ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH

LẬP TRÌNH NÂNG CAO (CO2039)

ASSIGNMENT REPORT

DESIGN PATTERNS
TRONG LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Giảng viên hướng dẫn: Thầy Trương Tuấn Anh
Lớp: CN01
Sinh viên thực hiện: Lê Hoàng Phúc – MSSV: 2152239

TP. Hồ Chí Minh, tháng 6/2023

Mục lục

1 Design Pattern là gì? Có bao nhiêu Patterns trong lập trình hướng đối tượng?

1.1 Định nghĩa Design Pattern

Design Patterns (tạm dịch là "mẫu thiết kế") là các giải pháp điển hình cho các vấn đề thường xảy ra trong thiết kế phần mềm. Chúng giống như các bản thiết kế có sẵn mà chúng ta có thể tùy chỉnh để giải quyết vấn đề thường gặp trong lúc lập trình.

Design Patterns không phải là một đoạn code cụ thể nào, mà là một khái niệm chung, một concept chung dùng để giải quyết một vấn đề đặc thù. Nó không như các hàm hoặc thư viện có sẵn, các Design Patterns không thể copy y chang vào trong code của mình, ta chỉ có thể làm theo chi tiết của từng mẫu và tự thực hiện theo mẫu sao cho phù hợp với chương trình thực tế của mình.

Design Patterns thường bị nhầm lẫn với các giải thuật (algorithms), bởi vì cả hai khái niệm đều mô tả các giải pháp điển hình cho một số vấn đề đã biết. Giải thuật thì luôn xác định một tập hợp rõ ràng các hành động có thể đạt được một số mục tiêu, còn Design Pattern thì mô tả cao cấp hơn về một giải pháp. Code của cùng một pattern khi áp dụng cho hai chương trình khác nhau có thể sẽ rất khác nhau. Giải thuật như công thức nấu ăn (đều gồm các bước rõ ràng), còn Design Pattern thì như một bản thiết kế (chỉ thấy kết quả và tính năng, còn để thực hiện thì lại tùy vào người xây dựng).

Hầu hết các Design Patterns đều được mô tả chính thức nên mọi người đều có thể "tái tạo" chúng trong nhiều hoàn cảnh. Các mô tả thường gồm những phần như:

- Mục đích của pattern để giải quyết vấn đề nào và giải pháp
- Giải thích đặc điểm của vấn đề và sự phù hợp của pattern trong việc giải quyết
- Mô hình của pattern và chức năng của các thành phần
- Code ví dụ, thường được viết bằng mã giả (pseudocode) hay các ngôn ngữ lập trình phổ biến để người học hiểu sâu hơn

1.2 Đôi nét về lịch sử của Design Pattern

Design Patterns là các giải pháp điển hình cho các vấn đề phổ biến trong lập trình hướng đối tượng. Khi một giải pháp được lặp đi lặp lại trong nhiều dự án khác nhau, cuối cùng ai đó sẽ đặt tên cho nó và mô tả chi tiết giải pháp đó. Đó là cách mà một Pattern được phát hiện. Vậy, Design Patterns không phải phát minh, mà là những phát hiện trong quá trình thiết kế phần mềm của những nhà phát triển.

Khái niệm về các "patterns" được Christopher Alexander mô tả lần đầu tiên trong "A Pattern Language: Towns, Buildings, Construction". Cuốn sách mô tả một “ngôn ngữ” để thiết kế môi trường đô thị. Các đơn vị của ngôn ngữ này là các pattern. Chúng có thể mô tả cửa sổ nên cao bao nhiêu, tòa nhà nên có bao nhiêu tầng, diện tích cây xanh trong khu phố nên là bao nhiêu, v.v.

Năm 1994, Erich Gamma, John Vlissides, Ralph Johnson và Richard Helm cho xuất bản cuốn sách "Design Patterns: Elements of Reusable Object-Oriented Software", áp dụng khái niệm mẫu thiết kế vào lập trình. Cuốn sách giới thiệu 23 mẫu giải quyết các vấn đề khác nhau của thiết kế hướng đối tượng và đã trở thành cuốn sách bán chạy nhất rất nhanh chóng. Do cái tên dài dòng của nó, mọi người bắt đầu gọi nó là "cuốn sách của nhóm bốn người" hay "cuốn sách GoF".

Kể từ đó, hàng chục patterns hướng đối tượng khác đã được phát hiện. Việc tìm ra các patterns trở nên phổ biến trong lĩnh vực lập trình, và hiện đã có rất nhiều patterns nằm ngoài lập trình hướng đối tượng.

1.3 Số lượng và phân loại Design Pattern trong lập trình hướng đối tượng

Như đã giới thiệu ở trên, hiện trên thế giới có rất nhiều Design Patterns được phát hiện sau khi GoF xuất bản sách về chúng vào năm 1994. Tuy nhiên, 23 Design Patterns được giới thiệu trong sách là chuẩn mực nhất và được áp dụng rộng rãi nhất. Vậy chúng ta sẽ đi vào nghiên cứu 23 mẫu này.

Căn cứ vào mục đích sử dụng, ta có thể chia 23 Design Patterns thành 3 nhóm:

- **Creational Pattern** (gồm 5 mẫu) cung cấp các cơ chế tạo đối tượng giúp tăng tính linh hoạt và tái sử dụng code hiện có.
- **Structural Pattern** (gồm 7 mẫu) tập hợp các đối tượng và lớp thành các cấu trúc lớn hơn, đồng thời giữ cho các cấu trúc này linh hoạt và hiệu quả.
- **Behavioral Pattern** (gồm 11 mẫu) quản lý việc giao tiếp hiệu quả và phân công trách nhiệm giữa các đối tượng.

1.4 Lý do nên học Design Pattern

- Design Patterns là một bộ công cụ gồm các giải pháp đã được thử nghiệm và kiểm tra cho các vấn đề phổ biến trong thiết kế phần mềm. Ngay cả khi chúng ta chưa bao giờ gặp phải những vấn đề này, thì việc biết các mẫu vẫn hữu ích vì nó dạy chúng ta cách giải quyết tất cả các loại vấn đề bằng cách sử dụng các nguyên tắc của lập trình hướng đối tượng.
- Design Patterns có thể thiết lập một ngôn ngữ chung để dễ dàng làm việc với đồng nghiệp của mình hiệu quả hơn. Chẳng hạn như câu nói: “Chỗ này dùng Singleton đi.”, người biết về Design Pattern sẽ hiểu ý tưởng và không cần giải thích Singleton là gì khi đã biết tên và công dụng của nó.

2 Mô tả chi tiết các Patterns

2.1 Creational Patterns

Creational Patterns là nhóm các Patterns cung cấp các cơ chế khởi tạo đối tượng (object creation mechanisms), giúp tăng tính linh hoạt (flexibility) và tái sử dụng (reuse) code có sẵn. Nhóm này gồm 5 mẫu: **Factory Method**, **Abstract Factory**, **Builder**, **Prototype** và **Singleton**.

Creational Patterns trừu tượng hóa quá trình khởi tạo, tạo ra một hệ thống độc lập với cách các đối tượng của nó được cấu thành và biểu thị. Một lớp (class) Creational Pattern sử dụng tính kế thừa để thay đổi lớp được khởi tạo, trong khi một đối tượng Creational Pattern sẽ ủy quyền khởi tạo cho một đối tượng khác.

Khi các hệ thống phát triển và trở nên phụ thuộc nhiều hơn vào thành phần đối tượng hơn là sự kế thừa (class inheritance), Creational Patterns càng trở nên quan trọng. Khi đó, việc code sẽ có xu hướng chuyển từ một tập hợp các hành vi cố định (fixed set of behaviors) tới một tập hợp nhỏ hơn với các hành vi cơ bản (fundamental behaviors) mà có thể được tổng hợp lại thành bất kỳ tập hợp phức tạp hơn. Do đó, việc tạo ra các đối tượng với các hành vi cụ thể đòi hỏi nhiều hơn là chỉ khởi tạo một class.

Các mẫu Creational Patterns có hai tính chất:

- Thứ nhất, chúng đều gói gọn mọi thứ về class cụ thể mà hệ thống sử dụng.
- Thứ hai, chúng ẩn cách mà các thể hiện của class (instance) được tạo ra và tổng hợp.

Tất cả những gì hệ thống nói chung biết về các đối tượng là các giao diện (interfaces) của chúng được định nghĩa bởi các lớp trừu tượng (abstract classes). Vì vậy, các Creational Patterns mang lại cho chúng ta rất nhiều sự linh hoạt (flexibility) về việc tạo ra cái gì (what), ai tạo ra nó (who), nó được tạo ra như thế nào (how) và khi nào (when).

Chúng cho phép chúng ta thiết lập cấu hình một hệ thống với các đối tượng "sản phẩm" rất khác nhau về cấu trúc và chức năng. Cấu hình có thể là tĩnh (static) - nghĩa là được chỉ định tại thời điểm biên dịch (compile-time), hoặc động (dynamic) - tại thời điểm chạy (run-time).

Các đoạn code mẫu cho Creational Pattern có ở đường link sau:
https://github.com/PhucLe03/OOP/tree/main/Design%20Patterns/Creational%20Patterns
hoặc mã QR bên dưới.

2.1.1 Factory Method

Định nghĩa

Factory Method là một Design Pattern cung cấp một giao diện (interface) để tạo đối tượng trong class cha nhưng cho phép class con của nó ghi đè để tạo đối tượng theo những kiểu khác nhau của bài toán.

Nói cách khác, Factory Method là một Design Pattern của lập trình hướng đối tượng trong thiết kế phần mềm, nhằm giải quyết vấn đề tạo một đối tượng mà không cần thiết chỉ ra class nào sẽ được tạo. Factory Method giải quyết vấn đề này bằng cách định nghĩa một phương thức cho việc tạo đối tượng, và các class con thừa kế có thể ghi đè để chỉ rõ đối tượng nào sẽ được tạo.

Cách sử dụng

Ví dụ:

Giả sử một công ty vận tải đang sử dụng một ứng dụng để quản lý vận chuyển. Ban đầu công ty chỉ sử dụng xe tải (Truck) để vận chuyển hàng hóa. Sau một thời gian, công ty này từ một công ty nhỏ chỉ vận chuyển hàng hóa dần dần lấn sang vận chuyển hành khách, loại phương tiện mà công ty thêm vào biên chế là xe khách (Bus), sau này công ty còn có ý định thêm xe taxi (Cab). Vấn đề là ứng dụng mà công ty sử dụng ban đầu chỉ có mục đích quản lý xe tải, bây giờ cần phải quản lý thêm các phương tiện khác. Vậy, những dòng code cũ cần phải sửa đổi để phù hợp với nhu cầu mới của công ty và khách hàng.

Giải pháp thông thường:

Ta xét chương trình viết bằng C++ sau:

- Abstract class của các phương tiện giao thông:

```cpp
// A design without factory pattern
#include <iostream>
using namespace std;

// Library classes
class Vehicle {
public:
   virtual void printVehicle() = 0;
};
```

- Các phương tiện cụ thể:

```cpp
class Truck : public Vehicle
{
public:
   void printVehicle()
   {
      cout << "Truck incoming" << endl;
   }
};
class Bus : public Vehicle
{
public:
   void printVehicle()
   {
      cout << "Bus incoming" << endl;
   }
};
```

- Các phương thức dành cho phía khách hàng:

```
// Client (or user) class
class Client {
public:
   Client(int type)
   {
      // Client calls vehicle according to type
      if (type == 1)
         pVehicle = new Truck();
      else if (type == 2)
         pVehicle = new Bus();
      else
         pVehicle = NULL;
   }

   ~Client()
   {
      if (pVehicle) {
         delete pVehicle;
         pVehicle = NULL;
      }
   }

   Vehicle* getVehicle() { return pVehicle; }

private:
   Vehicle* pVehicle;
};
```

- Ví dụ mô phỏng cho các thao tác của khách hàng trong ứng dụng:

```
// Driver program
int main()
{
   Client* pClient = new Client(1);
   Vehicle* pVehicle = pClient->getVehicle();
   pVehicle->printVehicle();
   return 0;
}
```

Kết quả chạy

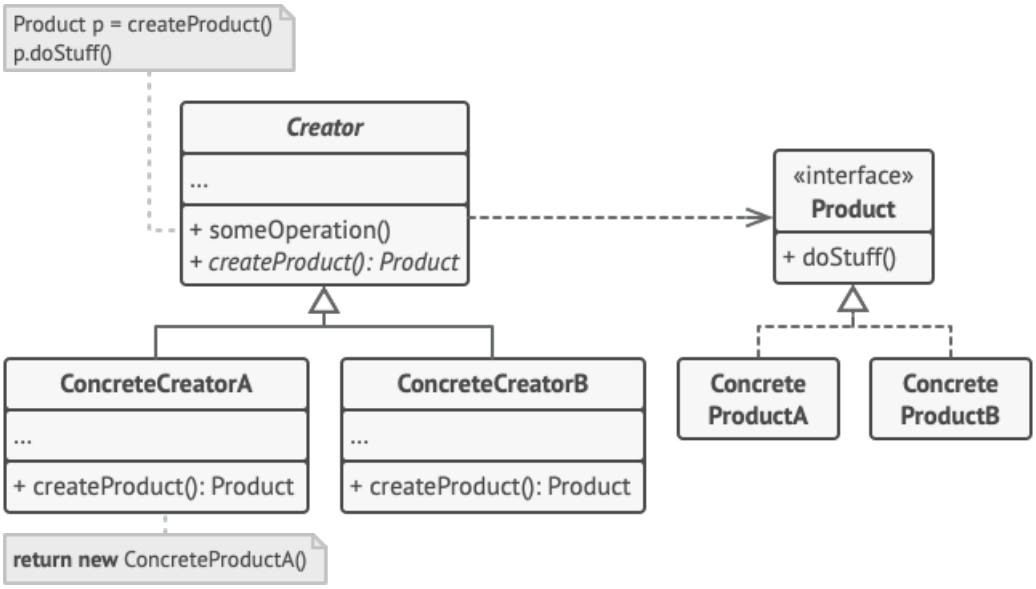
Truck incoming

Giả sử các class về phương tiện thuộc thư viện trong máy chủ của công ty, còn class Client tượng trưng cho ứng dụng của khách hàng. Nếu như sau này công ty muốn thêm các phương tiện khác vào biên chế, trong constructor của class Client phải thêm các dòng if-else để gửi tạo yêu cầu cho server.

Điều này có nghĩa là nếu thêm bất kỳ phương tiện nào thì ta cũng phải sửa code ở cả thư viện và ở client. Vậy để tránh phiền phức, ta có thể dùng Factory Method như sau.

Các thành phần trong Factory Method Pattern:

- **Product**: Là interface mà có đặc tính phổ biến nhất để tạo các sản phẩm trong class con.
- **Concrete Products**: Là các sản phẩm cụ thể sau khi được tạo thông qua Factory Method.
- **Creator**: Đây sẽ là nơi tạo ra Factory Method. Điểm quan trọng ở đây đó là kiểu trả về của Factory Fethod phải là kiểu Product. Ta có thể khai báo Factory Method với kiểu abstract để các class con có thể ghi đè lên.



Factory Method Pattern:

- Product, là abstract class của các phương tiện

```
// Factory method design pattern
#include <iostream>
using namespace std;

enum VehicleType {
   Truck, Bus, Cab, Bike, Plane
};

// Library classes
class Vehicle {
public:
   virtual void printVehicle() = 0;
   static Vehicle* Create(VehicleType type);
   virtual ~Vehicle() = 0;
};
```

- Concrete Products, là các phương tiện

```
class _Truck : public Vehicle {
public:
   void printVehicle() {cout << "Truck incoming" << endl;}
};

class _Bus : public Vehicle {
public:
   void printVehicle() {cout << "Bus incoming" << endl;}
};

class _Cab : public Vehicle {
public:
   void printVehicle() {cout << "Cab incoming" << endl;}
};

class _Bike : public Vehicle {
public:
   void printVehicle() {cout << "Bike incoming" << endl;}
};

class _Plane : public Vehicle {
public:
   void printVehicle() {cout << "Plane incoming" << endl;}
};
```

- Creator

```
Vehicle* Vehicle::Create(VehicleType type)
{
    if (type == Truck)
        return new _Truck();
    else if (type == Bus)
        return new _Bus();
    else if (type == Cab)
        return new _Cab();
    else if (type == Bike)
        return new _Bike();
    else if (type == Plane)
        return new _Plane();
    else return NULL;
}
```

- Client và mô phỏng ứng dụng

```
// Client class
class Client
{
public:
    // Client doesn't explicitly create objects
    // but passes type to factory method "Create()"
    Client(VehicleType type)
    {
        pVehicle = Vehicle::Create(type);
    }
    ~Client()
    {
        if (pVehicle)
        {
            delete pVehicle;
            pVehicle = NULL;
        }
    }
    Vehicle* getVehicle()
    {
        return pVehicle;
    }

private:
    Vehicle *pVehicle;
};

// Driver program
int main() {
    Client *pClient = new Client(Plane);
    Vehicle * pVehicle = pClient->getVehicle();
    pVehicle->printVehicle();
    delete pClient;
    return 0;
}
```

Kết quả chạy

Plane incoming

Với đoạn code trên, code ở client không cần thay đổi mà ta vẫn có thể thêm bất cứ phương tiện vận chuyển nào, và client có thể gọi tất cả các phương tiện đó.

Ưu nhược điểm của Factory Method Pattern

- Ưu điểm
  - Giúp tránh được việc gắn chặt việc tạo sản phẩm với bất kỳ một loại sản phẩm cụ thể nào.
  - Có thể gom code tạo sản phẩm về một nơi trong chương trình, giúp việc bảo trì dễ dàng hơn.
  - Chúng ta có thể dễ dàng thêm một kiểu sản phẩm mới mà không làm ảnh hưởng đến code hiện tại.
- Nhược điểm
  - Code sẽ trở nên phức tạp khi có quá nhiều class con được sử dụng để triển khai Pattern này.

2.1.2 Abstract Factory

Định nghĩa

Abstract Factory là một Pattern cung cấp một một interface có chức năng tạo ra các đối tượng liên quan mà không quan tâm đó là lớp cụ thể nào ở thời điểm thiết kế.

Abstract factory có thể tưởng tượng như là một nhà máy lớn chứa nhiều nhà máy nhỏ, trong các nhà máy đó có những xưởng sản xuất, các xưởng đó tạo ra những sản phẩm khác nhau.

Cách sử dụng

Ví dụ:

Giả sử ta đang thiết kế một app cần phải chạy cả trên Linux và Windows. App này gồm menu và các nút bấm cho các tính năng riêng.

Xét mẫu chương trình bằng C++ sau:

```cpp
#include <iostream>
#define LINUX

using namespace std;

// Abstract base product.
class Widget
{
public:
    virtual void draw() = 0;
};

// Concrete product family 1.
class LinuxButton : public Widget
{
public:
    void draw() { cout << "LinuxButton\n"; }
};
class LinuxMenu : public Widget
{
public:
    void draw() { cout << "LinuxMenu\n"; }
};

// Concrete product family 2.
class WindowsButton : public Widget
{
public:
    void draw() { cout << "WindowsButton\n"; }
};
class WindowsMenu : public Widget
{
public:
    void draw() { cout << "WindowsMenu\n"; }
};

/**
 * Here's a client, which uses concrete products directly.
 * It's code filled up with nasty switch statements
 * which check the product type before its use.
 */

```

```
43 class Client
44 {
45 public:
46     void draw()
47     {
48 #ifdef LINUX
49         Widget *w = new LinuxButton;
50 #else // WINDOWS
51         Widget *w = new WindowsButton;
52 #endif
53         w->draw();
54         display_window_one();
55         display_window_two();
56     }
57     void display_window_one() {
58 #ifdef LINUX
59         Widget *w[] =
60         {
61             new LinuxButton,
62             new LinuxMenu
63         };
64 #else // WINDOWS
65         Widget *w[] =
66         {
67             new WindowsButton,
68             new WindowsMenu
69         };
70 #endif
71         w[0]->draw();
72         w[1]->draw();
73     }
74     void display_window_two() {
75 #ifdef LINUX
76         Widget *w[] =
77         {
78             new LinuxMenu,
79             new LinuxButton
80         };
81 #else // WINDOWS
82         Widget *w[] =
83         {
84             new WindowsMenu,
85             new WindowsButton
86         };
87 #endif
88         w[0]->draw();
89         w[1]->draw();
90     }
91 };
92
93 int main() {
94     Client *c = new Client();
95     c->draw();
96     delete c;
97     return 0;
98 }
```

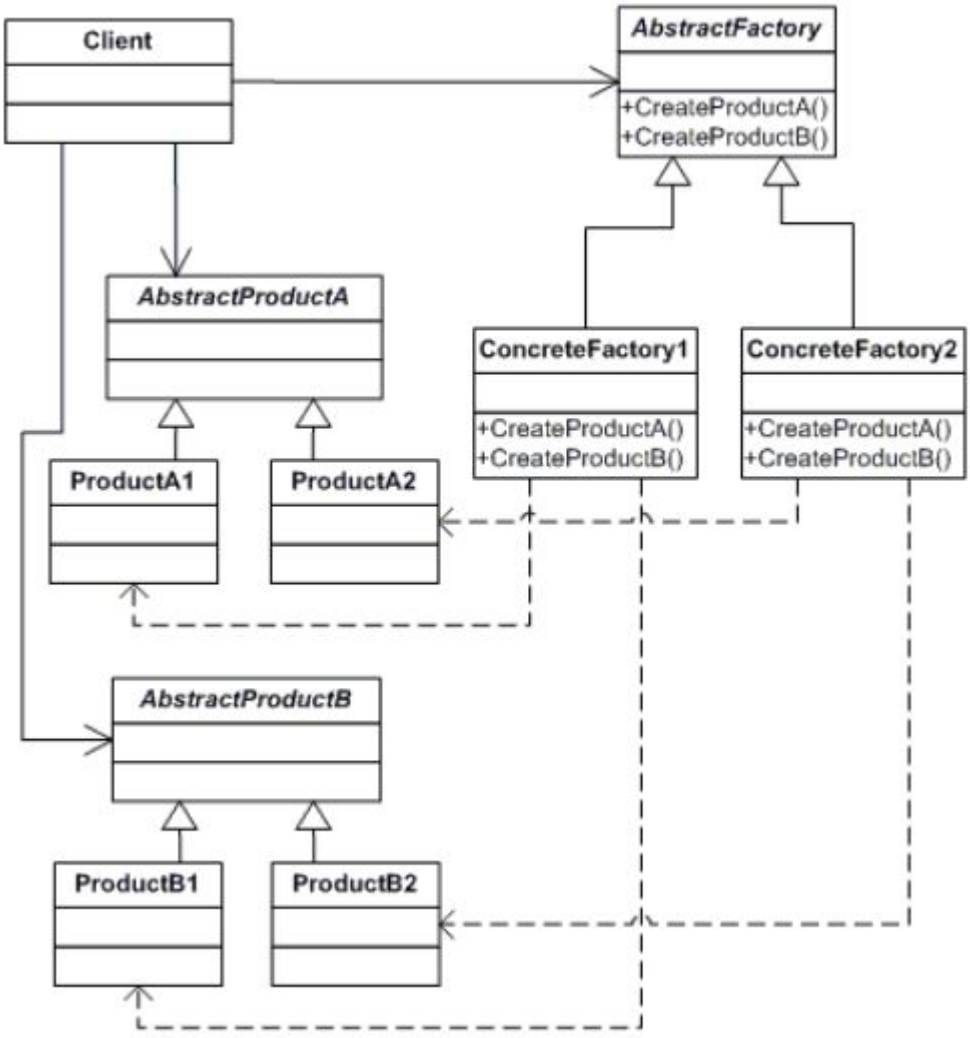
Ở ví dụ trên, client phải tự kiểm tra xem có đang sử dụng LINUX hay không, nếu có thì gọi class của LINUX, ngược lại gọi class của WINDOWS.

Tuy nhiên, vì chính client phải tự kiểm tra hệ điều hành trong từng chức năng riêng lẻ nên tổng thể code nhìn rất rối rắm, rất khó bảo trì hay tái sử dụng.

Ta xét thêm hướng tiếp cận Abstract Factory Pattern như sau.

Các thành phần trong Abstract Factory Pattern:

- **AbstractFactory** : Định nghĩa interface cho các thao tác tạo đối tượng trừu tượng.
- **ConcreteFactory**: Hiện thực các thao tác được định nghĩa trong AbstractFactory để tạo đối tượng thực.
- **AbstractProduct**: Định nghĩa interface cho các sản phẩm trừu tượng.
- **ConcreteProduct**: Định nghĩa đối tượng sản phẩm được khởi tạo bởi các ConcreteFactory tương ứng và hiện thực các thao tác của AbstractProduct.
- **Client**: Sử dụng interface định nghĩa trong AbstractFactory và AbstractProduct.



Abstract Factory Pattern:

- AbstractProduct

```
#include <iostream>
#define LINUX

using namespace std;

// Abstract base product.
class Widget
{
 public:
  virtual void draw() = 0;
};
```

- ConcreteProduct

```
// Concrete product family 1.
class LinuxButton : public Widget
{
 public:
  void draw() { cout << "LinuxButton\n"; }
};
class LinuxMenu : public Widget
{
 public:
  void draw() { cout << "LinuxMenu\n"; }
};

// Concrete product family 2.
class WindowsButton : public Widget
{
 public:
  void draw() { cout << "WindowsButton\n"; }
};
class WindowsMenu : public Widget
{
 public:
  void draw() { cout << "WindowsMenu\n"; }
};
```

- AbstractFactory

```
// Abstract factory defines methods to create all related products.
class Factory
{
 public:
  virtual Widget *create_button() = 0;
  virtual Widget *create_menu() = 0;
};
```

- ConcreteFactory

```
/**
 * Each concrete factory corresponds to one product
 * family. It creates all possible products of
 * one kind.
 */
class LinuxFactory : public Factory
{
 public:
  Widget *create_button()
  {
    return new LinuxButton;
  }
  Widget *create_menu()
  {
    return new LinuxMenu;
  }
};

/**
 * Concrete factory creates concrete products, but
 * returns them as abstract.
 */
class WindowsFactory : public Factory
{
 public:
  Widget *create_button()
  {
    return new WindowsButton;
  }
  Widget *create_menu()
  {
    return new WindowsMenu;
  }
};
```

- Client

```
/**
 * Client receives a factory object from its creator.
 */
class Client
{
 private:
  Factory *factory;

 public:
  Client(Factory *f)
  {
    factory = f;
  }

  void draw()
  {
    Widget *w = factory->create_button();
    w->draw();
    display_window_one();
    display_window_two();
  }

  void display_window_one()
  {
    Widget *w[] = {
        factory->create_button(),
        factory->create_menu()
    };
    w[0]->draw();
    w[1]->draw();
  }

  void display_window_two()
  {
    Widget *w[] =
    {
        factory->create_menu(),
        factory->create_button()
    };
    w[0]->draw();
    w[1]->draw();
  }
};
```

- Hàm main()

```
int main()
{
  Factory *factory;
#ifdef LINUX
  factory = new LinuxFactory;
#else // WINDOWS
  factory = new WindowsFactory;
#endif

  Client *c = new Client(factory);
  c->draw();
  delete c;
  return 0;
}
```

Khác với đoạn code trước, ở đây chúng ta sử dụng thêm class Factory để quản lý việc kiểm tra hệ điều hành thay cho các class khác. Chương trình trở nên gọn gàng, dễ sửa chữa, dễ nâng cấp hơn.

