

 [chukimmuoi](#) / [Clean-Code---Tiang-Viet](#) Public

forked from [quocdinguyen8/Clean-Code---Tiang-Viet](#)

[Code](#)

[Pull requests](#)

[Actions](#)

[Projects](#)


[Security](#)


[Insights](#)


 master ▼

...

[Clean-Code---Tiang-Viet](#) / [Markdown](#) / [Chaper09.md](#)

 [chukimmuoi](#) Add Chaper09 - ver4.md History

 1 contributor

 329 lines (264 sloc) | 33.4 KB

...

Kiểm tra đơn vị



Nghề của chúng ta đã đi một chặng đường dài trong mười năm qua. Năm 1997, không ai nghe nói về **Phát triển theo hướng kiểm tra**. Đối với đại đa số nhà phát triển, các bài kiểm tra đơn vị là những đoạn mã ngắn được viết để đảm bảo chương trình “hoạt động”. Chúng tôi sẽ cẩn thận viết các lớp và phương thức của mình rồi sau đó tạo ra một số mã đặc biệt để kiểm tra chúng. Thông thường, điều này sẽ liên quan đến một số loại chương trình điều khiển đơn giản cho phép tương tác thủ công với chương trình đã viết.

Tôi nhớ mình đã viết một chương trình C++ cho một hệ thống nhúng thời gian thực vào giữa những năm 90. Chương trình là một bộ đếm thời gian đơn giản với chữ ký sau:

```
void Timer::ScheduleCommand(Command* theCommand, int milliseconds)
```

Ý tưởng rất đơn giản; phương thức **thực thi** của **Command** sẽ được thực thi trong một luồng mới sau số mili giây được chỉ định. Vấn đề là, làm thế nào để kiểm tra nó. Tôi đã viết một chương trình đơn giản để lắng nghe bàn phím. Mỗi khi một ký tự được nhập, nó sẽ lên lịch một lệnh sẽ nhập ký tự đó vào năm giây sau. Rồi tôi gõ một giai điệu nhịp nhàng trên bàn phím và đợi nó xuất hiện trên màn hình sau năm giây.

“I . . . want-a-girl . . . just . . . like-the-girl-who-marr . . . ied . . . dear . . . old . . . dad.”

Tôi thực sự đã hát giai điệu đó khi gõ dấu "." và sau đó tôi hát lại khi các dấu chấm xuất hiện trên màn hình.

Đó là thử nghiệm của tôi! Khi tôi thấy nó hoạt động và chứng minh nó cho các đồng nghiệp của mình, tôi đã xóa mã thử nghiệm đi.

Như tôi đã nói, nghề của chúng ta đã trải qua một chặng đường dài. Ngày nay, tôi sẽ viết một bài kiểm tra để đảm bảo rằng mọi góc ngách của đoạn mã đó đều hoạt động như tôi mong đợi. Tôi sẽ tách mã của mình khỏi hệ điều hành thay vì chỉ gọi các hàm định thời gian tiêu chuẩn. Tôi sẽ mô phỏng các chức năng thời gian đó để tôi kiểm soát tuyệt đối thời gian. Tôi sẽ lên lịch các lệnh đặt cờ boolean, và sau đó tôi sẽ thay đổi thời gian đến tương lai, xem các cờ đó và đảm bảo rằng chúng chuyển từ sai thành đúng khi tôi thay đổi thời gian thành giá trị phù hợp.

Khi tôi có một bài kiểm tra để vượt qua, tôi sẽ đảm bảo rằng những bài kiểm tra đó thuận tiện để chạy cho bất kỳ ai khác cần làm việc với mã. Tôi sẽ đảm bảo rằng các bài kiểm tra và mã đã được kiểm tra cùng nhau vào cùng một gói nguồn.

Vâng, chúng ta đã đi một chặng đường dài; nhưng chúng ta phải đi xa hơn. Các động thái Agile và TDD(Test-Driven Development - Hướng phát triển thử nghiệm) đã khuyến khích nhiều lập trình viên viết các bài kiểm tra đơn vị tự động và nhiều hơn nữa đang gia nhập hàng ngũ này mỗi ngày. Nhưng trong cơn sốt điên cuồng muốn thêm kiểm tra vào kỷ luật của chúng tôi, nhiều lập trình viên đã bỏ lỡ một số điểm tinh tế và quan trọng hơn của việc viết bài kiểm tra tốt.

