Chapter 1: Clean Code

Trước hết, bạn đọc cuốn sách này bởi hai lý do:
Đầu tiên, bạn là một lập trình viên.
Thứ hai, bạn muốn trở thành một lập trình viên tốt.

There Will Be Code

Người ta có thể cho rằng bằng cách nào đó về sau viết code sẽ không còn là vấn đề nữa; rằng chúng ta chỉ cần quan tâm đến các Mô hình và các Yêu cầu là đủ. Thực tế một số người cho rằng thời đại của chúng ta đang tiến gần tới sự kết thúc của việc viết Code. Lập trình viên sẽ không cần thiết nữa bởi vì những khách hàng kinh doanh sẽ tự tạo ra các chương trình bằng cách nhập các thông số kỹ thuật.

Điều đó là phi lý! Chúng ta sẽ không bao giờ thoát ra khỏi các đoạn code, bởi vì code đại diện cho chi tiết của các yêu cầu. Chi tiết ở một vài mức độ không thể bỏ qua hay trừu tượng hóa được mà phải được đặc tả rõ ràng. Và đặc tả chi tiết các yêu cầu như vậy để máy tính có thể thực hiện được chúng thì chính là việc lập trình, cụ thể ở đây là việc viết code.

Vẫn hy vọng rằng mức độ trừu tượng của các ngôn ngữ lập trình sẽ tiếp tục tăng, có thêm nhiều ngôn ngữ cụ thể hơn nữa. Đó là điều tốt, tuy nhiên nó không thể loại bỏ việc viết code. Dù có các công cụ hỗ trợ nhưng nó vẫn phải được thực hiện một cách nghiêm túc, chính xác để máy tính có thể hiểu được và thực hiện đúng các chi tiết.

Nhiều người nghĩ rằng code một ngày nào đó sẽ biến mất, máy tính có thể hiểu được những yêu cầu mơ hồ và thực hiện chính xác một cách hoàn hảo đáp ứng những nhu cầu đó. Họ vẫn mơ rằng sẽ có một ngày chúng ta sẽ tạo ra được một cỗ máy có thể làm được những gì chúng ta muốn thay vì những gì chúng ta nói.

Điều đó là phi thực tế!

Hãy nhớ rằng code thực sự là ngôn ngữ cuối cùng mà chúng ta thể hiện các yêu cầu. Chúng ta có thể tạo ra các ngôn ngữ gần với yêu cầu, tạo ra các công cụ giúp chúng ta phân tích và tập hợp những yêu cầu thành cấu trúc chính thức. Nhưng chúng ta sẽ không bao giờ loại bỏ code.

Bad Code

Tác giả nói về việc đọc lời tựa trong cuốn sách Implementation Patterns của Kent Beck: "... Sách được viết dựa theo một giả thiết khá mong manh: Đó là vấn đề về good code" và phủ định nó, cho rằng good code là tiền đề trong những cơ sở quan trọng nhất của lĩnh vực lập trình. Đội ngũ của tác giả đã phải đối phó với vấn đề thiếu thốn good code quá lâu rồi.

Tiếp theo là một ví dụ về một ứng dụng (killer), nó rất phổ biến, được rất nhiều chuyên gia mua và sử dụng. Một chương trình hot như Ngọc Trinh nhưng cũng "não ngắn" như Ngọc Trinh, thành ra chương trình càng ngày càng sa sút sau mỗi đợt phát hành, đợt sau tàn tạ hơn đợt trước đến nỗi tác giả phải shutdown và say goodbye. Và bầu sô của Ngọc Trinh cũng sập tiệm một thời gian sau đó.

Hai thập kỷ sau, Robert C.Martin gặp lại ekip lăng xê của Ngọc Trinh và có hỏi tại sao em nó tàn tạ đến như vậy. Câu trả lời khẳng định lại nỗi lo sợ của tác giả ở trên. Họ vội vã lăng xê sản phẩm ra thị trường và thực hiện trong nó một đống lộn xộn các code. Sau mỗi lần update mông má thêm thì như ị thêm shit vào bãi rác, và bãi rác thành một bãi bự hơn. Khi đó thì công nhân môi trường không thể dọn dẹp được nữa. Bãi rác đó kéo công ty đi xuống.

Đã bao giờ bạn đâm đầu vào mã bẩn? Chắc chắn nếu bạn là một lập trình viên có kinh nghiệm, bạn đã rất nhiều lần cụng đầu vào nó. Chính xác hơn là lội qua mã bẩn. Bơi qua một bãi lầy đầy bụi gai và cạm bẫy ẩn hiện, tự tìm đường đi và hy vọng có một ít gợi ý, một số manh mối, về chuyện quái gì đang xảy ra, bơi trong đầm lầy nhưng không thấy đâu là bờ cuối cùng đành chấp nhận chết chìm trong nó.

Tất nhiên là bạn chết chìm trong nó - Vậy thì, cuối cùng tại sao lại xuất hiện cái đầm lầy đó?

Có thể bạn cố gắng làm nhanh? Vội vàng? Không có thời gian để làm tốt hơn vì deadline, sếp thúc đít, rằng sếp sẽ tức giận nếu bỏ thời gian làm sạch mà không làm cho xong. Có thể mệt mỏi muốn làm cho xong phứt đi rồi over. Thế nên dù code của bạn có tởm lợm như nào nhưng chạy không có lỗi gì cũng xem như xong rồi. Tự nhủ rằng sẽ quay lại và dọn dẹp nó sau nhưng

chúng ta không biết một điều, luật của LeBlanc: Later equals never (Để sau có nghĩa là không bao giờ).

The Total Cost of Owning a Mess

Code bẩn (Messy Code) sẽ làm cho công việc chậm lại một cách đáng kể. Lúc đầu, team thực hiện rất nhanh khi bắt đầu dự án. Mỗi lần thay đổi, lại phá vỡ vài phần khác của Code. Mỗi lần thêm hoặc sửa, hệ thống lại rối rắm, xoắn quẩy,... Qua thời gian thì sự lộn xộn đó lớn dần lên. Và không có cách nào dọn sạch nó.

Năng suất theo đó giảm dần và tiệm cận về 0 (Năng suất giảm -> Management thêm nhân viên -> Nhân viên mới không hiểu hệ thống -> Làm và làm -> Áp lực -> Tiêu diêu về miền cực lạc (R.I.P)...).

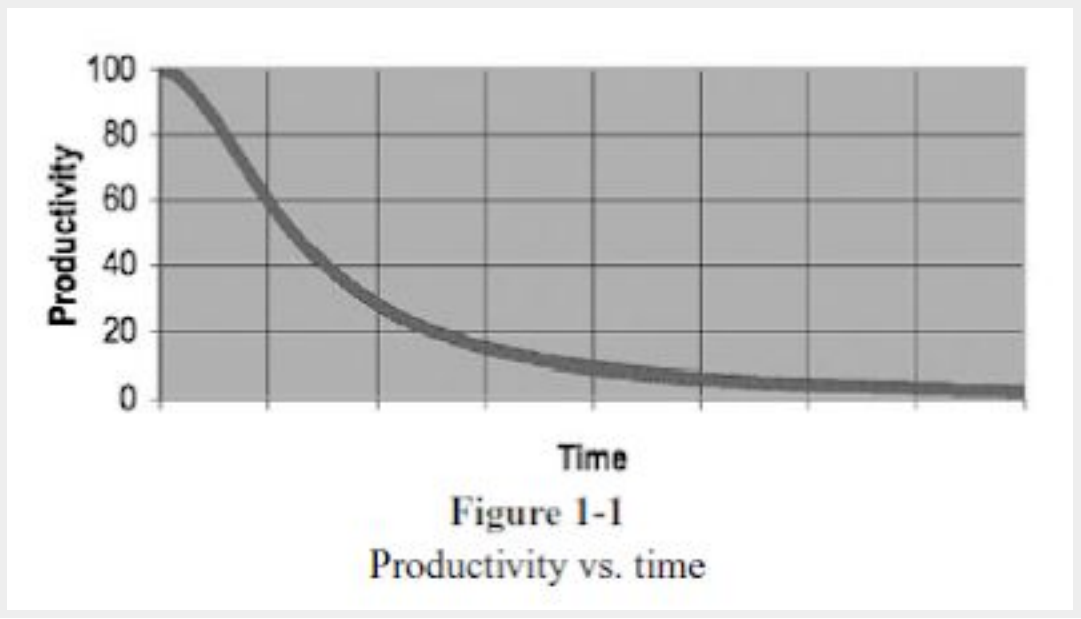


Figure 1-1
Productivity vs. time

The Grand Redesign in the Sky (Cuộc tái thiết vĩ đại giữa lưng chừng trời)

Chuyện về việc trong 10 năm, quản lý thực hiện việc phát triển song song dự án cũ và dự án mới thay thế cho dự án cũ vì Code bẩn.

Cuối cùng họ không chịu nổi nữa và thông báo với quản lý rằng họ không thể tiếp tục với mớ mã nguồn hỗn loạn này nữa. Họ yêu cầu được thiết kế lại. Quản lý không muốn phải dùng cạn chi phí cho việc thiết kế lại dự án, nhưng không thể phủ nhận rằng, năng suất hiện đang rất là khủng khiếp. Cuối cùng họ cũng nhượng bộ đội ngũ phát triển và đồng ý bắt đầu cuộc tái thiết vĩ đại.

Một team hổ mới được lựa chọn. Mọi người đều muốn vào team này bởi vì nó là một dự án mới toanh. Họ sẽ được bắt đầu lại từ đầu và làm một thứ gì đó thật tốt đẹp. Nhưng chỉ những người giỏi nhất và có triển vọng nhất được lựa chọn. Những người còn lại vẫn phải tiếp tục bảo trì hệ thống hiện tại.

Bây giờ thì cả hai team đều đang ở trong một cuộc đua. Team hổ phải xây dựng hệ thống mới làm được những gì mà hệ thống cũ đang làm. Không chỉ vậy, họ phải theo kịp những gì thay đổi đang tiếp tục diễn ra ở trên hệ thống cũ. Quản lý sẽ không thay đổi hệ thống cũ cho đến khi hệ thống mới có thể làm được tất cả mọi thứ mà hệ thống cũ làm.

Cuộc đua này đã diễn ra trong một thời gian rất dài. Nó đã mất đến 10 năm. Theo thời gian, những thành viên thuộc team hổ đã rời đi, và những thành viên hiện tại đang tiếp tục yêu cầu tái thiết kế lại hệ thống mới vì hệ thống mới đã lại là một mớ hỗn độn.

Chuyện giữ mã sạch không chỉ là chi phí mà nó còn là vấn đề sống còn của lập trình viên.

Attitude (Thái độ)

Bạn đã bao giờ lội qua một mớ hỗn độn trong vài tuần trong khi chỉ cần thực hiện nó trong vài giờ? Những gì bạn thấy là thay đổi một dòng thay vì thực hiện ở hàng trăm module khác nhau?

Vậy thì điều gì xảy đến với Code? Tại sao Good Code nhanh chóng bị mục nát trở thành Bad Code? Có rất nhiều lý do, nhưng yêu cầu thay đổi ngăn cản thiết kế ban đầu, lịch làm việc quá chặt chẽ, stupid manager và khách hàng không khoan nhượng,... Nhưng lỗi ở trong chính chúng ta.

Nếu là một lập trình viên không chuyên nghiệp, bạn sẽ đổ lỗi cho những yêu cầu thay đổi bất thình lìn của khách hàng, deadline của manager cứ thúc vào đít,..

Nói vậy thì chả khác nào ông bác sĩ không rửa tay làm bệnh nhân nhiễm khuẩn mà chết rồi ra trước tòa cãi lại rằng: thằng cha bệnh nhân yêu cầu tôi mổ luôn cho nhanh, không cần phải rửa tay, tui có ghi âm lại đàng hoàng nè.

