MMTx_API Documentation

For Release 2.3

Guy Divita

Table of Contents

Introduction

MMTx maps text to UMLS Metathesaurus concepts. As part of this mapping process, MMTx tokenizes text into sections, sentences, phrases, terms, and words. MMTx maps the noun phrases of the text to the best matching UMLS concept or set of concepts that best cover each phrase.

UMLS
Metathesaurus Concepts

A UMLS Metathesaurus Concept is a set of terms that roughly mean the same thing, That is, a concept is a grouping of synonymous terms. One of the terms from each set of synonymous terms is chosen as the label for the concept. We refer to this term as the concept name. Each set of synonymous terms, or concept, is labeled with a unique identifier, known as a concept unique identifier, also known as a cui, sometimes pronounced as a “Cueie”

A UMLS Metathesaurus term, in turn, is really a grouping of strings that are lexical variants of each other. These groupings include strings that differ by case, inflection, and minor spelling and punctuation differences. Each grouping of strings in this way also gets a unique identifier, a lexical unique identifier, or LUI, sometimes pronounced as a “Lueie”. A representative from this grouping of strings is chosen as the term name.

A UMLS Metathesaurus string is text, or terms that comes from one or more Medical controlled vocabularies. That is, a UMLS Metathesaurus string is a grouping of logographically equivalent strings that come from one or more source medical vocabularies. Each UMLS Metathesaurus string is labeled with a unique identifier, or string unique identifier, also known as a SUI, sometimes pronounced as a “sueie” It should be noted that there is no meaning associated with the strings. What gives these strings meaning is what groups of terms they get assigned to. To make this point, a string, such as the string “aids” is attached to three concepts, C0021588-Artificial insemination by donor, C0001175-Acquired Immunodeficiency Syndrome, and C0449435-Manufactured aid.

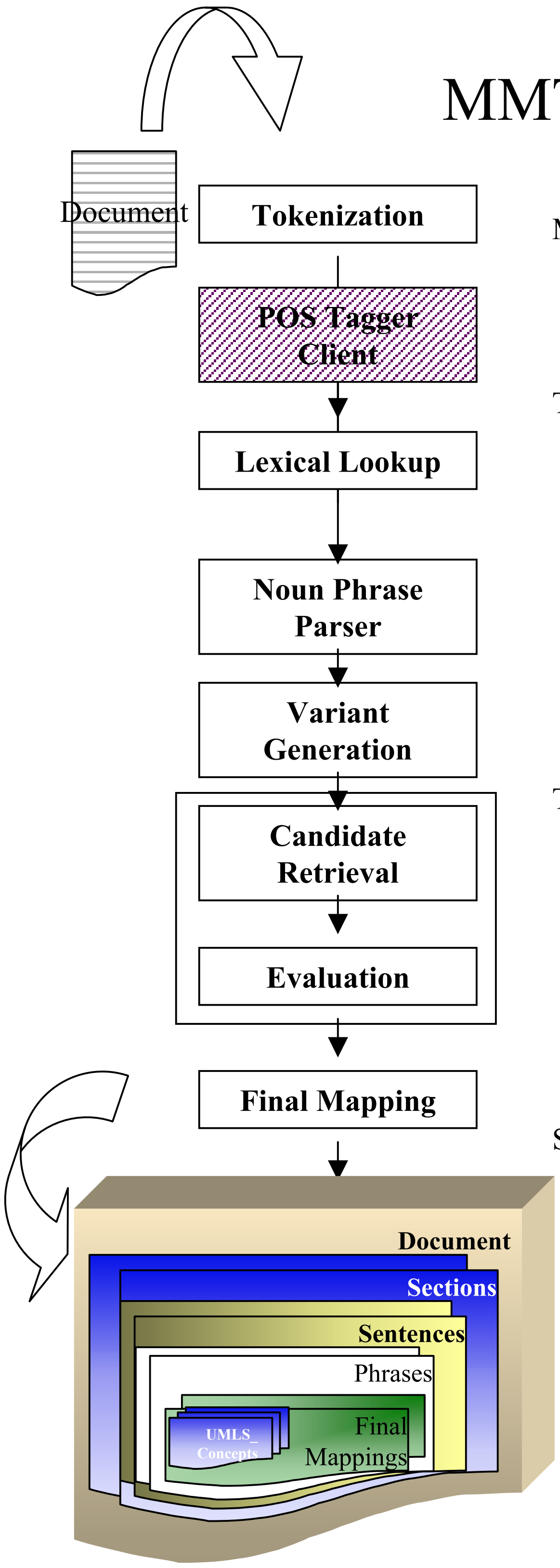
UMLS
Metathesaurus Concepts

Each UMLS Concept has been labeled with one or more UMLS Semantic Types. A UMLS Semantic type is a category that comes from the UMLS Semantic Network. The UMLS Semantic network is a rough or high level categorization of the medical domain. The Semantic Network could be thought of as a surrogate for an upper level domain ontology for medicine. The semantic network is composed of 135 semantic types and 54 relationships binding them together. The primary link in the Network is the "isa" link. This establishes the hierarchy of types within the Network and is used for deciding on the most specific semantic type available for assignment to a Metathesaurus concept. In addition, a set of non-hierarchical relations between the types has been identified. These are grouped into five major categories, which are themselves relations: "physically related to", "spatially related to", "temporally related to", "functionally related to", and "conceptually related to".

The assigned UMLS Semantic Types associated with each concept are an aid to clarify meaning and the context that the concept comes from.

This bit of background is needed to understand the magic behind MMTx. MMTx matches phrases to the closest matching or best covering UMLS Metathesaurus Strings. Because each UMLS Metathesaurus String is associated with one or more UMLS Metathesaurus Concepts, MMTx retrieves the relevant concept and semantic type information for later display.

MMTx Overview



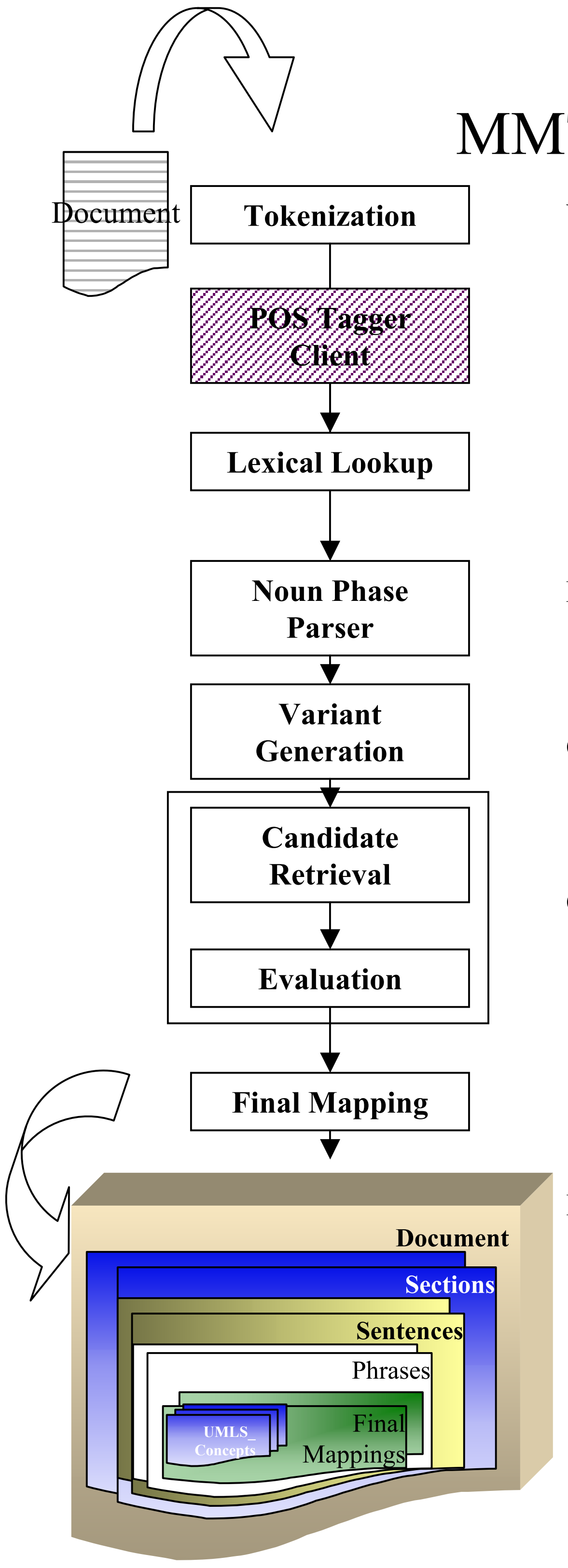
MMTx passes text into tokenizer that tokenize the text into sections containing sentences and sentences containing word tokens.

