ASSIGNMENT 2

Nguyen Minh Quang

GCH18589 GCH0709 – Advanced Programming

Contents

Introduction

This report gives the solution to the scenario that I mentioned in previous report and how I used design pattern to solve it. I will also show you the updated diagram for the scenario and the code implementation of the diagram.

Scenario

Let's recall the scenario in the previous report. The scenario: You are selling electric-bicycle online. When the customers look for their bike, they can add additional accessories for its like add a basket for the bike or different handle bar. You need to program the application to calculate the total cost of their order.

The design pattern I used

From the scenario, the store already has the electric bikes and the accessories as the object that means we don't need to use creational patterns to create new objects. They already have a system for ordering and calculating the cost of the bike and now they also want to use it to order and calculate the cost of the additional accessories that the customer bought. So that means I need to use a design pattern that deal with composition of the objects or in other word, I need to use a structural pattern to program the application.

The first structural pattern that I considered is adapter. The adapter pattern converts the interface of a class into another interface client expect. The adapter lets classes work together that couldn't otherwise because of incompatible interfaces. In this case, it should allow the bike objects and the many different accessories objects to work with each other. In theory, when the interface calls the method to get the order of the customer, the adapter will allow both the name of the bike and the accessories that they bought to appear in a single order by creating an instance of the bike (adapter) and adding functionalities to it.

The second pattern that I considered is decorator. The decorator pattern attaches additional responsibilities to an object dynamically. In other means, this pattern allows a user to add new functionality to an existing object without altering its structure. In this scenario by using the decorator pattern, I will create a decorator class that wraps around the original bike class and provides additional functionalities without changing any class methods of the original bike class.

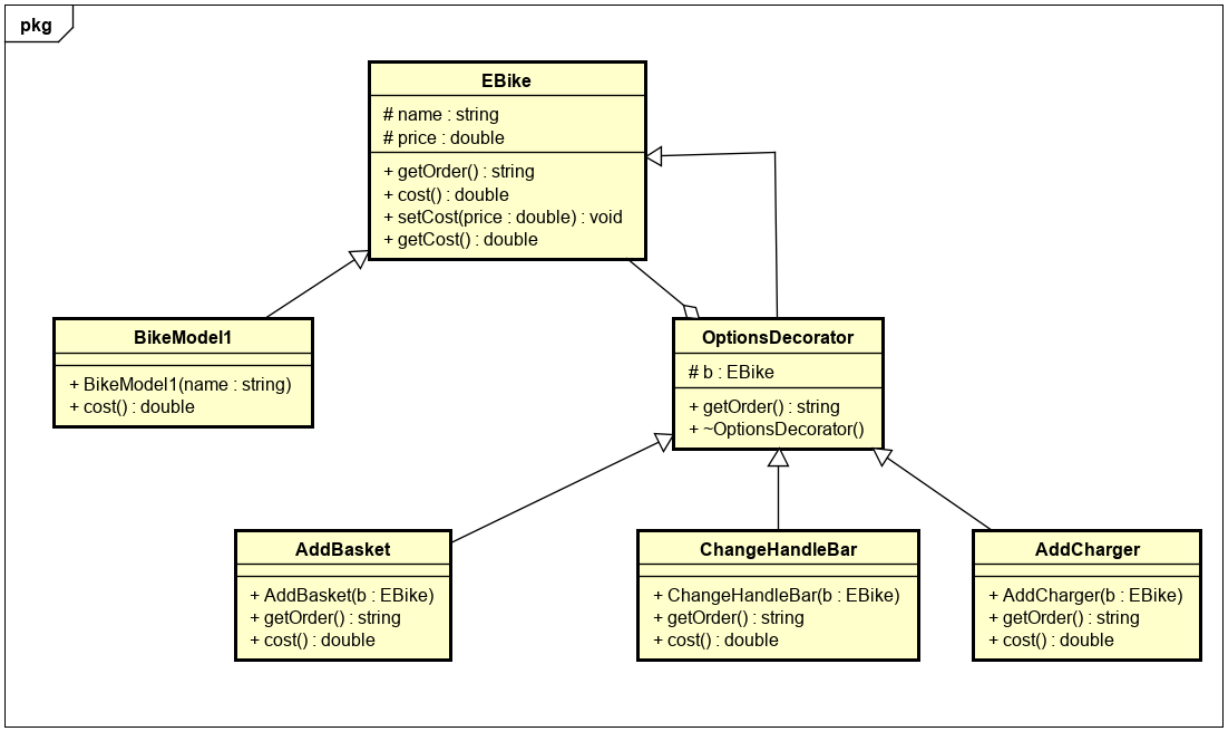
After some thoughts, even though both patterns are quite similar on how they add more function to the original class, I think the decorator pattern is more suitable for this scenario because:

- With multiple accessories work with one original bike class, you need a lot of adapters and that's more coding to do while the decorator only need one decorator class to wrap around the bike class and the accessories only need to run their methods through the decorator.

