CLEAN CODE

A handbook of agile software craftsmanship

(Code sạch – Cẩm nang của lập trình viên)

CHƯƠNG 3

***

HÀM

Trong những buổi đầu của việc lập trình, chúng tôi soạn thảo các hệ thống câu lệnh và các chương trình con. Sau đó, trong thời đại của Fortran và PL/1, chúng tôi soạn thảo các hệ thống chương trình, chương trình con, và các hàm. Ngày nay, chỉ còn các hàm là tồn tại. Các hàm là những viên gạch xây dựng nên chương trình. Và chương này sẽ giúp bạn tạo nên những viên gạch chắc chắn cho chương trình của bạn.

Hãy xem xét code trong Listing 3-1. Thật khó để tìm thấy một hàm dài. Nhưng sau một lúc tìm kiếm, tôi đã thấy nó. Nó không chỉ dài, mà còn có code trùng lặp, nhiều thứ dư thừa, các kiểu dữ liệu và API lạ,… Xem bạn phải sử dụng bao nhiêu giác quan trong ba phút tới để hiểu được nó:

Listing 3-1

HtmlUtil.java (FitNesse 20070619)

```
public static String testableHtml(
PageData pageData,
boolean includeSuiteSetup)
throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup =
                    PageCrawlerImpl.getInheritedPage(
                            SuiteResponder.SUITE_SETUP_NAME, wikiPage);
            if (suiteSetup != null) {
                WikiPagePath pagePath =
                    suiteSetup.getPageCrawler().getFullPath(suiteSetup);
                String pagePathName = PathParser.render(pagePath);
                buffer.append("!include -setup .")
                    .append(pagePathName)
                    .append("\n");
            }
        }
        WikiPage setup =
            PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
        if (setup != null) {
            WikiPagePath setupPath =
                wikiPage.getPageCrawler().getFullPath(setup);
            String setupPathName = PathParser.render(setupPath);
            buffer.append("!include -setup .")
                .append(setupPathName)
                .append("\n");
```

Listing 3-1

HtmlUtil.java (FitNesse 20070619)

```
        }
    }
    buffer.append(pageData.getContent());
    if (pageData.hasAttribute("Test")) {
        WikiPage teardown =
        PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
        if (teardown != null) {
            WikiPagePath tearDownPath =
                wikiPage.getPageCrawler().getFullPath(teardown);
            String tearDownPathName = PathParser.render(tearDownPath);
            buffer.append("\n")
                .append("!include -teardown .")
                .append(tearDownPathName)
                .append("\n");
        }
        if (includeSuiteSetup) {
            WikiPage suiteTeardown =
                PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_TEARDOWN_NAME,
                wikiPage);
            if (suiteTeardown != null) {
                WikiPagePath pagePath =
                    suiteTeardown.getPageCrawler()
                        .getFullPath(suiteTeardown);
                String pagePathName = PathParser.render(pagePath);
                buffer.append("!include -teardown .")
                    .append(pagePathName)
                    .append("\n");
            }
        }
    }
    pageData.setContent(buffer.toString());
    return pageData.getHtml();
}
```

Bạn có hiểu hàm trên sau ba phút đọc không? Chắc chắn là không. Có quá nhiều thứ xãy ra với nhiều mức độ trừu tượng khác nhau. Các chuỗi kỳ lạ, các lời gọi hàm trộn lẫn cùng các câu lệnh `if` lồng nhau,…

Tuy nhiên, chỉ với một vài phép rút gọn đơn giản, đặt lại vài cái tên, và một chút tái cơ cấu lại hàm, tôi đã có thể nắm bắt được mục đích của hàm này trong chín dòng lệnh. Thử sức lại với nó trong ba phút tiếp theo nào:

Listing 3-2

HtmlUtil.java (refactored)

```
public static String renderPageWithSetupsAndTeardowns(
PageData pageData, boolean isSuite
) throws Exception {
    boolean isTestPage = pageData.hasAttribute("Test");
    if (isTestPage) {
        WikiPage testPage = pageData.getWikiPage();
        StringBuffer newPageContent = new StringBuffer();
        includeSetupPages(testPage, newPageContent, isSuite);
        newPageContent.append(pageData.getContent());
        includeTeardownPages(testPage, newPageContent, isSuite);
        pageData.setContent(newPageContent.toString());
    }
    return pageData.getHtml();
}
```

Trừ khi bạn là học viên của FitNesse, nếu không bạn sẽ không hiểu đầy đủ hàm này. Nhưng không sao, bạn có thể hiểu rằng hàm này thực hiện một số việc thiết lập và chia nhỏ trang, sao đó hiển thị chúng dưới dạng HTML. Nếu đã quen với JUnit, bạn có thể nhận ra hàm này thuộc về một framework nào đó. Và, dĩ nhiên, những gì tôi vừa nói là hoàn toàn chính xác. Việc dự đoán chức năng của hàm từ Listing 3-2 khá dễ dàng, nhưng trong Listing 3-1 điều đó gần như là không thể.

Vậy điều gì làm cho hàm trong Listing 3-2 dễ đọc và dễ hiểu? Bằng cách nào chúng ta có thể tạo nên một hàm thể hiện được chức năng của nó? Những đặc tính nào cho phép một người đọc bình thường hiểu được chương trình mà họ đang cùng làm việc?

Nhỏ!!!