The sentences can be passed to a part of speech tagger client to have the part of speech tags from a tagger matched up with and assigned to the word tokens. MMTx does not have a part of speech tagger, (hence the grayed out Tagger client box in the diagram) but does include hooks to a tagger via a programming interface.

The sentence's word tokens are matched against terms from the SPECIALIST Lexicon in the Lexical Lookup module to combine the word tokens into multi-word terms and retrieve their parts of speech. The result are lexical elements that are made up from the word tokens.

Sentences are passed to a noun phrase parser (or chunker) to tokenize into phrases. The noun phrase parser is a barrier category parser that uses the part of speech categories from lexical elements and the part of speech tags from a tagger if used. The result are phrases that are made up of the lexical elements. These phrases put into the sentences they came from.

MMTx Overview



Variants, including synonyms, spelling, derivations, inflections, acronym and abbreviations, acronym and abbreviation expansions and recursive combinations of these, are retrieved for the words and lexical elements of each phrase in a variant generation module. The result of this process are phrases containing variants. Each variant is marked with a cost or distance of how many transformations it took to get from the original form to the variant form.

Phrases and their variants are used to retrieve UMLS Strings that match. The set of UMLS Strings that match a phrase, or a variant of the phrase are called candidates.

Candidates are are evaluated against each phrase based on several criteria. The result of this step is a score between 0 and 1000, with 1000 denoting an exact match.

Concept and semantic type information is gathered for each candidate. The result of this candidate generation and evaluation step is a set of UMLS_Concept_Pointer associated with each phrase. Each UMLS_Concept_Pointer includes an evaluation score, a set of UMLS_String_Pointer, and a set of UMLS_Semantic_Type_Pointer.

If the phrase was not completely covered by one candidate UMLS String, combinations of candidates are put together to best cover the phrase in a final mapping module. The result of this step is a set of Final_Mapping. Each Final_Mapping includes the set of UMLS_Concept_pointers pointing to the concepts that best covers the phrase. Each Final_Mapping also includes a final mapping score.

MMTxAPI

The MMTxAPI is the recommended way to embed MMTx within other applications. There are methods to map text from an entire document along with methods to map a single term, and many in between. This is an instance based class, to be instantiated once, with the methods of this instance being called over and over again.

Options and settings are passed into the MMTxAPI via the MMTx/config/MMTxRegistry.cfg configuration file, and via a String array containing MMTx command line options.

Constructor Summary
MMTxAPI() Constructor for MMTxAPI. This constructor takes no parameters. All MMTx Options and settings are picked up from the MMTx/config/MMTxRegistry.cfg file.
MMTxAPI(String[] args) This constructor takes a String[] that is filled with MMTx command line arguments. Options and settings for the MMTxAPI are taken from the MMTx/config/MMTxRegistry.cfg file as well as the command line arguments. The command line arguments override the configuration file settings.

MMTxAPI

Once instantiated, the instance takes as input either a container such as an already made Document, Section or Sentence instance or a text. The methods that take the container classes as input add to that same container. The methods that take text as input return a container instance. .The container classes including Document, Sentence, and Phrase will be covered in greater detail after the simple examples.

MMTxAPI

MMTxAPI()
MMTxAPI(String[] args)

void	processDocument	(Document	pDocument)	
Document	processDocument	(File	pFile)	
Document	processDocument	(String	pDocumentText)	
Void	processSection	(Section	pSection)	
void	processSentence	(Sentence	pSentence)	
Sentence	processSentence	(String	pSentenceText)	
Sentence	processString	(String	pString,	boolean pTermProcessing)
void	processStringAux	(Sentence	pSentence,	boolean pTermProcessing)
Phrase	processTerm	(String	pTerm)	

MMTxAPI Simple Examples

Suppose you have a term, sentence or document already in hand and would like to retrieve the best mapping to UMLS Concepts for it. This is a common use of MMTx. The forthcoming examples, MMTxAPI methods are shown to map from a term, a sentence or a document. The full versions of these programs can be found in the MMTx/examples directory of the distribution. Note that not all the classes have been introduced. The details of those yet to be defined classes come after these examples.

A few implementation details need to be gotten out of the way to get these programs working.

Classpath	The classpath needs to include the MMTx installation's config directory and the path to the MMTx.jar file. For example: java -cp "**/nls/MMTx/config;/nls/MMTx/lib/MMTxProject.jar**" MMTxTermExample

Imports	Each of these programs need to import the following packages. These imports are not shown in the examples that follow due to space and clarity considerations
	
	`import  java.util.*;`
	`import java.io.*;`
	
	`import gov.nih.nlm.nls.utils.U;            // - --------------------`
	`Import gov.nih.nlm.nls.nlp.textfeatures.*; // - Included in MMTx.jar`
	`import gov.nih.nlm.nls.MMTx.MMTxAPI;       // - --------------------`

MMTxAPI Term Example

This program creates a new MMTxApi instance, then uses the processTerm method with a string parameter. This method returns an analized phrase that includes final mappings and UMLS concept references. Here we are printing out the analysis with an example method that demonstrates how me might get at various pieces of information on the analysis and print it. It should be noted that there are a number of formatting methods on each of the container classes. We could have minimialistically used the method aPrhase.toString() to print out the contents of the analysis.

```
// =======================================+ Create a MMTxAPI object +==
MMTxAPI MMTx = new MMTxAPI( );

// ==============================================+ Analyze the Term +==
Phrase aPhrase = MMTx.processTerm("sleep disorders" );

String phraseString = displayPhrase( aPhrase );

System.out.println( phraseString );
```

MMTxAPI Term Example

This method demonstrates itterating through and printing out each final map. A final map is a combination one or more UMLS Metathesaurus concepts that best covers or maps to the phrase. This will be explained in greater detail in a subsequent section. In this example, we are relying on the FinalMapping's toString() method to print out some reasonable output.

```
// ===================================+ Display Phrase and Concepts +==
String displayPhrase( Phrase aPhrase ) throws Exception {

// ==============================================+ Get the Mappings +==

List finalMappings = aPhrase.getFinalMappings();

if ( finalMappings != null ) {
  Iterator mappingIterator = finalMappings.iterator();

  // ================================+ Iterate through the Mappings +==
  while (mappingIterator.hasNext()) {
     FinalMapping aMapping = (FinalMapping) mappingIterator.next();
     System.out.println( aMapping );
     }
  }
}
```

MMTxAPI Sentence Example

This example shows the existence of a processSentence method. If your input was a bunch of already determined sentences, this method would analyze them without having to create a document. An instance of the container class sentence is returned from the processSentence method. There are methods to traverse through the components of a sentence, as in this example, the getPhrases() method.

```
// ==========================================+ Create a MMTxAPI object +==
MMTxAPI MMTx = new MMTxAPI(  );

// ============================================+ Analyze the Sentence +==
Sentence aSentence = MMTx.processSentence("Insomnia is a symptom of a sleep
    disorder" );

Iterator phraseIterator = aSentence.getPhrases().iterator() ;

// ====================================+ Iterate through the Phrases +==
while ( phraseIterator.hasNext() ) {
  Phrase aPhrase = (Phrase) phraseIterator.next();

  System.out.println( displayPhrase( aPhrase) );

}
```

MMTxAPI Document Example

This example demonstrates one of the ways to process a document. There is a processDocument method that takes a FILE instance as a parameter. There exists a processDocument method that takes a String as a parameter as well. These methods return an analyzed document via an instance of the container class Document. Note that there is a convenience class off the document class that returns all the phrases from the document.

```
// ====================================================+ Get a document +==
File aFile = new File("example.txt");

// =============================================+ Create a MMTxAPI object +==
MMTxAPI MMTx = new MMTxAPI(  );

// ================================================+ Analyze the Document +==
Document aDocument = MMTx.processDocument(aFile);
Iterator phraseIterator = aDocument.getPhrases().iterator() ;

// ========================================+ Iterate through the Phrases +==
while ( phraseIterator.hasNext() ) {
  Phrase aPhrase = (Phrase) phraseIterator.next();
  System.out.println( displayPhrase( aPhrase) );
}
```