Vậy nên nghĩa vụ của một lập trình viên chuyên nghiệp là bảo vệ Code của mình (mặc dù chậm Deadline nhưng vẫn phải sạch) trước ông manager. Vì ông manager cũng như tay bệnh nhân kia, chả biết quái gì về Code sạch (và cái giá phải trả cho hàng loạt Code bẩn).

The Primal Conundrum (Bài toán căn bản)

Muốn đúng hạn phải viết Code sạch. Bởi vì Code lộn xộn sẽ làm bạn chậm lại ngay lập tức. Cách duy nhất để đáp ứng tiến độ, cách duy nhất để đi nhanh đó là giữ cho mã nguồn của bạn luôn sạch nhất có thể.

The Art of Clean Code? (Nghệ thuật của mã sạch)

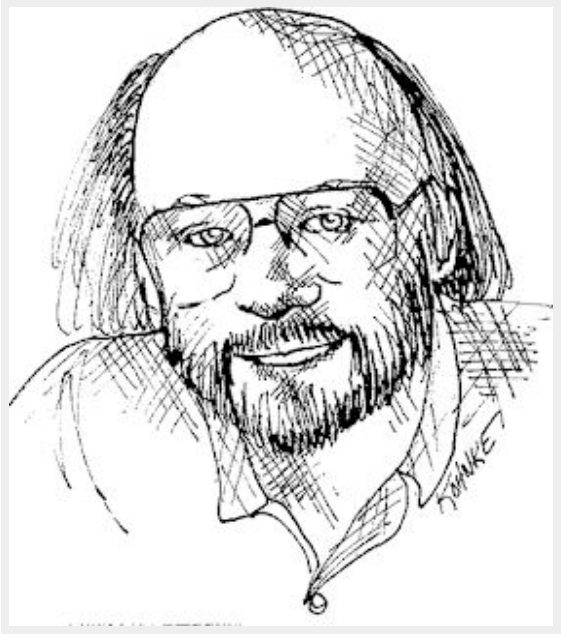
Làm thế nào để viết Code sạch?

Viết Code sạch là một nghệ thuật, đơn giản bạn có thể nhìn vào và nhận biết đâu là Code sạch, đâu là Code bẩn, nhưng không có nghĩa là chúng ta biết làm thế nào để viết Code sạch (Viết code sạch tương tự như chúng ta vẽ một bức tranh, phần lớn chúng ta đều có thể nhận ra được tranh được vẽ đẹp hay xấu, nhưng nhân thức được cái đẹp không có nghĩa là chúng ta biết làm thế nào để vẽ).

Để viết mã sạch bạn phải có code-sense của cleanliness (cảm giác mã về sự sạch sẽ), có thể luyện tập được thông qua quá trình áp dụng hàng nghìn kỹ thuật nhỏ vào việc viết code. Khi có được sẽ nhận biết được đâu là Code sạch đâu là Code bẩn, và có thể biến Code bẩn thành Code sạch, có được cảm giác và lựa chọn thay đổi tốt nhất.

Lập trình viên viết Code sạch là một nghệ sĩ có thể vẽ lên một màn hình trống rỗng thành một hệ thống mã thanh lịch.

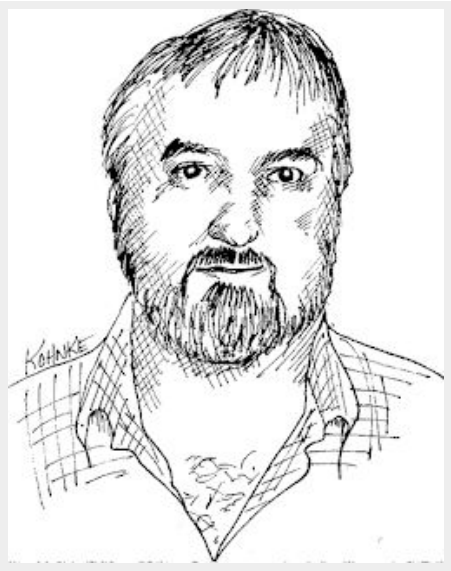
What Is Clean Code?



Bjarne Stroustrup (Cha đẻ C++): "Tôi thích code của tôi thanh lịch (đẹp, tinh tế và đơn giản) và hiệu quả. Logic nên đơn giản để làm cho nó khó gặp những lỗi ẩn, có tối thiểu những sự phụ thuộc để dễ dàng bảo trì, xử lý lỗi hoàn chỉnh theo những chiến lược thống nhất, và hiệu suất gần như tối ưu để không cám dỗ người ta làm mã lộn xộn với những trò tối ưu hóa lố lăng. Code sạch là code làm tốt một thứ."

Grady Booch (Object Oriented Analysis and Design with Applications): "Đơn giản và trực tiếp. Code sạch được đọc như đọc văn xuôi được viết trôi chảy. Không làm lu mờ đi mục đích thiết kế, nhưng vẫn đầy đủ các khái niệm trừu tượng hóa rõ ràng (nhanh nhạy quyết đoán và cần thiết) và những dòng mã điều khiển đơn giản dễ hiểu."

“Big” Dave Thomas (Sáng lập OTI, Bố già của Eclipse): "Có thể được đọc và cải tiến bởi một developer khác chứ không nhất thiết phải là tác giả bản đầu. Có Unit Test trên phương diện lập trình viên và Acceptance Test trên phương diện khách hàng. Có Meaningfull Names. Chỉ đưa ra một cách để làm một thứ chứ không phải nhiều cách để giải quyết vấn đề đó. Tối giản sự phụ thuộc, và luôn được định nghĩa một cách rõ ràng, tối thiểu hóa các API. Code nên dễ đọc, bởi vì tùy từng ngôn ngữ không phải tất cả các thông tin cần thiết đều được thể hiện rõ ràng trong một mình Code."



Michael Feathers (Working Effectively with Legacy Code): "Written by someone who cares, bạn có thể làm nó tốt hơn." => Mã sạch là mã được viết ra bởi một người có tâm.

Ron Jeffries (Extreme Programming Installed và Extreme Programming Adventures in C#):
Quy tắc của Beck về mã nguồn xếp theo mức độ ưu tiên:
- Chạy hết các test.
- Không có sự trùng lặp.
- Thể hiện các ý tưởng thiết kế vào trong hệ thống.
- Giảm số lượng các thực thể như class, method, function,...
Tính rõ nghĩa: Không chỉ là meaningful name, nó còn nằm ở do one thing.
=> Giảm thiểu sự trùng lặp, do one thing, rõ nghĩa và xây dựng những mô hình trừu tượng hóa nhỏ.

Ward Cunningham (Wiki, Fit, eXtreme Programming,): "Bạn có biết bạn vẫn đang làm trên code sạch khi mỗi đoạn code mà bạn đọc đúng như những gì bạn mong đợi. Bạn có thể gọi nó là mã đẹp khi mã đó làm cho ngôn ngữ trông như là ngôn ngữ được tạo ra để giải quyết chính vấn đề đó."

=> Bạn đọc mã mà không có bất cứ sự ngạc nhiên nào. Bạn không cần phải tốn nhiều công sức để đọc mã sạch. Bạn đọc nó và nó đúng với những gì bạn mong đợi. Nó rõ ràng, đơn giản và thuyết phục. Module này nối tiếp module khác, nó sẽ cho bạn biết những gì sẽ được viết tiếp theo.
=> Mã đẹp: Không phải ngôn ngữ được sinh ra để thiết kế giải quyết vấn đề của chúng ta. Vấn đề làm cho ngôn ngữ trông thật đơn giản để cứ như được sinh ra để giải quyết vấn đề này vậy.

Và trách nhiệm đó là của chúng ta.

Schools of Thoughts (Các trường phái)

Trong võ đạo thì mỗi người sáng lập là có một trường phái khác nhau, môn đệ mỗi môn phái thì đắm chìm trong lời dạy của người sáng lập, môn đệ sau khi tốt nghiệp thì có thể ra lập võ đường riêng.

Không ai trong số các trường phái đó đều hoàn toàn đúng đắn hết.

Vì vậy cho nên, cuốn sách này chỉ là kinh nghiệm mà tác giả đã thực hành được, cung cấp những lời dạy cần thiết và mang lại lợi ích nhất định, nó không có nghĩa là đúng tuyệt đối. Những cuốn sách khác của những lập trình viên chuyên nghiệp khác cũng cung cấp những lợi ích khác mà nó mang lại. Và đều có thể học hỏi từ họ những điều tốt.

Cuốn sách này do nhóm của tác giả đúc rút từ nhiều năm kinh nghiệm, lặp lại những lần thử nghiệm và các vấn đề gặp phải. Cho nên, tin hay không tùy bạn :v

We Are Authors (Chúng ta là những tác giả)

Chúng ta là những tác giả. Và mỗi tác giả đều có đọc giả của họ. Hãy nhớ, bạn là một tác giả, mỗi dòng code của bạn đều viết cho độc giả đọc, hãy để cho đọc giả đánh giá sự cố gắng của bạn.

Chúng ta liên tục phải đọc lại mã cũ. Việc này chiếm một phần nhất định trong quá trình viết code mới. Thế nên việc làm thế nào để cho mã trở nên dễ đọc cũng thực sự giúp cho việc viết code mới trở nên dễ dàng hơn.

Muốn làm cho nhanh, muốn làm cho xong, muốn viết Code dễ hơn, hãy làm cho nó dễ đọc!

The Boy Scout Rule (Luật hướng đạo sinh)

Code cần được giữ sạch theo thời gian. Muốn Code không bị sụt giảm hay xấu đi cần tuân theo luật. Một luật đơn giản của Hướng đạo sinh đó là:

"Rời khỏi khu cắm trại sạch hơn khi bạn đến!"

Làm sạch mã không phải một cái gì đó quá lớn lao, đơn giản chỉ là đổi tên một biến cho tốt hơn, tách một hàm lớn thành những hàm nhỏ hơn, loại bỏ những trùng lặp, dọn dẹp những câu lệnh if phức tạp,...

Prequel and Principles

SOLID:

- Single Responsibility Principle (SRP)
- Open Closed Principle (OCP)
- Dependency Inversion Principle (DIP)

Conclusion

Cuốn sách không chắc chắn giúp bạn trở thành một nghệ sĩ, một lập trình viên giỏi, nó chỉ có thể cho bạn thấy quá trình để trở thành một lập trình viên giỏi, các thủ thuật, kỹ thuật, và các công cụ những lập trình viên giỏi sử dụng.

Chapter 2: Meaningful Names

Name are everywhere!

Use Intention-Revealing Names (Sử dụng tên thể hiện mục đích)

Có thể nói đơn giản là thể hiện ý nghĩa qua tên gọi. Chọn một cái tên cần thời gian nhưng để nhớ nó còn tốn thời gian hơn nữa. Chú ý đến những cái tên bạn chọn và thay đổi chúng nếu bạn tìm thấy một cái tên thay thế tốt hơn.

Tên là câu trả lời cho một câu hỏi lớn. Nó sẽ nói cho bạn biết tại sao nó tồn tại, nó dùng để làm gì, và nó sử dụng ra sao.

Chọn tên để làm cho nó dễ dàng hơn trong việc hiểu và thay đổi mã nguồn.

Example:

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

1. What kinds of things are in the List?
2. What is the significance of the zeroth subscript of an item in the List?
3. What is the significance of the value 4?
4. How would I use the list being returned?

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

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

Avoid Disinformation (Tránh sai lệch thông tin)

Tránh đặt những tên gợi ý sai lầm làm lu mờ ý nghĩa thực sự của code. Ví dụ như viết tắt có thể hiểu nhầm sang nghĩa khác.

Ví dụ: Đừng quy một nhóm tài khoản thành accountList nếu nó không thực sự là một danh sách, vì có thể làm hiểu sai ý nghĩa. Nên thay bằng accountGroup, bunchOfAccount hay chỉ đơn giản là accounts.