Nguyên tắc đầu tiên của hàm là chúng phải nhỏ. Nguyên tắc thứ hai là chúng phải nhỏ hơn nữa. Đây không phải là một khẳng định mà tôi có thể chứng minh. Tôi không thể cung cấp bất kỳ tài liệu hay nghiên cứu nào khẳng định rằng hàm nhỏ là tốt hơn. Những gì tôi có thể nói với bạn là trong gần bốn thập kỷ, tôi đã viết các hàm với nhiều kích cỡ khác nhau. Tôi đã viết 3000 dòng lệnh ghê tởm, tôi đã

viết các hàm trong phạm vi từ 100 đến 300 dòng, và tôi đã viết các hàm dài từ 20 đến 30 dòng. Kinh nghiệm đã dạy tôi một điều quý giá rằng, các hàm nên rất nhỏ.

Vào những năm 80, chúng tôi cho rằng một hàm không nên lớn hơn một màn hình. Dĩ nhiên, chúng tôi nói điều đó khi các màn hình VT100 chỉ có 24 dòng cùng 80 cột, và 4 dòng đầu thì được dùng cho mục đích quản trị. Ngày nay, với một phông chữ thích hợp và một màn hình xịn, bạn có thể phủ đến 150 ký tự cho 100 dòng hoặc nhiều hơn trên một màn hình. Các dòng code không nên dài quá 150 ký tự. Các hàm không nên "chạm nóc" 100 dòng, và độ dài thích hợp nhất dành cho hàm là không quá 20 dòng lệnh.

Vậy thu gọn một hàm bằng cách nào? Năm 1999 tôi có đến thăm Kent Beck tại nhà của ông ở Oregon. Chúng tôi ngồi xuống và cùng nhau viết một số chương trình nhỏ. Ông ấy đã cho tôi xem một chương trình nhỏ được viết bằng Java/Swing mà ông ấy gọi là Sparkle (Tia Sáng). Nó tạo ra một hiệu ứng hình ảnh trên màn hình rất giống với cây đũa thần của các bà tiên đỡ đầu. Khi bạn di chuyển chuột, các tia sáng lấp lánh sẽ "nhỏ giọt" từ con trỏ chuột xuống đáy cửa sổ, cứ như bị lực hấp dẫn kéo xuống vậy. Khi Kent cho tôi xem mã nguồn, tôi đã bị ấn tượng bởi độ nhỏ gọn của các hàm […]. Mọi hàm trong chương trình này chỉ dài hai, ba hoặc bốn dòng. Mỗi hàm đều rõ ràng. Mỗi hàm kể một câu chuyện. Và mỗi hàm dẫn bạn đến hàm tiếp theo hấp dẫn hơn. Đó là cách hàm của bạn trở nên ngắn gọn.

Hàm của bạn sẽ ngắn như thế nào? Chúng thường phải ngắn hơn Listing 3-2! Thật vậy, Listing 3-2 thực sự nên được rút gọn thành Listing 3-3.

