Document Object Model based XML Handling in ITK

Release 0.00

Ren-Hui Gong[1] and Ziv Yaniv[1]

September 21, 2012

[1]Sheikh Zayed Institute for Pediatric Surgical Innovation, Children's National Medical Center, USA

Abstract

The Insight Segmentation and Registration Toolkit (ITK) previously provided a framework for parsing Extensible Markup Language (XML) documents using the Simple API for XML (SAX) framework. While this programming model is memory efficient, it places most of the implementation burden on the user. We provide an implementation of the Document Object Model (DOM) framework for parsing XML documents. Using this model, user code is greatly simplified, shifting most of the implementation burden from the user to the framework. The provided implementation consists of two tiers. The lower level tier provides functionality for parsing XML documents and loading the tree structure into memory. It then allows the user to query and retrieve specific entries. The upper tier uses this functionality to provide an interface for mimicking a serialization and de-serialization mechanism for ITK objects. The implementation described in this document was incorporated into ITK as part of release 4.2.

Latest version available at the Insight Journal [http://hdl.handle.net/10380/3387]

Contents

1 Introduction

The Extensible Markup Language (XML) (http://www.w3.org/XML) has been widely used to perform tasks such as providing application settings, storing intermediate or final states to disk, transferring objects over the network, and so on.

Reading or writing an XML document can be realized using one of two common application programming interfaces (APIs): Simple API for XML (SAX) (http://www.saxproject.org), and Document Object Model (DOM) (http://www.w3.org/DOM). Each API has pros and cons. SAX-based XML reading/writing is more appropriate for reading large documents, while DOM-based XML reading/writing is much easier for the user to implement.

Previously the Insight Segmentation and Registration Toolkit (ITK) (www.itk.org), only provided a SAX-based XML reading/writing framework. To enhance the XML handling functionality in ITK, we implemented a DOM-based XML reading/writing framework. In addition we define an interface that allows the user to mimic serialization via XML, building upon the DOM functionality. This allows the user to define ITK objects using XML, easily writing an object description to disk or loading an object from disk directly into memory.

We next provide a short review of the SAX based XML framework in ITK. We then describe the new DOM based framework, and introduce its core and associated utility classes. Finally we provide examples showing how to use the framework both for reading and writing XML (tier one), and for mimicking serialization (tier two).

2 SAX-based XML handling

The SAX based XML processing model is event driven. In the case of reading, the XML document is fed into a SAX-based reader as a textual stream, and the reader instantly processes each piece of incoming information. That is, for each possible event (start tag, end tag, character read) the user is required to implement a callback function which will be invoked when the event occurs.

Thus, in order to read a specific XML document the user implements her own reader class that inherits from `itk::XMLReader<T>`, and implements three virtual functions that comprise the SAX interface:

```
virtual void StartElement(const char *name, const char **atts) = 0;
virtual void EndElement(const char *name) = 0;
virtual void CharacterDataHandler(const char *inData, int inLength) = 0;
```

From the reader's point of view, SAX-based XML reading is a *push* based approach with the reader waiting for events to occur, triggering the XML stream processing. This model makes efficient use of computer

tb

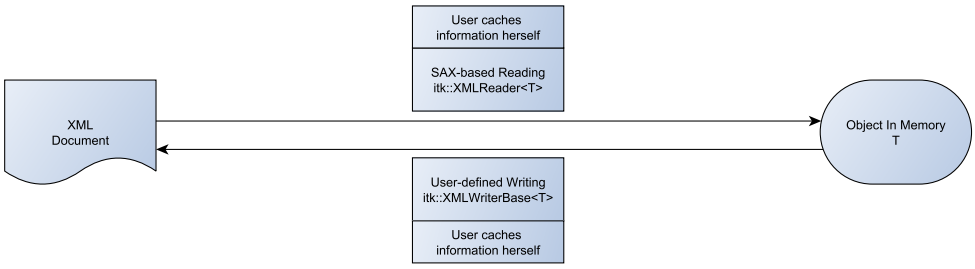

Figure 1: SAX-based XML reading/writing in ITK: user caches the required information herself until reading of the object is complete.

memory and is able to handle very large documents as there is no need to load the complete text into memory prior to processing it. However, the implementor of a SAX based reader has to accumulate all of the relevant pieces of information on their own, a user managed cache. If there are dependencies between data elements the logical validity of the document can only be confirmed after all of the processing has been completed.

This event based approach is similar for writing, with the user required to implement her own writer class, inheriting from `itk::XMLWriterBase<T>`, and implementing the virtual function required by the SAX API:

```
virtual int WriteFile() = 0;
```

In practice this means that the user needs to implement an XML writer from scratch for each new document, as the framework provides minimal automation. Figure 1 provides a schematic description of the information flow in the SAX based framework.

3 DOM-based XML handling

3.1 Basic use (Tier 1)

The DOM framework is based on the use of an intermediate tree data structure residing in memory which represents the complete XML document. In our implementation this data structure uses the `itk::DOMNode` and `itk::DOMTextNode` classes, with the root of the structure being a `DOMNode`.

To read an XML document into memory the user does not need to implement a thing. The XML document is parsed and loaded into memory using the `itk::DOMNodeXMLReader` class, with the user only required to set the file name and invoke the `Update()` method. Once all the data is in memory the user obtains the root of the tree and traverses the structure to obtain the desired information. This is a *pull* based approach with the user initiating the data collection from the data structure.

The approach to writing is similar, in this case the user needs to actively construct the document structure in memory. Once the structure is created they use the `itk::DOMNodeXMLWriter` to write the XML file. This only requires providing the root to the writer, setting a file name, and invoking the `Update()` method.

To the user, the main advantage of this framework as compared to the SAX one is that obtaining data values from the tree structure is much simpler than obtaining them using the callback mechanism. In addition the

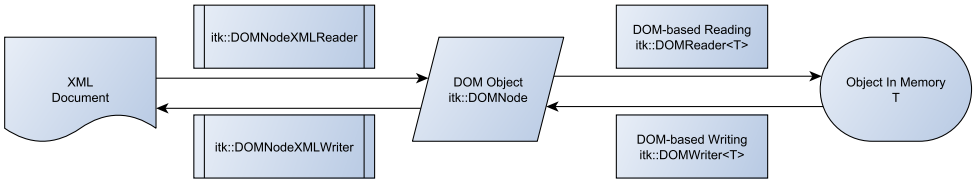

Figure 2: New DOM-based XML reading/writing in ITK 4.2+: all information is pre-cached in the intermediate DOM object.

resulting code can be better organized and is much more readable to developers, as data acquisition can be localized in a single method instead of distributed across a set of callbacks.

3.2 Mimicking serialization (Tier 2)

In many cases we would like to use XML to configure our programs. That is, we use XML to describe a specific object instance in memory (a limited form of serialization). This results in a recurring pattern, load the information from disk into the tree data structure and then set the values for the object instance variables by traversing the data structure.

When mimicking serialization in this manner, we can consolidate the two step process in a single class. In our implementation this is done by the abstract `itk::DOMReader<T>` class. To implement a specific object reader the user derives a class from `itk::DOMReader<T>` and implements a single function:

```
virtual void GenerateData(const DOMNode* inputdom, const void* userdata) = 0;
```

This function is responsible for creating the output object using the information in the given tree structure, and possibly some additional information. The later serves as supplemental information in cases where the XML does not contain all of the required data. This option is rarely used but is useful on occasion.

Object writing is similar. The user inherits from the abstract `itk::DOMWriter<T>` class, and implements one virtual function that constructs the intermediate tree structure from the object we want to serialize. Again, the user may supply additional information if it is required for writing the object to file. For example, if the object contains multiple images, the user will need to specify file names so that the images are written to the specific files and the XML file contains the relevant file names.

```
virtual void GenerateData(DOMNode* outputdom, const void* userdata) const = 0;
```

Figure 2 shows a schematic description of our DOM based serialization framework.

This serialization mimicking mechanism alleviates the need for the programmer to generate correct textual output per object type. As a result, the developer can focus on the simpler task of translating between the tree data structure and the specific object instantiation, either when reading or writing.