Container Classes and Processes

This section describes MMTx's Container classes by describing how MMTx uses and populates these classes when you call one of the process methods:

processDocument

processSentence

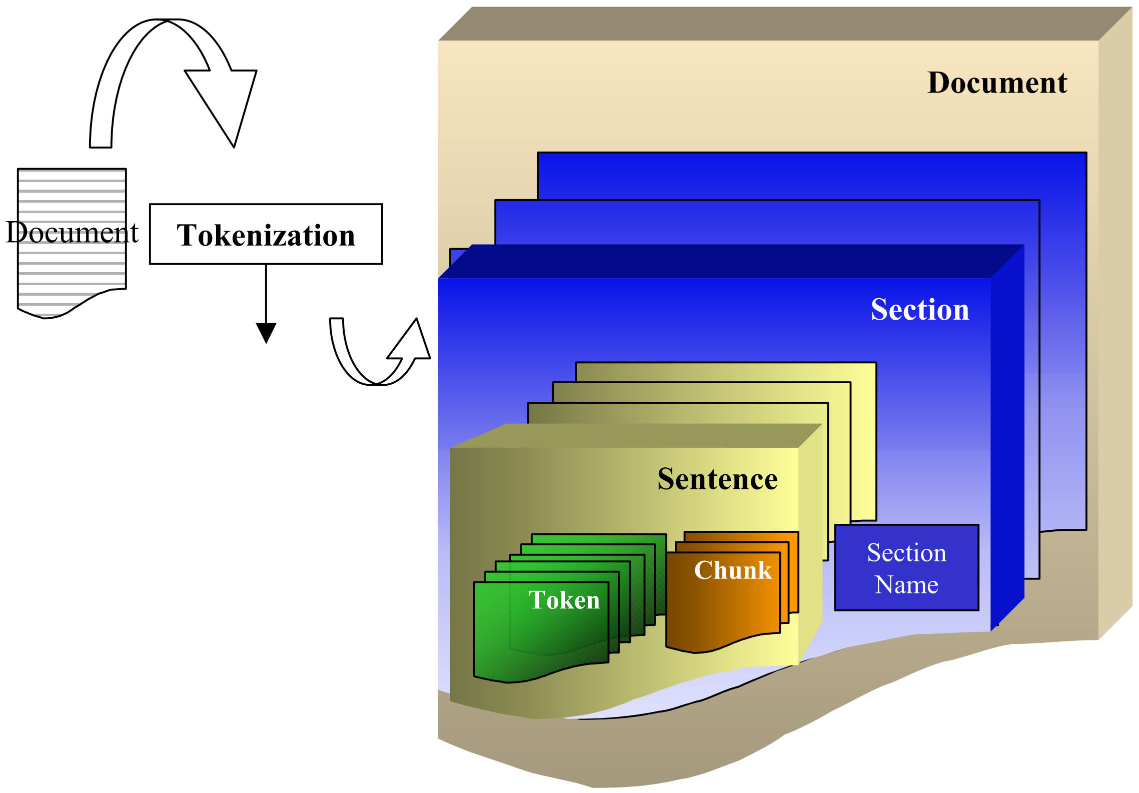
processString

processTerm

Container Classes

[filled in within the Tokenization Process]

Document,Section,Sentence …

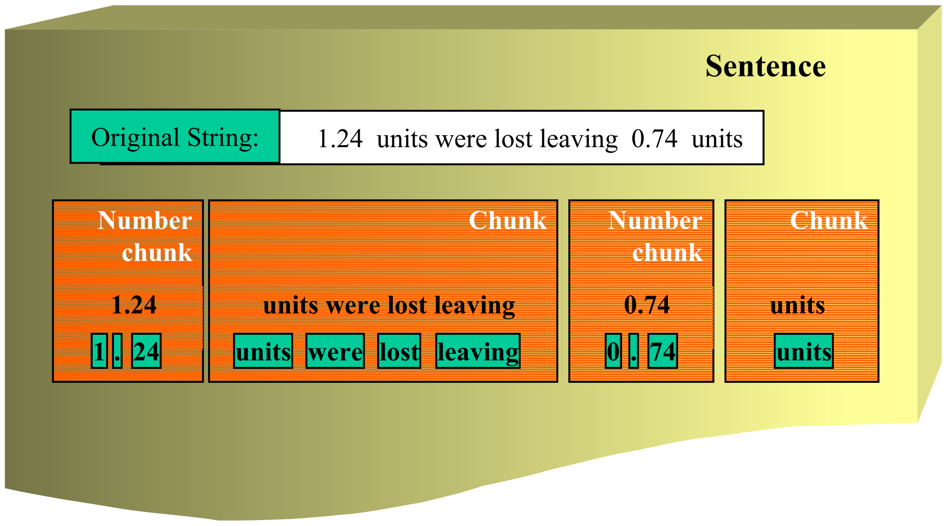


During the tokenization process, input text is transformed into an instance of a **Document** class. A Document is further tokenized into **sections**. When dealing with unstructured documents, sections are paragraphs. When dealing with MEDLINE Citations, each citation section, such as the Title section, the Author section, the abstract section equates to a separate section. The tokenizer considers a number of HTML tags including <p>, <tr>, <h1> with sections when dealing with HTML input. Sections can be labeled. For instance it might be useful to know that the section at hand is a title section. Each section contains a set of sentences. Each **Sentence** contains a number of containers but only the chunk and [word] **token** containers are filled out during this tokenization process.

Container Classes

[filled in within the Tokenization Process]

Sentence,Chunk,Token

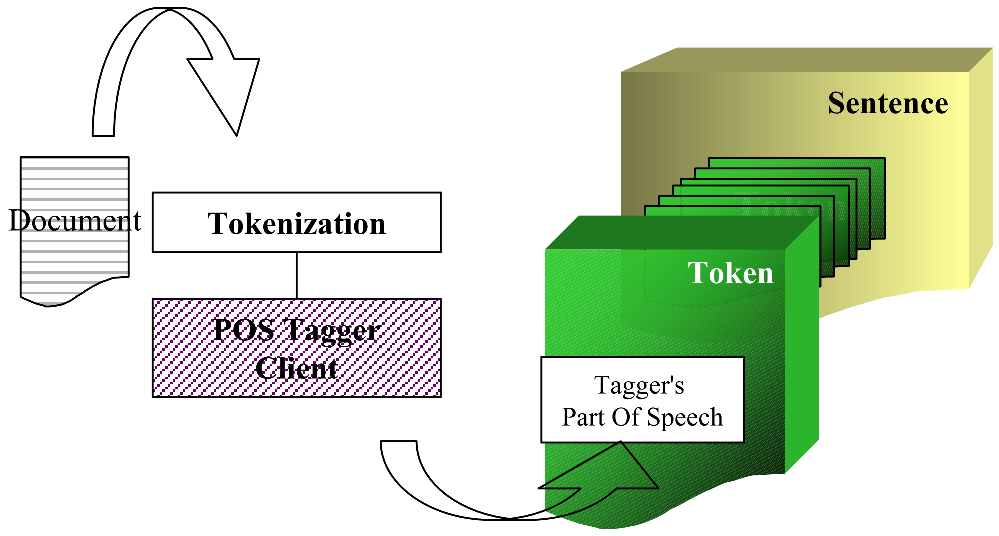


Each sentence is sent through a shape tokenizer to grab chunks of text that can be identified as a contiguous unit by some pattern recognition software. Chunks include dates, urls, emails, real numbers and similar patterns. Chunks are currently identified by a small set of compiled regular expressions. Those chunks that get identified are labeled. The text around the labeled chunks are also made into chunks, but without a label. Each chunk is then tokenized into word **tokens**. Those chunks that are labeled become **shapes** or lexicalElements with a Shape label during the lexical lookup process, and do not get otherwise looked up in the lexicon.

In the above example, the sentence is broken into 4 chunks, two labeled, and two unlabeled. The chunks shown contain the initial instances of the tokens associated with the sentence. It should be noted that a container referencing these tokens is created on each sentence. (Shown in the prior figure)

Chunks are an interim container that are no longer referenced after lexicalElements are made. The lexicalElements carry on the chunk labels.

