



Australian Government
Department of Defence
Defence Science and
Technology Organisation

NEFTool: System Design

Benjamin Morrall

Command, Control, Communications and Intelligence Division
Defence Science and Technology Organisation

DSTO-TN-0789

ABSTRACT

The text processing team from the Intelligence Analysis discipline has experimented with the viability of using machine-learning models to automatically tag English words with syntax and functional labels within a text document. The NEFTool was developed to assist with testing different machine-learning models. The system has a modular architecture, and was designed to be extensible, allowing support for rapid prototyping and for new functionality to be added as required to support research in text processing.

RELEASE LIMITATION

Approved for public release

Published by

*Command, Control, Communications and Intelligence Division
DSTO Defence Science and Technology Organisation
PO Box 1500
Edinburgh South Australia 5111 Australia*

*Telephone: (08) 8259 5555
Fax: (08) 8259 6567*

*© Commonwealth of Australia 2007
AR- 014-048*

November 2007

APPROVED FOR PUBLIC RELEASE

NEFTool: System Design

Executive Summary

One of the many aspects of Information Extraction (IE) involves finding references to named items of interest (mentions) in a text document. The text processing team in the Intelligence Analysis Discipline is experimenting with the viability of using machine-learning models to discover mentions within a text document with minimal human interaction with the system. The **NEFTool** was developed to provide a test environment to facilitate this research, and to support other text processing applications.

The **NEFTool** has a modular architecture with the separation of data, process and display as a major design goal. As a result it is possible to modify the various components or even add new components to the **NEFTool** without affecting existing functionality.

This report is aimed at readers interested in gaining an insight into the component-based architecture of the **NEFTool**. It provides an outline of the fundamental framework of the **NEFTool**, and describes an implementation of the system used to find references to mentions within a document. The report focuses on the framework and implementation aspects of the system. Details of the IE process are not addressed.

Contents

1. INTRODUCTION	1
2. CORE SYSTEM	3
2.1 Core Layer (dsto.nef.core).....	4
2.1.1 CoreEnvironment	4
2.2 Document Package (dsto.nef.core.document)	5
2.2.1 NEFDocument.....	5
2.2.1.1 getContent(Class<T extends NEFContent> contentClass): T	5
2.2.2 NEFContent.....	6
2.2.3 NEFPrimitive.....	6
2.3 Process Package (dsto.nef.core.process).....	6
2.3.1 NEFProcess.....	7
2.3.2 ProcessFailedException	7
2.4 Server Processing Package (dsto.nef.core.serverprocessing)	8
2.4.1 NEFServer.....	9
2.4.2 NEFSession.....	10
2.4.3 ServerSession.....	11
2.5 Event Package (dsto.nef.core.event)	11
2.5.1 DocumentChangedListener	12
2.5.1.1 Possible use as a MDI system	12
2.5.1.2 documentChanged method	13
2.5.1.3 clearDocument method	14
2.5.2 DestructionListener	14
2.5.3 LoadingStateListener	15
2.5.4 EventController.....	16
2.6 SwingUI Package (dsto.nef.core.swingui)	16
2.6.1 WindowTemplate.....	17
2.6.2 WindowTemplateFactory.....	17
2.6.3 NEFWindowSystem.....	17
2.6.4 SDIWindowSystem	18
2.6.4.1 Opening a NEFSession in a new NEFWindow	19
2.6.4.2 SDIEventController	19
2.6.4.3 WindowCloser Internal Class	20
2.6.5 NEFWindow.....	20
3. TOKEN SYSTEM.....	22
3.1 TokenEnvironment.....	22
3.2 Document Package (dsto.nef.entity.document)	22
3.2.1 TokenContent.....	22
3.2.2 NEFSentence	23
3.2.3 NEFToken.....	23
3.3 Process Package (dsto.nef.token.process).....	24
3.3.1 TextCleanerProcess	24

3.3.2	SentenceBreakerProcess.....	24
3.3.3	PTBTokenizerProcess.....	24
3.4	SwingUI Package (dsto.nef.token.swingui).....	25
3.4.1	JTokenTable.....	25
4.	ENTITY SYSTEM	26
4.1	Entity Environment (dsto.nef.entity)	26
4.1.1	EntityEnvironment.....	27
4.1.2	DictionaryLoader.....	27
4.1.3	ThemeLoader	27
4.2	Document Package (dsto.nef.entity.document)	28
4.2.1	EntityContent	28
4.2.2	NEFEntity	29
4.2.3	NEFTheme	29
4.3	Process Package – Server (dsto.nef.entity.process).....	30
4.3.1	AbstractCRFProcess	31
4.3.2	MaxEntPosTaggerProcess	32
4.3.3	PostPosFixProcess	32
4.3.3.1	PosDictionary	32
4.3.4	CRFWFGTaggerProcess	33
4.3.5	MaxEntChunkFromPosProcess	33
4.3.6	CRFChunkFromWFGProcess	33
4.3.7	PosEntityFinderProcess	34
4.3.7.1	ThemeEntityFinder.....	34
4.4	Process Package – Client (dsto.nef.entity.process.output)	34
4.4.1	FXExportProcess.....	34
4.4.2	XMLOutputProcess	35
4.5	SwingUI Package (dsto.nef.entity.swingui)	35
4.5.1	EntityTemplate.....	35
4.5.1.1	EntityTemplateFactory	35
4.5.2	EntityContentPanel	36
4.5.3	JEntitiesTable.....	36
4.5.4	JHighlightedTextArea	37
4.5.4.1	JHighlightedTextAreaController.....	38
4.5.4.2	JTarget	38
5.	EXTRAS SYSTEM	39
5.1	ServerProcessing Implementations.....	39
5.1.1	LocalServer	39
5.1.2	GenericHandlerSession.....	39
5.2	Process Configuration Framework	39
5.2.1	Reading From the File System	39
5.2.2	ProcessConfigurator.....	40
5.2.3	OperationGroup	40
5.2.4	OperationProcess.....	41
5.2.5	OperationProcessParam	41

5.2.6	JProcessPanel.....	41
5.3	Batch Control	42
5.3.1	BatchController	42
5.3.2	JBatchDialog.....	43
5.3.3	JStringListDialog.....	43
6.	REFERENCES.....	45
APPENDIX A:	STARTING NEFTOOL	46
APPENDIX B:	PROGRAMMING GUIDELINES	47
B.1.	General	47
B.2.	Document Package.....	47
B.2.1	NEFDocument.....	47
B.2.2	NEFContent	47
B.3.	NEFProcess Package	47
B.3.1	NEFProcess	47

1. Introduction

The text processing team from the Intelligence Analysis discipline has experimented with the viability of using machine-learning models to find named entities¹ within a text document. The **NEFTool** has been designed as a testbed framework for experimenting with implementations of different information extraction methodologies. The system has a modular architecture with the separation of data, process and display as a major design goal. As a result the system is extensible making it possible to modify its various components or even add new components without affecting its existing functionality, thus providing a test environment to support research in text processing.

This report provides an outline of the fundamental framework of the **NEFTool** and describes an implementation of the system used to find named entities within a document. The report focuses on the framework and implementation aspects of the system. Details of the Information Extraction (IE) processes are not addressed.

The IE processes implemented include the following steps:

1. Sentence Breaking and Tokenisation:
The document is broken up into *tokens* (words) and *sentences*. The information extraction process is targeted at tokens in the context of a sentence.
2. Tagging:
Tokens are analysed by a series of Machine-Learning models, with each trained model assigning additional metadata to the tokens. Each stage of the machine learning process relies on previously discovered tags² (metadata) as well as tags on surrounding tokens in the sentence.
3. Entity Discovery:
Named Entities are found in the document by an application of rules that search for a particular sequence of tags indicating the presence of an entity.

Core System	Extras System
Token System	
Entity System	

Figure 1 System components of the NEFTool

¹ Named Entities are textual references to items of interest, such as a Person, Organisation, Place, Date/Time, etc.

² An example of a tag for a token is its *Part Of Speech* (POS) such as a noun or verb.

The current implementation of the **NEFTool** comprises four components shown in Figure 1. Each component contributes to the information extraction pipeline described above. The components of the system are:

- The **Core System**, described in Chapter 2, provides the base framework of the **NEFTool** that the other systems extend.
- The **Token System**, described in Chapter 3, is responsible for the sentence breaking and tokenisation of the document, as well as methods for associating metadata with individual tokens and normalisation³ of tokens.
- The **Entity System**⁴, described in Chapter 4, is responsible for the Tagging and Entity discovery tasks of the information extraction process by extending the tokenisation framework provided by the **Token System** and adding machine-learning models.
- The **Extras System**, described Chapter 5, contains a collection of utility objects, graphical widgets and various templates useful for adding additional features to the system whilst not being a necessary part of it.

³ For example, converting quotes by Microsoft Word (") into the standardised ASCII quote (`). This is to ensure that all tokens use a charset that matches the charset used by the training data for the machine learning models.

⁴ The **Entity System** currently comprises of Tagging and Entity Finding, this system will be divided into separate components at a future date.

2. Core System

The **Core System** is the most basic, yet most important component of the **NEFTool**. It contains the many interfaces that constitute the entire framework of the system. The **Core System** essentially makes the **NEFTool** an integrated development environment (IDE) that can have features added to it incrementally to construct an entire system. Figure 2 shows the tiered layered structure of the **Core System** of the **NEFTool**.

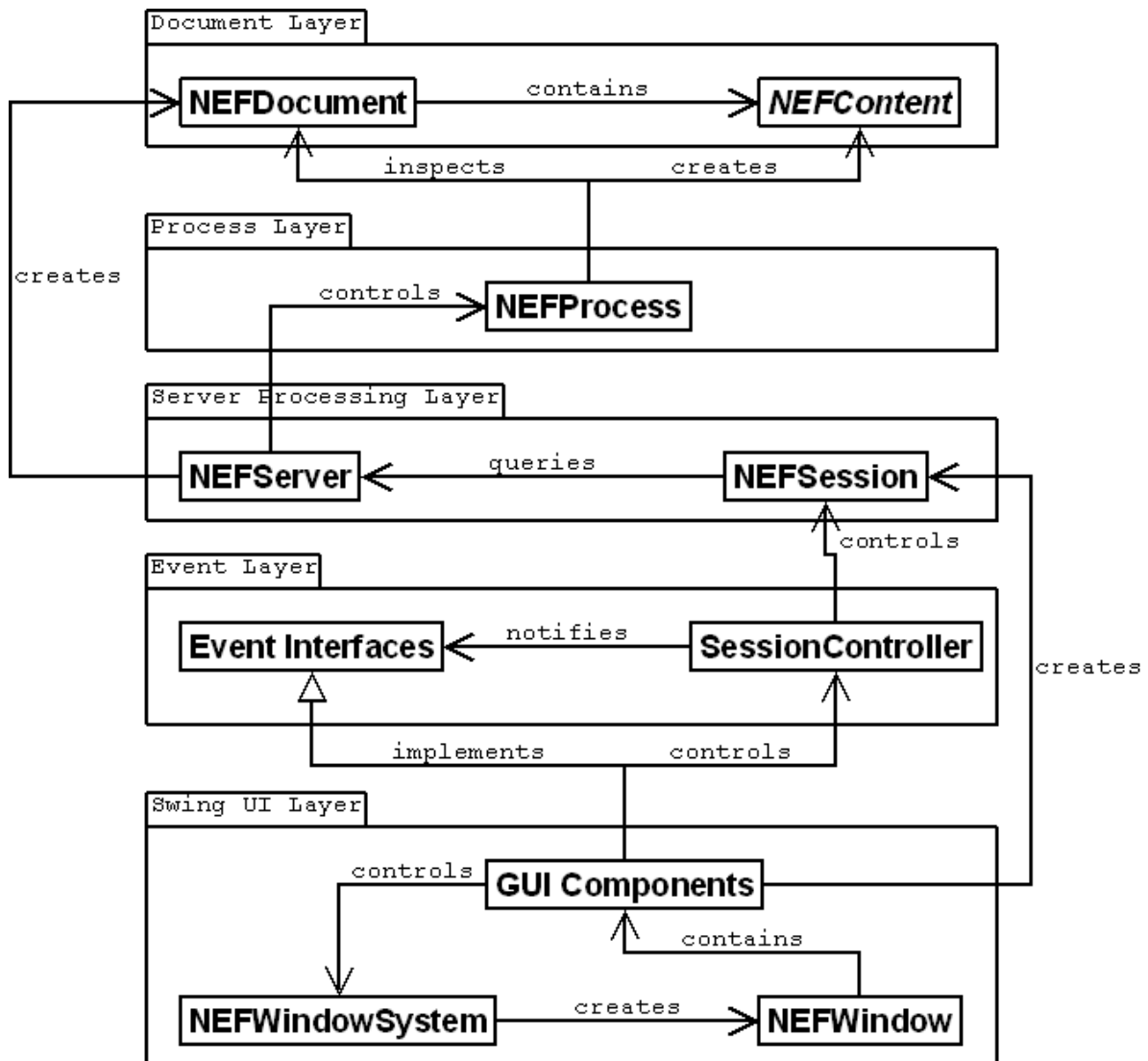


Figure 2 Common Packages in the NEFTool

The **Core System** has been split into five packages (layers) that contain components that are of a tiered design. The tiered design is mirrored in the other systems of the **NEFTool**. The layers are:

- The **Document Package** contains the source document and all the associated markup of the document.

- The **Process Package** is responsible for identifying information in the document and adding the found information to the Document Package.
- The **Server Processing Package** is responsible for controlling the Process Package in marking up a document. It also handles the process of getting a document from an external source.
- The **Event Package** is responsible for separating logic from the user interface (UI). By using a set of interfaces, UI components can be made without relying on other UI components in the system.
- The **SwingUI Package** [1][2] is responsible for displaying the information from the document layer, and for allowing a user to control the flow of information.

2.1 Core Layer (dsto.nef.core)

The **Core Layer** of the system is a holder for all variables that are used by the **Core System** and other layers that reference the core system.

2.1.1 CoreEnvironment

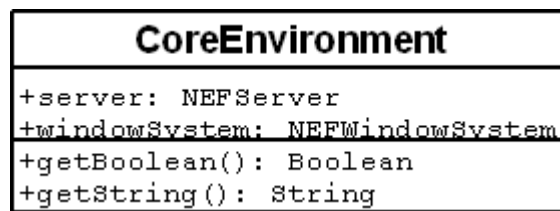


Figure 3 *dsto.nef.core.CoreEnvironment*

CoreEnvironment is designed to hold singleton-like constants that the system commonly uses. As these objects cannot be created by the get methods, they have a set method that is triggered to raise an error in the event that a non-null variable will be replaced.

