VIETNAM GENERAL CONFEDERATION OF LABOUR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY



**Refactoring First Example**

**and Principles in Refactoring**

Instructor: **NGUYEN THANH PHUOC**

Executor: **Nguyễn Văn Vũ Anh Thiện - 518H0059**

Class **: 18H50202**

Executor: **Đặng Thọ Duy - 518H0168**

Class **: 18H50203**

Course  **: 22**

**HO CHI MINH CITY, 2020**

VIETNAM GENERAL CONFEDERATION OF LABOUR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY



**Refactoring First Example**

**and Principles in Refactoring**

Instructor: **NGUYEN THANH PHUOC**

Executor: **Nguyễn Văn Vũ Anh Thiện - 518H0059**

Class **: 18H50202**

Executor: **Đặng Thọ Duy - 518H0168**

Class **: 18H50203**

Course  **: 22**

**HO CHI MINH CITY, 2020**

Lời Cảm ơn

Trong thời gian thực hiện báo cáo, em đã nhận được rất nhiều sự giúp đỡ, góp ý và chỉ bảo tận tình của các thầy cô, em xin chân thành cảm ơn thầy Nguyễn Thành Phước, giáo viên bộ môn Thiết kế, đã tận tình hướng dẫn em trong thời gian thực hiện bài báo cáo. quá trình.

Em cũng xin chân thành cảm ơn các thầy cô trong TDT Unversity Khoa CNTT đã dạy cho em những kiến thức cần thiết, tạo cơ sở lý luận vững vàng và giúp đỡ em trong suốt quá trình học tập để hoàn thành bài báo cáo.

**ĐỒ ÁN ĐƯỢC HOÀN THÀNH**

**TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

Tôi cam kết rằng sản phẩm này thuộc về chúng tôi với sự giám hộ của Mr.Nguyễn Thành Phước. Nội dung nghiên cứu, kết quả của dự án này là trung thực và chưa được công bố dưới bất kỳ hình thức nào trước đây. Số liệu trong các bảng phân tích, nhận xét và đánh giá được tác giả thu thập từ nhiều nguồn khác nhau trong phần tài liệu tham khảo.

Bên cạnh đó, dự án của chúng tôi đã sử dụng nhiều loại nhận xét, đánh giá cũng như dữ liệu của các tác giả, tổ chức khác. Dự án này có trích dẫn và chú thích gốc.

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

*TP. Hồ Chí Minh, ngày tháng năm*

*Tác giả*

*(ký tên và ghi rõ họ tên)*

*(Đã Ký)*

*Nguyen Van Vu Anh Thien*

*(Đã Ký)*

*Dang Tho Duy*

CONFIRMATIONAND EVALUATION OF LECTURERS

Confirmation of lecturers

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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)

SUMMARY

Contents

[Chương 1 .Giới thiệu về Refactoring ( Tái cấu trúc ) 2](#_Toc66140668)

[1.1. Ví dụ : 2](#_Toc66140669)

[1.2. Khái niệm: 4](#_Toc66140670)

[1.3. Phân tách và phân phối lại phương thức statement 4](#_Toc66140671)

[1.4. Trích xuất cách tính điểm để thành khách hàng thân thuộc. 5](#_Toc66140672)

[1.5. Xoá những biến tạm: 6](#_Toc66140673)

[1.6. Thay thế tính toán Logic bằng tính đa hình 7](#_Toc66140674)

[1.7. Thừa kế ở cuối 9](#_Toc66140675)

[1.8. Kết luận: 10](#_Toc66140676)

[Chương 2. Các nguyên tắc tái cấu trúc 11](#_Toc66140677)

[2.1. Xác định tái cấu trúc: 11](#_Toc66140678)

[1. Noun form: 11](#_Toc66140679)

[2. Verb form: 11](#_Toc66140680)

[3. Tái cấu trúc để làm gì? 11](#_Toc66140681)

[2.2. Tại sao lại phải tái cấu trúc: 11](#_Toc66140682)

[4. Tái cấu trúc cải thiện thiết kế phần mềm: 11](#_Toc66140683)

[5. Tái cấu trúc giúp phần mềm dễ hiểu hơn 12](#_Toc66140684)

[2.3. Khi nào nên tái cấu trúc: 13](#_Toc66140685)

[6. Khi cần thêm tính năng mới 13](#_Toc66140686)

[7. Khi cần fix bug 13](#_Toc66140687)

[8. Khi xem tổng quan code 13](#_Toc66140688)

[2.4. Các vấn đề gặp phải khi tái cấu trúc: 13](#_Toc66140689)

[9. Cơ sở dữ liệu 14](#_Toc66140690)

[10. Interface 14](#_Toc66140691)

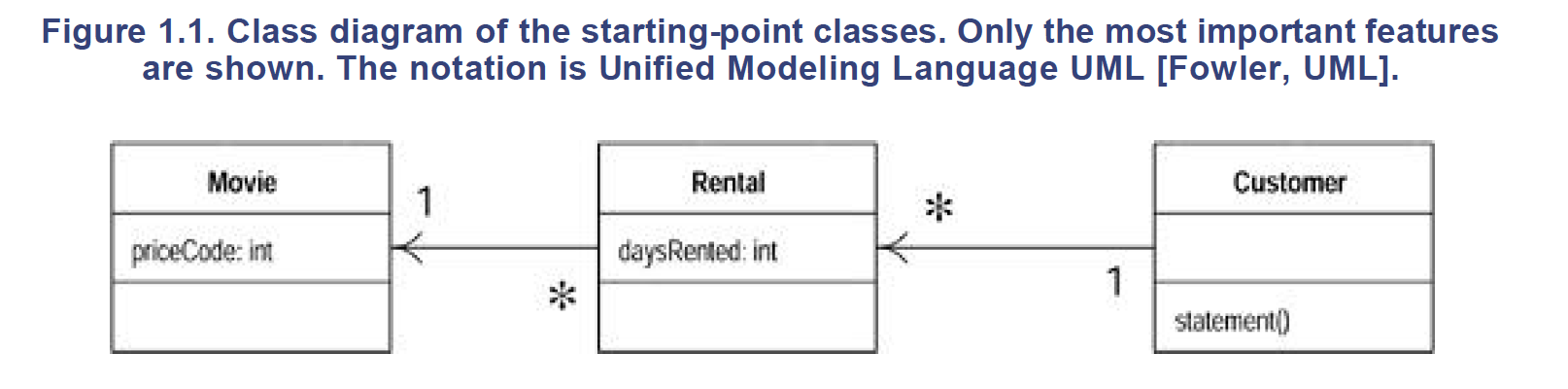
[REFERENCES 15](#_Toc66140692)

