

The LEXAN API and the LEXAN source distribution

The LEXAN API and the LEXAN source distribution

Version 1.0, March 2020

Table of Contents

Introduction	iv
1. The LEXAN source distribution	1
The structure of the sources as a Maven project	1
Building the API	1
Generating Javadoc documentation	2
2. The API interfaces and objects	3
The lexan-api source structure	3
The TextAnalyzer interface	3
The TextAnalyzerContext interface	5
API packages and classes	6
3. Implementing an Analyzer	8
Build as a separate project	8
Build as a new module in the LEXAN project	8
Testing the analyzer	8

Introduction

This is still an **incomplete** documentation of the LEXAN API and the LEXAN source distribution. LEXAN is a mechanism to extend ELAN with natural-language processing (NLP) or more general text processing functionality. It was once thought of as an interfacing component between ELAN and LEXUS (the TLA lexicon tool), but evolved to an API for extending ELAN when development of LEXUS stopped and a (simple) LexiconComponent was added to ELAN.

Apart from the sources of the API, this distribution also contains the sources of the Analyzer implementations created by MPI. The API models the interfacing between a text analyzer and a host application, i.e. ELAN. A compiled version of the API is part of the ELAN distribution in the form of a `lexan-api-x.x.jar`. Analyzer implementations should compile against this jar, but the sources of the API are in the LEXAN source distribution and a convenient way of developing an analyzer is by adding a module to the project in the source distribution.

There are two main flavors of text analyzers, those that require access to an ELAN lexicon and those that don't. The former can implement the `TextAnalyzer` interface, the latter should implement the `LexiconTextAnalyzer` interface.

Chapter 1. The LEXAN source distribution

The structure of the sources as a Maven project [<https://maven.apache.org>]

The LEXAN sources are setup as a Maven project with modules, the `lexan-api` being one of the modules. In the root of the project folder there is the `pom.xml`, the Maven build script. Each module is in a sub-folder containing the module's `pom.xml` and a `src` folder. So the folder hierarchy is like this:

- **lexan-x.x**
 - `pom.xml`
- **lexan-api**
 - `pom.xml`
 - `src`
- **analyzer-gloss**
 - `pom.xml`
 - `src`
- **annotyzer-whitespace-split**
- ...

The parent `pom.xml` lists the modules and sets the version of the `lexan-api` as a common dependency.

Building the API

The LEXAN API jar and the analyzers can be build from the command line. When in the root folder use the command:

```
mvn package
```

or

```
mvn clean package
```

After BUILD SUCCESS the api library `lexan-api-x.x.jar` can be found in the `lexan-api/target` folder. It is also possible to only build the API library by changing directory to `lexan-api` and run the command `mvn package` from there or by removing all other modules from the parent `pom.xml`.

Note

ELAN is no longer compatible with Java 1.6. The current distributions contain a Java 11 or higher JRE. The parent `pom.xml` configures the source level to a minimum of Java 9.

Generating Javadoc documentation

HTML Javadoc documentation of the API is not included in the source distribution but can be generated from the sources. To do so on the command line `cd` into the `lexan-api` folder and execute the command:

```
mvn javadoc:javadoc
```

After completion the documentation can be found in the sub-folder `target/site/apidocs`

Note

The `mvn javadoc:javadoc` command requires Java 1.7 or higher.

Chapter 2. The API interfaces and objects

The `lexan-api` source structure

The `lexan-api` source folder contains interfaces that have to be implemented by analyzers and some helper classes that can or should be used by each analyzer. Here is a simplified representation of the structure of the `lexan-api` source code:

- **lexan-api**
 - pom.xml
 - src
 - **main/java/nl/mpi/lexan/analyzers** (collapsing sub-folders into one line)
 - **helpers**
 - **lexicon**
 - LexiconTextAnalyzer.java
 - TextAnalyzer.java
 - TextAnalyzerContext.java
 - TextAnalyzerLexiconContext.java

The TextAnalyzer interface

The main interface for an analyzer to implement is `TextAnalyzer` or `LexiconTextAnalyzer`, the latter in case the analyzer wants to access an ELAN lexicon. An analyzer communicates with the host application via an instance of the section called “The TextAnalyzerContext interface” `TextAnalyzerContext` and/or the section called “The TextAnalyzerContext interface” `TextAnalyzerLexiconContext`. These interfaces are implemented by ELAN, being the host application for the analyzer plug-in, and are described in the next section.

