**December 10**

Group 11: Outliers

Nguyễn Hoàng Bảo Minh - 18125100

Trương Công Anh - 18125047

STRATEGY &

DECORATOR

PATTERN



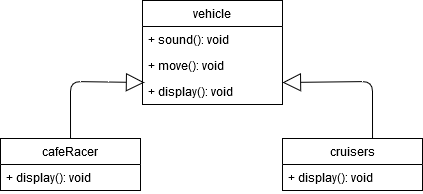
STRATEGY PATTERN

# Problem

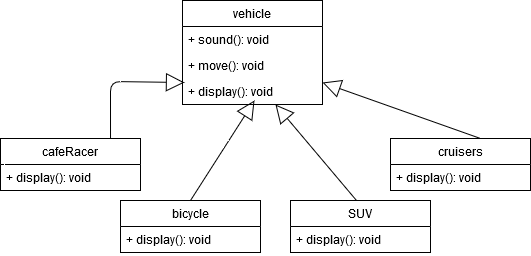
You want to build a vehicle simulator, it can demonstrate many kinds of vehicle. Specifically, how the vehicle move and how its engine sound.

* Using OO design techniques create one super abstract class vehicle, and other types of vehicle inherit from it.

Class Diagram:



However, when the simulator has more kinds of vehicle such as: SUV, bicycle,etc…



These vehicle don’t have the same move and sound behaviors. For example: bicycles make no sound, and SUV moves with 4 wheels, not 2 wheels like others.

With this approach, you have to overridden move() and sound() functions for each types of vehicle.

* The disadvantages of of this approach is that it is hard to gain knowledge of all vehicle behaviors, and its changes unintentionally affect the other vehicle.

# Strategy Pattern

**Definition and Intent:**

*The strategy pattern defines a family of algorithms, encapsulates each one, and make them interchangeable. Strategy let the algorithms vary independently from users that use it.*

*Capture the abstraction in an interface, bury implementation details in derived classes.*

There’s a design principle applied by Strategy Pattern.

**Design Principle:** *Identify the aspects of the application that vary and separate them from what stays the same.*

It means picking what varies and encapsulate it, so these parts can be easily extended or altered without affect other parts of the code.

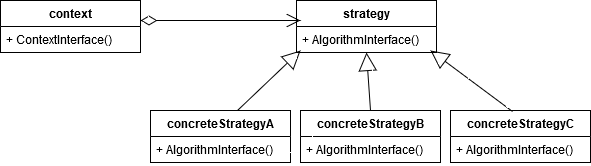
* Strategy Pattern applies the design principle mentioned above. And by using this pattern, the difficulty when desigining the vehicle simulator can be solved.

**Design Principle:** *Program to an interface, not an implementation.*

Apply this principle, an interface can be used to represent each behavior of vehicle class, move behavior and sound behavior.

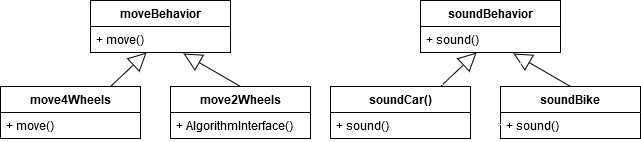
So each implementation of behavior will implement one of those interfaces.

# Structure



* context:
  + Has strategy class as an object.
  + Maintains the reference to each concreteStrategy object.
* strategy: declares interface to support all algorithms build in concreteStrategy, and Context use this interface to call the algorithms defined by concreteStrategy.
* concreteStrategy: implements the algorithm using strategy interface.

# Strategy Pattern solves the problem

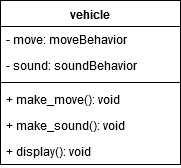


*Class Diagram moveBehavior and soundBehavior*

Vehicle behaviors are taken out from vehicle class, which means those behaviors will work independently. Therefore, vehicle class does not need to know any of implementation details for their own behaviors.

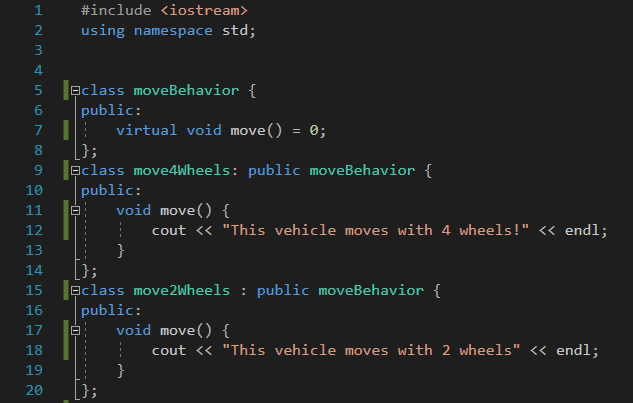
moveBehavior and soundBehavior are interfaces. All new move and sound classes just need to implement the move() and sound() method.

