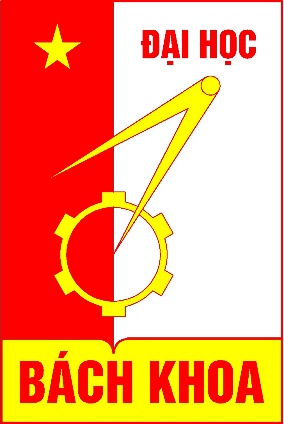
**TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI**

**VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG**

──────────────



**BÀI TẬP LỚN**

**NHẬP MÔN CÔNG NGHỆ PHẦN MỀM**

***Đề bài: Tìm hiểu về Refactoring***

|  |  |  |
| --- | --- | --- |
| ***Giáo viên hướng dẫn*** | **:** | **ThS. Nguyễn Mạnh Tuấn** |
| ***Sinh viên thực hiện*** | **:** | **Vũ Văn Cảnh** |
| ***MSSV*** | **:** | **201810406** |
| ***Lớp*** | **:** | **KS2 – CNTT – Khóa 2** |

**MỤC LỤC**

1. **Định nghĩa Refactoring**
2. **Các vấn đề khi Refactor**
3. **Một số vấn đề thường gặp ngăn cản việc Refactoring code**
4. **Nhận biết mã nguồn Refactor**
5. **Các phương pháp Refactor thường dùng**
6. **Tìm hiểu về một công cụ Refactor**
7. **Định nghĩa Refactoring**

* Khi còn là 1 sinh viên và ra trường thậm chí được vài năm. Vấn đề không những mình mà có thể là nhiều bạn khác quan tâm đó chính là làm sao code chạy được và không bị lỗi. Nhưng sau một thời mình nhận ra 1 điều đơn giản rằng Code thì không bao giờ là không bị lỗi, và việc làm sao để phát hiện được lỗi và fix nó 1 cách nhanh chóng thực sự không phải là điều đơn giản. Và một điều nữa là ban đầu code có thể chạy được nhưng khi chương trình phình to lên có thể đoạn code đó không thể nào chạy được nữa đơn giản là đoạn code đó không thể làm đáp ứng được khả năng mở rộng cho các feature mới hoặc bị giảm hiệu năng… Và như một điều tất yếu đoạn code đó bị đập đi. Và xây dựng lại đoạn code khác. Thật mất thời gian và công sức, đôi khi còn có thể bị nghi ngờ về khả năng và kỷ luật trong làm việc. Khi đã trải qua các vấn đề trên bản thân người lập trình mới bắt đầu nhìn nhận lại tất cả. Từ cách tiếp cận vấn đề khi mới bắt tay vào code, cho đến lúc implement các module, fix bug, thêm tính năng và thậm chí là ngay cả cách đặt tên biến như thế nào...Để từ đó có thể dễ dàng mở rộng code, maintain hệ thống hoặc đơn giản là xác định nhanh chóng lỗi ở đâu để sửa lỗi.
* Định nghĩa: Refactoring là một kỹ thuật có kỷ luật để tái cấu trúc một bộ mã hiện có, thay đổi cấu trúc bên trong của nó mà không thay đổi hành vi bên ngoài của nó.
* Cốt lõi của nó là một chuỗi các hành vi nhỏ bảo tồn các biến đổi. Mỗi phép biến đổi (được gọi là "tái cấu trúc") thực hiện rất ít, nhưng một chuỗi các phép biến đổi này có thể tạo ra sự tái cấu trúc đáng kể. Vì mỗi lần tái cấu trúc là nhỏ, nên nó ít có khả năng sai. Hệ thống được duy trì hoạt động hoàn toàn sau mỗi lần tái cấu trúc, làm giảm khả năng hệ thống có thể bị hỏng nghiêm trọng trong quá trình tái cấu trúc.
* Tại sao phải Refactoring code:

+ Refactor có thể cải thiện được thiết kế hệ thống của bạn

+ Refactor giúp chương trình dễ hiểu hơn(dễ đọc, dễ bảo trì...)

+ Refactor giúp dễ tìm lỗi hơn.

+ Refactor giúp bạn code nhanh hơn.