Chương 1 .Giới thiệu về Refactoring ( Tái cấu trúc )

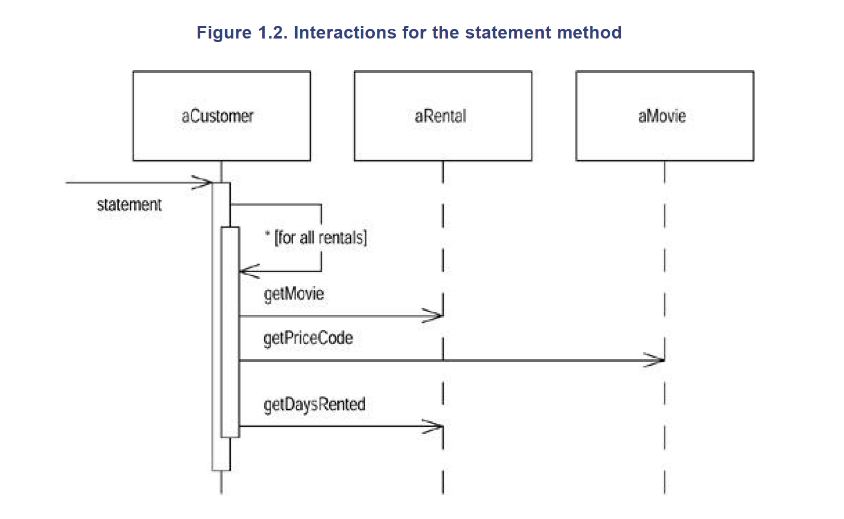
# 1.1. Ví dụ :

- Để nói về tái cấu trúc (refactoring), ta sẽ bắt đầu với chương trình đơn giản, đây là một chương trình dùng để tính toán và in trạng thái (statement) về chi phí của khách hàng tại một cửa hàng. Chương trình sẽ cho biết những bộ phim mà khách hàng đã thuê , xác định thời gian thuê. Sau đó sẽ tính các khoản phí nảy sinh tuỳ thuộc vào thời gian phim được thuê. Có 3 loại phim: Thường xuyên, trẻ em, mới phát hành.

- Đây là một class diagram để thể hiện chúng:



Với lần lượt là các lớp dữ liệu đơn giản của Movie, Rental và Customer. Customer cũng có method để tạo ra một statement. Figure 1.2 cho thấy các tương tác cho phương thức này.



- Nhận xét về chương trình, đây là một hệ thống có thể thay đổi hệ thống, trình biên dịch không quan tâm là tốt hay không tốt, và dễ thay đổi chương trình, nếu khó có thể tìm ra những gì cần thay đổi thì khả năng cao là lập trình viên sẽ mắc lỗi và tạo ra bug trong chương trình. Trong trường hợp này, có một thay đổi mà người dùng muốn thực hiện. Đầu tiên họ muốn method statement được in ra HTML. Hãy xem xét tác động mà sự thay đổi này sẽ có. Khi bạn nhìn vào mã, bạn có thể thấy rằng không thể sử dụng lại bất kì hành vi( behavior ) nào của method statement() hiện tại để in ra HTML. Cách duy nhất là viết một chương trình hoàn toàn mới, sao chép phần lớn để thay đổi các câu lệnh.

- Điều quan trọng là cần phải thay đổi cả statement và htmlStatêmnt để đảm bảo việc sửa lỗi là nhất quán. Giống như việc người dùng muốn thay đổi cách họ phân loại phim và sẽ ảnh hưởng đến người thuê trả tiền cho phim này và cách tính tiền. Phương thức statement là nơi cần phải được thay đổi để đối phó với các thay đổi trong quy tắc phân loại và tính phí. Tuy nhiên sao chép statement thành hàm htmlStatement thì cần đảm bảo mọi thay đổi là hoàn toàn phù hợp.

# 1.2. Khái niệm:

- Bước đầu tiên trong Refactoring ( tái cấu trúc ) là cần xây dựng một bộ test case vững chắc cho phần code được viết ra. Các test luôn là điều cần thiết bởi vì dù tuân theo các cấu trúc để tránh xảy ra bug thì vẫn sẽ rất dễ xảy ra việc mắc cái lỗi ( bug ). Do đó việc cần test case vững chắc là điều hoàn toàn cần thiết. Bởi vì kết quả của hàm statement tạo ra một chuỗi, tạo ra một vài khách hàng và cung cấp cho mỗi khách hàng một vài dịch vụ cho thuê các loại phim khác nhau và tạo ra chuỗi statement. Sau đó tôi thực hiện so sánh giữa chuỗi mới và một số chuỗi tham chiếu, thiết lập tất cả các test này để tôi có thể chạy chúng từ một lệnh Java command trên command line.

- Một phần quan trọng tiếp theo là cách nó báo cáo kết quả của nó. Điều quan trọng là làm cho các test tự kiểm tra. Nếu bạn không làm thế, cuối cùng bạn sẽ dành thời gian để kiểm tra test và điều đó khiến quá trình bị chậm lại.

