Số phát hành Vol.7
09
2020

Tạp_Chí
{LậpTrình ():}

Tạp chí LẬP TRÌNH

tapchilaptrinh.vn

CLEAN CODE

MỤC LỤC

04 Clean code là gì?

07 Simple Design – Chìa khóa của mã sạch

11 Comment trong Clean Code

14 Expressive Clean Code

18 Lợi ích của Clean Code

21 Định dạng mã nguồn

24 Mã sạch, chất lượng cao:
Cách trở thành một lập trình viên giỏi hơn.

VOL. 7

LỜI MỞ ĐẦU

Quý bạn đọc thân mến,

Mã sạch là tiêu chuẩn đầu tiên được sử dụng để đánh giá về chất lượng của mã nguồn, hoàn toàn có thể khẳng định như vậy. Hay nói theo một cách khác, thật khó chấp nhận việc một lập trình viên lành nghề viết ra những dòng mã mà không tuân thủ những gợi ý mà cộng đồng lập trình viên đã công nhận là "tốt". Tuy vậy, thật đáng ngạc nhiên khi có một khảo sát(*) đã chỉ ra rằng gần 60% lập trình viên không hề biết đến các khái niệm liên quan như mã bẩn, mã sạch và tái cấu trúc mã nguồn.

Đối với những người mới bắt đầu đến với công việc lập trình, đôi khi chúng ta dành tất cả sự chú ý vào việc giải quyết vấn đề mà quên mất đi việc tuân thủ những "tiêu chuẩn" quan trọng về mã nguồn. Điều này dẫn đến mã nguồn trở nên rối rắm, khó đọc, khó hiểu, khó nâng cấp, khó thay đổi, khó tích hợp... Thậm chí, có trường hợp dẫn đến tình trạng không kiểm soát được mã nguồn, nếu sửa chỗ này thì sẽ ảnh hưởng rất nhiều đến những nơi khác, và hệ thống sẽ bị hỏng.

Ấn phẩm lần này của Tạp chí Lập trình mong muốn phổ biến lại đến cộng đồng những khái niệm không còn là mới mẻ gì nữa, nhưng lại là những tiêu chuẩn không thể thiếu được mà bất cứ lập trình viên nào cũng phải biết và áp dụng. Cùng với các nội dung liên quan đến Design Pattern, Refactoring, Automated Testing, Agile Developer... ở các ấn phẩm trước đó, BBT Tạp chí Lập trình mong rằng mỗi lập trình viên sẽ có ý thức hơn, kỷ luật hơn và trân trọng hơn những dòng mã của mình; Đây là những bước tiến đầu tiên trên chặng đường nâng cao chất lượng phần mềm, nâng cao tiêu chuẩn ngành nghề và bước vào hàng ngũ của những lập trình viên tốt.

Chúc bạn đọc thật nhiều tiến bộ.

Ban biên tập Tạp chí Lập trình

(*) *Khảo sát của Phòng nghiên cứu Simula*

C.L.E.A.N. C.O.D.E.

CLEAN CODE LÀ GÌ?

Dịch: Dương Nhật Minh

Trong một lĩnh vực rất rộng như software engineering, việc đọc sách liên tục là điều cần thiết để xây dựng được kiến thức nền tảng vững chắc và cập nhật nhiều hơn những kiến thức mới. Trong bất kỳ ngành nghề nào cũng luôn có một hay nhiều những cuốn sách được cho là phải đọc qua. Và đối với software engineering thì cuốn sách đó là Clean Code của R.Martin. Dưới đây là một số bài học kinh nghiệm từ qua trải nghiệm đọc cuốn sách này của tôi.

Dưới nhiều góc nhìn, Clean Code có cấu trúc giống như sổ tay hướng dẫn sử dụng của lập trình viên. Đây là một cuốn sách mang dấu ấn cá nhân mãnh liệt của tác giả, giống như hầu hết các cuốn sách hay.

Martin mô tả các phương pháp tốt nhất, khuyên răn những người chưa nhận ra những thói quen xấu của mình khi lập trình. Martin rất cởi mở về những thất bại của mình. Với nhiều năm kinh nghiệm trong việc đọc (và đôi khi viết) mã xấu đã khiến ông ta có khuynh hướng giúp các dev khác tránh khỏi những điều không nên. Tất cả những kết hợp lại với nhau đã tạo ra hàng trăm đề xuất đáng để làm theo. Điều đó nói rằng, hầu hết các dev đều học code đúng qua các sai lầm- bằng cách trải qua đủ thứ mã tồi mà họ muốn làm cho nó tốt hơn.

Nhưng Clean Code còn nhiều điều hơn thay vì chỉ là một bản hướng dẫn. Nó không chỉ mô tả cách thức mà còn là lý do tại sao để viết mã tốt. Hơn cả việc học thuộc lòng các quy tắc (mặc dù nhiều quy tắc của cuốn sách gắn bó với tôi),

tôi đã đọc để hiểu ra những giá trị bất khả xâm phạm của một lập trình viên giỏi. Đối với tôi, tất cả xoay quanh ba khái niệm chính.

1. Các vấn đề xung quanh sản phẩm

Sẽ hiếm có khi nào mà một lập trình viên có thể tự do ngồi với một đoạn mã cho đến khi nó trở nên hoàn hảo (hoàn hảo ở đây chỉ là ảo tưởng chúng ta cố nhắm đến, sẽ luôn có sai sót). Tuy nhiên, viết một đoạn mã tốt nhất có thể luôn là ưu tiên hàng đầu của chúng ta. Đôi khi, điều đó có thể đồng nghĩa với việc lùi lại deadline, yêu cầu của khách hàng, yêu cầu quản lý, đi kèm với đó là sự mệt mỏi. Về phía công việc luôn đặt ra ranh giới về những gì "có thể", nhưng một lập trình viên ít nhất cũng phải có sự kiểm soát chất lượng với những gì anh ta hoặc cô ta đã viết ra.

Điểm mấu chốt ở đây là: Mã "hoạt động" không phải lúc nào cũng đồng nghĩa với "xong". Nếu công việc của một lập trình viên là viết ra một sản phẩm để giải quyết một vấn đề, thì liệu cái cách mà vấn đề đó được giải quyết như thế nào có quan trọng không? Câu trả lời là có. Phần mềm được viết tốt sẽ mang lại lợi ích cho tất cả mọi người về mặt lâu dài - khách hàng, người dùng, công ty và lập trình viên. Mã lộn xộn, thiếu sự chăm chút, hoặc "đủ tốt" là mã tập trung vào phần chuyển giao được trong ngắn hạn.

Martin cho rằng chìa khóa để viết mã xuất sắc là đầu vào chất lượng cao (mã được viết với sự chu đáo, dễ bảo trì, linh hoạt) dẫn đến đầu ra chất lượng cao (lợi nhuận kinh doanh dài hạn).

2. Nỗ lực nhiều hơn vào hôm nay sẽ giảm vất vả cho ngày mai

Tất cả chúng ta chắc đều đã trải qua nỗi thất vọng khi sử dụng một sản phẩm chất lượng kém. Hãy nghĩ đến chiếc áo sơ mi mới bị sờn mép chỉ sau vài lần cho qua máy giặt hoặc đồ chơi bằng nhựa bị vỡ chỉ sau khi con bạn lấy nó ra khỏi hộp. Mã được viết kém cũng không khác gì - nó sẽ không khác gì là một công tắc tự hủy trong tương lai rất gần.

Nguyên tắc tương tự cũng áp dụng cho mã. Đầu vào chất lượng cao không chỉ tạo ra sản phẩm có giá trị lâu dài hơn mà còn tiết kiệm chi phí và xây dựng lòng trung thành với khách hàng. Việc tái cấu trúc và kiểm tra mã của bạn một cách tỉ mỉ có thể mất nhiều thời gian lúc mới bắt đầu, nhưng nó sẽ rất giá trị trong quá trình phát triển và bảo trì sau này.

3. Mã của bạn không phải của riêng bạn

Bạn tự hào về công việc của mình, điều đó là rất quan trọng. Và một điều cũng quan trọng không kém là bạn nhận ra rằng mã bạn viết không phải là của riêng bạn.

Điều này có vẻ hơi nghịch lý - sau tất cả, việc viết mã sạch là tùy thuộc ở bạn. Tuy nhiên, mã nguồn tốt không thực sự là của bạn vì nó theo lý thì là dành cho những người khác: đồng đội, khách hàng của bạn và thậm chí cả bản thân bạn trong tương lai. Điều gì xảy ra khi một số người bảo trì dự án trong tương lai (thậm chí có thể là chính bạn) không thể hiểu tại sao bạn lại viết những dòng mã như vậy? Họ chắc chắn sẽ phải dành nhiều thời gian và năng lượng để giải mã mớ bòng bong mà bạn đã để lại.

Clean Code là gì?

Ai cũng có có check-list riêng, tiêu chuẩn riêng của mình về mã sạch, cuốn sách của Martin bắt đầu với một bộ sưu tập các định nghĩa được đưa ra bởi các nhà sáng chế phần mềm khác nhau. Dựa trên những gì tôi đã học được trong sách. Định nghĩa của tôi là như sau:

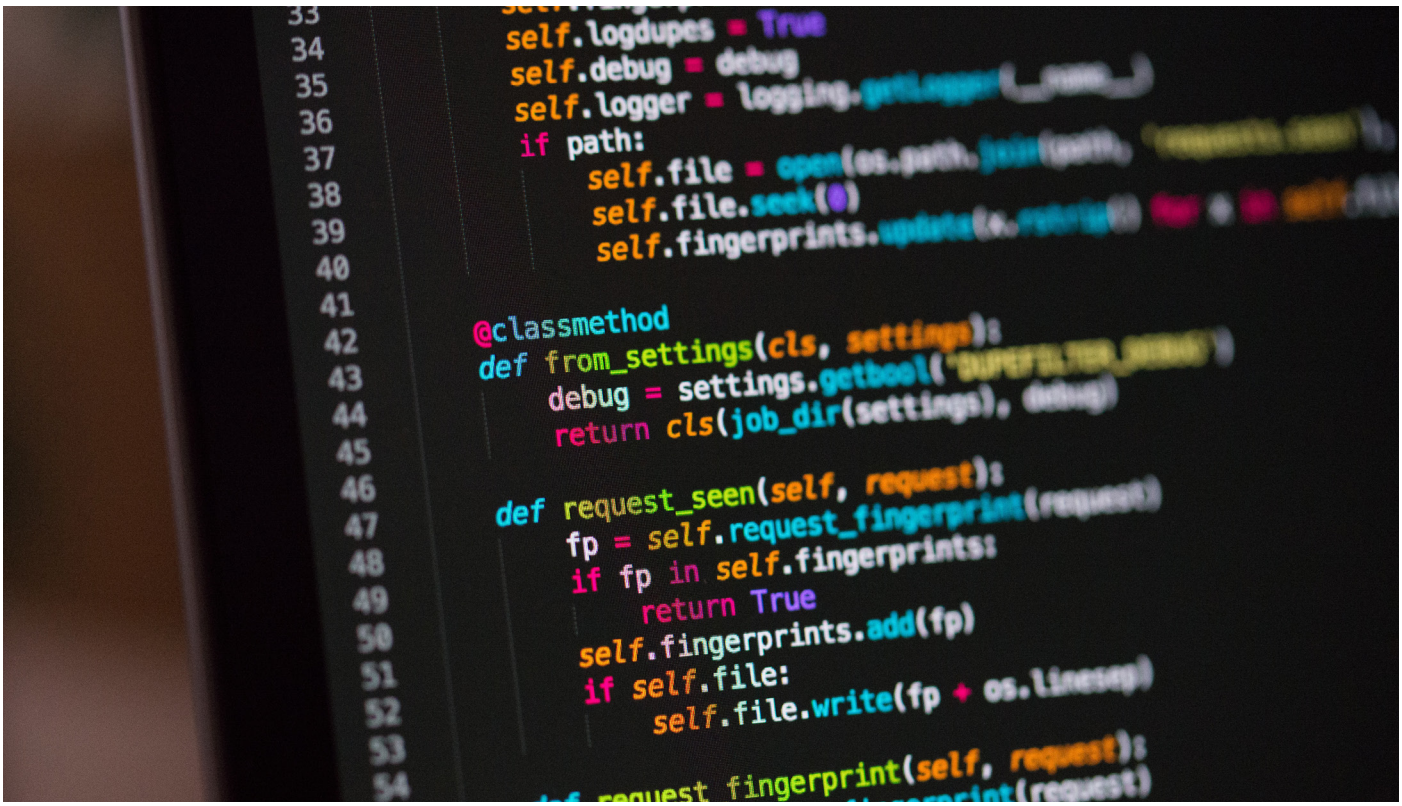
1. Mã sạch phải thật đơn giản. Có lẽ không đơn giản ở mức độ thuật toán hoặc hệ thống, nhưng chắc chắn là đơn giản trong việc thực hiện. Các thủ thuật, hack và đường tắt trong quá trình viết mã sẽ chỉ mang lại niềm vui cho người viết mà thôi và làm giảm giá trị lâu dài của mã. Điều tương tự như vậy cũng xảy ra với những đoạn mã dài dòng, mất nhiều thời gian để đi đến vấn đề.
2. Mã sạch phải dễ đọc. Nếu các quy ước đặt tên, đặt khoảng cách, cấu trúc và luồng được sử dụng trong một chương trình được thiết kế mà không chú trọng đến góc nhìn của những người phải đọc nó, thì người đọc đó gần như chắc chắn sẽ không hiểu được ý định của tác giả ban đầu. Các quy ước về cách viết mã có

thể đọc được có vẻ giáo điều hoặc thiếu tính diễn đạt, nhưng chúng làm cho mã trở nên mang tính cộng đồng hơn.

3. Mã sạch là sự chu đáo. Mã không hoạt động tốt trong việc truyền đạt đến người đọc trong tương lai là những dòng mã thiếu trách nhiệm không tôn trọng thời gian của họ. Mã sạch nên được viết với giả định rằng người tiêu dùng trong tương lai là người thông minh, chu đáo (như bạn) và nó phải cố gắng hết sức để giúp họ.
4. Mã sạch là đã được test kĩ càng. Không ai viết mã hoàn hảo, không có lỗi trong lần thử đầu tiên. Ngay cả khi có thể làm như vậy, không có gì đảm bảo rằng đoạn mã hoàn hảo đó sẽ không bị hỏng sau này. Viết mã sạch nghĩa là viết mã đã kiểm tra. Bằng cách đó, người dùng trong tương lai có thể tự tin rằng họ đang dùng một thứ gì đó hiệu quả, đã kiểm chứng. Hơn nữa, khi thực hiện thay đổi, họ sẽ có một bộ thử nghiệm được tạo sẵn để xác nhận rằng không có gì bị sai lệch với lúc đầu.
5. Mã sạch được phải được trau dồi thường xuyên. Viết mã sạch đòi hỏi trí nhớ, cơ bắp của bạn ghi nhớ nó , giống như việc chơi nhạc cụ, đá bóng hoặc rán trứng. Cách tốt nhất để học viết mã sạch - và quan trọng hơn là giữ được kỹ năng đó - là làm điều đó mọi lúc. Khi bạn đang ở nhà làm một dự án cá nhân, hãy làm điều đó với mã sạch - ngay cả khi có thể sẽ không ai nhìn thấy nó.
6. Mã sạch được tái cấu trúc liên tục. Mã sạch phải ở trạng thái tái cấu trúc liên tục. Với một nền tảng tốt để sao lưu mã của bạn, bạn có thể cấu trúc lại mã tùy thích và không bao giờ lo lắng về việc bị hỏng.
7. Mã sạch tuân theo bộ nguyên tắc SOLID. Mã tốt cũng giống với một mẫu thiết kế đẹp ở chỗ cũng đề cao sự sạch sẽ. Tuân theo các nguyên tắc SOLID là một cách để đảm bảo rằng mã của bạn linh hoạt, có thể bảo trì và lâu dài

Nếu chúng ta coi trọng sách của Martin như một cuốn kinh thánh, thì các yếu tố tạo nên "mã sạch" có thể nhân lên theo cấp số nhân. Nhưng danh sách trên, được chọn lọc từ việc qua việc đọc kỹ Clean Code, có lẽ là một nơi tốt để tham khảo. Việc tuân thủ những nguyên tắc này đòi hỏi sự chú ý liên tục đến từng chi tiết và sẵn sàng thừa nhận rằng những gì bạn đã viết ngày hôm qua vẫn có thể được cải thiện vào ngày hôm nay - ngay cả khi nó hoạt động tốt trước đó.

Nguồn: https://medium.com/s/story/reflections-on-clean-code-8c9b683277ca



SIMPLE DESIGN
CHÌA KHÓA CỦA MÃ SẠCH

Nguyễn Bình Sơn

Các mùi xấu của mã là các dấu hiệu cho thấy có vấn đề ở trong mã. Khử đi các dấu hiệu xấu cũng giống như đang băng một vết thương. Tổn thương có thể sẽ lành lại hoặc cũng có thể chỉ được chữa trị ở bề ngoài. Tương tự như thế khi chúng ta thực hiện khử mùi xấu cho mã, vấn đề của mã có thể thực sự biến mất hoặc cũng có thể vẫn còn tồn tại ở một dạng khác.

Làm thế nào để có thể nhận diện được mã còn vấn đề hay không? Thông thường thì bằng cách kiểm tra xem có xuất hiện dấu hiệu mã xấu khác hay không. Cũng giống như thực hiện thêm xét nghiệm hay tìm đến một bác sĩ giỏi hơn. Tuy vậy hiệu quả của cách làm này phụ thuộc rất nhiều vào kinh nghiệm. Liệu có ở đâu đó một phương án tổng quát, đơn giản, thanh nhã và hiệu quả để phát hiện và gợi ý được giải pháp cải thiện chất lượng của mã nguồn hay không?

Bài viết này đề cập đến thiết kế, mà cụ thể là đề cập đến khung thiết kế Simple Design. Nhưng trọng tâm ở đây là mã sạch. Nói như vậy là bởi đứng sau Simple Design là các nguyên tắc thiết kế đơn giản và phổ quát mà có thể áp dụng ngay lập tức vào bất cứ thời điểm nào của công việc viết mã, giúp khử đi tất cả các mã xấu và tạo ra mã tốt nhất có thể có, trên tất cả các phạm vi của mã nguồn, từ từng dòng mã nhỏ lên tới cấp độ module hay cao hơn.

Thiết kế đơn giản

Cái tên Simple Design được giới thiệu bởi *Kent Beck* (nhà sáng lập *Extreme Programming*) thông qua nhiều phát ngôn và tài liệu khác nhau, và sau đó được xuất hiện chính thức trong cuốn *Sách Trắng* của ông. Theo đó, một thiết kế được gọi là "đơn giản" khi nó tuân thủ bốn nguyên tắc:

- Vượt qua tất cả các kiểm thử
- Không có tính trùng lắp
- Thể hiện rõ ý định của lập trình viên
- Có số lượng lớp và phương thức ở mức tối thiểu

Các nguyên tắc trên được sắp xếp theo thứ tự độ quan trọng giảm dần. Sau đây là diễn giải chi tiết.

Vượt qua tất cả các kiểm thử

Đặc điểm đầu tiên và quan trọng nhất của một thiết kế tốt là hệ thống phải hoạt động như dự kiến. Một hệ thống chỉ được thiết kế tốt "trên giấy" là rất đáng ngờ. Hệ thống phải luôn giữ được tính chất "có thể kiểm thử", nghĩa là nó phải có một bộ kiểm thử toàn diện, và luôn vượt qua tất cả các kiểm thử đó. Một hệ thống không thể kiểm thử là một hệ thống không đáng tin cậy. Và một hệ thống không đáng tin cậy thì không bao giờ nên deploy.

Cái hay ở đây là nỗ lực để tạo nên một hệ thống phần mềm có thể kiểm thử sẽ thúc đẩy chúng ta tạo ra một thiết kế với các thành phần đủ nhỏ và đơn trách nhiệm, bởi như thế sẽ dễ viết kiểm thử hơn. Tương tự, các mã bị couping luôn gây khó khăn khi viết kiểm thử. Và thế là càng viết nhiều kiểm thử, chúng ta càng có xu hướng áp dụng nhiều các nguyên tắc như Đảo Ngược Phụ Thuộc, các công cụ như tiêm phụ thuộc, giao diện, và trừu tượng, để tránh mã bị couping.

Nguyên tắc phát biểu một cách đơn giản và hiển nhiên rằng hệ thống cần được kiểm thử. Và điều đó lại dẫn đến sự tuân thủ của hệ thống trước các nguyên tắc thiết kế hướng đối tượng. Nghĩa là càng viết nhiều kiểm thử, thiết kế của hệ thống càng trở nên tốt hơn.

Kiểm thử, Tái cấu trúc, và ba nguyên tắc còn lại

Có được kiểm thử nghĩa là chúng ta đã nắm được trong tay sức mạnh để giữ mã nguồn được sạch sẽ. Điều đó được thực hiện bằng các thao tác tái cấu trúc. Sau mỗi vài dòng mã mới, chúng ta tạm dừng và suy nghĩ về thiết kế mới hình thành. "Có phải thiết kế vừa yếu đi không?". Nếu đúng như vậy thì hành động tiếp theo sẽ là dọn sạch những mã xấu và chạy lại kiểm thử để chắc chắn rằng không có chức năng nào bị hỏng. Sự tồn tại của kiểm thử giúp chúng ta vượt qua được các rủi ro khi tái cấu trúc mã.

Khi tái cấu trúc, chúng ta có thể áp dụng bất kỳ kiến thức nào về phần mềm tốt. Có thể tăng tính cố kết, giảm couping, chia tách các khía cạnh, mô-đun hóa, tách nhỏ các hàm và lớp đối tượng, đặt lại tên biến... Tất cả những điều vừa liệt kê là đất diễn của ba nguyên tắc còn lại trong bộ nguyên tắc Thiết Kế Đơn Giản: loại bỏ sự trùng lắp, đảm bảo sự rõ ý, và giảm tối thiểu số lượng các lớp và phương thức.