1. **Các vấn đề khi Refactor**

* Một số vấn đề gặp phải khi Refactoring code:

1. **Không sử dụng Refactoring tool**

* Rất thường là nguyên nhân phổ biến nhất của lỗi tái cấu trúc là do thực hiện thủ công và không phải do mã, do không có sẵn công cụ thích hợp. Việc tái cấu trúc như thay đổi signature của phương thức, đổi tên biến và phương thức, di chuyển biến giữa các lớp, v.v ... có thể dẫn đến các lỗi không xác định và gây hại cho toàn bộ mã. Ví dụ, việc thay đổi tên của một biến cục bộ có thể nghe có vẻ là một nhiệm vụ đơn giản, nhưng nếu có một biến cấp độ lớp tồn tại có cùng tên thì tên mới sẽ ẩn nó. Mặt khác, các công cụ tái cấu trúc có tính đến tất cả các cảnh báo này, trong khi thực hiện thủ công có thể gây hiểu nhầm. Do đó, giải pháp là bắt đầu tái cấu trúc của bạn với một công cụ phù hợp.

1. **Refactoring không kèm theo Unit tests**

* Unit test tạo thành một thực tiễn cần thiết của quá trình tái cấu trúc và nếu không được thực hiện sau khi tái cấu trúc có thể dẫn đến nguy cơ rõ ràng là đưa ra các khiếm khuyết không bị phát hiện trong mã tái cấu trúc. Do đó, xác nhận cho các thay đổi là một nhiệm vụ khó khăn. Tuy nhiên, một xem xét khác mà các nhà phát triển thường có thể là bình thường là mã kế thừa có thể đã có một lỗi không được phát hiện mà không được biết cho đến khi quá trình tái cấu trúc được thực hiện. phản ứng của nhà phát triển sẽ được giả định rằng những thay đổi được giới thiệu trong các mã, có thể tôi đã giới thiệu các lỗi.
* Điều đó sẽ dẫn đến một tìm kiếm thông qua các thay đổi mới nhất được thực hiện để tìm lỗi, mặc dù vấn đề thực sự nằm ở mã vẫn chưa được xử lý.
* Vì vậy, tùy chọn tốt nhất là trước tiên tạo một tập các thử nghiệm đơn vị cho mã hiện có để nếu có bất kỳ lỗi nào trước khi tái cấu trúc, chúng có thể được phát hiện trước khi kết hợp bất kỳ thay đổi nào. Sự phát triển theo hướng thử nghiệm cũng đạt được động lực dựa trên cùng một hiệu nguyên tắc rằng bất cứ khi nào bổ sung mới được thực hiện cho mã, các nhà phát triển phải thiết kế các trường hợp thử nghiệm cho các yêu cầu và mã được cho là vượt qua nó và được tái cấu trúc cho một thiết kế tốt hơn. Đôi khi TDD được sử dụng để cải thiện thiết kế hoặc gỡ lỗi mã kế thừa.

1. **Refactoring code bị ràng buộc với nhiều Interfaces**

* Code tương tác với nhiều giao diện bên ngoài có thể phức tạp và dễ bị lỗi khi tái cấu trúc, đặc biệt đối với các tham số chuỗi được gõ lỏng lẻo hoặc chuỗi. Ví dụ, tái cấu trúc hệ thống có thể dẫn đến nhiều thay đổi và có thể vô tình tạo ra một số defects. Một dữ liệu cụ thể mà hệ thống máy thu bất ngờ khi được gửi bởi hệ thống bên ngoài đã hoạt động sớm hơn nhưng do bất ngờ bởi người nhận có thể không hoạt động sau khi thay đổi. Những lỗi này rất khó phát hiện và tìm thấy chúng trong chu kỳ phát triển gần như không thể. Chúng chỉ có thể được phát hiện trong một chu kỳ QA đầy đủ hoặc trong quá trình sản xuất.
* Giải pháp tối ưu cho loại vấn đề này là ghi lại tất cả các bộ dữ liệu được truyền giữa đoạn mã bên trong và các giao diện (interface) bên ngoài trong một khoảng thời gian xác định. Sau đó thực hiện tái cấu trúc hệ thống và kiểm tra thiết kế cho hệ thống bằng cách sử dụng dữ liệu đã chụp. Kết quả kiểm tra sẽ thiết lập nếu hệ thống có khả năng xử lý lưu lượng trực tiếp chính xác hay không.

