caCORE EVS API

Technical Guide

NATIONAL CANCER INSTITUTE®

Center for Bioinformatics

This is a U.S. Government work.

June 18, 2008

CONTENTS

About This Guide 1
Purpose 1
Release Schedule 1
Audience 2
Additional caCORE Documentation 2
Organization of This Guide 2
Text Conventions Used 3
Credits and Resources 4

Chapter 1
About the Enterprise Vocabulary Services and LexBIG 7
Introduction 7
Key EVS Terminologies 8
NCI Thesaurus 8
NCI Metathesaurus 8
About the UMLS Metathesaurus 9
Preservation of Terminology 9
Disambiguation of Terminology 9
Defined Relationships 10
Categorization of Concepts 10
About Knowledge Representation 10
First-Order Predicate Logic 11
Description Logic 12
Description Logic in the NCI Thesaurus 14
Concept Edit History in the NCI Thesaurus 15
Working with the NCI Thesaurus 16
Downloading the NCI Thesaurus 16
OWL Encoding of the NCI Thesaurus 18
Ontylog Name Conversion 19
Ontylog Mappings 21
Mapping of Gene Ontology to Ontylog 21

Mapping of MedDRA to Ontylog 22
Vocabulary Hierarchy Structure 22
Concept Codes and Names 23
Roles 23
Properties 23
Mapping of MGED Ontology to Ontylog 24
Vocabulary Hierarchy Structure 24
Concept IDs, Codes, and Names 24
Roles 24
Properties 24
About LexBIG 25

Chapter 2
About caCORE 27

Architecture Overview 27
Components of caCORE 28
Enterprise Vocabulary Services (EVS) 28
Cancer Data Standards Repository (caDSR) 28
Cancer Bioinformatics Infrastructure Objects (caBIO) 28
Common Security Model (CSM) 29
Common Logging Module (CLM) 29
About the caCORE 4.0 Release Paradigm 29

Chapter 3
About the caCORE EVS Architecture 31

caCORE EVS System Architecture 31
Client Technologies 33
caCORE EVS Software Packages 34
Domain Packages 34
System Packages 35

Chapter 4
Interacting with caCORE EVS 37

caCORE EVS Components 38
EVS 3.2 Object Model 39
EVS 3.2 Domain Object Catalog 40
EVS Data Sources 41
NCI Thesaurus 41
NCI Metathesaurus 41
Installing and Configuring the EVS 3.2 Java API 41
Software Requirements 42

This is a U.S. Government work.

Downloading the Java API Client Package 42
Installing the Package 43
Verifying the Installation 45
Search Paradigm 47
About EVSQuery and EVSQueryImpl 48
EVSQuery Methods and Parameters 48
Using EVSQuery 50
Accessing Secured Vocabularies 50
Creating an EVS Search Request 51
Examples of Use 51
Web Services API 55
Configuration 55
Operations 56
Considerations 57
Examples of Use 58
Limitations 61
XML-HTTP API 62
Service Location and Syntax 62
Examples of Use 63
Working with Result Sets 64
Retrieving Related Results Using XLinks 64
Controlling the Number of Items Returned 64
Paging Results 64
Limitations 64
About the XMLUtility Class 65
Overview 65
Example of Use 65
Distributed LexBIG API 67
Overview 67
Architecture 67
LexBIG Annotations 68
Aspect Oriented Programming Proxies 68
LexBIG API Documentation 69
LexBIG Installation and Configuration 69
Example of Use 69
Distributed LexBIG Adapter Example of Use 71

Appendix A
Resources Used to Create This Guide 79

Books and Articles 79
caBIG Material 79

caCORE Material 79
Software Products 80

Appendix B
Additional Examples 81

Find Tree for Concept and Association 81
Search Metathesaurus for a Specific Concept/Search Term 84

Glossary 87

Index 91

 This is a U.S. Government work.

About This Guide

This section introduces you to the *caCORE EVS API Technical Guide*.

Topics in this section:

- *Purpose* on this page
- *Release Schedule* on this page
- *Audience* on page 2
- *Additional caCORE Documentation* on page 2
- *Organization of This Guide* on page 2
- *Text Conventions Used* on page 3
- *Credits and Resources* on page 4

Purpose

The *caCORE EVS API Technical Guide* describes the Enterprise Vocabulary Services (EVS) component of the Cancer Common Ontologic Representation Environment (caCORE). caCORE is an open-source, standards-based, semantics computing environment and tool set created by the National Cancer Institute Center for Bioinformatics (NCICB). EVS is a set of services and resources that address the NCI's needs for a controlled vocabulary. It provides the semantic base upon which the data semantics of caCORE depend.

This guide explains the following:

- the purpose, architecture, and components of caCORE EVS
- the APIs for accessing the caCORE EVS system, including Java, Web services, and XML-HTTP
- the API providing direct remote access to the native LexBIG Service Layer
- an overview of UML.

Release Schedule

This guide is updated for each caCORE EVS release. It may be updated between releases if errors or omissions are found. The current guide refers to the 4.1 version of caCORE EVS, which was released in June 2008 by the NCICB.

Audience

This guide is written for application developers who want to learn about the architecture and use of the caCORE EVS APIs and who may want to access those APIs. caCORE EVS is generated using the caCORE Software Development Kit (SDK). For more information, see the caCORE SDK 4.0 Developer's Guide.

This guide assumes that you are familiar with the Java programming language and/or other programming languages, database concepts, and the Internet. If you intend to use caCORE EVS resources in software applications, the guide assumes that you have experience with building and using complex data systems. Neither caCORE EVS nor this documentation is intended for end users such as individual health professionals or members of the general public, unless they are also software developers.

Additional caCORE Documentation

- The *caCORE EVS 4.1 Release Notes* include a description of the end user tool enhancements and bug fixes included in this release.
- The *caCORE EVS 4.1 JavaDocs* include the current caCORE EVS API specification.
- The *caCORE SDK 4.1 Developer's Guide* includes detailed instructions on the use of the SDK. It also explains how the SDK aids in creating a caCORE-like software system.

Organization of This Guide

This brief overview explains what you will find in each section of this guide.

- *About This Guide* (this section) provides an overview of this guide.
- *Chapter 1, About the Enterprise Vocabulary Services and LexBIG,* provides an overview of the Enterprise Vocabulary Services (EVS) project.
- *Chapter 2, About caCORE,* provides an overview of the NCICB caCORE infrastructure.
- *Chapter 3, About the caCORE EVS Architecture,* describes the architecture of the caCORE EVS.
- *Chapter 4, Interacting with caCORE EVS,* describes the caCORE EVS API, the service interface layer provided by the EVS API architecture. It gives examples of how to use the EVS API. It also describes the distributed LexBIG API.
- *Appendix A, Resources Used to Create This Guide,* provides a list of print and online resources that were used to produce this guide or which were referenced in this guide.
- *Appendix B, Additional Examples,* provides two additional code examples.

Text Conventions Used

This section explains conventions used in this guide. The various typefaces represent interface components, keyboard shortcuts, toolbar buttons, dialog box options, and text that you type.

Convention	*Description*	*Example*
Bold	Highlights names of option buttons, check boxes, drop-down menus, menu commands, command buttons, or icons.	Click **Search**.
URL	Indicates a Web address.	http://domain.com
text in SMALL CAPS	Indicates a keyboard shortcut.	Press ENTER.
text in SMALL CAPS + text in SMALL CAPS	Indicates keys that are pressed simultaneously.	Press SHIFT + CTRL.
Italics	Highlights references to other documents, sections, figures, and tables.	See *Figure 4.5*.
Italic boldface monospaced type	Represents text that you type.	In the **New Subset** text box, enter ***Proprietary Proteins.***
Note:	Highlights information of particular importance	**Note:** This concept is used throughout the document.
{ }	Surrounds replaceable items.	Replace {last name, first name} with the Principal Investigator's name.

Credits and Resources

The following people contributed to the development of this guide.

caCORE EVS Development and Management Teams		
Development	***Technical Guide***	***Program Management***
Johnita Beasley[2]	Johnita Beasley[2]	Bill Britton[6]
Steve Hunter[7]	Eddie VanArsdall[4]	Peter Covitz[1]
Norval Johnson[6]	Frank Hartel[1]	Frank Hartel[1]
Sriram Kalyanasundaram[6]	Shaziya Muhsin[2]	Charles Griffin[7]
Doug Kanoza[6]	Kim Ong[3]	Jason Lucas[3]
Alan Klink[6]	Konrad Rokicki[2]	Krishnakant Shanbhag[1]
Shaziya Muhsin[2]	Tracy Safran[2]	Denise Warzel[1]
Kim Ong[3]		
John Park[4]		
Konrad Rokicki[2]		
Tracy Safran[2]		
Ye Wu[2]		
Robert Wynne[4]		
Robert Wysong[6]		
David Yee[3]		
[1] National Cancer Institute Center for Bioinformatics (NCICB)	[2] Science Application International Corporation (SAIC)	[3] Northrup Grumman
[4] Lockheed Martin	[5] ScenPro, Inc.	[6] Terpsys
[7] Ekagra Software Technologies		

Contacts and Support	
NCICB Application Support	http://ncicbsupport.nci.nih.gov/sw/ Telephone: 301-451-4384 Toll free: 888-478-4423

LISTSERV Facilities Pertinent to the caCORE EVS		
LISTSERV	***URL***	***Name***
NCI EVS Listserv	https://list.nih.gov/archives/ncievs-l.html	NCI Vocabulary Services Information
caCORE_SDK_Developers	https://list.nih.gov/archives/cacore_sdk_dev-l.html	caCORE SDK Developers Discussion Forum
caCORE_SDK_Users	https://list.nih.gov/archives/cacore_sdk_users-l.html	caCORE SDK Users Discussion Forum

CHAPTER 1

ABOUT THE ENTERPRISE VOCABULARY SERVICES AND LEXBIG

This chapter provides an overview of the Enterprise Vocabulary Services (EVS) project.

Topics in this chapter:

- *Introduction* on this page
- *Key EVS Terminologies* on page 8
- *About the UMLS Metathesaurus* on page 9
- *About Knowledge Representation* on page 10
- *Concept Edit History in the NCI Thesaurus* on page 15
- *Working with the NCI Thesaurus* on page 16
- *Ontylog Mappings* on page 21
- *About LexBIG* on page 25

Introduction

The NCI Enterprise Vocabulary Services (EVS) is a partnership between the NCI Center for Bioinformatics and the NCI Office of Communications. Since 1997, EVS has worked to harmonize and integrate the many diverse terminologies and coding frameworks used by the NCI and its partners.

EVS serves a critical need by providing a well-designed ontology covering cancer science. Such an ontology is required for data annotation, inferencing, and other functions. Annotated data range from genomic sequences and case report forms to cancer image data.

EVS is active in NIH-wide harmonization initiatives, federal standards development efforts, and other standards development organizations. These activities help to develop terminology resources and software tools to facilitate compatible coding, retrieval, and aggregation of biomedical information.

Key EVS Terminologies

The establishment of controlled vocabularies is important to any application involving electronic data sharing. The importance of controlled vocabularies is perhaps most apparent in clinical trials data collection and data reporting. It is also important in general data annotation of any kind.

To respond to the need for consistency among various NCI projects and initiatives, the NCI publishes the *NCI Thesaurus* and the *NCI Metathesaurus*. The caCORE EVS interfaces, discussed later in this guide, provide access to both of these terminologies, which are discussed in the next two sections.

NCI Thesaurus

EVS publishes the NCI Thesaurus (NCIT) as a core reference terminology and biomedical ontology. Implemented as a Description Logic vocabulary, the NCIT is a self-contained and logically consistent terminology, providing rich textual and ontological descriptions of some 50,000 key biomedical concepts.

The NCIT was developed by EVS in response to a need for consistent shared vocabularies among the various projects and initiatives at the NCI, as well as in the entire cancer research community. Published monthly, the NCIT is used in a growing number of NCI and other systems.

NCI Metathesaurus

The NCI Metathesaurus is a comprehensive biomedical terminology database that contains 1,100,000 concepts mapped to 2,500,000 terms with 5,000,000 relationships. Based on the Unified Medical Language System Metathesaurus (UMLS) developed by the National Library of Medicine (NLM), the NCI Metathesaurus includes most UMLS terms and supplements them with additional cancer-centric vocabulary. It excludes certain proprietary vocabularies and includes others with restricted use.

Some of the NCI Metathesaurus vocabularies were developed locally by the NCI, and others were licensed. *Table 1.1* describes those vocabularies that were developed locally. Note that a limited model of the NCI Thesaurus is accessible through the Metathesaurus as the NCI Source. Additional external, proprietary vocabularies include MedDRA, SNOMED, and ICD-O-3, among others.

Vocabulary	***Content***	***Usage***
NCI Source	Limited model of the NCI Thesaurus	Reference terminology for cancer research applications
NCIPDQ	Expanded and reorganized PDQ	CancerLit indexing and clinical trials accrual
NCISEER	SEER terminology	Incidence reporting
CTEP	CTEP terminology	Clinical trials administration

Table 1.1 NCI local source vocabularies included in the Metathesaurus

Vocabulary	***Content***	***Usage***
MDBCAC	Topology and morphology	Cancer genome research
ELC2001	NCBI tissue taxonomy	Tissue classification for genetic data such as cDNA libraries
ICD03	Oncology classifications	Cancer genome research and incidence reporting
MedDRA	Regulatory reporting terminology	Adverse event reporting
MMHCC	Mouse Cancer Database terminology	Mouse Models of Human Cancer Consortium
CTRM	Core anatomy, diagnosis, and agent terminology	Translational research by NCICB applications

Table 1.1 NCI local source vocabularies included in the Metathesaurus (Continued)

Unlike the NCI Thesaurus, the NCI Metathesaurus is not designed to provide unequivocal or consistent definitions. Like the UMLS, its purpose is to provide mappings of terms across vocabularies.

For more information about the UMLS Metathesaurus, see the next section.

About the UMLS Metathesaurus

The NCI Metathesaurus is based on the National Library of Medicine's Unified Medical Language System Metathesaurus (UMLS Metathesaurus). This section provides a brief overview of the UMLS Metathesaurus features that are relevant to accessing the NCI Metathesaurus. More detailed information about the UMLS Metathesaurus is available on the UMLS Knowledge Sources Web site at http://www.nlm.nih.gov/research/umls/umlsdoc.html.

Preservation of Terminology

The UMLS Metathesaurus is a unifying database of concepts that brings together terms occurring in over 100 different controlled vocabularies used in biomedicine. When editors add terms to the UMLS Metathesaurus, they preserve all of the original meanings, attributes, and relationships defined in the source vocabularies, and they retain explicit source information. They also add basic information about each concept and introduce new associations that help to establish synonymy and other relationships among concepts from different sources.

Disambiguation of Terminology

Given the large number of related vocabularies incorporated in the UMLS Metathesaurus, instances occur in which the same concept may be known by many different names. In other cases, the same names are intended to convey different concepts. To avoid ambiguity, the UMLS uses an elaborate indexing system that assigns a *concept unique identifier* (CUI) to each concept name. Similarly, each unique concept name or string in the Metathesaurus is assigned a *string unique identifier* (SUI).

In cases where one string is associated with multiple concepts, a numeric tag is appended to that string to render it unique and to reflect its multiplicity. UMLS

Metathesaurus editors can also create an alternative name for the concept that is more indicative of its intended interpretation. In such cases, all three concept names are preserved.

Defined Relationships

The UMLS Metathesaurus uses several types of defined relationships. The NCI Metaphrase interface captures the four relationships described in *Table 1.2*.

Relationship	Description
Broader (RB)	The related concept has a more general meaning.
Narrower (RN)	The related concept has a more specific meaning.
Synonym (SY)	The two concepts are synonymous.
Other related (RO)	The relationship is not specified, but it is something other than broader, narrower, or synonymous.

Table 1.2 Relationships defined in the UMLS Thesaurus

Categorization of Concepts

The UMLS *Semantic Network* is an independent construct that provides consistent categorization for all concepts contained in the UMLS Metathesaurus and defines a useful set of relationships among those concepts. As of the 2005AC release, the Semantic Network defined a set of 135 basic semantic types or categories that could be assigned to concepts, as well as 54 relationships that could hold among the various types.

Major groupings of semantic types include organisms, anatomical structures, biologic function, chemicals, events, physical objects, and concepts or ideas. Each UMLS Metathesaurus concept is assigned at least one semantic type. In some cases, several types are assigned. In all cases, the most specific semantic type available in the network hierarchy is assigned to the concept.

About Knowledge Representation

Knowledge representation has long been a prime focus in artificial intelligence research. This area of research asks how we can accurately encode the rich and highly detailed world of information that is required for the application area being modeled and yet, at the same time, capture the implicit common sense knowledge. One of the most common approaches to this problem in the 1970s was to use *frame-based representations*.

The basic idea of a *frame* is that important objects in our world fall into natural classes, and that all members of these classes share certain properties or attributes, called *slots*. For example, all dogs have four legs, a tail (or a vestige of one), and whiskers. Restaurants generally have tables, chairs, eating utensils, and menus. Thus, when we enter a new restaurant or encounter a new dog, we already have a frame of reference and some expectation about the properties and behaviors of these entities.

In a seminal 1975 paper by Marvin Minsky, the author placed the frame representation paradigm in the context of a semantic network of *nodes*, *attributes*, and *relations*. *Figure 1.1* shows this simple frame-based representation of an earthquake as it might be used in a semantic network of news stories.[1]

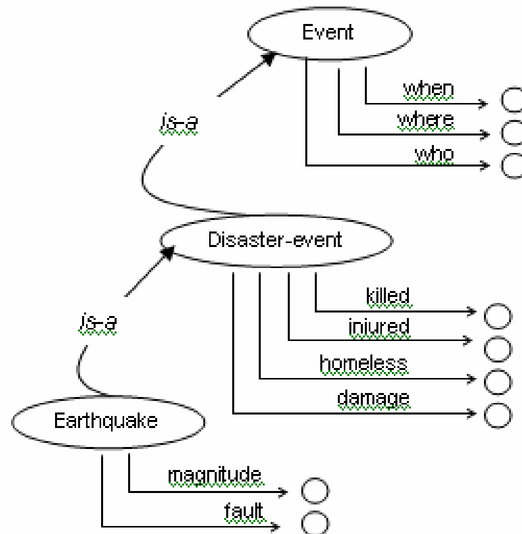

Figure 1.1 An earthquake in a semantic network of news stories

First-Order Predicate Logic

At the same time that frame-based representations were being explored, a popular alternative approach was to use some subset of *first-order predicate logic* (FOL), often implemented as a Prolog program. While propositional logic allows us to make simple statements about concrete entities, a complete first-order logic allows us to make general statements about anonymous elements with the introduction of variables as placeholders. *Table 1.3* contrasts the difference in expressivity between propositional logic and FOL.

Propositional Logic	***First-order Predicate Logic***
All men are mortal. Socrates is a man. Socrates is mortal.	∀x : Man(x) => Mortal(x) Man(Socrates) => Mortal(Socrates)

Table 1.3 Propositional logic vs. first-order predicate logic

In other words, using FOL, you can express general rules of inference that can be applied to all entities whose attributes satisfy the left-hand side of the inference operator. Thus, simply asserting *Man*(Socrates) entails *Mortal*(Socrates).

