CLEAN CODE

A handbook of agile software craftsmanship

(Code sạch – Cẩm nang của lập trình viên)

CHƯƠNG 1

***

CODE SẠCH

Bạn đang đọc quyển sách này vì hai lý do. Thứ nhất, bạn là một lập trình viên. Thứ hai, bạn muốn trở thành một lập trình viên giỏi. Tuyệt vời! Chúng tôi cần lập trình viên giỏi.

Đây là một quyển sách nói về cách để bạn code tốt hơn, và nó chứa đầy code. Chúng ta sẽ xem xét code từ nhiều phương diện, từ trên xuống dưới, từ dưới lên trên, và từ trong ra ngoài. Khi xong việc, chúng ta sẽ được biết thêm rất nhiều về code. Hơn nữa, chúng ta sẽ nói về sự khác biệt giữa code "xịn" (good code) và code "rởm" (bad code), biết cách tạo nên code "xịn", và học cách hô biến code "rởm" thành code "xịn".

Sẽ (vẫn) có code

Nhiều người cho rằng việc viết code, sau vài năm nữa sẽ không còn là vấn đề, rằng chúng ta nên quan tâm đến những mô hình và các yêu cầu. Thực tế, một số người cho rằng việc viết code đang dần đến lúc phải kết thúc, code sẽ được tạo ra thay vì được viết hay gõ. Và các lập trình viên "gà mờ" sẽ phải tìm công việc khác vì khách hàng của họ có thể tạo nên một chương trình chỉ bằng cách nhập vào các thông số cần thiết…

Oh sh*t, rõ là vô lý! Code sẽ không bao giờ bị loại bỏ vì chúng đại diện cho nội dung của các yêu cầu của khách hàng. Ở một mức độ nào đó, những nội dung đó không thể bỏ qua hoặc trừu tượng hóa; chúng phải được thiết lập. Việc thiết lập các nội dung mà máy tính có thể hiểu và thi hành, được gọi là *lập trình*, và *lập trình* thì cần có *code*.

Tôi hy vọng mức độ trừu tượng của các ngôn ngữ lập trình và số lượng các DSL (Domain-Specific Language – ngôn ngữ chuyên biệt dành cho các vấn đề cụ thể) sẽ tăng lên. Đó là một dấu hiệu tốt. Nhưng dù điều đó xãy ra, nó vẫn không "đá đít" code. Các đặc điểm kỹ thuật được viết bằng những ngôn ngữ bậc cao và DSL vẫn là code. Nó vẫn cần sự nghiêm ngặt, chính xác, và theo đúng các nguyên tắc, tường tận đến nỗi một cỗ máy có thể hiểu và thực thi nó.

Những người nghĩ rằng việc viết code đang đi đến ngày tàn cũng giống như việc một nhà toán học hy vọng khám phá ra một thể loại toán học mới không chứa nguyên tắc, định lý hay bất kỳ công thức nào. Họ hy vọng một ngày nào đó, các lập trình viên sẽ tạo ra những cỗ máy có thể làm bất cứ điều gì họ muốn (chứ không cần ra lệnh bằng giọng nói). Những cỗ máy này phải có khả năng hiểu họ tốt đến nỗi, chúng có khả năng dịch những yêu cầu mơ hồ thành các chương trình hoàn hảo, đáp ứng chính xác những yêu cầu đó.

Dĩ nhiên, chuyện đó chỉ xãy ra trong phim viễn tưởng. Ngay cả con người, với tất cả các giác quan và sự sáng tạo, cũng không thể thành công trong việc hiểu những cảm xúc mơ hồ của người khác. Thật vậy, nếu được hỏi quá trình phân tích các yêu cầu của khách hàng đã dạy cho chúng tôi điều gì, thì câu trả lời là các yêu cầu được chỉ định rõ ràng, trông giống như code và có thể hoạt động trong quá trình kiểm tra.

Hãy nhớ một điều rằng code thực sự là một ngôn ngữ mà trong đó, công việc cuối cùng của chúng ta là thể hiện các yêu cầu. Chúng tôi có thể tạo ra các ngôn ngữ gần với các yêu cầu, hoặc tạo ra các công cụ cho phép phân tích cú pháp và lắp ráp chúng vào các chương trình. Nhưng chúng tôi sẽ không bao giờ bỏ qua các yêu cầu cần thiết – vì vậy, code sẽ luôn tồn tại.

Code tồi, Code "rởm"

Gần đây, tôi có đọc phần mở đầu của quyển *Implementation Patterns.1* của Kent Beck. Ông ấy nói rằng *"…cuốn sách này dựa trên một tiền đề khá mong manh: đó là vấn đề code sạch…"* Mong manh ư? Tôi không đồng ý chút nào. Tôi nghĩ tiền đề đó là một trong những tiền đề mạnh mẽ nhất, nhận được sự ủng hộ lớn nhất từ các nhân viên (và tôi nghĩ là Kent biết điều đó). Chúng tôi biết các vấn đề về code sạch vì chúng tôi đã phải đối mặt với nó quá lâu rồi.

Tôi có biết một công ty, vào cuối những năm 80, đã phát hành một ứng dụng *X*. Nó rất phổ biến, và nhiều chuyên gia đã mua và sử dụng nó. Nhưng sau đó, các chu kỳ cập nhật bắt đầu bị kéo dài ra, nhiều lỗi thì không được sửa từ phiên bản này qua phiên bản khác, thời gian tải và sự cố cũng theo đó mà tăng lên. Tôi vẫn nhớ ngày mà tôi đã ngưng sử dụng sản phẩm trong sự thất vọng và không dùng lại nó một lần nào nữa. Chỉ một thời gian sau, công ty đó cũng ngừng hoạt động.

Hai mươi năm sau, tôi gặp một trong những nhân viên ban đầu của công ty đó và hỏi anh ta chuyện gì đã xãy ra. Câu trả lời đã khiến tôi lo sợ : Họ đưa sản phẩm ra thị trường cùng với một đống code hỗn độn trong đó. Khi các tính năng mới được thêm vào ngày càng nhiều, code của chương trình lại ngày càng tệ, tệ đến mức họ không thể kiểm soát được nữa, và đặt dấu chấm hết cho công ty. Và, tôi gọi những dòng code đó là code "rởm".

Bạn đã bao giờ bị những dòng code "rởm" gây khó dễ chưa? Nếu là lập trình viên hẳn bạn đã từng trải nghiệm cảm giác đó vài lần. Chúng tôi có một cái tên cho nó, đó là *bơi* (từ gốc: wade – làm việc vất vả). Chúng tôi *bơi* qua những dòng code tởm lợm, *bơi* trong một mớ lộn xộn những cái bug được giấu kín. Chúng tôi cố gắng theo cách của chúng tôi, hy vọng tìm thấy những gợi ý, những manh mối hay biết chuyện gì đang xãy ra với code; nhưng tất cả những gì chúng tôi thấy là những dòng code ngày càng vô nghĩa.

Nếu bạn đã từng bị những dòng code "rởm" cản trở như tôi miêu tả, vậy thì – tại sao bạn lại tạo ra nó?

Bạn đã thử đi nhanh? Bạn đã vội vàng? Có lẽ vậy thật. Hoặc bạn cảm thấy bạn không có đủ thời gian để hoàn thành nó; hay sếp sẽ nổi điên với bạn nếu bạn dành thời gian để dọn dẹp đống code lộn xộn đó. Cũng có lẽ bạn đã quá mệt mỏi với cái chương trình chết tiệt này và muốn kết thúc nó ngay. Hoặc có thể bạn đã xem xét phần tồn đọng của những thứ khác mà bạn đã hứa sẽ hoàn thành và nhận ra rằng bạn cần phải kết hợp module này với nhau để bạn có thể chuyển sang phần tiếp theo. Yeah! Chúng ta đã tạo ra con quỷ như thế đó.

Tất cả chúng ta đều nhìn vào đống lộn xộn mà chúng ta vừa tạo ra, và chọn *một ngày đẹp trời nào đó* để giải quyết nó. Tất cả chúng ta đều cảm thấy nhẹ nhõm khi thấy "chương trình lộn xộn" của chúng ta hoạt động và cho rằng: thà có còn hơn không. Tất cả chúng ta cũng đã từng tự nhủ rằng, *sau này* chúng ta sẽ trở lại và dọn dẹp mớ hỗ lốn đó. Dĩ nhiên, trong những ngày như vậy chúng ta không biết đến quy luật của LeBlanc: *SAU NÀY đồng nghĩa với KHÔNG BAO GIỜ!*

Cái giá của sự lộn xộn

Nếu bạn là một lập trình viên đã làm việc trong 2 hoặc 3 năm, rất có thể bạn đã bị mớ code lộn xộn của người khác kéo bạn lùi lại. Nếu bạn đã là một lập trình viên lâu hơn 3 năm, rất có thể bạn đã tự làm chậm sự phát triển của bản thân bằng đống code do bạn tạo ra. Trong khoảng 1 hoặc 2 năm, các đội đã di chuyển rất nhanh ngay từ khi bắt đầu một dự án, thay vì phải di chuyển thận trọng như cách họ nhìn nhận nó. Vì vậy, mọi thay đổi mà họ tác động lên code sẽ phá vỡ vài đoạn code khác. Không có thay đổi nào là không quan trọng. Mọi sự bổ sung hoặc thay đổi chương trình đều tạo ra các mớ boòng boong, các nút thắt,... Chúng ta cố gắng hiểu chúng chỉ để tạo ra thêm sự thay đổi, và lặp lại việc tạo ra chính chúng. Theo thời gian, code của chúng ta trở nên quá "cao siêu" mà không thành viên nào có thể hiểu nổi. Chúng ta không thể "làm sạch" chúng, hoàn toàn không có cách nào cả ☹.

Khi đống code lộn xộn được tạo ra, hiệu suất của cả đội sẽ bắt đầu tuột dốc về phía 0. Khi hiệu suất giảm, người quản lý làm công việc của họ - đưa vào nhóm nhiều thành viên mới với hy vọng cải thiện tình trạng. Nhưng những nhân viên mới lại thường không nắm rõ cách hoạt động hoặc thiết kế của hệ thống, họ cũng không chắc thay đổi nào sẽ là phù hợp cho dự án. Hơn nữa, họ và những người cũ trong nhóm phải chịu áp lực khủng khiếp cho tình trạng tồi tệ của nhóm. Vậy là, càng làm việc, họ càng tạo ra nhiều code rối, và đưa cả nhóm (một lần nữa) dần tiến về cột mốc 0.

Đập đi xây lại

Cuối cùng, cả nhóm quyết định nổi loạn. Họ thông báo cho quản lý rằng họ không thể tiếp tục phát triển trên nền của đống code lộn xộn này nữa, rằng họ muốn *thiết kế lại dự án*. Dĩ nhiên ban quản lý không muốn mất thêm tài nguyên cho việc tái khởi động dự án, nhưng họ cũng không thể phủ nhận sự thật rằng hiệu suất làm việc của cả nhóm quá *tàn tạ*. Cuối cùng, họ chiều theo yêu cầu của các lập trình viên và cho phép bắt đầu lại dự án.

Một nhóm mới được chọn. Mọi người đều muốn tham gia nhóm này vì nó năng động và đầy sức sống. Nhưng chỉ những người giỏi nhất mới được chọn, những người khác phải tiếp tục duy trì dự án hiện tại.

Và bây giờ, hai nhóm đang trong một cuộc đua. Nhóm mới phải xây dựng một hệ thống mới với mọi chức năng của hệ thống cũ, không những vậy họ phải theo kịp với những thay đổi dành cho hệ thống cũ. Ban quản lý sẽ không thay thế hệ thống cũ cho đến khi hệ thống mới làm được tất cả công việc của hệ thống cũ đang làm.

Cuộc đua này có thể diễn ra trong một thời gian rất dài. Tôi đã từng thấy một cuộc đua như vậy, nó mất đến 10 năm để kết thúc. Và vào thời điểm đó, các thành viên ban đầu của *nhóm mới* đã nghỉ việc, và các thành viên hiện tại đang yêu cầu thiết kế lại hệ thống vì code của nó đã trở thành một mớ lộn xộn.

Nếu bạn đã từng trải qua, dù chỉ một phần nhỏ của câu chuyện bên trên, hẳn bạn đã biết rằng việc dành thời gian để giữ cho code sạch đẹp không chỉ là câu chuyện về chi phí, mà đó còn là vấn đề sống còn của lập trình viên chuyên nghiệp.

Thay đổi cách nhìn

Bạn đã bao giờ *bơi* trong một đống code lộn tùng phèo để nhận ra thay vì cần một giờ để xử lý nó, bạn lại tốn vài tuần? Hay thay vì ngồi lọ mọ sửa lỗi trong hàng trăm module, bạn chỉ cần tác động lên một dòng code. Nếu thật vậy, bạn không hề cô đơn, ngoài kia có hàng trăm ngàn lập trình viên như bạn.

Tại sao chuyện này lại xảy ra? Tại sao code đẹp lại nhanh chóng trở thành đống lộn xộn được? Chúng tôi có rất nhiều lời giải thích dành cho nó. Chúng tôi phàn nàn vì cho rằng các yêu cầu đã thay đổi theo hướng ngăn cản thiết kế ban đầu của hệ thống. Chúng tôi rên ư ử vì lịch làm việc quá bận rộn. Chúng tôi chửi rủa những nhà quản lý ngu ngốc và những khách hàng bảo thủ và cả những cách tiếp thị vô dụng. Nhưng thưa Dilbert, lỗi không nằm ở mục tiêu mà chúng ta hướng đến, lỗi nằm ở chính chúng ta, do chúng ta không chuyên nghiệp.

Đây có thể là một sự thật không mấy dễ chịu. Bằng cách nào những đống code rối tung rối mù này lại là lỗi của chúng tôi? Các yêu cầu vô lý thì sao? Còn lịch làm việc dày đặc? Và những tên quản lý ngu ngốc, hay các cách tiếp thị vô dụng – Không ai chịu trách nhiệm cả à?

