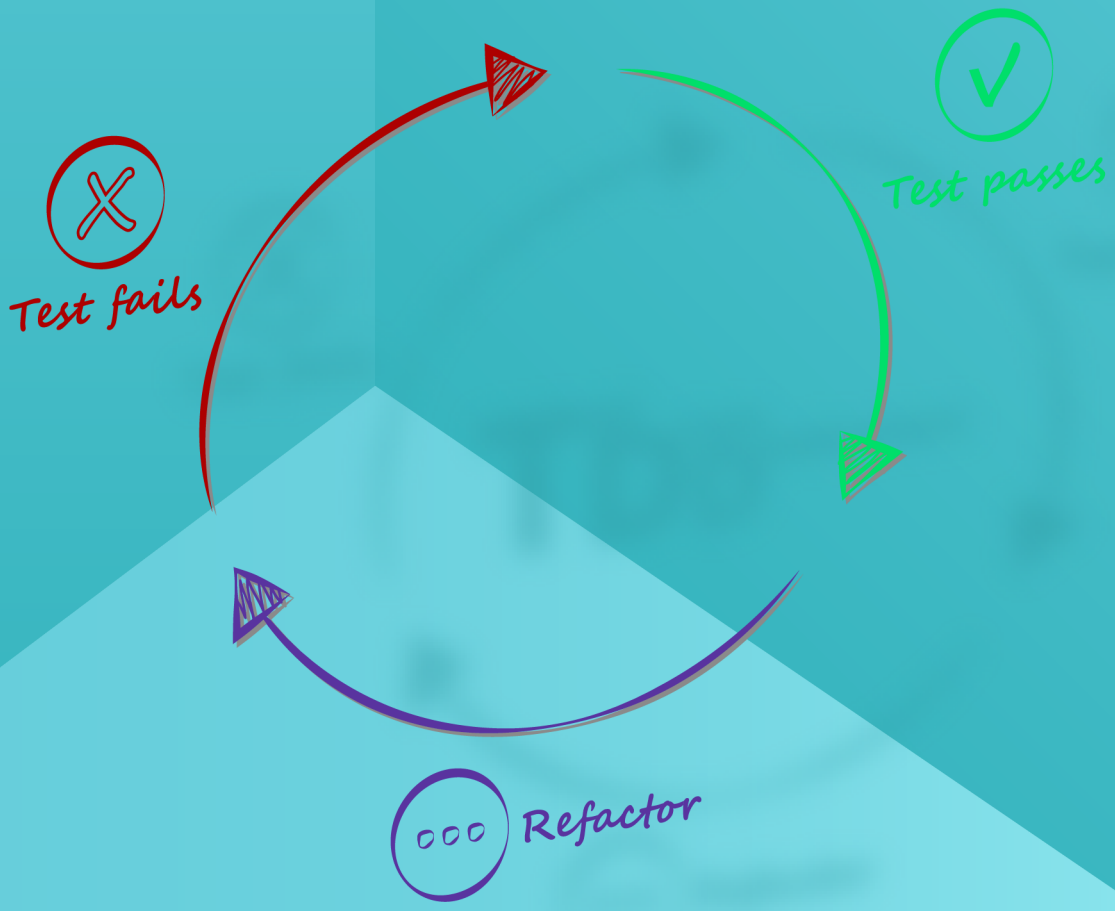
Tạp_Chí
{LậpTrình ():}

Số phát hành Vol.8 09 2020

Tạp chí LẬP TRÌNH

tapchilaptrinh.vn



Tại sao tái cấu trúc lại tốt?

Những dấu hiệu mã xấu

Tái cấu trúc mã nguồn

VOL. 8

MỤC LỤC

LỜI MỞ ĐẦU

Quý bạn đọc thân mến,

Refactoring (Tái cấu trúc), cùng với Clean Code, Automated Testing, TDD, Design Pattern là những chủ đề gần gũi và liên quan chặt chẽ đến nhau, tạo nên một khối nội dung vô cùng hữu ích và thiết thực đối với lập trình viên. Có thể nói rằng đây là những kỹ thuật cốt lõi để tạo nên những thiết kế tốt và giúp cho mã nguồn của chúng ta ngày càng trở nên tốt hơn, chúng ta ngày càng làm việc dễ dàng hơn.

Refactoring cũng tương đồng với tư duy của cải tiến liên tục - áp dụng những thay đổi nhỏ một cách liên tục để tạo nên một tác động lớn về lâu dài. Nếu xét về phương diện kỹ thuật thì hầu hết các cách tái cấu trúc mã nguồn đều khá dễ thực hiện, nhưng đó chưa phải là mấu chốt của vấn đề. Cái quan trọng nhất để triển khai thành công refactoring đó là duy trì được thói quen của các cá nhân và hình thành được văn hóa trong các nhóm. Đây là việc làm đòi hỏi sự thay đổi về mặt tư duy, nỗ lực và bền bỉ trong thời gian dài.

Ấn phẩm Tạp chí Lập trình số này tập trung vào những nội dung Refactoring với mong muốn tạo nên một không gian để chúng ta cùng nhau trao đổi về chủ đề thiết thực này, qua đó thúc đẩy tinh thần và chia sẻ những hiểu biết cốt lõi nhất để giúp nhau tiến bộ.

Chúc quý độc giả thật nhiều thành công và tiến bộ.

--

Ban biên tập Tạp chí Lập trình

TÁI CẤU TRÚC MÃ NGUỒN **(CODE REFACTORING)**

Martin Fowler

Tái cấu trúc là một quá trình cơ học, hình thức và trong nhiều trường hợp rất đơn giản để làm việc với mã của hệ thống đã tồn tại để chúng trở nên "tốt hơn". Khái niệm "tốt hơn" là một khái niệm mang tính chủ quan, và không đồng nghĩa với việc làm cho ứng dụng chạy nhanh hơn mà thường được hiểu là theo các kỹ thuật hướng đối tượng, tăng an toàn kiểu dữ liệu, dễ đọc, dễ bảo trì và mở rộng.

Hiệu quả của tái cấu trúc

Sản xuất phần mềm sẽ không hiệu quả nếu như bạn không thể theo kịp thay đổi của thế giới. Nếu như chúng ta chỉ sản xuất ra các phần mềm trong một vài ngày thì đơn giản hơn rất nhiều. Nhưng trong thế giới này chúng ta có rất nhiều đối thủ cạnh tranh. Nên nếu bạn không tái cấu trúc phần mềm của mình, khi đối thủ có một số tính năng hữu ích mới mà bạn không cập nhật thì sản phẩm của bạn nhanh chóng bị lạc hậu. Bởi thế là một lập trình viên bạn phải đón nhận và hành động một cách thích hợp với những thay đổi. Và khi thực hiện tái cấu trúc mã là bạn đang làm điều đó.

Cải thiện thiết kế

Nếu không áp dụng tái cấu trúc khi phát triển ứng dụng, thì thiết kế sẽ ngày càng tồi đi. Vì khi phát triển ứng dụng thì ta sẽ ưu tiên cho các mục tiêu

ngắn hạn (đặc biệt khi áp dụng các quy trình phát triển linh hoạt), nên mã ngày càng mất đi cấu trúc. Một trong những tên của vấn đề này gọi là technical debt (nợ kỹ thuật). Khi xảy ra vấn đề thì rất khó để quản lý và dễ bị tổn thương. Thế nên việc áp dụng tái cấu trúc sẽ giúp cho mã giữ được thiết kế tốt hơn là ưu điểm quan trọng.

Mã dễ đọc hơn

Khi lập trình là chúng ta đang giao tiếp với máy tính để yêu cầu chúng làm điều mình muốn. Nhưng còn có người khác tham gia vào quá trình này là các lập trình viên khác hay chính chúng ta trong tương lai. Chúng ta biết khi lập trình thường sẽ có người phải đọc để kiểm tra xem có vấn đề với mã đó không hoặc để mở rộng hệ thống.

Nhưng có một vấn đề là khi làm việc, lập trình viên thường không nghĩ tới những người đó trong tương lai. Vậy thì trong trường hợp này tái cấu trúc đóng vai quan trọng là giúp cải thiện thiết kế của hệ thống, từ đó cũng giúp đọc mã dễ hơn.

Lợi ích hệ quả

Từ những lợi ích cơ bản ở trên ta có thêm các lợi ích khác: do hệ thống hiện thời có một thiết kế tốt hơn và mã dễ hiểu hơn, từ đó thì việc mở rộng hệ thống dễ dàng hơn, khó bị tổn thương hơn, nên tốc độ phát triển hệ thống luôn được duy trì; mã và thiết kế dễ đọc hơn, từ đó giúp tìm ra lỗi dễ dàng hơn; vì những mục tiêu ngắn hạn lập trình viên có thể chấp nhận một lỗ hổng nào đó về công nghệ hay thiết kế mà hiện thời không gây ảnh hưởng gì tới hệ thống, nhưng khi hệ thống lớn dần thì những lỗ hổng này được tích tụ và làm cho hệ thống dễ bị tổn thương, thế nên việc tái cấu trúc giúp nhanh chóng sửa những lỗ hổng này.

Thời điểm thực hiện

Khi thêm một chức năng mới

Khi thêm một chức năng mới, ta phải đọc lại mã để hiểu. Như vậy nếu lúc này ta thực hiện việc tái cấu trúc, mã sẽ dễ hiểu hơn, cộng với đó là này ta cũng dễ dàng hiểu mã hơn vì mình là người đã đọc và thực hiện việc tái cấu trúc. Một lý do khác là khi thực hiện việc tái cấu trúc vào thời điểm này thì thiết kế của hệ thống sẽ tốt hơn, từ đó việc mở rộng cũng dễ dàng hơn.

Khi sửa lỗi

Khi sửa lỗi ta cũng phải đọc mã, và như vậy việc tái cấu trúc làm mã dễ đọc hơn, có cấu trúc rõ ràng hơn từ đó dễ dàng phát hiện lỗi là điều cần thiết. Và bởi thế nếu bạn được gán là người sửa một lỗi nào đó thì bạn cũng thường được gán là người phải tái cấu trúc mã.

Khi rà soát mã

Nhiều tổ chức thực hiện việc rà soát mã (code review). Rà soát mã giúp cho các lập trình viên giỏi truyền lại cho các lập trình viên ít kinh nghiệm hơn, giúp cho mọi người viết mã rõ ràng hơn. Mã có thể là rất rõ ràng với tác giả, nhưng với người khác thì có thể không, bởi thế rà soát mã sẽ làm cho nhiều người đọc mã hơn. Có nhiều người đọc mã thì mã phải dễ đọc hơn và có nhiều ý tưởng hơn được trao đổi giữa các thành viên trong nhóm hơn. Bởi thế khi bạn thực hiện rà soát bạn phải đọc mã. Lần đầu bạn đọc bạn bắt đầu hiểu mã. Lần tiếp theo bạn sẽ có nhiều ý tưởng hơn để tái cấu trúc mã, từ đó bạn có thể thực hiện việc tái cấu trúc.

Các "mã bẩn" thường gặp

Chúng ta đã biết cần thực hiện tái cấu trúc khi nào, nhưng có một câu hỏi khác là mã như thế nào thì cần tái cấu trúc? Khái niệm mã bẩn (code smell) là mã có thể sinh vấn đề một cách lâu dài, sẽ giúp ta phát hiện mã cần phải tái cấu trúc. Sau đâu chúng ta sẽ liệt kê một số loại mã bẩn thường gặp:

Mã lặp

Là những đoạn mã xuất hiện nhiều hơn một lần trong một hoặc nhiều ứng dụng của một chủ thể. Đó là hệ quả của các hành động: sao chép mã; các chức năng tương tự được viết bởi các lập trình viên khác nhau. Hệ quả là mã trở nên dài hơn, khó hiểu hơn và khó bảo trì hơn.

