MusicXML Library Version 2

A toolbox to support the MusicXML format.

D.Fober, S.Letz, Y.Orlarey

{fober, letz, orlarey}@grame.fr

Grame - Research Lab.
Centre national de création musicale
FR - Lyon

May 2008

Summary

The MusicXML format

The MusicXML format represents common Western musical notation from the 17th century onwards. It is an xml format that organizes the music into a header followed by the core music data. The core music data may be organized as *partwise* or *timewise* data:

- *partwise* data are organized into parts containing measures,
- *timewise* data are organized into measures containing parts.

The music notation complexity is reflected by the significant number of MusicXML elements: 343 elements are defined by the version 2.0 of the format.

More details and DTDs on `http://www.recordare.com/`

Issues in the library design

The main issues in designing a C++ library to support the format are related to the significant the number of MusicXML elements.

1. cost of describing all the MusicXML elements,
2. design of an adequate and efficient memory representation,
3. avoiding additional complexity to the MusicXML format,
4. easiness to maintain and to update to new versions of the format.

The first version of the MusicXML library was quite good on points 2 and 3, but rather weak on points 1 and 4.

libmusicxml v.2: what's new?

- supports the MusicXML format version 2,
- easy to upgrade to new versions of the MusicXML format from the DTDs,
- adheres strictly to the MusicXML DTDs: each element has a corresponding C++ class,
- designed using a single homogeneous `xmlelement` class and automatic typing using templates,
- provides STL iterators to browse the memory representation,
- is not compatible with libmusicxml version 1.xx.

The main point is the simplified design: 4 classes instead of 150 to build a MusicXML memory representation.

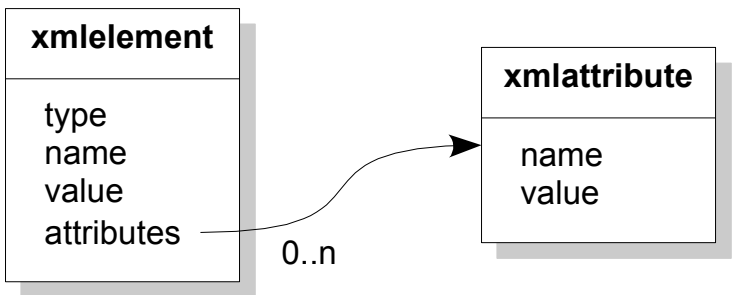
libmusicxml v.2: what remains unchanged?

- automatic memory management using smart pointers,
- support of the `visitor` mechanism,
- provides rolled and unrolled browsing,
- provides previous visitors (musicxml2guido, midivisitor, transposition...)

Memory representation

The MusicXML format is represented by:

- a single `xmlelement` class
- simple methods to query an element
- derived into as many types as MusicXML elements using templates
- organized into a tree



Memory representation

The MusicXML format is represented by:

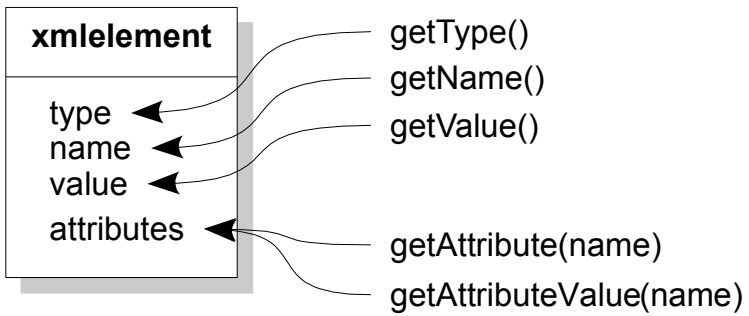
- a single `xmlelement` class
- simple methods to query an element
- derived into as many types as MusicXML elements using templates
- organized into a tree

homogeneous design leads to simplicity.

