ECE 452/ CS 446 D4 Design Activity

Strategy Pattern

Team Goose:

Anurag Joshi (a35joshi)
Daniel Weisberg (dweisber)
Gunin Khanna (g3khanna)
Xinjue Lu (x89lu)

Purpose/Motivation

The strategy pattern is a behavioral software design pattern that allows selection of a specific algorithm at runtime. The basic idea here is that instead of duplicating code and writing the same algorithm all over the place, the code implements a specific strategy based on each client and based on this strategy, it executes one algorithm from a family of algorithms. Here, an algorithm refers to any function that has been implemented within the code. So as our professor mentioned the other day if you are writing code for a class called Animals and if you want to implement a method called "Flying" for birds specifically, you don't want to duplicate the same code for all other non-flying animals such as cats or dogs. Code which uses this pattern often makes each algorithm within the family of algorithms **interchangeable** with one another.

The **purpose** of the strategy pattern is to make the task of adding extra features and functionality to existing code very easy and convenient. This is done while ensuring that there is minimum modification done to the current code. The strategy pattern also allows the code to interchangeably pick and choose specific behaviors at **runtime** based on the pre-existing requirements. This pattern also reduces coupling between an algorithm and a client tied to it by making the algorithm implementation independent of the client using it. This is evident from the UML diagram where the client who further calls the context object isn't linked to the concrete strategies at all.

Any typical implementation of the strategy pattern consists of a **strategy** interface and some **concrete** strategy classes that implement this strategy interface. We also have **context** objects whose behavior changes as per their strategy objects. The strategy object changes the executing algorithm of the context object.

The **motivation** behind using the strategy pattern is that it supports the "open/close" principle which states that code should be open for extension but closed for modification. Similarly, in the strategy pattern, it is very easy to extend and add extra strategies without modifying the strategy interface. This strategy helps us switch between different algorithms based on the needs of the client calling it. Lastly, we use the strategy pattern because different clients share different strategies and they can all use each other's strategies as appropriate. This reduces code duplication and promotes reuse of code.

Vocabulary

1. Strategy Interface

An interface that provides a link between a concrete class and a context object. The interface usually consists of definitions of functions which are common to many concrete class objects.

2. Concrete Strategy classes

These classes implement the strategy interface and provide an actual implementation to the algorithm in question. These implementations are very likely to differ from one another in a unique way.

3. Context objects

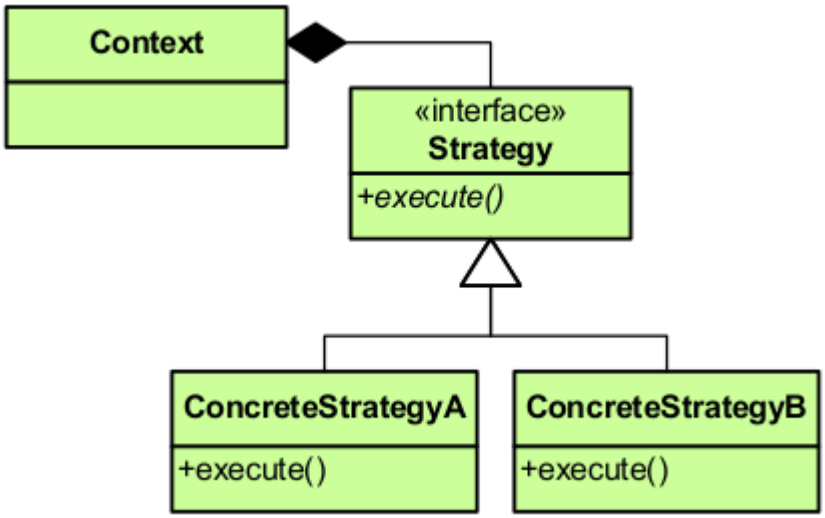
The context objects provide a link between the client code or function and the concrete strategy class.

Intended Use Case:

It is recommended to use the strategy pattern when we have a large collection of algorithms to choose from. That is, we expect that the number of available options for the choice of algorithms will scale up exponentially in the near future. One would also want to use the strategy pattern when a requirement is to change the behavior of objects dynamically at runtime depending on what type of object is being initialized.

Let us consider an example. Imagine that you are going to Subway to get your favorite Subway sandwich. When you get there, the cashier first asks you if you would like a 6 inch sandwich or a footlong sandwich. Let's say you want a 6 inch sandwich. The cashier then picks the algorithm or the function to pick up the 6 inch bread instead of the foot long bread. Here each size of bread is a concrete strategy and you are the client.
Precisely, what's happening here is that the **strategy interface(the cashier)** implements an **algorithm(a size of bread)** based on the **client**'s needs at **runtime**. This example can be applied to several other scenarios as well in this context. For example, the act of choosing which vegetables you want to top up your sandwich, what sauce or which type of meat you prefer follows the strategy pattern as well.

UML Structure



Real Life Example (to present to the class)

Let us consider a situation where we are required to compute the result of the following polynomial expression: **(4*x) + (3*y)**

This is how the Strategy interface would look.
Roles:
Robin: Context, Anurag: Strategy interface, Daniel(concrete strategy): Multiply, Gunin(concrete strategy): Add

```
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

The concrete strategy classes for the Add and Multiply operations are shown below

```
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}

public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

The context object would be declared the following way

```
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

The main would be executed the following way and it should print out the result of the polynomial expression.

```
public static void main(String[] args) {
  int x = args[0];
  int y = args[1];

  Context context = new Context(new OperationMultiply());
  int temp1 = context.executeStrategy(4, x));

  context = new Context(new OperationMultiply());
  int temp2 = context.executeStrategy(3,y));

  context = new Context(new OperationAdd());
  System.out.println(context.executeStrategy(temp1, temp2));
}
```

Advantages (Positive Consequences)

It is easier to modify an algorithm as each of the different algorithms are distinct from each other. Changing one algorithm would not affect any of the other algorithms.

Makes the code significantly more readable by removing clumsy ‘if’ ‘else’ statements for each possible scenario.

Makes the code more versatile and easier to add new implementations for it.

Deferring the decision about which algorithm to use until runtime allows the calling code to be more flexible and reusable.

Disadvantage (Negative Consequences)

Since each strategy requires us to create new classes, we end up with one class for each algorithm which exponentially increases the number of objects defined in code. This may negatively impact performance and cause high memory usage.

Another issue is that the clients are required to know about the algorithms in advance. For example, they should know what all operations are supported such as add, multiply or divide in order to be able to supply a meaningful input.

Relationship with Non-Functional Properties

1 Reusability (improve)
1 Algorithms can easily be reused. The specific functions applied by specific team members won’t be affected by the presence of others.
2 It used composition over inheritance, which reuses code by containing instances of other classes that implement the desired functionality

2 Extensibility (improve)
We can easily add new algorithms to an existing family of algorithms. Other algorithms in this family are not affected by the addition of a new algorithm. The algorithms can then be used interchangeably to alter application behavior without changing its architecture.

3 Scalability and Maintainability (improve):
It makes it easier to extend and incorporate new behavior into the different algorithms without changing the core application logic.

4 Security (improve):
Security problems in one algorithm will only affect clients that use this algorithm. Other clients remain unaffected and can proceed as usual.

5 Performance (degrade):
Each strategy requires us to create new classes which exponentially increases the number of objects defined in code. This negatively impacts application performance.