

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH  
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN  
KHOA CÔNG NGHỆ PHẦN MỀM



BÁO CÁO CUỐI KỲ  
**Môn học: MẪU THIẾT KẾ**

**Giảng viên hướng dẫn:**

Ths. Trần Anh Dũng

**Nhóm sinh viên thực hiện:**

Phan Xuân Quang : 20521008

Trần Văn Thiệt : 20521956

Nguyễn Ngọc Trinh : 20520824

Thành phố Hồ Chí Minh, tháng 06 năm 2023

## **LỜI CẢM ƠN**

Nhóm em trân trọng cảm ơn thầy Trần Anh Dũng đã tạo điều kiện và hướng dẫn chúng em để chúng em có cơ hội tìm hiểu về môn học này. Chúng em cũng cảm ơn tất cả các bạn cùng lớp đã chia sẻ những tìm hiểu và kinh nghiệm về những kiến thức bổ ích liên quan đến Mẫu thiết kế. Trải qua 14 tuần học cùng với những buổi seminar đầy mới lạ đã cho chúng em cái nhìn chi tiết về các Mẫu thiết kế mới lạ và hấp dẫn, để chúng em sử dụng sau này. Chúng em tin rằng những kiến thức này sẽ hỗ trợ chúng em trên con đường sự nghiệp sắp tới. Một lần nữa, cảm ơn thầy và các bạn đã đồng hành và hỗ trợ nhóm chúng em trong môn học này.

Thành phố Hồ Chí Minh, ngày 30 tháng 06 năm 2023

Đại diện nhóm

Phan Xuân Quang

## NHẬN XÉT CỦA GIẢNG VIÊN

# MỤC LỤC

<b>LỜI CẢM ƠN.....</b>	<b>2</b>
<b>NHẬN XÉT CỦA GIẢNG VIÊN .....</b>	<b>3</b>
<b>DANH MỤC HÌNH ẢNH.....</b>	<b>10</b>
<b>CHƯƠNG 1. KHÁI NIỆM.....</b>	<b>13</b>
1.1. GIỚI THIỆU DESIGN PATTERN .....	13
1.2. LỢI ÍCH CỦA VIỆC SỬ DỤNG DESIGN PATTERN .....	13
1.3. PHÂN LOẠI DESIGN PATTERN .....	13
<b>CHƯƠNG 2. CREATIONAL PATTERN.....</b>	<b>14</b>
2.1. SINGLETON .....	14
2.1.1. Tổng quan .....	14
2.1.2. Motivation.....	14
2.1.3. Đặc điểm .....	15
2.1.4. Khả năng ứng dụng .....	15
2.1.5. Hết quả .....	16
2.1.6. Các mẫu liên quan.....	16
2.2. FACTORY METHOD .....	16
2.2.1. Tổng quan .....	16
2.2.2. Motivation.....	16
2.2.3. Đặc điểm .....	17
2.2.4. Khả năng ứng dụng .....	17
2.2.5. Hết quả .....	18
2.2.6. Các mẫu thiết kế liên quan.....	18
2.3. ABSTRACT FACTORY .....	18
2.3.1. Tổng quan .....	19
2.3.2. Motivation.....	19
2.3.3. Đặc điểm .....	20
2.3.4. Khả năng ứng dụng .....	21
2.3.5. Hết quả .....	21
2.3.6. Các mẫu thiết kế liên quan.....	22
2.4. BUILDER .....	22

2.4.1. Tổng quan .....	22
2.4.2. Motivation.....	22
2.4.3. Đặc điểm .....	22
2.4.4. Khả năng ứng dụng.....	23
2.4.5. Hết quả .....	24
2.4.6. Các mẫu thiết kế liên quan.....	24
2.5. PROTOTYPE.....	24
2.5.1. Tổng quan .....	24
2.5.2. Motivation.....	24
2.5.3. Đặc điểm .....	25
2.5.4. Khả năng ứng dụng.....	27
2.5.5. Hết quả .....	27
2.5.6. Các mẫu thiết kế liên quan.....	28
<b>CHƯƠNG 3. STRUCTURAL PATTERN.....</b>	<b>29</b>
3.1. ADAPTER .....	29
3.1.1. Tổng quan .....	29
3.1.2. Motivation.....	30
3.1.3. Đặc điểm .....	30
3.1.4. Khả năng ứng dụng.....	31
3.1.5. Hết quả .....	31
3.1.6. Các mẫu thiết kế liên quan.....	32
3.2. BRIDGE .....	32
3.2.1. Tổng quan .....	32
3.2.2. Motivation.....	32
3.2.3. Đặc điểm .....	33
3.2.4. Khả năng ứng dụng: .....	34
3.2.5. Hết quả .....	34
3.2.6. Các mẫu thiết kế liên quan.....	35
3.3. COMPOSITE .....	35
3.3.1. Tổng quan .....	36
3.3.2. Motivation.....	36
3.3.3. Đặc điểm .....	37

3.3.4. Khả năng ứng dụng .....	38
3.3.5. Hết quả .....	38
3.3.6. Các mẫu thiết kế liên quan .....	39
<b>3.4. DECORATOR.....</b>	<b>39</b>
3.4.1. Tổng quan .....	39
3.4.2. Motivation.....	40
3.4.3. Đặc điểm .....	40
3.4.4. Khả năng ứng dụng .....	41
3.4.5. Hết quả .....	41
3.4.6. Các mẫu thiết kế liên quan .....	42
<b>3.5. FACADE .....</b>	<b>43</b>
3.5.1. Tổng quan .....	43
3.5.2. Motivation.....	43
3.5.3. Đặc điểm .....	45
3.5.4. Khả năng ứng dụng .....	45
3.5.5. Hết quả .....	46
3.5.6. Các mẫu thiết kế liên quan .....	46
<b>3.6. FLYWEIGHT.....</b>	<b>47</b>
3.6.1. Tổng quan .....	47
3.6.2. Motivation.....	47
3.6.3. Đặc điểm .....	48
3.6.4. Khả năng ứng dụng .....	49
3.6.5. Hết quả .....	49
3.6.6. Các mẫu thiết kế liên quan .....	50
<b>3.7. PROXY .....</b>	<b>50</b>
3.7.1. Tổng quan .....	50
3.7.2. Motivation.....	50
3.7.3. Đặc điểm .....	51
3.7.4. Khả năng ứng dụng .....	52
3.7.5. Hết quả .....	53
3.7.6. Các mẫu thiết kế liên quan .....	53
<b>CHƯƠNG 4. BEHAVIOR PATTERN .....</b>	<b>54</b>

4.1. CHAIN OF RESPONSIBILITY .....	54
4.1.1. Tông quan .....	54
4.1.2. Motivation.....	54
4.1.3. Đặc điểm .....	57
4.1.4. Khả năng ứng dụng.....	58
4.1.5. Hết quả .....	58
4.1.6. Các mẫu thiết kế liên quan.....	59
4.2. COMMAND .....	59
4.2.1. Tông quan .....	59
4.2.2. Motivation.....	59
4.2.3. Đặc điểm .....	62
4.2.4. Khả năng ứng dụng.....	63
4.2.5. Hết quả .....	63
4.2.6. Các mẫu thiết kế liên quan.....	64
4.3. INTERPRETER .....	65
4.3.1. Tông quan .....	65
4.3.2. Motivation.....	66
4.3.3. Đặc điểm .....	66
4.3.4. Khả năng ứng dụng.....	68
4.3.5. Hết quả .....	68
4.3.6. Các mẫu thiết kế liên quan.....	68
4.4. MEDIATOR .....	69
4.4.1. Tông quan .....	69
4.4.2. Motivation.....	69
4.4.3. Đặc điểm .....	70
4.4.4. Khả năng ứng dụng.....	71
4.4.5. Hết quả .....	72
4.4.6. Các mẫu thiết kế liên quan.....	72
4.5. MEMENTO .....	73
4.5.1. Tông quan .....	73
4.5.2. Motivation.....	73
4.5.3. Đặc điểm .....	75

4.5.4. Khả năng ứng dụng .....	75
4.5.5. Hệ quả .....	76
4.5.6. Các mẫu thiết kế liên quan .....	76
4.6. OBSERVER .....	76
4.6.1. Tổng quan .....	76
4.6.2. Motivation .....	76
4.6.3. Đặc điểm .....	77
4.6.4. Khả năng ứng dụng .....	78
4.6.5. Hệ quả .....	78
4.6.6. Các mẫu thiết kế liên quan .....	79
4.7. STATE .....	79
4.7.1. Tổng quan .....	79
4.7.2. Motivation .....	79
4.7.3. Đặc điểm .....	80
4.7.4. Khả năng ứng dụng .....	80
4.7.5. Hệ quả .....	81
4.7.6. Các mẫu thiết kế liên quan .....	81
4.8. STRATEGY .....	81
4.8.1. Tổng quan .....	81
4.8.2. Motivation .....	82
4.8.3. Đặc điểm .....	82
4.8.4. Khả năng ứng dụng .....	83
4.8.5. Hệ quả .....	83
4.8.6. Các mẫu thiết kế liên quan .....	83
4.9. TEMPLATE METHOD .....	84
4.9.1. Tổng quan .....	84
4.9.2. Motivation .....	84
4.9.3. Đặc điểm .....	84
4.9.4. Khả năng ứng dụng .....	85
4.9.5. Hệ quả .....	86
4.9.6. Các mẫu thiết kế liên quan .....	86
4.10. VISITOR .....	86

4.10.1. Tổng quan .....	87
4.10.2. Motivation.....	87
4.10.3. Đặc điểm .....	87
4.10.4. Khả năng ứng dụng .....	88
4.10.5. Hết quả .....	88
4.10.6. Các mẫu thiết kế liên quan.....	89
4.11. ITERATOR.....	89
4.11.1. Tổng quan .....	89
4.11.2. Motivation.....	89
4.11.3. Đặc điểm .....	92
4.11.4. Khả năng ứng dụng .....	93
4.11.5. Hết quả .....	93
4.11.6. Các mẫu thiết kế liên quan.....	93

## DANH MỤC HÌNH ẢNH

Hình 1.1 Các loại Design Pattern .....	13
Hình 2.1 Minh họa mẫu thiết kế Singleton.....	14
Hình 2.2 Minh họa giải pháp của Singleton .....	14
Hình 2.3 Sơ đồ lớp của mẫu thiết kế Singleton.....	15
Hình 2.4: Sơ đồ lớp của mẫu thiết kế Factory Method .....	17
Hình 2.5: Minh họa Abstract Factory .....	18
Hình 2.6: Các biến thể với các loại sản phẩm .....	19
Hình 2.7: Sơ đồ lớp của mẫu thiết kế Abstract Factory .....	20
Hình 2.8: Sơ đồ lớp của mẫu thiết kế Builder .....	23
Hình 2.9: Sơ đồ lớp của mẫu thiết kế Prototype dạng cơ bản .....	25
Hình 2.10 Sơ đồ lớp của mẫu thiết kế Prototype dạng Prototype Registry.....	25
Hình 2.11: Cơ chế của Shallow Copy và Deep Copy .....	26
Hình 3.1 Minh họa mẫu thiết kế Adapter .....	29
Hình 3.2: Sơ đồ lớp của mẫu thiết kế Adapter dạng đối tượng.....	30
Hình 3.3: Sơ đồ lớp của mẫu thiết kế Adapter dạng lớp .....	30
Hình 3.4 Minh họa mẫu thiết kế Bridge .....	32
Hình 3.5: Sơ đồ lớp của mẫu thiết kế Bridge .....	33
Hình 3.6 Minh họa mẫu thiết kế Composite.....	35
Hình 3.7: Ví dụ minh họa về Composite .....	36
Hình 3.8: Sơ đồ lớp của mẫu thiết kế Composite.....	37
Hình 3.9: Sơ đồ lớp của mẫu thiết kế Decorator .....	40
Hình 3.10 Minh họa mẫu thiết kế Facade.....	43
Hình 3.11: Mô tả Façade .....	44
Hình 3.12: Sơ đồ lớp của mẫu thiết kế Facade.....	45
Hình 3.13: Tiêu thụ RAM của game bắn súng khi tạo mới từng đối tượng.....	47
Hình 3.14: Áp dụng Flyweight vào việc tạo đối tượng cho game bắn súng .....	47
Hình 3.15: Sơ đồ lớp của mẫu thiết kế Flyweight.....	48
Hình 3.16: Sơ đồ lớp và trình tự của mẫu thiết kế Flyweight theo GoF .....	48
Hình 3.17: Sơ đồ lớp của mẫu thiết kế Proxy.....	51
Hình 4.1: Mô tả về CoR.....	54

Hình 4.2: Quy trình từ khi Request đến khi tới được Ordering System.....	54
Hình 4.3: Ví dụ khi các hành động cần thực hiện ngày càng lớn lên.....	55
Hình 4.4: Quá trình xử lý Request qua các Handler.....	55
Hình 4.5: Mô tả chuỗi có thể được hình thành từ nhánh của cây.....	56
Hình 4.6: Mô tả về quá trình gọi hỗ trợ được thông qua bởi nhiều điện thoại viên.....	56
Hình 4.7 Sơ đồ lớp của mẫu thiết kế CoR.....	57
Hình 4.9: Mô tả về động lực của CoR.....	58
Hình 4.10: Hộp thoại có các nút.....	60
Hình 4.11: Button với những chức năng khác nhau.....	60
Hình 4.12: Với mỗi loại button lại có code riêng của nó .....	60
Hình 4.13: Mỗi button gửi riêng lẻ với từng mục đích .....	61
Hình 4.14: Các button được thông nhất về cùng một Command .....	61
Hình 4.15: Sơ đồ lớp mức phân tích.....	62
Hình 4.16: Sơ đồ lớp của mẫu thiết kế Command .....	62
Hình 4.17 Minh họa mẫu thiết kế Interpreter .....	65
Hình 4.18: Chuyển đổi các định dạng thời gian .....	66
Hình 4.19: Sơ đồ lớp của mẫu thiết kế Interpreter .....	67
Hình 4.20 Sơ đồ quan hệ giữa các component khi không dùng Mediator .....	69
Hình 4.21 Sơ đồ quan hệ giữa các component khi dùng Mediator .....	70
Hình 4.22: Sơ đồ lớp của mẫu thiết kế Mediator .....	70
Hình 4.23 Minh họa mẫu thiết kế Memento.....	73
Hình 4.24: Mô tả về việc lưu lại bản chụp của đối tượng .....	74
Hình 4.25 Mô tả về tương tác giữa các thành phần trong Memento .....	74
Hình 4.26 Cấu trúc mẫu của Memento.....	75
Hình 4.27 Minh họa bài toán thực tế của Observer .....	77
Hình 4.28 Cấu trúc mẫu của Observer .....	78
Hình 4.29 Mô hình trạng thái của tài liệu.....	80
Hình 4.30 Cấu trúc mẫu của State .....	80
Hình 4.31 Minh họa mẫu thiết kế Strategy.....	81
Hình 4.32: Sơ đồ lớp của mẫu thiết kế Strategy .....	82
Hình 4.33: Sơ đồ lớp của mẫu thiết kế Template Method .....	85
Hình 4.34 Minh họa mẫu thiết kế Visitor .....	86

Hình 4.35: Sơ đồ lớp của mẫu thiết kế Visitor .....	87
Hình 4.36 Một số dạng tập hợp .....	90
Hình 4.37 Mô hình DFS và BFS .....	90
Hình 4.38 Sơ đồ lớp của DFS và BFS.....	91
Hình 4.39 Cấu trúc mẫu của Iterator .....	92

# Chương 1. KHÁI NIỆM

## 1.1. Giới thiệu Design Pattern



Hình 1.1 Các loại Design Pattern

Mẫu thiết kế (Design Pattern) là những giải pháp đặc trưng cho những vấn đề hay xảy ra trong thiết kế phần mềm. Chúng chỉ như những bản phác thảo mà chúng ta cần tùy chỉnh lại cho phù hợp với bài toán cụ thể.

Mẫu thiết kế thường bị nhầm lẫn với thuật toán, vì cả hai đều mô tả cho những giải pháp cho những vấn đề thường thấy. Trong khi một thuật toán luôn định nghĩa một cách rõ ràng một bộ hành động có thể đạt được một vài mục tiêu, một mẫu thiết kế chỉ là sự mô tả ở cấp bậc tổng quát. Vì thế, code của cùng một mẫu thiết kế áp dụng cho 2 chương trình khác nhau có thể khác nhau.

## 1.2. Lợi ích của việc sử dụng Design Pattern

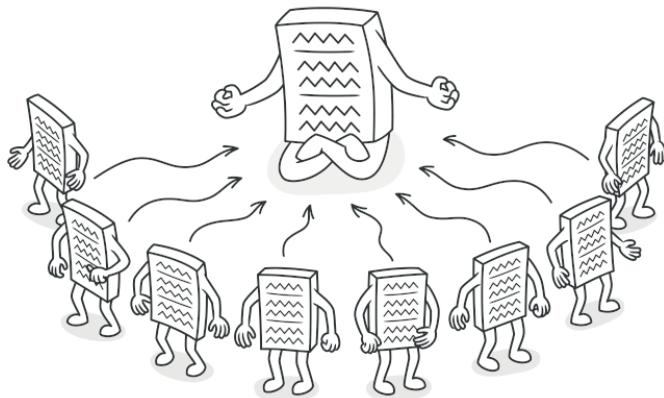
- Tăng tốc độ phát triển phần mềm.
- Giúp code trở nên tường minh, dễ đọc cho chính bản thân và đồng đội.
- Tái sử dụng code.
- Hạn chế lỗi tiềm ẩn.
- Dễ bảo trì, nâng cấp hoặc mở rộng.

## 1.3. Phân loại Design Pattern

- **Creational Pattern:** Các mẫu này cung cấp các cơ chế tạo đối tượng khác nhau, giúp tăng tính linh hoạt và khả năng tái sử dụng code hiện có.
- **Structural Pattern:** Những này giải thích cách tập hợp các đối tượng và lớp thành các cấu trúc lớn hơn trong khi vẫn giữ cho các cấu trúc này linh hoạt và hiệu quả.
- **Behavioral Pattern:** Các mẫu này liên quan đến các thuật toán và sự phân công trách nhiệm giữa các đối tượng.

## Chương 2. CREATIONAL PATTERN

### 2.1. Singleton



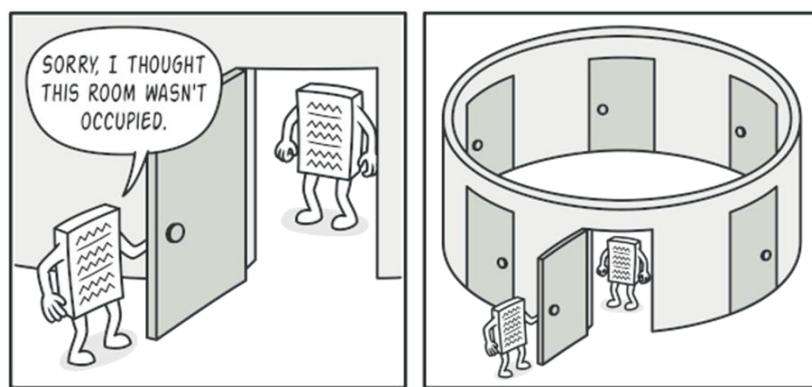
Hình 2.1 Minh họa mẫu thiết kế Singleton

#### 2.1.1. Tổng quan

Mẫu thiết kế Singleton đảm bảo rằng một lớp chỉ có một thể hiện (instance) duy nhất. Do thể hiện này có tiềm năng sử dụng trong suốt chương trình, nên mẫu thiết kế Singleton cũng cung cấp một điểm truy cập toàn cục đến nó.

#### 2.1.2. Motivation

Trong quá trình code chúng ta sẽ gặp những trường hợp mong muốn có một đối tượng duy nhất tồn tại để truy xuất đến nhiều nơi. Ví dụ khi làm việc với MVP, MVVM, ... chúng ta muốn tạo ra 1 đối tượng repository để vừa có thể truy xuất đến database, vừa có thể truy xuất đến các class thao tác API. Trong trường hợp này chúng ta có thể áp dụng Singleton Pattern để giải quyết vấn đề này. Còn rất nhiều trường hợp trong code mà chúng ta muốn tạo ra 1 instance suốt project thì chúng ta cũng có thể áp dụng Singleton Pattern.



Hình 2.2 Minh họa giải pháp của Singleton

Singleton giải quyết được 2 vấn đề cùng lúc:

- Đảm bảo rằng một lớp chỉ có một instance duy nhất. Tại sao mọi người lại muốn kiểm soát số lượng instance mà một lớp có? - Lý do phổ biến nhất cho điều này là kiểm soát quyền truy cập vào một số tài nguyên được chia sẻ.

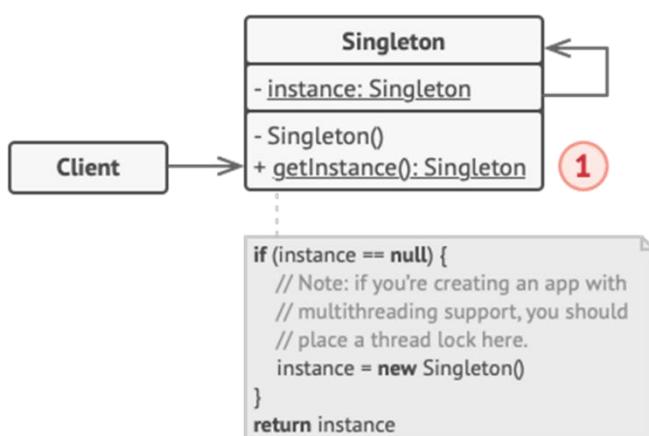
*Lưu ý: Khi thao tác với Singleton thì không thể thực hiện với một constructor thông thường vì một lời gọi constructor phải luôn trả về một đối tượng mới theo thiết kế. mà phải thao tác với method trả về instance của đối tượng.*

- Cung cấp một method truy cập mọi nơi cho instance đó. Nhờ vậy mà ta có thể gọi instance đã tạo ở bất cứ đâu trong chương trình.

*Lưu ý: Chỉ tạo method để truy cập instance, không được phép tạo method để thay đổi instance đó.*

### 2.1.3. Đặc điểm

#### 2.1.3.1. Cấu trúc mẫu



Hình 2.3 Sơ đồ lớp của mẫu thiết kế Singleton

#### 2.1.3.2. Các thành phần

- Lớp Singleton khai báo phương thức tĩnh getInstance trả về cùng một thể hiện của lớp riêng của nó.
- Phương thức khởi tạo của Singleton nên được ẩn khỏi mã máy khách. Gọi getInstance là phương thức duy nhất để lấy đối tượng Singleton.

### 2.1.4. Khả năng ứng dụng

- Trường hợp giải quyết các bài toán: Shared resource, Logger, Configuration, Caching, Thread pool,...
- Một số design pattern khác cũng sử dụng Singleton để triển khai: Abstract Factory, Builder, Prototype, Facade,...
- Đã được sử dụng trong một số class của Java Core.

## **2.1.5. Hệ quả**

### **2.1.5.1. Ưu điểm**

- Có thể chắc chắn rằng một lớp chỉ có một instance.
- Có khả năng truy cập đến instance từ mọi nơi (global access).
- Đối tượng singleton chỉ được khởi tạo duy nhất một lần khi nó được yêu cầu lần đầu.
- Kiểm soát truy cập đến instance duy nhất.
- Giảm số lượng namespace.

### **2.1.5.2. Nhược điểm**

- Vi phạm Single Responsibility Principle. Mẫu này giải quyết hay vấn đề trên cùng một thời điểm.
- Singleton pattern có thể thay đổi thiết kế kém, ví dụ khi các thành phần của chương trình biết quá nhiều về nhau.
- Có thể sinh ra khó khăn trong việc unit test client code của Singleton bởi nhiều test frameworks dựa vào kế thừa khi sản sinh mock objects.

## **2.1.6. Các mẫu liên quan**

- Abstract Factory: thường là Singleton để trả về các đối tượng factory duy nhất.
- Builder: dùng để xây dựng một đối tượng phức tạp, trong đó Singleton được dùng để tạo một đối tượng truy cập tổng quát (Director).
- Prototype: dùng để sao chép một đối tượng, hoặc tạo ra một đối tượng khác từ Prototype của nó, trong đó Singleton được dùng để chắc chắn chỉ có một Prototype.

## **2.2. Factory Method**

### **2.2.1. Tổng quan**

Factory Method định nghĩa một giao diện (interface) cho việc tạo một đối tượng, nhưng để các lớp con quyết định lớp nào sẽ được tạo. Factory Method giao việc khởi tạo một đối tượng cụ thể cho lớp con. Mục đích của Factory Method là:

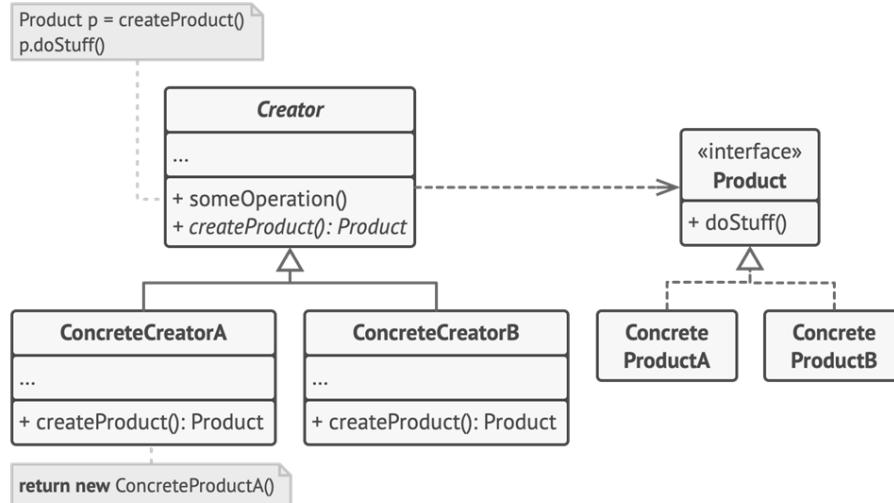
- Tạo ra một cách khởi tạo object mới thông qua một interface chung
- Che giấu quá trình xử lý logic của phương thức khởi tạo
- Giảm sự phụ thuộc, dễ dàng mở rộng.
- Giảm khả năng gây lỗi.

### **2.2.2. Motivation**

- Tạo một đối tượng mà không cần thiết chỉ ra một cách chính xác lớp nào sẽ được tạo.

### 2.2.3. Đặc điểm

#### 2.2.3.1. Cấu trúc mẫu



Hình 2.4: Sơ đồ lớp của mẫu thiết kế Factory Method

#### 2.2.3.2. Thành phần trong cấu trúc mẫu

- Thành phần:
  - Product: Là giao diện, là thứ dùng chung cho các đối tượng được tạo ra bởi Creator và các lớp dẫn xuất của nó.
  - ConcreteProduct: Là các lớp hiện thực của giao diện Product.
  - Creator: Là lớp trừu tượng, trong đó có phương thức xuất xưởng thứ sẽ trả về kiểu dữ liệu trùng với giao diện của đối tượng cần tạo (interface Product). Đây là phương thức trừu tượng để đảm bảo các lớp dẫn xuất phải triển khai.
  - ConcreteCreator: Là các lớp dẫn xuất của lớp trừu tượng Creator, đã triển khai phương thức trừu tượng để nó trả về một loại Product.
- Sự cộng tác: Clients tùy thuộc vào business logic sẽ tạo ra đối tượng ConcreteCreatorA hoặc ConcreteCreatorB. Lúc này đối tượng sẽ đảm nhiệm việc tạo ra đối tượng Product bằng phương thức createProduct. Vì createProduct là phương thức trừu tượng nên nó sẽ thực hiện việc tạo ra Product đúng với định nghĩa trong các phương thức đã triển khai ở các lớp dẫn xuất ConcreteCreator.

### 2.2.4. Khả năng ứng dụng

- "Factory method" thường được dùng trong bộ phát triển (toolkit) hay framework, ở đó, đoạn mã của framework cần thiết phải tạo một đối tượng là những lớp con của ứng dụng sử dụng framework đó. Client tạo ra được đối tượng mới mà không cần quan tâm đến cách nó được tạo ra.

### 2.2.5. Hệ quả

#### 2.2.5.1. Ưu điểm

- Nó khuyến khích việc liên kết lỏng lẻo (loose-coupling) bằng cách loại bỏ nhu cầu ràng buộc. Tránh sự liên kết giữa creator với concrete products.
- Đảm bảo các nguyên lý của SOLID:
  - Single Responsibility Principle: Có thể đưa code tạo sản phẩm sang một nơi khác để dễ sử dụng hơn. Nó sẽ tách mã xây dựng sản phẩm khỏi mã thực sự sử dụng sản phẩm.
  - Open/Closed Principle: Dễ dàng mở rộng, thêm những đoạn code mới vào chương trình mà không cần tác động đến các đối tượng ban đầu.

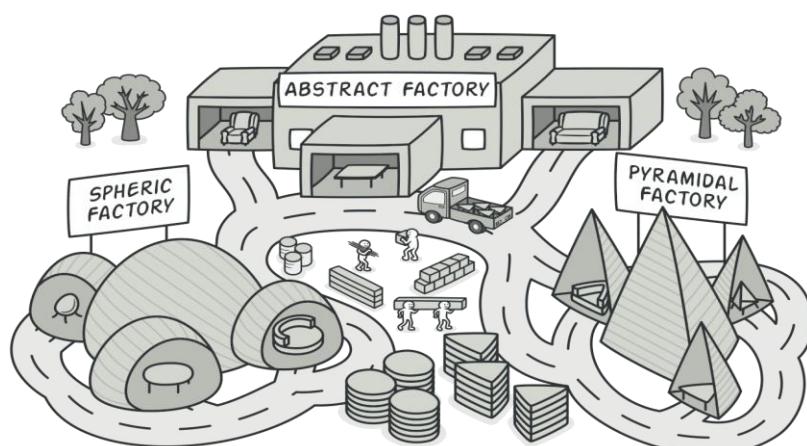
#### 2.2.5.2. Nhược điểm

- Code trở nên phức tạp hơn khi phải sử dụng nhiều class mới để triển khai mẫu.