1. **Thay đổi signature của phương thức**

* Tái cấu trúc signature phương thức thay đổi liên quan đến việc thay đổi tên phương thức, thay đổi tên tham số hoặc kiểu trả về, thêm hoặc xóa tham số mới hoặc sắp xếp lại các tham số, thay đổi phạm vi hiển thị, v.v.
* Nhưng việc thao tác signature phương thức có thể dẫn đến các lỗi mã không thể nhận dạng dễ dàng, đặc biệt là đối với các phương thức có tất cả các tham số cùng một lúc. Ví dụ: một phương thức có hai chuỗi là tham số, nhưng vì một số lý do hợp lý, nhà phát triển thay đổi thứ tự của các tham số, vì cả hai đều cùng loại, tình huống này có thể tạo ra sự mơ hồ vì bất kỳ nhà phát triển nào khác cũng không thể nhận ra sự thay đổi và bởi bất kỳ cơ hội nếu nhà phát triển quên thay đổi thứ tự trong các cuộc gọi phương thức, hệ thống sẽ không dự đoán đó là lỗi.
* Do đó, để tránh bất kỳ sự mơ hồ nào, tốt nhất là phải tỉ mỉ trong đó và thao tác signature phương thức, tốt nhất nên được tự động hóa để tránh mọi rủi ro như vậy. Một trong những giải pháp hiện được đề xuất để đảm bảo tái cấu trúc an toàn hơn là mã hóa các điều kiện tiên quyết trong phân tích động. Mặc dù phân tích tĩnh có thể kiểm tra hiệu quả các điều kiện tiên quyết cú pháp, nhưng kiểm tra các điều kiện tiên quyết ngữ nghĩa là một nhiệm vụ tương đối phức tạp trong các ngôn ngữ như Java, nơi phân tích tĩnh khó có thể phân tích đường dẫn thực thi nào sẽ được theo dõi.
* Trong cài đặt thử nghiệm đã chỉ ra rằng các nhà phát triển có thể hiểu biết thấp về ý nghĩa của tái cấu trúc và các điều kiện tiên quyết đặt ra trên mã nguồn, và đôi khi các cảnh báo hoặc xem trước có thể bị bỏ qua, dẫn đến tái cấu trúc không an toàn.

1. **Vấn đề khi Extracting Interfaces và class Base**

* Cải thiện đoạn mã trong khi tái cấu trúc thường liên quan đến việc tạo Interfaces mới cho lớp nhưng quy trình thủ công khá mệt mỏi vì việc viết signature phương thức và các thuộc tính khác có thể là nguyên nhân phổ biến của lỗi và lỗi mã, có sẵn tự động hóa phù hợp. Nhưng sẽ là một ý tưởng tuyệt vời nếu Interfaces được trích xuất sẽ quét qua các lớp khác và nhắc người dùng về các lớp có khả năng thực hiện cùng một giao diện, nó sẽ tiết kiệm rất nhiều nỗ lực thủ công.
* Một mối quan tâm khác là trong khi trích xuất một super class từ một lớp rất hữu ích bằng cách tạo một lớp cơ sở bằng cách trích xuất đoạn mã phổ biến nhất. Nhưng trong Java đôi khi vấn đề có thể phát sinh là mặc dù chúng ta đã tạo một lớp cơ sở từ lớp rất hữu ích này, chúng ta muốn lớp này được sử dụng lại trong lớp cơ sở khác nhưng điều đó sẽ không thể xảy ra vì lớp này đã có lớp cha và nhiều kế thừa không được phép trong Java.

1. **Refactor với độc lập ngôn ngữ**

* Vấn đề chính khi tái cấu trúc mã code được đề xuất là mô hình hoặc phụ thuộc ngôn ngữ. Nhưng có một số phần mềm khác nhau cấu thành mã thuộc hai mô hình khác nhau, chẳng hạn như mã AspectJ được nhúng trong Java. Có các phép tái cấu trúc được thực hiện để sửa đổi mã từ OOP sang AOP hoặc từ ngôn ngữ thủ tục sang AOP (như C đến AspectC hoặc Objective C). Do đó, một cân nhắc quan trọng với công cụ tái cấu trúc cho phép công cụ sẽ là hỗ trợ tái cấu trúc mô hình chéo. Nghiên cứu được thực hiện trong lĩnh vực này hoạt động trong việc thực hiện các phép tái cấu trúc không dành riêng cho ngôn ngữ hoặc mô hình và có thể được áp dụng mà không cần bận tâm về tính tương thích.

