trong hành động trong Visual Studio 2013.

Khi bạn hoàn thành tất cả các câu lệnh Given / When / Then, tất cả những gì bạn thực hiện là chạy các bài kiểm tra đơn vị từ cùng một trình đơn ngữ cảnh thể hiện trong hình 4-1.

Hình 4-2 cho thấy một kết quả có thể.



Hình 4-2 Các kết quả của việc chạy các bài kiểm tra đơn vị từ các bài kiểm tra SpecFlow.

**Thực tiễn chung về kiểm thử phần mềm**

Trong khi mọi người đồng ý về tầm quan trọng của việc kiểm tra trong các dự án phần mềm, vẫn còn một số điểm gây tranh cãi lớn về kiểm tra mà mọi người giải quyết theo cách riêng của họ. Những điểm mở này xoay quanh vai trò của thử nghiệm, thời gian bạn viết cho họ, số lượng và vị trí của bài kiểm tra, v.v ... Hãy nhanh chóng giải quyết một số trong số họ với hy vọng làm rõ mọi thứ một chút. Mục đích của chúng tôi ở đây là tổng hợp các vị trí khác nhau, thêm các chú thích và nhận xét riêng của chúng tôi và tạo ra một bản tóm tắt trung thực các sự kiện và quan điểm. Nói cách khác, chúng tôi không phải là bác sĩ, vì vậy chúng tôi sẽ không viết đơn thuốc!

**Phát triển dựa trên thử nghiệm**

Phát triển dựa trên thử nghiệm (TDD) là một quá trình dẫn các nhà phát triển viết mã của họ bắt đầu từ các bài kiểm tra. Đối với mỗi lớp hoặc phương pháp, một nhà phát triển TDD bắt đầu bằng một bài kiểm tra ban đầu không thành công đại diện cho hành vi dự kiến. Sau đó, lặp đi lặp lại, nhà phát triển thêm mã vào phương thức hoặc lớp để vượt qua bài kiểm tra và refactors mã để cải thiện nó, bất kể điều đó có ý nghĩa gì. Tái cấu trúc này bao gồm thêm nhiều chức năng, cải thiện thiết kế, cải thiện tính dễ đọc, tiêm phụ thuộc, tuân thủ các hướng dẫn của một số loại, vân vân.

Nó chỉ ra rằng TDD mang lại một vài sự kiện cơ bản cho bảng:

■ Các thử nghiệm không phải là mục tiêu chính của TDD. Khá ngược lại: Các bài kiểm tra tập trung vào việc cho phép thiết kế lý tưởng xuất hiện tự nhiên trong quá trình tái cấu trúc liên tục.

  ■ Mục đích cuối cùng của các bài kiểm tra trong TDD không phải là độ bao phủ mã cao mà là thiết kế tốt hơn.

  ■ Tái cấu trúc liên tục không phải là tùy chọn. Tái cấu trúc lại là nội dung của TDD nhiều hơn là kiểm tra.

TDD là một trong những chủ đề phân chia mọi người thành hai nhóm đối nghịch và mạnh mẽ (kể cả ightight). Nhiều người nghĩ rằng chỉ vì TDD là rộng rãi dựa trên các bài kiểm tra, đơn vị kiểm tra là mục tiêu cuối cùng (và bảo hiểm mã cùng với nó). Trong nháy mắt thứ hai, những người này tự hỏi nơi mà các lợi ích của TDD thực sự là, bởi vì viết các bài kiểm tra ngớ ngẩn cho một phương pháp cơ bản có vẻ như, tốt, giống như rất nhiều trên không. Tiếp theo, những người này hình thành ý kiến rằng tốt các bài kiểm tra có thể được thêm vào một thời gian sau và chỉ trong các lĩnh vực mã nơi bạn cần chúng để được để nắm bắt hồi quy. Cuối cùng, những người này có xu hướng chống lại TDD và đôi khi nói rằng nó là ít sử dụng.

Khi chúng tôi cố gắng tóm tắt trong các điểm đầu tiên của phần này, chúng tôi cảm thấy rằng có một sự hiểu lầm phổ biến về TDD. Kiểm tra là phương tiện chứ không phải là kết thúc. TDD, mặt khác, là một quá trình phát triển phần mềm đồng bằng đó là tốt như nhiều người khác ra khỏi đó. TDD không phải là một cuộc trưng cầu trên các bài kiểm tra đơn vị. Để sử dụng các từ của Robert C. Martin, "Hành vi viết bài kiểm tra đơn vị là một hành động thiết kế chứ không phải veriication". Cuối cùng, TDD là về thiết kế và chỉ sử dụng các phép thử làm phương tiện để đạt được thiết kế tốt. Tuy nhiên, vào cuối của quá trình này, bạn hy vọng sẽ có mã được thiết kế tốt, và như một phí bảo hiểm miễn phí, một loạt các bài kiểm tra có mức độ bảo vệ mã cao.

Bạn phải biết rõ vai trò của các bài kiểm tra nếu bạn nắm lấy TDD, và bạn phải nhận thức tương tự về vai trò của việc tái cấu trúc. Bạn không thể yêu cầu bồi thường bạn đang làm TDD nếu bạn dừng lại ở mã mà làm cho bất kỳ thử nghiệm vượt qua. Refactoring là công cụ bạn sử dụng để có được thiết kế tốt hơn

Giống như bất kỳ quá trình khác, trong bất kỳ thuật ngữ của ngành công nghiệp, TDD không phải là hoàn hảo. Đặc biệt, TDD liên quan đến một số công việc và liên tục ném đi các bài kiểm tra và mã. Trong một cách, TDD không bao giờ cho bạn cảm giác rằng một lớp học hay một phương pháp được thực hiện tốt tại một số điểm, nhưng cuối cùng thì đó là bản chất của lặp đi lặp lại.

Dù sao, viết bài kiểm tra trên con đường để hiểu mã ngụ ý một sự thay đổi của tư duy. Một số công ty buộc thay đổi này đối với các nhà phát triển. Ý kiến của chúng tôi là nếu bạn đang tìm kiếm một cải tiến quy mô lớn về chất lượng mã và các kỹ năng của các nhà phát triển cá nhân, TDD có thể là-tất cả mọi thứ được coi là lựa chọn tốt nhất bởi vì nó cung cấp cho bạn một thủ tục từng bước: Không kiểm tra, thêm mã, làm cho vượt qua bài kiểm tra, refactor, lặp lại. Bởi vì mã của bạn phải vượt qua các bài kiểm tra, bạn sẽ làm cho thiết kế của nó bị lỏng lẻo, xử lý các phụ thuộc cẩn thận, tránh các phương pháp tĩnh và toàn cầu, giữ cho các lớp học luôn gọn gàng và trung bình, luôn chọn lựa chọn đơn giản nhất - tất cả những điều làm cho mã trở nên tốt hơn.