### 2.2.6. Các mẫu thiết kế liên quan

- Nhiều mẫu 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 thành Abstract Factory, Prototype hoặc Builder (linh hoạt đơn nhưng phức tạp hơn).
- Factory Method là mẫu đặc biệt của Template Method. Ngoài ra thì Factory Method có thể đóng vai trò là một bước trong Template Method lớn.

## 2.3. Abstract Factory



Hình 2.5: Minh họa Abstract Factory

### 2.3.1. Tổng quan

Abstract Factory là một design pattern thuộc nhóm Creational Pattern Design – những mẫu thiết kế cho việc khởi tạo đối tượng của lớp. Abstract Factory được xây dựng dựa trên Factory Pattern và nó được xem là một factory cao nhất trong hệ thống phân cấp. Pattern này sẽ tạo ra các factory là class con của nó và các factory này được tạo ra giống như cách mà factory tạo ra các sub-class. (Như một nhà máy lớn với bên trong là các nhà máy nhỏ hơn sản xuất các sản phẩm, linh kiện có liên quan đến nhau).

Mục đích của Abstract Factory là:

- Cung cấp một interface cho việc khởi tạo các tập hợp của những object có đặc điểm giống nhau mà không cần quan tâm lớp cụ thể của chúng.
- Tạo ra một hệ thống phân cấp được đóng gói.
- Sử dụng các phương thức khởi tạo thay vì từ khóa new ở phía Client.

### 2.3.2. Motivation

Giả sử bạn đang kinh doanh 1 cửa hàng nội thất với các dòng sản phẩm chính là Ghế, Sofa và Bàn cà phê. Trong mỗi dòng sản phẩm lại có các biến thể như Hiện đại, Victoria, Art deco.



Hình 2.6: Các biến thể với các loại sản phẩm

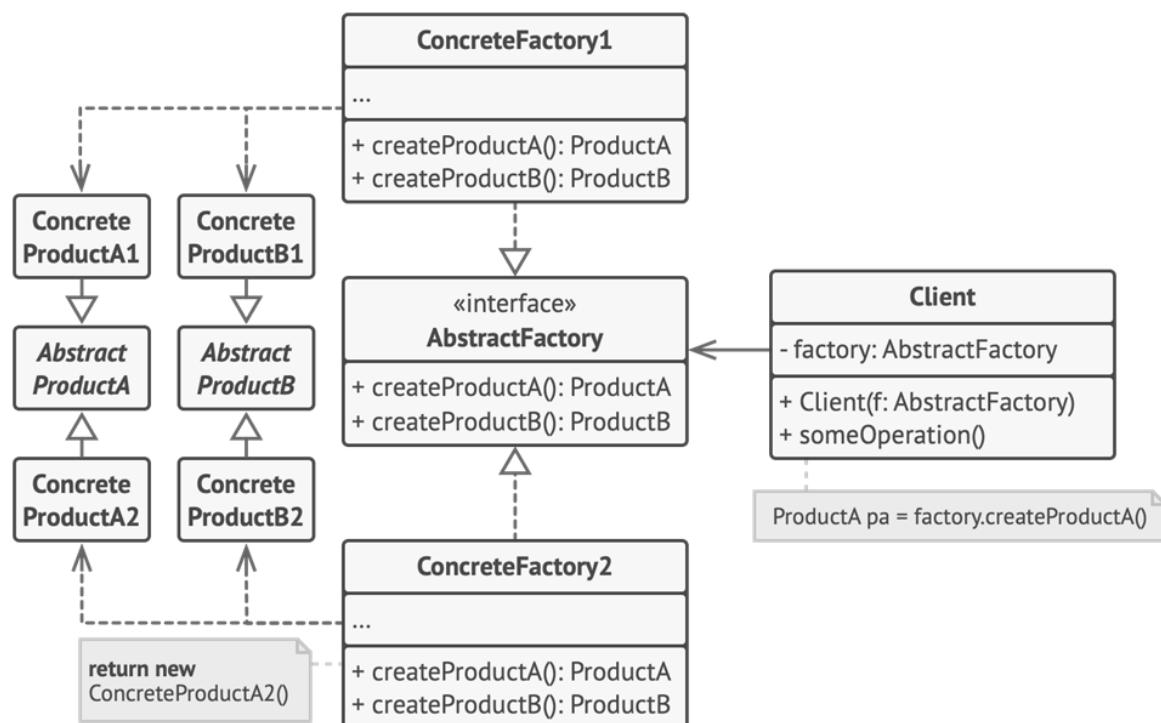
Khách hàng đến và yêu cầu bạn giao cho họ một bộ sản phẩm. Tuy nhiên, các sản phẩm bạn giao đến lại không phù hợp với nhau. Ví dụ một chiếc ghế sofa hiện đại đi kèm với một chiếc ghế Victoria.

Cần có một cách tạo ra mỗi object riêng lẻ sao cho chúng phù hợp với các object khác trong cùng một biến thể với nhau? Đồng thời xử lý làm sao để quá trình sau đó bạn muốn nhập thêm các dòng sản phẩm mới, các biến thể mới mà không phải thay đổi code?

- Abstract Factory Pattern giúp đảm bảo rằng các product mà bạn nhận được từ một factory đều tương thích với nhau.
- Abstract Factory Pattern giúp gom các đoạn code tạo ra product vào một nơi trong chương trình, nhờ đó giúp dễ theo dõi và thao tác.
- Với Abstract Factory Pattern, chúng ta có thể thoải mái thêm nhiều loại product mới vào chương trình mà không làm thay đổi các đoạn code nền tảng đã có trước đó.

### 2.3.3. Đặc điểm

#### 2.3.3.1. Cấu trúc mẫu



Hình 2.7: Sơ đồ lớp của mẫu thiết kế Abstract Factory

#### 2.3.3.2. Thành phần trong cấu trúc mẫu

- Abstract Product: Interface hoặc abstract class định nghĩa các phương thức tạo ra một tập hợp các sản phẩm có liên quan đến nhau (Chair, Sofa, CoffeeTable)

- Concrete Product: Là các cách triển khai khác nhau của Abstract Product gồm nhiều các biến thể. Mỗi nhóm Abstract Product phải được thực hiện cùng các biến thể nhất định. (ModernChair, VictorianSofa, ...)
- Abstract Factory: Interface hoặc abstract class với các phương thức tạo ra các đối tượng Abstract Product
- Concrete Factory: Xây dựng, cài đặt, thực hiện các phương thức được tạo nên từ Abstract Factory. Mỗi Concrete Factory tương ứng với một biến thể khác của product và chỉ tạo ra những biến thể của sản phẩm đó.
- Client: Đối tượng sử dụng Abstract Factory, Abstract Product và client cũng có thể làm việc với bất kỳ biến thể nào của Concrete Factory, Concrete Product, miễn là nó có giao tiếp với đối tượng (object) thông qua abstract interface

#### **2.3.4. Khả năng ứng dụng**

- Sử dụng Abstract Factory khi cần làm việc với các product có tính chất gần giống nhau và liên quan đến nhau mà không muốn phụ thuộc vào các lớp cụ thể của những sản phẩm đó:
  - Phía client sẽ không phụ thuộc vào việc những sản phẩm được tạo ra như thế nào.
  - Ứng dụng sẽ được cấu hình với một hoặc nhiều họ sản phẩm.
  - Các đối tượng cần phải được tạo ra như một tập hợp để có thể tương thích với nhau.

#### **2.3.5. Hết quả**

##### **2.3.5.1. Ưu điểm**

- Có thể đảm bảo các đối tượng product nhận được từ factory sẽ tương thích với nhau.
- Tránh được những ràng buộc chặt chẽ giữa concrete products và client code.
- Nguyên tắc đơn lẻ (Single Responsibility Principle): Có thể trích xuất code tạo product vào một nơi và hỗ trợ code dễ dàng.
- Nguyên tắc mở/ đóng (Open/Closed Principle): có thể khởi tạo những biến thể mới của product mà không cần phá vỡ client code hiện có.

##### **2.3.5.2. Nhược điểm**

- Code có thể trở nên phức tạp hơn mức bình thường, vì có rất nhiều interfaces và classes được khởi tạo cùng với pattern.

### **2.3.6. Các mẫu thiết kế liên quan**

- Abstract Factory thường dựa trên một tập hợp các Factory Method, nhưng cũng có thể sử dụng Prototype để kết hợp các phương thức trên các lớp này.
- Builder tập trung vào việc xây dựng các đối tượng phức tạp theo từng bước. Còn Abstract Factory chuyên tạo các tập hợp các đối tượng liên quan đến nhau. Abstract Factory trả lại product ngay lập tức, trong khi Builder cho phép bạn chạy một số bước xây dựng bổ sung trước khi trả về product ở bước cuối cùng.
- Abstract Factory, Builder và Prototype đều có thể được triển khai dưới dạng Singleton.

## **2.4. Builder**

### **2.4.1. Tổng quan**

Builder là một trong những design pattern thuộc nhóm Creational Pattern và được tạo ra để xây dựng một đối tượng phức tạp bằng cách sử dụng các đối tượng đơn giản và sử dụng tiếp cận từng bước, việc xây dựng các đối tượng độc lập với các đối tượng khác. Mục đích:

- Tách việc xây dựng đối tượng phức tạp khỏi thể hiện của nó. Lúc này chung một quy trình xây dựng đối tượng có thể tạo ra các thể hiện khác nhau.
- Tách việc xây dựng đối tượng khỏi phần code logic của ứng dụng.
- Việc xây dựng các đối tượng độc lập với các đối tượng khác

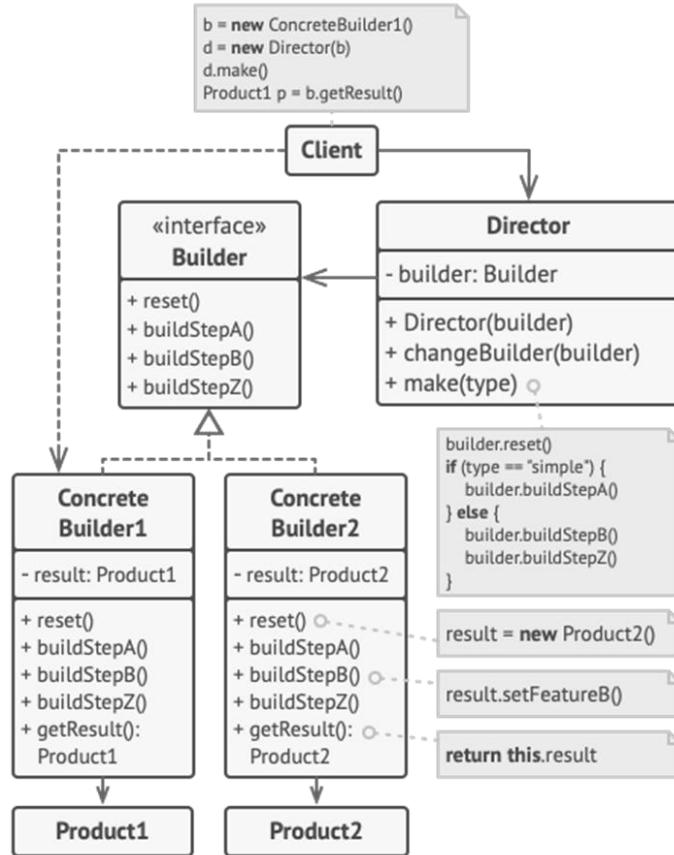
### **2.4.2. Motivation**

Builder được xây dựng để khắc phục một số nhược điểm của Factory Method, Abstract Factory:

- Quá nhiều tham số đầu vào truyền từ client.
- Tham số truyền vào có thể có tùy chọn, nếu không có sẽ mặc định,
- Object quá nhiều thuộc tính, tạo một lúc sẽ gây ra phức tạp.

### **2.4.3. Đặc điểm**

#### **2.4.3.1. Cấu trúc mẫu**



Hình 2.8: Sơ đồ lớp của mẫu thiết kế Builder

#### 2.4.3.2. Thành phần trong cấu trúc mẫu

- Product (Đại diện cho đối tượng phức tạp, nhiều thuộc tính)
- Builder (Abstract class/Interface khai báo phương thức tạo đối tượng)
- ConcreteBuilder (Hiện thực của lớp Builder, nắm giữ các instance mà nó tạo ra, đồng thời cung cấp các phương thức trả về các instance đó)
- Director (là nơi sẽ gọi tới Builder để tạo ra đối tượng, nắm giữ các các thức để tạo ra đối tượng)
- Sự cộng tác:
  - Client tạo ra Director object.
  - Director sẽ thông báo cho builder khi nào một thuộc tính của Product sẽ thực hiện.
  - Builder nhận yêu cầu và thêm thuộc tính đó vào Product.
  - Client sẽ lấy Product từ Builder

#### 2.4.4. Khả năng ứng dụng

- Tạo một đối tượng phức tạp
- Khi nạp chòng quá nhiều hàm constructor, bạn nên nghĩ đến Builder.
- Muốn kiểm soát quá trình xây dựng (thứ tự trước sau,...)

- Khi Client mong đợi nhiều cách khác nhau cho đối tượng được xây dựng.
- Lợi ích to lớn là giúp việc xây dựng đối tượng được tường minh → dễ hiểu, dễ bảo trì ứng dụng

#### **2.4.5. Hệ quả**

##### **2.4.5.1. Ưu điểm**

- Giảm bớt số lượng constructor.
- Code tường minh, dễ đọc, dễ bảo trì hơn.
- Đối tượng xây dựng sẽ an toàn hơn.
- Kiểm soát tốt hơn trong quá trình xây dựng.
- Single responsibility: tách việc xây dựng 1 đối tượng phức tạp khỏi business logic

##### **2.4.5.2. Nhược điểm**

- Bị duplicate code khá nhiều (mỗi thuộc tính từ Product class sẽ có 1 hàm cài đặt trong Builder class).
- Tăng độ phức tạp code (tổng thể) do số lượng class và hàm tăng lên.

#### **2.4.6. Các mẫu thiết kế liên quan**

- Factory Method và Abstract factory: bạn có thể sử dụng mẫu Abstract Factory để tạo một đối tượng nhà máy sản xuất một dòng sản phẩm có liên quan, sau đó sử dụng mẫu Builder để tạo từng bước các sản phẩm đó với các cấu hình khác nhau.
- Composite

### **2.5. Prototype**

#### **2.5.1. Tổng quan**

Prototype là mẫu thiết kế thuộc nhóm Creational Pattern dùng để chỉ định loại của các đối tượng cần tạo bằng cách sử dụng một đối tượng mẫu và tạo các đối tượng mới bằng cách sao chép đối tượng này.

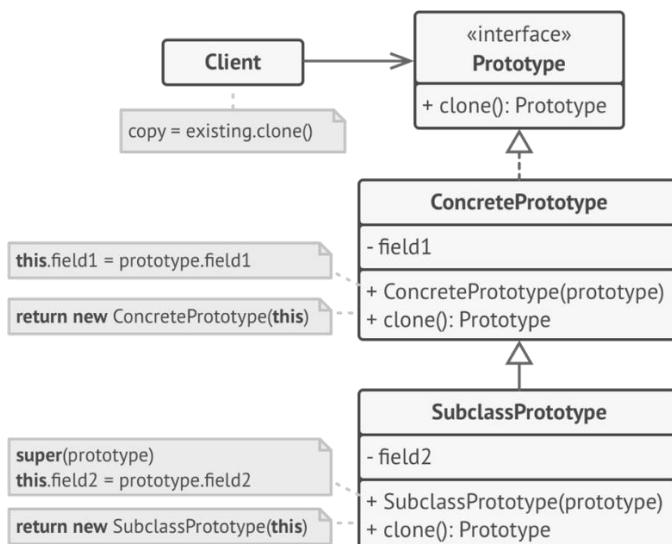
#### **2.5.2. Motivation**

- Khi việc tạo một đối tượng mới tốn nhiều chi phí và thời gian trong khi bạn đã có một đối tượng tương tự đã tồn tại
- Sử dụng Prototype Pattern khi bạn không muốn code của mình phụ thuộc vào các lớp đối tượng cụ thể mà bạn muốn sao chép
- Khi bạn muốn giảm số lượng các lớp con mà các lớp áy chỉ khác nhau về cách khởi tạo các đối tượng tương ứng.

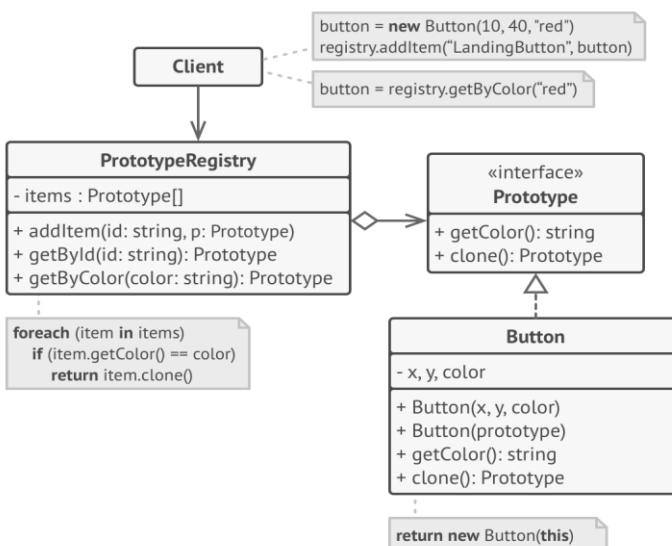
- Prototype không được sử dụng phổ biến trong việc xây dựng các ứng dụng nghiệp vụ (business application) mà thường được sử dụng trong các kiểu ứng dụng xác định như đồ họa máy tính, CAD (Computer Assisted Drawing), GIS (Geographic Information Systems) và các trò chơi.
- Trong .NET định nghĩa sẵn một interface có tên là ICloneable có phương thức Clone trả về một đối tượng là một bản sao của đối tượng gốc. Khi sử dụng nó, bạn cần chú ý đến việc deep copy và shallow copy.
- Nhân bản item/quái trong game.

### 2.5.3. Đặc điểm

#### 2.5.3.1. Cấu trúc mẫu



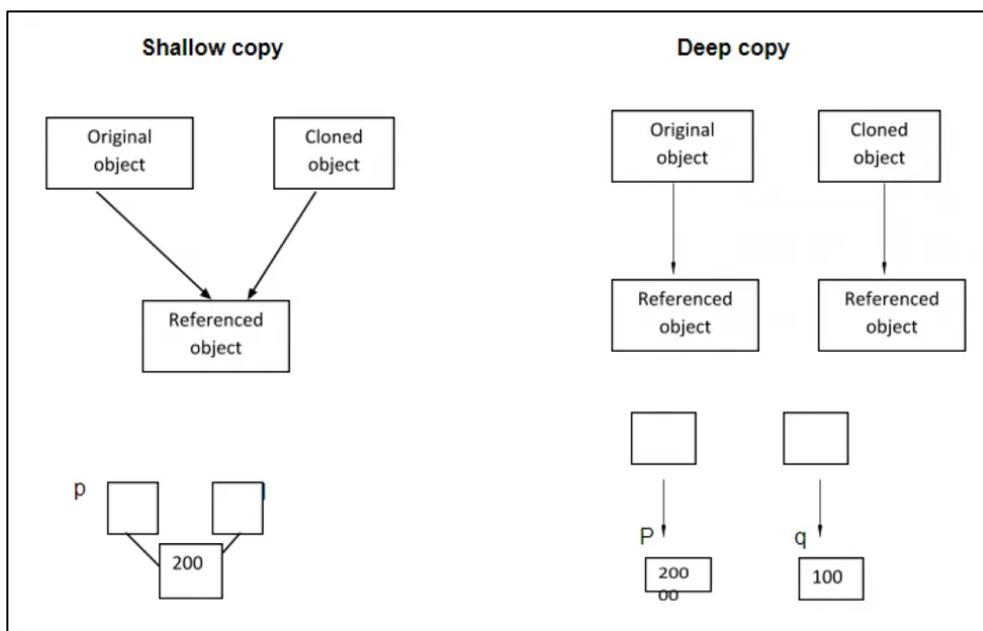
Hình 2.9: Sơ đồ lớp của mẫu thiết kế Prototype dạng cơ bản



Hình 2.10 Sơ đồ lớp của mẫu thiết kế Prototype dạng Prototype Registry

### 2.5.3.2. Thành phần trong cấu trúc mẫu

- Prototype interface: Định nghĩa các phương thức clone
- Lớp ConcretePrototype: Hiện thực phương thức trên
- Client: Tạo ra một đối tượng mới bằng việc gọi phương thức clone để nhân bản chính nó.
- Sự cộng tác: Cung cấp một cách dễ dàng để truy cập các prototype được sử dụng thường xuyên. Nó lưu trữ một tập hợp các đối tượng đựng sẵn đã sẵn sàng để sao chép.
- Để cài đặt Prototype Pattern, tạo ra một Clone Method ở lớp cha, và triển khai ở các lớp con. Có thể triển khai bản sao nông và bản sao sâu:
  - Shallow copy: Các đối tượng con bên trong chỉ được copy reference. Nghĩa là chỉ nhân bản được value type. (Đối tượng ban đầu và đối tượng tạo mới đều trỏ tới chung 1 đối tượng con bên trong) (Chỉ các đối tượng cấp cao nhất được sao chép và các cấp độ thấp hơn chứa các tham chiếu)
  - Deep copy: Các đối tượng con bên trong cũng được copy lại toàn bộ các thuộc tính. Nghĩa là nhân bản được value type và reference type. (Tất cả các đối tượng được sao chép)



Hình 2.11: Cơ chế của Shallow Copy và Deep Copy

Có một đối tượng là X liên quan đến đối tượng A và B. Tạo X2 là một Shallow copy của X thì X2 vẫn còn liên quan đến đối tượng A và B. Còn nếu X2 là một Deep copy của X thì X2 liên quan đến đối tượng A2 và B2 là những bản sao của A và B.

#### **2.5.4. Khả năng ứng dụng**

- Sử dụng Prototype Pattern khi bạn không muốn code của mình phụ thuộc vào các lớp đối tượng cụ thể mà bạn muốn sao chép.
- Khi bạn muốn giảm số lượng các lớp con mà các lớp ấy chỉ khác nhau về cách khởi tạo các đối tượng tương ứng.

#### **2.5.5. Hệ quả**

##### **2.5.5.1. Ưu điểm**

- Giảm chi phí đáng kể so với tạo lập 1 đối tượng theo phương thức chuẩn, gọi toán tử new (gọi hàm tạo lập cụ thể)
- Đối tượng được tạo ra nhờ việc sao chép không cần phụ thuộc vào lớp của đối tượng gốc
- Che giấu sự phức tạp của việc tạo các instance mới ở phía client
- Thêm và loại bỏ lớp concrete trong lúc thực thi: Prototype hợp nhất quá trình tạo một đối tượng mới vào hệ thống đơn giản chỉ bằng cách đăng ký một thực thể nguyên mẫu với client. Điều đó linh động hơn chút so với các mẫu kiến tạo khác bởi vì client có thể cài đặt và loại bỏ các nguyên mẫu tại thời điểm chạy chương trình.
- Cho phép khởi tạo đối tượng mới bằng cách thay đổi một vài attribute của đối tượng (các đối tượng này có ít điểm khác biệt nhau): Ta có thể tùy chỉnh các thuộc tính đối tượng mẫu để tạo ra những đối tượng mẫu mới. Hay có thể hiểu là ta tạo ra một “chuẩn” class mới từ class có sẵn mà không cần viết code để định nghĩa.
- Tạo đối tượng mới bằng cách thay đổi cấu trúc: Rất nhiều ứng dụng xây dựng hệ thống từ nhiều phần và các phần con. Các phần con lại khởi tạo từ nhiều phần con khác. Prototype cũng hỗ trợ điều này. Nghĩa là các phần đó có thể được khởi tạo từ việc copy một nguyên mẫu từ một “cấu trúc” khác. Miễn là các phần kết hợp đều thể hiện Clone() và được sử dụng với cấu trúc khác nhau làm nguyên mẫu.
- Giảm việc phân lớp: Đôi khi hệ thống quá phức tạp vì có quá nhiều class, và cây thừa kế của lớp khởi tạo có quá nhiều lớp song song cùng mức. Prototype pattern rõ ràng làm giảm số lớp và sự phức tạp của cây thừa kế (class hierarchy). Một điểm hạn chế của prototype pattern là đôi khi việc phải implement mỗi phương thức Clone() của mỗi lớp con của Prototype sẽ gặp khó khăn. Ví dụ như việc

implement Clone() sẽ khó khăn khi mà những đối tượng nội hàm của chúng không hỗ trợ việc copy hoặc có một reference không copy được reference.

#### **2.5.5.2. Nhược điểm**

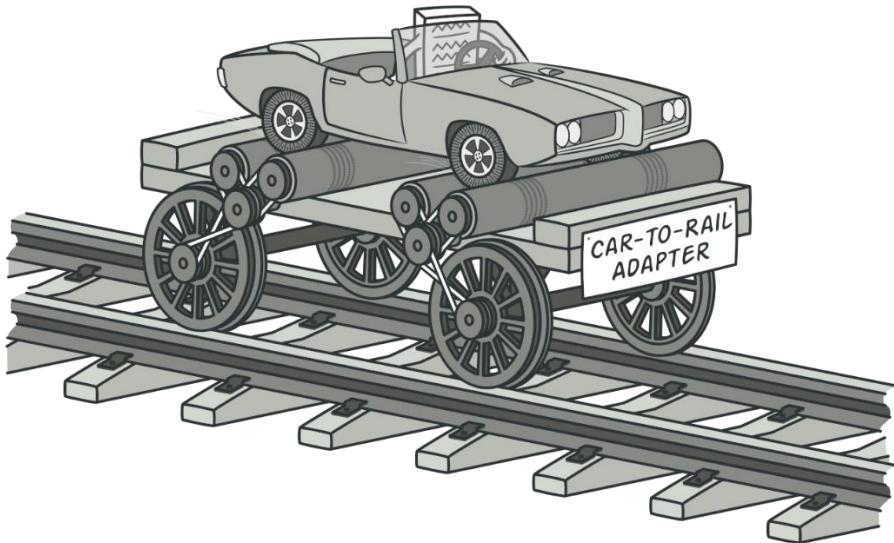
- Việc nhân bản đối tượng phức tạp có phụ thuộc vòng có thể rất khó.

#### **2.5.6. Các mẫu thiết kế liên quan**

- Abstract Factory
- Factory Method
- Builder
- Command
- Composite
- Decorator
- Memento
- Singleton

## Chương 3. STRUCTURAL PATTERN

### 3.1. Adapter



Hình 3.1 Minh họa mẫu thiết kế Adapter

#### 3.1.1. Tổng quan

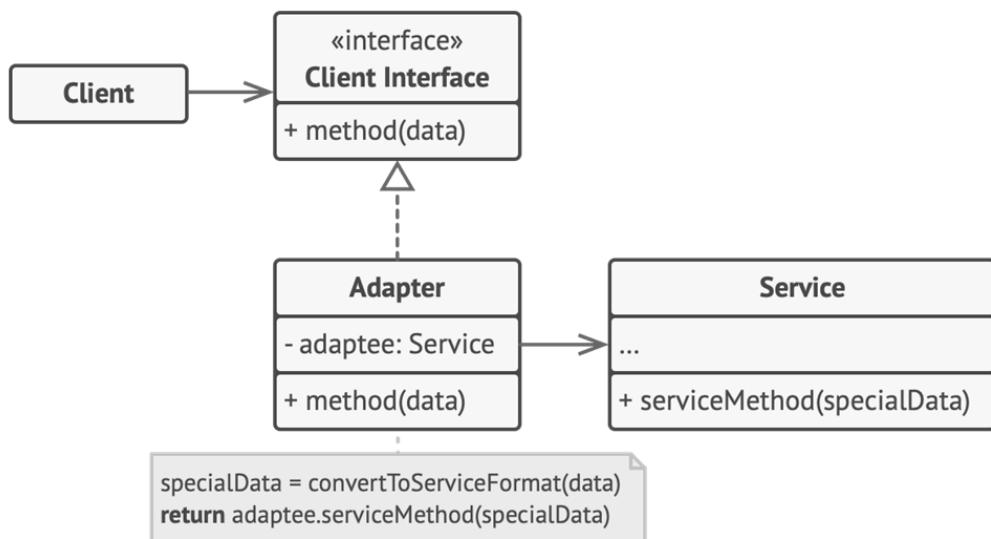
- Adapter là mẫu thiết kế chuyên đổi interface của một lớp thành một interface mà phía clients muốn → Cho phép 2 interface không liên quan làm việc cùng nhau
- Khi thiết kế hệ thống mới, đôi khi chúng ta cần tái sử dụng lại các lớp trong hệ thống cũ. Tuy nhiên vấn đề xảy ra khi các class được tái sử dụng không tương thích với các lớp mới do khác interface.
- 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.
- Adapter còn gọi là Wrapper do cung cấp interface “bọc ngoài” tương thích cho hệ thống có sẵn, có dữ liệu và hành vi phù hợp nhưng có interface không tương thích với lớp đang viết. Đối tượng được bọc thậm chí không biết về Adapter.
- Mục đích:
  - Chuyển đổi interface của một lớp thành interface khác → Adapter cho phép các lớp cộng tác với nhau - cái mà vốn dĩ chúng không làm được do khác interface.
  - Bao bọc lớp bằng một interface mới.

