Lexicon Query Service Specification

New Edition: June 2000
Version 1.0

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form at *http://www.omg.org/library/issuerpt.htm*.

Contents

Contents

Contents

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

Associated OMG Documents

In addition to the CORBA Transportation specifications, the CORBA documentation set includes the following:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBA Languages,* a collection of language mapping specifications. See the individual language emapping specifications.
- *CORBAservices: Common Object Services Specification,* a collection of OMG's Object Services specifications.
- *CORBAfacilities: Common Facilities Specification,* a collection of OMG's Common Facility specifications.
- *CORBA Manufacturing*: Contains specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
- *CORBA Med*: Comprised of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
- *CORBA Finance*: Targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Telecoms*: Comprised of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- 2AB
- 3M Health Information Systems
- Ardent Software, Inc.
- Care Data Systems, Inc.
- CareFlow/Net, Inc.
- FUJITSU LIMITED
- HBO & Company
- HealthMagic, Inc.
- HUBlink, Inc.
- IBM Corporation
- IDX Systems Corporation
- INPRISE Corporation
- IONA Technologies PLC
- Oacis Healthcare Systems
- Object Design, Inc.
- Objectivity, Inc.
- Oracle Corporation
- Persistence Software, Inc.
- Protocol Systems, Inc.
- Secant Technologies, Inc.
- Sholink Corporation
- Sun Microsystems, Inc.
- Versant Object Technology Corporation

Service Description 1

Contents

This chapter contains the following topics.

1.1 Overview

The scope of this specification is to specify a set of common, read-only methods for accessing the content of medical terminology systems. What constitutes a medical terminology system can vary widely, from a simple list consisting of a set of codes and phrases at one extreme, to a dynamic, multi-hierarch classification and categorization scheme at the other. The focus was to determine what could be construed to be “common” elements of terminology systems. By “common,” we mean the set of elements in which the semantics are fairly widely accepted, even though they may not be present in all or even many of the terminology systems available today. The goal was to produce a specification that could be used to implement a reasonable and useful interface to any of the major medical coding schemes.

A key goal of this specification was to provide a single, agreed-upon way to ask a question of a terminology system. Terminology systems may vary radically in their forms of representation and access. For example, the question “Is penicillin an antibiotic?” could be presented to one system in the form “Does there exist a subtype relationship in which the concept code for antibiotic is the supertype and the concept code for penicillin is the subtype?” In another system, the question may be presented as “Is there a record in the drug database whose key is ‘penicillin’ that has the value of ‘Yes’ in the antibiotic column?”

The intention of this specification is to provide only one specific interface that may be used to answer any question, regardless of the underlying implementation.

This specification provides read-only services. Read-only services have been further subdivided into two categories: 1) High volume on-line services and 2) Perusal and browsing services. This specification focuses on the first category of services which are services used by an on-line production system. The services include translation, inference, presentation, and the like. The second category was addressed only as necessary to satisfy specific RFP requirements.

1.2 Use Scenarios

The following scenarios describe some of what are believed to be typical uses of terminology systems. This list of uses is not exhaustive. However, it has served as a guideline in designing the terminology service interface. The set of uses is subdivided into six sections:

1. **Information Acquisition -** Using terminology services to aid in the process of entering coded data.
2. **Information Display -** Using terminology services to translate coded data elements into human or machine-readable external forms.
3. **Mediation -** Using terminology services to transform messages or data records from one form or representation into another.
4. **Indexing and Inference -** Using terminology services to inquire about associations which may or may not pertain between various data elements and to assist in the location of various data record sets, which may contain information relevant to the specific topic or entity.
5. **Browsing -** Using the terminology services to determine the structure and meaning of a terminology system.
6. **Composite Concept Manipulation -** Using the terminology services to aid in the entry, validation, translation, and simplification of composite concepts.

Each of these sections is dealt with in greater detail in the paragraphs that follow.

1.2.1 Information Acquisition

A key factor in coding data is the ability to quickly and precisely translate an external term, phrase, or image in the user's mind into the code or codes that represent the information to be conveyed. The terminology services provide several means of assisting in this translation process.

1.2.1.1 Text Lookup

The data-entry application knows the specific code that represents a target concept. The terminology service receives the code and coding scheme from the application and returns the preferred phrase that represents the specific code.

Example: A user wishes to encode the fact that a patient has a mild atrial flutter. The user is familiar with the ICD-9 coding system and wishes to start with the code 427. The terminology service is asked for the phrase that corresponds with the code 427 in the ICD-9 coding scheme and returns the text "Cardiac Dysrhythmias."

1.2.1.2 Phrase Lookup

The user of the data-entry application knows the precise text that represents a target code or set of codes. The user supplies the string and the name of the target coding scheme to the terminology service. The service returns the code or codes whose presentations match the string, along with a list of the preferred presentations for each code.

The user may constrain the search by supplying a list of contexts in which the text is used.

Example: An application can be used to locate all concepts that correspond to the text "Cold" in the UMLS coding scheme. It supplies the text "Cold" and the language indicator for "English" to the terminology services and asks for all matching concepts in the UMLS coding scheme. The terminology system returns two concepts:

C0009443 - Common Cold

C0009264 - Cold <1>

The user can limit the search by supplying the information that the resulting concepts must belong to the domain "Enterovirus Infections," which would then constrain the output to the single concept representing the disease. The user could also indicate that the supplied text was used as a short column heading, which might further constrain the results.

1.2.1.3 Phrase Matching

This case is identical to the previous case, except the user knows only an approximate string that represents the target concept(s). The string may contain wild cards and other information to direct the search. With the exception of the string itself, the information supplied to the terminology service is identical to the case described above.

The terminology service returns a set of concepts that might match the supplied phrase. The set is ordered by match likelihood with the most probable matches supplied first.

Example: The user could supply the string “Atr* Fibrillation” and the language indicator for “English” to the terminology service and ask for all matching concepts in the UMLS coding scheme. The terminology service might return the following set of concepts:

Code	Phrase	Weight
C0155709	Atrial fibrillation and flutter	0.9
C0004238	Atrial Fibrillation	0.5

1.2.1.4 *Keyword Matching*

This case is identical to the previous two cases, with the exception that the user knows one or more keywords, which are used to locate the target concept(s). For example: the user could know the words “heart,” “valve,” and “flutter.” The terminology service is asked for matching concepts in the UMLS coding scheme and returns the appropriate set of matching codes.

1.2.1.5 *Code Refinement*

The user of the services wishes to determine the best possible concept code to represent a specific situation (e.g., the condition of a patient). The user first selects a starting concept through some other mechanism. Given this starting concept, the user wishes to supply additional words or phrases, along with a relationship (more general, more specific, synonymous). The terminology service returns an ordered set of concept codes that participates in the supplied relationship with the starting code, as well as contains the additional words or phrases in their external representation. As with phrase lookup, the order of the set is based on match quality with exact matches occurring first.

Example: The user may start with the concept for <Cardiac Dysrhythmias> in the ICD-9 coding system. The terminology services are supplied with this concept, the words “atrial” and “flutter,” and the relationship “more specific.” The coding system would return an ordered set of concepts which partially or completely meet these criteria, and the user could select the concept (and code) which most closely represented his own image of the situation (which, in this case, would probably be code 427.32, Atrial Flutter).

1.2.1.6 *Possible Value Enumeration*

The user wishes to examine a list of the possible codes, along with their corresponding phrases or other external representations that might be supplied for a specific data-entry field.

The terminology service is supplied with the value domain representing the data-entry field, the coding scheme to be used in the field, and the language and usage context in which the selection list is to be presented. The terminology services return a list of codes and the appropriate presentation for each code in the given context. The return list may also contain indicators showing which code(s) are defaults for the selection.

Example: The user wishes to encode the patient's gender into an HL7 message, using the HL7 Version 2.3 Table 001. The terminology service is given the "domain" identifier CX0001234, representing the concept *Patient Sex Value Set.* It is also given the information the user is interested in as an English selection set for use in a short, textual list.

The terminology services return a list containing the following values:

Table 1-1 Example of a Possible Value List

Code	Presentation Text
M	Male
F	Female
U	Unknown
O	Other

1.2.1.7 Field Validation

The application needs to determine whether a specific code for a field in a data record or message is valid. The application passes the code, the value domain, and the coding scheme to the terminology service. The service returns TRUE if the code is valid for the field; FALSE otherwise.

Example: An application has just received an HL7 message and needs to determine whether it is valid and can be processed further. As the application iterates over the various coded entries within the message, querying the terminology service about the validity of the entries, it encounters the field in which the patient gender is encoded. It passes the concept representing the gender domain "CX0001234," the code itself "M," and the identifier of the HL7 Version 2.3 Table 001 coding scheme to the terminology service. The service returns TRUE indicating that "M" is a valid code for the specified domain in the coding system.

1.2.1.8 Pick List Generation

The application can present a formatted, ordered list of possible selections for an input field. The specific user, facility, or some other factor may customize this list. The specific list to be returned may depend upon the application requesting it, the specific user, and other context-specific information. The list may also need to specify which of the selections may be considered as default selections.

1.2.2 Information Display

It is necessary to be able to translate a code from a coding scheme into the appropriate form for presentation to an external viewer. This translation may have as its target a printout, a video display screen, a sound generation device, or some other medium. The display will depend upon the application doing the presentation, the target medium (screen, hard copy, etc.), the language spoken by the viewing user, and, possibly, the user's identity and classification.

1.2.2.1 Code Translation

An application wishes to represent the meaning of a specific code from a specific coding scheme to an end-user. The terminology service is given a code and the observer's preferred language, and, possibly, additional context describing how the concept is to be presented. It returns the most appropriate presentation of the concept, given the situation.

Example: An application may have a data record that contains the code "123" in the laboratory test field. The application wishes to display a small (< 30 characters) textual string representing that particular code to an English-speaking person. The terminology service is given the code, its coding scheme, the code for the English language, and an ordered list of contexts (e.g., 30-character string, heading, abbreviation, textual name) defining an acceptable presentation. The services return the most reasonable match, if any.

1.2.3 Mediation

One of the "holy grails" of terminology services is the ability to translate coded information from one database and/or coding system into another, independently developed, database and coding system. While the terminology services will not provide this capability by any stretch of the imagination, they should provide some building blocks upon which more complete information translation may be based. The use cases below describe some of these building blocks.

1.2.3.1 Code Transformation

Perhaps the simplest situation in mediation is that of two different systems using different coding schemes to represent identical information. While there may not necessarily be a 100% mapping in either direction, it is possible to supply the terminology service with a code, a source coding scheme, a target coding scheme, and, optionally, a specific value domain and have the service *transform* the code from the source code into the target code.

Example: An external system may encode all of the coded values in a message (e.g., gender, location, laboratory test identifier, etc.) as positive integers, with each domain beginning at the number 1. In this case, each of the specific value domains represents a separate coding scheme. For example, the gender coding scheme might represent "Male" with a 1, "Female" with a 2, etc. Similarly, the system might encode laboratory tests with "Serum Creatinine" as 1, "Serum Chloride" as 2, etc. In this situation, it is

possible to construct a cross-scheme map which transforms the gender coding scheme above into the HL7 Table 1 coding scheme and transforms the laboratory test scheme above into the LOINC coding scheme.

1.2.3.2 Code Mapping

It is not uncommon for two different systems to use different coding schemes to represent *similar* information. In this situation, there may not be exact alignment between the codes. Also, a code within one system may represent more specific information than a code in another system. In certain situations, the user may determine that this imprecision is acceptable and may use the terminology service to determine the closest match for a given code between systems.

1.2.3.3 Structural Composition/Decomposition

Some terminology systems provide the capability to represent a concept as a set of related concepts within the same coding scheme. This provides the rudiments of the capability to change the structural representation of a concept between databases. A simple example of this situation might appear in the representation of a urine drug screen. One system may treat this test as a set of {drug, result} tuples, with the domain of drug being {Amphetamines, Cannabinoids, Cocaine, Methamphetimines} and the domain of result being {pos, neg}. A second system might treat the test as a set of domains {AmphetimineUrineScreen, CannabinoidsUrineScreen, CocaineUrineScreen}, with each domain having a {pos, neg} value set. A third system might treat the same test as a set of {drug}, where the presence of a drug in the list implies a positive result.

There are countless variations on this theme, and while a terminology service may assist in these transformations, the ability to do this sort of transformation is still well outside the scope of this specification.

See Section 1.2.6, “Composite Concept Manipulation,” on page 1-9 scenario for more detail.

1.2.3.4 Field Validation

Verify that a specific presentation is a valid value for a specific domain within a domain usage context. See Section 1.2.1.7, “Field Validation,” on page 1-5 for more detail.

1.2.4 Indexing and Inference

The presence of coded information within a computerized system provides an opportunity to augment the system with “decision support” software. This class of software examines information entering the system, looking for situations which may warrant additional action, such as posting alerts, generating additional data. One example of such a system is a drug/drug interaction monitor that examines all incoming drug orders looking for potential adverse reactions with current patient medications.

This type of software can use terminology services to make “inferences” about information in incoming records and to assist in locating information in existing databases.These cases are described below.

1.2.4.1 Relationship Inquiry

Decision support programs are often written in terms of classes of concepts and they must be able to inquire about associations between various classes. For example, a particular module may be written to scan new drug orders looking for drugs that belong to the class or classes recorded in patient allergy records. A patient allergy record might contain the fact that the patient is allergic to Penicillin. The decision support program would supply the code for an ordered drug (e.g., Pen-VK) to the terminology service, along with its allergy class (e.g., Penicillin) and ask the service whether a “hasSubtypes” relationship exists between these entities. If the response is TRUE, the application would then take appropriate action to warn the patient or pharmacist of the potential problem.

1.2.4.2 Data Element Location

A decision support application might need to locate existing information in a database based on some classification scheme. If we extend the example above, a second decision support program might be written to examine all patient allergy entries. If a record is entered showing the patient to be allergic to Penicillin, the decision support program might wish to scan the database looking for all references to orders for Penicillin.

1.2.5 Browsing

1.2.5.1 Service Browsing

A user wishes to determine which coding schemes, value domains, and versions of each are supported by the terminology service. For each version of each coding scheme, the user wishes to determine which languages, sources, usage contexts, presentation formats, and other coded data properties are supported by that version. The user also wishes to determine what systemizations are supported by that version of the coding scheme, and what association types are represented within the systemizations. For each value domain version, the user wishes to determine which coding schemes have concepts in that value domain.

1.2.5.2 Concept Attribute Discovery

A user wishes to discover all of the concept codes supported by a specific coding scheme, and by a particular version of that coding scheme. Within the context of a coding scheme version, the user wishes to discover all presentations, definitions, comments, and instructions associated with a code.

For a given systemization, the user wishes to determine in which association types a concept code participates, the role that it plays in each, and what entities are associated with the concept.

1.2.5.3 Association Discovery

In a given systemization, the user wishes to list all of the association types that participate in the systemization and the behavioral characteristics of each.

1.2.6 Composite Concept Manipulation

1.2.6.1 Concept Attributes Retrieval

The user has selected a specific concept from a coding scheme and needs to know which additional attributes and values may be applied to the concept. The terminology service is given the concept and the coding scheme. It returns a set of attribute-value pairs that further define the characteristics of the concept.

1.2.6.2 Composition

The user has identified a composite concept consisting of two or more concepts and a hierarchy of relationships between them (e.g., <inflammation> hasLocation <liver>), and wishes to translate this concept into a single, closely matching code (e.g., <hepatitis>). The terminology service is passed the set of hierarchical relationships and, ideally, returns a single node - a code that represents the simplified translation of the passed concept.

1.2.6.3 Decomposition

The user has a single, complex concept that he wishes to see represented in terms of a set of relationships among the concepts that form the underlying composition. The composite concept, along with the context of the desired translation, is passed into the terminology services and a hierarchical structure representing the group of concepts is returned.

1.2.6.4 Normalization

The user wishes to see a composite concept represented in the simplest canonical form. The terminology service is passed the set of hierarchical relationships and concepts, and returns another hierarchy that represents the canonical form of the set of concepts.

1.3 Reference Model

This section describes the reference model for the terminology services. The purpose of this model is to define the various entities that appear in the IDL service definition. It is not intended to describe how a terminology system should be structured or built.

It is highly unlikely that any single terminology system implementation will contain all of the elements described in the following sections. The model represents a “common union” of the aspects of several different systems. Much of the functionality is optional and should be implemented only if it makes sense for the particular terminology system. It is the intention of this model to provide a model that describes a broad spectrum, from a simple system consisting of a list of codes and phrases to a significant portion of a sophisticated “third generation” terminology system.[8]

Be aware that there will not always be an obvious, direct mapping between the reference model and the IDL itself. This is because the reference model attempts to describe *what* is being done in the terminology service, where the IDL describes *how* it is accomplished. Many factors, including naming conventions, compiler restrictions, performance considerations, pre-established styles, etc., may cause the IDL description of the interface to look substantially different from the corresponding entity described in the model.

Note – Appendix C contains a brief description of the graphic notation encountered in the following sections.

1.4 Model Overview

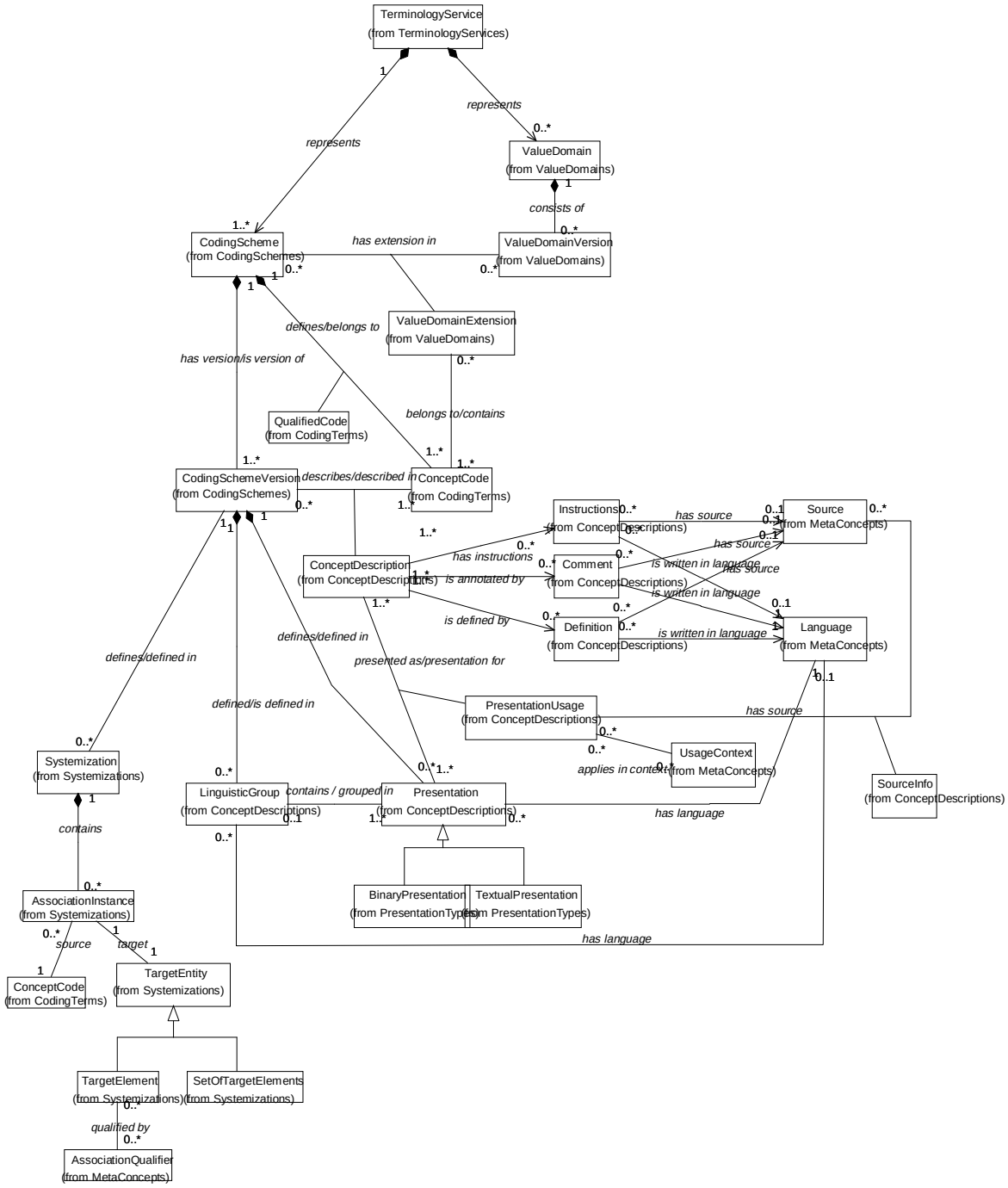


Figure 1-1 Model Overview

Figure 1-1 presents a general overview of the entities described in the sections that follow. It is included to provide a reference point when viewing individual diagram sections below.

1.5 Data Type Definitions

1.5.1 Basic Types

The classes, as shown in Figure 1-2, represent the data type extensions that are used in this model. These extensions are in addition to the basic data types described in the OMG's Object Management Architecture (OMA). [8]

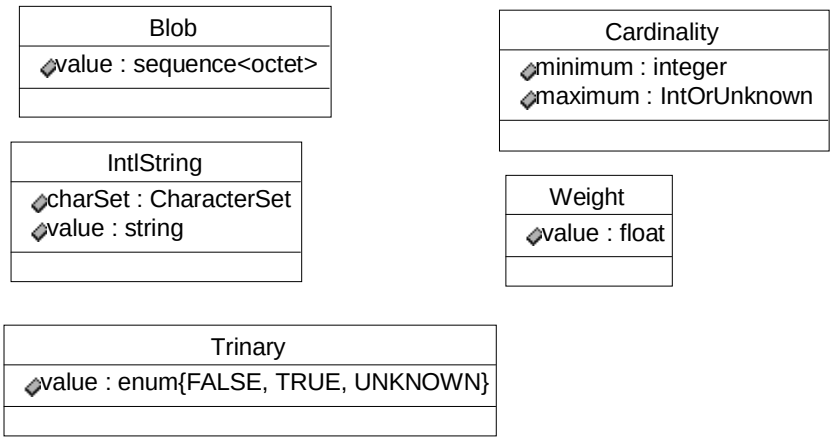


Figure 1-2 Basic Types

1.5.1.1 Blob

An opaque stream of bytes which is unaltered by the service or transport. The blob data type is used to carry non-textual presentations and machine-readable instructions. A typical use of this data type would be to return a sound bite of a spoken word.

1.5.1.2 IntlString

A string which is intended to be presented using the supplied character set. IntlStrings occur in places where the language of the string is not necessarily that of the user of the system, and automatic conversion of the string to the character set of the viewer's native language would not be useful or desirable.

1.5.1.3 Trinary

A type which represents three possible values. This type is returned by operations which must be able to indicate that there is insufficient information to answer a question as well as the more traditional TRUE and FALSE results.

1.5.1.4 Weight

A relative measure of the “closeness” of a match. The range of the value of a weight is $0.0 \leq \text{value} \leq 1.0$. Weights have no absolute meaning, and may only be compared with other weights that are returned as a sequence from the same method invocation.

1.5.1.5 Cardinality

An entity representing minimum and maximum possible occurrences of an element in an association. The minimum value must be a non-negative integer and maximum value must be either a positive integer or a special token representing “unknown.”

1.5.2 Naming Authority

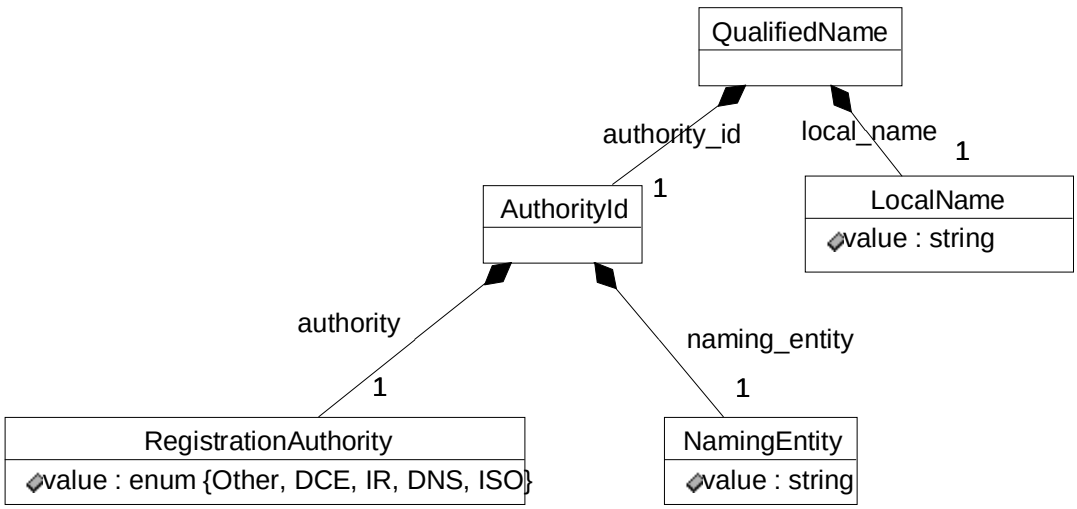


Figure 1-3 Naming Authority

Naming authorities provide a means of giving globally unique names to name spaces and hence the names within those name spaces.

1.5.2.1 RegistrationAuthority

Identifies the root of the name space authority. An entity (e.g., person or organization) may be registered with multiple different roots (RegistrationAuthorities) and be able to assign names and other name spaces within each root. These may be used for the same or for different needs. For this reason there is no guarantee of any equality in the different name spaces managed by an entity. There are currently no means available to determine whether a given authority in an ISO hierarchy is the same authority as one specified in a DNS hierarchy.

Other

This form of a naming authority should be used sparingly, and only in experimental or localized situations. It is the responsibility of the implementing institution to guarantee uniqueness within the names themselves, and there is no uniqueness guarantee outside of the source institution. Services that define default naming authorities (and possibly also names) may also use the Other root to forego long AuthorityIds. In this case, the specification of the service must name AuthorityIds that may be expected with the Other root and still maintain name space integrity.

ISO

International Standards Organization. [9] The ISO specifies a registration hierarchy, identified by a series of named/numbered nodes. Many of the coding schemes used in the medical environment are or can be registered within the ISO naming tree. The ISO root form is one of the recommended forms when the naming authority is internationally recognized, such as international coding schemes, or when the authority is to be used across two or more different enterprises. ISO provides for the recording of a responsible person and address for each node in the authority hierarchy.

DNS

Domain Name Services. [10] Internet domains are recorded with a central, global registration authority. Subhierarchies within the domains are then maintained locally by the registered organization or person. The DNS form is recommended as an alternative to the ISO naming tree when the specific naming authority needs identity and uniqueness, but is not in an ISO registration. By using this common characteristic of many organizations it gives the ability to create globally unique name spaces and names without the need to register as an ISO name authority. It is up to the organization itself to maintain the integrity of the name space(s) (e.g., not reusing names or name spaces).

IR

The OMG Interface Repository. [11] The CORBA Architecture specifies a means of uniquely identifying entities within the interface repository, via the use of a *RepositoryId*. CORBA repository id's may be in either the OMG IDL format, the DCE UUID format, or the LOCAL format. Within this specification, the "IR" root refers only to the IDL format. The DCE format may be represented within the DCE root and the Local format within the Other root. The IDL authority may prove very useful when registering CORBA/IDL-specific objects such as value sets, interface specifications, etc. It should be noted that OMG does not currently manage the repository name space in any rigorous fashion, and it is quite possible that two different developers may arrive at exactly the same repository id for entirely different entities. For this reason some people give the repository id a prefix that consists of their reverse DNS that is '/' separated instead of '.' separated. This root type may be very useful when the names within the name space are defined in IDL. For example it could be the RepositoryId for an enumerated type or a module that has constant integers or strings defined for each name within the name space.

DCE

The Distributed Computing Environment. [12] While they don't actually register coding schemes or other entities, they do provide a means of generating a globally unique 128-bit identifier, called a Universally Unique Id (UUID). This UUID may be used to guarantee the uniqueness of a name space in situations where it is not necessary for the identity of the authority to be known outside of the specific implementation.

1.5.2.2 NamingEntity

Identifies a specific name in the syntax and format specified by the corresponding registration authority. The various naming authorities tend to provide a fair amount of leeway as far as the actual format of the registered names. As there may be situations where the full semantics of a specific authority's name comparison will not be available to an application, we have chosen to select a specific subset of the syntax of each representation. The intention is to be able to determine whether two registered entities are identical or not solely through the use of string comparison. The specific name formats are described below:

Other

An arbitrary string, syntax undefined locally by a specific service specification and/or by particular implementations and installations. The colon ":" character is illegal to use as it is reserved as a separator of components in the stringified version of **AuthorityId** and **UniqueName**.

ISO

The name should be represented using the *NameForm* of the *ObjectIdentifierValue* as specified in ISO/IEC Recommendation 8824-1.10 Each name component should be separated by a single space.

Example: "joint-iso-ccitt specification characterString"

DNS

The domain name and path in the form mandated in RFC 1034.12 The path name is represented as a dot separated tree which traverses up the hierarchy. Since DNS names are not case-sensitive, only lower-case letters should be used so that simple string comparisons can determine equality. However, it is okay to use case-insensitive comparisons as well.

Example: "pidsserv.slc.mmm.com"

IR

The OMG RepositoryId format specified in the CORBA Architecture V2.0 manual, in the form: "<node>/<node>/Ö/<node>". The "IDL:" prefix and the version number suffix should NOT be used in this format.

Example: "CosNaming/NamingContext/NotFoundReason"

DCE

The UUID in the external form <nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn>, where <n> represents one of the digits 0-9 and the characters A-F. The alpha characters should all be upper case.

Example: 6132A880-9A34-1182-A20A-AF30CF7A0000”

1.5.2.3 AuthorityId, AuthorityIdStr

The combination of a Registration Authority and Naming Entity which identifies a specific naming authority. In situations where a given naming entity may have more than one naming authority, it should be agreed upon in advance which of the specific names for the entity is to be used. This specification makes no guarantees about the ability to recognize, for example, that an authority in the ISO structure is identical to an authority within the OMG structure.

The string version (**AuthorityIdStr**) is useful for situations where unique names are required in a string format. The string is created as <stringified **RegistrationAuthority**>:<**NamingEntity**>. The stringified **RegistrationAuthority** is given by the following:

Other	->	“”
ISO	->	“ISO”
DNS	->	“DNS”
IR	->	“IDL”
DCE	->	“DCE”

The names are short to make string comparisons quick.

1.5.2.4 LocalName, UniqueName, UniqueNameStr

A local name is a name within (relative to) a namespace. It is simply a string representation.

A **UniqueName** is a globally unique name for an entity by the fact that it carries the naming **AuthorityId** of the name space and the **LocalName** within that name space.

The **UniqueNameStr** is a stringified **UniqueName**. The format of the string is <stringified **RegistrationAuthority**>:<**NamingEntity**>:<**LocalName**>. Notice that even though the colon character “:” cannot be used for **NamingEntity**, it can be used for the **LocalName**.

1.5.3 Basic Identifiers

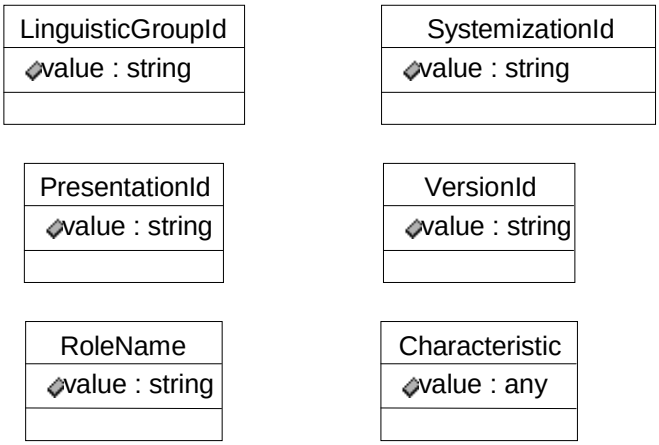


Figure 1-4 Basic Identifiers

1.5.3.1 LinguisticGroupId

The unique identifier of linguistic group within the context of a coding scheme version.

1.5.3.2 PresentationId

The unique identifier of presentation within the context of a coding scheme version.

1.5.3.3 RoleName

A string which serves as a synonym for either the “source” or the “target” portion of the ordered pair of types in an association. As an example, the hasComponents association has the **RoleName** “composite” as a synonym for the source type and “components” as a synonym for the corresponding target type.

1.5.3.4 SystemizationId

The name of a specific categorization or organization of concepts within a version of a coding scheme. A systemization id is unique within the context of a coding scheme version.

1.5.3.5 VersionId

The unique identifier of a specific version of a coding scheme or value domain. There is no implied ordering on version identifiers. A version identifier may be composed of both letters and digits, and must be unique within the context of the coding scheme or

value domain. **VersionId** has one distinguished value *DEFAULT* which represents the "production" or the latest validated version of the specific entity that is ready for use. The *DEFAULT* version of an entity need not be the most recent.

1.5.3.6 Characteristic

A characteristic represents a non-coded "property" or "attribute" which is associated with a concept code. A characteristic may be of any type other than a concept code.

1.5.4 Terminology Identifiers

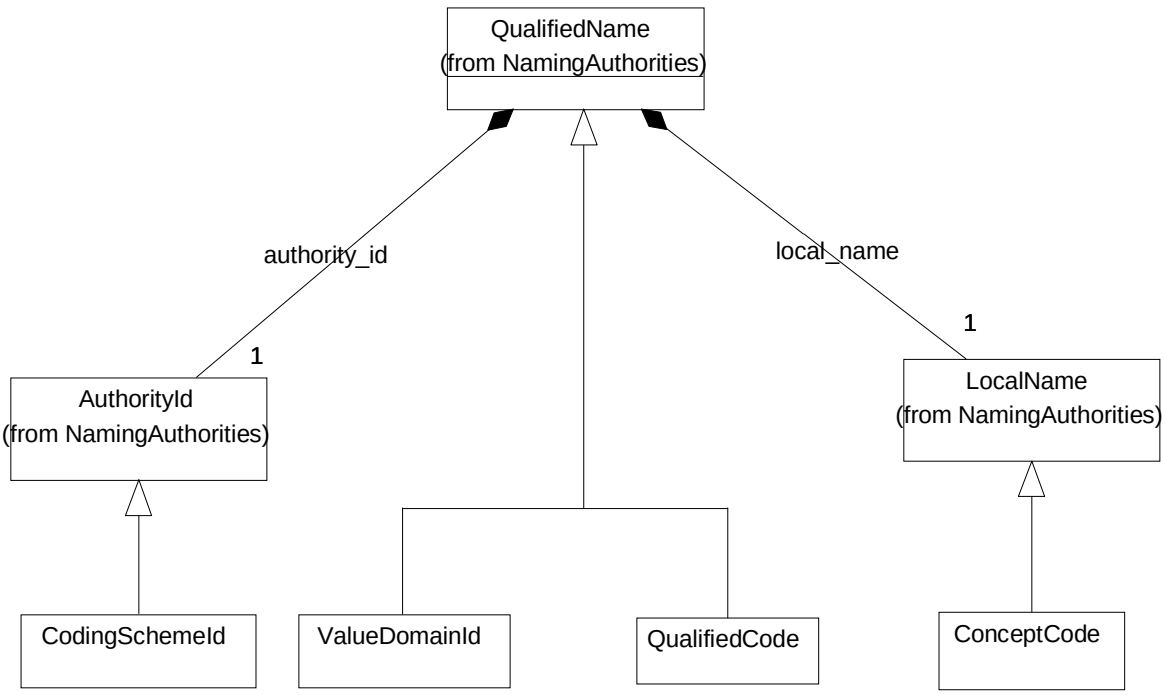


Figure 1-5 Terminology Identifiers

1.5.4.1 CodingSchemeId

A **CodingSchemeId** is a globally unique identifier of a specific coding scheme, a coding scheme is a naming authority that manages a set of concept codes (local names) within its name space.

