Group 3

STRATEGY PATTERN

Phạm Trần Hiền Dung 18125074

Nguyễn Thị Bích Vân 18125032

Nguyễn Trần Minh Khuê 18125092

Võ Minh Nhân 18125136

Strategy pattern

Observation

In computer programming, the strategy pattern (also known as the policy pattern) is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.

Strategy lets the algorithm vary independently from clients that use it. Strategy is one of the patterns included in the influential book Design Patterns by Gamma et al. that popularized the concept of using design patterns to describe how to design flexible and reusable object-oriented software. Deferring the decision about which algorithm to use until runtime allows the calling code to be more flexible and reusable.

For instance, a class that performs validation on incoming data may use the strategy pattern to select a validation algorithm depending on the type of data, the source of the data, user choice, or other discriminating factors. These factors are not known until run-time and may require radically different validation to be performed. The validation algorithms (strategies), encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication.

Typically the strategy pattern stores a reference to some code in a data structure and retrieves it. This can be achieved by mechanisms such as the native function pointer, the first-class function, classes or class instances in object-oriented programming languages, or accessing the language implementation's internal storage of code via reflection.

Definition

- The strategy pattern defines a family of algorithms, encapsulates each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.
- Encapsulated separately from the validating objects, it may be used by other validating objects in different areas of the systems without code duplication.

Usage

Strategy Pattern should be used when you begin to notice recurring algorithms but in different variations. This way, you need to separate the algorithms into classes and feed them based on want in your program.

Next, if you notice recurring conditional statements around a related algorithm.

When most of your classes have related behaviors. It will be time to move them into classes.

Advantages

- Reduces long lists of conditionals.
- Avoid duplicate code. Keep class changes from forcing other class changes (reuse/interchange algorithms/behaviors) depends on the context

- Encapsulating the algorithm make it easier to switch, understand and extend.
- Encapsulate/hide complicate/secret code from users
- The client can choose among strategies with different implementations.
- Separation of Concerns: Related behaviors and algorithms are separated into classes and strategies.
- Easy maintainability and refactoring.
- Choice of algorithms to use.

Disadvantages

- Increased number of objects/classes.
- Change the structure of program easily to confuse if you don’t know the structure well.
- A client must understand how Strategies differ before it can select the appropriate one.
- Communication overhead between Strategy and Context. There will be times when the context creates and initializes parameters that never get used.

Example

Let’s take the sorting algorithms we have for example. Sorting algorithms have a set of rule specific to each other they follow to effectively sort an array of numbers. We have the