Ba định luật TDD(Test-Driven Development - Hướng phát triển thử nghiệm)

Bây giờ mọi người đều biết rằng TDD(Test-Driven Development - Hướng phát triển thử nghiệm) yêu cầu chúng ta viết các bài kiểm tra đơn vị trước, trước khi chúng ta viết mã sản xuất. Nhưng quy luật đó chỉ là phần nổi của tảng băng chìm. Hãy xem xét ba luật sau:

- **Luật thứ nhất:** Bạn không được viết mã sản xuất cho đến khi bạn viết một bài kiểm tra đơn vị không đạt.
- **Luật thứ hai:** Bạn không được viết nhiều bài kiểm tra đơn vị hơn mức đủ để không đạt, và không biên dịch sẽ không thành công.
- **Luật thứ ba:** Bạn không được viết nhiều mã sản xuất hơn mức đủ để vượt qua bài kiểm tra hiện đang không đạt.

Ba luật này ép bạn vào một chu trình có lẽ dài ba mươi giây. Các bài kiểm tra và mã sản xuất được viết cùng nhau, với việc các bài kiểm tra viết trước mã sản xuất vài giây.

Nếu chúng ta làm việc theo cách này, chúng ta sẽ viết hàng chục bài kiểm tra mỗi ngày, hàng trăm bài kiểm tra mỗi tháng và hàng nghìn bài kiểm tra mỗi năm. Nếu chúng tôi làm việc theo cách này, những bài kiểm tra đó sẽ bao gồm hầu như tất cả các mã sản xuất của chúng tôi. Phần lớn các thử nghiệm đó, có thể sánh ngang với kích thước của chính mã sản xuất, có thể đưa ra một vấn đề khó khăn cho quản lý .

Giữ sạch các bài kiểm tra

Vài năm trước, tôi được yêu cầu đào tạo một nhóm đã quyết định rõ ràng rằng mã thử nghiệm **không nên** được duy trì theo cùng tiêu chuẩn chất lượng như mã sản xuất. Họ đã cho phép phá vỡ các quy tắc trong các bài kiểm tra đơn vị. "Nhanh chóng và bẩn thỉu" là từ khóa. Các biến không cần phải được đặt tên tốt, các hàm kiểm không cần phải ngắn gọn và mang tính mô tả. Mã thử nghiệm không cần phải được thiết kế tốt và phân vùng cẩn thận. Miễn là mã thử nghiệm hoạt động và miễn là nó bao gồm mã dự án, thì nó đã đủ tốt.

Một số bạn đọc đến đây có thể thông cảm với quyết định đó. Có lẽ, rất lâu trong quá khứ, bạn đã viết các bài kiểm tra kiểu như tôi đã viết cho lớp **Timer**. Đó là một bước quan trọng từ việc viết bài kiểm tra rồi xoá đi đến việc viết một bộ bài kiểm tra đơn vị tự động. Vì vậy, giống như đội mà tôi đang huấn luyện, bạn có thể quyết định rằng có những bài kiểm tra bẩn sẽ tốt hơn là không có bài kiểm tra nào.

Điều mà nhóm này không nhận ra là việc có các bài kiểm tra bẩn tương đương với việc không có bài kiểm tra nào. Vấn đề là các bài kiểm tra phải thay đổi khi mã sản xuất phát triển. Các bài kiểm tra càng bẩn, chúng càng khó thay đổi. Mã kiểm tra càng rối, bạn càng có nguy cơ dành nhiều thời gian để nhồi nhét, viết các bài kiểm tra mới hơn là viết mã sản xuất mới. Khi bạn sửa đổi mã sản xuất, các bài kiểm tra cũ bắt đầu không thành công và sự lộn xộn trong mã kiểm tra khiến bạn khó có thể vượt qua các bài kiểm tra đó. Vì vậy, trách nhiệm cho các bài kiểm tra ngày càng tăng.

Từ khi phát hành, chi phí duy trì bộ thử nghiệm của nhóm tôi đã tăng lên. Cuối cùng nó đã trở thành vấn đề lớn nhất cho các nhà phát triển. Khi các nhà quản lý hỏi tại sao lại cần thời gian lớn như vậy, các nhà phát triển đã đổ lỗi cho các bài kiểm tra. Cuối cùng, họ buộc phải loại bỏ hoàn toàn bộ thử nghiệm.