Không trùng lặp

Lặp là kẻ thù số một của thiết kế. Lặp đại diện cho thêm việc, thêm rủi ro, và thêm sự phức tạp không cần thiết trong thiết kế. Lưu ý rằng sự trùng lắp có thể xảy ra dưới nhiều hình thức. Mã hoàn toàn giống nhau tất nhiên là mã lặp. Mã gần giống nhau cũng thế, và chúng có thể được nắn lại cho giống nhau hơn nữa nhằm tạo tiền đề cho tái cấu trúc. Nhưng trùng lắp còn cũng có thể tồn tại ở những dạng thức khó nhận ra hơn, chẳng hạn như dưới đây, chúng ta có thể có hai phương thức như sau trong cùng một lớp:

```
int size() {}
boolean isEmpty() {}
```

Hai phương thức trên có thể được triển khai riêng biệt. Có thể là `isEmpty` theo dõi một biến `boolean` trong khi `size` thì dựa trên một biến đếm. Không có mã lặp, nhưng đây là một sự trùng lắp về mặt chức năng. Chúng ta có thể loại bỏ sự trùng lắp này bằng cách sử dụng `isEmpty` trong mã của `size`:

```
boolean isEmpty() {
  return 0 == size();
}
```

Tạo ra một hệ thống sạch yêu cầu ý chí và nỗ lực loại bỏ sự trùng lắp, kể cả trên chỉ một vài dòng mã. Hãy xem xét ví dụ dưới đây:

```
public void scaleToOneDimension(
    float desiredDimension, float imageDimension) {
  if (Math.abs(desiredDimension - imageDimen-
  sion) < errorThreshold)
      return;
  float scalingFactor = desiredDimension / image-
  Dimension;
  scalingFactor = (float)(Math.floor(scalingFactor
  * 100) * 0.01f);
  RenderedOp newImage = ImageUtilities.getScale-
  dImage(
    image, scalingFactor, scalingFactor);
  image.dispose();
  System.gc();
  image = newImage;
}

public synchronized void rotate(int degrees) {
  RenderedOp newImage = ImageUtilities.getRota-
  tedImage(
    image, degrees);
  image.dispose();
  System.gc();
  image = newImage;
}
```

Có một sự trùng lắp nhỏ giữa hai phương thức `scaleToOneDimension` và `rotate` mà chúng ta có thể tái cấu trúc lại:

```
public void scaleToOneDimension(
      float desiredDimension, float imageDimen-
      sion) {
  if (Math.abs(desiredDimension - imageDimen-
  sion) < errorThreshold)
      return;
  float scalingFactor = desiredDimension / image-
  Dimension;
  scalingFactor = (float)(Math.floor(scalingFactor
  * 100) * 0.01f);
  replaceImage(ImageUtilities.getScaledImage(
      image, scalingFactor, scalingFactor));
}

public synchronized void rotate(int degrees) {
```

```
        replaceImage(ImageUtilities.getRotatedIm-
age(image, degrees));
}

privatex void replaceImage
(RenderedOp newImage) {
  image.dispose();
  System.gc();
  image = newImage;
}
```

Sự trích xuất phần mã bị trùng lắp đã làm lộ diện dấu hiệu vi phạm nguyên tắc Đơn Trách Nhiệm. Phương thức mới được trích xuất nên được chuyển sang một lớp khác. Điều này có thể nâng cao khả năng sử dụng của nó. Sau đó một nhà phát triển khác có thể sẽ nhận ra cơ hội để tiếp tục trừu tượng hóa phương thức mới nhằm tái sử dụng nó trong một bối cảnh khác. Vậy là thao tác "tái sử dụng nhỏ" này có thể khiến độ phức tạp của hệ thống giảm đi đáng kể. Hiểu cách để tạo ra khả năng tái sử dụng ở quy mô nhỏ là điều cần thiết để có thể đạt được khả năng tái sử dụng ỏ quy mô lớn.

Sử dụng mẫu thiết kế TEMPLATE METHOD là kỹ thuật thường thấy để xóa bỏ những sự trùng lắp ở mã mức cao. Dưới đây là ví dụ về hai phương thức dùng để tính kỳ nghỉ tích lũy, trong đó `accrueUSDivisionVacation` tính theo luật Hoa Kỳ và `accrueEUDivisionVacation` tính theo luật EU:

```
public class VacationPolicy {
  public void accrueUSDivisionVacation() {
    // mã tính kỳ nghỉ theo số giờ làm việc tích
    lũy
    // ...
    // mã để đáp ứng luật về kỳ nghỉ tối thiểu
    theo luật Hoa Kỳ
    // ...
    // mã để khớp với nhật ký bảng lương
    // ...
}

public void accrueEUDivisionVacation() {
    // mã tính kỳ nghỉ theo số giờ làm việc tích
    lũy
    // ...
    // mã để đáp ứng luật về kỳ nghỉ tối thiểu
    theo luật EU
    // ...
    // mã để khớp với nhật ký bảng lương
    // ...
  }
}
```

Mã của chúng phần lớn giống nhau, ngoại trừ các dòng mã để đáp ứng luật về kỳ nghỉ tối thiểu. Những mã trùng lắp có thể được khử bằng cách áp dụng mẫu thiết kế TEMPALTE METHOD:

```
abstract public class VacationPolicy {
  public void accrueVacation() {
    calculateBaseVacationHours();
    alterForLegalMinimums();
    applyToPayroll();
  }

  private void calculateBaseVacationHours() { /*
  mã tính kỳ nghỉ theo số giờ làm việc tích lũy
  */ };
  abstract protected void alterForLegalMini-
  mums();
  private void applyToPayroll() { /* mã để khớp
  với nhật ký bảng lương */ };
}

public class USVacationPolicy extends Vacation-
Policy {
  @Override
  protected void alterForLegalMinimums() {
    // mã để đáp ứng luật về kỳ nghỉ tối thiểu
    theo luật Hoa Kỳ
  }
}

public class EUVacationPolicy extends Vacation-
Policy {
  @Override
  protected void alterForLegalMinimums() {
    // mã để đáp ứng luật về kỳ nghỉ tối thiểu
    theo luật EU
  }
}
```

Các lớp con đã "điền vào chỗ trống" của thuật toán accrueVacation, cung cấp các chỉ dẫn không bị trùng lặp.

Rõ ý

Phần lớn chúng ta đều đã gặp phải mã khó đọc. Và nhiều khi chúng là do tự chúng ta viết ra. Thật dễ dàng hiểu mã do chính mình viết, vào lúc chính mình đang viết, bởi lúc đó chúng ta hiểu sâu sắc vấn đề mà mình đang đối diện. Ai đó khác, vào một thời điểm khác sẽ không có cơ hội như thế.

Phần lớn chi phí của một dự án phần mềm là để dành cho bảo trì. Để giảm thiểu các sai sót khi đưa ra thay đổi trên phần mềm, việc chúng ta hiểu được hệ thống vận hành như thế nào là rất quan trọng. Khi hệ thống phần mềm trở nên phức tạp thì nỗ lực phải bỏ ra để hiểu được cách hoạt động của nó sẽ càng lớn, và rủi ro hiểu nhầm lại càng cao. Do đó mã phải thể hiện thật rõ ràng ý định của người viết nên nó. Càng rõ ý thì người khác sẽ càng dễ hiểu. Điều này sẽ giảm tỷ lệ tỳ vết cũng như chi phí bảo trì.

Rõ ý có thể bắt đầu từ những tên được đặt thật tốt. Chúng ta muốn đọc vào một lớp hay một hàm mà không phải ngạc nhiên khi phát hiện ra trách nhiệm thực sự của chúng.

Hoặc có thể làm cho mã rõ ý hơn bằng cách giữ cho các lớp và các hàm nhỏ lại. Nhỏ hơn thì dễ đặt tên, dễ viết, và dễ hiểu.

Sử dụng danh pháp tiêu chuẩn cũng là một hành dụng tốt. Các Mẫu Thiết Kế là một ví dụ điển hình. Bằng cách đưa tên chuẩn của các mẫu, như COMMAND hay VISITOR, vào tên của những lớp triển khai các mẫu thiết kế đó, bạn có thể thông báo một cách ngắn gọn về thiết kế của mình cho các nhà phát triển khác.

Các kiểm thử đơn vị được viết tốt cũng có tính rõ ý. Mục đích chính của các kiểm thử là hoạt động như một tài liệu hệ thống. Ai đó có thể đọc các kiểm thử và hiểu được rất nhanh các chi tiết về một lớp.

Nhưng cách thực hiện quan trọng nhất là để tâm cố gắng. Thông thường chúng ta làm cho mã hoạt động được và sau đó chuyển sang vấn đề tiếp theo mà không suy nghĩ đầy đủ để khiến mã sẵn sàng cho người khác đọc. Hãy nhớ rằng "người khác" đó có thể sẽ là bạn.

Vậy nên hãy cổ vũ tinh thần thợ lành nghề của bạn một chút. Dành một khoảng thời gian cho từng hàm từng lớp một. Chọn tên tốt hơn, chia nhỏ hàm thành các hàm nhỏ hơn. Sự săn sóc là một nguồn lực quý báu.

Số lượng lớp và phương thức tối thiểu

Ngay cả những khái niệm cơ bản như khử lặp, rõ ý và đơn trách nhiệm cũng có thể bị đẩy đi quá xa. Trong nỗ lực giữ cho các lớp và phương thức được nhỏ nhất có thể, chúng ta có thể tạo ra quá nhiều lớp và phương thức nhỏ. Vì vậy, quy tắc này gợi ý chúng ta nên giữ số lượng hàm và lớp ở mức thấp.

Quá nhiều lớp và phương thức đôi khi là kết quả của thói quan liêu. Thử nghĩ đến một tiêu chuẩn yêu cầu tất cả mọi lớp đều phải ở dưới một giao diện, hay buộc các trường và hành vi phải luôn được tách biệt thành lớp dữ liệu và lớp hành vi. Nên chống lại những giáo điều như vậy và sử dụng một cách tiếp cận thực dụng hơn.

Mục tiêu của chúng ta là giữ nhỏ tổng thể hệ thống trong khi vẫn giữ nhỏ được các hàm và các lớp. Tuy nhiên cần lưu ý rằng quy tắc này có ưu tiên thấp nhất trong bốn nguyên tắc của Thiết Kế Đơn Giản. Vì vậy mặc dù giữ cho số lượng các hàm và các lớp ở mức thấp là quan trọng, nhưng việc có thể kiểm thử, không lặp và rõ ý còn quan trọng hơn.

Kết luận

Không có có bộ nguyên tắc nào có thể thay thế được cho kinh nghiệm. Nhưng nhìn từ một khía cạnh khác, các nguyên tắc thực hành được mô tả ở bài viết này là kết tinh kinh nghiệm nhiều thập kỷ mà các tác giả của nó đã trải qua. Việc tuân theo thực hành thiết kế đơn giản có thể cổ vũ và tạo cơ hội cho các nhà phát triển tuân thủ được các nguyên tắc và mẫu thiết kế tốt mà thường phải mất nhiều năm học hỏi.

