LexEVS 6.x Loader Implementation

Contents of this Page

LexEVS 6.x Programmers Links

- Programmer's Guide Main Page
  - LexEVS API
  - LexEVS 6.0 CTS2 API
  - LexEVS 6.x CTS2 API Quick Start
- Value Set and Pick List Guide
- LexEVS 6.0 Main Page
- LexEVS Current Release

Overview

LexEVS provides classes to extend and interfaces to implement that help provide the framework for in-memory transformations of source files into the LexGrid model. If the target source is large, batch loading may well be a better solution, in which case the Loader Framework built on Spring Batch may be a better match for source loading. On the other hand, users may find the standard LexEVS loader interface a little easier to implement.

Writing loaders requires both programming skill and content expertise. Mapping source into the LexGrid data model requires knowledge of how the source defines entities, relationships between entities and any qualifiers or properties for these elements. However, some study of other presentations of that content, and documentation of the source files, can help most programmers make informed choices as to that mapping. Examples of source mapped into LexGrid are documented here.

Where to Start

The LexEVS API programming environment is currently available in a github repository here: LexEVS Source Code

Cloning this repository produces a large set of eclipse friendly projects that can be imported into Eclipse or Intellij.

Interfaces and Abstract classes to Extend

Loader interfaces must extend the interface *org.LexGrid.LexBIG.Extensions.Load.Loader* from the *lbInterfaces* project and implementations of your new interface must also extend the abstract class *org.LexGrid.LexBIG.Impl.loaders.BaseLoader* in the *lbImpl* project. Let's take as an example the MedDRA loader interfaces and classes as it is a relatively simple implementation.

The extension of *Loader* is *org.LexGrid.LexBIG.Extensions.Load.MedDRA_Loader.java* and is found in the *lbInterfaces* project.

Extend Loader

```
public interface MedDRA_Loader extends Loader {
    public final static String name = "MedDRALoader";
    public final static String description = "This loader loads MedDRA files into the LexGrid format.";
```

The implementation of MedDRA_Loader (and extension of BaseLoader) is *org.LexGrid.LexBIG.Impl.loaders.MedDRALoaderImpl*.

Extension of Baseloader and Implementation of MedDRA_Loader

```
public class MedDRALoaderImpl extends BaseLoader implements MedDRA_Loader
```

This defines load and validation methods and creates initialized class attributes *name* and *description*. The name of this interface will eventually be used by the LexEVS extension function to call the loader into existence. (Loaders are always extensions to LexEVS).

Implementation Example and Discussion

Call the Super Constructor

Looking at the implementation of this interface in *lbImpl*, *org.LexGrid.LexBIG.Impl.loaders.MedDRALoaderImp*l, notice that implementation is kept relatively clean thanks the fact that much of the mechanism of loading into LexEVS is taken care of under the covers by the *BaseLoader* class. *MedDRALoaderImpl* creates a constructor that always calls the *BaseLoader* constructor, then prevents the use of manifests. (If you wish you may choose to allow manifest use, since it allows load time manipulation of coding scheme metadata. Often source files do not provide all that needs to be known about the source. A manifest file allows values that may not be present in the source such as copyrights, authoring information, version definition and formal coding scheme name to be added to the load.)

Loader Constructor

```
    public MedDRALoaderImpl() {
        super();
        this.setDoApplyPostLoadManifest(false);
    }
```

Preventing the use of a manifest is not typical loader behavior.

Define the Extension

The *buildExtensionDescription* method provides a background method for the registration of this loader to take place within LexEVS. It should be created the same way for each loader.

Extension definition

```
    @Override
    protected ExtensionDescription buildExtensionDescription(){
        ExtensionDescription temp = new ExtensionDescription();
        temp.setExtensionBaseClass(MedDRALoaderImpl.class.getInterfaces()[0].getName());
        temp.setExtensionClass(MedDRALoaderImpl.class.getName());
        temp.setDescription(description);
        temp.setName(name);
        temp.setVersion( getMedDRAVersion() );
        return temp;
    }
```

Validate the Source (Optional)

The *validate* method is not always implemented, but can and should be when a mechanism exists to insure that this is a correctly structured version of the source. When source is XML formatted and has a schema or dtd document commonly available for validation, this is a relatively easy process. However any free text files that do not have any associated java-based validation API, would rely on the developer to create validation functions. The validation method is not required to create a loader for LexEVS.

Validate the Source

```
 @Override
    public void validate(URI sourceDir, int validationLevel) throws LBParameterException {
```

Pass in the URI for the path to the source file and, if desired, provide an integer value for a level of validation for this source. (I.e. what level of errors or issues can this load live with. This assumes a very detailed validation of source)

Implement the Abstract Method *doLoad()* (and possibly override the *load()* method)

The developer could at this point implement only *doLoad* by mapping content to a coding scheme object before pass control over to *BaseLoader* which will call the *doLoad* method and set default load options in its load method and kick off the processes to persist a coding scheme object to the database. The other option is to implement *doLoad* and override the *load* method which sets up end user option choices for the loader. Most loaders implement the *load* method, customizing load options to provide to the end user. In the case of the MedDRA loader, a CUI load option is provided to the end user.

Load Kickoff Implementation in doLoad

```
    @Override
    protected URNVersionPair[] doLoad() throws Exception{
        LgMessageDirectorIF messages = new CachingMessageDirectorImpl(this.getMessageDirector());
        MedDRA2LGMain mainTxfm = new MedDRA2LGMain();
        //Set the source for the CUI file when using the GUI
        if(UMLSCUISource == null){
            UMLSCUISource = this.getOptions().getURIOption(UMLSCUI_FILE_OPTION).getOptionValue();
        }
        CodingScheme codingScheme = mainTxfm.map(UMLSCUISource, this.getResourceUri(), this.
getMessageDirector());

        if(codingScheme != null){
            messages.info("Completed mapping.  Now saving to database");
            this.persistCodingSchemeToDatabase(codingScheme);

            messages.info("Saved to database.  Now constructing version pairs");
            return this.constructVersionPairsFromCodingSchemes(codingScheme);
        }

         return null;
    }
```

This allows the BaseLoader functions to access the results of the load of the MedDRA source file into a LexEVS coding scheme object.

Override the load Method

```
    @Override
    public void load(URI uri, URI cuiUri, boolean stopOnErrors, boolean async) throws LBParameterException,
            LBInvocationException {
        this.getOptions().getBooleanOption(FAIL_ON_ERROR_OPTION).setOptionValue(stopOnErrors);
        this.getOptions().getBooleanOption(ASYNC_OPTION).setOptionValue(async);
        this.getOptions().getURIOption(UMLSCUI_FILE_OPTION).setOptionValue(cuiUri);
        UMLSCUISource = cuiUri;
        this.load(uri);
    }
```

This method adds a UMLS CUI source file option for the end user to load as a supplement to the regular MedDRA load.

Less Structure Beyond the *Loader, BaseLoader* Implementation and Extension

Beyond these methods, the structuring of code is largely and necessarily left up to the developer. However a few common patterns are fairly consistent in this implementation. Generally speaking, there is a central mapping class where the coding scheme creation is kicked off. Other classes, when necessary, are supportive to this central class.

Tasks to be Accomplished

These classes are generally classified by those that are responsible for either reading a source file or accessing an API that reads the file for you and those that map objects created from that file into LexEVS coding scheme metadata, entity and relationship objects.

In summary:

- Read source
- Map to objects related to source structure
- Map from source structure objects to LexEVS coding scheme object

Mapping Entry Point

In the *lbConverter* project the *edu.mayo.informatics.lexgrid.convert.directConversions.medDRA* package contains the classes that do much of the work of the MedDRA load into LexGrid. *edu.mayo.informatics.lexgrid.convert.directConversions.medDRA.MedDRA2LGMain* provides a central kickoff point with some methods that can be wrapped for load and validation responsibilities down further up the execution chain.