3.3 Core classes

The core classes used in the implementation of the DOM framework are shown in Figure 3.

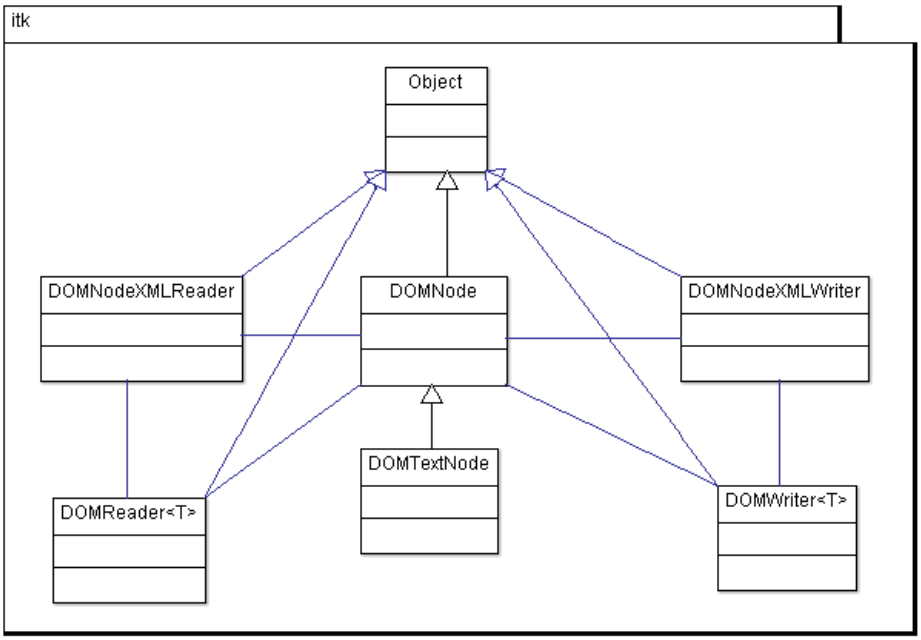

Figure 3: Core classes for DOM-based XML reading/writing in ITK.

The main class used to construct the tree data structure in memory is `itk::DOMNode`. Except for the root node, each node is associated with a parent node, a set of attributes, and a set of child nodes. The class provides a number of methods for setting and accessing the tag name, parent, attributes and children. Each attribute is represented using a `<Key,Value>` pair, and both key and value are text strings.

One special attribute with Key="id" is internally used to distinguish a node from its siblings. This means that the attribute "id" (all combinations of upper and lower case) should be used cautiously as we assume that the value of this attribute is unique among its siblings (i.e. nodes that share the same parent as this node) in an XML tree structure.

An `itk::DOMNode` in a DOM tree structure can be retrieved using one of the following methods:

1. By using the index among its immediate siblings;
2. By using the offset with respect to an immediate sibling;
3. By using the XML tag name and an optional index (the index is required when the parent has multiple children with the same tag name);
4. By using the optional "id" attribute; and
5. By using a path or query string that concatenates one or more of the above methods.

The class `itk::DOMTextNode` represent XML tag values that only contain one text string. It has no attributes and children, and can only be used as a leaf node. To retrieve an `itk::DOMTextNode`, all above-mentioned methods except for (4) can be used. In the case of (3), the special tag name "!" is used to represent this type of nodes.