Câu trả lời là KHÔNG. Các nhà quản lý và các nhà tiếp thị tìm đến chúng tôi vì họ cần thông tin để tạo ra những lời hứa và cam kết của chương trình; và ngay cả khi họ không tìm chúng tôi, chúng tôi cũng không ngại nói cho họ biết suy nghĩ của mình. Khách hàng tìm đến chúng tôi để xác thực các yêu cầu phù hợp với hệ thống. Các nhà quản lý tìm đến chúng tôi để giúp tạo ra những lịch trình làm việc phù hợp. Chúng tôi rất mệt mỏi trong việc lập kế hoạch cho dự án và phải nhận rất nhiều trách nhiệm khi thất bại, đặc biệt là những thất bại liên quan đến code lởm.

"Nhưng khoan!" – bạn nói – "Nếu tôi không làm những gì mà sếp tôi bảo, tôi sẽ bị sa thải". Ồ, không hẳn vậy đâu. Hầu hết những ông sếp đều muốn sự thật, ngay cả khi họ hành động trông không giống như vậy. Những ông sếp đều muốn chương trình được code đẹp, dù họ đang bị ám ảnh bởi lịch trình dày đặt. Họ có thể thay đổi lịch trình và cả những yêu cầu, đó là công việc của họ. Đó cũng là công việc *của bạn* – bảo vệ code bằng niềm đam mê.

Để giải thích điều này, hãy tưởng tượng bạn là bác sĩ và có một bệnh nhân yêu cầu bạn hãy ngưng việc rửa tay để chuẩn bị cho phẫu thuật, vì việc rửa tay mất quá nhiều thời gian. Rõ ràng bệnh nhân là thượng đế, nhưng bác sĩ sẽ luôn từ chối yêu cầu này. Tại sao? Bởi vì bác sĩ biết nhiều hơn bệnh nhân về những nguy cơ về bệnh tật và nhiễm trùng. Rõ là ngu ngốc khi bác sĩ lại đồng ý với những yêu cầu như vậy.

Tương tự như vậy, quá là nghiệp dư cho các lập trình viên luôn tuân theo các yêu cầu của sếp – những người không hề biết về nguy cơ của việc tạo ra một chương trình đầy code rối.

Vấn đề nan giải

Các lập trình viên phải đối mặt với một vấn đề nan giải về các giá trị cơ bản. Những lập trình viên với hơn 1 năm kinh nghiệm biết rằng đống code lộn xộn đó đã kéo họ xuống. Tuy nhiên, tất cả họ đều cảm thấy áp lực khi tìm cách giải quyết nó theo đúng hạn. Tóm lại, họ không dành thời gian để tạo nên hướng đi vững vàng.

Các chuyên gia thật sự biết rằng phần thứ hai của vấn đề là sai, đống code lộn xộn kia sẽ không thể giúp bạn hoàn thành công việc đúng hạn. Thật vậy, sự lộn xộn đó sẽ làm chậm bạn ngay lập tức, và buộc bạn phải trễ thời hạn. Cách duy nhất để hoàn thành đúng hạn – cách duy nhất để bước đi vững vàng – là giữ cho code luôn sạch sẽ nhất khi bạn còn có thể.

Kỹ thuật làm sạch code?

Giả sử bạn tin rằng code lởm là một chướng ngại đáng kể, giả sử bạn tin rằng cách duy nhất để có hướng đi vững vàng là giữ sạch code của bạn, thì bạn cần tự hỏi bản thân mình : "Làm cách nào để viết code cho sạch?". Nếu bạn không biết ý nghĩa của việc code sạch, tốt nhất bạn không nên viết nó.

Tin xấu là, việc tạo nên code sạch sẽ giống như cách chúng ta vẽ nên một bức tranh. Hầu hết chúng ta đều nhận ra đâu là tranh đẹp, đâu là tranh xấu – nhưng điều đó không có nghĩa là chúng ta biết cách vẽ. Vậy nên, việc bạn có thể lôi ra vài dòng code đẹp trong đống code lởm không có nghĩa là chúng ta biết cách viết nên những dòng code sạch.

Viết code sạch sẽ yêu cầu sự khổ luyện liên tục những kỹ thuật nhỏ khác nhau, và sự cần cù sẽ được đền đáp bằng cảm giác "sạch sẽ" của code. *Cảm giác (hay giác quan)* này chính là chìa khóa, một số người trong chúng ta được Chúa ban tặng ngay từ khi sinh ra, một số người khác thì phải đấu tranh để có được nó. Nó không chỉ cho phép chúng ta xem xét code đó là *xịn* hay *lởm*, mà còn cho chúng ta thấy những kỹ thuật đã được áp dụng như thế nào.

Một lập trình viên không có *giác quan code* sẽ không biết phải làm gì khi nhìn vào một đống code rối. Ngược lại, những người có *giác quan code* sẽ bắt đầu nhìn ra các cách để thay đổi nó. *Giác quan code* sẽ giúp lập trình viên chọn ra cách tốt nhất, và vạch ra con đường đúng đắn để hoàn thành công việc.

Tóm lại, một lập trình viên viết code "sạch đẹp" thật sự là một nghệ sĩ. Họ có thể tạo ra các hệ thống thân thiện chỉ từ một màn hình trống rỗng.

Code sạch là cái chi chi?

Có thể là có rất nhiều định nghĩa. Vì vậy, chúng tôi phỏng vấn một số lập trình viên nổi tiếng và giàu kinh nghiệm về khái niệm này:

Bjarne Stroustrup – cha đẻ của ngôn ngữ C++, và là tác giả của quyển *The C++ Programming Language:*

"Tôi thích code của tôi trông thanh lịch và hiệu quả. Sự logic nên được thể hiện rõ ràng để làm cho các lỗi khó lẫn trốn, sự phụ thuộc được giảm thiểu để dễ bảo trì, các lỗi được xử lý bằng các chiến lược rõ ràng, và hiệu năng thì gần như tối ưu để không lôi kéo người khác tạo nên code rối bằng những cách tối ưu hóa tạm bợ. Code sạch sẽ tạo nên những điều tuyệt vời".

Bjarne sử dụng từ *thanh lịch.* Nó khá chính xác. Từ điển trong Macbook của tôi giải thích về nó như sau: vẻ đẹp duyên dáng hoặc phong cách dễ chịu, đơn giản nhưng *làm hài lòng* mọi người. Hãy chú ý đến nội dung *làm hài lòng.* Rõ ràng Bjarne cho rằng code sạch sẽ dễ đọc hơn. Đọc nó sẽ làm cho bạn mỉm cười nhẹ nhàng như một chiếc hộp nhạc.

Bjarne cũng đề cập đến sự hiệu quả – hai lần. Không có gì bất ngờ từ người phát minh ra C++, nhưng tôi nghĩ còn nhiều điều hơn là mong muốn đạt được hiệu suất tuyệt đối. Các tài nguyên bị lãng phí, chuyện đó chẳng dễ chịu chút nào. Và bây giờ hãy để ý đến từ mà Bjarne dùng để miêu tả hậu quả – *lôi kéo.* Có một sự thật là, code lởm "thu hút" những đống code lởm khác. Khi ai đó thay đổi đống code đó, họ có xu hướng làm cho nó tệ hơn.

[…]

Bjarne cũng đề cập đến việc xử lý lỗi phải được thực hiện đầy đủ. Điều này tạo nên thói quen chú ý đến từng chi tiết nhỏ. Việc xử lý lỗi qua loa sẽ khiến các lập trình viên bỏ qua các chi tiết nhỏ: nguy cơ tràn bộ nhớ, hiện tượng tranh giành dữ liệu (race condition), hay đặt tên không phù hợp,…Vậy nên, việc code sạch sẽ tạo được tính kỹ lưỡng cho các lập trình viên.

Bjarne kết thúc cuộc phỏng vấn bằng khẳng định *code sạch sẽ tạo nên những điều tuyệt vời.* Không phải ngẫu nhiên mà tôi lại nói – những nguyên tắc về thiết kế phần mềm được cô đọng lại trong lời khuyên đơn giản này. Tác giả sau khi viết đã cố gắng truyền đạt tư tưởng này. Code rởm đã tồn tại đủ lâu, và không có lý do gì để giữ nó tiếp tục. Bây giờ, code sạch sẽ được tập trung phát triển. Mỗi hàm, mỗi lớp, mỗi mô-đun thể hiện sự độc lập, và không bị *ô nhiễm* bởi những thứ quanh nó.

Grady Booch, tác giả quyển Object Oriented Analysis and Design with Applications

"Code sạch đơn giản và rõ ràng. Đọc nó giống như việc bạn đọc một đoạn văn xuôi. Code sạch sẽ thể hiện rõ ràng ý đồ của lập trình viên, đồng thời mô tả rõ sự trừu tượng và các dòng điều khiển đơn giản".

[…]

Dave Thomas, người sáng lập OTI, godfather of the Eclipse strategy:

"Code sạch có thể được đọc và phát triển thêm bởi những lập trình viên khác. Nó đã được kiểm tra, nó có những cái tên ý nghĩa, nó cho bạn thấy cách để làm việc. Nó giảm thiểu sự phụ thuộc giữa các đối tượng với những định nghĩa rõ ràng, và cung cấp các API cần thiết. Code nên được hiểu theo cách diễn đạt, không phải tất cả thông tin cần thiết đều có thể được thể hiện rõ ràng chỉ bằng code".

[…]

Michael Feathers, tác giả quyển Working Effectively with Legacy Code:

"Tôi có thể liệt kê tất cả những phẩm chất mà tôi thấy trong code sạch, nhưng tất cả chúng được bao quát bởi một điều – code sạch trông như được viết bởi những người tận tâm. Dĩ nhiên, bạn cho rằng bạn sẽ làm nó tốt hơn. Điều đó đã được họ (những người tạo ra code sạch) nghĩ đến, và nếu bạn cố gắng "rặn" ra những cải tiến, nó sẽ đưa bạn về lại vị trí ban đầu. Ngồi xuống và tôn trọng những dòng code mà ai đó đã để lại cho bạn – những dòng code được viết bởi một người đầy tâm huyết với nghề".

[…]

Ward Cunningham, người tạo ra Wiki:

"Bạn biết bạn đang làm việc cùng code sạch là khi việc đọc code hóa ra yomost hơn những gì bạn mong đợi. Bạn có thể gọi nó là code đẹp khi những dòng code đó trông giống như cách mà bạn trình bày và giải quyết vấn đề".

[…]

Những môn phái

Còn tôi (chú Bob) thì sao? Tôi nghĩ code sạch là gì? Cuốn sách này sẽ nói cho bạn biết, đảm bảo chi tiết đến mức mệt mỏi những gì tôi và các đồng nghiệp nghĩ về code sạch. Chúng tôi sẽ cho bạn biết những gì chúng tôi nghĩ về tên biến sạch, hàm sạch, lớp sạch,…Chúng tôi sẽ trình bày những ý kiến này dưới dạng tuyệt đối, và chúng tôi sẽ không xin lỗi vì sự ngông cuồng này. Đối với chúng tôi, ngay lúc này, điều đó là tuyệt đối. Đó chính là trường phái của chúng tôi về code sạch.

Không có môn võ nào là hay nhất, cũng không có kỹ thuật nào là "vô đối" trong võ thuật. Thường thì các võ sư bậc thầy sẽ hình thành trường phái riêng của họ và thu nhận đệ tử để truyền dạy. Vì vậy, chúng ta thấy Nhu thuật Brazil (Jiu Jitsu) được sáng tạo và truyền dạy bởi dòng tộc Gracie ở Brazil. Chúng ta thấy Hakko Ryu Jiu Jitsu (một môn nhu thuật của Nhật Bản) được thành lập và truyền dạy bởi Okuyama Ryuho ở Tokyo. Chúng ta thấy Triệt Quyền Đạo, được phát triển và truyền dạy bởi Lý Tiểu Long tại Hoa Kỳ.

Môn đồ của các môn phái này thường đắm mình trong những lời dạy của sư phụ. Họ dấn thân để khám phá kiến thức mà sư phụ dạy, và thường loại bỏ giáo lý của ông thầy khác. Sau đó, khi kỹ năng của họ phát triển, họ có thể tìm một sư phụ khác để mở rộng kiến thức và va chạm thực tế nhiều hơn. Một số khác tiếp tục hoàn thiện kỹ năng của mình, khám phá các kỹ thuật mới và thành lập võ đường của riêng họ.

Không một giáo lý của môn phái nào là đúng hoàn toàn. Tuy nhiên trong một môn phái, chúng ta chấp nhận những lời dạy và những kỹ thuật đó là đúng. Sau tất cả, vẫn có cách để áp dụng đúng Triệt Quyền Đạo hay Nhu thuật. Nhưng việc đó không làm những lời dạy của môn phái khác mất tác dụng.

Hãy xem quyển sách này là một quyển bí kíp về *Môn phái Code sạch*. Các kỹ thuật và lời khuyên bên trong giúp bạn thể hiện khả năng của mình. Chúng tôi sẵn sàng khẳng định nếu bạn làm theo những lời khuyên này, bạn sẽ được hưởng những lợi ích như chúng tôi, bạn sẽ học được cách tạo nên những dòng code sạch sẽ và đầy chuyên nghiệp. Nhưng làm ơn đừng nghĩ chúng tôi đúng tuyệt đối, còn có những bậc thầy khác, họ sẽ đòi hỏi bạn phải chuyên nghiệp hơn. Điều đó sẽ giúp bạn học hỏi khá nhiều từ họ đấy.

Sự thật là, nhiều lời khuyên trong quyển sách này đang gây tranh cãi. Bạn có thể không đồng ý với tất cả chúng, hoặc một vài trong số đó. Không sao, chúng tôi không thể yêu cầu việc đó được. Mặt khác, các lời khuyên trong sách là những thứ mà chúng tôi phải trải qua quá trình suy nghĩ lâu dài và đầy khó khăn mới có được. Chúng tôi đã học được nó qua hàng chục năm làm việc, thí nghiệm và sửa lỗi. Vậy nên, cho dù bạn đồng ý hay không, đó sẽ là hành động sỉ nhục nếu bạn không xem xét, và tôn trọng quan điểm của chúng tôi.

