



PRÁCTICA 02 - PATRONES

PATRONES DE COMPORTAMIENTO

DESARROLLO RÁPIDO DE APLICACIONES

Por Juan José Camacho Hidalgo
UNIVERSIDAD DE ALMERÍA

Contenido

1.	Introducción	2
2.	Patrones de comportamiento	2
2.1.	Interpreter.....	2
2.1.1.	Funcionamiento (base: InterpreterConsoleApp)	2
2.1.2.	Ejecución	3
2.2.	Template Method	3
2.2.1.	Funcionamiento (base: TemplateMethodConsoleApp)	3
2.2.2.	Ejecución	4
2.2.3.	Ampliación.....	4
2.3.	Chain of Responsibility	4
2.3.1.	Funcionamiento (base: ChainOfResponsabilityConsoleApp)	4
2.3.2.	Ejecución	5
	5
2.3.3.	Ampliación.....	6
2.4.	Command.....	6
2.4.1.	Funcionamiento (base: CommandConsoleApp)	6
2.4.2.	Ejecución	7
2.5.	Iterator	8
2.5.1.	Funcionamiento (base: IteratorConsoleApp).....	8
2.5.2.	Ejecución	8
2.6.	Mediator	9
2.6.1.	Funcionamiento (base: MediatorConsoleApp)	9
2.6.2.	Ejecución	9
2.7.	Memento	10
2.7.1.	Funcionamiento (base: MementoConsoleApp)	10
2.7.2.	Ejecución	11
2.8.	Observer.....	11
2.8.1.	Funcionamiento (base: ObserverConsoleApp)	11
2.8.2.	Ejecución	12
2.9.	State.....	13
2.9.1.	Funcionamiento (base: StateConsoleApp).....	13
2.9.2.	Ejecución	13
2.10.	Strategy.....	14
2.10.1.	Funcionamiento (base: StrategyConsoleApp).....	14
2.10.2.	Ejecución	14
2.11.	Visitor.....	15
2.11.1.	Funcionamiento (base: VisitorConsoleApp).....	15
2.11.2.	Ejecución	16

1. Introducción

Este documento tiene como objetivo la explicación y justificación del uso de patrones de construcción en la programación, mediante ejemplos claros escritos en C#.

2. Patrones de comportamiento

Describen la comunicación entre objetos o clases.

2.1. Interpreter

2.1.1. Funcionamiento (base: InterpreterConsoleApp)

Este programa comprueba que una consulta tiene el formato deseado según una gramática planteada en la clase abstracta *Expresion*. Para ello se pueden usar el *OperadorY* y el *OperadorO*, para realizar una consulta. Por tanto, concluye, mediante un *OperadorBinario*. Este patrón proporciona un marco para evaluar, interpretar, expresiones escritas en el lenguaje que se propone.

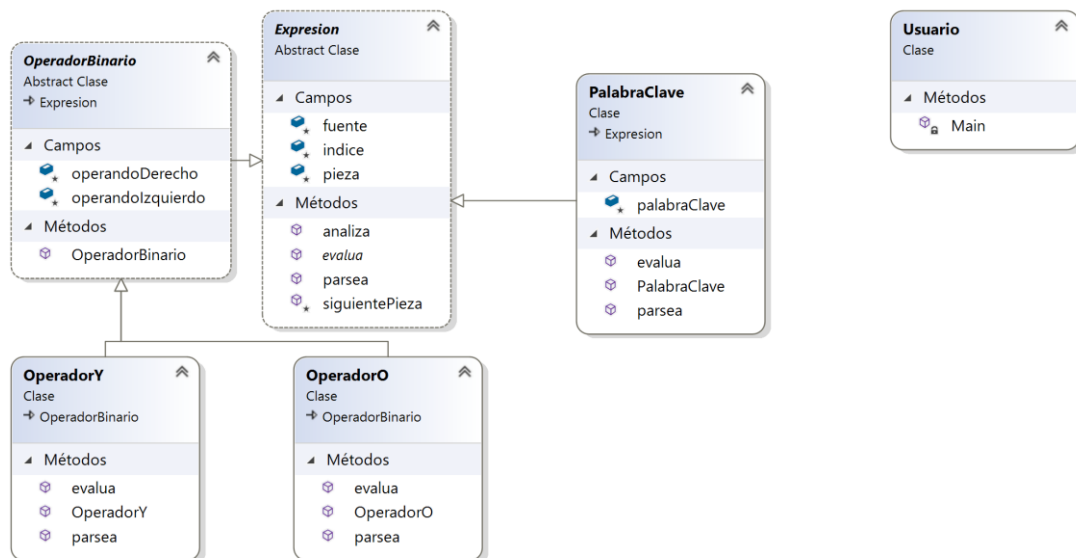


Diagrama de clases de *InterpreterConsoleApp*.