The CoreEnvironment also loads a Properties file; this allows for values to be saved so that they can be loaded at startup. Any variable or setting that is important to only the **Core System** should be added to the core environment. However if a variable is important only to a particular System, that variable should be added to the Environment class directly associated with that system.

2.2 Document Package (dsto.nef.core.document)

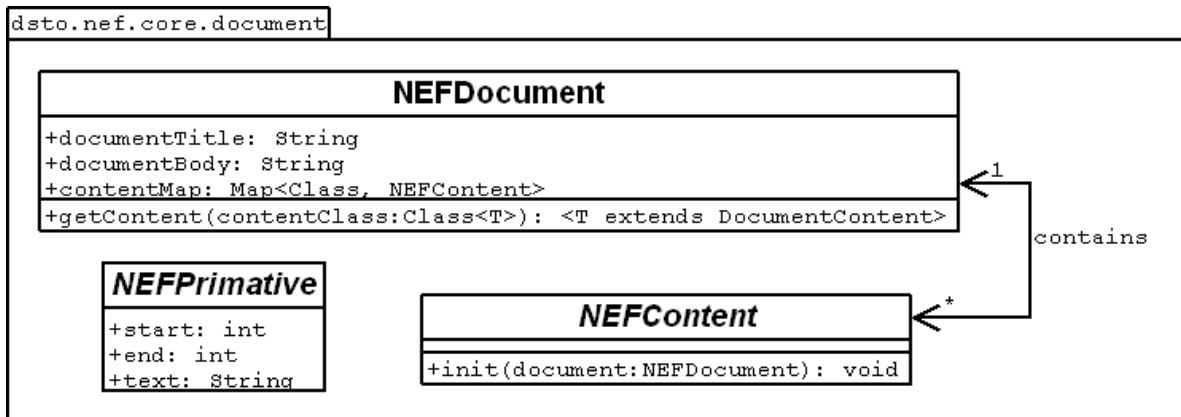


Figure 4 *dsto.core.nef.document Package*

The **Document Package** contains basic collection-like objects used to hold the markup information of a single text document.

Objects extended from the **Document Package** do not contain methods for determining markup information. For example, in the case of *sentence breaking*, it would be possible to have a single “SentenceContent” object capable of breaking a text document into sentences. However, it is preferable to have a separate “setSentences” method and a “SentenceBreaking” process, as it allows for different sentence breaking methods to be substituted.

2.2.1 NEFDocument

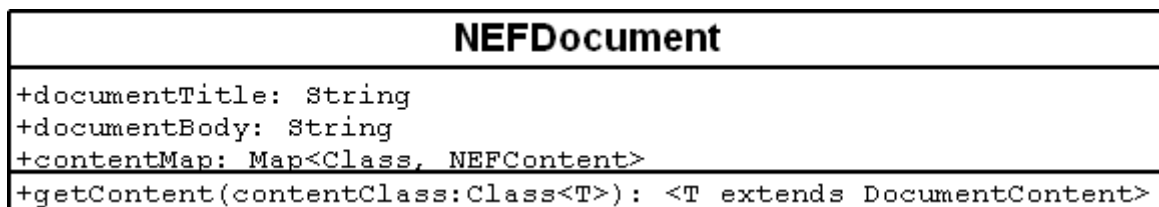


Figure 5 *dsto.nef.core.document.NEFDocument*

The `NEFDocument` class is a container for all the different `NEFContent` objects that are used for the markup of a single document. As well as containing `NEFContent` Objects, `NEFDocument` also contains all details of the original document used to create the `NEFDocument`.

2.2.1.1 `getContent(Class<T extends NEFContent> contentClass): T`

This method searches for a `NEFContent` object in the `contentMap` variable that has previously been created by a `getContent(Class)` call. If no instance of `contentClass` is found, a new instance of `contentClass` is created, stored and returned to the user.

Errors could be generated by using inheritance with `NEFContent` objects, so it is recommended that `NEFContent` child classes are defined as `final`.

2.2.2 NEFContent



Figure 6 *dsto.nef.core.document.NEFContent*

NEFContent is an abstract class used to represent a type of markup relevant to a text document and is used to control the use of the `getContent(Class)` method of NEFDocument. It follows that NEFContent objects should have a featureless Constructor in order for an instance to be generated.

The NEFContent Class does not contain any methods. This is to enforce the requirements that NEFDocument only contains objects that are specifically content-based and are associated with a single document.

Another useful guideline is to make a NEFContent child class a final class. This would remove any errors that `NEFDocument.getContent(Class)` may create due to polymorphism.

2.2.3 NEFPrimitive

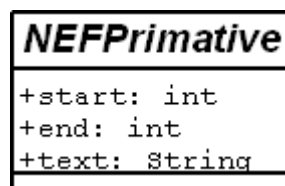


Figure 7 *dsto.nef.core.document.NEFPrimitive*

NEFPrimitive is the most basic form of information regarding a piece of markup. It is used to simplify code and to provide a wrapper for the most basic information that a piece of markup can hold. This should be used as the basis for all objects that represent a section of text.

2.3 Process Package (dsto.nef.core.process)

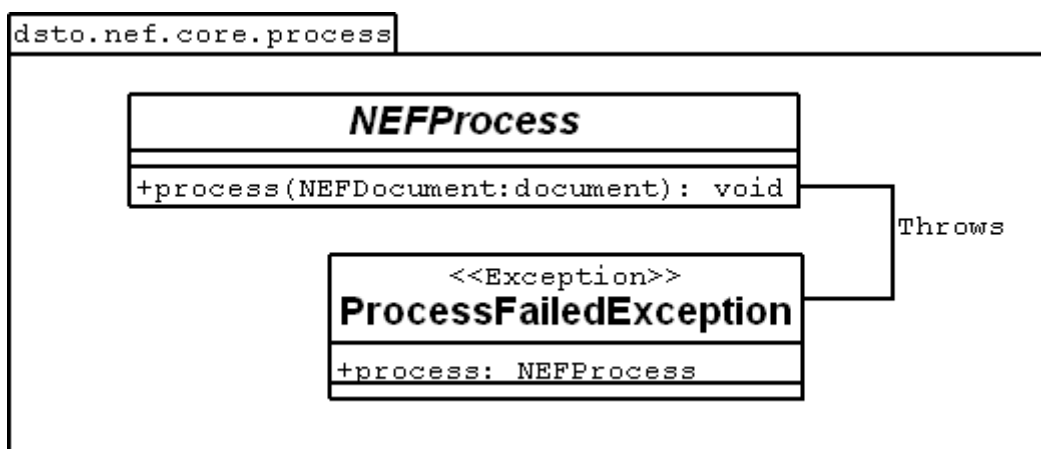


Figure 8 *dsto.nef.core.process Package*

The **Process Package** is where all operations on a NEFDocument are performed. Processes can range from breaking sentences to detecting whether “Bin Laden” has been mentioned in a text that talks about terrorism.

Processes are solely responsible for:

- a) Identifying information found in the source text document and NEFContent objects that have been set by previous processes.
- b) Saving the found information into the NEFContent objects it was designed to populate.

Processes are designed to be single-task oriented with the idea of a "God-process" to be avoided at all costs. This is to allow for one process to be swapped with another process that does the same task, but uses a different method. This also has the additional benefit of making debugging easier. For example, if a document has not been tokenized properly, it would be correct to assume that the tokenizing process is not working properly.

The **Process Package** and the **Document Package** are the key packages of the system and are not dependant on any other package (including the **Server Processing Package**).

2.3.1 NEFProcess



Figure 9 *dsto.nef.core.document.NEFProcess*

NEFProcess is a common interface that represents a single task that can be performed on a document. Each Process is designed to only do one particular task in order to make it possible to swap one similar process with another.

NEFProcess objects are designed to be “create once, use many”. Therefore no memory of previous documents should be held within the object.

Not all NEFProcess objects are used to generate data in a NEFDocument. Some may be linked to an output process, and are only used to output discovered data to an external source. These NEFProcess objects are referred to as *client processes*.

2.3.2 ProcessFailedException



Figure 10 *dsto.nef.core.document.ProcessFailedException*

A `ProcessFailedException` is thrown by a `NEFProcess` in the event that a particular error is thrown by a method call if the process is in an un-expected state, or a required piece of information in the `NEFDocument` has not been set by a previous process.

2.4 Server Processing Package (dsto.nef.core.serverprocessing)

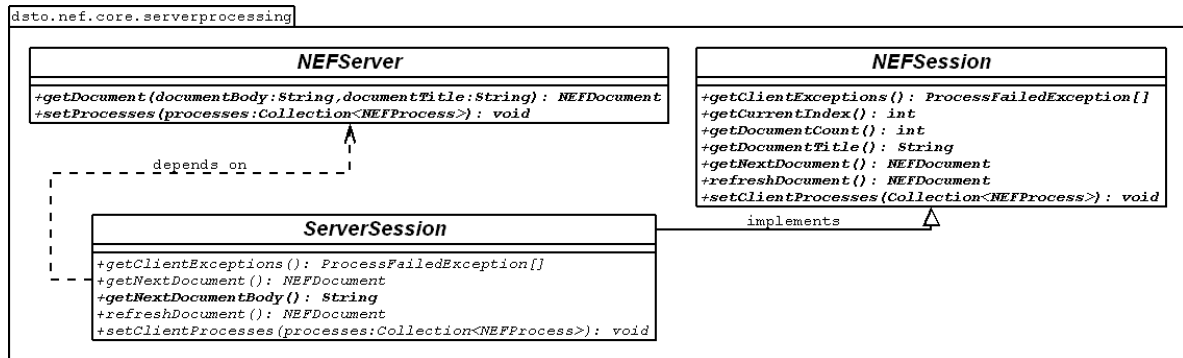


Figure 11 *dsto.nef.core.serverprocessing* Package

Input sources for the **NEFTool** can range from a file of a specific data format, to an entire database of documents. Regardless of the data type, all types of content need to be converted into a `NEFDocument`. The **Server Processing Package** is designed to:

- Allow different data formats to be added to the system, and
- Manage a `NEFProcess` queue (if that is the desired conversion method).

The **Server Processing Package** was designed so that modifying a `NEFServer` should not affect the operation of a `NEFSession`. This allows for quick modification to Server Processing classes without having to make any fundamental changes to the classes that depend on it.

The most common method of converting a data source into a `NEFDocument` involves creating a `NEFSession` class that can extract the document content and then pass it onto an object that can convert it into a `NEFDocument` (usually a `NEFServer`).

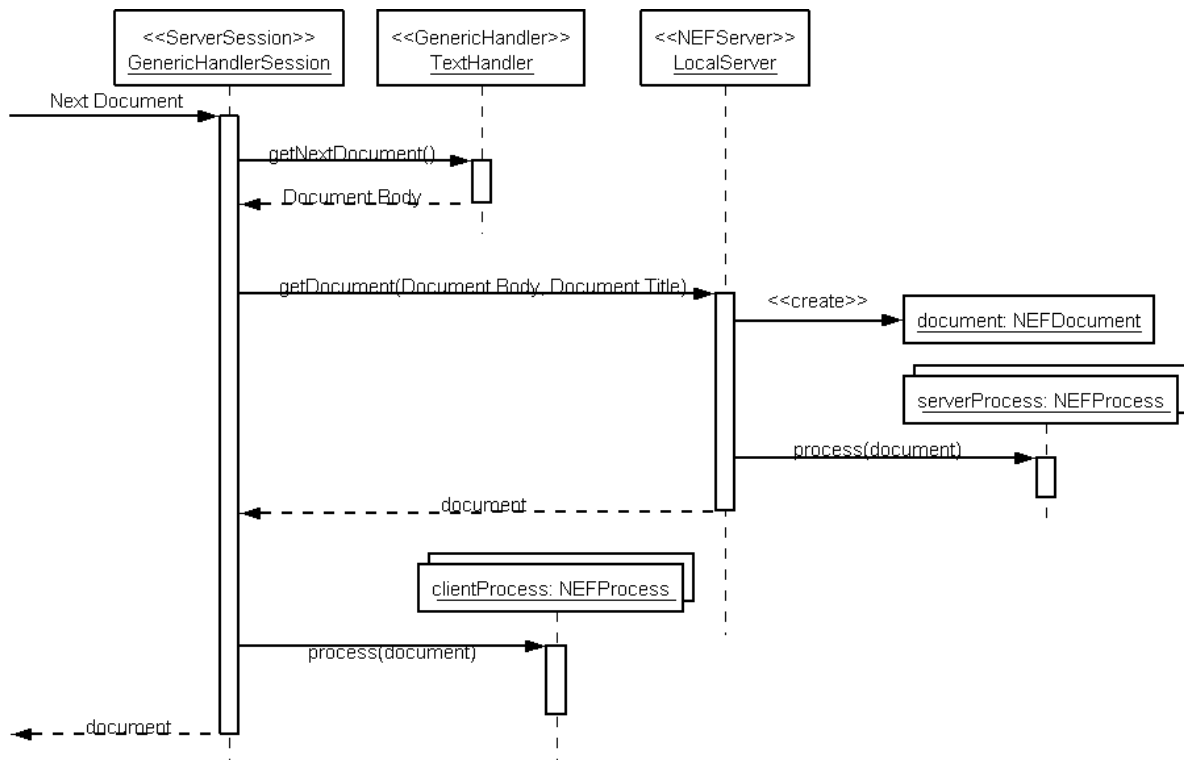


Figure 12 Requesting the next document in a GenericHandlerSession

Figure 12 shows an example of the processes involved in getting a NEFDocument from a GenericHandler (which handles the content extraction). The GenericHandlerSession does the following actions:

1. Queries the GenericHandler for the document content.
2. Sends the content to a LocalServer instance of a NEFServer, which runs a process queue over the content.
3. The NEFDocument is processed by several client NEFProcess objects within the NEFSession before being returned to the user.

2.4.1 NEFServer

NEFServer	
<pre> +getDocument (documentBody:String,documentTitle:String): NEFDocument +setProcesses (processes:Collection<NEFProcess>): void </pre>	

Figure 13 dsto.nef.core.serverprocessing.NEFServer

NEFServer is an abstract class used to convert a string (i.e. a document's content) into a NEFDocument object. This is generally managed by using a collection of NEFProcess objects as a miniature production queue; however it is possible to use other methods of converting the string.

An alternative design of the NEFServer would be to have it as a singleton that holds a collection of NEFProcess objects however the NEFServer was made polymorphic in order to

allow for swapping with different methods of document conversion. Some examples of a NEFServer that could be created are:

1. A NEFServer that uses IE server on a network to convert a document, or
2. A NEFServer that interfaces with and extracts data from an *Unstructured Information Management Architecture* (UIMA) [3] based framework.

