Design Pattern

Table of Contents

Decorator

Problem

Giả sử, bạn đang làm nhân viên IT của một công ty chuyên sản xuất và lắp ráp xe hơi. Giám đốc công ty giao cho bạn nhiệm vụ viết phần mềm để quản lý dây chuyền lắp ráp. Công ty của bạn từ trước đến nay chỉ sản xuất một dòng xe cơ bản. Là một lập trình viên, bạn đã định nghĩa ra một class tên là Car, đại diện cho dòng sản phẩm của công ty của bạn chế tạo và phương thức CalculateCost() để tính toán giá thành sản phẩm như sau:

```
class Car {
    private:
        float costOfWheels = 4.0;
        float costOfChassis = 10.5;
        float costOfEngine = 35;

    public:
        float CalculateCost() {
            return costOfWheels + costOfChassis + costOfEngine;
        }
};
```

Do nhu cầu phát triển và đổi mới của công ty, vị giám đốc quyết định cho ra dòng xe mới có gắn thiết bị GPS dùng để định vị và chỉ đường.

Bạn có trách nhiệm thay đổi và nâng cấp hệ thống để bảo đảm dây chuyền sản xuất có thể đám ứng nhu cầu của vị giám đốc. Bạn thấy là dòng xe mới kia thật ra chẳng khác gì chiếc xe truyền thống, và chỉ gắn thêm thiết bị GPS trên xe mà thôi. Với kinh nghiệm của mình, bạn nhận thấy rằng sử dụng tính chất thừa kế trong lập trình hướng đối tượng có thể giải quyết

```
class GpsCar : Car {
   private:
      float costOfGpsDevice = 9.3;
   public:
      float CalculateCost() {
         return costOfGpsDevice + Car::CalculateCost();
      }
};
```

Hào hứng với kết quả kinh doanh phát triển nhờ dòng xe mới, vị giám đốc liền lên kế hoạch cho ra thêm tiếp 3 dòng xe mới nữa với những tính năng hấp dẫn:

- SafeCar: Dòng xe được gắn cảm biến nhằm đảm bảo an toàn tốc độ cho người điều khiển xe.
- HiFiCar: Dòng xe giải trí, được trang bị thống thống loa và âm li tiên tiến để đáp ứng nhu cầu âm nhạc.
- AutoCar: Dòng xe thông minh thế hệ cao, được trang bị hệ thống xử lý thông minh, có thể tự động vận hành.

Bạn sẽ thấy ổn thôi, thêm 3 dòng xe mới, vậy thì sẽ có 3 class con mới, mọi việc sẽ được giải quyết nhanh thôi:

```
class SafeCar : Car {
   private:
      float costOfSafeSensors = 15;
   public:
      float CalculateCost() {
         return costOfSafeSensors + Car::CalculateCost();
      }
};

class HiFiCar : Car {
   private:
      float costOfSpeaker = 9;
      float costOfAmpli = 15;
   public:
      float CalculateCost() {
         return costOfSpeaker + costOfAmpli + Car::CalculateCost();
      }
};

class AutoCar : Car {
   private:
      float costOfAutoProcessor = 20;
   public:
      float CalculateCost() {
         return costOfAutoProcessor + Car::CalculateCost();
      }
};
```

Mọi chuyện sẽ tốt đẹp cho đến một ngày, vị giám đốc kia lại nảy ra một ý tưởng mới. Ông ta muốn kết hợp những tính năng trước đây lại để sản xuất:

- GPS and Safe car
- GPS and Hifi car
- GPS and Auto car
- Safe and Hifi car

- Safe and Auto car
- etc.

Ồ, mọi chuyện thật đơn giản, bao nhiêu dòng xe thì bấy nhiêu class, cũng đơn giản thôi mà. Uhm, bạn suy nghĩ một lúc ... nhưng mà hình như có gì đó không ổn, cách làm này không ổn chút nào, số lượng class được tạo ra quá nhiều. Giả sử sau này vị giám đốc kia tiếp tục ra thêm những tính năng khác nữa và kết hợp với nhau thì số lượng class còn tăng lên gấp nhiều lần. Đang lúc rồi bời, vị giám đốc xuất hiện bên bạn và bảo bạn:

"Anh thử xem xét tới Decorator Pattern xem, có thể nó giải quyết vấn đề của chúng ta đấy"

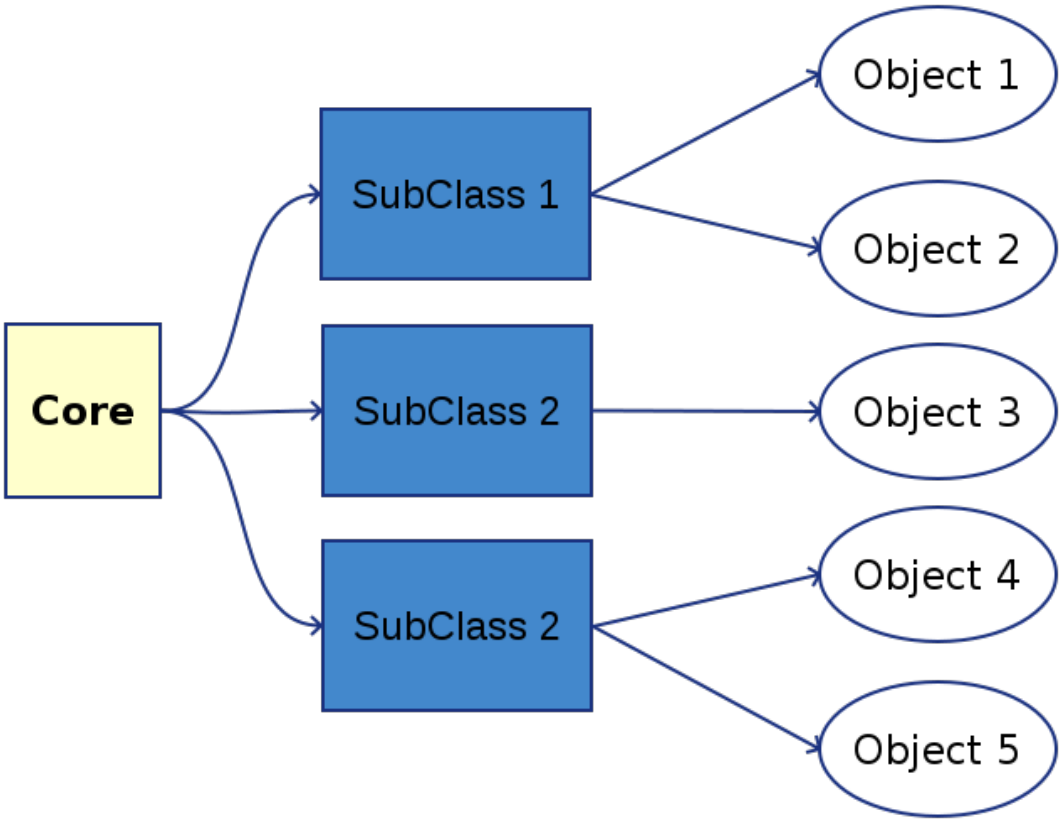
Defination

Định nghĩa trong quyển sách GOF về Decorator Pattern như sau:

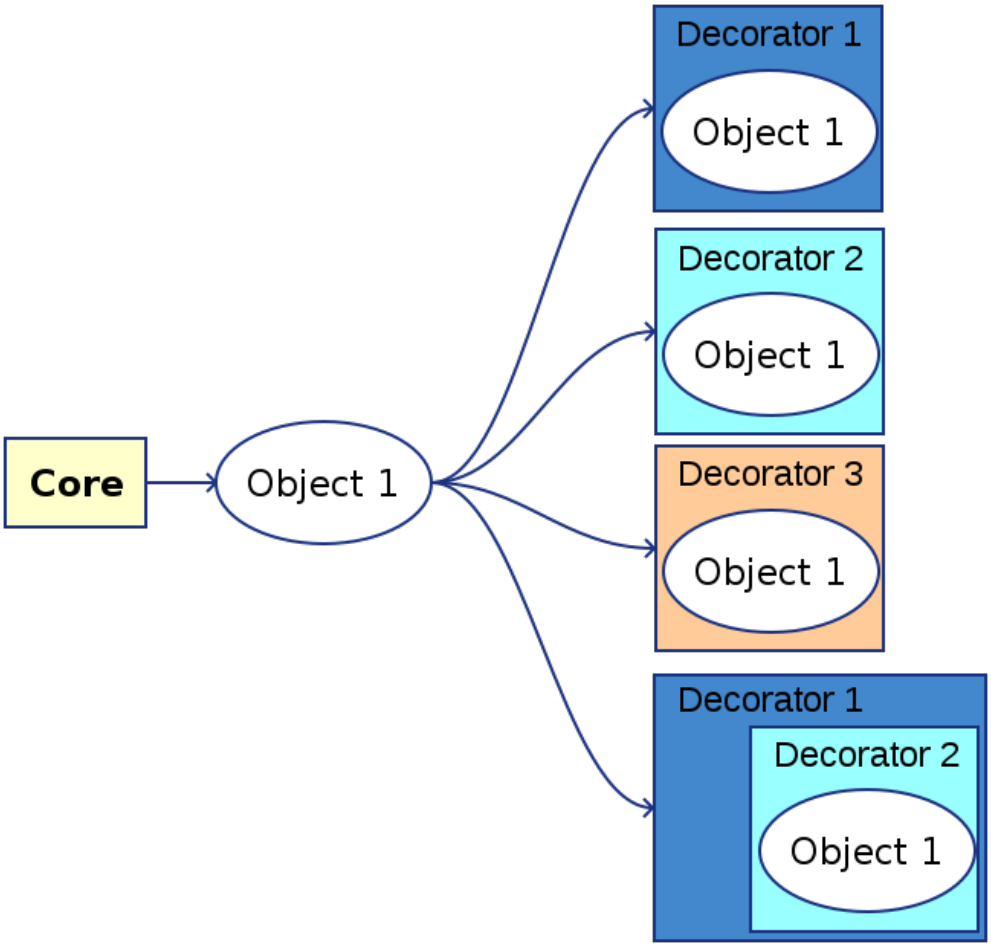
> Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator pattern cung cấp cách thức để thêm tính năng vào một object một cách linh động và cung cấp một cách khác để mở rộng tính năng cho một object.

Đôi khi bạn muốn mở rộng tính năng của một đối tượng có sẵn. Tính thừa kế trong OOP (hướng đối tượng) là một trong những cách thức để làm việc đó. Phạm vi ảnh hưởng của tính thừa kế trong OOP là mức class. Việc đó trong một số tình huống thật sự không linh động do ta chỉ muốn mở rộng dựa trên từng object nhất định chứ không phải toàn bộ class. Ngoài ra, tính năng mở rộng của object phụ thuộc vào class con.

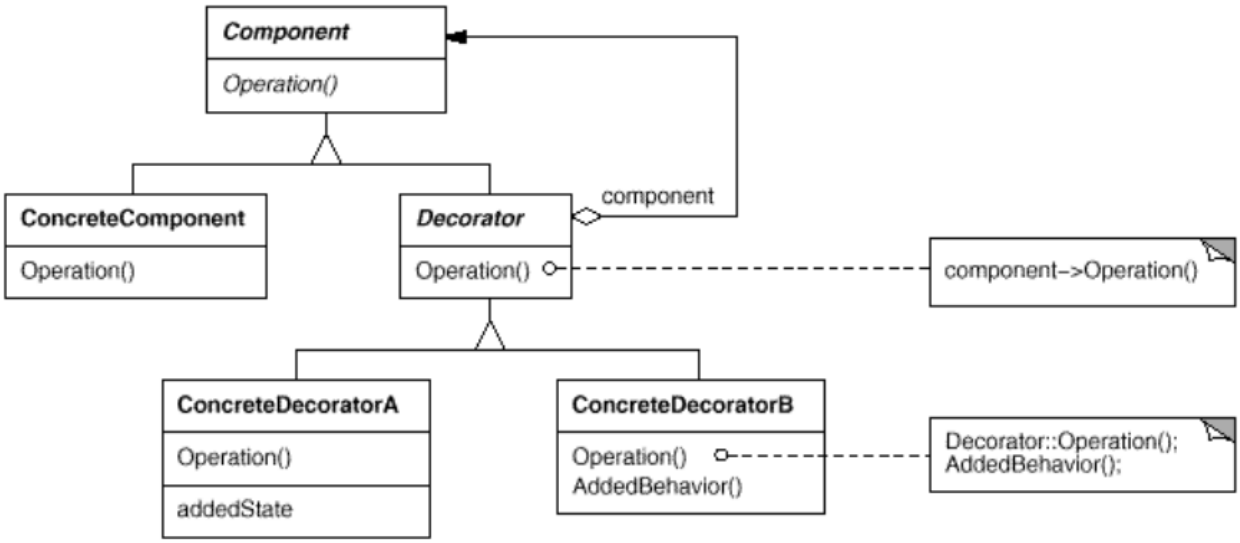


Một cách thức khác để mở rộng tính năng của một object đó là decorator. Decorator pattern dựa trên nguyên tắc wrapper để giải quyết vấn đề mở rông tính năng của một object. Decorator sẽ như lớp vỏ, bọc xung quanh object cần mở rộng và cung cấp tính năng cần thiết cho object đấy. Và do thế, phạm vi ảnh hưởng của Decorator chỉ nằm trên mức object riêng lẻ và thể thể linh hoạt thêm tính năng tùy ý.