### 3.1.2. Motivation

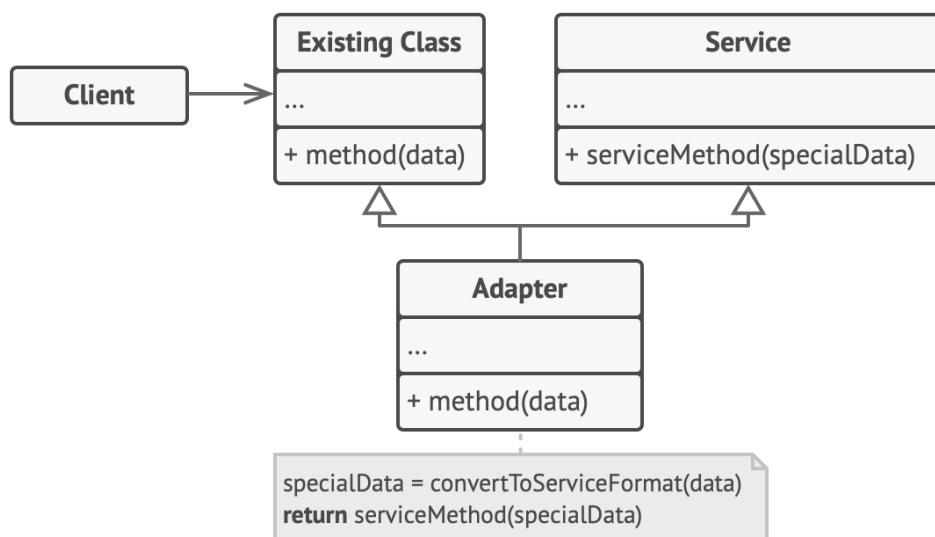
- Muốn sử dụng một số class có sẵn nhưng interface của nó không tương thích với code hiện tại
- Muốn sử dụng lại một số subclass hiện có thiếu một số chức năng và không thể thêm vào lớp cha.
- Adapter thường được sử dụng trong môi trường lập trình nơi các thành phần mới hoặc ứng dụng mới cần được tích hợp và hoạt động cùng với các thành phần hiện có.

### 3.1.3. Đặc điểm

#### 3.1.3.1. Cấu trúc mẫu



Hình 3.2: Sơ đồ lớp của mẫu thiết kế Adapter dạng đổi tượng



Hình 3.3: Sơ đồ lớp của mẫu thiết kế Adapter dạng lớp

### **3.1.3.2. Thành phần trong cấu trúc mẫu**

- Client bao gồm business logic hiện tại
- Client Interface thì định nghĩa những protocols hoặc interface nào ta cần theo nếu muốn làm việc với Client
- Services thì bao gồm một số class, methods cần thiết (Client không thể làm việc với lớp này trực tiếp do nó có implement interface không tương thích)
- Adapter là lớp làm việc với cả Client Interface và Service. Adapter sẽ được call từ client thông qua Adapter Interface, chuyển đổi nó làm sao cho Service nó hiểu được.
- So sánh Object Adapter với Class Adapter:
  - Sự khác biệt chính là Class Adapter sử dụng Inheritance (ké thừa) để kết nối Adapter và Adaptee trong khi Object Adapter sử dụng Composition (chứa trong) để kết nối Adapter và Adaptee.
  - Trong cách tiếp cận Class Adapter, nếu một Adaptee là một class và không phải là một interface thì Adapter sẽ là một lớp con của Adaptee. Do đó, nó sẽ không phục vụ tất cả các lớp con khác theo cùng một cách vì Adapter là một lớp phụ cụ thể của Adaptee.
  - Object Adapter sẽ tốt hơn vì nó sử dụng Composition để giữ một thể hiện của Adaptee, cho phép một Adapter hoạt động với nhiều Adaptee nếu cần thiết.

### **3.1.4. Khả năng ứng dụng**

- Adapter thường được sử dụng trong môi trường lập trình nơi các thành phần mới hoặc ứng dụng mới cần được tích hợp và hoạt động cùng với các thành phần hiện có.

### **3.1.5. Hệ quả**

#### **3.1.5.1. Ưu điểm**

- Single Responsibility Principle: Có thể tách interface hoặc các đoạn code chuyên đổi dữ liệu khỏi logic nghiệp vụ chính của chương trình
- Open/Closed Principle: Giúp code không bị ảnh hưởng từ các thay đổi hoặc các lần cập nhật phiên bản mới từ API hoặc dịch vụ từ bên thứ ba (thay đổi tên hàm, tên lớp, ...)
- Tính đóng gói

- Giúp cho việc tái sử dụng, linh hoạt code hơn

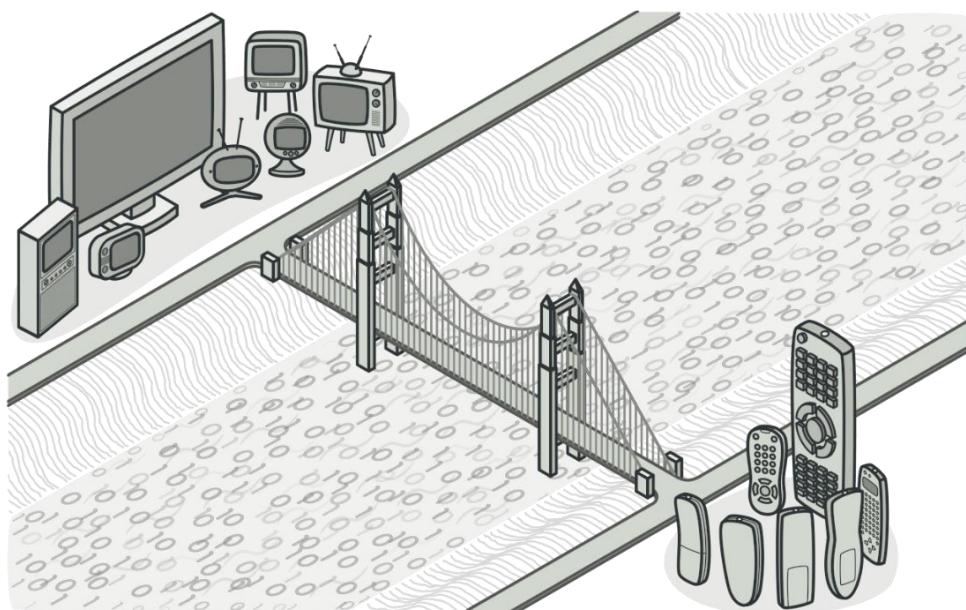
### **3.1.5.2. Nhược điểm**

- Code trở nên phức tạp hơn. Đôi khi, việc thay đổi lớp dịch vụ sao cho phù hợp với phần còn lại của mã của bạn sẽ đơn giản hơn.
- Tăng chi phí và hiệu năng do phải làm việc qua trung gian

### **3.1.6. Các mẫu thiết kế liên quan**

- Bridge
- Decorator
- Proxy
- Facade

## **3.2. Bridge**



*Hình 3.4 Minh họa mẫu thiết kế Bridge*

### **3.2.1. Tổng quan**

Bridge là mẫu thiết kế thuộc Structural Pattern, có tên khác là Handle/Body (Một thuật ngữ với Handle là Abstraction và Body là Implementation), dùng để phân chia logic nghiệp vụ hoặc một class khổng lồ thành các hệ thống phân cấp riêng biệt có thể được phát triển độc lập. Một trong những hệ thống phân cấp này (Abstraction) sẽ nhận tham chiếu đến một đối tượng của hệ thống phân cấp thứ hai (Implementation).

### **3.2.2. Motivation**

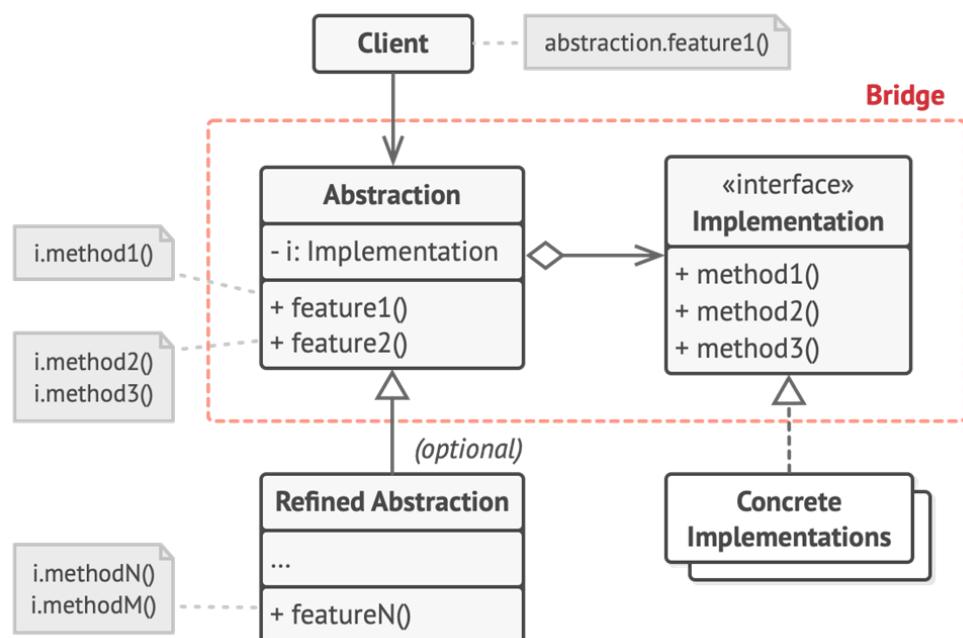
- Tách tính trừu tượng (Abstraction) ra khỏi tính hiện thực (Implementation) và cho phép chúng thay đổi độc lập (Cho phép bạn chia một lớp lớn hoặc một tập hợp các

lớp có liên quan chặt chẽ thành hai hệ thống phân cấp riêng biệt - trừu tượng (abstraction) và hiện thực (implementation) - có thể được phát triển độc lập với nhau. Từ đó có thể dễ dàng chỉnh sửa hoặc thay thế mà không làm ảnh hưởng đến những nơi có sử dụng lớp đó ban đầu).

- Tránh sự ràng buộc vĩnh viễn giữa Abstraction và Implementation. Ví dụ khi triển khai có thể được chọn hoặc chuyển đổi trong lúc chương trình đang chạy.
- Cả Abstraction và Implementation của chúng sẽ được mở rộng độc lập bằng các subclass trong tương lai
- Sử dụng ở những nơi mà những thay đổi được thực hiện trong Implementation không ảnh hưởng đến phía client
- Muốn chia sẻ một Implementation giữa nhiều đối tượng

### 3.2.3. Đặc điểm

#### 3.2.3.1. Cấu trúc mẫu



Hình 3.5: Sơ đồ lớp của mẫu thiết kế Bridge

#### 3.2.3.2. Thành phần trong cấu trúc mẫu

- Abstraction (Image): Định nghĩa giao diện của lớp trừu tượng, quản lý việc tham chiếu đến đối tượng hiện thực cụ thể (Implementation) thông qua field i
- Refined Abstraction (BMPImage, PNGImage): Kế thừa Abstraction, là các biến thể của Abstraction. Giống với lớp cha, chúng cũng làm việc với các implementation thông qua interface Implementation

- Implementation (ImageImp): Định nghĩa interface chung cho các Concrete Implementation. Một Abstraction chỉ có thể giao tiếp với một đối tượng Implementation thông qua các phương thức được khai báo ở đây
- Concrete Implementation (WinImp, LinuxImp): Ké thừa Implementation và định nghĩa chi tiết hàm thực thi
- Client: Liên kết đối tượng Abstraction với một Implementation cụ thể

#### **3.2.4. Khả năng ứng dụng:**

- Tránh sự ràng buộc vĩnh viễn giữa Abstraction và Implementation. Ví dụ khi triển khai có thể được chọn hoặc chuyển đổi trong thời gian chạy.
- Cả Abstraction và Implementation của chúng sẽ được mở rộng độc lập bằng các subclass trong tương lai. Sử dụng ở những nơi mà những thay đổi được thực hiện trong Implementation không ảnh hưởng đến phía client
- Muốn chia sẻ một Implementation giữa nhiều đối tượng.

#### **3.2.5. Hệ quả**

##### **3.2.5.1. Ưu điểm**

- Giảm phụ thuộc giữa Abstraction và Implementation: Tính kế thừa trong OOP thường gắn chặt abstraction và implementation lúc build chương trình. Bridge Pattern có thể được dùng để cắt đứt sự phụ thuộc này và cho phép chúng ta chọn implementation phù hợp lúc chương trình đang chạy.
- Giảm số lượng những lớp con không cần thiết: Một số trường hợp sử dụng tính inheritance sẽ tăng số lượng subclass rất nhiều.
- Code sẽ gọn gàng hơn và kích thước ứng dụng sẽ nhỏ hơn.
- Dễ bảo trì hơn: các Abstraction và Implementation của nó sẽ dễ dàng thay đổi lúc chương trình đang chạy cũng như khi cần thay đổi thêm bớt trong tương lai.
- Dễ dàng mở rộng về sau, giúp việc phát triển phần mềm về lâu dài được linh hoạt và dễ dàng hơn.
- Cho phép ẩn các chi tiết implement từ client: do abstraction và implementation hoàn toàn độc lập nên chúng ta có thể thay đổi một thành phần mà không ảnh hưởng đến phía Client.

##### **3.2.5.2. Nhược điểm**

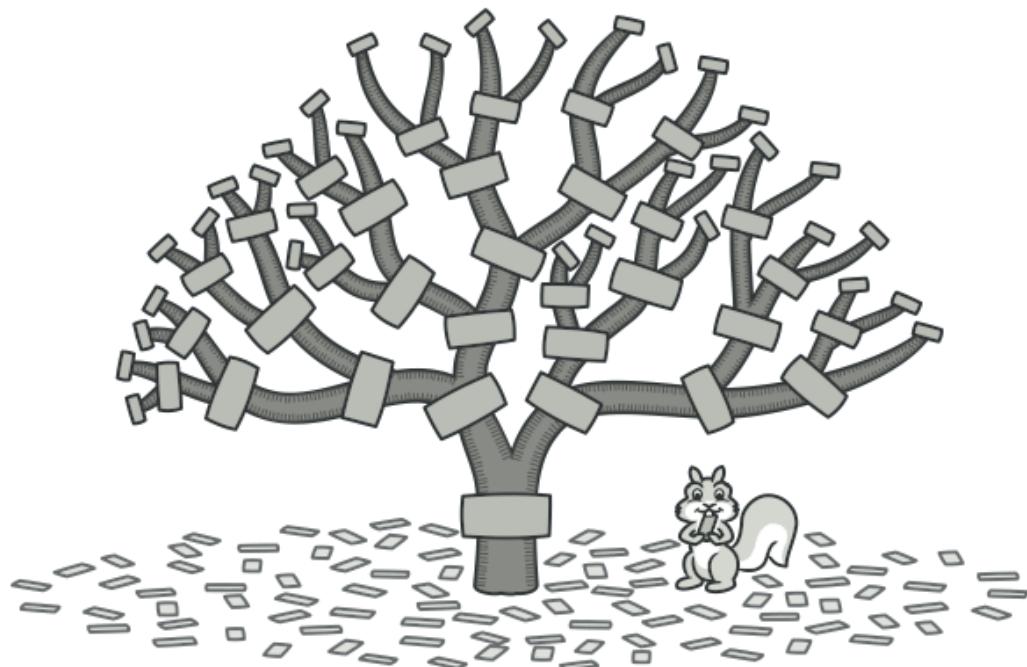
- Có thể làm tăng độ phức tạp khi áp dụng cho một lớp có tính gắn kết cao
- Ủy quyền cho Implementation có thể giảm hiệu năng

### 3.2.6. Các mẫu thiết kế liên quan

- State
- Strategy
- Abstract Factory
- Builder
- Adapter

	<b>Adapter</b>	<b>Bridge</b>
Nhờ vào một lớp khác để thực hiện một số xử lý bất kỳ.		
Mục đích sử dụng	Được dùng để biến đổi một class/interface sang một dạng khác có thể sử dụng được, giúp các lớp không tương thích hoạt động cùng nhau.	Được sử dụng để tách thành phần trừu tượng (abstraction) và thành phần thực thi (implementation) riêng biệt.
Thời điểm ứng dụng	Làm cho mọi thứ có thể hoạt động với nhau sau khi chúng đã được thiết kế (đã tồn tại)	Được thiết kế trước khi phát triển hệ thống để Abstraction và Implementation có thể thực hiện một cách độc lập.

### 3.3. Composite



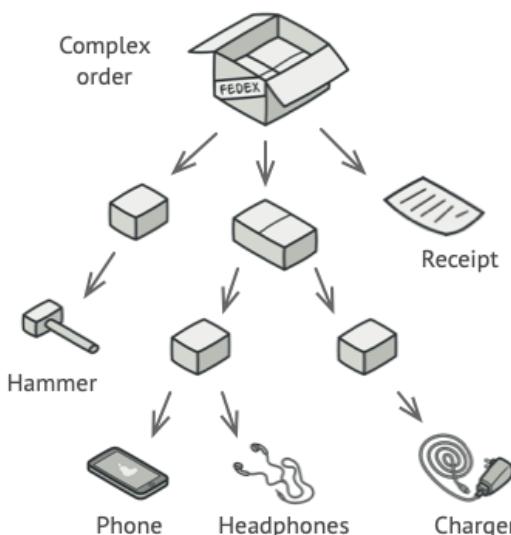
Hình 3.6 Minh họa mẫu thiết kế Composite

### 3.3.1. Tổng quan

Composite là mẫu thiết kế thuộc nhóm Structural Pattern, có tên gọi khác là Object Tree, được tạo ra để gom nhóm các object theo cấu trúc cây để mô tả quan hệ phân cấp một.

- Mục đích:
- Biểu diễn các hệ thống phân cấp đối tượng phức tạp theo cách dễ làm việc và thao tác bằng cách sử dụng mẫu chung cho cả đối tượng riêng lẻ và tập hợp các đối tượng. Cho phép Client có thể viết code giống nhau để tương tác với composite object này, bất kể đó là một đối tượng riêng lẻ hay tập hợp các đối tượng.
- Tăng tính tái sử dụng mã bằng cách cho phép tạo ra đối tượng mới thông qua việc kết hợp các đối tượng cũ. Việc này tiết kiệm thời gian và công sức khi phát triển tính năng mới và thay đổi code.
- Cung cấp lớp trừu tượng tách phần client code với các chi tiết của đối tượng phân cấp. Nó khiến cho code trở nên module và dễ bảo trì, thay đổi cấu trúc phân cấp mà không làm thay đổi client code phần - toàn phần.

### 3.3.2. Motivation



Hình 3.7: Ví dụ minh họa về Composite

Một trong những công việc quan trọng nhất của lập trình viên là biểu diễn thế giới thực trong môi trường số.

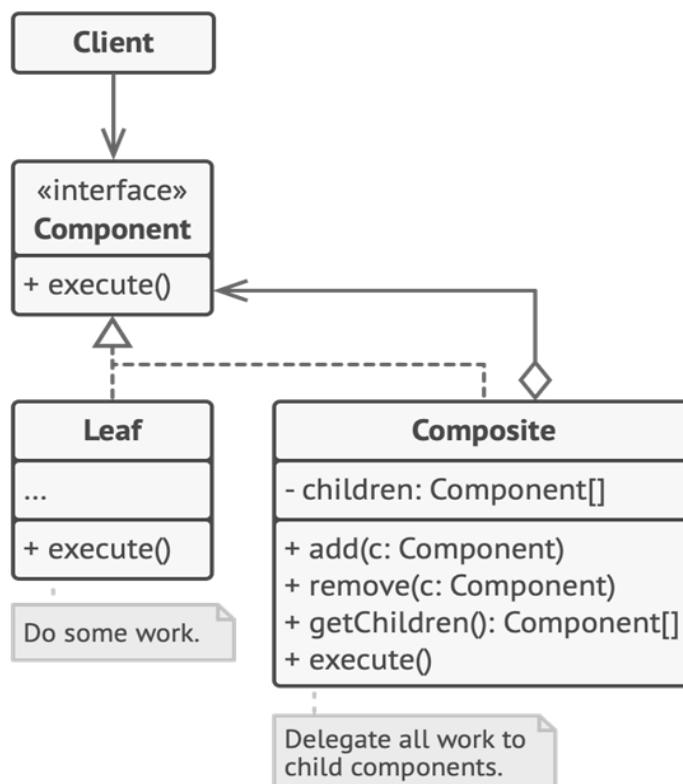
Thế nhưng làm sao biểu diễn hộp giao hàng, chứa nhiều hộp bên trong nó. Làm sao để tính giá tiền của object phức tạp như thế này, trong khi hộp top-level không biết trực tiếp nó chứa gì?

Cách xử lý thông thường: lưu vào 1 mảng nhưng không đảm bảo được quan hệ cha con, cấu trúc có thể bị thay đổi.

- Việc thể hiện hệ thống phân cấp đối tượng nảy sinh ra vấn đề làm thế nào để dễ dàng thao tác và làm việc. Mẫu Composite giải quyết vấn đề bằng các cung cấp mẫu chung cho cả đối tượng riêng lẻ và tập hợp các đối tượng. Việc này cho phép xử lý cả 2 loại đối tượng theo cách thông nhất mà không cần phải biết các chi tiết cụ thể của cấu trúc phân cấp.
- Tạo ra đối tượng mới thông qua việc kết hợp các đối tượng cũ.
- Chia tách client code với chi tiết của đối tượng phân cấp.

### 3.3.3. Đặc điểm

#### 3.3.3.1. Cấu trúc mẫu



Hình 3.8: Sơ đồ lớp của mẫu thiết kế Composite

#### 3.3.3.2. Thành phần trong cấu trúc mẫu

- Component: Interface chung cho các thành phần của Composite. Component cũng cài đặt các hàm mặc định cho các lớp con, đặc biệt là Leaf. Interface của Component khai báo các hành vi của thành phần, và các hàm truy cập và quản lý các thành phần con. Composite (Abstract class/Interface khai báo phương thức tạo đối tượng)

- Leaf: Lớp cụ thể cài đặt Interface của Component. Một trường hợp đặc biệt không thể có thành phần con. Leaf có thể được coi là lớp đơn vị. Leaf không cài đặt các hàm quản lý thành phần con
- Composite: Lớp cụ thể cài đặt Interface của Component. Trường hợp có thể có thành phần con.
- Sự cộng tác:
  - Client sử dụng Composite thông qua Component
  - Nếu yêu cầu Client đến tay Composite, thường Composite sẽ chuyển tiếp yêu cầu tới các thành phần con của nó.
  - Nếu yêu cầu Client đến tay Leaf, yêu cầu sẽ xử lý trực tiếp bởi Leaf.
  - Lớp Composite có thể làm một chút xử lý trước khi / sau khi gửi yêu cầu cho thành phần con

### **3.3.4. Khả năng ứng dụng**

- Xây dựng các hệ thống có cấu trúc phân cấp.
- Bảo đảm rằng tất cả các đối tượng trong cùng một cây hiện thực cùng một tập hợp các hàm, cho phép chúng được sử dụng theo cùng một cách.

### **3.3.5. Hệ quả**

#### **3.3.5.1. Ưu điểm**

- Đơn giản hóa client: Client không nên và không được biết trong component có gì, là leaf hay composite, ... và ứng xử với nó theo cùng một kiểu. Việc client không quan tâm đến component là gì khiến code đơn giản hơn.
- Tính linh hoạt: Cho phép tạo ra hệ thống phân cấp phức tạp, dễ sửa đổi và mở rộng. Có thể thêm các loại đối tượng hoặc thay đổi cấu trúc của hệ thống phân cấp mà không làm ảnh hưởng đến code còn lại.
- Làm việc với cấu trúc một cách thuận tiện hơn bằng việc sử dụng tính đa hình và đệ quy
- Gom nhóm các thành phần đơn giản thành một Composite, và Composite này lại gom nhóm theo một cách đệ quy.
- Note: Mô hình cây này biểu diễn một số thứ trực quan hơn
- Dễ mở rộng thêm Leaf & Composite, bởi chúng có thể hoạt động trên cơ sở đã có.

- Note: Client không cần thay đổi khi thêm Leaf & Composite, khiến việc thêm chúng dễ dàng.
- Khả năng sử dụng lại: Bạn có thể tạo ra các đối tượng mới bằng cách kết hợp các đối tượng hiện có.
- Tính trừu tượng: Cung cấp lớp trừu tượng để tách client code với các chi tiết của hệ thống phân cấp đối tượng.

### **3.3.5.2. Nhược điểm**

- Hiệu suất: Tốn bộ nhớ, tốc độ xử lý nên cần phải tối ưu và thiết kế cẩn thận. Do mẫu sử dụng cấu trúc cây, dẫn đến việc thực hiện một số lượng lớn lời gọi phương thức và tham chiếu đối tượng.
- Độ phức tạp: Cấu trúc cây lúc nào cũng phức tạp.
- An toàn: Do việc sử dụng cùng interface cho cả đối tượng riêng lẻ và tập hợp đối tượng nên khó đảm bảo an toàn về loại cho một số ngôn ngữ (không có phương thức kiểm tra đối tượng thuộc loại nào).

### **3.3.6. Các mẫu thiết kế liên quan**

- Builder: Tạo ra composite phức tạp
- Iterator: Duyệt cây.
- Visitor: thực hiện thao tác trên toàn bộ cây
- Decorator: đã có bảng so sánh. Có thể phối hợp để mở rộng tính năng của Leaf một cách đơn giản và ít chi phí thiết kế.
- Flyweight: để tiết kiệm bộ nhớ bằng việc quản lý các đối tượng có tính lặp lại của composite, lúc này khi đối tượng đã tồn tại thì chỉ cần sử dụng, không cần tạo mới. (màu sắc của các khối trong cây)

<b>Composite</b>	<b>Decorator</b>
Có (0, n) thành phần con	Có (0, 1) thành phần con
Tổng hợp kết quả yêu cầu của thành phần con	Thêm chức năng cho thành phần con

## **3.4. Decorator**

### **3.4.1. Tổng quan**

Decorator là mẫu thiết kế thuộc nhóm Structural Pattern và được tạo ra để chúng ta gắn thêm hành động cho đối tượng bằng cách thêm đối tượng đó vào một đối tượng bao bọc đặc biệt có chứa hành động.

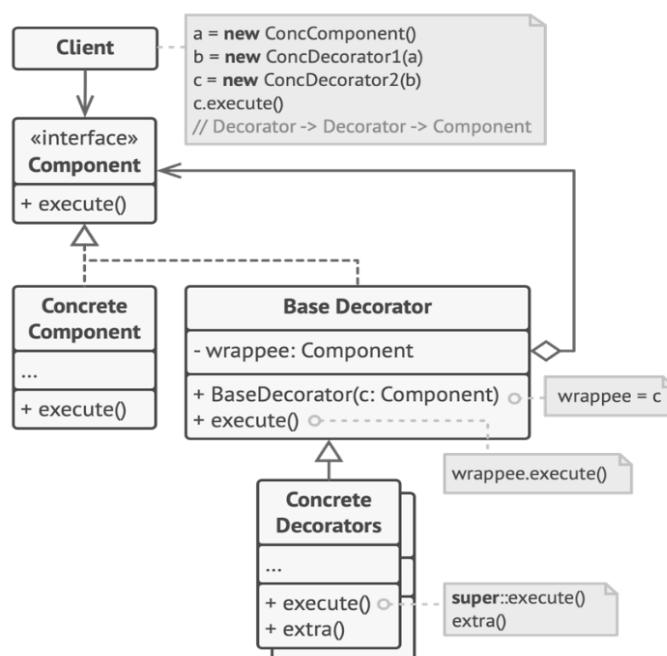
Decorator thường được sử dụng mỗi khi cần thêm tính năng mới, đối tượng hiện có được wrap trong một đối tượng mới (decorator class)

### 3.4.2. Motivation

- Giả sử muôn tạo một li trà sữa. Lúc này ta có các loại nguyên liệu như trà sữa, trân châu trắng, trân châu đen, oreo, kem cheese, v.v với các loại combo như trà sữa trân châu đường đen, trà sữa trân châu oreo, v.v.v...
- Thường thì có thể xây dựng các lớp có hành động như tạo trà sữa, thêm đường...
- Nhưng rồi một ngày có một vị khách order ly siêu đặc biệt, chứa nhiều loại topping khác nhau thì phải làm sao. Trong khi lúc xây dựng đâu có loại trà sữa nào như vậy. Tuy nhiên ta có nguyên liệu đó, chỉ là không biết tạo ra như nào. Lúc này ta cần một cơ chế mới để có thể hướng dẫn tạo ra trà sữa mà có thể thay đổi một cách linh động, ngay cả khi chương trình đang chạy.
- Khi muôn thêm hành động mới cho đối tượng mà không làm ảnh hưởng đến nó.
- Khi muôn kế thừa một lớp mà lớp đó không thể kế thừa (sealed bên c#).
- Việc sử dụng kế thừa tốn công sức: muôn thêm hành động mà có thể phải hiện thực các hành động khác không liên quan. Ngoài ra việc kế thừa đó là tĩnh, không thể thay đổi hành động của đối tượng đã tồn tại mà chỉ có thể tạo ra đối tượng mới.