All that would be required of such a NEFServer instance would be to send, receive and parse the received information into a NEFDocument structure. The separation of document and process was intended to allow for this type of experimentation.

A NEFServer object has no memory of previous document conversions, or any methods to read an unknown file format. These methods are handled by a NEFSession object.

2.4.2 NEFSession

NEFSession
<pre> +getClientExceptions(): ProcessFailedException[] +getCurrentIndex(): int +getDocumentCount(): int +getDocumentTitle(): String +getNextDocument(): NEFDocument +refreshDocument(): NEFDocument +setClientProcesses(Collection<NEFProcess>): void </pre>

Figure 14 dsto.nef.core.serverprocessing.NEFSession

A NEFSession is a handler for a collection of documents that will be converted into NEFDocument objects. A collection is typically a folder containing text documents in a specified format. However, a collection can also consist of a NEFDocument object that has been saved to the file system, or a single compressed file containing several documents. The corpora sourced from the *Message Understanding Conference*⁵ (MUC) is an example of the latter.

A NEFSession object is responsible for extracting the text body of a document of a particular format and then handing that information to an object designed to produce a NEFDocument (for example, a NEFServer). It *does not* carry out the actual conversion into a NEFDocument. This design encourages a single, manageable method for creating NEFDocument objects, and simplifies the design of NEFSession objects.

It should be noted, however, that a NEFSession may contain a collection of client processes. These processes are only included to export information to external sources (such as a database) while batch processing, and do not make any changes to the NEFDocument itself.

⁵ A Conference initiated and financed by DARPA to encourage the development of new and better methods of information extraction. [4]

The use of NEFSession objects to read (and convert) any input source allows for a user interface to be quickly modified to accept different file formats. All the programmer needs to do is to create a new NEFSession object and call the methods to display the NEFSession.

2.4.3 ServerSession



Figure 15 *dsto.nef.core.serverprocessing.ServerSession*

ServerSession is an abstract NEFSession class for NEFSession objects that relies on a NEFServer instance to generate NEFDocument objects. The NEFServer instance used can be found at `dsto.nef.core.CoreEnvironment`.

The combined use of a NEFSession with a NEFServer allows several different file formats to use the same process queue for information extraction. This is the recommended parent class of nearly all NEFSession objects.

2.5 Event Package (dsto.nef.core.event)

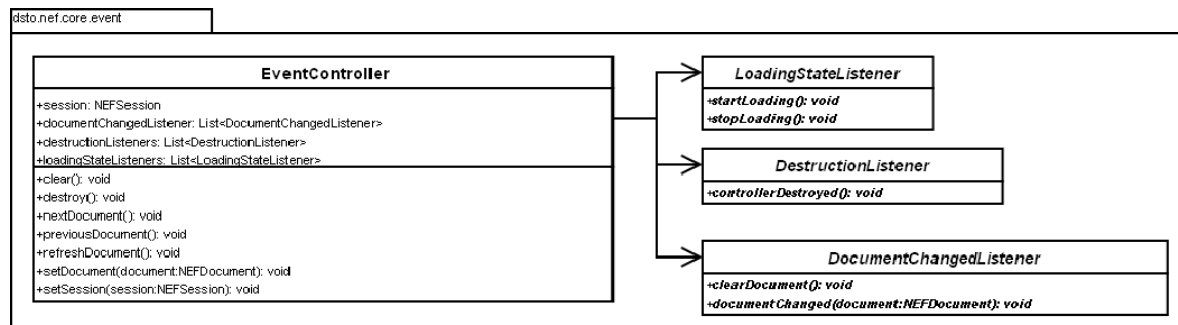


Figure 16 *dsto.nef.core.event* Package

As many different methods are available for generating NEFDocument objects and controlling the conversion process, linking display components to the system can become convoluted. The **Event Package** is designed to provide a further abstraction layer from processing layers and the user interface.

Display components are created which implement a specific set of interfaces based on their desired use and operation. These components are notified of changes to the system by a Controller object, usually a derivative of EventController.

2.5.1 DocumentChangeListener

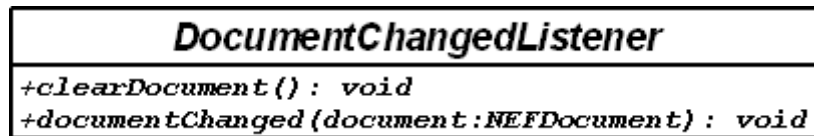


Figure 17 *dsto.nef.core.event.DocumentChangeListener*

The DocumentChangeListener interface is designed to allow for the separation of display components from the underlying logic used to generate NEFDocument objects. Its main purpose is to notify components that the current document in focus has been changed and to update the display accordingly.

As a DocumentChanged event is usually the last thing caused by changing a NEFDocument, it is possible to use a DocumentChangeListener to mark when a EventController (or other underlying class) has been updated. Although a DocumentChangeListener ignores the underlying document conversion process, components implementing the interface can be aware of this process.

2.5.1.1 Possible use as a MDI system

An MDI (Multiple Document Interface) is a type of display layout that has one or more bank of controls that are always visible, but modifies only the current window (or NEFDocument) held in focus. Manipulating the behaviour of DocumentChangeListener objects could trick the system into following this design pattern. By triggering a DocumentChanged event every time a new window is brought into focus, the associated views and controls are updated to focus on the document represented by the focus window. An example of an MDI-based program is shown in Figure 18.

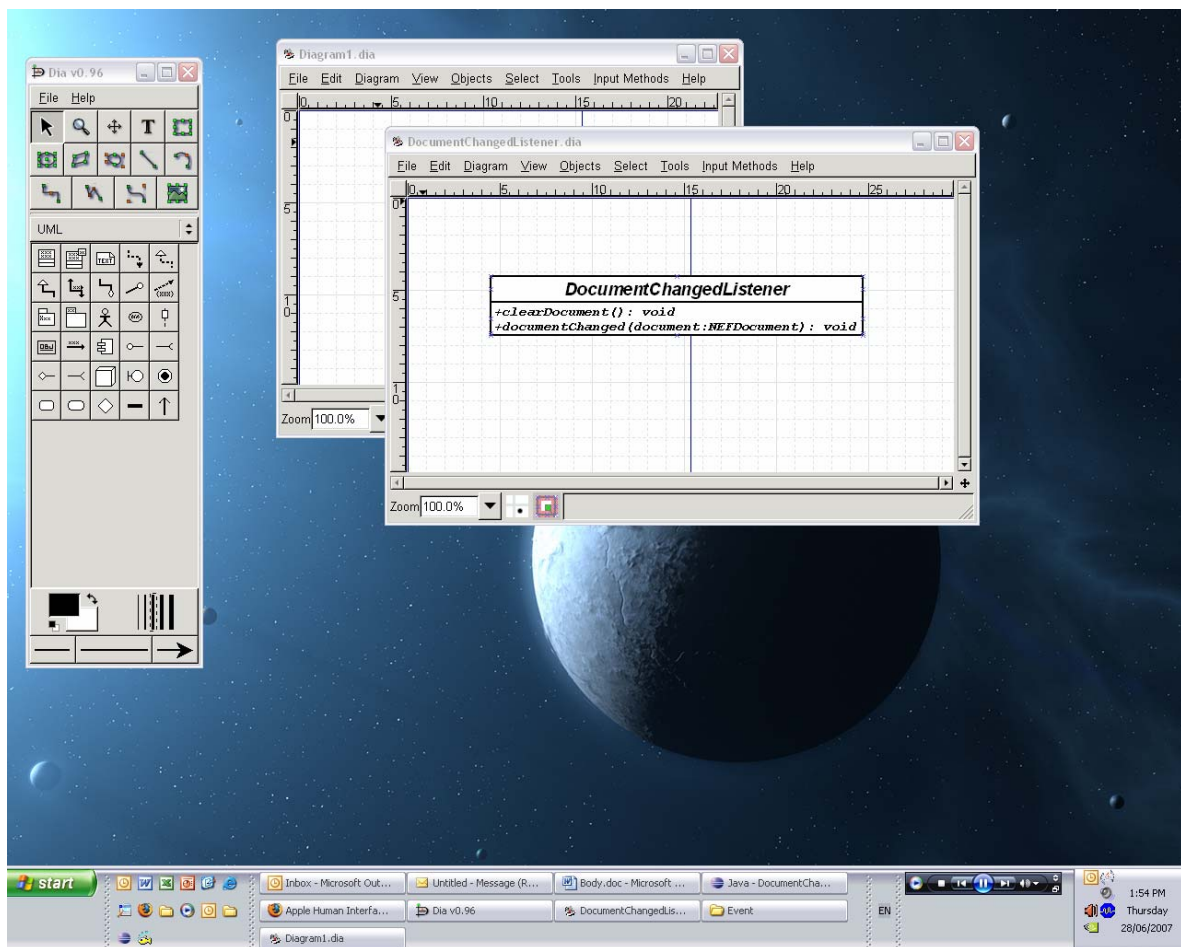


Figure 18 Dia (<http://live.gnome.org/Dia>) uses a typical MDI Layout

Note: A static variable (or a private variable in the NEFWindowSystem singleton) representing the current window (combined with DocumentChanged events) would be needed for updating the controller target. This would possibly require components in the MDI system to know about this variable, which would limit the reuse of these components.

2.5.1.2 *documentChanged* method

The method `documentChanged (NEFDocument)` notifies components to update their contents to display a changed document. Figure 19 illustrates this process.

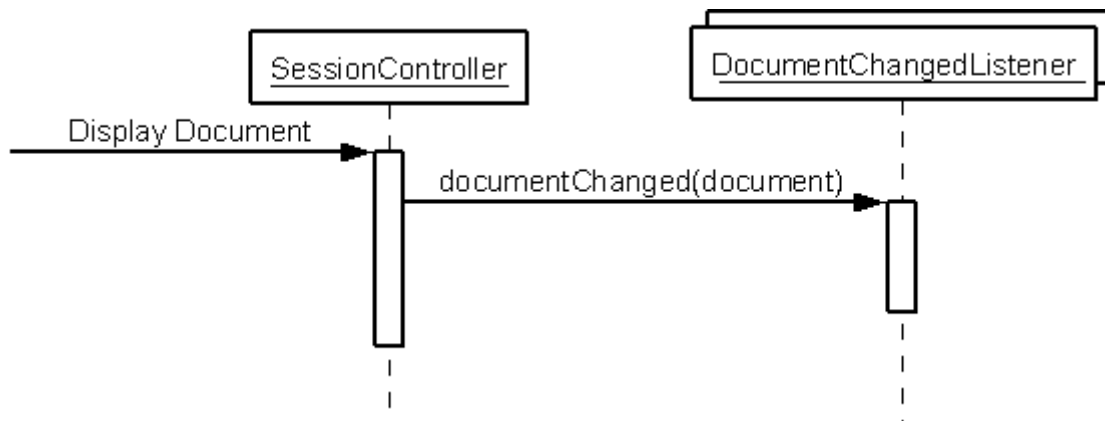


Figure 19 Displaying a Document

2.5.1.3 *clearDocument* method

The method `clearDocument()` is used to reset all display components to their default no-information state, and is mainly used in the event that a `NEFSession` object has been closed. Figure 20 shows a structure similar to Figure 19 with `clearDocument()` being used instead of `documentChanged(NEFDocument)`. It is possible that `documentChanged(NEFDocument)` with a null argument could be used instead.

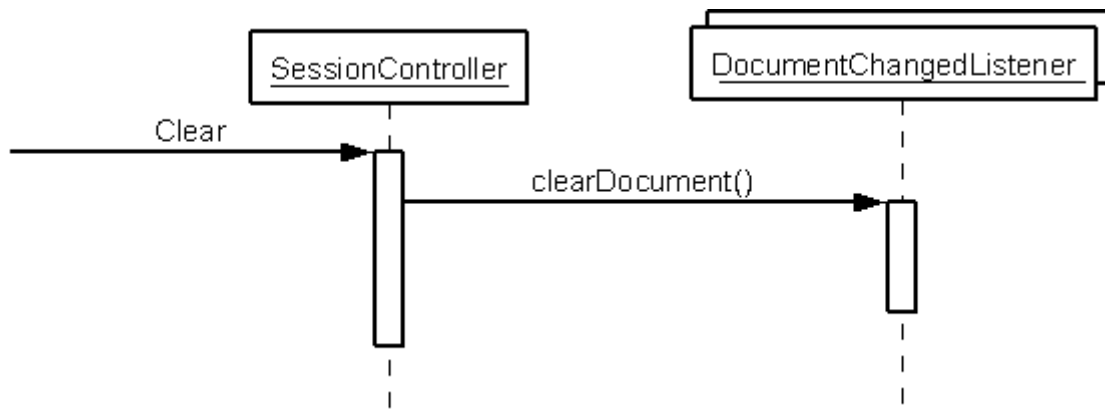


Figure 20 Clearing a Document

2.5.2 DestructionListener

Figure 21 *dsto.nef.core.event.DestructionListener*

Some objects link themselves as listeners or connect to external resources that need to be released when they are no longer used. Therefore, a form of notification is needed when a particular view or window has been closed. While the **NEFTool** currently relies on `JFrame` objects to report a closing window, it is not a dependable source of such information.

Therefore the `DestructionListener` interface was created to allow for different systems to still notify the necessary components of a closing window.

Components that have been called by a `DestructionListener` should sever all known links to the outside world so that garbage collection can retrieve them as quickly and efficiently as possible.

2.5.3 LoadingStateListener



Figure 22 *dsto.nef.core.event>LoadingStateListener*

Certain tasks carried out by the system may take considerable time, leaving the user potentially confused as to what is happening. Therefore some form of feedback is needed when a process intensive task is being carried out. Since there is separation between the GUI and the conversion process, a form of abstract interface is needed to indicate this type of change.

Due to the varying performance of the different operations performed, depending on task complexity and document size, it is impossible to know the exact progress of a task. Therefore only two methods are used, a start and a stop method. Several processes may be stacked and be overlapping, so a form of counter is implemented to count the number of processes currently running. Synchronized methods should be used to prevent multiple thread-based errors while counting.

A component that implements `LoadingStateListener` does not need to be complex. A simple progress bar at the bottom of a window, or an animated waiting icon, is usually enough to signal that the machine is currently “thinking”.

