UNIVERSIDAD
REY JUAN CARLOS

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA INFORMÁTICA

Curso Académico 2014/2015

Proyecto de Fin de Carrera

HIPERSERIALIZER
(Generador de serializadores de alto rendimiento)

Autor: Víctor Parra Santiago

Tutor: Agustín Santos Méndez

TABLA DE CONTENIDOS

Resumen del proyecto

Un **serializador** es un programa que codifica un objeto en un stream (de texto o de bytes) para almacenarlo o transmitirlo por la red con el fin de que pueda ser usado para crear un nuevo objeto idéntico al original. Existen muchos serializadores en el mercado, cada uno con sus ventajas y desventajas, válidos para determinados tipos de objetos y no para otros, que generan código serializado para ser almacenado y/o transportado en diversos formatos y contextos.

Este proyecto pretende, tras analizar una amplia variedad de serializadores, obtener el *HiperSerializador*, un generador de serializadores, que creará en cada uso un serializador versátil para cualquier tipo de objeto, generando cualquier tipo de representación, transportable y almacenable en cualquier contexto. Es capaz de reconocer los atributos estándar o cualquier atributo particular que se desee generar y es ampliable mediante plug-ins. Los serializadores generados obtienen muy buen rendimiento.

Esta aplicación es un **generador de serializadores particulares de alto rendimiento**.

La aplicación posee las siguientes características:

- **Velocidad** de proceso. Para contextos en los que haya que serializar y/o deserializar un gran número de objetos la ventaja aumenta. Y si todos los objetos a serializar son de un mismo tipo, la mejora pasa de lineal a exponencial.

- Permite serializar **cualquier tipo de datos**. Trabaja con todos los tipos de datos existentes en .Net, a diferencia de otros serializadores donde por ejemplo los arrays multidimensionales no son admitidos en la serialización, o es necesario el uso de ciertos atributos para que sea posible realizarla.

- Permite generar la serialización en **múltiples formatos**. Nuestra aplicación admite por defecto la serialización en formato XML y CSV. Está preparada para incluir otros formatos como HTML, JSON, binario, o cualquier otro que se requiera.

- Abre la puerta a funcionalidades adicional, como **convertir** cualquier tipo de objeto en cualquier otro tipo de formato. Esto abriría la posibilidad de identificar tipos de datos por ejemplo .avi y codificarlos en .mpg, o cualquier otro tipo, con lo que en la práctica esta aplicación puede servir como un **transcriptor de objetos**, un conversor de tipos de datos o cualquier otra posibilidad que con estas características se nos pudiera ocurrir.

- Permitiría estas mejoras **a partir de plug-ins**. Se podría usar este mecanismo para ampliar las funcionalidades, por ejemplo las indicdas anteriormente.

1
Introducción

1.1. Serialización y entornos distribuidos

La serialización es uno de los procesos típicos en cualquier sistema informático que contemple la posibilidad de compartir (y también almacenar) información entre distintos elementos del sistema, o incluso entre distintos sistemas entre sí.

Como dato anecdótico, el chip más rápido que existe en la actualidad, el Link-On-Chip (LOC) está incrustado en el Gran Colisionador de Hadrones (LHC) y es un serializador. [1]

Existen muchos tipos de serialización. Dentro de las distintas vertientes de la programación (web, de escritorio, distribuida, etc.), con una gran variedad de lenguajes, entornos, sistemas y contextos distintos en los que se puede abordar un problema, una de las necesidades que más a menudo surge es la de compartir la información contenida en distintos tipos de elementos. Para ello se utilizan muy a menudo aplicaciones **serializadoras**.

En entornos distribuidos, esta tarea cobra especial importancia, ya que los componentes del sistema están separados tanto física como lógicamente, y se comunican y se sincronizan a través de mensajes que son transmitidos por la red entre los componentes.

En estos casos, la sincronización entre procesos incluye como uno de sus elementos fundamentales el **paso de parámetros** entre procesos. Lo que significa que la información que en el programa está representada mediante estructuras de datos, tiene que ser transformada en secuencia de bytes para formar parte de los mensajes de sincronización entre procesos distribuidos. Esta transformación se realiza con programas serializadores (en estos contextos se denominan *marshalling*), que convierten las estructuras de datos en secuencia de bytes transmisibles por la red, y que en el otro lado de la red recogen esa secuencia de bytes y reconstruyen las estructuras para usarlas como si fueran propias.

Para cada mensaje entre dos componentes del sistema distribuido se produce una serialización y una deserialización de los parámetros que se envíen. Esto supone que el proceso de serialización es un punto clave en el rendimiento del sistema, lo cual da una muy buena aproximación de la importancia de este proceso, y del impacto que cualquier mejora en su rendimiento pudiera suponer.

Todos los serializadores tienen una serie de **ventajas** e **inconvenientes**. De ahí la profusión de distintas herramientas que realizan la serialización a distintos niveles y de distintas maneras. Tomemos como ejemplo un serializador como XMLSerlizer. Es uno de los múltiples serializadores de que dispone .Net, profusamente utilizado. Genera el código serializado en formato XML con una determinada estructura, en la que el nodo principal es el nombre de la clase y los nodos hijos son los nombres de los elementos a serializar.

Este tipo de serializador permite la estructuración de la información que serializa, pero a costa de generar un código muy profuso. Tiene las ventajas de poder ser utilizada eficientemente para compartir objetos por la web, de producir un código serializado con una

representación estandarizada, además de amigable y legible, incluso de permitir compartir objetos entre distintas plataformas (por ejemplo Java); pero tiene la desventaja de generar un conjunto de información bastante grueso, en el sentido de que es necesario generar gran cantidad de información para guardar unos pocos datos. Por ejemplo, para guardar un simple número, necesita al menos dos etiquetas que ocupan el doble de espacio que el dato, además del propio número. Y esto se multiplica en el momento en que empecemos a serializar objetos más complejos.

Hay distintos factores a tener en cuenta cuando se realiza el proceso de serialización de cualquier objeto. El principal es la **velocidad**, es importante que el proceso sea lo más rápido posible, una premisa fundamental en cualquier proceso informático. Además, en entornos distribuidos, cobra especial importancia otro factor a la hora de compartir objetos utilizando la serialización: el **tamaño** de los objetos serializados que se enviarán a través de la red. Otro factor importante es el **nivel de aplicación** del serializador. Casi todos los serializadores existentes tienen algún handicap a la hora de serializar (arrays multidimensionales, tipos genéricos, etc.).

- VELOCIDAD. Uno de los objetivos fundamentales en este proyecto es conseguir una herramienta que produzca el proceso de serialización más rápido posible. Y la manera óptima de conseguir esta rapidez máxima es que el serializador generado esté pensado únicamente para serializar ese tipo determinado de objetos. Esto es lo que tratamos de conseguir: generar el serializador perfecto para cada tipo de objeto. Si una aplicación está pensada para serializar sólo un determinado tipo de objeto, lo hará de la manera más rápida posible. Además, cuantos más objetos de ese mismo tipo serialice la mejora se multiplicará.

- TAMAÑO. Para dar respuesta al segundo factor relevante, obtener el tamaño mínimo en el conjunto de datos serializados para ser transmitidos con rapidez en entornos distribuidos, hemos considerado que el resultado de la serialización pudiera estar codificado en distintos formatos. De este modo, si lo que interesa es que los objetos serializados sean lo más livianos posibles, se puede generar un mecanismo de serialización cuyo código serializado ocupe lo menos posible, por ejemplo todos los valores serializados separados por comas, o en formato binario. Como el deserializador correspondiente será el adecuado para ese tipo de serialización, entenderá perfectamente como deserializar, y lo hará a la perfección y rápidamente, sin más ayudas que un determinado formateo del código serializado.

  En este contexto, una aplicación particular a la que tratamos de dar solución es la de compartir dinámicamente en sistemas distribuidos objetos que se van generando dinámicamente, consiguiendo esto a partir de la definición de sus clases. Para ello, se genera el serializador adecuado, se ejecuta y se consigue el código serializado a distribuir, transmitiéndolo por la red a modo de bytes. Al

otro lado de la red se recrea el serializador (o se transmite también), se capturan los datos del objeto serializado y se ejecuta dinámicamente la serialización adecuada para ese objeto.

- NIVEL DE APLICACIÓN. Casi todos los serializadores tienen algún punto débil a la hora de serializar. XMLSerializer o DataContractSerializer no admiten la serialización de arrays multidimensionales, SharpSerializer sólo admite la serialización de propiedades, no de campos, etc. Tratamos de conseguir un serializador que no tenga restricciones y que se pueda usar con cualquier clase sin importar el tipo de elementos que tenga en su interior.

2
Objetivo del proyecto

2.1. Objetivo del proyecto

Este proyecto consiste en el desarrollo de una aplicación que permita generar serializadores particulares para cualquier tipo de objeto, con el fin de conseguir a través de un método general, la creación de la herramienta particular para serializar y deserializar un determinado tipo de objeto de una manera eficaz y con características similares, e incluso mejoradas, respecto a otros serializadores ya existentes.

Hemos construido una aplicación que genere de manera dinámica un serializador particular para el tipo de objeto que reciba como entrada, identificando los miembros de ese objeto de modo que el programa serializador realice la serialización y deserialización del contenido de esos miembros.

Tratamos de conseguir el máximo rendimiento a la hora de serializar múltiples objetos de cualquier tipo. Obteniendo un serializador particular para el tipo de objeto en cuestión, y consiguiendo que ese serializador trabaje lo más rápidamente posible tanto para serializar como para deserializar, cuantos más objetos de la misma clase se serialicen, mayor será la ganancia en rendimiento comparado. Esta característica es ideal para contextos en los que haya que realizar un número muy alto de serializaciones de objetos del mismo tipo, y en los que el tiempo de proceso de la serialización o el tamaño y formato de la representación de los objetos serializados sean factores clave para el rendimiento del sistema. Por ejemplo, compartición de información en sistemas distribuidos.

También admite la flexibilidad de poder escoger el tipo de código serializado que se desea obtener, de cara a almacenarlo o transmitirlo de la forma más eficiente según el caso. Esto se consigue gracias a una mejora que admite el proyecto: la capacidad, mediante un mecanismo de plug-ins, de obtener la representación de objetos con distintos formatos, aparte del XML o CSV que genera por defecto. De esta manera, el proyecto se convierte en algo moldeable, que admitirá cualquier tipo de formato de serialización, como característica adicional a las ya comentadas.

Además permite el uso de atributos en la definición de las clases susceptibles de ser serializadas, para ampliar las posibilidades de personalización de la serialización. Usando distintos atributos, se identifican las clases o partes de ellas que se desean serializar, o las que no se desean serializar. Y usando atributos sobre los miembros de las clases, se pueden definir características particulares de la serialización del elemento en cuestión, como el nombre de etiqueta que se le dará a determinado miembro cuando la serialización sea en formato XML o si la codificación de los datos numéricos se hará con el formato big endian o little endian cuando la serialización sea binaria (programado en el plug-in correspondiente). Se pretende que la aplicación admita todos o al menos los principales atributos ya existentes en .Net, y además admita la incorporación de atributos personalizados gracias a los plug-ins.

Esto es, en resumen lo que hace esta aplicación:

- **Genera el código de un serializador particular que trabajará lo más rápido posible y altamente personalizable para un tipo de objeto dato**.

- A continuación, compila ese código para **obtener en memoria** el serializador **particular** para ese tipo de objetos.

- El resultado final es un objeto instanciado que se devuelve al programa que lo invocó. Este objeto, a través de la ejecución de sus métodos **Encode** y **Decode**, realizará el proceso de serializado y deserializado, respectivamente, de objetos de ese tipo.

- Y todo esto, con la posibilidad de ampliar su funcionamiento a través de plug-ins y atributos estándar o personalizados.

Con la esta aplicación habremos conseguido un objetivo múltiple:

- Generar un serializador para cualquier tipo de clase, que admite cualquier tipo de objeto con cualquier tipo de miembro en su interior. Gestiona correctamente la serialización y deserialización de cualquiera de los tipos básicos, además de los tipos complejos como arrays (unidimensinales, multidimensionales, arrays de arrays), listas en cualquiera de sus modalidades (List, Queue, Stack) y diccionarios (Dictionary y SortedList) o cualquier otro tipo de datos complejo (ArrayList, Hashtable, Queue, Stack), así como objetos dentro de objetos y clases heredadas.

- Que la serialización y deserialización de cualquier tipo de objeto sea lo más rápida posible. Quizás para objetos complejos o con determinadas características sea más rápido utilizar otro tipo de serializadores ya existentes, pero a nivel global este serializador es el más rápido en términos relativos. Véase las tablas comparativas de tiempos de ejecución en el capítulo 5.

- Obtener la información intermedia (serializada) en el formato que más convenga en cada caso, para adaptarlo a las necesidades particulares del programador que usa la herramienta. Así, en el caso de objetos que hay que distribuir en tiempo real, se trataría de obtener el tamaño mínimo del objeto serializado, poniendo especial énfasis en la simplicidad en los datos serializados, con la seguridad de que el deserializador sabrá exactamente como realizar el proceso inverso. En ese caso se podría obtener la serialización como un conjunto de campos separados por comas o en formato binario. En cambio para intercambiar información por internet pudiera ser más interesante que estos datos se conviertan en XML, manejable desde múltiples plataformas y con la capacidad de ser usados con distintos formatos (XHTML, RSS, Applets, etc.) o en JSON (desarrollo pendiente introducible mediante plug-in).

3
Otros serializadores Semejanzas y diferencias

3.1. Otros serializadores

Existen serializadores de muy distintos tipos, para muy distintos ámbitos. Según la información que se quiera compartir, de dónde viene, a dónde va o dónde se va a almacenar, por qué canal y qué uso se va a hacer de ella, tendremos la posibilidad de elegir entre un amplio abanico de herramientas que permiten la serialización y deserialización de nuestra información.

En el contexto en el que se desarrolla este proyecto, Microsoft .Net Framework 3.5, existen múltiples serializadores integrados en el framework, además de otros open-source que también hemos analizado:

- **BinarySerializer**
  El más rápido en su implementación. Permite serializar propiedades y campos tanto públicos como privados. El código serializado se representa como un array de bytes, fácil y rápidamente transportable por una red. Tiene la característica (desventaja o la ventaja) de que el contenido de un objeto serializado no es fácilmente identificable a simple vista.

- **SOAPSerializer**
  Permite serializar tanto miembros públicos como privados. Permite la serialización en formato XML o JSON, con una estructuración bien definida de los datos.

- **XMLSerializer**
  Sólo permite serializar miembros públicos. Es el tipo de serializador especialmente indicado para compartir información entre distintos entornos y para ser transmitida por la web. De este modo, es muy útil para enviar objetos por ejemplo a aplicaciones Java o de cualquier otro entorno distinto al original. [9] [10]

- **DataContractSerializer**
  Ese el serializador que usa Windows Communication Fundation (WCF) para sus comunicaciones. Genera la representación de una serialización en código XML. Requiere del uso de atributos propios ([DataContract] y [DataMember]). [11] [12] [13]

- **SharSerializer**
  Serializador open-source rápido que puede serializar en XML de una manera muy rápida sin tomar en consideración atributos ni otras restricciones de seguridad.

- **Protobuf-net**.
  Proyecto de Google que ha obtenido buenos resultados en cuanto a rendimiento, pero que obliga a algunos requisitos adicionales a la hora de serializar. [15]

3.2. Semejanzas y diferencias

En los cuadros que se adjuntan a continuación, se describen las características de los principales serializadores, así como la comparación con el generador de serializadores que se ha creado en este proyecto. Interesa conocer las propiedades de los distintos serializadores existentes, para identificar sus puntos fuertes y débiles y tener en cuenta esta información de cara a la definición de nuestro generador de serializadores dinámico.

Nombre	Origen	XML	Binario	Stream	JSON	Otros
XMLSerializer	.Net	X				
DataContractSerializer	.Net	X			X	
BinaryFormatter	.Net		X			
SOAPFormatter	.Net	X		X		X
SharpSerializer	OpenSource	X	X			
NetSerializer	OpenSource			X		
Protobuf-net	Google		X			
HiperSerializer	*URJC*	X	X	X	X	X

Figura 1. Comparativa de formatos de representación en los serializadores analizados en relación a este proyecto

Nombre	Propied.	Campos	Arrays	Arrays multidim y nested	Lists	Generic Lists	Herencia	Struct	Enum
XMLSerializer	X	X	X	-	X	X(*)	X	X	X
DataContract Serializer	X	X	X	-	X	X(*)	X	X	X
BinaryFormatter	X	X	X	X	X	-	X	X	X
SOAPFormatter	X	X	X	-	X	-	X	X	X
SharpSerializer	X	-	X	X	X	X	X	X	X
NetSerializer	X	X	X	-	X	X	X	X	X
Protobuf-net	X	X	X	-	X	X	X	X(?)	X
HiperSerializer	X	X	X	X	X	X	X	X	X

Figura 2: Comparativa de tipos de datos serializables en los serializadores analizados en relación a este proyecto

(*): Se pueden serializar si se indica en el constructor el tipo al que corresponden los objetos

(?): Se han reportado problemas con algunas estructuras (System.Drawing.Color, System.DateTimeOffset, …)

4
Desarrollo del proyecto

4.1. Introducción

Hemos dividido el trabajo a desarrollar siguiendo la típica estrategia de “divide y vencerás” de una manera incremental. Para ello hemos subdividido el proyecto en fases, cada una de las cuales cubre un aspecto necesario para ir avanzando en el desarrollo final del proyecto.

A continuación se resumen las fases y sus objetivos, y posteriormente se irán detallando las acciones realizadas en cada una de las fases, así como los problemas encontrados y las soluciones elegidas en cada caso.

- FASE 1. Compilación y ejecución dinámica de una clase.

  Una de las premisas fundamentales de nuestro proyecto es que se puedan generar el código del serializador al vuelo, en tiempo de ejecución. Este código tiene que poder ser compilado en tiempo de ejecución, dando lugar a un ensamblado en memoria. Dicho ensamblado podrá ser ejecutado, generando una instancia de esa clase que permita ejecutar sus métodos **Encode** y **Decode**, que es finalmente la funcionalidad esperada.
  Como resultado de esta fase obtendremos la forma de compilar y ejecutar dinámicamente cualquier clase generando su código dentro de un string (o stream).

- FASE 2. Identificar el modo de uso de la clase serializadora.

  En esta fase se trata de averiguar cuál es el modelo más eficiente para invocar la serialización y deserialización de clases, dentro de las distintas posibilidades en las que podemos plantear la definición de esta clase. Se trata de averiguar cuál es la mejor manera de invocar la serialización de un objeto concreto, contemplando las distintas alternativas de invocación utilizando el objeto a procesar (pasándolo por valor o por referencia), pasando por el uso de métodos de extensión e incluso valorando si la clase serializadora tendrá sus principales métodos estáticos o de instancia.

- FASE 3. Escritura del código de la clase serializadora.

  Se generar el código para generar un serializador que permita serializar cualquier clase, almacenándolo en una variable string o stream. A continuación se compila y se invoca la ejecución de la serialización de un objeto de una clase determinada. Se establece la base de como se comportará el programa a la hora de serializar cualquier tipo de clase generando el serializador para clases particulares (las usadas en la fase anterior para probar cuáles son las características que tendrá el serializador) para observar las particularidades que se producen al poner en práctica toda la teoría anterior. También se define como se generará el código serializado, así como su estructura.

- FASE 4. Generar el código de la clase serializador refactorizando el código de la fase anterior.

En esta fase se continúa con la anterior, en el sentido de observar las similitudes entre todos los códigos generados anteriormente, para obtener una programación general que se pueda aplica a cualquier objeto de cualquier clase. Se contemplan las particularidades de los tipos de datos Collection, generando la programación para el tratamiento de todos ellos.

- FASE 5. Otro punto de vista.

Tras observar la complicación de la aplicación en la fase 4, se replantea la forma de identificar y tratar los tipos a serializar. De este modo, se abstrae la codificación de cada elemento a su subdivisión en sus tipos básicos, y la codificación de éstos.
De este modo, se consigue ir simplificando el problema hasta conseguir llegar a los elementos primitivos, que son trivialmente serializables. Con este nuevo enfoque la aplicación crece en abstracción y generalidad.

- FASE 6. Aplicación de atributos.

Una vez conseguido el objetivo principal, se pretende mejorar la aplicación con características adicionales que poseen el resto de serializadores. Una de las más comunes consiste en el uso de atributos a la hora de serializar. De hecho, alguno de los serializadores utilizados para comparar con nuestro proyecto no funcionan si no poseen unos atributos concretos. Para habilitar el uso de atributos, se establece una función que comprueba los atributos de cada clase o miembro para saber si es necesario aplicarle la serialización y en qué condiciones se producirá ésta.

4.2. Reflection

Durante todas las fases del proyecto, desde la inicial hasta la última contemplada y seguramente cualquier otra que se plantee en un futuro para mejorar el proyecto, hemos utilizado una característica propia del lenguaje C# (y de muchos otros lenguajes de programación) llamada **Reflection**.

Reflection consiste en una serie de utilidades dentro del lenguaje que permiten examinar, modificar y trabajar con los tipos de datos de un programa EN TIEMPO DE EJECUCIÓN. Es muy útil en muy distintos contextos, desde simplemente identificar las características de una variable según su tipo, hasta la ejecución de métodos privados de una clase instanciada.

En nuestro caso, vamos a utilizar durante todo el proyecto esta característica para muy distintas tareas, que a continuación se detallan:

- Generar el código que trabajará con un determinado tipo de dato, reconociendo sus campos y propiedades así como sus peculiaridades (tipo, atributos, etc.).

- Compilar en tiempo de ejecución el código que vamos generando dinámicamente a partir de los tipos de datos de una clase, a través de la ejecución de la clase Activator. También permitiría, en caso de ser necesario, realizar la invocación de cualquier método de ese objeto instanciado en tiempo de ejecución de una forma dinámica, y con cualquier parámetro que también dinámicamente se decidiese usar.

- Capturar los atributos que estén definidos en la definición de la clase, ya sean predefinidos o personalizados, para hacer que la herramienta los tenga en cuenta y se comporte de una manera u otra según sus atributos.

- Permitir la gestión de distintos plug-ins para la generación del serializador con unas u otras características (tipo y formato del código serializado). Entre las mejoras que admite este proyecto, está la inclusión de otros tipos de serialización a través de un mecanismo de plug-in, para lo que la aplicación está preparada.

4.3. Fase 1

Objetivo: Compilación y ejecución dinámica de un objeto.

En esta fase comenzamos a definir una de las herramientas básicas que usaremos a lo largo del proyecto. Se trata de obtener la manera de compilar un código que se genera dinámicamente durante la ejecución de la aplicación. El código se irá generando en una variable durante la ejecución de la aplicación, y su contenido variará en función del tipo de datos para el cual se definirá el serializador. A continuación, se procede a compilar de manera dinámica ese código, y a invocar y mantener en memoria una instancia de la clase compilada. Esta instancia es la salida del generador. Para no tener problemas de tipado, se devolverá como una variable de tipo **dynamic**.

Como primera aproximación compilamos y ejecutamos la típica aplicación “Hola mundo” de manera dinámica. Se trata de introducir el código de la clase, incluyendo su definición comenzando por Class, como si se escribiera en un editor, pero introducido en una variable que pueda contener texto. En principio usaremos una variable de tipo **String** llamada *codigo*.

Posteriormente, esta variable pudiera ser un stream, ya que una de las posibilidades a la hora de usar esta aplicación podría consistir en que el código que se va a ir generando dinámicamente para el serializador de determinada clase, sea compartido o enviado a través de la red, justo antes de enviar los datos, o sea, las instancias serializadas de esas clases. De este modo, en el extremo opuesto se contará con el código previamente, se podrá compilar y ejecutar de la misma manera que hacemos en este extremo (o el objeto dinámico que ya contenga el serializador particular para esa clase), y cuando a continuación se reciban los datos correspondientes a la serialización de objetos de esa clase, se podrá realizar la deserialización de manera rápida, y seguros de que contamos con todas las herramientas necesarias para hacerlo.

En cualquier caso, una vez que contamos con el código que define la clase a compilar dentro de una variable, el siguiente paso es ejecutar la compilación en tiempo de ejecución de ese código, para generar la clase que describe. Para realizar esta compilación en tiempo de ejecución se utiliza el mecanismo de .Net para estas tareas, **CodeDOM** (Code Document Object Model). Este mecanismo requiere la parametrización del **grafo CodeDOM**, que vincula todos los elementos que usa este mecanismo.

El espacio de nombres CodeDOM se usa para compilar y ejecutar programas en tiempo de ejecución, creando una representación del código fuente compilado en memoria o en un archivo ejecutable (a través del parámetro **GenerateInMemory** de la clase **CompilerParameters** se indica que queremos dejar el objeto instanciado en memoria y no en un archivo). Es un mecanismo aplicable a cualquiera de los lenguajes de programación incluidos en .Net. De hecho, podría servir para ejecutar en un lenguaje código creado en otro distinto. Por ejemplo, un código fuente escrito en VisualBasic .Net podría ser invocado y ejecutado en un programa C# con este método.

Su funcionamiento es el siguiente:

- En primer lugar, hay que usar el espacio de nombres **CodeDOM.Compiler** para poder utilizar esta característica y todas sus funcionalidades. Para ello, usaremos:

```
using System.CodeDom.Compiler;
```

- Para su utilización hay que instanciar un objeto que implementa la intefaz **ICodeCompiler** así:

```
ICodeCompiler loCompiler = new CSharpCodeProvider().CreateCompiler();
```

- A continuación se incluyen las referencias necesarias para compilar el código. Entre ellas hay que añadir la referencia al ensamblado del tipo para el que se va a generar la serialización (tipo.Assembly.Location), ya que si no se hace, el programa creado no será capaz de trabajar con el tipo que va a serializar. Se usa para ello el objeto del espacio de nombres CodeDOM **CompilerParameters**.

```
CompilerParameters loParameters = new CompilerParameters();
loParameters.ReferencedAssemblies.Add("System.dll");
loParameters.ReferencedAssemblies.Add("System.Xml.dll");
loParameters.ReferencedAssemblies.Add(tipo.Assembly.Location);
```

- El código compilado lo dejaremos en memoria, ya que inmediatamente vamos a trabajar con él creando una instancia para devolverla como resultado de la ejecución del programa.

```
loParameters.GenerateInMemory = true;
```

- Ejecutamos la compilación, con lo que obtenemos un objeto del tipo CompilerResults, que nos permitirá controlar si se ha producido correctamente la compilación, o si ha habido algún error. Este mismo objeto es el que nos permitirá instanciar el objeto final.

```
CompilerResults cr = loCompiler.CompileAssemblyFromSource(loParameters, strCodigo);
```

- Por último, se crea una instancia del tipo que hemos creado, que podemos recoger como un objeto básico.

```
Assembly loAssembly = cr.CompiledAssembly;
Object loObject = loAssembly.CreateInstance("Serializer." + tipo.Name + "Codec");
```

Una cosa importante a tener en cuenta, es que es necesario incluir entre los parámetros para la invocación del método **CompileAssemblyFromSource** de la clase **CodeComProvider** una referencia al ensamblado de la clase que se va a serializar, de modo que se pueda invocar tanto para serializar como para deserializar y que en el contexto del programa generado exista dicha referencia y se pueda trabajar con ese tipo. Si no se hace esto, no se puede utilizar la instancia del objeto identificada con su tipo dentro del contexto del generador de serializadores.