Since logic programming is based on the tenets of classical logic and comes equipped with automated theorem-proving mechanisms, this approach enabled the development of inference systems whose soundness and completeness could be rigorously demonstrated. But while many of these early inference systems were logically sound

1. Patrick Winston, *Artificial Intelligence* (Massachusetts: Addison-Wesley, 1984).

and complete, they were often not very useful, as they could only be applied to highly proscribed areas, or “toy problems.” The problem was that a complete first-order predicate logic is itself computationally intractable, as certain statements may prove *undecidable*.

Suppose, for example, that we are trying to establish that some theorem, *P(x)*, is true. The way a theorem prover works is to first negate the theorem and, subsequently, to combine the negated theorem ($\neg P(x)$) with stored axioms in the body of knowledge to show that this leads to a logical contradiction. Ultimately, when the theorem prover derives the conclusion that $P(x) \wedge \neg P(x)$ is inconsistent—that it results in the null set—the program terminates and the theorem is considered proven.

This method of proof by refutation is guaranteed to terminate when it is indeed upheld by the body of knowledge. The problems arise when the initial theorem is not valid, as its negation may not produce a logical contradiction, and thus the program may not terminate.

In contrast, frame representations offered a rich, intuitive means of expressing domain knowledge, yet they lacked the inference mechanisms and rigor that predicate logic systems could provide. As suggested by *Figure 1.1* on page 11, the frame representation captures a good deal of implicit knowledge. For example, we expect that all disaster events, including earthquakes, have information about fatalities and injuries and the extent of loss and property damage. In addition, we expect that these events will have locations, dates, and individuals associated with them.

Early efforts to apply predicate logic to frame representations to make information explicit, however, soon revealed that the problem was computationally intractable. This occurred for two reasons: (1) The frame representation was too permissive; more rigorous definitions were required to make the representation computational; and (2) first-order predicate logic itself was computationally intractable.

Several subsets of complete FOL have since been defined and successfully applied to develop useful computational models capable of significant reasoning. For example, the Prolog programming language is based on a subset of FOL that severely limits the use of negation.

The family of *description logic* (DL) systems is a more recent development. Because these systems function as an auto-classifier, they are especially well-suited to the development of ontologies, taxonomies, and controlled vocabularies.

Description Logic

You can view description logic as a combination of the frame-based approach with FOL. Both models had to be scaled back to achieve an effective solution. Like frames, the DL representation allows for concepts and relationships among concepts, including simple taxonomic relations as well as other meaningful types of association. Certain restrictions however, are placed on these relations. Specifically, any relation that involves class membership, such as the *is-a* or *inverse is-a* relations, must be strictly acyclic.

The predicate logic used in a description logic system is also limited in various ways, depending on the implementation. For example, the minimal form of a DL does not allow any form of existential quantification. This limitation allows for a very easily computed solution space, but the resulting expressivity is severely diminished. The

next step up in representational power allows limited existential quantification without atomic negation.

Today there is a large family of description logics that have been realized with varying levels of expressivity and resulting computational complexities. In general, DLs are decidable subsets of FOL, and the decidability is due in large part to their acyclicity. The theory behind these models is beyond the scope of this discussion. For more information, read *The Description Logic Handbook* by Franz Baader, *et al.* (eds.), Cambridge University Press, 2007, ISBN 978-0-521-87625.

The two main ingredients of a DL representation are *concepts* and *roles*. A major distinction between description logics and other subsets of FOL is its emphasis on set expression $Person \cap Young$ can be interpreted as the set of all children, with the corresponding FOL expression $Person(x) \wedge Young(x)$. Syntactically then, DL expressions are variable free, with the understanding that the concepts always reference sets of elements.

A DL *role* is used to indicate a relationship between the two sets of elements referenced by a pair of concepts. In general, DL notations are rather terse, and the concept (or set of elements) of interest is not explicitly represented. Thus, to represent the set of individuals whose children are all female, we would use \/x *hasChild.Female*. The equivalent expression in FOL might be something like this:

$$\backslash/x :hasChild(y.x) \rightarrow female(x)$$

In terms of set theory, a role potentially defines the Cartesian product of the two sets. Roles can have restrictions, however, which place limitations on the possible relations. A *value* restriction limits the type of elements that can participate in the relation; a *number* restriction limits the number of such relations in which an element can participate.

In addition, each role defines a directed relation. For example, if x is the child of y, y is not also the child of x. In the above example, the parent concept *hasChild* is considered the domain of the relation, and the child is considered the range. Elements belonging to the set of objects defined by the range concept are also called *role fillers*. Number restrictions apply to the number of role fillers that are required or allowed in a relation. For example, a parent can be defined as a person having at least one child:

$$Person \cap (child)$$

A DL representation is constructed from a ground set of *atomic concepts* and *atomic roles, w*hich are simply asserted. *Defined concepts* and *defined roles* are then derived from these atomic elements, using the set operations such as intersection, union, and negation. Most DLs also allow existential and universal quantifiers, as in the above examples. Note, however, that these quantifiers always apply to the role fillers only.

The fundamental inference operation in DL is *subsumption*, and is usually indicated with subset notation. Concept A is said to subsume B, or $A \subseteq B$ when all members of concept B are contained in the set of elements defined by concept A, but not vice versa. That is, if B is a proper subset of A, then A subsumes B. This capability has far-reaching repercussions for vocabulary and ontology developers, as it enables the system to automatically classify newly introduced concepts. Moreover, correct subsumption inferencing can be highly nontrivial, as this generally requires examining

all of the relationships defined in the system and the concepts that participate in those relations.

Description Logic in the NCI Thesaurus

The NCI Thesaurus is currently developed using the proprietary Apelon, Inc., Ontylog™ implementation of description logic. Ontylog is distributed as a suite of tools for terminology development, management, and publishing. Although the underlying inference engine of Ontylog is not exposed, the implementation has the characteristics of what is called an AL- (Attributive Language) or FL- (Frame Language) description logic. It does not support atomic negation but does appear to provide all other basic description logic functionality.

The NCI Thesaurus is currently edited and maintained in the Terminology Development Environment (TDE) provided by Apelon. The TDE is an XML-based system that implements the DL model of description logic based on Apelon's Ontylog Data Model. The Data Model uses four basic components: *Concepts*, *Kinds*, *Properties*, and *Roles*. Use of the Apelon TDE for editing and maintenance of the NCI Thesaurus will change with the BioMedGT Wiki and Protégé 1.2 Tool Releases expected in early 2008.

As in other DL systems, concepts correspond to nodes in an acyclic graph, and roles correspond to directed edges defining relations between concept members. Each concept has a unique kind. Formally, kinds are disjoint sets of concepts and represent major subdivisions in the NCI Thesaurus.

More concretely, kinds are used in the role definitions to constrain the domain and range values for that role. Each role is a directed relation that defines a triplet consisting of two concepts and the way in which they are related. The domain defines the concept to which the role applies, and the range defines the possible values—in other words, concepts that can fill that role. For example, the role *geneEncodes* might have its domain restricted to the *Gene_Kind* and its range to the *Protein_Kind*. This role then essentially states that *Genes encode Proteins*.

As in all DLs, all roles are passed from parent to child in the inheritance hierarchy. For example, a *Malignant Breast Neoplasm* has the role *located-in*, connecting it to the concept *Breast*. Thus, since the concept *Breast Ductal Carcinoma* is-a Malignant Breast Neoplasm, it inherits the *located_in* relation to the Breast concept. These lateral non-hierarchical relations among concepts are referred to as *associative* or *semantic* roles, in contrast to the hierarchical relations that reflect the *is-a* roles. In the first-order algebra upon which Ontylog DL is based, every defined relationship also has a defined inverse relationship. For example, if A is contained by B, then B contains A. Inverse relationships are useful and are expected by human users of ontologies. However, they have a computational cost. If the edges connecting concept nodes are bi-directional, then the computation quickly becomes intractable. Therefore in the Ontylog implementation of DL, inverse relationships are not stored explicitly but computed on demand.

Concept Edit History in the NCI Thesaurus

One of the primary uses of the NCI Thesaurus is as a resource for defining tags or retrieval keys for the curation of information artifacts in various NCI repositories. However, since these tags are defined at a fixed point in time, they necessarily reflect the content and structure of the NCI Thesaurus at that time only.

Given the rapidly evolving terminologies associated with cancer research, there is no guarantee that the tags used at the time of curation in the repository will still have the same definition in subsequent releases of the Thesaurus. In most cases the deprecation or redefinition of a previously defined tag is not disastrous, but it may compromise the completeness of the information that can be retrieved.

To address this issue, the EVS team has developed a history mechanism for tracing the evolution of concepts as they are created, merged, modified, split, or retired. (In the NCI Thesaurus, no concept is ever deleted.) The basic idea is that each time an edit action is performed on a concept, a record is added to a history table. This record contains information about relations that held for that concept at the time of the action. It also contains other information such as the version number and time stamp that can be use to reconstruct the state when the action was taken (*Table 1.4*).

Column Name	***Description***
History_ID	Unique consecutive number for use as the database primary key
Concept_Code	Concept code for the concept currently being edited
Action	Edit action: {Create, Modify, Split, Merge, Retire}
Baseline_Date	Date of NCI Thesaurus Baseline (see the following discussion)
Reference_Code	Shows the concept code of a second concept either participating in or affected by the editor’s action. This captures critical information concerning the affect of the edit actions on other concepts. The value will always be null if the action is *Create* or *Modify*.

Table 1.4 Summary of information stored in the history table

Capturing the history data for a *Split*, *Merge*, or *Retire* action is more complicated. In a Split, a concept is redefined by partitioning its defining attributes between two concepts, one of which retains the original concept's code and one that is newly created. This action is taken when ambiguities in the original concept's meaning require clarification by narrowing its definition.

In the case of a Split, three history records are created: one for the newly created concept (with a *null* Reference_Code), and two for the original concept that is being split. In the first of these two records, the Reference_Code is the code for the new concept; in the second, it is the code of the split concept (*Figure 1.2*).

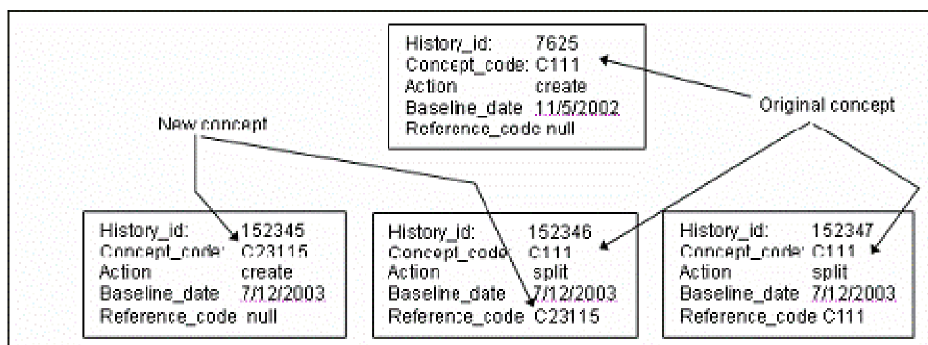

Figure 1.2 History records for the Split action

A Merge action are is similar to a Split. In a Merge, two ambiguous concepts must be combined, and only one of the original concepts is retained. Like a Split action, the Merge action creates three history records: two for the concept that will be retired during the merge, and one for the retained concept. The Reference_Code in the history record for the retained concept is the same as the Concept_Code; that is, the concept points to itself as a descendant in the Merge action. The Reference_Code is null in one of the entries for the retiring concept, while the second entry has the code of the retained concept; thus, this Reference column points to the concept into which the concept in the Concept_Code column is being merged.

Finally, if the action is *Retire*, there are as many history entries as the concept has parent concepts. The Reference column in these entries contains the concept code of the parent concepts—one parent concept per history entry. The motivation for this is that end users with documents coded by such retired concepts may find a suitable replacement among the concept's parents at the time of retirement.

The caCORE EVS APIs support concept history queries.

Working with the NCI Thesaurus

Downloading the NCI Thesaurus

You can download the NCI Thesaurus in several formats, including simple tab-delimited ASCII and OWL (the Web Ontology Language). The files are available for download on the NCICB download site.

The format of the ASCII flat file is extremely simple. For each concept, the download file includes the following information:

- The *concept code*: all terms have the “C” prefix, followed by the integer index;
- The *concept name*: this name may contain embedded punctuation and spaces;
- A pipe-delimited list of parent concepts, as identified in the NCI Thesaurus by *is-a* relations;
- A pipe-delimited list of synonyms, the first of which is the preferred name; and
- One of the NCI definitions for the term, if one exists. Each of these separate types of information is tab-delimited; within a given category, the individual entries are separated by a pipe symbol (|). Only the third and fourth

categories—*i.e.*, the parent concepts and synonyms—have multiple entries requiring the pipe separators.

Note that while much of the information available from the interactive Metaphrase server is included in the download, any information outside the NCI Thesaurus description logic vocabulary (*e.g.*, Diagnosis, Laboratory, Procedures, etc.) is not.

The following example shows the contents of the flat file download for the term *Mercaptopurine*:

```
C6 Mercaptopurine Immunosuppressants|Purine Antagonists

Mercaptopurine|1,3-AZP|1,7-Dihydro-6H-purine-6-thione|
    3H-Purine-6-thiol|6

Thiohypoxanthine|6 Thiopurine|6-MP|6-Mercaptopurine|
    6-Mercaptopurine

Monohydrate|6-Purinethiol|6-Thiopurine|6-Thioxopurine|
    6H-Purine-6-thione,1,7-dihydro- (9CI)|6MP|
    7-Mercapto-1,3,4,6-tetrazaindene|AZA|
    Alti-Mercaptopurine|Azathiopurine|BW 57-323H|CAS

50442|Flocofil|Ismipur|Leukerin|Leupurin|MP|Mercaleukim|
    Mercaleukin|Mercap|Mercaptina|Mercapto-6-purine|
    Mercaptopurinum|Mercapurin|Mern|NCI-C04886|NSC755|
    Puri-Nethol|Purimethol|Purine-6-thiol (8CI)|Purine-6-thiol

Monohydrate|Purine-6-thiol, Monohydrate|Purinethiol|Purinethol|
    U-4748|WR-2785

An anticancer drug that belongs to the family of drugs called
    antimetabolites.
```

If you want to use an encoded format rather than the simple ASCII form, download the OWL encoding of the NCI Thesaurus, which is described in the next section.

OWL Encoding of the NCI Thesaurus

OWL, as specified and proposed by the World Wide Web Consortium (W3C), is an emerging standard for the representation of semantic content on the Web. Building on the earlier groundwork laid by XML, the Resource Description Framework (RDF) and RDF schema, and subsequently by DAML + OIL, OWL represents the culmination of what has been learned from these previous efforts.

While XML provides surface syntax rules and XML schema provide methods for validating a document's structure, neither can impose semantic constraints on how a document is interpreted. RDF provides a data model for specifying objects (resources) and their relations, and RDF schema allow us to associate properties with the individual resources as well as taxonomic relations among the objects. Yet, even these extensions could not provide the breadth and depth of representation needed to encode nontrivial, real-world information. OWL adds vocabulary for describing arbitrary non-hierarchical relations between classes, cardinality constraints, resource equivalences, richer typing of properties, and enumerated classes.

A major focus of the W3C is the establishment of the Semantic Web, which is a far-reaching infrastructure with the purpose of providing a framework whereby autonomous self-documenting agents and Web services can exchange meaningful information without human intervention (see http://www.w3.org/2001/sw/). OWL is the first step towards realizing this vision.

Because of collaborative efforts with Dr. James Hendler and the University of Maryland, the NCI Thesaurus is now available for download in OWL format. This section describes the mapping of the NCI Thesaurus to OWL format, which proceeds via the Ontylog XML elements declared in Apelon's Ontylog DTD. The four basic elements are *kinds*, *concepts*, *roles*, and *properties*, each described in the following list:

- *Kinds* are the top-level superclasses in the Thesaurus. They enumerate the different possible categories of all concepts and include such things as Anatomy, Biological Processes, Chemicals and Drugs. *Each NCI Thesaurus kind is converted to an owl:Class*.
- An NCI Thesaurus concept describes a specific concept under one of the kind categories. *Each NCI Thesaurus concept is converted to an owl:Class*.
- *Roles* capture how concepts relate to one another. Generally, roles have restricted domains and ranges, which limit the sets of concepts that can participate in the role according to their categories (*e.g.*, kinds). The *defining roles* within a concept definition provide these local restrictions on the ranges of roles. *Each NCI Thesaurus role is converted to an owl:ObjectProperty*.
- NCI Thesaurus properties encode the attributes that pertain to a class; they contain metadata that describes the class, but not its instantiations or subclasses. *Each NCI Thesaurus property is converted to an owl:AnnotationProperty*.

Most of the NCI Thesaurus comprises concept definitions. This is also where the most complex semantics occur. Each concept in the Thesaurus has three main types of associated data: *defining concepts*, *defining roles*, and *properties*. A *defining concept* is essentially a superclass; the defined concept in OWL has an *rdfs:subClassOf* relationship to the defining concept.

The defining roles and properties are mapped as described above. The *owl:Annotation-Property* is actually a subclass of r*df:Property*, and, like *rdfs:comment* and *rdfs:label*, can be attached to any class, property, or instance. This allows properties from the Thesaurus to be associated directly with a concept's corresponding class without violating the rules of OWL.

In addition to any explicitly named properties, each element in the Thesaurus also has a uniquely defined *code* and *id* attribute associated with it. These are used as unique identifiers in the Apelon development software and, as such, are not defined explicitly as roles or properties. In mapping these identifying attributes to OWL, we have treated these as special cases of the explicit property elements. Just like other properties in the Thesaurus, they are mapped as *owl:AnnotationProperties*. *Table 1.5* on page 19 summarizes the mapping of elements in the Ontylog DTD to OWL elements.

Ontylog Name Conversion

In mapping to OWL, all Ontylog concept names must be converted to proper RDF identifiers (rdf:id) following the RDF naming rules. This is achieved by removing any spaces in the original names and substituting all illegal characters with underscores. Names that begin with numbers are also prefixed with underscores to make them legal. The original concept name however, is preserved as an rdfs:label.

The process of converting names follows these steps:

1. Replace any plus sign characters (“+”) with the word *plus*.
2. Prefix all role names are prefixed with the letter “r” to ensure that roles and properties with the same name do not clash.
3. Replace any characters that are not alphanumeric with an underscore (“_”).
4. Prefix all names that have leading digits with an underscore.
5. Replace multiple adjacent underscores in the corrected name with a single underscore.

Ontylog Element	***OWL Element***	***Comment***
kindDef	owl:Class	
roleDef	owl:ObjectProperty	
propertyDef	owl:AnnotationProperty	
conceptDef	owl:Class	
name*	rdf:ID	Applies to the name sub-element of kindDef, roleDef, propertyDef, and conceptDef.
name	rdfs:label	Because the conceptDef name contains some useful semantics, the original form is retained as an rdfs:label. No other name elements are retained in rdfs:label.