2.5.4 EventController

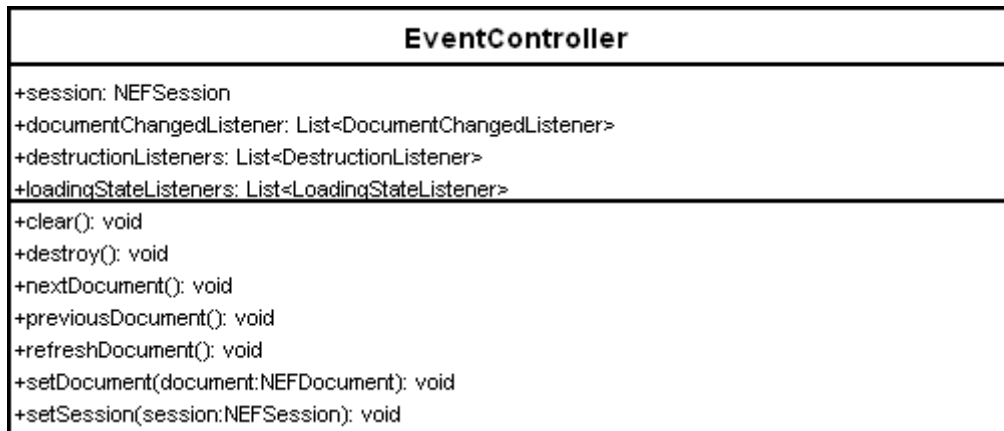


Figure 23 *dsto.nef.core.event.EventController*

While a **NEFSession** controls the process of generating a **NEFDocument**, a **EventController** controls the **NEFSession** object and uses the **Event Package** to provide a connection to the UI.

The **EventController** notifies the UI by notifying **DocumentChangeListener**, **DestructionListener** and **LoadingStateListener** objects and providing a separation from the UI and **NEFSession** objects. This allows the UI to display **NEFDocument** objects and control a **NEFSession** without requiring the UI to directly reference the **NEFSession**.

2.6 SwingUI Package (dsto.nef.core.swingui)

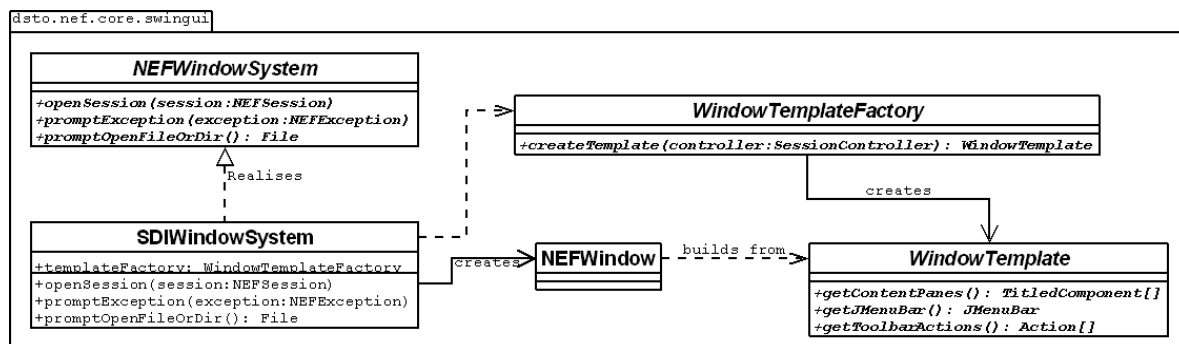


Figure 24 *dsto.nef.core.swingui Package*

The **Swing UI Package** utilizes Swing and the **Event Package** to create an easily customisable GUI for displaying document objects. Adding GUI components to the **NEFTool** has been designed to be as easy as possible.

Instead of creating an entire GUI for each change in the system, GUI development has been split into creating individual panels and connecting them to an internal event system. Although separating the components leads to a more complex system; the splitting of components allows for smaller tightly focussed components that are simple to maintain and modify. The only aspect of the system that requires effort is actually creating the initial components themselves.

2.6.1 WindowTemplate

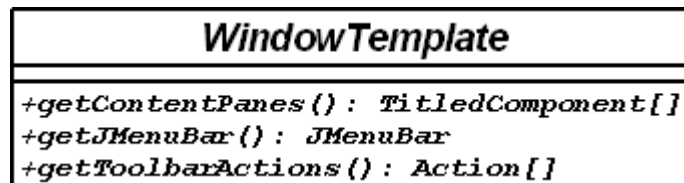


Figure 25 *dsto.nef.core.swingui.WindowTemplate*

As the GUI is designed to be as adaptable as possible, forcing the user to use a single object is very limiting, so the WindowTemplate interface was created as a template for what a NEFWindow should display. All components are created in the WindowTemplate class and are returned by various methods which define where the content is likely to be displayed.

2.6.2 WindowTemplateFactory



Figure 26 *dsto.nef.core.swingui.WindowTemplateFactory*

WindowTemplateFactory is a simple helper class for creating WindowTemplate objects. Its main purpose is to provide NEFWindowSystem with a mechanism for creating WindowTemplate objects without having to change the underlying code of the NEFWindowSystem.

For each new WindowTemplate class created by a programmer, a matching WindowTemplateFactory should be created, so that an instance of the WindowTemplate can be made by a NEFWindowSession.

2.6.3 NEFWindowSystem

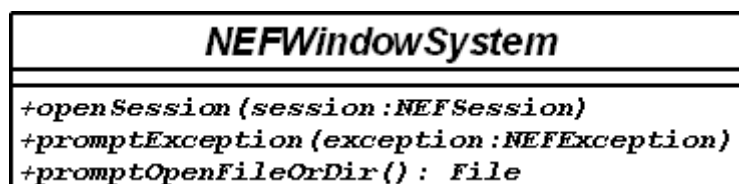


Figure 27 *dsto.nef.core.swingui.NEFWindowSession*

The NEFWindowSystem is an interface responsible for controlling the GUI delivered to the user. In most forms, this involves creating and controlling multiple instances of NEFWindow, and their containing window or frames.

The NEFSystem is intended to be a singleton, but, in order to accommodate polymorphism the instance has been moved to CoreEnvironment. This is the location to which all references to NEFWindowSystem should be made.

By making changes to the NEFWindowSystem, the window organisation behaviour of the **NEFTool** can be changed from a SDI (Single Document Interface) system to an MDI (Multiple Document Interface), TDI (Tabbed Document Interface) or IDE (IDE-Style Interface) system just by creating a new NEFWindowSystem class.

While the NEFWindowSystem was designed to control several NEFWindow instances, it is not limited to only creating NEFWindow objects. Since all components are only interacting with an EventController object, the NEFWindow technically is not needed at all.

2.6.4 SDIWindowSystem

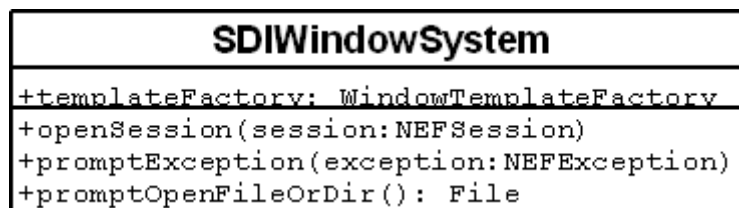


Figure 28 *dsto.nef.core.swingui.SDIWindowSystem*

The SDIWindowSystem is an instance of NEFWindowSystem that uses a SDI (Single Document Interface), which involves creating a new frame for each new NEFSession object it receives.

A typical frame is a single JFrame that holds a NEFWindow object. Each JFrame is separate from other frames, with its own individual controls and menu bar. The SDIWindowSystem keeps an account of all open JFrame objects. When all objects have been closed, the system is shutdown.

All NEFWindow creating details are generated by a WindowTemplateFactory instance that is defined in the constructor.

2.6.4.1 Opening a NEFSession in a new NEFWindow

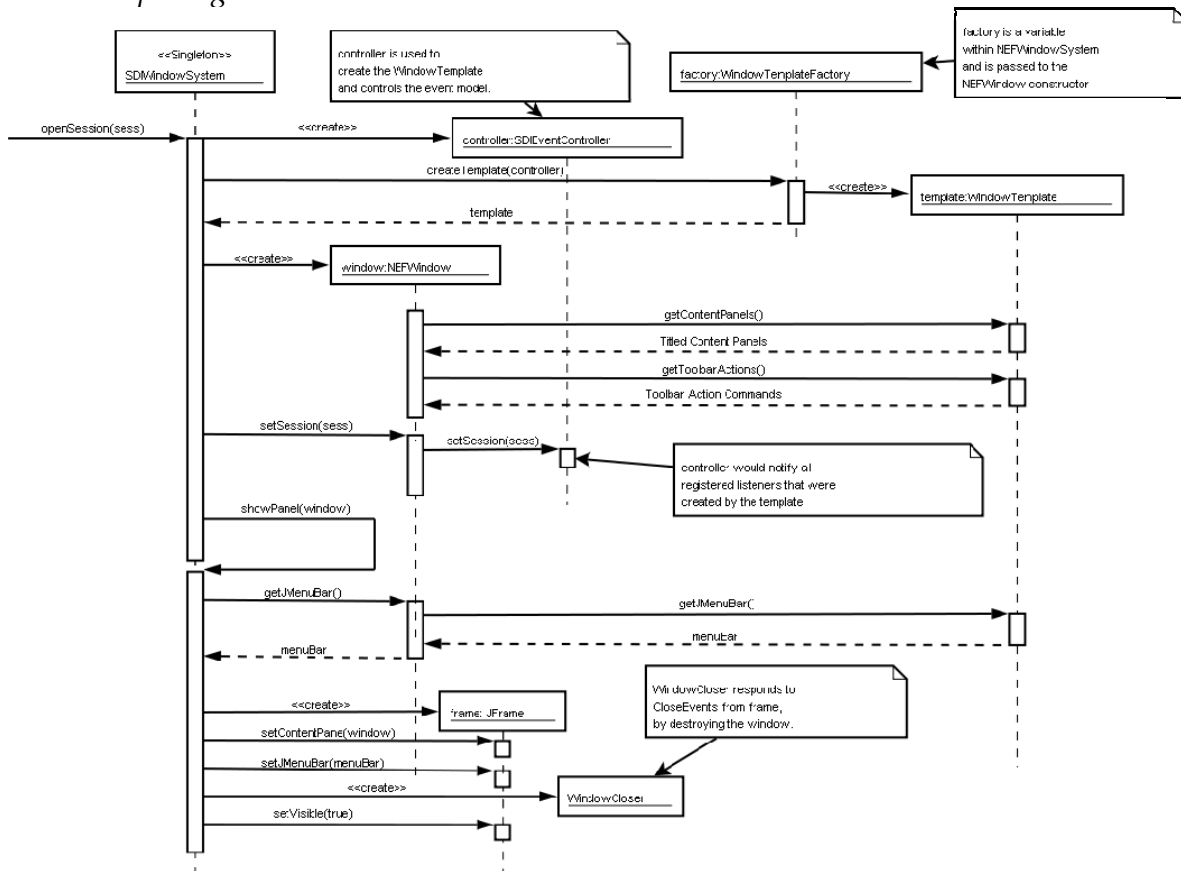


Figure 29 Creating a NEFWindow with an SDIWindowSystem

While the NEFWindowSystem does not need NEFWindow objects to run, it is still the preferred way of displaying the results from a NEFDocument. The process of creating a NEFWindow from an SDIWindowSystem seems complicated, but on a programming level it is simple, as it has been divided into smaller objects that manage the work, while providing a simple interface.

As Figure 29 shows, creating a NEFWindow involves multiple steps, but is easy to follow:

1. The SDIWindowSystem receives a request to open a NEFWindow.
2. An SDIEventController is created.
3. A WindowTemplate is created from the WindowTemplateFactory, which is controlled by the NEFWindowSystem. The SDIEventController is used in the process.
4. A NEFWindow object is created with the WindowTemplate and the SDIEventController.
5. The SDIEventController is set to display the desired NEFSession.
6. A JFrame is created to hold the NEFWindow and is displayed to the user.

2.6.4.2 SDIEventController

This is a custom EventController that links the behaviour of the EventController class with an instance of the NEFWindowSystem.

Normally, when a `EventController` receives a new `NEFSession` object, the old object is replaced and the `NEFWindow` displays the new document. The `SDIEventController` is different in that it sends the request to the `NEFWindowSystem` to open the new `NEFSession` instead of the current `EventController`.

This behaviour can be controlled by setting the Boolean property defined by the `CoreEnvironment.MULTIPLE_WINDOWS_PROPERTY` key to the desired value. `SetMultipleWindowsAction` is a class which uses a `JMenuItem` to change this setting.

2.6.4.3 *WindowCloser Internal Class*

The `WindowCloser` class is a simplistic “controller” for a `JFrame` that holds a `NEFWindow`. It works by implementing two event listeners: a `java.awt.WindowListener` Interface to listen for `JFrame` closing events, and a `DocumentChangeListener` used to update the title of the `JFrame`.

As the `NEFWindow` is displayed as a `JPanel` that does not contain a `JFrame`, implementing a `WindowCloser` is necessary to manage the closing of the `JFrame`. By not making the `NEFWindow` a child of `JFrame`, or a `JFrame` controller, many different Swing-Based Interfaces can be trialled, without needing to change the `NEFWindow` code.

2.6.5 `NEFWindow`



Figure 30 *dsto.nef.core.swingui.NEFWindow*

The `NEFWindow` is a panel that displays the content of a `NEFDocument` object and is the most viewed component presented to the user. The `NEFWindow` really consists of a dummy container for an `EventController` object, and a collection of components that implement the `DocumentChangeListener` interface. Even the components themselves are not created by a `NEFWindow` object, but a `WindowTemplate` instance. All the `NEFWindow` does is position the components in a simple border layout. Figure 31 shows a `NEFWindow` using the default entity-based components.