1. **Refactor lại thiết kế có thể không phải là một sự lựa chọn tốt**

* Tái cấu trúc có thể được sử dụng tốt trong việc cải thiện thiết kế mã nhưng để tin tưởng một cách ngây thơ rằng nó có thể thay đổi hoàn toàn thiết kế và những sai lầm sâu xa và sẽ tạo ra một thiết kế trả trước sẽ là một giả định quá mức. Kể từ khi nhận ra các khiếm khuyết và khoảng trống mất nhiều thời gian trong thế giới thực, khi người ta bắt đầu tái cấu trúc mùi thực tế, rất nhiều khiếm khuyết đã chất đống sau khi áp dụng các bản vá khác nhau và sửa chữa liên tục, mã phát triển thành một hình dạng khác nhau. Đó có lẽ là khi nhà phát triển nhận ra rằng phần mềm cần một kiến trúc khác hoặc các bộ trừu tượng khác nhau.

1. **Một số vấn đề thường gặp ngăn cản việc Refactoring code**

* Trình độ kém: do khi nhóm phát triển không có hiểu biết sâu sắc về OOP thì đương nhiên những đoạn code được viết ra ban đầu sẽ là những đoạn code có chất lượng "dở", nhưng quan trọng là họ không hề biết về sự tồn tại của những điều "dở" đó. Nếu không thực hiện code refactoring thì sẽ gây nguy hại về lâu về dài cho chính sản phẩm của họ bởi nhóm phát triển sẽ vẫn mãi duy trì năng lực hiện có.
* Việc chấp nhận những đoạn code chất lượng kém: Sau một thời gian dài, nhóm phát triển nhận ra có nhiều đoạn code “dở” nhưng vì lúc này khối lượng code của chương trình đã rất lớn và có chứa nhiều đoạn code “dở” đó nên họ mang tư tưởng “sống chung vỡ lũ” hoặc nghĩ đến việc thiết kế viết lại toàn bộ hệ thống.
* Không có thời gian: nghe có vẻ đây là một lý do khá xác đáng, bởi khách hàng hoàn toàn không nhận được lợi ích trực tiếp từ việc refactoring nên sẽ khó thuyết phục để họ trả tiền cho nhóm phát triển trực tiếp refactor lại những đoạn mã. Và điều này sẽ làm này sinh nghi ngờ từ khách hàng rằng năng lực hoặc thái độ làm việc của nhóm phát triển kém đi.
* Vậy thì khi nào cần Refactor?

+ Khi thêm một Feature/function mới cho hệ thống.

+ Khi thực hiện sửa, vá lỗi.

+ Khi thực hiện review code

+ Khi thực hiện Unit test.

1. **Nhận biết mã nguồn Refactor**

* Một mã nguồn đã được Refactor là mã nguồn dễ đọc hơn ban đầu, được tổ chức lại một cách khoa học và có thể được kiến trúc lại tốt hơn.
* Việc này giống như là chúng ta sắp đặt lại hệ thống điện trong nhà theo một cách khoa học hơn nhưng vẫn đảm bảo y nguyên chức năng của những ổ cắm hay công tắc trên tường.
* Việc sắp xếp lại các đoạn mã một cách hợp lý nhưng vẫn giữ nguyên được chức năng giúp chúng ta có thể thêm vào đó các chức năng mới mà khi đó có thể khối lượng code có thể tăng lên nhưng không tăng thêm độ phức tạp của các đoạn mã. Giúp chúng ta dễ dàng bảo trì sản phẩm cũng như là nhanh chóng tìm ra lỗi khi hệ thống có vấn đề.
* Một số đoạn mã thường gặp và cần phải Refactor:

+ Method, functions quá dài: việc viết code update liên tục vào một function, methods khiến cho lượng code trong methods ngày càng lớn. Mục đích của method có khi không còn giống như lúc đầu nữa nên việc đọc lại ngày càng khó khăn và tốn thời gian hơn.

void printPet() {

printBackground();

//print details

System.out.println("name: " + name);

System.out.println("legs: " + getNumOfLeg());

}

Sau khi Refactor, tạo thêm một method nữa:

void printPet() {

printBackground();

printDetails(getNumOfLeg());

}

void printDetails(int legs) {

System.out.println("name: " + name);

System.out.println("legs: " + legs);

}

+ Quá nhiều parameters trong method, functions: có quá nhiều parameter trong method sẽ dễ gây nhầm lẫn, khó nhớ, khó hiểu khi đọc lại code.