Table 1.5 Ontylog DTD to OWL Conversions

Ontylog Element	OWL Element	Comment
Code	owl:AnnotationProperty	Defined as an owl:AnnotationProperty with rdf:ID="code". Code values remain the same for each concept.
ID	owl:AnnotationProperty	Defined as an owl:AnnotationProperty with rdf:ID="ID". ID values remain the same for each concept.
definingConcepts	rdfs:subClassOf	The concept supplement of definingConcepts is mapped to the rdf:resource attribute of the rdfs:subClassOf element.
Domain	rdfs:domain	
Range	rdfs:range	
definingRoles / role / name	owl:onProperty	definingRoles are converted to OWL restrictions on properties. The name child element of definingRoles/role is taken as the rdf:resource attribute of the owl:onProperty element.
definingRoles / role / name	owl:someValuesFrom	definingRoles are converted to OWL restrictions on properties. The value child element of definingRoles/role is taken as the rdf:resource attribute of the owl:someValuesFrom element.

Table 1.5 Ontylog DTD to OWL Conversions

Note: Name Ontology elements are converted to rdf:ID as described in the Ontylog Name Conversion section. *namespaceDef* and namespace elements are not mapped to OWL.

Additional information about Ontylog encoding is available in the Ontylog DTD, which can be downloaded from the NCICB EVS FTP site along with the zipped ASCII flat file and the Ontylog XML encoding. The current OWL translation of the NCI Thesaurus contains over 500,000 triples and is available in zipped format from the FTP site, as well as in unzipped format at http://ncicb.nci.nih.gov/xml/owl/EVS/Thesaurus.owl, the mindswap Web site for download or online viewing.

Ontylog Mappings

Mapping of Gene Ontology to Ontylog

The LexBIG Terminology Server provides access to the Gene Ontology™ Consortium's (GO) controlled vocabulary. The GO ontologies are widely used, most likely because of their simple design and their potential for automated transfer of biological annotations—from model organisms to more complex organisms—based on sequence similarities.

GO comprises three independent controlled vocabularies (ontologies) encoding biological process, molecular function, and cellular components for eukaryotic genes. GO terms are connected by two relations, *is-a* and *part-of*, which define a directed acyclic graph. Although concepts in the ontologies were initially derived from only three model systems (yeast, worm, and fruit fly), the goal was to encode concepts in such a way that the information is applicable to all eukaryotic cells. Thus, GO vocabularies do not represent species-specific anatomies, as this would not support a unifying reference for species-divergent nomenclatures.

Each month, the NCI will load the latest version of GO into a test instance of the DTS server and, following validation in the Ontylog environment, will promote it to a production server for programmatic access by NCI applications. The NCI converts GO into the Ontylog XML representation (necessary for import into the DTS server) using a stylesheet transformation followed by some post-processing to satisfy Ontylog constraints. The NCI intends to ensure that the version of GO on the DTS server will not be more than a month behind the current version available from http://www.geneontology.org. However, skipping releases might be necessary if unforeseen complications arise.

Table 1.6 and *Table 1.7* summarize the encoding of GO elements into Ontylog.

Ontology Element	***Instance Name (and optional description)***
namespaceDef	GO
kindDef	GO_Kind
roleDef	part-of This role is unused, but the software required that at least one role be declared.
propertyDef	Preferred_Name
propertyDef	Synonym
propertyDef	DEFINITION
propertyDef	Dbxref Complex property containing two XML GO entitites: *go:database_symbol* and *go:reference*. These entities respectively use the following tags: *database_symbol* and *reference*.

Table 1.6 Ontylog elements used for GO mapping

Ontology Element	Instance Name (and optional description)
propertyDef	part-of Complex property containing two XML GO entities: *go:name* and *go:accession*. These entities respectively use the following tags: *go-term* and *go-id*.

Table 1.6 Ontylog elements used for GO mapping (Continued)

The *go:name* stored in *Preferred_Name* is as declared in GO. However, the *go:name* used in the Ontylog name might have been modified during the conversion process by appending underscores to make the Ontylog name unique.

GO term element	conceptDef element	(propertyDef)
go:accession	Code	
go:name	Name	
go:isa	definingConcepts	
go:name	Property	Preferred_Name
go:synonym	Property	Synonym
go:definition	Property	DEFINITION
go:part-of	Property	part-of
go:dbxref	Property	dbxref

Table 1.7 Mapping of GO term to Ontylog conceptDef

Mapping of MedDRA to Ontylog

Vocabulary Hierarchy Structure

The Ontylog version of MedDRA reflects the native hierarchy with terms organized according to their term type, as shown in *Figure 1.3*.

SOC (System Organ Class)

|__ HLGT (High Level Group Term)

|__ HLT (High Level Term)

|__ PT (Preferred Term)

|__LLT (Lowest Level Term)

Figure 1.3 Hierarchy of MedDRA

The Special Search Categories (SSC) are under the concept *AssociativeTerm-Group(SSCs)*, which the EVS created as a header concept for the SSC terms to be grouped together. All the System Organ Class (SOC) concepts, as well as the top header concept for the SSCs, are under the *MedDRA[V-MDR]* root node. Although Low Level Terms (LLTs) can have any type of relationship to their Preferred Term (PT) (for example, a synonym of the PT), the Ontylog version presents them all as children concepts. The *Associative Term Group (SSCs)* concept has a special code and term type not found in MedDRA to distinguish it from other terms in the vocabulary.

Concept Codes and Names

The concept name is created from the MedDRA term followed by the MedDRA code enclosed in brackets. The Ontylog concept name must be unique so that including the code in the name guarantees uniqueness. For display purposes, the property *Preferred_Name* should be used instead of the concept name; it contains the unadorned MedDRA term. The Ontylog concept code is the MedDRA code.

Roles

A single role has been defined. The role *has_associated_term* is used to relate SSC top-level categories with their associated PT terms. All the concepts in the vocabulary are primitive.

Properties

Table 1.8 shows the properties defined for MedDRA 6 in Ontylog. It also indicates their provenance in the MedDRA distribution. Properties that are not derived directly from MedDRA show a dash in the **MedDRA Entity** column.

Ontology Property	***MedDRA Entity***
Code_in_Source	MedDRA code (llt_code, pt_code, hlt_code, etc.)
Cross-reference, cross reference to WHOART	COSTART, ICD9-CM, and so forth
Descriptor_ID	pt_code of an LLT
MedDRA_Abbreviation	soc_abbrev, spec_abbrev
NCI_META_CUI	-
Preferred_Name	MedDRA name (llt_name, pt_name, hlt_name, and so forth)
Primary_SOC	pt_soc_code of a PT
Serial_Code_International_SOC_Sort_Order	intl_ord_code
Term_Type	-
UMLS_CUI	-

Table 1.8 Properties defined in the Ontylog version of MedDRA

Of the MedDRA-derived properties, only *Cross-reference* is not a straightforward name-value pair. This property has subfields encoded in XML. The XML elements are *source* and *source code*, where the source code contains a code or symbol assigned by an external vocabulary source to a specific term.

Two properties are *not* derived from MedDRA: *NCI_META_CUI* and *UMLS_CUI*. These properties contain the Concept Unique Identifier (CUI) of concepts in the NCI Metathesaurus containing MedDRA terms. The property name indicates whether the CUI is assigned to the concept by Unified Medical Language System (UMLS) or by the NCI. The *Term_Type* property is indirectly derived from MedDRA and indicates the hierarchy level of a term with the term types as shown in *Figure 1.3*. In addition, the term type for Obsolete Lower Level Terms (OLLT) is also used.

Mapping of MGED Ontology to Ontylog

The native MGED Ontology (MO) is edited in OilEd and distributed in the Defense Advanced Research Projects Agency (DARPA) Agent Markup Language (DAML) + Ontology Inference Layer (OIL) XML format. DAML+OIL can be converted to the Ontylog Description Logic (DL) in a relatively straightforward manner. However, some valid DAML+OIL constructions cannot be represented in Ontylog DL, including enumerations and specific combinations of ObjectProperties that result in classification cycles in Ontylog.

In MO version 1.1.9, two ObjectProperties have been asserted near the top of the hierarchy on the MGEDCoreOntology class. On conversion to Ontylog, these assertions generate classification cycles; however, the data cannot be massaged as was done in preliminary conversions with previous versions of MO because the fix would have required modifications to every converted concept. Consequently, beginning with MO v 1.1.9, all the ObjectProperties in DAML+OIL are converted to Ontylog properties (rather than Ontylog roles), which are annotations ignored by the classifier.

Vocabulary Hierarchy Structure

The Ontylog conversion preserves the MO class hierarchy structure. One minor difference is that the Ontylog concept hierarchy also represents MO class instances, since there is no distinction between classes and instances in Ontylog. A non-MO, top-level concept, OrphanConcepts has been added in the Ontylog representation to hold MO instances of *Thing*.

Concept IDs, Codes, and Names

MO classes and instances are identified solely by their name; no codes or numeric IDs are assigned. For the conversion to Ontylog, MO class or instance names are retained as concept names.

As Ontylog concepts also require unique codes and IDs, a code and an ID are created during the conversion. The ID reflects the position of the class or instance in the XML tree. The code is derived from the ID by adding the prefix *X-MO-*; therefore, the code is not guaranteed to remain invariant from version to version of the MO. A mapping table is available whenever the MO is updated.

Roles

No roles have been defined. All concepts are primitive.

Properties

All the object and datatype properties defined in MO have been converted to Ontylog properties. With the exception of *has_reason_for_deprecation* and *has_database*, all the properties have been manually propagated to children concepts in the database to mimic the expected role inheritance. In addition, new properties have been defined as shown in *Table 1.9*.

Ontylog Property	MGED Ontology Entity
DEFINITION	rdfs:comment value
Preferred_Name	rdf:about value
Synonym	rdf:about value
Concept_Type	-

Table 1.9 New properties defined in the Ontylog version of the MGED ontology

The *Preferred_Name* property is recommended for display purposes, while *Synonym* is recommended for searches by dependent applications. (Even though the value of both properties is the same, the EVS tries to maintain a certain consistency in the usage of properties for the benefit of all users.) The *Concept_Type* property holds one of two values: *mged_class*, or *mged_instance*.

About LexBIG

caCORE EVS is the adopter site for the open source, public domain terminology server *LexBIG*, developed by the Mayo Clinic as part of the caBIG Program. The goal of caCORE EVS is to adopt LexBIG as the sole terminology server infrastructure for the EVS.

The Apelon DTS server is a proprietary server that does not allow exposure of the API. As a result, caCORE EVS 3.2 and earlier versions have provided a custom API that communicates directly with the DTS Server and is publicly available. The caCORE EVS 4.0 release begins the transition to LexBIG by re-exposing the caCORE EVS 3.2 custom API with LexBIG as the back-end terminology server. Additionally, the caCORE EVS 3.2 API will continue to be supported for approximately one year. This gives users who have implemented the caCORE EVS 3.2 API time to plan for the retirement of the Apelon DTS server by the NCI EVS team.

LexBIG is based on the LexGrid Model, Mayo's proposal for standard storage of controlled vocabularies and ontologies. The LexGrid Model defines how vocabularies should be formatted and represented programmatically, and it is intended to be flexible enough to accurately represent a wide variety of vocabularies and other lexically based resources. The model also defines several different server storage mechanisms such as relational databases, LDAP, and XML format.

The LexGrid Model provides the core representation for all data managed and retrieved through the LexBIG system. It is now rich enough to represent vocabularies provided in numerous source formats, including Open Biomedical Ontologies (OBO), Web Ontology Language (OWL), and the Unified Medical Language System (UMLS) Rich Release Format (RRF). This common model is a critical component of the LexGrid project. Once disparate vocabulary information can be represented in a standardized model, users can build common repositories to store vocabulary content and common programming interfaces and tools to access and manipulate that content.

LexBIG has three major components:

- Service management tools to load, index, and manage vocabulary content for the vocabulary server;
- an API providing Java interfaces to various functions, including lexical queries, graph representation and traversal, and NCI change event history; and
- a Graphical User Interface providing access to service management and API functions.

The LexBIG API enables querying information stored in the LexGrid model. Similar APIs have been developed for LexBio that are used at the National Center for Bioontologies (NCBO). The NCI EVS has adopted and modified the NCBO's BioPortal as a Web browser for LexBIG. You can access the NCI BioPortal at http://bioportal.nci.nih.gov.

To summarize, LexBIG provides the following features:

- A robust, scalable, open source implementation of EVS-compliant vocabulary services. The API specification will be based on, but not limited to, fulfillment of the caCORE EVS API. The specification will be further refined to accommodate changes and requirements based on prioritized needs of the caBIG™ community.
- A flexible implementation for vocabulary storage and persistence, allowing for alternative mechanisms without affecting client applications or end users. Initial development will focus on delivery of open source, freely available solutions, though this does not preclude the ability to introduce commercial solutions such as Oracle.
- A standard tool for loading and distributing vocabulary content. This includes, but is not limited to, support of standardized representations such as UMLS Rich Release Format (RRF), OWL, and Open Biomedical Ontologies (OBO).

CHAPTER 2
ABOUT CACORE

This chapter provides an overview of the NCICB caCORE infrastructure.

Topics in this chapter:

- *Architecture Overview* on this page
- *Components of caCORE* on page 28

Architecture Overview

The NCI Center for Bioinformatics (NCICB) provides biomedical informatics support and integration capabilities to the cancer research community. The NCICB has created a core infrastructure called Cancer Common Ontologic Representation Environment (caCORE), a data management framework designed for researchers who need to be able to navigate through a large number of data sources.

By providing a common data management framework, caCORE helps streamline the informatics development throughout academic, government, and private research labs and clinics. The components of caCORE support the semantic consistency, clarity, and comparability of biomedical research data and information. caCORE is open-source enterprise architecture for NCI-supported research information systems, built using formal techniques from the software engineering and computer science communities.

The four characteristics of caCORE include

- Model Driven Architecture (MDA)
- n-tier architecture with open Application Programming Interfaces (APIs)
- Use of controlled vocabularies, wherever possible
- Registered metadata

The use of MDA and n-tier architecture, both standard software engineering practices, enables easy access to data by other applications. The use of controlled vocabularies

and registered metadata, less common in conventional software practices, requires specialized tools that are generally unavailable.

As a result, the NCICB (in cooperation with the NCI Office of Communications) has developed the Enterprise Vocabulary Services (EVS) system to supply controlled vocabularies, and the Cancer Data Standards Repository (caDSR) to provide a dynamic metadata registry.

When a system meets all four development characteristics listed earlier in this section, it is said to be “caCORE-like.” Such systems provide several advantages:

- With its open APIs, the n-tier architecture frees the end user (whether human or machine) from having to understand the implementation details of the underlying data system to retrieve information.
- The maintainer of the resource can move the data or change implementation details without affecting the ability of remote systems to access the data.
- Most importantly, the system is *semantically interoperable*; it uses runtime-retrievable information to provide an explicit definition and complete data characteristics for each object and attribute that can be supplied by the data system.

Components of caCORE

The components that comprise caCORE are EVS, caDSR, caBIO, CSM, and CLM. The following subsections provide a brief description of each component.

Enterprise Vocabulary Services (EVS)

EVS provides controlled vocabulary resources that support the life sciences domain, implemented in a description logics framework. EVS vocabularies provide the semantic raw material from which data elements, classes, and objects are constructed.

Cancer Data Standards Repository (caDSR)

The caDSR is a metadata registry based upon the ISO/IEC 11179 standard. It is used to register the descriptive information needed to render cancer research data reusable and interoperable. The caBIO, EVS, and caDSR data classes are registered in the caDSR, as are the data elements on NCI-sponsored clinical trials case report forms.

Cancer Bioinformatics Infrastructure Objects (caBIO)

The caBIO model and architecture is the primary programmatic interface to caCORE. Each of the caBIO domain objects represents an entity found in biomedical research.

Unified Modeling Language™ (UML) models of biomedical objects are implemented in Java as middleware connected to various cancer research databases to facilitate data integration and consistent representation. Examining the relationships between these objects can reveal biomedical knowledge that was previously buried in various primary data sources.

Common Security Model (CSM)

The Common Security Model (CSM) provides a flexible solution for application security and access control with three main functions:

- Authentication to validate and verify a user's credentials
- Authorization to grant or deny access to data, methods, and objects
- User Authorization Provisioning to allow an administrator to create and assign authorization roles and privileges.

Common Logging Module (CLM)

The Common Logging Module (CLM) provides a separate service under caCORE for audit and logging capabilities. It also comes with a Web-based locator tool.

Client applications can use the CLM directly, without the application using any other components such as the CSM.

About the caCORE 4.0 Release Paradigm

In September 2007, the NCICB Infrastructure and Product Management Team made the decision to separate the caCORE Components that had previously been bundled and released together. This decision was geared toward allowing each of the infrastructure product teams to be more responsive in addressing specific needs of the user community. caCORE EVS is the first component to release under the new release paradigm. With each component release there is a product-specific technical guide.

This guide focuses on the EVS component of caCORE 4.1, a minor release addressing defects and feature requests. For more details on the other components, refer to the caCORE Overview page at http://ncicb.nci.nih.gov/NCICB/infrastructure/cacore_overview, which directs you to other product-specific technical guides.

CHAPTER 3
ABOUT THE CACORE EVS ARCHITECTURE

This chapter describes the architecture of the caCORE EVS. It includes information about the client-server communication and describes the system software packages.

Topics in this chapter:

- *caCORE EVS System Architecture* on this page
- *Client Technologies* on page 33
- *caCORE EVS Software Packages* on page 34

caCORE EVS System Architecture

The caCORE EVS infrastructure exhibits an *n-tiered* architecture with client interfaces, server components, back-end objects, and additional back-end systems (*Figure 3.1*). This n-tiered system divides tasks or requests among different servers and data stores. This isolates the client from the details of where and how data is retrieved from the LexBIG terminology server.

Clients such as browsers and applications receive information from back-end objects. Java applications also communicate with back-end objects via domain objects packaged within the EVS Client Archive (JAR file). Non-Java applications can communicate via SOAP (Simple Object Access Protocol). Back-end objects communicate directly with the LexBIG API.

Most of the caCORE EVS infrastructure is written in the Java programming language and leverages reusable, third-party components. *Figure 3.1* (page 32) illustrates the architectural overview.

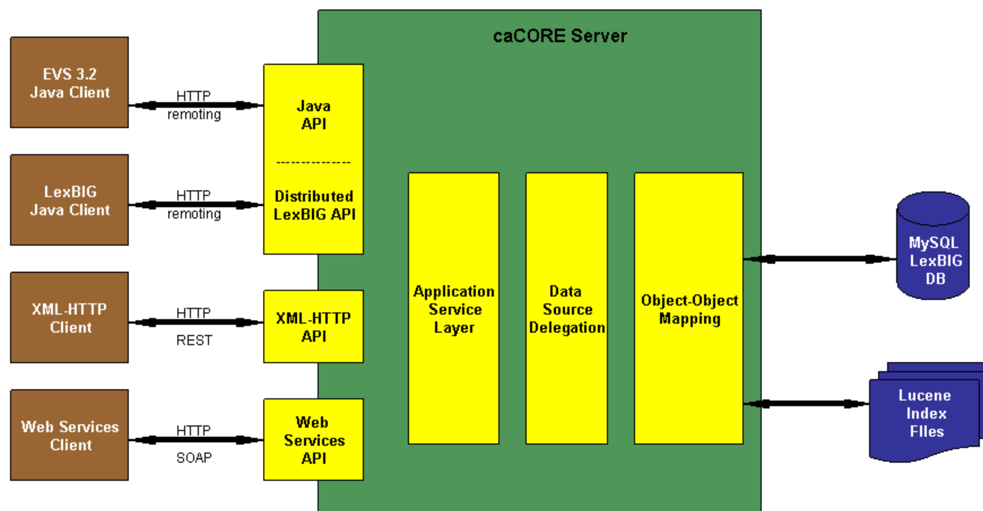

Figure 3.1 caCORE EVS architecture

The caCORE EVS infrastructure is composed of the layers described in *Table 3.1*.