Structure

Cấu trúc của Decoration Pattern được bao trọn trong class diagram sau:



Component: vai trò như interface, được hiện thực bởi ConcreateComponent và Decorators. Khi sử dụng thực tế, đây là đại diện để các decorators có thể add chức năng vào.

ConcreateComponent: Hiện thực của component, là object gốc cần mở rộng chức năng. ConcreateComponent nên tinh gọn và cô đọng. ConcreateComponent sẽ thừa kế Component.

Decorator: đóng vai trò như interface của những decorators. Decorator sẽ thừa kế Component và cũng chứa một component trong nó.

ConcreateDecoratorA, ConcreateDecoratorB: hiện thực của Decorator, đóng vai trò như những wrapper, mở rộng chức năng cho Component được truyền vào. Những ConcreateDecorators sẽ thừa kế Decorators

Resolving

Để hiểu rõ hơn về Decorator pattern, chúng ta hãy áp dụng vào vào trường hợp của vị giám đốc và anh chàng kỹ thuật thuật của công ty lắp ráp xe hơi nhé.

Đầu tiên, ta hãy xóa hết các class con mà anh chàng này tạo ra, chúng ta bắt đầu với thành phần đầu tiên của Decorator pattern, Component:

```
class Car {
    public:
        virtual double CalculateCost();
}
```

Tiếp theo, chúng ta sẽ định nghĩa ConcreateComponent, đây là hiện thực của interface Car:

```
class DefaultCar : Car {
   private:
      float costOfWheels = 4.0;
      float costOfChassis = 10.5;
      float costOfEngine = 35;

   public:
      float CalculateCost() {
         return costOfWheels + costOfChassis + costOfEngine;
      }
};
```

Vậy là xong nhánh bên Component. Chúng ta tiếp đến với interface Decorator:

```
class Decorator : Car {
   private:
      Car car;
   public:
      Decorator(Car paramCar) {
         car = paramCar;
      }
         virtual float CalculateCost();
};
```

Decorator sẽ đóng vai trò như interface của tất cả các ConcreateDecorator. Nó sẽ yêu cầu truyền interface Car vào khi khởi tạo. Và chính Decorator thừa kế từ interface Car nên những ConcreateDecorator có thể truyền ConcreateDecorator khác vào. Về phần ConcreateDecorator, chúng ta sẽ tạo lần lượt 4 cái tương ứng với từng features: GPS, Safe, Hifi, và Auto

```
class GPSDecorator: public Decorator {
   private:
      float costOfGpsDevice = 9.3;
   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 = 9;
      float costOfAmpli = 15;
   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;
      }
};
```

Và khi sử dụng, chúng ta chỉ đơn giản là sử dụng decorator mình mong muốn để mở rộng tính năng cho DefaultCar

