ÍNDICE

1. RESUMEN
2. INTRODUCCIÓN
3. EL PROYECTO
4. SERIALIZADORES
5. DESARROLLO DEL PROYECTO

4.a. FASE 1. Compilación y ejecución dinámica de un objeto (OK)

4.b. FASE 2. Modo de uso de la clase serializadora (CASI OK)

4.c. FASE 3. Escritura del código serializador para una determinada clase

4.d. FASE 4. Refactorización del código de la clase serializador de la fase anterior

4.e. FASE 5. Otro punto de vista

4.f. FASE 6. Aplicación de atributos

4.g. FASE 7. Distintos métodos de serialización aplicables mediante plug-in

1. RESULTADOS OBTENIDOS
2. CONCLUSIONES

###### **Portada**

* + Resumen (de una página)
  + Índice
  + Introducción.
  + Objetivos (incluyendo descripción del problema, estudio de alternativas y metodología empleada).
  + Descripción informática (puede incluir especificación, diseño e implementación).
  + Conclusiones (incluyendo los logros principales alcanzados y posibles trabajos futuros).
  + Bibliografía
  + Apéndices

**0. RESUMEN**

Existen muchos serializadores en el mercado, cada uno con sus ventajas y desventajas, válidos para un determinado tipo 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 serializador versátil que sirva para cualquier tipo de objeto, generando cualquier tipo de código serializado transportable y almacenable en cualquier contexto, ampliable mediante plug-ins y por supuesto, con los mejores resultados de cara al rendimiento.

La principal característica de este serializador será la rapidez. La primera premisa que se ha contemplado a la hora de desarrollar un programa de este tipo ha sido conseguir la mayor velocidad a la hora de serializar y deserializar. Simplemente esto justificaría, si obtenemos un serializador cuya velocidad de proceso sea sensiblemente menor que la de la mayoría de serializadores habituales, el trabajo realizado.

Pero con esta aplicación conseguimos ir un paso más allá. No solo es un serializador que funciona rápido y con cualquier tipo de datos. Posee unas características adicionales que cada una por separado es un punto fuerte en su ámbito, y que al reunirlas en un mismo programa le da un valor extra al mismo. Estas características son:

* 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, el incremento 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 admitir la serialización.
* Permite generar la serialización en múltiples formatos. Nuestra aplicación admite por defecto la serialización en formato XML y CSV, y estamos trabajando en obtener la serialización en formato binario.
* Admite la posibilidad de añadir nuevas funcionalidades a partir de plugins. Permite añadir otros tipos de serialización, nuevos formatos para el código serializado, como por ejemplo HTML, JSON, binario, cualquier otra estructura XML que la definida en el serializado original, etc.
* Como funcionalidad colateral, permitiría codificar y decodificar 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 coversor de tipos de datos o cualquier otra posibilidad que con estas características se nos pudiera ocurrir.

**1. INTRODUCCIÓN**

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

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 almacenada en distintos tipos de elementos. Para ello se utilizan muy a menudo aplicaciones **serializadoras**.

|  |
| --- |
| **Serialización** es el proceso de convertir un objeto en un stream de datos que contiene toda la información del mismo con el fin de almacenarla, compartirla o distribuirla.  **Deserialización** es el proceso inverso, a partir de un stream de datos se recompone el estado original del objeto.  Ambos procesos son complementarios y se usan principalmente para el almacenaje o transporte de información que posteriormente pueda ser reutilizada. |

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. Este tipo de serializador permite la estructuración de la información que serializa, pero a costa de ser sumamente [... la palabreja que usó Agustín ...]. 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. Y esto se multiplica en el momento en que empecemos a serializar objetos más complejos.

En cualquier caso, todos los procesos de serialización tienen unas características comunes:

- funcionan a nivel genérico, permitiendo la serialización de cualquier clase, independientemente de su contenido y del tipo de los miembros que contenga

- contemplan tanto el proceso de serialización como el de deserialización como dos partes complementarias de un mismo proceso.

- generan un resultado intermedio que puede ser almacenado o transportado (compartido).

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 en la que el proceso de serialización sea el más rápido posible. Y la manera óptima de conseguir esta rapidez máxima es que el serializador que realice el proceso esté pensado únicamente para serializar ese tipo determinado de objetos. Esto es lo que tratamos de conseguir: crear el serializador perfecto para cada tipo de objeto. Si una aplicación está pensada para serializar sólo un determinado tipo de objeto, podrá hacerlo de la manera más rápida posible. Además, cuando haya que serializar más objetos de ese mismo tipo, trabajará igual de eficientemente para todos ellos.

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 por ejemplo con 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, 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. EL 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. Lo que hemos hecho en este proyecto es construir 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 serializa y deserializa exactamente el contenido de esos miembros.

Con este proyecto tratamos de dar respuesta a la necesidad 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 comparativo con la utilización de este serializador.

Esta aplicación es por tanto 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 sea un factor 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 el código serializado con distintos formatos, aparte del XML o CSV que se 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 anteriormente citadas.

También 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 plugin 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.

Resumiendo, lo que hace básicamente esta aplicación es **generar 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, a través de la ejecución de sus métodos **encode** y **decode** (serialize y deserialize), 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 aplicación generada hemos 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 la tabla comparativa de tiempos de ejecución en un benchmark al final de este capítulo.

[tabla comparativa, generar un proyecto que realice la medida de todos los objetos serializados y deserializados con los serializadores existentes, y luego con el nuestro]

- Generar 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 plugin).

En definitiva, una herramienta útil y versátil que puede ser utilizada en diferentes contextos, y que puede ser fácilmente extendida para adaptarla a las necesidades del programador.

**3. OTROS SERIALIZADORES (SEMEJANZAS Y DIFERENCIAS)**

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, existen cuatro serializadores principales integrados en el framework:

- **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.

- **CustomSerializer**

Permite que el programador realice de manera manual la serialización de ciertos miembros del objeto a serializar. Sirve para dar una oportunidad de serializar aquellos miembros que por su naturaleza en principio quedan fuera de las posibilidades automáticas de serialización de los serializadores anteriores. Para realizar la serialización, la clase tiene que implementar la interfaz *ISerializable*, lo que obliga a programar el comportamiento de la serialización con los miembros que se elijan, dejando la responsabilidad de indicar cómo se producirá esta serialización en el programador.

Existen además otras herramientas externas a .Net que sirven para el mismo propósito, cada una con sus características. Destacan herramientas opensource como **SharSerializer** o **NetSerializer** (el más parecido a nuestro proyecto), o el proyecto de Google **Protobuf-net**.

En los diferentes 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.

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 compartidas** | **Contratos Colectivos** |
| **Binario** | BinaryFormatter |  |
| **XML** | NetDataContractSerializer | XmlSerializer  DataContractSerializer |
| **Json** |  | DataContractJsonSerializer |
| **SOAP** | SoapFormatter |  |

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.

