Metadata Extraction Tool Software Architecture

Version: 3.5.

Table of Contents

Background

The *National Library of New Zealand Te Puna Mātauranga o Aotearoa (National Library)* has been developing its approach to the management of electronic material. Highlighted by this development is the need for a Digital Archive and the desire to improve access to its collections via digitisation. As part of this work it has become clear that preservation of digital materials is emerging as a new business need for the Library and that preservation metadata is an integral component of successfully meeting that need.

A high proportion of the preservation metadata will be in narrative format and will require manual entry by Library staff. A significant subset of the data however, relating to technical file characteristics, can be automatically extracted from the digital object by reading the file header details. This successful extraction of preservation metadata has been proved in a previous National Library proof of concept project. The automated capture of this information will significantly reduce the amount of manual data entry required from Library staff.

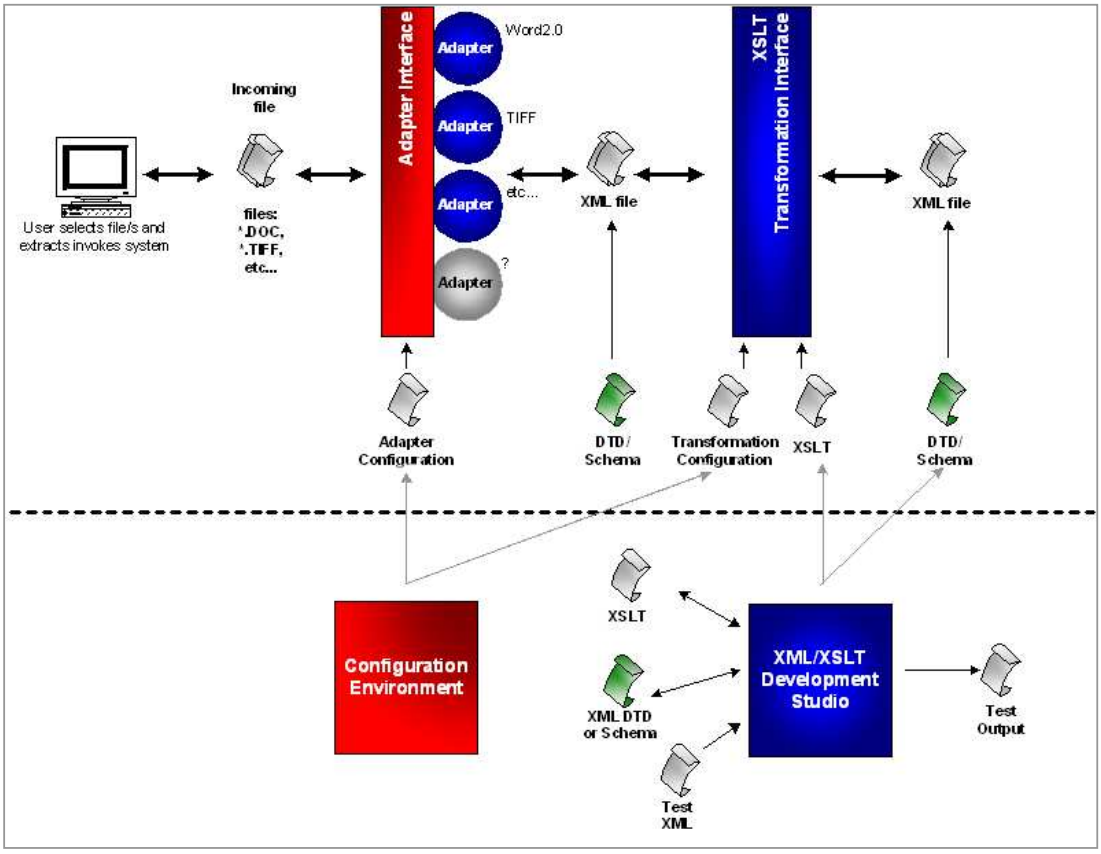
The work that is the focus of this document is to take the software developed in the Proof of concept and package it to support the full set of requirements for a metadata extraction tool. Key requirements include:

- Support for objects (complex and simple objects) where files are grouped into objects.
- Logging
- Preparing the code base, design and deployment for production.

This document is intended to be a foundation for further development in this area by the National Library.

