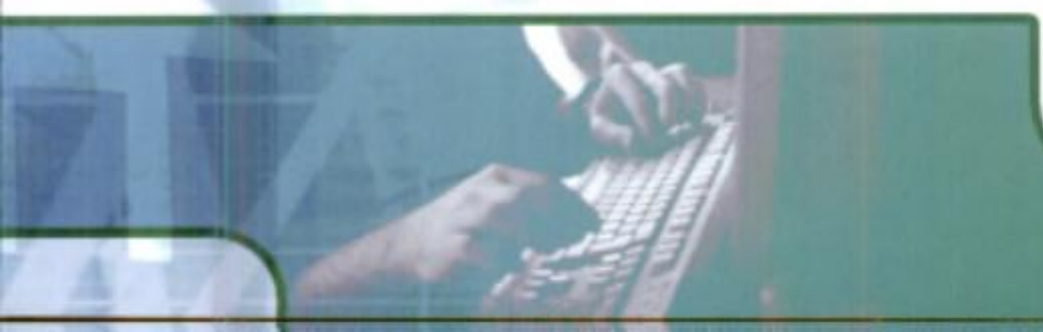
Colección EXPERT IT

Patrones de diseño para C#

Los 23 modelos de diseño: descripción y soluciones ilustradas en UML 2 y C#

Laurent DEBRAUWER

Descarga
www.ediciones-eni.com



eni ediciones

INFORMÁTICA TÉCNICA

Colección
EXPERT IT

Patrones de diseño para C#

Los 23 modelos de diseño: descripción y soluciones ilustradas en UML 2 y C#

ISBN: 978-2-7460-7260-2
Edición original: 978-2-7460-6753-0

Ediciones ENI
P° Ferrocarriles Catalanes, 97-117, 2a pl. of. 18
08940 - Cornellà de Llobregat (Barcelona)

Tel: 934 246 401
Fax: 934 231 576

e-mail: info@ediciones-eni.com
http://www.ediciones-eni.com

Autor: Laurent DEBRAUWER
Edición española: Francisco Javier PIQUERES JUAN
Colección **Expert IT** dirigida por Joëlle MUSSET

4 Patrones de diseño para C#

Los 23 modelos de diseño

Parte 3: Patrones de estructuración

Capítulo 9
Introducción a los patrones de estructuración

Capítulo 10
El patrón Adapter

Capítulo 11
El patrón Bridge

Capítulo 12
El patrón Composite

Capítulo 13
El patrón Decorator

Capítulo 14
El patrón Facade

Capítulo 25
El patrón State

Capítulo 26
El patrón Strategy

Capítulo 27
El patrón Template Method

Capítulo 30
Los patrones en el diseño de aplicaciones

Anexos
Ejercicios

14 Patrones de diseño para C#

Los 23 modelos de diseño

Capítulo 1
Introducción a los patrones de diseño

1. Design patterns o patrones de diseño

Un design pattern o patrón de diseño consiste en un diagrama de objetos que forma una solución a un problema conocido y frecuente. El diagrama de objetos está constituido por un conjunto de objetos descritos por clases y las relaciones que enlazan los objetos.

Los patrones responden a problemas de diseño de aplicaciones en el marco de la programación orientada a objetos. Se trata de soluciones conocidas y probadas cuyo diseño proviene de la experiencia de los programadores. No existe un aspecto teórico en los patrones, en especial no existe una formalización (a diferencia de los algoritmos).

Los patrones de diseño están basado en las buenas prácticas de la programación orientada a objetos. Por ejemplo, la figura 1.1 muestra el patrón `Template method` que se describe en el capítulo El patrón Template Method. En este patrón, el método `calculaPrecioConIVA` invoca al método `calculaIVA` abstracto de la clase `Pedido`. Está definido en las subclases de `Pedido`, a saber las clases `PedidoEspaña` y `PedidoFrancia`. En efecto, el IVA varía en función del país. Al método `calculaPrecioConIVA` se le llama "patrón" (`Template method`). Introduce un algoritmo basado en un método abstracto.

2. Descripción de los patrones de diseño

Hemos decidido describir los patrones de diseño con ayuda de los siguientes lenguajes:

- el lenguaje de modelización UML introducido por el OMG (http://www.omg.org);
- el lenguaje de programación C# creado por la empresa Microsoft.

Los patrones de diseño se describen en las partes 2 a 4. Para cada patrón se presentan los siguientes elementos:

- el nombre del patrón;
- la descripción del patrón;
- un ejemplo describiendo el problema y la solución basada en el patrón descrito mediante un diagrama de clases UML. En este diagrama, se describe el cuerpo de los métodos utilizando notas;
- la estructura genérica del patrón, a saber:
  - su esquema, extraído de cualquier contexto particular, bajo la forma de un diagrama de clases UML;
  - la lista de participantes del patrón;
  - las colaboraciones en el patrón;
- los dominios de la aplicación del patrón;
- un ejemplo, presentado esta vez bajo la forma de un programa C# completo y documentado. Este programa no utiliza una interfaz gráfica sino exclusivamente las entradas/salidas por pantalla y teclado.

3. Catálogo de patrones de diseño

En este libro se presentan los veintitrés patrones de diseño descritos en el libro de referencia del "GoF". Estos patrones son diversas respuestas a problemas conocidos de la programación orientada a objetos. La lista que sigue no es exhaustiva y es resultado, como hemos explicado, de la experiencia.

- **Abstract Factory**: tiene como objetivo la creación de objetos reagrupados en familias sin tener que conocer las clases concretas destinadas a la creación de estos objetos;
- **Builder**: permite separar la construcción de objetos complejos de su implementación de modo que un cliente pueda crear estos objetos complejos con implementaciones diferentes;
- **Factory Method**: tiene como objetivo presentar un método abstracto para la creación de un objeto reportando a las subclases concretas la creación efectiva;
- **Prototype**: permite crear nuevos objetos por duplicación de objetos existentes llamados prototipos que disponen de la capacidad de clonación;
- **Singleton**: permite asegurar que de una clase concreta existe una única instancia y proporciona un método único que la devuelve;
- **Adapter**: tiene como objetivo convertir la interfaz de una clase existente en la interfaz esperada por los clientes también existentes para que puedan trabajar de forma conjunta;
- **Bridge**: tiene como objetivo separar los aspectos conceptuales de una jerarquía de clases de su implementación;
- **Composite**: proporciona un marco de diseño de una composición de objetos con una profundidad de composición variable, basando el diseño en un árbol;
- **Decorator**: permite agregar dinámicamente funcionalidades suplementarias a un objeto;
- **Facade**: tiene como objetivo reagrupar las interfaces de un conjunto de objetos en una interfaz unificada que resulte más fácil de utilizar;
- **Flyweight**: facilita la compartición de un conjunto importante de objetos con granularidad muy fina;

- **Proxy**: construye un objeto que se substituye por otro objeto y que controla su acceso;
- **Chain of responsibility**: crea una cadena de objetos tal que si un objeto de la cadena no puede responder a una petición, la pueda transmitir a sus sucesores hasta que uno de ellos responda;
- **Command**: tiene como objetivo transformar una consulta en un objeto, facilitando operaciones como la anulación, la actualización de consultas y su seguimiento;
- **Interpreter**: proporciona un marco para dar una representación mediante objetos de la gramática de un lenguaje con el objetivo de evaluar, interpretándolas, expresiones escritas en este lenguaje;
- **Iterator**: proporciona un acceso secuencial a una colección de objetos sin que los clientes se preocupen de la implementación de esta colección;
- **Mediator**: construye un objeto cuya vocación es la gestión y el control de las interacciones en el seno de un conjunto de objetos sin que estos elementos se conozcan mutuamente;
- **Memento**: salvaguarda y restaura el estado de un objeto;
- **Observer**: construye una dependencia entre un sujeto y sus observadores de modo que cada modificación del sujeto sea notificada a los observadores para que puedan actualizar su estado;
- **State**: permite a un objeto adaptar su comportamiento en función de su estado interno;
- **Strategy**: adapta el comportamiento y los algoritmos de un objeto en función de una necesidad concreta sin por ello cargar las interacciones con los clientes de este objeto;
- **Template Method**: permite reportar en las subclases ciertas etapas de una de las operaciones de un objeto, estando éstas descritas en las subclases;
- **Visitor**: construye una operación a realizar en los elementos de un conjunto de objetos. Es posible agregar nuevas operaciones sin modificar las clases de estos objetos.

A continuación se muestra un ejemplo de adaptación del patrón `Template method` a partir del ejemplo presentado en la primera sección de este capítulo. La estructura genérica de este patrón se muestra en la figura 1.2.

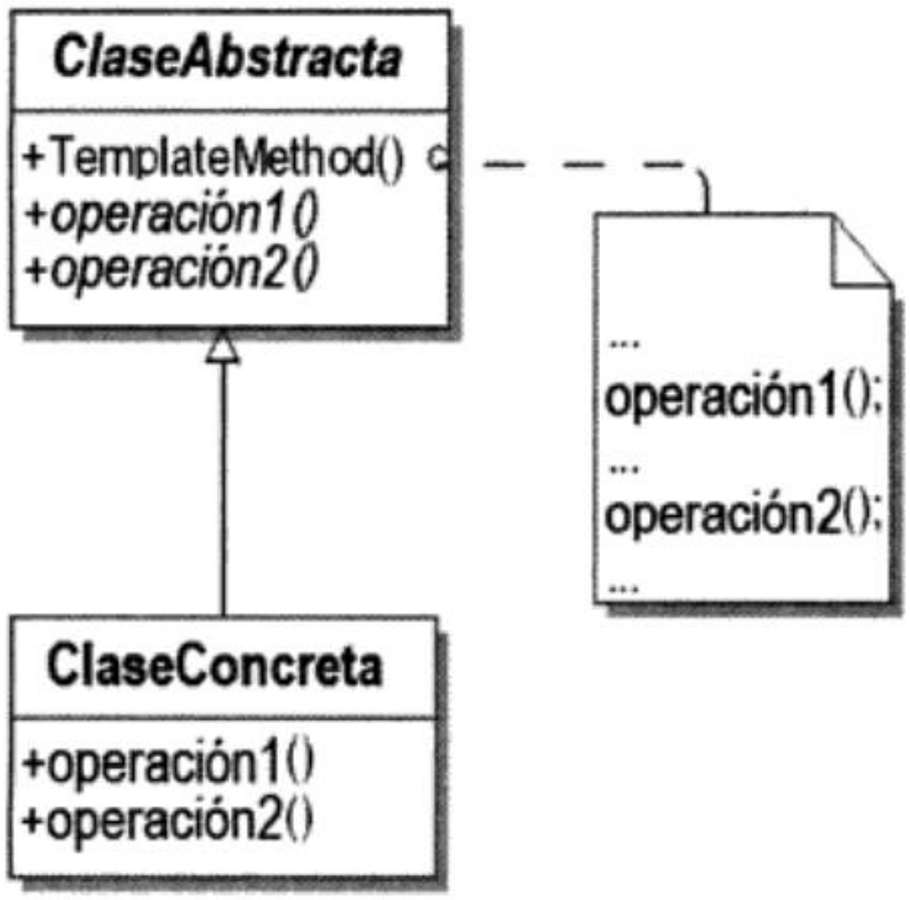


Figura 1.2 - Estructura genérica del patrón Template Method

Queremos adaptar esta estructura en el marco de una aplicación comercial donde el método de cálculo del IVA no esté incluido en la clase `Pedido` sino en las subclases concretas de la clase abstracta `Pais`. Estas subclases contienen todos los métodos de cálculo de los impuestos específicos de cada país.

El método `calculaIVA` de la clase `Pedido` invocará a continuación al método `calculaIVA` de la subclase del país afectado mediante una instancia de esta subclase. Esta instancia puede pasarse como parámetro en la creación del pedido.

La figura 1.3 ilustra el patrón adaptado listo para su uso en la aplicación.

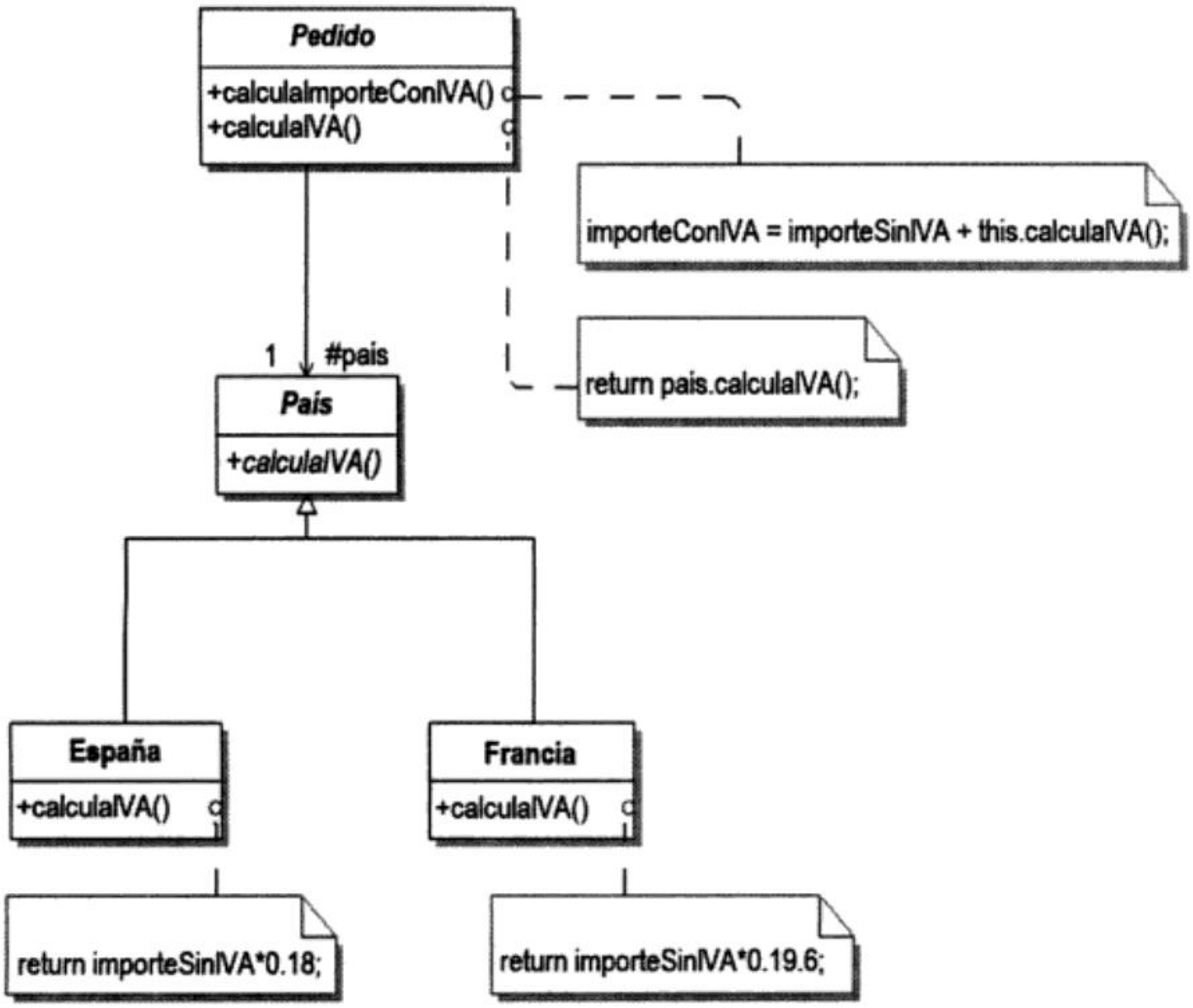


Figura 1.3 - Ejemplo de uso con adaptación del patrón Template Method

5. Organización del catálogo de patrones de diseño

Para organizar el catálogo de patrones de diseño, retomamos la clasificación del "GoF" que organiza los patrones según su vocación: construcción, estructuración y comportamiento.

Los patrones de construcción tienen como objetivo organizar la creación de objetos. Se describen en la parte 2 del libro. Son un total de cinco: Abstract Factory, Builder, Factory Method, Prototype y Singleton.

Capítulo 2
Caso de estudio: venta online de vehículos

1. Descripción del sistema

En este libro tomaremos un ejemplo de diseño de un sistema para ilustrar el uso de los veintitrés patrones de diseño.

El sistema que vamos a diseñar es un sitio Web de venta online de vehículos como, por ejemplo, automóviles o motocicletas. Este sistema autoriza distintas operaciones como la visualización de un catálogo, la recogida de un pedido, la gestión y el seguimiento de los clientes. Además estará accesible bajo la forma de un servicio Web.

2. Cuaderno de carga

El sitio permite visualizar un catálogo de vehículos puestos a la venta, realizar búsquedas en el catálogo, realizar el pedido de un vehículo, seleccionar las opciones para el mismo mediante un sistema de carro de la compra virtual. Las opciones incompatibles también deben estar gestionadas (por ejemplo "asientos deportivos" y "asientos en cuero" son opciones incompatibles). También es posible volver a un estado anterior del carrito de la compra.

El sistema debe administrar los pedidos. Debe ser capaz de calcular los impuestos en función del país de entrega del vehículo. También debe gestionar los pedidos pagados al contado y aquellos que están ligados a una petición de crédito. Para ello, se tendrá en cuenta las peticiones de crédito. El sistema administra los estados del pedido: en curso, validado y entregado.

Al realizar el pedido de un vehículo, el sistema construye el conjunto de documentos necesarios como la solicitud de matriculación, el certificado de cesión y la orden de pedido. Estos documentos estarán disponibles en formato PDF o en formato HTML.

El sistema también permite rebajar los vehículos de difícil venta, como por ejemplo aquellos que se encuentran en stock pasado un tiempo.

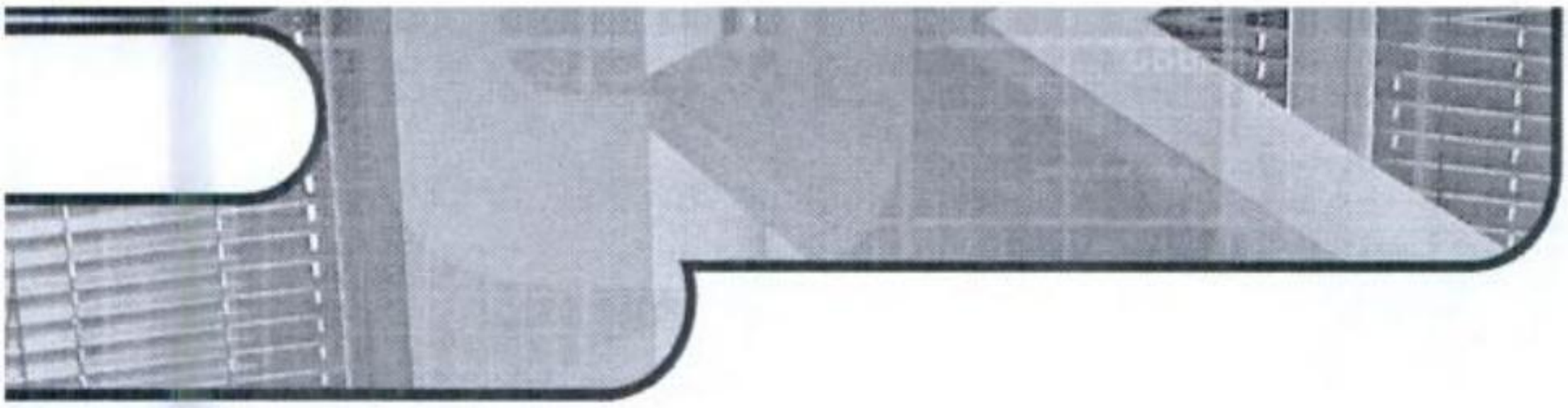
También permite realizar una gestión de los clientes, en particular de empresas que poseen filiales para proporcionarles, por ejemplo, la compra de una flota de vehículos.

Tras la virtualización del catálogo, es posible visualizar animaciones asociadas a un vehículo. El catálogo puede presentarse con uno o tres vehículos por cada línea de resultados.

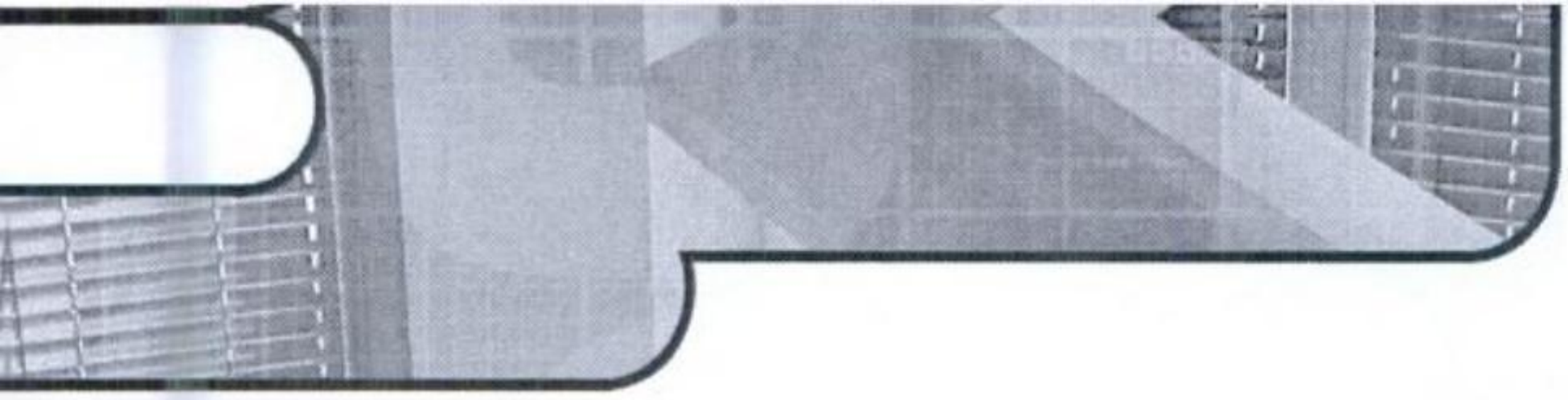
La búsqueda en el catálogo puede realizarse con ayuda de palabras clave y de operadores lógicos (y, o).

Es posible acceder al sistema mediante una interfaz Web clásica o a través de un sistema de servicios Web.

Descripción de la sección	Patrón de diseño
Enviar propuestas comerciales por correo electrónico a ciertas empresas clientes.	Visitor



Parte 2
Patrones de construcción



Capítulo 3
Introducción a los patrones de construcción

1. Presentación

Los patrones de construcción tienen la vocación de abstraer los mecanismos de creación de objetos. Un sistema que utilice estos patrones se vuelve independiente de la forma en que se crean los objetos, en particular, de los mecanismos de instanciación de las clases concretas.

Estos patrones encapsulan el uso de clases concretas y favorecen así el uso de las interfaces en las relaciones entre objetos, aumentando las capacidades de abstracción en el diseño global del sistema.

De este modo el patrón `Singleton` permite construir una clase que posee una instancia como máximo. El mecanismo que gestiona el acceso a esta única instancia está encapsulado por completo en la clase, y es transparente a los clientes de la clase.

2. Problemas ligados a la creación de objetos

2.1 Problemática

En la mayoría de lenguajes orientados a objetos, la creación de objetos se realiza gracias al mecanismo de instanciación que consiste en crear un nuevo objeto mediante la llamada al operador `new` configurado para una clase (y eventualmente los argumentos del constructor de la clase cuyo objetivo es proporcionar a los atributos su valor inicial). Tal objeto es, por consiguiente, una instancia de esta clase.

Los lenguajes de programación más utilizados a día de hoy, como C++ o C#, utilizan el mecanismo del operador `new`.

En C#, una instrucción de creación de un objeto puede escribirse de la siguiente manera:

```
objeto = new Clase();
```

En ciertos casos es necesario configurar la creación de objetos. Tomemos el ejemplo de un método `construyeDoc` que crea los documentos. Puede construir documentos PDF, RTF o HTML. Generalmente el tipo de documento a crear se pasa como parámetro al método mediante una cadena de caracteres, y se obtiene el código siguiente:

```
public Documento construyeDoc(string tipoDoc)
{
  Documento resultado;

  if (tipoDoc.Equals("PDF"))
    resultado = new DocumentoPDF();
  else if (tipoDoc.equals("RTF"))
    resultado = new DocumentoRTF();
  else if (tipoDoc.equals("HTML"))
    resultado = new DocumentoHTML();
  // continuación del método
}
```

Este ejemplo muestra que es difícil configurar el mecanismo de creación de objetos, la clase que se pasa como parámetro al operador `new` no puede sustituirse por una variable. El uso de instrucciones condicionales en el código del cliente a menudo resulta práctico, con el inconveniente de que un cambio en la jerarquía de las clases a instanciar implica modificaciones en el código de los clientes. En nuestro ejemplo, es necesario cambiar el código del método `construyeDoc` si se quiere agregar nuevos tipos de documento.

Observación

En lo sucesivo, ciertos lenguajes ofrecen mecanismos más o menos flexibles y a menudo bastante complejos para crear instancias a partir del nombre de una clase contenida en una variable de tipo `string`.

La dificultad es todavía mayor cuando hay que construir objetos compuestos cuyas componentes pueden instanciarse mediante clases diferentes. Por ejemplo, un conjunto de documentos puede estar formado por documentos PDF, RTF o HTML. El cliente debe conocer todas las clases posibles de las componentes y de las composiciones. Cada modificación en el conjunto de las clases se vuelve complicada de gestionar.

2.2 Soluciones propuestas por los patrones de construcción

Los patrones `Abstract Factory`, `Builder`, `Factory Method` y `Prototype` proporcionan una solución para parametrizar la creación de objetos. En el caso de los patrones `Abstract Factory`, `Builder` y `Prototype`, se utiliza un objeto como parámetro del sistema. Este objeto se encarga de realizar la instanciación de las clases. De este modo, cualquier modificación en la jerarquía de las clases sólo implica modificaciones en este objeto.

El patrón `Factory Method` proporciona una configuración básica sobre las subclases de la clase cliente. Sus subclases implementan la creación de los objetos. Cualquier cambio en la jerarquía de las clases implica por consiguiente una modificación de la jerarquía de las subclases de la clase cliente.

Capítulo 4
El patrón Abstract Factory

1. Descripción

El objetivo del patrón `Abstract Factory` es la creación de objetos agrupados en familias sin tener que conocer las clases concretas destinadas a la creación de estos objetos.

2. Ejemplo

El sistema de venta de vehículos gestiona vehículos que funcionan con gasolina y vehículos eléctricos. Esta gestión está delegada en el objeto `Catálogo` encargado de crear tales objetos.

Para cada producto, disponemos de una clase abstracta, de una subclase concreta derivando una versión del producto que funciona con gasolina y de una subclase concreta derivando una versión del producto que funciona con electricidad. Por ejemplo, en la figura 4.1, para el objeto Scooter, existe una clase abstracta `Scooter` y dos subclases concretas `ScooterElectricidad` y `ScooterGasolina`.

El objeto `Catálogo` puede utilizar estas subclases concretas para instanciar los productos. No obstante si fuera necesario incluir nuevas clases de familias de vehículos (diésel o mixto gasolina-eléctrico), las modificaciones a realizar en el objeto `Catálogo` pueden ser bastante pesadas.

El patrón `Abstract Factory` resuelve este problema introduciendo una interfaz `FábricaVehículo` que contiene la firma de los métodos para definir cada producto. El tipo devuelto por estos métodos está constituido por una de las clases abstractas del producto. De este modo el objeto `Catálogo` no necesita conocer las subclases concretas y permanece desacoplado de las familias de producto.

Se incluye una subclase de implementación de `FábricaVehículo` por cada familia de producto, a saber las subclases `FábricaVehículoElectricidad` y `FábricaVehículoGasolina`. Dicha subclase implementa las operaciones de creación del vehículo apropiado para la familia a la que está asociada.

El objeto `Catálogo` recibe como parámetro una instancia que responde a la interfaz `FábricaVehículo`, es decir o bien una instancia de `FábricaVehículoElectricidad`, o bien una instancia de `FábricaVehículoGasolina`. Con dicha instancia, el catálogo puede crear y manipular los vehículos sin tener que conocer las familias de vehículo y las clases concretas de instanciación correspondientes.

3. Estructura

3.1 Diagrama de clases

La figura 4.2 detalla la estructura genérica del patrón.

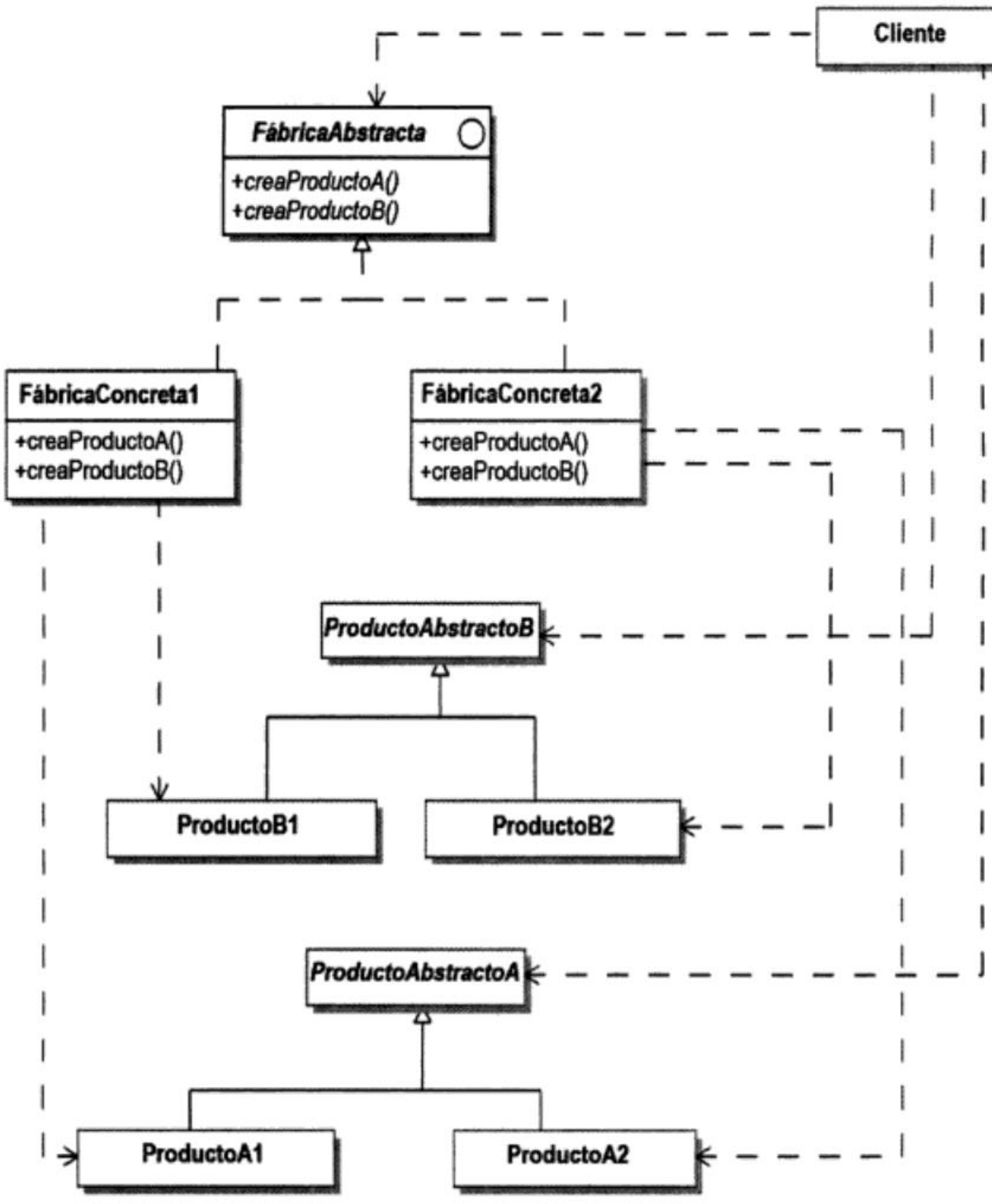


Figura 4.2 - Estructura del patrón `Abstract Factory`

3.2 Participantes

Los participantes del patrón son los siguientes:

- `FábricaAbstracta` (`FábricaVehículo`) es una interfaz que define las firmas de los métodos que crean los distintos productos;
- `FábricaConcreta1`, `FábricaConcreta2` (`FábricaVehículoElectricidad`, `FábricaVehículoGasolina`) son las clases concretas que implementan los métodos que crean los productos para cada familia de producto. Conociendo la familia y el producto, son capaces de crear una instancia del producto para esta familia;
- `ProductoAbstractoA` y `ProductoAbstractoB` (`Scooter` y `Automóvil`) son las clases abstractas de los productos independientemente de su familia. Las familias se introducen en las subclases concretas;
- `Cliente` es la clase que utiliza la interfaz de `FábricaAbstracta`.

3.3 Colaboraciones

La clase `Cliente` utiliza una instancia de una de las fábricas concretas para crear sus productos a partir de la interfaz `FábricaAbstracta`.

Observación

Normalmente sólo es necesario crear una instancia de cada fábrica concreta, que puede compartirse por varios clientes.

4. Dominios de uso

El patrón se utiliza en los dominios siguientes:

- Un sistema que utiliza productos necesita ser independiente de la forma en que se crean y agrupan estos productos;
- Un sistema está configurado según varias familias de productos que pueden evolucionar.

5. Ejemplo en C#

Presentamos a continuación un pequeño ejemplo de uso del patrón escrito en C#. El código C# correspondiente a la clase abstracta `Automovil` y sus subclases aparece a continuación. Es muy sencillo, describe los cuatro atributos de los automóviles así como el método `mostrarCaracteristicas` que permite visualizarlas.

```
using System;

public abstract class Automovil
{
  protected string modelo;
  protected string color;
  protected int potencia;
  protected double espacio;

  public Automovil(string modelo, string color, int
    potencia, double espacio)
  {
    this.modelo = modelo;
    this.color = color;
    this.potencia = potencia;
    this.espacio = espacio;
  }

  public abstract void mostrarCaracteristicas();
}


using System;

public class AutomovilElectricidad : Automovil
{
  public AutomovilElectricidad(string modelo, string
    color, int potencia, double espacio) : base(modelo,
    color, potencia, espacio){}

  public override void mostrarCaracteristicas()
  {
    Console.WriteLine(
      "Automóvil eléctrico de modelo: " + modelo +
      " de color: " + color + " de potencia: " +
```

```
using System;

public class ScooterElectricidad : Scooter
{
  public ScooterElectricidad(string modelo, string color,
    int potencia) : base(modelo, color, potencia){}

  public override void mostrarCaracteristicas()
  {
    Console.WriteLine("Scooter eléctrica de modelo: " +
      modelo + " de color: " + color +
      " de potencia: " + potencia);
  }

}


using System;

public class ScooterGasolina : Scooter
{
  public ScooterGasolina(string modelo, string color,
    int potencia) : base(modelo, color, potencia){}

  public override void mostrarCaracteristicas()
  {
    Console.WriteLine("Scooter eléctrica de modelo: " +
      modelo + " de color: " + color +
      " de potencia: " + potencia);
  }

}
```

Ahora podemos introducir la interfaz `FabricaVehiculo` y sus dos clases de implementación, una para cada familia (eléctrico/gasolina). Es fácil darse cuenta de que sólo las clases de implementación utilizan las clases concretas de los vehículos.

```
using System;

public interface FabricaVehiculo
{
  Automovil creaAutomovil(string modelo, string color,
    int potencia, double espacio);

  Scooter creaScooter(string modelo, string color, int
    potencia);
}


using System;

public class FabricaVehiculoElectricidad : FabricaVehiculo
{
  public Automovil creaAutomovil(string modelo, string
    color, int potencia, double espacio)
  {
    return new AutomovilElectricidad(modelo, color,
      potencia, espacio);
  }

  public Scooter creaScooter(string modelo, string
    color, int potencia)
  {
    return new ScooterElectricidad(modelo, color,
      potencia);
  }
}


using System;

public class FabricaVehiculoGasolina : FabricaVehiculo
{
  public Automovil creaAutomovil(string modelo, string
    color, int potencia, double espacio)
  {
    return new AutomovilGasolina(modelo, color,
      potencia, espacio);
  }

  public Scooter creaScooter(string modelo, string
```

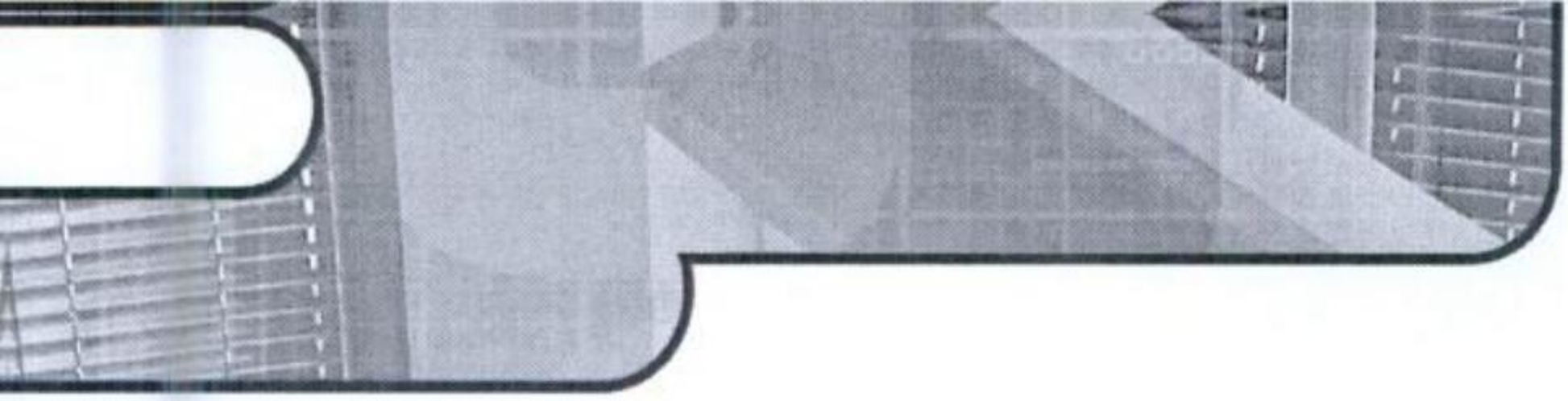
```
        foreach (Automovil auto in autos)
          auto.mostrarCaracteristicas();
        foreach (Scooter scooter in scooters)
          scooter.mostrarCaracteristicas();
    }

}
```

A continuación se muestra un ejemplo de ejecución para vehículos eléctricos:

```
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
```

En este ejemplo de ejecución se ha creado una fábrica de vehículos eléctricos, de modo que el catálogo se compone de automóviles y scooters eléctricos.



Capítulo 5
El patrón Builder

1. Descripción

El objetivo del patrón `Builder` es abstraer la construcción de objetos complejos de su implementación de modo que un cliente pueda crear objetos complejos sin tener que preocuparse de las diferencias en su implantación.

2. Ejemplo

Durante la compra de un vehículo, el vendedor crea todo un conjunto de documentos que contienen en especial la solicitud de pedido y la solicitud de matriculación del cliente. Es posible construir estos documentos en formato HTML o en formato PDF según la elección del cliente. En el primer caso, el cliente le provee una instancia de la clase `ConstructorDocumentaciónVehículoHtml` y, en el segundo caso, una instancia de la clase `ConstructorDocumentaciónVehículoPdf`. El vendedor realiza a continuación la solicitud de creación de cada documento mediante esta instancia.

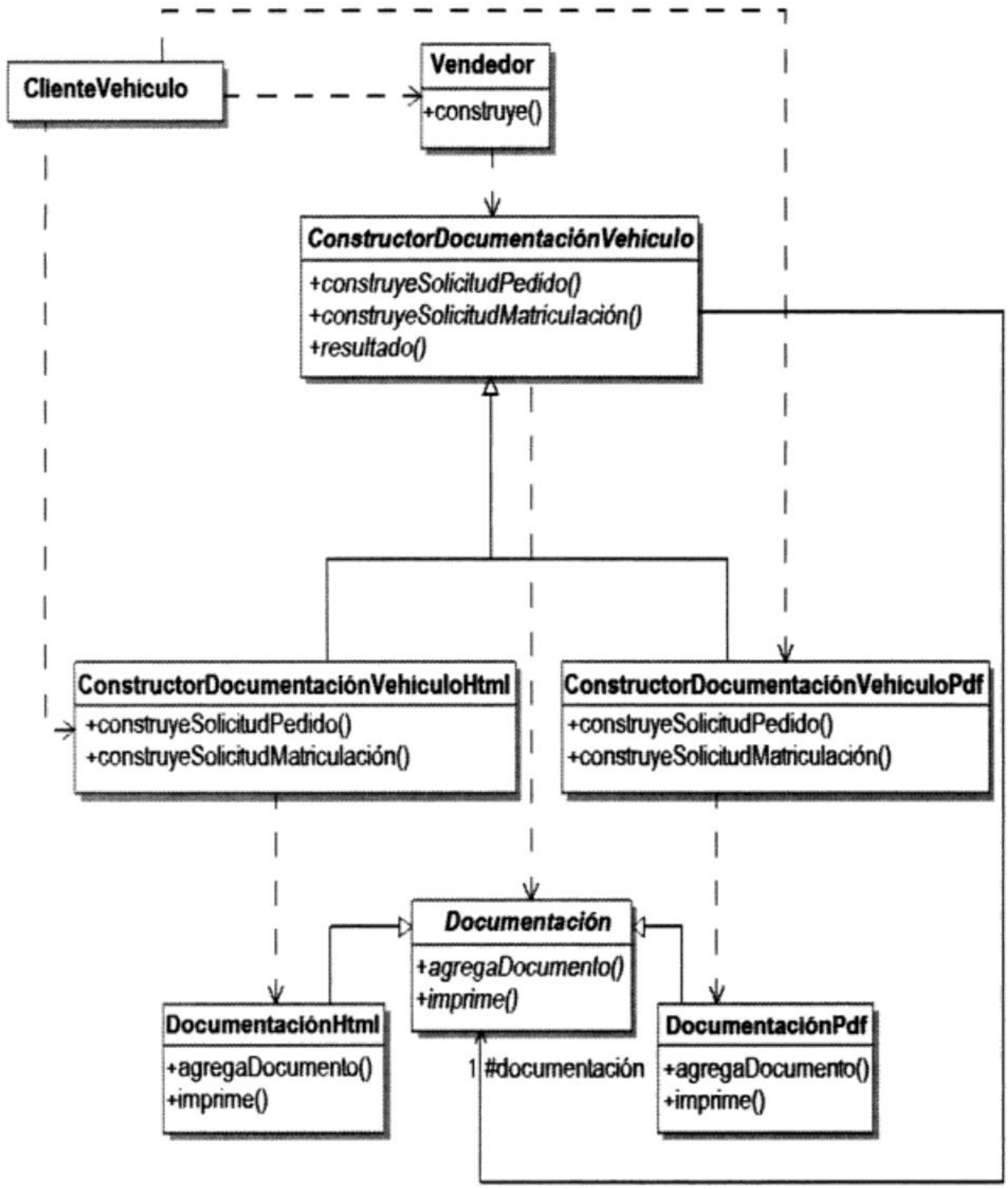


Figura 5.1 - El patrón `Builder` aplicado a la generación de documentación

3. Estructura

3.1 Diagrama de clases

La figura 5.2 detalla la estructura genérica del patrón.

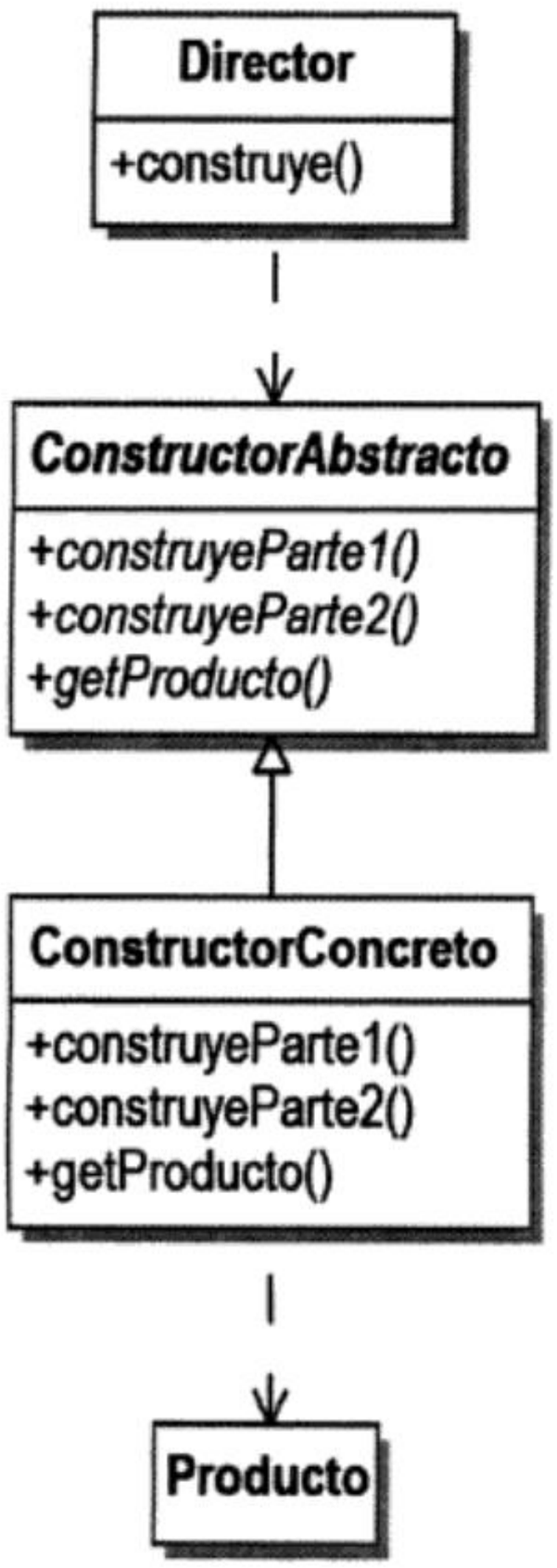


Figura 5.2 - Estructura del patrón `Builder`

La figura 5.3 ilustra este funcionamiento con un diagrama de secuencia UML.

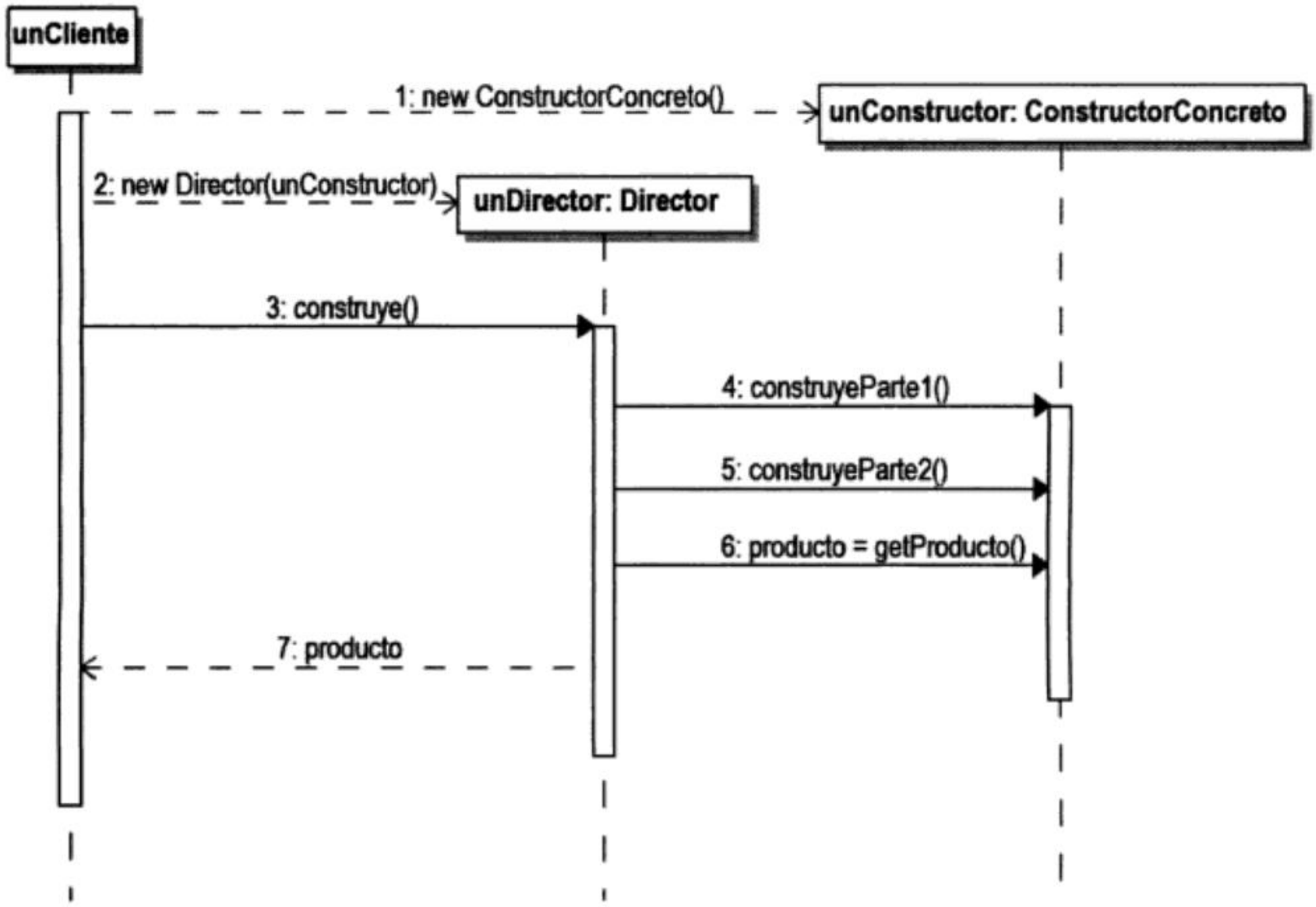


Figura 5.3 - Diagrama de secuencia del patrón `Builder`

4. Dominios de uso

El patrón se utiliza en los dominios siguientes:

- un cliente necesita construir objetos complejos sin conocer su implementación;
- un cliente necesita construir objetos complejos que tienen varias representaciones o implementaciones.

5. Ejemplo en C#

Presentamos a continuación un ejemplo de uso del patrón escrito en C#. El código C# correspondiente a la clase abstracta `Documentacion` y sus subclases aparece a continuación. Por motivos de simplicidad, los documentos son cadenas de caracteres para la documentación en formato HTML y PDF. El método `imprime` muestra las distintas cadenas de caracteres que representan los documentos.

```
using System;
using System.Collections.Generic;

public abstract class Documentacion
{
    protected IList<string> contenido =
        new List<string>();

    public abstract void agregaDocumento(string documento);
    public abstract void imprime();
}


using System;

public class DocumentacionHtml : Documentacion
{
    public override void agregaDocumento(string documento)
    {
        if (document.StartsWith("<HTML>"))
            contenido.Add(documento);
    }

    public override void imprime()
    {
        Console.WriteLine("Documentación HTML");
        foreach (string s in contenido)
            Console.WriteLine(s);
    }
}


using System;
```

```
public class DocumentacionPdf : Documentacion
{
    public override void agregaDocumento(string documento)
    {
        if (documento.StartsWith("<PDF>"))
            contenido.Add(documento);
    }

    public override void imprime()
    {
        Console.WriteLine("Documentación PDF");
        foreach (string s in contenido)
            Console.WriteLine(s);
    }
}
```

El código fuente de las clases que generan la documentación aparece a continuación.

```
using System;

public abstract class ConstructorDocumentacionVehiculo
{
    protected Documentacion documentacion;

    public abstract void construyeSolicitudPedido(string
      nombreCliente);

    public abstract void construyeSolicitudMatriculacion
      (string nombreSolicitante);

    public Documentacion resultado()
    {
        return documentacion;
    }
}


using System;

public class ConstructorDocumentacionVehiculoHtml :
  ConstructorDocumentacionVehiculo
{
    public ConstructorDocumentacionVehiculoHtml()
```

```
    public override void construyeSolicitudMatriculacion
      (string nombreSolicitante)
    {
        string documento;
        documento =
        "<PDF>Solicitud de matriculación Solicitante: " +
        nombreSolicitante + "</PDF>";
        documentacion.agregaDocumento(documento);
    }
}
```

La clase `Vendedor` se describe a continuación. Su constructor recibe como parámetro una instancia de `ConstructorDocumentacionVehiculo`. Observe que el método `construye` toma como parámetro la información del cliente, aquí limitada al nombre del cliente.

```
using System;

public class Vendedor
{
    protected ConstructorDocumentacionVehiculo constructor;

   public Vendedor(ConstructorDocumentacionVehiculo constructor)
    {
        this.constructor = constructor;
    }

    public Documentacion construye(string nombreCliente)
    {
        constructor.construyeSolicitudPedido(nombreCliente);
        constructor.construyeSolicitudMatriculacion
           (nombreCliente);
        Documentacion documentacion = constructor.resultado();
        return documentacion;
    }
}
```

Por último, se proporciona el código C# del cliente del constructor, a saber la clase `ClienteVehiculo` que constituye el programa principal. El inicio de este programa solicita al usuario el constructor que debe utilizar, y se lo proporciona a continuación al vendedor.

```
using System;

public class ClienteVehiculo
{
    static void Main(string[] args)
    {
        ConstructorDocumentacionVehiculo constructor;
        Console.WriteLine("Desea generar " +
          "documentación HTML (1) o PDF (2):");
        string seleccion = Console.ReadLine();
        if (seleccion == "1")
        {
            constructor = new
ConstructorDocumentacionVehiculoHtml();
        }
        else
        {
          constructor = new ConstructorDocumentacionVehiculoPdf();
        }
        Vendedor vendedor = new Vendedor(constructor);
        Documentacion documentacion =
vendedor.construye("Martín");
        documentacion.imprime();
    }
}
```

Un ejemplo de ejecución para una documentación PDF sería:

```
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>
```

Capítulo 6
El patrón Factory Method

1. Descripción

El objetivo del patrón `Factory Method` es proveer un método abstracto de creación de un objeto delegando en las subclases concretas su creación efectiva.

2. Ejemplo

Vamos a centrarnos en los clientes y sus pedidos. La clase `Cliente` implementa el método `creaPedido` que debe crear el pedido. Ciertos clientes solicitan un vehículo pagando al contado y otros clientes utilizan un crédito. En función de la naturaleza del cliente, el método `creaPedido` debe crear una instancia de la clase `PedidoContado` o una instancia de la clase `PedidoCrédito`. Para realizar estas alternativas, el método `creaPedido` es abstracto. Ambos tipos de cliente se distinguen mediante dos subclases concretas de la clase abstracta `Cliente`:

- la clase concreta `ClienteContado` cuyo método `creaPedido` crea una instancia de la clase `PedidoContado`;
- la clase concreta `ClienteCrédito` cuyo método `creaPedido` crea una instancia de la clase `PedidoCrédito`.

3.2 Participantes

Los participantes del patrón son los siguientes:

- `CreadorAbstracto` (`Cliente`) es una clase abstracta que implementa la firma del método de fabricación y los métodos que invocan al método de fabricación;
- `CreadorConcreto` (`ClienteContado`, `ClienteCrédito`) es una clase concreta que implementa el método de fabricación. Pueden existir varios creadores concretos;
- `Producto` (`Pedido`) es una clase abstracta que describe las propiedades comunes de los productos;
- `ProductoConcreto` (`PedidoContado`, `PedidoCrédito`) es una clase concreta que describe completamente un producto.

3.3 Colaboraciones

Los métodos concretos de la clase `CreadorAbstracto` se basan en la implementación del método de fabricación en las subclases. Esta implementación crea una instancia de la subclase adecuada de `Producto`.

4. Dominios de uso

El patrón se utiliza en los casos siguientes:

- una clase que sólo conoce los objetos con los que tiene relaciones;
- una clase quiere transmitir a sus subclases las elecciones de instanciación aprovechando un mecanismo de polimorfismo.

5. Ejemplo en C#

El código fuente de la clase abstracta `Pedido` y de sus dos subclases concretas aparece a continuación. El importe del pedido se pasa como parámetro al constructor de la clase. Si la validación de un pedido al contado es sistemática, tenemos la posibilidad de escoger, para nuestro ejemplo, aceptar únicamente aquellos pedidos provistos de un crédito cuyo valor se sitúe entre 1.000 y 5.000.

```
using System;

public abstract class Pedido
{
    protected double importe;

    public Pedido(double importe)
    {
        this.importe = importe;
    }

    public abstract bool valida();

    public abstract void paga();
}


using System;

public class PedidoContado : Pedido
{
    public PedidoContado(double importe) : base(importe) { }

    public override void paga()
    {
        Console.WriteLine(
          "El pago del pedido por importe de: " +
          importe + " se ha realizado.");
    }

    public override bool valida()
    {
        return true;
    }
```

```
}


using System;

public class PedidoCredito : Pedido
{
    public PedidoCredito(double importe) : base(importe) { }

    public override void paga()
    {
        Console.WriteLine(
          "El pago del pedido a crédito de: " +
          importe + " se ha realizado.");
    }

    public override bool valida()
    {
        return (importe >= 1000.0) && (importe <= 5000.0);
    }
}
```

El código fuente de la clase abstracta `Cliente` y de sus subclases concretas aparece a continuación. Un cliente puede realizar varios pedidos, y sólo los que se validan se agregan en su lista.

```
using System.Collections.Generic;

public abstract class Cliente
{
    protected IList<Pedido> pedidos =
        new List<Pedido>();

    protected abstract Pedido creaPedido(double importe);

    public void nuevoPedido(double importe)
    {
        Pedido pedido = this.creaPedido(importe);
        if (pedido.valida())
        {
            pedido.paga();
            pedidos.Add(pedido);
        }
    }
```

Un ejemplo de ejecución del usuario daría la salida siguiente:

```
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.
```

Se puede constatar que la solicitud de un pedido provisto de un crédito por valor de 10.000 ha sido rechazada.

Capítulo 7
El patrón Prototype

1. Descripción

El objetivo de este patrón es la creación de nuevos objetos mediante duplicación de objetos existentes llamados prototipos que disponen de la capacidad de clonación.

2. Ejemplo

Durante la compra de un vehículo, un cliente debe recibir una documentación compuesta por un número concreto de documentos tales como el certificado de cesión, la solicitud de matriculación o incluso la orden de pedido. Existen otros tipos de documentos que pueden incluirse o excluirse a esta documentación en función de las necesidades de gestión o de cambios de reglamentación. Introducimos una clase `Documentación` cuyas instancias son documentaciones compuestas por diversos documentos obligatorios. Para cada tipo de documento, incluimos su clase correspondiente.

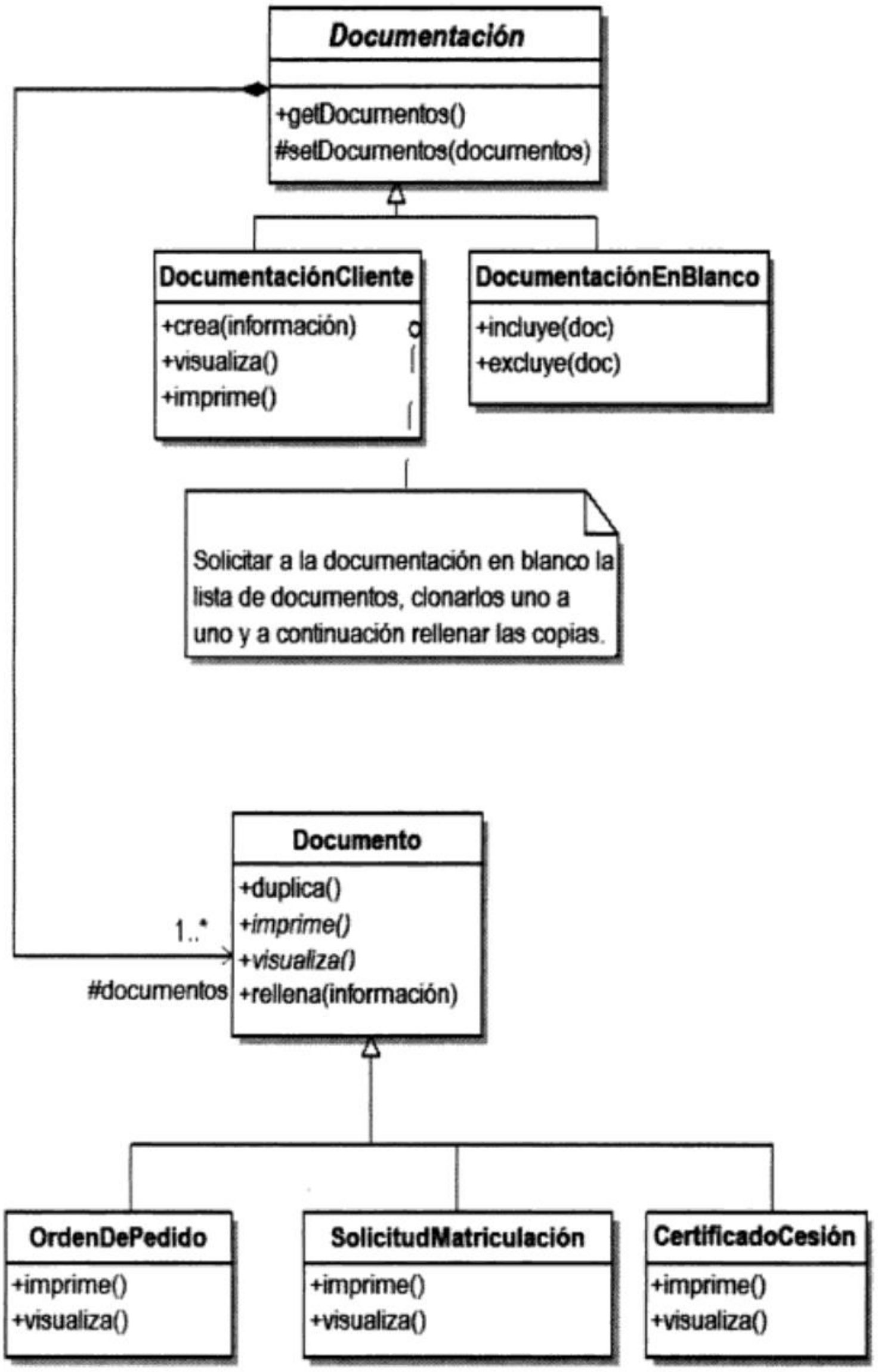


Figura 7.1 - El patrón `Prototype` aplicado a la creación de documentación de contenido variable

3.2 Participantes

Los participantes del patrón son los siguientes:

- `Cliente` (`Documentación`, `DocumentaciónCliente`, `DocumentaciónEnBlanco`) es una clase compuesta por un conjunto de objetos llamados prototipos, instancias de la clase abstracta `Prototype`. La clase Cliente necesita duplicar estos prototipos sin tener por qué conocer ni la estructura interna del `Prototype` ni su jerarquía de subclases.
- `Prototype` (`Documento`) es una clase abstracta de objetos capaces de duplicarse a sí mismos. Incluye la firma del método "duplica".
- `PrototypeConcreto1` y `PrototypeConcreto2` (`OrdenDePedido`, `SolicitudMatriculación`, `CertificadoCesión`) son las subclases concretas de `Prototype` que definen completamente un prototipo e implementan el método `duplica`.

3.3 Colaboración

El cliente solicita a uno o varios prototipos que se dupliquen a sí mismos.

4. Dominios de uso

El patrón `Prototype` se utiliza en los dominios siguientes:

- un sistema de objetos debe crear instancias sin conocer la jerarquía de clases que las describe;
- un sistema de objetos debe crear instancias de clases dinámicamente;
- el sistema de objetos debe permanecer simple y no incluir una jerarquía paralela de clases de fabricación.

5. Ejemplo en C#

El código fuente de la clase abstracta `Documento` y de sus subclases concretas aparece a continuación. Para simplificar, a diferencia del diagrama de clases, los métodos `duplica` y `rellena` se concretan en la clase `Documento`. El método `duplica` utiliza el método `MemberwiseClone` que proporciona C#.

```
using System;

public abstract class Documento
{
    protected string contenido = "";

    public Documento duplica()
    {
        Documento resultado;
        resultado = (Documento)this.MemberwiseClone();
        return resultado;
    }

    public void rellena(string informacion)
    {
        contenido = informacion;
    }

    public abstract void imprime();
    public abstract void visualiza();
}


using System;

public class OrdenDePedido : Documento
{
    public override void visualiza()
    {
        Console.WriteLine("Muestra la orden de pedido: " +
          contenido);
    }

    public override void imprime()
    {
```

```
        Console.WriteLine("Imprime la orden de pedido: " +
        contenido);
    }
}


using System;

public class SolicitudMatriculacion : Documento
{
  public override void visualiza()
  {
    Console.WriteLine(
      "Muestra la solicitud de matriculación: " + contenido);
  }

  public override void imprime()
  {
    Console.WriteLine(
      "Imprime la solicitud de matriculación: " + contenido);
  }
}


using System;

public class CertificadoCesion : Documento
{
  public override void visualiza()
  {
    Console.WriteLine(
      "Muestra el certificado de cesión: " + contenido);
  }

  public override void imprime()
  {
    Console.WriteLine(
      "Imprime el certificado de cesión: " + contenido);
  }
}
```

El código fuente de la clase abstracta `Documentacion` es el siguiente:

```
using System.Collections.Generic;

public abstract class Documentacion
{
    public IList<Documento> documentos { get; protected set; }
}
```

El código fuente de la subclase `DocumentacionEnBlanco` de `Documentacion` aparece a continuación. Este código utiliza el patrón `Singleton` que se presenta en el capítulo siguiente y que tiene como objetivo asegurar que una clase sólo posee una única instancia.

```
using System.Collections.Generic;

public class DocumentacionEnBlanco : Documentacion
{
    private static DocumentacionEnBlanco _instance = null;

    private DocumentacionEnBlanco()
    {
        documentos = new List<Documento>();
    }

    public static DocumentacionEnBlanco Instance()
    {
        if (_instance == null)
            _instance = new DocumentacionEnBlanco();
        return _instance;
    }

    public void incluye(Documento doc)
    {
        documentos.Add(doc);
    }

    public void excluye(Documento doc)
    {
        documentos.Remove(doc);
    }
}
```

Por último, veamos el código fuente de la clase `Usuario` cuyo método principal (main) comienza construyendo la documentación en blanco y, en particular, su contenido. A continuación este método crea y visualiza la documentación de ambos clientes.

```
using System;

public class Usuario
{
    static void Main(string[] args)
    {
        DocumentacionEnBlanco documentacionEnBlanco =
DocumentacionEnBlanco.Instance();
        documentacionEnBlanco.incluye(new OrdenDePedido());
        documentacionEnBlanco.incluye(new CertificadoCesion());
        documentacionEnBlanco.incluye(new
SolicitudMatriculacion());
        // creación de documentación nueva para dos clientes
        DocumentacionCliente documentacionCliente1 = new
DocumentacionCliente("Martín");
        DocumentacionCliente documentacionCliente2 = new
DocumentacionCliente("Simón");
        documentacionCliente1.visualiza();
        documentacionCliente2.visualiza();
    }
}
```

El resultado de la ejecución es el siguiente:

```
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
```

El sistema de documentación que debe entregarse al cliente tras la compra de un vehículo (como el certificado de cesión, la solicitud de matriculación y la orden de pedido) utiliza la clase `DocumentaciónEnBlanco` que sólo posee una instancia. Esta instancia referencia todos los documentos necesarios para el cliente. Esta instancia única se llama la documentación en blanco pues los documentos a los que hace referencia están todos en blanco. El uso completo de la clase `DocumentaciónEnBlanco` se explica en el capítulo dedicado al patrón `Prototype`.

La figura 8.1 ilustra el uso del patrón `Singleton` para la clase `DocumentaciónEnBlanco`. El atributo de clase `instance` contiene o bien `null` o bien la única instancia de la clase `DocumentaciónEnBlanco`. El método de clase `Instance` reenvía esta instancia única devolviendo el valor del atributo `instance`. Si este atributo vale `null`, se inicializa previamente mediante la creación de la instancia única.

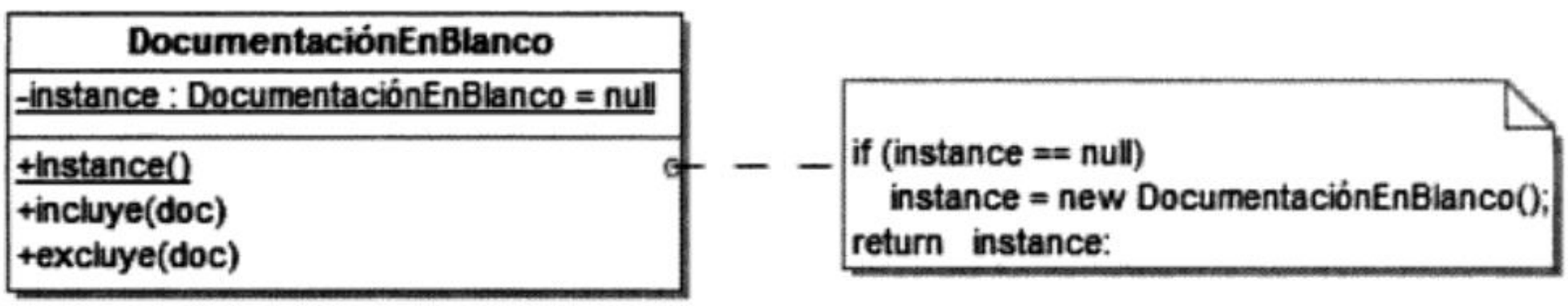


Figura 8.1 - El patrón `Singleton` aplicado a la clase `DocumentaciónEnBlanco`

3. Estructura

3.1 Diagrama de clases

La figura 8.2 detalla la estructura genérica del patrón.

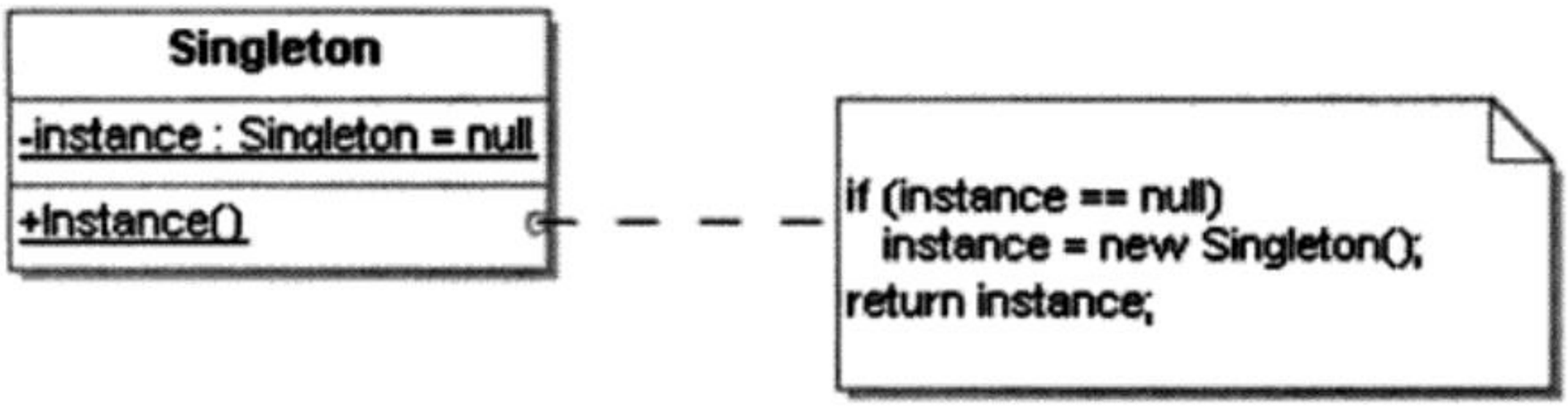


Figura 8.2 - Estructura del patrón `Singleton`

3.2 Participante

El único participante es la clase `Singleton` que ofrece acceso a la instancia única mediante el método de clase `Instance`.

Por otro lado, la clase `Singleton` posee un mecanismo que asegura que sólo puede existir una única instancia. Este mecanismo bloquea la creación de otras instancias.

3.3 Colaboración

Cada cliente de la clase `Singleton` accede a la instancia única mediante el método de clase `Instance`. No puede crear nuevas instancias utilizando el operador habitual de instanciación (operador `new`) que está bloqueado.

Laurent DEBRAUWER

Laurent Debrauwer es doctor en informática por la Universidad de Lille 1. Autor de programas en el dominio de la lingüística y de la semántica, ejerce como consultor independiente y especialista en orientación a objetos. Es profesor de Ingeniería del Software, Patrones de Diseño y Programación en Java en la Universidad de Luxemburgo.

En **www.ediciones-eni.com**:
➜ Código fuente de los ejemplos utilizados en el libro.

ISBN 978-2-7460-7260-2

Patrones de diseño para C#

Los 23 modelos de diseño: descripción soluciones ilustradas en UML 2 y C#

Este libro presenta de forma concisa y práctica los **23 mod los de diseño** (**Design Patterns**) fundamentales, ilustrán los mediante ejemplos adaptados y rápidos de comprende Cada ejemplo se describe **en UML y en C#** bajo la forma un pequeño programa completo y ejecutable. Para ca patrón, el autor detalla su nombre, el **problema correspo diente**, la **solución propuesta**, sus **dominios de aplicació** su **estructura genérica**.

El libro está dirigido a aquellos **diseñadores y desarrol dores que trabajen con Programación Orientada a Objet** Para comprenderlo bien, es preferible tener conocimient previos de los principales elementos de los diagramas clases UML y la última versión del lenguaje C# (a partir de versión 3.0). El libro está organizado en tres partes que corresponden con las tres familias de patrones de diseñ los **patrones de construcción**, los **patrones de estructu ción** y los **patrones de comportamiento**.

Un capítulo presenta tres variantes de patrones existent mostrando la gran flexibilidad existente a la hora de imp mentar estos modelos.

Los ejemplos utilizados en estas páginas son el resultado una aplicación de venta online de vehículos y pueden d cargarse en el sitio web www.ediciones-eni.com.

Los capítulos del libro

eni ediciones

www.ediciones-eni.com

Colección EXPERT IT