Container Classes
[filled in within the Tagger Client Process]
Token



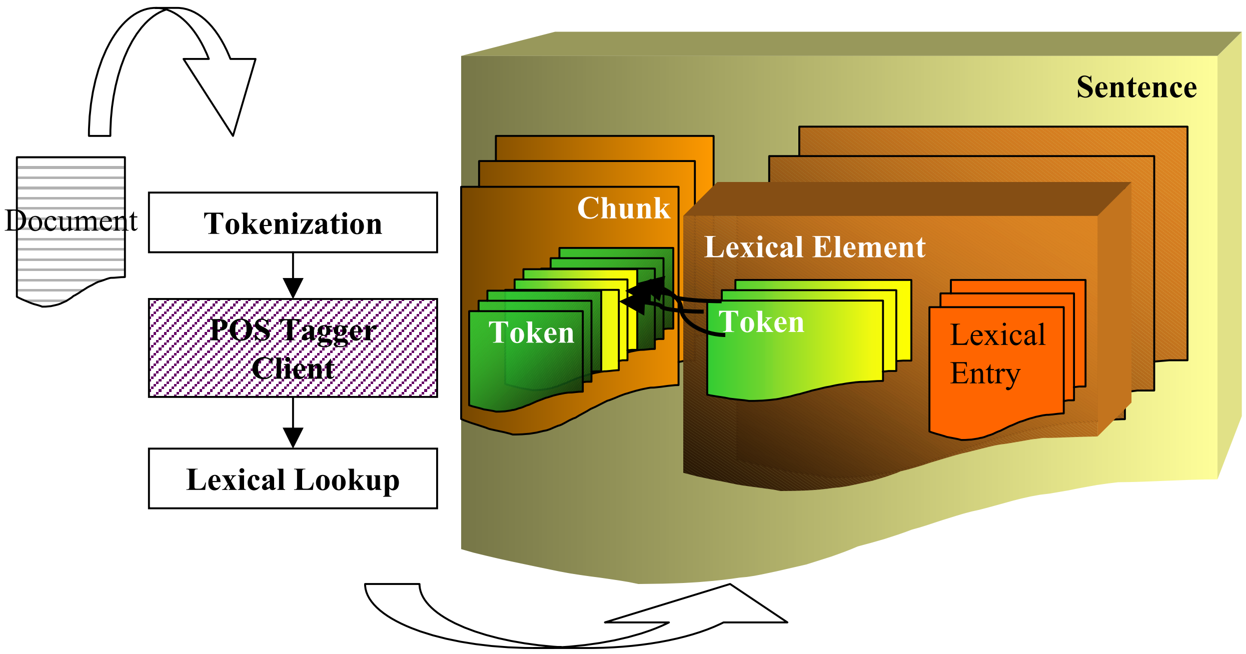
If a tagger client has been implemented and used*, the tagger client process will align the output from the tagger with each [word] token from each sentence, and deposit the tagger's part of speech on each token. This part of speech is useful during the noun phrase parsing process to disambiguate words that have multiple parts of speech.

* We currently only have one tagger client implemented for the an internal server around a Xerox Parc Part of Speech Tagger. We cannot distribute the server nor can we distribute the part of speech tagger. Therefore, this tagger client is turned off for external users. It is being provided as an example of how to implement other part of speech tagger clients.

Container Classes

[filled in within the Lexical Lookup Process]

LexicalElement, LexicalEntry

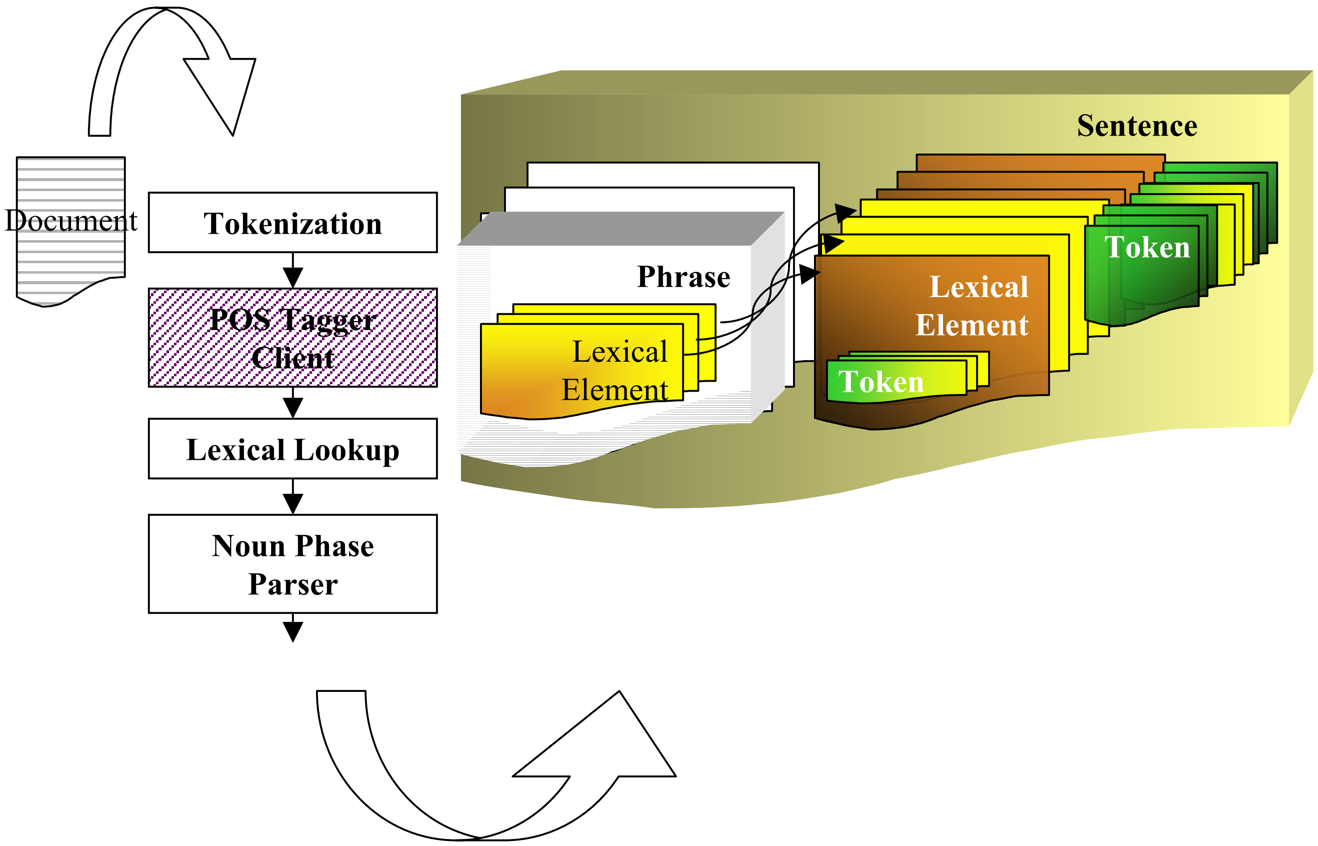


The Lexical Lookup process iterates through the chunks. The tokens from each labeled chunk become the tokens of a new LexicalElement. Lexical Elements that are created from labeled chunks include patterns include dates, numbers, and money. The tokens of the unlabeled chunks are used to identify and match to multi-word terms from the SPECIALIST Lexicon. Those matched terms are made into new instances of a Lexical Element which holds those tokens that make up the term. Lexical Elements are added to each sentence during this Lexical Lookup process. Lexical Elements that come from the Lexicon contain additional syntactic information from the Lexicon in a container holding LexicalEntries. Each LexicalEntry corresponds to an entry in the SPECIALIST Lexicon.. The tokens contained within each LexicalElement are not new instances, but are references to the tokens that were created during the tokenization process.

Container Classes

[filled in within the Noun Phrase Parser Process]

Phrase

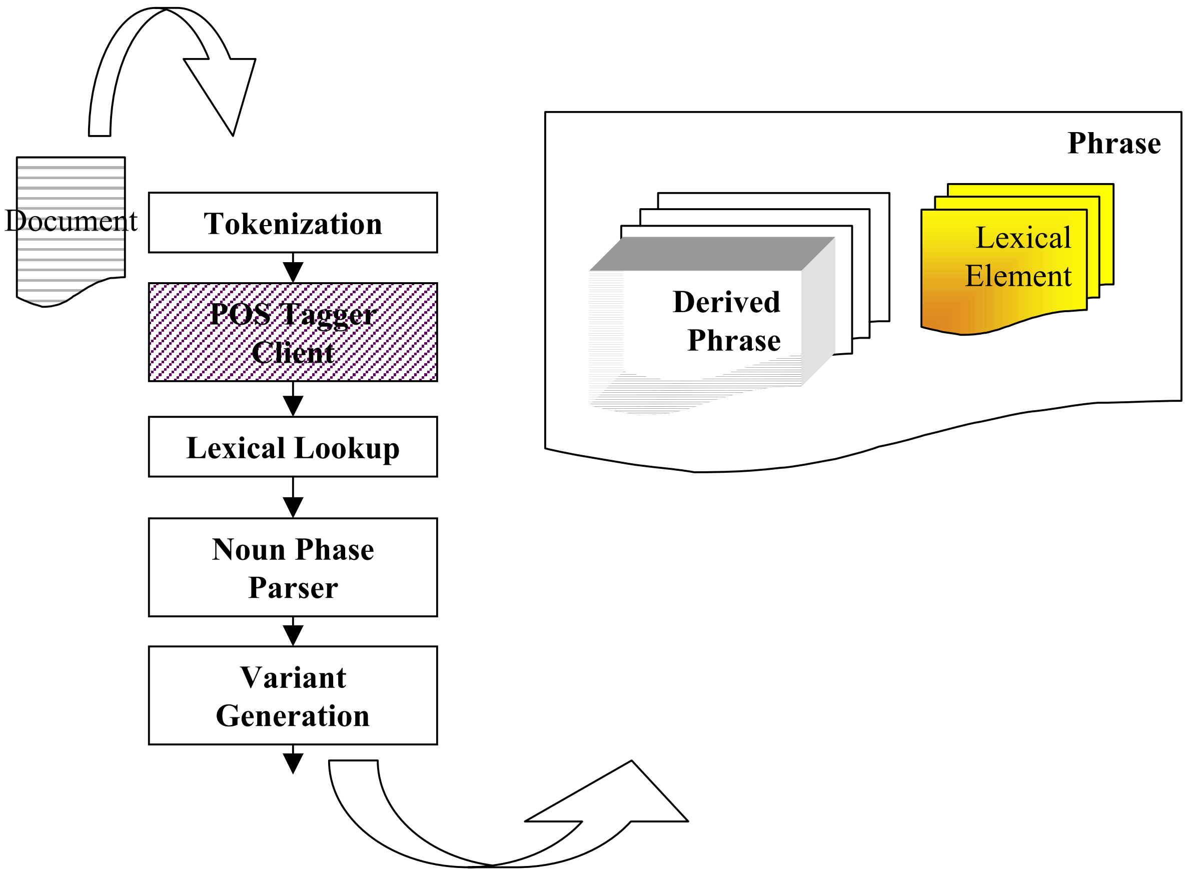


The Noun Phrase Parser Process combines lexical elements into a set of Phases. Each Phrase contains a set of references to those LexicalElements created during the Lexical Lookup process.

Container Classes

[filled in within the Variant Generation Process]

Derived Phrase

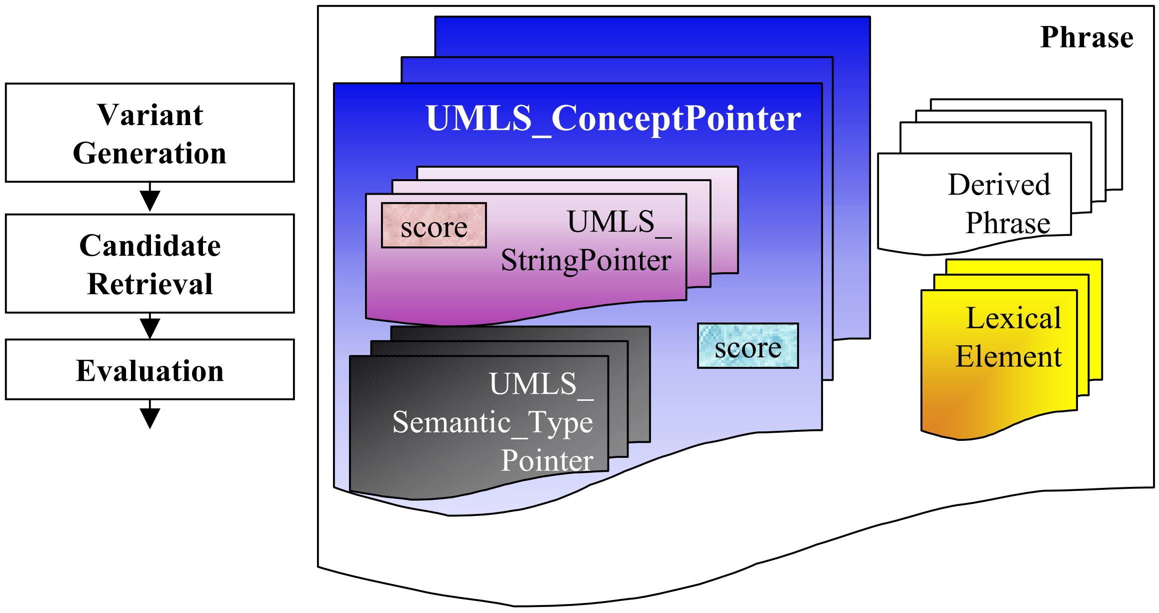


The Variant Generation process adds derived phrases to the phrase. Derived phrases are composed of permutations of variants of the lexical elements of the phrase. These variants include a recursive combination of spelling variants, synonyms, derivations, acronyms acronym expansions, and the inflections of each. The set of derived phrases are added each phrase. Derived phrases are internal to MMTx and it is not envisioned that they are useful toAPI developers. They are being covered here to show what the variant generation process does.

Container Classes

[filled in within the Candidate Retrieval/Evaluation Process]

UMLS_ConceptPointer,UMLS_StringPointer, UMLS_SemanticTypePointer



The phrase and derived phrases are used as the elements to match against indexes into the Metathesaurus during the candidate retrieval process. Those UMLS strings that come back are put into UMLS_StringPointers. The UMLS concept information is also gathered for these UMLS_Strings. The semantic type information is gathered for the UMLS concepts. The UMLS_Strings are subsequently put into UMLS_ConceptPointers. The set of UMS_ConceptPointers are added to the phrase.

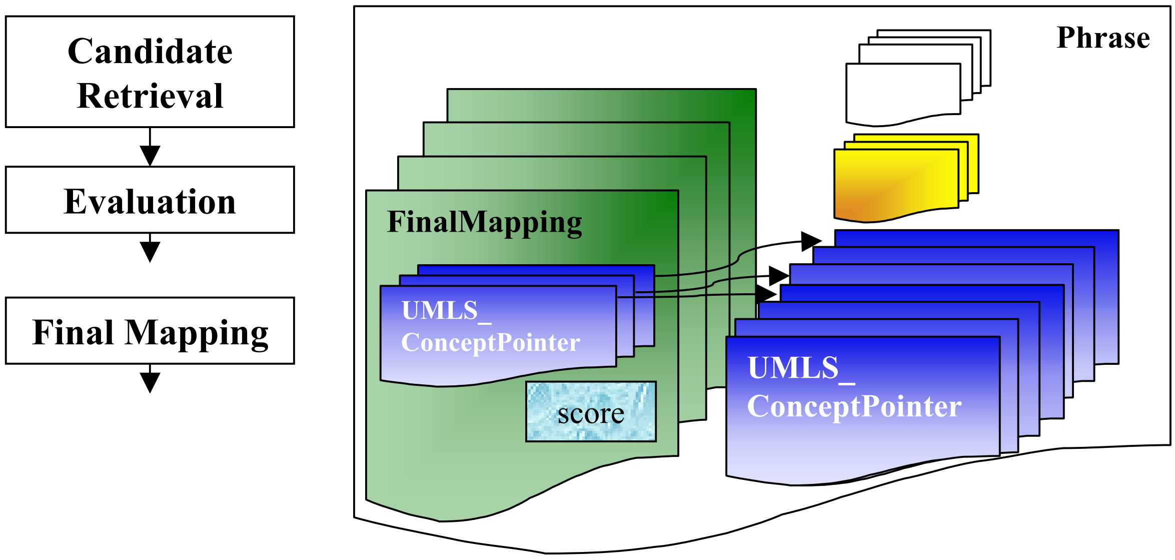
During the evaluation process, each UMLS_StringPointer is evaluated as to how well it matches the phrase. This score is added to each UMLS_StringPointer. The best of these string scores become the UMLS_ConceptPointer's score. The set of UMLS_ConceptPointers are sorted by score in descending order.

As an aside, UMLS_ConceptPointer and UMLS_StringPointers are labeled as such to make the distinction that these classes don't include all the data associated with UMLS Strings and Concepts and they also include scores that only make sense in the context of a phrase.

Container Classes

[filled in within the FinalMapping Process]

FinalMapping

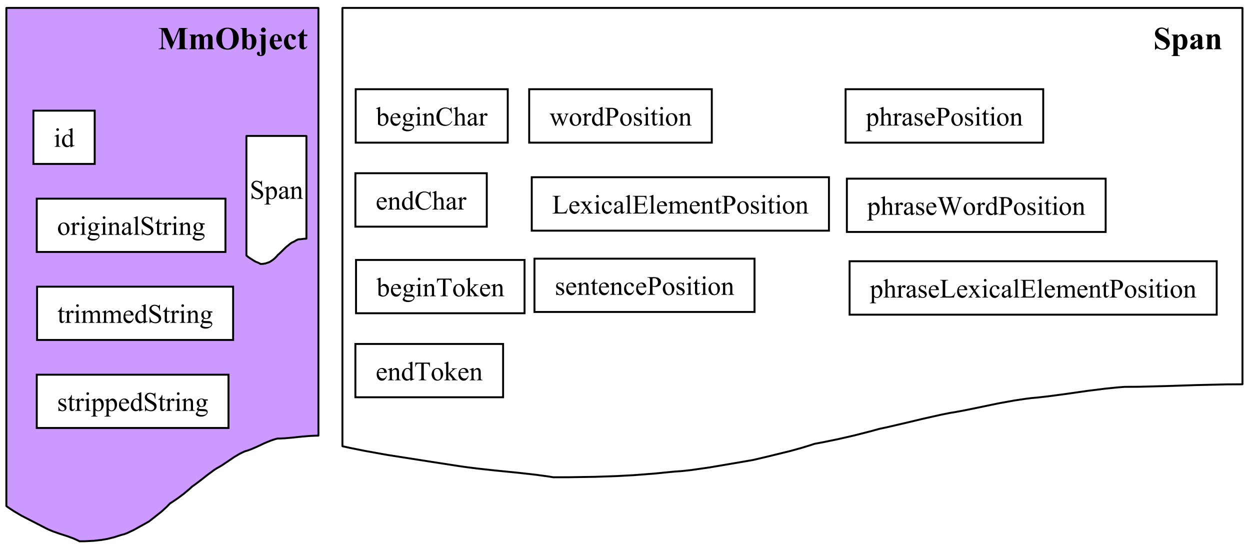


During the final mapping process, combinations of UMLS_ConceptPointers are grouped to best cover the the contents of the phrase. Each final mapping contains references to the UMLS_ConceptPointers that were created in the retrieval and evaluation process. Each final mapping is evaluated as to how well the combination of concepts cover the phrase. This score is added to each final mapping. The set of final mappings are added to each phrase.

There are applications where you want a set of closest matching concepts, not final mappings. In such cases, such as during term processing, the set of UMLS_ConceptPointers off the phrase is more appropriate. This set is an ordered set of concepts sorted in descending order by the evaluation score.

Two Well Used Classes

MmObject and Span

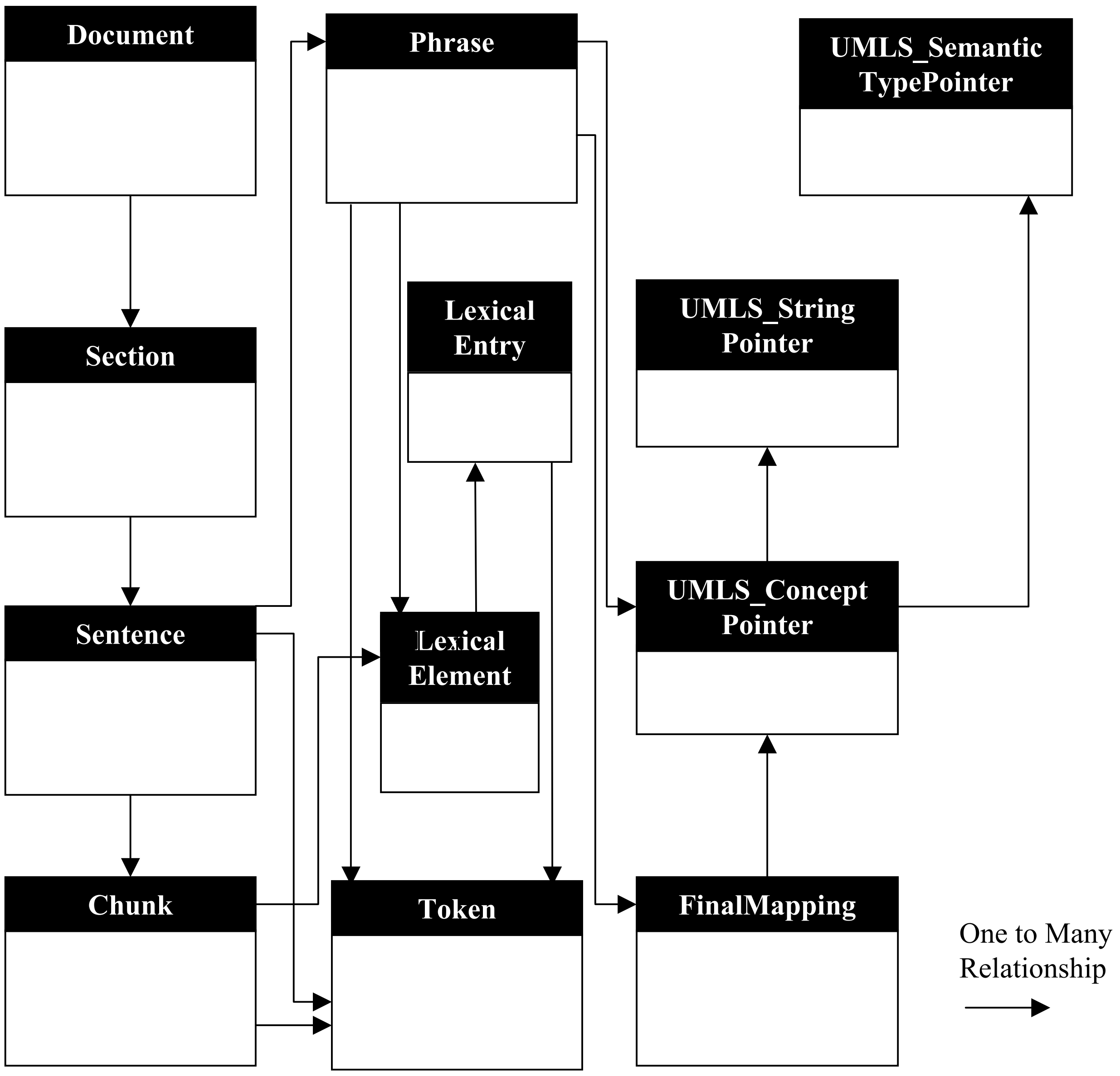


There are two well used classes within the NLP Tools and MMTx, MmObject and Span.

The MmObject class is inherited by just about every other textfeature class from Document on down to the tokens. Instances of MmObject all contain an Id, the original string, and a span. MmObjects also contain variants of the original string including a space trimmed string, and a punctuation stripped version of the string.

A span object contains the book keeping to tie this object back to the original document. A span contains character and word offsets that are relative to the document, and offsets that are relative to the phrase. It goes without saying that the phrase offsets are not set until phrases are created during the parsing phase, and the LexicalElementPositions are not set until the LexicalLookup phase.

