TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

VIỆN CÔNG NGHỆ THÔNG TIN & TRUYỀN THÔNG

PROJECT 2

DESIGN PATTERNS

Giảng viên hướng dẫn: TS. Nguyễn Bá Ngọc
Họ và tên sinh viên: Trần Hữu Hiếu
MSSV: 20180078

Hà Tĩnh, ngày 24 tháng 7 năm 2020

Chương 1: Mô tả chung

1. Giới thiệu về mẫu thiết kế

Thiết kế phần mềm hướng đối tượng đã khó và thiết kế phần mềm hướng đối tượng có thể tái sử dụng còn khó hơn. Tại sao?. Vì phải tìm các đối tượng thích hợp, phân chia chúng thành các lớp ở mức độ chi tiết phù hợp, xác định các giao diện lớp và phân cấp kế thừa, đồng thời thiết lập các mối quan hệ chính giữa chúng. Thiết kế của bạn phải đáp ứng cụ thể cho vấn đề hiện tại nhưng cũng đủ tổng quát để giải quyết các vấn đề và yêu cầu trong tương lai. Vì nếu không, chúng ta sẽ phải thiết kế lại, và phải thay đổi rất nhiều. Các nhà thiết kế hướng đối tượng có kinh nghiệm sẽ cho biết rằng một thiết kế ngay lần đầu tiên có thể linh hoạt và là rất khó nếu không muốn nói là không thể trở nên "đúng". Do đó, để hoàn thành một thiết kế họ thường cố gắng tái sử dụng nhiều lần, và mỗi lần như thế sẽ sửa đổi và hoàn thiện.

Đối với các nhà thiết kế hướng đối tượng có kinh nghiệm, họ có khả năng tạo ra những thiết kế tốt. Trong khi đó, các nhà thiết kế nghiệp dư bị choáng ngợp bởi các tùy chọn có sẵn và có xu hướng quay trở lại thói quen cũ với các kỹ thuật hướng đối tượng mà họ đã sử dụng trước đây. Phải mất một thời gian dài để những người đó tìm hiểu xem làm thế nào là thiết kế tốt và xấu. Vậy sự khác nhau giữa những nhà thiết kế chuyên nghiệm và nghiệp dư là gì?.

Việc tự nhiên đầu tiên mà các nhà thiết kế tốt làm là cố gắng sử dụng lại các giải pháp đã từng hiệu quả với họ trong quá khứ. Khi họ tìm thấy một giải pháp tốt, họ sử dụng nó nhiều lần.. Kinh nghiệm như vậy là một phần của những gì làm cho họ trở thành chuyên gia. Do đó, bạn sẽ tìm thấy các mẫu lặp lại của các lớp và các đối tượng giao tiếp trong nhiều hệ thống hướng đối tượng. Các mẫu này giải quyết các vấn đề thiết kế cụ thể và làm cho các thiết kế hướng đối tượng linh hoạt hơn, dễ mở rộng hơn và cuối cùng là có thể **TÁI SỬ DỤNG.** Họ giúp các nhà thiết kế sử dụng lại các thiết kế thành công bằng cách dựa trên các thiết kế mới dựa trên kinh nghiệm trước đó. Một nhà thiết kế đã quen thuộc với các mẫu như vậy có thể áp dụng chúng ngay lập tức vào các vấn đề thiết kế mà không cần phải nghĩ hay tìm lại chúng.

Vậy mẫu thiết kế là gì?. Là một **GIẢI PHÁP** cho một **VẤN ĐỀ** trong một **NGỮ CẢNH CỤ THỂ.**

NGỮ CẢNH là tình huống mà chúng ta sử dụng mẫu thiết kế. Nó nên được lặp đi lặp lại

VẤN ĐỀ liên quan đến mục đích mà chúng ta cố gắng đạt được trong ngữ cảnh này, nó cũng liên quan đến các điều kiện rằng buộc xuất hiện trong ngữ cảnh.

GIẢI PHÁP là một mẫu thiết kể tổng quát mà bất kì ai cũng có thể áp dụng để giải quyết vấn đề và những rằng buộc xung quanh nó. Giải pháp này được khám phá ra chứ không phải phát minh.

Hầu như tất cả các pattern và nguyên lý thiết kế đều hướng đến giải quyết các vấn đề về sự thay đổi trong lĩnh vực phần mềm ngày nay. Nó cho phép một phần hệ thống có thể thay đổi và độc lập với các phần còn lại. Và chúng ta thường cố gắng tìm ra và đóng gói chúng.

2. Các loại design patterns

Có hai cách phân loại design pattern: Phân loại the mục đích sử dụng và phân loại theo hướng xử lý lớp hay đối tượng.

Theo mục đích thì có thể phân loại thành 3 nhóm:

- Creational Pattern (nhóm khởi tạo) gồm: Abstract Factory, Factory Method, Singleton, Builder, Prototype. Nó sẽ giúp bạn trong việc khởi tạo đối tượng, như bạn biết để khởi tạo bạn phải sử dụng từ khóa new, nhóm Creational Pattern sẽ sử dụng một số thủ thuật để khởi tạo đối tượng mà bạn sẽ không nhìn thấy từ khóa này. Ngoài ra nó cũng giúp tách khách hàng ra khỏi đối tượng cần được khởi tạo.

- Structural Pattern (nhóm cấu trúc) gồm: Adapter, Bridge, Composite, Decorator, Facade, Proxy và Flyweight.. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng, và bao gồm các đối tượng hay các lớp trong một cấu trúc lớn.

- Behavioral Pattern gồm: Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy và Visitor. Nhóm này dùng trong thực hiện các hành vi của đối tượng.

Theo hướng giải quyết với lớp hay đối tượng thì có 2 nhóm:

- Mẫu thiết kế lớp mô tả mối quan hệ của các lớp được định nghĩa qua nguyên lý kế thừa. Nó này được xác định vào thời gian biên dịch. Loại này có thể bao gồm Template Method, Adapter, Singleton, Factory Method…

- Mấu thiết kế đối tượng mô tả mối quan hệ giữa các đối tượng và được định nghĩa qua nguyên lý kết tập. Nó thường năng động và linh hoạt, được tạo ra vào thời gian chạy. Loại này có thể bao gồm: Decorator, Facade, Abstract Factory,…

Chúng ta có thể tóm tắt lại thành bảng như sau:

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

3. Các nguyên tắc thiết kế hướng đối tượng

1. Đóng gói những sự thay đổi.

Nói cách khác, nếu bạn có một số khía cạnh mà mã của bạn đang thay đổi, với mọi yêu cầu mới, thì bạn biết rằng bạn có một hành vi cần được kéo ra và tách biệt khỏi tất cả những thứ cố định.

Đây là một cách khác để suy nghĩ về nguyên tắc này: lấy các phần khác nhau và gói gọn chúng lại để sau này bạn có thể thay đổi hoặc mở rộng các phần thay đổi mà không ảnh hưởng đến những phần cố định. Một mẫu thiết kế kinh điển cho nguyên tắc này là Stradegy Pattern.

Khái niệm đơn giản này tạo cơ sở cho hầu hết mọi mẫu thiết kế. Tất cả các mẫu cung cấp một cách để cho phép một số phần của hệ thống thay đổi độc lập với tất cả các phần khác.

2. Nên ưu tiên nguyên lý kế tập hơn nguyên lý kế thừa.

Việc tạo hệ thống bằng cách sử dụng kết tập mang lại cho chúng ta nhiều khả năng hơn. Nó không chỉ cho phép bạn đóng gói một nhóm thuật toán vào tập hợp các lớp của riêng chúng, mà còn cho phép bạn thay đổi hành vi trong thời gian chạy miễn là đối tượng bạn đang soạn thảo cài đặt chính xác giao diện hành vi.

Kết tập được sử dụng trong nhiều mẫu thiết kế và có nhiều ưu và nhược điểm.

3. Program to an interface, not an implementation.

Chúng tôi sẽ sử dụng giao diện để đại diện cho từng hành vi và mỗi lần tạo ra một hành vi thì sẽ cài đặt một trong những giao diện đó.

Giao diện từ dễ bị nhầm lẫn ở đây. Có khái niệm về giao diện về nhưng cũng có giao diện cấu trúc Java. Bạn có thể lập trình với một giao diện mà không cần phải thực sự sử dụng một giao diện Java. Vấn đề là làm sao khai thác tính đa hình bằng cách lập trình cho

một kiểu cha để đối tượng trong thời gian không bị khoá bởi mã nguồn. Và chúng ta có thể diễn đạt "kiểu được khai báo của các biến phải là supertype, thường là một lớp trừu tượng hoặc giao diện, để các đối tượng được gán cho các biến đó có thể là bất kỳ cài đặt/ thuật toán cụ thể nào của supertype, có nghĩa là lớp khai báo chúng không cần phải biết về các kiểu đối tượng thực tế! "

Dưới đây là một ví dụ đơn giản về việc sử dụng kiểu đa hình - hãy tưởng tượng một lớp trừu tượng Động vật, với hai cách cài đặt cụ thể, Chó và Mèo.

Lập trình theo cài đặt có thể là:

```
Cat c = new cat();
Cat.meo();
```

Trong khi đó lập trình theo giao diện có thể là:

```
Animal animal = new Cat();
Animal.makeSound();
```

. Trong thời gian chạy: a = getAnimal(); a. makesound();

Chúng ta không biết kiểu của Animal là gì, chúng ta chỉ cần quan tâm nó biết đáp lại thông điệp makeSound().

4. Loose coupling.

Low coupling, loose coupling có nghĩa là các component ít phụ thuộc vào nhau, sự thay đổi trong component này ít khi, hoặc không ảnh hưởng đến component kia. Ngược lại, high coupling và tight coupling cho thấy các component phụ thuộc nhiều vào nhau, khi thay đổi 1 component thì các component kia đều bị ảnh hưởng và có khả năng phải thay đổi theo. Tất nhiên, low coupling là mục tiêu chúng ta cần hướng đến để đảm bảo cho hệ thống ít bị ảnh hưởng khi có thay đổi và do đó, tăng tốc độ thực hiện công việc và bảo trì.

Pattern đại diện nguyên tắc này là Oberser pattern.

5. The open-closed Principle

Chúng ta nên mở các lớp khi mở rộng. Lúc đó có thể thoải mái mở rộng lớp mà với những hành vi mới khi yêu cầu của bạn thay đôi.

Chúng ta nên đóng các lớp khi sửa đổi. Lúc đó, chúng ta sẽ dành rất nhiều thời gian để làm để debug và làm cho mã code đúng, vì vậy chúng ta không để người khác thay đổi.

Mặc dù có vẻ mâu thuẫn, nhưng có những kỹ thuật cho phép mở rộng mã mà không cần sửa đổi trực tiếp

Hãy cẩn thận khi chọn các vùng mã cần được mở rộng; áp dụng nguyên tắc Đóng mở MỌI NƠI là lãng phí, không cần thiết và có thể dẫn đến mã phức tạp, khó hiểu.

Một Pattern tuân thủ theo nguyên tắc này là Decorator Pattern.

6. The dependency Inversion Principle

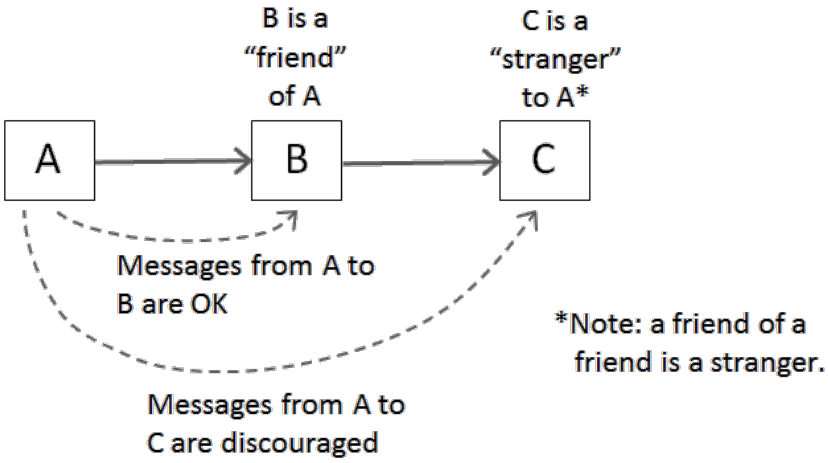
Nguyên lý này có nghĩ là phụ thuộc vào giao diện, không phụ thuộc vào các lớp riêng biệt.

Đầu tiên, nguyên tắc này nghe giống như nguyên tắc 4 phải không? Nó cũng tương tự như vậy; tuy nhiên, Nguyên tắc này đưa ra một tuyên bố mạnh mẽ hơn về tính trừu tượng. Nó gợi ý rằng các component cấp cao của chúng ta không nên phụ thuộc vào các componet cấp thấp của chúng ta; thay vào đó, cả hai đều nên phụ thuộc vào những phần trừu tượng. Tất nhiên, nó giúp chúng ta giảm sự phụ thuộc giữa lớp với nhau.

Tại sao lại có từ " Inversion"?. Là bởi vì nó đảo ngược cách bạn thường nghĩ về thiết kế OO của mình.

MỘt mẫu thiết kế điển hình cho nguyên lý này là Abstract Factory.

7. The principle of least Knowledge.(Law of Demeter).



Nguyên tắc trên hướng dẫn chúng ta giảm bớt sự tương tác giữa các đối tượng xuống chỉ với một vài "người bạn" thân thiết. Khi bạn thiết kế một hệ thống, cho bất kỳ đối tượng nào, hãy cẩn thận về số lượng các lớp mà nó tương tác và cũng như cách nó tương tác với các lớp đó.

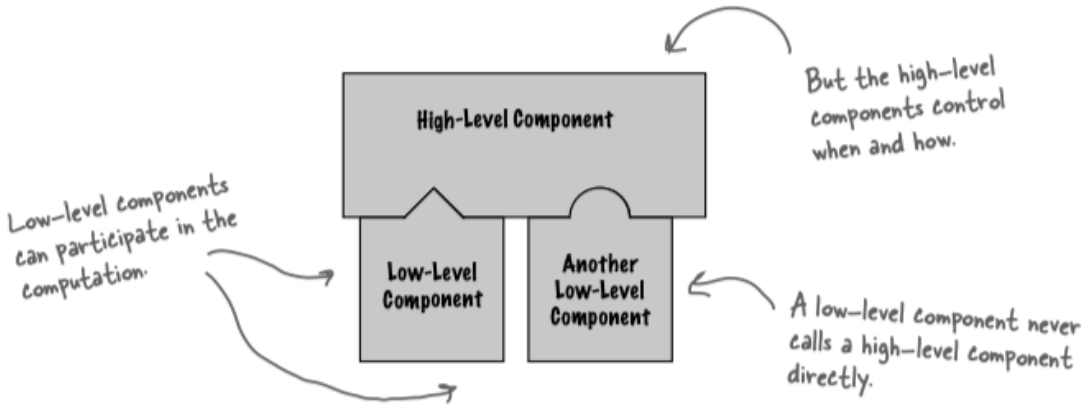
Nguyên tắc này ngăn cản chúng ta tạo ra các thiết kế có một số lượng lớn các lớp được ghép nối với nhau khiến cho việc thay đổi một phần của hệ thống ảnh hưởng đến các phần khác(loose coupling). Khi bạn xây dựng nhiều dependencies giữa nhiều lớp, bạn đang xây dựng một hệ thống dễ vỡ khi tốn kém để duy trì và phức tạp để những người khác hiểu.

8. The hollywood Principle: Đừng gọi cho chúng tôi, chúng tôi sẽ gọi cho bạn.

Dễ nhớ, phải không? Nhưng nó liên quan gì đến thiết kế OO?

Nguyên tắc Hollywood cung cấp cho chúng ta một cách để ngăn chặn Sự phụ thuộc. Điều này xảy ra khi bạn có các component cấp cao phụ thuộc vào các component cấp thấp. Khi "Dependency rot" xuất hiện, không ai có thể dễ dàng hiểu được cách một hệ thống được thiết kế.

Với Nguyên tắc Hollywood, chúng tôi cho phép các components cấp thấp tự kết nối với nhau thành một hệ thống, nhưng các thành phần cấp cao sẽ xác định khi nào components cấp thấp đó cần thiết và như thế nào. Nói cách khác, các thành phần cấp cao mang lại cho các thành phần cấp thấp cách xử lý "đừng gọi cho chúng tôi, chúng tôi sẽ gọi cho bạn"



Một pattern điển hình cho nguyên lý này là Template method.

9. Single Responsibility: Một lớp chỉ nên có một lý do để thay đổi.

Chúng ta biết rằng chúng ta muốn tránh thay đổi trong một lớp như bệnh dịch - mã đang sửa đổi có thể khiến cho chương trình xảy ra vấn đề . Có hai cách để thay đổi sẽ làm tăng xác suất lớp đó sẽ thay đổi trong tương lai và khi nó xảy ra, nó sẽ ảnh hưởng đến hai khía cạnh của thiết kế của bạn.

Giải pháp? Nguyên tắc trên hướng dẫn chúng ta giao mỗi trách nhiệm cho một lớp, và chỉ đúng một lớp.

Đúng vậy, thật dễ dàng như vậy, nhưng lại không: tách trách nhiệm trong thiết kế là một trong những việc khó làm nhất. Bộ não của chúng ta quá giỏi trong việc nhìn thấy một tập hợp các hành vi và nhóm chúng lại với nhau ngay cả khi thực sự có hai hoặc nhiều trách nhiệm. Cách duy nhất để thành công là siêng năng kiểm tra các thiết kế của bạn và để ý các tín hiệu cho thấy một lớp đang thay đổi theo nhiều cách khi hệ thống của bạn phát triển.