The main methods of `TextAnalyzer` to be implemented by an analyzer:

- `getInformation()` - should return a `nl.mpi.lexan.analyzers.helpers.Information` object containing the name, a description (optional) and a list of `nl.mpi.lexan.analyzers.helpers.parameters.Parameter` objects that inform the host context of e.g. the type and number of source and target tiers (optional). If no `Parameters` are provided, ELAN assumes one source and one target tier, the target being a child tier of the source.
- `setAnalyzerContext(TextAnalyzerContext tc)` - connects the analyzer to the host context. The analyzer needs this context to deliver the results of its processing tasks.
- `setCacheFolder(File f)` - informs the analyzer of the location of the general, shared folder where analyzers can store data persistently. The analyzer can create and use its own sub-folder within that folder.
- `getConfigurationComponent(String s)` - can return a user interface component which allows the user to configure the behavior of the analyzer (optional). The component (usually a

`JPanel`) will be added to an analyzer configuration dialog by the host context (ELAN). The `String` argument identifies a particular source-target configuration for which to edit the settings, `null` indicates global settings.

- `isConfigurable()` - returns whether or not the analyzer supports configuration of settings. This is a shorthand for the host context so that it doesn't have to call `getConfigurationComponent(String s)` (which might trigger the creation of panel) unnecessarily.
- `load(List<SourceTargetConfiguration> lc)` - provides a list of source tier to target tier(s) combinations to the analyzer. The tiers are passed wrapped in a `nl.mpi.lexan.analyzers.helpers.Position` object, that identify the tiers by their name (`String`). The analyzer needs this information, when analyzing contents from a source tier, to find the correct target tier(s) to deliver results to.
- `partLoad(List<SourceTargetConfiguration> lc)` - same as `load` but for loading additional configurations.
- `partUnload(List<SourceTargetConfiguration> lc)` - counterpart of `partLoad(...)`, called when the analyzer should no longer analyze for the given configurations. Can be called as part of the process of closing the analyzer(s) and the entire document. The analyzer can store cached information and free resources.
- `unload()` - counterpart of `load(...)`, also see `partUnload()`.
- `saveState()` - informs the analyzer to store its internal state, settings, cached data etc.
- `loadState()` - informs the analyzer to restore data that has previously been stored.
- `loadSettingsFor(String s)` - tells the analyzer to load stored settings for a specific source-target configuration. `null` indicates global settings. If there are no stored settings (yet), default settings can be initialized.
- `analyze(Position p)` - asks the analyzer to analyze the contents at `Position p`, so the actual work of the analyzer. In general the analyzer has to perform four steps when this method is called:
 - check if `p` corresponds to one of the source positions in the list of source-target configurations (i.e. if the `tierId` of `p` equals the `tierId` of one of the source positions).
 - call the `readAnnotation(Position p)` method of the `TextAnalyzerContext`. The specified position contains the name of a tier and the start and end time of a segment. In most cases this corresponds to (the position of) one annotation and the analyzer gets access to the annotation via this method. The method returns a list of annotations wrapped in `Suggestion` objects, the list will in most cases contain one item.
 - process the annotation. The actual work, the type of processing the analyzer is designed for (parsing, tagging, splitting etc.).
 - return the results to the host context by calling one of the `newAnnotation()` methods of `TextAnalyzerContext` (see there). The results need to be wrapped in `Suggestion` or `SuggestionSet` objects for the target tier(s) as defined in the source-target configurations.

The `LexiconTextAnalyzer` interface extends `TextAnalyzer` and the main methods it adds are:

- `setLexiconContext(TextAnalyzerLexiconContext lc)` - connects the analyzer to the lexicon host context.
- `setLexiconLinkForPosition(Position p, LexanLexicon ll, String s)` - informs the analyzer that the tier specified in `p` is linked to field `s` in lexicon `ll`.

- `removeLexiconLinkForPosition(Position p, String s)` - informs the analyzer that the tier in `p` is no longer linked to entry field `s`.

The TextAnalyzerContext interface

The main methods of `TextAnalyzerContext` are:

- `newAnnotation(Position p, Suggestion s)` - passes one suggestion for a new annotation at the location on the target tier specified by `p`. The new annotation will immediately be created, if possible, without user interaction. Usage example: look up of the input with only one possible result.
- `newAnnotation(Position p, List<Suggestion> l)` - passes a list of suggestions for annotations to be created at the location specified by `p`. The list represents a single sequence of annotations, not a set of alternatives, the annotations will be created immediately (if possible) without user interaction. Usage example: segmentation of the input into smaller units, e.g. a sentence divided into words.
- `newAnnotations(List<SuggestionSet> l)` - passes a list of `SuggestionSet`s which are alternatives for the user to choose from. Each `SuggestionSet` can contain suggestions for one or more annotations and for one or more tiers. The host context (i.e. ELAN) will create a user interface to present the alternatives to the user. Usage example: decomposition of a word into morphemes combined with part-of-speech tags.
- `remAnnotation(Position p)` - requests the host context to remove the annotation(s) at the specified location.
- `changeAnnotation(Position p, Suggestion s)` - requests the host context to change the annotation at position `p` to the suggested value.
- `readAnnotation(Position p)` - retrieves the annotation(s) at the specified position. This returns a `List<Suggestion>` object (although the annotations are not suggestions but are merely wrapped in that kind of object). The list can be empty if there are no annotations at location `p`.
- `readParentAnnotation(Position p)` - retrieves the parent annotation(s) of the annotation(s) at position `p`. This returns a `List<Suggestion>` object because there can be multiple annotations at the specified position, which might not all have the same parent annotation. Typically this will return a list with one element representing the parent annotation of the annotation at `p`.
- `readSiblingAnnotations(Position p)` - retrieves a list of all annotations that have the same parent annotation as the one at position `p`.
- `prompt(Prompt pr)` - not implemented yet.
- `getConfigurationComponent(Information i, String s, boolean b)` - allows the host to get a configuration component through the context instance. The information object identifies the analyzer, the string parameter a specific configuration (null meaning global settings) and the boolean determines if the component should be created if not already there (cached).
- `addSuggestionSelectionListener(LexanSuggestionSelectionListener sl)` - registers `sl` as listener for suggestion selection events. Such an event will be sent after one of the `newAnnotations()` methods has been called and it will inform the listeners that a suggestion was selected, including which one, or that the suggestions were cancelled without making a selection.
- `removeSuggestionSelectionListener(LexanSuggestionSelectionListener sl)` - removes `sl` from the list of registered listeners.

The `TextAnalyzerLexiconContext` interface extends `TextAnalyzerContext` and adds the following methods (of which only two are implemented so far):

- `getLexiconNames()` - returns a list of known lexicon names.
- `getLexicon(String s)` - returns an instance of `LexanLexicon`.
- `addToLexicalEntry(LexanLexicon l, LexEntry e, LexItem i)` - **Not Implemented Yet** - adds a new item or field to lexical entry `e`.
- `addEntryToLexicon(LexanLexicon l, String s)` - **Not Implemented Yet** - adds a new entry to the lexicon with `s` as the value for the main field of the entry (head word, lexeme).
- `addEntryToLexicon(LexanLexicon l, Map<String, String> ev)` - **Not Implemented Yet** - adds a new entry to the lexicon based on the field-name to value mapping in `ev`.

API packages and classes

The main packages and classes that are part of the API are:

- **Package:** `nl.mpi.lexan.analyzers.helpers`
 - `Position` - contains a tier name, begin time and end time.
 - `PositionLexicon` - extends `Position`, adds fields for a lexicon name and an identifier of a field in a lexical entry.
 - `Suggestion` - contains a `Position`, a content `String`, a list of child suggestions (`List<Suggestion>`) and a lexical entry (`LexEntry`).
 - `SuggestionSet` - contains a list of `Suggestions`, a source `Position` and an optional information label (`String`) to be displayed along with the set in a user interface.
 - `Information` - an object to identify an analyzer, contains a name, a description and an optional list of `Parameters`.
 - `TierNodeType` - an enum with constants for `SOURCE` (tier) and `TARGET` (tier).
 - configuration classes, listeners other helper classes.
- **Package:** `nl.mpi.lexan.analyzers.helpers.parameters`
 - `Parameter` - abstract class, contains a description and a value (`Object`).
 - `TierTypeParameter` - adds a `TierNodeType` to the parameter and a constraint (one of the constants `NO_CONSTRAINT`, `DIRECT_CHILD_OF_PARAMETER`, `ANY_CHILD_OF_PARAMETER` or `SIBLING_OR_CHILD_OF_PARAMETER`). This is used by ELAN in the panel where analyzers can be configured as to the number and type of source and target tiers. This class also has a flag to indicate whether the tier should be connected to a lexicon or lexical entry field.
- **Package:** `nl.mpi.lexan.analyzers.helpers.settings`
 - `AnalyzerSet` - a set of settings/preferences for an analyzer, can contain single `Setting` objects (key - value pairs) and `SettingsGroup` objects (settings grouped together).
 - `SettingsIO` - a singleton class that analyzers can use to store and load settings or preferences in (e.g.) the cache folder provided by the host application.
 - other settings related classes