Kết quả chạy của cả hai hướng tiếp cận

LinuxButton
LinuxButton
LinuxMenu
LinuxMenu
LinuxButton

Ưu nhược điểm của Abstract Factory Pattern

- Ưu điểm
  - Các lợi ích của Abstract Factory Pattern cũng tương tự như Factory Method Pattern như: cung cấp hướng tiếp cận với giao diện, che giấu sự phức tạp của việc khởi tạo các đối tượng với người dùng (client), độc lập giữa việc khởi tạo đối tượng và hệ thống sử dụng,...
  - Giúp tránh được việc sử dụng điều kiện logic bên trong Factory Method Pattern. Khi một Factory Method lớn (có quá nhiều sử lý if-else hay switch-case), chúng ta nên sử dụng theo mô hình Abstract Factory để dễ quản lý hơn (cách phân chia có thể là gom nhóm các class con cùng loại vào một Factory).
  - Abstract Factory Pattern là factory của các factory, có thể dễ dạng mở rộng để chứa thêm các factory và các class con khác.
  - Dễ dàng xây dựng một hệ thống đóng gói (encapsulate): sử dụng được với nhiều nhóm đối tượng (factory) và tạo nhiều sản phẩm khác nhau.
- Nhược điểm
  - Code có thể trở nên phức tạp hơn khi chúng ta cần giới thiệu nhiều lớp con mới để triển khai pattern. Trường hợp tốt nhất là khi ta sử dụng pattern này vào hệ thống phân cấp hiện có của các lớp khởi tạo.

2.1.3 Builder

Định nghĩa

Builder Pattern là một mẫu thiết kế được dùng để cung cấp một giải pháp linh hoạt cho các vấn đề tạo đối tượng khác nhau trong lập trình hướng đối tượng.

Builder Pattern cho phép chúng ta xây dựng các đối tượng phức tạp bằng cách sử dụng các đối tượng đơn giản và tiếp cận từng bước.

Builder Pattern còn có thể tạo ra các kiểu thể hiện khác nhau của một đối tượng bằng cách sử dụng cùng một code khởi tạo (constructor code).

Cách sử dụng

Ví dụ:

Giả sử chúng ta đang hiện thực constructor cho class tên là Student, class này gồm các thông tin về sinh viên như tên, mã số sinh viên, năm sinh, lớp học, địa chỉ, số điện thoại...

Vấn đề đặt ra là đối tượng của chúng ta có thể được khởi tạo với rất nhiều những tham số truyền vào và có thể một vài trong số đó không nhất thiết phải truyền vào trong lúc khởi tạo.

Xét đoạn code Java sau đây:

```
1  public Student(String id, String firstName, String lastName, String dayOfBirth,
       String currentClass, String address, String phone)
2  {
3     this.id = id;
4     this.firstName = firstName;
5     this.lastName = lastName;
6     this.dayOfBirth = dayOfBirth;
7     this.currentClass = currentClass;
8     this.address = address;
9     this.phone = phone;
10 }
```

Để giải quyết việc một vài tham số không được truyền vào khi khởi tạo đối tượng, ta có thể dùng kỹ thuật quá tải hàm, hay nạp chồng, ta viết thêm các constructor như sau:

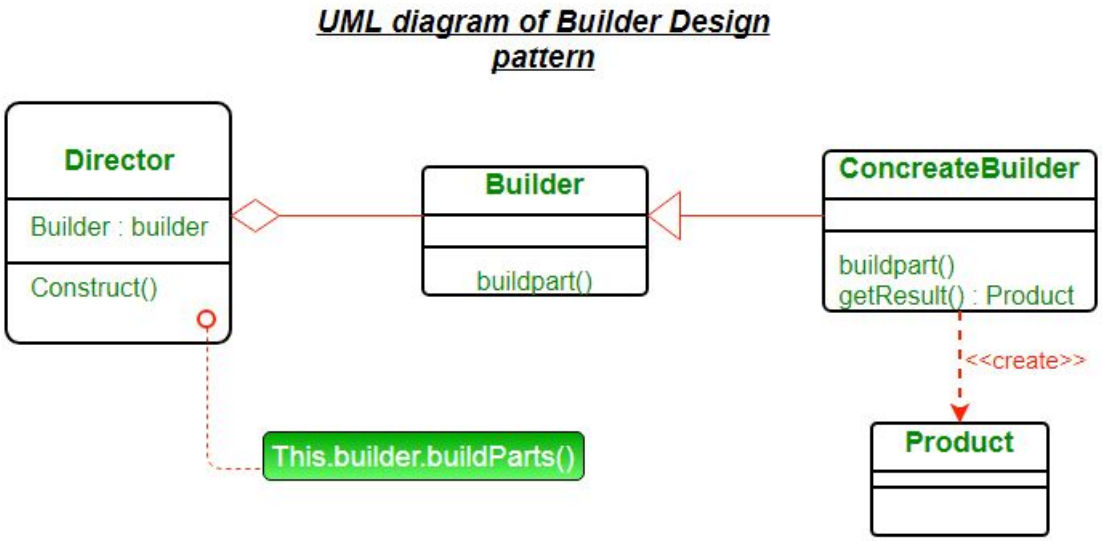
```
1  public Student(String id, String firstName, String lastName) {
2     this.id = id;
3     this.firstName = firstName;
4     this.lastName = lastName;
5  }
6
7  public Student(String id, String currentClass, String address, String phone) {
8     this.id = id;
9     this.currentClass = currentClass;
10    this.address = address;
11    this.phone = phone;
12 }
```

Cách này có vẻ có khả năng giải quyết được vấn đề của chúng ta. Tuy nhiên, nếu làm như vậy, ta phải viết từng constructor cho từng trường hợp khác nhau và sẽ gặp nhiều khó khăn trong việc xác định thứ tự tham số truyền vào.

Lúc này, Builder Pattern là một cách để giải quyết khó khăn cho chúng ta.

Builder Pattern bao gồm:

- **Product**: đại diện cho đối tượng cần tạo, đối tượng này phức tạp, có nhiều thuộc tính.
- **Builder**: là abstract class hoặc interface khai báo phương thức tạo đối tượng.
- **ConcreteBuilder**: kế thừa Builder và cài đặt chi tiết cách tạo ra đối tượng. Nó sẽ xác định và nắm giữ các thể hiện mà nó tạo ra, đồng thời nó cũng cung cấp phương thức để trả các các thể hiện mà nó đã tạo ra trước đó.
- **Director**: là nơi sẽ gọi tới Builder để tạo ra đối tượng.



Builder Pattern:

- Product (Student.java)

```
public class Student {
    private String id;
    private String firstName;
    private String lastName;
    private String dayOfBirth;
    private String currentClass;
    private String phone;

    public Student(String id, String firstName, String lastName, String dayOfBirth
        , String currentClass, String phone) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.dayOfBirth = dayOfBirth;
        this.currentClass = currentClass;
        this.phone = phone;
    }
}
```

- Builder (StudentBuilder.java)

```
public interface StudentBuilder {

    StudentBuilder setId(String id);

    StudentBuilder setFirstName(String firstName);

    StudentBuilder setLastName(String lastName);

    StudentBuilder setDayOfBirth(String dayOfBirth);

    StudentBuilder setCurrentClass(String currentClass);

    StudentBuilder setPhone(String phone);

    Student build();
}
```

- ConcreteBuilder (StudentConcreteBuilder.java)

```java
public class StudentConcreteBuilder implements StudentBuilder {

    private String id;
    private String firstName;
    private String lastName;
    private String dayOfBirth;
    private String currentClass;
    private String phone;

    @Override
    public StudentBuilder setId(String id) {
        this.id = id;
        return this;
    }

    @Override
    public StudentBuilder setFirstName(String firstName) {
        this.firstName = firstName;
        return this;
    }

    @Override
    public StudentBuilder setLastName(String lastName) {
        this.lastName = lastName;
        return this;
    }

    @Override
    public StudentBuilder setDayOfBirth(String dayOfBirth) {
        this.dayOfBirth = dayOfBirth;
        return this;
    }

    @Override
    public StudentBuilder setCurrentClass(String currentClass) {
        this.currentClass = currentClass;
        return this;
    }

    @Override
    public StudentBuilder setPhone(String phone) {
        this.phone = phone;
        return this;
    }

    @Override
    public Student build() {
        return new Student(id, firstName, lastName, dayOfBirth, currentClass,
            phone);
    }
}
```

- Cuối cùng là Director, ở đây ta sẽ sử dụng ở hàm main()

```
public static void main(String[] args)
{
        StudentBuilder studentBuilder = new StudentConcreteBuilder()
                .setFirstName("Le")
                .setLastName("Hoang Phuc");
        System.out.println(studentBuilder.build());
}
```

Ta có thể viết thêm các dòng lệnh in ra thông tin của đối tượng để kiểm tra tính đúng đắn của kết quả "build" đối tượng này.

Ưu nhược điểm của Builder Pattern

- Ưu điểm
  - Giảm bớt được lượng tham số cho hàm khởi tạo đồng thời và các tham số này được cung cấp trong các lệnh gọi phương thức "dễ đọc" (highly readable). Vì thế hạn chế được việc phải gán NULL cho một số tham số tùy chọn của đối tượng.
  - Giữ cho các đối tượng luôn được khởi tạo ở trạng thái hoàn chỉnh.
  - Các đối tượng bất biến có thể được xây dựng mà không cần nhiều logic phức tạp trong quá trình xây dựng đối tượng.
- Nhược điểm
  - Để code có được tính linh hoạt và dễ đọc, số lượng dòng code phải tăng ít nhất là gấp đôi.
  - Cần phải tạo riêng từng ConcreteBuilder cho từng sản phẩm khác nhau.

2.1.4 Prototype

Định nghĩa

Prototype Pattern được sử dụng với mục đích tạo ra các đối tượng mới bằng cách sao chép (clone) các đối tượng đã có sẵn, cho phép chúng ta tạo ra các đối tượng mới chỉ với một lần khởi tạo ban đầu.

Khi muốn tạo ra một đối tượng mới, ta có thể sao chép một đối tượng đã có thay vì tạo mới từ đầu, giúp giảm thiểu thời gian và tài nguyên được sử dụng.

Cách sử dụng

Ví dụ:

Giả sử chúng ta đang viết code thư viện bằng C++ cho một tựa game lấy bối cảnh thời Tam Quốc, ta phải thiết kế hệ thống quản lý các nhân vật, và trong game lại có một số lượng lớn nhân vật thuộc các phe cánh khác nhau.

Cách thông thường:

Ta có thể kế thừa một class Character để tạo ra các class con thể hiện nhân vật ở các phe khác nhau.

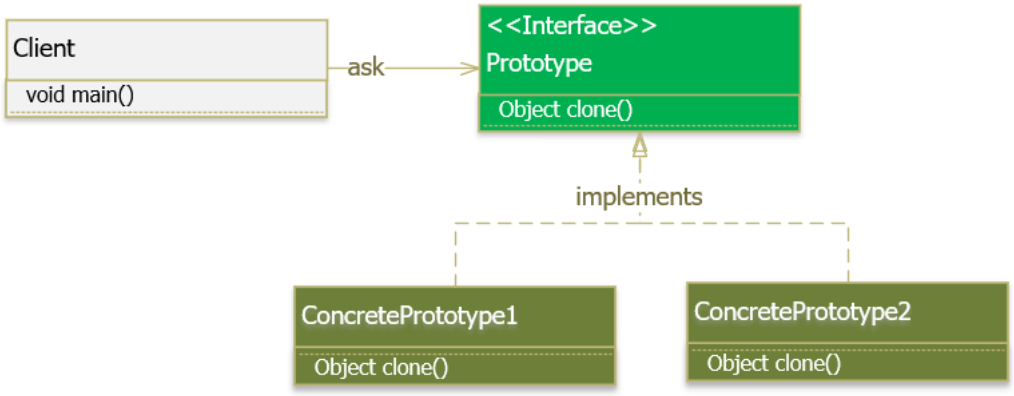
```
#include <iostream>

using namespace std;

class Character
{
protected:
    string name;
    string title;

public:
    virtual void sayHello() = 0;

    void setName(string n) {
        name = n;
    }
    string getName() {
        return name;
    }
    void setTitle(string t) {
        title = t;
    }
    string getTitle() {
        return title;
    }
};

class TaoNguy : public Character
{
public:
    TaoNguy(string n = "Unknown", string m = "Unknown") {
        name = n; title = m;
    }
    void sayHello() {
        cout << "Ta la " << name << ", " << title << " cua Nguy quoc, ta se san bang
            Thuc Han va Dong Ngo!" << endl;
    }
};

```

```
39
40 class ThucHan : public Character
41 {
42 public:
43     ThucHan(string n = "Unknown", string m = "Unknown") {
44         name = n; title = m;
45     }
46     void sayHello() {
47         cout << "Ta la " << name << ", " << title << " cua nuoc Thuc, ta tuan menh Thuc
               Vuong thao phat nghich tac, trung hung Dai Han!" << endl;
48     }
49 };
50
51 class DongNgo : public Character
52 {
53 public:
54     DongNgo(string n = "Unknown", string m = "Unknown") {
55         name = n; title = m;
56     }
57     void sayHello() {
58         cout << "Ta la " << name << ", " << title << " cua Dong Ngo, ta the chet bao ve
               lanh tho nuoc Ngo!" << endl;
59     }
60 };
61
62 int main()
63 {
64     TaoNguy* TaoPhi = new TaoNguy();
65     TaoPhi->setName("Tao Phi");
66     TaoPhi->setTitle("Vuong");
67
68     ThucHan* GiaCatLuong = new ThucHan();
69     GiaCatLuong->setName("Gia Cat Luong");
70     GiaCatLuong->setTitle("Thua Tuong");
71
72     DongNgo* TonQuyen = new DongNgo();
73     TonQuyen->setName("Ton Quyen");
74     TonQuyen->setTitle("Vuong");
75
76     TaoNguy* TuMaY = new TaoNguy();
77     TuMaY->setName("Tu Ma Y");
78     TuMaY->setTitle("Dai Do Doc");
79
80     TaoPhi->sayHello();
81     GiaCatLuong->sayHello();
82     TonQuyen->sayHello();
83     TuMaY->sayHello();
84
85     delete GiaCatLuong;
86     delete TonQuyen;
87     delete TaoPhi;
88     delete TuMaY;
89
90     return 0;
91 }
```

Ngoài cách trên, ta có thể khai thác tính đa hình để có thể dễ dàng quản lý các nhân vật bằng cách sử dụng Prototype Pattern.

Các thành phần trong Prototype Pattern:

- **Prototype**: khai báo một class, interface hoặc abtract class cho việc clone chính nó.
- **ConcretePrototype**: thực thi interface (hoặc kế thừa từ abstract) được cung cấp bởi Prototype để nhân bản chính bản thân nó. Các class này chính là thể hiện cụ thể phương thức clone(); có thể không cần thiết nếu như Prototype là một class và nó đã implement việc clone chính nó.
- **Client**: tạo mới đối tượng bằng cách gọi Prototype thực hiện clone chính nó.



Prototype Pattern:

- Prototype

```
#include <iostream>

using namespace std;

class Character {
protected:
    string name;
    string title;

public:
    virtual void sayHello() = 0;
    virtual Character* clone() = 0;
    void setName(string n) {
        name = n;
    }
    string getName() {
        return name;
    }
    void setTitle(string t) {
        title = t;
    }
    string getTitle() {
        return title;
    }
};
```

- ConcretePrototype

```
class TaoNguy : public Character {
public:
    TaoNguy(string n = "Unknown", string m = "Unknown") {
        name = n; title = m;
    }
    void sayHello() {
        cout << "Ta la " << name << ", " << title << " cua Nguy quoc, ta se san
            bang Thuc Han va Dong Ngo!" << endl;
    }
    Character* clone() {
        return new TaoNguy(*this);
    }
};

class ThucHan : public Character {
public:
    ThucHan(string n = "Unknown", string m = "Unknown") {
        name = n; title = m;
    }
    void sayHello() {
        cout << "Ta la " << name << ", " << title << " cua nuoc Thuc, ta tuan menh
             Thuc Vuong thao phat nghich tac, trung hung Dai Han!" << endl;
    }
    Character* clone() {
        return new ThucHan(*this);
    }
};
class DongNgo : public Character {
public:
    DongNgo(string n = "Unknown", string m = "Unknown") {
        name = n; title = m;
    }
    void sayHello() {
        cout << "Ta la " << name << ", " << title << " cua Dong Ngo, ta the chet
            bao ve lanh tho nuoc Ngo!" << endl;
    }
    Character* clone() {
        return new DongNgo(*this);
    }
};
```

- Client

```
int main() {
    TaoNguy* nguy = new TaoNguy();
    ThucHan* thuc = new ThucHan();
    DongNgo* ngo = new DongNgo();

    Character* TaoPhi = nguy->clone();
    TaoPhi->setName("Tao Phi");
    TaoPhi->setTitle("Vuong");

    Character* GiaCatLuong = thuc->clone();
    GiaCatLuong->setName("Gia Cat Luong");
    GiaCatLuong->setTitle("Thua Tuong");

    Character* TonQuyen = ngo->clone();
    TonQuyen->setName("Ton Quyen");
    TonQuyen->setTitle("Vuong");

    Character* TuMaY = nguy->clone();
    TuMaY->setName("Tu Ma Y");
    TuMaY->setTitle("Dai Do Doc");

    TaoPhi->sayHello();
    GiaCatLuong->sayHello();
    TonQuyen->sayHello();
    TuMaY->sayHello();

    delete thuc;
    delete GiaCatLuong;

    delete ngo;
    delete TonQuyen;

    delete nguy;
    delete TaoPhi;
    delete TuMaY;

    return 0;
}
```

Ở đây, các nhân vật được tạo mới bằng cách "clone" theo loại nhân vật của các lãnh thổ khác nhau. Class Character có thể quản lý tất cả các nhân vật ở bất cứ phe nào, đồng thời các nhân vật ở phe khác nhau đều có đặc điểm riêng của từng phe.

Kết quả chạy của cả hai hướng tiếp cận

```
Ta la Tao Phi, Vuong cua Nguy quoc, ta se san bang Thuc Han va Dong Ngo!
Ta la Gia Cat Luong, Thua Tuong cua nuoc Thuc, ta tuan menh Thuc Vuong thao phat nghich tac, trung hung Dai Han!
Ta la Ton Quyen, Vuong cua Dong Ngo, ta the chet bao ve lanh tho nuoc Ngo!
Ta la Tu Ma Y, Dai Do Doc cua Nguy quoc, ta se san bang Thuc Han va Dong Ngo!
```

Ưu nhược điểm của Prototype Pattern

- Ưu điểm
  - Có thể "clone" các đối tượng mà không cần phải nối với class cụ thể của chúng.
  - Tránh được việc lặp đi lặp lại code khi muốn clone đối tượng đã có sẵn.
  - Giảm bớt thời gian khi xây dựng các đối tượng phức tạp.
- Nhược điểm
  - Phải cẩn thận khi clone các đối tượng có tham chiếu xoay vòng (circular references).

2.1.5 Singleton

Định nghĩa

Singleton là một Design Pattern rất phổ biến trong lập trình hướng đối tượng. Singleton được sử dụng để đảm bảo có một và chỉ một đối tượng duy nhất của class được tạo ra.

Cách sử dụng

Các bước hiện thực một class Singleton:

- Đưa tất cả constructor về private
- Xóa hết mọi copy constructor (constructor tạo ra một đối tượng mới và phân biệt với đối tượng cùng kiểu có sẵn)
- Tạo một con trỏ private static trỏ vào cùng đối tượng của class
- Thêm một hàm public trả về con trỏ đối tượng Singleton duy nhất

Singleton Pattern:

```
#include <iostream>

using namespace std;

class Singleton {
    static Singleton *instance;
    int data;

    // Private constructor
    Singleton() {
        data = 0;
    }

public:
    static Singleton *getInstance();

    int getData() {
        return this->data;
    }

    void setData(int data) {
        this->data = data;
    }
};

Singleton * Singleton::getInstance() {
    if (!instance) {
    instance = new Singleton;
    cout << "Create new instance\n";
    }
    else cout << "Instance has been created already\n";
    return instance;
}

// Initialize pointer to zero so that
// it can be initialized in first call to getInstance
Singleton *Singleton::instance = 0;

```

```
39 int main() {
40     Singleton *s = s->getInstance();
41     cout << s->getData() << endl;
42     s->setData(100);
43     cout << s->getData() << endl;
44     Singleton *s1 = s1->getInstance();
45     cout << s1->getData() << endl;
46     return 0;
47 }
```

Kết quả chạy

```
Creating new instance
0
100
Instance has been created already
100
```

Theo như kết quả chạy, chỉ có một đối tượng Singleton được tạo ra để lưu một số nguyên trong suốt quá trình chạy, chương trình không thể tạo thêm bất cứ đối tượng Singleton mới nào.

Tuy nhiên, trên đây chỉ là phiên bản dành cho lập trình tuần tự. Nếu đưa vào chạy song song, đa luồng thì khó tránh khỏi xung đột giữa các luồng (race condition). Để làm điều đó, ta chỉ cần điều chỉnh một chút.

Singleton phiên bản thread-safety

Khi Singleton trong môi trường đa luồng, để tránh race condition ta chỉ cần thêm khóa mutex để bảo vệ các thread khi thao tác với Singleton như sau:

```
1  #include <mutex>
2
3  mutex instance_mtx;
4  Singleton * Singleton::getInstance() {
5      instance_mtx.lock();
6      if (!instance) {
7          instance = new Singleton;
8          cout << "Create new instance\n";
9      }
10     else cout << "Instance has been created already\n";
11     instance_mtx.unlock();
12     return instance;
13 }
```

Để kiểm tra thử Singleton Pattern trong môi trường đa luồng, ta cần một hàm sử dụng Singleton trong thread (ta cũng sử dụng thêm mutex để đảm bảo không vướng phải race condition).

```
1  #include <thread>
2
3  mutex worker_mtx;
4  void worker(int times) {
5      worker_mtx.lock();
6      Singleton * s = s->getInstance();
7      for (int i=0 ; i<times; i++) s->setData(s->getData()+1);
8      cout << "Worker() added " << times << ", singleton's data became "
9      << s->getData() << endl;
10     worker_mtx.unlock();
11 }
```

Và sau đây là các bước thiết lập thread và cho chúng chạy song song với nhau.

```
#define MAX_THREADS 10
#define MAX_LOOPS 100000000

int main() {
    thread * t[MAX_THREADS];
    for (int i=0; i<MAX_THREADS; i++) t[i] = new thread(worker, MAX_LOOPS);
    for (int i=0; i<MAX_THREADS; i++) t[i]->join();

    Singleton * s = s->getInstance();
    cout << "Final singleton's data: " << s->getData() << endl;
    for (int i=0; i<MAX_THREADS; i++) delete t[i];
    return 0;
}
```

Kết quả chạy (trên Linux)

```
Create new instance
Worker() added 100000000, singleton's data became 100000000
Instance has been created already
Worker() added 100000000, singleton's data became 200000000
Instance has been created already
Worker() added 100000000, singleton's data became 300000000
Instance has been created already
Worker() added 100000000, singleton's data became 400000000
Instance has been created already
Worker() added 100000000, singleton's data became 500000000
Instance has been created already
Worker() added 100000000, singleton's data became 600000000
Instance has been created already
Worker() added 100000000, singleton's data became 700000000
Instance has been created already
Worker() added 100000000, singleton's data became 800000000
Instance has been created already
Worker() added 100000000, singleton's data became 900000000
Instance has been created already
Worker() added 100000000, singleton's data became 1000000000
Instance has been created already
Final singleton's data: 1000000000
```

Ưu nhược điểm của Singleton Pattern

- Ưu điểm
  - Ai cũng có thể truy cập được instance của singleton, có thể gọi ở bất cứ đâu.
  - Singleton vượt trội hơn static class bởi các yếu tố như hỗ trợ giao diện hay hỗ trợ kế thừa.
- Nhược điểm
  - Làm tăng kết nối giữa các script và điều này thì không tốt, các scripts con phụ thuộc quá nhiều vào các singletons và khi singletons thay đổi có thể gây ra bug hoặc lỗi.
  - Không sử dụng được tính đa hình.

2.2 Structural Patterns

Structural Patterns là nhóm các Patterns phục vụ việc lắp ráp các class thành các cấu trúc lớn hơn mà vẫn đảm bảo tính linh hoạt và hiệu quả. Nhóm gồm này gồm 7 mẫu: **Adapter**, **Bridge**, **Composite**, **Decorator**, **Facade**, **Flyweight**, **Proxy**.

Các class Structural Patterns sử dụng tính kế thừa để tổng hợp interfaces hoặc implementations (code triển khai, code hiện thực). Structural Patterns đặc biệt hữu dụng khi muốn các class trong thư viện có thể phát triển một cách độc lập với nhau, hoặc khi muốn cung cấp một abstraction thống nhất cho các interface khác nhau. Ví dụ như cách mà tính kế thừa gom hai hoặc nhiều class lại làm một, kết quả là một class sở hữu những thuộc tính của những class cha của nó; cách này được sử dụng khi cần cho nhiều class riêng biệt hoạt động chung với nhau. Một ví dụ khác là cách triển khai của Adapter Pattern. Adapter Pattern làm cho một interface (của Adaptee) phù hợp với interface khác, nhờ đó cung cấp cho các interface khác nhau một sự thống nhất một cách trừu tượng.

Ngoài tổng hợp interface và implementation, các đối tượng Structural Pattern mô tả những cách tạo đối tượng để hiện thực những chức năng mới. Tính linh hoạt được bổ sung vào các thành phần đối tượng xuất phát từ khả năng thay đổi thành phần trong run-time, điều mà các thành phần static class không thể làm.