Habría entonces que usar la conversión implícita a Object para trabajar con el objeto a serializar, y la única alternativa para obtener los valores de sus miembros sería a través de **Reflection**. Hacerlo así provocaría que la ejecución de la serialización fuera mucho más lenta, y la deserialización se complicaría mucho con variables auxiliares innecesarias para simular la estructura del objeto destino. La inclusión de la referencia del ensamblado se añade con **CompilerParameters.ReferencedAssemblies**.

Una vez compilado y en memoria, se genera un objeto de esa clase a través de una llamada con el método **Activator.CreateInstance**. Otra alternativa es hacerlo a través del método **Assembly.CreateInstance** sobre un objeto de tipo **CompilerResults.CompiledAssembly** generado a través de un **ICodeCompiler**. De cualquier de los dos modos se obtiene una instancia de la clase.

Nota: En las últimas versiones se advierte de que el uso de `Activator.CreateInstance` está en desuso (deprecated), y se aconseja utilizar mejor `Assembly.CreateInstance` para asegurarse la compatibilidad en el futuro.

En el ejemplo siguiente se utiliza **Reflection** para invocar el método de prueba sobre la instancia de la clase compilada. Para ello se utiliza **InvokeMember** sobre el tipo creado, utilizando el **BindingFlag** adecuado y se muestra el resultado de la ejecución del método. En la aplicación usaremos este mismo mecanismo para obtener los elementos a tratar en el serializador que genera el programa.

Éste es el código que compila y ejecuta en vivo un código almacenado en una variable:

```
using System;
namespace PruebaCodigoEnString
{
    class MainClass
    {
        // El código se introduce en un string
        public static string codigo = @"
            class ClaseEjemplo
            {
                public static void MostrarMensaje ()
                {
                    Console.WriteLine (""Hola mundo!"");
                }
            }";

        public static void Main (string[] args)
        {
            ICodeCompiler cc = new CSharpCodeProvider().CreateCompiler();
            CompilerParameters cp = new CompilerParameters();
            cp.ReferencedAssemblies.Add("System.dll");
            cp.GenerateInMemory = true;

            CompilerResults cr = cc.CompileAssemblyFromSource(cp, codigo);

            Assembly as = cr.CompiledAssembly;
            Object obj = as.CreateInstance("Nombre");
            Object res = obj.GetType().InvokeMember("MostrarMensaje",
                BindingFlags.InvokeMethod, null, loObject, null);
            Console.WriteLine(res);
        }
    }
}
```

Con este ejemplo como referencia, la forma en la que procedemos para realizar la compilación e instanciación del serializador que genera nuestro programa sigue estos pasos:

- Se instancia la herramienta que permite crear un compilador para un determinado lenguaje de programación **CSharpCodeProvider().CreateCompiler()**.

- Se definen los parámetros para la compilación con el uso de una instancia de la clase **CompilerParameters**. Aquí se definen los distintos parámetros que se usarán en la compilación, tales como la referencia a los ensamblados que se necesitarán (**ReferencedAssemblies.Add**), o si la compilación se llevará a cabo generando un fichero externo o se mantendrá en memoria (**GenerateInMemory = true**), o el nombre del archivo de salida, si es que se generará. En nuestro caso utilizaremos la generación del objeto compilado en memoria, para favorecer la rapidez a la hora de invocar un objeto del tipo creado para devolverlo como resultado de nuestra ejecución.

- Una vez generado el código para la clase a compilar (que es uno de los objetivos principales del proyecto) se realiza la invocación de **CompileAssemblyFromSource** sobre el objeto ICodeCompiler creado previamente, pasándole el texto del código generado (*codigo*) y los parámetros definidos (*cp*), y se obtiene un objeto de tipo **CompilerResults** con el ensamblado de la clase compilada.

- A continuación se puede comprobar que no se han producido errores en la compilación, consultando el miembro *errors* del **CompilerResults** obtenido anteriormente. Si no los hay, podemos capturar el ensamblado (*Assembly*) que es la propiedad **CompiledAssembly** de este objeto, y podemos instanciar un objeto de la clase compilada invocando al método **CreateInstance** del Assembly (aquí estamos usando Reflection para invocar el método *MostrarMensaje*). Este objeto es el compilador específico creado a partir del código que hemos generado al vuelo, y será el que devolvamos al programa llamador que solicita el serializador para una clase dada.

4.4. Fase 2

Objetivo: Estructura más eficiente de la clase serializadora

Antes de comenzar a codificar la programación que generará la clase serializadora para cualquier tipo de objeto, y pensando en la velocidad de proceso de las conversiones, nos planteamos cuál sería la mejor arquitectura de la clase a la hora de definir como se invocará dicha clase, como recibirá los parámetros de entrada, y como generará la salida.

A continuación se detallan las distintas aproximaciones que se han contemplado para la clase serializador que se va a generar:

- Uso A

  La clase contendrá dos métodos: **Encode** y **Decode**. Los métodos son normales (no son estáticos). El método Encode recibe un parámetro, la instancia de la clase a serializar, enviada por **valor** al método y devuelve un string o stream con la codificación del objeto. El método Decode recibe un parámetro, la cadena o stream que contiene la codificación del objeto, pasado por **valor**, y devuelve el objeto con los valores para cada miembro correctamente introducidos.

- Uso B

  La clase contendrá dos métodos **Encode** y **Decode**. Los métodos son normales (no son estáticos). El método Encode recibe un parámetro, la instancia de la clase a serializar, enviada por **referencia** al método. Se realiza la serialización directamente con la instancia a serializar, no sobre una copia de la misma. Se devuelve un string o stream con el objeto codificado. El método Decode recibe un parámetro, la cadena o stream que contiene la codificación del objeto, y una instancia de la clase a deserializar, pasado por **referencia**. En el propio método se rellenan los valores de cada miembro de la clase sobre el propio objeto, que se devuelve con los valores de cada miembro correctamente introducidos.

- Uso C

  Se genera una clase con **métodos de extensión** para la clase a serializar. Estos métodos son **Encode** y **Decode**, que se ejecutarán directamente como si fueran métodos de la clase que contiene el objeto serializado. Se pueden usar dos variantes de este método. Una en la que el parámetro que utilizarán los métodos serializar y deserializar se pasa por **valor** y otra en la que se pasa por **referencia**.

- Uso D

  La clase tendrá el atributo **static** para los métodos **Encode** y **Decode**. De este modo, siempre se utilizará la misma clase para todas las veces que haya que serializar el mismo tipo de objetos, generándose en memoria una sola vez. Cuando se ha cargado la primera vez, se mantendrá durante toda la ejecución para serializar o deserializar cada instancia de ese mismo tipo.

La lógica dice que esta estructura es la más adecuada, al menos en teoría. Es razonable pensar que un único serializador sea el que realice el trabajo para cualquier instancia de la misma clase. Por tanto, lo más adecuado debería ser utilizar una única instancia del serializador, que ejecute siempre los mismos métodos, métodos de clase, para codificar y decodificar tantas instancias como sea preciso del mismo tipo de objetos. Además en la práctica se demuestra que esta decisión, además de ser la más adecuada lógicamente, también lo es en cuanto a su rendimiento, porque es la opción que más rápidamente realiza las tareas al registrar los tiempos de ejecución en un benchmarck.

Con los dos primeros métodos, A y B simplemente comprobamos si es más útil la inclusión del objeto a serializar o deserializar en los métodos apropiados como parámetro pasado por valor o por referencia. Por tanto, en estas dos fórmulas se utilizan clases similares, con la única diferencia de que en la invocación de los métodos Encode y Decode en el primer caso se pasará el objeto con el que se trabaja por valor y en el segundo por referencia. A continuación se indican las definiciones correspondientes a cada método:

- Uso A:

```
public string Encode(ClaseBasica01 obj);
public ClaseBasica01 Decode(string s);
```

- Uso B:

```
public string Encode(ref ClaseBasica01 obj);
public void Decode(string s, ref ClaseBasica01 obj);
```

- Uso C:

Utilizaremos una característica de .Net denominada **métodos de extensión**. Consiste en crear una clase estática con una serie de métodos asociados a un determinado tipo (tipo primitivo o clase generada por el usuario), con una notación particular en sus parámetros, que por el hecho de incluirse en el proyecto permite que la clase o tipo indicado utilice de manera natural (como si fueran métodos suyos) los métodos de la clase de extensión. La única limitación es que en el interior de un método de extensión no se tiene acceso a los atributos de tipo privado de la clase. Como en principio nuestro generador de serializadores no va a contemplar la serialización de miembros privados, podríamos utilizar esta alternativa, aunque quedaría descartada si quisiéramos ampliar la funcionalidad de la aplicación para admitir la serialización de miembros privados.

Para utilizar esta característica, los métodos de extensión, tendremos que trabajar con la versión 3.5 del .Net Framework y asegurarnos de que el proyecto contiene una referencia a **System.Core.dll** y se añade la instrucción **using System.Linq**.

La forma de definir una clase con métodos de este tipo es con el atributo **static** (aunque luego sus métodos se invoquen como si no lo fueran) y el primer parámetro

de cada método de extensión será una variable del tipo al que se pretende extender precedido de la palabra reservada **this**. Así, para asociar la clase con métodos de referencia a cualquier objeto, solo hay que incluir en el programa donde se quiera usar el espacio de nombres con una cláusula **using**. Desde ese momento, y en tiempo de compilación, los métodos de extensión que hagan referencia a una clase determinada se anclarán a ésta y se podrán usar como si fueran métodos propios.

- Uso D:

Sólo se definen los métodos **Encode** y **Decode** como **estáticos** en la clase. Esto supone que se usará la misma instancia cada vez que se realice la serialización de un objeto de determinado tipo, por lo que no es necesario crear una nueva instancia del serializador para objetos del mismo tipo. Esto será especialmente útil al serializar un gran número de objetos del mismo tipo, ya que con una única instancia del serializador realizaremos todas las operaciones.

A continuación se describen las clases usadas para realizar las pruebas. Se trata de trabajar con objetos variados para identificar las diferencias y sobre todo las limitaciones a la hora de trabajar con distintas estructuras de datos:

- Clase01Basica. Clase normal, con dos propiedades nada más.

```
public class Clase01Basica
{
      public int var1 { get; set;}
      public string var2 { get; set; }
}
```

- Clase02ArrayNormal. Clase normal, con dos arrays unidimensionales, de tipo entero y string.

```
public class Clase02ArrayNormal
{
    public int[] var1 { get; set; }
    public string[] var2 { get; set; }
}
```

- Clase03Array. Clase con una propiedades de cada tipo de array: los dos arrays anteriores, un array bidimensional de enteros, uno tridimensional de enteros y un array de arrays de enteros. Además una propiedad de tipo Dictionary con claves string y valores enteros, y un array de objetos de una clase definida dentro de esta clase (DentroDelArray).

```
public class Clase03Array
{
    public int[] var1 { get; set; }
    public string[] var2 { get; set; }
    public int[,] var3 { get; set; }
    public int[,,] var4 { get; set; }
    public int[][] var5 { get; set; }
    public Dictionary<string, int> var6 { get; set; }
    public DentroDelArray[] var7;
}
```

- Clase04Struct. Clase normal y una estructura. La clase tiene una propiedades de tipo struct cuya estructura se define en la misma clase.

```
public class Clase04Struct
{
    public struct estructura
    {
        public int valor1;
        public string valor2;

        public estructura(int v1, string v2)
        {
            valor1 = v1;
            valor2 = v2;
        }
    }
    public estructura valor3;
}
```

- Clase05Clase. Clase normal, con una clase interna cuya definición está en el mismo archivo y una propiedad de ese tipo.

```
public class Clase05Clase
{
    public class ClaseInterna
    {
        public int var1 { get; set; }
        public string var2 { get; set; }
    }

    public ClaseInterna var3 { get; set; }

    public Clase05Clase()
    {
        var3 = new ClaseInterna();
    }
}
```

- Clase06ClaseDerivada. Dos definiciones de clase. Una clase base y una clase que deriva de ella. Se serializarán y deserializarán las propiedades de ambas clases, asociadas a la clase derivada.

```
public class ClaseBase
{
    public int var1 { get; set; }
    public string var2 { get; set; }
}

public class Clase06ClaseDerivada : ClaseBase
{
    public int var3 { get; set; }
}
```

- Clase07ClaseConTodo. Clase con una propiedad de cada tipo posible. Una de un tipo enumerado, una de tipo lista, un entero, un array de cada tipo, y una propiedad más de tipo estático, además de una propiedad con característica *protected* y dos *private* (para comprobar que no se puede acceder de manera simple a ella).

```
public class ClaseBase2
{
    public int basePublicInt { get; set; }
    protected string baseProtectedString { get; set; }
}

public class Clase07ClaseConTodo: ClaseBase2
{
    public enum colores
    {
        ROJO,
        AMARILLO,
        VERDE
    }
    public static colores publicStaticColores;
    public List<int> lista { get; set; }
    public int publicInt { get; set; }
    public int[] publicArrayInt { get; set; }
    public int[,] publicArray2DInt { get; set; }
    public int[ ][ ] publicArrayMatrizEscalonadaInt { get; set; }

    protected string protectedString { get; set; } // Falta esto

    private static int privateStaticInt;
    private float privateFloat { get; set; } // Falta esto
}
```

La prueba realizada para averiguar la estructura más adecuada consiste en repetir 10.000 veces la serialización y deserialización de cada una de las clases, registrando el tiempo que lleva cada serialización y deserialización por separado. Cada prueba se repite 3 veces con cada clase.

En las siguientes tablas se pueden ver los resultados obtenidos:

	Uso A	Uso B	Uso C	Uso D
Clase01	681 690 718	819 719 770	830 830 898	710 737 721
Clase02	715 709 846	670 1314 705	927 1898 952	828 822 754
Clase03	46305 55934 45344	6186 9047 6783	1753 1821 1933	45767 55888 51045
Clase04	753 897 737	730 1423 737	1284 1754 1232	730 668 704
Clase05	816 1589 818	763 1586 813	1055 1094 1084	841 792 770
Clase06	1261 1183 1176	2208 1173 1224	2226 1623 1570	1158 1274 1175
Clase07	7360 8882 7539	8488 866 7142	414 402 413	7832 10516 9567

Figura 3: Tabla de tiempos de la ejecución del método **Encode**. Tiempo en milisegundos

	Uso A	Uso B	Uso C	Uso D
Clase01	700 720 732	774 644 667	811 806 1261	632 688 783
Clase02	82 63 68	661 715 666	302 336 367	719 765 792
Clase03	52898 65150 55377	56834 66471 55256	4809 4266 3351	58367 69630 58628
Clase04	719 709 732	720 664 712	1017 1671 1002	692 1509 673
Clase05	945 868 1539	830 847 1600	204 238 427	779 833 1014
Clase06	1161 1210 1190	1183 1197 1173	236 238 221	1143 1108 1093
Clase07	2996 3889 3219	4970 3954 3046	236 200 204	1487 2808 1423

Figura 4: Tabla con los tiempos de ejecución del método **Decode**. Tiempo en milisegundos

Con estos datos queda claro que la mejor forma para invocar al serializador sería utilizar la característica de métodos estáticos (Uso C) para la codificación, y los métodos de extensión (Uso D) para la decodificación.

Para utilizar solo uno de los modos de uso, tomamos en cuenta que los métodos de extensión obligan a unos requisitos que pueden restringir su utilidad en aplicaciones antiguas (versión 3.5 del .Net). Finalmente nos decantamos por el Uso D, los **métodos estáticos** para la clase serializadora generada por nuestro programa, debido a su mayor generalidad.

4.5. Fase 3

Objetivo: Generar el código para el serializador para una determinada clase, almacenándolo en una cadena.

El objetivo principal del proyecto es la generación dinámica de una clase cuya funcionalidad sea serializar y deserializar los datos de cualquier tipos de objeto, conocidos en tiempo de ejecución. En esta fase se programa la creación de esa clase, la cual una vez compilada permita la serialización y deserialización. El código se va creando actuando mediante reflexión sobre el tipo objetivo, creando los métodos de codificación (Encode) y decodificación (Decode) para ese tipo recorriendo todos sus miembros capturados por Reflection. En un primer momento este código se guardará en una variable de tipo String. Una variante sencilla de implementar sería convertir esta variable en un array de bytes susceptible de ser enviado por red para distribuir esta clase.

En una primera aproximación, hemos dividido en distintas funciones la generación de las distintas partes que compondrán el código, recogiendo todas ellas en el orden adecuado en una variable llamada **codigoGenerado**. Dichas funciones se han definido para identificar las partes del código que en cada una de ellas se va a generar usando reflexión. Así, tenemos funciones como getAccesibilidad, getModificador, getCodigoByMembers, abrir, cerrar, mostrarValor.

Hemos elegido como primer formato para generar el código serializado el XML, principalmente para tener un control más exhaustivo y una visión más clara de como se va generando la información serializada con el fin de identificar exactamente qué datos del objeto son necesarios almacenar y de cada propiedad o campo serializado.

Uno de los principales problemas con los que nos encontramos en esta fase es como identificar una estructura para trabajar con sus campos en lugar de con propiedades y campos que es lo que se maneja en el caso de encontrarnos con una clase. Hay bastante literatura al respecto para identificar las estructuras con reflection, y parece que no es fácil. La estrategia es identificar por descarte cuándo nos encontramos ante una estructura:

- Primero descartamos si estamos ante una clase a partir de la propiedad **IsClass**.
- Descartamos si estamos ante un objeto Enum con **IsEnum**.
- Descartamos que estemos ante objetos de tipo primitivo con **IsPrimitive**.
- Un struct devolverá TRUE para la propiedad **IsValueType**, aunque éste será el mismo valor que devuelven múltiples tipos primitivos (ver referencia 3 sobre este apartado). Por eso primero descartamos que nos encontremos ante un tipo primitivo.

Esta comprobación es necesaria, ya que durante la ejecución del serializador se comprueba si los valores que tienen los campos o propiedades del objeto a serializar son nulos, para que en la descodificación se pongan a nulo también esos valores. Con las estructuras (igual que con los datos primitivos numéricos) no se puede hacer la comprobación de si es un valor nulo, con lo que hay que realizar estas comprobaciones en todas las partes donde se desee generar una programación distinta en función de si tiene valor o no el elemento. (Ver ejemplos en las referencias 4 y 5 al final de este bloque).

Para capturar los elementos susceptibles de ser serializados, distinguimos dentro de una clase entre **campos** (o variables) y **propiedades**. En principio esta distinción puede parecer que no es necesaria, ya que simplemente tenemos que capturar el tipo del campo/propiedad y su valor. El problema es que para hacer esto con Reflection, hay que utilizar dos clases distintas según el elemento sea un campo o una propiedad. Y aunque ambas clases (PropertyInfo y FieldInfo) son clases derivadas de la misma (MemberInfo), su uso es un poco distinto, y las invocaciones de los métodos necesarios para obtener el valor de cada elemento requieren distinto número de parámetros. Es por ello que hemos hecho la distinción desde un inicio, para trabajar de manera independiente con uno u otro tipo de objeto.

Los **campos**, o variables miembro, son variables definidas en la clase para almacenar valores. Se corresponden con posiciones de memoria de distinto tamaño en función del tipo de dato que almacenen.

Las **propiedades** son variables que tienen asociados métodos para acceder a ciertos campos de la clase, que suelen ser privados. Tienen típicamente métodos GET y SET. Se diferencian también de los campos en que al no ser estrictamente variables, no se pueden utilizar en una llamada a un método como parámetros de tipo **ref** o de tipo **out**.

A la hora de utilizarlos, desde el punto de vista del contexto de nuestra aplicación, pueden parecer iguales, ya que se usan de la misma manera. Pero su comportamiento puede ser distinto. Las propiedades suelen apoyarse en variables de tipo **private**, que son las que realmente almacenan un valor, y en sus métodos asociados **GET** y **SET** se puede modificar el valor de esas variables. Pero a la hora de serializar invocaremos implícitamente al método get para obtener el valor y al método set para modificarlo, sobre la propiedad pública que finalmente modificará a un campo privado.

En esta primera aproximación a la solución del problema, estamos tratando de acceder a todos los campos y a todas las propiedades de una clase, sean públicas o privadas. Mediante Reflection accedemos a toda la información que una instancia de la clase nos proporcione. De este modo, si una propiedad está asociada a un campo privado para controlar su valor, podemos modificar éste sin ningún problema accediendo al campo privado, pudiendo obviar las propiedades (si nos aseguramos de que tendrán siempre un campo privado asociado).

Uno de los principales problemas que nos encontramos es la manera de diferenciar las propiedades y campos que son susceptibles de ser serializados. En un primer momento, nos decantamos por la utilización del método GetMembers parametrizado con los siguientes flags:

- BindingFlags.Public. Nos da los campos y propiedades públicas, tanto propias como heredadas.
- BindingFlags.NonPublic. Nos da los campos y propiedades privadas y protected.
- BindingFlags.Static. Nos da los campos y propiedades estáticos, solo propios, no heredados.
- BindingFlags.Instance. Nos da los campos y propiedades de instancia, no los estáticos.
- BindingFlags.FlattenHierarchy. Nos da los campos y propiedades estáticos heredados.

Al recoger los miembros de esta manera, obtenemos todos los miembros susceptibles de serialización, pero además una serie de miembros identificados con el nombre “BackingField" que son miembros privados que no nos interesa procesar. Los eliminamos preguntando por el nombre del miembro, evitando procesar los que tengan ese nombre. Lo mejor sería identificar este tipo de miembros a partir de alguna de sus características.

Finalmente comenzamos a trabajar con el objeto Type a través de su método **GetFields**. Esto devolvía todos los campos del tipo indicado, pero no se obtenían las propiedades del tipo (campos con getter y setter).

Por otro lado utilizamos el método **GetProperties** del mismo objeto Type, lo que devolvía las propiedades del tipo (campo con getter y setter), pero no los campos.

Como se verá más adelante, esto producía una duplicidad en el código y una serie de comprobaciones para cada paso que se iban dando que atentaba contra las buenas práctica en programación, ya que obligaba a repetir bloques enteros de código, simplemente para aplicarlos a un tipo de objeto u otro. Se explicará entonces como se consiguió solventar esta situación.

Para capturar los valores de las propiedades y campos que sean Arrays, en primer lugar hay que identificarlos. Para conseguirlo, usamos PropertyInfo.PropertyType.IsArray (o FieldInfo.FieldType.IsArray) que devolverá **true** si la propiedad es un Array.

4.5.1. Formato de salida (Encode)

El sistema tratará de permitir la salida de distintas maneras, para lo cual se ideará un mecanismo de plug-ins que permita extender la funcionalidad de codificar a distintos formatos. En una fase posterior se explicará como se ha ideado este mecanismo, aunque en esta primera aproximación del código ya contempla y encapsula el método que servirán para este cometido (`getCodigoByMembers`).

El sistema en este punto tiene un formatos de salida disponible para el código serializado, el **XML**. También se plantea generar la salida en formato de datos separados por comas (**CSV**) o en formato binario, aunque eso será en posteriores fases. El formato XML que se usa en la serialización es el siguiente:

```
<serializacion>
  <accesibilidad>VALOR</accesibilidad>
  <tipoDeObjeto>VALOR<tipoDeObjeto>
  <elementos>
    <elemento>
      <tipoDeElemento>VALOR</tipoDeElemento>
      <nombre>VALOR</nombre>
      <tipo>VALOR</tipo>
      <isArray>VALOR</isArray>
      <valor>VALOR ó
        <count>VALOR</count>
        <type>VALOR</type>
        <rank>VALOR</rank>
        <rankData>
          <data>VALOR</data>
            <longitud>VALOR</longitud>
            <valorMenor>VALOR</valorMenor>
          </data>
        </rankData>
        <tipoDeList>VALOR</tipoDeList>
        <valores>
          <elemento>
            <tipoDeElemento>VALOR</tipoDeElemento>
            <nombre>VALOR</nombre>
            <tipo>VALOR</tipo>
            <esArray>VALOR</esArray>
            <valor>VALOR ó … </valor>
          </elemento>
        <valores>
      </valor>
    <elemento>
  <elementos>
<serializacion>
```

Figura 5: Ejemplo del formato generado para la representación en XML

4.5.2. Organigrama del proceso en esta fase

Para la generación de la clase serializadora-deserializadora se usa este flujo:

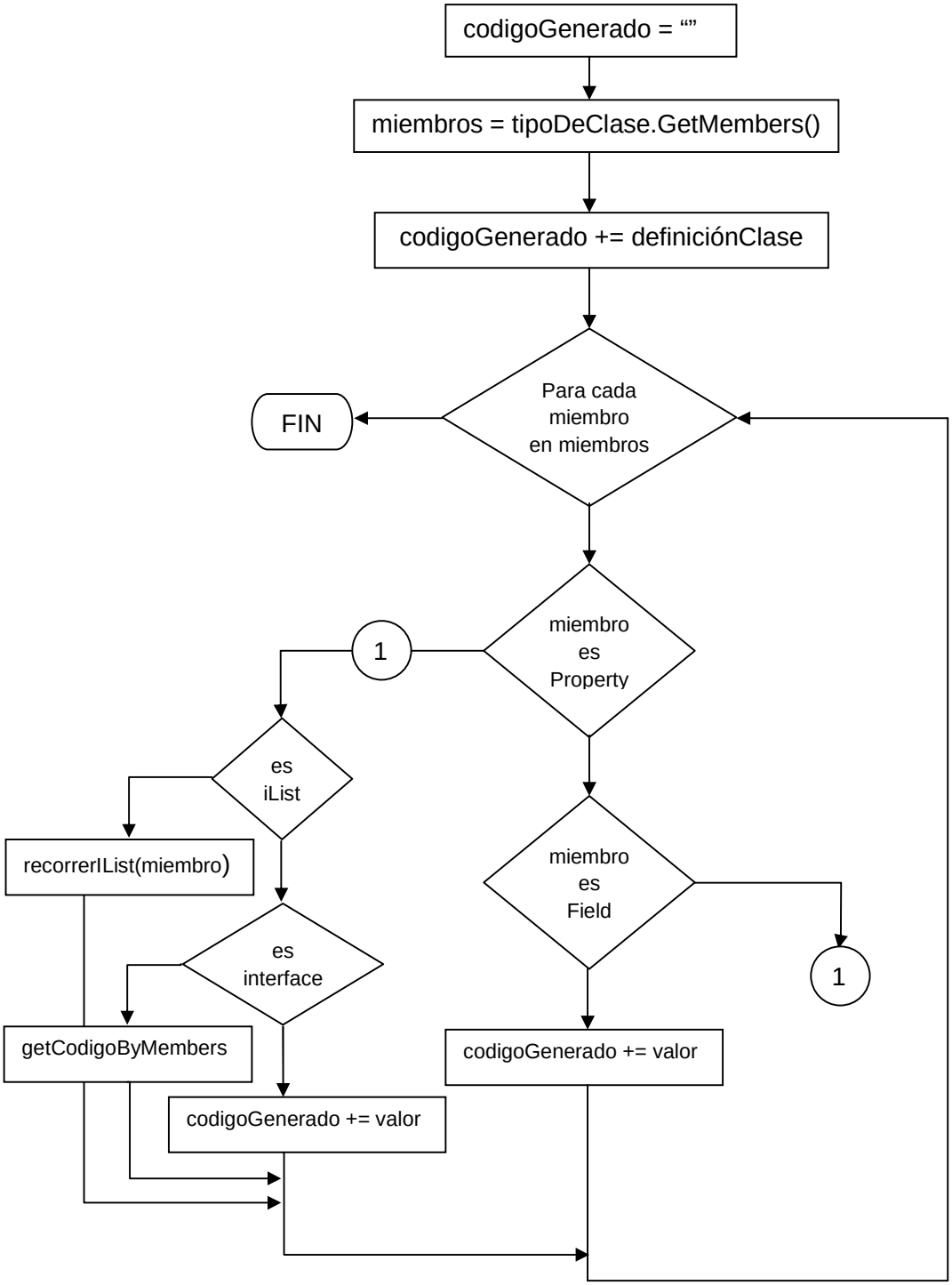


Figura 6: Pseudocódigo del generador de serilaizador en esta fase

4.5.3. Problemas encontrados y soluciones

Es difícil lidiar con estructuras complejas en tiempo de ejecución. A la hora de capturar los datos almacenados por el Encode, he tenido especial deificultad para identificar los arrays. Esto es debido a que los arrays pueden tener distinta estructura:

- arrays monodimensionales
- array multidimensionales
- arrays anidados (arrays de arrays)

La cuestión es que para codificarlos no hay ningún problema, ya que este tipo de objetos es derivado de **IEnumerable**, por lo que se pueden recorrer con un simple **foreach** y ya tenemos todos los elementos que contienen codificados, sin importar cuál sea su índice en el array. Si el array es de una dimensión, los índices son correlativos desde el 0 y no hay problema. Pero si el array tiene más de una dimensión, la cosa se complica, porque cada dimensión puede tener distinto número de elementos, y además puede haber múltiples dimensiones. Por tanto, la codificación no tiene problema, porque con el foreach se recorren todos los elementos del primero al último, sin importar el índice o la dimensión a la que correspondan dentro del array (cabe recordar que en memoria la estructura del array solo ayuda a saber dónde buscar el elemento a partir de una posición inicial).