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

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

  return 0;
}
```

Vậy là chàng kỹ thuật viên sống hạnh phúc với ông giám đốc đến trọn đời :D

Features

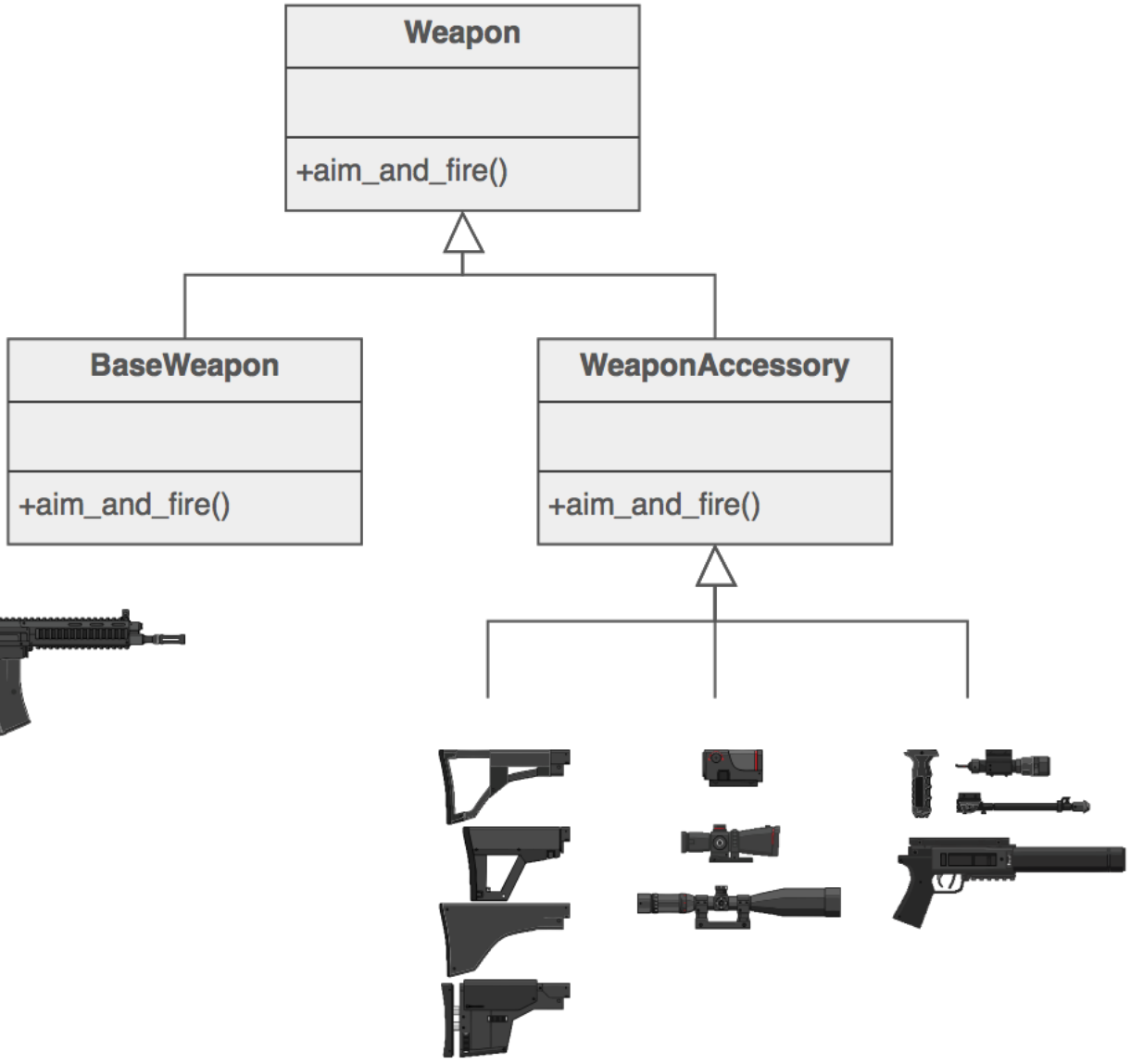
- Decorator có thể mở rộng chức năng của object riêng biệt chứ không phải của toàn bộ class.
- Dùng Decorators có thể mở rộng một object lúc runtime. Do việc mở rộng bằng những class decorators nên có thể chủ động lúc runtime. Trong khi dùng tính chất thừa kế của OOP bị hạn chế trong lúc định nghĩa.
- Các Decorators không bị phụ thuộc lẫn nhau. Do được định nghĩa trên các class khác nhau, việc sửa đổi hay chỉnh sửa Decorator này không làm ảnh hưởng tới các decorator khác.
- Như ví dụ ở trên, các decorators có thể xử dụng trỗn lẫn với nhau để cho ra object có chức năng như mong muốn.
- Có thể sử dụng một decorator hai lần. Ví dụ một chiếc xe có 5 thiết bị GPS hoặc có 3 hàn âm thanh Hifi.

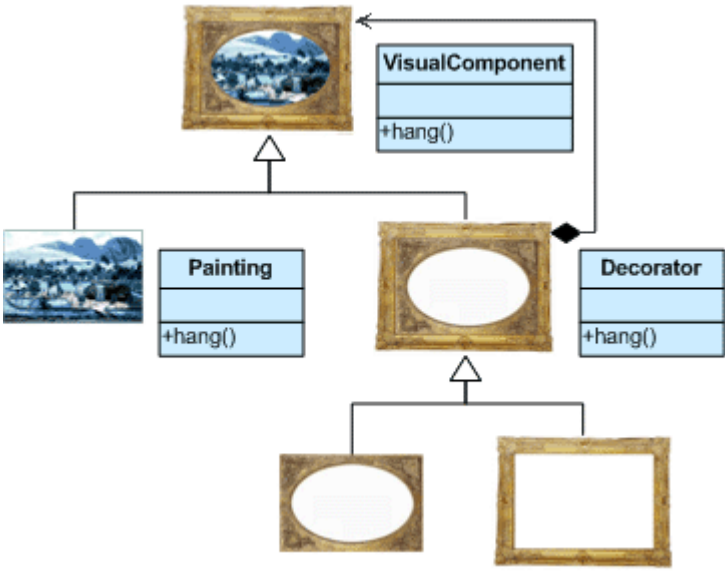
Usages

- Được sử dụng trong I/O API stream

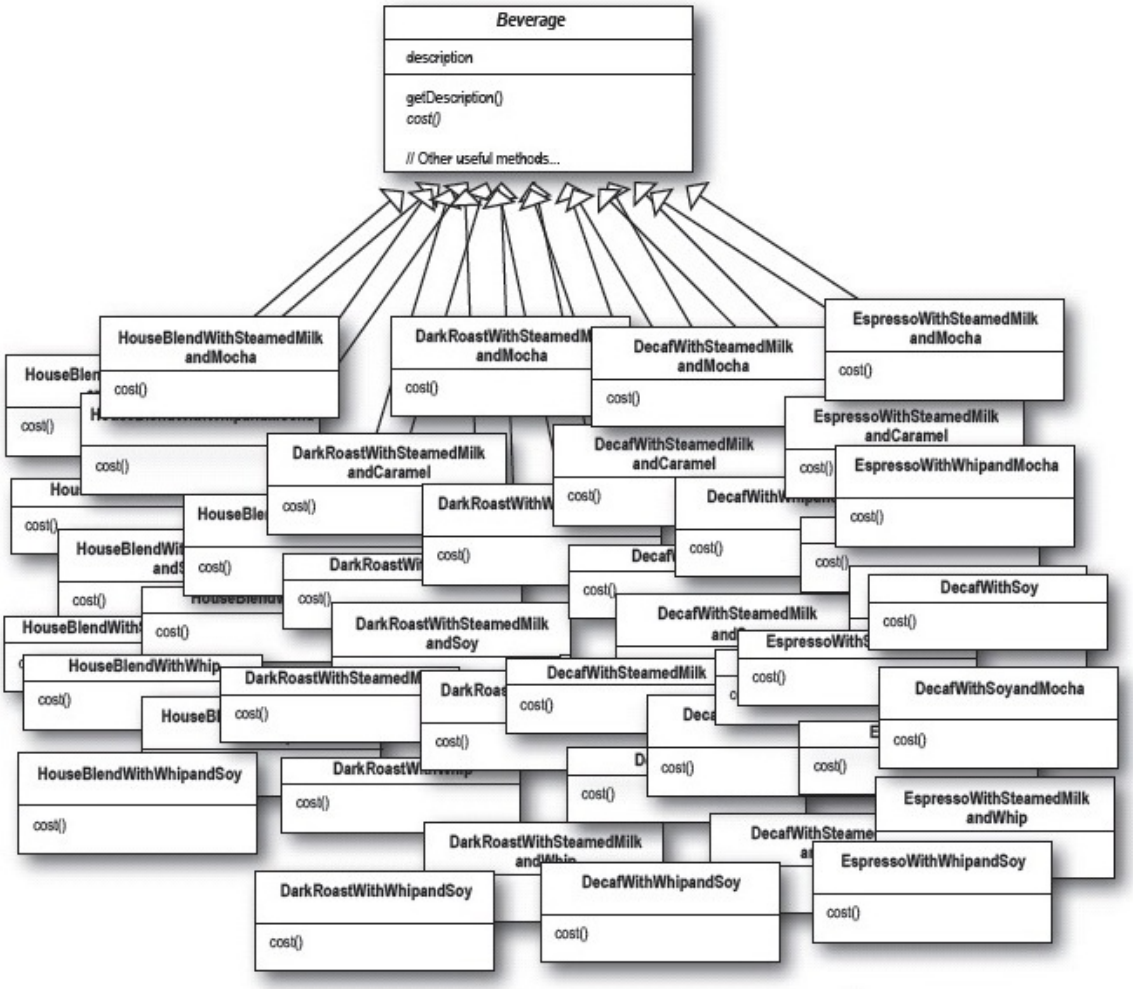
Component:	System.IO.Stream
Decoration:	System.IO.BufferedStream
Decoration:	System.IO.FileStream
Decoration:	System.IO.MemoryStream
Decoration:	System.Net.Sockets.NetworkStream
Decoration:	System.Security.Cryptography.CryptoStream

- Được sử dụng nhiều trong GUI programming
- Được sử dụng khi bạn muốn mở rộng tính năng của một class nhưng không thể kế thừa từ class đó.
- Được sử dụng khi mở rộng tính năng của một object lúc runtime hoặc static.
- Muốn tránh tình trạng quá nhiều class con được tạo ra như trong ví dụ ở trên.
- Muốn thay đổi chức năng của một object nhưng không muốn ảnh hưởng tới những object cùng class đó.
- Vài ví dụ vui để hiểu rõ hơn về Decorator Pattern:


Weapon
+aim_and_fire()
BaseWeapon
+aim_and_fire()
WeaponAccessory
+aim_and_fire()



- Một hậu quả của việc không không sử dụng Decoration Pattern mà sử dụng Inheritance:



Facade pattern

Problem