**Tôi có thể viết bài kiểm tra của mình sau đó không?**

Bởi vì TDD là một quá trình phát triển phần mềm đơn giản, nó không phải là một sự lựa chọn bắt buộc và không phải là cách an toàn duy nhất để làm sạch mã. Bạn có thể thiết kế mã của bạn một cách hiệu quả bằng các kỹ thuật khác, có giá trị như nhau. Tuy nhiên, khi bạn có mã, có thể nói bạn cần phải kiểm tra. Nếu bạn không làm TDD-nơi các bài kiểm tra là một phần của quá trình kiểm tra trở thành kết thúc của bạn tại một số điểm. Và bạn chỉ cần viết chúng khi bạn cần chúng và cho các lĩnh vực mã mà bạn nghĩ rằng chúng được yêu cầu. Bạn chắc chắn có thể viết bài kiểm tra sau khi bạn viết mã. Điều đó không nhất thiết phải tốt hay xấu; Nó chỉ là một quá trình phát triển phần mềm khác. Trong trường hợp này, các bài kiểm tra là để xác nhận và hồi quy và đại diện cho một khoản mục và chi phí bổ sung trong không gian làm việc của dự án. Như vậy, nó là một mục (và một chi phí) có thể được cắt bỏ khi phải đối mặt với thời hạn gần đây hoặc ngân sách chặt chẽ. Trong TDD, các bài kiểm tra là một phần không tách rời của quá trình phát triển phần mềm. Nếu bạn viết mã irst và các bài kiểm tra sau khi bạn cần chúng, sau đó kiểm tra và mã là các thực thể riêng biệt. Cuối cùng, các bài kiểm tra phục vụ cho mục đích xác nhận mã và bắt hồi quy. Và đây là những khía cạnh cơ bản của sự phát triển mã hiện đại. Có thể tìm thấy một bài đọc thú vị về cuộc thi thử nghiệm so với kỳ thi sau khi thử nghiệm tại trang web sau: <http://hadihariri.com/2012/10/01/tdd-your-insurance-policy>.

**Tôi có nên quan tâm về bảo hiểm mã?**

Mục đích chính của bài kiểm tra đơn vị và tích hợp là làm cho nhóm phát triển thống nhất về chất lượng của phần mềm họ đang sản xuất. Về cơ bản, kiểm tra đơn vị thông báo cho nhóm cho dù họ đang làm tốt và đang đi đúng hướng. Nhưng làm thế nào đáng tin cậy là kết quả của các xét nghiệm của bạn? Nói chung, thử nghiệm thành công dường như là một thước đo chất lượng tốt nếu các bài kiểm tra bao gồm đủ mã; Nhưng không có sự tương quan thực tế nào đã được chứng minh là tồn tại giữa bảo vệ mã nguồn và chất lượng phần mềm thực tế. Và tương tự, không có thỏa thuận chung nào đã từng đạt được về những gì "đủ" có nghĩa là số lượng. Một số nói rằng 80 phần trăm của bảo hiểm mã là tốt; Một số thậm chí không bận tâm trích dẫn một igure. Chắc chắn, các hình thức bảo hiểm mã đầy đủ thực sự không thực tế để đạt được thậm chí không thể. Phạm vi bảo hiểm của mã là một thuật ngữ khá chung chung có thể đề cập đến các tiêu chí tính toán khá khác nhau, chẳng hạn như chức năng, tuyên bố, quyết định và phạm vi đường dẫn. Chức năng bảo hiểm đo lường cho dù mỗi chức năng trong chương trình đã được thực hiện trong một số xét nghiệm. Tuyên bố bảo hiểm có vẻ chi tiết hơn ở các dòng cá nhân của mã nguồn. Quyết định bao gồm các biện pháp đo lường các chi nhánh (như một tuyên bố nếu) đánh giá, trong khi phạm vi bảo hiểm đường đi kiểm tra xem mọi tuyến đường có thể thông qua một phần nhất định của mã đã được thực hiện. Mỗi tiêu chí cung cấp một quan điểm vào mã, nhưng những gì bạn nhận được chỉ là con số được giải thích. Vì vậy, nó có thể có vẻ như là thử nghiệm tất cả các dòng mã (có nghĩa là, nhận được một giả thuyết bảo hiểm 100 phần trăm) là một điều tuyệt vời; Tuy nhiên, giá trị cao hơn cho vùng phủ sóng của đường dẫn có lẽ là mong muốn hơn bởi vì nó nhìn vào đường dẫn mã thực. Phạm vi bảo vệ mã là hữu ích vì nó cho phép bạn xác định mã nào chưa bị chạm vào bởi các bài kiểm tra. Đồng thời, nó không cho bạn biết nhiều về các bài kiểm tra tốt đã thực hiện các mã. Bạn muốn một ví dụ tốt đẹp? Hãy tưởng tượng một phương pháp xử lý một số nguyên. Bạn có thể có báo cáo bảo hiểm 100 phần trăm cho nó, nhưng nếu bạn thiếu một bài kiểm tra trong đó phương pháp nhận được một giá trị không hợp lệ, nằm ngoài phạm vi phạm vi, bạn có thể có một ngoại lệ trong thời gian chạy mặc dù đã kiểm tra thành công tất cả các bài kiểm tra.

**Mã bảo hiểm và tai nạn của Ariadne 5**

Theo báo cáo, vào năm 1996, tên lửa Ariadne 5 tự hủy 40 giây sau khi cất cánh vì một ngoại lệ tràn ngập không được kiểm soát thông qua mã cho đến khi nó đạt đến mô-đun tự hủy, nó nói với tên lửa đó Quỹ đạo và nó an toàn hơn để tiêu diệt nó. (Câu chuyện đầy đủ là ở đây:

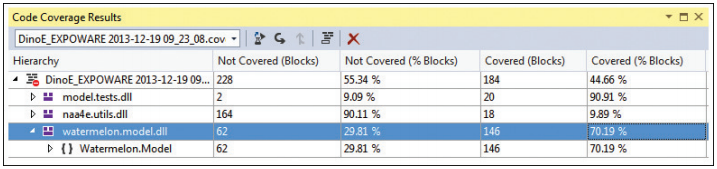
<http://www.around.com/ariane.html>.)