Chúng ta là tác giả

Trường *@author* của Javadoc cho chúng ta biết chúng ta là ai – chúng ta là tác giả. Và tác giả thì phải có đọc giả. Tác giả có trách nhiệm giao tiếp tốt với các đọc giả của họ. Lần sau khi viết một dòng code, hãy nhớ rằng bạn là tác giả - đang viết cho những đọc giả, những người đánh giá sự cố gắng của bạn.

Và bạn hỏi: Có bao nhiêu code thật sự được đọc cơ chứ? Nỗ lực viết nó để làm gì?

Bạn đã bao giờ xem lại những lần chỉnh sửa code chưa? Trong những năm 80 và 90, chúng tôi đã có những chương trình như Emacs, cho phép theo dõi mọi thao tác bàn phím. Bạn nên làm việc trong một giờ rồi sau đó xem lại các phiên bản chỉnh sửa – như cách xem một bộ phim được tua nhanh. Và khi tôi làm điều này, kết quả thật bất ngờ.

Đa phần là hành động cuộn và điều hướng sang những mô-đun khác:

Bob vào mô-đun.

Anh ấy cuộn xuống chức năng cần thay đổi.

Anh ấy dừng lại, xem xét các biện pháp giải quyết.

Ồ, anh ấy cuộn lên đầu mô-đun để kiểm tra việc khởi tạo biến.

Bây giờ anh ta cuộn xuống và bắt đầu gõ.

Ooops, anh ấy xóa chúng rồi.

Anh ấy nhập lại.

Anh ấy lại xóa.

Anh ấy lại nhập một thứ gì đó, rồi lại xóa.

Anh ấy kéo xuống hàm khác đang gọi hàm mà anh ta chỉnh sửa để xem nó được gọi ra sao.

Anh ấy cuộn ngược lại, và gõ những gì anh vừa xóa.

Bob tạm ngưng.

Anh ta lại xóa nó.

Anh ta mở một cửa sổ khác và nhìn vào lớp con, xem hàm đó có bị ghi đè (overriding) hay không.

...

Thật sự lôi cuốn. Và chúng tôi nhận ra thời gian đọc code luôn gấp 10 lần thời gian viết code. Chúng tôi liên tục đọc lại code cũ như một phần trong những nỗ lực để tạo nên code mới.

Vì quá mất thời gian nên chúng tôi muốn việc đọc code trở nên dễ dàng hơn, ngay cả khi nó làm cho việc viết code khó hơn. Dĩ nhiên không có cách nào để viết code mà không đọc nó, do đó làm nó dễ đọc hơn, cũng là cách làm nó dễ viết hơn.

Không còn cách nào đâu. Bạn không thể mở rộng code nếu bạn không đọc được code. Code bạn viết hôm nay sẽ trở nên khó hoặc dễ mở rộng tùy vào cách viết của bạn. Vậy nên, nếu muốn chắc chắn, nếu muốn hoàn thành nhanh, nếu bạn muốn code dễ viết, dễ mở rộng, dễ thay đổi, hãy làm cho nó dễ đọc.

Nguyên tắc của hướng đạo sinh

Nhưng vẫn chưa đủ. Code phải được giữ sạch theo thời gian. Chúng ta đều thấy code "bốc mùi" và suy thoái theo thời gian. Vì vậy, chúng ta phải có hành động tích cực trong việc ngăn chặn sự suy thoái đó.

Các hướng đạo sinh của Mỹ có một nguyên tắc đơn giản mà chúng ta có thể áp dụng cho vấn đề này:

Khi bạn rời đi, khu cắm trại phải sạch sẽ hơn cả khi bạn đến.

Nếu chúng ta làm cho code sạch hơn mỗi khi chúng ta kiểm tra nó, nó sẽ không thể lên mùi. Việc dọn dẹp không phải là thứ gì đó to tát: đặt lại một cái tên khác tốt hơn cho biến, chia nhỏ một hàm quá lớn, đá đít vài sự trùng lặp không cần thiết, dọn dẹp vài điều kiện `if`,…

Liên tục cải thiện code, làm cho code của dự án tốt dần theo thời gian chính là một phần quan trọng của sự chuyên nghiệp.

Prequel and Principles

Với cách nhìn khác, quyển sách này là một "tiền truyện" của một quyển sách khác mà tôi đã viết vào năm 2002, nó mang tên Agile Software Development: Principles, Patterns, and Practices (PPP). Quyển PPP liên quan đến các nguyên tắc của thiết kế hướng đối tượng, và các phương pháp được sử dụng bởi các lập trình viên chuyên nghiệp. Nếu bạn chưa đọc PPP, thì đó là quyển sách kể tiếp câu chuyện của quyển sách này. Nếu đã đọc, bạn sẽ thấy chúng giống nhau ở vài đoạn code.

[…]

Kết luận

Một quyển sách về nghệ thuật không hứa đưa bạn thành nghệ sĩ, tất cả những gì nó làm được là cung cấp cho bạn những kỹ năng, công cụ, và quá trình suy nghĩ mà các nghệ sĩ đã sử dụng. Vậy nên, quyển sách này không hứa sẽ làm cho bạn trở thành một lập trình viên giỏi, cũng không hứa sẽ mang đến cho bạn *giác quan code.* Tất cả những gì nó làm là cho bạn thấy phương pháp làm việc của những lập trình viên hàng đầu, cùng với các kỹ năng, thủ thuật, công cụ,…mà họ sử dụng.

Như những quyển sách về nghệ thuật khác, quyển sách này đầy đủ chi tiết. Sẽ có rất nhiều code. Bạn sẽ thấy code tốt và code tồi. Bạn sẽ thấy cách chuyển code tồi thành code tốt. Bạn sẽ thấy một danh sách các cách giải quyết, các nguyên tắc và kỹ năng. Có rất nhiều ví dụ cho bạn. Còn sau đó thì, tùy bạn.

Hãy nhớ tới câu chuyện vui về nghệ sĩ violin đã bị lạc trên đường tới buổi biểu diễn. Anh hỏi một ông già trên phố làm thế nào để đến Carnegie Hall (nơi được xem là thánh đường âm nhạc). Ông già nhìn người nghệ sĩ và cây violin được giấu dưới cánh tay anh ta, nói to: *Luyện tập, con trai. Là luyện tập!*

Tham khảo

Implementation Patterns, Kent Beck, Addison-Wesley, 2007.

Literate Programming, Donald E. Knuth, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

CHƯƠNG 2

***

NHỮNG CÁI TÊN RÕ NGHĨA

Viết bởi Tim Ottinger

Giới thiệu

Những cái tên có ở khắp mọi nơi trong phần mềm. Chúng ta đặt tên cho các biến, các hàm, các đối số, các lớp và các gói của chúng ta. Chúng ta đặt tên cho những file mã nguồn và thư mục chứa chúng. Chúng ta đặt tên cho những file **.jar*, file **.war*,... Chúng ta đặt tên và đặt tên. Vì chúng ta đặt tên rất nhiều, nên chúng ta cần làm tốt điều đó. Sau đây là một số quy tắc đơn giản để tạo nên những cái tên tốt.

Dùng những tên thể hiện được mục đích

Điều này rất dễ. Nhưng chúng tôi muốn nhấn mạnh rằng chúng tôi nghiêm túc trong việc này. Chọn một cái tên "xịn" mất khá nhiều thời gian, nhưng lại tiết kiệm (thời gian) hơn sau đó. Vì vậy, hãy quan tâm đến cái tên mà bạn chọn và chỉ thay đổi chúng khi bạn sáng tạo ra tên "xịn" hơn. Những người đọc code của bạn (kể cả bạn) sẽ *sung sướng* hơn khi bạn làm điều đó.

Tên của biến, hàm, hoặc lớp phải trả lời tất cả những câu hỏi về nó. Nó phải cho bạn biết lý do nó tồn tại, nó làm được những gì, và dùng nó ra sao. Nếu có một comment đi kèm theo tên, thì tên đó không thể hiện được mục đích của nó.

```
int d;  // elapsed time in days
```

Tên **`d`** không tiết lộ điều gì cả. Nó không gợi lên cảm giác gì về thời gian, cũng không liên quan gì đến ngày. Chúng ta nên chọn một tên thể hiện được những gì đang được cân đo, và cả đơn vị đo của chúng:

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Việc chọn tên thể hiện được mục đích có thể làm cho việc hiểu và thay đổi code dễ dàng hơn nhiều. Hãy đoán xem mục đích của đoạn code dưới đây là gì?

```
public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if (x[0] == 4)
            list1.add(x);
    return list1;
}
```

Tại sao lại nói khó mà biết được đoạn code này đang làm gì? Không có biểu thức phức tạp, khoảng cách và cách thụt đầu dòng hợp lý, chỉ có 3 biến và 2 hằng số được đề cập. Thậm chí không có các lớp (class) và phương thức đa hình nào, nó chỉ có một danh sách mảng (hoặc thứ gì đó trông giống vậy).

Vấn đề không nằm ở sự đơn giản của code mà nằm ở ý nghĩa của code, do bối cảnh không rõ ràng. Đoạn code trên bắt chúng tôi phải tìm câu trả lời cho các câu hỏi sau:

1. `theList` chứa cái gì?
2. Ý nghĩa của chỉ số 0 trong phần tử của `theList`?
3. Số `4` có ý nghĩa gì?
4. Danh sách được `return` thì dùng kiểu gì?

Câu trả lời không có trong code, nhưng sẽ có ngay sau đây. Giả sử chúng tôi đang làm game *dò mìn*. Chúng tôi thấy rằng giao diện trò chơi là một danh sách các ô vuông (cell) được gọi là `theList`. Vậy nên, hãy đổi tên nó thành `gameBoard`.

Mỗi ô trên màn hình được biểu diễn bằng một sanh sách đơn giản. Chúng tôi cũng thấy rằng chỉ số của số 0 là vị trí biểu diễn giá trị trạng thái (status value), và giá trị 4 nghĩa là trạng thái *được gắn cờ (flagged)*. Chỉ bằng cách đưa ra các khái niệm này, chúng tôi có thể cải thiện mã nguồn một cách đáng kể:

```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
        if (cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.add(cell);
    return flaggedCells;
}
```

Cần lưu ý rằng mức độ đơn giản của code vẫn không thay đổi, nó vẫn chính xác về toán tử, hằng số, và các lệnh lồng nhau,…Nhưng đã trở nên rõ ràng hơn rất nhiều.

Chúng ta có thể đi xa hơn bằng cách viết một lớp đơn giản cho các ô thay vì sử dụng các mảng kiểu `int`. Nó có thể bao gồm một hàm thể hiện được mục đích (gọi nó là *isFlagged – được gắn cờ* chẳng hạn) để giấu đi những con số ma thuật *(Từ gốc: magic number – Một khái niệm về các hằng số, tìm hiểu thêm tại https://en.wikipedia.org/wiki/Magic_number_(programming)).*

```
public List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();
    for (Cell cell : gameBoard)
        if (cell.isFlagged())
            flaggedCells.add(cell);
    return flaggedCells;
}
```

Với những thay đổi đơn giản này, không quá khó để hiểu những gì mà đoạn code đang trình bày. Đây chính là sức mạnh của việc chọn tên tốt.

Tránh sai lệch thông tin

Các lập trình viên phải tránh để lại những dấu hiệu làm code trở nên khó hiểu. Chúng ta nên tránh dùng những từ mang nghĩa khác với nghĩa cố định của nó. Ví dụ, các tên biến như `hp`, `aix` và `sco` là những tên biến vô cùng tồi tệ, chúng là tên của các nền tảng Unix hoặc các biến thể. Ngay cả khi bạn đang code về cạnh huyền (hypotenuse) và `hp` trông giống như một tên viết tắt tốt, rất có thể đó là một cái tên tồi.

Không nên quy kết rằng một nhóm các tài khoản là một `accountList` nếu nó không thật sự là một danh sách (`List`). Từ *danh sách* có nghĩa là một thứ gì đó cụ thể cho các lập trình viên. Nếu các tài khoản không thực sự tạo thành danh sách, nó có thể dẫn đến một kết quả sai lầm. Vậy nên, `accountGroup` hoặc `bunchOfAccounts`, hoặc đơn giản chỉ là `accounts` sẽ tốt hơn.

Cẩn thận với những cái tên gần giống nhau. Mất bao lâu để bạn phân biệt được sự khác nhau giữa `XYZControllerForEfficientHandlingOfStrings` và `XYZControllerForEfficientStorageOfStrings` trong cùng một module, hay đâu đó xa hơn một chút? Những cái tên gần giống nhau như thế này thật sự, thật sự rất khủng khiếp cho lập trình viên.

Một kiểu khủng bố tinh thần khác về những cái tên không rõ ràng là ký tự L viết thường và O viết hoa. Vấn đề? Tất nhiên là nhìn chúng gần như hoàn toàn giống hằng số không và một, kiểu như:

```
int a = l;
if ( O == l ) a = O1;
else l = 01;
```

Bạn nghĩ chúng tôi *xạo*? Chúng tôi đã từng khảo sát, và kiểu code như vậy thực sự rất nhiều. Trong một số trường hợp, tác giả của code đề xuất sử dụng phông chữ khác nhau để tách biệt chúng. Một giải pháp khác có thể được sử dụng là truyền đạt bằng lời nói hoặc để lại tài liệu cho các lập trình viên sau này có thể hiểu nó. Vấn đề được giải quyết mà không cần phải đổi tên để tạo ra một sản phẩm khác.

Tạo nên những sự khác biệt có nghĩa

Các lập trình viên tạo ra vấn đề cho chính họ khi viết code chỉ để đáp ứng cho trình biên dịch hoặc thông dịch. Ví dụ, vì bạn không thể sử dụng cùng một tên để chỉ hai thứ khác nhau trong cùng một khối lệnh hoặc cùng một phạm vi, bạn có thể bị "dụ dỗ" thay đổi tên một cách tùy tiện. Đôi khi điều đó làm bạn cố tình viết sai chính tả, và người nào đó quyết định sửa lỗi chính tả đó, khiến trình biên dịch không có khả năng hiểu nó (cụ thể – tạo ra một biến tên *klass* chỉ vì tên *class* đã được dùng cho thứ gì đó).

