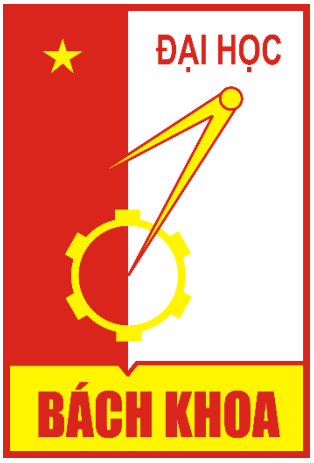
TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



BÁO CÁO PROJECT II

ĐỀ TÀI: DESIGN PATTERN

Giảng viên hướng dẫn: TS. Nguyễn Bá Ngọc

Sinh viên: Nguyễn Duy Hoài Lâm

MSSV: 20173225

Mục Lục

Giới Thiệu 3
I. Abstract Factory 4
II. Singleton 8
III. Builder 10
IV. Prototype 13
V. Adapter 17
VI. Proxy 21
VII. Composite 24
VIII. State 27
IX. Command 30
X. Observer 34
Tài liệu tham khảo: 37

Giới Thiệu

Design Pattern là một kỹ thuật trong lập trình hướng đối tượng, nó rất quan trọng và mọi lập trình viên muốn giỏi đều phải biết. Được sử dụng thường xuyên trong các ngôn ngữ OOP. Nó sẽ cung cấp cho bạn các "mẫu thiết kế", giải pháp để giải quyết các vấn đề chung, thường gặp trong lập trình. Các vấn đề mà bạn gặp phải có thể bạn sẽ tự nghĩ ra cách giải quyết nhưng có thể nó chưa phải là tối ưu. Design Pattern giúp bạn giải quyết vấn đề một cách tối ưu nhất, cung cấp cho bạn các giải pháp trong lập trình OOP.

Design Patterns không phải là ngôn ngữ cụ thể nào cả. Nó có thể thực hiện được ở phần lớn các ngôn ngữ lập trình, chẳng hạn như Java, C#, C++ thậm chí là Javascript hay bất kỳ ngôn ngữ lập trình nào khác.

Việc dử dụng Design Patterns giúp thiết kế của chúng ta linh hoạt, dễ dàng thay đổi và bảo trì hơn.

Vì thế trong đề tài này dưới sự hướng dẫn của Ngọc em nghiên cứu và tìm hiểu về các loại Design Patterns cơ bản trong C++.

Về cơ bản Design Patterns chia làm 3 nhóm với các mẫu tương ứng mà em sẽ nghiên cứu như sau.

- Nhóm 1: Creational Patterns (Nhóm khởi tạo: giúp chúng ta khởi tạo đối tượng)
  + Abstract Factory
  + Singleton
  + Builder
  + Prototype
- Nhóm 2: Structural Patterns (Nhóm cấu trúc: giúp ta thiết lập, định nghĩa quan hệ giữa các đối tượng)
  + Adapter
  + Proxy
  + Composite
- Nhóm 3: Behavioral Patterns (Nhóm hành vi: tập trung thực hiện các hành vi của đối tượng)
  + State
  + Command
  + Observer

I. Abstract Factory

Abstract Factory (AF) Pattern là Design Pattern thuộc nhóm Creational Patterns. AF là sự mở rộng của tính chất đa hình trong OOP. Mục tiêu hướng đến là có thể tạo ra các đối tượng mà chưa biết trước chính xác kiểu dữ liệu của chúng, giúp mã nguồn dễ bảo trì nếu có sự thay đổi.

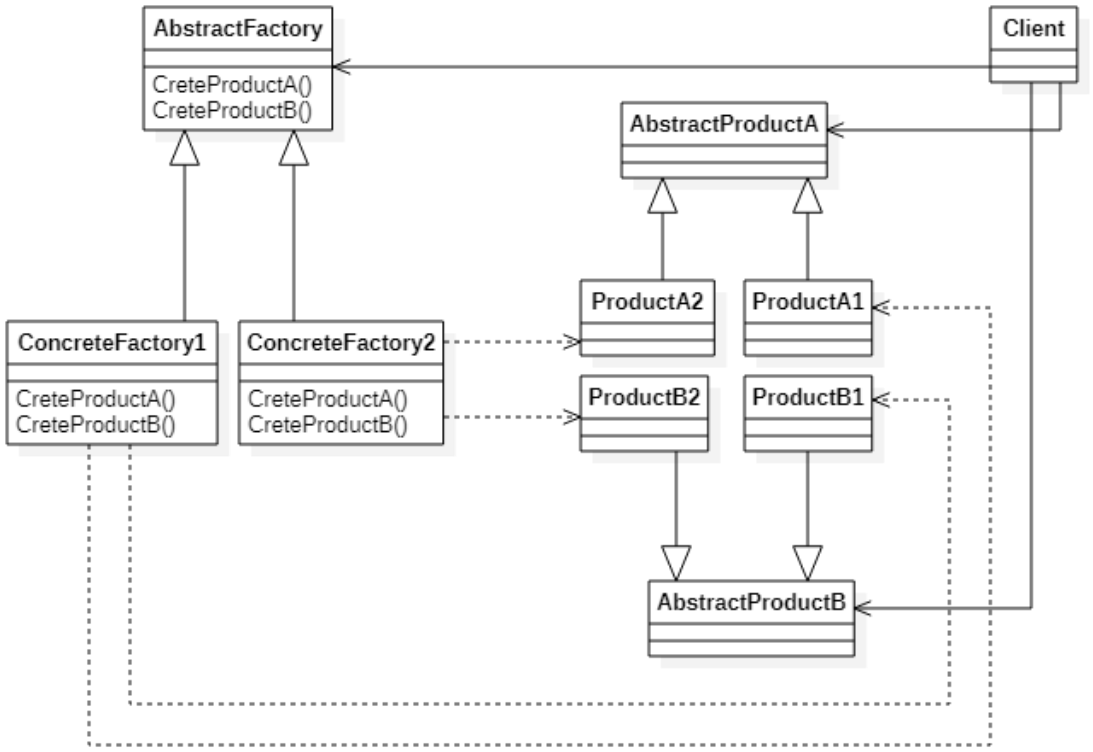
Sử dụng AF khi:

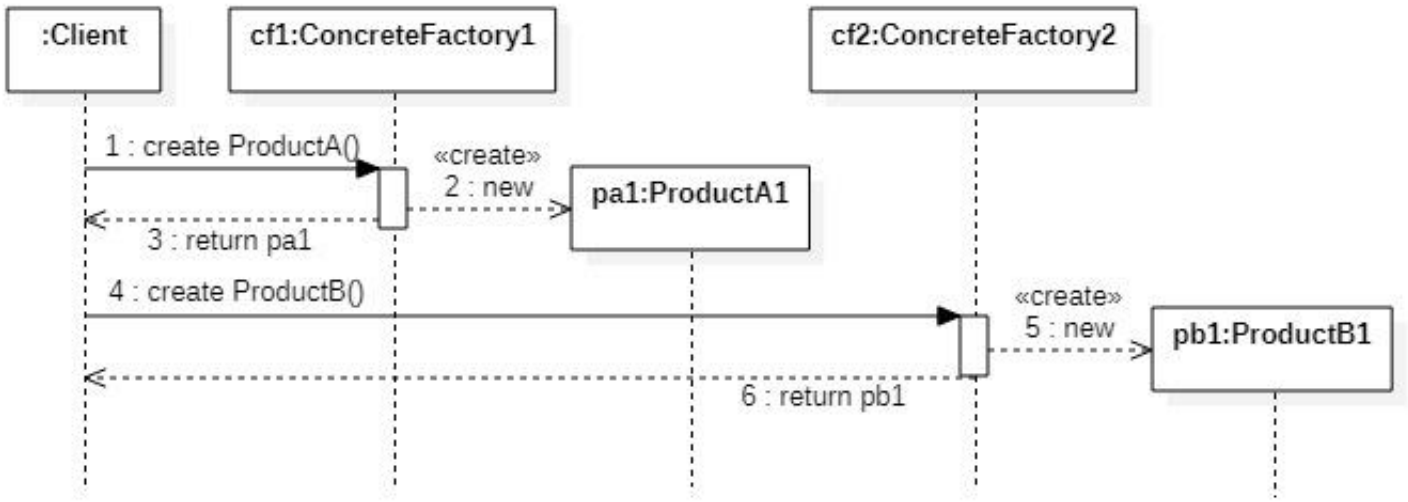
- Một hệ thống phải độc lập với cách các sản phẩm của nó được tạo ra.
- Hệ thống được cấu hình với một trong nhiều họ sản phẩm.
- Một nhóm các đối tượng sản phẩm liên quan được thiết kế để được sử dụng cùng nhau và bạn cần thực thi ràng buộc này.
- Bạn muốn cung cấp một thư viện sản phẩm và bạn chỉ muốn tiết lộ giao diện chứ không phải cách triển khai.

