org.jax.mgi.shr.ioutils Design Document

Author: Mike Walker

Created: March 20, 2003

Last Modified: November 7, 2005 14:18

1 Purpose of Document

This document describes the classes belonging to the org.jax.mgi.shr.ioutils package and provides source code examples for common usage patterns.

2 Introduction

The purpose of the ioutils package is to provide a set of classes for reading and writing data files in text format and additionally for reading data files in xml format. This document is divided into three main sections. The first section covers only those classes involved in text file parsing. The second section covers xml data parsing and the third section covers text file output.

This package integrates with the following packages: org.jax.mgi.shr.config , org.jax.mgi.shr.exception and org.jax.mgi.shr.log.

You must be pointing to Java1.4 in your classpath when using this product. The **InputDataFile** uses the Java1.4 nio package (new io). This provides performance improvements over previous java implementations by replacing the stream paradigm with the buffer paradigm. Additionally Java Regular Expressions are used from Java1.4.

3 Common Usage Patterns

There are common usage patterns utilized within this java package for both text file parsing and xml file parsing. One pattern involves the act of iterating over data elements from within an input file, whether the file is in text or xml format. The Iterator pattern involves the use of a next() and a hasNext() method and is commonly seen throughout the java language and various java libraries. The ioutils classes utilize this pattern for iterating through records from a text file or xml elements from a xml file. In the case of text file parsing, the **RecordDataIterator** iterates over records and in the case of xml file parsing, the **XMLDataIterator** iterates over xml elements.

Another common pattern is the Interpreter pattern. This involves taking the raw input, whether it is xml or text format, and creating a java object from the record or xml element. In the case of text parsing, the **RecordDataInterpreter** is used and in case of xml parsing, the **XMLDataInterpreter** is used.

These two patterns are typically combined, so that an Interator has an Interpreter and instead of iterating over raw text or xml, the Interpreter is used by the Iterator to convert the raw data to a

Java object. So, from the application perspective the Iterator is return Java objects on the call to the next() method.

4 Classes for Text File Parsing

4.1 Purpose

The purpose of the text file parsing classes is to provide a way to iterate over records from a text file and convert the records into Java application objects. Additionally there are ways to configure these classes from a configuration file to control things like begin and end record delimiters, buffer sizes, use of regular expressions in the delimiters and the type of input character set.

4.2 Overview of Classes

The **InputDataFile** class represents an input file. One instance is created per file. It provides a **RecordDataIterator** for the file.

The **RecordDataIterator** is an interface modeled after the java Iterator interface and provides method definitions used for iterating through the records of a data file. It is obtained from the **InputDataFile**. The **InputDataFile** has an inner class which implements this interface and provides instances of it on the call to the getIterator() method. This inner class uses a lower level class called the **RecordDataReader** for reading record records from a file.

The **RecordDataReader** object reads records from a file which are delimited by begin and end delimiters which can be byte arrays or regular expressions. This class uses the Java 1.4 nio classes for obtaining higher performance when parsing large data files such as genbank files.

The **RecordDataInterpreter** is an interface for creating java data objects based on the record data from the input files. A user could create a class which implements this interface and pass it to the **RecordDataIterator** so that records in the data file can be iterated through and returned as java data objects.

The **InputDataFileCfg** class is used to read configuration parameters from a configuration file or files and the java command line arguments in order to initialize an instance of an **InputDataFile**.

4.3 Class Diagram

4.4 Configuration

Configuration is accomplished by use of the org.jax.mgi.shr.config package. A configurator class is included in that package which is called **InputDataFileCfg**. This class is used to handle the reading of configuration parameters. Details of how this is done can be found in the system documentation for the config package, but in brief, the configuration parameters are read in from configuration files and the java system properties. The configuration file names are indicated on the java command line by use of the CONFIG system property (see example 1). Configuration parameters are accessed through the **InputDataFileCfg** object which provides getter methods (see example 2). You can use multiple **InputDataFileCfg** objects to configure more than one **InputDataFile** objects. Parameter prefixing is used to distinguish the configuration parameters intended for each instance. By prefixing the configuration parameters with a chosen string, a grouping of configuration parameters is established which can then be referred to by providing the prefix string as a parameter to the **InputDataFileCfg** constructor. An example configuration file which show two sets of parameters is shown in Example 3. The instantiation of the configuration objects for this configuration file is shown in Example 4.

