

Report on Strategy Pattern and Decorator Pattern

GROUP 3



November 5, 2023

VNUHCM – University of Science

Faculty of Information Technology

Table of Contents

[I. Strategy pattern: 2](#_Toc152184913)

[1) A real-world problem: 2](#_Toc152184914)

[2) A naive solution: 2](#_Toc152184915)

[3) Problems with the naive solution: 2](#_Toc152184916)

[4) Pattern introduction: 2](#_Toc152184917)

[5) General class diagram: 3](#_Toc152184918)

[6) Class diagram for the problem: 3](#_Toc152184919)

[7) Implementation of the pattern: 3](#_Toc152184920)

[8) Pros and cons of the pattern: 5](#_Toc152184921)

[9) Real-world applications: 5](#_Toc152184922)

[II. Decorator pattern: 6](#_Toc152184923)

[1) A real-world problem: 6](#_Toc152184924)

[2) A naive solution: 6](#_Toc152184925)

[3) Problems with the naive solution: 7](#_Toc152184926)

[4) Pattern introduction: 7](#_Toc152184927)

[5) General class diagram: 8](#_Toc152184928)

[6) Class diagram for the problem: 9](#_Toc152184929)

[7) Implementation of the pattern: 9](#_Toc152184930)

[8) Pros and cons of the pattern: 12](#_Toc152184931)

[9) Real-world applications: 13](#_Toc152184932)

[Reference: 13](#_Toc152184933)

# 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.

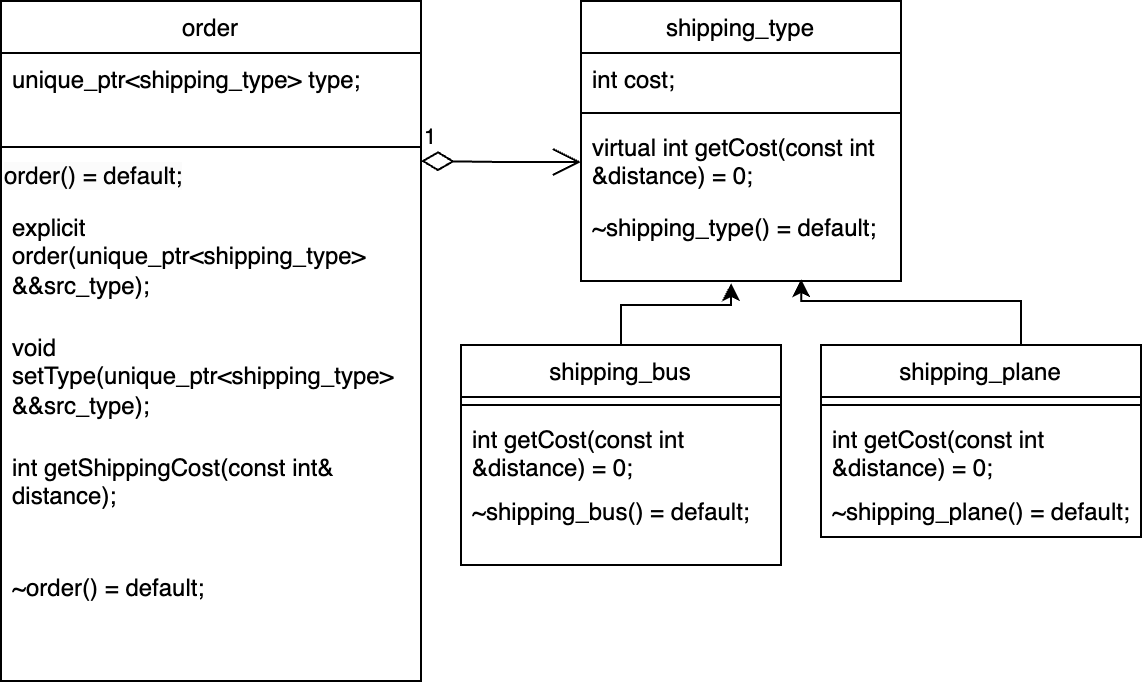
Strategy Pattern is a Class Behavior, that can be changed easily at run time. Strategy Design Pattern is used for creating objects which helps in representing different types of strategies and context objects whose behavior varies. It can also change the executing Algorithm of the Object.

## 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). The list of syrups includes hazelnut, caramel, and vanilla. Customers can also add toppings namely pistachio, sprinkles, boba and aiyu jelly. Jane wants to calculate the price of each order.