Layer	***Description***
Application Service	Consolidates incoming requests from the various interfaces and translates them to native query requests that are then passed to the data layers. All interfaces provide full, anonymous, read-only access to all data.
Data Source Delegation	Conveys each query that it receives to the respective LexBIG Service objects that can perform the query.
Object-Object Mapping (OOM)	Following the façade design pattern, provides an additional interface layer that enables access to a large number of modules and functions in the caCORE EVS system.
LexBIG Service API	Comprises four primary subsystems: • **Service Management** provides administration control for loading a vocabulary and activating a service. • **Service Metadata** provides external clients with information about the vocabulary content (the NCI Thesaurus) and appropriate licensing information. • **Query Operations** provide numerous functions for querying and traversing vocabulary content. • **Extensions** provides a mechanism to extend the specific service functions, such as Loaders, or re-wrap specific query operations into convenience methods. The LexBIG Service is designed to run standalone or as part of a larger network of services. For more information refer to the LexBIG Programmer's Guide.

Table 3.1 caCORE infrastructure layers

Client Technologies

Applications that use the Java programming language can access EVS directly through the domain objects provided by the EVS Client Archive (see *Chapter 4*). The network details of the communication to the caCORE EVS server are transparent to developers, preventing them from having to deal with issues such as network and database communication and enabling them to concentrate on the biological problem domain.

The caCORE EVS system also enables non-Java applications to use SOAP clients to interface with caCORE EVS Web services. SOAP is a lightweight, XML-based protocol for the exchange of information in a decentralized, distributed environment. It consists of an envelope that describes the message and a framework for message transport. caCORE EVS uses the open source Apache Axis package to provide SOAP-based Web services to users. This allows other languages such as Python or Perl to communicate with caCORE EVS objects in a straightforward manner.

The caCORE EVS architecture includes a presentation layer that uses the technologies described in *Table 3.2*.

Technology	***Description***
J2SE application server	Tomcat or JBoss
Java Server Pages (JSPs)	Web pages that have Java embedded in the HTML and can incorporate dynamic content in the page.
Java servlets	Server-side Java programs that Web servers can run to generate content in response to client requests.
JavaBeans	Reusable software components that work with Java.

Table 3.2 caCORE EVS presentation layer technologies

Communication between the client interfaces and the server components occurs over the Internet using the HTTP protocol. The server components are deployed in a Web application container as a Web archive (`.war`) file that communicates with the LexBIG terminology server.

caCORE EVS Software Packages

The caCORE EVS software comprises several domain and system packages. The following subsections discuss both package categories.

Domain Packages

Table 3.3 describes several Java domain packages that are a significant part of the caCORE EVS software.

Package	Description
caCORE EVS domain (gov.nih.nci.evs.domain)	Provides access to the Java 3.2 interfaces and classes such as `DescLogicConcept` and `MetaThesaurusConcept` (*Figure 3.2*). For a complete list of domain objects, see the EVS 3.2 Object Model.
query (gov.nih.nci.evs.query)	Contains classes that facilitate a custom query mechanism for EVS domain objects. This package is discussed in *Chapter 4, Interacting with caCORE EVS,* on page 37.
lexbig (gov.nih.nci.lexbig)	Contains the Distributed LexBIG (DLB) Adapter classes described in *Table 3.4* on page 35.

Table 3.3 caCORE EVS domain packages

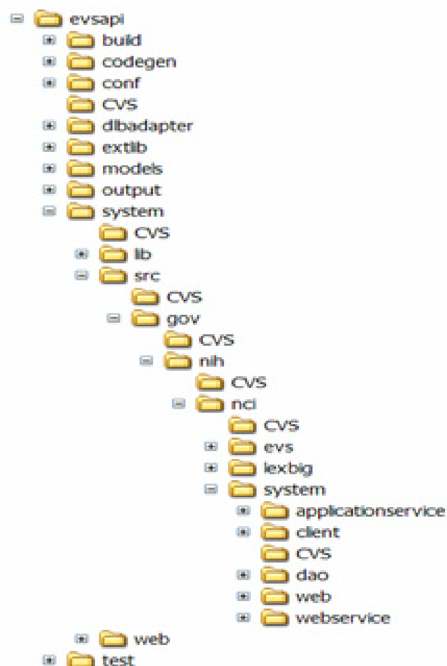

Figure 3.2 caCORE EVS packages

System Packages

In addition to domain packages, the caCORE EVS API specification includes several system packages. A system package contains several subpackages, including

- a `lib` folder that holds the third-party libraries required to deploy the system, and
- a `src` folder that contains the bulk of the EVS system code; the subpackages in this folder include the `EVSQuery` classes, the Distributed LexBIG Adapter (DLBAdapter) classes, and the `system` package.

The `system` package includes the following types of subpackage:

- application service package (described in *Client Technologies* on page 33)
- data access package
- delegate/service locator package
- proxy package
- Web service package.

Table 3.4 describes several key system packages.

Package	***Description***
EVS Query (gov.nih.nci.evs.query)	Contains the `evsQuery` and `evsQueryImpl` Java interface and class.
DLBAdapter (gov.nih.nci.lexbig.ext)	Contains the `DLBAdapter` classes. These convenience methods are used to supplement access to the Distributed LexBIG API (described in *Chapter 4*).
Data Access (gov.nih.nci.system.dao)	Layer at which the query is parsed from objects to the native query, the query is executed, and the result sets are converted back to domain objects results. This layer has implementation for external data access layer for querying other subsystems. It also contains the security objects required to support the controlled access requirements to the MedDRA data source.
Proxy Interface (gov.nih.nci.system.client.proxy)	Serves as the gateway for requests from Java and platform-independent Web service clients.
Web Service (gov.nih.nci.system.webservice)	Contains the Web service wrapper class that uses Apache's Axis.
Web (gov.nih.nci.system.web)	Contains useful utilities.

Table 3.4 caCORE EVS system packages

CHAPTER 4

INTERACTING WITH CACORE EVS

This chapter describes the components of the caCORE EVS and the service interface layer provided by the EVS API architecture. It gives examples of how to use the EVS APIs. It also describes the Distributed LexBIG API and the Distributed LexBIG Adapter.

Topics in this chapter:

- *caCORE EVS Components* on page 38
- *EVS 3.2 Object Model* on page 39
- *EVS 3.2 Domain Object Catalog* on page 40
- *EVS Data Sources* on page 41
- *Installing and Configuring the EVS 3.2 Java API* on page 41
- *Search Paradigm* on page 47
- *About EVSQuery and EVSQueryImpl* on page 48
- *Web Services API* on page 55
- *XML-HTTP API* on page 62
- *About the XMLUtility Class* on page 65
- *Distributed LexBIG API* on page 67
- *Distributed LexBIG Adapter Example of Use* on page 71

caCORE EVS Components

The caCORE EVS API is a public domain, open source wrapper that provides full access to the LexBIG Terminology Server. LexBIG hosts the NCI Thesaurus, the NCI Metathesaurus, and several other vocabularies. Java clients accessing the NCI Thesaurus and Metathesaurus vocabularies communicate their requests via the open source caCORE EVS APIs, as shown in *Figure 4.1*.

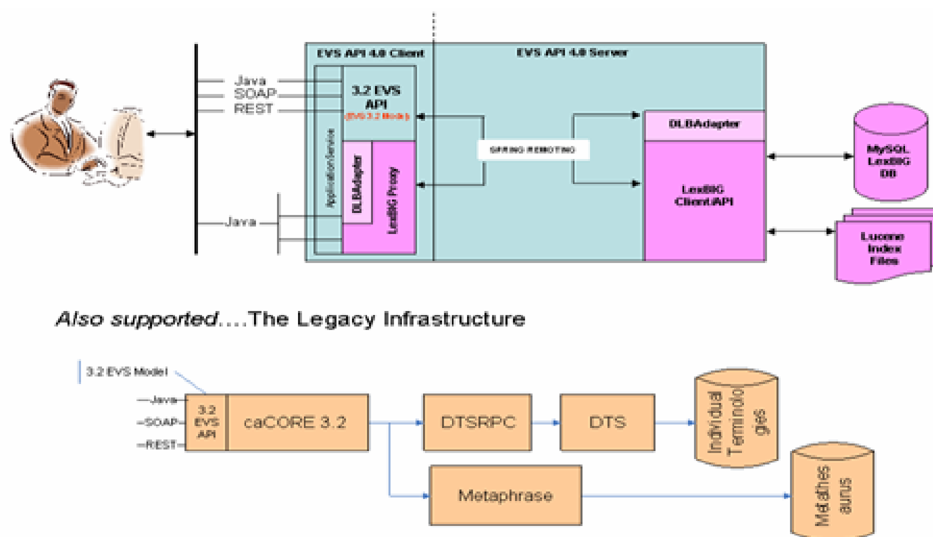

Figure 4.1 Overview of the caCORE EVS 4.0 release components

The open source interfaces provided as part of caCORE EVS 4.x include Java APIs, a SOAP interface, and an HTTP REST interface. The Java APIs are based on the EVS 3.2 object model and the LexBIG Service object model.

The EVS 3.2 model, exposed as part of caCORE 3.2, has been re-released with LexBIG as the back-end terminology service in place of the proprietary Apelon DTS back end. The SOAP and HTTP REST interfaces are also based on the 3.2 object model. The SDK 4.0 was used to generate the EVS 3.2 Java API, as well as the SOAP and HTTP REST interfaces.

The only difference between the EVS 3.2 API exposed as part of the caCORE EVS 4.x and the API exposed as part of caCORE 3.2 is the back-end terminology server used to retrieve the vocabulary data. The interface (API calls) are the same and should only require minor adjustments to user applications.

Note: You cannot integrate caCORE 3.2 components with caCORE EVS 4.x. If you used multiple components of caCORE 3.2 (for example, EVS with caDSR), you need to continue to work with the caCORE 3.2 release until the other caCORE 4.0 components are available.

The LexBIG object model was developed by the Mayo Clinic. In its native form, the associated API assumes a local, non-distributed means of access. With caCORE EVS 4.x, a proxy layer enables EVS API clients to access the native LexBIG API from anywhere without having to worry about the underlying data sources. This is called the Distributed LexBIG (DLB) API.

The DLB Adapter is another option for caCORE EVS 4.x clients who choose to interface directly with the LexBIG API. This is essentially a set of convenience methods intended to simplify the use of the LexBIG API. For example, a series of method calls against the DLB API might equate to a single method call to the DLB Adapter.

Note: The DLB Adapter is not intended to represent a complete set of convenience methods. As part of the caCORE EVS 4.x release, the intention is that users will work with the DLB API and suggest useful methods of convenience to the EVS Development Team.

EVS 3.2 Object Model

The EVS 3.2 Java API is based on the EVS 3.2 object model. The UML Class diagram (object model) in *Figure 4.2* provides an overview of the EVS 3.2 domain object classes. The `DescLogicConcept` and `MetaThesaurusConcept` are two central Concept classes in the model, with most of the other classes organized around these entities. The `Vocabulary` and `SecurityToken` classes were added as part of the caCORE 3.2 release. The `SecurityToken` class can be used to specify security credentials such as username, password, and security token.

The DAO Security model provides data level security to vocabularies. The `MedDRASecurity` class, which implements the `DAOSecurity` interface, validates a token against the MedDRA vocabulary and prevents unauthorized users from performing any queries against MedDRA. To access MedDRA using the EVS 3.2 Java API, users must obtain a valid token from the NCICB.

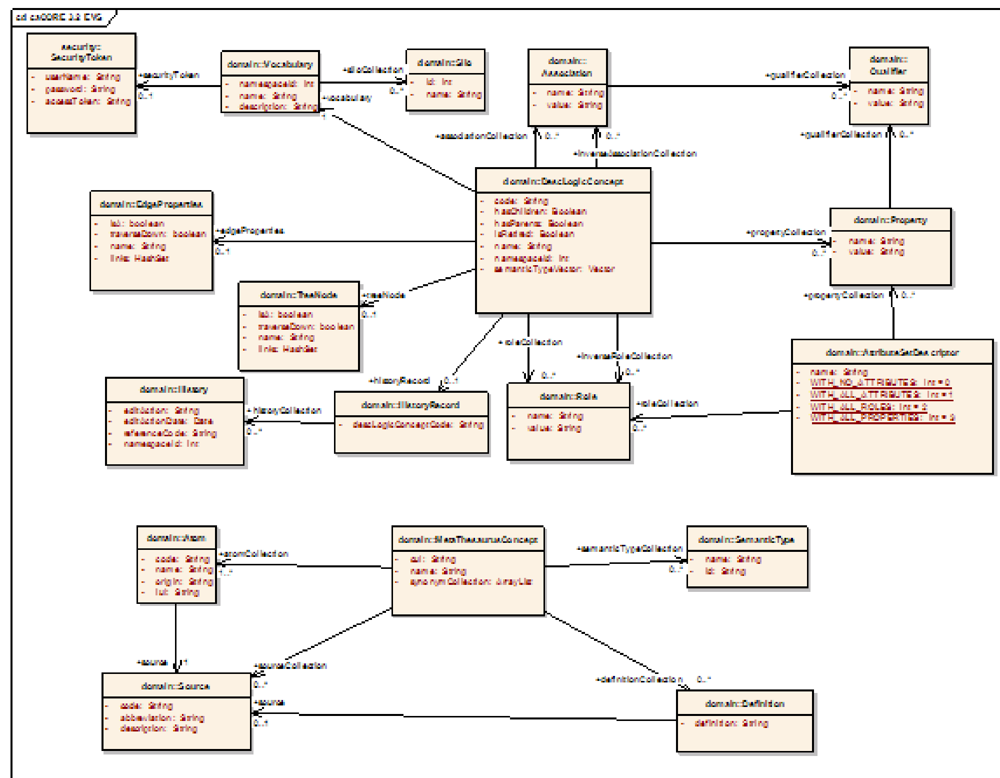

Figure 4.2 caCORE EVS 3.2 Java API domain object classes

Note: Since the EVS API is generated using the SDK, note that the EVS API diverges somewhat from the other caCORE domain models (caDSR and caBIO) in its search mechanisms. While the other APIs have direct access to their databases, the EVS API does not. Since all EVS queries are passed through the LexBIG APIs, the search and retrieval capabilities are effectively proscribed by the features implemented by the open terminology server.

EVS 3.2 Domain Object Catalog

The caCORE EVS domain objects are implemented as JavaBeans in the `gov.nih.nci.evs.domain` package. The only interface implemented by the EVS domain objects is `java.io.serializable`.

Table 4.1 lists and describes each class. For more detailed descriptions about each class and its methods, see the caCORE EVS 4.1 JavaDocs.

EVS Domain Object	***Description***
Association	Relates a concept or a term to another concept or term. Association has three categories: • concept association • term association • synonyms (concept-term associations)
Atom	An occurrence of a term in a source.
`AttributeSetDescriptor`	Set of concept attributes that should be retrieved by a given operation.
Definition	Textual definition from an identified source.
`DescLogicConcept`	Fundamental vocabulary entity in the NCI Thesaurus.
`EdgeProperties`	Specifies the relationship between a concept and its immediate parent when a DefaultMutableTree is generated using the `getTree` method.
`EditActionDate`	Stores edit action and date information. This class is deprecated and will be removed from a future release. Use the History class instead.
History	Stores concept history information.
`HistoryRecord`	Stores the `DescriptionLogicConcept` code.
`MetaThesaurusConcept`	Fundamental vocabulary entity in the NCI Metathesaurus.
Property	Attribute of a concept. Examples of properties are *Synonym*, *Preferred_Name*, and *Semantic_Type*.
Qualifier	Attached to associations and properties of a concept.
Role	Entity that defines a relationship between two concepts.
`SemanticType`	Category defined in the semantic network. A semantic type can be used to group similar concepts.

Table 4.1 caCORE EVS domain objects and descriptions

EVS Domain Object	Description
Silo	A repository of customized concept terminology data from a knowledge base. Single silo or multiple silos can exist, with each silo consisting of semantically related concepts and extracted character strings associated with those concepts.
`SecurityToken`	Stores security information for a vocabulary.
Source	A knowledge base.
`TreeNode`	Specifies the relationship between a concept and its immediate parent when a `DefaultMutableTree` is generated using the `getTree` method. This class is deprecated and will be removed from a future release. Use `EdgeProperties` instead.
Vocabulary	Vocabulary entity or namespace.

Table 4.1 caCORE EVS domain objects and descriptions (Continued)

EVS Data Sources

The EVS data source is the open source LexBIG terminology server. EVS clients interface with the LexBIG API to retrieve desired vocabulary data. The EVS provides the NCI with services and resources for controlled biomedical vocabularies, including the NCI Thesaurus and the NCI Metathesaurus.

NCI Thesaurus

The NCI Thesaurus is composed of over 27,000 concepts represented by about 78,000 terms. The Thesaurus is organized into 18 hierarchical trees covering areas such as Neoplasms, Drugs, Anatomy, Genes, Proteins, and Techniques. These terms are deployed by the NCI in its automated systems for uses such as key wording and database coding.

NCI Metathesaurus

The NCI Metathesaurus maps terms from one standard vocabulary to another, facilitating collaboration, data sharing, and data pooling for clinical trials and scientific databases. The Metathesaurus is based on the Unified Medical Language System (UMLS) developed by the National Library of Medicine (NLM). It is composed of over 70 biomedical vocabularies.

Installing and Configuring the EVS 3.2 Java API

The EVS 3.2 Java API bundled with the caCORE EVS 4.x release provides direct access to domain objects and all service methods. Because caCORE EVS is natively built in Java, this API provides the fullest set of features and capabilities.

The EVS API home page provides a user interface (UI) to the Java API. The interface is available at http://evsapi.nci.nih.gov/evsapi40.

Note: The caCORE 3.2 release also provides an EVS 3.2 Java client API. The difference between the 3.2 and the 4.x clients is the back-end terminology server. caCORE 3.2 uses the proprietary Apelon DTS and caCORE EVS 4.x uses LexBIG. The API is the same and should only require minor updates to a client application wanting to migrate to the EVS 3.2 Java API provided with caCORE EVS 4.x.

Software Requirements

The caCORE EVS Java 3.2 API uses the following software on the client machine (*Table 4.2*).

