TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

TIỂU LUẬN MÔN

MẪU THIẾT KẾ

MAKING METHOD

CALLS SIMPLER

Người hướng dẫn: **TS NGUYỄN HỮU PHƯỚC**

Người thực hiện: **NGUYỄN THÀNH QUANG HUY – 518H0020**

NGUYỄN THÀNH KHANG – 518H0372

Lớp : **18H50202**

18H50303

Khoá : **22**

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2021

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

TIỂU LUẬN MÔN

MẪU THIẾT KẾ

MAKING METHOD

CALLS SIMPLER

Người hướng dẫn: **TS NGUYỄN HỮU PHƯỚC**

Người thực hiện: **NGUYỄN THÀNH QUANG HUY – 518H0020**

NGUYỄN THÀNH KHANG – 518H0372

Lớp : **18H50202**

18H50303

Khoá : **22**

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2021

LỜI CẢM ƠN

Đây là bài tiểu luận về môn Mẫu thiết kế giữa kì của chúng em, tuy nhiên vẫn còn nhiều thiếu sót, mong Thầy sẽ góp ý cho chúng em thêm lời khuyên để tiếp thêm kiến thức cũng như động lực cho chúng em trên con đường học tập sắp tới. Chân thành cảm ơn thầy!

TIỂU LUẬN ĐƯỢC HOÀN THÀNH TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

Tôi xin cam đoan đây là sản phẩm tiểu luận của riêng tôi / chúng tôi và được sự hướng dẫn của TS Nguyễn Hữu Phước. Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo.

Ngoài ra, trong đồ án còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc.

Nếu phát hiện có bất kỳ sự gian lận nào tôi xin hoàn toàn chịu trách nhiệm về nội dung đồ án của mình. Trường đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

TP. Hồ Chí Minh, ngày 26 tháng 03 năm 2021

Tác giả

(ký tên và ghi rõ họ tên)

Nguyễn Thành Quang Huy

Nguyễn Thành Khang

PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN

Phần xác nhận của GV hướng dẫn

_____

_____

_____

_____

_____

_____

_____

Tp. Hồ Chí Minh, ngày tháng năm
(kí và ghi họ tên)

Phần đánh giá của GV chấm bài

_____

_____

_____

_____

_____

_____

_____

Tp. Hồ Chí Minh, ngày tháng năm
(kí và ghi họ tên)

Table of Contents

CHƯƠNG 1 – GIỚI THIỆU METHODS

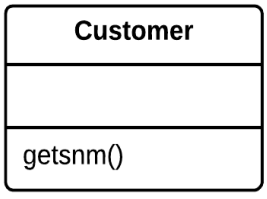
Đối tượng(Objects) là tất cả về giao diện(Interfaces). Tạo ra các giao diện dễ hiểu và dễ sử dụng là một kỹ năng quan trọng để phát triển phần mềm hướng đối tượng(OOP) tốt. Thông qua tiểu luận lần này, mục đích chính là khám phá việc tái cấu trúc để làm cho các giao diện trở nên đơn giản hơn.

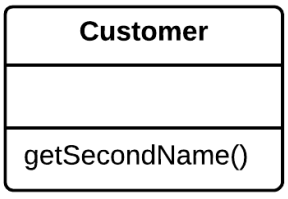
Bên dưới đây là một số các phương thức(Methods) giúp ích nhiều trong việc hỗ trợ phát triển và xây dựng phần mềm đơn giản hơn.

1.1 Rename Method.

Thường thì điều đơn giản và quan trọng nhất bạn có thể làm là thay đổi tên của một phương thức. Đặt tên là một công cụ quan trọng trong giao tiếp. Nếu bạn hiểu chương trình đang làm gì, bạn không nên ngại sử dụng phương thức đổi tên(Rename Method) để truyền lại kiến thức đó. Bạn cũng có thể (và nên) đổi tên các biến và lớp. Nhìn chung, những cách đổi tên này là sự thay thế văn bản khá đơn giản, vì vậy tôi đã không thêm cấu trúc lại bổ sung cho chúng.

Vấn đề	Giải pháp
Tên của một phương thức không giải thích những gì phương thức đó làm được.	Đổi tên phương thức.





Tại sao cần cấu trúc lại(Refactor)?

Có lẽ ngay từ đầu một phương thức đã được đặt tên kém - ví dụ, ai đó đã tạo ra phương thức một cách vội vàng và không quan tâm đúng mức đến việc đặt tên cho nó.

Hoặc có lẽ lúc đầu phương thức được đặt tên tốt nhưng khi chức năng của nó phát triển, tên phương thức không còn là một bộ mô tả tốt nữa.

Những lợi ích

Khả năng đọc mã. Cố gắng đặt cho phương pháp mới một cái tên phản ánh những gì nó thực hiện. Ví dụ như như createOrder (), renderCustomerInfo (), v.v.

Làm thế nào để cấu trúc lại?

Xem liệu phương thức được định nghĩa trong lớp cha hay lớp con. Nếu vậy, bạn cũng phải lặp lại tất cả các bước trong các lớp này.

Phương pháp tiếp theo là quan trọng để duy trì chức năng của chương trình trong quá trình tái cấu trúc. Tạo một phương thức mới với một tên mới. Sao chép mã của phương pháp cũ vào nó. Xóa tất cả mã trong phương thức cũ và thay vào đó, hãy chèn một lệnh gọi cho phương thức mới.

Tìm tất cả các tham chiếu đến phương pháp cũ và thay thế chúng bằng các tham chiếu đến phương pháp mới.

Xóa phương pháp cũ. Nếu phương pháp cũ là một phần của giao diện công khai, không thực hiện bước này. Thay vào đó, hãy đánh dấu phương pháp cũ là không được dùng nữa.

1.2 Add Parameter.

Bản thân các thông số có vai trò khá lớn đối với các giao diện. Thêm tham số và xóa tham số là những cách tái cấu trúc phổ biến.

Vấn đề	Giải pháp
Một phương thức không có đủ dữ liệu để thực hiện các hành động nhất định.	Tạo một tham số mới để truyền dữ liệu cần thiết.

Tại sao cần cấu trúc lại(Refactor)?

Bạn cần thực hiện các thay đổi đối với một phương thức và những thay đổi này yêu cầu thêm thông tin hoặc dữ liệu mà trước đây phương thức không có sẵn.

Những lợi ích

Sự lựa chọn ở đây là giữa việc thêm một tham số mới và thêm một trường riêng tư mới có chứa dữ liệu mà phương thức cần. Một trường thích hợp hơn khi bạn cần một số dữ liệu không thường xuyên hoặc thường xuyên thay đổi mà không có ích lợi gì để giữ nó trong một đối tượng mọi lúc. Trong trường hợp này, một tham số mới sẽ phù hợp hơn so với một trường riêng tư và việc tái cấu trúc sẽ có kết quả. Nếu không, hãy thêm một trường riêng tư và điền vào nó với dữ liệu cần thiết trước khi gọi phương thức.

Hạn chế

Thêm một tham số mới luôn dễ dàng hơn là xóa nó, đó là lý do tại sao tham số thường liệt kê các hộp chú giải với kích thước kỳ cục. Vấn đề này được gọi là **Long Parameter List**.