# 1.3. Phân tách và phân phối lại phương thức statement

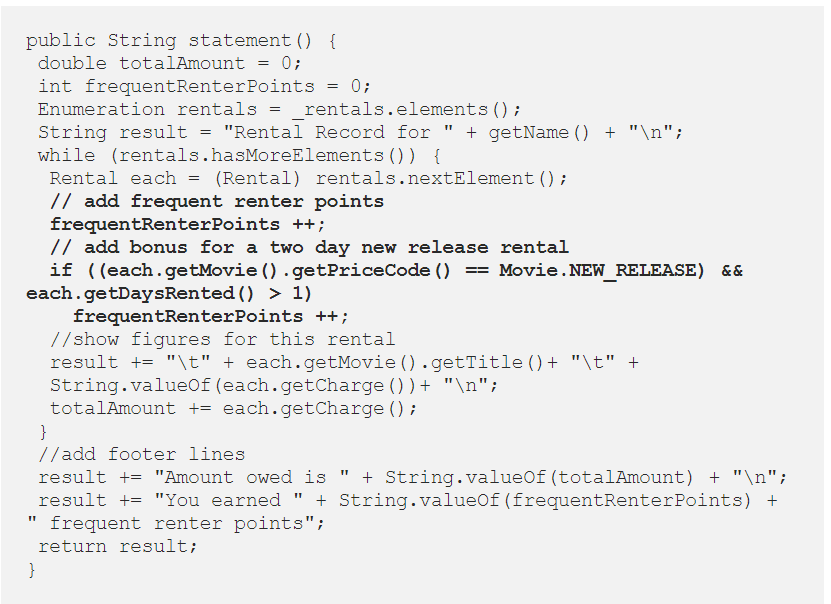
- Mục tiêu đầu tiên mà tôi chú ý rõ ràng là phương thức statement quá dài. Khi nhìn vào một phương thức dài như thế, tôi đang tìm cách phân tách phương thức đó thành các phần nhỏ hơn. Các đoạn mã nhỏ hơn có xu hướng làm cho mọi thứ dễ quản lý hơn. Mục đích của tôi là làm cho việc viết một phương thức statement in ra HTML dễ dàng hơn so với việc copy code.

- Bước đầu tiên của tôi là tìm một cụm mã hợp lý và sử dụng Extract Method (Trích xuất phương thức). Một mảnh rõ ràng ở đây là switch statement. Điều này có vẻ như nó sẽ tạo ra một đoạn tốt để trích xuất thành phương thức riêng của nó. Khi tôi trích xuất một phương thức, như trong bất kỳ tái cấu trúc nào, tôi cần biết điều gì có thể sai. Nếu trích xuất không tốt, nó có thể đưa lỗi vào chương trình. Vì vậy, trước khi thực hiện tái cấu trúc cần tìm ra cách thực hiện một cách an toàn.

- Ví dụ như khi viết chương trình bằng Java chuyển đổi double thành int tuy nhiên nó sẽ không báo lỗi, nhưng với một bộ test case tốt, việc tìm ra những lỗi nhỏ khá dễ dàng.

# 1.4. Trích xuất cách tính điểm để thành khách hàng thân thuộc.

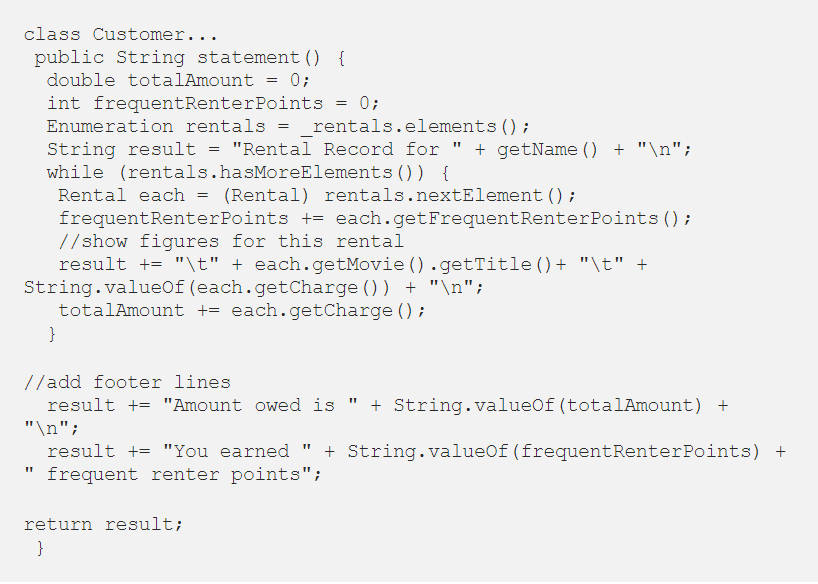
­- Bước tiếp theo là làm một điều tương tự cho việc tính điểm cho khách hàng thườn xuyên . Các quy tắc tính điểm cho khách hàng thường xuyên sẽ có sự thay đổi, mặc dù có ít thay đổi hơn so với charging. Có vẻ hợp lý để đặt trách nhiệm về Rental. Trước tiên, chúng ta cần sử dụng Extract Method trên phần frequent renter points ( In Đậm )



- Các biến tạm khác được sử dụng là frequentRenterPoints, trong trường hợp này, frequentRenterPoints có giá trị trước. Tuy nhiên, phần thân của phương thức trích xuất không đọc giá trị, vì vậy chúng ta không cần truyền nó vào dưới dạng tham số miễn là chúng ta sử dụng phép gán bổ sung.