Cẩn thận với tên có một sự thay đổi nhỏ. Sẽ mất bao lâu để nhận ra sự khác biệt giữa: XYZControllerForEfficientHandlingOfStringsvà XYZControllerForEfficientStorageOfStrings? Chúng có định dạng giống nhau đến khủng khiếp.

Sử dụng cách viết tương tự với khái niệm là Information, cách viết không nhất quán là Disinformation.

Ví dụ khủng khiếp khác là sử dụng tên ở mức thấp: l(L) hoặc O(o), dễ gây nhầm lẫn với số 1 và 0.

Make Meaningful Distinctions (Làm cho sự riêng biệt trở nên rõ ràng)

Lập trình viên tự tạo vấn đề cho chính họ khi viết ra code chỉ để thỏa mãn trình biên dịch hoặc một trình thông dịch chạy. Ví dụ: Bởi vì bạn không thể tạo cùng một tên để tham chiếu đến 2 điều khác nhau trong cùng một phạm vi, bạn có thể bị cám dỗ để thay đổi một tên cho phù hợp một cách tùy tiện. Đôi khi điều này được thực hiện bởi một lỗi chính tả, dẫn đến sửa lỗi chính tả và làm trình biên dịch không thể biên dịch. Nếu phải đặt một tên khác, nên lựa chọn tên dựa vào sự khác nhau giữa 2 điều đó.

Number-Series (a1, a2,... aN) nó mâu thuẫn với việc đặt tên có mục đích. Tên không cung cấp thông tin, không cung cấp ý định của người viết.

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

Hàm này sẽ dễ đọc hơn khi sử dụng biến source và destination như tham số truyền vào.

Noise work cũng là một sự riêng biệt vô nghĩa. Giả sử bạn có một class Product. Nếu bạn gọi hàm ProductInfo hoặc ProductData, bạn đã thực hiện thay đổi tên gọi khác nhau nhưng Data và Info là những noise work không rõ ràng như a, an và the. Không có gì sai khi sử dụng những noise work nếu họ đảm bảo làm cho sự riêng biệt có ý nghĩa nhưng cũng có vấn đề khi sử dụng nó.

Noise work là một sự dư thừa. Một biến không bao giờ có tên variable. Từ table không nên xuất hiện trong tên bảng. Tên NameString tốt hơn Name ở chỗ nào?

Cái tên khác biệt trong trường hợp này giúp bạn biết được sự khác biệt được đưa ra.

Use Pronounceable Names (Sử dụng tên đọc được)

Một phần quan trọng của bộ não chúng ta dành riêng cho khái niệm của từ ngữ. Những từ ngữ này sẽ được định nghĩa, và phát âm.

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

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

Rõ ràng, đoạn mã được phiên âm rõ ràng thì người sử dụng sẽ dễ dàng hiểu được vấn đề hơn.

Use Searchable Names (Sử dụng tên tìm kiếm được)

Tên biến và tên hằng có một vấn đề là không dễ dàng gì để xác định được vị trí của chúng. Nếu biến và hằng được sử dụng nhiều nơi trong đoạn mã thì nên đặt tên để tìm kiếm thân thiện hơn.

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

```
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;
}
```

Biến sum không phải một tên riêng việt có nghĩa nhưng nó dễ dàng tìm kiếm.

Avoid Encodings

Bảng mã hóa hiện tại đã đủ dùng vì vậy không cần bổ sung thêm vào. Việc thêm mã hóa lạ vào tên đơn giản làm tăng gánh nặng cho việc giải mã. Điều đó là không hợp lý khi yêu cầu nhân viên mới đọc hiểu những mã hóa lạ thêm vào trong khi những mã hóa có sẵn đã đủ dùng.

Hungarian Notation:
Ngày xửa ngày xưa, khi chúng ta làm việc với ngôn ngữ name-length-challenged, chúng ta đã vi phạm những quy tắc cần thiết. Fortran buộc phải mã hóa bằng cách thao tác với những chữ cái đầu tiên. Phiên bản đầu tiên của BASIC chỉ cho phép một ký tự với một chữ số. Hungarian Notation đã lên một tầm cao mới.

Trong ngôn ngữ lập trình hiện đại, các kiểu dữ liệu phong phú hơn, các trình biên dịch nhớ và thực thi được nhiều loại hơn. Hơn nữa xu hướng bây giờ các lớp nhỏ hơn, các hàm ngắn hơn nên mọi người thường có thể thấy điểm khai báo của mỗi biến mà họ đang sử dụng.

Member Prefixes (Thành phần tiền tố)
Bạn không cần phải đặt tên với tiền tố m_ ở mọi nơi. Các lớp và hàm đủ nhỏ để không phải sử dụng đến các tiền tố đó. Bạn nên dùng editing environment để đánh dấu hoặc bôi đậm các lớp hoặc các hàm để phân biệt chúng.

```
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 còn bỏ qua các tiền tố hoặc hậu tố để tên có ý nghĩa hơn.

Interfaces and Implementations
Đôi khi có trường hợp đặc biệt cần đến mã hóa. Ví dụ xây dựng một Abstract Factory, Factory được tạo ra dưới dạng một Interface và được thực thi bởi một class cụ thể. (Class : Interface) Có thể đặt tên như thế nào? IShapeFactory hay ShapeFactory? Thích lựa chọn AbcFactory hơn bởi vì khi đặt I ở trước có nhiều sự phân tán và thông tin không tốt. Không muốn khách hàng biết đang bàn giao cho họ một Interface, chỉ muốn họ biết rằng đó là một AbcFactory. Do đó nếu chọn mã hóa implementation hay interface chọn implementation. Gọi nó là ShapeFactoryImp hoặc CShapeFactory, nó thích hợp hơn là mã hóa interface

Avoid Mental Mapping (Tránh ánh xạ tinh thần)

Chắc chắn bộ đếm vòng lặp có thể được đặt tên là i, j, k (không bao giờ là l) nếu phạm vi của nó nhỏ và không có tên nào xung đột với nó. Bởi vì tên single-letter này là cho vòng lặp truyền thống. Tuy nhiên trong hầu hết những trường hợp khác nó là một sự lựa chọn nghèo nàn, nó khiến cho người đọc phải ánh xạ tinh thần tới một khái niệm thực tế.

"Lập trình viên nói chung là những người khá thông minh. Những người thông minh đôi khi thích thể hiện sự thông minh của mình bằng cách chứng minh khả năng tự sướng tinh thần của họ"

Sự khác nhau giữa lập trình viên thông minh và lập trình viên chuyên nghiệp đó là lập trình viên chuyên nghiệp hiểu được rằng "Sự rõ ràng chính là vua". Lập trình viên chuyên nghiệp dùng năng lực của họ để viết code làm cho người khác có thể hiểu được.

Class Names

Tên Classes và Objects nên là danh từ hoặc cụm danh từ. Tránh những từ như Manager, Processor, Data, or Info làm tên một class. Tên class cũng tránh là một động từ.

Method Names

Tên của Method nên là một động từ hoặc cụm động từ. Các phương thức truy xuất, thay đổi nên được đặt tên với giá trị của chúng và tiền tố như get, set, is.

```
string name = employee.getName();
```

customer.setName("mike");
if (paycheck.isPosted())...

Khi Constructors overload, sử dụng static factory method cùng với tên và đối số.

Complex fulcrumPoint = Complex.FromRealNumber(23.0);

tốt hơn là:

Complex fulcrumPoint = new Complex(23.0);

Don't Be Cute (Đừng tỏ ra dễ thương)

Nên lựa chọn những cái tên rõ ràng hơn là một cái tên tỏ ra hóm hỉnh hoặc cố tỏ ra thông minh và nguy hiểm (Ngu thì đừng tỏ ra nguy hiểm). Một lựa chọn tường minh thường tốt hơn là một lựa chọn mang tính chất giải trí.

Say what you mean (Hãy nói những gì bạn thấy có ý nghĩa). Mean what you say.

Pick One Word per Concept (Chọn một từ cho mỗi khái niệm)

Chọn một từ cho một khái niệm trừu tượng và gắn nó vào. Ví dụ, khá là khó để nhớ được các method tương đương ở trong các class khác nhau như fetch, retrieve, và get. Bạn có thể nhớ được những tên nào đã được sử dụng, nếu không sẽ mất khá nhiều thời gian cho việc tìm kiếm.

Các IDE hiện đại hỗ trợ chức năng tìm kiếm và thêm các tham số cần thiết khi khai báo function. Tên chức năng phải đứng một mình, và nó phải phù hợp để các bạn đúng phương pháp mà không thăm dò bổ sung.

Một thuật ngữ nhất quán đem lại lợi ích rất lớn cho những ai sử dụng mã nguồn của bạn.

Don't Pun (Không chơi chữ)

Không sử dụng một từ cho hai mục đích. Sử dụng một thuật ngữ cho hai ý tưởng khác nhau đơn giản đó chỉ là một trò chơi chữ.

Nếu bạn tuân thủ quy tắc "một từ một khái niệm", bạn có thể sử dụng nó cho nhiều class. Tuy nhiên cần cân nhắc khi vì lý do nhất quán những phương thức bạn sử dụng lại không có cùng nghĩa. Ví dụ, phương thức add không quan trọng giá trị trả về hay tham số truyền vào như nào, nó là tương đương. Tuy nhiên cần cân nhắc sử dụng phương thức add bạn định thêm vào không cùng ngữ nghĩa, có thể hiểu nhầm sang insert hoặc append. Giả sử chúng ta có nhiều classes với phương thức add, sẽ tạo một giá trị mới bằng cách thêm vào hoặc nối 2 giá trị có sẵn. Bây giờ chúng ta tạo một class mới với phương thức đặt một tham số vào một tập hợp. Lúc này không thể đặt tên phương thức là add nữa vì nó đã không còn phù hợp. trong trường hợp này chúng ta dùng insert hoặc append thì phù hợp hơn. Đặt tên phương thức là add lúc này một trò chơi chữ.

Use Solution Domain Names (Dùng tên của miền giải pháp)

Hãy nhớ rằng người đọc code của bạn cũng là những lập trình viên. Vì vậy nên dùng những từ ngữ khoa học máy tính, toán học, thuật toán, các tên kiểu mẫu,... Sẽ không là khôn ngoan nếu thay thế tên miền bằng một tên khác để những lập trình viên khác phải vào lại yêu cầu của khách hàng để xem chi tiết trong khi thực sự thì họ đã biết điều đó.

Use Problem Domain Names (Dùng tên của miền nghiệp vụ)

Khi làm việc với những thứ không phải của lập trình viên, hãy sử dụng tên nghiệp vụ. Ít nhất người bảo trì cũng nắm rõ được nghiệp vụ thông qua ý nghĩa của nó. Tách solution và problem là một phần công việc của lập trình viên và thiết kế tốt. Mã mà cần quan tâm tới những khái niệm nghiệp vụ nên mang tên nghiệp vụ.

Add Meaningful Context (Thêm bối cảnh có ý nghĩa)

Có rất ít tên mà tự thân nó không có nghĩa, hầu như là không. Với những tên này, phải đặt tên vào bối cảnh của người đọc và đặt nó vào tên class, function, namespace. Khi thất bại, có thể đặt thêm tiền tố cần thiết như một lựa chọn cuối cùng.

Hãy tưởng tượng bạn có các biến với tên như sau: firstName, lastName, street, houseNumber, city, state và zipcode. Đặt chúng cạnh nhau thì rất rõ là sẽ tạo thành một địa chỉ. Nhưng biến state có nghĩa gì nếu bạn thấy nó một mình trong một phương thức? Liệu bạn có hiểu ngay đó là một phần của một địa chỉ?

Bạn có thể thêm bối cảnh bằng cách sử dụng tiền tốt: addrFirstName, addrLastName, adddrState, v.v. Ít nhất người đọc sẽ hiểu rằng những biến này là phần của một cấu trúc lớn hơn. Dĩ nhiên, giải pháp tốt hơn là tạo một lớp có tên là Address. Lúc đó thì thậm chí trình biên dịch cũng biết là những biến này thuộc một câu trúc lớn hơn.

Don't Add Gratuitous Context (Đừng thêm những bối cảnh vô căn cứ)

Tên ngắn sẽ tốt hơn là tên dài nếu chúng rõ ràng. Việc không thêm bối cảnh vào tên là cần thiết.

Final Words

Việc khó nhất là đặt tên bởi nó yêu cầu kỹ năng mô tả tốt và một nền văn hóa chia sẻ. Đó là việc dạy hơn là vấn đề kỹ thuật, kinh doanh hay quản lý. Kết quả là, nhiều người trong lĩnh vực này không học để làm tốt.

Chapter 3: Functions

Trong những ngày đầu của lập trình, chúng ta biên soạn hệ thống của chúng ta bằng chương trình (routines - thủ tục, hàm, chương trình con) và những chương trình con (subroutines). Sau đó là thời đại của Fortran và PL/1 hệ thống của chúng ta bao gồm những chương trình (programs), chương trình con (subprograms), và chức năng (functions). Đến nay chỉ có function là tồn tại được từ những ngày đầu tiên. Function là đơn vị đầu tiên tổ chức nên bất cứ chương trình nào. Rất khó để hiểu một hàm chức năng dài với nhiều mức trừu tượng khác nhau, các lệnh if lồng nhau, các chuỗi xa lạ và những chức năng lẻ khác được gọi. Tuy nhiên, chỉ với một vài phương pháp rút gọn đơn giản, thay đổi một số tên, và tái cấu trúc lại một ít, là có thể nắm được những cơ bản chính ở trong function.