Thành phần:

- AcstractFactory: Khai báo một giao diện cho các hoạt động tạo ra các đối tượng AbstractProduct.
- ConcreteFactory: Thực hiện giao tiếp AbstractFactory để tạo ra các đối tượng product cụ thể.
- AbstractProduct: Khai báo một giao diện cho một loại đối tượng product.
- ConcreteProduct: Định nghĩa một đối tượng product được khởi tạo bởi ConcreteFactory tương ứng. Thực hiện giao diện AbstractProduct tương ứng.
- Client: Chỉ sử dụng các giao diện được khai báo bởi các lớp AbstractFactory và AbstractProduct.

Mô hình:


AbstractFactory
CreteProductA()
CreteProductB()
Client
AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreteProductA()
CreteProductB()
ConcreteFactory2
CreteProductA()
CreteProductB()
ProductB2
ProductB1
AbstractProductB


:Client
cf1:ConcreteFactory1
cf2:ConcreteFactory2
1 : create ProductA()
«create»
2 : new
pa1:ProductA1
3 : return pa1
4 : create ProductB()
«create»
5 : new
pb1:ProductB1
6 : return pb1

Code mẫu:

AcstractFactory:

```
class AbstracFactory
{
public:

	Plant* createPlant(int ID)
	{
		return getPlant(ID);
	}

	Animal* createAnimal(int ID)
	{
		return getAnimal(ID);
	}
};
```

ConcreteFactory Plant:

```
Plant*  getPlant(int ID)
{
	switch (ID)
	{
	case PLT_GRASS:
		printf("Plant Grass\n");
		return new Grass();
	case PLT_TREE:
		printf("Plant Tree\n");
		return new Tree();
	case PLT_FLOWER:
		printf("Plant Flower\n");
		return new Flower();
	default:
		break;
	}
}
```

ConcreteFactory Animal:

```
Animal* getAnimal(int ID)
{
	switch (ID)
	{
	case ANL_RABIT:
		printf("Animal Rabit\n");
		return new Rabit();
	case ANL_RAT:
		printf("Animal Rat\n");
		return new Rat();
	default:
		break;
	}
}
```

Product Plant:

```
class Plant
{
public:
	virtual void draw() = 0;
};
class Grass : public Plant
{
public:
	void draw() {};
};
class Tree : public Plant
{
public:
	void draw() {};
};
class Flower : public Plant
{
public:
	void draw() {};
};
```

Product Animal:

```
class Animal
{
public:
	virtual void draw() = 0;
};
class Rabit : public Animal
{
public:
	void draw() {};
};
class Rat : public Animal
{
public:
	void draw() {};
};
```

II. Singleton

Singleton là Design Pattern thuộc nhóm Creational Patterns đảm bảo chỉ duy nhất một thể hiện được tạo ra và nó sẽ cung cấp cho bạn một method để có thể truy xuất được thể hiện duy nhất đó mọi lúc mọi nơi trong chương trình.

Sử dụng Singleton khi chúng ta muốn:

- Đảm bảo rằng chỉ có một thể hiện của lớp.
- Việc quản lý việc truy cập tốt hơn vì chỉ có một thể hiện duy nhất.
- Có thể quản lý số lượng thể hiện của một lớp trong giới hạn chỉ định.

Thành phần:

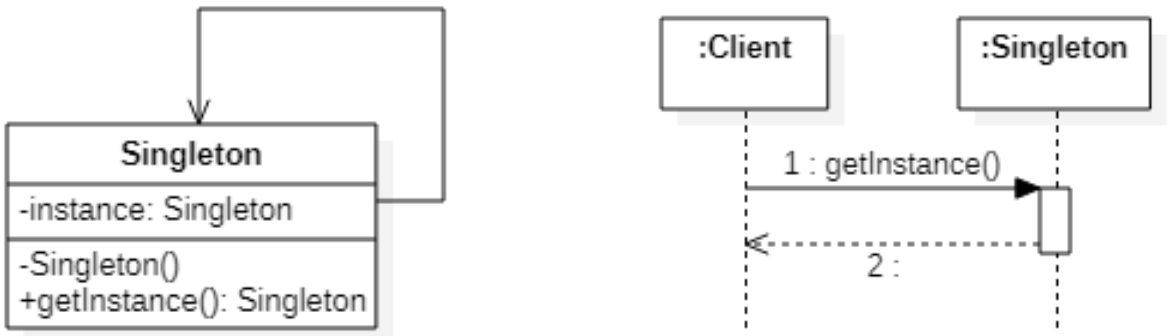
- Singleton: Định nghĩa một hoạt động Instance cho phép khách hàng truy cập vào thể hiện duy nhất của nó.

Khi implement singleton pattern phải đảm bảo rằng chỉ có một thể hiện duy nhất được tạo ra và thể hiện tại có thể dùng mọi lúc mọi nơi. Nói cách khác, khi xây dựng lớp cần một kỹ thuật để có thể truy xuất được vào các thành viên public của lớp mà không tạo ra một thể hiện nào (thông qua việc khởi tạo bên ngoài lớp).

Để chắc chắn rằng, không có bất kỳ thể hiện nào bên ngoài lớp được tạo ra, constructor của lớp đó sẽ có phạm vi truy cập là private. Điều đó cũng có nghĩa là thể hiện duy nhất được tạo ra thông qua chính lớp mà chúng ta xây dựng. Vì vậy, ta sử dụng thành phần static để thực hiện điều này. Ta tạo một thể hiện của lớp kiểu private static và một refractor trả về đối tượng thuộc chính lớp đó.

Trong C++, singleton định nghĩa một thuộc tính static là một con trỏ trỏ tới thể hiện duy nhất của lớp đó.

Mô hình:



Code mẫu:

Khai báo lớp:

```
class Singleton
{
   private:
         static Singleton* m_instance;
         Singleton();
   public:
         static Singleton* getInstance();
         void method();
};
```

Định nghĩa lớp:

```
Singleton* Singleton::m_instance = NULL;
Singleton* Singleton::getInstance()
{
   if (m_instance == NULL)
   {
         m_instance = new Singleton();
   }
   return m_instance;
}
void Singleton::method()
   {
   cout << "This is singleton parttern";
   }
```

Như vậy, thông qua các truy cập Singleton::getInstance() ta luôn có một đối tượng duy nhất, đồng thời có thể sử dụng nó để truy cập mọi thành phần public trong lớp. Ví dụ để gọi phương thức method trong lớp ta dùng Singleton::getInstance()->method().

III. Builder

Builder pattern là Design Pattern thuộc nhóm Creational Patterns, mô hình này cung cấp một trong những cách tốt nhất để tạo ra một đối tượng.Builder pattern là mẫu thiết kế đối tượng được tạo ra để xây dựng một đôi tượng phức tạp bằng cách sử dụng các đối tượng đơn giản và sử dụng tiếp cận từng bước, việc xây dựng các đối tượng độc lập với các đối tượng khác.

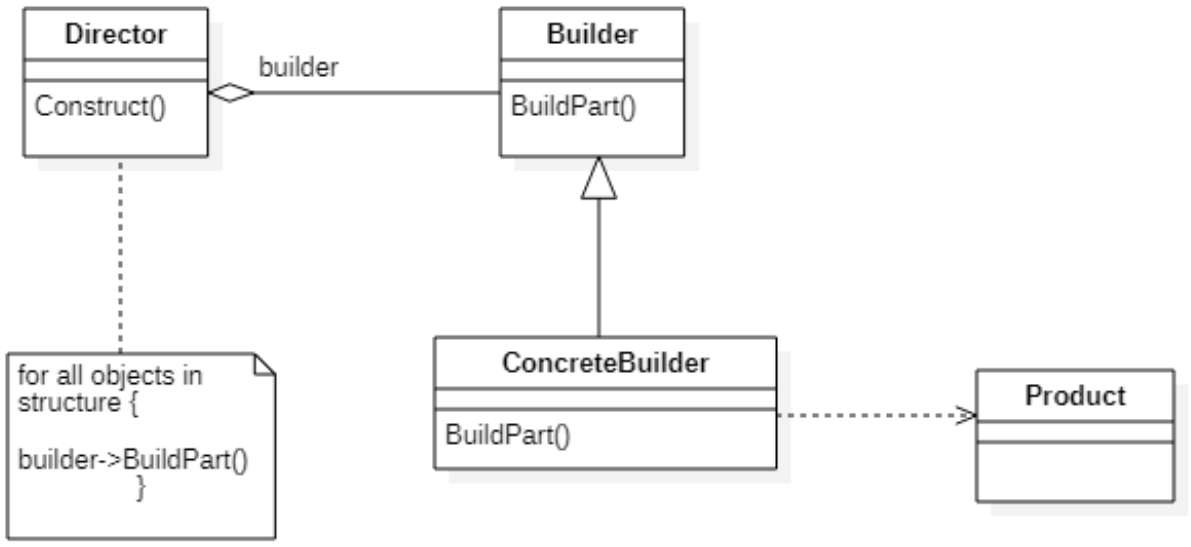
Sử dụng Builder khi:

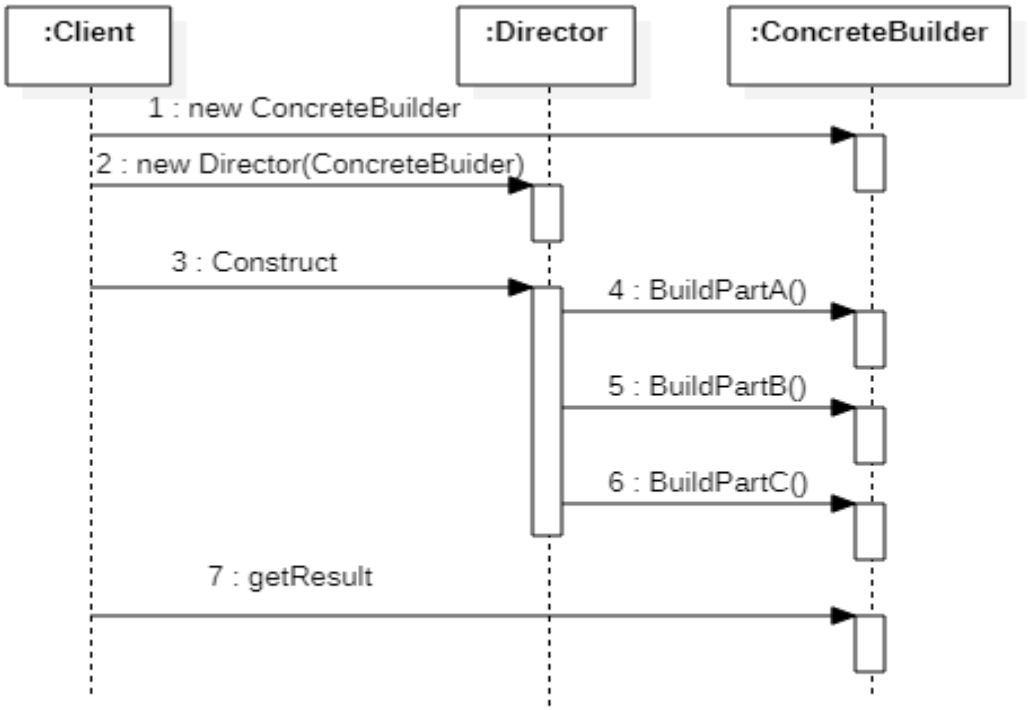
- Thuật toán tạo một đối tượng phức tạp phải độc lập với các bộ phận tạo nên đối tượng và cách chúng được lắp ráp.
- Quá trình khởi tạo phải cho phép các biểu diễn khác nhau cho đối tượng được khởi tạo.

Thành phần:

- Builder: Là thành phần định nghĩa lớp trừu tượng (abstract class) để tạo ra một hoặc nhiều thành phần của đối tượng Product.
- ConcreteBuilder: Là thành phần triển khai, cụ thể hoá các lớp trừu tượng để tạo ra các thành phần và tập hợp các thành phần đó với nhau, xác định và nắm giữ các thành phần mà nó tạo ra.
- Product: Thành phần này đại diện cho đối tượng phức tạp được tạo ra.
- Director: Thông báo với Builder khi nào một thành phần cần được tạo ra.

Mô hình:





Code mẫu:

Animal:

```
class Animal
{
   public:
   Body* body[4];
   Facade* facade;
   Food* food;

   void specifications()
   {
      cout << "Animal with" << "body ," << body[0]->weight << "facade" <<
      facade->color << "food " << food->plant << endl;
   }
};
```

Builder:

```
class Builder
{
      public:
      virtual Body* getBody() = 0;
      virtual Facade* getFacade() = 0;
      virtual Food* getFood() = 0;
};
```

Director:

```
class Director
{
   Builder* builder;

   public:
   void setBuilder(Builder* newBuilder)
   {
         builder = newBuilder;
   }

   Animal* getAnimal()
   {
         Animal* animal = new Animal();

         animal->facade = builder->getFacade();

         animal->food = builder->getFood();

         animal->body[0] = builder->getBody();
         animal->body[1] = builder->getBody();
         animal->body[2] = builder->getBody();
         animal->body[3] = builder->getBody();

         return animal;
   }
};
```

IV. Prototype

Prototype là Design Pattern thuộc kiểu Creational Patterns với ý tưởng tạo mới một đối tượng từ việc clone từ đối tượng được cung cấp trước hỗ trợ trong việc tạo các đối tượng lớn tốn nhiều tài nguyên.

Để cài đặt Prototype pattern ta tạo một phương thức Clone() ở lớp cha và triển khai ở lớp con.

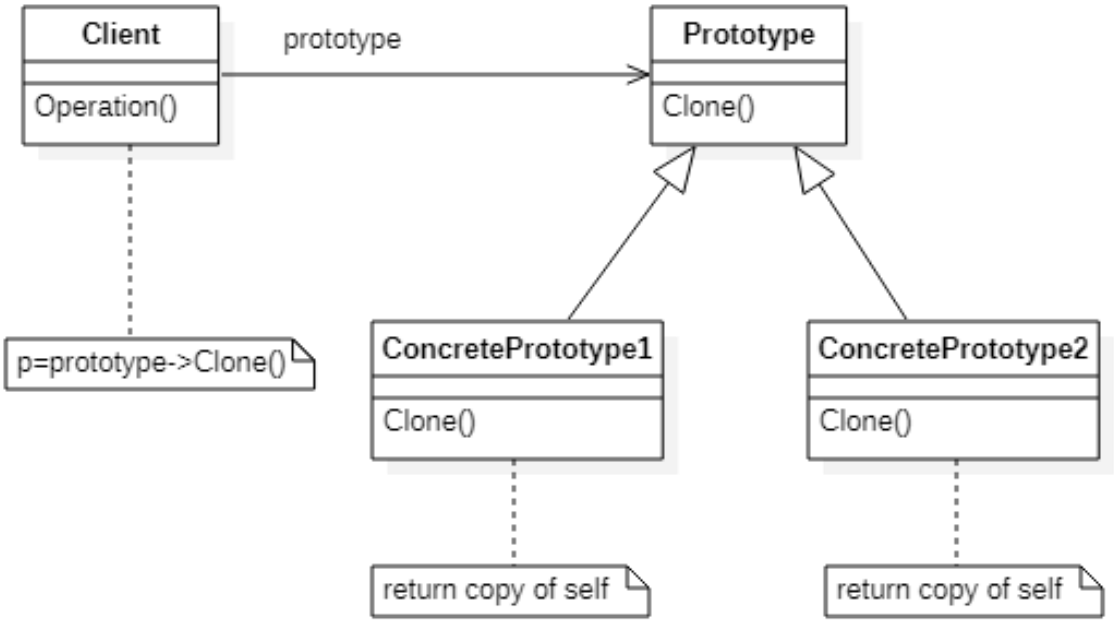
Sử dụng Prototype khi:

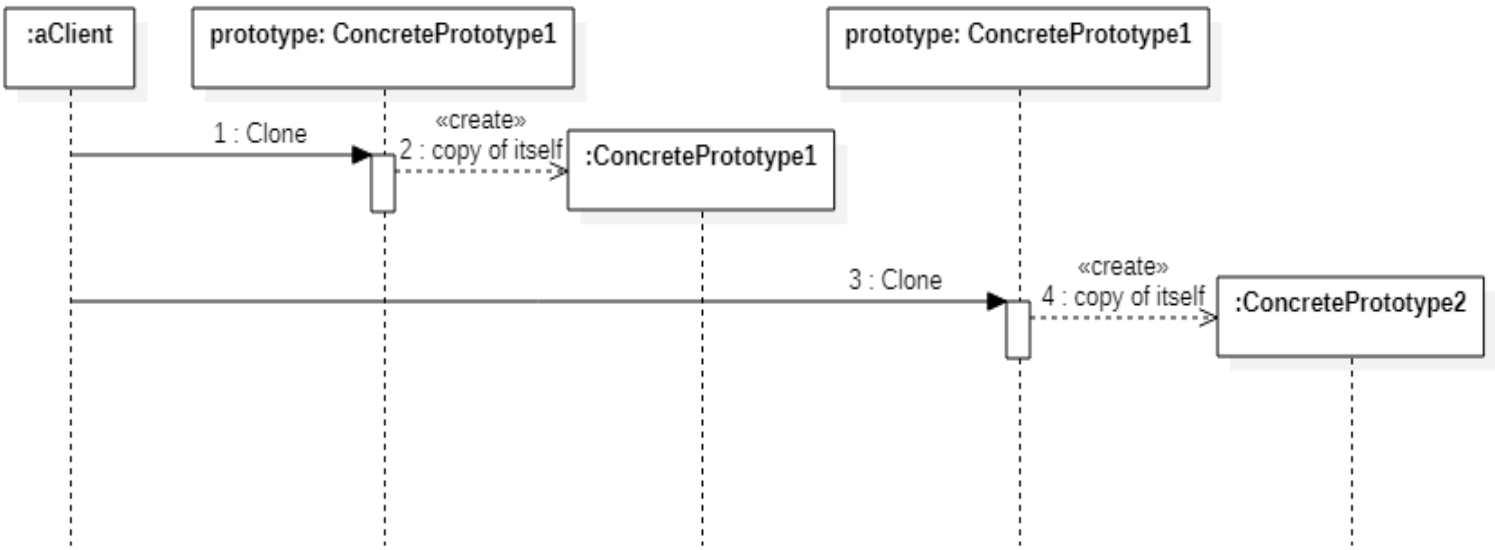
- Một hệ thống phải độc lập với cách các sản phẩm của nó được tạo ra.
- Khi các lớp để khởi tạo được chỉ định vào thời gian chạy.
- Để tránh xây dựng hệ thống phân cấp lớp tương đồng với phân cấp lớp của sản phẩm.
- Khi các thể hiện của một lớp có thể có một trong nhiều trạng thái kết hợp khác nhau. Có thể thuận tiện hơn khi cài đặt một số nguyên mẫu tương ứng và sao chép chúng thay vì khởi tạo lớp theo cách thủ công, mỗi lần với trạng thái thích hợp.

