



PRÁCTICA 02 - PATRONES

PATRONES DE CONSTRUCCIÓN

DESARROLLO RÁPIDO DE APLICACIONES

Por Juan José Camacho Hidalgo
UNIVERSIDAD DE ALMERÍA

Contenido

1. Introducción	2
2. Patrones de construcción.....	2
2.1. Abstract Factory	2
2.1.1. Funcionamiento (base: AbstractFactoryConsoleApp).....	2
2.1.2. Ejecución	3
2.1.3. Ampliación.....	3
2.2. Builder	4
2.2.1. Funcionamiento (base: BuilderConsoleApp).....	4
2.2.2. Ejecución	5
2.2.3. Ampliación.....	5
2.3. Factory Method.....	6
2.3.1. Funcionamiento (base: FactoryMethodConsoleApp)	6
2.3.2. Ejecución	7
2.3.3. Ampliación.....	7
2.4. Prototype.....	8
2.4.1. Funcionamiento (base: PrototypeConsoleApp)	8
2.4.2. Ejecución	9
2.4.3. Ampliación.....	9
2.5. Singleton.....	10
2.5.1. Funcionamiento (base: SingletonConsoleApp)	10
2.5.2. Ejecución	11

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 construcción

Los patrones de construcción son patrones que ayudan a resolver problemas relacionados con la creación de instancias de una clase, y por ende separar la implementación del cliente con la de los objetos que se utilizan. Nos ayudan a abstraer y encapsular su creación.

2.1. Abstract Factory

2.1.1. Funcionamiento (base: AbstractFactoryConsoleApp)

En primer lugar, mostramos el diagrama de clases de la solución que presenta el patrón a estudiar Abstract Factory.