Software	***Version***	***Required?***
Java 2 Platform Standard Edition Software 5.0 Development Kit (JDK 5.0)	1.5.0 or higher	Yes
Apache Ant	1.6.5 or higher	Yes

Table 4.2 caCORE EVS Java API client software

Downloading the Java API Client Package

The client package is available on the NCICB Web site. To download it, follow these steps:

1. Using your browser, go to http://ncicb.nci.nih.gov.
2. On the left navigation bar of the NCICB welcome page, click the **more** link to the right of the DOWNLOADS category (*Figure 4.3*).

More link

Figure 4.3 Downloads section of the NCICB Web site

3. When prompted, enter your name, e-mail address, and institution name.
4. Click **Enter the Download Area**.
5. Read and accept the license agreement.
6. On the caCORE EVS downloads page, download the EVS Zip file from the Primary Distribution section.

Installing the Package

After downloading the caCORE EVS client package, extract the contents of the downloadable archive to a directory on your hard drive (for example, `c:\evsapi` on Windows or `/usr/local/evsapi` on Linux).

The extraction includes the directories and files listed in *Table 4.3*.

Directory	*Files*	*Description*	*Component*
./build	build.xml	Ant build file	Build file
./conf	application-config-client.xml		
	xml-mapping.xml		
	log4j.properties.xml	Logging utilities configuration properties	
	*.xsd		
./lib	spring.jar	Spring framework	HTTP remoting
	acegi-security-1.0.4.jar	Spring Security	
	asm.jar		
	antlr-2.7.6.jar	Apache Ant	
	log4j-1.2.14.jar	Logging utilities	Logging
	commons-logging-1.1.jar		
	commons-codec-1.3.jar		
	commons-collections-3.2.jar		
	commons-pool-1.3.jar		
	evsapi40-beans.jar	EVS API beans	Domain classes
	evsapi40-framework.jar	caCORE EVS framework	
	lb*.jar	LexBIG classes	
	lg*.jar	LexGRID classes	LexBIG
	lucene*.jar	Index search	LexBIG

Table 4.3 Extracted directories and files in caCORE EVS client package

Directory	Files	Description	Component
	castor-1.0.2.jar	Castor serializer/ deserializer	XML conversion
	xercesImpl.jar	Apache Xerces XML parser	
	sdk-client-framework.jar	XML schemas for objects	
./src	TestClient.java	Java API client samples (for local, remote, and Web service clients)	Sample code
	TestEVS.java	Java API EVS client sample	
	TestFilterRM.java		
	TestGetXMLClient .java	XML utility sample	
	TestServerConfig.java		
	TestServerConfigLoader .java		
	TextXMLClient.java		

Table 4.3 Extracted directories and files in caCORE EVS client package (Continued)

The following files are required to use the Java API:

- all of the files in the `conf` directory, and
- all of the `jar` files in the `lib` directory of the caCORE EVS client package.

When building applications, include these files in the Java classpath. The included `build.xml` file demonstrates how to do this when using Ant for command-line builds. If you are using an integrated development environment (IDE) such as Eclipse, refer to the tool's documentation for information on how to set the classpath.

Verifying the Installation

After installing the caCORE EVS client, follow these steps to run the simple example program:

1. Open a command prompt or terminal window from the directory where you extracted the downloaded archive.
2. Type this command: `ant`.

   This command displays a list of Ant targets that identify the test execution options, as shown in the following code example:

```
 1  Buildfile: build.xml
 2
 3  help:
 4     [echo] ================================
 5     [echo] caCORE EVS API - HELP
 6     [echo]
 7     [echo] To run the test programs, use the following commands:
 8     [echo]
 9     [echo] 1. TestClient - ant run
10     [echo] 2. TestEVS    - ant runevs
11     [echo] 3. TestXML    - ant runxml
12     [echo]
13
14  BUILD SUCCESSFUL
15  Total time: 0 seconds
```

3. Enter the desired Ant command to compile and run the associated test class.

   The following is a short segment of code from the `TestClient` class with an explanation of its functioning. Successfully running similar code indicates that you have properly installed and configured the caCORE EVS client.

```
 1  EVSApplicationService appService =
 2     (EVSApplicationService)ApplicationServiceProvider
          .getApplicationService("EvsServiceInfo");
 3
 4  try
 5  {
 6     DescLogicConcept dlc = new DescLogicConcept();
 7     dlc.setName("ear*");
 8     Collection results =
          appService.search("gov.nih.nci.evs.domain
          .DescLogicConcept", dlc);
 9     System.out.println("Results: "+ results.size());
10
11     for(Object o : results)
12     {
13        DescLogicConcept obj = (DescLogicConcept)o;
14        System.out.println("Concept name : "+ obj.getName()
             +"\t"+ obj.getCode());
15        Vector propList = (Vector)obj.getPropertyCollection();
16        if (propList == null)
17        {
18           System.out.println("NO properties found");
19        }
```

```
20            else
21            {
22               System.out.println("Properties: "+
                    propList.size());
23            }
24
25            Vector roleList = (Vector) obj.getRoleCollection();
26            if (roleList == null)
27            {
28               System.out.println("No roles found");
29            }
30            else
31            {
32               System.out.println("Roles: "+ roleList.size());
33            }
34      }
35
36   } catch(Exception e) {
37      System.out.println(">>>"+e.getMessage());
38      e.printStackTrace();
39   }
```

Table 4.4 explains specific statements in the code example by line number.

Line Number	***Explanation***
1	Creates an instance of a class that implements the `EVSApplicationService` interface. This interface defines the service methods used to access data objects. **NOTE:** The `EvsServiceInfo` is passed to the `getApplicationService()` method to support the integration and coexistence of the other CORE applications (*i.e.*, no naming clashes).
4	Creates a criterion object that defines the attribute values for which to search.
6	Calls the search method of the `EVSApplicationService` implementation with parameters that determine • the returned object type: `gov.nih.nci.evs.domain.DescLogicConcept` • the criteria that returned objects must meet, defined by the `dlc` object.
9 - 25	Instructs the search method to return objects in a collection, iterate through the objects, and print basic information about them.

Table 4.4 Explanation of statements in the installation verification procedure

Although this is a fairly simple example of the use of the EVS Java API, you can follow a similar sequence with more complex criteria to perform sophisticated manipulation of the data provided by caCORE EVS. The following sections provide additional information and examples.

Search Paradigm

The caCORE EVS architecture includes a service layer that provides a single, common access paradigm to clients that use any of the provided interfaces. As an object-oriented middleware layer designed for flexible data access, caCORE EVS relies heavily on strongly typed objects and an *object-in/object-out* mechanism.

Accessing and using a caCORE EVS system requires the following steps:

1. Ensure that the client application has access to the objects in the domain space.
2. Formulate the query criteria using the domain objects.
3. Establish a connection to the server.
4. Submit the query objects and specify the desired class of objects to be returned.
5. Use and manipulate the result set as desired.

caCORE EVS systems use four native application programming interfaces (APIs). Each interface uses the same paradigm to provide access to the caCORE EVS domain model, with minor changes specific to the syntax and structure of the clients. The following sections describe each API, identify installation and configuration requirements, and provide code examples.

The sequence diagram in *Figure 4.4* illustrates the caCORE EVS API search mechanism implemented to access the NCI EVS vocabularies.

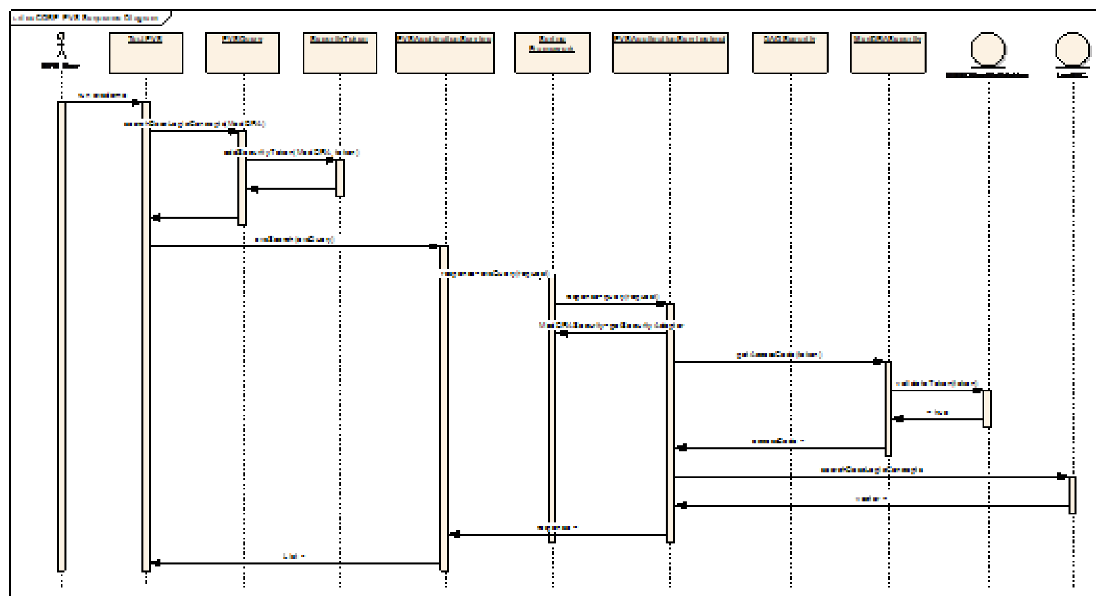

Figure 4.4 Sequence diagram - caCORE 4.0 EVS API search mechanism

To perform an EVS search, call the `evsSearch` operation defined in the `ApplicationService` class:

```
List evsSearch(EVSQuery evsQuery);
```

About EVSQuery and EVSQueryImpl

The `gov.nih.nci.evs.query` package consists of the `EVSQuery.java` interface and the `EVSQueryImpl.java` class. The methods defined in the `EVSQuery.java` file can be used to query the LexBIG Terminology Server. The query object generated by this class can hold one query at a time.

The following example code segment demonstrates an EVSQuery object that calls the `searchDescLogicConcept` method.

```
1   String vocabularyName = "GO";
2   String conceptCode = "GO:0005667";
3   EVSQuery evsQuery = new EVSQueryImpl();
4   evsQuery.searchDescLogicConcept(vocabularyName,conceptCode,
       true);
```

To perform a search on the Description Logic Vocabulary, you must specify the vocabulary name. In most instances, methods that do not require vocabulary names are NCI Metathesaurus queries.

EVSQuery Methods and Parameters

Most of the methods defined in the `EVSQuery` accept concept names or concept codes. If a method requires a vocabulary name as a parameter along with a concept code or name, you must pass a valid `DescLogicConcept` name or code to the search method.

Note: A search term is a String and is not considered a valid concept name. To get a valid `DescLogicConcept` name, you must perform a search using the `searchDescLogicConcept` method. Likewise, to get a valid `MetaThesaurusConcept` name or *CUI* (Concept Unique Identifier) you must perform a search using the `searchMetaThesaurus` method. Most of the search methods defined in the `EVSQuery` require a valid concept name or code.

Some of the methods defined in the `EVSQuery` are listed in *Table 4.5*.

Method Name	***Parameter***	***Comments***	***Returned by evsSearch***
`searchDescLogicConcept`	String vocabularyName	A valid Description Logic vocabulary name such as *NCI_Thesaurus*, *GO*, or *HL7*.	Returns one or more `DescLogicConcepts` in a List.
	String searchTerm	Any string value	
	int limit	Maximum number of records	

Table 4.5 Methods defined in the EVSQuery

Method Name	Parameter	Comments	Returned by evsSearch
searchMetaThesaurus	String searchTerm	Any String value or a valid Concept Unique Identifier. A Concept Unique Identifier is used to uniquely identify concepts in the Metathesaurus.	Returns one or more MetaThesaurusConcepts in a List.
	int Limit	Maximum number of records.	
	String Source	Source abbreviation. Each Source has a source abbreviation that can uniquely identify a source.	
	boolean CUI	Set to true if a concept unique identifier is used as a search term.	
	boolean shortResponse	Set to true for short response.	
	boolean score	Set to true for score.	
getHistoryRecords	String vocabularyName	A valid Description Logic vocabulary name such as *NCI_Thesaurus*, *GO*, or *HL7*.	Returns one or more HistoryRecords in a List.
	String conceptCode	A valid code of a DescLogicConcept.	
getVocabularyNames			Returns one or more Source objects in a List.
getMetaSources			Returns one or more MetaThesaurusConcepts in a List.
searchSourceByCode	String code	A valid Atom code.	Returns one or more MetaThesaurusConcepts in a List.
	String sourceAbbreviation	A valid source abbreviation.	

Table 4.5 Methods defined in the EVSQuery (Continued)

Method Name	Parameter	Comments	Returned by evsSearch
getTree	String vocabularyName	A valid Description Logic vocabulary name.	Returns a DescLogicConcept tree in a List.
	String rootName	A valid DescLogicConcept name.	
	boolean direction	Set to true if traverse down.	
	boolean isaFlag	Set to true if relationship is child.	
	int attributes	Sets an AttributeSetDescriptior value.	
	int levels	Depth of the tree.	
	Vector roles	Valid role names.	

Table 4.5 Methods defined in the EVSQuery (Continued)

Using EVSQuery

Accessing Secured Vocabularies

Secured vocabularies such as MedDRA require a valid security token for access. The following example shows the syntax for using `EVSQuery` to set a security token and access a secured Vocabulary.

```
1  gov.nih.nci.evs.query.EVSQuery evsQuery = new
2  gov.nih.nci.evs.query.EVSQueryImpl();
3  gov.nih.nci.evs.security.SecurityToken token = new
4  gov.nih.nci.evs.security.SecurityToken();
5  token.setAccessToken("123456");
6  evsQuery.addSecurityToken("MedDRA", token);
7  evsQuery.getDescLogicConcept("MedDRA", "Blood", false);
```

Note: You must obtain a valid security token from the NCICB to access MedDRA through the caCORE EVS API. The security token value used in the example is not valid.

Creating an EVS Search Request

To create an EVS search request, follow these steps:

1. Add the `EvsServiceInfo` parameter.
2. Create an `ApplicationService` instance:

```
EVSApplicationService appService = (EVSApplicationService)
   ApplicationServiceProvider.getApplicationService
   ("EvsServiceInfo");
```

3. Instantiate an `EVSQuery` instance, then set the method name and parameters:

```
EVSQuery evsQuery = new EVSQueryImpl();
evsQuery.searchDescLogicConcepts("NCI_Thesaurus","blood*",10);
```

4. Set the security token value.

   Note: You can omit this step if the vocabulary does not require a security token.

```
gov.nih.nci.evs.security.SecurityToken token =
   new gov.nih.nci.evs.security.SecurityToken();
token.setAccessToken("xxxxxx");
evsQuery.addSecurityToken(vocabularyName, token);
```

5. Call the `evsSearch` method defined in the `ApplicationService` class to query EVS.

```
List evsResults = (List)appService.evsSearch(evsQuery);
```

The result objects are populated. The return type varies based on the search method call set in the `EVSQuery` instance.

Examples of Use

Example 4.1: Search for DescLogicConcepts by Term

```
 1  public static void main(String[] args)
 2  {
 3     try {
 4        EVSApplicationService appService =
 5           (EVSApplicationService)ApplicationServiceProvider.
 6           getApplicationService("EvsServiceInfo");
 7        EVSQuery evsQuery = new EVSQueryImpl();
 8        evsQuery.searchDescLogicConcepts("NCI_Thesaurus",
             "blood*",10);
 9        List evsResults = (List)appService.evsSearch(evsQuery);
10     } catch(ApplicationException ex){
11     }
12  }
```

Table 4.6 explains specific statements in the code by line number.

Line Number	Explanation
3	Creates an instance of a class that implements the `ApplicationService` interface. This interface defines the service methods used to access data objects.
4	Creates a new `EVSQuery` object.
7	Specifies the search method and parameters. The `searchDescLogicConcept` method performs a search in the `NCI_Thesaurus` vocabulary for a term that starts with *blood* and returns a maximum of ten concepts if found.
6	Calls the `evsSearch` method of the `ApplicationService` implementation passing the `EVSQuery` object. This method returns a `List` Collection. The type of object that is returned depends on the search parameters set in the `EVSQuery` object. In this case the `searchDescLogicConcept` method was invoked, and the resulting objects are of type `DescLogicConcept`

Table 4.6 Explanation of statements in Example 4.1

Example 4.2: Search MetaThesaurusConcepts by Atom

```
 1  try {
 2     EVSApplicationService appService =
          (EVSApplicationService)ApplicationServiceProvider.
          getApplicationService(“EvsServiceInfo”);
 3     EVSQuery evsQuery = new EVSQueryImpl();
 4     evsQuery.searchSourceByAtomCode("10834-0","*");
 5     List evsResults = (List)appService.evsSearch(evsQuery);
 6
 7     for(int m=0; m<evsResults.size(); m++ ){
 8        MetaThesaurusConcept concept =
             (MetaThesaurusConcept)evsResults.get(m);
 9        System.out.println("\nConcept code: "+concept.getCui()
             +"\n\t"+concept.getName());
10        List sList = concept.getSourceCollection();
11        System.out.println("\tSource-->" + sList.size());
12        for(int y=0; y<sList.size(); y++){
13           Source s = (Source)sList.get(y);
14           System.out.println("\t - "+s.getAbbreviation());
15        }
16
17        List semanticList = concept.getSemanticTypeCollection();
18        System.out.println("\tSemanticType---> count ="+
             semanticList.size());
19        for(int z=0; z<semanticList.size(); z++){
20           SemanticType sType = (SemanticType)
                semanticList.get(z);
21           System.out.println("\t- Id: "+sType.getId()+"\n\t-
                Name: "+sType.getName());
22        )
23
24        List atomList = concept.getAtomCollection();
25        System.out.println("\tAtoms -----> count = "+
             atomList.size());
26        for(int i=0; i<atomList.size(); i++){
27           Atom at = (Atom)atomList.get(i);
28           System.out.println("\t -Code: "+ at.getCode()+" -Name:
                "+ at.getName() +" -LUI: "+ at.getLui()+" -Source:
                "+ at.getSource().getAbbreviation());
29        }
30
31        List synList = concept.getSynonymCollection();
32        System.out.println("\tSynonyms -----> count = "+
             synList.size());
33        for(int i=0; i < synList.size(); i++){
34           System.out.println("\t - "+ (String) synList.get(i));
35        }
36     }
37
38  } catch(ApplicationException ex) {
39
40  }
```

Table 4.7 explains specific statements in the code by line number.

Line Number	Explanation
2	Creates an instance of the `EVSApplicationService`.
3	Creates a new `EVSQuery` object.
4	Specifies the search method and parameters. The `searchSourceByAtomCode` method performs a search on all the sources specified in the Metathesaurus for `MetaThesaurusConcepts` that have an Atom code value 10834-0. The source abbreviation specified is an asterisk (*). Thus, all sources are searched for the specified atom.
5	Calls the `evsSearch` method of the `EVSApplicationService` implementation passing the `EVSQuery` object. This method returns a `List` collection. The type of object that is returned depends on the search parameters set in the `EVSQuery` object; in this case, the `searchSourceByAtomCode` method was invoked, and the resulting objects are of type `MetaThesaurusConcept`.
7 - 36	• Iterates through the result set (line 7) • Casts the result object to a `MetaThesaurusConcept` (line 8) • Prints the attributes and association values of the MetaThesaurusConcept (lines beginning with `System.out.println`)

Table 4.7 Explanation of statements in Example 4.2

Web Services API

The caCORE EVS Web Services API enables access to caCORE EVS data and vocabulary data from development environments where the Java API cannot be used, or where use of XML Web services is more desirable. This includes non-Java platforms and languages such as Perl, C/C++, .NET framework (C#, VB.Net), and Python.