Ngoài các nguyên tắc kinh điển trên chúng ta còn có một số nguyên tắc khác (có thể phát triển từ các nguyên tắc trên) như Boy Scout Rule, Explicit Dependencies, YAGNI, Liskov Substitution Principle, SOLID(tổ hợp của 5 nguyên tắc), State Dependency; tell, Don’t Ask.

CHƯƠNG 2: CÁC MẪU THIẾT KẾ

1. Stradegy Pattern

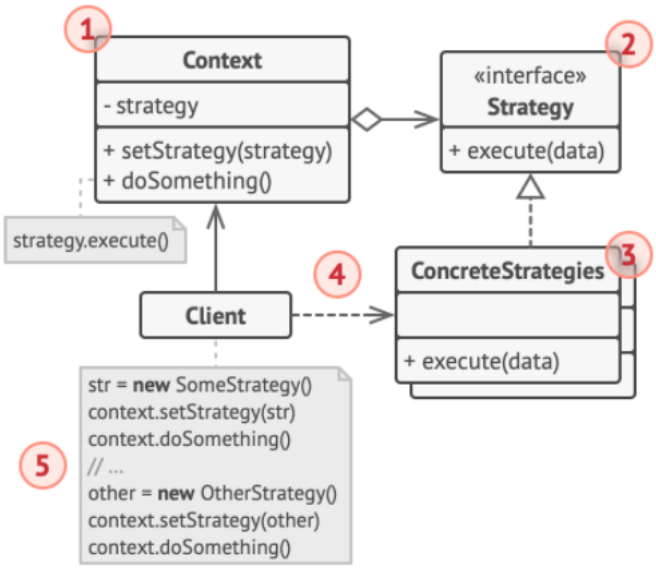
Có một vài trường hợp, các lớp chỉ khác nhau về hành vi của chúng. Trong trường hợp như vậy, ý tưởng tốt là chúng ta sẽ tách biệt các hành vi đó trong các lớp riêng biệt để có khả năng lựa chọn các thuật toán khác nhau trong thời gian chạy (run-time). Ý tưởng này được gọi là Strategy Pattern, một pattern giúp chúng ta giải quyết vấn đề về sự thay đổi, tương tự như State design pattern.

1.1 Định nghĩa

Mô hình Chiến lược là một họ các thuật toán, đóng gói từng thuật toán và làm cho chúng có thể hoán đổi cho nhau. Chiến lược cho phép thuật toán thay đổi độc lập với các khách hàng sử dụng nó

Ý nghĩa thực sự của Strategy Pattern là giúp tách rời phần xử lý một chức năng cụ thể ra khỏi đối tượng (Principle 1). Sau đó tạo ra một tập hợp các thuật toán để xử lý chức năng đó và lựa chọn thuật toán nào mà chúng ta thấy đúng đắn nhất khi thực thi chương trình. Mẫu thiết kế này thường được sử dụng để thay thế cho sự kế thừa, khi muốn chấm dứt việc theo dõi và chỉnh sửa một chức năng qua nhiều lớp con.

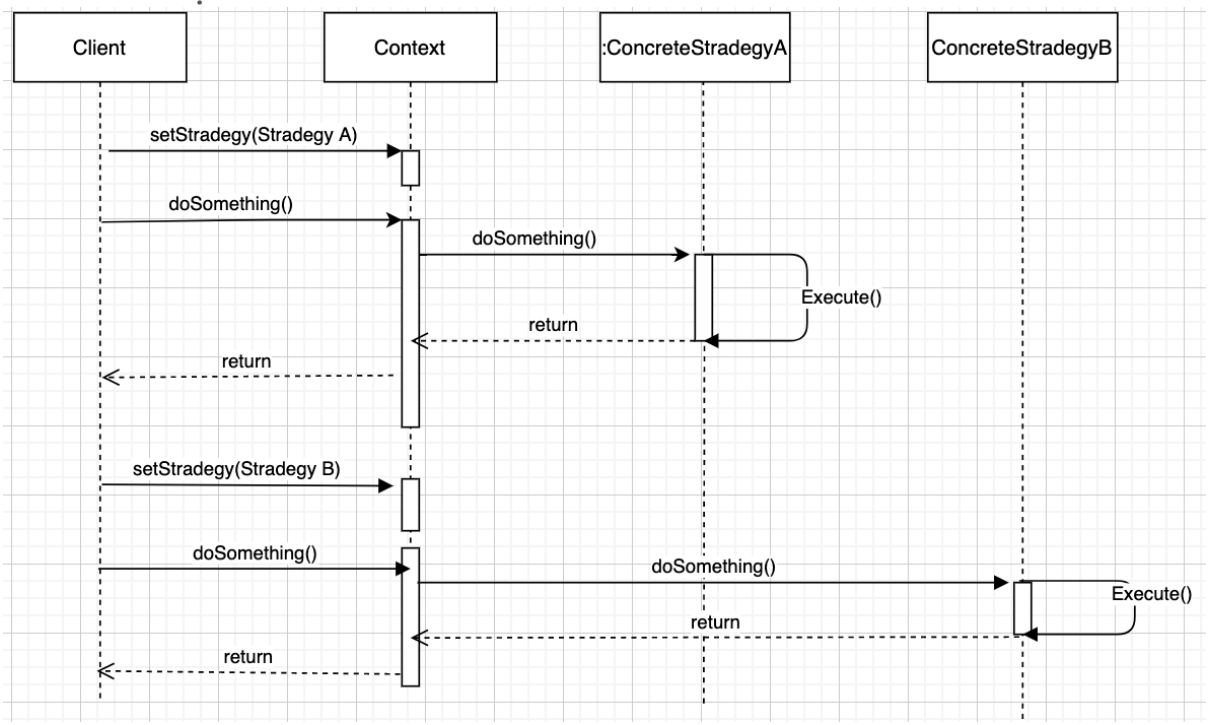
1.2 Cấu trúc mẫu thiết kế.



- **Context** duy trì tham chiếu đến một trong những chiến lược cụ thể và chỉ giao tiếp với đối tượng này thông qua giao diện chiến lược(Design Principle 3).
- **Stradegy** là giao diện chung cho tất cả các chiến lược cụ thể. Nó khai báo một phương thức mà **Context** sử dụng để thực thi một chiến lược.

- **ConcreteStradegies**: Context gọi phương thức thực thi trên đối tượng chiến lược được liên kết mỗi khi nó cần chạy thuật toán. Context không biết nó hoạt động với loại chiến lược nào hoặc thuật toán được thực thi như thế nào.

- **Biểu đồ tuần tự:**



1.3 Sử dụng Stradegy khi nào?.

Sử dụng mẫu Chiến lược khi bạn muốn sử dụng các phương án khác nhau của thuật toán trong một đối tượng và có thể chuyển từ thuật toán này sang thuật toán khác trong thời gian chạy. Mẫu Chiến lược cho phép bạn gián tiếp thay đổi hành vi của đối tượng trong thời gian chạy bằng cách liên kết nó với các đối tượng khác nhau có thể thực hiện các nhiệm vụ cụ thể theo những cách khác nhau.

Sử dụng mẫu Chiến lược khi bạn có nhiều lớp giống nhau chỉ khác nhau về cách chúng thực hiện một số hành vi. Mẫu Chiến lược cho phép bạn trích xuất các hành vi khác nhau thành một hệ thống phân cấp lớp riêng biệt và kết hợp các lớp ban đầu thành một, do đó giảm mã trùng lặp.

Sử dụng mẫu này để tách biệt logic nghiệp vụ của một lớp khỏi các chi tiết triển khai của các thuật toán có thể không quan trọng trong ngữ cảnh của logic đó. Mẫu Chiến lược cho phép bạn tách biệt mã, dữ liệu nội bộ và các phần phụ thuộc của các thuật toán khác nhau khỏi phần còn lại của mã(Principle 1). Các ứng dụng khách khác nhau có được một giao diện đơn giản để thực thi các thuật toán và chuyển đổi chúng trong thời gian chạy.

Sử dụng mẫu này khi lớp của bạn có một toán tử điều kiện lớn chuyển đổi giữa các phương án khác nhau của cùng một thuật toán. Mẫu chiến lược cho phép bạn loại bỏ điều kiện như vậy bằng cách trích xuất tất cả các thuật toán thành các lớp riêng biệt, tất cả đều triển khai cùng một giao diện. Đối tượng ban đầu ủy quyền việc

thực thi cho một trong những đối tượng này, thay vì triển khai tất cả các phương án của thuật toán.

1.4 Ưu điểm và nhược điểm.

Ưu điểm:

- Bạn có thể hoán đổi các thuật toán được sử dụng bên trong một đối tượng trong thời gian chạy.
- Bạn có thể tách các chi tiết triển khai của một thuật toán khỏi mã sử dụng nó.
- Bạn có thể thay thế kế thừa bằng thành phần.
- Đản bảo Open-closed principle. Bạn có thể sử dụng các chiến lược mới mà không cần phải thay đổi Context.
- Đảm bảo nguyên tắc Single responsibility : Trong trường hợp một lớp định nghĩa nhiều hành vi và chúng xuất hiện dưới dạng với nhiều câu lệnh có điều kiện. Thay vì nhiều điều kiện, chúng ta sẽ chuyển các nhánh có điều kiện liên quan vào lớp Strategy riêng lẻ của nó (State Pattern cũng có khá giống thế này).
- Đảm bảo các nguyên tắc khác như Favor composition over Inheritence, program to interface, not implentation

Nhược điểm:

- Nếu bạn chỉ có một vài thuật toán và chúng hiếm khi thay đổi, thì không nên sử dụng mẫu này
- Khách hàng phải nhận thức được sự khác biệt giữa các chiến lược để có thể chọn một chiến lược phù hợp.
- Rất nhiều ngôn ngữ lập trình hiện đại có hỗ trợ kiểu hàm cho phép bạn triển khai các phiên bản khác nhau của thuật toán bên trong một tập hợp các hàm ẩn danh. Sau đó, bạn có thể sử dụng các chức năng này chính xác như bạn đã sử dụng các đối tượng chiến lược, nhưng không làm tăng mã của bạn với các lớp và giao diện bổ sung.
-

1.5 Mối quan hệ với các Pattern khác.

- State, Strategy (và ở một mức độ nào đó là Adapter) có cấu trúc khá giống nhau. Thật vậy, tất cả các mẫu này đều dựa trên kết tập, tức là ủy thác công việc cho các đối tượng khác. Tuy nhiên, tất cả chúng đều giải quyết các vấn đề khác nhau.

- Command và Stradegy có thể trông giống nhau vì bạn có thể sử dụng cả hai để tham số hóa một đối tượng bằng một số hành động. Tuy nhiên, hai mẫu lại có mục đích rất khác nhau.

    - Bạn có thể sử dụng Command để chuyển đổi bất kỳ thao tác nào thành một đối tượng. Các tham số của hoạt động trở thành các trường của đối tượng đó. Việc chuyển đổi cho phép bạn trì hoãn việc thực hiện thao tác, xếp hàng đợi, lưu trữ lịch sử các lệnh, gửi lệnh đến các dịch vụ từ xa, v.v.

    - Mặt khác, Strategy thường mô tả các cách khác nhau để thực hiện cùng một việc, cho phép bạn hoán đổi các thuật toán này trong một lớp ngữ cảnh duy nhất

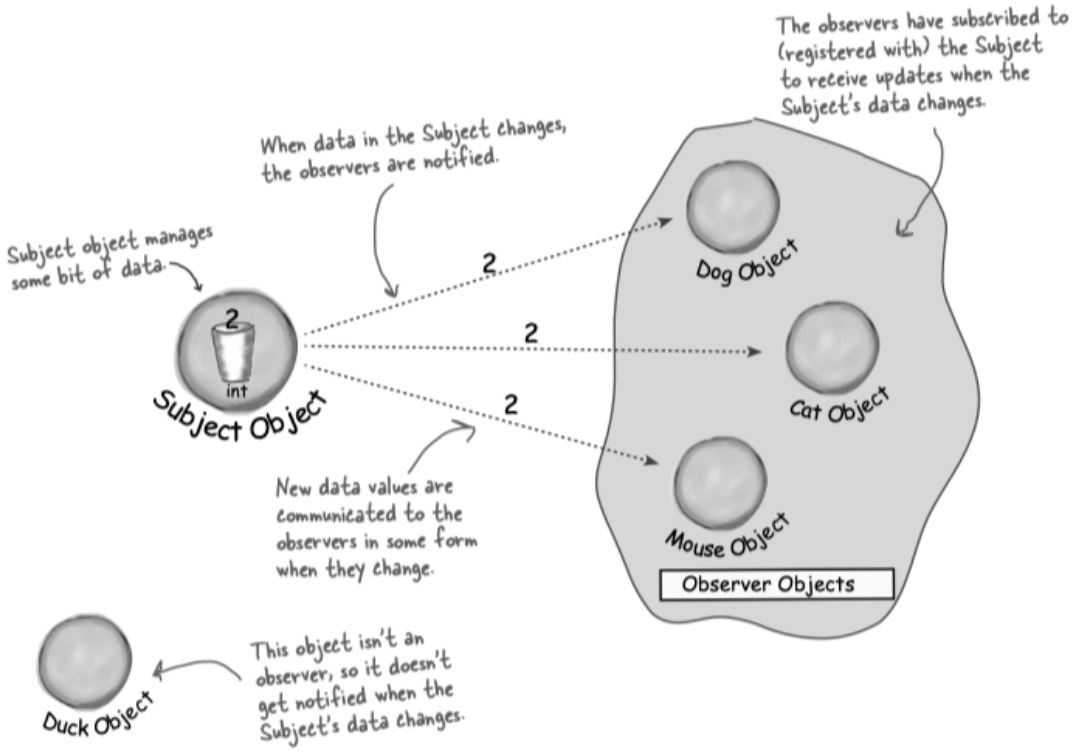
- Template method dựa trên sự kế thừa: nó cho phép bạn thay đổi các phần của một thuật toán bằng cách cài đặt các phần đó trong các lớp con. Stradegy dựa trên kết tập: bạn có thể thay đổi các phần trong hành vi của đối tượng bằng cách cung cấp cho đối tượng các chiến lược khác nhau tương ứng với hành vi đó. Tempate method hoạt động ở cấp lớp, vì vậy nó là tĩnh. Chiến lược hoạt động ở cấp độ đối tượng, cho phép bạn chuyển đổi hành vi trong thời gian chạy nên rất linh hoạt.

2. The observer Pattern

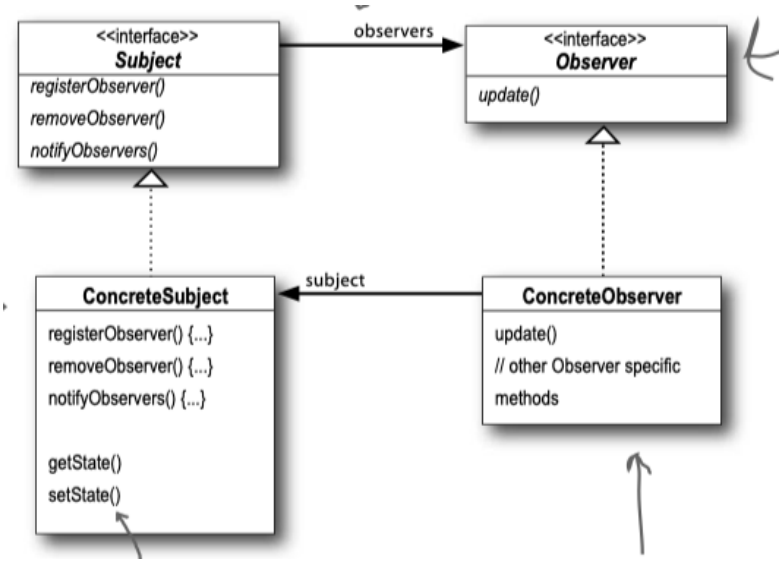
Chúng ta không thể nói về OOP mà không xem xét trạng thái của các đối tượng. Tất cả các chương trình hướng đối tượng là về các đối tượng và sự tương tác của chúng. Trong trường hợp khi một số đối tượng nhất định cần được thông báo thường xuyên về những thay đổi xảy ra trong các đối tượng khác.. Mẫu thiết kế **Observer** (quan sát) có thể được sử dụng bất cứ khi nào mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.

2.1 Định nghĩa.