Solution Architecture

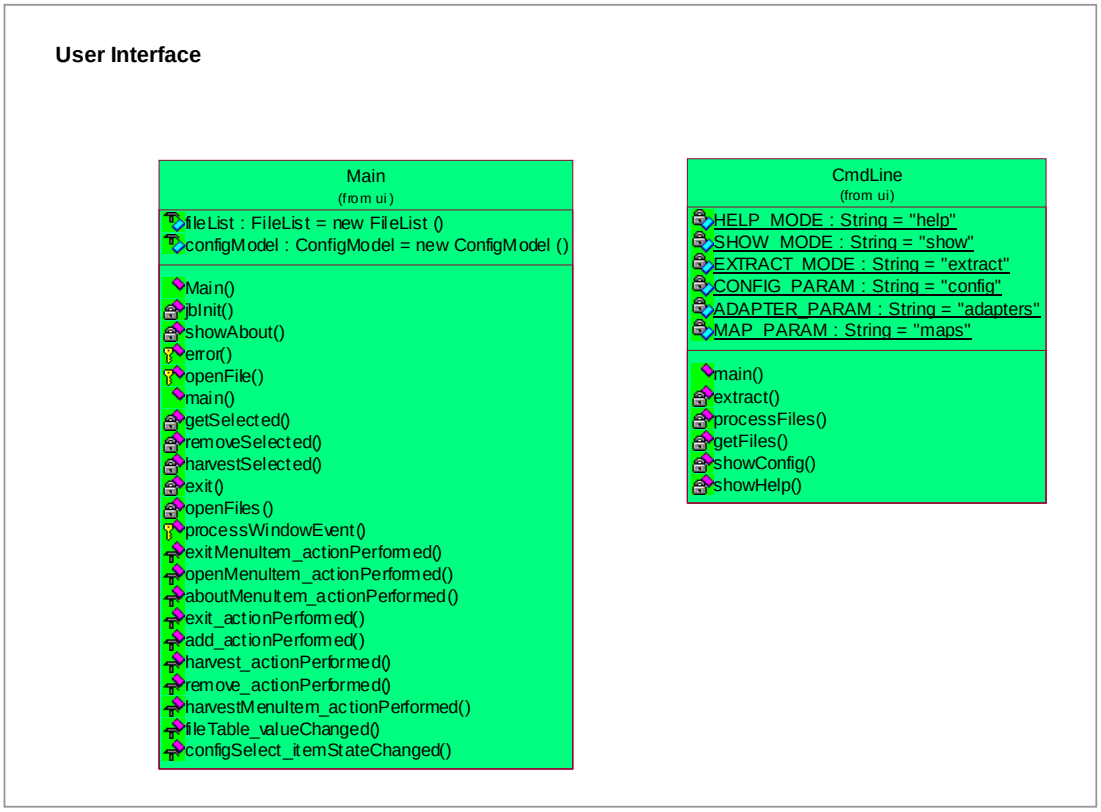
The document: Metadata Extraction Tool – Solution Architecture, outlines the solution at a high level. This document begins where the solution architecture finishes and details the software components of the system.



The software components for each stage will be presented in following sections.

User Interface

UI Classes



The user interface base classes appear in the diagram below. The Main class is the primary Graphical User interface component. It handles all user interaction via Java Swing components. It is multithreaded so that the act of extracting metadata can take place in it's own low priority thread. Independent threading is required for several reasons:

1. If the thread is given a low priority it will cause less contention for the CPUs time. When the CPU is free the extraction software will use 100% of the available processing cycles. However, when the processor has a more important task the software will yield processor cycles to that task.
2. The user can be given updates via the UI about the progress being made. If processing occurred in the event handler thread the UI would not be updated until processing was complete - which is too late.
3. The user can cancel the extraction process if required because the event handler thread will be available to handle the press of the stop button.

Files

Incoming files that are selected for processing are parsed by a set of adapters (parsers) that know how to read the binary file format of the file. Each file will have it's own unique file format which is obtained from the public domain, or in some cases the software vendor itself. Because of the proprietary nature of some file formats it may be necessary to obtain the formats by leveraging the National Libraries status as a non-profit, educational organisation.

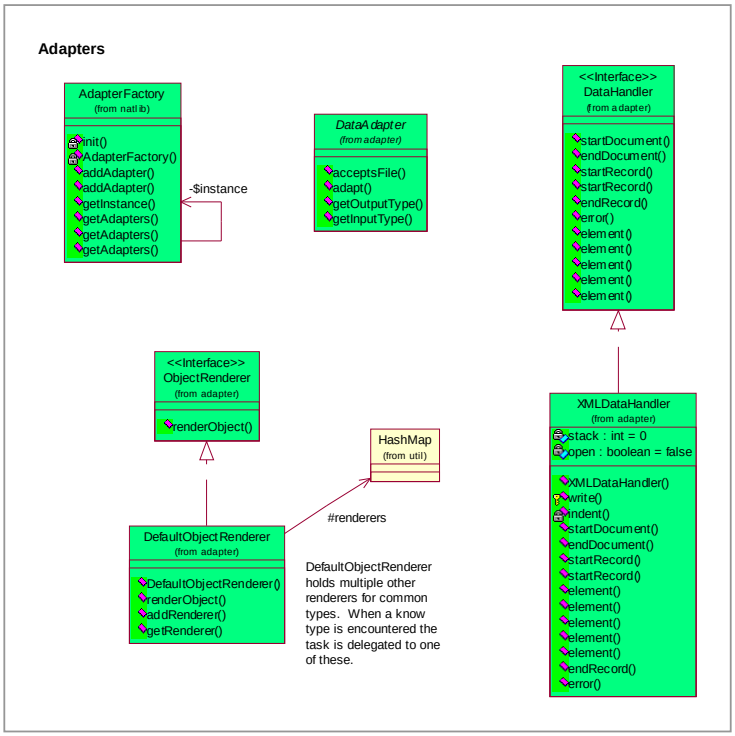
For the *Proof of Concept (POC)* it was decided to focus on two types of files, TIFF and Microsoft Word. While TIFF has not changed format since creation in the 80's MS Word files have gone through several iterations. The POC clearly identified the viability of opening and reading input files across types and versions within types. TIFF and .DOC files have been selected as good file formats to prove this.