+ Class quá lớn: một class có quá nhiều properties, methods trở nên khó hiểu, khó thay đổi. Nên cân nhắc chia class lớn đó thành nhiều class nhỏ hơn để dễ dàng quản lý cũng như sắp xếp các đoạn mã cho hợp lý, thực hiện đúng chức năng.

Class house{

int table;

int freeze;

int keyboard;

int monitor;

....

}

Class Computer{

int keyboard;

int monitor;

...

}

Class house{

int table;

int freeze;

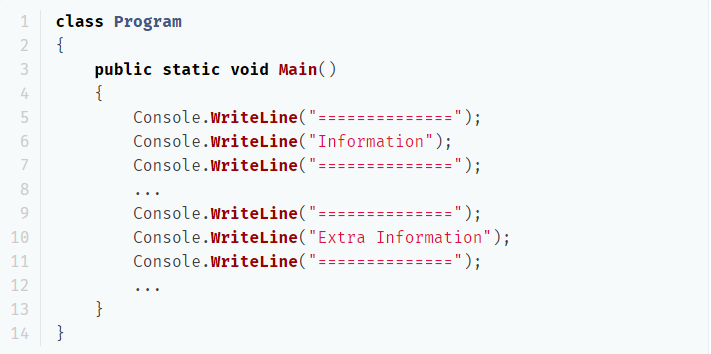
Computer computer;

....

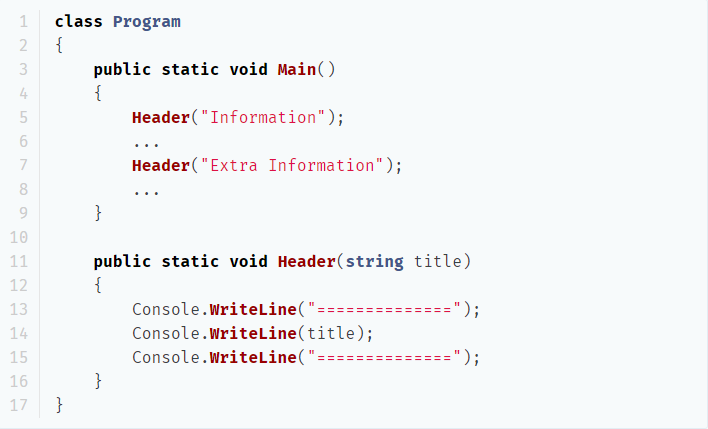
}

1. **Các phương pháp Refactor thường dùng**
2. Tách Method

* Kỹ thuật refactor thường thấy nhất đó là “Tách method” (Extract method). Kỹ thuật này đơn giản chỉ là tìm một đoạn code dùng nhiều lần ở nhiều nơi, tách nó ra và cho vào một method riêng. Sau đó, tại vị trí cũ, ta gọi method vừa mới tạo.



Sau khi Refactor

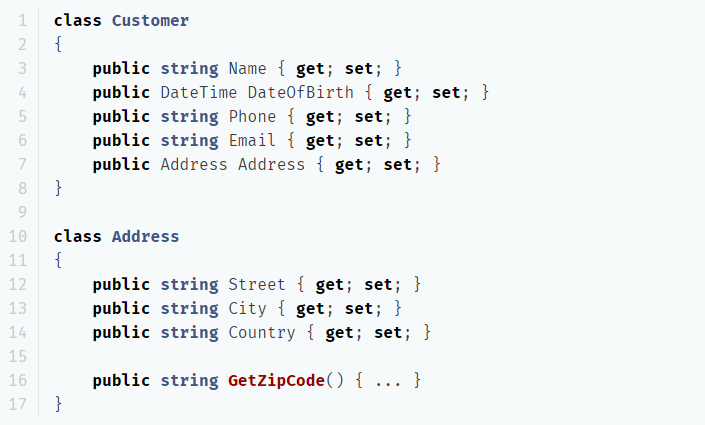


1. Tách Class

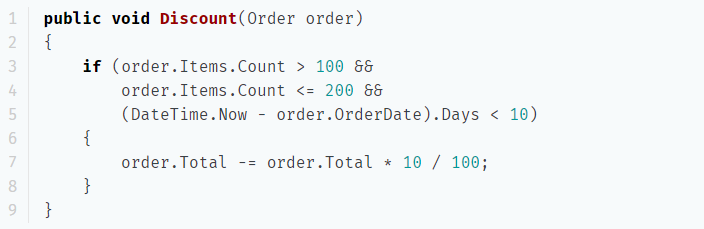
* Tách class là kỹ thuật refactor được áp dụng cho những class lớn. Trong lập trình hướng đối tượng, dữ liệu và phương thức có liên quan sẽ được gom thành một class. Tuy nhiên, khi thiết kế, đôi lúc ta thêm nhiều chức năng không thuộc class đó. Đây là lúc nên áp dụng kỹ thuật tách class.