Entity Relationship Diagram for the textfeature package



MmObject Class

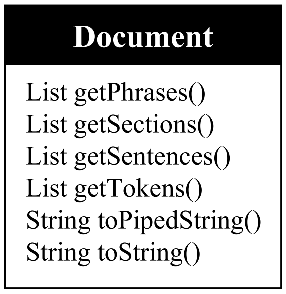
MmObject
void **appendOriginalString**(java.lang.String pContent)
int **getCharOffset**()
int **getId**()
String **getOriginalString**()
Span **getSpan**()
String **getStrippedString**()
String **getTrimmedString**()
void **setId**(int id)
void **setOriginalString**(java.lang.String origString)
void **setSpan**(int pBeginChar, int pEndChar)
void **setStrippedString**()
void **setTrimmedString**()
void **toString**()

Span Class

Span
int **getBeginCharacter**()
int **getBeginToken**()
int **getBeginWord**()
int **getEndCharacter**()
int **getEndToken**()
int **getEndWord**()
int **getLexicalElementPosition**()
int **getNumberOfCharacters**()
int **getPhraseLexicalElementPosition**()
int **getPhrasePosition**()
int **getPhraseWordPosition**()
int **getSentencePosition**()
int **getWordPosition**()
String **toString**()

Span (continued)
Span()
Span(int pStartChar, int pEndChar)
Span(Span pSpan)
void **setBeginCharacter**(int beginCharIndex)
void **setBeginToken**(int pPos)
void **setEndCharacter**(int endCharIndex)
void **setEndToken**(int pPos)
void **setLexicalElementPosition**(int pPos)
void **setPhraseLexicalElementPosition**(int pPos)
void **setPhrasePosition**(int pPos)
void **setPhraseWordPosition**(int pPos)
void **setSentencePosition**(int pPos)
void **setSpan**(int pBeginChar, int pEndChar)
void **setTokenPosition**(int pStartPos, int pEndPos)
void **setWordPosition**(int pPos)
void **setWordSpan**(int pBeginWord, int pEndWord)

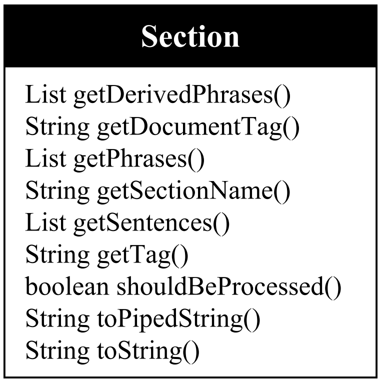
Document Class



Document

List getPhrases()
List getSections()
List getSentences()
List getTokens()
String toPipedString()
String toString()

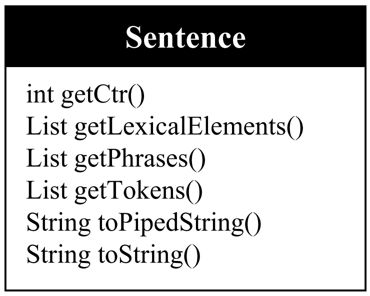
Section Class



Section

List getDerivedPhrases()
String getDocumentTag()
List getPhrases()
String getSectionName()
List getSentences()
String getTag()
boolean shouldBeProcessed()
String toPipedString()
String toString()

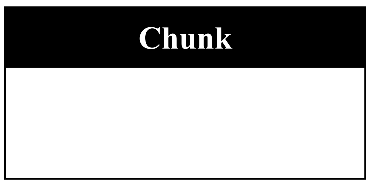
Sentence Class



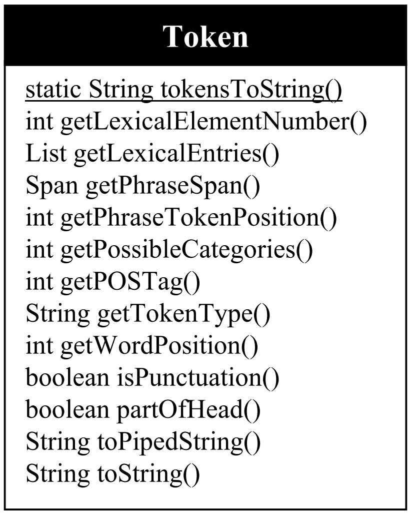
Sentence

```
int getCtr()
List getLexicalElements()
List getPhrases()
List getTokens()
String toPipedString()
String toString()
```

Chunk Class



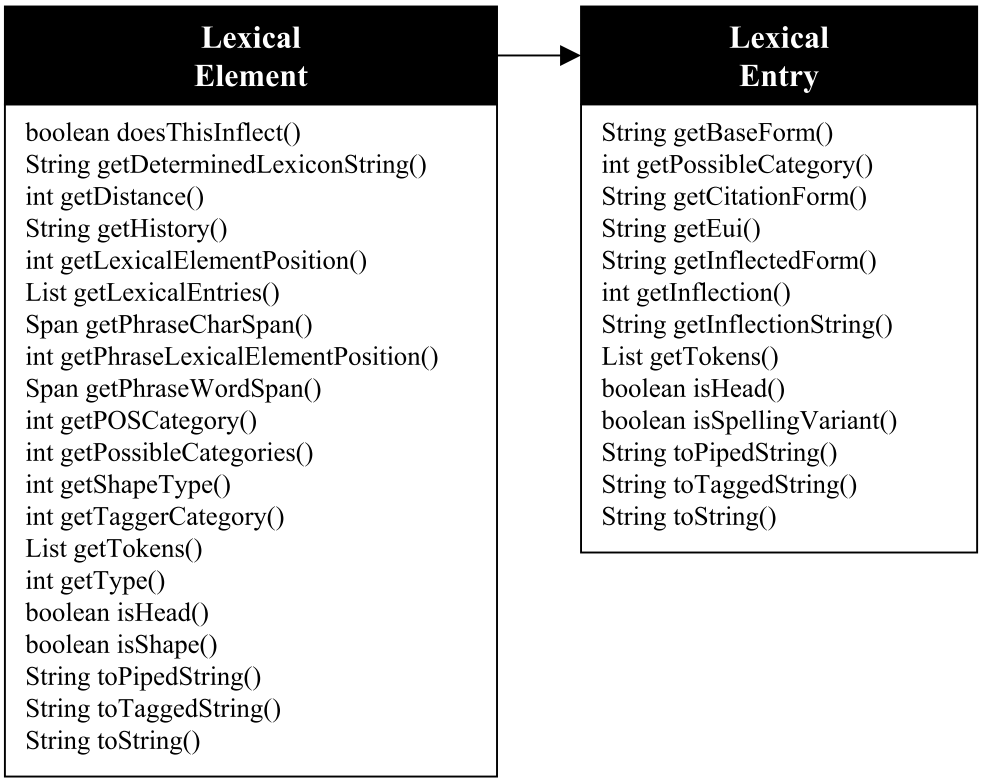
Token Class



Token

static String tokensToString()
int getLexicalElementNumber()
List getLexicalEntries()
Span getPhraseSpan()
int getPhraseTokenPosition()
int getPossibleCategories()
int getPOSTag()
String getTokenType()
int getWordPosition()
boolean isPunctuation()
boolean partOfHead()
String toPipedString()
String toString()

Lexical Element Class
LexicalEntry Class



Lexical Element

```
boolean doesThisInflect()
String getDeterminedLexiconString()
int getDistance()
String getHistory()
int getLexicalElementPosition()
List getLexicalEntries()
Span getPhraseCharSpan()
int getPhraseLexicalElementPosition()
Span getPhraseWordSpan()
int getPOSCategory()
int getPossibleCategories()
int getShapeType()
int getTaggerCategory()
List getTokens()
int getType()
boolean isHead()
boolean isShape()
String toPipedString()
String toTaggedString()
String toString()
```

Lexical Entry

```
String getBaseForm()
int getPossibleCategory()
String getCitationForm()
String getEui()
String getInflectedForm()
int getInflection()
String getInflectionString()
List getTokens()
boolean isHead()
boolean isSpellingVariant()
String toPipedString()
String toTaggedString()
String toString()
```

One to Many
Relationship

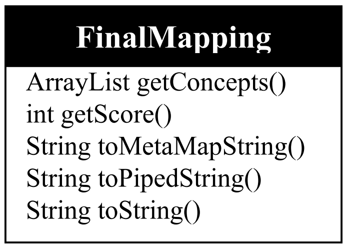
$\longrightarrow$

Phrase Class

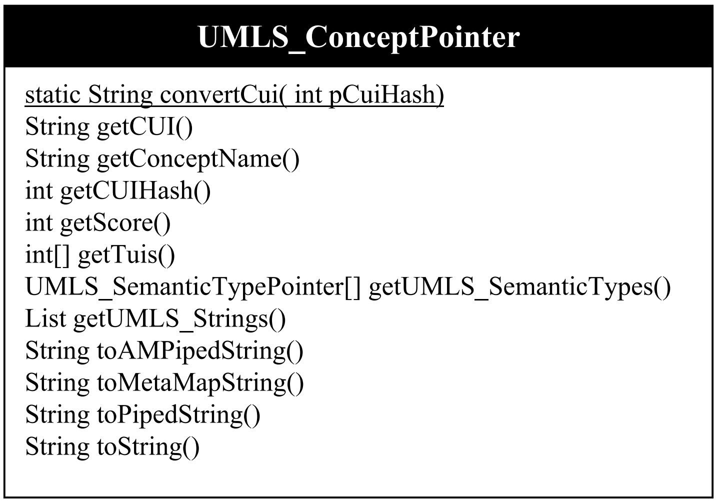
Phrase
String displayTags()
String displayVariants()
List getAllVariants()
UMLS_ConceptPointer getConceptPointer()
UMLS_ConceptPointer[] getConcepts()
List getDerivedPhrases()
ArrayList getFinalMappings()
List getLexicalElements()
List getNp()
String getNpString()
List getNpTokens()
String getOriginalString()