Nếu bạn cần thêm một tham số mới, đôi khi điều này có nghĩa là lớp của bạn không chứa dữ liệu cần thiết hoặc các tham số hiện có không chứa dữ liệu liên quan cần thiết. Trong cả hai trường hợp, giải pháp tốt nhất là xem xét việc di chuyển dữ liệu đến lớp chính hoặc đến các lớp khác mà các đối tượng của chúng đã có thể truy cập từ bên trong phương thức.

Làm thế nào để cấu trúc lại

Xem liệu phương thức được định nghĩa trong lớp cha hay lớp con. Nếu phương thức có trong chúng, bạn cũng cần phải lặp lại tất cả các bước trong các lớp này.

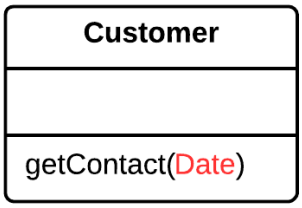
Bước sau là rất quan trọng để giữ cho chương trình của bạn hoạt động trong quá trình tái cấu trúc. Tạo một phương thức mới bằng cách sao chép phương thức cũ và thêm tham số cần thiết vào nó. Thay thế mã cho phương thức cũ bằng một lệnh gọi đến phương thức mới. Bạn có thể thêm bất kỳ giá trị nào vào tham số mới (chẳng hạn như null cho các đối tượng hoặc một số 0 cho các số).

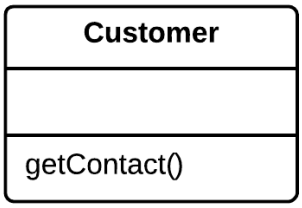
Tìm tất cả các tham chiếu đến phương pháp cũ và thay thế chúng bằng các tham chiếu đến phương pháp mới.

Xóa phương pháp cũ. Không thể xóa nếu phương thức cũ là một phần của giao diện công khai. Nếu đúng như vậy, hãy đánh dấu phương thức cũ là không được dùng nữa.

1.3 Remove Parameter.

Vấn đề	Giải pháp
Một tham số không được sử dụng trong phần thân của một phương thức.	Loại bỏ tham số không sử dụng.





Tại sao Refactor?

Mọi tham số trong một lệnh gọi phương thức buộc người lập trình đọc nó để tìm ra thông tin nào được tìm thấy trong tham số này. Và nếu một tham số hoàn toàn không được sử dụng trong phần thân của phương thức, thì việc "cào bằng không" này là vô ích.

Và trong mọi trường hợp, các tham số bổ sung là mã bổ sung phải được chạy.

Đôi khi chúng tôi thêm các tham số với tầm nhìn về tương lai, dự đoán các thay đổi đối với phương thức mà tham số có thể cần thiết. Tương tự, kinh nghiệm cho thấy rằng tốt hơn là chỉ thêm một tham số khi nó thực sự cần thiết. Rốt cuộc, những thay đổi được dự đoán trước thường vẫn chỉ là - dự đoán.

Những lợi ích

Một phương thức chỉ chứa các tham số mà nó thực sự yêu cầu.

Khi nào không sử dụng

Nếu phương thức được triển khai theo các cách khác nhau trong các lớp con hoặc trong một lớp cha và tham số của bạn được sử dụng trong các triển khai đó, hãy để nguyên tham số.

Làm thế nào để cấu trúc lại

Xem liệu phương thức được định nghĩa trong lớp cha hay lớp con. Nếu vậy, tham số có được sử dụng ở đó không? Nếu tham số được sử dụng trong một trong những cách triển khai này, hãy tạm dừng kỹ thuật tái cấu trúc này.

Bước tiếp theo là quan trọng để giữ chương trình hoạt động trong quá trình tái cấu trúc. Tạo một phương thức mới bằng cách sao chép phương thức cũ và xóa tham số có liên quan khỏi nó. Thay thế mã của phương thức cũ bằng một lệnh gọi đến phương thức mới.

Tìm tất cả các tham chiếu đến phương pháp cũ và thay thế chúng bằng các tham chiếu đến phương pháp mới.

Xóa phương pháp cũ. Không thực hiện bước này nếu phương pháp cũ là một phần của giao diện công khai. Trong trường hợp này, hãy đánh dấu phương pháp cũ là không được dùng nữa.

Methods Treatment Long Parameter List

Các lập trình viên mới làm quen với các đối tượng thường sử dụng danh sách tham số dài, đây là đặc trưng của các môi trường phát triển khác. Các đối tượng cho phép bạn giữ cho danh sách tham số ngắn gọn, và một số khác liên quan đến việc tái cấu trúc cung cấp cho bạn cách để rút ngắn chúng. Nếu bạn đang chuyển một số giá trị từ một đối tượng, hãy sử dụng ***Preserve Whole Object*** để giảm tất cả các giá trị xuống một đối tượng duy nhất. Nếu đối tượng này không tồn tại, bạn có thể tạo nó bằng ***Introduce Parameter Object***. Nếu bạn có thể lấy dữ liệu từ một đối tượng mà phương thức đã có quyền truy cập, bạn có thể loại bỏ các tham số bằng ***Replace Parameter with Method***. Nếu bạn có các tham số được sử dụng để xác định hành vi có điều kiện, bạn có thể sử dụng ***Replace Parameter with Explicit Methods***. Bạn có thể kết hợp một số phương pháp tương tự bằng cách thêm một tham số với ***Parameterize Method***.

1.4 Preserve Whole Object

Vấn đề	Giải pháp
Bạn nhận một số giá trị từ một đối tượng và sau đó chuyển chúng dưới dạng tham số cho một phương thức.	Thay vào đó, hãy thử chuyển toàn bộ đối tượng.
int low = daysTempRange.getLow(); **int** high = daysTempRange.getHigh(); **boolean** withinPlan = plan.withinRange(low, high);	**boolean** withinPlan = plan.withinRange(daysTempRange);

Tại sao Refactor?

Vấn đề là mỗi lần trước khi phương thức của bạn được gọi, các phương thức của đối tượng tham số tương lai phải được gọi. Nếu các phương thức này hoặc số lượng dữ liệu thu được cho phương thức bị thay đổi, bạn sẽ cần phải cẩn thận tìm ra hàng tá vị trí như vậy trong chương trình và thực hiện các thay đổi này trong từng vị trí đó.

Sau khi bạn áp dụng kỹ thuật tái cấu trúc này, mã để lấy tất cả dữ liệu cần thiết sẽ được lưu trữ ở một nơi - chính phương thức này.

Những lợi ích

Thay vì một dãy các tham số, bạn sẽ thấy một đối tượng duy nhất có tên dễ hiểu.

Nếu phương thức cần thêm dữ liệu từ một đối tượng, bạn sẽ không cần phải viết lại tất cả những nơi mà phương thức được sử dụng - chỉ đơn thuần là bên trong chính phương thức đó.

Hạn chế

Đôi khi sự chuyển đổi này khiến một phương thức trở nên kém linh hoạt hơn: trước đây phương thức có thể lấy dữ liệu từ nhiều nguồn khác nhau nhưng hiện tại, do cấu trúc lại, chúng tôi đang giới hạn việc sử dụng nó đối với chỉ các đối tượng có giao diện cụ thể.