```
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();
}
```

Nếu như bạn không phải là một thành viên thực hiện, chắc chắn bạn không thể nào hiểu hết được tất cả những chi tiết bên trong nó. Tuy nhiên, bạn hoàn toàn có thể hiểu được chức năng của hàm này bao gồm cài đặt và xóa bỏ trang test sau đó trả lại về trang HTML.

Vì vậy hàm trên là dễ đọc, dễ hiểu hơn. Làm thế nào chúng ta có thể truyền đạt mục tiêu của một hàm chức năng? Những thuộc tính nào đưa vào function thì có thể làm cho một người đọc bất kỳ có thể cảm nhận được sự sống của nó bên trong chương trình?

Small!

Nguyên tắc đầu tiên của functions là chúng cần phải nhỏ. Nguyên tắc thứ hai là chúng cần phải nhỏ hơn nữa.

Đó không phải là một lời khẳng định mà tác giả có thể chứng minh. Không có bất cứ liên hệ với các nghiên cứu nào cho thấy function nhỏ thì sẽ tốt hơn. Những gì có thể nói cho bạn đó là trong gần bốn thập kỷ, những kinh nghiệm về viết các function với những kích thước khác nhau cho thấy, có một sự khó chịu kinh khủng với function 3000 dòng, function với 100-300 dòng. Và cuối cùng là những function với 20-30 dòng. Kinh nghiệm từ những sai lầm đã chỉ ra, function nên được viết rất nhỏ.

Function không nên dài quá 20 dòng và một dòng không nên quá 150 ký tự (không vượt quá màn hình 100-120 ký tự).

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

Blocks and Indenting
Những khối câu lệnh if, else, while nên được chứa trong một dòng, và những dòng này nên được đặt thành một lời gọi hàm. Điều này không chỉ giữ cho function nhỏ mà còn cho biết thêm thông tin cụ thể về việc nó đang làm bằng một cái tên mô tả độc đáo.

Điều này cũng có nghĩa là các function không nên có những cấu trúc lồng nhau. Vì thế, mức thụt lề của function nên chỉ ở mức một hoặc hai. Tất nhiên, nó sẽ làm cho function dễ đọc và dễ hiểu hơn.

Do One Thing

Function chỉ nên thực hiện một thứ.
Nó nên thực hiện điều đó cho tốt.
Nó nên thực hiện một điều duy nhất.

Vấn đề đặt ra ở đây là, thật khó để biết được "một thứ" (one thing) đó là gì?

Nên phân biệt rõ one thing với multi steps. Một function không chỉ là gồm một bước mà bao gồm nhiều công đoạn. Mỗi bước có thể lại là một function, mục tiêu là để phân rã một khái niệm lớn, nói cách khác tên của function là tập hợp các bước ở một mức độ tiếp theo của tính trừu tượng.

Thực hiện một điều bằng cách chia thành nhiều phần.

One Level of Abstraction per Function

Để đảm bảo chắc chắn rằng function của chúng ta đang thực hiện một điều duy nhất cần đảm bảo rằng các câu lệnh ở trong function đều ở cùng một mức độ trừu tượng.

Trộn lẫn các mức độ trừu tượng ở trong function luôn làm cho nó khó hiểu. Người đọc có thể không biết được biểu hiện cụ thể của một chi tiết hay khái niệm quan trọng. Tồi tệ hơn, như một cửa sổ bị phá vỡ, các chi tiết trộn lẫn với khái niệm quan trọng, và nó lớn dần lên bên trong hàm.

Reading Code from Top to Bottom: The Stepdown Rule
Chúng ta muốn đọc code như một câu chuyện từ trên xuống. Mỗi function được sắp xếp tuân theo theo mức độ của sự trừu tượng từ trên xuống. Giảm dần mức độ trừu tượng khi chúng ta đọc từ trên xuống. Nó gọi là nguyên tắc Stepdown.

```
To Include (Setups & TearDowns):
   Include (Setups)
   Include (Test Page Content)
   Include (TearDowns)
     To Include (Setups):
        Include (Suite Setup)
        Include (Regular Setup)
          To Include (Suite Setup):
             Search the parent hierarchy for the "SuiteSetUp"
             Add an Include statement with the path of that page
...
```

Rất khó khăn để các lập trình viên học và viết function ở một mức đơn cấp trừu tượng. Nhưng học thủ thuật này rất quan trọng. Nó giữ cho function ngắn và luôn thỏa mãn "one thing".

Switch Statements

Thật khó để rút gọn câu lệnh switch. Chúng ta không thể nào luôn luôn tránh câu lệnh switch, nhưng chúng ta chắc chắn rằng mỗi câu lệnh switch luôn cất giữ bởi một class ở mức thấp hơn và không bao giờ được sử dụng lại. Chúng ta làm được, bằng đa hình.

```
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);
  }
}
```

Có một số vấn đề với function này. Đầu tiên là nó lớn, và khi có thêm một loại nhân viên mới được thêm vào, nó sẽ phát triển thêm. Thứ hai, nó rõ ràng là làm nhiều hơn một điều. Thứ ba, vi phạm Single Responsibility Principle bởi vì có nhiều hơn một lý do cho trường hợp nó thay đổi. Thứ tư, vi phạm Open Close Principle bởi vì nó phải thay đổi bất cứ khi nào có kiểu nhân viên mới được thêm vào. Nhưng vấn đề tồi tệ nhất với function này là có một số lượng không giới hạn các function khác sẽ có cấu trúc tương tự.

Cách giải quyết vấn đề này là chôn vùi lệnh switch trong tầng hầm của một ABSTRACT FACTORY, và không bao giờ để bất kỳ ai nhìn thấy nó. Factory sẽ sử dụng lệnh switch để tạo ra các trường hợp thích hợp của nhân viên, và các chức năng khác nhau, tất cả được gửi đi bằng đa hình thông qua Employee interface.

```
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 cho câu lệnh switch là có thể được chấp nhận nếu nó chỉ xuất hiện một lần, được sử dụng để tạo ra các đối tượng đa hình, được ẩn đằng sau một mối quan hệ thừa kế để phần còn lại của hệ thống không thể nhìn thấy chúng. Tất nhiên trong mọi hoàn cảnh, có những lúc tôi vi phạm một hoặc nhiều phần của nguyên tắc đó.

Use Descriptive Names

Sử dụng tên mô tả những gì function làm.

Đừng ngại bởi vì nó là một cái tên dài. Tên mô tả dài tốt hơn là một tên ngắn bí ẩn. Tên mô tả dài tốt hơn là một đoạn bình luận mô tả. Sử dụng một quy tắc đặt tên là dùng những từ ngữ để đọc để làm tên hàm và dùng những từ khóa để mô tả chức năng của nó làm gì.

Sử dụng tên mô tả sẽ làm rõ việc thiết kế các module trong tâm trí bạn và giúp bạn cải thiện nó.

Function Arguments

Số lượng những tham số lý tưởng trong hàm là 0 (niladic), tiếp đến là 1 (monadic), theo sau là 2 (dyadic). Ba (triadic) tham số thì nên tránh nếu có thể. Nhiều hơn ba (polyadic) đòi hỏi phải có một sự lý giải rất đặc biệt và không nên sử dụng thế nào cũng được.

Tham số thậm chí còn khó khăn hơn khi đứng ở góc độ kiểm thử. Tưởng tượng rằng viết test case khó khăn thế nào khi phải đảm bảo tất cả tổ hợp các tham số đều hoạt động đúng. Nếu không có tham số, điều này là bình thường. Nếu có một tham số nó cũng không quá khó. Nếu hai tham số thì vấn đề trở nên khó khăn hơn một chút. Với nhiều hơn hai tham số, điều này quả thực là cả một vấn đề.

Tham số đầu ra khó để hiểu hơn là tham số đầu vào. Khi chúng ta đọc một hàm, chúng ta sử dụng ý tưởng về thông tin đầu vào qua các tham số, và giá trị trả về ở đầu ra. Không mong đợi thông tin trả về lại thông qua một tham số. Nên tham số đầu ra thường khiến chúng ta phải lấy hai lần (double-take).

Common Monadic Forms
Hai hình thức phổ biến khi sử dụng một tham số là:
Một hàm lý luận, trả về true hoặc false
Một hàm lập luận, biến nó thành một cái gì khác và trả lại nó.
Một hình thức ít phổ biến hơn là hàm xử lý sự kiện, có một tham số đầu vào nhưng không có đối số đầu ra.

Cố gắng tránh bất cứ Monadic Functions nào mà không theo những form trên.

Flag Arguments
Không gì tồi tệ hơn khi tham số truyền vào là một flag argument, bởi vì nó đã làm nhiều hơn một điều: một khi flag có giá trị đúng, và khi flag có giá trị sai.
Chúng ta nên tách trường hợp này thành 2 hàm.

Dyadic Functions
Hàm có hai tham số sẽ khó để hiểu hơn hàm có một tham số.
Có những trường hợp 2 tham số là hoàn toàn hợp lý, Ví dụ: Point p = new Point(0,0);
Tuy nhiên 2 tham số trong trường hợp này được sắp xếp làm thành phần của một giá trị duy nhất.
=> Hai tham số phải có sự gắn kết tự nhiên hoặc là sắp xếp của một trật tự tự nhiên.

Triads
Những hàm phải đưa ra tới 3 đối số càng khó để hiểu hơn là hai. Khuyên bạn nên thực sự cẩn thận khi đưa ra một hàm với một bộ 3 tham số.
(Overload khi đợi các tham số)
Tuy nhiên đôi khi cũng cần phải sử dụng. Ví dụ: assertEquals (1.0, amount, .001)

Argument Objects
Khi một số hàm thực sự cần nhiều hơn 2 hay 3 tham số, có khả năng là một số trong những tham số đó có thể gói gọn trong một lớp của riêng chúng:

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

Argument Lists
Đôi khi chúng ta muốn vượt qua số lượng biến số của tham số vào một hàm.

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

Nếu các tham số đều được đối xử tương tự nhau, có thể gộp chung lại thành một tham số đơn kiểu List.

Verbs and Keywords

Trong trường hợp là một monad function, hàm và tham số nên tạo thành một động từ hoặc một cặp danh từ đẹp.

Điều cuối cùng là dạng từ khóa của tên hàm. Sử dụng hình thức này chúng ta mã hóa tên các tham số vào tên hàm. Ví dụ: assertEquals có thể được viết tốt hơn là assertExpectedEqualsActual(expected, actual), điều này giảm nhẹ vấn đề phải nhớ thứ tự các tham số.