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.

Comparativa de formatos de serialización en los serializadores analizados en relación a este proyecto:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **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 |  |  |  |  |  |
| HiperSerializer | URJC | X | X | X | X | X |

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

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Nombre** | **Propiedades** | **Campos** | **Arrays** | **Arrays**  **multidim**  **y nested** | **Lists** | **Generic Lists** | **Herencia** | **Struct** | **Enum** | **Otros** |
| XMLSerializer | X | X | X | - | X | - |  |  |  |  |
| DataContract  Serializer | X | X | X | - | X |  |  |  | X |  |
| BinaryFormatter | X | X | X |  |  |  | X | X |  |  |
| SOAPFormatter | X | X | X |  | X | - | X | X |  | X |
| SharpSerializer | X | - | X | X | X | X | X |  |  |  |
| NetSerializer |  |  | X |  |  |  |  | X |  |  |
| Protobuf-net |  |  | X |  |  |  |  |  |  |  |
| HiperSerializer | X | X | X | X | X | X | X | X | X | X |

REFERENCIAS

Información sobre XML Serializer

<http://msdn.microsoft.com/es-es/library/bb552764%28v=vs.90%29.aspx>

Ejemplos de XMLSerializer:

<https://msdn.microsoft.com/en-us/library/58a18dwa%28v=vs.110%29.aspx>

Comparativa de la ejecución de serialización de algunos serializadores:

<http://mono.servicestack.net/benchmarks/NorthwindDatabaseRowsSerialization.100000-times.2010-08-17.html>

An overview of .Net serializers:

<http://blogs.msdn.com/b/youssefm/archive/2009/04/15/an-overview-of-net-serializers.aspx>

[Características de DataContractSerializer:](http://blogs.msdn.com/b/youssefm/archive/2009/04/15/an-overview-of-net-serializers.aspx)

<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>

Tipos de datos serializables con DataContractSerializer:

<http://msdn.microsoft.com/en-us/library/ms731923.aspx>

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>

Comparativa de serialización con ProtoBuf:

<http://code.google.com/p/protobuf-net/wiki/Performance>

DataContractSerializer

<https://msdn.microsoft.com/es-es/library/ms731073%28v=vs.110%29.aspx>

Tipos de datos que admite serializar DataContractSerializer

<https://msdn.microsoft.com/es-es/library/ms731923(v=vs.110).aspx>

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>

Listado de los tipos de elementos con los que podemos encontrarnos dentro de un objeto:

<https://msdn.microsoft.com/es-es/library/ybcx56wz.aspx>

**4. EL DESARROLLO DEL PROYECTO**

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.

FASE 1. Compilación y ejecución dinámica de un objeto.

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 generada 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 serializador para una determinada clase.

Se generar el código para el serializador de una determinada clase, almacenándolo en una variable string. A continuación se compila y se invoca la ejecución de la serialización de un objeto de esa clase. Se establece la base de cómo 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 cómo 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 cada tipos de los miembros 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, una mejora consiste en la toma en consideración de atributos a la hora de serializar. Para ello 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.

FASE 7. Distintos métodos de serialización aplicables mediante plugins.

En esta fase se mejora la aplicación para conseguir que se pueda ampliar su funcionalidad con la inserción de distintos plugins cuya funcionalidad sea la de modificar la manera en que se realiza tanto la codificación cómo la descodificación de los datos, adaptándola al formato que el plugín indique.

[Continuará]

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 de 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 según su uso.
* Permitir la gestión de distintos plugins para la generación del serializador con unas u otras características (tipo y formato del código serializado).

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, en función del tipo de datos para el cual se definirá el serializador. Se procede a compilar de manera dinámica el código, y a invocar y mantener en memoria un objeto de la clase compilada, que es con al que se va a trabajar.

Como primera aproximación compilamos y ejecutamos la típica aplicación “Hola mundo” de manera dinámica. Se trata de introducir el código de una 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 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 clases serializadas. 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, 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 en una variable, el siguiente paso es ejecutar la compilación en tiempo de ejecución de ese código, para generar la clase que describe. Para compilarlo en tiempo de ejecución se utiliza el mecanismo que .Net tiene para estas tareas, **CodeDOM** (Code Document Object Model). Este mecanismo requiere la parametrización del **grafo CodeDOM**, que vincula todos los elementos que genera el uso de este mecanismo.

El espacio de nombres CodeDOM se usa para crear, compilar y ejecutar programas en tiempo de ejecución, creando una representación del código fuente en memoria o en un archivo ejecutable (se indica a través del parámetro **GenerateInMemory** de la clase **CompilerParameters** que queremos dejar el objeto instanciado en memoria). 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#.

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 (con valores que se usan para parametrizar como se ejecutarán los métodos asociados al tipo de datos) y se muestra el resultado de la ejecución del método.

En nuestra aplicación utilizaremos este mismo mecanismo para generar el objeto que realizará la codificación y decodificación de objetos del tipo dado a partir de la compilación en tiempo real del código que genera el serializador, para invocar sobre dicho objeto los métodos encode y decode

Éste es el código que produce la compilación y ejecución en vivo de 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.ReferencedAssemblies.Add("System.Xml.dll");  //cp.ReferencedAssemblies.Add(tipo.Assembly.Location);  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 **CodeComProvider.CreateProvider**. Recibe cómo parámetro el lenguaje para el que se va a crear el compilador, en este caso “CSharp”.
* Se definen los parámetros para la compilación con el uso de una instancia de la clase **CompilerParameters**. Aquí se puede definir los distintos parámetros que se usarán en la compilación, tales como la referencia a los ensamblados que se necesitarán, o si la compilación se llevará a cabo generando un fichero externo o se mantendrá en memoria, 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 el objetivo principal del proyecto) se realiza la invocación de **CompileAssemblyFromSource** sobre el objeto CodeComProvider creado previamente, pasándole el texto del código generado y los parámetros definidos, 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 objeto 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). 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.

A continuación se muestra el código que compila y devuelve una instancia de la clase compilada a partir del código en un string.

|  |
| --- |
| // Ejecutar la compilación on tye fly  // Instanciar un objeto de la clase compilada y devolverlo  private Object compile(Type tipo) // Pasar el tipo completo  {  try  {  ICodeCompiler loCompiler = new CSharpCodeProvider().CreateCompiler();  CompilerParameters loParameters = new CompilerParameters();  loParameters.ReferencedAssemblies.Add("System.dll");  loParameters.ReferencedAssemblies.Add("System.Xml.dll");  loParameters.ReferencedAssemblies.Add(tipo.Assembly.Location);  loParameters.GenerateInMemory = true;  CompilerResults loCompiled = loCompiler.CompileAssemblyFromSource(loParameters, strCodigo);  // Se comprueba si hubo errores  if (!loCompiled.Errors.HasErrors)  {  Assembly loAssembly = loCompiled.CompiledAssembly;  Object loObject = null;  loObject = loAssembly.CreateInstance("Serializer." + tipo.Name + "Codec");  return loObject;  }  else  {  return null;  }  }  catch (Exception e)  {  return null;  }  } |

REFERENCIAS PARA LA FASE 1

Compiling and running code at runtime

<http://simeonpilgrim.com/blog/2007/12/04/compiling-and-running-code-at-runtime/>

Generación y compilación dinámica de código fuente

<https://msdn.microsoft.com/es-es/library/ms404245%28v=vs.110%29.aspx>

Compiling with CodeDOM

<http://www.codeproject.com/Articles/7119/Compiling-with-CodeDom>

Microsoft .Net CodeDOM Technology

<http://www.codeguru.com/vb/gen/article.php/c19573/Microsoft-NET-CodeDom-Technology.htm>

[Compilando en tiempo de ejecución](http://www.codeguru.com/vb/gen/article.php/c19573/Microsoft-NET-CodeDom-Technology.htm)

<http://foro.elhacker.net/net/compilando_en_tiempo_de_ejecucion_iquestmake_builder_c-t313428.0.html;msg1553157>

Generación y compilación dinámicas de código fuente

[http://msdn.microsoft.com/es-es/library/650ax5cx(v=vs.110).aspx](http://msdn.microsoft.com/es-es/library/650ax5cx%28v=vs.110%29.aspx)

Referencias sobre como hacer para compilar una clase con CodeDOM:  
<http://www.codeproject.com/Articles/7119/Compiling-with-CodeDom>

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

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 alternativa a la hora de definir cómo se invocará dicha clase, cómo recibirá los parámetros de entrada, y cómo generará la salida.

En esta fase se trata de averiguar cuál es la estructura de clase más eficiente para invocar la serialización y deserialización de clases, dentro de las distintas posibilidades de plantear esta clase que pueden existir. 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), con el uso de métodos de extensión e incluso valorando si la clase serializadora tendrá sus principales métodos estáticos o de instancia.

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. Esta clase posee los métodos 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 el mismo método, método 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 del 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, siendo éstas las definiciones correspondientes a cada método:

Uso A:

public string encode(ClaseBasica01 aux);

public ClaseBasica01 decode(string s);

Uso B:

public string encode(ref ClaseBasica01 aux);

public void decode(ref ClaseBasica01 aux, string s);

Para el 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 en la clase o tipo indicado se utilicen 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. Cómo 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 fuera) 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.

Para el caso D, únicamente se definen los métodos encode y decode como estáticos en la clase. Esto supondrá que cada vez que se desee realizar la serialización de un objeto de determinado tipo, utilizando su clase serializadora particular, se usará la misma entidad, asegurándonos de que no creamos un nuevo objeto de ese tipo para serializar el mismo tipo de objetos. Esto será especialmente útil cuando tratemos de serializar un gran número de objetos del mismo tipo, ya que con una única instancia del serializador para el mismo tipo, podremos realizar todas las operaciones.

A continuación se describen las clases que se usarán para realizar las pruebas. Se trata de que sean objetos de lo más variado 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; }

}

- Clase02Metodos. Clase normal, con dos métodos que manejan atributos privados. No habrá que serializar nada. En principio, la serialización de miembros no públicos queda fuera del ámbito de este proyecto. Hemos decidido que esto sea así, ya que la principal premisa de este proyecto es la de conseguir realizar la serialización de objetos de la manera más rápida y moldeable posible, y si hay que contemplar el menejo de miembros no públicos habría que tomar otras estrategias que incluyeran la gestión de los miembros de los objetos instaciados por reflection, lo que aumentaría considerablemente el tiempo de ejecución de los procesos de serialización y deserialización.

public class Clase02Metodos

{

private int var1;

private String var2;

public int metodo1()

{

this.var1 = 1;

return this.var1;

}

public string metodo2()

{

this.var2 = "2";

return this.var2;

}

}

- Clase03Array. Clase normal, con una propiedades de cada tipo de array: un array unidimensional de tipo entero, otro unidimensional de tipo string, uno bidimensional de tipo entero, uno tridimensional de tipo entero y un array de arrays de tipo entero.

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; }

}

- 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 propiedades más de tipo estático, además de una propiedad con característica protected (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

}

- Clase08List. Clase con cuatro listas, una de tipo entero, otra de tipo clase (una clase definida en el propio archivo) y dos dictionarys, uno de tipo primitivo y otro de tipo clase.

La prueba realizada para averiguar la estructura más adecuada consiste en repetir hasta 10.000 veces la serialización y deserialización de cada una de las clases, registrando el tiempo que lleva cada serialización y deserialización por separado. Con esos datos se sacarán las conclusiones adecuadas.

Tabla de ejecución del método encode. Tiempo en milisegundos:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 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 |

Tabla de ejecución del método decode. 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 |

Con estos datos queda claro que la mejor forma para invocar al serializador sería utilizar la característica de los métodos de extensión, para aplicar la codificación y decodificación directamente a la clase que se quiera serializar.

REFERENCIAS PARA LA FASE 2

La base de los métodos de extensión.

<http://msdn.microsoft.com/es-es/library/bb383977.aspx>

Cómo implementar e invocar un método de extensión personalizado.

<http://msdn.microsoft.com/es-es/library/bb311042.aspx>

FASE 3

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

Entre los objetivos principales del proyecto está la generación dinámica de una clase cuya funcionalidad sea serializar y deserializar los datos de determinados 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. El código que se va creando actúa mediante reflexión sobre el tipo objetivo, creando los métodos de codificación (encode) y decodificación (decode) para ese tipo. En un primer momento este código se guardará en una variable de tipo String. El objetivo final es convertir esta variable en un array de bytes susceptible de ser enviado por red para distribuir esta clase.

Uno de los principales problemas es 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 (<http://msdn.microsoft.com/en-us/library/bfft1t3c.aspx>). Por eso primero descartamos encontrarnos 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.

Ejemplos:

<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>

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 la invocación 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 y 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).

Diferencias entre campos y propiedades:

<http://www.mundonet.es/campos-propiedades-ejercicio-1.html?Pg=Entrega8.htm>

Utilizar propiedades:

<https://msdn.microsoft.com/es-es/library/w86s7x04.aspx>

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, opté por utilizar el método GetMembers, que 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. …

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. Queda como pendiente.

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

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

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

Para recorrer los elementos del array, usamos esta rutina:

TODO: También hay que identificar cuando estemos ante Listas. Quizás haya una forma de identificar ambos tipos de elementos a la vez.

La hay, es ésta:

IList list = propertyInfo.GetValue(obj, null) as IList;

Queda por confirmar si se pueden tratar igual un array de una lista, sobre todo a nivel de índices y contenido de cada elemento.

FORMATO DE LA SALIDA DEL SERIALIZADOR (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 explicar’a c’pomo se ha ideado este mecanismo, aunque en esta primera aproximación del código ya contempla y encapsula los métodos que servirá para este cometido.

El sistema tiene varios formatos de salida disponible para el código serializado. El formato por defecto para la serialización es XML. Si no se indica nada, éste será el formato que se usará. Pero también se puede generar la salida en formato de datos separados por comas, e incluso en formato binario.

El formato XML que se usará 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>

Se puede incorporar el DTD directamente en esta definición, si vemos que es fácil hacerlo, y no tiene que ser dinámico. (TODO)

PSEUDOCÓDIGO

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

getSerializador(tipo) // Comprobar si hay que generar el serializador o ya existe en “serializadores”, una tabla hash con todos los serializadores generados hasta ahora

si(existe serializadores(tipo))

obj = serializadores(tipo)

sino

generateSerializer()

obj = compile(nombre)

registerSerializer(nombre, obj)

finsi

registerSerializer(nombre, obj)

guardar en hashTable (nombre, obj)

generateSerializer()

getCabecera()

generateEncodeAndDecodeMethods()

generateAuxiliarMethods()

getCierre()

generateEncodeAndDecodeMethods()

encode =

public void encode(object objeto, ref String str)

{

tipo.Name obj = (tipo.Name)objeto;

decode =

public object decode(String str, ref object obj)

{

encode +=

string texto = "";

texto += " + abrir("serializacion");";

texto += " + abrir("accesibilidad");";

texto += " + getAccesibilidad(tipo);";

texto += " + cerrar("accesibilidad");";

bool pintar = false;

modificador = getModificador(tipo, ref pintar);

si(pintar)

texto += " + abrir("modificador");";

texto += " + modificador;

texto += " + cerrar("modificador");";

finsi

texto += " + abrir("tipoDeObjeto");";

texto += " + getTipoDeObjeto(tipo);";

texto += " + cerrar("tipoDeObjeto");";

MemberInfo[] miembros = tipo.GetMembers();

encode +=

texto += " + abrir("elementos");";

decode +=

texto += "var elementos = get(str, ""elementos"");"; // Codificar get(txt, elem)

Para cada miembro en miembros

encode +=

texto += " + abrir("elemento");";

texto += " + abrir("nombre");";

texto += " + mostrarValor(nombre);";

texto += " + cerrar("nombre");";

texto += " + abrir("tipo");";

texto += " + mostrarValor(t.FullName);";

texto += " + cerrar("tipo");";

texto += " + abrir("valor");";

decode +=

texto += var elemento = elementos.get("elemento", " + nombre + ");";

texto += elemento = elementos.get("elemento", " + nombre + ");";

si(es tipo Primitivo o String)

encode +=

texto += obj." + nombre + ".ToString();";

decode +=

texto += "obj." + nombre + " = elemento.valor;";

sino si(es tipo Array)

encode +=

texto += " + abrir("count");";

texto += obj." + nombre + ".Count;";

texto += " + cerrar("count");";

texto += " + abrir("tipoDeElementos");";

texto += " + t.GetElementType().FullName;";

texto += " + cerrar("tipoDeElementos");";

texto += " + abrir("rank");";

texto += " + t.GetArrayRank();";

texto += " + cerrar("rank");";

texto += " + abrir("datosDeLosRangos");";

decode +=

texto += var count = elemento.Count;";

texto += var tipoDeElementos = " + t.GetElementType().FullName;";

texto += var rank = " + t.GetArrayRank();

para i=0 hasta t.GetArrayRank() paso 1

{

texto += " + abrir("datosDeRango");";

texto += " + abrir("longitud");";

texto += obj." + nombre + ".GetLength(" + i + ");";

texto += " + cerrar("longitud");";

texto += " + abrir("valorMenor");";

texto += obj." + nombre + ".GetLowerBound(" + i + ");";

texto += " + cerrar("longitud");";

texto += " + cerrar("datosDeRango");";

}

texto += " + cerrar("datosDeLosRangos");";

texto += " + abrir("valores");";

foreach (object elemento in obj." + nombre + ")";

{

texto += " + abrir("cadaValor");";

texto += " + abrir("valor");"; // Ver si es de otro tipo

texto += elemento.ToString();";

texto += " + cerrar("valor");";

texto += """ + cerrar("cadaValor");";

}

sino si(es tipo List)

…

sino si(es tipo Dictionary)

…

sino si(es tipo Objeto externo)

// Generar serializador o capturarlo de

si(existe serializadores(tipo))

getSerializador(tipo)

sino

crearSerializador(tipo)

finsi

encode +=

texto += " + getSerializador(tipo);";

finsi

finpara

encode +=

texto += " + cerrar("elementos");";

texto += " + cerrar("serializacion");";

return texto;

decode +=

return obj;

fin generateEncodeAndDecodeMethods

PROBLEMAS QUE ME HE ENCONTRADO

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 índicees 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. Cómo digo, 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 poco eficiente, porque obliga a recorrer uno tras otro todos los elementos del array, y 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 rango (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 recorremos 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, pero de un lugar de en un …

Resumiendo el problema de los arrays:

- A la hora de codificar un array, simplemente recorro todos sus elementos con un foreach, y se guarda cada valor. Sin problemas. Pero también guardo datos que necesitaré 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, probé con varias alterativas que me permitieran rellenar correcta y rápidamente el array. En un primer momento, contemplé ir rellenando un array auxiliar con la misma estructura que el campo que quiero deserializar. Aquí tengo 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, he 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 funcionamiento de este algoritmo, con el 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 cómo se generan, pero es trivial adivinar 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 mayor de la dimensión).

Para que no haya solapamientos con los nombres, como se hace 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 cómo 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 es trivial 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:

// if(" + nombre + " == null) " + nombre + " = Array.CreateInstance(" + tipoElemento + ", " + longitudes + ");";

|  |
| --- |
| 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 no admite la serialización y deserialización de arrays de arrays.

Otro problema nos lo encontrábamos cuando alguno de cualquiera 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>

FASE 4

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

En primer lugar se reorganiza cómo funciona el generador, y cómo 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 par de métodos auxiliares que se invocarán para cada miembro de la clase susceptible de ser serializada.

A continuación se detallan estos métodos:

* private void getCodigoByMembers(System.Reflection.MemberInfo[], Object)

El deserializador sigue las pautas marcadas por el serializador para recuperar todos los datos serializados, encapsulándolos en los elementos pertinentes de un objeto que se recibe cómo parámetro por referencia. Esto significa que se irá rellenando el objeto con cada valor adecuado para cada atributo o campo.

Una de las principales dificultades a la hora de recuperar los datos es la recuperación de arrays y de listas. 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. TODO: rank > 1.

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.