1.5.4.2 ValueDomainId

A **ValueDomainId** is a globally unique identifier of a specific value domain. The identifier consists of a naming authority and the actual domain name. The naming authority provides a unique name space, while the domain name identifies a given field in a message, a column in a database, an entry field on a screen or some other data value which may contain coded information.

1.5.4.3 QualifiedCode

A globally unique concept code formed by combining the coding scheme id and the local concept code within that coding scheme.

1.5.4.4 ConceptCode

An arbitrary string that identifies a unique entity within a given coding scheme. The coding scheme forms the naming authority and the concept codes is unique within that space.

1.5.5 Meta Concepts

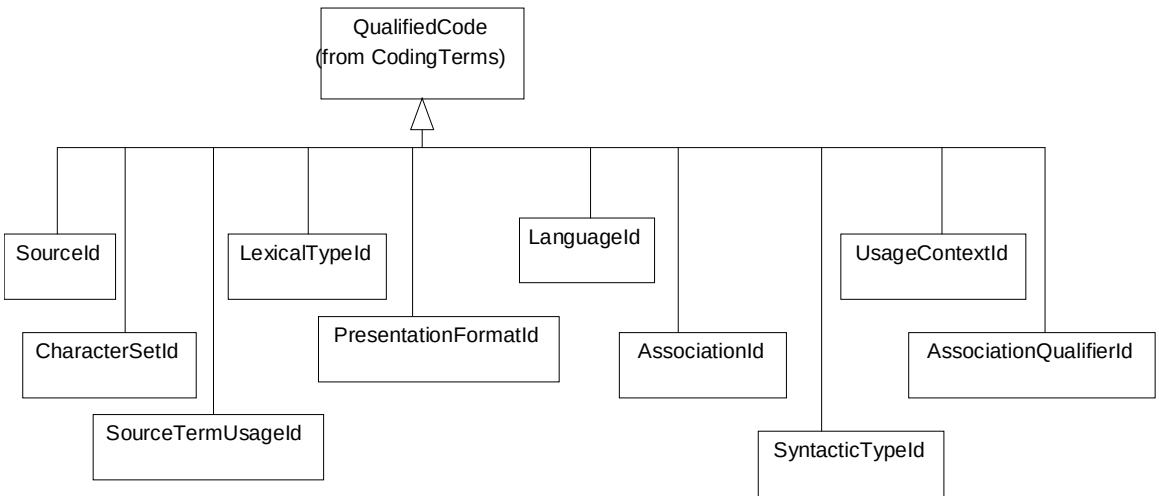


Figure 1-6 Meta Concepts

Figure 1-6 represents the coded entities that are used to access the terminology services. The semantic details and valid codes for each of these different entities are described in more detail in Section 3.2, “Meta-Terminology,” on page 3-3.

1.5.5.1 SourceId

A code that identifies a book, publication, person, or other citation. It is used to name the source from which definitions, presentations and other information within a coding scheme are drawn.

1.5.5.2 CharacterSetId

A code that identifies an international character set. It is used to specify how a given definition, presentation is to be printed or displayed.

1.5.5.3 SourceTermUsageId

A code that identifies a specific way that a string is used within a source. Example source term types include “Adjective,” “Disease Name,” “Language Qualifier.”

1.5.5.4 LexicalTypeId

The code for type which may be assigned to a presentation usage. Lexical types are such things as “abbreviation,” “Acronym,” “Eponym,” “Trade name.”

1.5.5.5 PresentationFormatId

A code that identifies the format that a given presentation is in. Example formats could include “plain text,” “html,” “.wav,” “word 7.0 document.”

1.5.5.6 LanguageId

A code that identifies a spoken or written language. Example languages include “English,” “French.”

1.5.5.7 AssociationId

A code that identifies an association type. Association types are described in more detail in Section 3.2.1, “Association,” on page 3-3.

1.5.5.8 SyntacticTypeId

A code which identifies a type of variation that a presentation takes from the preferred form within a specific linguistic group. Example syntactic types include “spelling variant,” “singular,” “plural,” and “word order.”

1.5.5.9 UsageContextId

A code which identifies a specific context in which a presentation associated with a given context code is to be used. Example usage contexts could be such things as “column heading,” “ADT application,” “long textual description.”

1.5.5.10 AssociationQualifierId

A code which qualifies or provides further information about a specific association instance.

1.5.6 Composite Types

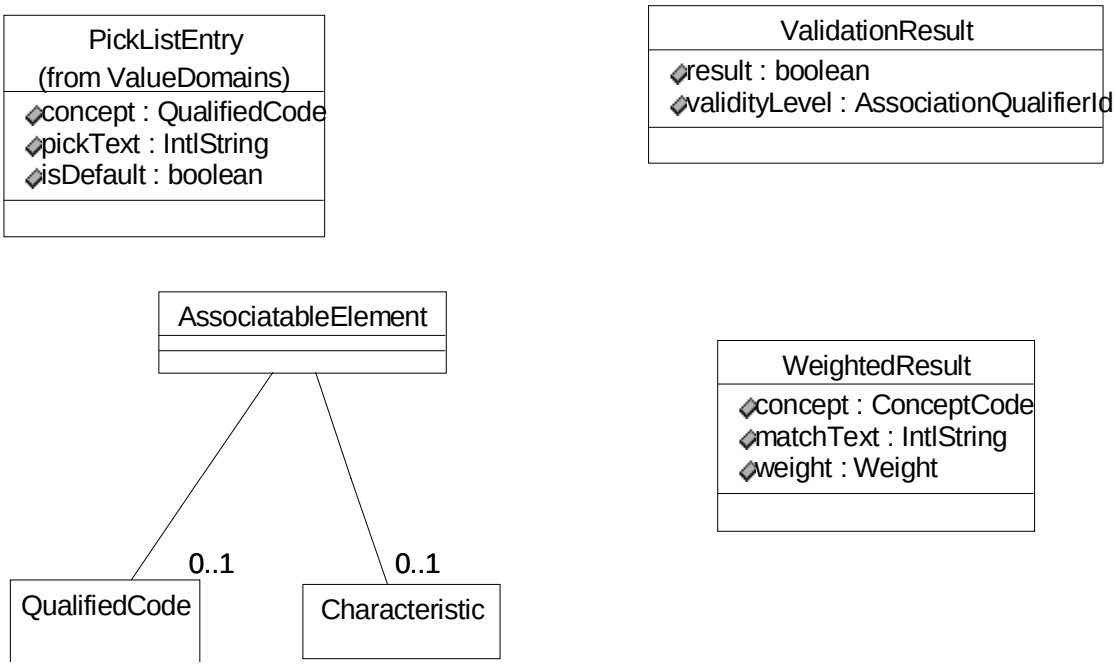


Figure 1-7 Composite Types

Figure 1-7 represents several miscellaneous composite types that are used elsewhere in this specification.

1.5.6.1 PickListEntry

This entity is used to pass a qualified concept code along with an appropriate textual representation. A **PickListEntry** also contains a flag to indicate whether it should be considered “pre-picked” by default. The **PickListEntry** is used as a return type in Section 1.6.2, “Value Domains,” on page 1-40.

1.5.6.2 ValidationResult

The result returned by the *ValidateConceptExpression* method in the *Systemization* class. Validation result contains a **Boolean** result value plus a slot for an additional qualifier on the result

1.5.6.3 AssociatableElement

A choice of either a qualified code or a characteristic. The target of an association may either be a concept code or a non-coded characteristic which represents a property or attribute of the concept code being described.

1.5.6.4 WeightedResult

An entry in a weighted result list from a match function. A weighted result includes the matching concept code, the string (if any) which resulted in the match and the relative weight assigned to the match.

1.5.7 Collections

This model uses two types of collections: *sets* and *sequences*. The model uses the name of the class to distinguish these types. Collections of the form <*entity*>*Set* represent an unordered set of entities of type <*entity*>. Set collections do not have duplicates. As an example, the type *ConceptCodeSet* is returned from the *CodingScheme.GetAllConcepts()* method. It contains an unordered list of all the unique concept codes that are managed by the coding scheme.

Collections of the form <*entity*>*Sequence* represent an ordered collection of entities of type <*entity*>. A sequence collection may contain duplicates when appropriate. An example sequence would be *CodingSchemeVersionSequence*, which is returned from the *CodingSchem.GetAllCodingSchemeVersions()* operation, and contains a list of all the supported versions of the coding scheme in reverse chronological order.

Some of the collections defined in this document may well be extremely large. It is anticipated that implementations of the abstract model will need to add additional semantics to the collections to allow for streaming style and buffered, clustered retrieval.

1.6 Terminology Service

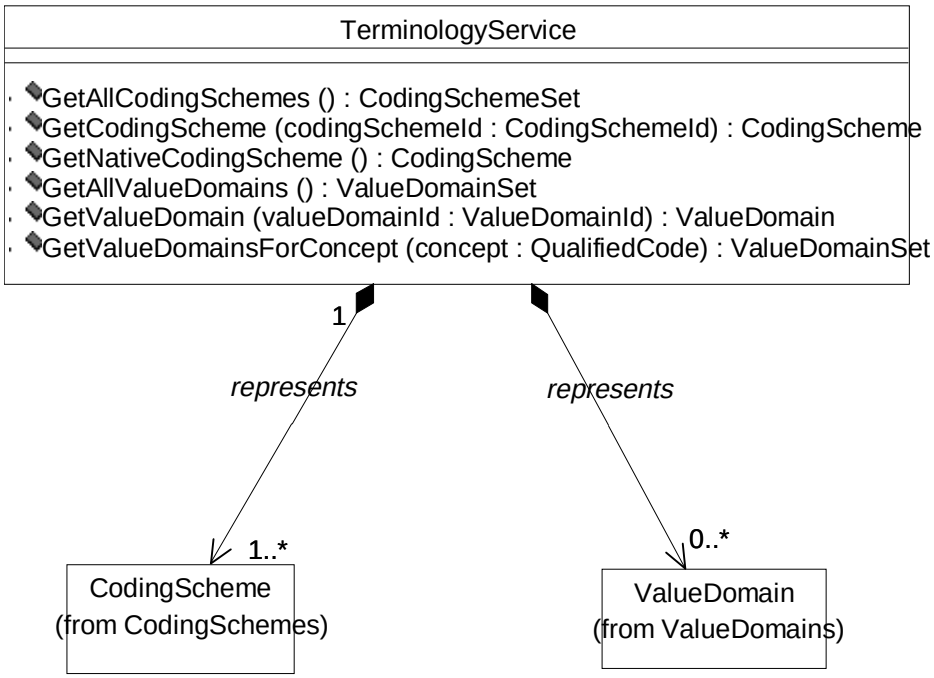


Figure 1-8 Terminology Service

A terminology service represents one or more partial or complete coding schemes. In addition it may represent a set of value domains which identify groups of concepts within and across coding schemes.

The terminology service class can return a list of all the supported coding schemes (*GetAllCodingSchemes*), return a named coding scheme (*GetCodingScheme*), or the coding scheme which has been designated the “native” coding scheme by the service provider, if any (*GetNativeCodingScheme*).

The terminology service can also list all of the value domains that are supported by the service (*GetAllValueDomains*) or access a specific value domain by name (*GetValueDomain*). In addition, it can return a list of all the value domains which contain a specified concept code (*GetValueDomainsForConcept*).

1.6.1 Coding Schemes

1.6.1.1 Coding Scheme

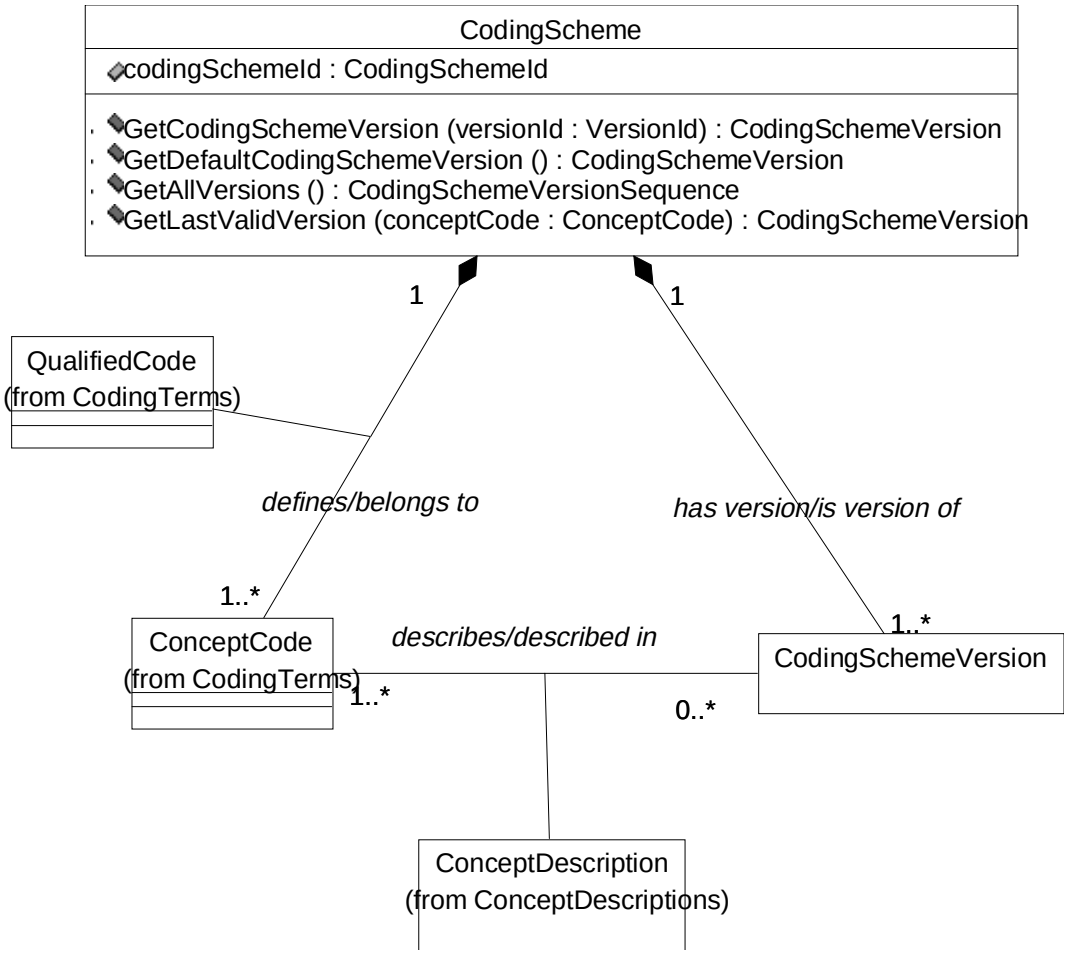


Figure 1-9 Coding Scheme

A coding scheme defines and/or describes a set of one or more concept codes. These codes are unique within the namespace of the coding scheme, and are globally unique when coupled with the name (*codingSchemeId*) of the coding scheme itself. The *QualifiedCode* class represents this globally unique combination of a concept code and a coding scheme. While not explicitly shown in this model, a qualified code is actually a subtype of a *QualifiedName* from the naming authority module.

A coding scheme may consist of more than one revision or *version*. Each version represents a consistent, reproducible state of the coding scheme. Because new concept codes may be added to a coding scheme and existing concept codes may be “retired,”

not all concept codes in a coding scheme may be described in any single coding scheme version. A vendor implementation may not maintain multiple versions of a coding scheme.

Earlier versions may be removed from the service at the vendor's discretion. It is up to the vendor to decide whether concept codes that are not described in any available coding scheme version are to be retained in the coding scheme itself.

A code can have only one "meaning" within a given coding scheme[1]. One version of a coding scheme cannot use concept code "123" to represent the color *red* while another version uses the same code to represent the color *green.* Within this constraint, definitions, comments, external representations may vary for the same concept code across the versions of the coding scheme. The definitions, comments for a concept code in a coding scheme version are represented by the *ConceptDescription* class, which will be described in detail later in this section.

The CodingScheme class provides a means (*GetAllVersions*) of listing all the versions that are available in a terminology service implementation. This list is provided in reverse chronological order with the most recent version being returned first. The *CodingScheme* class also allows direct access to a specific named version (*GetCodingSchemeVersion*). In any coding scheme, exactly one of the versions must be identified as the "default" version. This version does not have to be the latest version of the scheme. It designates the version that is preferred for general use at the given point in time. The *GetDefaultVersion* method provides direct access to the default version.

The class also provides a means of locating the chronologically latest version, if any, in which a coded concept is considered valid. The intent of this method is to allow a client to locate a coding scheme version that contains presentations, etc., even for concept codes that have been rendered obsolete.

---

1. Coding schemes like ICD-9 fall into a gray area. If we ignore the NOC (not otherwise classified) issue of the 20,000 some odd codes in ICD-9, a few (10-20) may change in meaning between revisions. Is ICD-9 1989 a different coding scheme than ICD-9 1997? It is recommended that terminology service vendors take the pragmatic view in situations like this and provide the most reasonable solution for the circumstances. In this case it would probably be representing revisions as versions.

1.6.1.2 Coding Scheme Version

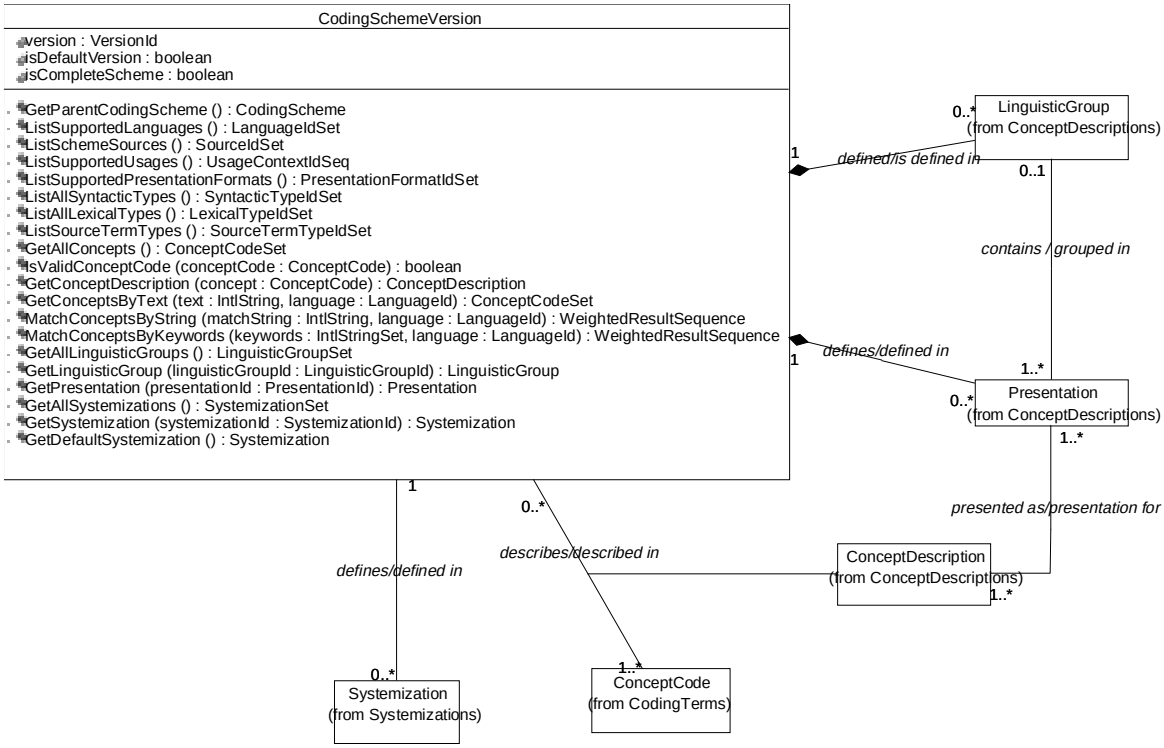


Figure 1-10 Coding Scheme Version

A coding scheme version describes or defines one or more of the concept codes contained in the coding scheme. It associates definitions, comments, instructions, and various external representations with a concept code. The *ConceptDescription* class represents a concept code as it appears in a given coding scheme version. The coding scheme version class also may provide one or more orderings, classifications, or categorizations between some or all of the concept codes within the version. This is represented by the *Systemization* class.

Some coding schemes maintain a separate list of all the unique representations that can be associated with concept codes. This list allows cross-referencing and access to additional syntactic and linguistic information. The *Presentation* class represents members of this list. Some coding schemes may also group syntactically similar presentations together into uniquely identified “linguistic groups.”

Each *CodingSchemeVersion* class has its version identifier as an attribute. This identifier uniquely names the version within the context of the parent coding scheme. With the exception of the *isDefaultVersion* flag, the contents and behavior of a named coding scheme version must be consistent and reproducible over time.

The *CodingSchemeVersion* class contains a flag *(isDefaultVersion)* that indicates whether it is considered to be the default version of the coding scheme *at that point in time.* The class also contains a second flag *(isCompleteScheme)* that is used to indicate whether the specific coding scheme version contains the entire contents of the given coding scheme or some subset thereof.

CodingSchemeVersion methods can be divided into four general groups of methods.

1. The first group consists of a set of methods with all the discovery of the characteristics of the coding scheme. *GetParentCodingScheme* allows backward traversal to the coding scheme itself. The methods *ListSupportedLanguages, ListSchemeSources, ListSupportedUsages, ListSupportedPresentationFormats, ListAllSyntacticTypes, ListAllLexicalTypes* and *ListSourceTermUsages* provide the ability to list all of the languages, sources, usage contexts, presentation formats, syntactic types, lexical types, and source term types which are partially or fully supported by this version of the coding scheme.

2. The second group provides several ways to access the concept codes that are described in the coding scheme version. They allow the user to list all of the concept codes described in the version (*GetAllConcepts*), get a detailed description of a given code in the version (*GetConceptDescription*), and match all of the concepts which have specific text (*GetConceptsByText*), text patterns (*MatchConceptsByString*) or keywords associated with them (*MatchConceptsByKeywords*). There is also a method to determine whether a given code is included in this version of the coding scheme (*IsValidConceptCode*).

3. The third group provides access to all linguistic groups defined within the coding scheme (*GetAllLinguisticGroups*), a specific named linguistic group (*GetLinguisticGroup*), or presentation (*GetPresentation*).

4. The fourth group provides the ability to access all systemizations (*GetAllSystemizations*), a specific named systemization (*GetSystemization*), or the default systemization (*GetDefaultSystemization*), if any, associated with the specific coding scheme version.

1.6.1.3 ConceptDescription – Part 1

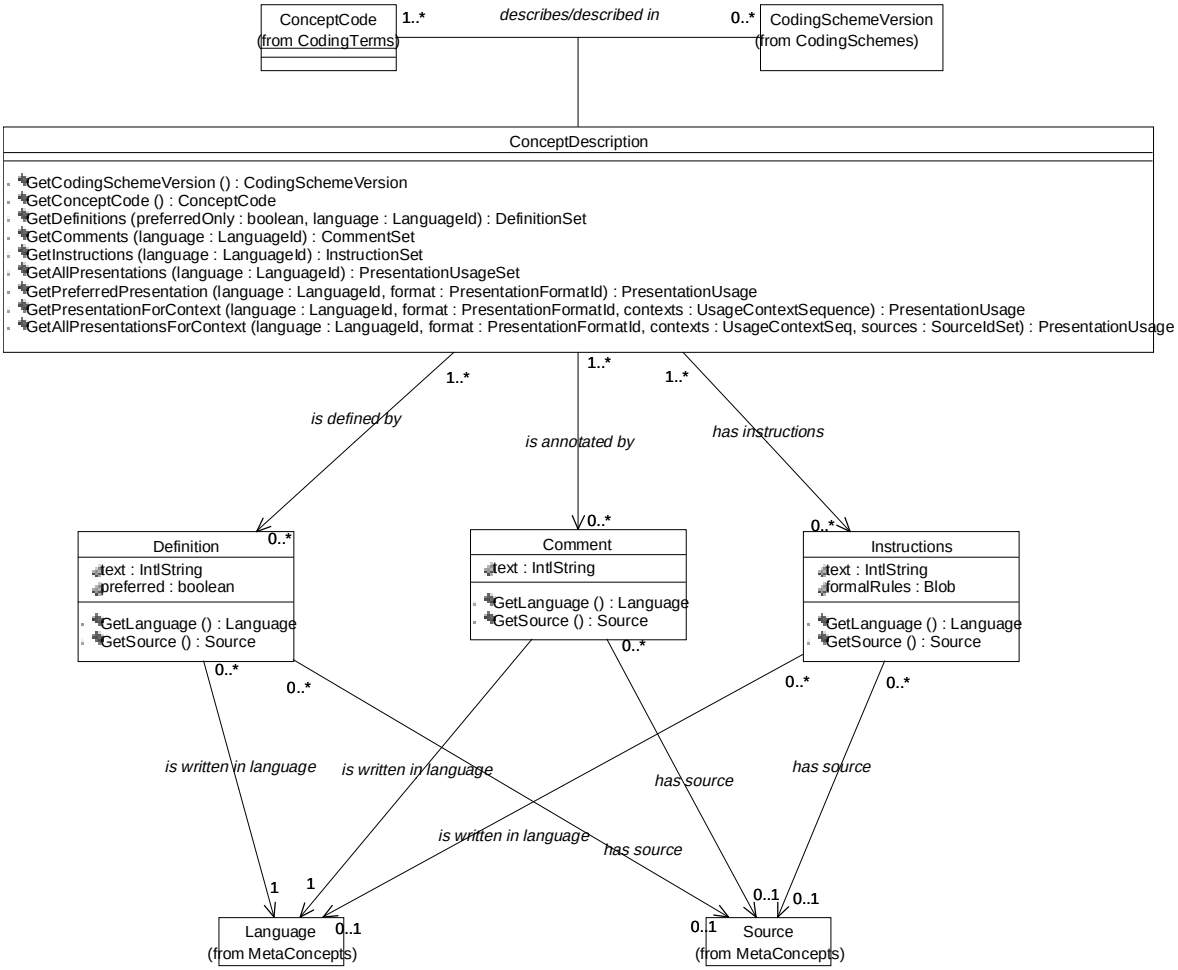


Figure 1-11 Concept Description 1

The description and definition of a concept code may include definitions, comments, presentations, etc. The exact content of each of these entities may be dependent upon the version of the coding scheme. While the “meaning” of a concept will not change within a coding scheme, it is possible for the definitions, comments and other attributes of the concept to undergo significant change over time. The entity *Concept Description* represents the description of a concept code within a specific coding scheme version.

A concept description may include several definitions, comments, and/or associated use instructions. In this model a definition is constrained to be a textual, human readable definition written in a specific language. At most one definition per language may be marked as the preferred definition for the concept code. Definitions may be attributed to a source.

Comments are non-definitional annotations and are associated with a language. Comments may be attributed to a source as well. An instruction may consist of textual instructions, machine-readable instructions, or both. If the textual portion of an instruction is present it may have a specific language. Instructions may also be attributed to sources.

The *ConceptDescription* methods allow access to the coding scheme version (*GetCodingSchemeVersion*) and concept code with which it is associated (*GetConceptCode*). The methods also provide a means to access definitions (*GetDefinitions*), comments (*GetComments*), and instructions (*GetInstructions*). All three access methods allow the provision of an optional language. If the language is specified, only entities that are associated with the supplied language are returned. Without a language, all entities are returned. The definition access method also provides the ability to specify whether the preferred definition(s) (there is at most one preferred definition *per language*) should be returned or whether all definitions associated with the concept code should be returned.

The rest of the methods associated with *ConceptDescription* are described below.

1.6.1.4 ConceptDescription – Part 2

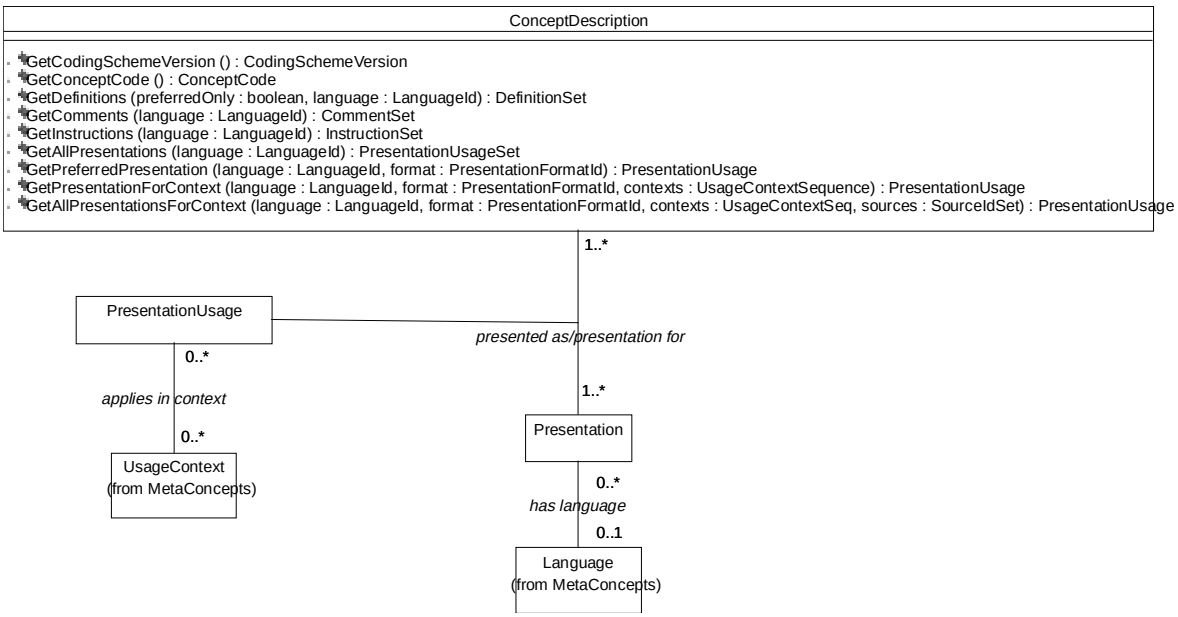


Figure 1-12 Concept Description 2

A *ConceptDescription* is the description of a concept code within a version of a coding scheme. This association includes one or more external representations that are used to present the concept code to the outside world. The *Presentation* class above represents these external representations. It is possible for a presentation to represent more than one concept code in a version of a coding scheme. The *PresentationUsage* class represents a unique association between a presentation and a concept code in a version

of a coding scheme. Coding schemes may associate a language with presentations. Presentations for a given concept code may have additional usage information associated with them, as represented by the *UsageContext* class.

In the above model, the presentation *"cold"* in the English language may be a presentation for two or more different concept codes. One code could represent a temperature and the second code represents an upper respiratory infection. While there would be one presentation, there would be two *PresentationUsage* entities, one for each presentation/concept association. The *UsageContext* for the upper respiratory infection might indicate that the text *"cold"* is to be used only when presenting the term to non-medical professionals. The *UsageContext* for the temperature might indicate that the presentation is to be used for laymen and physicians alike.

The *ConceptDescription* class allows the user to retrieve all presentations for a concept code in a coding scheme version (*GetAllPresentations*). The *language* parameter allows the selective retrieval of all concept code presentations for a specified language. The class also has a method that allows the retrieval of the preferred presentation for concept code in a given language and presentation format (*GetPreferredPresentation*).

The *ConceptDescription* class can also be used to retrieve the "best" presentation associated with a concept code given a set of usage contexts (*GetPresentationsForContext*). There is also a method that allows the user to retrieve all possible presentations for a concept code to be retrieved given a usage context set (*GetAllPresentationsForContext*).

Details about presentations are described in the following section.

Presentations

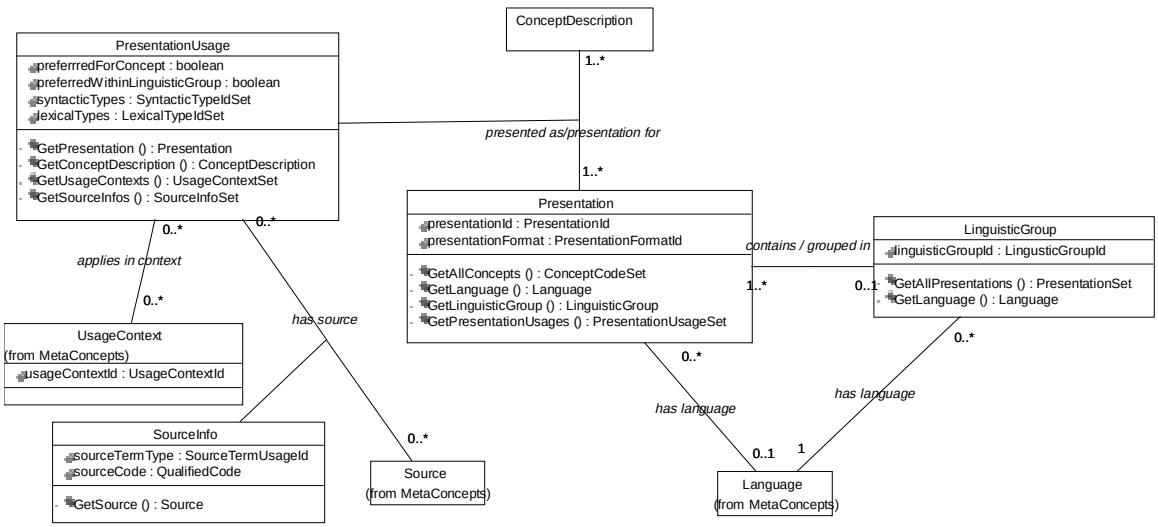


Figure 1-13 Presentations

A presentation is an external sign or symbol used to represent a concept code. Some coding schemes provide a unique identifier for each unique presentation in a given language. This identifier is represented by the *presentationId*.

Each presentation has a format type (*presentationFormat*) which identifies additional external processing which may be necessary to properly display the presentation. Possible format types include *plain text, html, rtf, .wav sound byte,* etc.

The presentation class provides the ability to determine all of the concepts which use the particular presentation (*GetAllConcepts*), to determine the language of the presentation, if any (*GetLanguage*) and to determine which linguistic group that the concept belongs to, if any (*GetLinguisticGroup*). The class also provides a means of accessing all presentation usage entities associated with it (*GetPresentationUsages).*

Some coding schemes group syntactically similar presentations into linguistic groups. This grouping is independent of how the presentations are used, and a presentation belongs to at most one linguistic group. If present in the coding scheme, a linguistic group will have an identifier (*linguisticGroupId*) which is unique within the coding scheme. The linguistic group class allows the enumeration of all presentations contained within the group (*GetAllPresentations*).

The PresentationUsage class represents the association between a concept code and a presentation in a given version of a coding scheme. This class allows one presentation to be identified as the preferred presentation for the concept (*preferredForConcept*). If the coding scheme includes linguistic groups, one presentation per linguistic group may also be identified as the preferred presentation for that concept in that group (*preferredWithinLinguisticGroup*). If the coding scheme includes linguistic groups, the *syntacticTypes* may specify how non-preferred presentations vary from the preferred presentation for the concept in the group. Possible types may include *plural, spelling, word order variation,* etc. The presentation usage may also be associated with one or more lexical types. Typical lexical types include *acronym, eponym, trade name,* etc.

A presentation associated with a concept code in a coding scheme may be appropriate only in certain situations or contexts. The *UsageContext* class allows specific contexts, in which a presentation applies. Typical usage contexts might include *short column heading, single-line text, presentation for physician, presentation for layman,* etc. The *GetUsageContexts* method provides access to all of the applicable usage contexts.

A given presentation for a concept code may be attributed to one or more external sources. Typical sources could include dictionaries, terminology manuals, thesauri, as well as different coding schemes. The method *GetSourceInfos* provides access to the various source references. Each source reference may be attributed with the additional information about how the presentation for the concept code is used in the specific source (*sourceTermUsage*). A typical *sourceTermUsage* might be *adjective, finding name, machine permutation, etc.* Each attribution may also carry a code (*sourceCode*) which is associated with that presentation in that source.

