A PERSISTENT CROSS-PLATFORM XML-BASED CLASS OBJECTS CONTAINER

MICHAL BLIŽŇÁK[1], TOMÁŠ DULÍK[2], VLADIMÍR VAŠEK[3]
Department of Applied Informatics[1,2], Department of Automation and Control Engineering[3]
Faculty of Applied Informatics, Tomas Bata University
Nad Stráněmi 4511, 760 05, Zlín
CZECH REPUBLIC
bliznak@fai.utb.cz[1], dulik@fai.utb.cz[2], vasek@fai.utb.cz[3]

Abstract: This paper introduces new open-source cross-platform C++ software library able to store, serialize and deserialize hierarchically arranged class instances and their data members via XML files. The library is based on mature cross-platform library called wxWidgets so it can be successfully used on many target platforms such as MS Windows, Linux or OS X. The article describes an inner structure of the software library, used principles, and illustrates the usage on simple examples as well.

Keywords: Data, class, persistence, container, serialization, XML, tree, list, C++, wxWidgets, wxXmlSerializer, wxXS

1 Introduction

The ability to store complex data processed by software applications is one of the most important features provided by various programming frameworks or software libraries. Most of them already offer some suitable technology, such as various types of configuration files, integrated XML parsers/builders or database layers. Unfortunately, majority of these technologies are designed to store only raw data (e.g. via database layers) or they require a lot of additional programming to process complex data types. Typical and most difficult case is the implementation of persistent storage for class instances and their hierarchy. In high-level programming languages like Java or Python, it is very easy to solve this by using serialization, however, there are virtually no options for this in lower-level programming languages like C++.

If the requirement is to implement a simple storage for raw data, the best solution is probably to use a suitable database system/layer. Nowadays database servers and software libraries supporting various database clients are able to store even pretty complex data via technologies designated for mapping of program objects (class instances) to database records (for example JDO for Java programming language [2]). Moreover, some of current database systems allow user to store the data not only to local or remote database server but also to local file using SQL commands (for example SQLite database [3] or Firebird embedded [4]).

Unfortunately, all of these database technologies lack the ability to preserve the hierarchical relations between object class instances. It means the user can store data records but cannot define their hierarchy (who is the parent and who is the child, etc).

The goal of this paper is to introduce a new simple software library called wxXmlSerializer [8] (shortly wxXS) which fills the gap in the nowadays offer of available data persistence technologies. The wxXS is designed for storing not only raw data, but also their hierarchical relationship.

2 What the wxXmlSerializer is

Generally, the wxXS is a cross-platform software library written in C++ programming language based on wxWidgets [1] which offers a functionality needed for creation of **persistent hierarchical data containers** able to store various **C++ class instances** (can be regarded as complex **data records**). wxXS allows users to easily serialize hierarchically arranged class instances and their data members to an XML structure (storable in various output streams) and deserialize them later. Currently supported data types serializable by the wxXS are:

- Generic data types such as: `bool`, `char`, `int`, `long`, `float`, `double`
- Most frequently used wxWidgets data types: `wxString`, `wxPoint`, `wxSize`, `wxRealPoint`, `wxPen`, `wxBrush`, `wxFont`, `wxColour`,
- `wxArrayString`, array of `wxRealPoint` values, arrays of common generic data types and list of `wxRealPoint` values

- Dynamic or static instances of the serializable base class itself.

Moreover, the library architecture allows user to extend built-in list of supported data types thus new I/O handlers for custom or currently unsupported data types (even user-defined data structures) can be created by the user using a set of code macros provided by the library and small amount of manual programming.

The library can be used for wide range of application scenarios, e.g.:

- simple saving and loading of the program settings/configurations
- as a persistent dynamic n-ary tree-based data container with methods needed for comfortable management of its items (useful for the software applications managing their data in tree controls, applications working with diagrams, etc.).

3 The Technological Background

wxXS library was created as an add-on to well known cross-platform software library wxWidgets [1]. wxWidgets gives the programmers a single, easy-to-use API for writing their applications on multiple platforms that still utilize the native platform's controls and utilities. Link with the appropriate library for your platform (Windows/Unix/Mac, others coming shortly) and compiler (almost any popular C++ compiler), and your application will adopt the look and feel appropriate to that platform. On top of great GUI functionality, wxWidgets supports network programming, streams, clipboard and drag and drop, multithreading, image loading and saving in a variety of popular formats, database support, HTML viewing and printing, and much much more [1].

wxXS library uses the **streams**, **XML** and **RTTI** classes provided by the wxWidgets so it can be used only together with this library. However, this "disadvantage" is balanced by the fact, that these crucial technologies are maintained and improved continuously by the wide and reliable open-source community. Moreover, the license policy [6] used by the wxWidgets library do not restrict the programmer in any way so the applications and derivates based on the wxWidgets can be distributed both as an open-source projects and commercial applications. wxXS itself is created under the same license so it can be used absolutely freely for any purpose, even for commercial projects.

4 The Library Structure

wxXS consists of three main classes encapsulating its basic functionality. It includes also several auxiliary classes encapsulating the I/O functionality for various data types and implements typed data containers used by the library. Now let's take a look to the purpose of the three main library classes which are:

- **`wxXmlSerializer`** class
- **`xsSerializable`** class
- **`xsProperty`** class

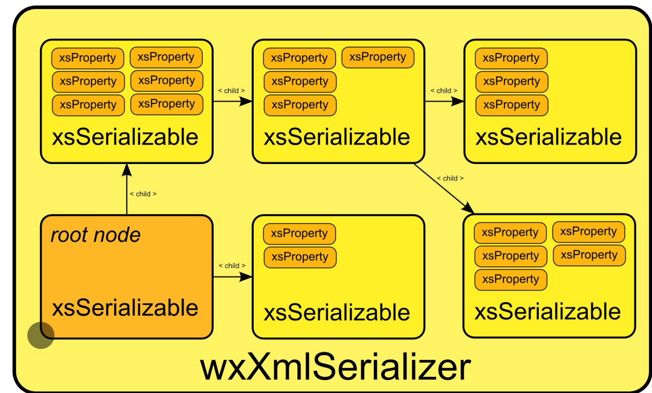

Figure 1: The library structure

`wxXmlSerializer` class is the main **data manager** class and implements the common data container functionality. Its member functions allow user to manage instances of serializable classes (encapsulated by the `xsSerializable`) and provide the I/O functionality (serialization and deserialization of stored serializable class objects). This class can be used as it is or as a base class for various derivations enhancing its built-in functionality.

`xsSerializable` class is the base class for so called **"serializable" classes** (i.e. classes manageable by the `wxXmlSerializer` class). It provides functionality needed for hierarchical arrangement of serialized class instances (every class instance includes linked list of another `xsSerializable` class instances, i.e. its *children*), I/O operations and it also holds information about serialized data members (instances of `xsProperty` class).

`xsProperty` class encapsulates a single data members (**properties**) of serialized class object (an instance of `xsSerializable` class). It stores information about memory address of the data member, its data type, default value, a name of the data member in the XML structure and flag telling

whether the property should be serialized or not.

I/O and data conversion operations provided by the `xsSerializable` class are performed via the **`xsPropertyIO`** I/O handler class and its derivates, which are responsible for conversion of serialized property values to/from its string representation and for reading and writing of this textual information from/to the XML structure.

Except built-in support for common data types the user can simply create new I/O handler class by using set of code macros defined in the library headers. This powerful feature will be discussed in more details later.

5 wxXS: From Simple Data To Hierarchical Data Container

wxXS library can be used in many ways but the main idea is following: the user can create classes derived from **`xsSerializable`** base class and then define which of the class members will be serialized and which not. For serialization of these class instances, it is necessary to add them to a data manager, which is an instance of **`wxXmlSerializer`** class. There is also one instance of **xsSerializable** class called a **root item** included in the data manager class as its member object. All the other serialized objects added to the data manager are inserted into the linked list of the root item. Serialized class data members called properties are encapsulated by an **xsProperty** class

The xsProperty class instances can be created in several ways:

- using universal macros `XS_SERIALIZE(member, field)` or `XS_SERIALIZE_EX(member, field, defval)`
- using one of defined macros designed for particular data type (e.g. `XS_SERIALIZE_LONG(member, field)` or `XS_SERIALIZE_LONG_EX(member, field, defval)`
- using the function `xsSerializable::AddProperty(xsProperty *property)`

The argument *member* is the name of data member, which should be serialized and the argument *field* is a name used for its identification in the output XML structure. The macros must be placed somewhere in a class implementation code (typically in a constructor).

Macros with suffix "`_EX`" in their names allow user to define default property value. In this case, the property is serialized only if its current value differs from the default one. This approach leads to smaller size of the output XML structure because only changed property values are serialized.

Now let's illustrate these mechanisms on a simple example implementing an application settings class serialized via wxXmlSerializer.

The first step is declaration of a serializable class encapsulating the application settings.

Example 1:

```
class Settings : public xsSerializable
{
public:
    // RTTI and clone ability must be
    // supported by the class
    XS_DECLARE_CLONABLE_CLASS(Settings);
    // Default constructor.
    Settings();
    // Copy constructor needed by the
    // serializer's class
    Settings(static Settings &obj);
    // Destructor.
    virtual ~Settings(){;}
protected:
    // Protected serialized data members.
    wxString m_sTextData;
};
```

The implementation of the serializable class declared above is straightforward as well:

```
// Define RTTI and the clone ability
XS_IMPLEMENT_CLONABLE_CLASS(Settings,
xsSerializable);

// Default constructor
Settings::Settings()
{
    // Initialize member data
    m_sTextData = wxT("Textual data
encapsulated by the class object");
    // Mark serialized members
    XS_SERIALIZE(m_sTextData, wxT("text"));
}

// Copy constructor
Settings::Settings(static Settings &obj)
: xsSerializable(obj)
{
    // Copy the member data.
    m_sTextData = obj.m_sTextData;
    // Mark the data members which should be
serialized.
    XS_SERIALIZE(m_sTextData, wxT("text"));
}
```

Now the data manager (serializer) class must be defined to handle an instance of the *Settings* class. In

this example, the standard wxXmlSerializer class is used. Generally, there are two ways how to handle the serializable class objects by the serializer class:

- one serializable objects can be set as a root node of the serializer,
- several serializable objects can be appended to a default root node included in the serializer or to another already managed serializable objects..

These quite different approaches differ in the way the stored serialized class instance can be handled and accessed. For better understanding, both of these ways are discussed bellow.

Let us use the first mentioned way to serialize the Settings class object to a file called *"settings.xml"* stored on a harddrive. The code bellow can be placed anywhere in the application implementation:

Example 2:

```
// Create a serializer object.
wxXmlSerializer m_XmlIO;

// Initialize the serializer.
m_XmlIO.SetSerializerOwner(wxT("SettingsSamp
le"));
m_XmlIO.SetSerializerRootName(wxT("settings"
));
m_XmlIO.SetSerializerVersion(wxT("1.0.0"));

// Create a serialized settings class object
with its default values.
Settings *m_pSettings = new Settings();
if( m_pSettings )
{
    // Insert the object into serializer as
    // its root node.
    m_XmlIO.SetRootItem(m_pSettings);

    m_XmlIO.SerializeToXml(wxT("settings.xml
"), xsWITH_ROOT);
}
```

A content of the output XML file is:

```
<?xml version="1.0" encoding="utf-8"?>
<settings owner="SettingsSample"
version="1.0.0">
    <settings_properties>
        <object type="Settings">
            <property name="text"
type="string">Textual data encapsulated by
the class object</property>
        </object>
    </settings_properties>
</settings>
```

The stored XML file can be loaded back in a simple way:

```
if( wxFileExists(wxT("settings.xml")) )
{
    // Load settings from file
    m_XmlIO.DeserializeFromXml(wxT("settings
.xml"));
}
```

After deserialization, loaded data can be accessed in a very simple way using a function called wxXmlSerializer::GetRootNode(), which returns a pointer to the serializer's root node (in our example to a class object encapsulating the stored data).

