ESQUEMA PARA LA PRESENTACIÓN DE PFC

1. INTRODUCCIÓN

2.

Existen muchos serializadores en el mercado, cada uno con sus ventajas y desventajas, válidos para determinados tipos de objetos y no para otros, que generan código serializado para ser almacenado y/o transportado en diversos formatos y contextos.

3.

Con este proyecto hemos conseguido, tras analizar una amplia variedad de serializadores, obtener lo que hemos denominado el *HiperSerializador*, un generador de serializadores versátil que sirva para cualquier tipo de objeto, generando cualquier tipo de código serializado transportable y almacenable en cualquier contexto,…

4.

… capaz de reconocer los atributos estándar o cualquier atributo particular que se desee introducir, ampliable mediante plug-ins y por supuesto, con los mejores resultados en cuanto a rendimiento.

5.

Esta aplicación es un **generador de serializadores particulares de alto rendimiento para objetos en .Net**.

2. SERIALIZACIÓN Y SERIALIZADORES

6.   
Una definción lo más simple posible de serialización: es el proceso consistente en agrupar todos los valores de los elementos de un objeto bajo un determinado formato con el fin de almacenar o distribuir el estado actual de ese objeto.

La deserialización es el proceso inverso, que a partir de la información de los elementos formateada reconstruye el objeto en cuestión en el mismo estado en el que fue serializado.

7.  
Hay una serie de factores o características a tener en cuenta cuando queramos analizar la eficiencia de un programa serializador.

Velocidad: El tiempo que dedica el programa en realizar las tareas de serialización o deserialización de un objeto particular. Influye directamente en el rendimiento. Influye en esta velocidad el número y tipo de los elementos que contenga el objeto.

8.  
Tamaño del stream: Al serializar se genera un código intermedio con la información del objeto. Si este código se tiene que distribuir, el transporte será más eficiente si el código es más reducido. El transporte siempre es el cuello de botella de cualquier aplicación distribuida. Conseguir que el mensaje a enviar sea lo más reducido posible supone aumentar la velocidad del transporte, y por tanto el rendimiento.

9.  
Aplicabilidad: La podemos definir como la capacidad de aplicar la serialización a cualquier tipo de elemento contenido en un objeto. Hay serializadores que tienen problemas o directamente no pueden trabajar con algunos tipos de datos, como arrays multidimensionales, genéricos, ciertos tipos de estructuras, etc.

10.  
Por tanto, con todos estos factores en la mente, el objetivo de nuestro proyecto consiste en generar el serializador idóneo para cada tipo de objeto, tenga los elementos que tenga, obteniendo el código serializado en cualquier formato que el usuario (programador) espere, y si es posible dotarle de opciones avanzadas tales como atributos, formatos personalizados de código intermedio, y añadirle otras funcionalidades distintas a las de la propia serialización.

11.  
Otros serializadores.   
Antes de comenzar con nuestro proyecto, hemos analizado una amplia variedad de serializadores tanto comerciales como open-source, con el fin de identificar sus características, y como se comportan ante distintos tipos de objetos.

En el cuadro de la figura se puede ver los serializadores que hemos estudiado, y que utilizaremos para comparar con nuestro serializador, que aparece el último en la lista, marcado en verde.

Vemos que hay cuatro serializadores de .Net, aunque hay bastantes más, cada uno con sus particularidades, y muchos son subconjuntos de éstos, hemos escogido estos como los más significativos.

Además hay otros serializadores opensource como NetSerializer o SharpSerializer que son muy reputados, y hemos incluido también el proyecto Protobuf de Google en su versión para .Net.

12.  
Comparativa de las características de formatos de salida para nuestro proyecto respecto a los otros serializadores.

Cada serializador genera el código serializado en uno u otro formato. Hay serializadores que admiten generar el código en más de un formato. En la tabla se aprecian los distintos formatos en los que se puede generar el código intemedio con cada serializador. Como se aprecia, nuestro serializador admite cualquier formato de código. De facto, admite XML y CSV, pero se puede programar fácilmente para que admita binario, stream, Json, o cualquier otro tipo de formato.

13   
Siguiendo con la comparación de características, vemos que los serializadores generados por nuestra aplicación admitirían cualquier tipo de datos, a diferencia del resto que tienen ciertas limitaciones. En las conclusiones veremos las diferencias en cuanto a velocidad de proceso, la última de las características que estamos tomando en consideración.

3. HIPERSERIALIZADOR

14.  
Qué es HiperSerializer

Es un pequeño programa que tiene una gran potencia. Consta de 1.500 líneas de código en la versión que genera código intermedio en formato XML y CSV. Escrito en C# con .Net 4.5, consta de una única clase con este constructor.