Since the development of the POC, support has been added for the following types:

- Bitmap
- Microsoft Excel
- GIF
- HTML
- JPG
- MP3
- Open Office
- PDF
- Microsoft PowerPoint
- TIFF
- Wave Audio
- Microsoft Word
- WordPerfect
- Microsoft Works
- XML
- FLAC
- BWF
- ARC

Adapters

Each adapter registered in the system (see configuration) is responsible for parsing a specific input file type. Each adapter must extend the DataAdapter class, which defines several methods:



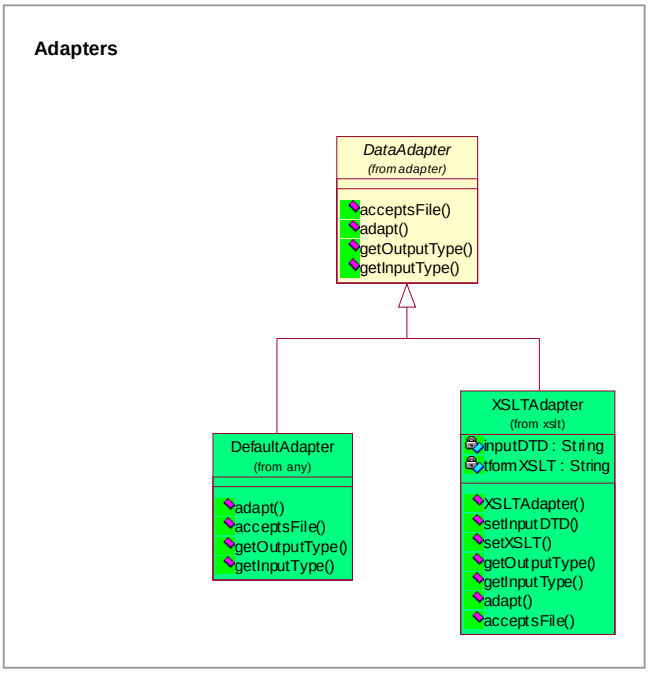
1. acceptsFile(File). This method returns true or false depending on whether the adapter can process the given input file.
2. adapt(File, DataAdapter). This method is the main method of any adapter. When called the adapter must parse the given file into the DataAdapter
3. getOutputType(). Returns a string value containing the output DTD of the data that the adapter returns as a result of parsing.
4. getInputType(). This method returns a string value containing the type of file (Mime Type) that the adapter reads. It is possible that Adapters might be chained together in the future so provision is made for the Input type to be a DTD in addition to a mime type.

Adapter base components

Other base classes that fall into the Adapter package are:

- DataHandler. This class is modelled after the SAX Document Handler class. It is to be implemented by classes that are interested in the progress of a parser as it parses a file. Adapters use this class as a sort of “Output stream” to write specific information out about a file.
- XMLDataHandler. This concrete class is a specific implementation of the DataHandler class. The XMLDataHandler is given an output stream upon creation, which it uses to output correctly formatted XML to. It handles all events, and outputs the passed in information into the output stream.
- AdapterFactory. This class is responsible for returning an adapter that is capable of parsing a given input file. It does this by holding a list of all adapters and their associated input types (see configuration) and using this list to identify the appropriate Adapter. It also uses the acceptsFile() method that all Adapters implement to query each adapter about their ability to parse a file. This means that if there are a lot of Adapters the AdapterFactory may take some time to find the correct Adapter for a file. Also not handled by the tool is the case where multiple adapters think they can process a file – at this stage the first adapter that returns true from the acceptsFile() method gets that job (so ordering in the configuration file is important)

Generic Adapter Components

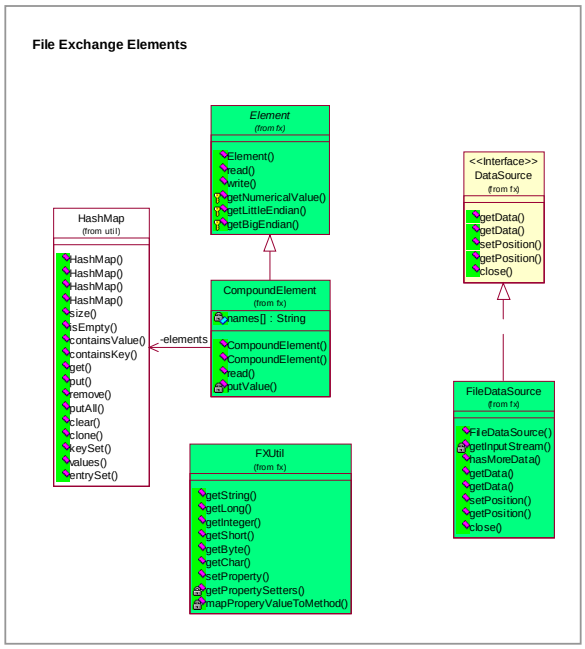


There are two adapters in the system that are considered generic:

- DefaultAdapter. Will handle any file, outputting information about the file gathered from the file system (i.e. no parsing). Care should be taken when registering this Adapter in the configuration so that it is the *last* adapter registered. This is because it will always accept any file, stealing other adapters chances of parsing that file.
- XMLAdapter. This adapter is capable of taking any input XML file and processing it according to an XSLT script. Setup parameters for this Adapter are:
    - The input DTD filename.
    - The XSLT script filename.
    - The output DTD filename.