Figure 4 shows an XML description and the corresponding data structure in memory.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<registration_result id="005">
    <description>
        CT atlas to X-ray registration.
    <description>
    <final_transform>
        <transform id="deformable">
            12.30 230.14 -11.85
        </transform>
        <transform id="rigid" rotation_center="0 0 20.45">
            0 0 0 -133.01 -4.71 0.79
        </transform>
    </final_transform>
</registration_result>
```

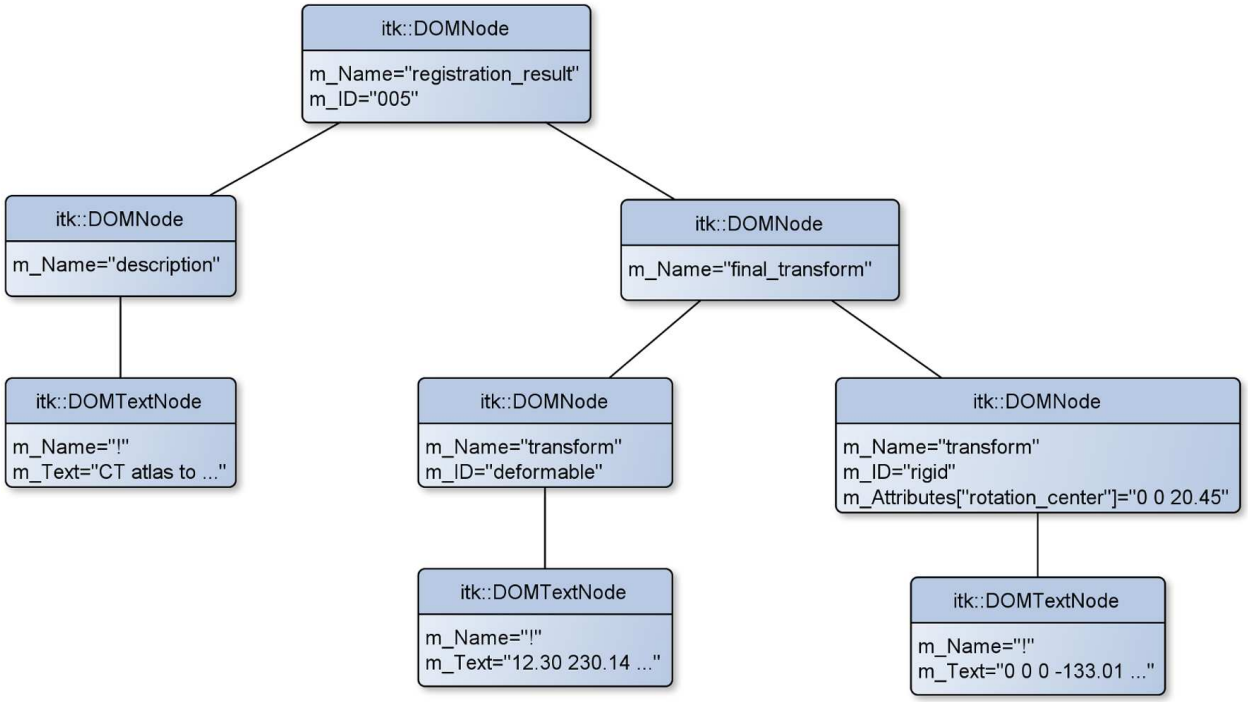

Figure 4: XML document and corresponding data structure in memory.

The classes itk::DOMNodeXMLReader and itk::DOMNodeXMLWriter are used to parse an XML document into a DOM object and to write a DOM object to an XML document on disk. These classes should be used when dealing with an XML document which describes several loosely related pieces of information. Once the structure is in memory the user extracts the information by traversing the tree nodes.

The classes itk::DOMReader<T> and itk::DOMWriter<T> are abstract classes from which the user derives their own specific readers/writers, mimicking serialization/de-serialization of a single object. The derived class simply encapsulates the traversal and extraction of the data from the internal DOM tree structure.

Note that the classes itk::DOMReader<T> and itk::DOMWriter<T> are derived from itk::Object instead of itk::ProcessObject, though they have similar member functions such as SetFileName(), Update(), and so on. This decision was made to allow reading and writing of objects with any type, instead of limiting it to itk::DataObject.

3.4 Utility classes

Reading/writing XML documents on disk and objects in memory involves a lot of string processing and string-based data input/output. To facilitate these operations, several utility classes are provided.

The class itk::StringTools provides operations to read/write primitive data, vectors and ITK arrays from/to strings. It also provides additional functions for string manipulation such as trimming, case conversion, splitting, sub-string testing, and so on. All operations in this class are static functions.

The class itk::FancyString inherits from the C++ std::string and adds all itk::StringTools operations as member functions. The streaming operators >> and << have been overloaded for this class such that it can be used as a string stream. This avoids explicitly creating an intermediate string stream from a string when it needs to input/output data from/to a string. In addition, a manipulator, itk::ClearContent, is provided to clear the content of such a string.

The class itk::FileTools provides two functions to create directories or files if they don't exist. The functions are based on existing ITK file manipulation tools (defined in SystemTools.hxx), and are provided to make sure that data files can be written to disk.

Although the above utility classes are provided to facilitate DOM-based XML handling, they are useful outside of this framework.

4 Examples

4.1 Tier 1 XML handling

In this example we demonstrate how to use the Tier 1 technique (Section 3.1) to read/write two user variables. The first variable is a string holding some descriptive text, and the second variable is a score of double type. The corresponding XML document is:

```
<my_settings>
    <desc>
        Some user notes go here.
    </desc>
    <score value="90"/>
</my_settings>
```

The following code snippet shows how to read the variables from such an XML file:

```
//////////////////////////////////////////////////////
// Tier 1 reading: first read a DOM object from the XML file,
// then extract the information of interest from the DOM object.
//////////////////////////////////////////////////////

// Step 1: read the DOM object from an XML file
itk::DOMNode::Pointer outputDOMObject;
const char* inputXMLFileName = ...
itk::DOMNodeXMLReader::Pointer reader = itk::DOMNodeXMLReader::New();
reader->SetFileName( inputXMLFileName );
reader->Update();
outputDOMObject = reader->GetOutput();

// Step 2: read the variables from the DOM object
if ( outputDOMObject->GetName() != "my_settings" )
  {
  throw "Unrecognized input XML document!";
  }
std::string desc = "";
  {
  itk::DOMNode* node = outputDOMObject->GetChild( "desc" );
  desc = node->GetTextChild()->GetText();
  }
double score = 0.0;
  {
  itk::DOMNode* node = outputDOMObject->GetChild( "score" );
  itk::FancyString fs = node->GetAttribute( "value" );
  fs >> score;
  }
```

The following code snippet shows how to write the variables to an XML file:

```
//////////////////////////////////////////////////////
// Tier 1 writing: first write the information of interest to a DOM object,
// then write the DOM object to an XML file.
//////////////////////////////////////////////////////

// Step 1: write the variables to a DOM object
itk::DOMNode::Pointer inputDOMObject = itk::DOMNode::New();
inputDOMObject->SetName( "my_settings" );
std::string desc = ...
  {
  // create a node and add it to the DOM object
  itk::DOMNode::Pointer node = itk::DOMNode::New();
  node->SetName( "desc" );
  inputDOMObject->AddChildAtEnd( node );
  // add a text child to the newly created node
  node->AddTextChild( desc );
  }
double score = ...
  {
  // create a node and add it to the DOM object
```

```
  itk::DOMNode::Pointer node = itk::DOMNode::New();
  node->SetName( "score" );
  inputDOMObject->AddChildAtEnd( node );
  // add an attribute to the newly created node
  itk::FancyString fs;
  fs << score;
  node->SetAttribute( "value", fs );
  }

// Step 2: write the DOM object to an XML file
const char* outputXMLFileName = ...
itk::DOMNodeXMLWriter::Pointer writer = itk::DOMNodeXMLWriter::New();
writer->SetInput( inputDOMObject );
writer->SetFileName( outputXMLFileName );
writer->Update();
```

4.2 Tier 2 XML handling

In this example we demonstrate how to use the Tier 2 technique (Section 3.2) to read/write a simple testing object named `DOMTestObject`. It contains a single member variable `m_FooValue` of type “float” as well as the corresponding set/get methods. The XML document describing this object is:

```
<DOMTestObject>
    <foo value="123.45"/>
</DOMTestObject>
```

The following code example demonstrates the reading of such an object. It first implements a DOM-based reader that reads an XML file and produces the corresponding test object, then uses the implemented reader to perform object reading in a user program.

```
////////////////////////////////////////////////////////////////
// File DOMTestObjectDOMReader.h
////////////////////////////////////////////////////////////////

// The reader derives from itk::DOMReader<T> and, except for the common definitions
// required by itk::Object, it needs only to implement one protected virtual function, GenerateData.