Memory representation

The MusicXML format is represented by:

- a single `xmlelement` class
- simple methods to query an element
- derived into as many types as MusicXML elements using templates
- organized into a tree



Memory representation

The MusicXML format is represented by:

- a single `xmlelement` class
- simple methods to query an element
- derived into as many types as MusicXML elements using templates
- organized into a tree

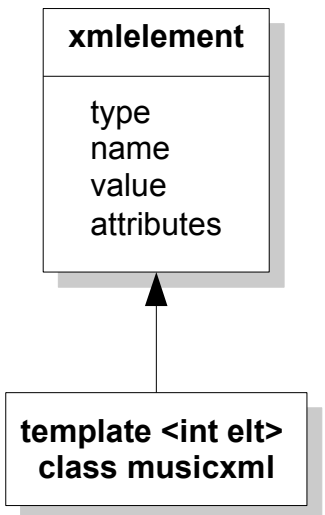
Makes the DTDs usable as the library documentation:
e.g.

```
measure->getAttributeValue("number")
```

Memory representation

The MusicXML format is represented by:

- a single `xmlelement` class
- simple methods to query an element
- derived into as many types as MusicXML elements using templates
- organized into a tree



Memory representation

The MusicXML format is represented by:

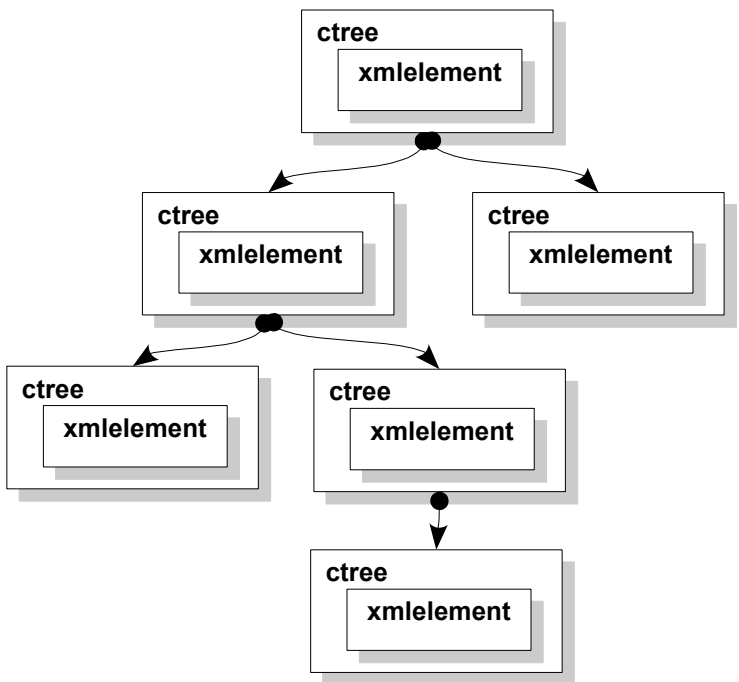
- a single `xmlelement` class
- simple methods to query an element
- derived into as many types as MusicXML elements using templates
- organized into a tree

Allows the visitor mechanism to operate

Memory representation

The MusicXML format is represented by:

- a single `xmlelement` class
- simple methods to query an element
- derived into as many types as MusicXML elements using templates
- organized into a tree



Memory representation

The MusicXML format is represented by:

- a single `xmlelement` class
- simple methods to query an element
- derived into as many types as MusicXML elements using templates
- organized into a tree

support STL iterators

MusicXML DTDs as documentation

- types are consistently derived from the MusicXML element names
- attributes can be retrieved using their MusicXML names
- browsing the memory representation is like reading the MusicXML file

<!ELEMENT part-name>
=> class: S_part_name
=> constant: k_part_name

<!ATTLIST measure
number CDATA #REQUIRED
...

```
measure->getAttributeValue("number")
measure->getAttributeIntValue("number",default)
```

- Elements and attributes names and values are available as strings but also support automatic conversion to numeric types.
- Supports xml comments and processing instruction as well.