Mặc dù trình biên dịch có thể làm việc với những tên này, nhưng điều đó không có nghĩa là bạn được phép dùng nó. Nếu tên khác nhau, thì chúng cũng có ý nghĩa khác nhau.

Những tên dạng chuỗi số (`a1, a2,… aN`) đi ngược lại nguyên tắc đặt tên có mục đích. Mặc dù những tên như vậy không phải là không đúng, nhưng chúng không có thông tin. Chúng không cung cấp manh mối nào về ý định của tác giả. Ví dụ:

```
public static void copyChars(char a1[], char a2[]) {
    for (int i = 0; i < a1.length; i++) {
        a2[i] = a1[i];
    }
}
```

Hàm này dễ đọc hơn nhiều khi *nguyên nhân* và *mục đích* của nó được đặt tên cho các đối số.

Những từ gây nhiễu tạo nên sự khác biệt, nhưng là sự khác biệt vô dụng. Hãy tưởng tượng rằng bạn có một lớp `Product`, nếu bạn có một `ProductInfo` hoặc `ProductData` khác, thì bạn đã thành công trong việc tạo ra các tên khác nhau nhưng về mặt ngữ nghĩa thì chúng là một. `Info` và `Data` là các từ gây nhiễu, giống như `a`, `an` và `the`.

Lưu ý rằng không có gì sai khi sử dụng các tiền tố như `a` và `the` để tạo ra những khác biệt hữu ích. Ví dụ, bạn có thể sử dụng `a` cho tất cả các biến cục bộ và tất cả các đối số của hàm. `a` và `the` sẽ trở thành vô dụng khi bạn quyết định tạo một biến `theZork` vì trước đó bạn đã có một biến mang tên `Zork`.

Những từ gây nhiễu là không cần thiết. Từ `variable` sẽ không bao giờ xuất hiện trong tên biến, từ `table` cũng không nên dùng trong tên bảng. `NameString` sao lại tốt hơn `Name`? `Name` có bao giờ là một số đâu mà lại? Nếu `Name` là một số, nó đã phá vỡ nguyên tắc *Tránh sai lệch thông tin.* Hãy tưởng tượng bạn đang tìm kiếm một lớp có tên `Customer`, và một lớp khác có tên `CustomerObject`. Chúng khác nhau kiểu gì? Cái nào chứa lịch sử thanh toán của khách hàng? Còn cái nào chứa thông tin của khách?

Có một ứng dụng minh họa cho các lỗi trên, chúng tôi đã thay đổi một chút về tên để bảo vệ tác giả. Đây là những thứ chúng tôi thấy trong mã nguồn:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

Tôi thắc mắc không biết các lập trình viên trong dự án này phải `getActiveAccount` như thế nào!

Trong trường hợp không có quy ước cụ thể, biến `moneyAmount` không thể phân biệt được với `money`; `customerInfo` không thể phân biệt được với `customer`; `accountData` không thể phân biệt được với `account` và `theMessage` với `message` được xem là một. Hãy phân biệt tên theo cách cung cấp cho người đọc những khác biệt rõ ràng.

Dùng những tên phát âm được

Con người rất giỏi về từ ngữ. Một phần quan trọng trong bộ não của chúng ta được dành riêng cho các khái niệm về từ. Và các từ, theo định nghĩa, có thể phát âm được. Thật lãng phí khi không sử dụng được bộ não mà chúng ta đã tiến hóa nhằm thích nghi với ngôn ngữ nói. Vậy nên, hãy làm cho những cái tên phát âm được đi nào.

Nếu bạn không thể phát âm nó, thì bạn không thể thảo luận một cách bình thường: "Hey, ở đây chúng ta có *bee cee arr three cee enn tee*, và *pee ess zee kyew int*, thấy chứ?" – Vâng, tôi thấy một thằng thiểu năng. Vấn đề này rất quan trọng vì lập trình cũng là một hoạt động xã hội, chúng ta cần trao đổi với mọi người.

Tôi có biết một công ty dùng tên `genymdhms` (generation date, year, month, day, hour, minute, and second – phát sinh ngày, tháng, năm, giờ, phút, giây), họ đi xung quanh tôi và "gen why emm dee aich emm ess" (cách phát âm theo tiếng Anh). Tôi có thói quen phát âm như những gì tôi viết, vì vậy tôi bắt đầu nói "gen-yah-muddahims". Sau này nó được gọi bởi một loạt các nhà thiết kế và phân tích, và nghe vẫn có vẻ ngớ ngẩn. Chúng tôi đã từng troll nhau như thế, nó rất thú vị. Nhưng dẫu thế nào đi nữa, chúng tôi đã chấp nhận những cái tên xấu xí. Những lập trình viên mới của công ty tìm hiểu ý nghĩa của các biến, và sau đó họ nói về những từ ngớ ngẩn, thay vì dùng các thuật ngữ tiếng Anh cho thích hợp. Hãy so sánh:

```
class DtaRcrd102 {
    private Date genymdhms;
    private Date modymdhms;
    private final String pszqint = "102";
    /* ... */
};
```

và

```
class Customer {
    private Date generationTimestamp;
    private Date modificationTimestamp;
    private final String recordId = "102";
    /* ... */
};
```

Cuộc trò chuyện giờ đây đã thông minh hơn: "Hey, Mikey, take a look at this record! The generation timestamp is set to tomorrow's date! How can that be?"

Dùng những tên tìm kiếm được

Các tên một chữ cái và các hằng số luôn có vấn đề, đó là không dễ để tìm chúng trong hàng ngàn dòng code.

Người ta có thể dễ dàng tìm kiếm `MAX_CLASSES_PER_STUDENT`, nhưng số 7 thì lại rắc rối hơn. Các công cụ tìm kiếm có thể mở các tệp, các hằng, hoặc các biểu thức chứa số 7 này, nhưng được sử dụng với các mục đích khác nhau. Thậm chí còn tồi tệ hơn khi hằng số là một số có giá trị lớn và ai đó vô tình thay đổi giá trị của nó, từ đó tạo ra một lỗi mà các lập trình viên không tìm ra được.

Tương tự như vậy, tên `e` là một sự lựa chọn tồi tệ cho bất kỳ biến nào mà một lập trình viên cần tìm kiếm. Nó là chữ cái phổ biến nhất trong tiếng anh và có khả năng xuất hiện trong mọi đoạn code của chương trình. Về vấn đề này, tên dài thì tốt hơn tên ngắn, và những cái tên tìm kiếm được sẽ tốt hơn một hằng số trơ trọi trong code.

Sở thích cá nhân của tôi là chỉ đặt tên ngắn cho những biến cục bộ bên trong những phương thức ngắn. *Độ dài của tên phải tương ứng với phạm vi hoạt động của nó*. Nếu một biến hoặc hằng số được nhìn thấy và sử dụng ở nhiều vị trí trong phần thân của mã nguồn, bắt buộc phải đặt cho nó một tên dễ tìm kiếm. Ví dụ:

```
for (int j=0; j<34; j++) {
    s += (t[j]*4)/5;
}
```

và

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
    int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
    int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
    sum += realTaskWeeks;
}
```

Lưu ý rằng `sum` ở ví dụ trên, dù không phải là một tên đầy đủ nhưng có thể tìm kiếm được. Biến và hằng được cố tình đặt tên dài, nhưng hãy so sánh việc tìm kiếm `WORK_DAYS_PER_WEEK` dễ hơn bao nhiêu lần so với số 5, đó là chưa kể cần phải lọc lại danh sách tìm kiếm và tìm ra những trường hợp có nghĩa.

Tránh việc mã hóa

Các cách mã hóa hiện tại là đủ với chúng tôi. Mã hóa các kiểu dữ liệu hoặc phạm vi thông tin vào tên chỉ đơn giản là thêm một gánh nặng cho việc giải mã. Điều đó không hợp lý khi bắt nhân viên phải học thêm một "ngôn ngữ" mã hóa khác khác ngoài các ngôn ngữ mà họ dùng để làm việc với code. Đó là một gánh nặng không cần thiết, các tên mã hóa ít khi được phát âm và dễ bị đánh máy sai.

Ký pháp Hungary

Ngày trước, khi chúng tôi làm việc với những ngôn ngữ mà độ dài tên là một thách thức, chúng tôi đã loại bỏ sự cần thiết này. Fortran bắt buộc mã hóa bằng những chữ cái đầu tiên, phiên bản BASIC ban đầu của nó chỉ cho phép đặt tên tối đa 6 ký tự. Ký pháp Hungary (KH) đã giúp ích cho việc đặt tên rất nhiều.

KH thực sự được coi là quan trọng khi Windows C API xuất hiện, khi mọi thứ là một số nguyên, một con trỏ kiểu `void` hoặc là các chuỗi,.... Trong những ngày đó, trình biên dịch không thể kiểm tra được các lỗi về kiểu dữ liệu, vì vậy các lập trình viên cần một cái phao cứu sinh trong việc nhớ các kiểu dữ liệu này.

Trong các ngôn ngữ hiện đại, chúng ta có nhiều kiểu dữ liệu mặc định hơn, và các trình biên dịch có thể phân biệt được chúng. Hơn nữa, mọi người có xu hướng làm cho các lớp, các hàm trở nên nhỏ hơn để dễ dàng thấy nguồn gốc dữ liệu của biến mà họ đang sử dụng.

Các lập trình viên Java thì không cần mã hóa. Các kiểu dữ liệu mặc định là đủ mạnh mẽ, và các công cụ sửa lỗi đã được nâng cấp để chúng có thể phát hiện các vấn đề về dữ liệu trước khi được biên dịch. Vậy nên, hiện nay KH và các dạng mã hóa khác chỉ đơn giản là một loại chướng ngại vật. Chúng làm cho việc đổi tên biến, tên hàm, tên lớp (hoặc kiểu dữ liệu của chúng) trở nên khó khăn hơn. Chúng làm cho code khó đọc, và tạo ra một hệ thống mã hóa có khả năng đánh lừa người đọc:

```
PhoneNumber phoneString;
     // name not changed when type changed!
```

Các thành phần tiền tố

Bạn cũng không cần phải thêm các tiền tố như `m_` vào biến thành viên (member variable) nữa. Các lớp và các hàm phải đủ nhỏ để bạn không cần chúng. Và bạn nên sử dùng các công cụ chỉnh sửa giúp làm nổi bật các biến này, làm cho chúng trở nên khác biệt với phần còn lại.

```
public class Part {
    private String m_dsc; // The textual description
    void setName(String name) {
        m_dsc = name;
    }
}
/*...*/
public class Part {
    String description;
    void setDescription(String description) {
        this.description = description;
    }
}
```

Bên cạnh đó, mọi người cũng nhanh chóng bỏ qua các tiền tố (hoặc hậu tố) để xem phần có ý nghĩa của tên. Càng đọc code, chúng ta càng ít thấy các tiền tố. Cuối cùng, các tiền tố trở nên vô hình, và bị xem là một dấu hiệu của những dòng code lạc hậu.

Giao diện và thực tiễn

Có một số trường hợp đặc biệt cần mã hóa. Ví dụ: bạn đang xây dựng một ABSTRACT FACTORY. Factory sẽ là giao diện và sẽ được thực hiện bởi một lớp cụ thể. Bạn sẽ đặt tên cho chúng là gì? `IShapeFactory` và `ShapeFactory` ? Tôi thích dùng những cách đặt tên đơn giản. Trước đây, `I` rất phổ biến trong các tài liệu, nó làm chúng tôi phân tâm và đưa ra quá nhiều thông tin. Tôi không muốn người dùng biết rằng tôi đang tạo cho họ một giao diện, tôi chỉ muốn họ biết rằng đó là `ShapeFactory`. Vì vậy, nếu phải lựa chọn việc mã hóa hay thể hiện đầy đủ, tôi sẽ chọn cách thứ nhất. Gọi nó là `ShapeFactoryImp`, hoặc thậm chí là `CShapeFactory` là cách hoàn hảo để che giấu thông tin.

Tránh "hiếp râm não" người khác

Những lập trình viên khác sẽ không cần phải điên đầu ngồi dịch các tên mà bạn đặt thành những tên mà họ biết. Vấn đề này thường phát sinh khi bạn chọn một thuật ngữ không chính xác.

Đây là vấn đề với các tên biến đơn. Chắc chắn một vong lặp có thể sử dụng các biến được đặt tên là `i`, `j` hoặc `k` (không bao giờ là `l` – dĩ nhiên rồi) nếu phạm vi của nó là rất nhỏ và không có tên khác xung đột với nó. Điều này là do việc đặt tên có một chữ cái trong vòng lặp đã trở thành truyền thống. Tuy nhiên, trong hầu hết trường hợp, tên một chữ cái không phải là sự lựa chọn tốt. Nó chỉ là một tên đầu gấu, bắt người đọc phải điên đầu tìm hiểu ý nghĩa, vai trò của nó. Không có lý do nào tồi tệ hơn cho cho việc sử dụng tên `c` chỉ vì `a` và `b` đã được dùng trước đó.

Nói chung, lập trình viên là những người khá thông minh. Và những người thông minh đôi khi muốn thể hiện điều đó bằng cách hack não người khác. Sau tất cả, nếu bạn đủ khả năng nhớ `r` là *the lower-cased version of the url with the host and scheme removed*, thì rõ ràng là – bạn cực kỳ thông minh luôn.

Sự khác biệt giữa lập trình viên thông minh và lập trình viên chuyên nghiệp là họ – những người chuyên nghiệp hiểu rằng sự rõ ràng là trên hết. Các chuyên gia dùng khả năng của họ để tạo nên những dòng code mà người khác có thể hiểu được.

Tên lớp

Tên lớp và các đối tượng nên sử dụng danh từ hoặc cụm danh từ, như `Customer`, `WikiPage`, `Account`, và `AddressParser`. Tránh những từ như `Manager`, `Processor`, `Data`, hoặc `Info` trong tên của một lớp. Tên lớp không nên dùng động từ.

Tên các phương thức

Tên các phương thức nên có động từ hoặc cụm động từ như `postPayment`, `deletePage`, hoặc `save`. Các phương thức truy cập, chỉnh sửa thuộc tính phải được đặt tên cùng với `get`, `set` và `is` theo tiêu chuẩn của Javabean.

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

Khi các hàm khởi tạo bị nạp chồng, sử dụng các phương thức tĩnh có tên thể hiện được đối số sẽ tốt hơn. Ví dụ:

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

sẽ tốt hơn câu lệnh