Otra alternativa es almacenar los índices junto a su valor, pero esto es menos eficiente que un foreach. La complicación viene cuando hay que descodificar los valores, y volver a dejar el array con su misma estructura y cada dato asociado al mismo índice. Sin conocer los índices en la codificación, no se puede almacenar cada dato en su posición.

Para ayudarnos nos valemos de un par de valores que sí podemos almacenar asociados al array: el rango (array.Rank), o número de dimensiones que tiene el array, y la longitud de cada rango (array.GetLength(unRango)), o número de elementos que posee cada dimensión. Con estos datos, además de la propia longitud del array (array.Length), que es el número de elementos totales que tiene el array, podemos recomponer el array y colocar cada valor en su posición.

Simplemente, a la hora de decodificar, tenemos que definir un array auxiliar con la misma estructura que el original, apoyándonos en el rango y la longitud de cada dimensión. Posteriormente, solo tenemos que recorrer este array en todas sus posiciones, y colocar en cada una de ellas uno de los valores codificados, con la seguridad de que se posicionará en su lugar correcto. Para ello solo tenemos que recorrer el array con el mismo método que lo hicimos a la hora de codificar: un foreach.

Pero nos encontramos con otro problema: no podemos recorrer con un foreach el array para almacenar un valor en cada posición, ya que el ínidce del foreach se convierte en una propiedad inmutable (read only) cuando está en el bucle. Esto significa que podemos leer su valor, pero no podemos variarlo.

La solución a este inconveniente pasa por un pequeño truco: no podemos cambiar la variable de iteración del bucle foreach, pero sí podemos cambiar los miembros de dicha variable. Entonces lo que tenemos que hacer es convertir esa variable de iteración en un objeto con un miembro al que se pueda dar el valor que deseemos. Este objeto es una estrutura, llamada ArrayParaTodo que contiene un único elemento: un objeto llamado valor, que será donde guardaremos cada valor del array al recorrerlo en el bucle foreach.
Cuando salgamos del bucle foreach habremos sido capaces de guardar todos los valores del array en las posiciones correctas.

Resumiendo:

- A la hora de codificar un array, simplemente se recorren todos sus elementos con un foreach, y se guarda cada valor. Sin problemas. Pero también se guardan datos que se necesitarán después para deserializar el array. Estos datos son:
    - El **rango** del array (miArray.Rank), para saber cuántas dimensiones tiene
    - La **longitud de cada dimensión** (miArray.GetLength(i)), para saber cuántos elementos tiene cada una de ellas
    - El **valor mínimo de cada dimensión** (miArray.GetLowerBound(i)), para saber cuál es el primer índice de cada dimensión, ya que puede que no empiecen por 0)
    - El **valor máximo de cada dimensión** (miArray.GetUpperBound(i)), para saber cuál es el último índice de cada dimensión
- A la hora de decodificar un array, probamos con varias alterativas que me permitieran rellenar correcta y rápidamente el array. En un primer momento, contemplamos ir rellenando un array auxiliar con la misma estructura que el campo que quiero deserializar. Aquí tenemos dos posibilidades, que no he conseguido que funcionen ninguna de ellas:
    - Con este array auxiliar, utilizar un foreach para recorrer todos sus elementos y en cada elemento "simplemente" volcar cada uno de los valores que tenía serializados. Pongo "simplemente" entre comillas porque me he encontrado con que no es tan simple. Resulta que al recorrer una estructura con foreach utilizando como variable de la iteración el elemento del array, no puedo cambiar su valor dentro del foreach, y por tanto no puedo volcar el valor a cada elemento del array. No he sabido como conseguir hacerlo con este método.
    - Recorrer todos los elementos serializados del array, y cada uno de ellos meterlo en una posición del array. Esto me ha sido todavía más complejo, ya que aunque creo que tengo todos los elementos para saber a qué índice correspondería cada valor (rango, longitud de cada dimensión y posición inicial)

Finalmente, hemos conseguido aplicar un pequeño algoritmo que me permite en cada lectura de los datos codificados obtener la combinación de índices consecutiva para volcar esos datos en la posición correcta del array. Así, de una sola pasada con la longitud del array se almacenan todos los valores.

El algoritmo se basa en la utilización de los datos básicos que obtenemos del array en la codificación:

- longitud total. El número de elementos total que tiene el array sumados los elementos de todas sus dimensiones.
- el número de dimensiones del array.
- longitud de cada dimensión.
- límite inferior de cada dimensión
- límite superior de cada dimensión

A continuación se describe brevemente el código que implementa la parte de este algoritmo con la que se obtienen los índices de cada elemento del array a deserializar.

```
// Montar tantos FOR anidados como dimensiones tenga el array
string indices = "[";
string longitudes = "[";
for (int i = 0; i < t.GetArrayRank(); i++)
{
    strDecode += @"
    for(int auxIndice" + i + " = aux" + nombreAux + "GetLowerBound" + i + ";
auxIndice" + i + " <= aux" + nombreAux + "GetUpperBound" + i + "; auxIndice"
+ i + "++){";
    indices += "auxIndice" + i + ",";
    longitudes += "aux" + nombreAux + "Length" + i + ",";
}
indices = indices.Substring(0, indices.Length - 1);
indices += "]";
longitudes = longitudes.Substring(0, longitudes.Length - 1);
longitudes += "]";

strDecode += @"
    if(" + nombre + " == null) " + nombre + " = new " + tipoElemento +
longitudes + ";";
strDecode += @"
    nr.Read(); // Leer un valor de un elemnto del array";
getValueXML(t.GetElementType(), "elementoAux" + nombreAux, nombre + indices,
true);

for (int i = 0; i < t.GetArrayRank(); i++)
{
    strDecode += @"
    }";
}
```

Así, para cada rango del array se va montando un bucle **for** que recorrerá todos los elementos de cada dimensión del array, a la vez que va capturando las **longitudes** y los **índices**.

Estos valores serán utilizarlos posteriormente en la inicializacion del array y en la carga de sus valores respectivamente.

La inicialización, aunque se hace dentro de los bucles for anidados que se generan para cada dimensión y que recorre todos los elementos de una dimensión (no confundir con el bucle for que recorre las dimensiones del array), se hace dentro de una condición **if** para asegurarnos de que solo se hace la primera vez, cuando el array no ha sido aún inicializado.

Finalmente, cuando se ha generado todo el contenido de los bucles for anidados que recorren cada conjunto de índices de una dimensión, se vuelven a recorrer todas las dimensiones para cerrar tantos bucles anidados como se hubieran abierto, quedando así generado el código para obtener los datos deserializados de todo el array.

Hay una serie de variables que se utilizan como variables itinerantes de los bucles internos, y que dependen de los atributos de cada dimensión del array. No se ha explicado en esta parte del código como se generan, pero es trivial entender que de una manera similar, recorriendo todas las dimensiones del array, y para cada una de ellas obteniendo en una variable los datos anteriormente mencionados (**longitud**, **valor menor** y **valor mayor** de la dimensión).

Para que no haya solapamientos con los nombres, como se ha hecho en diversas ocasiones a lo largo del proyecto, se utiliza el propio nombre del campo que se está procesando para definir sus variables asociadas. Así, si hubiera más de un array que tratar, no habría problemas en definir variables auxiliares para cada uno de ellos, y se evitarían definiciones duplicadas de los mismos nombres de variables y problemas con contenido inválido en alguna de ellas por corresponder a datos de otro elemento.

Además de distinguirse estas variables con un sufijo que corresponde al nombre de la variable, también se añade un último elemento numérico que corresponde al índice del rango en cada caso. Esto nos asegura que para cada rango de cada array tendremos un conjunto de variables que guardarán sus datos correspondientes (longitud, índice menor e índice mayor).

Otro problema planteado surgió precisamente a la hora de inicializar un determinado campo que corresponda a un array. Conociendo las dimensiones y longitudes de cada una de ellas es trivial hacerlo a partir del tipo de dato que almacena. Esto sirve para los arrays unidimensionales y multidimensionales. A continuación se indica la línea que consigue esta inicialización sin problemas:

```
if("+nombre+" == null) "+nombre+" = new "+tipoElemento+longitudes+";";
```

Siendo **nombre** el nombre del campo a inicializar, **tipoElemento** el tipo de elemento del array y **longitudes** el conjunto de longitudes de todas las dimensiones de ese array, obtenido en el algoritmo anterior y expresado como una sucesión de números entre corchetes.

Así, por ejemplo si estamos procesando un array que se llama miArray de tipo int con dos dimensiones de 3 y 4 elementos respectivamente, este código generará la siguiente instrucción en la clase serializador:

```
miArray = new int[3,4];
```

El problema se produce cuando nos encontramos ante un jagged array, ya que sólo tendrá una dimensión, pero el tipo de elemento que contiene es en realidad arrays, con lo que la definición quedaría algo así:

```
miArray = new int[ ][3];
```

Y esta definición es errónea, debería quedar de esta otra manera:

```
miArray = new int[3][ ];
```

No ha sido fácil encontrar la forma de solucionar este problema. Una posibilidad pasaría por definir el array de otra manera, utilizando para ello la clase Array, que contiene un método llamado CreateInstance, que permite definir un array a partir del tipo de sus elementos, y de la longitu de cada una de sus dimensiones y los valores mínimos de ellas. Sería algo así:

```
nombre+" = Array.CreateInstance("+tipoElemento+", "+longitudes+");"
```

Un ejemplo:

```
miArray = Array.CreateInstance(int, new int32[2] {3, 4});
```

Nuevamente, esta definición tiene un problema con los jagged arrays, y es que genera un error de compilación cuando el tipo de elemento que se pasa como primer parámetro contiene corchetes. O sea, no se puede indicar que queremos generar un array cuyo tipo de elementos es a su vez un array.

Por tanto, a falta de encontrar una solución para este problema, nuestro programa en este punto no admitía la serialización y deserialización de arrays de arrays.

Otro problema nos lo encontrábamos cuando alguna de las clases no había sido inicializado previamente a la serialización. Había que contemplar, sobre todo de cara a la deserialización, esta eventualidad, para no tratar de asociar ningún valor a esos elementos. Es más, si en la codificación este elemento estaba vacío, para la decodificación teníamos que asegurarnos de que el elemento seguía quedando vacío, por lo que se fuerza a que se borre cualquier posible contenido que tuviera en el objeto que se usa para la deserialización. No hacerlo podría provocar que antes de la serialización, ese mismo elemento tuviera un valor en el objeto que se usa para deserializar, y el resultado fuera un objeto inconsistente con la serialización.

Serializar y deserializar enumerados ha sido complicado no tanto la serialización, sino la deserialización de los valores de variables que correspondan a tipos enumerados. Afortunadamente, .Net siempre nos ofrece la solución para realizar las conversiones que necesitemos, de una forma más o menos sencilla/compleja.

En este caso, la solución ha resultado un comando sencillo, aunque algo complejo:
https://msdn.microsoft.com/es-es/library/essfb559%28v=vs.110%29.aspx

4.6. Fase 4

Objetivo: Generar el código de la clase serializador refactorizando el código de la fase anterior.

En la fase 2 se han creado manualmente los programas que generaban la serialización y deserialización simulada de cada una de las clases con las que hemos hecho las pruebas que nos han facilitado la conclusión de definir la estructura de la clase serializadora que vamos a crear al vuelo. Estas clases, además de servirnos para averiguar qué coste temporal suponía cada una de las alternativas en la definición de la clase serializadora, nos ha dado una información muy útil: nos ha facilitado el código exacto que nuestro serializador tiene que tener para serializar y deserializar ese determinado tipo de objeto. Vamos a analizar ese código particular y a obtener un organigrama general que nos sirva para generar el programa que sirva para cualquier clase.

En primer lugar se reorganiza el funcionamiento del generador y la manera en la que genera el serializador. También se recoge el proyecto de la fase anterior, y se refactoriza para conseguir que la codificación funcione de una manera recursiva. Para ello, se utilizan un método que se invoca para cada miembro de la clase susceptible de ser serializada. Este método es:

getCodigoByMembers(System.Reflection.MemberInfo[ ], Object)

El deserializador sigue las pautas marcadas por el serializador para recuperar todos los datos serializados, poniéndolos como valores de los elementos pertinentes de un objeto que se recibe como parámetro por referencia. O sea, el método deserializador irá rellenando el objeto con cada valor adecuado para cada propiedad o campo.

Una de las principales dificultades a la hora de recuperar los datos es la recuperación de arrays. Tenemos que crear estructuras auxiliares para ir rellenándolas a medida que se van deserializando sus valores.

Para ello, primero tenemos que capturar los datos necesarios para saber el tipo de objeto ante el que nos encontramos. Estos datos son:

- **tipo** de dato del array. El propio array o lista está definido por un tipo.
- **cantidad** de elementos que contiene. Si es un array, es necesario saber cuántos elementos contendrá al inicializarlo.
- **rango** del array. No es lo mismo un array monodimensional, que una matriz, que un array de arrays. A la hora de definirlo, hay que tenerlo en cuenta.

Estas estructuras auxiliares se tienen que inicializar con el tipo exacto para ir almacenando cada elemento del array, y una vez relleno volcar ese elemento a su correspondiente en el objeto a deserializar.

Para ello utilizamos **Array.CreateInstance**, que permite crear un tipo de array con las características que nosotros deseemos. Es el único tipo de elementos en C# que admite este tipo de inicialización.

Array.CreateInstance admite tres o más parámetros: el tipo de elementos que contendrá el array, la longitud (dimensión), y el rango. Si el rango es mayor que 1, se dividen los elementos de la longitud entre el rango para saber a cuántos elementos toca cada rango (quizás esto no es así en matrices mixtas, aunque parece que estas matrices están en desuso).

Para ingresar un elemento en el array, se usa SetValue. Admite dos o más parámetros: el elemento que se quiere meter (del tipo adecuado) y el índice al que se asociará en cada uno de los rangos.

```
Array stringArray = Array.CreateInstance(typeof(String), 3);
stringArray.SetValue("Mahesh Chand", 0);
stringArray.SetValue("Raj Kumar", 1);
stringArray.SetValue("Neel Beniwal", 2);

Array intArray3D = Array.CreateInstance(typeof(Int32), 2, 3, 4);
for (int i = intArray3D.GetLowerBound(0); i <= intArray3D.GetUpperBound(0); i++)
    for (int j = intArray3D.GetLowerBound(1); j <= intArray3D.GetUpperBound(1); j++)
        for (int k = intArray3D.GetLowerBound(2); k <= intArray3D.GetUpperBound(2);
k++)
        {
            intArray3D.SetValue((i * 100) + (j * 10) + k, i, j, k);
        }
```

Para recoger todos los elementos del array, se puede usar perfectamente un foreach, ya que los arrays son objetos que implementan la intefaz **IEnumerable**.

```
foreach (int ival in intArray3D)
{
    Console.WriteLine(ival);
}
```

Los tipos de matrices que se pueden encontrar en C#, según como se posicionan sus elementos son: Single-dimensional arrays, Multidimensional arrays or rectangular arrays, Jagged arrays y Mixed arrays.

Es importante saber los tipos de matrices que podemos utilizar en nuestro proyecto. C# admite matrices de una dimensión, matrices multidimensionales (matrices rectangulares) y matrices de matrices (matrices escalonadas). El siguiente ejemplo muestra como declarar diferentes tipos de matrices:

Matrices unidimensionales:

```
int[ ] numbers;
```

Matrices multidimensionales:

```
string[ , ] names;
```

Matrices de matrices (escalonadas):

```
byte[  ][  ] scores;
```

La declaración de matrices (como se acaba de mostrar) no crea realmente las matrices. En C#, las matrices son objetos cuyas instancias deben crearse. Los siguientes ejemplos muestran cómo crear matrices:

Matrices unidimensionales:

```
int[ ] numbers = new int[5];
```

Matrices multidimensionales:

```
string[,] names = new string[5,4];
```

Matrices de matrices (escalonadas):

```
byte[ ][ ] scores = new byte[5][ ];
for (int x = 0; x < scores.Length; x++)
{
  scores[x] = new byte[4];
}
```

Puede haber matrices más grandes. Por ejemplo, una matriz rectangular de tres dimensiones:

```
int[ , , ] buttons = new int[4,5,3];
```

Incluso, se pueden combinar matrices rectangulares y escalonadas. Por ejemplo, el siguiente código declara una matriz unidimensional que contiene matrices tridimensionales de matrices bidimensionales de tipo **int**:

```
int[ ][,,][,] numbers;
```

…

A continuación se describe la forma de deserializar el código serializado en CSV. Una vez que se ha generado la codificación en formato CSV, para decodificarlo lo mejor es utilizar

una cola, de modo que se vuelquen todos los elementos y a medida que se van procesando se vayan sacando de la misma. De este modo, se procesarán secuencialmente todos los valores introducidos en la cola.

La primera tentación es la de trocear el CSV para que quede cada elemento en un índice de un array, pero es más fácil usar un List o un Queue. Ya que lo que queremos es exactamente la acción de una cola, utilizamos ésta.

…

Hasta ahora estoy procesando las propiedades por un lado y los campos por otro, en dos bloques de código que hacen lo mismo, pero con dos tipos distintos de objetos.
Pero resulta que ambos tipos de objeto (PropertyInfo y FieldInfo) derivan del mismo tipo, MemberInfo.

Quizás podría combinar esos dos bloques en uno solo. Esto no hará la aplicación más rápida, pero se ganará en legibilidad. Quizás hará la aplicación hasta más lenta, si es que los métodos que utilizo para procesar cada uno de los tipos de miembros no es igual en ambos casos, y hay que andar preguntando si estamos ante uno u otro tipo para ejecutarlos.

4.6.1 DECODIFICACIÓN

La decodificación se realiza de manera secuencial, igual que se va generando el código serializado. De hecho, se va generando el código del método **Decode** a la vez que el del método **Encode**. Por cada generación de código en el método Encode se genera el correspondiente código en Decode. De este modo, el proceso siempre será coherente, ya que trabajará con los mismos datos, primero codificándolos (generando la codificación)y luego decodificándolos (capturando los datos codificados y devolvíendolos a un objeto instanciado. Además, para los tipos de codificación donde no se identifiquen de ninguna manera los nombres, tipos de los elementos serializados y demás información superflua y sólo estén los datos, se recogerán éstos de manera directa, uno detrás de otro, sin necesitar siquiera los nombres de los campos a los que corresponden. Es el caso de JSON, CSV o la codificación binaria.

Para decodificar los elementos de un array o list, se usa la misma técnica, pero con una pequeña variación. Como no se sabe exactamente el número de elementos que nos podemos encontrar en el proceso de creación del serializador, tenemos que usar un bucle **foreach** para recorrer todos los elementos a serializar, o que vayan a ser deserializados, y añadirlos uno por uno al array o list correspondiente. Como la codificación se realizó con el mismo proceso, el resultado que se obtiene es el esperado, y cada valor se coloca en su posición adecuada (importante sobre todo en los arrays).

4.6.2. Decodificación de arrays, lists, collections

Para montar de una manera más dinámica los métodos de codificación y decodificación, contamos en el Generador con una función llamada **getValue**, que recibe como parámetros el tipo de dato a procesar, el nombre de la propiedad o campo en la clase, el nombre auxiliar que usaremos para los campos complejos y una serie de banderas que nos indicarán si dicho tipo de dato corresponde a un campo básico, o a un elemento de un array, de una lista o de un diccionario.

Esto se hace así porque el comportamiento tiene que ser distinto si de lo que se trata es de volcar directamente el valor a una variable, o de insertarlo en un array, o en una lista o un diccionario, donde necesitamos dos datos para realizar la inserción.

El método servirá lo mismo para generar el código de codificación que irá en el método Encode como el que irá en el método Decode. Como hemos dicho anteriormente, ambos métodos se van rellenando de manera paralela, y es esta función la que generará el código que servirá para rellenar la parte más importante de esos métodos.

Es una función, getValue, que se podrá llamar recursivamente, ya que a la hora de codificar, cuando procesamos un elemento, éste puede ser un dato de tipo primitivo (incluimos

en este apartado también al tipo String, aunque no lo sea formalmente), o puede ser un array, lista o diccionario del que tenemos que obtener todos sus elementos, o puede ser una instancia de una clase externa que hay que procesar de manera paralela, sacando todos sus campos, y codificándolos como si del objeto primero se tratara.

Además, para los tipos de datos complejos que correspondan a estructuras dinámicas, tales como arrays, listas, diccionarios, no se sabe a priori datos necesarios para su inicialización, como puede ser la cantidad de elementos que contiene en todas sus posibles dimensiones. Es por esto que el código de los métodos Encode y Decode tiene que capturar esta información para la instancia que se esté procesando.

A la hora de volver a rellenar los campos o propiedades que correspondan a arrays, al haber utilizado arrays auxiliares para ello, tenemos que hacer el volcado. Pero como hemos definido los arrays auxiliares con el tipo System.Array, a la hora de volcar hay que tener en cuenta que no se puede hacer a través de una conversión implícita al tipo de array que tuviera originariamente el campo. Es necesario realizar una conversión explícita de esta manera:

```
Array aux = new Array();
// aquí se rellena el array auxiliar

...

// Aquí se vuelca el array aux al campo o propiedad campoArray,
// siendo int[ ] el tipo de dato de campoArray
obj.campoArray = aux as int[ ];
```

Por tanto, el pseudocódigo de esta función recursiva que se encarga de generar el código que codificará por un lado y descodificará por otro la clase procesada, se muestra a continuación.

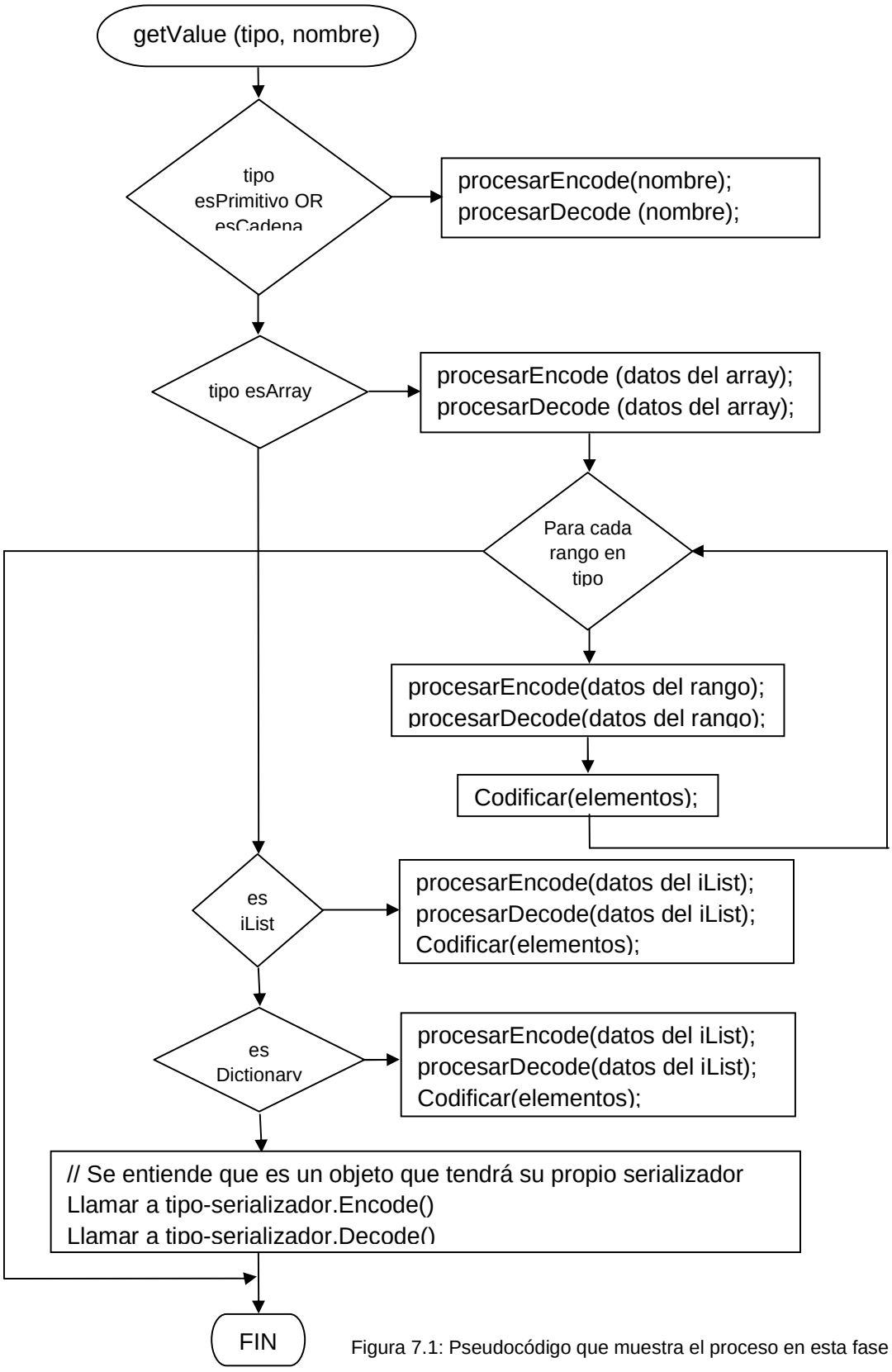


Figura 7.1: Pseudocódigo que muestra el proceso en esta fase

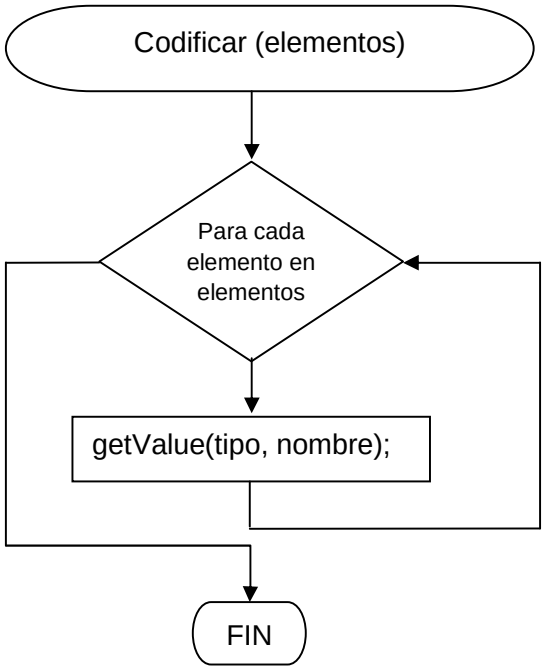


Figura 7.2: Pseudocódigo que muestra el proceso en esta fase

4.6.3. Problemas y soluciones

4.6.3.1. Problema al serializar dictionarys

Algunos de los serializadores habituales no pueden serializar Dictionarys. Por supuesto, nuestro serializador sí contempla la serialización de este tipo de datos.

