



Aplicando Patrón Estrategia Usando C# 2.0

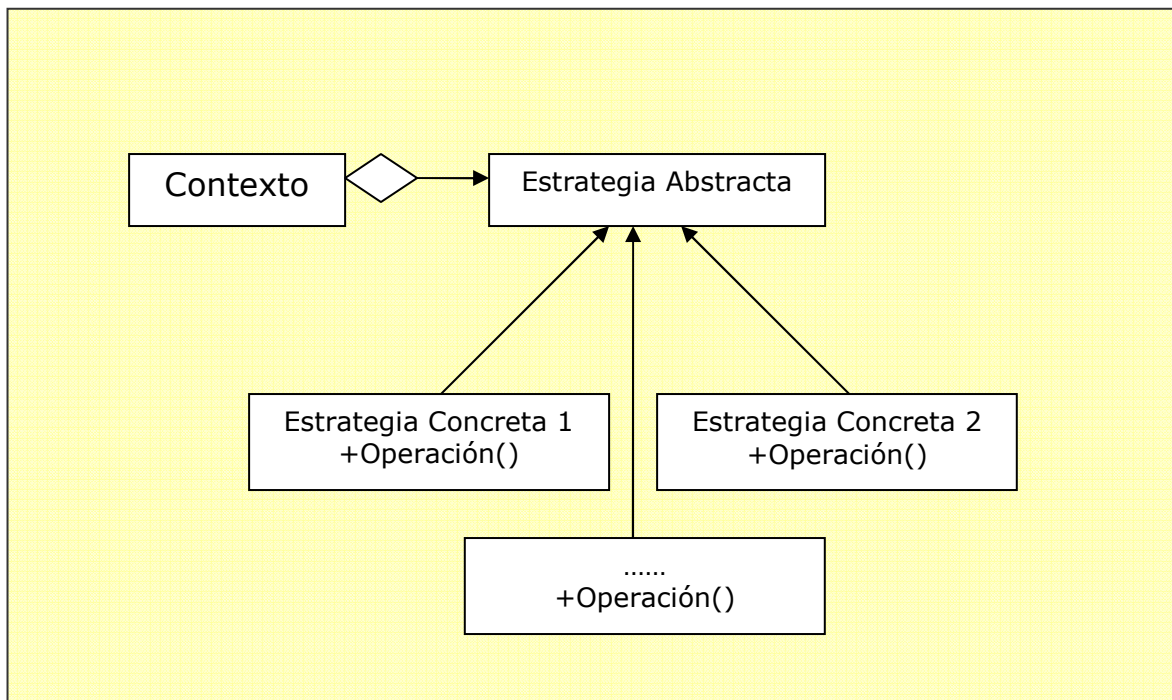
Antes de empezar con este patrón veamos un pequeño concepto del mismo, este fragmento es extraído de Wikipedia.

El patrón **Strategy** consiste en un conjunto de algoritmos encapsulados en un contexto determinado **Context**. El cliente puede elegir el algoritmo que prefiera de entre los disponibles o puede ser el mismo objeto **Context** el que elija el más apropiado para cada situación. Cualquier programa que ofrezca un servicio o función determinada, la cual puede ser realizada de varias maneras, es candidato a utilizar el patrón **Strategy**. Puede haber cualquier número de estrategias y cualquiera de ellas podrá ser intercambiada por otra en cualquier momento.

Propósito

Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.

Normalmente (según la presión del proyecto) los desarrolladores optamos por la solución más rápida y simple (inconsistente), frente a una variación del resultado según el algoritmo, ¿Por qué inconsistente?, veamos un ejemplo:



Vamos a ver un ejemplo basado en el siguiente modelo.

El contexto es un CONTENEDOR (CONTAINER) un contenedor es un MR de transporte de mercaderías manifestadas en conocimientos de embarque, un contenedor puede participa en operaciones de importación y exportación, a través de diferentes vías de transporte (marítima, aérea, terrestre, etc.).

La figura es un Terminal de Almacenamiento, lugar donde se pueden almacenar los contenedores que han llegado (para este ejemplo) en una importación declarados en un manifiesto de carga, por lo que para hacer el servicio tenemos que generar una orden de servicio ó carta de aceptación.

Al momento de recibir el contenedor debemos tener presente el tipo de contenedor, y diferentes variables, como dimensión, clase de bultos que se declaran en los conocimientos de embarque, etc.

Vamos a enfocarnos exclusivamente en aplicar una tarifa de descuento según el tipo de contenedor.