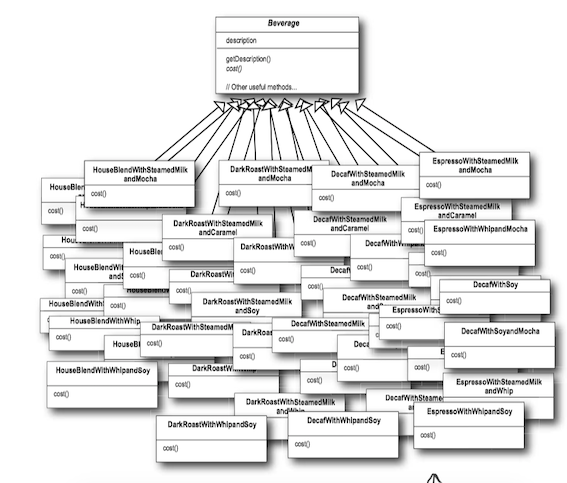
## 2) A naive solution:

Create a base class named beverage and add numerous subclasses such as: espresso with hazelnut syrup and pistachio, espresso with caramel syrup, ice blend with sprinkles and pistachio and so on. And you will specific the price for each drink in each subclass.

Diagram went from this: A diagram of a software application

Description automatically generated with medium confidence

To this:



## 3) Problems with the naive solution:

Jane’s program is now remarkably messy and hard to manage. Every future update will be a real challenge for developers. It is easier to introduce bugs with this style of coding.

## 4) Pattern introduction:

In this case, we should use Decorator Pattern.

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

The main target of the Decorator pattern is to allow the client to add whatever features are required, dynamically, safely and in the easiest possible way.

## 5) General class diagram:

A diagram of a component

Description automatically generated

## 6) Class diagram for the problem:

A diagram of a computer

Description automatically generated

## 7) Implementation of the pattern:

- beverage.h

//  
// Created by Nguyễn Bạch Trường Giang on 29/11/2023.  
//  
  
#ifndef DECORATORPATTERN\_COFFEESHOP\_H  
#define DECORATORPATTERN\_COFFEESHOP\_H  
  
#endif //DECORATORPATTERN\_COFFEESHOP\_H  
#include <iostream>  
#include <string>  
using namespace std;  
  
class beverage\_component {  
private:  
 string description;  
public:  
 virtual string getDescription() = 0;  
 virtual double getCost() = 0;  
 virtual ~beverage\_component() = default;  
};  
  
class espresso : public beverage\_component{  
public:  
 string getDescription() override;  
 double getCost() override;  
};  
  
class oolong\_tea : public beverage\_component{  
public:  
 string getDescription() override;  
 double getCost() override;  
};  
  
class ice\_blend : public beverage\_component{  
public:  
 string getDescription() override;  
 double getCost() override;  
};  
  
class customizations\_decorator : public beverage\_component{  
public:  
 virtual string getDescription() = 0;  
};  
  
class syrup\_vanilla : public customizations\_decorator{  
private:  
 beverage\_component\* beverage;  
public:  
 explicit syrup\_vanilla(beverage\_component\* src);  
 string getDescription() override;  
 double getCost() override;  
};  
  
class syrup\_hazelnut : public customizations\_decorator{  
private:  
 beverage\_component\* beverage;  
public:  
 explicit syrup\_hazelnut(beverage\_component\* src);  
 string getDescription();  
 double getCost();  
};  
  
class pistachio : public customizations\_decorator{  
private:  
 beverage\_component\* beverage;  
public:  
 explicit pistachio(beverage\_component\* src);  
 string getDescription() override;  
 double getCost() override;  
};  
  
class cinnamon : public customizations\_decorator{  
private:  
 beverage\_component\* beverage;  
public:  
 explicit cinnamon(beverage\_component\* src);  
 string getDescription() override;  
 double getCost() override;  
};

- beverage.cpp:

//  
// Created by Nguyễn Bạch Trường Giang on 29/11/2023.  
//  
  
#include "beverage.h"  
  
string espresso::getDescription(){  
 return "Espresso";  
}  
double espresso::getCost(){  
 return 2.5;  
}  
  
string oolong\_tea::getDescription(){  
 return "Oolong tea";  
}  
double oolong\_tea::getCost(){  
 return 3;  
}  
  
string ice\_blend::getDescription(){  
 return "Matcha ice blended";  
}  
  
double ice\_blend::getCost(){  
 return 3.5;  
}

- decorators.cpp:

//  
// Created by Nguyễn Bạch Trường Giang on 29/11/2023.  
//  
  
#include "beverage.h"  
  
syrup\_vanilla::syrup\_vanilla(beverage\_component\* src){  
 beverage = src;  
}  
string syrup\_vanilla::getDescription(){  
 return beverage->getDescription() + " + Vanilla syrup";  
}  
double syrup\_vanilla::getCost(){  
 return beverage->getCost() + 0.5;  
}  
  
syrup\_hazelnut::syrup\_hazelnut(beverage\_component\* src) {  
 beverage = src;  
}  
string syrup\_hazelnut::getDescription(){  
 return beverage->getDescription() + " + Hazelnut syrup";  
}  
double syrup\_hazelnut::getCost(){  
 return beverage->getCost() + (0.7);  
}  
  
pistachio::pistachio(beverage\_component\*src) {  
 beverage = src;  
}  
string pistachio::getDescription(){  
 return beverage->getDescription() + " + Pistachio crumble";  
}  
double pistachio::getCost(){  
 return beverage->getCost() + 1.2;  
}  
  
cinnamon::cinnamon(beverage\_component\* src) {  
 beverage = src;  
}  
string cinnamon::getDescription(){  
 return beverage->getDescription() + " + Cinnamon powder";  
}  
double cinnamon::getCost(){  
 return beverage->getCost() + 1;  
}

- main.cpp:

#include "beverage.h"  
  
int main() {  
 beverage\_component\* giang\_order = new oolong\_tea;  
 giang\_order = new syrup\_vanilla(giang\_order);  
 giang\_order = new pistachio(giang\_order);  
 cout << "Giang's order: " << giang\_order->getDescription() << ": " << giang\_order->getCost() << "$" << endl;  
  
 beverage\_component\* jane\_order = new espresso;  
 jane\_order = new syrup\_hazelnut(jane\_order);  
 jane\_order = new cinnamon(jane\_order);  
 jane\_order = new pistachio(jane\_order);  
 cout << "Jane's order: " << jane\_order->getDescription() << ": " << jane\_order->getCost() << "$" << endl;  
  
 delete giang\_order;  
 delete jane\_order;  
 return 0;  
}

## 8) Pros and cons of the pattern:

* Pros:
  + You can extend an object’s behavior without making a new subclass.
  + You can add or remove responsibilities from an object at runtime.
  + You can combine several behaviors by wrapping an object into multiple decorators.
  + *Single Responsibility Principle*. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.
* Cons:
  + It’s hard to remove a specific wrapper from the wrappers stack.
  + It’s hard to implement a decorator in such a way that its behavior doesn’t depend on the order in the decorators stack.
  + The initial configuration code of layers might look pretty ugly.

## 9) Real-world applications:

- Cases when you should apply Decorator Pattern:

* Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.
* Use the pattern when it’s awkward or not possible to extend an object’s behavior using inheritance.

# Reference:

* (No date a) *3 Decorator Pattern*. Available at: <https://techlarry.github.io/Language/Java/Head%20First%E8%AE%BE%E8%AE%A1%E6%A8%A1%E5%BC%8F/3%20Decorator%20Pattern/> (Accessed: 29 November 2023).
* (No date b) *Top strategy design pattern interview questions (2023)*. Available at: <https://www.codeusingjava.com/interview/strat> (Accessed: 28 November 2023).
* Ahmaditabar, M. (2021a) *Solid principles*. Available at: <https://maj-ahd.medium.com/solid-principles-63df9745ffce> (Accessed: 27 November 2023).
* Ahmaditabar, M. (2021b) *Strategy design pattern*. Available at: <https://www.linkedin.com/pulse/strategy-designpattern-majid-ahmadi/> (Accessed: 27 November 2023).
* Krishna, A. (2023) *A beginner’s guide to the strategy design pattern*. Available at: <https://www.freecodecamp.org/news/a-beginners-guide-to-the-strategy-design-pattern/> (Accessed: 27 November 2023).
* Shvets, A. (no date a) *Decorator*, *Refactoring.Guru*. Available at: <https://refactoring.guru/design-patterns/decorator> (Accessed: 29 November 2023).
* *Strategy* (no date) *Refactoring.Guru*. Available at: <https://refactoring.guru/design-patterns/strategy> (Accessed: 27 November 2023).