With this design, other types of object can reuse move and sound behavior. Additionally, it is easier to add new behaviors without modifying existing behaviors.

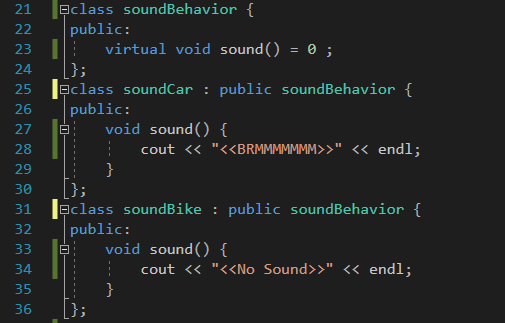


Variables move and sound are instance variables which hold reference to a specific behavior at run time. Method make\_move() and make\_sound() replace for move() and sound().

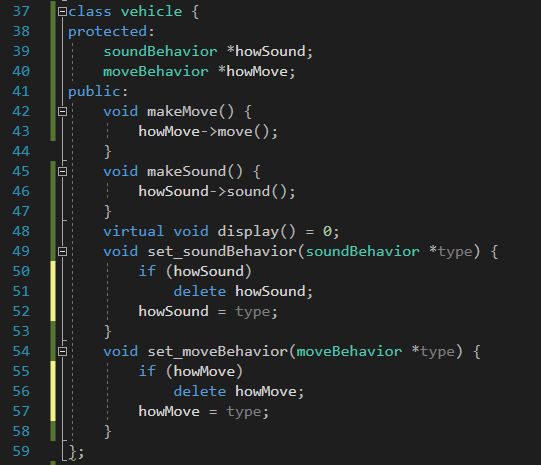
# Code



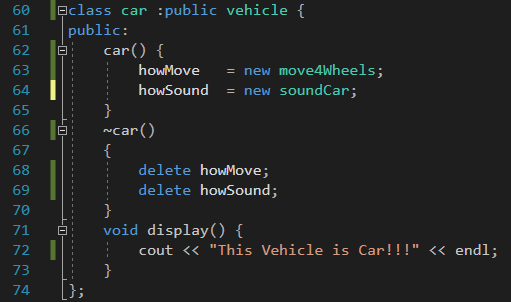
*Move Behavior*



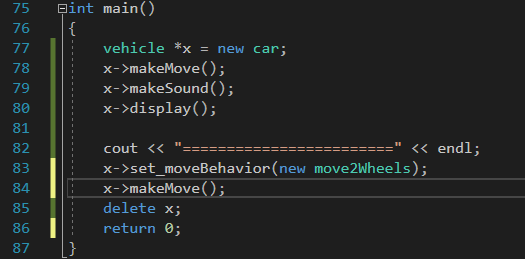
*Sound Behavior*

**

*Vehicle Class*

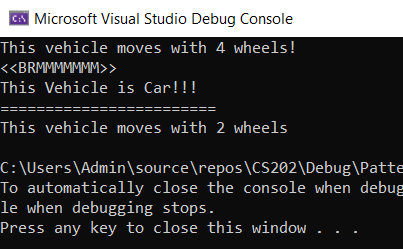
**

*Car class*

**

*Main*

**Console prints out:**

****

# Advantages and Disadvantages

## Pros:

 Families of related algorithms. Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse. Inheritance can help factor out common functionality of the algorithms.

 An alternative to subclassing. Inheritance offers another way to support a variety of algorithms or behaviors. Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.

 Strategies eliminate conditional statements. The Strategy pattern offers an alternative to conditional statements for selecting desired behavior.

 A choice of implementations. Strategies can provide different implementations of the same behavior. The client can choose among strategies with different time and space trade-offs

## Cons:

 Clients must be aware of different Strategies. Users must understand how Strategies differ before choosing the appropriate one. In addition, clients might be exposed to implementation problems. Therefore use the Strategy pattern only when the variation in behavior is relevant to users.

 Communication overhead between Strategy and Context. The Strategy interface is shared by all ConcreteStrategy classes whether the algorithms they implement are trivial or complex. Hence there’s a drawback that some ConcreteStrategies won't use all the information inherited from Strategy class. That means there will be times when the context creates and initializes parameters that never be used.

 Increased number of objects. Strategies increase the number of objects in an application.

# Other Problem Solved By Strategy Pattern

1. Customer comes to the cashier
2. System accepts customer details
3. System calculates bill amount
4. System apply discount based on day of the week (This is where the strategy pattern comes into the picture)
   * Monday – 10 %
   * Friday – 50 %
5. System output the total bill for the customer

System has to select which discount option assign to the bill based on the day of the week

* *Problem* – System has to apply discounts to bills
* *Solution* – There are 3 options for discounts based on days – system has to select one and calculate the bill amount

# References

http://java.boot.by/scea5-guide/ch07s03.html

https://www.javagists.com/strategy-design-pattern

Head First Design Pattern

Design Patterns: Elements of Reuseable Object-Oriented Software.