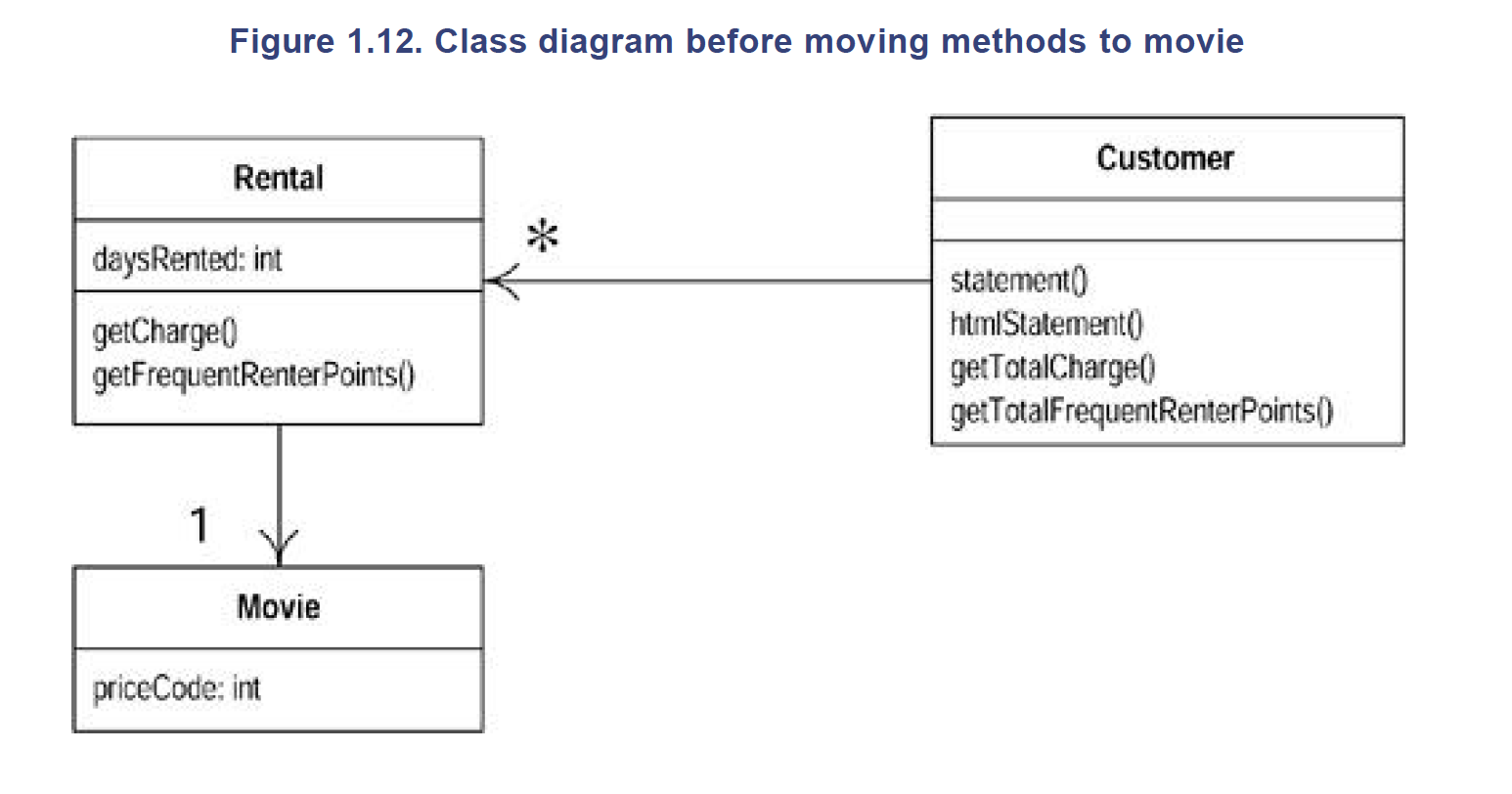
# 1.5. Xoá những biến tạm:

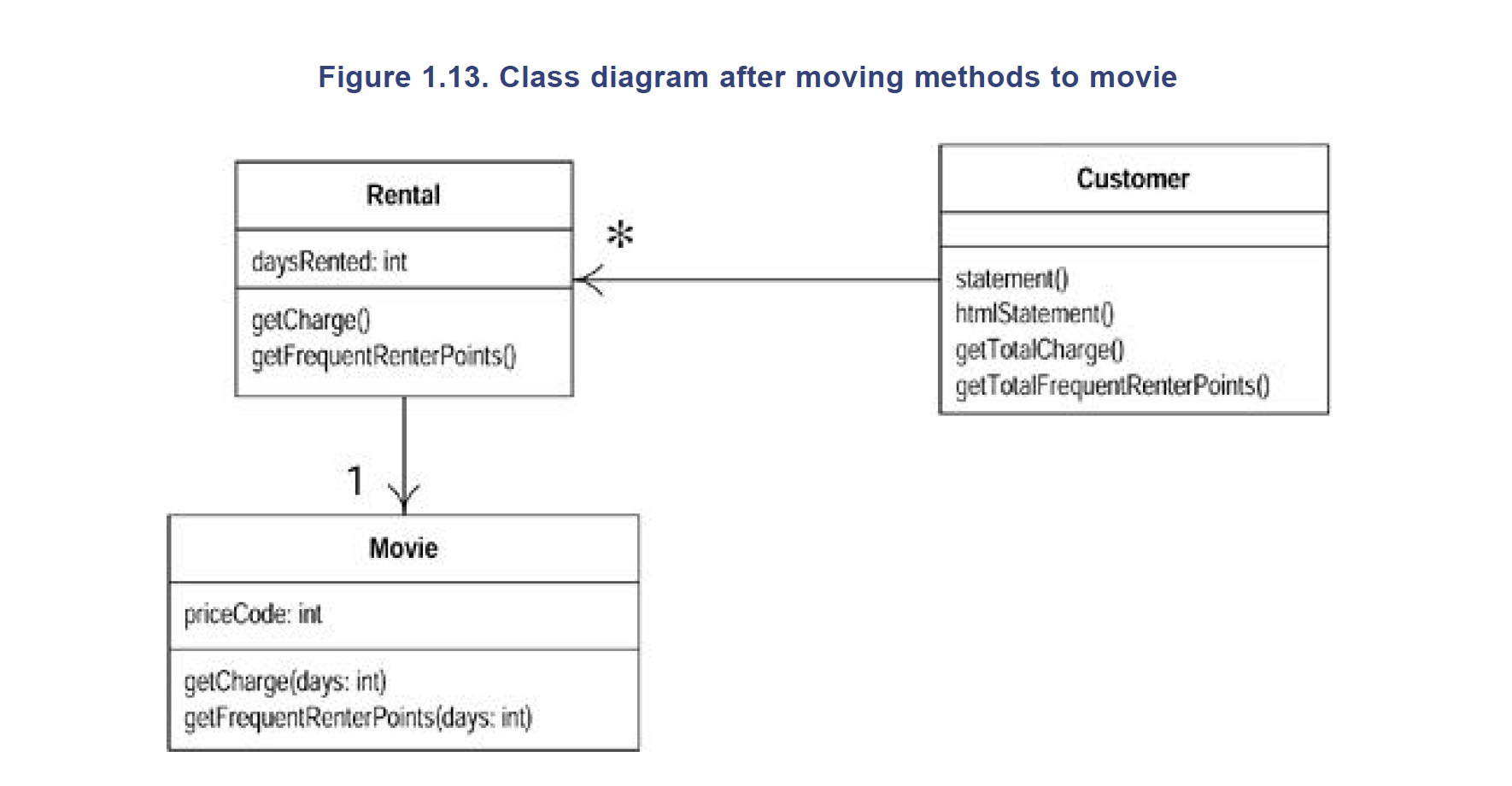
- Trong trường hợp này, chúng tôi có hai biến tạm thời, cả hai biến này đang được sử dụng để lấy tổng số tiền cho thuê gắn liền với khách hàng. Cả hai phiên bản ASCII và HTML đều yêu cầu các tổng số này. Nên sử dụng Replace Temp with Query để thay thế totalAmount và frequentRentalPoints bằng các phương thức truy vấn. Các truy vấn có thể truy cập được với bất kỳ phương thức nào trong lớp và do đó khuyến khích một thiết kế tốt hơn mà không cần các phương thức phức tạp và dài.

- Bằng cách trích xuất các phép tính, có thể tạo phương thức htmlStatement và sử dụng lại tất cả các mã tính toán có trong phương thức statement gốc. Nếu quy tắc tính toán thay đổi, chỉ cần sửa một vị trí trong mã. Bất kỳ loại statement nào khác cũng sẽ được thực hiện nhanh chóng và dễ dàng chuẩn bị. Việc tái cấu trúc không mất nhiều thời gian.

# 1.6. Thay thế tính toán Logic bằng tính đa hình

- Truyền tham số là thời gian thuê, tất nhiên đó là dữ liệu từ việc cho thuê. Method sử dụng hiệu quả hai mẫu dữ liệu, thời lượng cho thuê và loại phim. Những thay đổi được đề xuất là tất cả về việc thêm các loại mới. Biên dịch phương thức này vào class Movie và thay đổi getCharge trong class Rental.



