TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

MẪU THIẾT KẾ HƯỚNG ĐỐI TƯỢNG

DESIGN PATTERN

1612891 Phan Quốc Thắng - Lớp 16CNTN

1612829 Nguyễn Quốc Vương - Lớp 16CNTN

1612842 Lê Thành Công - Lớp 16CNTN

1612774 Nguyễn Thanh Tuấn - Lớp 16CNTN

2017 - 2018

PHƯƠNG PHÁP LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

GVBM: NGUYỄN MINH HUY

MỤC LỤC

CHƯƠNG I:

GIỚI THIỆU VÀ TỔNG QUAN

1.1. Lịch sử design pattern

Ý tưởng dùng mẫu thiết kế xuất phát từ ngành kiến trúc, Alexander, Ishikawa, Silverstein, Jacobson, Fiksdah-King và Angel (1977) lần đầu tiên đưa ra ý tưởng dùng các mẫu chuẩn trong thiết kế xây dựng và truyền thông. Họ đã xác định và lập sưu liệu các mẫu có liên quan để có thể dùng để giải quyết các vấn đề thường xảy ra trong thiết kế các cao ốc. Mỗi mẫu này là một cách thiết kế, chúng đã được phát triển hàng trăm năm như là các giải pháp cho các vấn đề mà người ta thường làm trong lĩnh vực xây dựng thường gặp. Các giải pháp tốt nhất có được ngày hôm nay là qua một quá trình sàng lọc tự nhiên. Mặc dù ngành công nghệ phần mềm không có lịch sử phát triển lâu dài như ngành kiến trúc, xây dựng nhưng Công nghệ phần mềm là một ngành công nghiệp tiên tiến, tiếp thu tất cả những gì tốt đẹp nhất từ các ngành khác. Mẫu được xem là giải pháp tốt để giải quyết vấn đề xây dựng hệ thống phần mềm. Suốt những năm đầu 1990, thiết kế mẫu được thảo luận ở các hội thảo workshop, sau đó người ta nổ lực để đưa ra danh sách các mẫu và lập sưu liệu về chúng. Những người tham gia bị dồn vào việc cần thiết phải cung cấp một số kiểu cấu trúc ở một mức quan niệm cao hơn đối tượng và lớp để cấu trúc này có thể được dùng để tổ chức các lớp. Đây là kết quả của sự nhận thức được những cải tiến đáng kể đối với chất lượng cũng như hiệu quả của công việc phát triển phần mềm. Mẫu được xem là cách tổ chức việc phát triển hướng đối tượng, cách đóng gói các kinh nghiệm của những người đi trước rất hiệu quả trong thực hành.

Năm 1994 tại hội nghị PloP (Pattern Language of Progamming Design) đã được tổ chức. Cũng trong năm này quyển sách Design pattern: Elements of Reusable Object Oriented Software (Gamma, Johnson, Helm và Vhissdes, 1995) đã được xuất bản đúng vào thời điểm diễn ra hội nghị OOPSLA’94. Đây là một tài liệu còn phôi thai trong việc làm

nổi bật ảnh hưởng của mẫu đối với việc phát triển phần mềm, sự đóng góp của nó là xây dựng các mẫu thành các danh mục (catalogue) với định dạng chuẩn được dùng làm tài liệu cho mỗi mẫu và nổi tiếng với tên Gang of Four (bộ tứ), và các mẫu nó thường được gọi là các mẫu Gang of Four. Còn rất nhiều các cuốn sách khác xuất hiện trong 2 năm sau, và các định dạng chuẩn khác được đưa ra.

Năm 2000 Evitts có tổng kết về cách các mẫu xâm nhập vào thế giới phần mềm (sách của ông lúc bấy giờ chỉ nói về những mẫu có thể được sử dụng trong UML chứ chưa đưa ra khái niệm những mẫu thiết kế một cách tổng quan). Ông công nhận Kent Beck và Ward Cunningham là những người phát triển những mẫu đầu tiên với SmallTalk trong công việc của họ được báo cáo tại hội nghị OOPSLA'87. Có 5 mẫu mà Kent Beck và Ward Cunningham đã tìm ra trong việc kết hợp các người dùng của một hệ thống mà họ đang thiết kế. Năm mẫu này đều được áp dụng để thiết kế giao diện người dùng trong môi trường Windows.

1.2. Khái niệm design pattern

Design pattern là tập các giải pháp cho vấn đề phổ biến trong thiết kế các hệ thống máy tính. Đây là tập các giải pháp đã công được công nhận là tài liệu có giá trị, những người phát triển có thể áp dụng giải pháp này để giải quyết các vấn đề tương tự. Giống như với các yêu cầu của thiết kế và phân tích hướng đối tượng (nhằm đạt được khả năng sử dụng các thành phần và thư viện lớp), việc sử dụng các mẫu cũng cần phải đạt được khả năng tái sử dụng các giải pháp chuẩn đối với vấn đề thường xuyên xảy ra. Christopher Alexander nói rằng: "Mỗi một mẫu mô tả một vấn đề xảy ra lặp đi lặp lại trong môi trường và mô tả cái cốt lõi của giải pháp để cho vấn đề đó. Bằng cách nào đó bạn đã dùng nó cả triệu lần mà không làm giống nhau 2 lần".

