Conceptual model overview

Ken Zook

November 22, 2008

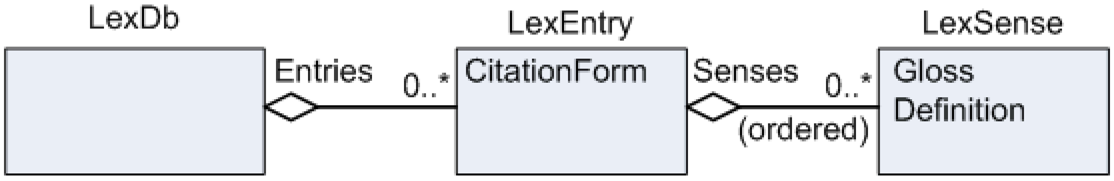
Contents

1 Introduction

FieldWorks uses an object-oriented conceptual model that defines the structure for every kind of data it stores such as data notebook, lexical database, Scripture, and interlinear text. This model consists of a hierarchy of classes and properties on those classes. Properties may hold one or more instances of other classes, or it may hold a basic object such as a string or an integer. Properties may also reference instances of classes. We use Unified Modeling Language (UML) diagrams.

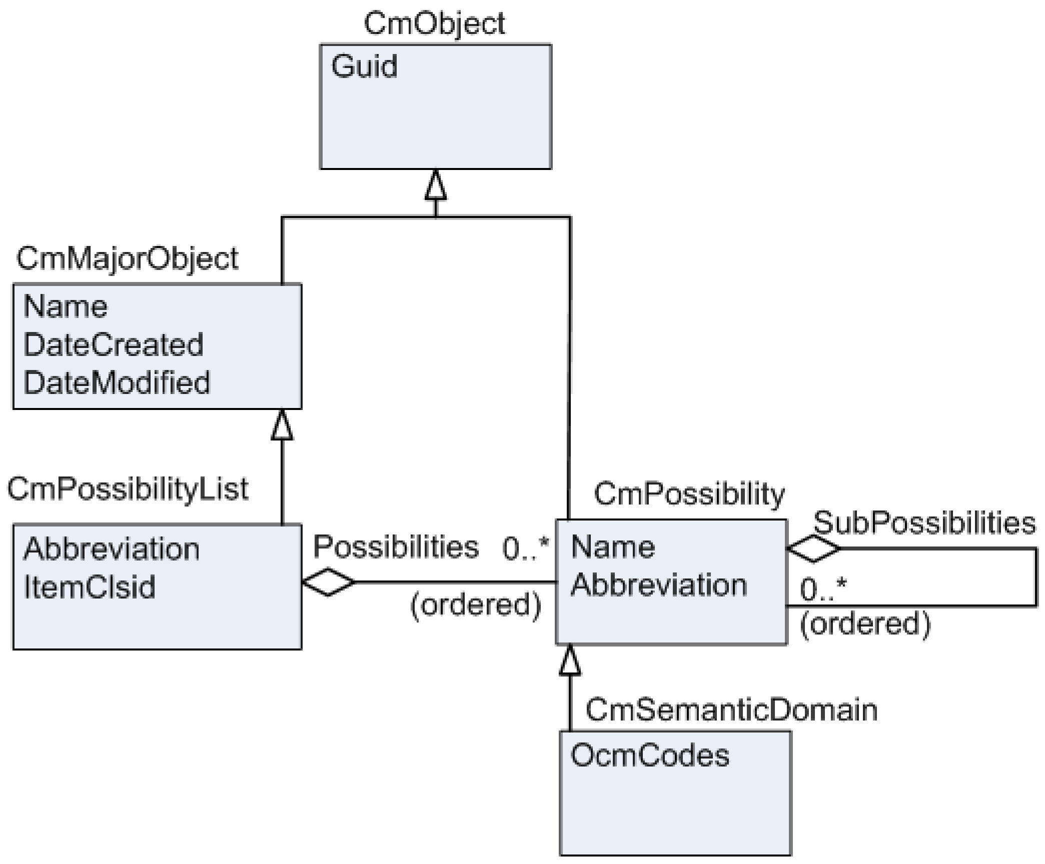
Note: In order to support the possibility of using the Firebird database engine in addition to Micrsoft SQL Server, and due to limited length of names in Firebird, some class and property names were shortened in FieldWorks 5.4 compared to earlier versions. The spreadsheet, Model name changes.xls, lists the changes that were made.

1.1 Basic ownership



LexDb is a class that has an Entries property that owns a collection of LexEntry. The diamond indicates an owning relationship. An object usually has a single owner, but a few do not have owners. An object can never be owned by more than one object. “0..*” indicates this property can hold any number of objects. “0..1” or just “1” indicates the property can only own a single object (e.g., it’s an atomic property). A sequence property has inherent order while a collection has no inherent order. Either one can be sorted when displayed. On a diagram a sequence property has an “(ordered)” label. LexEntry has a CitationForm property that holds a string. It also has a Senses property that owns a sequence of LexSense. LexSense has a Gloss property holding a string and a Definition property holding a string.

1.2 Inheritance



Classes can inherit properties from other classes. The root class for inheritance is CmObject. It is an *abstract class* since there are no instances of this class. Instead, there are instances of *concrete subclasses* of CmObject. Current diagrams do not indicate whether a class is abstract or concrete. Go to the class definition for that information. CmObject has a number of properties such as a Globally Unique Identifier (Guid) which is a computer-generated identifier that is guaranteed to be unique in the entire world.

CmMajorObject inherits from CmObject. Inheritance is indicated by a line with an open arrow pointing to the superclass. CmMajorObject is also an abstract class. It contains properties for Name, DateCreated, and DateModified and also inherits all the properties of superclasses. Thus, it also has a Guid property that is inherited from CmObject.

CmPossibilityList is a concrete class that inherits all of the properties from CmMajorObject as well as from CmObject. In addition, it has Abbreviation and ItemClsid properties and a Possibilities property that owns a sequence of CmPossibility.

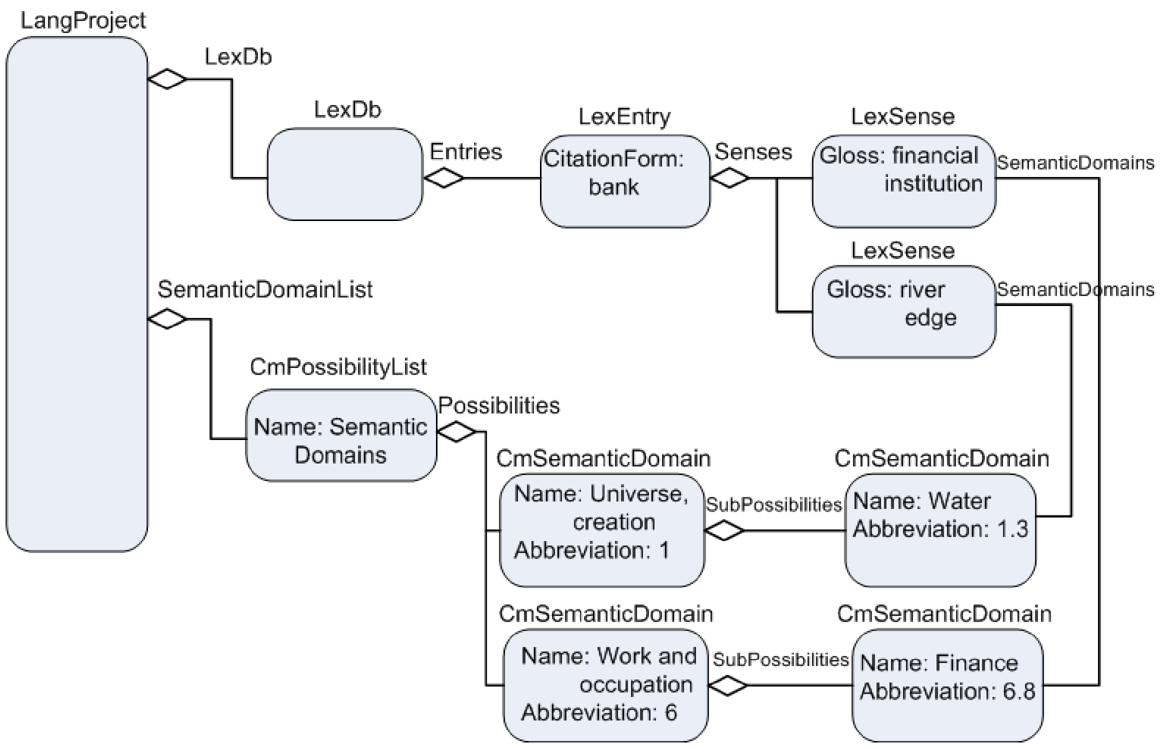
CmPossibility is a concrete class that inherits from CmObject. This is typically not shown on diagrams, since everything that does not inherit directly from some other class *must* inherit from CmObject. CmPossibility has Name and Abbreviation properties in addition to a SubPossibilities property that owns a sequence of CmPossibility. Rather than adding multiple fields for translations of list items in each entry or sense as done by MDF, FieldWorks stores the translations one time for each item in the list. The multiUnicode Name and Abbreviation properties allow users to add any number of languages and/or writing systems to the list. The default views show the top analysis writing system for most list references, but in the configurable dictionary view and in export, FieldWorks can show more than one writing system for list item references.

Numerous lists in FieldWorks use CmPossibilities or their subclasses. One is Ron Moe's semantic domain list. CmSemanticDomain is a subclass of CmPossibility, so it inherits Name and Abbreviation from this class plus Guid from CmObject. It adds OcmCodes, a string that lists the OCM codes that relate to this semantic domain. Other lists are Parts of Speech, OCM Codes, and Entry Types.

All items in a list are owned either directly by a CmPossibilityList through the Possibilities property or by another item in the list through the SubPossibilities property. The ItemClsid property is used by the program to know what kind of class to add to this list.

1.3 Reference relationship

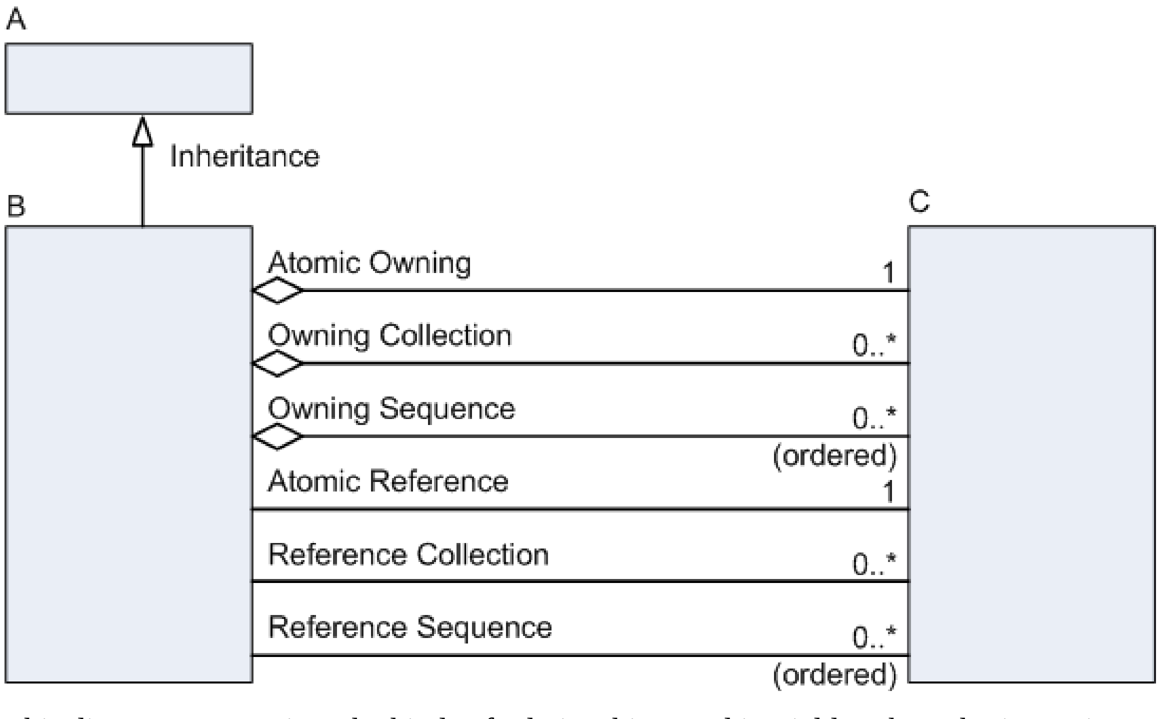
This picture puts some of this together. Rounded corners indicate an instance diagram rather than a class diagram. It can show how actual objects are interconnected in the database.