2.1.2. Ejecución

Aquí hacemos uso del lenguaje permitido para realizar una consulta para ver si disponen de un Opel Astra o Renault Megane.

```
C:\WINDOWS\system32\cmd.exe
```

```
Introduzca su consulta: opel astra o renault megane
Introduzca la descripción de un vehículo:
opel
La descripción responde a la consulta
```

Como ampliación, se podrían crear otros tipos diferentes de gramática para las consultas.

2.2. Template Method

2.2.1. Funcionamiento (base: TemplateMethodConsoleApp)

Este patrón permite delegar en las subclases las operaciones de un objeto, por tanto, en una operación, define el algoritmo, pero dejando algunos pasos a las subclases. Ahora, aplicado a la solución concreta, tenemos un Usuario que realiza varios pedidos. Se realiza un pedido, definido por la clase Pedido, pero el calculo del IVA, que es un paso del algoritmo, lo realizan las clases específicas: PedidoLuxemburgo y PedidoEspania, ya que el IVA es distinto en cada una.

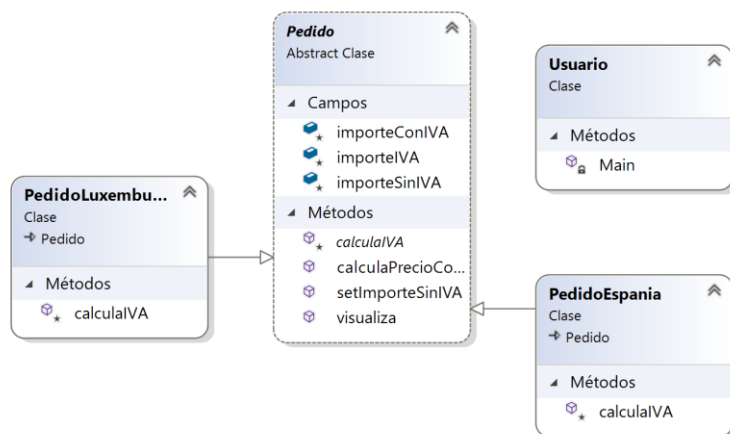
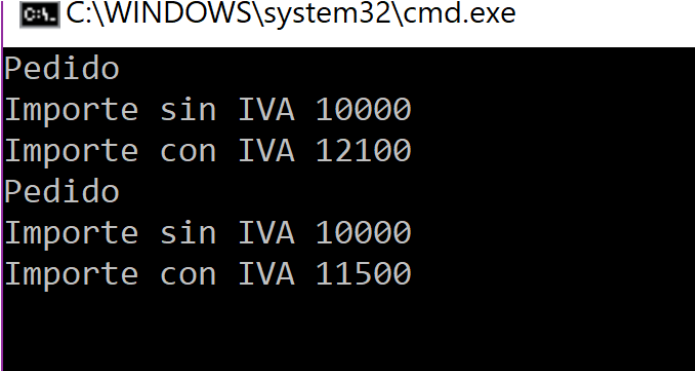


Diagrama de clases de TemplateMethodConsoleApp.

2.2.2. Ejecución

Aquí en la ejecución vemos como el pedido sin IVA es igual, ya que lo realiza la clase Pedido y esa es común, pero el importe con IVA varía según el estado, y por tanto, el método proviene de clases distintas.



```
C:\WINDOWS\system32\cmd.exe
Pedido
Importe sin IVA 10000
Importe con IVA 12100
Pedido
Importe sin IVA 10000
Importe con IVA 11500
```

2.2.3. Ampliación

Para justificar más aún el patrón, podríamos añadir una clase PedidoFrancia que herede de la clase Pedido, con el método propio calculaIVA, sobrescribiendo al heredado, y calculando desde Usuario el IVA de este estado también. El importe sin IVA seguiría intacto.

2.3. Chain of Responsibility

2.3.1. Funcionamiento (base: ChainOfResponsabilityConsoleApp)

Este patrón nos permite definir una cadena de objetos, de forma que se realiza una solicitud, y todos los objetos intentan responderla, pero si no pueden, la transmiten hacia el siguiente.