Thành phần:

- Client: Tạo mới đối tượng bằng cách gọi Prototype thực hiện Clone() chính nó.
- Prototype: Khai báo một inteface hoặc Abtract class cho việc clone chính nó.
- ConcretePrototype: Triển khai các phương thức cho việc Clone() chính nó.
- ObjectFactory: Quản lý các thể hiện của các Prototype và tạo bản sao của chúng.

Mô hình:


Client
Operation()
prototype
Prototype
Clone()
p=prototype->Clone()
ConcretePrototype1
Clone()
ConcretePrototype2
Clone()
return copy of self
return copy of self


:aClient
prototype: ConcretePrototype1
prototype: ConcretePrototype1
1 : Clone
«create»
2 : copy of itself
:ConcretePrototype1
3 : Clone
«create»
4 : copy of itself
:ConcretePrototype2

Code mẫu:

Prototype:

```
class Prototype
{
protected:
   std::string type;
   int value;

public:
   virtual Prototype* clone() = 0;

   std::string getType()
   {
         return type;
   }

   int getValue()
   {
         return value;
   }
};
```

ConcretePrototype1:

```
class ConcretePrototype1 : public Prototype
{
public:
   ConcretePrototype1(int number)
   {
         type = "Type1";
         value = number;
   }

   Prototype* clone()
   {
         return new ConcretePrototype1(*this);
   }
};
```

ConcretePrototype2:

```
class ConcretePrototype2 : public Prototype
{
public:
   ConcretePrototype2(int number)
   {
         type = "Type2";
         value = number;
   }

   Prototype* clone()
   {
         return new ConcretePrototype2(*this);
   }
};
```

ObjectFactory:

```
class ObjectFactory
{
   static Prototype* type1value1;
   static Prototype* type1value2;
   static Prototype* type2value1;
   static Prototype* type2value2;

public:
   static void initialize()
   {
         type1value1 = new ConcretePrototype1(1);
         type1value2 = new ConcretePrototype1(2);
         type2value1 = new ConcretePrototype2(1);
         type2value2 = new ConcretePrototype2(2);
   }

   static Prototype* getType1Value1()
   {
         return type1value1->clone();
   }

   static Prototype* getType1Value2()
   {
         return type1value2->clone();
   }

   static Prototype* getType2Value1()
   {
         return type2value1->clone();
   }

   static Prototype* getType2Value2()
   {
         return type2value2->clone();
   }
};
```

V. Adapter

Adapter Pattern là Design Pattern thuộc nhóm Structural Patterns. Adapter Pattern cho phép các inteface không liên quan tới nhau có thể làm việc cùng nhau. Đối tượng giúp kết nối các interface gọi là Adapter.

Sử dung Adapter khi:

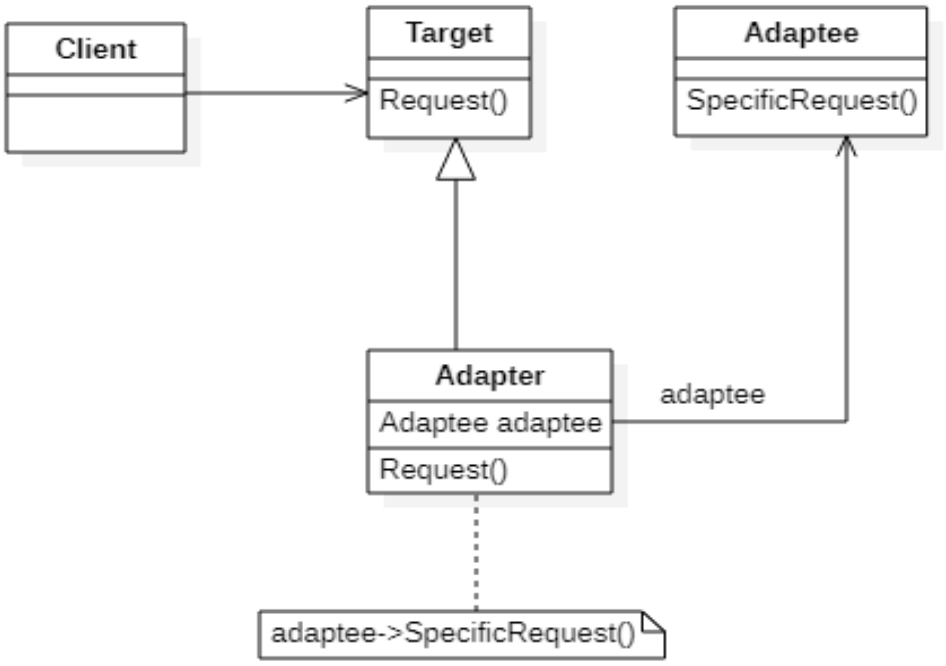
- Bạn muốn sử dụng một lớp hiện có và giao diện của nó không khớp với lớp bạn cần.
- Bạn muốn tạo một lớp có thể sử dụng lại, hợp tác với các lớp không liên quan hoặc không lường trước được, nghĩa là các lớp không nhất thiết phải có giao diện tương thích.
- Bạn cần sử dụng một số lớp con hiện có, nhưng không thực tế để điều chỉnh giao diện của chúng bằng cách phân lớp mỗi lớp. Một đối tượng adapter có thể điều chỉnh giao diện của lớp cha của nó.

Thành phần cơ bản của Adapter Pattern:

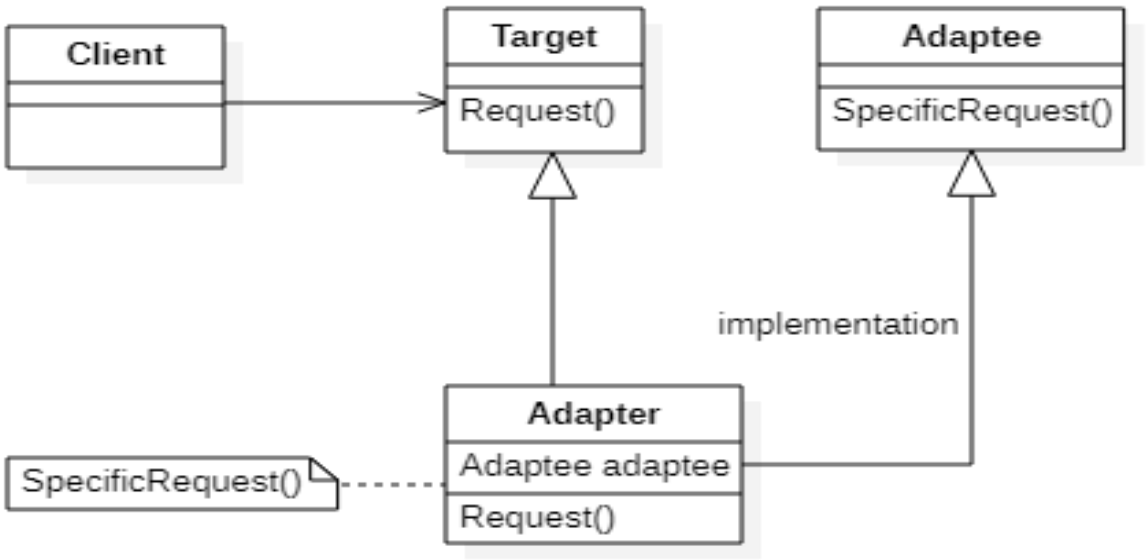
- Target: Một interface chứa các chức năng được sử dụng bởi Client.
- Client: Lớp sử dụng các đối tượng có interface Target.
- Adaptee: Định nghĩa interface không tương thích để tích hợp vào.
- Adapter: Lớp tích hợp, giúp tích hợp interface không tương thích với interface đang làm việc. Chuyển đổi interface cho Adaptee và kết nối Adaptee với Client.

