report on strategy and decorator patterns

**GROUP 3**

VNUHCM – University of Science

CS202 – Programing Systems

Table of Contents

[I. Strategy pattern: 2](#_Toc152001257)

[1) A real-world problem: 2](#_Toc152001258)

[2) A naive solution: 2](#_Toc152001259)

[3) Problems with the naive solution: 2](#_Toc152001260)

[4) Pattern introduction: 2](#_Toc152001261)

[5) General class diagram: 3](#_Toc152001262)

[6) Class diagram for the problem: 3](#_Toc152001263)

[7) Implementation of the pattern: 3](#_Toc152001264)

[8) Pros and cons of the pattern: 5](#_Toc152001265)

[9) Real-world applications: 5](#_Toc152001266)

[II. Decorator pattern: 6](#_Toc152001267)

[1) A real-world problem: 6](#_Toc152001268)

[2) A naive solution: 6](#_Toc152001269)

[3) Problems with the naive solution: 6](#_Toc152001270)

[4) Pattern introduction: 6](#_Toc152001271)

[5) General class diagram: 6](#_Toc152001272)

[6) Class diagram for the problem: 6](#_Toc152001273)

[7) Implementation of the pattern: 6](#_Toc152001274)

[8) Pros and cons of the pattern: 6](#_Toc152001275)

[9) Real-world applications: 6](#_Toc152001276)

[Reference: 6](#_Toc152001277)

# I. Strategy pattern:

## 1) A real-world problem:

Imagine you are the owner of an express company. At first, you just ship by bus, so you only need to know the cost of one shipping method. At this point, your application for price calculating is very simple. However, as your business grows, you want to add more functions to compute the cost of each method namely by airplane, by cargo ships, by motorbikes and the like. You also want to be able to switch between each method to give the best advice for your customers about the most economical choice for delivering their goods. So what do we do?

## 2) A naive solution:

We can just simply update the method for shipping cost calculation, extending it with conditional statements:

class order{  
public:  
 int getCost(const string& type, const int& distance);  
};

int order::getCost(const string& type, const int& distance){  
 if (type == "bus") return distance\*30;  
 if (type == "car") return distance\*25;  
 if (type == "motorbike") return distance\*10;  
 if (type == "ship") return distance\*40;  
 if (type == "plane") return distance\*70;  
}

order Giang;  
cout << Giang.getCost("ship", 10) << endl;  
cout << Giang.getCost("motorbike", 10) << endl;

## 3) Problems with the naive solution:

The method is not closed to changes from the outside because every time you want to add a new shipping method, you need to modify it. It is also complicated, long and harder to read as your business grows with new types of shipping. It can easily lead to bugs with such vague clarity.

## 4) Pattern introduction:

Then how about turning each algorithm into a separate class? You have one common duty to do is shipping, but you have multiple ways of doing it. The idea is to break them apart and every time you want to add more, you only need to create a new class, not lengthen your already-working method. You will work with them through some “context” class, or an interface.

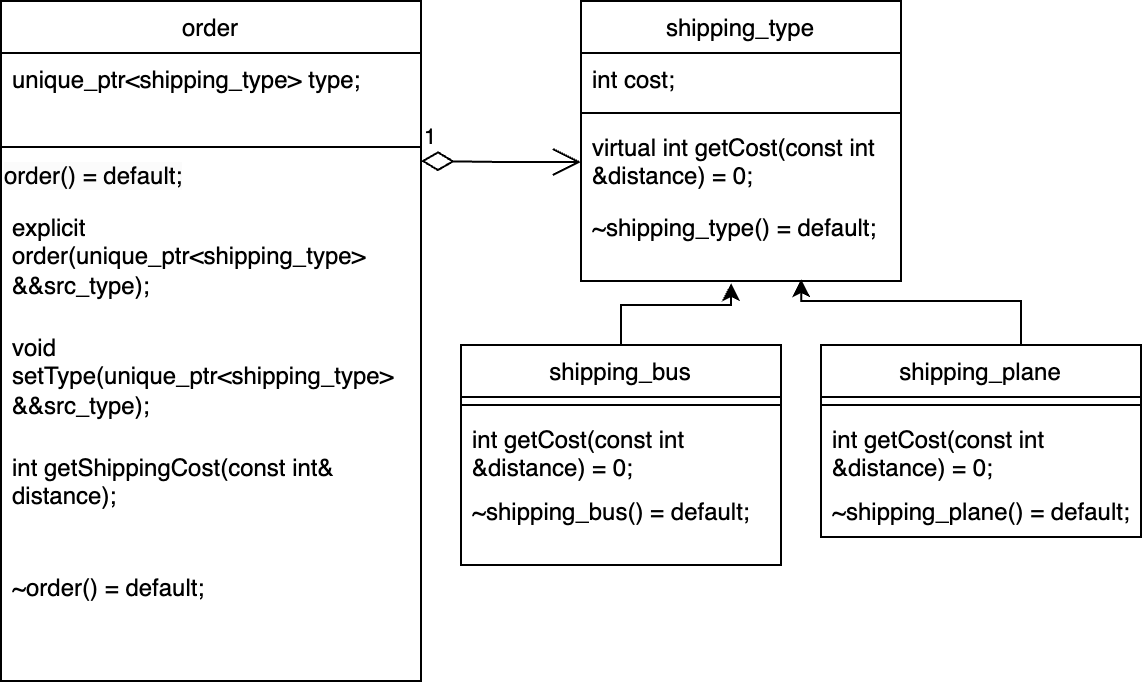
Strategy Pattern is a Class Behavior, that can be changed easily at run time. Strategy Design Pattern is used for creating objects which helps in representing different types of strategies and context objects whose behavior varies. It can also change the executing Algorithm of the Object.

