XSIL: Java/XML for Scientific Data

Roy Williams

California Institute of Technology

6/27/00

Abstract: This paper describes the XSIL (Extensible Scientific Interchange Language) system that connects scientific XML to a Java object model and powerful visualization. There is an XML format for scientific data objects, and a corresponding Java object model, so that XML files can be read, transformed, and visualized. We describe the Xlook object browser, which reads XSIL files and allows them to be visualized in tree-like way, with viewers including a chart widget and a sortable table. We describe how to customize the XML format, and build custom viewing components that are dynamically loaded by Xlook. All code is open-source, obtainable from `http://www.cacr.caltech.edu/XSIL`.

1. Introduction

A. What is this?

1. XML is a “Language to make languages”, a simple, yet effective way to build structured documents that is revolutionizing business practices on the Internet. An XML document is a hierarchy of “elements”, each of which has a textual “tag”, some “attributes”, some text, and may also contain other elements.

2. XSIL is some basic syntactic structure for scientific data (Table, Array, Param, Time, etc), together with a mechanism to extend both the XML side and the Java object side. XSIL also offers a modular, extensible viewing platform called Xlook.

3. An XML document may refer to a DTD (document type definition) that expresses its structure. The DTD is analogous to the Class of object-oriented software, and the XML document is an instance of that DTD in the same way that the Object is an instance of a Class, or an Apple is an instance of Fruit.

4. For example:

```
<Book ISBN=”0439139597”>
    <Author>D. Scarlatti</Author>
    <Title>The Big Fight: Harpsichord vs. Violin</Title>
    <Author>A. Stradivarius
        <Email>strad@violin.org</Email>
    </Author>
</Book>
```

5. This is a record that expresses structured information about a book. The DTD for such an XML scheme defines syntactic information, such as the idea that a book has one or more authors, but exactly one title; that an author may have zero or more email addresses, but a title cannot have an email address. An example of an *attribute* is the ISBN attribute associated with the book element. When an XML file is parsed (by any XML application, not just XSIL), the syntax is compared against the DTD, and non-compliant files detected. An XML editor is powerful because it ‘knows’ the document structure through the DTD: once a title has been entered for the book record, the editor will not allow another title.

6. The XSIL DTD defines a few basic elements that form the core of scientific data. The Table is modelled on the relational table and expresses a collection of records, all of similar structure; the Array is for low-dimensional data cubes such a images, spectra, time-series, voxels, and so on. The Stream element provides a link to external and encoded data through files, URL’s, Big/Little-endian and Base64 encodings, as well as delimited text.

7. The idea is that XSIL can be fashioned to meet the needs of a particular scientific discipline by building combinations of these base elements that have semantic meaning in that discipline, then extending XSIL appropriately to handle these. Dynamic loading ensures flexibility without relinking code or struggling with makefiles. The easiest way to extend XSIL does not entail DTD changes, but use of a special XML attribute `Type=”MyStuff.MyObject”`.

8. Such extensions can also be used to build viewing extensions for custom objects, extensions that can use all the

power that comes with Java, including the Swing GUI components, advanced imaging, Java3D, the LiveTable component with direct database access, and the JChart component for graphs.

9. XSIL is a pure-Java application, so that portability and programming efficiency have been emphasized over CPU speed. However, it can deal effectively with very large quantities of data, using the following three strategies.

• First, the use of external files separates metadata (XML) from data (binary). Very large XML files are not efficient, and most XML software cannot handle more than a few megabytes. However, if a user wishes to create very large XML files, there is a utility with the XSIL distribution (Splitter) to break up such a file into a small metadata file, in XML, and many external data files.

• Second, data is not fetched from external files until it is needed, so that an XSIL file can contain hundreds of links to external data, but the parsing can still be fast.

• Third, XSIL stores data efficiently, so each 4-byte integer takes up only 4 bytes asymptotically. When data is fetched from a link, efficient calls are available for bulk transfer.

B. More information on XML and XSIL can be found at:

• XSIL web site: `http://www.cacr.caltech.edu/XSIL`

• The World Wide Web Consortium standardization effort for XML: `http://www.w3c.org/XML`

• Seybold publishing XML site: `http://www.xml.com/`

C. Reasons to use XSIL

1. XSIL provides an XML model for data transfer which is being adopted in the gravitational-wave and astronomical communities. Furthermore, it is XML, with an increasing range of transformation filters, making it a very plastic language. There is a **named, typed hierarchy** through the collection object, and base objects are **Table, Array, Param, Time,** and others, and the arrays and tables may be in these primitive types: boolean, byte, short, int, long, float, double, floatComplex, doubleComplex, and String. There is a Stream object to extract byte streams and primitive streams from files, URLs, databases, and delimited text.

2. An **object model** with Java API. Users can provide a file name or URL to the XSIL library, which returns an object which is the root of a tree. This generic container (class XSIL) may have been extended to form one of the objects described below (eg. Time, Table, Array, Param, etc). The structure of an XSIL file is defined by its DTD, saying, for example, that each `Array` must have at least one `Dim` element to give its dimensions. This model may be extended by adding domain-specific tags and the corresponding code to XSIL. New entries would be added to the DTD to define the syntax of the new structure.

3. The object model is extensible in a different, looser, sense: if the XML file has a tag such as `<XSIL Type="Banana">`, then the XSIL reader will seach for a class called `Banana` in the place where user extensions are built, for `class Banana extends XSIL`. In this way we establish a highly flexible connection between the XML file and the code that handles it. If such code is not found, then the XML element is assumed to be a generic collection object. This method of extension does not entail changing the DTD. Several extensions are in the XSIL distribution, an example being `TimeSeries`. This element consists of a one-dimensional array and a pair of parameters that define the start time and the delta-time between samples. Because of the dynamic linking, object handlers (not just objects) may be emailed to colleagues.

4. One major code written to the XSIL library is the **object browser (Xlook)**, that is intended as a way to view XSIL files and their contents in the same way as a web browser can be used to view collections of files. There is a tree-representation of the elements of the file, shown alongside the XML itself. When an object is “Viewed”, then its specific viewer is displayed.

5. The object browser **can be extended with user-written code.** An example of this is the Java3D extension to XSIL: certain arrays are defined in XSIL, and an extension defined (as in (2) above). This extension treats these arrays as coordinates, colors, etc, and can display them in a Java3D window. Similarly the TimeSeries can be graphed.