Design pattern không phải là một phần của UML cốt lõi,nhưng nó lại được sử dụng rộng rãi trong thiết kế hệ thống hướng đối tượng và UML cung cấp các cơ chế biểu diễn mẫu dưới dạng đồ họa.

CHƯƠNG II:

HỆ THỐNG CÁC MẪU DESIGN PATTERN

Hệ thống các mẫu design pattern hiện nay có 23 mẫu được định nghĩa trong cuốn "Design patterns Elements of Reusable Object Oriented Software". Hệ thống các mẫu này có thể nói là đủ và tối ưu cho việc giải quyết hết các vấn đề của bài toán phân tích thiết kế và xây dựng phần mềm trong thời điểm hiện tại. Hệ thống các mẫu design pattern được chia thành 3 nhóm: Creational, Structural, Behavioral.

2.1. Nhóm Creational

Gồm có 5 patterns: AbstractFactory, Abstract Method, Builder, Prototype, và Singleton. Nhóm này liên quan tới việc tạo ra các thể nghiệm (instance) của đối tượng, tách biệt với cách được thực hiện từ ứng dụng. Muốn xem xét thông tin cảu các mẫu trong nhóm này hì phải dựa vào biểu đồ nào phụ thuộc vào chính mẫu đó, mẫu thiên về hành vi hay cấu trúc.

2.2. Nhóm Structural

Gồm có 7 patterns: Adapter, Bridge, Composite, Decorator, Facade, Proxy, và Flyweight. Nhóm này liên quan tới các quan hệ cấu trúc giữa các thể nghiệm, dùng kế thừa, kết tập, tương tác. Để xem thông tin về mẫu này phải dựa vào biểu đồ lớp của mẫu.

2.3. Nhóm Behavioral

Gồm có 11 patterns: Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, và Visitor. Nhóm này liên quan đến các quan hệ gán trách nhiệm để cung cấp các chức năng giữa các đối tượng trong hệ thống. Đối với các mẫu thuộc nhóm này ta có thể dựa vào biểu đồ cộng tác và

biểu đồ diễn tiến. Biểu đồ cộng tác và biểu đồ diễn tiến sẽ giải thích cho ta cách chuyển giao của các chức năng.

2.4. Các mẫu thiết kế thông dụng

Trong đồ án này chúng em xin trình bày về tìm hiểu bài toán, giải pháp, cài đặt và ưu khuyết điểm của một mẫu thiết kế Gang of Four. Bao gồm: Singleton, Flyweight, Decorator, Factory Method.

CHƯƠNG III:

PHÂN TÍCH NỘI DUNG CÁC MẪU DESIGN PATTERN

3.1. Nhóm Creational

3.1.1. Factory method

3.1.1.1. Định nghĩa

Factory method, tên đầy đủ là Factory method pattern, là thiết kế mẫu hướng đối tượng trong việc thiết kế phần mềm cho máy tính, nhằm giải quyết vấn đề tạo một đối tượng mà không cần thiết chỉ ra một cách chính xác lớp 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 lớp con thừa kế có thể override để chỉ rõ đối tượng nào sẽ được tạo. Về cơ bản, "factory method" thường được áp dụng cho những phương thức mà nhiệm vụ chính của nó là tạo ra đối tượng.

Bản chất của mẫu thiết kế Factory là "Định nghĩa một giao diện (interface) cho việc tạo một đối tượng, nhưng để các lớp con quyết định lớp nào sẽ được tạo. "Factory method" giao việc khởi tạo một đối tượng cụ thể cho lớp con.

Áp dụng

"Factory method" thường được dùng trong bộ phát triển (toolkit) hay framework, ở đó, đoạn mã của framework cần thiết phải tạo một đối tượng là những lớp con của ứng dụng sử dụng framework đó.

Tại sao lại phải dùng Factory pattern thay cho việc khởi tạo đối tượng thông thường? Dùng Factory pattern có lợi ích gì?

Factory pattern đưa ra 1 ý tưởng mới cho việc khởi tạo các instance phù hợp với mỗi request từ phía Client. Sử dụng Factory pattern sẽ có những ưu điểm sau:

- Tạo ra 1 cách mới trong việc khởi tạo các Object thông qua 1 interface chung.
- Khởi tạo các Objects mà che giấu đi xử lí logic của việc khởi tạo đấy.
- Giảm sự phụ thuộc giữa các module, các logic với các class cụ thể, mà chỉ phụ thuộc vào interface hoặc abstract class.