Observer Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó định nghĩa mối phụ thuộc **một** – **nhiều** giữa các đối tượng để khi mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.
Có thể mô tả như sau:



2.2 Cấu trúc

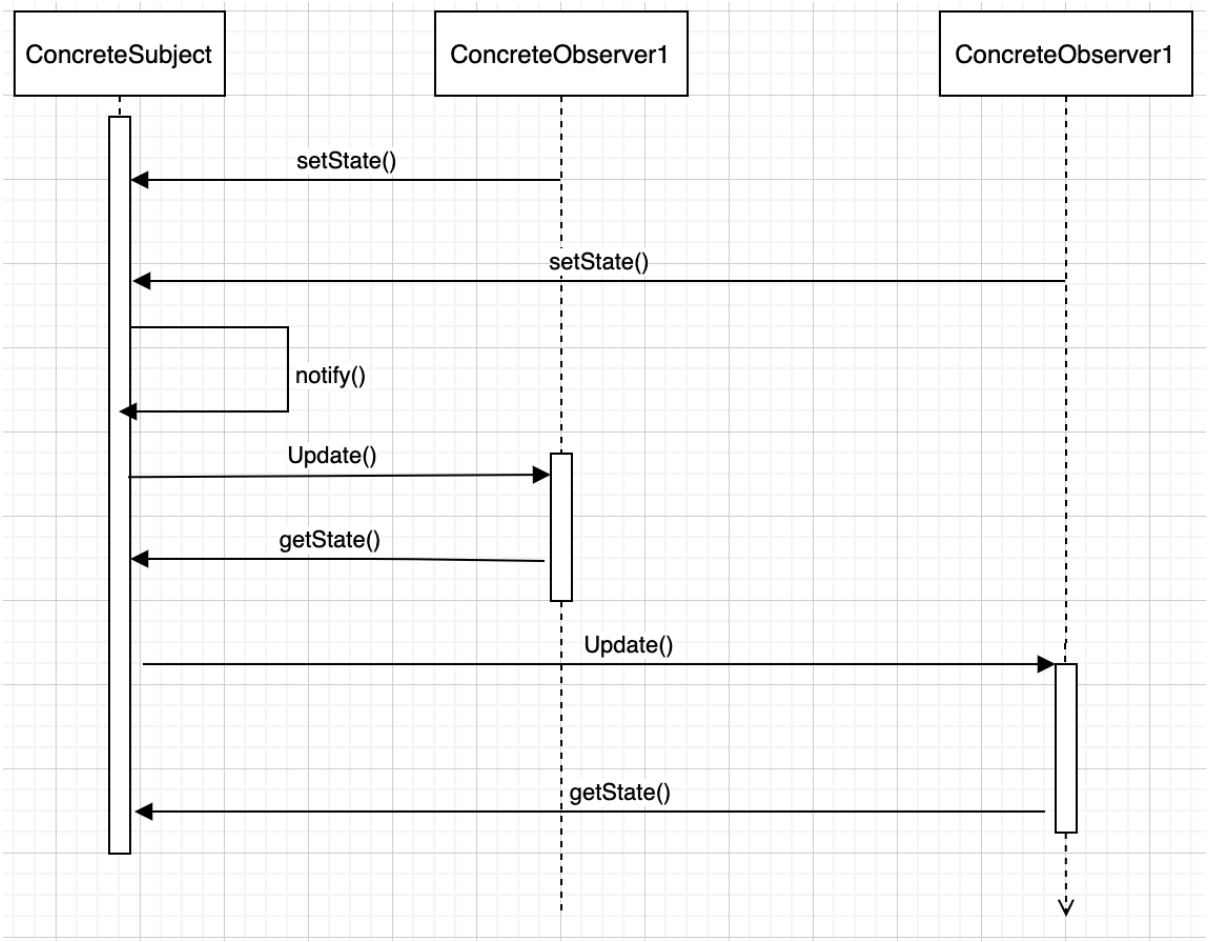


Các thành phần tham gia Observer Pattern:

- **Subject** : chứa danh sách các observer, cung cấp phương thức để có thể đăng ký và huỷ đăng ký observer.
- **Observer** : định nghĩa một phương thức update() cho các đối tượng sẽ được subject thông báo đến khi có sự thay đổi trạng thái.
- **ConcreteSubject** : cài đặt các phương thức của Subject, lưu trữ trạng thái danh sách các ConcreateObserver, gửi thông báo đến các observer của nó khi có sự thay đổi trạng thái.
- **ConcreteObserver** : cài đặt các phương thức của Observer, lưu trữ trạng thái của subject, thực thi việc cập nhật để giữ cho trạng thái đồng nhất với subject gửi thông báo đến.

Sự tương tác giữa subject và các observer như sau: mỗi khi subject có sự thay đổi trạng thái, nó sẽ duyệt qua danh sách các observer của nó và gọi phương thức cập nhật trạng thái ở từng observer, có thể truyền chính nó vào phương thức để các observer có thể lấy ra trạng thái của nó và xử lý.

- **Biểu đồ tuần tự:**



2.3 Khi nào sử dụng Observer Pattern?.

- Sử dụng mẫu Observer khi các thay đổi trạng thái của một đối tượng có thể yêu cầu thay đổi các đối tượng khác và tập đối tượng thực tế không được biết trước hoặc thay đổi động.

- Mẫu Observer cho phép bất kỳ đối tượng nào triển khai giao diện người đăng ký đăng ký nhận thông báo sự kiện trong đối tượng nhà xuất bản. Bạn có thể thêm cơ chế đăng ký vào các nút của mình, cho phép khách hàng kết nối mã tùy chỉnh của họ thông qua các lớp người đăng ký tùy chỉnh.

- Sử dụng mẫu khi một số đối tượng trong ứng dụng của bạn phải quan sát những đối tượng khác, nhưng chỉ trong thời gian giới hạn hoặc trong các trường hợp cụ thể. Danh sách đăng ký là động, vì vậy người đăng ký có thể tham gia hoặc rời khỏi danh sách bất cứ khi nào họ cần.

- Khi thay đổi một đối tượng, yêu cầu thay đổi đối tượng khác và chúng ta không biết có bao nhiêu đối tượng cần thay đổi, những đối tượng này là ai.

- Sử dụng trong ứng dụng broadcast-type communication.

- Sử dụng để quản lý sự kiện (Event management).

- Sử dụng trong mẫu mô hình MVC (Model View Controller Pattern) : trong MVC, mẫu này được sử dụng để tách Model khỏi View. View đại diện cho Observer và Model là đối tượng Observable.

2.4 Ưu nhược điểm

Ưu điểm:

- Dễ dàng mở rộng với ít sự thay đổi : mẫu này cho phép thay đổi Subject và Observer một cách độc lập. Chúng ta có thể tái sử dụng các Subject mà không cần tái sử dụng các Observer và ngược lại. Nó cho phép thêm Observer mà không sửa đổi Subject hoặc Observer khác. Vì vậy, nó đảm bảo nguyên tắc Open/Closed Principle (OCP).
- Mình có thể thiết luật mối quan hệ giữa các đối tượng lúc Run-Time.
- Sự thay đổi trạng thái ở 1 đối tượng có thể được thông báo đến các đối tượng khác mà không phải giữ chúng liên kết quá chặt chẽ.
- Một đối tượng có thể thông báo đến một số lượng không giới hạn các đối tượng khác.

Nhược điểm:

- Observer được thông báo một cách ngẫu nhiên.

2.5 Mối quan hệ với các Pattern khác:

- Command lệnh thiết lập các kết nối một chiều giữa người gửi và người nhận.
- Observer cho phép người nhận đăng ký động và hủy đăng ký nhận yêu cầu.

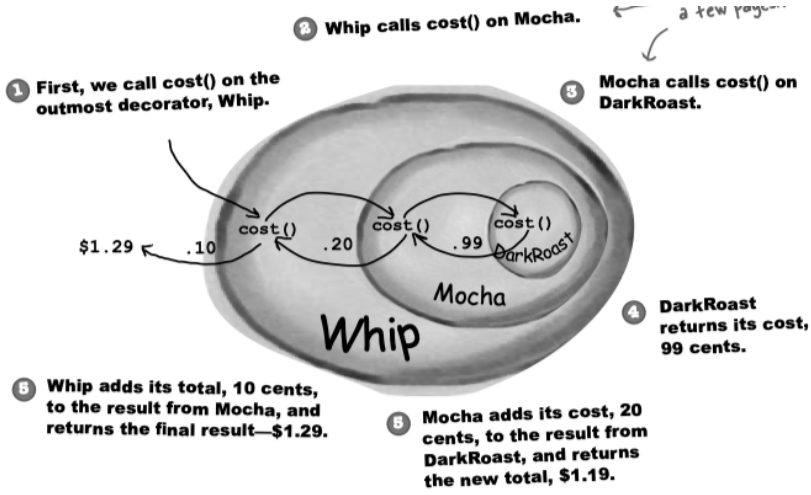
3. Decorator Pattern

Một trong những khía cạnh quan trọng nhất trong quá trình phát triển một ứng dụng mà các lập trình viên phải đối đầu là sự thay đổi. Khi muốn thêm hoặc loại bỏ một tính năng của một đối tượng, điều đầu tiên chúng ta nghĩ đến là thừa kế (extends). Tuy nhiên, thừa kế không khả thi vì nó là static, chúng ta không thể thêm các lớp con mới vào một chương trình khi nó đã được biên dịch và thực thi. Để giải quyết vấn đề này, chúng ta có thể sử dụng **Decorator Pattern** được giới thiệu trong phần tiếp theo của bài viết này.

3.1 Định nghĩa

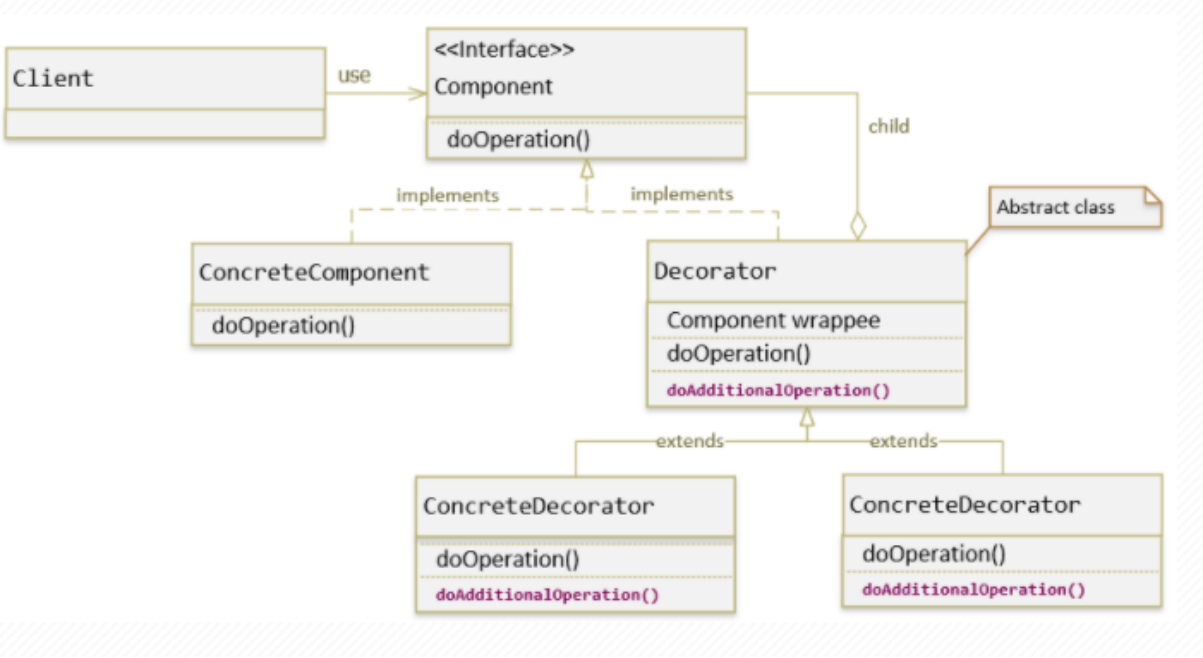
Decorator pattern là một trong những Pattern thuộc nhóm cấu trúc Structure Pattern. Nó cho phép người dùng thêm chức năng mới vào đối tượng hiện tại mà không muốn ảnh hưởng đến các đối tượng khác. Kiểu thiết kế này có cấu trúc hoạt động như một lớp bao bọc (wrap) cho lớp hiện có. Mỗi khi cần thêm tính năng mới, đối tượng hiện có được wrap trong một đối tượng mới (decorator class).

Ví dụ sau đây là DarkRoast được wrap bởi mocha, whip



Decorator pattern sử dụng composition thay vì inheritance (thừa kế) để mở rộng đối tượng. Decorator pattern còn được gọi là **Wrapper** hay **Smart Proxy**.

3.2 Cấu trúc của mẫu thiết kế

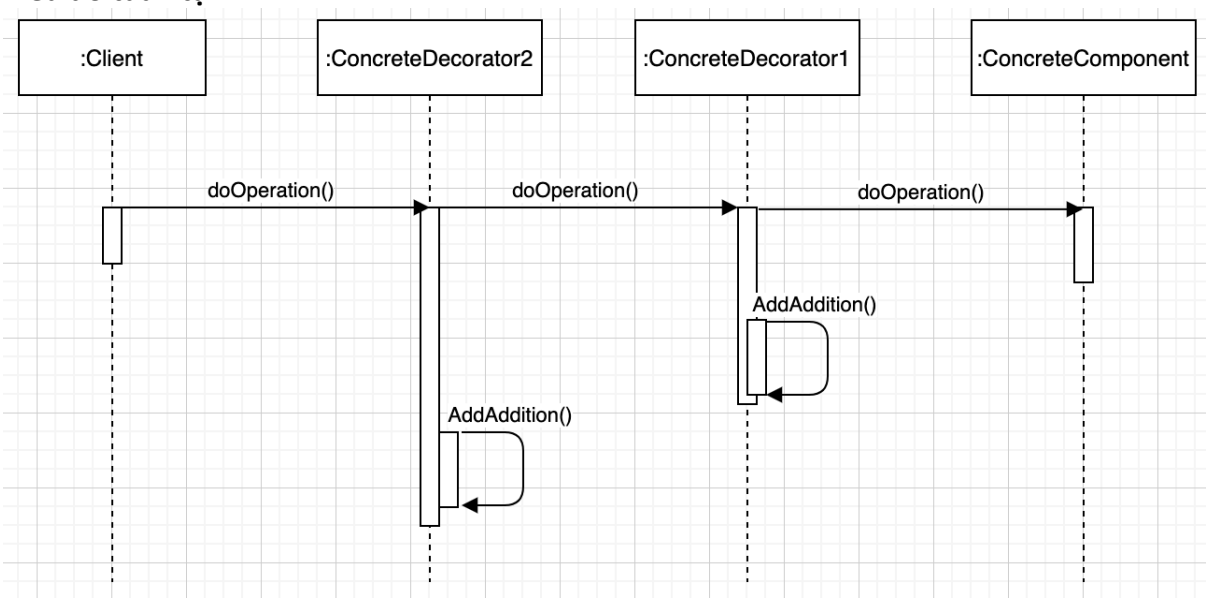


Các thành phần trong decorator pattern:

- **Component**: là một interface quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.
- **ConcreteComponent** : là lớp hiện thực (implements) các phương thức của Component.
- **Decorator** : là một abstract class dùng để Wrap Component và đồng thời cài đặt các phương thức của Component interface.
- **ConcreteDecorator** : là lớp hiện thực (implements) các phương thức của Decorator, nó cài đặt thêm các tính năng mới cho Component.

- **Client** : đối tượng sử dụng Component.

  - **Biểu đồ tuần tự:**



3.3 Sử dụng Decorator Pattern khi nào?

- Khi muốn thêm tính năng mới cho các đối tượng mà không ảnh hưởng đến các đối tượng này.
- Khi không thể mở rộng một đối tượng bằng cách thừa kế (inheritance). Chẳng hạn, một class sử dụng từ khóa final, muốn mở rộng class này chỉ còn cách duy nhất là sử dụng decorator.
- Trong một số nhiều trường hợp mà việc sử dụng kế thừa sẽ mất nhiều công sức trong việc viết code. Ví dụ trên là một trong những trường hợp như vậy.

3.4 Ưu nhược điểm

Ưu điểm:

- Bạn có thể mở rộng hành vi của đối tượng mà không cần tạo lớp con mới.
- Bạn có thể thêm hoặc xóa chức năng khỏi một đối tượng trong thời gian chạy.
- Bạn có thể kết hợp một số hành vi bằng cách gói một đối tượng thành multiple decorator.
- Nguyên tắc Single responsility: Bạn có thể chia một lớp thực hiện nhiều phương án của một hành vi có thể thành một số lớp nhỏ hơn.

Nhược điểm:

- Thật khó để xóa một sự bao bọc cụ thể khỏi ngăn xếp sự bao bọc.
- Thật khó để triển khai deocrator theo cách mà hành vi của nó không phụ thuộc vào thứ tự trong ngăn xếp trình trang trí. (Một thứ tự khác
- Mã cấu hình ban đầu của các lớp có thể trông khá xấu.
- Người trang trí có thể tạo ra nhiều đối tượng nhỏ trong thiết kế và việc sử dụng quá mức có thể phức tạp.

3.5 Mối quan hệ với các Patterns khác.

- Adapter thay đổi giao diện của một đối tượng hiện có, trong khi Decorator thêm chức năng cho một đối tượng mà không thay đổi giao diện của nó. Ngoài ra, Decorator hỗ trợ thành phần đệ quy, điều này không thể thực hiện được khi bạn sử dụng Adapter.
- Adapter cung cấp một giao diện khác cho đối tượng được bao bọc để tương thích, Proxy cung cấp cho nó một giao diện để điều khiển truy cập và Decorator cung cấp cho nó một giao diện thêm chức năng.
- Composite và Decorator có sơ đồ cấu trúc tương tự vì cả hai dựa vào thành phần đệ quy để tổ chức một số đối tượng.
    - Decorator giống như Composite nhưng chỉ có một thành phần con. Có một sự khác biệt đáng kể khác: Decorator thêm các trách nhiệm cho đối tượng được bao bọc, trong khi Composite chỉ "tổng hợp" các kết quả con của nó.
    - Tuy nhiên, các mẫu cũng có thể hợp tác: bạn có thể sử dụng Decorator để mở rộng hành vi của một đối tượng cụ thể trong cây Composite.
- Decorator cho phép bạn thay đổi giao diện của một đối tượng, trong khi Strategy cho phép bạn thay đổi ruột.
- Decorator và Proxy có cấu trúc tương tự, nhưng mục đích rất khác nhau. Cả hai mẫu đều được xây dựng trên nguyên lý kết tập, trong đó một đối tượng được cho là ủy thác một số Task cho đối tượng khác. Sự khác biệt là Proxy thường tự quản lý vòng đời của đối tượng dịch vụ của nó, trong khi Decorator luôn được kiểm soát bởi client.

4. Factory Pattern

Factory Pattern đúng nghĩa là một **nhà máy**, và nhà máy này sẽ "**sản xuất**" các đối tượng theo yêu cầu của chúng ta. Nói một cách khác, nhiệm vụ của Factory Pattern là quản lý và trả về các đối tượng theo yêu cầu, giúp cho việc khởi tạo đối tượng một cách linh hoạt hơn. Tất cả các Factory đều đóng gói sự khởi tạo này

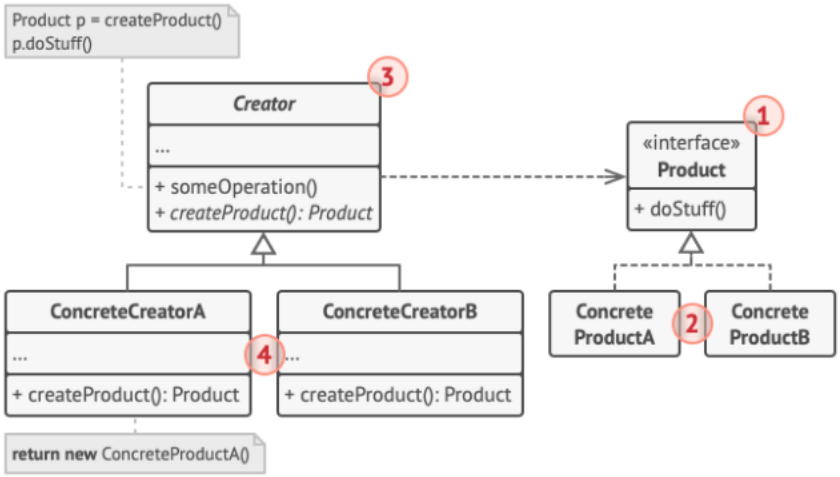
4.1 Factory method Pattern

4.1.1 Định nghĩa

Factory Method Pattern là một giao diện để tạo một đối tượng, nhưng cho phép các lớp con quyết định lớp nào để khởi tạo. Factory method cho phép một lớp đưa việc khởi tạo cho các lớp con.

Các Factory đóng gói sự khởi tạo của các lớp. Sự khởi tạo này được uỷ nhiệm xuống các lớp con mà triển khai factory method để tạo ra đối tượng. Do dó Factory method dựa trên nguyên lý kế thừa.

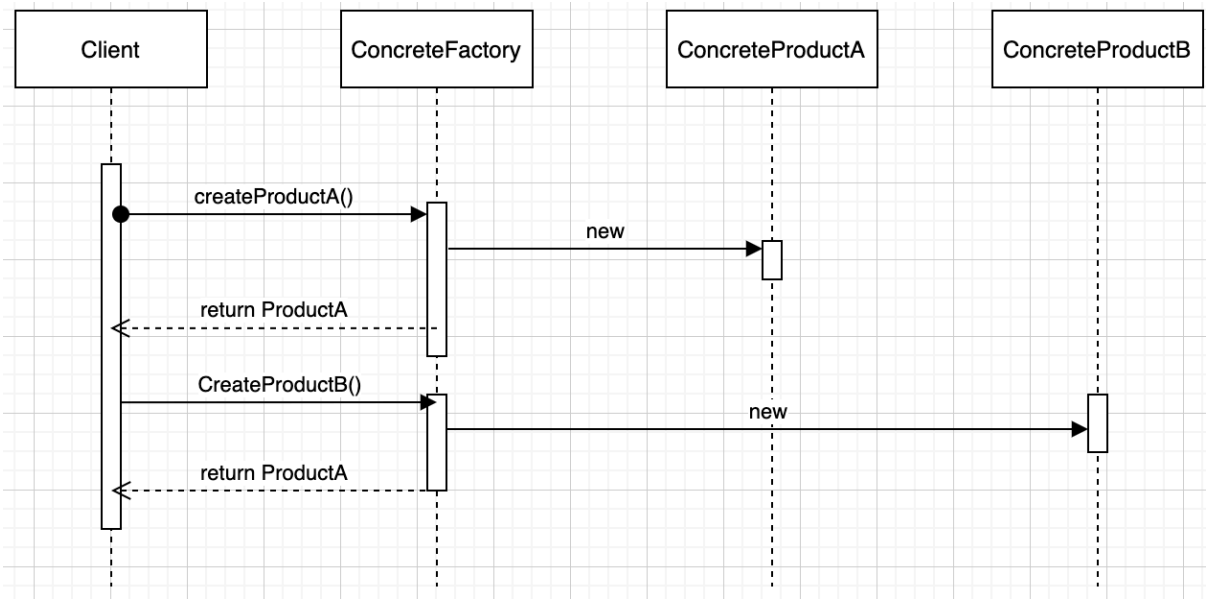
4.1.2 Cấu trúc



Các thành phần cấu trúc của mẫu thiết kế:

- **Product** khai báo giao diện, giao diện này chung cho tất cả các đối tượng có thể được tạo ra bởi người tạo và các lớp con của nó.
- **Concrete Products** là các cài đặt khác nhau của giao diện sản phẩm.
- **Lớp Creator** khai báo Factory method để trả về các đối tượng sản phẩm mới. Điều quan trọng là loại trả về của phương thức này phải phù hợp với giao diện sản phẩm.

    - Bạn có thể khai báo phương thức factory là trừu tượng để buộc tất cả các lớp con cài đặt các phiên bản phương thức riêng của chúng. Thay vào đó, phương thức nhà máy cơ sở có thể trả về một số loại sản phẩm mặc định.
    - Lưu ý, mặc dù tên của nó, việc tạo ra sản phẩm không phải là trách nhiệm chính của lớp creator. Đây là một ví dụ: một công ty phát triển phần mềm lớn có thể có một bộ phận đào tạo cho các lập trình viên. Tuy nhiên, chức năng chính của toàn bộ công ty vẫn là viết mã, không sản xuất lập trình viên.
- **Concrete Creators** ghi đè phương thức gốc của nhà máy để nó trả về một loại sản phẩm khác. Lưu ý rằng phương pháp gốc không phải lúc nào cũng phải tạo các phiên bản mới. Nó cũng có thể trả về các đối tượng hiện có từ một bộ nhớ cache, một nhóm đối tượng hoặc một nguồn khác.
- **Biểu đồ tuần tự:**



4.1.3 Sử dụng Factory method Pattern khi nào?.

- Sử dụng Factory method pattern khi bạn không biết trước các loại và sự phụ thuộc chính xác của các đối tượng mà mã nguồn của bạn sẽ hoạt động.
    - Factory method phân tách mã xây dựng sản phẩm với mã thực sự sử dụng sản phẩm(Principle 1). Do đó, mã này độc lập với phần còn lại nên việc mở rộng và thay đổi sẽ dễ giàng hơn
    - **Ví dụ**: để thêm một loại sản phẩm mới vào ứng dụng, bạn chỉ cần tạo ra một lớp creator mới và ghi đè phương thức gốc trong đó. (Rất tiện để mở rộng).

- Sử dụng factory method khi bạn muốn cung cấp cho người dùng thư viện hoặc Framwork một cách để có thể mở rộng các thành phần bên trong của nó.
    - Kế thừa có lẽ là cách dễ nhất để mở rộng hành vi mặc định của một thư viện hoặc một framework. Nhưng làm thế nào để framwork nhận ra rằng lớp con của bạn nên được sử dụng thay cho một standard component
    - Giải pháp là giảm mã xây dựng các thành phần trên framwork thành một phương thức gốc và cho phép bất kỳ ai ghi đè phương thức này ngoài việc mở rộng thành phần chính..

- Sử dụng Factory method khi bạn muốn tiết kiệm tài nguyên hệ thống bằng cách sử dụng lại các đối tượng hiện có thay vì xây dựng lại chúng mỗi lần.
    - Bạn thường gặp phải nhu cầu này khi xử lý các đối tượng lớn, sử dụng nhiều tài nguyên như kết nối cơ sở dữ liệu, hệ thống tệp và tài nguyên mạng.
    - Do đó, bạn cần phải có một phương pháp thường xuyên có khả năng tạo các đối tượng mới cũng như sử dụng lại các đối tượng hiện có. Điều đó nghe rất giống một factory method pattern.

4.1.4 Ưu nhược điểm

Ưu điểm:

- Bạn tránh được tight coupling giữa creator và conrete product(Loose coupled principle)
- Đản bảo nguyên tắc Single responsibility: Bạn có thể di chuyển **mã tạo sản phẩm** vào một nơi trong chương trình, giúp mã hỗ trợ dễ dàng hơn.
- Đản bảo open / close principle: Bạn có thể thêm các loại product mới vào chương trình mà không cần phá vỡ mã hiện có.

Nhược điểm:

- Mã có thể trở nên phức tạp hơn vì bạn cần giới thiệu nhiều lớp con mới để triển khai mẫu thiết kế.

4.1.5 Mối quan hệ với các Patterns khác.

- Nhiều thiết kế bắt đầu bằng cách sử dụng Factory Method (ít phức tạp hơn và có thể tùy chỉnh nhiều hơn thông qua các lớp con) và phát triển theo hướng Abstract Factory, Prototype hoặc Builder (linh hoạt hơn nhưng phức tạp hơn).

- Các lớp Abstract Factory thường dựa trên một tập hợp các factory methods.

- Bạn có thể sử dụng Factory Method cùng với Iterator để cho phép các lớp con của collectiones trả về các loại iterators khác nhau tương thích với các Collections.

- Factory Method dựa trên kế thừa nhưng không yêu cầu bước khởi tạo.

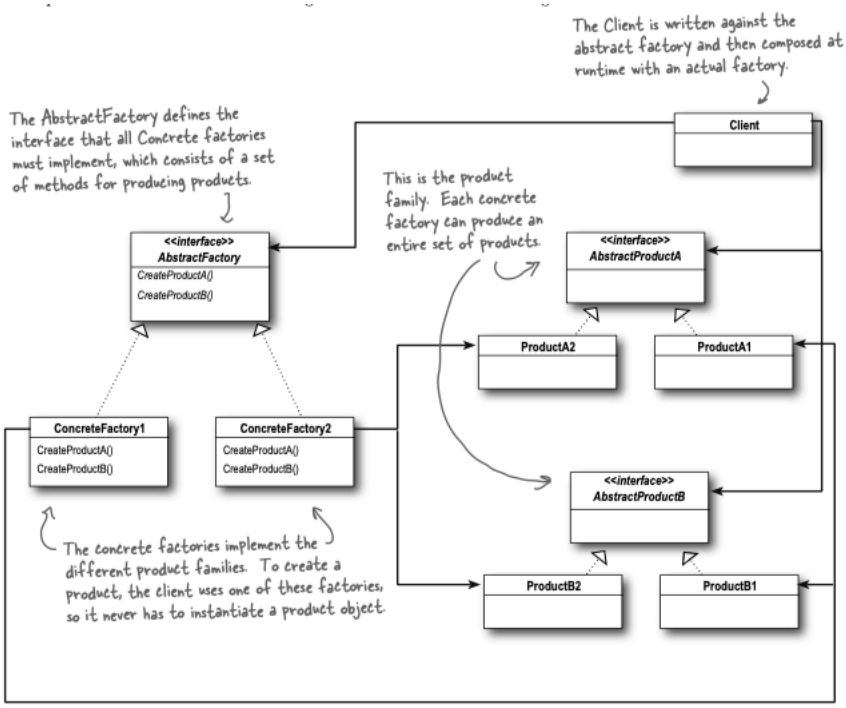
- Factory Method là một sự đặc biệt hoá của template method. Đồng thời, a Factory Method có thể đóng vai trò là một bước trong a big Template Method.

4.2 Abstract Factory

4.2.1. Định nghĩa

Abstract Factory cung cấp một giao diện để tạo họ các đối tượng liên quan hoặc phụ thuộc mà không chỉ định các lớp cụ thể của chúng(Principle 3: Program to interface, not to implementation). Abstract Factory dưa trên nguyên lý kết tập

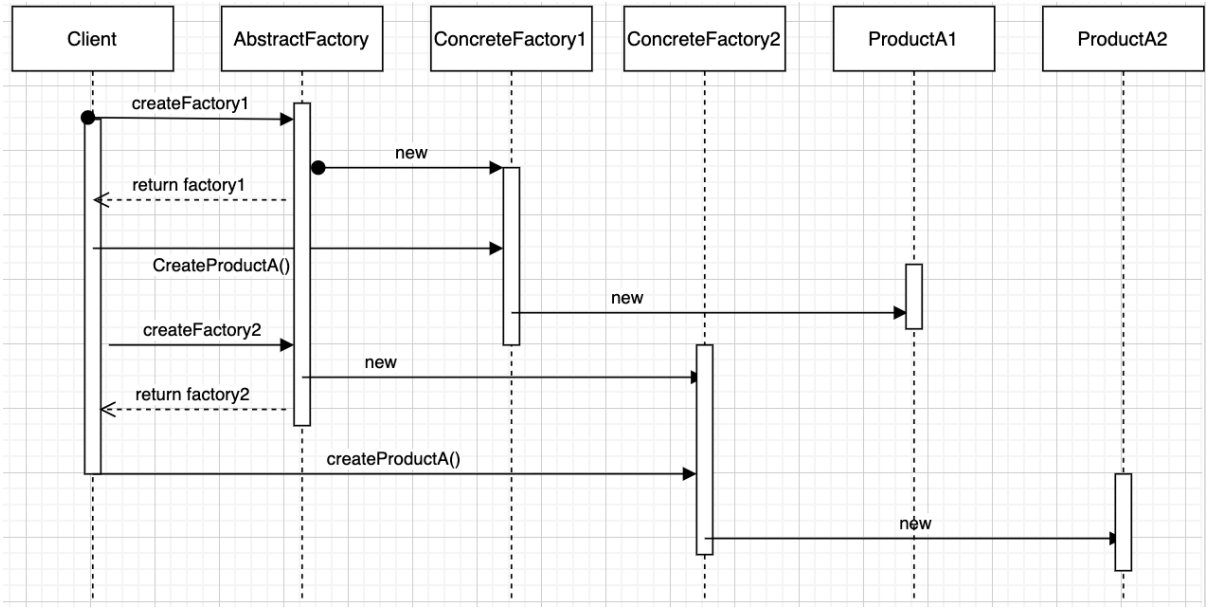
4.2.2 Cấu trúc



Các thành phần cụ thể của mẫu thiết kế:

- **The client** tương tác với concrete factory trong thời gian chạy qua abstract factory.
- **AbstratFactory**: định nghĩa giao diện mà tất cả các Conrete Factories phải cài đặt, giao diện này bao gồm tập các method để tạo ra product.
- Các **Conrete Factories** cài đặt họ các product khác nhau. Để tạo ra Product, Khách hàng sự dung một trong các factories, do đó không bao giờ cần phải khởi tạo a product object.
- **AbstractProduct** là giao diện của một họ các sản phẩm. Mỗi lớp product triển khai giao diện này. Và mỗi mỗi Concrete Factories tạo ra một product đó.

- **Biểu đồ tuần tự:**



4.2.3 Sử dụng abstract Factory khi nào?.

- Sử dụng Abstract Factory khi code của bạn cần hoạt động với nhiều nhóm sản phẩm liên quan khác nhau, nhưng bạn không muốn nó phụ thuộc vào các lớp cụ thể của các sản phẩm đó — chúng có thể chưa được biết trước hoặc bạn sẽ thêm dần dần vào trong tương lai.
- Abstract Factory cung cấp cho bạn một giao diện để tạo các đối tượng từ mỗi lớp của họ sản phẩm.
- Trong một chương trình được thiết kế tốt*, phải đản bảo nguyên tắc single responsibility.* Khi một lớp xử lý nhiều loại sản phẩm, nó có thể đáng giá khi tác ra các Factory methods của nó thành một lớp nhà máy độc lập.

4.2.4 Ưu nhược điểm

Ưu điểm:

- Bạn có thể chắc chắn rằng các sản phẩm bạn nhận được từ nhà máy tương thích với nhau.
- Avoid tight coupling giữa Khách hàng và concrete products.
- Đản bảo Nguyên tắc Singple responsilibity: Bạn có thể tách mã tạo sản phẩm vào một nơi. Và từ dễ quản lý cũng như bảo trì hay mở rộng
- Đản bảo open closed principle: Bạn có thể giới thiệu các phương án mới của sản phẩm mà không cần phá vỡ mã khách hàng hiện có.

Nhược điểm:

- Mã có thể trở nên phức tạp hơn mức bình thường, vì rất nhiều giao diện và lớp mới được giới thiệu cùng với mẫu.

4.2.5 Mối quan hệ của Abstract Factory với các pattern khác.

- Builder tập trung vào việc xây dựng các đối tượng phức tạp theo từng bước. Abstract Factory chuyên tạo một họ các đối tượng liên quan với nhau. Abstract Factory trả lại sản phẩm ngay lập tức, trong khi Builder cho phép bạn chạy một số bước xây dựng bổ sung trước khi tìm nạp sản phẩm.

- Abstract Factory có thể phục vụ như một giải pháp thay thế cho Facade khi bạn chỉ muốn ẩn cách các đối tượng trong hệ thống con được tạo ra khỏi client code.