Otra forma de crear un array con reflection es ésta:

<https://msdn.microsoft.com/en-us/library/hc2e4ft8(v=vs.110).aspx>

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 ya no se usarán más).

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);

}

foreach (int ival in intArray3D)

{

Console.WriteLine(ival);

}

Ejemplo:

Array names = Array.CreateInstance(typeof(String), 2, 4);

names.SetValue("Rosy", 0, 0);

names.SetValue("Amy", 0, 1);

names.SetValue("Peter", 0, 2);

names.SetValue("Albert", 0, 3);

names.SetValue("Mel", 1, 0);

names.SetValue("Mongee", 1, 1);

names.SetValue("Luma", 1, 2);

names.SetValue("Lara", 1, 3);

int items1 = names.GetLength(0);

int items2 = names.GetLength(1);

for (int i = 0; i < items1; i++)

for (int j = 0; j < items2; j++)

Console.WriteLine(i.ToString() + "," + j.ToString() + ": " + names.GetValue(i, j));

Console.WriteLine(stringArray.GetLength(0).ToString());

Console.WriteLine(stringArray.GetLowerBound(0).ToString());

Console.WriteLine(stringArray.GetUpperBound(0).ToString());

A continuación se detallan los tipos de matrices que se pueden encontrar en C#, según cómo se posicionan sus elementos:

* Single-dimensional arrays
* Multidimensional arrays or rectangular arrays
* Jagged arrays
* Mixed arrays

(<http://www.c-sharpcorner.com/UploadFile/mahesh/WorkingWithArrays11232005064036AM/WorkingWithArrays.aspx>)

Es importante saber los tipos de matrices que podemos utilizar en C#:

C# admite matrices de una dimensión, matrices multidimensionales (matrices rectangulares) y matrices de matrices (matrices escalonadas). El siguiente ejemplo muestra cómo 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 muestra arriba) no crea realmente las matrices. En C#, las matrices son objetos (se explica más adelante en este tutorial) 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];  
}

También se pueden utilizar matrices más grandes. Por ejemplo, se puede utilizar 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;

<https://msdn.microsoft.com/es-es/library/aa288453%28v=vs.71%29.aspx>

Para control de errores, por ejemplo que no se pueda asignar un valor al tipo que corresponde la variable o campo, se puede usar el método IsAssignableFrom cómo se explica aquí:

<http://www.iteramos.com/pregunta/20340/casting-de-una-variable-utilizando-una-variable-de-tipo>

Como se indicaba arriba, con Array.CreateInstance tenemos la posibilidad de definir cualquier tipo de matriz con cualquier estructura.

PROBLEMA SERIALIZANDO DICTIONARY

Los serializadores habituales no pueden serializar Dictionarys.

Un XMLSerializador no puede hacerlo de manera natural:

<https://msdn.microsoft.com/en-us/library/ms950721.aspx>

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.

FORMA DE PROCESAR POR IGUAL LAS PROPIEDADES Y LOS CAMPOS

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.

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>

PROCESAR GENERIC (Dictionary<,>)

<https://msdn.microsoft.com/en-us/library/b8ytshk6(v=vs.110).aspx>

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).

DECODIFICACIÓN DE ARRAYS, LISTS, COLLECTIONS

Para montar de una maner más dinámica los métodos de codificación y decodificación, contamos en el Generador con una función llamada **getValue**, que recibe cómo 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 cómo arrays, listas, diccionarios, no se sabe a priori datos necesarios para su inicialización, cómo 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.

Por tanto, el organigrama de esta función recursiva que se encarga de generar el código que codificará por un lado y descodificará por otro la clase procesada es éste:

FUNCIÓN getValue (TIPO t, CADENA nombre, CADENA nombreAuxiliar)

SI t ES PRIMITIVO O CADENA ENTONCES

globalCodigoEncode += “texto.Append(nombre)”

globalCodigoDecode += “nombre = texto.Extract()”

SINO SI t ES ARRAY ENTONCES

tipoElemento = t.GetElementType.FullName

globalCodigoEncode += “texto.Append(nombre.Length)”

globalCodigoDecode += “int length-nombre = texto.Extract() como int”

globalCodigoEncode += “texto.Append(t.GetElementType.FullName)”

globalCodigoDecode += “tipo = texto.Extract() como Type”

globalCodigoEncode += “texto.Append(t.GetRank())”

globalCodigoDecode += “ranto = texto.Extract() como int”

PARA i=0 HASTA t.GetRank() HACER

globalCodigoEncode += “texto.Append(nombre.GetLength(i))”

globalCodigoDecode += “int auxLength-i = texto.Extract() como int”

globalCodigoEncode += “texto.Append(nombre.GetLowerBound(i))”

globalCodigoDecode += “int auxLowerBound-i = texto.Extract() como int”

globalCodigoEncode += “texto.Append(nombre.GetUpperBound(i))”

globalCodigoDecode += “int auxUpperBound-i = texto.Extract() como int”

FIN PARA

indices = “[“

longitudes = “[“

PARA i=0 HASTA t.GetRank() HACER

globalCodigoDecode += “for(int indice-i = auxLowerBound-i; indice-i < auxUpperBound-i; indice-i++){“

indices += indice-i

longitudes += indice-i

FIN PARA

indices += “]“

longitudes = “]”

globalCodigoDecode += “if(nombre==null) nombre = new tipoElemento

globalCodigoEncode += “foreach(tipoElemento elementoAux in nombre){“

getValue(tipoElemento, elementoAux, nombre)

globalCodigoEncode += “}“

PARA i=0 HASTA t.GetRank() HACER

globalCodigoDecode += “}”

FIN PARA

SINO SI t ES LIST

…

SINO SI t ES DICTIONARY

…

SINO // Se entiende que t es un objeto a procesar

globalCodigoEncode += “texto.Append(t-Codec.encode(nombre))”

globalCodigoDecode += “t objAux = new t() // Si t es propiedad no se puede pasar como parámetro REF”

globalCodigoDecode += “t-Codec.decode(ref elementos, ref objAux)”

globalCodigoDecode += “nombre = objAux”

clasesADefinir.Add(t)

FIN SI

<https://books.google.es/books?id=_Y0rWd-Q2xkC&pg=PA239&lpg=PA239#v=onepage&q&f=false>

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 cómo 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

obj.campoArray = aux as int[ ]; // siendo int[ ] el tipo de dato de campoArray

DECODIFICACIÓN DE 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. Cómo lo que queremos es exactamente la acción de una cola, utilizamos ésta.

Sacar elementos de array en C# no es sencillo:

<http://stackoverflow.com/questions/455237/pop-off-array-in-c-sharp>

Instanciar Array genérico con uno o más rangos:

<https://msdn.microsoft.com/es-es/library/zb3cfh7k%28v=vs.110%29.aspx>

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.