The owner of most things in a FieldWorks project is the LangProject which does not have an owner. It owns a single LexDb in the LexDb property and a single CmPossibilityList in the SemanticDomainList property. One of the entries owned in the Entries property of the LexDb is a LexEntry with a CitationForm of 'bank'. That LexEntry owns two senses in its Senses property. The first LexSense has a Gloss of 'financial institution' and the second LexSense has a Gloss of 'river edge'. The Semantic Domains possibility list owns two CmSemanticDomains in its Possibilities property: a Name "Universe, creation" with an abbreviation of “1” and a Name “Work and occupation” with an Abbreviation of “6”. The first semantic domain owns another CmSemanticDomain through its SubPossibilities property. This semantic domain has a Name “Water” and Abbreviation “1.3”. The second semantic domain also owns another CmSemanticDomain through its SubPossibilities property. This semantic domain has a Name “Finance” and an Abbreviation “6.8”. Several more layers of semantic domains are in the outline but are not included in the Language project illustration.

Users may want the “financial institution” sense to be related to the *Finance* semantic domain. Both objects are already owned, so they cannot use an owning relationship. They may have many senses related to the Finance semantic domain and do not want to have to type this string over and over. The solution is a reference property called *SemanticDomains* which essentially points from the sense to the semantic domain. The diagrams show a reference as a line without the diamond. The reference property name is close to the class that defines that relationship. Like owning relationships, reference relationships can also be atomic, a collection (unordered), or a sequence (ordered). In the diagram, the “river edge” sense points to the “water” semantic domain through the SemanticDomains reference property.

1.4 Summary of relationships



This diagram summarizes the kinds of relationships used in Fieldworks. Inheritance is a single arrow pointing to the superclass. In this case, A is the superclass and B is the subclass. B inherits all properties from A. Owning relationships are lines with a diamond and name by the class that defines this relationship. Reference relationships are lines without arrows with a name by the class that defines this relationship. Owning and Reference relationships can be

- atomic, where the property can hold 0 or 1 object

- collection, where the property can hold any number of unordered objects, or
- sequence, where the property can hold any number of ordered objects.

Owning and Reference relationships always define the class of object they can own or refer to—in this case C. This means the property can hold instances of C or any subclasses of C, but no other classes.

When a relationship property is defined in FieldWorks, the designer must choose one of the following types. The object type code shown in parentheses is stored in the Field$ table in the database.

- OwningAtom (23)
- ReferenceAtom (24)
- OwningCollection (25)
- ReferenceCollection (26)
- OwningSequence (27)
- ReferenceSequence (28)

1.5 Basic properties

1.5.1 Strings

FieldWorks uses two basic types of strings: Unicode and String. Unfortunately, these are poorly named since both types are strings and both types store Unicode data, but changing the names now would be very involved. Better names would be FieldWorks Unicode strings and FieldWorks Strings.

A FieldWorks Unicode string is a plain sequence of Unicode code points. There is no indication of a font or writing system for a Unicode string. Unicode strings cannot have any embedded writing systems, formatting, or hot links.

A FieldWorks String is a sequence of Unicode code points. Along with the code points it maintains additional information on each code point that always includes the writing system it represents and optionally

- a format style to use for display purposes
- hard-coded formatting including font, face, point size, and color
- embedded objects such as hot links to external files or FieldWorks objects, and
- overlay tags which are links to possibility items.

A FieldWorks String is a rich Unicode string with a writing system and may hold many embedded elements. A FieldWorks Unicode string is just a raw Unicode string with *no* additional information.

In the FieldWorks conceptual model, properties (fields) can hold a

- single FieldWorks Unicode string
- collection of FieldWorks Unicode strings
- single FieldWorks String, or
- collection of FieldWorks Strings.

The collections are called MultiUnicode and MultiString. Each string in the collection has a unique corresponding writing system. For example, a MultiString can have one

FieldWorks String for French, another for English, and yet another for Spanish. It *can*not have two FieldWorks Strings for Spanish. Each FieldWorks String can include information such as nested writing systems and formatting. MultiUnicode works the same, except it cannot have nested writing systems or formatting since the strings in a MultiUnicode property are FieldWorks Unicode strings.

In the database, FieldWorks provides for short strings limited to 4,000 characters or long strings that can hold a billion characters. In the database, some common string operations such as sorting and searching work on short strings but not on long strings. FieldWorks refers to the long versions as BigUnicode, BigString, MultiBigUnicode, and MultiBigString.

When a string property is defined in FieldWorks, the designer must choose one of the following types. The object type code shown in parentheses is stored in the Field$ table in the database.

- String (13)
- MultiString (14)
- Unicode (15)
- MultiUnicode (16)
- BigString (17)
- MultiBigString (18)
- BigUnicode (19)
- MultiBigUnicode (20) (unused)

Another type of property appears to the user as a multiparagraph basic property. This actually uses an owning atomic property holding an StText, but appears to be a field that works very similar to a Word document. You can press Enter to start a new paragraph and format paragraphs and characters similar to Word.

1.5.2 Other basic properties

The FieldWorks conceptual model provides for three types of numbers: Integer, Numeric, and Float. At this point, only integers are used. The designer can specify that integers have an optional minimum and maximum value. By default, an integer can be -2,147,483,648 through 2,147,483,647 (4 bytes). If min/max are used to limit the range to -32,768 through 32,767, the database will use a *smallint* (2 bytes). If min/max are used to limit the range to 0 through 255, the database will use a *tinyint* (1 byte). In some cases, an integer can represent an enumeration, although this is not enforced by the database.

FieldWorks also provides the following:

- A Boolean type that is stored in the database as a single bit
- A Guid type that maps to a uniqueidentifier (16 bytes) in the database
  A typical string form for a Guid is 6F9619FF-8B86-D011-B42D-00C04FC964FF.
- Two types for binary data: Image (up to 2,147,483,647 bytes) and Binary (up to 8,000 bytes)
  At this point Image is not used (except as part of string formatting). Binary is implemented in the database as *varbinary (8000).*

- Two types of dates: Time and GenDate
  - Time uses a *datetime* type in the database that can store a date and time from January 1, 1753 through December 31, 9999 with an accuracy of three-hundredths of a second or 3.33 milliseconds.
  - A GenDate is stored as an integer (4 bytes). It can be used to store dates such as “unknown”, “January 3, 2455”, “before 1852”, and “around February, 1492”. It is a decimal representation of a generic date without a time that can range from 21474 BC through 21474 AD. The format is [-]YYYYMMDDP[1]

The designer must choose one of the following types for non-string basic properties. The object type code shown in parentheses is stored in the Field$ table in the database.

- Boolean (1)
- Integer (2) (with optional minimum and maximum values)
- Numeric (3) (unused)
- Float (4) (unused)
- Time (5)
- Guid (6)
- Image (7) (unused)
- GenDate (8)
- Binary (9)

2 Conceptual model documentation

The main documentation for the FieldWorks conceptual model is in c:\Program Files\SIL\FieldWorks\Helps\ModelDocumentation.chm. The upper left pane allows users to choose between Classes, Diagrams, or Dictionary.

2.1 Classes

Click *Classes* in the upper left pane and the lower left pane gives a list of all of the FieldWorks conceptual model classes. Click one of these and the right pane shows detail for that class. The detail includes

- basic class information
- basic attributes (properties) defined on that class (not ones inherited from superclasses)
- owning and reference attributes (properties) defined on that class, and
- back references (other classes that own or refer to this class).

[1] YYYY is the 1–5 digit year (negative is BC). 0000 is unknown.
MM is the 2 digit month (for BC months it is 13 - M). 00 is unknown.
DD is the 2 digit day (for BC days it is 32 - D). 00 is unknown.
P is one of the following:
    0 = Before (If GenDate = 0, it means nothing is entered)
    1 = Exact
    2 = Approximate
    3 = After

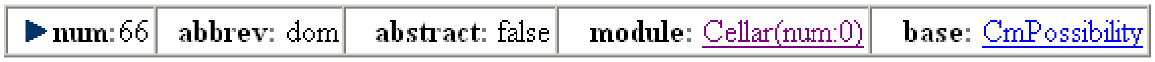
Many properties have a triangle at the left that users can click to toggle the display of documentation for that property. They can also use buttons at the top to turn on or off all documentation for this class. The illustrations shown here come from the CmSemanticDomain class.

CmObject contains built-in information that does not show up in this help file.

The information shown in the class view is generated directly from a Unified Modeling Language (UML) XML file that defines all the classes used in FieldWorks (other than CmObject). This file is maintained using the MagicDraw program. This XML file is transformed in various ways

- to provide this documentation file
- to provide code for generating the tables and fields in the database, and
- to generate basic source code for accessing the data via programming languages.

2.1.1 Basic class information



▶ **num**: 66	**abbrev**: dom	**abstract**: false	**module**: Cellar(num:0)	**base**: CmPossibility

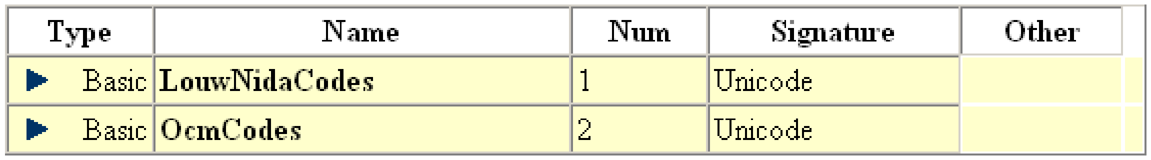
Classes have abbreviations, but they are basically unused at this point. Some classes are just designed for inheritance purposes, so instances of this class cannot occur in the database. The “abstract” flag for these classes is *true*. The flag is *false* for classes that can be instantiated in the database. The “base” column contains a link to the superclass for this class where you can see the properties that are inherited.

Classes are organized into six modules: Cellar, FeatSys, LangProj, Ling, Notebk, and Scripture. Each module has a number.

Class information shows the module and the module number in which this class is defined. Each class has a unique number, or *class id*, that is made by appending the 3-digit class num value to the module *num* value. In this case the class id is 0066, or just 66.

Each property (attribute, or field) in the class also has a unique number, or *field id*, that is made by appending the 3-digit num to the class id. For example, the field id for the LouwNidaCodes property is 66001.

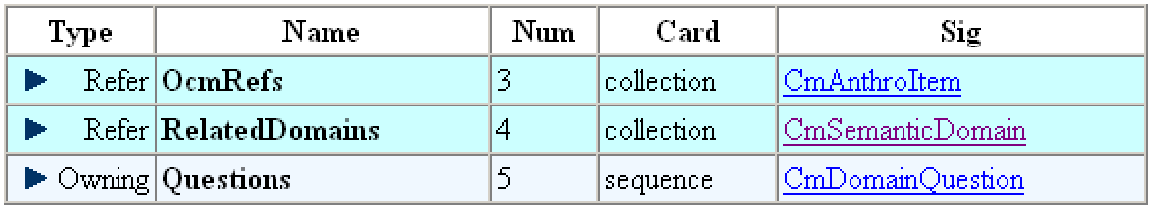
2.1.2 Basic attributes



Type	Name	Num	Signature	Other
▶ Basic	**LouwNidaCodes**	1	Unicode	
▶ Basic	**OcmCodes**	2	Unicode	

This section lists all the basic properties defined on this class. Each property has a name and a number that is used to build up the field id. The *Signature* indicates the type of basic property stored in this field. The *Other* column lists any other flags such as *min* and *max* values for integers.

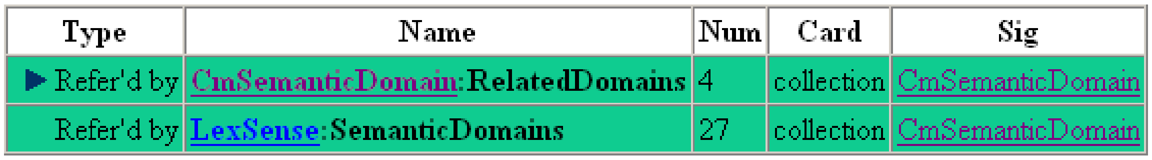
2.1.3 Owning and reference attributes



Type	Name	Num	Card	Sig
▶ Refer	**OcmRefs**	3	collection	CmAnthroItem
▶ Refer	**RelatedDomains**	4	collection	CmSemanticDomain
▶ Owning	**Questions**	5	sequence	CmDomainQuestion

This section lists all the owning and reference properties defined on this class. Each property has a name and number that is used to build up the field id. The *Card* column indicates whether the field is atomic, a collection, or a sequence. The *Sig* column indicates the class that is owned or referenced. This property can hold instances of this class or any of its subclasses.

2.1.4 Back references



Type	Name	Num	Card	Sig
▶ Refer'd by	CmSemanticDomain:**RelatedDomains**	4	collection	CmSemanticDomain
Refer'd by	LexSense:**SemanticDomains**	27	collection	CmSemanticDomain

This section shows any other classes that own or reference the current class, which is shown in the Sig column. The Name includes the class name and the name of the property on that class that owns or references the current class. The Num and Card columns refer to the properties described in the Name column.