Làm thế nào để cấu trúc lại

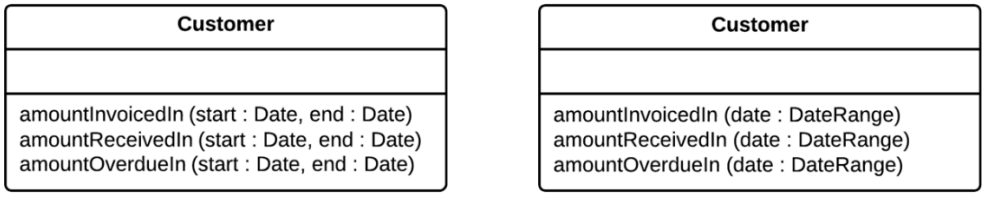
Tạo một tham số trong phương thức cho đối tượng mà từ đó bạn có thể nhận được các giá trị cần thiết.

Bây giờ bắt đầu xóa từng tham số cũ khỏi phương thức, thay thế chúng bằng lời gọi đến các phương thức liên quan của đối tượng tham số. Kiểm tra chương trình sau mỗi lần thay thế một tham số.

Xóa mã getter khỏi đối tượng tham số trước cuộc gọi phương thức.

1.5 Introduce Parameter Object

Vấn đề	Giải pháp
Các phương thức của bạn chứa một nhóm tham số lặp lại.	Thay thế các tham số này bằng một đối tượng.



Tại sao Refactor?

Các nhóm tham số giống hệt nhau thường gặp trong nhiều phương pháp. Điều này gây ra sự trùng lặp mã của cả bản thân các tham số và các hoạt động liên quan. Bằng cách hợp nhất các tham số trong một lớp, bạn cũng có thể di chuyển các phương thức để xử lý dữ liệu này ở đó, giải phóng các phương thức khác khỏi mã này.

Những lợi ích

Mã dễ đọc hơn. Thay vì một dãy các tham số, bạn sẽ thấy một đối tượng duy nhất có tên dễ hiểu.

Các nhóm tham số giống hệt nhau nằm rải rác ở đây tạo ra kiểu trùng lặp mã của riêng chúng: trong khi mã giống hệt nhau không được gọi, các nhóm tham số và đối số giống hệt nhau liên tục gặp phải.

Hạn chế

Nếu bạn chỉ di chuyển dữ liệu sang một lớp mới và không có kế hoạch di chuyển bất kỳ hành vi hoặc hoạt động liên quan nào ở đó, điều này bắt đầu có mùi của một Lớp dữ liệu.

Làm thế nào để cấu trúc lại

Tạo một lớp mới sẽ đại diện cho nhóm tham số của bạn. Làm cho lớp trở nên bất biến.

Trong phương thức mà bạn muốn cấu trúc lại, hãy sử dụng **Add Parameter**, đây là nơi đối tượng tham số của bạn sẽ được truyền vào. Trong tất cả các lệnh gọi phương thức, hãy chuyển đối tượng được tạo từ các tham số phương thức cũ sang tham số này.

Bây giờ bắt đầu xóa từng tham số cũ khỏi phương thức, thay thế chúng trong mã bằng các trường của đối tượng tham số. Chạy thử chương trình sau mỗi lần thay thế tham số.

Khi hoàn tất, hãy xem liệu có bất kỳ điểm nào trong việc di chuyển một phần của phương thức (hoặc đôi khi thậm chí là toàn bộ phương thức) sang một lớp đối tượng tham số hay không. Nếu vậy, hãy sử dụng **Move Method** hoặc **Extract Method**.

1.6 Replace Parameter with Explicit Methods

Vấn đề	Giải pháp
Một phương thức được chia thành nhiều phần, mỗi phần được chạy tùy thuộc vào giá trị của một tham số.	Trích xuất các phần riêng lẻ của phương thức thành các phương thức của riêng chúng và gọi chúng thay vì phương thức ban đầu.

void setValue(**String** name, **int** value) {
if (name.equals("height")) {
height = value;
return; }
if (name.equals("width")) {
width = value;
return; }
Assert.shouldNeverReachHere();
	void setHeight(**int** arg) {
	height = arg;
	}
	void setWidth(**int** arg) {
	width = arg;
	}

Tại sao Refactor?

Cải thiện khả năng đọc mã. Có thể hiểu mục đích của startEngine () dễ hơn nhiều so với setValue ("engineEnabled", true).

Những lợi ích

Mã dễ đọc hơn. Thay vì một dãy các tham số, bạn sẽ thấy một đối tượng duy nhất có tên dễ hiểu.

Khi nào không sử dụng

Không thay thế một tham số bằng các phương thức rõ ràng nếu một phương thức hiếm khi được thay đổi và các biến thể mới không được thêm vào bên trong nó.

Làm thế nào để cấu trúc lại

Đối với mỗi biến thể của phương pháp, hãy tạo một phương thức riêng biệt. Chạy các phương thức này dựa trên giá trị của một tham số trong phương thức chính.

Tìm tất cả những nơi mà phương thức gốc được gọi. Ở những nơi này, hãy thực hiện lệnh gọi cho một trong các biến thể phụ thuộc tham số mới.

Khi không còn lệnh gọi nào đến phương thức ban đầu, hãy xóa nó.

1.7 Replace Parameter with Method Call

Vấn đề	Giải pháp
Trước khi gọi phương thức, phương thức thứ hai được chạy và kết quả của nó được gửi lại phương thức đầu tiên dưới dạng đối số. Nhưng giá trị tham số có thể đã được lấy bên trong phương thức đang được gọi.	Thay vì chuyển giá trị qua một tham số, hãy đặt mã nhận giá trị bên trong phương thức.
int basePrice = quantity * itemPrice; **double** seasonDiscount = **this**.getSeasonalDiscount(); **double** fees = **this**.getFees(); **double** finalPrice = discountedPrice(basePrice, seasonDiscount, fees);	**int** basePrice = quantity * itemPrice; **double** finalPrice = discountedPrice(basePrice);

Tại sao Refactor?

Một danh sách dài các tham số thật khó hiểu. Ngoài ra, các lệnh gọi đến các phương thức như vậy thường giống như một loạt các tầng, với các phép tính giá trị quanh co và thú vị mà khó điều hướng vẫn phải được chuyển cho phương thức. Vì vậy, nếu một giá trị tham số có thể được tính toán với sự trợ giúp của một phương thức, hãy thực hiện việc này bên trong chính phương thức đó và loại bỏ tham số.

Những lợi ích

Chúng tôi loại bỏ các tham số không cần thiết và đơn giản hóa các cuộc gọi phương thức. Các thông số như vậy thường được tạo ra không phải cho dự án như hiện tại, mà là để hướng tới những nhu cầu trong tương lai có thể không bao giờ đến.

Những hạn chế

Bạn có thể cần tham số vào ngày mai cho các nhu cầu khác ... khiến bạn phải viết lại phương thức.