Uso de XMLReader para recorrer secuencialmente un documento XML:

<https://msdn.microsoft.com/es-es/library/system.xml.xmlreader.readouterxml(v=vs.110).aspx>

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, pero [...]

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.

Converting string to type c#:

<http://stackoverflow.com/questions/11107536/convert-string-to-type-c-sharp>

Convertir una cadena en número (o cualquier otro tipo de dato básico):

<https://msdn.microsoft.com/es-es/library/bb397679.aspx>

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 cómo 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

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.

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)

Utilizar foreach con matrices

<https://msdn.microsoft.com/es-es/library/2h3zzhdw.aspx>

<http://www.dotnetperls.com/array-foreach>

Intento de modificación de un array dentro de un foreach (fallido)

<http://stackoverflow.com/questions/5449347/c-sharp-array-foreach-alternative-that-replaces-elements>

Cómo no se puede modificar los elementos de un array: <http://www.dotnetperls.com/array-foreach>

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>

Unsafe:

<https://msdn.microsoft.com/es-es/library/chfa2zb8.aspx>

<https://msdn.microsoft.com/en-us/library/aa664784(v=vs.71).aspx>

XML

El formato por defecto de la codificación de nuestro programa es el XML. Se genera un archivo XML bien formado que sigue el siguiente DTD: [TODO].

A la hora de decodificar, simplemente hay 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.

ENLACES DE INTERÉS:

<http://www.devjoker.com/contenidos/C/29/Como-leer-XML-con-C.aspx>

FASE 5

Ejecutar el objeto serializer creado dinámicamente.

Una vez que hemos creado la clase serializadora al vuelo, se devuelve 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 se deben indicar los métodos que tendrá este objeto, ya que a continuación los vamos a ejecutar, y en tiempo de compilación no existen.

Una alterativa sería 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 (raramente) y estos dos métodos para ser eficientes tienen que ser estáticos.

Aquí se explica cómo se puede evitar esta restricción, aunque no hay muchas garantías de que funcione:

<http://stackoverflow.com/questions/9415257/how-can-i-implement-static-methods-on-an-interface>

Otra posibilidad es utilizar clases de tipo **dynamic**.

Aquí se explica qué es esto:

<https://msdn.microsoft.com/es-es/library/ee461504.aspx>

Cómo instanciar una clase dinámicamente:

<https://msdn.microsoft.com/es-es/library/dd264736.aspx>

Hay más literatura al respecto:

<http://social.technet.microsoft.com/wiki/contents/articles/24446.mvp-articles-for-c-and-visual-basic.aspx>

Lo primero es que se necesita 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).

Aunque en tiempo de compilación no se produce ningún error usando una variable de tipo dynamic par-a capturar la clase serializador 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 es ejecutar el método a través de reflection, con GetType().InvokeMethod().

No sé si es una alternativa válida.

CAMBIO DE STRING POR STRINGBUILDER

El código de la clase serializadora, hasta ahora, 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 que se desarrolle en la menor cantidad de tiempo 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 con la concatenación se produce esta situación. 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 según la necesidad de la cadena. Así, es mucho más efectivo en estos casos utilizar varibles de tipo StringBuilder.

<https://msdn.microsoft.com/es-es/library/system.text.stringbuilder%28v=vs.110%29.aspx>

FASE 6

Objetivo: Definir cómo se indicarán las clases o partes de las clases susceptibles de ser serializadas.

Los principales datos que necesitamos para realizar esta tarea son el TIPO de objeto que vayamos a invocar y el lenguaje en el que está escrito el código fuente para generar el objeto.

Para identificar las clases y las partes de la misma susceptibles de ser serializadas, se ha ideado una etiqueta que se incorporará al código fuente de la clase. Aquellos atributos que tengan que ser serializados se marcan con esa etiqueta.

PROBLEMAS

El principal problema es el tiempo. Se trata de que el proceso de serialización y deserialización sea lo más rápido posible (de alto rendimiento). Para solucionarlo hemos realizado 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.

Ejemplos de programación en paralelo.

<https://code.msdn.microsoft.com/Samples-for-Parallel-b4b76364>

ESTRUCTURA DE LA APLICACIÓN

A continuación se describe la composición de la aplicación, sobre todo de cara a su modularización.

GENERADOR.CS

Clase principal, 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
* System.CodeDom.Compiler

Consta de varios atributos 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.

Se divide el trabajo de este método en distintos métodos que realizan parte de la serialización:

* getCabecera()
* generateEncodeAndDecodeMethod()
* generateAuxiliarMethods()

El método compile() se encarga de compilar el código creado en el método anterior y devuelve una instancia del objeto creado. Este objeto será el que se devuelva al cliente para que pueda trabajar con él.

La estructura del programa que se genera es la siguiente:

namespace Serializer

{

public static class *nombreDeLaClase*Codec{

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);

}

}

PLUGINS PARA DISTINTOS TIPOS DE CODIFICACIÓN (XML, CSV, JSON, BINARY, ETC.)

Para favorecer que la salida del encode se pueda generar en diferentes formatos utilizo un mecanismo de plug-ins. Consiste en encapsular las diferentes maneras de ejecutar la codificación dentro de ensamblados diferentes, y se cargará y ejecutará una u otra según la codificación que se desee.

Para ello, contamos inicialmente con un inteface que define los métodos necesarios para realizar la codificación. Posteriormente se puede sustituir esta dependencia del inteface por genéricos.

Aquí se explica cómo hacerlo:

<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>

Otra forma de hacerlo, más compleja, en español:

<http://www.thecoldsun.com/es/content/01-2009/sistema-de-plugins-con-c-parte-i-conceptos>

CÓMO CONVERTIR UN OBJETO A STRING (PARA COMPARAR un objeto con su representación como cadena. Se podría suponer que si la representación como cadena del objeto es igual al propio objeto, es trata de un valor mínimo o básico).

<http://stackoverflow.com/questions/2961656/generic-tryparse>

EL COSTE DE USAR REFLECTION

<http://stackoverflow.com/questions/25458/how-costly-is-net-reflection>

<http://www.nuget.org/packages/FastMember> (PLUGIN QUE ACELERA EL ACCESO)

CÓMO OBTENER LAS PROPIEDADES Y LOS CAMPOS JUNTOS

<http://stackoverflow.com/questions/12680341/how-to-get-both-fields-and-properties-in-single-call-via-reflection>

CÓMO IDENTIFICAR LOS TIPOS BÁSICOS

A la hora de obtener los valores de cada propiedad o campo de un objeto hay que saber cómo identificar los tipos básicos, para saber hasta dónde hay que bucear en el contenido interno de cada elemento.

Se está utilizando una estructura recursiva (ver si es mejor esto o una estructura no recursiva) para identificar, dentro del tipo de una propiedad o campo del objeto a serializar, sus elementos internos. Así, si nos encontramos ante un array, identificaremos el tipo de sus elementos, y si no son tipos básicos damos otra vuelta al bucle localizando el tipo y los valores del elemento.