DECORATOR PATTERN

# Problem:

Consider an Ice cream store sell ice creams cone with two flavors: Chocolate, Vanilla . A waffle cone has the price of 1000, while an chocolate ice cream ball and vanilla ice cream ball has the same price of 5000.The store want to print ice creams description and price of some of the combinations of flavors of ice cream.

# Approach Solution:

Use inheritance.

# Disadvantage of this solution:

In this problem using inheritance to solve, for every combination of flavors we need to create a class. Ex:

+ JustCone//Cone without ice cream

+ Choco: public JustCone//Ice cream with chocolate flavor

+Vanilla: public JustCone//Ice cream with vanilla flavor

+ ChocoVani: public Choco, Vanilla//Ice cream with two flavors: chocolate and vanilla

+DoubleChoco: public Choco // Ice cream with two chocolates ice cream ball: chocolate

+….

-> It makes the code very complicated when the numbers of flavors ice cream ball and waffle cone types become big.

- There’s another solution is Decorator Design Pattern.

# Decorator Design Pattern

**Definition:** In object-oriented programming, the decorator pattern is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class.

**+Overview:**

The Decorator design pattern is one of the twenty – three well known GoF design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, object that are implement, change, text and reuse.

*What problems can it solve?*

- Responsibilities should be added to an object dynamically at run-time.

- A flexible alternative to subclassing for extending functionality should be provided.

# Structure:

**A screenshot of a cell phone

Description automatically generated**

The decorator pattern can be used to extend (decorate) the functionality of a certain object statically, or in some cases at run-time, independently of other instances of the same class, provided some groundwork is done at design time. This is achieved by designing a new *Decorator* class that wraps the original class. This wrapping could be achieved by the following sequence of steps:

* Subclass the original *Component* class into a *Decorator* class (see UML diagram);
* In the *Decorator* class, add a *Component* pointer as a field;
* In the *Decorator* class, pass a *Component* to the *Decorator* constructor to initialize the *Component* pointer;
* In the *Decorator* class, forward all *Component* methods to the *Component* pointer; and
* In the ConcreteDecorator class, override any *Component* method(s) whose behavior needs to be modified.

# Decorator Pattern solves the problem

//Class diagram

A screenshot of a cell phone

Description automatically generated

# Code:

class BasCream {//interface

public:

virtual ~BasCream() {}

virtual int getPrice() = 0;

virtual std::string getDescription() = 0;

};

class JustCone : public BasCream {

public:

~JustCone(){}

int getPrice(){

return 1000;

}

std::string getDescription() {

return"Waffle Cone ";

}

};

class Flavour :public BasCream{

protected:

BasCream\*Icecream;

public:

Flavour(BasCream\*newIcecream) {

Icecream = newIcecream;

}

std::string getDescription() {

return Icecream->getDescription();

}

int getPrice()

{

return Icecream->getPrice();

}

~Flavour() {

delete Icecream;

}

};

class Choco :public Flavour {

public:

Choco(BasCream\* newIcecream):Flavour(newIcecream) {

}

std::string getDescription() {

return Icecream->getDescription() + ", Chocolate";

}

int getPrice()

{

return Icecream->getPrice() + 4000;

}

};

class Vanilla :public Flavour {

public:

Vanilla(BasCream\* newIcecream) :Flavour(newIcecream) {

}

std::string getDescription() {

return Icecream->getDescription() + ", Vanilla";

}

int getPrice()

{

return Icecream->getPrice() + 4000;

}

};

# Advantages and Disadvantages of Decorator Pattern

**+Advantages:**

- Decorators provide a flexible alternative to subclassing for extending functionality.

- Decorators allow behavior modification at runtime rather than going back into existing code and making changes.

- Decorators are a nice solution for permutation because you can wrap a component with any numbers of decorators.

- The decorator pattern supports the principle that classes should be open for extension but closed for modification.

**+Disadvantages:**

- Decorator can result in many small objects in our design, and overuse can be complex.

- Decorator can cause issues if the client relies heavily on the components concreate type.

- Decorator can complicate the process of instantiating the component because you not only have to instantiate the component but wrap it in a number of decorators.

- It can be complicated to have decorators keep track of other decorators because to look back into multiple layers of the decorator chain starts to push the decorator pattern beyond its true intent.

# Decorator Pattern in Real World.

- The java.io package is largely based on Decorator.

- Here is an example using Decorator: InputStream class.

A screenshot of a cell phone

Description automatically generated

- The output streams have the same design.

- The Reader/Writer streams designs are similar to the above design.

## ***Reference:***

-https://neillmorgan.wordpress.com/2010/02/07/decorator-pattern-pros-and-cons/

-Head First : Design Pattern by Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra

https://www.youtube.com/watch?v=j40kRwSm4VE&list=LLWFZ9lF\_3nl8P0Gl70s4wAw&index=7&t=57s