The Web services interface can be used in any language-specific application that provides a mechanism for consuming XML Web services based on the Simple Object Access Protocol (SOAP). In those environments, connecting to caCORE EVS can be as simple as providing the end-point URL. Some platforms and languages require additional client-side code to handle the implementation of the SOAP envelope and the resolution of SOAP types.To view a list of packages that cater to different programming languages, visit http://www.w3.org/TR/SOAP/ and http://www.soapware.org/.

To maximize standards-based interoperability, the caCORE Web service conforms to the Web Services Interoperability Organization (WS-I) basic profile. The WS-I basic profile provides a set of non-proprietary specifications and implementation guidelines that enable interoperability between diverse systems. For more information about WS-I compliance, visit http://www.ws-i.org.

On the server side, Apache Axis is used to provide SOAP-based, inter-application communication. Axis provides the appropriate serialization and deserialization methods for the JavaBeans to achieve an application-independent interface. For more information about Axis, visit http://ws.apache.org/axis/.

Configuration

The caCORE/EVS WSDL file is located at http://evsapi.nci.nih.gov/evsapi41/services/evsapi41Service?wsdl. In addition to describing the protocols, ports, and operations exposed by the caCORE EVS Web service, this file can be used by a number of IDEs and tools to generate stubs for caCORE EVS objects. This enables code on different platforms to instantiate native objects for use as parameters and return values for the Web service methods. For more information on how to use the WSDL file to generate class stubs, consult the specific documentation for your platform.

The caCORE EVS Web services interface has a single end point called `evsapiService`, which is located at http://evsapi.nci.nih.gov/evsapi41/services/evsapi41Service. Client applications should use this URL to invoke Web service methods.

Operations

Through the caCOREService endpoint, developers have access to the following operations:

Operation	getVersion
Input Scheme	None
Output Schema	<complexType> <sequence> <element type="xsd:string"/> </sequence> </complexType>
Description	Returns an xsd:string containing the version of the running caCORE system (e.g., caCORE 4.0).

Operation	queryObject
Input Schema	<complexType> <sequence> <element name="in0" type="xsd:string"/> <element name="in1" type="xsd:anyType"/> </sequence> </complexType>
Output Schema	<sequence> <element name="queryReturn" type="ArrayOf_xsd_anyType"/> </sequence>
Description	Performs a search for objects that conform to the criteria defined by input parameter in1 and whose resulting objects are of the type reached by traversing the node graph specified by parameter in0. The result is a set of serialized objects. The type ArrayOf_xsd_anyType resolves to a sequence of xsd:anyType elements.

Operation	Query
Input Schema	<complexType> <sequence> <element name="in0" type="xsd:string"/> <element name="in1" type="xsd:anyType"/> <element name="in2" type="xsd:int"/> <element name="in3" type="xsd:int"/> </sequence> </complexType>
Output Schema	<sequence> <element name="queryReturn" type="ArrayOf_xsd_anyType"/> </sequence>
Description	Identical to the previous queryObject method except that it allows control over the result set by specifying the row number of the first row (in2) and the maximum number of objects to return (in3).

Be aware that a significant decision has been made regarding the behavior of the Web services interface. When a query is performed with this interface, returned objects do not contain nor refer to their associated objects. (A notable exception is with the EVS domain model discussed in the *Limitations* section on page 61.) This means that a separate query invocation must be performed for each set of associated objects that need to be retrieved. *Example 4.3* (page 58) demonstrates this functionality.

Considerations

The EVS domain objects are unique in the way they are used with the Web services interface. EVS classes that can be queried from Web services always provide associations to their related objects. This enables access to the objects that are not of type `DescLogicConcept`, `MetaThesaurusConcept`, or `HistoryRecord`.

Because of the unique behavior and properties of the EVS domain model, queries using the Web services interface can be performed only on the selected attribute values listed in *Table 4.8*.

Class	***Available Search Attributes***
`DescLogicConcept`	Name
	Code
	Property name and value
	Role name and value
`MetaThesaurusConcept`	Name
	CUI (Concept Unique Identifier)
	Atom code and source abbreviation
`HistoryRecord`	`DescLogicConcept` name or code (`HistoryRecord` is the targetObject and the `DesLogicConcept` is the criteriaObject).

Table 4.8 Allowable attributes for searching the EVS domain model

Examples of Use

Example 4.3: Simple Search (NCI Thesaurus)

This example demonstrates a simple Java query that uses the Web services API. It uses Apache Axis on the client side to handle the type mapping, SOAP encoding, and operation invocation.

```
 1 try {
 2    String endpointURL = "http://evsapi.nci.nih.gov/evsapi41/
         services/evsapi41Service";
 3    String methodName = "queryObject";
 4    Service service = new Service();
 5    Call call = (Call) service.createCall();
 6
 7    call.setTargetEndpointAddress(new java.net.URL(endpointURL));
 8    call.setOperationName(new QName("EVSService", "queryObject"));
 9    call.addParameter("arg1",
         org.apache.axis.encoding.XMLType.XSD_STRING,
         ParameterMode.IN);
10    call.addParameter("arg2",
         org.apache.axis.encoding.XMLType.XSD_ANYTYPE,
         ParameterMode.IN);
11    call.setReturnType(org.apache.axis.encoding.XMLType.
         SOAP_ARRAY);
12
13    QName qnDLCArr = new QName("urn:domain.evs.nci.nih.gov",
         "ArrayOf_tns1_DescLogicConcept");
14    call.registerTypeMapping(DescLogicConcept.class, qnDLCArr,
         new org.apache.axis.encoding.ser.BeanSerializerFactory(),
         new org.apache.axis.encoding.ser.BeanDeserializerFactory());
15
16    DescLogicConcept dlc = new DescLogicConcept();
17    dlc.setName("ear*");
18
19    call.setReturnType(qnDLCArr);
20
21    Object thesarusParams =
         new Object[]{"gov.nih.nci.evs.domain.DescLogicConcept",
         dlc};
22    DescLogicConcept[] dlcs =
         (DescLogicConcept[])call.invoke(thesaurusParams);
23
24    char counter = ‘a’;
25
26    for (DescLogicConcept d: dlcs) {
27       System.out.println("\t" + counter + ") Concept name; " +
            d.getName());
28       System.out.println("\t code: " + d.getCode());
29       System.out.println("\t ----------------------------------");
30       counter++;
31    }
32    System.out.println("\tNumber of items returns from Thesaurus "
         + dlcs.length);
33 } catch (Exception ex) {
34    System.out.println(ex.getMessage());
35 }
```

Table 4.9 explains specific statements in the code by line number.

Line Number	Explanation
4 - 5	Defines a new Web service call.
7 - 11, 19	Sets call properties, including the name of the operation to invoke, the target property address, the input parameters that will be sent, and the return type.
13 - 14	Maps a serialized object to its Java equivalent using the qualified name of the object from the WSDL file. In this case, the XML element `urn:domain.evs.nci.nih.gov` namespace is mapped to the Java `DescLogicConcept` array.
16 - 17	Creates a `DescLogicConcept` and sets the name attribute to `ear*`.
21 - 22	Invokes the Web service operation using an array of two objects (target class name and criteria object) as input parameters; expects an object array as the result.
26 - 31	Casts each object in the result array to type `DescLogicConcept` and prints the name and code.

Table 4.9 Explanation of statements in Example 4.3

Example 4.4: EVS Domain Search (NCI Metathesaurus)

This example demonstrates use of the Web services interface to query data from the NCI Metathesaurus using EVS domain objects.

```
 1 try {
 2    String endpointURL = "http://evsapi.nci.nih.gov/evsapi41/
         services/evsapi41Service";
 3    String methodName = "queryObject";
 4    Service service = new Service();
 5    Call call = (Call) service.createCall();
 6
 7    call.setTargetEndpointAddress(new java.net.URL(endpointURL));
 8    call.setOperationName(new QName("EVSService", "queryObject"));
 9    call.addParameter("arg1",
         org.apache.axis.encoding.XMLType.XSD_STRING,
         ParameterMode.IN);
10    call.addParameter("arg2",
         org.apache.axis.encoding.XMLType.XSD_ANYTYPE,
         ParameterMode.IN);
11    call.setReturnType(org.apache.axis.encoding.XMLType.
         SOAP_ARRAY);
12
13    QName qnMTCArr = new QName("urn:domain.evs.nci.nih.gov",
         "ArrayOf_tns1_MetaThesaurusConcept");
14    call.registerTypeMapping(MetaThesaurusConcept.class, qnMTCArr,
         new org.apache.axis.encoding.ser.BeanSerializerFactory(),
         new org.apache.axis.encoding.ser.BeanDeserializerFactory());
15
16    MetaThesaurusConcept mtc = new MetaThesaurusConcept();
17    MTC.setName("blood*");
18
19    call.setReturnType(qnMTCArr);
20
```

```
21    Object metaParams = new
         Object[]{"gov.nih.nci.evs.domain.MetaThesaurusConcept",
         mtc};
22    MetaThesaurusConcept[] meta = null;
23
24    try {
25       meta = (MetaThesaurusConcept[])call.invoke(metaParams);
26       char counter = 'a';
27
28       for (MetaThesaurusConcept m: meta) {
29          System.out.println("\t" + counter + ") Concept name; " +
               m.getName());
30          System.out.println("\t code: " + m.getCui());
31          System.out.println("\t ----------------------------------
               ");
32          counter++;
33       }
34       System.out.println("\tSize" + meta.length);
35    } catch (Exception ex) {
36       System.out.println("Error: " + ex);
37    }
38 } catch (Exception ex) {
39    System.out.println(ex.getMessage());
40 }
```

Table 4.10 explains specific statements in the code by line number.

Line Number	*Explanation*
4 - 5	Defines a new Web service call.
7 - 11, 14, 19	Sets call properties, including the name of the operation to invoke, the target property address, the input parameters that will be sent, and the return type.
13 - 14	Maps a serialized object to its Java equivalent using the qualified name of the object from the WSDL file. In this case, the XML element `urn:domain.evs.nci.nih.gov` namespace is mapped to the Java `MetaThesaurusConcept` array.
16 - 17	Creates a `MetaThesaurusConcept` and sets the name attribute to `blood*`.
21 - 22	Invokes the Web service operation using an array of two objects (target class name and criteria object) as input parameters; expects an object array as the result.
28 - 33	Casts each object in the result array to type `MetaThesaurusConcept` and prints the name and code.

Table 4.10 Explanation of statements in Example 4.4

Limitations

Note the following limitations of the Web Services API and the recommendations for handling them:

- By default, the `queryObject` operation limits the result set to 1000 objects, even if the size of the result set is larger. To retrieve more than 1000 objects, you must use the query operation and specify the first object index (parameter `in2`) as greater than 1000.
- Result sets serialized and returned by the Web services interface do not currently include associations to related objects. A consequence of this behavior is that nested criteria objects with one-to-many associations that are passed to the query or `queryObject` operations will throw an exception.
- Because the Web services invocation routinely times out, queries that take a long time to execute may not complete. If this is the case, use the query method to specify a smaller result count.
- Access to the EVS domain model is limited by the Web services interface, as explained in *Table 4.11*.

Typical Behavior	***EVS Model Behavior***
Can query for any object in the object model	Can query only for the following: • `DescLogicConcept` • `HistoryRecord` • `MetaThesaurusConcept`
Does not populate the association values of the caCORE domain objects. You need to run a second query to get associated values.	Populates all attributes of the result object.
Can perform queries on any attribute value	Can perform queries only on selected attribute values. For more information, see *Table 4.5* on page 48.

Table 4.11 Access to the EVS domain model

XML-HTTP API

The caCORE EVS XML-HTTP API, based on the REST (Representational State Transfer) architectural style, provides a simple interface using the HTTP protocol. In addition to its ability to be invoked from most Internet browsers, developers can use this interface to build applications that do not require any programming overhead other than an HTTP client. This is particularly useful for developing Web applications using AJAX (Asynchronous JavaScript and XML).

Service Location and Syntax

The CORE EVS XML-HTTP interface uses the following URL syntax:

```
http://{server}/{servlet}?query={returnClass}&{criteria}&
   resultCounter={counter}&startIndex={index}&
   pageSize={pageSize}&pageNumber={pageNumber}
```

Table 4.12 explains the syntax, indicates whether specific elements are required, and gives examples.

Element	Meaning	Required	Example
server	Name of the Web server on which the caCORE EVS 4.1 Web application is deployed.	Yes	evsapi.nci.nih.gov /evsapi41
servlet	URI and name of the servlet that will accept the HTTP GET requests.	Yes	evsapi41/GetXML evsapi41/GetHTML
returnClass	Class name indicating the type of objects that this query should return.	Yes	query=DescLogicCon cept
criteria	Search request criteria describing the requested objects.	Yes	DescLogicConcept [@id=2]
counter	Number of top-level objects returned by the search.	No	resultCounter=500
index	Starting index of the result set.	No	startIndex=25
pageSize	Number of records to display on each page.	No	pageSize=50
pageNumber	The number of the page for which to display results.	No	pageNumber=3

Table 4.12 URL syntax used by the caCORE EVS XML-HTTP interface

The caCORE EVS architecture currently provides two servlets that accept incoming requests:

- *GetXML* returns results in an XML format that can be parsed and consumed by most programming languages and many document authoring and management tools.
- *GetHTML* presents result using a simple HTML interface that can be viewed by most modern Internet browsers.

Within the request string of the URL, the criteria element specifies the search criteria using XQuery-like syntax. Within this syntax, square brackets ([and]) represent attributes and associated roles of a class, the *at* symbol (@) signals an attribute name/value pair, and a forward slash character (/) specifies nested criteria.

Criteria statements in XML-HTTP queries generally use the following syntax (although you can also build more complex statements):

```
{ClassName}[@{attributeName}={value}] [@{attributeName}={value}]…

ClassName}[@{attributeName}={value}]/

{ClassName}[@{attributeName}={value}]/…
```

Table 4.13 explains the syntax for criteria statements and gives examples.

Parameter	***Meaning***	***Example***
`ClassName`	The name of a class.	`DescLogicConcept`
`attributeName`	The name of an attribute o.f the return class or an associated class	`name`
`value`	The value of an attribute.	`ear*`

Table 4.13 Criteria statements within XML-HTTP queries

Examples of Use

The examples in *Table 4.14* demonstrate the usage of the XML-HTTP interface. In actual usage, these queries would either be submitted by a block of code or entered in the address bar of a Web browser.

Note that the servlet name *GetXML* in each of the examples can be replaced with *GetHTML* to view with layout and markup in a browser.

Query	http://evsapi.nci.nih.gov/evsapi41/GetXML?query=DescLogicConcept[name=blood*]
Semantic Meaning	Find all objects of type `DescLogicConcept` whose name starts with *blood*.
Biological Meaning	Find all concepts that refer to blood.

Table 4.14 XML-HTTP interface examples

Working with Result Sets

Because HTTP is a stateless protocol, the caCORE EVS server cannot detect the context of any incoming request. Consequently, each invocation of *GetXML* or *GetHTML* must contain all of the information necessary to retrieve the request, regardless of previous requests. Developers should consider this when working with the XML-HTTP interface.

Retrieving Related Results Using XLinks

When using the GetXML servlet to retrieve results as XML, associations between objects are converted to XLinks in the XML markup. The link notation, shown below, enables the client to make a subsequent request to retrieve the associated objects.

```
<class name="gov.nih.nci.evs.domain.DescLogicConcept"
   recordNumber="1">
…
<field name="Vocabulary"
   xlink:type="simple"
   xlink:href="http://evsapi.nci.nih.gov/evsapi41/
      GetXML?Query=Vocabulary&DescLogicConcept[@id=5]"> getVocabulary
</field>
…
</class>
```

Controlling the Number of Items Returned

The GetXML servlet provides a throttling mechanism that enables developers to define the number of results returned on any single request and specify where in the result set to start. The following example are used with a search request that returns 500 results:

- `resultCounter=450` will return only the last 50 records.
- `startIndex=50` will return only the first 50 records.

Paging Results

In addition to controlling the number of results to display, the GetXML servlet also provides a mechanism to support *paging*. Common to many Web sites, paging displays results over a number of pages so that, for example, a request that yields 500 objects could be displayed over 10 pages of 50 objects each.

When the paging feature is used, the GetXML servlet includes XLinks to each of the result pages in an XML `<page>` element. The element data of the `<page>` element is the number of the page, suitable for output as text or HTML when used with an XSL stylesheet:

```
<page number="1">
   xlink:type="simple"
   xlink:href="http://evsapi.nci.nih.gov/evsapi41/
      GetXML?query={query}& pageNumber=4
      &resultCounter=1000&startIndex=0"> 4
</page>
```

Limitations

When specifying attribute values in the query string, note that use of the following characters generates an error:

[ ] / \ # & %

About the XMLUtility Class

Overview

When accessible through the SDK framework, caCORE provides an `XMLUtility` class that converts caCORE EVS domain objects between native Java objects and XML serializations based on standard XML schemas. This utility is available in the `gov.nih.nci.common.util` package.

The XML schemas for all caCORE EVS domain objects that are directly generated from the UML model (described on page 39) are included in the downloadable archive in the `lib` directory. Currently, the `XMLUtility` class generates XML that includes only object attributes; associated objects are not included.

Two files include the properties used by the `XMLUtility` class:

- `xml.properties` defines basic information needed by the class and also contains a property defining the name of the second file.
- `xml-mapping.xml` (the default name) defines the binding between the class and attribute names and the corresponding XML element and attribute names.

The caCORE EVS client provides a default marshaller and unmarshaller. Developers who want to use their own marshaller or demarshaller should provide the fully qualified name of the two classes in the `xml.properties` file.