Listing 3-3

HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
PageData pageData, boolean isSuite) throws Exception {
    if (isTestPage(pageData))
        includeSetupAndTeardownPages(pageData, isSuite);
    return pageData.getHtml();
}
```

Các khối lệnh và thụt dòng

Điều này có nghĩa là các khối lệnh bên trong câu lệnh `if`, `else`, `while`,…phải dài một dòng. Và dòng đó nên là một lời gọi hàm. Điều này không chỉ giữ các hàm kèm theo nhỏ mà còn bổ sung thêm giá trị tài liệu cho code của bạn, vì các hàm được gọi có thể có một cái tên thể hiện được mục đích của nó.

Điều này cũng có nghĩa các hàm không nên được thiết kế lớn để chứa các cấu trúc lồng nhau. Do đó, lời gọi hàm không nên thụt lề quá mức hai. Điều này, dĩ nhiên là làm cho các hàm dễ đọc và dễ viết hơn.

Thực hiện MỘT việc

Rõ ràng là Listing 3-1 đang làm nhiều hơn một việc. Nó tạo bộ đệm, tìm nạp trang, tìm kiếm các trang được kế thừa, hiển thị đường dẫn, thêm chuỗi phức tạp và tạo HTML,…. Listing 3-1 bận rộn làm nhiều việc khác nhau. Mặt khác, Listing 3-3 làm một việc đơn giản. Nó tạo các thiết lập và hiển thị nội dung vào các trang thử nghiệm.

Lời khuyên dưới đây đã xuất hiện nhiều lần, dưới dạng này hoặc dạng khác trong hơn 30 năm qua:

> *"HÀM CHỈ NÊN THỰC HIỆN MỘT VIỆC. CHÚNG NÊN LÀM TỐT VIỆC ĐÓ, VÀ CHỈ LÀM DUY NHẤT VIỆC ĐÓ"*

Vấn đề là, chúng ta khó biết "một việc" ở đây là việc gì. Listing 3-3 có làm một việc không? Thật dễ để chỉ ra nó đang làm 3 việc:

1. Xác định đây có phải là trang thử nghiệm hay không
2. Nếu phải, nạp vào các cài đặt và tái thiết lập nó
3. Hiển thị trang bằng HTML

Vậy, cái gì đây? Hàm đang thực hiện một việc hay ba việc? […] Chúng ta có thể mô tả hàm bằng cách xem nó như một đoạn TO ngắn (Ngôn ngữ LOGO sử dụng từ khóa TO giống như cách Ruby và Python sử dụng def. Vì vậy, mọi hàm đều bắt đầu bằng từ TO. Điều này tạo nên một hiệu ứng thú vị trên các hàm được thiết kế):

TO RenderPageWithSetupsAndTeardowns (ĐỂ hiển thị trang với các cài đặt và tái nạp), chúng tôi kiểm tra xem trang có phải là trang thử nghiệm hay không và nếu có, chúng tôi sẽ đưa vào các cài đặt và tái thiết lập nó. Sau đó, chúng tôi sẽ hiển thị trang bằng HTML.

Nếu hàm thực hiện các chức năng thấp hơn tên của hàm, thì hàm đó vẫn đang làm một việc. Sau tất cả, lý do chúng tôi viết các hàm là để phân tích một khái niệm lớn thành các khái niệm nhỏ hơn (nói cách khác, là phân tích tên hàm thành các tên ở mức độ thấp hơn).

Rõ ràng là Listing 3-1 gồm nhiều chức năng với nhiều mức độ khác nhau, và hiển nhiên là nó đang làm nhiều hơn một việc. Ngay cả Listing 3-2 cũng có hai mức độ, và đã được chứng minh bằng cách thu gọn nó. Nhưng sẽ rất khó để rút gọn Listing 3-3. Chúng ta có thể trích xuất câu lệnh `if` thành một hàm có tên `includeSetupsAndTeardownsIfTestPage`, nhưng điều đó chỉ đơn giản là mang code đến nơi khác mà không thay đổi mức độ trừu tượng của nó.

Vì vậy, một cách khác để biết hàm đang làm nhiều hơn "một việc" là khi bạn có thể trích xuất một hàm khác từ nó, nhưng với một cái tên khác so với chức năng của nó ở trong hàm.

[…]

Mỗi hàm là một cấp độ trừu tượng

Để đảm bảo các hàm của chúng ta đang thực hiện "một việc", chúng ta cần chắc chắn rằng các câu lệnh trong hàm của chúng ta đều ở cùng cấp độ trừu tượng. Hãy xem cách Listing 3-1 vi phạm quy

tắc này. Có những khái niệm trong đó có mức trừu tượng rất cao, chẳng hạn như `getHtml();` những thứ khác ở mức trừu tượng trung gian, chẳng hạn như: `String pagePathName = PathParser.render (pagePath);` và những người khác có mức độ thấp đáng kể, chẳng hạn như: `.append("\n").`

Việc trộn lẫn các cấp độ trừu tượng với nhau trong một hàm sẽ luôn gây ra những hiểu lầm cho người đọc. […]

Đọc code từ trên xuống dưới: Nguyên tắc Stepdown

Chúng tôi muốn code được đọc tuần tự từ trên xuống. Chúng tôi muốn mọi hàm được theo sau bởi các hàm có cấp độ trừu tượng lớn hơn để chúng tôi có thể đọc chương trình. Và khi chúng tôi xem xét một danh sách các khai báo hàm, mức độ trừu tượng của chúng phải được giảm dần. Tôi gọi đó là nguyên tắc Stepdown (tạm dịch: nguyên tắc ruộng bậc thang).

Nói cách khác, chúng tôi muốn đọc chương trình như thể đọc một bài văn có nhiều đoạn. Mỗi phần mô tả một cấp độ trừu tượng hiện tại và liên kết tới các đoạn văn tiếp theo, với cấp độ trừu tượng tiếp theo.

> *To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.*
>
> *To include the setups, we include the suite setup if this is a suite, then we include the regular setup.*
>
> *To include the suite setup, we search the parent hierarchy for the "SuiteSetUp" page and add an include statement with the path of that page.*
>
> *To search the parent. . .*

Sự thật là rất khó để các lập trình viên học cách tuân theo nguyên tắc này và viết các hàm ở một mức độ trừu tượng duy nhất. Nhưng học thủ thuật này cũng rất quan trọng. Nó là chìa khóa để đảm bảo các hàm ngắn gọn và giữ cho các chúng làm "một việc". Làm cho code của bạn đọc như một đoạn văn là kỹ thuật hiệu quả để duy trì sự đồng nhất của các cấp trừu tượng.

[…]

Câu lệnh switch

Thật khó để tạo nên một câu lệnh `switch` nhỏ (và cả chuỗi lệnh `if/else`). Ngay cả câu lệnh `switch` chỉ có 2 trường hợp. Và cũng rất khó để tạo ra một câu lệnh `switch` mà chỉ làm "một việc". Bởi bản chất của chúng, các câu lệnh `switch` luôn thực hiện N việc. Rất tiếc là, không phải lúc nào chúng tôi cũng tránh được chúng, nhưng chúng tôi có thể đảm bảo rằng các câu lệnh `switch` được chôn giấu trong một lớp cơ sở và không bao giờ được lặp lại. Chúng tôi làm việc này, dĩ nhiên, bằng tính chất đa hình.

Xem xét Listing 3-4 dưới đây. Nó hiển thị hoạt động dựa vào loại nhân viên:

Listing 3-4

Payroll.java

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
        case COMMISSIONED:
            return calculateCommissionedPay(e);
        case HOURLY:
            return calculateHourlyPay(e);
        case SALARIED:
            return calculateSalariedPay(e);
        default:
            throw new InvalidEmployeeType(e.type);
    }
}
```

Nguyên tắc Đơn nhiệm: Mỗi lớp chỉ nên chịu trách nhiệm về một nhiệm vụ cụ thể nào đó mà thôi.

Nguyên tắc Đóng & mở: Chúng ta nên hạn chế việc chỉnh sửa bên trong một Class hoặc Module có sẵn, thay vào đó hãy xem xét mở rộng chúng.