Giả sử, bạn là nhân viên IT của một tập đoàn xe hơi, bạn phụ trách tất cả vấn đề kỹ thuật của cả công ty. Sắp tới, công ty bạn có buổi thuyết trình với một khách hàng đặc biệt và bạn cần in vài hồ sơ. Rủi thay, toàn bộ máy in của công ty bạn có vấn đề, bạn không thể chờ đến lúc nhân viên máy in sửa xong được. Bạn đành phải xách USB ra ngoài tiệm in và khó khăn bắt đầu xuất hiện.

"Tôi có thể in tài liệu được không" - bạn bước vào một cửa hàng in ấn.

"Được chứ. Nhưng phiền anh tự in được không, tôi đang dở tay, anh dùng máy tính gần máy in ấy" - vị chủ tiệm đang lúi cúi chất hàng.

"Được thôi" - Dù gì bạn cũng là nhân viên kỹ thuật, chẳng lẽ in một tập tài liệu lại làm khó đến bạn.

Bạn cắm USB vào máy tính, mở tập tin cần in ra. Bạn tìm nút "Print" trên chiếc máy in nhưng gần như vô vọng, có hàng tá nút trên ấy, nhưng chẳng cái nào là nút in cả.

"Ông chủ ơi, tôi mở tập tin lên rồi, nhưng làm sao in đây..."

"À, anh phải nhấn nút CheckPaper trên máy in để kiểm tra giấy"

"Rồi, vậy bây giờ in được rồi chứ ?"

"Chưa đâu, giờ tiếp tục nhấn nút CheckInk để kiểm tra mực in"

"Rồi, vậy bây giờ in được rồi chứ ?"

"Chưa, anh nhấn thêm nút Warming để làm nóng máy in"

"Đã nhấn, vậy giờ máy chuẩn bị in rồi đúng không ?"

"Anh có vẻ gấp đấy nhỉ, nhưng chưa đâu, anh giờ phải nhấn tiếp nút GetDocument để tải tập tin từ máy tính vào máy in"

"Chúng ta còn phải nhấn bao nhiêu nút nữa hả ông chủ" - bạn có vẻ hơi khó chịu về cách cái máy in hoạt động.

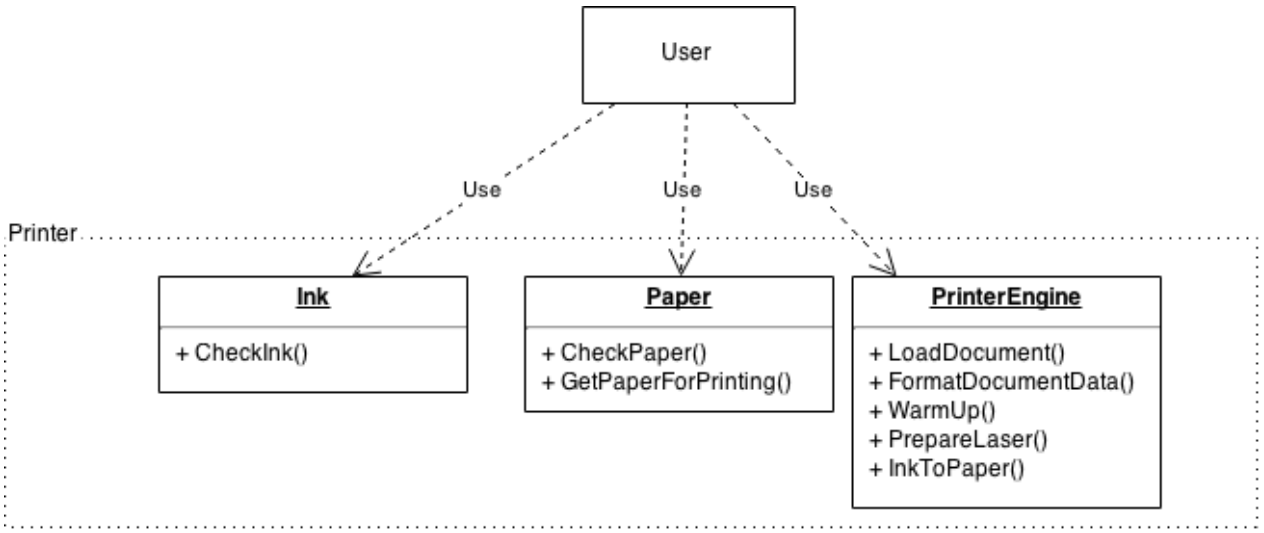
"Anh bạn trẻ ạ, anh còn phải nhấn FormatData để chuyển tài liệu của anh thành format của máy in, rồi phải nhấn tiếp nút GetPaper để chuyển giấy lên phần in, rồi nút PrepareLaser để chuẩn bị đầu đọc, ..."

Lúc này, bạn có vẻ mất bình tĩnh: "Tại sao lại có chiếc máy in như vầy nhỉ. Những chiếc máy in tôi biết chỉ cần nhấn nút Print, thế là nó sẽ làm mọi việc cho tôi".

"À há" - Ông chủ thốt lên - "Ý tưởng của anh có vẻ hay đấy, chiếc máy in này do tôi tạo ra, và ý tưởng của anh có vẻ hay đấy. Tôi sẽ sửa lại chiếc máy in, giờ anh tiếp tục bấm thêm vài nút nữa nhé."

Bạn tiếp tục cho xong công việc của mình, miệng lẩm bẩm: "Giá mà ông biết về Facade Pattern sớm thì hay biết mấy".

Để hiểu rõ hơn, tôi xin mô tả lại chiếc máy in nọ bằng code và diagram