Presentation Types

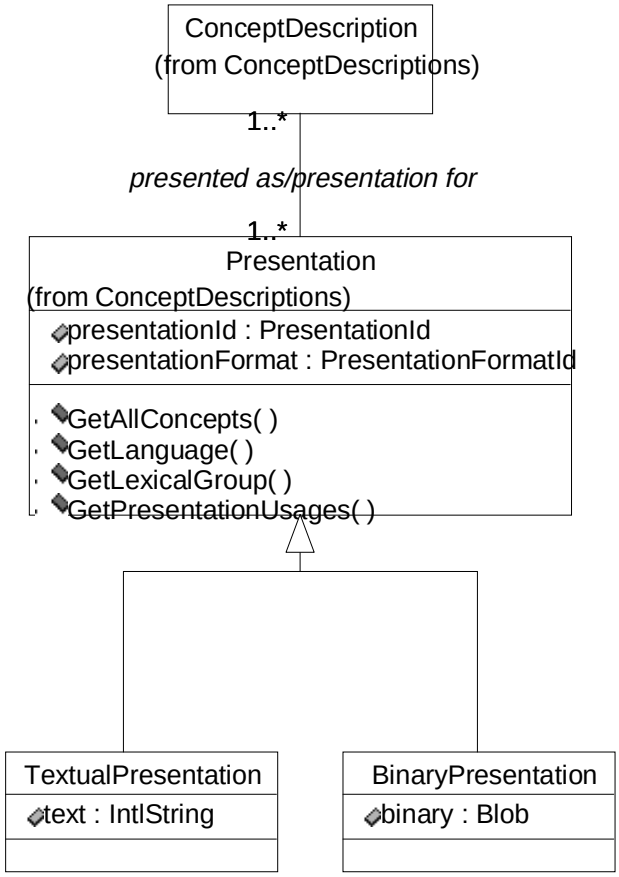


Figure 1-14 Presentation Types

This model shown in Figure 1-14 is restricted to presentations that represent linguistic concepts. A presentation is expected to represent a word, phrase, or set of phrases in a human language. Presentations may include such things as HyperText Markup Language (HTML) documents, formatted text such as Rich Text Format (RTF) documents, .wav files which represent spoken words. This model is not designed to include non-textual pictures, icons, sounds beside spoken words.

Each presentation has a format type (*presentationFormat*) which identifies additional external processing which may be necessary to properly display the presentation. Possible format types include *plain text, html, rtf, .wav sound byte,* etc. A presentation may be either straight text or may contain binary data. The *TextualPresentation* subtype represents textual data and the *BinaryPresentation* format represents non-textual and/or binary data. No association should be inferred between *presentationFormat* and the type of presentation. It is quite possible that a terminology system may represent *plain text as a BinaryPresentation* or some other type as a *TextualPresentation.*

1.6.1.5 Systemizations

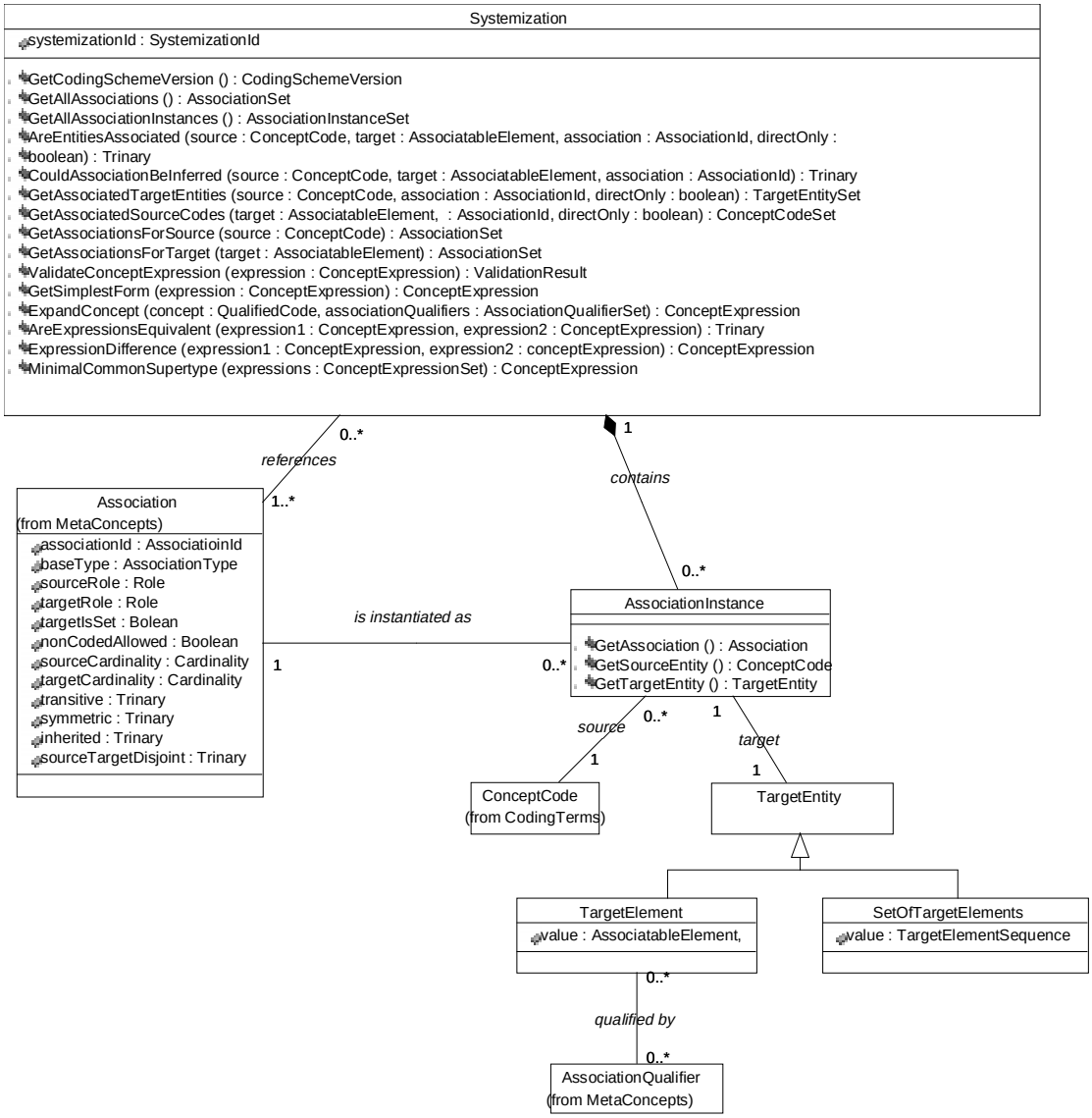


Figure 1-15 Systemization

A systemization represents an ordering, classification, and/or categorization of a set of concept codes. The purpose of a systemization is to further define and describe the concept codes within a coding scheme, and possibly to define the relationship between these concept codes and concept codes in other coding schemes.

The systemization class references one or more association types, which are then instantiated as association instances between concept codes and other concept codes or characteristics.

The source of an association instance must be a concept code from the local coding scheme. The target of an association depends upon the association type. It may either be a single target element or a set of target elements. A single target element may either be a qualified code or a characteristic.[2]

The *systemizationId* serves to uniquely identify the systemization within the coding scheme version. The *GetCodingSchemeVersion* method serves to access the coding scheme version in which the systemization is implemented.

GetAllAssociations returns a list of all the association types that participate in the systemization. See Section 3.2.1, “Association,” on page 3-3 for details on the *Association* attributes.

GetAllAssociationInstances returns a (potentially large) list of all the associations that are contained within the systemization.

The systemization class has several methods that are used to query specific associations within the systemization. The first method, *AreEntitiesAssociated*, asks whether an instance of the named association exists in which the supplied concept code has the source role and the supplied associatable element has the target role. The *directOnly* flag indicates whether only direct associations are to be considered (*directOnly = TRUE*) or whether a transitive paths between the source and target are also to be taken into account (*directOnly = FALSE*).

As an example, if the following associations were present in a systemization:

<Computer> hasComponents {<MotherBoard>, <Monitor>, <Keyboard>}

<MotherBoard> hasComponents { <ALU>, <Clock>, <Memory>}

the query *AreEntitiesRelated(<Computer>, <ALU>, hasComponents, TRUE)* would yield a result of FALSE, as there is no direct association involving <Computer> and <ALU>. The query *AreEntitiesRelated(<Computer>, <ALU>, hasComponents, FALSE)* would yield TRUE, as there *is* an indirect path along the *hasComponents* association between <Computer> and <ALU>. Queries on non-transitive or intransitive associations behave as if the *directOnly* flag is always TRUE. The *AreEntitiesRelated* query may also return “unknown”, indicating that the systemization has insufficient information to be able to determine whether a given association exists or not.

The *AreEntitiesRelated* query only returns entities which are directly or indirectly associated with the input entity using the *supplied association*. Subtype associations are not taken into account. For example:

<memory> hasSubtypes (<disk>, <ram>, <rom>)
<Computer> hasComponents {<MotherBoard>, <Monitor>, <Keyboard>}
<MotherBoard> hasComponents { <ALU>, <Clock>, <Memory>}

---

2. Note that the distinction between a qualified code and characteristic is often imprecise. As an example, one coding scheme or terminology vendor may choose to represent a color attribute using a simple string, while another vendor may encode a list of possible colors. A client should be coded in such a way that it can cope with either situation.

The query *AreEntitiesRelated(<Motherboard>, <ram>, hasComponents, FALSE)* would yield a result of FALSE. The query *CouldAssociationBeInferred* allows terminology vendors to expose more sophisticated inferencing capabilities, crossing subtype and other associations in the process of reaching the result. The query *CouldAssociationBeInferred(<Motherboard>, <ram>, hasComponents)* would probably return a TRUE value. It is anticipated that terminology vendors who implement the *CouldAssociationBeInferred* will probably add additional, proprietary methods to provide inference explanations and other more sophisticated properties.

GetAssociatedTargetEntities returns the set of all target entities that participate in the named association with the source code. If *directOnly* is TRUE, only the target entities directly associated with the source codes are supplied. If FALSE, all of the target entities in the transitive closure of the association are returned.

As an example, given the following association instances:

<Anti-Infective Agent> hasSubtypes {<Amebacide>, <Anthelmintic>}
<Amebacide> hasSubtypes {<hydroxyquinoline derivatives>,
<arsenical anti-infectives>}
<Anthelmintic> hasSubtypes <{quinoline derivatives}>

The query *GetAssociatedTargetEntities(<Anti-Infective Agent>, hasSubtypes, TRUE)* would return a set consisting of one element (another set):

{{<Amebacide>, <Anthelmintic>}}

while the query *GetAssociatedTargetEntities(<Anti-Infective Agent>, hasSubtypes, FALSE)* would return the set consisting of four elements:

{{<Amebacide>, <Anthelmintic>}, {<hydroxyquinoline derivatives>, <arsenical anti-infectives>}, <{quinoline derivatives}}.

Similarly, *GetAssociatedSourceCodes* returns all of the source qualified codes that participate in the named association with the supplied target element. The query *GetAssociatedSourceCodes(<arsenical anti-infectives>, hasSubtypes, TRUE)* would return the set:

{<Amebacide>}

while the query *GetAssociatedSourceCodes(<arsenical anti-infectives>, hasSubtypes, FALSE)* would yield

{{<Amebacide> <Anti-Infective Agent>}

The method *GetAssociationsForSource* returns the set of all associations in which the supplied qualified code participates in the source role. *GetAssociationsForTarget* returns the set of all associations in which the supplied target element participates in the target role.

Concept Expressions

Concept expressions consist of the logical conjunction of a set of base concept codes, each of which is optionally qualified by one or more attribute value pairs which serve to further define or constrain the class of entities which the concept code may represent.

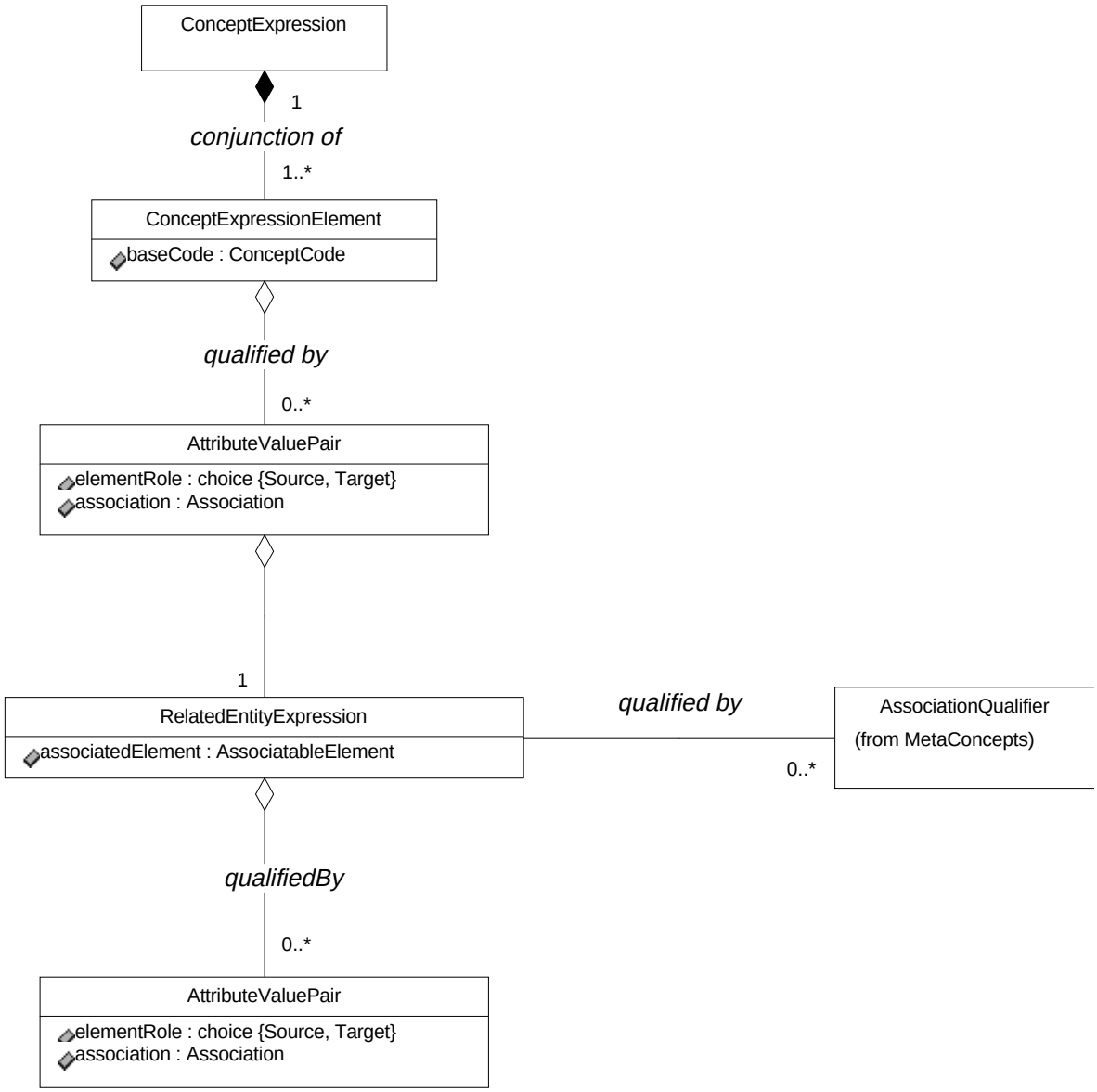


Figure 1-16 Concept Expression

1.6.1.6 *ConceptExpression*

A concept expression is the logical conjunction of one or more *ConceptExpressionElements*, each of which represents a base concept code and a set of optional, nested qualifiers.

1.6.1.7 ConceptExpressionElement

A concept expression represents a concept code whose scope or "meaning" has been further constrained or refined by the addition of qualifiers. It has, as its base, a single concept code. One or more *AttributeValuePairs* may further qualify this concept code. An *AttributeValuePair* consists of an association, an *AssociatableElement*, and an optional set of association qualifiers. Additional *AttributeValuePairs* may further qualify the *AssociatableElement* within an *AttributeValuePair.* An *AssociatableElement* represents a qualified name in the case where the *baseRole* is *Target* (when the base concept occupies the target role and the *AssociatableElement* the source). It may represent either a qualified name or a characteristic in the case where the *baseRole* is *Source*.

1.6.1.8 RelatedEntityExpression

An entity expression is identical to a concept expression with the exception that the "qualified" or base entity can be an *AssociatableElement*, which may either be a qualified code or a characteristic.

Expression and simplification

The next seven methods in a systemization expose functionality associated with concept expressions. The notation for the following examples is borrowed heavily from the GALEN CORE notation. [13] Given the following example, which represents the upper lobe of the left lung:

Lobe which <*is-part-of* (*Lung* which *has-laterality* Left)

has-location "upper">

this could be expressed as a concept expression as follows:

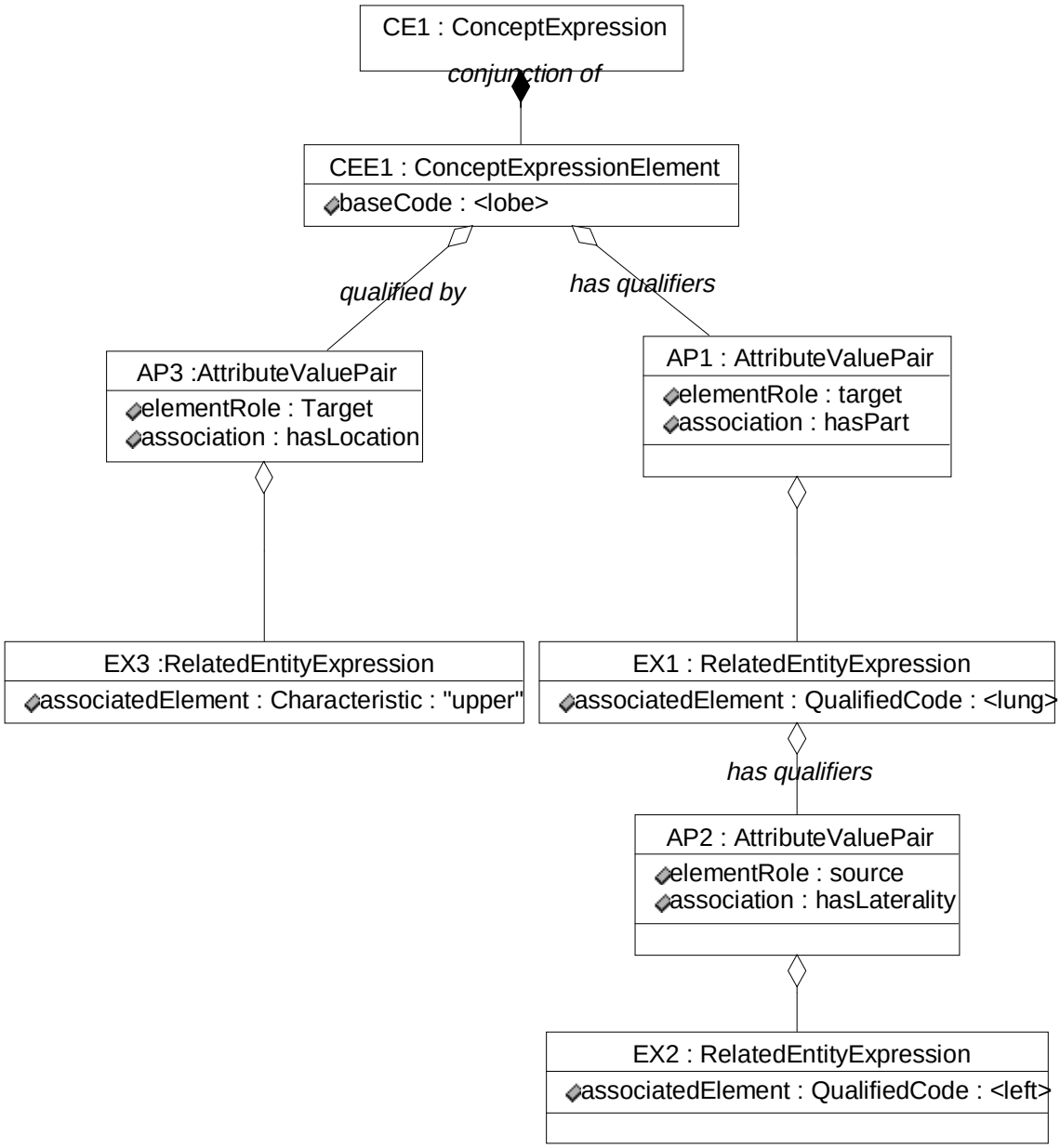


Figure 1-17 Concept Expression Example

In Figure 1-17, the text “upper” has been typed as a characteristic to provide a simple example of the representation of characteristics.Typically it would be a coded concept in a real terminology service.

The systemization class provides the methods listed below for manipulating concept expressions.

ValidateConceptExpression – Returns FALSE if the supplied concept expression is not considered valid. If the return is TRUE, an optional association qualifier may also be returned to further qualify the conditions in which the TRUE return applies. As a hypothetical example, a systemization might return a qualifier of "sensible." If the concept expression described the *middle* lobe of the left lung, the systemization might return a qualifier of "grammatical", indicating that, while there *isn't* a middle lobe of the left lung, the expression still made grammatical sense.

GetSimplestForm – Returns the concept expression which represents the simplest form in which the supplied concept expression may be expressed. Using the example above, a terminology system might have a concept code that represented the left lung. The result of a *GetSimplestForm* call with the example above might yield:

Lobe which <*is-part-of* LeftLung

has-location Upper>

ExpandConcept – Takes the supplied concept and returns the "canonical" concept expression that serves to define the concept. Were *ExpandConcept* supplied with the concept code <LeftLung> in the above scenario, it might return:

Lung which has-laterality Left

AreExpressionsEquivalent – Given two concept expressions, this method determines whether these two expressions could be considered equivalent.

ExpressionDifference – Determines the "difference" between the two concept expressions and returns it in the form of a third concept expression.

MinimalCommonSupertype – Returns the concept expression which is the "closest" valid supertype of the supplied list of concepts expressions. The application is notified if there is no valid minimal common supertype short of the *universal type.*

MaximalCommonSubtype – Returns the concept expression which is the "closest" valid subtype of the supplied list of concepts expressions. The application is notified if there is no valid maximal common subtype short of the *absurd type.*

The association between concept expressions and the systemization

Any concept expression that is deemed "valid" by the systemization is presumed to have a corresponding association within the systemization itself. If the systemization does not support the *CouldAssociationBeInferred*, each valid base code / attribute / value association in a concept expression should receive a TRUE return when supplied as parameters to the *AreEntitiesAssociated* operation. If the systemization supports *CouldAssociationBeInferred*, it is required that this operation return TRUE given the same set of input parameters.

1.6.2 Value Domains

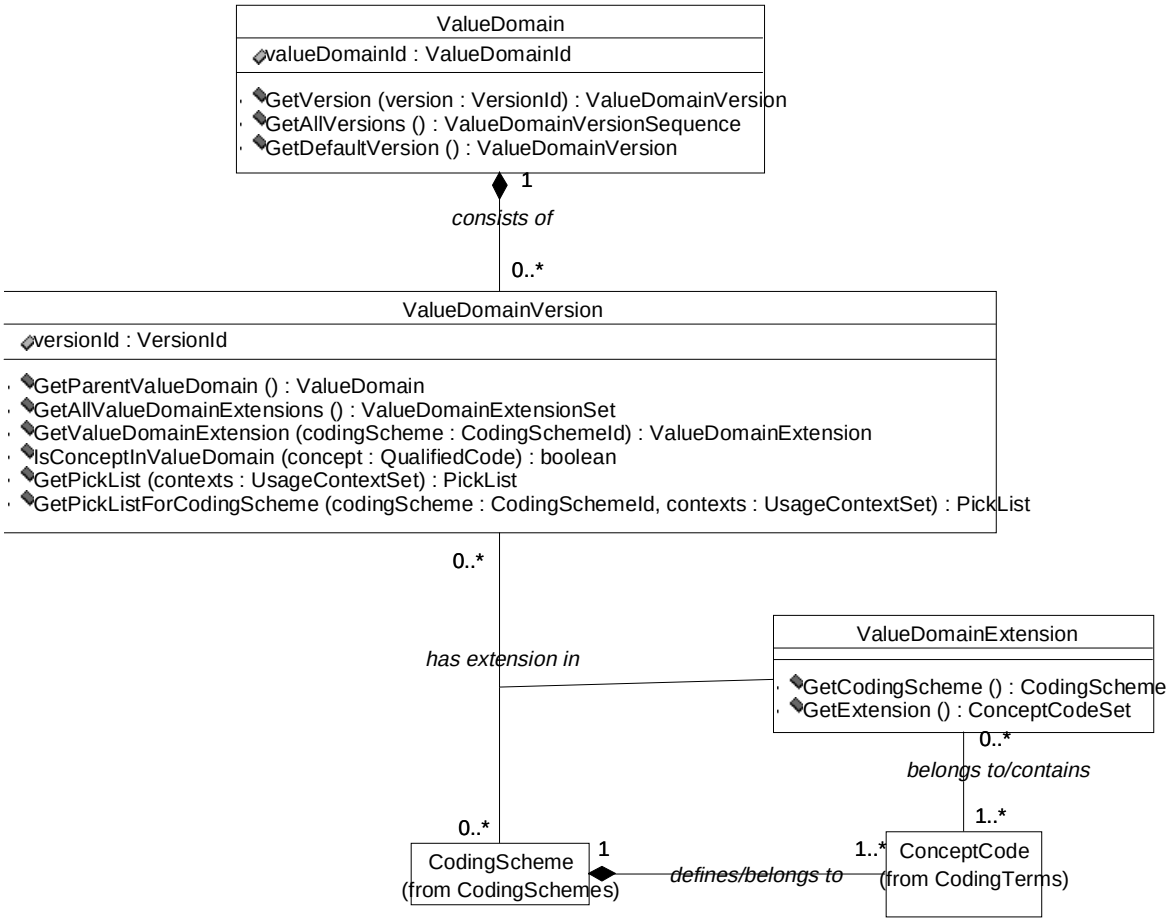


Figure 1-18 Value Domain

A value domain is typically associated with a field on a data-entry screen, a column in a database, a field in a message, or some other entity that may contain a concept code. Each value domain instance is uniquely identified by the *valueDomainId*, a qualified code. Each value domain may have more than one version which is used to record the changing contents of a value domain over time. The ValueDomain class can return all versions (*GetAllVersions*), a specific version by identifier (*GetVersion*), or the default version *(GetDefaultVersion*) for the given value domain.

Each value domain version has an “extension” in one or more different coding schemes. An extension is defined in this context as a list of concept codes from the given coding scheme. The *ValueDomainExtension* class represents this extension. Using this class, an application may retrieve all extensions for a value domain (*GetAllValueDomainExtensions*), or a specific extension for a given coding scheme

(*GetValueDomainExtension*). The application may also query as to whether a specified concept code is included in one of the value domain extensions *(IsConceptInValueDomain*).

A pick list is an ordered subset of all the concept codes which are included in a given value domain version. The order and contents of this list may be tailored for a specific user or group of users, a given application, or some other usage context. Each concept code in a pick list contains a textual presentation which will be used to represent the concept to the user as well as a flag which indicates whether the particular code is to be considered a default selection for the list. Pick lists may be selected either across all coding schemes (*GetPickList*) or from one specific scheme *(GetPickListForCodingScheme*).

1.7 IDL Interface

This specification consists of three modules:

- **NamingAuthority** - A general-purpose module that provides a means of providing unique names to entities such as concept codes, components. This module is shared with the Patient Identification Services (PIDS) specification.
- **Terminology Services** -This module defines the services which are the focus of this document.
- **Terminology Service Values** - This module defines the codes and coding schemes which are used by the terminology services.

1.8 Notation

1.8.1 Sequences and Sets

Entities which end in the suffix "**Seq**" are typically not described in the document below. **<Entity>Seq** is used to represent both an unordered set of **Entity** and an ordered sequence of **Entity**. Entity is presumed to be an unordered set unless it is otherwise stated in the accompanying text.

1.8.2 Iterators

Entities which end in the suffix "Iter" are also not further described. **<Entity>Iter** represents an iterator for objects of type **Entity**. All iterator objects contain the following interface methods:

1.8.2.1 max_left

This returns an approximation of the number of **Entity** yet to be retrieved. **Max_left** will never return a value that is less than the total remaining elements. Client applications should use this attribute sparingly, as it may be very costly in some implementations across large databases.

1.8.2.2 *next_n*

This operation returns a sequence of **Entity** elements. The number of elements in the returned sequence will never be more than the input value "**n**."

1.8.2.3 *destroy*

This operation should be invoked when the client is finished retrieving entities from the iterator. It is not necessary to iterate to the end of the list before destroying the iterator.

Modules and Interfaces 2

Contents

This chapter contains the following topics.

2.1 NamingAuthority Module

```
//File: NamingAuthority.idl

#ifndef _NAMING_AUTHORITY_IDL_
#define _NAMING_AUTHORITY_IDL_

#include <orb.idl>

#pragma prefix "omg.org "

module NamingAuthority
{
    enum RegistrationAuthority {
        OTHER,
        ISO,
        DNS,
        IDL,
        DCE };
```

```
    typedef string NamingEntity;

    struct AuthorityId {
        RegistrationAuthority  authority;
        NamingEntity  naming_entity;
    };
    typedef string AuthorityIdStr;

    typedef string LocalName;
    struct QualifiedName {
        AuthorityId authority_id;
        LocalName local_name;
    };
    typedef string QualifiedNameStr;

    exception InvalidInput {};

    interface translation_library
    {
        AuthorityIdStr authority_to_str(
                in AuthorityId authority )
            raises(
                InvalidInput );
        AuthorityId str_to_authority(
                in AuthorityIdStr authority_str )
            raises(
                InvalidInput );

        QualifiedNameStr qualified_name_to_str(
                in QualifiedName qualified_name )
            raises(
                InvalidInput );
        QualifiedName str_to_qualified_name(
                in QualifiedNameStr qualified_name_str )
            raises(
                InvalidInput );
    };
};

#endif // _NAMING_AUTHORITY_IDL_
```

The **NamingAuthority** module provides a means of giving globally unique names to name spaces and hence the names within those name spaces. The fundamental need is the ability to compare two names for equality. If they are equal, they are known to represent the same entity, concept, or thing. This is needed when independent entities are generating names that may get compared for equality. However, the reverse is not guaranteed to be true. That is an entity that may have several names.

The authority for the name space may derive from several different types of roots, the choice of which depends upon the user requirements as each root has different qualities of management and uniqueness. The various root types are defined below.

#pragma prefix "org.omg"

In order to prevent name pollution and name clashing of IDL types this module (and all modules defined in this specification) uses the pragma prefix that is the reverse of the OMG's DNS name.

2.1.1 RegistrationAuthority

Identifies the root of the name space authority. An entity (e.g., person or organization) may be registered with many different roots (RegistrationAuthorities) and be able to assign names and other name spaces within each root. These may be used for the same or for different needs. For this reason there is no guarantee of any equality in the different name spaces managed by an entity. There are currently no means available to determine whether a given authority in an ISO hierarchy is the same authority as one specified in a DNS hierarchy.

Other

This form of a naming authority should be used sparingly, and only in experimental or localized situations or special purposes. It is the responsibility of the implementing institution to guarantee uniqueness within the names themselves, and there is no uniqueness guarantee outside of the source institution. Services that define default naming authorities (and possibly also names) may also use the Other root to forego long AuthorityIds. In this case the specification of the service must name AuthorityIds that may be expected with the Other root and still maintain name space integrity for that service.

ISO

International Standards Organization [9] - The ISO specifies a registration hierarchy, identified by a series of named/numbered nodes. Many of the coding schemes used in the medical environment are or can be registered within the ISO naming tree. The ISO root form is one of the recommended forms when the naming authority is internationally recognized, such as international coding schemes, or when the authority is to be used across two or more different enterprises. ISO provides for the recording of a responsible person and address for each node in the authority hierarchy.

DNS

Domain Name Services [10] - Internet domains are recorded with a central, global registration authority. Subhierarchies within the domains are then maintained locally by the registered organization or person. The DNS form is recommended as an alternative to the ISO naming tree when the specific naming authority needs identity and uniqueness, but is not in an ISO registration. By using this common characteristic of many organizations it gives the ability to create globally unique name spaces and

names without the need to register as an ISO name authority. It is up to the organization itself to maintain the integrity of the name space(s) (e.g., not reusing names or name spaces).

IDL

The OMG Interface Repository [11] - The CORBA Architecture specifies a means of uniquely identifying entities within the interface repository, via the use of a *RepositoryId*. CORBA repository id's may be in either the OMG IDL format, the DCE UUID format or the LOCAL format. Within this specification, the "IDL" root refers only to the IDL format. The DCE format may be represented within the DCE root and the Local format within the Other root. The IDL authority may prove very useful when registering CORBA/IDL specific objects such as value sets, interface specifications. It should be noted that OMG does not currently manage the repository name space in any rigorous fashion, and it is quite possible that two different developers may arrive at exactly the same repository ID for entirely different entities. For this reason some people give the repository ID a prefix that consists of their reverse DNS that is '/' separated instead of '.' separated. This root type may be very useful when the names within the name space are defined in IDL. For example, it could be the RepositoryId for an enumerated type or a module that has constant integers or strings defined for each name within the name space.

DCE

The Distributed Computing Environment [12] - While they don't actually register coding schemes or other entities, they do provide a means of generating a globally unique 128-bit ID, called a Universally Unique ID (UUID). This UUID may be used to guarantee the uniqueness of a name space in situations where it is not necessary for the identity of the authority to be known outside of the specific implementation.

2.1.2 NamingEntity

Identifies a specific name in the syntax and format specified by the corresponding registration authority. The various naming authorities tend to provide a fair amount of leeway as far as the actual format of the registered names. As there may be situations where the full semantics of a specific authority's name comparison will not be available to an application, we have chosen to select a specific subset of the syntax of each representation. The intention is to be able to determine whether two registered entities are identical or not solely through the use of string comparison. The specific name formats are described below:

OTHER

An arbitrary string, syntax undefined except locally by a specific service specification and/or by particular implementations and installations. The "/" character is illegal to use as it is reserved as a separator of components in the stringified version of QualifiedName.

ISO

The name should be represented using the *NameForm* of the *ObjectIdentifierValue* as specified in ISO/IEC Recommendation 8824-1. Each name component should be separated by a single space.

Example: “joint-iso-ccitt specification characterString”

DNS

The domain name and path in the form mandated in RFC 1034. The path name is represented as a dot separated tree which traverses up the hierarchy. Since DNS names are not case-sensitive only lower-case letter should be used such that simple string comparisons can determine equality. However it is OK to use case-insensitive comparisons as well.

Example: “pidsserv.slc.mmm.com”

IDL

The OMG RepositoryId format specified in the CORBA Architecture V2.0 manual, in the form: “<node>/<node>/Ö/<node>.” The “IDL:” prefix and the version number suffix should NOT be used for the **NamingEntity**. The “IDL:” prefix is prepended to create the **AuthorityIdStr**.

Example: “**CosNaming**/**NamingContext**/**NotFoundReason**” is the **NamingEntity** for:

```
module CosNaming {
...
interface NamingContext {
...
enum NotFoundReason { ... };
...
};
};
```

DCE

The UUID in the external form <nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn>, where <n> represents one of the digits 0-9 and the characters A-F. The alpha characters should all be upper case.

Example: “6132A880-9A34-1182-A20A-AF30CF7A0000”

2.1.3 AuthorityId, AuthorityIdStr

The combination of a Registration Authority and Naming Entity, which identifies a specific naming authority. In situations where a given naming entity may have more than one naming authority, it should be agreed upon in advance which of the specific names for the entity is to be used. This specification makes no guarantees about the ability to recognize, for example, that an authority in the ISO structure is identical to an authority within the IDL structure.

The string version (**AuthorityIdStr**) is useful for situations where unique names are required in a string format. The string is created as <stringified **RegistrationAuthority**>:<**NamingEntity**>.

