SimpleNLGv4

Change History:

Version	*Date*	*Author*	*Reason*
0.1	21/10/09	D. Westwater	Original file

Table of Contents

Introduction

The purpose of this document is to give an overview of the initial design concept for the SimpleNLGv4 architecture. The main documentation for the actual implementation consist of the generated JavaDoc, which exists in two formats: user model and developer model. The user model gives a user's view of how to use the architecture (i.e. the public API), whereas the developer model gives a description on the entire code.

Why the Need?

The primary reason for rewriting the architecture of SimpleNLGv4 comes from the desire for a more modularised approach to the system. As it stands, the code does not fully represent the different stages involved in realising text in natural language. It is therefore very difficult, approaching the nearly impossible, for external developers to extend the application by writing their own modules. Thus, users are restricted to only being able to use the system 'out-of-the-box'.

A secondary reason for the requiring the modularisation of the system comes from IPR protected. Currently, SimpleNLG contains code from an external source that can only be used for non-commercial and research purposes. Having a more modularised system will allow commercial applications to write their own version of protected code.

Finally, it is an aim of this restructuring to enhance the documentation of the system. This applies to the tutorials for using the base system and the customisation of the system; and the API documentation, which frequently falls out of date with on-going development.

So What's New?

As far as is feasible to do so, SimpleNLGv4 will be backwards compatible with version v3.8. It is one of the additional requests placed on the restructuring. A lot of the changes to the architecture will be done behind the scenes from the user's point of view. Existing classes and methods will become wrappers around the new architecture to ensure v4 still runs on systems built with v3.8. New development will be encouraged to use the new architecture, meaning any undesirable classes and methods will be deprecated.

Overview

The development work for the new architecture is split into three key areas:

Control Framework	The control framework contains the base classes and interfaces that define what it means to be a SimpleNLG module or core element.
Lexicon	The lexicon forms the link between the SimpleNLGv4 system and the stored *word list*. The Lexicon part contains the interface defining the lexicon and some hard-coded examples of accessing word lists.
Predefined Modules	Although the main focus of the architecture is to open up the system to make it easier to customise it should still be possible to run the system 'out-of-the-box'. In order to do this some modules need to be included.

A Note About Utilities

At the lowest level of the new architecture, a series of utility tools will be introduced. These tools

are the generic libraries that will aid the system in specific tasks. These tools have already been written for use on the BabyTalk project and can be readily used by SimpleNLGv4.

Three utilities that will be of particular use are:

Aries Basic DOM and XPath utilities for reading, writing and parsing XML file. Parts of the new architecture will have XML files to define the configuration.

Ban A database handler interface that will be useful for the Lexicon.

Carados A generic ontology interface allowing applications to communicate with any type of ontology.

Control Framework

The control framework contains the base classes and interfaces that define what it means to be a SimpleNLG module or core element. Having a core element definition ensures that all the modules have the same expectations on what the data is like, while having a base definition of a module allows the modules to execute other modules.

The classes and interfaces defined here will be placed in the *simplenlg.framework* package.

NLGElement

NLGElement is an abstract class that represents all the components of language. This covers both the syntactical components and the structural components of documents, two examples being Noun Phrase (syntax) and Paragraph (document structure). The NLGElement can refer to other elements in a tree-like fashion. Thus a top-level document element can contain a series of sections, which in turn lead further down the tree to paragraphs, sentences, various types of phrases and eventually to the words themselves.

Fields

TYPE	NAME	DEFINITION
String	baseForm	For words, this represents the base form of the word used before it is has been modified by any modules.
ElementCategory	category	Defines the type of NLGElement this is.
List<NLGElement>	complements	This is effectively the child components of this element. The complements put together form this type of element. For example, in a phrase the complements might represent the individual words.
HashMap<String, Object>	features	A list of features relevant to this NLGElement. Features are mapped on a one-to-one basis, i.e. there cannot be two features with the same name. The value, however, may be a collection or an array.
HashSet<String>	flags	Boolean features could be added to the features map as mentioned above. However, this field has a more convenient way of handling the boolean flags. Only those flags that are **true** are stored in the set.
NLGElement	head	Used primarily for phrases, the head represents the main item of the phrase.
NLGElement	parent	A reference to the parent node in the *tree*.
String	realisation	The current realisation of the element. In other words, the textual representation of the element.