En este caso, Vehiculo, Marca y Modelo, son la cadena de objetos, por eso heredan las características de ObjetoBasico, y todas tienen el método devuelveDescripción. Si pueden devolver la descripción, la devuelven y le dan la oportunidad al siguiente de hacer lo mismo. Si no pueden devolver la descripción devuelven una por defecto y pasan al siguiente. Si no hay siguiente, se cierra la solicitud. Esa es la función de devuelveDescripcion().

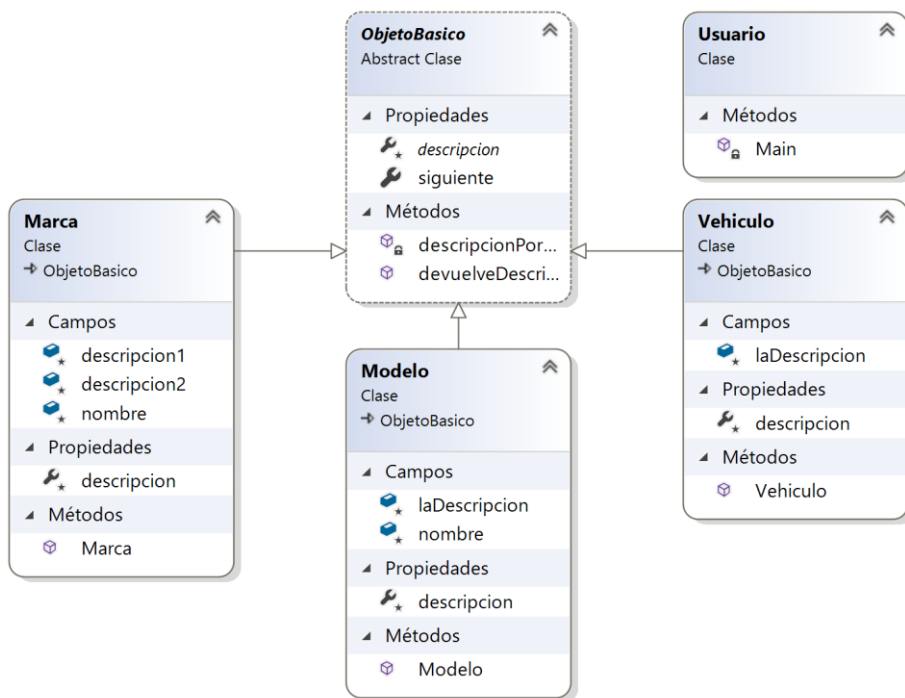


Diagrama de clases de ChainOfResponsabilityConsoleApp.

2.3.2. Ejecución

En la ejecución podemos ver como el vehículo Auto++ devuelve su descripción, después el Modelo y después la Marca. Después, según el código, se crea un modelo sin descripción, por eso genera “Descripción por defecto” y por último, se crean dos nuevos vehículos. Como ninguno aparece distinto de null, no se muestra nada ya que no pueden atender la solicitud.

```

C:\WINDOWS\system32\cmd.exe


Auto++ KT500 Vehículo de ocasión en buen estado
Modelo KT400 : Vehículo amplio y confortable
Marca Auto++ : Marca del automóvil de gran calidad
descripción por defecto

0 referencias | Camacho Hidalgo, Juan Jose, Hace 26 días | 1 autor, 1 cambio
static void Main(string[] args)
{
    ObjetoBasico vehiculo1 = new Vehiculo("Auto++ KT500 Vehículo de ocasión en buen estado");
    Console.WriteLine(vehiculo1.devuelveDescripcion());
    ObjetoBasico modelo1 = new Modelo("KT400", "Vehículo amplio y confortable");
    ObjetoBasico vehiculo2 = new Vehiculo(null);
    vehiculo2.siguiente = modelo1;
    Console.WriteLine(vehiculo2.devuelveDescripcion());
    ObjetoBasico marca1 = new Marca("Auto++", "Marca del automóvil", "de gran calidad");
    ObjetoBasico modelo2 = new Modelo("KT700", null);
    modelo2.siguiente = marca1;
    ObjetoBasico vehiculo3 = new Vehiculo(null);
    vehiculo3.siguiente = modelo2;
    Console.WriteLine(vehiculo3.devuelveDescripcion());
    ObjetoBasico vehiculo4 = new Vehiculo(null);
    Console.WriteLine(vehiculo4.devuelveDescripcion());

    Console.ReadLine();
}
  
```

2.3.3. Ampliación

Podríamos añadir descripción a Vehículo3 y veríamos como Vehículo si atiende la solicitud, antes de que lo haga modelo o marca.