- **Package:** `nl.mpi.lexan.analyzers.helpers.statistics`
 - `SuggestionMemory` - a class for saving and loading statistics of how often suggestions have been selected by the user. It now also has a `removePermanently(SuggestionSet s)` method to permanently remove the specified output for this input.
- **Package:** `nl.mpi.lexan.analyzers.lexicon`
 - `LexanLexicon` - an interface defining a lexicon that serves as a proxy for a lexicon maintained by the host application (ELAN's Lexicon component). The methods of this interface allow to query the lexicon and retrieve (parts of) lexical entries that match the query. Some of the methods are:
 - `getName()` - returns the name of the lexicon.
 - `getEntryFieldNames()` - returns a list of field names that can exist in entries in this lexicon.
 - `getEntries(LexAtom la)` - requests the lexicon to return all entries that match the query stored in `la`. The query consists of a field name and a value to match. This returns a `List<LexEntry>` in which each entry contains the field of the query and some standard fields.
 - `getEntries(LexAtom la, List<String> fi)` - same as the previous method, only this time all fields that are in the list of field id's `fi` will be present in the returned `LexEntry` objects.
 - `getEntryById(String id)` - returns a single `LexEntry` with the specified id. The entry includes only a minimal set of the fields.
 - `getEntryById(String id, List<String> fi)` - same as previous method, only this time all fields that are in the list of field id's `fi` will be present in the returned `LexEntry`.
 - `LexContainer` - interface providing methods to retrieve `LexItems`; `getLexItems()`, `getLexItem(String qu)` and `getLexItems(String qu)`.
 - `LexEntry` - interface that extends `LexContainer` and adds a `getId()` method.
 - `LexItem` - interface defining one method `getType()` which returns the type or name of the item (a field in a lexical entry).
 - `LexAtom` - implements `LexItem` and contains fields and getters for the type of the item and the value of the item.
 - `LexCont` - implements `LexItem` and `LexContainer` and has a field for the type of this container and a list of `LexItems` it contains. There are getters for both fields and methods to add or remove individual `LexItems` to or from the list.

Chapter 3. Implementing an Analyzer

Build as a separate project

When developing an analyzer in a new Java project (in an IDE, using Maven or Ant or just the command line), the `lexan-api-x.x.jar` doesn't need to be built from the sources. Adding the `.jar` from an ELAN distribution (in the `lib` folder on Windows or Linux, in the `Java` folder on macOS) to the classpath of the project (via the IDE or as a dependency in the Maven `pom.xml` etc.) will suffice. The project needs to contain a class that implements the `TextAnalyzer` or `LexiconTextAnalyzer` interface. After compilation the class(es) need to be packaged in a `.jar` file which can be copied to the `extensions` folder of ELAN.

Build as a new module in the LEXAN project

Another, and maybe more convenient, way of implementing an analyzer is to add a new module to the LEXAN Maven project. For this, a new sub-folder can be added to the `lexan-api` folder, which should contain a Maven `pom.xml` file and a `src/main/java/etc./` folder structure, analogous to e.g. the existing `analyzer-*` folders. The new module can then be added to the `<modules></modules>` section of the parent `pom.xml`. Building the LEXAN sources then also builds and packages the new module.

Since the amount and the level of detail of the documentation, here in this document and in the Javadoc comments of the API sources, is limited, the best approach might be to take an existing implementation as an example to follow. For an implementation of `TextAnalyzer` the `WhitespacesAnalyzer` (which is in the `annotyzer-whitespaces-split` folder) might serve as an example, for a `LexiconTextAnalyzer` implementation the `GlossAnalyzer` (in the `analyzer-gloss` folder) might do the same. These examples can illustrate how to deal with the `SourceTargetConfigurations`, with the `Information` and `ConfigurationComponent` objects and with `Suggestion` and `SuggestionSet` for reading annotations and for returning results.

Testing the analyzer

When the new analyzer has been built and packaged in a `.jar` file, it can be tested by copying it to the `extensions` folder of ELAN (the version corresponding to the `lexan-api`). (Re)launch ELAN, open a file, switch to the Interlinearization Mode (if it isn't already the selected mode) and click the `Edit Configurations` button. In the window that appears, click in a cell in the `Analyzer` column and check if the name of the new analyzer is listed. If so, try to configure a source and a target tier (based on the tier types) and apply the configuration. If you then right click an annotation on a tier that is configured as a source tier and choose `Analyze / Interlinearize` the `analyze(Position p)` method of the new analyzer should be called.