Have No Side Effects (Không có tác dụng phụ)

Hàm của bạn hứa hẹn làm một điều, nhưng đôi khi có những phản ứng không mong muốn. Đôi khi nó làm thay đổi bất thường các biến trong class của chính nó. Đôi khi nó để các thông số thông qua và vào hàm hoặc hệ thống toàn cục. Trong cả 2 trường hợp nó sinh ra nhưng hiểu lầm quanh co và tai hại dẫn đến những sự phụ thuộc và liên kết kỳ lạ.

=> Do One Thing

Output Arguments
Ví dụ:

appendFooter (s);

Có phải hàm này nối thêm s và Footer vào cái gì không? Hay là nối thêm footer vào s? s là input hay output?

public void appendFooter(StringBuffer report)

Lúc trước, khi lập trình hướng đối tượng đôi khi cần thiết có tham số đầu ra. Tuy nhiên, phần lớn các tham số đầu ra biến mất dần, bởi vì this như một dụng ý hành động như một tham số đầu ra. Nói cách khác nó sẽ là tốt hơn khi gọi appendFooter là:

report.appendFooter();

Nói chung, nên tránh tham số đầu ra.

Command Query Separation (Tách lệnh truy vấn)

Hàm nên thực hiện hoặc làm một điều gì đó hoặc trả lời một cái gì đó, nhưng không bao gồm cả hai. Hàm của bạn nên thay đổi trạng thái của đối tượng hoặc trả về thông tin của một đối tượng. Làm cả hai sẽ dẫn đến sự nhầm lẫn.

Ví dụ:

public boolean set(String attribute, String value);

Hàm này đặt giá trị của một thuộc tính và trả về true nếu thành công, và false nếu thuộc tính không tồn tại.

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

Vấn đề: Không biết set này dùng theo cách
- Nếu "username" đã tồn tại giá trị "unclebob" trước chưa
- Set thuộc tính "username" với giá trị "unclebob"...
=> Gây nhầm lẫn

Giải quyết vấn đề bằng cách đặt lại tên và phân tách các câu lệnh truy vấn để sự mơ hồ không xảy ra:

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

Prefer Exceptions to Returning Error Codes

Trả về các đoạn code báo lỗi ở trong các câu lệnh của hàm là một vi phạm tinh tế của tách câu lệnh truy vấn. Nó đẩy mạnh các lệnh sử dụng như biểu thức trong các vị từ của câu lệnh if.

```
if (deletePage(page) == E_OK) {
  if (registry.deleteReference(page.name) == E_OK) {
    if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
      logger.log("page deleted");
    } else {
      logger.log("configKey not deleted");
    }
  } else {
    logger.log("deleteReference from registry failed");
  }
} else {
  logger.log("delete failed");
  return E_ERROR;

}
```

Điều này không bị nhầm lẫn giữa các động từ/tính từ nhưng cũng dẫn đến các cấu trúc lồng nhau. Khi tạo ra các mã lỗi, bạn tạo ra vấn đề mà người gọi phải đối phó ngay lập tức.

Mặt khác, nếu bạn sử dụng sự ngoại lệ để thay thế một đoạn code lỗi, sau đó lỗi xử lý code được tách ra, không ảnh hưởng tới những vấn đề khác.

```
try {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
  logger.log(e.getMessage());
}
```

Extract Try/Catch Blocks
Tách các khối xử lý Try/Catch ra khỏi các thành phần chức năng riêng của chính nó.

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
}
private void deletePageAndAllReferences(Page page) throws Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}
private void logError(Exception e) {
  logger.log(e.getMessage());
}
```

Error Handling Is One Thing
Function nên làm một việc (do one thing). Xử lý lỗi là một việc (one thing). Vậy nên một chức năng xử lý lỗi không nên làm bất cứ thứ gì nữa.

The Error.java Dependency Magnet

Một class hoặc Enum chứa tất cả định nghĩa các lỗi. Class đó như một nam châm phụ thuộc; nhiều class khác phải nhập vào và sử dụng nó. Vậy nên khi danh sách lỗi thay đổi thì tất cả những class khác đều phải được biên dịch và bố trí lại. Lập trình viên không muốn điều đó xảy ra nên sử dụng lại các Code lỗi cũ thay vì thêm mới.

Khi sử dụng các exception hơn là các error code, việc thêm một ngoại lệ mới là dẫn xuất của lớp exception nên chúng có thể được thêm vào mà không cần phải biên dịch hay bố trí lại.

Don't Repeat Yourself

Sự trùng lặp là một vấn đề lớn vì nó làm tăng khối lượng mã nguồn lên và sẽ phải yêu cầu thay đổi nhiều lần nếu thuật toán có sự thay đổi. Nó cũng là cấp số nhân nhiều lần cho một lỗi thiếu sót nào đó.

Mục tiêu là giảm thiểu sự trùng lặp.

Trùng lặp có thể là gốc rễ mọi tội lỗi ở trong phần mềm. Nhiều nguyên tắc và thông lệ được tạo ra nhằm kiểm soát và hạn chế sự trùng lặp đó. Lập trình hướng đối tượng, lập trình cấu trúc,...

Structured Programming

Nguyên tắc Dijkstra trong lập trình cấu trúc: Mọi Function và mọi block ở trong function nên có một đầu vào và một đầu ra. Điều đó có nghĩa là chỉ nên có một statement trả về trong hàm, không có break hay continue trong một vòng lặp, và không bao giờ có chạy mãi mãi hay bất cứ lệnh goto nào.

Nó mang lại lợi ích rất ít khi hàm chức năng rất nhỏ. Chỉ khi hàm chức năng lớn hơn theo nguyên tắc này mới mang lại lợi ích đáng kể.

Vì vậy bạn nên giữ cho hàm của bạn nhỏ, sau đó thêm các phát biểu return, break, hay continue không gây tổn hại gì, đôi khi còn mang lại ý nghĩa hơn là tuân theo nguyên tắc một đầu vào, một đầu ra. Mặt khác, goto chỉ nên sử dụng khi các hàm lớn, nên không khuyến khích sử dụng.

How Do You Write Functions Like This?

Viết phần mềm cũng như viết bất cứ một văn bản nào khác. Khi bạn viết một bài văn hay một bài báo, bạn sẽ có những ý tưởng đầu tiên, sau đó bạn uốn nắn nó để có thể đọc được tốt hơn. Phác thảo đầu tiên có thể vụng về và thiếu tổ chức, nhưng bạn có thể rèn câu cú, tái cấu trúc và sàng lọc lại cho đến khi nó đọc được theo cách bạn muốn.

Khi viết một hàm, nó bắt đầu rất dài và phức tạp. Nó có nhiều vết lõm và vòng lặp lồng nhau. Nó có một danh sách dài các tham số. Tên thì tùy tiện và có code trùng lặp. Nhưng cũng có một bộ Unit Test bao gộp những dòng code vụng về đó.

Vì vậy, uốn nắn và sàng lọc lại code, phân chia các chức năng, thay đổi tên, loại bỏ sự trùng lặp. Co rút các phương pháp và sắp xếp nó lại. Đôi khi bùng nổ các class nhưng sau tất cả vẫn giữ cho test passing.

Conclusion

Functions là động từ, Classes là danh từ. Nghệ thuật của lập trình là nghệ thuật của ngôn ngữ thiết kế.

Các lập trình viên bậc thầy nghĩ rằng các hệ thống như một câu chuyện để kể chứ không phải là một chương trình để viết.

Setup TeardownIncluder

```
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;

  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();
}

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);
        }
}

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");
    }
}
```

Chapter 4: Comments

"Đừng bình luận trên Code xấu - Hãy viết lại nó."
-Brian W.Kernighan và P.J.Plaugher-

Comment có thể giúp đỡ rất tốt nếu đặt đúng vị trí, nhưng nó cũng có thể trở nên tồi tệ, gây nhiễu loạn không tin khi đặt sai hay cung cấp thông tin không chính xác.

Sử dụng comment như một sự bù đắp cho thất bại trong việc thể hiện code. Comment luôn luôn là thất bại. Chúng ta phải sử dụng nó vì chúng ta không thể luôn luôn tìm ra cách để thể hiện code ra, nhưng sử dụng nó không phải là một nguyên nhân đáng để ăn mừng.

Điều tin tưởng tuyệt đối nhất là: Code. Chỉ có Code mới nói cho chúng ta biết chính xác nhất nó đang làm gì. Mặc dù đôi khi comments là cần thiết, nhưng chúng ta sẽ tiêu hao đáng kể năng lượng cho việc giảm thiểu chúng.

Comments Do Not Make Up for Bad Code (Comment không dùng để trang trí cho mã xấu)

Một trong những động lực phổ biến thúc đẩy việc viết comment là mã xấu. Chúng ta viết một module và chúng ta biết rằng nó lộn xộn và vô tổ chức. Chúng ta biết nó bừa bộn. Nên chúng ta nói với chính mình "Ohh, tôi nên giải thích nó!" Không! Bạn nên dọn dẹp nó!

Dọn dẹp và diễn đạt code cùng với một vài comments tốt hơn nhiều là code lộn xộn cùng với một đống comments. Thay vì dành thời gian cho việc viết comments cho một đống lộn xộn, bạn nên dọn dẹp nó.

Explain Yourself in Code

Code chắc chắn là lời giải thích tốt nhất. Trong nhiều trường hợp, thay thế code rắc rối bằng cách tạo ra những hàm mới đã nói lên đủ những lời comment mà bạn muốn viết dành riêng cho đoạn code rắc rối đó.

Good Comments

Một số comment là cần thiết hoặc có lợi ích.

Legal Comments (Bình luận pháp lý)
Điều này tùy thuộc vào tiêu chuẩn viết code của công ty. Ví dụ: Bản quyền và quyền tác giả là điều cần thiết và hợp lý để đưa vào comment lúc bắt đầu của mỗi tập tin mã nguồn.

Nếu có thể, hãy tham khảo tới một giấy phép tiêu chuẩn hay một tài liệu ở bên ngoài hơn là đặt tất cả các điều khoản và điều kiện pháp lý vào trong comment.

Informative Comments (Bình luận thông tin)
Đôi khi khá là hữu dụng khi cung cấp những thông tin cơ bản với một bình luận.

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

Một bình luận như vậy đôi khi khá là hữu ích, tuy nhiên tốt hơn hết là nên truyền tải thông tin thông qua tên của function nếu có thể. (responderBeingTested.)

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

Trong trường hợp này bình luận nhằm cho chúng ta biết định dạng về thời gian trong hàm SimpleDateFormat.format với một chuỗi xác định. Tuy nhiên, nó vẫn có thể được làm tốt hơn, rõ ràng hơn nếu mã này được chuyển sang một class đặc biệt với mục đích chuyển đổi định dạng thời gian. Sau đó, những comment có thể không cần thiết nữa.

Explanation of Intent (Giải thích về mục đích)
Đôi khi một lời bình luận còn vượt xa hơn cả mong đợi thông tin hữu ích về việc thực hiện và cung cấp mục đích đằng sau một quyết định.

Clarification (Làm sáng tỏ)
Dịch ý nghĩa của các lập luận khó hiểu, hoặc trả lại một giá trị rõ ràng ở bên phải chính nó. Nhưng chỉ khi một phần thư viện chuẩn hoặc trong code mà bạn không thể thay đổi được, một bình luận Clarifying có thể hữu ích.