```
class Ink {
   public:
      void CheckInk() {
         cout << "+ Check ink done" << "\n";
      }
};
```

```
class Paper {
   public:
      void CheckPaper() {
         cout << "+ Check paper" << "\n";
      }
      void GetPaperForPrinting() {
         cout << "+ Get paper for printing" << "\n";
      }
};
```

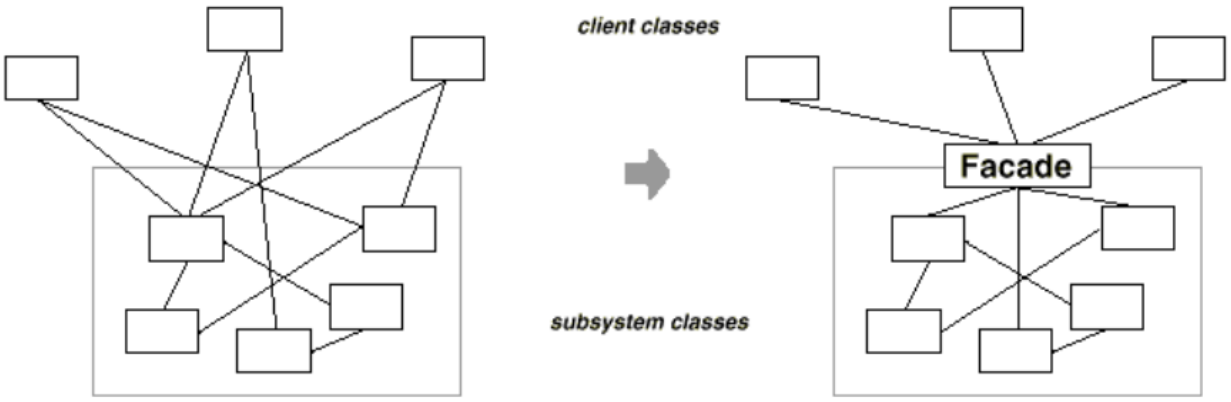
```
class PrinterEngine {
   public:
      void LoadDocument() {
         cout << "+ Load document from computer" <<
"\n";
      }
      void FormatDocumentData() {
         cout << "+ Format data" << "\n";
      }
      void WarmUp() {
         cout << "+ Engine was warm up" << "\n";
      }
      void PrepareLaser() {
         cout << "+ Prepare laser" << "\n";
      }
      void InkToPaper() {
         cout << "+ Ink to paper" << "\n";
      }
};
```

```
int main() {
    cout << "I want to print document" << "\n";

    Ink ink;
    Paper paper;
    PrinterEngine engine;

    ink.CheckInk();
    paper.CheckPaper();
    engine.LoadDocument();
    engine.FormatDocumentData();
    paper.GetPaperForPrinting();
    engine.PrepareLaser();
    engine.WarmUp();
    engine.InkToPaper();

    cout << "I had printed document" << "\n";
}
```

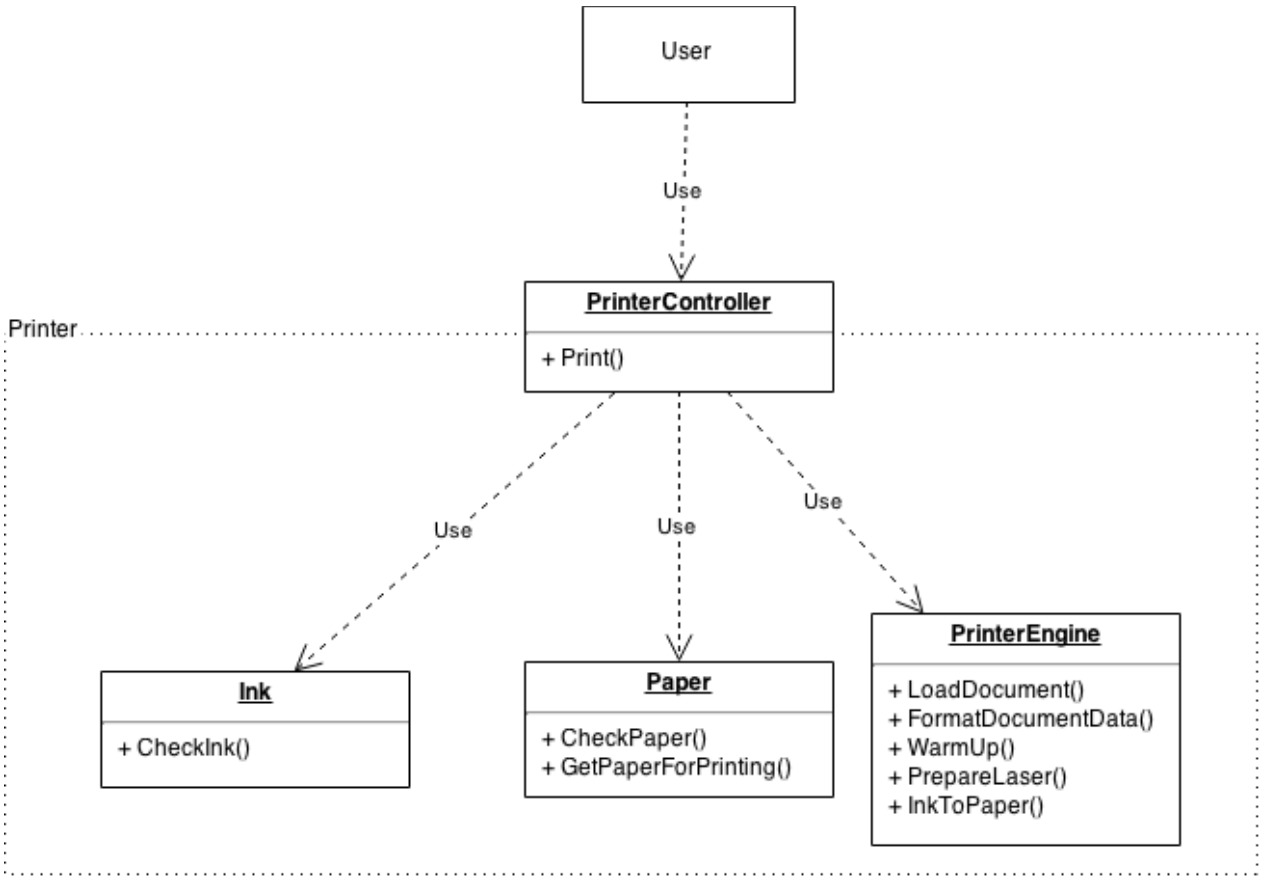
Defination

Qua ví dụ, bạn có thể thấy user, người sử dụng máy in phải bận tâm quá nhiều vào sự phức tạp của cấu tạo và cách thức hoạt động bên trong máy in, trong khi họ không cần phải biết về chúng quá nhiều. Và Facade pattern ra đời để giải quyết bài toán đó.

Facade pattern cung cấp một đại diện cho những cho những thành phần trong hệ thống. Thành phần đại diện này trong facade giúp cho hệ thống đó trở nên bớt phức tạp, dễ hiểu và dễ sử dụng hơn.



Resolving



Để áp dụng Facade giải quyết bài toán của chiếc máy in nhiều nút, ta tạo ra một Facade class tên là PrinterController. Class này dùng để điều khiển các bước hoạt động khi in, và người dùng chỉ cần biết đến class này mà thôi, không cần quan tâm đến sự phức tạp của hệ thống máy in bên trong.