> The XmlSerializer cannot process classes implementing the **IDictionary** interface. This was partly due to schedule constraints and partly due to the fact that a hashtable does not have a counterpart in the XSD type system. The only solution is to implement a custom hashtable that does not implement the **IDictionary** interface.

…

4.6.3.2. Problemas con la decodificación

Es importante destacar que a la hora de decodificar, hay que identificar qué tipo de dato es el que hay que volcar a la propiedad correspondiente, ya que al ser C# un lenguaje fuertemente tipado, no se puede simplemente volcar el contenido serializado (que será un string) a la propiedad o campo, que puede ser de cualquier tipo.

Ya que contamos en todo momento con el tipo de dato que se va a deserializar, solo tenemos que usar el método **Parse**, disponible para todos los tipos básicos de datos, para realizar la conversión. Este método convierte un string en el tipo sobre el que se invoca. Por ejemplo, si queremos guardar el valor capturado como texto desde la serialización en su correspondiente propiedad o campo y ésta es de tipo Int32, tendremos que utilizar una instrucción similar a ésta:

```
obj.propiedad = Int32.Parse(nr.Value); // Siendo nr un objeto XMLNodeReader
obj.propiedad = Int32.Parse(elementos.pop()); // Siendo elementos un StringBuilder
```

Para un posible control de errores, podríamos utilizar previamente el método **TryParse**, que devuelve TRUE ó FALSE según si se puede realizar el casting o no.

…

4.6.3.3. Tipo a partir de su nombre

Para asignar los valores correspondientes a cada propiedad o campo del objeto a deserializar, hemos de generar objetos auxiliares de los tipos internos de cada propiedad o campo. Así, por ejemplo, si tenemos un array cuyo contenido son objetos de cualquier tipo, tenemos que asignar a cada elemento del array una instancia de dicho tipo. Para trabajar con estas instancias usamos variables auxiliares que se definen con el mismo tipo.

Para solucionar el problema que se plantea cuando se trata de invocar al tipo de un objeto que no está dentro del ensamblado básico (mscorlib) hay que añadir el ensamblado como segundo parámetro de Type.GetType.

…

4.6.3.4. Las propiedades no se pueden usar como parámetros ref

A la hora de invocar el método Decode con objetos internos a un objeto, si éstos corresponden a propiedades, no se pueden pasar como parámetro por referencia a la invocación de Decode, ya que las propiedades no se pueden utilizar como parámetros out o ref y produce el correspondiente error de compilación.

```
Una propiedad, un indizador o un acceso a miembro dinámico no se pueden pasar
como parámetro out o ref
```

Para solucionar esto, instanciamos un objeto de ese mismo tipo justo antes de la llamada a Decode, y será esta instancia la que se pase por referencia en la llamada a Decode. Cuando vuelva, ya con su valor correspondiente una vez decodificado, se asigna a la propiedad adecuada sin ningún problema

…

4.6.3.5. Listas cuyo contenido son objetos instanciables

A la hora de decodificar listas, arrays y dictionarys cuyos elementos sean objetos instanciables me encontré con el problema de que aparecía un error de difícil localización de referencia nula.

La cuestión es que antes de asignar contenido a los miembros de estos objetos, hay que recordar que hay que instanciarlos. La simple definición de los mismos dentro del objeto no indica que exista la reserva de memoria para contenerlos, y al tratar de asignar un valor a sus elementos se genera este error si previamente no se ha instanciado.

…

4.6.3.6. Volcar datos a array multidimensionales

A la hora de codificar los arrays multidimensionales, para favorecer la rapidez en el volcado, recorremos el array con la función foreach, que realiza un recorrido secuencial del mismo, devolviendo en cada pasada uno de sus elementos. Lo que pasa es que así no nos quedamos con los índices del array. Pues bien, a la hora de decodificar, cabría esperar que de la misma manera se pudieran rellenar todos los elementos del array, pero esto no es posible de una manera tan sencilla.

A la hora de decodificar esos mismos arrays multidimensionales, lo lógico es realizarlo de la misma manera. Así, inicializamos el array con las dimensiones y tamaño para cada una de ellas que le correspondan, para lo cual guardamos estos datos también en la codificación. De este modo, recorriendo con la función foreach este array, se podrían ir introduciendo los valores tal como se sacaron de cada uno de sus índices en el método Encode. Pero el método con el que se realizó el recorrido secuencial para codificar todos los elementos del array no sirve para volcar datos en él. A la hora de recorrer el array vacío con un foreach para ir llenándolo con los valores codificados, descubrimos que no se pueden modificar los elementos del array, ya que foreach realiza un recorrido del array como colección enumerable que es, y en este contexto no se permite la modificación del elemento de iteración, que es justo lo que necesitamos.

Una alternativa sería montar en el método Decode (recordamos que lo estamos generando al vuelo sin saber cuáles son los datos que se van a procesar) tantos bucles for como dimensiones tuviera el array, y en el interior de todos los bucles for generar el índice que capturará cada elemento.

Otra alternativa sería tratar de usar un array auxiliar unidimensional que pudiera capturar con un único índice todos los elementos serializables, y posteriormente realizar un volcado de ese array unidimensional al array multi-dimensional. Para ello usamos un array unidimensional con la misma longitud total que la suma de todas las dimensiones del array multidimensional. Se trató de realizar esta conversión por muy diversos medios, desde **Array.Copy** hasta la conversión explícita de un tipo de array a otro con **as**. En todos los casos el problema era que no estábamos ante el mismo tipo de arrays.

Una posible solución pasaría por reconocer que los arrays son en definitiva posiciones consecutivas de memoria a partir de un punto inicial identificado con el nombre del array. Así, internamente, son lo mismo un array unidimensional de 6 elementos (object[6])que uno de dos dimensiones con esta definición: object[2,3]. No deja de ser 6 objetos guardados consecutivamente a partir del nombre del array. Lo que ocurre es que esta posibilidad pasaría por trabajar con punteros en el contexto de un proyecto C#, y .Net tiene muy restringida esta posibilidad. La única opción viable pasaría por encapsular esta acción en un bloque unsafe, y por tanto se descarta esta posibilidad.

Finalmente, la mejor opción consiste en generar dinámicamente en función del número de dimensiones (rank).

…

4.6.3.7. Manejo del formato XML

A la hora de decodificar, simplemente habría que seguir nodo por nodo los elementos que han sido codificados. Para ello, se utilizan las clases que C# facilita para tratar secuencialmente archivos XML, XMLNodeReader. Tan simple como ir leyendo de manera secuencial las etiquetas en la decodificación, tal y como se han ido generando en la codificación, conseguiremos capturar y regenerar la clase en uso.

Se genera un archivo XML bien formado con la siguiente estructura:

```
<elementos>
	<elemento>
		<nombre>VALOR</nombre>
		<tipo>VALOR</tipo>
		<valor>VALOR</valor>
	</elemento>
	<elemento>
		<nombre>VALOR</nombre>
		<tipo>VALOR</tipo>
		<valor>
			<count>VALOR</count>
			<tipoDeElementos>VALOR</tipoDeElementos>
			<rank>VALOR</rank>
			<datosDeLosRangos>
				<datosDeRango>
					<longitud>VALOR</longitud>
					<valorMenor>VALOR</valorMenor>
					<valorMayor>VALOR</valorMayor>
				</datosDeRango>
			</datosDeLosRangos>
			<valores>
				<valor>VALOR</valor>
			</valores>
		</valor>
	<elemento>
<elementos>
```

Conseguimos esto gracias a que vamos generando el código de la decodificación a la vez que generamos el de la codificación. Solo hay que realizar una lectura secuencial por cada etiqueta que se fue añadiendo en el Encode, y capturar el contenido de la etiqueta que contenga los datos que necesitamos en cada momento.

4.7. Fase 5

En las fases 3 y 4 hemos generado el código de un serializador que usando Reflection identifica todos los miembros (campos o propiedades) del objeto a serializar, realiza la serialización y la deserialización correspondiente, y lo hemos compilado y ejecutado al vuelo para obtener una instancia de la herramienta que realiza esta serialización.

Pero la herramienta creada no es la más rápida para realizar ese trabajo, ya que consume mucho tiempo utilizando Reflection para analizar, identificar, trabajar y devolver los elementos que integran cualquier clase.

La aplicación que nuestro programa genere de manera dinámica tiene que pasar de ser un serializador genérico, al serializador particular para un determinado tipo de dato. De esta manera conseguiremos que ejecute exactamente las acciones mínimas imprescindibles para serializar y deserializar los miembros concretos de ese determinado tipo de datos, de esa clase o estructura.

Así, en lugar de generar dinámicamente el código que busca, recorre y procesa los miembros de la clase en uso, lo que hacemos es recorrer esos miembros y para cada uno de ellos generar dinámicamente el código que los serializa y deserializa agrupando ese código en el serializador que será compilado al vuelo.

Así, en lugar de añadir a la programación del serializador que se genera al vuelo la captura de todos sus miembros por reflection, y recorrerlos uno por uno para obtener su tipo y valor para la serialización, lo que hacemos es recorrer todos los miembros de la clase en el momento de generar el código del serializador, y para cada uno de ellos simplemente añadimos a la programación del serializador el código que devuelve su tipo y valor (o incluso solo su valor).

Con esto lo que conseguimos es que el trabajo más costoso se realice en la generación del serializador, y no en la ejecución de la serialización en sí. O sea, conseguimos crear el serializador particular para cada clase, y que el proceso que realiza el serializador sea lo más rápido posible.

Además, como el serializador creará el código de serialización de los miembros de la clase en una determinada secuencia (la que indique el recorrido de todos los miembros de la clase en el generador de serializadores) podemos generar a la vez el código que serializa y el que deserializa, con la certeza de que el funcionamiento de ambas tareas estará sincronizado, ya que se van generando a la vez para tratar el mismo tipo de dato.

Un ejemplo; queremos serializar una clase que contiene dos campos, un entero y un string. El generador del serializador para esta clase funcionaría así:

- Primero recoge por reflection todos sus miembros, el entero y el string.
- Recorre cada uno de ellos, generando el código de los métodos Encode y Decode a la vez.
- Para el primer campo, el entero, genera la serialización, que simplemente puede ser escribir ese número, seguido de una coma.
- A continuación, genera la deserialización para el mismo campo, que simplemente consistirán en recoger el primer elemento una vez separados a partir de la coma la cadena del objeto serializado. Ese dato se pone como valor del campo entero.
- Seguimos con el segundo campo, y se genera su serialización, que simplemente consiste en escribir el texto contenido en el string.
- Por último, se captura el segundo elemento de la cadena de serialización, y se asigna como valor del campo de tipo string.

Ni siquiera sería necesario almacenar el tipo de dato de cada campo del objeto original, ya que la serialización y la deserialización están sincronizadas, en el sentido de que el programa sabrá a qué campo tiene que asociar cada valor ya que se han ido generando de una manera secuencial, y secuencialmente se pueden deserializar.

Este mismo ejemplo serviría para la serialización en formato XML, ya que sabemos exactamente cómo va a ser la estructura y qué etiquetas se van a ir generando a cada paso que el método Encode vaya dando. Por tanto, solo tenemos que recorrer las etiquetas de manera secuencial en el método Decode y estaremos capturando los valores correspondientes a cada elemento de manera secuencial. Lo mismo se podría aplicar al caso de serialización en binario, en JSON, o cualquier otro formato que deseáramos y que, como hemos ido avanzando y se explicará más adelante, a partir de un mecanismo de plug-ins se podría implementar fácilmente.

Hay una salvedad que hace que esto no sea en la práctica tan fácil como se ha explicado con este ejemplo tan trivial, y se refiere al contenido de los campos cuyos tipos sean arrays, listas o dictionaries. En esos casos, hasta el momento de la ejecución de la serialización sabemos el tipo de datos, pero para poder deserializarlos tendríamos que saber el número de elementos que contienen. Es por ello que en la serialización necesitamos almacenar ese dato, además de algunos datos adicionales que permitan reconstruir sobre el objeto a deserializar la estructura que tuviera el mismo campo antes de la serialización.

Por ejemplo, ante un campo de tipo array tendríamos que identificar cuál es la longitud del array cuando se serializó para inicializar el mismo elemento con la misma longitud en el objeto a deserializar, y posteriormente introducir en cada índice el elemento adecuado, sin dejarnos ninguno ni intentar almacenar datos de más en el array. Esta necesidad de conocer la longitud lleva a tener que almacenar este dato junto a los datos contenidos en el array.

Lo mismo ocurre con el rango del array, es necesario almacenarlo para conocer cuántas longitudes necesitamos almacenar de modo que se pueda recuperar e instanciar el array correctamente. Como dijimos en etapas anteriores, a la hora de codificar no hay problema, ya que se pueden recorrer todos los elementos de un array con un **foreach** sin conocer su longitud; pero a la hora de devolver esos elementos a sus correspondientes índices en la decodificación, tenemos que tener previamente inicializado el array con las longitudes correctas para todas sus dimensiones. Es por esto que necesitamos guardar toda esta información: el rango del array, la longitud de cada una de sus dimensiones, y los valores mínimo y máximo para cada una de ellas. Afortunadamente, toda esta información está disponible al ejecutar *Encode*, por lo que solo hay que añadirla a la información del objeto serializado, y el método *Decode* sabrá exactamente dónde está esta información (como hemos explicado anteriormente), la recogerá, instanciará el campo con las dimensiones y longitudes adecuadas, y asignará cada valor a los índices que les corresponda. Todo ello sincronizado con la codificación, ya que ambos métodos se generan a la vez.

Algo parecido ocurre con los Dictionary. Necesitamos almacenar además de su valor, su clave asociada y el tipo de ambos. Y si además se almacena también la longitud que tiene en el objeto serializado, se facilita la deserialización permitiendo que almacenar de nuevo los elementos del dictionary se realice de manera sencilla con un bucle que se repita el número de veces indicado por la longitud almacenada.

…

Otro cambio fundamental en la codificación del programa serializador es que se irán generando tantos programas serializadores como clases distintas haya dentro de la clase objetivo de la serialización. Esto significa que, salvo los tipos predefinidos en .Net, cualquier clase que haya dentro de la clase objetivo será susceptible de tener su propio serializador dentro de nuestra aplicación. Se trata de ir obteniendo todos los tipos y subtipos de cada elemento de la clase, y generar para cada uno de ellos un serializador, hasta que nos encontremos con un tipo predefinido en C# que sea trivialmente serializable. Con este nuevo enfoque la aplicación crece en abstracción, generalidad y reutilización de elementos.

Para ello, en la aplicación añadimos un campo estático denominado **clases**, que no es más que un Dictionary que almacenará una colección de objetos (**Object**) asociados a su tipo de datos (**Type**). Cada objeto contenido en el dictionary será el serializador correspondiente a una clase, y el tipo de clase al que corresponde el serializador será la clave. De este modo, a medida que el programa se va encontrando con distintos tipos de datos a serializar, generará no un serializador, sino tantos como clases que no correspondan con tipos predefinidos se

vayan encontrando. Y cuando se encuentre con que tiene que serializar un tipo de clase para el que ya había generado previamente un serializador, no lo vuelve a crear, sino que invoca el programa ya existente, con lo que a la larga el programa gana en rapidez y abstracción.

Por ejemplo, una clase Clase1 que contenga en su interior un miembro del tipo Clase2, y a su vez éste tenga un miembro de tipo Clase3 generará al ejecutar nuestra aplicación tres serializadores:

- El serializador correspondiente a la clase original (Clase1Serializer).
- El serializador correspondiente a la clase miembro (Clase2Serializer).
- El serializador correspondiente a la otra clase miembro (Clase3Serializer).

De este modo, si además dentro de la misma clase Clase1 existiera otro miembro que contuviera por ejemplo un elemento de tipo Clase2, ya no se volvería a generar el serializador para esa clase, sino que se utilizaría para serializar ese miembro el objeto instanciado que ya existe en el dictionary *clases*. Es una manera de optimizar la serialización de elementos del mismo tipo, pues ya tenemos generados todos los serializadores desde el primer momento en que se ejecuta la serialización.

A la hora de serializar en el primer nivel los miembros de Clase1, cuando nos encontremos con un elemento del tipo Clase2 lo único que tenemos que hacer es invocar a la serialización de ese tipo (Clase2Serializer.Encode) o a su deserialización (Clase2Serializer.Decode).

…

Otra de las novedades consiste en la arquitectura del propio serializador. Una vez que hemos creado la clase serializadora al vuelo, se devuelve un objeto instanciado de esa clase como resultado de la ejecución de la llamada al método **getSerializer** de **Generador**. Pero para poder usar esta instancia, no sirve con indicar que es un **Object**, ya que en tiempo de compilación no existen los métodos que se invocarán del objeto, y por tanto se produce un error de compilación:

'object' does not contain a definition for 'codificar' and no extension method 'codificar' accepting a first argument of type 'object' could be found (are you missing a using directive or an assembly reference?)

Figura 8: El error que se recoge cuando se trata de invocar los métodos del objeto creado dinámicamente

Una posibilidad para solventar este problema podría ser definir una interface que contenga los dos métodos, Encode y Decode, y hacer que el objeto que se recoja la instancia serializadora sea de ese tipo. Pero hay un problema, y es que no se puede definir una interface con métodos estáticos, y estos dos métodos para ser eficientes hemos decidido que tienen que ser estáticos. Hay fórmulas para evitar esta restricción, aunque no son muy fiables.

Otra posibilidad, que es por la que nos hemos decidido finalmente, es darle al serializador el tipo **dynamic**, lo que evitaría que se analizaran en tiempo de compilación las llamadas a los métodos estáticos **Encode** y **Decode**, lo que nos permitiría salvar el obstáculo anterior.

Lo primero que se necesita para esto es que se añadan los espacios de nombres siguientes:

```
using System.IO;
using System.Dynamic;
```

La clase que se genere, y que va contener el serializador, tiene que heredar de **DynamicObject**. O en su defecto, se indicará que el objeto que vuelva de la ejecución de getSerializer será de este tipo (indicado con el tipo **dynamic**).

De no hacerlo así, aunque en tiempo de compilación no se produce ningún error usando una variable de tipo dynamic para capturar la clase serializadora que se devuelve al ejecutar getSerializer, lo cierto es que no se puede ejecutar directamente ningún método de esta clase, ya que se indica que no corresponde a la clase Object.

La alternativa podría haber sido ejecutar el método a través de **reflection**, con GetType().InvokeMethod(), aunque no parece una alternativa muy elegante. A fin de cuentas, tratamos de evitar el uso de Reflection en este punto del desarrollo, sobre todo para obtener la máxima velocidad pero también para que la ejecución de nuestra clase resultante sea lo más limpia posible.

…

Hasta esta fase, el código de la clase serializadora se ha ido generando mediante la concatenación de cada línea en un **String**. Como una de las premisas fundamentales de este proyecto es la de conseguir la mayor rapidez posible, una forma para aumentar la velocidad de proceso a la hora de generar el serializador consiste en sustituir el tipo de datos de la variable que guarda el código de **String** a **StringBuilder**.

Según la documentación oficial, una de las situaciones en las que es más aconsejable utilizar StringBuilder en lugar de String es pecisamente cuando se van a realizar múltiples modificaciones en la cadena. En nuestro caso, concatenaciones. Con el tipo String, cada operación que se realice sobre el objeto supone en sí mismo generar un nuevo objeto, y la concatenación no es una excepción. En cambio, con StringBuilder, cada vez que se concatena una cadena a la variable, se hace uso de un buffer que va manteniendo el objeto, y que si se completa se va duplicando para asumir las necesidades de añadir nuevos elementos a la cadena. Así, es mucho más efectivo en estos casos utilizar variables de tipo StringBuilder que String, y por eso hacemos este cambio. [51]

…

Se ha dividido la estructura del programa en métodos diferenciados que encapsulan cada acción necesaria para llevar a buen término la generación, compilado y ejecución de un serializador particular para la clase dada. Esto facilita la lectura del código, además de su mantenimiento. Pero tiene una función más: la de encapsular cada paso para permitir la inserción de un mecanismo de plug-in que permita realizar de distinta manera la serialización y deserialización de las clases. Más adelante se volverá sobre el tema, aunque tiene que quedar claro que la estructuración del código ya ha previsto esta posibilidad.

A continuación se describe la composición de la aplicación, sobre todo de cara a su modularización. El archivo principal es Generador.cs, que contiene la clase que se invoca desde el cliente para utilizar la herramienta. Incluye los módulos necesarios para realizar la tarea de crear por reflexión una clase que contiene los métodos **Encode** y **Decode** para un determinado objeto. Estos módulos son:

- System.Reflection. Permite realizar todo el trabajo de averiguar los elementos internos del tipo de objeto con el que se trabajará y sus atributos, datos fundamentales para realizar el trabajo de creación del serializador para ese tipo de datos.

- System.CodeDom.Compiler. Permite realizar la compilación al vuelo del código que se va generando en la clase Generador con los datos de los elementos capturados por Reflection. También es el responsable de la instanciación del objeto compilado, y de devolver esa instancia como resultado final de la aplicación: la instancia del serializador para el tipo dado.

La clase principal, Generador, contiene varios elementos de carácter auxiliar. Además posee un método principal, que es el que se invocará para lanzar el proceso de crear el serializador, compilarlo y enviarlo al cliente. Este método se denomina **getSerializer()**. Su labor es la de generar el serializador, compilarlo y devolver el objeto generado al llamante.

El método **generateSerializer()** se encarga de conseguir mediante reflection y a partir del tipo del objeto que se desea serializar el código de la clase serializadora.

El método **compile()** se encarga de compilar el código creado en el método anterior y de generar una instancia en tiempo de ejecución. Esta instancia será el que se devuelva al cliente para que pueda trabajar con ella.

Los dos métodos GenerateCodeXML y GenerateCodeCSV, además de los métodos internos a los que se invocan (getValueXML y getValueCSV) son los que formarían parte de la clase susceptible de sacarse a un plug-in, o con la posibilidad de incorporar otros formatos de serialización, incluso otras funcionalidades distintas a la serialización y deserialización.

Los métodos **getValueXML** y **getValueCSV** corresponderían al método getValue explicado en la fase 4, con algunas diferencias inherentes a la contemplación de atributos. Por

ejemplo, a la hora de codificar se comprobaría si se ha añadido algún atributo que cambie la forma en que se codifica el elemento.

Por otro lado, el método **esSerializable** devolverá un valor booleano que indica si el miembro en cuestión, dentro de todos los miembros que posee la clase con la que se está trabajando, hay que serializarlo o no. Esto se puede modificar a partir de atributos tanto estándares como particulares, que permitan discernir los elementos que se serializarán o no (ver la fase 6).

Otra diferencia clave es la del Dictionary **clases**, que guardará en todo momento una instancia de las clases serializadoras que se van generando, de modo que si se pretende procesar un tipo no predefinido se buscará primero si ya existe una clase serializadora. Si es así, se usará el objeto instanciado de esa clase, y si no es así se añadirá al Dictionary para que se genere el serializador correspondiente para que esté disponible cuando se invoque. De este modo, para todos los tipos que contenga la clase objeto del procesamiento existirá un serializador apropiado cuando se finalice la generación de los serializadores.