Tuy nhiên, nếu không có các bài kiểm tra, họ sẽ mất khả năng đảm bảo rằng các thay đổi đối với mã của họ hoạt động như mong muốn. Nếu không có bài kiểm tra, họ không thể đảm bảo rằng các thay đổi đối với một phần trong hệ thống không ảnh hưởng đến các phần khác. Vì vậy, tỷ lệ sai sót bắt đầu tăng lên. Khi số lượng lỗi phát sinh ngoài ý muốn tăng lên, họ bắt đầu sợ phải thay đổi. Họ ngừng làm sạch mã sản xuất của mình vì sợ những thay đổi sẽ gây hại nhiều hơn là có lợi. Mã sản xuất bắt đầu không được bảo trì. Cuối cùng, không có thử nghiệm nào, mã sản xuất rối rắm và nhiều lỗi, khách hàng thất vọng và cảm giác mọi nỗ lực đổ sông đổ bể.

Theo một cách nào đó họ đã đúng. Nỗ lực kiểm tra đã thất bại. Nhưng quyết định để cho phép các bài kiểm tra lộn xộn là nguyên nhân của sự thất bại đó. Nếu giữ các bài kiểm tra của mình rõ ràng, tường minh, nỗ lực kiểm tra sẽ không thất bại. Tôi có thể nói điều này một cách chắc chắn bởi vì tôi đã tham gia và huấn luyện rất nhiều đội đã thành công với các bài kiểm tra đơn vị trong sạch.

Đạo lý của câu chuyện rất đơn giản: **Mã kiểm tra cũng quan trọng như mã sản xuất**. Nó không phải là một công dân hạng hai. Nó đòi hỏi sự suy nghĩ, thiết kế và cẩn thận. Nó phải được giữ sạch sẽ như mã sản xuất.

Kiểm tra Kích hoạt khả năng

Nếu không giữ sạch các bài kiểm tra của mình, bạn sẽ xóa chúng. Và nếu không có chúng, bạn sẽ mất đi thứ giữ cho mã sản xuất linh hoạt. Vâng, bạn đã đọc đúng điều đó. Đó là các bài kiểm tra đơn vị giữ cho mã của chúng ta linh hoạt, có thể bảo trì và có thể tái sử dụng. Lý do rất đơn giản. Nếu có các bài kiểm tra, bạn không sợ phải thay đổi mã! Nếu không có bài kiểm tra, mọi thay đổi đều có thể gây ra lỗi. Cho dù kiến trúc của bạn có linh hoạt đến đâu, cho dù thiết kế của bạn được phân vùng độc đáo như thế nào, nếu không có các thử nghiệm, bạn sẽ miễn cưỡng thực hiện các thay đổi vì sợ rằng sẽ tạo ra các lỗi không được phát hiện.

Nhưng với những bài kiểm tra nổi sợ hãi hầu như biến mất. Mức độ bao phủ của các bài kiểm tra càng cao, thì sự lo lắng càng ít. Bạn có thể thực hiện các thay đổi mà gần như không cần lo lắng đến kiến trúc, thiết kế và phạm vi. Thật vậy, bạn có thể cải thiện kiến trúc và thiết kế đó mà không sợ hãi!

Vì vậy, có một bộ kiểm tra đơn vị tự động bao gồm mã sản xuất là chìa khóa để giữ cho thiết kế và kiến trúc của bạn sạch sẽ nhất có thể. Kiểm tra cho phép tất cả các khả năng, bởi vì kiểm tra cho phép **thay đổi**.

Vì vậy, nếu các bài kiểm tra bị bẩn, thì khả năng thay đổi mã bị cản trở và bạn bắt đầu mất khả năng cải thiện cấu trúc của mã đó. Các bài kiểm tra càng bẩn, mã càng trở nên bẩn hơn. Cuối cùng, bạn xóa các bài kiểm tra và mã của bạn không được bảo trì.

Kiểm tra sạch

Điều gì tạo nên một bài kiểm tra sạch? Ba thứ. Khả năng đọc, khả năng đọc và khả năng đọc. Khả năng đọc có lẽ còn quan trọng hơn trong các bài kiểm tra đơn vị so với trong mã sản xuất. Điều gì làm cho các bài kiểm tra có thể đọc được? Điều tương tự làm cho tất cả mã có thể đọc được: rõ ràng, đơn giản và mật độ của biểu thức. Trong một bài kiểm tra, bạn muốn biểu đạt nhiều với ít cách diễn đạt nhất có thể.