2.1.4 *LocalName, QualifiedName, QualifiedNameStr*

A local name is a name within (relative to) a namespace. It is simply a string representation.

A **QualifiedName** is a globally unique name for an entity by the fact that it carries the naming **AuthorityId** of the name space and the **LocalName** within that name space.

The **QualifiedNameStr** is a stringified version of the **QualifiedName**. The format of the string is <stringified **RegistrationAuthority**>:<**NamingEntity**>/<**LocalName**>. Notice that even though the character "/" cannot be used within the name of a **NamingEntity**, it can be used within the LocalName.

The following table summarizes the format for **QualifiedNameStr**. Columns 1-3 give the format for an **AuthorityIdStr**.

Table 2-1 Registration Authority Formats

Registration Authority	**(1) Stringified Registration Authority**	**(2) RA-NE Delimiter**	**(3) NamingEntity Format**	**(4) NE-LN Delimiter**	**(5) LocalName Format**
OTHER	""	":" optional	<no '/'>	"/" optional	<no '/'>
ISO	"ISO"	":"	<use ISO rules>	"/"	<any characters>
DNS	"DNS"	":"	<use DNS rules>	"/"	<any characters>
IDL	"IDL"	":"	<use IDL rules>	"/"	<no '/'>
DCE	"DCE"	":"	<use DCE rules>	"/"	<any characters>

The definitions for type OTHER are defined to allow using a **QualifiedNameStr** format in contexts where an IDL 'string' is currently used. A normal IDL string is a **QualifiedNameStr** with no **RegistrationAuthority** and no **NamingEntity**. The limitation is that any normal IDL strings that start with one of the **RegsitrationAuthority** strings cannot be mapped into the **QualifiedNameStr** since they would be interpreted by the rules in this module.

The string for the 'OTHER' type of **RegistrationAuthority** being a blank string ("") makes it easier for locally defined names to be usable with no requirements on the format except they cannot start with one of the strings reserved for the other **RegsitrationAuthority** types. The ':' delimiter is optional for type OTHER. If the **NamingEntity** is "" for type OTHER then the '/' delimiter is also optional.

2.1.5 Exceptions

The InvalidInput exception is raised when the input parameter for the **TranslationLibrary** interface operations is determined to be of an invalid format.

2.1.6 TranslationLibrary Interface

This interface is meant to be a local library for translating between the structured version and stringified version of **AuthorityIds** and **QualifiedNames**.

authority_to_str, str_to_authority, qualified_name_to_str, str_to_qualified_name

Each of these operations take either a structured version or stringified version of a type and return the opposite. The data content returned is the same as that passed in. Only the representation of the data is changed.

2.2 Terminology Service Module

```
//File: TerminologyServices.idl
//
#ifndef _TERMINOLOGY_SERVICES_IDL_
#define _TERMINOLOGY_SERVICES_IDL_
#pragma prefix "omg.org"
#include <orb.idl>
#include <NamingAuthority.idl>

// ***************************************************************
//      module:  TerminologyService
// ***************************************************************

module TerminologyServices {
    // . . .

};

#endif /* _TERMINOLOGY_SERVICES_IDL_ */
```

The **TerminologyServices** module consists of type and interface definitions. Each interface represents an object class or some portion of an object class from the abstract model. Separate interface classes have been created in some cases to provide for optional implementation conformance points. The body of the **TerminologyServices** module is organized as follows:

2.2.1 Type Definitions

- Basic Terms
- Meta Types (The **TerminologyServiceValues** module contains the type constants)

- Coding Terms
- Coding Scheme and Coded Concept Terms
- Advanced Query Terms
- Systemization Terms
- Value Domain Terms

2.2.2 Exceptions

Interfaces

- TranslationLibrary
- TerminologyService
- LexExplorer
- CodingSchemeLocator
- ValueDomainLocator
- CodingSchemeVersionAttributes
- CodingSchemeVersion
- PresentationAccess
- LinguisticGroupAccess
- AdvancedQuery
- SystemizationAccess
- Systemization
- ValueDomainVersion

2.2.3 Basic Coding Terms

```
//*****************************************************************
//    Basic Terms
//*****************************************************************

typedef NamingAuthority::LocalName  ConceptCode;
typedef sequence<ConceptCode  ConceptCodeSeq;

typedef NamingAuthority::AuthorityId CodingSchemeId;
typedef sequence<CodingSchemeId  CodingSchemeIdSeq;

struct QualifiedCode {
    CodingSchemeId  coding_scheme_id;
    ConceptCode  a_code;
};
typedef sequence <QualifiedCode> QualifiedCodeSeq;
```

```
typedef  string VersionId;
typedef  sequence<VersionId>  VersionIdSeq;
const VersionId DEFAULT = "";

struct TerminologyServiceName {
    NamingAuthority::QualifiedName  the_name;
    VersionId   the_version;
};
```

ConceptCode

A string of characters that identifies a unique entity within a given coding scheme. The coding scheme forms the naming authority and the concept code is unique within that space.

CodingSchemeId

A coding scheme is assigned a global, unique name. A coding scheme is a naming authority that manages the set of concept codes as local names within its name space.

QualifiedCode

A globally unique concept code, consisting of the combination of the coding scheme id and the concept code.

VersionId

The unique identifier of a specific version of a terminology service, coding scheme, and value domain. There is no implied ordering on version identifiers. A version identifier may be composed of both letters and digits and must be unique within the context of the given service, coding scheme, or value domain. **VersionId** has a distinguished value, DEFAULT, which represents the "production" or latest validated and ready for use version of the specific entity. The DEFAULT version of an entity is not necessarily the most current.

TerminologyServiceName

The **TerminologyServiceName** serves to uniquely identify an instance of a terminology service. It consists of a globally unique name and the version identifier. There is no equivalent to **TerminologyServiceName** in the model.

2.2.4 Meta Types

```
//*****************************************************************
//    Meta Types
//    See the TerminologyServiceValues module for consts
//*****************************************************************
```

```
typedef QualifiedCode AssociationQualifierId;
typedef sequence<AssociationQualifierId> AssociationQualifierIdSeq;

typedef QualifiedCode LexicalTypeId;
typedef sequence<LexicalTypeId> LexicalTypeIdSeq;

typedef QualifiedCode SourceId;
typedef sequence<SourceId> SourceIdSeq;

typedef QualifiedCode SourceTermUsageId;
typedef sequence<SourceTermUsageId> SourceTermUsageIdSeq;

typedef QualifiedCode SyntacticTypeId;
typedef sequence<SyntacticTypeId> SyntacticTypeIdSeq;

typedef QualifiedCode UsageContextId;
typedef sequence<UsageContextId> UsageContextIdSeq;

typedef ConceptCode AssociationId;
typedef sequence<AssociationId> AssociationIdSeq;

typedef ConceptCode LanguageId;
typedef sequence<LanguageId> LanguageIdSeq;

typedef ConceptCode PresentationFormatId;
typedef sequence<PresentationFormatId> PresentationFormatIdSeq;
```

The meta types above are either concept codes or qualified codes. If the type is a concept code, the coding scheme has been pre-determined and codes from that particular scheme *must* be used when communicating with the terminology service. If the type is a qualified code, the set of valid values may be derived from one or more coding schemes at the discretion of the terminology service vendor. All of the types named below implement the entities of the same name in the abstract model.

2.2.4.1 Qualified Code Types

AssociationQualifierId

A code which qualifies or otherwise provides further information about the occurrence of a target element within an association instance. Association qualifiers are used to describe element- level optionality and to add additional detail about the validity and meaning of the given association instance.

LexicalTypeId

The code for type which may be assigned to a presentation usage. Lexical types are such things as "abbreviation," "Acronym," "Eponym," "Trade name."

SourceId

A code that identifies a book, publication, person, or other citation. It is used to identify the source from which definitions, presentations, and other information within a coding scheme are derived.

SourceTermUsageId

A code that identifies a specific way that a string is used within a source. Examples include "Adjective," "Disease Name," "Language Qualifier."

SyntacticTypeId

A code which identifies a type of variation that a presentation takes from the preferred form within a specific linguistic group. Examples include "spelling variant," "singular," "plural," "word order."

UsageContextId

A code which identifies a specific context in which a presentation associated with a given context code is to be used. Example usage contexts could be such things as “column heading,” “ADT application,” “long textual description.”

2.2.4.2 Coded Concept Types

AssociationId

An identifier of an association type. Base association types are described in Section 3.2.1, “Association,” on page 3-3.

LanguageId

A code that identifies a spoken or written language. Example languages include “English,” “French.”

PresentationFormatId

A code that identifies the format that a given presentation is in. Examples could include “plain text,” “html,” “.wav,” “word 7.0 document.”

Coding Terms

```
//*********************************************************************
//    Coding Terms
//*********************************************************************
interface LexExplorer;
interface CodingSchemeLocator;
interface ValueDomainLocator;
interface CodingSchemeVersion;
interface PresentationAccess;
```

```
interface LinguisticGroupAccess;
interface SystemizationAccess;
interface AdvancedQueryAccess;
interface Systemization;
interface ValueDomainVersion;

typedef string IntlString;
typedef sequence<IntlString> OrderedIntlStringSeq;
typedef sequence<IntlString> IntlStringSeq;
typedef sequence<octet> Blob;
enum Trinary { IS_FALSE, IS_TRUE, IS_UNKNOWN };
typedef sequence<Trinary> TrinarySeq;
typedef sequence<boolean> BooleanSeq;
```

This section creates forward references for the interfaces that follow. This section also defines a set of root types that are used throughout the rest of the interfaces.

IntlString

IntlString represents a string of characters and an identifier that states which character set the string should be presented in. The IDL Extensions Specification [11] states that the character set of string types will be negotiated and converted by the ORBs themselves. Each ORB will have a character set or set of character sets as part of its context. This should provide a sufficient framework for character set identifiers for the time being. The **IntlString** type is maintained as a separate type in case further refinement is needed in a future version.

OrderedIntlStringSeq

This is an ordered list of **IntlStrings**, which is used as a parameter to the **match_concepts_by_keywords** operation. The order determines the importance of each key word in the list.

Blob

The blob is used to carry non-textual presentations and machine-readable instructions. A typical use of this data type might be to return a sound bite of a spoken word.

Trinary

A type which represents one of three possible values. This type is used as a return from several methods that respond to questions about the associations between concept codes. These methods need to have a third return state which indicates that the service has insufficient information to know whether the concept codes are associated or not.

2.2.5 Coded Concept and Coding Scheme Terms

```
//*****************************************************************
//    Coding Scheme and Coded Concept Terms
//*****************************************************************
```

```
typedef string PresentationId;
typedef sequence<PresentationId> PresentationIdSeq;
typedef string LinguisticGroupId;
typedef string SystemizationId;
typedef sequence<SystemizationId> SystemizationIdSeq;

struct CodingSchemeInfo {
    CodingSchemeId  scheme_id;
    VersionId  version_id;
    LanguageId  language_id;
};

struct CodingSchemeVersionRefs {
    CodingSchemeId  coding_scheme_id;
    VersionId  version_id;
    LanguageId  language_id;
    boolean  is_default_version;
    boolean  is_complete_scheme;
    CodingSchemeVersion  coding_scheme_version_if;
    PresentationAccess  presentation_if;
    LinguisticGroupAccess  linguistic_group_if;
    SystemizationAccess  systemization_if;
    AdvancedQueryAccess  advanced_query_if;
};

struct ConceptInfo {
    ConceptCode  a_code;
    IntlString  preferred_text;
};
typedef sequence<ConceptInfo> ConceptInfoSeq;
typedef sequence<ConceptInfoSeq> ConceptInfoSeqSeq;

interface ConceptInfoIter {
    unsigned long max_left();
    boolean next_n(
        in unsigned long n,
        out ConceptInfoSeq concept_info_seq
    );
    void destroy();
};

struct QualifiedCodeInfo {
    QualifiedCode  a_qualified_code;
    IntlString  preferred_text;
};
typedef sequence<QualifiedCodeInfo> QualifiedCodeInfoSeq;

struct Definition {
    IntlString  text;
    boolean  preferred;
    LanguageId  language_id;
```

```
        SourceId  source_id;
};

typedef sequence<Definition> DefinitionSeq;
struct Comment {
        IntlString text;
        LanguageId language_id;
        SourceId source_id;
};
typedef sequence<Comment> CommentSeq;

struct Instruction {
        IntlString  text;
        Blob  formal_rules;
        LanguageId  language_id;
        SourceId  source_id;
};
typedef sequence<Instruction> InstructionSeq;

struct SourceInfo {
        SourceId  source_id;
        SourceTermUsageId  usage_in_source;
        QualifiedCode  code_in_source;
};
typedef sequence<SourceInfo> SourceInfoSeq;

struct PresentationInfo {
        PresentationId  presentation_id;
        PresentationFormatId  presentation_format_id;
        LanguageId  language_id;
        LinguisticGroupId  linguistic_group_id;
};
typedef sequence<PresentationInfo> PresentationInfoSeq;

enum PresentationType {TEXT, BINARY};
union PresentationValue switch(PresentationType) {
        case TEXT : IntlString the_text;
        case BINARY : Blob a_Blob;
};

struct Presentation {
        PresentationId  presentation_id;
        PresentationValue  presentation_value;
};
typedef sequence<Presentation> PresentationSeq;

struct PresentationUsage {
        ConceptCode  concept;
        PresentationId  presentation_id;
        boolean  preferred_for_concept;
        boolean  preferred_for_linguistic_group;
```

```
        SyntacticTypeIdSeq  syntactic_type_ids;
        UsageContextIdSeq  usage_context_ids;
        SourceInfoSeq  source_infos;
        LexicalTypeIdSeq  lexical_type_ids;
};
typedef sequence<PresentationUsage> PresentationUsageSeq;

struct LinguisticGroupInfo {
        LinguisticGroupId linguistic_group_id;
        LanguageId language_id;
        PresentationIdSeq presentation_ids;
};

typedef float Weight;

struct WeightedResult {
        ConceptInfo  the_concept;
        IntlString  matching_text;
        Weight  the_weight;
};
typedef sequence<WeightedResult> WeightedResultSeq;

interface WeightedResultsIter {
        unsigned long max_left();
        boolean next_n(
                in unsigned long n,
                out WeightedResultSeq weighted_results
        );
        void destroy();
};
```

This section defines entities that are used in the coding scheme interface and its components. The list below is alphabetized for easier reference.

CodingSchemeInfo

The IDL specification deviates slightly from the model when it comes to language. The model shows language as a parameter to operations that require it. The IDL specification requires that a language be selected when a coding scheme version interface object is initially referenced. This language is the hidden default for all of the operations that have language as an input parameter. It is necessary to acquire another coding scheme version interface *reference* if interactions are required in another language.

Note – The previous paragraph does not imply that coding scheme versions are language-dependent. There is only one underlying coding scheme version. A given reference acts as a filter, presenting a view of the underlying scheme.

The **CodingSchemeInfo** structure represents a coding scheme identifier, coding scheme version, and language. The **CodingSchemeInfo** uniquely identifies a coding scheme version in the implementation.

CodingSchemeVersionRefs

This structure is returned from the **CodingSchemeLocator** methods **get_coding_scheme_version** and **get_native_coding_scheme_version**. The structure carries common identity information and references to a set of optional interfaces that implement various facets of a coding scheme version object. The content of this structure is identical to the **CodingSchemeVersionAttributes** interface.

Comment

This structure implements the *Comment* class in the abstract model. It includes the comment text, the language in which the comment is written, and an optional source reference.

ConceptInfo

Some of the operations on the following pages return concept codes accompanied by their preferred textual representation. This structure represents one of these elements.

ConceptInfoSeqSeq

This represents a *sequence of sequences* of **ConceptInfo**. It is used as a return type from batch methods, which perform more than one lookup operation per invocation.

Definition

This structure implements the *Definition* class in the abstract model. It includes the definition text, the language in which the text is written, an optional source reference for the definition, and a flag which indicates whether this is the preferred definition for a concept code in the specified language.

Instruction

This structure implements the *Instructions* class in the abstract model. It includes instruction text and/or the formal binary instruction rules. If text is included, the language of the text should be supplied. There is also space for an optional reference to the source of the instructions.

LinguisticGroupId

A unique identifier of a linguistic group within the context of a coding scheme version.

LinguisticGroupInfo

This structure implements the *LinguisticGroup* class in the abstract model. It represents a grouping of syntactically similar presentations. It contains the group identifier and the associated language.

Presentation

This entity implements one portion of the Presentation class in the model. It includes the presentation identifier and the value. The rest of the Presentation class is implemented in the PresentationInfo structure

PresentationId

A unique identifier of a presentation within the context of a coding scheme version.

PresentationInfo

This structure implements a portion of the *Presentation* class in the model. The other part of the *Presentation* class is implemented in the Presentation structure below. It contains the identifier of the presentation and the format. It also contains the presentation language of the presentation and the identifier of the linguistic group to which the presentation belongs, if any.

Note – The model states the presentation identifier is optional. Because some presentations have the potential of being extremely large, it is necessary to succinctly identify each unique presentation in the implementation of this specification. Terminology vendors must supply a unique identifier for each unique presentation. This identifier could potentially be the presentation text itself if it is short. *The presentation identifier should not be stored externally. Different vendors may use different identifiers for identical presentations. Presentation identifiers are to be used strictly as local names of objects for interfacing with an implementation of this specification.*

PresentationType, PresentationValue

These entities implement the *TextualPresentation* and *BinaryPresentation* classes in the model. The TEXT/BINARY types should be viewed and used in the same way as the ASCII/BINARY transfer modes of FTP. A TEXT presentation is restricted to printable characters, and should be usable even if it undergoes character set and other representational transformations between ORBS. A BINARY presentation may contain any information and is guaranteed to be transferred between server and client unaltered.

Note – The **PresentationType** is not dependent upon the presentation format. Clients should be able to handle BINARY or TEXT presentations regardless of the format. It will be possible that a “plain text” format may arrive in a BINARY presentation.

QualifiedCodeInfo

Some of the operations on the following pages return a list of qualified codes along with their preferred or primary textual representation. This structure represents one of these elements.

PresentationUsage

This entity implements the *PresentationUsage* class in the model. It represents the association between a presentation and a concept code. It contains the associated concept code and presentation identifier. It also indicates whether the presentation is the preferred presentation for the concept code in the default language, and whether the presentation is the preferred presentation for the concept code within the linguistic group, if present. The syntactic type identifier(s) state how this presentation varies from the preferred presentation for this concept within the same lexical group. The optional usage context identifier lists the set of contexts in which it is appropriate to use this presentation for this concept code. It also includes an optional set of one or more lexical types which serve to indicate that the terms are "not appropriate for stemming and other natural language techniques." [5]

The optional **SourceInfo** structure set lists all of the sources for this presentation/concept code association.

SourceInfo

This structure implements the *SourceInfo* class in the abstract model. The *SourceInfo* class represents the association between a presentation for a specific concept code and a source of that presentation. It contains the identifier of the source and an optional identifier indicating how the presentation is used in the source. It also may contain qualified code associated with the specific presentation in the source itself. Note that no synonym associations should be inferred between this code and the presentation concept code.

SystemizationId

The name of a specific categorization, classification, or organization of concept codes within a coding scheme.

Weight

This implements *Weight* in the model. It is a relative measure of the "closeness" of a match. The range of the value of a weight is $0.0 \leq \text{value} \leq 1.0$. Weights have no absolute meaning, and may only be compared with other weights that are returned as a sequence from the same method invocation.

WeightedResult

This implements *WeightedResult* in the model, a list entry returned from a match function. It contains a **ConceptInfo** structure, which carries the concept code and preferred text. It also has text of the presentation which was used to determine the match (if any), and the relative weight of this match as compared to the other entries in the return list.

2.2.6 Advanced Query Terms

```
//*****************************************************************
//      Advanced Query Terms
//*****************************************************************

typedef string Constraint;
typedef NamingAuthority::QualifiedNameStr ConstraintLanguageId;
typedef sequence<ConstraintLanguageId> ConstraintLanguageIdSeq;
typedef NamingAuthority::QualifiedNameStr PolicyName;
typedef sequence<PolicyName> PolicyNameSeq;
typedef any PolicyValue;

struct Policy {
    PolicyName name;
    PolicyValue value;
};
typedef sequence<Policy> PolicySeq;
```

The advanced query terms represent the various entities that are used in the advanced query interface. They are described along with the interface in Section 2.2.19, “AdvancedQueryAccess Interface.

2.2.7 Systemization Definitions

```
//*********************************************************
//*       Systemization Terms
//*********************************************************

typedef string RoleName;
typedef string Characteristic;
enum AssociationRole {SOURCE_ROLE, TARGET_ROLE};
enum MinimumCardinality {OPTIONAL, MANDATORY};
enum MaximumCardinality {SINGLE, MULTIPLE};
struct Cardinality {

    MinimumCardinality  minimum;
    MaximumCardinality  maximum;
};

enum ElementType {
    EXTERNAL_CODE_TYPE,
```

```
    LOCAL_CODE_TYPE,
    CHARACTERISTIC_TYPE
};

union RestrictedTargetElement switch(ElementType) {
    case EXTERNAL_CODE_TYPE:QualifiedCode  a_qualified_code;
    case CHARACTERISTIC_TYPE:Characteristic  the_characteristic;
};

union AssociatableElement switch(ElementType) {
    case EXTERNAL_CODE_TYPE:QualifiedCode  a_qualified_code;
    case LOCAL_CODE_TYPE:ConceptCode  a_local_code;
    case CHARACTERISTIC_TYPE:Characteristic  the_characteristic;
};

struct TargetElement {
    AssociatableElement  target_element;
    AssociationQualifierIdSeq  association_qualifiers;
};
typedef sequence<TargetElement> TargetElementSeq;
typedef sequence<TargetElementSeq> TargetElementSeqSeq;
interface TargetElementSeqIter {
    unsigned long max_left();
    boolean next_n(
        in unsigned long n,
        out TargetElementSeqSeq an_element_seq
    );
    void destroy();
};

typedef ConceptCodeAssociationBaseTypeId;

typedef sequence<unsigned long> IndexList;
struct GraphEntry {
    TargetElement      an_entity;
    IndexList          associated_nodes;
};
typedef sequence<GraphEntry> EntityGraph;

struct AssociationDef {
    AssociationId              association_id;
    AssociationBaseTypeId      base_type;
    RoleName                   source_role;
    Cardinality                source_cardinality;
    RoleName                   target_role;
    Cardinality                target_cardinality;
    boolean                    target_is_set;
    boolean                    non_coded_allowed;
    Trinary                    transitive;
    Trinary                    symmetric;
    Trinary                    inherited;
```

```
    Trinary                 source_target_disjoint;

struct AssociationInstance {
    AssociationId           association_id;
    ConceptCode             source_concept;
    TargetElementSeq        target_element_seq;
};
typedef sequence<AssociationInstance> AssociationInstanceSeq;

interface AssociationInstanceIter {
    unsigned long max_left();
    boolean next_n(
        in unsigned long n,
        out AssociationInstanceSeq association_instance_seq
    );
    void destroy();
};

struct ValidationResult {
    boolean is_valid;
    AssociationQualifierId  validity_level;
};

// Constraint - the "any" below must be of type AttributeValuePair. It
// is "any" because IDL won't allow recursive struct definitions
struct RelatedEntityExpression {
    AssociatableElement     associated_element;
    AssociationQualifierIdSeq association_qualifiers;
    any                     base_qualifiers;
};

struct AttributeValuePair {
    AssociationRole         element_role;
    AssociationId           the_association_id;
    RelatedEntityExpression     the_entity_expression;
};
typedef sequence<AttributeValuePair  AttributeValuePairSeq;

struct ConceptExpressionElement {
    ConceptCode             base_code;
    AttributeValuePairSeq   base_qualifiers;
};
typedef sequence<ConceptExpressionElement> conceptExpression;
typedef sequence<ConceptExpression> conceptExpressionSeq;
```

RoleName

A string that serves as a synonym for either the *source* or the *target* portion of an ordered pair of entities in an association type.

Characteristic

Any non-coded “property” or “attribute” associated with a concept code. In the IDL specification, characteristic has been further constrained from the model to only carry string entities.

AssociatableElement

This is a second implementation of the *AssociatableElement* in the abstract model. The **LOCAL_CODE_TYPE** case is used when the *QualifiedCode* belongs to the coding scheme of which the questions are being asked. The **EXTERNAL_CODE_TYPE** case is used when the *QualifiedCode* belongs to a different coding scheme. This distinction serves two purposes. The first is to simplify the interface. The second purpose is to allow the client application to readily determine whether the target concept code is still in the scope of the systemization being queried.

AssociationDef

This structure implements the *Association* class defined in the model. It consists of the association identifier, which uniquely names the association along with an optional base type. There are five possible base types, which are described in Section 3.2.1, “Association,” on page 3-3. **AssociationDef** also includes the source and target role name, whether the target is a set of an individual element, whether the target can include only concept codes or concept codes and characteristics, the source and target cardinality, as well as additional information about the given association.

AssociationInstance

This implements the class by the same name in the abstract model. It consists of an association id identifier, the concept code that is in the source role, and the target element that is in the target role in this instance.

AssociationRole

A tag which is used to determine which role an entity plays in an association. Used in the construction of ConceptExpressions.

AttributeValuePair

This implements the class by the same name in the model. It consists of an association role, which defines the role which the qualified **ConceptExpression** or **RelatedEntityExpression** plays in the contained association. It also contains the identifier of the association as well as the entity that serves the other role in the association.

ConceptExpression

This implements the class by the same name in the model. It represents the logical conjunction of one or more **ConceptExpressionElements**.

ConceptExpressionElement

This represents a base concept code and an optional set of attribute/value qualifiers. It is identical to a **RelatedEntityExpression** with the exception that it may only be a concept code. This reflects the fact that the root node of a concept expression must always be a concept code that is defined in the coding scheme of which the questions are being asked.

EntityGraph

The **EntityGraph** structure provides a mechanism to represent a directed graph of an association. Each node in the graph contains a **TargetElement** and any association qualifiers which may apply to that node. Each node also contains a set of zero-based subscripts into other nodes in the **EntityGraph**. Each index represents a vertex in an association hierarchy.

MinimumCardinality, MaximumCardinality, Cardinality

A partial implementation of *Cardinality* in the abstract model. The following four cardinalities may be represented in this interface: 0..1, 1..1, 0..N, 1..N, which are represented as {OPTIONAL, SINGLE}, {MANDATORY, SINGLE}, {OPTIONAL, MULTIPLE} and {MANDATORY, MULTIPLE} respectively.

RelatedEntityExpression

This implements the class by the same name in the abstract model. It contains the associated element, which may be a qualified code when the entity is in the source role and either a qualified code, a concept code, or a characteristic when the entity is in the target role. It also carries any qualifiers that apply to that particular association instance. Additional **AttributeValuePairs** may further qualify a **RelatedEntityExpression**. As IDL does not allow recursive structure definitions, the “any” node is used to represent the **AttributeValuePair**.

RestrictedTargetElement

This is one implementation of the *AssociatableElement* in the abstract model. It is used as an input in the **LexExplorer list_associated_source_codes** interface.

TargetElement

This implements both the *TargetEntity* and the *TargetElement* classes as defined in the abstract model. *TargetEntity* is defined as being a set of *TargetElements*, while *TargetElement* is defined as being a single element. In this implementation, there is no way to distinguish a single target element from a set that consists of one element. If the distinction is required, it is necessary to refer to the **target_type** of the corresponding **AssociationDef**.

A target element represents the target of an association instance. It contains a **QualifiedCode**, a **ConceptCode,** or a **Characteristic**. It also carries an optional list of association qualifiers.

ValidationResult

This implements the class by the same name in the abstract model. It is returned by the **Systemization** interface operation **validate_concept_expression**. If the expression is considered valid, additional qualifiers may be returned which allow the service to further supply the circumstances in which this would be the case. Example circumstances could include “syntactically valid,” “semantically valid.

2.2.8 Value Domain Terms

```
//*********************************************************************
//      Value Domain Terms
//*********************************************************************

typedef QualifiedCode ValueDomainId;
typedef sequence<ValueDomainId> ValueDomainIdSeq;

interface ValueDomainIdIter {
    unsigned long max_left();
    boolean next_n(
        in unsigned long n,
        out ValueDomainIdSeq value_domain_id_seq
    );
    void destroy();
};

struct PickListEntry {
    QualifiedCode a_qualified_code;
    IntlString pick_text;
    boolean is_default;
};
typedef sequence<PickListEntry> PickListSeq;  //  Ordered

interface PickListIter {
    unsigned long max_left();
    boolean next_n(
        in unsigned long n,
        out PickListSeq pick_list
);
    void destroy();
};
```

ValueDomainId

The value domain identifier is a code that names a field on a screen, a parameter in an interface, a row in a database, or some other external entity in which it is possible to enter coded data. It serves to identify the set of possible concept codes that may be entered into this field. These concept codes may come from one or more different coding schemes.

PickListEntry

This represents a single entry in a PickList. It carries a fully qualified concept code along with a string of text that represents the concept code externally. It also contains a flag to indicate whether the given entry should be presented as being “pre-picked.”

2.2.9 *Terminology Exceptions*

```
//*************************************************************
//          TerminologyService Exceptions
//*************************************************************
//   Used in Multiple Interfaces
//      typically LexExplorer ++

exception NotImplemented{
};
exception UnknownCode {
     ConceptCode bad_code;
};
exception UnknownCodingScheme{
     CodingSchemeId bad_coding_scheme_id;
};
exception UnknownVersion{

     VersionId bad_version_id;
};
exception UnknownValueDomain{
     ValueDomainId bad_value_domain_id;
};
exception NoNativeCodingScheme {
};

exception TranslationNotAvailable {
};
exception TooManyToList {
};
exception NoPickListAvailable {
};
exception AssociationNotInSystemization{
     AssociationId bad_association_id;
};
exception NoSystemizationForCodingScheme {
};
exception ParameterAlignmentError {
};

// CodingSchemeLocator Exceptions

exception LanguageNotSupported {
     LanguageId bad_language_id;
};
```

```
// CodingSchemeVersion exceptions

exception NoPreferredText {
};
exception NoTextLocated {
};

// PresentationAccess exceptions

exception PresentationNotInCodingSchemeVersion{
    PresentationId bad_presentation_id;
};
exception NoPreferredPresentation{
};
exception UnknownPresentationFormat{
    PresentationFormatId bad_presentation_format_id;
};
exception NoPresentationLocated{
};

// LinguisticGroupAccess exceptions

exception LinguisticGroupNotInCodingSchemeVersion{
    LinguisticGroupId bad_linguistic_group_id;
};

// AdvancedQueryAccess exceptions
exception IllegalConstraint {
    Constraint bad_constraint;
};
exception IllegalPolicyName {
    PolicyName name;
};
exception DuplicatePolicyName {
    PolicyName name;
};
exception PolicyTypeMismatch {
    Policy bad_policy;
};
// SystemizationAccess exceptions

exception NoDefaultSystemization{
};
exception UnknownSystemization {
    SystemizationId systemization_id;
};

// Systemization Exceptions

exception ConceptNotExpandable {
    ConceptCode the_concept;
```

```
};
exception NoCommonSubtype {
};
exception NoCommonSupertype {
};
exception InvalidExpression {
    ConceptExpression the_expression;
};
exception UnabletoEvaluate {
    ConceptExpression the_expression;
};
```

These are the possible exceptions that might be raised by the operations described on the following pages. The significance of the individual exceptions will be explained in the context of the operation(s) that may raise them.

2.2.10 TranslationLibrary Interface

```
// ***************************************************************
//      Translation Library
// ***************************************************************

interface TranslationLibrary{

    exception InvalidQualifiedName {
    };
    QualifiedCodestr_to_qualified_code(
        in NamingAuthority::QualifiedNameStr qualified_name_str
    ) raises (
        InvalidQualifiedName
    );

    NamingAuthority::QualifiedNameStr qualified_code_to_name_str(
        in QualifiedCode qualified_code
    );
};
```

The **TranslationLibrary** interface describes a set of functions which will typically be locally implemented.

str_to_qualified_code

This function takes a qualified name string and translates it into the corresponding qualified code structure. The InvalidQualifiedName exception is thrown if the string format is unrecognizable or invalid.

qualified_code_to_name_str

This function converts a qualified code into the corresponding qualified name string.

2.2.11 TerminologyService Interface

```
// ************************************************************
//       TerminologyService
// ************************************************************

interface TerminologyService{

    readonly attribute TerminologyServiceNameterminology_service_name;

    readonly attribute LexExplorer lex_explorer;
    readonly attribute CodingSchemeLocator coding_scheme_locator;
    readonly attribute ValueDomainLocator value_domain_locator;

    CodingSchemeId  Seq   get_coding_scheme_ids();

    CodingSchemeInfo  get_native_coding_scheme_info(
    ) raises (
        NoNativeCodingScheme
    );
};
```

The Lexicon Query Service is based on a component model patterned after the OMG Trader Service. [2] This pattern makes it possible for a client to obtain a reference to any of the primary terminology service interfaces and easily discover which other interfaces have been implemented. It is expected that as possible future interfaces for Terminology Services are defined (such as authoring), the component will be expanded to accommodate those interfaces.

A terminology service has one mandatory and two optional interfaces that it may implement. There are a variety of systems and applications that may require different functionality from a terminology service. If multiple objects are used to implement the component they must all maintain consistency so the client can treat them as a single terminology service. That is, all the attributes on the **TerminologyService** must return identical results.

terminology_service_name

Each implementation instance of **TerminologyService** must have a unique name. The chosen name does not necessarily need to have any meaning. The name makes it possible for clients traversing a graph of **TerminologyServices** to recognize services they have encountered before. The name is static over time. The version within the name can change over time to represent different revisions of the same service.

If there are two or more objects with the same terminology service name they must be replicas of each other. The mechanism used to maintain consistency between the replicas is implementation-dependent and is not exposed as standard interfaces are.

lex_explorer

The **LexExplorer** interface provides a subset of terminology services which are useful for many of the common use cases in a single interface that is (hopefully) simple to understand and use. All terminology service vendors must implement this interface.

coding_scheme_locator

The **CodingSchemeLocator** allows exploration of the functionality supported by the various coding schemes and navigation to a **CodingSchemeVersion** that meets the criteria of the client. This interface is optional and may not be present in all terminology service implementations. If not present, this attribute should return the NULL object reference.

value_domain_locator

The **ValueDomainLocator** interface allows discovery of the value domains which are implemented by the **TerminologyService**, and navigation to a **ValueDomainVersion** that meets the criteria of the client. This interface is optional and may not be present in all terminology service implementations. If not present, this attribute should return the NULL object reference.

get_coding_scheme_ids

This provides an unordered list of all the coding scheme identifiers that are provided by this terminology service. This implements the *TerminologyService.GetAllCodingSchemes* from the abstract model.

get_native_coding_scheme_info

Returns information about the coding scheme that is designated as “native” by the terminology service vendor, along with the current default version of that scheme and the preferred language used by that coding scheme. An exception is thrown if there has been no coding scheme designated as native by the service provider.