Sau khi refactor



1. Đơn giản hóa biểu thức điều kiện

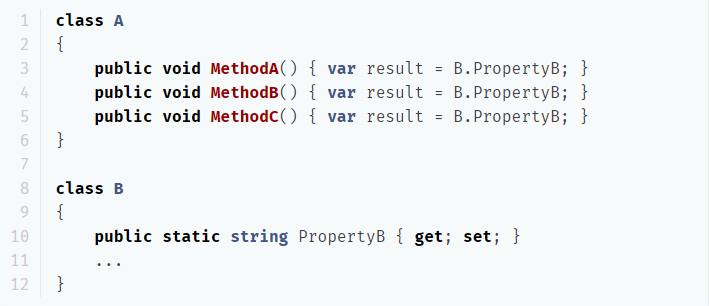


* Biểu thức điều kiện trong câu lệnh if quá phức tạp do phải xác thực nhiều dữ kiện khác nhau. Do đó, để dễ dàng cho việc bảo trì, ta nên đơn giản hóa bằng cách tách nó ra thành một method riêng.

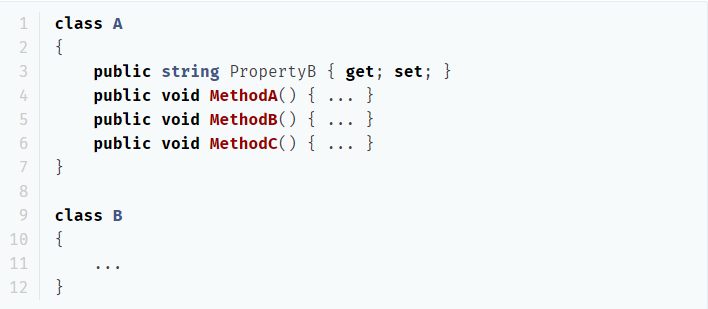
Sau khi refactor



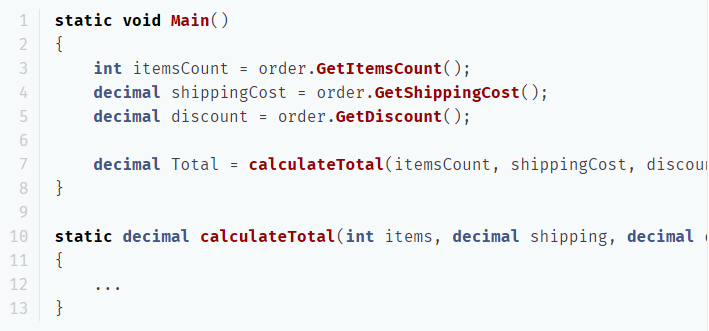
1. Di chuyển dữ liệu



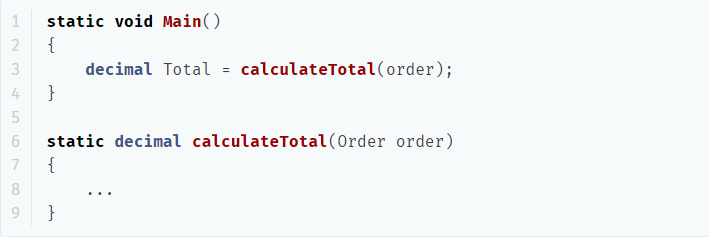
Sau khi refactor



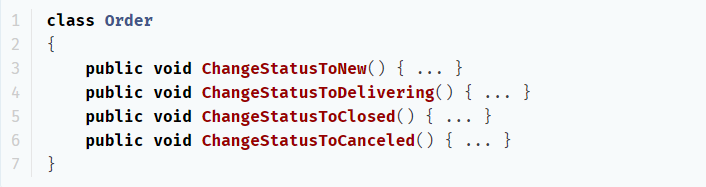
1. Tạo đối tượng cho danh sách tham số



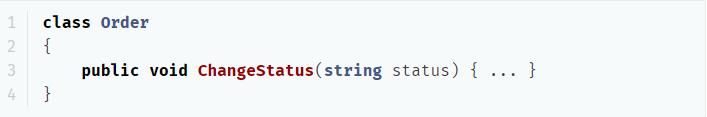
Sau khi refactor

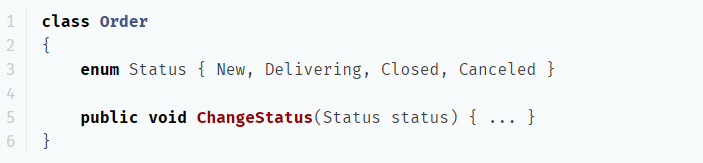


1. Tham số hóa Method



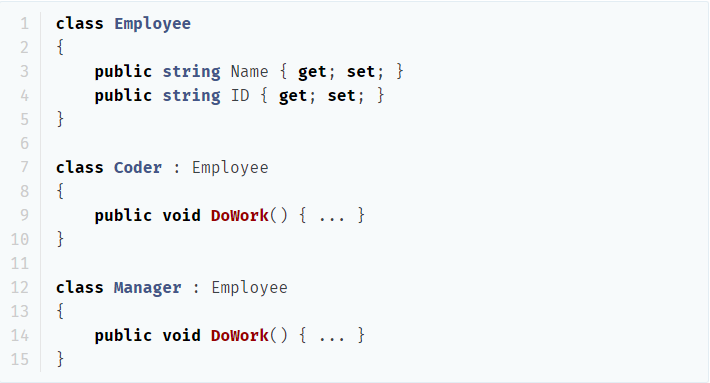
Sau khi refactor



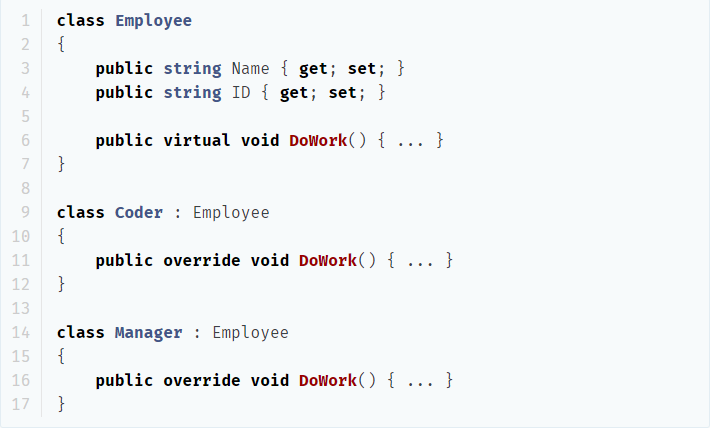


1. Kéo method và dữ liệu lên lớp cha