Vậy thì khi nào chúng ta nên sử dụng Factory pattern?

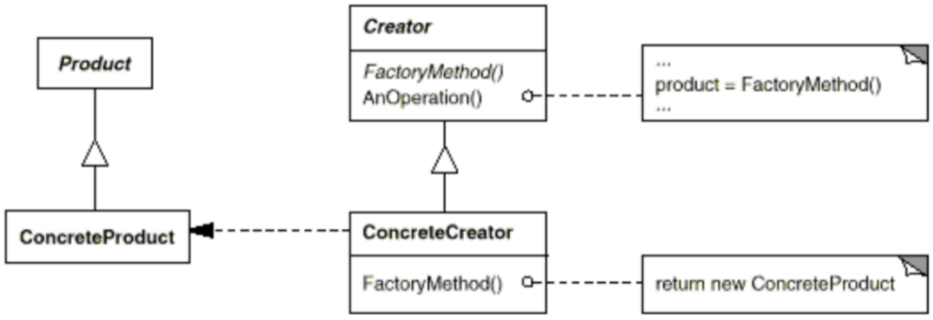
Dựa vào lợi ích của việc sử dụng Factory pattern mà ta sẽ dùng chúng với một số mục đích sau:

- Tạo ra 1 cách mới trong việc khởi tạo Object.
- Che giấu xử lý logic của việc khởi tạo trong trường hợp bạn đang muốn viết 1 thư viện để người khác sử dụng.
- Giảm sự phụ thuộc vì dễ dàng cho việc mở rộng trong trường hợp bạn chưa biết chắc số lượng đối tượng là đã đủ cho bài toán của mình chưa.

Tóm lược

Factory method gói gọn lại việc tạo đối tượng. Điều này hữu dụng nếu quá trình tạo phức tạp. Ví dụ như nó phụ thuộc vào những điều chỉnh trong tập tin cấu hình hay phụ thuộc vào thông tin của người dùng nhập vào.

3.1.1.2. Sơ đồ UML



Product (Page): định nghĩa interface của đối tượng mà Factory Product tạo ra.

Concrete Product (SkillsPage, EducationPage, ExperiencePage): cài đặt giao diện Product. Lớp thể hiện đối tượng sản phẩm cần tạo, hiện thực của interface Product.

Creator (Document): khai báo Factory Method mà trả về một đối tượng của kiểu Product. Sự kiến tạo này cũng có thể định nghĩa một cài đặt mặc định của Factory Method trả về một đối tượng ConcreteProduct mặc định. Có thể gọi FactoryMethod để tạo ra một đối tượng Product.

ConcreteCreator (Report, Resume): Thừa kế lớp Creator, override Factory Method để trả về một đối tượng của lớp ConcreteProduct tương ứng.

3.1.1.3. Sample code

1. Không dùng Factory Method

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

enum VehicleType {
   VT_TwoWheeler, VT_FourWheeler
```

```
};

// Library classes
class Vehicle {
public:
   virtual void printVehicle() = 0;
};
class TwoWheeler : public Vehicle {
public:
   void printVehicle()  {
         cout << "I am two wheeler" << endl;
   }
};
class FourWheeler : public Vehicle {
public:
   void printVehicle()  {
         cout << "I am four wheeler" << endl;
   }
};

int main() {
   Vehicle*p = NULL;
   int type = VT_TwoWheeler;
   if (type == VT_TwoWheeler){
         p = new TwoWheeler;
   }
   p->printVehicle();
   delete p;
   type = VT_FourWheeler;
   if (type == VT_FourWheeler){
         p = new FourWheeler;
   }
   p->printVehicle();
   return 0;
}
```

Các vấn đề với thiết kế trên là gì?

Như bạn đã thấy trong ví dụ trên, Client tạo các đối tượng của TwoWheeler hoặc FourWheeler dựa trên một số đầu vào trong khi xây dựng đối tượng của nó.

Như vậy, nếu ta thêm một lớp mới ThreeWheeler để biểu diễn xe ba bánh. Chuyện gì sẽ xảy ra? Client sẽ thay code mới theo điều kiện nhất định để tạo đối tượng ThreeWheeler.

Vì vậy, mỗi lần thay đổi mới được thực hiện ở phía thư viện, Client sẽ cần phải thực hiện một số thay đổi tương ứng ở hàm main và biên dịch lại mã.

2. Dùng Factory Method

```
// C++ program to demonstrate factory method design pattern
#include <iostream>
using namespace std;

enum VehicleType {
   VT_TwoWheeler, VT_ThreeWheeler, VT_FourWheeler
};