I/O Utilities

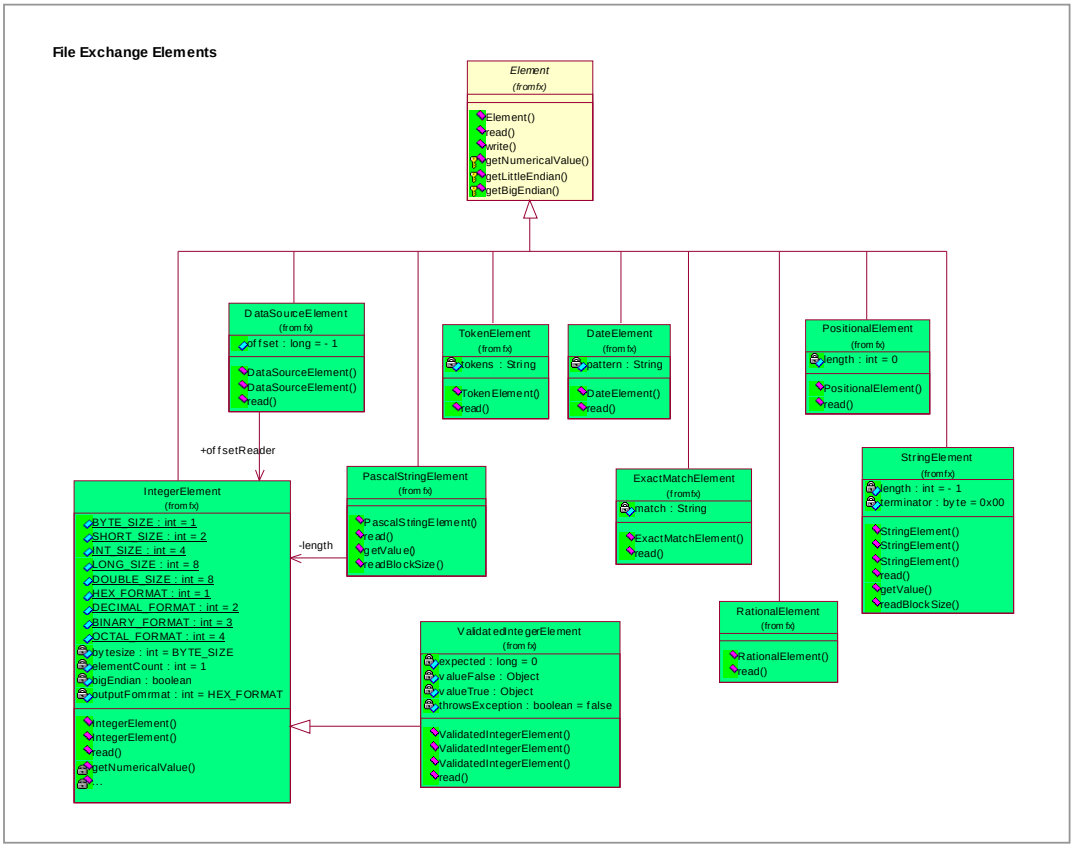
There is a number of parser utilities included with the application code. These utility classes include.



- Element: This class is the root class for all Elements. An element is a part of a file that represents a complete object. A complete object can be a number with a certain word length, a Date or a number of other base types. An element can also be a compound collection of elements that together have meaning (e.g. a header record).
- DataSource: A DataSource is a generic source of data that is to be parsed. The DataSource interface must be implemented by all sources of Data. A DataSource is meant to be a generic layer that overlays a random access data source of some description. DataSources are supposed to be generic however so non-random access data sources (FTP, HTTP, socket etc…) should be able to be accommodated.
- FileDataSource: A concrete implementation of a DataSource. This DataSource represents a file on the local file system.

- FXUtil. A collection of methods to be used by specific implementations of Element for the conversion of values to primitive types.
- CompoundElement. This Element is a collection of smaller elements. It is useful for reading header blocks. In fact it is expected that most files are made up of elements within compound elements within parent compound elements (i.e. a "tree" of elements). A compound element usually returns the results of the read() method as a HashMap of all the results of it's children elements. A useful addition to CompoundElement is the capability of passing a JavaBean class to the Compound element for the results of the read() to be stored in.

There are several implementations of the Element class, key ones are:



- IntegerElement. This element reads an integer with a specified word length from the DataSource. It is capable of reading the integer as most significant byte first (big endian: Java & general Macintosh Default – i.e. the right way) or least significant byte first (little endian: C/C++ and general Intel platform).
- DateElement. This element reads a date from the DataSource. The format of the date is the specified pattern.
- ExactMatchElement. This element reads a block of bytes that should exactly match a specified block of bytes. If it doesn't, an exception is thrown. This is useful to build into parsers that require certain versions of the input file to work correctly.
- PascalStringElement. This is a string element where the first 2 bytes represent the length of the string. (Like Pascal)
- PositionalElement. Reads a variable length block of bytes. It doesn't do any interpretation of the block.
- RationalElement. Reads two longs (4bytes). The first one is the numerator, the second is the denominator (divisor). The resulting double value is returned.
- StringElement. Reads a string from the DataSource. The string can be either a fixed length or null (00x0) terminated.

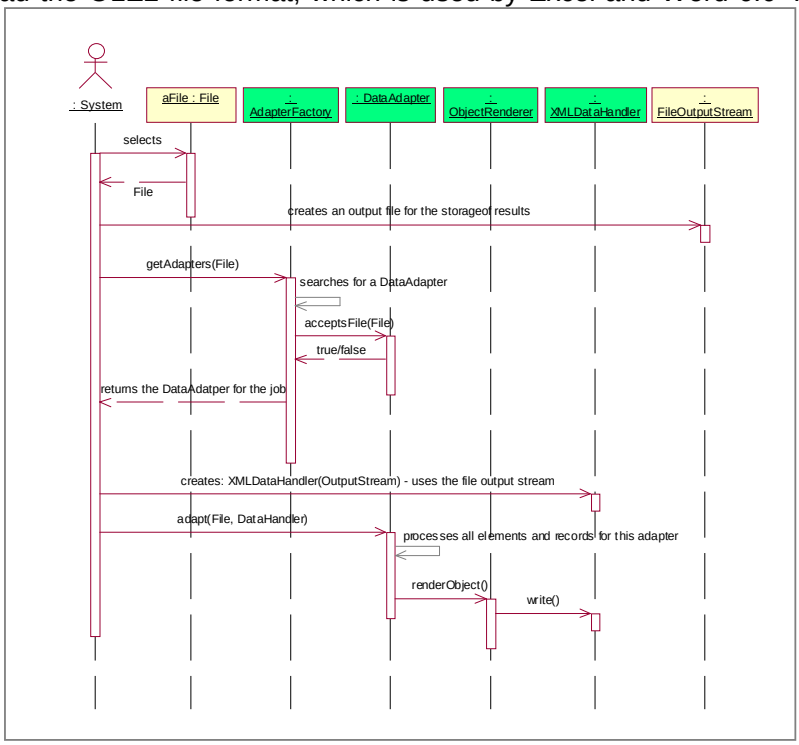
- TokenElement. Reads a block of bytes until the given termination character is reached. This will be useful for the reading of comma-separated files.
- ValidatedIntegerElement. This element reads an integer with a specified word length that should exactly match an input. If it doesn't an exception is thrown. This is useful to build into parsers that require certain versions of the input file to work correctly.

POI

POI is a third party tool similar to the "elements" listed above. The POI specification and library is designed to read the OLE2 file format, which is used by Excel and Word 6.0+. Additional information about POI can be found at: http://jakarta.apache.org/poi/index.html. See appendix B for information about how the POC uses the POI library.

Parsing a file

The way in which a file is parsed is best described as a sequence diagram (right). The user selects the file and the process of "harvesting" information from it is begun. First the AdapterFactory is asked to return an Adapter to do the job. The adapter is then asked to parse the file into a DataHandler, which in turn streams characters into an output stream. The description above is a simplistic version of what actually happens. Normally there is another layer of translation to take the output from the Adapter and turn it into a uniform XML file (i.e. a specified DTD like nlnz_presmet.xsd). This translation is covered in the next section.

