Annals of DAAAM for 2010 & Proceedings of the 21st International DAAAM Symposium, Volume 21, No. 1, ISSN 1726-9679
ISBN 978-3-901509-73-5, Editor B. Katalinic, Published by DAAAM International, Vienna, Austria, EU, 2010
Make Harmony Between Technology and Nature, and Your Mind will Fly Free as a Bird
Annals of DAAAM International

METHODS OF HANDLING XML FILES FOR A DECISION SUPPORT SYSTEM FRAMEWORK

STANCIU, C[ristina] O[felia] & COJOCARIU, A[drian]

Abstract: *XML provides an organized and elegant way of storing data, the main advantage is adaptability, and also that XML modeled data are readable by any user. The paper shows original methods of handling XML files used for data representation within a decision support system framework.*
Key words: *XML, decision support system, abstract methods*

1. INTRODUCTION

Storing data used for a decision support system in the XML file has been inspired by the open source project WEKA. The idea of using the C# programming language together with the XML format has proven to be a great success, as they are new and modern technologies. The assembly of the data representation in the XML format, together with the XSD schema and applications developed with object oriented programming languages that offer function libraries for processing these data models represents a powerful, efficient and mostly elegant solution [Stanciu 2009].

The XML document handled by the integrated system represents a so-called relation, the *Relation* element being the root node of the XML tree. This element aggregates two major sub-elements: the attribute definition list (*Attributes*) and the instance list (*Instances*).

2. LOADING AND SAVING XML FILES

The framework needs to be able to load and save files representing decision relations. For this purpose we attempted to develop a unitary procedure for input/output operations on XML files. In this way we materialized a generic abstraction, a high-level representation of an XML file which implements the standard input/output operations (all-purpose, applicable to a wide category of XML files) and also other operations and published properties.

Technically this abstraction begins from a .NET abstract class named *AbstractXmlDocument*. The class prototype is declared like the following:

```
public abstract class AbstractXmlDocument
```

This class will be "aware" of the name of the XML file it represents. This name can be missing at first (set to *null*), if the XML file corresponds to a new file; the file name will not be set until the class instance is saved. This is achieved with the technique known as overloading, in this particular case *constructor overloading*, which presumes the existence of two or more methods in the class that share the same name but have a different number of parameters or different parameter types. Essentially, the prototypes of the two overloaded constructors are presented like the following:

```
public AbstractXmlDocument(string fileName)
```

and

```
public AbstractXmlDocument()
```

In this way any built system that uses this class will be able to choose from one of the following variants:

- Creating an XML file with an associated file name (through the *filename* input parameter) and loading the XML contents into memory, provided that the file exists physically on disk;
- Using the XML abstraction without specifying a file name and wait for after populating the XML contents to attempt to save the XML file on disk using a file name specified during this step.

3. INPUT/OUTPUT OPERATIONS WITH XML FILES

It must be mentioned from the beginning that the input/output operations with XML files represented using this abstract class are making use of the *DOM* technology [van der Vlist 02]. Adjacently, these input/output operations will be aware of an XML schema which they will be using to validate the XML file.

Given that the *AbstractXmlDocument* class knows nothing yet about the format and contents of the actual XML file on disk, the XML schema details to validate against will have to be provided by upper levels in the class hierarchy, from derived classes.

In our situation, these details are being materialized through:

- The name of the integrated resource that identifies the *XSD* file representing the XML schema;
- The *XML namespace* or the *xmlns* attribute value that represents the starting point in schema validation;
- Optional, the name of the *.NET assembly* containing the XML schema resource. This will be used only when the integrated schema resource is not embedded in the same .NET assembly as the class being implemented (extending the *AbstractXmlDocument* class).

Loading the XML file from disk (and simultaneously validating it against the schema, if any is specified) is implemented using the *Load* method. This method has the task of creating a proper environment prior to reading the XML file contents (serializing it through the *DOM* parser engine). One of these steps is reading the XML schema, if any, specifying there is going to be a schema validation step and pointing to a .NET *delegate* method which will be receiving and processing any validation errors or warnings emerged out of the parsing process. It is only after finalizing all the prerequisite steps (which will all be omitted if no XML schema is specified) that an object of the type *System.Xml.XmlDocument* will be instantiated and its Load method will be invoked, which will result in transferring the whole XML contents into memory.

Talking about the *XmlDocument* class, at this point we can highlight the fact that the *AbstractXmlDocument* class can be looked at as a *wrapper* class over *XmlDocument*; we can state that the class described here *wraps up* the .NET *XmlDocument* class while adding new features like the XML schema validation.

The *Load* method implementation is technically described below:

```
Assembly assembly;
if (_assemblyName == "")
    assembly = Assembly.GetExecutingAssembly();
else
    assembly = Assembly.Load(assemblyName);
Stream xsdStream =
assembly.GetManifestResourceStream(schemaResource);
schemaReader = new XmlTextReader(xsdStream);
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add(namespace, schemaReader);
settings.ValidationEventHandler += new
   ValidationEventHandler(LoadValidationErrorHandler);
reader = XmlReader.Create(new
XmlTextReader(fileName), settings);
XmlDocument doc = new XmlDocument();
doc.Load(reader);
this.Load(doc);
```

The last line of code in this listing is remarkable because it transfers the processing to another *Load* method that is supplied with a parameter of the *XmlDocument* type. This method is abstract and overloaded at the same time (which implicitly forces the class to be abstract as well). This method offers derived classes the possibility of extracting data from the XML contents locally (using the *XmlDocument* input parameter). The method prototype will be:

```
protected abstract void Load(XmlDocument doc);
```

The *Load* method is supplied with the *protected* access specifier in order to be accessed only in *AbstractXmlDocument*-derived classes.

The *delegate* method *LoadValidationErrorHandler* is in charge of receiving notifications about errors or warnings that come up during the parsing process of the XML contents. The default implementation in the *AbstractXmlDocument* class is that of redirecting error messages to the standard error display (this message will not be visible unless the application using the class is running in a console window) and of raising an exception when encountering critical errors. Nevertheless, this method is declared using the *virtual* keyword offering derived classes the possibility of overriding it and supplying a new behavior when encountering processing errors or warnings.

The method header is the following:

```
protected virtual void LoadValidationErrorHandler(
            object sender, ValidationEventArgs args)
```

Once we defined the abstraction of a generic XML file through the *AbstractXmlDocument* class, we will exemplify with one of its customizations achieved by deriving from it. In this case the XML document that needs to be read and written by the integrated system represents a decision *relation*. Thus, we will be defining the *RelationDocument* class as the following:

```
public class RelationDocument: AbstractXmlDocument
```

Given that the name of the XML file will still be unknown at the level of the class representing the decision relation (this file name being a dynamic property of this class – as well as of the base class), this will remain at the stage of a parameter passed through the constructor to the *RelationDocument* class as well. What we do know, though, are the details referring the XML schema that will be used to validate any XML document representing decision relations. At this point of software development the place within the project of the integrated resource representing this XML schema should be known, as well as the namespace for XML files of this type.

Practically, for XML files describing in this paper we have chosen the namespace **urn:localhost:ML.Relation**. Also, with respect to a hierarchical architecture of the files within the project and placing the XML schema appropriately within this hierarchy, the full name of the integrated resource representing the XSD schema should be **ML.Lib.Relation.Relation.xsd**.

In these circumstances the constructor of the *RelationDocument* class should take the following form:

```
public RelationDocument(string fileName) : base(fileName,
        "urn:localhost:ML.Relation",
        "ML.Lib.Relation.Relation.xsd")
```

As it can be noticed, the *filename* parameter stays in place, being passed to a superior level, as opposed to parameters identifying XML schema details, which are defined as constants at this level and passed to the level of the base class (*AbstractXmlDocument*) for later use.

A portion of the developer-friendliness offered by the *AbstractXmlDocument* class is observed here: the fact that only one line of source code supplies the necessary and sufficient information to load, save and validate an XML document against the schema. These emerge from the strongly-developed abstraction of the base class. This provides highly simplified means of instantiating and modeling a specialized class for handling and managing a certain type of XML document.

The powerful abstraction of the *AbstractXmlDocument* class reaches its pinnacle at the stage of implementation of the loading and saving routines in derived classes. This is remarkable through the abstract methods *Load* and *Save* defined in the base class, methods that any derived class will be obliged to implement. Within the derived class, these methods will have the chance to focus strictly on the XML contents and extracting information from it, respectively storing data into it.

The form of these methods in the derived class is:

```
protected override void Load(XmlDocument doc)
```

and

```
protected override void Save(XmlDocument doc)
```

The *XmlDocument*-typed parameter represents the XML document used to extract or store data. No other operation is required upon this parameter, like creating the XML document or writing the XML signature line in the physical file on disk.

4. CONCLUSION

As presented in this paper, using the *AbstractXmlDocument* class powerful abstraction levels will result in having only two basic concerns when implementing a class that needs to serialize data to and from an XML file, and perhaps verify that the file conforms to a pattern defined by an XML schema: that of extracting data from the XML contents and storing data in this XML hierarchy.

5. REFERENCES

Hamilton, H., Gurak, E., Findlater, L., Olive, W. (2002), *Knowledge Discovery in Databases*, University of Regina, Canada

Holmes, G., Donkin, A., Witten, I.H. (1994), *Weka: A machine learning workbench*, Proceedings of Second Australia and New Zealand Conference on Intelligent Information Systems, Brisbane, Australia

HyoungDo, K., *An XML-based modeling language for the open interchange of decision models*, Decision Support Systems, Volume 31, Issue 4, Pages 429-441, October 2001

Stanciu, Cristina – Ofelia, A. Cojocariu, *XML Technologies for Improving Data Management for Decision Algorithms*, 20th DAAAM International Conference, Vienna, 2009

van der Vlist, E. (2002), *The W3C's Object-Oriented Descriptions for XML*, O'Reilly Publishing