Methods

RETURN	NAME	PARAMETERS	DEFINITION
void	addComplement	NLGElement	Adds a *child* element to this element.
void	clearAllComplements		Removes all the *child* elements from this element.
void	clearAllFeatures		Removes all features from this element.
void	clearAllFlags		Removes all flags from this element, effectively setting flags to **false**.
String	getBaseForm		Returns the base form of the word.
ElementCategory	getCategory		Retrieves the type of this element.
List<NLGElement>	getComplements		Returns the *child* elements of this element.
Object	getFeature	String	Returns the value of the named feature as an object.
String	getFeatureAsString	String	Returns the value of the named feature as a string representation.
NLGElement	getHead		Returns the head element of this element.
NLGElement	getParent		Returns the parent element of this element.
String	getRealisation		Returns the current realisation of this element.
boolean	hasFeature	String	Determines if this element has a value associated with the named feature.
boolean	isA	ElementCategory	Determines is this element is of the named type.
boolean	isFlagTrue	String	Determines if the named flag is **true**.
void	removeComplement	NLGElement	Removes the named *child* from this element.
void	removeFeature	String	Removes the named feature from this element.
void	removeFlag	String	Removes the named flag from this element.
void	setBaseForm	String	Sets the base form of this element.
void	setCategory	ElementCategory	Sets the type of this element.
void	setComplement	NLGElement	Removes all existing *children* and adds only this *child*.
void	setFeature	String, Object	Sets the named feature of this element to be the given object.
void	setFlag	String, boolean	Sets the named flag of this element to be the given boolean value.
void	setHead	NLGElement	Sets the head element of this element.
void	setParent	NLGElement	Sets the parent element of this element.
void	setRealisation	NLGElement	Sets the current realisation of this element.

NLGModule

NLGModule is an abstract class defining an individual module of the SimpleNLGv4 framework. All implemented modules ***should*** extend from this class. It is possible to link modules together by adding modules to existing modules. The added modules will be executed ***before*** the main module itself. Modules can be added with a notion of salience-like organisation, where the module with the lowest salience value is executed before any other module.

Fields

TYPE	NAME	DEFINITION
static final String	SALIENCE_MODIFER	The default modifier to salience values primarily used when adding new modules to the beginning or end of the list of current modules. Value is 100.
TreeMap<Integer, NLGModule>	modules	The list of modules that need to be executed before this module is run.

Methods

RETURN	NAME	PARAMETERS	DEFINITION
int	addModule	NLGModule	Adds a module to the end of the tree of current modules. The applied salience value is returned.
NLGModule	addModule	NLGModule, int	Inserts a module into the tree at the relevant point for its salience. If the tree already contains a module with that salience it is overwritten and the old module is returned.
void	clearModules		Removes all modules from the tree.
Collection<NLGModule>	getModules		Returns the collection of modules associated with this module in the preserved salience order.
abstract void	initialise		Allows the module to perform some initialisation before it is run.
abstract String	realise	List<NLGElement>	This the main processing thread of the module and it performs its actions on the given list of NLGElements.
abstract String	realise	NLGElement	This the main processing thread of the module and it performs its actions on the given NLGElement.
NLGModule	removeModule	int	Removes the module with the given salience from the tree. The removed module is returned.
NLGModule	removeModule	NLGModule	Removes the named module from the tree of modules. The removed module is returned.

ElementCategory

ElementCategory is an interface that should be implemented by classes or enumerated types used to define the syntactical or structural categories. It is anticipated that the category definitions themselves will be enumerated types. Having this interface allows several enumerated types to be used (such as having a separation of the syntactical structure and document structure) while ensuring the modules have a generic access to all these types.

Methods

RETURN	NAME	PARAMETERS	DEFINITION
boolean	equalTo	Object	Determines if the given object is equal to this category.

The Lexicon (by Ehud)

The lexicon is a database of information about words. Data about each word includes

- Base form (eg, "dog")
- Syntactic category (eg, "noun")
- Unique ID (internally generated, or from DB)
    - We can't use base form, as there is a verb "dog" as well as a noun "dog". I'd also like to leave open the possibility of multiple entries for the same base-form/cat (eg, one "mouse" for animal, another "mouse" for computer gadget); I'm not sure we'll go down this route, but lets keep this possibility open.
- Feature values. This includes
    - Orthographic info (capitalisation, a vs an)
        - Acronyms, spelling variants (eg, UK vs US) have their own lex entries, which are linked to main entry
    - Morphological info (inflection type, irregular forms)
    - Syntactic info (subcat (eg count vs mass), complements, adj position, etc)
    - Semantic info possible in future (eg, link to corresponding KB entry)

Classes

Word (preferred) or *LexicalItem*

Currently LexicalItem is an interface implemented by the class Word, maybe best just to have Word?): getters for above information. This class can also define getters for common orthographic properties. We should also have subclasses of for common syntactic categories (Noun, Verb, etc), which have getters for common properties for this type of word (eg, getPlural for noun, isTransitive for verb).