Ví dụ: đoạn mã tính giá trị trung bình của một mảng số nguyên trên C

```
extern int array1[];
extern int array2[];
int sum1 = 0;
int sum2 = 0;
int average1 = 0;
int average2 = 0;
for (int i = 0; i &lt; 4; i++)
{
   sum1 += array1[i];
}
average1 = sum1/4;
for (int i = 0; i &lt; 4; i++)
{
   sum2 += array2[i];
}
average2 = sum2/4;
```

Ta thấy hai vòng lặp for là giống nhau.

Hàm dài

Hàm dài là quá phức tạp, có lượng mã lớn. Nên khó để hiểu, triển khai, bảo trì và tái sử dụng. Nên việc tách thành các hàm nhỏ hơn là điều cần thiết.

Lớp lớn

Lớp lớn là lớp chứa quá nhiều thuộc tính và chức năng, thường là của nhiều lớp khác. Bởi thế cúng khó để đọc, bảo trì và tái sử dụng. Nên cần thực hiện các kỹ thuật cần thiết để phân bổ thành nhiều lớp khác nhau.

Hàm có nhiều tham số đầu vào

Khi một hàm có nhiều tham số đầu vào sẽ gây khó khăn để đọc, dùng và thay đổi. Với các ngôn ngữ lập trình hướng đối tượng ta có thể nhóm các tham số có liên quan vào một đối tượng để giảm số lượng tham số đầu vào.

Tính năng không phải của lớp

Là hiện tượng một phương thức không nên thuộc một lớp, nhưng do phương thức muốn sử dụng các dữ liệu của lớp đó nên lập trình viên đã gán phương thức cho lớp. Đây là một vi phạm trong lập trình hướng đối tượng. Ta cần trả phương thức đó về đúng đối tượng.

Lớp có quan hệ quá gần gũi

Đó là hiện tượng hai lớp có thể truy xuất vào các thuộc tính riêng tư của nhau một cách không cần thiết. Điều đó dẫn đến là chính các lớp đó hay lớp con của chúng có thể thay đổi các thuộc tính của lớp con lại một cách "vô thức".

Lớp quá nhỏ

Việc tạo, bảo trì và hiểu một lớp tốn tài nguyên. Vậy nếu lớp đó quá nhỏ thì ta nên xóa bỏ lớp đó đi.

Lệnh switch

Một trong những dấu hiệu tốt của lập trình hướng đối tượng là việc không dùng lệnh switch. Vấn đề của lệnh switch chủ yếu là vấn đề lặp mã. Bạn thường gặp những lệnh switch để phân bổ các chức năng ở nhiều nơi khác nhau trong cùng một ứng dụng. Và mỗi khi bạn thêm một tính năng mới, bạn phải tìm ở tất cả các lệnh này để thay đổi.

Từ chối kế thừa

Điều này xảy ra khi một lớp được thừa kế dữ liệu cũng như các tính năng của lớp cha, nhưng lại không cần phải dùng tới chúng.

Định danh quá dài hoặc quá ngắn

Các định danh cần mô tả đủ ý nghĩa để mã dễ đọc hơn và tránh gây hiểu nhầm. Bởi thế các định danh quá ngắn thường không mô tả hết ý nghĩa và gây ra nhầm lẫn. Nhưng các định danh quá dài cũng có vấn đề tương tự. Bởi thế trong trường hợp này chúng ta có thể viết tắt hay tìm định danh thay thế.

Dùng quá nhiều giá trị

Nếu trong mã có nhiều giá trị thì khi cần phải thay đổi các giá trị đó vì một lý do nào đó (ví dụ thay đổi độ chính xác) thì bạn cần phải thay đổi ở nhiều nơi. Không chỉ thế mà không phải ai và lúc nào cũng nhớ những giá trị đó có ý nghĩa gì, nên làm cho mã trở nên khó đọc hơn. Trong trường hợp này bạn có thể thay bằng cách đặt tên cho các hằng.

TẠI SAO
TÁI CẤU TRÚC LẠI TỐT?

Nguyễn Bình Sơn

Mã nguồn có hai phạm vi giá trị: những gì nó làm được bây giờ, và những gì nó sẽ làm được vào những ngày sau. Phần lớn thời gian chúng ta tập trung vào những gì mã nguồn làm được vào ngày hôm nay. Mỗi khi chúng ta sửa lỗi hay thêm chức năng, chúng ta tạo giá trị cho mã nguồn ngày hôm nay bằng cách tăng khả năng của nó.

Bạn không thể viết mã mãi được nếu không nhận thức được rằng mã hiện tại chỉ là một phần của câu chuyện, một góc nhỏ trong bức tranh tổng thể của mã nguồn khi nhìn bằng con mắt bốn chiều. Nếu bạn vẽ được góc bức tranh hôm nay nhưng khiến cho bức tranh không thể vẽ tiếp được, chúng ta không thể gọi đó là hiệu quả.

Và mặc dù biết như vậy, bạn cũng không thể dự chắc được điều gì về mã nguồn của ngày mai. Có thể bạn sẽ viết cái này, có thể bạn sẽ viết cái kia, cũng có thể là một cái gì đó bạn chưa bao giờ nghĩ đến. Nhưng nếu bạn không có hành động gì cho ngày mai, ngày mai đáng lẽ phải đến sẽ không bao giờ đến.

Tái cấu trúc chính là chuẩn bị. Nếu bạn thấy những phương án của ngày hôm qua trở nên vô nghĩa vào ngày hôm nay, hãy thay đổi phương án. Rồi thì bạn có thể yên tâm làm công việc của hôm nay. Ngày mai nếu việc đó tái diễn, bạn hãy tiếp tục.

Tại sao tái cấu trúc lại có tác dụng? Vậy tại sao chương trình lại khó viết? Chúng ta thường gặp khó ở bốn điểm:

- Những chương trình khó đọc thì khó sửa
- Những chương trình logic lặp lại thì khó sửa
- Những chương trình đòi hỏi bạn sửa cả những hành vi khác khi bạn thực hiện một thay đổi, thì khó sửa
- Những chương trình có nhiều logic rẽ nhánh phức tạp thì khó sửa

Vậy là chúng ta sẽ muốn chương trình của chúng ta dễ đọc, mỗi logic nằm ở một và chỉ một vị trí, không bắt chúng ta phải thực hiện những thay đổi nguy hiểm trên những chức năng khác có sẵn, và các logic điều kiện được thể hiện đơn giản hết mức có thể.

Tái cấu trúc không làm thay đổi chức năng của chương trình, nhưng tái cấu trúc giúp chúng ta tất cả những điều trên. Thế nên tái cấu trúc giúp chúng ta viết mã hiệu quả hơn.

CÁC NGUYÊN TẮC TÁI CẤU TRÚC

Nguyễn Bình Sơn

Nguyên tắc về cách hiểu

Có một vài ràng buộc khắt khe trong cách chúng ta hiểu về Tái Cấu Trúc. Đầu tiên, tái cấu trúc nghĩa là thay đổi hành vi bên trong của một phần mềm để phần mềm trở nên dễ hiểu và dễ sửa đổi hơn mà không phải thay đổi hành vi nhìn thấy từ bên ngoài.

Để rõ hơn, điều này có nghĩa là tái cấu trúc sẽ làm cho mã trở nên sạch hơn. Nếu bạn đã thực hành có chủ đích một vài kỹ thuật tái cấu trúc, bạn sẽ thấy chúng giúp bạn làm sạch mã một cách hiệu quả. Bạn biết có vấn đề gì, bạn biết phải làm gì, và bạn biết phải làm như thế nào. Nếu nhìn rộng ra, chúng ta có thể coi rằng chỉ những thay đổi nào không làm thay đổi hành vi của phần mềm nhưng lại làm cho phần mềm dễ hiểu và dễ sửa hơn mới được gọi là tái cấu trúc. Một ví dụ đối lập đó là các phép tối ưu hiệu năng, chúng không làm thay đổi hành vi của phần mềm, nhưng chúng thường làm cho mã trở nên khó hiểu hơn, tất nhiên bạn vẫn phải tối ưu hiệu năng, nhưng đó không gọi là tái cấu trúc.

Một ý nữa để nhấn mạnh rằng tái cấu trúc không làm thay đổi hành vi của phần mềm. Phần mềm vẫn có cùng chức năng như trước. Và bất cứ người dùng nào cũng có thể nói rằng mọi chuyện vẫn như cũ. Điều này dẫn chúng ta đến với ẩn dụ của Kent Beck về hai chiếc mũ.

Hai chiếc mũ

Bạn phải luôn hiểu rằng khi thực hành tái cấu trúc, bạn phân biệt rành mạch hai hoạt động khác nhau: thêm chức năng, và tái cấu trúc. Khi bạn thêm chức năng, bạn không thay đổi mã hiện có; chỉ thêm khả năng mới mà thôi. Bạn làm việc đó bằng cách bổ sung kiểm thử và tập trung vượt qua kiểm thử. Khi bạn tái cấu trúc, bạn không thêm chức năng mới và tập trung vào tái cấu trúc. Bạn không bổ sung bất cứ kiểm thử nào (trừ khi bổ sung một phép thử còn thiếu); bạn chỉ thay đổi mã nguồn khi thực sự cần, để đáp ứng sự thay đổi từ giao diện mã nguồn.

Bạn sẽ đội lần lượt hai chiếc mũ và thường xuyên tráo đổi. Bạn thêm chức năng, và nhận ra rằng sẽ dễ hơn nếu tái cấu trúc một chút, vậy nên bản đổi mũ và tái cấu trúc. Một khi mã đã tốt hơn, bạn đổi mũ tiếp và bổ sung chức năng. Rồi lại đổi mũ tiếp và tái cấu trúc. Tất cả có thể diễn ra chỉ trong vòng mười phút, nhưng bạn phải luôn nhận thức rõ ràng mình đang đội chiếc mũ nào.

Nguyên tắc về động cơ

Tái cấu trúc không phải viên đạn bạc để giải quyết mọi vấn đề. Nó là một công cụ giá trị, nhưng cho những mục đích cụ thể.

Cải thiện thiết kế của phần mềm

Không thực hành tái cấu trúc, thiết kế của chương trình sẽ ngày càng xấu đi. Cứ mỗi khi mã nguồn được sửa đổi cho những mục đích ngắn hạn, mã sẽ dần mất đi kết cấu tổng thể. Sẽ ngày càng khó nhận ra được thiết kế đằng sau của mã, và như thế mọi chuyện càng tệ đi nhanh hơn. Tái cấu trúc giúp giữ lại đôi chút. Tái cấu trúc thường xuyên sẽ giúp giữ cho mã nguồn được sắc sảo.

Những thiết kế tệ thường khiến mã trở nên dài dòng, bởi hay có những khối mã nhỏ lặp lại ở đâu đó. Và khử những mã lặp này là một trong những nhiệm vụ quan trọng của thiết kế. Khử mã lặp không giúp chương trình chạy nhanh hơn, bởi những thao tác của chương trình hầu như giữ nguyên. Nhưng càng nhiều mã thì càng khó sửa đổi mã. Mã cần phát ngôn mỗi thứ một lần và chỉ một lần. Nhiệm vụ của thiết kế, và trong trường hợp cải thiện thiết kế có sẵn thì tái cấu trúc chính là công cụ để thực hiện.

Khiến phần mềm trở nên dễ hiểu hơn

Lập trình, theo khía cạnh nào đó thì là đối thoại với máy tính. Bạn muốn máy tính làm điều gì đó, và máy tính cần làm đúng như thế. Ở chỗ trống giữa điều bạn muốn và điều sẽ diễn ra chính là mã nguồn. Nhưng không chỉ có máy tính, rồi sẽ có ai đó cũng đọc mã nguồn của bạn, để thực hiện một vài thay đổi cần thiết. Có khi đó mới là người đọc quan trọng nhất. Không ai lo lắng nếu máy tính phải chạy thêm vài vòng lặp, nhưng sẽ rất nhiều người quan tâm nếu ai đó mất cả tuần để thực hiện một thay đổi mà lẽ ra chỉ mất một giờ, nếu anh ta hiểu được mã nguồn của bạn.