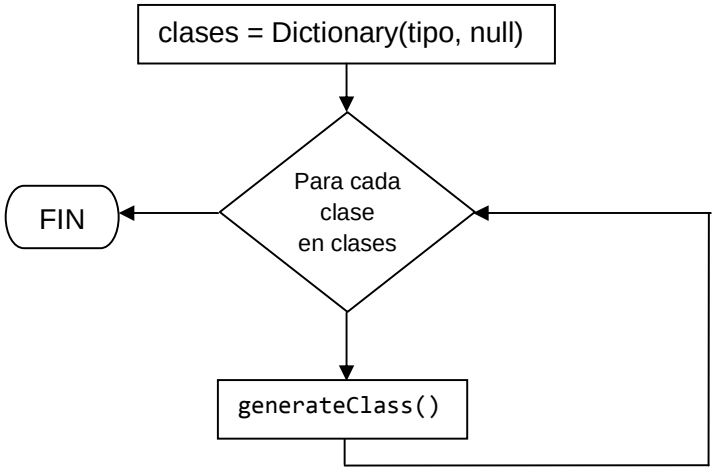
La estructura del programa que se genera es la siguiente:

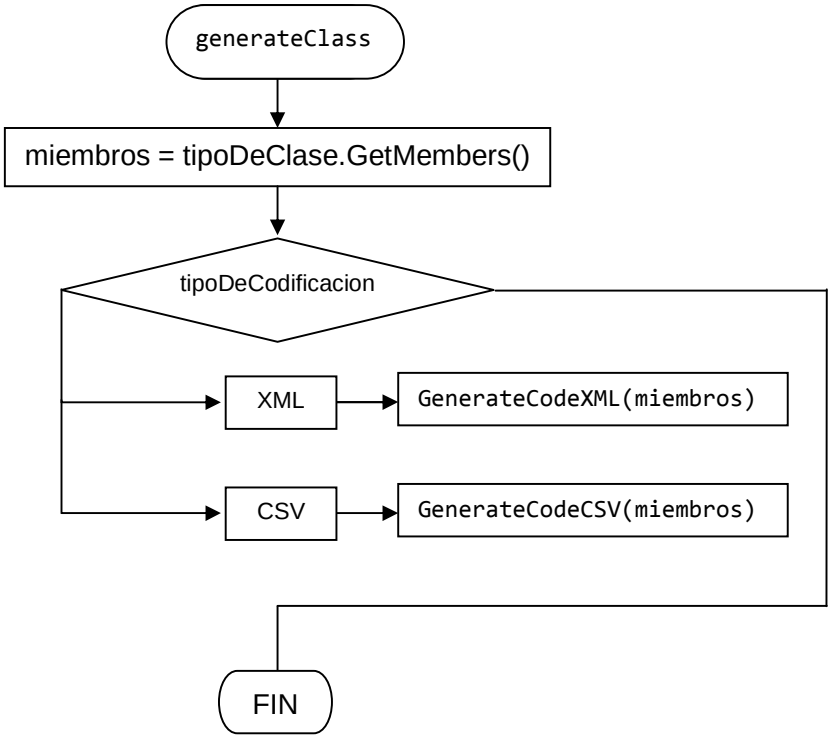
```
namespace Serializer
{
   public class nombreDeLaClaseCodec{
      public static Type tipo;
      public static String nombreClase;
      public void Encode (object obj, ref string str);
      public object Decode (string str, object obj);
      private static string abrir (string texto);
      private static string cerrar (string texto);
      private static getAccesibilidad (Type tipo);
      private static getModificador (Type tipo, ref bool pintar);
      private static getTipoDeObjeto (Type tipo);
      private static string getCodigoByMembers (MemberInfo[ ] miembros, object obj);
      private static string recorrerIList(PropertyInfo propertyInfo, object obj);
      private static string mostrarValor(object texto);
      private Dictionary<string,object> getMembersByCodigo(string str, ref object obj);
      private Boolean isStopWord(string word);
      private object getElementValue(XmlNode nodo);
   }
}
```

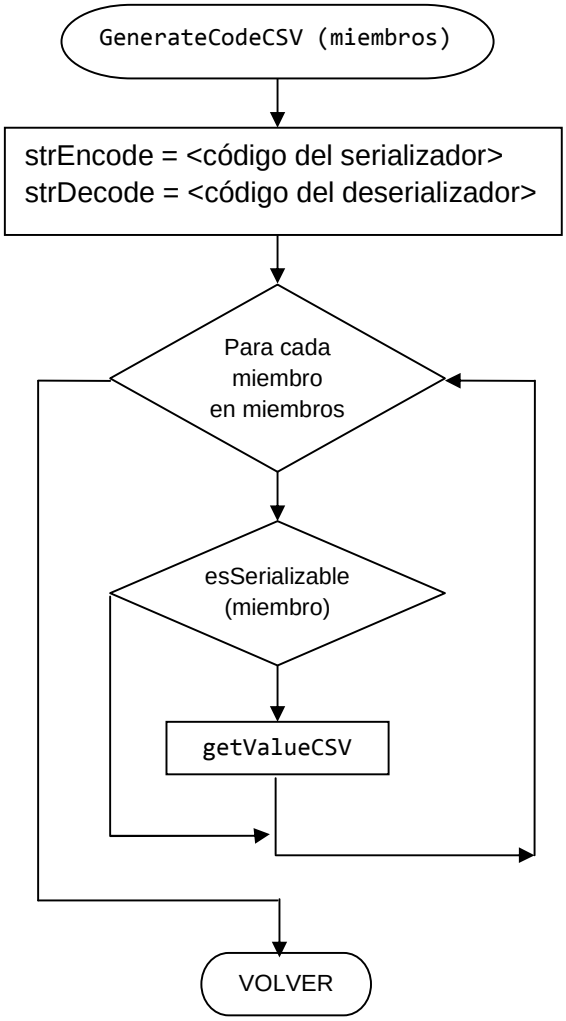
Figura 9: definición de los métodos utilizados en el generador en esta fase

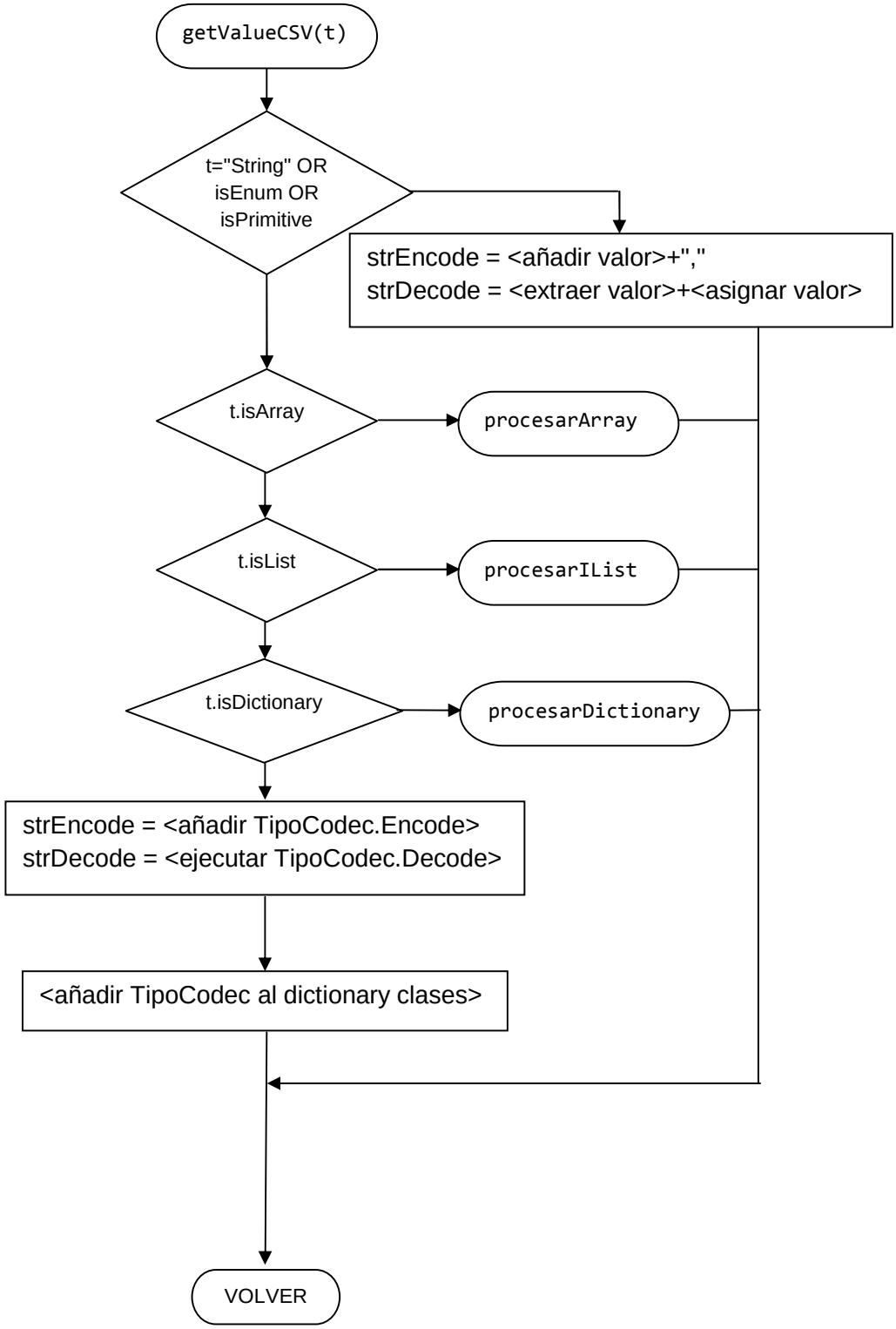
4.7.1. Organigrama en esta fase

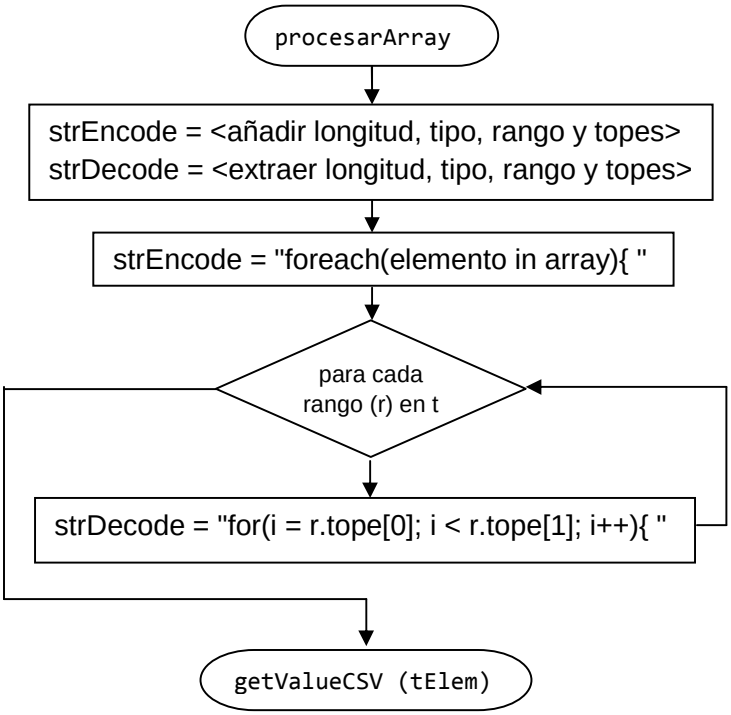
Para la generación de la clase serializadora-deserializadora se usa este flujo, resumido:

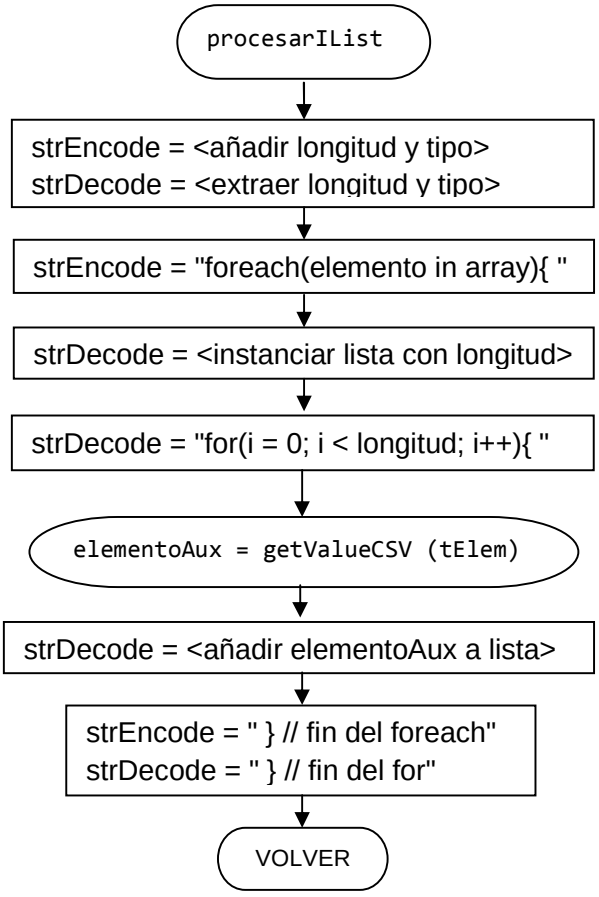


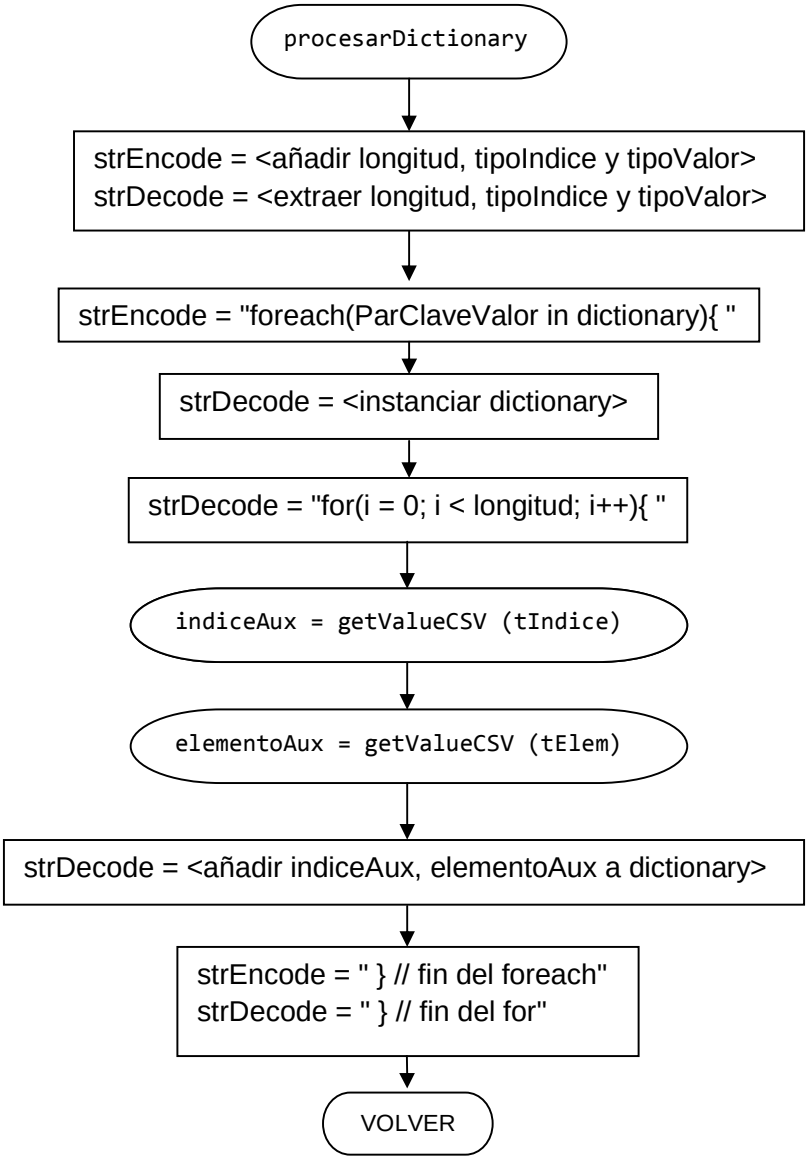



GenerateCodeCSV (miembros)
strEncode = <código del serializador>
strDecode = <código del deserializador>
Para cada miembro en miembros
esSerializable (miembro)
getValueCSV
VOLVER


getValueCSV(t)
t="String" OR isEnum OR isPrimitive
strEncode = <añadir valor>+","
strDecode = <extraer valor>+<asignar valor>
t.isArray
procesarArray
t.isList
procesarIList
t.isDictionary
procesarDictionary
strEncode = <añadir TipoCodec.Encode>
strDecode = <ejecutar TipoCodec.Decode>
<añadir TipoCodec al dictionary clases>
VOLVER


procesarArray
strEncode = <añadir longitud, tipo, rango y topes>
strDecode = <extraer longitud, tipo, rango y topes>
strEncode = "foreach(elemento in array){ "
para cada
rango (r) en t
strDecode = "for(i = r.tope[0]; i < r.tope[1]; i++){ "
getValueCSV (tElem)


procesarIList
strEncode = <añadir longitud y tipo>
strDecode = <extraer longitud y tipo>
strEncode = "foreach(elemento in array){ "
strDecode = <instanciar lista con longitud>
strDecode = "for(i = 0; i < longitud; i++){ "
elementoAux = getValueCSV (tElem)
strDecode = <añadir elementoAux a lista>
strEncode = " } // fin del foreach"
strDecode = " } // fin del for"
VOLVER


procesarDictionary
strEncode = <añadir longitud, tipoIndice y tipoValor>
strDecode = <extraer longitud, tipoIndice y tipoValor>
strEncode = "foreach(ParClaveValor in dictionary){ "
strDecode = <instanciar dictionary>
strDecode = "for(i = 0; i < longitud; i++){ "
indiceAux = getValueCSV (tIndice)
elementoAux = getValueCSV (tElem)
strDecode = <añadir indiceAux, elementoAux a dictionary>
strEncode = " } // fin del foreach"
strDecode = " } // fin del for"
VOLVER

4.8. Fase 6

Objetivo: Utilización de atributos para indicar las clases o partes de las clases susceptibles de ser serializadas.

Con la aplicación trabajando con las velocidades y funcionalidades deseadas, el siguiente paso en nuestro proyecto consiste en añadir alguna mejora sustancial sobre la funcionalidad básica. Una de las características avanzadas de los serializadores consiste en contemplar el uso de atributos para modificar el comportamiento de la serialización en aquellas clases que los integren en su definición o la de sus elementos. De hecho, algunas de las aplicaciones con que hemos comparado nuestra aplicación exigen para poder realizar la tarea de serialización que aquellos elementos que tengan que ser serializados se marquen con algún atributo.

Los atributos son una característica que poseen los lenguajes de alto nivel para modificar la definición de sus elementos. Básicamente son fragmentos de **metainformación** que se pueden aplicar a casi cualquier entidad dentro de una aplicación. Así, se pueden asociar atributos a nivel de clase, de método, de evento, de campo o propiedad. Ejemplos de atributos son los modificadores de acceso para clases, campos o propiedades (public, protected, private).

Existen una serie de atributos predefinidos, que suponen una alteración de las características del elemento al que se aplican. Así, por ejemplo, el atributo **[Not Serializable]** aplicado a un campo vendría a indicar que ese campo no sería serializado cuando se pretenda serializar el objeto que lo contiene.

Para acceder a los atributos aplicados a una definición de clase o de cualquier elemento interno de ellas se usa (como en todo este proyecto estamos usando) las funcionalidades de Reflection. Así, podemos acceder a los atributos que posee cualquier tipo de datos asociado a sus elementos y conocer su contenido para analizarlo y actuar en consecuencia.

Además, .Net incorpora la capacidad de definir atributos personalizados (**custom attributes**), que amplían las capacidades de modelado de los objetos con los que se trabaja al desarrollar una aplicación cualquiera de los lenguajes de este entorno. Se pueden aplicar atributos personalizados a todas las entidades que intervienen en un desarrollo de aplicaciones: **clase**, **método**, **evento**, **campo**, **propiedad**.

En nuestro proyecto hemos tomado en consideración algunos de los atributos predefinidos, lo que nos permite reconocer para qué elementos tenemos que aplicar la funcionalidad de nuestra aplicación y para cuáles no. Estos atributos son:

- [System.Serializable]. Indica que el elemento es serializable.

- [System.NonSerialized]. Para campos y propiedades, indica que no se deben serializar.

- [System.Obsolete("mensaje de advertencia", bool generarError)]. Evita la serialización.

Para aplicar atributos serializables tendríamos que identificar en qué puntos de la aplicación hay que tomar en consideración a éstos. Según las características del propio atributo y las consecuencias que tuviera para el proceso de serialización, tendríamos que definir distintos lugares en los que habría que comprobar la existencia o el valor de cada atributo. Por ejemplo, no es lo mismo procesar un atributo que identifique si una clase o alguno de sus elementos tuviera que ser o no serializado, que otro atributo que a la hora de serializar en formato XML un elemento personalice la etiqueta que lo va a envolver.

Uno de los principales puntos de la aplicación en la que se analizarán los atributos, porque en ese punto se van a analizar más de un atributo, es el punto en el que se decide si un miembro es serializable o no. En nuestra aplicación, ese lugar es el método **esSerializable**. Entre los criterios que usaremos para saber si un elemento (campo o propiedad) es susceptible de ser serializado, está el estudio de los atributos de ese elemento.

Por supuesto este lugar no es el único, aunque es el único que se ha implementado. Cabe destacar que esta es otra de las funcionalidades susceptibles de ser extendidas a través del mecanismo de plug-ins que hemos mencionado durante el proyecto.

Como se ha mencionado anteriormente, existe también la posibilidad de crear atributos propios, atributos personalizados. Éstos son extensiones de los atributos predefinidos que el programador puede crear a voluntad para utilizar a lo largo de su código. La manera de crear atributos personalizados es generando una clase que derive directa o indirectamente de System.Attribute. Se define esta clase con el atributo [System.AttributeUsage] donde se definen para qué tipo de entidades se aplicará, y otra información adicional como el número de etiquetas que se pueden aplicar para la misma entidad.

Ejemplo:

```
[System.AttributeUsage(System.AttributeTargets.Class |
      System.AttributeTargets.Struct, AllowMultiple = true)]
public class MiAtributo : System.Attribute
{
      private string parametroPosicional;
      public int parametroConNombre;
      public int otroParametroConNombre {get; set; }
      public MiAtributo(parametroPosicional)
      {
            parametroConNombre = 0;
            otroParametroConNombre = 1;
      }
}
```

En este ejemplo, se ha definido el atributo MiAtributo, que se puede utilizar para entidades Clase y Estructura, y además permite que se repita múltiples veces en cada definición (AllowMultiple=true). Para su invocación es necesario que se indique en primera posición el parámetro parametroPosicional, y se pueden indicar además los parámetros con nombre, en este caso sin ninguna restricción de posición, pero hay que indicar su nombre para usarlos.

La manera de usar este atributo sería así:

```
[MiAtributo(valorParametroPosicional, parametroConNombre=3, parametroConNombre=4)]
[MiAtributo(otroValorParametroPosicional, parametroConNombre=5, parametroConNombre=6)]
public class …
```

5
Resultados obtenidos Conclusiones

A continuación se detallan los resultados comparativos de la ejecución de nuestro serializador dinámico generado al vuelo para una determinada clase, con respecto a los principales serializadores de .Net y algunos de los serializadores más utilizados en el mercado.

Se han realizado las pruebas con el conjunto de clases que hemos trabajado durante todo el proyecto para ir perfilando las funcionalidades que finalmente han cristalizado en esta aplicación. Son un conjunto representativo de clases con todo tipo de elementos en su interior, lo que permite obtener un abanico de resultados significativo sobre todo de cara a la comparación con el resto de serializadores.

En las gráficas que se muestran a continuación se puede apreciar la diferencia en el tiempo de ejecución de la serialización y deserialización para cada clase con los principales serializadores de .Net (XMLSerializer, BinaryFormatter, SOAPFormatter, DataContractSerializer), otros serializadores gratuitos (SharpSerializer, en XML y binario y Protobuf .Net) y nuestro serializador (**HR HiperSerializer**, en XML y CSV).

Los tiempos mostrados son la media en milisegundos tras haber realizado 10 veces la ejecución de cada operación con todos los serializadores.

La operación consiste en repetir 10.000 veces la serialización de una clase de cada tipo con datos de prueba.

5.1. Resultados

COMPARATIVA CON LA CLASE CLASE01BASICA. CODIFICACIÓN

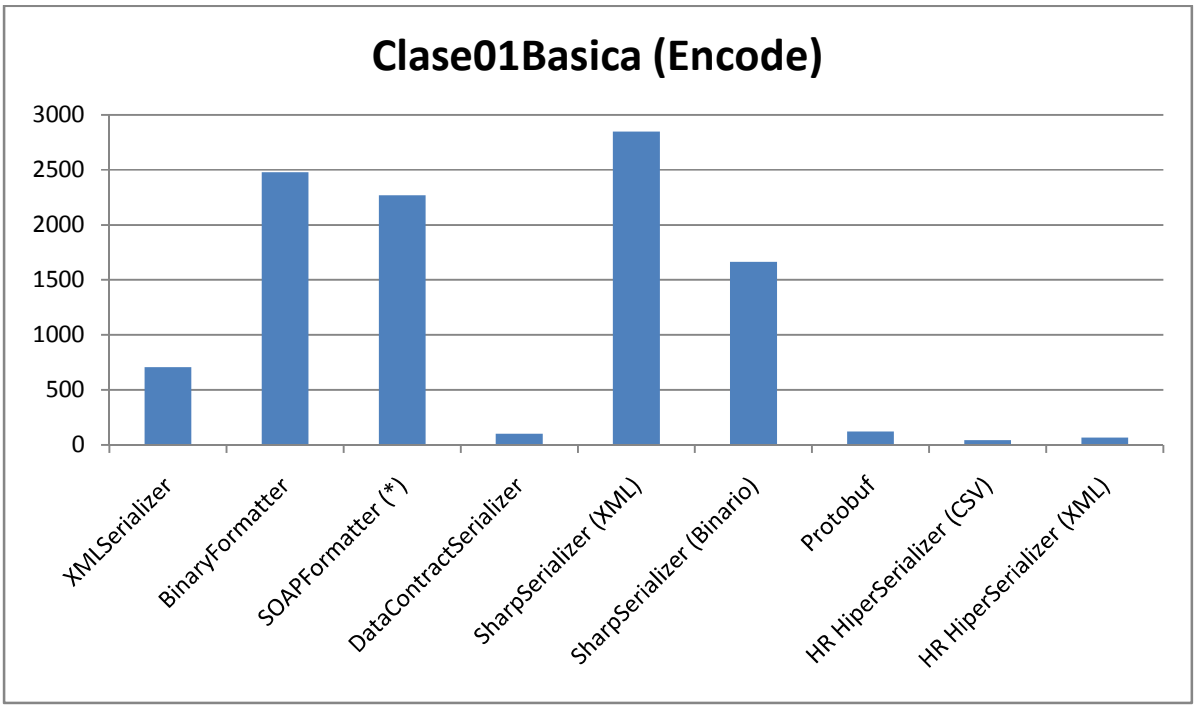


Figura 10: Comparativa de tiempos serializando la clase Clase01Basica

(*) Los tiempos de este serializador son casi 9 veces más de los indicados. Se han alterado para que la gráfica no se distorsionara mucho debido a la amplitud del rango. El resto de los tiempos son los correctos, y permiten identificar de un vistazo la diferencia entre cada serializador.