### 3.4.3. Đặc điểm

#### 3.4.3.1. Cấu trúc mẫu



Hình 3.9: Sơ đồ lớp của mẫu thiết kế Decorator

### **3.4.3.2. Thành phần trong cấu trúc mẫu**

- Cấu trúc:
  - Component: Là interface quy định các phương thức chung cần phải có cho cả đối tượng bọc và đối tượng được bọc.
  - Concrete Component: Là lớp hiện thực các phương thức của interface. Là lớp đối tượng được bọc
  - Base Decorator: Abstract class dùng để duy trì một tham chiếu của đối tượng Component và đồng thời cài đặt phương thức của interface.
  - Concrete Decorator: lớp hiện thực (implements) các phương thức của Decorator, nó cài đặt thêm các tính năng mới cho Component. Nó sẽ override phương thức ở lớp cha và thực hiện phương thức được thêm trước hoặc sau phương thức chính.
- Sự cộng tác:
  - Dùng composition thay vì inheritance. Khác ở đây là composition là has-a còn inheritance. Việc này giúp cho multilevel inheritance.  
Cả 2 cách này đều dùng để tái sử dụng code. Nhưng kế thừa được kết hợp một cách chặt chẽ trong khi thành phần thì liên kết lỏng lẻo.  
Không có kiểm soát truy cập khi kế thừa (trừ C++)  
Linh động hơn khi sử dụng các phương thức hữu ích.
  - Tất cả các wrapper có một trường để lưu trữ một giá trị của một đối tượng gốc.

### **3.4.4. Khả năng ứng dụng**

- Khi muốn thêm hành động mới cho đối tượng mà không làm ảnh hưởng đến đối tượng.
- Khi muốn kế thừa một lớp mà lớp đó không thể kế thừa (sealed bên c#).
- Việc sử dụng kế thừa tôn công sức: muốn thêm 1 hành động mà có thể phải hiện thực các hành động khác không liên quan. Ngoài ra việc kế thừa đó là tĩnh, không thể thay đổi hành động của đối tượng đã tồn tại ở runtime mà bạn chỉ có thể tạo ra đối tượng mới.

### **3.4.5. Hệ quả**

#### **3.4.5.1. Ưu điểm**

- Tăng cường khả năng mở rộng của đối tượng, bởi vì những thay đổi được thực hiện bằng cách implement trên các lớp mới.
- Client sẽ không nhận thấy sự khác biệt khi bạn đưa cho nó một wrapper thay vì đối tượng gốc.
- Cho phép thêm hoặc xóa tính năng của một đối tượng trong lúc chạy chương trình.

#### **3.4.5.2. Nhược điểm**

- Khó gỡ 1 wrapper khi wrap chồng lên nhau
- Khó cài đặt 1 decorator mà không liên quan tới stack wrapper mà nó ở nằm trong (khó cài đặt khi mà phương thức mới không dựa vào thứ tự của ngăn xếp)
- Nếu wrapper quá nhiều có thể gây dễ nhầm lẫn trong cài đặt

#### **3.4.6. Các mẫu thiết kế liên quan**

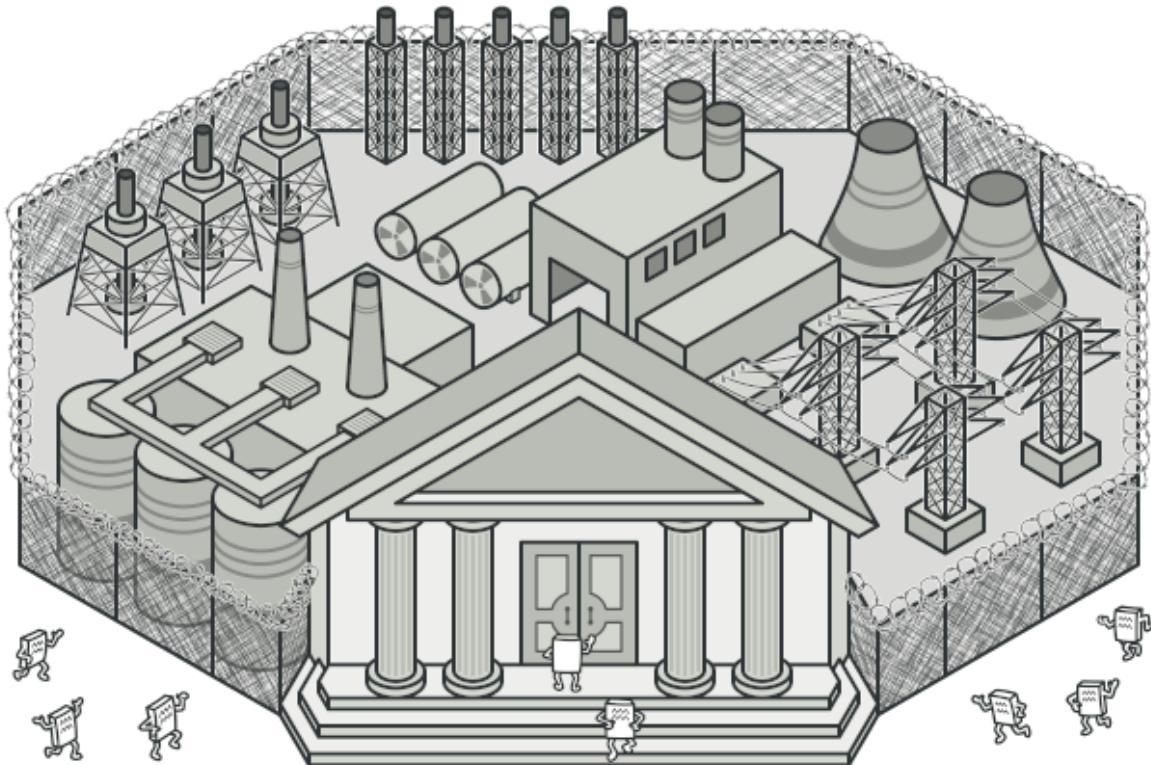
- Strategy: Decorator thay đổi lớp vỏ của một đối tượng bằng cách bọc nó, Strategy thì cho phép thay đổi nội dung của phương thức.
- Chain of Responsibility: Có cấu trúc rất giống nhau, đều có thành phần đệ quy để chuyển việc thực thi qua một loạt đối tượng. Nhưng có một số khác biệt.
- Decorator: đã có bảng so sánh. Có thể phối hợp để mở rộng tính năng của Leaf một cách đơn giản và ít chi phí thiết kế.
- Sử dụng kết hợp Decorator, Composite, Prototype để nhân bản ra đối tượng phức tạp thay vì tạo lại từ các mẫu nhỏ.

Một số so sánh với mẫu thiết kế khác cùng loại:

<b>Composite</b>	<b>Decorator</b>
Có (0,n) thành phần con	Có (0,1) thành phần con
Tổng hợp kết quả yêu cầu của thành phần con	Thêm chức năng cho thành phần con

<b>Decorator</b>	<b>Adapter</b>
Nâng cấp đối tượng đã tồn tại mà không làm thay đổi interface	Thay đổi interface của đối tượng đã tồn tại
Hỗ trợ đê quy thành phần	Không hỗ trợ đê quy thành phần
Cung cấp cho nó giao diện nâng cấp	Cung cấp giao diện khác cho đối tượng được bao bọc

### 3.5. Facade



Hình 3.10 Minh họa mẫu thiết kế Facade

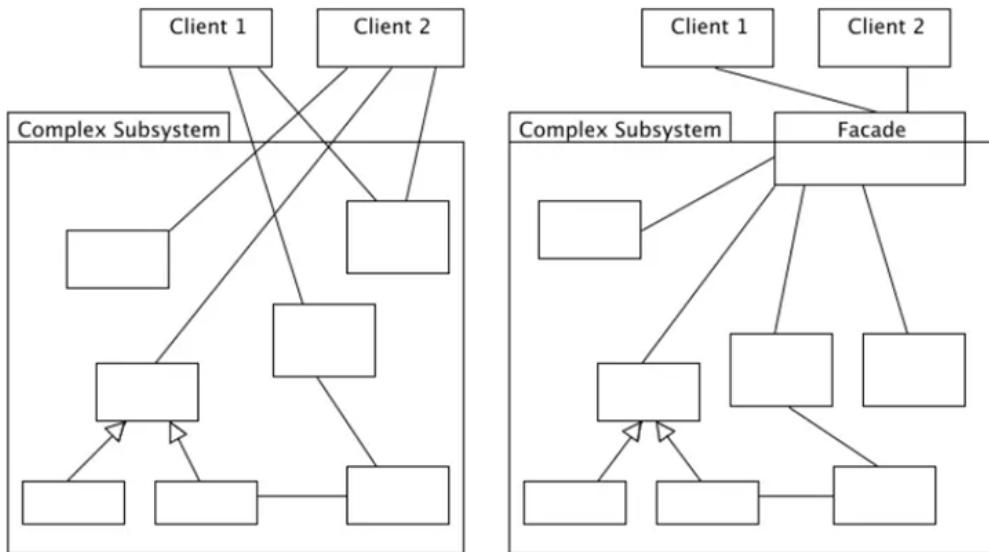
#### 3.5.1. Tổng quan

Façade là mẫu thiết kế thuộc nhóm Structural Pattern với việc cung cấp một interface hợp nhất cho một tập hợp các giao diện trong một hệ thống con (subsystem). Façade định nghĩa một interface cấp cao giúp subsystem dễ sử dụng hơn.

- Interface cấp cao này có ý định không hỗ trợ tất cả các trường hợp sử dụng của hệ thống, mà chỉ hỗ trợ những trường hợp quan trọng nhất
- Khi cần thiết client vẫn sẽ truy cập được vào các thành phần của subsystem
- Mục tiêu là che giấu các hoạt động phức tạp bên trong subsystem, làm cho subsystem dễ sử dụng hơn.

#### 3.5.2. Motivation

Cung cấp một interface đơn giản hóa, cải tiến hoặc hướng đối tượng hơn cho một subsystem quá phức tạp (hoặc có thể đơn giản là phức tạp hơn mức cần thiết cho việc sử dụng hiện tại), được thiết kế kém, một hệ thống kế thừa đã cũ hoặc không phù hợp với hệ thống tiêu thụ nó.

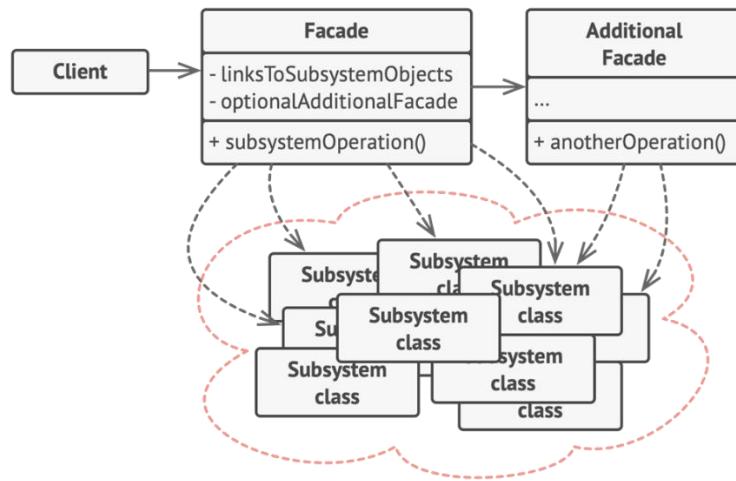


Hình 3.11: Mô tả Façade

- Khi cần một interface có giới hạn nhưng đơn giản cho một subsystem phức tạp: Thông thường, các subsystem 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 subsystem có thể trở nên linh hoạt hơn và dễ sử dụng lại hơn trong các ngữ cảnh khác nhau, nhưng số lượng cấu hình và mã soạn sẵn mà nó yêu cầu từ Client ngày càng lớn hơn. Facade cố gắng khắc phục sự cố này bằng cách cung cấp lối tắt đến các tính năng được sử dụng nhiều nhất của subsystem phù hợp với hầu hết các yêu cầu của Client
- Khi muốn gom nhóm chức năng lại để Client dễ sử dụng: Khi hệ thống có rất nhiều lớp làm người sử dụng rất khó để có thể hiểu được quy trình xử lý của chương trình. Và khi có rất nhiều subsystem mà mỗi subsystem đó lại có những giao diện riêng lẻ của nó nên rất khó cho việc sử dụng phối hợp. Khi đó có thể sử dụng Facade Pattern để tạo ra một giao diện đơn giản cho người sử dụng một hệ thống phức tạp
- Khi muốn cấu trúc subsystem thành các lớp (layer): Khi bạn muốn phân lớp các subsystem. Dùng Facade Pattern để định nghĩa cổng giao tiếp chung cho mỗi subsystem, do đó giúp giảm sự phụ thuộc của các subsystem vì các hệ thống này chỉ giao tiếp với nhau thông qua các cổng giao diện chung đó
- Đóng gói nhiều chức năng, che giấu thuật toán phức tạp
- Cần một interface không rắc rối mà dễ sử dụng

### 3.5.3. Đặc điểm

#### 3.5.3.1. Cấu trúc mẫu



Hình 3.12: Sơ đồ lớp của mẫu thiết kế Facade

#### 3.5.3.2. Thành phần trong cấu trúc mẫu

- Facade:
  - Nắm rõ subsystem nào đảm nhiệm việc đáp ứng yêu cầu của Client
  - Ủy quyền yêu cầu của Client cho subsystem tương ứng
- Additional Facade: Được tạo ra để tránh làm cho một Facade trở nên phức tạp, có thể được sử dụng bởi Client hoặc Facade
- Complex Subsystem:
  - Bao gồm nhiều đối tượng khác nhau, được cài đặt các chức năng của subsystem, xử lý công việc được gọi bởi Facade.
  - Các lớp của subsystem không biết đến Facade và không tham chiếu đến nó, chúng chỉ hoạt động trong hệ thống và làm việc với nhau
- Client: Sử dụng Facade để tương tác với subsystem thay vì gọi trực tiếp

### 3.5.4. Khả năng ứng dụng

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

- Khi muốn gom nhóm chức năng lại để Client dễ sử dụng: Khi hệ thống có rất nhiều lớp làm người sử dụng rất khó để có thể hiểu được quy trình xử lý của chương trình. Và khi có rất nhiều subsystem mà mỗi subsystem lại có những giao diện riêng nên rất khó để sử dụng phối hợp. Khi đó có thể sử dụng Facade Pattern để tạo ra một giao diện đơn giản cho người sử dụng một hệ thống phức tạp.
- Khi muốn cấu trúc subsystem thành các lớp (layer): Khi bạn muốn phân lớp các subsystem. Dùng Facade Pattern để định nghĩa cổng giao tiếp chung cho mỗi subsystem, do đó giúp giảm sự phụ thuộc của các subsystem vì các hệ thống này chỉ giao tiếp với nhau thông qua các cổng giao diện chung đó.
- Đóng gói nhiều chức năng, che giấu thuật toán phức tạp.
- Cần một interface không rắc rối mà dễ sử dụng.

### **3.5.5. Hệ quả**

#### **3.5.5.1. Ưu điểm**

- Ân việc triển khai subsystem khỏi Client, giảm độ phức tạp khi truy cập vào các subsystem, giúp subsystem dễ sử dụng hơn
- Thúc đẩy sự liên kết yếu giữa subsystem và các Client của nó, cho phép thay đổi mà không ảnh hưởng đến Client
- Tăng khả năng độc lập và khả chuyển, giảm sự phụ thuộc
- Giảm thiểu sự phụ thuộc biên dịch vào hệ thống con
- Không ngăn chặn việc sử dụng lớp subsystem nếu Client cần. Client vẫn có thể lựa chọn giữa tính dễ sử dụng và tính tổng quát (Bỏ qua Facade và sử dụng các đối tượng trong subsystem)

#### **3.5.5.2. Nhược điểm**

- Khi một Facade có quá nhiều nhiệm vụ với nhiều hàm chức năng, nó sẽ trở thành một anti-pattern God object (Là đối tượng tham chiếu đến một lượng lớn các loại riêng biệt khác, chứa quá nhiều các phương thức không liên quan hoặc phương thức không được phân loại).
- Việc sử dụng Facade cho các hệ thống đơn giản, ko quá phức tạp trở nên dư thừa

### **3.5.6. Các mẫu thiết kế liên quan**

- Adapter: Adapter cố gắng điều chỉnh interface hiện có có thể sử dụng được, trong khi Facade định nghĩa interface mới, đơn giản hóa mục đích

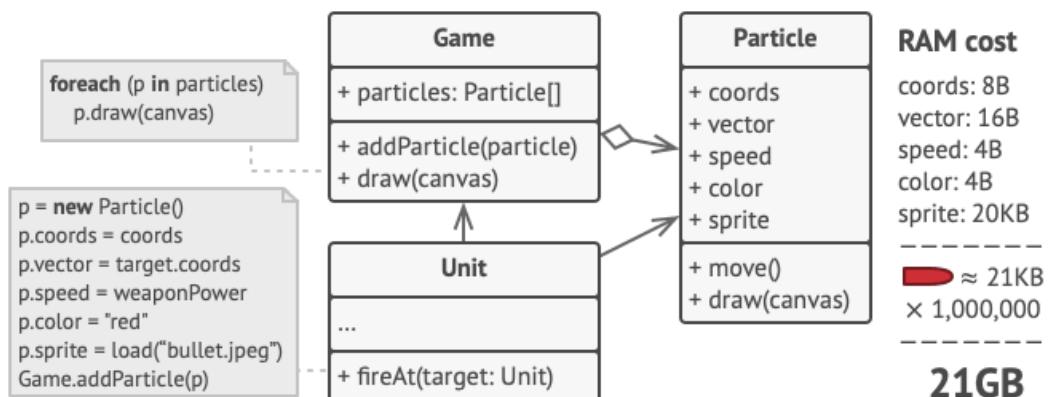
- Flyweight: Flyweight đưa ra cách xử lý nhiều đối tượng nhỏ, trong khi Facade đưa ra cách xử lý đối với một đối tượng duy nhất đại diện cho toàn bộ subsystem
- Abstract Factory
- Mediator
- Singleton có thể hoạt động như điểm truy cập duy nhất cho hệ thống con phức tạp
- Proxy

### 3.6. Flyweight

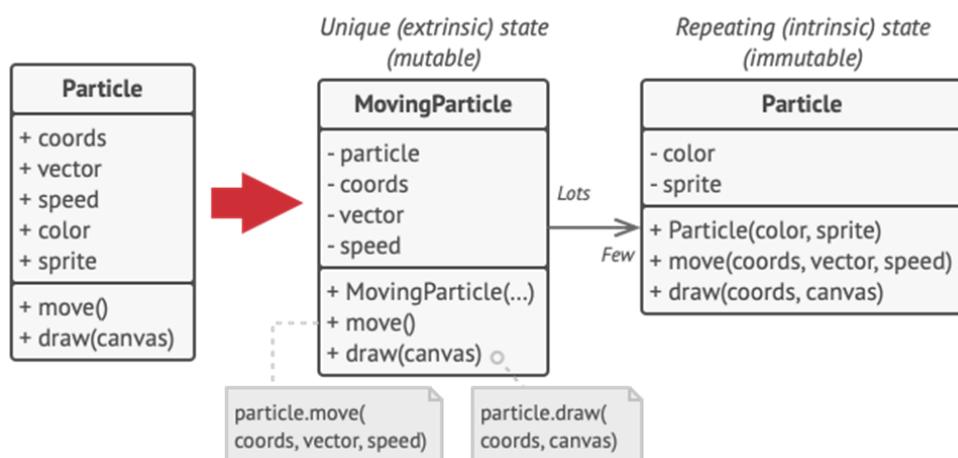
#### 3.6.1. Tổng quan

Flyweight là mẫu thiết kế thuộc lớp Structural Pattern với mục tiêu cho phép chương trình chứa nhiều đối tượng hơn vào dung lượng RAM có sẵn bằng cách chia sẻ phần trạng thái chung giữa nhiều đối tượng thay vì giữ tất cả dữ liệu trong mỗi đối tượng khi chạy chương trình. Mục đích của Flyweight là sử dụng việc share để hỗ trợ thao tác với một lượng lớn đối tượng kích thước nhỏ, giải quyết vấn đề hiệu suất.

#### 3.6.2. Motivation



Hình 3.13: Tiêu thụ RAM của game bắn súng khi tạo mới từng đối tượng.

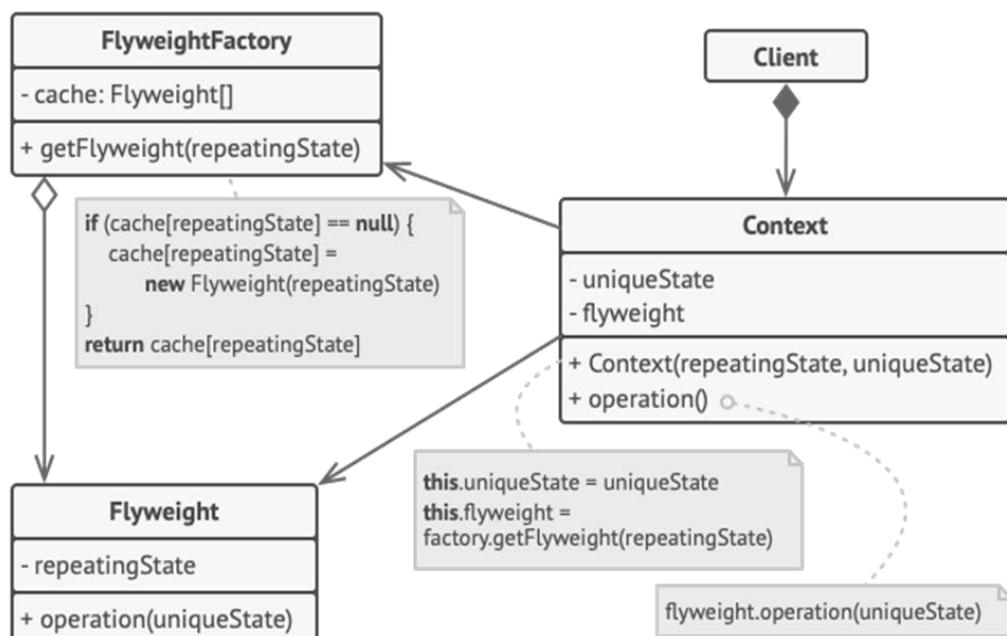


Hình 3.14: Áp dụng Flyweight vào việc tạo đối tượng cho game bắn súng

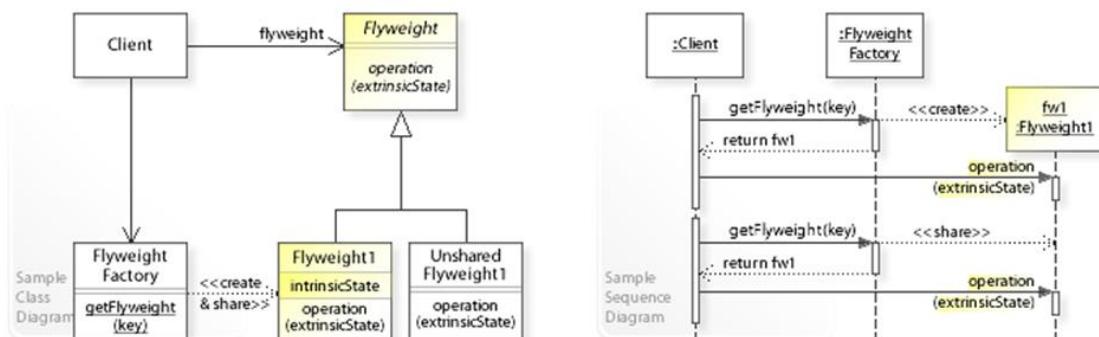
- Flyweight object là immutable, nghĩa là không thể thay đổi khi nó đã được khởi tạo.
- Chỉ sử dụng mẫu Flyweight khi chương trình của bạn phải hỗ trợ một số lượng lớn các đối tượng hầu như không vừa với RAM khả dụng
- Tái sử dụng đối tượng tương tự đã tồn tại bằng cách lưu trữ nó hoặc tạo đối tượng mới khi không tìm thấy đối tượng phù hợp.
- Tạo một số lượng lớn các đối tượng của một lớp nào đó.
- Giảm tải cho bộ nhớ thông qua cách chia sẻ các đối tượng. Vì vậy mà performance của hệ thống được tăng lên.

### 3.6.3. Đặc điểm

#### 3.6.3.1. Cấu trúc mẫu



Hình 3.15: Sơ đồ lớp của mẫu thiết kế Flyweight



Hình 3.16: Sơ đồ lớp và trình tự của mẫu thiết kế Flyweight theo GoF

### **3.6.3.2. Thành phần trong cấu trúc mẫu**

- Intrinsic state: trạng thái này chứa dữ liệu không thể thay đổi (unchangeable) và không phụ thuộc (independent) vào ngữ cảnh (context) của đối tượng flyweight
- Extrinsic state: thể hiện tính chất phụ thuộc ngữ cảnh của đối tượng flyweight. Trạng thái này chứa các thuộc tính và dữ liệu được áp dụng hoặc được tính toán trong thời gian thực thi.
- Cấu trúc:
  - Flyweight: là một interface/ abstract class, định nghĩa các thành phần của một đối tượng, lấy trạng thái bên ngoài và thực hiện hoạt động.
  - Flyweight1: class chứa các thành phần có thể sử dụng lại để làm mẫu chung.
  - Context: class chứa những thành phần riêng mà từng object phải có đồng thời sử dụng các thành phần chung từ flyweight
  - FlyweightFactory: quản lý các flyweight

### **3.6.4. Khả năng ứng dụng**

- Chỉ sử dụng mẫu Flyweight khi chương trình của bạn phải hỗ trợ một số lượng lớn các đối tượng hầu như không vừa với RAM khả dụng
- Tái sử dụng đối tượng tương tự đã tồn tại bằng cách lưu trữ nó hoặc tạo đối tượng mới khi không tìm thấy đối tượng phù hợp.
- Tạo một số lượng lớn các đối tượng của một lớp nào đó.
- Giảm tải cho bộ nhớ thông qua cách chia sẻ các đối tượng. Vì vậy mà performance của hệ thống được tăng lên.

### **3.6.5. Hệ quả**

#### **3.6.5.1. Ưu điểm**

- Tiết kiệm ram khi chương trình sử dụng nhiều đối tượng giống nhau

#### **3.6.5.2. Nhược điểm**

- Có thể đánh đổi giữa việc sử dụng RAM với chu kỳ CPU.
- Trong mẫu thiết kế Flyweight, một số dữ liệu ngữ cảnh cần được tính toán lại mỗi khi trạng thái nội tại của đối tượng flyweight vẫn giữ nguyên, nhưng trạng thái bên ngoài (tức là dữ liệu ngữ cảnh) thay đổi. Điều này là do trạng thái bên trong được chia sẻ giữa nhiều đối tượng, trong khi trạng thái bên ngoài là duy nhất cho từng

đối tượng. Ví dụ, trong ví dụ mặt cười, trạng thái bên trong của đối tượng có trọng lượng có thể là hình dạng của khuôn mặt cười, trong khi trạng thái bên ngoài có thể là vị trí, kích thước và màu sắc của từng khuôn mặt cười. Nếu hình dạng của mặt cười giống nhau cho tất cả 100 mặt cười, nó có thể được lưu trữ dưới dạng trạng thái nội tại và được chia sẻ giữa tất cả các mặt cười. Tuy nhiên, nếu vị trí, kích thước hoặc màu sắc của mặt cười thay đổi, đối tượng trọng lượng bay cần tính toán lại tọa độ, bán kính hoặc màu của mặt cười. Đây có thể là sự đánh đổi giữa mức sử dụng bộ nhớ và chu kỳ CPU.

- Mã trở nên phức tạp. Các thành viên mới sẽ thắc mắc tại sao trạng thái của một thực thể lại bị tách ra theo cách này.
- Tách trường chung ra khỏi đối tượng để tạo flyweight → thêm vào một lớp khác làm khó khăn trong việc bảo trì và mở rộng.

### 3.6.6. Các mẫu thiết kế liên quan

Flyweight	Façade	Singleton
<ul style="list-style-type: none"> <li>- Dùng để cài đặt nhiều đối tượng nhỏ.</li> <li>- Có thể nhiều đối tượng với trạng thái nội tại khác nhau.</li> <li>- Là immutable</li> </ul>	<ul style="list-style-type: none"> <li>- Dùng cài đặt 1 đối tượng duy nhất đại diện cho toàn bộ hệ thống con</li> </ul>	<ul style="list-style-type: none"> <li>- Chỉ nên có một phiên bản Singleton</li> <li>- Đối tượng Singleton có thể thay đổi được.</li> </ul>

## 3.7. Proxy

### 3.7.1. Tổng quan

Proxy (tên gọi khác là Surrogate) là mẫu thiết kế thuộc nhóm Structural Pattern. Mẫu cung cấp một “đại diện thay thế” hoặc “giữ chỗ” cho một đối tượng khác để kiểm soát quyền truy cập vào nó:

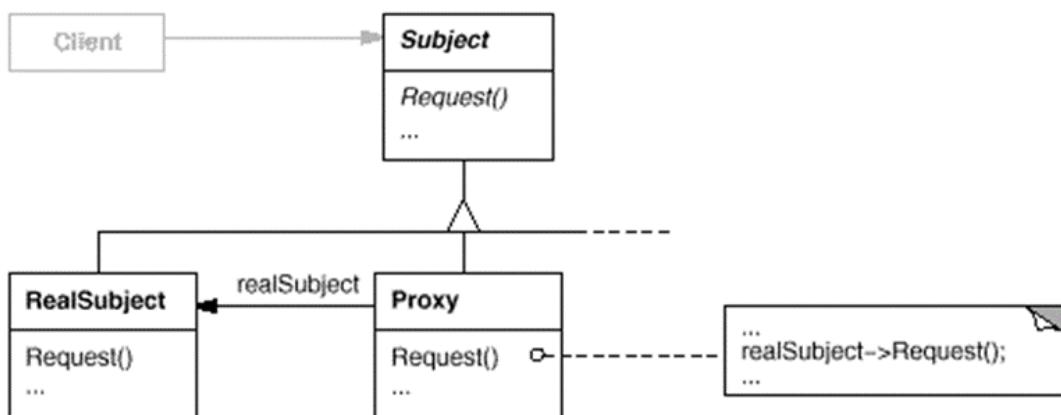
- Remote Proxy: Cung cấp 1 đại diện cục bộ (local representative) cho 1 đối tượng thuộc một không gian địa chỉ khác.
- Virtual Proxy: Tạo ra một đối tượng “tốn kém” khi được yêu cầu.
- Protection Proxy: Kiểm soát quyền truy cập của đối tượng thực.
- Giải quyết vấn đề về security, performance, validation,...