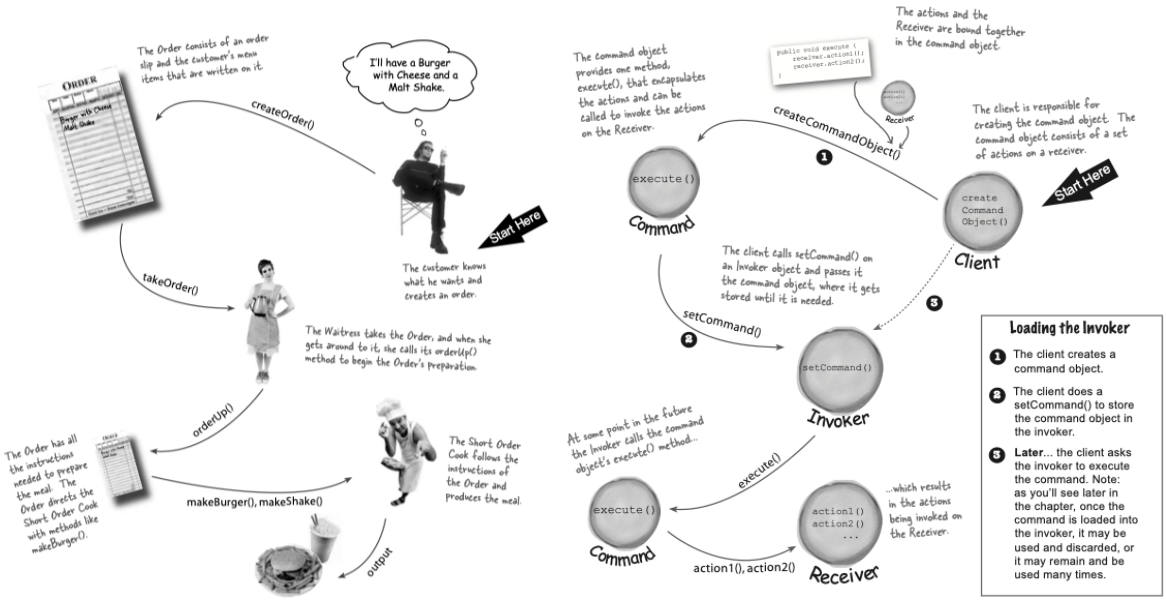
- Abstract factories đều có thể được triển khai dưới dạng Singletons.

- Factory method sử dụng nguyên lý kế thừa. Còn Abstract Factory sử dụng nguyên lý kết tập. Nhưng cả 2 đều giảm bới sự phụ thuộc của ứng dụng vào các concrete class.(tức là tuân thủ nguyên tắc Loose Coupling).

5. Command Pattern

Đôi khi chúng ta cần gửi các yêu cầu cho các đối tượng mà không biết bất cứ điều gì về hoạt động được yêu cầu hoặc người nhận yêu cầu. Ví dụ khi chúng có một ứng dụng văn bản, khi click lên button undo/ redo, save, … yêu cầu sẽ được chuyển đến hệ thống xử lý, chúng ta sẽ không thể biết được đối tượng nào sẽ nhận xử lý, cách nó thực hiện như thế nào.(Principle 3). **Command Pattern** là một Pattern được thiết kế cho những ứng dụng như vậy.

Ngoài ra chúng ta còn có ví dụ ngoài đời thực như khi order đồ ăn như hình dưới



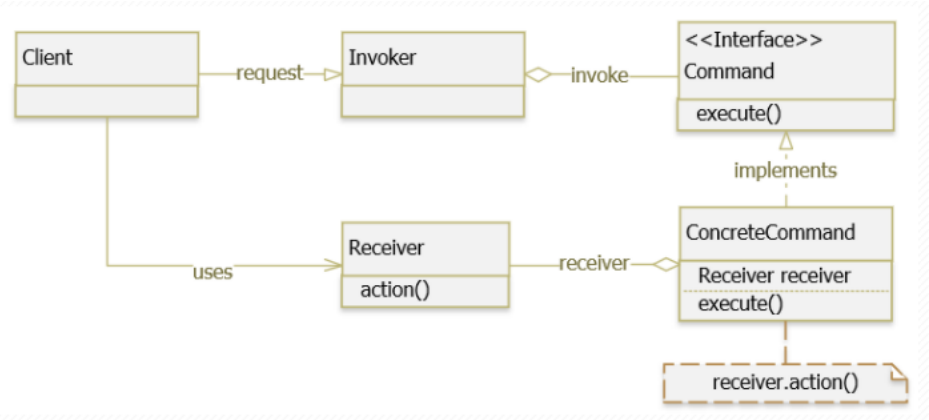
5.1 Định nghĩa

Command Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó cho phép chuyển yêu cầu thành đối tượng độc lập, có thể được sử dụng để tham số hóa các đối tượng với các yêu cầu khác nhau như log, queue (undo/redo), transtraction.

Command Pattern cho phép tất cả những Request gửi đến object được lưu trữ trong chính object đó dưới dạng một object Command. Khái niệm Command Object giống như một class trung gian được tạo ra để lưu trữ các câu lệnh và trạng thái của object tại một thời điểm nào đó.

Có thể hiểu như sau: Command dịch ra nghĩa là ra lệnh. Commander nghĩa là chỉ huy, người này không làm mà chỉ ra lệnh cho người khác làm. Như vậy, phải có người nhận lệnh và thi hành lệnh. Người ra lệnh cần cung cấp một class đóng gói những mệnh lệnh. Người nhận mệnh lệnh cần phân biệt những interface nào để thực hiện đúng mệnh lệnh

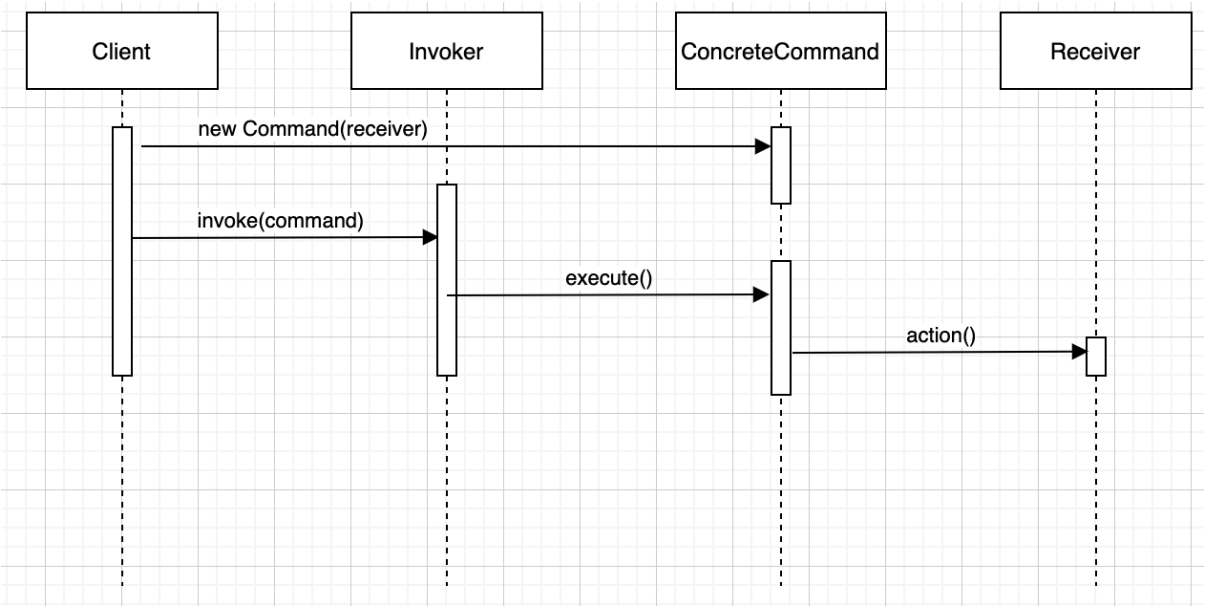
5.2 Cấu trúc



Các thành phần tham gia trong command Pattern:

- **Command** : là một interface hoặc abstract class, chứa một phương thức trừu tượng thực thi (execute) một hành động (operation). Request sẽ được đóng gói dưới dạng Command.
- **ConcreteCommand** : là các implementation của Command. Định nghĩa một sự gắn kết giữa một đối tượng Receiver và một hành động. Thực thi execute() bằng việc gọi operation đang hoãn trên Receiver. Mỗi một ConcreteCommand sẽ phục vụ cho một case request riêng.
- **Client** : tiếp nhận request từ phía người dùng, đóng gói request thành ConcreteCommand thích hợp và thiết lập receiver của nó.
- **Invoker** : tiếp nhận ConcreteCommand từ Client và gọi execute() của ConcreteCommand để thực thi request.
- **Receiver** : đây là thành phần thực sự xử lý nghiệp vụ cho các trường hợp request. Trong phương execute() của ConcreteCommand chúng ta sẽ gọi method thích hợp trong Receiver.

  - **Biểu đồ tuần tự:**



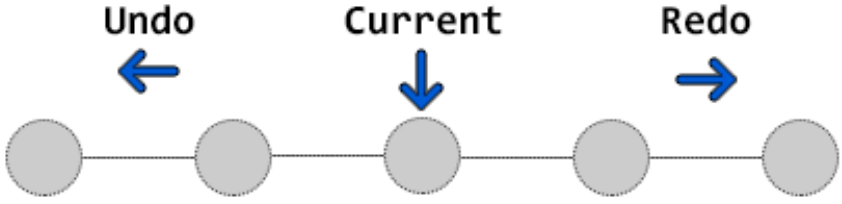
5.3 Khi nào dùng Command Pattern.

- Sử dụng mẫu lệnh khi bạn muốn tham số hóa các đối tượng bằng các phép toán.
    - Mẫu lệnh có thể biến một lời gọi phương thức cụ thể thành một đối tượng độc lập. Thay đổi này mở ra rất nhiều cách sử dụng thú vị: bạn có thể chuyển các lệnh làm argument của một phương thức, lưu trữ chúng bên trong các đối tượng khác, chuyển đổi các lệnh được liên kết trong thời gian chạy, v.v.

- Sử dụng mẫu Lệnh khi bạn muốn xếp hàng đợi các thao tác, lập lịch thực thi hoặc thực thi chúng từ xa.(Vi dụ khi bạn muốn dùng remote control để ra điều khiển mọi)

  Như với bất kỳ đối tượng nào khác, một lệnh có thể được tuần tự hóa, ghi vào tệp hoặc có nghĩa là chuyển đổi nó thành một chuỗi để có thể dễ dàng cơ sở dữ liệu. Sau đó, chuỗi có thể được khôi phục như đối tượng lệnh ban đầu. Do đó, bạn có thể trì hoãn và lên lịch thực hiện lệnh. Nhưng còn nhiều hơn thế nữa! Theo cách tương tự, bạn có thể xếp hàng, ghi nhật ký hoặc gửi lệnh qua mạng.

- Sử dụng Lệnh khi bạn muốn triển khai các hoạt động có thể đảo ngược.



Mặc dù có nhiều cách để thực hiện hoàn tác / làm lại, nhưng Command Pattern có lẽ là phổ biến nhất. Để có thể undo các hoạt động, bạn cần lưu lại lịch sử các hoạt động đã thực hiển. Lịch sử lệnh là một ngăn xếp chứa tất cả các đối tượng lệnh đã thực thi cùng với các bản sao lưu có liên quan về trạng thái của ứng dụng. Phương pháp này có **hai nhược điểm**.

- Đầu tiên, không dễ dàng để lưu trạng thái của ứng dụng vì một số trong số đó có thể là riêng tư..
- Thứ hai, các bản sao lưu trạng thái có thể tiêu tốn khá nhiều RAM. Do đó, đôi khi bạn có thể sử dụng một cách triển khai thay thế: thay vì khôi phục trạng thái trong quá khứ **thực hiện thao tác nghịch đảo.** Hoạt động ngược lại cũng có một cái giá: nó có thể trở nên khó khăn hoặc thậm chí không thể thực hiện được.

5.4 Ưu nhược điểm

Ưu điểm:

- Đảm bảo **nguyên tắc Single Responsibility**: Bạn có thể tách các lớp invoke operation từ các lớp thực hiện operation(Người nhận lệnh và người thực hiện lệnh).
- Đảm bảo **open/ closed principle**: Bạn có thể đưa mở rộng ứng dụng bằng cách đưa lệnh mới vào mà không cần thay đổi lệnh đã hiện có.
- Đóng gói một yêu cầu trong một đối tượng. Dễ dàng chuyển dữ liệu dưới dạng đối tượng giữa các thành phần hệ thống.
- Bạn có thể thực hiện undo/redo
- Bạn có thể triển khai thực hiện hoãn lại các hoạt động.
- Có thể tạo tạo ra macro commnd từ các lệnh đơn giản.
- Cho phép tham số hóa các yêu cầu khác nhau bằng một hành động để thực hiện.

Nhược điểm:

- Với mỗi command tương ứng với một lớp nên nếu có nhiều lệnh thì sẽ có nhiều lớp.

5.5 Mối quan hệ của command Pattern với các Patterns khác.

- **Chain of Responsibility, Command, Mediator và Observer** giải quyết các cách khác nhau để kết nối giữa senders và receivers.
  - **Chain of Responsibility** chuyển một yêu cầu tuần tự dọc theo một chuỗi động gồm những người nhận tiềm năng cho đến khi một trong số họ xử lý nó.
  - **Command** thiết lập các kết nối một chiều giữa người gửi và người nhận.
  - **Mediator** loại bỏ các kết nối trực tiếp giữa người gửi và người nhận, buộc họ phải giao tiếp gián tiếp thông qua một **đối tượng trung gian.**
  - **Observer** cho phép người nhận đăng ký động và hủy đăng ký nhận yêu cầu.

- Command và Stradegy có thể trông giống nhau vì bạn có thể sử dụng cả hai để tham số hóa một đối tượng bằng một số hành động. Tuy nhiên, họ có mục đích khác nhau:

  - Bạn có thể sử dụng Command để chuyển đổi bất kỳ thao tác nào thành một đối tượng. Các tham số của hoạt động trở thành các trường của đối tượng đó. Việc chuyển đổi cho phép bạn trì hoãn việc thực hiện thao tác, xếp hàng đợi, lưu trữ lịch sử các lệnh, gửi lệnh đến các dịch vụ từ xa, v.v.

- Mặt khác, Strategy thường mô tả các cách khác nhau để thực hiện cùng một hành động, cho phép bạn hoán đổi các thuật toán này trong một context.

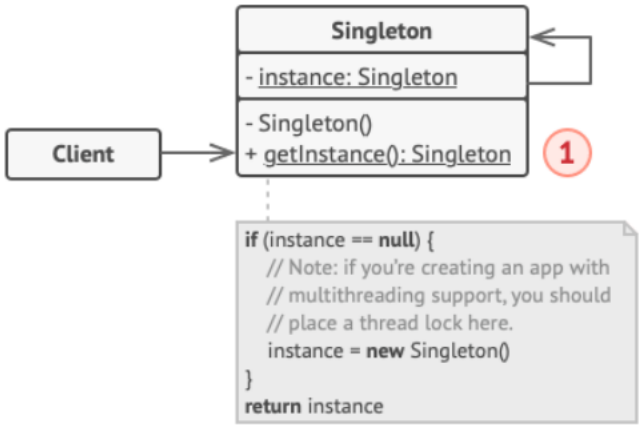
6. Singleton Pattern

Đôi khi, trong quá trình phân tích thiết kế một hệ thống, chúng ta mong muốn có những đối tượng cần tồn tại duy nhất và có thể truy xuất mọi lúc mọi nơi. Ví dụ như thread pools, caches, dialog boxes. Làm thế nào để hiện thực được một đối tượng như thế khi xây dựng mã nguồn? Chúng ta có thể nghĩ tới việc sử dụng một biến toàn cục (global variable : public static final). Tuy nhiên, việc sử dụng biến toàn cục nó phá vỡ quy tắc của **OOP (encapsulation)**. Để giải bài toán trên, người ta hướng đến một giải pháp là sử dụng **Singleton pattern**.

6.1 Định nghĩa

Singleton đảm bảo chỉ duy nhất một thể hiện (instance) được tạo ra và nó sẽ cung cấp cho bạn một method để có thể truy xuất được thể hiện duy nhất đó mọi lúc mọi nơi trong chương trình.

6.2 Cấu trúc



Lớp Singleton khai báo phương thức tĩnh getInstance trả về cùng một thể hiện của lớp riêng của nó
Constructor được che dấu(Khác với global variable là đảm bảo tính encapsulation). Do đó calling getInstance() là cách duy nhất để có được lớp Singleton.