D. Uses for XSIL

1. As a flexible and general transport format between disparate applications in a distributed archiving and

computing system; a text-based object serialization that can be handled by common tools, or

2. As a documentation mechanism for collections of data resulting from experiments or simulations; with all the parameters, structure, filenames and other information needed to keep a complete scientific record.

3. As an “ultra-light” data format: a user can, if he wants, simply read the markup, then delete all except the actual data, or all except for the filenames where the data may be found.

4. All XML files including XSIL, can be created and edited with a large range of high-quality, customizable tools.

5. XML files can be stored in an XML database, such as eXcelon (www.exceloncorp.com). Such products provide rich tools for storing, querying, presenting, and connecting to XML data.

E. Future Extensions

1. Data access: In addition to local files, URL’s, and XML-encoded data, we would like to work with databases directly, using JDBC to view and edit database tables.

2. Output formats: In addition to the Matlab output mechanism, we would like to consider data output as IDL, multichannel Photoshop, FITS, Ligo Frame files, and Microsoft Excel.

3. Viewers:a viewer for Ligo Frame files will be connected, and for astronomical applications, a FITS viewer also .

4. Service definitions: objects that tell a client application how to format and send requests to a remote service, and the kinds of response that the service will provide. Such a definition could contain an XHTML form both to provide a human input, but also to define the syntax of the request that the service is expecting.

2. XML Basics

A. Syntax

1. An XML file is a hierarchical structure of elements that may contain other elements. An *element* generally consists of a *start tag*, a *body*, and an *end tag*, for example: `<Fruit>Banana</Fruit>`, where start and end tags are distinguished by the presence of a slash.

a) An element may be ***empty***, meaning that there is only a single tag, with no body, for example `<EmptyEle-ment/>`; note the position of the slash.

b) Elements may contain ***attributes***, for example: `<Fruit color=”yellow”>`. There can only be one instance of a given attribute name in any tag.

2. XML is case-sensitive, so that `<Apple>`, `<apple>` and `<APPLE>` are all different tags.

3. Comments in XML are delimited like this:

```
<!-- This is a comment -->
```

4. An XML document can be handled by both human and computer. If a human sees the document, then the XML is *presented*, usually by means of a style language file, or through a filter. If a computer sees the document, there is an API to control a parser of the document.

3. XSIL files

A. Header

1. All XML files, which includes all XSIL files, have the following as their first line:

```
<?xml version="1.0"?>
```

2. The second line of an XML file may make a reference to an externally defined Document Type Definition (DTD), which defines the syntax of XSIL files. Thus every XSIL file has the following as its second line one of these:

```
<!DOCTYPE XSIL SYSTEM "http://www.cacr.caltech.edu/projects/xsil/xsil.dtd">
<!DOCTYPE XSIL SYSTEM “XSIL.dtd">
```

The DTD can be referenced from a remote location via the URL syntax, or locally with the file name. This file is presented in section 8.

3. There may follow other XML elements, but the XSIL parser will ignore these until it finds a XSIL element. Only the first XSIL element is then considered, so there should never be more than one in a file. Thus, in general, the third line is:

```
<XSIL>
```

and the last line of the file closes this element:

```
        </XSIL>
```

4. When contained in this first, plain container element, containers may be typed, leading to user-side code being called. Currently,the Type of the outermost element is not handled properly.

B. Object Name and Type

1. Any of the XSIL elements may have a Name attribute, for example:

```
        <Table Name="Bake_Out_Temperature">
```

While the names may be useful in presentation of the XSIL, the major intended use is in navigating the object model, through methods that find an object of a given name.

2. Any of the XSIL elements may have a Type attribute, which may also be used for navigating the object model. However, we are most interested in this attribute to link this XML element to a Java object, thus extending the container object:

```
        <XSIL Type="MyStuff.MyObject" Name="Jack">
```

causes the linker to look for the file $XSIL_HOME/extensions/MyStuff/MyObject.class to act as a handler for the object called `Jack`.

C. The Container Object

1. **XSIL**

There is a generic `<XSIL>` element which is a container for other elements, including other `<XSIL>` elements, thus inducing a hierarchy. Each of these may have a `Name` attribute, to provide hierarchical naming that is visible from the API. When the file is parsed, there is a representation of the first XSIL element that is found in the document. Here is a XSIL fragment that consists of a container hierarchy with an array at the second level and a parameter at the third level:

```
    <XSIL Name="Fruit">
        <XSIL Name="YellowFruit">
            <Array><Dim>7</Dim></Array>
            <XSIL Name="Banana">
                <Param Name="Inductance">1.34</Param>
            </XSIL>
        </XSIL>
    </XSIL>
```

2. The XSIL object is the only one that can contain other XSIL objects. All the others (below) can only contain only:

a) The elements explicitly defined in their description, or

b) Stream object from which data may be requested by the code handling the object.

D. XSIL Base Objects

1. **Comment**

A comment object in XSIL is not meant to be interpreted or parsed. It appears only in the presentation of the document. For example:

```
    <Comment>The data that follows is probably wrong</Comment>
```

2. **Param**

A parameter in XSIL is an association between a name and a value. In addition, it may have a Unit attribute. For example:

```
    <Param Name="Fruit_Mass" Unit="kg">0.387</Param>
```

The meaning here is "Fruit_mass = 0.387 kg", usually found in "parameter files" or “header files” in scientific computing. If the element is empty, then the value is read from the corresponding stream. Note that the value of the parameter is not typed as float, int, etc, but rather it is string-valued.

3. **Time**

In the LIGO observatory, as with many other experiments in physical science, it is critical that timing information be not only accurate, but also easy to understand. The `<Time>` element in XSIL can represent either “natural” time (ISO-8601 standard, YYYY-MM-DD HH:MM:SS.mmmuuunnn), or GPS time, or “Unix time” (seconds since 1/1/1970). The different formats are differentiated by the `Type` attribute in the tag:

```
    <Time Type="ISO-8601">1998-11-08 17:40:00.032</Time>
    <Time Type="GPS">594582000.032</Time>
    <Time Type=”Unix”>910546800.032</Time>
```

The default for the type is ISO-8601.

4. **Table**

A table is an unordered set of *records*, each of the same format, where a record is an ordered list of values. The contents of a record are defined by column headings, each of which may have a unit and a type. This definition of a table should be thought of as similar to the table object that is found in a relational database; we should point out that this is *not* the complex and exotic typographical beast of TeX or HTML.

The only tag specific to the Table object is `<Column>`, which specifies the name, type, and possibly units associated with one of the columns of the table. It can be thought of as the heading of a column in a table. Shown below is an example table that includes a stream from which the table can be populated.

```
    <Table>
        <Column Name="ChannelName" Type="string"/>
        <Column Name="Site"        Type="float"    Unit="meter"/>
        <Column Name="Clock"       Type="float"    Unit="hour"/>
        <Column Name="Description" Type="string"/>
        <Stream Delimiter=”,”>
            “History Channel”, 405.0, 3.7, “Another Channel”
            “Math Channel”, 307.0, 2.1, “Channel about Math”
        </Stream>
    </Table>
```

5. **Array**

An array is a collection of numbers (or other primitive type) referenced by subscripts, which is a list of integers whose maximum values are given by the list of dimensions of the array. This definition is very close conceptually to a Fortran or C array, with the `Type` attribute of the `<Array>` tag specifying which primitive type is contained in the array (float, int, etc.). The Dim element specifies the dimensions of the array, but it does not specify the subscript ranges. For a dimension of 5, a Fortran binding of the API would label subscripts from 1 to 5, but a Java or C++ binding would have subscripts from 0 to 4.

As with other XSIL objects, the Array tag may have `Name` and `Type` attributes. The Type attribute is interpreted as the data-type for the data in the aray. The only element specific to this class is `<Dim>`, which may have a Name attribute to specify the name of that dimension of the data for presentation purposes. For example:

```
    <Array Type=”int”>
        <Dim Name=”X-axis”>5</Dim>
        <Dim Name=”Y-axis”>3</Dim>
    </Array>
```

which specifies a 5x3 array of integers, with the last dimension changing fastest. The presumption is that 15 integers may be read from the Stream associated with this Array.

6. Url

This tag is quite similar to an HTML link:

```
    <Url href="http://www.cacr.caltech.edu/XSIL">XSIL Home Page</Url>
    <Url href="http://www.caltech.edu/">Caltech Home Page</Url>
```

There is an `href` attribute, containing the URL, and the text part is a message associated with the link.

4. Streams

A. Connecting Streams and XSIL Objects

1. Some of the objects from the previous section are self-contained, for example Param and Comment. Others (XSIL, Table, Array) define the structure of a data object, without necessarily making available the data itself. The Stream object provides the input that allows these structures to be filled.

2. A XSIL document may define many streams and many XSIL objects (Table, Array etc.). In general a stream may contain the data for many different objects. We use the collection mechanism induced by the XSIL tag to provide this association.

3. For a stream to be available to a XSIL object, either:

- The object contains the stream explicitly between its start and end tags, or
- the collection in which the object resides must contain exactly one Stream. The objects in the collection will then be read sequentially from the Stream.

4. Delimited Text

```
<Stream Type="Local" Delimiter=",">
   4.76,5.77,8.99,3.44,2.11,0.93
</Stream>
```

The delimiter string are additions to the default delimiter string, consisting of the newline character. Thus a string read in this way cannot contain a newline, and the String primitive in XSIL cannot contain a newline.

5. Remote data

```
<Stream Type="Remote">
   datafile1.dat
</Stream>
```

The stream may have Type `Local` — the data is contained in the XSIL file itself — or it may be `Remote`, as in this case. Local is the default type for a stream. Data is read from the local file system. The name is assumed to be relative to the XSIL filename, not relative to the current working directory. Thus XSIL files and their data can be packaged and moved without renaming files.

If the file name is fully qualified, it can be either in Unix (`/blah/blah`) or Windows style (`C:\blah\blah`).

6. Endian encoding

```
<Stream Type="Remote" Encoding="Littleendian">
   datafile1.dat
</Stream>
```

Java reads and writes bigendian binary, as this is the same byte-ordering as Sun computers have. Windows machines, though have the opposite ordering, so binary files made by a PC will need to be converted from little-endian format as above.

7. URL streams

```
<Stream Type="Remote">
   http://www.cacr.caltech.edu/blahblah/datafile1.dat
</Stream>
```

If the stream has type `Remote`, and the text in this element contains the string "`://`", then it is interpreted as a URL-type locator for the data that this stream represents. The file:// and the ftp:// and http:// protocols are supported.

8. Base64 streams

```
<Stream Encoding="Base64">
   AAAAAAAAAAEAAAACAAAAAwAAAAQAAAAFAAAABgAAAACAAAAIAAAACQ==
</Stream>
```

The first ten integers have been encoded in Base64 and put directly into the XML document.

B. Stream Content

1. The XSIL data stream can be considered to be an arbitrary sequence of bytes. In the ImageList demo (section 7C), streams get the image data, to be delivered as gif. Some of the images are external files, others are base64 encoded as part of the XSIL file.

2. In the ImageList code, a Java `InputStream` is obtained from the `<Stream>` tag in the XML. The stream of bytes is read in and converted to `java.awt.image` and displayed with a chooser. The metadata for each image is also shown.

3. In other cases, however, for example the Array and Table implementations, the Stream contains something higher level than raw bytes: a sequence of primitive types.

4. The types that are available for Tables and Arrays are listed in the following (with the corresponding number of bits), and some alternate spellings:

- `boolean` (1)
- `byte` (8)
- `short` (16) (int_2s)
- `int` (32) (int_4s)
- `long` (64) (int_8s)
- `float` (32) (real_4)
- `double` (64) (real_8)
- `floatComplex` (64) (complex_8)
- `doubleComplex` (128) (complex_16)

• `String` (arbitrary length) (lstring, char, character)

5. To read such a stream, there may be some filtering process. A base64 encoded stream is converted to binary byte stream, available as a java.io.InputStream. If the stream is to be read as a sequence of primitives, it is converted to java.io.DataInputStream, and the byte swapping necessary for little-endian conversion may then be applied.

6. If the stream is text-based, either local or remote, the newline character is always a delimiter, plus any other characters nominated in the Delimiter attribute of the Stream element. The text is read line by line from the source, and primitives generated. If there is data that cannot be converted to the relevant type (eg. the number 3.56A7464), then an exception is **not** thrown, but rather a default value is put in place.

7. The array of primitive objects contains strings, it cannot be read from a binary stream. Strings must be handled as delimited text.

8. Any primitive type can be converted to any other: to convert from complex we take the real part, to convert to boolean we ask if it is nonzero, to convert to String we use the toString() method.

C. Stream Encoding

1. The `Encoding` attribute specifies how the data in a `Stream` is encoded. It is a comma-separated list chosen from the values:

• `Text, Binary, base64, BigEndian, LittleEndian, Delimiter`

2. The Text attribute is assumed by default for Local streams, the Binary attribute is the default for Remote streams.

3. When the stream is Base64, it is decoded by the XSIL library. When the stream specifies an Endian order, it is converted to the Endian order appropriate to the current machine before being delivered to the application code.