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

class TwoWheeler : public Vehicle {
public:
   void printVehicle() {
         cout << "I am two wheeler" << endl;
   }
};
class ThreeWheeler : public Vehicle {
public:
   void printVehicle() {
         cout << "I am three wheeler" << endl;
   }
};
class FourWheeler : public Vehicle {
public:
   void printVehicle() {
         cout << "I am four wheeler" << endl;
```

```
    }
};

// Factory method to create objects of different types.
// Change is required only in this function to create a new object type
Vehicle* Vehicle::Create(VehicleType type) {
    if (type == VT_TwoWheeler)
        return new TwoWheeler();
    else if (type == VT_ThreeWheeler)
        return new ThreeWheeler();
    else if (type == VT_FourWheeler)
        return new FourWheeler();
    else return NULL;
}

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

    Vehicle* getVehicle() {
        return pVehicle;
    }

    Vehicle* getVehicle(VehicleType type) {
        return Vehicle::Create(type);
    }
};
```

```
int main() {
   Factory *f = new Factory();
   Vehicle * pVehicle = f->getVehicle();
   pVehicle->printVehicle();

   return 0;
}
```

Trong ví dụ trên, chúng ta đã tách hoàn toàn việc lựa chọn kiểu Vehicle cho việc tạo đối tượng từ Client trong hàm main. Thư viện bây giờ có trách nhiệm quyết định kiểu đối tượng nào sẽ được tạo dựa trên đầu vào. Client chỉ cần truyền kiểu Vehicle mình muốn vào hàm Create trong phần cài đặt Factory mà không cần thay đổi mã nguồn hàm main.

Ví dụ khác: Cần tạo kết nối dữ liệu tới 2 loại CSDL là SQLServer và MySQL, và chưa biết trước cần phải khởi tạo kết nối tới CSDL nào vì tùy thuộc yêu cầu khách hàng.

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

class Connection
{
public:
        virtual void connectionName() = 0;
};

class SQLServerConnection :public Connection
{
public:
        void connectionName(){
                cout << "I'm SQL Server Connection" << endl;
        }
};

class MySQLConnection : public Connection
{
public:
        void connectionName(){
```

```
                cout << "I'm MySQL Connection" << endl;
        }
};
//Abstract class Creator of Factory Method
class ConnectionCreator
{
public:
        virtual Connection* createConnection() = 0;
        void showConnectionName(){
                Connection* connection = createConnection();
                connection->connectionName();
        }
};

class createSQLServerConnection :public ConnectionCreator
{
public:
        Connection* createConnection(){
                return new SQLServerConnection();
        }
};

class createMySQLConnection : public ConnectionCreator
{
        Connection* createConnection(){
                return new MySQLConnection();
        }
};

void main()
{
        ConnectionCreator *f;
        f = new createMySQLConnection();
        f->showConnectionName();

        f = new createSQLServerConnection();
        f->showConnectionName();
        delete f;
}
```

3.1.1.4. Ưu điểm

- Tạo sự linh hoạt trong việc sử dụng lại code bằng cách loại bỏ việc tạo ra các lớp ứng dụng cụ thể, chỉ thao tác với interface Product và có thể làm việc với mọi ConcreteProduct hỗ trợ interface này.
- Client thao tác dựa trên interface mà không quan tâm sản phẩm gì được trả về, quá trình tạo sản phẩm được che dấu hoàn toàn.
- Dễ dàng cập nhật phần mềm.

3.1.1.5. Khuyết điểm

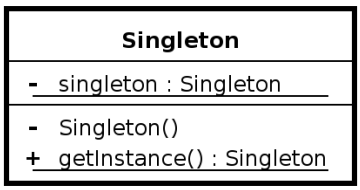
- Hạn chế có thể thấy ngay là một khi cần tạo một loại đối tượng mới, ta phải thừa kế lại lớp Creator và cài đặt phương thức khởi tạo.
- Khi thêm một sản phẩm mới, ta phải sửa đổi và bổ sung mã chương trình khá nhiều (thêm hàm sản xuất, lớp sản xuất…) điều này làm gia tăng độ phức tạp của hệ thống.

3.1.2. Singleton

3.1.2.1. Định nghĩa

Singleton là mẫu thiết kế nhằm đảm bảo chỉ có duy nhất một thể nghiệm và cung cấp điểm truy cập của nó một cách thống nhất toàn cục.