* Việc kéo method và dữ liệu lên lớp cha có liên quan tới tính chất kế thừa trong lập trình hướng đối tượng. Để biết nên kéo thành phần nào lên lớp cha, ta xem trong các lớp con có thành phần nào giống nhau hay không. Nếu có thì nó chính là thứ mà ta cần phải kéo lên lớp cha.



Sau khi refactor



1. Đẩy method và dữ liệu xuống lớp con

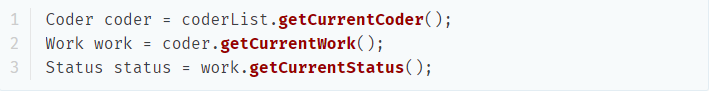
* Cũng giống như phương pháp ở trên, nhưng chỉ khác chiều di chuyển. Thay vì kéo chúng lên, ta sẽ đẩy chúng từ lớp cha xuống lớp con. Vậy làm sao để biết những thành phần nào nên đẩy xuống? Rất đơn giản, bạn hãy tìm những thành phần mà chỉ hữu ích cho một lớp con mà không được dùng trong những lớp con khác.



Sau khi refactor



1. Chuỗi gọi method



Sau khi refactor



1. Một chức năng, một method

* Trong lập trình, nguyên tắc separation of concerns luôn là một thói quen tốt mà ta nên tuân theo.