Phải thừa nhận rằng, chúng tôi không xem xét mã nguồn của dự án. Tuy nhiên, nếu báo cáo là chính xác, chúng ta có thể dễ dàng đoán rằng mã chứa một phương pháp cố gắng chuyển đổi một điểm loating thành một số nguyên unsigned. Chúng tôi cũng không có ý tưởng về phạm vi bảo hiểm mã cho dự án, nhưng chúng tôi giả định bạn không cố gắng khởi động một tên lửa 7 tỷ đô la mà không có nhiều kiểm tra.

Không có vấn đề gì về phạm vi bảo hiểm mã thực tế là, chúng tôi muốn nói rằng một bài kiểm tra kiểm tra một giá trị điểm loating rất lớn, ngoài phạm vi phạm vi đã được dễ dàng để viết và đã có thể cảnh báo về vấn đề. Hoặc một văn bản như vậy đã được viết nhưng kết quả của nó đã được hạnh phúc

Hình 4-3 hiển thị đầu ra của công cụ mã che phủ trong Visual Studio 2013. Như bạn thấy, nó là

Chỉ cần số có nghĩa là rất ít mà không có giải thích và bối cảnh.



Hình 4-3 Giao diện người dùng Bảo vệ Mã trong Visual Studio 2013.

Cuối cùng, mã bảo hiểm chỉ là một số đối tượng đo lường cụ thể. Sự liên quan của các bài kiểm tra là điều thực sự quan trọng. Mù quáng tăng phạm vi bảo vệ mã hoặc, tệ hơn nữa, yêu cầu các nhà phát triển đạt đến một ngưỡng bảo hiểm nhất định không đảm bảo bất cứ điều gì. Một ứng dụng được kiểm tra tốt là một ứng dụng có phạm vi phủ sóng cao về các tình huống có liên quan.

**Tôi nên kiểm tra phần nào trong mã của tôi?**

Trong hình 4-3, bạn nhận được trung bình 44 phần trăm cho dự án, nhưng mức độ phủ sóng khá thấp trong một thư viện tiện ích và cao hơn nhiều trong một thư viện thực hiện mô hình được ứng dụng sử dụng. Điều đó có nghĩa gì không?

Như chúng ta thấy mọi thứ, một thư viện tiện ích là một điều tương đối tĩnh: hoặc nó hoạt động hoặc nó không. Nếu một chức năng có được tìm thấy là lỗi, nó được ixed, và đó là nó. Nếu tiện ích bạn viết ở đó-nói, một chức năng rút ngắn chuỗi và thêm một dấu chấm lửng ở đầu - khá chung chung và có thể được gọi từ nhiều nơi, bạn vẫn có thể muốn có một số bài kiểm tra để thực hiện nó, đặc biệt là với các trường hợp cạnh. Nhưng các bài kiểm tra đơn vị có thể có liên quan hơn và mở rộng đến các trường hợp và trường hợp thông thường trong các lĩnh vực mà mã hoạt động kinh doanh.

Nói chung, bạn có thể gọi nó là lớp kinh doanh, cụ thể hơn và phù hợp với tầm nhìn của cuốn sách này, lớp miền. Bạn nên tập trung nỗ lực thử nghiệm của mình khi thực hiện các quyết định quan trọng liên quan đến logic của ứng dụng. Hình 4-4 cho thấy một worklow điển hình sau một hành động của người dùng. Lệnh đi qua lớp ứng dụng và kết quả trong một chuỗi các cuộc gọi được sắp xếp tới các phần khác nhau của đầu cuối, bao gồm các dịch vụ bên ngoài, mô hình miền, dịch vụ tên miền và lớp tồn tại. Khi một phản hồi đã được tạo ra, nó được định dạng trở lại cho giao diện người dùng. Hình bóng trong Hình 4-4 cho biết mức độ ưu tiên của kiểm tra, với bóng tối nhất thể hiện các mục có mức độ ưu tiên cao nhất cho thử nghiệm.

External services

Workflow behind the

user action

Preparing response

User interface

Domain logic

and services

Persistence

Hình 4-4 Màu tối nhất cho biết các bài kiểm tra là ưu tiên cao nhất.

Lớp miền là phần phức tạp nhất và phần bị ảnh hưởng nhiều nhất bởi các yêu cầu churn. Sau đó, nó là phần mà hầu hết các lỗi có thể hiển thị. Việc sắp xếp và các bộ điều hợp tới và từ các định dạng trình bày có nhiều phần cơ học hơn của ứng dụng và hành vi của chúng có thể được quan sát bằng nhiều cách khác nhau: bằng kiểm tra mã, kiểm tra, kiểm tra chất lượng và gỡ lỗi đồng bằng.

Theo nguyên tắc chung, chúng tôi khuyên bạn nên ưu tiên thử nghiệm cho các thành phần được tiêu thụ bởi rất nhiều thành phần khác, các thành phần logic cần hiểu sâu về yêu cầu và các thành phần thuộc sở hữu của nhiều nhà phát triển. Đừng nhắm mắt nhắm mắt và viết bài kiểm tra, và đừng ép các nhà phát triển của bạn vào đó. Thật ngớ ngẩn khi có rất nhiều bài kiểm tra mà không có cơ hội để thất bại.

**Lưu ý** Khi viết một bài kiểm tra đơn vị, bạn nên biết rất nhiều chi tiết về nội bộ của đơn vị bạn đang thử nghiệm. Thử nghiệm đơn vị là một hình thức thử nghiệm hộp trắng, ngược với việc kiểm tra hộp đen, trong đó người kiểm tra không cần biết kiến thức của nội bộ và giới hạn kiểm tra để nhập các giá trị đầu vào nhất định và mong đợi các giá trị đầu ra cho trước.

**Tự động tăng cường các bài kiểm tra đơn vị**

Giả sử bạn không phải là người cuồng tín TDD. Tại một số điểm, bạn có thể chỉ cần xảy ra để có một lớp C # và không có bài kiểm tra đơn vị cho nó. Bạn sẽ bắt đầu từ đâu? Bởi vì bạn đã viết mã, bạn là người tốt nhất để xác định điều gì là quan trọng để kiểm tra. Tuy nhiên, luôn luôn quá dễ dàng cho con người để bỏ lỡ, bỏ qua, hoặc đánh giá thấp một trường hợp cạnh, đặc biệt là trong một codebase có tiềm năng lớn. Kết quả là, tất cả các kiểm tra có thể vượt qua, nhưng mã vẫn có thể không thành công.

Tiện ích Microsoft Pex dành cho Visual Studio có thể giúp bạn giải quyết vấn đề này vì nó nhằm hiểu được logic của mã của bạn và đề xuất các bài kiểm tra phù hợp bạn cần có. Bên trong, Pex sử dụng kỹ thuật phân tích tĩnh để xây dựng kiến ​​thức về hành vi của mã của bạn.

Ngoài ra, nếu bạn đã tham số hóa các bài kiểm tra trong dự án thử nghiệm của bạn, Pex có thể định ra sự kết hợp các tham số nào cần được truyền để cung cấp cho bạn toàn bộ các kịch bản khả thi. Cuối cùng, nếu bạn sử dụng các Hợp đồng Mã NET trong mã của bạn, Pex sẽ sử dụng thông tin đó để điều chỉnh các bài kiểm tra đơn vị mà nó gợi ý hoặc tạo cho bạn.

Tóm lại, Pex là một công cụ thử nghiệm hộp trắng sáng tạo có thể được sử dụng theo hai cách: như là một trợ giúp để tạo ra các bài kiểm tra đơn vị không quan trọng, và như một người đánh giá ngang hàng có thể nhanh chóng xem mã của bạn và các lỗ hổng ind và thiếu sót trong đó. Bạn có thể tải Pex từ <http://research.microsoft.com/en-us/projects/pex>.

**The practice of code extensibility**

As mentioned by Dave Thomas and Andy Hunt, authors of the aforementioned book The Pragmatic Programmer (Addison-Wesley, 1999), all programming work is a form of maintenance. Only a few minutes after being typed for the irst time, a class enters its ininite maintenance cycle. Most of the time, maintenance entails refactoring the existing codebase to patterns or just to a cleaner design.

However, under the capable umbrella of the term maintenance also falls the attribute of code extensibility—namely, extending existing code with new functions that integrate well with existing functions.

We can say for code extensibility nearly the same that Donald Knuth said of optimization: when prematurely done, it can be the root of all software evil. The ego of many developers is strongly stimulated by the idea of writing code that can be extended with limited effort. This might easily lead to over-engineering. A picture is worth a thousand words here: Figure 4-5 shows what we mean. The picture was inspired by the comic strip at <http://xkcd.com/974>.

|  |  |  |
| --- | --- | --- |
| **Would you pass the salt?** | |  | | --- | | **You bet!**  **Just wait a moment ...** | |

### MINUTES LATER

**You’ll save time in**

**the long run.**

**Hey, I asked for the salt.**

**I’m not supposed to**

**wait 20 minutes to**

**get some salt ...**

**I’m developing a**

**service to pass**

**arbitrary condiments ...**

**Hinh 4-5**

Các vấn đề chung của quá kỹ thuật.

Khi thuộc tính mở rộng là cơ bản, bạn có thể muốn xem xét ba khía cạnh sau đây trong mã của bạn: thiết kế dựa trên giao diện, kiến trúc plugin và các máy trạng thái.

Quan trọng Hầu hết thời gian, bạn cần phải chứng minh rằng bạn cần một extensible 818/5000 và thiết kế lexible. Sự cần thiết cho một thiết kế trừu tượng hơn và lexible là kết quả điển hình của một phiên tái cấu trúc. Một thiết kế mở rộng hơn cũng có thể bề mặt trong giai đoạn đầu của sprints khi bạn sắp triển khai một tính năng mới.

**Thiết kế dựa trên giao diện**

Giao diện đại diện cho một hợp đồng giữa hai bên phần mềm. Khi một thành phần phần mềm-nói, một lớp- khai báo nó có thể xử lý một giao diện nhất định, hiệu quả ròng là sau đó nó có thể làm việc với bất kỳ thành phần nào khác, hoặc là một trong những ngày hiện tại hoặc sẽ được viết trong tương lai, để lộ cùng một giao diện.

Khái niệm này là ở nền tảng của mã mở rộng.

Sự phê phán của cách tiếp cận này là trong số các thành phần phần mềm mà bạn nên thiết kế với khả năng mở rộng trong tâm trí. Việc thực hiện tất cả các lớp được thiết kế để mở rộng và tất cả các chiến lược mà họ cần phải giải quyết rất khó thực hiện. Không đúng vào thời điểm này là những gì dẫn bạn có quyền quá kỹ thuật.

Thử thách - và tin tưởng chúng ta, đó thực sự là một thách thức khó khăn - chỉ là một sự thách thức khi trích xuất một giao diện từ chiến lược và chia sẻ nó giữa nhiều thành phần là cái gì đó thực sự mang lại cho bạn nhiều giá trị hơn.

**Kiến trúc trình cắm**

Khi sự mở rộng của từ được diễn tả, nhiều người chỉ nghĩ đến plugin. Plugin là một thành phần bên ngoài mà thêm một tính năng đặc biệt vào một ứng dụng hiện có mà không phải là một phần của nó. Tuy nhiên, để điều này xảy ra, ứng dụng máy chủ lưu trữ phải được thiết kế sao cho nó phát hiện và xử lý các plugin.

Ứng dụng máy chủ điều khiển giao diện lập trình của các plugin, xác định đơn phương những plugin dữ liệu nào nhận được sau khi chúng được tải và đặc tả khi chúng được gọi và nơi chúng phải được đặt để được công nhận. Ứng dụng máy chủ lưu trữ cũng cần có một lớp đặc biệt của mã xử lý việc tải và bốc dỡ các plugin bổ sung có sẵn.

Có rất nhiều cách để viết một kiến ​​trúc dựa trên plugin. Nội dung của kiến ​​trúc như vậy được tóm tắt tốt bằng mô hình Plugin như mô tả của Martin Fowler. (Xem http://martinfowler.com/ eaaCatalog / plugin.html.) Mã của bạn tại một số điểm (gọi là điểm mở rộng) sử dụng một thành phần để thực hiện một nhiệm vụ nhất định. Nhiệm vụ được trừu tượng hóa cho một giao diện hoặc lớp cơ sở. Lớp học sử dụng một lớp đặc biệt để đọc từ một trình cắm thêm đã được đăng ký tập trung, và nó sử dụng một nhà máy để khởi tạo chúng. Ví dụ kinh điển về kiến ​​trúc dựa trên plugin là một trang web sử dụng khung Inversion of Control (IoC) và có sự tập trung tập trung vào một phần của trang web.conig ile.

**Máy trạng thái**

Máy trạng thái là một cách phổ biến để phá vỡ các tác vụ phức tạp thành các bước dễ quản lý để thực hiện một số hành vi hẹp. Ở dạng đơn giản nhất, một máy trạng thái là một câu lệnh switch. Trong một tuyên bố chuyển đổi, bạn có thể dễ dàng thêm một trường hợp thêm với mã liên quan. Thật dễ dàng và hiệu quả.

Bất cứ khi nào bạn có một khu vực mã mà có thể mở rộng để kết hợp nhiều nhiệm vụ và các trường hợp, gói nó trong một máy nhà nước, hoặc nói chung là trong một loại hộp đen, là một bước tiến tốt hướng tới e xtensibility.

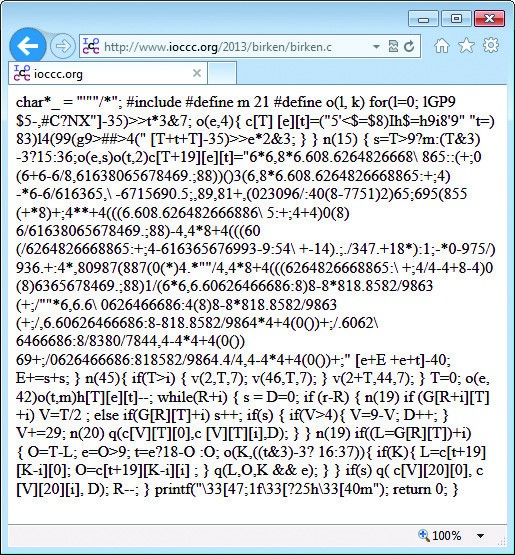
Khi bạn có một kiến ​​trúc dựa trên plugin, một vấn đề bạn đôi khi gặp phải là chia sẻ trạng thái của ứng dụng với các plugin. Vấn đề có thể không hiển thị nếu tất cả các plugin chỉ hoạt động với dữ liệu được truyền qua các phương thức tại các điểm mở rộng đã cho. Nếu cách tiếp cận này không hoạt động, bạn nên cân nhắc tập trung mô hình dữ liệu cho hệ thống để tất cả các plugin có quyền truy cập bình đẳng vào dữ liệu.

**Viết mã mà người khác có thể đọc**

Cả hai chúng tôi đều xem mái tóc còn lại của chúng tôi trở nên xám, có nghĩa là chúng tôi bắt đầu viết mã vào thời điểm những người bình thường xem xét các nhà phát triển là những con người kỳ quặc-một nửa nhà khoa học và một nửa nhà ảo thuật. Tất cả chúng ta đều là những người tiên phong cách đây 20 năm và chúng tôi thường làm một điểm - thường chỉ là một điểm hài hước - giấu mã của chúng tôi từ những người khác. Không phải vì chúng tôi không muốn người khác đọc mã của chúng tôi, nhưng chỉ để chứng minh chúng tôi thông minh hơn. Vì vậy, nó đã được coi là thông minh để viết mã theo cách mà người khác không thể đọc và hiểu được.

Thái độ này có thể không bao giờ thay đổi nhiều theo thời gian nếu một trong những vấn đề then chốt mà các kiến ​​trúc sư phần mềm phải đối mặt khi làm việc trong một nhóm đảm bảo rằng mã này là dễ hiểu và được viết theo các quy ước chung.

Bạn có nhớ Cuộc thi C Code Obfuscated C Code Contest (IOCCC)? Nếu không, chúng tôi khuyên bạn nên truy cập vào http://www.ioccc.org. IOCCC là một cuộc thi mở diễn ra mỗi năm. Những người tham gia gửi một số mã C làm việc để giải quyết vấn đề - bất kỳ vấn đề nào. "Thỏa thuận là gì?" Bạn có thể hỏi. Vâng, điều buồn cười là người chiến thắng không được chọn vì một thuật toán sáng tạo, hiệu suất, compactness, hoặc các chỉ số phổ biến khác. Các chỉ số thực sự quan trọng là làm thế nào che khuất và mơ hồ mã C là. Bạn có thể ind mã nguồn của chương trình chiến thắng của năm qua tại http://www.ioccc.org/years.html. Hình 4-6 cho thấy danh sách của một chương trình như vậy.



Hình 4-6 Mã nguồn của chương trình tham gia cuộc thi IOCCC.

Mã được đảm bảo hoạt động tốt, nhưng khá khó để hiểu ý nghĩa của nó.

Vượt ra ngoài niềm vui của một cuộc thi C obfuscated, dễ đọc trong lập trình là rất quan trọng. Nếu mã được kiểm tra là hoàn hảo và sẽ không được xúc động nữa vào bất kỳ lúc nào, bạn có thể chấp nhận rằng nó được viết bằng một cách tồi tệ. Trong bất kỳ trường hợp nào khác, các nhà phát triển khác (và thường là nhà phát triển ban đầu đã viết nó) có thể khó có thể hiểu được mã bởi vì nó không được viết bằng những thực tiễn tốt.

Chúng tôi đã nói và nghe điều này một ngàn tỷ lần; Chúng tôi biết nó. Tuy nhiên, tính dễ đọc thường bị lãng quên - có lẽ không phải lúc nào cũng hoàn toàn bị bỏ rơi, mà chỉ lãng phí đủ để dẫn tới một số lãng phí thời gian (và tiền bạc) khi mã hiện tại cần phải được làm lại hoặc người mới cần được đào tạo về nội bộ của hệ thống Được xây dựng.

**Tính dễ đọc như một thuộc tính phần mềm**

Khả năng bảo trì được biết đến là một trong những thuộc tính quan trọng của phần mềm, ít nhất là theo tài liệu ISO / IEC 9126. Để biết thêm thông tin, hãy tham khảo http://en.wikipedia.org/wiki/ISO/IEC\_9126.