COMMENT
TRONG CLEAN CODE

Dư Thanh Hoàng

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

Chỉ có code mới thực sự có thể cho bạn biết nó đang có gì và làm gì. Đây là nguồn thông tin thực sự chính xác duy nhất. Do đó, mặc dù comment là đôi khi là cần thiết, nhưng chúng ta sẽ tìm cách để tối thiểu nó trong code của mình.

1. Comments Do Not Make Up for Bad Code (Comment không dùng để trang trí cho code tồi)

Một trong những lý do chính để bạn comment là bad code. Bạn viết xong 1 đoạn code, bạn đọc lại và thấy nó thật khó hiểu và vô tổ chức. Sau đó bạn thêm một vài đoạn comment vào giải thích cho các đoạn code đó và tự nhủ với mình "Đoạn code đã tốt hơn rồi". Không, sẽ tốt hơn là bạn nên xóa nó đi. Một đoạn mã tốt với chỉ với một vài comment sẽ tốt hơn rất nhiều so với sự lộn xộn và phức tạp của một đoạn mã với quá nhiều comment. Thay vì dành thời gian để viết comment cho những mớ hỗn độn của mình thì bạn nên dọn dẹp nó.

2. Explain Yourself in Code: Tự giải thích mình trong code

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

Xem xét đoạn code sau với comment

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

Bạn thấy ổn không? Có comment giải thích đoạn if cho bạn. Vậy tại sao không để nó tự giải thích như thế này chẳng hạn

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

Chỉ mất vài giây để người đọc hiểu đoạn code muốn đề cập đến vấn đề gì thay vì đọc comment.

3. Good Comments

Comment nên hạn chế được đưa vào code, tuy nhiên dưới đây là một số good comment bạn nên đưa vào để bổ sung các thông tin cho các đoạn code của bạn.

3.1. Legal Comments: Comment về pháp lý

Đó là các comment để cho người khác biết ai viết đoạn code đó. Bạn nên comment để cho người khác biết. Các đoạn comment này được sinh mỗi khi bạn tạo file:

```
<?php
/**
 *
 * Created by PhpStorm.
 * User: Du Thanh Hoang ( @hoangdt )
 * Date: 8/27/20
 * Time: 11:11
 */
```

3.2. Các comment chứa thông tin

Đó là các thông tin khá hữu ích, cung cấp các thông tin cơ bản nhất về 1 hàm (chẳng hạn đầu vào, đầu ra):

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

Hoặc đôi khi là format dữ liệu đầu ra để bổ sung cho lập trình viên

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

Tuy nhiên lời khuyên ở đây là tốt nhất bạn vẫn nên để tên hàm nói lên các thông tin này. Nếu không thể hoặc muốn bổ trợ thì hãy dùng comment.

3.3. Giải thích thêm cho mục đích, quyết định

Đôi khi, comment nên giải thích các thông tin hữu ích về việc thực thi và cung cấp ý định đằng sau quyết định, các đoạn code xử lý (trả lời câu hỏi Why).

3.4. Đưa ra cảnh báo hậu quả

Đôi khi rất hữu ích khi cảnh báo những người lập trình khác về những hậu quả nhất định. Ví dụ dưới đây đưa ra các cảnh báo sử dụng hàm nếu bạn không có thời gian để tắt nó 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);
}
```

3.5. TO DO comment

Đó là comment các công việc bạn chưa kịp thực hiện hoặc các chức năng bạn có thể phát triển trong tương lai:

```
//TODO-MdM these are not needed
// We expect this to go away when we do the check-
out model
protected VersionInfo makeVersion() throws Ex-
ception
{
    return null;
}
```

4. Bad comments

Đa số các comment đều thuộc thể loại này. Kể cả đối với các good comment người ta vẫn khuyên bạn không nên đưa nó vào trong code nếu không thật sự cần thiết.

4.1. Comment thừa

Những comment như thế này là những comment không có ý nghĩa, đôi khi có làm gián đoạn hay khó chịu với người đọc. Xét ví dụ sau:

```
i++; // increment i
```

Chắc hẳn ví dụ này hơi ngây thơ. Thôi thử vào Laravel đi, lớp "User" mặc định nhé:

```
class User extends Authenticatable
{
  use Notifiable;

  /**
  * The attributes that are mass assignable.
  *
  * @var array
  */
  protected $fillable = [
    'name', 'email', 'password',
  ];

  /**
  * The attributes that should be hidden for
  arrays.
  *
  * @var array
  */
  protected $hidden = [
    'password', 'remember_token',
  ];
}
```

Những comment như trên quả thật là thừa. Gần như ai cũng biết các biến $fillable, $hidden hay thâm chí là $table, $primaryKey để lưu làm gì rồi (vì nó đã có trong document và quy ước chung), comment như vậy không bố trợ thêm mà gây dài dòng khó chịu cho đoạn code.

4.2. Commented-Out Code

Đó là những comment như thế nào. Đó là những dòng code không được chạy nhưng vẫn được comment kiểu như thế này:

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
  writeEnd();
    this.pngBytes  =  resizeByteArray(this.png-
    Bytes, this.maxPos);
}
else {
    this.pngBytes = null;
}

return this.pngBytes;
```

Những người khác thấy đoạn code đã comment sẽ không có đủ can đảm để xóa nó. Họ cho rằng có lý do để nó nằm ở đó. Họ sẽ đặt ra các câu hỏi, tại sao đoạn code này ở đây? Nó có quan trọng không?

Hãy mạnh tay xóa nó đi, bây giờ chúng ta đã có các hệ thống quản lý mã nguồn rất tốt (git chẳng hạn). Hệ thống sẽ nhớ những đoạn code thay đổi giúp chúng ta và những comment như vậy thật sự không cần thiết nữa.

4.3. Đừng comment khi bạn có thể sử dụng biến để thay thế

Hãy ưu tiên sử dụng biến, hàm hơn là comment. Thay vì comment nhiều như thế này

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

Hãy biến chúng thành các biến:

```
ArrayList moduleDependees = smodule.getDepend-
Subsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

Lần đầu viết đoạn code bạn có thể comment và sau đó viết đoạn code để thực hiện comment đó và xóa đoạn comment đó đi.

4.4. Nonlocal Information

Khi bạn comment gì thì bạn tập trung vào vấn đề đó, hãy chắc chắn nó miêu tả cho đoạn code xuất hiện gần đó. Không được cung cấp thông tin toàn hệ thống hay chung chung:

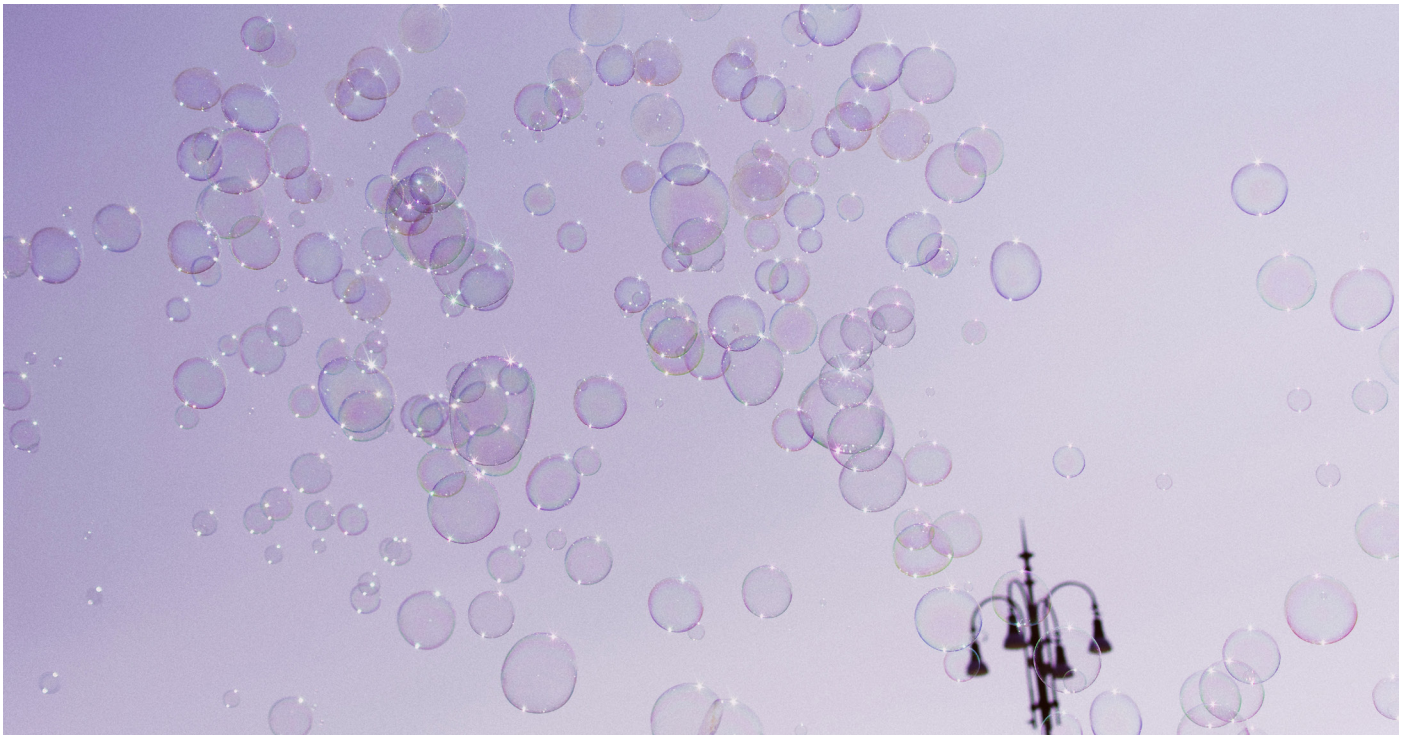
```
/**
* Port on which fitnesse would run. Defaults to
<b>8082</b>.
*
* @param fitnessePort
*/
public void setFitnessePort(int fitnessePort)
{
    this.fitnessePort = fitnessePort;
}
```

Tổng kết

Hạn chế đưa comment vào code của bạn, hãy cố gắng để các tên hàm, tên biến, tên lớp tự nói lên chức năng của nó.

Lý thuyết chỉ đưa ra lời khuyên để code của bạn dễ đọc hơn chứ không bắt buộc bạn phải tuân theo, bạn có thể tùy biến miễn là code của bạn dễ đọc, dễ hiểu để có thể dễ dàng bảo trì và mở rộng.

EXPRESSIVE
CLEAN CODE



Dịch: Dương Nhật Minh

Một người kể chuyện tài ba là người có năng lực ngôn ngữ mạnh mẽ, anh/cô ấy biết nên dùng những từ gì, ngữ pháp gì ở đúng nơi, đúng thời điểm để kể ra một câu chuyện hay.

Tương tự, bạn - lập trình viên cần phải biết những danh từ và động từ (ít nhất là cũng phải ở mức cơ bản) để có thể lập trình ra được một chương trình "hay" cho bất kì ai đọc.

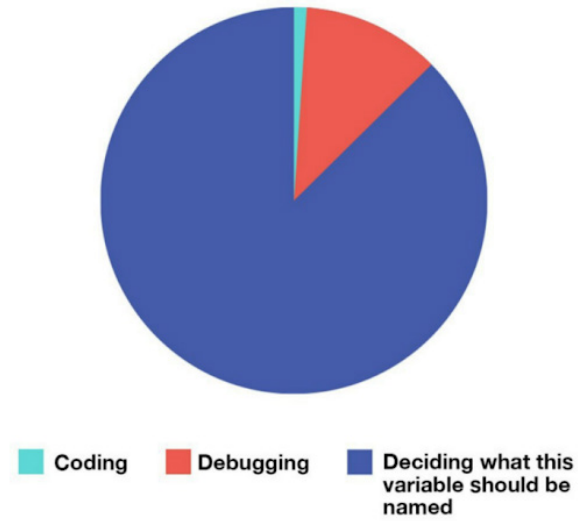
Các danh từ ở đây coi như là các biến và các lớp, và các động từ là các hàm.

Trong bài viết này, chúng ta sẽ học cách tạo và sử dụng chúng một cách chính xác.

Bạn đừng tưởng rằng việc đặt tên các biến, hàm và lớp của mình một cách tường minh là công việc dễ dàng, nó có thể không khó như bộ môn hóa học nhưng nó vẫn khó hơn bạn tưởng nhiều đấy.

Được rồi, hãy bắt đầu.

Clean Variable Names (Tên biến "sạch")



Đừng nên tốn quá nhiều thời gian để tìm ra một cái tên phù hợp - chỉ cần hình dung ra mã của bạn với các tên biến khác nhau và xem tên nào phù hợp nhất với mã đó.

Một quy tắc đơn giản mà bạn luôn có thể tuân thủ là: Tên biến phải thể hiện được câu trả lời cho 3 câu hỏi sau.

- Tại sao lại có dòng code này?
- Nó làm gì?
- Nó được sử dụng như thế nào?

Dưới đây là một số bổ sung:

- Việc phân biệt hai biến không nên gây mất quá nhiều thời gian. Ví dụ, hai biến này cần mất quá nhiều thời gian để có thể phân biệt.