Hãy xem xét mã từ FitNesse trong Listing 9-1. Ba bài kiểm tra này khó hiểu và chắc chắn có thể được cải thiện. Đầu tiên, có một lượng lớn mã trùng lặp [G5] trong các lệnh gọi lặp đi lặp lại tới **addPage** và **assertSubString**. Quan trọng hơn, mã này chỉ được tải với các chi tiết can thiệp vào tính biểu đạt của bài kiểm tra.

Listing 9-1

SerializedPageResponderTest.java

```
public void testGetPageHieratchyAsXml() throws Exception {
    crawler.addPage(root, PathParser.parse("PageOne"));
    crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
    crawler.addPage(root, PathParser.parse("PageTwo"));

    request.setResource("root");
    request.addInput("type", "pages");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
        (SimpleResponse) responder.makeResponse(
            new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/xml", response.getContentType());
    assertSubString("<name>PageOne</name>", xml);
    assertSubString("<name>PageTwo</name>", xml);
    assertSubString("<name>ChildOne</name>", xml);
}

public void testGetPageHieratchyAsXmlDoesntContainSymbolicLinks() throws Exception {
    WikiPage pageOne = crawler.addPage(root, PathParser.parse("PageOne"));
    crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
    crawler.addPage(root, PathParser.parse("PageTwo"));

    PageData data = pageOne.getData();
    WikiPageProperties properties = data.getProperties();
    WikiPageProperty symLinks = properties.set(SymbolicPage.PROPERTY_NAME);
    symLinks.set("SymPage", "PageTwo");
    pageOne.commit(data);

    request.setResource("root");
```

```

request.addInput("type", "pages");
Responder responder = new SerializedPageResponder();
SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
String xml = response.getContent();

assertEquals("text/xml", response.getContentType());
assertSubString("<name>PageOne</name>", xml);
assertSubString("<name>PageTwo</name>", xml);
assertSubString("<name>ChildOne</name>", xml);
assertNotSubString("SymPage", xml);
}

public void testGetDataAsHtml() throws Exception {
    crawler.addPage(root, PathParser.parse("TestPageOne"), "test page");
    request.setResource("TestPageOne");
    request.addInput("type", "data");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
        (SimpleResponse) responder.makeResponse(
            new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/xml", response.getContentType());
    assertSubString("test page", xml);
    assertSubString("<Test", xml);
}

```

Ví dụ, hãy xem các cuộc gọi **PathParser**. Chúng chuyển đổi các chuỗi thành các **PagePath** được trình thu thập thông tin sử dụng. Sự chuyển đổi này hoàn toàn không liên quan đến thử nghiệm hiện tại và chỉ mục đích để làm xao nhãng ý định ban đầu. Các chi tiết xung quanh việc tạo ra **responder**, thu thập và tạo **response** cũng chỉ là để gây nhiễu thông tin. Sau đó, có một cách thực tế là URL yêu cầu được tạo từ một **resource** và một đối số. (Tôi đã giúp viết mã này, vì vậy tôi có thể thoải mái phê bình nó.)

Cuối cùng, mã này không được thiết kế để đọc. Người đọc bị ngập trong đống chi tiết cần phải hiểu trước khi tìm ra ý nghĩa thực sự của bài kiểm tra.

Bây giờ, hãy xem xét các bài kiểm tra được cải thiện trong Listing 9-2. Những bài kiểm tra này thực hiện những điều tương tự, nhưng chúng đã được cấu trúc lại thành một hình thức rõ ràng hơn và dễ giải thích hơn. **Listing 9-2 SerializedPageResponderTest.java (refactored)**

```

public void testGetPageHierarchyAsXml() throws Exception {
    makePages("PageOne", "PageOne.ChildOne", "PageTwo");

    submitRequest("root", "type:pages");

    assertResponseIsXML();
    assertResponseContains(
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>" );
}

public void testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception {
    WikiPage page = makePage("PageOne");
    makePages("PageOne.ChildOne", "PageTwo");

    addLinkTo(page, "PageTwo", "SymPage");

    submitRequest("root", "type:pages");

    assertResponseIsXML();
    assertResponseContains(
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>" );
    assertResponseDoesNotContain("SymPage");
}

public void testGetDataAsXml() throws Exception {
    makePageWithContent("TestPageOne", "test page");

    submitRequest("TestPageOne", "type:data");

    assertResponseIsXML();
    assertResponseContains("test page", "<Test>");
}

```

Mô hình XÂY DỰNG-VẬN HÀNH-KIỂM TRA được thể hiện rõ ràng bởi cấu trúc của bài kiểm tra này. Mỗi bài kiểm tra được chia thành ba phần rõ ràng. Phần đầu tiên xây dựng dữ liệu thử nghiệm, phần thứ hai hoạt động trên dữ liệu thử nghiệm đó và phần thứ ba kiểm tra hoạt động đó có mang lại kết quả mong đợi hay không.