 C:\WINDOWS\system32\cmd.exe

```
Auto++ KT500 Vehículo de ocasión en buen estado
Modelo KT400 : Vehículo amplio y confortable
Turboauto+ de calidad máxima
descripción por defecto
```

2.4. Command

2.4.1. Funcionamiento (base: CommandConsoleApp)

Este patrón se basa en ejecutar operaciones sin conocer los detalles de la implementación de estas. El nombre viene de que cada operación se le puede llamar comando. Siguiendo esta filosofía, el propósito es encapsular la ejecución de una acción como un objeto. En este caso, el Usuario dispone de vehículos en Stock (tipo Vehiculo), y mediante el Catalogo los revisa y puede emitir una solicitud de rebaja. Entonces el Usuario realiza esta operación mediante el método de Catalogo, y hace uso de la clase SolicitudRebaja, que es donde se esconde la implementación. Facilita operaciones como anulación, encolamiento de solicitudes y seguimiento de esta manera.

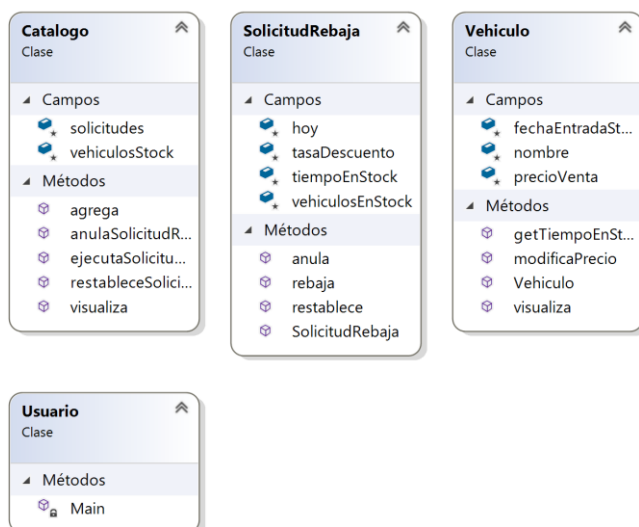


Diagrama de clases de InterpreterConsoleApp.

2.4.2. Ejecución

En la ejecución vemos como se realizan varias solicitudes, y se muestra el catálogo. Cada solicitud tiene su propio contenido. Una buena ampliación de este programa sería añadir otro tipo de solicitudes.

```
C:\WINDOWS\system32\cmd.exe
Visualización inicial del catálogo
A01 precio: 1000 fecha entrada stock 1
A11 precio: 2000 fecha entrada stock 6
Z03 precio: 3000 fecha entrada stock 2

Visualización del catálogo tras ejecutar la primera solicitud
A01 precio: 900 fecha entrada stock 1
A11 precio: 2000 fecha entrada stock 6
Z03 precio: 2700 fecha entrada stock 2

Visualización del catálogo tras ejecutar la segunda solicitud
A01 precio: 450 fecha entrada stock 1
A11 precio: 2000 fecha entrada stock 6
Z03 precio: 1350 fecha entrada stock 2

Visualización del catálogo tras anular la primera solicitud
A01 precio: 500 fecha entrada stock 1
A11 precio: 2000 fecha entrada stock 6
Z03 precio: 1500 fecha entrada stock 2

Visualización del catálogo tras restablecer la primera solicitud
A01 precio: 450 fecha entrada stock 1
A11 precio: 2000 fecha entrada stock 6
Z03 precio: 1350 fecha entrada stock 2
```


2.5. Iterator

2.5.1. Funcionamiento (base: IteratorConsoleApp)

Este ejemplo muestra como este patrón crea un acceso secuencial a una colección de vehículos de tipo Vehículo. Se crea una estructura de datos de tipo CatalogoVehiculo, y se añaden vehículos a dicha estructura. Mediante la clase IteradorVehiculo, se recorre uno a uno la estructura de datos y realiza la operación solicitada sobre este. En este caso, se realiza la búsqueda de un vehículo económico. Al utilizar el iterador para realizar la operación, no se exponen los detalles de la representación del elemento, en este caso, Vehiculo.