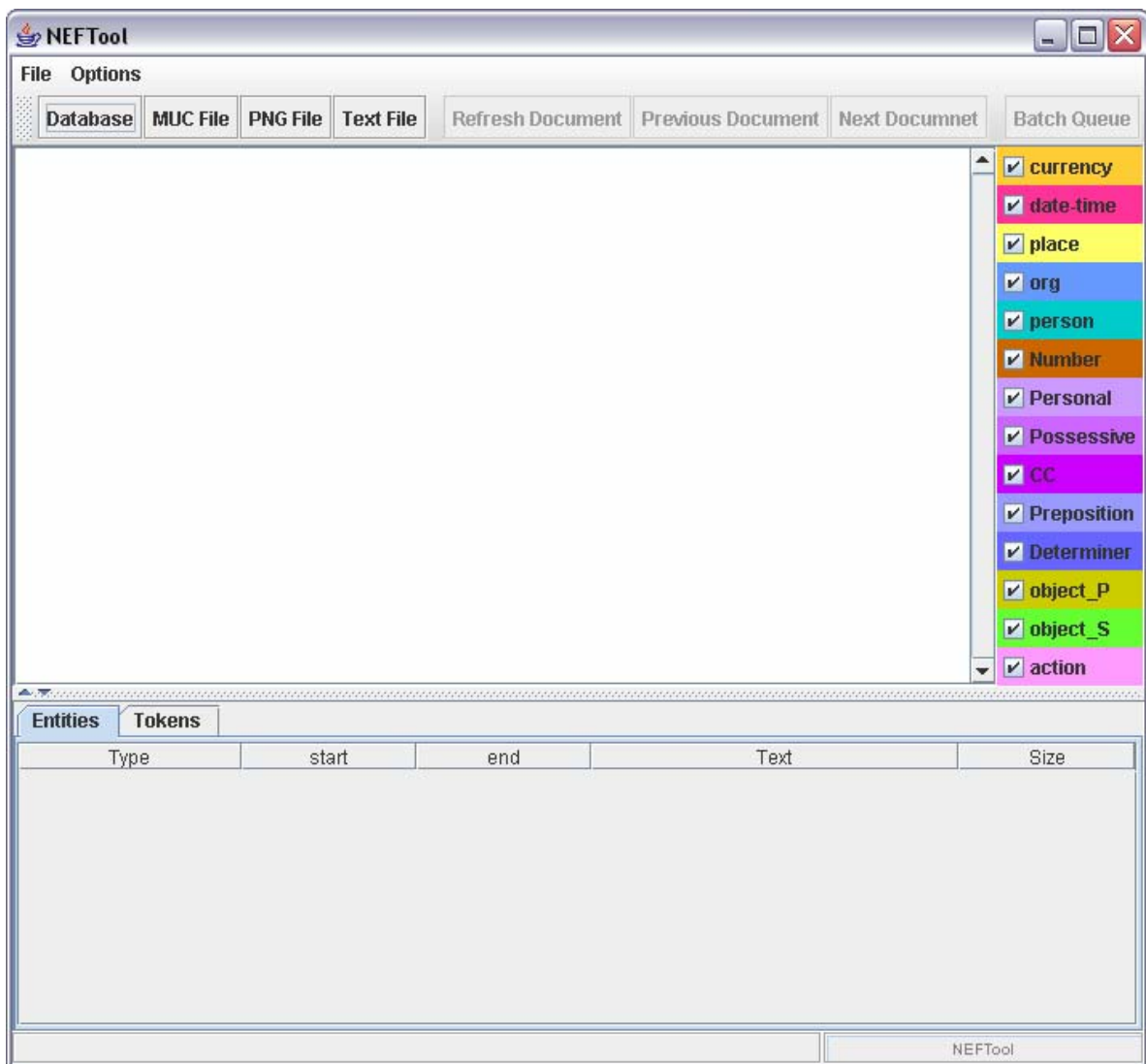


Figure 31. A NEFWindow configured by an EntityTemplate

3. Token System

3.1 TokenEnvironment

TokenEnvironment is simply a collection of static methods that allows for objects in the document package to be converted into string objects and vice versa. It is mainly used for simple debugging of the system.

3.2 Document Package (dsto.nef.entity.document)

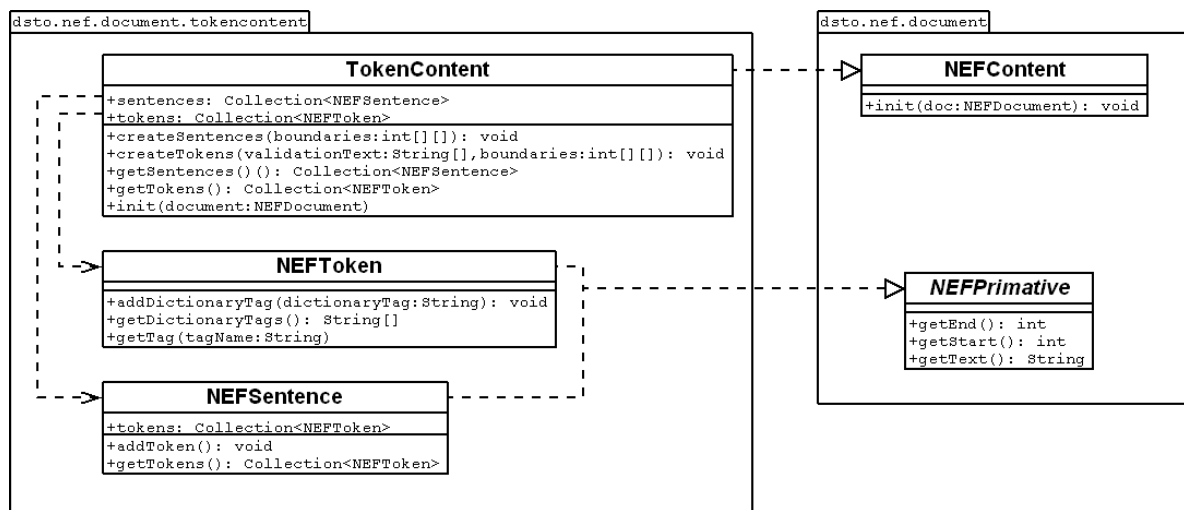


Figure 32 the Token Document Package

The **Token Content** classes concern themselves with the tokenization information in a document, and are represented as an object-oriented tree structure. It is the most commonly used group of classes in the **NEFTool**, due to its ability to assign text tags to tokens, which are useful for major operations on the document to support information extraction.

3.2.1 TokenContent

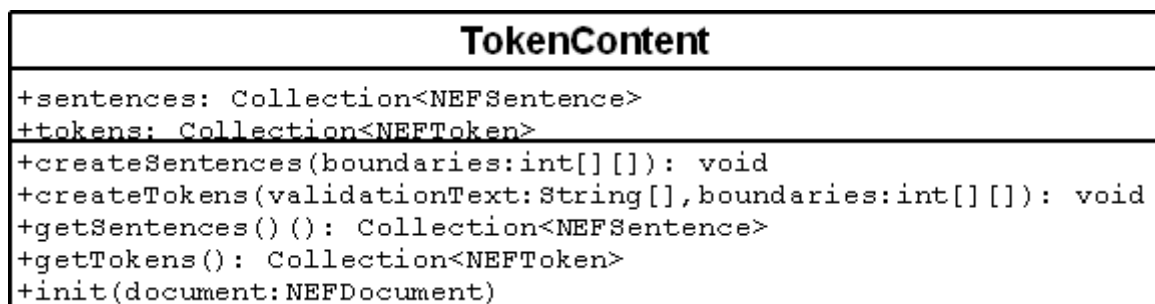


Figure 33 dsto.nef.token.TokenContent

TokenContent is a child of NEFContent, and serves as a container for NEFSentence and NEFToken objects. It represents an object-oriented tokenized markup of the finalised document.

3.2.2 NEFSentence

NEFSentence
+tokens: Collection<NEFToken>
+addToken(): void
+getTokens(): Collection<NEFToken>

Figure 34 dsto.nef.token.NEFSentence

NEFSentence is a collection of NEFTokens in sequence that represents a sentence. It is mainly used to simplify sentence-based processing of documents.

3.2.3 NEFToken

NEFToken
+addDictionaryTag(dictionaryTag:String): void
+getDictionaryTags(): String[]
+getTag(tagName:String): String
+setTag(tagName:String, tagValue:String): void

Figure 35 dsto.nef.token.NEFToken

NEFToken represents a single token in a document. It contains a get and set Tag method for setting attributes to the token. Currently in the **NEFTool**, three different tag sets are assigned to tokens: part-of-speech (POS) tags, word function group (WFG) tags, and chunk information tags.

Tags have been made abstract in order to allow existing tag sets to be extended and new tag sets to be added without having to change code. The key for any tags used should be stored in the corresponding system-environment object. That is, tags for entities are saved in EntityEnvironment).

3.3 Process Package (dsto.nef.token.process)

3.3.1 TextCleanerProcess

TextCleanerProcess
<pre>+configure(configuration:Map<String,Object>): NEFProcess +process(document:NEFDocument): void</pre>

Figure 36 *dsto.nef.token.cleaner.TextCleanerProcess*

Some documents have additional whitespace at the start and end of each line. The TextCleanerProcess trims leading and trailing spaces on each line of the document, making it more legible. It should be noted that this process overrides all content classes within the document. Therefore, if this process is to be run, it should be run first.

3.3.2 SentenceBreakerProcess

SentenceBreakerProcess
<pre>+configure(configuration:Map<String,Object>): NEFProcess +process(document:NEFDocument): void</pre>

Figure 37 *dsto.nef.token.sentencebreak.SentenceBreakerProcess*

Some processes evaluate unmarked documents one sentence at a time, requiring sentence boundaries to be determined. SentenceBreakerProcess is a simple process for converting a document into an array of NEFSentence objects using an algorithm to determine sentence boundaries.

While it is an important part of the system, it can be removed from the **Process Package** in future iterations if required. For instance, if an input source contains markedup token or sentence boundaries the SentenceBreakerProcess will cause conflicts with the input source, and will need to be removed from the processing pipeline.

3.3.3 PTBTokenizerProcess

PTBTokenizerProcess
<pre>+configure(configuration:Map<String,Object>): NEFProcess +process(document:NEFDocument): void</pre>

Figure 38 *dsto.nef.token.process.tokenization.PTBTokenizerProcess*

A document has to be broken up into tokens (words) in order for the individual tokens to be analysed by subsequent text processing processes. The PTBTokenizerProcess uses a modified *Penn Treebank*⁶ (PTB) [5] tokenizer to determine the boundaries of the tokens, and uses this information to create NEFToken objects within a TokenContent object.

⁶ The Penn Treebank Project annotates naturally-occurring text for linguistic structure [5].

3.4 SwingUI Package (dsto.nef.token.swingui)

3.4.1 JTokenTable

start	end	NEFToken	POS	CHUNK	SENT	WFG	Lookup
0	5	Sweden	NNP	B-NP	B_S	TH	
7	11	Frees	NNP	I-NP	I_S	TH	
13	15	Man	NNP	I-NP	I_S	TH	
17	24	Detained	NNP	I-NP	I_S	TH	
26	27	in	IN	B-PP	I_S	PR	
29	37	Hijacking	NNP	B-NP	I_S	TH	
39	42	Case	NNP	I-NP	E_S	TH	
47	53	October	NNP	B-NP	B_S	TH	mtb
55	55	1	CD	I-NP	I_S	TH	NUM
56	56	,	,	O	I_S	O	

Figure 39 Example output from a JTokenTable

JTokenTable is a simple JPanel containing a JTable held by a JScrollPane, which displays all tokens in a document and their associated markup. It relies on DocumentChangeListener events to update its contents.

4. Entity System

An early goal of the **NEFTool** was to discover and extract named entities occurring in unstructured text documents. Named entities are mentions of items of interest such as names of people, places and organisations. In the sentence, “Bob went to Adelaide”, the mentions “Bob” and “Adelaide” are both examples of named entities. A range of techniques can be applied to discover named entities in text. In order to support experimentation with different IE techniques, the ability to swap processes was a critical design goal for the system.

Some IE methods used by the **NEFTool** involve assigning tags to tokens. Therefore the **Entity System** is highly dependent on the **Token System**. Tagging processes of the **Entity System** are interchangeable as long as they use the same tags when saving to NEFToken objects.

4.1 Entity Environment (dsto.nef.entity)

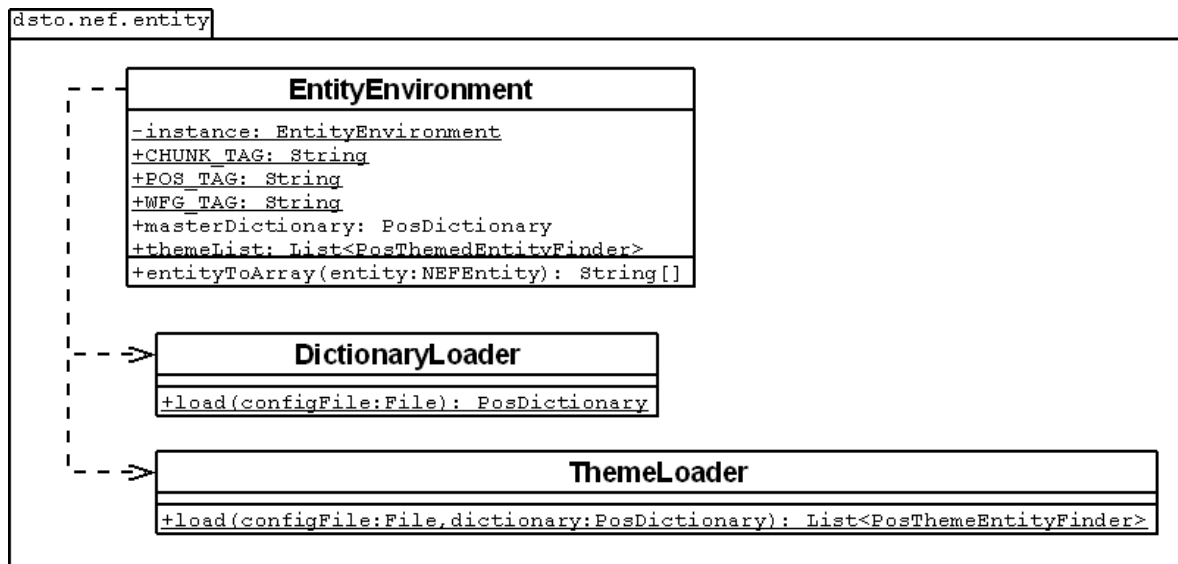


Figure 40 dsto.nef.entity Package

As the name implies, the **Entity Environment** contains all the constants that are necessary for the other objects in the **Entity System**. Once again, this package is used so that polymorphism can be implemented in the system with minimal code modification, a task which is difficult to accomplish with a straight singleton.

4.1.1 EntityEnvironment

EntityEnvironment
<pre> -instance: EntityEnvironment +CHUNK_TAG: String +POS_TAG: String +WFG_TAG: String +masterDictionary: PosDictionary +themeList: List<PosThemedEntityFinder> +entityToArray(entity:NEFEntity): String[] </pre>

Figure 41 *dsto.nef.entity.EntityEnvironment*

Similar to the TokenEnvironment, the EntityEnvironment class is a holder for variables and constants that are used by classes in the **Entity System**.

As the DictionaryLoader and ThemeLoader are important to the system, both are called to create the necessary PosDictionary object and list of NEFTheme objects necessary for most server processes in the Entity Package. However both variables can be overridden by calling set methods.