MusicXML DTDs as documentation

- types are consistently derived from the MusicXML element names
- attributes can be retrieved using their MusicXML names
- browsing the memory representation is like reading the MusicXML file

<!ELEMENT part-name>
=> class: S_part_name
=> constant: k_part_name

<!ATTLIST measure
number CDATA #REQUIRED
...

```
measure->getAttributeValue("number")

measure->getAttributeIntValue("number",default)
```

- Elements and attributes names and values are available as strings but also support automatic conversion to numeric types.
- Supports xml comments and processing instruction as well.

MusicXML DTDs as documentation

- types are consistently derived from the MusicXML element names
- attributes can be retrieved using their MusicXML names
- browsing the memory representation is like reading the MusicXML file

<!ELEMENT part-name>
=> class: S_part_name
=> constant: k_part_name

<!ATTLIST measure
number CDATA #REQUIRED
...

```
measure->getAttributeValue("number")

measure->getAttributeIntValue("number",default)
```

- Elements and attributes names and values are available as strings but also support automatic conversion to numeric types.
- Supports xml comments and processing instruction as well.

Browsing the memory representation

- supports the acyclic visitor pattern
- supports STL iterators

Count using a visitor

```
class countnotes : public visitor<S_note>
{
    public:
    int fCount;

            countnotes() : fCount(0)    {}
    virtual ~countnotes() {}
    void visitStart( S_note& elt ) { fCount++; }
};
```

Count using iterators and STL

```
struct countnotes {
    bool operator () (const Sxmlelement elt) const
        return elt->getType() == k_note;
};

countnotes p;
int count = count_if(elt->begin(), elt->end(), p);
```

Browsing the memory representation

- supports the acyclic visitor pattern

Count using a visitor

```
class countnotes : public visitor<S_note>
{
    public:
    int fCount;

            countnotes() : fCount(0)    {}
    virtual ~countnotes() {}
    void visitStart( S_note& elt ) { fCount++; }
};
```

- supports STL iterators

Count using iterators and STL

```
struct countnotes {
    bool operator () (const Sxmlelement elt) const
        return elt->getType() == k_note;
};

countnotes p;
int count = count_if(elt->begin(), elt->end(), p);
```

Main files

Files, folders	Purpose
`xml.h, types.h, ctree.h`	MusicXML memory representation
`factory.h`	to generate MusicXML elements
`typedefs.h, elements.h`	types and constant definitions
`the visitors folder`	many visitors... usable as sample code as well

WARNING!

The following files are automatically generated by the DTDs analyser and should not be modified:

`elements.h, typedefs.h, factory.cpp`

Main files

Files, folders	Purpose
`xml.h, types.h, ctree.h`	MusicXML memory representation
`factory.h`	to generate MusicXML elements
`typedefs.h, elements.h`	types and constant definitions
`the visitors folder`	many visitors... usable as sample code as well

WARNING!

The following files are automatically generated by the DTDs analyser and should not be modified:

`elements.h, typedefs.h, factory.cpp`

DTDs Analysis

A fast way to update to new version of the MusicXML format.

The MusicXML DTDs are automatically analyzed to generate source code, types and constants.

'-' are replaced with '_' in MusicXML elements or attribute names to comply to the C/C++ identifiers lexical definition.

- a makefile and a shell script are used for analysis and generation
- templates are provided in the `template` folder
- generates types (`typedefs.h`), constants (`elements.h`) and source code (`factory.cpp`)

For Further Reading


MusicXML
The MusicXML home page.
http://www.recordare.com/xml.html


M. Good.
The virtual score, MusicXML for notation and analysis.
In W. B. Hewlett and E. Selfridge-Field, editors, *Computing in Musicology*, volume 12, pages 113–124. MIT Press, 2001.


A. Alexandrescu.
Modern C++ Design: Generic Programming and Design Patterns Applied.
Addison-Wesley, 2001.


E. Gamma, R. Helm, R. Johnson, and J. Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1995.