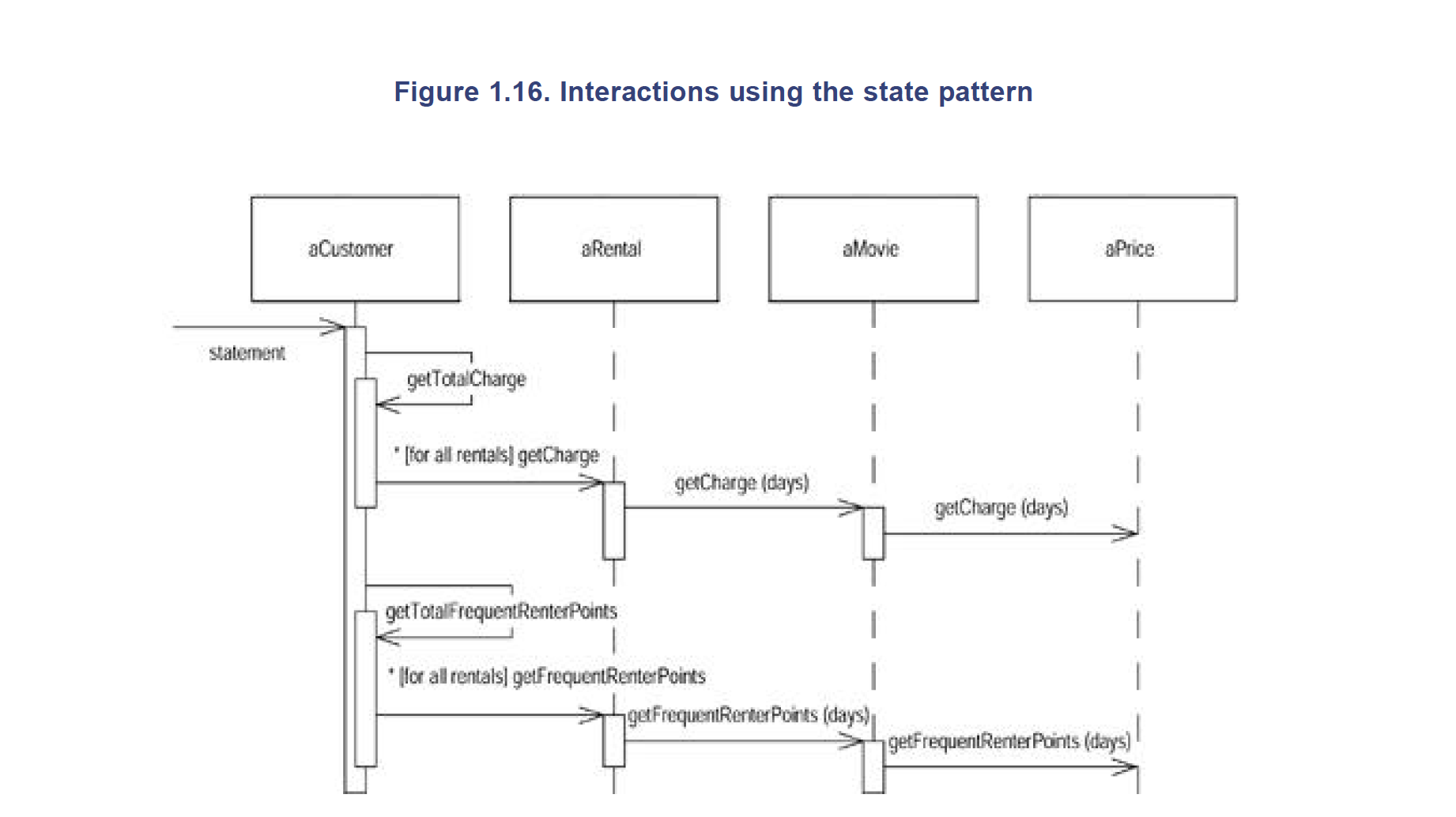


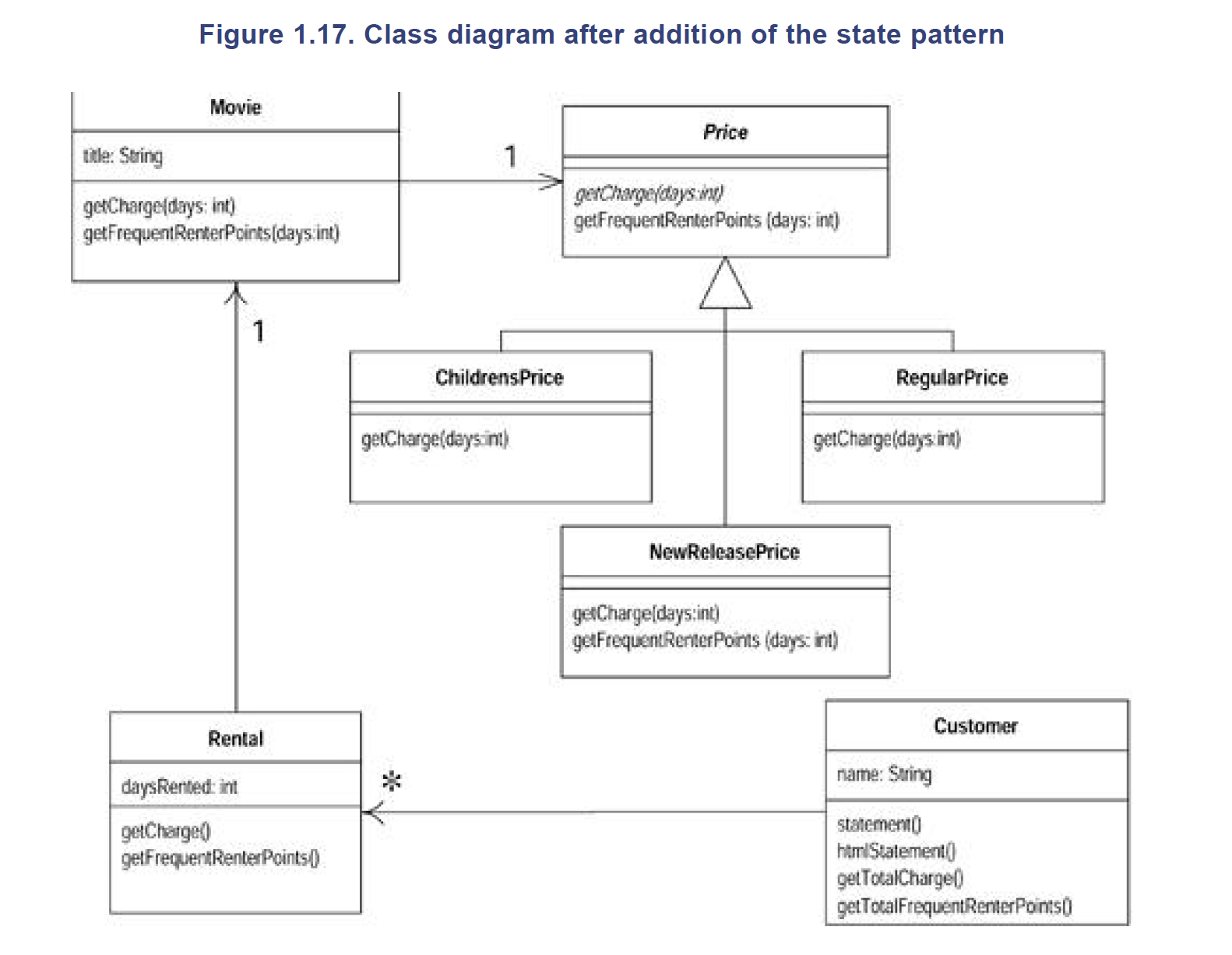
# 1.7. Thừa kế ở cuối

- Để giới thiệu State pattern tôi sẽ sử dụng ba phép tái cấu trúc. Trước tiên, tôi sẽ chuyển hành(behavior) vi mã loại vào State pattern bằng Replace Type Code with State/Strategy(Thay thế Mã loại bằng State/Strategy). Sau đó có thể sử dụng Move Method(Di chuyển phương thức) để di chuyển câu lệnh chuyển sang lớp Price. Cuối cùng là sử dụng Replace Conditional with Polymorphism(Thay thế có điều kiện bằng Đa hình) để loại bỏ câu lệnh switch.