### 3.7.2. Motivation

- Kiểm soát quyền truy cập vào một đối tượng để trì hoãn toàn bộ chi phí tạo và khởi tạo đối tượng.
- Vấn đề:
  - Bạn cần truy cập vào 1 đối tượng lớn và đối tượng này chiếm một lượng lớn tài nguyên hệ thống. Nhưng không hẳn lúc nào cũng sử dụng
  - Implement lazy initialization - chỉ tạo khi cần. Khi đó client muốn truy cập đều phải chạy qua đoạn code này, tuy nhiên vấn đề phát sinh là sẽ khiến code duplicate
  - Điều hay nhất là có thể là đưa dòng code này vào chính đối tượng đó. Nhưng nếu lớp này là 3rd party thì không thể
- Giải pháp:
  - Mẫu Proxy gợi ý bạn nên tạo một lớp proxy có cùng interface với đối tượng dịch vụ đó.

### 3.7.3. Đặc điểm

#### 3.7.3.1. Cấu trúc mẫu



Hình 3.17: Sơ đồ lớp của mẫu thiết kế Proxy

#### 3.7.3.2. Thành phần trong cấu trúc mẫu

- Subject: Định nghĩa interface cho Real Subject và Proxy. Cho phép mọi Client “đối xử” với proxy như một Service thực sự.
- RealSubject: Là đối tượng thực hiện các công việc thực tế mà proxy sẽ đại diện.
- Proxy:
  - Chứa một tham chiếu đến đối tượng RealSubject cho phép Proxy truy cập vào RealSubject

- Sau khi proxy hoàn tất quá trình xử lý (VD: lazy initialization, logging, access control, caching, v.v.), nó sẽ chuyển tiếp yêu cầu đến đối tượng RealSubject
  - Kiểm soát quyền truy cập vào RealSubject và có thể chịu trách nhiệm tạo và xóa nó
  - Một vài trách nhiệm khác tùy vào loại như:
  - Remote Proxy: Mã hóa yêu cầu và các đối số của nó và gửi yêu cầu được mã hóa tới RealSubject trong một không gian địa chỉ khác
  - Virtual Proxy: Cache thông tin bổ sung về RealSubject để có thể trì hoãn việc truy cập nó
  - Protection Proxy: Kiểm tra xem Client có quyền truy cập cần thiết để thực hiện yêu cầu hay không
- Client: Có thể làm việc với cả Service và Proxy thông qua interface chung của chúng

#### **3.7.4. Khả năng ứng dụng**

- Khi nào có nhu cầu tham chiếu đến một đối tượng linh hoạt hoặc tinh vi hơn so với một con trỏ thông thường.
- Lazy initialization: Trì hoãn quá trình khởi tạo đối tượng cho đến thời điểm cần thiết (Khi đối tượng sử dụng là một đối tượng heavyweight luôn hoạt động gây lãng phí tài nguyên hệ thống nhưng lại ít khi được sử dụng).
- Kiểm soát truy cập (Protection proxy): Khi muốn chỉ những Client cụ thể mới có thể sử dụng đối tượng.
- Local execution of a remote service (remote proxy): Đây là khi đối tượng service được đặt trên một máy chủ từ xa.
- Logging requests (logging proxy): Khi bạn muốn giữ lịch sử của các yêu cầu đối với đối tượng service.
- Caching request results (caching proxy): Khi bạn cần lưu trữ kết quả của các yêu cầu máy khách và quản lý vòng đời của bộ nhớ cache này, đặc biệt nếu kết quả khá lớn.
- Smart reference: Khi bạn cần loại bỏ một đối tượng nặng khi không có máy khách nào sử dụng nó.

### **3.7.5. Hệ quả**

#### **3.7.5.1. Ưu điểm**

- Cải thiện Performance thông qua lazy loading.
- Nó cung cấp sự bảo vệ cho đối tượng thực từ thế giới bên ngoài.
- Giảm chi phí khi có nhiều truy cập vào đối tượng có chi phí khởi tạo ban đầu lớn.
- Nguyên tắc Mở/Đóng: Thêm proxy mới mà không cần thay đổi service hoặc Client.

#### **3.7.5.2. Nhược điểm**

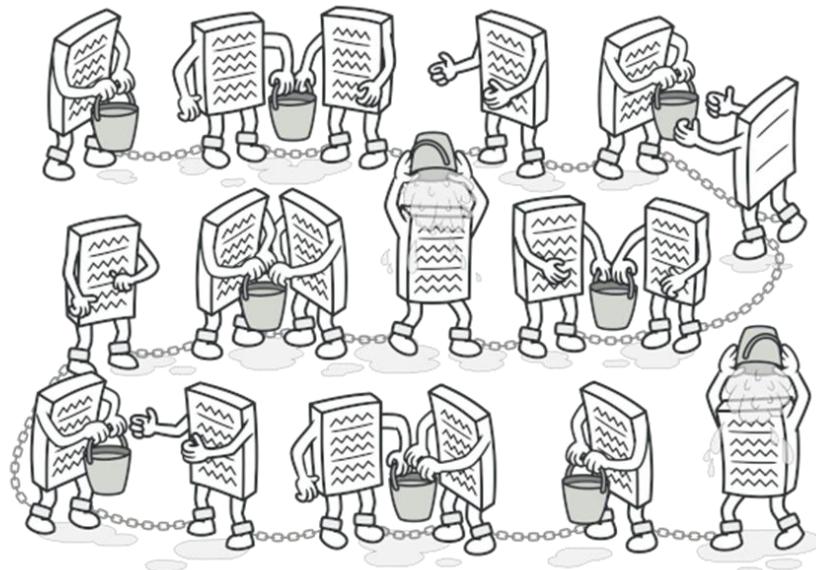
- Code trở nên phức tạp.
- Phản hồi từ phía đối tượng thực có thể bị delay.

### **3.7.6. Các mẫu thiết kế liên quan**

- Adapter tạo ra một interface khác cho đối tượng đang được bọc, Proxy cung cấp cùng một interface với đối tượng và Decorator cung cấp một interface nâng cao hơn.
- Decorator và Proxy có cấu trúc tương tự, nhưng mục đích của chúng khác nhau. Decorator thêm hành vi vào đối tượng, trong khi Proxy kiểm soát quyền truy cập

## Chương 4. BEHAVIOR PATTERN

### 4.1. Chain of Responsibility



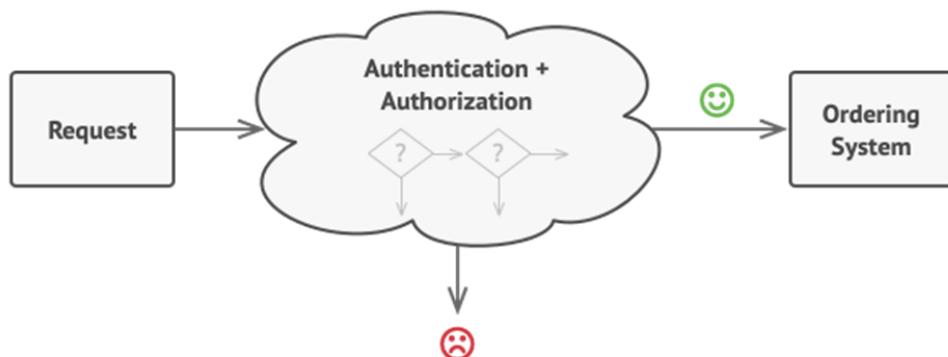
Hình 4.1: Mô tả về CoR

#### 4.1.1. Tổng quan

Chain of Responsibility (CoR) là mẫu thiết kế thuộc nhóm Behavioral Pattern được tạo ra với mục đích cho phép một đối tượng gửi một yêu cầu nhưng không biết đối tượng nào sẽ nhận và xử lý nó. Điều này được thực hiện bằng cách kết nối các đối tượng nhận yêu cầu thành một chuỗi (chain) và gửi yêu cầu theo chuỗi đó cho đến khi có một đối tượng xử lý nó.

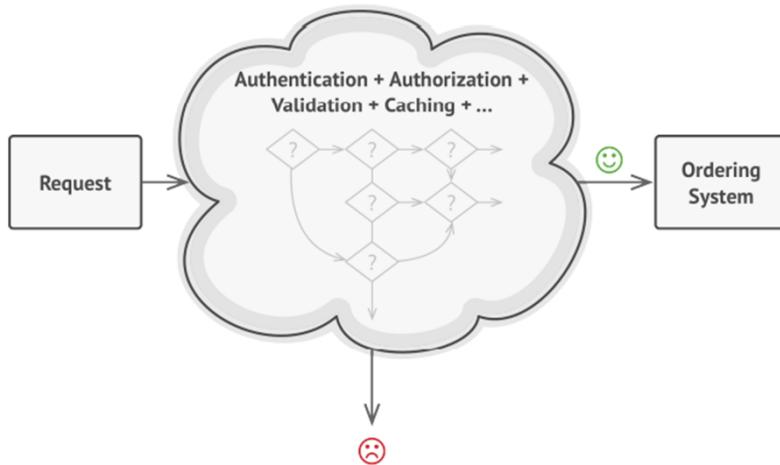
#### 4.1.2. Motivation

Vấn đề: Giả sử bạn làm việc với một hệ thống đặt hàng online. Bạn muốn hạn chế quyền truy cập vào hệ thống, chỉ cho user đã đăng nhập tạo các đơn đặt hàng. Mặt khác những user có quyền admin sẽ được toàn quyền truy cập vào các đơn đặt hàng.



Hình 4.2: Quy trình từ khi Request đến khi tới được Ordering System

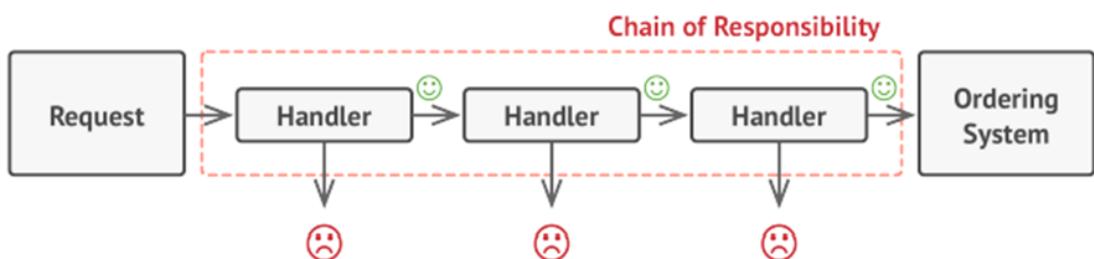
- Những giai đoạn kiểm tra (như kiểm tra user đã đăng nhập, user có quyền admin) cần phải thực hiện tuần tự. Ví dụ nếu việc kiểm tra user đăng nhập bị thất bại thì chúng ta không có lí do gì để kiểm tra tiếp tục các điều kiện khác.
- Thêm chức năng caching (lưu trữ tạm thời – lưu trạng thái đăng nhập) để tránh thực hiện lại các yêu cầu giống nhau, kiểm tra Password có đúng format hay không... Mỗi khi thêm 1 chức năng hàm kiểm tra ngày càng phức tạp, cho đến khi 1 ngày bạn cấu trúc lại code.



*The bigger the code grew, the messier it became.*

Hình 4.3: Ví dụ khi các hành động cần thực hiện ngày càng lớn lên

Giải pháp: Chuyển từng hành vi thành những object cụ thể gọi là handler. Mỗi kiểm tra sẽ extract thành 1 hàm duy nhất. Yêu cầu được truyền dọc theo các hàm này. Tất cả tạo nên chuỗi liên kết, cho đến khi yêu cầu được xử lý, đến hết mắt xích cuối.

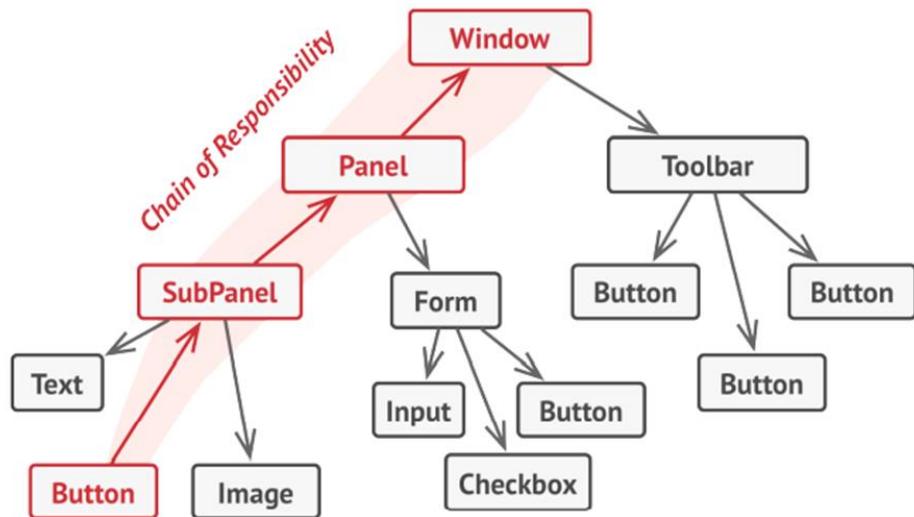


*Handlers are lined up one by one, forming a chain.*

Hình 4.4: Quá trình xử lý Request qua các Handler

- Một ví dụ khác trong lập trình windows forms. Các control trên một form chia sẻ sự kiện click. Giả sử bạn click vào một nút bấm, sự kiện

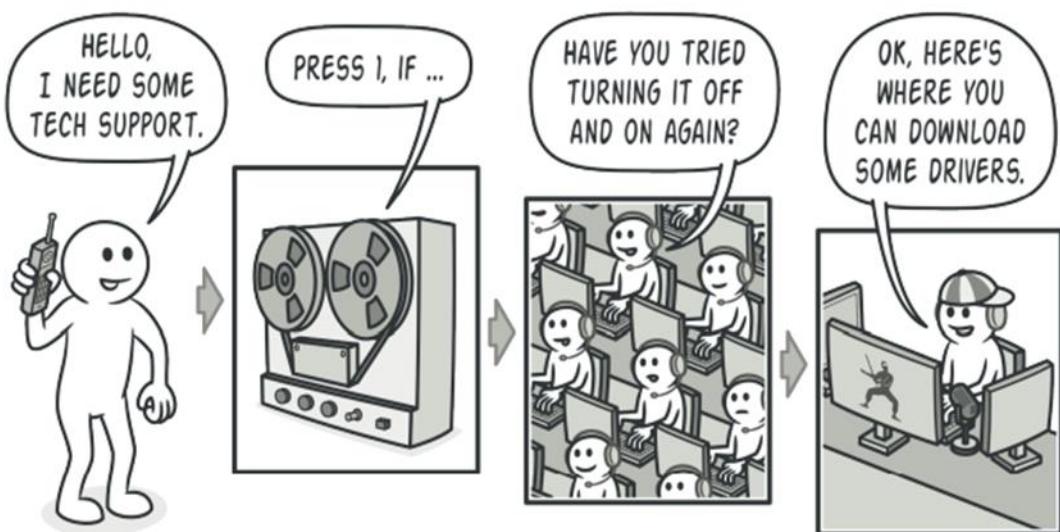
“click” sẽ chạy ngược chuỗi object từ Button tới Window (cấp cao nhất) để tìm đến đúng control có khả năng xử lý nó.



*A chain can be formed from a branch of an object tree.*

Hình 4.5: Mô tả chuỗi có thể được hình thành từ nhánh của cây

- Mẫu này gắn liền với nhiều hệ thống xử lý yêu cầu. Ví dụ tổng đài hay bất kì quy trình làm việc nào cũng sẽ đi theo tư tưởng của Chain of Responsibility. Khi khách hàng cần hỗ trợ thì nhân viên chăm sóc khách hàng sẽ chuyển yêu cầu của khách hàng đến những bộ phận liên quan để giải quyết hoặc là chuyển tiếp yêu cầu đến các bộ phận liên quan khác để giải quyết yêu cầu của khách hàng.

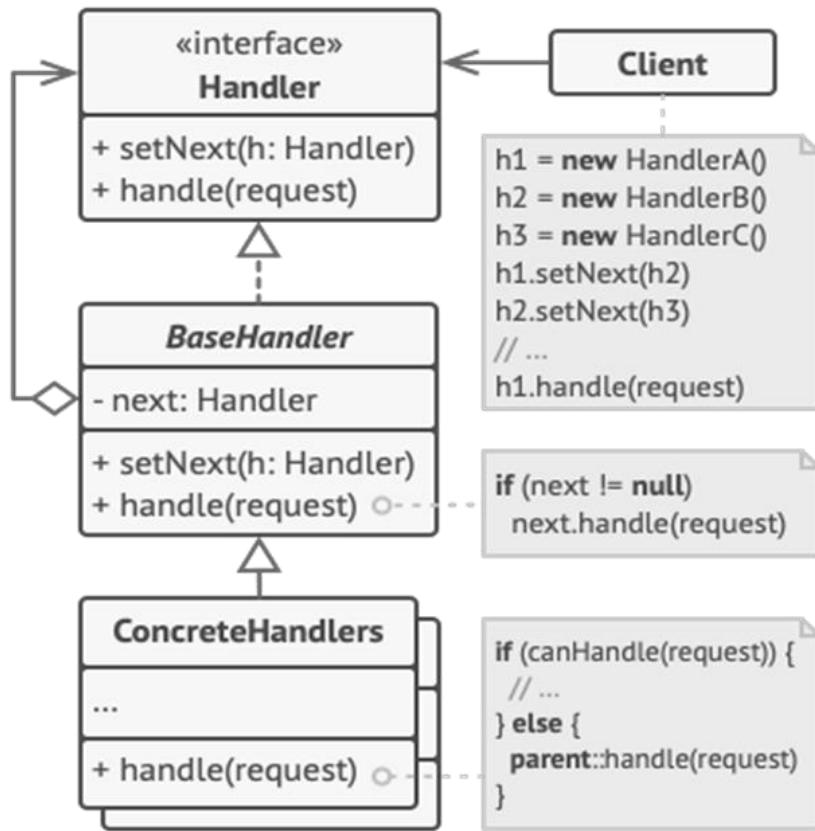


*A call to tech support can go through multiple operators.*

Hình 4.6: Mô tả về quá trình gọi hỗ trợ được thông qua bởi nhiều điện thoại viên

### 4.1.3. Đặc điểm

#### 4.1.3.1. Cấu trúc mẫu



Hình 4.7 Sơ đồ lớp của mẫu thiết kế CoR

#### 4.1.3.2. Thành phần trong cấu trúc mẫu

- Client: tạo ra các yêu cầu và yêu cầu đó sẽ được gửi đến các object tiếp nhận.
- Handler: định nghĩa 1 interface để xử lý các yêu cầu. Gán giá trị cho đối tượng successor (không bắt buộc).
- BaseHandler: lớp trừu tượng không bắt buộc. Có thể cài đặt các hàm chung cho Chain of Responsibility ở đây.
- ConcreteHandler: xử lý yêu cầu. Có thể truy cập đối tượng successor (thuộc class Handler). Nếu đối tượng ConcreteHandler không thể xử lý được yêu cầu, nó sẽ gởi lời yêu cầu cho successor của nó.
- Sự tương tác:
  - Client gọi hàm handle của Handler đầu Chain , truyền vào nội dung yêu cầu / tham số yêu cầu.
  - Handler đầu tiên xác định yêu cầu có xử lý được không. Nếu không xử lý được, handler gọi handler tiếp theo (nếu có).

#### 4.1.4. Khả năng ứng dụng

- Chain of Responsibility Pattern hoạt động như một danh sách liên kết kết hợp với việc đệ quy duyệt qua các phần tử .



Hình 4.8: Mô tả về động lực của CoR

- Khắc phục việc ghép cặp giữa bộ gửi và bộ nhận thông điệp, các đối tượng nhận thông điệp được kết nối thành một chuỗi và thông điệp được chuyển dọc theo chuỗi này đến khi gặp được đối tượng xử lý nó.
- Tách rời nơi gửi yêu cầu khỏi nơi xử lý yêu cầu
- Cho nhiều object có xử lý yêu cầu một cách tuần tự. Nếu 1 object không xử lý được yêu cầu, object sau sẽ xử lý.
- Tăng khả năng mở rộng và tái sử dụng của hệ thống. Các đối tượng xử lý trong chuỗi có thể được thay đổi hoặc mở rộng một cách độc lập mà không ảnh hưởng đến các thành phần khác trong hệ thống

#### 4.1.5. Hệ quả

##### 4.1.5.1. Ưu điểm

- Giảm kết nối (loose coupling): Mô hình Chain of Responsibility giúp giảm sự phụ thuộc giữa sender và receiver bằng cách tách rời chúng trong một chuỗi handler. Điều này giúp giảm sự ràng buộc và tạo ra sự linh hoạt trong việc thay đổi và mở rộng hệ thống.
- Tăng tính linh hoạt: Mô hình này tuân theo nguyên tắc Open/Closed, cho phép thay đổi hoặc mở rộng hành vi xử lý yêu cầu bằng cách thêm hoặc thay đổi handler trong chuỗi mà không cần sửa đổi code hiện có.
- Phân chia trách nhiệm: Mỗi handler chỉ chịu trách nhiệm xử lý một phần cụ thể của yêu cầu. Điều này giúp đảm bảo Single Responsibility và dễ dàng quản lý và bảo trì code.
- Khả năng thay đổi dây chuyền: Mô hình này cho phép thay đổi dây chuyền handler trong thời gian chạy. Bạn có thể thêm, loại bỏ hoặc thay đổi thứ tự các handler trong chuỗi mà không ảnh hưởng đến các thành phần khác của hệ thống.

#### **4.1.5.2. Nhược điểm**

- Một số yêu cầu có thể không được xử lý: Trường hợp tất cả Handler đều không xử lý.

#### **4.1.6. Các mẫu thiết kế liên quan**

- Command (Command Design Pattern): Mô hình Command cũng tách rời người gửi yêu cầu (sender) và người nhận yêu cầu (receiver), nhưng thay vì sử dụng một chuỗi handler, nó sử dụng các đối tượng Command riêng biệt để đóng gói yêu cầu và xử lý yêu cầu theo một cách tuần tự hoặc không tuần tự.
- Observer (Observer Design Pattern): Mô hình Observer cung cấp một cách để theo dõi và thông báo về sự thay đổi trong trạng thái của đối tượng. Tương tự như mô hình Chain of Responsibility, mô hình Observer cũng giảm sự phụ thuộc và tăng tính linh hoạt trong việc xử lý yêu cầu, nhưng theo một cách khác.
- Decorator (Decorator Design Pattern): Mô hình Decorator cho phép thêm chức năng bổ sung cho một đối tượng mà không làm thay đổi giao diện của nó. Tương tự, trong mô hình Chain of Responsibility, mỗi handler có thể thực hiện xử lý bổ sung trước hoặc sau khi chuyển tiếp yêu cầu cho handler tiếp theo trong chuỗi.
- Strategy (Strategy Design Pattern): Mô hình Strategy cho phép lựa chọn một thuật toán cụ thể từ một tập hợp các thuật toán khác nhau và giúp đối tượng sử dụng thuật toán mà không cần biết chi tiết về nó. Tương tự, trong mô hình Chain of Responsibility, mỗi handler có thể đại diện cho một thuật toán xử lý cụ thể và quyết định xem yêu cầu có được xử lý như thế nào.

### **4.2. Command**

#### **4.2.1. Tổng quan**

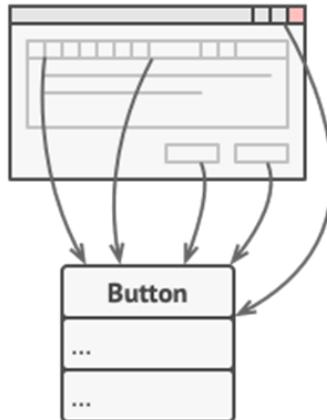
Command là mẫu thiết kế thuộc nhóm Behavioral Pattern, có tên khác là Action, transaction, cho phép bạn chuyển đổi một request thành một object độc lập chứa tất cả thông tin về request. Việc chuyển đổi này cho phép bạn tham số hóa các methods với các yêu cầu khác nhau như log, queue (undo/redo).

#### **4.2.2. Motivation**

Đặt vấn đề:

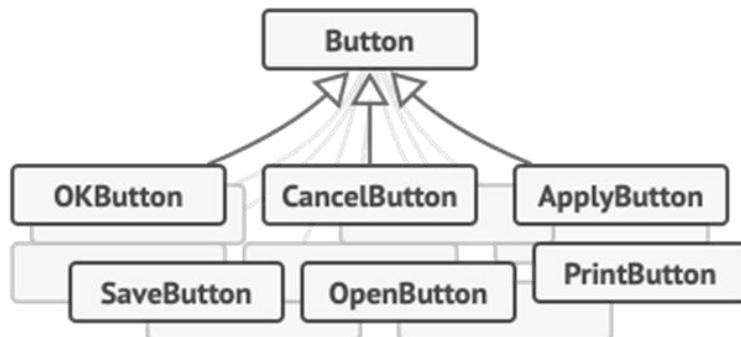
Hãy tưởng tượng rằng bạn đang làm việc trên một ứng dụng soạn thảo văn bản mới. Nhiệm vụ hiện tại của bạn là tạo một thanh công cụ với một loạt các nút cho các thao tác khác nhau của trình soạn thảo. Bạn đã tạo một lớp Nút rất gọn gàng có thể

được sử dụng cho các nút trên thanh công cụ, cũng như cho các nút chung trong các hộp thoại khác nhau.



*Hình 4.9: Hộp thoại có các nút*

Mặc dù tất cả các nút này trông giống nhau, nhưng tất cả chúng đều phải làm những việc khác nhau. Bạn sẽ đặt mã cho các trình xử lý nhập chuột khác nhau của các nút này ở đâu? Giải pháp đơn giản nhất là tạo hàng tấn lớp con cho mỗi nút được sử dụng. Các lớp con này sẽ chứa mã sẽ phải được thực thi khi nhấp vào nút.



*Hình 4.10: Button với những chức năng khác nhau*

Chẳng bao lâu sau, bạn nhận ra rằng cách tiếp cận này có nhiều thiếu sót. Đầu tiên, bạn có một số lượng lớn các lớp con và điều đó sẽ ồn ào nếu bạn không mạo hiểm vi phạm mã trong các lớp con này mỗi khi bạn sửa đổi lớp Nút cơ sở. Nói một cách đơn giản, mã GUI của bạn đã trở nên phụ thuộc một cách khó xử vào mã không ổn định của logic nghiệp vụ.



*Hình 4.11: Với mỗi loại button lại có code riêng của nó*

Giải pháp:

Thiết kế phần mềm tốt thường dựa trên nguyên tắc tách biệt các mối quan tâm (separation of concerns), điều này thường dẫn đến việc chia ứng dụng thành các lớp. Ví dụ phổ biến nhất: một lớp cho giao diện người dùng đồ họa và một lớp khác cho logic nghiệp vụ.

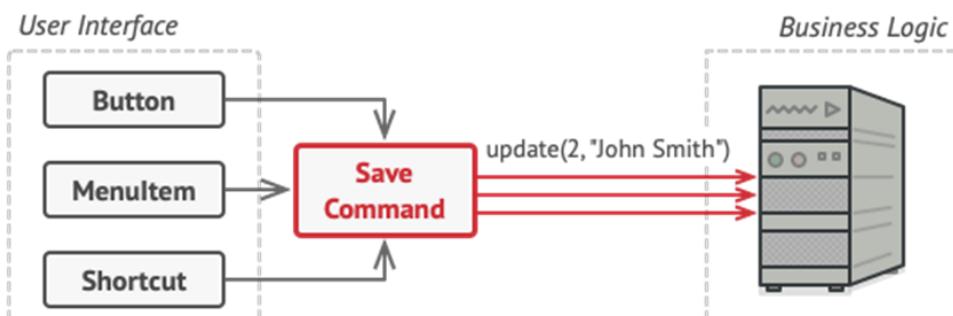
Trong mã nguồn, điều này có thể trông như sau: một đối tượng giao diện người dùng gọi một phương thức của một đối tượng logic nghiệp vụ, truyền cho nó một số đối số. Quá trình này thường được mô tả như một đối tượng gửi yêu cầu đến một đối tượng khác.