- The adapter can't add new functionalities without change the original class methods, which is important as we need it original class to calculate the price of the entire order including the bike and one or many accessories that the customers bought. Using the adapter, I will run the risk of miscalculating the cost of the order because the original method that display the cost of the bike and the accessories now only display the price of accessories since the method is overridden by the method of displaying accessories' price.

Class diagram

I have updated the class diagram since the previous report so that it's easier to program it.



Updated Class Diagram

Now the customer can type how much they want for the bike and the accessories and the application will print out their order and the total cost of the order. When customer want to buy a bike, a bike object is created and added to the order and when they type the price that they want for the bike, that price will go to the total cost of the order. The same will happen for the accessories as well, an accessory will be created and added to the order if the customer want to buy it and they will name the price that they want for each accessory they bought.

The code

Now I will implement the code for the application. First, let's create the base class "EBike".

```
#ifndef EBIKE_H
#define EBIKE_H
#include <string>
#include <iostream>
using namespace std;

class EBike //Our Abstract base class
{
        protected:
                string name;
                double price;
        public:

                EBike();
                virtual string getOrder();
                virtual double cost();
                void setCost(double price);
                virtual double getCost();
                virtual ~EBike();

};
```

"EBike" is our abstract class because the "getOrder" and "cost" method will be overridden by the subclass "BikeModel1" to show the order and the cost of the bike. The "setCost" and "getCost" are the methods to get the price that the customers want and put it in the order. And finally, we have the deconstructor of class when we need to delete it. The variable "name" and "price" in the class attributes are the name and price of the bike that the customers want.

Next, it's the subclass "BikeModel1".

```
class BikeModell : public EBike //Where the bike object is created
{
        public:
                BikeModell();
                virtual double cost();
                ~BikeModell();

};

#endif // EBIKE_H
```

When the customers want to buy an electric bike, a bike object is created in the application and the name of the bike is added to the order via “getOrder” method. After that, the customer will give the application the price they want and add it to the cost of the order. From that, this is how I implemented the functions for both of classes mentioned above.

```
#include "EBike.h"

EBike::EBike()//When the order is empty
{
     name = "Unknown Bike";
}
string EBike::getOrder()//get the customer's order
{
    return name;
}
double EBike::cost()//show the price of the order
{
    return price;
}
void EBike::setCost(double price)//get the price that the customer want
{
    this->price=price;
}
```

Now, I will create the decorator class to the accessories to the order.

```
#ifndef OPTIONSDECORATOR_H
#define OPTIONSDECORATOR_H
#include <string>
#include <iostream>
#include "EBike.h"
using namespace std;

class OptionsDecorator : public EBike //Decorator Base class
{
        protected:    EBike *_b;
        public:
                virtual string getOrder()=0;
                virtual ~OptionsDecorator();

};
```

The “getOrder” method will override the same method in the base abstract class to add the accessories that the customers are buying. The variable “*_b” is the order with the bike that the customers are buying, it’s here so that the order and the cost of the accessories won’t overwrite the current order of the bike by overriding the methods.

Next, it’s the accessory options that the customers have.

```
class AddBasket: public OptionsDecorator//where the basket is created and added to the order
{
        public:
                AddBasket(EBike *b);
                string getOrder();
                double cost();
                ~AddBasket();

};

class AddCharger: public OptionsDecorator//where the charger is created and added to the order
{
        public:
                AddCharger(EBike *b);
                string getOrder();
                double cost();
                ~AddCharger();

};
```

```
class ChangeHandleBar: public OptionsDecorator//where the fee of changing the handlebar is created and added to the order
{
       public:
               ChangeHandleBar(EBike *b);
               string getOrder();
               double cost();
               ~ChangeHandleBar();

};

#endif // OPTIONSDECORATOR_H
```