En cambio, si el tipo es básico, simplemente recogemos su valor

Así, tenemos que tener claro que los valores básicos de cada elemento tiene que terminar siendo uno de estos tipos básicos:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [Byte](https://msdn.microsoft.com/es-es/library/system.byte(v=vs.90).aspx) | Entero sin signo | 8 | 0 a 255 |  |
| sbyte | [SByte](https://msdn.microsoft.com/es-es/library/system.sbyte(v=vs.90).aspx) | Entero con signo | 8 | -128 a 127 |
| int | [Int32](https://msdn.microsoft.com/es-es/library/system.int32(v=vs.90).aspx) | Entero con signo | 32 | -2.147.483.648 a 2.147.483.647 |
| uint | [UInt32](https://msdn.microsoft.com/es-es/library/system.uint32(v=vs.90).aspx) | Entero sin signo | 32 | 0 a 4294967295 |
| short | [Int16](https://msdn.microsoft.com/es-es/library/system.int16(v=vs.90).aspx) | Entero con signo | 16 | -32.768 a 32.767 |
| ushort | [UInt16](https://msdn.microsoft.com/es-es/library/system.uint16(v=vs.90).aspx) | Entero sin signo | 16 | 0 a 65535 |
| long | [Int64](https://msdn.microsoft.com/es-es/library/system.int64(v=vs.90).aspx) | Entero con signo | 64 | -922337203685477508 a 922337203685477507 |
| ulong | [UInt64](https://msdn.microsoft.com/es-es/library/system.uint64(v=vs.90).aspx) | Entero sin signo | 64 | 0 a 18446744073709551615 |
| float | [Single](https://msdn.microsoft.com/es-es/library/system.single(v=vs.90).aspx) | Tipo de punto flotante de precisión simple | 32 | -3,402823e38 a 3,402823e38 |
| double | [Double](https://msdn.microsoft.com/es-es/library/system.double(v=vs.90).aspx) | Tipo de punto flotante de precisión doble | 64 | -1,79769313486232e308 a 1,79769313486232e308 |
| char | [Char](https://msdn.microsoft.com/es-es/library/system.char(v=vs.90).aspx) | Un carácter Unicode | 16 | Símbolos Unicode utilizados en el texto |
| bool | [Boolean](https://msdn.microsoft.com/es-es/library/system.boolean(v=vs.90).aspx) | Tipo Boolean lógico | 8 | True o false |
| object | [Object](https://msdn.microsoft.com/es-es/library/system.object(v=vs.90).aspx) | Tipo base de todos los otros tipos |  |  |
| string | [String](https://msdn.microsoft.com/es-es/library/system.string(v=vs.90).aspx) | Una secuencia de caracteres |  |  |
| decimal | [Decimal](https://msdn.microsoft.com/es-es/library/system.decimal(v=vs.90).aspx) |  |  |  |

Todos estos tipos, excepto el tipo OBJECT, suponen que el contenido es un valor final.

A partir de un Dictionary que contiene índices con todos los tipos de datos básicos, se reconoce si un elemento está en dicho dictionary. Si es así, ese tipo es básico, y por tanto no hay que ahondar buscando elementos internos, sino que se capturará su valor cómo valor del elemento.

He tratado de identificar los tipos básicos a partir de los miembros que se recogen con GetMembers, pero en todos los casos se capturaban siempre propiedades o campos, lo que impedía identificarlos con este código:

bool hayPropiedadesOCampos = false;

foreach (MemberInfo m in miembrosClase)

{

if (m.MemberType == MemberTypes.Property || m.MemberType == MemberTypes.Field)

{

hayPropiedadesOCampos = true;

break;

}

}

if (hayPropiedadesOCampos)

{

codigo += mostrarValor(getCodigoByMembers(miembrosClase, objeto)); // Controlar si es un tipo no básico

}

else

{

codigo += mostrarValor(objeto);

}

Esto no ha funcionado, ya que incluso para los tipos básicos, al obtener sus propiedades y campos obtenemos algunos valores. Por ejemplo, para el tipo Int32, al obtener sus miembros aprecen tres propiedades m\_value, min\_value y max\_value, que el sistema interpreta como datos internos de una estructura o tipo complejo.

¿Es más eficaz la serialización de propiedades que la de campos? Comprobar

Parece ser que las propiedades, por el hecho de serlo, generan metadatos en el código compilado, lo cual le permite [algo] “en la biblioteca de clases básicas usando lógica sofisticada”

<http://www.dotnetperls.com/serialize-list>

INTRODUCCIÓN O CONTEXTO

QUÉ HAY POR AHI DISTINTO O PARECIDO A LO MIO

CÓMO HE ABORDADO EL PROBLEMA (DISTINTAS ÓPTICAS DESDE LA QUE HEMOS ABORDADO EL PROBLEMA HASTA DAR CON LA ADECUADA)

CUÁLES SON LOS RESULTADOS OBTENIDOS

6. USO DE ATRIBUTOS Y ATRIBUTOS PERSONALIZADOS

Los atributos son una característica que poseen los lenguajes de alto nivel como C# 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.

También existe la posibilidad de crear atributos propios, atributos personalizados, que 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;

}

}

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 …

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 de .Net. Así, podemos acceder a los atributos que posee cualquier tipo de dato asociado a sus elementos, conocer su contenido y actuar en consecuencia. Para ello usaremos

En .Net se han incorporado 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 en .Net. 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, para reconocer para qué elementos tenemos que aplicar la funcionalidad de nuestra aplicación y para cuales no. Estos atributos son:

* [System.Serializable]
* [System.NonSerialized]. Para campos, indica que no se deben serializar.
* [System.Obsolete(“mensaje de advertencia”, bool generarError)]

Uno de los puntos en los que hacemos esto es a través del 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.

Sobre atributos:

<https://msdn.microsoft.com/es-es/library/z0w1kczw.aspx>

Atributos comunes:

<https://msdn.microsoft.com/es-es/library/z371wyft.aspx>

Crear atributos personalizados:

<https://msdn.microsoft.com/es-es/library/sw480ze8.aspx>

Acceder a atributos personalizados y sus parámetros:

<https://msdn.microsoft.com/es-es/library/z919e8tw.aspx>

Atributos para XML:

<https://msdn.microsoft.com/en-us/library/2baksw0z%28v=vs.110%29.aspx>

7. AÑADIENDO PLUGINS

El sistema está preparado 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 un mecanismo de plug-ins cualquier otro tipo de serialización (JSON, binario, otros esquemas de XML, etc.)

Un plugin 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 plugin 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 plugin de manera que se ajuste a sus necesidades (otros tipos de codificación, traducción de tipos de objeto mediante la deserialización, etc.).