EXAMPLE 1. setting the configuration

```
java -DCONFIG=configfile -DPARM1=OVERRIDE -DPARM2=OVERRIDE2 <app>
```

EXAMPLE 2. using a InputDataSourceCfg object

```
// the constructor will read all configuration files
// and all java command line parameters

InputDataFileCfg config = new InputDataFileCfg();

// values are now accessible

String filename = config.getInputFileName();
```

EXAMPLE 3. configuration file with parameter prefixing

```
SOURCE1_INFILE_NAME=file1
SOURCE1_INFILE_DELIMITER=/|/|
SOURCE1_INFILE_BUFFERSIZE=51200000

SOURCE2_INFILE_NAME=file2
SOURCE2_INFILE_DELIMITER=^/
```

EXAMPLE 4. creating InputDataSourceCfg objects from file in Example 3

```
InputDataFileCfg config1 = new InputDataFileCfg('SOURCE1');
InputDataFileCfg config2 = new InputDataFileCfg('SOURCE2');

InputDataFile source1 = new InputDataSource(config1);
InputDataFile source2 = new InputDataSource(config2);
```

If you are not using a configuration file or command line arguments then an **InputDataFile** can be created using a constructor which accepts the filename as a method parameter. The other attributes can be set by setter methods. The following summarizes these attributes:

INFILE_NAME

The name of the data input file. By default the class will try and open a file in the current directory named 'input'. A special character string, 'STDIN' is recognized in order to enable reading from standard-in instead of a file.

INFILE_BEGIN_DELIMITER

The string used to identify the beginning of a record. This string will be included within the record. A value of null can be applied which is interpreted as no begin delimiter. The default is such that each line is treated as a separate record. This default is applied only if both the begin and end delimiters are not set.

INFILE_END_DELIMITER

The string used to identify the ending of a record. This string will be included within the record. A value of null can be applied which is interpreted as no end delimiter. The default is such that each line is treated as a separate record. This default is applied only if both the begin and end delimiters are not set.

INFILE_USE_REGEX

Indicator of whether or not to use regular expressions for the begin and end delimiters. If this value is false then regular expressions are not used. The default is true.

INFILE_BUFFERSIZE

The internal buffer size to use. The default value is 512000 bytes. By increasing this value you may be able to gain a performance boost at the expense of higher internal memory usage.

INFILE_CHARSET

The character set to use when decoding bytes sequences from the input file. The default is ISO-8859-1.

An alternate way of configuring the InputDataFile class is to use the setter methods directly. There is a corresponding setter method for each configuration parameter, except for **INFILE_NAME**, which is alternatively set through the constructor. The configuration methods are setBeginDelimiter(String), setEndDelimiter(String), setOkToUseRegex(boolean), and setBufferSize(int).

4.5 Record Delimiters

The record delimiters are used to distinguish the start and end of records within the input file. These delimiters can be java regular expressions. This gives more expression possibilities but hampers performance. Alternatively, these delimiters can be interpreted as byte arrays. This approach is faster, but sometimes it is not possible to express the delimiter without using regular expressions. There are two caveats when using regular expressions. The term must exist on a line of its own and if one of the characters of your delimiter is a regular expression metacharacter itself then you must apply a backslash to it so it is not interpreted by the regular expression compiler. Additionally, if you are setting the delimiter within the java code (through a setter method on the **InputDataFile** object), then you must be careful to apply a backslash character to any java metacharacter so that java does not interpret it while parsing the String. For example, if your delimiter is "||", then since "|" is a regular expression metacharacter, the delimiter should be set as "\|\|" in the configuration file. And if this was set within the java code as a String then the backslash must be backslashed since the backslash is a java metacharacter. So the code in the source file would read String s = new String("\\|\\|"). The default is set so that each line is considered a record if nei-

ther a begin nor an end delimiter is specified. Regular expressions are used by default. To turn this off in order to use byte arrays, set the **INFILE_USE_REGEX** to false within the configuration or call the setOkToUseRegex(false) on the InpurDataFile class.

4.6 Reading Records from an Input File

In the ioutils package, reading records from an input file is designed around the iterator pattern. An interface called **RecordDataIterator** has been defined for this purpose. It is as follows:

```
public interface RecordDataIterator
{
  public void setInterpreter(RecordDataInterpreter interpreter);

  public boolean hasNext();

  public java.lang.Object next() throws MGIException;
}
```

The setInterpreter() method will be discussed shortly. The hasNext() and next() methods are used for iterating through the input records (see Example 5).

EXAMPLE 5. common use of a RecordDataIterator

```
while (iterator.hasNext())
{
  String record = (String)iterator.next();
  // process record
}
```

Note that in Example 5 a cast was made to the return of the next() method. The method is defined with a return type of java.lang.Object. This is the base class of all java objects, so an object of any type can be returned. By default the return type is String, but with the use of a **RecordDataInterpreter**, any java return type is possible. You will be required to cast these return types to their appropriate types. For example, when reading records from a Genbank input file, a predefined Sequence object can be returned for each iteration. The setInterpreter() method is used to set what **RecordDataInterpreter** is to be used for creating these objects. The **RecordDataInterpreter** is an interface. It is defined as follows:

```
public interface RecordDataInterpreter
{
  public Object interpret(String in) throws MGIException;

  public boolean isValid(String s);
}
```

Once this class has been written, you would plug it into the iterator as in example 6.