Có một số vấn đề với hàm này. Đầu tiên, nó lớn, và khi có loại nhân viên mới được thêm vào, nó sẽ to ra. Thứ hai, rất rõ ràng, nó đang làm nhiều hơn "một việc". Thứ ba, nó vi phạm Nguyên tắc Đơn nhiệm (Single Responsibility Principle – SRP) vì có nhiều lý do để nó thay đổi. Thứ tư, nó vi phạm Nguyên tắc Đóng & mở (Open Closed Principle – OCP) vì nó phải thay đổi khi có loại nhân viên khác được thêm vào. Nhưng vấn đề tồi tệ nhất của hàm này là có vô hạn các hàm khác có cùng cấu trúc. Ví dụ, chúng ta có thể có:

```
isPayday(Employee e, Date date),
```

hoặc

```
deliverPay(Employee e, Money pay),
```

hoặc một loạt những hàm khác. Tất cả đều có cấu trúc giống nhau!

Giải pháp cho vấn đề này (xem Listing 3-5) là chôn câu lệnh `switch` trong một lớp cơ sở của ABSTRACT FACTORY (tìm hiểu ABSTRACT FACTORY tại: https://vi.wikipedia.org/wiki/Abstract_factory), và không bao giờ để người khác trông thấy nó. ABSTRACT FACTORY sẽ sử dụng câu lệnh `switch` để tạo ra các trường hợp thích hợp của các dẫn xuất của `Employee`,và các hàm khác như `calculatePay`, `isPayday`, và `deliverPay`, sẽ được gọi bằng tính đa hình thông qua *interface* của `Employee`.

Listing 3-5

Employee and Factory

```
public abstract class Employee {
    public abstract boolean isPayday();
    public abstract Money calculatePay();
    public abstract void deliverPay(Money pay);
}
/*...*/
public interface EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws
InvalidEmployeeType;
}
/*...*/
public class EmployeeFactoryImpl implements EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws
     InvalidEmployeeType {
        switch (r.type) {
            case COMMISSIONED:
                return new CommissionedEmployee(r) ;
            case HOURLY:
                return new HourlyEmployee(r);
            case SALARIED:
                return new SalariedEmploye(r);
            default:
                throw new InvalidEmployeeType(r.type);
        }
    }
}
```

Nguyên tắc chung của tôi dành cho các câu lệnh `switch` là chúng có thể được tha thứ nếu chúng chỉ xuất hiện một lần, được sử dụng để tạo các đối tượng đa hình, và được ẩn đằng sau bằng tính kế thừa để phần còn lại của hệ thống không nhìn thấy chúng. Tất nhiên, nguyên tắc cũng chỉ là nguyên tắc, và trong nhiều trường hợp tôi đã vi phạm một hoặc nhiều phần của nguyên tắc đó.

Dùng tên có tính mô tả

Trong Listing 3-7, tôi đã thay đổi tên hàm ví dụ từ `testableHtml` thành `SetupTeardownIncluder.render`. Đây là một tên tốt hơn nhiều vì nó mô tả được chức năng của hàm đó. Tôi cũng đã cung cấp cho từng phương thức riêng (private method) một tên mô tả như

`isTestable` hoặc `includeSetupAndTeardownPages`. Thật khó để xem thường giá trị của những cái tên tốt. Hãy nhớ đến nguyên tắc của Ward: *"Bạn biết bạn đang làm việc cùng code sạch là khi việc đọc code hóa ra yomost hơn những gì bạn mong đợi"*. Một nửa chặn đường để đạt được nguyên tắc đó là chọn được một cái tên "xịn" cho những hàm làm "một việc". Hàm càng nhỏ và càng cô đặc thì càng dễ chọn tên mô tả cho nó hơn.

Đừng ngại đặt tên dài. Một tên dài nhưng mô tả đầy đủ chức năng của hàm luôn tốt hơn những cái tên ngắn. Và một tên dài thì tốt hơn một ghi chú (comment) dài. Dùng một nguyên tắc đặt tên cho phép dễ đọc nhiều từ trong tên hàm, và những từ đó sẽ cho bạn biết hàm đó hoạt động ra sao.

Đừng ngại dành thời gian cho việc chọn tên. Thật vậy, bạn nên thử một số tên khác nhau và đọc lại code ngay sau đó. Các IDE hiện đại như Eclipse hay IntelliJ làm cho việc đổi tên trở nên dễ dàng hơn rất nhiều. Sử dụng một trong những IDE đó và thử đặt các tên khác nhau cho đến khi bạn tìm thấy một cái tên có tính mô tả.

Chọn một cái tên có tính mô tả tốt sẽ giúp bạn vẽ lại thiết kế của mô-đun đó vào não, và việc cải thiện nó sẽ đơn giản hơn. Nhưng điều đó không có nghĩa là bạn sẽ bất chấp tất cả để "săn" được một cái tên tốt hơn để thay thế tên hiện tại.

[…]

Đối số của hàm

Số lượng đối số lý tưởng cho một hàm là không (niladic), tiếp đến là một (monadic), sau đó là hai (dyadic). Nên tránh trường hợp ba đối số (triadic) nếu có thể. Các hàm có nhiều hơn ba đối số (polyadic) chỉ cần thiết trong các trường hợp đặc biệt, và sau đó nên hạn chế sử dụng chúng đến mức thấp nhất.