4.1.2 DictionaryLoader

DictionaryLoader
<pre> +load(configFile:File): PosDictionary </pre>

Figure 42 *dsto.nef.entity.DictionaryLoader*

DictionaryLoader is responsible for creating the CoreEnvironment.masterDictionary object and loading dictionary files from the local file system.

The DictionaryLoaderXMLConstants class is the container for all XML configuration file constants used by the DictionaryLoader.

4.1.3 ThemeLoader

ThemeLoader
<pre> +load(configFile:File,dictionary:PosDictionary): List<PosThemeEntityFinder> </pre>

Figure 43 *dsto.nef.entity.ThemeLoader*

ThemeLoader is responsible for creating instances of PosThemeEntityFinder from an XML-based configuration file. All "Theme" variables found in the theme configuration files are saved into a collection within the PosDictionary. As this is a required variable, the Dictionary Configuration file should be loaded before ThemeLoader (usually via DictionaryLoader).

In a similar naming scheme to DictionaryLoader, the XML configuration file constants are found in the ThemeLoaderXMLConstants class.

4.2 Document Package (dsto.nef.entity.document)

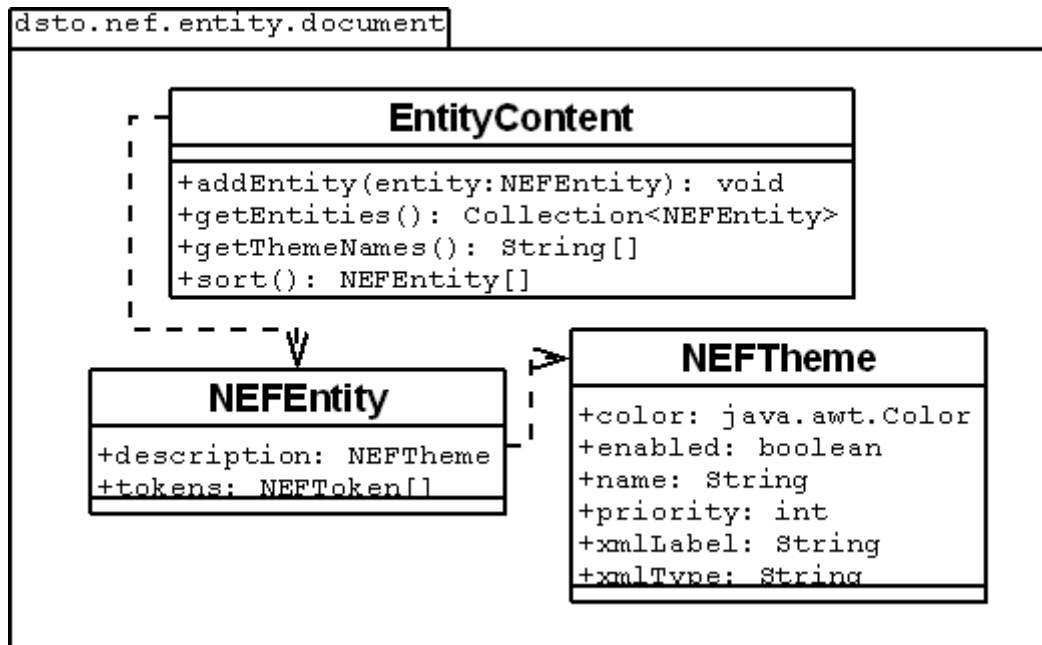


Figure 44 *dsto.nef.entity.document* Package

The essential purpose of the **NEFTool** is to find references to mentions in a document. The **Entity Content** Classes are a set of Objects that are used to store information based around mentions.

In keeping with the document-process model, EntityContent objects are unable to find mentions and rely on external NEFProcess objects to find entities.

4.2.1 EntityContent

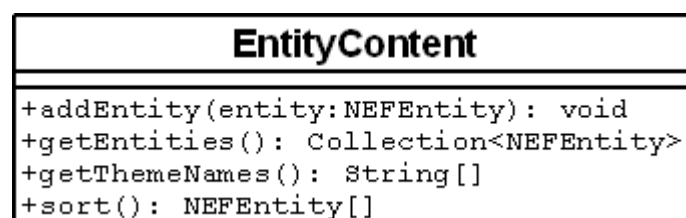


Figure 45 *dsto.nef.entity.document.EntityContent*

EntityContent is the container for all mentions found within a document. It is a child of NEFContent, and is meant to be generated by a call to the `getContent(Class)` method found in the NEFDocument.

The `getEntities()` method returns `NEFEntity` objects in the order they were put into the system. The `sort` method should be used to return a sorted set of `NEFEntity` objects.

4.2.2 NEFEntity



Figure 46 *dsto.nef.entity.document.NEFEntity*

`NEFEntity` is a container for tokens that belong to a particular mention found in a `NEFDocument`. The `NEFEntity` only contains the basic details of where the mention was referenced, not the details. However there is a “description” variable which returns a `NEFTheme` object, which is shared by `NEFEntity` objects that are of the same type.

4.2.3 NEFTheme

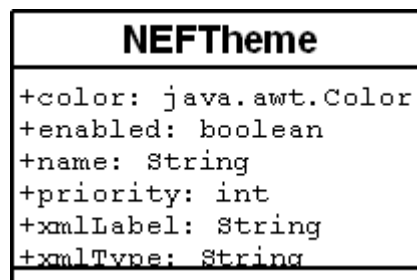


Figure 47 *dsto.nef.entity.document.NEFTheme*

`NEFTheme` is a common description of a particular type of mention (such as a name or person).

`NEFTheme` originally contained methods to determine mentions within a series of tokens. However, in keeping with the principle of separating `NEFDocument` objects and `NEFProcess` objects, this was removed and put into the `PostThemeEntityFinder` (`dsto.nef.entity.process`) class.

4.3 Process Package – Server (dsto.nef.entity.process)

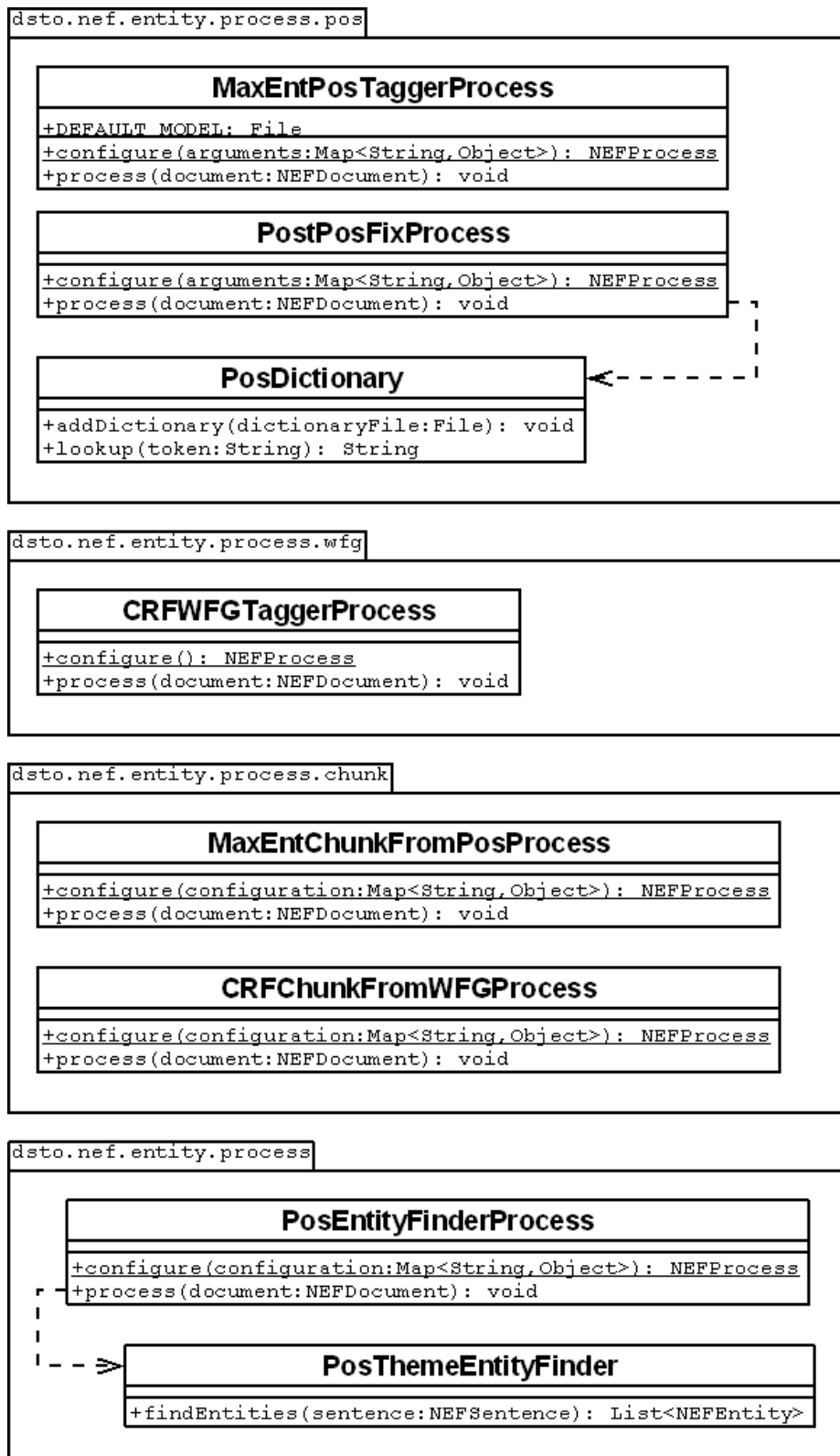


Figure 48 dsto.nef.entity Package

The Entity Process Package, shown in Figure 48, contains all the processes needed to extract references of mentions from a NEFDocument. While it is currently possible to use several different methods for extracting mentions, the commonly used method relies on a series of steps:

1. Determine POS tags.
2. Determine WFG tags.
3. Determine chunk information regions (marked as using the *IOB*⁷ tagsets)
4. Use the resulting information to find entities following defined patterns.

Each process has a number of NEFProcess objects that can carry them out, allowing for experimentation with different machine learning algorithms.

4.3.1 AbstractCRFProcess



Figure 49 dsto.nef.core.process.crf.AbstractCRFProcess

As one of two machine learning approaches employed by the **NEFTool**, *Conditional Random Fields*⁸ (CRF) is a probabilistically-based approach to assigning labels to sequences of tokens, based on the learning of associations between tokens and labels in a training model. The 'CRF++' toolkit [6] is an implementation of CRF used by **NEFTool**. While CRF++ is used for several different processes, the code employed in each process is very similar. So AbstractCRFProcess was created to simplify code and allow for rapid development of CRF - based processes.

Two methods need to be implemented: a method to turn a NEFToken into an array of String Objects, and a second method to add the additional information (also in a String array) back into the NEFToken object.

The size of the array, and the tags set to it, depends on the CRF model file used.

⁷ Tags with a prefix used to markup a sequence in the format: I = Inside, O = Outside, B = Begin.

⁸ CRF can be used for the labeling or parsing of sequential data, such as natural language text [7].

4.3.2 MaxEntPosTaggerProcess

MaxEntPosTaggerProcess
+DEFAULT MODEL: File
+configure(arguments:Map<String, Object>): NEFProcess
+process(document:NEFDocument): void

Figure 50 *dsto.nef.entity.process.pos.MaxEntPosTaggerProcess*

The 'Maximum Entropy'⁹ (MaxEnt) method is an alternative machine learning-model used to assign labels to tokens. MaxEntPosTaggerProcess labels tokens with POS tags using a MaxEnt model. The resulting POS tags are saved to the corresponding NEFToken objects in the TokenContent class.

4.3.3 PostPosFixProcess

PostPosFixProcess
+configure(arguments:Map<String, Object>): NEFProcess
+process(document:NEFDocument): void

Figure 51 *dsto.nef.entity.process.pos.PostPosFixProcess*

The PostPosFixProcess does a look-up of a PosDictionary containing user-defined Tags, and supplements NEFToken objects with “dictionary tags” where appropriate.

MaxEntPosTaggerProcess has also been known to markup a few tokens incorrectly. For instance, the capitalisation of letters and numerals has been known to cause inaccuracies in the POS tagger. These erroneous POS Tags are replaced with the desired value by the PostPosFixProcess.

4.3.3.1 PosDictionary

PosDictionary
+addDictionary(dictionaryFile:File): void
+lookup(token:String): String

Figure 52 *dsto.nef.entity.process.pos.PosDictionary*

PosDictionary is a collection of POS Tags that are loaded via a text file while **NEFTool** is being started. It is used by PostPosFixProcess to determine additional dictionary tags that are relevant to a NEFToken. Dictionary files are lists with each word on a new line.

⁹ “...maximum entropy is a method for analyzing the available information in order to determine the most probable probability distribution...” [8].

4.3.4 CRFWFGTaggerProcess

CRFWFGTaggerProcess
<u>+configure(): NEFProcess</u>
<u>+process(document:NEFDocument): void</u>

Figure 53 *dsto.nef.entity.process.wfg.CRFWFGTaggerProcess*

CRFWFGTaggerProcess finds the word WFG tags using the CRF toolkit, assigning WFG tags on the basis of learnt associations between WFG tags, POS tags and tokens. It is a child class of AbstractCRFProcess in order to reduce coding complexity.

4.3.5 MaxEntChunkFromPosProcess

MaxEntChunkFromPosProcess
<u>+configure(configuration:Map<String, Object>): NEFProcess</u>
<u>+process(document:NEFDocument): void</u>

Figure 54 *dsto.nef.entity.process.chunk.MaxEntChunkFromPosProcess*

MaxEntChunkFromPosProcess uses a MaxEnt model to determine the chunk¹⁰ information in a document. It requires token and POS tag information to generate the tag.

4.3.6 CRFChunkFromWFGProcess

CRFChunkFromWFGProcess
<u>+configure(configuration:Map<String, Object>): NEFProcess</u>
<u>+process(document:NEFDocument): void</u>

Figure 55 *dsto.nef.process.chunk.CRFChunkFromWFGProcess*

CRFChunkFromWFGProcess is an AbstractCRFProcess that determines the chunk information in a document using the CRF++ toolkit, based on prior learning of POS and WFG tag information to generate the tag.

¹⁰ Chunking is the process of segmenting non-overlapping tokens (words) into phrases including noun, verb or preposition phrases, for example the phrase "We saw the yellow dog." can be broken up as [We/B-NP] [saw/B-VP] [the/B-NP yellow/I-NP dog/I-NP][./O]. The individual phrases found by the chunking process are called chunks [9].

4.3.7 PosEntityFinderProcess

PosEntityFinderProcess
<code>+configure(configuration:Map<String,Object>): NEFProcess</code>
<code>+process(document:NEFDocument): void</code>

Figure 56 *dsto.nef.entity.process.PosEntityFinderProcess*

PosEntityFinderProcess finds references to mentions within a document, by using an array of ThemedEntityFinder objects.

Conflicts caused by the different outputs of the ThemeEntityFinder objects are resolved and duplicate entities are removed with the theme of highest priority kept.

4.3.7.1 ThemeEntityFinder

PosThemeEntityFinder
<code>+findEntities(sentence:NEFSentence): List<NEFEntity></code>

Figure 57 *dsto.nef.entity.process.PosThemeEntityFinder*

ThemeEntityFinder was originally part of NEFTheme (used by EntityContent) but was removed to encourage the separation between document and processes. It finds entities by using a combination of dictionary lookup values and patterns of POS and dictionary tags defined in the theme configuration files. It currently only uses POS and dictionary tags to find information, but it will eventually be modified to also use WFG and Chunk tags.

4.4 Process Package – Client (dsto.nef.entity.process.output)

The client processes in the **Entity System** are responsible for outputting EntityContent objects to an external source while batch processing.

4.4.1 FXExportProcess

FXExportProcess
<code>+outputDir: File</code>
<code>+configure(configuration:Map<String,Object>): NEFProcess</code>
<code>+process(document:NEFDocument): void</code>

Figure 58 *dsto.nef.entity.process.output.FXExportProcess*

The **NEFTool** is used to output entity information either to a group of text files which are determined by lists of words that pertain to a theme, or an XML file.

Output is essentially the text component of several NEFEntity objects (with the same theme), placed in a file named [themeName].fx.

4.4.2 XMLOutputProcess

XMLOutputProcess	
+outputDir: File	
+configure(configuration:Map<String, Object>): NEFProcess	
+process(document:NEFDocument): void	

Figure 59 *dsto.nef.entity.process.outpu.XMLOutputProcess*

XMLExportProcess is a more detailed alternative to the FXExportProcess. Output is generated using NEFXMLWriter to generate the output using the path: ./XMLOut/[filename].xml.

The output is intended to allow for future programs to reconstruct the broken-down document. The process is controlled by XMLExportProcess class which uses XMLConstants for all XMLTag labels.

4.5 SwingUI Package (dsto.nef.entity.swingui)

4.5.1 EntityTemplate

EntityTemplate is a simple WindowTemplate for displaying mention information in a document. EntityTemplate objects are intended to be created by EntityTemplateFactory objects, which is further responsible for creating EntityContentPanel objects to be used as the main window.

Nearly all JMenuBar and JToolBar controls used are defined in the **Extras System**. This might be controlled by a type of factory in future revisions.

4.5.1.1 EntityTemplateFactory

EntityTemplateFactory acts like any other WindowTemplateFactory in providing an interface that allows a NEFWindowSession object to create instances of NEFWindow that use EntityTemplate as a layout guide.

4.5.2 EntityContentPanel

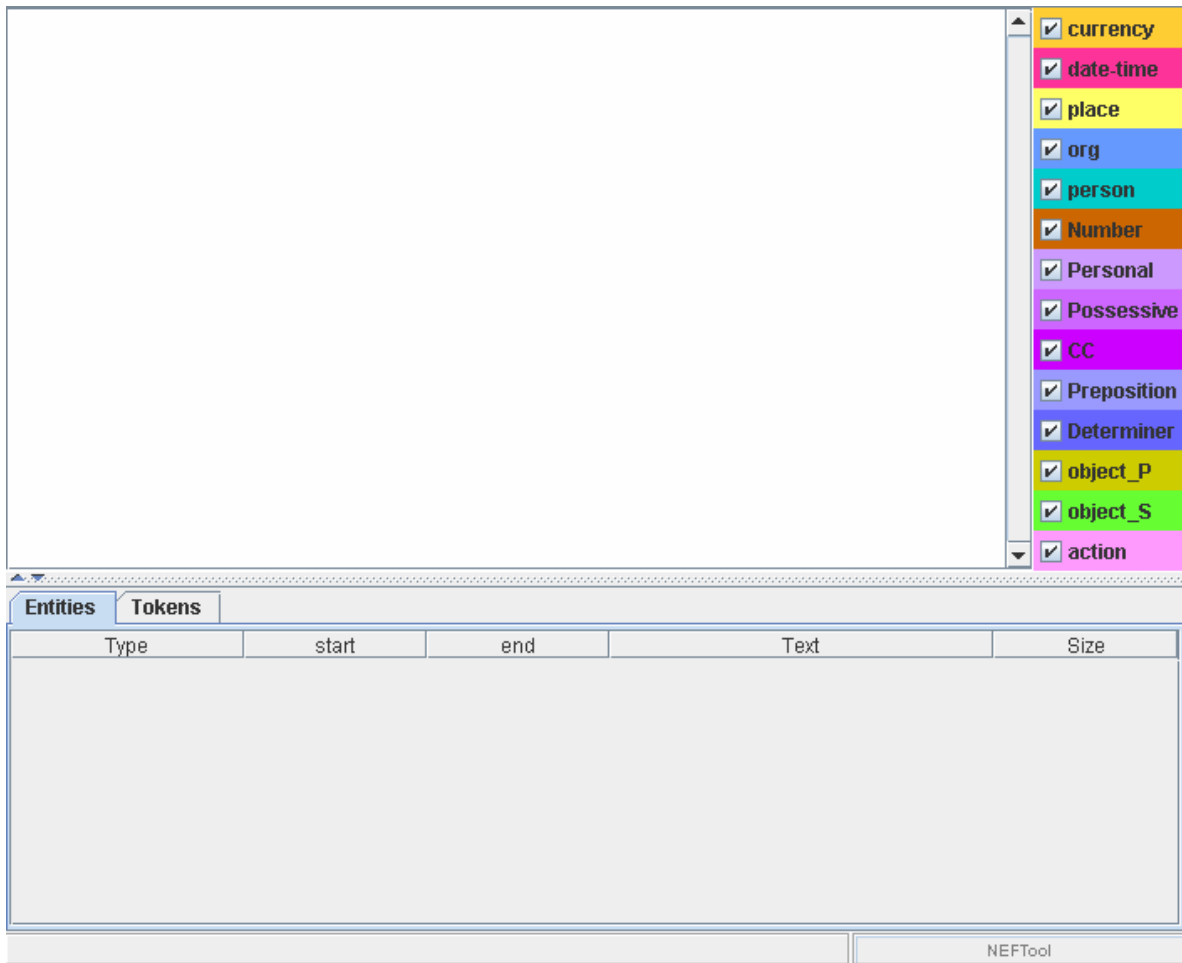


Figure 60 a EntityContentPanel without any document Loaded

EntityContent panel is the main component of an EntityTemplate-based NEFWindow. It consists of a:

- JHighlightedTextArea,
- JThemeChooserPanel,
- JEntitesTable and a
- JTokenTable.

All components are displayed on a single panel divided by a JSplitPane. The JHighlightedText area is set to load documents that are dragged onto its content.

4.5.3 JEntitiesTable

A Simple JPanel, consisting of a JTable and a JScrollPane, which displays all tokens in the document and their associated markup. Table Columns are:

- Type of Entity,
- Start Position,
- End Position,
- Text in Entity,
- Number of Tokens.

All information is received by implementing the DocumentChangeListener and connecting the JEntitiesPanel to a EventController.

Type	start	end	Text	Size
date-time	228	232	today	1
date-time	525	537	two weeks ago	3
date-time	2190	2194	today	1
place	3130	3149	Central Intelligence	2
org	1262	1271	open court	2
org	1362	1371	Government	1
org	2039	2043	court	1
org	2201	2218	Justice Department	2
org	3107	3124	Defense Department	2

Figure 61 Example output from a JEntitiesPanel

4.5.4 JHighlightedTextArea

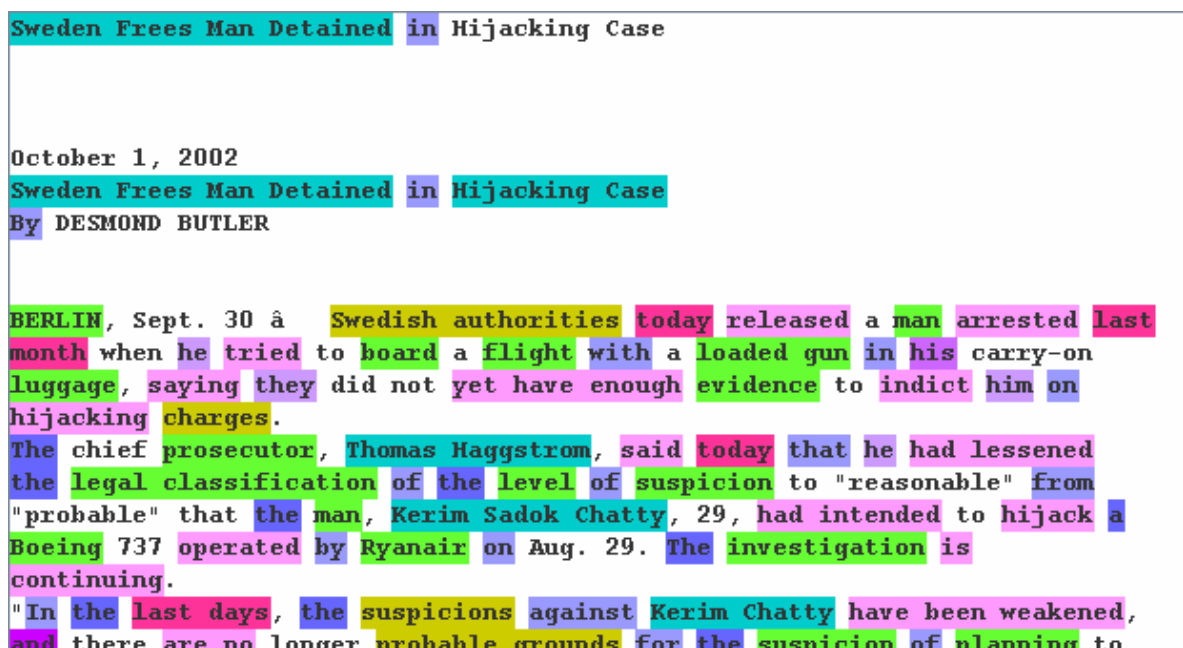


Figure 62 A JHighlightedTextArea showing the Entity Content of a NEFDocument.

JHighlightedTextArea is a JTextArea with added methods to simplify the process of highlighting words with different colours.

4.5.4.1 JHighlightedTextAreaController

HighlightedTextAreaController manages the content of a JHighlightedTextArea component. It provides the logic to convert a NEFDocument objects into a form that HighlightedTextArea can render. This requires that the NEFDocument contains a defined EntityContent “Content” object with entities that have been found.

4.5.4.2 JTarget

This is a drop target listener for any Swing component. An ActionListener can be attached to it for it to report any “drop” Events. It is mainly used for handling open text document commands, caused by a user dropping a text file onto the main display.

5. Extras System

5.1 ServerProcessing Implementations

5.1.1 LocalServer

`LocalServer` is a basic implementation of a `NEFServer` that runs locally within the current runtime of the system. This is the most basic implementation possible of a `NEFServer`.

5.1.2 GenericHandlerSession

`GenericHandlerSession` is a `NEFSession` that acts as a wrapper for a `GenericHandler`. The original version of the **NEFTool** used an interface called a `GenericHandler` to read the contents of files of different formats and is reusable in different systems.

5.2 Process Configuration Framework

This is essentially similar to the **Process Package** described in previous sections. Processes are controllers for the different operations that can be performed on a text document in order to determine the markup.

The development of future processes may lead to different processes performing the same task. As the **NEFTool** is designed to test different processes, it would be useful to “hot-swap” different processes and repeat the previously performed operation without restarting the system.

The Process Configuration Framework provides a framework for swapping processes and also for allowing processes to be added to the system without modifying the source code.

5.2.1 Reading From the File System

The `ProcessConfiguration` can be saved as an XML file which can be read by the `ProcessConfigXMLReader` (`dsto.nef.extras.processconfig.io`) object. The XML file is in the temporary format:

```
<ROOT_ELEMENT>
  <GROUP_ELEMENT GROUP_LABEL="label" GROUP_OPTIONAL="true|false">
    <PROCESS_ELEMENT PROCESS_CLASS="Class" PROCESS_LABEL="label">
      <PARAM_ELEMENT PARAM_TYPE="Converter Type"
        PARAM_VALUE="Converter Value" />
      <!-- Other Parameters -->
    </PROCESS_ELEMENT>
    <!-- Other Candidate Processes -->
  </GROUP_ELEMENT>
  <!--Other Groups-->
</ROOT_ELEMENT>
```

Since converting a PARAM element's PARAM_VALUE to the desired object is a complex process, it needs to be changed; the element's name should be the object type NOT a PARAM label. This was due to a relative inexperience with XML at the time of writing and will be amended at a later date.

5.2.2 ProcessConfigurator

ProcessConfigurator
<code>+clientConfigurator: ProcessConfigurator</code>
<code>+serverConfigurator: ProcessConfigurator</code>
<code>+createOperation(groupName:String, optional:boolean): OperationGroup</code>
<code>+generateQueue(): List<NEFProcess></code>
<code>+getOperations(): List<OperationGroup></code>

Figure 63 *dsto.nef.extras.processconfig.ProcessConfigurator*

ProcessConfigurator is an object-oriented representation of the process configuration of a current system. It is essentially a collection of OperationGroup objects, but it has the extra functionality of being able to generate a queue of NEFProcess objects which can be used by the system.

ProcessConfigurator contains two static instances: one for client processes, and one for server processes. However, no methods are in place to set these preferences to a NEFServer or NEFSession instance.

5.2.3 OperationGroup

OperationGroup
<code>+enabled: boolean</code>
<code>+groupName: String</code>
<code>+optional: boolean</code>
<code>+createProcess(processClass:Class<? extends NEFProcess>): OperationProcess</code>
<code>+generateSelectedProcess(): NEFProcess</code>
<code>+setSelectedProcess(index:int): void</code>

Figure 64 *dsto.nef.extras.processconfig.OperationGroup*

OperationGroup is a collection of OperationProcess "candidates" that can be converted into NEFProcess objects. It allows a OperationProcess to be "selected" as the generator of a single NEFProcess instance.