3.1.2.2. Sơ đồ UML



3.1.2.3. Sample code

```
class Singleton
{
private:
   static    Singleton*  m_Instance;
   Singleton();
   ~Singleton();
public:
   static void init();
   static void release();
   static Singleton* getInstance();

   void method();
};

Singleton*    Singleton::m_Instance = NULL;

Singleton::Singleton()
{
}

Singleton::~Singleton()
{
}

Singleton* Singleton::getInstance()
{
   if (m_Instance == NULL)
         Singleton::init();
   return m_Instance;
}

void    Singleton::init()
{
   if (m_Instance == NULL)
         m_Instance = new Singleton();
}

void    Singleton::release()
{
   if (m_Instance)
   {
         delete m_Instance;
         m_Instance = NULL;
   }
}

void    Singleton::method()
{
   cout<<"This is a singleton method."<<endl;
}

void main()
{
   Singleton::getInstance()->method();
}
```

3.1.2.4. Ưu nhược điểm

- Quản lý việc truy cập tốt hơn vì chỉ có một thể hiện đơn nhất
- Cho phép cải tiến lại các tác vụ (operations) và các thể hiện (representation) do pattern có thể được kế thừa và tùy biến lại thông qua một thể hiện của lớp con
- Quản lý số lượng thể hiện của một lớp, không nhất thiết chỉ có một thể hiện mà có số thể hiện xác định.
- Khả chuyển hơn so với việc dùng một lớp có thuộc tính là static, vì việc dùng lớp static chỉ có thể sử dụng một thể hiện duy nhất, còn Singleton Pattern cho phép quản lý các thể hiện tốt hơn và tùy biến theo điều kiện cụ thể.

3.2. Nhóm Structional

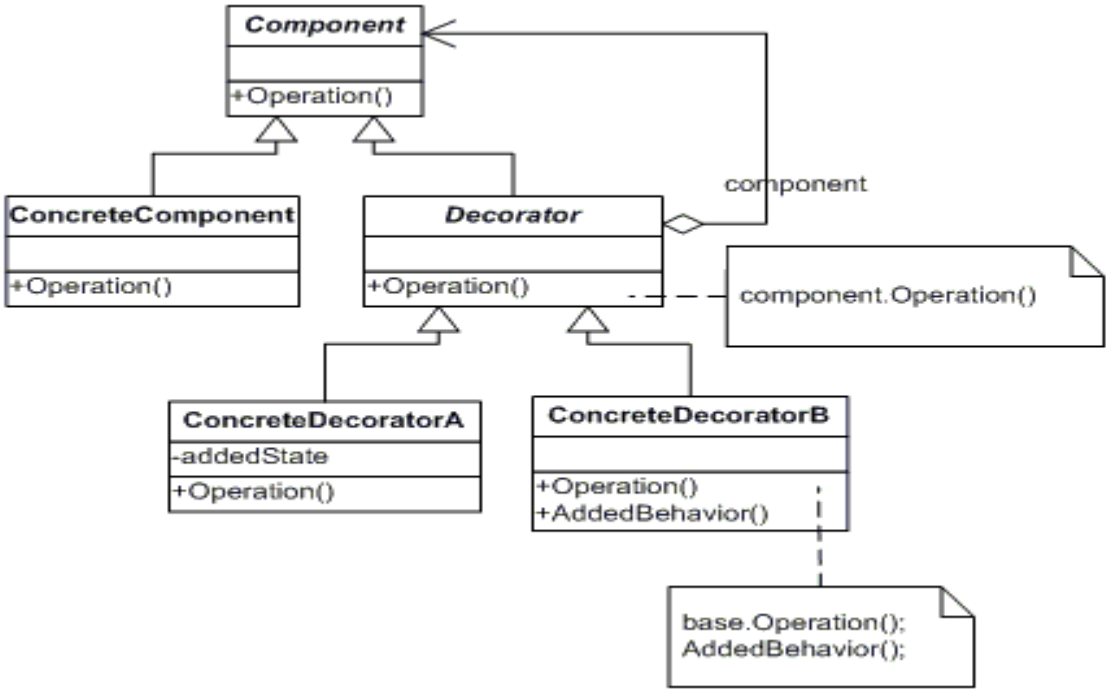
3.2.1. Decorator

3.2.1.1. Định nghĩa

Decorator thuộc nhóm cấu trúc:

- Decorator pattern được sử dụng để mở rộng chức năng của một đối tượng một cách linh hoạt mà không cần phải thay đổi mã nguồn của class gốc.
- Nó được thực hiện bằng cách tạo ra một đối tượng wrapper, được gọi là decorator (nhà trang trí). “Trang trí” xung quanh đối tượng ban đầu.

3.2.1.2. Sơ đồ UML



Component: là thành phần đại diện để chứa các hành vi chung của đối tượng. Nó có thể là abstract class hoặc interface.

ConcreteComponent: kế thừa từ component. Là đối tượng gốc mà các chức năng sẽ được bổ sung thêm.

Decorator: là một abstract class, chứa một quan hệ HAS-A tham chiếu tới Component và thực hiện các phương thức trong component.

ConcreteDecorator: kế thừa từ decorator và có thêm các chức năng mở rộng riêng của nó.