```
Complex fulcrumPoint = new Complex(23.0);
```

Xem xét việc thực thi chúng bằng các hàm khởi tạo private tương ứng.

Đừng thể hiện rằng bạn cute

Nếu tên quá hóm hỉnh, chúng sẽ chỉ được nhớ bởi tác giả và những người bạn. Liệu có ai biết chức năng của hàm `HolyHandGrenade` không? Nó rất thú vị, nhưng trong trường hợp này, `DeleteItems` sẽ là tên tốt hơn. Chọn sự rõ ràng thay vì giải trí.

Sự cute thường xuất hiện dưới dạng phong tục hoặc tiếng lóng. Ví dụ: đừng dùng `whack()` thay thế cho `kill()`, đừng mang những câu đùa trong văn hóa nước mình vào code, như `eatMyShorts()` có nghĩa là `abort()`.

Say what you mean. Mean what you say.

Chọn một từ cho mỗi khái niệm

Chọn một từ cho một khái niệm và gắn bó với nó. Ví dụ, rất khó hiểu khi `fetch`, `retrieve` và `get` là các phương thức có cùng chức năng, nhưng lại đặt tên khác nhau ở các lớp khác nhau. Làm thế nào để nhớ phương thức nào đi với lớp nào? Buồn thay, bạn phải nhớ tên công ty, nhóm hoặc cá nhân nào đã viết ra các thư viện hoặc các lớp, để nhớ cụm từ nào được dùng cho các phương thức. Nếu không, bạn sẽ mất thời gian để tìm hiểu chúng trong các đoạn code trước đó.

Các công cụ chỉnh sửa hiện đại như Eclipse và IntelliJ cung cấp các định nghĩa theo ngữ cảnh, chẳng hạn như danh sách các hàm bạn có thể gọi trên một đối tượng nhất định. Nhưng lưu ý rằng, danh sách thường không cung cấp cho bạn các ghi chú bạn đã viết xung quanh tên hàm và danh sách tham

số. Bạn may mắn nếu nó cung cấp tên tham số từ các khai báo hàm. Tên hàm phải đứng một mình, và chúng phải nhất quán để bạn có thể chọn đúng phương pháp mà không cần phải tìm hiểu thêm.

Tương tự như vậy, rất khó hiểu khi `controller`, `manager` và `driver` lại xuất hiện trong cùng một mã nguồn. Có sự khác biệt nào giữa `DeviceManager` và `ProtocolController`? Tại sao cả hai đều không phải là `controller` hay `manager`? Hay cả hai đều cùng là `driver`? Tên dẫn bạn đến hai đối tượng có kiểu khác nhau, cũng như có các lớp khác nhau.

Một từ phù hợp chính là một ân huệ cho những lập trình viên phải dùng code của bạn.

Đừng chơi chữ

Tránh dùng cùng một từ cho hai mục đích. Sử dụng cùng một thuật ngữ cho hai ý tưởng khác nhau về cơ bản là một cách chơi chữ.

Nếu bạn tuân theo nguyên tắc *Chọn một từ cho mỗi khái niệm*, bạn có thể kết thúc nhiều lớp với một…Ví dụ, phương thức `add`. Miễn là danh sách tham số và giá trị trả về của các phương thức `add` này tương đương về ý nghĩa, tất cả đều tốt.

Tuy nhiên, người ta có thể quyết định dùng từ `add` khi người đó không thực sự tạo nên một hàm có cùng ý nghĩa với cách hoạt động của hàm `add`. Giả sử chúng tôi có nhiều lớp, trong đó `add` sẽ tạo một giá trị mới bằng cách cộng hoặc ghép hai giá trị hiện tại. Bây giờ, giả sử chúng tôi đang viết một lớp mới và có một phương thức thêm tham số của nó vào mảng. Chúng tôi có nên gọi nó là `add` không? Có vẻ phù hợp đấy, nhưng trong trường hợp này, ý nghĩa của chúng là khác nhau, vậy nên chúng tôi dùng một cái tên khác như `insert` hay `append` để thay thế. Nếu được dùng cho phương thức mới, `add` chính xác là một kiểu chơi chữ.

Mục tiêu của chúng tôi, với tư cách là tác giả, là làm cho code của chúng tôi dễ hiểu nhất có thể. Chúng tôi muốn code của chúng tôi là một bài viết ngắn gọn, chứ không phải là một bài nghiên cứu […].

Dùng thuật ngữ

Hãy nhớ rằng những người đọc code của bạn là những lập trình viên, vậy nên hãy sử dụng các thuật ngữ khoa học, các thuật toán, tên mẫu (pattern),…cho việc đặt tên. Sẽ không khôn ngoan khi bạn đặt tên của vấn đề theo cách mà khách hàng định nghĩa. Chúng tôi không muốn đồng nghiệp của chúng tôi phải tìm khách hàng để hỏi ý nghĩa của tên, trong khi họ đã biết khái niệm đó – nhưng là dưới dạng một cái tên khác.

Tên `AccountVisitor` có ý nghĩa rất nhiều đối với một lập trình viên quen thuộc với mô hình VISITOR (VISITOR pattern). Có lập trình viên nào không biết `JobQueue`? Có rất nhiều thứ liên quan đến kỹ thuật mà lập trình viên phải đặt tên. Chọn những tên thuật ngữ thường là cách tốt nhất.

Thêm ngữ cảnh thích hợp

Chỉ có một vài cái tên có nghĩa trong mọi trường hợp – số còn lại thì không. Vậy nên, bạn cần đặt tên phù hợp với ngữ cảnh, bằng cách đặt chúng vào các lớp, các hàm hoặc các không gian tên (namespace). Khi mọi thứ thất bại, tiền tố nên được cân nhắc như là giải pháp cuối cùng.

Hãy tưởng tượng bạn có các biến có tên là `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state` và `zipcode`. Khi kết hợp với nhau, chúng rõ ràng tạo thành một địa chỉ. Nhưng nếu bạn chỉ thấy biến `state` được sử dụng một mình trong một phương thức thì sao? Bạn có thể suy luận ra đó là một phần của địa chỉ không?

Bạn có thể thêm ngữ cảnh bằng cách sử dụng tiền tố: `addrFirstName`, `addrLastName`, `addrState`,... Ít nhất người đọc sẽ hiểu rằng những biến này là một phần của một cấu trúc lớn hơn. Tất nhiên, một giải pháp tốt hơn là tạo một lớp có tên là `Address`. Khi đó, ngay cả trình biên dịch cũng biết rằng các biến đó thuộc về một khái niệm lớn hơn.

Hãy xem xét các phương thức trong *Listing 2-1*. Các biến có cần một ngữ cảnh có ý nghĩa hơn không? Tên hàm chỉ cung cấp một phần của ngữ cảnh, thuật toán cung cấp phần còn lại. Khi bạn đọc qua hàm, bạn thấy rằng ba biến, `number`, `verb` và `pluralModifier`, là một phần của thông báo "giả định thống kê". Thật không may, bối cảnh này phải suy ra mới có được. Khi bạn lần đầu xem xét phương thức, ý nghĩa của các biến là không rõ ràng.

Listing 2-1

Biến với bối cảnh không rõ ràng.

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
        number = "no";
        verb = "are";
        pluralModifier = "s";
    } else if (count == 1) {
        number = "1";
        verb = "is";
        pluralModifier = "";
    } else {
        number = Integer.toString(count);
        verb = "are";
        pluralModifier = "s";
    }
```

```
    String guessMessage = String.format("There %s %s %s%s", verb,
                                number, candidate, pluralModifier);
    print(guessMessage);
}
```

Hàm này hơi dài và các biến được sử dụng xuyên suốt. Để tách hàm thành các phần nhỏ hơn, chúng ta cần tạo một lớp `GuessStatisticsMessage` và tạo ra ba biến của lớp này. Điều này cung cấp một bối cảnh rõ ràng cho ba biến. Chúng là một phần của `GuessStatisticsMessage`. Việc cải thiện bối cảnh cũng cho phép thuật toán được rõ ràng hơn bằng cách chia nhỏ nó thành nhiều chức năng nhỏ hơn. (Xem *Listing 2-2*.)

Listing 2-2

Biến có ngữ cảnh

```
public class GuessStatisticsMessage {
    private String number;
    private String verb;
    private String pluralModifier;
    public String make(char candidate, int count) {
        createPluralDependentMessageParts(count);
        return String.format("There %s %s %s%s",verb, number, candidate,
                                        pluralModifier );
    }
    private void createPluralDependentMessageParts(int count) {
        if (count == 0) {
            thereAreNoLetters();
        } else if (count == 1) {
            thereIsOneLetter();
        } else {
            thereAreManyLetters(count);
        }
    }
    private void thereAreManyLetters(int count) {
        number = Integer.toString(count);
        verb = "are";
        pluralModifier = "s";
    }
    private void thereIsOneLetter() {
        number = "1";
        verb = "is";
```

```
        pluralModifier = "";
    }
    private void thereAreNoLetters() {
        number = "no";
        verb = "are";
        pluralModifier = "s";
    }
}
```

Tên ngắn thường tốt hơn tên dài, miễn là chúng rõ ràng. Thêm đủ ngữ cảnh cho tên sẽ tốt hơn khi cần thiết.

Tên `accountAddress` và `customerAddress` là những tên đẹp cho trường hợp đặc biệt của lớp `Address` nhưng có thể là tên tồi cho các lớp khác. `Address` là một tên đẹp cho lớp. Nếu tôi cần phân biệt giữa địa chỉ MAC, địa chỉ cổng (port) và địa chỉ web thì tôi có thể xem xét `MAC`, `PostalAddress` và `URL`. Kết quả là tên chính xác hơn. Đó là tâm điểm của việc đặt tên.

Lời kết

Điều khó khăn nhất của việc lựa chọn tên đẹp là nó đòi hỏi kỹ năng mô tả tốt và nền tảng văn hóa lớn. Đây là vấn đề về học hỏi hơn là vấn đề kỹ thuật, kinh doanh hoặc quản lý. Kết quả là nhiều người trong lĩnh vực này không học cách làm điều đó.

Mọi người cũng sợ đổi tên mọi thứ vì lo rằng người khác sẽ phản đối. Chúng tôi không chia sẻ nỗi sợ đó cho bạn. Chúng tôi thật sự biết ơn những ai đã đổi tên khác cho biến, hàm,…(theo hướng tốt hơn). Hầu hết thời gian chúng tôi không thật sự nhớ tên lớp và những phương thức của nó. Chúng tôi có các công cụ giúp chúng tôi trong việc đó để chúng tôi có thể tập trung vào việc code có dễ đọc hay không. Bạn có thể sẽ gây ngạc nhiên cho ai đó khi bạn đổi tên, giống như bạn có thể làm với bất kỳ cải tiến nào khác. Đừng để những cái tên tồi phá hủy sự nghiệp coder của mình.

Thực hiện theo một số quy tắc trên và xem liệu bạn có cải thiện được khả năng đọc code của mình hay không. Nếu bạn đang bảo trì code của người khác, hãy sử dụng các công cụ tái cấu trúc để giải quyết vấn đề này. Mất một ít thời gian nhưng có thể làm bạn nhẹ nhõm trong vài tháng.

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.

CLEAN CODE

A handbook of agile software craftsmanship

(Code sạch – Cẩm nang của lập trình viên)

CHƯƠNG 4

***

CƠM-MỪN

"Đừng biến đống code gớm ghiếc của bạn thành comment – hãy viết lại nó"

– Brian W. Kernighan and P. J. Plaugher

Không gì hữu ích bằng một comment được đặt đúng chỗ. Không gì có thể làm lộn xộn đống code của bạn, ngoại trừ những comment ngu xuẩn và dối trá. Và không gì có thể gây nguy hiểm bằng một comment cộc lốc từ đời nào và lại không đúng sự thật.

Các comment không phải là *Bản danh sách của Schindler* (Schindler's List – https://vi.wikipedia.org/wiki/B%E1%BA%A3n_danh_s%C3%A1ch_c%E1%BB%A7a_Schindler). Chúng không tốt hoàn toàn như bạn nghĩ. Các comment, tốt nhất, nên trở thành sự lựa chọn cuối cùng. Nếu ngôn ngữ lập trình của chúng ta có đầy đủ khả năng diễn đạt, hoặc nếu chúng ta có đủ tài năng sử dụng code để thể hiện hết ý định của mình thì chúng ta đã không cần đến những dòng comment.

Việc dùng đúng các comment là một cách để bù đắp cho sự thất bại của chúng ta trong việc thể hiện ý nghĩa của những dòng code. Hãy lưu ý rằng tôi đã dùng từ thất bại. Chính xác là vậy – comment luôn luôn là sự thất bại. Chúng ta phải có chúng vì chúng ta không thể thể hiện hết ý nghĩa của code, và việc sử dụng chúng không phải là nguyên nhân được tán dương.

Vậy nên, khi bạn thấy mình cần viết comment, hãy suy nghĩ kỹ xem liệu có cách nào đó để biến các dòng code thành chính xác những gì bạn muốn thể hiện hay không. Mỗi khi bạn thể hiện chính xác ý nghĩa của code, hãy tự khích lệ mình. Mỗi khi bạn viết comment, bạn nên tự vả vào mặt cho sự thất bại đó .

Tại sao tôi lại thất vọng về các comment? Vì chúng đầy dối trá. Không phải lúc nào cũng vậy, nhưng vấn đề này lại rất thường xuyên xãy ra: Comment càng cũ và càng không liên quan tới code mà nó mô tả, càng có nhiều khả năng nó sai. Lý do rất đơn giản, các lập trình viên không thể bảo trì chúng.