The second possible way of managing the stored serializable objects is illustrated in the next example. Assume existing serializable class called TreeNode created in a similar way like the Setting class from the previous example. This class poses as a node in a tree structure encapsulating some user-defined data. Let us create a few of these nodes, add them to the serializer object and store the serializer content to a disk file:

Example 3:

```
// Create a serializer object.
wxXmlSerializer m_XmlIO;

// Initialize the serializer.
m_XmlIO.SetSerializerOwner(wxT("TreeSample")
);
m_XmlIO.SetSerializerRootName(wxT("tree"));
m_XmlIO.SetSerializerVersion(wxT("1.0.0"));

// Create new tree node object...
TreeNode *m_pNode = new TreeNode();
// ... and assign it to the serializer as a
child of the root node.
m_XmlIO.AddItem(n_XmlIO.GetRootNode(),
m_pNode);
// Another possible way how to assign a new
// serializable object as a
// child to existing parent object is for
// example like this:
m_pNode->AddChild(new TreeNode());
TreeNode *m_pNode2 = m_pNode->AddChild(new
TreeNode());
m_pNode2->AddChild(new TreeNode());

// Serialize tree data to given output file.
m_XmlIO.SerializeToXml(wxT("data.xml"));
```

A content of the output XML file created by previous code is:

```
<?xml version="1.0" encoding="utf-8"?>
<tree owner="TreeSample" version="1.0.0">
```

```
    <object type="TreeNode">
        <object type="TreeNode" />
         <object type="TreeNode">
            <object type="TreeNode" />
        </object>
    </object>
</tree>
```

Whole tree structure can be re-built by a single program line later:

```
// Load tree data from file
m_XmlIO.DeserializeFromXml(wxT("data.xml"));
```

After successful deserialization the stored class instances can be accessed by member functions of `xsSerializable` class like:

- `xsSerializable::GetParent`,
- `xsSerializable::GetFirstChild`,
- `xsSerializable::GetLastChild`,
- `xsSerializable::GetSibbling`,

etc. In addition, various member functions of `wxXmlSerializer` class can be used. For more information about all available data handling functions supported by the wxXS library please see the reference documentation available at [8].

6 Extending The wxXmlSerializer

There are many built-in data types supported directly by the wxXS but aside of them, also custom data types can be processed by the library. For this case, the wxXS provides set of code macros and classes suitable for a creation and registration of user-defined I/O handlers.

In the following example a new I/O handler suitable for serialization/deserialization of `wxColourData` class is created and used.

First of all, the user must declare new I/O handler class and define code macros used for marking of a serialized data members. This should be done in appropriate header file. The declaration code can be as follows:

Example 4:

```
// Declaration of a class
// 'xsColourDataPropIO' encapsulating the
// custom  property I/O handler for
// 'wxColouData' data type.
XS_DECLARE_IO_HANDLER(wxColourData,
xsColourDataPropIO);

// Code macros which create new serialized
// wxColourData property
#define XS_SERIALIZE_COLOURDATA(x, name)
XS_SERIALIZE_PROPERTY(x, wxT("colourdata"),
name);

#define XS_SERIALIZE_COLOURDATA_EX(x, name,
def) XS_SERIALIZE_PROPERTY_EX(x,
wxT("colourdata"), name,
xsColourDataPropIO::ToString(def));
```

Let us discuss the code listed above in more details. The macro `XS_DECLARE_IO_HANDLERS` declares a new class called `xsColourDataPropIO` suitable for processing of data members with data type `wxColourData` (which is a class provided by the wxWidgets used for data transfer between an application and the color picker dialog). User-defined macros `XS_SERIALIZE_COLOURDATA` and `XS_SERIALIZE_COLOURDATA_EX` can be later used in the implementation code to mark `wxColourData` class members in the similar way as the `XS_SERIALIZE` macro was used in the previous examples. Note that the text string "*colourdata*" must be a unique identifier used for identification of this data type in serialized XML structure.