Chúng ta có xu hướng viết để mã hoạt động được, mà quên nghĩ về những người đọc mã sau này. Tái cấu trúc là thao tác giúp bạn thay đổi nhịp điệu quen thuộc đó. Bạn tái cấu trúc một khối mã nguồn hoạt động tốt nhưng không được cấu trúc tốt. Bạn bỏ ra một chút thời gian, và mã nguồn trở nên rõ ý hơn, cứ như là mã nói lên chính ý bạn muốn nói.

Một khi mã nguồn đã rõ ràng hơn, bạn sẽ không còn phải chiến đấu để hiểu được các chi tiết vụn vặt, bạn sẽ dần nhìn ra được thiết kế và như thế làm cho mã sạch hơn đã trở thành nền tảng để bạn xây dựng nên kiến trúc tốt hơn.

Tái cấu trúc giúp dễ tầm lỗi hơn

Dĩ nhiên mã dễ hiểu hơn thì dễ tầm lỗi hơn, nhưng không phải bao giờ chúng ta cũng có mã sạch. Một vài người trong chúng ta có thể tìm ra lỗi giữa một "nùi" mã, một số khác thì không. Khi thực hành tái cấu trúc, chúng ta ngâm rất sâu vào ý nghĩa của mã cũng như cấu trúc của chương trình, và như thế lỗ hổng trong mã cứ như thể tự hiện lên trong tâm trí.

Tái cấu trúc giúp lập trình nhanh hơn

Tái cấu trúc giúp tăng chất lượng mã, cải thiện thiết kế, tăng tính dễ đọc, giảm lỗi. Chúng ta nghĩ gì mà lại cho rằng những điều đó không làm cho chúng ta viết mã nhanh hơn?

Lưu ý rằng một trong những điểm cộng lớn nhất của thiết kế tốt đó là giúp phát triển nhanh chóng. Với thiết kế tốt, bạn đi rất nhanh. Còn nếu không thì ngược lại, bạn sẽ dành thời gian đáng lẽ để phát triển tính năng mới hay cải thiện chất lượng cho công việc tìm và sửa lỗi. Thời gian bạn bỏ ra để sửa đổi mã sẽ ngày càng dài, chức năng mới sẽ ngày càng cần nhiều mã, ngày càng nhiều lỗi, ngày càng có nhiều bản vá và ngày càng cần nhiều thời gian để vá.

Thiết kế tốt giúp chúng ta giữ được tốc độ phát triển nhanh và ổn định. Tái cấu trúc giúp cải thiện thiết kế và giữ cho thiết kế không bị suy hại. Tái cấu trúc giúp chúng ta phát triển nhanh hơn.

Khi nào cần tái cấu trúc

Nhiều nhóm phát triển đặt một khung thời gian nhất định cho tái cấu trúc. Có thể là định kỳ hàng tháng, hàng quý, hay khi phát triển xong một cụm chức năng lớn. Đó không phải là chiến thuật tốt. Tái cấu trúc không nên được thực hiện vào một khung giờ bên lề. Nó nên được thực hiện mọi lúc, trong những ngăn xếp thời gian nhỏ. Chúng ta không quyết định tái cấu trúc, tái cấu trúc không phải là một mục tiêu. Bạn muốn làm một thứ gì đó, và tái cấu trúc giúp bạn làm thứ đó.

Quy tắc số ba

Đây là tên một quy tắc để ra quyết định khi nào cần tái cấu trúc. Lần đầu tiên bạn làm chức năng gì đó, bạn cứ làm thôi. Lần thứ hai bạn làm một thứ tương tự, bạn biết và bạn sợ mã lặp, nhưng bạn vẫn làm. Lần thứ ba bạn làm một thứ tương tự, đó là lúc bạn tái cấu trúc.

Tái cấu trúc khi thêm chức năng

Thời điểm tái cấu trúc thường gặp nhất là khi bổ sung chức năng vào phần mềm. Tái cấu trúc vào lúc này có thể giúp chúng ta hiểu thêm về mã của những gì cần phải sửa đổi. Bên cạnh việc bạn làm cho mã hiện có trở nên tốt hơn, bạn cũng làm cho chức năng bạn sắp viết thêm vào trở nên tốt hơn.

Đôi khi, bạn còn không thể thêm chức năng (một cách dễ dàng) nếu không tái cấu trúc, đó là bởi thiết kế hiện tại không sẵn sàng để thêm chức năng mới. Bạn sửa "lỗi" thiết kế đó bằng tái cấu trúc. Dĩ nhiên ai đó rồi sẽ tạ ơn bạn, nhưng trước mắt thì bạn thực hiện tái cấu trúc đó để khiến bởi vì đó là cách nhanh nhất và dễ nhất để bạn thực hiện được công việc của mình.

Tái cấu trúc khi sửa lỗi

Bạn cần tái cấu trúc khi thực hiện sửa lỗi, và phần lớn là để mã dễ hiểu hơn. Thông thường hoạt động đó khiến bạn nhanh chóng tìm được vấn đề. Cần nhớ rằng, sự hiện diện của lỗi là một dấu hiệu cho thấy bạn cần tái cấu trúc, bởi vì mã đã không đủ rõ ràng để ai đó có thể nhìn vào và phát hiện ra ngay vấn đề.

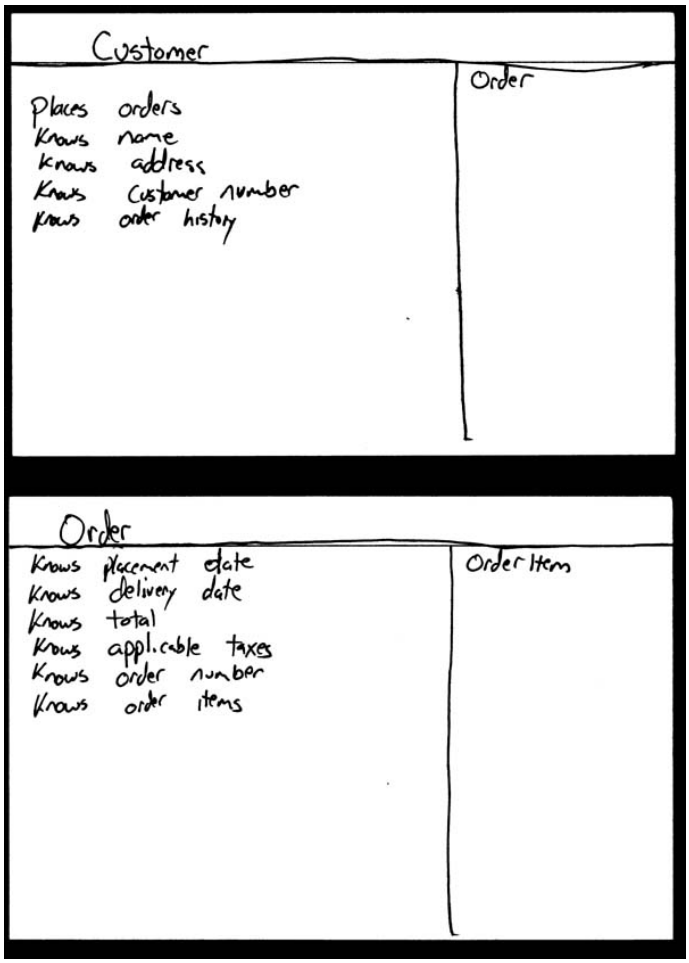
Tái cấu trúc khi review mã

Chúng ta nên thường xuyên review và nhờ người khác review mã. Review chéo giúp phát tán tri thức cho cả nhóm phát triển, giúp lập trình viên có kinh nghiệm truyền đạt tri thức cho những người non tay hơn, giúp mọi người hiểu thêm về nhiều khía cạnh các nhau trong hệ thống phần mềm. Không chỉ thế review cũng rất quan trọng để viết mã sạch. Mã nguồn của bạn dễ hiểu đối với bạn, nhưng nhóm của bạn thì không thấy thế. Đó là lẽ tự nhiên, và review tạo cơ hội để mọi người nảy ra những đề xuất hữu ích.

Khi review, trước tiên bạn đọc mã, cố gắng hiểu ý định của mã đó, và đưa ra gợi ý. Bạn đề nghị với tác giả của mã những gợi ý của mình, rằng có thể dễ đọc hơn ở chỗ này dễ triển khai hơn ở chỗ kia. Hãy thảo luận, và sau đó bạn hãy thực hiện những tái cấu trúc đã chốt. Làm như thế rồi thì ý định của mã nguồn sẽ hiển hiện trước mắt bạn, bạn không còn cần phải tưởng tượng nữa, điều đó có thể sẽ tạo điều kiện cho bạn đưa ra được những gợi ý ở tầm cao hơn, thứ mà bạn có thể sẽ không thể nghĩ ra được trước đó.

Bạn chỉ cần một nhóm hai người để review mã. Bạn bình phẩm và đưa ra gợi ý, hai bạn cùng đưa ra quyết định về những thay đổi có thể thực hiện. Và hai bạn cùng thực hiện.

Với những review phạm vi rộng, chúng ta có thể tổ chức một nhóm review lớn. Lúc này đọc mã nguồn một cách trực tiếp có thể không phải là cách tốt. Các lược đồ UML và các bản CRC sẽ hữu ích hơn. Chúng ta thực hiện review mã nguồn với cá nhân, và review thiết kế với nhóm.



Nói như thế nào với sếp

Đây là rào cản cuối cùng, làm thế nào để nói chuyện với sếp về tái cấu trúc. Nếu sếp của bạn quen thuộc với công việc phát triển thì trao đổi về vấn đề này sẽ không quá khó. Nếu sếp của bạn là người truy cầu chất lượng, bạn hãy đề xuất về hoạt động tái cấu trúc khi review mã. Khó vô số tài liệu, sách, nghiên cứu, kinh nghiệm... đều đã chỉ ra rằng review mã có tác dụng tốt thế nào trong việc giảm lỗi và tăng tốc độ phát triển. Điều đó sẽ có sức thuyết phục rất lớn với sếp của bạn.

Tất nhiên sẽ có những sếp nói rằng chúng ta còn có cả tiến độ và deadline nữa. Bác Bob có đưa ra một lời khuyên gây tranh cãi trong tình huống này: Thế thì bạn cứ yên lặng mà làm thôi!

Đây không phải là chống đối. Các nhà phát triển phần mềm là những chuyên gia trong nghành của mình. Công việc của chúng ta là xây dựng phần mềm một cách hiệu quả nhất có thể. Chúng ta có kinh nghiệm rằng tái cấu trúc là một công cụ hỗ trợ rất lớn trong việc tạo ra phần mềm chất lượng cao và nhanh chóng. Nếu chúng ta cần thực hiện chức năng mới, chúng ta biết rằng nếu tái cấu trúc trước rồi sau đó mới thêm chức năng sẽ nhanh hơn. Nếu cần sửa lỗi, chúng ta cần hiểu cách hoạt động — và tái cấu trúc cũng lại là cách nhanh nhất để thực hiện việc này. Sếp của chúng ta muốn chúng ta thực hiện công việc theo cách nhanh nhất có thể. Cách nhanh nhất là tái cấu trúc; thế nên chúng ta tái cấu trúc.

ÁP DỤNG LUẬT DEMETER

Đặng Huy Hòa

"Không nói chuyện với người lạ"

Giới thiệu

Luật Demeter có thể giúp bạn áp dụng các quy tắc trừu tượng hơn của Lập trình Hướng đối tượng (OOP) để đảm bảo ứng dụng có thể mở rộng và có thể bảo trì.