Code của dự án thì thay đổi và phát triển liên tục. Các khối dữ liệu di chuyển từ nơi này đến nơi kia. Những khối đó tách ra và tái lập để tạo nên chương trình. Thật không may, những comment không phải lúc nào cũng theo kịp chúng. Comment thì thường xuyên bị tách ra khỏi code mà nó mô tả và trở nên đứt đoạn với độ chính xác giảm dần. Ví dụ, hãy xem những gì xãy ra với comment này và dòng code mà nó dự định mô tả:

```
MockRequest request;
private final String HTTP_DATE_REGEXP =
 "[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s"+
 "[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT";
private Response response;
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;
// Example: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Các biến khác có thể được thêm vào sau đó đặt xen kẽ vào giữa giá trị `HTTP_DATE_REGEXP`. Giá trị `HTTP_DATE_REGEXP` được viết lại và comment của bạn thì tan nát. ☺

Có quan điểm cho rằng các lập trình viên nên có đủ kỷ luật để sửa cả những dòng comment khi họ đã thay đổi code. Tôi đồng ý, họ nên như thế. Nhưng tôi sẽ chọn cách làm cho code rõ ràng dễ hiểu, đến mức nó không cần các comment ngay từ đầu.

Những dòng comment bậy còn tồi tệ hơn là không comment. Chúng lừa gạt và đánh lạc hướng người đọc. Chúng đưa ra những dự tính không bao giờ được thực hiện. Chúng đặt ra những quy tắc cũ, không cần, hoặc không nên được tuân theo nữa.

Chân lý chỉ có thể tìm thấy tại một nơi duy nhất: Code. Chỉ có code mới nói cho bạn biết thật sự những gì nó làm. Nó là nguồn thông tin chính xác duy nhất. Do đó, mặc dù comment đôi khi là cần thiết, nhưng chúng tôi sẽ cố gắng để giảm thiểu nó.

Đừng dùng comment để làm màu cho code

Một trong những động lực to lớn để viết comment là do code tồi, code tối nghĩa. Chúng ta viết một hàm và chúng ta biết nó khó hiểu, nó vô tổ chức, chúng ta biết nó là một đống hỗ lốn. Vậy nên chúng ta tự nhủ rằng: "Ồ, tốt hơn nên viết comment ở đây!". Không! Tốt hơn bạn nên viết lại code!

Code sáng nghĩa và rõ ràng với ít comment sẽ tuyệt vời hơn so với code tối nghĩa, phức tạp với nhiều comment. Thay vì dành thời gian để viết comment giải thích mớ code hỗ lốn đó, hãy dành thời gian để dọn dẹp nó.

Giải thích ý nghĩa ngay trong code

Chắc chắn có những lúc code của bạn rất khó để giải thích nó đang làm gì. Thật không may, điều này làm cho nhiều lập trình viên cho rằng code hiếm khi là một cách tốt để giải thích. Điều đó hoàn toàn sai. Bạn muốn nhìn thấy điều gì? Cái này:

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
(employee.age > 65))
```

Hay cái này?

```
if (employee.isEligibleForFullBenefits())
```

Chỉ mất vài giây suy nghĩ để truyền hết thông điệp của bạn vào code. Trong nhiều trường hợp, nó chỉ đơn giản là tạo ra một hàm có tên giống với comment mà bạn muốn viết

Good Comments

Một số comment là cần thiết hoặc có ích. Chúng ta sẽ xem xet một vài trường hợp mà tôi cho là xứng đáng để bạn bỏ công ra viết. Tuy nhiên, hãy nhớ rằng comment thật sự tốt là comment không cần phải viết ra.

Comment pháp lý

Đôi khi các tiêu chuẩn mã hóa doanh nghiệp buộc chúng ta phải viết một số comment nhất định vì lý do pháp lý. Ví dụ, tuyên bố bản quyền và quyền tác giả là những điều cần thiết và hợp lý để đưa vào comment khi bắt đầu mỗi source code file.

```
/* Copyright (C) 2003,2004,2005 by Object Mentor, Inc.
 * All rights reserved.
 * Released under the terms of the GNU General Public License version
 * 2 or later.
 */
```

Đừng đưa cả hợp đồng hay những điều luật vào comment. Nếu có thể, hãy tham khảo một vài giấy phép tiêu chuẩn hay tài liệu bên ngoài khác thay vì đưa tất cả những điều khoản và điều kiện vào.

Comment cung cấp thông tin

Cung cấp thông tin cơ bản với một vài dòng comment đôi khi rất hữu ích. Ví dụ, hãy xem cách mà comment này giải thích về giá trị trả về của một phương thức trừu tượng:

```
// Returns an instance of the Responder being tested.
protected abstract Responder responderInstance();
```

Một comment như thế này, đôi khi, có thể là hữu ích. Nhưng tốt hơn là sử dụng tên của hàm để truyền đạt thông tin nếu có thể. Ví dụ, trong trường hợp này, chúng ta có thể dọn dẹp comment trên bằng cách đặt lại tên hàm thành `responderBeingTested`.

Trường hợp dưới đây có vẻ tốt hơn một chút:

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
  "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

Trong trường hợp này, comment cho chúng ta biết rằng biểu thức được tạo ra khớp với thời gian và ngày tháng, và được định dạng bằng hàm `SimpleDateFormat.format`. Tuy nhiên, nó có thể rõ ràng hơn nếu code này được chuyển sang một lớp đặc biệt chuyển đổi định dạng của ngày và thời gian. Và sau đó, bạn có thể đưa comment trên vào sọt rác.

Giải thích mục đích

Đôi khi comment không chỉ cung cấp thông tin về những dòng code mà còn cung cấp ý định đằng sau nó. Trong trường hợp sau đây, chúng tôi thấy một comment ghi lại một quyết định thú vị: khi so sánh hai đối tượng, tác giả quyết định rằng anh ta muốn sắp xếp các đối tượng của lớp mình *luôn* cao hơn các đối tượng khác.

```
public int compareTo(Object o)
{
    if(o instanceof WikiPagePath)
    {
        WikiPagePath p = (WikiPagePath) o;
        String compressedName = StringUtil.join(names, "");
        String compressedArgumentName = StringUtil.join(p.names, "");
        return compressedName.compareTo(compressedArgumentName);
```

```
    }
    return 1; // we are greater because we are the right type.
}
```

Dưới đây là một ví dụ tốt hơn. Có thể bạn không đồng tình với cách giải quyết vấn đề của tác giả, nhưng ít ra bạn biết được anh ấy đang cố gắng làm gì

```
public void testConcurrentAddWidgets() throws Exception {
    WidgetBuilder widgetBuilder = new WidgetBuilder(new
Class[]{BoldWidget.class});
    String text = "'''bold text'''";
    ParentWidget parent =
    new BoldWidget(new MockWidgetRoot(), "'''bold text'''");
    AtomicBoolean failFlag = new AtomicBoolean();
    failFlag.set(false);
    //This is our best attempt to get a race condition
    //by creating large number of threads.
    for (int i = 0; i < 25000; i++) {
        WidgetBuilderThread widgetBuilderThread =
        new WidgetBuilderThread(widgetBuilder, text, parent,
                                          failFlag);
        Thread thread = new Thread(widgetBuilderThread);
        thread.start();
    }
    assertEquals(false, failFlag.get());
}
```

> *Race condition* là gì: https://en.wikipedia.org/wiki/Race_condition
> Hoặc link tiếng việt: https://vi.wikipedia.org

Làm dễ hiểu

Đôi khi bạn cần dùng comment để diễn giải ý nghĩa của các đối số khó hiểu hoặc giá trị trả về, để biến chúng thành thứ gì đó có thể hiểu được. Nhưng tốt nhất vẫn là tìm cách làm cho các đối số hoặc giá trị trả về đó trở nên rõ ràng theo cách của bạn. Nhưng khi nó là một phần của thư viện, hoặc thuộc về một phần code mà bạn không có quyền tùy chỉnh, thì một comment giải thích dễ hiểu có thể có ích trong trường hợp này.

```
public void testCompareTo() throws Exception
{
    WikiPagePath a = PathParser.parse("PageA");
    WikiPagePath ab = PathParser.parse("PageA.PageB");
    WikiPagePath b = PathParser.parse("PageB");
    WikiPagePath aa = PathParser.parse("PageA.PageA");
    WikiPagePath bb = PathParser.parse("PageB.PageB");
```

```
    WikiPagePath ba = PathParser.parse("PageB.PageA");

    assertTrue(a.compareTo(a) == 0); // a == a
    assertTrue(a.compareTo(b) != 0); // a != b
    assertTrue(ab.compareTo(ab) == 0); // ab == ab
    assertTrue(a.compareTo(b) == -1); // a < b
    assertTrue(aa.compareTo(ab) == -1); // aa < ab
    assertTrue(ba.compareTo(bb) == -1); // ba < bb
    assertTrue(b.compareTo(a) == 1); // b > a
    assertTrue(ab.compareTo(aa) == 1); // ab > aa
    assertTrue(bb.compareTo(ba) == 1); // bb > ba
}
```

Dĩ nhiên, khả năng các comment dạng này cung cấp thông tin không chính xác là khá cao. Xem lại các ví dụ trước để thấy rằng việc xác nhận thông tin từ comment khó thế nào. Điều này giải thích lý do tại sao làm cho comment dễ hiểu là cần thiết, mặc dù điều đó đồng nghĩa với sự mạo hiểm. Vậy nên trước khi bạn viết những comment như thế này, hãy chắc chắn rằng không còn cách nào tối ưu hơn, vì sau đó bạn cần quan tâm đến độ chính xác của chúng mỗi khi bạn chỉnh sửa lại code.

Các cảnh báo về hậu quả

Đôi khi nó rất hữu ích để cảnh báo các lập trình viên khác về hậu quả xảy ra. Ví dụ, đây là một comment giải thích tại sao test case này lại bị tắt:

```
// Don't run unless you
// have some time to kill - Đừng chạy hàm này, trừ khi mày quá rảnh
public void _testWithReallyBigFile() {
    writeLinesToFile(10000000);
    response.setBody(testFile);
    response.readyToSend(this);
    String responseString = output.toString();
    assertSubString("Content-Length: 1000000000", responseString);
    assertTrue(bytesSent > 1000000000);
}
```

Tất nhiên là ngày nay, chúng tôi tắt các test case bằng cách sử dụng thuộc tính `@Ignore` với một chuỗi giải thích thích hợp: `@Ignore ("It takes too long to run")`. Nhưng trước khi JUnit 4 xuất hiện, việc đặt một dấu gạch dưới vào trước tên hàm là một quy tắc rất phổ biến. Các comment đồng thời được xem như là cách đánh dấu để lập trình viên chú ý đến cảnh báo của hàm hơn.

Đây là một ví dụ đau khổ khác:

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
    //SimpleDateFormat is not thread safe,
    //so we need to create each instance independently.
```

```
    SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy
HH:mm:ss z");
    df.setTimeZone(TimeZone.getTimeZone("GMT"));
    return df;
}
```

Có thể còn nhiều cách tốt hơn. Tôi đồng ý. Nhưng comment trong trường hợp này là hoàn toàn hợp lý, nó sẽ ngăn được một số lập trình viên ham hố sử dụng một phương thức khởi tạo tĩnh.

TODO comments

Đôi khi việc để lại các dòng comment dạng `//TODO` là điều cần thiết. Trong trường hợp dưới đây, comment dạng TODO giải thích tại sao hàm này lại là hàm suy biến, và tương lai của hàm sẽ như thế nào:

```
// TODO-MdM these are not needed - Hàm này không cần thiết
// We expect this to go away when we do the checkout model
// Nó sẽ bị xóa khi chúng tôi thực hiện mô hình thanh toán
protected VersionInfo makeVersion() throws Exception
{
    return null;
}
```

Những comment dạng TODO là những công việc mà lập trình viên cho rằng nên được thực hiện, nhưng vì lý do nào đó mà họ không thể thực hiện nó ngay lúc này. Nó có thể là một lời nhắc để xóa một hàm không dùng nữa, hoặc yêu cầu người khác xem xét một số vấn đề: đặt lại một cái tên khác tốt hơn, lời nhắc thay đổi code của hàm khi kế hoạch của dự án thay đổi,.. Nhưng dù TODO có là gì đi nữa, nó chắc chắn không phải là lý do để bạn quăng đống code ẩu, code bừa vào dự án.

Ngày nay, hầu hết các IDE đều cung cấp các tính năng đặc biệt để định vị các comment TODO, do đó bạn không cần lo việc bỏ quên/lạc mất nó. Tuy vậy, bạn sẽ không muốn code của bạn bị lấp đầy bởi TODO. Vậy nên hãy thường xuyên để mắt tới chúng và dọn dẹp chúng ngay khi có thể.

Khuếch đại

Comment có thể được dùng để khuếch đại tầm quan trọng của một cái gì đó có vẻ không quan trọng:

```
String listItemContent = match.group(3).trim();
// the trim is real important. It removes the starting
// spaces that could cause the item to be recognized
// as another list.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

Javadocs in Public APIs

Không có gì hữu ích và tuyệt vời bằng một public API được mô tả tốt. Các javadoc của thư viện chuẩn của Java là một trường hợp điển hình. Sẽ rất khó để viết các chương trình Java mà thiếu chúng.

Nếu đang viết một public API, chắc chắn bạn nên viết javadoc tốt cho nó. Nhưng hãy ghi nhớ những lời khuyên còn lại của chương này, các javadoc có thể là một cú lừa, hoặc mang lại phiền toái như bất kỳ comment nào khác .

Bad Comments

Đa số các comment rơi vào thể loại này. Chúng thường được sử dụng như cái cớ cho việc viết code rởm hoặc biện minh cho các cách giải quyết đầy thiếu sót, các giá trị không đầy đủ so với cách lập trình viên nghĩ về nó.