Example of Use

Example 4.5: Using the XMLUtility Class

In the following example code, `XMLUtility` is used to serialize an object and save it to a file stream. It is then used to instantiate a new object from the file.

```
1   EVSApplicationService appService =
       (EVSApplicationService)ApplicationServiceProvider
       .getApplicationService(“EvsServiceInfo”);
2
3   Marshaller marshaller = new caCOREMarshaller("xml-mapping.xml",
       false);
4   Unmarshaller unmarshaller = new
       caCOREUnmarshaller("unmarshaller-xml-mapping.xml", false);
5   XMLUtility myUtil = new XMLUtility(marshaller, unmarshaller);
6   Class klass = Vocabulary.class;
7   Object o = klass.newInstance();
8   System.out.println("Searching for "+klass.getName());
9   try {
10     Collection results = appService.search(klass, o);
11     for(Object obj : results) {
12        File myFile = new File("./output/" + klass.getName() +
             "_test.xml");
13        FileWriter myWriter = new FileWriter(myFile);
14        myUtil.toXML(obj, myWriter);
15        myWriter.close();
16        printObject(obj, klass);
17        DocumentBuilder parser =
             DocumentBuilderFactory.newInstance()
             .newDocumentBuilder();
```

```
18          Document document = parser.parse(myFile);
19          SchemaFactory factory =
               SchemaFactory.newInstance(XMLConstants
               .W3C_XML_SCHEMA_NS_URI);
20
21          try {
22             System.out.println("Validating " + klass.getName() + "
                  against the schema......\n\n");
23             Source schemaFile = new
                  StreamSource(Thread.currentThread().getContextClass
                  Loader().getResourceAsStream(klass.getPackage().get
                  Name() + ".xsd"));
24             Schema schema = factory.newSchema(schemaFile);
25             Validator validator = schema.newValidator();
26             validator.validate(new DOMSource(document));
27             System.out.println(klass.getName() + " has been
                  validated!!!\n\n");
28          } catch (Exception e) {
29             System.out.println(klass.getName() + " has failed
                  validation!!! Error reason is: \n\n" +
                  e.getMessage());
            }
30
31          System.out.println("Un-marshalling " + klass.getName() +
               " from " + myFile.getName() + "......\n\n");
32          Object myObj = (Object) myUtil.fromXML(myFile);
33          printObject(myObj, klass);
34          myWriter.close
35          break;
36       }
37    } catch(Exception e) {
38       System.out.println("Exception caught: " + e.getMessage());
39       e.printStackTrace();
40    }
```

Table 4.15 explains specific statements in the code by line number.

Line Number	***Explanation***
1	Instantiates the `EVSApplicationService`.
3 - 5	Instantiates the marshaller and unmarshaller using the appropriate mapping files and uses them to instantiate the XMLUtility.
10	Searches against the Vocabulary class.
11	Iterates through all of the returned results.
12	Instantiates a new XML file based on the search class.
13 - 16	Uses the XML utility to convert the returned object to XML and write to the output stream.
16 - 29	Validates the generated XML file.
31 - 40	Unmarshalls the generated XML object from the written file and prints to `System.out`.

Table 4.15 Explanation of statements in Example 4.5

Distributed LexBIG API

Overview

In place of the existing EVS 3.2 object model, caCORE EVS is making a gradual transition toward a pure LexBIG back-end terminology server and exposure of the LexBIG Service object model. caCORE 3.2 and earlier required a custom API layer between external users of the system and the proprietary Apelon Terminology Server APIs. With the transition to LexBIG, caCORE EVS can publicly expose the open source terminology service API without requiring a custom API layer.

To allow users of caCORE EVS 3.2 and earlier to prepare for transition to the LexBIG API, caCORE EVS 4.0 provides a *Distributed LexBIG* (DLB) API in addition to the EVS API (based on the 3.2 object model). The DLB API provides remote access to the LexBIG API residing on the caCORE EVS server.

Architecture

The LexBIG API is exposed by the EVS caCORE System for remote, distributed access (*Figure 4.5*). The caCORE System's `EVSApplicationService` class implements the `LexBIGService` interface, effectively exposing LexBIG via caCORE.

Since in many cases the objects returned from the `LexBIGService` are not merely beans, but full-fledged data access objects (DAOs), the caCORE EVS client is configured to proxy method calls into the LexBIG objects and forward them to the caCORE server so that they execute within the LexBIG environment.

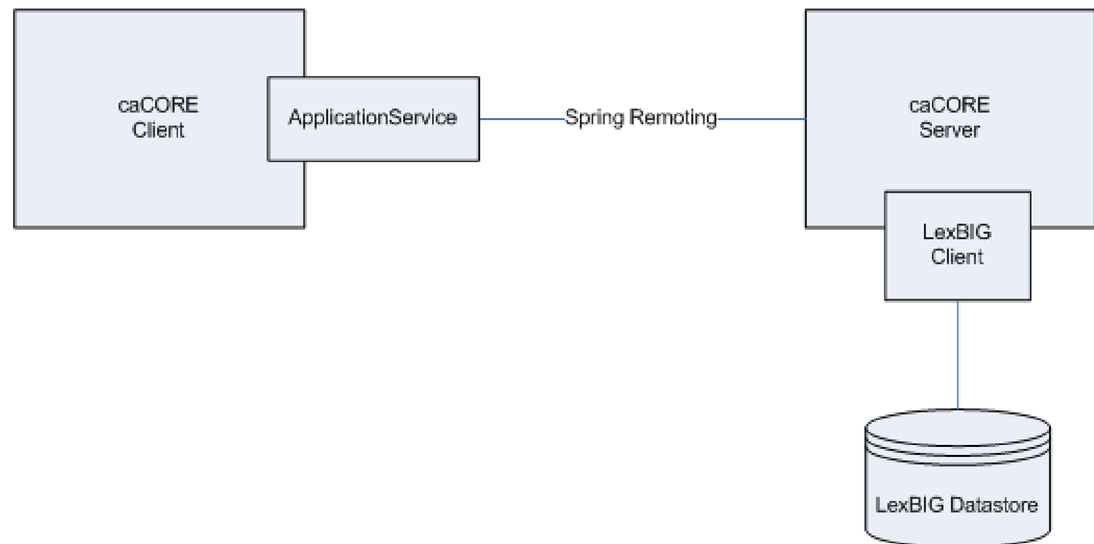

Figure 4.5 DLB Architecture

The DLB environment will be configured on the caCORE EVS Server (http://evsapi.nci.nih.gov/evsapi41). This will give the server access to the LexBIG database and other resources. The client must therefore go through the caCORE EVS server to access any LexBIG data.

LexBIG Annotations

To address LexBIG DAOs, the LexBIG API integration incorporated the addition of (1) Java annotation marking methods that can be safely executed on the client side; and (2) classes that can be passed to the client without being wrapped by a proxy. The annotation is named `@lgClientSideSafe`. Every method in the LexBIG API that is accessible to the caCORE EVS user had to be considered and annotated if necessary.

Aspect Oriented Programming Proxies

LexBIG integration with caCORE EVS was accomplished using Spring Aspect Oriented Programming (AOP) to proxy the LexBIG classes and intercept calls to their methods. The caCORE EVS client wraps every object returned by the `LexBIGService` inside an AOP Proxy with advice from a `LexBIGMethodInterceptor` ("the interceptor").

The interceptor is responsible for intercepting all client calls on the methods in each object. If a method is marked with the `@lgClientSideSafe` annotation, it proceeds normally. Otherwise, the object, method name, and parameters are sent to the caCORE EVS server for remote execution.

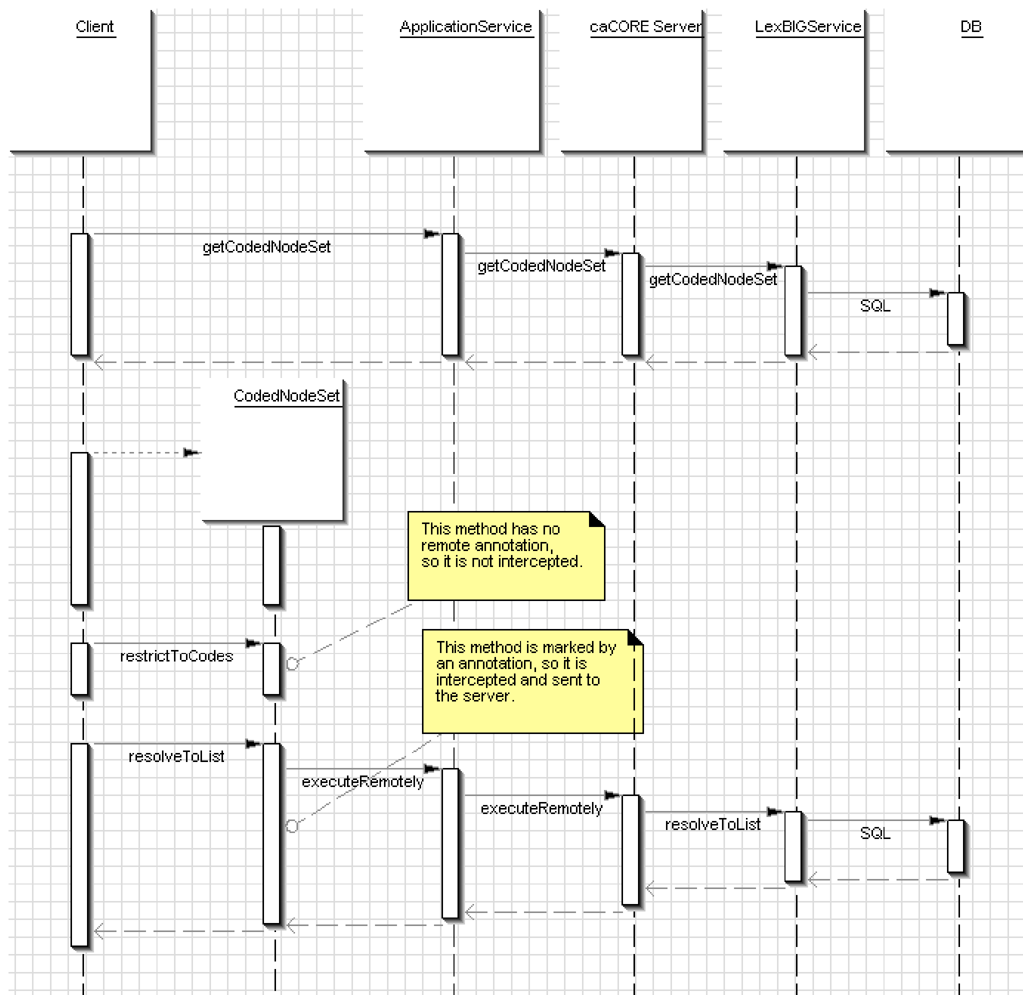

Figure 4.6 Sequence diagram showing method interception

LexBIG API Documentation

The Mayo Clinic wrote the LexBIG 2.1.1 API. Documentation describing the LexBIG Service Model is available on the LexGRID Vocabulary Services for caBIG GForge site at https://gforge.nci.nih.gov/frs/?group_id=14.

LexBIG Installation and Configuration

The DLB API is strictly a Java interface and requires Internet access for remote connectivity to the caCORE EVS server. Access to the DLB API requires access to the `evsapi-client.jar` file, available for download on the NCICB Web site. The `evsapi-client.jar` file needs to be available in the classpath. For more information, see *Installing and Configuring the EVS 3.2 Java API* on page 41.

Example of Use

Example 4.6: Using the DLB API

The following code sample shows use of the DLB API to retrieve the list of available coding schemes in the LexBIG repository.

```
1   public class Test {
2      /**
3      * Initialize program variables
4      */
5
6      private String codingScheme = null;
7      private String version = null;
8
9      DLBAdapter adapter = null;
10     LexBIGService lbSvc;
11
12     public Test(String codingScheme, String version)
13     {
14        //Set the EVS URL (for remote access)
15        String evsUrl = "http://evsapi.nci.nih.gov/evsapi41/http/
             remoteService";
16
17        boolean isRemote = true;
18        this.codingScheme = codingScheme;
19        this.version = version;
20
21        // Get the LexBIG service reference from EVS Application
             Service
22        lbSvc = EVSApplicationService.getRemoteInstance(evsUrl);
23
24        // Set the vocabulary to work with
25        Boolean retval = adapter.setVocabulary(codingScheme);
26
27        codingSchemeMap = new HashMap();
28        try {
29           // Get the LexBIG service from the RemoteServerUtil
30           lbSvc = RemoteServerUtil.createLexBIGService();
31
32           // Using the LexBIG service, get the supported coding
                schemes
```

```
33          CodingSchemeRenderingList csrl =
               lbSvc.getSupportedCodingSchemes();
34
35          // Get the coding scheme rendering
36          CodingSchemeRendering[] csrs =
               csrl.getCodingSchemeRendering();
37
38          // For each coding scheme rendering...
39          for (int i=0; i<csrs.length; i++) {
40             CodingSchemeRendering csr = csrs[i];
41
42             // Determine whether the coding scheme rendering is
                  active or not
43             Boolean isActive =
                  csr.getRenderingDetail().getVersionStatus()
                  .equals(CodingSchemeVersionStatus.ACTIVE);
44             if (isActive != null &&
                  isActive.equals(Boolean.TRUE)) {
45
46             // Get the coding scheme summary
47             CodingSchemeSummary css =
                  csr.getCodingSchemeSummary();
48
49             // Get the coding scheme formal name
50             String formalname = css.getFormalName();
51
52             //Get the coding scheme version
53             String representsVersion =
                  css.getRepresentsVersion();
54             CodingSchemeVersionOrTag vt = new;
55             CodingSchemeVersionOrTag();
56             vt.setVersion(representsVersion);
57
58             // Resolve coding scheme based on the formal name
59             CodingScheme scheme = null;
60
61             try {
62                scheme =
                     lbSvc.resolveCodingScheme(formalname, vt);
63                if (scheme != null)
64                {
65                   codingSchemeMap.put((Object) formalname,
                        (Object) scheme);
66                }
67             } catch (Exception e) {
68                // Resolve coding scheme based on the URN
69                String urn = css.getCodingSchemeURN();
70                try {
71                   scheme = lbSvc.resolveCodingScheme(urn, vt);
72                   if (scheme != null)
73                   {
74                      codingSchemeMap.put((Object) formalname,
                           (Object) scheme);
75                   }
76                } catch (Exception ex) {
77                String localname = css.getLocalName();
```

```
78
79                      // Resolve coding scheme based on the local name
80                      try {
81                         scheme = lbSvc.resolveCodingScheme(localname,
                              vt);
82                         if(scheme != null)
83                         {
84                            codingSchemeMap.put((Object) formalname,
                                 (Object) scheme);
85                         }
86                      } catch (Exception e2) {
87                      }
88                   }
89                }
90             }
91          }
92          } catch (Exception e) {
93             e.printStackTrace();
94          }
95    }
96
97    /**
98     *Main
99     */
100   public static void main (String[] args)
101   {
102      String name = "NCI Thesaurus";
103      String version = "06.12d";
104
105      // Instantiate the Test Class
106      Test test = new Test(name, version);
107      }
108   }
```

Distributed LexBIG Adapter Example of Use

The DLB Adapter is an extension of the DLB API. It provides a set of convenient methods for simplifying or making familiar use of the DLB API. Access is achieved first through the EVS API and the DLB API.

The DLB Adapter is designed with dual-mode functionality. It can be used as

- a *local* installation: direct interface to a local installation of the LexBIG API, or
- a *remote* installation: an additional layer to the NCI DLB API.

The DLB Adapter is packaged as a `.jar` file (`dlbadapter.jar`). Once this file has been added to the application classpath, the programmer can decide how to interface with LexBIG (locally or remotely).

Note: The Javadocs for the DLB Adapter are available at http://gforge.nci.nih.gov/docman/view.php/366/11758/evsapi_4_1_javadoc.zip.

The following code example represents how to use DLB Adapter in both local and remote modes.

```
 1 public class Test {
 2    /**
 3     *Initialize program variables
 4     */
 5    private String codingScheme = null;
 6    private String version = null;
 7
 8    DLBAdapter adapter = null;
 9    LexBIGService lbSvc;
10
11    /**
12     *Test constructors
13     */
14    public Test(String codingScheme)
15    {
16       this.codingScheme = codingScheme;
17       this.version = "";
18    }
19
20    public Test(String codingScheme, String version
21    {
22
23    /**
24     * Establish a reference to the EVS Production URL (remote
         access).
25     */
26    String evsUrl = "http://evsapi.nci.nih.gov/evsapi41/http/
         remoteService";
27
28    /**
29     * Set the isRemote (LexBIG installation) variable
30     */
31    boolean isRemote = true;
32    this.codingScheme = codingScheme;
33    this.version = version;
34
35    /**
36    * The DLB Adapter allows you to connect to a co-located/local
         installation of LexBIG or a remote installation of LexBIG.
37    */
38    if (isRemote)
39    {
40       lbSvc = EVSApplicationService.getRemoteInstance(evsUrl);
41       adapter = new DLBAdapter((EVSApplicationService)lbSvc);
42
43       System.out.println("\n***********************************
            ********************************");
44       System.out.println
            ("\n*****  Instantiate EVSApplicationService *****");
45       System.out.println("\n***********************************
            ********************************");
46    }
47    else
```

```
48    {
49       System.out.println("\n***********************************
            *******************************");
50       System.out.println
            ("\n***** Instantiate LexBIGServiceImpl *****");
51       System.out.println("\n***********************************
            *******************************");
52       adapter = new DLBAdapter();
53    }
54       /**
55        * Set the vocabulary to the desired coding scheme.
56        */
57       Boolean retval = adapter.setVocabulary(codingScheme);
58    }
59
60    /**
61     * testGetTree()
62     */
63    public void testGetTree()
64    {
65       String rootname = "C25218";
66       boolean direction = true;
67       int levels = -1;
68       boolean isaflag = true;
69       Vector rolenames = new Vector();
70       rolenames.add("Technique_Has_Sample_Or_Specimen_Anatomy");
71
72       /**
73        * Call the getTree() method passing;
74        * 1. The root name
75        * 2. The direction
76        * 3. The number of levels of depth to return
77        * 4. The valid rolenames
78        */
79       DefaultMutableTreeNode dmtn = adapter.getTree
            (rootname, direction, levels, 0, isaflag, rolenames)
80       /**
81        * Print the resulting tree.
82        */
83       adapter.printTree(resultFile, dmtn);
84       }
85
86       /**
87        * testMetadata
88        */
89       public void testMetadata()
90       {
91          /**
92           * Set the name of the desired terminology
93           */
94          String urn = "NCI_Thesaurus";
95
96          /**
97           * Invokes the getMetadataProperties() method passing
               the terminoloy name (URN).
98           * Returns a NameAndValue array.
```

```
 99            */
100           NameAndValue[] nv_array =
                 adapter.getMetadataProperties(urn);
101
102           /**
103            * Loops through each returned array element and prints
                 the name and content values.
104            */
105           for (int j=0; j<nv_array.length; j++)
106           {
107              NameAndValue nv = (NameAndValue) nv_array[j];
108              System.out.println("CS: " + urn + "   Metadata Name:
                    " + nv.getName() + "   Metadata Value: " +
                    nv.getContent());
109              }
110           }
111
112           /**
113            * testMetadata()
114            */
115           public void testTreeTraversal(String codingSchemeName)
116           {
117              /**
118               * Get ALL root concepts
119               */
120              CodedEntry[] a = adapter.getRootConcepts();
121
122              /**
123               * Loop through each returned value and print.
124               */
125              for (int i=0; i<a.length; i++)
126              {
127                 CodedEntry ce = (CodedEntry) a[i];
128                 printNode(ce, 0);
129              }
130           }
131
132           /**
133            * printNode()
134            */
135           public void printNode(CodedEntry ce, int level)
136           {
137              /**
138               * Get the concept code for the coded entry
139               */
140              String code = ce.getConceptCode();
141
142              /**
143               * Get the subconcepts for the concept code
144               */
145              Vector subconcepts =
                    adapter.getSubConcepts(code, true, true);
146
147              /**
148               * Loop through each of the subconcepts
149               */
```

```
150             for (int j=0; j<subconcepts.size(); j++)
151             {
152                String subconceptcode = (String)
                      subconcepts.elementAt(j);
153
154                   /**
155                    * Execute the findConceptByCode() method and
                         print the node.
156                    */
157                   CodedEntry sub =
                         adapter.findConceptByCode(subconceptcode,
                         false);
158                   printNode(sub, level+1);
159                }
160             }
161
162             /**
163              * testMeta()
164              */
165             public void testMeta()
166             {
167                adapter.setVocabulary("NCI MetaThesaurus");
168                String code = "MTHU000096";
169                String source = "LNC";
170                   /**
171                    * Get properties by code
172                    */
173                   Vector v =
                         adapter.getPropertiesByCode("CL347198");
174
175                   /**
176                    * Find concepts with source code matching...
177                    */
178                   Vector v =
                         adapter.findConceptsWithSourceCodeMatching(
                         source, code, 1);
179
180                   /**
181                    * Find coded entries with source code
                         matching...
182                    */
183                   Vector v =
                         adapter.findCodedEntriesWithSourceCodeMatch
                         ing("NCI MetaThesaurus", "LNC",
                         "MTHU000096", 1);
184                   if (v != null)
185                   {
186                      /**
187                       * Loop through each of the coded entries
                            and print the concept code and name.
188                       */
189                      for (int i=0; i<v.size(); i++)
190                      {
191                         CodedEntry ce = (CodedEntry)
                               v.elementAt(i);
192                         System.out.println(ce.getConceptCode());
```

```
193                     }
194                 }
195             }
196
197             /**
198              * testSearchConcepts()
199              */
200             public void testSearchConcepts()
201                 int limit = 1000;
202                 String source = "MDR";
203                 boolean cui = false;
204                 boolean shortResult = false;
205                 boolean score = false
206                 String scheme = "NCI MetaThesaurus";
207                 String s = "artery";
208
209                     try {
210                         /**
211                          * Search all concepts in the
                              MetaThesaurus where the source is
                              MDR, the search term is 'artery'.
212                          */
213                         CodedEntry[] a =
                              adapter.searchConcepts(scheme, s,
                              limit, source, cui, shortResult,
                              score);
214                         int j = i+1;
215                         System.out.println("(" + j +") " +
                              ce.getConceptCode() + "[" +
                              adapter.getName(ce) + "]");
216                     }
217
218                     } catch (Exception e) {
219                         e.printStackTrace();
220                     }
221                 }
222             }
223         }
224         /**
225          * Main
226          */
227         public static void main(String[] args)
228         {
229             String name = "NCI_Thesaurus";
230             String version = "06.12d";
231
232             // Instantiate the Test class
233             Test test = new Test(name, version);
234
235             // Execute the testGetTree() method
236             test.testGetTree();
237
238             // Execute the testSearchConcepts() method
239             test.testSearchConcepts();
240
241             // Execute the testMetadata() method
```

```
242                 test.testMetadata();
243             }
244         }
```