Luật Demeter có các nguyên tắc sau:

- Mỗi đơn vị nên biết ít về đơn vị khác
- Mỗi đơn vị chỉ nên nói chuyện với bạn của nó (nói các khác, không được nói chuyện với người lạ).
- Chỉ nói chuyện với bạn thân

Trong thiết kế phần mềm (ở mức chi tiết), loose coupling (tạm hiểu là "liên kết lỏng kéo") là một thuật ngữ thường được đề cập đến. Đây một trong những yếu tố giúp phần mềm dễ dàng mở rộng và bảo trì.

Luật Demeter sẽ giúp giảm thiểu tight coupling (phụ thuộc chặt chẽ với nhau) giữa các thành phần trong ứng dụng. Nhờ đó, chúng ta có thể dễ dàng thay thế hoặc mở rộng khi có nhu cầu.

Luật Demeter

Lý thuyết

Luật Demeter cho hàm yêu cầu có phương thức M trong object O có thể chỉ liên quan tới các phương thức thuộc các loại object sau:

1. O - chính nó
2. Object là tham số của hàm M
3. Bất kì object nào được khởi tạo bên trong M
4. Object là một thuộc tính của O
5. Biến global, có thể truy cập được qua O (trong phạm vi M)

Ví dụ tốt

Để hiểu rõ về nội dung có phần nặng lý thuyết ở trên, chúng ta có thể xem qua ví dụ sau để có được hình dung đầu tiên. Ví dụ này được viết với ngôn ngữ lập trình Java:

```
public class LawOfDemeter
{
  private Topping cheeseTopping;
  public void goodExamples(Pizza pizza)
  {
    Foo foo = new Foo();
    // (1) Trường hợp gọi phương thức của chính nó
    doSomething();
    // (2) Trường hợp gọi các phương thức của đối
tượng là tham số đầu vào
    int price = pizza.getPrice();
   // (3) Trường hợp gọi phương thức của bất kì đối
tượng nào được khởi tạo bên trong class này
    cheeseTopping = new CheeseTopping();
    float weight = cheeseTopping.getWeightUsed();
   // (4) Tương tự trường hợp (3)
    foo.doBar();
  }
  private void doSomething()
  {
    // ...
  }
}
// Mã nguồn tham khảo tại https://alvinalexander.
com/java/java-law-of-demeter-java-examples/
```

Ví dụ chưa tốt

Dưới đây là một ví dụ vi phạm các nguyên tắc của Luật Demeter:

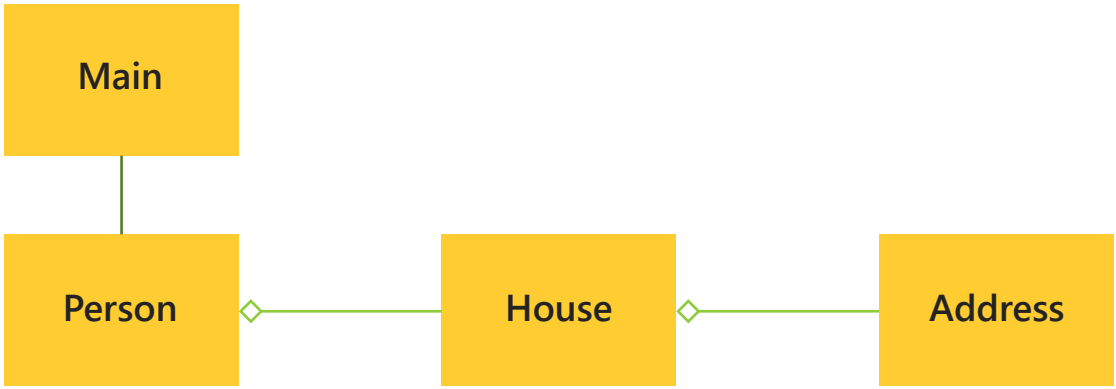
```
class BasicExample {
 public String getStreetName(Employee employee) {
  Address address = employee.getAddress();
  return address.getStreetName();
 }
}
```

Để có được address, chúng ta gọi phương thức getAddress() của employee tại dòng thứ ba. Đối tượng address không phải là đối tượng được liệt kê trong danh sách các tình huống mà chúng ta được phép tương tác trực tiếp. Thế nhưng tại dòng thứ tư, chúng ta thấy rằng kết quả trả về được lấy từ phương thức getStreetName() thuộc đối tượng address.

Giải pháp khắc phục

Thay vì gọi trực tiếp getStreetName() từ address để lấy được thông tin tên đường phố, chúng ta tách đoạn mã trên và khai báo một phương thức trong khai báo class Employee để trả về giá trị từ getStreetName(). Nhờ thế, chúng ta có thể gọi trực tiếp phương thức getStreetName() này với đối tượng employee như dưới đây:

```
class Employee {
 private Address address;
 public String getStreetName() {
  return address.getStreetName();
 }
}
class BasicExample {
 public String getStreetName(Employee emplyee) {
  employee.getStreetName();
 }
}
```



Lợi ích

Việc tuân thủ các nguyên tắc trong Luật Demeter sẽ giúp chúng ta đạt được những lợi ích dưới đây khi lập trình:

- Các đối tượng sẽ tránh được việc phụ thuộc chặt chẽ với nhau
- Tái sử dụng class dễ dàng hơn
- Dễ dàng thay thế class khi yêu cầu thiết kế thay đổi
- Mã nguồn dễ kiểm thử hơn
- Các class được thiết kế theo hướng này sẽ ít lỗi hơn

Kết luận

Hiểu về Luật Demeter và áp dụng được vào lập trình sẽ giúp bạn có sản phẩm được thiết kế tốt hơn. Bên cạnh đó, khi kết hợp với những kỹ năng về Refactoring (Tái cấu trúc mã nguồn - Chủ đề của Tạp Chí Lập Trình số này) sẽ giúp bạn cải thiện được chất lượng mã nguồn. Hy vọng bài viết này giúp bạn có thêm chút kiến thức để dễ dàng ra quyết định khi thực hiện các nhiệm vụ liên quan đến refactoring.

TÁI CẤU TRÚC HÀM (FUNCTION)

Dư Thanh Hoàng

Chúng ta có thể dễ dàng bảo trì và nâng cấp mã JavaScript của mình bằng dọn dẹp và tối ưu nó thường xuyên.

Trong bài viết này, chúng ta sẽ xem xét một số số ý tưởng tái cấu trúc để có thể tối ưu được các Hàm và Lớp trong JavaScript.

Hạn chế việc sử dụng tham số trong Hàm

Chúng ta nên hạn chế việc sử dụng thẳng tham số để thao tác trong Hàm, thay vào đó chúng ta nên gắn giá trị tham số đó cho 1 biến trước khi thao tác với nó.

Ví dụ: thay vì viết như sau:

```
const discount = (subtotal) => {
  if (subtotal > 50) {
    subtotal *= 0.8;
  }
}
```

Chúng ta viết:

```
const discount = (subtotal) => {
  let _subtotal = subtotal;
  if (_subtotal > 50) {
    _subtotal *= 0.8;
  }
}
```

Thay thế Phương thức(Method) bằng một Hàm(Funtion)

Chúng ta có thể biến Phương thức thành một Hàm riêng để tất các Lớp(Class) đều có thể truy cập nó.

Ví dụ, thay vì viết như sau:

```
class Foo {
  hello() {
    console.log('hello');
  }
  //...
}
class Bar {
  hello() {
    console.log('hello');
  }
  //...
}
```

Chúng ta có thể tách phương thức hello thành một hàm riêng như sau:

```
const hello = () => {
  console.log('hello');
}class Foo {
  //...
}class Bar {
  //...
}
```

Vì phương thức hello không phụ thuộc vào this và bị lặp lại ở cả 2 lớp nên chúng ta chuyển nó thành hàm riêng để tránh việc trùng lặp code.

Các thuật toán thay thế

Chúng ta hãy thay thế một thuật toán bằng một thuật toán rõ ràng hơn.

Ví dụ: thay vì viết như sau:

```
const doubleAll = (arr) => {
  const results = [];
  for (const a of arr) {
    results.push(a * 2);
  }
  return results;
}
```

Chúng ta viết như sau:

```
const doubleAll = (arr) => {
  return arr.map(a => a * 2);
}
```

Chúng ta đã làm cho hàm doubleAll của mình ngắn gọn hơn bằng cách thay thế vòng lặp bằng các phương thức có sẵn của mảng có chức năng tương tự.

Cả hai đều nhân đôi giá trị các phần tử trong mảng và trả về nó.

Nếu như có một cách đơn giản hơn để làm điều gì đó, hãy dùng nó.

Di chuyển phương thức

Chúng ta có thể di chuyển một phương thức(-method) từ lớp(class) này sang lớp(class) khác, nơi nó được gọi thường xuyên hơn.

Ví dụ: thay vì viết như sau:

```
class Foo {
  method() {}
}
class Bar {}
```

Chúng ta viết như sau:

```
class Foo {}
class Bar {
  method() {}
}
```

Chúng ta di chuyển các phương thức để nó nằm trong lớp sử dụng phương thức đó nhiều nhất.

Tách Lớp(Class)

Nếu Class của chúng ta phức tạp và thực hiện nhiều công việc thì chúng ta nên tách các chức năng phụ của Class thành những Class mới.

Ví dụ: thay vì viết như sau:

```
class Person {
  constructor(name, phoneNumber) {
    this.name = name;
    this.phoneNumber = phoneNumber;
  }
  addAreaCode(areaCode) {
    return `${areaCode}-${this.phoneNumber}`
  }
}
```

Chúng ta viết như sau:

```
class PhoneNumber {
  constructor(phoneNumber) {
    this.phoneNumber = phoneNumber;
  }
  addAreaCode(areaCode) {
    return `${areaCode}-${this.phoneNumber}`
  }
}
class Person {
  constructor(name, phoneNumber) {
    this.name = name;
    this.phoneNumber = new PhoneNumber(phoneNumber);
  }
}
```

Bằng cách này, chúng ta chuyển chức năng thêm mã vùng (addAreaCode) từ class Person và một Class mới là PhoneNumber, như vậy sẽ rõ ràng và chuyên biệt hơn.

Khi đó, cả hai Class chỉ làm một việc thay vì để một Class làm nhiều việc.

Tổng kết

- Chúng ta có thể tách một Lớp(Class) phức tạp đang thực hiện nhiều công việc thành những Class riêng thực hiện một loại công việc.
- Ngoài ra, chúng ta có thể di chuyển các Phương thức(Method) và Thuộc tính (Attribute) đến nơi nó được sử dụng nhiều nhất.
- Việc thao tác thẳng với tham số rất dễ bị gây ra sai sót, vì vậy chúng ta nên gán chúng cho một biến trước khi làm việc với chúng.

TÁI CẤU TRÚC MÃ NGUỒN: KHI NÀO CẦN, KHI NÀO KHÔNG!

Phan Văn Luân

Code refactoring - Tái cấu trúc mã nguồn là một quá trình được sử dụng trong phương pháp phát triển phần mềm DevOps bao gồm việc chỉnh sửa và làm sạch mã nguồn đã viết trước đó mà không thay đổi chức năng của mã.

Mục đích cơ bản của việc tái cấu trúc mã nguồn là làm cho mã nguồn hiệu quả hơn và có thể bảo trì được. Đây là chìa khóa trong việc giảm chi phí kỹ thuật vì việc xóa mã ngay bây giờ tốt hơn nhiều so với việc fix các lỗi tốn kém sau này. Tái cấu trúc mã, cải thiện khả năng đọc, làm cho quá trình QA và fix bug diễn ra suôn sẻ hơn nhiều. Và mặc dù nó không loại bỏ được bug, nhưng nó chắc chắn có thể giúp ngăn chặn chúng trong tương lai.

Và đó là lý do tại sao cần phải cấu trúc lại mã thường xuyên.