Cấu trúc của Adapter Pattern có 2 kiểu dựa theo implement của chúng:

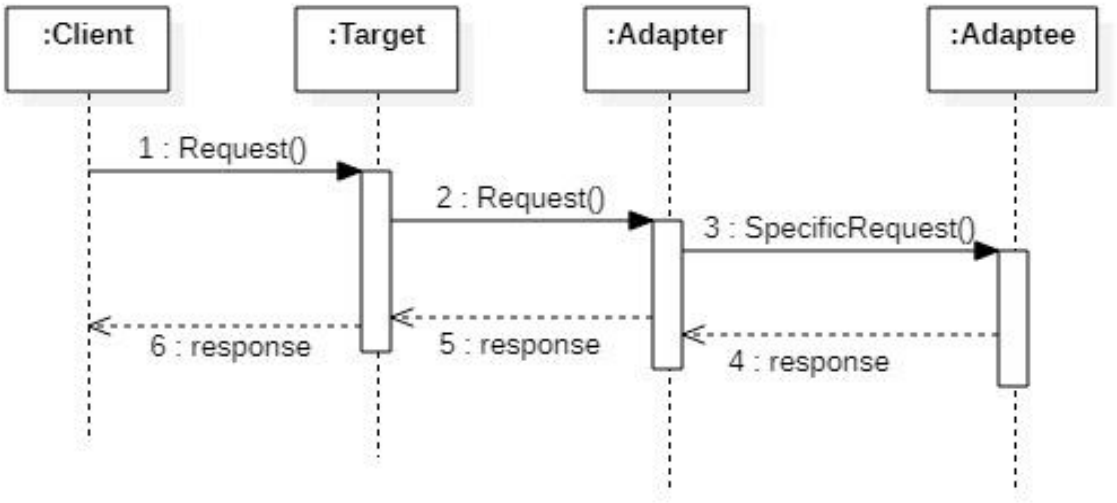
- Object Adapter – Compositon: Một lớp mới sẽ tham chiếu đến một (hoặc nhiều) đối tượng của lớp có sẵn với interface không tương thích (Adaptee), đồng thời cài đặt interface mà người dùng mong muốn (Target). Trong lớp mới này, khi cài đặt các phương thức của interface người dùng mong muốn, sẽ gọi phương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



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



Sequence diagram:



Code mẫu:

Target:

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

Adaptee:

```
class LegacyRectangle
{
public:
	LegacyRectangle(Coordinate x1, Coordinate y1, Coordinate x2, Coordinate
y2)
	{
		x1_ = x1;
		y1_ = y1;
		x2_ = x2;
		y2_ = y2;
		printf("LegacyRectangle: create: (%d,%d) => (%d,%d) \n", x1_,
y1_, x2_, y2_);
	}
	void oldDraw()
	{
		printf("LegacyRectangle: oldDraw: (%d,%d) => (%d,%d) \n", x1_,
y1_, x2_, y2_);
	}
private:
	Coordinate x1_;
	Coordinate y1_;
	Coordinate x2_;
	Coordinate y2_;
};
```

Adapter:

```
class RectangleAdapter : public Rectangle, private LegacyRectangle
{
public:
      RectangleAdapter(Coordinate x, Coordinate y, Dimension w, Dimension h)
            : LegacyRectangle(x, y, x + w, y + h)
      {
            printf("RectangleAdapter: create. (%d,%d) width = %d height = %d
\n", x, y, w, h);
      }
      virtual void draw()
      {
            printf("RectangleAdapter: draw \n");

            oldDraw();
      }
};
```

VI. Proxy

Proxy Pattern là Design Pattern thuộc nhóm Structural Patterns. Proxy Pattern cho phép một đối tượng sẽ đại diện cho một đối tượng khác. Tùy theo yêu cầu bài toán mà chúng ta áp dụng một cách linh hoạt, với mục đích:

- Kiểm soát quyền truy xuất các phương thức của đối tượng.
- Bổ sung thêm chức năng trước khi thực thi phương thức.
- Tạo ra đối tượng mới có chức năng nâng cao hơn đối tượng ban đầu.
- Giảm chi phí khi có nhiều truy cập vào đối tượng có chi phí khởi tạo ban đầu lớn.

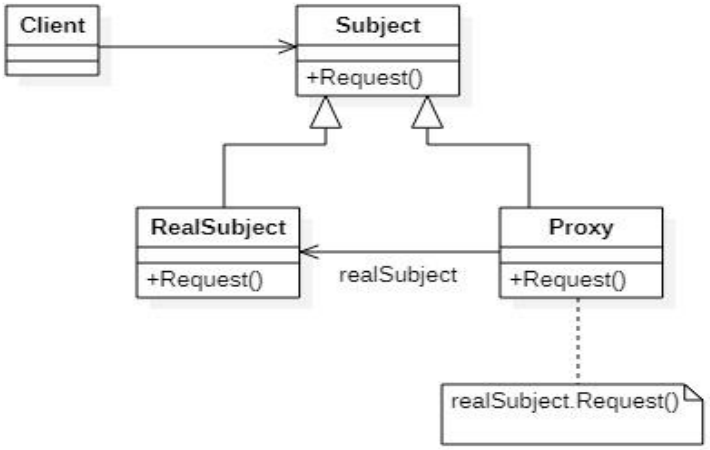
Sử dụng Proxy Pattern khi:

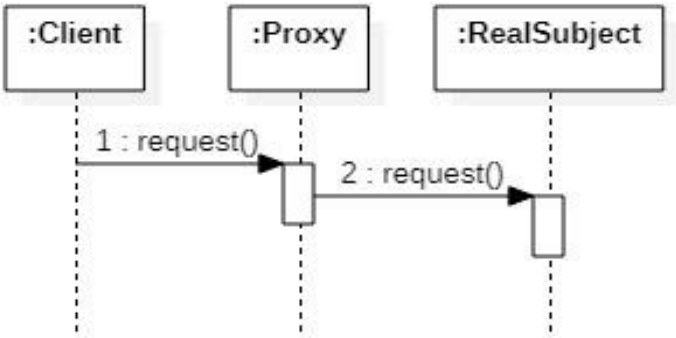
- Remote Proxy: Trường hợp bạn muốn thực thi một phương thức của đối tượng khác vùng địa chỉ hoặc ở máy tính khác.
- Virtual proxy: Khi có nhiều lượt truy xuất vào một đối tượng có cấu trúc phức tạp, chứa dữ liệu lớn chúng ta sẽ tạo một đối tượng đại diện cho nó ở lần truy xuất đầu tiên và tái sử dụng ở những lần tiếp theo.
- Protection proxy: Muốn kiểm tra một yêu cầu có quyền truy cập vào một nội dung nào đó hay không.
- Smart reference proxy: Sự thay thế cho một con trỏ rỗng cho phép thực hiện các chức năng thêm vào khi một đối tương được truy nhập.

Thành phần cơ bản của Proxy Pattern:

- Subject : Đối tượng này xác định giao diện chung cho RealSubject và Proxy để Proxy có thể được sử dụng bất cứ nơi nào mà RealSubject mong đợi.
- Proxy : Nó duy trì một tham chiếu đến RealSubject để Proxy có thể truy cập nó. Nó cũng thực hiện các giao diện 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.
- RealSubject : Đây là đối tượng chính mà proxy đại diện.

Mô hình:





Code mẫu:

```
class Image
{
public:
	virtual void displayImage() = 0;
};
```

RealSubject:

```
class CRealImage : public Image
{
private:
	const char* m_path;
public:
	CRealImage(const char* path)
	{
		m_path = path;
		loadImage();
	}
	~CRealImage()
	{
		delete m_path;
	}
	void displayImage()
	{
		printf("Display image! %s\n", m_path);
	}
	void loadImage()
	{
		printf("Load image! %s\n", m_path);
	}

};
```

Proxy:

```
class CImageVirtualProxy : public Image
{
private:
	CRealImage* m_realImage;
	const char* m_path;

public:
	CImageVirtualProxy(const char* path)
	{
		m_path = path;
		m_realImage = NULL;
	}

	~CImageVirtualProxy()
	{
		delete m_path, m_realImage;
	};

	void displayImage()
	{
		if (m_realImage == NULL)
			m_realImage = new CRealImage(m_path);
		m_realImage->displayImage();
	}
};
```

VII. Composite

Composite Pattern là Design Pattern thuộc nhóm Structural Patterns. Một đối tượng Composite được tạo thành từ một hay nhiều đối tượng tương tự như nhau. Ý tưởng là thao tác trên một nhóm đối tượng như trên một đối tượng duy nhất. Các đối tượng của nhóm phải có các thao tác chung.

Composite là mẫu thiết kế để tạo ra các đối tượng trong các cấu trúc cây để biểu diễn hệ thống phân lớp: bộ phận – toàn bộ. Cho phép các client tác động đến từng đối tượng và các thành phần của đối tượng một các thống nhất.