2.2.12 LexExplorer Interface

```
// ***************************************************************
//       LexExplorer
// ***************************************************************

interface LexExplorer : TerminologyService{

    IntlString  get_preferred_text(
        in QualifiedCode  a_qualified_code,
        in UsageContextId  Seq context_ids
    ) raises (
        UnknownCodingScheme,
        UnknownCode
    );

    IntlStringSeq get_preferred_text_for_concepts(
```

```
        in QualifiedCodeSeq qualified_codes,
        in UsageContextIdSeq context_ids
);

Definition get_preferred_definition(
        in QualifiedCode qualified_code
) raises (
        UnknownCodingScheme,
        UnknownCode
);

ConceptInfoSeq translate_code(
        in QualifiedCode from_qualified_code,
        in CodingSchemeId to_coding_schemeId
) raises (
        UnknownCode,
        UnknownCodingScheme,
        TranslationNotAvailable
);

ConceptInfoSeqSeq translate_codes(
        in QualifiedCodeSeq from_qualified_codes,
        in CodingSchemeId to_coding_scheme_id
) raises (
        UnknownCodingScheme
);

void list_concepts(in CodingSchemeId coding_scheme_id,
        in unsigned long how_many,
        out ConceptInfoSeq concept_info_seq,
        out ConceptInfoIter concept_info_iter
) raises (
        UnknownCodingScheme,
        TooManyToList
);

void list_value_domain_ids (
        in unsigned long how_many,
        out ValueDomainIdSeq value_domain_ids,
        out ValueDomainIdIter value_domain_id_iter
) raises (
        TooManytoList
);

boolean is_concept_in_value_domain (
        in QualifiedCode qualified_code,
        in ValueDomainId value_domain_id
) raises (
        UnknownValueDomain
);
```

```
TrinarySeq are_concepts_in_value_domains (
      in QualifiedCodeSeq  qualified_codes,
      in ValueDomainIdSeq value_domains
) raises (
      ParameterAlignmentError
);

void get_pick_list(
      in ValueDomainId value_domain_id,
      in UsageContextIdSeq context_ids,
      out PickListSeq pick_list,
      out PickListIter pick_list_iter
) raises (
      TooManyToList,
      UnknownValueDomain,
      NoPickListAvailable
);

Trinary association_exists(
      in QualifiedCode  source_code,
      in TargetElement  target_element,
      in AssociationId  association_id,
      in boolean  direct_only
) raises (
      AssociationNotInSystemization,
      NoSystemizationForCodingScheme,
      UnknownCode
);

TrinarySeq associations_exist(
      in QualifiedCodeSeq  source_codes,
      in TargetElementSeq  target_elements,
      in AssociationIdSeq  association_ids,
      in boolean  direct_only
) raises (
      ParameterAlignmentError
);

void list_associated_target_elements (
      in QualifiedCode  qualified_code,
      in AssociationId  association_id,
      in boolean  direct_only,
      in unsigned long  how_many,
      out TargetElementSeqSeq  related_target_seq,
      out TargetElementSeqIter   related_target_iter
) raises (
      AssociationNotInSystemization,
      NoSystemizationForCodingScheme,
      UnknownCode
);
```

```
        void list_associated_source_codes (
            in RestrictedTargetElement  target_element,
            in CodingSchemeId  source_coding_scheme_id,
            in AssociationId  association_id,
            in boolean  direct_only,
            in unsigned long  how_many,
            out ConceptInfoSeq  concept_info_seq,
            out ConceptInfoIter  concept_info_iter
        ) raises (
            AssociationNotInSystemization,
            NoSystemizationForCodingScheme,
            UnknownCode
        );
};
```

The **LexExplorer** interface provides a simplified or "flattened" interface to some of the more common terminology service functions. A terminology service may consist solely of the **LexExplorer** interface if appropriate. When other interfaces are implemented within the terminology service, a functionally equivalent operation in **LexExplorer** and the more complete **CodingSchemeVersion** or **ValueDomainVersion** must return identical results.

get_preferred_text

This operation returns the preferred text associated with the **QualifiedCode** when supplied with an ordered list of contexts. This operation must ignore any contexts that it does not recognize. The language is established outside of the scope of this specification. The result will be identical to what would be returned by the **get_preferred_text** operation on the default **CodingSchemeVersion** whose **CodingSchemeId** matches the one in the Qualified Code where the concept is defined. An exception is thrown if the coding scheme of the qualified code is not supported or recognized by the terminology service, or if the concept code is not included in the particular version of the coding scheme supported by the service.

get_preferred_text_for_concepts

This is a batch equivalent of the **get_preferred_text** operation above. A list of qualified codes is supplied and a corresponding list of text strings is returned. The single list of contexts applies to all of the supplied contexts. A null string ("") in a return slot indicates that a problem has occurred and the text was not located. The **get_preferred_text** operation may then be invoked to determine exactly what the problem was.

get_preferred_definition

This operation returns the preferred definition of the **QualifiedCode**. The language is established outside of the scope of this specification. The result will be identical to what would be returned by the **get_preferred_definition** operation if the default **CodingSchemeVersion** whose **CodingSchemeId** matches the one in the Qualified Code where the concept is defined. An exception is thrown if the coding scheme of

the qualified code is not supported or recognized by the terminology service, or if the concept code is not included in the version of the coding scheme supported by the service.

translate_code

This operation will translate the supplied code into a list of synonymous codes from the target coding scheme. If a translation cannot be performed, the result should be returned as follows:

- If the terminology service knows about the code to be translated and the target coding scheme, it should assert that no translation exists by returning a zero length list.
- If the coding scheme in the qualified code or the **to_coding_schemeId** is not recognized, the operation should throw the UnknownCodingScheme exception.
- If the concept code in the qualified code is not recognized, the operation should throw the UnknownCode exception.
- If the translation function is not supported for the supplied code or target coding scheme, the operation should throw the TranslationNotAvailable exception.

translate_codes

This operation is the batch equivalent of the **translate_code** operation above. It is supplied with a list of concept codes which are to be translated into the target coding scheme and then it returns a corresponding list of translation results. This method will throw the UnknownCodingScheme exception if the **to_coding_scheme_id** parameter is not recognized. All other problems will be reflected by a zero length sequence in the corresponding slot. The client will need to use the **translate_code** operation to determine exactly why the translation did not occur.

list_concepts

This provides the ability to list all of the concept codes supported by a given coding scheme. The intended purpose of this **LexExplorer** interface is to provide list access to relatively small coding schemes. If the number of concept codes in the coding scheme exceeds the **how_many** parameter, the terminology vendor may choose the TooManyToList exception rather than return the list. The UnknownCodingScheme exception is thrown if the supplied coding scheme id is not supported and/or recognized.

list_value_domain_ids

This provides the ability to list all of the value domains supported by the terminology service. The intended purpose of this interface as specified within the **LexExplorer** is to provide list access to a relatively small set. If the number of value domains in the terminology service exceeds the **how_many** parameter, the terminology vendor may choose the TooManyToList exception rather than return the list.

is_concept_in_value_domain

This returns TRUE if the supplied concept belongs to the supplied value domain, FALSE otherwise. An exception is thrown if the supplied value domain is not recognized.

are_concepts_in_value_domains

This is a batch equivalent of the previous operation. A list of qualified codes is supplied along with a corresponding list of value domains. The corresponding return sequence returns TRUE if the concept belongs, FALSE if it doesn't, and UNKNOWN if the corresponding value domain is not recognized. The ParameterAlignmentError exception is thrown if the number of value domains is not the same as the number of qualified codes.

get_pick_list

This returns an appropriate pick list, if any, for the supplied value domain and ordered set of usage context identifiers. An exception is thrown if the value domain is unrecognized, if the service is unable to provide a pick list for the value domain, or if the pick list is deemed too large in the opinion of the terminology service vendor.

association_exists

This operation returns true if an association instance of the supplied type exists between the qualified source code and the target element. The association is tested in terms of the default systemization of the coding scheme named in the source code. If the coding scheme does not have a default systemization, the NoSystemizationForCodingScheme exception is thrown. The **direct_only** flag indicates whether the transitive closure of the association is to be considered (FALSE), or only immediate children (TRUE). An exception is thrown if the named association is not included in the systemization, if the coding scheme of the source or target element is not recognized, or if the source or target concept code does not belong to the supplied coding scheme.

associations_exist

This is the batch equivalent of the above operation. It takes a list of source codes, a list of target elements and a list of association identifiers, and returns a list of results. The **direct_only** flag applies to all of the elements. If an association is not recognized or a code is not recognized, an **UNKNOWN** value is returned. The client may use the **association_exists** function to determine what went wrong. The ParameterAlignmentError exception is thrown if there isn't the same number of source, target, and association elements.

list_associated_target_elements

This operation returns a list of target elements associated with the supplied qualified code via the supplied association. The default systemization for the coding scheme supplied in the qualified code is used. If the coding scheme does not have a default systemization, the NoSystemizationForCodingScheme exception is thrown. The **direct_only** flag indicates whether the transitive closure of the association is to be

considered (FALSE), or only immediate children (TRUE). An exception is thrown if the named association is not included in the systemization, if the coding scheme of the source element is not recognized, or if the source concept code does not belong to the supplied coding scheme.

list_associated_source_elements

This operation returns a list of source elements associated with the supplied target element via the supplied association. The default systemization for the supplied coding scheme is used. If the coding scheme does not have a default systemization, the NoSystemizationForCodingScheme exception is thrown. The **direct_only** flag indicates whether the transitive closure of the association is to be considered (FALSE), or only immediate children (TRUE). An exception is thrown if the named association is not included in the systemization, if the coding scheme of the target element is not recognized, or if the source or target concept code does not belong to the supplied coding scheme.

2.2.13 CodingSchemeLocator Interface

```
// ************************************************************
//       CodingSchemeLocator
// ************************************************************
interface CodingSchemeLocator:TerminologyService{

    VersionIdSeq get_version_ids(
        in CodingSchemeId coding_scheme_id
    ) raises (
        UnknownCodingScheme
    );

    LanguageIdSeq get_supported_languages(
        in CodingSchemeId coding_scheme_id
    ) raises (
        UnknownCodingScheme
    );

    CodingSchemeVersionRefs get_coding_scheme_version(
        in CodingSchemeId coding_scheme_id,
        in VersionId version_id,
        in LanguageId language_id
    ) raises (
        UnknownCodingScheme,
        UnknownVersion,
        LanguageNotSupported
    );

    CodingSchemeVersionRefs get_native_coding_scheme_version(
    ) raises(
        NoNativeCodingScheme
```

```
    );

    VersionId get_last_valid_versions (
        in ConceptCode a_code
    ) raises (
        UnknownCode
    );
};
```

The **CodingSchemeLocator** component provides navigational capabilities to enumerate and access the **CodingScheme** and **CodingSchemeVersion** objects as defined in the abstract model. The **CodingScheme** object and methods have been flattened into the **CodingSchemeLocater** interface.

get_version_ids

Returns an ordered list of all of the version identifiers supported in the named coding scheme. The list is ordered chronologically from latest to earliest. An exception is thrown if the supplied coding scheme identifier is not recognized by the terminology service. This implements *CodingScheme.GetAllVersions* from the abstract model.

get_supported_languages

Returns a list of all the languages supported by the named coding scheme. An exception is thrown if the supplied coding scheme identifier is not recognized by the terminology service. This indirectly implements *CodingSchemeVersion.ListSupportedLanguages* from the abstract model. Note that some of the languages returned by this method may not be supported by all versions of the coding scheme.

get_coding_scheme_version

Returns a reference to the named **CodingSchemeVersion** object, given the name of the coding scheme, the name of the desired version and language in which the version object is to communicate. The version identifier may be set to DEFAULT, which specifies that the current production version is desired. This method will throw an exception if the coding scheme is not recognized, the version is not recognized, or the language is not supported in the coding scheme version. This implements both the **GetCodingSchemeVersion** and **GetDefaultCodingSchemeVersion** methods. It also serves to validate and establish the language identifier that will be used in language-dependent operations.

get_native_coding_scheme_version

Returns a reference to the default version of the native coding scheme using the preferred language. The service provider throws an exception if there has been no coding scheme designated as native. This implements the *TerminologyService.GetNativeCodingScheme* method.

get_last_valid_version

Returns the identifier of the chronologically most recent coding scheme version that contains the supplied concept code. Typically the version identifier will be that of the latest version except in cases where concept codes have become obsolete. A null version identifier is returned if there is no longer any version which contains the supplied concept code. An exception is thrown if the concept code does not belong to the coding scheme.

2.2.14 *ValueDomainLocator Interface*

```
// ************************************************************
//       ValueDomainLocator
// ************************************************************
interface ValueDomainLocator:TerminologyService {

    void list_value_domain_ids(
        in unsigned long how_many,
        out ValueDomainIdSeq value_domain_ids,
        out ValueDomainIdIter value_domain_id_iter
    );

    VersionIdSeq get_version_ids(
        in ValueDomainId value_domain_id
    ) raises(
        UnknownValueDomain
    );

    ValueDomainVersion get_value_domain_version(
        in ValueDomainId value_domain_id,
        in VersionId version_id
    ) raises(
        UnknownValueDomain,
        UnknownVersion
    );

    ValueDomainIdSeq get_value_domain_ids_for_concept (
        in QualifiedCode qualified_code
    );
};
```

The **ValueDomainLocator** component provides the navigational capabilities necessary to enumerate and access the *ValueDomain* and *ValueDomainVersion* objects as defined in the model. The *ValueDomain* object and methods have been flattened into the **ValueDomainLocater** interface.

get_version_ids

This operation returns an ordered list of all of the supported versions in a specific value domain. The list is ordered chronologically from latest to earliest. An exception is thrown if the supplied coding scheme identifier is not recognized and/or supported by the terminology service. This implements the *ValueDomain.GetAllVersions* method in the model.

get_value_domain_version

This operation returns a reference to the named **ValueDomainVersion** object, given the identification of the value domain and the desired version. The version identifier may be set to DEFAULT, which specifies that the current production version is desired. This implements both the *ValueDomain.GetValueDomainVersion* and *ValueDomain.GetDefaultVersion* methods.

get_value_domain_ids_for_concept

This operation will return the list of value domain ids which include this concept in their current default version.

2.2.15 CodingSchemeAttributes Interface

```
//*************************************************************
//        CodingScheme interfaces
//*************************************************************

//*************************************************************
// A coding scheme consists of the following interfaces
//          interface CodingSchemeVersion:CodingSchemeVersionAttributes
//          interface PresentationAccess:CodingSchemeVersionAttributes
//          interface LinguisticGroupAccess:CodingSchemeVersionAttributes
//          interface SystemizationAccess:CodingSchemeVersionAttributes
/           interface AdvancedQuery:CodingSchemeVersionAttributes
//*************************************************************
//*************************************************************
//          interface CodingSchemeVersionAttributes
//*************************************************************
interface CodingSchemeVersionAttributes {
     readonly attribute CodingSchemeId coding_scheme_id;
     readonly attribute VersionId version_id;
     readonly attribute LanguageId language_id;
     readonly attribute boolean is_default_version;
     readonly attribute boolean is_complete_scheme;
     readonly attribute CodingSchemeVersion coding_scheme_version_if;
     readonly attribute PresentationAccess presentation_if;
     readonly attribute LinguisticGroupAccess linguistic_group_if;
     readonly attribute SystemizationAccess systemization_if;
     readonly attribute AdvancedQueryAccess advanced_query_if;
};
```

CodingSchemeVersionAttributes is an abstract interface that is inherited by the **CodingSchemeVersion**, **PresentationAccess**, **LinguisticGroupAccess**, **SystemizationAccess**, and **AdvancedQuery** interfaces. All of these interfaces are tightly coupled in the component model and must all return identical values for these attributes. Each of the attributes is defined below:

coding_scheme_id

The identifier of the coding scheme represented by any of these interfaces.

version_id

The version of the coding scheme represented by any of these interfaces.

language_id

The language which is represented by any of these interfaces.

is_default_version

A flag that indicates whether this version was considered the default version for the coding scheme at the time that the interface was acquired.

is_complete_scheme

A flag that indicates whether the version of the coding scheme is considered "complete" or exhaustive by the terminology vendor or whether it represents a subset of the total named scheme.

CodingSchemeVersion,
PresentationAccess,
LinguisticGroupAccess,
SystemizationAccess,
AdvancedQuery

All of these interfaces are optional. If not supplied by the terminology vendor, these attributes return a reference to the NULL object. They are described in the following pages.

2.2.16 CodingSchemeVersion Interface

```
//************************************************************
//          interface CodingSchemeVersion
//************************************************************

interface CodingSchemeVersion : CodingSchemeVersionAttributes {

    SyntacticTypeIdSeq get_syntactic_types();
    SourceTermUsageIdSeq get_source_term_usages();
    SourceIdSeq get_scheme_source_ids();
    UsageContextIdSeq get_usage_contexts();
```

```
void list_concepts(
    in unsigned long how_many,
    out ConceptInfoSeq concept_info_seq,
    out ConceptInfoIter concept_info_iter
);

boolean is_valid_concept(
    in ConceptCode a_code
);

DefinitionSeq get_definitions(
    in ConceptCode a_code
) raises(
    UnknownCode
);
Definition get_preferred_definition(
    in ConceptCode a_code
) raises(
    UnknownCode
);

CommentSeq get_comments(
    in ConceptCode a_code
) raises (
    NotImplemented,
    UnknownCode
);

InstructionSeq get_instructions(
    in ConceptCode a_code
) raises (
    NotImplemented,
    UnknownCode
);

IntlStringSeq get_all_text(
    in ConceptCode a_code
) raises (
    UnknownCode
);

IntlString get_preferred_text (
    in ConceptCode a_code
) raises (
    UnknownCode
    NoPreferredText
);

IntlString get_text_for_context(
    in ConceptCode a_code,
```

```
        in UsageContextIdSeq context_ids
    ) raises (
        UnknownCode,
        NoTextLocated
    );

    ConceptCodeSeq get_concepts_by_text(
        in string text
    );

    void match_concepts_by_string(
        in IntlString match_string,
        in unsigned long how_many,
        out WeightedResultSeq weighted_results,
        out WeightedResultsIter weighted_result_iter
    ) raises (
        NotImplemented
    );

    void match_concepts_by_keywords(
        in OrderedIntlStringSeq keywords,
        in unsigned long how_many,
        out WeightedResultSeq weighted_results,
        out WeightedResultsIter weighted_results_iter
    ) raises(
        NotImplemented
    );
};
```

The **CodingSchemeVersion** interface implements portions of both the *CodingSchemeVersion* and the *ConceptDescription* interface as described in the abstract model. Presentations, linguistic groups, and systemizations have been factored into separate interfaces.

get_syntactic_types,
get_source_term_usages,
get_scheme_source_ids,
get_usage_contexts

These operations implement the methods *ListAllSyntacticTypes, ListSourceTermUsages, ListSchemeSources,* and *ListSupportedUsages* respectively. They allow run-time discovery of the entities supported by this version of the coding scheme.

list_concepts

This implements the *GetAllConcepts* method. It returns an iterator of all the concept codes defined in this version of the coding scheme. Note that the iterator returns both the concept code and the preferred presentation.

is_valid_concept

Returns TRUE if the supplied concept code is valid for this particular version of the coding scheme, FALSE otherwise. This implements the *IsValidConcept* method in the model.

get_definitions

Returns all of the definitions for the supplied concept code in the language specified for this object reference. An exception is thrown if the concept code is not recognized. This implements the *ConceptDescription.GetDefinitions(FALSE)* method in the model.

get_preferred_definition

Returns the preferred definition for the supplied concept code, if any. An exception is thrown if the concept code is not recognized. This implements the *ConceptDescription.GetDefinitions(FALSE)* method in the model.

get_comments

Returns any comments associated with the concept code in the language specified for the **CodingSchemeVersion** object reference. An exception is thrown if the concept code is not recognized. This implements the *ConceptDescription.Comments* method in the model.

get_instructions

Returns any instructions associated with the concept code. The language is ignored. An exception is thrown if the concept code is not recognized. This implements the *ConceptDescription.GetInstructions* method in the model.

get_all_text

Returns all plain text ASCII presentations associated with the supplied concept code. An exception is thrown if the concept code is not recognized. This partially implements the *ConceptDescription.GetAllPresentations* method in the model.

get_preferred_text

Returns the preferred text in the language specified for the **CodingSchemeVersion** object reference. An exception is thrown if no text is preferred in the given language, or the concept code is not recognized. This partially implements the *ConceptDescription.GetPreferredPresentation* method in the model.

get_text_for_context

Returns the appropriate text in the language specified for the **CodingSchemeVersion** object reference for the supplied list of context identifiers. The list of context identifiers should be ordered from most to least important. The service ignores unknown context identifiers when searching for the matching text. An exception is thrown if the concept code is not recognized or the service is unable to come up with the appropriate text. This partially implements the *ConceptDescription.GetPresentation* method in the model.

get_concepts_by_text

Returns a list of all concept codes that have a textual presentation in the language specified for the **CodingSchemeVersion** object reference which matches exactly the supplied text. This implements the *CodingSchemeVersion.GetConceptsByText* method in the model.

match_concepts_by_string

Returns a weighted list of concept codes which have text that matches the supplied string. The weighted list is ordered by match likelihood with the most likely matches occurring first in the list. The returned list contains the concept code, the relative likelihood of match ($0.0 < \text{likelihood} \leq 1.0$) along with the textual string which matched, and the preferred presentation for the matching concept code.

The default matching algorithm recognizes the asterisk (*), question mark (?) and back slash (\) as special characters, which represent zero or more matching characters, exactly one matching character and the escape character respectively. Case is ignored during the matching process. Thus the match string “Card*ly” would match all textual presentations which began with “card” regardless of case and ended with “ly.” Similarly, the string “wid??et” would match all seven character strings beginning with “wid” and ending with “et.” This implements *MatchConceptsByString* from the abstract model. This operation is optional and the *NotImplemented* exception should be thrown if it is not implemented.

Note – It is anticipated that terminology vendors may extend this algorithm substantially. For this reason, there are no conformance points specified regarding the set of elements to be returned. The set of concept codes and weights returned are entirely at the discretion of the terminology vendor.

---

match_concepts_by_keywords

This is identical to the **match_concepts_by_string** with the exception that a list of “keywords” is provided instead of a single match string. The supplied keyword list is ordered, with the highest priority being assigned to the first word in the list. Keywords may have the “*” and “?” wild cards embedded. A keyword may not have a white space character embedded (e.g., tab, “ “, etc.), as it is intended to match exactly one word. The terminology services return concept codes that have text matching one or more of the supplied keywords. This implements *MatchConceptsByKeywords* from the abstract model. This operation is optional and the *NotImplemented* exception should be thrown if it is not implemented.

Note – As with the proceeding match function, it is the intention of this specification to give the terminology service provider a fair amount of leeway in how the match strings are interpreted.

2.2.17 PresentationAccess Interface

```
//*************************************************************
//        PresentationAccess
//*************************************************************
interface PresentationAccess : CodingSchemeVersionAttributes {

    PresentationFormatIdSeq get_presentation_format_ids();

    Presentation get_presentation(
        in PresentationId presentation_id
    ) raises(
        PresentationNotInCodingSchemeVersion
    );

    PresentationInfo get_presentation_info(
        in PresentationId presentation_id
    ) raises(
        PresentationNotInCodingSchemeVersion
    );

    PresentationUsageSeq get_presentation_usages(
        in PresentationId presentation_id
    ) raises(
        PresentationNotInCodingSchemeVersion
    );

    PresentationUsageSeq get_all_presentations_for_concept(
        in ConceptCode a_code
    ) raises(
        UnknownCode
    );

    PresentationUsage get_preferred_presentation(
        in ConceptCode a_code,
        in PresentationFormatId presentation_format_id
    ) raises(
        UnknownPresentationFormat,
        UnknownCode,
        NoPreferredPresentation
    );

    PresentationUsage get_presentation_for_context(
        in ConceptCode a_code,
        in UsageContextIdSeq context_ids,
        in PresentationFormatId presentation_format_id
    ) raises (
        UnknownPresentationFormat,
        UnknownCode,
        NoPresentationLocated
```

```
    );

    PresentationUsage get_all_presentations_for_context (
        in ConceptCode a_code,
        in UsageContextIdSeq context_ids,
        in PresentationFormatId presentation_format_id
    ) raises (
        UnknownPresentationFormat,
        UnknownCode,
        NoPresentationLocated
    );
};
```

The **PresentationAccess** component is optional and may or may not be implemented depending upon the needs of the client and terminology service vendor. It provides a more "sophisticated" level of access to presentations and their associated entities. The various operations are described below.

get_presentation_format_ids

Returns a list of all the presentation formats supported by this module. This implements *CodingSchemeVersion.ListSupportedPresentationFormats* from the abstract model.

get_presentation

Returns the actual presentation given a presentation identifier. An exception is thrown if the presentation identifier is not in the coding scheme. This implements a portion of *CodingSchemeVersion.GetPresentation* from the abstract model.

get_presentation_info

Returns a presentation info structure given a presentation identifier. An exception is thrown if the presentation identifier is not in the coding scheme. This implements a portion of *CodingSchemeVersion.GetPresentation* from the abstract model. This operation and **get_presentation** above fully implement *GetPresentation*.

get_presentation_usages

Returns a list of all the **PresentationUsage** structures that reference the supplied presentation identifier. An exception is thrown if the presentation id is not in the coding scheme version. This implements *Presentation.GetPresentationUsages* from the abstract model.

get_all_presentations_for_concept

Returns a list of all the **PresentationUsage** structures that contain the supplied concept code. These structures may then be used to acquire further information about the associations and the presentation itself. An exception is thrown if the concept code is not valid in the coding scheme. This implements *ConceptDescription.GetAllPresentations* from the abstract model.

get_preferred_presentation

Returns the **PresentationUsage** structure that represents the concept code/presentation association that is preferred for the supplied concept code in the default language. An exception is thrown if the presentation format is not recognized, if the concept code is not valid in the coding scheme version, or if there is no preferred presentation in this format and/or language. This implements *ConceptDescription.GetPreferredPresentation* from the abstract model.

get_presentation_for_context

Returns the **PresentationUsage** structure that represents the code/presentation association that is most appropriate for ordered context list for the supplied concept code and format. The list of contexts is ordered by the relative importance to the calling application. The service ignores unrecognized contexts in the list. An exception is thrown if the presentation format is not recognized, if the concept code is not valid in the coding scheme version, or if there is no preferred presentation in this format. This implements *ConceptDescription.GetPresentationForContext* from the abstract model.

get_all_presentations_for_context

This operation is identical to the preceding operation with the exception that it returns *all* of the **PresentationUsage** structures that could be appropriate. It implements *ConceptDescription.GetAllPresentationsForContext* from the model.

2.2.18 LinguisticGroupAccess Interface

```
//*************************************************************
//      LinguisticGroupAccess
//*************************************************************
interface LinguisticGroupAccess : CodingSchemeVersionAttributes {

    LexicalTypeIdSeq get_lexical_types();

    LexicalGroupInfo get_lexical_group(
        in LexicalGroupId lexical_group_id
    ) raises(
        LexicalGroupNotInCodingSchemeVersion
    );
}
```

The **LinguisticGroupAccess** component is an optional component. A linguistic group associates one or more syntactically similar presentations. The interface provides the two operations listed below. (The method *CodingSchemeVersion.GetAllLinguisticGroups* is implemented in the **TerminologyService** base module.)

get_linguistic_group

Returns a structure which represents the named lexical group. An exception is thrown if the group is not recognized. This implements the *CodingSchemeVersion.GetLinguisticGroup* method.

2.2.19 AdvancedQueryAccess Interface

```
//************************************************************
//      AdvancedQueryAccess
//************************************************************

interface AdvancedQueryAccess : CodingSchemeVersionAttributes {
    readonly attribute PolicyNameSeq supported_policies;
    readonly attribute ConstraintLanguageIdSeq
        supported_constraint_languages;

    struct query_policies {
        unsigned long return_maximum;
        boolean concept_as_source;
        boolean concept_as_target;
        boolean current_scheme_only;
        boolean direct_associations_only;
    };

    void query (
        in Constraint constr,
        in PolicySeq search_policy,
        in unsigned long how_many,
        out WeightedResultSeq results,
        out WeightedResultsIter results_iter
    ) raises (
        IllegalConstraint,
        IllegalPolicyName,
        PolicyTypeMismatch,
        DuplicatePolicyName
    );
}
```

The **AdvancedQueryAccess** interface is an optional interface which provides a means by which a client can enumerate concepts that satisfy multiple associations.

The constraint "**constr**" is the means by which the client states the requirements. If the "**constr**" does not obey the syntax rules for a legal constraint expression, then an IllegalConstraint exception is raised.

The OMG Trader Specification, appendix B [2] defines the OMG Trader constraint language. This document should be considered the specification for the Terminology Service query constraint. The information provided here defines the Property Names that may be used, as well as a summary of the constraint language as applied to terminology services.

A statement in the constraint language is a string. Other constraint languages may be supported by a particular terminology service implementation; the constraint language used by a client of the terminology service is indicated by embedding "<<Identifier major.minor>>" at the beginning of the string. If such an escape is not used, it is equivalent to embedding "<<OMG 1.0>>".

The constraint expressions in a query can be constructed from the Properties defined in this specification. The TerminologyService Values Module includes standard names that can be used as Property values to construct terminology service constraint queries. [2] These property names are defined to searches via **AssociationId** in combination with text and keyword search under control of the client.The constraint language in which these expressions are written consists of the following items (examples of these expressions are shown in square brackets below each bulleted item):

- Comparative functions: == (equality),! = (inequality), >, >=, <, <=, ~ (substring match), in (element in sequence); the result of applying a comparative function is a boolean value ["Cost < 5" implies only consider offers with a Cost property value less than 5; "'Visa' in CreditCards" implies only consider offers in which the CreditCards property, consisting of a set of strings, contains the string 'Visa']
- Boolean connectives: and, or, not ["Cost >= 2 and Cost <= 5" implies only consider offers where the value of the Cost property is in the range 2 <= Cost <= 5]
- Property existence: exist
- Property names
- Numeric and string constants
- Mathematical operators: +, -, *, / ["10 < 12.3 * MemSize + 4.6 * FileSize" implies only consider offers for which the arithmetic function in terms of the value of the MemSize and FileSize properties exceeds 10]
- Grouping operators: (,)

Note that the keywords in the language are case-sensitive. Please see the OMG Trader specification for a complete definition of the constraint language.

The "policies" parameter allows the importer to specify how the search should be performed as opposed to what criteria should be used to determine a match. This can be viewed as parameterizing the algorithms within the terminology service implementation. The "policies" are a sequence of name-value pairs. The names available to an importer depend on the implementation of the terminology service. However, some names are standardized where they represent policies that all terminology services should support. If a policy name in this parameter does not obey the syntactic rules for legal **PolicyName**'s, then an IllegalPolicyName exception is raised. If the type of the value associated with a policy differs from that specified in this specification, then a PolicyTypeMismatch exception is raised. If subsequent processing of a **PolicyValue** yields any errors (e.g., the terminology service determines a policy value is malformed), then an InvalidPolicyValue exception is raised. If the same policy name is included two or more times in this parameter, then the DuplicatePolicyName exception is raised.

The returned concept codes are passed back via a **WeightedResultSeq** which has iterator access provided.

The following standard policies are defined for the terminology service query operation:

concept_as_source

Specifies whether to include concepts that participate as the source in the association.

concept_as_target

Specifies whether to include concepts that participate as the target in an association named in the query.

current_scheme_only

Specifies whether to include concepts that may be defined as synonyms in this coding scheme, but are not a part of the coding scheme. Setting this to TRUE will limit the concepts returned to those that are a part of the current coding scheme.

direct_associations_only

Specifies whether or not to include concepts which participate in an association via inheritance rather than directly. Setting this to TRUE would limit the returned concepts to those that are directly involved in the named association. Setting this to FALSE would allow the transitive closure of the particular named association to be included. Behavior would be identical to that in the **are_concepts_related** operation of the systemization. Specifically, subtypes and other associations are not included in the results of the query.

2.2.20 SystemizationAccess Interface

```
//************************************************************
//      SystemizationAccess
//************************************************************
interface SystemizationAccess : CodingSchemeVersionAttributes {
    SystemizationIdSeq get_systemization_ids();

    Systemization get_systemization(
        in SystemizationId systemization_id
    ) raises(
        UnknownSystemization
    );

    Systemization get_default_systemization(
    ) raises(
        NoDefaultSystemization
    );
}
```

The **SystemizationAccess** interface is an optional component. It provides access to **Systemizations** associated with a coding scheme version.

get_systemization_ids

Returns a list of all the identifiers of all the systemizations supported in the coding scheme version. This implements *CodingSchemeVersion.GetAllSystemizations* from the abstract model.

get_systemization

Returns a reference to the systemization object that implements the named systemization. An exception is thrown if the systemization name is unrecognized. This implements *CodingSchemeVersion.GetSystemization* from the abstract model.

get_default_systemization

Returns a reference to the "default" systemization for the coding scheme version. An exception is thrown if the terminology vendor has not specified a default. This implements *CodingSchemeVersion.GetDefaultSystemization* from the abstract model.

2.2.21 Systemization Interface

```
//*************************************************************
//        Systemization
//*************************************************************
interface Systemization  {

    readonly attribute SystemizationId systemization_id;
    readonly attribute CodingSchemeVersion coding_scheme_version;

    AssociationIdSeq get_association_ids();

    AssociationDef get_association_definition(
        in AssociationId association_id
    )raises (
        AssociationNotInSystemization
    );

    void list_all_association_instances(
        in unsigned long how_many,
        out AssociationInstanceSeq association_instance_seq,
        out AssociationInstanceIter association_instance_iter
    );

    Trinary are_entities_associated(
        in ConceptCode  source_code,
        in AssociatableElement  target_element,
        in AssociationId   association_id,
        in boolean  direct_only
```

```
) raises (
      AssociationNotInSystemization
);

Trinary could_association_be_inferred(
      in ConceptCode  source_code,
      in AssociatableElement  target_element,
      in AssociationId  association_id
) raises (
      AssociationNotInSystemization,
      NotImplemented
);

void list_associated_target_entities (
      in ConceptCode  source_code,
      in AssociationId  association_id,
      in boolean  direct_only,
      in unsigned long  how_many,
      out TargetElementSeqSeq  related_elements,
      out TargetElementSeqIter  related_elements_iter
) raises (
      AssociationNotInSystemization
);

void list_associated_source_codes (
      in AssociatableElement  target_element,
      in AssociationId  association_id,
      in boolean  direct_only,
      in unsigned long  how_many,
      out ConceptInfoSeq  concept_info_seq,
      out ConceptInfoIter  concept_info_iter
) raises (
      AssociationNotInSystemization
);
EntityGraph get_entity_graph (
      in AssociatableElement  root_node,
      in AssociationId  association_id,
      in AssociationRole  node_one_role,
      in boolean  direct_only
) raises (
      AssociationNotInSystemization,
      NotImplemented,
      TooManyToList
);
AssociationIdSeqget_associations_for_source (
      in ConceptCodesource_code
);
AssociationIdSeqget_associations_for_target (
      in AssociatableElement  target_element
);
```

```
ValidationResult validate_concept_expression (
    in ConceptExpression expression
) raises (
    InvalidExpression,
    NotImplemented,
    AssociationNotInSystemization
);

ConceptExpression get_simplest_form (
    in ConceptExpression expression
) raises (
    InvalidExpression,
    NotImplemented,
    AssociationNotInSystemization
);

ConceptExpression expand_concept (
    in ConceptCode concept,
    in AssociationQualifierIdSeq association_qualifier_seq
) raises (
    ConceptNotExpandable,
    UnknownCodingScheme,
    NotImplemented,
    AssociationNotInSystemization
);

Trinary are_expressions_equivalent (
    in ConceptExpression expression1,
    in ConceptExpression expression2
) raises (
    InvalidExpression,
    UnknownCodingScheme,
    AssociationNorInSystemization,
    NotImplemented,
    UnableToEvaluate
);
ConceptExpression expression_difference(
    in ConceptExpression expression1,
    in ConceptExpression expression2
) raises (
    InvalidExpression,
    UnknownCodingScheme,
    AssociationNotInSystemization,
    NotImplemented,
    UnableToEvaluate
);

ConceptExpression minimal_common_supertype (
    in ConceptExpressionSeq expressions
) raises (
    InvalidExpression,
```

```
            AssociationNotInSystemization,
            NotImplemented,
            NoCommonSupertype
        );

        ConceptExpression maximal_common_subtype (
            in ConceptExpressionSeq expressions
        ) raises (
            InvalidExpression,
            AssociationNotInSystemization,
            NotImplemented,
            NoCommonSubtype
        );
};
```

A systemization represents an ordering, classification and/or categorization of a set of concept codes. The purpose of a systemization is to further define and describe the concept codes within a coding scheme, as well as to define the relationship between these concept codes and other concept codes and/or characteristics in other coding schemes.