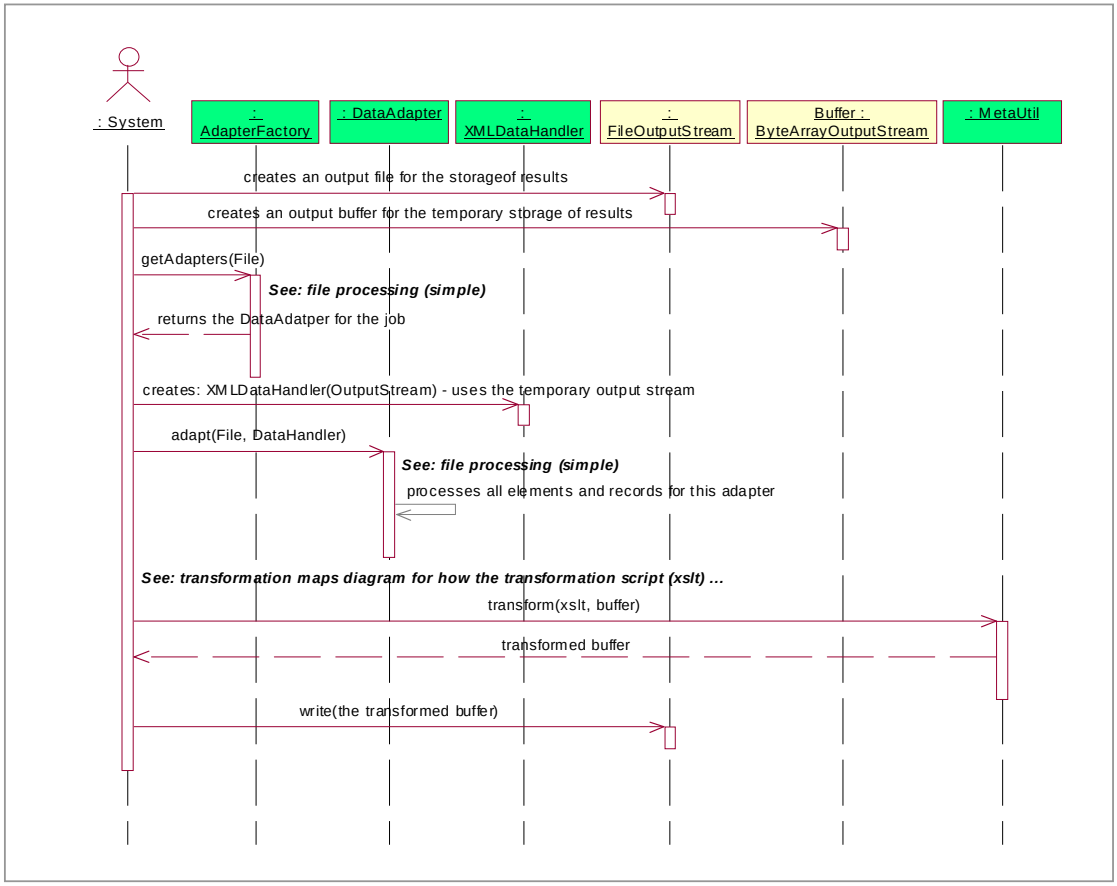


XSLT Translation Interface

Before a file is output in its' native file format it may be translated into another, more generic file format. The component that does this is an XSLT translator that acts like a DataHandler. This special type of DataHandler implements all the standard DataHandler methods but instead of outputting directly into an output stream it imposes a layer of translation *before* outputting into an output stream. The XSLT scripts make use of Java extensions to call functions that have been written in Java to perform some functions best done in a procedural language.

Processing adapter output with XSLT

The XSLT scenario looks very similar to the previous file-processing diagram but has additional steps that transform the output of the adapters into a generic format. XSLT



scripts are used to perform this function. The library used to perform the translation is the Apache Xalan project. Xalan is an open source library for the processing of XML files. It conforms to the JAXP standard (from Sun Microsystems), which is a standard feature of Java 1.4 (but must be included separately under Java 1.3 or less). For more information on Xalan see the web site: http://xml.apache.org/xalan-j/index.html.

Output files

The tool is configured to output several formats of XML

Format (DTDs)

In addition to the native DTD for a particular file there is one other type of standard metadata file (although any number can be added):

1. nlnz_presmet.xsd. This is an XML schema that closely represents the National Library Preservation Metadata Data Dictionary.

Objects

Objects are logical groupings of a collection of files' metadata. This means that there will be many files inputting metadata information into a single output file (known as a complex object). The 'grouping' will be entirely arbitrary and in many cases will be directed by curators at the library. An example of an object that has related collections of files is a website. The site itself is the object and all files and folders found within it are logically part of that object.

Logging

The logging window displays the current log. You can elect to clear the current log and start it fresh, or filter the current log to only show a certain level of logging.

Critical.

The application has had a critical failure, harvesting could not be considered unstable and the application should be restarted.

Error.

An error is an application problem or an problem while harvesting metadata that is isolated to the object being harvested. Chances are other objects were unaffected and harvesting can continue.

Debug messages

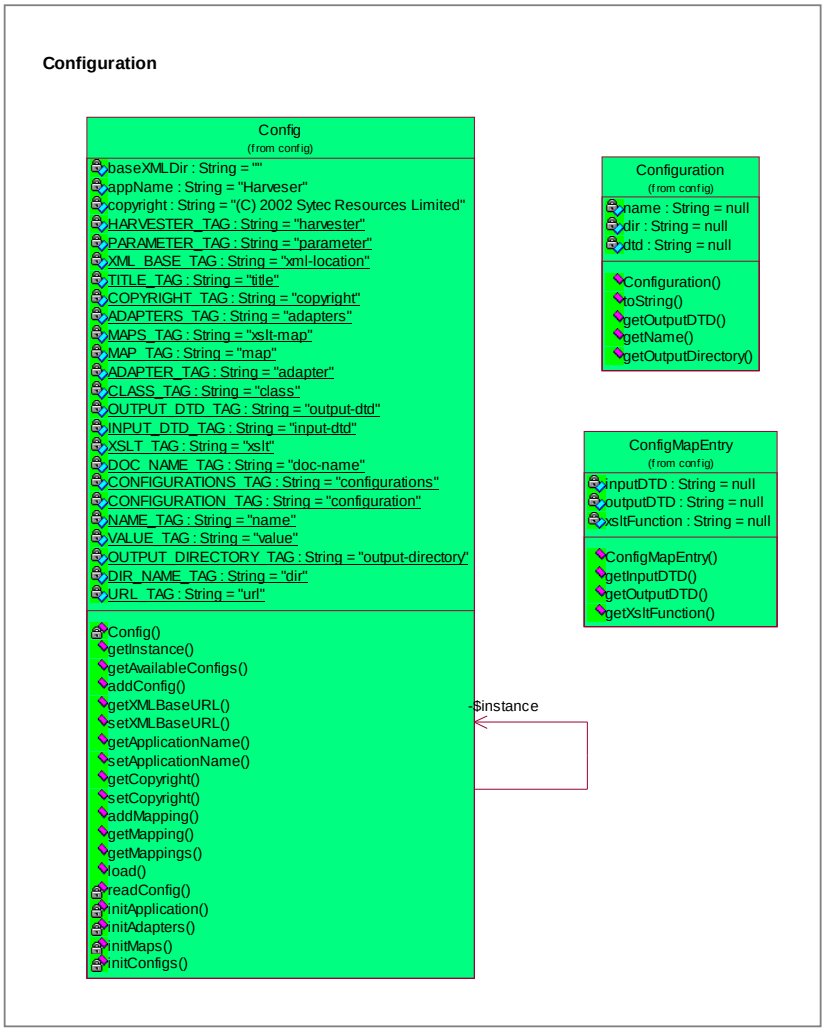
Information about program behaviour, there should be very little of these messages in the production system.