In this case, CmSemanticDomain is referenced by the RelatedDomains property of CmSemanticDomain as well as by the SemanticDomains property of LexSense.

2.2 Diagrams

Click *Diagrams* in the upper left pane and the lower left pane gives a list of diagrams that illustrate certain parts of the conceptual model. These diagrams are semi-generated by MagicDraw with numerous hand edits. Click a link in the lower left pane and the right pane displays that diagram. The diagrams are fairly accurate, but occasionally things are missing that should be there. For any question, go to the class diagrams for accurate information.

In the diagram, each class is indicated by a rectangle. The name of the class is at the top and basic properties are listed inside the rectangle. Owning, reference, and inheritance lines connect the boxes. Click a class rectangle and the window jumps to show that class definition. Click a relationship line and the display jumps to show details on that relationship.

Click *Dictionary* in the upper left pane and the right pane gives an alphabetical list of all classes and relationship links along with the description for each one.

2.2.1 Possibility lists

The conceptual model diagram does not provide enough information to identify the type of possibility owned in each list. The following table lists the properties that own each

CmPossibilityList, and then gives the name of that list and the type of possibility or subclass that goes in that list.

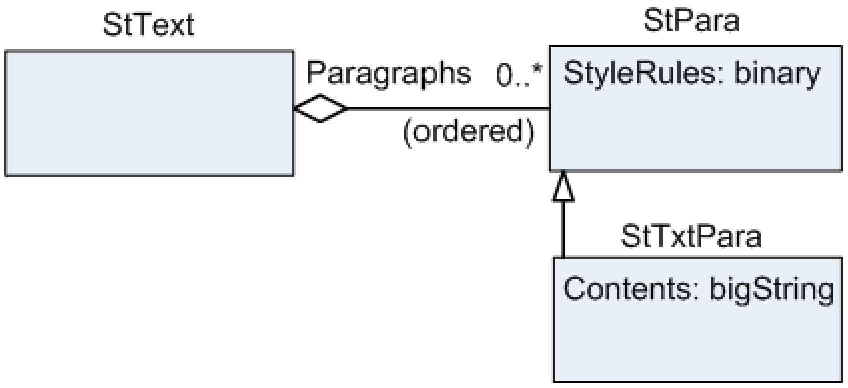
Class and Owning Property	**Name of list**	**Class in list**
DsDiscourseData_ChartMarkers	Chart Markers	CmPossibility
DsDiscourseData_ConstChartTempl	Constituent Chart Templates	CmPossibility
LangProject_AffixCategories (unused)	<none>	CmPossibility
LangProject_AnalysisStatus	Possible Status	CmPossibility
LangProject_AnnotationDefs	Annotation Definitions	CmAnnotationDefn
LangProject_AnthroList	Thesaurus	CmAnthroItem
LangProject_CheckLists	<none>	CmObject
LangProject_ConfidenceLevels	Confidence Levels	CmPossibility
LangProject_Education	Education Levels	CmPossibility
LangProject_GenreList	Genres	CmPossibility
LangProject_Locations	Locations	CmLocation
LangProject_PartsOfSpeech	Parts Of Speech	PartOfSpeech
LangProject_People	People	CmPerson
LangProject_Positions	Positions	CmPossibility
LangProject_Restrictions	Restrictions	CmPossibility
LangProject_Roles	Roles	CmPossibility
LangProject_SemanticDomainList	Semantic Domains	CmSemanticDomain
LangProject_TimeOfDay	Time Of Day	CmPossibility
LangProject_TranslationTags	Translation Types	CmPossibility
LangProject_WeatherConditions	Weather	CmPossibility
LexDb_AllomorphConditions	Minor Entry Conditions	CmPossibility
LexDb_DomainTypes	Academic Domains	CmPossibility
LexDb_EntryTypes	Entry Types	LexEntryType
LexDb_MorphTypes	Major Entry Types	MoMorphType
LexDb_References	Lexical Reference Types	LexRefType
LexDb_SenseTypes	Sense Types	CmPossibility
LexDb_Status	Sense Status	CmPossibility
LexDb_UsageTypes	Usages	CmPossibility
MoMorphData_ProdRestrict	Productivity Restrictions	CmPossibility
ReversalIndex_PartsOfSpeech	Parts Of Speech-English	PartOfSpeech
ReversalIndex_PartsOfSpeech	<none>	PartOfSpeech
RnResearchNbk_EventTypes	Event Types	CmPossibility
Scripture_NoteCategories	Scripture Note Categories	CmPossibility

The conceptual model diagram does not provide enough information to be able to identify the owning list for a possibility item (or subclass) that is referenced by a particular property. The following table lists properties that reference a possibility and gives the owning property of the list that owns these referenced items.

Reference to a CmPssibility or subclass	**Owner of CmPossibilityList owning items**
CmAnnotation_AnnotationType	LangProject_AnnotationDefs
CmOverlay_PossItems	Any possibility items
CmPerson_Education	LangProject_Education
CmPerson_PlaceOfBirth	LangProject_Locations
CmPerson_PlacesOfResidence	LangProject_Locations
CmPerson_Positions	LangProject_Positions
CmPossibility_Confidence	LangProject_ConfidenceLevels
CmPossibility_Researchers	LangProject_People
CmPossibility_Restrictions	LangProject_Restrictions
CmPossibility_Status	LangProject_AnalysisStatus
CmSemanticDomain_OcmRefs	LangProject_AnthroList
CmSemanticDomain_RelatedDomains	LangProject_SemanticDomainList

CmTranslation_Type	LangProject_TranslationTags
DsChart_Template	DsDiscourseData_ConstChartTempl
LexEntry_Condition	LexDb_AllomorphConditions
LexEntry_EntryType	LexDb_EntryTypes
LexPronunciation_Location	LangProject_Locations
LexSense_AnthroCodes	LangProject_AnthroList
LexSense_DomainTypes	LexDb_DomainTypes
LexSense_SemanticDomains	LangProject_SemanticDomainList
LexSense_SenseType	LexDb_SenseTypes
LexSense_Status	LexDb_Status
LexSense_ThesaurusItems	LexDb_Thesaurus (unused)
LexSense_UsageTypes	LexDb_UsageTypes
MoAffixAllomorph_MsEnvPartOfSpeech	LangProject_PartsOfSpeech
MoCompoundRule_ToProdRestrict	MoMorphData_ProdRestrict
MoDerivAffMsa_AffixCategory	LangProject_AffixCategories (unused)
MoDerivAffMsa_FromPartOfSpeech	LangProject_PartsOfSpeech
MoDerivAffMsa_FromProdRestrict	MoMorphData_ProdRestrict
MoDerivAffMsa_ToPartOfSpeech	LangProject_PartsOfSpeech
MoDerivAffMsa_ToProdRestrict	MoMorphData_ProdRestrict
MoDerivStepMsa_PartOfSpeech	LangProject_PartsOfSpeech
MoDerivStepMsa_ProdRestrict	MoMorphData_ProdRestrict
MoForm_MorphType	LexDb_MorphTypes
MoInflAffMsa_AffixCategory	LangProject_AffixCategories (unused)
MoInflAffMsa_FromProdRestrict	MoMorphData_ProdRestrict
MoInflAffMsa_PartOfSpeech	LangProject_PartsOfSpeech
MoStemMsa_FromPartsOfSpeech	LangProject_PartsOfSpeech
MoStemMsa_PartOfSpeech	LangProject_PartsOfSpeech
MoStemMsa_ProdRestrict	MoMorphData_ProdRestrict
MoUnclassifiedAffixMsa_PartOfSpeech	LangProject_PartsOfSpeech
Nothing_LexRefType It just uses owned objects	LexDb_References
ReversalIndexEntry_PartOfSpeech	ReversalIndex_PartsOfSpeech
RnAnalysis_Status	LangProject_AnalysisStatus
RnEvent_Locations	LangProject_Locations
RnEvent_Sources	LangProject_People
RnEvent_TimeOfEvent	LangProject_TimeOfDay
RnEvent_Type	RnResearchNbk_EventTypes
RnEvent_Weather	LangProject_WeatherConditions
RnGenericRec_AnthroCodes	LangProject_AnthroList
RnGenericRec_Confidence	LangProject_ConfidenceLevels
RnGenericRec_PhraseTags	Any possibility items
RnGenericRec_Researchers	LangProject_People
RnGenericRec_Restrictions	LangProject_Restrictions
RnRoledPartic_Participants	LangProject_People
RnRoledPartic_Role	LangProject_Roles
ScrImportSource_NoteType	LangProject_AnnotationDefs
ScrMarkerMapping_NoteType	LangProject_AnnotationDefs
ScrScriptureNote_Categories	Scripture_NoteCategories
StJournalText_CreatedBy	LangProject_People
StJournalText_ModifiedBy	LangProject_People
Text_Genres	LangProject_GenreList
WfiAnalysis_Category	LangProject_PartsOfSpeech
WordFormLookup_AnthroCodes	LangProject_AnthroList
WordFormLookup_ThesaurusItems	LexDb_Thesaurus (unused)

2.2.2 Structured text



An StText class presents users with a field that has multiple paragraphs and can be formatted similar to Word. StText owns a sequence of StPara which is an abstract class that holds StyleRules which holds compressed binary information about the paragraph style. Typically, it just uses a namedStyle property giving the name of an StStyle associated with the paragraph. It can also hold hard-coded style information such as text direction, line height, and indents. At this point, the only concrete subclass is StTxtPara which contains a FieldWorks String.

ModelDocumentation.chm contains a Structured Text diagram that gives more detail. For Scripture purposes, two subclasses of StText were added: StFootnote and StJournalText. CmTranslation was also added to StTxtPara to hold information such as back translations.

There are currently three stylesheets in a language project that hold StStyle instances defining paragraph or character styles.

- One is owned by Scripture_Styles and is used for Scripture.
- One is owned by LexDb_Styles and is used for the lexical database and interlinear text.
- One is owned by LanguagProject_Styles and is used by the Data Notebook and List Editor.

2.2.3 Language project

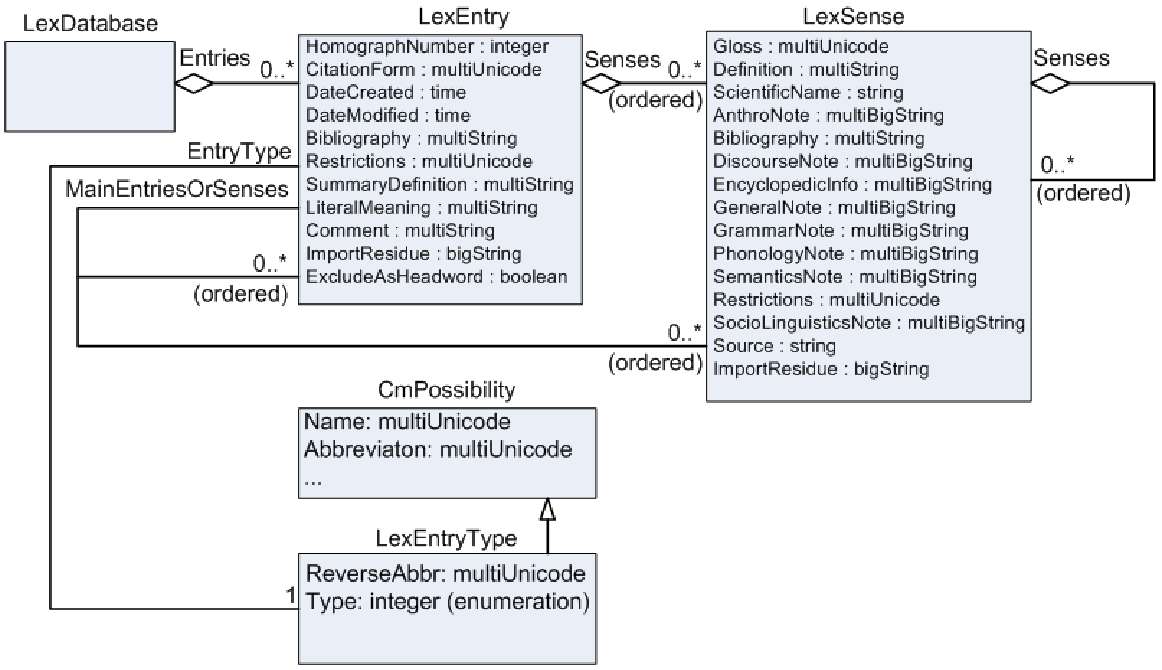
Each database holds one language project. The Language Project diagram in ModelDocumentation.chm gives more details. Among other things, a language project owns

- a lexical database
- a wordform inventory
- a collection of text (interlinear text)
- a research notebook
- Scripture
- morphological data
- a collection of annotations (largely for interlinear text)
- collections of picture and media objects pointing to external files, and
- numerous possibility lists.