Độc thoại

Quăng vào một comment chỉ vì bạn thấy *thích*, hoặc chỉ vì quá trình xử lý cần đến nó, điều đó được gọi là *hack* (giải quyết vấn đề không theo cách thường mà dùng thủ thuật, đường tắt,...). Nếu bạn quyết định viết comment, hãy dành thời gian cho nó để đảm bảo đó là comment tốt nhất mà bạn có thể viết.

Dưới đây là một ví dụ tôi tìm thấy trong FitNesse, comment này có thể hữu ích. Nhưng tác giả đã vội vàng hoặc đã không quan tâm nhiều đến nó. Cách anh ta độc thoại làm cho người đến sau cảm thấy hoang mang:

```
public void loadProperties()
{
    try
    {
        String propertiesPath = propertiesLocation + "/" +
PROPERTIES_FILE;
        FileInputStream propertiesStream = new
FileInputStream(propertiesPath);
        loadedProperties.load(propertiesStream);
    }
    catch(IOException e)
    {
        // No properties files means all defaults are loaded
    }
}
```

Comment trong khối `cacth` trên nghĩa là gì? Hẳn là nó có ý nghĩa gì đó đối với tác giả, nhưng lại không được diễn giải đầy đủ. Rõ ràng, nếu `IOExeption` xãy ra, điều đó có nghĩa là không có thuộc tính file, và trong trường hợp này các thuộc tính mặc định sẽ được tải. Nhưng, cái gì sẽ tải thuộc tính

mặc định? Những thuộc tính mặc định đó đã được tải trước khi gọi `loadProperties()`? Hay chúng được tải khi `loadedProperties.load(propertiesStream)` phát sinh Exception? Hay chúng được tải sau khi gọi `loadProperties()` ? Có phải tác giả chỉ đang cố làm hài lòng chính bản thân anh ta về việc bỏ trống khối `catch`? Hoặc, kinh khủng hơn – tác giả đã dùng comment như một dấu hiệu, để sau này quay lại và viết các đoạn code tải các thuộc tính mặc định vào khối `catch`?

Cách duy nhất là kiểm tra code của hệ thống để tìm hiểu những gì xảy ra. Bất kỳ comment nào khiến bạn phải lục tung code của hệ thống lên để tìm hiểu thì comment đó không truyền tải tốt thông tin cho bạn, và nó không xứng đáng với số bit nó chiếm trong mã nguồn.

Các comment thừa thải

Listing 4-1 cho thấy một hàm đơn giản với các comment ở đầu hoàn toàn thừa thải. Comment này, có lẽ, còn làm người đọc mất thời gian hơn so với việc đọc code của hàm.

Listing 4-1
waitForClose

```
// Utility method that returns when this.closed is true. Throws an
exception
// if the timeout is reached.
// Phương thức return khi this.closed là true, phát sinh ngoại lệ nếu
hết thời gian chờ
public synchronized void waitForClose(final long timeoutMillis)
 throws Exception
{
    if(!closed)
    {
        wait(timeoutMillis);
        if(!closed)
            throw new Exception("MockResponseSender could not be
                                 closed");
    }
}
```

Comment này nhằm mục đích gì? Nó chắn chắn không cung cấp nhiều thông tin hơn code. Nó không diễn giải cho code, không cung cấp mục đích hoặc lý do. Nó không dễ hiểu hơn code. Sự thật là, nó ít chính xác hơn code nhưng lại lôi cuốn người đọc chấp nhận sự thiếu chính xác đó thay cho hiểu biết thật sự.

Bây giờ hãy xem xét vô số comment vô ích và dư thừa của javadocs trong Listing 4-2 được lấy từ Tomcat. Những comment này chỉ để làm lộn xộn và làm code thêm khó hiểu. Chúng tồn tại nhưng không vì mục đích cung cấp thông tin mà chỉ khiến mọi thứ tệ hơn. Tôi chỉ chỉ ra cho bạn được một vài dòng đầu tiên, còn nhiều hơn nữa nếu bạn xem qua hết module này.

Listing 4-2

ContainerBase.java (Tomcat)

```
public abstract class ContainerBase
 implements Container, Lifecycle, Pipeline,
 MBeanRegistration, Serializable {
    /**
     * The processor delay for this component.
     */
    protected int backgroundProcessorDelay = -1;
    /**
     * The lifecycle event support for this component.
     */
    protected LifecycleSupport lifecycle = new LifecycleSupport(this);
    /**
     * The container event listeners for this Container.
     */
    protected ArrayList listeners = new ArrayList();
    /**
     * The Loader implementation with which this Container is
     * associated.
     */
    protected Loader loader = null;
    /**
     * The Logger implementation with which this Container is
     * associated.
     */
    protected Log logger = null;
    /**
     * Associated logger name.
     */
    protected String logName = null;
    /**
     * The Manager implementation with which this Container is
     * associated.
     */
    protected Manager manager = null;
    /**
     * The cluster with which this Container is associated.
     */
    protected Cluster cluster = null;
    /**
     * The human-readable name of this Container.
     */
    protected String name = null;
    /**
     * The parent Container to which this Container is a child.
     */
    protected Container parent = null;
```

```
    /**
     * The parent class loader to be configured when we install a
     * Loader.
     */
    protected ClassLoader parentClassLoader = null;
    /**
     * The Pipeline object with which this Container is
     * associated.
     */
    protected Pipeline pipeline = new StandardPipeline(this);
    /**
     * The Realm with which this Container is associated.
     */
    protected Realm realm = null;
    /**
     * The resources DirContext object with which this Container
     * is associated.
     */
    protected DirContext resources = null;
```

Các comment sai sự thật

Đôi khi, với mục đích hoàn toàn trong sáng, một lập trình viên diễn đạt ý định của anh ta trong comment, nhưng nó lại không đủ chính xác. Hãy dành một ít thời gian để xem lại comment dư thừa và sai lệch trong Listing 4-1.

Bạn có phát hiện ra comment ở Listing 4-1 sai lệch như thế nào chưa? Phương thức không `return` *khi* `this.closed` là `true`, nó return *nếu* `this.closed` là `true`. Mặt khác, nó giả vờ chờ một khoản thời gian và sinh ra một Exception *nếu* `this.closed` vẫn chưa là `true`.

Một chút thông tin sai lệch, ẩn nấp trong một comment khó đọc hơn cả code mà nó diễn giải. Điều này có thể khiến một lập trình viên khác gọi hàm này và mong muốn nó `return` ngay khi `this.closed` trở thành `true`. Sau đó anh ta phải debug chương trình để tìm câu trả lời cho việc nó thực thi quá chậm.

Các comment bắt buộc

Thật điên rồ khi cho rằng tất cả các hàm đều phải có javadoc, hoặc mọi biến đều phải có comment. Những comment như vậy chỉ làm rối code, đưa ra những lời bịa đặt, và ủng hộ việc gây nhầm lẫn và vô tổ chức.

Ví dụ, javadoc được yêu cầu cho mọi hàm dẫn đến sự kinh khủng khiếp như Listing 4-3. Tình trạng lộn xộn này không giúp ích được gì mà ngược lại,nó chỉ làm xáo trộn code và ngầm tạo ra những cú lừa khác,...

Listing 4-3

```
/**
 *
 * @param title The title of the CD
 * @param author The author of the CD
 * @param tracks The number of tracks on the CD
 * @param durationInMinutes The duration of the CD in minutes
 */
public void addCD(String title, String author,
 int tracks, int durationInMinutes) {
    CD cd = new CD();
    cd.title = title;
    cd.author = author;
    cd.tracks = tracks;
    cd.duration = duration;
    cdList.add(cd);
}
```

Các comment nhật ký

Đôi khi mọi người thêm một comment vào đầu module mỗi khi họ chỉnh sửa nó. Những comment này tích lũy như một loại nhật ký lưu lại mọi thay đổi đã từng được thực hiện. Tôi đã thấy một số module với hàng trăm dòng như thế này:

```
/*
 * Changes (from 11-Oct-2001)
 * --------------------------
 * 11-Oct-2001 : Re-organised the class and moved it to new package
 * com.jrefinery.date (DG);
 * 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
 * class (DG);
 * 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
 * class is gone (DG); Changed getPreviousDayOfWeek(),
 * getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
 * bugs (DG);
 * 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
 * 29-May-2002 : Moved the month constants into a separate interface
 * (MonthConstants) (DG);
 * 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
 * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 * 13-Mar-2003 : Implemented Serializable (DG);
 * 29-May-2003 : Fixed bug in addMonths method (DG);
 * 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);
 * 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
 */
```

Ngày xửa ngày xưa, chúng tôi có lý do chính đáng để tạo và cập nhật các nhật ký này khi bắt đầu một module. Ở thời điểm đó, các hệ thống quản lý mã nguồn (source control) chưa xuất hiện và chúng tôi phải thực hiện điều này. Nhưng ngày nay, những nhật ký này chỉ làm module lộn xộn hơn. Chúng nên được loại bỏ hoàn toàn.

Các comment gây nhiễu

Đôi khi bạn gặp các comment không cung cấp thông tin gì ngoài sự phiền phức, chúng lặp lại một vấn đề hiển nhiên và không cung cấp thêm thông tin mới

```
/**
 * Default constructor.
 */
protected AnnualDateRule() {
}
```

Rìa-lý? Còn cái này:

```
/** The day of the month. */
private int dayOfMonth;
```

Và sau đó là sự thừa thãi này:

```
/**
 * Returns the day of the month.
 *
 * @return the day of the month.
 */
public int getDayOfMonth() {
    return dayOfMonth;
}
```

Những comment này phiền đến mức chúng tôi phải học cách bỏ qua chúng. Khi chúng tôi đọc code, mắt chúng tôi chỉ lướt qua chúng. Cuối cùng, các comment bắt đầu sai lệch khi code xung quanh chúng thay đổi, còn chúng thì không – ai lại để ý đến chúng chứ.

Comment đầu tiên trong Listing 4-4 có vẻ hợp lý, nó giải thích tại sao khối `catch` được bỏ qua. Nhưng comment thứ hai chỉ đơn thuần là gây nhiễu cho người đọc. Rõ ràng lập trình viên này đã rất mệt mỏi với việc viết các khối `try/catch` và anh ta đang trút giận.

Listing 4-4
startSending

```
private void startSending()
{
    try
    {
        doSending();
```

```
    }
    catch(SocketException e)
    {
        // normal. someone stopped the request.
    }
    catch(Exception e)
    {
        try
        {
            response.add(ErrorResponder.makeExceptionString(e));
            response.closeAll();
        }
        catch(Exception e1)
        {
            //Give me a break! – Đưa tao ra khỏi đây!
        }
    }
}
```

Thay vì trút giận vào một comment vô giá trị và gây bối rối cho người đọc, lập trình viên nên nhận ra rằng vấn đề của anh ta có thể được giải quyết bằng cách tái cấu trúc lại phần code đó. Anh ta nên sử dụng năng lượng của mình cho việc chuyển khối `try/catch` đó thành một hàm riêng biệt, như trong Listing 4-5

Listing 4-5
startSending (refactored)

```
private void startSending()
{
    try
    {
        doSending();
    }
    catch(SocketException e)
    {
        // normal. someone stopped the request.
    }
    catch(Exception e)
    {
        addExceptionAndCloseResponse(e);
    }
}
private void addExceptionAndCloseResponse(Exception e)
```

```
{
    try
    {
        response.add(ErrorResponder.makeExceptionString(e));
        response.closeAll();
    }
    catch(Exception e1)
    {
    }
}
```

Tránh việc viết comment gây nhiễu bằng quyết tâm làm sạch code. Bạn sẽ thấy điều đó làm cho bạn trở thành một lập trình viên đẳng cấp hơn, và hạnh phúc hơn.

(Lại là) các comment gây nhiễu – nhưng đáng sợ hơn

Các javadoc cũng có thể trở nên phiền phức. Mục đích của các javadoc sau là gì (từ một thư viện mã nguồn mở nổi tiếng)? Trả lời: Không gì cả. Chúng chỉ là những comment dư thừa, bị sao chép lại như một khao khát cung cấp tài liệu cho lập trình viên.

```
/** The name. */
private String name;
/** The version. */
private String version;
/** The licenceName. */
private String licenceName;
/** The version. */
private String info;
```

Xem những comment trên một lần nữa, bạn có thấy lỗi cắt-dán không? Nếu tác giả đã không chú ý đến comment khi viết (hoặc dán) chúng, sao chúng ta có thể trông mong chúng giúp ích được chúng ta?

Đừng dùng comment khi bạn có thể dùng hàm hoặc biến

Hãy xem xét đoạn code sau:

```
// does the module from the global list <mod> depend on the
// subsystem we are part of?
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))
```

Chúng có thể được viết lại mà không cần comment:

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

Có thể tác giả đã viết comment trước (tôi đoán thế), và sau đó viết code để hoàn thành nội dung của comment. Tuy nhiên, tác giả nên tái cấu trúc code, như tôi đã làm, để comment được xóa mà không ảnh hưởng đến thông điệp của đoạn code đó.

Đánh dấu lãnh thổ

Thỉnh thoảng vài lập trình viên thích đánh dấu một vị trí trong tệp mã nguồn. Ví dụ, gần đây tôi đã thấy thứ này trong một chương trình tôi đang xem xét:

```
// Actions ////////////////////////////////
```

Rất hiếm khi các hàm có cùng chức năng được tập hợp bên dưới một câu miêu tả như thế này. Nhưng nói chung, chúng đang bừa bộn nên cần phải loại bỏ, đặc biệt là cả tá dấu slash ở cuối câu.

Theo cách nghĩ này, một comment đánh dấu sẽ gây chú ý mạnh nếu bạn không thường xuyên nhìn thấy chúng. Vì vậy, hãy hạn chế dùng, và chỉ dùng nếu chúng mang lại lợi ích đáng kể. Nếu bị lạm dụng, chúng sẽ bị người đọc cho vào vùng "phiền phức" và nhanh chóng bị bỏ qua.

