**Multi Document**

**Introduction**

**MultiDocument** is a reusable library which allows reading, modification and conversion of different file formats. This library allows adding support for new user supplied file formats. **MultiDocument** is a very flexible and extensible library.

One of the core principles of **MultiDocument** library is possibility of extension functionality without recompiling source code. This extensibility is achieved by using an **Abstract Factory** and **Builder** patterns. Even though some basic formats (e.g. “**binary**” and “**xml**”) are supported by default, their functionality are also provided via factories, so you can extend functionality by providing your own factory which can either provide its own logic or provide logic which extends existing one.

**Architecture**

The basic class of **MultiDocument** library is **MDocument<T>**. You can use this class as some kind of generic container (like **List<T>**). It implements **IMReader<T>**, **IMWriter<T>** and **IMStream** interfaces. **IMReader<T>** interface provides functionality for reading operations, **IMWriter<T>** provides functionality for writing operations and **IMStream** provides functionality for getting count of elements.

First at all you should understand that **MDocument<T>** is not the only way you could get functionality of **IMReader<T>**, **IMWriter<T>** and **IMStream** interfaces. There are two basic classes for both reading and writing operations: **MBinaryReader<T>**, **MXmlReader<T>**, **MBinaryWriter<T>** and **MXmlWriter<T>**. As long as this classes are also implements those interfaces you can use them separately or in conjunction with other library classes to get more flexibility.

As you can see all these classes are generic classes. The template parameter **T** represents a type of record that will be saved in the target file. It can be structure or class. The main goal of using this library is that you can select only some bunch of fields and properties you want to be used (serialized) in serialization process. To do this you should mark each field and property you want to be serialized with **ProcessableAttribute** attribute. It gives you a very flexible way for managing of creation of file format by specifying the record format. So you do not have to hard-code which field should be serialized at reader-writer level. The library lets you do this on the user level while you’re defining a class of record. Otherwise we had to write new classes of reader and writer on each new record.

By setting up different properties of **ProcessableAttribute** attribute you can tune the process of serialization:

1. setting up the alias for field or property (it's useful for XML format) - **Alias** property;
2. setting up the rank for field or property (so you can specify the sequence at which the specified properties will be serialized) - **Rank** property;
3. setting up the serialization factory for field or property (so this factory will be used by library when it need to serialize/deserialize specified field or property) - **SerializerFactory** property;
4. setting up the data verifier (when this property is set, the library will use it to verify specified property or field) - **DataVerifier** property.

**NOTE. It’s important to mark the properties and fields you want to be serialized with ProcessableAttribute attribute if you use standard classes (like MBinaryReader<T>, MXmlReader<T>, MBinaryWriter<T> and MXmlWriter<T>). Also note that these standard classes are used internally by MDocument<T> class.**

Also note that it’s not required specify this attribute for records if your custom classes that implement **IMReader<T>** and **IMWriter<T>** interfaces do not use this attribute in their logic.

The library also provide functionality for data converting. All classes in library, that use converters (**MDocument<T>**, **MBinaryReader<T>** and **MXmlReader<T>**) use converters that implement **IDataConverter<T>** interface. The **MBinaryReader<T>** and **MXmlReader<T>** classes have in their constructor parameter converter of type **IDataConverter<T>** which is by default has null value. You can specify a concrete converter in constructor of one of these readers, so you can then call **ConvertDocument()** method to convert all reader data using specified converter. So, all default readers (**MBinaryReader<T>** and **MXmlReader<T>**) implement Builder pattern for conversion purposes. There are two basic data converter: **MBinaryConverter<T>** and **MXmlConverter<T>**. They can be used in conjunction with readers as implementation of **Builder** pattern or they can be used as separate data converters.

**Interfaces**

**IMWriter<T>**

interface IMWriter<T> : IMStream where T : new()

{

/// <summary>

/// Adds the record to the end of the stream

/// </summary>

/// <param name="record">The record to be added to the end of the stream </param>

void Add(T record);

/// <summary>

/// Adds the records of the specified collection to the end of the stream

/// </summary>

/// <param name="records">The collection whose records should be added to the end of the stream</param>

void Add(IEnumerable<T> records);

/// <summary>

/// Replaces the record at specified position with new record value

/// </summary>

/// <param name="pos">Specifies the record position to be replaced by the new value</param>

/// <param name="newRecord">Specifies the new record value</param>

void Replace(int pos, T newRecord);

/// <summary>

/// Replaces all occurrences of a specified record in the stream with another specified record

/// </summary>

/// <param name="oldRecord">Specifies the old record value to be replaced by the new record value</param>

/// <param name="newRecord">Specifies the new record value</param>

void Replace(T oldRecord, T newRecord);

/// <summary>

/// Removes a record from the stream

/// </summary>

/// <param name="pos">The zero-based position of record to be deleted</param>

void Remove(int pos);

/// <summary>

/// Removes the specified record from the stream

/// </summary>

/// <param name="record">The record to be removed from the stream</param>

void Remove(T record);

/// <summary>

/// Removes all records from the stream

/// </summary>

void Clear();

/// <summary>

/// Flushes all document to file

/// </summary>

void Flush();

}

**IMReader<T>**

interface IMReader<T> : IMStream where T : new()

{

/// <summary>

/// Reads the record at specified position

/// </summary>

/// <param name="pos">An index of the record to read</param>

/// <returns>Return the record that has been read from a stream</returns>

T Read(int pos);

/// <summary>

/// Reads all records from a stream

/// </summary>

/// <returns>A list of all records that has been read from a stream</returns>

IList<T> ReadAll();

}

**IMStream**

interface IMStream

{

/// <summary>

/// Returns the number of elements in a stream

/// </summary>

int Count { get; }

}

**IMDataConverter<T>**

interface IMDataConverter<T> where T : new()

{

/// <summary>

/// Converts a value of type T and adds converted value to the result document

/// </summary>

/// <param name="value">The value to be converted</param>

void Convert(T value);

/// <summary>

/// Clears all result document, so you can start another conversion process

/// </summary>

void ClearDocument();

/// <summary>

/// Flushes all converted data to file

/// </summary>

void Flush();

}

**IMDocumentFactory<T>**

interface IMDocumentFactory<T> where T : new()

{

/// <summary>

/// Gets a new instance of writer class that implements IMWriter<T> interface with specified

/// path and file format

/// </summary>

/// <param name="path"> An absolute path for the file that the current IMWriter<T> object

/// will encapsulate.</param>

///

/// <param name="format">A string that determines format of file specified by path</param>

///

/// <returns> A new instance of writer class that implements IMWriter<T> interface with

/// specified path and file format</returns>

IMWriter<T> GetWriter(string path, string format);

/// <summary>

/// Gets a new instance of reader class that implements IMReader<T> interface with specified path

/// and file format

/// </summary>

/// <param name="path"> An absolute path for the file that the current IMReader<T> object will

/// encapsulate.</param>

///

/// <param name="format">A string that determines format of file specified by path</param>

///

/// <returns> A new instance of reader class that implements IMReader<T> interface with

/// specified path and file format</returns>

IMReader<T> GetReader(string path, string format);

/// <summary>

/// Gets a new instance of converter class that implements IMDataConverter<T> interface with specified

/// path and file format

/// </summary>

/// <param name="path"> An absolute path for the file that the current IMDataConverter<T> object will

/// encapsulate.</param>

///

/// <param name="format">A string that determines format of file specified by path</param>

///

/// <returns> A new instance of converter class that implements IMDataConverter<T> interface

/// with specified path and file format</returns>

IMDataConverter<T> GetConverter(string path, string format);

/// <summary>

/// Gets a collection of supported file formats

/// </summary>

IEnumerable<string> SupportedFormats { get; }

}

**IDataVerifier**

interface IDataVerifier

{

/// <summary>

/// Validates the specified object.

/// </summary>

/// <param name="obj">The value of the object to validate.</param>

void Verify(object obj);

}

**IDataSerializer**

interface IDataSerializer

{

/// <summary>

/// Serializes the object to the byte array.

/// </summary>

/// <param name="obj">Object to be serialized</param>

/// <returns>A byte array that represents serialized object</returns>

byte[] Serialize(object obj);

/// <summary>

/// Deserializes the specified byte array into an object

/// </summary>

/// <param name="buffer">An array of bytes that represents an object</param>

/// <param name="type">The type of deserialized object</param>

/// <returns>Deserialized object</returns>

object Deserialize(byte[] buffer, Type type);

}

**DataSerializerFactory**

public abstract class DataSerializerFactory

{

/// <summary>

/// Gets the data serializer for specified type

/// </summary>

/// <param name="type">The type of object to get data serializer for</param>

/// <returns>Returns the data serializer for specified type</returns>

public abstract IDataSerializer GetDataSerializer(Type type);

}

**UML Class Diagrams**

In this section are shown UML class diagrams which illustrate interaction between classes. The interaction between readers, writers and data convertors are shown below (see Figure 1). As you can see interaction between readers and converters represents **Builder** pattern.

The interaction between **MDocument<T>** class, abstract factories, readers, writers and data converters are shown in the Figure 2. As you can see **MDocument<T>** class only aggregates a reference to **IMDocumentFactory<T>**, so it depends only from this factory. This technique provides a big flexibility in extension library functionality. All you have to do to extend functionality of library is supply constructor of **MDocument<T>** class with concrete factory that can extend existing **BasicDocumentFactory<T>** factory functionality or provide a brand new logic.



Figure 1: Interaction between readers, writers and converters



Figure 2: Interaction between MDocument<T> class, abstract factories,

readers, writers and data converters

**Advantages and disadvantages**

Advantages:

1. Attribute-based development provides very flexible control over data serialization process via ability of decorating the properties and fields with the specific attribute, data validation, serialization and ranking.
2. Ability to extend the library functionality by providing specific abstract factory without recompiling the library source code.
3. A loosely coupled code that let you use readers, writers and converters as part of some classes or as a separate entities that provide the same effect.

Disadvantages:

1. The base non-custom classes (e.g. **MBinaryReader<T>**, **MBinaryWriter<T>**, **MXmlReader<T>**, **MXmlWriter<T>**, etc.) can only serialize records that have properties and fields based on primitive types (e.g. System.Byte, System.Int16, System.String, System.DateTime, etc.).
2. There is no any synchronization for that moment.

**Possible future improvements**

Improvements:

1. Add data synchronization.
2. Implement **IEnumerable<T>** interface in **MDocument<T>** and all readers and writers.
3. Add support of indexing.
4. Add support for asynchronous processing.