Main Entry Point to Loader Code

```
public class MedDRA2LGMain {

        //providing parameters for the source directory, the UMLS CUI file and a logging object from LexEVS
           public CodingScheme map(URI cuiUri, URI sourceDir, LgMessageDirectorIF lg_messages)

        //Validate choice for the MedDRA source only
        private boolean validateSourceDir(URI sourceDir)
```

Read the Data to Source Model Objects

This package also contains *edu.mayo.informatics.lexgrid.convert.directConversions.medDRA.MedDRAMapToLexGrid* with a *readMedDRAFile* method that reads from a CSV file and persists it to objects defined in package *edu.mayo.informatics.lexgrid.convert.directConversions.medDRA.Data*. While this data package defines beans that model the content of lines read from the CSV file, it also organizes them into structures that are easier for the mapping code to use. The individual manner of implementation is left up to the developer but this is a good example of how a third party library was used to process the file and how the resulting objects were stored in a data structure that's easy for the mapping code to consume.

Read the Source File

```
    public void readMedDRAFiles() {
        String input;

        for(int i=0; i < meddraMetaData.length; i++){
            input = medDRASourceDir.getPath() + meddraMetaData[i].filename();
            try {

                FileReader fileReader = new FileReader(input);
                CSVReader reader = new CSVReader(fileReader, '$');
                ColumnPositionMappingStrategy<DatabaseRecord> strat = new
ColumnPositionMappingStrategy<DatabaseRecord>();
                strat.setType(meddraMetaData[i].classname());
                String[] columns = getFields(meddraMetaData[i].classname());

                strat.setColumnMapping(columns);

                CsvToBean<DatabaseRecord> csv = new CsvToBean<DatabaseRecord>();
                List<DatabaseRecord> list = csv.parse(strat, reader);
                meddraDatabase.add(meddraMetaData[i].tablename(), list);
            } catch (FileNotFoundException e) {
                messages_.error("MedDRA input file missing.", e);
            } catch (Exception e) {
                messages_.error("Failed to read MedDRA files.");
            }
        }
    }
```

CSVReader is a third party CSV reader.

Map the Model Objects to LexGrid Model Objects

Once the source is read and persisted to the appropriate model objects the *MedDRAMapToLexGrid* class can map these data objects derived from the MedDRA source into a complete coding scheme object.

Map to Coding Scheme

```
    public void mapToLexGrid(CodingScheme csclass) {
        try {
            loadCodingScheme(csclass);
            loadConcepts(csclass);
            loadRelations(csclass);
        } catch (Exception e) {
            messages_.error("Failed to map MedDRA data to LexEVS.");
        }

        messages_.info("Mapping completed, returning to loader");
    }
```

Pass Control Back to BaseLoader

The tasks are to read the file into some kind of logical model or bean class object, organize these objects or make them available to be mapped into LexEVS objects, tie all objects together as a coding scheme, and pass this potentially large coding scheme object to the LexEVS *BaseLoader* to be persisted to the database. Back up the execution chain in *MeDRALoaderImpl* the doLoad method first calls the mapping method of the conversion class *MedDRA2LGMain* to get the reading and mapping done, then passes control of the resulting coding scheme to *BaseLoader* by calling *this.persistCodingSchemeToDatabase* method on the *BaseLoader* super class.

Persistance is handled in the BaseLoader

```
    if(codingScheme != null){
            messages.info("Completed mapping.  Now saving to database");
            this.persistCodingSchemeToDatabase(codingScheme);

            messages.info("Saved to database.  Now constructing version pairs");
            return this.constructVersionPairsFromCodingSchemes(codingScheme);
        }
```

While it's not required, most loaders have been written in this way.