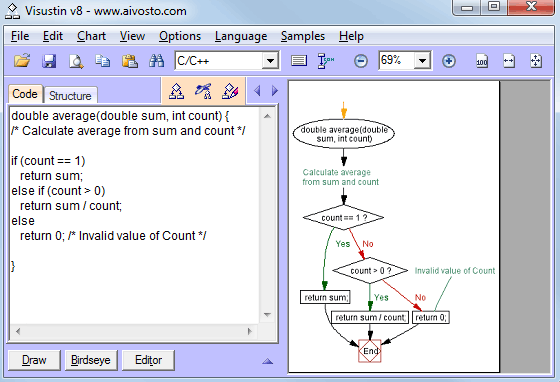
Sau khi refactor



1. **Tìm hiểu về một công cụ Refactor**
2. **Visustin – Refactoring với flow charts**

* Công cụ này hiển thị biểu đồ luồng cho một đoạn mã thực thi nhất định, có thể là Visual Basic, C/C++, C#, Java, Pascal, Cobol hoặc một số ngôn ngữ lập trình khác.
* Lỗi logic dễ dàng xuất hiện trong mã có chứa các điều kiện, vòng lặp và bước nhảy lồng nhau. Biểu đồ luồng hiển thị logic chương trình theo cách dễ hiểu hơn, không bị ràng buộc với thứ tự mà mã cơ sở được viết. Bằng cách đọc mã nguồn và biểu đồ luồng cạnh nhau, bạn có thể tìm thấy các lỗ hổng logic có thể bị phát hiện bằng cách nhìn vào mã chỉ có.
* Bên cạnh việc phát hiện lỗi, biểu đồ luồng giúp tìm cách tái cấu trúc mã theo cách có khả năng dễ đọc hơn hoặc tối ưu hóa hơn. Giải thích trực quan về mã có thể tiết lộ các mẫu không dễ nhìn thấy trong nguồn.
* Một thủ tục dài có thể chứa cùng một dòng mã ở một số vị trí. Có thể bạn có thể thay đổi logic để loại bỏ các dòng trùng lặp.
* Một cấu trúc logic có thể được nhân đôi trong hai hoặc nhiều thủ tục (có thể thông qua sao chép và dán mã hóa). Khi được phát hiện, logic này tốt nhất được chuyển sang một chức năng mới và được gọi từ các chức năng khác.
* Một thuật toán phức tạp thường là tốt nhất để phân chia trong một số chức năng. Biểu đồ luồng hữu ích cho việc tìm các khối từ đó hình thành các hàm mới.
* Decompose có điều kiện là một kỹ thuật tái cấu trúc trong đó một biểu thức điều kiện phức tạp được loại bỏ (và viết lại dưới dạng hàm). Biểu thức phức tạp rất dễ phát hiện trong biểu đồ flow charts. Cũng có thể viết lại logic theo cách đơn giản hơn.
* Điều kiện ngược có nghĩa là đảo ngược logic của câu lệnh có điều kiện bằng cách sử dụng toán tử Not. Trong nhiều trường hợp, loại bỏ Không phải là cách dễ đọc nhất. Biểu đồ flow charts thường tiết lộ việc sử dụng logic đảo ngược khá đơn giản để tái cấu trúc.

1. **Áp dụng Visustin trong một ví dụ đơn giản**



Hoặc ví dụ ta có một đoạn mã như sau:

void selectionSort(int data[], int count)

// selection sort function module in C

// Source: http://en.wikipedia.org/wiki/Selection\_sort

// This code is licensed under the GNU Free Documentation License.

// It is from the Wikipedia article "Selection sort" dated 2006-11-07.

{

int i, j, min, temp;

for (i = 0; i < count - 1; i++) {

/\* find the minimum \*/

min = i;

for (j = i+1; j < count; j++) {

if (data[j] < data[min]) {

min = j;

}

}

/\* swap data[i] and data[min] \*/

temp = data[i];

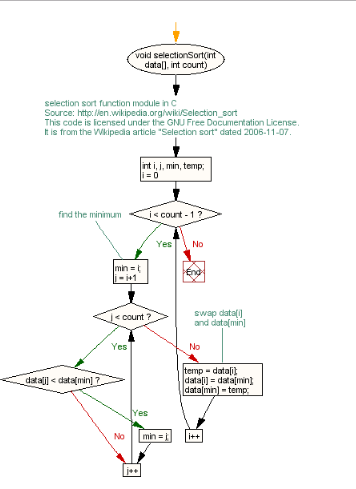
data[i] = data[min];

data[min] = temp;

}

}

Khi áp dụng tool Visustin để tự động generate flow chart



Tài liệu tham khảo

[1] <https://www.aivosto.com/articles/refactoring.html>

[2] <https://viblo.asia/p/code-refactoring-PaLkDYldvlX>

[3] <https://hieusensei.com/ma-xau-code-smells-va-refactor>