5.2.4 OperationProcess

OperationProcess
+label: String
+createParam(paramLabel:String,value:Object): OperationProcessParam
+generateInstance(): NEFProcess

Figure 65 *dsto.nef.extras.processconfig.OperationProcess*

This is used for creating “customizable” instances of NEFProcess. OperationProcess contains a list of parameters and the basic details of a process in an OperationGroup.

For a NEFProcess object to be created, it must have a static method with the same name as the STATIC_CONFIGURE_METHOD variable with a single Map<String, Object> argument. This method has been implemented on most NEFProcess objects used in the **NEFTool**.

5.2.5 OperationProcessParam

OperationProcessParam
+method: String
+value: Object

Figure 66 *dsto.nef.extras.processconfig.OperationProcessParam*

Representing a parameter of a NEFProcess, OperationProcessParam is used to set a value for a newly created NEFProcess object before it is returned by an OperationProcess object.

5.2.6 JProcessPanel

JProcessPanel is a JPanel used to “configure” a ProcessConfiguration object.

OperationGroup Objects are displayed as cells containing the following components:

- A JLabel: A display label of the OperationGroup.label variable.
- A JCheckBox: If the OperationGroup is optional, changing the value selects whether the OperationGroup is used or not. If the OperationGroup is not optional, then the JCheckBox is selected and greyed out.
- A JComboBox: If the OperationGroup has several candidate processes, the drop-down menu allows the user to select the desired candidate to use.

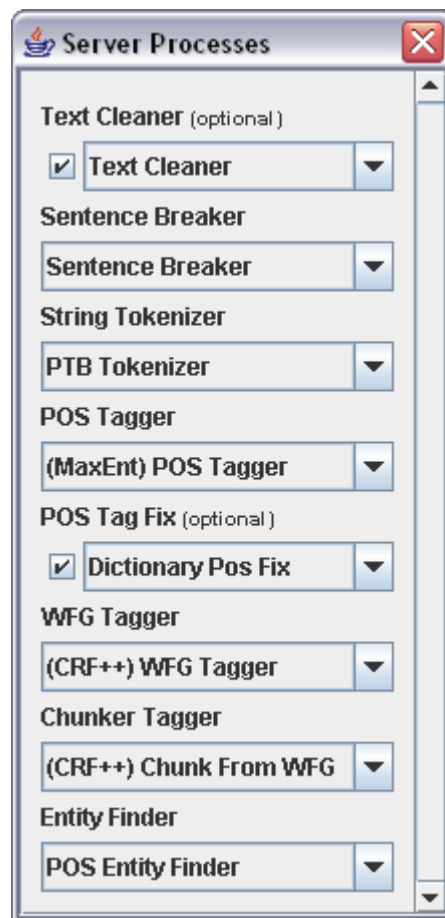


Figure 67 A JProcessPanel showing a Server Configuration

All changes made to the JProcessPanel are immediately set to the ProcessConfiguration object defined in the constructor. It implements the DocumentChangeListener and the DestructionListener, which should be linked to an EventController on creation.

5.3 Batch Control

5.3.1 BatchController

BatchController is a simple controller for managing a BatchProcess on its own independent thread. It uses ActionListener and PropertyChangeListener events in order to update "display" objects of changes in the process queue. A BatchController needs to be created for each batch process as it isn't "thread-safe".

The ActionListener Framework is used for listening to changes made to the batch process.

The processes used are:

BATCH_END	Signals that a batch process has completed its list of tasks.
BATCH_START	Signals that a batch process has started
BATCH_STOP	Signals that a batch process was stopped by the user.

PropertyChangeListeners are used to notify any display components of a change in sequence. The string PropertyValues are:

BATCH_INDEX_VAL	An integer that marks current position in the list of document queue that the BatchController is processing (i.e. 2/50).
BATCH_TOTAL_VAL	An integer value of the number of documents the BatchController is processing (i.e. 20)

5.3.2 JBatchDialog

This is a simple JDialog that pairs with a BatchController to display the progress of a batch process. The dialog is simple, containing the following components:

JLabel	Displays the current progress of the batch process.
JProgressBar	A progress bar that fills during the batch process.
JButton	A button to stop the currently running process.

If several documents cause a ProcessFailedException to be raised, a JStringListDialog is displayed listing the documents that have failed at the end of the process.

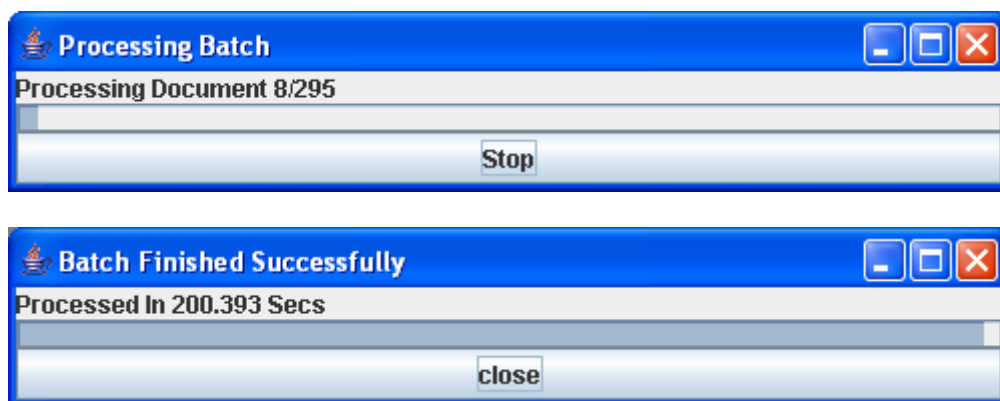


Figure 68 JBatch Dialog processing a Queue

5.3.3 JStringListDialog

JStringListDialog provides a simple list of all of strings in a JTable. It is mainly used to display the failed documents in a batch process.

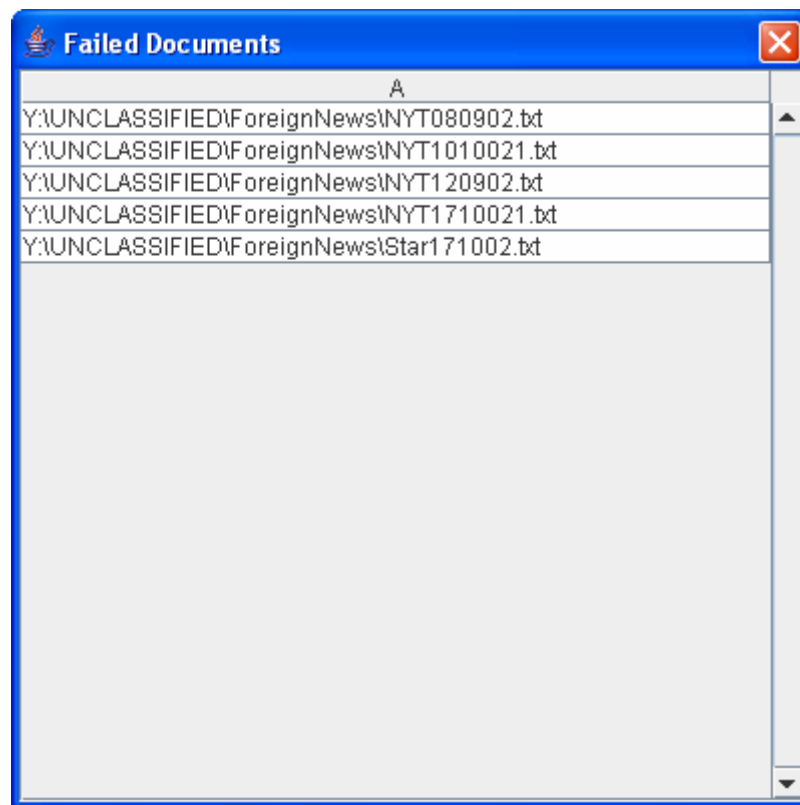


Figure 69 JFailedDocumentDialog presenting a list of failed documents

6. References

- [1] Swing widget toolkit for Java, http://en.wikipedia.org/wiki/Swing_%28Java%29.
- [2] Swing widget toolkit for Java,
<http://java.sun.com/javase/6/docs/technotes/guides/swing/index.html>.
- [3] Unstructured Information Management Architecture,
<http://incubator.apache.org/uima/>.
- [4] Grishman, R. & Sundheim, B, "Message Understanding Conference - 6: A Brief History", <http://acl.ldc.upenn.edu/C/C96/C96-1079.pdf>.
- [5] The Penn Treebank Project, <http://www.cis.upenn.edu/~treebank/>.
- [6] The CRF++ Toolkit, <http://crfpp.sourceforge.net/>.
- [7] Conditional Random Field,
http://en.wikipedia.org/wiki/Conditional_random_field.
- [8] Maximum Entropy, http://en.wikipedia.org/wiki/Maximum_Entropy.
- [9] Bird, S., Klen, E. & Loper, E. "Introduction to Natural Language Processing",
<http://nltk.org/doc/en/chunk.html>.

Appendix A: Starting NEFTool

A static main method for **NEFTool** can be found at `dsto.nef.StartProcessConfiguration`. It loads a dynamically loaded process configuration which is controlled by a `SDIWindowSystem` with an `EntityTemplateFactory`.

The following calls are needed to start the system:

```
ProcessConfigurator serverConfiguration, clientConfiguration;

/* Display Warning Message - Insecure System */
Toolkit.getDefaultToolkit().beep();
String message = "WARNING: XMLExportProcess (A Client Process) outputs
documents to ./XMLOut/\n"
    + "!!! NOT RECCOMENDED FOR >= RESTRICTED DOCUMENTS !!!";
JOptionPane.showMessageDialog(null,message,"NEFTool",
    JOptionPane.WARNING_MESSAGE);

/* Setup basic logger setttings */
BasicConfigurator.configure();
Logger.getRootLogger().setLevel(Level.WARN);
Logger.getLogger(EventController.class).setLevel(Level.DEBUG);
Logger.getLogger(DefaultSessionControllerModel.class)
    .setLevel(Level.DEBUG);

/* Setup the Process Environment */
CoreEnvironment coreEnvironment = CoreEnvironment.start();

/* Load the Server Configuration */
File serverConfigFile = new
File(CoreEnvironment.SUPPORT_DIRECTORY,"process_server.cfg.xml");
serverConfiguration = ProcessConfigXMLReader.getConfig(serverConfigFile);
ProcessConfigurator.setServerConfigurator(serverConfiguration);

/* Load the Client Configuration */
File clientConfigFile = new File(CoreEnvironment.SUPPORT_DIRECTORY,
    "process_client.cfg.xml");
clientConfiguration = ProcessConfigXMLReader.getConfig(clientConfigFile);
ProcessConfigurator.setClientConfigurator(clientConfiguration);

/* Create the Server */
LocalServer server = new LocalServer();
server.setProcesses(serverConfiguration.generateQueue());
coreEnvironment.setServer(server);

/* Start the Window System */
EntityTemplateFactory templateFactory = new EntityTemplateFactory();
SDIWindowSystem system = new SDIWindowSystem(templateFactory);
CoreEnvironment.getInstance().setWindowSystem(system);

system.newWindow();
```

Appendix B: Programming Guidelines

B.1. General

- Exceptions are extremely useful when used properly; errors should not be ignored, because they may cause inaccuracies in the final outcome. Exceptions should always be reported in such a way that a developer might find useful.
- Code either works properly or not at all: there are few things more frustrating than trying to discover a problem that was concealed several methods' calls previously.
- Log4J (<http://logging.apache.org/log4j/>) is extremely useful for reporting abnormal behaviour of the system as well as for tracing standard operation. It should be used where possible.

B.2. Document Package

B.2.1 NEFDocument

- This class should really be fine as it is; do not add extra information to it that could be handled by a NEFContent Object.

B.2.2 NEFContent

- NEFContent objects only contain information; it should never be responsible for generating/parsing this information. However, it should be able to validate the information being sent to it.
- It is recommended that NEFContent objects are declared as final. This stops potential problems that could be caused by inheritance and the `NEFDocument.getContent(Class)` method.

B.3. NEFProcess Package

B.3.1 NEFProcess

- Ensure NEFProcess objects do only one task. This allows for NEFProcess objects to be swapped with other NEFProcess objects that do the same task.
- A NEFProcess object must not have a "memory" of previous documents. It should be possible to make all methods static (not variables) in order to test this.
- NEFProcess objects should be made "thread-safe". Adding a synchronized tag to the `process(NEFDocument)` method helps in this regard.
- Any possible errors should be wrapped in a `ProcessFailedException` and thrown back to the user. The NEFWindow GUI has dialogs for displaying `ProcessFailedExceptions` to the user.
- A "public static NEFProcess configure(Map<String, Object> configMap)" should be provided. This allows for the Process Configuration Packages in "NEF Extended" to dynamically generate NEFProcess objects, which allows the user experiment with different NEFProcess objects during runtime.

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA					
				1. PRIVACY MARKING/CAVEAT (OF DOCUMENT)	
2. TITLE NEFTool: System Design			3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION) Document (U) Title (U) Abstract (U)		
4. AUTHOR(S) Benjamin Morrall			5. CORPORATE AUTHOR DSTO Defence Science and Technology Organisation PO Box 1500 Edinburgh South Australia 5111 Australia		
6a. DSTO NUMBER DSTO-TN-0789		6b. AR NUMBER AR- 014-048		6c. TYPE OF REPORT Technical Note	
				7. DOCUMENT DATE November 2007	
8. FILE NUMBER 2007/1138993/1		9. TASK NUMBER CCT07/201		10. TASK SPONSOR EXEC DIR CTSTC	
				11. NO. OF PAGES 47	
				12. NO. OF REFERENCES 8	
13. URL on the World Wide Web http://www.dsto.defence.gov.au/corporate/reports/DSTO-TN-0789.pdf			14. RELEASE AUTHORITY Chief, Command, Control, Communications and Intelligence Division		
15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT <i>Approved for public release</i>					
OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111					
16. DELIBERATE ANNOUNCEMENT No Limitations					
17. CITATION IN OTHER DOCUMENTS Yes					
18. DSTO RESEARCH LIBRARY THESAURUS http://web-vic.dsto.defence.gov.au/workareas/library/resources/dsto_thesaurus.htm Information Extraction, Machine Learning, Framework.					
19. ABSTRACT The text processing team from the Intelligence Analysis discipline has experimented with the viability of using machine-learning models to automatically tag English words with syntax and functional labels within a text document. The NEFTool was developed to assist with testing different machine-learning models. The system has a modular architecture, and was designed to be extensible, allowing support for rapid prototyping and for new functionality to be added as required to support research in text processing.					