```
{
  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
}
```

Thực tế nó là một rủi ro, clarifying comment là không chính xác. Clarification là cần thiết và nó cũng đầy rủi ro. Vì vậy, trước khi viết bình luận như vậy hãy quan tâm xem có cách nào tốt hơn không, và sau đó dành nhiều quan tâm hơn để lựa chọn đúng đắn nhất.

Warning of Consequences (Cảnh báo về các hậu quả)
Cảnh báo những lập trình viên khác về những hậu quả nhất định. Ví dụ, giải thích lý do tại sao một test case cụ thể bị tắt đi:

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

Có thể bạn phàn nàn rằng có những cách tốt hơn để giải quyết vấn đề này. Về điều này hoàn toàn đồng ý với bạn. Nhưng bình luận được đưa ra ở đây là hoàn toàn hợp lý. Nó ngăn chặn một số lập trình viên khác

TODO Comments
TODO là những công việc mà lập trình viên nghĩ là nên làm, nhưng vì một số lý do mà chưa thể làm ở hiện tại. Nó có thể là một lời nhắc nhở về việc xóa một tính năng bị phản đối, một lời bào chữa cho ai đó khi nhìn vào vấn đề. Nó có thể là một yêu cầu dành cho ai đó để nghĩ ra một cái tên hay hơn, hay nhắc nhở về một sự thay đổi nào đó cho sự kiện đã được lên kế hoạch từ trước. Bất kể sử dụng TODO như thế nào nó cũng không phải là một cái cớ để để lại mã xấu ở lại trong hệ thống.

Bạn cũng không muốn code của mình rải rác với TODO ở một vài nơi. Vì vậy nên quét qua chúng thường xuyên và giải quyết những gì bạn có thể.

Amplification (Phóng đại)
Comment có thể được dùng để khuếch đại một điều gì đó quan trọng mà nó có vẻ như là không quan trọng.

Javadocs in Public APIs
Tương tự Microsoft .AspNet.WebApi.HelpPage

Bad Comments

Tất cả những gì không phải là Good Comments

- Mumbling
- Redundant Comments
- Misleading Comments
- Mandated Comments
- Journal Comments

- Noise Comments
- Scary Noise
- Don't use a Comment When You Can Use a Function or a Variable
- Position Markers
- Closing Brace Comments
- Attributions And Bylines
- Commented-Out Code
- HTML Comments
- Nonlocal Information
- Too Much Information
- Inobvious Connection
- Function Headers
- Javadocs in Nonpublic Code
- Example

Chapter 5: Formatting

Khi mọi người nhìn vào, chúng tôi muốn code được nhìn vào gọn gàng, nhất quán, chi tiết. Muốn mọi người cảm nhận được đây là do một chuyên gia viết ra. Nếu thay vào đó họ thấy được một khối lượng code loằng ngoằng trông giống như được viết bởi một thủy thủ say rượu, họ có thể kết luận rằng sự thiếu chuyên nghiệp tràn ngập trong mọi khía cạnh khác của dự án.

Bạn nên quan tâm để code của bạn được định dạng một cách đẹp mắt.

The Purpose of Formatting

Điều đầu tiên, hãy dọn dẹp. Hình thức của code rất quan trọng. Định dạng code là giao tiếp, và giao tiếp là điều đầu tiên mà các doanh nghiệp xác định lập trình viên chuyên nghiệp.

Vertical Formatting (Theo chiều dọc)

The Newspaper Metaphor

Hãy tư duy như viết một bài báo tốt. Bạn đọc nó từ trên xuống. Ở phía trên, tiêu đề cho bạn biết bài viết đang nói về cái gì. Đoạn đầu tiên tóm tắt toàn bộ câu chuyện, ẩn các chi tiết và mang những khái niệm trải rộng ra. Ngay khi đi xuống các chi tiết sẽ được tăng lên.

Chúng ta muốn mã nguồn giống như một bài báo. Tên đơn giản nhưng giải thích rõ ràng. Cái tên cho chúng ta biết có đang ở đúng module hay không. Trên đầu cung cấp những khái niệm ở cấp cao và các thuật toán. Chi tiết sẽ tăng thêm khi chúng ta di chuyển xuống dưới. Cho đến khi kết thúc, chúng ta tìm những hàm ở mức thấp nhất và chi tiết của file nguồn.

Nếu một bài báo là một câu chuyện dài với sự hội tụ của một kết cấu vô tổ chức các sự kiện với ngày tháng, đơn giản chúng tôi sẽ không đọc nó.

Vertical Openness Between Concepts (Sự cởi mở giữa các khái niệm)
Gần như tất cả code đều được đọc từ trái sang phải, từ trên xuống dưới. Mỗi dòng đều đại diện cho một biểu thức hoặc một mệnh đề. Những khái niệm nên được tách ra với một dòng trắng. Mục đích nhằm tăng khả năng đọc hiểu code

Vertical Density (Mật độ)
Cởi mở chia tách các khái niệm, sau đó mật độ liên kết chặt chẽ lại.

=> Comment làm phá hỏng sự liên kết:

```
public class ReporterConfig {
  /**
  * The class name of the reporter listener
  */
  private String m_className;
  /**
  * The properties of the reporter listener
  */
  private List<Property> m_properties = new ArrayList<Property>();
  public void addProperty(Property property) {
  m_properties.add(property);
}
```

=> Dễ dàng hơn để đọc:

```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();
  public void addProperty(Property property) {
  m_properties.add(property);
}
```

Vertical Distance (Khoảng cách)
Các khái niệm có liên quan với nhau nên được giữ gần nhau, tránh tách chúng ra thành từng file khác nhau trừ khi có lý do chính đáng.

- Khai báo biến: Nên khai báo biến ở càng gần nơi mà nó sử dụng càng tốt. Bởi vì hàm ngắn, nên biến cục bộ nên đặt đầu mỗi hàm. Các biến điều khiển cho vòng lặp nên được khai báo ở trong vòng lặp.
- Biến đối tượng: Khai báo biến ở trên cùng của một lớp.
- Các hàm phụ thuộc: Nếu một hàm gọi một hàm khác thì nên để chúng gần nhau.

Vertical Ordering
Hàm chức năng được gọi thì nên ở phía dưới hàm gọi nó. Tạo một dòng chảy mã nguồn từ mức cao đến mức thấp. Cũng đừng thu nhỏ font chữ để tăng số lượng ký tự trên 1 dòng màn hình.

Horizontal Formatting (Theo chiều ngang)

Cố gắng giữ cho dòng được ngắn. Đừng bao giờ phải di chuyển sang phải.

Horizontal Openness and Density
Sử dụng khoảng trắng để kết hợp những thứ có liên quan và tách những thứ ít liên quan hơn với nhau.

```
public class Quadratic {
  public static double root1(double a, double b, double c) {
    double determinant = determinant(a, b, c);
    return (-b + Math.sqrt(determinant)) / (2*a);
  }

  public static double root2(int a, int b, int c) {
    double determinant = determinant(a, b, c);
    return (-b - Math.sqrt(determinant)) / (2*a);
  }

  private static double determinant(double a, double b, double c) {
    return b*b - 4*a*c;
  }
}
```

Horizontal Alignment (Dóng hàng)

Dóng hàng như vậy không thực sự hữu ích. Cách này như làm nổi lên những vấn đề sai trái và làm đi lệch ra khỏi mục đích thật sự. Không cần phải dóng thẳng hàng theo từng cột.

Indentation
Thụt lề phân cấp các phạm vi.

Dummy Scopes
while (dis.read(buf, 0, readBufferSize) != -1);

Team Rules

Mỗi lập trình viên đều có một nguyên tắc định dạng riêng, nhưng khi làm việc nhóm phải tuân theo nguyên tắc của nhóm. Một nhóm phát triển nên thỏa thuận về một kiểu định dạng duy nhất, sau đó mỗi thành viên trong nhóm nên sử dụng kiểu định dạng đó. Điều này tạo ra tính nhất quán.

Uncle Bob's Formatting Rules

Chapter 7: Error Handling

Nghe có vẻ kỳ lạ khi nói về xử lý lỗi trong cuốn sách về clean code. Chỉ là xử lý lỗi là một trong những điều mà tất cả chúng ta phải làm khi chúng ta lập trình. Xử lý lỗi là quan trọng nhưng nếu nó che lấp đi logic thì nó là sai lầm.

Use Exceptions Rather Than Return Codes (Sử dụng ngoại lệ hơn là trả về Codes)

Bạn có thể thiết lập một error flag hoặc trả về một mã lỗi mà người gọi có thể kiểm tra.

```
public class DeviceController {
  ...
  public void sendShutDown() {
    DeviceHandle handle = getHandle(DEV1);
          // Check the state of the device
    if (handle != DeviceHandle.INVALID) {
      // Save the device status to the record field
      retrieveDeviceRecord(handle);
      // If not suspended, shut down
      if (record.getStatus() != DEVICE_SUSPENDED) {
        pauseDevice(handle);
        clearDeviceWorkQueue(handle);
        closeDevice(handle);
      } else {
        logger.log("Device suspended. Unable to shut down");
      }
    } else {
      logger.log("Invalid handle for: " + DEV1.toString());
    }
  }
  ...
}
```

Vấn đề ở đây là nó lộn xộn với người gọi. Người gọi phải kiểm tra trực tiếp sau khi gọi. Không may, nó rất dễ bị quên lãng. Vì lý do đó, tốt hơn là cho nó ra một exception khi bạn gặp phải lỗi. Nó làm mã được sạch hơn và logic không bị che khuất bởi xử lý lỗi.

```
public class DeviceController {
  ...
```

```
  public void sendShutDown() {
    try {
      tryToShutDown();
    } catch (DeviceShutDownError e) {
      logger.log(e);
    }
  }

  private void tryToShutDown() throws DeviceShutDownError {
    DeviceHandle handle = getHandle(DEV1);
    DeviceRecord record = retrieveDeviceRecord(handle);

    pauseDevice(handle);
    clearDeviceWorkQueue(handle);
    closeDevice(handle);
  }

  private DeviceHandle getHandle(DeviceID id) {
    ...
    throw new DeviceShutDownError("Invalid handle for: " + id.toString());
    ...
  }
  ...
}
```

Code tốt hơn bởi vì có hai quan hệ lộn xộn ở đây, thuật toán cho thiết bị tắt máy và xử lý lỗi, và bây giờ nó đã được tách ra.

Write Your Try-Catch-Finally Statement First

Try - Như transactions.
Catch - Rời khỏi chương trình và không vấn đề xảy ra trong try.

Sử dụng try-catch-finally là một cách tốt để bắt đầu viết code khi bạn đang viết code và ném đi những ngoại lệ. Điều này giúp chúng ta định nghĩa được những gì mà người dùng nên mong đợi, không vấn đề gì xảy ra đối với các đoạn code ở trong try.

Ví dụ: Viết một đoạn code với mục đích truy cập file và đọc một số đối tượng theo sắp xếp theo thứ tự.

Chúng ta bắt đầu với Unit Test cho chúng ta biết ngoại lệ khi file không tồn tại:

```
@Test(expected = StorageException.class)
```

```
public void retrieveSectionShouldThrowOnInvalidFileName() {
  sectionStore.retrieveSection("invalid - file");
}
```

Test này thúc đẩy chúng ta viết bản sơ khai:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  // dummy return until we have a real implementation
  return new ArrayList<RecordedGrip>();
}
```

Test này thất bại nó chưa ném ra ngoại lệ. Tiếp theo chúng ta thay đổi hiện thực, do đó mà nó cố gắng truy cập vào một tập tin không hợp lệ. Thao tác này throw một exception:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  try {
    FileInputStream stream = new FileInputStream(sectionName)
  } catch (Exception e) {
    throw new StorageException("retrieval error", e);
  }
  return new ArrayList<RecordedGrip>();
}
```

Test passes bởi vì nó đã bắt lấy một exception. Tái cấu trúc lại:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  try {
    FileInputStream stream = new FileInputStream(sectionName);
    stream.close();
  } catch (FileNotFoundException e) {
    throw new StorageException("retrieval error”, e);
  }
  return new ArrayList<RecordedGrip>();
}
```

Use Unchecked Exceptions

Checked: are the exceptions that are checked at compile time.
Unchecked are the exceptions that are not checked at compiled time. C# doesn’t have checked exceptions.

Checked exceptions can sometimes be useful if you are writing a critical library: You must catch them. But in general application development the dependency costs outweigh the benefits.