Hình 4.12: Mỗi button gửi riêng lẻ với từng mục đích

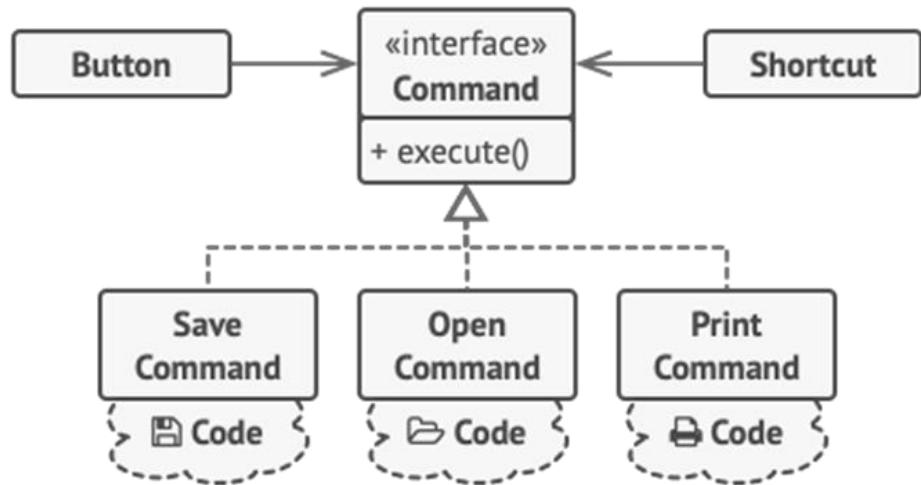
Command đề xuất rằng các đối tượng GUI không nên gửi các yêu cầu này trực tiếp. Thay vào đó, bạn nên trích xuất tất cả các chi tiết yêu cầu, chẳng hạn như đối tượng được gọi, tên của phương thức và danh sách các đối số, vào một lớp command riêng biệt với một phương thức duy nhất để kích hoạt yêu cầu này.

Đối tượng Command hoạt động như một liên kết giữa các đối tượng GUI và logic kinh doanh. Từ bây giờ trở đi, đối tượng GUI không cần phải biết đối tượng logic kinh doanh nào sẽ nhận yêu cầu và cách xử lý nó. Đối tượng GUI chỉ kích hoạt lệnh, và lệnh sẽ xử lý tất cả các chi tiết liên quan.



Hình 4.13: Các button được thống nhất về cùng một Command

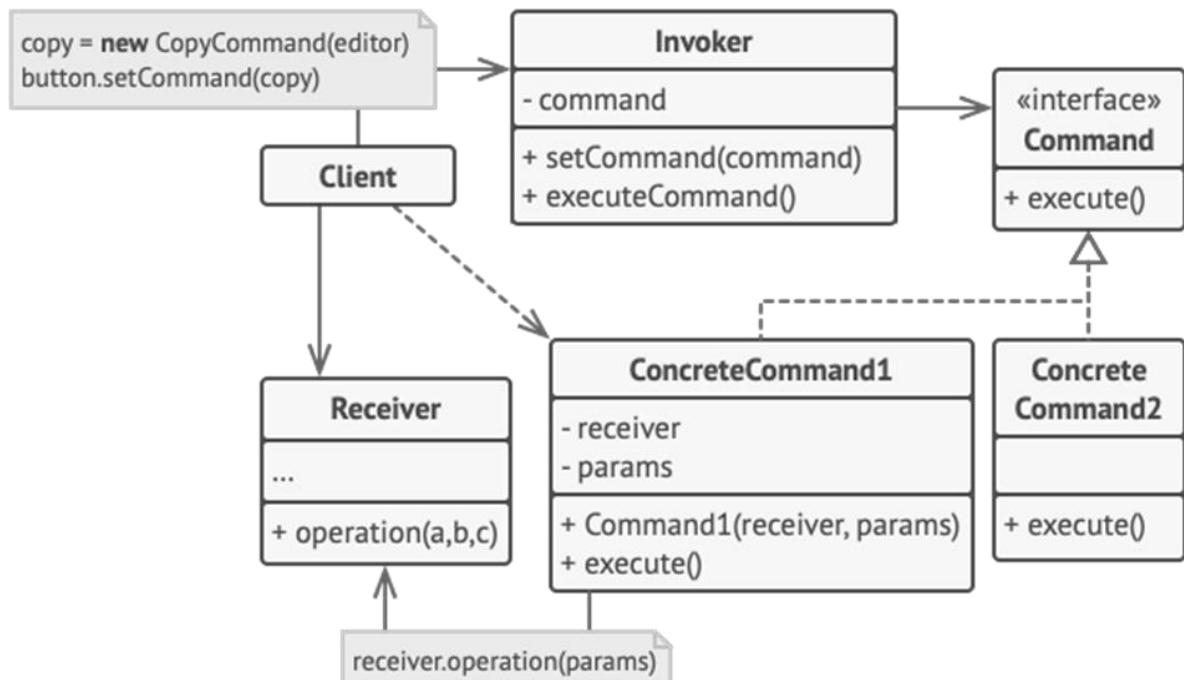
Bước tiếp theo là làm cho các lớp command của bạn triển khai cùng 1 interface. Thông thường, interface này chỉ có một phương thức thực thi duy nhất không có tham số. interface này cho phép bạn sử dụng các lệnh khác nhau với cùng một người gửi yêu cầu, mà không liên kết với các lớp command cụ thể. Như một lợi ích, bây giờ bạn có thể chuyển đổi các đối tượng command liên kết với người gửi yêu cầu, hiệu quả thay đổi hành vi của người gửi yêu cầu tại thời điểm chạy.



Hình 4.14: Sơ đồ lớp mức phân tích

#### 4.2.3. Đặc điểm

##### 4.2.3.1. Cấu trúc mẫu



Hình 4.15: Sơ đồ lớp của mẫu thiết kế Command

#### **4.2.3.2. Thành phần trong cấu trúc mẫu**

- 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 hoàn trên Receiver.
- 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 business logic cho case request. Trong phương thức execute() của ConcreteCommand chúng ta sẽ gọi method thích hợp trong Receiver.

#### **4.2.4. Khả năng ứng dụng**

- Khi cần tham số hóa các đối tượng theo một hành động thực hiện.
- Khi cần tạo và thực thi các yêu cầu vào các thời điểm khác nhau.
- Khi cần hỗ trợ tính năng undo, log, callback hoặc transaction.

#### **4.2.5. Hệ quả**

##### **4.2.5.1. Ưu điểm**

- Tính linh hoạt và mô-đun hóa: Mẫu Command Pattern tách biệt yêu cầu và hành động thực hiện, cho phép dễ dàng thay đổi và mở rộng các yêu cầu và hành động mà không ảnh hưởng đến các thành phần khác trong hệ thống.
- Hỗ trợ hoàn tác và làm lại: Command Pattern cung cấp khả năng hoàn tác và làm lại các hành động đã thực hiện. Các lệnh được lưu trữ trong một danh sách lịch sử, cho phép lặp lại các hành động hoặc hoàn ngược chúng theo nhu cầu.
- Dễ quản lý và theo dõi: Các yêu cầu và hành động được đóng gói trong các đối tượng Command, điều này giúp quản lý và theo dõi chúng một cách dễ dàng. Bạn có thể kiểm soát việc thực hiện các yêu cầu và ghi nhật ký các hành động đã thực hiện.

- Hỗ trợ mô hình giao diện người dùng: Command Pattern thích hợp cho việc xây dựng các giao diện người dùng tương tác, nơi người dùng có thể tương tác với các yêu cầu và thực hiện các hành động.
- Dễ dàng mở rộng: Mẫu Command Pattern cho phép dễ dàng mở rộng hệ thống bằng cách thêm mới các yêu cầu và hành động mới mà không cần sửa đổi nhiều mã hiện có.

#### **4.2.5.2. Nhược điểm**

- Tăng độ phức tạp: Mẫu Command Pattern có thể làm tăng độ phức tạp của hệ thống do sự tách biệt của yêu cầu và hành động. Điều này có thể tạo ra nhiều lớp và đối tượng, làm tăng khối lượng mã và khó hiểu hơn.
- Quản lý bộ nhớ: Sử dụng Command Pattern có thể tạo ra một số lượng lớn các đối tượng Command và lịch sử lệnh, ảnh hưởng đến việc quản lý bộ nhớ và hiệu suất của hệ thống.
- Khó khăn trong việc xử lý lỗi: Command Pattern có thể gây khó khăn trong việc xử lý lỗi, đặc biệt khi cần hoàn nguyên trạng thái trước khi lỗi xảy ra.

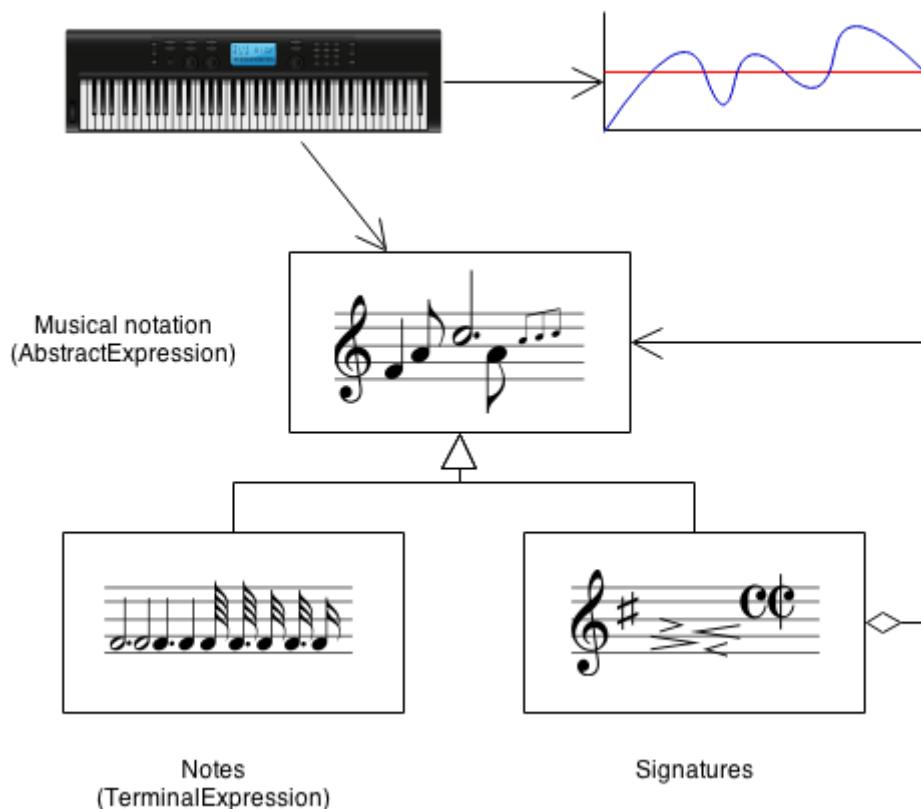
#### **4.2.6. Các mẫu thiết kế liên quan**

- Mặc dù Command và Strategy có vẻ tương tự vì cả hai đều được dùng để tham số hóa một đối tượng với một hành động nào đó, nhưng chúng có ý định khác nhau.
  - Bạn có thể sử dụng Command để chuyển đổi bất kỳ hoạt động nào thành một đối tượng. Các tham số của hoạt động trở thành các trường của đối tượng đó. Việc chuyển đổi này cho phép bạn trì hoãn việc thực thi hoạt động, đưa vào hàng đợi, lưu trữ lịch sử các lệnh, gửi các 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 thay đổi các thuật toán này trong một lớp ngõ cảnh duy nhất.
- Prototype có thể hữu ích khi bạn cần lưu trữ các bản sao của các Command vào lịch sử.
- Bạn có thể coi Visitor như một phiên bản mạnh mẽ hơn của mẫu Command. Các đối tượng Visitor có thể thực hiện các hoạt động trên các đối tượng khác nhau thuộc các lớp khác nhau.

Ngoài ra, các Pattern như Chain of Responsibility, Command, Mediator, Observer: giải quyết nhiều cách khác nhau để kết nối người gửi và người nhận yêu cầu.

- Chain of Responsibility: nhận yêu cầu của người gửi, và gửi yêu cầu đó (request) dọc theo một chuỗi những người nhận tiềm năng (processor) cho đến khi một trong số chúng xử lý nó. (1 - 0...n).
- Command: thiết lập các kết nối đơn hướng giữa người nhận với người gửi với một phân lớp (1-1).
- Mediator: loại bỏ các kết nối trực tiếp giữa người gửi và người nhận, buộc chúng phải liên lạc gián tiếp thông qua một đối tượng Mediator (n-n).
- Observer: định nghĩa một interface tách biệt cho phép nhiều người nhận đăng ký và hủy đăng ký nhận yêu cầu tại thời điểm chương trình thực thi (1-n).

### 4.3. Interpreter



Hình 4.16 Minh họa mẫu thiết kế Interpreter

#### 4.3.1. Tổng quan

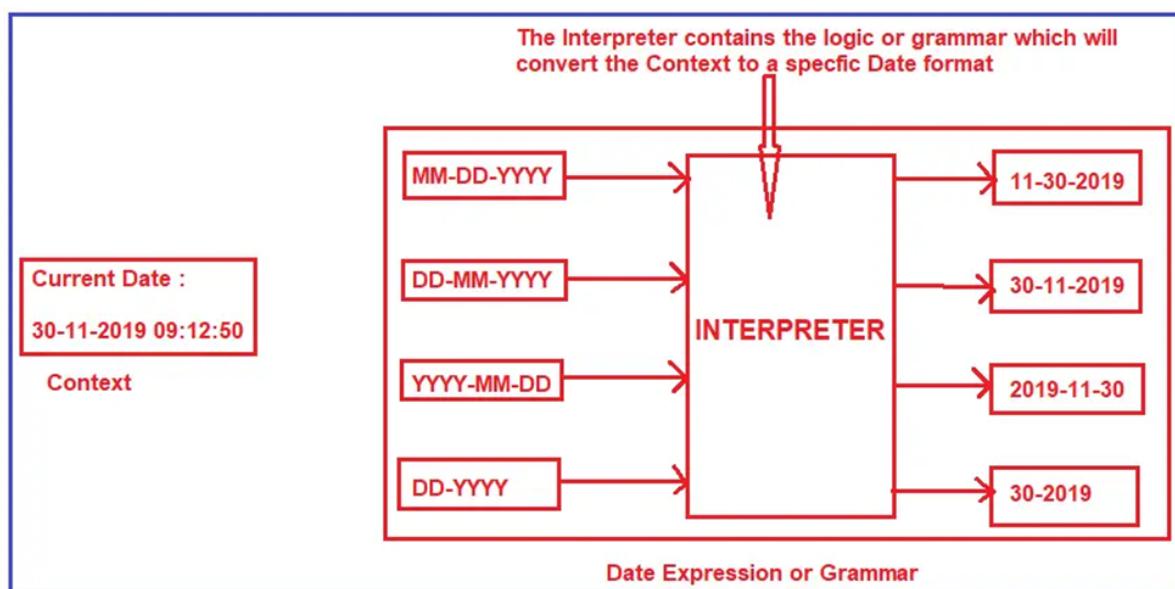
Interpreter (Thông dịch viên) là mẫu thiết kế thuộc nhóm Behavioral Pattern, được sử dụng để xác định biểu diễn ngữ pháp cho một ngôn ngữ và cung cấp trình

thông dịch để xử lý ngữ pháp này. Hay giúp người lập trình có thể “xây dựng” những đối tượng “động” bằng cách đọc mô tả về đối tượng rồi sau đó “xây dựng” đối tượng đúng theo mô tả đó.

### 4.3.2. Motivation

Nếu một loại vấn đề cụ thể xảy ra đủ thường xuyên, thì có thể đáng để diễn đạt các trường hợp của vấn đề dưới dạng các câu bằng một ngôn ngữ đơn giản. Sau đó, bạn có thể xây dựng một trình thông dịch giải quyết vấn đề bằng cách diễn giải các câu này.

Ở phía trái, bạn có thể thấy Context. Context không gì khác ngoài giá trị mà chúng ta muốn diễn giải. Ở đây, giá trị context là ngày hiện tại. Ở phía phải, bạn có thể thấy biểu diễn ngày hoặc bạn có thể nói là cú pháp. Chúng ta có các loại biểu diễn ngày khác nhau như (MM-DD-YYYY, DD-MM-YYYY, YYYY-MM-DD và DD-YYYY).

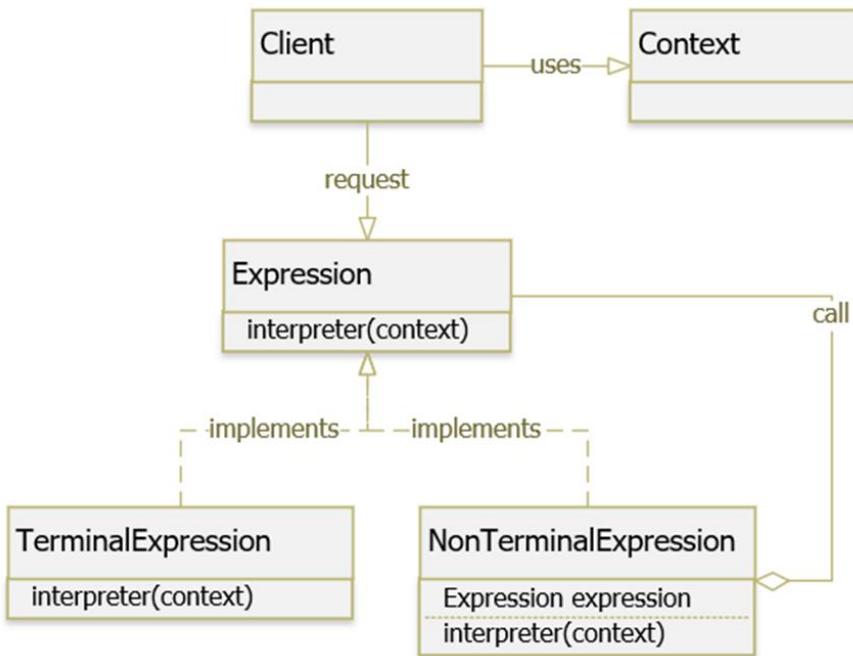


Hình 4.17: Chuyển đổi các định dạng thời gian

Giả sử bạn muốn có ngày trong định dạng MM-DD-YYYY, bạn cần truyền giá trị Context và Biểu diễn Ngày bạn muốn (tức là MM-DD-YYYY) vào trình thông dịch. Trình thông dịch sẽ chuyển đổi giá trị Context thành định dạng ngày mà bạn truyền cho nó. Vì vậy, cơ bản, trình thông dịch chứa logic hoặc ngữ pháp để chuyển đổi đối tượng Context thành một định dạng có thể đọc cụ thể.

### 4.3.3. Đặc điểm

#### 4.3.3.1. Cấu trúc mấu



Hình 4.18: Sơ đồ lớp của mẫu thiết kế Interpreter

#### 4.3.3.2. Thành phần trong cấu trúc mẫu

- Context : là phần chứa thông tin biểu diễn mẫu chúng ta cần xây dựng.
- Expression : là một interface hoặc abstract class, định nghĩa phương thức interpreter chung cho tất cả các node trong cấu trúc cây phân tích ngữ pháp. Expression được biểu diễn như một cấu trúc cây phân cấp, mỗi implement của Expression có thể gọi một node.
- TerminalExpression (biểu thức đầu cuối): cài đặt các phương thức của Expression, là những biểu thức có thể được diễn giải trong một đối tượng duy nhất, chứa các xử lý logic để đưa thông tin của context thành đối tượng cụ thể.
- NonTerminalExpression (biểu thức không đầu cuối): cài đặt các phương thức của Expression, biểu thức này chứa một hoặc nhiều biểu thức khác nhau, mỗi biểu thức có thể là biểu thức đầu cuối hoặc không phải là biểu thức đầu cuối. Khi một phương thức interpret() của lớp biểu thức không phải là đầu cuối được gọi, nó sẽ gọi đệ quy đến tất cả các biểu thức khác mà nó đang giữ.
- Client : đại diện cho người dùng sử dụng lớp Interpreter Pattern. Client sẽ xây dựng cây biểu thức đại diện cho các lệnh được thực thi, gọi phương thức interpreter() của node trên cùng trong cây, có thể truyền context để thực thi tất cả các lệnh trong cây.
- Sự tương tác:

- Khi Client gọi phương thức interpret() trên Expression, quá trình diễn giải bắt đầu. Expression sẽ kiểm tra và so khớp các phần tử trong Context và thực hiện các thay đổi cần thiết trên Context. Sự cộng tác giữa các thành phần này cho phép diễn giải và thực thi các biểu diễn ngôn ngữ phức tạp, từ việc xác định các phần tử cơ bản đến áp dụng các quy tắc và luật lệ phức tạp.

#### **4.3.4. Khả năng ứng dụng**

Sử dụng Interpreter Pattern khi chúng ta muốn:

- Bộ ngữ pháp đơn giản. Pattern này cần xác định ít nhất một lớp cho mỗi quy tắc trong ngữ pháp. Do đó ngữ pháp có chứa nhiều quy tắc có thể khó quản lý và bảo trì.
- Không quan tâm nhiều về hiệu suất. Do bộ ngữ pháp được phân tích trong cấu trúc phân cấp (cây) nên hiệu suất không được đảm bảo.

Interpreter Pattern thường được sử dụng trong trình biên dịch (compiler), định nghĩa các bộ ngữ pháp, rule, trình phân tích SQL, XML, ...

#### **4.3.5. Hệ quả**

##### **4.3.5.1. Ưu điểm**

- Giảm sự phụ thuộc giữa abstraction và implementation (loose coupling).
- Giảm số lượng những lớp con không cần thiết.
- Code sẽ gọn gàng hơn và kích thước ứng dụng sẽ nhỏ hơn.
- Dễ bảo trì hơn.
- Dễ dàng mở rộng về sau.
- Cho phép ẩn các chi tiết implement từ client.

##### **4.3.5.2. Nhược điểm**

- Ngôn ngữ đặc tả được xây dựng đòi hỏi có cấu trúc ngữ pháp đơn giản.
- Hiệu suất không đảm bảo

#### **4.3.6. Các mẫu thiết kế liên quan**

- Composite Pattern: Mẫu Composite cho phép xây dựng cấu trúc cây thành phần để biểu diễn các biểu thức phức tạp. Kết hợp mô hình Composite với mô hình Interpreter, chúng ta có thể xây dựng cây biểu diễn một biểu thức ngữ pháp hoặc một ngôn ngữ và sử dụng Interpreter pattern để diễn giải và xử lý các phần tử trong cây.

- Flyweight Pattern: Mẫu Flyweight cho phép chia sẻ chung các đối tượng nhẹ để tiết kiệm bộ nhớ. Khi áp dụng mô hình Interpreter, một số thành phần của biểu thức có thể được chia sẻ giữa các biểu thức khác nhau để tối ưu hóa bộ nhớ.
- Iterator Pattern: Mẫu Iterator cung cấp một cách duyệt qua các phần tử của một tập hợp mà không cần tiết lộ cấu trúc bên trong. Trong ngữ cảnh của mô hình Interpreter, Iterator pattern có thể được sử dụng để lặp qua các phần tử của một biểu thức ngữ pháp hoặc cây cú pháp để thực hiện các hoạt động như tính toán, kiểm tra và xử lý.
- Visitor Pattern: Mẫu Visitor cho phép thêm các thao tác mới cho các đối tượng mà không cần thay đổi cấu trúc. Trong ngữ cảnh của mô hình Interpreter, Visitor pattern có thể được sử dụng để thực hiện các thao tác khác nhau trên các phần tử của một biểu thức ngữ pháp hoặc cây cú pháp, như in ra, tính giá trị, tối ưu hóa,...

#### 4.4. Mediator

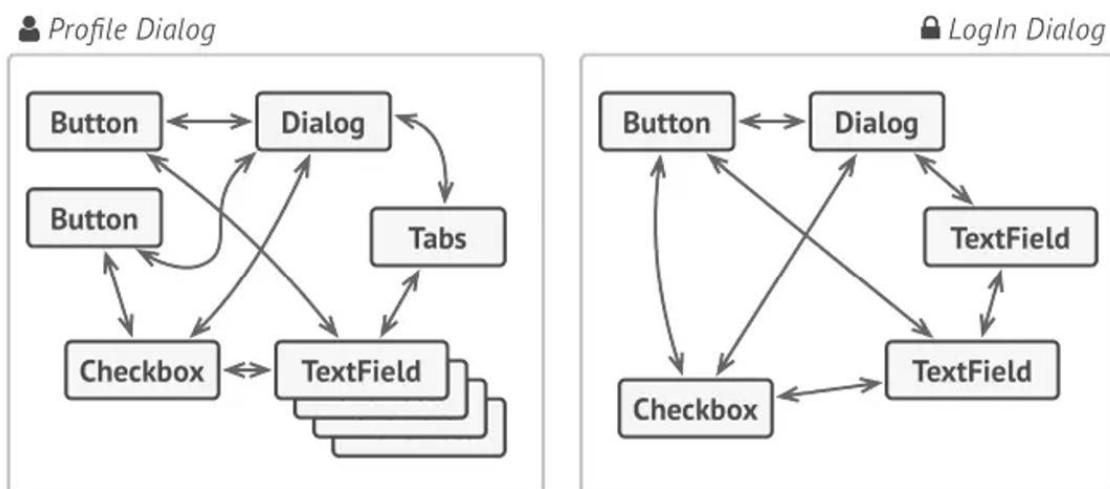
##### 4.4.1. Tổng quan

Mediator Pattern được dùng để giảm sự phức tạp trong “giao tiếp” giữa các lớp và các đối tượng. Mô hình này cung cấp một lớp trung gian có nhiệm vụ xử lý thông tin liên lạc giữa các tầng lớp, hỗ trợ bảo trì mã code dễ dàng bằng cách khớp nối lỏng lẻo.

Mediator Pattern hoạt động như cầu nối thúc đẩy mối quan hệ nhiều – nhiều giữa các đối tượng tương ứng với nhau để đạt đến được kết quả mong muốn.

##### 4.4.2. Motivation

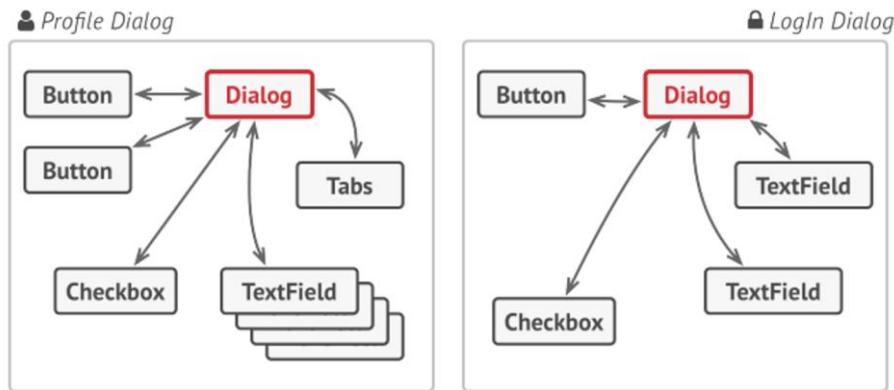
Giả sử bạn có một cái dialog để tạo và chỉnh sửa thông tin khách hàng. Nó gồm nhiều thành phần như text fields, buttons, checkboxes, ...



Hình 4.19 Sơ đồ quan hệ giữa các component khi không dùng Mediator

Một vài thành phần sẽ tương tác với vài thành phần khác. Ví dụ chọn checkbox thì sẽ hiện ra text field bị ẩn để nhập vào số lượng con của người dùng.

Nếu triển khai những logic này trực tiếp vào từng thành phần, bạn sẽ làm cho các thành phần này khó tái sử dụng hơn.



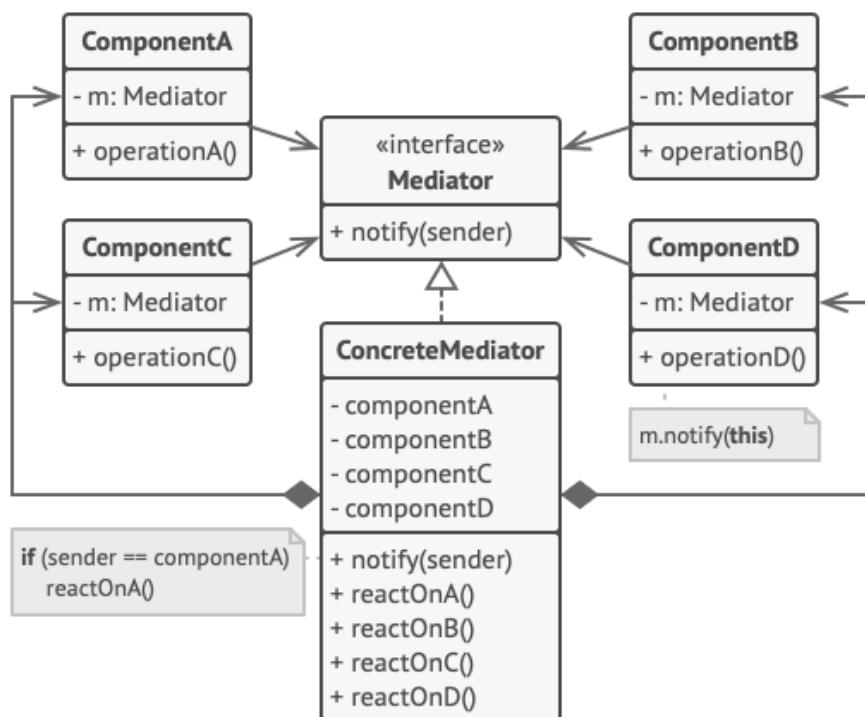
Hình 4.20 Sơ đồ quan hệ giữa các component khi dùng Mediator

Để các thành phần này sẽ giao tiếp gián tiếp bằng cách gọi một đối tượng Mediator đặc biệt để đối tượng này chuyển lời gọi đến các thành phần thích hợp.

Các thành phần lúc này sẽ chỉ phụ thuộc vào một lớp Mediator duy nhất thay vì phải kết nối với rất nhiều thành phần khác như ban đầu.