6.3 Khi nào sử dụng singleton Pattern?.

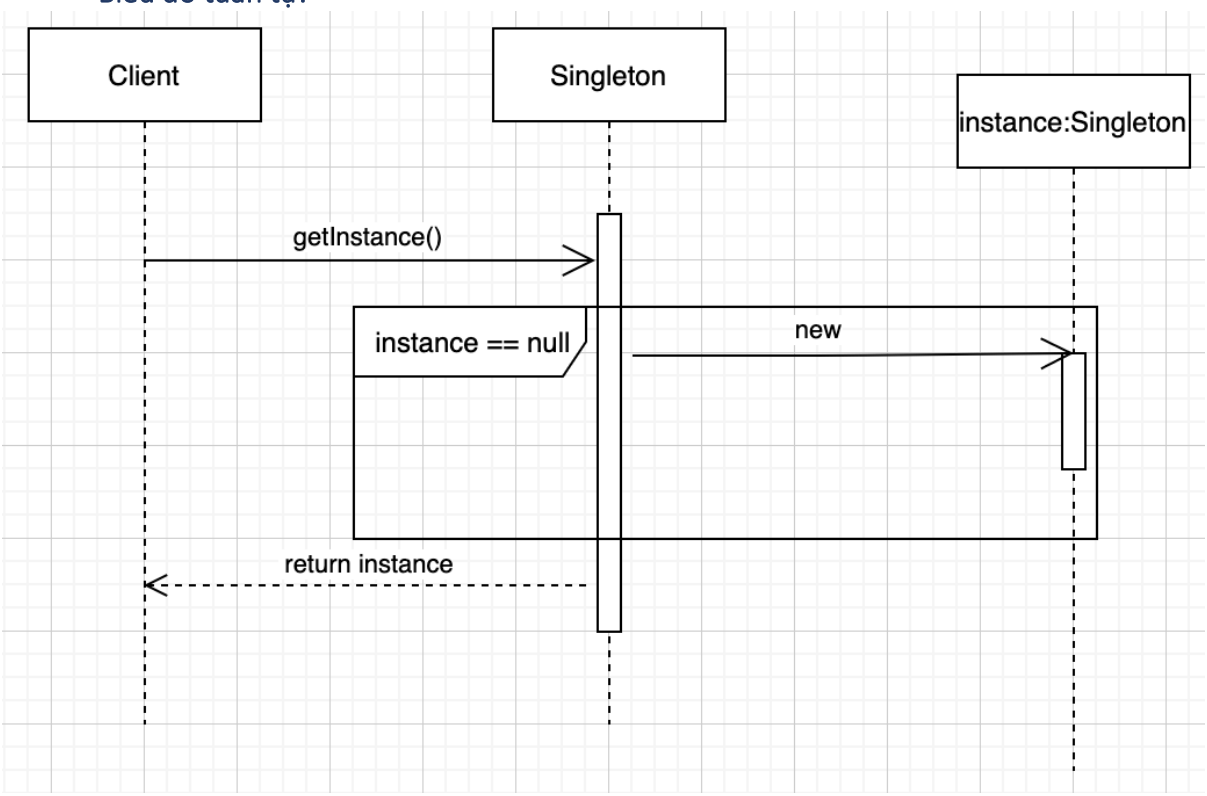
- Sử dụng mẫu Singleton khi một lớp trong chương trình của bạn chỉ nên có thực thể (Tài nguyên) duy nhất có sẵn cho tất cả các máy khách; ví dụ, một đối tượng cơ sở dữ liệu duy nhất được chia sẻ bởi các phần khác nhau của chương trình.

    - Mẫu Singleton vô hiệu hóa tất cả các phương thức tạo đối tượng khác của một lớp ngoại trừ phương thức tạo đặc biệt. Phương thức này tạo một đối tượng mới **hoặc** trả về một đối tượng nếu nó đã được tạo trước đó.

- Sử dụng mẫu Singleton khi bạn cần kiểm soát chặt chẽ hơn đối với các biến toàn cục.

    - Không giống như các biến toàn cục, mẫu Singleton đảm bảo rằng chỉ có một thực thể của một lớp. Không có gì, ngoại trừ chính lớp Singleton, có thể thay thế phiên bản được lưu trong bộ nhớ cache.
    - Lưu ý rằng bạn luôn có thể điều chỉnh giới hạn này và cho phép tạo bất kỳ số lượng cá thể Singleton nào. Đoạn mã duy nhất cần thay đổi là phần thân của phương thức getInstance. (Nhưng như thế sẽ mất đi trách nhiệm thứ nhất).

- Biểu đồ tuần tự:



6.4 Ưu nhược điểm của Singleton Pattern.

Ưu điểm:

- Bạn có thể chắc chắn rằng một lớp chỉ có một cá thể duy nhất.
- Bạn có được một điểm truy cập toàn cầu vào đối tượng đó.
- Đối tượng singleton chỉ được khởi tạo khi nó được yêu cầu lần đầu tiên.
- Xử lý trong môi trường có nhiều luồng

Nhược điểm:

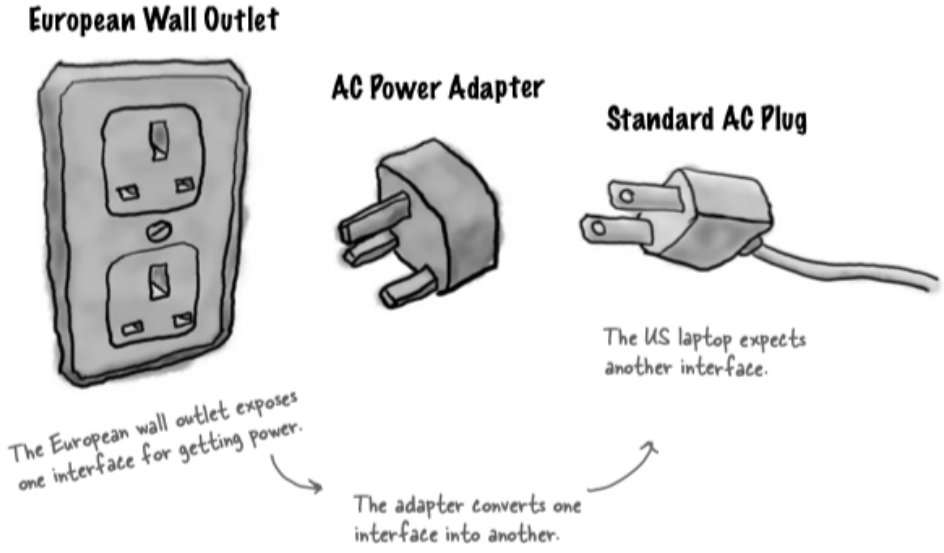
- Không tuân theo nguyên tắc Single Responsibility khi có đến 2 trách nhiệm.
- Ví dụ, mẫu Singleton có thể che giấu thiết kế xấu khi các thành phần của chương trình biết quá nhiều về nhau.
- Việc cài đặt sẽ khó khăn và có nhiều cách cài đặt khác nhau.

6.5 Mối quan hệ với các Pattern khác

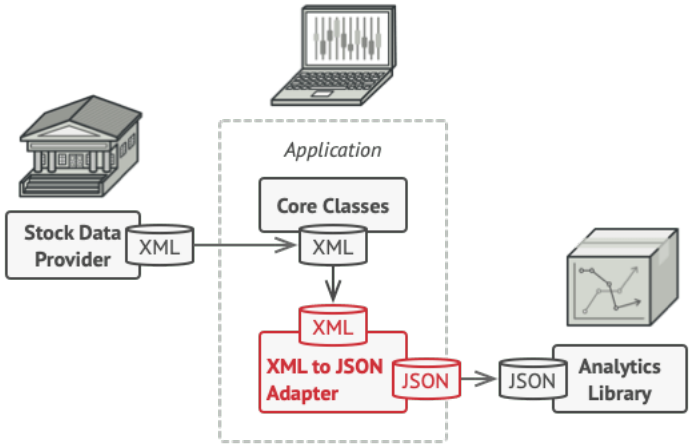
- Một Facade class có thể được chuyển đổi thành một Singleton vì một đối tượng Facade duy nhất là đủ trong hầu hết các trường hợp.

7. Adapter Pattern

Adapter có rất nhiều trong đời sống xung quanh chúng ta. Hình ảnh dưới đây là một ví dụ:



Một ví dụ khác khi chúng ta cần lấy dữ liệu để phân tích:



Ở trên chúng ta dùng Adapter để chuyển đổi XML sang JSON.

7.1 Định nghĩa

Adapter Pattern (Người chuyển đổi) là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Adapter Pattern cho phép các inteface (giao diện) không liên quan tới nhau có thể làm việc cùng nhau. Đối tượng giúp kết nối các interface gọi là Adapter.

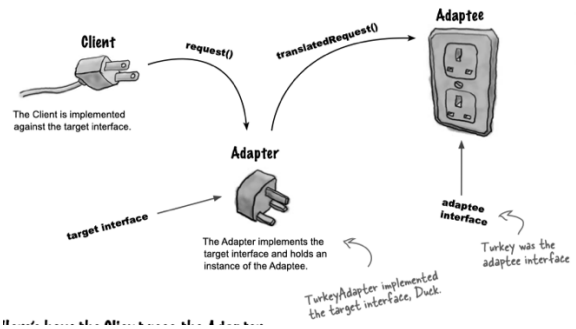
Adapter Pattern giữ vai trò trung gian giữa hai lớp, chuyển đổi interface của một hay nhiều lớp có sẵn thành một interface khác, thích hợp cho lớp đang viết. Điều này cho phép

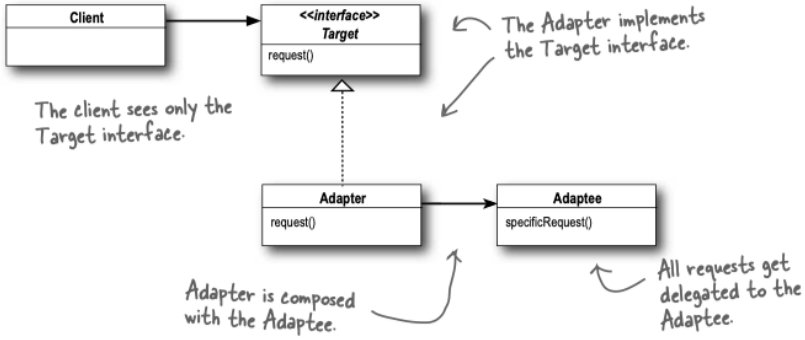
các lớp có các interface khác nhau có thể dễ dàng giao tiếp tốt với nhau thông qua interface trung gian, không cần thay đổi code của lớp có sẵn cũng như lớp đang viết(tuân thủ nguyên tắc open/closed principle).

7.2 Cấu trúc

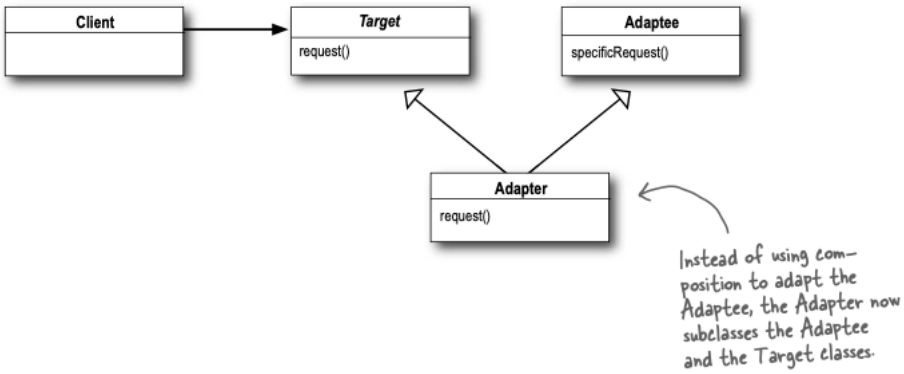
Có 2 cách để thực hiện Adapter Pattern dựa theo cách cài đặt của chúng : Object Adapter và Class Adapter.

- Object Adapter: trong mô hình này, một lớp mới (Adapter) sẽ tham chiếu đến một (hoặc nhiều) đối tượng của lớp có sẵn với interface không tương thích (Adaptee), đồng thời cài đặt interface mà người dùng mong muốn (Target). Trong lớp mới này, khi cài đặt các phương thức của interface người dùng mong muốn, sẽ gọi phương thức cần thiết thông qua đối tượng thuộc lớp có interface không tương thích.



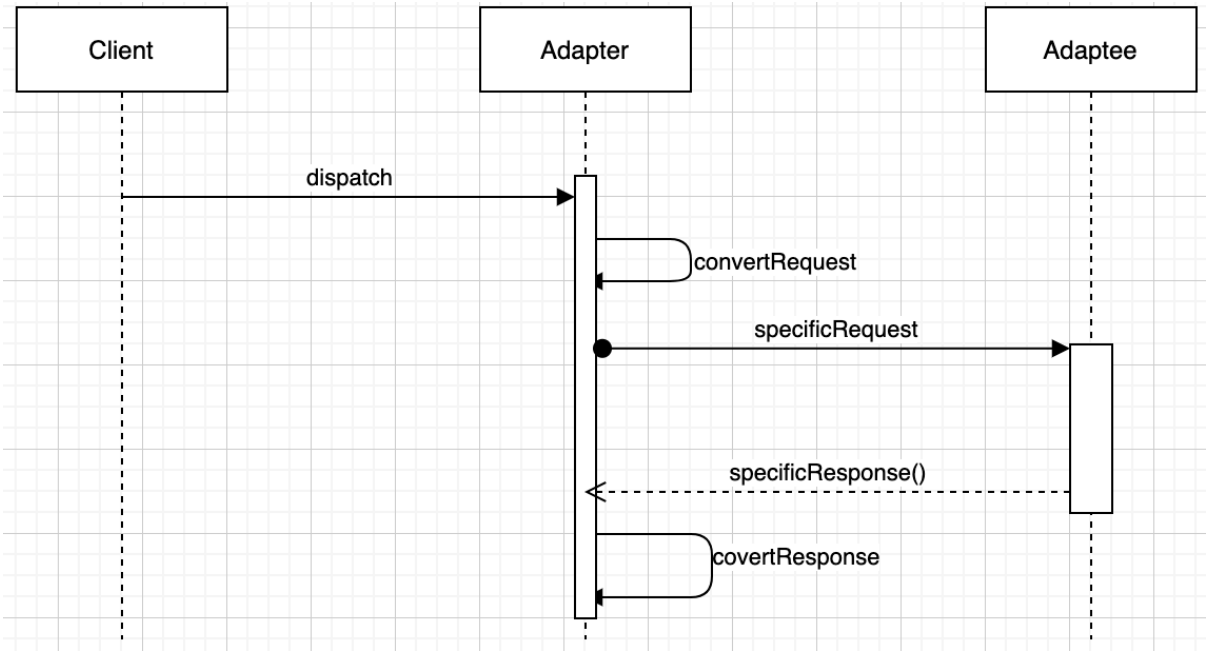


- **Class Adapter – Inheritance** (Kế thừa) : trong mô hình này, một lớp mới (Adapter) sẽ kế thừa lớp có sẵn với interface không tương thích (Adaptee), đồng thời cài đặt interface mà người dùng mong muốn (Target). Trong lớp mới, khi cài đặt các phương thức của interface người dùng mong muốn, phương thức này sẽ gọi các phương thức cần thiết mà nó thừa kế được từ lớp có interface không tương thích.



- Điểm yếu của class Adapter là không thể kế thừa được nhiều lớp trong java. Còn Object adapter sử dụng nguyên lý kết tập nên cho phép Adapter có thể giữ nhiều thể hiện của Adaptee nếu cần thiết.

- **Biểu đồ tuần tự:**



7.3 Khi nào sử dụng Adapter Pattern?.

- Sử dụng Adatper Pattern khi bạn muốn sử dụng một số lớp hiện có, nhưng giao diện của nó không tương thích với phần còn lại.
    - Adatper Pattern cho phép bạn tạo một lớp trung gian đóng vai trò là người dịch giữa mã của bạn với lớp kế thừa, lớp của bên thứ 3 hoặc bất kỳ lớp nào khác có giao diện kỳ lạ.
- Sử dụng Adatper Pattern khi bạn muốn sử dụng lại một số lớp con hiện có thiếu một số chức năng phổ biến không thể thêm vào lớp cha. (ví dụ Enumerate với Iterator class)
- Khi muốn sử dụng một lớp đã tồn tại trước đó nhưng interface sử dụng không phù hợp như mong muốn.
- Khi muốn tạo ra những lớp có khả năng sử dụng lại, chúng phối hợp với các lớp không liên quan hay những lớp không thể đoán trước được và những lớp này không có những interface tương thích.
- Cần phải có sự chuyển đổi interface từ nhiều nguồn khác nhau.
- Khi cần đảm bảo nguyên tắc **Open/ Close** trong một ứng dụng.

7.4 Ưu nhược điểm

Ưu điểm:

- Cho phép nhiều đối tượng có interface giao tiếp khác nhau có thể tương tác và giao tiếp với nhau.
- Tuân thủ 2 nguyên tắc Opern/Close và nguyên tắc Single Responsibility.

Nhược điểm:

- Tất cả các yêu cầu được chuyển tiếp(qua trung gian), do đó làm tăng thêm một ít chi phí.

- Đôi khi có quá nhiều Adapter được thiết kế trong một chuỗi Adapter (chain) trước khi đến được yêu cầu thực sự.

7.5 Mối quan hệ với những Patterns khác.

- Adapter Pattern thay đổi giao diện của một đối tượng hiện có, trong khi Decorator thêm trách nhiệm mà không thay đổi giao diện của nó. Ngoài ra, Decorator hỗ trợ thành phần đệ quy, điều này không thể thực hiện được khi bạn sử dụng Adapter.

- Facade định nghĩa một giao diện mới cho các đối tượng hiện có để tách sự phức tạp của hệ thống con ra khỏi khách hàng, trong khi Adapter cố gắng làm cho giao diện hiện có có thể sử dụng được. Adatper thường chỉ bao bọc một đối tượng, trong khi Facade hoạt động với toàn bộ hệ thống con của các đối tượng.

- Bridge, State, Strategy (và ở một mức độ nào đó là Adatper) có cấu trúc rất giống nhau. Thật vậy, tất cả các mẫu này đều dựa trên Nguyên lý kết tập, tức là ủy thác công việc cho các đối tượng khác. Tuy nhiên, tất cả chúng đều giải quyết các vấn đề khác nhau.