It also references a sequence of active writing systems for vernacular, analysis, and pronunciations that determine the writing systems currently used to display your data.

The vernacular writing systems should always be different ways of representing the same vernacular language, such as standard orthography, IPA, PINYIN, and Romanized. Analysis writing systems can be any language or writing system for languages used for information such as glossing, definitions, and sentence translations.

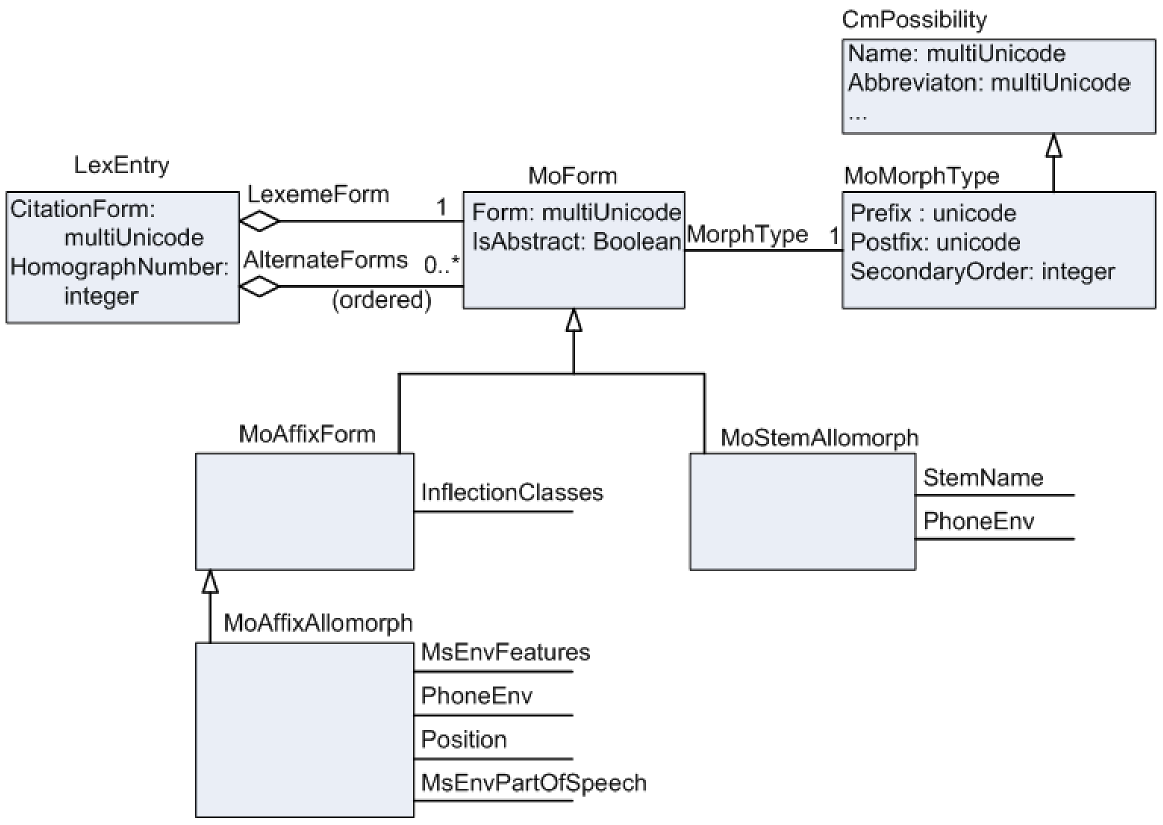
2.2.4 Lexical database


LexDatabase
Entries
0..*
LexEntry
HomographNumber : integer
CitationForm : multiUnicode
DateCreated : time
DateModified : time
Bibliography : multiString
Restrictions : multiUnicode
SummaryDefinition : multiString
LiteralMeaning : multiString
Comment : multiString
ImportResidue : bigString
ExcludeAsHeadword : boolean
Senses
0..*
(ordered)
LexSense
Gloss : multiUnicode
Definition : multiString
ScientificName : string
AnthroNote : multiBigString
Bibliography : multiString
DiscourseNote : multiBigString
EncyclopedicInfo : multiBigString
GeneralNote : multiBigString
GrammarNote : multiBigString
PhonologyNote : multiBigString
SemanticsNote : multiBigString
Restrictions : multiUnicode
SocioLinguisticsNote : multiBigString
Source : string
ImportResidue : bigString
Senses
0..*
(ordered)
EntryType
MainEntriesOrSenses
0..*
(ordered)
0..*
(ordered)
CmPossibility
Name: multiUnicode
Abbreviaton: multiUnicode
...
LexEntryType
ReverseAbbr: multiUnicode
Type: integer (enumeration)
1

The lexical database holds a collection of LexEntry. These entries may be main entries, subentries, or variants, depending on the EntryType. The Abbreviation of the LexEntryType is the abbreviation of the minor entry when referring to the major entry (e.g., der. of). The ReverseAbbr is the abbreviation used in the major entry to refer to the minor entry (e.g., der.) They may or may not show up in a printed dictionary based on the ExcludeAsHeadword flag. For subentries and variants, MainEntriesOrSenses references one or more entries or senses. The Type integer determines whether this entry works like a main entry (0), subentry (2), or minor entry (1).

Entries hold a sequence of senses which may be nested. The Lexical database diagram shows the basic properties of LexEntry and LexSense plus related classes. For full details, see the Lexical Database diagram in ModelDocumentation.chm.

2.2.5 Lexeme form and alternate forms

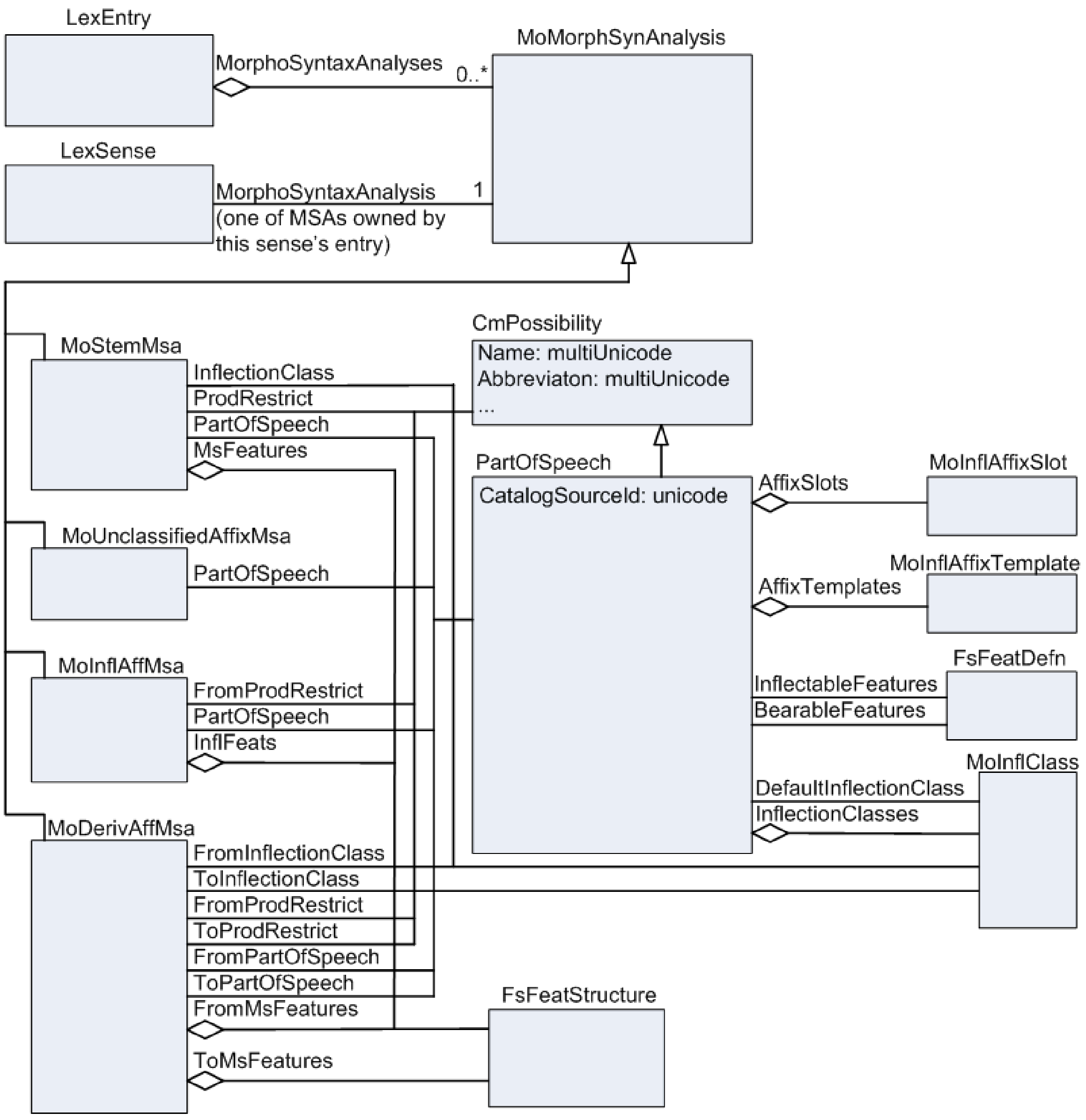


The Lexeme form is an owning property that holds an atomic instance of a subclass of MoForm. Alternate forms is a sequence of subclasses of MoForm. Concrete subclasses of MoForm are MoStemAllomorph and MoAffixAllomorph. (The MoAffixProcess has not been implanted yet.)

MoForm has a multiUnicode Form field. This allows for multiple vernacular writing systems such as standard orthography, IPA, PINYIN, and Romanized. It holds an IsAbstract flag for forms. It also references a MoMorphType such as root, prefix, and suffix. The MoMorphType contains strings that are attached before or after the citation form or lexeme form when producing the headword. Users can change these if needed. The SecondaryOrder determines the sort order when the form and homograph numbers are identical.

The allomorph has different reference properties depending on whether it is a MoAffixAllomorph or a MoStemAllomorph. More details on these classes are defined in the Lexical Database and Morphology diagrams of ModelDocumentation.chm.

2.2.6 Categories (parts of speech)



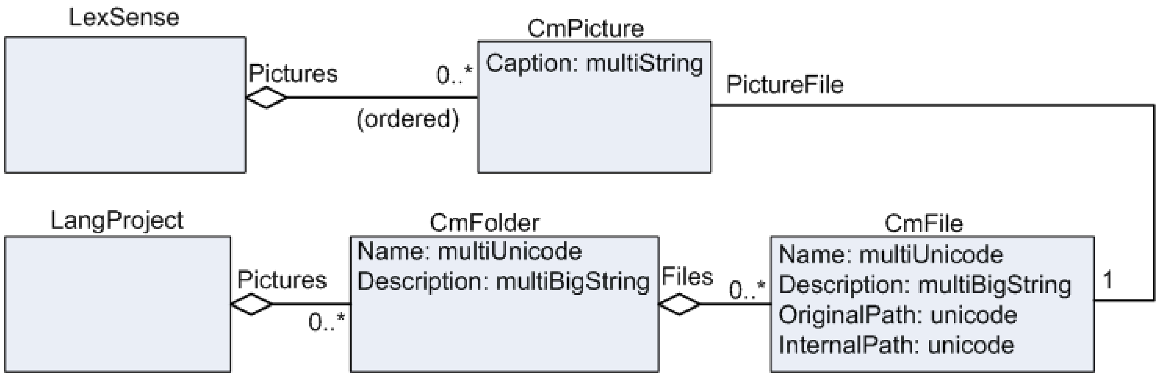
Categories (parts of speech) are much more complex than a simple string as in an SFM file. What would normally be a part of speech is actually a complex bundle that may include information such as parts of speech, features, inflections, and productivity restrictions. The PartOfSpeech is a subclass of CmPossibility and may own information such as affix templates, affix slots, and inflection classes. The Categories (parts of speech) diagram is a simplification of the LexDb and Morphology diagrams in ModelDocumentation.chm.

The additional information helps to define the grammar for your language. This information does not have to be available until it is needed. This is part of the stealth-to-wealth design of Flex. By adding additional information, the Flex parser becomes more accurate in predicting good morphological analyses and helps make the grammar sketch more complete.

Since the parser uses this information, it needs to be available on the entry rather than the sense. The entry contains information relevant to the grammar and the sense deals with the semantics, or meaning. Multiple senses in an entry may frequently use the same bundle of information. As a result, LexEntry owns a collection of MoMorphSynAnalysis (MSA) in the MorphoSyntaxAnalyses property. Each sense then references the MSA that contains the bundle it needs via the MorphoSyntaxAnalysis property (Grammatical info in the UI). MoMorphSynAnalysis is actually an abstract class. The actual instances are subclasses: MoStemMsa, MoUnclassifiedAffixMsa, MoInflAffMsa, and MoDerivAffMsa. Information for each of these appears at the bottom of the entry in the *Grammatical Info. Details* section(s).