Provide Context with Exceptions (Cung cấp bối cảnh cho ngoại lệ)

Mỗi trường hợp ngoại lệ bạn nên cung cấp ngữ cảnh để xác định nguồn gốc và vị trí của lỗi. Trong Java, bạn sẽ nhận được một stack truy tìm (trace) từ bất kỳ ngoại lệ nào. Tuy nhiên stack đó không cho bạn biết ý nghĩ của các ngoại lệ không thành công.

Tạo một thông điệp báo lỗi và truyền cho chúng với ngoại lệ của bạn. Đề cập đến các hành động gây ra lỗi và loại lỗi.

Define Exception Classes in Terms of a Caller's Needs the Normal Flow

Có nhiều cách để phân loại lỗi. Chúng ta có thể phân loại theo nguồn gốc: Chúng đến từ thành phần hay những gì khác? Loại của chúng: Lỗi thiết bị, lỗi mạng, hay lỗi lập trình? Tuy nhiên khi chúng ta định nghĩa các lớp ngoại lệ ở trong ứng dụng, mối quan tâm lớn nhất của chúng ta là làm cách nào chúng bắt được.

```
ACMEPort port = new ACMEPort(12);

try {
   port.open();
} catch (DeviceResponseException e) {
   reportPortError(e);
   logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
   reportPortError(e);
   logger.log("Unlock exception", e);
} catch (GMXError e) {
   reportPortError(e);
   logger.log("Device response exception");
} finally {
   …
}
```

Bạn đừng ngạc nhiên có rất nhiều sự trùng lặp ở đây. Chúng ta ghi lại mỗi lỗi ở đây và đảm bảo rằng có thể tiến hành. Trong trường hợp này, chúng ta biết công việc chúng ta đang làm gần như là giống nhau bất kể ngoại lệ nào, chúng ta có thể đơn giản code bằng wrapping API mà chúng ta gọi và đảm bảo rằng nó sẽ trả về một ngoại lệ chung.

```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportError(e);
  logger.log(e.getMessage(), e);
} finally {
  …
}
```

Class LocalPort chỉ là một wrapper đơn giản catch và translates ngoại lệ thown bởi class ACMEPort:

```
public class LocalPort {
  private ACMEPort innerPort;

  public LocalPort(int portNumber) {
    innerPort = new ACMEPort(portNumber);
  }

  public void open() {
    try {
      innerPort.open();
    } catch (DeviceResponseException e) {
      throw new PortDeviceFailure(e);
    } catch (ATM1212UnlockedException e) {
      throw new PortDeviceFailure(e);
    } catch (GMXError e) {
      throw new PortDeviceFailure(e);
    }
  }
  …
}
```

Wrapper giống như những gì chúng ta định nghĩa cho ACMEPort có thể rất dễ sử dụng. Thực tế, wrapping từ một bên API thứ ba là một thực hành tốt nhất. Khi bạn làm điều đó, bạn sẽ giảm thiểu sự phụ thuộc, hoặc di chuyển nó đến một thư viện khác trong tương lai mà ko có nhiều vấn đề. Nó cũng rất dễ dàng để gọi mock (thử nghiệm) khi kiểm thử Code.

Thông thường một class ngoại lệ duy nhất là tốt cho một khu vực cụ thể của code. Giúp các thông tin được gửi với mã ngoại lệ có thể phân biệt được các lỗi. Sử dụng các class khác nhau chỉ khi có lúc bạn muốn bắt một ngoại lệ và cho phép những cái khác đi qua.

Define the Normal Flow

Những điều ở trên giúp bạn tách biệt business logic và xử lý lỗi. Phần lớn code của bạn sẽ trông giống như không được sạch sẽ. Tuy nhiên quá trình thực hiện điều này thúc đẩy việc phát hiện lỗi trên các khía cạnh của chương trình bạn. Bạn wrap bên ngoài API nên bạn có thể throw ngoại lệ của chính bạn, định nghĩa một hàm xử lý trên code, vậy nên bạn có thể đối phó với bất kỳ tính toán bị phá bỏ nào. Hầu hết thời gian đó là một cách tiếp cận tuyệt vời, nhưng thỉnh thoảng bạn có thể không muốn phá bỏ.

Dưới đây là một mã vụng về tính tổng chi phí trong một ứng dụng thanh toán.

```
try {
  MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
  m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
  m_total += getMealPerDiem();
}
```

Logic ở đây có tình trạng lộn xộn. Nếu chúng ta không có ngoại lệ code của chúng ta sẽ đơn giản hơn nhiều.

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

```
public class PerDiemMealExpenses implements MealExpenses {
  public int getTotal() {
    // return the per diem default
  }
}
```

Đây là một trường hợp mẫu đặc biệt. Bạn tạo một lớp hoặc cấu hình một đối tượng để xử lý trường hợp đặc biệt cho bạn. Khi bạn làm, Client code sẽ không phải đối phó với những hành vi đặc biệt. Hành vi đó được đóng gói trong một đối tượng với trường hợp đặc biệt.

Don't Return Null

Kiểm tra null

```
public void registerItem(Item item) {
  if (item != null) {
    ItemRegistry registry = peristentStore.getItemRegistry();
    if (registry != null) {
```

```
        Item existing = registry.getItem(item.getID());
        if (existing.getBillingPeriod().hasRetailOwner()) {
          existing.register(item);
        }
      }
    }
}
```

Nếu bạn làm việc trong một cơ sở code với code giống như vậy, nó có thể không đánh giá hết tất cả những gì xấu là dành cho bạn, nhưng nó tệ! Khi chúng ta trả về null, chúng ta tự tạo công việc cho bản thân và thêm vấn đề cho người gọi nó. Tất cả vấn đề chính là thiếu mất một kiểm tra null để rồi gửi ứng dụng ra khỏi tầm kiểm soát.

Thực tế là kiểm tra null ở if lồng thứ hai có được kiểm tra chưa? Điều gì xảy ra trong thời gian chạy nếu persistentStore là null? Chúng ta sẽ có một NullPointerException trong thời gian chạy, và một ai đó đang bắt NullPointerExceptionở cấp cao nhất hoặc không. Và điều nào cũng tồi tệ.

Giải quyết vấn đề chúng ta lại sử dụng một API bên thứ ba, nó cân nhắc bao gói phương thức cùng với phương thức hoặc ném các ngoại lệ hoặc trả về các đối tượng đặc biệt.

Trong nhiều trường hợp, vấn đề các đối tượng đặc biệt có biện pháp khắc phục dễ dàng. Ví dụ bạn có đoạn code sau:

```
List<Employee> employees = getEmployees();
if (employees != null) {
  for(Employee e : employees) {
    totalPay += e.getPay();
  }
}
```

Bây giờ getEmployees có thể trả về null, nhưng nó có cần thiết phải làm như vậy? Nếu chúng ta thay đổi getEmployees rằng nó trả về một danh sách rỗng, chúng ta có thể dọn dẹp code như sau:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
  totalPay += e.getPay();
}

public List<Employee> getEmployees() {
  if( .. there are no employees .. )
    return Collections.emptyList();
// Trả về một danh sách xác định trước không thay đổi trong Java
```

```
}
```

Nếu code như vậy sẽ giảm thiểu nguy cơ NullPointerExceptions và code sẽ sạch hơn.

Don't Pass Null

Trả về null từ phương thức đã xấu, nhưng vượt qua null còn tồi tệ hơn. Trừ khi bạn đang làm việc với một API hy vọng vượt qua null, bạn nên tránh passing null trong code bất cứ khi nào có thể,

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    return (p2.x – p1.x) * 1.5;
  }
  …
}
```

Điều gì xảy ra nếu đi qua một tham số gán null? calculator.xProjection(null, new Point(12, 13));

Chúng ta sẽ có một NullPointerException. Sửa chữa lại:

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    if (p1 == null || p2 == null) {
      throw InvalidArgumentException(
        "Invalid argument for MetricsCalculator.xProjection");
    }
    return (p2.x – p1.x) * 1.5;
  }
}
```

Có một thay thế khác tốt hơn:

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    assert p1 != null : "p1 should not be null";
    assert p2 != null : "p2 should not be null";
    return (p2.x – p1.x) * 1.5;
  }
```

}

Trong hầu hết các ngôn ngữ lập trình không có cách nào đối phó với việc thông qua một Null với một lời gọi vô tình. Bởi vì đây là trường hợp tiếp cận với cấm truyền Null theo mặc định. Khi bạn làm bạn có thể code với sự hiểu biết rằng một Null xuất hiện trong danh sách tham số là dấu hiệu của vấn đề và kết thúc nó với ít lỗi bất cẩn nhất.

Conclusion

Code sạch là code có thể đọc được, nhưng nó cũng cần phải mạnh mẽ. Đây không phải là điều mâu thuẫn. Chúng ta có thể viết code sạch và mạnh mẽ nếu chúng ta thấy được xử lý lỗi là một mối quan tâm riêng, đôi khi có thể xem nó như không phụ thuộc với logic cơ bản của chúng ta. Để đến mức độ chúng ta làm được điều đó, chúng ta cần lý giải về nó một cách độc lập, và chúng ta có thể tiến tới bước tiến lớn trong việc bảo trì code của chúng ta.

Chapter 10: Classes

Class Organization (Tổ chức của Class)

Theo quy ước, class nên bắt đầu với một danh sách các biến (variables). Hằng số tĩnh công khai (public static constants) nếu có nên để đầu tiên. Sau đó đến các biến private (private static variables). Hiếm khi có một lý do hợp lý để đặt biến công khai (public variable).

Public functions nên được thực hiện dưới danh sách các biến. Đặt các private function được gọi bởi một public function ngay sau public function đó. Điều này tuân theo quy tắc Stepdown và giúp chương trình được đọc như một bài báo.

Encapsulation
Chúng tôi muốn giữ các biến và các function ở chế độ private, nhưng cũng không nên cuồng tín quá về nó. Đôi khi chúng tôi giữ một biến hoặc method ở chế độ protected, do đó nó có thể được truy cập bởi một unit test. Đối với chúng tôi, test là nguyên tắc. Nếu một test ở trong cùng một package cần gọi một hàm hoặc truy cập vào một biến, chúng tôi sẽ để nó dạng protected hoặc package scope. Tuy nhiên, điều đầu tiên chúng ta làm là tìm kiếm một cách để duy trì sự riêng tư. Nới lỏng tính bao gói luôn là một cách thức cuối cùng.

=> List of variables: public static constants - private static variable - (public variable)

Classes Should Be Small

Nguyên tắc đầu tiên, các class nên được làm nhỏ. Nguyên tắc thứ hai, các class nên được làm nhỏ hơn nữa. Không, chúng ta sẽ không lặp lại chính xác những gì ở trong chương Functions. Nhưng, cũng như Functions, nhỏ hơn là nguyên tắc đầu tiên khi nói đến thiết kế classes. Như functions, câu hỏi trực tiếp được đặt ra luôn là "Nhỏ như thế nào?"

Với functions, chúng ta đo lường bằng cách đếm số dòng. Với Classes, chúng ta đo bằng cách khác, chúng ta đếm trách nhiệm.

Ví dụ về một class, SuperDashboard, với khoảng 70 public methods. Hầu hết các developer đều cho rằng nó có một siêu kích thước. Một số khác cho rằng nó như một "God class".

Tiếp theo cũng là SuperDashboard, nhưng chỉ chứa các phương pháp:

```
public class SuperDashboard extends JFrame implements MetaDataUser
  public Component getLastFocusedComponent()
  public void setLastFocused(Component lastFocused)
  public int getMajorVersionNumber()
  public int getMinorVersionNumber()
  public int getBuildNumber()
}
```

Năm methods không phải là nhiều đúng không? Trong trường hợp này có thể xem như là vậy, nhưng mặc dù số lượng method nhỏ, tuy nhiên SuperDashboard lại có nhiều trách nhiệm.