#include <itkDOMReader.h>
#include "DOMTestObject.h"

class DOMTestObjectDOMReader : public itk::DOMReader<DOMTestObject>
{
// Common definitions for itk::Object go here.
...
protected:
  virtual void GenerateData( const DOMNodeType* inputdom, const void* )
  {
    // First check whether the user has supplied a correct XML document.
    if ( inputdom->GetName() != "DOMTestObject" )
      {
      itkExceptionMacro( "tag name DOMTestObject is expected" );
      }
```

```
    // The user may have already provided an instance of the test object
    // as the output. If so, retrieve this object for subsequent reading.
    OutputType* output = this->GetOutput();

    // If the user hasn't provided an instance as the output, create one.
    if ( output == NULL )
      {
      OutputType::Pointer object = OutputType::New();
      output = (OutputType*)object;
      this->SetOutput( output );
      }

    // We will use the itk::FancyString to facilitate data reading from string.
    itk::FancyString s;

    // Retrieve the child node with the tag name "foo".
    const DOMNodeType* node = inputdom->GetChild( "foo" );
    if ( node == NULL )
      {
      itkExceptionMacro( "Child foo not found!" );
      }

    // Now retrieve the value of the attribute "value", which is a text string,
    // and convert it to type float.
    float fooValue = 0;
    s = node->GetAttribute( "value" );
    s >> fooValue;

    // Finally assign the obtained value to the output object.
    output->SetFooValue( fooValue );
  }
};

//////////////////////////////////////////////////////////////////////
// File main.cpp
//////////////////////////////////////////////////////////////////////

// This user program reads a test object from an XML file using the reader
// described above, and then performs subsequent processing.

#include "DOMTestObjectDOMReader.h"

int main( int argc, char* argv[] )
{
  // Variable to store the output test object.
  DOMTestObject::Pointer outputObject;

  // Read the object from an XML file.
  const char* inputXMLFileName = ...
  DOMTestObjectDOMReader::Pointer reader = DOMTestObjectDOMReader::New();
  reader->SetFileName( inputXMLFileName );
  reader->Update();
  outputObject = reader->GetOutput();
```

```
  // Perform subsequent processing on the output object.
  ...
}
```

The following code example demonstrates the writing of a test object. It first implements a DOM-based writer that accepts an input test object and writes the corresponding XML document to a disk file, then uses the implemented writer to perform object writing in a user program.

```
////////////////////////////////////////////////////////////////////
// File DOMTestObjectDOMWriter.h
////////////////////////////////////////////////////////////////////

// The writer derives from itk::DOMWriter<T> and, except for the common definitions
// required by itk::Object, it needs only to implement one protected virtual function, GenerateData.

#include <itkDOMWriter.h>
#include "DOMTestObject.h"

class DOMTestObjectDOMWriter : public itk::DOMWriter<DOMTestObject>
{
// Common definitions for itk::Object go here.
...
protected:
  virtual void GenerateData( DOMNodeType* outputdom, const void* ) const
  {
    // First set the tag name for the intermediate DOM object.
    outputdom->SetName( "DOMTestObject" );

    // Retrieve the test object to be written out.
    const InputType* input = this->GetInput();

    // We will use the itk::FancyString to facilitate data writing to string.
    itk::FancyString s;

    // Create a child node with the tag name "foo", and add it to the DOM object.
    DOMNodePointer node = DOMNodeType::New();
    node->SetName( "foo" );
    outputdom->AddChild( node );

    // Finally retrieve the foo value from the input test object, convert it
    // to a string, and set the value for the attribute "value" in the newly created
    // child node.
    float fooValue = input->GetFooValue();
    s << fooValue;
    node->SetAttribute( "value", s );
  }
};

////////////////////////////////////////////////////////////////////
// File main.cpp
////////////////////////////////////////////////////////////////////
```

```
// This user program produces a test object, and then write it to an
// XML file using the writer described above.

#include "DOMTestObjectDOMWriter.h"

int main( int argc, char* argv[] )
{
  // Generate the test object.
  DOMTestObject::Pointer inputObject;
  ...