Lưu ý rằng phần lớn các chi tiết gây nhiễu đã được loại bỏ. Các bài kiểm tra đi đúng vào vấn đề và chỉ sử dụng các kiểu dữ liệu và chức năng mà chúng thực sự cần. Bất cứ ai đọc các bài kiểm tra này sẽ có thể tìm ra những gì họ cần rất nhanh chóng, mà không bị đánh lừa hoặc bị choáng ngợp bởi các chi tiết không liên quan.

Ngôn ngữ kiểm tra miền cụ thể

Các bài kiểm tra trong Listing 9-2 chứng minh kỹ thuật xây dựng ngôn ngữ miền cụ thể cho các bài kiểm tra. Thay vì sử dụng các API mà các lập trình viên sử dụng để thao tác với hệ thống, chúng tôi xây dựng một tập hợp các chức năng và tiện ích sử dụng các API đó và làm cho các bài kiểm tra thuận tiện hơn để viết và dễ đọc hơn. Các chức năng và tiện ích này trở thành một API chuyên biệt được sử dụng bởi các bài kiểm tra. Chúng là một ngôn ngữ kiểm tra mà các lập trình viên sử dụng để giúp viết các bài kiểm tra và giúp những người đọc các bài kiểm tra đó sau này.

API thử nghiệm này không được thiết kế trước; thay vào đó, nó phát triển từ việc tiếp tục tái cấu trúc mã thử nghiệm đã trở nên quá bấn bởi chi tiết bị xáo trộn. Giống như bạn đã thấy tôi tái cấu trúc Listing 9-1 thành Listing 9-2, các nhà phát triển có kỷ luật cũng sẽ cấu trúc lại mã thử nghiệm của họ thành các dạng ngắn gọn và biểu đạt hơn.

Tiêu chuẩn kép

Theo một nghĩa nào đó, đội mà tôi đã đề cập ở đầu chương này đã làm đúng. Mã trong API thử nghiệm có một bộ tiêu chuẩn kỹ thuật khác với mã sản xuất. Nó vẫn phải đơn giản, ngắn gọn và diễn đạt, nhưng nó không cần phải hiệu quả như mã sản xuất. Xét cho cùng, nó chạy trong môi trường thử nghiệm, không phải môi trường sản xuất và hai môi trường đó có những nhu cầu rất khác nhau.

Hãy xem xét thử nghiệm trong Listing 9-3. Tôi đã viết bài kiểm tra này như một phần của hệ thống kiểm soát môi trường mà tôi đang tạo mẫu. Nếu không đi sâu vào chi tiết, bạn có thể biết rằng bài kiểm tra này kiểm tra xem báo động nhiệt độ thấp, lò sưởi và quạt gió đều được bật khi nhiệt độ “quá lạnh”.

Listing 9-3

EnvironmentControllerTest.java

```
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
}
```

Tất nhiên, có rất nhiều chi tiết ở đây. Ví dụ, chức năng **tic** đó nói về cái gì? Trên thực tế, tôi không muốn bạn lo lắng về điều đó khi đọc bài kiểm tra này. Tôi muốn bạn chỉ lo lắng về việc liệu bạn có đồng ý rằng trạng thái cuối của hệ thống có phù hợp với nhiệt độ “quá lạnh” hay không.

Lưu ý, khi bạn đọc bài kiểm tra, mắt bạn cần phải đảo qua lại giữa tên của trạng thái được kiểm tra và phán đoán trạng thái được kiểm tra. Bạn thấy **heaterState**, và sau đó mắt bạn liếc sang trái để thấy **assertTrue**. Bạn thấy **coolerState** và mắt bạn phải theo dõi bên trái để thấy **assertFalse**. Điều này là tẻ nhạt và không đáng tin cậy. Nó làm cho bài kiểm tra khó đọc.

Tôi đã cải thiện khả năng đọc của bài kiểm tra này rất nhiều bằng cách chuyển nó thành Listing 9-4.