- Đưa vào State pattern là một lợi ích , nếu thay đổi bất kỳ hành vi nào về giá, thêm giá mới hoặc thêm hành vi phụ thuộc vào giá, thay đổi sẽ dễ thực hiện hơn nhiều.

- Sẽ dễ dàng hơn nhiều để thay đổi cấu trúc phân loại của phim và thay đổi các quy tắc tính phí và hệ thống điểm người thuê phim thường xuyên. Hình 1.16 và 1.17 cho thấy State pattern hoạt động như thế nào với thông tin





# 1.8. Kết luận:

- Như đã nêu trên về ví dụ về việc tái cấu trúc (refactoring). Các số phép tái cấu trúc đã sử dụng: Extract Method, Move Method, and Replace Conditional with Polymorphism. Tất cả các phép trên đều giúp code được phân tối tốt và dễ dàng bảo trì hơn.

Chương 2. Các nguyên tắc tái cấu trúc

# 2.1. Xác định tái cấu trúc:

## Noun form:

Một thay đổi được thực hiện đối với cấu trúc bên trong của phần mềm giúp dễ hiểu hơn và chi phí sửa chửa thấp hơn mà không thay đổi hành vi có thể quan sát được của nó.

## Verb form:

Tái cấu trúc phần mềm bằng cách áp dụng một loạt các tái cấu trúc mà không thay đổi hành vi có thể quan sát được của nó.

## Tái cấu trúc để làm gì?

Tái cấu trúc làm cho mã phần mềm dễ hiểu và dễ sửa đổi hơn. Ngoài ra ta có thể thay đổi trong phần mềm một số lượng nhỏ hoặc thậm chí là không.

Tái cấu trúc không thay đổi các hàm các hành vi, phần mềm vẫn được thự hiện cùng chức năng như lúc ban đầu.

# 2.2. Tại sao lại phải tái cấu trúc:

## Tái cấu trúc cải thiện thiết kế phần mềm:

Nếu không tái cấu trúc thiết kế của chương trình sẽ bị rời rạc không có sự liên kết.

Khi thay đổi code — các thay đổi để thực hiện các mục tiêu trong khoảng thời gian ngắn, các thay đổi được thực hiện mà không có sự hiểu biết đầy đủ về thiết kế của của chương trình sẽ làm mất mất cấu trúc của nó. Và trở nên khó khăn hơn để xem thiết kế bằng cách đọc mã.

Mất cấu trúc của chương trình gây các tác động tích lũy. Càng khó nhìn thấy thiết kế trong chương trình, càng khó bảo quản và càng nhanh hỏng.

Tái cấu trúc thường xuyên giúp mã giữ nguyên thiết kế.

Tầm quan trọng của điều này nằm ở những sửa đổi trong tương lai đối với mã. Giảm số lượng mã sẽ không làm cho hệ thống chạy nhanh hơn, vì ảnh hưởng đến dấu vết của các chương trình hiếm khi là đáng kể. Tuy nhiên, giảm số lượng mã sẽ tạo ra sự khác biệt lớn trong việc sửa đổi mã. Càng nhiều mã ở đó nghĩa là càng khó sửa đổi một cách chính xác. Có nhiều mã hơn để hiểu.

Bạn thay đổi đoạn mã này ở đây, nhưng hệ thống không thực hiện những gì bạn mong đợi bởi vì bạn đã không thay đổi một chút ở đó mà làm nhiều điều tương tự trong một bối cảnh hơi khác. Bằng cách loại bỏ các bản sao, bạn đảm bảo rằng mã nói mọi thứ một lần và chỉ một lần, đó là bản chất của thiết kế tốt.

# Tái cấu trúc giúp phần mềm dễ hiểu hơn

Cấu trúc lại giúp làm cho mã của mình dễ đọc hơn. Khi cấu trúc lại, ta có mã hoạt động nhưng không có cấu trúc lý tưởng. Một chút thời gian dành cho việc tái cấu trúc có thể giúp mã truyền đạt tốt hơn mục đích của nó.

Tính dễ hiểu này cũng hoạt động theo một cách khác. Ta sử dụng cấu trúc lại để giúp ta hiểu mã không quen thuộc. Khi ta nhìn vào mã không quen thuộc, ta phải cố gắng hiểu nó làm gì. Thay đổi mã để phản ánh sự hiểu biết của mình tốt hơn và sau đó kiểm tra sự hiểu biết đó bằng cách chạy lại mã để xem liệu nó có còn hoạt động hay không.

# 2.3. Khi nào nên tái cấu trúc:

## Khi cần thêm tính năng mới Thực sự, việc cấu trúc lại code giúp chúng ta dễ dàng thêm chức năng mới, thử tưởng tượng, với mỗi chức năng mà ta thiết kế như một class và phân chia ra file, với mỗi chức năng sau này được thêm vào sẽ chỉ cần thêm file và thực thi trên class đó.

## Khi cần fix bug

Trong việc sửa lỗi, phần lớn việc sử dụng tái cấu trúc đến từ việc tạo mã dễ hiểu hơn. Thường thì tôi thấy rằng điều này hoạt động quy trình làm việc với mã giúp tìm ra lỗi. Nhìn vào một báo cáo lỗi, bạn sẽ dễ dàng nhận ra lỗi đó nằm ở chỗ nào và chỉ việc đến tệp tin đó và sữa chữa.