Làm thế nào để cấu trúc lại

Đảm bảo rằng mã nhận giá trị không sử dụng các tham số từ phương thức hiện tại, vì chúng sẽ không có sẵn từ bên trong phương thức khác. Nếu vậy, việc di chuyển mã là không thể.

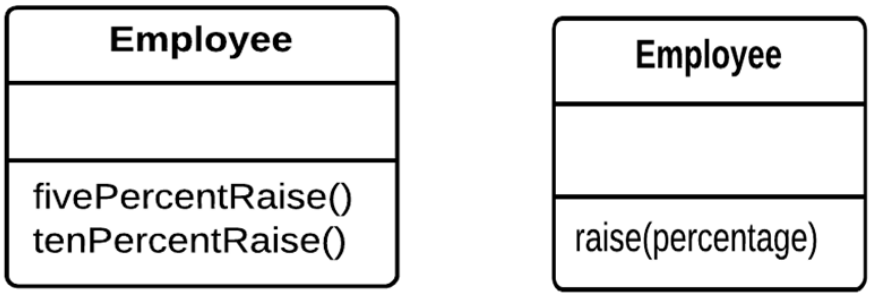
Nếu mã có liên quan phức tạp hơn một phương thức hoặc lệnh gọi hàm đơn lẻ, hãy sử dụng **Extract Method** để tách mã này trong một phương thức mới và làm cho cuộc gọi trở nên đơn giản.

Trong mã của phương thức chính, hãy thay thế tất cả các tham chiếu đến tham số đang được thay thế bằng các lệnh gọi đến phương thức nhận giá trị.

Sử dụng **Remove Parameter** để loại bỏ tham số hiện không sử dụng.

1.8 Parameterize Method

Vấn đề	Giải pháp
Nhiều phương thức thực hiện các hành động tương tự chỉ khác nhau về giá trị, số hoặc phép toán bên trong của chúng.	Kết hợp các phương thức này bằng cách sử dụng một tham số sẽ chuyển giá trị đặc biệt cần thiết.



Tại sao Refactor?

Nếu bạn có các phương pháp tương tự, bạn có thể có mã trùng lặp, với tất cả các hậu quả mà điều này kéo theo.

Hơn nữa, nếu bạn cần thêm một phiên bản khác của chức năng này, bạn sẽ phải tạo thêm một phương pháp khác. Thay vào đó, bạn có thể chỉ cần chạy phương thức hiện có với một tham số khác.

Những hạn chế

Đôi khi kỹ thuật tái cấu trúc này có thể bị đi quá xa, dẫn đến một phương pháp phổ biến dài và phức tạp thay vì nhiều phương pháp đơn giản hơn.

Ngoài ra, hãy cẩn thận khi chuyển kích hoạt / hủy kích hoạt chức năng sang một tham số. Điều này cuối cùng có thể dẫn đến việc tạo ra một toán tử điều kiện lớn sẽ cần được xử lý thông qua **Replace Parameter with Explicit Methods**.

Làm thế nào để cấu trúc lại

Tạo một phương thức mới với một tham số và di chuyển nó đến mã giống nhau cho tất cả các lớp, bằng cách áp dụng **Extract Method**. Lưu ý rằng đôi khi chỉ một phần nhất định của các phương pháp là giống nhau. Trong trường hợp này, việc tái cấu trúc chỉ bao gồm việc trích xuất phần tương tự sang một phương thức mới.

Trong mã của phương thức mới, hãy thay thế giá trị đặc biệt / khác biệt bằng một tham số.

Đối với mỗi phương thức cũ, hãy tìm các vị trí mà nó được gọi, thay thế các lệnh gọi này bằng các lệnh gọi đến phương thức mới bao gồm một tham số. Sau đó xóa phương pháp cũ.

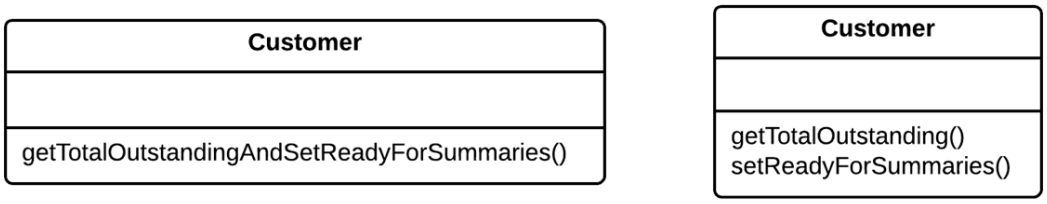
Methods Treatment Parameter Lists

Doug Lea đã đưa cho tôi một cảnh báo về việc tái cấu trúc làm giảm danh sách tham số. Lập trình đồng thời thường sử dụng danh sách tham số dài. Thông thường, điều này xảy ra để bạn có thể truyền vào các tham số không thay đổi, như các đối tượng giá trị và đối tượng tích hợp thường là như vậy. Thông thường, bạn có thể thay thế danh sách tham số dài bằng các đối tượng bất biến, nhưng nếu không, bạn cần phải thận trọng với nhóm tái cấu trúc này.

Một trong những quy ước có giá trị nhất mà tôi đã sử dụng trong nhiều năm là tách biệt rõ ràng các phương thức thay đổi trạng thái (bổ ngữ) khỏi các phương thức thay đổi trạng thái (truy vấn). Tôi không biết đã bao nhiêu lần tôi tự gặp rắc rối, hoặc chứng kiến người khác gặp rắc rối bằng cách trộn lẫn những thứ này với nhau. Vì vậy, bất cứ khi nào tôi thấy chúng kết hợp với nhau, tôi sử dụng ***Separate Query from Modifier*** để loại bỏ chúng.

1.9 Separate Query from Modifier.

Vấn đề	Giải pháp
Bạn có một phương thức trả về một giá trị nhưng cũng thay đổi một cái gì đó bên trong một đối tượng không?	Chia phương pháp thành hai phương pháp riêng biệt. Như bạn mong đợi, một trong số chúng sẽ trả về giá trị và cái còn lại sửa đổi đối tượng.



Tại sao cần cấu trúc lại(Refactor)?

Kỹ thuật bao thanh toán này thực hiện **Command and Query Responsibility Segregation**. Nguyên tắc này yêu cầu chúng ta tách mã chịu trách nhiệm lấy dữ liệu từ mã thay đổi thứ gì đó bên trong một đối tượng.

Mã để lấy dữ liệu được đặt tên là một truy vấn. Mã để thay đổi mọi thứ ở trạng thái hiển thị của một đối tượng được đặt tên là một công cụ sửa đổi. Khi một truy vấn và công cụ sửa đổi được kết hợp, bạn không có cách nào để lấy dữ liệu mà không thực hiện các thay đổi đối với điều kiện của nó. Nói cách khác, bạn đặt một câu hỏi và có thể thay đổi câu trả lời ngay cả khi nó đang được nhận. Vấn đề này thậm chí còn trở nên nghiêm trọng hơn khi người gọi truy vấn có thể không biết về "tác dụng phụ" của phương pháp, điều này thường dẫn đến lỗi thời gian chạy.