Listing 9-4

EnvironmentControllerTest.java (refactored)

```
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
}
```

Tất nhiên tôi đã ẩn chi tiết của hàm **tic** bằng cách tạo một hàm **wayTooCold**. Nhưng điều cần lưu ý là chuỗi kỳ lạ trong **assertEquals**. Chữ hoa có nghĩa là “bật”, chữ thường có nghĩa là “tắt” và các chữ cái luôn theo thứ tự sau: {**heater, blower, cooler, hi-temp-alarm, lo-temp-alarm**}.

Mặc dù điều này gần như là vi phạm quy tắc về lập bản đồ tâm trí, nó có vẻ phù hợp trong trường hợp này. Chú ý, một khi bạn biết nghĩa, mắt bạn lướt qua chuỗi đó và bạn có thể nhanh chóng diễn giải kết quả. Đọc bài kiểm tra gần như trở thành một niềm vui. Chỉ cần xem qua Listing 9-5 và xem mức độ dễ hiểu của các thử nghiệm này.

Listing 9-5

EnvironmentControllerTest.java (bigger selection)

```
@Test
public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
    tooHot();
    assertEquals("hBChl", hw.getState());
}

@Test
public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
    tooCold();
}
```

```
        assertEquals("HBchl", hw.getState());
    }

    @Test
    public void turnOnHiTempAlarmAtThreshold() throws Exception {
        wayTooHot();
        assertEquals("hBCHl", hw.getState());
    }

    @Test
    public void turnOnLoTempAlarmAtThreshold() throws Exception {
        wayTooCold();
        assertEquals("HBchl", hw.getState());
    }
}
```

Hàm **getState** được hiển thị trong Listing 9-6. Lưu ý rằng đây không phải là mã hiệu quả. Để làm cho nó hiệu quả, có lẽ tôi nên sử dụng một StringBuffer.

Listing 9-6

MockControlHardware.java

```
public String getState() {
    String state = "";
    state += heater ? "H" : "h";
    state += blower ? "B" : "b";
    state += cooler ? "C" : "c";
    state += hiTempAlarm ? "H" : "h";
    state += loTempAlarm ? "L" : "l";
    return state;
}
```

StringBuffers hơi xấu. Ngay cả trong mã sản xuất, tôi sẽ tránh chúng nếu chi phí nhỏ; và bạn có thể tranh luận rằng chi phí của mã trong Listing 9-6 là rất nhỏ. Tuy nhiên, ứng dụng này rõ ràng là một hệ thống thời gian thực được nhúng, và có khả năng tài nguyên máy tính và bộ nhớ rất hạn chế. Tuy nhiên, môi trường thử nghiệm không có khả năng bị hạn chế.

Đó là bản chất của tiêu chuẩn kép. Có những điều bạn có thể không bao giờ làm trong môi trường sản xuất như trong môi trường thử nghiệm. Thông thường chúng liên quan đến các vấn đề về bộ nhớ hoặc hiệu suất CPU. Nhưng vấn đề sạch sẽ thì luôn được quan tâm.

Một xác nhận cho mỗi bài kiểm tra

Có một trường phái tư tưởng nói rằng mọi hàm kiểm tra trong một bài kiểm tra JUnit nên có một và chỉ một câu lệnh **assert**. Quy tắc này có vẻ hà khắc, nhưng lợi thế có thể được nhìn thấy trong Listing 9-5. Những thử nghiệm đó đưa ra một kết luận duy nhất nhanh chóng và dễ hiểu.

Nhưng còn Listing 9-2 thì sao? Có vẻ không hợp lý khi bằng cách nào đó chúng ta có thể dễ dàng hợp nhất xác nhận rằng đầu ra là XML và nó chứa các chuỗi con nhất định. Tuy nhiên, chúng ta có thể chia thử nghiệm thành hai thử nghiệm riêng biệt, mỗi thử nghiệm có **assertion** cụ thể của riêng nó, như được hiển thị trong Listing 9-7.

Listing 9-7

SerializedPageResponderTest.java (Single Assert)

```
public void testGetPageHierarchyAsXml() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");
    whenRequestIsIssued("root", "type:pages");
    thenResponseShouldBeXML();
}

public void testGetPageHierarchyHasRightTags() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");
    whenRequestIsIssued("root", "type:pages");
    thenResponseShouldContain(
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
    );
}
```

Lưu ý rằng tôi đã thay đổi tên của các hàm để sử dụng quy ước chung **trước - khi - thì**. Điều này làm cho các bài kiểm tra thậm chí còn dễ đọc hơn. Thật không may, việc chia nhỏ các bài kiểm tra dẫn đến rất nhiều mã trùng lặp.