La solución clásica, rápida, sin definir una buena solución y que no debemos hacer es la que normalmente hacemos:

NOTA: El código adjunto no está optimizado, la intención del código es mostrar de manera rápida y practica el armado de la estructura y como trabaja el patrón **estrategia (strategy)**. No se maneja ningún tipo de técnica de control de recursos, objetos en memoria, entre otros.

```
public double calcularTarifaDescuento()
{
    if (Contenedor.getTipo() == Contenedor.Refrigerado) {
```

```
        return (Contenedor.getPrecio())*0.02;
    }
    else if (Contenedor.getTipo()==Contenedor.Consolidado){
        return (Contenedor.getPrecio())*0.25;
    }
    else if (Contenedor.getTipo()==Contenedor.Exclusivo){
        return (Contenedor.getPrecio())*0.835;
    }
    else
        throw new ExcepcionDescuentoContenedor();
}
```

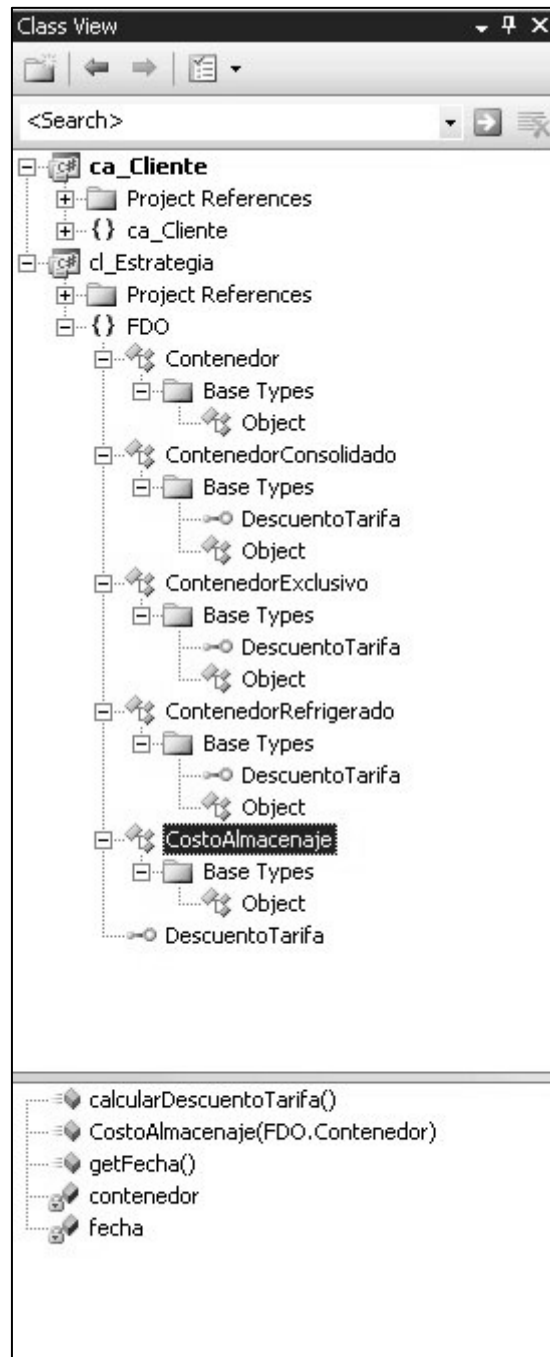
¿Cuál es el problema aquí?

En este ejemplo, los problemas concretos son dos:

- Si se introdujera un nuevo tipo de Contenedor, habría que modificar el código del cliente.
- Si se cambiaran los descuentos (por ejemplo, en época de rebajas), habría que modificar el código del cliente. El cliente está fuertemente acoplado con la política de descuentos.

Esta es una solución muy inestable, el principal problema, imaginen tenemos que recompilar todo nuestro cliente por un solo tipo de contenedor. Ahora hay muchas formas de resolver esto, pero que nos demuestra el patrón, pues veamos:

Una vista de la estructura de los objetos:



Ahora veamos paso a paso como implementamos el patrón estrategia usando C# 2.0.