Tuy nhiên, khả năng duy trì được là do các yếu tố khác nhau, một trong số đó chắc chắn là dễ đọc. Mã khó đọc cũng khó hiểu cho mọi người, kể cả tác giả. Người bạn của chúng tôi, Kevlin Henney, đã bao gồm 140 ký tự của một tweet một mảng thông minh về khả năng đọc mã. Kevlin tweeted rằng một sai lầm phổ biến là giả sử rằng các tác giả của mã không thể hiểu được bằng cách nào đó sẽ có thể thể hiện rõ ràng và rõ ràng trong ý kiến, như là một cách để giải thể bất kỳ thiếu rõ ràng về mã.

Chúng tôi không thể đồng ý nhiều hơn. Nhận xét là một phần của mã, nhưng đó không phải là nhận xét làm tăng đáng kể khả năng đọc của một đoạn mã. Nhận xét giống như đóng băng trên bánh; Ý kiến ​​không phải là bánh. Hơn nữa, ý kiến ​​không phải là loại bánh kem bạn thích có trên bánh.

**Hướng đến một sự khôn ngoan có thể đọc được**

Nếu bạn không hoàn toàn tin tưởng về tầm quan trọng của việc có thể đọc được mã, hãy xem xét rằng bất kỳ nhà phát triển nào đặt tay vào mã mà họ không hiểu rõ có thể làm cho mã thậm chí còn tồi tệ hơn. Nó không chỉ đơn giản là vấn đề nhằm đạt được chất lượng cao nhất; Đó chỉ là vấn đề tiết kiệm tiền.

Thật không may, khả năng đọc là một vấn đề rất chủ quan và sắp xếp một số công cụ tự động để kiểm tra và báo cáo mức độ dễ đọc của mã là gần như không thể. Dù sao, chúng tôi sẽ đặt cược ngôi nhà rằng ngay cả trong kịch bản không tưởng của việc có công cụ đo lường khả năng đọc tự động có sẵn, hầu hết mọi người sẽ đơn giản coi chúng không đáng tin cậy.

Chúng tôi dám nói rằng khả năng đọc là một thuộc tính của mã mà các nhà phát triển nên học ngay từ đầu sự nghiệp của họ và phát triển và cải thiện theo thời gian. Giống như phong cách và thiết kế đẹp, không nên dành cho các chuyên gia, và quan trọng hơn là không bị trì hoãn khi bạn chỉ có đủ thời gian cho nó.

Là một nhà phát triển dày dạn hôm nay, bạn biết rất rõ rằng bạn sẽ không bao giờ có đủ thời gian để làm gì, chỉ cần viết mã. Đó là lý do tại sao các nhà phát triển tốt quản lý để viết mã có thể đọc được "theo mặc định".

**Một vấn đề tôn trọng**

Đưa bất kỳ nỗ lực nào vào việc giữ mã có thể đọc được chủ yếu là vấn đề tôn trọng các nhà phát triển khác trong nhóm. Như một người sử dụng StackOverlow đã đăng ở một số điểm, bạn nên luôn luôn viết mã của bạn như thể người kết thúc việc duy trì những gì bạn viết là một kẻ thái nhân cách bạo lực biết nơi bạn sinh sống. Ít gây ấn tượng hơn, nhằm mục đích làm cho mã của bạn có thể chỉnh sửa được bởi bất kỳ nhà phát triển nào trong khi bạn đang tuyến đường và đi bộ đường dài ở giữa thung lũng Death. Ngoài ra, bạn không nên quên rằng nhà phát triển kết thúc việc duy trì mã của bạn, một ngày nào đó, có thể chỉ là bạn.

Có một câu chuyện hay mà chúng ta có thể biết về điểm này.

Cách đây vài năm, Dino đã tìm kiếm nội dung trên một diện tích khôn khéo của Microsoft .NET Framework. Google và Bing kết hợp có thể chỉ trả lại một bài báo từ một số tạp chí trực tuyến từ xa. Bài báo không đủ để làm rõ những điểm tối tăm mà Dino đang phải đối mặt. Vì vậy, một chút nản lòng, Dino đã phàn nàn về tác giả của bài báo, người đã không làm điều đó tuyệt vời của một công việc. Nhưng tác giả, tốt, đó là anh ta.

Khi đọc mã của người khác, một vài điều thường làm bạn say đắm. Một khía cạnh làm cho việc đọc mã khó khăn là mục tiêu không rõ ràng của cấu trúc dữ liệu và thuật toán được sử dụng trong mã. Một là các nhà phát triển chiến lược áp dụng trong mã đôi khi không hiển nhiên đối với người đọc và không được chú ý thông qua nhận xét. Có vẻ như tác giả đang cố gắng đánh lừa bạn, như thể cô ấy đang tham gia vào cuộc thi mã ngớ ngẩn. Đây là những ý kiến ​​tốt được yêu cầu.

**Khả năng đọc cũng là về tiền bạc**

Hãy tưởng tượng các kịch bản sau đây. Bạn có một codebase lớn được viết bằng Java và cần phải chuyển nó sang .NET. Nếu nó nghe kỳ lạ ở irst, hãy xem xét rằng có ít nhất một vài tình huống thực tế, trong đó nó thực sự có thể xảy ra. Một là khi bạn cố gắng chuyển một ứng dụng Android gốc sang nền tảng Windows Phone. Một là khi bạn đang trong quá trình xây dựng một thư viện cho các nền tảng khác nhau. Giả sử bạn đối mặt với một số mã như sau:

/ / Tìm số nhỏ nhất trong một danh sách các số nguyên int mininList(params int[] numbers)

{ var min = Int32.MaxValue; for (var i = 0; i < numbers.length; i++) { int number = numbers[i]; if (number < min) min = number;

}

return min;

}

And then later on, in the same ile, you ind something like this: var number = mininList(x, y);

Là một lợi nhuận (nhưng không phải biên), hãy cân nhắc rằng không có nhà phát triển C # hiểu biết có thể sẽ viết mã tương tự. Trong C #, trên thực tế, bạn có thể dựa vào LINQ để giảm mã cho một hoặc hai dòng. Nhưng giả sử rằng việc có mã như vậy là chấp nhận được. Có nhiều điều cần nói.

