CS680: HW19

Explanation:

Decorator design pattern allows the user to add functionalities of the object at the run time without affecting other class instances. And an individual object can get their modified behavior. So in future user can add as many as features easily to the current project structure without doing many modifications.

An example code in which has many conditional statements but not used Decorator Design Pattern:

Calculate Car price based on its different functionalities:

Like Normal car has some amount of price and then after when will you add more other functionalities into that its price will increase. Like,

1)fuel type
2)Car body style
3)Gear type
4)2wheel drive / 4wheel drive

Some basic conditions:

```
if(car has 2wd && body_stye)
{
// return some intVal;
}else if(car has 4wd)_{
// return some intVal;
} else if(car has 4wd ||  gear){
// return some intVal;
}….. may more conditions
```

Inside basic condition some more specific conditions: like this

```
if(fuel=’petrol’ && body_style = ’SUV’ && gear = ”with_gear” && wheel_type=’2wd’)
{
   // return some intVal;}
else if(fuel=diesel && body_style = ’Sedan’ && gear = ”with_gear” && wheel_type=’2wd’){
   // return some intVal;
}
else if(fuel=’petrol’ && body_style = ’Sedan’ && gear = ”with_gear” && wheel_type=’4wd’){
   // return some intVal;
}
```

…. Many more conditions may come.

So in this case we need to write so many conditions that match with the possible user requirement. As a software developer we need to keep in kind cost, processing speed and reusability of code thing. So we have to make our code more reducible and less amount code cost. So in this case we wrote all code which will execute at the compile time.
But instead of doing this if we use Decorator design pattern then the user can add their functionalities at the run time and we eliminate conditional statements code.

Same code but used Decorator design pattern and eliminates conditional statements:

Component interface:

```
Public interface Car
{
   public int getPrice();
}
```

Component implementation:

```
Public class NormalCar implements Car
{
Public int getPrice()
   {
   return 4000;// return some integer value
   }
}
```

Decorator: implement component interface

```
Public class  CarDecorator implemets Car{
Public Car car;
Public CarDecorator(Car c)
{
   this.car = c;
}
Public int getPrice()
   {
         this.car.getPrice();
   }

   }
}
```

Concrete Decorators:

```
Public class FuelCar extends CarDecorator
{
   FuelCar(Car c){
   Super(c);
}
Public int getPrice()
   {
   super.getPrice();
   return 1000;
   // Comment: So this is add Normal car price with current price and return it
   }
}


Public class WheelDriveCar extends CarDecorator
{
   WheelDriveCar (Car c){
   super(c);
}
Public int getPrice()
   {
   super.getPrice();
   return 2000;
   // Comment: So this is add Normal car price with current price and return it
   }
}



Public class CarBodyCar extends CarDecorator
{
   CarBodyCar (Car c){
   super(c);
}
Public int getPrice()
   {
   super.getPrice();
   return 2000;
   // Comment: So this is add Normal car price with current price and return it}
}
```

```
Public class GearCar extends CarDecorator
{
   GearCar (Car c){
   super(c);
}
Public int getPrice()
   {
   super.getPrice();
   return 1500;
   // Comment: So this is add Normal car price with current price and return it
   }
}
```

So like this way in future we can also extend these four concrete class with many more other their specific classes. So as the run time we can pass the instance as per our need and we will get our result. So like this way Decorator design strategy is working.

```
   Call specific model getPrice like this:
   Car GearWhellDriveCar gwdc = new GearCar (new WheelDriveCar(new NormalCar()));
   gwdc.getPrice();
```

Java API designer decided to use Decorator design pattern in java.io because there are many classes inside java.io. They may be able to add new functionalities in current classes. Moreover they have to make code which avoids complexity, avoid dependency, make more transparent code. If they didn't follow this design pattern strategy then they meet the dependency problems, more conditional statements and much more.

Name: Punit Rajendra Patel
Student ID: 01603970
Email: punitpatel26515@gmail.com