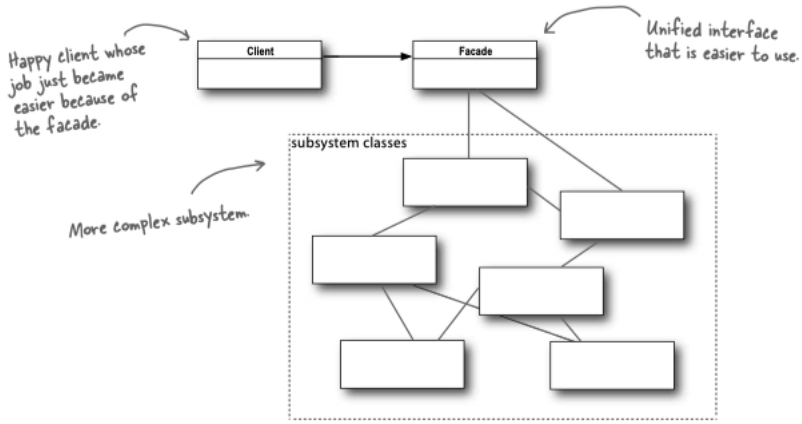
8. Facade Pattern

Để sử dụng Facade Pattern, chúng ta tạo một lớp đơn giản hóa và hợp nhất một tập hợp các lớp phức tạp hơn thuộc về một số hệ thống con. Không giống như rất nhiều mẫu, Facade khá đơn giản; không phức tạp . Nhưng điều đó không làm cho nó kém mạnh mẽ: Mẫu này cho phép chúng tránh tránh tight coupling giữa client và hệ thống con, và cũng giúp chúng ta tuân thủ nguyên tắc hướng đối tượng

8.1 Định nghĩa

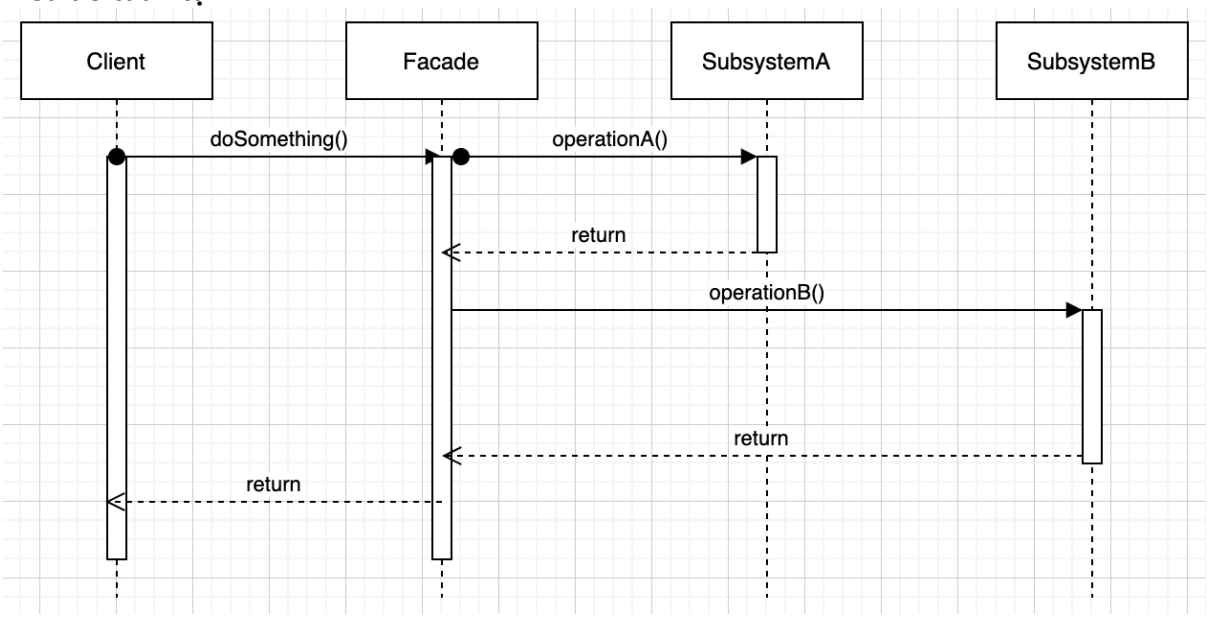
Facade Pattern cung cấp một giao diện duy nhất cho một tập hợp các giao diện trong một hệ thống con. Facade định là một giao diện cấp cao hơn giúp hệ thống con dễ sử dụng hơn.

8.2 Cấu trúc



Rõ ràng Pattern này là 1 ví dụ cho nguyên tắc Principle of Least Knowledge.

- **Biểu đồ tuần tự:**



8.3 Khi nào sử dụng Facade Pattern?.

- Sử dụng mẫu này khi bạn cần có một giao diện hạn chế nhưng đơn giản cho một hệ thống con phức tạp.
    - Thông thường, các hệ thống con trở nên phức tạp hơn theo thời gian. Ngay cả việc áp dụng các mẫu thiết kế thường dẫn đến việc tạo ra nhiều lớp hơn. Một hệ thống con có thể trở nên linh hoạt hơn và dễ dàng sử dụng lại trong các ngữ cảnh khác nhau, nhưng số lượng cấu hình và mã soạn sẵn mà nó yêu cầu từ khách hàng ngày càng lớn hơn. Facade cố gắng khắc phục sự cố này bằng cách cung cấp một lối tắt đến các tính năng được sử dụng nhiều nhất của hệ thống con phù hợp với các yêu cầu của khách hàng.

- Sử dụng Facade Pattern khi bạn muốn cấu trúc một hệ thống con thành các layers.
    - Tạo các Facade để xác định các điểm vào cho mỗi cấp của một hệ thống con. Bạn có thể loose coupling giữa nhiều hệ thống con bằng cách yêu cầu chúng chỉ giao tiếp thông qua các Facade pattern.

- Khi người sử dụng phụ thuộc nhiều vào các lớp cài đặt. Việc áp dụng Façade Pattern sẽ tách biệt hệ thống con của người dùng và các hệ thống con khác, do đó **tăng khả năng độc lập và khả chuyển** của hệ thống con, dễ chuyển đổi nâng cấp trong tương lai.

8.4 Ưu nhược điểm.

- Bạn có thể tách mã của mình khỏi một hệ thống con phức tạp

- Một facade có thể trở thành một đối tượng couple với tất cả các lớp của một ứng dụng.

8.5 Mối quan hệ với các Pattern khác.

- Facade định nghĩa một giao diện mới cho các đối tượng hiện có, trong khi Adapter cố gắng làm cho giao diện hiện có có thể sử dụng được. Adapter thường chỉ bao bọc một đối tượng, trong khi Facade hoạt động với toàn bộ hệ thống con của các đối tượng.

- Abstract Factory có thể phục vụ như một giải pháp thay thế cho Facade khi bạn chỉ muốn ẩn cách các đối tượng hệ thống con được tạo ra khỏi mã máy khách.

- Một lớp Facade thường có thể được chuyển đổi thành một Singleton vì một đối tượng mặt tiền duy nhất là đủ trong hầu hết các trường hợp.

- Facade tương tự như Proxy ở chỗ cả hai đều tạo giao diện cho 1 thực thể phức tạp. Không giống như Facade, Proxy có giao diện giống với đối tượng dịch vụ của nó, điều này làm cho chúng có thể hoán đổi cho nhau.

9. Template Method Pattern

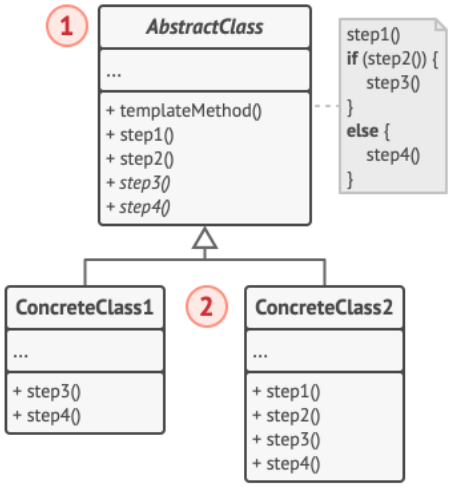
Trong quá trình phát triển ứng dụng, chúng ta có các component khác nhau có sự tương đồng đáng kể, nhưng chúng không sử dụng interface/ abstract class chung, dẫn đến code duplicate ở nhiều nơi. Nếu muốn thay đổi chung cho tất cả component, chúng ta phải đi sửa ở từng nơi trong component, làm tốn nhiều chi phí không cần thiết. Một trong những cách để giải quyết vấn đề này là sử dụng **Template Method Pattern**.

9.1 Định nghĩa.

Là một bộ khung của một thuật toán trong một chức năng, chuyển giao việc thực hiện nó cho các lớp con. Mẫu Template Method cho phép lớp con *định nghĩa lại cách thực hiện* của một thuật toán, mà *không phải thay đổi cấu trúc thuật toán*".

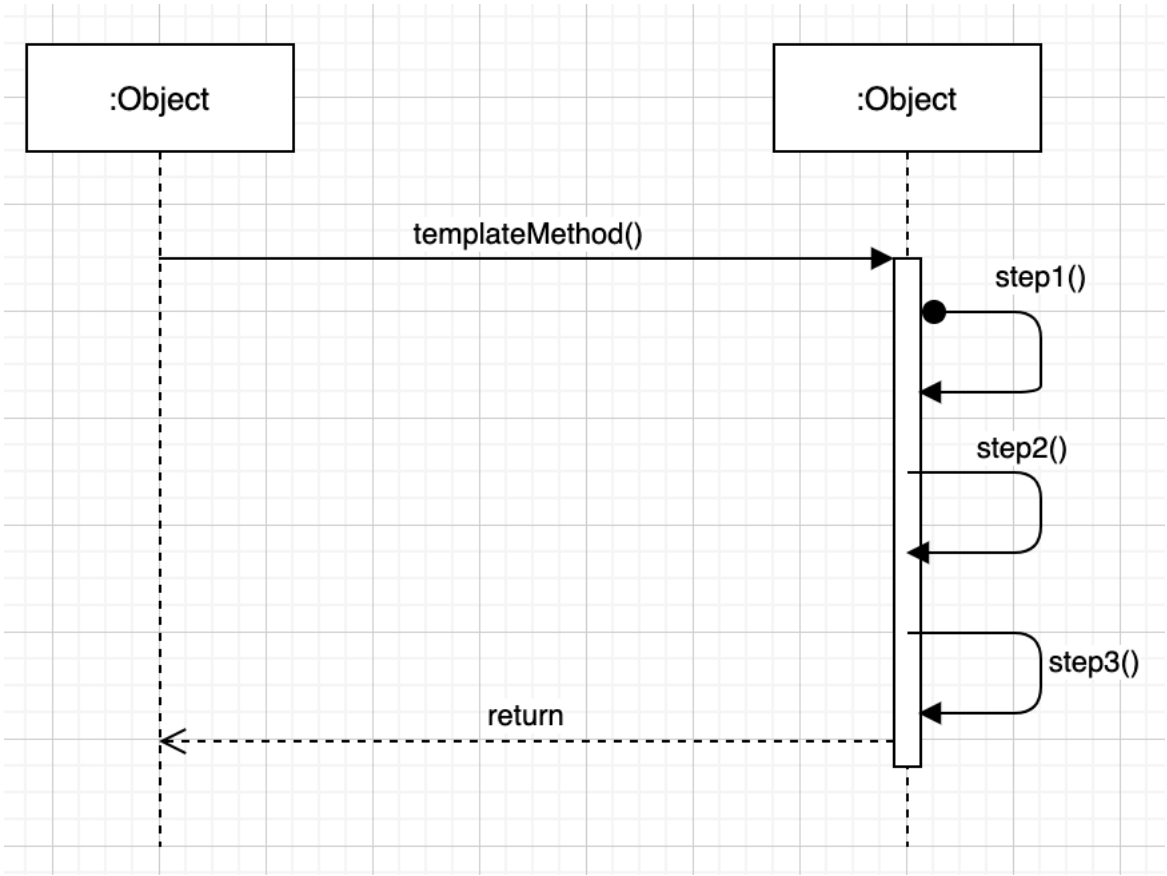
Để tránh các các lớp con thay đổi thuật toán trong template method, thì nên khai báo các method ở dạng Final.

9.2 Cấu trúc



Các thành phần trong cấu trúc bao gồm:

- Các lớp Concrete có thể ghi đè tất cả các bước, nhưng thể ghi đè template method.
- **Abstract Class**:
    - Định nghĩa các phương thức trừu tượng cho từng bước có thể được điều chỉnh hay ghi đè bởi các lớp con.
    - Cài đặt một **phương thức duy nhất(TEMPLATE METHOD)** điều khiển thuật toán và gọi các bước riêng lẻ đã được cài đặt ở các lớp con.



9.3 Khi nào sử dụng Template method?.

- Khi có một thuật toán với nhiều bước và mong muốn cho phép tùy chỉnh chúng trong lớp con.
- Mong muốn chỉ có một triển khai phương thức trừu tượng duy nhất của một thuật toán.
- Mong muốn hành vi chung giữa các lớp con nên được đặt ở một lớp chung.
- Các lớp cha có thể gọi các hành vi trong các lớp con của chúng một cách thống nhất (step by step).

9.4 Ưu nhược điểm

Ưu điểm:

- Bạn chỉ có thể cho phép khách hàng ghi đè một số phần nhất định của một thuật toán lớn, giúp chúng ít bị ảnh hưởng bởi những thay đổi xảy ra với các phần khác của thuật toán.
- Bạn có thể kéo mã trùng lặp của các lớp con vào chỉ chung một lớp cha.
- Đảm bảo nguyên tắc **HOOLYWOOD PRINCIPLE**

Nhược điểm:

- Một số khách hàng có thể bị giới hạn bởi khung được cung cấp của một thuật toán.
- Các template method có xu hướng khó bảo trì hơn khi chúng có nhiều bước.

9.5 Mối quan hệ với các Pattern khác.

- **Factory method** là một trường hợp đặc biệt của template method. Đồng thời, Factory method có thể đóng vai trò là một bước trong large template method.

- **Template method** dựa trên sự kế thừa: nó cho phép bạn thay đổi các phần của một thuật toán bằng cách mở rộng các phần đó trong các lớp con. **Stradegy** dựa trên nguyên lý kết tập: bạn có thể thay đổi các phần trong hành vi của đối tượng bằng cách cung cấp cho đối tượng các chiến lược khác nhau tương ứng với hành vi đó. Do đó template method mang tính tĩnh. Stradegy hoạt động ở cấp độ đối tượng, cho phép bạn chuyển đổi hành vi trong thời gian chạy.

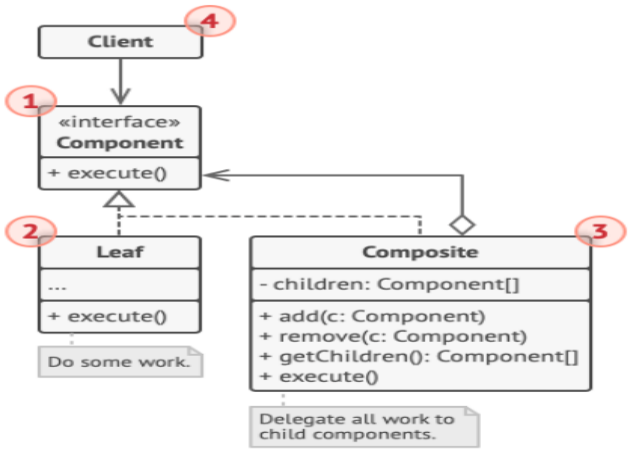
10. Composite Pattern

10.1 Định nghĩa

Composite là một mẫu thiết kế thuộc nhóm cấu trúc (**Structural Pattern**). Composite Pattern cho phép bạn sắp xếp xác đối tượng thành cấu trúc cây để mô ta quan hệ thứ tự

Khách hàng có thể **xử lý một nhóm đối tượng tương tự theo cách xử lý 1 object**.

10.2 Cấu trúc



Các thành phần của cấu trúc gồm có:

- **Component interface** mô tả các hoạt động giống nhau cho các lá và composite của cây.
- **Leaf** là phần tử cơ bản của cây và không có components.

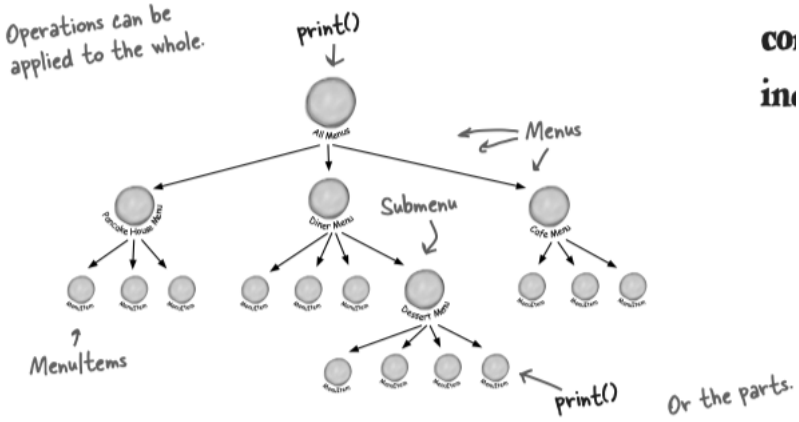
Thông thường, các thành phần lá thực hiện hầu hết các công việc thực sự, vì chúng không có bất kỳ ai để ủy quyền công việc.

- **Composite** là một phần tử có các phần tử con: lá hoặc các composite khác. Một composite không biết các cài đặt cụ thể của các con của nó. Nó chỉ tương tác với con thông qua component interface.