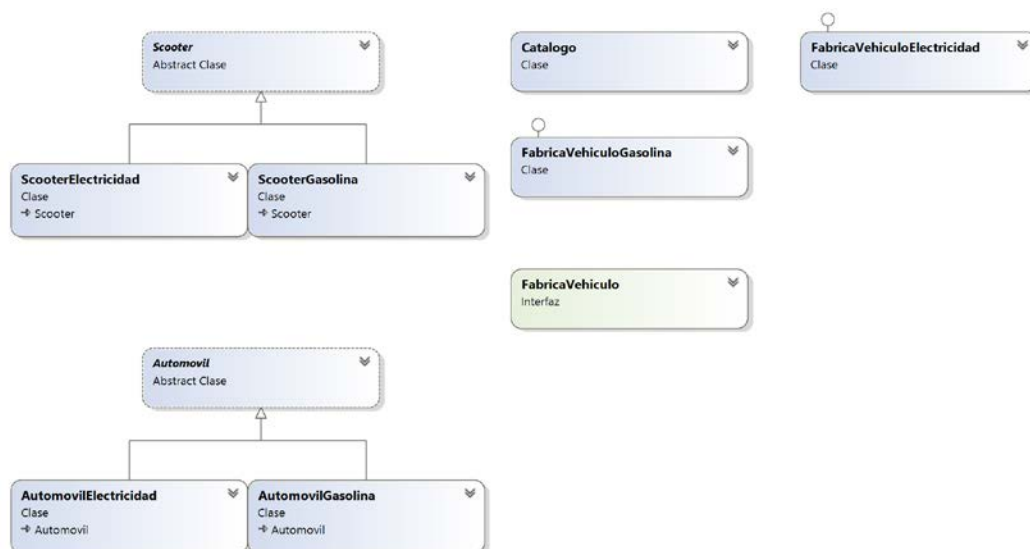


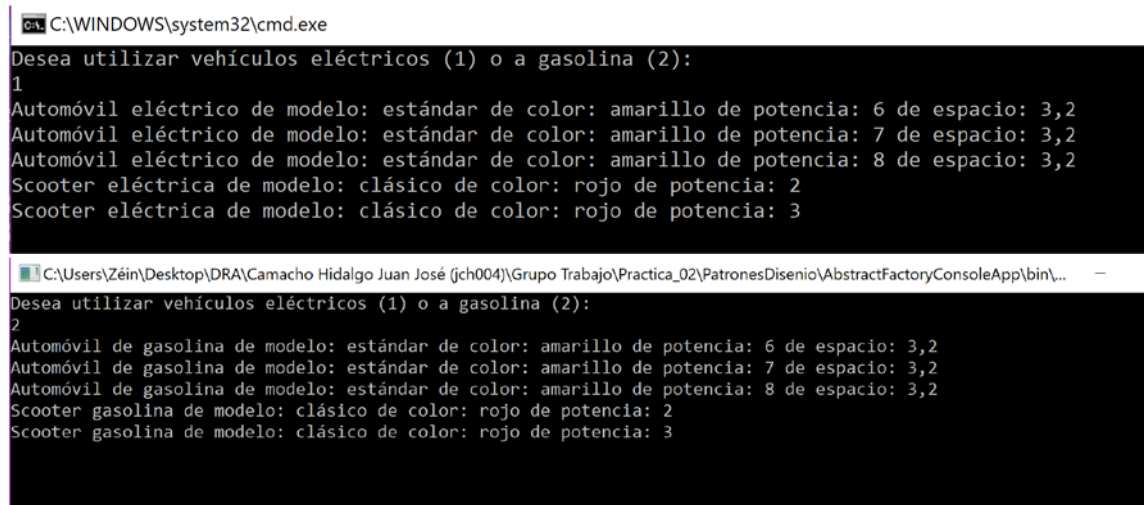
Diagrama de clases de la solución específica para el patrón Abstract Factory.

Como podemos observar en el código, mediante la interfaz **FabricaVehiculo.cs**, crea familias de tipo **Scooter** y **Automovil**. No conoce las clases con los atributos concretos (ejemplo: **ScooterElectricidad** o **AutomovilGasolina**). Las clases **FabricaVehiculoElectricidad.cs** y **FabricaVehiculoGasolina.cs** hacen uso de la interfaz **FabricaVehiculo.cs**, la cual implementan, para crear ambos tipos de vehiculo sin tener acceso directo a las clases que se encargan de esto. Por tanto, se crea una separación lógica. Esto es muy útil si se quiere cambiar la lógica de

fabricación específica de vehículos más adelante, ya que permite mantener la interfaz, hacer uso de ella, y cambiar las clases de fabricación de vehículos, o introducir nuevos tipos de vehículos sin afectar al funcionamiento de la aplicación.

2.1.2. Ejecución

En la ejecución del programa vemos como introduciendo la opción 1, genera vehículos eléctricos, obteniendo los datos de cada uno. Mientras, con la opción 2, generamos vehículos de gasolina.

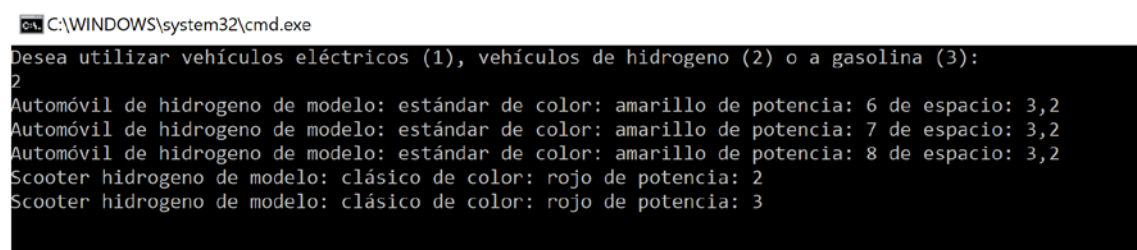


```
C:\WINDOWS\system32\cmd.exe
Desea utilizar vehículos eléctricos (1) o a gasolina (2):
1
Automóvil eléctrico de modelo: estándar de color: amarillo de potencia: 6 de espacio: 3,2
Automóvil eléctrico de modelo: estándar de color: amarillo de potencia: 7 de espacio: 3,2
Automóvil eléctrico de modelo: estándar de color: amarillo de potencia: 8 de espacio: 3,2
Scooter eléctrica de modelo: clásico de color: rojo de potencia: 2
Scooter eléctrica de modelo: clásico de color: rojo de potencia: 3

C:\Users\Zéin\Desktop\DRA\Camacho Hidalgo Juan José (jch004)\Grupo Trabajo\Practica_02\PatronesDiseño\AbstractFactoryConsoleApp\bin\...
Desea utilizar vehículos eléctricos (1) o a gasolina (2):
2
Automóvil de gasolina de modelo: estándar de color: amarillo de potencia: 6 de espacio: 3,2
Automóvil de gasolina de modelo: estándar de color: amarillo de potencia: 7 de espacio: 3,2
Automóvil de gasolina de modelo: estándar de color: amarillo de potencia: 8 de espacio: 3,2
Scooter gasolina de modelo: clásico de color: rojo de potencia: 2
Scooter gasolina de modelo: clásico de color: rojo de potencia: 3
```

2.1.3. Ampliación

Como comprobación de todo lo mencionado anteriormente, hemos añadido otro tipo de vehículos. En este caso hemos añadido vehículos de hidrógeno. Como vemos, sin modificar la interfaz y la FabricaVehiculo, conseguimos crear otro tipo de vehículos y justificar así el uso del patrón eficazmente.



```
C:\WINDOWS\system32\cmd.exe
Desea utilizar vehículos eléctricos (1), vehículos de hidrogeno (2) o a gasolina (3):
2
Automóvil de hidrogeno de modelo: estándar de color: amarillo de potencia: 6 de espacio: 3,2
Automóvil de hidrogeno de modelo: estándar de color: amarillo de potencia: 7 de espacio: 3,2
Automóvil de hidrogeno de modelo: estándar de color: amarillo de potencia: 8 de espacio: 3,2
Scooter hidrogeno de modelo: clásico de color: rojo de potencia: 2
Scooter hidrogeno de modelo: clásico de color: rojo de potencia: 3
```

Mostramos el nuevo diagrama de clases resultante de la inclusión de este nuevo tipo de vehículo:

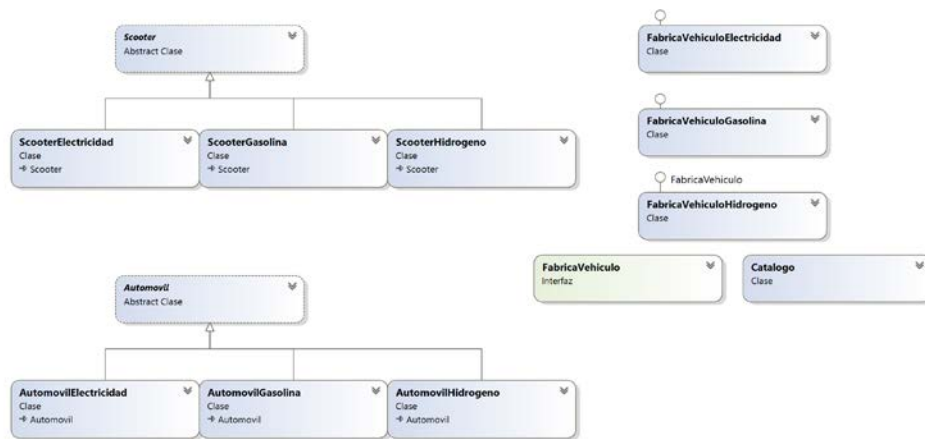


Diagrama de clases de la solución específica ampliada para el patrón Abstract Factory.

2.2. Builder

2.2.1. Funcionamiento (base: BuilderConsoleApp)

En primer lugar, mostramos el diagrama de clases de la solución que presenta el patrón a estudiar Builder.

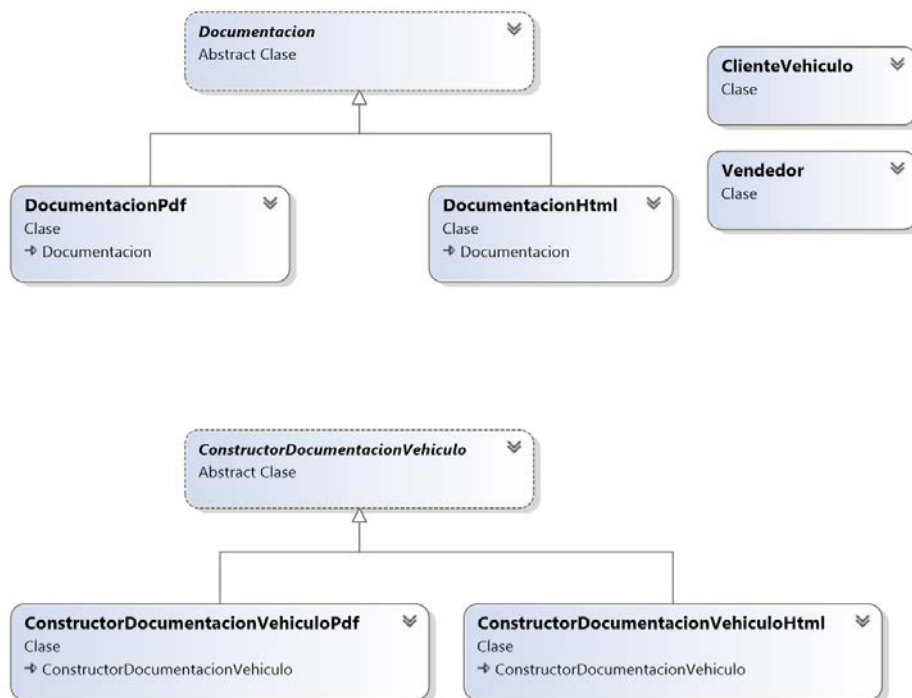



Diagrama de clases de la solución específica para el patrón Builder.


Como podemos ver, la lógica a seguir es parecida a la de Abstract Factory. En este caso, el patrón separa la construcción de un objeto (en este caso, la documentación de un vehículo que un cliente adquiere a un vendedor). La documentación puede ser generada en HTML o en PDF, según la preferencia del cliente. La clase ConstructorDocumentaciónVehículo es la encargada de generar esa documentación para el Vendedor. Para ello, ConstructorDocumentaciónVehículo es la encargada de construir la solicitud de pedido, matriculación, y además, de generar la documentación mediante la clase Documentación. Tanto el constructor como la clase que genera la documentación, son clases abstractas, que dependen de la representación de sus clases hijas para generar una documentación según las preferencias del cliente.

2.2.2. Ejecución

En la ejecución del programa vemos como introduciendo la opción 1, genera la documentación en HTML, obteniendo los datos de pedido, y de solicitante en el formato adecuado. Mientras, con la opción 2, generamos lo mismo, pero en formato PDF.

 C:\Users\Zéin\Desktop\DRA\Camacho Hidalgo Juan José (jch004)\Grupo Trabajo\Practica_02\PatronesDis

```
Desea generar documentación HTML (1) o PDF (2):  
1  
Documentación HTML  
<HTML>Solicitud de pedido Cliente: Martín</HTML>  
<HTML>Solicitud de matriculación Solicitante: Martín</HTML>
```

 C:\Users\Zéin\Desktop\DRA\Camacho Hidalgo Juan José (jch004)\Grupo Trabajo\Practica

```
Desea generar documentación HTML (1) o PDF (2):  
2  
Documentación PDF  
<PDF>Solicitud de pedido Cliente: Martín</PDF>  
<PDF>Solicitud de matriculación Solicitante: Martín</PDF>
```

2.2.3. Ampliación

Como comprobación de todo lo mencionado anteriormente, hemos añadido otro tipo de formato. En este caso hemos añadido el formato RTF (Rich Text Format) . Como vemos, sin modificar las clases abstractas, conseguimos crear otro tipo de formato y justificar así el uso del patrón eficazmente ya que nos permite que la empresa o vendedor añada cualquier formato sin preocuparse de las diferencias en la implementación, así como permitir el desconocimiento del cliente por parte de la estructura interna.

C:\WINDOWS\system32\cmd.exe

Desea generar documentación HTML (1) , RTF (2) o PDF (3):

2

Documentación RTF

<RTF>Solicitud de pedido Cliente: Martín</RTF>

<RTF>Solicitud de matriculación Solicitante: Martín</RTF>

Mostramos el nuevo diagrama de clases resultante de la inclusión de este nuevo tipo de formato:

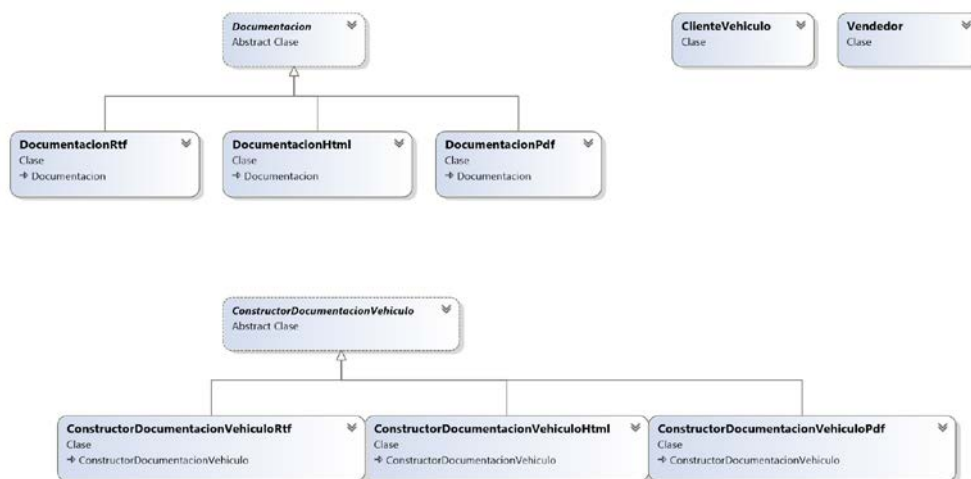
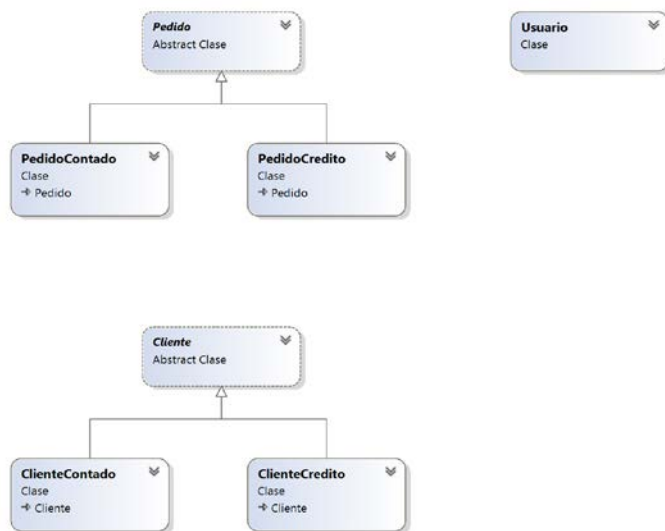


Diagrama de clases de la solución específica ampliada para el patrón Builder.

2.3. Factory Method


2.3.1. Funcionamiento (base: FactoryMethodConsoleApp)

Una clase **Usuario**, que actúa como cliente de una clase abstracta, crea instancias de subclases, en este caso **ClienteContado**, y **ClienteCredito**, que no conoce. Estas a su vez llaman a **Pedido**, de las cuales tampoco conoce sus subclases, ya que es abstracto también. El cliente genera un pedido mediante distintos tipos de pago. El patrón se basa en dar a conocer cuando se crea, pero no de qué tipo se crea. Se pueden añadir de esta forma, más tipos de pago, y además, no solo se pueden crear pedidos, sino que podemos también añadir solicitudes de distintos tipos, por ejemplo, de forma abstracta como en este programa.



2.3.2. Ejecución


En la ejecución vemos como realiza tres pedidos, dos al contado y uno a crédito. En el código realiza cuatro pedidos, pero solo se muestran tres ya que no se puede realizar un pedido a crédito de 10000, y no permite que se realice.

 C:\WINDOWS\system32\cmd.exe

```
El pago del pedido por importe de: 2000 se ha realizado.
El pago del pedido por importe de: 10000 se ha realizado.
El pago del pedido a crédito de: 2000 se ha realizado.
```

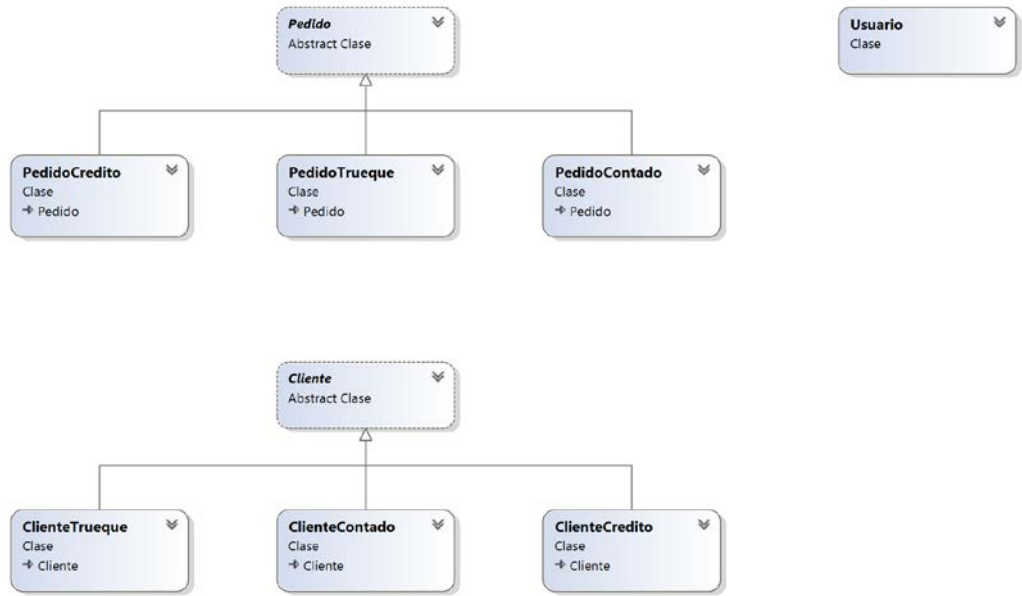
2.3.3. Ampliación

He realizado una ampliación añadiendo un tipo de pedido nuevo, en este caso a trueque por Bitcoins. Recibo un bien o bienes, y cambio se realiza un trueque por bitcoins. Podría haber añadido solicitudes, por ejemplo, con subtipos de solicitudes, pero cuando se añade un tipo nuevo de pedidos se muestra más claramente el funcionamiento del patrón.

 C:\WINDOWS\system32\cmd.exe

```
El pago del pedido por importe de: 2000 se ha realizado.
El pago del pedido por importe de: 10000 se ha realizado.
El pago del pedido a crédito de: 2000 se ha realizado.
El pago del pedido a trueque de: 2000 bitcoins se ha realizado.
El pago del pedido a trueque de: 10000 bitcoins se ha realizado.
```


Mostramos el nuevo diagrama de clases resultante de la inclusión de este nuevo tipo de pedido:

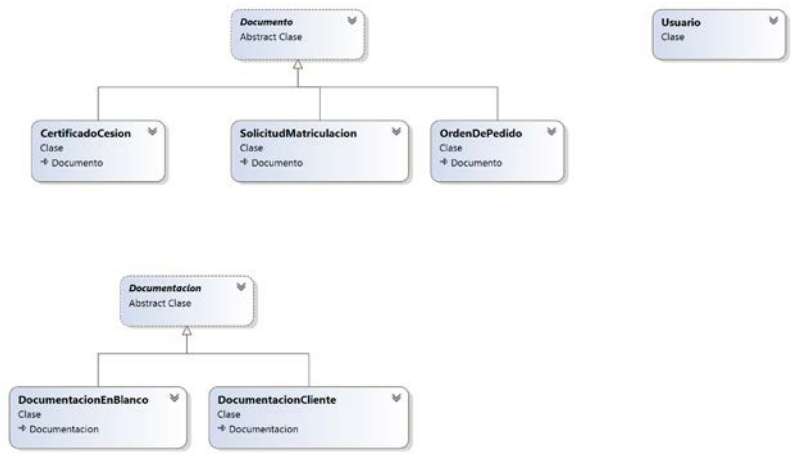


2.4. Prototype

2.4.1. Funcionamiento (base: PrototypeConsoleApp)

Este patrón se encarga de realizar duplicados de los objetos (prototipos) y mediante este sistema clona un esquema realizado para un objeto creado anteriormente, creando uno con distinto contenido pero misma superficie, dicho en lenguaje algo común.

Para profundizar esto, vamos a usar este ejemplo, mostrado en el diagrama de clases:



Usuario es la clase control del patrón. Esta necesita crear la documentación referente a varios clientes. Para generarla, tiene un esquema ya creado: Se genera DocumentacionEnBlanco, y sobre esta, se genera la DocumentacionCliente. Cada documentación del cliente, integra CertificadoCesion, SolicitudMatriculacion, y OrdenDePedido. Por tanto, cada vez que genere una documentación para el cliente, usará ese esquema, que como se entiende, para cada persona será distinta, pero tendrá la misma estructura y el mismo tipo de datos.

Esto reduce la cantidad de clases creadas para la generación de documentación, y nos permite modificar la estructura para todos los clientes a nuestro gusto.

2.4.2. Ejecución

En la ejecución del programa vemos como se genera la misma estructura organizativa mostrando la documentación de dos clientes: Martín y Simón.

```
C:\WINDOWS\system32\cmd.exe
Muestra la orden de pedido: Martín
Muestra el certificado de cesión: Martín
Muestra la solicitud de matriculación: Martín
Muestra la orden de pedido: Simón
Muestra el certificado de cesión: Simón
Muestra la solicitud de matriculación: Simón
```

2.4.3. Ampliación

Para comprobar el buen funcionamiento del patrón, y justificar lo mencionado anteriormente, extendemos el programa, añadiendo una documentación de un cliente nuevo: Juan. Como vemos, la estructura se trata de la misma:

```
C:\WINDOWS\system32\cmd.exe
Muestra la orden de pedido: Martín
Muestra el certificado de cesión: Martín
Muestra la solicitud de matriculación: Martín
Muestra la orden de pedido: Simón
Muestra el certificado de cesión: Simón
Muestra la solicitud de matriculación: Simón
Muestra la orden de pedido: Juan
Muestra el certificado de cesión: Juan
Muestra la solicitud de matriculación: Juan
```

2.5. Singleton

2.5.1. Funcionamiento (base: SingletonConsoleApp)

Este patrón tiene como objetivo asegurar que la clase, en este caso Comercial, tiene una única instancia en la aplicación, y asegurar un punto de acceso.



Diagrama de clases de SingletonConsoleApp.

Para ello, y como podemos observar, el constructor de la clase Comercial es privado. Esto restringe el acceso desde la clase que hace uso, TestComercial, e impide la creación del objeto mediante new. Para obtener la instancia del Comercial, se crea un método estático en Comercial, Instance(), para poder asegurar ese acceso, y permitir esa única instancia.

```
0 referencias | Camacho Hidalgo, Juan Jose, Hace 25 días | 1 autor, 1 cambio
public class TestComercial
{
    0 referencias | Camacho Hidalgo, Juan Jose, Hace 25 días | 1 autor, 1 cambio
    static void Main(string[] args)
    {
        // inicialización del comercial en el sistema
        Comercial elComercial = Comercial.Instance();
        elComercial.nombre = "Comercial Auto";
        elComercial.direccion = "Madrid";
        elComercial.email = "comercial@comerciales.com";
        // muestra el comercial del sistema
        visualiza();

        Console.ReadLine();
    }

    1 referencia | Camacho Hidalgo, Juan Jose, Hace 25 días | 1 autor, 1 cambio
    public static void visualiza()
    {
        Comercial elComercial = Comercial.Instance();
        elComercial.visualiza();
    }
}
```

```

public class Comercial
{
    2 referencias | Camacho Hidalgo, Juan Jose, Hace 25 días | 1 autor, 1 cambio
    public string nombre { get; set; }
    2 referencias | Camacho Hidalgo, Juan Jose, Hace 25 días | 1 autor, 1 cambio
    public string direccion { get; set; }
    2 referencias | Camacho Hidalgo, Juan Jose, Hace 25 días | 1 autor, 1 cambio
    public string email { get; set; }

    private static Comercial _instance = null;

    1 referencia | Camacho Hidalgo, Juan Jose, Hace 25 días | 1 autor, 1 cambio
    private Comercial() { }

    2 referencias | Camacho Hidalgo, Juan Jose, Hace 25 días | 1 autor, 1 cambio
    public static Comercial Instance()
    {
        if (_instance == null)
            _instance = new Comercial();
        return _instance;
    }

    1 referencia | Camacho Hidalgo, Juan Jose, Hace 25 días | 1 autor, 1 cambio
    public void visualiza()
    {
        Console.WriteLine("Nombre: " + nombre);
        Console.WriteLine("Dirección: " + direccion);
        Console.WriteLine("Email: " + email);
    }
}

```

2.5.2. Ejecución

La captura de la ejecución del programa nos justifica la eficacia del patrón, obteniendo solo esa única instancia del comercial.



C:\WINDOWS\system32\cmd.exe

```

Nombre: Comercial Auto
Dirección: Madrid
Email: comercial@comerciales.com

```