Trung tâm của sơ đồ UML là lớp Decorator. Nó bao gồm hai loại quan hệ với giao diện Component:

***Is-a:**

Quan hệ is-a thể hiện bởi một mũi tên tam giác Decorator đến Component, chỉ

ra rằng Decorator thực thi giao diện Component. Thực tế là Decorator thừa kế từ Component có nghĩa là đối tượng Decorator có thể được sử dụng bất cứ nơi nào các đối tượng Component được mong đợi. Lớp ConcreteComponent cũng là một mối quan hệ is-a với Component.

***Has-a:**

Quan hệ has-a thể hiện bởi hình thoi mở trên Decorator, liên kết với Component. Điều này cho thấy rằng các Decorator khởi tạo một hoặc nhiều đối tượng Component và các đối tượng được trang trí có thể sống lâu hơn bản gốc. Decorator sử dụng các thuộc tính thành phần (loại Component) để gọi bất kỳ hoạt động thay thế có thể được ghi đè lên. Đây là cách mẫu Decorator đạt được mục tiêu của mình.

3.2.1.3. Sample code

Cho đề bài: Một chiếc xe mặc định gồm 3 bộ phận: bánh xe, gầm xe và động cơ với giá lần lượt là: 4.0, 10.5, 35.0.
Dựa vào mẫu thiết kế Decorator, thiết kế các class sao cho người dùng có thể mở rộng chức năng của đối tượng (thêm các bộ phận như: GPS, thiết bị an toàn, Hifi, thiết bị tự động… với các mức giá cho trước). Từ đó, có thể đưa ra các mức giá mới tùy thuộc vào các bộ phận được gắn thêm.

```
//Component
class Car {
public:
	virtual float CalculateCost() = 0;
};


//Concrete Component
class DefaultCar : public Car {
private:
	float costOfWheels = 4;
	float costOfChassis = 10;
	float costOfEngine = 30;
public:
	float CalculateCost() {
		return costOfWheels + costOfChassis + costOfEngine;
	}
};
```

```
//Decorator
class Decorator : public Car {
protected:
	Car* car;
public:
	Decorator(Car* paramCar) {
		car = paramCar;
	}
	virtual float CalculateCost() = 0;
};

//Concrete Decorator
class GPSDecorator : public Decorator {
private:
	float costOfGpsDevice = 9.5;
public:
	GPSDecorator(Car* paramCar) : Decorator(paramCar) {}
	float CalculateCost() {
		return car->CalculateCost() + costOfGpsDevice;
	}
};

class SafeDecorator : public Decorator {
private:
	float costOfSafeSensors = 15;
public:
	SafeDecorator(Car* paramCar) : Decorator(paramCar) {}
	float CalculateCost() {
		return car->CalculateCost() + costOfSafeSensors;
	}
};

class HifiDecorator : public Decorator {
private:
	float costOfSpeaker = 8;
	float costOfAmpli = 14;
public:
	HifiDecorator(Car* paramCar) : Decorator(paramCar) {}
	float CalculateCost() {
		return car->CalculateCost() + costOfSpeaker + costOfAmpli;
	}
};


class AutoDecorator : public Decorator {
private:
	float costOfAutoProcessor = 20;
public:
	AutoDecorator(Car* paramCar) : Decorator(paramCar) {}
	float CalculateCost() {
		return car->CalculateCost() + costOfAutoProcessor;
	}
};
```

```
//Client
void main() {
      Car* defaultCar = new DefaultCar();
      Car* safeCar = new SafeDecorator(defaultCar);
      Car* GPSAndSafeCar = new GPSDecorator(safeCar);
      Car* HifiAndGPSAndSafeCar = new HifiDecorator(GPSAndSafeCar);
     Car* AutoAndHifiAndGPSAndSafeCar = new
     AutoDecorator(HifiAndGPSAndSafeCar);

      cout << "Cost of Default car: " << defaultCar->CalculateCost() <<
      endl;
      cout << "Cost of Safe car: " << safeCar->CalculateCost() << endl;
      cout << "Cost of GPS and Safe car: " <<
      GPSAndSafeCar->CalculateCost() << endl;
      cout << "Cost of GPS and Safe and HiFi car: " <<
      HifiAndGPSAndSafeCar->CalculateCost() << endl;
      cout << "Cost of Auto and GPS and Safe and HiFi car: " <<
      AutoAndHifiAndGPSAndSafeCar->CalculateCost() << endl;

      delete defaultCar;
      delete safeCar;
      delete GPSAndSafeCar;
      delete HifiAndGPSAndSafeCar;
      delete AutoAndHifiAndGPSAndSafeCar;
}
```

3.2.1.4. Ưu điểm

- Decorator Pattern cung cấp một giải pháp linh hoạt hơn khi thêm các chức năng cho một đối tượng so với cách kế thừa (inheritance) truyền thống.
- Các decorator cung cấp giải pháp để dễ dàng thay đổi hành vi của đối tượng.
- Ngoài ra, code còn dễ dàng hơn đáng kể. Bởi vì bạn sẽ code class chức năng riêng rẽ cụ thể, thay vì gộp chung nhiều chức năng khác nhau vào một class -> Điều này làm cho các thành phần dễ mở rộng hơn trong tương lai.