Composite Pattern là một ví dụ Structural Patterns mô tả cách dựng một cấu trúc phân cấp class tạo bởi các class dùng cho hai dạng đối tượng: nguyên thủy (primitive) và hỗn hợp (composite). Các đối tượng hỗn hợp cho phép kết hợp các đối tượng nguyên thủy và các đối tượng hỗn hợp khác thành các cấu trúc phức tạp tùy ý. Đối với Proxy Pattern, một Proxy hành động như một vật thay thế (surrogate) hoặc vật giữ chỗ (placeholder) cho một đối tượng khác. Proxy có thể được dùng bằng nhiều cách khác nhau; Proxy có thể đóng vai trò là một đại diện cục bộ (local local representative) cho một đối tượng ở xa xôi, có thể đại diện cho một đối tượng lớn mà cần nhiều thời gian để tải, có thể bảo vệ truy cập vào những đối tượng nhạy cảm. Các Proxy có thể hạn chế, tăng cường hoặc thay đổi các thuộc tính cụ thể của các đối tượng.

Flyweight Pattern định nghĩa một cấu trúc để chia sẻ các đối tượng. Các đối tượng được chia sẻ do một số lý do như: hiệu quả (efficiency) và nhất quán (consistency). Flyweight tập trung vào việc chia sẻ để đạt hiệu quả về không gian. Các ứng dụng sử dụng nhiều đối tượng phải chú ý cẩn thận đến chi phí của từng đối tượng. Chia sẻ đối tượng tiết kiệm được đáng kể so với sao chép chúng.

Facade Pattern giúp tạo một đối tượng duy nhất đại diện cho toàn bộ hệ thống con (subsystem). Một Facade là một đại diện cho một tập hợp các đối tượng. Facade thực hiện trách nhiệm của mình bằng cách chuyển tiếp các thông điệp đến các đối tượng mà nó đại diện. Bridge Pattern tách phần trừu tượng (abstraction) của đối tượng khỏi phần triển khai (implementation) của nó để bạn có thể thay đổi chúng một cách độc lập.

Decorator Pattern diễn tả cách bổ sung động (dynamically) các chức năng cho đối tượng. Decorator Pattern tổng hợp các đối tượng bằng đệ quy, cho phép sự bổ sung chức năng, trách nhiệm không giới hạn. Các đối tượng Decorator được bổ sung chức năng bằng cách lồng các Decorator lại với nhau. Để làm được điều đó, mỗi đối tượng Decorator trong hệ thống phải có interface phù hợp với nhau và phải truyền được thông điệp cho nhau.

Phần mềm bị thay đổi tính năng là điều xảy ra như "cơm bữa". Nếu không muốn mỗi lần thay đổi là một lần phải "đập đi xây lại cả hệ thống", thì chúng ta nên có cách tổ chức linh hoạt để có thể thực hiện sự thay đổi dễ dàng hơn.

Các đoạn code mẫu cho Structural Pattern có ở đường link sau:
https://github.com/PhucLe03/OOP/tree/main/Design%20Patterns/Structural%20Patterns
hoặc mã QR bên dưới.

2.2.1 Adapter

Định nghĩa

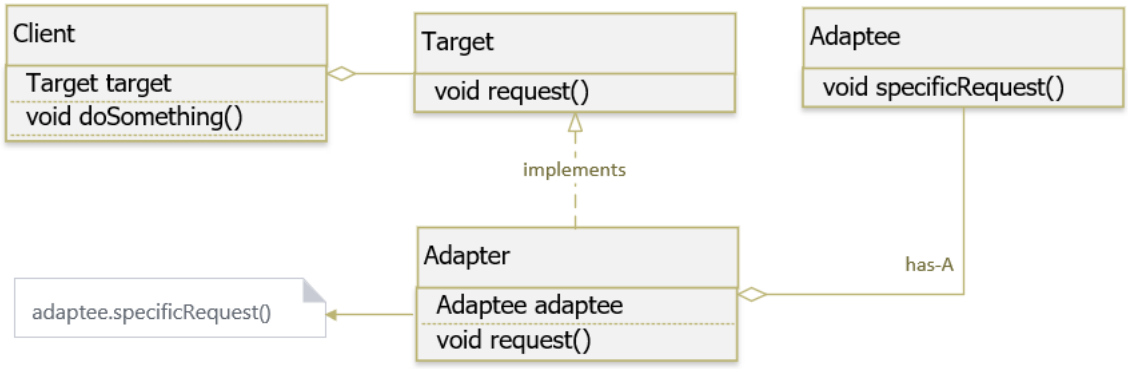
Adapter Pattern cũng có chức năng tương tự như Adapter trong đời sống. Adapter Pattern là một đối tượng trung gian giữa hai giao diện không tương thích với nhau, cho phép các giao diện đó làm việc với nhau. Sử dụng Adapter Pattern giúp ta không cần phải chỉnh sửa class có sẵn cũng như class đang viết.

Ngoài ra, Adapter Pattern còn gọi là Wrapper Pattern do cung cấp một giao diện "bọc ngoài" tương thích cho một hệ thống có sẵn, có dữ liệu và hành vi phù hợp nhưng có giao diện không tương thích với class đang viết.

Cách sử dụng

Các thành phần trong Adapter Pattern:

- **Adaptee**: định nghĩa interface không tương thích, cần được tích hợp vào.
- **Adapter**: tích hợp, giúp interface không tương thích tích hợp được với interface đang làm việc. Thực hiện việc chuyển đổi interface cho Adaptee và kết nối Adaptee với Client.
- **Target**: một interface chứa các chức năng được sử dụng bởi Client (domain specific).
- **Client**: sử dụng các đối tượng có interface Target.



Ví dụ:

Giả sử ta có một chiếc laptop sản xuất tại Mỹ, có hiệu điện thế hiệu dụng là 120V và phích cắm 3 chân, nhưng chúng ta muốn sử dụng bằng nguồn điện 220V với ổ cắm 2 chân của Việt Nam, như vậy ta không còn cách nào khác ngoài việc phải xài một bộ adapter chuyển đổi điện áp để laptop không bị hỏng. Ta có thể mô phỏng Adapter Pattern qua ví dụ này bằng C++ như sau:

Adapter Pattern:

Các nguồn điện Việt Nam và Mỹ:

```
class GlobalSocketInterface {
public:
    virtual int voltage() = 0;
};
class VNSocketInterface : public GlobalSocketInterface {
public:
    int voltage() {return 220;}

    virtual int daynong() {return 1;}
    virtual int daynguoi() {return -1;}
};
class USSocketInterface : public GlobalSocketInterface {
public:
    int voltage() {return 120;}

    virtual int daynong() {return 1;}
    virtual int daynguoi() {return -1;}
    virtual int daythuba() {return 0;}
};
```

Chiếc laptop sản xuất tại Mỹ:

```
class USLaptop {
public:
    void plugin(USSocketInterface * powersupply) {this->powersource = powersupply;}
    void run() {
        if (powersource->voltage()>120) {
            cout << "Boom!\n"; return;
        }
        else if (powersource->daynong()==1 && powersource->daynguoi()==-1 &&
            powersource->daythuba()==0) {
            cout << "Chay tot, khong van de gi!\n"; return;
        }
    }
private:
    USSocketInterface * powersource;
};
```

Như vậy, chiếc laptop hiện chỉ có thể sử dụng với nguồn điện và ổ điện của Mỹ. Để dùng ở Việt Nam, ta cần thêm adapter như sau:

```
class Adapter : public USSocketInterface {
public:
    void plugin(VNSocketInterface * outlet) {this->socket = outlet;}
    int voltage() {return 110;}
    int daynong() {return socket->daynong();}
    int daynguoi() {return socket->daynguoi();}
    int daythuba() {return 0;}
private:
    VNSocketInterface * socket;
};
```

Vậy là chúng ta có thể sử dụng laptop với nguồn điện của Việt Nam thông qua adapter:

```
int main()
{
    VNSocketInterface* dienVN = new VNSocketInterface;
    USSocketInterface* dienUS = new USSocketInterface;
    USLaptop* laptop = new USLaptop;

    cout << "---Ket noi laptop voi nguon dien cua US---\n";
    laptop->plugin(dienUS);
    laptop->run();
    cout << "---Ket noi laptop voi nguon dien cua VN su dung Adapter---\n";
    Adapter* adapter = new Adapter;
    adapter->plugin(dienVN);
    laptop->plugin(adapter);
    laptop->run();

    delete dienVN;
    delete adapter;
    delete dienUS;
    delete laptop;

    return 0;
}
```

Kết quả chạy

—Ket noi laptop voi nguon dien cua US—
Chay tot, khong van de gi!
—Ket noi laptop voi nguon dien cua VN su dung Adapter—
Chay tot, khong van de gi!

Tới đây, ta có thể thấy rất rõ ràng việc dùng adapter giúp người lập trình không cần phải thay đổi code cũ mà vẫn tương tác, kết nối được các giao diện dù không tương thích với nhau.

Ưu nhược điểm của Adapter Pattern

- Ưu điểm
  - Cho phép nhiều đối tượng có giao diện giao tiếp khác nhau có thể tương tác và giao tiếp với nhau.
  - Tăng khả năng sử dụng lại thư viện mà không cần phải thay đổi giao diện.
- Nhược điểm
  - Tất cả các yêu cầu được chuyển tiếp, do đó làm tăng thêm một ít chi phí.
  - Đôi khi có quá nhiều Adapter được thiết kế trong một chuỗi Adapter (Adapter chain) trước khi đến được yêu cầu thực sự.

2.2.2 Bridge

Định nghĩa

Bridge Pattern được định nghĩa là kỹ thuật tách tính trừu tượng (abstraction) ra khỏi thực thể (implementation) của nó để cho chúng có thể hiệu chỉnh một cách độc lập.

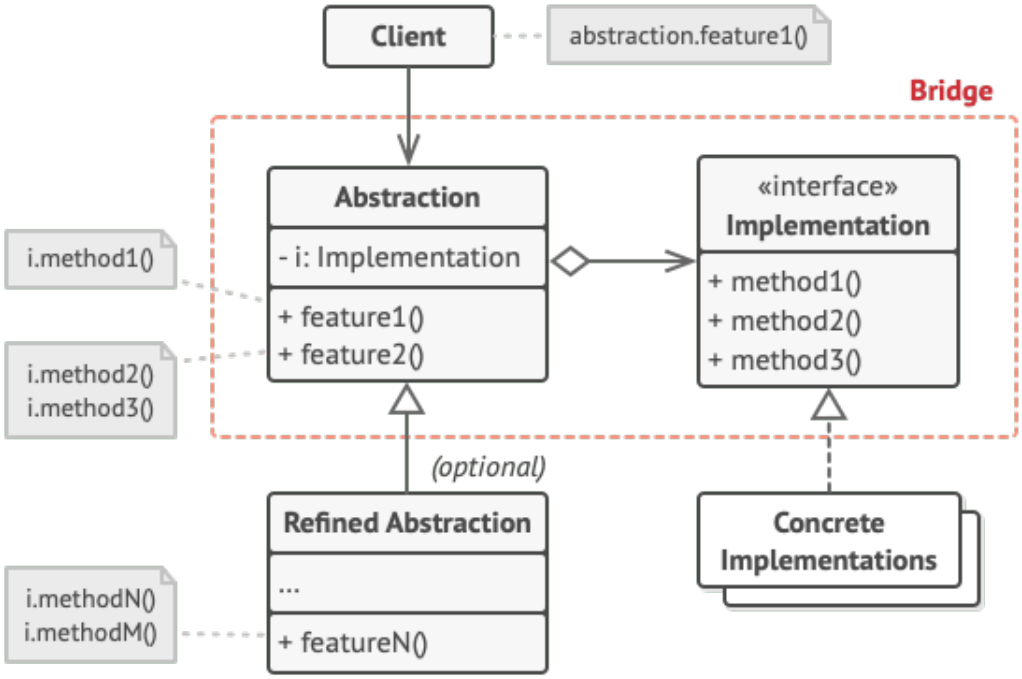
Thường thì Bridge Pattern được sử dụng trong các hệ thống có nhiều chức năng khác nhau nên sẽ có nhiều loại phần trừu tượng hóa cần được thực hiện. Với Bridge Pattern, các loại phần trừu tượng có thể được triển khai độc lập và chúng hoạt động kết hợp với nhau thông qua một interface chung. Bridge Pattern cho phép 2 lớp có các các đặc tính khác nhau tương tác với nhau một cách linh hoạt ngay cả trong runtime.

Ví dụ như chức năng in ấn áp dụng cho nhiều loại tài liệu như word, pdf, excel... và được dùng cho các loại máy in khác nhau như máy in trắng đen, máy in màu... Ở đây, tài liệu được coi là trừu tượng, còn máy in là thực thể. Bridge Pattern cho phép chúng ta phát triển tài liệu và hệ thống in ấn một cách độc lập mà không ảnh hưởng đến nhau.

Cách sử dụng

Bridge Pattern bao gồm:

- **Abstraction**: định nghĩa interface của abstract class, quản lý việc tham chiếu đến đối tượng hiện thực cụ thể.
- **RefinedAbstraction**: những class kế thừa Abstraction.
- **Implementation**: định nghĩa interface cho các lớp hiện thực. Thông thường nó là interface định ra các tác vụ nào đó của Abstraction.
- **ConcreteImplementation**: kế thừa Implementation và định nghĩa chi tiết hàm thực thi.



Xét một bài toán sau:

Trên hệ thống phân phối game của một công ty X đang muốn triển khai các phương thức thanh toán game bằng thẻ Visa và Momo. Vấn đề là hai phương thức thanh toán này có tính chất khác hẳn nhau từ công đoạn đăng ký cho tới lúc thanh toán, cần có giải pháp để quản lý chung hai phương thức này.

Sau đây là mẫu chương trình C++ mô phỏng quá trình thanh toán game bằng hai phương thức trên.

Bridge Pattern:

Đầu tiên là interface thể hiện phương thức thanh toán.

```
#include <iostream>
#include <string>

using namespace std;

class PaymentMethod
{
public:
    virtual void pay(int amount) = 0;
};
```

Hai phương thức thanh toán bằng Visa và Momo sẽ kế thừa abstract class trên.

```
class Visa : public PaymentMethod
{
public:
    Visa(string cardNumber, string expDate, string cvv) {
        this->cardNumber = cardNumber;
        this->expDate = expDate;
        this->cvv = cvv;
    }
    void pay(int amount) {
        cout << "Paid an amount of " << amount
        << " with Visa card " << cardNumber << endl;
    }
private:
    string cardNumber;
    string expDate;
    string cvv;
};

class Momo : public PaymentMethod
{
public:
    Momo(string phoneNumber) {this->phoneNumber = phoneNumber;}
    void pay(int amount) {
        cout << "Paid an amount of " << amount
        << " with Momo account " << phoneNumber << endl;
    }
private:
    string phoneNumber;
};
```

Sau khi đã triển khai phương thức thanh toán, ta sẽ hiện thực class thể hiện sản phẩm được bán, cụ thể ở ví dụ này là game.

```
class Product
{
public:
    void choosePaymentMethod(PaymentMethod * paymentMethod) {
        this->paymentMethod = paymentMethod;
    }
    void buy() {paymentMethod->pay(price);}
protected:
    PaymentMethod * paymentMethod;
    int price;
};

class Game : public Product
{
public:
    Game(int price) {this->price = price;}
};
```

Trong hàm main() dưới đây, ta sẽ mô phỏng trường hợp một khách hàng đang muốn mua hai tựa game là Skyrim và Call of Duty, mỗi game sẽ thanh toán bằng một phương thức khác nhau.

```
int main()
{
    Visa * visa = new Visa("1234.5678.xxx","03/05","123");
    Momo * momo = new Momo("0944681103");

    Game * skyrim = new Game(1000);
    skyrim->choosePaymentMethod(visa);
    skyrim->buy();

    Game * callofduty = new Game(2000);
    callofduty->choosePaymentMethod(momo);
    callofduty->buy();

    delete visa;
    delete momo;
    delete skyrim;
    delete callofduty;

    return 0;
}
```

Kết quả chạy

```
Paid an amount of 1000 with Visa card 1234.5678.xxx
Paid an amount of 2000 with Momo account 0944681103
```

Ưu nhược điểm của Bridge Pattern

- Ưu điểm
  - Giảm sự phục thuộc giữa abstraction và implementation (loose coupling).
  - Giảm số lượng những lớp con không cần thiết, code gọn gàn hơn và kích thước ứng dụng nhỏ hơn.
  - Dễ bảo trì, dễ mở rộng.
  - Cho phép ẩn các chi tiết hiện thực từ client.
- Nhược điểm
  - Có thể làm tăng độ phức tạp khi áp dụng cho một lớp có tính gắn kết cao.

2.2.3 Composite

Định nghĩa

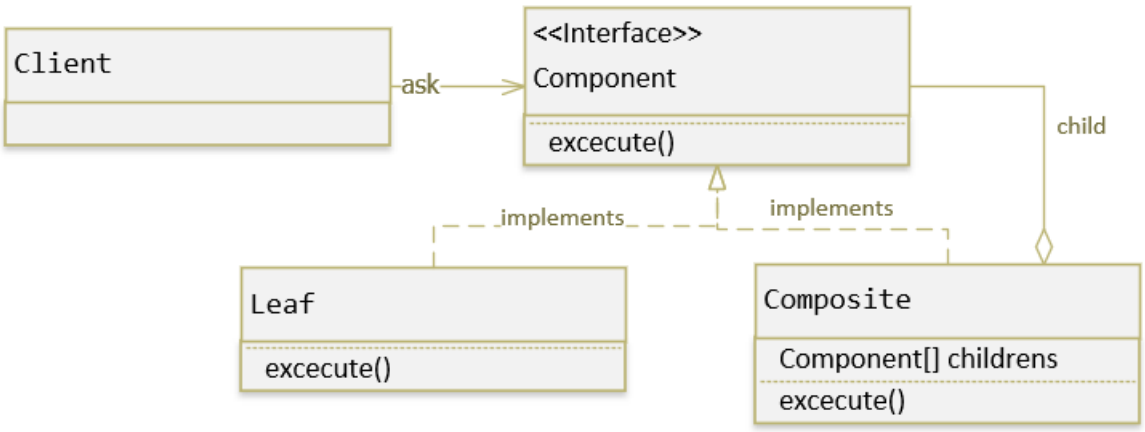
Composite Pattern là phương pháp thiết kế các đối tượng thành một cấu trúc dạng cây và các đối tượng này hoạt động như thể một đối tượng đơn lẻ.

Composite đã trở thành một giải pháp khá phổ biến cho hầu hết các vấn đề yêu cầu xây dựng cấu trúc cây. Tính năng tuyệt vời của Composite là khả năng chạy các phương thức đệ quy trên toàn bộ cấu trúc cây và tổng hợp kết quả.

Cách sử dụng

Các thành phần trong Composite Pattern:

- **Client**: đại diện cho class, hàm nơi mà sẽ làm việc trực tiếp với các đối tượng trong Composite.
- **Component**: là Interface định nghĩa là phương thức chung cho các đối tượng tham gia vào mẫu thiết kế này
- **Leaf**: là đơn vị nhỏ nhất trong Composite, đồng thời cũng là 1 thể hiện của Component
- **Composite**: là 1 instance của Component, trong Composite chứa nhiều đơn vị nhỏ hơn là Leaf.



Composite Pattern thường được sử dụng để biểu thị thứ bậc của các thành phần giao diện người dùng hoặc chương trình hoạt động với đồ thị.

Nếu chúng ta có cây đối tượng và mỗi đối tượng của cây là một phần của cùng một hệ thống phân cấp lớp, thì đây rất có thể là một Composite Pattern.

Cũng với cây đối tượng đó, nếu các phương thức của các lớp ủy thác công việc cho các đối tượng con của cây và thực hiện nó thông qua lớp cơ sở hay giao diện của hệ thống phân cấp, thì đây chắc chắn là một Composite Pattern.

Ví dụ:

Giả sử chúng ta cần viết một chương trình quản lý nhân viên của một phòng ban trong công ty. Xét một phòng ban, ta thấy một phòng sẽ bao gồm nhiều nhân viên, có thể áp dụng hệ thống phân cấp để quản lý dữ liệu, vậy composite sẽ là một giải pháp. Dưới đây là một mẫu code C++ quản lý nhân viên.

Composite Pattern:

- Component

```cpp
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class Employee {
public:
    string getDestination() {return destination;}
    virtual void printEmployeeDetails() {
        cout << "Name: " << name << ", ID: " << ID << ", destination: " <<
            destination << endl;
    }
protected:
    string name;
    int ID;
    string destination;
};
```

- Leaf

```cpp
class Manager: public Employee {
public:
    Manager(string name, int ID) {
        this->name = name;
        this->ID = ID;
        this->destination = "Manager";
    }

};

class Developer: public Employee {
public:
    Developer(string name, int ID) {
        this->name = name;
        this->ID = ID;
        this->destination = "Developer";
    }
};
```

- Composite

```
class Directory : public Employee {
public:
    Directory() = default;
    void addEmployee(Employee* employee) {
        if (employee->getDestination() == "Manager") {
            bossList.push_back(employee);
        }
        else employeeList.push_back(employee);
    }
    void printEmployeeDetails() {
        for (size_t i=0; i<bossList.size(); i++)
            bossList[i]->printEmployeeDetails();
        for (size_t i=0; i<employeeList.size(); i++)
            employeeList[i]->printEmployeeDetails();
    }
private:
    vector<Employee*> employeeList;
    vector<Employee*> bossList;
};

class Department : public Directory {
public:
    Department(string n) {name = n;}
    void printName() {
        cout << "Department name: " << name << endl;
    }
};
```

- Client

```
int main() {
    Employee * lhp = new Developer("Le Hoang Phuc", 1);
    Employee * hpl = new Developer("Hoang Phuc Le", 2);
    Employee * plh = new Developer("Phuc Le Hoang", 3);

    Employee * boss = new Manager("Boss", 0);

    Department * hailua03 = new Department("HaiLua03");
    hailua03->addEmployee(lhp);
    hailua03->addEmployee(hpl);
    hailua03->addEmployee(plh);
    hailua03->addEmployee(boss);

    hailua03->printEmployeeDetails();

    delete hailua03;
    delete lhp;
    delete hpl;
    delete plh;
    delete boss;

    return 0;
}
```

Kết quả chạy

```
Name: Boss, ID: 0, destination: Manager
Name: Le Hoang Phuc, ID: 1, destination: Developer
Name: Hoang Phuc Le, ID: 2, destination: Developer
Name: Phuc Le Hoang, ID: 3, destination: Developer
```

Chương trình trên sử dụng một abstract class Employee (chỉ nhân viên chung); các class Manager (chỉ trưởng phòng), Developer (chỉ nhà phát triển), Directory (chỉ tập hợp nhân viên chung) đều kế thừa class Employee này, còn class Department (chỉ phòng ban) kế thừa class Directory. Vậy, dữ liệu phòng ban và nhân viên được cấu trúc thành dạng cây có phân cấp bậc và có thể triển khai các chức năng theo cơ chế đệ quy trên cây.

Mở rộng thêm, ta có thể viết thêm các class cấp cao hơn chẳng như chi nhánh (Branch) để quản lý các phòng ban tại một chi nhánh nhỏ hay trụ sở chính (HeadQuarter) để quản lý các chi nhánh đó dựa trên mẫu Composite này.

Ưu nhược điểm của Composite Pattern

- Ưu điểm
  - Kết hợp với đa hình và đệ quy, dễ dàng làm việc với cấu trúc cây phức tạp.
  - Có thể thêm mới các class, đối tượng mới vào cây hệ thống mà không phá vỡ cấu trúc hiện có của nó.
- Nhược điểm
  - Khi một giao diện có nhiều chức năng phức tạp, sử dụng pattern có thể khiến code trở nên khó hiểu.

2.2.4 Decorator

Định nghĩa

Decorator Pattern hỗ trợ người lập trình đính kèm động (dynamically attach) các chức năng bổ sung vào một đối tượng có sẵn. Cách làm này giúp thay thế cho việc phải kế thừa quá mức dẫn đến việc khó quản lý đối tượng.

Giống như tên gọi, Decorator sẽ trang trí đối tượng của chúng ta bằng "giấy bọc" (wrapping) xung quanh đối tượng, các wrappings tượng trưng cho các chức năng mà ta muốn bổ sung cho đối tượng.

Cách sử dụng

Xét bài toán sau:

Một quán trà sửa mới mở đang muốn thiết lập một hệ thống thanh toán tự động. Khi khách hàng mua trà sữa, sẽ có người yêu cầu thêm toppings như trân châu, thạch... và toppings sẽ tính vào hóa đơn khách hàng.

Các phương án hiện thực:

Ta thử xét cách làm sau: đối với từng loại topping, ta sẽ tạo ra một class riêng, nếu trà sữa có hai loại toppings, thì ta sẽ tạo riêng một class cho trường hợp này. Nếu như quán mở rộng và thêm một lượng lớn các loại toppings vào menu, thì ta phải tạo thêm biết bao nhiêu class nếu như tiếp tục phương án này. Như vậy, cách tạo riêng sẽ gây ra nhiều bất lợi.