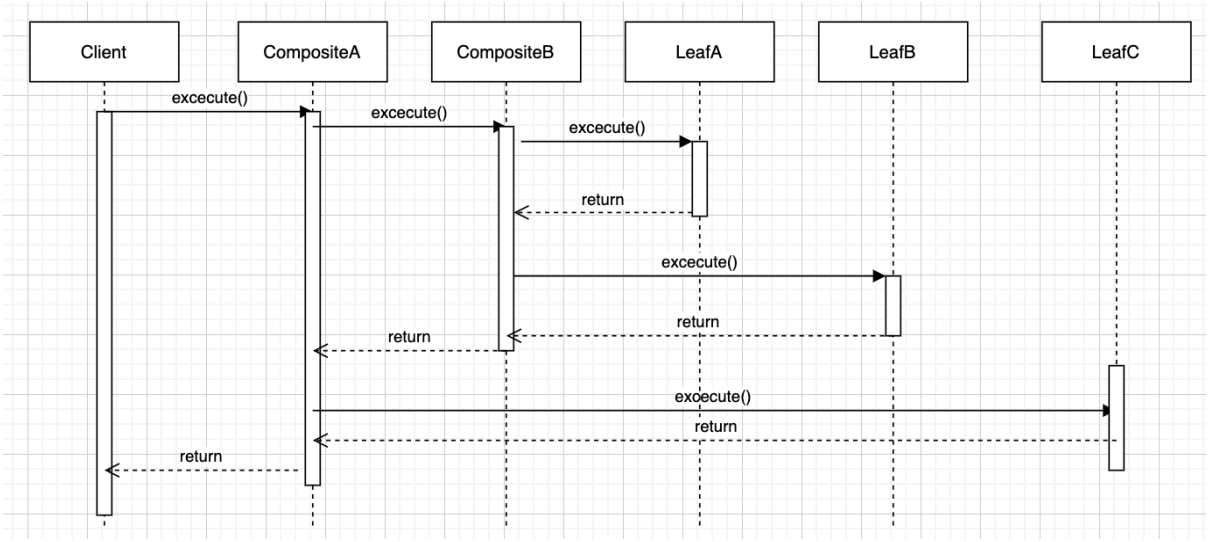
Khi nhận được yêu cầu, một composite sẽ ủy quyền công việc cho các phần tử con của nó, xử lý các kết quả trung gian và sau đó trả lại kết quả cuối cùng cho khách hàng.

- **Client** làm việc với tất cả các phần tử chỉ thông qua component interface(tuân thủ nguyên tắc Program to interface, not to implementation). Do đó, khách hàng có thể làm việc theo cùng một cách với cả các phần tử đơn giản hoặc phức tạp của cây.

Ví dụ dưới đây cho ta một cây khi mỗi children là một item, và mỗi composite là một menu. Khách hàng có thể gọi phương thức print() duyệt qua tất cả các phần từ bất kể nó là menu hay là item.



- **Biểu đồ tuần tự:**



10.3 Khi nào sử dụng Composite Pattern

- Composite Pattern chỉ nên được áp dụng khi nhóm đối tượng phải hoạt động như một đối tượng duy nhất (theo cùng một cách).
- Composite Pattern có thể được sử dụng để tạo ra một cấu trúc giống như cấu trúc cây.

10.4 Ưu nhược điểm

Ưu điểm:

- Bạn có thể làm việc với các cấu trúc cây phức tạp thuận tiện hơn: sử dụng đa hình và đệ quy để có lợi cho bạn.
- Đảm bảo open/closed principle: Bạn có thể thêm các loại phần tử mới vào cây mà không cần phải thay đổi mã hiện có.

Nhược điểm:

- Có thể khó cung cấp một giao diện chung cho các lớp có chức năng khác nhau quá nhiều. Trong một số trường hợp nhất định, bạn cần phải tổng quát hóa quá mức giao diện thành phần, khiến nó khó hiểu hơn.

10.5 Mối quan hệ với các Pattern khác.

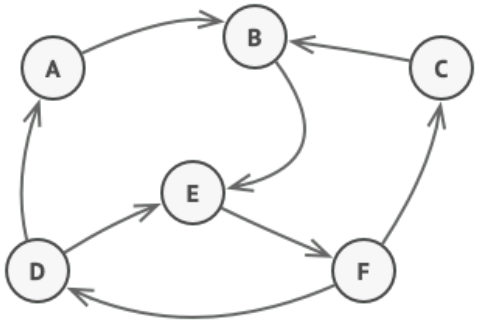
- Có thể dùng Iterator để duyệt composite Pattern

- Composite và Decorator có sơ đồ cấu trúc tương tự nhau vì cả hai đều dựa vào thành phần đệ quy để tổ chức một số lượng đối tượng .

    - Decorator giống như Composite nhưng chỉ có một child component so với composite thì có nhiều. Có một sự khác biệt đáng kể khác: Decorator thêm các trách nhiệm bổ sung cho đối tượng được bao bọc, trong khi Composite chỉ "tổng hợp" các kết quả con của nó.

    - Tuy nhiên, hai mẫu cũng có thể hợp tác: bạn có thể sử dụng Decorator để mở rộng hành vi của một đối tượng cụ thể trong cây Composite.

11. State Pattern

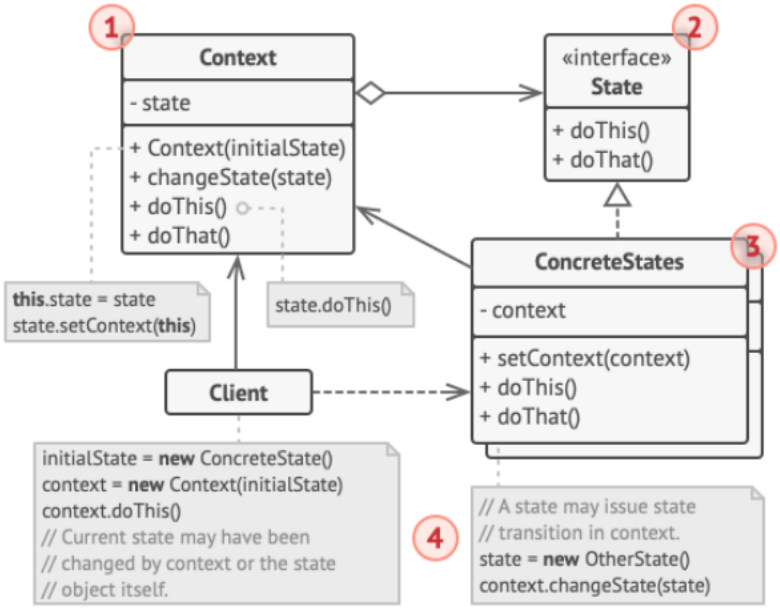


Hình bên trên là một máy trạng thái bao gồm hữu hạn các trạng thái khác nhau. Trong bất kỳ trạng thái nào, thì máy cũng hoạt động theo cách khác nhau có thể được chuyển từ trạng thái này sang trạng thái khác ngay lập tức. Tuy nhiên, tùy thuộc vào trạng thái hiện tại, máy có thể chuyển hoặc không chuyển sang một số trạng thái khác. Các quy tắc chuyển trạng thái này, được gọi là Transitions, cũng là hữu hạn và được xác định trước. Ta có thể dùng state Pattern để thiết kế mô hình này.

11.1 Định nghĩa.

State Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó cho phép một đối tượng thay đổi hành vi của nó khi trạng thái nội bộ của nó thay đổi. Đối tượng sẽ xuất hiện để thay đổi lớp của nó.

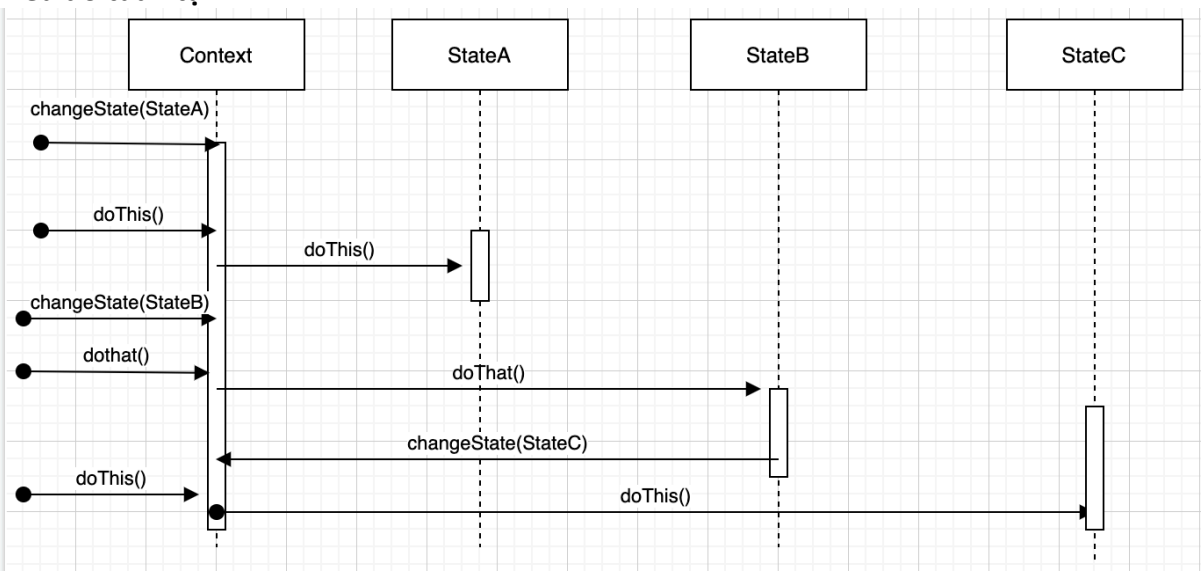
11.2 Cấu trúc



Các thành phần tham gia State Pattern:

- **Context** : được sử dụng bởi Client. Client không truy cập trực tiếp đến State của đối tượng. Lớp Context này chứa thông tin của ConcreteState object, cho biết hành vi nào tương ứng với trạng thái nào hiện đang được thực hiện.
- **State** : là một interface hoặc abstract class xác định các đặc tính cơ bản của tất cả các đối tượng ConcreteState. Chúng sẽ được sử dụng bởi đối tượng Context để truy cập chức năng có thể thay đổi.
- **ConcreteState** : cài đặt các method của State. Mỗi ConcreteState có thể thực hiện logic và hành vi của riêng nó tùy thuộc vào Context. Chú ý khi thêm các trạng thái mới không làm ảnh hưởng đến trạng thái chức năng khác.

  - **Biểu đồ tuần tự:**



11.3 Khi nào sử dụng State Pattern?.

- Sử dụng State pattern khi bạn có một đối tượng hành động khác nhau tùy thuộc vào trạng thái hiện tại của nó, số lượng trạng thái là rất lớn và mã của trạng thái cụ thể thường xuyên thay đổi.
    - Mẫu đề xuất rằng bạn trích xuất tất cả mã dành riêng cho trạng thái thành một tập hợp các lớp riêng biệt. Do đó, bạn có thể thêm các trạng thái mới hoặc thay đổi các trạng thái hiện có một cách độc lập với nhau, giảm chi phí bảo trì. (Tuân thủ nguyên tắcOpen/closed Principle).

- Sử dụng mẫu khi bạn có một lớp với rất nhiều điều kiện làm thay đổi cách lớp hoạt động theo các giá trị hiện tại của các trường của lớp. Mẫu trạng thái cho phép bạn trích xuất các nhánh của các điều kiện này thành các phương thức của các lớp trạng thái tương ứng.
- Sử dụng State pattern khi bạn có nhiều mã trùng lặp qua các trạng thái và chuyển đổi của máy trạng thái dựa trên điều kiện. State Pattern cho phép tạo ra cấu trúc phân cấp lớp trạng thái và giảm duplicate code bằng cách trích xuất mã chung vào các lớp cơ sở trừu tượng(Khá giống với template method ở đoạn này).

11.4 Ưu nhược điểm của State Pattern

Ưu điểm:

- Đảm bảo nguyên tắc Single Responsibility: Tổ chức mã liên quan đến các trạng thái cụ thể thành các lớp riêng biệt.
- Đảm bảo open/closed Principle: Thêm các trạng thái mới mà không thay đổi các lớp State hiện có hoặc Context.
- Đơn giản hóa mã của Context bằng cách loại bỏ các điều kiện máy trạng thái cồng kềnh.

Nhược điểm:

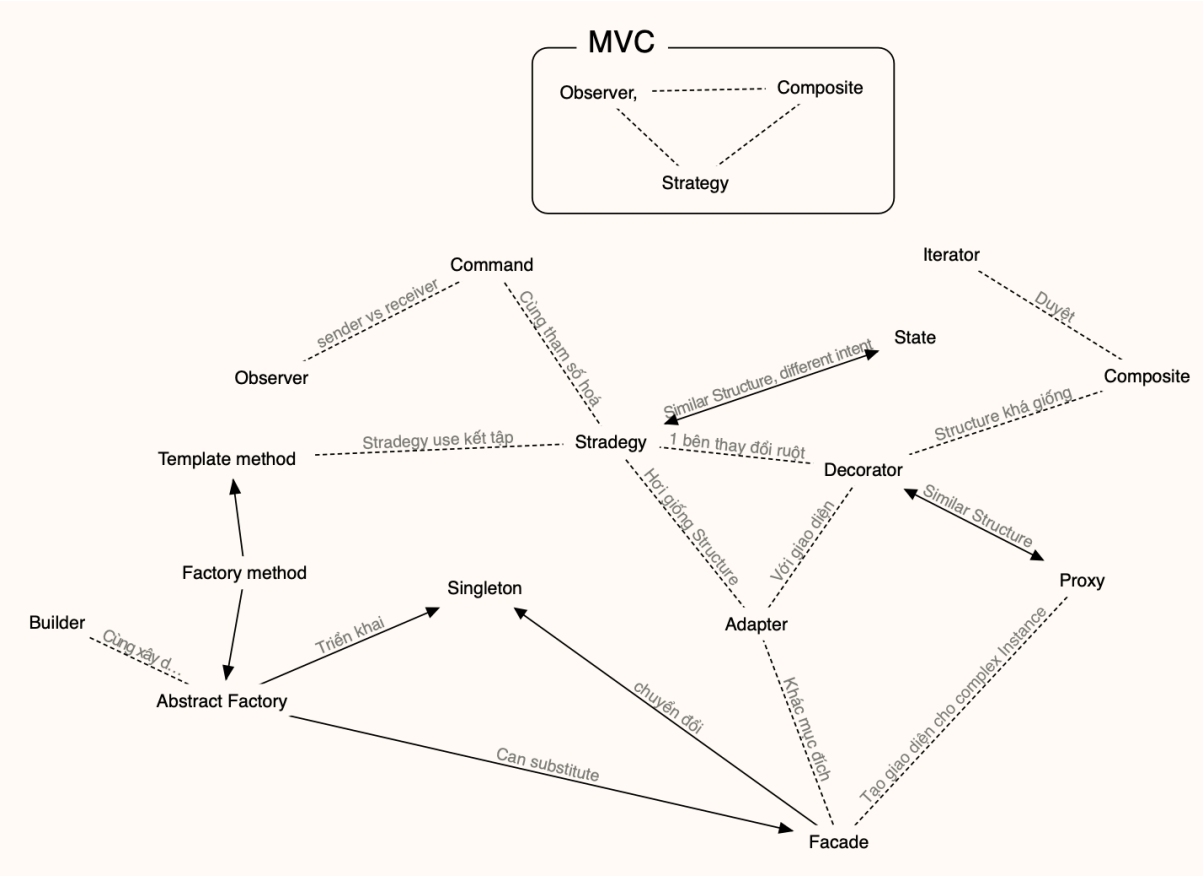
- Nếu chỉ có một hay hai trạng thái, hay các trạng thái ít thay đổi thì không nên dùng state Pattern

11.5 Mối quan hệ với các Pattern :

- Bridge, State, Strategy (và ở một mức độ nào đó là Adapter) có cấu trúc rất giống nhau. Thật vậy, tất cả các mẫu này đều dựa trên nguyên lý kết tập, tức là ủy thác công việc cho các đối tượng khác. Tuy nhiên, tất cả chúng đều giải quyết các vấn đề khác nhau.
- State có thể được coi là một phần mở rộng của Stradegy. Cả hai mẫu đều dựa trên thành phần: chúng thay đổi hành vi của Context bằng cách ủy quyền một số công việc cho các đối tượng trợ giúp. Stradegy làm cho các đối tượng này hoàn toàn độc lập và không biết về nhau. Tuy nhiên, State không hạn chế sự phụ thuộc giữa các trạng thái cụ thể, cho phép chúng thay đổi trạng thái của context theo ý muốn.

Chương 3: Tổng kết

1. Mối liên hệ giữa một số các Pattern.



2. Kết luận

Việc tìm hiểu cách Sử dụng Design pattern là một trong những cách giúp chúng ta có thể bảo trì, mở rộng, và tái sử dụng. Nó giúp chúng ta cách để làm sao để xây dựng một hệ thống với chất lượng thiết kế tốt. Nó đưa ra giải pháp chung và từ đó có thể áp dụng vào những hoàn cảnh riêng đối với mỗi phần mềm ứng dụng.

Em xin cảm ơn Thầy Nguyễn Bá Ngọc đã hướng dẫn, giúp đỡ em trong quá trình làm, để em có thể hoàn thành báo cáo này, và học thêm được kiến thức mới.

Tài liệu tham khảo

1. https://www.amazon.com/Head-First-Design-Patterns-Brain-Friendly/dp/0596007124
2. https://gpcoder.com/
3. https://cuongquach.com/ebook-head-first-java-2nd-edition-pdf.html
4. https://refactoring.guru/
5. https://www.journaldev.com/31902/gangs-of-four-gof-design-patterns