Các đối số có những vấn đề của nó. Nó làm mất nhiều khái niệm của chương trình khi xuất hiện. Đó là lý do tại sao tôi đã loại bỏ gần như tất cả chúng ở ví dụ trên. Hãy xem xét `StringBuffer` trong ví dụ: Chúng tôi có thể đưa nó vào lời gọi hàm để tạo đối số thay vì để nó làm một biến thể hiện thông thường, nhưng sau đó độc giả của chúng ta sẽ phải "tự thông não" mỗi khi họ nhìn thấy nó. Khi bạn đọc một câu chuyện được viết nên bởi mô-đun, `includeSetupPage()` sẽ dễ hiểu hơn `includeSetupPageInto(newPageContent)`. Đối số có mức trừu tượng khác tên hàm và buộc bạn phải để tâm đến nó, mặc dù nó không quá quan trọng ở thời điểm đó.

[...]

Các đối số đầu ra khó hiểu hơn các đối số đầu vào. Khi chúng ta đọc một hàm, chúng ta quen với ý tưởng thông tin đi vào hàm thông qua các đối số, và kết quả nhận được thông qua giá trị trả về. Chúng tôi thường không nghĩ rằng thông tin trả về được truyền qua các đối số. Vì vậy, các đối số đầu ra thường khiến chúng tôi bất ngờ.

Hàm có một đối số đầu vào sẽ là tốt nhì (tốt nhất vẫn là hàm không có đối số). `SetupTeardownIncluder.render(pageData)` khá dễ hiểu. Rõ ràng là chúng ta sẽ kết xuất (render) dữ liệu của đối tượng `pageData`.

Hình thức chung của hàm một đối số (monadic)

Có hai lý do phổ biến để bạn truyền một đối số vào hàm. Bạn có thể đặt một câu hỏi đúng - sai cho đối số đó, như `boolean fileExists("MyFile")`. Hoặc bạn có thể thao tác trên đối số đó, biến nó thành một thứ gì khác và trả lại nó. Ví dụ, `InputStream fileOpen("MyFile")` biến đổi một chuỗi tên tệp thành một giá trị `InputStream`. Người đọc thường chỉ mong đợi hai cách này khi nhìn vào hàm có một đối số. Bạn nên chọn tên hàm thấy được sự phân biệt rõ ràng và luôn sử dụng hai hình thức này trong cùng một ngữ cảnh.

Một dạng ít phổ biến hơn nhưng vẫn rất hữu ít dành cho hàm có một đối số, đó là các sự kiện (event). Các hàm dạng này có một đối số đầu vào nhưng không có đối số đầu ra. Toàn bộ chương trình được hiểu là để thông dịch các lời gọi hàm như một sự kiện, và sử dụng các đối số để thay đổi trạng thái của hệ thống, ví dụ, `void passwordAttemptFailedNtimes(int attempts)`. Hãy cẩn thận với các hàm kiểu này, nó phải cực rõ ràng để người đọc biết đây là một sự kiện, nhớ chọn tên và ngữ cảnh một cách cẩn thận.

Cố gắng tránh bất kỳ hàm một đối số nào không tuân theo các mẫu trên, ví dụ: `void includeSetupPageInto(StringBuffer pageText)`. Sử dụng một đối số đầu ra thay vì một giá trị trả về khá là khó hiểu. Nếu một hàm chuyển đổi đối số đầu vào của nó, kết quả của phép biến đổi nên xuất hiện dưới dạng giá trị trả về. Thật vậy, `StringBuffer transform(StringBuffer in)` là tốt hơn khi so với `void transform-(StringBuffer out)`. [...]

Đối số luận lý

"Việc chuyển một đối số boolean vào hàm là một cái gì đó rất khủng khiếp. Nó ngay lập tức chỉ ra hàm của bạn đang là nhiều hơn một việc. Một việc nó làm khi đối số đúng, và một việc được làm khi đối số sai". Tuy nhiên, không phải lúc nào việc này cũng tởm lợm như bạn nghĩ. Ở một số trường hợp, việc này là hoàn toàn bình thường.

Hàm có hai đối số (dyadic)

Hàm có hai đối số sẽ khó hiểu hơn hàm có một đối số. Ví dụ `writeField(name)` sẽ dễ hiểu hơn `writeField(output-Stream, name)`. Mặc dù ý nghĩa của cả hai đều như nhau, đều dễ hiểu khi lần đầu nhìn vào. Nhưng hàm thứ hai yêu cầu bạn phải dừng lại, cho đến khi bạn học được cách bỏ qua tham số đầu tiên. Và, dĩ nhiên, có một vấn đề khi bạn bỏ qua đoạn code nào đó, thì khả năng đoạn code đó chứa lỗi là rất cao.

Tất nhiên luôn có những lúc hai đối số sẽ hợp lý hơn một đối số. Ví dụ: `Point p = new Point(0,0);` là hoàn toàn hợp lý khi bạn đang code về tọa độ mặt phẳng. Chúng tôi sẽ cảm thấy bối rối khi thấy `new Point(0);` trong trường hợp này.

Ngay cả hàm dyadic rõ ràng như `assertEquals(expected, actual)` vẫn có vấn đề. Đã bao nhiêu lần bạn nhầm lẫn vị trí giữa `expected` và `actual`? Hai đối số không có thứ tự tự nhiên. Thứ tự `expected`, `actual` là một quy ước đòi hỏi bạn phải nhớ nó trong đầu.