Bubble Sort

Linear Search

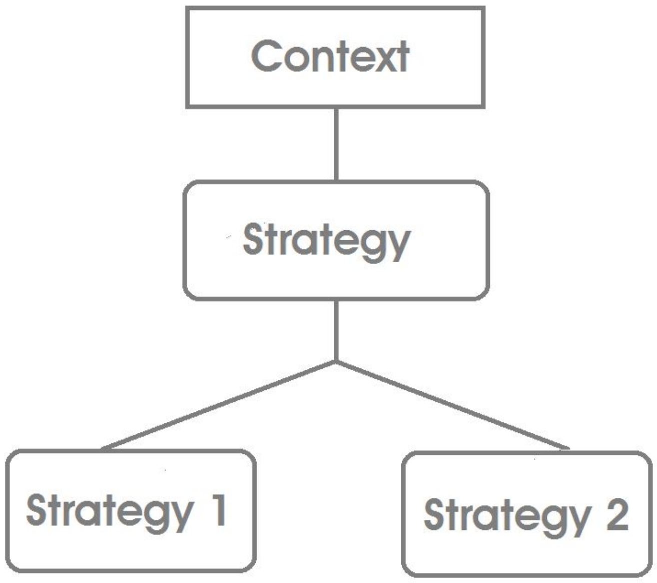
Heap Sort

Merge Sort

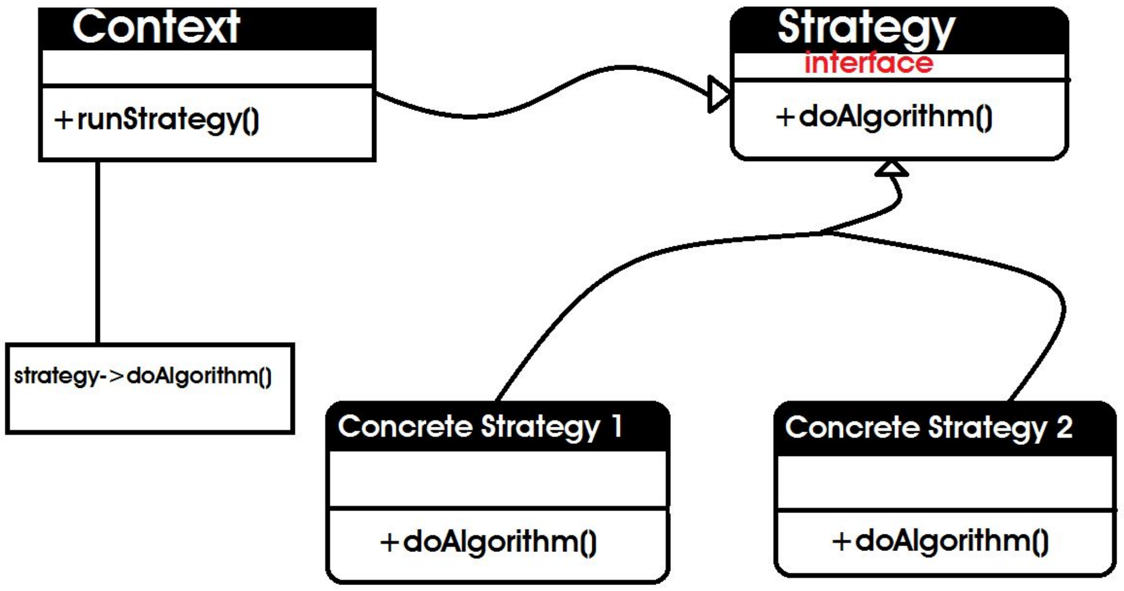
Selection Sort

to name a few.

Structure (Diagram)



In the figure above, the Context class depends on the Strategy. During execution or runtime, different strategies of Strategy type are passed to the Context class. The Strategy provides the template by which the strategies must abide by for implementation.



In the above UML class diagram, the Concrete class depends on an abstraction, Strategy interface. It doesn’t implement the algorithm directly. The Context from its method runStraegy calls the doAlgorithm in the Strategy concretion passed to it. The Context class is independent of the method and doesn't know and doesn't need to know how the doAlgorithm method is implemented. By virtue of Design by Contract, the class implementing the Strategy interface must implement the doAlgorithm method.

In strategy design pattern, there are three main entities: Context, Strategy, and ConcreteStrategy.

The Context is the body composing the concrete strategies where they play out their roles.

Strategy is the template that defines how all startegies must be configured.

ConcreteStrategy is the implementation of the Strategy template(interface).

Source code

```
//.h file code:

#include <string>
#include <vector>
#include <iostream>

class IBillingStrategy;

class StrategyPatternWiki
{
public:
	static void
Main(std::vector<std::wstring>
&args);
};


class Customer
{
private:
	IBillingStrategy *Strategy;

	std::vector<double> drinks;

	// Get/Set Strategy
public:
	IBillingStrategy *getStrategy()
const;
	void
setStrategy(IBillingStrategy *value);

	Customer(IBillingStrategy
*strategy);

	void Add(double price, int
quantity);


	// Payment of bill
	void PrintBill();
};
```

```
class IBillingStrategy
{
public:
	virtual double
GetActPrice(double rawPrice) = 0;
};

// Normal billing strategy (unchanged
price)
class NormalStrategy : public
IBillingStrategy
{
public:
	double GetActPrice(double
rawPrice) override;
};

// Strategy for Happy hour (50%
discount)
class HappyHourStrategy : public
IBillingStrategy
{
public:
	double GetActPrice(double
rawPrice) override;
};

//.cpp file code:

#include "snippet.h"

void
StrategyPatternWiki::Main(std::vector
<std::wstring> &args)
{
	// Prepare strategies
	auto normalStrategy = new
NormalStrategy();
	auto happyHourStrategy = new
HappyHourStrategy();

	auto firstCustomer = new
Customer(normalStrategy);

	// Normal billing
	firstCustomer->Add(1.0, 1);
```

```
        // Start Happy Hour

firstCustomer->setStrategy(happyHo
urStrategy);
        firstCustomer->Add(1.0, 2);

        // New Customer
        Customer *secondCustomer =
new Customer(happyHourStrategy);
        secondCustomer->Add(0.8, 1);
        // The Customer pays
        firstCustomer->PrintBill();

        // End Happy Hour

secondCustomer->setStrategy(norm
alStrategy);
        secondCustomer->Add(1.3, 2);
        secondCustomer->Add(2.5, 1);
        secondCustomer->PrintBill();

        delete secondCustomer;
        delete firstCustomer;
}

IBillingStrategy
*Customer::getStrategy() const
{
        return Strategy;
}

void
Customer::setStrategy(IBillingStrateg
y *value)
{
        Strategy = value;
}

Customer::Customer(IBillingStrategy
*strategy)
{
        this->drinks =
std::vector<double>();
        this->setStrategy(strategy);
}
```

```
void Customer::Add(double price, int
quantity)
{

this->drinks.push_back(this->getStrat
egy()->GetActPrice(price * quantity));
}

void Customer::PrintBill()
{
        double sum = 0;
        for (auto drinkCost :
*this->drinks)
        {
                sum += drinkCost;
        }
        std::wcout << L"Total due:
{sum}." << std::endl;
        this->drinks.clear();
}


double
NormalStrategy::GetActPrice(double
rawPrice)
{
        return rawPrice;
}


double
HappyHourStrategy::GetActPrice(do
uble rawPrice)
{
        return rawPrice * 0.5;
}
```