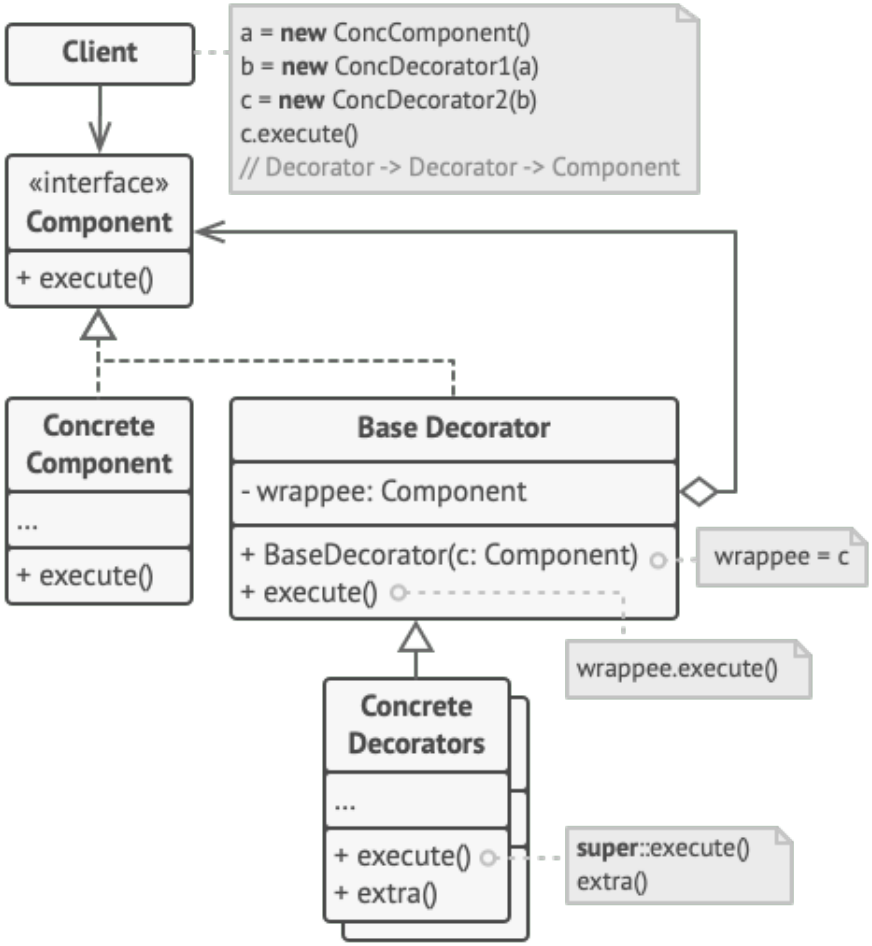
Xét thêm cách này: Khi tạo abstract class cho đối tượng là đồ uống, ta đính kèm thêm các boolean (true/false) cho toppings, khi tra cứu, ta sẽ duyệt qua các boolean này và sẽ có được kết quả như mong muốn. Nghe có vẻ hợp lý, ta thử hiện thực cách này (các hàm getDesc() và getCost() dưới đây dùng để lấy thông tin đồ uống và giá tiền).

```
#include <iostream>

using namespace std;

class DoUong
{
public:
    virtual string getDesc() = 0;
    virtual float getCost() = 0;
private:
    bool tranchau;
    bool thach;
    bool pudding;
};
```

Giả sử quán bán thêm trà dâu, trà dâu thì có các loại toppings như dâu tây, dâu tằm... vậy dãy boolean sẽ có thêm những loại toppings mới cùng với rất nhiều dòng if-else của các boolean này khi lấy thông tin hay lấy giá tiền. Khi chúng ta tạo đối tượng trà sữa, phải đặt false cho toppings trà dâu và ngược lại, vì có ai uống trà sữa mà có topping dâu đâu. Nếu như quán mà mở rộng nữa thì sẽ rất khó quản lý hết các toppings này. Chúng ta cần một phương pháp tối ưu hơn, đó là sử dụng Decorator Pattern.

Các thành phần trong Decorator Pattern:

- **Component**: là một interface quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.
- **ConcreteComponent**: hiện thực các phương thức của Component.
- **Decorator**: là một abstract class dùng để duy trì một tham chiếu của đối tượng Component và đồng thời hiện thực các phương thức của Component.
- **ConcreteDecorator**: hiện thực các phương thức của Decorator, nó cài đặt thêm các tính năng mới cho Component.
- **Client**: đối tượng sử dụng Component với những yêu cầu mở rộng đính kèm.



Decorator Pattern:

Interface đồ uống của chúng ta sẽ chỉnh lại một chút từ đoạn code trên.

```
#include <iostream>

using namespace std;

class DoUong {
public:
    virtual string getDesc() = 0;
    virtual float getCost() = 0;
};
```

Đối với từng loại đồ uống, ta sẽ thực hiện theo mẫu sau:

```
class TraSua : public DoUong {
public:
    string getDesc() {return "Tra sua";}
    float getCost() {return 15.0;}
};
```

Điểm mấu chốt ở đây là wrapping, nó sẽ hành xử như đối tượng chính, đồng thời sẽ chứa một đối tượng này bên trong, nghĩa là vừa kế thừa vừa có khai báo một thành phần kiểu class cha, trong trường hợp này class cha là DoUong.

```
class Topping : public DoUong {
public:
    Topping(DoUong * doUong) : nuoc(doUong) {}
    string getDesc() {return nuoc->getDesc();}
    float getCost() {return nuoc->getCost();}
private:
    DoUong * nuoc;
};
```

Từ class Topping này, ta có thể kế thừa để tạo ra các đối tượng toppings tùy nhu cầu của mình.

```
class TranChau : public Topping {
public:
    TranChau(DoUong * doUong) : Topping(doUong) {}
    string getDesc() {
        return Topping::getDesc() + ", tran chau";
    }
    float getCost() {
        return Topping::getCost() + 3.0;
    }
};

class Thach : public Topping {
public:
    Thach(DoUong * doUong) : Topping(doUong) {}
    string getDesc() {
        return Topping::getDesc() + ", thach";
    }
    float getCost() {
        return Topping::getCost() + 5.0;
    }
};

class Pudding : public Topping {
public:
    Pudding(DoUong * doUong) : Topping(doUong) {}
    string getDesc() {
        return Topping::getDesc() + ", pudding";
    }
    float getCost() {
        return Topping::getCost() + 4.0;
    }
};
```

Như vậy thì ta có thể quản lý đồ uống với bất cứ topping nào chỉ với một class là DoUong bằng cách sử dụng pattern này, sau đây là ví dụ cho bài toán (hàm xuathoadon() sẽ hỗ trợ in ra thông tin ly trà sữa và giá tiền cần thanh toán).

```
1  void xuathoadon(DoUong * nuoc) {
2      cout << nuoc->getDesc() << " | Tong cong: " << nuoc->getCost() << endl;
3  }
4
5  int main() {
6      cout << "Order 1\n";
7      DoUong * tatua = new TraSua();
8      xuathoadon(tatua);
9
10     cout << "Order 2\n";
11     DoUong * tatua1 = new TraSua();
12     tatua1 = new TranChau(tatua1);
13     xuathoadon(tatua1);
14
15     cout << "Order 3\n";
16     DoUong * tatua2 = new TraSua();
17     tatua2 = new TranChau(tatua2);
18     tatua2 = new Thach(tatua2);
19     xuathoadon(tatua2);
20
21     cout << "Order 4\n";
22     DoUong * tatua3 = new TraSua();
23     tatua3 = new TranChau(tatua3);
24     tatua3 = new Thach(tatua3);
25     tatua3 = new Pudding(tatua3);
26     xuathoadon(tatua3);
27
28     delete tatua; delete tatua1; delete tatua2; delete tatua3;
29     return 0;
30 }
```

Kết quả chạy

```
Order 1
Tra sua | Tong cong: 15
Order 2
Tra sua, tran chau | Tong cong: 18
Order 3
Tra sua, tran chau, thach | Tong cong: 23
Order 4
Tra sua, tran chau, thach, pudding | Tong cong: 27
```

Ưu nhược điểm của Decorator Pattern

- Ưu điểm
  - Có thể mở rộng hành vi của đối tượng mà không cần tạo lớp con mới.
  - Có thể thêm hoặc xoá tính năng của một đối tượng trong lúc thực thi.
  - Một đối tượng có thể được bao bọc bởi nhiều wrapper cùng một lúc.
  - Có thể chia nhiều cách thực thi của một phương thức trong một lớp cho nhiều lớp nhỏ hơn.
- Nhược điểm
  - Khó để xóa một wrapper cụ thể khỏi stack.
  - Khó để triển khai decorator theo cách mà phương thức của nó không phụ thuộc vào thứ tự trong stack.

2.2.5 Facade

Định nghĩa

Facade Pattern cung cấp một interface chung đơn giản thay cho một nhóm các interfaces có trong một hệ thống con (subsystem). Facade Pattern định nghĩa một interface ở một cấp độ cao hơn để giúp cho người dùng có thể dễ dàng sử dụng hệ thống con này.

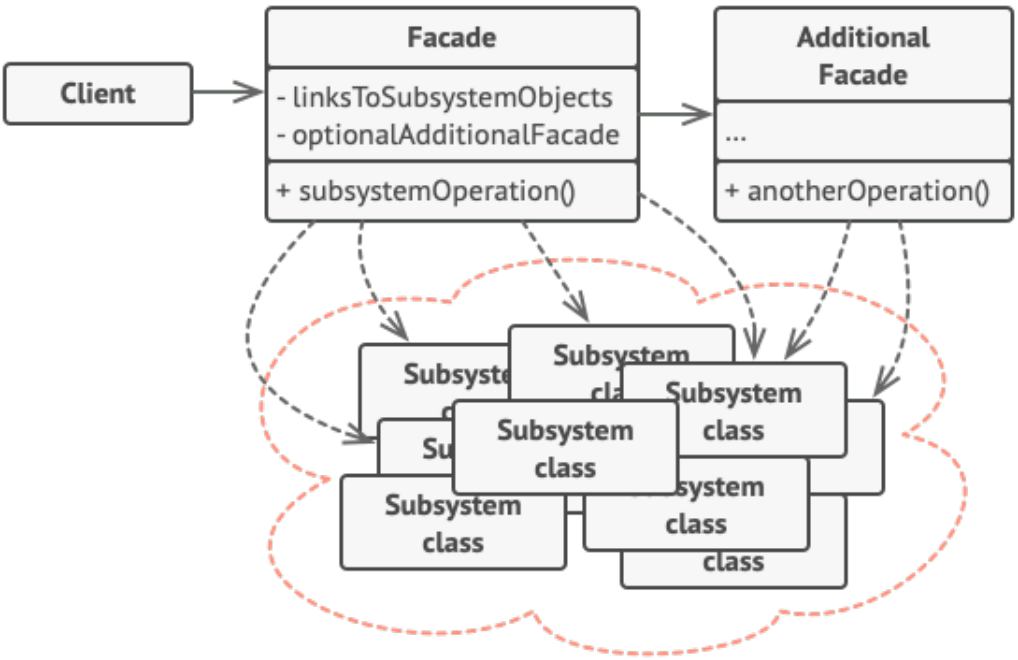
Facade Pattern cho phép các đối tượng truy cập trực tiếp interface chung này để giao tiếp với các interface có trong hệ thống con. Mục tiêu là che giấu các hoạt động phức tạp bên trong hệ thống con, làm cho hệ thống con dễ sử dụng hơn.

Façade Pattern tương tự với Adapter Pattern. Hai Pattern này làm việc theo cùng một cách, nhưng mục đích sử dụng của chúng khác nhau. Adapter Pattern chuyển đổi mã nguồn để làm việc được với mã nguồn khác còn Façade Pattern cho phép bao bọc mã nguồn gốc để nó có thể giao tiếp với mã nguồn khác dễ dàng hơn.

Cách sử dụng

Các thành phần cơ bản của một Facade Pattern:

- **Facade**: biết rõ lớp của hệ thống con nào đảm nhận việc đáp ứng yêu cầu của client, sẽ chuyển yêu cầu của client đến các đối tượng của hệ thống con tương ứng.
- **Subsystems**: cài đặt các chức năng của hệ thống con, xử lý công việc được gọi bởi Facade. Các lớp này không cần biết Facade và không tham chiếu đến nó.
- **Client**: đối tượng sử dụng Facade để tương tác với các hệ thống con.



Ví dụ:

Chúng ta đang có một ứng dụng có thể vẽ được các hình cơ bản như hình chữ nhật, hình tròn, hình tam giác. Mẫu code C++ sau mô phỏng những interface vẽ các hình.

Facade Pattern:

```
#include <iostream>

using namespace std;

class Shape {
public:
    virtual void draw() = 0;
};

class Rectangle : public Shape {
public:
    void draw() {cout << "Rectangle::Draw()\n";}
};

class Circle : public Shape {
public:
    void draw() {cout << "Circle::Draw()\n";}
};

class Triangle : public Shape {
public:
    void draw() {cout << "Triangle::Draw()\n";}
};
```

Trên đây coi như một hệ thống nhỏ của ứng dụng hỗ trợ vẽ hình. Theo đó, nếu muốn vẽ các hình khác nhau thì phía client phải tạo một đối tượng riêng, trường hợp muốn gộp hệ thống lại thì phải dùng tới Facade. Class Facade sẽ có dạng như sau.

```
class ShapeMaker {
public:
    ShapeMaker() {
        rectangle = new Rectangle;
        circle = new Circle;
        triangle = new Triangle;
    }
    void drawRectangle() {rectangle->draw();}
    void drawCircle() {circle->draw();}
    void drawTriangle() {triangle->draw();}
    ~ShapeMaker() {
        delete rectangle;
        delete circle;
        delete triangle;
    }
private:
    Shape * rectangle;
    Shape * circle;
    Shape * triangle;
};
```

Class Facade ShapeMaker này sẽ làm đại diện cho các Shape và client chỉ cần tương tác với ShapeMaker là có thể sử dụng các Shape.

```
int main() {
    ShapeMaker * shapeMaker = new ShapeMaker;
    shapeMaker->drawRectangle();
    shapeMaker->drawCircle();
    shapeMaker->drawTriangle();

    delete shapeMaker;
    return 0;
}
```

Kết quả chạy

```
Rectangle::Draw()
Circle::Draw()
Triangle::Draw()
```

Ưu nhược điểm của Facade Pattern

- Ưu điểm
  - Giúp cho hệ thống trở nên đơn giản hơn trong việc sử dụng và trong việc hiểu nó, vì một mẫu Facade có các phương thức tiện lợi cho các tác vụ chung.
  - Giảm sự phụ thuộc của các đoạn code bên ngoài với hiện thực bên trong của thư viện, vì hầu hết các code đều dùng Facade, vì thế cho phép sự linh động trong phát triển các hệ thống.
- Nhược điểm
  - Nếu quá lạm dụng Facade Pattern sẽ có thể dẫn tới việc Facade trở thành đối tượng thượng đế (god-object, đối tượng quản lý quá nhiều) cho các đối tượng trong chương trình.

2.2.6 Flyweight

Định nghĩa

Flyweight Pattern cho phép tái sử dụng đối tượng tương tự đã tồn tại bằng cách lưu trữ chúng và chỉ tạo đối tượng mới khi không tìm thấy đối tượng phù hợp. Flyweight Pattern được sử dụng khi chúng ta cần tạo một số lượng lớn các đối tượng của 1 lớp nào đó. Do mỗi đối tượng đều đòi hỏi chiếm giữ một khoảng không gian bộ nhớ, nên với một số lượng lớn đối tượng được tạo ra có thể gây nên vấn đề nghiêm trọng đặc biệt đối với các thiết bị có dung lượng nhớ thấp.

Flyweight Pattern có thể được áp dụng để giảm tải cho bộ nhớ thông qua cách chia sẻ các đối tượng. Vì vậy hiệu suất của hệ thống sẽ được tối ưu. Ngoài ra, đối tượng Flyweight không thể thay đổi (immutable) sau khi khởi tạo.

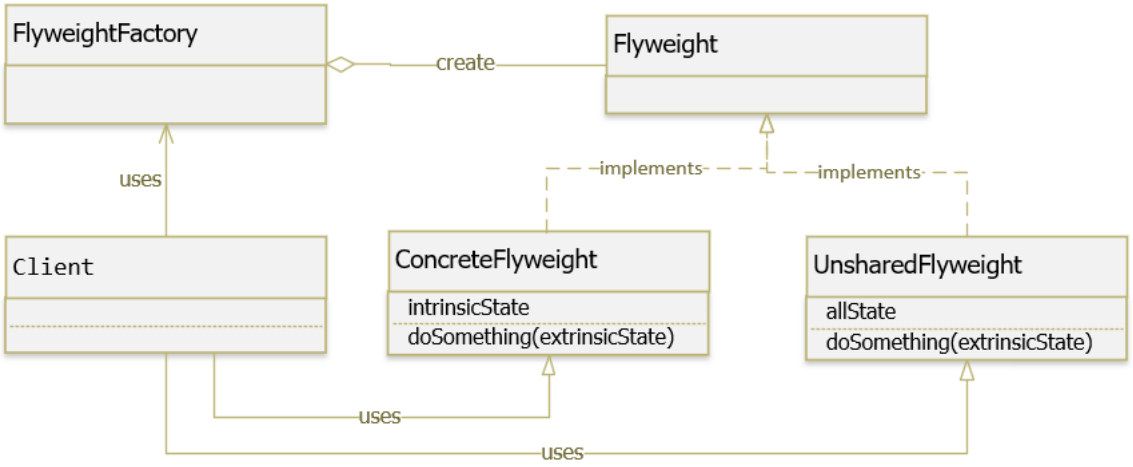
Mục tiêu chính của Flyweight Pattern là giảm bộ nhớ bằng cách chia sẻ các đối tượng. Điều này có thể đạt được bằng cách tách các thuộc tính của đối tượng thành hai trạng thái: độc lập và phụ thuộc. Hay còn gọi là Intrinsic (trạng thái nội tại) và Extrinsic (trạng thái bên ngoài):

- **Intrinsic State (trạng thái nội tại)**: chứa dữ liệu không thể thay đổi (unchangeable) và không phụ thuộc (independent) vào ngữ cảnh (context) của đối tượng Flyweight . Những dữ liệu đó có thể được lưu trữ vĩnh viễn bên trong đối tượng Flyweight. Vì vậy mà Flyweight object có thể chia sẻ. Dữ liệu nội tại là phi trạng thái (stateless) và thường không thay đổi (unchanged). Tính năng này cho phép khả năng tái tạo các thuộc tính đối tượng Flyweight giữa các đối tượng tương tự khác. Điều quan trọng cần lưu ý là các đối tượng Flyweight chỉ nên nhận trạng thái bên trong của chúng thông qua các tham số của hàm tạo và không cung cấp các biến public hay các phương thức setter (set/get đối với các biến private).
- **Extrinsic State (trạng thái bên ngoài)**: 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 (runtime). Do đó, những dữ liệu đó không được lưu trữ trong bộ nhớ. Vì trạng thái bên ngoài là phụ thuộc ngữ cảnh và có thể thay đổi nên các đối tượng đó không thể được chia sẻ. Do đó, client chịu trách nhiệm truyền dữ liệu liên quan đến trạng thái bên ngoài cho đối tượng Flyweight khi cần thiết, có thể thông qua các tham số (argument).

Cách sử dụng

Các thành phần trong Flyweight Pattern:

- **Flyweight**: là một interface/ abstract class, định nghĩa các các thành phần của một đối tượng.
- **ConcreteFlyweight**: triển khai các phương thức đã được định nghĩa trong Flyweight. Việc triển khai này phải thực hiện các khả năng của trạng thái nội tại. Đó là dữ liệu phải không thể thay đổi và có thể chia sẻ (shareable). Các đối tượng là phi trạng thái trong triển khai này. Vì vậy, đối tượng ConcreteFlyweight giống nhau có thể được sử dụng trong các ngữ cảnh khác nhau.
- **UnsharedFlyweight**: mặc dù mẫu thiết kế Flyweight cho phép chia sẻ thông tin, nhưng có thể tạo ra các thể hiện không được chia sẻ (not shared). Trong những trường hợp này, thông tin của các đối tượng có thể là có trạng thái (stateful).
- **FlyweightFactory (Cache)**: class này có thể là một Factory Pattern được sử dụng để giữ tham chiếu đến đối tượng Flyweight đã được tạo ra. Nó cung cấp một phương thức để truy cập đối tượng Flyweight được chia sẻ. FlyweightFactory bao gồm một Pool (có thể là HashMap, không cho phép bên ngoài truy cập vào) để lưu trữ đối tượng Flyweight trong bộ nhớ. Nó sẽ trả về đối tượng Flyweight đã tồn tại khi được yêu cầu từ Client hoặc tạo mới nếu không tồn tại.
- **Client**: sử dụng FlyweightFactory để khởi tạo đối tượng Flyweight.



Ví dụ:

Giả sử chúng ta cần thiết kế một game dạng Total War. Trong game sẽ tạo ra một số lượng nhân vật rất lớn để tham gia chiến đấu, các nhân vật sẽ thuộc những binh chủng khác nhau. Vì mỗi nhân vật chiếm một lượng bộ nhớ nhất định, nếu khởi tạo từng nhân vật theo các phương pháp truyền thống thì sẽ hao tổn rất nhiều thời gian và bộ nhớ, nếu số lượng nhân vật quá lớn có thể dẫn tới việc những máy tính cấu hình không cao có thể sẽ không đáp ứng đủ vùng nhớ cho ứng dụng.

Vậy, để tiết kiệm thời gian và bộ nhớ khi khởi tạo số lượng lớn các đối tượng tương tự như nhau, ta nên sử dụng Flyweight. Sau đây là đoạn code Java thể hiện quá trình khởi tạo những nhân vật trong game.

Flyweight Pattern:

- Flyweight chứa interface cho các binh chủng (ISoldier.java)

```java
public interface ISoldier {

    void promote(Context context);
}
```

- ConcreteFlyweight triển khai các phương thức trong Flyweight (Soldier.java)

```java
public class Soldier implements ISoldier {

    private final String name; // Intrinsic State

    public Soldier(String name) {
        this.name = name;
        System.out.println("Soldier is created! - " + name);
    }

    @Override
    public void promote(Context context) {
        System.out.println(name + " " + context.getId() + " promoted " + context.
            getStar());
    }
}
```

- UnsharedFlyweight chứa thông tin chưa chia sẻ (Context.java)

```java
public class Context {

    private String id;
    private int star;
    public Context(String id, int star) {
        this.id = id;
        this.star = star;
    }
    public String getId() {
        return this.id;
    }
    public int getStar() {
        return this.star;
    }
}
```

- FlyweightFactory giữ tham chiếu tới đối tượng Flyweight có sẵn, ở đây thời gian đợi để tạo ra một nhân vật lính là 3 giây (SoldierFactory.java)

```java
import java.util.HashMap;
import java.util.Map;

// FlyweightFactory
public class SoldierFactory {

    private static final Map<String, ISoldier> soldiers = new HashMap<>();

    private SoldierFactory() {
        throw new IllegalStateException();
    }

    public static synchronized ISoldier createSoldier(String name) {
        ISoldier soldier = soldiers.get(name);
        if (soldier == null) {
            waitingForCreateASoldier();
            soldier = new Soldier(name);
            soldiers.put(name, soldier);
        }
        return soldier;
    }

    public static synchronized int getTotalOfSoldiers() {
        return soldiers.size();
    }

    // Time for create a soldier is 3 seconds
    private static void waitingForCreateASoldier() {
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- Clent (GameApp.java)

```
import java.time.Duration;
import java.util.ArrayList;
import java.util.List;

// Client
public class GameApp {

    private static List<ISoldier> soldiers = new ArrayList<>();

    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();
        createSoldier(5, "LightInfantry", 1);
        createSoldier(5, "SpearMen", 1);
        createSoldier(3, "LightCavalry", 3);
        createSoldier(2, "LightInfantry", 2);
        createSoldier(3, "LightCavalry", 3);
        createSoldier(3, "LightCavalry", 3);
        createSoldier(3, "SpearMen", 1);
        long endTime = System.currentTimeMillis();
        System.out.println("---");
        System.out.println("Total soldiers made : " + soldiers.size());
        System.out.println("Total time worked : " + Duration.ofMillis(endTime -
            startTime).getSeconds() + " seconds");
        System.out.println("Total type of soldiers made : " + SoldierFactory.
            getTotalOfSoldiers());
    }

    private static void createSoldier(int numberOfSoldier, String soldierName, int
         numberOfStar) {
        for (int i = 1; i <= numberOfSoldier; i++) {
            Context star = new Context("Soldier" + (soldiers.size() + 1),
                numberOfStar);
            ISoldier soldier = SoldierFactory.createSoldier(soldierName);
            soldier.promote(star);
            soldiers.add(soldier);
        }
    }
}
```

Client đã yêu cầu tạo ra tổng cộng 3 binh chủng, 24 nhân vật. Nếu khởi tạo theo cách thông thường thì sẽ mất 72 giây (vì mỗi nhân vật cần 3 giây để hoàn thành khởi tạo), nhưng khi áp dụng Flyweight Pattern, ta chỉ cần đợi 9 giây, ta có thể thấy được điều này qua kết quả chạy sau.

Kết quả chạy