Những hàm dyadic không phải là những con quỷ dữ, và chắc chắn bạn phải viết chúng. Tuy nhiên bạn nên lưu ý rằng bạn sẽ phải trả giá cho việc đó, và nên tận dụng tối đa những thủ thuật hay lợi thế có sẵn để chuyển chúng về thành dạng monadic. Ví dụ, bạn có thể làm cho phương thức `writeField` trở thành một thành viên của `outputStream` để bạn có thể dùng lệnh `outputStream.writeField(name)`.

Hàm ba đối số (triadic)

Hàm có ba đối số khó hiểu hơn nhiều so với hàm hai đối số. Các vấn đề về sắp xếp, tạm ngừng và bỏ qua tăng gấp đôi. Tôi đề nghị bạn cẩn thận trước khi tạo ra nó.

[…]

Đối số đối tượng

Khi một hàm có vẻ cần nhiều hơn hai hoặc ba đối số, có khả năng một số đối số đó phải được bao bọc thành một lớp riêng của chúng. Ví dụ, hãy xem xét sự khác biệt giữa hai khai báo sau đây:

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

Giảm số lượng các đối số bằng cách tạo ra các đối tượng có vẻ như gian lận, nhưng không phải. Khi các nhóm biến được chuyển đổi cùng nhau, như cách x và y ở ví dụ trên, chúng có khả năng là một phần của một khái niệm xứng đáng có tên riêng.

Danh sách đối số

Đôi khi, chúng tôi muốn chuyển một số lượng đối số vào một hàm. Hãy xem xét ví dụ về phương thức `String.format`:

```
String.format("%s worked %.2f hours.", name, hours);
```

Nếu tất cả các đối số được xử lý giống nhau, như ví dụ trên, thì tất cả chúng tương đương với một đối số kiểu List. Bởi lý do đó, String.format thực chất là một hàm có hai đối số. Thật vậy, việc khai báo `String.format` như ví dụ dưới đây rõ ràng là một hàm dyadic:

```
void monad(Integer... args);
void dyad(String name, Integer... args);
void triad(String name, int count, Integer... args);
```

Động từ và các từ khóa

Chọn tên tốt cho một hàm có thể góp phần giải thích ý định của hàm và mục đích của các đối số. Trong trường hợp hàm monadic, hàm và đối số nên tạo thành một cặp động từ/danh từ hợp lý. `write(name)` là một ví dụ hoàn hảo trong trường hợp này. Dù cái tên này là gì, nó cho chúng

ta biết nó được viết. Một cái tên tốt hơn có lẽ là `writeField(name)`, nó cho chúng ta biết rằng tên là một trường.

Cuối cùng là một ví dụ về dạng từ khóa của tên hàm. Bằng cách này, chúng tôi mã hóa tên của các đối số thành tên hàm. Ví dụ, `assertEquals` có thể được cải tiến thành `assertExpectedEqualsActual(expected, actual)`. Điều này làm giảm vấn đề về việc nhớ vị trí của các đối số.

Không có tác dụng phụ

Tác dụng phụ (hay hiệu ứng lề) là một sự lừa dối. Hàm của bạn được hy vọng sẽ làm một việc, nhưng nó cũng làm những việc khác mà bạn không thấy. Đôi khi nó bất ngờ làm thay đổi giá trị biến của lớp của nó. Hoặc nó sẽ biến chúng thành các tham số được truyền vào hàm, hoặc các hàm toàn cục. Trong cả hai trường hợp, chúng tạo ra các sai lầm và làm sai kết quả.

Hãy xem xét hàm trong ví dụ dưới đây. Hàm này sử dụng thuật toán để kiểm tra `userName` và `password`. Nó trả về true nếu chúng khớp và trả về false nếu có gì sai. Nhưng nó cũng có tác dụng phụ. Ban phát hiện ra nó chứ?

Listing 3-6

UserValidator.java

```
public class UserValidator {
    private Cryptographer cryptographer;
    public boolean checkPassword(String userName, String password) {
        User user = UserGateway.findByName(userName);
        if (user != User.NULL) {
            String codedPhrase = user.getPhraseEncodedByPassword();
            String phrase = cryptographer.decrypt(codedPhrase, password);
            if ("Valid Password".equals(phrase)) {
                Session.initialize();
                return true;
            }
        }
        return false;
    }
}
```

Tác dụng phụ ở đây là lời gọi hàm `Session.initialize()`. Hàm `checkPassword`, theo cách đặt tên của nó, nói rằng nó chỉ kiểm tra mật khẩu. Tên hàm không thông báo rằng nó khởi tạo session (Tìm hiểu thêm: https://en.wikipedia.org/wiki/Session_(computer_science)). Vậy nên khi ai đó dùng hàm này, họ tin rằng mình chỉ kiểm tra tính hợp lệ của người dùng mà không biết dữ liệu của session hiện tại có nguy cơ bị mất.

Tác dụng phụ này tạo ra một mắt xích về thời gian. Đó là, `checkPassword` chỉ được gọi vào những thời điểm nhất định (nói cách khác, khi nó an toàn để khởi tạo session). Nếu được gọi lung tung, dữ liệu của session có thể vô tình bị mất. Mắt xích tạm thời này gây khó hiểu, đặc biệt là nó lúc ẩn lúc hiện. Nếu bạn có một mắt xích như vậy, bạn nên làm nó hiện rõ trong tên hàm. Trong trường hợp này, chúng ta có thể đổi tên hàm thành `checkPasswordAndInitializeSession`, mặc dù chắc chắn hàm này vi phạm nguyên tắc "Làm một việc".

Đối số đầu ra

[…]

Nói chung chúng ta nên tránh các đối số đầu ra. Nếu hàm của bạn phải thay đổi trạng thái của một cái gì đó, hãy thay đổi trạng thái của đối tượng sở hữu nó.