Information message

Superfluous information about application behaviour. This includes things like usage reporting.

Program Workings

Similar to debug, these messages are closely related to system functions – they may not be very meaningful to most operators, there should be very little of these messages in the production system.



Configuration

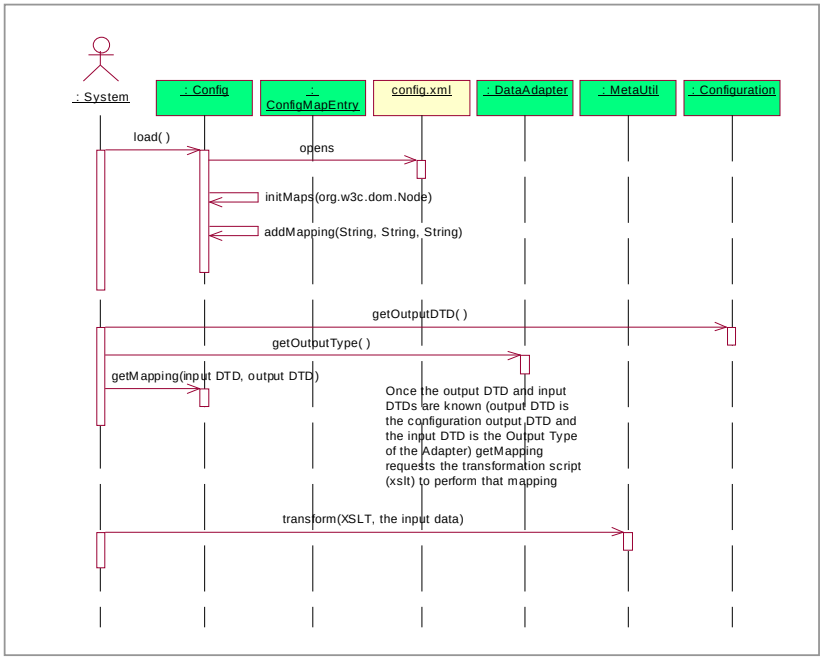
A single config file configures the tool. This file is read by the Config class on start-up. The config can control:

- some application attributes.
- Adapters
- XSLT maps.
- Output formats.

Each of these configuration elements is discussed later in this section.

The way in which the Config reads the config.xml file is as follows:

1. The method load() is called as soon as the Java Virtual Machine loads the class Config.class.
2. The Config class uses the System class loader to locate and read the config.xml file. For this reason the config.xml file must be located somewhere within the project classpath. If the application is being run from the NLNZ.jar file in the deployment package (see solution architecture) then the class loader can find the config.xml file in the same directory as the jar file.



3. Application properties are read from the file.
4. The Adapter list is read from the file.
5. The XSLT map is read from the file.
6. The required output formats are read from the file.

Developing a new Adapter

This section explains how to create/develop a new adapter into the Metadata extractor tool. As an example the MPEG-4 adapter is added to the existing tool. Presently the Version of Metadata Extractor Tool is 3.5.

Once you have decided to add extractor for a chosen format and clear on the format's technical specification, there are two phases to complete this enhancement to the MDE tool.

1. Creating a java adapter class for this format
2. Creating /modifying supporting files in order to enable this new adapter in the MDE tool

Phase-1: Development

Creating MPEG4 Adapter

1. Create a new package **nz.govt.natlib.adapter.*new_adapter_class*** under the C:\JavaDev\MDE\metadata-extractor\src\java folder. In our case we are creating the following package **nz.govt.natlib.adapter.mpeg4**

2. Create a class newformatAdapter.java in our example we create MPEG4Adapter.java that extends **nz.govt.natlib.adapter.DataAdapter.** Create **`nz.govt.natlib.adapter.mpeg4.MPEG4Adapter`**.