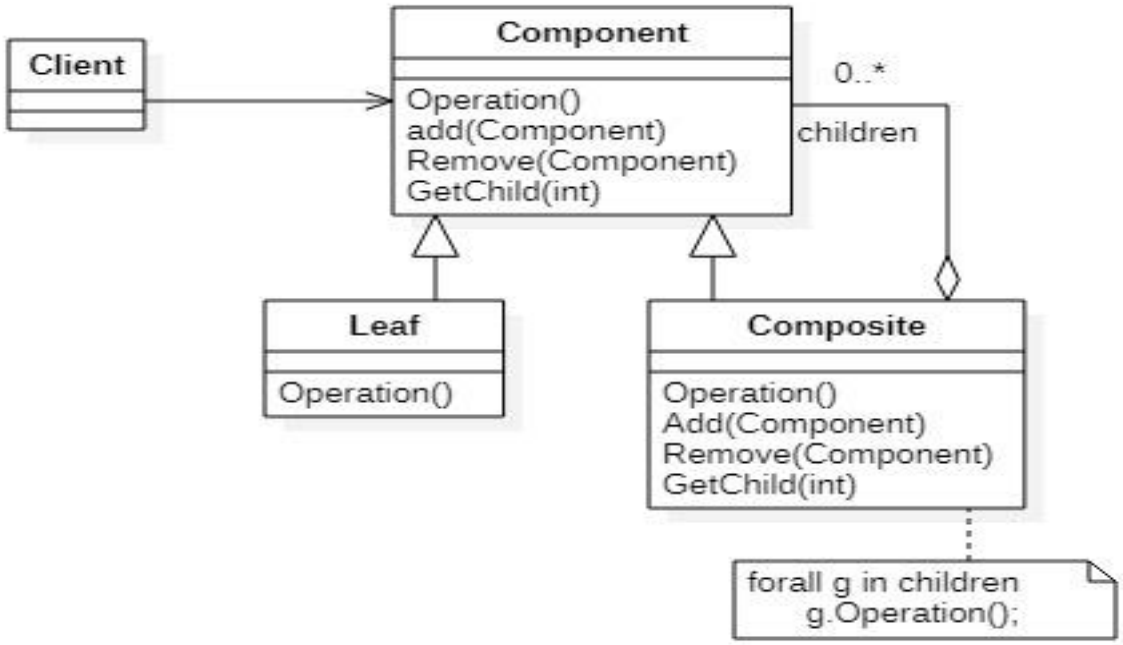
Sử dụng Composite khi:

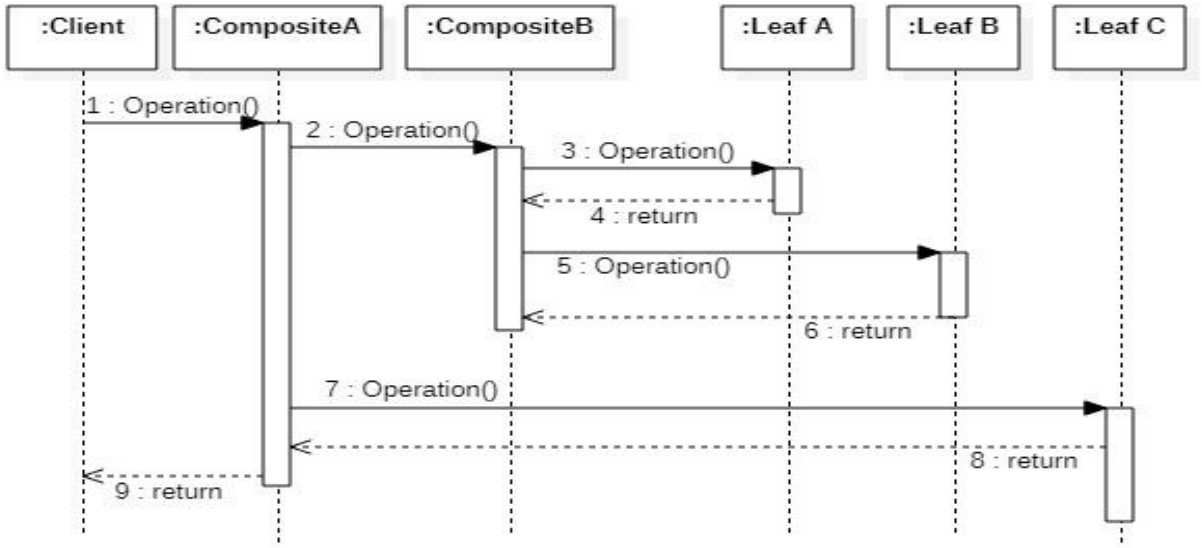
- Bạn muốn đại diện cho toàn bộ hệ thống phân cấp của các đối tượng.
- Bạn muốn clients có thể bỏ qua sự khác biệt giữa các thành phần của các đối tượng và các đối tượng riêng lẻ. Client sẽ xử lí thống nhất với tất cả các đối tượng trong cấu trúc hỗn hợp.

Thành phần cơ bản:

- Component: Khai báo giao diện cho thành phần đối tượng, thực thi thao tác mặc định, khai báo một giao diện cho phép truy cập đến các thành phần con.
- Leaf: biểu diễn các đối tượng lá trong thành phần đối tượng.
- Composite: Định nghĩa một thao tác cho các thành phần có thành phần con, lưu trữ thành phần con, thực thi quản lý các thành phần con của component.

Mô hình:





Code mẫu:

Component:

```
class PageObject {
public:
	virtual void Add(PageObject a)
	{
	}
	virtual void Remove()
	{
	}
	virtual void Delete(PageObject a)
	{
	}
};
```

Composite:

```
class Page : public PageObject {
public:
	void Add(PageObject a)
	{
		cout << "something is added to the page" << endl;
	}
	void Remove()
	{
		cout << "soemthing is removed from the page" << endl;
	}
	void Delete(PageObject a)
	{
		cout << "soemthing is deleted from page " << endl;
	}
};
```

Leaf:

```
class Copy : public PageObject {
	vector<PageObject> copyPages;
public:
	void AddElement(PageObject a)
	{
		copyPages.push_back(a);
	}
	void Add(PageObject a)
	{
		cout << "something is added to the copy" << endl;
	}
	void Remove()
	{
		cout << "something is removed from the copy" << endl;
	}
	void Delete(PageObject a)
	{
		cout << "something is deleted from the copy";
	}
};
```

VIII. State

State pattern là Design Pattern thuộc nhóm Behavioral Patterns cho phép một đối tượng thay đổi hành vi của nó khi trạng thái bên trong của nó thay đổi. Đối tượng sẽ xuất hiện để thay đổi lớp của nó. Điều này giúp giảm bớt rắc rối trong quá trình xử lý sự kiện và dễ dàng chỉnh sửa khi lập trình.

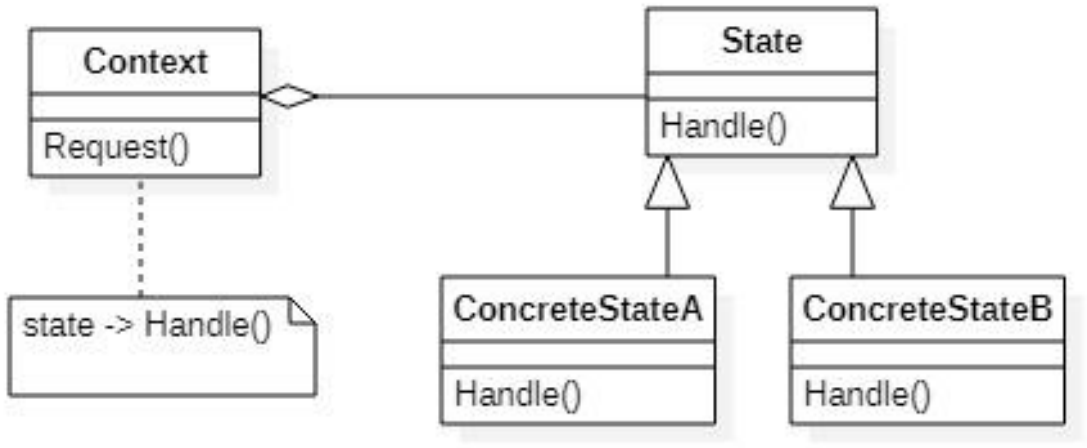
Sử dụng State khi:

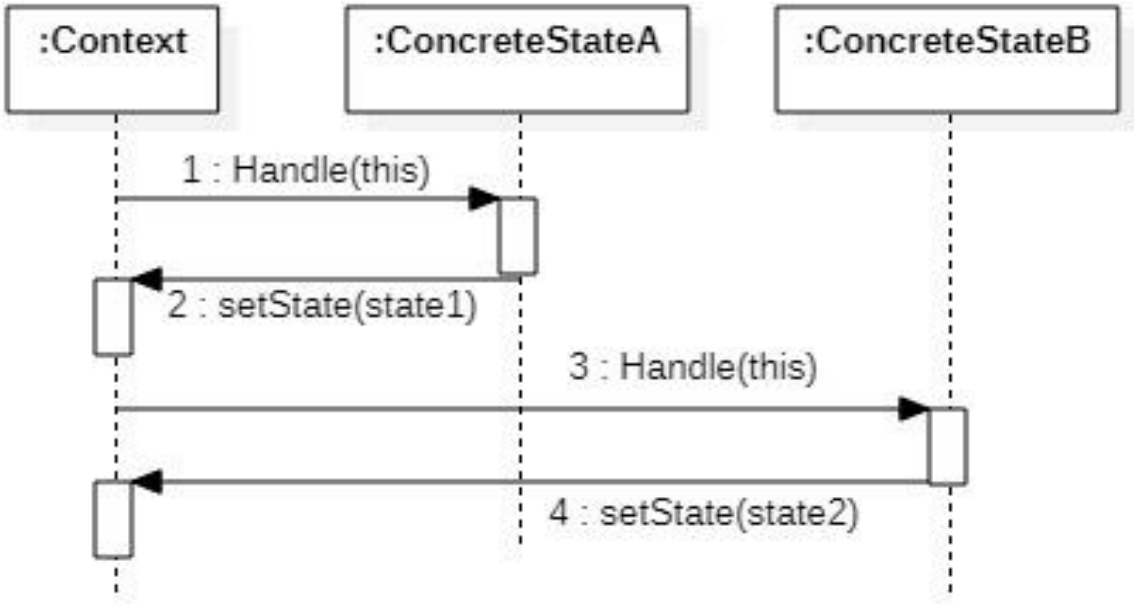
- Hành vi của một đối tượng phụ thuộc vào trạng thái của nó và nó phải thay đổi hành vi của nó trong thời gian chạy tùy thuộc vào trạng thái đó.
- Các hoạt động có các câu lệnh điều kiện lớn, nhiều phần phụ thuộc vào trạng thái của đối tượng. Trạng thái này thường được đại diện bởi một hoặc nhiều hằng số liệt kê. Thông thường, một số hoạt động sẽ chứa cấu trúc có điều kiện tương tự. State pattern đặt mỗi nhánh của điều kiện trong một lớp riêng biệt. Điều này cho phép bạn coi trạng thái của đối tượng là một đối tượng theo cách riêng của nó có thể thay đổi độc lập với các đối tượng khác.