APPENDIX A
RESOURCES USED TO CREATE THIS GUIDE

Books and Articles

- *The Description Logic Handbook*. Franz Baader, et al. (eds.). Cambridge University Press, 1993.
- *Artificial Intelligence*. Patrick Winston. Addison-Wesley, 1984.
- *Artificial intelligence*. Minsky M, Hillis D, Rudisch G. New England Journal of Medicine. 1980 Jun 26;302(26):1482.
- Java Programming: http://java.sun.com/learning/new2java/index.html
- Extensible Markup Language: http://www.w3.org/TR/REC-xml/
- XML Metadata Interchange: http://www.omg.org/technology/documents/formal/xmi.htm

caBIG Material

- caBIG: http://cabig.nci.nih.gov/
- caBIG Compatibility Guidelines: http://cabig.nci.nih.gov/guidelines_documentation

caCORE Material

- NCICB: http://ncicb.nci.nih.gov/NCICB/infrastructure
- caCORE: http://ncicb.nci.nih.gov/NCICB/infrastructure
- caBIO: http://ncicb.nci.nih.gov/NCICB/infrastructure/cacore_overview/caBIO
- caDSR: http://ncicb.nci.nih.gov/NCICB/infrastructure/cacore_overview/cadsr
- CSM: http://ncicb.nci.nih.gov/NCICB/infrastructure/cacore_overview/csm

Software Products

- Java: http://java.sun.com
- Ant: http://ant.apache.org/

APPENDIX B

ADDITIONAL EXAMPLES

This appendix provides two additional code examples:

- *Find Tree for Concept and Association* on this page
- *Search Metathesaurus for a Specific Concept/Search Term* on page 84

Find Tree for Concept and Association

This example shows how to locate a specific concept node and given association in the identified vocabulary hierarchy.

```
 1 /*
 2 * Copyright: (c) 2004-2007 Mayo Foundation for Medical
 3 * Education and Research (MFMER). All rights reserved.
 4 * MAYO, MAYO CLINIC, and the triple-shield Mayo logo are
 5 * trademarks and service makrs of MFMER.
 6 *
 7 * Except as contained in the copyright notice above, or as used
 8 * to identify MFMER as the author of this software, the trade
 9 * names, trademarks, service marks, or product names of the
10 * copyright holder shall not be used in advertising, promotion
11 * or otherwise in connection with this software without prior
12 * written authorization of the copyright holder.
13
14 * Licensed under the Eclipse Public License, Version 1.0
15 * (the "License"); you may not use this file except in
16 * compliance with the License. You may obtain a copy of the
17 * License at http://www.eclipse.org/legal/epl-v10.html.
18
19 * Modified By: NCICB
20
```

```
21 package org.LexGrid.LexBIG.example;
22
23 import org.LexGrid.LexBIG.DataModel.Collections
     .AssociationList;
24 import org.LexGrid.LexBIG.DataModel.Collections
     .NameAndValueList;
25 import org.LexGrid.LexBIG.DataModel.Collections
     .ResolvedConceptReferenceList;
26 import org.LexGrid.LexBIG.DataModel.Core.AssociatedConcept;
27 import org.LexGrid.LexBIG.DataModel.Core.Association;
28 import org.LexGrid.LexBIG.DataModel.Core.CodingSchemeSummary;
29 import org.LexGrid.LexBIG.DataModel.Core.
     CodingSchemeVersionOrTag;
30 import org.LexGrid.LexBIG.DataModel.Core.NameAndValue;
31 import org.LexGrid.LexBIG.DataModel.Core
     .ResolvedConceptReference;
32 import org.LexGrid.LexBIG.Exceptions.LBException;
33 import org.LexGrid.LexBIG.Impl.LexBIGServiceImpl;
34 import org.LexGrid.LexBIG.LexBIGService.LexBIGService;
35 import org.LexGrid.LexBIG.LexBIGService.CodedNodeSet
     .PropertyType;
36 import org.LexGrid.LexBIG.Utility.ConvenienceMethods;
37 import org.LexGrid.commonTypes.EntityDescription;
38 import gov.nih.nci.system.applicationservice.*;
39
40 /**
41  * Example showing how to determine a branch of associations,
    * starting from a specific concept code.
    */
42 public class FindTreeForCodeAndAssoc {
43    final static int MAX_DEPTH = 5;
44    public FindTreeForCodeAndAssoc() {
45       super();
46    }
47
48    /**
49     * Entry point for processing.
50     * @param args
51     */
52       public static void main(String[] args) {
53       if (args.length < 2) {
54          System.out.println("Example: FindTreeForCodeAndAssoc
               \"C25762\" \"hasSubtype\"");
55          return;
56       };
57
58       try {
59          String code = args[0];
60          String relation = args[1];
61          new FindTreeForCodeAndAssoc().run(code, relation);
62       } catch (Exception e) {
63          Util.displayAndLogError("REQUEST FAILED !!!", e);
64       }
65    }
66
```

```
67    public void run(String code, String rel)throws LBException {
68       String evsUrl = "http://evsapi.nci.nih.gov/evsapi40/http/
            remoteService";
69       CodingSchemeSummary css = Util.promptForCodeSystem();
70       if (css != null) {
71          LexBIGService lbSvc =
               EVSApplicationService.getRemoteInstance(evsUrl);
72          String scheme = css.getCodingSchemeURN();
73          CodingSchemeVersionOrTag csvt = new
               CodingSchemeVersionOrTag();
74          csvt.setVersion(css.getRepresentsVersion());
75          print(code, rel,0, lbSvc, scheme, csvt);
76       }
77    }
78
79    /**
80     * Handle one level of the tree, and recurse to the
81     * maximum depth.
82     * @param code
83     * @param relation
84     * @param depth
85     * @param lbSvc
86     * @param csvt
87     * @param scheme
88     * @param tagOrVersion
89     * @throws LBException
90     */
91    protected void print(String code, String relation, int depth,
         LexBIGService lbSvc, String scheme,
         CodingSchemeVersionOrTag csvt) throws LBException {
92
93    // Perform the query...
94    NameAndValue nv = new NameAndValue();
95    NameAndValueList nvList = new NameAndValueList();
96    nv.setName(relation);
97    nvList.addNameAndValue(nv);
98
99    ResolvedConceptReferenceList matches =
         lbSvc.getNodeGraph(scheme, csvt,
         null).restrictToAssociations(nvList,
         null).resolveAsList(ConvenienceMethods.createConceptReferen
         ce(code, scheme),true, false, 1, 1,null, new PropertyType[]
         {PropertyType.PRESENTATION},null, 1024);
100
101   // Analyze the result ...
102   if (matches.getResolvedConceptReferenceCount() > 0)
         {ResolvedConceptReference ref = (ResolvedConceptReference)
         matches.enumerateResolvedConceptReference().nextElement();
103
104   // Indent according to level
105   StringBuffer sb = new StringBuffer();
106   for (int i = 0; i < depth-1; i++) sb.append('\t');
107
108   // Print the associations
109   AssociationList sourceof = ref.getSourceOf();
110   Association[] associations = sourceof.getAssociation();
```

```
111     for (int i = 0; i < associations.length; i++) {
112        Association assoc = associations[i];
113        AssociatedConcept[] acl =
              assoc.getAssociatedConcepts().getAssociatedConcept();
114        for (int j = 0; j < acl.length; j++) {
115           // Print
116           AssociatedConcept ac = acl[j];
117           EntityDescription ed = ac.getEntityDescription();
118           Util.displayMessage("\t\t" +
                 ac.getConceptCode() + "/"+ (ed == null?"**No
                 Description**":ed.getContent()));
119
120           //Recurse
121           if (depth < MAX_DEPTH)
122              print(ac.getConceptCode(), relation, depth+1, lbSvc ,
                    scheme, csvt);
123           }
124        }
125     }
126 }
```

Search Metathesaurus for a Specific Concept/Search Term

This example has been commonly used by the caDSR Team. It shows how to search the Metathesaurus for a desired concept.

```
 1 import java.io.File;
 2 import java.lang.reflect.Method;
 3 import java.util.ArrayList;
 4 import java.util.Collection;
 5 import java.util.Enumeration;
 6 import java.util.List;
 7 import java.util.Vector;
 8 import java.util.jar.JarEntry;
 9 import java.util.jar.JarFile;
10
11 import gov.nih.nci.system.client.ApplicationServiceProvider;
12 import gov.nih.nci.system.applicationservice.*;
13 import gov.nih.nci.evs.query.*;
14 import gov.nih.nci.evs.domain.*;
15 import gov.nih.nci.evs.security.*;
16
17 public class TestcaDSR
18 {
19    TestcaDSR client = new TestcaDSR();
20    try
21    {
22       client.testSearch();
23    }
24       catch(Exception e)
25    {
26       e.printStackTrace();
27    }
28 }
```

```
29
30 public void testSearch() throws Exception
31 {
32    String termStr = "agent";
33    int metaLimit=100;
34    String metaSource = "*";
35
36    EVSApplicationService appService =
         (EVSApplicationService)ApplicationServiceProvider
         .getApplicationService("EvsServiceInfo");
37
38    try
39    {
40       EVSQuery evsQuery = new EVSQueryImpl();
41       evsQuery.searchMetaThesaurus(termStr, metaLimit,
            metaSource, false, false, false);
42
43       List metaResults = (List)evsService.evsSearch(evsQuery);
44
45       System.out.println("\n Running updated version....");
46       System.out.println("\n Results count = " +
            metaResults.size());
47       for (int i=0; i < metaResults.size(); i++) {
48          MetaThesaurusConcept metaConcept =
               (MetaThesaurusConcept)metaResults.get(i);
49          if ( metaConcept != null ) {
50             String conceptName = metaConcept.getName();
51             System.out.println("\n\t Concept Name = " +
                  conceptName);
52             String conceptID = metaConcept.getCui();
53             System.out.println("\n\t Concept ID = " +
                  conceptID);
54             ArrayList semantic =
                  metaConcept.getSemanticTypeCollection();
55             System.out.println("\n Semantic size = " +
                  semantic.size());
56             for (int ii=0; ii < semantic.size(); ii++) {
               System.out.println("\n\t Semantic = " +
                  (SemanticType)semantic.get(ii));
57             }
58          }
59       }
60    } catch(Exception e) {
61    System.out.println(">>>"+e.getMessage());
62       e.printStackTrace();
63    }
64 }
```

GLOSSARY

This glossary defines acronyms, abbreviations, and terminology used in this guide.

Term	***Definition***
AJAX	Asynchronous JavaScipt and XML
AL	Attributitive Language description logic
BioPortal	The NCBO Web browser for LexBIG
caBIG	cancer Biomedical Informatics Grid: A full cycle of integrated cancer research
caBIO	cancer Bioinformatics Infrastructure Objects: The model and architecture that serve as the primary programmatic interface to caCORE.
caCORE	cancer Common Ontologic Representation Environment
caDSR	cancer Data Standards Repository: Metadata registry based on the ISO/IEC 11179 standard and used to register the descriptive information needed to render cancer research data reusable and interoperable.
CLM	Common Logging Module: Provides a separate service under caCORE for audit and logging capabilities.
CSM	Common Security Module: Provides a flexible solution for application security and access control for caCORE.
CUI	Concept Unique Identifier
DAML	Agent Markup Language
DARPA	Defense Advanced Research Projects Agency
DL	Description Logic: A family of systems that is especially well suited to the development of ontologies, taxonomies, and controlled vocabularies.
DTS	A proprietary server that does not allow exposure of the API.
EVS	Enterprise Vocabulary Services: A collaborative effort of the NCI Center for Bioinformatics.
FL	Frame Language description logic
FOL	First-Order Predicate Logic
GO	Gene Ontology™ Consortium

Term	*Definition*
ICD-O-3	Additional vocabularies that are external to the NCI Metathesaurus
IDE	Integrated Development Environment
LexBIG	Open source public domain terminology server developed by the Mayo Clinic as part of the caBIG program
MDA	Model Driven Architecture
MedDRA	Additional vocabularies that are external to the NCI Metathesaurus
MGED	Microarray and Gene Expression Data
MO	Native MGED Ontology that is edited in OilEd and distributed in the Defense Advanced Research Projects Agency, Agent Markup Language, and Ontology Inference Layer XML format
NCBO	National Center for Bioontologies
NCI Metathesaurus	Comprehensive biomedical terminology database based on the National Library of Medicine’s Unified Medical Language System Metathesaurus (UMLS), supplemented with additional cancer-centric vocabulary.
NCI Thesaurus	A core reference terminology and biomedical ontology developed by the EVS in response to a need for consistent, shared vocabularies among the various NCI projects and initiatives and in the entire cancer research community.
NCICB	NCI Center for Bioinformatics
NLM	National Library of Medicine
OBO	Open Biomedical Ontologies
OIL	Ontology Inference Layer
OLLT	Obsolete Lower Level Terms
OWL	Web Ontology Language
RDF	Resource Description Framework
RRF	Rich Release Format
SDK	Software Development Kit: the caCORE SDK is a data management framework designed for researchers who need to be able to navigate through a large number of data sources. caCORE SDK is NCICB's platform for data management and semantic integration, built using formal techniques from the software engineering and computer science communities.
SNOMED	Additional vocabularies that are external to the NCI Metathesaurus
SOAP	Simple Object Access Protocol
SOC	System Organ Class
SSC	Special Search Categories
SUI	String Unique Identifier: assigned to each unique concept name or string in the NCI Metathesaurus.
TDE	Terminology Development Environment

Term	***Definition***
UML	Unified Modeling Language
UMLS	Unified Medical Language System Metathesaurus developed by the National Library of Medicine.
UMLS Semantic Network	An independent construct that provides consistent categorization for all concepts in the UMLS Metathesaurus and defines a useful set of relationships among those concepts.
W3C	World Wide Web Consortium: standards organization that develops interoperable technologies for the Web.
XML	Extensible Markup Language: A subset of the Standard Generalized Markup Language (SGML). Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML

Index

A

APIs
- Distributed LexBIG 67 - 71
- Java 34 - 35, 41 - 46
- Web Services 55 - 61
- XML-HTTP 62 - 64

C

caCORE
- architecture overview 27
- components 28 - 29
  - caBIO 28
  - caDSR 28
  - Common Logging Module (CLM) 29
  - Common Security Module (CSM) 29
  - EVS 28
- release paradigm 29

D

data sources for EVS 41

description logic
- in NCI Thesaurus 14
- overview 12

Distributed LexBIG Adapter example 71

Distributed LexBIG API
- architecture 67
- aspect oriented programming proxies 68
- documentation 69
- example of use 69
- installation and configuration 69
- LexBIG annotations 68
- overview 67

domain object catalog 40

E

EVS
- as caCORE component 28
- client technologies 33
- core components 38
- data sources 41
- domain object catalog 40
- key terminologies 8
- object model 39
- overview 7
- search paradigm 47
- software packages 34
- system architecture 31

EVSQuery
- accessing secured vocabularies 50
- creating search request 51
- examples of use 51 - 54
- methods and parameters 48
- overview 48

EVSQueryImpl, overview 48

F

first-order predicate logic 11

G

Gene Ontology (GO)
- mapping to Ontylog 21

J

J2SE, used with EVS 33

Java API 41 - 47
- domain packages 34
- downloading 42
- installing 43
- software requirements 42
- system packages 35
- verifying installation 45

K

kinds in NCI Thesaurus 18

knowledge representation
- description logic 12
- first-order predicate logic 11
- overview 10

L

LexBIG
- overview 25
- service API 32

M

MedDRA, mapping to Ontylog 22
MGED ontology, mapping to Ontylog 24

N

NCI Metathesaurus, overview 8
NCI Thesaurus
- concept edit history 15
- description logic use 14
- downloading 16
- overview 8
- OWL encoding 18

O

object model for EVS 39
Ontylog mappings
- Gene Ontology (GO) 21
- MedDRA 22
- MGED ontology 24
- OWL 19

OWL in NCI Thesaurus 18

R

Resource Description Framework (RDF)
- in NCI Thesaurus 18
- Ontylog name conversion 19

roles in NCI Thesaurus 18

S

search paradigm 47
Semantic Web and OWL
SOAP, used with EVS 33

U

UMLS Metathesaurus, about 9

W

Web Ontology Language
- *See* OWL in NCI Thesaurus

Web Services API
- configuration 55
- considerations 57
- examples of use 58 - 60
- limitations 61
- operations 56
- overview 55

X

XML-HTTP API
- examples of usel 63
- service location and syntax 62
- working with result sets 64

XML Utility Class
- example of use 65
- overview 65