```
Soldier is created! - LightInfantry
LightInfantry Soldier1 promoted 1
LightInfantry Soldier2 promoted 1
LightInfantry Soldier3 promoted 1
LightInfantry Soldier4 promoted 1
LightInfantry Soldier5 promoted 1
Soldier is created! - SpearMen
SpearMen Soldier6 promoted 1
SpearMen Soldier7 promoted 1
SpearMen Soldier8 promoted 1
SpearMen Soldier9 promoted 1
SpearMen Soldier10 promoted 1
Soldier is created! - LightCavalry
LightCavalry Soldier11 promoted 3
LightCavalry Soldier12 promoted 3
LightCavalry Soldier13 promoted 3
LightInfantry Soldier14 promoted 2
LightInfantry Soldier15 promoted 2
LightCavalry Soldier16 promoted 3
LightCavalry Soldier17 promoted 3
LightCavalry Soldier18 promoted 3
LightCavalry Soldier19 promoted 3
LightCavalry Soldier20 promoted 3
LightCavalry Soldier21 promoted 3
SpearMen Soldier22 promoted 1
SpearMen Soldier23 promoted 1
SpearMen Soldier24 promoted 1
—
Total soldiers made : 24
Total time worked : 9 seconds
Total type of soldiers made : 3
```

Ưu nhược điểm của Flyweight Pattern

- Ưu điểm
  - Giảm số lương đối tượng được tạo ra bằng cách chia sẻ đối tượng. Vì vậy tiết kiệm bộ nhớ và các thiết bị lưu trữ cần thiết.
  - Cải thiện khả năng cache dữ liệu vì thời gian đáp ứng nhanh.
  - Tăng hiệu suất cho hệ thống.
- Nhược điểm
  - Đánh đổi về mặt sử dụng CPU khi các đối tượng flyweight bị truy cập nhiều lần.
  - Code trở nên phức tạp và khó hiểu hơn nhiều.

2.2.7 Proxy

Định nghĩa

Proxy Pattern là mẫu thiết kế giúp chúng ta tạo ra vật thay thế (surrogate) hoặc giữ chỗ (placeholder) cho đối tượng khác. Một Proxy kiểm soát quyền truy cập vào đối tượng ban đầu, cho phép ta làm điều gì đó trước hoặc sau những yêu cầu (request) được truyền đến đối tượng ấy.

Trong Proxy Pattern, một class sẽ đại diện cho một class khác, nghĩa là tất cả các truy cập trực tiếp đến một đối tượng nào đó sẽ được chuyển hướng vào một đối tượng trung gian. Proxy Pattern đại diện cho một đối tượng khác thực thi các phương thức, phương thức đó có thể được định nghĩa lại cho phù hợp với mục đích sử dụng.

Các loại Proxy được biết

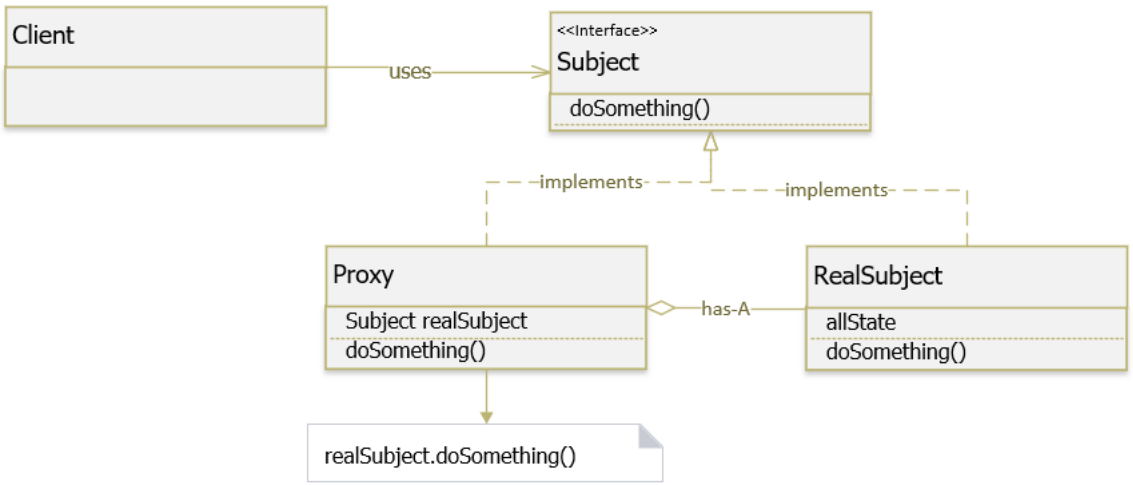
- **Virtual Proxy**: Virtual Proxy sẽ cung cấp một số kết quả mặc định và ngay lập tức nếu đối tượng thực mất rất nhiều thời gian để tạo ra kết quả. Các proxy này bắt đầu thao tác trên các đối tượng thực đồng thời cung cấp kết quả mặc định cho client. Sau khi thao tác trên đối tượng thực hoàn thành, các proxy này sẽ đưa dữ liệu thực đến client nơi mà nó đã cung cấp dữ liệu giả trước đó.
- **Protection Proxy**: Phạm vi truy cập của các client khác nhau sẽ khác nhau. Protection Proxy sẽ kiểm tra các quyền truy cập của client khi có một dịch vụ được yêu cầu.
- **Remote Proxy**: Client truy cập qua Remote Proxy để chiếu tới một đối tượng được bảo về nằm bên ngoài ứng dụng (trên cùng máy hoặc máy khác).
- **Smart Proxy**: Là nơi kiểm soát các hoạt động bổ sung mỗi khi đối tượng được tham chiếu.

Ngoài các loại phổ biến trên, còn có Firewall, Cache, Synchronization và Copy-On-Write Proxy.

Cách sử dụng

Các thành phần của Proxy Pattern:

- **Subject**: là một interface định nghĩa các phương thực để giao tiếp với client. Đối tượng này xác định interface chung cho RealSubject và Proxy để Proxy có thể được sử dụng bất cứ nơi nào mà RealSubject mong đợi.
- **RealSubject**: là một class dùng để thực hiện các thao tác thực sự. Đây là đối tượng chính mà proxy đại diện.
- **Proxy**: là một class sẽ thực hiện các bước kiểm tra và gọi tới đối tượng của class service thật để thực hiện các thao tác sau khi kiểm tra. Nó duy trì một tham chiếu đến RealSubject để Proxy có thể truy cập nó. Nó cũng thực hiện các interface tương tự như RealSubject để Proxy có thể được sử dụng thay cho RealSubject. Proxy cũng điều khiển truy cập vào RealSubject và có thể tạo hoặc xóa đối tượng này.
- **Client**: Đối tượng cần sử dụng RealSubject nhưng thông qua Proxy.



Ví dụ:

Một ví dụ đơn giản nhất về Proxy chính là mạng internet của trường học, những mạng internet như thế này chặn người dùng truy cập vào một số trang web. Proxy sẽ kiểm tra trước trang web mà chúng ta muốn truy cập, nếu chúng không thuộc vào những trang bị cấm thì ta sẽ truy cập được. Sau đây là code C++ mô phỏng Proxy này.

Proxy Pattern:

- Subject là interface của internet (Internet.java)

```
public interface Internet {
    public void connectTo(String serverhost) throws Exception;
}
```

- RealSubject gồm có chức năng kết nối (RealInternet.java)

```
public class RealInternet implements Internet {
    @Override
    public void connectTo(String serverhost)
    {
        System.out.println("Connected to " + serverhost);
    }
}
```

- Proxy lưu cú pháp của các trang web cấm và phương thức kiểm tra (ProxyInternet.java)

```
import java.util.ArrayList;
import java.util.List;

public class ProxyInternet implements Internet
{
    private Internet internet = new RealInternet();
    private static List<String> bannedSites;

    static
    {
        bannedSites = new ArrayList<String>();
        bannedSites.add("nyc.com");
        bannedSites.add("ex.com");
        bannedSites.add("nguoiyeucu.com");
        bannedSites.add("quaytrolai.org");
        bannedSites.add("tinhyeu.net");
    }

    @Override
    public void connectTo(String serverhost) throws Exception
    {
        System.out.println("Requesting connection to " + serverhost);
        if(bannedSites.contains(serverhost.toLowerCase()))
        {
            throw new Exception("Access Denied. " + serverhost + " refused
                connection!");
        }
        internet.connectTo(serverhost);
    }
}
```

- Client sẽ gửi yêu cầu truy cập các trang web tới Proxy (Client.java)

```
1  public class Client
2  {
3      public static void main (String[] args)
4      {
5          Internet internet = new ProxyInternet();
6          try
7          {
8              internet.connectTo("e-learning.hcmut.edu.vn");
9              internet.connectTo("tinhyeu.net");
10         }
11         catch (Exception e)
12         {
13             System.out.println(e.getMessage());
14         }
15     }
16 }
```

Kết quả chạy

```
Requesting connection to e-learning.hcmut.edu.vn
Connected to e-learning.hcmut.edu.vn
Requesting connection to tinhyeu.net
Access Denied. tinhyeu.net refused connection!
```

Những đặc điểm chung của các loại Proxy Patterns

- Cung cấp mức truy cập gián tiếp vào một đối tượng.
- Tham chiếu vào đối tượng đích và chuyển tiếp các yêu cầu đến đối tượng đó.
- Cả Proxy và đối tượng đích đều kế thừa hoặc thực thi chung một lớp giao diện. Mã máy dịch cho lớp giao diện thường "nhẹ" hơn các lớp cụ thể và do đó có thể giảm được thời gian tải dữ liệu giữa server và client.

Ưu nhược điểm của Proxy Pattern

- Ưu điểm
  - Thường được dùng để bảo mật cho hệ thống.
  - Tránh việc phải nhân bản những đối tượng lớn, phức tạp, tăng hiệu suất cho phần mềm.
- Nhược điểm
  - Đôi khi một số Client có thể truy cập trực tiếp đối tượng mà không thông qua Proxy, khi đó xảy ra sự hỗn loạn hành vi của code.

2.3 Behavioral Patterns

Behavioral Patterns là nhóm các Patterns liên quan đến giải thuật và sự phân công trách nhiệm, chức năng giữa các đối tượng với nhau. Behavioral Patterns không chỉ mô tả các lớp hay đối tượng mà còn mô tả cách thức mà chúng giao tiếp với nhau. Các Patterns này đặc trưng cho luồng điều khiển phức tạp khó theo dõi trong run-time. Chúng giúp ta không cần phải bận tâm vào những luồng điều khiển đó để tập trung vào cách mà các đối tượng liên kết với nhau. Nhóm gồm này gồm 11 mẫu: **Interpreter**, **Template Method**, **Chain of Responsibility**, **Command**, **Iterator**, **Mediator**, **Memento**, **Observer**, **State**, **Strategy**, **Visitor**.

Interpreter Pattern và Template Method Pattern sử dụng tính kế thừa để phân phối hành vi giữa các lớp. Interpreter Pattern đại diện cho ngữ pháp dưới dạng hệ thống phân cấp các lớp và triển khai thông dịch như một thao tác trên các instances của các lớp. Template Method Pattern thì đơn giản hơn, một Template Method Pattern là định nghĩa trừu tượng của của giải thuật; nó định nghĩa giải thuật theo từng bước một. Mỗi bước của giải thuật sẽ gọi một thao tác trừu tượng (abstract operation) hoặc một thao tác nguyên thủy (primitive operation). Các lớp con bổ sung giải thuật bằng cách định nghĩa các thao tác trừu tượng của Template Method.

Một số mẫu khác mô tả cách mà một nhóm đối tượng ngang hàng (peer objects) hợp tác để thực hiện một tác vụ mà không đối tượng đơn lẻ nào có thể tự thực hiện (quan trọng là các đối tượng ngang hàng đó biết nhau bằng cách nào). Các đối tượng ngang hàng có thể lưu trữ tham chiếu với nhau, nhưng điều này sẽ dẫn đến gia tăng liên kết (coupling) giữa chúng. Mediator Pattern giúp ta tránh được điều này bằng cách tạo ra một trung gian giữa các đối tượng, vật trung gian này sẽ cung cấp liên kết cho các khớp nối lỏng lẻo (loose coupling).

Chain of Responsibility Pattern cung cấp các khớp nối lóng hơn, cho phép ta gửi yêu cầu tới một đối tượng thông qua một chuỗi các đối tượng ứng cử viên (candidate objects). Các ứng cử viên đều có thể thực hiện yêu cầu dựa trên các điều kiện run-time. Số lượng ứng cử viên là tùy ý, ta có thể chọn bất kỳ để đưa vào chuỗi trong run-time.

Observer Pattern định nghĩa và duy trì sự phụ thuộc giữa các đối tượng. Ví dụ kinh điển của Observer là trong Model/View/Controller, khi mô hình thay đổi trạng thái thì tất cả các thành phần trong mô hình đó đều được thông báo.

Các Behavioral Patterns khác liên quan đến việc đóng gói hành vi trong một đối tượng và ủy thác các yêu cầu cho nó. Strategy Pattern đóng gói giải thuật trong một đối tượng, giúp dễ dàng chỉ định và thay đổi giải thuật mà một đối tượng sử dụng. Command Pattern đóng gói một yêu cầu trong một đối tượng để nó có thể được truyền dưới dạng tham số, được lưu trữ trong danh sách lịch sử hoặc được vận dụng theo các cách khác. State Pattern đóng gói các trạng thái của một đối tượng để đối tượng có thể thay đổi hành vi của nó khi đối tượng trạng thái (state object) của nó thay đổi. Visitor Pattern đóng gói hành vi hoặc phân phối hành vi cho các lớp. Iterator Pattern trừu tượng hóa cách truy cập và duyệt qua các đối tượng trong một hệ thống tổng hợp.

Mỗi lớp, đối tượng trong dự án sẽ chịu một trách nhiệm riêng, nhưng khi dự án trở nên lớn và công kềnh thì rất khó để đảm bảo được điều này. Vì vậy các Behavioral Patterns sẽ hỗ trợ quản lý hành vi, trách nhiệm của các class dễ dàng hơn.

Các đoạn code mẫu cho Behavioral Pattern có ở đường link sau:
https://github.com/PhucLe03/OOP/tree/main/Design%20Patterns/Behavioral%20Patterns
hoặc mã QR bên dưới.

2.3.1 Interpreter

Định nghĩa

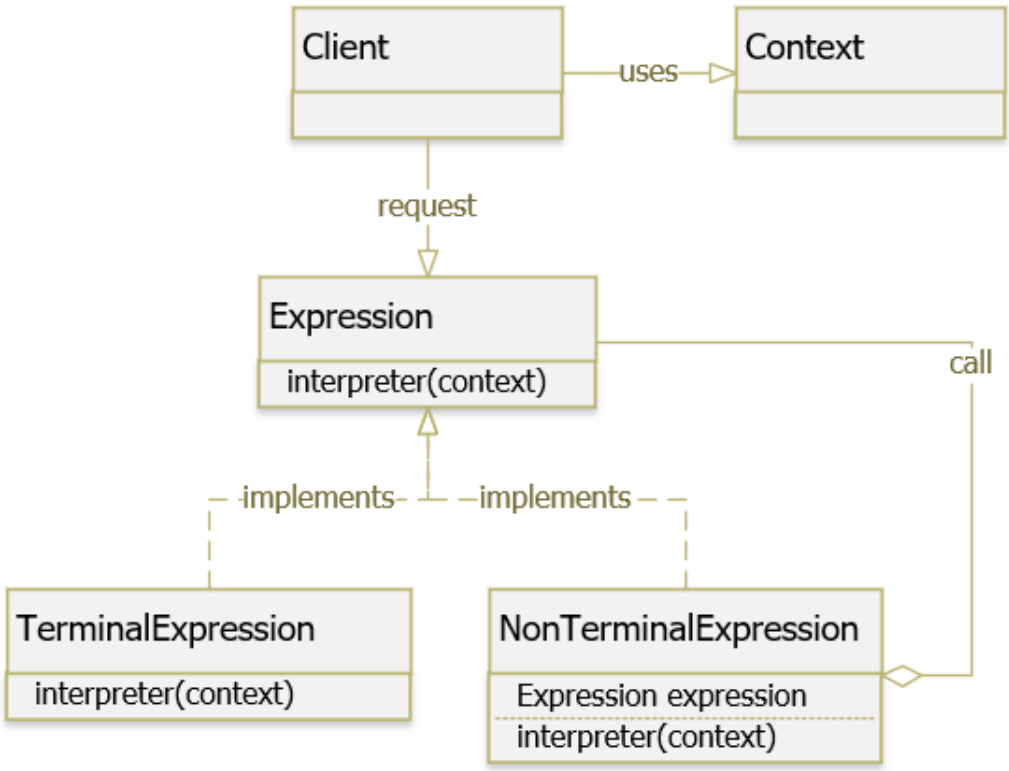
Interpreter nghĩa là thông dịch, Interpreter Pattern giúp người lập trình có thể xây dựng những đối tượng dynamic bằng cách đọc mô tả về đối tượng rồi sau đó xây dựng đối tượng đúng theo mô tả đó.

Interpreter Pattern có hạn chế về phạm vi áp dụng. Mẫu này thường được sử dụng để định nghĩa bộ ngữ pháp đơn giản (grammar), trong các công cụ quy tắc đơn giản (rule), ...

Cách sử dụng

Các thành phần trong Interpreter Pattern:

- **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.



Ví dụ:

Ta có thể dùng Interpreter để viết chương trình dịch một số La Mã thành số trong hệ thập phân.

Số La Mã có nguyên tắc sau:

- Chữ số La Mã được thể hiện bằng chữ cái của bảng chữ cái: I=1; V=5; X=10; L=50; C=100; D=500; M=1000.
- Một chữ cái có thể lặp lại giá trị của nó nhiều lần, tối đa ba lần. Ví dụ: XXX = 30, CC = 200,...
- Nếu một hoặc nhiều chữ cái được đặt sau một chữ cái có giá trị lớn hơn, cộng số trước đó. Ví dụ:
  - VI = 6 (5 + 1 = 6)
  - LXX = 70 (50 + 10 + 10 = 70)
  - MCC = 1200 (1000 + 100 + 100 = 1200)
- Nếu một chữ cái được đặt trước một chữ cái có giá trị lớn hơn, trừ đi số trước đó. Ví dụ:
  - IV = 4 (5 – 1 = 4)
  - XC = 90 (100 – 10 = 90)
  - CM = 900 (1000 – 100 = 900)

Interpreter Pattern:

- Context (Context.java)

```java
public class Context
{

    private String input;
    private int output;

    public Context(String input)
    {
        this.input = input;
    }

    public void setInput(String input)
    {
        this.input = input;
    }

    public String getInput()
    {
        return input;
    }

    public int getOutput()
    {
        return output;
    }

    public void setOutput(int output)
    {
        this.output = output;
    }
}
```

- Expression (Expression.java)

```
public abstract class Expression {

    public void interpret(Context context) {
        if (context.getInput().length() == 0) return;
        if (context.getInput().startsWith(nine())) {
            context.setOutput(context.getOutput() + 9 * multiplier());
            context.setInput(context.getInput().substring(2));
        } else if (context.getInput().startsWith(four())) {
            context.setOutput(context.getOutput() + 4 * multiplier());
            context.setInput(context.getInput().substring(2));
        } else if (context.getInput().startsWith(five())) {
            context.setOutput(context.getOutput() + 5 * multiplier());
            context.setInput(context.getInput().substring(1));
        }
        while (context.getInput().startsWith(one())) {
            context.setOutput(context.getOutput() + 1 * multiplier());
            context.setInput(context.getInput().substring(1));
        }
    }

    public abstract String one();

    public abstract String four();

    public abstract String five();

    public abstract String nine();

    public abstract int multiplier();
}
```

- TerminalExpression và NonTerminalExpression (ThousandExpression.java)

```
public class ThousandExpression extends Expression {
    @Override
    public String one() {return "M";}
    @Override
    public String four() {return " ";}
    @Override
    public String five() {return " ";}
    @Override
    public String nine() {return " ";}
    @Override
    public int multiplier() {return 1000;}
}
```

(HundredExpression.java)

```java
public class HundredExpression extends Expression {
    @Override
    public String one() {return "C";}
    @Override
    public String four() {return "CD";}
    @Override
    public String five() {return "D";}
    @Override
    public String nine() {return "CM";}
    @Override
    public int multiplier() {return 100;}
}
```

(TenExpression.java)

```java
public class TenExpression extends Expression {
    @Override
    public String one() {return "X";}
    @Override
    public String four() {return "XL";}
    @Override
    public String five() {return "L";}
    @Override
    public String nine() {return "XC";}
    @Override
    public int multiplier() {return 10;}
}
```

(OneExpression.java)

```java
public class OneExpression extends Expression {
    @Override
    public String one() {return "I";}
    @Override
    public String four() {return "IV";}
    @Override
    public String five() {return "V";}
    @Override
    public String nine() {return "IX";}
    @Override
    public int multiplier() {return 1;}
}
```

- Client (Client.java)

```java
import java.util.ArrayList;
import java.util.List;

public class Client {

    public static void main(String[] args) {
        String[] romans = { "XXX", "XI", "MMIII", "CCXV", "MMCCXXXIX" };
        for (String roman : romans) {
            convertRomanToNumber(roman);
        }
    }

    private static void convertRomanToNumber(String roman) {
        List<Expression> tree = new ArrayList<>();
        tree.add(new ThousandExpression());
        tree.add(new HundredExpression());
        tree.add(new TenExpression());
        tree.add(new OneExpression());

        Context context = new Context(roman);
        for (Expression exp : tree) {
            exp.interpret(context);
        }
        System.out.println(roman + " = " + context.getOutput());
    }
}
```

Kết quả chạy

XXX = 30
XI = 11
MMIII = 2003
CCXV = 215
MMCCXXXIX = 2239

Ưu nhược điểm của Interpreter Pattern

- Ưu điểm
  - Dễ dàng thay đổi và mở rộng ngữ pháp. Vì mẫu này sử dụng các lớp để biểu diễn các quy tắc ngữ pháp, chúng ta có thể sử dụng thừa kế để thay đổi hoặc mở rộng ngữ pháp. Các biểu thức hiện tại có thể được sửa đổi theo từng bước và các biểu thức mới có thể được định nghĩa lại các thay đổi trên các biểu thức cũ.
  - Cài đặt và sử dụng ngữ pháp rất đơn giản. Các lớp xác định các node trong cây cú pháp có các implement tương tự. Các lớp này dễ viết và các phân cấp con của chúng có thể được tự động hóa bằng trình biên dịch hoặc trình tạo trình phân tích cú pháp.
- Nhược điểm
  - Ngữ pháp nếu có chứa quá nhiều quy tắc có thể khó quản lý và bảo trì.
  - Do bộ ngữ pháp được phân tích trong cấu trúc phân cấp (cây) nên hiệu suất không được đảm bảo.

2.3.2 Template Method

Định nghĩa

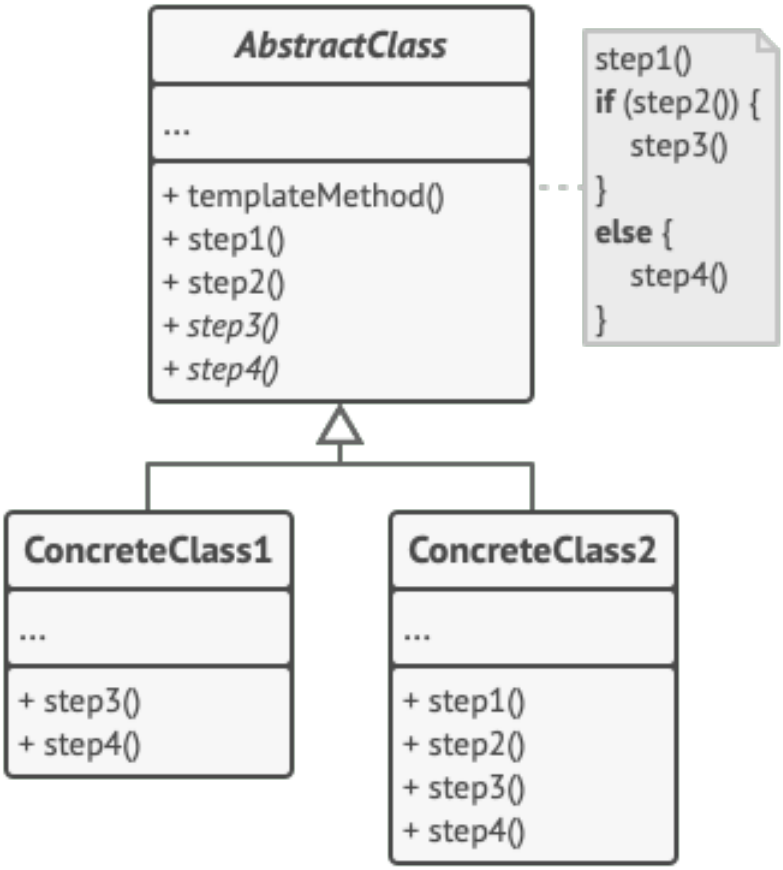
Template Method xây dựng một bộ khung giải thuật trong một toán tử, để lại việc định nghĩa một vài bước cho các class con mà không làm thay đổi cấu trúc chung của giải thuật.

Phương pháp này được áp dụng rất nhiều trong thiết kế framework vì nó cho phép chúng ta tái sử dụng đoạn code nhỏ ở những nơi khác nhau sau khi một số điều chỉnh nhất định. Điều này cũng góp phần tránh được trùng lặp code (duplicate).

Cách sử dụng

Các thành phần trong Template Method Pattern:

- **Abstract**: khai báo bộ khung giải thuật, gồm các bước. Các bước có thể được khai báo trừu tượng hoặc triển khai mặc định.
- **Concrete**: các class con dùng để ghi đè (override) lên các bước trong giải thuật khung, nhưng không được override lên bộ khung đó.



Ví dụ:

Mẫu code Java sau sẽ mô phỏng lại các quy trình xây dựng một căn nhà theo kiểu mẫu được quy định trước.

Template Method Pattern:

- Bộ khung chung của giải thuật (HouseTemplate.java)

```java
public abstract class HouseTemplate {

   public final void buildHouse(){
      buildFoundation();
      buildPillars();
      buildWalls();
      buildWindows();
      System.out.println("House is built.");
   }

   private void buildWindows() {
      System.out.println("Building Glass Windows");
   }

   public abstract void buildWalls();
   public abstract void buildPillars();

   private void buildFoundation() {
      System.out.println("Building foundation with cement, iron rods and sand");
   }
}
```

Hàm buildHouse() là một template dùng để xác định thứ tự các thao tác thực thi (gồm dựng móng, xây cột, xây tường, dựng cửa sổ).

- Các class con (WoodenHouse.java, GlassHouse.java)

  Đối với nhà gỗ, tường và cột sẽ được xây dựng bằng gỗ.

```java
public class WoodenHouse extends HouseTemplate {

   @Override
   public void buildWalls() {
      System.out.println("Building Wooden Walls");
   }

   @Override
   public void buildPillars() {
      System.out.println("Building Pillars with Wood coating");
   }
}
```

Đối với nhà kính, tường và cột sẽ được xây dựng bằng kính.