Chúng ta có thể loại bỏ sự trùng lặp bằng cách sử dụng mẫu TEMPLATE METHOD và đưa các phần đã **given/when** vào lớp cơ sở, các **then** trong các dẫn xuất khác nhau. Hoặc chúng ta có thể tạo một lớp kiểm tra hoàn toàn riêng biệt và đặt các phần **given** và **when** vào trong chức năng **@Before** và các phần **when** trong mỗi hàm **@Test**. Nhưng điều này có vẻ như là quá nhiều cho một vấn đề nhỏ như vậy. Cuối cùng, tôi thích nhiều **asserts** trong Liệt kê 9-2 hơn.

Tôi nghĩ rằng quy tắc **assert** duy nhất là một hướng dẫn tốt. Tôi thường cố gắng tạo một ngôn ngữ thử nghiệm miền cụ thể hỗ trợ nó, như trong Listing 9-5. Nhưng tôi không ngại đặt nhiều hơn một **assert** trong một bài kiểm tra. Tôi nghĩ điều tốt nhất chúng ta có thể nói là số lượng **assert** trong một bài kiểm tra phải được tối thiểu.

Khái niệm đơn cho mỗi thử nghiệm

Có lẽ một quy tắc tốt là chỉ muốn kiểm tra một việc duy nhất trong mỗi hàm kiểm tra. Chúng ta không muốn các chức năng thử nghiệm dài dòng và chồng chéo nhau. Listing 9-8 là một ví dụ về một thử nghiệm như vậy. Bài kiểm tra này nên được chia thành ba bài kiểm tra độc lập vì nó kiểm tra ba thứ độc lập. Việc hợp nhất tất cả chúng lại với nhau thành cùng một chức năng buộc người đọc phải tìm ra lý do tại sao mỗi phần lại ở đó và điều gì đang được kiểm tra bởi phần đó.

Listing 9-8

```
/**
 * Miscellaneous tests for the addMonths() method. */
public void testAddMonths() {
    SerialDate d1 = SerialDate.createInstance(31, 5, 2004);

    SerialDate d2 = SerialDate.addMonths(1, d1);
    assertEquals(30, d2.getDayOfMonth());
    assertEquals(6, d2.getMonth());
    assertEquals(2004, d2.getYYYY());

    SerialDate d3 = SerialDate.addMonths(2, d1);
    assertEquals(31, d3.getDayOfMonth());
    assertEquals(7, d3.getMonth());
    assertEquals(2004, d3.getYYYY());

    SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
    assertEquals(30, d4.getDayOfMonth());
    assertEquals(7, d4.getMonth());
    assertEquals(2004, d4.getYYYY());
}
```

Ba chức năng kiểm tra có thể phải như thế này:

- Với ngày cuối cùng của tháng có 31 ngày (như tháng 5):
 - i. Khi bạn thêm một tháng, sao cho ngày cuối cùng của tháng đó là ngày 30 (như tháng 6), thì ngày đó sẽ là ngày 30 của tháng đó, không phải ngày 31.
 - ii. Khi bạn thêm hai tháng vào ngày đó, sao cho tháng cuối cùng có 31 ngày, thì ngày đó sẽ là ngày 31.
- Với ngày cuối cùng của tháng với 30 ngày trong đó (như tháng 6):
 - i. Khi bạn thêm một tháng sao cho ngày cuối cùng của tháng đó có 31 ngày, thì ngày đó sẽ là ngày 30 chứ không phải ngày 31.

Nói như thế này, bạn có thể thấy rằng có một quy tắc chung ẩn giữa các bài kiểm tra. Khi bạn tăng tháng, ngày không được lớn hơn ngày cuối cùng của tháng. Điều này ngụ ý rằng tăng tháng vào ngày 28 tháng 2 sẽ mang lại ngày 28 tháng 3. Bài kiểm tra đó bị thiếu và sẽ là một bài kiểm tra hữu ích để viết.