Nhưng hãy nhớ rằng các tác dụng phụ chỉ nguy hiểm trong trường hợp các chất bổ trợ làm thay đổi trạng thái nhìn thấy của một vật thể. Chúng có thể là, ví dụ, các trường có thể truy cập từ giao diện công khai của một đối tượng, mục nhập trong cơ sở dữ liệu, trong tệp, v.v. Nếu một công cụ sửa đổi chỉ lưu vào bộ nhớ cache một hoạt động

phức tạp và lưu nó trong trường riêng của một lớp, nó khó có thể gây ra bất kỳ mặt nào các hiệu ứng.

Những lợi ích

Nếu bạn có một truy vấn không thay đổi trạng thái của chương trình, bạn có thể gọi nó bao nhiêu lần tùy thích mà không phải lo lắng về những thay đổi ngoài ý muốn trong kết quả chỉ do bạn gọi phương thức.

Hạn chế

Trong một số trường hợp, việc lấy dữ liệu sau khi thực hiện một lệnh sẽ rất tiện lợi. Ví dụ: khi xóa nội dung nào đó khỏi cơ sở dữ liệu, bạn muốn biết có bao nhiêu hàng đã bị xóa.

Làm thế nào để cấu trúc lại

Tạo một phương thức truy vấn mới để trả lại những gì phương thức ban đầu đã làm.

Thay đổi phương thức ban đầu để nó chỉ trả về kết quả của việc gọi phương thức truy vấn mới.

Thay thế tất cả các tham chiếu đến phương thức gốc bằng một lệnh gọi phương thức truy vấn. Ngay trước dòng này, hãy gọi phương thức bổ trợ. Điều này sẽ giúp bạn tránh khỏi các tác dụng phụ trong trường hợp nếu phương pháp ban đầu được sử dụng trong điều kiện của một toán tử hoặc vòng lặp có điều kiện.

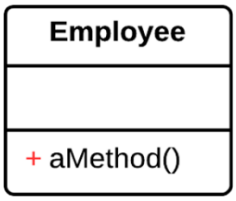
Loại bỏ mã trả về giá trị trong phương thức ban đầu, phương thức này hiện đã trở thành một phương thức bổ trợ thích hợp.

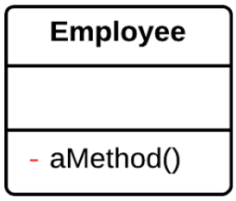
Methods Treatment Interfaces

Giao diện tốt chỉ hiển thị những gì họ phải và không hơn thế nữa. Bạn có thể cải thiện giao diện bằng cách ẩn mọi thứ. Tất nhiên, tất cả dữ liệu nên được ẩn (tôi hy vọng tôi không cần phải nói với bạn làm điều đó), nhưng cũng nên ẩn bất kỳ phương pháp nào có thể được ẩn. Khi cấu trúc lại, bạn thường cần hiển thị mọi thứ trong một thời gian và sau đó che chúng bằng ***Hide Method*** và ***Remove Setting Method***.

1.10 Hide Method.

Vấn đề	Giải pháp
Một phương thức không được sử dụng bởi các lớp khác hoặc chỉ được sử dụng bên trong hệ thống phân cấp lớp của chính nó.	Đặt phương thức ở chế độ riêng tư hoặc được bảo vệ.





Tại sao cần cấu trúc lại(Refactor)?

Thông thường, nhu cầu ẩn các phương thức để lấy và đặt giá trị là do sự phát triển của giao diện phong phú hơn cung cấp hành vi bổ sung, đặc biệt nếu bạn bắt đầu với một lớp bổ sung ít ngoài việc đóng gói dữ liệu đơn thuần.

Khi hành vi mới được tích hợp vào lớp, bạn có thể thấy rằng các phương thức public getter và setter không còn cần thiết nữa và có thể bị ẩn đi. Nếu bạn đặt phương thức getter hoặc setter ở chế độ riêng tư và áp dụng quyền truy cập trực tiếp vào các biến, bạn có thể xóa phương thức.

Những lợi ích

Các phương pháp ẩn giúp mã của bạn phát triển dễ dàng hơn. Khi bạn thay đổi một phương thức private, bạn chỉ cần lo lắng về việc làm thế nào để không phá vỡ lớp hiện tại vì bạn biết rằng phương thức này không thể được sử dụng ở bất kỳ nơi nào khác.

Bằng cách đặt các phương thức ở chế độ riêng tư, bạn nhấn mạnh tầm quan trọng của giao diện công khai của lớp và của các phương thức vẫn công khai.

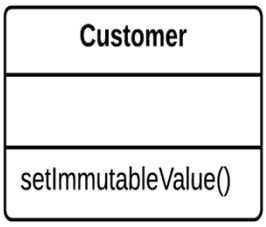
Làm thế nào để cấu trúc lại

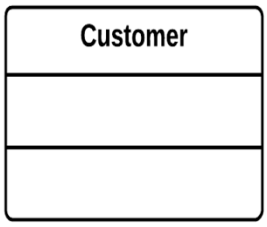
Thường xuyên cố gắng tìm các phương pháp có thể được đặt ở chế độ riêng tư. Phân tích mã tĩnh và phạm vi kiểm tra đơn vị tốt có thể mang lại một bước tiến lớn.

Đặt mỗi phương pháp càng riêng tư càng tốt.

1.11 Remove Setting Method.

Vấn đề	Giải pháp
Giá trị của một trường chỉ nên được đặt khi nó được tạo và không thay đổi bất kỳ lúc nào sau đó.	Vì vậy, hãy xóa các phương thức đặt giá trị của trường.





Tại sao cần cấu trúc lại(Refactor)?

Bạn muốn ngăn chặn bất kỳ thay đổi nào đối với giá trị của một trường.

Làm thế nào để cấu trúc lại

Giá trị của một trường chỉ được thay đổi trong hàm tạo. Nếu hàm tạo không chứa tham số để thiết lập giá trị, hãy thêm một tham số.

Tìm tất cả các cuộc gọi setter.

- Nếu một lệnh gọi setter nằm ngay sau một lệnh gọi hàm tạo của lớp hiện tại, hãy di chuyển đối số của nó sang lệnh gọi hàm tạo và xóa setter.
- Thay thế lời gọi setter trong hàm tạo bằng quyền truy cập trực tiếp vào trường.

Constructors

Constructors là một tính năng đặc biệt khó xử của Java và C ++ bởi vì chúng buộc bạn phải biết lớp của một đối tượng mà bạn cần tạo. Thường thì bạn không cần biết điều này. Những điều cần biết có thể được loại bỏ bằng ***Replace Constructor*** bằng ***Factory Method.***

1.12 Replace Constructor with Factory Method.

Vấn đề	Giải pháp
Bạn có một phương thức khởi tạo phức tạp không chỉ là thiết lập các giá trị tham số trong các trường đối tượng.	Tạo một phương thức factory và sử dụng nó để thay thế các lời gọi phương thức khởi tạo.

class Employee {
Employee(**int** type) {
this.type = type;
}
// ...
}
	class Employee {
	static Employee create(**int** type) {
	employee = **new**
	Employee(type);
	// do some heavy lifting.
	return employee;
	}
	// ...
	}

Tại sao cần cấu trúc lại(Refactor)?

