Strategy pattern

From Wikipedia, the free encyclopedia

In computer programming, the **strategy pattern** (also known as the **policy pattern**) is a behavioural software design pattern that enables an algorithm's behavior to be selected at runtime. The strategy pattern

- defines a family of algorithms,
- encapsulates each algorithm, and
- makes the algorithms interchangeable within that family.

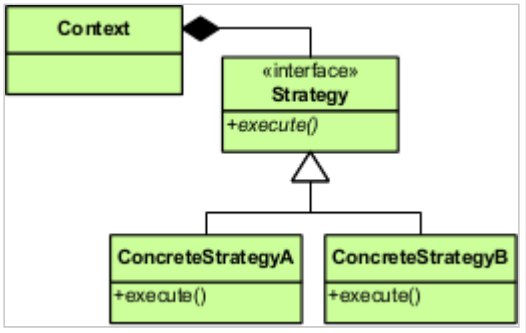
Strategy lets the algorithm vary independently from clients that use it.[1] Strategy is one of the patterns included in the influential book *Design Patterns* by Gamma et al. that popularized the concept of using patterns to describe software design.

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, 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.

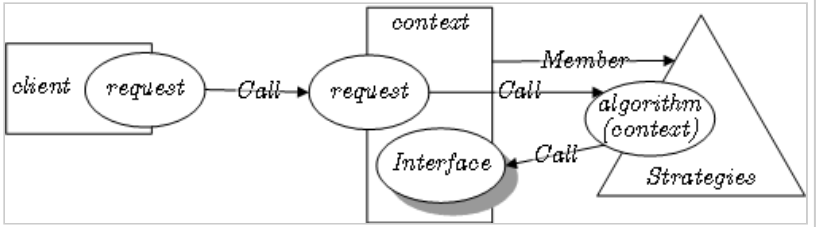
The essential requirement in the programming language is the ability to store a reference to some code in a data structure and retrieve it. This can be achieved by mechanisms such as the native function pointer, the first-class function, classes or class instances in object-oriented programming languages,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

C#

The following example is in C#.

```
using System;

public class Program
{
    public static void Main()
    {
        CalculateClient client = new CalculateClient(new Minus());

        Console.WriteLine("Minus: " + client.Calculate(7, 1));

        //Change the strategy
        client.strategy = new Plus();

        Console.WriteLine("Plus: " + client.Calculate(7, 1));
    }
}

//The interface for the strategies
public interface ICalculate
{
    int Calculate(int value1, int value2);
}

//strategies
//Strategy 1: Minus
public class Minus : ICalculate
{
    public int Calculate(int value1, int value2)
    {
        return value1 - value2;
    }
}

//Strategy 2: Plus
public class Plus : ICalculate
{
    public int Calculate(int value1, int value2)
    {
        return value1 + value2;
    }
}

//The client
public class CalculateClient
{
    public ICalculate strategy { get; set; }

    public CalculateClient(ICalculate _strategy)
    {
        strategy = _strategy;
    }

    //Executes the strategy
    public int Calculate(int value1, int value2)
    {
        return strategy.Calculate(value1, value2);
```

```
57     }
58 }
```