4. The other attribute defined in this section is `Delimiter`: It is only relevant in the case of a Text stream. The characters in the attribute are appended to a delimiter string that already contains newline.

5. Extending XSIL

A. Object Model of XSIL

1. When a XML file is read by an application program through the XSIL API, the hierarchical structure of the file is parsed to a hierarchical *base object*, which is then made available to the application. The XSIL software layer then extracts from the base object the first `<XSIL>` element, which is returned to the application. Thus XSIL can be mixed in a straightforward way with other kinds of XML or HTML, such as Math Markup Language, Chemistry Markup, or other XML languages.

2. As well as the parser, the XSIL API provides a rich set of methods to extract objects of given element-type, objects which have given attributes, given Name or Type, and so on.

3. Information can then be extracted from objects. For example once a Param object of given name has been found, the string which is its value can then be extracted. In this way we have a dictionary of name-value pairs. Similarly the rows and colums of a Table, or the start and end time of a TimeSeries object are available to the application.

4. We can extend XSIL in an informal, perhaps personal way, simply by creating a collection object with “well-known” parameter names. In this case, we expect the application that is reading the file to understand the special significance of the word `Type=”TimeSeries”`, and to know the names of the parameters that it expects to find.

B. Example: TimeSeries

1. The following example illustrates the collection of parameter names that would be appropriate for a time-series object. To make a valid TimeSeries there must be (in addition to data), Start time (t0), and delta time between samples (dt). Given these, and the number of samples (from the dimensionality of the array), the third parameter can be computed.

```
<XSIL Type=”TimeSeries.TimeSeries” Name=”My Time Series”>
    <Comment>A sample time series</Comment>
    <Param Name=”t0”>6.0</Param>
    <Param Name=”dt”>0.001</Param>
    <Array>
        <Dim>1000</Dim>
        <Stream Type=”Remote” Encoding=”LittleEndian”>
            mydata.dat
        </Stream>
    </Array>
</XSIL>
```

The XSIL parser looks from its current classpath to find code to handle this object. In this case, the code is in a classfile called TimeSeries in a directory called TimeSeries.

2. The code to handle the extension type could look something like this:

```
package extensions.TimeSeries;
import org.escience.XSIL.*;
import java.util.*;

public class TimeSeries extends XSIL {
    double t0 = 0.0;
    double dt = 0.01;
    int ndata = 0;
    Array a;

    public void construct(){
        for(int ichild=0; ichild < getChildCount(); ichild++){
             XSIL x = getChild(ichild);
            if(x instanceof Param){
                Param p = (Param)x;
                if(p.getName().equals("t0")){
                    t0 = new Double(p.getText()).doubleValue();
                }
                if(p.getName().equals("dt")){
                    dt = new Double(p.getText()).doubleValue();
                }
            }
            if(x instanceof Array && ((Array)x).getNdim() == 1){
                a = (Array)x;
                ndata = a.getNdata();
            }
        }
        if(ndata == 0){
            System.out.println("ERROR: improper TimeSeries");
        }
    }

    public double gett0() {return t0;}

    public double getdt() {return dt;}

    public int getNdata() {return ndata;}

    public double getData(int i) {
        return a.getPrimArray().getDouble(i);
    }
```

3. The constructor for the new object is handled by XSIL, and initalization of the new object is handled by overloading the `construct(XSIL x)` method. The vector of child objects is examined, looking for the data we need to make the TimeSeries, being some parameters, and an array. If the parameters have the right names, it is assumed that numbers can be read from the corresponding text, and if the array is one-dimensional, it is assumed that it is the TimeSeries data. Note that the TimeSeries may contain other XSIL objects, which are ignored by this piece of code.

4. The other methods provide access to the TimeSeries itself, the last being the data. Associated with each Array or Table object is a PrimArray object, which holds a sequence of any of the ten primitive types (boolean, byte, short, int, etc.) from which any desired type can be extracted.

5. The data making up the time series is not read at the initialization of the object, but rather at the first call to the getData method. Files are read or URL's resolved, or delimited text is parsed. If the number of objects in that data stream is not sufficient, default values are substituted instead.

6. The `getData` call in the TimeSeries implementation fetches one number at a time. Another call is available from XSIL to get large quantities of data, it is of the form `double[] getDoubleArray(int start, int end)`, which creates an array of doubles with `end-start` elements (subscripts from `start` to `end-1`). Othe calls are available for all ten primitive types.

7. Parent objects are constructed after their children. If a user-created object is a child of another user-created object, then the `construct()` method of the child has been called before the `construct()` of the parent.

C. Coding an XML transformation

1. The following code reads and processes an XSIL file without any reference to graphics or browsing. From a file name, the XSIL root element is created, and a recursive function is called over all the child objects. All the objects whihc are Param objects are then printed.

```
package extensions.ReadXML;
import org.escience.XSIL.*;
import java.io.*;

public class listParam extends XSIL {
    public static void main(String args[]) throws IOException {
        if (args.length != 1) {
            System.err.print("Usage: java extensions.ReadXML.listParam file.xml\n");
            System.exit(1);
        }
        XSIL root = new XSIL(args[0]);
        recurse(root);
    }

    public static void recurse(XSIL x) {
        for(int ichild=0; ichild < x.getChildCount(); ichild++){
            XSIL child = x.getChild(ichild);
            if(child instanceof Param){
                String Name = ((Param)child).getName();
                String Text = ((Param)child).getText();
                System.out.println("Name: " + Name + ", Text: " + Text);
            }
            recurse(child);
        }
    }
}
```

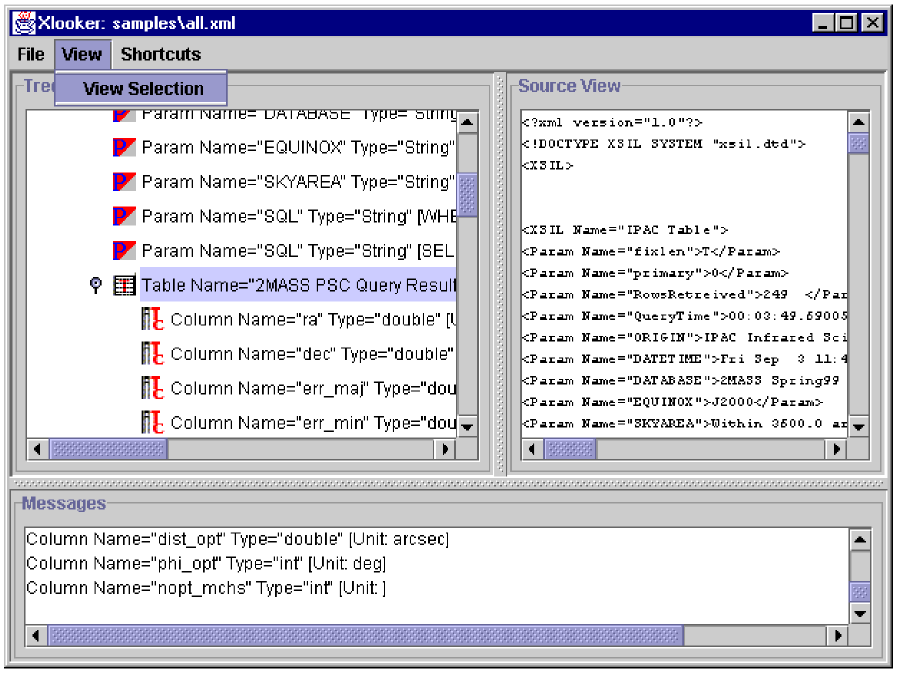
This program prints information about all of the `<Param>` elements that it finds in the XSIL file.

2. A similar scheme could be used for arbitrary reprocessing of XSIL data. In addition to metadata, the data streams are also available, either through the getInputStream method of Stream, or through higher-level constructions such as Tables and Arrays.

6. Xlook

A. The Main Window

1. The browser can be initiated from a command line with the “Xlook” command. The GUI is shown in Figure 1.



```
Param Name="DATABASE" Type="String"
Param Name="EQUINOX" Type="String"
Param Name="SKYAREA" Type="String"
Param Name="SQL" Type="String" [WHE
Param Name="SQL" Type="String" [SEL
Table Name="2MASS PSC Query Result
Column Name="ra" Type="double" [U
Column Name="dec" Type="double"
Column Name="err_maj" Type="dou
Column Name="err_min" Type="dou
```

```
<?xml version="1.0"?>
<!DOCTYPE XSIL SYSTEM "xsil.dtd">
<XSIL>


<XSIL Name="IPAC Table">
<Param Name="fixlen">T</Param>
<Param Name="primary">0</Param>
<Param Name="RowsRetreived">249   </Par
<Param Name="QueryTime">00:03:49.69005
<Param Name="ORIGIN">IPAC Infrared Sci
<Param Name="DATETIME">Fri Sep  3 11:4
<Param Name="DATABASE">2MASS Spring99
<Param Name="EQUINOX">J2000</Param>
<Param Name="SKYAREA">Within 3600.0 ar
```

```
Column Name="dist_opt" Type="double" [Unit: arcsec]
Column Name="phi_opt" Type="int" [Unit: deg]
Column Name="nopt_mchs" Type="int" [Unit: ]
```

Figure 1

The Xlook main window, showing XML source (top-right), messages (bottom), and tree view (top-left). The Table object has been highlighted, causing its toString() method to be called in the message window. The View/Selection method has been chosen to bring up the Tables’s view component.

2. This batch file expects an argument with is the XSIL file to be viewed:
`xlook samples\all.xml`

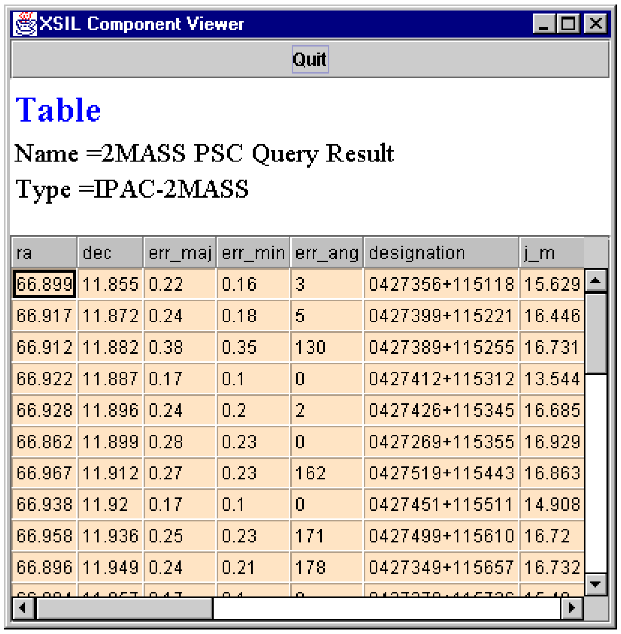
3. The main window comes up, with three panes. The lower section is a message window, although many relevant messages are still coming out in the command-line window instead. The top-right window shows the XML code, and the top-left (tree-view) window shows a hierarchical representation of the file.

4. Clicking on the nodes of the tree (little magnifying glasses) expands and contracts the tree branches in the usual way. The basic XSIL types have icons. When an XSIL object is selected with the mouse, it is highlighted in a blue box, and a short text description appears in the message window.

5. The description of the object in the tree-view window is the result of the `toString()` method of the object, which can be overloaded in extension code. The description in the message window is supposedly a little more complete, and is the result of the `toLongString()` method, which can also be overloaded

B. View Components

1. The view component for the Table object is shown in Figure 2.



Table

Name =2MASS PSC Query Result

Type =IPAC-2MASS

ra	dec	err_maj	err_min	err_ang	designation	j_m
66.899	11.855	0.22	0.16	3	0427356+115118	15.629
66.917	11.872	0.24	0.18	5	0427399+115221	16.446
66.912	11.882	0.38	0.35	130	0427389+115255	16.731
66.922	11.887	0.17	0.1	0	0427412+115312	13.544
66.928	11.896	0.24	0.2	2	0427426+115345	16.685
66.862	11.899	0.28	0.23	0	0427269+115355	16.929
66.967	11.912	0.27	0.23	162	0427519+115443	16.863
66.938	11.92	0.17	0.1	0	0427451+115511	14.908
66.958	11.936	0.25	0.23	171	0427499+115610	16.72
66.896	11.949	0.24	0.21	178	0427349+115657	16.732

Figure 2

The Table view component, using the LiveTable component from klgroup.com. The table can be sorted with a shift-click on any of the gray column headers. In this case, it is sorted on the second column.

2. The TimeSeries extension that comes with the XSIL distribution has a view component constructed from the KL Group's JChart component, as shown in Figure 3. There is a customization panel giving great control over the look of the chart, though this has not yet been switched on.

3. These light-weight versions of the components are freely usable and distributable, but the full-strength versions are not free. The light-weight versions are included in the XSIL distribution.

4. The view component for the `<Url>` tag is to spawn a web browser. Once the browser reads the MIME type of the associated data object, it may bring up a specialized application. In this way, XSIL can contain references to very complex data objects, such as Excel spreadsheets or astronomical FITS files.

5. As another example of a view component, we show a Java3D display. In this window, the shape is fully three-dimensional, and the graphics can take advantage of any OpenGLaccelerator that may be in the workstation. The left mouse button rotates the shape, the middle button scales it, and the right button translates. The XML that defines the icosahedron shape is as follows:

```
<XSIL Type="Java3D">
    <XSIL Type="Java3DGeometry">
```

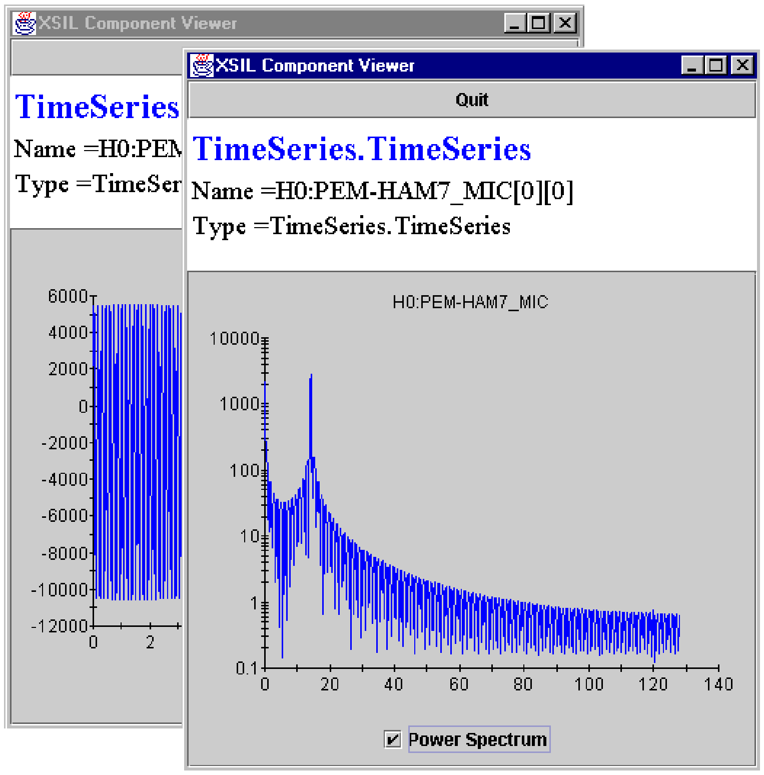


Figure 3

The view component for the TimeSeries extension. The name and type of the object are at the top, as with every view component. This microphone channel from Ligo engineering tests is shown in the time domain (below), and is perhaps more understandable in the frequency domain (above).

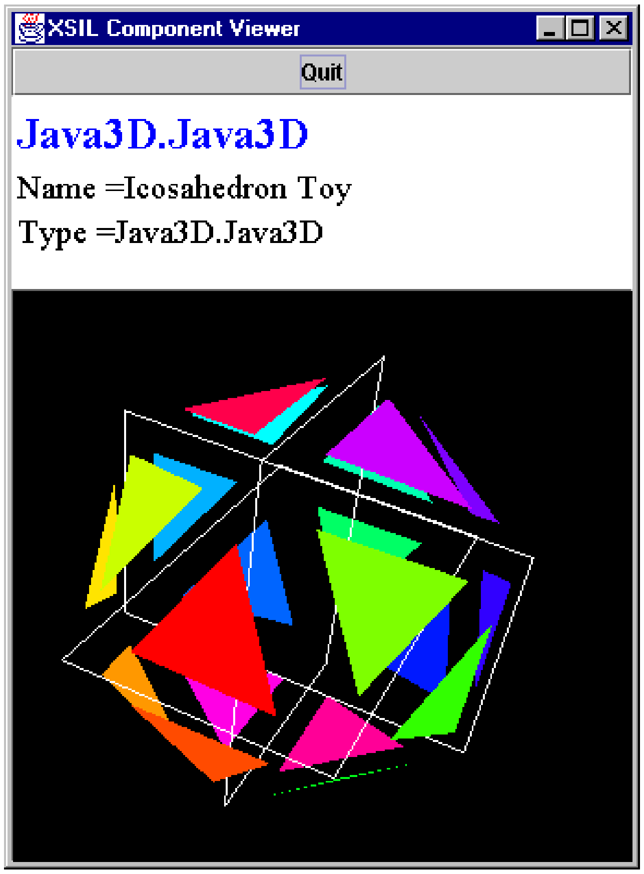


Figure 4

The view component for the Java3D extension. The three mouse buttons can be used for 3D rotation, scale, and translation of the image. This object consists of a TriangleArray (the colored triangles), and three LineArrays (the white rectangles). It illustrates how an icosahedron can be constructed from three golden rectangles.

The TriangleArray and LineArray objects are based on objects from Java3D API, but implemented with XSIL through Arrays and Params expressing what is coordinates and what is colors, what is triangles and what is lines.

```
        <Param Name="Java3DGeometryType">TriangleArray</Param>
        <Array Name="Coordinates" Type="float">
            <Dim>60</Dim>
            <Dim>3</Dim>
            <Stream Encoding="Text" Type="Remote" Delimiter=", ">icocoord.dat</Stream>
        </Array>
        <Array Name="Colors" Type="float">
            <Dim>60</Dim>
            <Dim>3</Dim>
            <Stream Encoding="Text" Type="Remote" Delimiter=",">icocolor.dat</Stream>
        </Array>
    </XSIL>

    <XSIL Type="Java3DGeometry">
        <Param Name="Java3DGeometryType">LineStripArray</Param>
        <Array Name="Coordinates" Type="float"><Dim>5</Dim><Dim>3</Dim>
            <Stream Encoding="Text" Type="Local" Delimiter=", ">
                -1.0,   0.0,  -1.61803399,
                -1.0,   0.0,   1.61803399,
                 1.0,   0.0,   1.61803399,
                 1.0,   0.0,  -1.61803399,
                -1.0,   0.0,  -1.61803399,
            </Stream>
        </Array>
    </XSIL>
    .... (two more rectangles like the one above) ....
</XSIL>
```

There is an outer enclosing element of type Java3D.Java3D, which contains a number of objects of type Java3D.Java3DGeometry, which are the Triangle Arrays, Line Arrays, and so on which can be used to create 3D objects in the view window. Of course, other types of data can be encoded in the XML, then rendered in a different way by modifying the rendering code (in the extensions/Java3D directory).