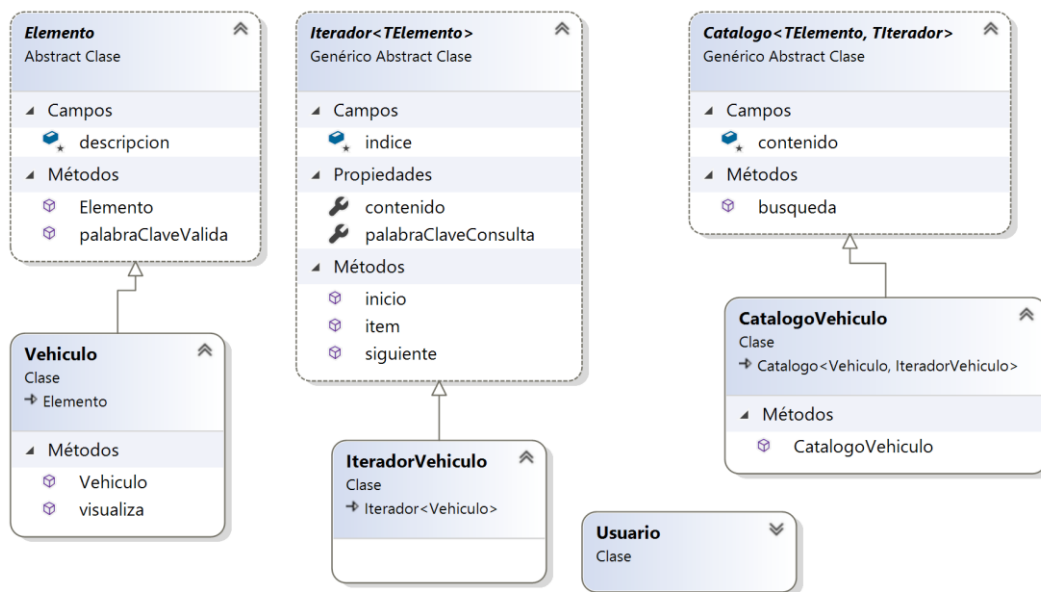


Diagrama de clases de IteratorConsoleApp.

2.5.2. Ejecución

Se realiza una búsqueda sobre la estructura de datos mediante el iterador y nos devuelve el vehículo que sea económico sin exponer ningún detalle. Podríamos buscar más parámetros, o crear más objetos dentro de la estructura, sin exponer la representación.

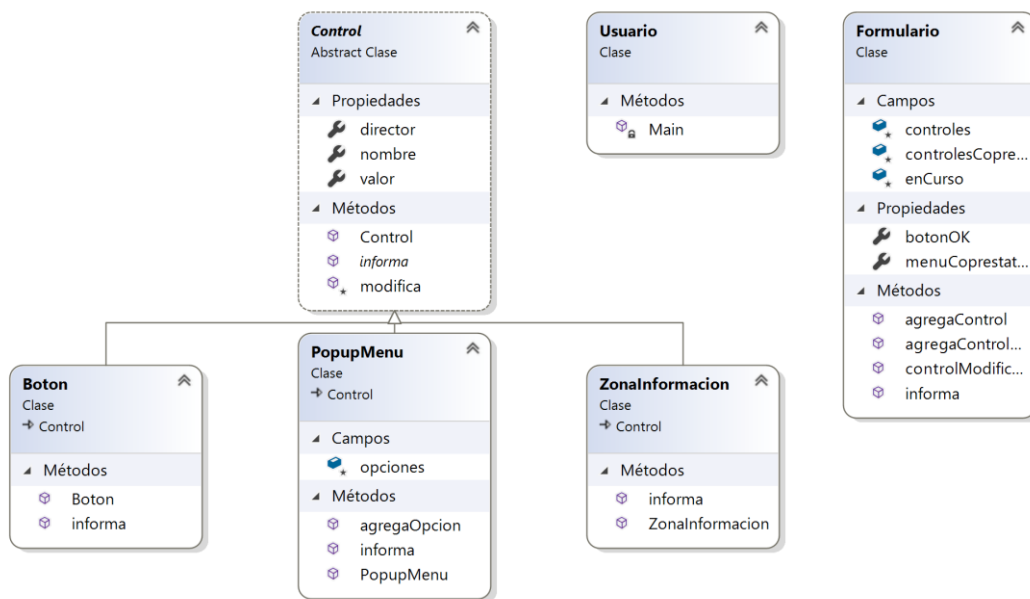
```
C:\WINDOWS\system32\cmd.exe
```

```
Descripción del vehículo: pequeño vehículo económico
```

2.6. Mediator

2.6.1. Funcionamiento (base: MediatorConsoleApp)

Este ejemplo tiene como objetivo construir un objeto tipo Control cuyo cometido es gestionar y controlar las interacciones entre Boton, PopupMenu y ZonalInformacion. Por tanto, Control realiza el rol de mediador, mientras que Formulario es la clase que se encarga de manejar ese Control. En este programa solo se crean subclases del mediador.