Tên class nên mô tả trách nhiệm những gì mà nó đáp ứng. Thực tế, việc đặt tên có lẽ là cách đầu tiên giúp xác định quy mô của một lớp. Nếu chúng ta không thể lấy được một cái tên ngắn gọn cho lớp, sau đó nó có khả năng là class quá lớn. Với một cái tên mơ hồ cho class, nhiều khả năng là nó đã chịu nhiều trách nhiệm.

Chúng ta có thể viết mô tả ngắn gọn cho class với khoảng 25 từ, mà không sử dụng những từ ngữ như "nếu", "và", "hoặc" hay "nhưng".

The Single Responsibility Principle
Một class hoặc module nên có một và chỉ một lý do để thay đổi. Nguyên tắc này cho chúng ta một định nghĩa của trách nhiệm và cả chủ trương về quy mô của class. Class nên có một trách nhiệm - một lý do duy nhất để thay đổi.

Ví dụ ở trên thì class SuperDashboard có hai lý do để thay đổi. Đầu tiên, nó theo dõi thông tin các version và dường như cần phải cập nhật mỗi khi phần mềm được vận chuyển. Thứ hai, nó quản lý thành phần Java Swing (một dẫn xuất của JFrame, Swing đại diện của một giao diện

window ở cấp cao). Không nghi ngờ khi chúng ta sẽ cập nhật version number nếu chúng ta thay đổi bất cứ gì ở trong Swing code, nhưng theo chiều ngược lại thì không nhất thiết là đúng: chúng ta có thể thay đổi thông tin version dựa trên sự thay đổi code chỗ khác ở trên hệ thống.

Cố gắng đồng nhất trách nhiệm thường giúp chúng ta nhìn nhận được và tạo ra sự trừu tượng hóa tốt hơn trong code. Chúng ta dễ dàng lấy ra 3 method trong SuperDashboard phân phối thông tin của version để tách ra một class tên là Version. Lớp này có tiềm năng để sử dụng lại ở trong các ứng dụng khác!

```
public class Version {
  public int getMajorVersionNumber()
  public int getMinorVersionNumber()
  public int getBuildNumber()
}
```

SRP là một trong những khái niệm quan trọng trong việc thiết kế hướng đối tượng. Nó cũng là một khái niệm đơn giản để hiểu và tuân thủ. Tuy nhiên, khó đỡ là SRP lại là nguyên tắc thường bị sỉ nhục nhất :)) Chúng tôi thường gặp phải những class mà làm quá nhiều thứ. Tại sao?

Để phần mềm chạy được và làm cho phần mềm sạch sẽ là hai việc hoàn toàn khác nhau. Hầu hết chúng ta đều bị giới hạn trong đầu là ưu tiên việc làm cho code chúng ta chạy được hơn là tổ chức và làm cho nó sạch sẽ. Điều này hoàn toàn phù hợp. Việc duy trì tách rời các quan hệ chỉ quan trọng khi chương trình của chúng ta hoạt động.

Vấn đề ở đây là có quá nhiều người trong chúng ta nghĩ rằng chúng ta chỉ đang thực hiện để chương trình hoạt động xong sau một lần. Chúng ta thất bại trong việc tổ chức lại và dọn dẹp chúng. Chúng ta chuyển sang các vấn đề khác hơn là trở lại và phá vỡ các lớp để nhồi nhét theo nguyên tắc mỗi class nên chịu một trách nhiệm duy nhất.

Đồng thời, nhiều developer lo ngại rằng một lượng lớn các class nhỏ với mục đích duy nhất sẽ làm khó khăn trong việc hiểu bức tranh tổng quát lớn. Họ lo lắng cho việc phải đi từ lớp này đến lớp khác để hiểu làm thế nào một mảnh lớn của công việc được hoàn thành.

Tuy nhiên hệ thống với nhiều class nhỏ cũng không di chuyển nhiều hơn so với ít class lớn là bao. Nó cũng nhiều như khi đọc hiểu một hệ thống với ít class lớn. Vậy nên câu hỏi ở đây là: Bạn muốn công cụ của bạn với nhiều ngăn kéo nhỏ, mỗi ngăn từng thành phần với khái niệm và nhãn hiệu riêng? Hay bạn muốn một ngăn kéo và tống khứ tất cả mọi thứ vào đó

Mỗi hệ thống lớn sẽ chứa đựng một số lượng lớn logic và độ phức tạp. Mục tiêu chính trong việc quản lý độ phức tạp như vậy là để tổ chức nó sao cho developer biết nơi để tìm thấy mọi thứ và chỉ cần hiểu được độ phức tạp của vấn đề cần tìm hiểu một cách tức thì. Ngược lại, một hệ thống với các class lớn và chứa đựng nhiều mục đích luôn cản trở chúng ta bằng cách cố bắt chúng ta lội qua rất nhiều thứ mà không nhất thiết cần phải biết ngay bây giờ.

=> Chúng tôi muốn hệ thống của chúng tôi hình thành bởi nhiều class nhỏ, không phải ít class nhưng to lớn. Mỗi class nhỏ tóm gọn bởi một trách nhiệm duy nhất, chỉ có một lý do duy nhất để thay đổi, tương tác với một vài đối tượng khác để đạt được trạng thái mà hệ thống muốn có.

Cohesion

Class nên có số lượng biến ít. Mỗi phương thức của một class nên thao tác với một hoặc nhiều biến đó. Mỗi biến ở trong class được sử dụng bởi mỗi phương thức nên có sự kết dính tối đa.

```
public class Stack {
  private int topOfStack = 0;
  List<Integer> elements = new LinkedList<Integer>();

  public int size() {
    return topOfStack;
  }

  public void push(int element) {
    topOfStack++;
    elements.add(element);
  }

  public int pop() throws PoppedWhenEmpty {
    if (topOfStack == 0)
      throw new PoppedWhenEmpty();
    int element = elements.get(--topOfStack);
    elements.remove(topOfStack);
    return element;
  }
}
```

Sự kết dính tối đa ở đây là các phương thức và các biến của class cùng phụ thuộc và có sự gắn kết với nhau như một tổng thể hợp lý.

Nói một cách đơn giản thì class nên hạn chế số lượng biến, và các method ở trong class đều sử dụng ít nhất một biến trong số này để gia tăng sự kết dính.

Maintaining Cohesion Results in Many Small Classes

Nếu chia function lớn thành nhiều function nhỏ hơn thì điều này đồng thời cùng làm gia tăng kích thước class. Giả sử rằng, function lớn với nhiều biến được khai báo ở trong nó. Bạn muốn

trích một hàm nhỏ hơn thì hàm đó ra. Tuy nhiên code của bạn muốn sử dụng 4 biến được khai báo ở trong hàm. Bạn có cần phải pass cả 4 biến đó vào hàm mới như một tham số không?

Not at all! Nếu chúng ta để 4 biến đó vào biến chung của class, sau đó chúng ta có thể trích xuất code mà không cần pass bất kỳ biến nào ở đó. Nó thật dễ dàng để chia function ra thành các mảnh nhỏ.

Không may thay, điều đó làm cho class không còn được kết dính chặt chẽ nữa, bởi vì chúng ta càng ngày càng chất đống biến vào class trong khi chỉ một vài chức năng cần chia sẻ chúng.

Vậy nên, nếu một vài function muốn chia sẻ một số biến nhất định, tại sao chúng ta không tạo một class cho riêng chúng? Điều đó là tất nhiên, nếu class đã mất đi đoàn kết, hãy chia rẽ nó!

Organizing for Change

Đối với mọi hệ thống, sự thay đổi là liên tục. Mỗi sự thay đổi đều dẫn đến rủi ro về phần còn lại của hệ thống không còn hoạt động như dự định. Trong một hệ thống sạch, chúng ta tổ chức các class để làm giảm nguy cơ của sự thay đổi.

Ví dụ: Class Sql - Vi phạm SRP

```
public class Sql {
  public Sql(String table, Column[] columns)
  public String create()
  public String insert(Object[] fields)
  public String selectAll()
  public String findByKey(String keyColumn, String keyValue)
  public String select(Column column, String pattern)
  public String select(Criteria criteria)
  public String preparedInsert()
  private String columnList(Column[] columns)
  private String valuesList(Object[] fields, final Column[] columns)
  private String selectWithCriteria(String criteria)
  private String placeholderList(Column[] columns)
}
```

Sửa lại:

```
abstract public class Sql {
  public Sql(String table, Column[] columns)
  abstract public String generate();
}
```

```
public class CreateSql extends Sql {
  public CreateSql(String table, Column[] columns)
  @Override public String generate()
}

public class SelectSql extends Sql {
  public SelectSql(String table, Column[] columns)
  @Override public String generate()
}

public class InsertSql extends Sql {
  public InsertSql(String table, Column[] columns, Object[] fields)
  @Override public String generate()
  private String valuesList(Object[] fields, final Column[] columns)
}

public class SelectWithCriteriaSql extends Sql {
  public SelectWithCriteriaSql(
    String table, Column[] columns, Criteria criteria)
  @Override public String generate()
}

public class SelectWithMatchSql extends Sql {
  public SelectWithMatchSql(
    String table, Column[] columns, Column column, String pattern)
  @Override public String generate()
}

public class FindByKeySql extends Sql
  public FindByKeySql(
    String table, Column[] columns, String keyColumn, String keyValue)
  @Override public String generate()
}

public class PreparedInsertSql extends Sql {
  public PreparedInsertSql(String table, Column[] columns)
  @Override public String generate() {
  private String placeholderList(Column[] columns)
}

public class Where {
  public Where(String criteria)
  public String generate()
```

```
}

public class ColumnList {
  public ColumnList(Column[] columns)
  public String generate()
}
```

=> Chúng tôi cấu trúc hệ thống của chúng tôi rác rưởi càng ít càng tốt khi chúng tôi cập nhật chúng với tính năng mới hay thay đổi tính năng. Trong một hệ thống lý tưởng, chúng tôi kết hợp tính năng mới bằng cách mở rộng hệ thống chứ không phải thay đổi code của hiện tại.

Isolating from Change (Loose Coupling - Dependency Invertion Principle)

Nhu cầu sẽ thay đổi, vậy nên code sẽ luôn thay đổi. Trong một class cụ thể, chứa những chi tiết hiện thực (code), các class trừu tượng cùng với đó là đại diện cho khái niệm. Một client class phụ thuộc vào một chi tiết cụ thể là nguy cơ khi những chi tiết thay đổi. Chúng ta có thể thông qua các interface và abstract class để giúp cho việc cách lý việc va chạm với các chi tiết đó.

Phụ thuộc vào một chi tiết cụ thể tạo ra những thách thức cho việc kiểm thử hệ thống.

Ví dụ: Chúng ta xây dựng một class Portfolio và nó phụ thuộc vào TokyoStockExchange API ở bên ngoài để lấy được giá trị của danh mục đầu tư. Test case sẽ bị tác động bởi sự biến động của một tra cứu như vậy. Nó rất khó để viết test khi câu trả lời của chúng ta sẽ khác đi chỉ sau 5 phút.

Thay vì thiết kế Portfolio để nó phụ thuộc trực tiếp vào TokyoStockExchange, chúng ta tạo một interface, StockExchange, và khai báo một method duy nhất:

```
public interface StockExchange {
  Money currentPrice(String symbol);
}

public Portfolio {
  private StockExchange exchange;
  public Portfolio(StockExchange exchange) {
    this.exchange = exchange;
  }
  // ...
}
```

Bây giờ test của chúng ta có thể thực hiện trên StockExchange interface giả lập cho TokyoStockExchange.

Nếu một hệ thống được tách rời, nó sẽ linh hoạt hơn và thúc đẩy việc tái sử dụng. Hạn chế coupling nghĩa là các yếu tố trong hệ thống đang bị cô lập từ những yếu tố khác và từ những sự thay đổi. Sự cô lập làm cho mỗi thành phần trong hệ thống trở nên dễ hiểu hơn.