#### 4.4.3. Đặc điểm

##### 4.4.3.1. Cấu trúc mẫu



Hình 4.21: Sơ đồ lớp của mẫu thiết kế Mediator

#### **4.4.3.2. Thành phần trong cấu trúc mẫu**

- Các Component là các lớp khác nhau có chứa vài logic nghiệp vụ như Button, TextField,... Mỗi component đều có một tham chiếu đến một Mediator, được khai báo với kiểu là Mediator interface. Component không quan tâm đến các lớp thật sự của Mediator. Vì vậy, có thể tái sử dụng component ở các chương trình khác và chỉ việc liên kết nó với một mediator khác.
- Mediator interface khai báo các phương thức để giao tiếp với các component, thường chỉ bao gồm một phương thức thông báo duy nhất. Component có thể truyền bất kỳ ngữ cảnh nào làm các đối số của phương thức này, bao gồm cả các đối tượng của chúng, nhưng chỉ theo cách không xảy ra sự ghép nối nào giữa thành phần nhận và lớp gửi.
- Concrete Mediator đóng gói các mối quan hệ giữa các component khác nhau. Các Concrete mediator thường giữ các tham chiếu đến tất cả component mà chúng quản lý và thường thậm chí quản lý cả vòng đời.
- Các component không cần quan tâm đến các component khác. Nếu có điều gì xảy ra với component thì chúng chỉ cần thông báo đến mediator. Khi mediator nhận thông báo, nó có thể dễ dàng xác định nơi gửi (điều này có thể vừa đủ để quyết định xem component nào nên được kích hoạt).

#### **4.4.4. Khả năng ứng dụng**

- Sử dụng khi khó thay đổi một vài lớp vì chúng đã được kết nối chặt chẽ với rất nhiều lớp khác.
- Sử dụng khi không thể tái sử dụng một component ở các chương trình khác vì chúng quá phụ thuộc vào các component khác.
- Sử dụng khi cảm thấy mình đang tạo ra rất nhiều lớp con component chỉ để tái sử dụng một vài hành vi đơn giản ở các ngữ cảnh khác nhau.
- Sử dụng khi tập hợp các đối tượng giao tiếp theo những cách thức được xác định rõ ràng nhưng cách thức đó quá phức tạp. Sự phụ thuộc lẫn nhau giữa các đối tượng tạo ra kết quả là cách tổ chức không có cấu trúc và khó hiểu.
- Sử dụng khi cần tái sử dụng một đối tượng nhưng rất khó khăn vì nó tham chiếu và giao tiếp với nhiều đối tượng khác.
- Sử dụng khi điều chỉnh hành vi giữa các lớp một cách dễ dàng, không cần chỉnh sửa ở nhiều lớp.

- Thường được sử dụng trong các hệ thống truyền thông điệp (message-based system), chẳng hạn như hệ thống chat.
- Khi giao tiếp giữa các object trong hệ thống quá phức tạp, có quá nhiều quan hệ giữa các object trong hệ thống. Một điểm chung để kiểm soát hoặc giao tiếp là cần thiết.

#### **4.4.5. Hệ quả**

##### **4.4.5.1. Ưu điểm**

- Đảm bảo nguyên tắc Single Responsibility Principle (SRP): chúng ta có thể trích xuất sự liên lạc giữa các component khác nhau vào trong một nơi duy nhất, làm cho nó được bảo trì dễ dàng hơn.
- Đảm bảo nguyên tắc Open/Closed Principle (OCP): chúng ta có thể tạo ra các mediator mới mà không cần thay đổi các component.
- Giảm thiểu việc gắn kết giữa các component khác nhau trong một chương trình.
- Tái sử dụng các component đơn giản hơn.
- Đơn giản hóa cách giao tiếp giữa các đối tượng, Một Mediator sẽ thay thế mối quan hệ nhiều-nhiều (many-to-many) giữa các component bằng quan hệ một-nhiều (one-to-many) giữa một mediator với các component.
- Quản lý tập trung, giúp làm rõ các component tương tác trong hệ thống như thế nào trong hệ thống

##### **4.4.5.2. Nhược điểm**

- Qua thời gian thì Mediator có thể trở thành God object.

#### **4.4.6. Các mẫu thiết kế liên quan**

- Chain of Responsibility, Command, Mediator và Observer là các cách khác nhau để giải quyết vấn đề kết nối giữa các receiver và các sender.
  - Chain of Responsibility truyền request tuần tự dọc theo một chuỗi động chứa các receiver tiềm năng cho đến khi có receiver thích hợp có thể giải quyết được. Command thì tạo ra các kết nối một chiều giữa các receiver và các sender.
  - Mediator loại bỏ các kết nối trực tiếp giữa các receiver và các sender rồi bắt buộc chúng phải giao tiếp không trực tiếp thông qua đối tượng mediator.

- Observer cho phép các receiver chủ động trong việc subscribe và unsubscribe receiving requests.
- Facade và Mediator có các công việc giống nhau là đều có gắt gỏng tổ chức sự hợp tác giữa nhiều lớp có gắn kết chặt chẽ với nhau.
  - Facade thì định nghĩa một interface được đơn giản hóa đến các đối tượng của hệ thống con nhưng nó không tạo thêm các chức năng mới. Hệ thống con bản thân nó không quan tâm đến Facade. Các đối tượng trong hệ thống con có thể giao tiếp trực tiếp với nhau.
  - Mediator thì sẽ trung gian hóa sự giao tiếp giữa các component trong hệ thống. Component chỉ biết về đối tượng mediator và không giao tiếp trực tiếp với các component khác.

#### 4.5. Memento



Hình 4.22 Minh họa mẫu thiết kế Memento

##### 4.5.1. Tổng quan

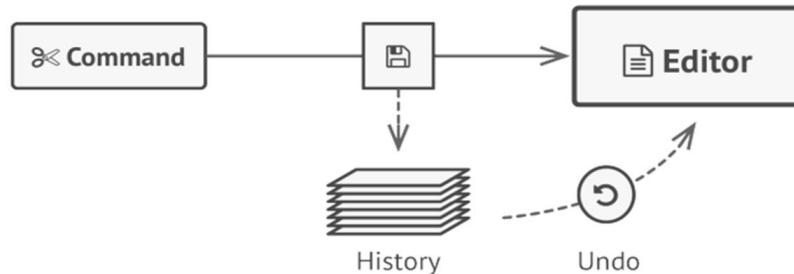
Memento (tên gọi khác là Token) là mẫu thiết kế thuộc nhóm Behavioral Pattern, cho phép người lưu trữ và hồi phục các phiên bản cũ của 1 object mà không can thiệp vào nội dung của object đó.

##### 4.5.2. Motivation

Đặt vấn đề:

- Tưởng tượng bạn đang tạo 1 text editor. Bao gồm các chức năng như chỉnh sửa text, format text, thêm ảnh, v.v...

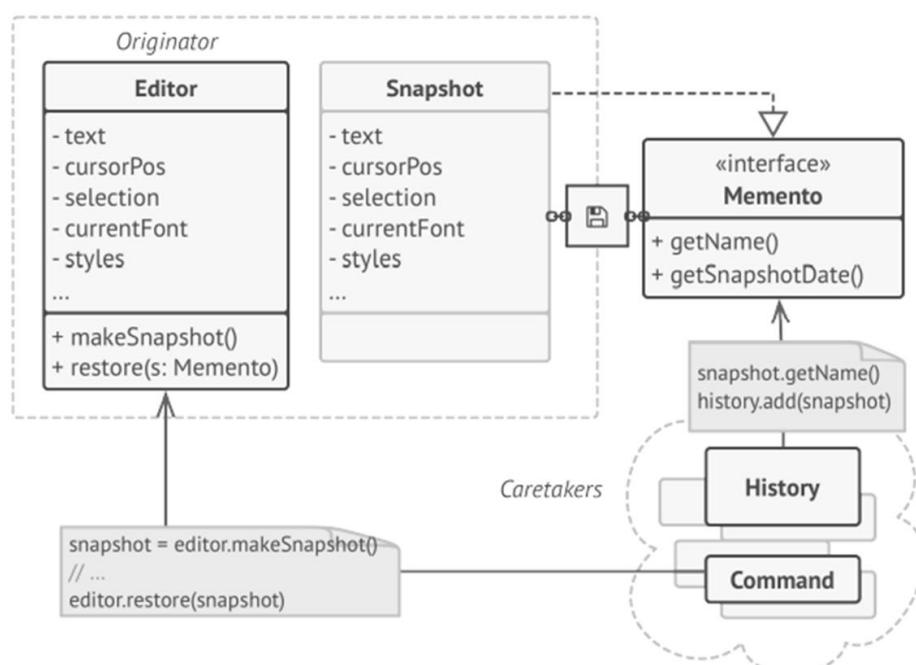
- Để phát triển thêm app, bạn quyết định cho phép người dùng undo và redo bất kỳ thao tác nào thực hiện trên tệp văn bản. Bằng cách trước khi thực hiện bất kì thao tác nào, app sẽ lưu state tất cả object vào trong một storage (take snapshot, lưu vào history). Sau đó, khi user cần undo 1 thao tác, app lấy state đã được lưu trước đó trong storage và dùng nó để restore state của tất cả object.



Hình 4.23: Mô tả về việc lưu lại bản chụp của đối tượng

- Nhưng cái khó ở đây là làm sao để thật sự take snapshot? Bạn phải cần duyệt qua tất cả field của object để lưu nó vào storage. Tuy nhiên, việc đó là không khả thi vì thực tế hầu hết các objects thường giàu phần lớn data trong các trường private.
- Có vẻ như chỉ để take snapshot, ta đã đưa app vào một tình thế rất gian nan: ta public tất cả các private fields của editor object khiến nó trở nên mong manh và tạo ra 1 class chuyên để copy editor object luôn phải thay đổi mỗi khi editor object thay đổi. Vậy còn cách nào khác để triển khai undo redo không?

Giải pháp:

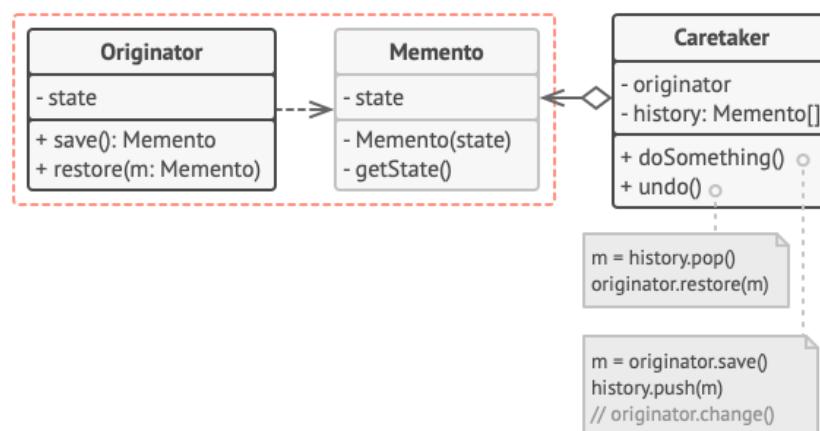


Hình 4.24 Mô tả về tương tác giữa các thành phần trong Memento

- Memento pattern giao việc tạo ra snapshot cho chính chủ nhân của state đó (originator object). Chính object đó sẽ dễ dàng tạo ra snapshot vì nó có toàn quyền truy cập state của nó.
- Memento gợi ý ta nên lưu state được copy từ object vào một object gọi là memento. Content của memento object không được truy cập từ các object khác ngoại trừ originator object. Các object khác phải giao tiếp với memento thông qua interface bị giới hạn chỉ cho phép lấy metadata của snapshot (metadata - những data về data chứ không phải là data: ngày tạo, tên action, v.v.)...

#### 4.5.3. Đặc điểm

##### 4.5.3.1. Cấu trúc mẫu



Hình 4.25 Cấu trúc mẫu của Memento

##### 4.5.3.2. Thành phần trong cấu trúc mẫu

- Originator: có khả năng tạo một snapshot (lưu) và khôi phục trạng thái của nó từ snapshot khi cần.
- Memento: là một object tương ứng với một snapshot trạng thái của Originator. Memento thường được làm cho bất biến, chỉ truyền dữ liệu một lần thông qua hàm khởi tạo.
- Caretaker: không chỉ biết khi nào và tại sao để nắm bắt trạng thái của Originator mà còn biết khi nào trạng thái nên được khôi phục. Caretaker có thể theo dõi lịch sử của Originator bằng cách lưu trữ một stack chứa các Memento.

#### 4.5.4. Khả năng ứng dụng

- Các ứng dụng cần chức năng cần Undo/ Redo: lưu trạng thái của một đối tượng bên ngoài và có thể restore/ rollback sau này.
- Thích hợp với các ứng dụng cần quản lý transaction.

#### **4.5.5. Hệ quả**

##### **4.5.5.1. Ưu điểm**

- Bảo bảo nguyên tắc đóng gói: sử dụng trực tiếp trạng thái của đối tượng có thể làm lộ thông tin chi tiết bên trong đối tượng và vi phạm nguyên tắc đóng gói.
- Đơn giản code của Originator bằng cách để Memento lưu giữ trạng thái của Originator và Caretaker quản lý lịch sử thay đổi của Originator.

##### **4.5.5.2. Nhược điểm**

- App tiêu thụ nhiều RAM và xử lý nếu clients tạo mementos quá thường xuyên.
- Caretaker phải theo dõi vòng đời của originator để hủy memento không dùng nữa.
- Hầu hết các ngôn ngữ hiện đại, hay cụ thể hơn là dynamic programming languages, ví dụ như PHP, Python và Javascript, không thể đảm bảo state bên trong memento được giữ không ai dụng tới.

#### **4.5.6. Các mẫu thiết kế liên quan**

- Có thể sử dụng Command và Memento cùng nhau khi thực hiện “hoàn tác”. Trong trường hợp này, các lệnh chịu trách nhiệm thực hiện các hoạt động khác nhau trên một đối tượng đích, trong khi các Memento lưu trạng thái của đối tượng đó ngay trước khi lệnh được thực thi.
- Có thể sử dụng Memento cùng với Iterator để nắm bắt trạng thái lặp lại hiện tại và khôi phục nó nếu cần.
- Đôi khi Prototype là giải pháp thay thế đơn giản hơn cho Memento. Điều này hoạt động nếu đối tượng, trạng thái mà bạn muốn lưu trữ trong lịch sử, khá đơn giản và không có liên kết đến tài nguyên bên ngoài hoặc các liên kết dễ thiết lập lại.

### **4.6. Observer**

#### **4.6.1. Tổng quan**

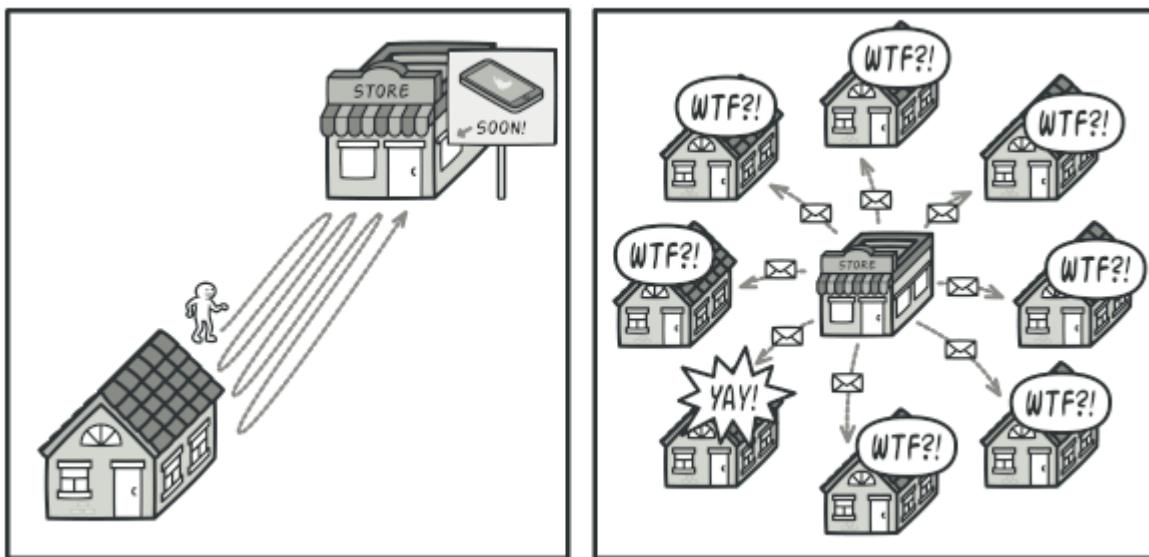
Observer (Event-Subscriber, Listener) là mẫu thiết kế thuộc nhóm Hành vi. Nó định nghĩa mối quan hệ phụ thuộc một – nhiều giữa các đối tượng, mà khi một đối tượng publisher có sự thay đổi trạng thái, thì tất cả các thành phần phụ thuộc vào publisher này subscriber sẽ được thông báo và cập nhật một cách tự động.

#### **4.6.2. Motivation**

Giả sử chúng ta có hai loại đối tượng: Customer và Store. Customer đặc biệt hứng thú với một hàng sản phẩm nào đó và luôn muốn mua ngay sản phẩm mới của hàng đó ngay khi nó vừa ra mắt. Để thỏa mãn nhu cầu của bản thân, họ dành một khoảng thời

gian nhở mỗi ngày chạy ra Store để kiểm tra xem Store đã có sản phẩm mới hay chưa. Chắc chắn sẽ có rất nhiều lần Customer hụt hẫng vì đi ra Store mà chả được ích lợi gì. Thế là Store nghĩ ra một giải pháp tình thế, chính là gửi thông báo (bằng một phương tiện nào đó như thư, mail, gọi điện,...) cho tất cả các Customer. Một vấn đề mới phát sinh, đó là không phải ai cũng muốn nhận thông báo. Những người phải nhận quá nhiều thông báo sẽ cảm thấy phiền nhiễu, khó chịu.

Ngoài trường hợp kể trên, chúng ta còn dễ dàng phát hiện ra nhiều bài toán tương tự, đặc biệt trong các ứng dụng mạng xã hội.



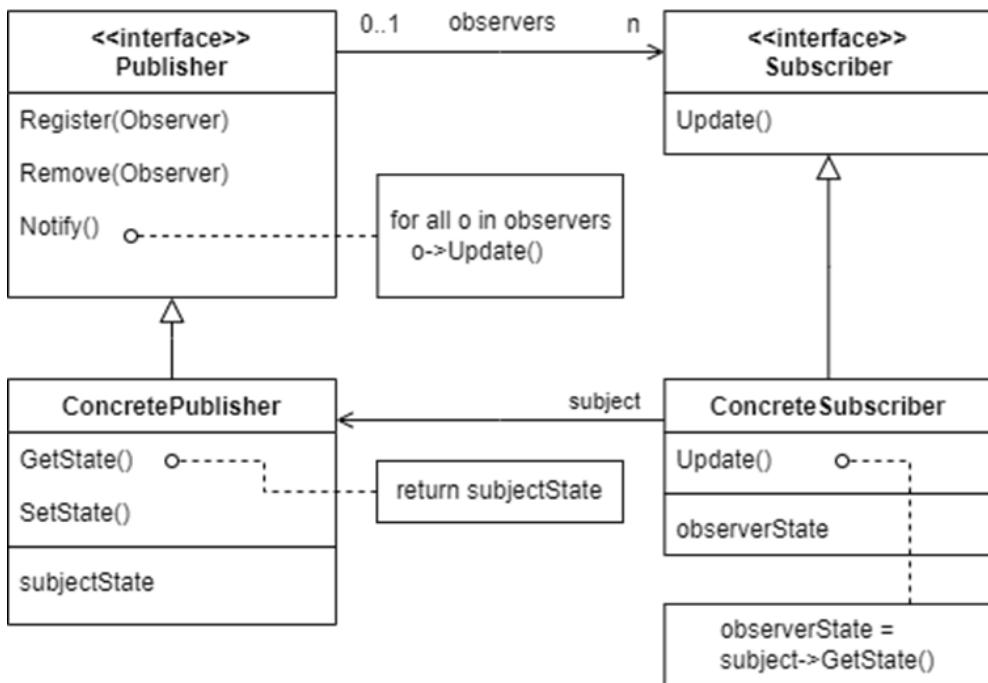
Hình 4.26 Minh họa bài toán thực tế của Observer

Mẫu chốt ở đây là, thay vì “tất cả”, ta chỉ thông báo đến một “bộ phận” mà thôi.

Thế nhưng, việc hiện thực hóa ý tưởng đó trong lập trình đòi hỏi ta phải tư duy nhiều hơn trong phân tích và thiết kế. Trong một phương thức Business của publisher mà ta muốn bảo rằng “Êh! Publisher thay đổi rồi nè, subscriber cập nhật theo đi!”, thay vì “tất cả”, tức là hành vi static của class Publisher, ta bỏ static đi và chỉ bắt một số object cụ thể gọi phương thức Update thôi. Nhưng những object cụ thể đó ở đâu ra? Truyền tham số vào phương thức Business dưới dạng một container chỉ hợp lý khi số lần cần cập nhật chỉ là một. Nếu không, ta sẽ phải thêm lại các object vào container mỗi lần thay đổi trạng thái. Ta khai báo như một thuộc tính của class Publisher được không? Cũng được, tuy nhiên không phải object Subscriber nào cũng theo Publisher đó đến cuối đời. Đây chính là lúc mẫu thiết kế Observer thể hiện hiệu quả.

#### 4.6.3. Đặc điểm

#### 4.6.3.1. Cấu trúc mẫu



Hình 4.27 Cấu trúc mẫu của Observer

#### 4.6.3.2. Thành phần trong cấu trúc mẫu

- Publisher: Cung cấp khả năng thêm, loại bỏ subscriber và thông báo đến subscriber.
- ConcretePublisher: Nơi trạng thái thay đổi và cần gửi thông báo đến các subscriber.
- Subscriber: Được thông báo về sự thay đổi trạng thái của publisher.
- ConcreteSubscriber: Nhận thông báo từ publisher, lưu trữ trạng thái của publisher và thực thi cập nhật tương ứng.

#### 4.6.4. Khả năng ứng dụng

- Sự thay đổi trạng thái của một đối tượng ảnh hưởng đến những đối tượng khác, và tập hợp những đối tượng đó không được biết trước hoặc biến động.
- Những subscriber chỉ cần theo dõi trong một khoảng thời gian nhất định hoặc trong những trường hợp đặc biệt.

#### 4.6.5. Hệ quả

##### 4.6.5.1. Ưu điểm

- Đảm bảo nguyên tắc Đóng/Mở (Open/Closed Principle): Thoải mái tạo thêm các class subscriber khác mà không cần thay đổi code của publisher.
- Tạo mối liên hệ giữa các đối tượng trong khi chạy chương trình.

#### **4.6.5.2. Nhược điểm**

- Các subscriber được thông báo theo thứ tự ngẫu nhiên.

#### **4.6.6. Các mẫu thiết kế liên quan**

- Chain of Responsibility, Command, Mediator và Observer giải quyết những bài toán khác nhau liên quan đến request để kết nối sender và receiver.
- Chain of Responsibility truyền một request tuần tự qua một chuỗi động gồm các receiver cho đến khi một trong số chúng giải quyết được request đó.
- Command thiết lập kết nối một chiều giữa sender và receiver.
- Mediator triệt tiêu những kết nối trực tiếp giữa sender và receiver, buộc chúng phải giao tiếp với nhau gián tiếp thông qua một đối tượng trung gian, gọi là mediator.
- Observer để receiver linh động đăng ký hoặc hủy đăng ký việc nhận request.
- Khá khó để phân biệt cách triển khai giữa Mediator và Observer, bởi Mediator có một cách triển khai phổ biến tương đồng với Observer.

### **4.7. State**

#### **4.7.1. Tổng quan**

State cho phép đối tượng thay đổi hành vi của mình khi trạng thái nội tại thay đổi. Nó giúp tách biệt logic xử lý và trạng thái của một đối tượng, giúp dễ dàng thêm mới các trạng thái và xử lý chúng.

#### **4.7.2. Motivation**

Mẫu State có liên quan mật thiết đến khái niệm Finite-State Machine (máy trang thái hữu hạn). Ý tưởng chính ở đây chính là khi một chương trình hoạt động, tại bất kỳ thời điểm nào, số lượng trạng thái mà chương trình đó có thể có là một con số hữu hạn. Và chương trình đó có thể chuyển đổi qua lại giữa các trạng thái ngay lập tức. Tuy nhiên, có một số trạng thái cần phải có điều kiện hoặc hoàn toàn không thể chuyển đổi qua lại với nhau. Các điều kiện, quy tắc để chuyển đổi trạng thái này được gọi là Transitions, được định nghĩa sẵn trong chương trình. Cách tiếp cận này còn có thể được áp dụng cho các đối tượng. Giả sử như một lớp “Document”. Trong đó, một Document có thể nằm trong 1 trong 3 trạng thái: Draft (bản nháp), Moderation (Bản kiểm tra), và Published (bản đã xuất bản). Khi đó, phương thức Publish của đối tượng Document sẽ hoạt động một cách khác nhau ở từng trạng thái.

- Trong Draft, nó chuyển Document sang trạng thái Moderation.

- Trong Moderation nó chuyển trạng thái của Document sang Published khi và chỉ khi người dùng hiện tại là admin.
- Trong Published, phương thức này không làm gì cả.

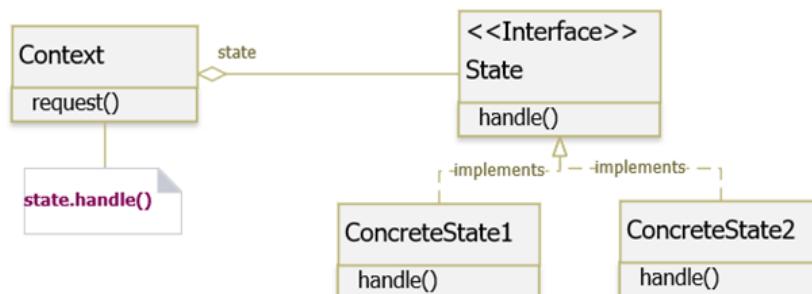


Hình 4.28 Mô hình trạng thái của tài liệu

Máy trạng thái thường được thực hiện với rất nhiều câu lệnh điều kiện như if hay switch để giúp bạn chọn các hành vi phù hợp ở trong từng trạng thái khác nhau.

#### 4.7.3. Đặc điểm

##### 4.7.3.1. Cấu trúc mẫu



Hình 4.29 Cấu trúc mẫu của State

##### 4.7.3.2. Thành phần trong cấu trúc mẫu

- Context: người dùng không truy cập trực tiếp vào State của object, Context chứa thông tin của ConcreteState cho hành vi tương ứng với state đang được thực hiện.
- State: xác định đặc tính cơ bản của tất cả ConcreteState.
- ConcreteState: cài đặt phương thức của State tương ứng với từng state xác định.

#### 4.7.4. Khả năng ứng dụng

- Muốn thay đổi hành vi của đối tượng theo trạng thái trong khi chạy chương trình.
- Có nhiều điều kiện phức tạp buộc đối tượng phụ thuộc theo trạng thái của nó.

#### **4.7.5. Hệ quả**

##### **4.7.5.1. Ưu điểm**

- Single Responsibility Principle: tách biệt mỗi state tương ứng với 1 class riêng biệt.
- Open/Closed Principle: có thể thêm state mới mà không ảnh hưởng các state đã có.

##### **4.7.5.2. Nhược điểm**

- Việc áp dụng mẫu có thể trở nên không cần thiết nếu máy trạng thái chỉ có một số trạng thái hoặc ít khi thay đổi trạng thái.

#### **4.7.6. Các mẫu thiết kế liên quan**

- State, Bridge và Strategy có cấu trúc khá giống nhau. Tuy nhiên, chúng có mục đích khác nhau. State thì làm nhiều việc cho một object, Strategy thì làm một việc bằng nhiều cách và Bridge thì làm việc độc lập cho một object.
- State có thể được xem là một phiên bản mở rộng của Strategy. Tuy nhiên, Strategy làm cho các object hoàn toàn độc lập với nhau và không biết đến sự tồn tại của nhau. Trong khi State thì các object State có thể giao tiếp và phụ thuộc vào nhau.

### **4.8. Strategy**



Hình 4.30 Minh họa mẫu thiết kế Strategy

#### **4.8.1. Tổng quan**

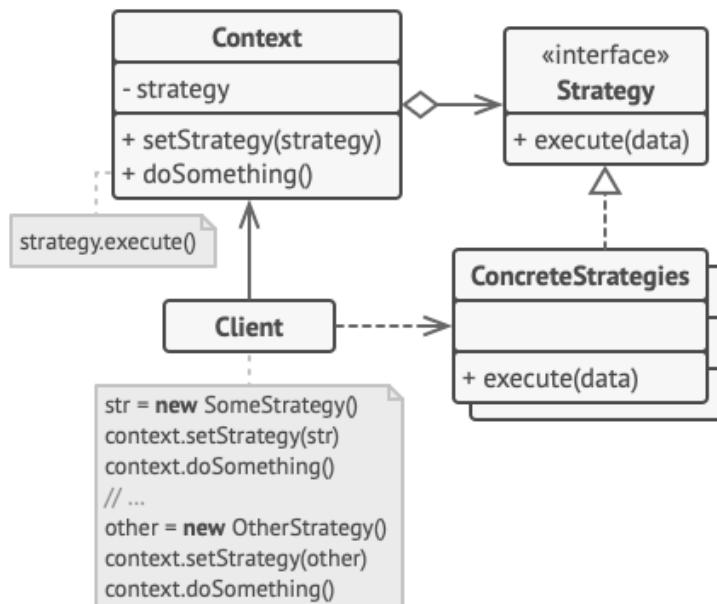
Strategy là một mẫu thiết kế thuộc nhóm Behavioral Pattern định nghĩa một tập hợp các thuật toán giống nhau, đóng gói chúng và khiến chúng có thể thay thế cho nhau. Strategy làm cho phần thuật toán độc lập khỏi client sử dụng nó.

#### 4.8.2. Motivation

Có những tình huống phổ biến khi các lớp chỉ khác nhau về hành vi của chúng. Đối với trường hợp này, nên tách các thuật toán trong các lớp riêng biệt để có khả năng chọn các thuật toán khác nhau trong thời gian chạy.