## Khi xem tổng quan code

Việc tái cấu trúc giúp xem lại mã của người khác hoặc muốn phát triển phần mềm với đội nhóm. Trước khi bắt đầu sử dụng cấu trúc lại, Ta có thể đọc mã, hiểu một số mức độ của nó, và đưa ra đề xuất.  
Để làm cho quá trình này hoạt động, bạn phải có các nhóm nhỏ đánh giá.  
Kinh nghiệm làm việc cho thấy có một người đánh giá và tác giả gốc làm việc trên mã với nhau. Với các đánh giá thiết kế lớn hơn, thường tốt hơn nếu có được một số ý kiến ​​trong một nhóm lớn hơn. Hiển thị mã thường xuyên không phải là thiết bị tốt nhất cho việc này.

# 2.4. Các vấn đề gặp phải khi tái cấu trúc:

Khi ta học một kỹ thuật mới giúp cải thiện đáng kể năng suất của ta, rất khó để nhìn thấy khi nó không áp dụng. Thông thường ta học nó trong một bối cảnh cụ thể, thường chỉ là một dự án duy nhất. Thật khó để xem những nguyên nhân kỹ thuật kém hiệu quả, thậm chí có hại. Refactoring là như vậy bây giờ. Chúng ta biết những lợi ích của việc tái cấu trúc.

## Cơ sở dữ liệu

Một vấn đề trong lĩnh vực tái cấu trúc là cơ sở dữ liệu. Hầu hết các ứng dụng kinh doanh được kết hợp chặt chẽ với lược đồ cơ sở dữ liệu hỗ trợ chúng. Đó là một lý do mà cơ sở dữ liệu rất khó thay đổi. Một lý do khác là di chuyển dữ liệu. Ngay cả khi đã phân lớp cẩn thận hệ thống của mình để giảm thiểu sự phụ thuộc giữa lược đồ cơ sở dữ liệu và mô hình đối tượng, việc thay đổi lược đồ cơ sở dữ liệu buộc bạn phải di chuyển dữ liệu, đây có thể là một nhiệm vụ lâu dài và đầy vất vả.

## Interface

Một trong những điều quan trọng về các đối tượng là chúng cho phép bạn thay đổi việc triển khai một mô-đun phần mềm một cách riêng biệt với việc thay đổi giao diện. Bạn có thể thay đổi bên trong một đối tượng một cách an toàn mà không cần ai khác phải lo lắng về nó, nhưng giao diện rất quan trọng - hãy thay đổi điều đó và bất cứ điều gì có thể xảy ra.  
Một điều đáng lo ngại về việc tái cấu trúc là nhiều lần tái cấu trúc thay đổi giao diện. Phương pháp là tất cả về việc thay đổi một giao diện.

2.5. Thiết kế:

Nếu bạn không tổ chức lại, sẽ có rất nhiều áp lực để thực hiện đúng thiết kế trả trước. Điều này có nghĩa là bất kỳ thay đổi nào sau này đối với thiết kế sẽ rất tốn kém. Do đó, bạn dành nhiều thời gian và nỗ lực hơn cho việc thiết kế trước để loại bỏ nhu cầu thay đổi như vậy.

Sự nhấn mạnh thay đổi với sự tái cấu trúc. Bạn vẫn thiết kế trước, nhưng bây giờ bạn không cố gắng tìm ra giải pháp. Thay vào đó, tất cả những gì bạn muốn là một giải pháp hợp lý. Bạn biết rằng khi bạn tạo ra một giải pháp, khi bạn tìm hiểu thêm về vấn đề, bạn sẽ nhận ra rằng giải pháp tốt nhất khác với những gì bạn đã tìm thấy ngay từ đầu. Với bất kỳ yêu cầu nào, tôi tự hỏi yêu cầu đó sẽ thay đổi như thế nào trong suốt thời gian tồn tại của hệ thống. Do thiết kế

thay đổi rất tốn kém, tôi sẽ tìm cách tạo ra thiết kế phù hợp với những thay đổi mà tôi có thể thấy trước. Vấn đề với việc tạo ra một giải pháp là sự linh hoạt về chi phí. Các giải pháp linh hoạt phức tạp hơn các giải pháp đơn giản. Phần mềm tạo ra khó bảo trì hơn bình thường, nhưng tôi nghĩ sẽ dễ dàng linh hoạt hơn theo hướng mà tôi nghĩ. Tất nhiên, sự thất vọng lớn nhất là tất cả sự linh hoạt này là không cần thiết. Có một số trong số họ, nhưng không thể đoán được đó là những bộ phận nào. Để có được sự linh hoạt, bạn phải cung cấp nhiều tính linh hoạt hơn mức bạn thực sự cần.

Với việc tái cấu trúc, bạn tiếp cận rủi ro thay đổi theo cách khác. Bạn vẫn đang suy nghĩ về những thay đổi tiềm ẩn, bạn vẫn đang nghĩ đến những giải pháp linh hoạt. Nhưng thay vì thực hiện các giải pháp linh hoạt này, bạn lại tự đặt câu hỏi: “Việc chuyển đổi một giải pháp đơn giản thành một giải pháp linh hoạt có thể khó khăn đến mức nào? »Nếu, như thường lệ, câu trả lời là" khá dễ dàng "thì bạn chỉ cần áp dụng các giải pháp đơn giản.

2.1. Hiệu suất:

Cấu trúc lại chắc chắn sẽ làm cho phần mềm hoạt động chậm hơn, nhưng nó cũng làm cho phần mềm hoạt động tốt hơn để điều chỉnh hiệu suất. Tôi đã thấy ba cách tiếp cận chung để viết phần mềm nhanh. Thành phần đó không được vượt quá ngân sách của nó, mặc dù một cơ chế trao đổi ngân sách thời gian được cho phép.

REFERENCES

1. <https://portal.mytum.de/wos/chrisitansFolder/Refactoring_improving_the_design_of_existing_code.pdf/>