Strategy pattern

From Wikipedia, the free encyclopedia

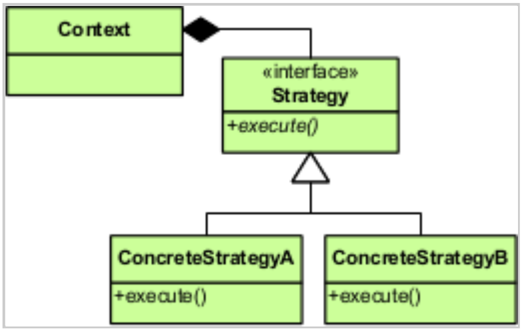
In computer programming, the **strategy pattern** (also known as the **policy pattern**) is a software design pattern, whereby an algorithm's behaviour can be selected at runtime. Formally speaking, the strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.[1] Strategy is one of the patterns included in the influential book "Design Patterns" by Gamma et al. that popularized the concept of using patterns in software design.

For instance, a class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type of data, the source of the data, user choice, and/or other discriminating factors. These factors are not known for each case until run-time, and may require radically different validation to be performed. The validation 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.

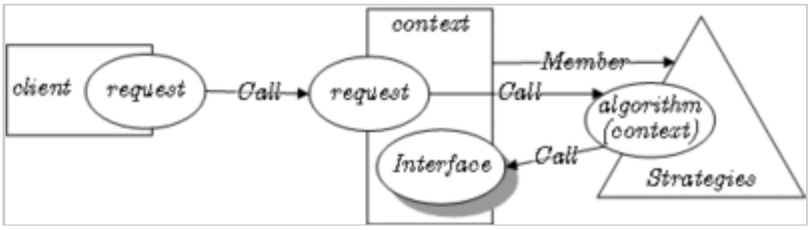
The essential requirement in the programming language is the ability to store a reference to some code in a data structure and retrieve it. This can be achieved by mechanisms such as the native function pointer, the first-class function, classes or class instances in object-oriented programming languages, or accessing the language implementation's internal storage of code via reflection.

Contents

Structure



Strategy Pattern in UML



Strategy pattern in LePUS3 (legend (http://lepus.org.uk/ref/legend/legend.xml))

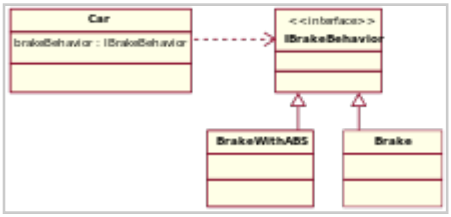
Example

The following example is in Java.

```java
/** The classes that implement a concrete strategy should implement this.
* The Context class uses this to call the concrete strategy. */
interface Strategy {
    int execute(int a, int b);
}

/** Implements the algorithm using the strategy interface */
class Add implements Strategy {
    public int execute(int a, int b) {
        System.out.println("Called Add's execute()");
        return a + b;  // Do an addition with a and b
    }
}

class Subtract implements Strategy {
    public int execute(int a, int b) {
        System.out.println("Called Subtract's execute()");
        return a - b;  // Do a subtraction with a and b
    }
}

class Multiply implements Strategy {
    public int execute(int a, int b) {
        System.out.println("Called Multiply's execute()");
        return a * b;   // Do a multiplication with a and b
    }
}
```

```java
// Configured with a ConcreteStrategy object and maintains
// a reference to a Strategy object
class Context {
    private Strategy strategy;

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

    public int executeStrategy(int a, int b) {
        return this.strategy.execute(a, b);
    }
}

/** Tests the pattern */
class StrategyExample {
    public static void main(String[] args) {
        Context context;

        // Three contexts following different strategies
        context = new Context(new Add());
        int resultA = context.executeStrategy(3,4);

        context = new Context(new Subtract());
        int resultB = context.executeStrategy(3,4);

        context = new Context(new Multiply());
        int resultC = context.executeStrategy(3,4);

        System.out.println("Result A : " + resultA );
        System.out.println("Result B : " + resultB );
        System.out.println("Result C : " + resultC );
    }
}
```

Strategy and open/closed principle



Accelerate and brake behaviors must be declared in each new car model

According to the strategy pattern, the behaviors of a class should not be inherited. Instead they should be encapsulated using interfaces. As an example, consider a car class. Two possible functionalities for car are *brake* and *accelerate*.

Since accelerate and brake behaviors change frequently between models, a common approach is to implement these behaviors in subclasses. This approach has significant drawbacks: accelerate and brake behaviors must be declared in each new Car model. The work of managing these behaviors increases greatly as the number of models increases, and requires code to be duplicated across models. Additionally, it is not easy to determine the exact nature of the behavior for each model without investigating the code in each.

The strategy pattern uses aggregation instead of inheritance. In the strategy pattern, behaviors are defined as separate interfaces and specific classes that implement these interfaces. Specific classes encapsulate these interfaces. This allows better decoupling between the behavior and the class that uses the behavior. The behavior can be changed without breaking the classes that use it, and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes. Behaviors can also be changed at run-time as well as at design-time. For instance, a car object's brake behavior can be changed from `BrakeWithABS()` to `Brake()` by changing the `brakeBehavior` member to:

```
brakeBehavior = new Brake();
```

This gives greater flexibility in design and is in harmony with the Open/closed principle (OCP) that states that classes should be open for extension but closed for modification.

See also

- Higher-order function
- List of object-oriented programming terms
- Mixin
- Policy-based design

References

1. ^ Eric Freeman, Elisabeth Freeman, Kathy Sierra and Bert Bates, *Head First Design Patterns*, First Edition, Chapter 1, Page 24, O'Reilly Media, Inc, 2004. ISBN 978-0-596-00712-6

External links

- Strategy Pattern in UML (Spanish, but english model) (http://design-patterns-with-uml.blogspot.com.ar/2013/02/strategy-pattern.html)

- The Strategy Pattern from the Net Objectives Repository (http://www.netobjectivesrepository.com/TheStrategyPattern)
- Strategy Pattern for Java article (http://www.javaworld.com/javaworld/jw-04-2002/jw-0426-designpatterns.html)
- Strategy Pattern for CSharp article (http://www.webbiscuit.co.uk/articles/the-strategy-pattern/)
- Strategy pattern in UML and in LePUS3 (http://www.lepus.org.uk

/ref/companion/Strategy.xml) (a formal modelling notation)
- Refactoring: Replace Type Code with State/Strategy (http://martinfowler.com/refactoring/catalog/replaceTypeCodeWithStateStrategy.html)

Categories: Software design patterns