## 5) General class diagram:

A diagram of a strategy

Description automatically generated

## 6) Class diagram for the problem:



## 7) Implementation of the pattern:

- Header file:

class shipping\_type{  
public:  
 virtual int getCost(const int &distance) = 0;  
 ~shipping\_type() = default;  
};  
  
class shipping\_bus : public shipping\_type{  
public:  
 int getCost(const int &distance);  
 ~shipping\_bus() = default;  
};  
  
class shipping\_plane : public shipping\_type{  
public:  
 int getCost(const int &distance);  
 ~shipping\_plane() = default;  
};  
  
class order{  
private:  
 unique\_ptr<shipping\_type> type;  
public:  
 order() = default;  
 explicit order(unique\_ptr<shipping\_type> &&src\_type);;  
 void setType(unique\_ptr<shipping\_type> &&src\_type);  
 int getShippingCost(const int& distance);  
 ~order() = default;  
};

- CPP file:

int shipping\_bus::getCost(const int &distance) {  
 return distance\*30;  
}  
  
int shipping\_plane::getCost(const int &distance) {  
 return distance\*70;  
}  
void order::setType(unique\_ptr<shipping\_type> &&src\_type){  
 this->type = move(src\_type);  
}  
  
int order::getShippingCost(const int& distance){  
 return type->getCost(distance);  
}  
  
order::order(unique\_ptr<shipping\_type> &&src\_type) : type(move(src\_type)){}

- main:

int main() {  
   
 order Giang(make\_unique<shipping\_plane>());  
 cout << Giang.getShippingCost(10) << endl;  
  
 Giang.setType(make\_unique<shipping\_bus>());  
 cout << Giang.getShippingCost(10) << endl;  
 return 0;  
}

## 8) Pros and cons of the pattern:

* Pros:
  + You can swap algorithms used inside an object at runtime.
  + You can isolate the implementation details of an algorithm from the code that uses it.
  + You can replace inheritance with composition.
  + *Open/Closed Principle*. You can introduce new strategies without having to change the context.
* Cons:
  + If you only have a couple of algorithms and they rarely change, there's no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern.
  + If you only have a couple of algorithms and they rarely change, there's no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern.
  + A lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions. Then you could use these functions exactly as you’d have used the strategy objects, but without bloating your code with extra classes and interfaces.

## 9) Real-world applications:

The Strategy Design Pattern can be useful in various scenarios, such as:

* Sorting algorithms: Different sorting algorithms can be encapsulated into separate strategies and passed to an object that needs sorting.
* Validation rules: Different validation rules can be encapsulated into separate strategies and passed to an object that needs validation.
* Text formatting: Different formatting strategies can be encapsulated into separate strategies and passed to an object that needs formatting.
* Database access: Different database access strategies can be encapsulated into separate strategies and passed to an object that needs to access data from different sources.
* Payment strategy: Different payment methods can be encapsulated into separate strategies and passed to an object that needs to process payments.

# II. Decorator pattern:

## 1) A real-world problem:

Jane has just opened a coffee shop that let customers customize their drinks with some given base beverage (espresso, ice blend, oolong tea).

## 2) A naive solution:

## 3) Problems with the naive solution:

## 4) Pattern introduction:

## 5) General class diagram:

## 6) Class diagram for the problem:

## 7) Implementation of the pattern:

## 8) Pros and cons of the pattern:

## 9) Real-world applications:

# Reference: