Capítulo 6
Serialización

1. Serialización en C#

Hoy en día, el lenguaje C# es uno de los más usados para desarrollar aplicaciones web. Uno de los problemas más frecuentes durante la comunicación en programación web es conseguir transferir objetos hacia y desde una aplicación. Para eso, el objeto se debe transformar en un formato universal. Esta transformación se llama serialización. Este proceso permite recuperar el objeto representado bajo un formato intercambiable; con frecuencia este formato tiene una forma textual.

Para que los datos puedan transitar, hay que usar un flujo de datos (llamado *stream* en inglés, de ahí el nombre de la clase base en C#: `Stream`). Estos flujos pueden tomar varias formas, como por ejemplo un flujo de datos en memoria (representado por la clase `MemoryStream` en C#) o incluso un flujo de datos hacia un archivo en el disco (representado por la clase `FileStream` en C#).

Hay varias maneras de serializar un objeto; estas son algunas de ellas:

- La serialización binaria, que permite representar un objeto en un formato binario.
- La serialización XML, que transforma el objeto en cadena de caracteres al formato XML.
- La serialización JSON, que transforma el objeto en cadena de caracteres al formato JSON.

En este capítulo vamos a abordar tres modos de serialización para transformar un objeto a un formato dado, pero también para poder recuperar un objeto desde una fuente de datos en el formato retenido.

2. Serialización binaria

El enfoque binario es el modo de serialización más simple para implantar. También es el que permite almacenar mayor cantidad de información en el tipo del objeto y tiene más fiabilidad de conversión de los valores y tipos almacenados. Sin embargo, su formato es propietario: no es portable ni compatible con otros lenguajes y soluciones, de manera que solo lo usaremos si el emisor y el destinatario son programas en C#.

Observación

Con la llegada de .NET 6 y C# 10, no se recomienda en absoluto usar este modo de serialización. Las explicaciones y el funcionamiento descritos en esta sección solo se proponen con fines de seguimiento, para los lectores que necesiten hacer el mantenimiento de un sistema usando este sistema de serialización. No se recomienda usar los objetos vistos en esta sección para aplicaciones nuevas porque están marcados como obsoletos en .NET 6 y se eliminarán en .NET 7.

Dos planteamientos permiten activar la serialización binaria:

– Usando los atributos apropiados en un tipo dado.

– Implementando la interfaz `ISerializable`.

Los atributos son más fáciles y rápidos de implantar, pero menos flexibles que la implementación de la interfaz `ISerializable`. Es recomendable elegir la mejor solución en función del objetivo.

2.1 Uso de los atributos

Este planteamiento es el más sencillo y rápido de implantar. Considerando la siguiente clase, solo hay que añadir el atributo `[Serializable]` encima de la declaración de la clase:

```
[Serializable]
public class Persona
{
    public string Nombre { get; set; }
    public string Apellido { get; set; }
}
```

De este modo, cuando se pida serializar una variable de tipo `Persona`, se informará al serializador, mediante la presencia del atributo, de que debe considerar la serialización de cada dato contenido en el interior del tipo, en forma de campos. Cada uno de los datos debe ser serializable; en caso contrario, se devolverá una excepción durante el proceso.

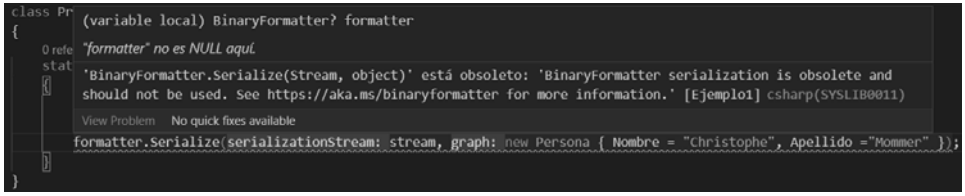
Una vez marcado el objeto, se puede usar el serializador binario, `BinaryFormatter`, que se encuentra en el espacio de nombres `System.Runtime.Serialization.Formatters.Binary`, para serializar el objeto en un stream de datos. El stream se puede guardar en memoria o en el sistema de archivos (este último caso es bastante habitual para guardar objetos en el disco entre dos ejecuciones de un programa). Se usa el método `Serialize` de este objeto para precisar el stream de destino, así como el objeto que se ha de serializar:

```
var formatter = new BinaryFormatter();
using(var stream = new FileStream("person.bin", FileMode. Create))
{
    formatter.Serialize(stream, new Persona { Nombre = "Christophe",
Apellido ="Mommer" });
}
```

Observación

La instrucción using delante de una variable se estudiará en el capítulo Conceptos avanzados, en la sección Gestión de la memoria.

Se comprueba que Visual Studio Code lanza un aviso de escritura de este código en .NET 6 porque el tipo ha sido devaluado y se eliminará en .NET 7, según la hoja de ruta de Microsoft:



Aviso en el editor con el uso del tipo BinaryFormatter

La clase `BinaryFormatter` también permite leer contenido desde un stream dado, gracias al método `Deserialize`. Se puede observar que este método devuelve una instancia de tipo `object`; por eso es necesario hacer un cast:

```
using(var reader = new FileStream("person.bin", FileMode.Open))
{
    var person = (Persona)formatter.Deserialize(reader);
    Console.WriteLine("Hola " + person.Nombre + " " + person.Apellido);
}
```

Dentro de la clase, se puede especificar que no se quiere serializar un dato concreto añadiendo el atributo `[NonSerialized]` encima del dato afectado.

Observación

Este atributo solo funciona en los campos y no en las propiedades, lo que demuestra una cierta debilidad del planteamiento de la serialización binaria.

```
[Serializable]
public class Persona
{
    public string Nombre { get; set; }
    public string Apellido { get; set; }
    [NonSerialized]
    public int Edad;
}

var formatter = new BinaryFormatter();
using (var stream = new FileStream("person.bin", FileMode.Create))
{
```

```
    formatter.Serialize(stream, new Persona { Nombre = " Christophe",
Apellido = "Mommer", Edad = 33 });
}

using (var reader = new FileStream("person.bin", FileMode.Open))
{
    var person = (Persona)formatter.Deserialize(reader);
    Console.WriteLine("Hola " + person.Nombre + " " + person.Apellido
+ ". Tiene " + person.Edad + " años"); // Mostrará Hola Christophe
Mommer. Tiene 0 años
}
```

El valor de un campo marcado como `NonSerialized` siempre será igual a `null` (en caso de tipo de referencia) o al valor predeterminado (en caso de tipo de valor), incluso si el constructor de la clase especifica un valor. El serializador binario es el único que no invoca al constructor de una clase en la deserialización.

Sin embargo, se puede definir un método al que se llamará durante la etapa de deserialización binaria. Este método no debe devolver nada y toma un único parámetro de tipo `StreamingContext`. Se añade el atributo `[OnDeserializing]` encima del método afectado:

```
[Serializable]
public class Persona
{
    public string Nombre { get; set; }
    public string Apellido { get; set; }
    [NonSerialized]
    public int Edad;

    [OnDeserializing]
    public void OnDeserializing(StreamingContext context)
    {
        Edad = 33; // aquí, se fija el valor 33 de manera permanente
en la deserialización
    }
}
```

También existen los siguientes atributos para crear métodos que se invocarán durante la (de)serialización:

– `[OnSerializing]`: se define en el método que se invoca durante la serialización.
– `[OnSerialized]`: se define en el método que se invoca cuando la serialización ha terminado.
– `[OnDeserialized]`: se define en el método que se invoca cuando la deserialización ha terminado.

Para finalizar la gestión por atributo, hay que recordar que el serializador binario puede ser sensible en caso de cambio del tipo serializado. En efecto, si el tipo de datos cambia entre la descripción de la clase y los datos serializados, se produce un error en la ejecución.

De la misma manera, si se ha añadido un dato, hay que mencionárselo al serializador porque, para él, deben estar presentes todos los datos no marcados como `[NonSerialized]`. Podemos usar el atributo `[OptionalField]` para especificar que el dato es en efecto opcional (dado que no estaba en una versión anterior). Este atributo también permite almacenar la versión donde se ha añadido el dato:

```
[Serializable]
public class Persona
{
    public string Nombre { get; set; }
    public string Apellido { get; set; }
    [NonSerialized]
    public int Edad;
    [OptionalField(VersionAdded = 2)]
    public DateTime FechaDeNacimiento;
}
```

2.2 Uso de la interfaz ISerializable

El otro planteamiento para serializar un objeto de manera binaria es hacer implementar la interfaz `ISerializable` mediante el objeto. Esta interfaz solo contiene un método:

```
void GetObjectData(SerializationInfo info, StreamingContext context);
```

Por lo tanto, hay que implementar este método para describir lo que se quiere serializar y cómo. El enfoque, aunque es más flexible, también es más complejo que el uso de los atributos.

Observación

A pesar de la implementación de la interfaz, sigue siendo necesario conservar el atributo `[Serializable]` encima de la clase.

El objeto `SerializationInfo` es un tipo de diccionario que permite asignar los datos de la clase que se va a serializar mediante clave/valor. Por ejemplo, si se retoma nuestra clase Persona con este enfoque, el código sería el siguiente:

```
[Serializable]
public class Persona2 : ISerializable
{
    public string Nombre { get; set; }
    public string Apellido { get; set; }
    public int Edad { get; set; }
    public DateTime FechaDeNacimiento { get; set; }

      public void GetObjectData(SerializationInfo info,
StreamingContext context)
    {
        info.AddValue("Nombre", Nombre);
        info.AddValue("Apellido", Apellido);
    }
}
```

Se constata que se han eliminado todos los atributos de gestión interna de la serialización, ya que la función del método `GetObjectData` es definir qué miembros se deben serializar.

Capítulo 2-4
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`.

Tal diseño está basado en el patrón `Factory Method`, el método `creaPedido` es el método de fabricación. El ejemplo se detalla en la figura 2-4.1.

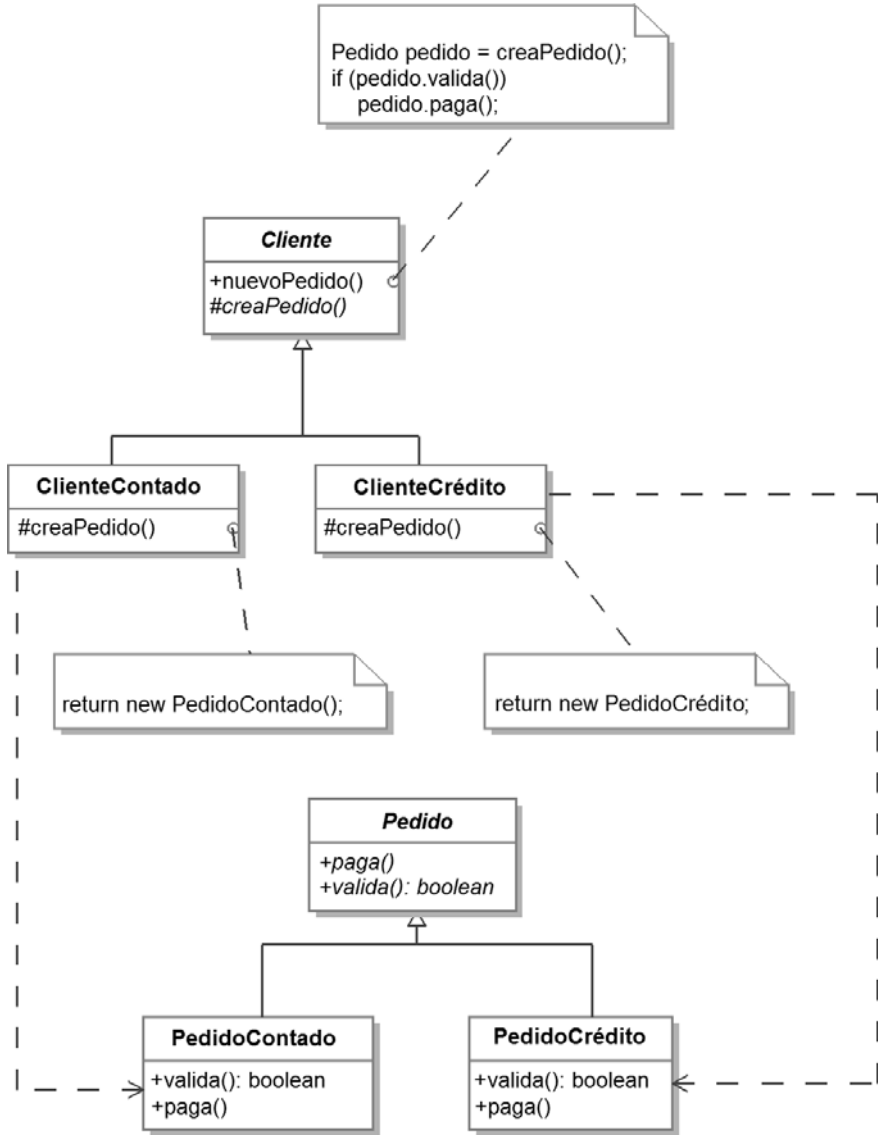


Figura 2-4.1 - El patrón `Factory Method` aplicado a los clientes y sus pedidos

3. Estructura

3.1 Diagrama de clases

La figura 2-4.2 detalla la estructura genérica del patrón.

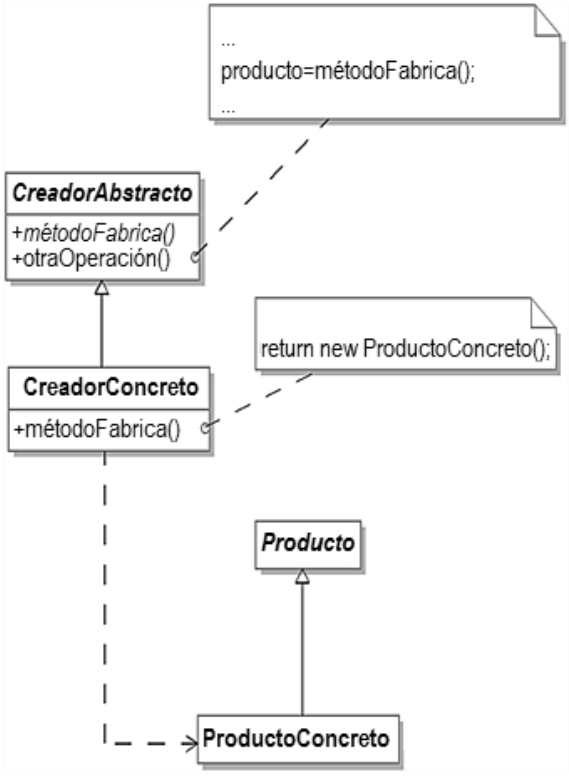


Figura 2-4.2 - Estructura del patrón `Factory Method`

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);
        }
    }
```

```
}


public class ClienteContado : Cliente
{
    protected override Pedido creaPedido(double importe)
    {
        return new PedidoContado(importe);
    }
}


public class ClienteCredito : Cliente
{
    protected override Pedido creaPedido(double importe)
    {
        return new PedidoCredito(importe);
    }
}
```

Por último, la clase `Usuario` muestra un ejemplo de uso del patrón `Factory Method`.

Observación

El nombre Usuario denota aquí un objeto usuario de un patrón.

```
using System;

public class Usuario
{
    static void Main(string[] args)
    {
        Cliente cliente;
        cliente = new ClienteContado();
        cliente.nuevoPedido(2000.0);
        cliente.nuevoPedido(10000.0);
        cliente = new ClienteCredito();
        cliente.nuevoPedido(2000.0);
        cliente.nuevoPedido(10000.0);
    }
}
```