Exercises

Ex01. Car Wash program

You know car wash can run on different grades of washing and cleaning depending on the money the driver has, the more the money the higher the wash level. Let's the Car Wash offers:

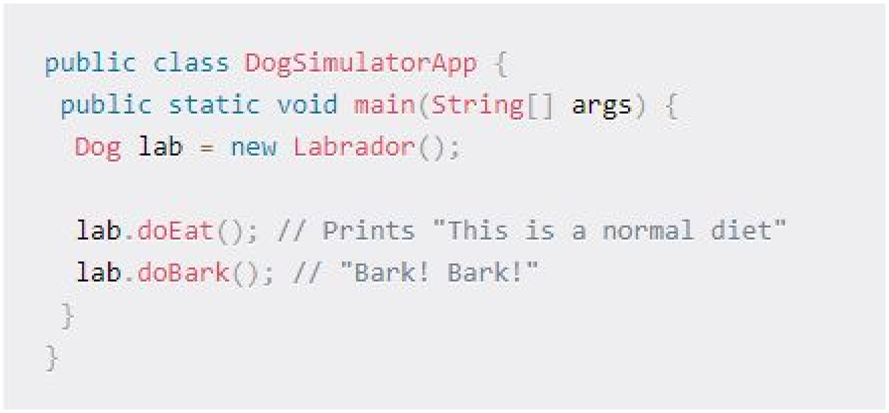
- Basic Wheel and Body washing
- Executive Wheel and Body washing

The Basic wheel and Body cleaning is just the normal soaping and rinsing for the body and brushing for the car.

Executive cleaning goes beyond that, they wax the body and the wheel to make it look shiny and then dry them. The cleaning depends on the level the driver pays for. level 1 gives you Basic cleaning for both body and wheels:

Ex02. Dog simulator program

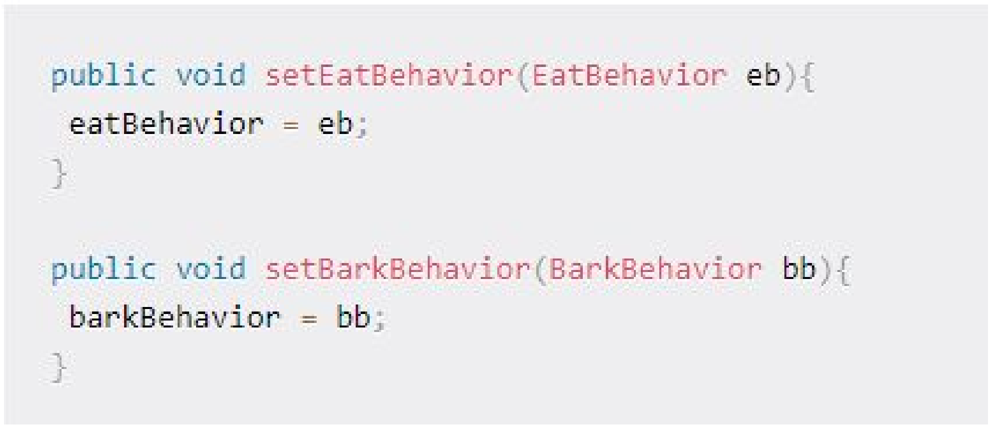
You want to create a dog simulator program to track and know what your dog eats and how it barks



```
public class DogSimulatorApp {
 public static void main(String[] args) {
  Dog lab = new Labrador();

  lab.doEat(); // Prints "This is a normal diet"
  lab.doBark(); // "Bark! Bark!"
 }
}
```