Each time the customer orders an extra accessory for the bike, a new accessory object will be created and added to the order via “getOrder” method and the cost will be added by using the “cost” method. Now, I will show the implementation of all the methods in the decorator and the accessories subclasses.

```
#include "OptionsDecorator.h"
#include <string>
#include <iostream>
using namespace std;
OptionsDecorator::~OptionsDecorator()//deconstructor
{
     cout<<"~OptionsDecorator()\n";
}

AddBasket::AddBasket(EBike *b)//create basket object
{
    _b = b;
}

string AddBasket::getOrder()//add the basket to the order
{
    return _b->getOrder() + ", Bike Basket";
}
double AddBasket::cost()//add the cost of basket to the order
{
    return price + _b->cost();
}
AddBasket::~AddBasket()//deconstrutor
{
    cout << "~AddBasket()\n";
    delete _b;
}
AddCharger::AddCharger(EBike *b)//create charger object
{
    _b = b;
}

string AddCharger::getOrder()//add the charger to the order
{
    return _b->getOrder() + ", Additional charger";
}
```

```
double AddCharger::cost()//add the cost of the charger to the order
{
    return price + _b->cost();
}
AddCharger::~AddCharger()//deconstrutor
{
    cout << "~AddCharger()\n";
    delete _b;
}
ChangeHandleBar::ChangeHandleBar(EBike *b)//create a different handlebar object
{
    _b = b;
}

string ChangeHandleBar::getOrder()//add the handlebar to the order
{
    return _b->getOrder()+ ", Different handlebar";
}

double ChangeHandleBar::cost() //add the fee of changing handlebar to the order
{
    return price + _b->cost();
}
ChangeHandleBar::~ChangeHandleBar()//deconstrutor
{
    cout << "~ChangeHandleBar()\n";
    delete _b;
}
```

Finally, let's program the interface where the customers write the price that they want for their order.

```
#include <string>
#include <iostream>
#include "EBike.h"
#include "OptionsDecorator.h"
using namespace std;


int main()
{
        //Create our Bike that we want to buy
        EBike *bike = new BikeModell();
        double price;//The price we want
        //name the price you want
        cout<< "How much is the bike?? \n";
        cin>>price;
        bike->setCost(price);
        //print out the order
        cout << "Base model of " << bike->getOrder() << " costs $" << bike->cost() << "\n";
        //create a basket that we want to buy
        EBike *basketBike = new AddBasket(bike);
```

```
//tell me your price
cout<< "How much is it for the basket? \n";
cin>>price;
basketBike->setCost(price);
//print out the order
cout << basketBike->getOrder() << " will cost you $" << basketBike->cost() << "\n";
//create a charger that we want to buy
EBike *charger = new AddCharger(basketBike);
//what is the price??
cout<< "How much is it for the charger? \n";
cin>>price;
charger->setCost(price);
//print out the order
cout << charger->getOrder() << " will cost you $" << charger->cost() << "\n";
//I want a different handle bar for my bike
EBike *handlebar = new ChangeHandleBar(charger);
cout<< "How much is it to change the handle bar? \n";
cin>>price;
handlebar->setCost(price);
//print out the order
cout << handlebar->getOrder() << " will cost you $" << handlebar->cost() << "\n";

                                    return 0;
                                }
```

Let's test the application out. For example, I want to buy an electric bike for 500$ and I also want to add a 10$ basket to it, an additional 25$ charger to carry it in the bike in case I need charging and I want change the handlebar for different one for another 20$. I will need to type the cost of each item in my order in the application so that the application will print out my order and the cost of it, which should output 555$.

```
How much is the bike??
500
Base model of BikeModel1 costs $500
How much is it for the basket?
10
BikeModel1, Bike Basket will cost you $510
How much is it for the charger?
25
BikeModel1, Bike Basket, Additional charger will cost you $535
How much is it to change the handle bar?
20
BikeModel1, Bike Basket, Additional charger, Different handlebar will cost you $555

Process returned 0 (0x0)   execution time : 30.132 s
Press any key to continue.
```

As you can see, the application works as intended. It added items in a single order and calculate the total cost of the order.

References

Baeldung.com, 2019. *Baeldung.com.* [Online]
Available at: https://www.baeldung.com/java-structural-design-patterns
[Accessed 13 November 2019].

GeeksforGeeks.org, 2019. *GeeksforGeeks.org.* [Online]
Available at: https://www.geeksforgeeks.org/adapter-pattern/
[Accessed 13 November 2019].

tutorialspoint.com, 2019. *tutorialspoint.com.* [Online]
Available at: https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm
[Accessed 13 November 2019].