Thành phần:

- Content: Xác định giao diện của client, duy trì thể hiện của lớp con ConcreteState xác định trạng thái hiện tại.
- State: Xác định một giao diện đóng gói các hành vi liên quan đến trạng thái ngữ cảnh cụ thể.
- ConcreteState: Lớp con thực hiện hành vi liên quan đến trạng thái ngữ cảnh.

Mô hình:





Code mẫu:

State:

```
class StateBase
{
public:
	virtual StateBase* GetNextState() = 0;
	virtual    char* ToString() = 0;
};
```

State Morning:

```
class Morning : public StateBase
{
public:
	virtual StateBase* GetNextState();
	virtual    char* ToString() {
	return "Class Morning";
	}
};

StateBase* Morning::GetNextState() {
	return new Evening();
	return new Night();
}
```

State Evening:

```
class Evening : public StateBase
{
public:
	virtual StateBase* GetNextState();
	virtual    char* ToString() {
		return "Class Evening";
	}
};

StateBase* Evening::GetNextState() {
	return new Night();
}
```

State night:

```
class Night : public StateBase
{
public:
	virtual StateBase* GetNextState();
	virtual    char* ToString() {
		return "Class Night";
	}
};

StateBase* Night::GetNextState() {
	return new Morning();
}
```

Sun:

```
class CSun
{
public:
	CSun() {}
	CSun(StateBase* pContext) {
		m_pState = pContext;
	}
	~CSun() {
		delete m_pState;
	}
	// Handles the next state
	void StateChanged() {
		if (m_pState) {
			StateBase* pState = m_pState->GetNextState();
			delete m_pState;
			m_pState = pState;
		}
	}
	char* GetStateName() {
		return m_pState->ToString();
	}
protected:
	StateBase* m_pState;
};
```

IX. Command

Command là Design Patterns thuộc nhóm Behavioral Patterns trong đó một đối tượng được sử dụng để đóng gói tất cả thông tin cần thiết để thực hiện một hành động hoặc kích hoạt một sự kiện sau đó. Thông tin này bao gồm tên phương thức, đối tượng sở hữu phương thức và giá trị cho các tham số phương thức.

Trong command pattern, request từ phía client sẽ được đóng gói dưới dạng một đối tượng command và đối tượng command này sẽ được chuyển cho một thành phần riêng biệt để thực thi (gửi trực tiếp hoặc đẩy vào một queue để lưu vết). Command pattern sẽ phân tách quá trình tiếp nhận request với quá trình thực thi request. Chính vì vậy, công việc xử lý request sẽ trở nên linh hoạt và dễ nâng cấp hơn.

Bên cạnh đó, trong Command pattern, chúng ta có thể lưu vết các command đã thực hiện. Nhờ điều này mà Command pattern còn được áp dụng cho tác vụ undo/redo trong các ứng dụng thực tế.

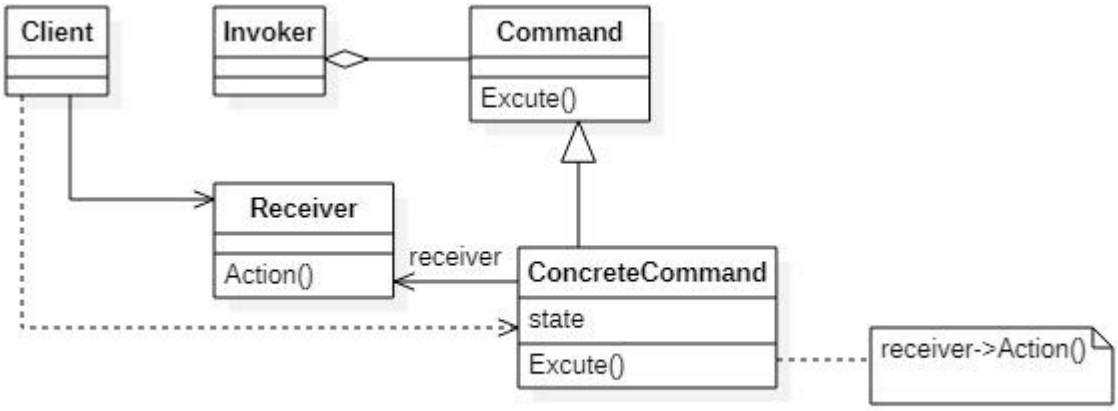
Sử dụng Command khi:

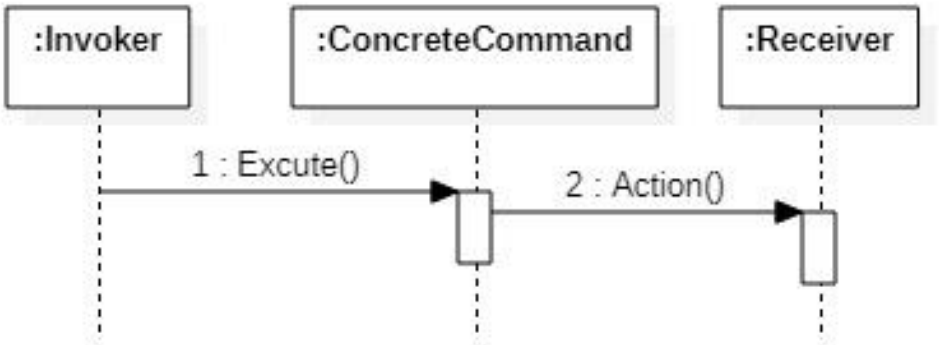
- Tham số hóa các đối tượng bằng một hành động để thực hiện. Bạn có thể diễn tả tham số hóa đó bằng ngôn ngữ thủ tục với chức năng gọi lại, nghĩa là một chức năng đã được đăng ký ở đâu đó để được gọi ở điểm sau. Các lệnh là một sự thay thế hướng đối tượng cho các cuộc gọi lại.
- Chỉ định, đưa vào hàng đợi và thực hiện các yêu cầu tại các thời điểm khác nhau. Một đối tượng Command có thể có thời gian độc lập với yêu cầu ban đầu. Nếu người nhận yêu cầu có thể được biểu diễn theo cách độc lập với không gian địa chỉ, thì bạn có thể chuyển một đối tượng lệnh cho yêu cầu sang một quy trình khác và thực hiện yêu cầu ở đó.
- Hỗ trợ Undo. Hoạt động Excute của Command có thể lưu trữ trạng thái để đảo ngược các hiệu ứng của nó trong chính lệnh đó. Giao diện Lệnh phải có thao tác Unexecute được thêm vào để đảo ngược các hiệu ứng của cuộc gọi trước đó đến Excute.

Thành phần:

- Command: Đây có thể là một interface hoặc abstract class, chứa một abstract method execute(). Request sẽ được đóng gói dưới dạng Command.
- ConcreteCommand: Là các implementation của Command, định nghĩa một ràng buộc giữa đối tượng Receiver và một hành động. Thực thi bằng cách gọi execute () trên Receiver.
- Client: Tạo ConcreteCommand, tiếp nhận request từ phía người dùng và đóng gói request thành ConcreteCommand thích hợp.
- Invoker: Hỏi Command để thự hiện yêu cầu.
- Receiver: Biết cách thực hiện các hoạt động liên quan đến một Receiver. Bất kỳ lớp nào cũng được phục vụ như một Receiver.

Mô hình:





Code mẫu:

Command:

```
class ICommand {
public:
	virtual ~ICommand() = default;
	virtual void execute() = 0;
};
```

Invoker:

```
class Switch {
private:
	std::queue<ICommand*> commands_;
public:
	void storeAndExecute(ICommand *command) {
		if (command) {
			commands_.push(command);
			command->execute();
			commands_.pop();
		}
	}
};
```

Receiver:

```
class Light {
public:
	void turnOn() {
		std::cout << "The light is on." << std::endl;
	}
	void turnOff() {
		std::cout << "The light is off." << std::endl;
	}
};
```

ConcreteCommand1:

```
class SwitchOnCommand : public ICommand {
private:
	Light *light_;
public:
	SwitchOnCommand(Light *light) :
		light_(light) {
	}
	void execute() {
		light_->turnOn();
	}
};
```

ConcreteCommand2:

```
class SwitchOffCommand : public ICommand {
private:
      Light *light_;
public:
      SwitchOffCommand(Light *light) :
            light_(light) {
      }
      void execute() {
            light_->turnOff();
      }
};
```

X. Observer

Obsever pattern là Design Pattern thuộc nhóm Behavioral Patterns dùng để tạo ra mối quan hệ phụ thuộc one-to-many giữa các đối tượng, khi một đối tượng thay đổi trạng thái, tất cả các phụ thuộc của nó sẽ được thông báo và cập nhật tự động.

Trong Observer Pattern, chúng ta sẽ cần phải define ra interface cho subject và observer, chúng sẽ làm việc với nhau thông quan interface mà không cần biết class cụ thể là gì. Điều này làm hệ thống dễ duy trì và mở rộng.

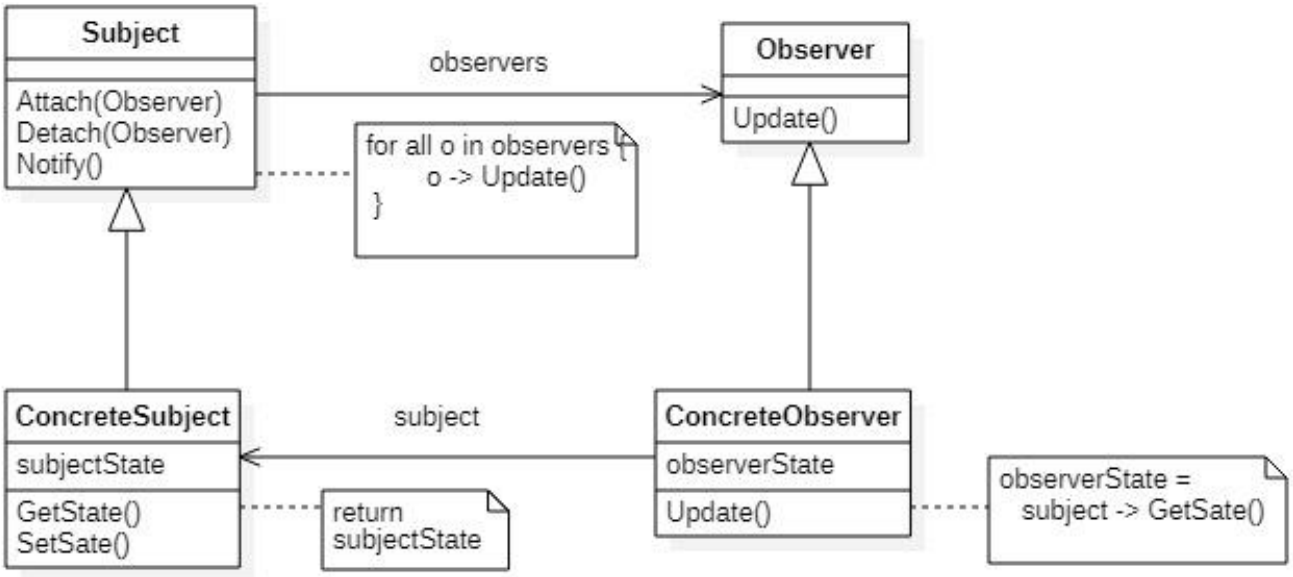
Sử dụng Observer khi:

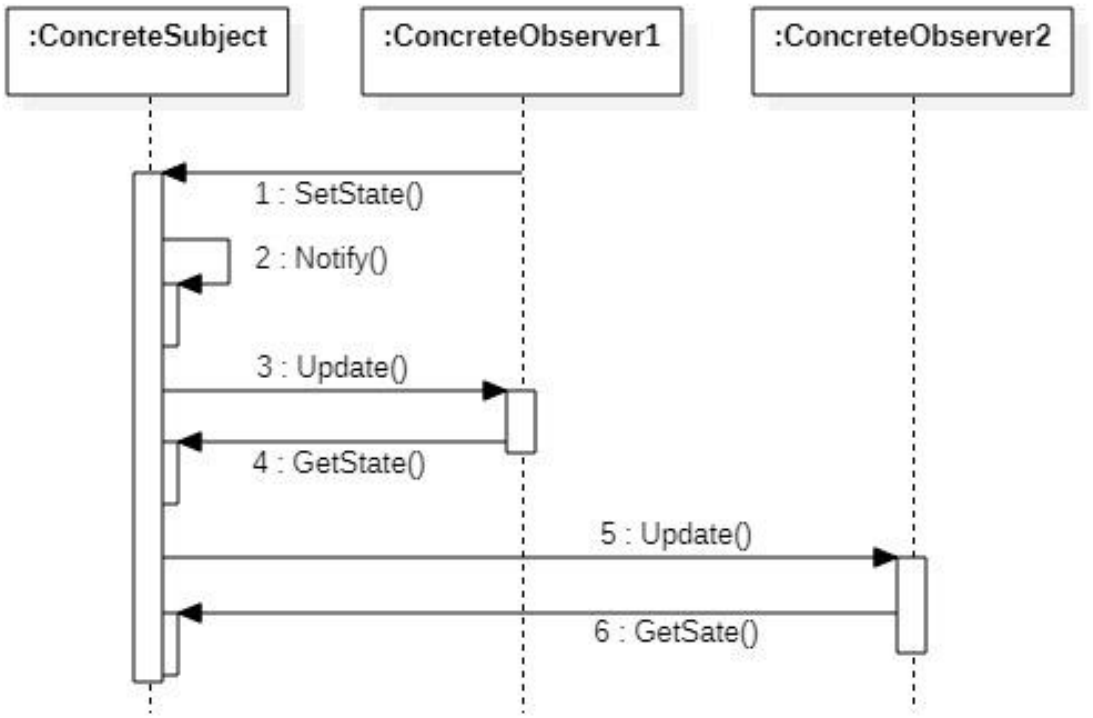
- Một sự trừu tượng có hai khía cạnh, một khía cạnh phụ thuộc vào khía cạnh khác. Đóng gói các khía cạnh này trong các đối tượng riêng biệt cho phép bạn thay đổi và tái sử dụng chúng một cách độc lập.
- Một đối tượng yêu cầu thay đổi đối tượng khác và bạn không biết có bao nhiêu đối tượng cần thay đổi.
- Một đối tượng có thể thông báo cho các đối tượng khác mà không cần đưa ra giả định về những đối tượng này là ai. Nói cách khác, bạn không muốn các đối tượng này được liên kết chặt chẽ

Thành phần:

- Subject: Là đối tượng mà Observer cần nhận thông tin
- Observer: Định nghĩa interface để cập nhật và thông báo về sự thay đổi trạng thái của đối tượng.
- ConcreteSubject: Lưu trữ trạng thái liên quan đến các đối tượng ConcreteObserver. Gửi thông báo khi trạng thái của nó thay đổi.
- ConcreteObserver: Duy trì một tham chiếu đến một đối tượng ConcreteSubject. Lưu trữ trạng thái phải phù hợp với Subject. Thực hiện giao diện cập nhật Observer để giữ trạng thái phù hợp với Subject.

Mô hình:


Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
Observer
Update()
for all o in observers {
o -> Update()
}
ConcreteSubject
subjectState
GetState()
SetSate()
subject
ConcreteObserver
observerState
Update()
return
subjectState
observerState =
subject -> GetSate()


:ConcreteSubject
:ConcreteObserver1
:ConcreteObserver2
1 : SetState()
2 : Notify()
3 : Update()
4 : GetState()
5 : Update()
6 : GetSate()

Code mẫu:

Observer:

```
class Observer
{
public:
	virtual void update(int value) = 0;
};
```

Subject:

```
class Subject
{
	int m_value;
	std::vector<Observer *> m_views;
public:
	void attach(Observer *obs){
		m_views.push_back(obs);
	}
	void set_val(int value){
		m_value = value;
		notify();
	}
	void notify(){
		for (int i = 0; i < m_views.size(); ++i)
			m_views[i]->update(m_value);
	}
};
```

ConcreteObserver1:

```
	class DivObserver : public Observer
	{
		int m_div;
	public:
		DivObserver(Subject *model, int div){
			model->attach(this);
			m_div = div;
		}
		void update(int v){
			printf("%d div %d is %d \n",v,m_div, v / m_div);
		}
	 };
```

ConcreteObserver2:

```
	class ModObserver : public Observer
	{
		int m_mod;
	public:
		ModObserver(Subject *model, int mod){
			model->attach(this);
			m_mod = mod;
		}
		void update(int v){
			printf("%d mod %d is %d \n", v, m_mod, v % m_mod);
		}
	};
```

Tài liệu tham khảo:

- Sách “Design Patterns Elements of Reusable Object-Oriented Software” của các tác giả Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.
-