Tái cấu trúc mã rất quan trọng nếu bạn muốn tránh bị mã "thối" đáng sợ. Lỗi mã là kết quả từ mã trùng lặp, vô số bản vá lỗi, phân loại sai và các sai lệch lập trình khác. Việc có một cánh cửa quay vòng của các nhà phát triển khác nhau viết theo phong cách riêng của họ cũng có thể góp phần làm "thối" mã, vì không có sự gắn kết với tập lệnh mã hóa tổng thể.

Khi nào bạn nên xem xét việc tái cấu trúc phần mềm?

Thời điểm tốt nhất để xem xét tái cấu trúc là trước khi thêm bất kỳ bản cập nhật hoặc tính năng mới nào vào mã hiện có. Quay lại và làm sạch mã hiện tại trước khi thêm vào chương trình mới sẽ không chỉ cải thiện chất lượng của sản phẩm mà còn giúp các nhà phát triển trong tương lai dễ dàng xây dựng trên mã gốc hơn.

Một thời điểm khác để nghĩ đến việc tái cấu trúc là ngay sau khi bạn đưa một sản phẩm ra thị trường. Vâng, điều đó nghe có vẻ vô lý?. Cuối cùng, bạn đã tung ra một sản phẩm mà bạn đã làm việc trong nhiều tháng, thậm chí có thể là nhiều năm và bây giờ bạn phải quay lại từ đầu? Đây thực sự là thời điểm hoàn hảo để thực hiện một chút công việc dọn dẹp các mã xấu, vì rất có thể các nhà phát triển có nhiều khả năng hơn để làm việc tái cấu trúc trước khi chuyển sang công việc tiếp theo.

Các kỹ thuật thực hiện tái cấu trúc mã nguồn

Như đã đề cập trước đây, cách tốt nhất để cấu trúc lại là thực hiện theo từng bước nhỏ. Điều quan trọng là phải thực hiện trước khi thêm bất kỳ chức năng hoặc tính năng mới nào. Việc tái cấu trúc mã sẽ không thay đổi bất kỳ điều gì về cách sản phẩm hoạt động.

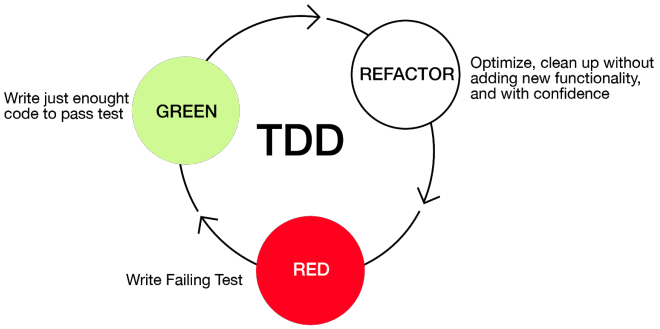
Tuy nhiên, có nhiều cách tiếp cận và kỹ thuật khác nhau để tái cấu trúc mã. Dưới đây là các cách phổ biến nhất:

Red-Green-Refactor

Một trong những kỹ thuật được sử dụng rộng rãi nhất để tái cấu trúc mã là quy trình Red-Green-Refactor được sử dụng trong phát triển theo phướng pháp Agile. Áp dụng phương pháp Red-Green-Refactor, các nhà phát triển chia nhỏ việc tái cấu trúc thành ba bước riêng biệt:

1. Hãy dừng lại và xem xét những gì cần được phát triển. [RED]

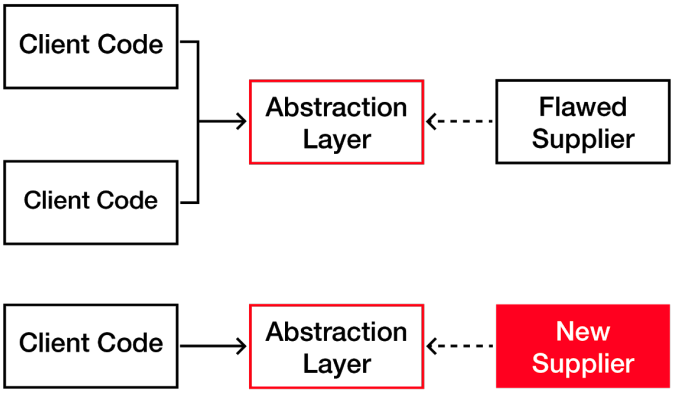
2. Nhận sự phát triển để vượt qua thử nghiệm cơ bản. [GREEN]
3. Thực hiện các cải tiến. [REFACTOR]



Tái cấu trúc bằng Abstraction

Tái cấu trúc theo hướng trừu tượng hoá là một phương pháp được sử dụng chủ yếu khi có một lượng lớn cấu trúc lại được thực hiện. Tính trừu tượng liên quan đến kế thừa lớp, hệ thống phân cấp và trích xuất. Mục tiêu của sự trừu tượng hóa là giảm bớt sự trùng lặp không cần thiết trong mã nguồn.

Một ví dụ về sự trừu tượng là phương pháp Pull-Up/Push-Down. Đây là hai hình thức tái cấu trúc đối lập liên quan đến các lớp. Phương thức Pull-Up kéo các phần mã vào một lớp cha để loại bỏ sự trùng lặp mã. Push-Down lấy nó từ một lớp cha và chuyển nó xuống các lớp con.



Composing Method

Việc soạn thảo liên quan đến việc hợp lý hóa mã để giảm sự trùng lặp. Điều này được thực hiện thông qua các quy trình khác nhau, bao gồm cả phương pháp chiết xuất và nội tuyến.

Khai thác liên quan đến việc chia nhỏ mã thành các phần nhỏ hơn để tìm và "trích xuất" sự phân mảnh. Sau đó, mã bị phân mảnh được chuyển sang một phương thức riêng biệt và được thay thế bằng một lệnh gọi đến phương thức mới này. Ngoài phương thức, trích xuất có thể liên quan đến các biến lớp, giao diện và cục bộ

Tái cấu trúc nội tuyến là một cách để giảm số lượng các phương thức không cần thiết trong khi đơn giản hóa mã. Bằng cách tìm tất cả các lệnh gọi đến phương thức và thay thế chúng bằng nội dung của phương thức, phương thức sau đó có thể bị xóa.

Simplifying Methods

Mã càng cũ thì càng bị cắt xén và phức tạp hơn. Do đó, nó có ý nghĩa khi đi vào và đơn giản hóa rất nhiều logic. Điều này có thể được thực hiện theo nhiều cách khác nhau, bao gồm hợp nhất các đoạn và biểu thức có điều kiện và thay thế có điều kiện bằng tính đa hình.

Việc đơn giản hóa các cuộc gọi phương thức liên quan đến việc điều chỉnh sự tương tác giữa các lớp. Thêm, xóa và giới thiệu các tham số mới cùng với việc thay thế các tham số bằng các phương thức rõ ràng và các cuộc gọi phương thức là tất cả các khía cạnh của sự đơn giản hóa.

Tính năng di chuyển giữa các đối tượng

Phương pháp này liên quan đến việc tạo các lớp mới và di chuyển chức năng giữa các lớp cũ và mới. Khi một lớp có quá nhiều thứ đang diễn ra, đã đến lúc chuyển một số mã đó sang lớp khác. Hoặc, mặt khác, nếu một lớp không thực sự hoạt động nhiều, bạn có thể di chuyển các tính năng của lớp đó sang một lớp khác và xóa hoàn toàn.

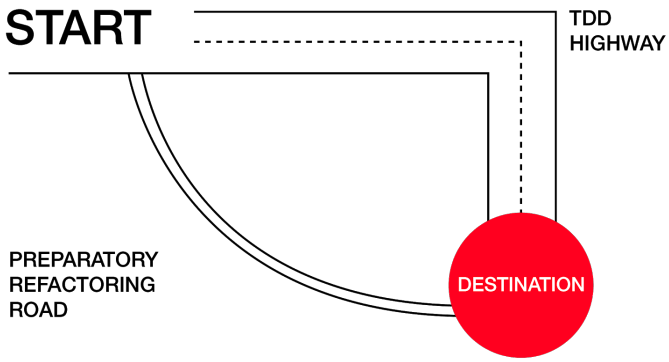
Preparatory Refactoring

Trong cuốn sách ***Refactoring: Improving the Design of Existing Code***, tác giả Martin Fowler nói về quá trình chuẩn bị cấu trúc lại. Điều này được thực hiện khi một nhà phát triển nhận thấy cần phải cấu trúc lại trong khi thêm một tính năng mới, vì vậy, đó thực sự là một phần của bản cập nhật phần mềm chứ không phải một quy trình tái cấu trúc riêng biệt. Bằng cách nhận thấy rằng mã cần được cập nhật tại thời điểm đó, nhà phát triển đang thực hiện phần việc của mình để giảm nợ kỹ

thuật trong tương lai.

Nhà phát triển phần mềm Jessica Kerr cung cấp một lời giải thích minh họa tuyệt vời cho việc tái cấu trúc chuẩn bị:

"Nó giống như tôi muốn đi 100 dặm về phía đông nhưng thay vì chỉ traipsing qua rừng, tôi sẽ lái xe 20 dặm về phía bắc đến đường cao tốc và sau đó tôi sẽ đi 100 dặm về phía đông ở ba lần so với tốc độ tôi có thể có nếu tôi chỉ đi thẳng đến đó. Khi mọi người thúc đẩy bạn chỉ cần đi thẳng đến đó, đôi khi bạn cần phải nói, "Chờ đã, tôi cần kiểm tra bản đồ và tìm tuyến đường nhanh nhất." Việc tái cấu trúc chuẩn bị thực hiện điều đó cho tôi. "



Bạn có thể tìm thấy hàng tá phương pháp khác để tái cấu trúc mã tại trang web của Martin Fowler và tại Refactoring.com. Giải pháp phù hợp nhất với bạn sẽ phụ thuộc vào quy mô và phạm vi giải pháp của bạn, khung thời gian bạn phải thực hiện tái cấu trúc và số lượng người có sẵn để hỗ trợ trong quá trình tổng thể.

Khi nào bạn không cần cấu trúc lại

Trước đó, chúng tôi đã nhấn mạnh rằng việc tái cấu trúc sẽ không bao giờ ảnh hưởng đến hiệu suất của một ứng dụng và nó chỉ được coi là một nỗ lực dọn dẹp các mã bẩn. Tuy nhiên, có những lúc, một ứng dụng cần được cải tiến hoàn toàn ngay từ đầu. Trong những trường hợp này, việc tái cấu trúc là không cần thiết, vì sẽ hiệu quả hơn nhiều nếu chỉ bắt đầu từ đầu.

Một tình huống khác mà sẽ là khôn ngoan nếu bạn bỏ qua việc tái cấu trúc là nếu bạn đang cố gắng đưa một sản phẩm ra thị trường trong một khung thời gian đã định. Tái cấu trúc có thể giống như đi xuống lỗ thỏ:

Một khi bạn bắt đầu, nó có thể trở nên khá tốn thời gian. Việc thêm bất kỳ mã hoặc thử nghiệm bổ sung nào vào một khung thời gian vốn đã chật hẹp sẽ dẫn đến sự thất vọng và chi phí bổ sung cho khách hàng của bạn.

Các phương pháp hay nhất để tái cấu trúc mã

Có một số phương pháp hay nhất và khuyến nghị liên quan đến việc tái cấu trúc mã. Một trong những cách tiếp cận thông minh nhất là áp dụng phương pháp Agile và thực hiện từng bước một, sau đó là kiểm thử. Đây là lý do tại sao rất nhiều nhà phát triển sử dụng phương pháp Agile là những người ủng hộ lớn cho việc tái cấu trúc mã.

Chia nhỏ quy trình tái cấu trúc thành các phần có thể quản lý và thực hiện kiểm thử kịp thời trước khi chuyển sang các bản cập nhật khác luôn dẫn đến ứng dụng chất lượng cao hơn và trải nghiệm phát triển tổng thể tốt hơn.