Trước tiên, có tên của phương pháp. Quy ước được sử dụng ở đây là đáng tranh cãi: cái tên bỏ qua một động từ và sử dụng logic hỗn hợp hỗn hợp. Một cái gì đó như GetMinFromList có lẽ sẽ là một cái tên tốt hơn. Thứ hai, chúng ta đạt đến điểm tranh cãi nhất của tư nhân qualiier được sử dụng trong khai báo và sử dụng các phương pháp trong lớp học. Bởi vì nó được đánh dấu riêng, nên phương pháp này chỉ được sử dụng trong lớp; Nó thực sự gọi chỉ một lần và chỉ để trả lại tối thiểu của hai số nguyên. Ngay cả khi không sử dụng LINQ, một cuộc gọi đến Math.Min đã có thể làm các trick.

Khi bạn gặp phải mã tương tự, bạn không thể tránh dừng lại và tự hỏi tại sao mã đó ở đó. Nó không phải là quá nhiều mà bạn muốn đổ lỗi cho tác giả; Nó chỉ là bạn muốn chắc chắn rằng bạn không thiếu bất kỳ phần quan trọng của nó. Mã đại diện cho các mẩu mã có thể dùng lại được có thể được gọi từ nhiều nơi trong toàn bộ mã. Vì vậy, đánh dấu nó như là tư nhân là vô nghĩa. Tuy nhiên, nếu phương pháp được sử dụng chỉ một lần, tại sao đánh dấu nó như là công cộng?

Các nhà phát triển biết rất rõ sức mạnh của quy tắc YAGNI-bạn không cần nó - và có xu hướng không để lộ mã mà không cần thiết. Vì vậy, nó có thể là tác giả của mã xem các chức năng có thể tái sử dụng nhưng không phải vào thời điểm đó đã được viết. Đó là lý do tại sao chức năng đó được viết để dễ dàng chuyển thành chức năng trợ giúp có thể tái sử dụng nhưng được đánh dấu là riêng tư chỉ hiển thị trong lớp lưu trữ.

Cuối cùng, không phải là không có mã kết quả từ một số chiến lược tốt-deined. Tuy nhiên, chiến lược - nếu có - rất khó để tạo ra cho người đọc bên ngoài. Đây chính là loại tình huống mà một vài dòng ý kiến ​​sẽ giải thích rõ ràng động lực đằng sau quyết định.

Khi tôi gặp mã đó, nó đã cho tôi một chút thời gian để irst nhìn rất cẩn thận hơn vào nó. Sau đó, tôi cảm thấy cần phải liên lạc với đội để đảm bảo tôi đã không bị mất điểm ẩn. Sau đó, tôi đã thay đổi việc thực hiện để sử dụng một cuộc gọi LINQ đơn giản. Đã được khá một vài năm trước, nhưng tôi đã được thuê để cổng mã để. NET. Cuối cùng, công ty đã trả thêm một giờ cho thời gian của tôi.

Nếu bạn bỏ lỡ thêm ý kiến ​​thích hợp, bạn không phải là một công dân tốt trong thành phố của mã.

**Một số quy tắc thực tiễn để cải thiện tính dễ đọc**

Khả năng đọc mã là một trong những chủ đề có tầm quan trọng được công nhận rộng rãi, nhưng thật khó để chính thức hóa. Đồng thời, nếu không có một số chính thức hoá, khả năng đọc mã gần như là một khái niệm rỗng. Nói chung, một loại quy tắc của ba Cs có thể được deined để deine dễ đọc: ý kiến, nhất quán, và rõ ràng.

**Bình luận**

Các IDE hiện đại giúp bạn dễ dàng đánh giá ý kiến ​​thể chế như một dòng mô tả chức năng, các tham số và giá trị trả lại. Tất cả bạn cần làm là nghĩ đến văn bản để thêm và để cho IDE làm phần còn lại. Bạn nên xem xét các nhận xét này bắt buộc, ít nhất đối với bất kỳ đoạn mã công khai nào bạn có thể có, và nỗ lực để viết văn bản có ý nghĩa cho họ.

Banning ý kiến ​​rõ ràng là một điểm cơ bản trên con đường để cải thiện khả năng đọc. Rõ ràng ý kiến ​​chỉ cần thêm tiếng ồn và không có thông tin có liên quan. Bằng cách deinition, một bình luận là bất kỳ văn bản giải thích cho bất kỳ quyết định bạn thực hiện trong các mã mà không rõ ràng per se. Một bình luận tốt chỉ có thể là một nhận xét sâu sắc về một khía cạnh cụ thể của mã. Mọi thứ khác chỉ ồn ào và có khả năng gây hại.

**Tính nhất quán**

Thứ hai "C" trên con đường để dễ đọc được tính nhất quán. Mỗi nhóm phát triển cần phải có hướng dẫn để viết mã, và tốt nhất cho các nguyên tắc như vậy là toàn công ty. Vấn đề không phải là có những hướng dẫn tốt hay không tốt. Đặc điểm đơn giản là có và tôn trọng các nguyên tắc. Giá trị của việc mã hóa cùng một điều luôn theo cùng một cách là vô giá.

Giả sử trong một khoảnh khắc bạn đang viết một thư viện thực hiện thao tác chuỗi. Ở một số nơi trong thư viện này, bạn cần kiểm tra xem một chuỗi có chứa một chuỗi con đã cho hay không. Bạn làm điều đó như thế nào?

Trong .NET Framework, cũng như trong Java SDK, bạn có ít nhất hai cách để đạt được điều tương tự: sử dụng phương pháp Chứa hoặc IndexOf. Tuy nhiên, hai phương pháp này phục vụ các mục đích khá khác nhau. Phương thức chứa trả về một giá trị Boolean và chỉ cho bạn biết liệu chuỗi con có chứa trong một chuỗi nhất định hay không. Phương pháp IndexOf, mặt khác, trả về chỉ số 0 dựa trên nơi mà chuỗi tìm kiếm được đặt. Nếu không có chuỗi con, IndexOf sẽ trả về -1.

Từ một quan điểm hoàn toàn chức năng, do đó, Chứa và IndexOf có thể được sử dụng để đạt được các mục tiêu tương tự; Tuy nhiên, họ đưa ra một thông báo khác cho người đọc mã và thường bắt người đọc xem lần thứ hai đoạn mã để xem liệu có một lý do đặc biệt để sử dụng IndexOf thay vì chứa để kiểm tra một chuỗi con hay không. Một lần vượt qua thứ hai của việc đọc một dòng mã không phải là một vấn đề, nhưng khi điều này xảy ra trên toàn bộ codebase của hàng ngàn dòng mã, nó sẽ có tác động về thời gian và sau đó là chi phí-không phải mã có thể đọc cao .