3.2.1.5. Khuyết điểm

Nhược điểm chính của việc sử dụng Decorator Pattern là việc bảo trì có thể là một vấn đề vì nó cung cấp rất nhiều loại object của các class chức năng.

***So sánh Decorator Pattern - Kế thừa:**

DECORATOR PATTERN	KẾ THỪA
-Được sử dụng để mở rộng chức năng của một đối tượng cụ thể -Không yêu cầu class con -Ngăn chặn sự gia tăng của các class con dẫn đến ít phức tạp và nhầm lẫn -Linh hoạt hơn -Client có thể lựa chọn linh hoạt các chức năng mong muốn -Dễ dàng thêm bất kỳ sự kết hợp của các chức năng. Chức năng tương tự có thể thậm chí được thêm hai lần hay nhiều lần	-Được sử dụng để mở rộng chức năng của một lớp của các đối tượng -Yêu cầu class con -Có thể dẫn đến nhiều class con, làm cho cây phân cấp class trở nên phức tạp -Kém linh hoạt hơn -Có các class con để kết hợp các chức năng bổ sung mà client mong đợi, có thể dẫn đến một sự gia tăng của lớp con -Rất khó

3.2.1.6. Ứng dụng

Decorator pattern được sử dụng khi:

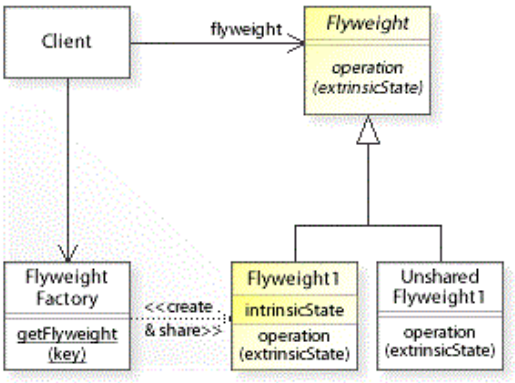
- Khi bạn muốn thay đổi động mà không ảnh hưởng đến người dùng, không phụ thuộc vào giới hạn các class con.
- Có một số đặc tính phụ thuộc mà bạn muốn ứng dụng một cách linh động và bạn muốn kết hợp chúng vào trong một đối tượng.
- Khi các class con không còn khả thi, khi mà cần một số lượng lớn các kết hợp trong một đối tượng.

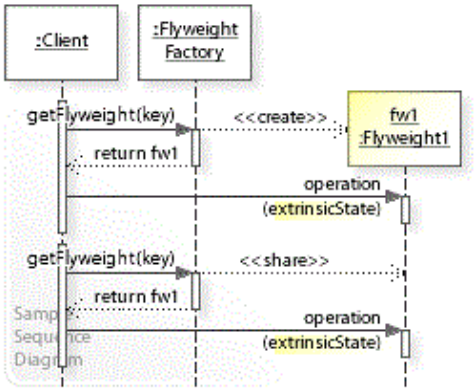
3.2.2. Flyweight

3.2.2.1. Định nghĩa

Flyweight là một mẫu thiết kế phần mềm. Khi nhiều đối tượng (objects) phải được xử lý mà chương trình không thể chịu nổi một lượng dữ liệu khổng lồ, thì cần dùng flyweight.

3.2.2.2. Sơ đồ UML





3.2.2.3. Sample code

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

using namespace std;

// The 'Flyweight' abstract class
class Character
{
public:
  virtual void Display(int pointSize) = 0;

protected:
  char symbol_;
  int width_;
  int height_;
  int ascent_;
```

```
  int descent_;
  int pointSize_;
};

// A 'ConcreteFlyweight' class
class CharacterA : public Character
{
public:
  CharacterA()
  {
    symbol_    = 'A';
    width_     = 120;
    height_    = 100;
    ascent_    = 70;
    descent_   = 0;
    pointSize_ = 0; //initialise
  }
  void Display(int pointSize)
  {
    pointSize_ = pointSize;
    cout<<symbol_<<" (pointsize "<<pointSize_<<" )"<<endl;
  }
};

// A 'ConcreteFlyweight' class
class CharacterB : public Character
{
public:
  CharacterB()
  {
    symbol_    = 'B';
    width_     = 140;
    height_    = 100;
    ascent_    = 72;
    descent_   = 0;
    pointSize_ = 0; //initialise
  }
  void Display(int pointSize)
  {
    pointSize_ = pointSize;
    cout<<symbol_<<" (pointsize "<<pointSize_<<" )"<<endl;
  }
```

```
};

//C, D, E,...