Lý do rõ ràng nhất cho việc sử dụng kỹ thuật tái cấu trúc này có liên quan đến **Thay thế mã loại bằng các lớp con**.

Bạn có mã trong đó một đối tượng đã được tạo trước đó và giá trị của kiểu mã đã được chuyển cho nó. Sau khi sử dụng phương pháp tái cấu trúc, một số lớp con đã xuất hiện và từ chúng, bạn cần tạo các đối tượng tùy thuộc vào giá trị của kiểu được mã hóa. Việc thay đổi hàm tạo ban đầu để làm cho nó trả về các đối tượng của lớp con là không thể, vì vậy thay vào đó chúng ta tạo một phương thức static factory sẽ trả về các đối tượng của các lớp cần thiết, sau đó nó sẽ thay thế tất cả các lệnh gọi đến hàm tạo ban đầu.

Các phương thức ban đầu cũng có thể được sử dụng trong các tình huống khác, khi các hàm tạo không đáp ứng được nhiệm vụ. Chúng có thể quan trọng khi cố gắng **Thay đổi Giá trị thành Tham chiếu**. Chúng cũng có thể được sử dụng để thiết lập các chế độ tạo khác nhau vượt ra ngoài số lượng và loại tham số.

Những lợi ích

Một phương thức factory không nhất thiết phải trả về một đối tượng của lớp mà nó được gọi. Thường thì đây có thể là các lớp con của nó, được chọn dựa trên các đối số được cung cấp cho phương thức.

Một phương thức gốc có thể có tên tốt hơn mô tả những gì và cách nó trả về những gì nó làm, ví dụ Troops :: GetCrew (myTank).

Một phương thức factory có thể trả về một đối tượng đã được tạo, không giống như một phương thức khởi tạo, luôn tạo ra một thể hiện mới.

Làm thế nào để cấu trúc lại

Tạo một phương thức factory. Đặt một cuộc gọi đến hàm tạo hiện tại trong đó.

Thay thế tất cả các lệnh gọi phương thức khởi tạo bằng các lệnh gọi đến phương thức gốc.

Khai báo phương thức khởi tạo private.

Kiểm tra mã phương thức khởi tạo và cố gắng tách mã không liên quan trực tiếp đến việc xây dựng một đối tượng của lớp hiện tại, di chuyển mã đó sang phương thức gốc.

Methods Treatment JAVA

Truyền là một điểm hạn chế khác trong cuộc sống của lập trình viên Java. Càng nhiều càng tốt, hãy cố gắng tránh làm cho người dùng của một lớp thực hiện việc truyền xuống nếu bạn có thể chứa nó ở nơi khác bằng cách sử dụng ***Encapsulate Downcast***.

Java, giống như nhiều ngôn ngữ hiện đại, có cơ chế xử lý ngoại lệ để giúp việc xử lý lỗi dễ dàng hơn. Các lập trình viên không quen với việc này thường sử dụng mã lỗi để báo hiệu sự cố. Bạn có thể sử dụng ***Replace Error Code with Exception*** lệ để sử dụng các tính năng đặc biệt mới. Nhưng đôi khi ngoại lệ không phải là câu trả lời đúng; bạn nên kiểm tra trước bằng ***Replace Exception with Test***.

1.13 Replace Error Code with Exception.

Vấn đề	Giải pháp
Một phương thức trả về một giá trị đặc biệt chỉ ra một lỗi?	Thay vào đó, hãy ném một ngoại lệ.

int withdraw(**int** amount) {
if (amount > _balance) {
return -1;
}
else {
balance -= amount;
return 0;
}
}
	void withdraw(**int** amount) **throws** BalanceException {
	if (amount > _balance) {
	throw new BalanceException();
	}
	balance -= amount;
	}

Tại sao cần cấu trúc lại(Refactor)?

Trả lại mã lỗi là một lưu giữ lỗi thời từ lập trình thủ tục. Trong lập trình hiện đại, việc xử lý lỗi được thực hiện bởi các lớp đặc biệt, được đặt tên là các ngoại lệ. Nếu sự cố xảy ra, bạn "ném" một lỗi, lỗi này sẽ được một trong các trình xử lý ngoại lệ "bắt". Mã xử lý lỗi đặc biệt, bị bỏ qua trong điều kiện bình thường, được kích hoạt để phản hồi.

Những lợi ích

Miễn phí mã từ nhiều điều kiện để kiểm tra các mã lỗi khác nhau. Các trình xử lý ngoại lệ là một cách ngắn gọn hơn nhiều để phân biệt các đường dẫn thực thi bình thường với các đường dẫn bất thường.

Các lớp ngoại lệ có thể triển khai các phương thức của riêng chúng, do đó chứa một phần chức năng xử lý lỗi (chẳng hạn như để gửi thông báo lỗi).

Không giống như các ngoại lệ, mã lỗi không thể được sử dụng trong một phương thức khởi tạo, vì một phương thức khởi tạo chỉ phải trả về một đối tượng mới.

Những hạn chế

Một người xử lý ngoại lệ có thể biến thành một chiếc nạng giống như goto. Tránh điều này! Không sử dụng ngoại lệ để quản lý việc thực thi mã. Các ngoại lệ chỉ nên được đưa ra để thông báo về một lỗi hoặc tình huống nghiêm trọng.

Làm thế nào để cấu trúc lại

Cố gắng thực hiện các bước tái cấu trúc này chỉ cho một mã lỗi tại một thời điểm. Điều này sẽ giúp bạn dễ dàng lưu giữ tất cả các thông tin quan trọng trong đầu và tránh sai sót.

Tìm tất cả các lệnh gọi đến một phương thức trả về mã lỗi và thay vì kiểm tra mã lỗi, hãy bọc nó trong các khối try / catch.

Bên trong phương thức, thay vì trả về mã lỗi, hãy đưa ra một ngoại lệ.

Thay đổi chữ ký phương thức để nó chứa thông tin về ngoại lệ được ném (@throws section).

1.14 Replace Exception with Test.

Vấn đề	Giải pháp
Bạn ném một ngoại lệ vào một nơi mà một bài kiểm tra đơn giản sẽ thực hiện công việc?	Thay thế ngoại lệ bằng một bài kiểm tra điều kiện.

double getValueForPeriod(**int** periodNumber) {
  try {
  return values[periodNumber];
  }
  catch (ArrayIndexOutOfBoundsException e){
  return 0;
  }
}
	double getValueForPeriod(**int** periodNumber) {
	  if (periodNumber >= values.length) {
	  return 0;
	  }
	  return values[periodNumber];
	}

Tại sao cần cấu trúc lại(Refactor)?

Các ngoại lệ nên được sử dụng để xử lý các hành vi bất thường liên quan đến một lỗi không mong muốn. Chúng không nên dùng để thay thế cho thử nghiệm. Nếu một ngoại lệ có thể tránh được bằng cách đơn giản xác minh một điều kiện trước khi chạy, thì hãy làm như vậy. Các trường hợp ngoại lệ nên được dành cho các lỗi thực sự.