The systemization references one or more association types and contains a set of association instances between various concept codes and characteristics. Each of the individual systemization entities is described below.

systemization_id

The unique name of the systemization within the context of the coding scheme version.

coding_scheme_version

A reference to the coding scheme version object in which this systemization is implemented.This implements *GetCodingSchemeVersion* from the abstract model.

get_association_ids

Returns a list of all the association type identifiers that are referenced by this systemization. The ability to retrieve the ids without the overhead of retrieving the association definitions is provided as a performance enhancement for browser clients. This could be used in conjunction with the **get_association_definition** to implements the semantics of the *GetAllAssociations* method of the model.

get_association_definition

Returns an **AssociationDef** that includes the formal definition of an association as documented in the model. This includes the association identifier, source and target roles and cardinality, target types and flags that describe the semantics of the association.

list_all_association_instances

This operation provides iterator access to all of the association instances within the systemization. This implements the *GetAllAssociationInstances* method defined in the model.

are_entities_associated

This operation determines whether an association instance of the named association exists in which the source concept code is associated with the target. An exception is thrown if the association identifier is not defined in the systemization. The operation returns TRUE if the association exists, FALSE if the service can assert that it doesn't exist, and UNKNOWN if the service has insufficient information to say one way or the other.

The **direct_only** flag indicates whether only direct associations are to be considered (*TRUE*) or whether a transitive path between the source and target are also considered (*FALSE).* This flag is ignored in the case of non-transitive associations. Note that the **direct_only** flag applies only to the named association. Subtyping associations are not considered by the **are_entities_associated** operation. This implements the *AreEntitiesAssociated* method of the Systemization model.

could_association_be_inferred

This operation extends the **are_entities_associated** above to include subtyping and other associations where appropriate. The input parameters are identical to the previous operation, with the exception that the **direct_only** flag is presumed to be FALSE. A service implementation may use additional means at its disposal to determine whether there is some finite probability of an association existing.

list_associated_target_entities

This operation returns iterator access to the set of all target entities that participate in the named association with the source code. If *directOnly* is TRUE, only the target entities directly associated with the source codes are supplied. If FALSE, all of the target entities in the transitive closure of the association are returned.This implements the *GetAssociatedTargetEntities* method of the model.

list_associated_source_concepts

This operation returns iterated access to the set of all source concepts that participate in the named association with the target entity. If *directOnly* is TRUE, only the source concepts directly associated with the source codes are supplied. If FALSE, all of the target entities in the transitive closure of the association are returned. This implements the *GetAssociatedSourceCodes* method of the model. The *directOnly* flag is ignored and presumed to be TRUE when the supplied association is not transitive.

get_entity_graph

This operation returns a graph of instances of the supplied association rooted at the supplied root node and based on the supplied association type. If the **root_node_role** is **SOURCE**, the directed graph traverses from source to target. If the **root_node_role** is **TARGET**, the graph traverses from target to source. The returned

graph may either carry direct associations (**directOnly** = TRUE) or the transitive closure of the associations, (**directOnly** = FALSE). An exception is thrown if the association is unrecognized or the returned graph is too large to reasonably return in simple structure.

get_associations_for_source

This operation returns the set of all associations in which the supplied qualified code participates in the source role. This implements the *GetAssociationsForSource* method of the model.

get_association_for_target

This operation returns the set of all associations in which the supplied target element participates in the target role. This implements the *GetAssociationsForTarget* method of the model.

validate_concept_expression

A concept expression consists of a base concept code and one or more associations which apply to that code. The notation for the following example is borrowed from the GALEN CORE notation [13]. One representation of the upper lobe of the left lung could be:

Lobe which <*is-part-of (Lung* which *has-laterality* Left) *has-location* Upper>

The **validate_concept_expression** operation returns FALSE if the supplied concept expression is not semantically valid according to the coding scheme. If the return is TRUE, an optional association qualifier may also be returned to further qualify the conditions in which the TRUE return applies. As a hypothetical example, a systemization might return a qualifier of “sensible.” If the concept expression described the *middle* lobe of the left lung, the systemization might return a qualifier of “grammatical”, indicating that, while there *isn’t* a *middle* lobe of the left lung, the expression still made grammatical sense. This implements the *ValidateConceptExpression* method of the Systemization model.

get_simplest_form

This operation returns a concept expression that represents the simplest form in which the supplied concept expression may be expressed. Using the example above, a terminology system might have a concept code which represented the left lung. The result of a *get_simplest_form* call with the example above might yield:

Lobe which <*is-part-of* LeftLung *has-location* Upper>

This implements the *GetSimplestForm* method of the Systemization model.

expand_concept

This operation takes the supplied concept code and association qualifiers and returns the “canonical” concept expression that serves to define the concept. If *expand_concept* is supplied with the concept code <LeftLung> and no qualifiers in the above scenario, it might return:

Lung which *has-laterality Left*

This implements the *ExpandConcept* method in the Systemization model.

are_expressions_equivalent

This operation is supplied with two concept expressions. It determines whether these two expressions could be considered equivalent. This implements the *AreExpressionsEquivalent* method of the model.

expression_difference

This operation, given two concept expressions, determines the “difference” between the two concept expressions and returns this difference in the form of a third concept expression. This implements the *ExpressionDifference* method of the model.

minimal_common_supertype

This operation, given a sequence of two or more concept expressions, returns a concept expression that is the “closest” valid supertype based on the concepts in the expressions. An exception is thrown if there is no valid minimal common supertype short of the *universal type*. This implements the *MinimalCommonSupertype* method of the model.

maximal_common_subtype

This operation, given a sequence of two or more concept expressions, returns a concept expression that is the “closest” valid subtype based on the concepts in the expressions. An exception is thrown if there is no valid maximum common subtype short of the *absurd type.* This implements the *MaximalCommonSubtype* method of the model.

2.2.22 ValueDomainVersion Interface

```
//*****************************************
// Value Domain Version
//*****************************************

interface ValueDomainVersion {
    readonly attribute ValueDomainId value_domain_id;
    readonly attribute VersionId value_domain_version_id;
    readonly attribute boolean is_default_version;

    CodingSchemeIdSeq get_schemes_with_extensions();

    QualifiedCodeInfoSeq get_all_extensions();
```

```
    ConceptInfoSeq get_extension_for_scheme(
        in CodingSchemeId coding_scheme_id
    ) raises (
        UnknownCodingScheme
    );

    boolean is_code_in_domain(
        in QualifiedCode qualified_code
    );

    void get_pick_list(
        in UsageContextIdSeq context_ids,
        out PickListSeq pick_list,
        out PickListIter pick_list_iter
    ) raises (
        TooManyToList,
        NoPickListAvailable
    );

    void get_pick_list_for_scheme(
        in CodingSchemeId coding_scheme_id,
        in UsageContextIdSeq usage_context_ids,
        out PickListSeq pick_list,
        outPickListIter pick_list_iter
    ) raises(
        TooManyToList,
        UnknownCodingScheme,
        NoPickListAvailable
    );
};
```

The **ValueDomainVersion** interface represents a snapshot of a value domain at a point in time. Terminology services which provide the value domain interface may implement a set of value domain versions for a given value domain, or may simply choose to maintain only the latest version of a given value domain. In the latter case, the vendor is encouraged to change the version number every time the value domain is modified. The value domain version interface exposes the following attributes and operations:

value_domain_id

The globally unique name of the value domain represented as a qualified code. This implements the *GetParentValueDomain* method in the model.

value_domain_version_id

An identifier that uniquely identifies the version of the domain within the context of the domain itself. This corresponds to the *versionId* attribute in the model.

is_default_version

TRUE indicates that this version is the recommended version of the value domain to use at this point in time. This corresponds to the *isDefaultVersion* attribute in the model.

get_schemes_with_extensions

Returns a list of all coding schemes which have one or more concept codes listed in this value domain.

get_all_extensions

Returns a list of all concept codes that are included in this value domain. This list includes the qualified code and the preferred textual presentation for each code. This implements the *GetAllValueDomainExtensions* method in the model.

get_extension_for_scheme

Returns a list of all concept codes from a given coding scheme which are included in this value domain. This list includes the concept code and its preferred textual presentation.

An exception is thrown if the coding scheme is not recognized by the terminology service. This implements the *GetValueDomainExtension* method in the model.

is_code_in_domain

Returns TRUE if the qualified code is included in the value domain, FALSE otherwise. This implements the *IsCodeInValueDomain* method in the model.

get_pick_list

Returns the appropriate pick list given an ordered set of usage contexts. An exception is raised if no pick list is available or the pick list is considered too large to return. Unrecognized usage contexts are ignored. This implements the *GetPickList* method in the model.

get_pick_list_for_scheme

Returns the appropriate pick list consisting of concept codes from the supplied coding scheme. An exception is raised if no pick list is available, the coding scheme is not recognized, or the pick list is considered too large to return. Unrecognized usage contexts are ignored. This implements the *GetPickListForCodingScheme* method in the model.

2.3 Terminology Service Values Module

```
//File: TerminologyServiceValues.idl
//
#ifndef _TERMINOLOGY_SERVICE_VALUES_IDL_
#define _TERMINOLOGY_SERVICE_VALUES_IDL_
```

```
#pragma prefix "omg.org"
#include <orb.idl>
#include <NamingAuthority.idl>
#include "TerminologyServices.idl"

// ***************************************************************
//       module:  TerminologyServiceValues
// ***************************************************************

module TerminologyServiceValues {

typedef TerminologyServices::ConceptCode ConceptCode;
typedef NamingAuthority::QualifiedNameStr QualifiedNameStr;
typedef NamingAuthority::AuthorityIdStr AuthorityIdStr;

//*************************************************
//          ValueDomainId Strings
//*************************************************
typedef QualifiedNameStr ValueDomainIdStr;

const ValueDomainIdStr ASSOCIATION_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/AssociationId";
const ValueDomainIdStr ASSOCIATION_QUALIFIER_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/AssociationQualifierId";
const ValueDomainIdStr ASSOCIATION_BASE_TYPE_DOMAIN =
     "IDL:omg.org/TerminologyService/AssociationBaseTypeId";
const ValueDomainIdStr  LANGUAGE_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/LanguageId";
const ValueDomainIdStr LEXICAL_TYPE_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/LexicalTypeId";
const ValueDomainIdStr PRESENTATION_FORMAT_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/PresentationFormatId";
const ValueDomainIdStr SOURCE_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/SourceId";
const ValueDomainIdStr SOURCE_USAGE_DOMAIN =
     "IDL:omg.org/TerminologyService/SourceUsageId";
const ValueDomainIdStr  SYNTACTIC_TYPE_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/SyntacticTypeId";
const ValueDomainIdStr  USAGE_CONTEXT_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/UsageContextId";

//*********************************************************
//          AssociationId
//*********************************************************
typedef ConceptCode AssociationId;
const NamingAuthority::AuthorityIdStr
ASSOCIATION_ID_AUTHORITY_STRING =
          "IDL:org.omg/TerminologyService/Association/";

const AssociationIdIS_COMPOSED_OF =
```

```
    “isComposedOf”;
const AssociationIdHAS_SUBTYPES =
    “hasSubtypes”;
const AssociationIdREFERENCES =
    “references”;
const AssociationIdHAS_ATTRIBUTES =
    “hasAttributes”;

//*********************************************************
//          AssociationBaseTypeId
//*********************************************************
typedef ConceptCode AssociationBaseTypeId;
const NamingAuthority::AuthorityIdStr
ASSOCIATION_BASE_TYPE_ID_AUTHORITY_STRING =
        "IDL:org.omg/TerminologyService/AssociationBaseType/";

const AssociationIdWHOLE_PART =
        "wholepart";
const AssociationIdSUBTYPE =
        "subtype";
const AssociationIdREFERENCE =
        "reference";
const AssociationIdNON_SEMANTIC =
        "nonSemantic";



//*****************************************************
//        AssociationQualifierId Strings
//*****************************************************
typedef QualifiedNameStr AssociationQualifierIdStr;

const AssociationQualifierIdStr MANDATORY =
        "IDL:omg.org/TerminologyService/AssociationQualifier/MAND";
const AssociationQualifierIdStr OPTIONAL =
        "IDL:omg.org/TerminologyService/AssociationQualifier/OPT";
const AssociationQualifierIdStr SINGLE =
        "IDL:omg.org/TerminologyService/AssociationQualifier/SING";
const AssociationQualifierIdStr PLURAL =
        "IDL:omg.org/TerminologyService/AssociationQualifier/PLUR";

//*****************************************************
//        LanguageIds
//*****************************************************
typedef ConceptCode LanguageId;

const NamingAuthority::AuthorityIdStr
LANGUAGE_ID_AUTHORITY_STRING =
        "DNS:usmarc.omg.org/041/";

const LanguageId DANISH ="DAN";
```

```
const LanguageId ENGLISH ="ENG";
const LanguageId FRENCH ="FRE";
const LanguageId GERMAN ="GER";
const LanguageId ITALIAN ="ITA";
const LanguageId SPANISH ="SPA";

//*****************************************************
//        LexicalTypeIds
//*****************************************************

typedef QualifiedNameStr LexicalTypeIdStr;

const LexicalTypeIdStr ABBREVIATION = “DNS:umls.hl7.omg.org/LT/ABB”;
const LexicalTypeIdStr EMBEDDED_ABBREVIATION =
“DNS:umls.hl7.omg.org/LT/ABX”;
const LexicalTypeIdStr ACRONYM = “DNS:umls.hl7.omg.org/LT/ACR”;
const LexicalTypeIdStr EMBEDDED_ACRONYM =
“DNS:umls.hl7.omg.org/LT/ACX”;
const LexicalTypeIdStr EPONYM = “DNS:umls.hl7.omg.org/LT/EPO”;
const LexicalTypeIdStr LAB_NUMBER = “DNS:umls.hl7.omg.org/LT/LAB”;
const LexicalTypeIdStr PROPER_NAME = “DNS:umls.hl7.omg.org/LT/NAM”;
const LexicalTypeIdStr SPECIAL_TAG = “DNS:umls.hl7.omg.org/LT/NON
NO”;
const LexicalTypeIdStr TRADE_NAME = “DNS:umls.hl7.omg.org/LT/TRD”;

//*****************************************************
//        PresentationFormatIds
//*****************************************************
typedef ConceptCode PresentationFormatId;

const NamingAuthority::AuthorityIdStr
PRESENTATION_FORMAT_AUTHORITY_STRING =
"DNS:omg.org/MIME/";

const PresentationFormatId PLAIN_TEXT =  "text/plain";
const PresentationFormatId RTF =  "application/rtf";
const PresentationFormatId ZIP =  "application/zip";
const PresentationFormatId PDF =  "application/pdf";
const PresentationFormatId GIF_IMAGE =  "image/gif";
const PresentationFormatId BASIC_AUDIO =  "audio/basic";

//*****************************************************
//       SourceIds
//*****************************************************

typedef QualifiedNameStr SourceIdStr;

//*****************************************************
//       SourceUsageTypeId
//*****************************************************
```

```
typedef QualifiedNameStr SourceUsageTypeIdStr;

//*****************************************************
//        SyntacticType
//*****************************************************

typedef ConceptCode SyntacticTypeId;

const NamingAuthority::AuthorityIdStr
SYNTACTIC_TYPE_AUTHORITY_STRING =
          "DNS:umls.hl7.omg.org/STT";
const SyntacticTypeId CASE_DIFFERENCE =  "C";
const SyntacticTypeId WORD_ORDER =  "W";
const SyntacticTypeId SINGULAR_FORM =  "S";
const SyntacticTypeId PLURAL_FORM =  "P";

//*****************************************************
//        Query Property Types
//*****************************************************

typedef string TerminologyServiceProperty;

const TerminologyServiceProperty LexicalTypeProperty = “LexicaltypeId”;
const TerminologyServiceProperty AssociationProperty = “AssociationId”;
const TerminologyServiceProperty PreferredTextProperty = “Preferred-
Text”;
const TerminologyServiceProperty DefinitionProperty = “Definition”;
const TerminologyServiceProperty PresentationProperty = “Presenta-
tionId”;

};

#endif /* _TERMINOLOGY_SERVICE_VALUED_IDL_ */
```

The above module provides the literal codes that can be used to access the terminology services. The first section contains string literals which name each value domain included in the terminology services interface specification. With the exception of the version suffix, these literals should match the interface repository (IR) identifier of each of the named entities exactly. As an example, the IDL code:

```
#pragma prefix "omg.org"
#include <orb.idl>
#include <NamingAuthority.idl>

// ****************************************************************
//       module:  TerminologyService
// ****************************************************************
module TerminologyServices {
    ...
TYPEDEF cONCEPTcODE lANGUAGEiD;
...
```

```
];

    // ...
};
```

should produce an entry in the interface repository in the form:

"IDL:omg.org/TerminologyService/LanguageId";

The literals described in the value domains section are not used directly in the terminology services, but would be the value domains that would be used when referencing the domains in a “terminology of terminology.”

The value domains section is followed by string literals for all of the pre-defined codes that are used in the terminology services itself. **ConceptCode** type literals may be used directly to access the appropriate methods. Literals of type **QualifiedNameStr** must first be converted into a qualified code before being used in the interface. This conversion may be accomplished by using the **TranslationLibrary** interface.

Terminology 3

Contents

This chapter contains the following topics.

3.1 Trader Service

The following definitions are Service Types defined for **TerminologyServices** components for use by the OMG Trader Service.

```
interface TerminologyService {
    interface TerminologyService;
    mandatory readonly property string terninology_service_name;
    mandatory readonly property StringSeq interfaces_implemented;
    mandatory readonly property StringSeq conformance_classes;
    mandatory readonly property StringSeq supported_coding_schemes;
    mandatory readonly property StringSeq default_coding_scheme;
    mandatory readonly property StringSeq supported_languages
};
```

Since all TerminologyServices implement the **TerminologyService** interface, only one Trader Service type is needed, which is also called 'TerminologyService.' The **TerminologyService** interface has attributes for the common characteristics for all TerminologyServices. These are used as properties for the service type. Additional properties are specified which are attributes of **CodingSchemeVersions** (a derived interface). The stringified versions of the attributes are used for properties since the standard Trader constraint language does not provide a way to filter on user-defined types.

The interface type returned from the Trader Service for this service type is a **TerminologyService**. All properties are mandatory. These are common to all implementations.

terminology_service_name

The **terminology_service_name** property contains the information from the **terminology_service_name** attribute of the **TerminologyService** interface. It is formatted as specified for **NamingAuthority::AuthorityIdStr**.

interfaces_implemented

This sequence contains the names of the interfaces the component has references to. The names are fully qualified names which include the module name.

conformance_classes

This sequence contains the conformance classes the implementation supports. The strings are identical to the way they are spelled and capitalized in the definition of the conformance classes for TerminologyServices.

supported_coding_schemes

This sequence contains the Coding Schemes which are supported by this terminology service. This is a sequence of all the stringified **CodingSchemeId** in this terminologyService. It is formatted as a **NamingAuthority::AuthorityIdStr**.

supported_languages

This sequence contains the supported languages for the **TerminologyService**. It is formatted as shown in the **TerminologyServiceValues** module.

default_coding_scheme

This is the default **CodingSchemeId** for this **TerminologyService**. It is formatted as a **NamingAuthority::AuthorityIdStr**.

3.2 Meta-Terminology

3.2.1 Association

Most terminology services deal with complex associations between coded concepts and, optionally, coded concepts and non-coded information. The terminology used to describe these associations varies from vendor to vendor. One vendor may use the term “relationship,” another the term “facet,” and a third the term “attribute.”

Each terminology system typically has its own set of codes for individual associations. One system may use the code “hasMember” to indicate subtyping, while another may use “isA" and a third may use “hasSubtypes.” To further complicate matters, one system may use the code “isA" to represent subtyping while a second system uses “isA" to represent a type of whole-part association. Regardless of the terminology used it is very important that the user of the service be able to determine the underlying intent behind a given association (relationship, facet, attribute) code.

This section defines some of the basic distinguishing characteristics of different types of associations. These associations are arranged within an arbitrary, pragmatic taxonomy. Each node in the taxonomy has an identifier that serves a code for the specific combination of characteristics. The set of codes then provides a base set of types which serve to distinguish the characteristics of vendor-specific associations.

As an example, a terminology vendor may provide an association named “broader than.” While one can infer possible intent from the association name itself, a generic client would not be able to utilize this association without further information. If, however, the generic client were able to determine that “broader than” was a type of subtyping association (or whole-part or whatever else), it could make certain assumptions about the characteristics of the “broader than” which would allow it to perform useful operations.

Please note that the taxonomy described below is intended solely as pragmatic ordering based on useful characteristics. This is *not* taxonomy of meaning, and one cannot presume that any of the characteristics are inherited through a classification of association based on terminological *meaning.*

As an example, take the classification:

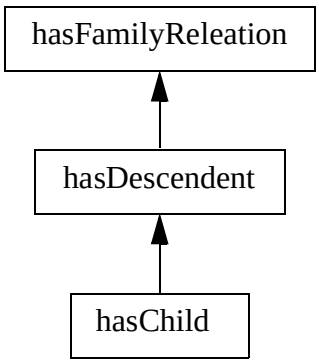


Figure 3-1 Sample Classification by Meaning

In Figure 3-1, the association *hasChild* is a type of *hasDescendent*, which in turn is a type of *hasFamilyRelation*. The *hasChild* association is intransitive and anti-symmetric, while the *hasDescendent* is *transitive* and anti-symmetric and the *hasFamilyRelation* is both transitive and symmetric.

The discussions in the sections that follow are based heavily on Kilov and Ross, *Information Modeling, An Object-Oriented Approach* [14] with additional guidance and input from Dr. Alan Rector of the University of Manchester.

3.2.1.1 Association Characteristics

Formally, an association can be defined as a binary, asymmetric relation defined across the cross-product of a set of types. An association type – a relation type - relates a source type to a target type. [14] A target type may either be a single element or a set of elements. An *instance* of a relation exists if and only if there exists an instance of its source type and an instance of its appropriate target type. (There is an exception to this rule, the Composition-Package type, but this type is not used in this specification). The following diagram represents the set of characteristics used to distinguish associations within this specification:

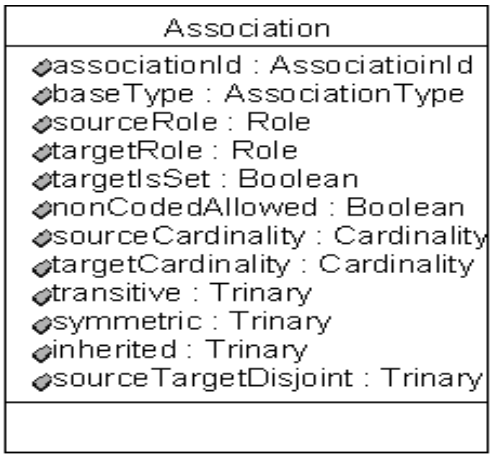


Figure 3-2 Association Characteristics

associationId

A unique identifier for the particular association.

baseType

An identifier for the general class of association. This identifier must be one reference one of the general classes described within this section.

sourceRole, targetRole

A synonym for the source and target portion of the association respectively. These synonyms are only unique within the particular association. As an example, the **WholePart** association has a **sourceRole** of “whole” and a **targetRole** of “parts,” which serve as synonyms for the source and target of the association respectively.

targetIsSet

For a given association type, the target is either a single type or a set of different types. As an example, the **Reference** association associates a source concept code with a single target concept code, while the **WholePart** association associates a source concept code with a *set* of target concept codes.

nonCodedAllowed

A FALSE value indicates that the target must either be composed of a single concept code or a set of concept codes, depending upon the **targetIsSet** setting. A TRUE value indicates that the target may be *either* a concept code or a string (Characteristic) representing some non-coded attribute value.

sourceCardinality

The minimum and maximum number of instances of an association in which an instance of an entity represented by the source may/must participate.

A minimum **sourceCardinality** of 0 indicates that an instance of the entity represented by the source code may exist without being associated with an instance of the corresponding target. An example of a source cardinality of 0 would be an optional reference association, where an instance of the entity represented by the source code may or may not reference an instance of the entity represented by the target code.

A minimum **sourceCardinality** of 1 or more indicates that an instance of the entity represented by the source code *always* co-occurs with an instance of the entity represented by the target. An example of a minimum **sourceCardinality** of 1 or more is exhaustive subtyping—subtyping in which *every* instance of the super type (source) is always associated with at least one instance of one of the associated target types.

A maximum **sourceCardinality** of 1 indicates that an instance of the entity represented by the source code may be associated with at most one target entity, while > 1 indicates that it may be associated with several different instances of the target entity.

targetCardinality

The minimum and maximum number of instances of association of a given type in which an instance of the entity represented by the target may/must participate.

A minimum **targetCardinality** of 0 indicates that an instance of the entity represented by the target code may exist without being associated with an instance of the corresponding source code. A Reference association is an example of this sort of association, where a target entity may or may not be referenced by an instance of the source.

A minimum **targetCardinality** of 1 or more indicates that an instance of the target entity *must* be associated with at least one instance of the source entity.

Note – The source and target cardinality does NOT make assertions about databases or other representation mediums. They assert *facts* about “real life” associations. The fact, for instance, that every person has a parent is a fact independent of whether or not tokens representing persons in a database are or are not associated with each other.

---

transitive

TRUE indicates that the given association is transitive. If the associations of “A rel B” and “B rel C” exist, the association “A rel C” may also be inferred to exist. No statement is made in this specification about whether “A rel C” is represented explicitly or solely by implication within a vendor implementation. An example transitive association may be an “is contained in” type of association.

UNKNOWN indicates that one may not infer “A rel C” from the first two associations. One may not infer “NOT (A rel C)” either.

FALSE indicates that the given association is intransitive, and the associations “A rel B” and “B rel C” may be used to infer “NOT (A rel C).”

symmetric

TRUE indicates that the association is symmetric, and the existence of the symmetric association “A rel B” implies that the association “B rel A” exists as well. UNKNOWN indicates that the association is not symmetric, and the existence of the association “A rel B” may neither be used to infer “B rel A” or “NOT(B rel A).” FALSE indicates that the association is anti-symmetric and one may infer “NOT(B rel A)” from “A rel B.”

inherited

If this is TRUE, any subtypes of the source type and/or target type(s) involved in this association inherit their participation in the association. If FALSE, only entities of the specified type participate in the association while the subtypes do not. UNKNOWN indicates that inheritance has not been determined one way or the other.

sourceTargetDisjoint

In most associations, for any association instance (and its transitive closure), the set of instances of its source and elementary target types have an empty intersection. As an example, in the **Composition** association, an entity may not be directly or indirectly composed of itself. The **Subtyping** association, however, requires that any instance of the target type also be an instance of the associated source. TRUE indicates that the intersection of the set of instances of the source and elementary target types are disjoint. FALSE indicates that they are NOT disjoint and UNKNOWN states that they may or may not be.

Note that this usually applies to instances, not to categories. It is perfectly reasonable to say that, for example, body parts are composed of body parts, recursively. In fact recursive part-whole relations are the norm rather than the exception. It is true that no one body part can be a member of itself, but at the level of the attribute definitions, this does prevent us, for example, from formulating the concept of “A building which contains (another) building.”

3.2.1.2 Specific Association Types

The sections below describe the characteristics of several common association types, as identified by a collection of characteristics. Please note that while the names given to each type are intended to be descriptive of their characteristic type use, one should not infer anything more into the name than what is contained in the description. Each name is an arbitrary code for a specific collection of characteristics and implies no more than that.

Reference Association

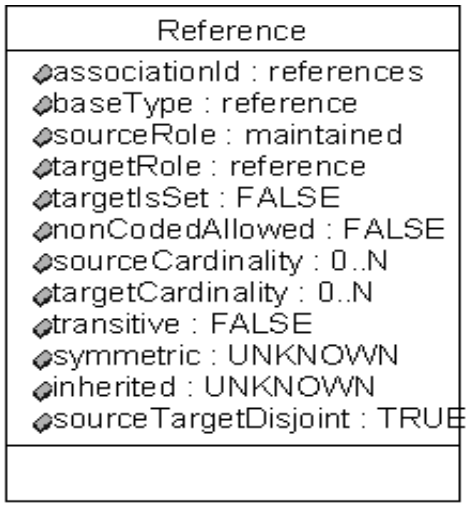


Figure 3-3 Reference Association

Any association that has a **baseType** of “reference” must have the following characteristics, and any association which has the following characteristics must have a **baseType** of “reference:”

- **targetIsSet** = **FALSE** - A parent concept code is associated with a single target concept code, not a set of codes.
- **nonCodedAllowed** = **FALSE** - The target must be a concept code, not a characteristic.
- **transitive** = **UNKNOWN** - A reference association is not transitive. Just because A references B and B references C, it may not be inferred that A references C.
- **sourceTargetDisjoint** = **TRUE** - An instance of a source code may not directly or indirectly imply the existence of itself.

Note – An additional (and important) invariant on a reference association is that some of the properties of the maintained instance are determined by the properties of its reference instance in this reference association.

An association that has a **baseType** of “reference” may vary the following characteristics:

- **sourceRole, targetRole** - These roles may be renamed to be more applicable if desired.
- **sourceCardinality** - The minimum cardinality of the source may be greater than or equal to zero. If greater than zero, one is asserting all instances of the source code must be associated with a (perhaps implicit) set of target instances. The

maximum cardinality may also be constrained to a finite number. As an example, the source cardinality may be set to 1..1, indicating that every source instance(maintained) must reference exactly one target (referenced) instance.

- **targetCardinality** - The minimum and maximum target cardinality may be varied, requiring and/or restricting the number of references from each source.
- **symmetric** - This may be TRUE, UNKNOWN, or FALSE.
- **inherited** - This may be TRUE, UNKNOWN, or FALSE.

Subtyping Association

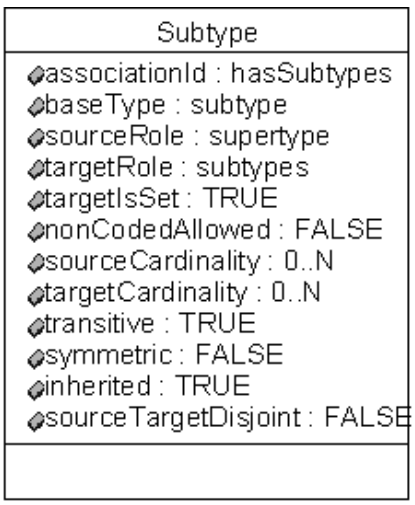


Figure 3-4 Subtyping Association

Any association which has a **baseType** of “subtype” must have the following characteristics, and any association which has the following characteristics must have a **baseType** of “subtype”:

- **targetIsSet** = **TRUE** - A parent concept code is associated with a set of one or more target concept codes.
- **nonCodedAllowed** = **FALSE** - The target must be a concept code, not a characteristic
- **transitive** = **TRUE** - The association is transitive. If A has Subtypes {B, C} and B has subtypes {D, E} then A has Subtypes {D, E, C}.
- **symmetric** = **FALSE** - If A is a subtype of B, B cannot be a subtype of A.
- **inherited** = **TRUE** - Subtypes of both the source or elementary target instances inherit the Subtyping association.
- **sourceTargetDisjoint** = **FALSE** - An instance of a subtype which participates in a Subtyping association must also be an instance of the associated supertype. The two sets are not disjoint. The instance of the subtype must have all of the inherited associations and properties of its supertype.

An association which has a **baseType** of “subtype” may vary the following characteristics:

- **sourceRole, targetRole** - These roles may be renamed to be more applicable if desired.
- **sourceCardinality** - The minimum cardinality of the source may be greater than or equal to 0. If the minimum cardinality is set to 1, one is asserting that this is an *exhaustive subtype* – that every instance of the supertype has a corresponding instance of the subtype. The maximum cardinality may be constrained to a finite integer to limit the number of possible ways that a source may be decomposed using the given association**.**
- **targetCardinality** - Setting the minimum target cardinality to “1” identifies a *static subtype*—a subtype association that is inherent in the target. Static subtyping is the classic decompositional subtyping where all instances of a given target type are also instances of the given source type. An example of a static subtype might be the assertion that <bodyOrgan> hasStaticSubtypes <lung, liver, heart>. Any instance of the entity <lung> is also an instance of a body organ.

  If the minimum target cardinality is “0”, it is possible for an instance of a target type to exist *without being a subtype of the source type*. This is often referred to as “dynamic” or role-based subtyping. An example of role-based subtyping might be that of <person>, <company>, and <customer>. One could assert that <customer> hasDynamicSubtypes {<person>, <company>}. An instance of <person> might or might not be a subtype of <customer> depending upon the circumstances.

  The maximum target cardinality may also be constrained to a finite number. Setting the maximum target cardinality to one would restrict multiple inheritence using the particular subtyping association.

Non-Semantic Association

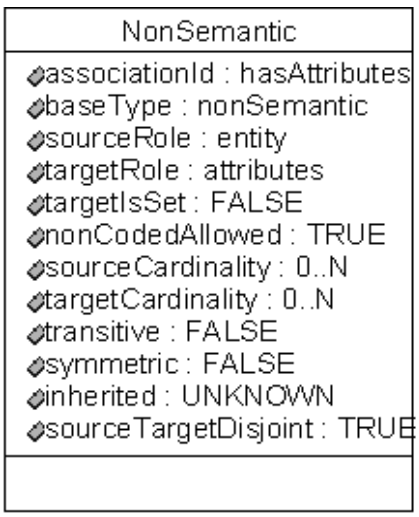


Figure 3-5 Non-Semantic Association

Any association which has a **baseType** of “nonSemantic” must have the following characteristics, and any association which has the following characteristics must have a **baseType** of “nonSemantic”:

- **targetIsSet = FALSE** - A parent concept code is associated with exactly one target entity.
- **nonCodedAllowed = TRUE** - The target may be either a concept code or a string representing a non-coded characteristic.
- **transitive = FALSE** - The range (coded concepts and non-coded strings) is not the same as the domain (coded concepts), this association is not transitive.
- **symmetric = FALSE** - The range (coded concepts and non-coded strings) is not the same as the domain (coded concepts), this association is not transitive.
- **sourceTargetDisjoint = TRUE.** An instance of a source code may not directly have a non-semantic association with itself.

An association which has a baseType of “nonSemantic” may vary in the following characteristics:

- **sourceRole, targetRole** - These roles may be renamed to be more applicable if desired.
- **sourceCardinality** - The minimum and maximum number of associations that an instance of a given source code may participate may vary. Setting the minimum source cardinality to a positive number indicates that all instances of the type indicated by the source concept code must have the particular property or characteristic indicated by the association.
- **targetCardinality** - The minimum and maximum number of associations that an instance of a given target entity may/must participate in. Non-coded targets are considered immutable and always participate in exactly one association. If the non-coded target “2.0” participates in two different associations, it is viewed as two separate instances.
- **inherited** - Non-semantic associations may either be inherited or not.

3.2.2 Vendor-Defined Associations

The association classes described above may be used directly in a terminology service. A terminology vendor typically has a pre-defined set of associations supplied with the terminology. To be generally useful, the terminology vendor is encouraged to provide the set of characteristics described above for each individual association type. In addition, if the set of characteristics for an association match any of the association types described above, the identifier of the matching type (wholepart, reference, subtype, nonSemantic) should be included in the **baseType** attribute of the association itself.