Phrase (cont.)
int getPhrasePosition()
int getSizeOfPhrase()
String getTrimmedString()
boolean isOfPhrase()
boolean isPrepPhrase()
String toMincoManString()
String toMoString()
String toPipedString()
String toString()
String toSyntaxString()

FinalMapping Class



UMLS_ConceptPointer Class



UMLS_ConceptPointer

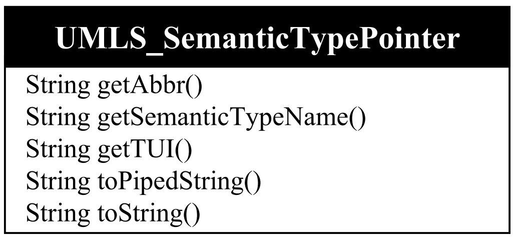
```
static String convertCui( int pCuiHash)
String getCUI()
String getConceptName()
int getCUIHash()
int getScore()
int[] getTuis()
UMLS_SemanticTypePointer[] getUMLS_SemanticTypes()
List getUMLS_Strings()
String toAMPipedString()
String toMetaMapString()
String toPipedString()
String toString()
```

UMLS_StringPointer Class

UMLS_StringPointer

static String convertSui(int pSuiHash)
int[] getCuis
String getName()
String getNormalizedString()
String getSUI()
int getSUIHash()
int getScore()
String toPipedString()
String toString()

UMLS_SemanticTypePointer Class



MMTxAPI Example

This example calls MMTx.processDocument() in the same way the simple example did.

This example retrieves all the sentences and iterates through them rather than retrieving the the phrases directly.

```
…

// ==========================+ Analyze the file +==
Document aDocument = MMTx.processDocument(aFile);

List              sentences = aDocument.getSentences() ;
String               sentence = null;
int         numberOfSentences = sentences.size();
String         sentenceString = null;

// ====================+ Print the Sentences out +==
for ( int i = 0; i < numberOfSentences; i++ ) {
  aSentence = (Sentence) sentences.get(i);
  sentenceString = displaySentence( aSentence );
  System.out.println( sentenceString);

} // End of Loop through sentences
```

MMTxAPI Example
displaySentence()

DisplaySentence() retrieves the sentence's piped string, grabs all the phrases and retrieves each of the phrases display information.

```
String displaySentence( Sentence pSentence ) throws Exception {

    StringBuffer buff = new StringBuffer();
    buff.append( pSentence.toPipedString() );
     //Sentence|id|start|end|TrimmedString|
    buff.append( U.NL );

    List      phrases = pSentence.getPhrases() ;
    int numberOfPhrases = phrases.size();
    Phrase      aPhrase = null;
    String phraseString = null;
    for ( int i = 0; i < numberOfPhrases; i++ ) {
      aPhrase = (Phrase) phrases.get(i);
      phraseString = displayPhrase( aPhrase);
      buff.append( phraseString );
     } // End of Loop through phrases

    return ( buff.toString() );
 } // *** End displaySentence()
```

MMTxAPI Example
displayPhrase()

DisplayPhrase() retrieves the phrases's piped string, grabs mappings and retrieves each of the mapping's display information. We could call the aMapping. toMetaMapString() here, but that we wrote our own displayMapping() here to demonstrate that we can control the output.

```
String displayPhrase( Phrase pPhrase ) throws Exception {

    StringBuffer buff = new StringBuffer();
    buff.append( pPhrase.toPipedString() );
    buff.append( U.NL );

    ArrayList finalMappings = pPhrase.getFinalMappings() ;
    FinalMapping   aMapping = null;
    String     mappingString = null;
    int     numberOfFinalMappings = finalMappings.size();

    for ( int i = 0; i < numberOfFinalMappings; i++ ) {
      aMapping = (FinalMapping) finalMappings.get(i);
      mappingString = displayMapping( aMapping);
      buff.append( mappingString );
     } // End of Loop through Mappings

    return ( buff.toString() );
 } // *** End displayPhrase()
```

MMTxAPI Example
displayMapping()

DisplayMapping() retrieves the mapping's concepts and retrieves each of the concept's display information.

```
String displayMapping( FinalMapping pMapping ) throws Exception {

    StringBuffer buff = new StringBuffer();

    ArrayList            concepts = pMapping.getConcepts() ;
    UMLS_ConceptPointer aConcept = null;
    String           umls_Strings = null;
    int         numberOfConcepts = concepts.size();

    for ( int i = 0; i < numberOfConcepts; i++ ) {
      aConcept = (UMLS_ConceptPointer) concepts.get(i);
      buff.append( aConcept.toPipedString() );
      umls_Strings = displayUMLS_Strings( aConcept);
      buff.append( umls_Strings );
     } // End of Loop through Concepts

    return ( buff.toString() );
 } // *** End displayMapping()
```

MMTxAPI Example
displayUMLS_Strings()

DisplayUMLS_Strings() retrieves the concept's UMLS strings (Alternatively, the interfaces of UMLS_Concept Pointer and UMLS_StringPointer provide other formatting options.)

```
String displayUMLS_Strings( UMLS_ConceptPointer pConcept ) throws
     Exception {

    StringBuffer buff = new StringBuffer();

    ArrayList      umlsStrings = pConcept.getUMLS_Strings() ;
    UMLS_StringPointer aString = null;
    String          stringName = null;
    String                 sui = null;
    int            stringScore = null;
    int        numberOfUMLSStrings = umlsStrings.size();

    for ( int i = 0; i < numberOfUMLSStrings; i++ ) {
      aString     = (UMLS_StringPointer) umlsStrings.get(i);
       stringName  = aString.getName();
      sui         = aString.getSUI();
      stringScore = aString.getScore();
      buff.append( sui + "|" + stringName + "|" + stringScore + U.NL );

     } // End of Loop through Strings
     return ( buff.toString() );
 } // *** End displayUMLS_Strings()
```

Adding Shape Identifiers
[during the Tokenization Process]

Additional labeled chunks can be added to the tokenizer by adding additional regular expressions to the shapeTokenizer. A strong caution should be given though to note that pattern recognition via regular expressions become extremely slow as the number of regular expressions increases.

The shapeTokenizer could be altered to plug in additional black boxes that create additional labeled chunks.You might want to do this to identify chemicals, drugs, gene names, or proper names.

This is done by adding additional methods to the ShapeTokenizer's *public Vector shapeTokenize(Vector pChunks)* method. Each additional method should take as an input a Vector of Chunk. The method should split an existing chunk into a new labeled chunk if one is found, new unlabeled chunks of the text surrounding the labeled chunk, and otherwise leave unaltered chunks that it does not find a pattern in. The method should alter the Vector of Chunks to include any of the new chunks created in the appropriate place within the Vector. The method should not tokenize into words the newly created chunks, as the word tokenization happens after all chunks are identified.

It is the current philosophy that once a chunk gets labeled, no other identifier should bother looking at it. This is being done solely for simplification purposes. We do not know how to handle overlapping chunks, or nested chunks. As a consequence, the order of pattern matchers employed matters, with the first ones employed being the first to segment and label the text to the exclusion of later shape identifiers. It is realized that this may change down the road, for instance, to be able to combine chunks into a chunk like recognizing that a number chunk and a unit of measure chunk could be combined into one unit of measure chunk.

Adding Shape Identifiers

[during the Tokenization Process]

Each Chunk should be created with the following information:

The string that makes up the chunk, a label from the Chunk class, and the character span relative to the sentence. The first two are parameters passed in to the Chunk class's constructor. The setSpan() method is needed to set the character span of this chunk within the sentence.

[Example chunk here]

MMTx Settable Options

The MMTxAPI has the same settable options that MMTx has. These options are set via the MMTx command line arguments as well as settings from the $MMTx/config/MMTxRegistry.cfg file.

Any entry in the MMTxRegistry file could be a command line argument to MMTx. This next section will elucidate the elements within the MMTxRegistry file.

References

This page is a placeholder for UMLS Metathesaurus semantic network, and MetaMap references. It is also a placeholder for links to useful additional information.

- umlsLex.nlm.nih.gov
  - SPECIALIST NLP Tools
- MMTx.nlm.nih.gov *

* The MMTx website and documentation is public. The download requires a UMLS license.

Appendix

MMTxAPIExample1.java
MMTxAPIExample2.java
example.txt