```
var XYZControllerForEfficientHandlingOfStrings
var XYZControllerForEfficientStorageOfStrings
```

- Cẩn thận khi sử dụng các ký tự như 1 (một) và l (L viết thường) hoặc O (O viết hoa) và 0 (không).
- Không đặt tên chỉ để thỏa mãn trình biên dịch. Khi trình biên dịch phát sinh lỗi do các tên trùng lặp trong cùng một phạm vi, không được thay đổi tên của biến thành một cái gì đó như biến2, biến3 hay một số tên ngẫu nhiên khác. Nếu các biến của bạn khác nhau, tên của chúng phải nêu rõ lý do tồn tại của chúng.
- Sử dụng những tên dễ phát âm để tiện hơn khi thảo luận về mã của bạn với người khác.
- Sử dụng tên có thể search ra được. IDE có thể giúp mình tìm kiếm mã của mình và thậm chí có thể đưa ra những đề xuất. Nhưng bạn chỉ có thể làm được điều này khi bạn đặt tên cho biến / hàm của mình. Điều này sẽ giúp bạn xem lại những biến / hàm từ bất kỳ phần nào trong dự án của mình mà không cần phải quay lại để kiểm tra tên bạn đã đặt cho nó.
- Không đặt những cái tên ngộ nghĩnh hoặc thử chơi chữ. Đặt cái tên theo đúng chức năng mà bán gán cho nó.

Viết hàm sạch

Bạn sẽ biết lúc mà bạn đang làm việc với mã sạch khi mỗi hàm hoạt động khá chính xác những gì mà bạn mong đợi nó thực hiện.

- Các hàm chỉ nên thực hiện tốt một nhiệm vụ.
- Các hàm chỉ nên thay đổi với một lý do. Và hàm phải tuân theo nguyên tắc trách nhiệm duy nhất (single responsibility principle)
- Các chức năng không nên bị không thay đổi khi có yêu cầu mới. Nó phải tuân theo nguyên tắc đóng mở (open-closed principle)

Một hàm bao gồm phần thân và một danh sách các tham số, cả hai đều rất quan trọng trong việc viết các hàm sạch.

Chúng ta hãy cùng xem xét một số điểm:

1. Phần thân của hàm:

- Một hàm cần phải thực hiện một chức năng gì đó hoặc trả về một cái gì đó - không phải cả hai cùng lúc. Hoặc nó sẽ sửa đổi trạng thái của đối tượng hoặc nó sẽ trả về một số thông tin về đối tượng. Việc thực hiện cả 2 việc sẽ dẫn đến nhầm lẫn trong tương lai.
- Các hàm cần phải thật gọn, nói cách khác, nó không được vượt qua 20 dòng mã (nhưng hãy cố gắng viết ngắn hơn nữa).
- Mức lùi dòng của một hàm không được lớn hơn một hoặc hai so với dòng mẹ. Điều này làm mã bạn dễ hiểu hơn.

Đây là một ví dụ:

```
$(
  function() {
    Function saveState() {
      if (this.options.saveState == true) {
        if (this.state == "normal") {
          $.ajax( {
            url : "bla"
          });
        } else {
          $
            .ajax( {
              url : "eclipse javascript format-
              ter is behaving weird with long
              texts"
            });
        }
      }
    }
  })(jQuery);
```

- Các khối bên trong câu lệnh if, câu lệnh else, câu lệnh while, v.v. nên chỉ dài một dòng. Nếu đó là một câu lệnh điều kiện dài, hãy cải tiến lại nó dưới dạng một hàm trả về giá trị true hoặc false.
- Một cái tên mô tả dài dễ hiểu sẽ tốt hơn một tên ngắn khó hiểu.

2. Đối số hàm

- Con số lý tưởng: Con số lý tưởng mà một hàm nên có là số không. Sau đó, khuyến khích không nhiều hơn một, hai, ba thì nên tránh nếu có thể. Nhiều hơn ba đối số chỉ được phép cho những trường hợp cực kì đặc biệt.
- Tại sao ư? Nhiều đối số gây rất nhiều khó khăn khi chạy các test case. Hãy tưởng tượng sự khó khăn khi viết tất cả các trường hợp test để đảm bảo tất cả các tổ hợp đối số khác nhau hoạt động chính xác.
- Vấn đề: Nếu không có đối số, thì ok. Nếu có một đối số, nó không quá khó. Với hai đối số, vấn đề trở nên khó khăn hơn một chút. Với nhiều hơn hai đối số, việc test mọi tổ hợp của các giá trị thích hợp trở nên khó khăn. Bạn có thể tránh điều đó bằng cách viết các hàm với số lượng đối số tối thiểu.
- Không pass qua được flag arguments: Pass một flag argument (true/false) là rất khó. Bởi vì nó làm cho hàm của bạn vi phạm SRP - vì hàm của bạn hoạt động theo một kiểu khi đối số là true và theo kiểu khác khi đối số là false.
- Bạn nên làm gì? Dùng đối tượng đối số: Việc đóng gói nhiều đối số bên trong một lớp có vẻ là một cách làm "tà đạo", nhưng thực tế thì không phải vậy. Khi có nhiều đối số liên quan đến nhau mà bạn cần gửi đến một hàm, chúng có thể liên kết đến nhau theo một cách nào đó. Việc để gói gọn các trường đó vào một lớp và đặt tên có ý nghĩa cho lớp đó thực chất lại rất tốt.

Nhiều ngôn ngữ cung cấp cho bạn nền tảng để viết các hàm nội tuyến. Ví dụ: bạn được yêu cầu viết một hàm nhận một báo cáo làm đầu vào và thêm phần footer vào báo cáo đó.

- Hướng giải quyết 1: Tạo một hàm lấy báo cáo làm đầu vào và thêm chân trang vào báo cáo. Hàm này nhận một đối số. Tuy nhiên, bạn vẫn có thể cải tiến nó

```
public void appendFooter(StringBuffer report)
{...}
```

- Hướng giải quyết 2: Tạo một hàm nội tuyến, không cần đối số, hoạt động trực tiếp với báo cáo.

```
report.appendFooter()
```

Cách để xử lý lỗi / ngoại lệ?

Các hàm xử lý lỗi chỉ được phép có chức năng là xử lý lỗi và không được làm bất cứ điều gì khác.

Sử dụng ngoại lệ hơn là trả về Codes. Khi bạn trả về mã lỗi, người gọi phải kiểm tra trực tiếp sau khi gọi. Điều này có nghĩa là nhiều mã soạn sẵn hơn.

Ở ví dụ dưới, chúng tôi đang làm networking để cập nhật giao diện người dùng của mình với danh sách các mặt hàng thực phẩm.

Lớp Networking:

```
// Inside some function
// Getting list from somewhere
val listOfFoodItems = getFoodItemsList()
// If list is null, means something went wrong
if (listOfFoodItems.isNull()) {
    callback(FoodOrder.Error)
}
// If all goes well, we send back list of food
items
else {
   callback(FoodOrder.FoodMenu(listOfFoodItems))
```

Lớp UI

```
// Getting list from somewhere
var status = getReturningStatus()
if (status == ERROR) {
    // Handle errors
} else if (status == FoodItemList) {
    // Do something with food list
}
```

Lớp networking sau khi cải tiến:

```
val listOfFoodItems = getFoodItemsList()
if (listOfFoodItems.isNull()) {
    throw some custom Exception
}
else {
   callback(FoodOrder.FoodMenu(listOfFoodItems))
}
```

Lớp UI sau khi cải tiến::

```
try {
  var listOfFoodItems = getAllFoodItems()
} catch (your exception type) {
  // Act accordingly
}
```

Ngay cả khi các trường hợp sử dụng lỗi của bạn tăng lên, bạn chỉ phải ném đi ngoại lệ (throw exception) khỏi lớp networking - và chỉ cần thêm nhiều catch block hơn trong lớp UI.

Nhân tiện, mã này vẫn tuân theo nguyên tắc đóng mở (open-closed principle)

Mẹo:

Nếu bạn đang thực hiện nhiều việc trong các khối try/catch, hãy bỏ ra các chức năng không phù hợp để làm cho mã của bạn trông sạch sẽ và dễ đọc hơn. Đừng để một khối try-catch lồng nhau.

Phần kết luận

Các lập trình viên bậc thầy nghĩ về hệ thống như những câu chuyện được kể hơn là những chương trình được viết.

Họ chọn các cấu trúc thể hiện được câu chuyện mà họ đang viết bằng mã. Chức năng là động từ của một ngôn ngữ và các lớp là danh từ. Nghệ thuật lập trình là về cách sử dụng các công cụ mà ngôn ngữ lập trình đó cung cấp để kể câu chuyện hay.

Nguồn: https://medium.com/better-programming/clean-code-give-meaning-to-your-code-to-exist-f966b3f00848

LỢI ÍCH CỦA CLEAN CODE

Dịch: Nguyễn Minh Quân

Tại sao mọi người thỉnh thoảng hay viết code kiểu như thế này?

```
var = float(str(alist[::-1][0]).split()
[1:4])/3+float(alist[4:])
```

Câu trả lời đó là để tiết kiệm thời gian cho việc tính toán. Khi mà đoạn code có thể được thay thế thành đoạn code chỉ dài nhiều hơn 3 dòng...