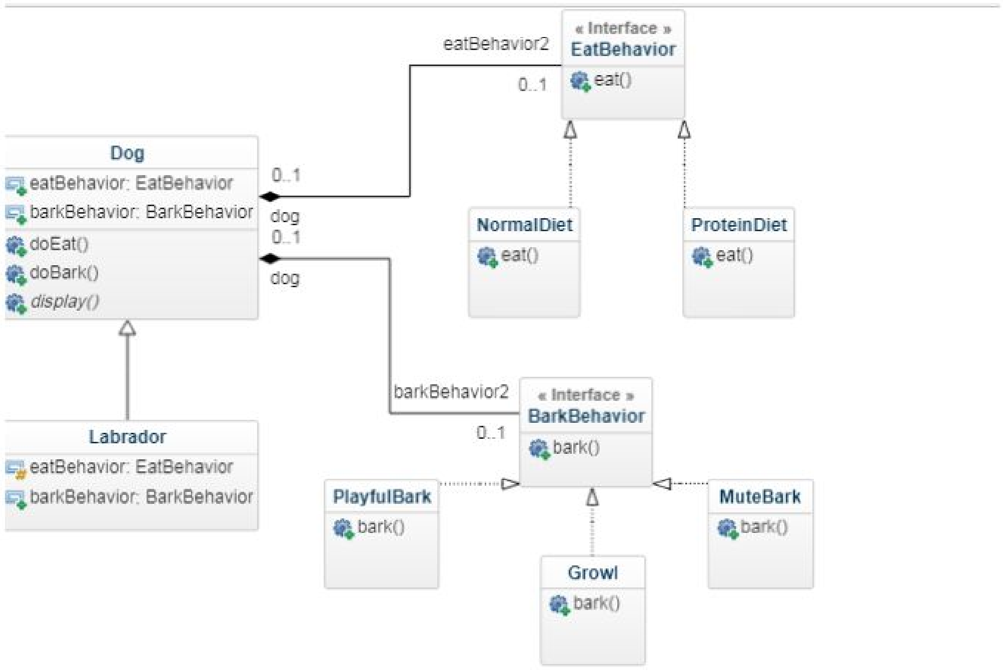
To make this program better, we need to add flexibility. Let’s add setter methods on the dog class to be able to swap at the run time



```
public void setEatBehavior(EatBehavior eb){
 eatBehavior = eb;
}

public void setBarkBehavior(BarkBehavior bb){
 barkBehavior = bb;
}
```

We have the Dog superclass and the ‘Labrador’ class which is a subclass of Dog. Then we have the family of algorithms (Behaviors) “encapsulated” with their respective behavior types.



Class Diagram

Ex 03: A Strategy defines a set of algorithms that can be used interchangeably. Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on trade-offs between cost, convenience, and time.

Ex04: Let’s taking a snap with mobile phone (Here we have iphone7 and Samsung Keystone). Write a program to take a picture with either front or rear camera using one of the above mobile phone.

```cpp
class MobilePhone
{
public:
        virtual void display() = 0;
        void takeRearCamera(){
               frontCameraBehavior->useFrontCamera();
        }
        void takeFrontCamera(){
               rearCameraBehavior->useRearCamera();
        }
        MobilePhone(RearCameraBehavior* rcb, FrontCameraBehavior* fcb):
        rearCameraBehavior(rcb),frontCameraBehvior(fcb){}
private:
        RearCameraBehavior * rearCameraBehavior;
        FrontCameraBehavior * frontCameraBehavior;
};
```

```cpp
class FrontCameraBehavior
{
public:
     virtual void useFrontCamera() = 0;
};
class RearCameraBehavior
{
public:
     virtual void useRearCamera() = 0;
};
```

```cpp
class Iphone7 :public MobilePhone
{
public:
	void display(){
		cout << "I am an Iphone7." << endl;
	}
	Iphone7():MobilePhone(new RearCamera(),new FrontCamera()){}
};
class SamsungKeystone :public MobilePhone
{
public:
	void display(){
		cout << "I am a samsung keystone." << endl;
	}
	SamsungKeystone():MobilePhone(new RearNoWay(),new FrontNoWay()){}
};
```

```cpp
class RearCamera :public RearCameraBehavior
{
public:
	void useRearCamera(){
		cout << "I'm taking a photo with rear camera." << endl;
	}
	RearCamera();
};
class RearNoWay :public RearCameraBehavior
{
public:
	void useRearCamera(){
		cout << "I don't have a rear camera." << endl;
	}
	RearNoWay();
};
```

```
class FrontCamera :public FrontCameraBehavior
{
public:
       void useFrontCamera(){
              cout << "I'm taking a selfie." << endl;
       }
       FrontCamera();
};
class FrontNoWay :public FrontCameraBehavior
{
public:
       void useFrontCamera(){
              cout << "I don't have a front camera." << endl;
       }
       FrontNoWay();
};

void main()
{
     MobilePhone* iphone7 = new Iphone7();
     iphone7->takeFrontCamera();
     iphone7->takeRearCamera();
}
```

References:

https://blog.bitsrc.io/keep-it-simple-with-the-strategy-design-pattern-c36a14c985e9

https://en.wikipedia.org/wiki/Strategy_pattern