Now see the implementation code part:

```
// Define custom data I/O handler
XS_DEFINE_IO_HANDLER(wxColourData,
xsColourDataPropIO);

// Two following static member functions of
// the data handler class MUST
// be defined manualy:

// wxString xsPropIO::ToString(T value) ->
// creates a string representation of the
// given value:
wxString
xsColourDataPropIO::ToString(wxColourData
value)
{
    wxString out;

    out << xsColourPropIO::ToString(
           value.GetColour());

    for(int i = 0; i < 16; i++)
    {
        out << wxT("|") <<
             xsColourPropIO::ToString(value.
             GetCustomColour(i));
    }
    return out;
}

// T xsPropIO::FromString(const wxString&
// value) -> converts data from
// given string representation to its
// relevant value:
```

```
wxColourData
xsColourDataPropIO::FromString(const
wxString& value)
{
    wxColourData data;

    if(!value.IsEmpty())
    {
        int i = 0;
        wxStringTokenizer tokens(value,
wxT("|"), wxTOKEN_STRTOK);

        data.SetColour(xsColourPropIO::FromS
tring(tokens.GetNextToken()));

        while(tokens.HasMoreTokens())
        {
            data.SetCustomColour(i,
                xsColourPropIO::FromString(
                tokens.GetNextToken()));
            i++;
        }
    s}
    return data;
}
```

The most of the implementation effort is hidden in the `XS_DEFINE_IO_HANDLER` macro, the user must manually create only two static functions responsible for conversion of processed data value to its string representation and vice versa. These static functions are then internally used be core library classes for serialization and deserialization but they can be used also for any other purposes. For example, in the code above you can see similar static functions called `xsColourPropIO::FromString()` and `xsColourPropIO::ToString()` defined by built-in I/O handler designed for processing of `wxColour` data members.

The last step needed for proper initialization of the new I/O handler class is its registration. It should be done as soon as possible, typically in the application initialization code. For registration of the I/O handler, the `XS_REGISTER_IO_HANDLER` macro can be used in the following way:

```
// Register new property I/O handler
// 'xsColourDataPropIO' for a data type with
// name 'colourdata'.
XS_REGISTER_IO_HANDLER(wxT("colourdata"),
xsColourDataPropIO);
```

7 Conclusion

The aim of the paper was to introduce a new cross-platform software library `wxXmlSerializer` (wxXS) suitable for easy and elegant creation of serializable hierarchical data containers using the wxWidgets library and C++ programming language. The library is available as an open-source project and can be used for both commercial and open-source software projects.

The wxXS was already successfully used as a technological background for various projects such as the wxShapeFramework [7] cross-platform graphics library or the UML code generation tool called CodeDesigner developed at the Tomas Bata University.

The library features described in this document are only a tiny fraction of comprehensive functionality provided by the library. For more information about its usage and abilities please see the library reference or code examples available at the Source Forge web site (http://www.sourceforge.net/wxxs).

8 Acknowledgements

This work was supported by the Ministry of Education of the Czech Republic under grant No. MSM7088352102.

9 References

[1] J. Smart, K. Hock, S. Csomor, *Cross-Platform GUI Programming with wxWidgets*, Prentice Hall PTR, 2006

Internet sources:

[2] Java Data Objects (JDO) at Sun Developer Network (SDN), 2008: http://java.sun.com/jdo/http://java.sun.com/jdo/

[3] SQLite database home website, 2008: http://www.sqlite.org/

[4] Firebird database home website, 2008: http://www.firebirdsql.org/

[5] wxWidgets home website, 2008: http://www.wxwidgets.org/

[6] wxWidgets license documents, 2008: http://www.wxwidgets.org/about/newlicen.htm

[7] wxShapeFramework library website, 2008: http://sourceforge.net/projects/wxsf

[8] wxXmlSerializer library website, 2008: http://sourceforge.net/projects/wxxs