Java

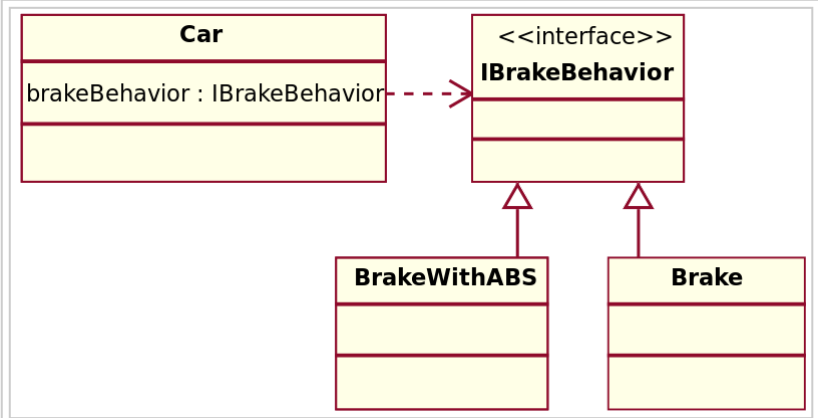
The following example is in Java.

```java
import java.util.ArrayList;
import java.util.List;

public class StrategyPatternWiki {

    public static void main(final String[] arguments) {
        Customer firstCustomer = new Customer(new NormalStrategy());

        // Normal billing
        firstCustomer.add(1.0, 1);

        // Start Happy Hour
        firstCustomer.setStrategy(new HappyHourStrategy());
        firstCustomer.add(1.0, 2);

        // New Customer
        Customer secondCustomer = new Customer(new HappyHourStrategy());
        secondCustomer.add(0.8, 1);
        // The Customer pays
        firstCustomer.printBill();

        // End Happy Hour
        secondCustomer.setStrategy(new NormalStrategy());
        secondCustomer.add(1.3, 2);
        secondCustomer.add(2.5, 1);
        secondCustomer.printBill();
    }
}

class Customer {

    private List<Double> drinks;
    private BillingStrategy strategy;

    public Customer(final BillingStrategy strategy) {
        this.drinks = new ArrayList<Double>();
        this.strategy = strategy;
    }

    public void add(final double price, final int quantity) {
        drinks.add(strategy.getActPrice(price*quantity));
    }

    // Payment of bill
    public void printBill() {
        double sum = 0;
        for (Double i : drinks) {
            sum += i;
        }
        System.out.println("Total due: " + sum);
        drinks.clear();
    }

    // Set Strategy
    public void setStrategy(final BillingStrategy strategy) {
        this.strategy = strategy;
    }

}

interface BillingStrategy {
    public double getActPrice(final double rawPrice);
}

// Normal billing strategy (unchanged price)
class NormalStrategy implements BillingStrategy {

    @Override
    public double getActPrice(final double rawPrice) {
```

```
        return rawPrice;
    }

}

// Strategy for Happy hour (50% discount)
class HappyHourStrategy implements BillingStrategy {

    @Override
    public double getActPrice(final double rawPrice) {
        return rawPrice*0.5;
    }

}
```

A much simpler example in "modern Java" (Java 8 and later), using lambdas, may be found here.

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 composition instead of inheritance. In the strategy pattern, behaviors are defined as separate interfaces and specific classes that implement 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();
```

```
/* Encapsulated family of Algorithms
 * Interface and its implementations
 */
public interface IBrakeBehavior {
    public void brake();
}

public class BrakeWithABS implements IBrakeBehavior {
    public void brake() {
        System.out.println("Brake with ABS applied");
    }
}

public class Brake implements IBrakeBehavior {
    public void brake() {
```

```
        System.out.println("Simple Brake applied");
    }
}

/* Client that can use the algorithms above interchangeably */
public abstract class Car {
    protected IBrakeBehavior brakeBehavior;

    public void applyBrake() {
        brakeBehavior.brake();
    }

    public void setBrakeBehavior(final IBrakeBehavior brakeType) {
        this.brakeBehavior = brakeType;
    }
}

/* Client 1 uses one algorithm (Brake) in the constructor */
public class Sedan extends Car {
    public Sedan() {
        this.brakeBehavior = new Brake();
    }
}

/* Client 2 uses another algorithm (BrakeWithABS) in the constructor */
public class SUV extends Car {
    public SUV() {
        this.brakeBehavior = new BrakeWithABS();
    }
}

/* Using the Car example */
public class CarExample {
    public static void main(final String[] arguments) {
        Car sedanCar = new Sedan();
        sedanCar.applyBrake();  // This will invoke class "Brake"

        Car suvCar = new SUV();
        suvCar.applyBrake();    // This will invoke class "BrakeWithABS"

        // set brake behavior dynamically
        suvCar.setBrakeBehavior( new Brake() );
        suvCar.applyBrake();    // This will invoke class "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

- Dependency injection
- Higher-order function
- List of object-oriented programming terms
- Mixin
- Policy-based design
- Type class

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

Wikimedia Commons has media related to ***Strategy (design pattern)***.

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.netobjectivestest.com/PatternRepository/index.php?title=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/posts/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)
- Implementation of the Strategy pattern in JavaScript (http://www.aleccove.com/2016/02/the-strategy-pattern-in-javascript)

The Wikibook *Computer Science Design Patterns* has a page on the topic of: ***Strategy implementations in various languages***

Retrieved from "https://en.wikipedia.org/w/index.php?title=Strategy_pattern&oldid=758306852"

Categories: Software design patterns