7. Extending Xlook

A. Object Model

1. An Xlook view component is an extension of a class XSILView, which extends javax.swing.JComponent, which is a generic graphics object in Swing.

2. The class implementing the viewing must have the same name as the object which it views, but with the suffix “View” added. In the code below, for example, the Label object is viewed by a LabelView object.

3. When an Xlook user requests a view component, an outer frame is made by Xlook, with the Name and Type of the object, and a scrollable panel (javax.swing.JScrollPane) is created to hold the client-supplied viewer.

4. The instantiate() method of the viewing object is called, with the argument guaranteed to be castable to the expected object, and the completed frame rendered. This code may create buttons and other widgets that can receive events, as in the TimeSeries viewer shown above, where a checkbox can cause a power-spectrum to be computed.

B. Complete Example

1. In this section we present a very simple, but complete example of the XSIL system, with the XML, the Object code, and the Viewer code. The Label is defined in XML like this:

```
    <?xml version="1.0"?>
    <!DOCTYPE XSIL SYSTEM "xsil.dtd">
    <XSIL>
    <XSIL Type="Simple.Label" Name="Example">
        <Param Name="Message">Hello Auntie Joan</Param>
        <Param Name="FontSize">96</Param>
    </XSIL>
    </XSIL>
```

So the label object is just a piece of text (Message) and an integer (FontSize).

2. Code to read this object looks like this. The children are examined to find parameters that can supply the message and fontsize fields.

```
package extensions.Simple;
import org.escience.XSIL.*;

public class Label extends XSIL {
    public String message = "No message";
    public int fontsize = 12;
    public void construct(){
        for(int ichild=0; ichild < getChildCount(); ichild++){
             XSIL x = getChild(ichild);
            if(x instanceof Param){
                Param p = (Param)x;
                if(p.getName().equals("Message")){
                    this.message = p.getText();
                }
                if(p.getName().equals("FontSize")){
                    this.fontsize = (new Integer(p.getText())).intValue();
                }
            }
        }
    }
}
```

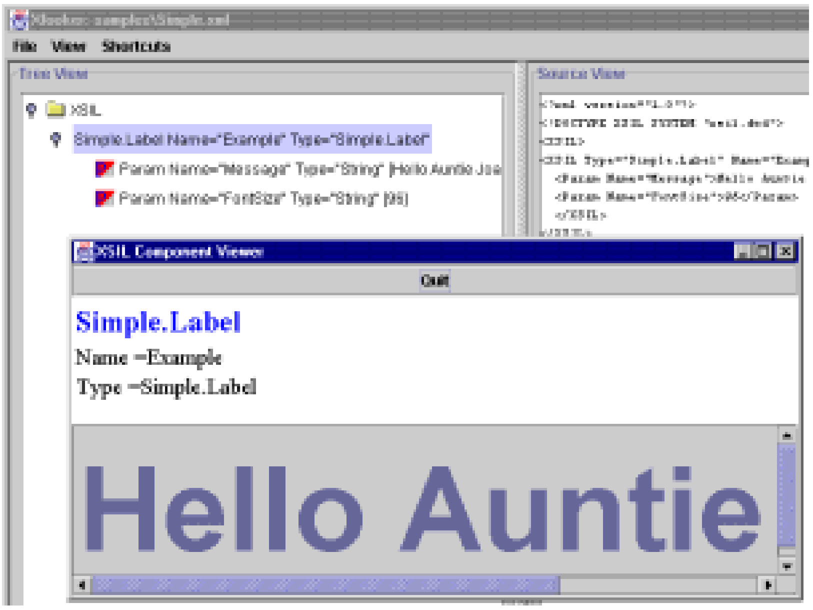
3. Once the label is complete, the view component may be called from Xlook. In this case a `JLabel` is created to show the message in the chosen pointsize.

```
package extensions.Simple;

import org.escience.XSIL.*;
import org.escience.Xlook.*;
import java.awt.*;
import javax.swing.*;

public class LabelView extends XSILView {
    Label s;

    public void instantiate(XSIL x) {
        this.s = (Label)x;
        this.setLayout(new FlowLayout());
        Font f = new Font("Helvetica", Font.BOLD, s.fontsize);
        JLabel label = new JLabel(s.message);
        label.setFont(f);
        this.add(label);
    }
}
```



XSIL
Simple.Label Name="Example" Type="Simple.Label"
Param Name="Message" Type="String" [Hello Auntie Joe
Param Name="FontSize" Type="String" [95]

```
<?xml version="1.0"?>
<!DOCTYPE XSIL SYSTEM "xsil.dtd">
<XSIL>
<XSIL Type="Simple.Label" Name="Examp
  <Param Name="Message">Hello Auntie
  <Param Name="FontSize">95</Param>
  </XSIL>
</XSIL>
```

Simple.Label
Name =Example
Type =Simple.Label

Hello Auntie

Figure 5

The view component defined by the above code, together with the Xlook window showing the XML that drives it.

C. Image List extension

1. Another extension of the Xlook browser is illustrated by the `ImageList` code. An XSIL file contains a collection of images,each with metadata that is defined by parameters with specific names. The images themselves are sometimes external files, sometimes in the XSIL document itself as base64 encoded binary.