Los tiempos medios recogidos para cada serializador al serializar Clase01Basica son éstos:

Serializador	Tiempo en ms.
XMLSerializer	708
BinaryFormatter	2.478
SOAPFormatter	2.269
DataContractSerializer	103
SharpSerializer (XML)*	1.709
SharpSerializer (Binario)*	9.992
Protobuf	123
HR HiperSerializer (CSV)	*43*
HR HiperSerializer (XML)	*69*

COMPARATIVA CON LA CLASE CLASE01BASICA. DECODIFICACIÓN

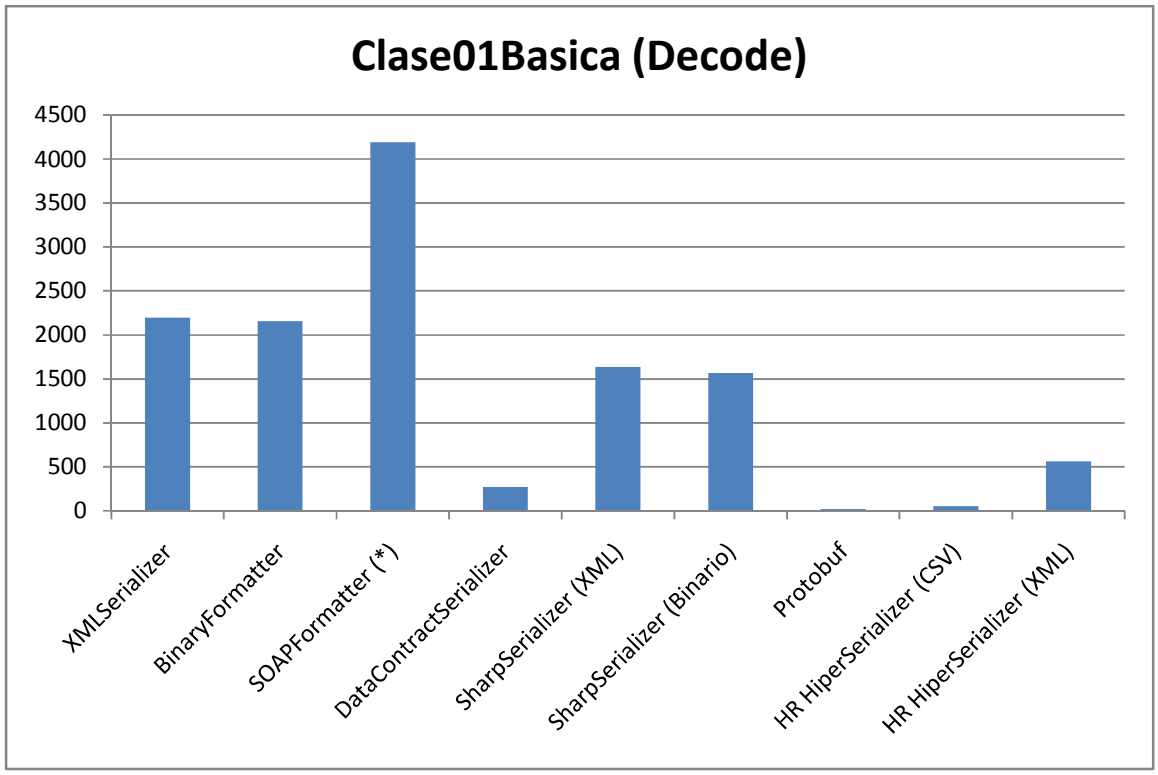


Figura 11: Comparativa de tiempos deserializando la clase Clase01Basica

(*) Los tiempos de este serializador son el doble de los indicados. Se han alterado para que la gráfica no se distorsionara mucho debido a la amplitud del rango. El resto de los tiempos son los correctos, y permiten identificar de un vistazo la diferencia entre cada serializador.

Los tiempos medios recogidos para cada serializador al deserializar Clase01Basica son éstos:

Serializador	Tiempo en ms.
XMLSerializer	2.198
BinaryFormatter	2.158
SOAPFormatter	4.190
DataContractSerializer	270
SharpSerializer (XML)*	1.638
SharpSerializer (Binario)*	1.565
Protobuf	22
HR HiperSerializer (CSV)	*53*
HR HiperSerializer (XML)	*562*

COMPARATIVA CON LA CLASE CLASE02ARRAYNORMAL. SERIALIZACIÓN

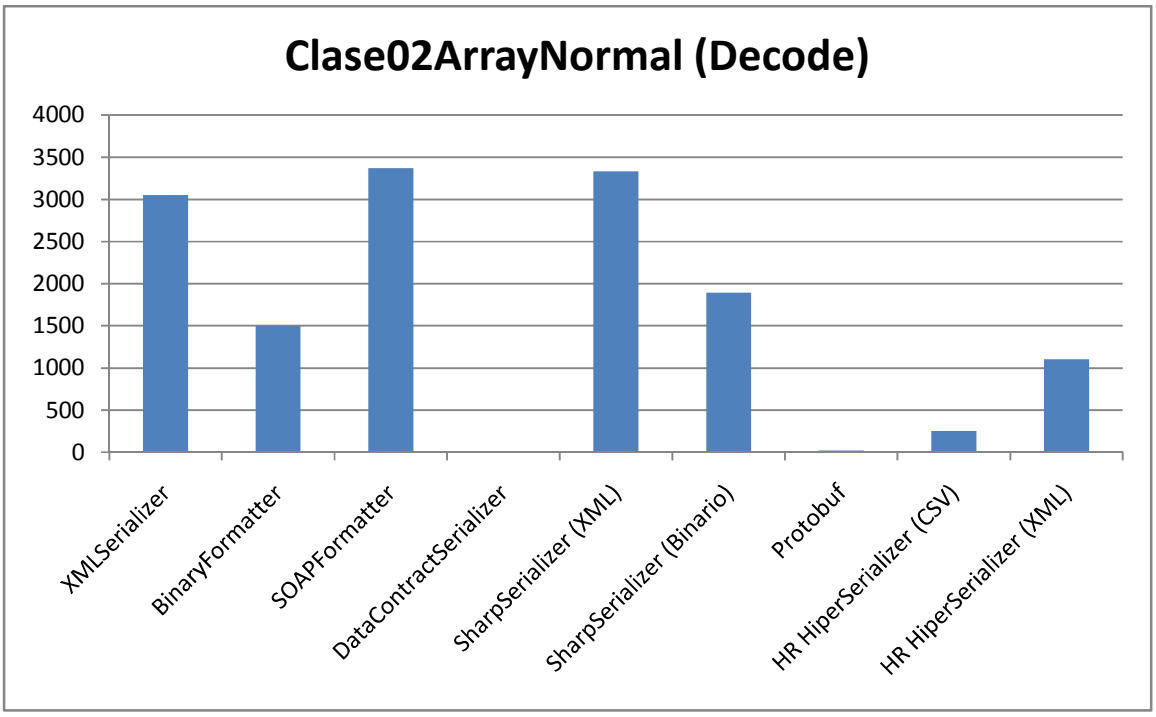


Figura 12: Comparativa de tiempos serializando la clase Clase02ArrayNormal

Para este caso el serializador DataContractSerializer no se ha podido evaluar, ya que al tratar de serializar un array devuelve el siguiente error:

> 'Fase02.Clase02ArrayNormal' with data contract name 'Clase02ArrayNormal:http://schemas.datacontract.org/2004/07/Fase02' is not expected. Consider using a DataContractResolver or add any types not known statically to the list of known types - for example, by using the KnownTypeAttribute attribute or by adding them to the list of known types passed to DataContractSerializer.

Los tiempos medios recogidos al serializar Clase02ArrayNormal son éstos:

Serializador	Tiempo en ms.
XMLSerializer	508
BinaryFormatter	1.235
SOAPFormatter (*)	1.760
DataContractSerializer	N/A
SharpSerializer (XML)*	1.525,3
SharpSerializer (Binario)*	1.185,9
Protobuf	78
HR HiperSerializer (CSV)	*80*
HR HiperSerializer (XML)	*114*

COMPARATIVA CON LA CLASE CLASE02ARRAYNORMAL. DESERIALIZACIÓN

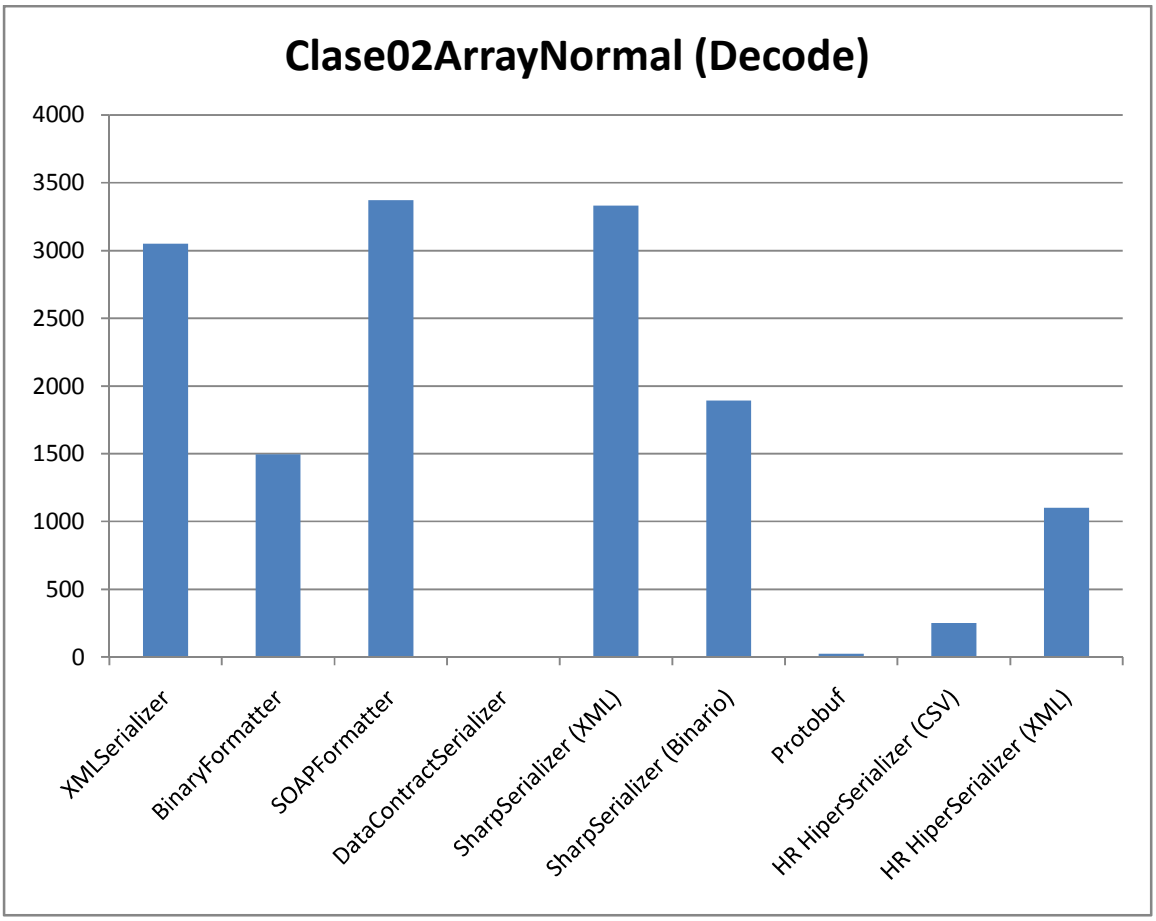


Figura 13: Comparativa de tiempos deserializando la clase Clase02ArrayNormal

No se ha podido evaluar la deserialización del serializador DataContractSerializer, al no haberse podido generar el código serializado.

Los tiempos medios recogidos al deserializar Clase02ArrayNormal son éstos:

Serializador	Tiempo en ms.
XMLSerializer	*3.049*
BinaryFormatter	*1.496*
SOAPFormatter (*)	*3.371*
DataContractSerializer	*N/A*
SharpSerializer (XML)	*3.330*
SharpSerializer (Binario)	*1.894*
Protobuf	*22*
HR HiperSerializer (CSV)	*250*
HR HiperSerializer (XML)	*1.101*

COMPARATIVA CON LA CLASE CLASE03ARRAYS. SERIALIZACIÓN

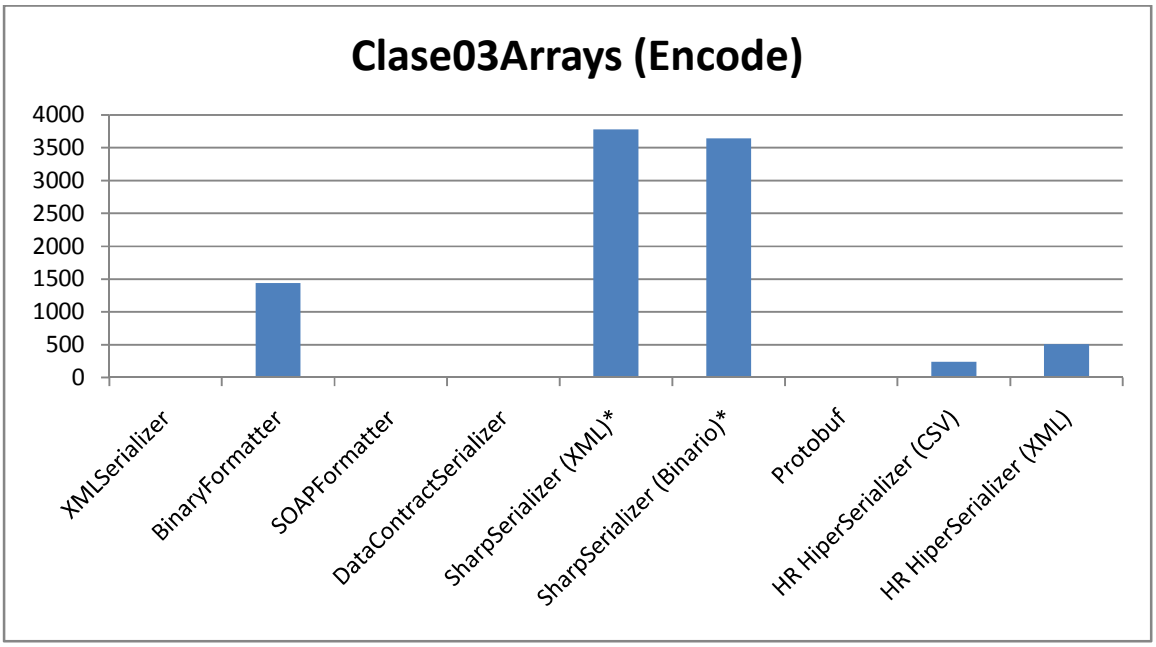


Figura 14: Comparativa de tiempos serializando la clase Clase03Arrays

(*) Los tiempos de este serializador son dos veces y media mayores que los indicados. Se han alterado para que la gráfica no se distorsionara mucho debido a la amplitud del rango. El resto de los tiempos son los correctos, y permiten identificar de un vistazo la diferencia entre cada serializador.

Para este caso el serializador XMLSerializer no se ha podido evaluar, ya que al tratar de serializar arrays multidimensionales devuelve el siguiente error:

```
An unhandled exception of type 'System.InvalidOperationException' occurred in System.Xml.dll
Additional information: There was an error reflecting type 'Fase02.Clase03Array'.
```

Para este caso el serializador SoapSerializer no se ha podido evaluar, ya que al tratar de serializar un dictionary devuelve el siguiente error:

```
Soap Serializer does not support serializing Generic Types :
System.Collections.Generic.Dictionary`2[System.String,System.Int32].
```

Para este caso el serializador DataContractSerializer no se ha podido evaluar, ya que al tratar de serializar un array multidimensional devuelve el siguiente error:

```
An unhandled exception of type 'System.NotSupportedException' occurred in
System.Runtime.Serialization.dll
Additional information: Multi-dimensional arrays are not supported
```

Para este caso el serializador Protobuf .Net no se ha podido evaluar, ya que al tratar de serializar arrays multidimensionales devuelve el siguiente error:

```
An unhandled exception of type 'System.NotSupportedException' occurred in protobuf-net.dll
Additional information: Multi-dimension arrays are supported
```

Los tiempos medios recogidos al serializar Clase03Arrays son éstos:

Serializador	Tiempo en ms.
XMLSerializer	N/A
BinaryFormatter	1.436
SOAPFormatter	N/A
DataContractSerializer	N/A
SharpSerializer (XML)*	11.327
SharpSerializer (Binario)*	10.926
Protobuf	N/A
HR HiperSerializer (CSV)	*241*
HR HiperSerializer (XML)	*508*

COMPARATIVA CON LA CLASE CLASE03ARRAYS. DESERIALIZACIÓN

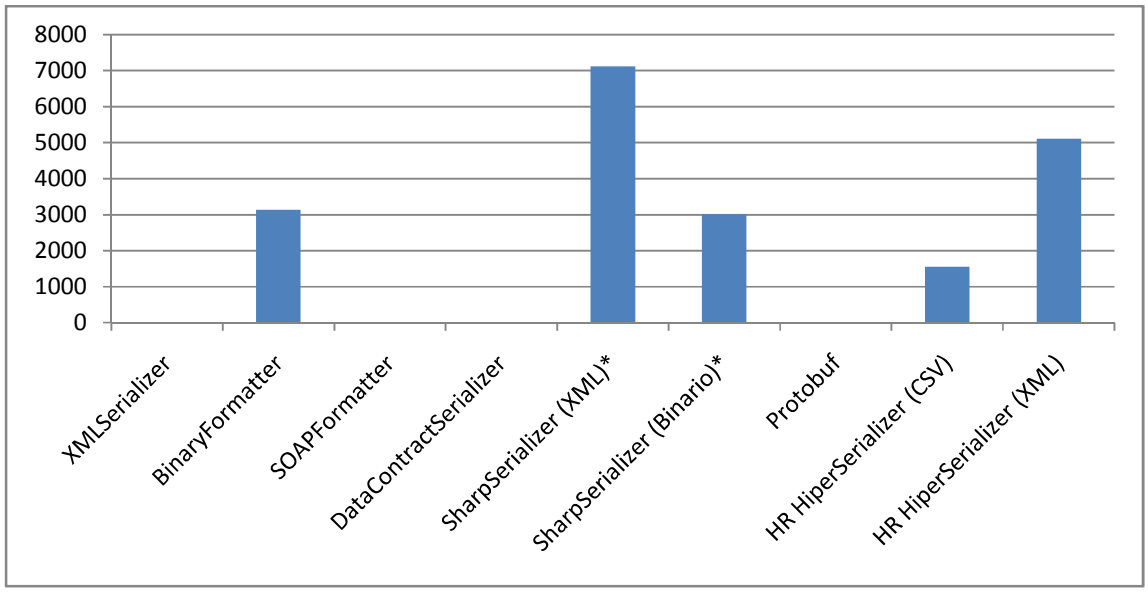


Figura 15: Comparativa de tiempos deserializando la clase Clase03Arrays

Para este caso el deserializador XMLSerializer no se ha podido evaluar, al no haberse podido generar el código serializado.

Para este caso el deserializador SoapSerializer no se ha podido evaluar, al no haberse podido generar el código serializado.

Para este caso el deserializador DataContractSerializer no se ha podido evaluar, al no haberse podido generar el código serializado.

Para este caso el deserializador Protobuf .Net no se ha podido evaluar, al no haberse podido generar el código serializado.

Los tiempos medios recogidos al deserializar Clase03Arrays son éstos:

Serializador	Tiempo en ms.
XMLSerializer	N/A
BinaryFormatter	3.132
SOAPFormatter	N/A
DataContractSerializer	N/A
SharpSerializer (XML)	7.120
SharpSerializer (Binario)	3.011
Protobuf	N/A
HR HiperSerializer (CSV)	*1.547*
HR HiperSerializer (XML)	*5.112*

COMPARATIVA CON LA CLASE CLASE04STRUCT. SERIALIZACIÓN

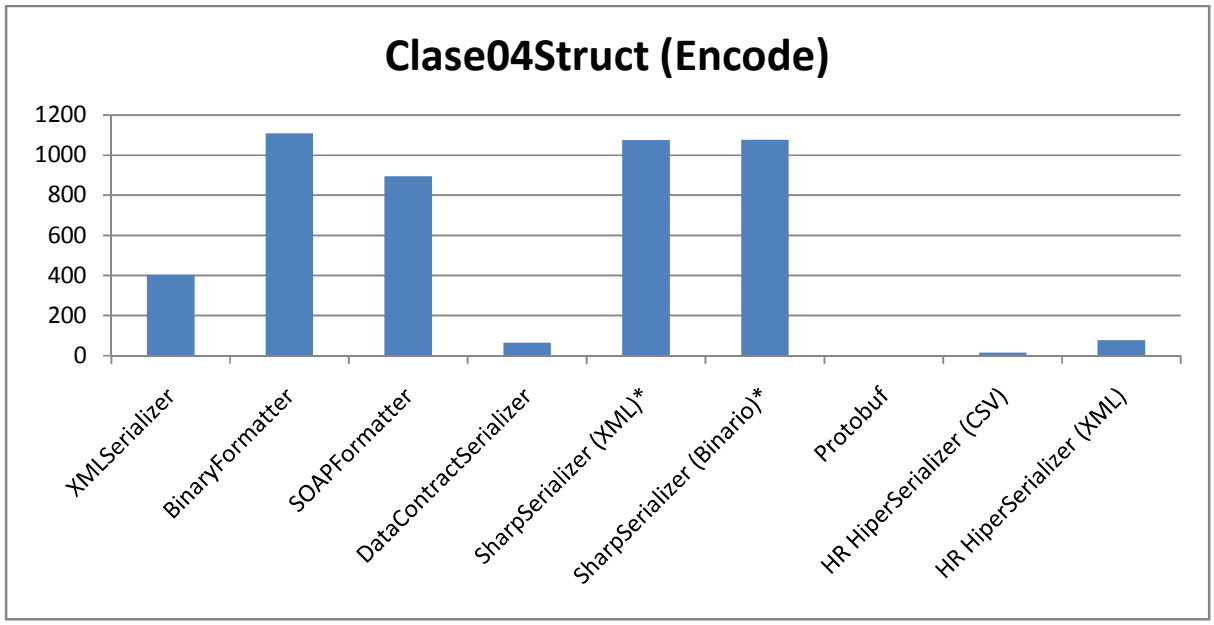


Figura 16: Comparativa de tiempos serializando la clase Clase04Struct

(*) Los tiempos de este serializador son seis veces mayores que los indicados. Se han alterado para que la gráfica no se distorsionara mucho debido a la amplitud del rango. El resto de los tiempos son los correctos, y permiten identificar de un vistazo la diferencia entre cada serializador.

Para este caso el serializador Protobuf .Net no se ha podido evaluar, ya que al tratar de serializar una estructura dentro de una clase devuelve el siguiente error:

```
An unhandled exception of type 'System.InvalidOperationException' occurred in protobuf-net.dll
Additional information: No serializer defined for type: Fase02.Clase04Struct+estructura
```

Los tiempos medios recogidos al serializar Clase04Struct son éstos:

Serializador	Tiempo en ms.
XMLSerializer	403
BinaryFormatter	1.109
SOAPFormatter	895
DataContractSerializer	65
SharpSerializer (XML)*	6.452
SharpSerializer (Binario)*	6.464
Protobuf	N/A
HR HiperSerializer (CSV)	*16*
HR HiperSerializer (XML)	*78*

COMPARATIVA CON LA CLASE CLASE04STRUCT. DESERIALIZACIÓN

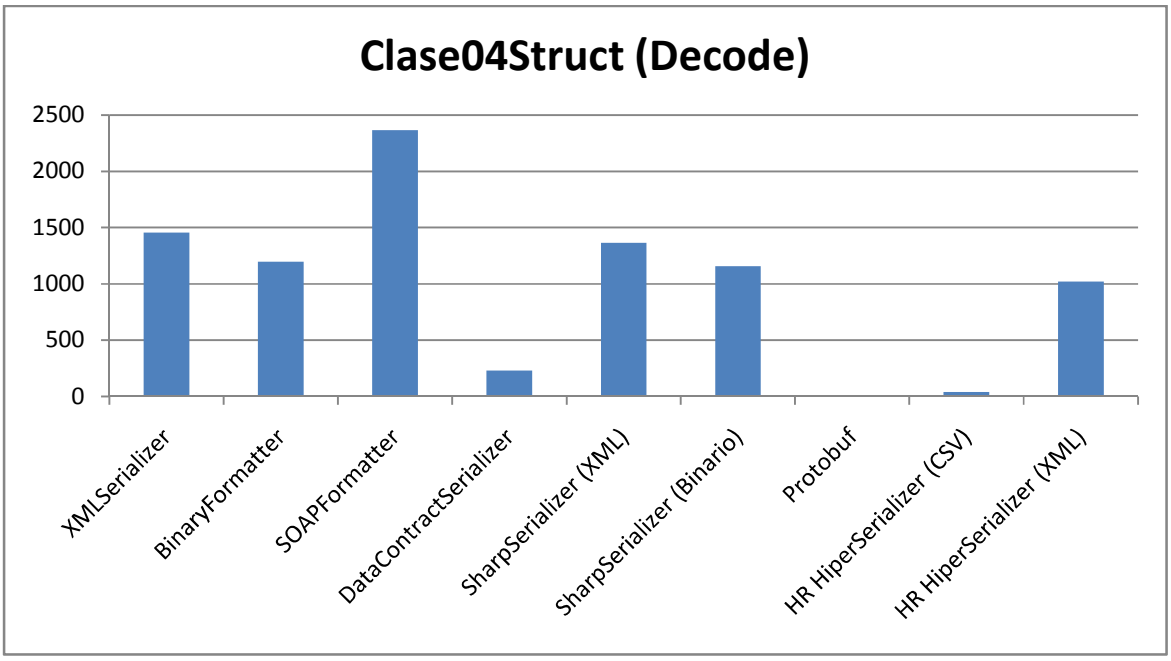


Figura 17: Comparativa de tiempos deserializando la clase Clase04Struct

Para este caso el deserializador Protobuf .Net no se ha podido evaluar, al no haberse podido generar el código serializado.

Los tiempos medios recogidos al deserializar Clase04Struct son éstos:

Serializador	Tiempo en ms.
XMLSerializer	1.455
BinaryFormatter	1.195
SOAPFormatter	2.368
DataContractSerializer	230
SharpSerializer (XML)	1.366
SharpSerializer (Binario)	1.157
Protobuf	N/A
HR HiperSerializer (CSV)	*40*
HR HiperSerializer (XML)	*1.022*

COMPARATIVA CON LA CLASE CLASE05CLASE. SERIALIZACIÓN

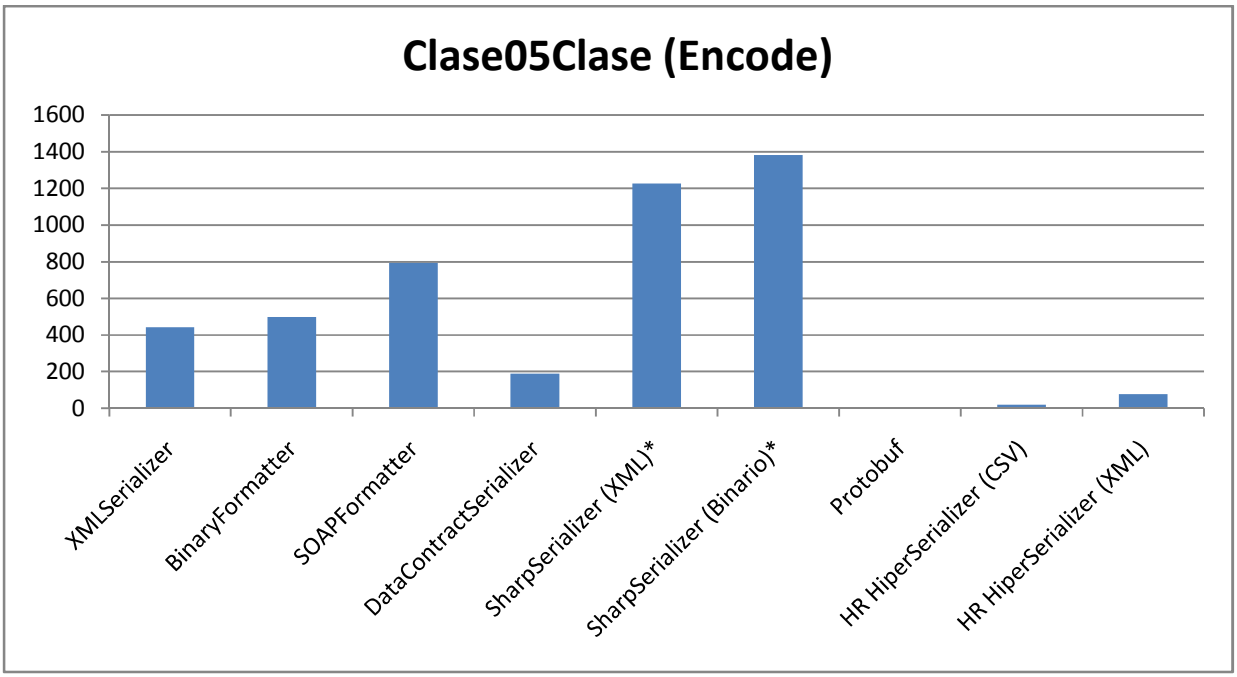


Figura 18: Comparativa de tiempos serializando la clase Clase05Clase

(*) Los tiempos de este serializador son cinco veces mayores que los indicados. Se han alterado para que la gráfica no se distorsionara mucho debido a la amplitud del rango. El resto de los tiempos son los correctos, y permiten identificar de un vistazo la diferencia entre cada serializador.