For more information on morphology, see Help…Resources…Introduction to Parsing.

2.2.7 Pictures

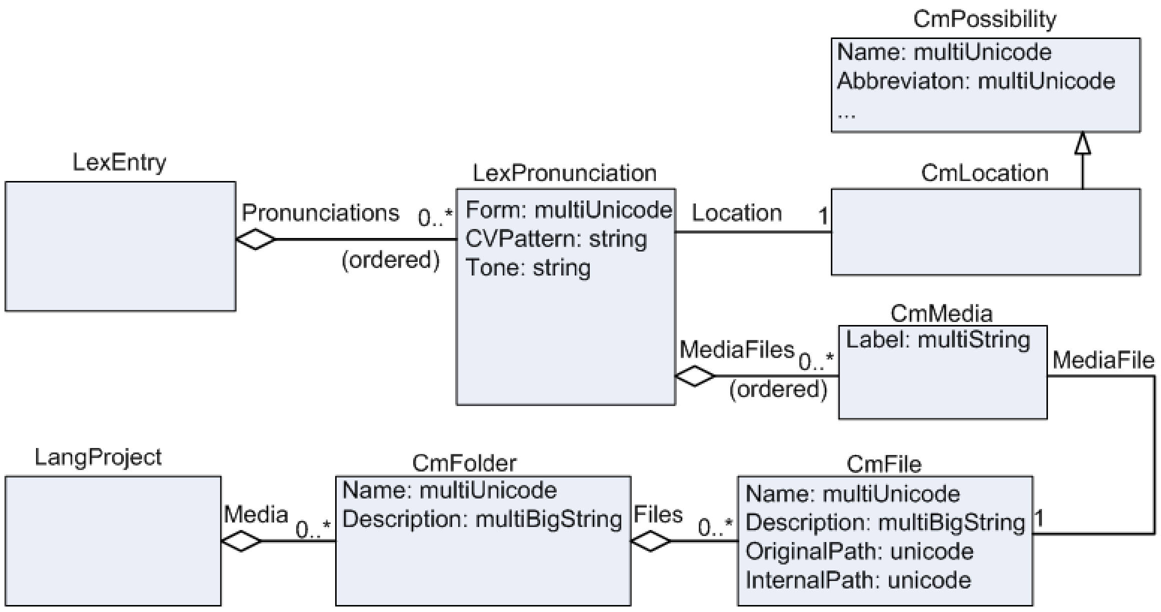


LexSense owns a sequence of CmPicture via the Pictures property. CmPicture has a Caption basic property and a PictureFile property which references a CmFile that locates the external file for the picture.

When users insert a picture file, Flex adds an instance of CmFile to the Files property of the CmFolder with a Name of *Local Pictures.* The CmFolder is owned in the Pictures owning collection property of LangProject. Although not shown, users can nest CmFolders. Currently, the original picture file is copied to the %ALLUSERSPROFILE%\Application Data\SIL\FieldWorks\Pictures directory (should become a user definable directory), InternalPath is set to the copied file, and OriginalPath is set to the original file. If the file already exists, Flex appends or increments an integer to the file name to keep it unique.

Note: %ALLUSERSPROFILE%\Application Data is c:\Documents and Settings\All Users\Application Data on Windows XP and c:\ProgramData on Vista.

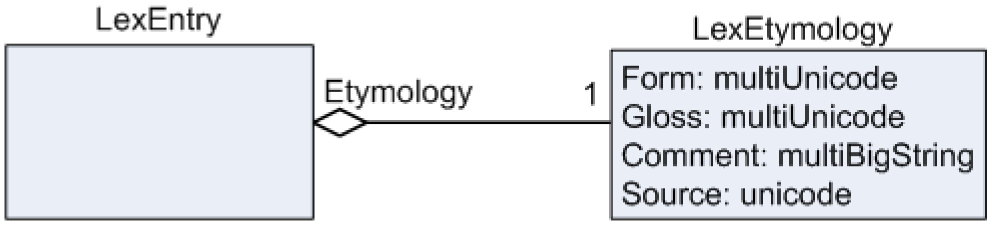
2.2.8 Pronunciations



LexEntry owns a sequence of LexPronunciation via the Pronunciations property. Each LexPronunciation has Form, CVPattern, and Tone basic properties. It has a reference to a CmLocation and also owns a sequence of CmMedia via the MediaFiles property. The CmMedia references a CmFile that locates the external file for the pronunciation, which can be a sound file or a video file.

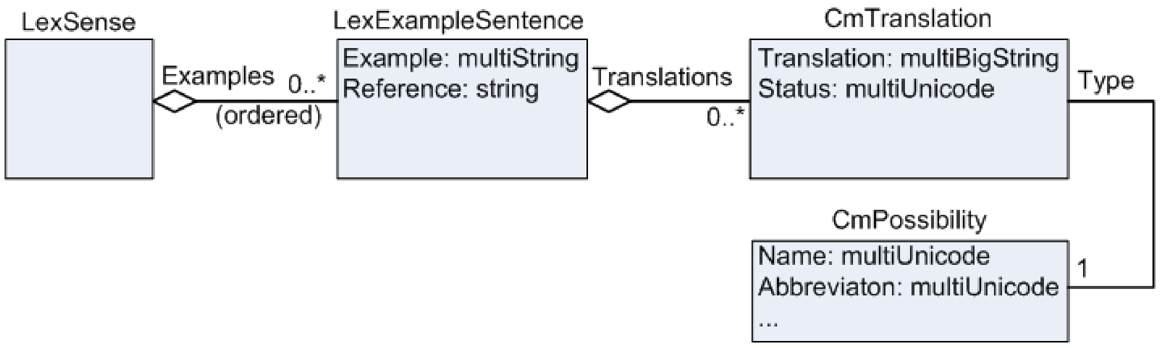
When users insert a media file, Flex adds an instance of CmFile to the Files property of the CmFolder with a Name of *Local Media*. The CmFolder is owned in the Media owning collection property of LangProject. Although not shown, users can nest CmFolders. Currently, the original media file is copied to the %ALLUSERSPROFILE%\Application Data\SIL\FieldWorks\Media directory (should become a user definable directory), InternalPath is set to the copied file, and OriginalPath is set to the original file. If the file already exists, Flex appends or increments an integer to the file name to keep it unique. Flex does not yet provide a UI to set or view media files but they are imported from LinguaLinks. Users will be able to access them as soon as the UI is available.

2.2.9 Etymology



A good model for etymology is not yet available. If users have a significant need for a better model, they should write a proposal. At this point, Flex contains something similar to MDF. LexEntry has an Etymology atomic owning property that owns a single instance of LexEtymology, which has Form, Gloss, Comment, and Source basic properties. (Source needs to be changed from Unicode to String so it contains a writing system.)

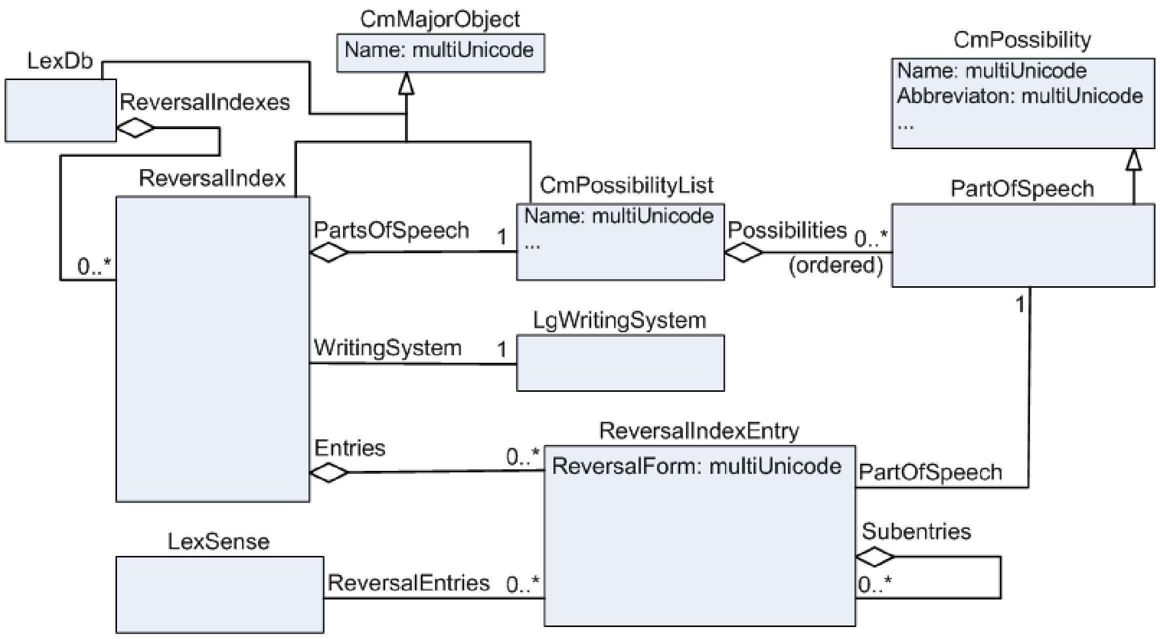
2.2.10 Example sentences



LexSense has an Examples owning sequence property holding LexExampleSentence which has an Example multiString property that allows the vernacular example to be in multiple writing systems. A Reference string property provides source information for the example. LexExampleSentence owns a collection of CmTranslation via the Translations property.

Each CmTranslation has a Type atomic reference property that refers to a CmPossibility owned in the TranslationTags property of LangProject. This allows users to specify whether it is a Free translation or a Literal translation (users can create new types if desired). CmTranslation has a multiBigString Translation property that allows users to enter translations in multiple languages and/or writing systems. This class is also used in the Scripture model where longer translations are needed, so this is a multiBigString instead of just multiString. The Status property is only used in the Scripture model.

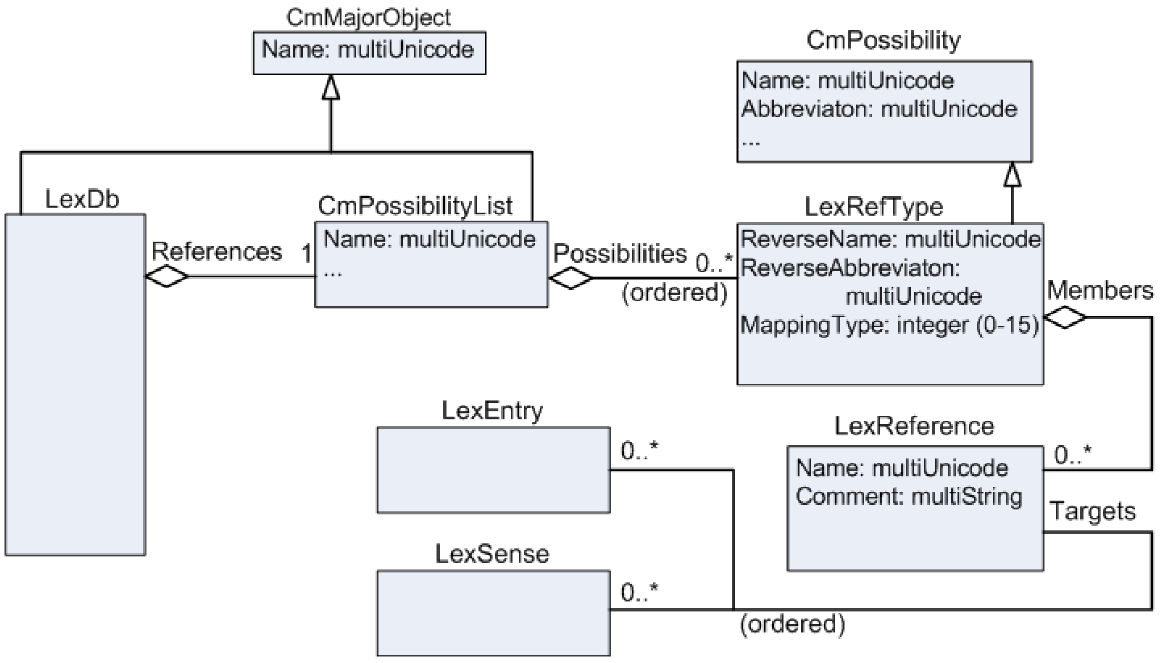
2.2.11 Reversal entries



In Flex, reversal indexes are separate index objects owned by the lexical database that hold a hierarchy of reversal entries. Each reversal index has a primary writing system. There is normally one reversal index for a given language, regardless of how many writing systems are used for that language. Reversal entries have a form capable of having multiple writing systems for the language of the primary writing system. (The language of a writing system is identified by the ICULocale string up to the first underscore.) Senses then reference these reversal entries. Senses can be linked to reversal entries from the Lexicon Edit view, the Bulk Edit Senses view, or the Reversal Indexes view.