Các comment kết thúc

Đôi khi các lập trình viên sẽ đưa ra các comment cụ thể về việc đóng dấu ngoặc nhọn, như trong Listing 4-6. Mặc dù điều này có thể có ý nghĩa với các hàm dài và có nhiều cấu trúc lồng nhau, nhưng chúng lại thường được dùng trong các hàm nhỏ, hoặc các hàm mà lập trình viên thấy thích. Vì vậy lần tới nếu muốn đánh dấu kết thúc một khối lệnh phức tạp, hãy thử chia nhỏ hoặc rút ngắn hàm của bạn.

Listing 4-6
wc.java

```
public class wc {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
        String line;
        int lineCount = 0;
        int charCount = 0;
        int wordCount = 0;
        try {
            while ((line = in.readLine()) != null) {
                lineCount++;
                charCount += line.length();
                String words[] = line.split("\\W");
```

```
                wordCount += words.length;
            } //while
            System.out.println("wordCount = " + wordCount);
            System.out.println("lineCount = " + lineCount);
            System.out.println("charCount = " + charCount);
        } // try
        catch (IOException e) {
            System.err.println("Error:" + e.getMessage());
        } //catch
    } //main
}
```

Thuộc tính và dòng tác giả

```
/* Added by Rick */
```

Hệ thống quản lý mã nguồn ghi nhớ rất tốt việc ai đã thêm gì, thêm khi nào,.. Không cần phải làm ô nhiễm code bằng những dòng tác giả. Bạn có thể nghĩ rằng những comment như vậy sẽ hữu ích, người khác sẽ biết ai đã làm việc với phần code đó. Nhưng thực tế là chúng có xu hướng ở lại đó trong nhiều năm, và ngày càng ít chính xác, và ít liên quan.

Nhắc lại lần nữa, hệ thống quản lý mã nguồn là nơi tốt hơn để lưu những thông tin này.

Comment-hóa code

Comment-hóa code (chuyển code thành comment) là một trong những thói quen khiến lập trình viên dễ bị ghét nhất. Đừng làm việc đó!

```
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(),
formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount()));
```

Những người khác khi thấy comment dạng này sẽ không đủ dũng cảm để xóa nó. Họ nghĩ rằng nó nằm đó vì một lý do nào đó, và chắc hẳn là nó quan trọng.

Hãy xem xét đoạn code dưới đây:

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
```

```
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
    writeEnd();
    this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
}
else {
    this.pngBytes = null;
}
return this.pngBytes;
```

Tại sao lại có hai dòng code bị chuyển thành comment? Chúng có quan trọng không? Có phải chúng để lại một lời nhắc cho một số thay đổi sắp xảy ra? Hay chúng chỉ là một dòng comment đã tồn tại từ nhiều năm trước và đơn giản là không ai thèm dọn dẹp?

Đã từng có một thời gian, vào những năm 60, khi những comment dạng này có ích. Nhưng hiện tại hệ thống quản lý mã nguồn đang và sẽ tiếp tục hoạt động tốt. Hệ thống quản lý mã nguồn sẽ nhớ code cho tôi và bạn, và chắc chắn không có lý do gì để comment-hóa một dòng code. Hãy xóa dòng code đó, chúng ta sẽ không bị mất nó. Tôi thề đấy.

HTML comment

Đưa HTML vào comment là một hành động xúc phạm lập trình viên. Như cách mà bạn thấy bên dưới. Nó làm comment khó đọc trực tiếp nhưng lại dễ đọc hơn trên trình soạn thảo hoặc IDE khác. Nếu các comment được trích xuất bởi công cụ và hiển thị như một trang web, việc tô vẽ cho comment bằng HTML là trách nhiệm của công cụ đó, chứ không phải của lập trình viên.

```
/**
 * Task to run fit tests.
 * This task runs fitnesse tests and publishes the results.
 * <p/>
 * <pre>
 * Usage:
 * &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
 * classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot;
 * classpathref=&quot;classpath&quot; /&gt;
 * OR
 * &lt;taskdef classpathref=&quot;classpath&quot;
 * resource=&quot;tasks.properties&quot; /&gt;
 * <p/>
 * &lt;execute-fitnesse-tests
 * suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;
 * fitnesseport=&quot;8082&quot;
```

```
 * resultsdir=&amp;quot;${results.dir}&amp;quot;
 * resultshtmlpage=&amp;quot;fit-results.html&amp;quot;
 * classpathref=&amp;quot;classpath&amp;quot; /&amp;gt;
 * </pre>
 */
```

Thông tin phi tập trung

Nếu bắt buộc phải viết một comment, hãy đảm bảo rằng nó giải thích cho phần code gần nó nhất. Đừng cung cấp thông tin của toàn bộ hệ thống trong một comment cục bộ. Xem xét ví dụ dưới đây: Dù thừa thải nhưng nó cung cấp thông tin về cổng mặc định. Tuy nhiên hàm này lại hoàn toàn không có quyền kiểm soát cổng mặc định. Comment không mô tả cho hàm, mà mô tả một phần nào đó trong hệ thống. Tất nhiên, không có gì đảm bảo comment này sẽ được thay đổi khi code mà nó mô tả thay đổi.

```
/**
 * Port on which fitnesse would run. Defaults to <b>8082</b>.
 *
 * @param fitnessePort
 */
public void setFitnessePort(int fitnessePort)
{
    this.fitnessePort = fitnessePort;
}
```

Quá nhiều thông tin

Đừng đưa các cuộc thảo luận hoặc mô tả không liên quan vào comment của bạn. Comment dưới đây được lấy ra từ một module có chức năng mã hóa và giải mã base64. Một người nào đó có thể đọc code này mà không cần những thông tin phức tạp từ comment.

```
/*
 RFC 2045 - Multipurpose Internet Mail Extensions (MIME)
 Part One: Format of Internet Message Bodies
 section 6.8. Base64 Content-Transfer-Encoding
 The encoding process represents 24-bit groups of input bits as
output
 strings of 4 encoded characters. Proceeding from left to right, a
 24-bit input group is formed by concatenating 3 8-bit input groups.
 These 24 bits are then treated as 4 concatenated 6-bit groups, each
 of which is translated into a single digit in the base64 alphabet.
 When encoding a bit stream via the base64 encoding, the bit stream
 must be presumed to be ordered with the most-significant-bit first.
 That is, the first bit in the stream will be the high-order bit in
```

```
 the first 8-bit byte, and the eighth bit will be the low-order bit
in
 the first 8-bit byte, and so on.
 */
```

Mối quan hệ khó hiểu

Sự kết nối giữa comment và code mà nó mô tả nên rõ ràng. Nếu bạn gặp khổ sở khi phải viết comment, ít nhất bạn cũng muốn người khác nhìn vào comment và hiểu những gì nó đang nói về code.

Xem xét ví dụ sau:

```
/*
 * start with an array that is big enough to hold all the pixels
 * (plus filter bytes), and an extra 200 bytes for header info
 * bắt đầu với một mảng đủ lớn để chứa tất cả các pixel
 * (cộng với một số byte của bộ lọc), và thêm 200 byte cho tiêu đề
 */
 this.pngBytes = new byte[((this.width + 1) * this.height * 3) +
200];
```

Một byte của bộ lọc nghĩa là gì? Nó có liên quan gì đến +1 không? Hay *3? Hay cả hai? Tại sao lại là giá trị 200 mà không phải giá trị khác? Mục đích của comment là để giải thích code, không phải giải thích chính nó. Thật đáng tiếc khi comment lại cần một lời giải thích cho riêng mình.

Comment làm tiêu đề cho hàm

Các hàm ngắn không cần nhiều mô tả. Đặt một cái tên đủ tốt cho hàm thường tốt hơn việc đặt một comment ở trước hàm đó.

Javadocs trong code không công khai

[...] Việc tạo các trang javadoc cho các lớp và các hàm trong hệ thống thường không hữu ích, và thủ tục bổ sung các comment không khác gì sự lộn xộn và làm mất thời gian.

Ví dụ

Tôi đã viết module trong Listing 4-7 cho *XP Immersion* đầu tiên. Nó được dự định là một ví dụ về phong cách viết code xấu và comment xấu. Kent Back sau đó đã tái cấu trúc code này lại dưới hình thức dễ chịu hơn trước mặt hàng tá sinh viên. Sau đó tôi đã điều chỉnh chúng để phù hợp với quyển sách *Agile Software Development, Principles, Patterns, and Practices* và bài viết *Craftsman* đầu tiên đã được xuất bản trên tạp chí *Software Development*.

Điều tôi cảm thấy thú vị trong module này là, đã có lúc nhiều người trong chúng tôi xem nó là một *tài liệu tham khảo hay*. Còn bây giờ chúng ta xem nó là một mớ lộn xộn. Hãy xem có bao nhiêu vấn đề về comment mà bạn có thể tìm thấy.

Listing 4-7

GeneratePrimes.java

```
/**
 * This class Generates prime numbers up to a user specified
 * maximum. The algorithm used is the Sieve of Eratosthenes.
 * <p>
 * Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
 * d. c. 194, Alexandria. The first man to calculate the
 * circumference of the Earth. Also known for working on
 * calendars with leap years and ran the library at Alexandria.
 * <p>
 * The algorithm is quite simple. Given an array of integers
 * starting at 2. Cross out all multiples of 2. Find the next
 * uncrossed integer, and cross out all of its multiples.
 * Repeat untilyou have passed the square root of the maximum
 * value.
 *
 * @author Alphonse
 * @version 13 Feb 2002 atp
 */
import java.util.*;
public class GeneratePrimes
{
    /**
    * @param maxValue is the generation limit.
    */
    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue >= 2) // the only valid case
        {
            // declarations
            int s = maxValue + 1; // size of array
            boolean[] f = new boolean[s];
            int i;
            // initialize array to true.
            for (i = 0; i < s; i++)
            f[i] = true;
            // get rid of known non-primes
            f[0] = f[1] = false;
            // sieve
```

Listing 4-7
GeneratePrimes.java

```
            int j;
            for (i = 2; i < Math.sqrt(s) + 1; i++)
            {
                if (f[i]) // if i is uncrossed, cross its multiples.
                {
                    for (j = 2 * i; j < s; j += i)
                    f[j] = false; // multiple is not prime
                }
            }
            // how many primes are there?
            int count = 0;
            for (i = 0; i < s; i++)
            {
                if (f[i])
                count++; // bump count.
            }
            int[] primes = new int[count];
            // move the primes into the result
            for (i = 0, j = 0; i < s; i++)
            {
                if (f[i]) // if prime
                primes[j++] = i;
            }
            return primes; // return the primes
        }
        else // maxValue < 2
        return new int[0]; // return null array if bad input.
    }
}
```

Trong Listing 4-8, bạn có thể thấy một phiên bản được tái cấu trúc của module trên. Chú ý rằng việc sử dụng comment được hạn chế tối đa. Chỉ có hai comment trong toàn bộ module và chúng đều hợp lý.

Listing 4-8
GeneratePrimes.java

```
/**
```

Listing 4-8

GeneratePrimes.java

```
 * This class Generates prime numbers up to a user specified
 * maximum. The algorithm used is the Sieve of Eratosthenes.
 * Given an array of integers starting at 2:
 * Find the first uncrossed integer, and cross out all its
 * multiples. Repeat until there are no more multiples
 * in the array.
 */
public class PrimeGenerator
{
    private static boolean[] crossedOut;
    private static int[] result;
    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue < 2)
            return new int[0];
        else
        {
            uncrossIntegersUpTo(maxValue);
            crossOutMultiples();
            putUncrossedIntegersIntoResult();
            return result;
        }
    }
    private static void uncrossIntegersUpTo(int maxValue)
    {
        crossedOut = new boolean[maxValue + 1];
        for (int i = 2; i < crossedOut.length; i++)
            crossedOut[i] = false;
    }
    private static void crossOutMultiples()
    {
        int limit = determineIterationLimit();
        for (int i = 2; i <= limit; i++)
            if (notCrossed(i))
                crossOutMultiplesOf(i);
    }
    private static int determineIterationLimit()
    {
```

Listing 4-8

GeneratePrimes.java

```
        // Every multiple in the array has a prime factor that
        // is less than or equal to the root of the array size,
        // so we don't have to cross out multiples of numbers
        // larger than that root.
        double iterationLimit = Math.sqrt(crossedOut.length);
        return (int) iterationLimit;
    }
    private static void crossOutMultiplesOf(int i)
    {
        for (int multiple = 2*i;
                multiple < crossedOut.length;
                multiple += i)
            crossedOut[multiple] = true;
    }
    private static boolean notCrossed(int i)
    {
        return crossedOut[i] == false;
    }
    private static void putUncrossedIntegersIntoResult()
    {
        result = new int[numberOfUncrossedIntegers()];
        for (int j = 0, i = 2; i < crossedOut.length; i++)
            if (notCrossed(i))
                result[j++] = i;
    }
    private static int numberOfUncrossedIntegers()
    {
        int count = 0;
        for (int i = 2; i < crossedOut.length; i++)
            if (notCrossed(i))
                count++;
        return count;
    }
}
```

Thật dễ để chỉ ra comment đầu tiên là dư thừa vì nó đọc rất giống hàm `generatePrimes`. Tuy nhiên, tôi nghĩ rằng comment giúp người đọc dễ tiếp cận thuật toán, vì vậy tôi có xu hướng để lại nó.

Comment thứ hai chắc chắn là cần thiết. Nó giải thích lý do đằng sau việc sử dụng giá trị căn bậc hai làm giới hạn vòng lặp. Tôi không thể tìm thấy tên biến đơn giản hay cách thức khác để làm rõ điểm này. Mặt khác, việc sử dụng căn bậc hai có thể để lại vài vấn đề. Tôi có tiết kiệm được nhiều thời gian khi dùng giá trị căn bậc hai làm giới hạn cho vòng lặp hay không? Thời gian tiết kiệm được có nhiều hơn thời gian tính căn bậc hai hay không?

Hmm, thật đáng để suy nghĩ. Sử dụng căn bậc hai làm giới hạn vòng lặp đem lại thỏa mãn cho máu hacker trong tôi, nhưng tôi không nghĩ người khác cũng có máu đó. Một comment ở vị trí đó sẽ giúp người đọc dễ dàng hiểu nó hơn.

Tham khảo

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