Tách lệnh truy vấn

Hàm nên làm một cái gì đó hoặc trả lời một cái gì đó, nhưng không phải cả hai. Hoặc là hàm của bạn thay đổi trạng thái của một đối tượng, hoặc nó sẽ trả về một số thông tin về đối tượng đó. Làm cả hai thường gây nên sự nhầm lẫn. Xem xét hàm ví dụ sau:

```
public boolean set(String attribute, String value);
```

Hàm này đặt giá trị cho thuộc tính nếu thuộc tính đó tồn tại. Nó trả về true nếu thành công, và false nếu thất bại. Điều này dẫn đến các câu lệnh lẻ như sau:

```
if (set("username", "unclebob"))...
```

Hãy tưởng tượng điều này từ quan điểm của người đọc. Nó có nghĩa là gì? Nó hỏi thuộc tính "`username`" đã được đặt thành "`unclebob`" chưa? Hay nó hỏi thuộc tính "`username`" trước đó có giá trị là "`unclebob`"? Thật khó để suy ra ý nghĩa của hàm vì không rõ từ "`set`" là động từ hay tính từ.

Dự định của tác giả là đặt `set` trở thành một động từ, nhưng trong ngữ cảnh của câu lệnh `if`, nó mang đến cảm giác như một tính từ […]. Chúng tôi thử giải quyết vấn đề này bằng cách đổi tên hàm đã đặt thành `setAndCheckIfExists`, nhưng điều đó không giúp ích gì nhiều trong ngữ cảnh của câu lệnh `if`. Giải pháp thực sự là tách lệnh khỏi truy vấn sao cho sự nhầm lẫn không thể xảy ra.

```
if (attributeExists("username")) {
    setAttribute("username", "unclebob");
    ...
}
```

[...]

Prefer Exceptions to Returning Error Codes

[...]

Đừng lặp lại code của bạn

Xem xét lại Listing 3-1 một cách cẩn thận, bạn sẽ nhận thấy rằng có một thuật toán được lặp lại bốn lần. Mỗi lần cho mỗi trường hợp `SetUp`, `SuiteSetUp`, `TearDown`, và `SuiteTearDown`. Không dễ dàng để phát hiện ra sự trùng lặp này vì cả bốn trường hợp code được trộn lẫn với code khác, và sự sao chép là không thống nhất. Tuy nhiên, việc trùng lặp code như vậy là một vấn đề, vì nó làm code của bạn phình to ra và khi cần sửa đổi, bạn sẽ phải sửa đổi bốn lần. Điều đó cũng đồng nghĩa với việc nguy cơ xuất hiện lỗi là bốn lần.

Vấn đề này được khắc phục bằng phương thức `include` trong Listing 3-7. Đọc lại code một lần nữa và bạn sẽ thấy khả năng đọc của toàn bộ mô-đun được tăng lên chỉ bằng cách giảm sự trùng lặp đó.

Sự trùng lặp có lẽ là gốc rễ của mọi tội lỗi trong lập trình. Nhiều nguyên tắc và kinh nghiệm đã được tạo ra cho mục đích kiểm soát hoặc loại bỏ nó. Lập trình cấu trúc, lập trình hướng đối tượng (OOP), lập trình hướng khía cạnh (Aspect Oriented Programming – AOP), lập trình hướng thành phần (Component Oriented Programming – COP), tất cả chúng đều có chiến lược để loại bỏ code trùng lặp. Nó chứng minh rằng kể từ khi chương trình con được phát minh, các sáng kiến trong ngành công nghiệp phát triển phần mềm đều nhắm đến việc loại bỏ những đoạn code trùng lặp ra khỏi mã nguồn.

Lập trình có cấu trúc

Một số lập trình viên đi theo nguyên tắc lập trình có cấu trúc của Edsger Dijkstra. Dijkstra nói rằng mọi hàm, và mọi khối trong hàm nên có một lối vào và một lối thoát. Điều đó có nghĩa là chỉ nên có một lệnh `return` trong hàm, không có câu lệnh `break`, `continue` trong một vòng lặp; và không bao giờ dùng bất kỳ câu lệnh `goto` nào.

Chúng tôi thông cảm với các nguyên tắc và mục tiêu của lập trình cấu trúc, nhưng các nguyên tắc này chỉ mang lại một chút lợi ích khi các hàm bạn viết rất nhỏ. Ở các hàm lớn hơn, lợi ích mà nó mang lại thật sự là không đáng kể.

Vậy nên nếu bạn có thể tiếp tục giữ cho các hàm của mình nhỏ, thì việc sử dụng các câu lệnh `return`, `break` hay `continue` là vô hại và đôi khi nó còn giúp hàm của bạn rõ ràng hơn nguyên tắc một lối vào, một lối thoát. Mặt khác, lệnh `goto` chỉ có ý nghĩa trong các hàm lớn, vì vậy nên tránh sử dụng nó.

Tôi đã viết các hàm này như thế nào?

Viết phần mềm cũng giống như viết các thể loại khác. Khi bạn viết một bài báo hay một văn kiện, bạn sẽ suy nghĩ trước, sau đó bạn nhào nặn nó cho đến khi nó trở nên mạch lạc, trơn tru. Các bản thảo

ban đầu có thể vụng về và rời rạc, vì vậy bạn vứt nó vào sọt rác và tái cơ cấu nó, tinh chỉnh nó cho đến khi nó được đọc theo cách mà bạn muốn.