2.6.2. Ejecución

En la ejecución, se informa sobre el coprestatario con nombre cualquiera.

```
C:\> C:\WINDOWS\system32\cmd.exe
Información de: Nombre
```

```

public class Usuario
{
    0 referencias | Camacho Hidalgo, Juan Jose, Hace 26 días | 1 autor, 1 cambio
    static void Main(string[] args)
    {
        Formulario formulario = new Formulario();
        formulario.agregaControl(new ZonaInformacion("Nombre"));
        formulario.agregaControl(new ZonaInformacion("Apellidos"));
        PopupMenu menu = new PopupMenu("Coprestatario");
        menu.agregaOpcion("sin coprestatario");
        menu.agregaOpcion("con coprestatario");
        formulario.agregaControl(menu);
        formulario.menuCoprestatario = menu;
        Boton boton = new Boton("OK");
        formulario.agregaControl(boton);
        formulario.botonOK = boton;
        formulario.agregaControlCoprestatario(new ZonaInformacion("Nombre del coprestatario"));
        formulario.agregaControlCoprestatario(new ZonaInformacion("Apellidos del coprestatario"));
        formulario.informa();

        Console.ReadLine();
    }
}

```

En cuanto a la ampliación reutilizar una clase es difícil porque tiene dependencias con otras clases, pero se puede añadir nuevas.

2.7. Memento

2.7.1. Funcionamiento (base: MementoConsoleApp)

Este patrón tiene como objetivo guardar el estado de un objeto en un momento concreto, sin violar la encapsulación. Para ello, en este ejemplo se crea MementoImpl, una clase que implementaría la interfaz Memento, que puede ser modificada. La clase MementoImpl presenta una propiedad de estado, donde almacenamos el estado del objeto que queremos guardar. En este caso, disponemos de un carrito de opciones (CarritoOpciones) donde nos permiten elegir si queremos asientos en cuero, reclinables o asientos deportivos para nuestro vehículo (esto sería cada uno una OpcionVehiculo). Las opciones que elijamos de nuestro carrito se guardarían temporalmente y solo dentro del tiempo de ejecución de nuestro programa, para mostrarse a continuación a demanda del usuario que lo solicite.

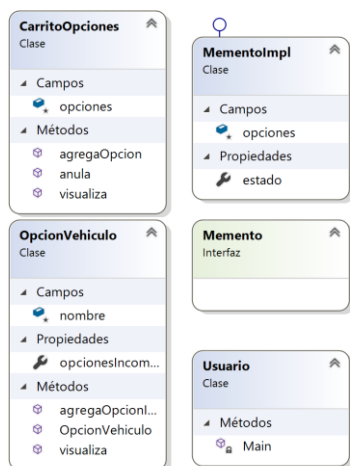


Diagrama de clases de MementoConsoleApp.

2.7.2. Ejecución

En la ejecución se muestra el contenido de las opciones seleccionadas del carrito para el cliente (en este caso Usuario).

C:\WINDOWS\system32\cmd.exe

```
Contenido del carrito de opciones
opción: Asientos en cuero
opción: Reclinables

Contenido del carrito de opciones
opción: Asientos deportivos

Contenido del carrito de opciones
opción: Asientos en cuero
opción: Reclinables
```

Una forma de ampliar este programa sería añadir más opciones al carrito, o incluso crear otro tipo de carrito para elegir opciones como tipo de llantas.

2.8. Observer

2.8.1. Funcionamiento (base: ObserverConsoleApp)

En esta solución, se utiliza el patrón Observer. Se dispone una interfaz Observador, la cual es implementada por VistaVehiculo. Esta clase tiene como objetivo modificar el texto de un objeto de tipo Vehiculo referente al precio, cuando se crea el objeto. Es decir, cada vez que se llama al constructor pero el precio ha sido modificado, este lo actualiza. Por tanto se crea una dependencia entre observadores y Vehiculo (observable).

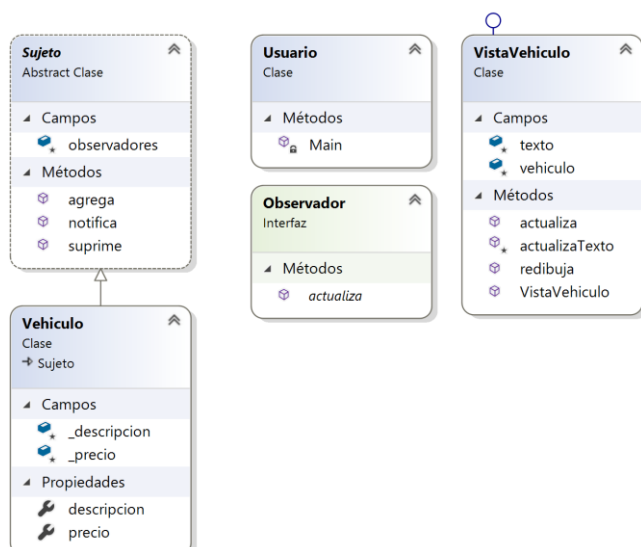



Diagrama de clases de ObserverConsoleApp

2.8.2. Ejecución

Como vemos en la ejecución, el precio del vehículo se modifica y se actualiza.

