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.