Ví dụ: bạn vào một bãi mìn và kích hoạt một quả mìn ở đó, dẫn đến một ngoại lệ; ngoại lệ đã được xử lý thành công và bạn đã được đưa lên không trung đến nơi an

toàn bên ngoài khu vực mỏ. Nhưng bạn có thể tránh được tất cả điều này bằng cách chỉ cần đọc biển cảnh báo trước bãi mìn để bắt đầu.

Những lợi ích

Một điều kiện đơn giản đôi khi có thể rõ ràng hơn mã xử lý ngoại lệ.

Làm thế nào để cấu trúc lại

Tạo một điều kiện cho một trường hợp cạnh và di chuyển nó trước khối try / catch.

Di chuyển mã từ phần bắt bên trong điều kiện này.

Trong phần catch, hãy đặt mã để ném một ngoại lệ không tên thông thường và chạy tất cả các bài kiểm tra.

Nếu không có ngoại lệ nào được đưa ra trong quá trình kiểm tra, hãy loại bỏ toán tử try / catch.

CHƯƠNG 2 – CODE DEMO

2.1 Source code các methods

Rename Method

```
class Person {
  //…
  public String getTelephoneNumber() {
    return ("(" + officeAreaCode + ") " + officeNumber);
  }
}

// Client code
phone = employee.getTelephoneNumber();
```

```
class Person {
  //…
  public String getOfficeTelephoneNumber() {
    return ("(" + officeAreaCode + ") " + officeNumber);
  }
}

// Client code
phone = employee.getOfficeTelephoneNumber();
```

Add Parameter

```
class Calendar {
  // ...
  private Set appointments;
  public ArrayList<Appointment> findAppointments(Date date) {
    Set result = new ArrayList();
    Iterator iter = kent.getCourses().iterator();
    while (iter.hasNext()) {
      Appointment each = (Appointment) iter.next();
      if (date.compareTo(each.date) == 0) {
        result.add(date);
      }
    }
    return result;
  }
}

// Somewhere in client code
Date today = new Date();
appointments = calendar.findAppointments(today);
```

```
class Calendar {
  // ...
  private Set appointments;
  public ArrayList<Appointment> findAppointments(Date date, String name) {
    Set result = new ArrayList();
    Iterator iter = kent.getCourses().iterator();
    while (iter.hasNext()) {
      Appointment each = (Appointment) iter.next();
      if (date.compareTo(each.date) == 0) {
        if (name == null || (name != null && name == each.name)) {
          result.add(date);
        }
      }
    }
    return result;
  }
}

// Somewhere in client code
Date today = new Date();
appointments = calendar.findAppointments(today, null);
```

Remove Parameter

```
int discount(int inputVal, int quantity, int yearToDate) {
  if (inputVal > 50) {
    inputVal -= 2;
  }
  if (quantity > 100) {
    inputVal -= 1;
  }
  if (yearToDate > 10000) {
    inputVal -= 4;
  }
  return inputVal;
}
```

```
int discount(final int inputVal, final int quantity, final int yearToDate) {
  int result = inputVal;
  if (inputVal > 50) {
    result -= 2;
  }
  if (quantity > 100) {
    result -= 1;
  }
  if (yearToDate > 10000) {
    result -= 4;
  }
  return result;
}
```

Separate Query from Modifier

```
class Guard {
  // ...
  public void checkSecurity(String[] people) {
    String found = findCriminalAndAlert(people);
    someLaterCode(found);
  }
  public String findCriminalAndAlert(String[] people) {
    for (int i = 0; i < people.length; i++) {
      if (people[i].equals("Don")) {
        sendAlert();
        return "Don";
      }
      if (people[i].equals("John")) {
        sendAlert();
        return "John";
      }
    }
    return "";
  }
}
```

```
class Guard {
  // ...
  public void checkSecurity(String[] people) {
    doSendAlert(people);
    String found = findCriminal(people);
    someLaterCode(found);
  }
  public void doSendAlert(String[] people) {
    if (findCriminal(people) != "") {
      sendAlert();
    }
  }
  public String findCriminal(String[] people) {
    for (int i = 0; i < people.length; i++) {
      if (people[i].equals ("Don")) {
        return "Don";
      }
      if (people[i].equals ("John")) {
        return "John";
      }
    }
    return "";
  }
}
```

Parameterize Method

```
class Employee {
  // ...
  public void promoteToManager() {
    type = Employee.MANAGER;
    salary *= 1.5;
  }
  public void tenPercentRaise() {
    salary *= 1.1;
  }
  public void fivePercentRaise() {
    salary *= 1.05;
  }
}

// Somewhere in client code
if (employee.yearsOfExperience > 5) {
  if (employee.clients.size() > 10) {
    employee.promoteToManager();
  }
  else {
    employee.fivePercentRaise();
  }
}
class Employee {
  // ...
  public void promoteToManager() {
    type = Employee.MANAGER;
    raise(0.5);
  }
  public void raise(double factor) {
    salary *= (1 + factor);
  }
}

// Somewhere in client code
if (employee.yearsOfExperience > 5) {
  if (employee.clients.size() > 10) {
    employee.promoteToManager();
  }
  else {
    employee.raise(0.05);
  }
}
```

Replace Parameter with Explicit Methods

```
class Order {
  // ...
  public static final int FIXED_DISCOUNT = 0;
  public static final int PERCENT_DISCOUNT = 1;

  public void applyDiscount(int type, double discount) {
    switch (type) {
      case FIXED_DISCOUNT:
        price -= discount;
        break;
      case PERCENT_DISCOUNT:
        price *= discount;
        break;
      default:
        throw new IllegalArgumentException("Invalid discount type");
    }
  }
}

// Somewhere in client code
if (weekend) {
  order.applyDiscount(Order.FIXED_DISCOUNT, 10);
}
if (order.items.size() > 5) {
  order.applyDiscount(Order.PERCENT_DISCOUNT, 0.2);
}
```

```
class Order {
  // ...
  public void applyFixedDiscount(double discount) {
    price -= discount;
  }
  public void applyPercentDiscount(double discount) {
    price *= discount;
  }
}

// Somewhere in client code
if (weekend) {
  order.applyFixedDiscount(10);
}
if (order.items.size() > 5) {
  order.applyPercentDiscount(0.2);
}
```

Preserve Whole Object

```
class Room {
  // ...
  public boolean withinPlan(HeatingPlan plan) {
    int low = getLowestTemp();
    int high = getHighestTemp();
    return plan.withinRange(low, high);
  }
}

class HeatingPlan {
  private TempRange range;
  public boolean withinRange(int low, int high) {
    return (low >= range.getLow() && high <= range.getHigh());
  }
}
```

```
class Room {
  // ...
  public boolean withinPlan(HeatingPlan plan) {
    return plan.withinRange(this);
  }
}

class HeatingPlan {
  private TempRange range;
  public boolean withinRange(Room room) {
    return (room.getLowestTemp() >= range.getLow() && room.getHighestTemp()
<= range.getHigh());
  }
}
```