 C:\WINDOWS\system32\cmd.exe

```
Descripción Vehículo económico Precio: 5000
Descripción Vehículo económico Precio: 4500
Descripción Vehículo económico Precio: 5500
Descripción Vehículo económico Precio: 5500
```

0 referencias | Camacho Hidalgo, Juan Jose, Hace 26 días | 1 autor, 1 cambio

```
static void Main(string[] args)
{
    Vehiculo vehiculo = new Vehiculo();
    vehiculo.descripcion = "Vehículo económico";
    vehiculo.precio = 5000.0;
    VistaVehiculo vistaVehiculo = new VistaVehiculo(vehiculo);
    vistaVehiculo.redibuja();
    vehiculo.precio = 4500.0;
    VistaVehiculo vistaVehiculo2 = new VistaVehiculo(vehiculo);
    vehiculo.precio = 5500.0;

    Console.ReadLine();
}
```

2.9. State

2.9.1. Funcionamiento (base: StateConsoleApp)

El patrón State se muestra aquí desde la clase de EstadoPedido, que permite a un objeto Pedido, según su estado (EstadoPedido), cambiar su comportamiento acorde a éste.

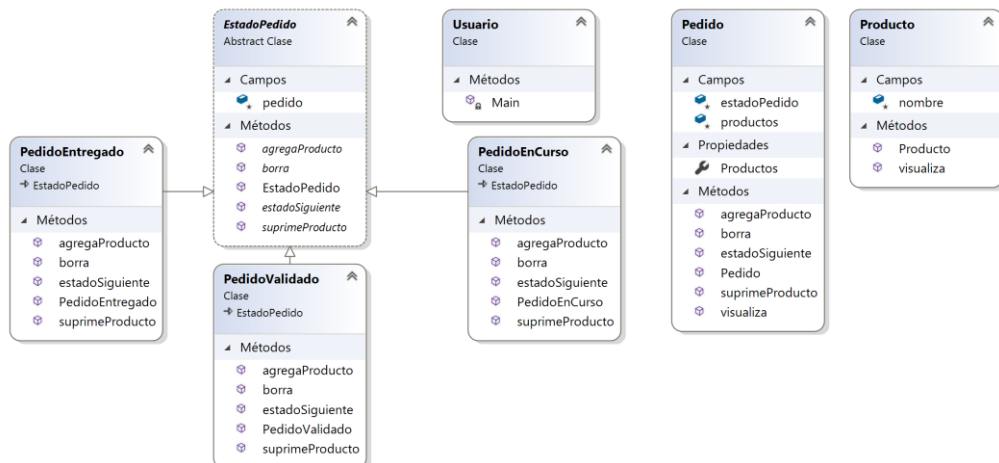


Diagrama de clases de StateConsoleApp.

2.9.2. Ejecución

En la ejecución mostramos el contenido del pedido vehículo 1 y accesorio 2, después pasa al estado siguiente, se añade el accesorio 3, pero se borra antes de mostrarse. Después se realiza un segundo pedido, se agrega un vehículo 11 con accesorio 21, se visualiza, después pasa al estado siguiente y así de nuevo hasta que pasa por los tres estados. Esto permite separar el comportamiento de cada estado, e incluso poder modificarse el pedido (aunque no es el caso particular de este programa).

C:\WINDOWS\system32\cmd.exe

```
Contenido del pedido
Producto: vehículo 1
Producto: accesorio 2

Contenido del pedido

Contenido del pedido
Producto: vehículo 11
Producto: accesorio 21

Contenido del pedido
Producto: vehículo 11
Producto: accesorio 21

Contenido del pedido
Producto: vehículo 11
Producto: accesorio 21
```

Como ampliación se propondría añadir más estados como cancelado, o devuelto, así como más productos dentro del pedido.

2.10. Strategy

2.10.1. Funcionamiento (base: StrategyConsoleApp)

Mediante una interfaz `DibujaCatalogo` implementada por las dos clases de métodos dibuja exclusivamente, el `Usuario` genera un catálogo de vehículos según el tipo de método seleccionado. Hace uso de `VistaCatalogo` para generar el catalogo y de `VistaVehiculo` para dibujar cada vehículo del Catálogo. Por tanto, se observa como el patrón Strategy permite encapsular cada uno de los métodos de dibujar e intercambiarlos a petición del usuario del programa adaptándose así a sus necesidades.