Dưới đây là một số phương pháp hay khác:

1. Refactor đầu tiên trước khi thêm bất kỳ tính năng mới nào

Bạn nên thực hiện tái cấu trúc bất cứ khi nào bạn được yêu cầu thêm các tính năng hoặc bản cập nhật mới vào giải pháp hiện có. Có, nó sẽ mất lâu hơn để hoàn thành dự án, nhưng nó cũng sẽ giảm số nợ kỹ thuật mà bạn hoặc chủ sở hữu sản phẩm sẽ phải giải quyết trong tương lai.

2. Lập kế hoạch dự án tái cấu trúc và tiến trình của bạn một cách cẩn thận

Một trong những phần khó nhất của quá trình tái cấu trúc mã là tìm thời gian để thực hiện đúng cách.

Suy nghĩ về mục tiêu tổng thể của bạn. Bạn chỉ muốn thay đổi tên biến để cải thiện khả năng đọc? Hay bạn muốn dọn dẹp toàn bộ? Cách tốt nhất để bạn tối ưu hóa mã trong một khung thời gian hợp lý là gì?

Kết quả quan trọng nhất của việc tái cấu trúc không chỉ là mã sạch hơn mà còn là nó thực sự

hoạt động. Và hãy nhớ rằng sẽ mất nhiều thời gian hơn bạn nghĩ, vì vậy hãy lập kế hoạch phù hợp và dành cho mình một chút thời gian.

3. Kiểm thử thường xuyên

Điều cuối cùng bạn muốn làm khi tái cấu trúc là làm rối tung thứ gì đó trong quy trình và tạo ra lỗi hoặc sự cố ảnh hưởng đến chức năng của sản phẩm. Đây là lý do tại sao việc kiểm thử trong suốt quá trình tái cấu trúc là bắt buộc. Đảm bảo rằng bạn có các bài kiểm tra thích hợp trước khi bắt đầu bất kỳ dự án tái cấu trúc nào.

4. Mời nhóm QA của bạn tham gia

Luôn luôn là một ý kiến hay để nhờ nhóm QA và thử nghiệm của bạn tham gia vào quá trình tái cấu trúc. Bất cứ khi nào bạn thực hiện các thay đổi đối với mã hiện có, ngay cả khi là một dự án dọn dẹp, nó có thể ảnh hưởng đến kết quả thử nghiệm.

Những thay đổi trong phân loại được thực hiện trong quá trình tái cấu trúc có thể khiến các bài kiểm tra cũ không thành công. Ngoài ra, các thử nghiệm mới có thể phải được tạo cho các hệ thống phần mềm cũ đã lỗi thời. Cả kiểm tra chuyên sâu và kiểm tra hồi quy phải được thực hiện như một phần của nỗ lực tái cấu trúc. Điều này sẽ đảm bảo rằng chức năng của giải pháp không bị ảnh hưởng theo bất kỳ cách nào.

Các nhóm phát triển sử dụng phương pháp Agile cho cả lập trình và thử nghiệm rất có thể đã ở trên cùng một trang liên quan đến tái cấu trúc.

5. Tập trung vào sự tiến bộ, không phải sự hoàn hảo

Tất cả mã cuối cùng trở thành mã kế thừa đáng sợ. Chấp nhận sự thật rằng bạn sẽ không bao giờ hài lòng 100%. Mã bạn hiện đang cấu trúc lại sẽ trở nên cũ và lỗi thời trong tương lai gần và sẽ yêu cầu cấu trúc lại toàn bộ.

Bạn phải bắt đầu nghĩ đến việc tái cấu trúc như một dự án bảo trì đang diễn ra. Giống như việc bạn phải dọn dẹp và sắp xếp nhà cửa trong suốt cả tuần, bạn sẽ cần phải dọn dẹp và sắp xếp mã của mình vào nhiều dịp khác nhau.

6. Thử tự động hóa tái cấu trúc mã nguồn lại

Như với hầu hết các quy trình, càng có thể tự động hóa thì việc tái cấu trúc càng trở nên dễ dàng và nhanh chóng.

Tự động hóa một số hoặc tất cả các quy trình tái cấu trúc đang ngày càng trở nên phổ biến hơn với các nhà phát triển. Có nhiều phím tắt và công cụ giúp việc tái cấu trúc bớt đau đớn hơn. Bạn có thể học được rất nhiều thứ bằng cách đọc sách của Martin Fowler.

Bài viết này chia sẻ một số nguyên tắc và lời khuyên dành cho bạn. Hãy tái cấu trúc mã càng sớm càng tốt.

Nguồn: https://www.altexsoft.com/blog/engineering/code-refactoring-best-practices-when-and-when-not-to-do-it/

NHỮNG DẤU HIỆU MÃ XẤU

Mai Công Sơn

Bloaters

Bloaters là những đoạn code, phương thức và lớp đã tăng về kích thước mà bản thân nó trở nên khó đọc, hiểu và bảo trì. Thông thường những điều này không xảy ra ngay lập tức, mà thay vào đó chúng tích lũy theo thời gian.

1. Phương thức dài

Phương thức ngắn thì dễ đọc, dễ hiểu và dễ xử lý khi có lỗi xảy ra hơn. Hãy cố gắng chia tách những phương thức dài thành những phương thức nhỏ hơn khi bạn có thể.

2. Lớp lớn

Các lớp có lượng mã lệnh lớn thường khó để đọc và hiểu. Liệu lớp bạn xây dựng có đảm trách nhiều vai trò quá không? Nếu vậy, bạn nên tái cấu trúc lớp đó thành nhiều lớp nhỏ hơn.

3. Biến nguyên thủy

Sử dụng các biến nguyên thủy thay vì các đối tượng nhỏ cho các tác vụ đơn giản (chẳng hạn như tiền tệ, phạm vi, chuỗi đặc biệt cho số điện thoại, v.v.)

Sử dụng hằng số để mã hoá thông tin (chẳng hạn như hằng số USER_ADMIN_ROLE = 1 để đề cập đến người dùng có quyền quản trị viên.)

Sử dụng hằng số chuỗi làm tên trường để sử dụng trong mảng dữ liệu.

4. Danh sách tham số dài

Phương thức càng nhiều tham số, nó càng phức tạp. Hãy hạn chế số lượng tham số cho một phương thức, hoặc bạn cần phải sử dụng một đối tượng chứa các tham số đó.

5. Nhóm dữ liệu

Đôi khi các phần khác nhau của mã chứa các nhóm biến giống hệt nhau (chẳng hạn như các tham số để kết nối với cơ sở dữ liệu). Các khối này nên được chuyển thành các lớp riêng của chúng.

Lạm dụng hướng đối tượng

Những đoạn mã bẩn này là do triển khai không đầy đủ hoặc không chính xác các nguyên tắc của lập trình hướng đối tượng

1. Biến tạm

Các biến tạm chỉ nhận các giá trị của chúng (và do đó các đối tượng cần) trong một số trường hợp nhất định. Bên ngoài những hoàn cảnh này, chúng trống rỗng.

2. Quan hệ kế thừa không hợp lý/logic

Nếu một lớp con chỉ sử dụng một số phương thức và thuộc tính được thừa kế từ lớp cha, thì hệ thống phân cấp sẽ khác. Các phương thức không cần thiết có thể đơn giản là không sử dụng hoặc được định nghĩa lại và đưa ra các ngoại lệ.

3. Đặc tả lớp không rõ ràng

Hai lớp thực hiện các chức năng giống hệt nhau nhưng có tên phương thức khác nhau.

Thay đổi

Những smell code này có nghĩa là nếu bạn cần thay đổi điều gì đó ở một nơi trong mã của mình, bạn cũng phải thực hiện nhiều thay đổi ở những nơi khác. Do đó, việc phát triển chương trình trở nên phức tạp và tốn kém hơn nhiều.

1. Cấu trúc lớp ít có tính khả biến

Bạn thấy mình phải thay đổi nhiều phương thức không liên quan khi bạn thực hiện thay đổi đối với một lớp. Ví dụ: khi thêm một loại sản phẩm mới, bạn phải thay đổi phương pháp tìm kiếm, hiển thị và đặt hàng sản phẩm.

2. Thiết kế lớp không hợp lý và phân rã

Thực hiện bất kỳ sửa đổi nào đòi hỏi bạn phải thực hiện nhiều thay đổi nhỏ cho nhiều lớp khác nhau.

3. Hệ thống phân cấp kế thừa song song

Bất cứ khi nào bạn tạo một lớp con cho một lớp, bạn thấy mình cần tạo một lớp con cho một lớp khác.

Không cần thiết

Là một cái gì đó vô nghĩa và không cần thiết. Sự vắng mặt của nó sẽ làm cho mã sạch hơn, hiệu quả hơn và dễ hiểu hơn.

1. Conment

Một phương pháp chứa đầy các các comment, giải thích.

2. Mã trùng lặp

Hai đoạn mã trông gần như giống hệt nhau.

3. Lớp dữ liệu

Một lớp dữ liệu đề cập đến một lớp chỉ chứa các trường và các phương thức để truy cập chúng (getters và setters). Đây chỉ đơn giản là các vùng chứa dữ liệu được sử dụng bởi các lớp khác. Các lớp này không chứa bất kỳ chức năng bổ sung nào và không thể hoạt động độc lập trên dữ liệu mà chúng sở hữu.

4. Mã chết

Một biến, tham số, trường, phương thức hoặc lớp không còn được sử dụng nữa.

Có một lớp, phương thức, trường hoặc tham số không được sử dụng.

Couplers

Tất cả các mã xấu trong nhóm này góp phần vào việc ghép nối quá mức giữa các lớp hoặc cho thấy điều gì sẽ xảy ra nếu thay thế khớp nối bằng sự ủy nhiệm quá mức.

1. Feature envy

Mô tả khi một đối tượng truy cập vào các trường của đối tượng khác để thực hiện một số hoạt động, thay vì chỉ cho đối tượng biết phải làm gì.

2. Inappropriate Intimacy

Hàm của class A cần → truy cập quá nhiều data và call nhiều hàm của class B để có thể thực hiện một task nào đó → Nên để class B thực hiện task đó thông qua một hàm nào đó, class A sẽ call hàm của class B

3. Message Chain

A->xxx()->yyy()->zzz() → Không nên làm thế này, chỉ call 1 tầng thôi

4. Lớp trung gian

Nếu một lớp chỉ thực hiện một hành động, ủy thác công việc cho một lớp khác, tại sao nó lại tồn tại?

TÁI CẤU TRÚC MÃ NGUỒN:

CHUẨN HÓA MÃ

Chuyển các phương thức về lớp phù hợp hơn.

Việc di chuyển phương thức giữa các lớp là công việc diễn ra thường xuyên trong tái cấu trúc. Việc di chuyển này giúp cho các lớp có kích thước phù hợp hơn, và các lớp ít phụ thuộc vào nhau mà khả năng hợp tác giữa các lớp tốt hơn.