```
var = alist[::-1][0]
var = str(var).split()[1:4]
var = float(var)/3
var += float(alist[4:]
```

...và những người tính toán chi phí cho việc tiết kiệm thời gian đều lắc đầu với phương án dưới và chọn phương án trên.

Họ cho rằng việc gán đi gán lại biến nhiều lần chiếm nhiều không gian tính toán nhiều đến mức số lần lặp lại đó trong đoạn code của họ có thể tạo ra sự khác biệt đáng kể.

Trong bài viết này, tôi sẽ khám phá chi phí tính toán thực sự của việc viết mã sạch với việc gán nhiều biến với nhiều các thí nghiệm kiểm thử.

Đầu tiên - Những lợi ích của việc gán nhiều biến

Đặc biệt là trong ngôn ngữ lập trình như Python, ở đó chúng ta có ít nhất mười cách để viết bất cứ thứ gì, và các developer thường sẽ viết nhiều các thao tác cần thực hiện trên một dòng code.

Việc gán nhiều biến cho phép người đọc tiếp nhận các chức năng được áp dụng trong các đoạn code ngắn hơn. Ngoài ra, nó giúp cho bạn dễ dàng chọn thông qua các lớp của dấu ngoặc đơn hiện tại khi mà có nhiều hơn 3 hàm của Python đang được sử dụng:

```
list(str(int(x)+1)+'1') #Phạm vi code này rất khó
để đọc.
```

Hơn thế nữa, nó rất khó để theo dõi tình trạng hiện tại của 1 biến khi tất cả các thao tác đều được viết trên một dòng. Nó giống như một giảng viên dạy môn toán học - giảng viên bắt đầu với môn số học trước, sau đó mới đến giải tích sau, thay vì dạy cả 2 môn học đó cùng 1 lúc.

Việc gán nhiều biến có nghĩa rằng người đọc có thể theo dõi những gì đang xảy ra với biến và biến đó đang ở trạng thái như thế nào và dễ dàng hơn rất nhiều so với việc so sánh 4 dòng code với 1 dòng code.

Trong nhiều trường hợp, việc gán nhiều biến giúp chúng ta tiết kiệm thời gian tính toán.

Ví dụ:

```
a = (b+5)*5 + (b+5)/4
Và có thể viết theo cách khác là
c = b+5
a = c*5 + c/4
```

Bằng cách gán `(b+5)` vào một biến, nó sẽ chỉ được tính toán một lần thay vì nhiều lần.

Thử nghiệm trên Python Lists và các Built-In Function

Đây là một thử nghiệm cho việc tính toán thời gian mà chúng ta sử dụng sẵn một số hàm được tích hợp phổ biến trong Python như `str()`, truy cập các phần từ của list qua chỉ số và các phép tính toán học:

```
float(int(str(alist[::-1][0]).split()[::-1][0])/
int(alist[:4][0]))/3
```

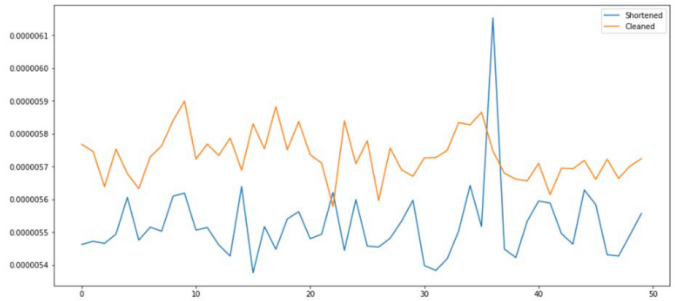
Có thể viết theo cách khác như sau:

```
var = str(alist[::-1][0]).split()
var = int(var[::-1][0])
var /= int(alist[:4][0])
var = float(var)/3

"alist" được tạo ra như sau:

alist = [random.randint(1,10) for j in range(100)]
```

Thời gian được sử dụng để tạo ra danh sách không được bao gồm trong thời gian tính toán. Chỉ có duy nhất hoạt động được tính toán thời gian là các dòng code mà được chạy để kiểm tra các hoạt động.

Hoạt động này được chạy 5,000,000 lần với việc tạo ra "alist" khác nhau trong mỗi lần chạy.

Trung bình được thực hiện 100,000 lần mỗi lần và được vẽ thành biểu đồ, trong đó "cleaned" (Code đã sạch) biểu thị cho phiên bản 4 dòng code và "shortened" (đoạn code ngắn gọn) biểu thị cho 1 dòng code.



Phiên bản code ngắn gọn rõ ràng được thực hiện gần như toàn bộ trong thời gian nhanh hơn, nhưng ở quy mô nhỏ như vậy, nó sẽ không có lợi chút nào.

Thời gian trung bình để chạy phiên bản code ngắn gọn là 0.000005521 và thời gian trung bình để chạy phiên bản code sạch là 0.000005733. Sự khác biệt đó là 0.000005521.

Điều đó có nghĩa rằng để thấy được sự khác biệt 1 phút trong thời gian hoạt động thì quá trình này sẽ cần được lặp lại ít nhất là 10,867,596 lần. Và để thấy được sự khác biệt 1 tiếng thì chương trình cần được lặp lại ít nhất 652,055,786 lần.

Thử nghiệm trên Pandas DataFrames

Trong thử nghiệm thứ 2 sẽ thực hiện các hoạt động không chỉ trong List của Python mà còn trên Pandas DataFrames của Python. Đây là loại dữ liệu cần thiết của machine learning và data science trong Python và nó giống như một bảng tính trong Excel.

Biểu thức thử nghiệm đó là:

```
df.loc[1:100][df.loc[1:100] > 5]['b'].dropna().
std() — df.loc[1:100][df.loc[1:100] < 5]['a'].
dropna().mean()
```

Trong đó `df` là DataFrames. Một số các tài liệu:

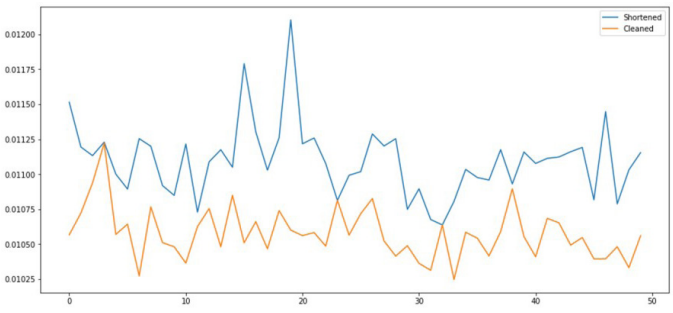
- `data.loc[x:y]` chọn ra các hàng của dữ liệu có chỉ số nằm giữa bao gồm cả x và y
- `data[data['column'] > 5]` chọn ra các hàng của dữ liệu có cột, trong trường hợp này tên của nó là `column`, lớn hơn 5 (hoặc một vài điều kiện khác) và trả về nan cho các hàng không phù hợp với điều kiện.
- `data.dropna()` xóa bất kỳ hàng nào có giá trị là nan
- `column.std()` lấy ra độ lệch tiêu chuẩn của 1 cột/dãy số.
- `column.mean()` lấy giá trị trung bình của cột/dãy số.

Biểu thức thử nghiệm trên có thể chia thành 6 dòng với 3 biến:

```
result0 = df.loc[1:100]
result = result0[result0 > 5]['b']
result = result.dropna().std()
result2 = result0[result0 < 5]['a']
result2 = result2.dropna().mean()
result = result — result2
```

Đoạn code thử nghiệm trên được thực hiện tạo ra ngẫu nhiên DataFrame với 2 cột là a với b và 200 hàng. Và tất cả các giá trị sẽ được sinh ra ngẫu nhiên từ 1 tới 10.

Đoạn code thử nghiệm trong cả phiên bản code sạch và code được rút gọn đều được chạy 50,000 lần, với trung bình thực hiện 1000 lần mỗi lần. Mỗi lần lặp lại đều được thực hiện trên một DataFrames mới, được tạo ra một cách ngẫu nhiên.



Kết luận

Bài học rút ra đó là viết code sạch (clean). Bạn đừng sợ việc tăng thời gian tính toán vì việc gán nhiều biến. Việc sử dụng nhiều biến không chỉ làm tăng độ rõ ràng của code mà nó còn có thể trong một vài trường hợp (như đã được chứng minh qua ví dụ sử dụng nó trong Pandas DataFrame) cải thiện hiệu suất tính toán.

Cảm ơn các bạn đã đọc bài!

Nguồn: https://towardsdatascience.com/the-computational-cost-of-writing-clean-code-fec649f330b9

ĐỊNH DẠNG MÃ NGUỒN

Mai Công Sơn

Cần định dạng mã nguồn để có thể dễ đọc hơn và là một phần quan trọng để giữ mã sạch.

Mã nguồn không được định dạng theo đúng cách gây ra khó đọc, khó hiểu

```
getData = (): Promise<Object> =>{ return new
Promise((resole, reject) =>{
let url = “https://jsonplaceholder.typicode.com/
pots”;
this.http.get(url).subscribe(res =>
{resole(res);} err =>{reject(err);})})}
```

Trong bài viết này, chúng ta sẽ cùng tìm hiểu về cách định dạng mã JavaScript để chúng có thể được đọc dễ dàng.

Tại sao chúng ta cần định dạng mã nguồn

Định dạng mã nguồn rất quan trọng vì mã của chúng ta có thể được đọc bởi những người khác.

Một đoạn mã nguồn mà là một mớ hỗn độn gây khó khăn cho việc đọc hiểu mã bởi các thành viên khác trong nhóm. Những thành viên khác không thể làm việc nếu không hiểu đoạn mã chúng ta viết gì.

Ngoài ra, chúng ta và các thành viên trong nhóm phải thường xuyên làm sạch mã. Rất khó để chỉnh sửa nếu định dạng mã xấu.

Định dạng theo chiều dọc

Số dòng mã trong một file phải dưới 500 dòng.

Việc tách nhỏ ra các file sẽ dễ đọc hơn để các file với số lượng dòng code lớn, Các file với số lượng dòng code lớn mất nhiều thời gian để đọc.

Mã nguồn giống như một bài báo. Chúng ta càng đi sâu xuống, bài viết càng chi tiết hơn. Điều này cũng tương tự đối với mã. Chúng ta có phần giới thiệu với các khai báo về biến và hàm và sau đó khi chúng ta xuống thấp hơn, chúng ta sẽ có thêm chi tiết triển khai của mã.

Dòng trống

Các dòng trống rất quan trọng giữa các quyền khác nhau. Chúng đặc biệt quan trọng giữa các hàm và định nghĩa lớp. Không có chúng, mọi thứ trở nên khó đọc và khiên người đọc bị rối mắt cảm giác lười dò tìm khi nhìn vào những trang code dài.

Ví dụ: sau mô tả sự khác biệt khi sử dụng dòng trống và không sử dụng:

```
class Foo {metho1(){}metho2(){// do something }}
class Bar {metho1(){}metho2(){// do something }}
```

Tuy nhiên, cách sau dễ đọc hơn nhiều:

```
class Foo {
  method1() {
    // do something
  }
  method2() {
    // do something
  }
}
class Bar {
  method1() {
    // do something
  }
  method2() {
    // do something
  }
}
```

Vì vậy, chúng ta nên đặt một số dòng trống trong mã nguồn của mình.

Thật khó để đọc vào một phần mã khi tất cả chúng đều được nhóm lại với nhau.

Mật độ

Như chúng ta có thể thấy ở trên, mã không được quá dày đặc theo chiều dọc. Luôn để lại khoảng trống giữa các nhóm mã.

Chúng ta có thể nhóm các khai báo biến với nhau mà không có dòng trống và các lớp trong nhóm riêng của chúng như sau:

```
let x = 1;
let y = 2;

class Foo {
  method1() {
    //do something
  }
  method2() {
    //do something
  }
}

class Bar {
  method1() {
    //do something
  }
  method2() {
    //do something
  }
}
```

Khoảng cách

Sẽ thật khó khăn khi phải chuyển từ chức năng này sang chức năng tiếp theo mà phải cuộn trang liên tục để đọc mã. Nó gây khó khăn cho việc đọc hiểu mã nguồn.

Ngoài ra chúng ta phải cuộn qua rất nhiều đoạn mã để tìm định nghĩa của một biến nào đó

Để hạn chế điều này các khái niệm liên quan đến nhau phải được định nghĩa ở gần nhau

Khai báo biến

Khai báo biến phải gần nơi chúng được sử dụng để có thể tìm thấy một cách nhanh chóng mà không cần cuộn lên trên để tìm hoặc chuyển qua các file khác nhau.

Các biến điều khiển vòng lặp phải được khai báo trong câu lệnh vòng lặp để người đọc có thể biết ngay rằng nó được sử dụng trong vòng lặp.

Biến thuộc tính

Các biến thuộc tính nên được khai báo trên đầu để có thể dễ dàng tìm thấy.

Chúng ta có thể đưa chúng vào hàm khởi tạo để có thể dễ dàng sử dụng hoặc thay đổi

Ví dụ lớp sau:

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  getPoint() {}
  setPoint() {}
}
```

Rõ ràng hơn như sau:

```
class Point {
  getPoint() {}
  setPoint() {}
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}
```

Hàm khởi tạo có mã khởi tạo, vì vậy chúng ta nên đặt nó lên trên cùng.

Các hàm phụ thuộc

Các hàm phụ thuộc vào từng hàm phải gần nhau để chúng ta không cần phải cuộn qua các file để tìm định nghĩa ban đầu của các hàm được gọi.

Ví dụ: nếu chúng ta có một chuỗi các lệnh gọi hàm như sau, chúng ta nên viết chúng gần nhau

```
class Foo {
  foo() {
    this.bar();
  }
```

```
  bar() {
    this.baz();
  }
  baz() {}
}
```

Mã liên quan

Các mã có liên quan phải gần nhau để chúng ta không cần phải cuộn trang để tìm các khái niệm liên quan. Chúng có thể không phụ thuộc trực tiếp, nhưng chúng liên quan chặt chẽ đến mức chúng ta cần tìm chúng.

Ví dụ: lớp sau có các phương thức liên quan chặt chẽ với nhau:

```
class Assert {
  assertTrue() {}
  assertFalse() {}
  assertNotUndefined() {}
}
```

Cả 3 đều khẳng định rằng một số điều kiện được đáp ứng, vì vậy chúng là các khái niệm có liên quan.

Đặt hàng dọc

Hàm được gọi phải nằm bên dưới một hàm thực hiện lệnh gọi. Điều này tạo ra một luồng tốt từ mức cao xuống mức thấp.

Các khái niệm cấp cao rất quan trọng, vì vậy chúng phải ở trên cùng.

Ví dụ:

```
class Foo {
  foo() {
    this.bar();
  }
  bar() {
    this.baz();
  }
  baz() {}
}
```

Phương thức `foo` gọi phương thức `bar` để làm điều gì đó và phương thức bar gọi phương thức `baz` để làm điều khác.Phương thức `foo` là cấp cao nhất.

Kết luận

Định dạng mã nguồn rất là quan trọng đặc biệt khi chúng ta thực hiện dự án lớn. Nếu không định dạng mã nguồn sẽ là một mớ hỗn độn khó đọc do đó khó bảo trì. Chúng ta phải chủ động định dạng mã nguồn có định dạng lộn xộn.

Để làm được điều này, trước tiên chúng ta phải xem xét định dạng dọc của mã nguồn. Các dòng trống cũng quan trọng giữa các nhóm mã như hàm và khai báo biến.

Ngoài ra, các khái niệm liên quan nên được nhóm lại gần nhau để giảm thời gian tìm kiếm các thực thể này.

Biến thuộc tính phải ở trên cùng để dễ tìm. Các khai báo biến khác cũng phải gần nhau, nhưng chúng phải ở gần nơi chúng sẽ được sử dụng.

Cuối cùng, mã cấp cao phải nằm trên mã cấp thấp hơn.

MÃ SẠCH, CHẤT LƯỢNG CAO: CÁCH TRỞ THÀNH MỘT LẬP TRÌNH VIÊN GIỎI HƠN

Dịch: Phan Văn Luân

Khi nhắc đến mã sạch, hay tái cấu trúc mã nguồn bạn có từng suy nghĩ:

> ***"Mã của tôi đang hoạt động tốt, trang web tôi xây dựng trông rất tuyệt và khách hàng của tôi rất vui. Vậy tại sao tôi vẫn quan tâm đến việc viết mã sạch?"***

Nếu điều này giống suy nghĩ của bạn ngay lúc này, thì hãy đọc tiếp.

Cách đây ít lâu, tôi đang thảo luận với một trong những người bạn của mình, Kabir. Kabir là một lập trình viên có kinh nghiệm. Anh ấy đang làm việc trong một dự án phức tạp, và anh ấy đang thảo luận một vấn đề với tôi. Khi tôi yêu cầu xem mã nguồn cho vấn đề đó, anh ấy nói, có vẻ tự hào, "Tôi đã xây dựng dự án này nên chúng tôi là những người duy nhất có thể hiểu mã nguồn của nó."

Tôi đã khá kinh hoàng. Tôi hỏi anh ta có phải anh ta cố tình viết mã bẩn không.

"Khách hàng đã không cho tôi đủ thời gian," bạn tôi nói với tôi. "Họ luôn vội vàng và thúc giục việc bàn giao sản phẩm, vì vậy tôi không có thời gian để nghĩ về việc viết mã sạch".

Đây hầu như luôn là lý do tôi nghe thấy khi hỏi về mã bẩn. Một số lập trình viên viết mã bẩn bởi vì họ dự định phát hành phiên bản đầu tiên và sau đó làm việc để làm cho nó sạch sẽ. Nhưng thường, điều đó sẽ không bao giờ xảy ra; không có khách hàng nào cho bạn thời gian để làm sạch mã. Khi phiên bản đầu tiên được phát hành, họ sẽ đẩy bạn lên phiên bản thứ hai. Vì vậy, hãy tạo thói quen viết mã sạch nhất có thể từ dòng mã đầu tiên.

Tôi luôn biết rằng việc sử dụng các nguyên tắc mã sạch mang lại nhiều lợi ích và bài viết này sẽ cho bạn biết lý do tại sao.

Công việc của người quản lý dự án, trưởng phòng kinh doanh hoặc khách hàng là hoàn thành dự án trong thời gian tối thiểu để họ có thể kiểm soát chi phí của dự án. Nhưng sản xuất chất lượng, mã sạch là nhiệm vụ của bạn với tư cách là lập trình viên.

Viết mã sạch không phải là một nhiệm vụ lớn hoặc tốn thời gian, nhưng biến nó thành thói quen của bạn và cam kết thực hiện nó, sẽ giúp bạn thăng tiến trong sự nghiệp và cải thiện khả năng quản lý thời gian của chính mình.

Mã sạch luôn trông giống như nó được viết bởi một người có tâm. Giống như Martin Fowler – một chuyên gia hàng đầu về phát triển phần mềm từng phát biểu:

"Bất kỳ kẻ ngốc nào cũng có thể viết mã mà máy tính có thể hiểu được. Các lập trình viên giỏi viết mã mà con người có thể hiểu được."

Có thể bạn đã đọc đến đây vì hai lý do: Thứ nhất, bạn là một lập trình viên. Thứ hai, bạn muốn trở thành một lập trình viên giỏi hơn. Tốt. Chúng tôi cần những lập trình viên giỏi hơn.

Hãy tiếp tục theo dõi để tìm hiểu tại sao mã sạch lại quan trọng và bạn sẽ trở thành một lập trình viên giỏi hơn

Tại sao chúng ta nên cố gắng làm cho mã sạch?

Mã sạch có thể đọc được và dễ hiểu bởi mọi người, cho dù người đọc là tác giả của nó hay một lập trình viên mới.

Viết mã sạch là một tư duy cần thiết. Cần thực hành để viết mã có cấu trúc và rõ ràng, và bạn sẽ học cách làm điều đó theo thời gian. Nhưng bạn cần bắt đầu với tư duy viết theo cách này. Và bạn sẽ quen với việc xem xét và sửa đổi mã của mình để mã sạch nhất có thể.

Không ai là hoàn hảo, và bạn cũng vậy. Bạn sẽ luôn tìm thấy một số cơ hội để cải thiện hoặc tái cấu trúc lại mã khi bạn quay lại để xem lại mã của mình sau vài ngày hoặc vài tuần.

Vì vậy, hãy bắt đầu viết mã rõ ràng nhất có thể từ dòng mã đầu tiên để sau này bạn có thể làm việc nhiều hơn về cải thiện hiệu suất và logic.

Lợi ích của mã sạch

"Tại sao tôi nên quan tâm đến việc viết mã sạch?" bạn vẫn có thể tự hỏi mình.

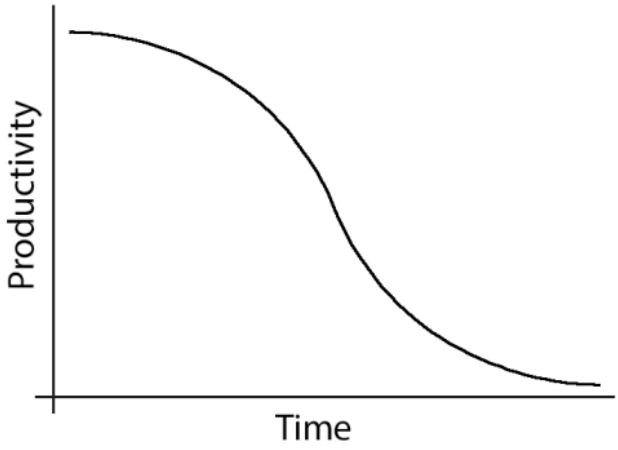
Có nhiều lý do để có được tư duy mã sạch mà tôi đã mô tả ở trên. Một số lý do quan trọng nhất là:

1. Sử dụng thời gian của bạn tốt hơn

Người hưởng lợi đầu tiên của mã sạch là lập trình viên

Nếu bạn đang thực hiện một dự án trong nhiều tháng, rất dễ quên những điều bạn đã làm trong mã nguồn cũ, đặc biệt là khi khách hàng của bạn quay lại với các thay đổi. Các dòng mã rõ ràng giúp bạn thực hiện các thay đổi dễ dàng hơn.

Năng suất lập trình có mối quan hệ trái chiều với thời gian.