Diagrama de clases de `StrategyConsoleApp`.

2.10.2. Ejecución

En la ejecución podemos observar como el patrón hace uso en el ejemplo, de los dos métodos de dibujar, y exporta el catálogo de dos formas distintas.

```
C:\WINDOWS\system32\cmd.exe
Dibuja los vehículos mostrando tres vehículos por línea
vehículo económico vehículo amplio vehículo rápido
vehículo confortable vehículo deportivo

Dibuja los vehículos mostrando un vehículo por línea
vehículo económico
vehículo amplio
vehículo rápido
vehículo confortable
vehículo deportivo
```

Como ampliación se propone generar nuevos tipos de métodos para dibujar el catálogo, y que el usuario haga uso de todos ellos. Por ejemplo, mostrando dos vehículos por línea, o mostrando una línea si y otra no.

2.11. Visitor

2.11.1. Funcionamiento (base: VisitorConsoleApp)

Esta solución presenta dos tipos de empresa: EmpresaMadre y EmpresaSinFilial. Ambas clases heredan de una clase abstracta Empresa, que es la clase a la que sobrescriben los métodos. Se presenta una interfaz Visitante, que tiene como objetivo añadir un método visita, sin alterar las clases ni métodos de las subclases de Empresa. Para ello, se implementa en VisitanteMailingComercial, creando un método visita que añade una solicitud nueva por email de propuesta comercial para un grupo. La estructura de empresas no se ve alterada por esta clase, gracias al patrón.

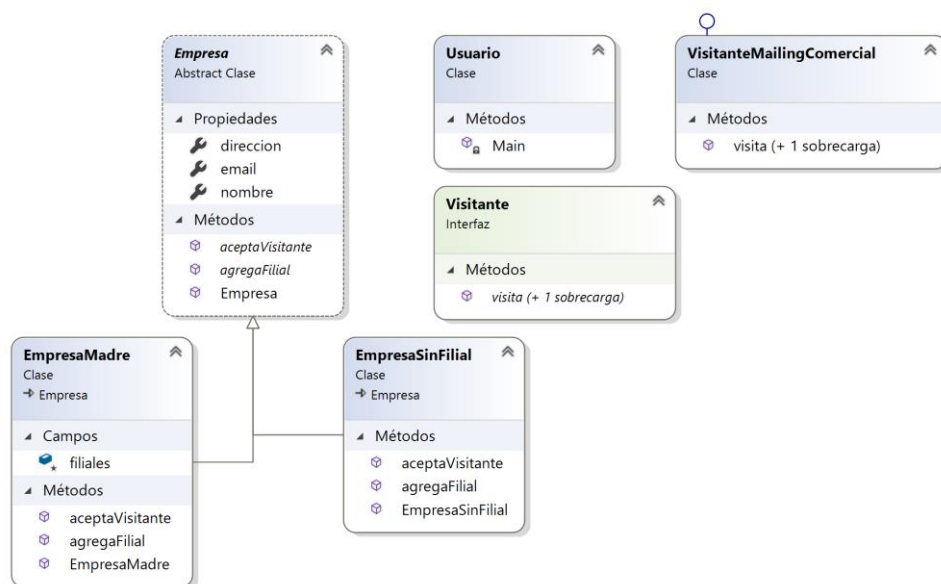


Diagrama de clases de VisitorConsoleApp.

2.11.2. Ejecución

En la ejecución, vemos como se muestran las diferentes propuestas comerciales, sin alterar las clases de empresa en cuestión. Todo esto, gracias a la interfaz de Visitor.

C:\WINDOWS\system32\cmd.exe

```
Envía un email a grupo2 dirección: info@grupo2.com Propuesta comercial para su grupo
Impresión de un correo electrónico para grupo2 dirección: calle del grupo 2 Propuesta comercial para su grupo
Envía un email a grupo1 dirección: info@grupo1.com Propuesta comercial para su grupo
Impresión de un correo electrónico para grupo1 dirección: calle del grupo 1 Propuesta comercial para su grupo
Envía un email a empresa1 dirección: info@empresa1.com Propuesta comercial para su empresa
Envía un email a empresa2 dirección: info@empresa2.com Propuesta comercial para su empresa
Envía un email a empresa3 dirección: info@empresa3.com Propuesta comercial para su empresa
```