Là một nhà phát triển, bạn nên nhằm mục đích viết code sạch sẽ trong thời gian irst và không hy vọng có đủ thời gian một ngày nào đó để làm sạch nó lên. Với tư cách là một trưởng nhóm, bạn nên thực thi nguyên tắc mã thông qua chính sách nhận phòng. Lý tưởng hơn, bạn không nên cho phép đăng ký bất kỳ mã nào không vượt qua bài kiểm tra thống nhất.

**Trong trẻo**

"C" thứ ba của khả năng đọc mã là rõ ràng. Mã là rõ ràng nếu bạn theo phong cách đó theo cách để làm cho nó đọc tốt và dễ dàng. Điều này bao gồm cả việc nhóm và làm tổ mã một cách thích hợp. Nói chung, IF báo cáo thêm rất nhiều tiếng ồn cho mã. Đôi khi các câu lệnh có điều kiện - một cột của các ngôn ngữ lập trình - không thể tránh được, nhưng giới hạn số lượng các câu lệnh IF giữ được làm tổ dưới sự kiểm soát và làm cho mã dễ đọc hơn. Đôi khi SWITCH rõ ràng hơn nhiều IFs. Các công cụ trợ giúp mã thường cung cấp để thay thế nhiều chi nhánh NẾN IF với một SWITCH đơn. Nó chỉ là một hoặc hai cú nhấp chuột: bạn thực sự không có lý do gì.

Một số nhiệm vụ có thể yêu cầu một vài dòng mã, và nó có thể không phù hợp để thực hiện tái cấu trúc "Giải pháp trích xuất" trên đó. Trong trường hợp này, nên giữ những dòng này trong các khối ngăn cách với dòng trống. Nó không thay đổi bản chất của sự vật; Nó chỉ làm cho mã một chút dễ dàng hơn để đọc.

Đây chỉ là hai ví dụ cơ bản của những điều bạn có thể làm để làm cho mã của bạn tốt hơn từ góc độ thẩm mỹ thuần túy. Cuối cùng, nếu bạn đang tìm kiếm cảm hứng về cách sắp xếp mã nguồn của mình, hãy xem một số dự án nguồn mở. Phần mềm nguồn mở được viết cho nhiều người đọc và hiểu nó. Nó có thể là một nguồn cảm hứng thực sự, và bạn có thể muốn trích xuất các nguyên tắc riêng của bạn từ đó.

**Short làm cho nó tốt hơn**

Bạn có bao giờ tự hỏi tại sao báo chí và tạp chí in văn bản của họ trong các cột? Vâng, có một lý do cho điều đó.

Lý do là những đường dài khiến mắt người ta khó đọc. Khi nói đến mã, bạn nên áp dụng nguyên tắc tương tự và giới hạn cả chiều dài ngang của đường và cuộn theo chiều dọc của các phương pháp.

Nhưng chiều dài lý tưởng của cơ thể của một phương pháp là gì?

Số tuyệt đối mà làm việc cho mỗi nhà phát triển và mọi dự án gần như không thể ind. Nói chung, chúng tôi nói rằng khi bạn đạt đến 30 dòng, chuông báo thức sẽ đổ chuông. Chiều dài này cho thấy đã đến lúc bạn cân nhắc tái cấu trúc. Nói chung, việc sử dụng hạn chế các thanh cuộn có xu hướng là một tuyên bố tốt về phong cách mã của bạn.

**Tóm lược**

Theo chúng tôi, chất lượng mã được đo dựa trên ba tham số: khả năng kiểm tra, khả năng mở rộng và khả năng đọc.

Khả năng kiểm tra đảm bảo ở vị trí irst rằng mã được khớp nối tốt trong các thành phần riêng biệt và tương tác, phần lớn, liên kết lỏng lẻo. Câu lệnh này xuất phát từ thực tế là bạn có thể viết các bài kiểm tra về mã của bạn. Nếu bạn có thể làm điều đó, mã của bạn là thành phần và mỗi thành phần có thể được kiểm tra một cách riêng biệt. Tìm lỗi, và đặc biệt là lỗi ixing, trở nên dễ dàng hơn. Khi đo chất lượng mã, bài kiểm tra đơn vị rất quan trọng. Nó không phải là quá nhiều vì mã vượt qua chúng, nhưng vì bạn có thể viết chúng dễ dàng.

Trong thời hiện đại, bất kỳ mã được viết để được mở rộng và cải tiến. Duy trì mã là cách duy nhất để viết code vào những ngày này. Điều này có nghĩa là việc tách các mối quan tâm, khớp nối lỏng, và các cơ chế được xây dựng sẵn để mở rộng và tiêm phụ thuộc là các khái niệm mà mọi nhà phát triển phải nắm vững và áp dụng. Là một nhà lãnh đạo nhóm, bạn nên nhằm mục đích để cải thiện tất cả các nhà phát triển trong nhóm của bạn về số đó.

Bất cứ ai nêu chủ đề về chất lượng mã được tiếp xúc với một phản đối phổ biến: viết code sạch sẽ là khó khăn và mất rất nhiều thời gian. Nói cách khác, mã sạch sẽ không phải là một hình thức phát triển bền vững cho nhiều nhóm. Dọn sạch mã dẫn đến mất năng suất.

Năng suất là quan trọng hơn cả, nhưng tập trung vào năng suất một mình quá tốn kém bởi vì nó có thể dẫn đến mã chất lượng thấp và tốn kém để duy trì. Và, nếu thật khó để duy trì, đâu là cơ hội?

**Hoàn thành với một nụ cười**

Xem http://www.murphys-laws.com để biết danh sách rộng rãi các luật và các hệ lụy liên quan đến máy tính (và phi máy tính). Dưới đây là một số ít bạn có thể thích:

■ Lập trình thực không bình luận mã của họ. Nếu rất khó để viết, thật khó hiểu.

■ 90% lỗi của một lập trình viên đến từ dữ liệu từ các lập trình viên khác.

■ Các lỗi phần mềm không thể phát hiện bởi bất kỳ ai ngoại trừ người dùng cuối