```
class PrinterController {
   public:
      void Print() {
         Ink ink;
         Paper paper;
         PrinterEngine engine;

         ink.CheckInk();
         paper.CheckPaper();
         engine.LoadDocument();
         engine.FormatDocumentData();
         paper.GetPaperForPrinting();
         engine.PrepareLaser();
         engine.WarmUp();
         engine.InkToPaper();
      }
}
```

Khi đó, đối với người dùng, mọi việc trở nên đơn giản hơn

```
int main() {
   cout << "I want to print document" << "\n";
   PrinterControler printer;
   printer.Print();
   cout << "I had printed document" << "\n";
}
```

Features

- Giảm thiểu số lượng class mà client phải quan tâm khi sử dụng hệ thống. Giảm bớt sự phức tạp của hệ thống khi sử dụng.
- Giảm thiểu sự phụ thuộc giữa phần code. Do chỉ cần quan tâm đến facade class, người sử dụng không bị ảnh hưởng khi các thành phần hoặc cấu trúc bên trong của hệ thống bị thay đổi.
- Giải pháp này không cản trở hay che dấu các class bên trong với client. Nên họ có thể quyết định cân nhắc việc cách nào cho phù hợp.

Usages

- Thường được sử dụng khi định nghĩa library. Facade đóng vai trò như người đại diện của library, giúp người dùng dễ dàng sử dụng library hơn.
- Sử dụng để giao tiếp giữa các high level component và low level component. Vd: GUI component giao tiếp với Persistance component. Facade lúc này đóng vai trò giống như những đầu nối giao tiếp giữa các components trong một hệ thống lớn của bạn.

Adapter Pattern

Problem

Defination

Structure

Resolving

Feature

Usages

Strategy Pattern

Problem

Giả sử, bạn là một lập trình viên của một công ty lắp ráp và sản xuất. Công ty của bạn được chia thành 5 nhóm làm việc chính, bao gồm: CEO, manager, salesperson, IT và janitor (lao công). Sếp của bạn nhờ bạn viết một phần mềm tính toán tiền thưởng cuối năm cho từng nhóm làm việc trong công ty, mỗi nhóm sẽ có công thức tính toán tiền thưởng riêng.

Bạn nhận nhiệm vụ và viết ngay một module dành cho việc tính toán lương như sau:

```
enum UserGroup {
    CEO,
    MANAGER,
    SALESPERSON,
    IT,
    JANITOR
};
```

```
class BonusProcessor {
   public:
      double GetBonus(UserGroup userGroup) {
         double bonus = 0;
         switch(userGroup) {
            case CEO:
               // a + b * c - z
               bonus = 100;
               break;
            case MANAGER:
               // a - b / c + z
               bonus = 70;
               break;
            case SALESPERSON:
               // a * b - c / z
               bonus = 50;
               break;
            case IT:
               // a * b - c
               bonus = 30;
               break;
            case JANITOR:
               // a
               bonus = 10;
               break;
         }
         return bonus;
      }
};
```

```
int main() {
    BonusProcessor* processor = new BonusProcessor();
    double bonus;

    bonus = processor->GetBonus(CEO);
    cout << "Bonus of CEO: " << bonus << "\n";

    bonus = processor->GetBonus(MANAGER);
    cout << "Bonus of MANAGER: " << bonus << "\n";

    bonus = processor->GetBonus(SALESPERSON);
    cout << "Bonus of SALESPERSON: " << bonus << "\n";

    bonus = processor->GetBonus(IT);
    cout << "Bonus of IT: " << bonus << "\n";

    bonus = processor->GetBonus(JANITOR);
    cout << "Bonus of JANITOR: " << bonus << "\n";
}
```

(phần logic để tính toàn bonus cho từng group trong function GetBonus, mình viết tinh gọn lại cho bạn dễ hiểu, thật ra chỗ đó sẽ công trừ nhân chia, kiểm tra logic rất nhiều)

Bạn chạy thử, kết quả như mong muốn của bạn. Nhưng bạn thấy có điều gì đó không ổn. Function GetBonus() của bạn chứa quá nhiều logic, bạn ôm đồm quá nhiều việc trong cùng một function. Việc này khiến code của bạn trở nên rối, khó bảo trì cũng như thay đổi. Bạn quyết định thay đổi class BonusProcessor một tí.

```
class BonusProcessor {
   public:
      double GetBonus(UserGroup userGroup) {
         double bonus = 0;
         switch(userGroup) {
            case CEO:
               bonus = CalculateCEOBonus();
               break;
            case MANAGER:
               bonus = CalculateManagerBonus();
               break;
            case SALESPERSON:
               bonus = CalculateSalespersonBonus();
               break;
            case IT:
               bonus = CalculateITBonus();
               break;
            case JANITOR:
               bonus = CalculateJanitorBonus();
               break;
         }
         return bonus;
      }
      double CalculateCEOBonus() {
         return 100 * 6 + 3 - 2 +7;
      }
      double CalculateManagerBonus() {
         return 200 / 2 + 4 +6;
      }
      double CalculateSalespersonBonus() {
         return 50 * 3 / 2 - 7;;
      }
      double CalculateITBonus() {
         return 20 + 7 - 2;
      }
      double CalculateJanitorBonus() {
         return 10 - 1 - 5;
      }
};
```

Class BonusProcessor của bạn khá hơn hẳn so với ban đầu, logic tính toán tiền của từng bộ phận được tách riêng nên bớt rối hơn.

Nhưng bạn vẫn đắn đo về một việc, logic tính toán thưởng được đặt trong 1 class duy nhất. Bạn cũng thấy cách viết này có nhiều hạn chế, bạn phải sử dụng nhiều câu lệnh điều kiện (switch) để kiểm tra từng nhóm được định nghĩa trước, việc này sẽ khó khăn trong việc phát triển ứng dụng sau này. Nếu sau này công ty bạn có thêm một phòng ban mới, ngoài việc định nghĩa công thức tính toán, bạn phải sửa lại logic của function GetBonus, việc này làm code của bạn trở nên dễ sai sót hơn.

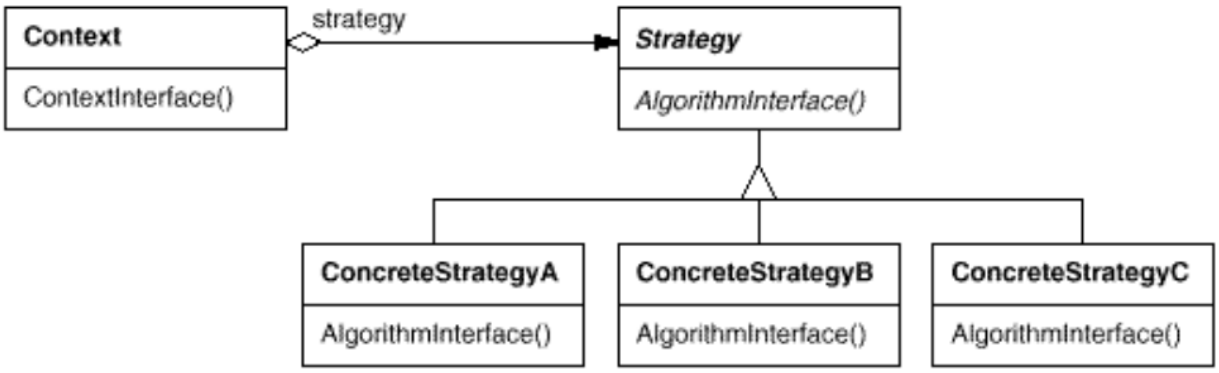
Trong khi bạn đang vò đầu bức tóc, thì giám đốc đáng mến đến gần và nói với bạn: "Anh thử sử dụng Strategy Pattern chưa ? Nó sẽ có ích trong trường hợp này đó".

Defination

Vậy Strategy pattern là gì, đây là một design pattern cho phép định nghĩa một tập các giải thuật có quan hệ với nhau, mỗi giải thuật được cô lập trong những class khác nhau, và chúng có thể thể hoán đổi cho nhau tùy vào hoàn cảnh. Strategy pattern cung cấp một cách thức cho phép chọn và sử dụng giải thuật phù hợp lúc runtime (lúc phần mềm đang được thực thi).

Structure

Cấu trúc của Strategy Pattern như sau:



Strategy: là interface chung để cho các ConcreateStrategy implement. Context sẽ sử dụng interface này thay vì sử dụng trực tiếp ConcreateStrategy.

ConcreateStrategy: hiện thực interface Strategy, sẽ chứa các logic cần thiết

Context: sẽ chứa interface Strategy và được configed với ConcreateStrategy tương ứng trong lúc runtime.

Resolving

Lý thuyết loằng ngoằng là thế, giờ chúng ta hãy xem xét và áp dụng strategy pattern vào để giải quyết bài toán tính lương thưởng nhé. Chúng ta hãy phân tích từng thành phần trong strategy và apply nó vào code nhé.

Đầu tiên là interface Strategy, đóng vai trò đại diện cho các class ConcreateStrategy:

```
class Group {
   public:
      Group() {};
      virtual double CalculateBonus() {};
}
```

Tiếp theo, ta sẽ tạo các class ConcreateStrategy tương ứng với từng nhóm. Những class này sẽ chứa cách thức tính toán lương thưởng cho từng nhóm

```
class CEOGroup : public Group {
    public:
        double CalculateBonus() {
            return 100 * 6 + 3 - 2 +7;
        };
}
class ManagerGroup : public Group {
    public:
        double CalculateBonus() {
            return 200 / 2 + 4 +6;
        };
}
class SalespersonGroup : public Group {
    public:
        double CalculateBonus() {
            return 50 * 3 / 2 - 7;
        };
}
class ITGroup : public Group {
    public:
        double CalculateBonus() {
            return 20 + 7 - 2;
        };
}
class JanitorGroup : public Group {
    public:
        double CalculateBonus() {
            return 10 - 1 - 5;
        };
}
```

Và khi sử dụng:

```
int main() {
    BonusProcessor* processor = new BonusProcessor();
    double bonus;
    int index;

    bonus = processor->GetBonus(new CEOGroup());
    cout << "Bonus of CEO: " << bonus << "\n";

    bonus = processor->GetBonus(new ManagerGroup());
    cout << "Bonus of MANAGER: " << bonus << "\n";

    bonus = processor->GetBonus(new SalespersonGroup());
    cout << "Bonus of SALESPERSON: " << bonus << "\n";

    bonus = processor->GetBonus(new ITGroup());
    cout << "Bonus of IT: " << bonus << "\n";

    bonus = processor->GetBonus(new JanitorGroup());
    cout << "Bonus of JANITOR: " << bonus << "\n";
}
```

Với cách thiết kế trên, cách tính toán lương thưởng được cô lập trong những class khác nhau. Việc này làm code trở nên rõ ràng, trong sáng, dễ chỉnh sửa hơn.

Việc mở rộng thêm tính năng cũng dễ dàng hơn, nếu có một phòng ban mới, ta chỉ việc tạo thêm ConcreateStrategy tương ứng và thừa kế từ interface Strategy. Việc loại bỏ câu lệnh điều kiện (switch) cũng giúp cho ứng dụng linh động hơn, không bị phụ thuộc vào các phòng ban được khai báo trong câu lệnh điều kiện.

Feature

- Giảm sự phức tạp do việc phải nhồi nhét qúa nhiều function, logic và giải thuật vào một class.
- Giảm sự khó khăn trong việc thêm logic và giải thuật, hoặc chỉnh sửa logic và giải thuật cũ.
- Giảm thiểu câu lệnh điều kiện (if, switch, conditional expression) để lựa chọn logic và giải thuật.

Drawback

Khi sử dụng Strategy, ta cần lưu ý về việc số lượng class được tạo ra sẽ nhiều hơn cách thông thường. Nên mọi người cần cân nhắc khi sử dụng.

Usages

- Strategy pattern thường được sử dụng khi nhiều class liên quan với nhau nhưng chỉ khác behaviour.
- Được sử dụng khi bạn có những câu lệnh điều kiện lớn và phức tạp, cần được đơn giản hóa.
- Bạn muốn logic và giải thuật của bạn có thể linh động và thay đổi lúc runtime.
- Thường được sử dụng trong Validation.

Template Method Pattern

Problem

Defination

Template Method định nghĩa ra cấu trúc, các bước thực hiện một thuật toán tại một method, gọi tên là template method. Việc implement cụ thệ từng bước đó sẽ nằm trong các Class con. Template pattern cho phép định nghĩa các bước trong một thuật toán mà không làm thay đổi cấu trúc cũng như các bước thực hiện của thuật toán ấy.

Structure

Resolving

Feature

- Giảm số lượng duplicated code (code chung, code lặp lại) của .
- Quản lý tập trung cấu trúc và các steps thực hiện thuật toán (tập trung trong Method Template).

Usages

- Được sử dụng nhiều trong lifecycle của webpage hoặc application.
- Dùng nhiều trong việc định nghĩa framework, workflow.
- Cân nhắc sử dụng khi muốn tách biệt giữa khung sườn, cấu trúc thực thi của thuật toán và hiện thực chi tiết từng bước của thuật toán.
- Dùng để giải quyết common code được viết đi viết lại nhiều lần.
- Dùng để hiện thực kỹ thuật hook.
- Thường được sử dụng kết hợp với strategy pattern

Notes

- Subclass sẽ phụ thuộc nhiều vào cấu trúc và các steps được định nghĩa trong TemplateMethod. Khi phải sửa cấu trúc và các steps, nó sẽ ảnh hưởng đến toàn bộ.
- Hạn chế số lượng các Primitive Method của subclass và interface, càng nhiều Primitive Method thì việc implement càng phức tạp.
- Cấu trúc và các steps thường cố định và không nên khác biệt đối với từng subclass.