Vì vậy, không phải nhiều **asserts** trong mỗi phần của Listing 9-8 gây ra sự cố. Đúng hơn, thực tế là có nhiều hơn một khái niệm đang được thử nghiệm. Vì vậy, có lẽ quy tắc tốt nhất là bạn nên giảm thiểu số lượng **asserts** cho mỗi khái niệm và chỉ kiểm tra một **asserts** cho mỗi hàm kiểm tra.

F.I.R.S.T.

Kiểm tra sạch tuân theo năm quy tắc tạo thành từ viết tắt ở trên:

Fast Kiểm tra phải nhanh chóng. Nó nên chạy thật nhanh. Khi các bài kiểm tra chạy chậm, bạn sẽ không muốn chạy chúng thường xuyên. Nếu bạn không chạy chúng thường xuyên, bạn sẽ không phát hiện ra các vấn đề đủ sớm để khắc phục chúng một cách dễ dàng. Bạn sẽ không cảm thấy thoải mái khi xóa mã. Cuối cùng thì mã sẽ bắt đầu không được bảo trì.

Independent Các bài kiểm tra độc lập không nên phụ thuộc vào nhau. Một thử nghiệm không nên thiết lập các điều kiện cho thử nghiệm tiếp theo. Bạn sẽ có thể chạy từng bài kiểm tra một cách độc lập và chạy các bài kiểm tra theo bất kỳ thứ tự nào bạn muốn. Khi các bài kiểm tra phụ thuộc vào nhau, thì bài kiểm tra đầu tiên không thành công sẽ gây ra hàng loạt các thất bại phía dưới, làm cho việc chẩn đoán trở nên khó khăn và che giấu các lỗi phía dưới.

Repeatable Các thử nghiệm lặp lại nên được lặp lại trong mọi môi trường. Bạn sẽ có thể chạy các bài kiểm tra trong môi trường sản xuất, trong môi trường QA và trên máy tính xách tay của mình khi đang đi tàu về nhà mà không có mạng. Nếu các bài kiểm tra của bạn không thể lặp lại trong bất kỳ môi trường nào, thì bạn sẽ luôn có lý do tại sao chúng thất bại. Bạn cũng sẽ thấy mình không thể chạy các bài kiểm tra khi môi trường không có sẵn.

Self-Validating Tự xác thực Các bài kiểm tra phải có đầu ra boolean. Đạt hoặc không đạt. Bạn không cần phải đọc qua tệp nhật ký để biết liệu các bài kiểm tra có vượt qua hay không. Bạn không cần phải so sánh thủ công hai tệp văn bản khác nhau để xem liệu các bài kiểm tra có vượt qua hay không. Nếu các bài kiểm tra không tự xác thực thì việc thất bại có thể trở nên bị động và việc chạy các bài kiểm tra có thể yêu cầu đánh giá thủ công lãng phí thời gian.

Timely Đúng lúc Các bài kiểm tra cần được viết đúng thời hạn. Các bài kiểm tra đơn vị nên được viết ngay trước mã sản xuất khiến chúng vượt qua. Nếu bạn viết các bài kiểm tra sau mã sản xuất, thì bạn có thể thấy mã sản xuất khó kiểm tra. Bạn có thể quyết định rằng một số mã sản xuất quá khó để kiểm tra. Bạn không thể thiết kế mã sản xuất để có thể kiểm tra được.

Kết luận

Tôi nghĩ rằng cả một cuốn sách có thể được viết về các bài kiểm tra sạch. Các bài kiểm tra cũng quan trọng đối với một dự án như mã sản xuất. Có lẽ chúng thậm chí còn quan trọng hơn, bởi vì các bài kiểm tra bảo tồn và nâng cao tính linh hoạt, khả năng bảo trì và khả năng tái sử dụng của mã sản xuất. Vì vậy, hãy giữ sạch các bài kiểm tra của bạn. Làm cho chúng diễn đạt và ngắn gọn. Các API thử nghiệm phát minh hoạt động như một ngôn ngữ dành riêng cho miền giúp bạn viết các thử nghiệm.

Nếu bạn không quan tâm các bài kiểm tra, thì mã của bạn cũng sẽ không được để ý. Giữ các bài kiểm tra của bạn sạch sẽ.

Thư viện

[**RSpec**]: RSpec: Behavior Driven Development for Ruby Programmers, Aslak Hellesøy, David Chelmsky, Pragmatic Bookshelf, 2008.

[**GOF**]: Design Patterns: Elements of Reusable Object Oriented Software, Gamma et al., Addison-Wesley, 1996.