1. Creamos la interfaz.

```
//Interfaz descuento a Tarifa por tipo de contenedor
//desde aquí aplicamos la estrategia abstracta
public interface DescuentoTarifa
{
    double calcularDescuentoTarifa(double precio);
}
```

2. Creamos las diferentes situaciones, el ejemplo lo he tratado de mostrar lo más simple posible, pero imaginemos una clase de encriptación que utilice diferentes algoritmos criptográficos ó una solución tipo Paint .NET que necesite aplicar diferentes algoritmos de formateo gráfico, etc.

```
/// <summary>
/// Objeto ContenedorExclusivo, esta clase es una de las estrategias
/// concretas.
/// </summary>
public class ContenedorExclusivo:DescuentoTarifa {
    // Constructor
    public ContenedorExclusivo() {}
    // Devuelve el precio con el descuento
    //(el % descuento lo podemos obtener de cualquier repositorio).
    public double calcularDescuentoTarifa(double precio) {
        return (precio * 0.50);
    }
}
/// <summary>
/// Objeto ContenedorConsolidado, esta clase es una de las estrategias
/// concretas.
/// </summary>
public class ContenedorConsolidado:DescuentoTarifa {
    // Constructor
    public ContenedorConsolidado() { }
    // Devuelve el precio con el descuento.
    //(el % descuento lo podemos obtener de cualquier repositorio).
    public double calcularDescuentoTarifa(double precio) {
        return (precio * 0.25);
    }
}
/// <summary>
/// Objeto ContenedorRefrigerado, esta clase es una de las estrategias
/// concretas.
/// </summary>
public class ContenedorRefrigerado:DescuentoTarifa {
    // Constructor
    public ContenedorRefrigerado() { }
    // Devuelve el precio con el descuento.
    //(el % descuento lo podemos obtener de cualquier repositorio).
    public double calcularDescuentoTarifa(double precio) {
        return (precio * 0.15);
    }
}
```

3. Creamos la clase contexto.

```
/// <summary>
/// Clase contexto Contenedor, representa un contenedor a almacenar
/// entregado por la línea.
/// </summary>
public class Contenedor {
    // Contenedor
    private String contenedor;
    // Precio de almacenaje.
    private double precio;
    // Tipo de tarifa descuento, según tipo de contenedor.
    private DescuentoTarifa DescuentoTarifa;

    public double getPrecio() {
        return precio;
    }

    public String getContenedor() {
        return contenedor;
    }

    public DescuentoTarifa getDescuentoTarifa() {
        return DescuentoTarifa;
    }

    public Contenedor(String _contenedor, double _precio, DescuentoTarifa
        _DescuentoTarifa)
    {
        if (_DescuentoTarifa == null)
        {
            // Cuando se crea un objeto Contenedor, es obligatorio
            // introducir
            // su tipo de tarifa de descuento.
            throw new Exception("Debe introducir un tipo de
                DescuentoTarifa");
        }
        else
        {
            this.contenedor = _contenedor;
            this.precio = _precio;
            this.DescuentoTarifa = _DescuentoTarifa;
        }
    }
}
```

4. Ahora creamos la clase que realizará el trabajo.

```
public class CostoAlmacenaje {

    // Fecha de Llegada.
    private DateTime fecha;
    // Contenedor
    private Contenedor contenedor;
```

```
public DateTime getFecha() {
    return fecha;
}
public CostoAlmacenaje(Contenedor _contenedor){
    this.fecha = new DateTime().Date;
    this.contenedor = _contenedor;
}

public double calcularDescuentoTarifa()
{
    return
    (contenedor.getDescuentoTarifa().calcularDescuentoTarifa(contenedor
    .getPrecio()));
}
}
```

5. Como trabajaría nuestra aplicación cliente, pues simple llamando los objetos y pasando el tipo de contenedor esto calculará la tarifa y el costo final de almacenaje.

```
using System;
using System.Collections.Generic;
using System.Text;
using FDO;
namespace ca_Cliente
{
    class Program
    {
        static void Main(string[] args)
        {
            Contenedor objContenedor = new Contenedor("TPPU2515257",
            500.10, new ContenedorRefrigerado());
            CostoAlmacenaje objCosto = new CostoAlmacenaje
            (objContenedor);
            Console.WriteLine("Código: " +
            objContenedor.getContenedor());
            Console.WriteLine("Tarifa Almacenaje: " +
            objContenedor.getPrecio());
            Console.WriteLine("Precio con Descuento Aplicado:" +
            objCosto.calcularDescuentoTarifa());
            Console.ReadLine();
        }
    }
}
```

6. Ahora F5 y a probar la aplicación.

Cualquier duda ó consulta estamos para apoyarnos.

www.FreedomDev.org – Promueve, Desarrolla e Investiga Interoperabilidad de Plataformas de Software.

Únete a nuestro proyecto Open Source:

Open Source Terminal of Storage

Plataforma Inicial Base: Microsoft Windows.

Máquina Virtual Base: Microsoft .NET Framework 2.0 Beta 2.

IDE's de Desarrollo Base: Microsoft Visual C# Express Edition Beta 2.

Base de Datos Inicial: PostgreSQL 8.0.

Contacto: aarroyor@freedomdev.org