2. Tham gia dễ dàng hơn cho các thành viên mới

Sử dụng các nguyên tắc mã sạch sẽ giúp các lập trình viên mới dễ tiếp cận với mã nguồn đang phát triển hơn. Không cần tài liệu để hiểu mã nguồn; lập trình viên mới có thể trực tiếp tham gia vào dự án. Điều này cũng tiết kiệm thời gian đào tạo lập trình viên mới cũng như thời gian để lập trình viên mới điều chỉnh theo dự án.

3. Debug dễ dàng hơn

Cho dù bạn viết mã bẩn hay mã sạch, bug là không thể tránh khỏi. Nhưng mã sạch sẽ giúp bạn gỡ lỗi nhanh hơn, bất kể bạn có bao nhiêu kinh nghiệm hoặc chuyên môn. Và không có gì lạ khi đồng nghiệp hoặc người quản lý của bạn giúp bạn giải quyết vấn đề. Nếu bạn đã viết mã sạch, không vấn đề gì: Họ có thể nhảy vào và giúp bạn một cách dễ dàng hơn. Nhưng nếu người quản lý của bạn phải xử lý mã bẩn của bạn, thì bạn có thể sẽ giống như Kabir, bạn của tôi.

4. Bảo trì hiệu quả hơn

> ***"Tất nhiên mã xấu có thể được viết lại. Nhưng nó rất tốn kém. "***
>
> **— Robert C. Martin**

Bảo trì không đề cập đến việc sửa lỗi. Khi phát triển bất kỳ dự án nào, nó sẽ cần các tính năng mới hoặc các thay đổi đối với các tính năng hiện có. Khi đó mã sạch sẽ giúp chúng ta dễ dàng bảo trì, hay nâng cấp tính năng mới hơn.

Ba điểm đầu tiên giải thích cách mã sạch có thể tiết kiệm thời gian của lập trình viên. Và, tiết kiệm một ít thời gian mỗi ngày sẽ có tác động kép đến thời gian hoàn thành và chi phí của phần mềm. Điều đó tốt cho công ty của bạn.

Vậy làm thế nào để chúng ta có thể cải thiện chất lượng mã của mình? Hãy cùng theo dõi tiếp nhé.

Cách viết mã sạch

> ***"Bạn nên đặt tên cho một biến bằng cách sử dụng cùng một cách mà bạn đặt tên cho đứa con đầu lòng."***
>
> **— Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship**

Một lập trình viên là một tác giả, nhưng họ có thể mắc sai lầm trong việc xác định đối tượng. Đối tượng của một lập trình viên là các lập trình viên khác, không phải máy tính. Nếu máy tính là đối tượng, thì bạn có thể viết mã bằng ngôn ngữ máy.

Vì vậy, để dễ hiểu, bạn nên sử dụng danh pháp có ý nghĩa cho các biến, hàm và lớp. Và làm cho nó dễ đọc hơn bằng cách sử dụng lùi đầu dòng, phương pháp rút gọn và câu lệnh ngắn, nếu thích hợp:

- Sử dụng tên dễ phát âm cho các biến và phương thức. Không sử dụng chữ viết tắt trong tên biến và phương thức. Sử dụng tên biến ở dạng đầy đủ để có thể dễ dàng phát âm và mọi người có thể hiểu được.

Dirty code examples	Clean code examples
public $notiSms; public $addCmt;	public $notifySms public $addComment;

foreach ($people as $x) {
echo $x->name;
}
	foreach ($people as $person)
	{
	echo $person->name;
	}

$user->createUser(); createUser method does not make sense as it is written in user class.	$user->create(); Remove redundancy

- Sử dụng tên để thể hiện đúng mục đích. Mục đích của biến phải dễ hiểu đối với người đọc tên của biến. Viết tên như bạn sẽ nói.

Dirty code examples	Clean code examples
protected $d; // elapsed time in days	protected $elapsedTimeInDays; protected $daysSinceCreation; protected $daysSinceModification; protected $fileAgeInDays;

if ('paid' === $application->status) {
//process paid application
}
	if ($application->isPaid()) {
	//process paid application
	}

- Đừng đổi mới; đơn giản. Cho thấy sự đổi mới trong logic, không phải trong việc đặt tên biến hoặc phương thức. Có một cái tên đơn giản khiến mọi người dễ hiểu.

Dirty code example	Clean code example
$order->letItGo();	$order->delete();

- Hãy kiên định. Sử dụng một từ cho các chức năng tương tự. Không sử dụng "get" trong một lớp và "fetch" trong lớp khác.
- Đừng ngần ngại sử dụng các thuật ngữ kỹ thuật trong tên. Hãy tiếp tục, sử dụng thuật ngữ kỹ thuật. Đồng nghiệp của bạn sẽ hiểu nó. Ví dụ: "jobQueue" tốt hơn "job".
- Sử dụng một động từ làm từ đầu tiên trong phương thức và sử dụng một danh từ chỉ lớp. Sử dụng camelCase cho tên biến và hàm. Tên lớp bắt đầu bằng từ viết hoa.

Dirty code examples	Clean code examples
public function priceIncrement()	public function increasePrice()
Public $lengthValidateSubDomain	Public $validateLengthOfSubdomain;
class calculationIncentive	class Incentive

- Sử dụng các quy ước đặt tên nhất quán. Luôn sử dụng chữ hoa và phân tách các từ bằng dấu gạch dưới.

Dirty code example	Clean code example
define('APIKEY', '123456');	define('API_KEY', '123456');

- Làm cho các hàm rõ ràng. Giữ một hàm càng ngắn càng tốt. Độ dài lý tưởng của một function là tối đa 15 dòng. Đôi khi nó có thể kéo dài hơn, nhưng về mặt khái niệm thì mã phải sạch để hiểu.
- Tham số của hàm nên nhỏ hơn hoặc bằng ba. (Nếu các tham số lớn hơn ba, thì bạn phải suy nghĩ để cấu trúc lại hàm thành một lớp.)

- Một lớp nên làm một việc. Nếu nó dành cho người dùng, thì tất cả các phương pháp phải được viết hoàn toàn cho trải nghiệm người dùng.
- Hạn chế comment vô tội vạ. Nếu bạn phải thêm comment để giải thích mã của mình, điều đó có nghĩa là bạn cần phải cấu trúc lại mã của mình. Chỉ comment nếu điều đó là bắt buộc về mặt pháp lý hoặc nếu bạn cần ghi chú về tương lai hoặc lịch sử của chương trình.

Dirty code example	Clean code example
// Check to see if the employee is eligible for full benefits if ($employee->flags && self::HOURLY_FLAG && $employee->age > 65)	if ($employee->isEligibleForFullBenefits())

- Sử dụng Git đánh version cho ứng dụng. Đôi khi, các tính năng thay đổi và các phương thức cần được viết lại. Thông thường, chúng tôi nhận xét mã cũ vì sợ rằng khách hàng sẽ thay đổi và yêu cầu phiên bản cũ hơn. Nhưng nếu bạn sử dụng hệ thống kiểm soát phiên bản Git, hệ thống này sẽ lưu trữ tất cả các phiên bản, vì vậy bạn không cần phải giữ mã bẩn. Xóa nó và làm cho mã của bạn sạch sẽ hơn.
- Tránh làm việc với một mảng lớn. Tránh tạo một mảng cho một tập dữ liệu lớn; thay vào đó, hãy sử dụng một lớp. Điều đó làm cho nó dễ đọc hơn, chưa kể rằng nó tạo ra một sự an toàn bổ sung cho ứng dụng của bạn.
- Không lặp lại mã. Mỗi khi bạn viết một phương thức, hãy tự hỏi bản thân xem liệu điều gì đó tương tự đã được xây dựng chưa. Kiểm tra thư viện mã hoặc tài liệu khác.
- Đừng "hardcode". Xác định hằng số hoặc sử dụng các biến thay vì fix cứng các giá trị. Việc sử dụng biến sẽ không chỉ làm cho nó có thể đọc được mà còn giúp bạn dễ dàng thay đổi nếu nó đang được sử dụng ở nhiều nơi.

Dirty code example	Clean code example

if (7 == $today) {
return 'It is holiday';
}
	const SATURDAY = 7;
	if (self::SATURDAY == $today) {
	return 'It is holiday';
	}

- Làm cho câu lệnh có thể đọc được. Để làm cho câu lệnh có thể đọc được, hãy giữ dòng ngắn gọn để bạn không cần phải cuộn theo chiều ngang để đọc hết dòng.

Một số mẹo khác cho mã sạch hơn

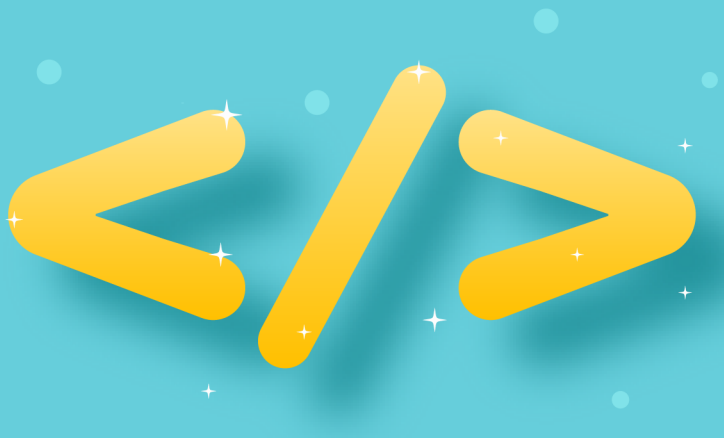
- Tự xem lại mã của bạn. Xem lại mã của bạn một lần sau một thời gian. Tôi chắc chắn rằng bạn sẽ tìm thấy một cái gì đó mới để cải thiện mỗi khi bạn xem lại nó.
- Xem lại mã của bạn với đồng nghiệp. Xem lại mã của đồng nghiệp của bạn và yêu cầu họ xem lại mã của bạn. Đừng ngần ngại xem xét các đề xuất, các phản hồi của đồng nghiệp rất tốt với bạn
- Sử dụng coding convention. Nếu bạn đang viết cho PHP, hãy sử dụng hướng dẫn về kiểu viết mã của PSR-2.
- Sử dụng TDD, bạn nên sử dụng phương pháp TDD và viết các bài kiểm tra đơn vị để tăng chất lượng mã nguồn...

Tôi mong rằng qua bài viết này, bạn sẽ nghiệm lại nhiều điều và khi đọc lại mã nguồn cũ, tôi tin bạn sẽ thốt ra: "mình đã code cái quái gì vậy". Thực hành Clean Code không phải chuyện ngày một ngày hai, hãy luyện tập từng ngày với những thay đổi nhỏ nhất. "Những thay đổi nhỏ, sẽ làm nên thành công to". Chúc bạn thành công.

Nguồn: https://simpleprogrammer.com/clean-code-principles-better-programmer

Ban biên tập

Nguyễn Khắc Nhật
Nguyễn Việt Khoa
Nguyễn Khánh Tùng
Nguyễn Bình Sơn
Đặng Huy Hoà
Dư Thanh Hoàng
Đỗ Minh Hải
Nguyễn Thị Hiền



Thiết kế

Đỗ Đình Tâm