#### 4.8.3. Đặc điểm

##### 4.8.3.1. Cấu trúc mẫu



Hình 4.31: Sơ đồ lớp của mẫu thiết kế Strategy

##### 4.8.3.2. Thành phần trong cấu trúc mẫu

- Thành phần:

- **Strategy:** Cung cấp một Interface chung cho tất cả các thuật toán được hỗ trợ. Nó khai báo một phương thức để Context sử dụng thực hiện một Strategy. Context sử dụng giao diện này để gọi thuật toán được xác định bởi ConcreteStrategy.
- **ConcreteStrategy:** Triển khai các thuật toán khác nhau cho Context sử dụng.
- **Context:** Duy trì một tham chiếu đến một ConcreteStrategy và chỉ giao tiếp với đối tượng này thông qua giao diện Strategy. Context gọi phương thức trên đối tượng Strategy được liên kết mỗi khi nó cần chạy thuật toán.
- **Client:** Có trách nhiệm tạo ra các đối tượng Strategy cụ thể và truyền đối tượng đó vào Context. Context hiển thị một trình thiết lập cho phép Client thay thế Strategy được liên kết với Context khi chạy.

- **Sự cộng tác:**

- Các đối tượng Context chứa một tham chiếu đến ConcreteStrategy nên được sử dụng. Khi một thao tác được yêu cầu thì thuật toán sẽ được chạy từ đối tượng Strategy. Context không nhận thức được việc thực hiện chiến lược và thuật toán được thực thi như thế nào. Nếu cần, các đối tượng bổ sung có thể được xác định để truyền dữ liệu từ đối tượng Context sang Strategy. Đối tượng Context nhận các yêu cầu từ máy khách và ủy thác chúng cho đối tượng chiến lược. Thông thường ConcreteStrategy được tạo bởi Client và được chuyển đến Context. Từ thời điểm này, khách hàng chỉ tương tác với Context.

#### **4.8.4. Khả năng ứng dụng**

- Muốn sử dụng các biến thể khác nhau của một xử lý trong một đối tượng và có thể chuyển đổi giữa các xử lý trong thời gian chạy.
- Khi có nhiều lớp tương đương chỉ khác cách chúng thực thi một vài hành vi.
- Khi muốn tách biệt business logic của một lớp khỏi implementation details của các xử lý.
- Khi lớp có toán tử điều kiện lớn chuyển đổi giữa các biến thể của cùng một xử lý.

#### **4.8.5. Hệ quả**

##### **4.8.5.1. Ưu điểm**

- Có thể thay thế các thuật toán linh hoạt với nhau trong một đối tượng khi chạy.
- Tách biệt phần thuật toán khỏi phần sử dụng thuật toán.
- Có thể thay thế việc kế thừa bằng việc đóng gói thuật toán.
- Tăng tính open-closed: Khi thay đổi thuật toán hoặc khi thêm mới thuật toán, không cần thay đổi code phần Context.

##### **4.8.5.2. Nhược điểm**

- Không nên áp dụng nếu thuật toán chỉ có một vài xử lý và hiếm khi thay đổi.
- Client phải nhận biết được sự khác biệt giữa các Strategy để chọn một Strategy phù hợp.

#### **4.8.6. Các mẫu thiết kế liên quan**

- Bridge: Cả hai mẫu đều có sơ đồ UML giống nhau. Nhưng chúng khác nhau về mục đích vì Strategy liên quan đến hành vi và Bridge dành cho cấu trúc. Hơn nữa,

sự kết hợp giữa Context và các Strategy chặt chẽ hơn so với sự kết hợp giữa Abstraction và Implementation trong mô hình cầu nối.

- Command: Khá giống nhau khi đều tham số hoá một đối tượng với một vài hành động tuy nhiên chúng 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, lưu trữ lịch sử của các lệnh, gửi lệnh đến các dịch vụ từ xa, ...
  - 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 Context duy nhất.
- Decorator: Decorator cho phép bạn thay đổi lớp vỏ của một đối tượng, trong khi Strategy cho phép bạn thay đổi nội dung.
- State: Có thể coi như một phần mở rộng của Strategy, đều dựa trên composition. Strategy làm cho các đối tượng hoàn toàn độc lập và không biết gì về nhau. Tuy nhiên, State không hạn chế sự phụ thuộc giữa các Concrete State, cho phép chúng thay đổi trạng thái của Context theo ý muốn.

## 4.9. Template Method

### 4.9.1. Tổng quan

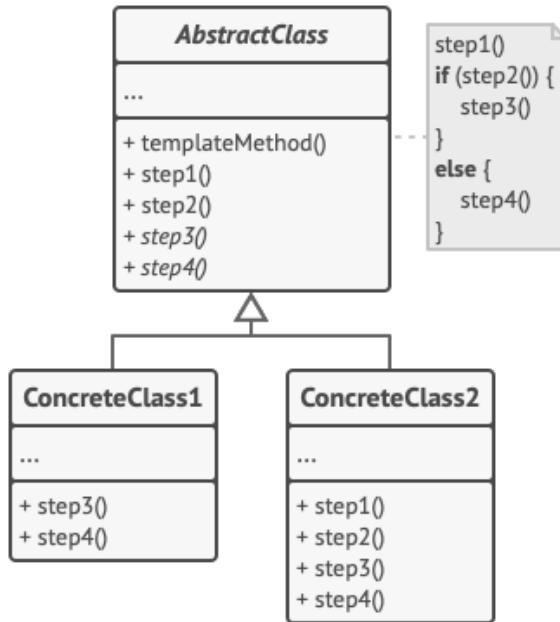
Template Method là mẫu thiết kế thuộc nhóm Behavioral Pattern xây dựng một bộ khung thuật toán trong một toán tử, để lại việc định nghĩa một vài bước cho các subclass mà không làm thay đổi cấu trúc chung của thuật toán.

### 4.9.2. Motivation

Nếu chúng ta xem định nghĩa từ điển về mẫu, chúng ta có thể thấy rằng mẫu là định dạng đặt trước, được sử dụng làm điểm bắt đầu cho một ứng dụng cụ thể để không phải tạo lại định dạng mỗi khi sử dụng. Trên cùng một ý tưởng là phương pháp mẫu dựa trên. Một Template Method xác định một thuật toán trong một lớp cơ sở bằng cách sử dụng các thao tác trừu tượng mà các lớp con ghi đè để cung cấp hành vi cụ thể.

### 4.9.3. Đặc điểm

#### 4.9.3.1. Cấu trúc mẫu



Hình 4.32: Sơ đồ lớp của mẫu thiết kế Template Method

#### 4.9.3.2. Thành phần trong cấu trúc mẫu

- Thành phần:
  - **AbstractClass:** Lớp khai báo các phương thức hoạt động như các bước của thuật toán, cũng như Template Method thực tế gọi các phương thức này theo một thứ tự cụ thể. Các bước có thể được khai báo trừu tượng hoặc có một số triển khai mặc định.
  - **ConcreteClass:** Các ConcreteClass có thể ghi đè tất cả các bước, nhưng không phải bắn thân Template Method.
- Sự cộng tác:
  - ConcreteClass dựa vào AbstractClass để thực hiện các bước bắt biên của thuật toán.

#### 4.9.4. Khả năng ứng dụng

- 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).

#### **4.9.5. Hệ quả**

##### **4.9.5.1. Ưu điểm**

- Tái sử dụng code, tránh trùng lặp code: Đưa những phần trùng lặp vào lớp cha (abstract class).
- Cho phép người dùng override chỉ một số phần nhất định của thuật toán lớn, làm cho chúng ít bị ảnh hưởng hơn bởi những thay đổi xảy ra với các phần khác của thuật toán.

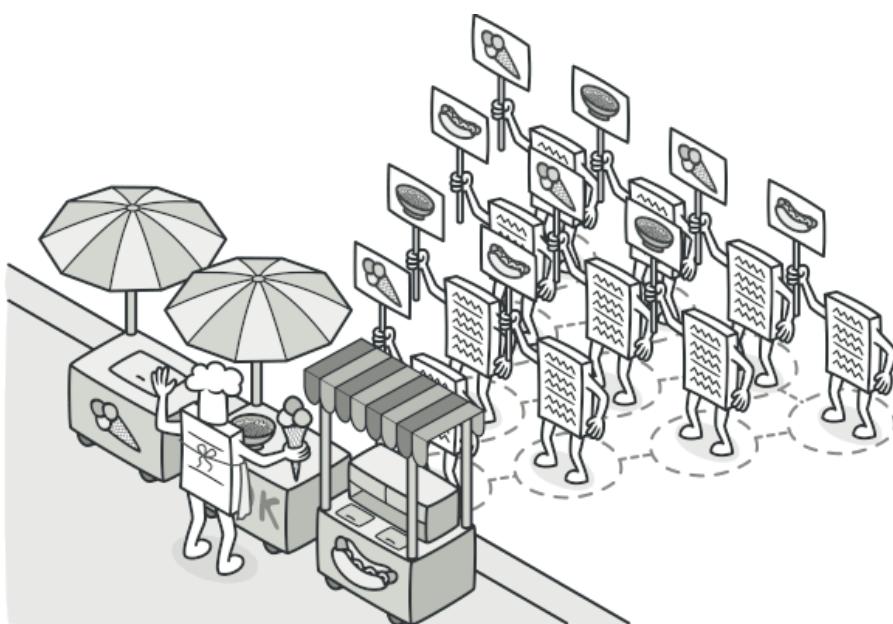
##### **4.9.5.2. Nhược điểm**

- Một số client có thể bị giới hạn bởi khung thuật toán được cung cấp.
- Template Method có càng nhiều bước để override càng khó bảo trì.

#### **4.9.6. Các mẫu thiết kế liên quan**

- Factory Method: Là một chuyên ngành của Template Method, có thể đóng vai trò như một bước trong Template Method.
- Template Method dựa trên kế thừa: Cho phép bạn thay đổi một phần thuật toán bằng cách mở rộng chúng trong subclass. Strategy dựa trên thành phần: bạn có thể thay đổi một phần của hành vi của object bằng cách cung cấp các chiến thuật khác nhau để phản hồi cho hành vi đó. Template Method làm việc ở lớp, nên nó “static”. Strategy làm việc ở đối tượng, cho phép thay đổi hành vi ngay trong khi chương trình đang chạy.

### **4.10. Visitor**



Hình 4.33 Minh họa mẫu thiết kế Visitor

#### 4.10.1. Tổng quan

Visitor là một mẫu thiết kế thuộc nhóm Behavior Pattern cho phép tách các thuật toán khỏi các đối tượng mà chúng hoạt động.

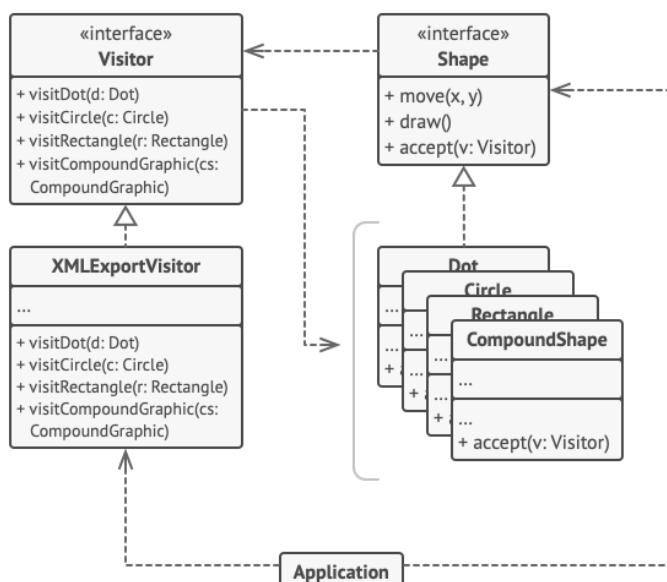
#### 4.10.2. Motivation

Bộ sưu tập là kiểu dữ liệu được sử dụng rộng rãi trong lập trình hướng đối tượng. Thường thì các bộ sưu tập chứa các đối tượng thuộc các loại khác nhau và trong những trường hợp đó, một số thao tác phải được thực hiện trên tất cả các phần tử của bộ sưu tập mà không cần biết loại.

Một cách tiếp cận khả thi để áp dụng một thao tác cụ thể trên các đối tượng thuộc các loại khác nhau trong bộ sưu tập sẽ là sử dụng các khối if kết hợp với 'instanceof' cho từng phần tử. Cách tiếp cận này không phải là một cách hay, không linh hoạt và không hướng đối tượng chút nào. Tại thời điểm này, chúng ta nên nghĩ đến nguyên tắc Open/Closed và từ đó chúng ta nên nhớ rằng chúng ta có thể thay thế các khối if bằng một lớp trừu tượng và mỗi lớp cụ thể sẽ thực hiện thao tác riêng của nó.

#### 4.10.3. Đặc điểm

##### 4.10.3.1. Cấu trúc mẫu



Hình 4.34: Sơ đồ lớp của mẫu thiết kế Visitor

##### 4.10.3.2. Thành phần trong cấu trúc mẫu

- Thành phần:

- Visitor interface khai báo một tập hợp các phương thức thăm có thể lấy các phần tử cụ thể của cấu trúc đối tượng làm đối số. Các phương thức này có

thể trùng tên nếu chương trình được viết bằng ngôn ngữ có hỗ trợ nạp chồng, nhưng kiểu tham số của chúng phải khác nhau.

- Mỗi Concrete Visitor triển khai một số phiên bản của các hành vi giống nhau, được điều chỉnh cho các lớp phần tử cụ thể khác nhau.
- Element interface khai báo một phương thức để "chấp nhận" các visitor. Phương thức này phải có một tham số được khai báo với kiểu là visitor interface.
- Mỗi Concrete Element phải triển khai thực hiện phương thức chấp nhận. Mục đích của phương thức này là chuyển hướng cuộc gọi đến phương thức của visitor thích hợp tương ứng với lớp phần tử hiện tại. Cần biết rằng ngay cả khi một lớp phần tử cơ sở triển khai phương thức này, tất cả các lớp con vẫn phải ghi đè phương thức này trong các lớp của chính chúng và gọi phương thức thích hợp trên đối tượng Visitor.
- Client thường đại diện cho một tập hợp hoặc một số đối tượng phức tạp khác (ví dụ, một cây Composite). Thông thường, các client không biết tất cả các lớp phần tử cụ thể vì chúng làm việc với các đối tượng từ tập hợp đó thông qua một số interface trừu tượng.

- **Sự cộng tác:**

- Một Client sử dụng mẫu Visitor truy cập phải tạo một đối tượng ConcreteVisitor và sau đó duyệt qua cấu trúc đối tượng, truy cập từng phần tử cùng với khách truy cập.
- Khi một phần tử được thăm, nó gọi hoạt động của Visitor truy cập tương ứng với lớp của nó. Phần tử cung cấp chính nó như một đối số cho thao tác này để cho phép Visitor truy cập vào trạng thái của nó, nếu cần.

#### **4.10.4. Khả năng ứng dụng**

- Sử dụng khi cần thực hiện thao tác trên tất cả các phần tử của cấu trúc đối tượng phức tạp.
- Sử dụng để làm sạch logic nghiệp vụ của các hành vi phụ trợ.
- Sử dụng khi một hành vi chỉ có ý nghĩa trong một số lớp của hệ thống phân cấp lớp, nhưng không có ý nghĩa trong các lớp khác.

#### **4.10.5. Hệ quả**

#### **4.10.5.1. Ưu điểm**

- Open/Closed Principle: có thể giới thiệu một hành vi mới có thể hoạt động với các đối tượng của các lớp khác nhau mà không cần thay đổi các lớp này.
- Single Responsibility Principle: có thể chuyển nhiều phiên bản của cùng một hành vi vào cùng một lớp.
- Một đối tượng visitor có thể tích lũy một số thông tin hữu ích khi làm việc với nhiều đối tượng khác nhau. Điều này có thể giúp ích khi ta muốn duyệt qua một số cấu trúc đối tượng phức tạp, chẳng hạn như cây đối tượng và áp dụng visitor cho từng đối tượng của cấu trúc này.

#### **4.10.5.2. Nhược điểm**

- Cần cập nhật tất cả visitor mỗi khi một lớp được thêm vào hoặc xóa khỏi hệ thống phân cấp phần tử.
- Các visitor có thể thiếu quyền truy cập cần thiết vào các trường riêng tư và phương thức của các phần tử mà họ phải làm việc với.

#### **4.10.6. Các mẫu thiết kế liên quan**

- Có thể xem Visitor là một phiên bản hiệu quả của Command. Các đối tượng của nó có thể thực thi các operation trên các đối tượng khác nhau của các lớp khác nhau.
- Có thể sử dụng Visitor để thực hiện một thao tác trên toàn bộ cây Composite.
- Có thể sử dụng Visitor cùng với Iterator để duyệt qua một cấu trúc dữ liệu phức tạp và thực hiện một số thao tác trên các phần tử của nó, ngay cả khi tất cả chúng đều có các lớp khác nhau.

### **4.11. Iterator**

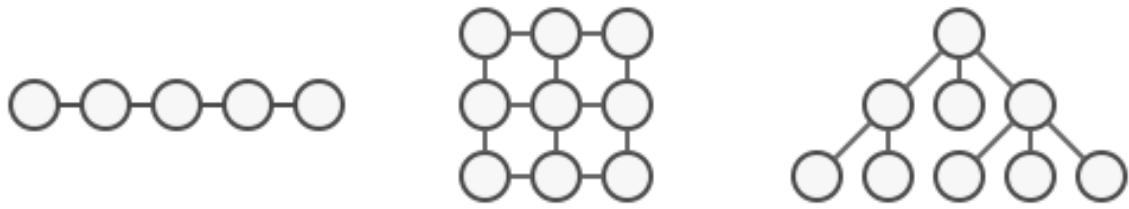
#### **4.11.1. Tổng quan**

Iterator Pattern cho phép truy cập tới các phần tử của một đối tượng tập hợp mà không cần phải tạo dựng riêng các phương pháp truy cập cho đối tượng này.

Một Iterator được thiết kế cho phép xử lý nhiều loại tập hợp khác nhau bằng cách truy cập những phần tử của tập hợp với cùng một phương pháp, cùng một cách thức định sẵn, mà không cần hiểu rõ về những chi tiết bên trong của những tập hợp này.

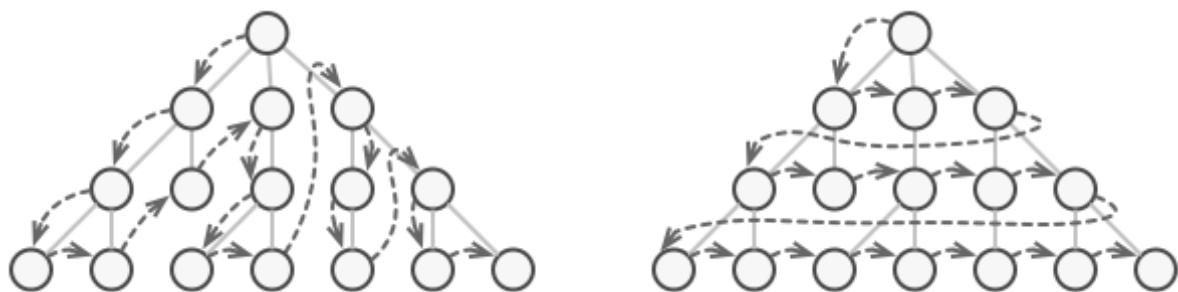
#### **4.11.2. Motivation**

- Tập hợp là một trong những kiểu dữ liệu được sử dụng phổ biến nhất. Tuy nhiên, nó chỉ là một nhóm bao gồm các đối tượng.



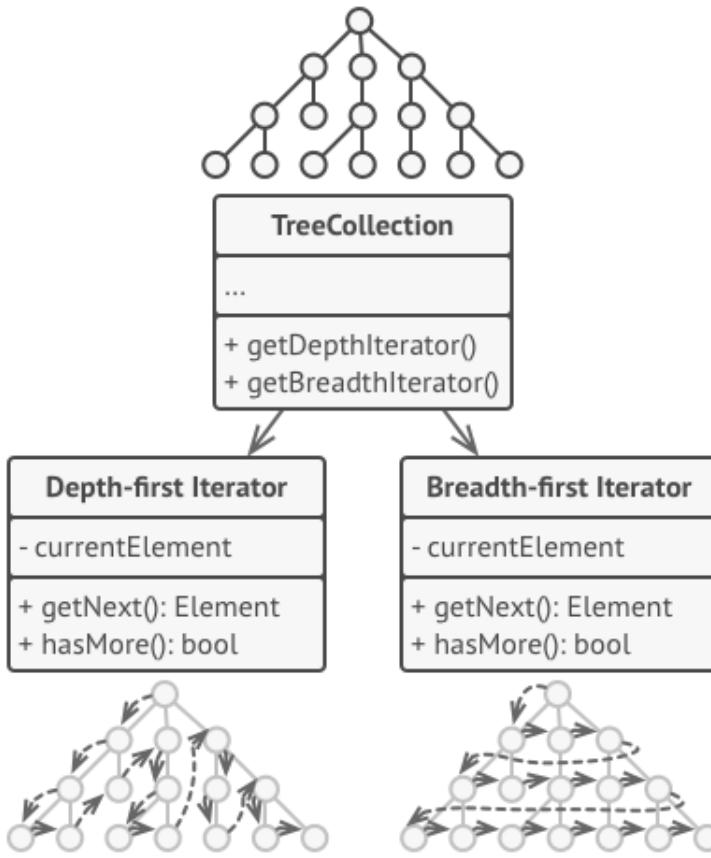
Hình 4.35 Một số dạng tập hợp

- Hầu hết các tập hợp lưu trữ các phần tử của chúng trong các list, stack, tree,... Nhưng cho dù tập hợp được cấu trúc như thế nào, nó vẫn phải cung cấp một số cách truy cập đến các phần tử của nó để sử dụng. Và cần có một cách để đi qua từng phần của tập hợp và không bị lặp lại các phần tử đã đi qua.
- Điều này nghe có vẻ như là một công việc dễ dàng nếu bạn có một tập hợp dựa trên một list. Bạn chỉ cần lặp lại tất cả các phần tử. Nhưng làm thế nào để bạn tuần tự đi qua các phần tử của một cấu trúc dữ liệu phức tạp như tree?



Hình 4.36 Mô hình DFS và BFS

- Hiện nay chúng ta có rất nhiều thuật toán áp dụng cho tập hợp, làm mờ đi mục đích chính của việc sử dụng tập hợp, đó là lưu trữ dữ liệu hiệu quả. Ngoài ra, một số thuật toán có thể được điều chỉnh cho một ứng dụng cụ thể, vì vậy việc gộp chung vào một lớp tập hợp chung chung sẽ rất kỳ lạ.
- Mặt khác, source code của chúng ta phải hoạt động với những tập hợp khác nhau, thậm chí có thể không quan tâm đến cách chúng lưu trữ của chúng. Tuy nhiên, vì tất cả các tập hợp đều cung cấp các cách khác nhau để truy cập các phần tử của chúng, bạn không có lựa chọn nào khác hơn là code riêng cho từng lớp xử lý.

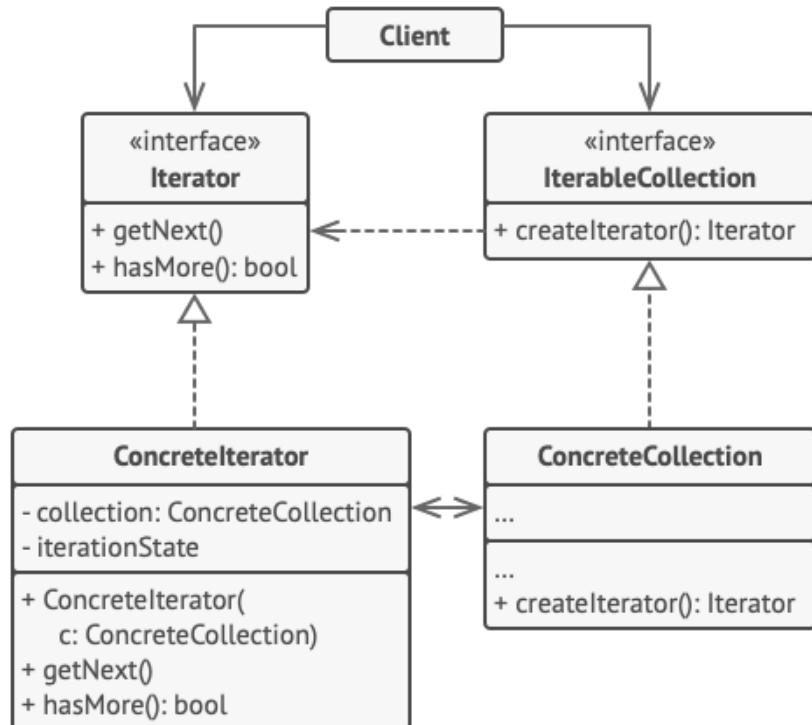


Hình 4.37 Sơ đồ lớp của DFS và BFS

- Iterator Pattern sẽ hỗ trợ giải quyết vấn đề này bằng trích xuất đường truyền hành vi của một tập hợp thành một đối tượng riêng biệt được gọi là iterator. Ngoài việc thực thi thuật toán xử lý, một iterator object sẽ đóng gói tất cả các chi tiết được truyền đi, chẳng hạn như vị trí hiện tại và bao nhiêu phần tử còn lại cho đến khi kết thúc.
- Thông thường, iterator cung cấp một method chính để tìm nạp các phần tử của tập hợp. Client có thể tiếp tục thực thi phương thức này cho đến khi nó không return lại bất cứ điều gì, điều đó có nghĩa là iterator sẽ đi qua tất cả các phần tử.
- Tất cả các iterator phải được implement cùng một interface. Điều này làm cho source code của client tương thích với bất kỳ loại tập hợp nào hoặc bất kỳ thuật toán so sánh nào, miễn là có một iterator thích hợp. Nếu bạn cần một cách đặc biệt để duyệt qua một tập hợp, bạn chỉ cần tạo một iterator, mà không cần phải thay đổi tập hợp hoặc client.
- Trong thực tế, khi đi du lịch, việc random hướng đi trong suy nghĩ, định hướng trong smartphone hoặc hướng dẫn viên du lịch hướng dẫn đường đi, là một ứng dụng của iterator với các địa điểm tham quan.

### 4.11.3. Đặc điểm

#### 4.11.3.1. Cấu trúc mẫu



Hình 4.38 Cấu trúc mẫu của Iterator

#### 4.11.3.2. Thành phần trong cấu trúc mẫu

- Iterator: là một interface hay abstract class, khai báo các hoạt động cần thiết để tra so sánh một tập hợp: tìm nạp phần tử tiếp theo, truy xuất vị trí hiện tại, bắt đầu lại lặp lại,...
- Concrete Iterator: implement các phương thức của Iterator, giữ index khi duyệt qua các phần tử. Cho phép một số trình vòng lặp đi qua cùng một tập hợp độc lập với nhau.
- Collection Interface: khai báo một hoặc nhiều phương thức để nhận được các Iterator tương thích với tập hợp. Lưu ý rằng kiểu trả về của các phương thức phải được khai báo dưới dạng Iterator Interface để các tập hợp cụ thể có thể trả về các loại Iterator.
- Concrete Collections: trả về các phiên bản mới của một lớp Concrete Iterator cụ thể mỗi khi client yêu cầu.
- Client: đối tượng sử dụng Iterator Pattern, nó yêu cầu một iterator từ một đối tượng tập hợp để duyệt qua các phần tử mà nó giữ. Các phương thức của iterator được sử dụng để truy xuất các phần tử từ collection theo một trình tự thích hợp.

#### **4.11.4. Khả năng ứng dụng**

Iterator thường được sử dụng khi:

- Cần truy cập nội dung của đối tượng trong tập hợp mà không cần biết nội dung cài đặt bên trong nó.
- Hỗ trợ truy xuất nhiều loại tập hợp khác nhau.
- Cung cấp một interface duy nhất để duyệt qua các phần tử của một tập hợp.

#### **4.11.5. Hệ quả**

##### **4.11.5.1. Ưu điểm**

- Đảm bảo nguyên tắc Single responsibility principle (SRP): chúng ta có thể tách phần cài đặt các phương thức của tập hợp và phần duyệt qua các phần tử (iterator) theo từng class riêng lẻ.
- Đảm bảo nguyên tắc Open/Closed Principle (OCP): chúng ta có thể implement các loại tập hợp mới và iterator mới, sau đó chuyển chúng vào code hiện có mà không vi phạm bất cứ nguyên tắc gì.
- Chúng ta có thể truy cập song song trên cùng một tập hợp vì mỗi đối tượng iterator có chứa trạng thái riêng của nó.

##### **4.11.5.2. Nhược điểm**

- Sử dụng Iterator có thể kém hiệu quả hơn so với việc duyệt qua các phần tử của tập hợp một cách trực tiếp.
- Có thể không cần thiết nếu ứng dụng chỉ hoạt động với các tập hợp đơn giản.

#### **4.11.6. Các mẫu thiết kế liên quan**

- Có thể sử dụng Iterator để duyệt qua Composite tree.
- Có thể sử dụng Factory Method cùng với Iterator để các lớp con của tập hợp trả về các iterator khác nhau tương thích với các tập hợp.
- Sử dụng Memento cùng với Iterator để nhận biết trạng thái iterator hiện tại và quay lại nếu cần thiết.
- Có thể sử dụng Visitor cùng với Iterator để duyệt qua một cấu trúc dữ liệu phức tạp và thực thi một số hoạt động trên các element của nó, ngay cả khi chúng ở các lớp khác nhau.