Para este caso el serializador Protobuf .Net no se ha podido evaluar, ya que al tratar de serializar una clase dentro de una clase devuelve el siguiente error:

```
An unhandled exception of type 'System.InvalidOperationException' occurred in protobuf-net.dll
Additional information: No serializer defined for type: Fase02.Clase05Clase+ClaseInterna
```

Los tiempos medios recogidos al serializar Clase05Clase son éstos:

Serializador	Tiempo en ms.
XMLSerializer	442
BinaryFormatter	499
SOAPFormatter	794
DataContractSerializer	189
SharpSerializer (XML)	6132
SharpSerializer (Binario)	6915
Protobuf	N/A
HR HiperSerializer (CSV)	*18*
HR HiperSerializer (XML)	77

COMPARATIVA CON LA CLASE CLASE05CLASE. DESERIALIZACIÓN

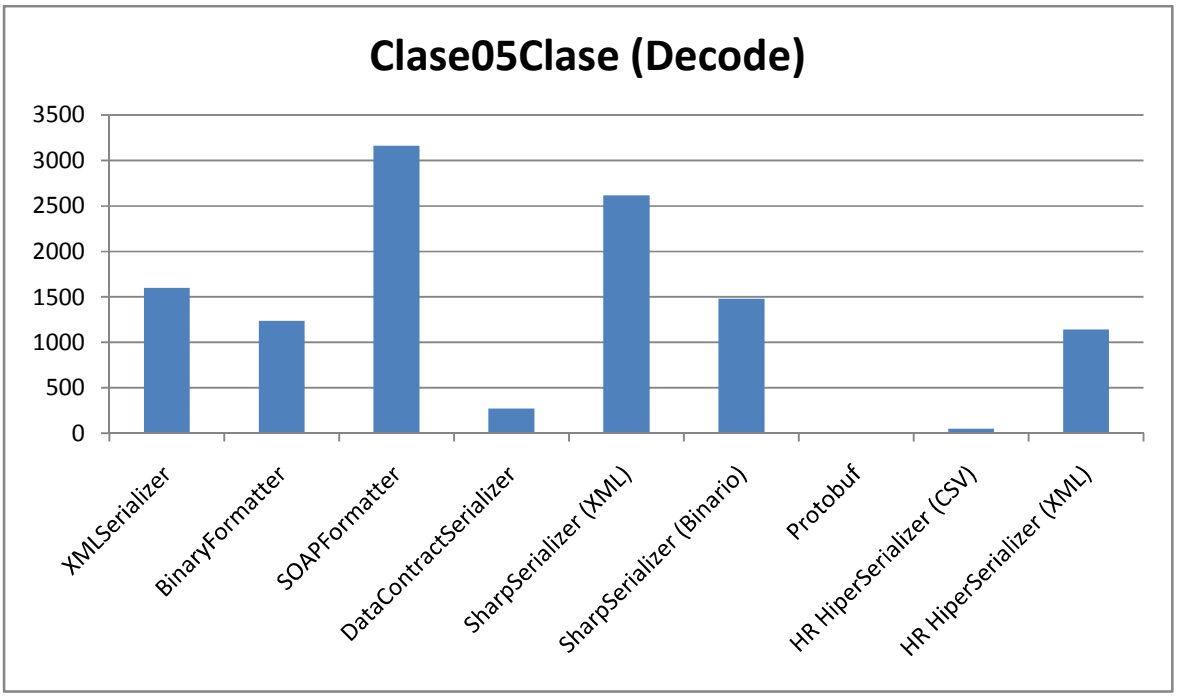


Figura 19: Comparativa de tiempos deserializando la clase Clase05Clase

Para este caso el deserializador Protobuf .Net no se ha podido evaluar, al no haberse podido generar el código serializado.

Los tiempos medios recogidos al deserializar Clase05Clase son éstos:

Serializador	Tiempo en ms.
XMLSerializer	1.596
BinaryFormatter	1.238
SOAPFormatter	3.163
DataContractSerializer	271
SharpSerializer (XML)	2.615
SharpSerializer (Binario)	1.481
Protobuf	N/A
HR HiperSerializer (CSV)	47
HR HiperSerializer (XML)	1.143

COMPARATIVA CON LA CLASE CLASE06CLASEDERIVADA. SERIALIZACIÓN

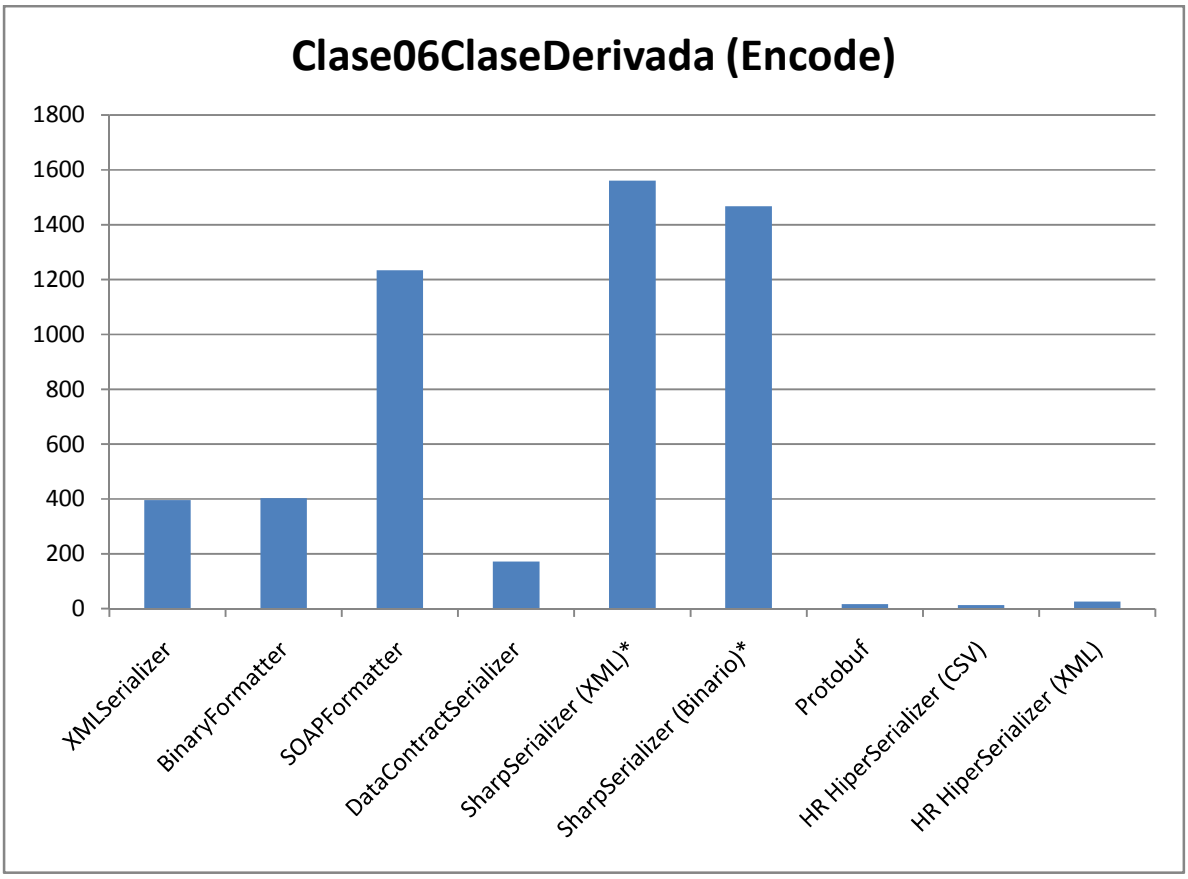


Figura 20: Comparativa de tiempos serializando la clase Clase06ClaseDerivada

(*) Los tiempos de este serializador son seis veces mayores que los indicados. Se han alterado para que la gráfica no se distorsionara mucho debido a la amplitud del rango. El resto de los tiempos son los correctos, y permiten identificar de un vistazo la diferencia entre cada serializador.

Los tiempos medios recogidos al serializar Clase06ClaseDerivada son éstos:

Serializador	Tiempo en ms.
XMLSerializer	395
BinaryFormatter	403
SOAPFormatter	1.234
DataContractSerializer	172
SharpSerializer (XML)*	9.368
SharpSerializer (Binario)*	8.801
Protobuf	16
HR HiperSerializer (CSV)	*12*
HR HiperSerializer (XML)	*26*

COMPARATIVA CON LA CLASE CLASE06CLASEDERIVADA. DESERIALIZACIÓN

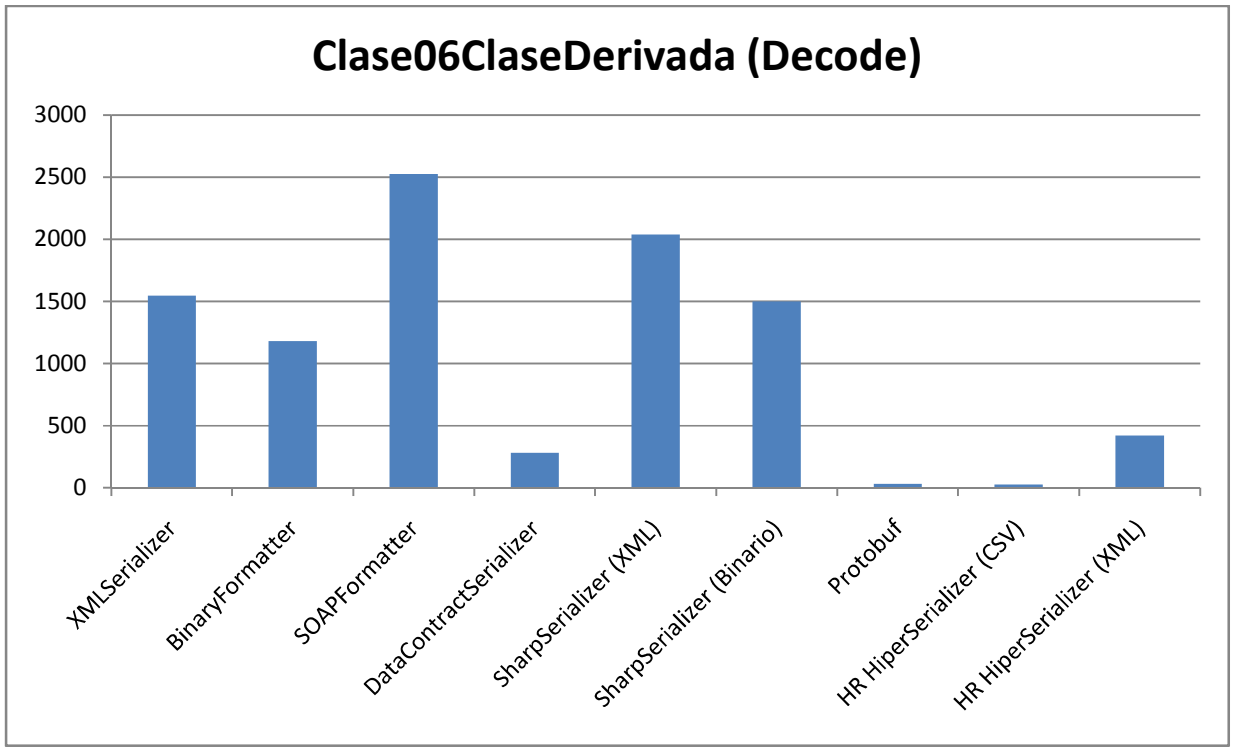


Figura 21: Comparativa de tiempos deserializando la clase Clase06ClaseDerivada

Los tiempos medios recogidos al deserializar Clase06ClaseDerivada son éstos:

Serializador	Tiempo en ms.
XMLSerializer	1547
BinaryFormatter	1182
SOAPFormatter	2526
DataContractSerializer	282
SharpSerializer (XML)	2040
SharpSerializer (Binario)	1499
Protobuf	31
HR HiperSerializer (CSV)	25
HR HiperSerializer (XML)	421

COMPARATIVA CON LA CLASE CLASE07CLASECONTODO. SERIALIZACIÓN

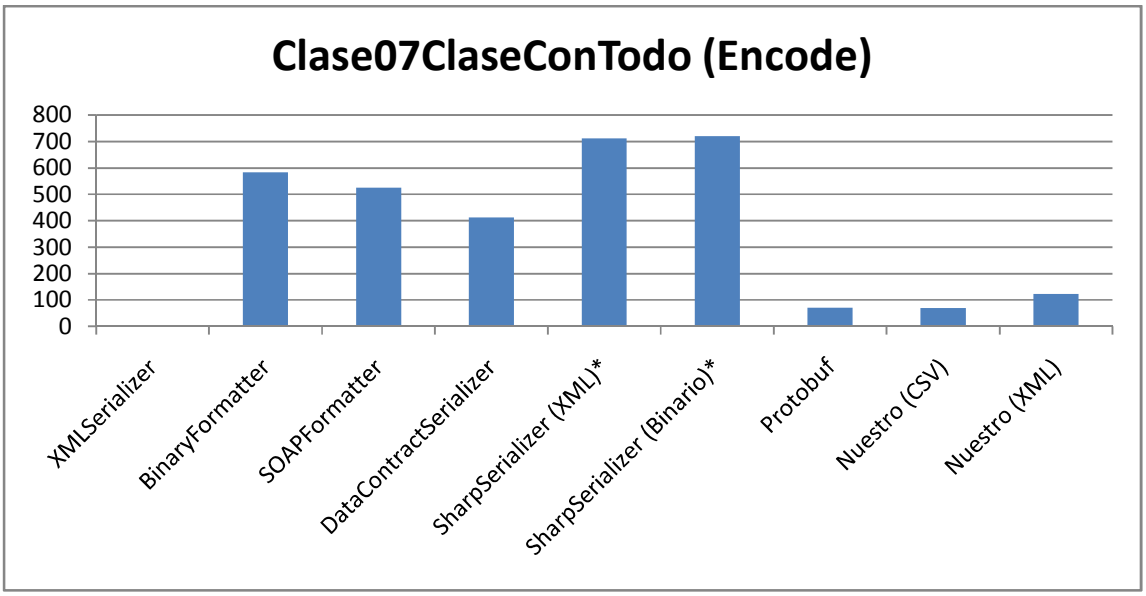


Figura 22: Comparativa de tiempos serializando la clase Clase07ClaseConTodo

(*) Los tiempos de este serializador son diez veces mayores que los indicados. Se han alterado para que la gráfica no se distorsionara mucho debido a la amplitud del rango. El resto de los tiempos son los correctos, y permiten identificar de un vistazo la diferencia entre cada serializador.

Para este caso el serializador XMLSerializer no se ha podido evaluar, ya que al tratar de serializar la clase devuelve el siguiente error:

```
An unhandled exception of type 'System.InvalidOperationException' occurred in System.Xml.dll
Additional information: There was an error reflecting type 'Fase02.Clase07ClaseConTodo'
```

Los tiempos medios recogidos al serializar Clase07ClaseConTodo son éstos:

Serializador	Tiempo en ms.
XMLSerializer	N/A
BinaryFormatter	583
SOAPFormatter	525
DataContractSerializer	412
SharpSerializer (XML)	7.116
SharpSerializer (Binario)	7.202
Protobuf	71
HR HiperSerializer (CSV)	*70*
HR HiperSerializer (XML)	*123*

COMPARATIVA CON LA CLASE CLASE07CLASECONTODO. DESERIALIZACIÓN

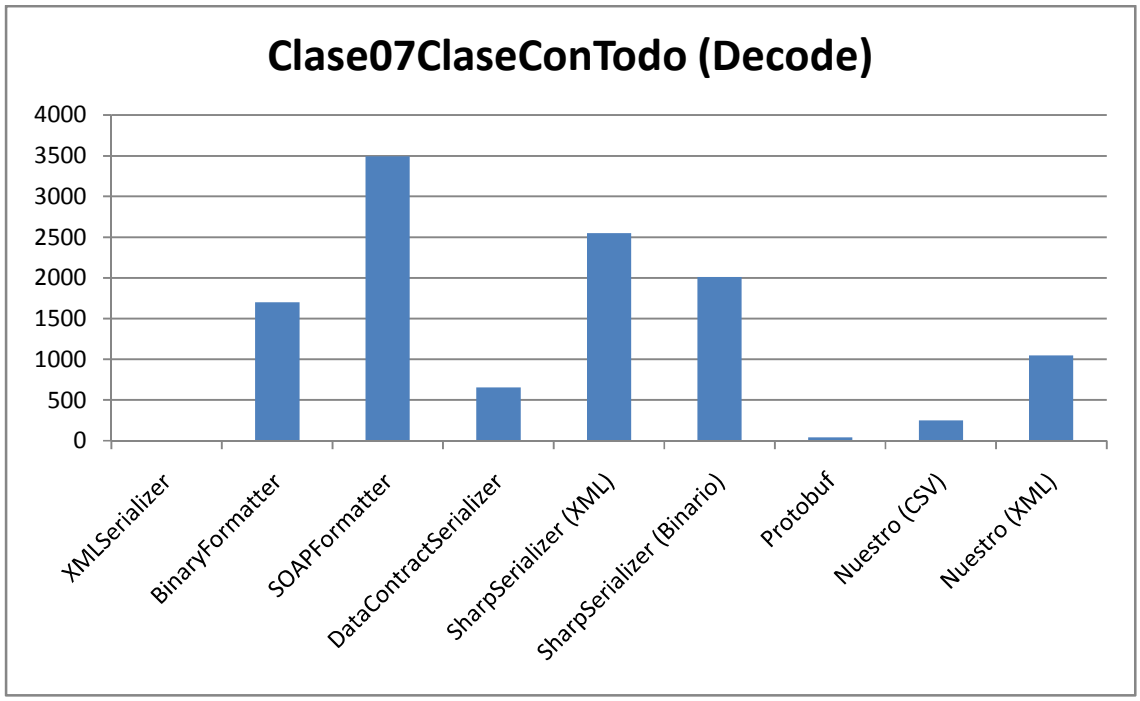


Figura 23: Comparativa de tiempos deserializando la clase Clase07ClaseConTodo

Serializador	Tiempo en ms.
XMLSerializer	
BinaryFormatter	1.699
SOAPFormatter	3.496
DataContractSerializer	656
SharpSerializer (XML)	2.548
SharpSerializer (Binario)	2.011
Protobuf	41
HR HiperSerializer (CSV)	*248*
HR HiperSerializer (XML)	*1.050*

5.2. Conclusiones

El serializador más rápido posible para una determinada clase es aquel que está programado exclusivamente para serializar esa clase. Además, será el más rápido posible si únicamente tiene que preocuparse de leer cada elemento serializable de la clase y generar la serialización correspondiente.

La deserialización será lo más rápida posible si la aplicación deserializadora sabe exactamente cómo se ha generado el código serializado para instancias de esa clase. De este modo, sabe exactamente que se va a encontrar cuando se halle ante un objeto serializado, y como obtener de la manera más rápida y segura los elementos a deserializar a partir de ese código.

El mejor serializador es aquel que permite serializar cualquier elemento de pueda contener una clase, sin limitaciones. Además debe ser escalable, para permitir cualquier tipo nuevo que se pueda añadir en futuras versiones del lenguaje sobre el que se trabaje. Y además de cualquier tipo nuevo, que sea capaz de contemplar cualquier atributo que pueda tener el código a serializar, sean atributos propios del lenguaje, o atributos personalizados o estándares de cualquier otro serializador.

Por último, será un mejor serializador aquel que permita generar el código serializado en el formato que el usuario espere o prefiera.

Con estas premisas en mente, hemos creado una aplicación cuya funcionalidad es la de generar el programa serializador particular para cualquier clase, de modo que cuando se haya generado tendremos una instancia activa de ese serializador óptimo para poder aplicar a cualquier instancia de esa clase.

Aún más, la serialización es una solo de las posibilidades de tratar la información del tipo que use la aplicación. Lo que estamos haciendo en realidad con esta aplicación es coger una instancia de un determinado tipo y mediante la serialización codificar su información de una determinada manera, y mediante la deserialización realizar la decodificación para obtener otra instancia exactamente igual.

Pero se podría aplicar el mismo mecanismo si lo que nos interesara fuera transformar una determinada información en la misma información pero con otro formato. Y esta aplicación podría ser la base para conseguir este fin.

La principal complejidad ha consistido en ser capaz de discernir cómo tenía que comportarse el programa para ir generando el código de otro programa que finalmente es el que hará la funcionalidad objetivo. En ocasiones ha sido un poco desquiciante.

6
Mejoras futuras

6.1. Serialización de los elementos privados

Este serializador solo es capaz de serializar los elementos públicos del tipo con el que trabaje. Como premisa básica, aquellos elementos de una clase con los que se puede trabajar es con los públicos.

Para la serialización no habría problema en trabajar con los elementos privados y protegidos, ya que Reflection permite acceder a ellos, incluso a sus valores. Pero el problema está en el método que se ha usado para deserializar. La deserialización se hace a partir de una instancia del mismo tipo, a la que se van asignando los valores serializados a cada elemento en el mismo orden en el que se serializó.

La propia definición de elementos privados o protegidos impide que desde una aplicación exterior se asignen valores a estos elementos.

Por tanto, para conseguir la deserialización de estos elementos habría que usar otra estrategia, consistente en utilizar Reflection para acceder a los elementos privados y protegidos y a través del método setValue asignar cada valor a cada elemento de la instancia con que se trabaje.

La pega que tiene este mecanismo, y por eso no se ha utilizado en esta aplicación, es que aumenta mucho el tiempo de ejecución. Usar Reflection es muy costoso en tiempo, y la decisión que tomamos fue utilizarlo en la generación del serializador, pero no en la ejecución del mismo, para no penalizar la rapidez en su ejecución.

6.2. Añadiendo plug-ins para distintas representaciones (json, binary, etc.)

6.2.1. Plug-ins para distintas representaciones (json, binary, etc.)

Para favorecer que la salida del Encode se pueda generar en diferentes formatos se puede utilizar un mecanismo de plug-ins. Consiste en encapsular las diferentes maneras de ejecutar la codificación (y por tanto la decodificación) dentro de ensamblados diferentes, y a la hora de ejecutar la aplicación, se cargará y ejecutará una u otra según la codificación que se desee.

La arquitectura del sistema está preparada para que se pueda ampliar el tipo de serialización que puede realizar. Por defecto realiza serializaciones en XML y CSV, pero se pueden incorporar mediante este mecanismo de plug-ins cualquier otro tipo de serialización (JSON, binario, otros esquemas de XML, etc.)

Un plug-in es una funcionalidad que se puede agregar (se enchufa) a una aplicación para realizar acciones que previamente la aplicación no hacía, o hacía de forma distinta. Y lo fundamental: se pueden incorporar estas funcionalidades sin necesidad de recompilar el proyecto original.

Además, esta funcionalidad puede ser modificada o sustituida por otro plug-in que se integraría en el sistema de la misma manera. Es un método muy útil para incrementar las funcionalidades de una aplicación sin necesidad de redistribuirla o de generar nuevas versiones de la misma. Además, permite que terceros puedan añadir su propia funcionalidad o su propia forma de realizar el proceso objeto del plug-in de manera que se ajuste a sus necesidades (otros tipos de codificación, traducción de tipos de objeto, etc.).

Una posible mejora para la aplicación consiste en conseguir que se pueda ampliar su funcionalidad con la inserción de distintos plug-ins cuya funcionalidad sea la de modificar la manera en que se realiza tanto la codificación como la descodificación de los datos, adaptándola al formato que el plug-in prevea.

6.2.2. Plug-ins para otras funcionalidades

Otra posibilidad consistiría en conseguir a partir de un plug-in que el serializador se convirtiera en un traductor, que recibiera un determinado tipo de datos y lo transcribiera en otro distinto. O que al generar el código serializado para un determinado objeto lo haga de tal manera que simule o sea de facto otro formato distinto del mismo objeto. Por ejemplo, una imagen en jpg se puede serializar cambiando sus cabeceras para que el código serializado sea de hecho la misma imagen en formato png.

Para añadir a nuestra aplicación un mecanismo de plug-ins que admita cualquier otra funcionalidad, es necesario tener en cuenta tres aspectos:

- Definir un contrato para los plug-ins. A partir de un interface se define como admitirá el sistema la inserción de un nuevo plug-in.

- Implementación de las tareas. Las tareas definidas en el interface anterior tienen que implementarse.

- Carga de plug-ins. Un loader se encargará de cargar en cada momento el plug-in específico que tenga que realizar esas tareas, normalmente en función de una comprobación inicial de existencia o llamada a un determinado plug-in.

La carga de los plug-ins se realizaría a partir de un **Plug-in Manager**. Éste manager se encarga de construir el sistema a partir de la información de plug-ins que se encuentra en el mismo. Podemos hacer que el mecanismo de plug-in identifique y monte los plug-ins necesarios en la aplicación almacenándolos todos en un mismo directorio. A continuación

habría que programar que el programa busque en dicho directorio (y recursivamente en sus subdirectorios) y aplique todos los plug-ins que encuentre. En sí mismo el Plug-in Manager es un plug-in, aunque un tanto especial.

Se podría mejorar aún más el sistema utilizando un mecanismo de atributos personalizados que identifiquen los plug-ins a cargar, dónde se encuentran, e incluso las dependencias entre varios plug-ins para cargarlos todos.