Các bước:

1. Kiểm tra có nên di chuyển các thuộc tính mà phương thức phải dùng
2. Kiểm tra xem có khai báo của phương thức ở lớp cha và lớp con
3. Khai báo phương thức ở lớp đích
4. Sao chép mã từ lớp nguồn tới lớp đích sao cho phương thức hoạt động được
5. Biên dịch lớp đích
6. Xác định tham chiếu phù hợp ở lớp nguồn
7. Chuyển phương thức ở nguồn gọi tới phương thức của lớp đích
8. Biên dịch và kiểm thử
9. Xác định xem có xóa phương thức ở lớp gốc và gọi trực tiếp tới phương thức ở lớp đích
10. Nếu xóa phương thức ở lớp gốc thì phải thay toàn bộ lời gọi từ phương thức này tới phương thức ở lớp mới
11. Biên dịch và kiểm thử

Ví dụ:

1. Ta có lớp Account (tài khoản). Nhưng khi nhiều loại tài khoản và mỗi loại tài khoản có cách tính tiền phí khác nhau, nên ta muôn chuyển hàm _overdraftCharge vào lớp AccountType

```
class Account...
  double overdraftCharge() {
    if (_type.isPremium()) {
      double result = 10;
      if (_daysOverdrawn &amp;amp;gt; 7) result
      += (_daysOverdrawn - 7) * 0.85;
      return result;
    }
    else return _daysOverdrawn * 1.75;
  }

  double bankCharge() {
    double result = 4.5;
    if (_daysOverdrawn &amp;amp;gt; 0) result
    += overdraftCharge();
    return result;
  }
  private AccountType _type;
  private int _daysOverdrawn;
```

2. Thuộc tính _daysOverdrawn được dùng ở phương thức là một thuộc tính là của mỗi tài khoản, nên ta không di chuyển thuộc tính.
3. Trong trường hợp này phương thức không được khai báo ở lớp cha cũng như lớp con.
4. Khai báo phương thức ở lớp đích và sao chép mã tới lớp đích sao cho phương thức hoạt động được

```
class AccountType...
  double overdraftCharge(int daysOverdrawn) {
    if (isPremium()) {
     double result = 10;
     if (daysOverdrawn &amp;amp;gt; 7) result +=
     (daysOverdrawn - 7) * 0.85;
     return result;
   }
    else return daysOverdrawn * 1.75;
  }
```

5. Chuyển phương thức ở nguồn gọi tới phương thức của lớp đích

```
class Account...
  double overdraftCharge() {
    return _type.overdraftCharge
    (_daysOverdrawn);
}
```

6. Ta xóa phương thức ở lớp gốc và gọi trực tiếp tới phương thức ở lớp đích

```
class AccountType...
  double overdraftCharge(Account account) {
    if (isPremium()) {
      double result = 10;
      if (account.getDaysOverdrawn()
      &amp;amp;gt; 7)
        result += (account.getDaysOverdrawn() -
        7) * 0.85;
      return result;
    }
    else return account.getDaysOverdrawn() *
    1.75;
  }
```

7. Biên dịch và kiểm thử

Chuyển các trường về lớp phù hợp hơn.

Nếu một trường (thuộc tính) nên được sử dụng ở một lớp khác phù hợp hơn hoặc khi ta thực hiện việc phân tách lớp thì thực hiện việc di chuyển các trường là điều cần thiết.

Các bước:

1. Nếu trường đó là public, ta phải bao gói nó
2. Biên dịch và kiểm thử
3. Tạo trường ở có getter và setter ở lớp đích
4. Biên dịch lớp đích
5. Xác định cách tham chiếu tới lớp đích
6. Xóa trường ở lớp nguồn
7. Thay thế tham chiếu tới lớp nguồn bằng lớp đích.
8. Biên dịch và kiểm thử

Ví dụ:

Ví dụ ta có lớp Account (tài khoản) và ta muốn chuyển trường _interestRate vào lớp AccountType

```
Class Account...
   private AccountType _type;
   private double _interestRate;
  double interestForAmount_days (double amount, int
days) {
       return _interestRate * amount * days / 365;
   }
```

1. Tạo trường ở có getter và setter ở lớp đích

```
class AccountType...
  private double _interestRate;
  void setInterestRate (double arg) {
    _interestRate = arg;
  }
  double getInterestRate () {
    return _interestRate;
  }
```

2. Thay thế tham chiếu tới lớp nguồn bằng lớp đích.

```
private double _interestRate;
  double interestForAmount_days (double amount,
  int days) {
    return _type.getInterestRate() * amount *
    days / 365;
  }
```

3. Biên dịch và kiểm thử

Đổi các định danh

Đổi tên phương thức, thuộc tính, lớp để không dài, đủ mô tả ý nghĩa, và phù hợp với chuẩn.

Các bước:

1. Kiểm tra xem phương thức có được triển khai ở lớp cha hay lớp con không. Nếu có thì phải thực hiện việc thay đổi này với từng triển khai này
2. Tạo phương thức với tên muốn đổi và sao chép mã của phương thức nguồn vào
3. Thay đổi các lời gọi tới phương thức cũ bằng phương thức mới
4. Xóa phương thức cũ.

Kéo lên (pull up) Chuyển một thành phần của các lớp con lên cho lớp cha.

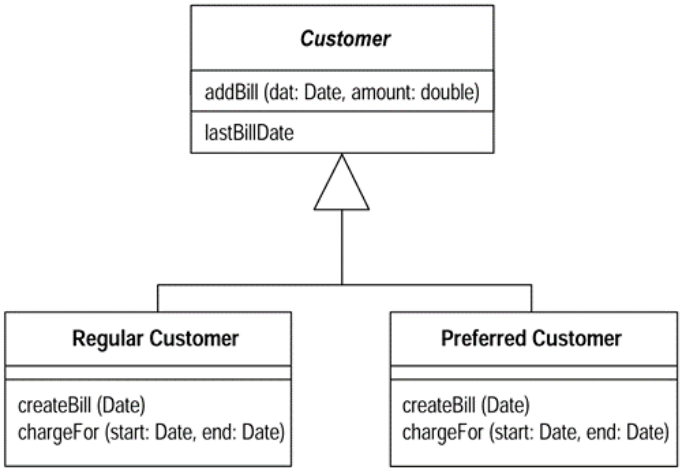
Việc xóa bảo mã lặp là một việc quan trọng. Mặc dù việc mã lặp vẫn hoạt động bình thường, nhưng sẽ khó khăn cho rất nhiều việc như khó bảo trì.

Các bước:

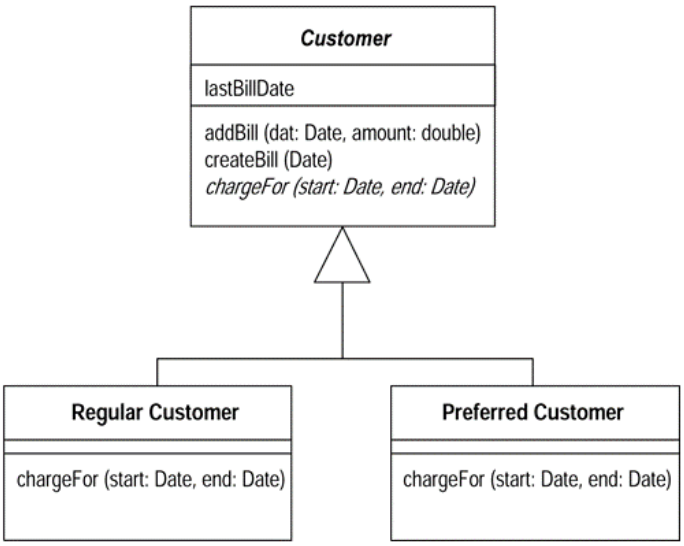
1. Kiểm tra xem các phương thức ở các lớp con có nguyên mẫu giống nhau không.
2. Nếu giống nhau thì sao phương thức từ một lớp con lên lớp cha
3. Xóa phương thức ở từng lớp con và kiểm thử xem có lỗi gì không tới khi chỉ còn phương thức ở lớp cha.

Ví dụ

1. Ví dụ ta có các lớp như Hình 6 và muốn kéo phương thức chargeFor lên lớp cha



2. Ta sẽ tạo phương thức giống hệt ở lớp con ở lớp Customer ta sẽ có như Hình 7

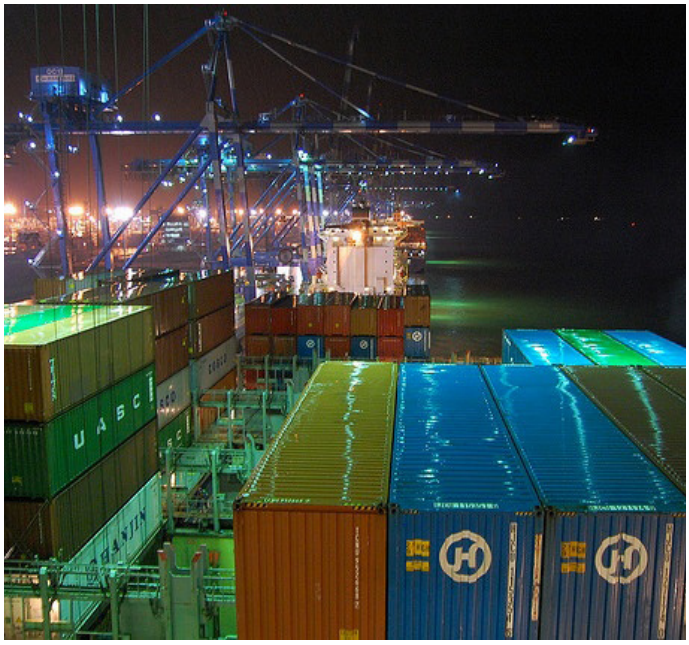


Đẩy xuống (push down)

Chuyển một thành phần của lớp cha xuống cho lớp con. Khi phương thức hoặc thuộc tính chỉ có ý nghĩa ở một lớp con cụ thể mà không phải tất cả. Khi đó thực hiện việc push down là cần thiết

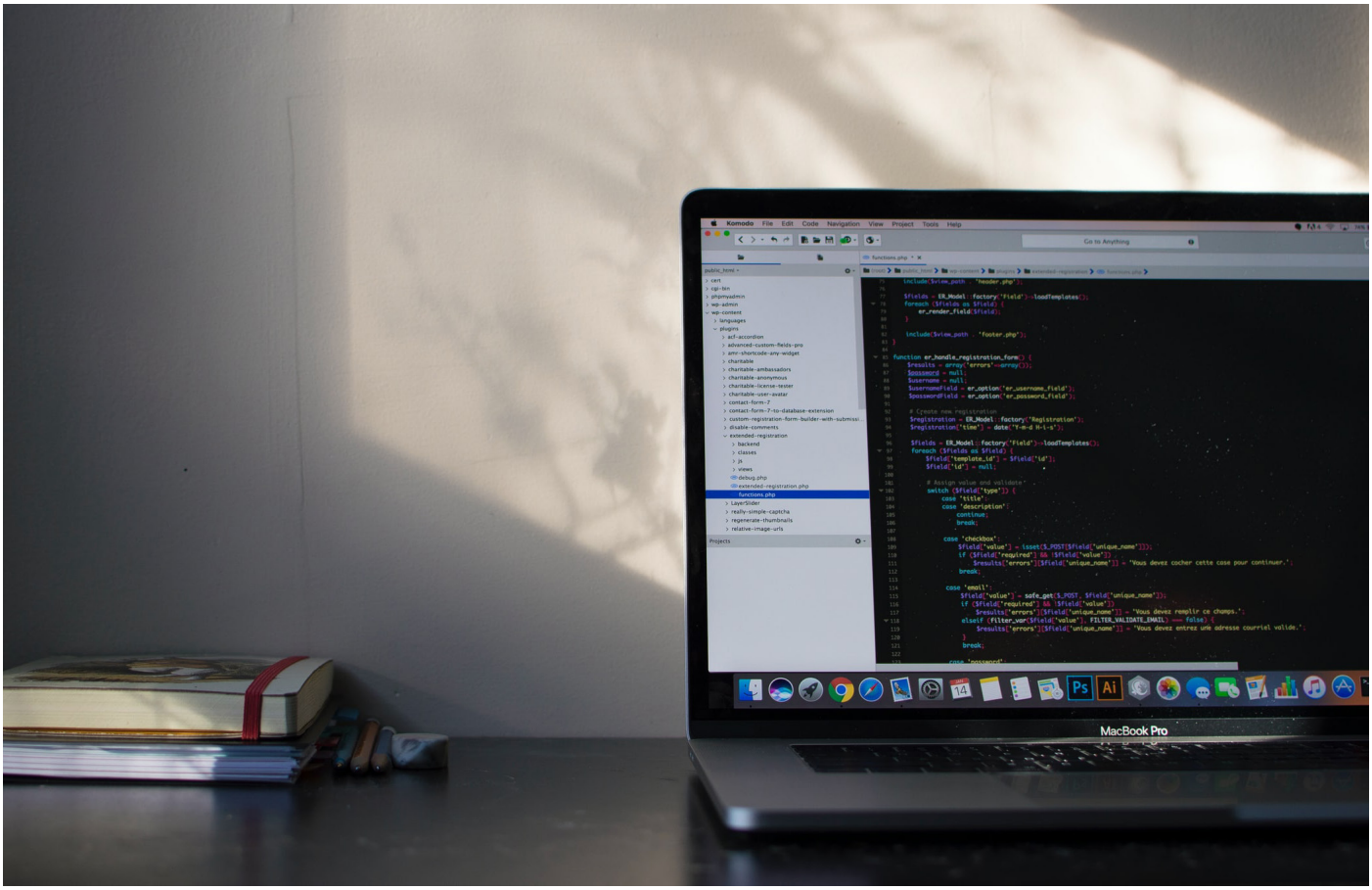
Các bước:

1. Khai báo phương thức ở tất cả các lớp con và sao chép thân từ lớp cha xuống lớp con.
2. Xóa phương thức ở lớp cha
3. Xóa các phương thức ở lớp con mà không cần thiết
4. Biên dịch và kiểm thử



TÁI CẤU TRÚC MÃ NGUỒN:

CHIA NHỎ MÃ NGUỒN



Tapchilaptrinh.vn

Tách phương thức

Tách từ một phương thức dài lấy một phương thức mới nhỏ hơn. Việc chia từ một phương thức dài thành nhiều phương thức nhỏ sẽ làm mã tốt hơn như: dễ hiểu, để bảo trì, dễ tái sử dụng hơn.

Các bước:

1. Tạo phương thức mới có tên phù hợp với chức năng.
2. Sao và dán đoạn mã muốn tách từ phương thức ban đầu vào phương thức mới.
3. Tìm tất cả các tham chiếu ở đoạn mã sao tới các biến của phương thức ban đầu. Các biến này sẽ là các biến cục bộ và tham số của phương thức mới.
4. Khai báo biên cục bộ cho tất cả các biến tạm ở đoạn mã sao.
5. Tìm tất cả các biến ở hàm gốc bị thay đổi giá trị ở đoạn mã sao. Nếu chỉ có một bị thay đổi thì có thể truyền giá trị đó vào bằng đối số và gán cho giá trị tương ứng. Nhưng nếu có nhiều hơn thì phải chú ý!
6. Truyền tất cả mọi biến được tham chiếu chỉ để đọc ở mã được sao vào phương thức mới như tham số.
7. Biên dịch để kiểm tra xem mọi biến cục bộ đã được xử lý.
8. Thay đoạn mã đã sao bằng phương thức mới.
9. Biên dịch và kiểm thử.

Ví dụ

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    // print banner
    System.out.println
    (“**************************”);
    System.out.println
    (“***** Customer Owes ******”);
    System.out.println
    (“**************************”);

    // calculate outstanding
    while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      outstanding += each.getAmount();
    }

    //print details
    System.out.println (“name:” + _name);
    System.out.println (“amount” + outstanding);
}
```

Dễ dàng tách đoạn mã hiển thị tiêu đề bằng các cắt và dãn

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      outstanding += each.getAmount();
    }
    //print details
    System.out.println (“name:” + _name);
    System.out.println (“amount” + outstanding);
}

void printBanner() {
    // print banner
    System.out.println
    (“**************************”);
    System.out.println
    (“***** Customer Owes ******”);
    System.out.println
    (“**************************”);
}
```

Ví dụ có biến cục bộ chỉ để đọc: Trong trường hợp này ta đơn giản là truyền chúng theo tham số. Ở ví dụ trên ta có thể tách phương thức để in ra thông tin chi tiết từ phương thức printOwning

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    printBanner();
    // calculate outstanding
    while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      outstanding += each.getAmount();
    }
    printDetails(outstanding);
}
void printDetails (double outstanding) {
    System.out.println (“name:” + _name);
    System.out.println (“amount” + outstanding);
}
```

Và bạn có thể truyền vào số lượng biến cục bộ tùy thích.

Ví dụ có gán giá trị cho biến cục bộ:

Trong trường hợp này chúng ta chỉ đề cập tới trường hợp biến cục bộ của hàm gốc, nếu là có thay đổi giá trị cho tham số thì xem kỹ thuật tái cấu trúc xóa gán cho tham số.

Từ phương thức printOwing có ở trên:

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();
    // calculate outstanding
    while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

Ta có thể tách thành:

```
void printOwing() {
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding);
}
double getOutstanding() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      outstanding += each.getAmount();
    }
    return outstanding;
}
```

Tách lớp

Tách một phần của lớp đã tồn tại thành một lớp mới. Một lớp có kích thước tăng dần và tới trở nên khó hiểu và khó bảo trì. Lúc đó ta phải tách nhỏ lớp đó ra.

Các bước:

1. Cách chia trách nhiệm của các lớp
2. Tạo lớp mới để chia sẻ trách nhiệm với lớp ban đầu
3. Tạo một liên kết từ lớp ban đầu tới lớp mới
4. Thực hiện viên di chuyển từng trường và phương thức từ lớp cũ sang lớp mới
5. Biên dịch và kiểm thử.

Ví dụ:Ta phải tách một lớp đơn giản:

```
class Person...
    public String getName() {
      return _name;
    }
    public String getTelephoneNumber() {
      return _officeAreaCode + " " + _officeNumber;
    }
    String getOfficeAreaCode() {
      return _officeAreaCode;
    }
    void setOfficeAreaCode(String arg) {
      _officeAreaCode = arg;
    }
    String getOfficeNumber() {
      return _officeNumber;
    }
    void setOfficeNumber(String arg) {
      _officeNumber = arg;
    }

    private String _name;
    private String _officeAreaCode;
    private String _officeNumber;
```

Cách chia trách nhiệm của các lớp: ta muốn chia thành một lớp chứa thông tin về số điện thoại.

Tạo lớp mới để chia sẻ trách nhiệm với lớp ban đầu

```
class TelephoneNumber {}
```

Tạo một liên kết từ lớp Person tới lớp TelephoneNumber

```
class Person
    private TelephoneNumber _officeTelephone = new
    TelephoneNumber();
```

Thực hiện viên di chuyển từng trường và phương thức từ lớp cũ sang lớp mới

```
class Person...
    public String getName() {
      return _name;
    }
    public String getTelephoneNumber(){
      return _officeTelephone.getTelephoneNumber();
    }
    TelephoneNumber getOfficeTelephone() {
      return _officeTelephone;
    }

    private String _name;
    private TelephoneNumber _officeTelephone = new
    TelephoneNumber();
  class TelephoneNumber...
    public String getTelephoneNumber() {
      return ("(" + _areaCode + ") " + _number);
    }
    String getAreaCode() {
      return _areaCode;
    }
    void setAreaCode(String arg) {
      _areaCode = arg;
    }
    String getNumber() {
      return _number;
    }
    void setNumber(String arg) {
      _number = arg;
    }
    private String _number;
    private String _areaCode;
```

Biên dịch và kiểm thử.

TÁI CẤU TRÚC MÃ NGUỒN:

TRỪU TƯỢNG HÓA

Tapchilaptrinh.vn

Bao gói các trường của lớp để ép người dùng phải dùng thông các cách truy xuất dán tiếp như các phương thức.

Việc bao gói dữ liệu giúp tăng tính an toàn của dữ liệu, che dấu dữ liệu, giúp cho lớp con có thể dễ dàng ghi đè các lấy dữ liệu.

Các bước:

1. Tạo phương thức setter/getter cho dữ liệu đó.
2. Thay tất cả các tham chiếu bằng getter/ setter tương ứng.
3. Chuyển dữ liệu đó thành private (riêng tư).
4. Kiểm tra lại lần cuối xem còn có tham chiếu nào không.
5. Biên dịch và kiểm thử lại lần cuối.

```
private int low, height;

public boolean isIn(int arg){
    return (low&amp;gt;=arg)&amp;amp;&amp;amp;amp;
    (arg&amp;lt;=height);
}
```

1. Mã chưa được bao gói

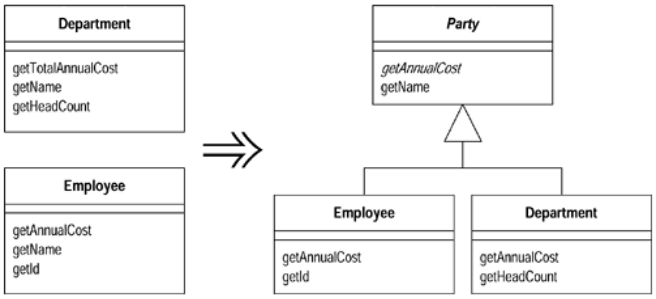
```
private int low, height;

public boolean isIn(int arg){
    return (getLow()&amp;gt;=arg)&amp;amp;&amp;amp;amp;
    (arg&amp;lt;=getHeight());
}

public int getHeight() {
    return height;
}

public int getLow() {
    return low;
}
```

2. Mã đã được bao gói

Tạo một kiểu chung cho những kiểu có chung mã.

Từ những kiểu cụ thể, tạo ra một kiểu chung chứa thuộc tính và phương thức chung cho tất cả các kiểu cụ thể. Mã bị lặp là nguyên nhân chính làm ta phải tiến hành việc tái cấu trúc này.

Việc tái cấu trúc này làm chúng ta phải viết ít mã hơn, nhiều mã được chia sẻ hơn, nên việc bảo trì, đọc cũng dễ dàng hơn.



3. Ví dụ tách lớp cha

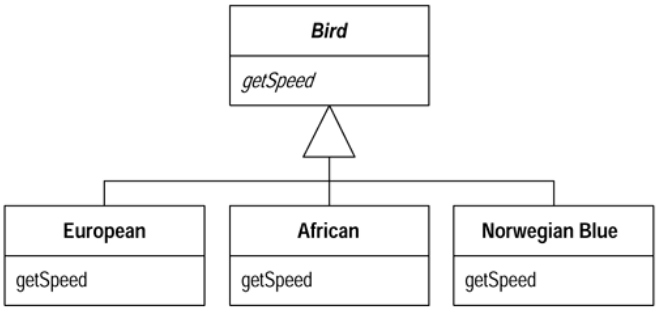
Thay các câu lệnh điều kiện bằng các sử dụng tính đa hình.

Khi bạn có một câu lệnh điều kiện phụ thuộc vào loại của đối tượng. Vấn đề lớn nhất của câu lệnh điều kiện này là xuất hiện ở nhiều nơi, và mỗi lần mà bạn muốn thêm một loại mới vào thì bản phải cập nhật ở tất cả những nơi này. Khi đó tính đa hình của lập trình hướng đối tượng sẽ giúp bạn.

Cách thức tiến hành việc tái cấu trúc: Tạo một phương thức trừu tượng ở lớp cha và chuyển khối lệnh ở lệnh điều kiện vào phương thức ghi đè ở mỗi lớp con.

```
double getSpeed() {
  switch (type) {
    case EUROPEAN:
      return getBaseSpeed();
    case AFRICAN:
      return getBaseSpeed() - getLoadFactor() *
        _numberOfCoconuts;
    case NORWEGIAN_BLUE:
      return (_isNailed) ? 0 : getBaseSpeed(_volt-
      age);
  }
  throw new RuntimeException
  (“Should be unreachable”);
}
```

4. Mã dùng lệnh điều kiện



5. Thiết kế sau có dùng tính đa hình