Currently these classes also have setters. I suggest we drop the setters, and say that the lexicon must be loaded from a DB or file.)

NOTE: There is a general design issue here. I'd like to have getters for common features, but I don't want to have getters for all features, because some of these are pretty rare, and I don't want to clutter the class. Also we should always allow the possibility of features which don't have getters, because users may add additional features to hand-built lexicons (or new features may appear in future releases of the NIH lexicon). But I don't know where to draw the line, maybe we need to think of a general policy? Of course very similar concerns apply to NLGElements such as SphraseSpec.

Lexicon

This is a generic interface/class which allows users to

- Get a Word, given its base form (and optionally syntactic category). Probably want one method to retrieve all matching Words, and one to retrieve just the first matching word
- Get a Word given its unique ID
- Get a Word given an inflected form (eg, "dogs" instead of "dog") and syntactic category. This method may be pretty unreliably initially, but I'd like to include it in the API

The current Lexicon class includes a lot of methods for getting morphological information (rules, plurals). These should be dropped from the Lexicon class, I would like morphology in simplenlg 4 to be done by a processor, not by the lexicon.

We should also have a defaultLexicon() static method, which returns an XMLLexicon which is loaded from a defaultLexicon.xml file which is included in the jar. This will include entries for the 10000 (?) most common words in English (including all function words), extracted from the NIH Specialist lexicon

DBLexicon

A class to get words from the NIH Specialist Lexicon. This should handle DB stuff, and also provide caching (ie, entries should be retrieved by the DB just once, and then stored in a cache). The current simplenlg uses a modified version of the Specialist lexicon, I'd like to check if it is possible to instead use the downloaded version (so users can use new releases of Specialist as they come out, without us needing to worry about modifying each new release).

XMLLexicon

A class to load words from an XML file, in the XML format used by NIH Specialist Lexicon. I guess we could also support an ASCIILexicon, using the Specialist ASCII format, but I think life will be easier if we stick to just one text format, XML. The entire file is read at load time (with appropriate index maps created).

MultiSourceLexicon

A class to represent multiple lexicons; basically this looks for the Word in each component lexicon, in the order in which they were added to the MultiSourceLexicon. The purpose of this is to allow users to create their own domain-specific lexicons, and be able to look for words both in these and in a general lexicon

Predefined Modules

Syntactic Module

The Syntatic Module is used to construct the syntax of the text through the creation of different types of phrases, e.g. noun phrase, verb phrase, adjective phrase. Calling the realise on these elements will return the base form of all the elements.

Aside from being an extension to the NLGModule class, the syntactic processor also contains a number of supporting classes for defining the language, such as a number of enumerated types for representing particular features (gender, form, tense). These will all appear under nlg.syntax.english. It should be possible to create additional syntax for other languages.

Orthographic Module

The Orthographic module takes the basic text and adds the punctuation and capitalisation. For example, "the courier delivered the green bicycle to Mary" would become "The courier delivered the green bicycle to Mary." The first letter of the sentence is capitalised and the sentence is terminated with a period. These actions are performed upon calling the realise method.

Morphological Module

The Morphology module controls the correct form of words. It handles pluralisation of change tense between the words. For example, the sentence "Mary chases Charles." is in the present tense if we wanted to make this past tense, the morphology module would convert "chases" to "chased" so the sentence would become "Mary chased Charles."

Aggregation Module

There may be scope within the time-scale of the project to introduce a basic aggregation module to SimpleNLG. Some code for handling aggregation was added to v3.8 but it is unclear how advanced or reliable this code is. Simple aggretation will join sentences together, typically through the use of 'and', 'or' and 'but'. However, it might be viewed that Aggregation correctly belongs in a microplanner and not in the realisation.

Default Realiser Module

The default realiser module essentially links together the above mentioned modules. So a user can construct their elements and run the realiser. It will call each module in turn to produce the final fully realised string.

Formatting Modules

Two types of formatting modules could be added to the SimpleNLG package. One formatter would create HTML mark-up in the returned text, e.g. paragraph tags "<p>" and lists "<ul>". A second formatter would create XML mark-up for storing the produced text into an appropriate XML file.