EXAMPLE 6. using a RecordDataInterpreter

```
// create the manager
InputDataSource input = new InputDataFile('filename');
// instantiate a class which implements the
// RecordDataInterpreter interface
```

```
SomeInterpreterClass interpreterClass = new SomeInterpreterClass();
// get a iterator and pass in the interpreter class
RecordDataIterator it = input.getIterator(interpreterClass);
// iterate through the file
// which is now returning some other object instead of a String
while (it.hasNext())
{
  // the object returned is determined by the interpreter class
  SomeObject o = (SomeObject)it.next();
}
```

To implement the **RecordDataInterpeter** interface, you would write a class and provide the interpret() method and the isValid() method. And declare the class as implementing the **RecordDataInterpreter** interface using the implements clause. In the interpret() method you would apply the parsing logic for the given record which is provided as a String and proceed to call setter methods on the java object you are creating. This object would then be returned as the return parameter of interpret(). If any error occurs, such as a badly formatted record, then you would throw a **RecordFormatException**. The isValid() method is used to determine whether or not a record should be used during iteration. For example you might opt to not use comment lines. Implementing isValid() entails evaluating the given String and returning true or false. The hasNext() method of the RecordDataIterator will continuing iterating through the data records until it gets a return of true from the isValid() method or it gets to the end of the file (see Example 7).

EXAMPLE 7. implementing a RecordDataInterpreter

```
public class className implements RecordDataInterpreter
{
   public Object interpret(String s) throws RecordFormatException
   {
     instantiate a new object

     parse the string and call setter methods on the new object

     return the new object
   }

   public boolean isValid(String s)
   {
     return true; // this will return all records regardless
   }

   // more methods defined if required
}
```

5 Classes for XML File Parsing

5.1 Purpose

The purpose of the xml parsing classes is to provide a way to iterate over xml input data and convert xml data into java application objects. Additionally, there ways to configure these classes through configuration files to control aspects like the name of the XML tag to iterate over.

5.2 Overview of Classes

The **InputXMLDataFile** represents an input file in xml format. There are one instance of this class for each input file. An **XMLDataIterator** is obtained from this class for iterating over the data elements.

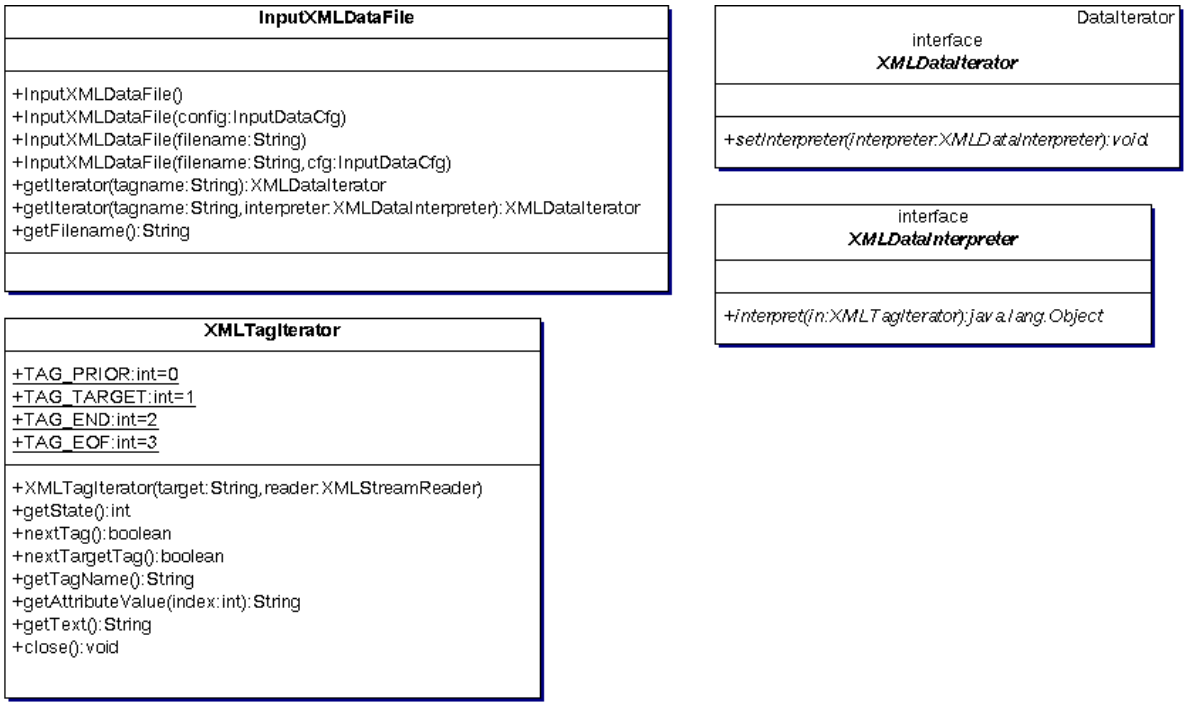
The **XMLDataIterator** provides a way to iterate over the elements of the input file. A next() and a hasNext() method is provided to the application code for doing this. The configuration variable **INFILE_XML_ITERATOR_TAG** sets the name of the xml to iterate over. A **XMLDataInterpreter** can be provided to the **XMLDataIterator** so that Java objects are returned to the application layer instead of raw xml data.

The **XMLTagIterator** represents the raw xml data. The **XMLDataIterator** iterates over these tags by using a **XMLTagIterator** and it is responsible for positioning to the appropriate tag and returning either the instance of the **XMLTagIterator** to the application or, in the case that it has an **XMLDataInterpreter**, a newly created Java application object is returned. The **XMLTagIterator** views an xml file as consisting of a simple sequence of tags and each tag consist of just a tag name, a tag value and a set of tag attributes (all of which can be accessed though the interface to the **XMLTagIterator**).

The **XMLDataInterpreter** is used for converting raw xml data into Java objects. The **XMLDataIterator** can be provided an instance of this class so that it can convert raw data “on the fly” while iterating over the input.

The **InputDataFileCfg** class is used to read configuration parameters from a configuration file or files and the java command line arguments in order to initialize an instance of an **InputXMLDataFile**.

5.3 Class Diagram



5.4 Configuration

An overview of configuration was provided in section 4.1. The configuration parameters which apply to xml parsing are as follows:

INFILE_NAME

The name of the data input file. By default the class will try and open a file in the current directory named ‘input’. A special character string, ‘STDIN’ is recognized in order to enable reading from standard-in instead of a file.

INFILE_XML_ITERATOR_TAG

The target tags to iterate over when reading an input xml file using an XMLDataIterator.

5.5 Parsing XML Tags Using the XMLTagIterator

The **XMLTagIterator** represents an XML file as a series of target tags which can be iterated over one by one. The target tags and the nested tags within them represent the data items that will be mapped to Java objects. The **XMLTagIterator** provides two methods for navigating through these tags, nextTargetTag() for moving to the next target tag and nextTag() for moving to the next nested tag within a target tag. A call to nextTag() does not move the pointer at all if it is already

positioned on the last child tag of the target tag. To know whether you are on the last tag you can use the getState() method. There are three methods for reading the XML data. These are getTagName(), getAttributeValue(int index) and getText(). A sample code snippet is provided below to demonstrate the use of this class.

```
PIRSFSuperFamily sf = new PIRSFSuperFamily();
try
{
    sf.recordID = it.getAttributeValue(0);
    it.nextTag();
    String store = null;
    while (it.getState() != it.TAG_END)
    {
        if ("pir-id".equals(it.getTagName()))
            sf.pirID = it.getText();
        else if ("pir-name".equals(it.getTagName()))
            sf.pirName = it.getText();
        else if ("mgi-id".equals(it.getTagName()))
            sf.mgiID = it.getText();
        else if ("trembl-ac".equals(it.getTagName()))
            sf.trembl.add(it.getText());
        else if ("sprot-ac".equals(it.getTagName()))
            sf.sprot.add(it.getText());
        else if ("refseq-ac".equals(it.getTagName()))
        {
            store = it.getText();
            if (store.indexOf(";") > 0)
            {
                String[] fields = store.split(";");
                if (!fields[0].startsWith("YP"))
                    sf.refseqID.add(fields[0]);
            }
            else if (!store.startsWith("YP"))
            {
                sf.refseqID.add(store);
            }
        }
        else if ("locus-id".equals(it.getTagName()))
            sf.locusID = it.getText();
        else if ("locus-name".equals(it.getTagName()))
            sf.locusName = it.getText();
        else if ("source-org".equals(it.getTagName()))
        {
            sf.source = it.getText();
            if (!sf.source.equals(TARGET_SOURCE))
                return null;
        }
        it.nextTag();
    }
}
catch (IOUException e)
{
    throw new InterpretException(
            "Cannot read data from xml", e);
}
return sf;
```