Para añadir a nuestra aplicación un mecanismo de plugins que admita la generación del auto-serializador para una clase determinada, es necesario tener en cuenta tres aspectos:

* Definir un contrato para los plugins. A partir de un interface se define cómo admitirá el sistema la inserción de un nuevo plugin.
* Implementación de las tareas. Las tareas definidas en el interface anterior tienen que implementarse.
* Carga de plugins. Un loader se encargará de cargar en cada momento el plugin específico que tenga que realizar esas tareas, normalmente en función de una comprobación inicial de existencia o llamada a un determinado plugin.

Para tener disponible la posibilidad de inclusión de nuevos plug-ins que permitan variar el modo de funcionamiento de la aplicación, se rediseña la arquitectura de la misma, organizándola a partir de un **Plugin Manager**. Éste se encarga de construir el sistema a partir de la información de plugins que se encuentra en el sistema. Podemos hacer que el mecanismo de plugin identifique y monte los plugins necesarios en la aplicación almacenando todos los plugins en un mismo directorio, haciendo que el programa busque en dicho directorio (y recursivamente en sus subdirectorios) y aplique el/los plugin/s que encuentre en él. Además utilizando un mecanismo de atributos personalizados que identifiquen los plugins a cargar, dónde se encuentran, e incluso las dependencias entre varios plugins para cargarlos todos (por desarrollar).

En sí mismo el Plugin Manager es un plugin, aunque un tanto especial.

Para identificar los plugins se utiliza nuevamente reflection, recoriendo un determinado directorio donde los “candidatos” a plugin deben estar cargados, y reconociendo entre las clases que contiene las que implementen la interfaz definida para los plugins. Así, serán cargados los plugins que la implementen, y serán desechados los que no lo hagan.

La nueva arquitectura se compone de varias capas:

* un núcleo del sistema, que se encargará de cargar el Plugin Manager y el resto de servicios necesarios para que la aplicación pueda arrancar y funcionar.
* un conjunto de plugins adicionales que se podrán cargar en el sistema según las necesidades del mismo a través del Plugin Manager, para ser invocados a voluntad.

Teoría de cómo crear un sistema de plugins en C#:

<http://www.thecoldsun.com/es/content/01-2009/sistema-de-plugins-con-c-parte-i-conceptos>

Otro ejemplo de creación de plugins:

<http://www.codeproject.com/Articles/14339/Enabling-Your-Application-to-Become-a-Plugin-Host>

Se crea la clase PluginManager, que contiene los métodos para cargar plugins, descargarlos, listar todos los plugins cargados, etc.

El método *GetPluginInfo* abre un ensamblado (que recibe como parámetro), y busca en su interior clases que implementen la interfaz IPlugin. Adicionalmente podría comprobar los atributos de esta clase para identificar qué tipo de plugin es y si requiere la carga previa de otros plugins para funcionar (dependencia). Esto se haría analizando sus atributos.

Ejemplo de un mecanismo de plug-ins simple:

<http://www.drdobbs.com/cpp/implementing-a-plug-in-architecture-in-c/184403942?pgno=3>

Otro ejemplo (con una medalla de bronce ganada):

<http://social.technet.microsoft.com/wiki/contents/articles/17501.creating-a-simple-plugin-mechanism.aspx>

Otro ejemplo simple en C++:

<https://msdn.microsoft.com/es-es/library/0x82tk9k.aspx>

MEJORAS EN EL SISTEMA DE PLUGIN

Este mecanismo de plugins 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 plugin, qué otros plugins 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 plugin 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 plugin en un thread independiente. Se puede tomar en consideración esta situación, y definir en los plugin insertables en esta aplicación si se desea que su ejecución se realice en un thread independiente. Puesto que la funcionalidad de los plugins 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 plugin 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 plugin.

- 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.

Más información sobre cómo crear un sistema de plugíns con C#

<https://hmadrigal.wordpress.com/?s=+plug-in+with+reflection>

Arrays multidimensionales. Como rellenarlos en decode:

Utilizando Array.Copy(ar1, im1, ar2, im2, num)

ar1: array1

im1: índice mímino (GetLowerBound)

ar2: array2

im2: índice mínimo

num: número de elementos a copiar

<https://msdn.microsoft.com/es-es/library/z50k9bft(v=vs.110).aspx>

PARA COMPARAR SERIALIZADORES

- 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;  
//Deserialize the Record object back into a new record object.  
Clase01Basica instanciaClase01Basica2 = (Clase01Basica )serializer.ReadObject(stream1);

- 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();  
 instanciaClase01Basica2 = (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();  
  
 instanciaClase01Basica2 = (Clase01Basica) formatter.Deserialize(fs);  
}  
catch (SerializationException e)   
{  
 Console.WriteLine("Failed to deserialize. Reason: " + e.Message);  
 throw;  
}  
finally   
{  
 fs.Close();  
}

- SharpSerializer se usa así:

Descargar el código: <http://www.sharpserializer.com/en/tutorial/index.html>

var serializer = **new SharpSerializer();**

// serialize

serializer.Serialize(instanciaClase01Basica, "test.xml");

// deserialize

Clase01Basica instanciaClase01Basica2 = serializer.Deserialize("test.xml");

// true - binary serialization, false - xml serialization

var serializer = new SharpSerializer(**true**);

// serialize

serializer.Serialize(instanciaClase01Basica, "test.bin");

// deserialize

Clase01Basica instanciaClase01Basica2 = serializer.Deserialize("test.bin");

- Descarga de SharpSerializer:

<http://www.sharpserializer.com/en/download/index.html>

- NetSerializer se usa así:

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

Descargar el código: <https://code.google.com/p/protobuf-net/downloads/detail?name=protobuf-net%20r282.zip>

// write to a file

Serializer.Serialize(outputStream, person);

// read from a file

var person = Serializer.Deserialize<Person>(inputStream);

Manual no oficial de Protobuf-net:

<http://www.codeproject.com/Articles/642677/Protobuf-net-the-unofficial-manual>

Serliazación de struct con Protobuf-net:

<http://stackoverflow.com/questions/20328590/how-to-use-protobuf-for-struct-serialization>

- Nuestro proyecto se usa así:

Generador g = new Generador(instanciaClase01Basica.GetType(), “FORMATO (CSV,BINARIO,etc)”);

dynamic serializador = g.getSerializer();

string codigo = serializador.codificar(instanciaClase01Basica);

serializador.decodificar(codigo, ref instanciaClase01Basica2);

Aunque internamente la clase generada serializador3a usará métodos estáticos, para la invocación se puede utilizar los métodos espejo sobre la instancia del serializador (que siempre existirá). Por supuesto, esta instancia es reutilizable para cualquier objeto de esa misma clase, pero solo para objetos de esa misma clase. Para otra clase, se tendrá que generar otro objeto correspondiente a otro serializador.

Mejoras al proyecto.

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.