Khi tôi bắt đầu viết các hàm, chúng dài và phức tạp. Chúng có rất nhiều vòng lặp lồng nhau, chúng có hàng tá đối số. Các tên được đặt tùy ý, và tồn tại nhiều code trùng lặp. Nhưng tôi cũng có một bộ unit test để đảm bảo cho tất cả những dòng code vụng về đó.

Và sau đó, tôi thay đổi và tinh chỉnh lại code đó, tách ra thành các hàm, đặt lại tên và loại bỏ code trùng lặp. Tôi thu nhỏ phương thức và sắp xếp lại chúng. Đôi khi tôi *đập tan nát* một lớp, trong khi vẫn giữ lại các bài test đã hoàn thành.

Cuối cùng, các hàm tôi hoàn thành đã tuân theo các nguyên tắc tôi đặt ra trong chương này. Tôi không tuân theo các nguyên tắc tôi đặt ra để bắt đầu viết nó, điều đó là không thể.

Kết luận

Mỗi hệ thống được xây dựng từ một DSL được thiết kế và mô tả bởi các lập trình viên. Các hàm là một động từ, và các lớp là một danh từ […]. Nghệ thuật lập trình, dĩ nhiên, luôn là nghệ thuật sử dụng ngôn ngữ.

Các lập trình viên tài năng xem các hệ thống như những câu chuyện kể, chứ không phải là các chương trình được viết. Họ sử dụng khả năng của ngôn ngữ lập trình mà họ chọn để diễn đạt *câu chuyện* phong phú hơn và giàu cảm xúc hơn. Một phần của các DSL là cấu trúc phân cấp của các hàm mô tả hành động diễn ra trong hệ thống đó. Và các hàm được định nghĩa để nói lên câu chuyện của riêng mình.

Chương này đã chỉ cho bạn về cách viết tốt các hàm. Nếu bạn tuân thủ các nguyên tắc trên, các hàm của bạn sẽ ngắn gọn, được đặt tên và được tổ chức tốt. Nhưng đừng bao giờ quên rằng mục tiêu của bạn là kể một câu chuyện về hệ thống, và các hàm bạn viết cần ăn khớp với nhau một cách rõ ràng và chính xác để giúp bạn hoàn thành việc đó.

SetupTeardownIncluder

Listing 3-7

SetupTeardownIncluder.java

```
package fitnesse.html;
import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;
public class SetupTeardownIncluder {
    private PageData pageData;
    private boolean isSuite;
    private WikiPage testPage;
    private StringBuffer newPageContent;
    private PageCrawler pageCrawler;
```

Listing 3-7

SetupTeardownIncluder.java

```
    public static String render(PageData pageData) throws Exception {
        return render(pageData, false);
    }

    public static String render(PageData pageData, boolean isSuite)
    throws Exception {
        return new SetupTeardownIncluder(pageData).render(isSuite);
    }

    private SetupTeardownIncluder(PageData pageData) {
        this.pageData = pageData;
        testPage = pageData.getWikiPage();
        pageCrawler = testPage.getPageCrawler();
        newPageContent = new StringBuffer();
    }

    private String render(boolean isSuite) throws Exception {
        this.isSuite = isSuite;
        if (isTestPage())
            includeSetupAndTeardownPages();
        return pageData.getHtml();
    }

    private boolean isTestPage() throws Exception {
        return pageData.hasAttribute("Test");
    }

    private void includeSetupAndTeardownPages() throws Exception {
        includeSetupPages();
        includePageContent();
        includeTeardownPages();
        updatePageContent();
    }

    private void includeSetupPages() throws Exception {
        if (isSuite)
            includeSuiteSetupPage();
        includeSetupPage();
```

Listing 3-7

SetupTeardownIncluder.java

```
    }

    private void includeSuiteSetupPage() throws Exception {
        include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
    }

    private void includeSetupPage() throws Exception {
        include("SetUp", "-setup");
    }

    private void includePageContent() throws Exception {
        newPageContent.append(pageData.getContent());
    }

    private void includeTeardownPages() throws Exception {
        includeTeardownPage();
        if (isSuite)
            includeSuiteTeardownPage();
    }

    private void includeTeardownPage() throws Exception {
        include("TearDown", "-teardown");
    }

    private void includeSuiteTeardownPage() throws Exception {
        include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
    }

    private void updatePageContent() throws Exception {
        pageData.setContent(newPageContent.toString());
    }

    private void include(String pageName, String arg) throws Exception {
        WikiPage inheritedPage = findInheritedPage(pageName);
        if (inheritedPage != null) {
            String pagePathName = getPathNameForPage(inheritedPage);
            buildIncludeDirective(pagePathName, arg);
        }
```

Listing 3-7

SetupTeardownIncluder.java

```
    }

    private WikiPage findInheritedPage(String pageName) throws Exception
{
        return PageCrawlerImpl.getInheritedPage(pageName, testPage);
    }

    private String getPathNameForPage(WikiPage page) throws Exception {
        WikiPagePath pagePath = pageCrawler.getFullPath(page);
        return PathParser.render(pagePath);
    }

    private void buildIncludeDirective(String pagePathName, String arg)
{
        newPageContent
        .append("\n!include ")
        .append(arg)
        .append(" .")
        .append(pagePathName)
        .append("\n");
    }
}
```

Tham khảo

[KP78]: Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGrawHill, 1978.

[PPP02]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2002.

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

[PRAG]: *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

[SP72]: *Structured Programming*, O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Academic Press, London, 1972.