Example: The terminology vendor has an association called “isA”, which supplies a non-exclusive, non-disjunctive subtyping type of association. It is determined to have the following characteristics:

targetIsSet : TRUE
nonCodedAllowed:FALSE
sourceCardinality:0..N
targetCardinality:0..N
transitive:TRUE
symmetric:FALSE
inherited:TRUE
sourceTargetDisjoint:FALSE

Note – Used grammatically, the term “isA” is preceded by a single target entity and succeeded by the source (automobile isA vehicle). This is one of the reasons that the source and target roles are very important, as they serve to clarify the direction implied by the association name. (subtype isA supertype – where subtype is a member of a target set and supertype is the source concept)

Because this matches the characteristics of the subtyping association (**targetIsSet**, **nonCodedAllowed**, **transitive**, **symmetric**, **inherited**, **sourceTargetDisjoint**), this would have a base type of “subtype”. The Association class for “isA” follows:

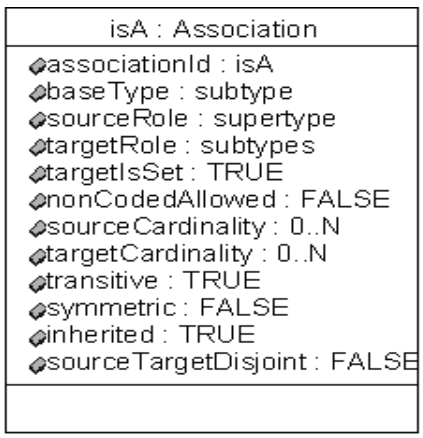


Figure 3-6 Sample Vendor Association

The set of characteristics shown in Figure 3-6 would be returned in the **AssociationDef** structure from the Systemization.**get_association_definition** method.

Associations may be less precisely defined in some terminology systems. As a worse case, a terminology vendor must supply the association identifier, a source role (which could be “source”), a target role (which could be “target”) whether the target is a set or element and whether non-coded targets are allowed. As an example suppose a terminology has an association called “is associated with.” It would not have a **baseType** as it would not match any of the types above. The description shown in Figure 3-7 might be returned for this generic association:

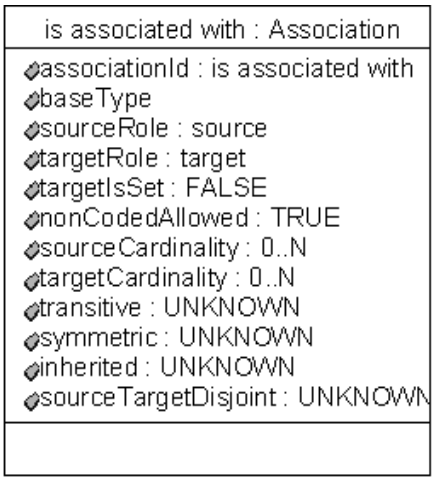


Figure 3-7 Sample Generic Association

3.2.2.1 *Predefined Association Codes*

Table 3-1 contains the association codes described in the preceding sections. These association codes may be used directly in an implementation. These codes *must* return the AssociationDef structures which correspond to their descriptions in the above sections. (i.e., “References” must return **targetIsSet : FALSE**, **nonCodedAllowed : FALSE**, **inherited : UNKNOWN**, etc.). Any change in the characteristics (e.g., **inherited : TRUE** for References) results in a new association which must be given a new, unique code.

Table 3-1 Association Codes

Characteristic Class	Code	QualifiedName
WholePart	isComposedOf	IDL:org.omg/TerminologyService/Association //isComposedOf
Reference	references	IDL:org.omg/TerminologyService/Association / references
Subtype	hasSubtypes	IDL:org.omg/TerminologyService/Association /hasSubtypes
NonSemantic	hasAttributes	IDL:org.omg/TerminologyService/Association /hasAttributes

3.3 Association Qualifier

Table 3-2 shows the four association qualifiers that are predefined in this specification which may be used to identify optionality and plurality qualifiers. All other association qualifiers are to be established by the terminology vendor. If not supplied, the default is that the association is optional and plural.

Table 3-2 Association Qualifier Codes

Qualifier	Meaning	Code	Qualified Code String
Optional	Association between source and target element is optional	OPT	IDL:omg.org/TerminologyService/AssociationQualifier/OPT
Mandatory	Association between source and target element is mandatory	MAND	IDL:omg.org/TerminologyService/AssociationQualifier/MAND
Single	At most one target element may be associated with the source	SING	IDL:omg.org/TerminologyService/AssociationQualifier/SING
Plural	More than one target element may be associated with the source	PLUR	IDL:omg.org/TerminologyService/AssociationQualifier/PLUR

3.4 CharacterSet

The implementation of this specification always assumes that a coding scheme version will have a specific language associated with it. In the CORBA specification, character set negotiation occurs between ORBs and is outside of the control of the application. For the time being we will presume that the language identifier determines an appropriate character set (or sets) and the negotiation is all covered by the ORBs themselves. As a consequence we won't explicitly expose character set codes in this document.

3.5 Coding Scheme

When this document was produced, we had high hopes that the ISO/IEC Standard on the Registration of Coding Schemes, [16] and [17], would serve the purpose of a central registry for the names of coding schemes. Unfortunately, this standard has not been heavily used and few coding schemes have been registered with this body to date.

It is critical to the success of this specification that applications be able to access a coding scheme by name without knowing the terminology services supplier in advance. To this end, we are proposing the following "stop-gap" measure which will be used until a more permanent solution comes into play.

1. The OMG DNS be used as the "registration authority" for the various coding schemes below, and that they then be subdivided according to their primary owner. (**"DNS:omg.org"**).

2. ASTM Committee E31 on Computerized Systems, [18], and HL7, [19], jointly maintain registry of medical coding schemes which are used in the HL7 chapters 4 and 7 (Orders and Results). Section 7.1.4, Coding schemes, in the HL7 provides a fairly extensive list of coding scheme codes. We propose that the set of coding scheme codes in Figures 7-2 and 7-3 in the HL7 manual be used as the primary designation when available. Local coding schemes such as “99zzz,” “L,” and “LB” should be coded using the **RegistrationAuthority** “Other” or the DNS of the owning facility, rather than using the HL7 reference. As an example, the ICD-9-CM coding scheme, [20], would be represented as:

DNS:hl7.omg.org/I9C”

Internal HL7 tables would be coded in the form “HL7xxx.”

“DNS:hl7.omg.org/HL7001”

As mentioned above, a local table would be represented in the form:

“OTHER:myprivatedomain/99173”

or, if possible:

“DNS:mycompany.com/99173”

3. Coding schemes that are not in the HL7 specification should be prefixed by the common name of the appropriate authority. It may be necessary to periodically publish the names of these other authorities until a more permanent scheme is arrived at. One that is used elsewhere in the document is the USMARC (U. S. Machine Readable Cataloging) codes which will be represented as:

“DNS:usmarc.omg.org/xxx”

where “xxx” represents the appropriate MARC Tag. The coding scheme for language would then be:

“DNS:usmarc.omg.org/041”

It is anticipated that this solution is temporary and that we can move to a central registry of coding schemes as soon as one becomes generally available.

3.6 Language

It is recommended that the set of language codes be supplied by the USMARC (Machine-Readable Cataloging) system. We will need to create the appropriate coding scheme identifier for USMARC. The codes themselves will all be represented in upper case. A list of current codes may be found at:

gopher://marvel.loc.gov:70/00/.listarch/usmarc/language

Table 3-3 contains the frequently used language codes along with the string form of their qualified name.

Table 3-3 Language Codes

Language	Code	Qualified Code String
Danish	DAN	DNS:usmarc.omg.org/041/DAN
English	ENG	DNS:usmarc.omg.org/041/ENG
French	FRE	DNS:usmarc.omg.org/041/FRE
German	GER	DNS:usmarc.omg.org/041/GER
Italian	ITA	DNS:usmarc.omg.org/041/ITA
Spanish	SPA	DNS:usmarc.omg.org/041/SPA

3.7 LexicalType

Table 3-4,The Unified Medical Language System (UMLS) Lexical Tag table will be used as the default lexical type. The types as extracted from the 1997 edition follow, although the most current edition should be referenced for the definitive set.

Table 3-4 Lexical Type Codes

Type	Code	Qualified Code String
Abbreviation	ABB	DNS:umls.hl7.omg.org/LT/ABB
Embedded abbreviation	ABX	DNS:umls.hl7.omg.org/LT/ABX
Acronym	ACR	DNS:umls.hl7.omg.org/LT/ACR
Embedded acronym	ACX	DNS:umls.hl7.omg.org/LT/ACX
Eponym	EPO	DNS:umls.hl7.omg.org/LT/EPO
Lab number	LAB	DNS:umls.hl7.omg.org/LT/LAB
Proper name	NAM	DNS:umls.hl7.omg.org/LT/NAM
Special tag	NON NO	DNS:umls.hl7.omg.org/LT/NON NO
Trade name	TRD	DNS:umls.hl7.omg.org/LT/TRD

3.8 PresentationFormat

The MIME [20], [21] format was selected to identify the format of a presentation. Table 3-5 presents some of the more common formats.

Table 3-5 MIME Codes

Type	Code	Qualified Code String
Plain text	text/plain	DNS:omg.org/MIME/text/plain
Rtf	application/rtf	DNS: omg.org /MIME/application/rtf
zip	application/zip	DNS: omg.org /MIME/application/zip

Table 3-5 MIME Codes

Type	Code	Qualified Code String
pdf	application/pdf	DNS: omg.org/MIME/application/pdf
Gif image	image/gif	DNS: omg.org /MIME/ image/gif
Audia	audio/basic	DNS: omg.org /MIME/audio/basic

MIME formats are rich enough to allow exotic formats, Internet links, etc.

3.9 Source

A source is a code of any form of bibliographic reference. At the time this document was published a definitive coding scheme which could be used for sources had not been located.

3.10 Source Term Type

The source term type is a code that identifies how a given presentation is used in the relevant source vocabulary. The best (and probably only) list of source term types may be found under the heading of *B.4 Concept Name Types* in the *1997 UMLS reference manual*. [5] Some samples are included in Table 3-6, although the user is referred to the source for an exhaustive set of codes.

Table 3-6 Sample Source Term Type Codes

Type	Code	Qualified Code String
Attribute type abbreviation	AA	DNS:umls.hl7.omg.org/TTY/AA
Abbreviation in any source vocabulary	AB	DNS:umls.hl7.omg.org/TTY/AB
Adjective	AD	DNS:umls.hl7.omg.org/TTY/AD

3.11 Syntactic Type

The UMLS is used as the reference for the base set of syntactic types. Table 3-7 lists the set of syntactic types in the UMLS. Note that the “Other” variant is not listed, as it is a code that could potentially change meaning as other syntactic types are added. If the syntactic type is not available, the terminology service should simply not supply it.

Table 3-7 Syntactic Type Codes

SyntacticType	Code	Qualified Code String
Varies from preferred presentation only in upper-lower case	C	DNS:umls.hl7.omg.org /STT/C

Table 3-7 Syntactic Type Codes

SyntacticType	Code	Qualified Code String
Contains same words as preferred form, disregarding order and punctuation	W	DNS:umls.hl7.omg.org /STT/W
Singular of the preferred form	S	DNS:umls.hl7.omg.org /STT/S
Plural of the preferred form	P	DNS:umls.hl7.omg.org /STT/P

3.12 Usage Context

This specification does not specify any usage context codes.

3.13 Value Domain

This specification defines a set of value domains that are used by the specification itself. This set of domains is defined in the "meta" coding scheme, identified by the **MetaSchemeId** option of the **schemeIdSource** in the **SchemeId**. The codes in Table 3-8 identify value domains within the meta schema:

Table 3-8 Value Domain Codes

Value Domain	Value Domain ID
Language	IDL:org.omg/TerminologyService/Language
Lexical Type	IDL:org.omg/TerminologyService/LexicalType
Presentation Format	IDL:org.omg/TerminologyService/PresentationFormat
Relation	IDL:org.omg/TerminologyService/Relation
Relationship Qualifier	IDL:org.omg/TerminologyService/RelationshipQualifier
Source	IDL:org.omg/TerminologyService/Source
Source Term Usage	IDL:org.omg/TerminologyService/SourceTermUsage
Syntactic Type	IDL:org.omg/TerminologyService/SyntacticType
Usage Context	IDL:org.omg/TerminologyService/UsageConctext

3.14 Conformance Points

This section describes the various conformance levels possible for an LQS-compliant terminology service provider.

3.14.1 Minimum Implementation

The minimum implementation which may still be deemed "LQS-compliant" must include two interfaces:

- **TerminologyService**
- **LexExplorer**

The **TerminologyService** interface may return a NULL object reference for either or both the **CodingSchemeLocator** and **ValueDomainLocator** attributes. It must return a valid reference for the **LexExplorer** attribute. The interface must support at least one coding scheme, meaning that it may not return a zero length list from the **get_coding_scheme_ids** operation. It may either throw an exception or implement the **get_native_coding_scheme_info**.

Each interface of the **LexExplorer** is described in the next table.

get_preferred_text	Must be implemented for at least one qualified code. The **context_ids** may be ignored.
get_preferred_text_ for_concepts	Must be implemented for at least one qualified code. The **context_ids** may be ignored.
get_preferred_definition	May return an empty structure or a definition.
translate_code	May throw TranslationNotAvailable in all cases.
translate_codes	May return a sequence of NULL pointers.
list_concepts	Must return a valid concept iterator if there is less than 1000 concepts in the scheme; otherwise, it may throw TooManyToList if it so chooses.
list_value_domain_ids	May return an empty iterator in all cases.
is_concept_in_value_domain	May throw UnknownValueDomain in all cases.
are_concepts_in_value_domains	May return a sequence of UNKNOWN of the same size as the number of passed codes. Does not have to validate parameter alignment if not implemented.
get_pick_list	May throw NoPickListAvailable in all cases.
association_exists	May throw NoSystemizationForCodingScheme in all cases.
associations_exist	May return a sequence of UNKNOWN of the same size as the number of passed codes. Does not have to validate parameter alignment if not implemented.
list_associated_target_elements	May throw NoSystemizationForCodingScheme in all cases.
list_associated_source_codes	May throw NoSystemizationForCodingScheme in all cases.

3.14.2 Additional Conformance Levels

3.14.2.1 CodingSchemeLocator Conformance

If the coding scheme locator interface is supported, all of the properties and methods in the interface must be implemented. The return structure, **CodingSchemeVersionRefs**, must have *at least one* of the

CodingSchemeVersionAccess, **PresentationAccess**, **LinguisticGroupAccess**, **AdvancedQueryAccess**, and **SystemizationAccess** interfaces implemented.

3.14.2.2 *CodingSchemeVersion Conformance*

If implemented, the **CodingSchemeVersion** interface must support all of the specified methods with the exception of **get_comments**, **get_instructions**, **match_concepts_by_string**, and **match_concepts_by_keyword**.

3.14.2.3 *PresentationAccess Conformance*

If present, the **PresentationAccess** interface must be implemented completely.

3.14.2.4 *LinguisticGroupAccess Conformance*

If present, the **LinguisticGroupAccess** interface must be implemented completely.

3.14.2.5 *AdvancedQueryAccess Conformance*

If present, the **AdvancedQueryAccess** interface must be implemented completely.

3.14.2.6 *SystemizationAccess Conformance*

If present, the **SystemizationAccess** interface must be implemented completely. This includes implementing the **Systemization** interface.

3.14.2.7 *Systemization Conformance*

All of the systemization interface must be implemented, with the exception of the operations, which may throw the NotImplemented Exception. These operations (**could_association_be_inferred**, **get_entity_graph**, and all of the operations associated with concept expressions) are not required.

3.14.3 *ValueDomainLocator Conformance*

If present, all of the **ValueDomainLocator** interfaces must be implemented. It is only necessary to support one (the default) version of any given value domain.

3.14.3.1 *ValueDomainVersion Conformance*

If present, all of the **ValueDomainVersion** interfaces must be implemented, with the exception of the two pick list operations, which may always throw PickListNotAvailable.

Lexicon Query Glossary

Glossary Terms

The definitions below are specific to this document. While attempts have been made to align the terminology of this document with accepted general definitions, there will be cases where the words used in this document will have a significantly different meaning than they have in general usage. Terms appearing in **boldface** type below are defined elsewhere within this glossary.

Association
An association is a binary predicate applied to an ordered pair of types. The first type is referred to as the **source type** and the second is the **target type**. In this specification, the **source type** must be a **concept code** and the **target type** must be either a single **target element** or a set of **target element**s.

Association Instance
An instance of an association; a binary predicate applied to a specific ordered pair of entities, which must be of the **source type** and **target type** specified in the association itself. The first entity in the ordered pair is referred to as the **source entity**. (n*ot to be confused with* ***Source,*** *as defined below*), and the second is the **target entity.**

Association Qualifier
A **qualified code** that may be attached to a **target element** to provide additional information about the nature of the particular association. With the exception of cardinalities, association qualifiers are left undefined in this specification.

Blob
Acronym for Binary Large Object; used in this document to represent an opaque string of bytes that is passed unchanged between the service and the client.

Characteristic
A non-coded property or attribute associated with a concept code. As defined in this specification, a characteristic provides non-semantic attributes for a concept code.

Coding Scheme	A relation between a set of **concept codes** and a set of **presentations, definitions, comments,** and **instructions,** which serves to designate the intended meaning behind the codes. A coding scheme may also have one or more **systemizations** defined across a subset of the concept codes within the scheme. Coding schemes are evolutionary in nature, with codes being added, deleted, and modified. The intended meaning behind a **concept code** must not change within a given coding scheme.
Coding Scheme Version	A specific release or version of a **coding scheme**. A coding scheme version represents a consistent, fixed image of a coding scheme at a point in time. Therefore, it may define and/or describe only a subset of the **concept codes** contained within the coding scheme itself. Each coding scheme version may also associate a different set of **presentations, definitions,** etc., with a given **concept code** so long as this association does not change the intended meaning of the code.
Comment	A non-defining text string, which is used to annotate or provide remarks about a **concept code** within a **coding scheme version**.
Concept Code	A **local name,** consisting of a fixed sequence of alphanumeric characters, that is used to designate one or more **presentations**, **definitions**, **comments**, **instructions**, etc., within a **coding scheme**.
Concept Description	The set of **definition**s, **comment**s, **instruction**s, and **presentation**s associated with a **concept code** in a given **coding scheme version**.
Concept Expression	A base concept qualified by one or more optional “attribute-value” pairs that serve to further define the total concept. An attribute-value pair consists of an **association** which serves to identify the “attribute” and either a **concept code** or a **characteristic** which serves to identify the value portion of the attribute. An attribute-value pair may also reference an optional list of **association qualifiers.**
Definition	A statement that describes a concept in order to permit its differentiation from related concepts. [4] In this document, definitions are *prose* descriptions, which describe the intended meaning of a **concept code,** permitting its differentiation from the meaning associated with other related **concept codes**.
Implementation Vendor	A company or other organization providing terminology software which presents itself through the interface specification provided in this document.
Instruction	Additional information in either machine- or human- readable form that describes when, where, and/or how a **concept code** should be used.
Language	A “natural language”—any spoken or written language, such as French, English, German, etc.,—as opposed to a formal language, such as Fortran, C, or FOPL.

Lexical Type — A tag indicating whether a presentation falls into any of several special types. The purpose of this tag is to indicate terms that are not generally appropriate for stemming and other natural language techniques. [5] Example: lexical types include “abbreviation,” “acronym,” “eponym,” “trade name,” etc.

Linguistic Group — A group of presentations which are lexical or syntactic variants of each other. As an example, the textual presentations “Atrial Fibrillation”, “Atrial Fibrillations”, “Fibrillation, Atrial” would all belong to the same lexical group, while the textual presentations “Auricular Fibrillation” and “Auricular Fibrillations” would belong to another.

Local Name — An identifier which is unique within the context of a **naming authority**. In this document, a **concept code** is a local name within the context of a **coding scheme**, which is a naming authority. See Section 2.1, “NamingAuthority Module,” on page 2-1 for further details.

Naming Authority — A registered authority which is responsible for managing a set of **local names**, all of which must be unique within the name space of the authority. In this document, a **coding scheme** is a type of naming authority that manages a set of **concept codes**. See Section 2.1, “NamingAuthority Module,” on page 2-1 for further details.

Native Coding Scheme — The primary **coding scheme** supported and provided by a terminology service. Although it is not formally required, the native coding scheme typically will have exact synonyms for *all* of the concepts contained in all of the non-native coding schemes supported by the terminology service.

Pick List — An ordered list of one or more **concept codes** along with a **presentation** deemed appropriate to represent the concept code in an external list or other selection mechanism.

Presentation — A sign or symbol used to represent a **concept code** externally.

Presentation Format — A code which identifies the type of external processing necessary to correctly present a **presentation**. Example: presentation formats include “plain text,” “HTML,” “Rich Text Format,” etc. The Internet MIME codes are the proposed way of representing presentation formats within this document.

Presentation Usage — The association of a **presentation** with a **concept code**.This association carries additional attributes about how the presentation is used in the context of the concept codes, references, etc.

Qualified Code — A **qualified name** which identifies a coded concept within the context of a coding scheme. A qualified name consists of the **coding scheme** identifier (the **naming authority**) and a **concept code** (the **local name**).

Qualified Name — A globally unique name for an entity. A qualified name consists of the combination of a **naming authority** and a **local name**. See Section 2.1, “NamingAuthority Module,” on page 2-1 for further details.

Registration Authority	An organization authorized to register and issue **naming authority** identifiers.
Role	A name which serves as a synonym for either the “source” position or “target” position within a specific association. As an example, a subtyping association places the supertype in the source position and the set of subtypes in the target position. Within this specific association, the role “supertype” would be a synonym for the source and the role “subtype” would be a synonym for the target.
Source	The document, book, person, or other reference from which a **definition**, **comment**, **instruction,** or **presentation** was drawn.
Source Term Type	The code for the use to which a specific **presentation** is put within a **source**. Example:source term types include “Disease Name,” “Language Qualifier,” “Adjective,” etc.
Syntactic Type	A syntactic form that a given **phrase** has within a **linguistic group**. Typical syntactic types may include “plural”, “different spelling”, “different word order”, etc.
Systemization	A structure applied across a set of **qualified codes** and, optionally, **characteristics,** which represents an organization, categorization, classification, or other structuring of the various entities.
Target Element	A choice of **concept code, qualified code,** or **characteristic** that appears in the **target** end of an **association.**
Terminology	A set of terms representing the system of concepts of a particular subject field. [6]
Terminology Service	An implementation of this specification, providing an interface to one or more **coding schemes,** as well as an optional set of **value domains** which serve to correlate the coding schemes with external screens, messages, databases, etc. A terminology service serves to present the contents of a **terminology** externally.
Usage Context	A qualified code that represents a context in which a **presentation** would be deemed appropriate. Usage contexts may include the type of application, user, display device, and other information that is used by the terminology service to pick the most appropriate **presentation** for a **concept code** or set of **concept codes.**
Value Domain	A value domain represents a set of values which may be used to fill a field on a data-entry screen, a column in a database, a field in a message, or some other external entity in which it is possible to record or transfer **concept codes**. A **terminology service** may be used to return a list of **qualified codes,** which are possible values for a field, etc., as represented by a value domain.