Each reversal index owns a private copy of a parts of speech list since the categories may be different in the writing system of the index. The reversal entries can reference parts of speech in this private list. These private parts of speech lists can be edited from the Lists area using the Reversal Index Categories view.

2.2.12 Lexical relations & cross references



Lexical relations (sense) and cross references (entry) are defined in a possibility list holding LexRefType instances. Each LexRefType owns a collection of LexReference instances that define specific relations between senses and/or entries. LexReference has Name and Comment basic properties, but we currently do not provide a way in the UI to use these. The data is transferred during LinguaLinks transfers so the data will be available once the UI is added. The Targets sequence reference property on LexReference has a signature of CmObject because that is the only superclass over LexEntry and LexSense. The UI only allows LexSense and LexEntry in this property.

Each LexRefType has a MappingType that determines how various parts of this type of reference work. All LexReferences owned by a LexRefType are of the same mapping type. The mapping types are defined as follows:

0 Sense collection—one name (Lexical Relation) e.g., Synonym
1 Sense pair—one name (Lexical Relation) e.g., Antonym
2 Sense pair—two names (Lexical Relation)
3 Sense tree—two names (Lexical Relation) e.g., Part/Whole
4 Sense sequence/scale—one name (Lexical Relation)
5 Entry collection—one name (Cross Reference)
6 Entry pair—one name (Cross Reference)
7 Entry pair—two names (Cross Reference)
8 Entry tree—two names (Cross Reference)
9 Entry sequence/scale—one name (Cross Reference)
10 Entry or sense collection—one name (Cross Reference or Lexical Relation)
11 Entry or sense pair—one name (Cross Reference or Lexical Relation)
12 Entry or sense pair—two names (Cross Reference or Lexical Relation)
13 Entry or sense tree—two names (Cross Reference or Lexical Relation)
14 Entry or sense sequence/scale—one name (Cross Reference or Lexical

There are five basic types of relations, and each of these types can hold senses (0–4), entries (5–9), or combinations of senses and entries (10–14). Any reference to a sense shows up under Lexical Relations in the sense detail view. Any reference to an entry shows up under Cross References in the entry detail view. For sense references, the sense number is only shown if it is not the first sense in the entry.

Set relation

A set relation (e.g., synonyms–types 0, 5, 10) points to any number of senses via the targets property. The order can be ignored for these references. The name and abbreviation for the relation comes from CmPossibility. When a new sense is added from the chooser, Flex checks to see if the current sense or the chosen sense is already part of a LexReference of this type. If not, a new LexReference is created with the current sense and the chosen sense. If so, all linked senses for the existing one or two LexReferences are merged together into a single LexReference that encompasses the current sense, the chosen sense, and any senses to which they were previously linked.

Example

LexRefType: Name = Synonyms, Abbreviation = syn, MappingType = 0
LexReference_Targets: house, bungalow, cottage
House detail view: Synonyms bungalow | cottage
House dictionary view: **house** *n. syn.* **bungalow**, **cottage**
Bungalow detail view: Synonyms house | cottage
Bungalow dictionary view: **bungalow** *n. syn.* **house**, **cottage**
Cottage detail view: Synonyms house | bungalow
Cottage dictionary view: **cottage** *n. syn.* **house**, **bungalow**

Pair relation with one name

A pair relation with one name (e.g., antonyms–types 1, 6, 11) points to two senses via the targets property. The order is ignored for these references. The name and abbreviation for the relation comes from CmPossibility. When a new sense is added from the chooser, a new LexReference is created with the current sense and the chosen sense.

Example

LexRefType: Name = Antonym, Abbreviation = ant, MappingType = 1
LexReference_Targets: fast, slow
Fast detail view: Antonym slow
Fast dictionary view: **fast** *adj. ant.* **slow**
Slow detail view: Antonym fast
Slow dictionary view: **slow** *adj. ant.* **fast**

Pair relation with two names

A pair relation with two names (e.g., individual-group – types 2, 7, 12) point to two senses via the targets property. The order is significant for these references. The name and abbreviation for the relation on the first sense come from CmPossibility. The name and abbreviation for the relation on the second sense come from LexRefType ReverseName and ReverseAbbreviation. When a new sense is added from the chooser, a new LexReference is created with the current sense and the chosen sense.

Example

LexRefType: Name = Individual, Abbreviation = ind, ReverseName = Group, ReverseAbbreviation = grp, MappingType = 2
LexReference_Targets: lion, pride
Lion detail view: Group pride
Lion dictionary view: **lion** *n. grp.* **pride**
Pride detail view: Individual lion
Pride dictionary view: **pride** *n. ind.* **lion**

Tree relation

For a tree relation (e.g., generic/specific and part/whole–types 3, 8, 13) the first target represents the generic/whole sense and the remainder of the sequence is used for the specific/part senses. The CmPossibility name and abbreviation specify the specific/part labels while the reverse name and abbreviation from LexRefType are used for the generic/whole label. The order after the first sense is ignored for these references. When a new sense is added from the chooser:
A. The label is a normal name (specific/part) if there is a LexReference with the current sense as the first item in the sequence. Flex appends the chosen sense to this LexReference. Otherwise, it creates a new LexReference and adds the current sense and then the chosen sense to this LexReference.
B. The label is a reverse name (generic/whole) if there is a LexReference with the chosen sense as the first item in the sequence. Flex appends the current sense to this LexReference. Otherwise, it creates a new LexReference and adds the chosen sense and then the current sense to this LexReference.

Example

LexRefType: Name = Parts, Abbreviation = pt, ReverseName = Whole, ReverseAbbreviation = wh, MappingType = 3
LexReference_Targets: room, walls, ceiling
Room detail view: Parts walls | ceiling
Room dictionary view: **room** *n. pt.* **walls**, **ceiling**

Walls detail view: Whole room
Walls dictionary view: **walls** *n. wh.* **room**
Ceiling detail view: Whole room
Ceiling dictionary view: **ceiling** *n. wh.* **room**

Scale relation

For a scale relation (e.g., rank–types 4, 9, 14) the senses in the scale are referenced by the targets property. In this case, the ordering is significant. The name and abbreviation for the relation come from CmPossibility. When a new sense is added from the chooser, Flex appends the new sense to the current LexReference or creates a new one if none exists. Users can right-click an item in the scale detail view and choose to move it to the right or left via a context menu. Unlike the other relations, the current sense is displayed in the list of senses for a scale relation.

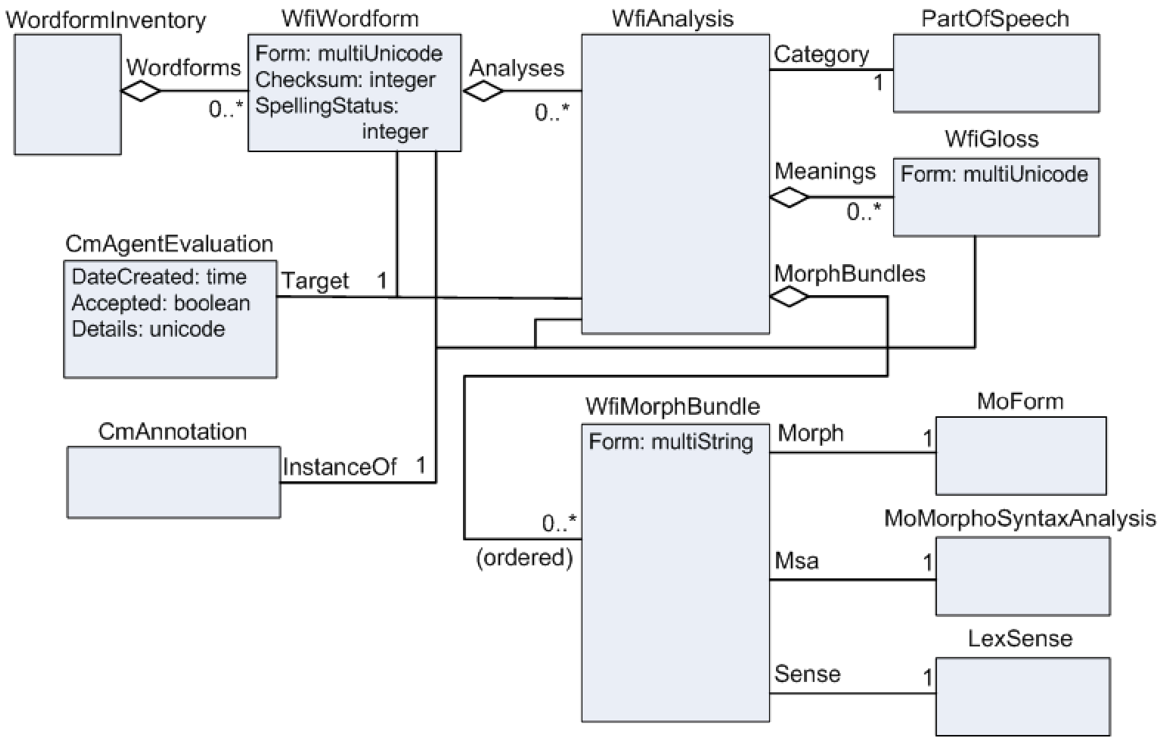
Example

LexRefType: Name = Calendar, Abbreviation = cal, MappingType = 4
LexReference_Targets: Monday, Tuesday, Wednesday
Monday detail view: Calendar Monday | Tuesday | Wednesday
Monday dictionary view: **Monday** *n. cal.* **Monday**, **Tuesday**, **Wednesday**
Tuesday detail view: Calendar Monday | Tuesday | Wednesday
Tuesday dictionary view: **Tuesday** *n. cal.* **Monday**, **Tuesday**, **Wednesday**
Wednesday detail view: Calendar Monday | Tuesday | Wednesday
Wednesday dictionary view: **Wednesday** *n. cal.* **Monday**, **Tuesday**, **Wednesday**

Because all the relations for a particular type are owned by the LexRefType, if one of these types is deleted from the Lexical Relation Type list, all the relations associated with it are also deleted.

In Flex, users never have to worry about adding lexical relations to both senses or entries. Adding one link automatically adds the corresponding link from the other reference.

2.2.13 Wordform inventory



The WordformInventory owns a collection of WfiWordform, representing all of the wordforms in interlinear texts. WfiWordform has a multiUnicode Form property to allow different writing systems for the same language. WfiWordform owns a collection of WfiAnalysis, each one representing a different morphological analysis of the wordform. The WfiAnalysis refers to a PartOfSpeech via the Category property. It owns a collection of WfiGloss holding word glosses in any number of languages. It also owns a sequence of WfiMorphBundle, one for each morpheme in the wordform. The WfiMorphBundle ties this morpheme to

- an entry—actually the MoForm of an entry, usually in its LexemeForm property
- an MSA owned by the entry, and
- a sense owned by the entry that refers to the MSA.

The WfiMorphBundle also has a Form basic property that can hold the morpheme string until it is analyzed to the references.

When users edit text in an interlinear text and move out of the Basline tab, the text is broken into wordforms. Any new wordforms are added to the wordform inventory. If the baseline text breaks in the wrong places, users can override the standard Unicode wordforming characters using WordFormingCharOverrides.xml located in their FieldWorks directory. This file currently affects all language projects on the computer.

Analyses are created as users work in the Interlinearize tab of an interlinear text. In this process, instances of CmBaseAnnotation (a subclass of the abstract CmAnnotation) are connected to the WfiWordform, WfiAnalysis, or WfiGloss, depending on the level of analysis. During this process, users may also add a WfiGloss, PartOfSpeech, or

WfiMorphBundle objects to the analysis. Also, CmAgentEvaluations are created indicating that a given wordform and/or analysis has been approved by the user.

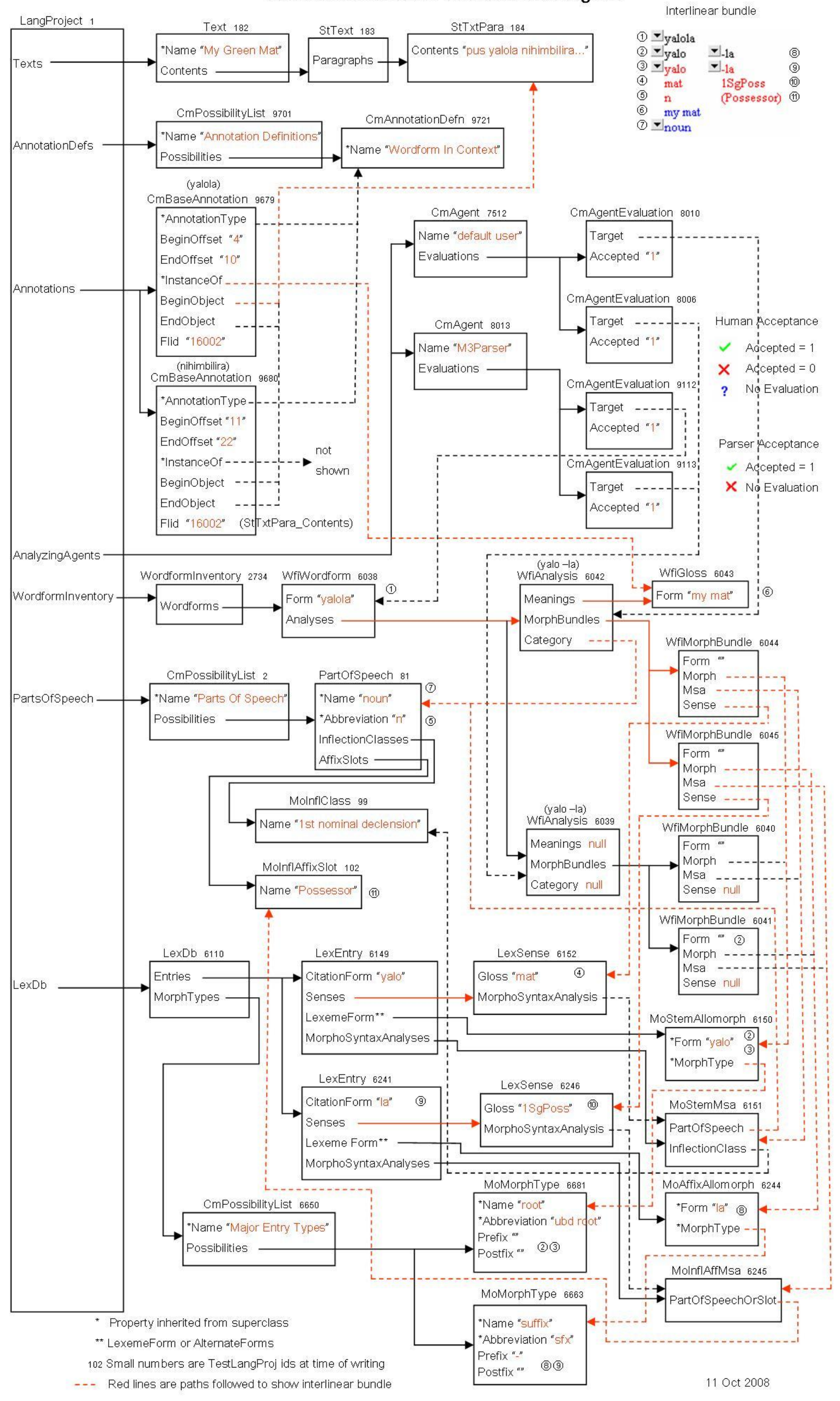
Analyses can also be created by the parser when users choose Parser…Start Parser from the Flex menu. The parser may also create WFiMorphBundles and fill in the Morph and Msa properties, but it will never fill in the Sense property. The parser does not fill in the Category or Meanings properties of the WfiAnalysis. The parser also creates CmAgentEvaluations identifying analyses that it makes.

Although WfiWordform has a multiUnicode Form property for holding forms in different writing systems of the same language, Flex is not yet prepared to interlinearize baseline text in more than one writing system. The missing piece is a way for users to match a wordform in a second writing system with the wordform that may already exist in the primary writing system, but is missing a form in the second writing system.

2.2.14 Interlinear text

Interlinear pictures such as the FieldWorks Interlinear Text Instance Diagram print better if you change the page to legal size. These diagrams use a non-UML format where owning properties are indicated with solid lines and reference properties are indicated with dotted lines. These diagrams are instance diagrams rather than conceptual model diagrams. They show a small example of an interlinear text illustrating two wordforms in the wordform inventory and how these are tied together between the text, wordform inventory, and lexical database. For details on the conceptual model, refer to Annotations, CmAgent, Interlinear Texts, Lexical Database, Morphology, Structured Text, and Wordform Inventory diagrams in ModelDocumentation.chm.

FieldWorks Interlinear Text Instance Diagram



An interlinear text is an instance of Text, which is a subclass of CmMajorObject. The Contents owning property holds an StText, which owns a sequence of StTxtPara. The baseline tab of the interlinear text is simply looking at the text from the StTxtPara. When users move out of the baseline tab, the text is broken into wordforms, creating new wordforms in the wordform inventory as needed. Flex also provides an option to show ScrSections, thus allowing a user to interlinearize the StTexts directly from Scripture.

When users go to the Interlinearize tab, Flex

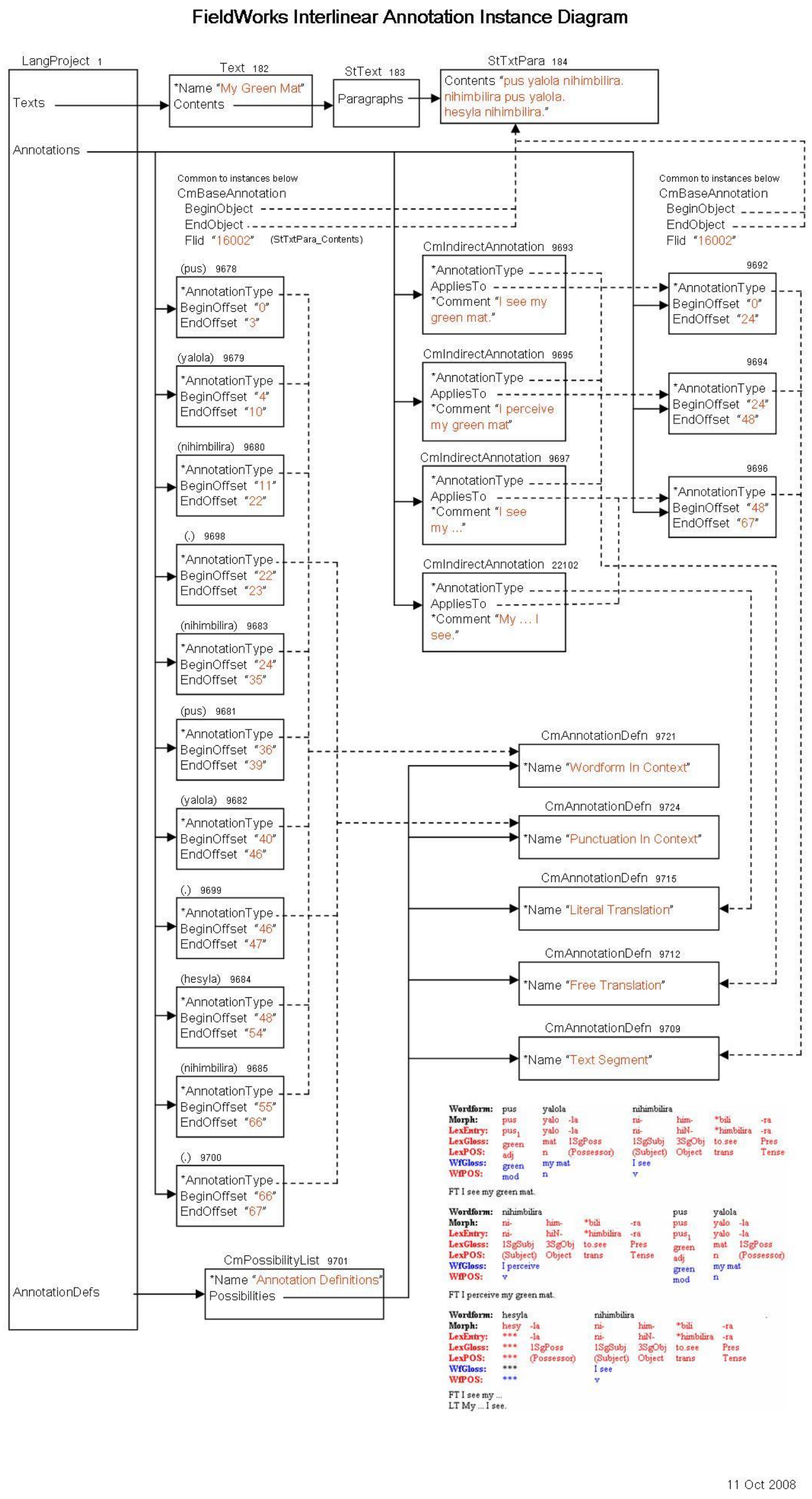
- uses existing CmBaseAnnotations for the wordforms in the text, or
- creates some in-memory CmBaseAnnotations for ones not yet user approved.

When users move the focus box, a CmBaseAnnotation will be created that represents the instance of that wordform in the text. This annotation points to the StTxtPara and holds beginning and ending offsets of the wordform in the paragraph string. It points to a "Wordform In Context" annotation type. The InstanceOf reference property is set to the WfiWordform initially. If users chose an analysis before moving the focus box, InstanceOf is set to the WfiAnalysis if there is no word gloss, otherwise it is set to the chosen WfiGloss.

When users edit the morph breaks, they create a new WfiAnalysis and set of WfiMorphBundles associated with these morph breaks. When users fill in the information for the entry, they set the properties on the WfiMorphBundle which ties the analysis to a sense, MSA, and MoForm (either LexemeForm or AlternateForms of the endty) in the lexicon.

When users approve the focus box, they also create a CmAgentEvaluation for the default user for that WfiAnalysis if it does not already exist.

The interlinear bundle inset on the upper right demonstrates that interlinear text in Flex is not a simple text string. It is a view of strings that are picked up throughout the conceptual model. The number for each string indicates where that string comes from in the instance diagram. The second (Morph) line comes from the WfiMorphBundle until the word is analyzed to a MoForm in a lexical entry, then it comes from the LexemeForm or an AlternateForm. The third (Lexical Entry) line shows homograph numbers and affix markers along with the LexemeForm.



```
Wordform:  pus     yalola              nihimbilira
Morph:     pus     yalo  -la           ni-        him-     *bili        -ra
LexEntry:  pus1    yalo  -la           ni-        hiN-     *himbilira   -ra
LexGloss:  green   mat   1SgPoss       1SgSubj    3SgObj   to.see       Pres
LexPOS:    adj     n     (Possessor)   (Subject)  Object   trans        Tense
WfGloss:   green   my mat              I see
WfPOS:     mod     n                   v
FT I see my green mat.

Wordform:  nihimbilira                                    pus     yalola
Morph:     ni-        him-     *bili        -ra           pus     yalo  -la
LexEntry:  ni-        hiN-     *himbilira   -ra           pus1    yalo  -la
LexGloss:  1SgSubj    3SgObj   to.see       Pres          green   mat   1SgPoss
LexPOS:    (Subject)  Object   trans        Tense         adj     n     (Possessor)
WfGloss:   I perceive                                     green   my mat
WfPOS:     v                                              mod     n
FT I perceive my green mat.

Wordform:  hesyla               nihimbilira
Morph:     hesy  -la            ni-        him-     *bili        -ra
LexEntry:  ***   -la            ni-        hiN-     *himbilira   -ra
LexGloss:  ***   1SgPoss        1SgSubj    3SgObj   to.see       Pres
LexPOS:    ***   (Possessor)    (Subject)  Object   trans        Tense
WfGloss:   ***                  I see
WfPOS:     ***                  v
FT I see my ...
LT My ... I see.
```

This FieldWorks Interlinear Annotation Instance Diagram illustrates more of the “Wordform In Context” and “Punctuation In Context” CmBaseAnnotations that are created as users analyze the text.

When moving to the Interlinearize tab, Flex breaks the baseline text into segments based on punctuation. It will automatically create a new segment for every period, question mark, exclamation point, or section sign (§). This may not be the place users desire to break segments, but at this point there is no other choice. For each segment, a CmBaseAnnotation is created with the AnnotationType pointing to the “Text Segment” CmAnnotationDefn.

If users add free translations, literal translations, or notes, a CmIndirectAnnotation is created with AnnotationType set to the “Free Translation”, “Literal Translation”, or “Note” CmAnnotationDefn. The translations and notes reference their segment CmBaseAnnotation using the AppliesTo reference property. The actual translation or note is held in the Comment property. This allows notes and translations in any number of languages and writing systems. For each segment, FieldWorks allows one free translation, one literal translation, and any number of notes.

There is a potentially serious problem in this version because of the automatic segment breaks that are automatically set up whenever users edit the baseline text. Everything works fine if users simply modify text within a segment, but as soon as they add or remove segment punctuation, reorder sentences, or combine paragraphs, translations and notes will likely be misaligned or lost. Flex tries to maintain these annotations, but it does not always work.