3. This class implements all the abstract methods of DataAdapter. The important methods are

```
public abstract boolean acceptsFile(File file);

public abstract void adapt(File file, ParserContext out) throws
IOException;
```

AcceptsFile()

Whenever a file is selected for metadata extraction, that file is passed on to the **acceptsFile()** method of the underlying adapter classes. The first matching adapter class is used to perform the metadata extraction for this file.
This method implementation can differ from format to format. It could be as simple as to identify the format by simply the file extension or device it's own logic based on the format specification of that file or use an external library to do this task. In case of the MPEG4 adapter, we make use of an external library for both format identification and extraction of the significant metadata for MPEG4 file format. For more details on the implementation of this library please refer to the following project page link
http://code.google.com/p/mp4parser/

```
public boolean acceptsFile(File file) {
     try {
             IsoFile isoFile = new IsoFile(new
             FileRandomAccessDataSource(file));
             BoxFactory boxFactory = new BoxFactory();

             IsoInputStream isoIn = new
                   IsoInputStream(isoFile.getOriginalIso());

             Box box = boxFactory.parseBox(isoIn, isoFile, null);
             if (box == null) {
                   LogManager.getInstance().logMessage(LogMessage.ERRO
                   R,"The input file does not implement ISO Mpeg4
                   Standards");
                   return false;
             }else {
                   return true;
             }
      }catch (Exception e) {
             return false;
      }

}
```

adapt()
This method extracts the different significant properties of the given file. It constructs an internal hash map and also an xml displaying the properties in the order they are supposed to appear according to the format specification for that file format.
The MPEG4 adapter is developed as a wrapper around a third party library, which extracts these properties and returns to the MPEG4 adapter. The MPEG4 adapter then builds the xml using these values.

```
public void adapt(File file, ParserContext ctx) {
            FileRandomAccessDataSource ds = null;
            try {
                  ctx.fireStartParseEvent("MPEG4");
                  writeFileInfo(file, ctx);
                  ctx.fireEndParseEvent("MPEG4");
                  ds = new FileRandomAccessDataSource(file);
                  IsoFile isoFile = new IsoFile(ds);
                  isoFile.parse();
                  Box[] boxes = isoFile.getBoxes();
                  openBox(boxes);
                  //Logic to extract metadata and write them into
                  the output xml as
                  //tags using the ParserContext goes here

            }
```

Phase-2: Configuration

Once the new adapter code is ready in the metadata extractor source code, the next step is to enable this adapter to be used by the MDE tool. In order to does that carry out the following steps.

1. Build.xml

Change the build.xml located in the metadata-extractor folder to include the new adapter while creating the build. Add an ant target to build this adapter.

```
<!-- prepare the files for the mp4 adapter -->
<antcall target="adapter_build">
      <param name="prm-adapter-type" value="mpeg4" />
      <param name="prm-adapter-name" value="mpeg4" />
      <param name="prm-adapter-dir" value="mp4_adapter_1_0" />
</antcall>
```

2. Format.dtd

In the metadata-extractor\src\xml folder create a dtd to specify the significant properties that the adapter has to extract. The name of the dtd file may be same as the adapter package name for convince although it is not mandatory. A file by name **mpeg4.dtd** has been created for the MPEG4 adapter. The content of this dtd determines the elements of the xml that will be created by the adapter. In some of the adapters this dtd is not being applied to validate the xml, but this dtd still needs to be there in the xml folder as it is used in deciding which adapter will be assigned to extract the metadata for a given file. For example in case of MPEG4 adapter it looks like the following snippet.

Dtd file content here

3. Format_to_nlnz_presmet.XSLT

In the metadata-extractor\src\xml folder create an XSLT file for the given format. This XSLT determines the structure of the xml that will be created by the adapter. A file by name **mpeg4_to_nlnz_presmet.XSLT** is created for the MPEG4 adapter. The following snippet shows the content of this XSLT.

4. execution scripts

The scripts extract.bat, extract.sh, metadata.bat, metadata.sh located in the folder metadata-extractor\src\scripts has to be changed to place the necessary jar files for this adapter into the classpath.

5. Config.xml

The config.xml located in the folder metadata-extractor\src\xml has to be modified in the following three locations

i `<adapters>`

The config file has to be updated to have the entries for the new adapter. This makes the adapter available for the application. The element <adapters> should have a new sub element <adapter> describing the new adapter. This step maps the adapter class from its jar to the format.dtd for a given adapter. For example in case of the MPEG4 the following lines are added.

```
<adapters>
….
<adapter Class="nz.govt.natlib.adapter.mpeg4.MPEG4Adapter"
jar="mpeg4_adapter_1_0.jar" output-dtd="mpeg4.dtd" />
….
</adapters>
```

ii `<XSLT-map>`

The mapping of the dtd, XSLT and the output xml is done in this tag. In case of MPEG4 the following lines are added.

```
<map>
    <input-dtd doc-name="mpeg4.dtd" />
    <output-dtd doc-name="nlnz_presmet.xsd" />
    <xslt doc-name="mpeg4_to_nlnz_presmet.xslt" />
</map>
```

iii `<profiles>`

The adapter profile has to be defined here to let the MDE tool create the relevant logs and be able to use the adapter individually. In case of MPEG4 the following lines are added under <Profiles> element.

```
<profiles default="Default">
  <profile name="Default">
      <input-files dir="METADATA_BASE" />
      <log-dir dir="METADATA_BASE/logs" />
    ....
    ...
    <adapter
    class="nz.govt.natlib.adapter.mpeg4.MPEG4Adapter"
    />
  </profile>
</profiles>
```