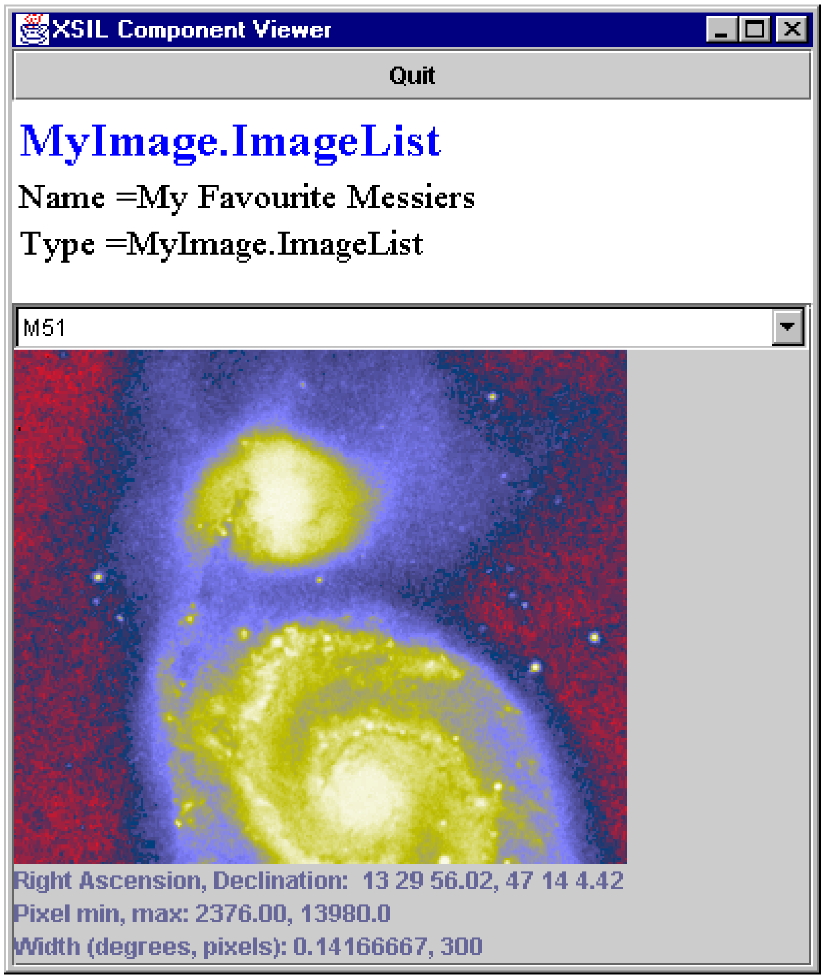
2. Here is a section of the XSIL file:

```
<XSIL Type="MyImage.ImageList" Name="My Favourite Messiers">
```

```
<XSIL Name="M54" Type="MyImage.MyImage">
<Param Name="PixelMin">4552.00</Param>
<Param Name="PixelMax">23423.0 </Param>
<Param Name="RegionWidth">0.14166667</Param>
<Param Name="ImageWidth">300</Param>
<Param Name="RightAscension">18 55 3.28</Param>
<Param Name="Declination">-30 28 42.58 </Param>
<Stream Type="Remote">m54.gif</Stream>
</XSIL>
```

Here we see the description of the first image, which is the galaxy M54. There are two objects defined here, `MyImage.MyImage` is a single image with metadata, and `MyImage.ImageList` is a list of such.

3. The corresponding view component looks like this:



MyImage.ImageList

Name =My Favourite Messiers

Type =MyImage.ImageList

Right Ascension, Declination: 13 29 56.02, 47 14 4.42

Pixel min, max: 2376.00, 13980.0

Width (degrees, pixels): 0.14166667, 300

Figure 6

Viewing the ImageList component. The image can be selected from a pulldown choice, and the image metadata appears below. The code for this is about 100 lines, and it is included in the XSIL distribution.

8. The DTD for XSIL

1. An XML file may be associated with a Document Type Definition (DTD) which defines the allowed tag names in the document, and how these fit together: which elements may contain which other elements, and how many of each element there may be.

```
<!ELEMENT XSIL ((XSIL|Comment|Url|Param|Table|Array|Stream)*)>
<!ATTLIST XSIL Name CDATA "" Type CDATA "">

<!ELEMENT Comment (#PCDATA)>

<!ELEMENT Param (#PCDATA)>
<!ATTLIST Param Name CDATA "" Type CDATA "" Unit    CDATA "" >

<!ELEMENT Url (#PCDATA)>
<!ATTLIST Url Name CDATA "" Type CDATA "" href    CDATA "" >

<!ELEMENT Array (Dim* , Stream?)>
<!ATTLIST Array Name CDATA "" Type CDATA "" Unit CDATA "">
<!ELEMENT Dim (#PCDATA)>
<!ATTLIST Dim Name CDATA "" Type "">

<!ELEMENT Table (Column* , Stream?)>
<!ATTLIST Table Name CDATA "" Type CDATA "">
<!ELEMENT Column EMPTY>
<!ATTLIST Column Name CDATA "" Type CDATA "" Unit CDATA "">

<!ELEMENT Stream(#PCDATA)>
<!ATTLIST Stream Name CDATA "" Type CDATA "" >
<!ATTLIST Stream Content CDATA ""Encoding CDATA "" Delimiter CDATA "">
```

9. Installation and Use

1. The installation is available as a zip file from

- `http://www.cacr.caltech.edu/XSIL/`

by following the link “available software”.

2. Javadoc documentation comes with the distribution, or else from

- `http://www.cacr.caltech.edu/XSIL/javadoc/`

3. The XSIL environment has been tested on Solaris and Windows 98 and NT.
4. To run the XSIL environment, you will need:

- A computer with Java Development Environment (JDK) 1.2 or later. You will need to know the directory where it is installed, meaning the directory which has the /bin and /lib subdirectories. For more information and free download, see
`http://java.sun.com/products/jdk/1.2/`

- One of the demonstration codes utilizes the Java3D package. If you wish to run this demo, you will need to install Java3D, which is available free from:
`http://java.sun.com/products/java-media/3D/index.html`
The Java3D is implemented with OpenGL, so it will probably find your fast graphics card. ***NOTE:*** Installation is much more difficult if you choose to put Java3D in a different directory from that suggested by the install wizard.

- You must also have an unzip or untar facility.

5. Unpack the distribution, and go to the XSIL root directory, which is the one that contains com, org, doc, and extensions directories. There should also be some scripts here, labelled `.bat` for Windows, `.sh` and `.source` for Unix.
6. Make sure the `JAVA_HOME` and `WEB_BROWSER` locations are correct in the setup.bat (Windows) or setup.source (Unix) script.

- The web browser is not necessary unless you intend to use the <URL> tag, which spawns a browser for viewing.

7. Start a command window and run the setup script. Type `setup` (for Windows) or `source setup.source` (for Unix). Make sure the Java interpreter is in your path: type `java`, it should give a usage message rather than command not found.
8. For Unix users,
```
% source setup.source
% xlook.sh samples/all.xml
```
9. For Windows users,
```
c: setup
c: xlook samples\all.xml
```
10. The browser should now come up, and there are a number of sample in the samples directory.
11. ***NOTE to DEVELOPERS***. The key to the Java package system, how it finds classes, how it compiles, making life easy, requires you to follow one rule: always run your compile (`javac`) and applications (`java`) from the XSIL home directory, the one with demo scripts.
12. You can also examine and modify the code in the `extensions` directory. To make a custom viewer, a good start might be a copy of the `extensions/Simple` directory.