Example: Starting with text in one paragraph

A B.
translation for A B
C D.
translation for C D
E F.
translation for E F
G H.
translation for G H

Example: Trying to merge the first two sentences by removing the period after B

A B C D.
translation for A B translation for C D
E F.
translation for E F
G H.
translation for G H

This probably does what you would expect. However, if you put the period back in the baseline, you will not return to your original state because the free translations have now been merged. The next example demonstrates this.

Example: From original state, if users break first sentence into two by adding a period after A

A.
translation for A B translation for C D
B.
<no translation>
C D.
translation for E F
E F.
translation for G H
G H.

Again, the results are what you would expect. You’ll need to cut the B portion of the free translation from A’s translation and paste it into B’s translation..

Example: From original, if users reorder the first two sentences

C D.
translation for A B
A B.

E F.
translation for E F
G H.
translation for G H

When reordering sentences like this, Flex fails. The translation for A B is incorrectly moved to CD and the original translation for C D is lost.

Example: From original, if users start a new paragraph after C D.

A B.
translation for A B
C D.
translation for C D
E F.
translation for E F
G H.
translation for G H

In this case everything worked correctly.

Example: If users merge the two paragraphs

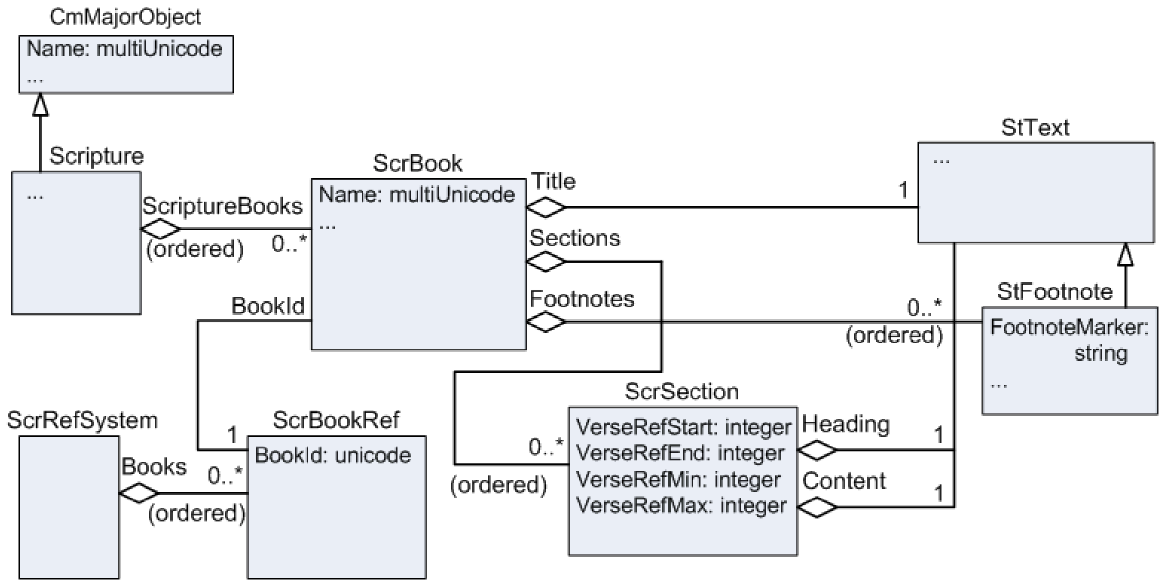
A B.
translation for A B
C D.
translation for C D
E F.
translation for E F
G H.
translation for G H

Again, this works as expected.

Changes to translations are only incorrect for the current paragraph, so users can mitigate against this problem by using short paragraphs.

FieldWorks 5.0 introduced moderate capabilities for interlinearizing baseline texts in more than one writing system for their vernacular language (e.g., IPA and standard orthography). It will work well if you follow carefully defined procedures. If you fail to follow these, you can end up with a mess that will be hard to fix, at least until some additional capabilities are added in future versions. See Flex tips.doc—13 Interlinearizing with multiple scripts for a technical discussion, or the help files for practical instructions.

2.2.15 Scripture

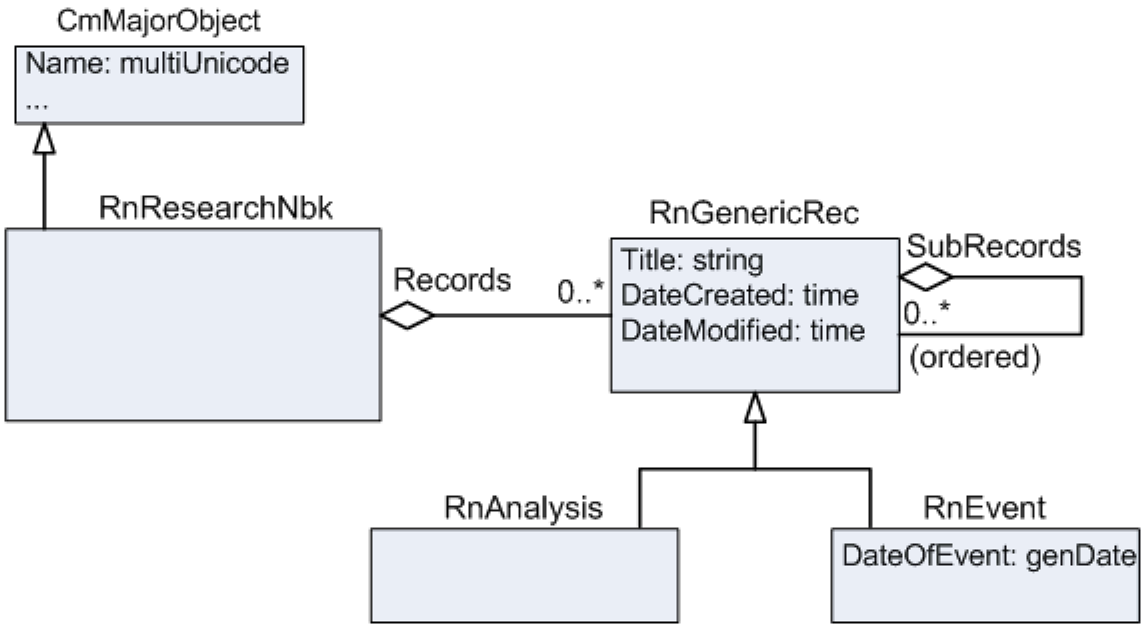


LangProject owns a single instance of Scripture. Scripture owns a sequence of ScrBook. Each book references a ScrBookRef through the BookId reference property. Each book owns a single StText in the Title owning property. The body of the book is a sequence of ScrSection owned in the Sections sequence property. Each section owns a single StText in the Heading property and a single StText in the Content property. StText owns a sequence of StTxtPara paragraphs, each with style and a content bigString with the text of the paragraph. ScrBook also owns a sequence of StFootnote, a subclass of StText.

The Translation Editor depends heavily on styles. Each StTxtPara has a style for paragraph formatting and various character styles are embedded in the paragraph string to hold items such as chapter numbers and verse numbers.

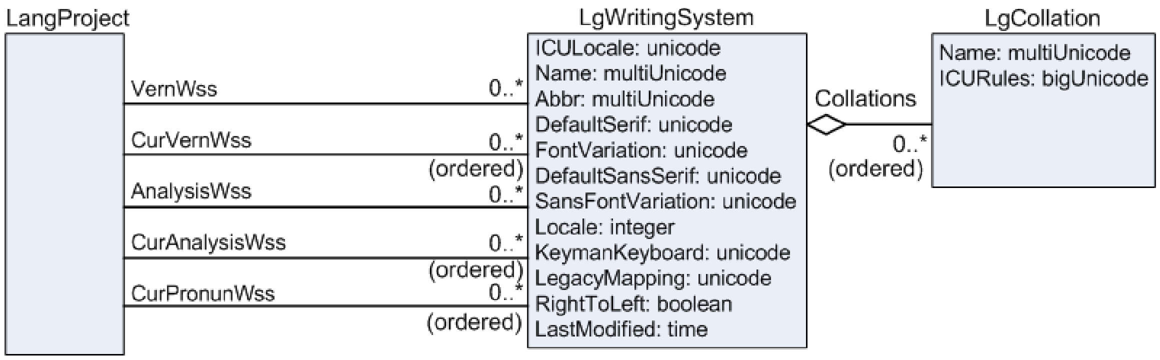
See the Scripture diagram in ModelDocumentation.chm for more detail on Scripture classes.

2.2.16 Data Notebook



The data notebook is implemented as RnResearchNbk, which owns a collection of Records. RnGenericRec is an abstract class that has RnAnalysis and RnEvent as concrete subclasses. Records have various owning properties that hold StText objects, plus numerous reference properties to possibility lists such as the People List and OCM Codes.

2.2.17 Writing Systems



Unlike most other objects, LgWritingSystem instances are unowned in the database[2]. The LangProject has several reference properties that determine the writing systems users see in various views. VernWss is a reference collection that identifies the vernacular writing systems that show up in the FieldWorks Project Properties Dialog…Writing System…Vernacular Writing Systems window. Of these writing systems, checked ones are indicated by the CurVernWss reference sequence property. The order of checked writing systems is important. For multistring vernacular properties, such as CitationForm, a line for each current writing system is displayed along with the Abbr of the writing system. For other vernacular properties, the writing system defaults to the first current

[2] Originally the plan was to have multiple language projects in one database and the LgWritingSystems were used by all projects in the database. This plan was abandoned for various reasons.

writing system. Vernacular writing systems should always be writing systems of the same language (e.g., standard orthography, IPA, Pinyin, and Romanized).

Analysis writing systems work similarly to vernacular writing systems. AnalysisWss is a reference collection that holds the writing systems in the Analysis Writing Systems window. CurAnalysisWss is a reference sequence for the writing systems that are checked. These writing systems determine the way analysis properties are shown in various views. Analysis writing systems can be any language or writing system within a language.

CurPronunWss is a reference sequence that determines the writing system(s) shown in the pronunciation field of lexical entries. It is changed by right-clicking on this field in a detail view and choosing the Writing System. Pronunciation writing systems should always be writing systems of the same language (e.g., standard orthography, IPA, Pinyin, and Romanized.)

LgWritingSystem holds most of the properties selected in the Writing System Properties…Name and Attribute tab. ICULocale is the key identification code of each writing system that should always be unique in each database. After clicking "Advanced," this writing system code can be seen in the Writing System Properties…Name tab. It consists of a language code with an optional region code and variant code separated by underscores. (The format is a result of ICU specifications.)

Examples

en	English standard orthography
en__IPA	English using an IPA (phonetic) writing system
en_US_IPA	English as spoken in the US using an IPA writing system
atd	Ata Monobo standard orthography
atd__EMC	Ata Monobo phonemic writing system

In several places, FieldWorks considers any writing system with the same code up to the first underscore as writing systems of the same language. The language portion of the code needs to use ISO-639-1 two-letter codes when defined in this standard. Otherwise use ISO-639-3 three-letter codes. If the language is not identified in ISO-639-3, Flex adds an "x" to the beginning of the first 3 letters of the language name to provide a four-letter code. In general, users should not specify a regional code unless it is required for some reason. For example, FieldWorks has many English strings for possibility lists that are encoded with "en". If users try to use en_US for their default analysis writing system, it will be considered a different writing system from the English that is already recorded in the system. They can use the region portion to identify a dialectal variant or a region of the area where they work where this distinction is needed. (More development is needed to handle dialects adequately.) The variant can be used for alternate writing systems for the language such as IPA (phonetic), EMC (phonemic), PIN (Pinyin), or ROM (Romanized) forms.

A writing system has a default font for normal use and a default font for headings. Each has a font variation property used by Graphite fonts to record various font features selected by the user.

A writing system records the system keyboard in the Locale property. Users can optionally include the name of a Keyman keyboard.

- LegacyMapping is a property for the default SIL Encoding Converter to convert legacy 8-bit data to Unicode. If missing, it is assumed the input file is in Unicode UTF-8. (TE always uses this for import. Flex allows users to override this.)
- The RightToLeft flag is set to 1 for Arabic or other writing systems that are rendered right-to-left.
- The LastModified property keeps track of the date the language definition XML file (in your FieldWorks languages directory) was last modified. It stores this time in GMT. If the language definition file has a different date, FieldWorks will load its information into the LgWritingSystem when that writing system is used. For more details, see *Writing systems in FieldWorks* section of ICU and writing systems.doc.

Writing systems can have one or more LgCollation instances that define collations to use when sorting or searching data. At present the UI only supports one LgCollation. For more details, go to the *Collation setup* section of of ICU and writing systems.doc.

Note: There may be more LgWritingSystem instances in the database than show up in the FieldWorks Project Properties Dialog…Writing System tab. FieldWorks does not currently provide a way in the UI to delete an LgWritingSystem from the database. For ways to do this, see *Removing a writing system* in FieldWorks XML model.doc and InstallLanguage section of ICU and writing systems.doc.