// A 'ConcreteFlyweight' class
class CharacterZ : public Character
{
public:
  CharacterZ()
  {
    symbol_    = 'Z';
    width_     = 100;
    height_    = 100;
    ascent_    = 68;
    descent_   = 0;
    pointSize_ = 0; //initialise
  }
  void Display(int pointSize)
  {
    pointSize_ = pointSize;
    cout<<symbol_<<" (pointsize "<<pointSize_<<" )"<<endl;
  }
};

// The 'FlyweightFactory' class
class CharacterFactory
{
public:
  virtual ~CharacterFactory()
  {
    while(!characters_.empty())
    {
      map<char, Character*>::iterator it = characters_.begin();
      delete it->second;
      characters_.erase(it);
    }
  }
  Character* GetCharacter(char key)
  {
    Character* character = NULL;
    if(characters_.find(key) != characters_.end())
    {
```

```
      character = characters_[key];
    }
    else
    {
      switch(key)
      {
      case 'A':
        character = new CharacterA();
        break;
      case 'B':
        character = new CharacterB();
        break;
      //...
      case 'Z':
        character = new CharacterZ();
        break;
      default:
        cout<<"Not Implemented"<<endl;
        throw("Not Implemented");
      }
      characters_[key] = character;
    }
    return character;
  }
private:
  map<char, Character*> characters_;
};


//The Main method
int main()
{
  string document = "AAZZBBZB";
  const char* chars = document.c_str();

  CharacterFactory* factory = new CharacterFactory;

  // extrinsic state
  int pointSize = 10;

  // For each character use a flyweight object
  for(size_t i = 0; i < document.length(); i++)
```

```
  {
    pointSize++;
    Character* character = factory->GetCharacter(chars[i]);
    character->Display(pointSize);
  }

  //Clean memory
  delete factory;

  return 0;
}
```

3.2.2.4. Ưu điểm

Cho phép trừu tượng ở mức tốt nhất (Một số ít đối tượng được tạo ra, các đối tượng có thể đại diện cho vô số các đối tượng quan hệ logic với nhau).

3.2.2.5. Khuyết điểm

- Mất thêm thời gian để cài đặt một đối tượng flyweight và nếu phải cài đặt bao bọc mọi thứ, có thể sẽ làm giảm hiệu năng hệ thống nhiều hơn mong đợi.
- Vì tách một lớp mẫu chung ra khỏi đối tượng để tạo flyweight, nên phải thêm vào một lớp khác trong việc lập trình, và có thể gây ra sự khó khăn trong việc bảo trì và mở rộng.
- Gặp một số rắc rối cho việc xử lí đa luồng.

TÀI LIỆU THAM KHẢO:

-Sách

+Design patterns Elements of Reusable Object Oriented Software - Erich Gamma, John Vlissides, Ralph Johnson, Richard Helm

+Design Patterns For Dummies - Steven Holzner

+Head First Design Patterns - Elisabeth Freeman, Kathy Sierra

http://diendan.congdongcviet.com/threads/t76502::tong-hop-series-bai-dich-thiet-ke-mau-design-patterns-for-dummies.cpp

https://toidicodedao.com/2016/03/01/nhap-mon-design-pattern-phong-cach-kiem-hiep/

http://geekswithblogs.net/subodhnpushpak/archive/2009/09/18/the-23-gang-of-four-design-patterns-.-revisited.aspx

https://techtalk.vn/design-pattterns-he-thong-23-mau-design-patterns.html

-Singleton

https://en.wikipedia.org/wiki/Singleton_pattern

https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

https://sourcemaking.com/design_patterns/singleton

https://www.stdio.vn/articles/read/224/design-pattern-singleton-pattern

http://www.oodesign.com/singleton-pattern.html

-Flyweight

https://en.wikipedia.org/wiki/Flyweight_pattern

https://sourcemaking.com/design_patterns/flyweight

https://www.tutorialspoint.com/design_pattern/flyweight_pattern.htm

https://www.merriam-webster.com/dictionary/flyweight

http://www.oodesign.com/flyweight-pattern.html

-Decorator

https://vi.wikipedia.org/wiki/Decorator_pattern

https://en.wikipedia.org/wiki/Decorator_pattern

https://sourcemaking.com/design_patterns/decorator

https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm

https://viblo.asia/p/hieu-biet-co-ban-ve-decorator-pattern-pVYRPjbVG4ng

http://oop.misamap.com/2013/03/decorator.html

-Factory Method

https://en.wikipedia.org/wiki/Factory_method_pattern

https://sourcemaking.com/design_patterns/factory_method

https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

https://viblo.asia/p/design-pattern-factory-pattern-part-2-XQZGxZqjkwA

http://www.geeksforgeeks.org/design-patterns-set-2-factory-method/

http://www.oodesign.com/factory-method-pattern.html

https://tndhuy.wordpress.com/2011/05/29/factory-method/