```
public class GlassHouse extends HouseTemplate {

   @Override
   public void buildWalls() {
      System.out.println("Building Glass Walls");
   }

   @Override
   public void buildPillars() {
      System.out.println("Building Pillars with glass coating");
   }
}
```

- Client (HouseClient.java)

```
public class HouseClient {

    public static void main(String[] args) {
      System.out.println("--Build Wooden House--");
      HouseTemplate houseType = new WoodenHouse();
      houseType.buildHouse();
      System.out.println("--Build Glass House--");
      houseType = new GlassHouse();
      houseType.buildHouse();
   }
}
```

Kết quả chạy

–Build Wooden House–
Building foundation with cement, iron rods and sand
Building Pillars with Wood coating
Building Wooden Walls
Building Glass Windows
House is built.
–Build Glass House–
Building foundation with cement, iron rods and sand
Building Pillars with glass coating
Building Glass Walls
Building Glass Windows
House is built.

Ưu nhược điểm của Template Method Pattern

- Ưu điểm
  - Tái sử dụng code, tránh trùng lặp code: đưa những phần trùng lặp vào abstract class.
  - Cho phép người dùng override chỉ một số phần nhất định của giải thuật lớn, làm cho chúng ít bị ảnh hưởng hơn bởi những thay đổi xảy ra với các phần khác của giải thuật.
- Nhược điểm
  - Càng nhiều bước để override càng khó bảo trì, nâng cấp.

2.3.3 Chain of Responsibility

Định nghĩa

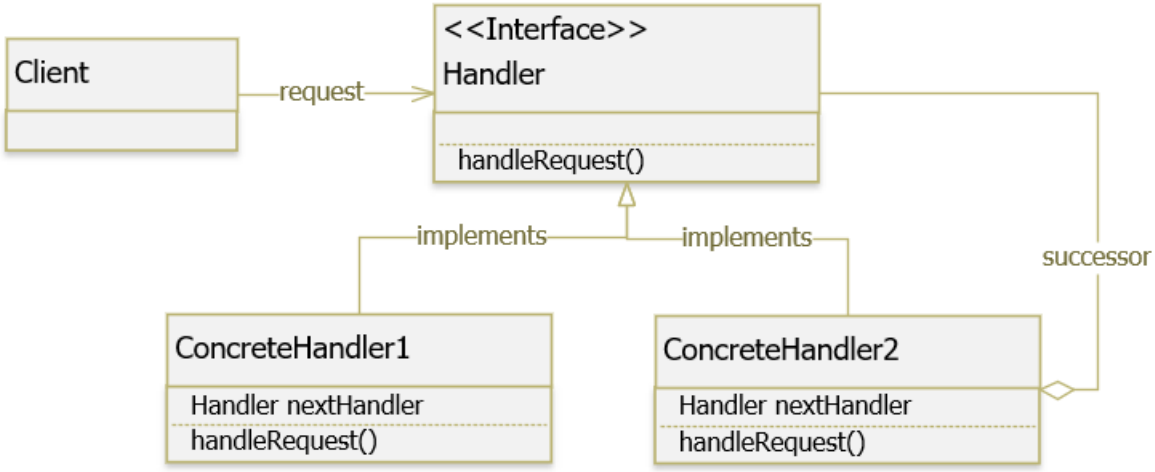
Chain of Responsiblity cho phép một đối tượng gửi một yêu cầu nhưng không biết đối tượng nào sẽ nhận và xử lý nó. Điều này được thực hiện bằng cách kết nối các đối tượng nhận yêu cầu thành một chuỗi (chain) và gửi yêu cầu theo chuỗi đó cho đến khi có một đối tượng xử lý nó.

Chain of Responsibility Pattern hoạt động như một danh sách liên kết (Linked list) với việc đệ quy duyệt qua các phần tử (recursive traversal).

Cách sử dụng

Các thành phần trong Chain of Responsibility:

- **Handler**: định nghĩa 1 interface để xử lý các yêu cầu. Gán giá trị cho đối tượng successor (không bắt buộc).
- **ConcreteHandler**: xử lý yêu cầu. Có thể truy cập đối tượng successor (thuộc class Handler). Nếu đối tượng ConcreateHandler không thể xử lý được yêu cầu, nó sẽ gởi lời yêu cầu cho successor của nó.
- **Client**: tạo ra các yêu cầu và yêu cầu đó sẽ được gửi đến các đối tượng tiếp nhận.



Client gửi một yêu cầu để được xử lý gửi nó đến chuỗi các trình xử lý (handlers), đó là các lớp mở rộng lớp Handler. Mỗi Hanlder trong chuỗi lần lượt cố gắng xử lý yêu cầu nhận được từ Client. Nếu trình xử lý đầu tiên (ConcreteHandler) có thể xử lý nó, thì yêu cầu sẽ được xử lý. Nếu không được xử lý thì sẽ gửi đến trình xử lý tiếp theo trong chuỗi (ConcreteHandler + 1).

Ví dụ:

Ta có thể tham khảo đoạn code Python dùng để "handle" một chuỗi ký tự như sau.

Chain of Responsibility Pattern:

- Handler

```
class Handler(object):
    def __init__(self, nxt):
        self._nxt = nxt
    def handle(self, request):
        handled = self.processRequest(request)
        if not handled:
            self._nxt.handle(request)
    def processRequest(self, request):
        raise NotImplementedError('First implement!')
```

- Các ConcreteHandler

```
class FirstConcreteHandler(Handler):
    def processRequest(self, request):
        if 'a' <= request <= 'e':
            print("{}: request '{}' is being handled".format(self.__class__.
                __name__, request))
            return True

class SecondConcreteHandler(Handler):
    def processRequest(self, request):
        if 'e' < request <= 'l':
            print("{}: request '{}' is being handled".format(self.__class__.
                __name__, request))
            return True

class ThirdConcreteHandler(Handler):
    def processRequest(self, request):
        if 'l' < request <= 'z':
            print("{}: request '{}' is being handled".format(self.__class__.
                __name__, request))
            return True

class DefaultHandler(Handler):
    def processRequest(self, request):
        print("{}: request '{}' has no handler.".format(self.__class__.__name__,
            request))
        return True
```

- Client

```
class User:
    def __init__(self):
        initial = None
        self.handler = FirstConcreteHandler(SecondConcreteHandler(
            ThirdConcreteHandler(DefaultHandler(initial))))

    def agent(self, user_request):
        for request in user_request:
            self.handler.handle(request)

if __name__ == "__main__":
    user = User()
    string = "cse-hcmut"
    requests = list(string)
    user.agent(requests)
```

Kết quả chạy

```
FirstConcreteHandler: request 'c' is being handled
ThirdConcreteHandler: request 's' is being handled
FirstConcreteHandler: request 'e' is being handled
DefaultHandler: request '-' has no handler.
SecondConcreteHandler: request 'h' is being handled
FirstConcreteHandler: request 'c' is being handled
ThirdConcreteHandler: request 'm' is being handled
ThirdConcreteHandler: request 'u' is being handled
ThirdConcreteHandler: request 't' is being handled
```

Ưu nhược điểm của Chain of Responsibility Pattern

- Ưu điểm
  - Giảm kết nối (loose coupling): Thay vì một đối tượng có khả năng xử lý yêu cầu chứa tham chiếu đến tất cả các đối tượng khác, nó chỉ cần một tham chiếu đến đối tượng tiếp theo. Tránh sự liên kết trực tiếp giữa đối tượng gửi yêu cầu (sender) và các đối tượng nhận yêu cầu (receivers).
  - Tăng tính linh hoạt, phân chia trách nhiệm cho các đối tượng.
  - Có khả năng thay đổi dây chuyền (chain) trong run time.
- Nhược điểm
  - Một số yêu cầu có thể không được xử lý.

2.3.4 Command

Định nghĩa

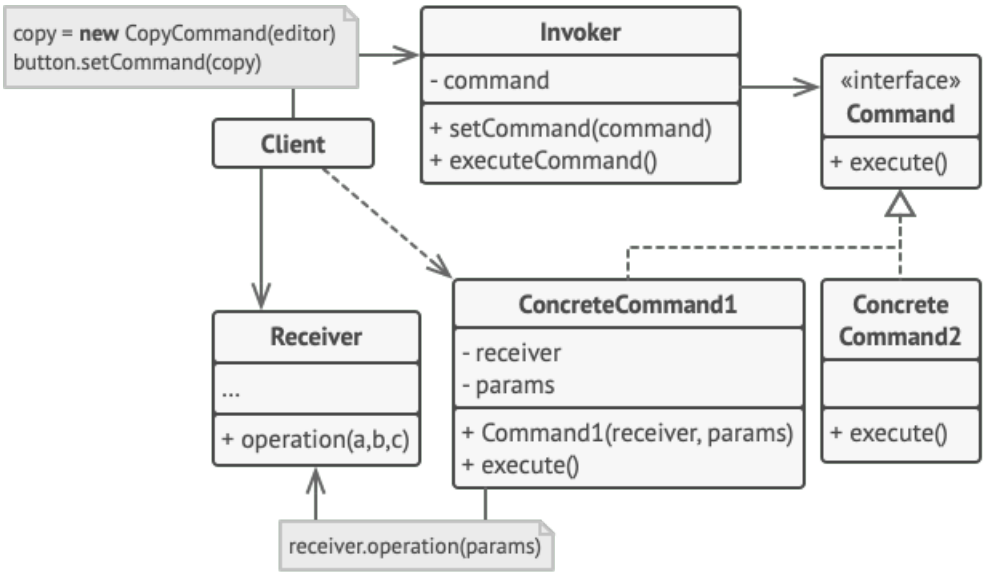
Command Pattern chuyển đổi yêu cầu thành một đối tượng độc lập hàm chứa tất cả thông tin về yêu cầu đó. Việc chuyển đổi này cho phép chúng ta tham số hóa yêu cầu với các vai trò khác nhau, có thể hỗ trợ những tác vụ tưởng chừng không thể (undoable).

Command đóng vai trò là trung gian được tạo ra để lưu trữ các câu lệnh và trạng thái của đối tượng tại một thời điểm nào đó. Đối tượng Command cung cấp rõ ràng các mệnh lệnh, và những đối tượng nhận nhiệm vụ thực hiện mệnh lệnh sẽ được cung cấp phương thức để thực hiện chúng.

Cách sử dụng

Các thành phần trong Command Pattern:

- **Command**: là một interface hoặc abstract class, chứa một phương thức trừu tượng thực thi (execute) một hành động (operation). Request sẽ được đóng gói dưới dạng Command.
- **ConcreteCommand**: cầu nối giữa đối tượng Receiver với một hành động.
- **Client**: tiếp nhận yêu cầu từ phía người dùng, đóng gói yêu cầu thành ConcreteCommand thích hợp và thiết lập Receiver của nó.
- **Invoker**: tiếp nhận ConcreteCommand từ Client và thực thi yêu cầu.
- **Receiver**: thành phần xử lý logic.



Ví dụ:

Một hệ thống ngân hàng cung cấp ứng dụng cho khách hàng (client) có thể mở (open) hoặc đóng (close) tài khoản trực tuyến. Hệ thống này được thiết kế theo dạng module, mỗi module sẽ thực hiện một nhiệm vụ riêng của nó, chẳng hạn mở tài khoản (OpenAccount), đóng tài khoản (CloseAccount). Do hệ thống không biết mỗi module sẽ làm gì, nên khi có yêu cầu client (chẳng hạn clickOpenAccount, clickCloseAccount), nó sẽ đóng gói yêu cầu này và gọi module xử lý.

Command Pattern:

- Command (Command.java)

```
1  public interface Command {
2
3      void execute();
4  }
```

- Receiver (Account.java)

```
1  public class Account {
2      private String name;
3
4      public Account(String name) {
5          this.name = name;
6      }
7
8      public void open() {
9          System.out.println("Account [" + name + "] Opened\n");
10     }
11
12     public void close() {
13         System.out.println("Account [" + name + "] Closed\n");
14     }
15 }
```

- ConcreteCommand (OpenAccount.java và CloseAccount.java)

```
1  public class OpenAccount implements Command {
2
3      private Account account;
4
5      public OpenAccount(Account account) {
6          this.account = account;
7      }
8
9      @Override
10     public void execute() {
11         account.open();
12     }
13 }
```

```java
public class CloseAccount implements Command {

    private Account account;

    public CloseAccount(Account account) {
        this.account = account;
    }

    @Override
    public void execute() {
        account.close();
    }
}
```

- Invoker (BankApp.java)

```java
public class BankApp {

    private Command openAccount;
    private Command closeAccount;

    public BankApp(Command openAccount, Command closeAccount) {
        this.openAccount = openAccount;
        this.closeAccount = closeAccount;
    }

    public void clickOpenAccount() {
        System.out.println("User click open an account");
        openAccount.execute();
    }

    public void clickCloseAccount() {
        System.out.println("User click close an account");
        closeAccount.execute();
    }
}
```

- Client (Client.java)

```java
public class Client {
    public static void main(String[] args) {
        Account account = new Account("LeHoangPhuc");

        Command open = new OpenAccount(account);
        Command close = new CloseAccount(account);
        BankApp bankApp = new BankApp(open, close);

        bankApp.clickOpenAccount();
        bankApp.clickCloseAccount();
    }
}
```

Kết quả chạy

```
User click open an account
Account [LeHoangPhuc] Opened

User click close an account
Account [LeHoangPhuc] Closed
```

Một ví dụ khác:

Ứng dụng văn bản cần một chức năng để thêm hoặc lưu trữ những hành động undo hay redo.

Lớp Document chỉ cung cấp phương thức ghi thêm một dòng văn bản mới hoặc xóa một dòng văn bản đã ghi trước đó.

Chúng ta sẽ xây dựng một interface Command để cung cấp hành động undo/redo. Để sử dụng Command chúng ta cần một DocumentInvoker, lớp này sử dụng tính năng của Stack (ngăn xếp) để lưu lại lịch sử những lần thêm mới và những lần xóa, tương ứng với undoCommands và redoCommands.

Stack là một cấu trúc dữ liệu trừu tượng hoạt động theo nguyên lý "vào sau ra trước" (Last In First Out (LIFO)), gồm có các phương thức cơ bản sau:

- **push()**: thêm phần tử trên đỉnh của Stack.
- **pop()**: xóa phần tử trên đỉnh của Stack và trả về phần tử bị xóa.
- **peek()**: lấy phần tử trên đỉnh của Stack, nhưng không xóa nó khỏi Stack.
- **clear()**: xóa tất cả các phần tử trong Stack.
- **isEmpty()**: kiểm tra Stack có chứa phần tử nào không.

Command Pattern:

- Command (Command.java)

```java
public interface Command {
    void undo();
    void redo();
}
```

- ConcreteCommand (DocumentEditorCommand.java)

```java
public class DocumentEditorCommand implements Command {

    private Document document;
    private String text;

    public DocumentEditorCommand(Document document, String text) {
        this.document = document;
        this.text = text;
        this.document.write(text);
    }

    public void undo() {
        document.eraseLast();
    }

    public void redo() {
        document.write(text);
    }
}
```

- Invoker (DocumentInvoker.java)

```java
import java.util.Stack;

public class DocumentInvoker {
    private Stack<Command> undoCommands = new Stack<>();
    private Stack<Command> redoCommands = new Stack<>();
    private Document document = new Document();

    public void undo() {
        if (!undoCommands.isEmpty()) {
            Command cmd = undoCommands.pop();
            cmd.undo();
            redoCommands.push(cmd);
        } else {
            System.out.println("Nothing to undo");
        }
    }

    public void redo() {
        if (!redoCommands.isEmpty()) {
            Command cmd = redoCommands.pop();
            cmd.redo();
            undoCommands.push(cmd);
        } else {
            System.out.println("Nothing to redo");
        }
    }

    public void write(String text) {
        Command cmd = new DocumentEditorCommand(document, text);
        undoCommands.push(cmd);
        redoCommands.clear();
    }

    public void read() {
        document.readDocument();
    }
}
```

- Receiver (Document.java)

```java
import java.util.Stack;

public class Document {
    private Stack<String> lines = new Stack<>();

    public void write(String text) {
        lines.push(text);
    }

    public void eraseLast() {
        if (!lines.isEmpty()) {
            lines.pop();
        }
    }

    public void readDocument() {
        for (String line : lines) {
            System.out.println(line);
        }
    }
}
```

- Client (Client.java)

```java
public class Client {

    public static void main(String[] args) {
        DocumentInvoker instance = new DocumentInvoker();
        instance.write("The 1st text. ");
        instance.undo();
        instance.read(); // EMPTY

        instance.redo();
        instance.read(); // The 1st text.

        instance.write("The 2nd text. ");
        instance.write("The 3rd text. ");
        instance.read(); // The 1st text. The 2nd text. The 3rd text.
        instance.undo(); // The 1st text. The 2nd text.
        instance.undo(); // The 1st text.
        instance.undo(); // EMPTY
        instance.undo(); // Nothing to undo
    }
}
```

Kết quả chạy

```
The 1st text.
The 1st text.
The 2nd text.
The 3rd text.
Nothing to undo
```

Ưu nhược điểm của Command Pattern

- Ưu điểm
  - Cho phép đóng gói yêu cầu thành đối tượng, dễ dàng chuyển dữ liệu giữa các thành phần hệ thống, có thể thiết lập cơ chế undo/redo.
  - Có thể bổ sung những Command mới vào ứng dụng mà không ảnh hưởng tới hệ thống; có thể gom các Command đơn giản thành một tập hợp phức tạp hơn.
- Nhược điểm
  - Khiến code trở nên phức tạp hơn, sinh ra các lớp mới gây phức tạp cho mã nguồn.

2.3.5 Iterator

Định nghĩa

Iterator Pattern cung cấp phương thức truy cập các phần tử của một đối tượng tổng hợp (aggregate object) một cách tuần tự (sequentially) mà không để lộ thuộc tính cơ bản của nó (danh sách, ngăn xếp, cây, v.v.).

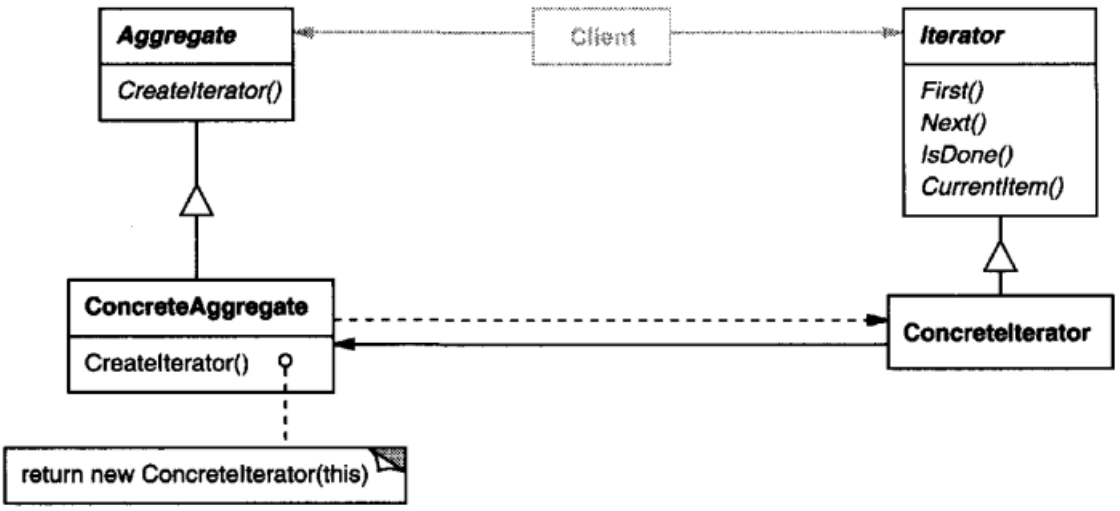
Tập hợp là một trong những kiểu dữ liệu được sử dụng nhiều nhất trong lập trình. Hầu hết các tập hợp lưu trữ các phần tử của chúng trong các danh sách đơn giản. Tuy nhiên, một số trong số chúng dựa trên ngăn xếp, cây, đồ thị và các cấu trúc dữ liệu phức tạp khác.

Nhưng bất kể tập hợp được cấu trúc như thế nào, nó phải cung cấp một số cách truy cập các phần tử của nó để mã khác có thể sử dụng các phần tử này. Vì vậy, rất khó để duyệt từng phần tử của tập hợp mà không cần truy cập lặp lại các phần tử giống nhau. Để làm được điều đó, chúng ta phải nghĩ tới Iterator Pattern.

Cách sử dụng

Các thành phần của Iterator Pattern:

- **Iterator**: là interface hay abstract class khai báo các hoạt động cần thiết để thao tác trên tập hợp (thêm/bớt phần tử, truy xuất vị trí, tính số phần tử, v.v.).
- **Concrete Iterator**: thực hiện các giải thuật cụ thể để thao tác trên tập hợp. Đối tượng trình lặp sẽ tự theo dõi tiến trình duyệt. Điều này cho phép một số trình lặp duyệt qua cùng một tập hợp độc lập với nhau.
- **Aggregate**: là interface hay abstract class khai báo các phương thức để có được các trình lặp tương thích với tập hợp. Kiểu trả về của các phương thức phải được khai báo dưới dạng giao diện Iterator để các tập hợp cụ thể có thể trả về các loại trình lặp khác nhau.
- **ConcreteAggregate**: trả về các phiên bản mới của một lớp trình lặp cụ thể mỗi khi Client yêu cầu.
- **Client**: làm việc với cả tập hợp Collection và trình lặp Iterator thông qua giao diện của chúng.



Ví dụ:

Ví dụ sau đây là một chương trình nhỏ dùng để duyệt qua các phần tử trong một danh sách cho trước, các phần tử là thông tin về tên và năm sinh của sinh viên.

Iterator Pattern:

- Character khai báo thông tin sinh viên.

```
#include <iostream>
#include <string>
using namespace std;

class Character
{
public:
    Character(string name, int year)
    {
        this->name = name;
        this->year = year;
    }
    string toString()
    {
        return (name + ", " + to_string(year));
    }

private:
    string name;
    int year;
};
```

- Iterator là interface với các thao tác cần thiết để duyệt danh sách.

```
#include <list>

template <typename Container>
class Iterator
{
public:
    Iterator(Container *_containerData)
    {
        containerData = _containerData;
        iter = containerData->charList.begin();
    }
    bool hasNext()
    {
        return (iter != containerData->charList.end());
    }
    list<Character>::iterator next()
    {
        return iter++;
    }

private:
    Container *containerData;
    list<Character>::iterator iter;
};
```

- Chronicle có chức năng thêm phần tử và một phương thức trả về Iterator để tạo trình lặp.

```
class Chronicle
{
public:
    void addCharacter(Character a)
    {
        charList.push_back(a);
    }
    Iterator<Chronicle> *createIterator()
    {
        return new Iterator<Chronicle>(this);
    }
    friend class Iterator<Chronicle>;

private:
    list<Character> charList;
};
```

- Client thao tác với cấu trúc dữ liệu này.

```
int main()
{
    Chronicle chronicle;

    chronicle.addCharacter(Character("LeHoangPhuc", 2003));
    chronicle.addCharacter(Character("PhucHoangLe", 2003));
    chronicle.addCharacter(Character("HoangPhucLe", 2003));
    chronicle.addCharacter(Character("PhucLeHoang", 2003));

    Iterator<Chronicle> *it = chronicle.createIterator();

    while (it->hasNext())
    {
        cout << it->next()->toString() << endl;
    }
    return 0;
}
```

Kết quả chạy

```
LeHoangPhuc, 2003
PhucHoangLe, 2003
HoangPhucLe, 2003
PhucLeHoang, 2003
```

Ưu nhược điểm của Iterator Pattern

- Ưu điểm
  - Làm sạch code client và các tập hợp bằng cách trích xuất các giải thuật duyệt cồng kềnh thành các lớp riêng biệt.
  - Có thể triển khai các kiểu tập hợp và trình lặp mới và chuyển chúng vào mã hiện có mà không vi phạm bất kỳ điều gì.
  - Có thể lặp song song trên cùng một tập hợp vì mỗi đối tượng trình lặp chứa trạng thái lặp riêng của nó, ngoài ra còn có thể trì hoãn một lần lặp lại và tiếp tục nó khi cần.
- Nhược điểm
  - Sẽ rất phí khi áp dụng Iterator Pattern nếu ứng dụng chỉ hoạt động với các tập hợp đơn giản.
  - Sử dụng trình lặp có thể kém hiệu quả hơn so với việc duyệt qua trực tiếp các phần tử của một số tập hợp chuyên biệt.

2.3.6 Mediator

Định nghĩa

Mediator Patern (mô hình trung gian) được sử dụng 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".

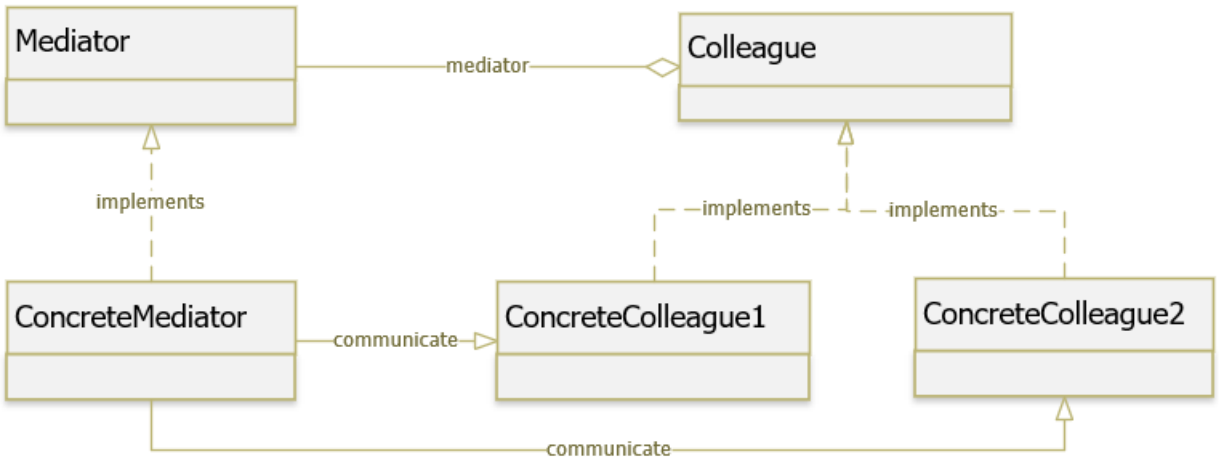
"Khớp nối lỏng lẻo" ở đây được hiểu là các đối tượng tương đồng không giao tiếp trực tiếp với nhau mà giao tiếp thông qua một trung gian, và nó cho phép thay thay đổi cách tương tác giữa chúng một cách độc lập.

Mediator Patern thúc đẩy mối quan hệ nhiều – nhiều (many-to-many) giữa các đối tượng với nhau để đạt đến được kết quả mong muốn.

Cách sử dụng

Các thành phần trong Mediator Pattern:

- **Colleague**: là một abstract class, giữ tham chiếu đến đối tượng Mediator.
- **ConcreteColleague**: cài đặt các phương thức của Colleague. Giao tiếp thông qua Mediator khi cần giao tiếp với Colleague khác.
- **Mediator**: là một interface, định nghĩa các phương thức để giao tiếp với các đối tượng Colleague.
- **ConcreteMediator**: cài đặt các phương thức của Mediator, biết và quản lý các Colleague object.



Ví dụ:

Mediator Pattern rất hữu ích trong việc quản lý tin nhắn trong nhóm chat của các trang mạng xã hội. Sau đây là một Chatroom đơn giản viết bằng Java, chương trình này dùng Mediator Pattern nhưng đã bỏ qua hiện thực các abstract class vì quy mô nhỏ.

Mediator Pattern:

- ConcreteMediator (ChatRoom.java)

```
1 import java.util.Date;
2
3 public class Chatroom {
4
5     public static void showMessage(ChatUser user, String message) {
6         System.out.println(new Date().toString() + "\n[" + user.getDisplayName() +
              "] : " + message);
7     }
8 }
```

- ConcreteColleague (ChatUser.java)

```java
public class ChatUser {
    private String name;
    private String nickname;

    public ChatUser(String name) {
        this.name = name;
        this.nickname = "";
    }

    public String getDisplayName() {
        if (nickname != "") return nickname;
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setNickName(String nickname) {
        this.nickname = nickname;
    }

    public void sendMessage(String message) {
        Chatroom.showMessage(this, message);
    }
}
```

- Demo.java

```java
public class Demo {
    public static void main(String[] args) {
        ChatUser phuc = new ChatUser("Phuc");
        phuc.setNickName("HaiLua03");
        ChatUser hacker = new ChatUser("Hacker");
        phuc.sendMessage("Hello");
        hacker.sendMessage("Hello");
    }
}
```

Kết quả chạy

```
Sun May 21 19:45:44 ICT 2023
[HaiLua03] : Hello
Sun May 21 19:45:44 ICT 2023
[Hacker] : Hello
```

Ta hãy xét thêm một phiên bản khác của hệ thống chat:

- Mediator (ChatMediator.java)

```java
public interface ChatMediator {

    void sendMessage(String msg, User user);
    void addUser(User user);
}
```

- ConcreteMediator (ChatMediatorImpl.java)

```java
import java.util.ArrayList;
import java.util.List;

public class ChatMediatorImpl implements ChatMediator {

    public ChatMediatorImpl(String groupName) {
        System.out.println("The group \"" + groupName + "\" is created");
    }

    private List<User> users = new ArrayList<>();

    @Override
    public void addUser(User user) {
        System.out.println("[" + user.name + "] joined this group");
        this.users.add(user);
    }

    @Override
    public void sendMessage(String msg, User user) {
        for (User u : this.users) {
            if (!u.equals(user)) {
                u.receive(msg);
            }
        }
    }
}
```

- Colleague (User.java)

```java
public abstract class User {
    protected ChatMediator mediator;
    protected String name;

    public User(ChatMediator med, String name) {
        this.mediator = med;
        this.name = name;
    }

    public abstract void send(String msg);

    public abstract void receive(String msg);

    @Override
    public int hashCode() {
        return name.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;

        }

        if (this.getClass() != obj.getClass()) {
            return false;
        }

        User user = (User) obj;
        return name.equals(user.name);
    }
}
```

- ConcreteColleague (UserImpl.java)

```java
public class UserImpl extends User {

    public UserImpl(ChatMediator mediator, String name) {
        super(mediator, name);
    }

    @Override
    public void send(String msg) {
        System.out.println("---");
        System.out.println("[" + this.name + "] is sending the message: " + msg);
        mediator.sendMessage(msg, this);
    }

    @Override
    public void receive(String msg) {
        System.out.println("[" + this.name + "] received the message: " + msg);
    }
}
```

- Client (ChatClient.java)

```java
public class ChatClient {

    public static void main(String[] args) {
        ChatMediator mediator = new ChatMediatorImpl("LTNC Group");

        User admin = new UserImpl(mediator, "Prof. Tuan Anh");
        User user1 = new UserImpl(mediator, "Le Hoang Phuc");
        User user2 = new UserImpl(mediator, "Hoang Phuc Le");
        User user3 = new UserImpl(mediator, "Phuc Le Hoang");

        mediator.addUser(admin);
        mediator.addUser(user1);
        mediator.addUser(user2);
        mediator.addUser(user3);

        admin.send("Hi All!");
        user1.send("Da em chao thay.");
    }
}
```

Kết quả chạy

```
The group "LTNC Group" is created
[Prof. Tuan Anh] joined this group
[Le Hoang Phuc] joined this group
[Hoang Phuc Le] joined this group
[Phuc Le Hoang] joined this group
—
[Prof. Tuan Anh] is sending the message: Hi All!
[Le Hoang Phuc] received the message: Hi All!
[Hoang Phuc Le] received the message: Hi All!
[Phuc Le Hoang] received the message: Hi All!
—
[Le Hoang Phuc] is sending the message: Da em chao thay.
[Prof. Tuan Anh] received the message: Da em chao thay.
[Hoang Phuc Le] received the message: Da em chao thay.
[Phuc Le Hoang] received the message: Da em chao thay.
```

Ưu nhược điểm của Mediator Pattern

- Ưu điểm
  - Giảm khớp nối giữa các thành phần, dễ dàng tái sử dụng các thành phần. Quản lý tập trung các thành phần và làm rõ được cách thức tương tác giữa các thành phần trong hệ thống.
  - 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 thành phần bằng quan hệ một-nhiều (one-to-many) giữa một Mediator với các thành phần đó.
- Nhược điểm
  - Tương tự như Facade Pattern, đối tượng Mediator có thể dễ dàng trở thành đối tượng thượng đế.

2.3.7 Memento

Định nghĩa

Memento Pattern cho phép lưu trữ và hồi phục các phiên bản cũ của một đối tượng mà không can thiệp vào nội dung của đối tượng đó.

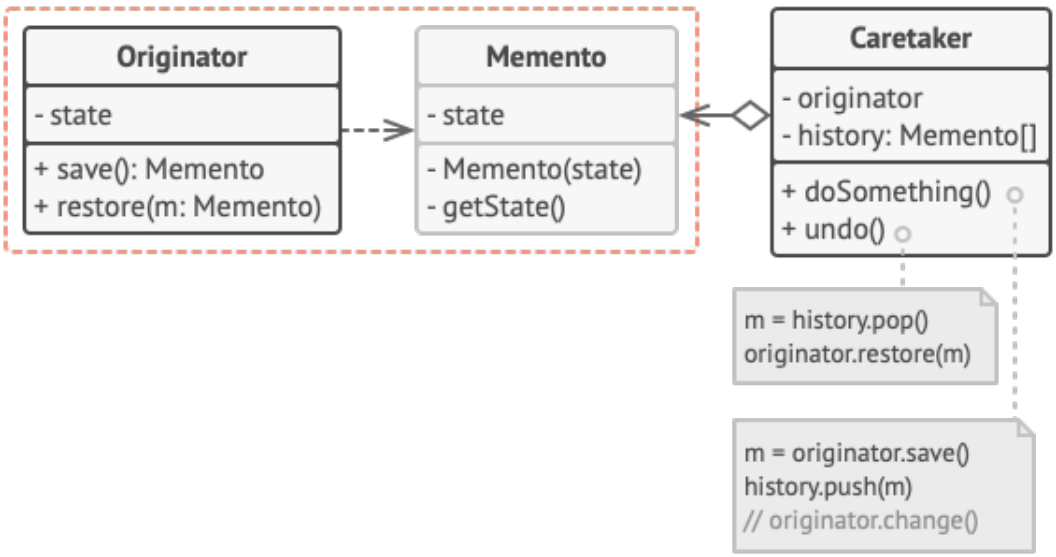
Dữ liệu trạng thái đã lưu trong đối tượng Memento không thể truy cập bên ngoài đối tượng được lưu và khôi phục. Điều này bảo vệ tính toàn vẹn của dữ liệu trạng thái đã lưu.

Memento được sử dụng để thực hiện thao tác undo. Điều này được thực hiện bằng cách lưu trạng thái hiện tại của đối tượng mỗi khi nó thay đổi trạng thái, từ đó chúng ta có thể khôi phục lại trong trường hợp có lỗi.

Cách sử dụng

Các thành phần trong Memento Pattern:

- **Originator**: đại diện cho đối tượng mà chúng ta muốn lưu. Sử dụng Memento để lưu và khôi phục trạng thái bên trong của nó.
- **Caretaker**: Không bao giờ thực hiện các thao tác trên nội dung của Memento và không kiểm tra nội dung. Caretaker chỉ giữ đối tượng Memento và chịu trách nhiệm bảo vệ an toàn cho các đối tượng. Để khôi phục trạng thái trước đó, nó trả về đối tượng Memento cho Originator.
- **Memento**: đại diện cho một đối tượng để lưu trữ trạng thái của Originator. Nó bảo vệ chống lại sự truy cập của các đối tượng khác ngoài Originator.
  - Lớp Memento cung cấp 2 interfaces: 1 interface cho Caretaker và 1 cho Originator. Interface Caretaker không được cho phép bất kỳ hoạt động hoặc bất kỳ quyền truy cập vào trạng thái nội bộ được lưu trữ bởi Memento. Interface Originator được truy cập bất kỳ biến trạng thái nào cần thiết để có thể khôi phục lại trạng thái trước đó.
  - Lớp Memento thường là một lớp bên trong của Originator. Vì vậy, Originator có quyền truy cập vào các trường của Memento, nhưng các lớp bên ngoài không có quyền truy cập vào các trường này.



Ví dụ:

Memeto Pattern được ứng dụng rất nhiều trong các ứng dụng soạn thảo văn bản hay các ứng dụng game có hỗ trợ autosave... Sau đây là một ví dụ về Memento Pattern sử dụng trong một ứng dụng game bằng C++ đơn giản.

- Các thư viện cần dùng cùng cấu trúc dữ liệu cho vị trí và vật phẩm của nhân vật

```cpp
#include <iostream>
#include <vector>

using namespace std;

class Inventory
{
public:
    vector<string> items;
    void addItem(string item)
    {
        items.push_back(item);
    }
    string getString()
    {
        string s = "";
        int count = 0;
        for (size_t i=0; i<items.size(); i++)
        {
            s += items[i] + ", ";
            count++;
        }
        if (count>0)
        {
            s.pop_back();
            s.pop_back();
        }
        return s;
    }
};

class Position
{
public:
    int x_pos;
    int y_pos;
    Position()
    {
        x_pos = 0; y_pos = 0;
    }
    Position(int x, int y)
    {
        x_pos = x; y_pos = y;
    }
};
```

- Trạng thái có thể lưu được của nhân vật

```
class State
{
public:
    State()
    {
        level = 1;
        health = 100;
    }
    void printState()
    {
        cout << "Character's state:\n";
        cout << "Level: " << level << ", Health: " << health
        << ", Location: (" << pos.x_pos << ", " << pos.y_pos << ")\n"
        << "Inventory: {" << inventory.getString() << "}\n";
    }
    void getDamage(int damage)
    {
        health -= damage;
    }
    void addItem(string item)
    {
        inventory.addItem(item);
    }
    void moveToPos(Position newPos)
    {
        pos = newPos;
    }
private:
    int level;
    int health;
    Position pos;
    Inventory inventory;
};
```

- Memento

```
class Memento
{
public:
    Memento(State _state) : state(_state) {}
    State getState()
    {
        return state;
    }
private:
    State state;
};
```

- GameCharacter đóng vai trò là Originator, thể hiện các tính năng của một nhân vật trong game

```
class GameCharacter
{
public:
    void printState()
    {
        state.printState();
    }
    void takeDamage(int damage)
    {
        state.getDamage(damage);
    }
    void addItem(string item)
    {
        state.addItem(item);
    }
    void moveToPos(Position newPos)
    {
        state.moveToPos(newPos);
    }
    Memento saveState()
    {
        return Memento(state);
    }
    void restore(Memento memento)
    {
        state = memento.getState();
    }
private:
    State state;
};
```

- CareTaker có hai chức năng là lưu trạng thái và khôi phục lại trạng thái trước đó.

```
class CareTaker
{
public:
    CareTaker(GameCharacter * _character) : character(_character) {}
    void saveState()
    {
        cout << "Saving state...\n";
        Memento memento = character->saveState();
        history.push_back(memento);
    }
    void undo()
    {
        cout << "Undoing state...\n";
        Memento memento = history.back();
        history.pop_back();
        character->restore(memento);
    }
private:
    GameCharacter * character;
    vector<Memento> history;
};
```

- Ví dụ game được chơi như sau:

```
int main()
{
    GameCharacter * gc = new GameCharacter;
    CareTaker * carer = new CareTaker(gc);
    gc->printState();

    gc->takeDamage(25);
    gc->addItem("Sword");
    gc->moveToPos(Position(3, 4));
    gc->printState();
    carer->saveState();

    gc->takeDamage(15);
    gc->addItem("Shield");
    gc->moveToPos(Position(9, 2));
    gc->printState();

    carer->undo();
    gc->printState();
    return 0;
}
```

Kết quả chạy

Character's state:
Level: 1, Health: 100, Location: (0, 0)
Inventory:
Character's state:
Level: 1, Health: 75, Location: (3, 4)
Inventory: Sword
Saving state...
Character's state:
Level: 1, Health: 60, Location: (9, 2)
Inventory: Sword, Shield
Undoing state...
Character's state:
Level: 1, Health: 75, Location: (3, 4)
Inventory: Sword

Ưu nhược điểm của Memento Pattern

- Ưu điểm
  - 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.
- Nhược điểm
  - Khi có một số lượng lớn Mementos được tạo ra có thể gặp vấn đề về bộ nhớ, hiệu suất của ứng dụng.
  - Caretakers phải theo dõi vòng đời của Originator để có thể hủy các Mementos không dùng nữa.
  - Hầu hết các ngôn ngữ hiện đại (cụ thể là dynamic programming languages như PHP, Python và Javascript) không thể đảm bảo trạng thái bên trong Memento được giữ không ai đụng tới.

2.3.8 Observer

Định nghĩa

Observer Pattern là một mẫu thiết kế phần mềm mà trong đó một chủ thể (gọi là Subject) duy trì một danh sách các thành phần phụ thuộc nó (gọi là Observer) và thông báo tới chúng một cách tự động về bất cứ thay đổi nào, thông báo thường được thực thi bằng cách gọi 1 hàm của chúng.

Mục tiêu của Observer Pattern là:

- Định nghĩa mối phụ thuộc một - nhiều (one-to-many) giữa các đối tượng để khi mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.
- Một đối tượng có thể thông báo đến một số lượng không giới hạn các đối tượng khác.

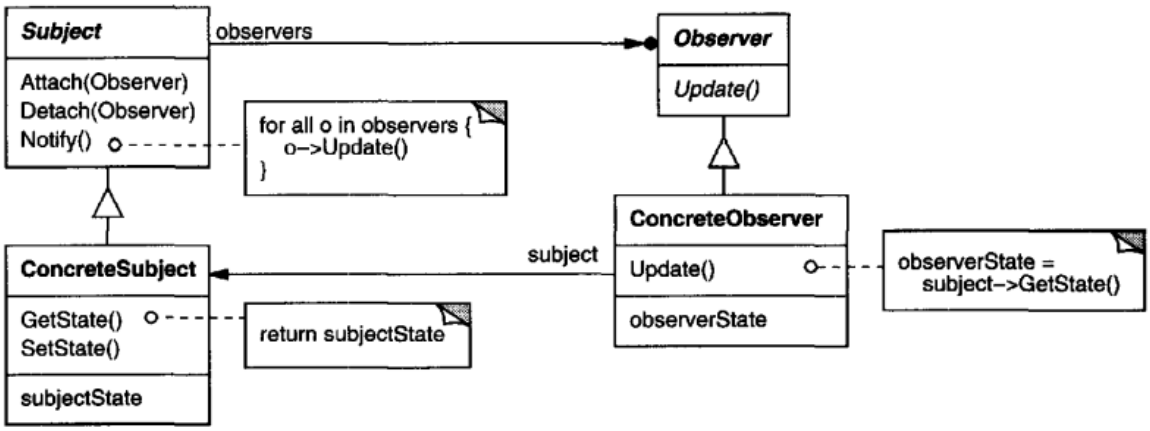
Observer Pattern được áp dụng khi:

- Sự thay đổi trạng thái ở 1 đối tượng có thể được thông báo đến các đối tượng khác mà không phải giữ chúng liên kết quá chặt chẽ.
- Cần mở rộng dự án với ít sự thay đổi nhất.

Cách sử dụng

Các thành phần trong Observer Pattern:

- **Subject**: cung cấp giao diện lưu trữ các thành phần phụ thuộc (các Observers), có thể thêm/bớt các Observers này.
- **Observer**: giao diện của các đối tượng được Subject cập nhật khi có sự thay đổi trạng thái.
- **ConcreteSubject**: lưu trữ trạng thái danh sách các ConcreateObserver, gửi thông báo đến các Observers của nó khi có sự thay đổi trạng thái.
- **ConcreteObserver**: lưu trữ trạng thái của Subject, thực thi việc cập nhật trạng thái theo Subject.



Khi Subject có sự thay đổi trạng thái, nó sẽ duyệt qua danh sách các Observers của nó và gọi phương thức cập nhật trạng thái ở từng Observer, có thể truyền chính nó vào phương thức để các Observers có thể lấy ra trạng thái của nó và xử lý.

Ví dụ:

Trong quân đội, tướng ra lệnh thì lính phải nghe theo, mỗi lần tướng quân ra lệnh, các sĩ tốt đều sẽ nhận được lệnh và thi hành theo lệnh vừa được ban. Đây là một ví dụ cho Observer Pattern trong lập trình hướng đối tượng. Vì vậy, ta có thể hiện thực một chương trình theo ví dụ này bằng Observer Pattern.

Observer Pattern:

- Subject (Subject.java)

```java
public interface Subject {

    public void attach(Observer observer);
    public void detach(Observer observer);
    public void announce(String message);
}
```

- Observer (Observer.java)

```java
public interface Observer {

    public void update(String message);
}
```

- Officer đóng vai trò là ConcreteSubject

```java
import java.util.ArrayList;
import java.util.List;

public class Officer implements Subject {

    private String name;
    private List<Observer> observers;

    public Officer(String name) {
        this.name = name;
        observers = new ArrayList<>();
    }

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
    }
    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
    }
    @Override
    public void announce(String message) {
        String notification = "Officer " + name + ": " + message;
        System.out.println("Officer " + name + " says: " + message);
        for (Observer observer : observers) {
            observer.update(notification);
        }
    }
}
```

- Soldier đóng vai trò là ConcreteObserver

```
public class Soldier implements Observer {
    private String name;
    public Soldier(String name) {
        this.name = name;
    }
    @Override
    public void update(String message) {
        System.out.println("Soldier " + name + " received order from " + message);
    }
}
```

- Client (Battle.java)

```
public class Battle {
    public static void main(String[] args) {
        Officer NguyenAnh = new Officer("Nguyen Anh");
        Soldier Le = new Soldier("Le");
        Soldier Hoang = new Soldier("Hoang");
        Soldier Phuc = new Soldier("Phuc");
        NguyenAnh.attach(Le);
        NguyenAnh.attach(Hoang);
        NguyenAnh.attach(Phuc);
        NguyenAnh.announce("Attack!");

        NguyenAnh.detach(Hoang);
        NguyenAnh.announce("Push forward!");
    }
}
```

Ưu nhược điểm của Observer Pattern

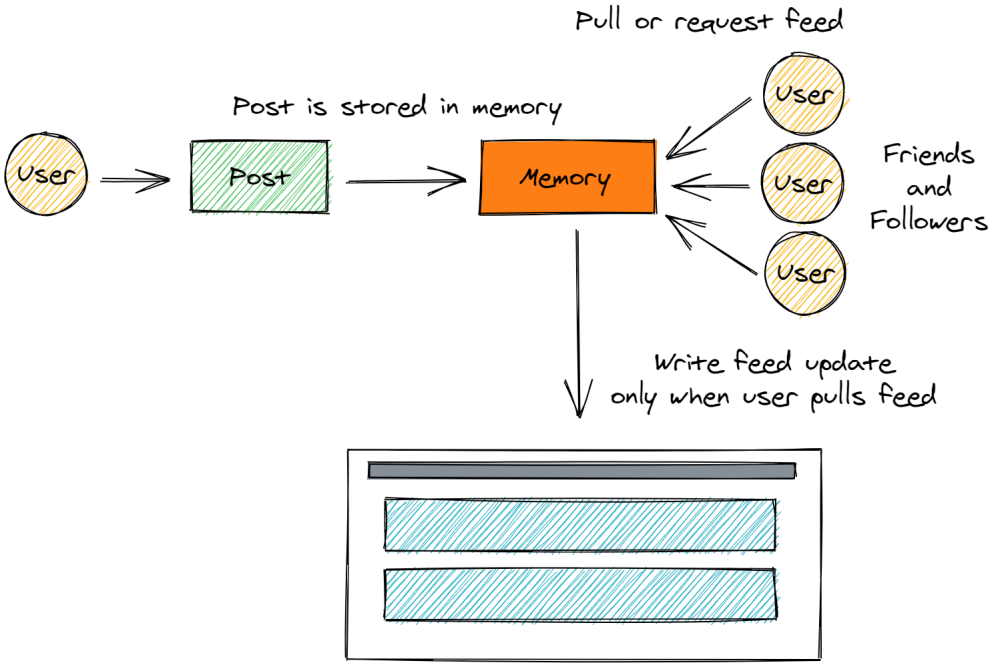
- Ưu điểm
  - Dễ mở rộng, nâng cấp.
  - Có thể thông báo đến các Observers mà không giữ liên kết chặt chẽ quá mức giữa chúng.
- Nhược điểm
  - Dù các Observers không giới hạn số lượng nhưng thứ tự được thông báo có thể rất ngẫu nhiên.

Mô hình Push và mô hình Pull

Trong các ứng dụng mạng xã hội lớn hiện nay như Facebook, Twitter, Instagram..., những nhà phát triển đã áp dụng Observer Pattern vào chức năng cập nhật news feed, đó là mô hình Push và mô hình Pull. Hai mô hình này giúp tăng tốc độ cập nhật news feed ngay cả khi một tài khoản người dùng theo dõi một số lượng rất lớn các tài khoản khác. Ta có thể nhận thấy tốc độ này mỗi khi chúng ta mở trang mạng xã hội lên, chỉ trong chốc lát ta đã có thể đọc được một bài viết của bạn bè hoặc người nổi tiếng mà chúng ta theo dõi vừa mới đăng độ chừng vài giây.

Mô hình Pull

Khi một người dùng đăng tải một bài viết, bài viết sẽ được lưu trong bộ nhớ của tài khoản. Kếu một người bạn hoặc một người theo dõi anh ta mở news feed của mình lên, một yêu cầu update feed hoặc một lệnh Pull sẽ được gọi và những bài viết mới nhất sẽ được nạp vào news feed.

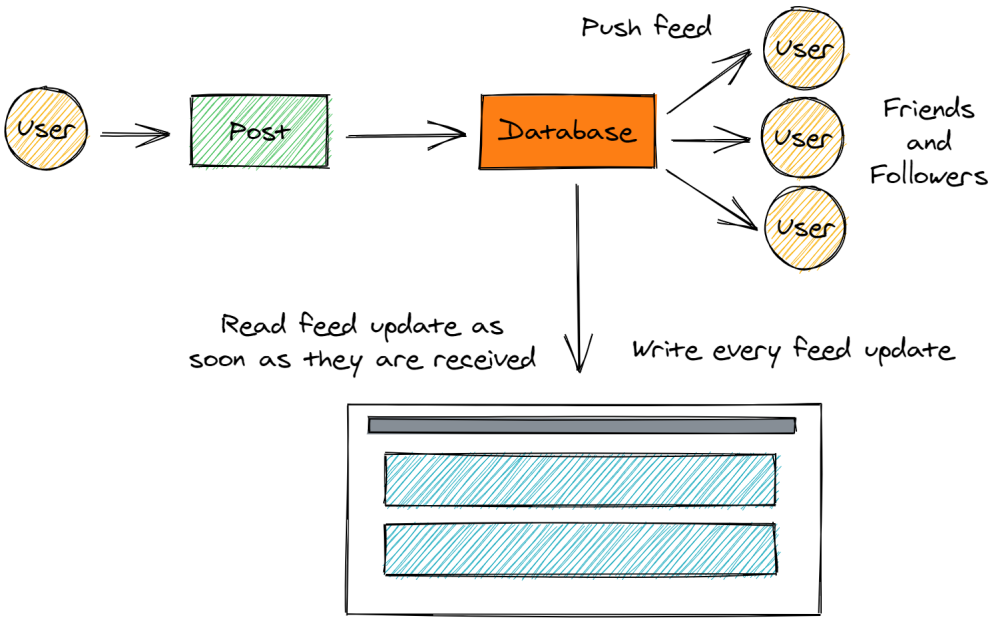


Ưu nhược điểm của mô hình Pull

- Ưu điểm
  - Tiết kiệm bộ nhớ, khi đăng bài thì chỉ cần lưu trữ trong database của người đó.
- Nhược điểm
  - Số lượng tài khoản followings của một người dùng càng nhiều sẽ kéo theo số lượng request càng nhiều.
  - Thời gian chờ render để ra news feed lâu.

Mô hình Push

Khi một người dùng đăng tải bài viết, mọi dữ liệu về bài viết sẽ được sao chép và gửi tới database (cơ sở dữ liệu) tất cả những followers và bạn bè của người đó. Điều này giúp hệ thống không phải duyệt qua từng người dùng trong danh sách những tài khoản đang theo dõi để kiểm tra xem có bất cứ cập nhật mới nào.



Ưu nhược điểm của mô hình Push

- Ưu điểm
  - Chỉ cần truy cập vào news feed của mình là có được dữ liệu.
  - Nhanh chóng nhận được những bài viết mới.
- Nhược điểm
  - Hao phí cơ sở dữ liệu: nếu một tài khoản người nổi tiếng có 2 triệu người theo dõi, khi người đó đăng 1 bài lên feed thì sẽ tạo ra 2 triệu bản sao (chưa tính tới việc người đăng chỉnh sửa bài viết vừa đăng lên).
  - Tốc độ nhận tin của một người follower có thể chậm hơn những người follower khác.

2.3.9 State

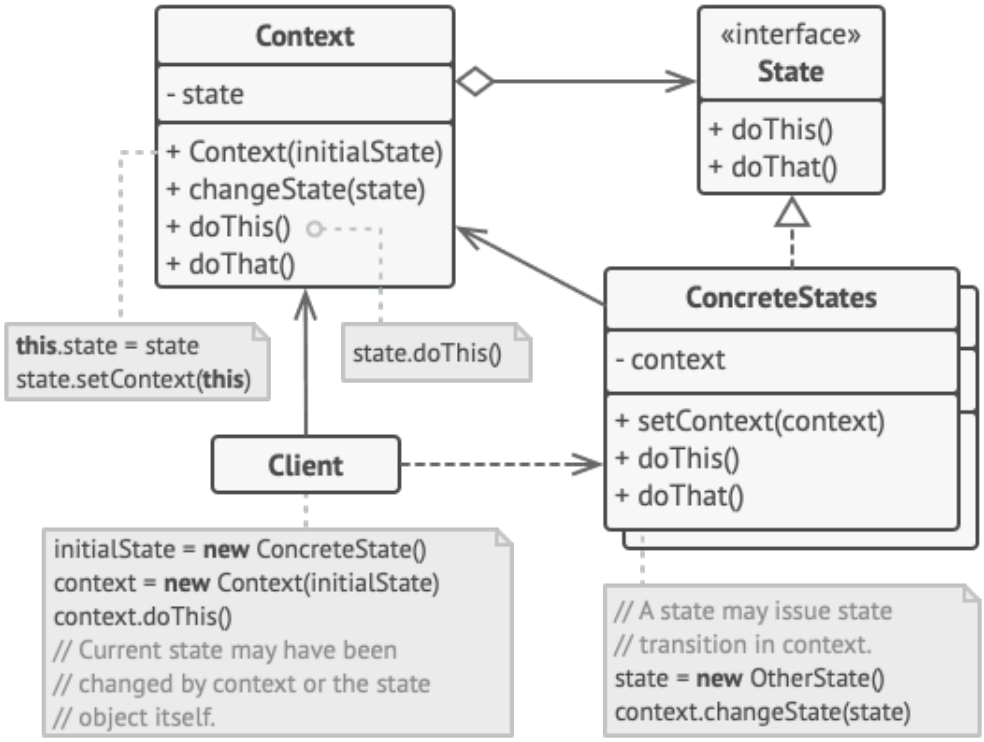
Định nghĩa

State Pattern cho phép một đối tượng thay đổi hành vi của nó khi trạng thái nội bộ của nó thay đổi. Đối tượng sẽ xuất hiện để thay đổi lớp của nó.

Cách sử dụng

Các thành phần trong State Pattern:

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



Lưu ý:

- Một đối tượng nên thay đổi hành vi của nó khi trạng thái bên trong của nó thay đổi.
- Mỗi State nên được xác định độc lập.
- Thêm các trạng thái mới sẽ không làm ảnh hưởng đến các trạng thái hoặc chức năng khác.

Ví dụ:

Giả sử chúng ta cần xây dựng một ứng dụng quản lý tài liệu (Document). Trong ứng dụng, một Document có thể có các trạng thái như: tạo mới (New), lưu (Save), đăng tải (Publish), Lưu trữ (Archive)...

Với những yêu cầu trên, ta có thể hiện thực chương trình qua đoạn code Java sau:

```
enum DocumentState {
    NEW, SAVE, PUBLISH, ARCHIVE
}

class DocumentService {
    private DocumentState state;

    public void setState(DocumentState state) {
        this.state = state;
    }

    public void handleRequest() {
        switch (state) {
        case NEW:
            System.out.println("Create a new document");
            break;
        case SAVE:
            System.out.println("Document is saved");
            break;
        case PUBLISH:
            System.out.println("Document has been published to cloud");
            break;
        case ARCHIVE:
            System.out.println("Document has been archived");
            break;
        default:
            break;
        }
    }
}

public class DocumentApp {

    public static void main(String[] args) {
        DocumentService service = new DocumentService();

        service.setState(DocumentState.NEW);
        service.handleRequest();

        service.setState(DocumentState.SAVE);
        service.handleRequest();

        service.setState(DocumentState.PUBLISH);
        service.handleRequest();
    }
}
```

Kết quả chạy

```
Create new document
Document is saved
Document has been published
```

Đối với hướng tiếp cận trên, ta có thể quản lý được một số lượng trạng thái của đối tượng. Tuy nhiên, khi cần nâng cấp lên nhiều trạng thái, ngoài việc thêm trạng thái, ta còn phải bổ sung vào đoạn code switch-case để quản lý các trạng thái vừa được thêm vào. Điều này khiến cho việc bảo trì và nâng cấp khó khăn vì phải chỉnh sửa rất nhiều nơi trong đoạn code. Do đó, để tránh việc phí phạm thời gian và công sức cho những đoạn code phức tạp không cần thiết, ta nên sử dụng State Pattern.

State Pattern:

- State (State.java)

```java
public interface State {
    void handleRequest();
}
```

- Các ConcreteStates

  NewState.java

```java
public class NewState implements State {
    @Override
    public void handleRequest() {
        System.out.println("Create a new document");
    }
}
```

  SaveState.java

```java
public class SaveState implements State {
    @Override
    public void handleRequest() {
        System.out.println("Document is saved");
    }
}
```

  PublishState.java

```java
public class PublishState implements State {
    @Override
    public void handleRequest() {
        System.out.println("Document has been published");
    }
}
```

  ArchiveState.java

```java
public class ArchiveState implements State {
    @Override
    public void handleRequest() {
        System.out.println("Document has been archived");
    }
}
```

- Context (DocumentContext.java)

```
1  public class DocumentContext {
2
3      private State state;
4
5      public void setState(State state) {
6          this.state = state;
7      }
8
9      public void applyState() {
10         this.state.handleRequest();
11     }
12 }
```

- Client (DocumentApp.java)

```
1  public class DocumentApp {
2
3      public static void main(String[] args) {
4          DocumentContext context = new DocumentContext();
5
6          context.setState(new NewState());
7          context.applyState();
8
9          context.setState(new SaveState());
10         context.applyState();
11
12         context.setState(new PublishState());
13         context.applyState();
14     }
15 }
```

Kết quả chạy

```
Create new document
Document is saved
Document has been published
```

Nếu chỉ xét kết quả chạy của hai hướng tiếp cận, ta chỉ thấy được kết quả là như nhau. Tuy nhiên, hướng tiếp cận sử dụng State Pattern giúp người lập trình khi muốn bổ sung trạng thái chỉ cần thêm một class dựa trên interface State mà không cần phải đụng tới bất kỳ đoạn code nào khác trong chương trình.

Ưu nhược điểm của State Pattern

- Ưu điểm
  - Tách biệt mỗi State tương ứng với 1 class riêng biệt.
  - Có thể thêm một State mới mà không ảnh hưởng đến State khác hay Context hiện có.
  - Giữ hành vi cụ thể tương ứng với trạng thái.
  - Giúp chuyển trạng thái một cách rõ ràng.
- Nhược điểm
  - Áp dụng Pattern này khi ứng dụng chỉ có một vài trạng thái hoặc ít khi thay đổi trạng thái sẽ gây ra lãng phí.

2.3.10 Strategy

Định nghĩa

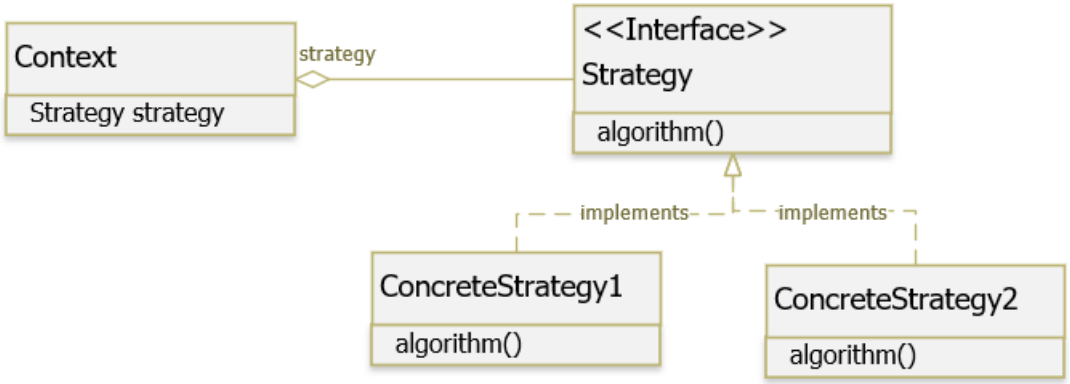
Strategy Pattern cho phép chúng ta định nghĩa tập hợp các giải thuật, đóng gói từng giải thuật, từ đó có thể dễ dàng thay đổi linh hoạt các giải thuật ở bên trong một đối tượng. Strategy Pattern giúp cho giải thuật biến đổi một cách độc lập khi người dùng sử dụng.

Ý nghĩa thực sự của Strategy Pattern là giúp tách rời phần xử lý một chức năng cụ thể ra khỏi đối tượng. Sau đó tạo ra một tập hợp các giải thuật để xử lý chức năng đó và lựa chọn giải thuật khi thực thi chương trình. Strategy Pattern thường được sử dụng để thay thế cho sự kế thừa, khi muốn chấm dứt việc theo dõi và chỉnh sửa một chức năng qua nhiều lớp con.

Cách sử dụng

Các thành phần trong Strategy Pattern:

- **Strategy**: định nghĩa các hành vi có thể có của một Strategy.
- **ConcreteStrategy**: cài đặt các hành vi cụ thể của Strategy.
- **Context**: chứa một tham chiếu đến đối tượng Strategy và nhận các yêu cầu từ Client, các yêu cầu này sau đó được ủy quyền cho Strategy thực hiện.



Ví dụ:

Một chương trình cung cấp các giải thuật sắp xếp khác nhau (quick sort, heap sort, selection sort, merge sort...) có thể được hiện thực bằng đoạn code C++ như sau:

Strategy Pattern:

- Interface SortStrategy định nghĩa chung các giải thuật có thể có trong chương trình

```
#include <iostream>
#include <vector>
using namespace std;

class SortStrategy {
public:
    virtual void sort(vector<int> list) = 0;
};
```

- Các ConcreteStrategy của SortStrategy cài đặt các giải thuật sắp xếp cụ thể

```
class QuickSort : public SortStrategy {
public:
    void sort(vector<int> list) {
        cout << "Quick sort\n";
        // Quick sort algorithm....
    }
};

class HeapSort : public SortStrategy {
public:
    void sort(vector<int> list) {
        cout << "Heap sort\n";
        // Heap sort algorithm....
    }
};

class SelectionSort : public SortStrategy {
public:
    void sort(vector<int> list) {
        cout << "Selection sort\n";
        // Selection sort algorithm....
    }
};

class MergeSort : public SortStrategy {
public:
    void sort(vector<int> list) {
        cout << "Merge sort\n";
        // Merge sort algorithm....
    }
};
```

- SortList đóng vai trò là Context, có tham chiếu tới Strategy

```
class SortList {
public:
    void setStrategy(SortStrategy * s) {
        this->strategy = s;
    }
    void addElement(int element) {
        this->items.push_back(element);
    }
    void sort() {
        this->strategy->sort(items);
    }
private:
    vector<int> items;
    SortStrategy * strategy;
};
```

- Client

```
int main() {
    SortList danhsach, danhsach1;
    danhsach.addElement(1);
    danhsach.addElement(2);
    danhsach.addElement(3);
    danhsach.setStrategy(new QuickSort());

    danhsach1.addElement(215);
    danhsach1.addElement(22);
    danhsach1.addElement(39);
    danhsach1.setStrategy(new MergeSort());

    danhsach.sort();
    danhsach1.sort();
    return 0;
}
```

Kết quả chạy

Quick sort
Merge sort

Ưu nhược điểm của Strategy Pattern

- Ưu điểm
  - Một giải thuật có thể được định nghĩa như một hệ thống phân cấp lớp và có thể được sử dụng thay thế cho nhau để thay đổi hành vi ứng dụng mà không thay đổi kiến trúc của nó.
  - Bằng cách đóng gói riêng biệt giải thuật, các giải thuật mới tuân theo cùng một interfacce có thể dễ dàng được giới thiệu.
  - Ứng dụng có thể chuyển đổi chiến lược tại thời điểm chạy.
  - Chiến lược cho phép khách hàng chọn giải thuật cần thiết mà không cần sử dụng quá nhiều câu lệnh "if-else".
  - Cấu trúc dữ liệu được sử dụng để thực hiện giải thuật hoàn toàn được gói gọn trong các lớp Strategy. Do đó, việc thực hiện một giải thuật có thể được thay đổi mà không ảnh hưởng đến lớp Context.
- Nhược điểm
  - Ứng dụng phải nhận thức được tất cả các chiến lược để chọn đúng chiến lược cho tình huống phù hợp.
  - Context và các lớp Strategy thường giao tiếp thông qua inteface được chỉ định bởi lớp cơ sở Strategy trừu tượng. Lớp cơ sở Strategy phải hiển thị interface cho tất cả các hành vi được yêu cầu, mà một số lớp Strategy cụ thể có thể không triển khai.
  - Trong hầu hết các trường hợp, ứng dụng định cấu hình Context với đối tượng Strategy được yêu cầu. Do đó, ứng dụng cần tạo và duy trì hai đối tượng thay cho một.

2.3.11 Visitor

Định nghĩa

Visitor Pattern ra đời nhằm giải quyết các hạn chế của Iterator Pattern, nó giúp chúng ta không cần quan tâm đến điểm bắt đầu và điều kiện kết thúc, từ đó chúng ta sẽ chỉ cần tập trung vào xử lý nghiệp vụ mà thôi.

Chúng ta cũng có thể tạo ra nhiều lớp Visitor khác nhau và sử dụng chúng mà không làm thay đổi bất kì thông tin hay cấu trúc dữ liệu.

Visitor được sử dụng khi:

- Cần thực hiện thao tác trên tất cả các phần tử của cấu trúc đối tượng phức tạp.
- Muốn làm sạch logic nghiệp vụ của các hành vi phụ trợ.
- 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.

Cách sử dụng

Ví dụ:

Giả sử ta cần thiết kế một ứng dụng quản lý một sàn thương mại điện tử đơn giản, cần duyệt qua các mặt hàng trong giỏ hàng và tính tổng giá tiền cần thanh toán. Một hướng làm đơn giản đó chính là khai thác tính đa hình của lập trình hướng đối tượng, ép kiểu các sản phẩm khác nhau để lưu trữ chung trong một danh sách trừu tượng, sau đó duyệt qua từng thành phần trong danh sách.

- Interface của sản phẩm (Product.java)

```
public interface Product
{
    public void printPrice();
    public int getPrice();
}
```

- Các sản phẩm cụ thể, là Book và Fruit (Book.java và Fruit.java)

```
public class Book implements Product
{
    private int price;
    private int id;

    public Book(int price,int id)
    {
        this.price = price;
        this.id = id;
    }

    @Override
    public void printPrice()
    {
        System.out.println("Book id " + this.id + " cost: " + this.price);
    }
    @Override
    public int getPrice()
    {
        return this.price;
    }
}
```

```
public class Fruit implements Product
{
    private int pricePerKg;
    private int weight;
    private String name;

    public Fruit(int pricePerKg, int weight, String name)
    {
        this.pricePerKg = pricePerKg;
        this.weight = weight;
        this.name = name;
    }

    @Override
    public void printPrice()
    {
        int finalPrice = this.pricePerKg * this.weight;
        System.out.println(this.name + " cost: " + finalPrice);
    }
    @Override
    public int getPrice()
    {
        return this.pricePerKg * this.weight;
    }
}
```

- Hóa đơn thanh toán giỏ hàng (Cart.java)

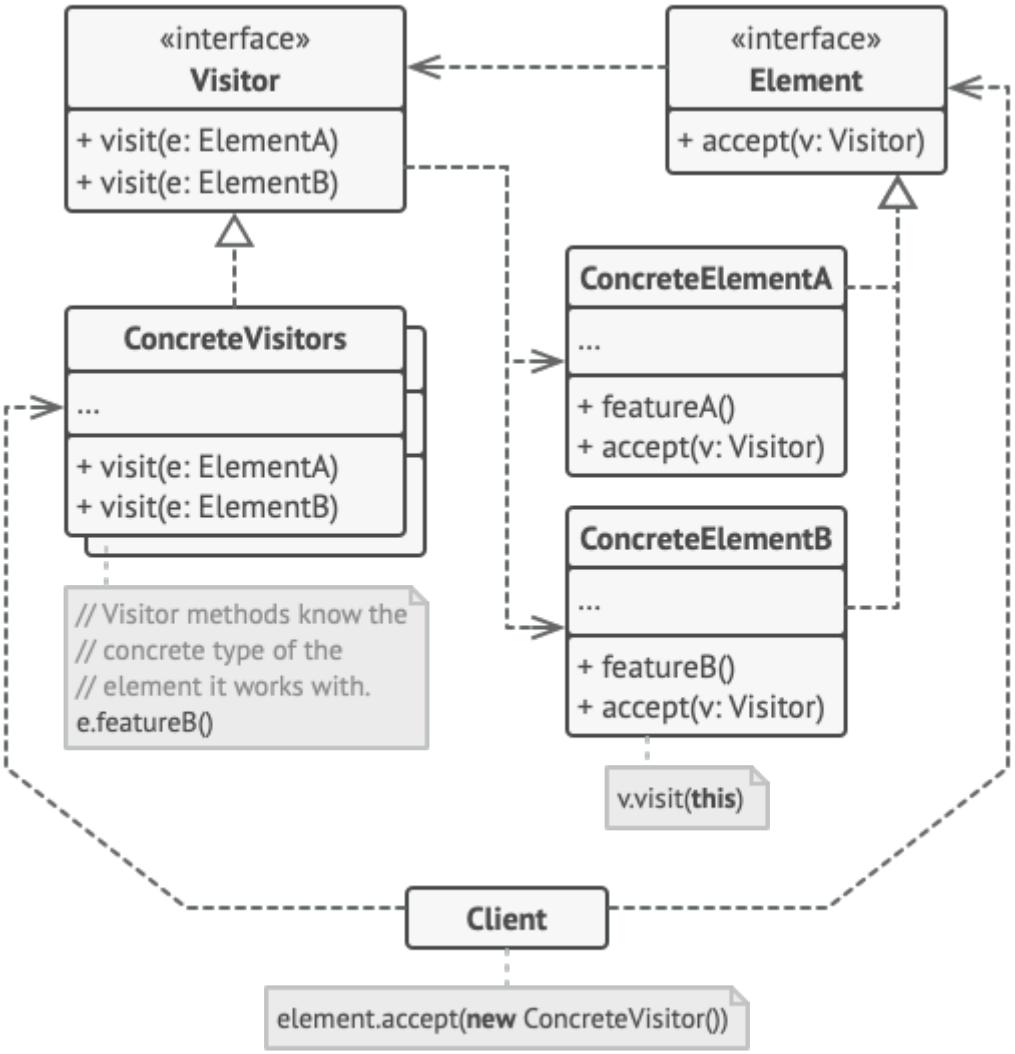
```
public class Cart {
    public static void main(String[] args) {
        Product[] cart = new Product[]{
            new Book(10,111),
            new Fruit(20,1,"Banana"),
            new Fruit(15,2,"Apple"),
            new Fruit(10,3,"Orange"),
            new Book(40,222),
            new Book(50,333)
        };
        for (Product product : cart) {
            product.printPrice();
        }
        int total = totalPrice(cart);
        System.out.println("Total cost: " + total);

    }
    private static int totalPrice(Product[] cart) {
        int sum = 0;
        for (Product product : cart) {
            sum += product.getPrice();
        }
        return sum;
    }
}
```

Kết quả chạy

```
Book id 111 cost: 10
Banana cost: 20
Apple cost: 30
Orange cost: 30
Book id 222 cost: 40
Book id 333 cost: 50
Total cost: 180
```

Nếu như interface Product có thêm một hay nhiều tính năng mới thì ta phải cập nhật các tính năng này cho tất cả các class con của nó, dù cho một số class con có thể không sử dụng tính năng này. Mỗi lần sửa hay cập nhật Product đều sẽ kéo theo một lượng thời gian và công sức đáng kể để sửa các class con.

Vì vậy, ta cần một phương pháp thiết kế mà interface bao quát mọi trường hợp, khi class con nào muốn thêm tính năng thì chỉ cần điều chỉnh ở class con đó mà không cần phải đụng vào interface. Phương pháp mà chúng ta cần tìm đó chính là Visitor Pattern.



Các thành phần trong Visitor Pattern:

- **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 (hay quá tải hàm), nhưng kiểu tham số của chúng phải khác nhau.
- **ConcreteVisitor**: 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.
- **ConcreteElement**: 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.

Visitor Pattern:

- Visitor (CartVisitor.java)

```java
public interface CartVisitor {

    int visit(Book book);
    int visit(Fruit fruit);
}
```

- ConcreteVisitor (ConcreteCartVisitor.java)

```java
public class ConcreteCartVisitor implements CartVisitor {

   @Override
   public int visit(Book book) {
      int cost = book.getPrice();
      System.out.println("Book id " + book.getID() + " cost: " + cost);
      return cost;
   }

   @Override
   public int visit(Fruit fruit) {
      int cost = fruit.getPricePerKg()*fruit.getWeight();
      System.out.println(fruit.getName() + " cost: " + cost);
      return cost;
   }
}
```

- Element (Product.java)

```java
public interface Product {

   public int accept(CartVisitor visitor);
}
```

Hàm accept() trả về giá trị là giá tiền của sản phẩm.

- ConcreteElement (Book.java và Fruit.java)

```java
public class Book implements Product {

   private int price;
   private int id;

   public Book(int cost, int id){
      this.price = cost;
      this.id = id;
   }

   public int getPrice() {
      return price;
   }

   public int getID() {
      return id;
   }

   @Override
   public int accept(CartVisitor visitor) {
      return visitor.visit(this);
   }
}
```

```java
public class Fruit implements Product {

   private int pricePerKg;
   private int weight;
   private String name;

   public Fruit(int priceKg, int weight, String name){
      this.pricePerKg=priceKg;
      this.weight=weight;
      this.name = name;
   }

   public int getPricePerKg() {
      return pricePerKg;
   }

   public int getWeight() {
      return weight;
   }

   public String getName(){
      return this.name;
   }

   @Override
   public int accept(CartVisitor visitor) {
      return visitor.visit(this);
   }

}
```

- Client (ShoppingClient.java)

```
1  public class ShoppingClient {
2
3     public static void main(String[] args) {
4        Product[] items = new Product[]{
5              new Book(20,1234),
6              new Book(100,5678),
7              new Fruit(10, 2, "Banana"),
8              new Fruit(5, 5, "Apple")
9          };
10       int total = calculatePrice(items);
11       System.out.println("Total cost: " + total);
12    }
13
14    private static int calculatePrice(Product[] items) {
15       CartVisitor visitor = new ConcreteCartVisitor();
16       int sum = 0;
17       for(Product item : items){
18          sum += item.accept(visitor);
19       }
20       return sum;
21    }
22 }
```

Kết quả chạy

```
Book id 1234 cost: 20
Book id 5678 cost: 100
Banana cost: 20
Apple cost: 25
Total cost: 165
```

Với Visitor Pattern, ta có thể chỉnh sửa các tính năng của các sản phẩm mà không phải đụng vào interface Product; trường hợp ta muốn thêm một loại sản phẩm, chỉ cần thêm nạp chồng thêm một hàm visit trong CartVisitor và hiện thực nó trong ConcreteCartVisitor.

Ưu nhược điểm của Visitor Pattern

- Ưu điểm
  - Có thể giới thiệu một hành vi mới hoạt động với các đối tượng của các lớp khác nhau mà không cần thay đổi các lớp này.
  - 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.
- Nhược điểm
  - Cần cập nhật tất cả Visitors mỗi khi một lớp được thêm vào hoặc xóa khỏi hệ thống phân cấp phần tử.
  - Các Visitors có thể thiếu quyền truy cập cần thiết vào các trường riêng tư và phương thức của các phần tử mà chúng phải làm việc cùng.

3 Tài liệu tham khảo

(1) Design Patterns: Elements of Reusable Object-Oriented Software - GoF

(2) Design Patterns - Refactoring.Guru

(3) Một ví dụ nhỏ về Factory method - Viblo

(4) Factory method for designing pattern - Geeks for Geeks

(5) Hướng dẫn Java Design Pattern – Abstract Factory - GP Coder

(6) Understanding Abstract Factory Pattern in C++ with an example - OpenGenus

(7) Builder Design Pattern - Viblo

(8) Builder Design Pattern - Geeks for Geeks

(9) Prototype in C++ - Refactoring.Guru

(10) Hướng dẫn Java Design Pattern – Prototype - GP Coder

(11) Implementation of Singleton Class in C++ - Geeks for Geeks

(12) Design Patterns - Singleton Pattern - BogoToBogo

(13) Explain C++ Singleton design pattern - Tutorials Point

(14) Singleton pattern là gì? Ứng vào trong lập trình - Blog Chia sẻ kĩ năng

(15) Mutex in C++ - cplusplus.com

(16) Hướng dẫn Java Design Pattern – Adapter - GP Coder

(17) Example of 'adapter' design pattern in C++ - GitHub

(18) Bridge Design Pattern - Trợ thủ đắc lực của Developers - Viblo

(19) Bridge Pattern: Có thể triển khai hệ thống thanh toán giống YOUTUBE - Youtube Tips Javascript

(20) C++ Design Pattern: Composite - Codecungnhau.com

(21) Composite Design Pattern - Programming Line

(22) Composite Pattern là gì? Sử dụng Composite Pattern trong PHP như thế nào? - CodeTuTam

(23) Decorator Design Pattern - Source Making

(24) Decorator - Refactoring.Guru

(25) Decorator Design Pattern - Trợ thủ đắc lực của Developers - Viblo

(26) Hướng dẫn Java Design Pattern – Facade - GP Coder

(27) Design Patterns - Facade Pattern - Tutorials Point

(28) Facade - Refactoring.Guru

(29) Hướng dẫn Java Design Pattern – Flyweight - TopDev

(30) Flyweight Design Pattern - Trợ thủ đắc lực của Developers - Viblio

(31) Hướng dẫn Java Design Pattern – Proxy - GP Coder

(32) Proxy Design Pattern - Geeks for Geeks

(33) Hướng dẫn Java Design Pattern – Interpreter - GP Coder

(34) Template Method Design Pattern - Trợ thủ đắc lực của Developers - Viblo

(35) Template Method Design Pattern in Java - DigitalOcean

(36) Hướng dẫn Java Design Pattern – Chain of Responsibility - GP Coder

(37) Chain of Responsibility – Python Design Patterns - Geeks for Geeks

(38) Command Design Pattern - Trợ thủ đắc lực của Developers - Viblo

(39) Hướng dẫn Java Design Pattern – Command - GP Coder

(40) Command - Refactoring.Guru

(41) Design Patterns: Iterator - Codecungnhau.com

(42) Iterator Design Pattern - Scaler

(43) Hướng dẫn Java Design Pattern – Mediator - GP Coder

(44) Mediator Design Pattern - Java Developer Central

(45) Design Patterns - Mediator Pattern - Tutorials Point

(46) Memento Design Pattern - Trợ thủ đắc lực của Developers - Viblo

(47) Hướng dẫn Java Design Pattern – Memento - GP Coder

(48) Memento Design Pattern - SBCODE Tutorials

(49) Design Pattern - Observer - Viblo

(50) Observer pattern được sử dụng triển khai news feed - Youtube Tips Javascript

(51) Push or Pull ngăn xếp công nghệ - Youtube Tips Javascript

(52) A Dive into the Facebook Newsfeed Architecture - AlgoDaily

(53) Hướng dẫn Java Design Pattern – State - GP Coder

(54) State - Refactoring.Guru

(55) Hướng dẫn Java Design Pattern – Strategy - GP Coder

(56) Strategy pattern cách sử dụng của Lv1 và Lv4 - Youtube Tips Javascript

(57) Tự học Design Pattern | Giới thiệu về Strategy Pattern và code ví dụ phần 1 - CafeDev

(58) Visitor Design Pattern - Trợ thủ đắc lực của Developers - Viblo

(59) Visitor Design Pattern in Java - DigitalOcean

(60) Visitor pattern: chuyến viếng thăm kỳ bí - Kata learns to code

Các đoạn code mẫu cho tất cả các Design Patterns có ở đường link sau:
`https://github.com/PhucLe03/OOP/tree/main/Design%20Patterns`
hoặc mã QR bên dưới.