Para identificar los plug-ins se utilizaría nuevamente Reflection, recorriendo un determinado directorio donde los "candidatos" a plug-in estarían almacenados. Reconociendo las clases en cada fichero del directorio, e identificando las que implementen la interfaz definida para los plug-ins (por ejemplo IPlugin), localizaríamos los plug-ins válidos. Así, serían cargados todos los plug-ins que implementen dicha interfaz, y serían desechados los que no lo hicieran.

La nueva arquitectura de la aplicación para admitir esta funcionalidad se compondría de varias capas:

- un núcleo del sistema, que se encargará de cargar el Plug-in Manager y el resto de servicios necesarios para que la aplicación pueda arrancar y funcionar.

- un conjunto de plug-ins adicionales que se podrán cargar en el sistema según las necesidades del mismo a través del Plug-in Manager, para ser invocados a voluntad.

- una inteface que defina los métodos que tuviera que implementar el plug-in. Otra posibilidad es sustituir esta dependencia del inteface por genéricos.

6.2.3. Sistema de plug-ins avanzado

Este mecanismo de plug-ins es muy básico y se puede ampliar en cuanto a sus características y funcionalidades, siguiendo parecidos mecanismos que los usados para su funcionamiento básico.

Un ejemplo podría ser el establecimiento de un mecanismo de dependencias por el cual se reconociera, para un determinado plug-in, qué otros plug-ins es necesario que estuvieran cargados antes de poder cargar aquél, incluso tomando en cuenta versiones de los mismos, lo que haría más afinado el sistema de dependencias. Tan solo se necesitaría definir los atributos adecuados (o parámetros en los atributos existentes) para identificar estas dependencias. Por ejemplo:
[IPluginDependence("NombrePlugin", "número versión")]

Otro ejemplo, se podría establecer un mecanismo de **Hooks** y **Servicios** (exposición de eventos que mediante un patrón suscriptor-publicador permite que se realicen distintas acciones al producirse dicho evento). Así, por ejemplo, se podría indicar que antes de sacar el código con la serialización de un objeto que va a ser transportado por la red, se encriptara el contenido, y que antes de la decodificación se desencriptara. Podría haber un plug-in que escuchara este evento y que lanzara los hooks que ejecuten la encriptación y la desencriptación de manera transparente cuando hiciera falta.

Nuevamente, para realizar esta tarea nos apoyaríamos en atributos personalizados, para identificar los eventos que pueden ser atendidos por estas utilidades, y los interceptores (hooks) que se lanzarán cuando dicho evento se produzca.

Ahondando en este mecanismo de hooks, se podría dar una vuelta de tuerca más, indicando en los atributos personalizados un mecanismo de “pesos” que estableciera en qué orden tendrían que ejecutarse para un mismo evento todos los posibles hooks que se suscriban a él.

Permitir la ejecución del plug-in en un thread independiente. Se puede tomar en consideración esta situación, y definir en los plug-in insertables en esta aplicación si se desea que su ejecución se realice en un thread independiente. Puesto que la funcionalidad de los plug-ins que se van a usar consisten en la ejecución de uno u otro mecanismo de serialización, se puede hacer que cada ejecución se realice en un thread independiente sin perjudicar al funcionamiento del resto del sistema. Esto podría suponer una mejora considerable en el tiempo de ejecución, ya que permitiría la serialización en paralelo de más de un objeto.

Y para conseguir esto sólo sería necesario incorporar en el plug-in una propiedad de tipo booleano que en función de su valor hiciera al sistema comportarse de una o de otra manera, generando o no un thread por cada ejecución de las acciones desarrolladas en el plug-in. (Véase referencias [57] a [64])

6.2.4. RELLENAR ARRAYS MULTIDIMENSIONALES EN LA FUNCIÓN DECODE

Una de las más costosas y engorrosas funcionalidades de los programas serializadores que nuestra aplicación genera es la de lidiar con arrays multidimensionales.

Se ha establecido un mecanismo para conseguir de manera rápida serializar todos los contenidos de un array multidimensional aprovechando la característica de los arrays de poder ser recorridos secuencialmente con un foreach.

Pero para la deserialización, la cosa se complica, ya que aunque se conozca de antemano el número de elementos de cada rango en el array multidimensional, técnicamente es costoso rehacer el array con cada elemento en su posición original.

Ya al final del desarrollo brotó una posibilidad, que no se ha contemplado para mejorar el rendimiento de esta parte del proceso. Consiste en utilizar el método Array.Copy como alternativa a los bucles for para volcar todo el contenido del array de una sola vez. No ha sido probado, ni comprobada su mejoría en cuanto a complejidad ciclomática, pero se apunta aquí como una posible mejora a tener en cuenta. (Ver [65])

La forma de utilizar esta funcionalidad es la siguiente:

```
Array.Copy(ar1, im1, ar2, im2, num)
ar1: array1
im1: índice mímino (GetLowerBound)
ar2: array2
im2: índice mínimo
num: número de elementos a copiar
```

6.2.5. Programación asíncrona

El principal problema es el tiempo a la hora de serializar y deserializar objetos. Se trata de que estos procesos sean lo más rápido posible (de alto rendimiento). Además, cuando los tipos que tengamos que procesar tengan varias clases generadas por el usuario en su interior, se tendrán que serializar varios tipos de objeto de manera consecutiva. Para conseguir mejores tiempos podríamos realizar el proceso de serialización de manera Asíncrona.

Para serializar las variables, por ejemplo float, hay que tener en cuenta si la codificación de su valor al convertirse a binario se realiza con formato big-endian ó little-endian.
Esto se indicará en los atributos que se coloquen en la propiedad o variable que se trate de serializar. (Ver [66])

6.2.6. Mejoras en la serialización en XML

Para la serialización en XML, se podría generar a la vez el esquema válido para el documento. En nuestra aplicación no se contempla el uso y/o la generación de un esquema para validar el XML que genera la serialización. Podría aprovecharse que se va generando dinámicamente el código XML para generar a la vez el esquema al que se adaptará dicho XML.

Añadir la codificación binaria con encriptación, compresión, comprobación checksum, y cualquier otra variante que resulte interesante.

6.2.7. Uso de LINQ

Se podría mejorar usando la característica de .Net 3.5 LINQ, característica que no he usado en este proyecto, pese a que está preparado para .Net 3.5.

Anexos

ANEXO A. CARACTERÍSTICAS DE LOS DISTINTOS SERIALIZADORES EN .Net

Se identifican dos tipos de serializadores y deserializadores de objetos para ser compartidos por una conexión, según la documentación oficial de .Net:

- Serializadores **de tipo compartido**, en los que se asume que la máquina que va a deserializar tiene los Assemblys necesarios para identificar los tipos que se deserialicen. Son mejores para implementar herencia en los objetos compartidos, pero peores en cuanto a rendimiento (se explica en cada mensaje el tipo de objetos que se envía) y en cuanto a que no son multiplataforma (no se puede deserializar algunos tipos desde otra plataforma que no sea .Net).

- Serializadores **de contrato compartido**, en los que el servidor (serializador) y el cliente (deserializador) saben de antemano qué tipos de datos se van a transmitir serializados. Esto se puede establecer de manera manual, con una conexión previa que envía esa información, o utilizando mecanismos SOA, mediante los cuales el servidor expone la definición de los objetos a serializar para que el cliente lo conozca antes de realizar la conexión. Incluso no es necesario que los tipos sean exactamente los mismos, sino que tienen que coincidir en la definición que el servidor publica de los mismos para poder ser compartidos (acoplamiento débil).

	TIPOS COMPARTIDOS	**CONTRATOS COLECTIVOS**
Binario	BynaryFormatter	
XML	NetDataContractSerializer	XmlSerializer DataContractSerializer
Json		DataContractJsonSerializer

El único conflicto se produce cuando se desea un serializador en XML con características de contrato colectivo, ya que para eso contamos con dos serializadores. DataContractSerializer sería más simple (no se pueden usar atributos en las etiquetas generadas), y si lo que necesitamos es generar un XML que se ajuste a determinada hoja de estilos, usaremos XmlSerializer.

ANEXO B. MODO DE USO DE LOS DISTINTOS SERIALIZADORES

En el benchmark de comparación de la rendimiento de los serializadores existentes en el mercado respecto a nuestro proyecto, utilizamos todos y cada uno de ellos. A continuación se describe como se utiliza cada serializador de los seleccionados para las pruebas:

- XML Serializer se usa así:

```
XmlSerializer serializer = new XmlSerializer(typeof(Clase01Basica));
TextWriter writer = new StreamWriter("fichero.txt");
serializer.Serialize(writer, instanciaClase01Basica);
writer.Close();

XmlSerializer serializer = new XmlSerializer(typeof(Clase01Basica));
FileStream fs = new FileStream("fichero.txt", FileMode.Open);
Clase01Basica clase;
clase = (Clase01Basica) serializer.Deserialize(fs);
```

- DataContractSerializer se usa así: (No admite matrices multidimensionales)

```
MemoryStream stream1 = new MemoryStream();
//Serialize the Record object to a memory stream using DataContractSerializer.
DataContractSerializer serializer = new DataContractSerializer(typeof(Clase01Basica));
serializer.WriteObject(stream1, instanciaClase01Basica);

stream1.Position = 0;
Clase01Basica clase = (Clase01Basica)serializer.ReadObject(stream1);
```

- SharpSerializer se usa así:

```
var serializer = new SharpSerializer();
serializer.Serialize(instanciaClase01Basica, "test.xml"); // serialize
Clase01Basica clase = serializer.Deserialize("test.xml"); // deserialize

// true - binary serialization, false - xml serialization
var serializer = new SharpSerializer(true); // serialize
serializer.Serialize(instanciaClase01Basica, "test.bin"); // deserialize
Clase01Basica clase = serializer.Deserialize("test.bin");
```

- Protobuf-net se usa así:(requiere que la clase tenga el atributo [ProtoContract] y los miembros [ProtoMember(1)]. Además, no soporta arrays multidimensionales)

```
// write to a file
Serializer.Serialize(outputStream, person);

// read from a file
var person = Serializer.Deserialize<Person>(inputStream);
```

- BinaryFormatter ser usa así: (requiere que la clase tenga el atributo [Serializable])

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
FileStream fs = new FileStream("DataFile.dat", FileMode.Create);
// Construct a BinaryFormatter and use it to serialize the data to the stream.
BinaryFormatter formatter = new BinaryFormatter();
try
{
	formatter.Serialize(fs, instanciaClase01Basica);
}
catch (SerializationException e)
{
	Console.WriteLine("Failed to serialize. Reason: " + e.Message);
	throw;
}
finally
{
	fs.Close();
}
FileStream fs = new FileStream("DataFile.dat", FileMode.Open);
try
{
	BinaryFormatter formatter = new BinaryFormatter();
	clase = (Clase01Basica) formatter.Deserialize(fs);
}
catch (SerializationException e)
{
	Console.WriteLine("Failed to deserialize. Reason: " + e.Message);
	throw;
}
finally
{
	fs.Close();
}
```

- SOAPFormatter se usa así:(requiere que la clase tenga el atributo [Serializable] y no admite la serialización de genéricos (tales como Dictionary)).

```
FileStream fs = new FileStream("DataFile.soap", FileMode.Create);

// Construct a SoapFormatter and use it
// to serialize the data to the stream.
SoapFormatter formatter = new SoapFormatter();
try
{
	formatter.Serialize(fs, instanciaClase01Basica);
}
catch (SerializationException e)
{
	Console.WriteLine("Failed to serialize. Reason: " + e.Message);
	throw;
}
finally
{
	fs.Close();
}

FileStream fs = new FileStream("DataFile.soap", FileMode.Open);
try
{
	SoapFormatter formatter = new SoapFormatter();

	clase = (Clase01Basica) formatter.Deserialize(fs);
}
catch (SerializationException e)
{
	Console.WriteLine("Failed to deserialize. Reason: " + e.Message);
	throw;
}
finally
{
	fs.Close();
}
```

- Nuestro proyecto se usa así:

```
// “FORMATO” es opcional. Por defecto XML, y por plug-in se puede usar cualquiera que se desee
Type tipo = claseASerializar.GetType();
Generador g = new Generador(tipo, “FORMATO (CSV,BINARIO,etc)”);
dynamic serializador = g.getSerializer();

stream codigo = serializador.codificar(claseASerializar);
serializador.decodificar(codigo, ref clase);
```

ANEXO C. CONCLUSIONES PARTICULARES

Este proyecto ha sido una gran oportunidad de poner en práctica muchos de los conocimientos adquiridos durante la carrera que ahora acabo. Desde el momento en que el tutor, Agustín Santos, comenzó a hablarme del proyecto en el que estaba embarcado, y para el que le surgió la necesidad de crear un serializador particular, me interesó muchísimo su propuesta. Y más aún cuando supe que el entorno en el que iba a desarrollarlo sería .Net.

Durante la carrera he tenido pocas oportunidades de lidiar con un entorno tan complejo y completo. Y aunque profesionalmente ya tenía cierta experiencia programando en C# y VB.Net, lo cierto es que hacía mucho que no me había tropezado con este entorno y sus lenguajes. Es por esto que acepté enseguida, y una de las principales razones fue porque el reto sería mayor si lo tenía que superar en este entorno.

El proyecto me ha ayudado a valorar mucho más, a reparar de nuevo en las bondades de esta herramienta. Hay que reconocerle a la tantas veces denostada y vilipendiada Microsoft,al albur del auge del código libre, que sabe lo que se hace cuando hablamos de programación y sus entornos. No solo la potencia del framework y sus innumerables ayudas y utilidades, sobre todo en cuestión de debug, sino que la potencia de los lenguajes y las librerías que ponen a nuestra disposición, permiten que se pueda realizar prácticamente cualquier cosa programando con sus lenguajes.

En este sentido, ha sido para mí un descubrimiento averiguar, no solo que C# permitiera la compilación y la instanciación en tiempo de ejecución, sino averiguar que posee hasta tres formas distintas de conseguir esa quimera (CodeDOM, EMIL y árboles de expresión).

Lo mismo me ha sucedido al entender cómo funciona una librería como Reflection. Es sorprendente cómo se puede en tiempo de ejecución bucear por todo el contenido y los entresijos de cualquier objeto con el que se esté trabajando, capturar sus elementos con todas sus características, incluso invocar a sus métodos de una forma dinámica, aunque no supiésemos que los tenía. Tiene una potencia increíble.

Quiero dejar una mención especial para el tutor. Agustín Santos, profesor investigador becado en la Universidad de Harvard, me ha enseñado lo más importante que he aprendido en este proyecto: cómo pensar cuando se está investigando, en contraposición al proceso de desarrollo normal de una solución comercial. Por supuesto, su apoyo ha sido fundamental en la conclusión de este modesto proyecto, pero lo que me ha aportado verle pensar, verle plantear las situaciones, los pasos a seguir, identificar lo que importaba y lo que era superfluo, ha sido una de las lecciones más importantes que he recibido durante este proceso.

Me quedo con esas enseñanzas.

Todo el proyecto se puede descargar de https://github.com/vitiparra/ProyectoFinDeCarrera

BIBLIOGRAFÍA

SERIALIZACIÓN
[1] - El chip más rápido del mundo es un serializador:
http://www.neoteo.com/19404-el-chip-mas-rapido-del-mundo-esta-en-el-lhc

[2] - Serialización predecible. Otra óptica que se aproxima a la nuestra (capítulo 5):
http://www.ctr.unican.es/jtr12/articulos/13-daniel-tejera.pdf

[3] - Importancia de la serialización en sistemas distribuidos:
http://www.dia.eui.upm.es/asignatu/sis_dis/Paco/Comunicacion.pdf

SERIALIZADORES ANALIZADOS Y COMPARADOS
[4] - Comparativa de la ejecución de serialización de algunos serializadores:
http://mono.servicestack.net/benchmarks/NorthwindDatabaseRowsSerialization.100000-times.2010-08-17.html

[5] - Prueba que muestra un ejemplo de distintos serializadores aplicados a un objeto normal.
http://blogs.msdn.com/b/youssefm/archive/2009/07/10/comparing-the-performance-of-net-serializers.aspx

[6] - An overview of .Net serializers:
http://blogs.msdn.com/b/youssefm/archive/2009/04/15/an-overview-of-net-serializers.aspx

[7] - Microsoft .Net posee una amplia variedad de serializadores.
http://blogs.msdn.com/b/youssefm/archive/2009/07/10/comparing-the-performance-of-net-serializers.aspx

[8] - Listado de los tipos de elementos con los que podemos encontrarnos dentro de un objeto:
https://msdn.microsoft.com/es-es/library/ybcx56wz.aspx

[9] - Información sobre XML Serializer
http://msdn.microsoft.com/es-es/library/bb552764%28v=vs.90%29.aspx

[10] - Ejemplos de XMLSerializer:
https://msdn.microsoft.com/en-us/library/58a18dwa%28v=vs.110%29.aspx

[11] - DataContractSerializer
https://msdn.microsoft.com/es-es/library/ms731073%28v=vs.110%29.aspx

[12] - Características de DataContractSerializer:
https://msdn.microsoft.com/es-es/library/system.runtime.serialization.datacontractserializer(v=vs.110).aspx
http://www.sitefinity.com/documentation/documentationarticles/developers-guide/deep-dive/client-side-programming/implementing-your-own-restful-wcf-service/serialization-with-datacontractserializer

[13] - Tipos de datos serializables con DataContractSerializer:
http://msdn.microsoft.com/en-us/library/ms731923.aspx
https://msdn.microsoft.com/es-es/library/ms731923(v=vs.110).aspx

[14] - Serialización con NetSerializer:
http://www.codeproject.com/Articles/351538/NetSerializer-A-Fast-Simple-Serializer-for-NET
https://github.com/tomba/netserializer

[15] - Serialización con ProtoBuf:
https://code.google.com/p/protobuf-net/
http://code.google.com/p/protobuf-net/wiki/Performance
https://code.google.com/p/protobuf-net/wiki/Capabilities
http://stackoverflow.com/questions/3462775/for-which-scenarios-is-protobuf-net-not-appropriate/3462828#3462828

[16] - Existen muchos otros serializadores:
https://staging.nuget.org/packages?q=Serializer

FASE 1 DEL DESARROLLO

[17] - Compiling and running code at runtime
http://simeonpilgrim.com/blog/2007/12/04/compiling-and-running-code-at-runtime/

[18] - Generación y compilación dinámica de código fuente
https://msdn.microsoft.com/es-es/library/ms404245%28v=vs.110%29.aspx

[19] - Microsoft .Net CodeDOM Technology
http://www.codeguru.com/vb/gen/article.php/c19573/Microsoft-NET-CodeDom-Technology.htm
http://www.codeproject.com/Articles/7119/Compiling-with-CodeDom

[20] - Compilando en tiempo de ejecución
http://foro.elhacker.net/net/compilando_en_tiempo_de_ejecucion_iquestmake_builder_c-t313428.0.html;msg1553157

[21] - Generación y compilación dinámicas de código fuente
http://msdn.microsoft.com/es-es/library/650ax5cx(v=vs.110).aspx

[22] - Referencias sobre como compilar una clase con CodeDOM:
http://www.codeproject.com/Articles/7119/Compiling-with-CodeDom

[23] - Mejor utilizar CodeComProvider que ICodeCompiler:
http://stackoverflow.com/questions/14406049/codedomprovider-createcompiler-is-obsolete
http://bytes.com/topic/c-sharp/answers/722688-codedomprovider-createcompiler-obsolete-fix
https://msdn.microsoft.com/es-es/library/system.codedom.compiler.codedomprovider(v=vs.110).aspx

FASE 2 DEL DESARROLLO
[24] - La base de los métodos de extensión.
http://msdn.microsoft.com/es-es/library/bb383977.aspx

[25] - Como implementar e invocar un método de extensión personalizado.
http://msdn.microsoft.com/es-es/library/bb311042.aspx

FASE 3 DEL DESARROLLO

[26] - Diferencias entre campos y propiedades:
http://www.mundonet.es/campos-propiedades-ejercicio-1.html?Pg=Entrega8.htm

[27] - Utilizar propiedades:
https://msdn.microsoft.com/es-es/library/w86s7x04.aspx

[28] - Identificar un Struct por reflection:
http://msdn.microsoft.com/en-us/library/bfft1t3c.aspx

[29] - Ejemplos de identificación de valores nulos para estructuras:
http://stackoverflow.com/questions/2713900/how-to-determine-if-a-net-type-is-a-custom-struct
http://stackoverflow.com/questions/2296288/how-to-decide-a-type-is-a-custom-struct

[30] - El coste de usar Reflection
http://stackoverflow.com/questions/25458/how-costly-is-net-reflection
http://www.nuget.org/packages/FastMember (plug-in que acelera el acceso)

[31] - Como obtener las propiedades y los métodos
http://stackoverflow.com/questions/12680341/how-to-get-both-fields-and-properties-in-single-call-via-reflection

FASE 4 DEL DESARROLLO
[32] - Sacar elementos de array en C# no es sencillo:
http://stackoverflow.com/questions/455237/pop-off-array-in-c-sharp

[33] - Instanciar Array genérico con uno o más rangos:
https://msdn.microsoft.com/es-es/library/zb3cfh7k%28v=vs.110%29.aspx

[34] - Uso de XMLReader para recorrer secuencialmente un documento XML:
https://msdn.microsoft.com/es-es/library/system.xml.xmlreader.readouterxml(v=vs.110).aspx

[35] - Otro punto de vista sobre la clase Array:
https://books.google.es/books?id=_Y0rWd-Q2xkC&pg=PA239&lpg=PA239#v=onepage&q&f=false

[36] - Converting string to type c#:
http://stackoverflow.com/questions/11107536/convert-string-to-type-c-sharp

[37] - Convertir una cadena en número (o cualquier otro tipo de dato básico):
https://msdn.microsoft.com/es-es/library/bb397679.aspx

[38] - Ver si se puede utilizar GetCustomAttributes para obtener cosas que no tengo hasta ahora:
http://geekswithblogs.net/sdorman/archive/2010/05/16/retrieving-custom-attributes-using-reflection.aspx

[39] - PROCESAR GENERIC (Dictionary<,>)
https://msdn.microsoft.com/en-us/library/b8ytshk6(v=vs.110).aspx

[40] - XMLSerializador no puede serializar Dictionarys de manera natural:
https://msdn.microsoft.com/en-us/library/ms950721.aspx

[41] - Utilizar foreach con matrices
https://msdn.microsoft.com/es-es/library/2h3zzhdw.aspx
http://www.dotnetperls.com/array-foreach

[42] - Intento de modificación de un array dentro de un foreach (fallido)
http://stackoverflow.com/questions/5449347/c-sharp-array-foreach-alternative-that-replaces-elements

[43] - Como no se puede modificar los elementos de un array:
http://www.dotnetperls.com/array-foreach

[44] - Otra posible solución, Array.Copy, tampoco funciona:
Uso de punteros (y contexto no seguro) para solucionarlo:
https://msdn.microsoft.com/en-us/library/aa664784(v=vs.71).aspx
https://msdn.microsoft.com/es-es/library/z50k9bft(v=vs.110).aspx

[45] - Unsafe:
https://msdn.microsoft.com/es-es/library/chfa2zb8.aspx
https://msdn.microsoft.com/en-us/library/aa664784(v=vs.71).aspx

[46] - Leer XML en C#:
http://www.devjoker.com/contenidos/C/29/Como-leer-XML-con-C.aspx

FASE 5 DEL DESARROLLO

[47] - Por qué no puede haber métodos estáticos en una clase que implemente una interfaz:
http://stackoverflow.com/questions/259026/why-doesnt-c-sharp-allow-static-methods-to-implement-an-interface

[48] - Como implementar métodos estáticos en un interface:
http://stackoverflow.com/questions/9415257/how-can-i-implement-static-methods-on-an-interface

[49] - Como instanciar una clase dinámicamente:
https://msdn.microsoft.com/es-es/library/dd264736.aspx

[50] - Como utilizar el tipo dynamic para objetos creados dinámicamente:
https://msdn.microsoft.com/es-es/library/ee461504.aspx
http://social.technet.microsoft.com/wiki/contents/articles/24446.mvp-articles-for-c-and-visual-basic.aspx

[51] - Uso de StringBuilder:
https://msdn.microsoft.com/es-es/library/system.text.stringbuilder%28v=vs.110%29.aspx

FASE 6 DEL DESARROLLO

[52] - Sobre atributos:
https://msdn.microsoft.com/es-es/library/z0w1kczw.aspx

[53] - Atributos comunes:
https://msdn.microsoft.com/es-es/library/z371wyft.aspx

[54] - Crear atributos personalizados:
https://msdn.microsoft.com/es-es/library/sw480ze8.aspx

[55] - Acceder a atributos personalizados y sus parámetros:
https://msdn.microsoft.com/es-es/library/z919e8tw.aspx

[56] - Atributos para XML:
https://msdn.microsoft.com/en-us/library/2baksw0z%28v=vs.110%29.aspx

[57] - Aquí se explica cómo hacer un mecanismo de plug-ins simple:
https://code.msdn.microsoft.com/windowsdesktop/Creating-a-simple-plugin-b6174b62
http://www.codeproject.com/Tips/546639/How-to-create-an-easy-plugin-system-in-Csharp

[58] - Otra forma de hacerlo, más compleja, en español:
http://www.thecoldsun.com/es/content/01-2009/sistema-de-plugins-con-c-parte-i-conceptos

[59] - Teoría de cómo crear un sistema de plug-ins en C#:
http://www.thecoldsun.com/es/content/01-2009/sistema-de-plugins-con-c-parte-i-conceptos

[60] - Otro ejemplo de creación de plug-ins:
http://www.codeproject.com/Articles/14339/Enabling-Your-Application-to-Become-a-Plugin-Host

[61] - Ejemplo de un mecanismo de plug-ins simple:
http://www.drdobbs.com/cpp/implementing-a-plug-in-architecture-in-c/184403942?pgno=3

[62] - Otro ejemplo (con una medalla de bronce ganada):
http://social.technet.microsoft.com/wiki/contents/articles/17501.creating-a-simple-plugin-mechanism.aspx

[63] - Otro ejemplo simple en C++:
https://msdn.microsoft.com/es-es/library/0x82tk9k.aspx

[64] - Más información sobre cómo crear un sistema de plugíns con C#
https://hmadrigal.wordpress.com/?s=+plug-in+with+reflection

[65] - Array.Copy:
https://msdn.microsoft.com/es-es/library/z50k9bft(v=vs.110).aspx

[66] - Ejemplos de programación en paralelo.
https://code.msdn.microsoft.com/Samples-for-Parallel-b4b76364

[67] - Descarga de SharpSerializer:
http://www.sharpserializer.com/en/download/index.html

[68] - Descargar el código de SharpSerializer:
http://www.sharpserializer.com/en/tutorial/index.html

[69] - Descargar el código de Protobuf-net:
https://code.google.com/p/protobuf-net/downloads/detail?name=protobuf-net%20r282.zip

[70] - Manual no oficial de Protobuf-net:
http://www.codeproject.com/Articles/642677/Protobuf-net-the-unofficial-manual

[71] - Serliazación de struct con Protobuf-net:
http://stackoverflow.com/questions/20328590/how-to-use-protobuf-for-struct-serialization