Replace Parameter with Method Call

```
class Order {
  // ...
  public double getPrice() {
    int basePrice = quantity * itemPrice;
    int discountLevel;
    if (quantity > 100) {
      discountLevel = 2;
    }
    else {
      discountLevel = 1;
    }
    double finalPrice = discountedPrice(basePrice, discountLevel);
    return finalPrice;
  }
  private double discountedPrice(int basePrice, int discountLevel) {
```

```
    if (discountLevel == 2) {
      return basePrice * 0.1;
    }
    else {
      return basePrice * 0.05;
    }
  }
}
```

```
class Order {
  public double getPrice() {
    return discountedPrice();
  }
  private double discountedPrice() {
    if (getDiscountLevel() == 2) {
      return getBasePrice() * 0.1;
    }
    else {
      return getBasePrice() * 0.05;
    }
  }
  private int getDiscountLevel() {
    if (quantity > 100) {
      return 2;
    }
    else {
      return 1;
    }
  }
  private double getBasePrice() {
    return quantity * itemPrice;
  }
}
```

Introduce Parameter Object

```
class Account {
  // ...
  private Vector transactions = new Vector();

  public double getFlowBetween(Date start, Date end) {
    double result = 0;
    Enumeration e = transactions.elements();
    while (e.hasMoreElements()) {
      Transaction each = (Transaction) e.nextElement();
      if (each.getDate().compareTo(start) >= 0 &&
each.getDate().compareTo(end) <= 0) {
        result += each.getValue();
      }
    }
    return result;
```

```
  }
}

class Transaction {
  private Date chargeDate;
  private double value;

  public Transaction(double value, Date chargeDate) {
    this.value = value;
    this.chargeDate = chargeDate;
  }
  public Date getDate() {
    return chargeDate;
  }
  public double getValue() {
    return value;
  }
}

// Somewhere in client code…
double flow = account.getFlowBetween(startDate, endDate);
```

```
class Account {
  // ...
  private Vector transactions = new Vector();

  public double getFlowBetween(DateRange range) {
    double result = 0;
    Enumeration e = transactions.elements();
    while (e.hasMoreElements()) {
      Transaction each = (Transaction) e.nextElement();
      if (range.includes(each.getDate())) {
        result += each.getValue();
      }
    }
    return result;
  }
}

class Transaction {
  private Date chargeDate;
  private double value;

  public Transaction(double value, Date chargeDate) {
    this.value = value;
    this.chargeDate = chargeDate;
  }
  public Date getDate() {
    return chargeDate;
  }
```

```
  public double getValue() {
    return value;
  }
}

class DateRange {
  private final Date start;
  private final Date end;

  public DateRange(Date start, Date end) {
    this.start = start;
    this.end = end;
  }
  public Date getStart() {
    return start;
  }
  public Date getEnd() {
    return end;
  }
  public boolean includes(Date arg) {
    return arg.compareTo(start) >= 0 && arg.compareTo(end) <= 0;
  }
}

// Somewhere in client code…
double flow = account.getFlowBetween(new DateRange(startDate, endDate));
```

Remove Setting Method

```
class Account {
  // ...
  private String id;

  public Account(String id) {
    setId(id);
  }
  public void setId(String id) {
    this.id = id;
  }
}
```

```
class Account {
  // ...
  private String id;

  public Account(String id) {
    initializeId(id);
  }
  protected void initializeId(String id) {
```

```
    this.id = "ID" + id;
  }
}

class InterestAccount extends Account {
  private double interestRate;
  public InterestAccount(String id, double interestRate) {
    initializeId(id);
    this.interestRate = interestRate;
  }
}
```

Hide Method

```
class Person {
  private Department department;

  public Department getDepartment() {
    return department;
  }
  public void setDepartment(Department arg) {
    department = arg;
  }
}

class Department {
  private String chargeCode;
  private Person manager;

  public Department(Person manager) {
    this.manager = manager;
  }
  public Person getManager() {
    return manager;
  }

  //…
}

// Somewhere in client code
manager = john.getDepartment().getManager();
```

```
class Person {
  private Department department;

  public void setDepartment(Department arg) {
    department = arg;
  }
  public Person getManager() {
```

```
    return department.getManager();
  }
}

class Department {
  private String chargeCode;
  private Person manager;

  public Department(Person manager) {
    this.manager = manager;
  }
  public Person getManager() {
    return manager;
  }

  //…
}

// Somewhere in client code
manager = john.getManager();
```

Replace Constructor with Factory Method

```
class Employee {
  // ...
  static final int ENGINEER = 0;
  static final int SALESMAN = 1;
  static final int MANAGER = 2;

  public Employee(int type) {
    this.type = type;
  }
}

// Some clinet code.
Employee eng = new Employee(Employee.ENGINEER);
```

```
class Employee {
  // ...
  static final int ENGINEER = 0;
  static final int SALESMAN = 1;
  static final int MANAGER = 2;

  public static Employee create(int type) {
    switch (type) {
      case ENGINEER:
        return new Engineer();
      case SALESMAN:
        return new Salesman();
```

```
      case MANAGER:
        return new Manager();
      default:
        return new Employee(type);
    }
  }
  private Employee(int type) {
    this.type = type;
  }
}
class Engineer extends Employee {
  // ...
}
class Salesman extends Employee {
  // ...
}
class Manager extends Employee {
  // ...
}

// Some clinet code.
Employee eng = Employee.create(Employee.ENGINEER);
```

Replace Error Code with Exception

```
class Account {
  // ...
  private int balance;

  public int withdraw(int amount) {
    if (amount > balance) {
      return -1;
    }
    else {
      balance -= amount;
      return 0;
    }
  }
}

// Somewhere in client code.
if (account.withdraw(amount) == -1) {
  handleOverdrawn();
}
else {
  doTheUsualThing();
}
```

```
class Account {
  // ...
  private int balance;

  public void withdraw(int amount) throws BalanceException {
    if (amount > balance) {
      throw new BalanceException();
    }
    balance -= amount;
  }
}
class BalanceException extends Exception {}

// Somewhere in client code.
try {
  account.withdraw(amount);
  doTheUsualThing();
} catch (BalanceException e) {
  handleOverdrawn();
}
```

Replace Exception with Test

```
class ResourcePool {
  // ...
  private Stack available;
  private Stack allocated;

  public Resource getResource() {
    Resource result;
    try {
      result = (Resource) available.pop();
      allocated.push(result);
      return result;
    } catch (EmptyStackException e) {
      result = new Resource();
      allocated.push(result);
      return result;
    }
  }
}
```

```
class ResourcePool {
  // ...
  private Stack available;
  private Stack allocated;

  public Resource getResource() {
    Resource result;
```

```
    if (available.empty()) {
      result = new Resource();
    }
    else {
      result = (Resource) available.pop();
    }
    allocated.push(result);
    return result;
  }
}
```

2.1 Hình ảnh demo code

CHƯƠNG 3: Tài liệu tham khảo

1) https://refactoring.guru/refactoring/techniques/simplifying-method-calls
2) "2002 - Refactoring: Improving the Design of Existing Code by Martin Fowler, Kent Beck (Contributor), John Brant (Contributor), William Opdyke, don Roberts".
3) refactoring-examples/interactive/java at master · RefactoringGuru/refactoring-examples (github.com)
4)