OMG IDL A

A.1 Full IDL

The following is the full IDL for this specification.

```
//File: NamingAuthority.idl

#ifndef _NAMING_AUTHORITY_IDL_
#define _NAMING_AUTHORITY_IDL_

#include <orb.idl>

#pragma prefix "omg.org "

module NamingAuthority
{
    enum RegistrationAuthority {
        OTHER,
        ISO,
        DNS,
        IDL,
        DCE };

    typedef string NamingEntity;

    struct AuthorityId {
        RegistrationAuthority    authority;
        NamingEntity             naming_entity;
    };
    typedef string AuthorityIdStr;

    typedef string LocalName;
    struct QualifiedName {
```

```
        AuthorityId authority_id;
        LocalName local_name;
    };
    typedef string QualifiedNameStr;

    exception InvalidInput {};

    interface translation_library
    {

        AuthorityIdStr authority_to_str(
                in AuthorityId authority )
            raises(
                InvalidInput );

        AuthorityId str_to_authority(
            in AuthorityIdStr authority_str )
        raises(
            InvalidInput );

    QualifiedNameStr qualified_name_to_str(
                in QualifiedName qualified_name )
        raises(
            InvalidInput );

    QualifiedName str_to_qualified_name(
                in QualifiedNameStr qualified_name_str )
        raises(
            InvalidInput );
    };
};

#endif // _NAMING_AUTHORITY_IDL_

//File: TerminologyServices.idl
//
#ifndef _TERMINOLOGY_SERVICES_IDL_
#define _TERMINOLOGY_SERVICES_IDL_
#pragma prefix "omg.org"
#include <orb.idl>
#include <NamingAuthority.idl>

// ***************************************************************
//      module:  TerminologyService
// ***************************************************************
module TerminologyServices {
    // . . .
};
#endif /* _TERMINOLOGY_SERVICES_IDL_ */

//***************************************************************
```

```
//    Basic Terms
//*****************************************************************

typedef NamingAuthority::LocalName  ConceptCode;
typedef sequence<ConceptCode  ConceptCodeSeq;

typedef NamingAuthority::AuthorityId CodingSchemeId;
typedef sequence<CodingSchemeId  CodingSchemeIdSeq;

struct QualifiedCode {
     CodingSchemeId  coding_scheme_id;
     ConceptCode  a_code;
};
typedef sequence <QualifiedCode> QualifiedCodeSeq;

typedef  string VersionId;
typedef  sequence<VersionId>  VersionIdSeq;
const  VersionId DEFAULT = "";

struct TerminologyServiceName {
     NamingAuthority::QualifiedName  the_name;
     VersionId  the_version;
};

//*****************************************************************
//    Meta Types
//    See the TerminologyServiceValues module for consts
//*****************************************************************

typedef QualifiedCode AssociationQualifierId;
typedef sequence<AssociationQualifierId> AssociationQualifierIdSeq;

typedef QualifiedCode LexicalTypeId;
typedef sequence<LexicalTypeId> LexicalTypeIdSeq;

typedef QualifiedCode SourceId;
typedef sequence<SourceId> SourceIdSeq;

typedef QualifiedCode SourceTermUsageId;
typedef sequence<SourceTermUsageId> SourceTermUsageIdSeq;

typedef QualifiedCode SyntacticTypeId;
typedef sequence<SyntacticTypeId> SyntacticTypeIdSeq;

typedef QualifiedCode UsageContextId;
typedef sequence<UsageContextId> UsageContextIdSeq;

typedef ConceptCode AssociationId;
typedef sequence<AssociationId> AssociationIdSeq;

typedef ConceptCode LanguageId;
```

```
typedef sequence<LanguageId> LanguageIdSeq;

typedef ConceptCode PresentationFormatId;
typedef sequence<PresentationFormatId> PresentationFormatIdSeq;

//****************************************************************
//    Coding Terms
//****************************************************************
interface LexExplorer;
interface CodingSchemeLocator;
interface ValueDomainLocator;
interface CodingSchemeVersion;
interface PresentationAccess;
interface LinguisticGroupAccess;
interface SystemizationAccess;
interface AdvancedQueryAccess;
interface Systemization;
interface ValueDomainVersion;

typedef string IntlString;
typedef sequence<IntlString> OrderedIntlStringSeq;
typedef sequence<IntlString> IntlStringSeq;
typedef sequence<octet> Blob;
enum Trinary { IS_FALSE, IS_TRUE, IS_UNKNOWN };
typedef sequence<Trinary> TrinarySeq;
typedef sequence<boolean> BooleanSeq;

//****************************************************************
//   Coding Scheme and Coded Concept Terms
//****************************************************************
typedef string PresentationId;
typedef sequence<PresentationId> PresentationIdSeq;
typedef string LinguisticGroupId;
typedef string SystemizationId;
typedef sequence<SystemizationId> SystemizationIdSeq;

struct CodingSchemeInfo {
    CodingSchemeId  scheme_id;
    VersionId  version_id;
    LanguageId  language_id;
};

struct CodingSchemeVersionRefs {
    CodingSchemeId  coding_scheme_id;
    VersionId  version_id;
    LanguageId  language_id;
    boolean  is_default_version;
    boolean  is_complete_scheme;
    CodingSchemeVersion  coding_scheme_version_if;
    PresentationAccess  presentation_if;
    LinguisticGroupAccess  linguistic_group_if;
```

```
    SystemizationAccess  systemization_if;
    AdvancedQueryAccess  advanced_query_if;
};

struct ConceptInfo {
    ConceptCode  a_code;
    IntlString  preferred_text;
};
typedef sequence<ConceptInfo> ConceptInfoSeq;
typedef sequence<ConceptInfoSeq> ConceptInfoSeqSeq;

interface ConceptInfoIter {
    unsigned long max_left();
    boolean next_n(
        in unsigned long n,
        out ConceptInfoSeq concept_info_seq
    );
    void destroy();
};

struct QualifiedCodeInfo {
    QualifiedCode  a_qualified_code;
    IntlString  preferred_text;
};
typedef sequence<QualifiedCodeInfo> QualifiedCodeInfoSeq;

struct Definition {
    IntlString  text;
    boolean  preferred;
    LanguageId  language_id;
    SourceId  source_id;
};
typedef sequence<Definition> DefinitionSeq;

struct Comment {
    IntlString  text;
    LanguageId  language_id;
    SourceId  source_id;
};
typedef sequence<Comment> CommentSeq;

struct Instruction {
    IntlString  text;
    Blob  formal_rules;
    LanguageId  language_id;
    SourceId  source_id;
};
typedef sequence<Instruction> InstructionSeq;

struct SourceInfo {
    SourceId  source_id;
```

```
        SourceTermUsageId  usage_in_source;
        QualifiedCode  code_in_source;
};
typedef sequence<SourceInfo> SourceInfoSeq;

struct PresentationInfo {
        PresentationId  presentation_id;
        PresentationFormatId  presentation_format_id;
        LanguageId  language_id;
        LinguisticGroupId  linguistic_group_id;
};
typedef sequence<PresentationInfo> PresentationInfoSeq;

enum PresentationType {TEXT, BINARY};
union PresentationValue switch(PresentationType) {
        case TEXT : IntlString the_text;
        case BINARY : Blob a_Blob;
};

struct Presentation {
        PresentationId  presentation_id;
        PresentationValue  presentation_value;
};
typedef sequence<Presentation> PresentationSeq;

struct PresentationUsage {
        ConceptCode  concept;
        PresentationId  presentation_id;
        boolean  preferred_for_concept;
        boolean  preferred_for_linguistic_group;
        SyntacticTypeIdSeq  syntactic_type_ids;
        UsageContextIdSeq  usage_context_ids;
        SourceInfoSeq  source_infos;
        LexicalTypeIdSeq  lexical_type_ids;
};
typedef sequence<PresentationUsage> PresentationUsageSeq;

struct LinguisticGroupInfo {
        LinguisticGroupId  Linguistic_group_id;
        LanguageId  language_id;
        PresentationIdSeq  presentation_ids;
};

typedef float Weight;

struct WeightedResult {
        ConceptInfo  the_concept;
        IntlString  matching_text;
        Weight  the_weight;
};
typedef sequence<WeightedResult> WeightedResultSeq;
```

```
interface WeightedResultsIter {
    unsigned long max_left();
    boolean next_n(
        in unsigned long n,
        out WeightedResultSeq weighted_results
    );
    void destroy();
};

//*****************************************************************
//      Advanced Query Terms
//*****************************************************************

typedef string Constraint;
typedef NamingAuthority::QualifiedNameStr ConstraintLanguageId;
typedef sequence<ConstraintLanguageId> ConstraintLanguageIdSeq;
typedef NamingAuthority::QualifiedNameStr PolicyName;
typedef sequence<PolicyName> PolicyNameSeq;
typedef any PolicyValue;

struct Policy {
    PolicyName name;
    PolicyValue value;
};
typedef sequence<Policy> PolicySeq;

//*******************************************************
//*        Systemization Terms
//*******************************************************

typedef string RoleName;
typedef string Characteristic;
enum AssociationRole {SOURCE_ROLE, TARGET_ROLE};
enum MinimumCardinality {OPTIONAL, MANDATORY};
enum MaximumCardinality {SINGLE, MULTIPLE};
struct Cardinality {
    MinimumCardinality  minimum;
    MaximumCardinality  maximum;
};

enum ElementType {
    EXTERNAL_CODE_TYPE,
    LOCAL_CODE_TYPE,
    CHARACTERISTIC_TYPE
};

union RestrictedTargetElement switch(ElementType) {
    case EXTERNAL_CODE_TYPE:QualifiedCode  a_qualified_code;
    case CHARACTERISTIC_TYPE:Characteristic  the_characteristic;
};
```

```
union AssociatableElement switch(ElementType) {
    case EXTERNAL_CODE_TYPE:QualifiedCode  a_qualified_code;
    case LOCAL_CODE_TYPE:ConceptCode  a_local_code;
    case CHARACTERISTIC_TYPE:Characteristic  the_characteristic;
};

struct TargetElement {
    AssociatableElement  target_element;
    AssociationQualifierIdSeq  association_qualifiers;
};
typedef sequence<TargetElement> TargetElementSeq;
typedef sequence<TargetElementSeq> TargetElementSeqSeq;
interface TargetElementSeqIter {
    unsigned long max_left();
    boolean next_n(
        in unsigned long n,
        out TargetElementSeqSeq an_element_seq
    );
    void destroy();
};

typedef ConceptCodeAssociationBaseTypeId;

typedef sequence<unsigned long> IndexList;
struct GraphEntry {
    TargetElement  an_entity;
    IndexList  associated_nodes;
};
typedef sequence<GraphEntry> EntityGraph;

struct AssociationDef {
    AssociationId           association_id;
    AssociationBaseTypeId   base_type;
    RoleName                source_role;
    Cardinality             source_cardinality;
    RoleName                target_role;
    Cardinality             target_cardinality;
    boolean                 target_is_set;
    boolean                 non_coded_allowed;
    Trinary                 transitive;
    Trinary                 symmetric;
    Trinary                 inherited;
    Trinary                 source_target_disjoint;
};

struct AssociationInstance {
    AssociationId           association_id;
    ConceptCode             source_concept;
    TargetElementSeq        target_element_seq;
};
```

```
typedef sequence<AssociationInstance> AssociationInstanceSeq;

interface AssociationInstanceIter {
    unsigned long max_left();
    boolean next_n(
        in unsigned long n,
        out AssociationInstanceSeq association_instance_seq
    );
    void destroy();
};

struct ValidationResult {
    boolean is_valid;
    AssociationQualifierId  validity_level;
};

// Constraint - the "any" below must be of type AttributeValuePair. It
// is "any" because IDL won't allow recursive struct definitions
struct RelatedEntityExpression {
    AssociatableElement         associated_element;
    AssociationQualifierIdSeq   association_qualifiers;
    any                         base_qualifiers;
};

struct AttributeValuePair {
    AssociationRole             element_role;
    AssociationId               the_association_id;
    RelatedEntityExpression     the_entity_expression;
};
typedef sequence<AttributeValuePair  AttributeValuePairSeq;

struct ConceptExpressionElement {
    ConceptCode                 base_code;
    AttributeValuePairSeq       base_qualifiers;
};
typedef sequence<ConceptExpressionElement   ConceptExpression;
typedef sequence<ConceptExpression  ConceptExpressionSeq;

//*****************************************************************
//      Value Domain Terms
//*****************************************************************

typedef QualifiedCode ValueDomainId;
typedef sequence<ValueDomainId> ValueDomainIdSeq;

interface ValueDomainIdIter {
    unsigned long max_left();
    boolean next_n(
        in unsigned long n,
        out ValueDomainIdSeq value_domain_id_seq
    );
```

```
        void destroy();
};

struct PickListEntry {
        QualifiedCode a_qualified_code;
        IntlString pick_text;
        boolean is_default;
};
typedef sequence<PickListEntry> PickListSeq;  //  Ordered

interface PickListIter {
        unsigned long max_left();
        boolean next_n(
                in unsigned long n,
                out PickListSeq pick_list
        );
        void destroy();
};

//************************************************************
//          TerminologyService Exceptions
//************************************************************

//   Used in Multiple Interfaces
//       typically LexExplorer ++
exception NotImplemented{
};
exception UnknownCode {
        ConceptCode bad_code;
};
exception UnknownCodingScheme{
        CodingSchemeId bad_coding_scheme_id;
};
exception UnknownVersion{
        VersionId bad_version_id;
};
exception UnknownValueDomain{
        ValueDomainId bad_value_domain_id;
};
exception NoNativeCodingScheme {
};
exception TranslationNotAvailable {
};
exception TooManyToList {
};
exception NoPickListAvailable {
};
exception AssociationNotInSystemization{
        AssociationId bad_association_id;
};
exception NoSystemizationForCodingScheme {
```

```
};
exception ParameterAlignmentError {
};

// CodingSchemeLocator Exceptions

exception LanguageNotSupported {
    LanguageId bad_language_id;
};

// CodingSchemeVersion exceptions

exception NoPreferredText{
};
exception NoTextLocated{
};
// PresentationAccess exceptions

exception PresentationNotInCodingSchemeVersion{
    PresentationId bad_presentation_id;
};
exception NoPreferredPresentation{
};
exception UnknownPresentationFormat{
    PresentationFormatId bad_presentation_format_id;
};
exception NoPresentationLocated{
};
// LinguisticGroupAccess exceptions

exception LinguisticGroupNotInCodingSchemeVersion{
    LinguisticGroupId bad_linguistic_group_id;
};

// AdvancedQueryAccess exceptions
exception IllegalConstraint {
    Constraint bad_constraint;
};
exception IllegalPolicyName {
    PolicyName name;
};
exception DuplicatePolicyName {
    PolicyName name;
};
exception PolicyTypeMismatch {
    Policy bad_policy;
};

// SystemizationAccess exceptions

exception NoDefaultSystemization{
```

```
};
exception UnknownSystemization {
    SystemizationId systemization_id;
};

// Systemization Exceptions

exception ConceptNotExpandable {
    ConceptCode the_concept;
};
exception NoCommonSubtype{
};
exception NoCommonSupertype{
};
exception InvalidExpression {
    ConceptExpression the_expression;
};
exception UnableToEvaluate {
    ConceptExpression the_expression;
};

// ************************************************************
//      Translation Library
// ************************************************************

interface TranslationLibrary{

    exception InvalidQualifiedName {
    };

    QualifiedCode      str_to_qualified_code(
        in NamingAuthority::QualifiedNameStr qualified_name_str
    ) raises (
        InvalidQualifiedName
);

NamingAuthority::QualifiedNameStr qualified_code_to_name_str(
        in QualifiedCode qualified_code
    );
};

// ************************************************************
//      TerminologyService
// ************************************************************

interface TerminologyService{

    readonly attribute TerminologyServiceNameterminology_service_name;
    readonly attribute LexExplorer lex_explorer;
    readonly attribute CodingSchemeLocator coding_scheme_locator;
    readonly attribute ValueDomainLocator value_domain_locator;
```

```
        CodingSchemeId  Seq   get_coding_scheme_ids();

        CodingSchemeInfo  get_native_coding_scheme_info(
        ) raises(
            NoNativeCodingScheme
        );
};

// ************************************************************
//       LexExplorer
// ************************************************************

interface LexExplorer : TerminologyService{

        IntlString  get_preferred_text(
            in QualifiedCode  a_qualified_code,
            in UsageContextId  Seq context_ids
        ) raises (
            UnknownCodingScheme,
            UnknownCode
        );

        IntlStringSeq get_preferred_text_for_concepts(
            in QualifiedCodeSeq qualified_codes,
            in UsageContextIdSeq context_ids
        );

        Definition get_preferred_definition(
            in QualifiedCode qualified_code
        ) raises (
            UnknownCodingScheme,
            UnknownCode
        );

        ConceptInfoSeq translate_code(
            in QualifiedCode from_qualified_code,
            in CodingSchemeId to_coding_schemeId
        ) raises (
            UnknownCode,
            UnknownCodingScheme,
            TranslationNotAvailable
        );

        ConceptInfoSeqSeq translate_codes(
            in QualifiedCodeSeq from_qualified_codes,
            in CodingSchemeId to_coding_scheme_id
        ) raises (
            UnknownCodingScheme
        );
```

```
void list_concepts(in CodingSchemeId coding_scheme_id,
    in unsigned long how_many,
    out ConceptInfoSeq concept_info_seq,
    out ConceptInfoIter concept_info_iter
) raises (
    UnknownCodingScheme,
    TooManyToList
);

void list_value_domain_ids(
    in unsigned long how_many,
    out ValueDomainIdSeq value_domain_ids,
    out ValueDomainIdIter value_domain_id_iter
) raises (
    TooManyToList
);
boolean is_concept_in_value_domain (
    in QualifiedCode qualified_code,
    in ValueDomainId value_domain_id
) raises (
    UnknownValueDomain
);

TrinarySeq are_concepts_in_value_domains (
    in QualifiedCodeSeq     qualified_codes,
    in ValueDomainIdSeq value_domains
) raises (
    ParameterAlignmentError
);

void get_pick_list(
    in ValueDomainId value_domain_id,
    in UsageContextIdSeq context_ids,
    out PickListSeq pick_list,
    out PickListIter pick_list_iter
) raises (
    TooManyToList,
    UnknownValueDomain,
    NoPickListAvailable
);

Trinary association_exists(
    in QualifiedCode        source_code,
    in TargetElement        target_element,
    in AssociationId        association_id,
    in boolean              direct_only
) raises (
    AssociationNotInSystemization,
    NoSystemizationForCodingScheme,
    UnknownCode
);
```

```
    TrinarySeq associations_exist(
        in QualifiedCodeSeq  source_codes,
        in TargetElementSeq      target_elements,
        in AssociationIdSeq      association_ids,
        in boolean               direct_only
    ) raises (
        ParameterAlignmentError
    );

    void list_associated_target_elements (
        in QualifiedCode         qualified_code,
        in AssociationId         association_id,
        in boolean               direct_only,
        in unsigned long         how_many,
        out TargetElementSeqSeq related_target_seq,
        out TargetElementSeqIter  related_target_iter
    ) raises (
        AssociationNotInSystemization,
        NoSystemizationForCodingScheme,
        UnknownCode
    );
};

void list_associated_source_codes (
        in RestrictedTargetElement      target_element,
        in CodingSchemeId               source_coding_scheme_id,
        in AssociationId                association_id,
        in boolean                      direct_only,
        in unsigned long                how_many,
        out ConceptInfoSeq              concept_info_seq,
        out ConceptInfoIter             concept_info_iter
    ) raises (
        AssociationNotInSystemization,
        NoSystemizationForCodingScheme,
        UnknownCode
    );
};

// ****************************************************************
//       CodingSchemeLocator
// ****************************************************************
interface CodingSchemeLocator:TerminologyService{

    VersionIdSeq get_version_ids(
        in CodingSchemeId coding_scheme_id
    ) raises (
        UnknownCodingScheme
);

    LanguageIdSeq get_supported_languages(
```

```
            in CodingSchemeId coding_scheme_id
        ) raises (
            UnknownCodingScheme
        );

        CodingSchemeVersionRefs get_coding_scheme_version(
            in CodingSchemeId coding_scheme_id,
            in VersionId version_id,
            in LanguageId language_id
        ) raises (
            UnknownCodingScheme,
            UnknownVersion,
            LanguageNotSupported
        );

        CodingSchemeVersionRefs get_native_coding_scheme_version(
        ) raises(
            NoNativeCodingScheme
        );

        VersionId get_last_valid_version(
            in ConceptCode a_code
        ) raises (
            UnknownCode
        );
};

// ***************************************************************
//       ValueDomainLocator
// ***************************************************************
interface ValueDomainLocator:TerminologyService {

        void list_value_domain_ids(
            in unsigned long how_many,
            out ValueDomainIdSeq value_domain_ids,
            out ValueDomainIdIter value_domain_id_iter
        );

        VersionIdSeq get_version_ids(
            in ValueDomainId value_domain_id
        ) raises(
            UnknownValueDomain
        );

        ValueDomainVersion get_value_domain_version(
            in ValueDomainId value_domain_id,
            in VersionId version_id
        ) raises(
        UnknownValueDomain,
            UnknownVersion
        );
```

```
        ValueDomainIdSeq get_value_domain_ids_for_concept(
            in QualifiedCode qualified_code
        );
};

//*************************************************************
//        CodingScheme interfaces
//*************************************************************

//*************************************************************
// A coding scheme consists of the following interfaces
//    interface CodingSchemeVersion:CodingSchemeVersionAttributes
//    interface PresentationAccess:CodingSchemeVersionAttributes
//    interface LinguisticGroupAccess:CodingSchemeVersionAttributes
//    interface SystemizationAccess:CodingSchemeVersionAttributes
//    interface AdvancedQuery:CodingSchemeVersionAttributes
//*************************************************************
//***********************************************************
//        interface CodingSchemeVersionAttributes
//***********************************************************
interface CodingSchemeVersionAttributes {
    readonly attribute CodingSchemeId coding_scheme_id;
    readonly attribute VersionId version_id;
    readonly attribute LanguageId language_id;
    readonly attribute boolean is_default_version;
    readonly attribute boolean is_complete_scheme;
    readonly attribute CodingSchemeVersion coding_scheme_version_if;
    readonly attribute PresentationAccess presentation_if;
    readonly attribute LinguisticGroupAccess linguistic_group_if;
    readonly attribute SystemizationAccess systemization_if;
    readonly attribute AdvancedQueryAccess advanced_query_if;
};

//***********************************************************
//        interface CodingSchemeVersion
//***********************************************************

interface CodingSchemeVersion : CodingSchemeVersionAttributes {

    SyntacticTypeIdSeq get_syntactic_types();
    SourceTermUsageIdSeq get_source_term_usages();
    SourceIdSeq get_scheme_source_ids();
    UsageContextIdSeq get_usage_contexts();

    void list_concepts(
        in unsigned long how_many,
        out ConceptInfoSeq concept_info_seq,
        out ConceptInfoIter concept_info_iter
    );
```

```
boolean is_valid_concept(
    in ConceptCode a_code
);

DefinitionSeq get_definitions(
    in ConceptCode a_code
) raises(
    UnknownCode
);

Definition get_preferred_definition(
    in ConceptCode a_code
) raises(
    UnknownCode
);

CommentSeq get_comments(
    in ConceptCode a_code
) raises (
    NotImplemented,
    UnknownCode
);

InstructionSeq get_instructions(
    in ConceptCode a_code
) raises (
    NotImplemented,
    UnknownCode
);

IntlStringSeq get_all_text(
    in ConceptCode a_code
) raises (
    UnknownCode
);

IntlString get_preferred_text(
    in ConceptCode a_code
) raises (
    UnknownCode,
    NoPreferredText
);

IntlString get_text_for_context(
    in ConceptCode a_code,
    in UsageContextIdSeq context_ids
) raises (
    UnknownCode,
    NoTextLocated
);
```

```
        ConceptCodeSeq get_concepts_by_text(
            in string text
        );

        void match_concepts_by_string(
            in IntlString match_string,
            in unsigned long how_many,
            out WeightedResultSeq weighted_results,
            out WeightedResultsIter weighted_result_iter
        ) raises (
            NotImplemented
        );

        void match_concepts_by_keywords(
            in OrderedIntlStringSeq keywords,
            in unsigned long how_many,
            out WeightedResultSeq weighted_results,
            out WeightedResultsIter weighted_results_iter
        ) raises(
            NotImplemented
        );
};

//*************************************************************
//      PresentationAccess
//*************************************************************
interface PresentationAccess : CodingSchemeVersionAttributes {

        PresentationFormatIdSeq get_presentation_format_ids();

        Presentation get_presentation(
            in PresentationId presentation_id
        ) raises(
            PresentationNotInCodingSchemeVersion
        );

        PresentationInfo get_presentation_info(
            in PresentationId presentation_id
        ) raises(
            PresentationNotInCodingSchemeVersion
        );

        PresentationUsageSeq get_presentation_usages(
            in PresentationId presentation_id
        ) raises(
            PresentationNotInCodingSchemeVersion
        );

        PresentationUsageSeq get_all_presentations_for_concept(
            in ConceptCode a_code
        ) raises(
```

```
            UnknownCode
        );

        PresentationUsage get_preferred_presentation(
            in ConceptCode a_code,
            in PresentationFormatId presentation_format_id
        ) raises(
            UnknownPresentationFormat,
            UnknownCode,
            NoPreferredPresentation
        );

        PresentationUsage get_presentation_for_context(
            in ConceptCode a_code,
            in UsageContextIdSeq context_ids,
            in PresentationFormatId presentation_format_id
        ) raises (
            UnknownPresentationFormat,
            UnknownCode,
            NoPresentationLocated
        );

        PresentationUsage get_all_presentations_for_context(
            in ConceptCode a_code,
            in UsageContextIdSeq context_ids,
            in PresentationFormatId presentation_format_id
        ) raises (
            UnknownPresentationFormat,
            UnknownCode,
            NoPresentationLocated
        );
    };

    //************************************************************
    //      LinguisticGroupAccess
    //************************************************************
    interface LinguisticGroupAccess : CodingSchemeVersionAttributes {

        LexicalTypeIdSeq get_lexical_types();
        LexicalGroupInfo get_lexical_group(
            in LexicalGroupId lexical_group_id
        ) raises(
            LexicalGroupNotInCodingSchemeVersion
        );
    };

    //************************************************************
    //      AdvancedQueryAccess
    //************************************************************

    interface AdvancedQueryAccess : CodingSchemeVersionAttributes {
```

```
        readonly attribute PolicyNameSeq supported_policies;
        readonly attribute ConstraintLanguageIdSeq
        supported_constraint_languages;

        struct query_policies {
            unsigned long return_maximum;
            boolean concept_as_source;
            boolean concept_as_target;
            boolean current_scheme_only;
            boolean direct_associations_only;
        };

        void query (
            in Constraint constr,
            in PolicySeq search_policy,
            in unsigned long how_many,
            out WeightedResultSeq results,
            out WeightedResultsIter results_iter
        ) raises (
            IllegalConstraint,
            IllegalPolicyName,
            PolicyTypeMismatch,
            DuplicatePolicyName
        );
    };

    //************************************************************
    //      SystemizationAccess
    //************************************************************
    interface SystemizationAccess : CodingSchemeVersionAttributes {

        SystemizationIdSeq get_systemization_ids();
        Systemization get_systemization(
            in SystemizationId systemization_id
        ) raises(
            UnknownSystemization
        );
        Systemization get_default_systemization(
        ) raises(
            NoDefaultSystemization
        );
    };

    //************************************************************
    //       Systemization
    //************************************************************
    interface Systemization  {

        readonly attribute SystemizationId systemization_id;
        readonly attribute CodingSchemeVersion coding_scheme_version;
```

```
AssociationIdSeq get_association_ids();

AssociationDef get_association_definition(
    in AssociationId association_id
)raises (
    AssociationNotInSystemization
);

void list_all_association_instances(
    in unsigned long how_many,
    out AssociationInstanceSeq association_instance_seq,
    out AssociationInstanceIter association_instance_iter
);

Trinary are_entities_associated(
    in ConceptCode          source_code,
    in AssociatableElement  target_element,
    in AssociationId        association_id,
    in boolean              direct_only
) raises (
    AssociationNotInSystemization
);

Trinary could_association_be_inferred(
    in ConceptCode          source_code,
    in AssociatableElement  target_element,
    in AssociationId        association_id
) raises (
    AssociationNotInSystemization,
    NotImplemented
);

void list_associated_target_entities (
    in ConceptCode          source_code,
    in AssociationId        association_id,
    in boolean              direct_only,
    in unsigned long        how_many,
    out TargetElementSeqSeq related_elements,
    out TargetElementSeqIter related_elements_iter
) raises (
    AssociationNotInSystemization
);

void list_associated_source_codes (
    in AssociatableElement  target_element,
    in AssociationId        association_id,
    in boolean              direct_only,
    in unsigned long        how_many,
    out ConceptInfoSeq      concept_info_seq,
    out ConceptInfoIter     concept_info_iter
) raises (
```

```
        AssociationNotInSystemization
);

EntityGraph get_entity_graph (
        in AssociatableElement    root_node,
        in AssociationId          association_id,
        in AssociationRole        node_one_role,
        in boolean                direct_only
) raises (
        AssociationNotInSystemization,
        NotImplemented,
        TooManyToList
);

AssociationIdSeq       get_associations_for_source (
        in ConceptCode         source_code
);

AssociationIdSeq  get_associations_for_target (
        in AssociatableElement    target_element
);

ValidationResult validate_concept_expression (
        in ConceptExpression expression
) raises (
        InvalidExpression,
        NotImplemented,
        AssociationNotInSystemization
);

ConceptExpression get_simplest_form (
        in ConceptExpression expression
) raises (
        InvalidExpression,
        NotImplemented,
        AssociationNotInSystemization
);

ConceptExpression expand_concept (
        in ConceptCode concept,
        in AssociationQualifierIdSeq association_qualifier_seq
) raises (
        ConceptNotExpandable,
        UnknownCodingScheme,
        NotImplemented,
        AssociationNotInSystemization
);

Trinary are_expressions_equivalent (
        in ConceptExpression expression1,
        in ConceptExpression expression2
```

```
    ) raises (
        InvalidExpression,
        UnknownCodingScheme,
        AssociationNotInSystemization,
        NotImplemented,
        UnableToEvaluate
    );

    ConceptExpression expression_difference(
        in ConceptExpression expression1,
        in ConceptExpression expression2
    ) raises (
        InvalidExpression,
        UnknownCodingScheme,
        AssociationNotInSystemization,
        NotImplemented,
        UnableToEvaluate
    );

    ConceptExpression minimal_common_supertype (
        in ConceptExpressionSeq expressions
    ) raises (
        InvalidExpression,
        AssociationNotInSystemization,
        NotImplemented,
        NoCommonSupertype
    );

    ConceptExpression maximal_common_subtype (
        in ConceptExpressionSeq expressions
    ) raises (
        InvalidExpression,
        AssociationNotInSystemization,
        NotImplemented,
        NoCommonSubtype
    );
};

//*******************************************
// Value Domain Version
//*******************************************

interface ValueDomainVersion {
    readonly attribute ValueDomainId value_domain_id;
    readonly attribute VersionId value_domain_version_id;
    readonly attribute boolean is_default_version;

    CodingSchemeIdSeq get_schemes_with_extensions();

    QualifiedCodeInfoSeq get_all_extensions();
```

```
        ConceptInfoSeq get_extension_for_scheme(
            in CodingSchemeId coding_scheme_id
        ) raises (
            UnknownCodingScheme
        );

        boolean is_code_in_domain(
            in QualifiedCode qualified_code
        );

        void get_pick_list(
            in UsageContextIdSeq context_ids,
            out PickListSeq pick_list,
            out PickListIter pick_list_iter
        ) raises (
            TooManyToList,
            NoPickListAvailable
        );

        void get_pick_list_for_scheme(
            in CodingSchemeId coding_scheme_id,
            in UsageContextIdSeq usage_context_ids,
            out PickListSeq pick_list,
            out PickListIter pick_list_iter
        ) raises(
            TooManyToList,
            UnknownCodingScheme,
            NoPickListAvailable
        );
};
};
#endif /* _TERMINOLOGY_SERVICES_IDL_ */

//File: TerminologyServiceValues.idl
//
#ifndef _TERMINOLOGY_SERVICE_VALUES_IDL_
#define _TERMINOLOGY_SERVICE_VALUES_IDL_

#pragma prefix "omg.org"
#include <orb.idl>
#include <NamingAuthority.idl>
#include "TerminologyServices.idl"

// ***************************************************************
//      module:  TerminologyServiceValues
// ***************************************************************

module TerminologyServiceValues {

typedef TerminologyServices::ConceptCode ConceptCode;
typedef NamingAuthority::QualifiedNameStr QualifiedNameStr;
```

```
typedef NamingAuthority::AuthorityIdStr AuthorityIdStr;

//***************************************************
//          ValueDomainId Strings
//***************************************************
typedef QualifiedNameStr ValueDomainIdStr;

const ValueDomainIdStr ASSOCIATION_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/AssociationId";
const ValueDomainIdStr ASSOCIATION_QUALIFIER_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/AssociationQualifierId";
const ValueDomainIdStr ASSOCIATION_BASE_TYPE_DOMAIN =
     "IDL:omg.org/TerminologyService/AssociationBaseTypeId";
const ValueDomainIdStr  LANGUAGE_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/LanguageId";
const ValueDomainIdStr LEXICAL_TYPE_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/LexicalTypeId";
const ValueDomainIdStr PRESENTATION_FORMAT_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/PresentationFormatId";
const ValueDomainIdStr SOURCE_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/SourceId";
const ValueDomainIdStr SOURCE_USAGE_DOMAIN =
     "IDL:omg.org/TerminologyService/SourceUsageId";
const ValueDomainIdStr  SYNTACTIC_TYPE_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/SyntacticTypeId";
const ValueDomainIdStr  USAGE_CONTEXT_VALUE_DOMAIN =
     "IDL:omg.org/TerminologyService/UsageContextId";

//***************************************************
//          AssociationId
//***************************************************
typedef ConceptCode AssociationId;
const NamingAuthority::AuthorityIdStr
ASSOCIATION_ID_AUTHORITY_STRING =
     "IDL:org.omg/TerminologyService/Association/";

const AssociationIdIS_COMPOSED_OF =
     "isComposedOf";
const AssociationIdHAS_SUBTYPES =
     "hasSubtypes";
const AssociationIdREFERENCES =
     "references";
const AssociationIdHAS_ATTRITUTES =
     "hasAttributes";

//***************************************************
//          AssociationBaseTypeId
//***************************************************
typedef ConceptCode AssociationBaseTypeId;
const NamingAuthority::AuthorityIdStr
ASSOCIATION_BASE_TYPE_ID_AUTHORITY_STRING =
```

```
    "IDL:org.omg/TerminologyService/AssociationBaseType/";

const AssociationIdWHOLE_PART =
    "wholepart";
const AssociationIdSUBTYPE =
    "subtype";
const AssociationIdREFERENCE =
    "reference";
const AssociationIdNON_SEMANTIC =
    "nonSemantic";

//***************************************************
//        AssociationQualifierId Strings
//***************************************************
typedef QualifiedNameStr AssociationQualifierIdStr;

const AssociationQualifierIdStr MANDATORY =
    "IDL:omg.org/TerminologyService/AssociationQualifier/MAND";
const AssociationQualifierIdStr OPTIONAL =
    "IDL:omg.org/TerminologyService/AssociationQualifier/OPT";
const AssociationQualifierIdStr SINGLE =
    "IDL:omg.org/TerminologyService/AssociationQualifier/SING";
const AssociationQualifierIdStr PLURAL =
    "IDL:omg.org/TerminologyService/AssociationQualifier/PLUR";

//***************************************************
//        LanguageIds
//***************************************************
typedef ConceptCode LanguageId;

const NamingAuthority::AuthorityIdStr
LANGUAGE_ID_AUTHORITY_STRING =
    "DNS:usmarc.omg.org/041/";

const LanguageId DANISH ="DAN";
const LanguageId ENGLISH ="ENG";
const LanguageId FRENCH ="FRE";
const LanguageId GERMAN ="GER";
const LanguageId ITALIAN ="ITA";
const LanguageId SPANISH ="SPA";

//***************************************************
//        LexicalTypeIds
//***************************************************

typedef QualifiedNameStr LexicalTypeIdStr;

const LexicalTypeIdStr ABBREVIATION =  "DNS:umls.hl7.omg.org/LT/ABB";
const LexicalTypeIdStr EMBEDDED_ABBREVIATION =
"DNS:umls.hl7.omg.org/LT/ABX";
const LexicalTypeIdStr ACRONYM =  "DNS:umls.hl7.omg.org/LT/ACR";
```

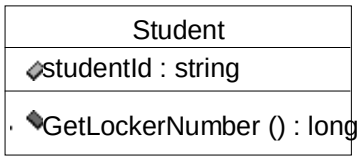
```
const LexicalTypeIdStr EMBEDDED_ACRONYM =
"DNS:umls.hl7.omg.org/LT/ACX";
const LexicalTypeIdStr EPONYM =  "DNS:umls.hl7.omg.org/LT/EPO";
const LexicalTypeIdStr LAB_NUMBER =  DNS:umls.hl7.omg.org/LT/LAB";
const LexicalTypeIdStr PROPER_NAME =
"DNS:umls.hl7.omg.org/LT/NAM";
const LexicalTypeIdStr SPECIAL_TAG =  "DNS:umls.hl7.omg.org/LT/NON
NO";
const LexicalTypeIdStr TRADE_NAME =  "DNS:umls.hl7.omg.org/LT/TRD";

//*****************************************************
//        PresentationFormatIds
//*****************************************************
typedef ConceptCode PresentationFormatId;
const NamingAuthority::AuthorityIdStr
PRESENTATION_FORMAT_AUTHORITY_STRING =
"DNS:omg.org/MIME/";
const PresentationFormatId PLAIN_TEXT =  "text/plain";
const PresentationFormatId RTF =  "application/rtf";
const PresentationFormatId ZIP =  "application/zip";
const PresentationFormatId PDF =  "application/pdf";
const PresentationFormatId GIF_IMAGE =  "image/gif";
const PresentationFormatId BASIC_AUDIO =  "audio/basic";

//*****************************************************
//       SourceIds
//*****************************************************

typedef QualifiedNameStr SourceIdStr;

//*****************************************************
//       SourceUsageTypeId
//*****************************************************

typedef QualifiedNameStr SourceUsageTypeIdStr;
//*****************************************************
//       SyntacticType
//*****************************************************

typedef ConceptCode SyntacticTypeId;

const NamingAuthority::AuthorityIdStr
SYNTACTIC_TYPE_AUTHORITY_STRING =
"DNS:umls.hl7.omg.org/STT";

const SyntacticTypeId CASE_DIFFERENCE =  "C";
const SyntacticTypeId WORD_ORDER =  "W";
const SyntacticTypeId SINGULAR_FORM =  "S";
const SyntacticTypeId PLURAL_FORM =  "P";

//*****************************************************
```

```
//          Query Property Types
//*****************************************************
typedef string TerminologyServiceProperty;

const TerminologyServiceProperty LexicalTypeProperty =  "LexicalTypeId";
const TerminologyServiceProperty AssociationProperty =  "AssociationId";
const TerminologyServiceProperty PreferredTextProperty =  "Preferred-
Text";
const TerminologyServiceProperty DefinitionProperty =  "Definition";
const TerminologyServiceProperty PresentationProperty =  "Presenta-
tionId";

};
#endif /* _TERMINOLOGY_SERVICE_VALUES_IDL_ */
```

Diagram Notation *B*

The notation used in this chapter is the authors' interpretation of a subset of the Unified Modeling Language (UML) notation [23]. It is described briefly below:

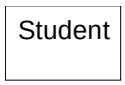
B.1 Class



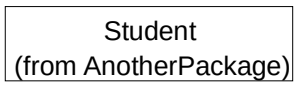
The above diagram represents a class. The name of the *class*, *Student*, is in the top segment of the diagram. The middle segment contains the attributes of the class in the form of <attribute : type>. In this diagram, instances of class *Student* are defined as having one attribute, named *studentId*, and having a data type of *long*. The icon to the left of the attribute indicates that it is publicly available outside of the class. The bottom segment contains the methods which the class implements in the form of <Method(arg list) : return type>. The *Student* class will have one method, *GetLockerNumber*, which returns a data element of type *long*. The icon to the left of the method also indicates that it is publicly available.

All of the attributes and methods described in the specification are defined as publicly available. All of the attributes described in this specification are also implicitly read-only, as the specification is constrained to read-only service access.

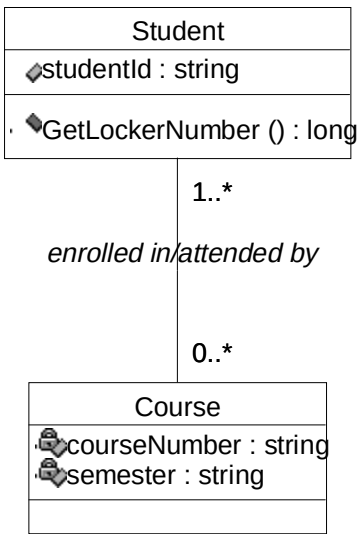
Class diagrams may have their attributes and methods hidden if it helps with the clarity of the overall diagram:



Class names may also have an annotation in parentheses. This annotation indicates the package from which the class is imported. These annotations are not of significance in this document.



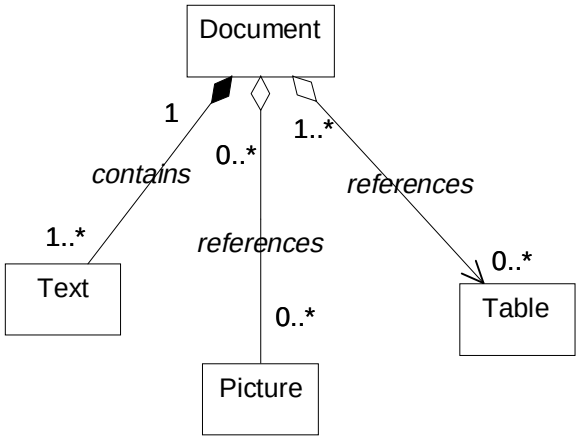
B.2 Association



Two classes may be associated. A solid line between the classes represents this association. The name on the association (e.g., *enrolled in/attended by*) provides a textual description of the specific association. The name before the slash in the association is usually the description of the role the upper or leftmost class takes with respect to the lower or rightmost class. The name after the slash, if any, describes the reverse. The numeric annotations represent the instance cardinality of the class. The above example can be read as:

"A student may be enrolled in zero or more courses." and "A course must be attended by one or more students."

B.2.1 Association Adornments

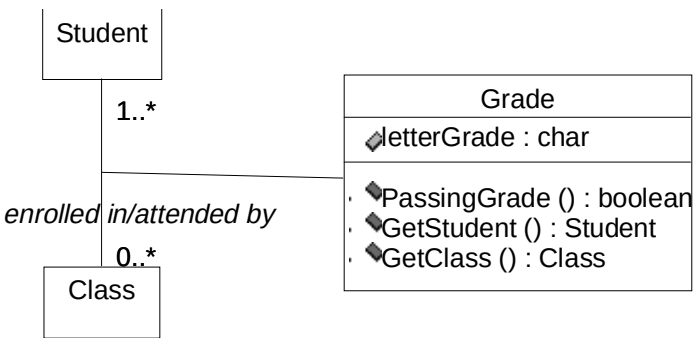


A diamond on one end of an association represents a whole/part association, with the diamond end representing the whole. In the above diagram, the document is the whole and it may consist of the parts *texts*, *pictures* and *tables*. If the diamond is solid, the existence of the parts depends upon the existence of the whole *(has-a)*. If the diamond is hollow, the part may exist outside of the whole *(holds-a)*. These are conventions of this document and not necessarily of UML. In the above diagram, an instance of the *Text* class may not exist if the corresponding instance of the *Document* class does not exist, while an instance of the *Picture* class may exist even when the referencing document is destroyed.

An open arrow indicates one-way navigability. In the above diagram, one would be able to locate all of the *Documents* in which an instance of the *Picture* class exists, but one would not be able to locate all of the *Documents* that reference a given *Table*.

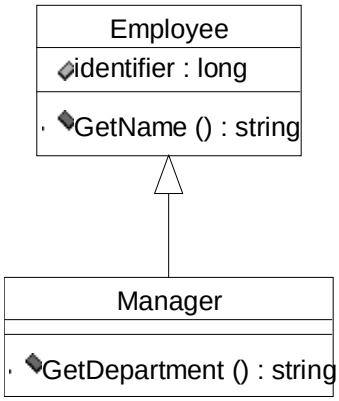
B.2.2 Association Classes

An association may have a class associated with it as well. The properties and methods of this class apply to instances of associations. An association class is marked with a dashed line between the class and the association:



In the above diagram, the class *Grade* applies to the *enrolled in* association. For each *Student/Class* association instance, there will be an instance of the *Grade* class which will have a *letterGrade* attribute as well as methods to determine whether the grade is passing or not. As a convention in this document, navigation between the association class and the associated classes is explicitly provided if needed. *GetStudent* and *GetClass* provide the means of accessing both ends of the specific association instance.

B.3 Inheritence



The hollow arrow indicates that one class is a subclass of another *(is-a)*. The arrow points in the direction of the superclass. A subclass inherits all of the attributes and methods of the superclass. An instance of a subclass will always be an instance of the superclass as well. The converse of this not necessarily true.

In the above example, the class *Manager* is a subclass of the class *Employee*. This means that an instance of the class *Manager* is also an instance of the class *Employee*. (All managers are employees). This does not imply, however, that all employees are managers. The *Manager* class includes the attribute *identifier* and the methods *GetName* and *GetDepartment*.

References C

[1] CORBAmed Lexicon Query Services RFP, January 1997. OMG CORBAmed Document 97-01-04. http://www.omg.org/docs/corbamed/97-01-04.rtf

[2] CORBAServices: Common Object Services Specification. OMG, November 1997. http://www.omg.org/corba/csindx.htm

[3] Telecommunications Topology Service RFP, January 1997. OMG Telecom Document 97-01-02. http://www.omg.org/docs/telecom/97-01-02.pdf

[4] CEN ENV 12264: 1995 (MoSe). Medical informatics – Categorial structures of systems of concepts – Model for representation of semantics. Brussels:CEN, 1995.

[5] Unified Medical Language System. 8th Edition. National Library of Medicine. January 1997. http://www.nlm.nih.gov/pubs/factsheets/umls.html

[6] ISO 1087:1990 – Terminology – Vocabulary.

[7] A. Rossi Mori, “ICNP: towards second and third generation of Terminology Systems.”. IMIA WG6 Conference on Natural Language and Medical Concept Representation. 1997.

[8]] Object Management Architecture Guide. Revision 3.0. Richard Soley, Christopher Stone. OMG, June 1997. http://www.omg.org/library/omaindx.htm

[9] ISO/IEC 8824-1 (1994) Information Technology—Abstract syntax Notation One (ASN.1)—specification of Basic Notation.

[10] P. Mockapetris, " Domain Names - Concepts and Facilities", RFC 1034, Information Sciences Institute, November 1987. http://andrew2.andrew.cmu.edu/rfc/rfc1034.html

[11] The Common Object Request Broker: Architecture and Specification. Revision 2.1. OMG, August 1997. http://www.omg.org/corba/c2indx.htm

[12] DCE 1.1: Remote Procedure Call. OpenGroup Document Number C706, August 1997. First access page: http://www.opengroup.org/public/pubs/catalog/c706.htm then link to http://www.rdg.opengroup.org/onlinepubs/9629399/apdxa.htm#tagcjh_20.

[13] AL Rector, A Gangemi, E Galeazzi, AJ Glowinski, A Rossi-Mori, “The GALEN CORE Model Schemata for Anatomy: Towards a Re-usable Application-Independent Model of Medical Concepts. MIE Proceedings, 1994.

[14] H. Kilov, J. Ross. *Information Modeling – An Object-Oriented Approach.* Prentice Hall, 1994.

[15] B. Potter, J. Sinclair and D. Till. *An Introduction to Formal Specification and Z.* International Series in Computer Science, Hemel Hempstead, UK: Prentice Hall, 1991. As quoted in Kilov.16

[16] ISO/IEC 7826-1 : 1994 Information technology - General Structure for the Interchange of Code Values, Part 1 - Identification of Coding Schemes.

[17] ISO/IEC 7826-2 : 1994 General Structure for the Interchange of Code Values, Part 2 - Registration of Coding Schemes.

[18] American Society of Testing Materials. http://www.astm.org/COMMIT/e-31.htm

[19] Health Level Seven (HL7) Version 2.3. Final Standard. 1997. http://www.mcis.duke.edu/standards/HL7/pubs/version2.3/html/httoc.htm

[20] Generic ICD-9-CM. US Department of Health and Human Services, 1997.

[21] RFC 1521, MIME Mechanisms for Specifying and Describing the Format of Internet Message Bodies

[22] Moore, K., "Representation of Non-Ascii Text in Internet Message Headers" RFC 1522, University of Tennessee, September 1993.

[23] UML Notation Guide, Version 1.1. Rational Software, September 1997. http://www.rational.com/uml/html/notation/

Index

Index

Index