Recibe como parámetro el tipo para el que va a generar el serializador, y opcionalmente, el formato en el que se espera obtener el código serializado. Por defecto dicho formato es CSV, y este parámetro podría tener cualquier valor si se programan otros formatos a partir de plug-ins.

Para generar el serializador, una vez instanciado, hay que invocar al método getserializer.

15.  
Así es como se invocaría la aplicación. A partir de un objeto que es el que se quiere serializar, se pasa su tipo al constructor del generador de serializadores.

A continuación se invoca al método getserialzier, que devuelve un objeto del tipo MiTipoCodec, que es el serializador particular para objetos del tipo MiTipo.

Como el tipo de este objeto (MiTipoCodec) no estaba definido en tiempo de compilación, sino que se ha generado en tiempo de ejecución, hay que meterlo en una variable de tipo Dynamic para que no haya problemas de compilación, ni se produzcan excepciones en tiempo de ejecución, que es lo que pasaría si usáramos un Object para contenerlo.

16.  
La clase MiTipoCodec posee dos métodos para serializar y deserializar llamados codificar y decodificar.

Su funcionamiento es el siguiente:

Cuando se desee serializar un objeto del tipo MiTipo, se invoca al método codificar, pasándole dicho objeto como parámetro, y recibiendo un string con el objeto serializado.

Cuando se desee serializar, a partir de ese string, se invoca al método deserializar, pasándole como parámetro el propio string que contiene la serialización del objeto, y pasando por referencia el objeto que se llenará con los datos serializados.

El string intermedio puede ser XML o CSV, en función del formato definido en el constructor de Generador.

17.  
Aquí vemos un ejemplo del uso de estos métodos para serializar y deserializar un objeto.

18.

3.2. Función de HiperSerializer

Generar un serializador para cualquier clase cuyo tipo se le indique en el constructor. Este serializador realiza la tarea de serializar y deserializar cualquier objeto de esa clase de manera óptima.

3.3. Cómo funciona el programa

El programa recibe como entrada un tipo de dato de .Net, realiza una comprobación de todos los elementos susceptibles de ser serializados, y va generando el código de una aplicación capaz de realizar la serialización de instancias de ese tipo. A continuación, compila al vuelo ese código, instancia un objeto de ese programa y eso es lo que devuelve como salida.

El programa generado realiza la tarea exclusiva de serializar y deserializar objetos de ese tipo. Además, lo hace de una manera muy eficiente y rápida, como se verá en las pruebas realizadas.

La aplicación Generador.dll, se integra en cualquier aplicación .net simplemente cargándose como un DLL más.

A partir de ese momento, se puede utilizar instanciando la clase Generador y llamando al método getSerializer. A la hora de instanciar, al constructor hay que pasarle como parámetro un System.Type. En el momento de invocar al método getSerializer, se obtendrá el serializador particular para ese tipo en una variable de tipo dinámico, ya que el objeto devuelto será de una clase particular no disponible en tiempo de compilación.

3.4. Cómo genera cada serializador

Una vez que se invoca el método Generador.getSerializer, el programa ya sabe para qué tipo tiene que generar el código de su serializador particular. Para ayudar en esta tarea se utiliza el espacio de nombres System.Reflection.

Reflection es una característica de .Net que permite trabajar con los elementos que componen un tipo, campos, propiedades, métodos, eventos, etc. Nos permite obtener su nombre, tipo, y muchas más cualidades (si es público o privado, sus atributos, etc.).

Se usa Reflection para capturar todos los elementos del tipo susceptibles de ser serializados. Para cada elemento, se generará el código que permitirá colocar en un string la información de su valor (serialización) o capturar de dicho string su valor para colocarlo en el elemento correspondiente (deserialización) según el formato que se desee para la salida de la serialización. Ambos procesos se hacen de manera sucesiva para cada elemento, por lo que el código de la serialización se genera a la vez que el código de la deserialización.

Por ejemplo: Clase01Basica

public class Clase01Basica

{

public string v1 = "1";

public int v2 = 2;

}

El código del serializador para esta clase se conformará así:

public class Clase01BasicaCodec{

public static string encode(Fase02.Clase01Basica obj){  
 StringBuilder texto = new StringBuilder();  
 texto.Append(obj.v1.ToString() + ",");  
 texto.Append(obj.v2.ToString() + ",");  
 return texto.ToString();  
 }

public static void decode(ref String codigo, ref Fase02.Clase01Basica obj){  
 Queue<string> elementos = new Queue<string>(codigo.Split(','));  
 obj.v1 = elementos.Dequeue();  
 obj.v2 = Int32.Parse(elementos.Dequeue());  
 }

}

Para cada elemento recogido con Reflection con getMembers() se añade la línea texto.Append(obj.v1.ToString() + “,”)

4. COMPARATIVA

5. MEJORAS

6. CONCLUSIONES