  // Write the test object to an XML file.
  const char* outputXMLFileName = ...
  DOMTestObjectDOMWriter::Pointer writer = DOMTestObjectDOMWriter::New();
  writer->SetInput( inputObject );
  writer->SetFileName( outputXMLFileName );
  writer->Update();

  return EXIT_SUCCESS;
}
```

4.3 Use of utility classes

The previous examples demonstrated some capabilities of the `itk::FancyString` class. Here we provide more examples to show its data I/O functionality:

```
itk::FancyString fs;

// Write a fundamental C data to the string.
int i = ...
fs << itk::ClearContent << i;
// Read a fundamental C data from the string.
fs >> i;

// Write a vector to the string.
std::vector<float> v = ...
fs << itk::ClearContent << v;
// Read all elements in the string to a vector.
fs >> v;
// Read a specified number of elements from the string to a vector.
v.resize( 3 );
fs.ToData( v );

// Write an ITK array to the string.
itk::Array<double> a = ...
fs << itk::ClearContent << a;
// Read all elements in the string to an ITK array.
fs >> a;
// Read a specified number of elements from the string to an ITK array.
a.SetSize( 5 );
fs.ToData( a );
```

The following example demonstrates the use of `itk::FileTools`. The file to be written is located in a directory that does not exist, and the `CreateFile()` function is called to create the directory as well as the file. This pre-processing is necessary to make sure that subsequent writing with the `std::ofstream` will be successful.

```
// We want to write some data to this file, which is located in a directory that does
// not exist.
const char* fn = ...

// Create the directory as well as the file.
itk::FileTools::CreateFile( fn );

// Open the file for writing.
std::ofstream ofs( fn );
if ( !ofs.is_open() )
  {
  itkExceptionMacro( "Cannot write file!" );
  }

// Write the data to the file.
...
```

5 Conclusion

We have provided an overview of the DOM-based XML framework introduced in ITK version 4.2. This framework provides a simpler approach to loading XML data than previously provided by ITK. In addition the resulting code is often cleaner, more understandable to developers, and thus easier to maintain.

6 Acknowledgement

This work was supported by NLM/NIH contract HHSN276201000578P.

A DOM Framework Files

The following listing shows the directories and files associated with the framework.

```
<ITKDIR>\Modules\IO\XML\
    include\
        itkDOMNode.h
        itkDOMNodeXMLReader.h
        itkDOMNodeXMLWriter.h
        itkDOMReader.h
        itkDOMReader.hxx
        itkDOMTextNode.h
        itkDOMWriter.h
        itkDOMWriter.hxx
        itkFancyString.h
        itkFancyString.hxx
        itkFileTools.h
        itkStringTools.h
        itkStringTools.hxx
    src\
        itkDOMNode.cxx
        itkDOMNodeXMLReader.cxx
        itkDOMNodeXMLWriter.cxx
        itkFancyString.cxx
        itkStringTools.cxx
    test\
        CMakeLists.txt
        itkDOMTest1.cxx
        itkDOMTest2.cxx
        itkDOMTest3.cxx
        itkDOMTest4.cxx
        itkDOMTest5.cxx
        itkDOMTest6.cxx
        itkDOMTest7.cxx
        itkDOMTest8.cxx
        itkDOMTestObject.h
        itkDOMTestObjectDOMReader.h
        itkDOMTestObjectDOMWriter.h

<ITKDIR>\Examples\IO\XML\
    CMakeLists.txt
    DOMFindDemo.cxx
    itkParticleSwarmOptimizerDOMReader.cxx
    itkParticleSwarmOptimizerDOMReader.h
    itkParticleSwarmOptimizerDOMWriter.cxx
    itkParticleSwarmOptimizerDOMWriter.h
    itkParticleSwarmOptimizerSAXReader.cxx
    itkParticleSwarmOptimizerSAXReader.h
    itkParticleSwarmOptimizerSAXWriter.cxx
    itkParticleSwarmOptimizerSAXWriter.h
    ParticleSwarmOptimizerReadWrite.cxx
```