Clinical Observations Access Service (COAS) 4

The OMG documents used to create this chapter were corbamed/99-03-25 and corbamed/99-05-02.

Contents

This chapter contains the following topics.

4.1 Introduction

The Clinical Observations Access Service (COAS) is a set interfaces and data structures with which a server can supply clinical observations.

4.1.1 Definition and Scope of Clinical Observations

To determine the scope of a Clinical Observations Access Service we might start with a definition of “clinical observations.” The 27th Edition of Dorland's Illustrated Medical Dictionary defines “clinical” as,

> “pertaining to a clinic or to the bedside; pertaining to or founded on actual observation and treatment of patients, as distinguished from theoretical or basic sciences.”

Webster's Ninth New Collegiate Dictionary defines “observation” as,

> “2 b: a record obtained by the act of recognizing and noting a fact or occurrence often involving measurement with instruments 3: a judgment on or inference from what one has observed.”

The COAS Request For Proposals (RFP) included the following definition of “clinical observations,”

> “any information that has been captured about a single patient's medical/physical state and relevant context information.”

Webster's Dictionary includes the following definitions of “information,”

> “2 a: (1) knowledge obtained from investigation, study, or instruction (2) INTELLIGENCE, NEWS (3) FACTS, DATA.”

The COAS RFP goes on to add,

> “This [information] may be derived by instruments such as in the case of images, vital signs and lab results or it may be derived by a health professional via direct examination of the patient and transcribed(sic). This term applies to information that has been captured whether or not it has been reviewed by an appropriate authority to confirm its applicability to the patient record.”

It is clear from the dictionary definitions of “observation” and “information” that the common usage of “clinical observations” includes, not just raw measurements and recordings, but also the knowledge and judgments obtained or inferred from them. Based on these definitions and conclusions, the following working definition of “clinical observations” is given, where the lists are intended to specifically include the areas mentioned rather than excluding other related areas:

> “any measurement, recording, or description of the anatomical, physiological, pathological, or psychological state or history of a human being or any sample from a human being, and any impressions, conclusions, or judgments made regarding that individual within the context of the current delivery of health care.”

All observations share a few common features:

> they are made on a specific subject of care, e.g. patient, organ, population;

they represent a snap-shot of that subject in time, either at a particular time, or over some specified interval of time (time in this context includes the notion of both date and time);

they are made, or recorded, by an instrument or a health care professional in some clinical context; and

they are given (either by the patient, the health care institution, or society) some degree of confidentiality.

Observations can be quantitative, qualitative, and recordings, e.g. vital signs and clinical laboratory results, trends in measured values, impressions from a clinical exam, correlation of several qualitative impressions, and images and manipulations of images such as digital subtraction angiography. For the purposes of our information model and the derived IDL, a clinical observation includes any clinically related item that has the necessary context information to enable it to be queried from a COAS server.

4.1.2 Previous Work

A number of the submitters and supporters of this specification have used CORBA for various observation access mechanisms.

3M - Observations are an integral part of the 3M Care Innovation Suite (http://www.mmm.com/market/healthcare/his/product/hems/menu.htm).

Care Data Systems - Observations are part of Care Data System's Integration and Access Channels and the Care Data Repository products (http://www.caredatasystems.com/guide/product-ov.htm).

CareFlow|Net - Observations are part of the CareFlow|Net transcription system (http://www.careflow.com/products.htm).

CERC - Observations are part of the Artemis project (http://www.cerc.wvu.edu/nlm/artemis.html).

HBO & Company - Observations are an integral part of the Clinical Information Systems products (http://206.217.199.68/caci/corporate/prodport.nsf/home).

Los Alamos National Laboratory - Observations are a major component of the TeleMed project (http://www.acl.lanl.gov/TeleMed/).

Philips Medical Systems - Observations are a major component of the MIRACLE project.

Protocol Systems - An observation service (COBS) is the major component of the Acuity Communications Option (ACO) vital signs server.

Sunquest - Observations are a central part of the Sunquest products (http://www.sunquest.com/marketing/).

Each of these projects brings different, complementary perspectives that have contributed to the COAS specification.

4.1.3 Information Model

There are a number of information models that deal with observations data. Some are associated with standards groups and are openly available. Others are the proprietary property of individual companies. The following lists most of the openly available information models that we know of that include observations data.

HL7 - The version 3.0 project is taking the knowledge developed during the previous HL7 standards and describing it in an information model (http://www.mcis.duke.edu/standards/HL7/data-model/HL7/modelpage.html). This is a generalized model for healthcare that does include observations data. This model is subject to change over the next year or two.

DICOM - The Structured Reporting document (supplement 23) of DICOM contains an implied information model for clinical reports which contain observations data (ftp://ftp.nema.org/MEDICAL/DICOM/SUPPS/sup23_fz.pdf).

UK NHS - The UK National Health Service has developed general information models for healthcare, based on a model called COSMOS that contains observations data. See http://smwww1.med.ic.ac.uk/dm/dmgm/ccpm2pt1.doc and http://smwww1.med.ic.ac.uk/dm/dmgm/ccpm2pt2.doc

European Consortia - The European Union has sponsored several projects whose purpose has been to develop and validate information models of healthcare. These include the GEHR and EHCR-SupA.

CEN-TC251 - The European Committee for Standardization Technical Committee 251 has developed several pre-standards that involve models of healthcare. In particular, the CEN/TC251/N97-024 prestandard on “Healthcare Information System Architecture (HISA).”

4.1.4 Dynamic Discovery

Clinical observations cover a very wide set of data types. Servers are likely to offer widely different kinds of data, data formats supported, etc. COAS servers need to expose to clients relevant context information, such as the patient population they deal with, what kinds of observation types are supported, what kind of data formats are supported, which interfaces are implemented, etc. We have made an effort to do this via the `AccessComponent` interface. See Section 4.4.6.2, “AccessComponent Interface,” on page 4-107 for details. However, it is not clear whether this effort will be sufficient to enable the discovery of all necessary capabilities.

4.1.5 Value Domains

The Lexicon Query Service (LQS) contains the ability to query for ValueDomains. ValueDomains are the set of possible codes that can be used for a particular parameter or field. It is expected the LQS ValueDomains can be used by COAS for publishing meta information about the particular service implementation.

4.1.6 Type Negotiation

Servers may support multiple formats for the same type of information, such as images in gif, tiff, and jpeg formats. COAS may need a way for clients to not only determine what formats are supported, but also to select which one(s) they can handle. Specifications for how this is to be accomplished has been left for future revisions of the COAS.

4.1.7 XML Usage

The eXtended Markup Language (XML) is gaining wide interest and support as a flexible format for describing highly structured information (documents).

COAS clients and servers may provide and use XML documents. XML is implicitly supported as a text string, for returned observations. Also, a COAS server could be easily designed to input an XML qualifier as a filter. See the client-implementation example “Progress Note (XML)” on page 4-191 for more details.

4.1.8 Roadmap for Extensions

The COAS needs to provide a basis for future CORBAmed standards for accessing healthcare related information. The COAS specification provides a small number of core definitions, but it is expected that future CORBAmed RFPs will develop additional data definitions that can be used by COAS without extension of the interface, as well as develop extensions to COAS.

At the time of submission, RFPs have been published for a Clinical Image Access Service (CIAS) and a Report Management Service (RMS). These are expected to utilize COAS and/or to extend it. Potential responders to the CIAS and RMS RFPs have contributed to this COAS specification. This specification also includes DsTimeSeries as an example, in the area of vital signs support, of an extension of the data types and operation of COAS.

4.2 Conformance Points

This section describes the various conformance levels possible for a COAS compliant provider of clinical observations.

4.2.1 Conformance Classes

The following taxonomy is defined for specific conformance classes of COAS implementations. An implementation claiming conformance to any of these classes must conform to all of the interfaces specified for that class. An implementation may claim conformance to multiple conformance classes as long as it is conformant to each one it claims. In order for an implementation to be COAS compliant, it must conform to at least one of the conformance classes in the table below.

Each row in the following table includes the specification for a different conformance class. The columns represent the interfaces on the AccessComponent. A star '*' in a column indicates the conformance class in that row includes the interface of that column.

Conformance Class	Query Access	Browse Access	Constraint Access	Asynch Access	Supplier Access	Consumer Access	Observation Loader
Simple COAS	*						
Browse COAS	*	*					
ConstraintLanguage COAS	*		*				
Asynchronous COAS				*			
Supplier COAS					*		
Consumer COAS						*	
Loader COAS							*

- '**Simple COAS**' - This class provides the mechanisms to access observations with a minimum of effort.
- '**Browse COAS**' - This conformance class adds the ability to make queries on the results of previous queries, which enables the more interactive activity of browsing.
- '**ConstraintLanguage COAS**' - This class adds, to the Simple COAS class, the ability to use a constraint language in the construction of queries.
- '**Asynchronous COAS**' - This conformance class is an alternative to the Simple COAS class in that it provides the same access to observations, but it uses an asynchronous connection between the client and server instead of the more common synchronous connection.
- '**Supplier COAS**' - This class is an alternative to the Simple COAS class in that it provides the same access to observations, but it is oriented towards providing access to observations that may arrive in the future, and it uses a messaging communication style to return the observations when they become available. The client must implement the Consumer COAS class (below) in order to receive the observations sent by the Supplier COAS class server.
- '**Consumer COAS**' - This conformance class is the client side to the server interfaces in the Supplier COAS class.
- '**Loader COAS**' - This class provides a mechanism whereby legacy systems can be wrapped with a client COAS interface and can push their data into a COAS server.

4.3 *Information Model*

This section describes the Clinical Observation Access Service information model. Throughout the development of this specification the model has undergone several modifications. The final version depicts a model that is flexible and reusable without adding flexibility that is unlikely to be used.

Several models were reviewed and used in determining the final model. Each model contained things that were valuable in helping us understand the problem and ensuring that we had a model that would accommodate the majority of needs.

Although this model is simplistic, it is also powerful enough to provide the extensibility that is needed in the health care domain. There are many individuals working on efforts to define and categorize health care information. However, there is not a great deal of consensus at this time. Consequently, we needed to provide a model that could accommodate the efforts of these individuals as their work progresses and at the same time make something available today to help in moving the health care information technology forward. “Finding a simple solution takes time and effort, which can be frustrating. People often react to a simple model by saying, “Oh yes, that's obvious” and thinking “So why did it take so long to come up with it?” But simple models are always worth the effort. Not only do they make things easier to build, but more importantly they make them easier to maintain and extend in the future.”[1]

This model presumes that all entities within a health care domain can be modeled as composite or atomic observations. The word *observation* has been a long struggle from the beginning because of the fact that it carried different connotations for various groups and individuals. It is hoped that the reader will understand that the name is merely a placeholder, no name is perfect.

4.3.1 Modeling Notation

The notation used in this chapter comes from a tool[2] that implements the Unified Modeling Language (UML)[3].

4.3.1.1 Modeling Definitions

Many of the definitions given here will be used throughout this chapter.

Class Diagram

A class diagram is a picture for describing generic descriptions of possible systems. Class diagrams and object diagrams are alternate representations of object models. Class diagrams contain classes and object diagrams contain objects.

Collaboration Diagram

Collaboration diagrams show objects, their links, and their messages. They can also contain simple class instances and class utility instances. Each collaboration diagram provides a view of the interactions or structural relationships that occur between objects and object-like entities in the current model.

---

1. Martin Fowler. *Analysis Patterns Reusable Object Models*. Addison Wesley. 1997. P 2.

2. Rational Rose® 98, Rose Enterprise Edition 1998. http://www.rational.com/

3. UML Notation Guide, Version 1.1. Rational Software, September 1997.

http://www.rational.com/uml/html/notation/

Object Diagram

An object diagram shows the existence of objects and their relationships in the logical design of a system. An object diagram may represent all or part of the object structure of a system, and primarily illustrates the semantics of mechanisms in the logical design. A single object diagram represents a snapshot in time of an otherwise transitory event or configuration of objects.

4.3.2 Clinical Observations Model

4.3.2.1 Clinical Observations Model - Class Diagram

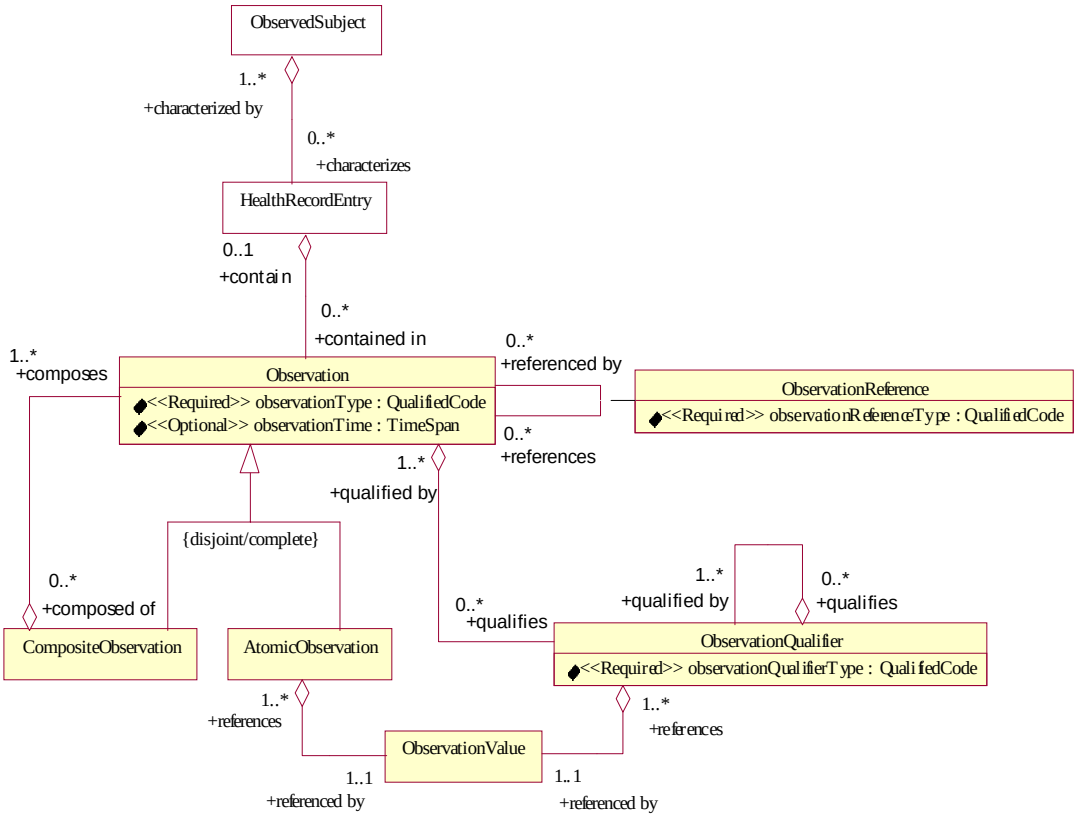

Figure 4-1 COAS Class Diagram

This is a Class Diagram of Clinical Observations created to assist in the design of the Clinical Observations Access Service (COAS). “The logical view of a system describes the existence and meaning of the key abstractions that form the design.”[4]

The HealthRecordEntry and ObservedSubject are represented in the model to show how they may fit into the overall design. Although they can both be supported by this model, we do not explicitly include any specialized services for them. We believe that this model, and the services derived from it, will accommodate them. In the section on Examples they will be discussed.

The following sections document the class diagram. Each of the entities in the class diagram will be discussed.

4.3.2.2 Observation

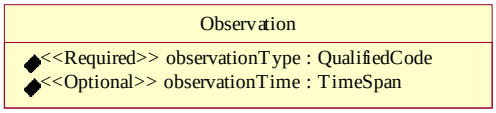

Figure 4-2 Observation

Observation is an abstract class containing attributes that are inherited when a CompositeObservation is needed or when an AtomicObservation is needed.

CompositeObservation and AtomicObservation both inherit from Observation.

Observation is complete and disjoint. Complete meaning no more subclassing can be done off of Observation and disjoint meaning that instances may have only one of the given subtypes as a type.

observationType:QualifiedCode

This is a QualifiedCode that names the Observation. For example, Cardiovascular Examination, Complete Blood Count, Systolic Blood Pressure, etc. The type of this attribute is denoted as a QualifiedCode which comes from the CORBAmed™ Lexicon Query Service[5](LQS). This attribute has been defined as a required attribute.

observationTime: TimeSpan

Denotes the time when the observation reflects a characteristic of the observed subject. (Please reference Section 4.3.3.1, “ObservedSubject - Model,” on page 4-30.) Although is has been defined as optional it is strongly recommended that this attribute exist.

---

4. Grady Booch. *Object Oriented Design with Applications*. Benjamin Cummings. 1991.

5. CORBAmed Lexicon Query Services, March 1998. OMG CORBAmed Document 98-03-22. http://www.omg.org/docs/corbamed/98-03-22.rtf

4.3.2.3 CompositeObservation

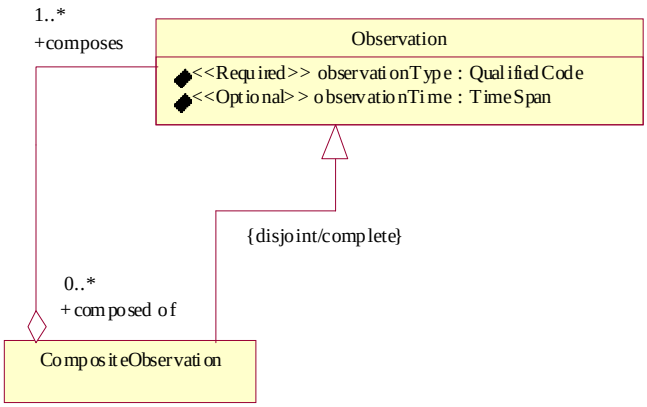

Figure 4-3 CompositeObservation

A CompositeObservation is a container for a set of Observations. Such a set may be a Cardiovascular Examination, a Complete Blood Count, a LabUrineBattery, etc. A CompositeObservation inherits the attributes of an Observation.

A CompositeObservation has no value associated with it, it is used to give some semantic meaning to the contents that it encapsulates. For example, a Complete Blood Count is a CompositeObservation that contains components which are AtomicObservations such as White Blood Count, Red Blood Count, Hematocrit, etc. The AtomicObservations Red Blood Count, etc. themselves have a value associated with them but not Complete Blood Count. Complete Blood Count is merely used to provide a name for the structure of information contained within it.

Relationships with Observation

- Zero or more CompositeObservations are composed of one or more Observations.
- One or more Observations compose zero or more CompositeObservations.

4.3.2.4 AtomicObservation

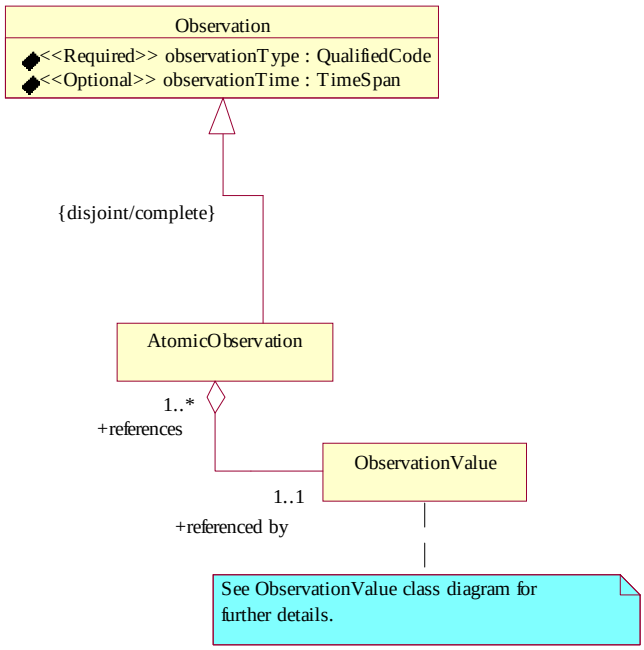

Figure 4-4 AtomicObservation

An AtomicObservation is a single object with an associated value. An AtomicObservation inherits the attributes of an Observation.

Examples of AtomicObservations can be such things as While Blood Count, UrineColor, Systolic Blood Pressure, etc.

Relationships with ObservationValue

- One or more AtomicObservations reference one and only one ObservationValue.
- One and only one ObservationValue is referenced by one or more AtomicObservations.

4.3.2.5 ObservationReference

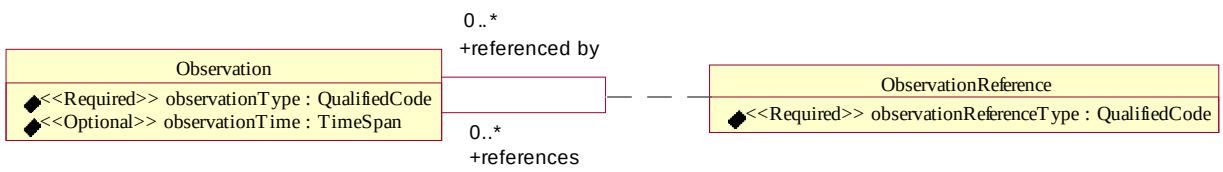

Figure 4-5 ObservationReference

ObservationReference is an associated class defining a relationship between Observations. The observationReferenceType attribute denotes the type of relationship and should come from a well-defined terminology system.

observationReferenceType:QualifiedCode

The observationReferenceType attribute is used to denote the type of relationship that exists between two Observations.

Our intention has been to reference other coding schemes where possible as opposed to creating our own. The CEN Pre-Standard PT27[6] has already started to create a list of these (Table A.5) and could be used as a starting point.

Relationships with Observation

- Zero or more Observations are referenced by zero or more Observations.
- Zero or more Observations references zero or more Observations.

4.3.2.6 ObservationQualifier

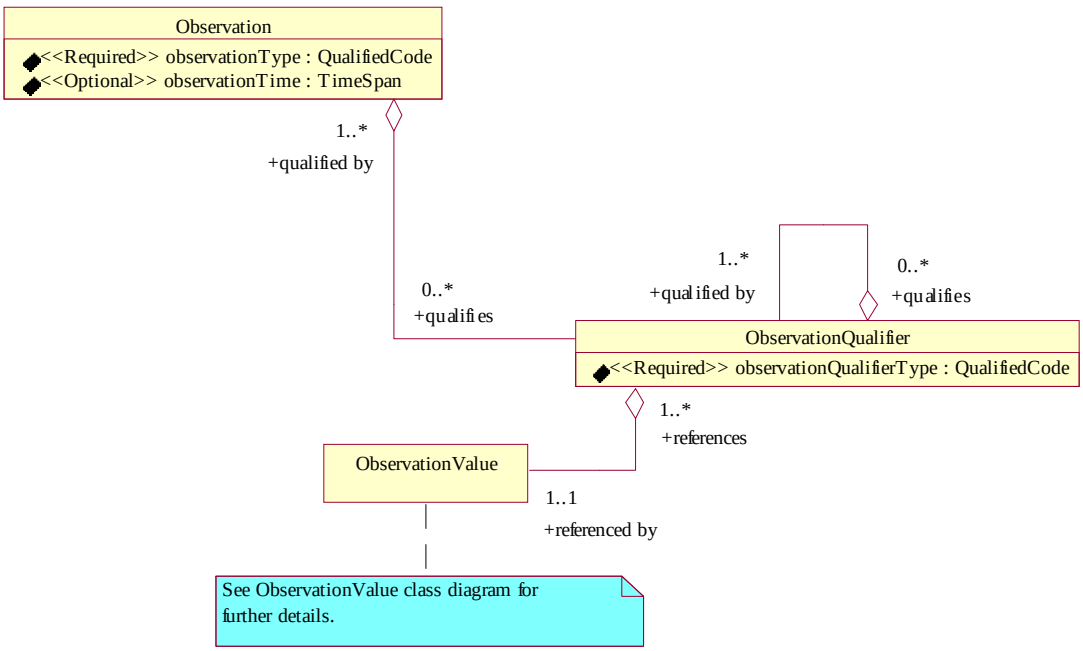

Figure 4-6 ObservationQualifier

6. European PreStandard PT27-N13. *Health Care Informatics Electronic Health Care Record Communication Part 2 - Domain Termlist.* vers.3.0 of 1998-12-01.

An ObservationQualifier is not capable of standing alone. The information represented by the ObservationValue modifies the Observation being qualified. The following tables outline some of the possibilities for ObservationQualifiers:

Dates	**Comments**
Dates of documenting	for such things as create, edit, attesting, storing in a database, transcribing, etc.
dictation	
transcribed	
sign-off	
attestation	
recorded	
Dates of awareness	for such things as reporting by patient, observing by professional, reading a message, etc.
results become available	
Dates of (clinically meaningful) events	for such things as sampling, observing, informing, operating, etc.
observation	
onset	
procedure	
projection	
consultation	
specimen drawn	
lab processing times	
verification	
QA review	
collection	

Roles
originator
collector
legal authenticator
technician/tester
treater
transcriptionist
auditors
observer
observed subject

Modifier
body site [where observed]
subject/Objective
projection [in time]
hypothesis

Instance Status
outside alarm limits [high/low]

outside measurement range [high/low]
critical alarm [high/low]
completion status
QA status
preliminary/final status
normalcy
confidence
report status
active/inactive/remission
rejected/current

Context
source system
patient record categories
facility/location [where]
equipment used
algorithm/formula used [Source data]
protocol/procedure/method
order number/requisition number
encounter number
encounter type
verifier
episode of care
accession number
specimen number
assessment plan case number
health record transaction

Types
allergen
reaction
prognosis
diagnosis
treatment related
pharmacy
expiration date
refills
dose/give rate
intervention type/time

Other
how it was collected
comments
coded comments
normal value
normal range
version
observer
rule out

severity
persistence/recurrence
onset (time?)
procedure time

observationQualifierType:QualifiedCode

The observationQualifierType attribute is a QualifiedCode and should come from a well-defined terminology system. It is used to identify the type of qualifier that is being used to qualify the observation.

Relationships with Observation

- Zero or more ObservationQualifiers qualifies one or more Observations.
- One or more Observations are qualified by zero or more ObservationQualifiers.

Relationships with ObservationValue

- One or more ObservationQualifiers references one and only one ObservationValue.
- One and only one ObservationValue is referenced by one or more ObservationQualifiers.

Relationships with ObservationQualifier

- Zero or more ObservationQualifiers qualifies one or more ObservationQualifiers.
- One or more ObservationQualifiers are qualified by zero or more ObservationQualifiers.

4.3.2.7 ObservationValue

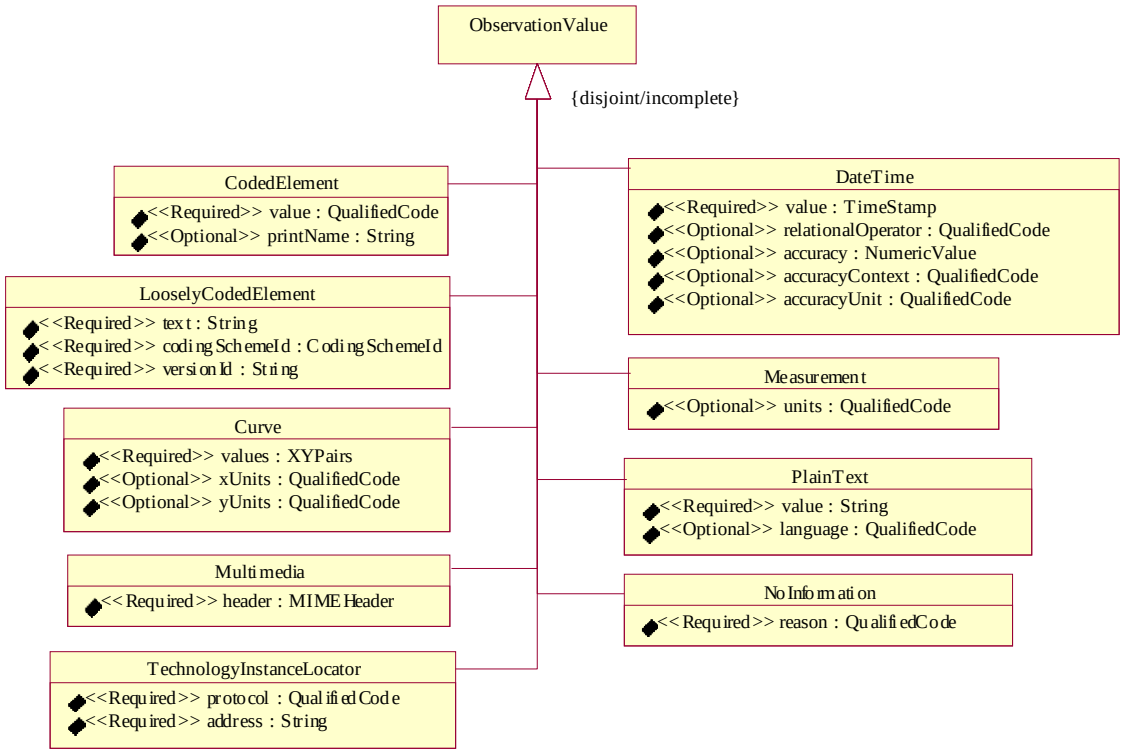

Figure 4-7 ObservationValue

This is a Class Diagram for ObservationValue.

An ObservationValue is a manifestation of forms of biological phenomenon. In this model we have selected a subset of all possible values. We realize that our set is not complete, yet we believe it to be disjoint. There are many efforts underway[7] in determining what these values should and should not be within the arena of healthcare. This model attempts to define those that are most importance at this time. Because ObservationValue is an abstract type, the ability to extend ObservationValue exists and should assist as new or modified ObservationValues are identified.

7. HL7 Version 3 Data Type Redesign Project http://aurora.rg.iupui.edu/v3dt/

CodedElement

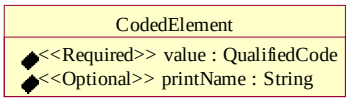

Figure 4-8 CodedElement

The CodedElement provides a mechanism to allow for values that have been coded in some form or another. Coded in the sense that they have a unique identifier. This unique identifier can then be used to ask a terminology system specific questions about the CodedElement, for example, its representation based on some context, or its definition, etc.

value:QualifiedCode

The value attribute is a QualifiedCode and should come from a well-defined terminology system.

printName:String

The **printName** attribute is a String and can be used in conjunction with the value attribute. It is used to provide a textual representation of the value, possibly overriding the definition provided by an LQS.

LooselyCodedElement

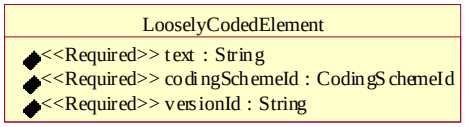

Figure 4-9 LooselyCodedElement

There are times when a code that the user wants cannot be realized or found within a terminology system, e.g. is not in the list of allowable values. In which case the LooselyCodedElement can be used to send text instead. Such instances may occur when there are incomplete lists of coded values or “starter sets” for a given domain, for example, sex, marital status, race, ethnicity, order priorities, etc. The expectation is that the value sent for this field is nearly always coded, but exceptions are allowed.

text:String

The text attribute is a **String** and is used when no CodedElement from a terminology system can be determined.

codingSchemeId:CodingSchemeId

The **codingSchemeId** attribute is of type CodingSchemeId which comes from an LQS and is used to identify the coding scheme where the text was intended.

versionId:String

The **versionId** attribute is a String and is used to identify the version of the coding scheme where the text was intended.

Curve

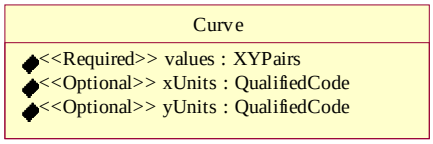

Figure 4-10 Curve

Some observation values can be plotted graphically. Curve is used to assist in the retrieval of such information. It is not the intention to fully identify all the necessary attributes that may be needed for formalized plotting algorithms but rather to supply enough information so that it is possible to plot information in a Cartesian coordinate.

values:XYPairs

The **XYPairs** attribute allows for a sequence of x,y values. Where the x represents those values to be plotted on the x-axis and the y represents those values to be plotted on the y axis.

xUnits:QualifiedCode

The **xUnits** attribute denotes the x axis units. In healthcare this is usually a time axis, (i.e., milliseconds, seconds or minutes). This attribute is a QualifiedCode and should come from a well-defined terminology system.

yUnits:QualifiedCode

The **yUnits** attribute denotes the y axis units. This attribute is a QualifiedCode and should come from a well-defined terminology system.

Multimedia

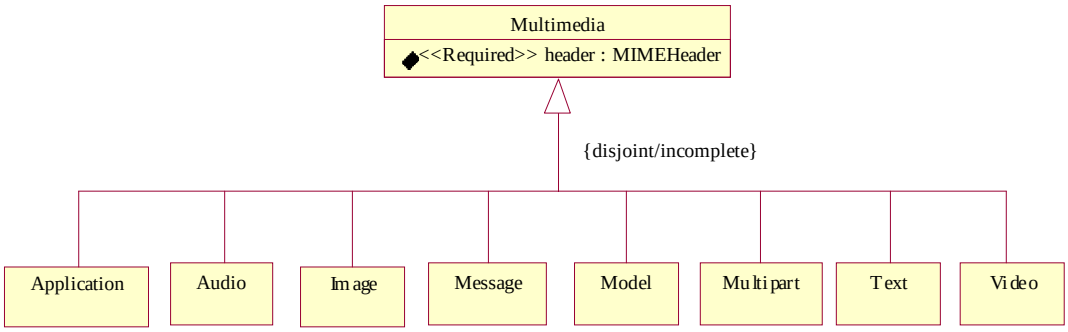

Figure 4-11 Multimedia

There exists a set of documents, collectively called the Multipurpose Internet Mail Extensions, or MIME, that specify a standard for conveying various media types over the Internet.

The MIME Content-Type header field and media type mechanism have been carefully designed to be extensible, and it is expected that the set of media type/subtype pairs and their associated parameters will grow significantly over time. In order to ensure that the set of such values is developed in an orderly, well-specified, and public manner, the MIME standard specifies a registration process which uses the Internet Assigned Numbers Authority (IANA)[8] as a central registry for MIME's various areas of extensibility.

With this in mind we have opted to utilize the MIME as the mechanism for retrieving multimedia information. Rather than attempt to provide a description of each of the media types (Application, Audio, Image, Message, Model, Mulitpart, Text and Video) it seems more reasonable to provide a reference to these. They can be found in the RFC2048[9] document.

header:MIMEHeader

The MIME specifications define a number of header fields that are used to describe the content of a MIME entity. These header fields occur in at least two contexts:

- As part of a regular message header.
- In a MIME body part header within a multipart construct.

The formal definition of these header fields is as follows:

- Entity-headers

---

8. The Internet Assigned Numbers Authority http://www.iana.org/listinfo.html

9. http://www.rfc-editor.org/rfc.html

- MIME-message-headers
- MIME-part-headers

The syntax of the various specific MIME header fields are described in the RFC2045[10] document.

The multimedia data itself follows immediately after the header fields that describe that portion of the data. This data is often encoded such that it is correctly conveyed via legacy internet mail servers which can only handle 7-bit ASCII characters.

TechnologyInstanceLocator

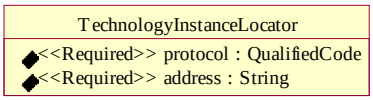

Figure 4-12 TechnologyInstanceLocator

A TechnologyInstanceLocator[11] is used to reference information that has some tie to a technology that can perform some action. It is a generalization of the well-known Universal Resource Locator, or Uniform Resource Locator (URL) concept.

protocol:QualifiedCode

This is the protocol associated with the address. The protocol indicates the technology to be used to interpret the address. This attribute, as a QualifiedCode, and should come from a well-defined terminology system. The following denotes some current internet protocols:

Protocols
HTTP
FTP

address:String

The address attribute contains some structured sequence of characters that the protocol knows how to interpret. For example, www.example.com.

10. http://www.rfc-editor.org/rfc.html

11. HL7 Version 3 Data Type Redesign Project http://aurora.rg.iupui.edu/v3dt/

DateTime

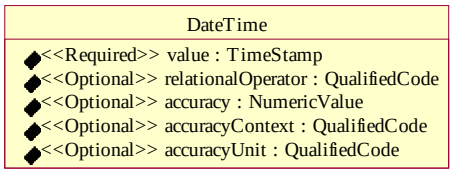

Figure 4-13 DateTime

A DateTime is used to communicate when some event occurred or when some observations was made, recorded, or verified.

value:TimeStamp

The value attribute contains the actual date and time information.

relationalOperator:QualifiedCode

The **relationalOperator** attribute is used to modify the meaning of the value attribute. This attribute is a QualifiedCode and should come from a well-defined terminology system. The basic relational operators are denoted as follows:

Symbolic Representation	Meaning
==	Equal to
!=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

The symbolic representation comes from the C language. The coding scheme may denote the symbolic representation differently based on the context (may be a programming language) but the meaning should be consistent with the foregoing. This attribute can be used to denote that an observation was not at some time value by using the not-equal-to meaning.

accuracy:NumericValue

The **accuracy** attribute allows for a measure of uncertainty to be associated with the DateTime value. For example, plus or minus 2 days, where plus or minus is the accuracyContext and days is the accuracyUnit.

accuracyContext:QualifiedCode

The **accuracyContext** attribute is a QualifiedCode and should come from a well-defined terminology system. The following denotes possible accuracyContexts.

AccurayContexts
Plus or minus
Within

accuracyUnit:QualifiedCode

The **accuracyUnit** attribute is a QualifiedCode and should come from a well-defined terminology system. The following denotes possible accuracyUnits.

AccuracyUnits
MilliSecond
Second
Minute
Hour
Day
Month
Year

Accuracy, accuracyContext and accuracyUnit should be used together as a set.

Measurement

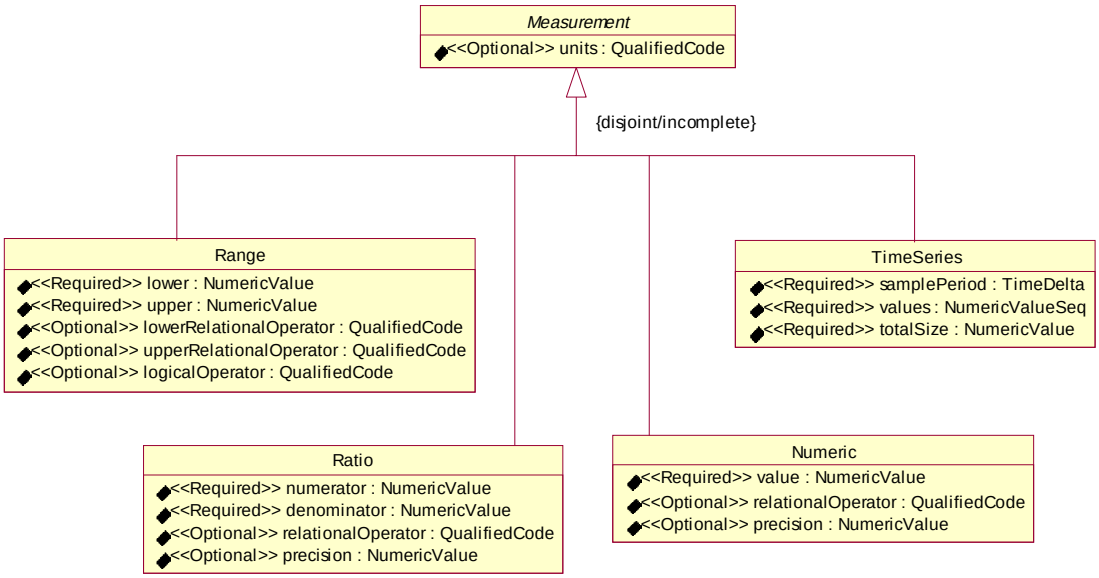

Figure 4-14 Measurement

This is a Class Diagram for Measurement.

In this model we have identified a subset of all possible Measurements. We realize that this is not complete, yet we believe it to be disjoint. Measurements can occur in a wide variety of forms. We have concentrated on those that we believed were widely used.

unit:QualifiedCode

This is the unit associated with the Range, Ratio, TimeSeries, or Numeric. This attribute is a QualifiedCode and should come from a well-defined terminology system.

Range

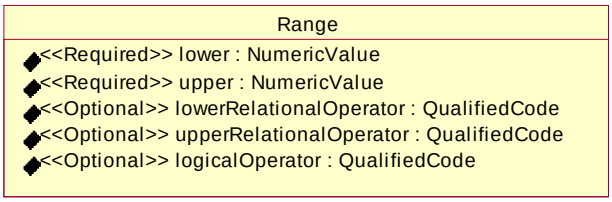

Figure 4-15 Range

Range is used to associate two related values together with the ability to apply relational and logical operators for combinatory expressions. For example, >= 1 && <= 5. It is assumed that the value in the lower attribute is less than or equal to the value in the upper attribute.

lower:NumericValue

This is the lower value of the range.

upper:NumericValue

This is the upper value of the range.

lowerRelationalOperator:QualifiedCode

This is the lower relational operator. This attribute is a QualifiedCode and should come from a well-defined terminology system.

The basic relational operators are denoted as follows:

Symbolic Representation	**Meaning**
==	Equal to
!=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

The symbolic representation comes from the C language. The terminology system may denote the symbolic representation differently based on some context (may be a programming language) but the meaning should be consistent with the foregoing.

upperRelationalOperator:QualifiedCode

This is the upper relational operator. This attribute is a QualifiedCode and should come from a well-defined terminology system. The representation and meaning are as defined for the lowerRelationalOperator described above.

logicalOperator:QualifiedCode

The logical operators allow for the ability to associate two values logically. This attribute is a QualifiedCode and should come from a well-defined terminology system.

The basic logical operators are denoted as follows:

Symbolic Representation	Meaning
&&	And
||	Or

The symbolic representation comes from the C language. The terminology system may denote the symbolic representation differently based on some context (may be a programming language) but the meaning should be consistent with the foregoing.

Ratio

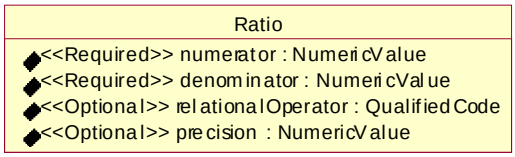

Figure 4-16 Ratio

A ratio value contains a numerator quantity and a denominator quantity. Ratio can be used when referring to clinical laboratory observations that are measured by serial dilution methods.[12] Thus, the ability to express titers which occur in laboratory medicine. A titer is the maximal dissolution at which an analyte can still be detected. Typical values of titers are: “1:32”, “1:64”,”1:128”, etc. Powers of 1/2 or 1/10 are also common. It should be noted that the ratio data type must not be used as a handy representation of two related values. In particular, blood pressure values, commonly reported as 120/80 mm Hg, are not ratios!

12. Dr. Stanley M. Huff et all. Linking a Medical Vocabulary to a Clinical Data Model using Abstract Syntax Notation 1.

numerator:NumericValue

This is the numerator value, the first number in the ratio.

denominator:NumericValue

This is the denominator value, the second number in the ratio. It must not be zero.

relationalOperator:QualifiedCode

This is the relational operator. This attribute is a QualifiedCode and should come from a well-defined terminology system.

The basic relational operators are denoted as follows:

Symbolic Representation	Meaning
==	Equal to
!=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

The symbolic representation comes from the C language. The terminology system may denote the symbolic representation differently based on some context (may be a programming language) but the meaning should be consistent with the foregoing.

precision:NumericValue

The precision attribute is used to provide a level of precision to the ratio. In this case the number of decimal places to the right of the decimal point. For whole number ratios, this attribute is not required

TimeSeries

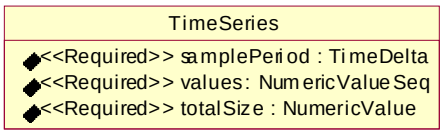

Figure 4-17 TimeSeries

TimeSeries supports the retrieval of an array of values. Within health care, arrays of values are typically samples over time, and so we have included an attribute for the sample period.

samplePeriod:TimeDelta

The samplePeriod is used to denote the length in time between the sampling of two sequential values. This is denoted in seconds.

values:NumericValueSeq

This is a sequence of the scalar values of the actual recordings. These can be octet, short, long, long long, float, double or any.

totalSize:NumericValue

The total number of observations recorded, or the number of values in the sequence.

Numeric

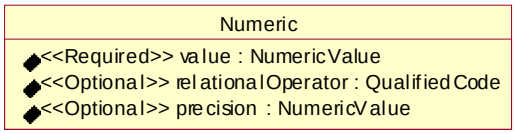

Figure 4-18 Numeric

Numeric is used to communicate a single measurement or quantitative value.

value:NumericValue

This attribute contains the value itself.

relationalOperator:QualifiedCode

This is the relational operator. This attribute is a QualifiedCode and should come from a well-defined terminology system.

The basic relational operators are denoted as follows:

Symbolic Representation	**Meaning**
==	Equal to
!=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

The symbolic representation comes from the C language. The terminology system may denote the symbolic representation differently based on some context (may be a programming language) but the meaning should be consistent with the foregoing.

precision:NumericValue

The precision attribute is used to provide a level of precision to the value. In this case the number of decimals places to the right of the decimal point. For whole numbers this attribute is not required.

PlainText

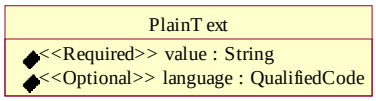

Figure 4-19 PlainText

PlainText is used to communicate observation values as ideas in the form of writing.

value:String

The value attribute is used to contain the text itself.

language:QualifiedCode

The language attribute is used to denote the type of written language used in conveying the value. This attribute is a QualifiedCode and should come from a well-defined terminology system.

The following denotes a subset of potential languages.

Languages
English
French
German
Italian
Spanish

NoInformation

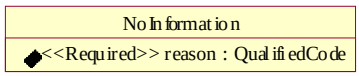

Figure 4-20 NoInformation

There are instances when it is appropriate to denote that information is unavailable or missing. A NoInformation value can occur in place of any other value to express both that specific information is missing and how or why it is missing.[13]

13. HL7 Version 3 Data Type Redesign Project http://aurora.rg.iupui.edu/v3dt/

Reason:QualifiedCode

The reason attribute is used to denote why the information is missing or unavailable. This attribute is a QualifiedCode and should come from a well-defined terminology system.

The following represents a potential set of reasons:

Meaning	Description
Unknown	No information at all. I.e. nothing more is known about the circumstances of missing information
Asked but unknown	The person asked could not supply the information (why?)
Not available	The person asked does have the information somewhere but not available right now (e.g. oh, I wrote down what the doctor said last time, but I didn't bring this piece of paper with me).
Not applicable	An answer to “gestational age” for a patient who is not pregnant.
Not asked	The person who should collect that information forgot to ask.

4.3.3 Examples

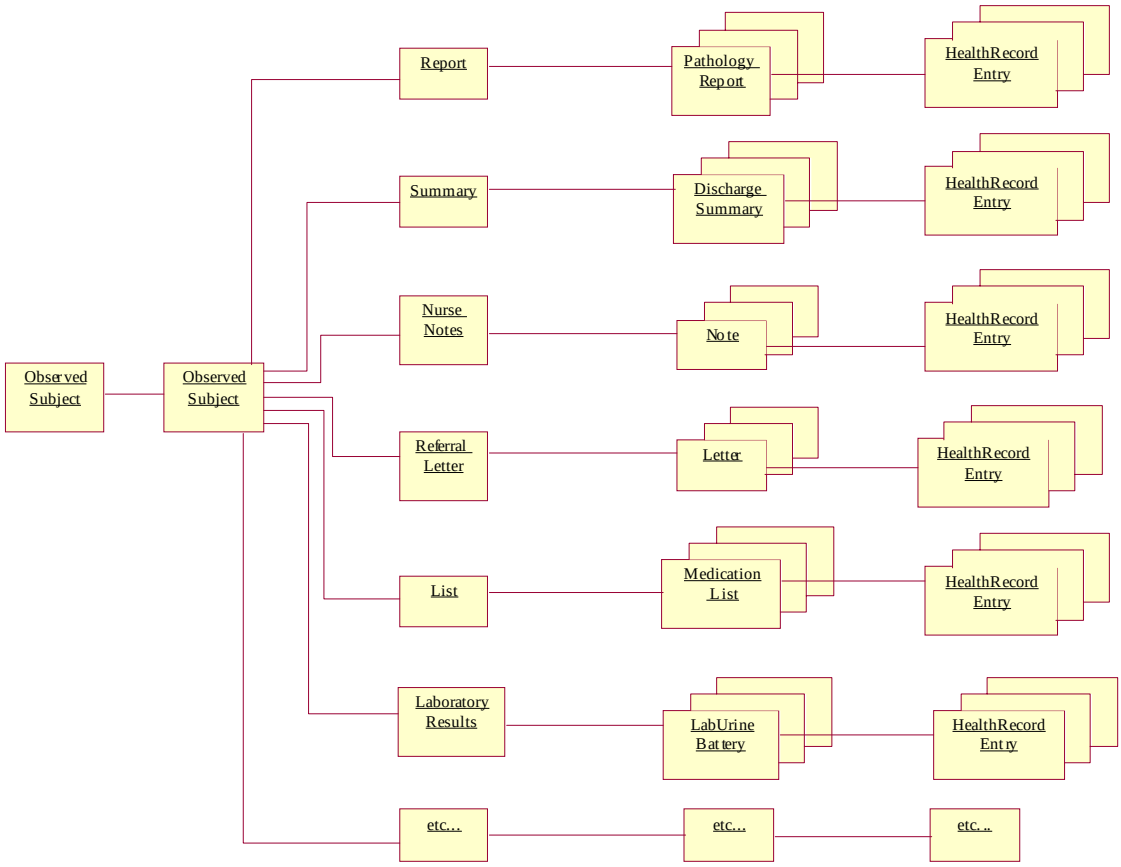

Figure 4-21 Example Health Records

This is a Collaboration Diagram for an example of the health records of an observed subject.

This diagram represents an example of how one might put together a representation of medical information. This diagramming technique is known as a collaboration diagram and is used to represent interactions. It provides a view of the interactions or structural relationships that occur between objects and object-like entities in the current model. In this case an ObservedSubject is considered a Person (patient) that has many links to specific types of medical information categories. For example; reports, nurse notes, and Laboratory Results. These categories themselves have links to specific instances of that type of medical information. These specific instances of medical information have links to specific information that gives meaning about that particular instance of medical information.

So, following one set of links, we see that a Person (patient) has Laboratory Results, which contains instances of LabUrineBatterys where each LabUrineBattery has a link to a HealthRecordEntry.

Also, the ObservedSubject (Person / patient) has links to another ObservedSubject, such as their parent, child, or spouse.

4.3.3.1 *ObservedSubject - Model*

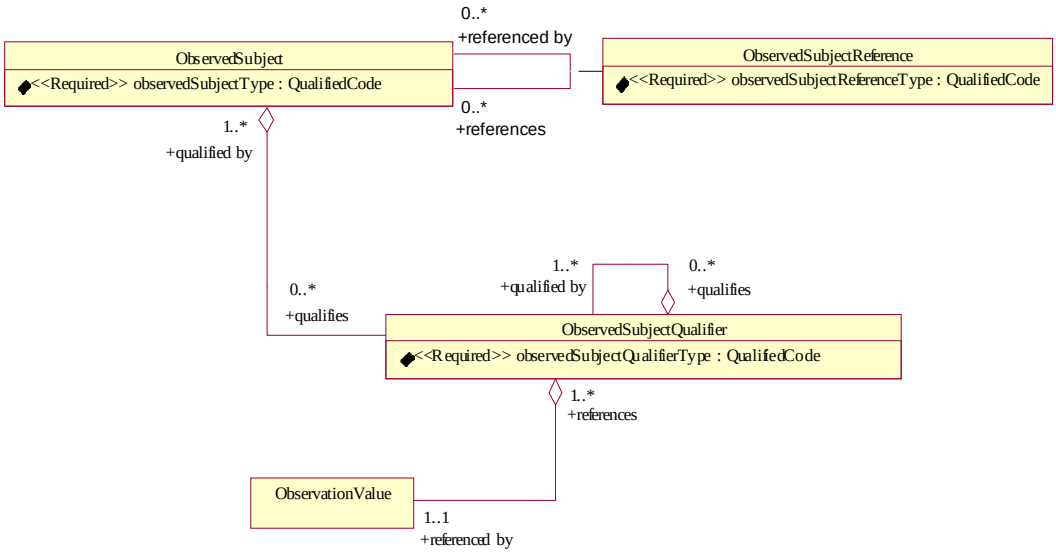

Figure 4-22 ObservedSubject - Model

As mentioned earlier, we have set ObservedSubject outside the scope of this specification and therefore we only include this model as an informational reference. Please notice the similarities with the Clinical Observations Model. The ObservedSubject could merely be placed on top of the Clinical Observations Model. In essence an ObservedSubject is a CompositeObservation.

We focused on the patient when developing this specification but were aware of other ObservedSubjects and modeled accordingly so as not to dismiss the notion of ObservedSubjects other than a patient. The following denotes potential ObservedSubjects:

ObservedSubjects
Patient
Family Unit
Population Cohort
Organ

4.3.3.2 ObservedSubject - Example

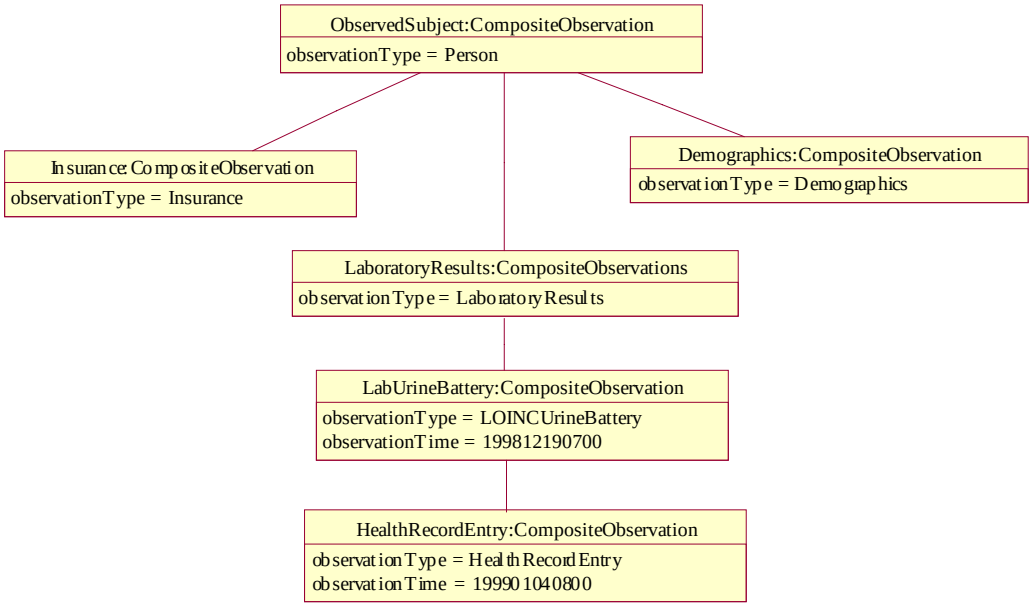

Figure 4-23 ObservedSubject - Example

This is an Object Diagram for one possible representation of an ObservedSubject in a health care information environment

ObservedSubject:CompositeObservation

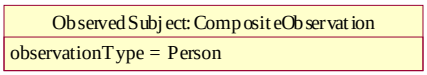

Figure 4-24 ObservedSubject:CompositeObservation

This instance of an ObservedSubject is typed as a Person (patient) and has a CompositeObservation link of type Insurance, a CompositeObservation link of type Demographic and a CompositeObservation link of type LaboratoryResult. This diagram is not meant to be normative but rather to show an example of what an ObservedSubject of type Person (patient) may have associated with it.

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the ObservedSubject. For example, Person, Organ, or Epidemic.

Insurance:CompositeObservation

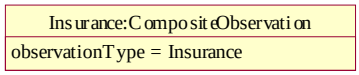

Figure 4-25 Insurance:CompositeObservation

A Person (patient) in a health care information environment usually has a link to some insurance information. This diagram does not fully exploit what a CompositeObservation of type Insurance has as its AtomicObservations or other CompositeObservations. It is merely shown as a possible scenario.

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the CompositeObservation. In this case Insurance.

Demographics:CompositeObservation

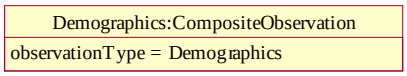

Figure 4-26 Demographics:CompositeObservation

A Person (patient) in a health care information environment usually has a link to some demographic information. This diagram does not fully exploit what a CompositeObservation of type Demographic has as its AtomicObservations or other CompositeObservations. It is merely shown as a possible scenario.

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the CompositeObservation. In this case Demographics.

LaboratoryResults:CompositeObservation

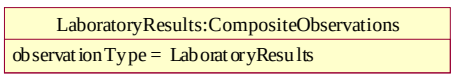

Figure 4-27 LaboratoryResults:CompositeObservation

A Person (patient) in a health care information environment usually has a link to some LaboratoryResults information. In this example the LaboratoryResults has a link to a CompositeObservation of type LabUrineBattery.

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the CompositeObservation. In this case LaboratoryResults.

LabUrineBattery:CompositeObservation

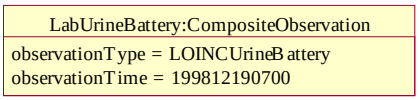

Figure 4-28 LabUrineBattery:CompositeObservation

LaboratoryResults have links to Laboratory Tests. In this case a LabUrineBattery has been depicted.

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the CompositeObservation. In this case LONICUrineBattery.

observationTime: TimeSpan

Denotes the time when the LabUrineBattery became a characteristic of the observed subject. In this case 1998 December 19 at 07:00am.

HealthRecordEntry:CompositeObservation

Figure 4-29 HealthRecordEntry:CompositeObservation

A HealthRecordEntry may be linked to a Laboratory Test. See the HealthRecordEntry Example in this section for a further description.

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the CompositeObservation. In this case HealthRecordEntry.

observationTime: TimeSpan

Denotes the time when the HealthRecordEntry became a characteristic of the LabUrineBattery. In this case 1999 January 1, at 08:00am.

4.3.3.3 LabUrineBattery - Example

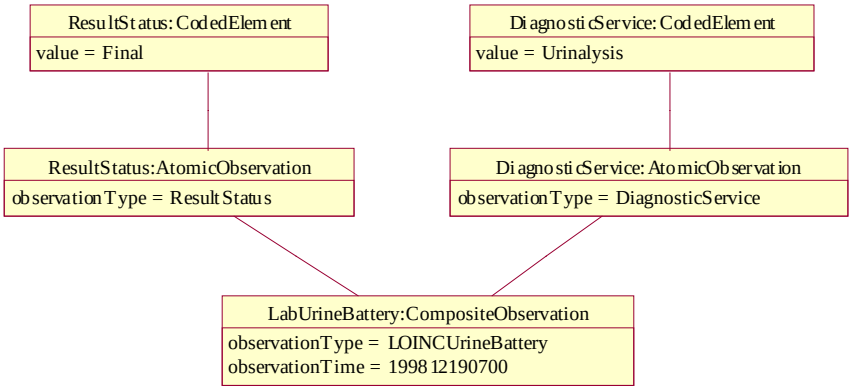

Figure 4-30 LabUrineBattery - Example

This is an Object Diagram for what might be a way to represent a CompositeObservation of type LONICLabUrineBattery. The LONIC™[14] database provides a set of universal names and ID codes for identifying laboratory and clinical observations.

LabUrineBattery:CompositeObservation

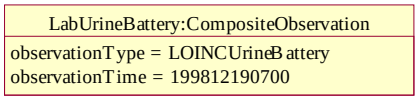

Figure 4-31 LabUrineBattery:CompositeObservation

A Laboratory Test, in this case a LabUrineBattery, has been depicted. This example shows two AtomicObservations being linked to the LabUrineBattery, a ResultStatus and a DiagnosticService.

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the CompositeObservation. In this case LONICUrineBattery.

observationTime: TimeSpan

Denotes the time when the LabUrineBattery became a characteristic of the observed subject. In this case 1998 December 19, at 07:00am.

14. http://www.mcis.duke.edu/standards/HL7/termcode/loinc.htm

ResultStatus:AtomicObservation

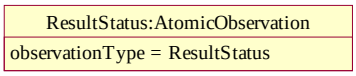

Figure 4-32 ResultStatus:AtomicObservation

LaboratoryResults usually have an indicator to identify the status of the result.

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case ResultStatus.

ResultStatus:CodedElement

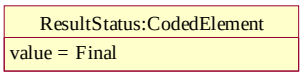

Figure 4-33 ResultStatus:CodedElement

ResultStatus is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a CodedElement and should come from a well defined terminology system.

value:QualifiedCode

The value for a CodedElement is of type QualifiedCode and in this case has been identified as Final.

DiagnosticService:AtomicObservation

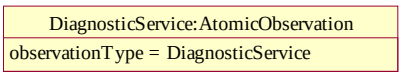

Figure 4-34 DiagnosticService:AtomicObservation

LaboratoryResults may have an indicator of the diagnostic service that performed the laboratory test.

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case DiagnosticService.

DiagnosticService:CodedElement

Figure 4-35 DiagnosticService:CodedElement

DiagnosticService is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a CodedElement and should come from a well defined terminology system.

value:QualifiedCode

The value for a CodedElement is of type **QualifiedCode** and in this case has been identified as Urinalysis.

4.3.3.4 LabUrineBattery - LabSegments

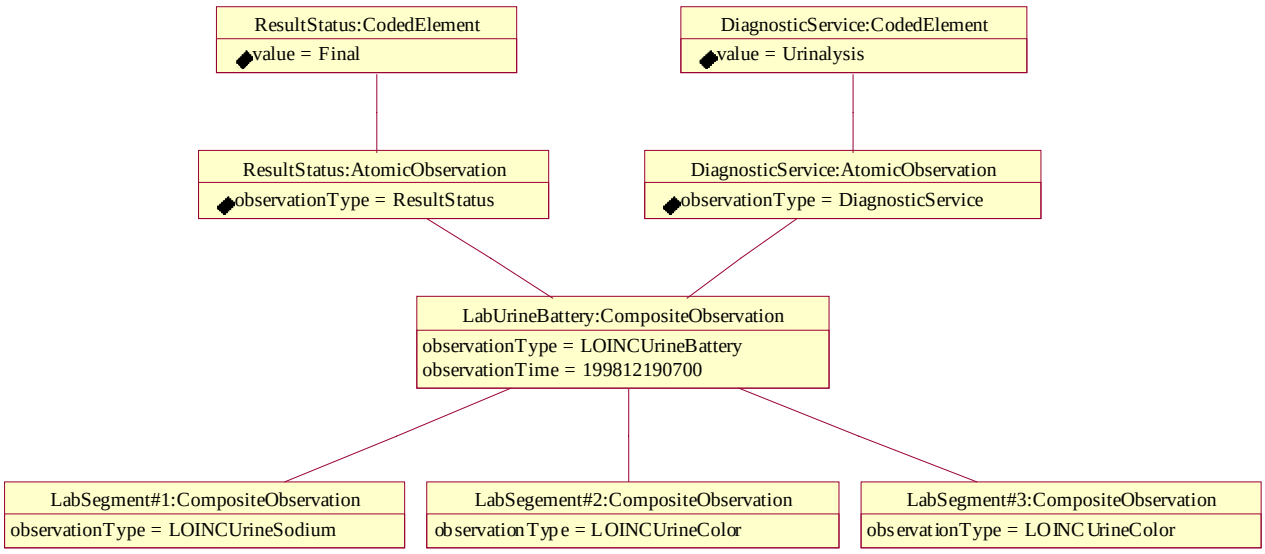

Figure 4-36 LabUrineBattery - LabSegments

This is an Object Diagram showing an extension to the previous LONICLabUrineBattery example with the addition of three specific test results.

LabSegment#1:CompositeObservation

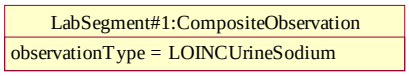

Figure 4-37 LabSegment#1:CompositeObservation

A CompositeObservation of type LOINCUrineSodium

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the CompositeObservation. In this case LOINCUrineSodium.

LabSegment#2:CompositeObservation

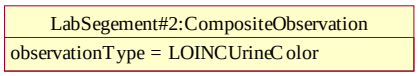

Figure 4-38 LabSegment#2:CompositeObservation

A CompositeObservation of type LOINCUrineColor

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the CompositeObservation. In this case LOINCUrineSodium.

LabSegment#3:CompositeObservation

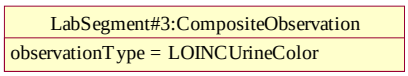

Figure 4-39 LabSegment#3:CompositeObservation

A CompositeObservation of type LOINCUrineColor

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the CompositeObservation. In this case LOINCUrineSodium.

4.3.3.5 LabUrineBattery - LabSegment#1 - LONICUrineSodium

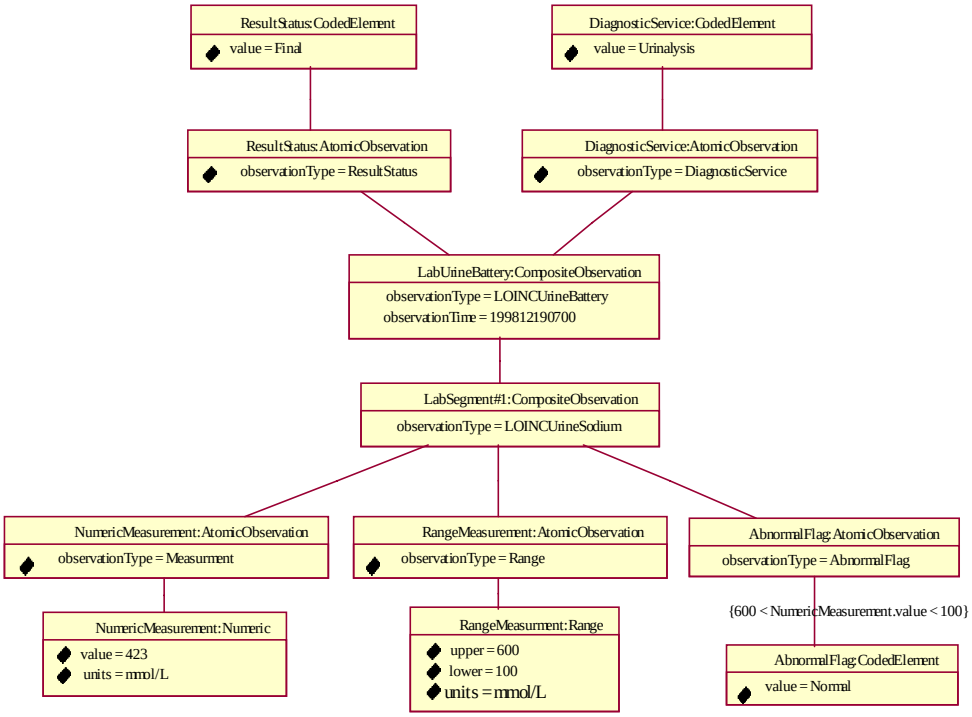

Figure 4-40 LabUrineBattery - LabSegment#1 - LOINCUrineSodium

This is an Object Diagram that shows an extension of the detail in one of the lab test results, namely the LONICLabUrineSodium.

NumericMeasurement:AtomicObservation

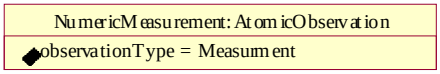

Figure 4-41 NumericMeasurement:AtomicObservation

LOINCUrineSodium has a NumericMeasurement linked to it.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case Measurement.

NumericMeasurement:Numeric

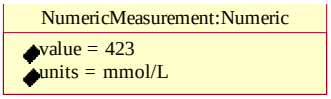

Figure 4-42 NumericMeasurement:Numeric

NumericMeasurement is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a numeric value.

value:NumericValue

The value in this instance is 423.

units:QualifiedCode

The units in this instance are mmol/L.

RangeMeasurement:AtomicObservation

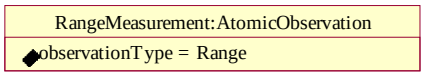

Figure 4-43 RangeMeasurement:AtomicObservation

LOINCUrineSodium has a RangeMeasurement linked to it.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case Range.

RangeMeasurement:Range

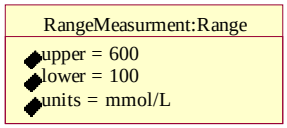

Figure 4-44 RangeMeasurement:Range

RangeMeasurement is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a range.

upper:NumericValue

The upper value of the range is 600.

lower:NumericValue

The lower value of the range is 100.

units:QualifiedCode

The units in this instance are mmol/L.

AbnormalFlag:AtomicObservation

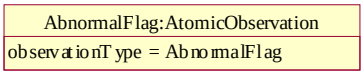

Figure 4-45 AbnormalFlag:AtomicObservation

LOINCUrineSodium has an AbnormalFlag linked to it.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case AbnormalFlag.

AbnormalFlag:CodedElement

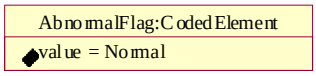

Figure 4-46 AbnormalFlag:CodedElement

AbnormalFlag is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a CodedElement and should come from a well defined terminology system.

value:QualifiedCode

The value for a CodedElement is of type QualifiedCode and in this case has been identified as Normal.

4.3.3.6 LabUrineBattery - LabSegment#2 - LONICUrineColor

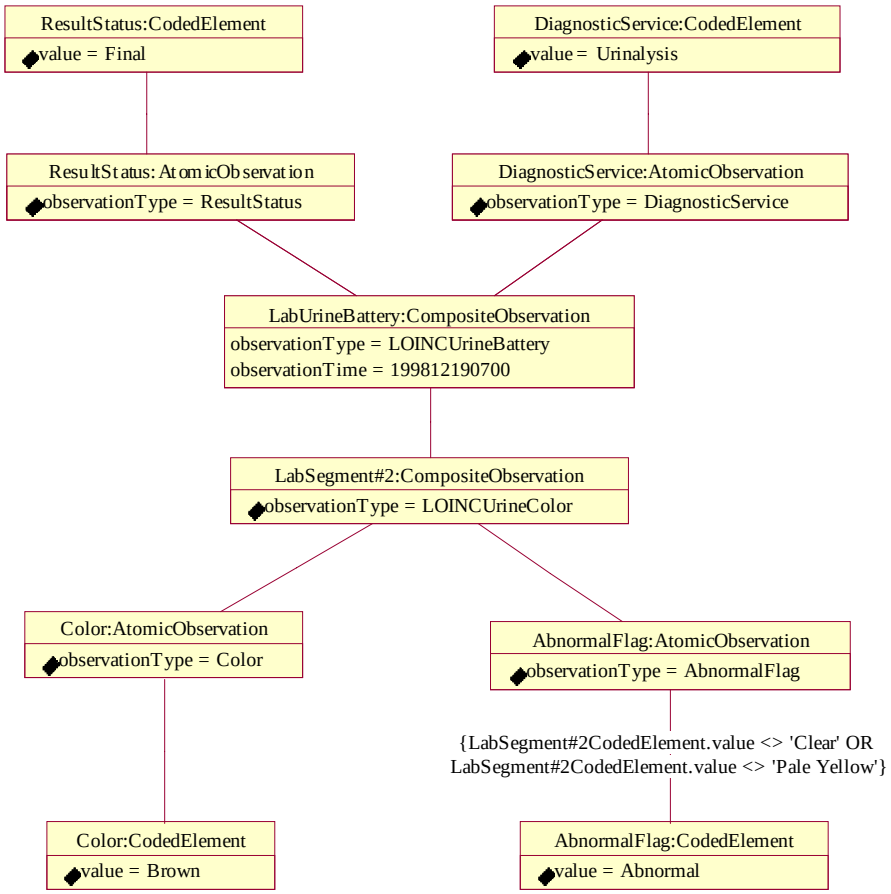

Figure 4-47 LabUrineBattery - LabSegment#2 - LOINCUrineColor

This is an Object Diagram for our example LabUrineBattery - LabSegment - LOINCUrineColor.

Color:AtomicObservation

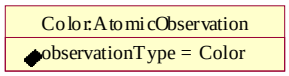

Figure 4-48 Color:AtomicObservation

LOINCUrineSodium has a Color linked to it.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case Color.

Color:CodedElement

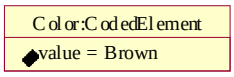

Figure 4-49 Color:CodedElement

Color is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a CodedElement and should come from a well defined terminology system.

value:QualifiedCode

The value for a CodedElement is of type QualifiedCode and in this case has been identified as Brown.

AbnormalFlag:AtomicObservation

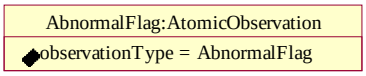

Figure 4-50 AbnormalFlag:AtomicObservation

LOINCUrineSodium has an AbnormalFlag linked to it.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case AbnormalFlag.

AbnormalFlag:CodedElement

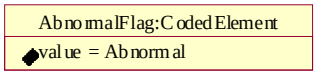

Figure 4-51 AbnormalFlag:CodedElement

AbnormalFlag is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a CodedElement and should come from a well defined terminology system.

value:QualifiedCode

The value for a CodedElement is of type QualifiedCode and in this case has been identified as Abnormal.

4.3.3.7 LabUrineBattery - LabSegment#3 - LOINCUrineColor

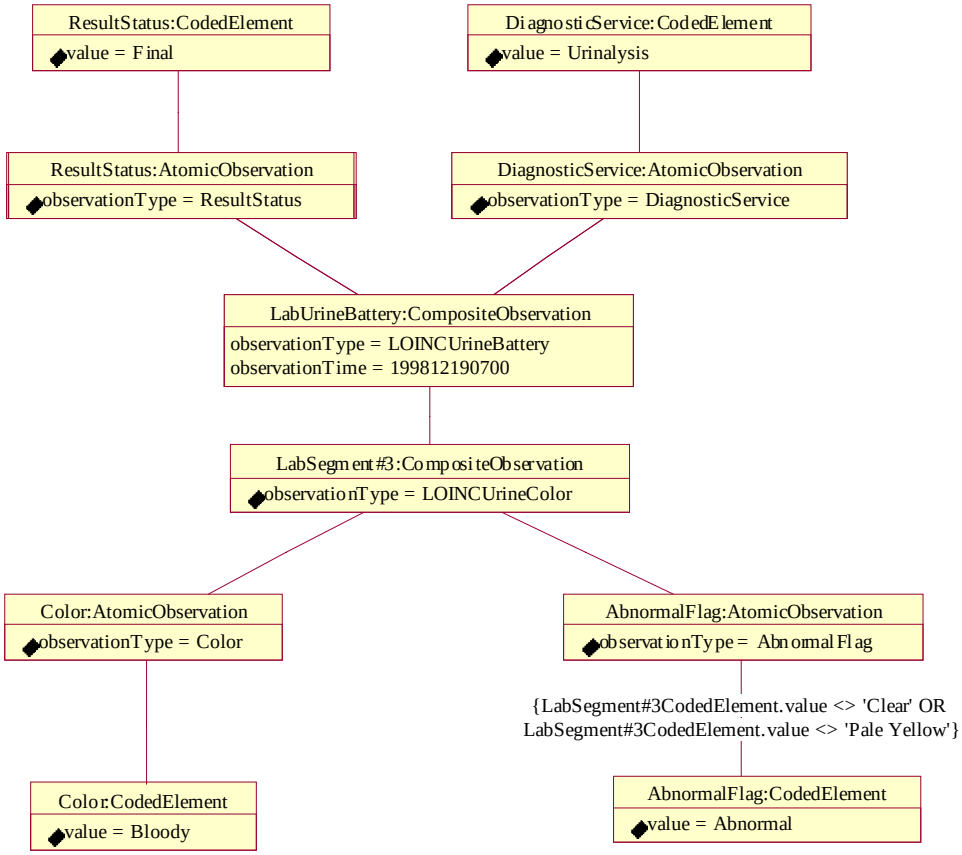

Figure 4-52 LabUrineBattery - LabSegment#3 - LOINCUrineColor

This is an Object Diagram for our example LabUrineBattery - LabSegment - LOINCUrineColor.

Color:AtomicObservation

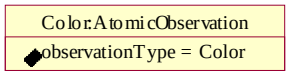

Figure 4-53 Color:AtomicObservation

LOINCUrineSodium has a Color linked to it.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case Color.

Color:CodedElement

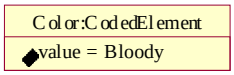

Figure 4-54 Color:CodedElement

Color is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a CodedElement and should come from a well defined terminology system.

value:QualifiedCode

The value for a CodedElement is of type QualifiedCode and in this case has been identified as Bloody.

AbnormalFlag:AtomicObservation

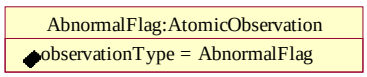

Figure 4-55 AbnormalFlag:AtomicObservation

LOINCUrineSodium has an AbnormalFlag linked to it.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case AbnormalFlag.

AbnormalFlag:CodedElement

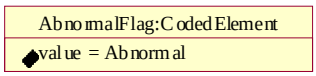

Figure 4-56 AbnormalFlag:CodedElement

AbnormalFlag is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a CodedElement and should come from a well defined terminology system.

value:QualifiedCode

The value for a CodedElement is of type QualifiedCode and in this case has been identified as Abnormal.

4.3.3.8 HealthRecordEntry - Model

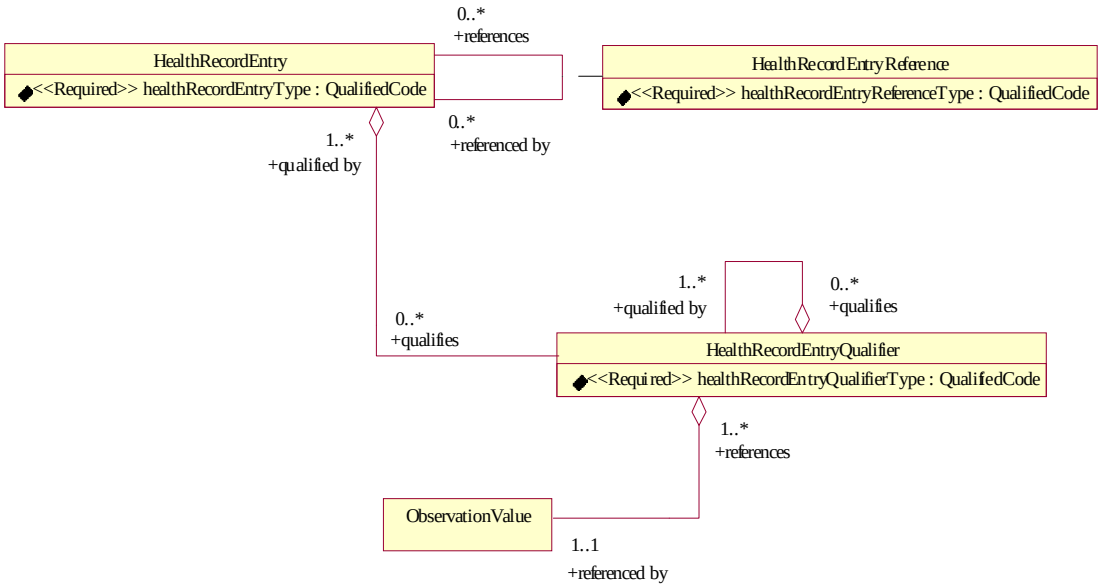

Figure 4-57 HealthRecordEntry - Possible Model

As mentioned early in this chapter, we have set HealthRecordEntry outside the scope of this specification and therefore we only include this example as an informational reference. Please notice the similarities with the Clinical Observations Model. The HealthRecordEntry could merely be placed on top of the Clinical Observations Model. In essence an HealthRecordEntry is a CompositeObservation.

4.3.3.9 HealthRecordEntry - Example

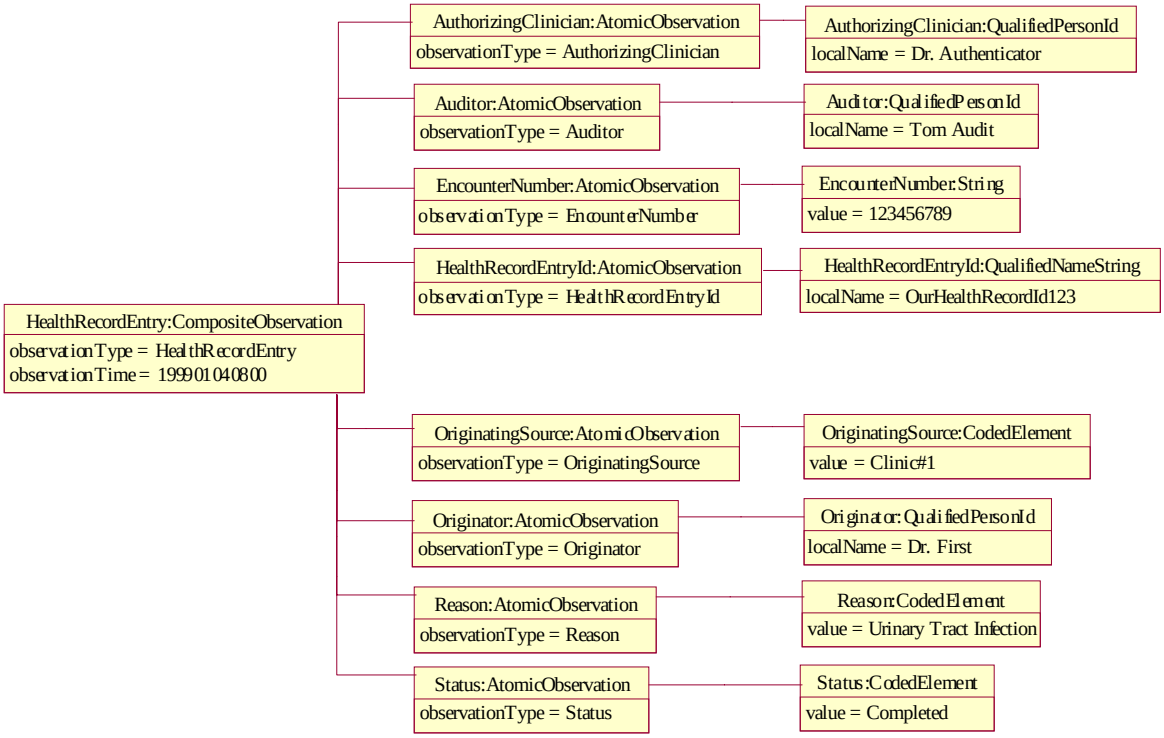

Figure 4-58 HealthRecordEntry - Example

This is an example Object Diagram for a possible HealthRecordEntry.

HealthRecordEntry:CompositeObservation

Figure 4-59 HealthRecordEntry:CompositeObservation

A HealthRecordEntry can be used to provide transactional information that is associated with an Observation.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the CompositeObservation. In this case HealthRecordEntry.

observationTime: TimeSpan

Denotes the time when the HealthRecordEntry became a characteristic of the subject of care. In this case 1999 January 1, at 08:00am.

AuthoringClinician:AtomicObservation

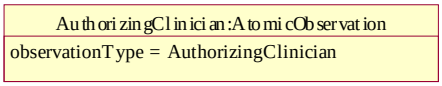

Figure 4-60 AuthoringClinician:AtomicObservation

The AuthoringClinician can be used to identify the responsible individual.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case AuthoringClinician.

AuthoringClinician:QualifiedPersonId

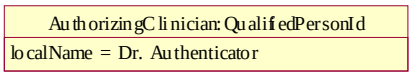

Figure 4-61 AuthoringClinician:QualifiedPersonId

AuthoringClinician is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a QualifiedPersonId and should come from some Enterprise Master Patient Index. There are other attributes associated with a QualifiedPersonId other than localName but not included in this example for brevity. Further information can be attained from the CORBAmed™ Person Identification Service[15] (PIDS).

localName:String

The localName is of type String and in this case has been identified as Dr. Authenticator.

Auditor:AtomicObservation

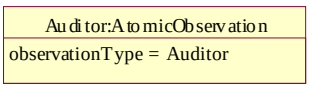

Figure 4-62 Auditor:AtomicObservation

15. CORBAmed Person Identification Services, March 1998. OMG CORBAmed Document 98-02-29. http://www.omg.org/docs/corbamed/98-02-29.rtf

The Auditor can be used to identify the individual from the medical records department that was responsible for finalizing this information.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case Auditor.

Auditor:QualifiedPersonId

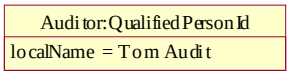

Figure 4-63 AuthoringClinician:QualifiedPersonId

AuthoringClinician is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a QualifiedPersonId and should come from some Enterprise Master Patient Index.

localName:String

The localName is of type String and in this case has been identified as Tom Audit.

EncounterNumber:AtomicObservation

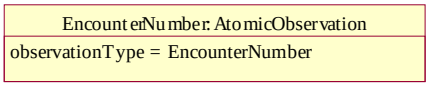

Figure 4-64 EncounterNumber:AtomicObservation

The EncounterNumber can be used as some unique system identifier for this particular instance of information.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case EncounterNumber.

EncounterNumber:String

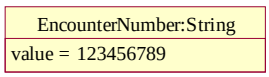

Figure 4-65 EncounterNumber:String

EncounterNumber is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a String.

value:String

The value is of type String and in this case has been identified as 123456789.

HelathRecordEntryId:AtomicObservation

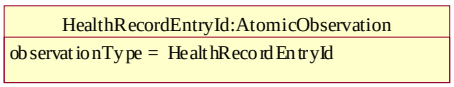

Figure 4-66 HelathRecordEntryId:AtomicObservation

The HealthRecordEntryId can be used as some unique system identifier for the HealthRecordEntry itself.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case HealthRecordEntryId.

HealthRecordEntryId:String

Figure 4-67 HealthRecordEntryId:String

HealthRecordEntryId is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a QualifiedNameString. There are other attributes associated with a QualifiedNameString other than localName but not included in this example for brevity. QualifiedNameString is identified in the CORBAmed™ LQS.

localName:String

The value is of type String and in this case has been identified as OurHealthRecordId123.

OriginatingSource:AtomicObservation

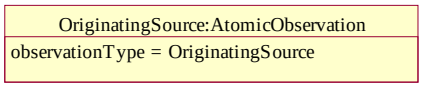

Figure 4-68 OriginatingSource:AtomicObservation

The OriginatingSource can be used to identify where this information originated from.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case OriginatingSource.

OriginatingSource:CodedElement

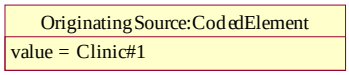

Figure 4-69 OriginatingSource:CodedElement

OriginatingSource is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a CodedElement.

value:QualifiedCode

The value for a CodedElement is of type QualifiedCode and in this case has been identified as Clinic#1.

Originator:AtomicObservation

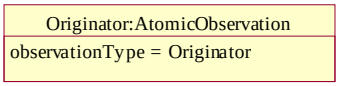

Figure 4-70 Originator:AtomicObservation

The Originator can be used to identify who was the originator of this information.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case Originator.

Originator:QualifiedPersonId

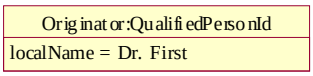

Figure 4-71 Originator:QualifiedPersonId

Originator is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a QualifiedPersonId and should come from some Enterprise Master Patient Index.

localName:String

The localName is of type String and in this case has been identified as Dr. First.

Reason:AtomicObservation

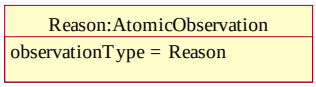

Figure 4-72 Reason:AtomicObservation

The Reason can be used to identify why this was necessary.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case Reason.

Reason:CodedElement

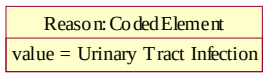

Figure 4-73 Reason:CodedElement

Reason is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a CodedElement.

value:QualifiedCode

The value for a CodedElement is of type QualifiedCode and in this case has been identified as Urinary Tract Infection.

Status:AtomicObservation

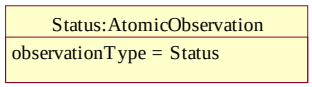

Figure 4-74 Status:AtomicObservation

The Status can be used to indicate the state of the information.

observationType: QualifiedCode

This is a QualifiedCode that provides the type of the AtomicObservation. In this case Status.

Status:CodedElement

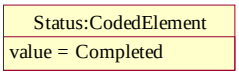

Figure 4-75 Status:CodedElement

Status is an AtomicObservation and therefore has an ObservationValue linked to it. In this case it is a CodedElement.

value:QualifiedCode

The value for a CodedElement is of type QualifiedCode and in this case has been identified as Completed.

4.4 DsObservationAccess Service

The **DsObservationAccess** service has many interfaces and definitions, and can be viewed from several perspectives. Several viewpoints are first shown by UML diagrams. Each viewpoint is chosen to describe one aspect of the entire service and its types. These initial viewpoints are not complete descriptions, showing only relevant information for a viewpoint while hiding irrelevant information.

After the viewpoints, all IDL types and interfaces are described in detail.

4.4.1 Viewpoints

This section provides an overview of the **DsObservationAccess** service. The service is presented from several viewpoints which may include overlapping information. The viewpoints are not meant to be orthogonal.

4.4.1.1 Navigable Relationships Viewpoint

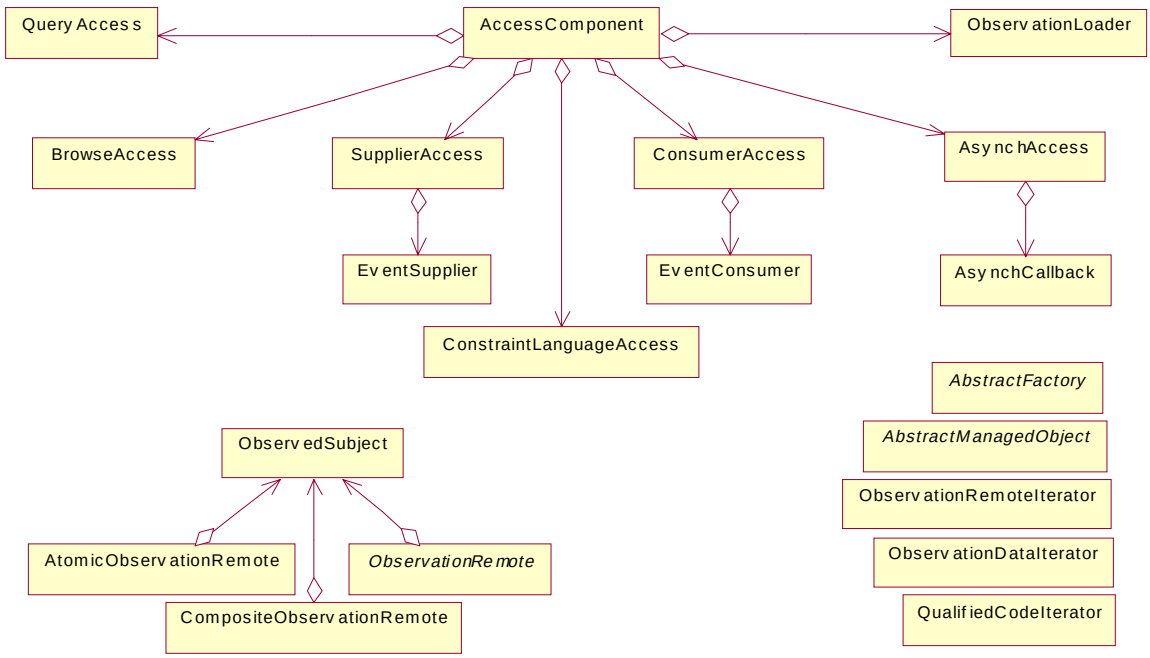

Figure 4-76 Direct navigation between interfaces.

All interfaces defined in the **DsObservationAccess** module are shown on the diagram above. Iterators and abstract interfaces do not have direct navigation. Attributes and operations are hidden in this diagram in order to focus in on the navigable relationships.

Only direct navigation is shown. Some of the query interfaces have indirect mechanisms to traverse to other interfaces as well. For example, a browse operation could return references to an **ObservedSubject** or **ObservationRemote**.

The starting point in the **DsObservationAccess** service is the **AccessComponent** interface. From there a client can traverse to the other core interfaces on the component. This traversal capability is one of the basis for the componentization (See Section 4.4.1.3, “Componentization Viewpoint,” on page 4-56).

4.4.1.2 Interface Inheritance Viewpoint

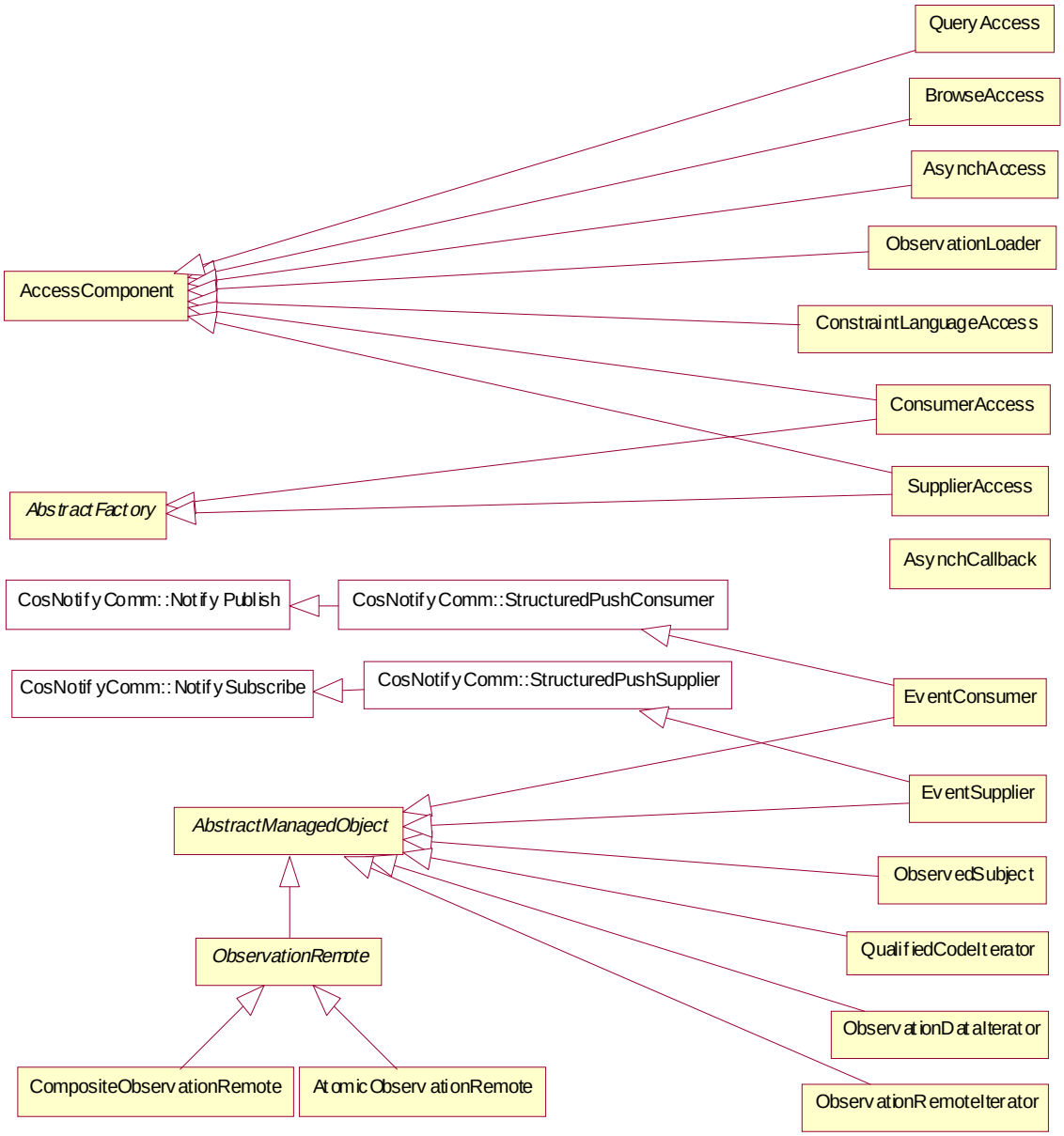

Figure 4-77 Inheritance relationship between the various **DsObservationAccess** interfaces.

This diagram shows the inheritance relationship between the **DsObservationAccess** service interfaces. The attributes and operations are hidden in this diagram in order to focus in on the inheritance relationships.

AccessComponent is the superclass for componentization (See Section 4.4.1.3, “Componentization Viewpoint,” on page 4-56).

The four interfaces from the **CosEvent** module are part of the OMG Event Service. The Event Service is not required for the **DsObservationAccess** event system, although its use is facilitated by the use of some common interfaces.

The **AbstractManagedObject** interface contains a single operation, **done()**, which allows a client to indicate when it is done with an object. All subclasses of **AbstractManagedObject** are instantiated or activated according to client requests, with their lifetime under server control. A well-behaved client will signal when it is done with such a remote object, and a savvy server will keep some timer for cleanup after ill-behaved clients or traumatic client termination.

The **ObservationRemote** object can have subtypes that are either composite or atomic observations. See Section 4.4.1.5, “Local/Remote Observations Viewpoint,” on page 4-58 for more details.

4.4.1.3 Componentization Viewpoint

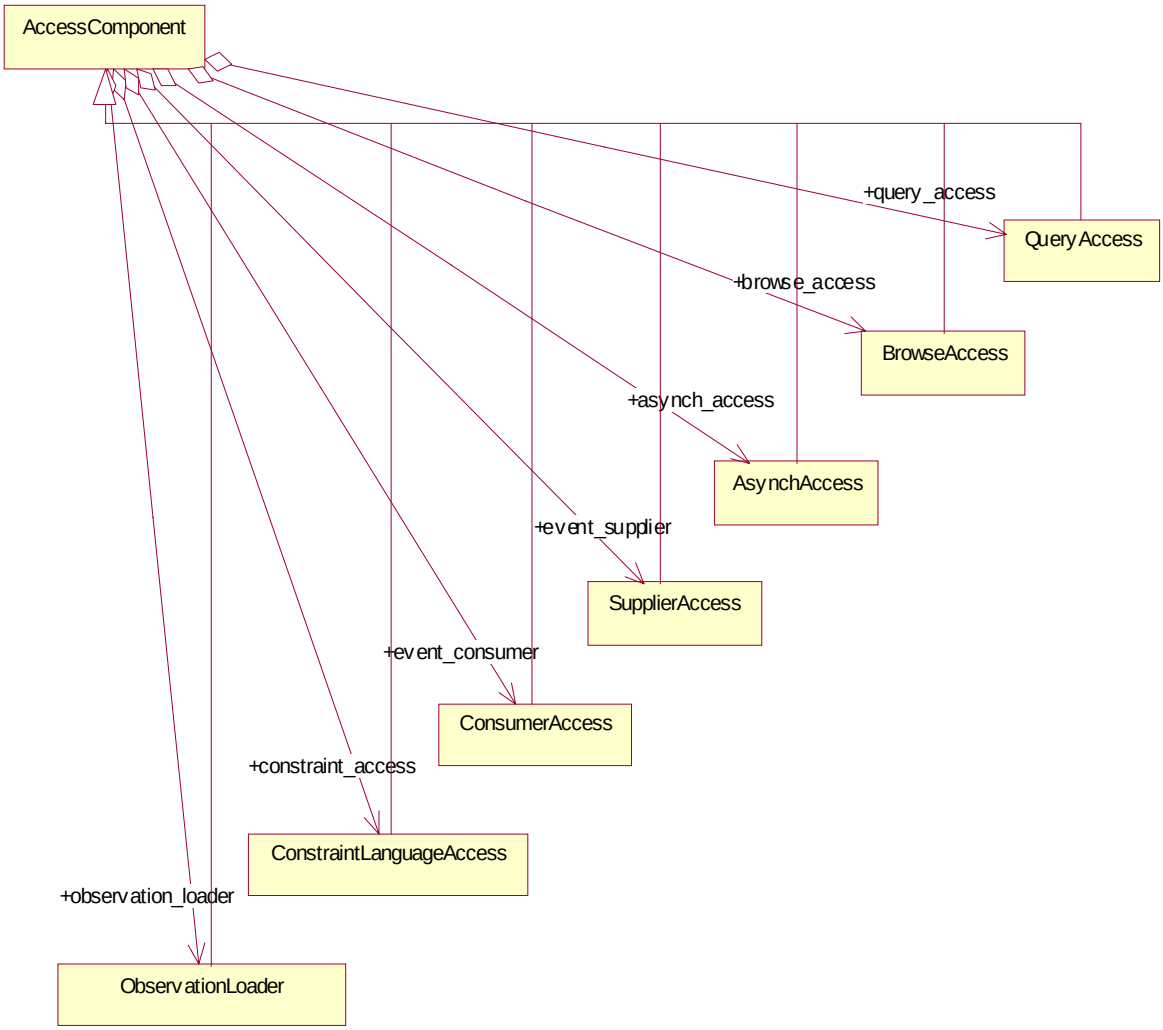

Figure 4-78 Simplified view of **ObservationAccess::AccessComponent** and its subclasses.

The base interface **AccessComponent** includes a means for dynamic discovery of all implemented components. Servers need implement only components which fit their purpose, according to conformance level.

The components each inherit from the **AccessComponent**, which in turn has references to other components, so a client of one component can navigate from one to another easily.

4.4.1.4 Full Component Viewpoint

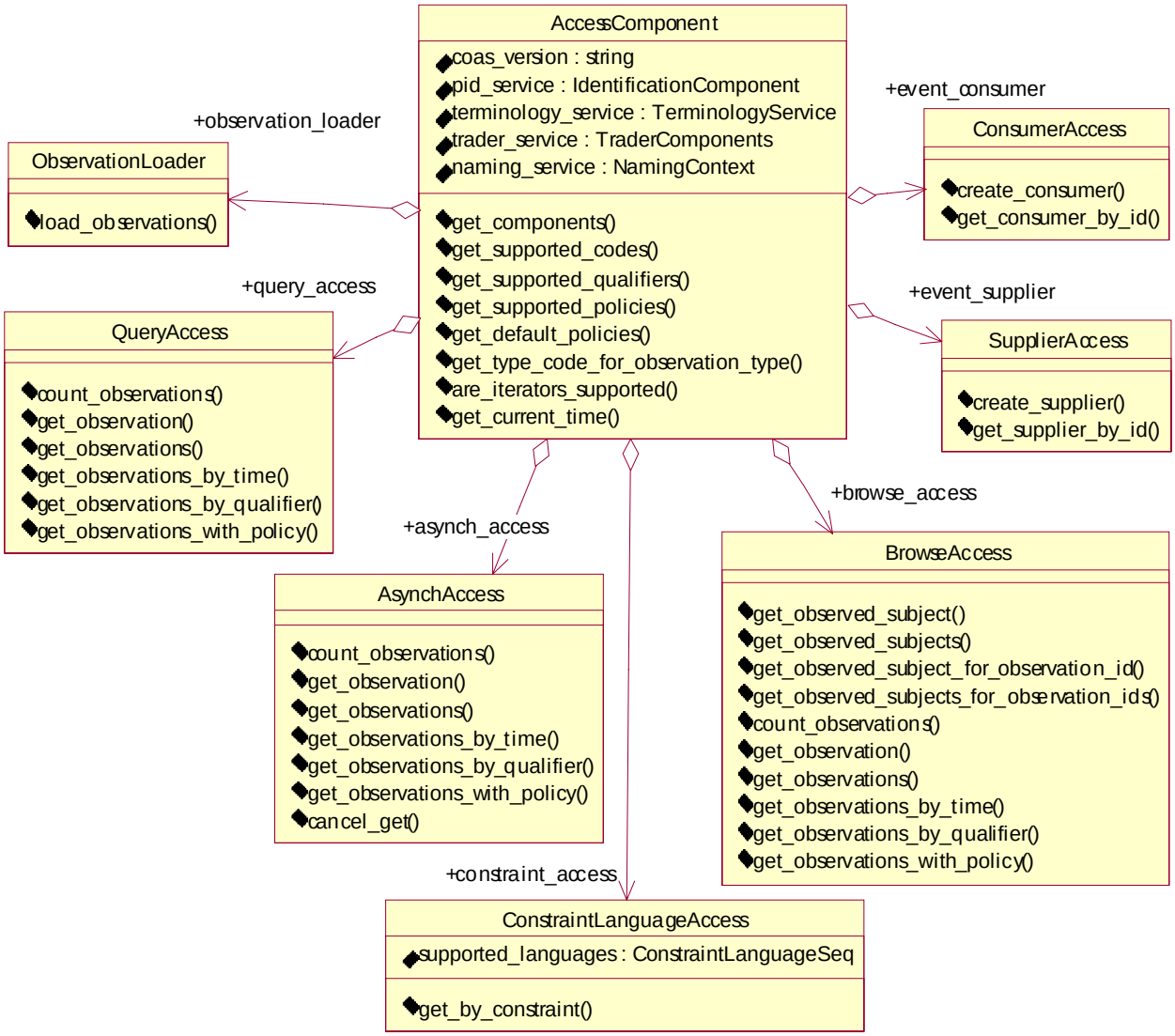

Figure 4-79 Full view of attributes and operations for **AccessComponent** and its subclasses.

The diagram above shows the components available from **AccessComponent**, and their attributes and operations. Several of the components share operations with similar names like "**get_observation()**", with similar semantics.

4.4.1.5 Local/Remote Observations Viewpoint

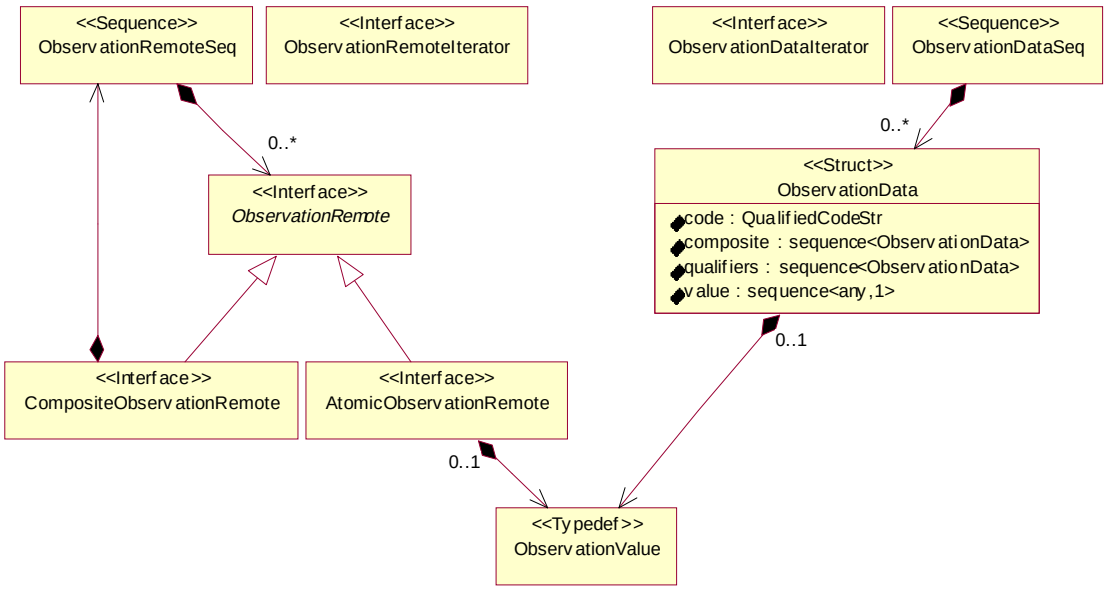

Figure 4-80 Showing a comparison between observations accessed by reference (remote) and observations accessed by value (local).

The **DsObservationAccess** service can support both reference and value access to observations. This viewpoint shows a comparison between observations returned by value (**ObservationData**) and those returned by reference (**ObservationRemote**). In both the local and remote flavors, only an “atomic” observation has an actual value, while a “composite” observation is a collection of other observations.

The division of observations into composite or atomic observations is accomplished differently for local access vs. remote access. The abstract interface **ObservationRemote** has concrete subclasses, so an **ObservationRemote** is either atomic or composite, with no possible ambiguity. However, **ObservationData** is a struct, with the potential to hold both values and an aggregation of other observations. Although the potential for ambiguity exists, there is a semantic requirement that each **ObservationData** be either atomic (have a value) or composite (have a non-zero aggregation of other observations), but not both at the same time.

Qualifiers which modify the observation(s) are available via **ObservationRemote.get_all_qualifiers()** or the “qualifiers” attribute on **ObservationData**.

Qualifiers modify all of the data “beneath” them in a hierarchy. For example, a modifier of “Normalcy=abnormal” found in a composite observation would apply to all the items in the composite. However, qualifiers found lower in a tree of data can override modifiers found higher up in the tree, so a leaf observation could have a modifier “Normalcy=normal” which applied to just that leaf, despite any qualifier higher-up in the tree.

See Section 4.4.1.7, “Remote Observations Viewpoint,” on page 4-60 for more detailed information about remote (by reference) observations.

See Section 4.4.1.6, “Local Observations Viewpoint,” on page 4-59 for more detailed information about local (by value) observations.

4.4.1.6 Local Observations Viewpoint

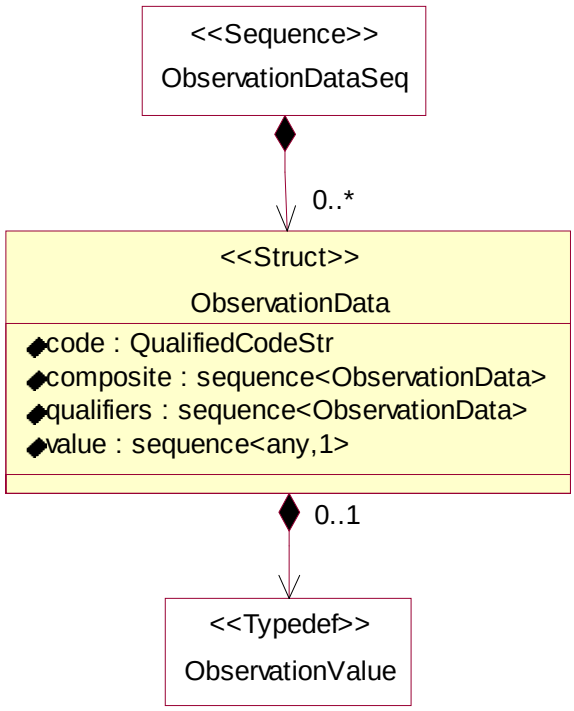

Figure 4-81 Detail UML for **ObservationData**.

ObservationData is the struct for passing “local” observations between client and server by value. Since **DsObservationAccess** does not use Object-by-Value (OBV), and structs have no polymorphism, the struct used for observations must encapsulate both composite observations and atomic observations. A composite observation will

have a non-zero amount of items in the **composite** attribute, and zero items in the **value** attribute. Conversely, an atomic local observation will have zero items in the **composite** attribute, and a single item in the **value** attribute.

See Section 4.4.2.6, “ObservationData,” on page 4-71 for the more details.

4.4.1.7 *Remote Observations Viewpoint*

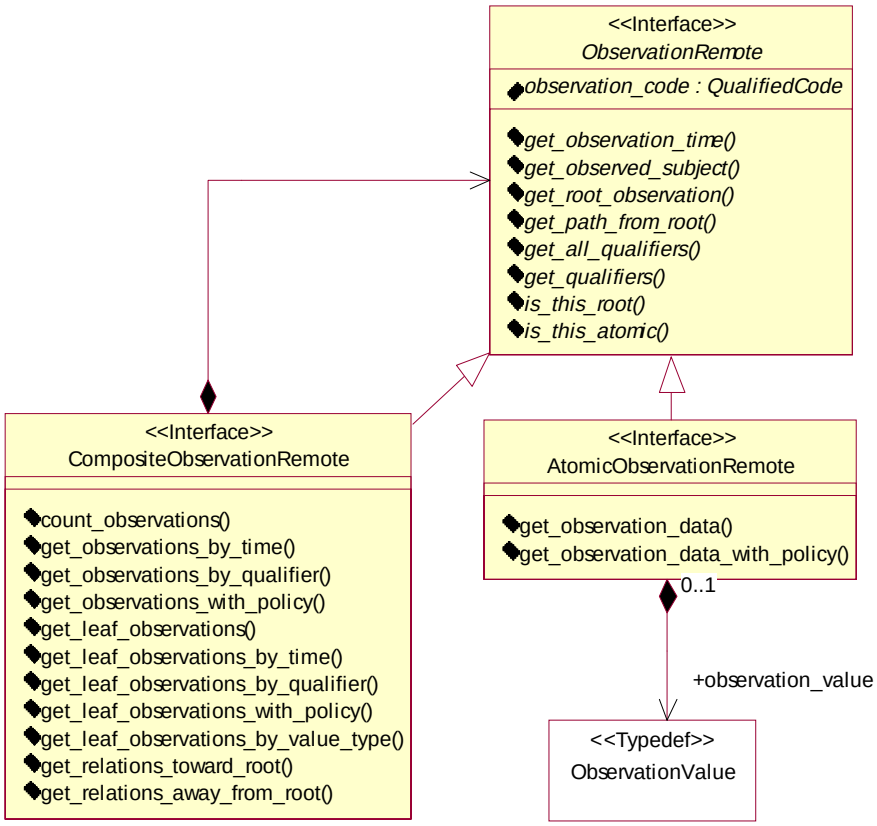

Figure 4-82 The operations and attributes for **ObservationRemote** and its subclasses.

ObservationRemote encapsulates remote references for observations. A remote observation is either a composite observation or an atomic observation. A composite observation aggregates a set of observations, like a set of lab values, each of which is an atomic observation, with a single data value.

See Section 4.4.1.10, “Browsing Access Viewpoint,” on page 4-63 for more information about the remote browsing style of access.

See Section 4.4.3.1, “ObservationRemote Interface,” on page 4-82 for the interface specification.

See Section 4.4.3.3, “CompositeObservationRemote Interface,” on page 4-84 for the type specification.

See Section 4.4.3.2, “AtomicObservationRemote Interface,” on page 4-83 for the type specification.

4.4.1.8 Common Access Operations Viewpoint

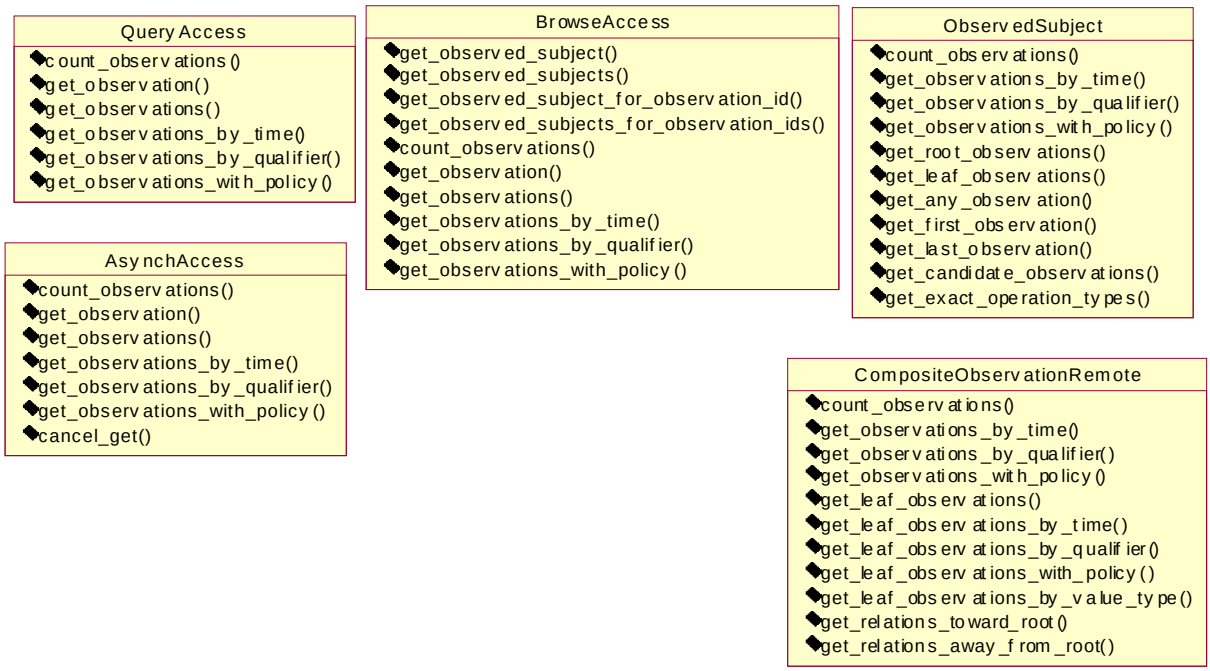

Figure 4-83 Common “**get_*()**” style operations on multiple interfaces.

This viewpoint shows that many interfaces have common operation names. A similar operation name implies similar semantics for the operation, though the return value may be local (**QueryAccess**), remote (**BrowseAccess**), or arriving asynchronously (**AsynchAccess**).

See the following for some of the different styles of access:

- Section 4.4.4.2, “QueryAccess Interface,” on page 4-95
- Section 4.4.1.10, “Browsing Access Viewpoint,” on page 4-63
- Section 4.4.1.11, “Asynchronous Access Viewpoint,” on page 4-64

4.4.1.9 Simple Query Access Viewpoint

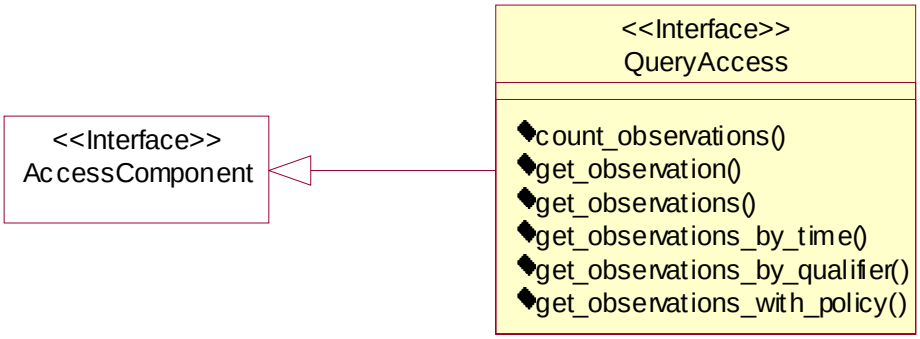

Figure 4-84 The **QueryAccess** interface is the simplest interface for query access.

QueryAccess is the most straightforward and fundamental of all the components. The client passes a query to the server and receives a response synchronously, as a local struct. The client blocks until the server returns the results or throws an exception.

QueryAccess has operations which provide a growing list of parameters for filtering the observations known by the server.

See Section 4.4.4.2, “QueryAccess Interface,” on page 4-95 for a detailed specification of the interface.

4.4.1.10 Browsing Access Viewpoint

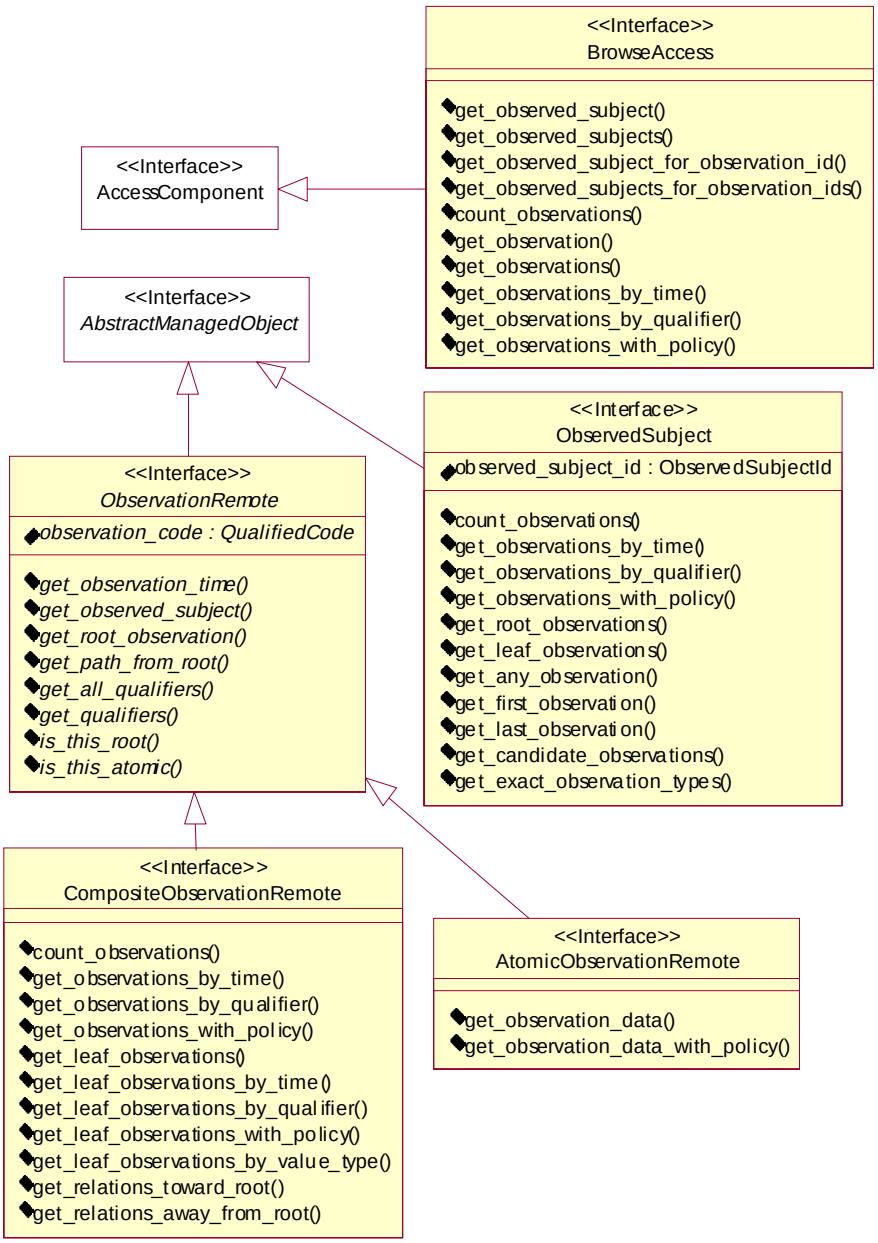

Figure 4-85 The main interfaces involved with browsing.

BrowseAccess makes use of remote proxies to explore the servers store of observations. A client can interactively access information a piece at a time. Each piece of information retrieved has links to other pieces of information that the client may access, with other queries possible based on the context of the previous requests. The server is required to keep context on the references passed back for this navigational convenience.

Interactive access may be useful when the client program displays the results and capabilities to the user after each command. A minimum of information has to be passed between the client and server with each action, although this mechanism adds responsibility to the server to maintain the lifecycle of a potentially large number of objects.

BrowseAccess has a number of operations that return object references to a remote **ObservedSubject** or **ObservationRemote**.

See Section 4.4.4.1, “BrowseAccess Interface,” on page 4-92 for a detailed description of this interface.

The ObservedSubject interface encapsulates the set of observations about one observed subject, typically a person, though a subject could be a tissue sample or an animal in a veterinary setting.

See Section 4.4.3.5, “ObservedSubject Interface,” on page 4-88 for a detailed description of this interface.

4.4.1.11 Asynchronous Access Viewpoint

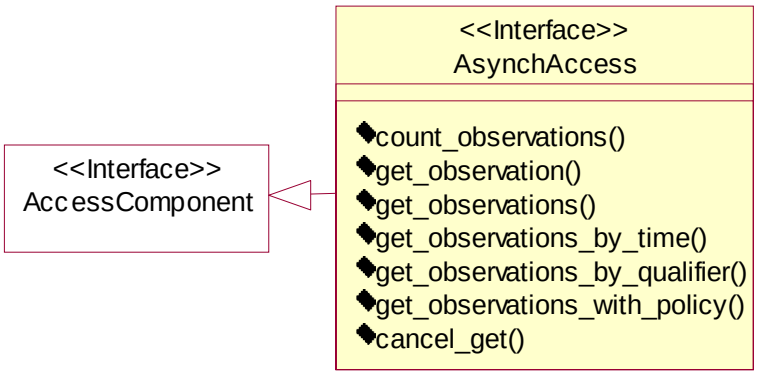

Figure 4-86 The interfaces dealing with asynchronous query invocations.

AsynchAccess allows a client to request information with the results delivered asynchronously. This prevents the client from having to do a blocking call to the server until the results can be returned. Asynchronous access may have various uses:

- Partial results: an asynchronous interface can return a result in pieces. This may be useful for something like image sets, to show the first one while receiving the rest, as well as for federation (send results back as they are received from various sources).
- Single-threaded clients: A single-thread GUI client could, for example, tend to repaint and user-click responsibilities while asynchronous requests are outstanding.
- Multiple requests: a client can post several simultaneous requests and process results in the order they are received, rather than proceeding serially from one to the next. Without this, results from a fast server could, in effect, wait on results from a slow server.
- Query portability between servers: an asynchronous request can be passed from one server to another, which responds directly to the client.
- Asynchronous model: for servers that get their data from asynchronous processes, an asynchronous mechanism may be the best fit. For example, DICOM can involve response times of millisecond to milli-decade (if the media is off-line), so a **DsObservationAccess** server which provides this data may want to provide it asynchronously, to match the source.

AsynchAccess affords asynchronous posting of results because the client passes in its own object reference to an **AsynchCallback** object. This points up some potential drawbacks to asynchronous access:

- Firewalls: a client behind a firewall may not be able to receive the callback.
- The client can no longer rely on TCP-level time-outs which bound a query duration for a synchronous call. Instead, the client must take responsibility to track outstanding requests and provide some ability to handle requests that fail because of a network outage or some other traumatic termination.
- If multiple requests are outstanding, the client must hold the state (**ClientCallId**) requests in order to identify them when fulfilled.
- The client must be prepared for multiple, partial returns to a single request.

The **AsynchAccess** interface has operations similar to the **QueryAccess** synchronous interface, though instead of “real” return values, the operations all return an **ServerCallId** value which simply identifies the request from the server point of view. **AsynchAccess** also has an operation to cancel an outstanding request. See Section 4.4.4.3, “AsynchAccess Interface,” on page 4-98 for a detailed description of these operations.

The **AsynchCallback** interface is implemented by the client to the **DsObservationAccess** server. The server calls it back with the results, or with an exception condition. See Section 4.4.4.4, “AsynchCallback Interface,” on page 4-100 for a detailed description of the interface.

4.4.1.12 Event Management Viewpoint

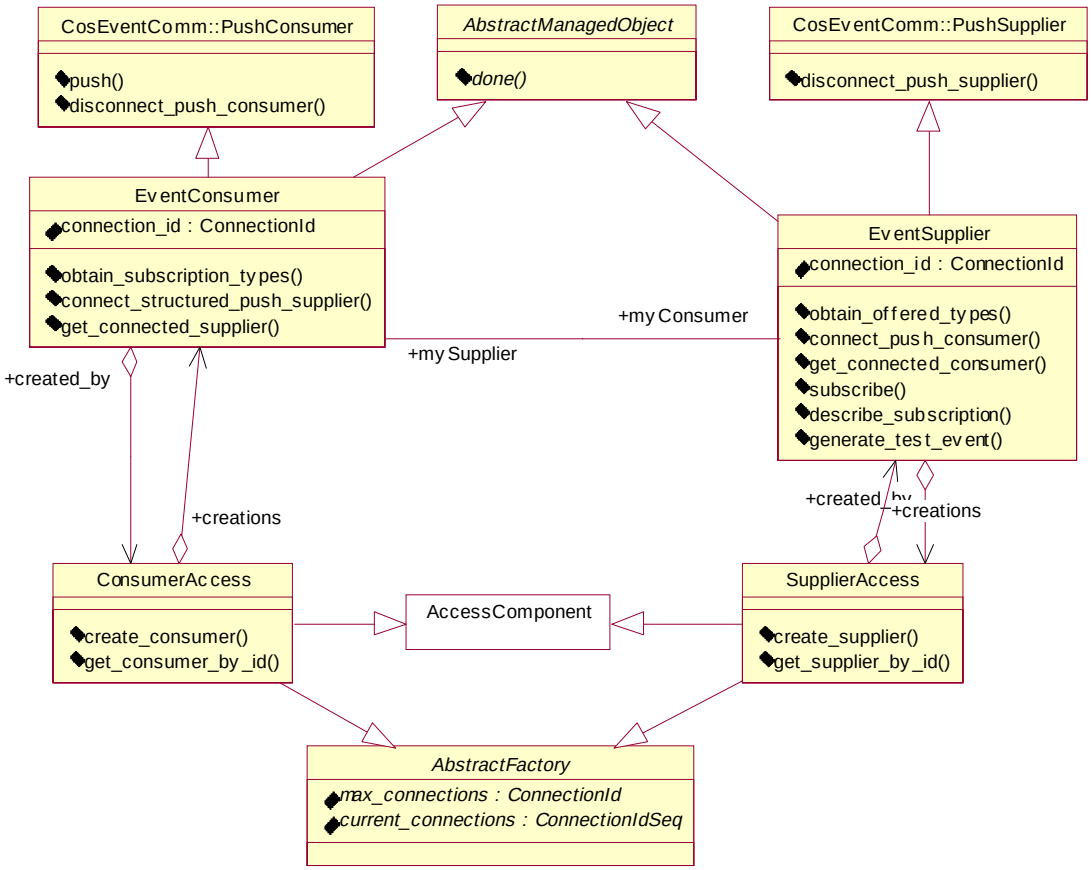

Figure 4-87 The consumer and supplier interfaces involved with event management.

The **DsObservationAccess** service supports querying for observations that occur in the future. This support is similar to asynchronous access in that a client (consumer) registers an interest in particular observations, and the server (supplier) calls them back with the information at some future time. However, the callback may happen repeatedly since the interest in particular observations translates into a subscription which lasts at least as long as the lifetime of the **EventSupplier**. Servers may add value (not required or specified herein) by offering a subscription qualifier for a persistent subscription, which survives across client and/or server restarts.

ConsumerAccess and **SupplierAccess** are the components that manage the registration to consume and supply future observations, respectively. The event mechanism was designed to give flexibility in connecting up event endpoints, including consideration to do the following:

- Facilitate the use of the OMG Notification Service or Event Service as an intermediary channel.
- Allow consumer and supplier endpoints to connect themselves to one another, without an intermediary channel.
- Allow the use of a external management application to connect consumer/supplier endpoints and channels, without explicit custom-coding assistance from the endpoints or channels for such an external management application.

The **AbstractFactory** interface contains two common attributes for connections (maximum and current amount) which are needed by both the **ConsumerAccess** and **SupplierAccess**. See Section 4.4.6.5, “AbstractFactory Interface,” on page 4-111 for the details.

The **ConsumerAccess** adds an operation to instantiate an **EventConsumer** and to access any formerly-created **EventConsumer** by its ID number, a unique number determined by the **ConsumerAccess** at instantiation. See Section 4.4.5.4, “ConsumerAccess Interface,” on page 4-105 for a detailed description of this interface.

The **SupplierAccess** extends the capability of the **AbstractFactory** just as did **ConsumerAccess**. See Section 4.4.5.3, “SupplierAccess Interface,” on page 4-105 for a detailed description of this interface.

The **EventConsumer** interface inherits from the **CosEventComm::PushConsumer** interface to facilitate use with the OMG **CosEvent** Service. See Section 4.4.5.2, “EventConsumer Interface,” on page 4-103 for a detailed description of the interface.

The **EventSupplier** interface inherits from the **CosEventComm::PushSupplier** interface to facilitate use with the OMG **CosEvent** Service. The **EventSupplier** includes operations to establish a connection, and to begin a subscription to events. See Section 4.4.5.1, “EventSupplier Interface,” on page 4-102 for a detailed description of the interface.

4.4.2 Data Type Definitions

The following sections describe all the IDL for the data types used within the **DsObservationAccess** module.

```
#ifndef _DS_OBSERVATION_ACCESS_IDL_
#define _DS_OBSERVATION_ACCESS_IDL_


...
module DsObservationAccess {
    ...

};

#endif  // _DS_OBSERVATION_ACCESS_IDL_
```

The “Ds” prefix of **DsObservationAccess** stands for “Domain Service.” All OMG specifications from a domain task force are expected to start with “Ds” to isolate a particular name space from potential clashes.

4.4.2.1 Include Files

```
#include <CosNaming.idl>
#include <CosTrading.idl>
#include <TerminologyServices.idl>
#include <NamingAuthority.idl>
#include <PersonIdService.idl>
#include <NamingAuthority.idl>
#include <CosEventComm.idl>
#include <CosEventChannelAdmin.idl>
#include <orb.idl>
```

4.4.2.2 External Typedefs

These definitions rename types from other standards. This section delineates all **DsObservationAccess** dependencies on other standards.

typedef PersonIdService::QualifiedPersonId ObservedSubjectId;

Observed subjects are identified with a **QualifiedPersonId** from the PIDS standard. The qualification with a naming authority is important, since there could be overlap in patient identifiers at two locations.

typedef TerminologyServices::QualifiedCode QualifiedCode;

A **QualifiedCode** has an embedded **NamingAuthority** which prevents collisions between common, local names.

typedef NamingAuthority::QualifiedNameStr QualifiedCodeStr;

QualifiedCodeStr has a one-to-one mapping with **QualifiedCode**. The format for the contenst of **QualifiedNameStr** is well defined. Strings must begin with a colon-delimited section containing one of the **NamingAuthority::RegistrationAuthority** items: either OTHER, ISO, DNS, IDL, or DCE. Following the **RegistrationAuthority** is a domain, followed by a slash “/”, and then the particular name (which can have additional slashes as namespace dividers).

For example, the **QualifiedCodeStr** “**DNS:omg.org/DsObservationAccess/ASYNC_OBSERVATION_COUNT**” has a registration authority of DNS (internet domain name service), a domain of omg.org, and a name within the **DsObservationAccess** namespace.

The **NamingAuthority::translation_library** interface is designed to be implemented locally by servers to translate between **QualifiedName** (we rename as **QualifiedCode**) and **QualifiedNameStr** (we call this **QualifiedCodeStr**).

typedef PersonIdService::DomainName IdDomainName;

Each COAS server will have one default PIDS domain which is identified by a **DomainName**.

typedef PersonIdService::IdentificationComponent IdentificationComponent;

The PIDS server is an instance of an **IdentificationComponent**.

typedef CosNaming::NamingContext NamingContext;

The relevant **CosNaming** server is an instance of a **NamingContext**.

typedef CosTrading::TraderComponents TraderComponents;

The relevant Trader service is an instance of a **TraderComponents**.

typedef TerminologyServices::TerminologyService TerminologyService;

The relevant **TerminologyService** is an instance of **TerminologyService**.

typedef CosEventComm::PushConsumer PushConsumer;

The **EventConsumer** is a subclass of **CosEventComm::PushConsumer**.

typedef CosEventComm::PushSupplier PushSupplier;

The **EventSupplier** is a subclass of **CosEventComm::PushSupplier**.

typedef CORBA::TypeCode TypeCode;

A **TypeCode** is a CORBA interface that is used to perform introspection on all IDL-defined data types.

4.4.2.3 Forward Declarations

interface AbstractFactory;
interface AbstractManagedObject;
interface AccessComponent;
interface AsynchCallback;
interface AsynchAccess;
interface AtomicObservationRemote;
interface BrowseAccess;
interface CompositeObservationRemote;
interface ConsumerAccess;
interface ConstraintLanguageAccess;
interface EventConsumer;
interface EventSupplier;
interface ObservationDataIterator;
interface ObservationLoader;
interface ObservationRemote;
interface ObservationRemoteIterator;
interface ObservedSubject;

```
interface QualifiedCodeIterator;
interface QueryAccess;
interface SupplierAccess;
```

These forward declarations for interfaces facilitates the grouping of definitions without concern for precedence, since all interfaces are declared here.

4.4.2.4 *AccessComponentData*

```
struct AccessComponentData {

    QueryAccess query_access;

    BrowseAccess browse_access;

    AsynchAccess asynch_access;

    ConstraintLanguageAccess constraint_access;

    ObservationLoader observation_loader;

    ConsumerAccess consumer_access;

    SupplierAccess supplier_access;

};
```

AccessComponentData provides a means to supply references to all implemented components via **AccessComponent.get_components()**. This is a convenience for clients that have a single reference to a single component, and wish to use a different component. Since different servers may have different levels of conformity, some will implement a given component and others will not. If a component is not implemented by the server, that attribute will be null.

For example, if a client has a reference to a **BrowseAccess** component, and now wishes to use a **QueryAccess** component, the client can call **get_components()** on his **BrowseAccess** component and examine the **query_access** field. If **query_access** is non-null, that component is implemented.

query_access

Holds **QueryAccess** reference if implemented by this server.

browse_access

Holds **BrowseAccess** reference if implemented by this server.

asynch_access

Holds **AsynchAccess** reference if implemented by this server.

constraint_access

Holds **ConstraintLanguageAccess** reference if implemented by this server.

observation_loader

Holds **ObservationLoader** reference if implemented by this server.

consumer_access

Holds **ConsumerAccess** reference if implemented by this server.

supplier_access

Holds **SupplierAccess** reference if implemented by this server.

4.4.2.5 AsynchException

```
struct AsynchException {
    QualifiedCodeStr exception_name;
    string message;
};
```

AsynchException is a struct because the asynchronous callback mechanism cannot employ the typical exception mechanism of CORBA synchronous call. Instead, a request which results in an exception must be delivered to the **AsynchCallback** interface, just as a regular result is delivered, with a struct.

exception_name

The name of the exception resulting from the asynchronous request

message

A text description of the exception.

4.4.2.6 ObservationData

```
struct ObservationData {
    QualifiedCodeStr code;
    sequence<ObservationData> composite;
    sequence<ObservationData> qualifiers;
    sequence<any,1> value;
};
```

ObservationData is the heart of the query mechanism. **ObservationData** encapsulates both composite and atomic observations, which is accomplished by including attributes for both an aggregation and a single value. These attributes, **composite** and **value**, are intended to be used in a mutually exclusive manner. One of the two attributes should be a zero-length sequence. An Observation must be a composite observation or an atomic observation, but not both.

code

The name of the observation type, as qualified by the **NamingAuthority** embedded in the **QualifiedCodeStr**.

composite

A sequence of observations which compose this observation. The attribute **composite** may have zero or more **ObservationData** items. The **composite** attribute must have zero items if this observation has a non-zero **value** attribute, which would make it an atomic, rather than composite, observation.

Note that each of the aggregated **ObservationData** items may, in turn, include other observations in their **composite** field, creating a “tree” of observations.

qualifiers

A sequence of observations that modify the observation(s) in the **value** or **composite** attribute. Qualifiers modify all of the data “beneath” them in a hierarchy. For example, a modifier of “Normalcy=abnormal” found in a composite observation would apply to all the items in the composite. However, qualifiers found lower in a tree of data can override modifiers found higher up in the tree, so a leaf observation could have a modifier “Normalcy=normal” which applied to just that leaf, despite any qualifier higher-up in the tree.

value

The payload for this observation. The payload must be empty (zero items in sequence) if this observation is a composite observation. The only reason that **value** is a sequence is to allow a zero-length sequence.

For an atomic observation, which has a payload, the contents within **value[0]**, within the **CORBA::any**, is a data type which associates with the “code” field. For each code used for an atomic observation, a single data type must be designated for the return value.

4.4.2.7 ObservationQualifier

typedef ObservationData ObservationQualifier;

This typedef shows that Qualifiers are simply other observations.

4.4.2.8 *ObservationId*

```
struct ObservationId {

   QualifiedCodeStr code;

   string opaque;
};
```

An **ObservationId** uniquely identifies a particular COAS observation within a server. It is persistent over time, and can be stored by a client for use later. However, a client may not create or modify an **ObservationId**.

The client is responsible for remembering the server associated with a given **ObservationId**. If the client connects to multiple servers, the client can, for example, keep all **ObservationId**s from a particular server in a single collection associated with the server, or store an **ObservationId** within some wrapping structure which provided fields for server identification as well.

There has been discussion of adding fields to **ObservationId** for a server name and domain. Currently, there is no for provision for the globally identifying server names in some federation of COAS servers, so it is not clear what would be appropriate for server identification field(s).

One possibility for handling **ObservationId**s within a federation of COAS servers can be implemented as follows:

Assume a federation of COAS servers where a higher-level server named “Middle” is a middleware conduit for some (static) group of lower-level COAS servers. All queries to Middle are routed to one of many lower-level COAS servers, and the resulting information is passed back to the client, including qualifiers like **ObservationId**s. However, when supplying these **ObservationId**s to its client, Middle must modify them slightly. The **ObservationId**s must allow Middle to recognize the original source for the observation. To accomplish this, Middle can prepend source-server information to the opaque string, followed by a clear delimiter. Upon receipt of the **ObservationId** from a client, Middle strips out this source-server information, using it to pass back a reconstituted **ObservationId** to the proper source server.

code

The code for this observation. This is read-only for a client, and can be used for grouping or separating **ObservationId**s.

opaque

Reserved for use by server.

4.4.2.9 *NameValuePair*

```
struct NameValuePair {

   QualifiedCodeStr name;
```

```
    any value;
};
```

A simple associate of name and value.

name

The code for this pair.

value

The value for this pair.

4.4.2.10 *Subscription*

```
struct Subscription {
    sequence<ObservedSubjectId> who;
    sequence<QualifiedCodeStr> what;
    sequence<ObservationQualifier> qualifier;
    sequence<NameValuePair> policy;
};
```

Subscription encapsulates all the parameters which make up a query for future data, as needed for a **SupplierAccess** component.

who

The observed subject(s) of the subscription.

what

The codes for the desired observation(s).

qualifier

Any modifying observation(s) with which to filter.

policy

Any policies that should override default policies of the server.

4.4.2.11 TimeStamp

typedef string TimeStamp; // ISO 8601 representation, with restrictions

TimeStamp is a string representation of date and time, following the ISO 8601:1988 standard, with some restrictions and modifications. The string format is restricted to the "extended" ISO 8601 format which includes delimiters, years must be specified with century digits, and a wildcard character is added. A complete TimeStamp format is:

YYYY-MM-DDThh:mm:ss.dddTZD

(e.g., 1997-07-16T19:20:30.45+01:00) where:

YYYY = four-digit year (1582 minimum, 9999 maximum)
MM = two-digit month (01=January, etc.)
DD = two-digit day of month (01 through 31)
T = date/time separator
hh = two digits of hour (00 through 23; am/pm NOT allowed)
mm = two digits of minute (00 through 59)
ss = two digits of second (00 through 60; 60 indicates a positive leap second)
ddd = one or more digits for decimal fraction of a second (no limit on number of digits)
TZD = time zone designator (Z to indicate UTC, or +hh:mm or -hh:mm from UTC)

Partial **TimeStamp** formats are allowed, which indicate "unknown" for items omitted. For example, a **TimeStamp** consisting only of

1993-02-14

is interpreted as an unknown time on the 14th of February, 1993, while

13:10:30

is interpreted as an unknown date, with time of 13:10:30 in the server's time zone (absence of a time zone designator indicates local time).

TimeStamp allows a character outside the ISO 8601 specification, a wildcard for individual **TimeStamp** elements. **TIME_WILDCARD** = "?" is provided in the constants section. Use this character to indicate that a specific field should be treated as "unknown" for **TimeStamp**s received from COAS (output), and as a "wildcard" for **TimeStamp** parameters supplied to COAS (input).

For example, receiving "1999-??-02T22:00:00Z" as an output **TimeStamp** would be equivalent to the concept of "2nd day of an unknown month in 1999, at 22:00:00 GMT". For an input **TimeStamp**, this string would represent, for matching purposes, "the 2nd day of any month in 1999, at 22:00:00 GMT".

The lower bound for **TimeStamp** is specified as "1582-10-15T00:00:00Z", the date when the Gregorian calendar was put into effect, putting month and day calculations on a firm basis.

4.4.2.12 TimeSpan

struct TimeSpan {

```
    TimeStamp start_time;

    TimeStamp stop_time;

};
```

TimeSpan encapsulates a duration of time with two bounding **TimeStamp**s. The semantics for interpreting the endpoints is INCLUSIVE. The endpoints are part of, included in, the span of time. This span is defined for use in COAS instead of employing the ISO 8601 notation of <timestamp>/<timestamp> within one string.

start_time

The starting time of the span.

stop_time

The ending time of the span.

4.4.2.13 Constants

```
const string EARLIEST_TIME   = "1582-10-15T00:00:00Z";
const string LATEST_TIME     = "9999-12-31T23:59:59Z";
const string TIME_WILDCARD = "?";

const QualifiedCodeStr PARTIAL_RESULT =
    "DNS:omg.org/DsObservationAccess/PARTIAL_RESULT";
const QualifiedCodeStr COMPLETING_RESULT =
    "DNS:omg.org/DsObservationAccess/COMPLETING_RESULT";
```

COMPLETING_RESULT and **PARTIAL_RESULT** are used by the **AsynchAccess** interface during a callback to indicate the status of the callback--completing a request, or only partially completing a request.

```
const QualifiedCodeStr ASYNC_OBSERVATION_COUNT =
    "DNS:omg.org/DsObservationAccess/ASYNC_OBSERVATION_COUNT";
typedef unsigned long ASYNC_OBSERVATION_COUNT_type;
```

ASYNC_OBSERVATION_COUNT is an observation type, used solely to identify the return value of the operation **AsynchAccess.count_observations()**. It does not make sense to use this code in a query, since **count_observations()** explicitly names the "what" part of the query parameters. Only the return value needs identification. The value in that returned **ObservationData** is an **unsigned long**, and shown by the typedef **ASYNC_OBSERVATION_COUNT_type**.

```
const QualifiedCodeStr EVENT_SOURCE_DOMAIN  =
    "DNS:omg.org/DsObservationAccess/EVENT_SOURCE_DOMAIN";
const QualifiedCodeStr EVENT_SOURCE_SERVER_NAME =
    "DNS:omg.org/DsObservationAccess/EVENT_SOURCE_SERVER_NAME";
const QualifiedCodeStr EVENT_NAME  =
    "DNS:omg.org/DsObservationAccess/EVENT_NAME";
const QualifiedCodeStr TEST_EVENT   =
    "DNS:omg.org/DsObservationAccess/TEST_EVENT";
```

typedef long TEST_EVENT_type;

EVENT_* constants apply to the **SupplierAccess** component:

EVENT_SOURCE_DOMAIN: the enterprise domain (likely a PIDS context) within which the event originated.

EVENT_SOURCE_SERVER_NAME: the name of the **DsObservationAccess** service which originated the event.

EVENT_NAME: this code is intended for use when a **CosNotification** service is employed. The **CosNotification** service allows filtering within the channel, based on name-value pairs, so this code would be used to identify the name of the particular event, with a value equal to the **QualifiedCodeStr** of the event itself.

TEST_EVENT is the observation code used by the **SupplierAccess** when responding to **SupplierAccess.generate_test_event()**. The value returned in the **ObservationData** is a **long**, as shown by the typedef **TEST_EVENT_type**.

```
const QualifiedCodeStr TRADER_1_0_CONSTRAINT_LANGUAGE = “DNS:omg.org/DsObservationAccess/TRADER_1_0_CONSTRAINT_LANGUAGE”;
const QualifiedCodeStr OCL_1_1_CONSTRAINT_LANGUAGE  = “DNS:omg.org/DsObservationAccess/OCL_1_1_CONSTRAINT_LANGUAGE”;
```

TRADER_1_0_CONSTRAINT_LANGUAGE and **OCL_1_1_CONSTRAINT_LANGUAGE** are two possible choices for the language used by **ConstraintLanguageAccess**. However, the choice of constraint language is left to the implementation.

```
const QualifiedCodeStr COAS_OBSERVATION_ID  = “DNS:omg.org/DsObservationAccess/COAS_OBSERVATION_ID”;
typedef ObservationId COAS_OBSERVATION_ID_type;
```

COAS_OBSERVATION_ID is the code for a qualifier which provides a unique COAS identifier for an observation. Any qualifier with this code will have, in its value **CORBA::any**, a struct of type **ObservationId**, as indicated by **COAS_OBSERVATION_ID_type**. In other words, the one-to-one association between a name-value pair are, in this instance, **COAS_OBSERVATION_ID** and **ObservationId**.

4.4.2.14 Internal Typedefs

typedef long EndpointId;

EndpointId is used by the Event system, **ConsumerAccess** and **SupplierAccess**, to identify event endpoints.

typedef string ConstraintExpression;

ConstraintExpression is used to supply a constraint to **ConstraintLanguageAccess**.

typedef QualifiedCodeStr ConstraintLanguage;

ConstraintLanguage is specified by the **ConstraintLanguageAccess**, as a language supported by that component.

typedef NameValuePair QueryPolicy;

Each policy is a name-value pair.

typedef long ServerCallId;

Within the **AsynchAccess**, each call from a client is identified by the server with a **ServerCallId**, unique within the lifetime of the server. This identifier can be used to cancel the request if necessary.

typedef long ClientCallId;

A client to the **AsynchAccess** should identify each of its calls to a server with a **ClientCallId**, unique within all outstanding requests. This identifier is returned to the client with the result, so that the client can match up requests with responses, should there be more than one call outstanding.

4.4.2.15 Sequences

typedef sequence<AtomicObservationRemote> AtomicObsRemoteSeq;

typedef sequence<ConstraintLanguage> ConstraintLanguageSeq;

typedef sequence<EndpointId> EndpointIdSeq;

typedef sequence<ObservationData> ObservationDataSeq;

typedef sequence<ObservationId> ObservationIdSeq;

typedef sequence<ObservationQualifier> ObservationQualifierSeq;

typedef sequence<ObservationRemote> ObservationRemoteSeq;

typedef sequence<ObservedSubjectId> ObservedSubjectIdSeq;

typedef sequence<ObservedSubject> ObservedSubjectSeq;

typedef sequence<QualifiedCodeStr> QualifiedCodeStrSeq;

typedef sequence<QueryPolicy> QueryPolicySeq;

typedef sequence<Subscription> SubscriptionSeq;

The above IDL defines the sequence data types for the **DsObservationAccess** service.

4.4.2.16 Exceptions

exception DuplicateCodes {

QualifiedCodeStrSeq codes;

};

The DuplicateCodes exception is raised when the same code is passed multiple times as a parameter to an operation. A complete list of distinct duplicated codes is returned.

```
exception DuplicateIds {

    ObservedSubjectIdSeq ids;

};
```

The DuplicateIds exception is raised when the same **ObservedSubjectId** is passed multiple times as a parameter to an operation. A complete list of distinct duplicated ids is returned.

```
exception DuplicateOids {

    ObservationIdSeq oids;

};
```

The DuplicateOids exception is raised when the same **ObservationId** is passed multiple times as a parameter to an operation. A complete list of distinct duplicated **ObservationIds** is returned.

```
exception DuplicatePolicies {

    QueryPolicySeq policies;

};
```

The DuplicatePolicies exception is raised when the same **QueryPolicy** is passed multiple times as a parameter to an operation. A complete list of distinct duplicated policies is returned.

```
exception DuplicateQualifiers {

    ObservationQualifierSeq qualifiers;

};
```

The DuplicateQualifiers exception is raised when the same **ObservationQualifierData** is passed multiple times as a parameter to an operation. A complete list of distinct duplicated qualifiers is returned.

```
exception InvalidCodes {

    QualifiedCodeStrSeq codes;

};
```

The InvalidCodes exception is raised when an unrecognized (unsupported) **QualifiedCodeStr** is passed as a parameter to an operation. A complete list of invalid codes is returned.

```
exception InvalidEndpointId {
```

```
    EndpointIdSeq endpoint_ids;

};
```

The InvalidEndpointId exception is raised when an invalid **EndpointId** is passed as a parameter to an operation. Only active connections may be specified. A complete list of invalid connection ids is returned.

```
exception InvalidConstraint {

    string constraint;

};
```

The InvalidConstraint exception is raised when a constraint is passed as a parameter to an operation and the server cannot parse the constraint in accordance with a supported language. The invalid constraint is returned.

```
exception InvalidIds {

    ObservedSubjectIdSeq ids;

};
```

The InvalidIds exception is raised when an **ObservedSubjectId** is passed as a parameter to an operation when the server does not know about that ID. A complete list of invalid ids is returned.

```
exception InvalidOids {

    ObservationIdSeq oids;

};
```

The InvalidOids exception is raised when a **ObservationId** is passed as a parameter to an operation when the server does not know about that observation ID. A complete list of invalid ids is returned.

```
exception InvalidPolicies {

    QualifiedCodeStrSeq policies;

};
```

The InvalidPolicies exception is raised when an unrecognized (unsupported) **QueryPolicy** is passed as a parameter to an operation. A complete list of invalid policies is returned.

```
exception InvalidQualifiers {

    QualifiedCodeStrSeq qualifiers;

};
```

The InvalidQualifiers exception is raised when an unrecognized (unsupported) **ObservationQualifierData** is passed as a parameter to an operation. A complete list of violating qualifiers is returned.

```
exception InvalidTimeSpan {

    TimeSpan span;

};
```

The InvalidTimeSpan exception is raised when an invalid **TimeSpan** is passed as a parameter to an operation. The time span may be incorrectly specified, with a start time greater than an ending time, or with unparsable items.

```
exception MaxConnectionsExceeded {

    unsigned long max_connections;

};
```

The MaxConnectionsExceeded exception is raised when an event access object (**EventSupplier** or **EventConsumer**) already has reached its maximum supported number of connections, and a client tries to create another one. The maximum number of connections is returned.

```
exception NotImplemented {

};
```

NotImplemented is raised when a particular COAS server does not implement a particular operation. This exception allows a conformance class to have optional operations. Any operation with this exception is optional.

```
exception NoSubscription {

};
```

The NoSubscription exception is raised trying to access subscription information on a **EventSupplier** when no subscription has been set.

4.4.3 Foundational Observation-Oriented Interface Specifications

The description of the **DsObservationAccess** interfaces begins with those that map most closely to the COAS Information Model, i.e. Observation-Oriented interfaces. They support the successive refinement and interactive browsing styles of data retrieval and data discovery.

4.4.3.1 ObservationRemote Interface

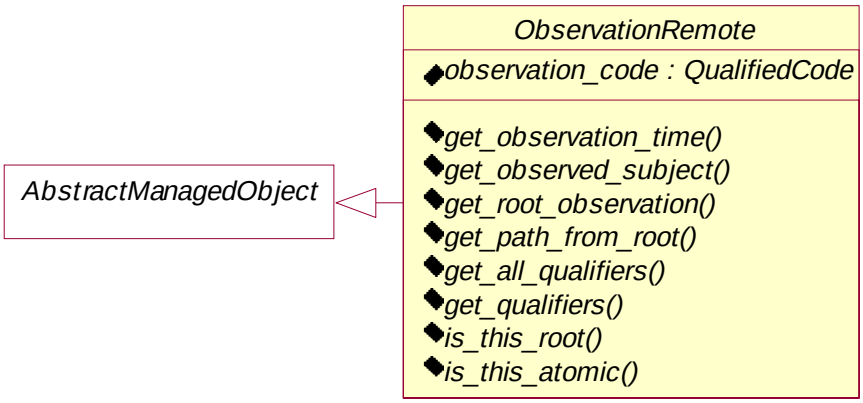

Figure 4-88 ObservationRemote Interface

```
interface ObservationRemote : AbstractManagedObject {

    readonly attribute QualifiedCodeStr observation_code;

    TimeSpan get_observation_time ();

    ObservedSubject get_observed_subject ();

    ObservationRemote get_root_observation ();

    ObservationData get_path_from_root ();

    ObservationQualifierSeq get_all_qualifiers ();

    ObservationQualifierSeq get_qualifiers (
            in QualifiedCodeStrSeq qualifier_names )
        raises (
            InvalidCodes );

    boolean is_this_root ();

    boolean is_this_atomic ();

};
```

observation_code

The code which identifies this observation.

get_observation_time()

Return the **TimeSpan** associated with this observation.

get_observed_subject()

Return a reference to the subject associated with this observation.

get_root_observation()

Return the root observation within which this observation is contained. If this observation is the root, returns reference to self. Server has responsibility to keep a context of all remote observations that are browsed, to keep track of their context.

get_path_from_root()

Return the root observation as an **ObservationData** containing the path elements to this observation. The **ObservationData** returned contains the structure of the real observation tree pruned of all observations that don't lead to this one.

get_all_qualifiers()

Return all qualifiers.

get_qualifiers()

Return the qualifier(s) specified by name in the input parameter **qualifier_names**.

is_this_root()

Returns true if this observation is a root observation.

is_this_atomic()

Returns true if this observation is actually a subclass, **AtomicObservationRemote**.

4.4.3.2 AtomicObservationRemote Interface

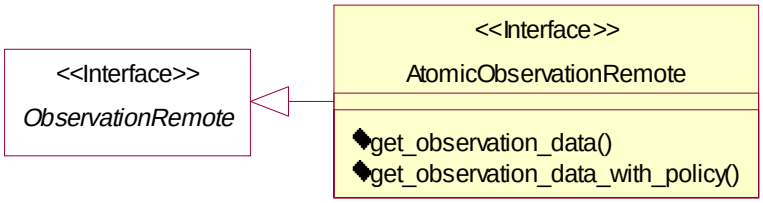

Figure 4-89 AtomicObservationRemote Interface

```
interface AtomicObservationRemote : ObservationRemote {

    ObservationData get_observation_data ();

    ObservationData get_observation_data_with_policy (
        in QueryPolicySeq policy );
```

};

get_observation_data()

Returns the (local) **ObservationData** item by value.

get_observation_data_with_policy()

Returns the (local) **ObservationData** item by value, according to overriding policies provided.

4.4.3.3 CompositeObservationRemote Interface

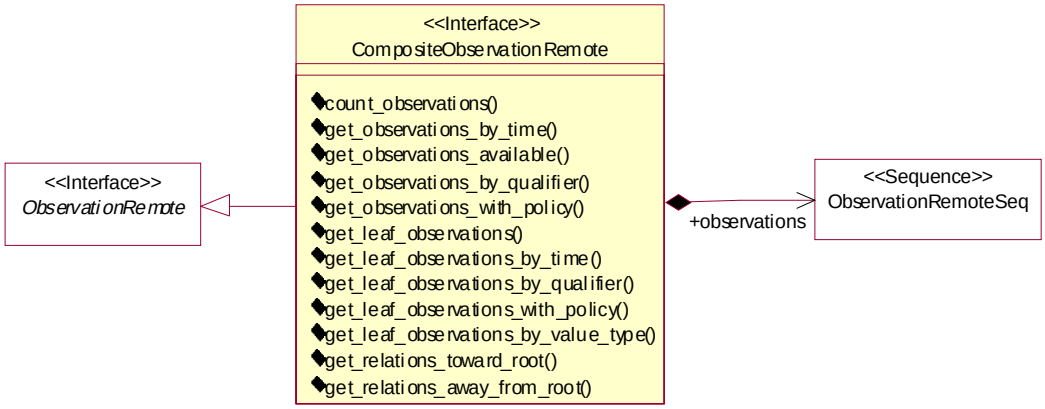

Figure 4-90 CompositeObservationRemote Interface

```
interface CompositeObservationRemote : ObservationRemote {

    unsigned long count_observations (
            in QueryPolicySeq search_depth_policy )
        raises (
            InvalidPolicies );

    ObservationRemoteSeq get_observations_by_time (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
        raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan );
```

```
ObservationRemoteSeq get_observations_by_qualifier (
        in QualifiedCodeStrSeq what,
        in TimeSpan when,
        in ObservationQualifierSeq qualifier,
        in unsigned long max_sequence,
        out ObservationRemoteIterator the_rest )
    raises (
        InvalidCodes,
        DuplicateCodes,
        InvalidTimeSpan,
        InvalidQualifiers,
        DuplicateQualifiers );

ObservationRemoteSeq get_observations_with_policy (
        in QualifiedCodeStrSeq what,
        in TimeSpan when,
        in ObservationQualifierSeq qualifier,
        in QueryPolicySeq policy,
        in unsigned long max_sequence,
        out ObservationRemoteIterator the_rest )
    raises (
        InvalidCodes,
        DuplicateCodes,
        InvalidTimeSpan,
        InvalidQualifiers,
        DuplicateQualifiers,
        InvalidPolicies,
        DuplicatePolicies );

AtomicObsRemoteSeq get_leaf_observations ();

AtomicObsRemoteSeq get_leaf_observations_by_time (
        in QualifiedCodeStrSeq what,
        in TimeSpan when,
        in unsigned long max_sequence,
        out ObservationRemoteIterator the_rest )
    raises (
        InvalidCodes,
        DuplicateCodes,
        InvalidTimeSpan );

AtomicObsRemoteSeq get_leaf_observations_by_qualifier (
        in QualifiedCodeStrSeq what,
        in TimeSpan when,
        in ObservationQualifierSeq qualifier,
        in unsigned long max_sequence,
        out ObservationRemoteIterator the_rest )
    raises (
        InvalidCodes,
        DuplicateCodes,
        InvalidTimeSpan,
        InvalidQualifiers,
        DuplicateQualifiers );
```

```
        AtomicObsRemoteSeq get_leaf_observations_with_policy (
                in QualifiedCodeStrSeq what,
                in TimeSpan when,
                in ObservationQualifierSeq qualifier,
                in QueryPolicySeq policy,
                in unsigned long max_sequence,
                out ObservationRemoteIterator the_rest )
            raises (
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan,
                InvalidQualifiers,
                DuplicateQualifiers,
                InvalidPolicies,
                DuplicatePolicies );

        AtomicObsRemoteSeq get_leaf_observations_by_value_type (
                in QualifiedCodeStrSeq what,
                in TimeSpan when,
                in ObservationQualifierSeq qualifier,
                in QualifiedCodeStr value_type,
                in unsigned long max_sequence,
                out ObservationRemoteIterator the_rest )
            raises (
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan,
                InvalidQualifiers,
                DuplicateQualifiers );

        ObservationDataSeq get_relations_toward_root (
                in QualifiedCodeStrSeq relation_name );

        ObservationDataSeq get_relations_away_from_root (
                in QualifiedCodeStrSeq relation_name );

};
```

count_observations()

Returns the number of observations held by this **CompositeObservationRemote**, according to the provided search-depth policy. Passing in a sequence of 0 policies indicates the use of the default policy for this server.

get_observation*()

These operations are similar to the operations of the same name on the **QueryAccess** interface, though returned as references to **ObservationRemote**.

However, observations are matched and returned only within the “searchable” target population of observations, associated by reference to this **CompositeObservationRemote**, at a depth of search according to the policy **SEARCH_DEPTH_POLICY**. For example, if the search-depth policy is **SEARCH_DEPTH_ONLY_ROOT**, only this **CompositeObservationRemote** will be searched (matched against). With a search-depth policy of

SEARCH_DEPTH_DEEPEST_POSSIBLE, the searchable population of observations consists of all observations which might be referenced by any of the directly held references, or their references, and so on.

get_leaf_observations()

Returns a sequence of all leaf observations that occur under this node in the observation tree. These leaf observations are by definition atomic (not composite), and the references returned are to **AtomicObservationRemote**.

get_leaf_observations_by_time()

As above, matching for the given observation code and time span in addition to atomicity. Time spans with end times greater than the server's current time are interpreted to mean "up till the current time." Indicate "all time previous to a given time" with a time stamp which has **EARLIEST_TIME** as the start time. Indicate "from a given time to now" with a time stamp which has **LATEST_TIME** as the end time. Therefore, a time span from **EARLIEST_TIME** to **LATEST_TIME** is equivalent to a "don't care" value. Note that the "who" parameter is already part of the context of this **CompositeObservationRemote**.

get_leaf_observations_by_qualifier()

As above, matching for the given observation qualifiers in addition.

get_leaf_observations_with_policy()

As above, but overriding the default policies with the ones provided.

get_relations_toward_root()

Return observations that are related to this observation in the direction toward of the root. This operation would be useful after navigating down through a tree of observations, and wishing to backtrack.

get_relations_away_from_root()

Return observations that are related to this observation in the direction away from the root. This would be the normal direction of exploration, from root out towards other related observations.

4.4.3.4 ObservationRemoteIterator Interface

```
interface ObservationRemoteIterator : AbstractManagedObject {

    unsigned long max_left ();

    boolean next_n (
            in unsigned long n,
            out ObservationRemoteSeq observation_remote_seq );
```

};

max_left()

This operation returns the number of items still left on the iterator.

next_n()

This operation returns the number of **ObservationRemote** objects as an out parameter as is indicated by the passed in ‘n’ parameter or the maximum left. Removes the returned objects from the iterator before returning.

4.4.3.5 *ObservedSubject Interface*

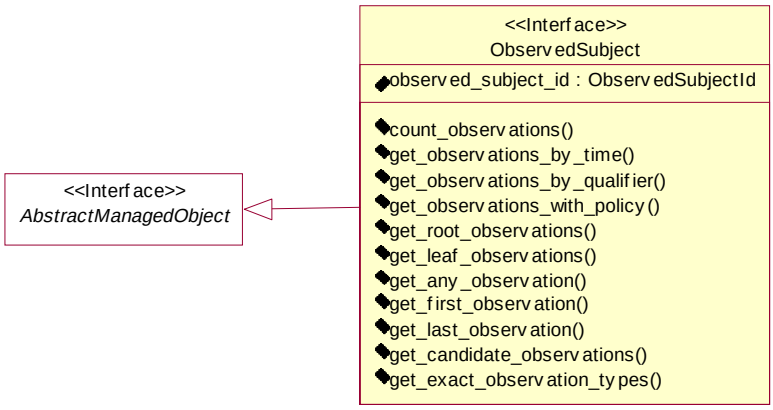

Figure 4-91 ObservedSubject Interface

```
interface ObservedSubject : AbstractManagedObject {

    readonly attribute ObservedSubjectId observed_subject_id;

    unsigned long count_observations (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in QueryPolicySeq policy )
        raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan,
            InvalidQualifiers,
            DuplicateQualifiers,
            InvalidPolicies,
            DuplicatePolicies );
```

```
ObservationRemoteSeq get_observations_by_time (
        in QualifiedCodeStrSeq what,
        in TimeSpan when,
        in unsigned long max_sequence,
        out ObservationRemoteIterator the_rest )
    raises (
        InvalidCodes,
        DuplicateCodes,
        InvalidTimeSpan );

ObservationRemoteSeq get_observations_by_qualifier (
        in QualifiedCodeStrSeq what,
        in TimeSpan when,
        in ObservationQualifierSeq qualifier,
        in unsigned long max_sequence,
        out ObservationRemoteIterator the_rest )
    raises (
        InvalidCodes,
        DuplicateCodes,
        InvalidTimeSpan,
        InvalidQualifiers,
        DuplicateQualifiers );

ObservationRemoteSeq get_observations_with_policy (
        in QualifiedCodeStrSeq what,
        in TimeSpan when,
        in ObservationQualifierSeq qualifier,
        in QueryPolicySeq policy,
        in unsigned long max_sequence,
        out ObservationRemoteIterator the_rest )
    raises (
        InvalidCodes,
        DuplicateCodes,
        InvalidTimeSpan,
        InvalidQualifiers,
        DuplicateQualifiers,
        InvalidPolicies,
        DuplicatePolicies );

ObservationRemoteSeq get_root_observations (
        in QualifiedCodeStrSeq what,
        in TimeSpan when,
        in unsigned long max_sequence,
        out ObservationRemoteIterator the_rest )
    raises (
        InvalidCodes,
        DuplicateCodes,
        InvalidTimeSpan );

AtomicObsRemoteSeq get_leaf_observations (
        in QualifiedCodeStrSeq what,
        in TimeSpan when,
        in unsigned long max_sequence,
        out ObservationRemoteIterator the_rest )
```

```
        raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan );

    ObservationRemote get_any_observation (
            in QualifiedCodeStrSeq what,
            in TimeSpan when )
        raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan );

    ObservationRemote get_first_observation (
            in QualifiedCodeStrSeq what,
            in TimeSpan when )
        raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan );

    ObservationRemote get_last_observation (
            in QualifiedCodeStrSeq what,
            in TimeSpan when )
        raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan );

    ObservationRemoteSeq get_candidate_observations (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
        raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan,
            InvalidQualifiers,
            DuplicateQualifiers );

    ObservationRemoteSeq get_exact_observation_types (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
        raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan );

};
```

observed_subject_id

The ID of the observed subject.

count_observations()

Return the number of observations which match the given search parameters.

get_observations*()

Analogous to the **QueryAccess** interface, except that the "who" is the current context's **ObservedSubject**. See Section 4.4.4.2, "QueryAccess Interface," on page 4-95 for details.

get_leaf_observations()

Return observations which are not composites, but rather the final, "leaf" nodes, with data. The server will match on any observation within an observation tree and return object references for each leaf observation in that tree. The server returns a zero-length sequence if no observations match the query.

get_any_observation()

This does a query for the observation types and time span. The server will match on any observation within an observation tree and return an object reference for any one of them. This is used when the client just needs a single response to the query and it does not matter which of the (potentially) multiple observations that match the query. The server returns a **null** object reference if no observations match the query.

get_first_observation(), get_last_observation()

This does a query for the observation types and time span for observations with this observation subject. The server will match on any observation within an observation tree and return an object reference for the first/last one in the time span. The server returns a **null** if no observations match the query.

get_candidate_observations()

This does a query for the observation types, time span and qualifiers for observations with this observation subject. The server uses its own matching engine to determine if a particular observation matches close enough to the query criteria. The results are returned with the ones matching best being returned first.

get_exact_observation_types()

This does a query for the observation types (codes) and time span for observations with this observation subject. This operation only returns observations which have codes that match exactly to one of the "what" values. This is a convenience method for employing the policy **SEARCH_SYNONYMOUS_CODES_FALSE**.

4.4.4 Query-Oriented Interface Specifications

The second set of **DsObservationAccess** interfaces to be discussed are those that are more function oriented, i.e. Query-Oriented interfaces. They support the use of query functionality for retrieval of a lot of data in a single request.

4.4.4.1 BrowseAccess Interface

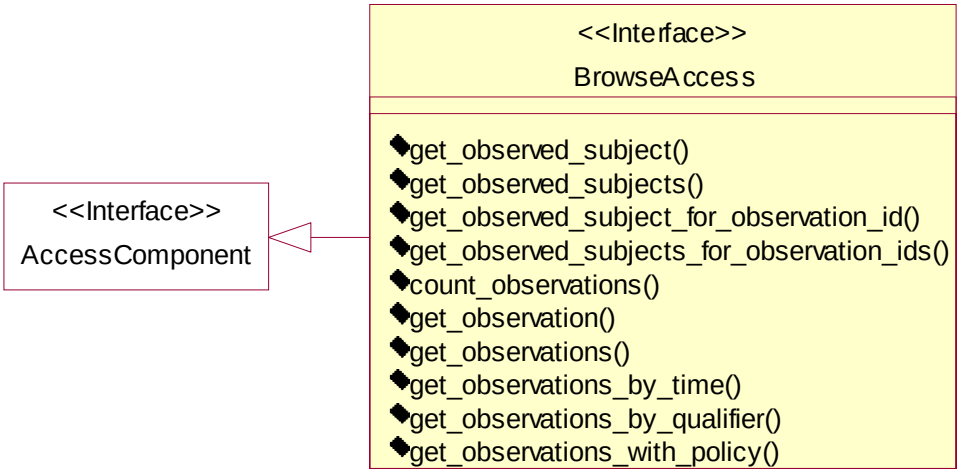

Figure 4-92 BrowseAccess Interface

```
interface BrowseAccess : AccessComponent {

    ObservedSubject get_observed_subject (
            in ObservedSubjectId who )
        raises (
            InvalidIds );

    ObservedSubjectSeq get_observed_subjects (
            in ObservedSubjectIdSeq who )
        raises (
            InvalidIds,
            DuplicateIds );

    ObservedSubject get_observed_subject_for_observation_id (
            in ObservationId observation_id )
        raises (
            InvalidOids );

    ObservedSubjectSeq get_observed_subjects_for_observation_ids (
            in ObservationIdSeq observation_ids )
        raises (
```

```
            InvalidOids,
            DuplicateOids );

unsigned long count_observations (
            in ObservedSubjectIdSeq who,
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in QueryPolicySeq policy )
        raises (
            InvalidIds,
            DuplicateIds,
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan,
            InvalidQualifiers,
            DuplicateQualifiers,
            InvalidPolicies,
            DuplicatePolicies );

ObservationRemote get_observation (
            in ObservationId observation_id )
        raises (
            InvalidOids );

ObservationRemoteSeq get_observations (
            in ObservationIdSeq observation_ids )
        raises (
            InvalidOids,
            DuplicateOids );

ObservationRemoteSeq get_observations_by_time (
            in ObservedSubjectId who,
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
        raises (
            InvalidIds,
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan );

ObservationRemoteSeq get_observations_by_qualifier (
            in ObservedSubjectIdSeq who,
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
        raises (
            InvalidIds,
            DuplicateIds,
            InvalidCodes,
            DuplicateCodes,
```

```
            InvalidTimeSpan,
            InvalidQualifiers,
            DuplicateQualifiers );

    ObservationRemoteSeq get_observations_with_policy (
            in ObservedSubjectIdSeq who,
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in QueryPolicySeq policy,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
        raises (
            InvalidIds,
            DuplicateIds,
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan,
            InvalidQualifiers,
            DuplicateQualifiers,
            InvalidPolicies,
            DuplicatePolicies );

};
```

get_observed_subject(), get_observed_subjects()

Returns **ObservedSubject** for the **ObservedSubjectId** passed in.

get_observed_subject_for_observation_id(),
get_subserved_subjects_for_observation_ids()

Returns an **ObservedSubject(Seq)** for the **ObservationId(Seq)** passed in. That is, the server determines the subject which is associated with a given observation.

get_observation*()

See Section 4.4.4.2, “QueryAccess Interface,” on page 4-95 for a complete definition of these operations. The difference here is that references to **ObservationRemote** are returned instead of (local) **ObservationData**.

4.4.4.2 QueryAccess Interface

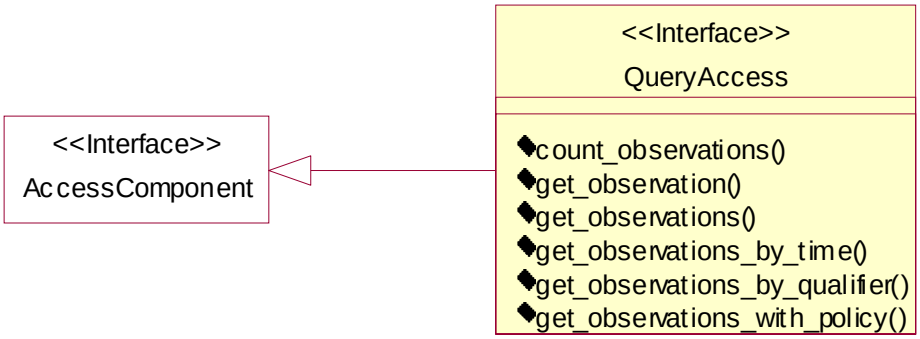

Figure 4-93 QueryAccess Interface

```
interface QueryAccess : AccessComponent {

    unsigned long count_observations (
            in ObservedSubjectIdSeq who,
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in QueryPolicySeq policy )
        raises (
            InvalidIds,
            DuplicateIds,
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan,
            InvalidQualifiers,
            DuplicateQualifiers,
            InvalidPolicies,
            DuplicatePolicies );

    ObservationData get_observation (
            in ObservationId observation_id )
        raises (
            InvalidOids );

    ObservationDataSeq get_observations (
            in ObservationIdSeq observation_ids )
        raises (
            InvalidOids,
            DuplicateOids );

    ObservationDataSeq get_observations_by_time (
            in ObservedSubjectId who,
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in unsigned long max_sequence,
```

```
            out ObservationDataIterator the_rest )
        raises (
            InvalidIds,
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan );

    ObservationDataSeq get_observations_by_qualifier (
            in ObservedSubjectIdSeq who,
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in unsigned long max_sequence,
            out ObservationDataIterator the_rest )
        raises (
            InvalidIds,
            DuplicateIds,
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan,
            InvalidQualifiers,
            DuplicateQualifiers );

    ObservationDataSeq get_observations_with_policy (
            in ObservedSubjectIdSeq who,
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in QueryPolicySeq policy,
            in unsigned long max_sequence,
            out ObservationDataIterator the_rest )
        raises (
            InvalidIds,
            DuplicateIds,
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan,
            InvalidQualifiers,
            DuplicateQualifiers,
            InvalidPolicies,
            DuplicatePolicies );

};
```

count_observations()

Return the number of observations which match the given search parameters.

get_observation(), get_observations()

Return the observation(s) corresponding to the passed in **ObservationId**(s).

get_observations_by_time()

Return all observations known by the server that match criteria specified by the “who,” “what,” and “when” parameters. A match is determined by the server’s matching engine, in accordance with default policies.

In essence, the “who,” “what,” and “when” filter the database of observations.

Time spans with end times greater than the server’s current time are interpreted to mean “through now”, so indicate the concept “from a given time through now” with a time stamp which has **LATEST_TIME** as the end time. Indicate the concept of “all time previous to a given time” with a time stamp which has **EARLIEST_TIME** as the start time. Therefore, a time span from **EARLIEST_TIME** to **LATEST_TIME** is equivalent to a “don’t care” for time, and includes all time possible through now.

A wildcard for individual TimeStamp elements, **TIME_WILDCARD** = “?”, is provided in the constants section. Use this character to indicate that a specific field should be treated as a wildcard. For example, “1999-??-02T22:00:00Z” would be equivalent to the concept of “the 2nd day of any month in 1999, at 22:00:00 GMT”.

Parsing for a wildcard is less efficient than a proper timestamp, so use the constants mentioned above, **EARLIEST_TIME** and **LATEST_TIME**, to indicate open-ended searches in the past and searches which include the most current information, rather than a **TimeStamp** filled with wildcard characters.

The “max_sequence” parameter indicates the maximum number to be returned within the **ObservationDataSeq**. A client may choose to receive many or few items via the synchronously returned **ObservationDataSeq** of **get_observation*()**. If the server determines that more than **max_sequence** observations meet the criteria for returning, the remaining observations are returned via the iterator “the_rest”.

Note – Note: a server may not support iterators (See Section 4.4.6.2, “AccessComponent Interface,” on page 4-107.), since iterators are remote objects and require the server to keep state. In the cases where all observations fit in the sequence, and where iterators are not supported, **the_rest** will be **null**, and the returned **ObservationDataSeq** will contain all the observations.

---

get_observations_by_qualifier()

Return observations which match the all parameters, including the additional qualifiers. The qualifiers provided in the parameter are for filtering the database, NOT to indicate what qualifiers to return. Specify what qualifiers to return with **QUALIFIER_RETURN_POLICY**.

get_observations_with_policy()

Return observations which match all parameters, according to the overriding policies specified in the “policy” parameter.

4.4.4.3 AsynchAccess Interface

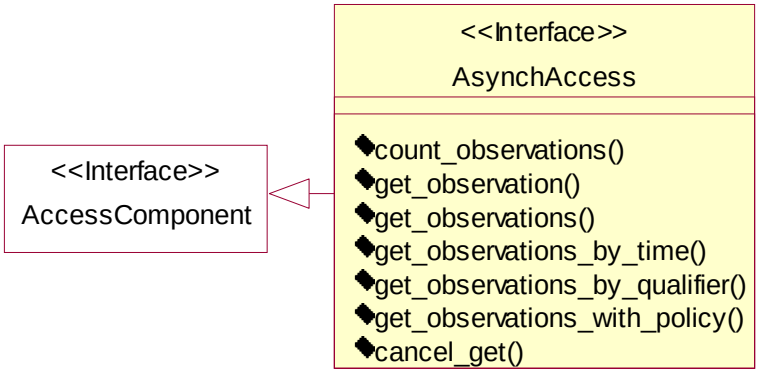

Figure 4-94 AsynchAccess Interface

```
interface AsynchAccess : AccessComponent {

    ServerCallId count_observations (
        in ObservedSubjectIdSeq who,
        in QualifiedCodeStrSeq what,
        in TimeSpan when,
        in ObservationQualifierSeq qualifier,
        in QueryPolicySeq policy,
        in ClientCallId client_call_id,
        in AsynchCallback client_callback );

    ServerCallId get_observation (
        in ObservationId observation_id,
        in ClientCallId client_call_id,
        in AsynchCallback client_callback );

    ServerCallId get_observations (
        in ObservationIdSeq observation_ids,
        in ClientCallId client_call_id,
        in AsynchCallback client_callback );

    ServerCallId get_observations_by_time (
        in ObservedSubjectId who,
        in QualifiedCodeStrSeq what,
        in TimeSpan when,
        in unsigned long max_sequence,
        in ClientCallId client_call_id,
        in AsynchCallback client_callback );

    ServerCallId get_observations_by_qualifier (
        in ObservedSubjectIdSeq who,
        in QualifiedCodeStrSeq what,
        in TimeSpan when,
```

```
        in ObservationQualifierSeq qualifier,
        in unsigned long max_sequence,
        in ClientCallId client_call_id,
        in AsynchCallback client_callback );

    ServerCallId get_observations_with_policy (
        in ObservedSubjectIdSeq who,
        in QualifiedCodeStrSeq what,
        in TimeSpan when,
        in ObservationQualifierSeq qualifier,
        in QueryPolicySeq policy,
        in unsigned long max_sequence,
        in ClientCallId client_call_id,
        in AsynchCallback client_callback );

    void cancel_get (
        in ServerCallId server_call_id );

};
```

The **AsynchAccess** component offers a means to make requests without blocking for the result synchronously. However, it adds complexity to the client. In particular, the client must instantiate a callback interface, register this CORBA object with the ORB, and take responsibility for timing out a request.

In contrast, a synchronous CORBA call can time-out a request in a relatively automatic fashion, with a timer in the TCP layer, typically set to 30 seconds or 1 minute, and generally provided within an ORB. An asynchronous request has no such automatic timer support in the ORB. A client must provide logic so that when a call does not complete, for whatever reason, the client does the right thing.

Also, there is no implied timing dependency between finishing a request and getting a reply. An asynchronous reply might begin before the request is completed. Clients must be prepared for an answer callback before they begin a request. In particular, the **client_call_id** should be ready for use at the callback implementation before the request is made, to identify any response if multiple calls are outstanding.

count_observations()

Returns the number of observations which match the given search parameters.

get_observation*()

The semantics for **get_observation*()** queries are the same as Section 4.4.4.2, “QueryAccess Interface,” on page 4-95. However, the results are delivered asynchronously.

In addition to the standard **get_observation*()** parameters, the client provides an object reference to an **AsynchCallback**. The server calls back to that object reference in order to return the results of the query.

Also, a **client_call_id** is provided by the client. The server returns this value when it calls the **AsynchCallback** so that the client can know which outstanding call is being returned (assuming there are multiple outstanding calls for this client).

Therefore, the client should make certain that each ID is unique within the scope of outstanding requests. For implementation, a simple count of requests since instantiation should be sufficient, if multiple calls can be outstanding at one time. If the client does not make multiple outstanding calls, the **client_call_id** has no utility and a constant can be used.

The **ServerCallId** returned from **get_observation*()** is an ID from the server for the request itself. The sole purpose of the **ServerCallId** is for cancellation. This identifier distinguishes the request uniquely within the server, among all requests from all clients. Again, for implementation, a simple count of incoming calls should be sufficient.

cancel_get()

A client can notify the server to cancel a request that has not yet completed. For example, consider a web browser when the user clicks on the “stop” button. In COAS, the client passes in the **ServerCallId**, which was previously returned from the **get_observation*()** call. (Another alternative would be to use ORB-specific calls to terminate the TCP connection, but that is outside the scope of COAS, and may not allow the server to properly terminate processing.)

Note – The **cancel_get()** function is a courtesy to the server only. The server is NOT responsible to actually stop the call. The call may complete, and the server may return the result by calling the **AsynchCallback** of the client. The client is responsible for discarding the answer in this case. Another alternative would be to unregister the **AsynchCallback** with the ORB.

4.4.4.4 AsynchCallback Interface

```
interface AsynchCallback {

    void put_observations (
        in ObservationDataSeq as_sequence,
        in ObservationDataIterator as_iterator,
        in ClientCallId client_call_id,
        in QualifiedCodeStrSeq result_status );

    void put_exception (
        in ClientCallId client_call_id,
        in AsynchException the_exception );

};
```

put_observations()

Called by **AsynchAccess** server to return the results from asynchronous queries. The **as_sequence** parameter contains the observations up to the maximum number specified in the original call with **max_sequence**. If there are more items than **max_sequence**, the parameter **as_iterator** will have one item, a reference to a **ObservationDataIterator**, from which the remaining observation items can be

retrieved from the server. The **as_iterator** parameter will be **null** if the returned observations fit within the **as_sequence** parameter or the server does not support iterators (see "AccessComponent Interface" on page 4-107). The **result_status** parameter supplies the client with **QualifiedCode**s constructed from constants **COMPLETING_RESULT** or **PARTIAL_RESULT** to indicate the status of the callback--completing a request, or only partially completing a request.

put_exception()

Called by the **AsynchAccess** server to return an exception condition.

4.4.4.5 *ConstraintLanguageAccess*

```
interface ConstraintLanguageAccess : AccessComponent {

    readonly attribute ConstraintLanguageSeq supported_languages;

    ObservationDataSeq get_by_constraint (
            in ConstraintExpression constraint,
            in QueryPolicySeq policy,
            in unsigned long max_sequence,
            out ObservationDataIterator the_rest )
        raises (
            InvalidConstraint,
            InvalidPolicies,
            DuplicatePolicies );

};
```

supported_languages

The sequence of constraint languages which are valid for constraint queries.

get_by_constraint()

Parse the given constraint and return matching observations. The policy parameter overrides any default policies. As with other **get_*()** calls, if more observations match the constraint than indicated by the **max_sequence** parameter, the remaining observations are returned via the iterator.

4.4.5 *Event and Notification Interface Specifications*

This section discusses the **DsObservationAccess** interfaces that subclass various interfaces in **CosEvent**. They support the notification of clients by one or more servers when an observation of interest has "arrived". They also send either the **ObservationData** itself, or sufficient information to retrieve the observation using another **DsObservationAccess** interface.

4.4.5.1 *EventSupplier Interface*

```
interface EventSupplier : AbstractManagedObject, PushSupplier {

    readonly attribute EndpointId endpoint_id;

    QualifiedCodeStrSeq obtain_offered_codes ();

    void connect_push_consumer (
            in PushConsumer push_consumer )
        raises (
            CosEventChannelAdmin::AlreadyConnected );

    PushConsumer get_connected_consumer ()
        raises (
            CosEventComm::Disconnected );

    void subscribe (
            in SubscriptionSeq subscriptions )
        raises (
            CosEventComm::Disconnected );

    SubscriptionSeq describe_subscriptions ()
        raises (
            NoSubscription );

    void generate_test_event (
            in ClientCallId clientId )
        raises (
            CosEventComm::Disconnected );

};
```

The **EventSupplier** interface encapsulates all that is necessary to supply events. Each supplier instance can be connected with exactly one **EventConsumer**. A server typically creates one or more suppliers for each client that wishes to receive events. A typical client implements the **EventConsumer** interface, and connects this consumer instance with an **EventSupplier** provided by the server's **SupplierAccess.create_supplier()**.

endpoint_id

When instantiated by the **SupplierAccess** factory, an **EventSupplier** receives an identifier from the factory. This identifier may be used to relocate the supplier by the factory.

obtain_offered_codes()

Returns a sequence of observation codes which this supplier can supply.

connect_push_consumer()

Establishes 1/2 of a connection, from the point of view of the supplier. The analogous **EventConsumer.connect_push_supplier()** must also be called to complete the connection from the client's point of view. The supplier can call **disconnect()** on the consumer in order to break the connection.

subscribe()

Establish an ongoing request for observations.

- The query is for future observations (as opposed to past observations).
- The time span is implied to be from the time **subscribe()** is called until this consumer is disconnected.
- The observations are returned within the **CosEventComm::push()** operation inherited by **EventConsumer**. The argument within this **push()** operation is an **Corba::any**. Within the **Corba::any** is an **ObservationData**.

The call to **subscribe()** begins a flow of events. Before the first call to **subscribe()**, no events flow. Supplier and consumer must be connected already, or exception Disconnected is thrown. Any subsequent call to **subscribe()** removes the previous subscription and begins a new subscription.

describe_subscription()

Returns the current subscription that has been set on the supplier.

generate_test_event()

Sends a test event to the consumer. This operation will be called by a savvy client after an interval of inactivity, to ascertain whether all is well in the event system and network. Without this direct request for a test event, a client might never know of network or event system problems.

The event resulting from this call will arrive, as with all events, in an **Corba::any**. Within the **Corba::any** will be an **ObservationData** as follows:

```
ObservationData
  code: TEST_EVENT  // see constants
  composite:  []       // empty
  qualifiers:   []      // empty
  value:   Any { clientId } // Any containing a long, the value of the input parameter
```

In other words, an **ObservationData** with a predetermined code **TEST_EVENT** from the constants section of this IDL, and with a payload of the given input parameter **clientId**.

4.4.5.2 EventConsumer Interface

```
interface EventConsumer : AbstractManagedObject, PushConsumer {

    readonly attribute EndpointId endpoint_id;

    SubscriptionSeq obtain_subscriptions ();
```

```
    void connect_push_supplier (
            in PushSupplier push_supplier )
        raises (
            CosEventChannelAdmin::AlreadyConnected );

    PushSupplier get_connected_supplier ()
        raises(
            CosEventComm::Disconnected );
};
```

The **EventConsumer** interface encapsulates all that is necessary to receive events. Each consumer instance can be connected with exactly one **EventSupplier**. A server would itself create an **EventConsumer** only when the server wished to receive events itself. A typically client would NOT call **ConsumerAccess.create_consumer()**, but rather implement the **EventConsumer** interface directly. After instantiating one of these "home grown" instances of **EventConsumer**, a typical client would connect this consumer instance with an **EventSupplier** provided by the server's **SupplierAccess.create_supplier()**.

endpoint_id

When instantiated by the **ConsumerAccess** factory, an **EventConsumer** receives an identifier from the factory. This identifier can be used to retrieve a reference to the **EventConsumer** via **ConsumerAccess.get_consumer_by_id()**.

Note that when the **EventConsumer** interface is implemented by a typical client (not a **DsObservationAccess** server), the identifier is not necessary nor relevant.

Note that when the **EventConsumer** interface is implemented by a typical client (not a **DsObservationAccess** server), the identifier is not necessary nor relevant.

obtain_subscriptions()

Returns a sequence of **Subscriptions** which this consumer would like to obtain. This operation is useful in an application management scenario. For example, a management application can use this operation to know what subscriptions to apply when connecting up a client and supplier without the explicit advance knowledge of this connection by those endpoint. Also, this operation could be reused by a client when subscribing, since it must have just such a list of subscription for **EventSupplier.subscribe()**.

connect_push_supplier()

Establishes 1/2 of a connection, from the point of view of the consumer. The analogous **EventSupplier.connect_push_consumer()** must also be called to complete the connection from the server point of view. The consumer can call **disconnect()** on the supplier in order to break the connection.

get_connected_supplier()

Returns a reference to the connected **EventSupplier**, or a Disconnected exception if no connection has been established yet.

4.4.5.3 SupplierAccess Interface

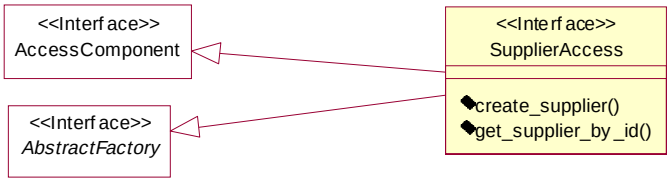

Figure 4-95 SupplierAccess Interface

```
interface SupplierAccess : AbstractFactory, AccessComponent {

    EventSupplier create_supplier ()
        raises (
            MaxConnectionsExceeded );

    EventSupplier get_supplier_by_id (
            in EndpointId endpoint_id )
        raises (
            InvalidEndpointId );

};
```

create_supplier()

Creates a new **EventSupplier** instance and returns it.

get_supplier_by_id()

This operation returns an object reference to the **EventSupplier** corresponding to the parameter **endpoint_id**. A **SupplierAccess** is responsible to keep track of all the **EventSupplier**s created, with their **EndpointId**s.

4.4.5.4 ConsumerAccess Interface

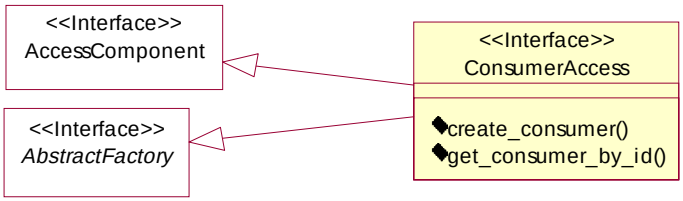

Figure 4-96 ConsumerAccess Interface

```
interface ConsumerAccess : AbstractFactory, AccessComponent {

    EventConsumer create_consumer ()
        raises (
            MaxConnectionsExceeded );

    EventConsumer get_consumer_by_id (
            in EndpointId endpoint_id )
        raises (
            InvalidEndpointId );

};
```

create_consumer()

Creates a new **EventConsumer** instance. Each consumer instance can be connected with exactly one **EventSupplier**. A server would create a consumer only when the server wished to receive events from another COAS server. A typical client would NOT call **create_consumer()**, but instead implement the **EventConsumer** interface, and connect this “home grown” instance with an **EventSupplier** provided by the **DsObservationAccess** server.

get_consumer_by_id()

This operation returns a reference to the **EventConsumer** corresponding to the parameter **endpoint_id**. To accomplish this, the **ConsumerAccess** factory should aggregate a reference and an **EndpointId** for all the **EventConsumer**s that it creates.

4.4.6 Utility Interface Specifications

The rest of the **DsObservationAccess** interfaces are described in this section.

4.4.6.1 ObservationLoader Interface

```
interface ObservationLoader : AccessComponent {

    void load_observations (
            in ObservationDataSeq observations );

};
```

load_observations()

Load observations into a **DsObservationAccess** server. Intended for use by legacy systems, which cannot be queried, but can output some stream of data.

4.4.6.2 AccessComponent Interface

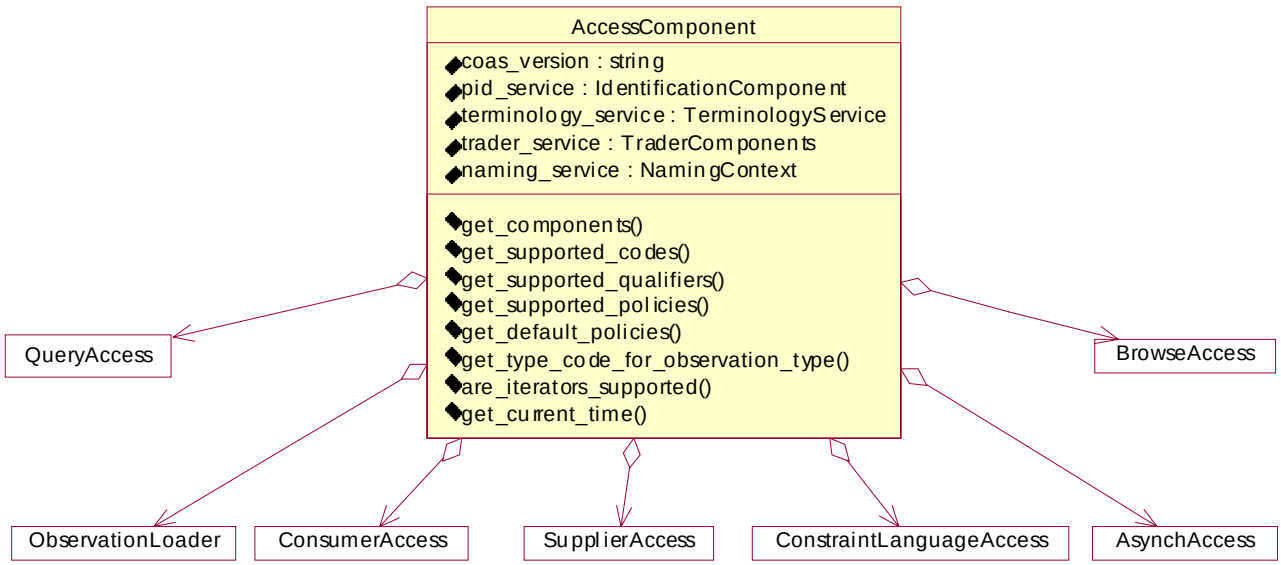

Figure 4-97 AccessComponent Interface

```
interface AccessComponent {

    readonly attribute string coas_version;

    readonly attribute IdentificationComponent pid_service;

    readonly attribute TerminologyService terminology_service;

    readonly attribute TraderComponents trader_service;

    readonly attribute NamingContext naming_service;

    AccessComponentData get_components ();

    QualifiedCodeStrSeq get_supported_codes (
        in unsigned long max_sequence,
        out QualifiedCodeIterator the_rest );

    QualifiedCodeStrSeq get_supported_qualifiers (
            in QualifiedCodeStr code )
        raises (
            InvalidCodes,
            NotImplemented );

    QualifiedCodeStrSeq get_supported_policies ();

    QueryPolicySeq get_default_policies ();
```

```
        TypeCode get_type_code_for_observation_type (
                in QualifiedCodeStr observation_type )
            raises (
                InvalidCodes,
                NotImplemented );

        boolean are_iterators_supported ();

        TimeStamp get_current_time ();

};
```

AccessComponent is the superclass of all components. **AccessComponent** allows discovery of the context of OMG services which a given component may use, in the form of references for pertinent services. These attributes may be **null**, indicating that the given service is lacking or unknown. Note that for each interface that provides the **AccessComponent** operations, those interfaces return the same response to each operation for a specific COAS server. So for example, a **QueryAccess.get_supported_codes()** operation will return the same response as the **BrowseAccess.get_supported_codes()** for the same COAS server.

coas_version

Version of COAS specification supported by this **DsObservationAccess** server, starting with “1.0” for the first approved specification.

pid_service, terminology_service, trader_service, naming_service

References to other OMG standard services which comprise the context of this **DsObservationAccess** server.

get_components()

This operation returns an **AccessComponentData**. **AccessComponentData** contains references to all implemented components as a convenience for clients that have one reference to a component, and wish to use a different component.

get_supported_codes()

A complete list of query codes for which this server can supply responses. Parameter **max_sequence** indicates the number of codes which the client wishes to be returned within the immediately returned sequence. Parameter **the_rest** contains an iterator for remaining items if and only if the number of codes is greater than **max_sequence**.

Note – Note: a query code is synonymous with a **QualifiedCode** from a terminology system and denotes a type of observation, such as Complete Blood Cound, Systolic Blood Pressure, etc.

---

get_supported_qualifiers()

A complete list of qualifiers which this server can match, and also supply as returned qualifiers, with respect to the given observation code. A server may be able to match/supply different sets of qualifiers for different codes.

get_supported_policies()

A complete list of policies which this server can employ when filtering on desired observations. The returned list is of codes only.

get_default_policies()

The policies which are in effect unless overridden via **get_observations_with_policy()**. The returned list is a list of name-value pairs, both the name of the policy and its default value.

get_type_code_for_observation_type()

Each **ObservationType** that a COAS server supports has a corresponding IDL data type which is the IDL type that observation is returned as from non-remote queries and other access mechanisms. This operation returns the **TypeCode** for the IDL type corresponding to the **ObservationType**.

The correspondence between a query code and the returned data type can be dynamically determined by querying the **AccessComponent.get_type_code_for_observation_type()** operation with a given code. However, a typical client will have such correspondences hardwired. A client will expect a certain data type for a given query code, and will have static programming to extract such data from the **CORBA::any**.

are_iterators_supported()

Returns a boolean describing whether this component can return iterator **ObservationDataIterator** and iterators for some of the data values in **DsObservationValues**. Iterators are remote objects.

If a server does not support iterators, all **ObservationData** and **ObservationValue** items are returned within sequences, and all out-parameter iterators returned as null. In this case, the input parameter **max_sequence** (present in many operations, indicating the client's preferred number of items returned in the sequence) is ignored by the server as it returns all observations within the sequence.

If a server supports iterators, the server will pay attention to the **max_sequence** input parameter, and an iterator will be instantiated and returned when the search for observations is successful and the input parameter **max_sequence** is set to less than the total number of observations found. Returning an iterator requires the server to be stateful, since the iterator is a remote object that must be instantiated and registered with the ORB for some lifetime.

For example, an implementer expecting a very large and busy set of clients may want to make a **QueryAccess** component which is stateless, and thus choose to return FALSE to **are_iterators_supported()**.

get_current_time()

Return a **TimeStamp** for the current time on the server. This can be useful for a client which resides in another timezone or which has questionable date/time settings (like a PC). A client can base a query on the server's time rather than the client's time.

4.4.6.3 ObservationDataIterator Interface

```
interface ObservationDataIterator : AbstractManagedObject {

    unsigned long max_left ();

    boolean next_n (
            in unsigned long n,
            out ObservationDataSeq observation_data_seq );

};
```

max_left()

This operation returns the number of items still left on the iterator.

next_n()

This operation returns the number of **ObservationData** objects as an out parameter as is indicated by the passed in 'n' parameter or the maximum left. Removes the returned objects from the iterator before returning.

4.4.6.4 QualifiedCodeIterator Interface

```
interface QualifiedCodeIterator : AbstractManagedObject {

    unsigned long max_left ();

    boolean next_n (
            in unsigned long n,
            out QualifiedCodeStrSeq codes );

};
```

max_left()

This operation returns the number of items still left on the iterator.

next_n()

This operation returns the number of **QualifiedCodeStr** objects as an out parameter as is indicated by the passed in 'n' parameter or the maximum left. Removes the returned objects from the iterator before returning.

4.4.6.5 AbstractFactory Interface

```
interface AbstractFactory {

    readonly attribute long max_connections;

    readonly attribute EndpointIdSeq current_connections;

};
```

max_connections

This attribute indicates the maximum number of connections the server will allow to be active at one time. Additional event suppliers and consumers will not be created beyond this limit.

current_connections

This attribute contains a sequence of endpoint IDs for the currently created event consumers or suppliers.

4.4.6.6 AbstractManagedObject Interface

```
interface AbstractManagedObject {

    void done ();

};
```

done()

Clients calls this operation when they are done using an object. This signals the server to deactivate or garbage collect the object. However, a savvy server will enforce a timeout after some amount of idle time for each managed object in order to cleanup after ill-behaved clients or traumatic client termination.

4.5 DsObservationValue

The **DsObservationValue** module defines the data containers for the Clinical Observations Access Service (COAS) specification. **ObservationValue** types are containers for the results of observing forms of biological phenomenon.

We have based this IDL on the Information Model presented in Section 4.3, “Information Model,” on page 4-6. We have selected a subset of all possible data containers, with the goal of making them as simple as possible. We realize that our set is not complete, yet we believe it to be disjoint.

If we had made use of Object-by-Value (OBV) technology, many of the data types defined in this module would have been sub-classes of an **ObservationValue** class. However, OBV was not available to a sufficient degree during the finalization of this specification. We tried to preserve the notion of inheritance even in defining our data

containers as **structs**, by using a comment **<struct name>:ObservationValue** to indicate this intended inheritance. A future revision of COAS may replace the **CORBA::any** in **ObservationData** with OBV.

4.5.1 Data Type Definitions

The following sections describe all the IDL for the data types used within the **DsObservationValue** module. To indicate which data types are intended to be sub-classes from **ObservationValue**, we have placed a comment immediately before those definitions containing the syntax "**<child class>: ObservationValue**".

```
// File: DsObservationValue.idl

#ifndef _DS_OBSERVATION_VALUE_IDL_
#define _DS_OBSERVATION_VALUE_IDL_

#include "DsObservationAccess.idl"

#pragma prefix "omg.org"

module DsObservationValue
{
...
};

#endif // _DS_OBSERVATION_VALUE_IDL_
```

The "Ds" prefix of **DsObservationValue** stands for "Domain Service." All OMG services are expected to start with "Ds" to isolate a particular name space from potential clashes.

4.5.2 Supporting Types

```
typedef TerminologyServices::ConceptCode   ConceptCode;
typedef NamingAuthority::QualifiedNameStr  QualifiedCodeStr;

typedef DsObservationAccess::AbstractManagedObject  AbstractManagedObject;
```

ConceptCode and **QualifiedCodeStr** are imported type definitions from the Lexicon Query Service (LQS) and Person Identification Service (PIDS) specifications.

AbstractManagedObject is an abstract interface that provides a convenience function for a client to notify the server when they are done using some remote object.

4.5.3 Time Types

```
// DateTime : ObservationValue;
typedef DsObservationAccess::TimeStamp DateTime;

// TimeSpan : ObservationValue;
typedef DsObservationAccess::TimeSpan TimeSpan;
```

These data types reuse the time definitions from **DsObservationAccess**. Descriptions for them can be found in “DateTime” on page 4-21 and in Section 4.4.2.11, “TimeStamp,” on page 4-75..

4.5.3.1 DateTime

A **DateTime** conveys a point in time, including the date.

4.5.3.2 TimeSpan

A **TimeSpan** conveys a period of time, with a beginning and end.

4.5.4 Person Type

```
// Person : ObservationValue;
typedef DsObservationAccess::ObservedSubjectId   Person;
```

This data type is reused from **DsObservationAccess**. A description for it can be found in Section 4.4.2.2, “External Typedefs,” on page 4-68.

4.5.4.1 Person

A Person contains an ID from a PIDS. It can be used to identify an organ, patient, health care provider, or population.

4.5.5 NoInformation Type

```
// NoInformation : ObservationValue;
struct NoInformation {
     QualifiedCodeStr reason;
     string text_description;
};

const QualifiedCodeStr NO_INFORMATION  =
    "DNS:omg.org/DsObservationValue/NO_INFORMATION";
```

There are instances when it is appropriate to convey that information is unavailable or missing. For further discussion and an example see “NoInformation” on page 4-27.

4.5.5.1 NoInformation

A **NoInformation** value indicates both that specific information is missing and how or why it is missing. It can occur in place of any other observation value.

reason

The **reason** attribute is used to denote why the information is missing or unavailable. This attribute is a **QualifiedCode** and should come from a well-defined terminology system.

text_description

The **text_description** attribute contains a text string to be displayed in support of the reason attribute.

NO_INFORMATION is a **QualifiedCode** to be used in an **AtomicObservation** to indicate that the value it contains is “NoInformation.” This code is defined here because we believe that this concept does not appear in existing standard coding schemes. It is our intention for this code to fill the gap until this concept appears in a standard coding scheme.

4.5.6 Text Types

```
// PlainText : ObservationValue;
typedef string PlainText;

// UniversalResourceIdentifier : ObservationValue;
struct UniversalResourceIdentifier {
    ConceptCode protocol;
    string address;
};

// PhysicalLocationDescription : ObservationValue;
typedef string PhysicalLocationDescription;
```

Although there are several data types that use a string to carry the information, only one communicates the observation directly. The others contain textual references or pointers to the location or resource where the data can be accessed.

4.5.6.1 Plain Text

PlainText is used to communicate observation values as ideas in the form of writing. It is expected that along with the text will be a qualifier that indicates the language in which the text is written.

4.5.6.2 UniversalResourceIdentifier

A **UniversalResourceIdentifier** is used to reference information that has some tie to a technology that can perform some action. Also see the discussion in “TechnologyInstanceLocator” on page 4-20.

protocol

This is the protocol associated with the address. The protocol indicates the technology to be used to interpret the address. For example, http.

address

The address attribute contains some structured sequence of characters that the protocol knows how to interpret. For example, www.example.com.

4.5.6.3 PhysicalLocationDescription

A **PhysicalLocationDescription** is used to reference information or items that are not located within some information space, but are instead located in some physical space.

4.5.7 Coded Types

```
// CodedElement : ObservationValue;
typedef TerminologyServices::QualifiedCodeInfo CodedElement;

// LooselyCodedElement : ObservationValue;
struct LooselyCodedElement {
    string text;
    TerminologyServices::CodingSchemeId coding_scheme_id;
    TerminologyServices::VersionId version_id;
};
```

The coded data types provide a mechanism to communicate observation values that have been coded in some form or another. Further information can be found in “CodedElement” on page 4-17 and “LooselyCodedElement” on page 4-17.

4.5.7.1 CodedElement

A **CodedElement** is coded in the sense that it is a unique identifier. This unique identifier can then be used to ask a terminology system specific questions about the **CodedElement**. For example, its representation based on some context, or its definition.

4.5.7.2 LooselyCodedElement

There are times when a code that the user wants cannot be realized or found within a terminology system (e.g., is not in the list of allowable values). In which case the **LooselyCodedElement** can be used to send text instead.

text

The **text** attribute is a String and is used when no **CodedElement** from a terminology system can be determined.

coding_scheme_id

The **coding_scheme_id** attribute is the id, from an LQS, that is used to identify the coding scheme where the text was intended.

version_id

The **version_id** attribute is used to identify the version of the coding scheme where the text was intended.

4.5.8 Multimedia Types

```
typedef sequence<octet> Blob;

interface MultimediaIterator  : AbstractManagedObject {

     unsigned long max_left ();

     boolean next_n (
          in unsigned long n,
          out Blob multimedia_part );
};

// Multimedia : ObservationValue;
struct Multimedia {
     string content_type;
     string other_mime_header_fields;
     Blob a_blob;
     unsigned long long total_size;
     MultimediaIterator the_iterator;
};
```

We define a supporting data type and an interface for the Multimedia data type.

Blob

A **Blob** is just an opaque container for data, even more opaque than a **CORBA::any**.

MultimediaIterator

The **MultimediaIterator** is used to retrieve data in chunks. Iterators in general are described in more detail in Section 4.4.6.3, “ObservationDataIterator Interface,” on page 4-110.

4.5.8.1 Multimedia

For the communication of observations such as images, audio or video recordings or large documents, we utilize the Multipurpose Internet Mail Extensions (MIME) standard. We refer you to “Multimedia” on page 4-19 for more information on MIME and references to the MIME documentation.

content_type

The **content_type** is a structured attribute that identifies the general media type e.g., Application, Audio, Image, Message, Model, Mulitpart,Text and Video, and the specific format used.

other_mime_header_fields

The **other_mime_header_fields** contains the rest of the MIME header. We have made this available so that clients can gain further information about what is contained in this data value.

a_blob

The **a_blob** attribute contains the observation value itself.

total_size

The **total_size** attribute contains the number of bytes of data in the Blob.

the_iterator

the_iterator may contain a reference to a multimedia iterator when the Blob is larger than the client wants to receive at one time. It can be used to retrieve the rest of the Blob in chunks.

4.5.9 Simple Measurement Types

```
// Numeric : ObservationValue;
struct Numeric {
     QualifiedCodeStr units;
     float value;
};

// Range : ObservationValue;
struct Range {
     QualifiedCodeStr units;
     float lower;
     float upper;
};

// Ratio : ObservationValue;
struct Ratio {
     float numerator;
     float denominator;
};
```

The simple measurement types are designed to contain single or paired numbers, that is quantitative measurements or observations. The units associated with the **Numeric** and **Range** types are **QualifiedCode**s and should come from a well-defined terminology system. All other attributes mentioned in “Measurement” on page 4-22 should be attached to the relevant **AtomicObseration** as qualifiers.

4.5.9.1 Numeric

Numeric is used to communicate a single measurement or quantitative value.

4.5.9.2 Range

Range is used to associate two related values together. For example, 1<= X <= 5. It is assumed that the value in the lower attribute is less than or equal to the value in the upper attribute.

4.5.9.3 Ratio

A **Ratio** value contains a numerator quantity and a denominator quantity, and is used in those situations where the ratio is more easily understood than the equivalent real number. It should be noted that the ratio data type must not be used as a handy representation of two related values. In particular, blood pressure values, commonly reported as 120/80 mm Hg, are not ratios!

4.5.10 Complex Measurement Types

```
struct XYPair {
    float x;
    float y;
};

typedef sequence<XYPair> XYPairSeq;

interface CurveIterator  : AbstractManagedObject {

    unsigned long max_left ();

    boolean next_n (
        in unsigned long n,
        out XYPairSeq curve_part );
};

// Curve : ObservationValue;
struct Curve {
    XYPairSeq xy_pairs;
    QualifiedCodeStr x_units;
    QualifiedCodeStr y_units;
    unsigned long long total_size;
    CurveIterator the_iterator;
};
```

In **DsObserationValue** we define one data type that contains many measurements. To support this data type several supporting methods must be defined.

XYPair, XYPairSeq

These are the low level data types for holding a vector of data pairs.

CurveIterator

The **CurveIterator**, like all other iterators, is the mechanism for retrieving the data in chunks.

4.5.10.1 Curve

Curve is a data type for retrieving paired measurements or values.

xy_pairs

The **xy_pairs** contains the data sequence.

x_units, y_units

The **x_units** and **y_units** are **QualifiedCode** and should come from a well-defined terminology system. In healthcare, the **x_units** is usually a time (e.g., milliseconds, seconds or minutes). The **y_units** is often a quantitative measurement.

total_size

The **total_size** attribute contains the total number of elements in the curve.

the_iterator

the_iterator may contain a reference to a **CurveIterator** that can be used to retrieve a very large curve data sequence in chunks.

4.6 DsObservationTimeSeries

The **DsObservationTimeSeries** module defines an extension to the basic data types and interfaces of the **DsObservationAccess** and **DsObservationValue** modules. The **TimeSeries** data types and operations were designed to support the unique features and needs of accessing vital sign waveforms.

4.6.1 Data Type Definitions

The following sections list all the IDL for the data types used within the **DsObservationTimeSeries** module.

```
// File: DsObservationTimeSeries.idl

#ifndef _DS_OBSERVATION_TIME_SERIES_IDL_
#define _DS_OBSERVATION_TIME_SERIES_IDL_

#include <DsObservationAccess.idl>

module DsObservationTimeSeries
{
...
};

#endif // _DS_OBSERVATION_TIME_SERIES_IDL_
```

Provides an **#ifdef** wrapper to preclude multiple inclusions.

4.6.2 External Typedefs

```
typedef DsObservationAccess::AbstractManagedObject  AbstractManagedObject;
typedef DsObservationAccess::NameValuePair          NameValuePair;
```

```
typedef DsDsObservationAccess::QueryPolicy              QueryPolicy;
typedef DsObservationAccess::QueryPolicySeq        QueryPolicySeq;
typedef DsObservationAccess::ObservationQualifierSeq ObservationQualifierSeq;
typedef DsObservationAccess::QualifiedCodeStr      QualifiedCodeStr;
typedef DsObservationAccess::TimeStamp             TimeStamp;
typedef DsObservationAccess::TimeSpan              TimeSpan;
```

Describes external dependencies.

4.6.3 Time Types

```
// TimeDelta : ObservationValue;
struct TimeDelta {
    float  delta;  // calculated with constants below, NOT with calendaring
    QualifiedCodeStr units;
};

// approximations for time deltas, NOT for calendaring
const float YEAR        = 31557600.0;  // 60*60*24*365.25
const float MONTH       = 2629800.0;   // 60*60*24*365.25/12
const float DAY         = 86400.0;     // 60*60*24
const float HOUR        = 3600.0;      // 60*60
const float MINUTE      = 60.0;        // 60
const float SECOND      = 1.0;         // 1
const float MILLISECOND = 0.001;       // 1/1000
```

TimeDelta is intended for calculation with the time constants provided. For example, an appropriate use of **TimeDelta** might be the time difference between the beginning of a EKG session and the end of the session. This difference would be expressed as seconds or milliseconds, with any necessary calculation (converting from minutes or hours) via the constants provided. This is different than UTC calculations based on the calendar. In particular, the number of seconds in a given calendar day or year may vary since the spin of the earth varies, and UTC is kept in relative harmony with that spin.

4.6.4 Typedef, Enum, Union, and Sequence Types

```
typedef NameValuePair  Filter;
typedef sequence<Filter>  FilterSeq;

enum ValueSeqType { OtherSeqDataType, OctetType, ShortType,
   LongType, LongLongType, FloatType, DoubleType
};

union ValueSeq switch ( ValueSeqType ) {
     case OctetType    : sequence< octet > octet_seq;
     case ShortType    : sequence< short > short_seq;
     case LongType     : sequence< long > long_seq;
     case LongLongType : sequence< long long > long_long_seq;
     case FloatType    : sequence< float > float_seq;
     case DoubleType   : sequence< double > double_seq;
     case OtherSeqDataType : any the_any;
};
```

```
typedef sequence<QualifiedCodeStr,1>  OptionalCodeSeq;
typedef sequence<float,1>  OptionalFloatSeq;
```

4.6.5 Iterator Types

```
interface TimeSeriesIterator  : AbstractManagedObject {

      unsigned long max_left ();

      boolean next_n (
            in unsigned long n,
            out ValueSeq curve_part );
};
```

4.6.6 TimeSeries

```
// TimeSeries : ObservationValue;
struct TimeSeries {
      TimeDelta sample_period;
      ValueSeq values;
      QualifiedCodeStr value_units;
      unsigned long long total_size;  // number of items in values + remaining on
     iterator
      TimeSeriesIterator the_iterator;
};
```

TimeSeries will include a non-null iterator if the number of items in the sequence "values" is greater than the current policy **RETURN_MAX_SEQUENCE_FOR_VALUE_POLICY**. In other words, specify the number of items desired in the sequence with this policy, and that will determine whether an iterator is returned also.

This policy is analogous to the parameter "max_sequence" in **QueryAccess.get_observations_by_time()** and similar operations. The input parameter "max_sequence" specifies the number of observations to return in a sequence. But a single observation which contains a **TimeSeries** payload in its **ObservationData.value (CORBA::any)** may have any number of items in the **TimeSeries.values (**a sequence**)**. The number of items desired by the client is specified via the **RETURN_MAX_SEQUENCE_FOR_VALUE_POLICY**.

4.6.7 Exceptions

```
exception OutOfRange{};

exception NotImplemented{};

exception FilterNotSupported{};

exception NoValidValues{};
```

4.6.8 TimeSeriesRemote

```
struct TimeSeriesRemoteAttributes {
    QualifiedCodeStr code;
    QualifiedCodeStr units;
    OptionalCodeSeq accuracy;
    OptionalFloatSeq precision;
    OptionalFloatSeq corner_frequency;
    OptionalFloatSeq highest_frequency;
    TimeSpan time_span;
    TimeDelta time_delta;
    unsigned long long total_size;
    QualifiedCodeStrSeq supported_filters;
    QueryPolicySeq supported_policies;
};

// TimeSeriesRemote : ObservationValue;
interface TimeSeriesRemote : AbstractManagedObject {

    readonly attribute QualifiedCodeStr code;
    readonly attribute QualifiedCodeStr units;
    readonly attribute OptionalCodeSeq accuracy;
    readonly attribute OptionalFloatSeq precision;
    readonly attribute OptionalFloatSeq corner_frequency;
    readonly attribute OptionalFloatSeq highest_frequency;
    readonly attribute TimeSpan time_span;
    readonly attribute TimeDelta time_delta;
    readonly attribute unsigned long long total_size;
    readonly attribute QualifiedCodeStrSeq supported_filters;
    readonly attribute QueryPolicySeq supported_policies;
    readonly attribute ValueSeqType default_value_type;

    TimeSeriesRemoteAttributes get_attributes ();

    float get_sample_number (
            in unsigned long long index,
            out ObservationQualifierSeq qualifiers )
        raises (
            OutOfRange );

    float get_sample (
            in TimeStamp time_stamp,
            out ObservationQualifierSeq qualifiers )
        raises (
            OutOfRange );

    TimeSeries get_snippet (
            in TimeSpan time_span,
            out ObservationQualifierSeq qualifiers )
        raises (
            OutOfRange );

    float get_max (
            in TimeSpan time_span )
        raises (
```

```
                OutOfRange,
                NoValidValues );

        float get_min (
                in TimeSpan time_span )
            raises (
                OutOfRange,
                NoValidValues  );

        float get_mean (
                in TimeSpan time_span )
            raises (
                OutOfRange,
                NoValidValues );

        float get_median (
                in TimeSpan time_span )
            raises (
                OutOfRange,
                NoValidValues );

        TimeSeries get_resampled (
                in TimeSpan time_span,
                in TimeDelta sample_rate,
                in QueryPolicySeq policy,
                out ObservationQualifierSeq qualifiers )
            raises (
                NotImplemented );

        TimeSeries get_rescaled (
                in TimeSpan time_span,
                in float scale_factor,
                in QueryPolicySeq policy,
                out ObservationQualifierSeq qualifiers )
            raises (
                NotImplemented );

        TimeSeries get_resampled_rescaled (
                in TimeSpan time_span,
                in TimeDelta sample_rate,
                in float scale_factor,
                in QueryPolicySeq policy,
                out ObservationQualifierSeq qualifiers )
            raises (
                NotImplemented );

        TimeSeries get_filtered (
                in TimeSpan time_span,
                in FilterSeq filters,
                in QueryPolicySeq policy,
                out ObservationQualifierSeq qualifiers )
            raises (
                NotImplemented,
                FilterNotSupported );
};
```

(partial documentation follows)

get_attributes()

Returns the structure containing the attributes pertaining to the specific **TimeSeriesRemote**.

get_sample()

Return a single data point corresponding to the timestamp and limiting qualifiers.

get_snippet()

Gets a series of data points (i.e. a waveform snippet) that correspond to the time period defined in the timespan.

get_max()

Returns the numeric maximum data value in the defined timespan.

get_min()

Returns the numeric minimum data value in the defined timespan.

get_mean()

Returns the arithmetic mean or average data value of all the individual data points included within the timespan specified.

get_median()

Returns the median data value of all the individual data points included within the timespan specified.

4.7 DsObservationRelations

This section describes the relations that can exist between observations. In COAS, a relation is modeled by a qualifying, composite observation which has a code describing the relationship. This qualifying, composite observation links an observation and its related observations.
For example, consider a relationship where Observation A is caused by a number of other observations. In the graphic below, a linking ObservationData structure, Observation B, holds the identity of that relationship, along with the list of related observations.

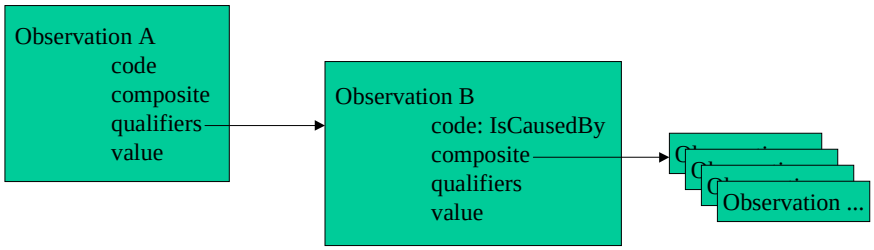

Figure 4-98 Observation B relates Observation A with other observations. A “IsCausedBy” others.

A starter set of codes for relations is defined below. The relations indicated by these codes are documented in the Comité Européen De Normalisation (CEN, European Committee For Standardization) First Working Document of Electronic Healthcare, Record Communication - Part 2: Domain Termlist, (CEN/TC 251/N98-116).

4.7.1 CEN Naming Convention

Code names from CEN/TC 251/N98-116, table A.5, are created as follows:

- start with “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/”.
- add relationship names from table A.5, translated as:
  - replace “/” with “_”.
  - replace space with nothing, capitalizing next word.
  - omit apostrophe, periods, parenthesis, and other punctuation.

4.7.2 Observation Type for Relations

Each observation code is associated with a particular IDL static type definition. All relation codes refer to composite observations. Hence their observation type in COAS is a composite observation, which is just **ObservationData**.

```
typedef DsObservationAccess::ObservationData RELATION_type;
```

4.7.3 Relation Codes

4.7.3.1 Produce

Relations that produce or are produced by healthcare activity.

```
const QualifiedCodeStr Produces = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/Produces";

const QualifiedCodeStr IsProducedBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsProducedBy";
```

4.7.3.2 Document

Relations that document or are documented by a healthcare activity.

```
const QualifiedCodeStr Documents = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/Documents";

const QualifiedCodeStr IsDocumentedBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsDocumentedBy";
```

4.7.3.3 Report

Relations that report or are reported by a healthcare activity.

```
const QualifiedCodeStr Reports = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/Reports";

const QualifiedCodeStr IsReportedBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsReportedBy";
```

4.7.3.4 Graphic

Relations that describe or are described by graphic properties of a graphic object.

```
const QualifiedCodeStr Describes = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/Describes";

const QualifiedCodeStr IsDescribedBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsDescribedBy";
```

4.7.3.5 Identified/Incorporated

Relations that are identified by or incorporates a graphic object within a study product.

```
const QualifiedCodeStr IsIdentifiedWithin = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsIdentifiedWithin";

const QualifiedCodeStr IsIncorporatedBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsIncorporatedBy";
```

4.7.3.6 Source/Derived

Relations that are sources for or are derived from a graphic property from a study product.

```
const QualifiedCodeStr IsSourceFor = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsSourceFor";

const QualifiedCodeStr IsDerivedFrom = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsDerivedFrom";
```

4.7.3.7 Compared/Reference

Relations that are compared to or are reference for a situation.

```
const QualifiedCodeStr IsComparedTo = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsComparedTo";

const QualifiedCodeStr IsReferenceFor = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsReferenceFor";
```

4.7.3.8 Recorded

Relations that are recorded against a family history.

```
const QualifiedCodeStr IsRecordedAgainst = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsRecordedAgainst";
```

4.7.3.9 Supercede

Relations that supercede or are superseded by a clinical state.

The relation “supersede” must not confused with mechanisms used to manage different versions of a document. This link in fact refers to different judgements performed at different times according to evolving evidence. For example, a change of diagnosis after new evidence is discovered.

```
const QualifiedCodeStr Supercedes = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/Supercedes";
const QualifiedCodeStr IsSupercededBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsSupercededBy";
```

4.7.3.10 Framework

Relations that are a framework for or is framed in.a situation, or document.

```
const QualifiedCodeStr IsFrameworkFor = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsFrameworkFor";
const QualifiedCodeStr IsFramedBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsFramedBy";
```

4.7.3.11 Phase

Relations that have phases or are phases of a healthcare activity.

```
const QualifiedCodeStr HasPhase = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasPhase";
const QualifiedCodeStr IsPhaseOf = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsPhaseOf";
```

4.7.3.12 Next Phase

Relations that have a next phase or are a next phase in a healthcare activity.

```
const QualifiedCodeStr HasNextPhase = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasNextPhase";
const QualifiedCodeStr IsNextPhaseWRT = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsNextPhaseWRT";
```

4.7.3.13 Associate

Relations that are associated with a condition.

```
const QualifiedCodeStr IsAssociateTo = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsAssociateTo";
```

4.7.3.14 Assigned/Setting

Relations that are assigned to or are a setting for situation assigned to a problem.

```
const QualifiedCodeStr IsAssignedTo = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsAssignedTo";
const QualifiedCodeStr IsSettingFor = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsSettingFor";
```

4.7.3.15 Interpretation

Relations that are interpretations of or are interpreted as a condition of findings, or reports.

```
const QualifiedCodeStr IsInterpretationOf = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsInterpretationOf";
const QualifiedCodeStr IsInterpretedAs = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsInterpretedAs";
```

4.7.3.16 Progress

Relations that have progress or are progress of a condition.

```
const QualifiedCodeStr HasProgress = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasProgress";
const QualifiedCodeStr IsProgressOf = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsProgressOf";
```

4.7.3.17 Cause

Relations that have causes or are causes of a condition.

```
const QualifiedCodeStr HasCause = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasCause";
const QualifiedCodeStr IsCauseOf = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsCauseOf";
```

4.7.3.18 Co-exists

Relations that co-exist with a condition.

```
const QualifiedCodeStr CoExistsWith = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/CoExistsWith";
```

4.7.3.19 Evidence

Relations that have evidence for or are evidence of a diagnosis.

```
const QualifiedCodeStr HasEvidence = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasEvidence";
const QualifiedCodeStr IsEvidenceFor = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsEvidenceFor";
```

4.7.3.20 Triggers

Relations that trigger or are triggered by presence of a risk state.

```
const QualifiedCodeStr Triggers = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/Triggers";
const QualifiedCodeStr IsTriggeredBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsTriggeredBy";
```

4.7.3.21 Goal

Relations that have goals or are goals of a healthcare activity.

```
const QualifiedCodeStr HasGoal = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasGoal";
const QualifiedCodeStr IsGoalOf = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsGoalOf";
```

4.7.3.22 Motivation

Relations that have motivation or are motivation for a healthcare activity.

```
const QualifiedCodeStr HasMotivation = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasMotivation";
const QualifiedCodeStr IsMotivationFor = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsMotivationFor";
```

4.7.3.23 Consequence

Relations that have consequences or are consequences of a healthcare activity.

```
const QualifiedCodeStr HasConsequence = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasConsequence";
const QualifiedCodeStr IsConsequenceOf = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsConsequenceOf";
```

4.7.3.24 Topic

Relations that have topics or are topics for informing.

```
const QualifiedCodeStr HasTopic = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasTopic";
const QualifiedCodeStr IsTopicFor = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsTopicFor";
```

4.7.3.25 Target

Relations that have targes or are targets for informing.

```
const QualifiedCodeStr HasTarget = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasTarget";
const QualifiedCodeStr IsTargetOf = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsTargetOf";
```

4.7.3.26 Provides Information

Relations that provide information about a condition.

```
const QualifiedCodeStr ProvidesInformationAbout = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/ProvidesInformationAbout";
```

4.7.3.27 Circumstances

Relations that have circumstances or are circumstances for supporting an activity.

```
const QualifiedCodeStr HasCircumstances = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasCircumstances";
const QualifiedCodeStr IsCircumstanceOf = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsCircumstanceOf";
```

4.8 DsObservationQualifiers

This section describes a set of codes defined for qualifiers. Qualifiers are observations which can be used to modify and refine the meaning of other observations. For example, **Date_TimeOfTheObservation** and **OrderingProvider** are common qualifiers. Along with an observation like the amount of glucose in a blood sample, COAS clients will often be interested in the time of the observation and the care provider who ordered it.

The codes below, mostly from HL7 v.2.3, provide a starter set of qualifiers. This set is in no way intended to imply an exhaustive set. However, by use of the COAS naming convention detailed below, the implication here is that all data definitions of HL7v2.3 are usable as observations and qualifiers.

Furthermore, definitions from the Comité Européen De Normalisation (CEN, European Committee For Standardization) First Working Document of Electronic Healthcare, Record Communication - Part 2: Domain Termlist, (CEN/TC 251/N98-116) are all potential qualifiers and observations. See Section 4.7.1, “CEN Naming Convention,” on page 4-125

These codes are defined with qualifiers in mind, but the codes can be used as query codes as well. For example, a COAS client might wish to query for all ordering providers for a given patient over a given time span. In this case, the code **OrderingProvider** would be used as the (query) observation code rather than in the list of qualifiers regarding some other observation.

4.8.1 HL7 Naming Convention

Code names from HL7v2.3 are created as follows: based on HL7 v3.2 standard distribution, appendix A. (APPA.doc), table A.6 DATA ELEMENT NAMES:

- start with “DNS:omg.org/DsObservationAccess/HL72.3/”
- add the HL7 segment, like OBX or PID, plus a slash
- add HL7 data element names taken from table A.6, translated as:
  - replace “/” with “_”
  - replace space with nothing, capitalizing next word
  - omit apostrophe, periods, parenthesis, and other punctuation.

Most of the examples below are HL7 components with multiple subcomponents. To identify individual subcomponents, additional slash(es) + subcomponent name(s) can follow the component names. For example, in the OBR (result) segment, one particular code,

```
const QualifiedCodeStr SpecimenSource = "DNS:omg.org/DsObservationAcess/HL72.3/OBR/SpecimenSource"
```

SpecimenSource, is a composite. One subcomponent of **SpecimenSource**, the body site, can be specified as

```
const QualifiedCodeStr SpecimenSourceBodySite = "DNS:omg.org/DsObservationAccess/HL72.3/OBR/SpecimenSource/BodySite";
```

by appending the name “/BodySite” as shown. Thus, **SpecimenSourceBodySite** refers to the specific subcomponent of **SpecimenSource**.

4.8.2 Observation Type for Qualifiers

Each observation code is associated with a particular IDL static type definition. Most of the examples below are HL7 components with multiple subcomponents. Hence their observation type in COAS is a composite observation, which is just **ObservationData**.

```
typedef DsObservationAccess::ObservationData COMPOSITE_OBSERVATION_type;
```

However, a small subcomponent, **SpecimenSourceBodySite**, is listed in HL7 documentation as having type (CE), coded element. This would correspond to a **QualifiedCodeStr** in COAS.

The association between code and data definition can be confirmed for a particular server with **AccessComponent.get_observation_type()**.

One way to indicate this association in static IDL is to list a code **<code>**, and immediately following it, a typedef for a type with name **<code>_type**. For example,

```
const QualifiedCodeStr SpecimenSourceBodySite = "DNS:omg.org/DsObservationAccess/HL72.3/OBR/SpecimenSource/BodySite";
typedef QualifiedCodeStr SpecimenSourceBodySite_type;
```

4.8.3 Qualifier Codes

The following qualifiers are identified as a starter set.

4.8.3.1 COAS - Specific

```
const QualifiedCodeStr COAS_OBSERVATION_ID = "DNS:omg.org/DsObservationAccess/COAS_OBSERVATION_ID";
```

4.8.3.2 HL7 - Clinical Times

```
const QualifiedCodeStr Date_TimeOfTheObservation = "DNS:omg.org/DsObservationAccess/HL72.3/OBX/Date_TimeOfTheObservation";
const QualifiedCodeStr EventOnsetDate_Time = "DNS:omg.org/DsObservationAccess/HL72.3/PEO/EventOnsetDate_Time";
const QualifiedCodeStr OrderEffectiveDate_Time = "DNS:omg.org/DsObservationAccess/HL72.3/ORC/OrderEffectiveDate_Time";
const QualifiedCodeStr ProcedureDate_Time = "DNS:omg.org/DsObservationAccess/HL72.3/PR1/ProcedureDate_Time";
const QualifiedCodeStr RequestedDate_Time = "DNS:omg.org/DsObservationAccess/HL72.3/OBR/RequestedDate_Time";
const QualifiedCodeStr VerificationDate_Time = "DNS:omg.org/DsObservationAccess/HL72.3/IN1/VerificationDate_Time";
const QualifiedCodeStr ActionDate_Time = "DNS:omg.org/DsObservationAccess/HL72.3/GOL/ActionDate_Time";
const QualifiedCodeStr AttestationDate_Time = "DNS:omg.org/DsObservationAccess/HL72.3/DG1/AttestationDate_Time";
const QualifiedCodeStr TranscriptionDate_Time = "DNS:omg.org/DsObservationAccess/HL72.3/TXA/TranscriptionDate_Time";
```

4.8.3.3 HL7 - Roles

```
const QualifiedCodeStr PatientIDExternalID = "DNS:omg.org/DsObservationAccess/HL72.3/PID/PatientIDExternalID";
const QualifiedCodeStr PatientIDInternalID = "DNS:omg.org/DsObservationAccess/HL72.3/PID/PatientIDInternalID";
const QualifiedCodeStr OrderingProvider = "DNS:omg.org/DsObservationAccess/HL72.3/OBR/OrderingProvider";
const QualifiedCodeStr ProducerID = "DNS:omg.org/DsObservationAccess/HL72.3/OBX/ProducerID";
const QualifiedCodeStr CollectorIdentifier = "DNS:omg.org/DsObservationAccess/HL72.3/OBR/CollectorIdentifier";
const QualifiedCodeStr ResponsibleObserver = "DNS:omg.org/DsObservationAccess/HL72.3/OBX/ResponsibleObserver";
const QualifiedCodeStr Technician = "DNS:omg.org/DsObservationAccess/HL72.3/OBR/Technician";
const QualifiedCodeStr PrincipalResultInterpreter = "DNS:omg.org/DsObservationAccess/HL72.3/OBR/PrincipalResultInterpreter";
```

4.8.3.4 HL7 - OBR (Request)

```
const QualifiedCodeStr SpecimenSource = "DNS:omg.org/DsObservationAccess/HL72.3/OBR/SpecimenSource";
const QualifiedCodeStr ReasonForStudy = "DNS:omg.org/DsObservationAccess/HL72.3/OBR/ReasonForStudy";
const QualifiedCodeStr DiagnosticServiceSectionID = "DNS:omg.orgDsObservationAccess/HL72.3/OBR/DiagnosticServiceSectionID";
```

```
const QualifiedCodeStr SpecimenSourceBodySite = "DNS:omg.orgDsObservationAccess/HL72.3/OBR/SpecimenSourceBodySite";
```

4.8.3.5 HL7 - OBX (Reply)

```
const QualifiedCodeStr AbnormalFlags = "DNS:omg.org/DsObservationAccess/HL72.3/OBX/AbnormalFlags";

const QualifiedCodeStr ObservationMethod = "DNS:omg.org/DsObservationAccess/HL72.3/OBX/ObservationMethod";

const QualifiedCodeStr Units = "DNS:omg.org/DsObservationAccess/HL72.3/OBX/Units";

const QualifiedCodeStr ReferencesRange = "DNS:omg.org/DsObservationAccess/HL72.3/OBX/ReferencesRange";

const QualifiedCodeStr ObservationIdentifier = "DNS:omg.org/DsObservationAccess/HL72.3/OBX/ObservationIdentifier";
```

4.8.3.6 HL7 - PV1 (Patient Visit)

```
const QualifiedCodeStr PatientLocation = "DNS:omg.org/DsObservationAccess/HL72.3/PV1/PatientLocation";
```

4.9 DsObservationPolicies

Policies are name-value pairs which instruct the server on how to search and return observations. They consist of a policy name (a **QualifiedCodeStr**), and a value (a **CORBA::any**). Each policy has a typedef to define what is inside the **CORBA::any**.

4.9.1 SEARCH_DEPTH_POLICY

```
const QualifiedCodeStr SEARCH_DEPTH_POLICY = "DNS:omg.org/DsObservationAccess/policy/SEARCH_DEPTH_POLICY";
typedef short SearchDepthPolicyType;

const SearchDepthPolicyType SEARCH_DEPTH_ONLY_ROOT = 0x0;
const SearchDepthPolicyType SEARCH_DEPTH_DEEPEST_POSSIBLE = 0xFFFF;
```

SEARCH_DEPTH_POLICY indicates how many levels down an item hierarchy a server is to look for a match to the input parameters. Only positive integers, including zero, make sense:

- 0 means just the root of the tree.
- 1 means to search the root and one level of items below the root.
- 2 means to search the root and two more levels down
- 3 means to search the root and three more levels down
- etc.

SEARCH_DEPTH_DEEPEST_POSSIBLE means to search all levels for a match.

Default = **SEARCH_DEPTH_DEEPEST_POSSIBLE**.

4.9.2 RETURN_DEPTH_POLICY

```
const QualifiedCodeStr RETURN_DEPTH_POLICY = “DNS:omg.org/DsObservationAccess/policy/RETURN_DEPTH_POLICY”;
typedef QualifiedCodeStr ReturnDepthPolicyType;

const ReturnDepthPolicyType RETURN_DEPTH_ROOT_ONLY = “DNS:omg.org/DsObservationAccess/policy/RETURN_DEPTH_ROOT_ONLY”;
const ReturnDepthPolicyType RETURN_DEPTH_ALL = “DNS:omg.org/DsObservationAccess/policy/RETURN_DEPTH_ALL”;
const ReturnDepthPolicyType RETURN_DEPTH_ALL_LEAVES = “DNS:omg.org/DsObservationAccess/policy/RETURN_DEPTH_ALL_LEAVES”;
```

```
const ReturnDepthPolicyType RETURN_DEPTH_LEAVES_OF_MATCHED =
“DNS:omg.org/DsObservationAccess/policy/RETURN_DEPTH_LEAVES_OF_MATCHED”;
const ReturnDepthPolicyType RETURN_DEPTH_MATCHED_ONLY = “DNS:omg.org/DsObservationAccess/policy/RETURN_DEPTH_MATCHED_ONLY”;
const ReturnDepthPolicyType RETURN_DEPTH_MATCHED_AND_DOWN =
“DNS:omg.org/DsObservationAccess/policy/RETURN_DEPTH_MATCHED_AND_DOWN”;
```

RETURN_DEPTH_POLICY indicates which items in a potential tree of items which get returned. After matching on certain items, these items may have various other related items contained in their “composite” field, making up a “tree” of items from the (matched) root item.

ROOT_ONLY means that only the root item is returned.

RETURN_ALL means the full item structure gets returned from the root, down to and including the leaves.

MATCHED_ONLY means to only return the item that was matched on, independent of where it is in the tree.

MATCHED_AND_DOWN means to return a tree of items starting with the one matched, down to and including the leaf items.

LEAVES_OF_MATCHED means to only return the leaf items of the part of the tree starting from the matched item on down but no BranchItems.

ALL_LEAVES means to return all LeafItems in the whole tree that had a match, starting from the root.

Default = **RETURN_DEPTH_MATCHED_AND_DOWN**.

4.9.3 SEARCH_SYNONYMOUS_CODES_POLICY

```
const QualifiedCodeStr SEARCH_SYNONYMOUS_CODES_POLICY =
“DNS:omg.org/DsObservationAccess/policy/SEARCH_SYNONYMOUS_CODES_POLICY”;
typedef QualifiedCodeStr SearchSynonymousCodesPolicyType;

const SearchSynonymousCodesPolicyType SEARCH_SYNONYMOUS_CODES_FALSE =
“DNS:omg.org/DsObservationAccess/policy/SEARCH_SYNONYMOUS_CODES_FALSE”;
const SearchSynonymousCodesPolicyType SEARCH_SYNONYMOUS_CODES_TRUE =
“DNS:omg.org/DsObservationAccess/policy/SEARCH_SYNONYMOUS_CODES_TRUE”;
```

SEARCH_SYNONYMOUS_CODES_POLICY indicates to search for all possible matches on a code, including any synonymous codes or subtype codes that the server might know as a result of a Terminology (LQS) service or otherwise.

For example, if searching for all “blood-cell count” observations, both a red-blood-cell count and white-blood-cell count would match, as subtypes.

SEARCH_SYNONYMOUS_CODES_TRUE means all synonyms and subtypes are considered matches too.

SEARCH_SYNONYMOUS_CODES_FALSE means that only an exact match will be returned. Thus, FALSE implies that the set of codes is treated as an XOR list.

default = **SEARCH_SYNONYMOUS_CODES_TRUE**

4.9.4 RETURN_OBSERVATION_VALUES_POLICY

```
const QualifiedCodeStr RETURN_OBSERVATION_VALUES_POLICY =
“DNS:omg.org/DsObservationAccess/policy/RETURN_OBSERVATION_VALUES_POLICY”;
typedef QualifiedCodeStr ReturnObservationValuesPolicyType;

const ReturnObservationValuesPolicyType RETURN_NO_OBSERVATION_VALUES =
“DNS:omg.org/DsObservationAccess/policy/RETURN_NO_OBSERVATION_VALUES”;
const ReturnObservationValuesPolicyType RETURN_OBSERVATION_VALUES =
“DNS:omg.org/DsObservationAccess/policy/RETURN_OBSERVATION_VALUES”;
```

RETURN_OBSERVATION_VALUES_POLICY is useful when only contextual (“meta”) information is desired. No values are returned, only qualifiers. That is, **ObservationData.value** sequences are returned empty, even for atomic observations.

Use this policy when, for example, a value is large, and the network traffic to download it to a client would be considerable. The client can display all the context information from qualifiers (observation time, ordering provider, etc.) in some list of observations, without downloading the actual item until a user clicks to examine the actual data.

default = **RETURN_OBSERVATION_VALUES**

4.9.5 SHORTCIRCUIT_SEARCH_..._POLICY

```
const QualifiedCodeStr SHORTCIRCUIT_SEARCH_CODES_ON_SUCCESS_POLICY =
“DNS:omg.org/DsObservationAccess/policy/SHORTCIRCUIT_SEARCH_CODES_ON_SUCCESS_POLICY”;
typedef boolean ShortcircuitSearchCodesOnSuccessPolicyType;

const ShortcircuitSearchCodesOnSuccessPolicyType SHORTCIRCUIT_SEARCH_CODES_ON_SUCCESS_FALSE = FALSE;
const ShortcircuitSearchCodesOnSuccessPolicyType SHORTCIRCUIT_SEARCH_CODES_ON_SUCCESS_TRUE  = TRUE;
```

SHORTCIRCUIT_SEARCH_CODES_ON_SUCCESS_POLICY is employed only when a sequence of query codes is passed in. If a successful match is found for one of the codes, this policy indicates to discard the rest of the codes, short circuiting the search for other codes. Such a policy might be useful in a situation where it is not clear what qualified code will work for a given server, so that multiple codes are used.

default = **SHORTCIRCUIT_SEARCH_CODES_ON_SUCCESS_FALSE**

4.9.6 SEARCH_SYNONYMOUS_IDS_POLICY

```
const QualifiedCodeStr SEARCH_SYNONYMOUS_IDS_POLICY = “DNS:omg.org/DsObservationAccess/policy/SEARCH_SYNONYMOUS_IDS_POLICY”;
typedef boolean SearchSynonymousIdsPolicyType;

const SearchSynonymousIdsPolicyType SEARCH_SYNONYMOUS_IDS_FALSE = FALSE;
const SearchSynonymousIdsPolicyType SEARCH_SYNONYMOUS_IDS_TRUE = TRUE;
```

SEARCH_SYNONYMOUS_IDS_POLICY indicates whether or not to search for all possible matches on an ID, including any synonyms that might be known by the server via a PIDS translation or otherwise.

default = **SEARCH_SYNONYMOUS_IDS_TRUE**

4.9.7 SHORTCIRCUIT_SEARCH_IDS_ON_SUCCESS_POLICY

```
const QualifiedCodeStr SHORTCIRCUIT_SEARCH_IDS_ON_SUCCESS_POLICY =
"DNS:omg.org/DsObservationAccess/policy/SHORTCIRCUIT_SEARCH_IDS_ON_SUCCESS_POLICY";
```

```
typedef boolean ShortcircuitSearchIdsOnSuccessPolicyType;

const ShortcircuitSearchIdsOnSuccessPolicyType SHORTCIRCUIT_SEARCH_IDS_ON_SUCCESS_FALSE = FALSE;
const ShortcircuitSearchIdsOnSuccessPolicyType SHORTCIRCUIT_SEARCH_IDS_ON_SUCCESS_TRUE = TRUE;
```

SHORTCIRCUIT_SEARCH_IDS_ON_SUCCESS_POLICY is used in a situation where a sequence of subject IDs is passed in. If a successful match is found for one of the Ids, the policy indicates to discard the rest of the Ids, shortcircuit any further searching for other codes. Such a policy might useful in a situation where it is not clear what Id will work for a given server.

default = **SHORTCIRCUIT_SEARCH_IDS_ON_SUCCESS_FALSE**

4.9.8 RETURN_ITEMS_IN_TIME_SPAN_POLICY

```
const QualifiedCodeStr RETURN_ITEMS_IN_TIME_SPAN_POLICY =
"DNS:omg.org/DsObservationAccess/policy/RETURN_ITEMS_IN_TIME_SPAN_POLICY";
typedef QualifiedCodeStr ReturnItemsInTimeSpanPolicyType;

const ReturnItemsInTimeSpanPolicyType RETURN_ITEMS_IN_TIME_SPAN_FIRST_ITEM_ONLY =
"DNS:omg.org/DsObservationAccess/policy/RETURN_ITEMS_IN_TIME_SPAN_FIRST_ITEM_ONLY";
const ReturnItemsInTimeSpanPolicyType RETURN_ITEMS_IN_TIME_SPAN_LAST_ITEM_ONLY =
"DNS:omg.org/DsObservationAccess/policy/RETURN_ITEMS_IN_TIME_SPAN_LAST_ITEM_ONLY";
const ReturnItemsInTimeSpanPolicyType RETURN_ITEMS_IN_TIME_SPAN_ALL_ITEMS =
"DNS:omg.org/DsObservationAccess/policy/RETURN_ITEMS_IN_TIME_SPAN_ALL_ITEMS";
```

RETURN_ITEMS_IN_TIME_SPAN_POLICY indicates whether to only return the first or last matched items in a time span.

default = **RETURN_ITEMS_IN_TIME_SPAN_ALL_ITEMS**.

4.9.9 MATCHING_STRENGTH_POLICY

```
const QualifiedCodeStr MATCHING_STRENGTH_POLICY = "DNS:omg.org/DsObservationAccess/policy/MATCHING_STRENGTH_POLICY";
typedef float MatchingStrengthPolicyType;

const MatchingStrengthPolicyType MATCHING_STRENGTH_WEAKEST = 0.0F;
const MatchingStrengthPolicyType MATCHING_STRENGTH_STRONGEST = 1.0F;
```

MATCHING_STRENGTH_POLICY indicates whether exact matches only are to be returned, or if close (as determined by the server) matches are returned too. This matching strength concept is similar to the PIDS **find_candidates()** operation.

default = **MATCHING_STRENGTH_STRONGEST**.

4.9.10 PARAM_CHECKING_POLICY

```
const QualifiedCodeStr PARAM_CHECKING_POLICY = "DNS:omg.org/DsObservationAccess/policy/PARAM_CHECKING_POLICY";
typedef boolean ParamCheckingPolicyType;

const ParamCheckingPolicyType PARAM_CHECKING_FALSE = FALSE;
const ParamCheckingPolicyType PARAM_CHECKING_TRUE = TRUE;
```

PARAM_CHECKING_POLICY allows a server to ignore parameters that it does not recognize (IDs, codes, qualifiers, **TimeStamps**, etc.) without throwing an exception. Unknown items are ignored in matching algorithms. If this policy is true, the server

will raise an exception when unknown IDs or codes are passed in. For a more narrowly-focused policy, see “IGNORE_UNMATCHABLE_QUALIFIERS_POLICY” on page 4-137.

default = **PARAM_CHECKING_TRUE**

4.9.11 QUALIFIER_RETURN_POLICY

```
const QualifiedCodeStr QUALIFIER_RETURN_POLICY = “DNS:omg.org/DsObservationAccess/policy/QUALIFIER_RETURN_POLICY”;
typedef sequence<QualifiedCodeStr> QualifierReturnPolicyType;

const QualifiedCodeStr QUALIFIER_RETURN_ALL = “DNS:omg.org/DsObservationAccess/policy/QUALIFIER_RETURN_ALL”;
const QualifiedCodeStr QUALIFIER_RETURN_NONE = “DNS:omg.org/DsObservationAccess/policy/QUALIFIER_RETURN_NONE”;

const QualifiedCodeStr QUALIFIER_NOT_TO_RETURN_POLICY =
“DNS:omg.org/DsObservationAccess/policy/QUALIFIER_NOT_TO_RETURN_POLICY”;
typedef sequence<QualifiedCodeStr> QualifierNotToReturnPolicyType;
```

QUALIFIER_RETURN_POLICY makes it possible for the client to indicate exactly which qualifiers should be returned with the **ObservationData**.

For a list of qualifiers: See “Qualifier Codes” on page 4-131.

Note there is a great difference between returning qualifiers, and filtering by qualifiers. The later happens as a result of passing in qualifiers via the **get_observations_by_qualifier()** operation and similar operations. The former is accomplished with this policy.

default = **QUALIFIER_RETURN_NONE**

4.9.12 RELATIONS_RETURN_POLICY

```
const QualifiedCodeStr RELATIONS_RETURN_POLICY = “DNS:omg.org/DsObservationAccess/policy/RELATIONS_RETURN_POLICY”;
typedef sequence<QualifiedCodeStr> RelationsReturnPolicyType;

const QualifiedCodeStr RELATIONS_RETURN_ALL = “DNS:omg.org/DsObservationAccess/policy/RELATIONS_RETURN_ALL”;
const QualifiedCodeStr RELATIONS_RETURN_NONE = “DNS:omg.org/DsObservationAccess/policy/RELATIONS_RETURN_NONE”;

const QualifiedCodeStr RELATIONS_NOT_TO_RETURN_POLICY =
“DNS:omg.org/DsObservationAccess/policy/RELATIONS_NOT_TO_RETURN_POLICY”;
typedef sequence<QualifiedCodeStr> RelationsNotToReturnPolicyType;
```

RELATIONS_RETURN_POLICY makes it possible for the client to indicate exactly which relations should be returned with the **ObservationData**.

For a list of relations: See Section 4.7.3, “Relation Codes,” on page 4-125.

default = **RELATIONS_RETURN_NONE**

4.9.13 RETURN_MOST_RECENT_N_OBSERVATIONS_POLICY

```
const QualifiedCodeStr RETURN_MOST_RECENT_N_OBSERVATIONS_POLICY =
"DNS:omg.org/DsObservationAccess/policy/RETURN_MOST_RECENT_N_OBSERVATIONS_POLICY";
typedef unsigned long ReturnMostRecent_N_ObservationsPolicyType;

const ReturnMostRecent_N_ObservationsPolicyType RETURN_MOST_RECENT_N_OBSERVATIONS_ALL = 0xFFFFFFFF;
```

RETURN_MOST_RECENT_N_OBSERVATIONS_POLICY provides a means to return items according to their temporal proximity to the current time of the server. This policy overrides any **TimeSpan** provided as an input parameter.

default = **RETURN_MOST_RECENT_N_OBSERVATIONS_ALL**.

4.9.14 TIME_SERIES_..._ALGORITHM_POLICY

```
const QualifiedCodeStr TIME_SERIES_REMOTE_RESAMPLE_ALGORITHM_POLICY =
“DNS:omg.org/DsObservationAccess/policy/TIME_SERIES_REMOTE_RESAMPLE_ALGORITHM_POLICY”;
typedef sequence<QualifiedCodeStr> TimeSeriesRemoteResampleAlgorithmPolicyType;
```

4.9.15 TIME_SERIES_..._PREFERENCE_POLICY

```
const QualifiedCodeStr TIME_SERIES_REMOTE_RETURN_TYPE_PREFERENCE_POLICY =
“DNS:omg.org/DsObservationAccess/policy/TIME_SERIES_REMOTE_RETURN_TYPE_PREFERENCE_POLICY”;
typedef DsObservationTimeSeries::ValueSeqType TimeSeriesRemoteReturnTypePreferencePolicyType;
```

4.9.16 RETURN_MAX_SEQUENCE_FOR_VALUE_POLICY

```
const QualifiedCodeStr RETURN_MAX_SEQUENCE_FOR_VALUE_POLICY =
"DNS:omg.org/DsObservationAccess/policy/RETURN_MAX_SEQUENCE_FOR_VALUE_POLICY";
typedef unsigned long ReturnMaxSequenceForValuePolicyType;
const ReturnMaxSequenceForValuePolicyType RETURN_MAX_SEQUENCE_FOR_VALUE_ALL = 0xFFFFFFFF;
```

RETURN_MAX_SEQUENCE_FOR_VALUE_POLICY is used when an **ObservationValue** can include an iterator. For example, **DsObservationValues::Multimedia** includes an iterator field “the_rest”. A non-null iterator is returned within the Multimedia struct only if the number of items in the sequence “values” is greater than the current **RETURN_MAX_SEQUENCE_FOR_VALUE_POLICY**. In other words, specify the number of items desired in the sequence with this policy, and that will determine whether an iterator is returned also.

This policy is analogous to the parameter “max_sequence” in **QueryAccess.get_observations_by_time()** and similar operations. The input parameter “max_sequence” specifies the number of observations to return in a sequence. But a single observation which contains a **Multimedia** payload in its **ObservationData.value** (a **CORBA::any**) may have any number of items in the **Multimedia.a_blob** (a sequence). The number of items desired by the client is specified via the **RETURN_MAX_SEQUENCE_FOR_VALUE_POLICY**.

default = **RETURN_MAX_SEQUENCE_FOR_VALUE_ALL**

4.9.17 IGNORE_UNMATCHABLE_QUALIFIERS_POLICY

```
const QualifiedCodeStr IGNORE_UNMATCHABLE_QUALIFIERS_POLICY =
"DNS:omg.org/DsObservationAccess/policy/IGNORE_UNMATCHABLE_QUALIFIERS_POLICY";
typedef boolean IgnoreUnmatchableQualifiersPolicyType;
const IgnoreUnmatchableQualifiersPolicyType IGNORE_UNMATCHABLE_QUALIFIERS_TRUE = TRUE;
const IgnoreUnmatchableQualifiersPolicyType IGNORE_UNMATCHABLE_QUALIFIERS_FALSE = FALSE;
```

IGNORE_UNMATCHABLE_QUALIFIERS_POLICY applies to the searching rules in a more specific manner than **PARAM_CHECKING_POLICY**. The latter turns off all exceptions, but the user may wish to have parameter checking except for qualifiers.

Hence **IGNORE_UNMATCHABLE_QUALIFIERS_TRUE** means that unknown or inapplicable qualifiers will not be considered in the matching algorithm. Otherwise, the introduction of an inapplicable qualifier would cause no matches to be found. A client can tell what qualifiers are applicable for a given query code from the method **AccessComponent.get_supported_qualifiers()**.

default = **IGNORE_UNMATCHABLE_QUALIFIERS_FALSE**

Glossary

List of Terms

The definitions below have special meaning for this specification. Either they started from general definitions and were refined during the development of this specification; or they are definitions of concepts from other OMG specifications, and were taken directly from those specifications; or they were important acronyms used in this specification. Terms appearing in **boldface** type are defined elsewhere within this section.

Access	The ability to retrieve or get, and the action of retrieving, information about **observations** and the **observations** themselves.
Blob	Acronym for Binary Large Object; used in this document to represent an opaque string of **octets** that is passed unchanged between the **server** and the **client**.[1]
Client	Any system or application that accesses or requests service from a **DsObservationAccess server**.
Coded Concept	A local name**,** consisting of a fixed sequence of alphanumeric characters, that is used to designate one or more presentations, definitions, comments or instructions. within a **coding scheme**.[2]
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. See the **LQS** specification for definitions of the terms presentations, definitions, comments and instructions.[2]
Context	The interrelated conditions in which something exists or occurs.
Domain Name	The name of an ID Domain in which an ID has meaning. That is, IDs are only relevant in a particular ID Domain. Each ID Domain has a Domain Name that is unique and different from all other ID Domain Names.[1]
Encounter	A meeting between two systems in which meaningful transactions are passed and processed.
Event	A noteworthy happening or activity.
LQS	The OMG's Lexicon Query Service

Observation	An act of recognizing and noting a fact or occurrence often involving measurement with instruments or a judgement on or inference from what one has observed or noted.
Observation Qualifier	One that satisfies requirements or meets a specified standard.
Observation Value	The fact, note, or result of an **observation.**
PIDS	The OMG's Person Identification Service
Policy	A definite course or method of action selected from among alternatives and in light of given conditions to guide and determine present and future decisions.
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).[2]
Relationship	A state of affairs existing between two systems that have dealings between each other.
Server	A **DsObservationAccess** system that offers services or performs actions on the behalf or interest of requests made by a **DsObservationAccess client**.
Subject of Care	A biological entity, patient, or population that is under **observation** or measurement.

1. Person Identification Service, OMG Formal Document formal/99-03-05.
2. Lexicon Query Service, OMG Formal Document formal/99-03-06.

Appendix A Complete IDL Specification

A.1 DsObservationAccess

```
// File: DsObservationAccess.idl

#ifndef _DS_OBSERVATION_ACCESS_IDL_
#define _DS_OBSERVATION_ACCESS_IDL_

#include <CosNaming.idl>
#include <CosTrading.idl>
#include <TerminologyServices.idl>
#include <NamingAuthority.idl>
#include <PersonIdService.idl>
#include <CosEventComm.idl>
#include <CosEventChannelAdmin.idl>
#include <orb.idl>

#pragma prefix “omg.org”


module DsObservationAccess {

    //
    // EXTERNAL TYPEDEFS
    //

    typedef PersonIdService::QualifiedPersonId  ObservedSubjectId;
    typedef TerminologyServices::QualifiedCode  QualifiedCode;
    typedef NamingAuthority::QualifiedNameStr   QualifiedCodeStr;
    typedef PersonIdService::DomainName         IdDomainName;

    typedef PersonIdService::IdentificationComponent IdentificationComponent;
    typedef CosNaming::NamingContext                 NamingContext;
    typedef CosTrading::TraderComponents             TraderComponents;
    typedef TerminologyServices::TerminologyService  TerminologyService;

    typedef CosEventComm::PushConsumer  PushConsumer;
    typedef CosEventComm::PushSupplier  PushSupplier;

    typedef CORBA::TypeCode  TypeCode;


    //
    // FORWARD DECLARATIONS
    //

    interface AbstractFactory;
    interface AbstractManagedObject;
    interface AccessComponent;
    interface AsynchCallback;
    interface AsynchAccess;
    interface AtomicObservationRemote;
    interface BrowseAccess;
    interface CompositeObservationRemote;
    interface ConsumerAccess;
    interface ConstraintLanguageAccess;
    interface EventConsumer;
    interface EventSupplier;
    interface ObservationDataIterator;
    interface ObservationLoader;
    interface ObservationRemote;
    interface ObservationRemoteIterator;
    interface ObservedSubject;
    interface QualifiedCodeIterator;
    interface QueryAccess;
    interface SupplierAccess;


    //
    // STRUCTS
    //
```

```
struct AccessComponentData {
        QueryAccess query_access;
        BrowseAccess browse_access;
        AsynchAccess asynch_access;
        ConstraintLanguageAccess constraint_access;
        ObservationLoader observation_loader;
        ConsumerAccess consumer_access;
        SupplierAccess supplier_access;
};

struct AsynchException {
        QualifiedCodeStr exception_name;
        string message;
};

struct ObservationData {
        QualifiedCodeStr code;
        sequence<ObservationData> composite;
        sequence<ObservationData> qualifiers;
        sequence<any,1> value;
};

typedef ObservationData ObservationQualifier;

struct ObservationId {
        QualifiedCodeStr code;
        string opaque;
};

struct NameValuePair {
        QualifiedCodeStr name;
        any value;
};

struct Subscription {
        sequence<ObservedSubjectId> who;
        sequence<QualifiedCodeStr> what;
        sequence<ObservationQualifier> qualifier;
        sequence<NameValuePair> policy;
};

typedef string TimeStamp;  // ISO 8601 representation, with restrictions

struct TimeSpan {
        TimeStamp start_time;
        TimeStamp stop_time;
};


//
// CONSTANTS
//

// for TimeStamp fields
const string EARLIEST_TIME  = "1582-10-15T00:00:00Z"; // beginning of Gregorian calendar
const string LATEST_TIME  = "9999-12-31T23:59:59Z"; // max possible in ISO 8601 specification
const string TIME_WILDCARD  = "?"; // replace individual digits

const QualifiedCodeStr PARTIAL_RESULT = "DNS:omg.org/DsObservationAccess/PARTIAL_RESULT";
const QualifiedCodeStr COMPLETING_RESULT = "DNS:omg.org/DsObservationAccess/COMPLETING_RESULT";
const QualifiedCodeStr ASYNC_OBSERVATION_COUNT = "DNS:omg.org/DsObservationAccess/ASYNC_OBSERVATION_COUNT";
typedef unsigned long ASYNC_OBSERVATION_COUNT_type;

const QualifiedCodeStr EVENT_SOURCE_DOMAIN = "DNS:omg.org/DsObservationAccess/EVENT_SOURCE_DOMAIN";
const QualifiedCodeStr EVENT_SOURCE_SERVER_NAME = "DNS:omg.org/DsObservationAccess/EVENT_SOURCE_SERVER_NAME";
const QualifiedCodeStr EVENT_NAME = "DNS:omg.org/DsObservationAccess/EVENT_NAME";
const QualifiedCodeStr TEST_EVENT = "DNS:omg.org/DsObservationAccess/TEST_EVENT";
typedef long TEST_EVENT_type;

const QualifiedCodeStr TRADER_1_0_CONSTRAINT_LANGUAGE =
"DNS:omg.org/DsObservationAccess/TRADER_1_0_CONSTRAINT_LANGUAGE";
const QualifiedCodeStr OCL_1_1_CONSTRAINT_LANGUAGE = "DNS:omg.org/DsObservationAccess/OCL_1_1_CONSTRAINT_LANGUAGE";

const QualifiedCodeStr COAS_OBSERVATION_ID = "DNS:omg.org/DsObservationAccess/COAS_OBSERVATION_ID";
typedef ObservationId COAS_OBSERVATION_ID_type;
```

```
//
// TYPEDEFS
//

typedef long EndpointId;

typedef string ConstraintExpression;

typedef QualifiedCodeStr ConstraintLanguage;

typedef NameValuePair QueryPolicy;

typedef long ServerCallId;

typedef long ClientCallId;


//
// SEQUENCES
//

typedef sequence<AtomicObservationRemote> AtomicObsRemoteSeq;

typedef sequence<ConstraintLanguage> ConstraintLanguageSeq;

typedef sequence<EndpointId> EndpointIdSeq;

typedef sequence<ObservationData> ObservationDataSeq;

typedef sequence<ObservationId> ObservationIdSeq;

typedef sequence<ObservationQualifier> ObservationQualifierSeq;

typedef sequence<ObservationRemote> ObservationRemoteSeq;

typedef sequence<ObservedSubjectId> ObservedSubjectIdSeq;

typedef sequence<ObservedSubject> ObservedSubjectSeq;

typedef sequence<QualifiedCodeStr> QualifiedCodeStrSeq;

typedef sequence<QueryPolicy> QueryPolicySeq;

typedef sequence<Subscription> SubscriptionSeq;


//
// EXCEPTIONS
//

exception DuplicateCodes {
        QualifiedCodeStrSeq codes;
};

exception DuplicateIds {
        ObservedSubjectIdSeq ids;
};

exception DuplicateOids {
        ObservationIdSeq oids;
};

exception DuplicatePolicies {
        QueryPolicySeq policies;
};

exception DuplicateQualifiers {
        ObservationQualifierSeq qualifiers;
};

exception InvalidCodes {
        QualifiedCodeStrSeq codes;
};
```

```
exception InvalidEndpointId {
        EndpointIdSeq endpoint_ids;
};

exception InvalidConstraint {
        string constraint;
};

exception InvalidIds {
        ObservedSubjectIdSeq ids;
};

exception InvalidOids {
        ObservationIdSeq oids;
};

exception InvalidPolicies {
        QualifiedCodeStrSeq policies;
};

exception InvalidQualifiers {
        QualifiedCodeStrSeq qualifiers;
};

exception InvalidTimeSpan {
        TimeSpan span;
};

exception MaxConnectionsExceeded {
        unsigned long max_connections;
};

exception NotImplemented {
};

exception NoSubscription {
};


//
// INTERFACES
//

// ABSTRACT FACTORY INTERFACE

interface AbstractFactory {
        readonly attribute long max_connections;
        readonly attribute EndpointIdSeq current_connections;
};

// ABSTRACT MANAGED OBJECT INTERFACE

interface AbstractManagedObject {
        void done ( );
};

// ACCESS COMPONENT INTERFACE

interface AccessComponent {
        readonly attribute string coas_version;
        readonly attribute IdentificationComponent pid_service;
        readonly attribute TerminologyService terminology_service;
        readonly attribute TraderComponents trader_service;
        readonly attribute NamingContext naming_service;

        AccessComponentData get_components ( );

        QualifiedCodeStrSeq get_supported_codes (
                in unsigned long max_sequence,
                out QualifiedCodeIterator the_rest );

        QualifiedCodeStrSeq get_supported_qualifiers (
                        in QualifiedCodeStr code )
                raises (
                        InvalidCodes,
```

```
                NotImplemented );

        QualifiedCodeStrSeq get_supported_policies ( );

        QueryPolicySeq get_default_policies ( );

        TypeCode get_type_code_for_observation_type (
                in QualifiedCodeStr observation_type)
            raises (
                InvalidCodes,
                NotImplemented );

        boolean are_iterators_supported ( );

        TimeStamp get_current_time ( );
};

// ASYNCH ACCESS INTERFACE

interface AsynchAccess : AccessComponent {

        ServerCallId count_observations (
            in ObservedSubjectIdSeq who,
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in QueryPolicySeq policy,
            in ClientCallId client_call_id,
            in AsynchCallback client_callback );

        ServerCallId get_observation (
            in ObservationId observation_id,
            in ClientCallId client_call_id,
            in AsynchCallback client_callback );

        ServerCallId get_observations (
            in ObservationIdSeq observation_ids,
            in ClientCallId client_call_id,
            in AsynchCallback client_callback );

        ServerCallId get_observations_by_time (
            in ObservedSubjectId who,
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in unsigned long max_sequence,
            in ClientCallId client_call_id,
            in AsynchCallback client_callback );

        ServerCallId get_observations_by_qualifier (
            in ObservedSubjectIdSeq who,
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in unsigned long max_sequence,
            in ClientCallId client_call_id,
            in AsynchCallback client_callback );

        ServerCallId get_observations_with_policy (
            in ObservedSubjectIdSeq who,
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in QueryPolicySeq policy,
            in unsigned long max_sequence,
            in ClientCallId client_call_id,
            in AsynchCallback client_callback );

        void cancel_get (
            in ServerCallId server_call_id );
};

// ASYNCH CALLBACK INTERFACE

interface AsynchCallback {

        void put_observations (
```

```
            in ObservationDataSeq as_sequence,
            in ObservationDataIterator as_iterator,
            in ClientCallId client_call_id,
            in QualifiedCodeStrSeq result_status );

        void put_exception (
            in ClientCallId client_call_id,
            in AsynchException the_exception );
};

// OBSERVATION REMOTE INTERFACE

interface ObservationRemote : AbstractManagedObject {
        readonly attribute QualifiedCodeStr observation_code;

        TimeSpan get_observation_time ( );

        ObservedSubject get_observed_subject ( );

        ObservationRemote get_root_observation ( );

        ObservationData get_path_from_root ( );

        ObservationQualifierSeq get_all_qualifiers ( );

        ObservationQualifierSeq get_qualifiers (
                in QualifiedCodeStrSeq qualifier_names )
            raises (
                InvalidCodes );

        boolean is_this_root ( );

        boolean is_this_atomic ( );
};

// ATOMIC OBSERVATION REMOTE INTERFACE

interface AtomicObservationRemote : ObservationRemote {

        ObservationData get_observation_data ( );

        ObservationData get_observation_data_with_policy (
            in QueryPolicySeq policy );
};

// BROWSE ACCESS INTERFACE

interface BrowseAccess : AccessComponent {

        ObservedSubject get_observed_subject (
                in ObservedSubjectId who )
            raises (
                InvalidIds );

        ObservedSubjectSeq get_observed_subjects (
                in ObservedSubjectIdSeq who )
            raises (
                InvalidIds,
                DuplicateIds );

        ObservedSubject get_observed_subject_for_observation_id (
                in ObservationId observation_id )
            raises (
                InvalidOids );

        ObservedSubjectSeq get_observed_subjects_for_observation_ids (
                in ObservationIdSeq observation_ids )
            raises (
                InvalidOids,
                DuplicateOids );

        unsigned long count_observations (
                in ObservedSubjectIdSeq who,
                in QualifiedCodeStrSeq what,
                in TimeSpan when,
                in ObservationQualifierSeq qualifier,
```

```
                in QueryPolicySeq policy )
            raises (
                InvalidIds,
                DuplicateIds,
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan,
                InvalidQualifiers,
                DuplicateQualifiers,
                InvalidPolicies,
                DuplicatePolicies );

        ObservationRemote get_observation (
                in ObservationId observation_id )
            raises (
                InvalidOids );

        ObservationRemoteSeq get_observations (
                in ObservationIdSeq observation_ids )
            raises (
                InvalidOids,
                DuplicateOids );

        ObservationRemoteSeq get_observations_by_time (
                in ObservedSubjectId who,
                in QualifiedCodeStrSeq what,
                in TimeSpan when,
                in unsigned long max_sequence,
                out ObservationRemoteIterator the_rest )
            raises (
                InvalidIds,
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan );

        ObservationRemoteSeq get_observations_by_qualifier (
                in ObservedSubjectIdSeq who,
                in QualifiedCodeStrSeq what,
                in TimeSpan when,
                in ObservationQualifierSeq qualifier,
                in unsigned long max_sequence,
                out ObservationRemoteIterator the_rest )
            raises (
                InvalidIds,
                DuplicateIds,
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan,
                InvalidQualifiers,
                DuplicateQualifiers );

        ObservationRemoteSeq get_observations_with_policy (
                in ObservedSubjectIdSeq who,
                in QualifiedCodeStrSeq what,
                in TimeSpan when,
                in ObservationQualifierSeq qualifier,
                in QueryPolicySeq policy,
                in unsigned long max_sequence,
                out ObservationRemoteIterator the_rest )
            raises (
                InvalidIds,
                DuplicateIds,
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan,
                InvalidQualifiers,
                DuplicateQualifiers,
                InvalidPolicies,
                DuplicatePolicies );
};

// COMPOSITE OBSERVATION REMOTE INTERFACE

interface CompositeObservationRemote : ObservationRemote {

        unsigned long count_observations (
```

```
            in QueryPolicySeq search_depth_policy )
        raises (
            InvalidPolicies );

    ObservationRemoteSeq get_observations_by_time (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
        raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan );

    ObservationRemoteSeq get_observations_by_qualifier (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
        raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan,
            InvalidQualifiers,
            DuplicateQualifiers );

    ObservationRemoteSeq get_observations_with_policy (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in QueryPolicySeq policy,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
        raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan,
            InvalidQualifiers,
            DuplicateQualifiers,
            InvalidPolicies,
            DuplicatePolicies );

    AtomicObsRemoteSeq get_leaf_observations ( );

    AtomicObsRemoteSeq get_leaf_observations_by_time (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
        raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan );

    AtomicObsRemoteSeq get_leaf_observations_by_qualifier (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
        raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan,
            InvalidQualifiers,
            DuplicateQualifiers );

    AtomicObsRemoteSeq get_leaf_observations_with_policy (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in QueryPolicySeq policy,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
        raises (
```

```
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan,
                InvalidQualifiers,
                DuplicateQualifiers,
                InvalidPolicies,
                DuplicatePolicies );

        AtomicObsRemoteSeq get_leaf_observations_by_value_type (
                in QualifiedCodeStrSeq what,
                in TimeSpan when,
                in ObservationQualifierSeq qualifier,
                in QualifiedCodeStr value_type,
                in unsigned long max_sequence,
                out ObservationRemoteIterator the_rest )
            raises (
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan,
                InvalidQualifiers,
                DuplicateQualifiers );

        ObservationDataSeq get_relations_toward_root (
            in QualifiedCodeStrSeq relation_name );

        ObservationDataSeq get_relations_away_from_root (
            in QualifiedCodeStrSeq relation_name );
};

// CONSTRAINT LANGUAGE ACCESS INTERFACE

interface ConstraintLanguageAccess : AccessComponent {
        readonly attribute ConstraintLanguageSeq supported_languages;

        ObservationDataSeq get_by_constraint (
                in ConstraintExpression constraint,
                in QueryPolicySeq policy,
                in unsigned long max_sequence,
                out ObservationDataIterator the_rest )
            raises (
                InvalidConstraint,
                InvalidPolicies,
                DuplicatePolicies );
};

// CONSUMER ACCESS INTERFACE

interface ConsumerAccess : AbstractFactory, AccessComponent {

        EventConsumer create_consumer ( )
            raises (
                MaxConnectionsExceeded );

        EventConsumer get_consumer_by_id (
                in EndpointId endpoint_id )
            raises (
                InvalidEndpointId );
};

// EVENT CONSUMER INTERFACE

interface EventConsumer : AbstractManagedObject, PushConsumer {
        readonly attribute EndpointId endpoint_id;

        SubscriptionSeq obtain_subscriptions ( );

        void connect_push_supplier (
                in PushSupplier push_supplier )
            raises (
                CosEventChannelAdmin::AlreadyConnected );

        PushSupplier get_connected_supplier ( )
            raises (
                CosEventComm::Disconnected );
};
```

```
// EVENT SUPPLIER INTERFACE

interface EventSupplier : AbstractManagedObject, PushSupplier {
        readonly attribute EndpointId endpoint_id;

        QualifiedCodeStrSeq obtain_offered_codes ( );

        void connect_push_consumer (
                        in PushConsumer push_consumer )
                raises (
                        CosEventChannelAdmin::AlreadyConnected );

        PushConsumer get_connected_consumer ( )
                raises (
                        CosEventComm::Disconnected );

        void subscribe (
                        in SubscriptionSeq subscriptions )
                raises (
                        CosEventComm::Disconnected );

        SubscriptionSeq describe_subscriptions ( )
                raises (
                        NoSubscription );

        void generate_test_event (
                        in ClientCallId clientId )
                raises (
                        CosEventComm::Disconnected );
};

// OBSERVATION DATA ITERATOR INTERFACE

interface ObservationDataIterator : AbstractManagedObject {

        unsigned long max_left ( );

        boolean next_n (
                in unsigned long n,
                out ObservationDataSeq observation_data_seq );
};

// OBSERVATION LOADER INTERFACE

interface ObservationLoader : AccessComponent {

        void load_observations (
                in ObservationDataSeq observations );
};

// OBSERVATION REMOTE INTERFACE

//      This interface is defined after AsynchCallBack and before AtomicObservationRemote

// OBSERVATION REMOTE ITERATOR INTERFACE

interface ObservationRemoteIterator : AbstractManagedObject {

        unsigned long max_left ( );

        boolean next_n (
                in unsigned long n,
                out ObservationRemoteSeq observation_remote_seq );
};

// OBSERVED SUBJECT INTERFACE

interface ObservedSubject : AbstractManagedObject {
        readonly attribute ObservedSubjectId observed_subject_id;

        unsigned long count_observations (
                        in QualifiedCodeStrSeq what,
                        in TimeSpan when,
                        in ObservationQualifierSeq qualifier,
                        in QueryPolicySeq policy )
                raises (
```

```
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan,
            InvalidQualifiers,
            DuplicateQualifiers,
            InvalidPolicies,
            DuplicatePolicies );

ObservationRemoteSeq get_observations_by_time (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
      raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan );

ObservationRemoteSeq get_observations_by_qualifier (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
      raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan,
            InvalidQualifiers,
            DuplicateQualifiers );

ObservationRemoteSeq get_observations_with_policy (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in ObservationQualifierSeq qualifier,
            in QueryPolicySeq policy,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
      raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan,
            InvalidQualifiers,
            DuplicateQualifiers,
            InvalidPolicies,
            DuplicatePolicies );

ObservationRemoteSeq get_root_observations (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
      raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan );

AtomicObsRemoteSeq get_leaf_observations (
            in QualifiedCodeStrSeq what,
            in TimeSpan when,
            in unsigned long max_sequence,
            out ObservationRemoteIterator the_rest )
      raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan );

ObservationRemote get_any_observation (
            in QualifiedCodeStrSeq what,
            in TimeSpan when )
      raises (
            InvalidCodes,
            DuplicateCodes,
            InvalidTimeSpan );

ObservationRemote get_first_observation (
```

```
                in QualifiedCodeStrSeq what,
                in TimeSpan when )
        raises (
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan );

    ObservationRemote get_last_observation (
                in QualifiedCodeStrSeq what,
                in TimeSpan when )
        raises (
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan );

    ObservationRemoteSeq get_candidate_observations (
                in QualifiedCodeStrSeq what,
                in TimeSpan when,
                in ObservationQualifierSeq qualifier,
                in unsigned long max_sequence,
                out ObservationRemoteIterator the_rest )
        raises (
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan,
                InvalidQualifiers,
                DuplicateQualifiers );

    ObservationRemoteSeq get_exact_observation_types (
                in QualifiedCodeStrSeq what,
                in TimeSpan when,
                in unsigned long max_sequence,
                out ObservationRemoteIterator the_rest )
        raises (
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan );
};

// QUALIFIED CODE ITERATOR INTERFACE

interface QualifiedCodeIterator : AbstractManagedObject {

    unsigned long max_left ( );

    boolean next_n (
        in unsigned long n,
        out QualifiedCodeStrSeq codes );
};

// QUERY ACCESS INTERFACE

interface QueryAccess : AccessComponent {

    unsigned long count_observations (
                in ObservedSubjectIdSeq who,
                in QualifiedCodeStrSeq what,
                in TimeSpan when,
                in ObservationQualifierSeq qualifier,
                in QueryPolicySeq policy )
        raises (
                InvalidIds,
                DuplicateIds,
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan,
                InvalidQualifiers,
                DuplicateQualifiers,
                InvalidPolicies,
                DuplicatePolicies );

    ObservationData get_observation (
                in ObservationId observation_id )
        raises (
                InvalidOids );
```

```
        ObservationDataSeq get_observations (
                in ObservationIdSeq observation_ids )
            raises (
                InvalidOids,
                DuplicateOids );

        ObservationDataSeq get_observations_by_time (
                in ObservedSubjectId who,
                in QualifiedCodeStrSeq what,
                in TimeSpan when,
                in unsigned long max_sequence,
                out ObservationDataIterator the_rest )
            raises (
                InvalidIds,
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan );

        ObservationDataSeq get_observations_by_qualifier (
                in ObservedSubjectIdSeq who,
                in QualifiedCodeStrSeq what,
                in TimeSpan when,
                in ObservationQualifierSeq qualifier,
                in unsigned long max_sequence,
                out ObservationDataIterator the_rest )
            raises (
                InvalidIds,
                DuplicateIds,
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan,
                InvalidQualifiers,
                DuplicateQualifiers );

        ObservationDataSeq get_observations_with_policy (
                in ObservedSubjectIdSeq who,
                in QualifiedCodeStrSeq what,
                in TimeSpan when,
                in ObservationQualifierSeq qualifier,
                in QueryPolicySeq policy,
                in unsigned long max_sequence,
                out ObservationDataIterator the_rest )
            raises (
                InvalidIds,
                DuplicateIds,
                InvalidCodes,
                DuplicateCodes,
                InvalidTimeSpan,
                InvalidQualifiers,
                DuplicateQualifiers,
                InvalidPolicies,
                DuplicatePolicies );
    };

    // SUPPLIER ACCESS INTERFACE

    interface SupplierAccess : AbstractFactory, AccessComponent {

        EventSupplier create_supplier ( )
            raises (
                MaxConnectionsExceeded );

        EventSupplier get_supplier_by_id (
                in EndpointId endpoint_id )
            raises (
                InvalidEndpointId );
    };

};

#endif  // _DS_OBSERVATION_ACCESS_IDL_
```

A.2 DsObservationValue

```
// File: DsObservationValue.idl

#ifndef _DS_OBSERVATION_VALUE_IDL_
#define _DS_OBSERVATION_VALUE_IDL_

#include “DsObservationAccess.idl”

#pragma prefix “omg.org”

module DsObservationValue
{


    //
    // EXTERNAL TYPEDEFS
    //

    typedef TerminologyServices::ConceptCode  ConceptCode;
    typedef NamingAuthority::QualifiedNameStr  QualifiedCodeStr;

    typedef DsObservationAccess::AbstractManagedObject
   AbstractManagedObject;

    // DateTime : ObservationValue;
    typedef DsObservationAccess::TimeStamp   DateTime;

    // TimeSpan : ObservationValue;
    typedef DsObservationAccess::TimeSpan    TimeSpan;

    // Person : ObservationValue;
    typedef DsObservationAccess::ObservedSubjectId  Person;


    //-------------------------------------------------
    // NoInformation
    //-------------------------------------------------

    // NoInformation : ObservationValue;
    struct NoInformation {
          QualifiedCodeStr reason;
          string text_description;
    };
    const QualifiedCodeStr NO_INFORMATION =
   “DNS:omg.org/DsObservationValue/NO_INFORMATION”;



    //-------------------------------------------------
    // Text Types
    //-------------------------------------------------

    // PlainText : ObservationValue;
    typedef string PlainText;
```

```
// UniversalResourceIdentifier : ObservationValue;
struct UniversalResourceIdentifier {
      ConceptCode protocol;
      string address;
};

// PhysicalLocationDescription : ObservationValue;
typedef string PhysicalLocationDescription;


//--------------------------------------------------
// Coded Types
//--------------------------------------------------

// CodedElement : ObservationValue;
typedef TerminologyServices::QualifiedCodeInfo CodedElement;

// LooselyCodedElement : ObservationValue;
struct LooselyCodedElement {
      string text;
      TerminologyServices::CodingSchemeId coding_scheme_id;
      TerminologyServices::VersionId version_id;
};


//--------------------------------------------------
// Multimedia
//--------------------------------------------------

typedef sequence<octet> Blob;

interface MultimediaIterator  : AbstractManagedObject {

      unsigned long max_left ( );

      boolean next_n (
            in unsigned long n,
            out Blob multimedia_part );
};

// Multimedia : ObservationValue;
struct Multimedia {
      string content_type;
      string other_mime_header_fields;
      Blob a_blob;
      unsigned long long total_size;
      MultimediaIterator the_iterator;
};


//--------------------------------------------------
// Measurements Types
//--------------------------------------------------
```

```
        // Numeric : ObservationValue;
        struct Numeric {
            QualifiedCodeStr units;
            float value;
        };

        // Range : ObservationValue;
        struct Range {
            QualifiedCodeStr units;
            float lower;
            float upper;
        };

        // Ratio : ObservationValue;
        struct Ratio {
            float numerator;
            float denominator;
        };

        struct XYPair {
            float x;
            float y;
        };
        typedef sequence<XYPair> XYPairSeq;

        interface CurveIterator  : AbstractManagedObject {

            unsigned long max_left ( );

            boolean next_n (
                in unsigned long n,
                out XYPairSeq curve_part );
        };

        // Curve : ObservationValue;
        struct Curve {
            XYPairSeq xy_pairs;
            QualifiedCodeStr x_units;
            QualifiedCodeStr y_units;
            unsigned long long total_size;
            CurveIterator the_iterator;
        };

    };

    #endif // _DS_OBSERVATION_VALUE_IDL_
```

A.3 DsObservationTimeSeries

```
// File: DsObservationTimeSeries.idl

#ifndef _DS_OBSERVATION_TIME_SERIES_IDL_
#define _DS_OBSERVATION_TIME_SERIES_IDL_

#include “DsObservationAccess.idl”
```

```
module DsObservationTimeSeries
{

    //
    // EXTERNAL TYPEDEFS
    //

    typedef DsObservationAccess::AbstractManagedObject AbstractManagedObject;
    typedef DsObservationAccess::NameValuePair  NameValuePair;
    typedef DsObservationAccess::QueryPolicy  QueryPolicy;
    typedef DsObservationAccess::QueryPolicySeq QueryPolicySeq;
    typedef DsObservationAccess::ObservationQualifierSeq ObservationQualifierSeq;
    typedef DsObservationAccess::QualifiedCodeStr QualifiedCodeStr;
    typedef DsObservationAccess::TimeStamp TimeStamp;
    typedef DsObservationAccess::TimeSpan  TimeSpan;

    typedef sequence < QualifiedCodeStr > QualifiedCodeStrSeq;

    //-------------------------------------------------------
    // Time Types
    //-------------------------------------------------------

    // TimeDelta : ObservationValue;
    struct TimeDelta {
        float delta;  // calculated with constants below, NOT with calendaring
        QualifiedCodeStr units;
    };

    // approximations for time deltas, NOT for calendaring
    //  all units here are seconds. Use scaling as necessary for units of TimeDelta
    const float YEAR     = 31557600.0;  // 60*60*24*365.25
    const float MONTH   = 2629800.0;   // 60*60*24*365.25/12
    const float DAY          = 86400.0;    // 60*60*24
    const float HOUR         = 3600.0;    // 60*60
    const float MINUTE         = 60.0;     // 60
    const float SECOND          = 1.0;     // 1
    const float MILLISECOND  = 0.001;  // 1/1000


    typedef NameValuePair Filter;
    typedef sequence < Filter > FilterSeq;

    enum ValueSeqType {
        OtherSeqDataType, OctetType, ShortType,
        LongType, LongLongType, FloatType, DoubleType
    };

    union ValueSeq switch ( ValueSeqType ) {
        case OctetType       : sequence < octet > octet_seq;
        case ShortType       : sequence < short > short_seq;
        case LongType        : sequence < long > long_seq;
        case LongLongType : sequence < long long > long_long_seq;
        case FloatType       : sequence < float > float_seq;
        case DoubleType      : sequence < double > double_seq;
```

```
        case OtherSeqDataType : any the_any;
};

typedef sequence < QualifiedCodeStr,1 > OptionalCodeSeq;
typedef sequence < float,1 > OptionalFloatSeq;

interface TimeSeriesIterator  : AbstractManagedObject {
        unsigned long max_left ( );

        boolean next_n (
                in unsigned long n,
                out ValueSeq curve_part );
};

// TimeSeries : ObservationValue;
struct TimeSeries {
        TimeDelta sample_period;
        ValueSeq values;
        unsigned long long total_size;
        TimeSeriesIterator the_iterator;
};

exception OutOfRange { };

exception NotImplemented { };

exception FilterNotSupported { };

exception NoValidValues { };

struct TimeSeriesRemoteAttributes {
        QualifiedCodeStr code;
        QualifiedCodeStr units;
        OptionalCodeSeq accuracy;
        OptionalFloatSeq precision;
        OptionalFloatSeq corner_frequency;
        OptionalFloatSeq highest_frequency;
        TimeSpan time_span;
        TimeDelta time_delta;
        unsigned long long total_size;
        QualifiedCodeStrSeq supported_filters;
        QueryPolicySeq supported_policies;
};

// TimeSeriesRemote : ObservationValue;
interface TimeSeriesRemote : AbstractManagedObject {
        readonly attribute QualifiedCodeStr code;
        readonly attribute QualifiedCodeStr units;
        readonly attribute OptionalCodeSeq accuracy;
        readonly attribute OptionalFloatSeq precision;
        readonly attribute OptionalFloatSeq corner_frequency;
        readonly attribute OptionalFloatSeq highest_frequency;
        readonly attribute TimeSpan time_span;
        readonly attribute TimeDelta time_delta;
        readonly attribute unsigned long long total_size;
```

```
readonly attribute QualifiedCodeStrSeq supported_filters;
readonly attribute QueryPolicySeq supported_policies;
readonly attribute ValueSeqType default_value_type;

TimeSeriesRemoteAttributes get_attributes ( );

float get_sample_number (
        in unsigned long long index,
        out ObservationQualifierSeq qualifiers )
    raises (
        OutOfRange );

float get_sample (
        in TimeStamp time_stamp,
        out ObservationQualifierSeq qualifiers )
    raises (
        OutOfRange );

TimeSeries get_snippet (
        in TimeSpan time_span,
        out ObservationQualifierSeq qualifiers )
    raises (
        OutOfRange );

float get_max (
        in TimeSpan time_span )
    raises (
        OutOfRange,
        NoValidValues );

float get_min (
        in TimeSpan time_span )
    raises (
        OutOfRange,
        NoValidValues );

float get_mean (
        in TimeSpan time_span )
    raises (
        OutOfRange,
        NoValidValues );

float get_median (
        in TimeSpan time_span )
    raises (
        OutOfRange,
        NoValidValues );

TimeSeries get_resampled (
        in TimeSpan time_span,
        in TimeDelta sample_rate,
        in QueryPolicySeq policy,
        out ObservationQualifierSeq qualifiers )
    raises (
        NotImplemented );
```

```
            TimeSeries get_rescaled (
                    in TimeSpan time_span,
                    in float scale_factor,
                    in QueryPolicySeq policy,
                    out ObservationQualifierSeq qualifiers )
                raises (
                    NotImplemented );

            TimeSeries get_resampled_rescaled (
                    in TimeSpan time_span,
                    in TimeDelta sample_rate,
                    in float scale_factor,
                    in QueryPolicySeq policy,
                    out ObservationQualifierSeq qualifiers )
                raises (
                    NotImplemented );

            TimeSeries get_filtered (
                    in TimeSpan time_span,
                    in FilterSeq filters,
                    in QueryPolicySeq policy,
                    out ObservationQualifierSeq qualifiers )
                raises (
                    NotImplemented,
                    FilterNotSupported );
        };

    };

    #endif // _DS_OBSERVATION_TIME_SERIES_IDL_
```

A.4 DsObservationRelations

```
// file DsObservationRelations.idl

#ifndef _DS_OBSERVATION_RELATIONS_IDL_
#define _DS_OBSERVATION_RELATIONS_IDL_

#pragma prefix “omg.org”

#include “DsObservationAccess.idl”

module DsObservationRelations {

    typedef DsObservationAccess::QualifiedCodeStr QualifiedCodeStr;

    // all relations are collections of observations (composite observations)
    typedef DsObservationAccess::ObservationData RELATION_type;

// from CEN/TC 251/N98-116, table A.5

// CEN description names translated according to the following rules:
//replace “/” with “_”
//replace space with nothing, Capitalizing next word
//replace apostrophe, periods, etc. with nothing


// produces /is produced by healthcare activity produces result, report, study product
    const QualifiedCodeStr Produces = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/Produces”;
    const QualifiedCodeStr IsProducedBy = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsProducedBy”;
```

```
// is documented by /documents healthcare activity is documented by note (3.15)
      const QualifiedCodeStr Documents = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/Documents";
      const QualifiedCodeStr IsDocumentedBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsDocumentedBy";

//is reported within /reports about property is reported within report (3.17)
      const QualifiedCodeStr Reports = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/Reports";
      const QualifiedCodeStr IsReportedBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsReportedBy";

//describes /is described by graphic property (3.22) describes graphic object (3.21)
      const QualifiedCodeStr Describes = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/Describes";
      const QualifiedCodeStr IsDescribedBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsDescribedBy";

//is identified within /incorporates graphic object is identified within study product (3.20)
      const QualifiedCodeStr IsIdentifiedWithin = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsIdentifiedWithin";
      const QualifiedCodeStr IsIncorporatedBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsIncorporatedBy";

//is derived from /is source for graphic property is derived from study product
      const QualifiedCodeStr IsSourceFor = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsSourceFor";
      const QualifiedCodeStr IsDerivedFrom = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsDerivedFrom";

//is compared to /is reference for situation, document is compared to situation, document
      const QualifiedCodeStr IsComparedTo = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsComparedTo";
      const QualifiedCodeStr IsReferenceFor = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsReferenceFor";

//is recorded against /is recorded against family history of x is recorded against no evidence of x (note 3)
      const QualifiedCodeStr IsRecordedAgainst = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsRecordedAgainst";

//superseds /is superseded by clinical state supersedes clinical state (note 4)
      const QualifiedCodeStr Supercedes = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/Supercedes";
      const QualifiedCodeStr IsSupercededBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsSupercededBy";


//organizational links

//is framework for /is framed in contact is framework for situation, document
      const QualifiedCodeStr IsFrameworkFor = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsFrameworkFor";
      const QualifiedCodeStr IsFramedBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsFramedBy";

//has phase /is phase of healthcare activity has phase healthcare (sub)activity
      const QualifiedCodeStr HasPhase = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasPhase";
      const QualifiedCodeStr IsPhaseOf = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsPhaseOf";

//is next phase wrt /has next phase healthcare activity is next phase wrt healthcare (sibling) activity
      const QualifiedCodeStr HasNextPhase = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasNextPhase";
      const QualifiedCodeStr IsNextPhaseWRT = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsNextPhaseWRT";


//is associate to /is associate to condition is associate to condition
      const QualifiedCodeStr IsAssociateTo = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsAssociateTo";

//is assigned to /is setting for situation is assigned to problem
      const QualifiedCodeStr IsAssignedTo = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsAssignedTo";
      const QualifiedCodeStr IsSettingFor = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsSettingFor";

//is interpretation of/ is interpreted as condition is interpretation of findings, report
      const QualifiedCodeStr IsInterpretationOf = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsInterpretationOf";
      const QualifiedCodeStr IsInterpretedAs = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsInterpretedAs";

//has progress /is progress of condition has progress condition (e.g. convalescence)
      const QualifiedCodeStr HasProgress = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasProgress";
      const QualifiedCodeStr IsProgressOf = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsProgressOf";

//has cause /is cause of condition has cause condition
      const QualifiedCodeStr HasCause = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasCause";
      const QualifiedCodeStr IsCauseOf = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsCauseOf";

//co-exists with /co-exists with condition co-exist with condition
      const QualifiedCodeStr CoExistsWith = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/CoExistsWith";

//is evidence for /has evidence finding is evidence for diagnosis
      const QualifiedCodeStr HasEvidence = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasEvidence";
      const QualifiedCodeStr IsEvidenceFor = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsEvidenceFor";

//triggers /is triggered by presence of prosthesis triggers risk state
      const QualifiedCodeStr Triggers = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/Triggers";
      const QualifiedCodeStr IsTriggeredBy = "DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsTriggeredBy";
```

```
//has goal /is goal of healthcare activity has goal achievable situation
        const QualifiedCodeStr HasGoal = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasGoal”;
        const QualifiedCodeStr IsGoalOf = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsGoalOf”;

//has motivation /is motivation for healthcare activity has motivation current situation
        const QualifiedCodeStr HasMotivation = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasMotivation”;
        const QualifiedCodeStr IsMotivationFor = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsMotivationFor”;

//has consequence /is consequence of healthcare activity, event has consequence situation (e.g. outcome)
        const QualifiedCodeStr HasConsequence = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasConsequence”;
        const QualifiedCodeStr IsConsequenceOf = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsConsequenceOf”;


//circumstantial links


//has topic /is topic for informing has topic record component
        const QualifiedCodeStr HasTopic = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasTopic”;
        const QualifiedCodeStr IsTopicFor = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsTopicFor”;

//has target /is target of informing has target person
        const QualifiedCodeStr HasTarget = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasTarget”;
        const QualifiedCodeStr IsTargetOf = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsTargetOf”;

//provides information about /is reported by person provides information about record component
        const QualifiedCodeStr ProvidesInformationAbout = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/ProvidesInformationAbout”;

//has circumstances /is circumstance for support activity has circumstance home circumstances
        const QualifiedCodeStr HasCircumstances = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/HasCircumstances”;
        const QualifiedCodeStr IsCircumstanceOf = “DNS:omg.org/DsObservationAccess/relation/CENTC251N98116/IsCircumstanceOf”;

};

#endif // _DS_OBSERVATION_RELATIONS_IDL_
```

A.5 DsObservationQualifiers

```
// file DsObservationQualifiers.idl

#ifndef _DS_OBSERVATION_QUALIFIERS_IDL_
#define _DS_OBSERVATION_QUALIFIERS_IDL_

#pragma prefix “omg.org”

#include “DsObservationAccess.idl”

module DsObservationQualifiers {

typedef DsObservationAccess::QualifiedCodeStr  QualifiedCodeStr;
typedef DsObservationAccess::TimeStamp        TimeStamp;

const QualifiedCodeStr COAS_OBSERVATION_ID = “DNS:omg.org/DsObservationAccess/COAS_OBSERVATION_ID”;

        // all the qualifiers listed here from HL7 are defined with
        // subcomponents in HL7 2.3, so they all have type ObservationData (composite observations)
typedef DsObservationAccess::ObservationData COMPOSITE_OBSERVATION_type;

        // naming convention:
        //start with “DNS:omg.org/DsObservationAccess/HL72.3/”
        //add the HL7 segment name, like OBX or PID, plus a slash
        //take HL7 data element names from HL7 v2.3 standard distribution,
        //appendix A, (APPA.doc), table A.6 DATA ELEMENT NAMES,
        //translated according to the following rules:
        //replace “/” with “_”
        //replace space with nothing, capitalizing next word
        //omit apostrophe, periods, parentheses, and other punctuation
        //to name subcomponents, additional slashes can follow the component names
        //see SpecimenSourceBodySite at bottom for example

        // see HL7 descriptions for composite returned by each of these data elements.


// clinical times;
```

```
const QualifiedCodeStr Date_TimeOfTheObservation = “DNS:omg.org/DsObservationAccess/HL72.3/OBX/Date_TimeOfTheObservation”;
const QualifiedCodeStr EventOnsetDate_Time = “DNS:omg.org/DsObservationAccess/HL72.3/PEO/EventOnsetDate_Time”;
const QualifiedCodeStr OrderEffectiveDate_Time = “DNS:omg.org/DsObservationAccess/HL72.3/ORC/OrderEffectiveDate_Time”;
const QualifiedCodeStr ProcedureDate_Time = “DNS:omg.org/DsObservationAccess/HL72.3/PR1/ProcedureDate_Time”;
const QualifiedCodeStr RequestedDate_Time = “DNS:omg.org/DsObservationAccess/HL72.3/OBR/RequestedDate_Time”;
const QualifiedCodeStr VerificationDate_Time = “DNS:omg.org/DsObservationAccess/HL72.3/IN1/VerificationDate_Time”;
const QualifiedCodeStr ActionDate_Time = “DNS:omg.org/DsObservationAccess/HL72.3/GOL/ActionDate_Time”;
const QualifiedCodeStr AttestationDate_Time = “DNS:omg.org/DsObservationAccess/HL72.3/DG1/AttestationDate_Time”;
const QualifiedCodeStr TranscriptionDate_Time = “DNS:omg.org/DsObservationAccess/HL72.3/TXA/TranscriptionDate_Time”;

// roles

const QualifiedCodeStr PatientIDExternalID = “DNS:omg.org/DsObservationAccess/HL72.3/PID/PatientIDExternalID”;
const QualifiedCodeStr PatientIDInternalID = “DNS:omg.org/DsObservationAccess/HL72.3/PID/PatientIDInternalID”;
const QualifiedCodeStr OrderingProvider = “DNS:omg.org/DsObservationAccess/HL72.3/OBR/OrderingProvider”;
const QualifiedCodeStr ProducerID = “DNS:omg.org/DsObservationAccess/HL72.3/OBX/ProducerID”;
const QualifiedCodeStr CollectorIdentifier = “DNS:omg.org/DsObservationAccess/HL72.3/OBR/CollectorIdentifier”;
const QualifiedCodeStr ResponsibleObserver = “DNS:omg.org/DsObservationAccess/HL72.3/OBX/ResponsibleObserver”;
const QualifiedCodeStr Technician = “DNS:omg.org/DsObservationAccess/HL72.3/OBR/Technician”;
const QualifiedCodeStr PrincipalResultInterpreter = “DNS:omg.org/DsObservationAccess/HL72.3/OBR/PrincipalResultInterpreter”;

// from OBR (orders)

const QualifiedCodeStr SpecimenSource = “DNS:omg.org/DsObservationAccess/HL72.3/OBR/SpecimenSource”;
const QualifiedCodeStr ReasonForStudy = “DNS:omg.org/DsObservationAccess/HL72.3/OBR/ReasonForStudy”;
const QualifiedCodeStr DiagnosticServiceSectionID = “DNS:omg.org/DsObservationAccess/HL72.3/OBR/DiagnosticServiceSectionID”;

// from OBX (results)

const QualifiedCodeStr AbnormalFlags = “DNS:omg.org/DsObservationAccess/HL72.3/OBX/AbnormalFlags”;
const QualifiedCodeStr ObservationMethod = “DNS:omg.org/DsObservationAccess/HL72.3/OBX/ObservationMethod”;
const QualifiedCodeStr Units = “DNS:omg.org/DsObservationAccess/HL72.3/OBX/Units”;
const QualifiedCodeStr ReferencesRange = “DNS:omg.org/DsObservationAccess/HL72.3/OBX/ReferencesRange”;
const QualifiedCodeStr ObservationIdentifier = “DNS:omg.org/DsObservationAccess/HL72.3/OBX/ObservationIdentifier”;

// from PV1

const QualifiedCodeStr PatientLocation = “DNS:omg.org/DsObservationAccess/HL72.3/PV1/PatientLocation”;

// note that elements of HL7 composites can be individually identified with this COAS naming standard.
// e.g.  SpecimenSource is listed in the OBR definitions above, and one segment
// of SpecimenSource, like Body Site, can have its own name.

const QualifiedCodeStr SpecimenSourceBodySite = “DNS:omg.org/DsObservationAccess/HL72.3/OBR/SpecimenSource/BodySite”;
typedef QualifiedCodeStr SpecimenSourceBodySite_type;

};

#endif // _DS_OBSERVATION_QUALIFIERS_IDL_
```

A.6 DsObservationPolicies

```
// file DsObservationPolicies.idl

#ifndef _DS_OBSERVATION_POLICIES_IDL_
#define _DS_OBSERVATION_POLICIES_IDL_

#pragma prefix “omg.org”

#include “DsObservationTimeSeries.idl”

module DsObservationPolicies {

    typedef DsObservationAccess::QualifiedCodeStr QualifiedCodeStr;
    typedef DsObservationAccess::TimeStamp       TimeStamp;

    const QualifiedCodeStr SEARCH_DEPTH_POLICY = “DNS:omg.org/DsObservationAccess/policy/SEARCH_DEPTH_POLICY”;
    typedef short SearchDepthPolicyType;
        const SearchDepthPolicyType SEARCH_DEPTH_ONLY_ROOT = 0x0;
        const SearchDepthPolicyType SEARCH_DEPTH_DEEPEST_POSSIBLE = 0xFFFF; // default

    const QualifiedCodeStr RETURN_DEPTH_POLICY = “DNS:omg.org/DsObservationAccess/policy/RETURN_DEPTH_POLICY”;
    typedef QualifiedCodeStr ReturnDepthPolicyType;
```

```
        const ReturnDepthPolicyType RETURN_DEPTH_ROOT_ONLY = “DNS:omg.org/DsObservationAccess/policy/RETURN_DEPTH_ROOT_ONLY”;
        const ReturnDepthPolicyType RETURN_DEPTH_ALL = “DNS:omg.org/DsObservationAccess/policy/RETURN_DEPTH_ALL”;
        const ReturnDepthPolicyType RETURN_DEPTH_ALL_LEAVES = “DNS:omg.org/DsObservationAccess/policy/RETURN_DEPTH_ALL_LEAVES”;
        const ReturnDepthPolicyType RETURN_DEPTH_LEAVES_OF_MATCHED =
“DNS:omg.org/DsObservationAccess/policy/RETURN_DEPTH_LEAVES_OF_MATCHED”;
        const ReturnDepthPolicyType RETURN_DEPTH_MATCHED_ONLY =
“DNS:omg.org/DsObservationAccess/policy/RETURN_DEPTH_MATCHED_ONLY”;
        const ReturnDepthPolicyType RETURN_DEPTH_MATCHED_AND_DOWN =
“DNS:omg.org/DsObservationAccess/policy/RETURN_DEPTH_MATCHED_AND_DOWN”; // default

        const QualifiedCodeStr SEARCH_SYNONYMOUS_CODES_POLICY =
“DNS:omg.org/DsObservationAccess/policy/SEARCH_SYNONYMOUS_CODES_POLICY”;
        typedef QualifiedCodeStr SearchSynonymousCodesPolicyType;
        const SearchSynonymousCodesPolicyType SEARCH_SYNONYMOUS_CODES_FALSE =
“DNS:omg.org/DsObservationAccess/policy/SEARCH_SYNONYMOUS_CODES_FALSE”;
        const SearchSynonymousCodesPolicyType SEARCH_SYNONYMOUS_CODES_TRUE =
“DNS:omg.org/DsObservationAccess/policy/SEARCH_SYNONYMOUS_CODES_TRUE”; // default

        const QualifiedCodeStr RETURN_OBSERVATION_VALUES_POLICY =
“DNS:omg.org/DsObservationAccess/policy/RETURN_OBSERVATION_VALUES_POLICY”;
        typedef QualifiedCodeStr ReturnObservationValuesPolicyType;
        const ReturnObservationValuesPolicyType RETURN_NO_OBSERVATION_VALUES =
“DNS:omg.org/DsObservationAccess/policy/RETURN_NO_OBSERVATION_VALUES”;
        const ReturnObservationValuesPolicyType RETURN_OBSERVATION_VALUES =
“DNS:omg.org/DsObservationAccess/policy/RETURN_OBSERVATION_VALUES”; // default

        const QualifiedCodeStr SHORTCIRCUIT_SEARCH_CODES_ON_SUCCESS_POLICY =
“DNS:omg.org/DsObservationAccess/policy/SHORTCIRCUIT_SEARCH_CODES_ON_SUCCESS_POLICY”;
        typedef boolean ShortcircuitSearchCodesOnSuccessPolicyType;
        const ShortcircuitSearchCodesOnSuccessPolicyType SHORTCIRCUIT_SEARCH_CODES_ON_SUCCESS_FALSE = FALSE;  // default
        const ShortcircuitSearchCodesOnSuccessPolicyType SHORTCIRCUIT_SEARCH_CODES_ON_SUCCESS_TRUE  = TRUE;

        const QualifiedCodeStr SEARCH_SYNONYMOUS_IDS_POLICY =
“DNS:omg.org/DsObservationAccess/policy/SEARCH_SYNONYMOUS_IDS_POLICY”;
        typedef boolean SearchSynonymousIdsPolicyType;
        const SearchSynonymousIdsPolicyType SEARCH_SYNONYMOUS_IDS_FALSE = FALSE;
        const SearchSynonymousIdsPolicyType SEARCH_SYNONYMOUS_IDS_TRUE  = TRUE; // default

        const QualifiedCodeStr SHORTCIRCUIT_SEARCH_IDS_ON_SUCCESS_POLICY =
“DNS:omg.org/DsObservationAccess/policy/SHORTCIRCUIT_SEARCH_IDS_ON_SUCCESS_POLICY”;
        typedef boolean ShortcircuitSearchIdsOnSuccessPolicyType;
        const ShortcircuitSearchIdsOnSuccessPolicyType SHORTCIRCUIT_SEARCH_IDS_ON_SUCCESS_FALSE = FALSE; // default
        const ShortcircuitSearchIdsOnSuccessPolicyType SHORTCIRCUIT_SEARCH_IDS_ON_SUCCESS_TRUE  = TRUE;

        const QualifiedCodeStr RETURN_ITEMS_IN_TIME_SPAN_POLICY =
“DNS:omg.org/DsObservationAccess/policy/RETURN_ITEMS_IN_TIME_SPAN_POLICY”;
        typedef QualifiedCodeStr ReturnItemsInTimeSpanPolicyType;
        const ReturnItemsInTimeSpanPolicyType RETURN_ITEMS_IN_TIME_SPAN_FIRST_ITEM_ONLY =
“DNS:omg.org/DsObservationAccess/policy/RETURN_ITEMS_IN_TIME_SPAN_FIRST_ITEM_ONLY”;
        const ReturnItemsInTimeSpanPolicyType RETURN_ITEMS_IN_TIME_SPAN_LAST_ITEM_ONLY =
“DNS:omg.org/DsObservationAccess/policy/RETURN_ITEMS_IN_TIME_SPAN_LAST_ITEM_ONLY”;
        const ReturnItemsInTimeSpanPolicyType RETURN_ITEMS_IN_TIME_SPAN_ALL_ITEMS =
“DNS:omg.org/DsObservationAccess/policy/RETURN_ITEMS_IN_TIME_SPAN_ALL_ITEMS”; // default

        const QualifiedCodeStr MATCHING_STRENGTH_POLICY = “DNS:omg.org/DsObservationAccess/policy/MATCHING_STRENGTH_POLICY”;
        typedef float MatchingStrengthPolicyType;
        const MatchingStrengthPolicyType MATCHING_STRENGTH_WEAKEST = 0.0F;
        const MatchingStrengthPolicyType MATCHING_STRENGTH_STRONGEST = 1.0F; // default

        const QualifiedCodeStr PARAM_CHECKING_POLICY = “DNS:omg.org/DsObservationAccess/policy/PARAM_CHECKING_POLICY”;
        typedef boolean ParamCheckingPolicyType;
        const ParamCheckingPolicyType PARAM_CHECKING_FALSE = FALSE;
        const ParamCheckingPolicyType PARAM_CHECKING_TRUE  = TRUE; // default

//
// QUALIFIER_RETURN_POLICY: see DsObservationQualifiers.idl for list of qualifiers
//

        const QualifiedCodeStr QUALIFIER_RETURN_POLICY = “DNS:omg.org/DsObservationAccess/policy/QUALIFIER_RETURN_POLICY”;
        typedef sequence<QualifiedCodeStr> QualifierReturnPolicyType;
        // two special codes for this policy;
        const QualifiedCodeStr QUALIFIER_RETURN_ALL = “DNS:omg.org/DsObservationAccess/policy/QUALIFIER_RETURN_ALL”;
        const QualifiedCodeStr QUALIFIER_RETURN_NONE = “DNS:omg.org/DsObservationAccess/policy/QUALIFIER_RETURN_NONE”; // default

        const QualifiedCodeStr QUALIFIER_NOT_TO_RETURN_POLICY =
“DNS:omg.org/DsObservationAccess/policy/QUALIFIER_NOT_TO_RETURN_POLICY”;
```

```
        typedef sequence<QualifiedCodeStr> QualifierNotToReturnPolicyType;

//
// RELATIONS_RETURN_POLICY: see DsObservationRelations.idl for list of relations
//

        const QualifiedCodeStr RELATIONS_RETURN_POLICY = "DNS:omg.org/DsObservationAccess/policy/RELATIONS_RETURN_POLICY";
        typedef sequence<QualifiedCodeStr> RelationsReturnPolicyType;
        // two special codes for this policy;
        const QualifiedCodeStr RELATIONS_RETURN_ALL = "DNS:omg.org/DsObservationAccess/policy/RELATIONS_RETURN_ALL";
        const QualifiedCodeStr RELATIONS_RETURN_NONE = "DNS:omg.org/DsObservationAccess/policy/RELATIONS_RETURN_NONE"; // default

        const QualifiedCodeStr RELATIONS_NOT_TO_RETURN_POLICY =
"DNS:omg.org/DsObservationAccess/policy/RELATIONS_NOT_TO_RETURN_POLICY";
        typedef sequence<QualifiedCodeStr> RelationsNotToReturnPolicyType;

        const QualifiedCodeStr RETURN_MOST_RECENT_N_OBSERVATIONS_POLICY =
"DNS:omg.org/DsObservationAccess/policy/RETURN_MOST_RECENT_N_OBSERVATIONS_POLICY";
        typedef unsigned long ReturnMostRecent_N_ObservationsPolicyType;
        const ReturnMostRecent_N_ObservationsPolicyType RETURN_MOST_RECENT_N_OBSERVATIONS_ALL = 0xFFFFFFFF; // default

        const QualifiedCodeStr TIME_SERIES_REMOTE_RESAMPLE_ALGORITHM_POLICY =
"DNS:omg.org/DsObservationAccess/policy/TIME_SERIES_REMOTE_RESAMPLE_ALGORITHM_POLICY";
        typedef sequence<QualifiedCodeStr> TimeSeriesRemoteResampleAlgorithmPolicyType;

        const QualifiedCodeStr TIME_SERIES_REMOTE_RETURN_TYPE_PREFERENCE_POLICY =
"DNS:omg.org/DsObservationAccess/policy/TIME_SERIES_REMOTE_RETURN_TYPE_PREFERENCE_POLICY";
        typedef DsObservationTimeSeries::ValueSeqType TimeSeriesRemoteReturnTypePreferencePolicyType;

        const QualifiedCodeStr RETURN_MAX_SEQUENCE_FOR_VALUE_POLICY =
"DNS:omg.org/DsObservationAccess/policy/RETURN_MAX_SEQUENCE_FOR_VALUE_POLICY";
        typedef unsigned long ReturnMaxSequenceForValuePolicyType;
        const ReturnMaxSequenceForValuePolicyType RETURN_MAX_SEQUENCE_FOR_VALUE_ALL = 0xFFFFFFFF; // default

        const QualifiedCodeStr IGNORE_UNMATCHABLE_QUALIFIERS_POLICY =
"DNS:omg.org/DsObservationAccess/policy/IGNORE_UNMATCHABLE_QUALIFIERS_POLICY";
        typedef boolean IgnoreUnmatchableQualifiersPolicyType;
        const IgnoreUnmatchableQualifiersPolicyType IGNORE_UNMATCHABLE_QUALIFIERS_TRUE = TRUE;
        const IgnoreUnmatchableQualifiersPolicyType IGNORE_UNMATCHABLE_QUALIFIERS_FALSE  = FALSE; // default

};

#endif // _DS_OBSERVATION_POLICIES_IDL_
```

Appendix BInteroperation

The Naming, Trader, PIDS and LQS Standards are considered building blocks and as such are of great value to COAS, hence, the following information is supplied in order to provide a level of understanding where each may play a role.

B.1 Naming/Trader

It is anticipated that the CORBA Naming and/or Trader Services may be used for acquiring pertinent information about the capabilities of a COAS compliant service. For these purposes the **AccessComponent** Interface has been defined. For the Naming Service the **naming_service** attribute will define the Naming Context.

For the **kind** field in **CosNaming:NameComponent** the following will be used:

- '**Query COAS**' - A COAS component that meets the conformance class of the same name.
- '**Browse COAS**' - A COAS component that meets the conformance class of the same name.
- '**ConstraintLanguage COAS**' - A COAS component that meets the conformance class of the same name.
- '**Asynchronous COAS**' - A COAS component that meets the conformance class of the same name.
- '**Supplier COAS**' - A COAS component that meets the conformance class of the same name.
- '**Consumer COAS**' - A COAS component that meets the conformance class of the same name.
- '**Loader COAS**' - A COAS component that meets the conformance class of the same name.

The following definitions are Service Types defined for COAS components for use by the Trader Service.

```
Service AccessComponent {
    Interface AccessComponent;
    Mandatory readonly property StringSeq components_implemented;
    Readonly attribute StringSeq pid_service;
    Readonly attribute StringSeq terminology_service;
    Readonly attribute StringSeq trader_service;
    Readonly attribute StringSeq naming_service;
    Readonly property String component_name;
    Readonly property String coas_version;
    Readonly property StringSeq supported_codes;
    Readonly property StringSeq supported_qualifiers;
    Readonly property StringSeq default_policies;
    Readonly property StringSeq supported_policies;
};
```

B.2 PIDS

The COAS specification has introduced the idea of an **ObservedSubject**, but has made the distinction that it lies outside the scope of this specification in order to allow this specification to be used in varying medical architectures. However, because an **ObservedSubject** can be a person(patient) we recognized the value in utilizing the PIDS specification in order to identify a person in an enterprise. We have an attribute in the **AccessComponent**, called **pid_service,** to refer to a PIDS service.

B.3 LQS

The COAS specification utilizes many of the concepts from the Lexicon Query Service (LQS) specification in order to provide a more dynamic and extensible specification. The COAS specification does not however mandate the use of any particular LQS but recognizes that it provides all the necessary interfaces for a client or server to attain information from coding schemes to assist in semantic interoperability at a coded level. We have also introduced the idea of an LQS terminology service via the **AccessComponent** interface attribute called **terminology_service** thereby providing a link to terminology services.

Appendix C Security Guidelines

The COAS interfaces may be used in many different environments with widely varying security requirements that range from no security to extreme security. For this reason the COAS interfaces do not expose any security information. COAS relies on the underlying CORBA infrastructure and services which provides all the security mechanisms needed without exposing it in the interfaces.

An attribute of security that of concern to many people is to maintain confidentiality of certain (sensitive) information about them. For COAS this implies being able to filter requests by:

- who is accessing the information,
- who the information is about,
- what information is being accessed.

Other common security concerns could be preventing unauthorized modification of data, tapping into communications to acquire sensitive information, and causing loss of service by over burdening a service.

CORBA Security provides robust mechanisms to address these and other concerns. Some of the security properties it does deal with includes authentication, authorization, encryption, audit trails, non-repudiation, etc. CORBA Security, in its default mode allows these security concerns to be addressed without the client and server software being aware of it. This is a powerful notion, allowing security policies to be created and enforced after applications and systems have been created and installed.

Other CORBA and CORBA Security features provide mechanisms for applications to extend these security capabilities. For example they can obtain credentials from the ORB and implement filters that can look at specific data passed to and returned from operations.

It is a requirement of the COAS to provide confidentiality of information that is stored about an individual. This requirement fuels the need for fine grained access control on clinical observations that are associated with identifiable observed subjects.

C.1 Security Requirements

For the COAS to be secure in its possible dissemination of information it needs to adhere to several requirements:

- The COAS needs to authenticate a client's principal identity, role, affiliation and other security attributes.
- The COAS needs to transmit information confidentially and with integrity.

The first requirement states that the entire COAS interface implementations must be able to identify a potential client. If it cannot authenticate a client, then the client may be severely limited in the particular requests that the COAS can service. The CORBA Security Service provides the mechanisms for a server to authenticate a client.

The second requirement provides for the confidentiality of the information. The client must communicate with the COAS using not only encryption to protect data, but signature as well, so as not to have data tampered with during communication. There is no sense in putting a Sensitivity level of “OwnerOnly” on an observation and have its value transmitted to the owner in the clear. The CORBA Security Service provides these capabilities, including SSL.

The problem is, How does one get CORBA to support this access policy model?

C.2 CORBA Security Overview

In an effort to keep the COAS interfaces security unaware, i.e. no extra visible security relevant parameters in methods, access policies must be adhered to from behind the interfaces. The CORBA security model offers several ways to apply security policies to method invocations.

The CORBA Security Specification (**CORBAsec**) is not a cookbook for using CORBA security in building applications. It is a specification of a general framework with which ORB vendors and application vendors can build a multitude of different security policy models. The **CORBAsec** also gives the interfaces which implementations of applications can use to access those security services that are supplied with a secure ORB.

A secure COAS implementation that can control access to specific observations must be aware of the security services offered by the ORB. This caveat also means that a client’s ORB may have to know the kind of ORB and the security services that is used by the COAS.

The CORBA security specification outlines a general security policy model. Although the specification is vague about which approach should be taken, it is specific enough to be able to choose from a couple of models that can be supported.

The CORBA security model bases itself on credentials and security domains. Credentials are data objects that contain attributes such as privileges, capabilities, and sensitivity levels, amongst others. Security domains are mappings from credentials to access rights. Credentials can be encrypted and signed to prevent tampering and to achieve a level of trust between client and server. CORBA credentials get passed with requests beneath the visible level of the interface. CORBA security services give the clients and servers the ability to authenticate/verify credentials in order to implement policies in the security domains.

Many different schemes, algorithms, services, and vendor implementations exist to provide implementation of security policies, and many different implementations of those schemes may be integrated into a CORBA compliant ORB. It is not the purpose of this specification to dictate the specific implementation of an ORB or the security services that should be used, but to outline the external requirements for the COAS implementation. These requirements and guidelines aid in selecting a secure ORB and the level of security functionality needed to implement the COAS access policy model.

C.3 Secure Interoperability Concerns

CORBA has built the communication bridge between distributed objects creating a interoperable environment that spans heterogeneous platforms and implementations. However, security adds another layer of complexity to the issue of interoperability. ORB implementations are neither required to include security services nor required to provide an interoperable mechanism of security services. However, a specification does exist for the target object to advertise, via the IOR, the security services that it supports and the services it requires from the client. Both the client and server ORBs must use compatible mechanisms of the same security technology.

The CORBA Common Secure Interoperability (CSI) Specification defines 3 levels of security functionality that ORBs may support. The levels are named, CSI Level 0, CSI Level 1, and CSI Level 2. Each level has increasing degrees of security functionality.

The CSI Level 0 supports identity based policies only and provides mechanisms for identity authentication and message protection with no privilege delegation. The CSI Level 1 adds unrestricted delegation. The CSI Level 2 can implement the entire CORBA Security Specification at Security Level 2.

Each CSI level is parameterized by mechanisms that can support the level of security functionality, such as SPKM for CSI Level 0, GSS Kerberos for CSI Level 0 or CSI Level 1, and CSI_ECMA for CSI Level 2. Future developments in security functionality and mechanism are not restricted, and mechanisms can be added to each level.

The ORB implementations may use different security technology with differing capabilities and underlying mechanisms, such as SSL, DCE, Kerberos, Sesame, or other standards. Choosing the ORB and its underlying security services will be critical to protecting COAS, and it will influence the implementation of the access policy that a secure COAS implementation must support.

For example, an ORB that only supports SPKM, i.e. CSI Level 0, can only authenticate clients and provide confidentiality and integrity of communication. It cannot support definition and use of security attributes beyond an access ID. Support for security attributes beyond an access ID require CSI Level 2. Therefore, using an ORB that only provides CSI Level 0 will require the COAS to maintain its own information on the credentials of clients.

Even if an ORB's security technology supports the definition of security attributes that can be delivered to the COAS, i.e. CSI Level 2, there are still concerns involving the trust between the client and the COAS.

C.4 Trust Models

The available trust models for the COAS are simplistic. Since the COAS is a communications end point and is not required to make requests to other services on a client's behalf, a delegation trust model is not needed. This simplifies the model and eliminates an absolute need for a CSI Level 1 or CSI Level 2 secure ORB (although they may use them).

There are two basic trust models for the COAS. If the COAS and its client are implemented using CSI Level 0 or CSI Level 1 ORBs, only the first trust model can be supported. If a CSI Level 2 ORB is used, both trust models can be supported. The trust models are:

1. The client's identity can and is trusted to be authenticated. However, the client is unable or untrusted to deliver the valid credentials.

2. The client is trusted to deliver the correct credentials.

In the first model, the client ORB is required to authenticate its principal (the user) and provide authentication information to the server ORB. The methods used to accomplish principal authentication is specific to the mechanisms (e.g. DCE or Kerberos) that the selected ORB supports. Management of those identities is also specific to the mechanism. The server ORB must have a compatible mechanism that verifies the authentication information and carries out mutual authentication of the client.

With this trust model, a secure COAS implementation must maintain and manage a map of identities to privilege attributes. CSI Level 0, 1, and 2 ORBs are able to support this trust model.

Even if the ORB has CSI Level 2 functionality, it may be a local policy that a COAS does not trust the credentials brought forth from an authenticated client. In that case, the COAS must maintain the map or use a default set of security attributes for requests from clients it does not trust.

In the second model, the client ORB is required to authenticate its principal and acquire its valid credentials. The methods used to accomplish principal authentication and acquisition of privilege attributes are specific to the mechanism that the selected ORB supports, such as DCE and Sesame. Management of those identities and attributes are also mechanism specific. A secure COAS installation using this trust model must take a careful look at that management scheme and operation, evaluate it, and decide to trust it. In such a scenario, the server ORB, which has CSI Level 2 functionality, automatically verifies the credentials on invocation.

A secure COAS built to the second model leaves management of identities and their attributes to the security services policy management system used by the ORB. The COAS may manage security attributes for the data itself.

A secure COAS built to the first model will have some scheme to manage trusted identities and their credentials. There is no interface or plan in the COAS to specify this kind of management.

C.5 CORBA Credentials

To adhere to the credential model that supports trait specific access policies, a set of credentials must contain privilege attributes such as the identity of the client, the role in which the client is actively represented, and the sensitivity level of information to which the client is allowed access. It will be the responsibility of a COAS implementation to advertise to potential client vendors the specifics of these attributes

and how to represent them externally. A client ORB needs to ascertain certain credentials about the user and must pass them to the COAS. An external representation of those credentials is needed so that credentials can be passed between client and server within the CORBA security services. The CORBA **Security** module defines the structure for this representation.

```
module Security {

   const SecurityAttributeType   AccessId  = 2;
   const SecurityAttributeType   Role      = 5;
   const SecurityAttributeType   Clearance = 7;

   struct SecAttribute {
      AttributeType     attribute_type;
      Opaque            defining_authority;
      Opaque            value;
   };
   typedef sequence<SecAttribute> AttributeList;
}
```

Listed above are the relevant pieces of the specification from the **Security** module that apply to externalizing credential information.

C.6 CORBA Security Domain Access Policy

In addition to a credential based scheme, CORBA defines security domains. The purpose of this section is to explain and illustrate the use of the standard CORBA security policy domain and the way in which it may be used to implement a security policy for the COAS. This section offers a recommendation to a COAS implementor in order to give a feel for the kinds of security policies a COAS implementation may need to support. It should also guide the implementor in evaluating a secure ORB and available security services.

A security domain governs security (access) policy for objects that are managed within that domain. In order to make scalable administration of security policy, these domains map sets of security credentials to certain sets of rights. A right is a sort of an internal security credential.

CORBA defines a standard set of rights that are granted to principals within a security domain. A security domain administrator manages that map through the **SecurityAdministration** module's **DomainAccessPolicy** interface. Access decision then can be based on a set of required rights and the rights granted to the client by the domain access policy, by virtue of the client's credentials.

ORB security service vendors will supply a security policy management infrastructure that implements the standard CORBA rights scheme. The COAS must use security services that can place different required rights on the COAS interfaces. Some ORB security services may allow a security domain to create special rights. However, CORBA defines a standard set of rights: get, set, and manage. This right set will suffice to handle the COAS.

In the model just described there is one security domain for all of the COAS components. The CORBA rights families scheme within a single security policy domain suffices to differentiate the security nature of the methods. More generally any number of domain models can be used, such as a separate security domain for each COAS component.

C.7 Request Content Based Policy

The CORBA standard domain access policy scheme only protects methods from invocation at the target and without regard to content of the request. The COAS needs a more fine grained access control in order to implement the content based access policy required (e.g. access policies for different observations). The COAS implementations must be made security aware to implement an access policy based on the value of arguments in a request. There are multiple ways to implement this policy using a secure CORBA implementation.

The CORBA Security Specification supplies two different schemes represented by an interface hierarchy, which are Security Level 1 and Security Level 2 (these should not be confused with CSI Levels 0, 1, and 2). These interfaces describe the level of security functionality that is available to security aware implementations.

Security Level 1

For the COAS to take advantage of CORBA security in order to implement its access policy model, the ORB must at least implement the CORBA Security Level 1 interfaces. A Security Level 1 compliant ORB supplies an interface to access the attributes of the credentials received from the client.

Using the **SecurityLevel1** interfaces, which is simplistic, gives the implementation of the COAS interfaces the ability to examine the client's credentials and compare them to the access control information that is managed as the access policy. However, the implementation of the COAS must be security aware.

```
module SecurityLevel1 {

   Current get_current();

   interface Current {
      Security::AttributeList get_attributes(
            in  Security::AttributeTypeList attributes
      );
   };
}
```

Using the Security Level 1 interfaces, each implementation of a COAS query interface must call the **get_attributes()** function on the **Current** pseudo object, examine the attributes, compare them to the access policy information, and make the access decision. If a COAS implementation chooses to use the Healthcare Resource Access Decision Facility, it constructs an appropriate resource name and operation name, and passes them to **ResourceAccessDecision::access_allowed()** along with the attributes received from **Current::get_attributes()**. Details on how COAS implementations must

use an HRAD Facility are provided in Section C.8, “Use of Healthcare Resource Access Decision Facility”. In the latter case, a COAS does not need to examine the attributes or implement any access decision logic. The COAS implementation should enforce the access decision according to the semantics of the particular COAS operation. It is the responsibility of the client's ORB to acquire the proper credentials securely. It is the responsibility of the server's ORB to authenticate the credentials received from the client, extract the security attributes from them, and make them available to the implementation through the **Current::get_attributes()** method.

Security Level 2

Using an ORB which supplies the Security Level 2 interfaces, the implementation can be somewhat free of making the access control decision in the implementation of the query interfaces. Having an implementation that is security unaware is attractive in CORBA, because security policy decisions can be made underneath the functionality, and they have the ability to be changed without retooling the application.

As with any framework, there are several ways in which to use the Security Level 2 interfaces. One approach could be to implement a replaceable security service for the access decision. Security Level 2 describes a method in which security can be enforced by the creation of an Access Decision object. The **AccessDecision** object would interact with a **DomainAccessPolicy** object to get effective rights and compare those to rights returned from the **RequiredRights** interface.

Some secure ORB implementations may allow the installation of specialized Access Decision objects to be used in conjunction with specialized **DomainAccessPolicy** objects. In the Security Level 2 interfaces, the specification implies access control only on the invocation of a method and not the contents of the request.

```
module SecurityReplaceable {

   interface AccessDecision {
      boolean access_allowed (
         in   SecuirytLvel2::CredentialList   red_list,
         in   CORBA::Object                   target,
         in   CORBA::Identifier               operation_name,
         in   CORBA::Identifier               interface_name
      );
   };
}
```

Currently, the **AccessDecision** object specified in the **SecurityReplaceable** module does not take the invocation **Request** as an argument. It only makes an access decision based on the credentials received from the client, the target object reference and operation name, and the target's interface name. This criteria is insufficient to implement the content based access policy, if needed by a COAS implementation to be automatically performed by the ORB.

Since the COAS requires access control on the contents of the method invocation (such as asking for the value of the HomePhone trait), this scheme of replacing these Security Level 2 components cannot be used. ORB security services that use the standard CORBA domain access policy may use third party implementations for these

components. This standard domain access policy functionality gives the COAS a high level of invocation protection that is orthogonal to the content based access policy. Some COAS servers may need the standard domain access policy functionality in addition to providing content based access policy. Therefore, another approach must be taken.

A content based access policy can be implemented in a Security Level 2 ORB by using an interceptor. A request level interceptor takes the **Request** as an argument and therefore, it can examine the content of the invocation arguments.

```
module CORBA {

   interface Interceptor { ... };
   interface RequestLevelInterceptor : Interceptor {
      void client_invoke( inout Request request );
      void target_invoke( inout Request request );
   };
}
```

Installing an interceptor on an ORB is ORB implementation specific, and each ORB vendor may have their own flavor of interceptors. The ORB calls the request level interceptor just before the invocation gets passed to the server implementation by using the **target_invoke()** operation. The interceptor uses the Dynamic Skeleton Interface (DSI) to examine values of the arguments of the invocation and make access decisions. These access decisions are also based on the credentials received from the client and the access policy. The interceptor will deny access to the invocation by raising an exception. The server's ORB will transmit this exception back to the client.

The use of the interceptor scheme frees the implementation of the COAS interfaces from the implementation of the access decision policy. If the access policy model changes, then the interceptor can be changed out without retooling the COAS implementation.

As awareness of the need for more powerful and flexible security policy management grows, more tools to create, manage, and analyze policy will come into existence. A COAS implementation relying on interceptors to implement its security policy may be able, with relative ease, to switch to using more robust policy services as they become developed.

C.8 Use of Healthcare Resource Access Decision Facility

Resource names used for obtaining access decisions from HRAD facility by COAS-compliant services, should be created in a predefined manner:

```
COAS_HRAC_Resource_Name ::=
'IDL:omg.org/DsObservationAccess' +
{‘ObservedSubjectId’, <ObservationData.observed_subject_id> } +
{‘QualifiedCodeStr’, <Stringified ObservationData.observation_type>}+
{‘TimeSpan’, <Stringified ObservationData.observation_time>}+
[{‘ObservationId’, <ObservationData.observation_id>}]
```

Text below explains the expression above in English.

If a COAS-compliant service uses Healthcare Resource Access Decision facility (HRAD), it shall:

- create HRAD resource names according to the following rules:

1. The “resource_naming_authority” data member of **ResourceName** shall adhere to the syntax of the **NamingAuthority::AuthorityIdStr** type. For the corresponding datum element of type **AuthorityId**, the value of authority shall be 'IDL'. The value of **naming_entity** shall be 'omg.org/DsObservationAccess'.

2. The first element of the **ResourceName** data member *resource_name_component_list* is mandatory. Its member *name_string* shall have a value of '**ObservedSubjectId**', and the value of *value_string* shall be the value of the *observed_subject_id* data member of the corresponding datum element of type **ObservationData** for the observation to be accessed.

3. The second element of the **ResourceName** data member *resource_name_component_list* is mandatory. Its member *name_string* shall have a value of '**QualifiedCodeStr**', and the value of *value_string* shall be the stringified, via **TerminologyServices::TranslationLibrary.qualified_code_to_name_str()**, value of the *observation_type* data member of the corresponding datum element of type **ObservationData** for the observation to be accessed.

4. The third element of the **ResourceName** data member *resource_name_component_list* is mandatory. Its member *name_string* shall have a value of '**TimeSpan**', and the value of the corresponding *value_string* shall be the value of the *observation_time* data member of the corresponding datum element of type **ObservationData** for the observation to be accessed.

5. The fourth element of the **ResourceName** data member *resource_name_component_list* is optional. If it is provided, its data member *name_string* shall have a value of ‘**ObservationId**’. The value of the corresponding *name_string* data member shall be the value of ‘**observation_id**’ of the corresponding datum element of type **ObservationData** for the observation to be accessed.

- Create HRAD operation name according to the following rules:

1. When serving invocations of operations that semantically mean “get”, operation in **DfResourceAccessDecision::access_allowed()** shall have value 'read'.

2. When serving invocations of operations that semantically mean “set”, operation in **DfResourceAccessDecision::access_allowed()** shall have value 'write'.
   - Obtain security attributes of the invoking principal via **SecurityLevel1::Current.get_attributes()** (See Section C.7, “Request Content Based Policy” or other means.
   - Obtain resource access decision(s) by invoking either **access_allowed()** or **multiple_access_allowed()** on **DfResourceAccessDecision::AccessDecision** interface.

- Enforce the decision according to the semantics of the operation the COAS-compliant service is serving.
- It is not mandated by this specification how exceptions caught during an attempt to invoke either **access_allowed()** or **multiple_access_allowed()** on **DfResourceAccessDecision::AccessDecision** interface are handled by a COAS-compliant service.

Appendix D Usage Patterns

There are a variety of scenarios for which patient observation data may need to flow between two systems or applications. A simple set of CORBA interfaces can be useful by deploying them in these different scenarios without having to redefine the interfaces for each scenario. Some of the factors determining how the interfaces may be used are:

- who initiates the conversation; is the connection temporary or permanent;
- who knows when and what should be sent for which patients;
- is the data coming from a human or machine observer;
- is the time span relative to a single encounter vs. a whole life time record;
- is the data going into a CDR/EMR or coming out;
- will it be used as one central database or distributed data resources; etc.

The subsections below will investigate some scenarios. One of the biggest determinants in these scenarios is who knows that a particular set of information needs to be passed between two applications. As you will see below, each scenario has a particular usefulness that depends on this issue.

D.1 Consumer Initiated

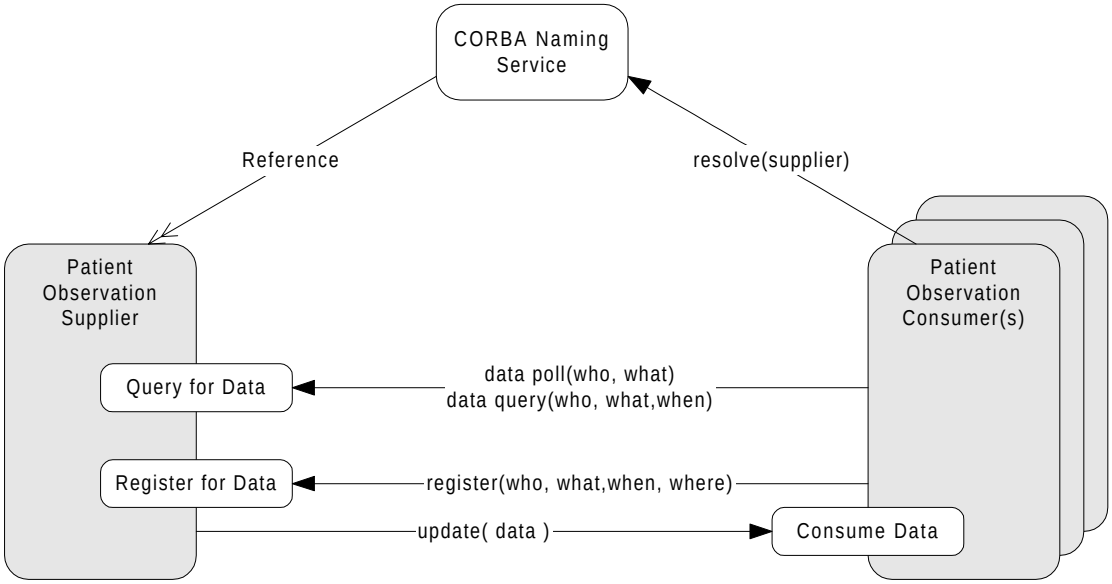

Figure 4-98 Data consumer initiated push and pull interaction models.

A supplier of patient observation data may need to allow clients to:

- poll for the current patient data (numeric vital signs and waveforms),
- query for data that has already been collected, and

- register for automatic updates at specified times or triggered by some other event.

The supplier may publish a reference to itself in a CORBA Naming Service for others (possibly many others) to access.

The arrows with solid heads in the diagram above represent the direction of one system calling another. The arrow with a wire head indicates the Patient Observation Supplier is in the CORBA Naming Service. The Patient Observation Consumer goes to the Naming Service (or any other valid mechanism) and gets an object reference to the supplier. The consumer then initiates any querying to, and registering with, the supplier. This mechanism would be used by an application that may come up with a user interface that allows the clinician to query for data or ask to be periodically/continuously updated.

The three interfaces are named with logical descriptions for what they do. See the specific interface sections for the actual name of the interfaces and a full description of their capabilities. Note that an observation supplier need only implement one or both interfaces.

Query for Data - This represents a CORBA interface that allows a client system/application to query for past patient observations or poll for the current patient information. This is a simple mechanism from the consumer's point of view since they only have to poll/query for data when they want it, although they must determine when to ask for the data. The polling is also simple for the supplier, but querying requires the storage of data to have occurred. This mechanism is more appropriate when the time that data is needed can not be predetermined.

Register for Data - This CORBA interface allows the client to register its Consume Data interface with the supplier of observation data to be updated with the indicated data and times. This is more complicated from the consumer's point of view since they have to implement a CORBA object. On the other hand the consumer does not have to deal with timers, etc. to determine when to poll for information. The supplier does not need to keep a data base of patient data for this mechanism but, they do need to keep a connection data base. This mechanism is best suited when the data availability can not be predetermined, such as needing data when an alarm or other event occurs.

Consume Data - This is the CORBA interface for the call back from the registration procedure that gets called with patient observation data.

The labels on the arrows contain pseudocode that specifies the kind of information that must be passed in each invocation. The actual information passed and the interfaces will be a lot more complicated than this simple picture in order to characterize the data fully and manage the registration.

who - Patients for which data is wanted. This may be specified by identifying patients by an identifier or by locations.

when - Times for which data is wanted. These could be specific times and/or events of interest. This is implied to be the current time or most recent data during polling.

what - The kinds of data wanted. This could be vitals signs, waveforms, alarm indications or other patient observations.

where - Where the data is going. This is implied for polling and queries since the data is returned to the system initiating the call.

The simplest and most straight forward way to access data is by polling and querying. The querying system only has to use the client side of CORBA. Registering for automatic updates requires more work including creating a CORBA object that can be called back. Most of the work for the registration capabilities is done by the service side.

D.2 Supplier Initiated

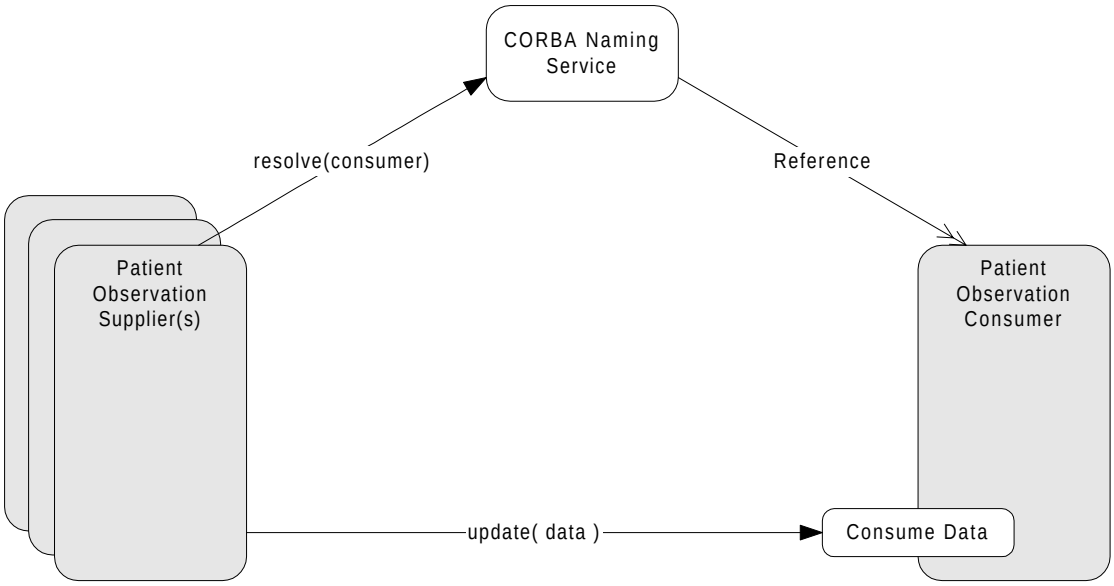

Figure 4-99 Supplier initiated push interaction model.

A consumer (sink) of patient observation data may need to allow clients to send (push) data to it. The consumer may publish a reference to itself in a CORBA Naming Service for others (possibly many others) to access it.

The supplier of the observation data can look up the consuming application in the CORBA Naming Service and send data to the consumer when the supplier deems necessary. An example where this scenario would be valid is when a nursing application or patient care management application needs to send nurse notes or manually collected vital signs to the EMR/CDR.

D.3 Third Party Initiated

In many cases, a system supplying observation data and a system consuming observation data do not know about each other. In these cases, a third party such as a System Administrator will set up and configure the connection between the two systems.

These are more useful ways when the two systems run in the background (continuously). For example, an ancillary system may need to send data to a Clinical Data Repository (CDR) or patient care management application on a periodic bases. Another example would be registering a nurse call system with a monitoring system in order to be notified of alarms of interest to that nurse.

In either of these cases, neither the supplier nor consumer know about each other. The System Administrator (or some other third party) will need to set up the connection between the two. The Patient Observation Consumer and Patient Observation Supplier would need to be in the CORBA Naming Service or the System Administration Application would need to get the object references through some other means.

D.4 Push Style

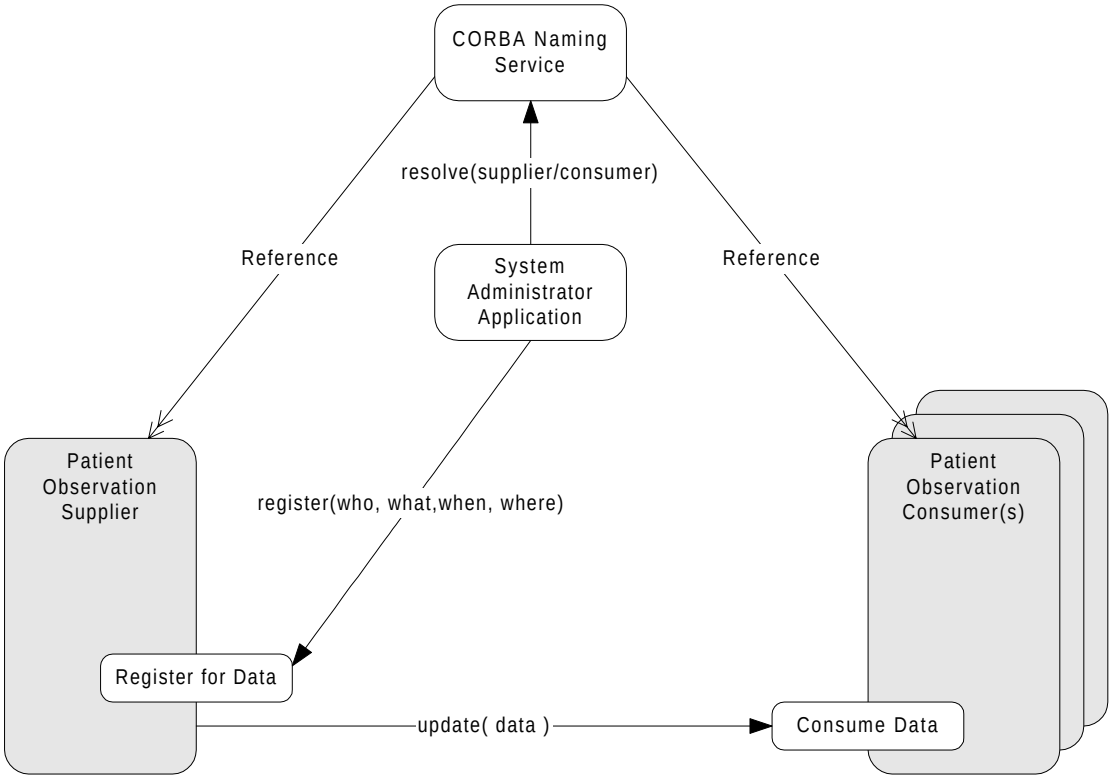

Figure 4-100 Third party interactions to set up a push style connection.

Figure 4-100 shows a slightly more complicated mechanism for registering a consumer with a data supplier. In this case, the consumer(s) need to implement the Consume Data interface. This works when the consumer is a data sink such as a data base.

The supplier only needs to implement the Register for Data interface. This is more complicated than just implementing the Query for Data interfaces since the supplier has to manage the set of consumers and the data base of the patient data. The supplier also has to monitor the timer and alarm events to know when the data should be sent to the consumer.

D.5 Pull Style

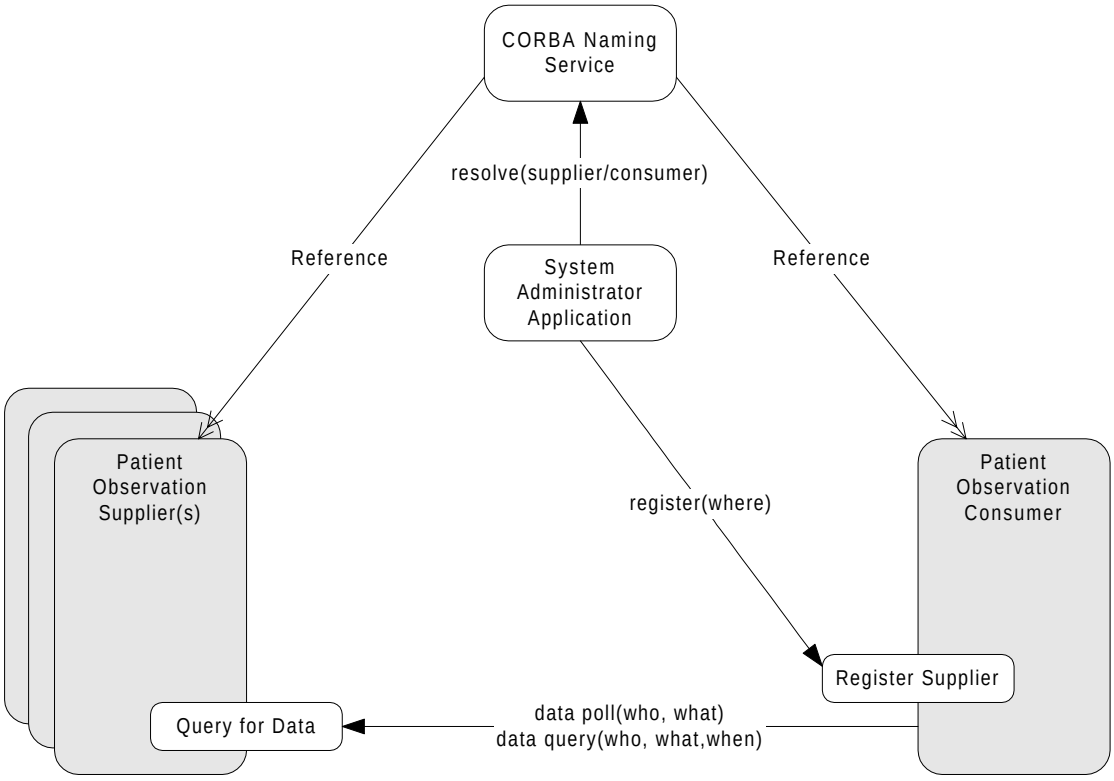

Figure 4-101 Third party interactions to set up a pull style connection.

Figure 4-101 shows another mechanism for registering a consumer with a data supplier. In this case, the consumer needs to implement the Register Supplier interface. The supplier only need implement the Query for Data interface. In many cases, this would be the simplest scenario for the supplier system to implement since it already has stored the data in a data base and needs to implement the logic to retrieve the data and return it to the caller.

This scenario adds a complication to the consumer since it now has to implement the Register Supplier interfaces and manage a set of suppliers from which to receive data.

D.6 Third Party Mediated

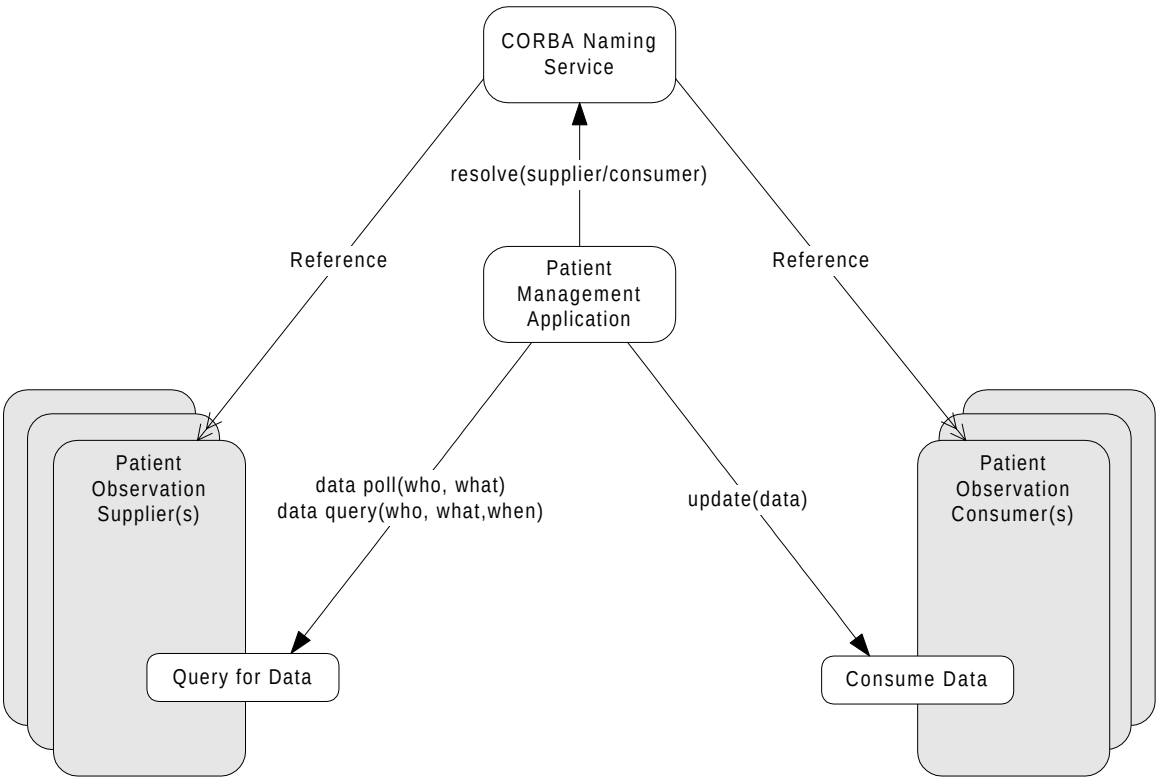

Figure 4-102 Third party mediator to convert pull style supplier to a push style consumer.

Figure 4-102 shows a scenario where the supplier and consumer have selected (maybe by necessity) to implement the simplest respective interfaces or at least non-compatible interfaces since neither can initiate the connection. A third party application mediates between the consumer and supplier. In some cases, this is a natural scenario since a Patient Management Application may be taking raw data from an instrument. The clinician would validate the data and then send the results to the CDR/EMR.

Another case may be an interface engine to bridge the two systems and the IT department (system administrator) would configure the interface engine directly.

Appendix E Usage Scenarios

E.1 Vital Signs Service

E.1.1 Nursing Station Scenario

A Nurse is doing his charting on a Clinical Information System (CIS). The CIS collects vital signs from the vital signs server (patient monitoring system) every minute.

The CIS polls the vital signs server every minute for the most representative vital signs values (median filter) over the last minute. This data is cached up for 24 hours for immediate access by the Nursing staff. Because this polling is done so often it is important the calls are efficient. For example it should only require a single call to acquire the data for all vital signs on all 16 patients in that unit.

The CIS also has the ability to show waveforms from the patient. Instead of storing the high volume of waveform data for the 24 hours it only requests them when a Nurse wants to view them.

The Nursing staff may sometimes want to see the very latest vital signs values, where as the stored data on the CIS is only one value per minute, and at any time the last value shown could be as much as 1 minute old. The CIS provides a function for the Nurse to request the very latest data. The CIS polls the vital signs server but asks for the very latest data available for each vital sign as long as it is no older then 15 seconds. The Nurse verifies these values with the monitoring system display and enters them into the patient record with a simple button push.

E.1.2 Doctor's Office Scenario

A Doctor has multiple patients admitted to a hospital and needs to make her rounds every day. Before going to the hospital she wants to review the patients condition over the last day.

A local application (or a web browser is used to download an applet which) connects up to the hospital intranet and queries the vital signs server for a 24 hour trend on the first patient. The trend is a sampling of the vital signs numerics (heart rate, blood pressure, etc.) over the past 24 hours. Since the vital signs may be collected continuously with changes on the order of every second or two (60*60*24=86400 samples per vital sign) it would take a long time to download. Instead the client application asks for only one sample every 5 minutes (12*24=288 samples) since the trend display area is only 288 pixels wide. A median filter is requested over each 5 minute period so that the most representative value is returned.

The Doctor notices a sudden drop in the blood pressure around 3:00 am and zooms in around that time. The application changes to a 30-minute view and does another query to the vital signs server. This time it asks for a trend over the 30 minutes with a resolution of 5 seconds.

The Doctor wants to see what the ECG and blood pressure waveforms are doing during this time, and so she changes views. The cursor was set at 3:05:20 am. When the Doctor changed to the waveform view the application queried the vital signs server for the waveforms around 3:05:20, requesting 20 seconds before through 20 seconds after that point in time. It centers the waveform on the screen, which shows a window of 10 seconds for each waveform.

After scrolling through the waveform the Doctor notices a short arrhythmia starting at 3:05:43. The doctor uses the application to see when other arrhythmias might have occurred through out the night, and sees a half dozen others.

She looks at a couple of them to make sure they really are problems and decides to put this patient high on her list to visit first during her rounds.

E.1.3 Remote Monitoring Scenario

A hospital has installed monitors throughout the enterprise, but realizes that most Nurses are not familiar with many of the difficulties that can be exposed with the monitor. They implement a central monitoring group (scope techs) that provides this functionality. Since there are so many monitors, they can not watch each one continuously, as is usually done with monitor techs.

The scope tech's applications are registered with the vital signs servers to be notified when alarms start and end. The application filters these alarms with a different algorithm for each vital sign in order to reduce false alarms. The alarms that get through the filter are displayed to the scope tech.

The application then polls the vital signs server for the waveforms (ECG, etc) starting at the beginning of the alarm event up to the present. This information is shown to the scope tech immediately. The application also registers with the vital signs server to be updated every second with the latest vital signs and the waveforms for the last second. As this data arrives the application appends the waveforms to that already displayed in a continuous manner.

It appears to the scope tech the data is being acquired and displayed continuously, but the data is always one second behind. This small delay is acceptable for the job of the scope tech. The delay is used so that only one packet of data is sent on the network per second, reducing the network bandwidth required.

E.1.4 Paging System Scenario

A hospital has a nurse paging system that is used for sending messages to nurses through out the day, as well as notifying them of code situations they may need to attend to immediately. They choose to connect the vital signs server to the paging system so that life threatening alarms can cause the responsible Nurse to be paged.

The paging system is registered with the vital signs server to have critical data pushed to it when certain events occur (life threatening arrythmias and apnea). Since there is a possibility of false alarms, other clinical information needs to be passed to the paging system as well so the Nurse can triage the severity of the alarm. A snap shot of the

waveform associated with the alarm (ECG or Respiration) is sent along with the latest vital signs values. Some Nurses carry large screen pagers that can display this extra data.

Due to the time criticality of the alarm, the data must be delivered to the Nurse quickly. From the point of view of the vital signs server it is just delivering (pushing) the requested data to a client at the times they registered interest in. It knows nothing about the client, except that it can accept the pushed data.

E.1.5 Logging System Scenario

Due to potential legal actions, a hospital has implemented an enterprise wide logging system of information that may be needed in case a law suit occurs. It does not have an electronic medical record system so it prints these out on paper that gets put into the patient's record.

The most critical information needed is when certain alarms occur, but information is also captured periodically during a shift. The period is determined by what unit they are on. The information collected includes an ECG snapshot of 7 seconds and certain vital signs (heart rate, oxygen saturation, blood pressure, and respiration rate), if they are available. Since the blood pressure is taking sporadically, only values within the last 15 minutes are included. All other vital signs are taken continuously and are included if a value exists within 5 seconds of the event.

There are several ways the logging system could get the information from the vital signs server - by polling, querying and registering.

Since the vital signs server keeps all data for 24 hours, the logging system could query for the information every 24 hours (or less). It could query for the times the alarms it is interested in had occurred through out the day. It could then query for the required vitals signs and ECG at these times and at the periodic times for that unit.

The logging system could be registered with the vital signs server to send the required vital signs and ECG at the periods in which data is logged for that unit. It could also register to have the same information sent when the alarms of interest occur.

Alternatively the logging system could poll for the needed vital signs and ECG at the periodic times assigned to that unit. At those same points in time it could query for which of the important alarms had occurred since the last period and query for the vital signs at those times.

Appendix F Client Implementation Examples

Following are some examples of how a client might access observations via the **DsObservationAccess** service. All codes, data, and clinical information are fabricated for illustration purposes.

F.1 Lipid Panel

Consider an example where a COAS client requests a lab result, using the **QueryAccess** component. The lab in question is a lipid panel for patientID “1234”, with the sample drawn on the morning of 11 Mar 1999.

For this example, assume the following definitions. First, there are several observation codes, one for a composite panel, and four individual measurements within the panel:

```
LIPID_PANEL// a battery of lipoproteins in blood sample
    TRIGLYCERIDES
    TOTAL_CHOLESTEROL
    LOW_DENSITY_LIPOPROTEIN
    HIGH_DENSITY_LIPOPROTEIN
```

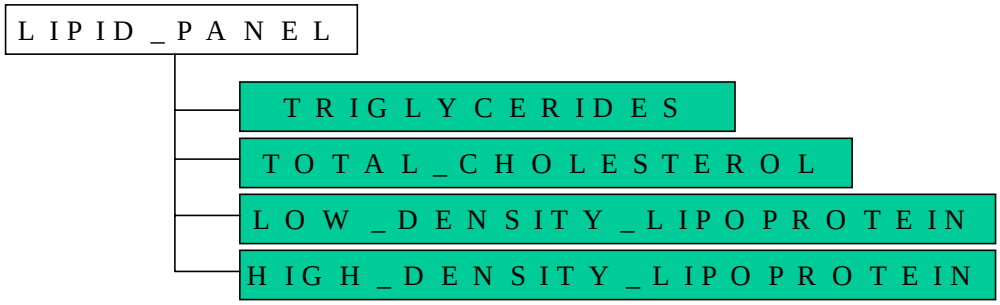

Figure 4-103 LIPID_PANEL is a composite observation with four elements.

That is, **LIPID_PANEL** is an **ObservationData** which contains other observations, so its **composite** field has four items while its **value** field has zero length. Meanwhile, the four contained observations are atomic observations. Their **composite** field is zero length, while their **value** field (a **CORBA::any**) is filled with a **DsObservationValue::Numeric** struct.

F.1.1 Qualifiers

Assume the following qualifier codes:

```
NORMAL_RANGE  // range for this measurement/gender
NORMALCY      // flag for this measurement
OBSERVATION_TIME   // time sample was drawn
RESULTS_AVAILABLE_TIME  // time result entered into system
```

Figure 4-104 NORMAL_RANGE is a qualifier which contains a Range struct within value.

Within the **DsObservationValue::Range** struct is a lower and upper bound. See **DsObservationValue** descriptions for more information about **Range**.

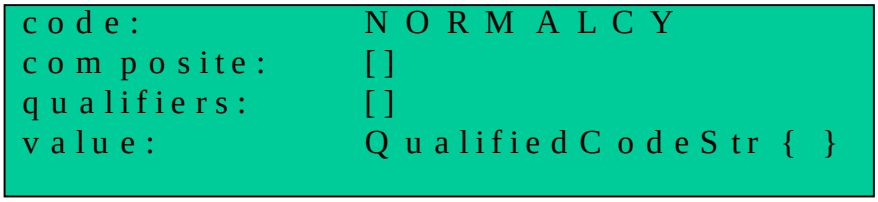

Figure 4-105 NORMALCY is a qualifier which contains a **QualifiedCodeStr** within value.

The enumeration of qualified codes for **NORMALCY** might include **NORMAL**, **ABNORMAL_HIGH**, **ABNORMAL_LOW**, and potentially other codes.

Figure 4-106 OBSERVATION_TIME is a qualifier which contains a TimeSpan within value.

The observation time can be a precise point in time, indicated by a **TimeSpan** with start_time = stop_time.

(ditto for **RESULTS_AVAILABLE_TIME**)

Finally, assume one more code, a value for units.

```
mg_PER_dL  // milligrams per deciliter
```

F.1.2 Request

The request might look something like the following, if we assume that a COAS object has been located and referenced as “myCoasServer” in a java syntax.

```
// “who” parameter
ObservedSubjectId who = new ObservedSubjectId(
    new AuthorityId( RegistrationAuthority.DNS, "myHospital.org/pids"),"1234" );

// “what” parameter
String[] what = new String[1];
what[0] = LIPID_PANEL;

// “when” parameter
TimeSpan when = new TimeSpan (
    "1999-03-11T00:00:00",
    "1999-03-11T11:59:00"
);

// “the_rest” parameter (a returned iterator, if # observations > max_sequence)
ObservationDataIteratorHolder() the_rest = new ObservationDataIteratorHolder();

ObservationData[] results = myCoasServer.get_observations_by_time(
    who,
    what,
    when,
    1000,    // max_sequence, largest number of observations allowed in returned sequence
    the_rest    // iterator for observations > max_sequence
);
```

F.1.3 Result

The result returned by the COAS server could look something like the following, depending on the default policies of the server. For this example, we assume the return of qualifiers **NORMAL_RANGE**, **NORMALCY**, **OBSERVATION_TIME**, and **RESULTS_AVAILABLE_TIME**. In other words, assume the default **QUALIFIER_RETURN_POLICY** contains these codes and no others which apply to the example observations.

In the example below, **Obs:<code>** indicates an **ObservationData** struct with **<code>** in the code field, with the other three fields of **ObservationData**, composite, qualifiers, and value, displayed in that order. Two brackets, “[]” indicate a sequence of length zero.

Indentation implies hierarchy, with leftmost items containing rightmost items. Initial capitals indicates a **DsObservationValue** struct name, like **Range**. These structs are found within the “value” field in an **ObservationData** (the value field is a **CORBA::any**).

```
Obs:LIPID_PANEL
```

```
composite:
   Obs:TRIGLYCERIDES
      composite: []
      qualifiers:
            Obs:NORMAL_RANGE
                  composite: []
                  qualifiers: []
                  value: Range { lower = 0, upper = 100 }
            Obs:NORMALCY
                  composite: []
                  qualifiers: []
                  value: QualifiedCode { ABNORMAL_HIGH }
      value: Numeric { value = 150, units = mg_PER_dL }
   Obs:TOTAL_CHOLESTEROL
      composite: []
      qualifiers:
            Obs:NORMAL_RANGE
                  composite: []
                  qualifiers: []
                  value: Range { lower = 0, upper = 200 }
            Obs:NORMALCY
                  composite: []
                  qualifiers: []
                  value: QualifiedCode { ABNORMAL_HIGH }
      value: Numeric { value = 220, units = mg_PER_dL }
   Obs:LOW_DENSITY_LIPOPROTEIN
      composite: []
      qualifiers:
            Obs:NORMAL_RANGE
                  composite: []
                  qualifiers: []
                  value: Range { lower = 0, upper = 130 }
            Obs:NORMALCY
                  composite: []
                  qualifiers: []
                  value: QualifiedCode { ABNORMAL_HIGH }
      value: Numeric { value = 150, units = mg_PER_dL }
   Obs:HIGH_DENSITY_LIPOPROTEIN
      ...
qualifiers:
   Obs:OBSERVATION_TIME
      value
         TimeSpan
            start_time = "1999-03-11T07:05:00-08"
            stop_time  = "1999-03-11T07:05:00-08"
   Obs:RESULTS_AVAILABLE_TIME
      value
         TimeSpan
            start_time = "1999-03-11T11:04:00-08"
            stop_time  = "1999-03-11T11:04:00-08"
value: []
```

F.2 Progress Note (XML)

Consider a COAS server which parses XML as an input qualifier, and returns XML documents as output. Just as with the previous example, the standard operations of **QueryAccess** are employed. The output is still a sequence of **ObservationData** items, with a single XML document as the string payload in the value field of an atomic observation.

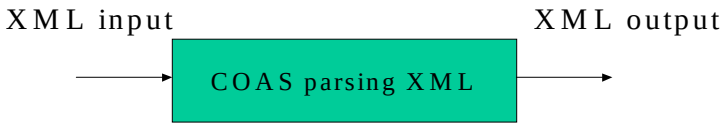

Figure 4-107 COAS server which parses incoming XML, and outputs XML.

This COAS server accepts XML input to create a template for matching. The example below illustrates an input document with XML fields as follows: document.type = “progress.note”, patient.id = “450023” and practitioner.id = “phys124”). The fields within the input document are matched, returning complete records which have matching information. Fields omitted from the input are considered “don’t care” for matching purposes.

Since both the input (qualifier) is an XML Progress Note and the output is an XML Progress Note, both input (qualifier) **ObservationData.code** and output **ObservationData.code** are the same: **PROGRESS_NOTE**.

For this Progress Note query example, assume the following full XML document format as shown:

```
<?xml version="1.0"?>
<!DOCTYPE LevelOne SYSTEM "LevelOne.dtd"[]>
<?xml-stylesheet type="text/xsl" href="himssdemo.xsl"?>
<LevelOne>
    <header>
        <document>
            <document.creation.date>
                1999-2-3T12:27:50
            </document.creation.date>
             <document.id>
                  <id.value>PRAAPN_CFN1999-02-03T12:27:51</id.value>
             </document.id>
             <document.originating.system>
                 <id.value>CFN</id.value>
                <organization.name>
```

```
            Sample HIMSS Hospital</organization.name>
     </document.originating.system>
     <document.originator.id>
         <id.value>VJ342</id.value>
     </document.originator.id>
     <document.state value="original"/>
    <document.title>
       Progress Note</document.title>
     <document.type value="progress.note"/>
</document>
<event>
       <event.id>
           <id.value></id.value>
       </event.id>
       <event.date>1999-2-3T12:27:51</event.date>
       <event.location.id>
           <id.value>4444444</id.value>
           <facility>
               <namespace.id>12345</namespace.id>
               <local.header>
                   DEPARTMENT OF FAMILY PRACTICE
               </local.header>
           </facility>
       </event.location.id>
</event>
<patient>
   <patient.id>
       <id.value>
           P013
       </id.value>
   </patient.id>
   <patient.name>
       <family.name>Presnell</family.name>
       <given.name>Tricia</given.name>
   </patient.name>
   <patient.date.of.birth>
       1992-09-14 00:00:00.0
   </patient.date.of.birth>
   <patient.sex value="female"/>
   <patient.address>
       <street.address>
           1944 Cone St. </street.address>
       <city>
       </city> <state.or.province>
       </state.or.province>
       <zip.or.postal.code>
       </zip.or.postal.code>
   </patient.address>
</patient>
<practitioner>
   <practitioner.id>
       <id.value>
           D3
       </id.value>
       <family.name>Ross </family.name>
```

```
                <given.name>Mark </given.name>
            </practitioner.id>
        </practitioner>
    </header>
    <body>
        <section>
            <section.title>Subjective</section.title>
            <paragraph>
                7 y.o. white female. Chief complaint: sore throat. Pt
complains of the onset yesterday afternoon of a sore throat. Mother
relates Pt had a fever to 104 F last night. She has been treating
with children's Tylenol since then, last dose 2 hours ago. No
headache, no abdominal pain. Nausea since yesterday evening, with
vomiting after breakfast this morning. No cough, no rhinorrhea, no
hoarseness. No dysuria or diarrhea. There are no sick contacts.
            </paragraph>
        </section>
        <section>
            <section.title>Objective</section.title>
            <paragraph>
                T 39.2C    BP 110/60 left arm, sitting    R 20    P
114  Allergies: None. General:  ill appearing 7 year old girl, non-
toxic, good eye contact, responsive to questions. HEENT: Eyes: EOMI,
pupils are equal, round, reactive to light, sclera are non-injected,
non-icteric Ears: tympanic membranes are pearly white bilaterally,
with good cones of light, and good landmarks, no otalgia. Nares: no
discharge, turbinates non-inflamed, no muco-pus..  Mouth:  There are
no gingival vesicular eruptions. Generalized swelling and erythema
of the pharynx.  Bilateral 3+ tonsils with moderate exudate.  Scarce
palatal petechiae
            </paragraph>
        </section>
        <section>
            <section.title>Assessment</section.title>
            <paragraph>
                Acute Pharyngitis.  R/O strep.
            </paragraph>
        </section>
        <section>
            <section.title>Plan of Care</section.title>
        </section>
        <section><section.title>Labs</section.title>
            <paragraph>
                strep screen
            </paragraph>
        </section>
        <section>
            <section.title>Rx</section.title>
            <paragraph>
                Penicillin 250mg, po, qid x 10 days
            </paragraph>
            <paragraph>
                Tylenol prn fever
            </paragraph>
            <paragraph>
```

```
                encourage po fluid
            </paragraph>
            <paragraph>
                RTC in 7 days or soon as worsens.
            </paragraph>
            <paragraph>
                Keep home from school, indoors until temp. less than
100 F for one full day.
            </paragraph>
        </section>
    </body>
</LevelOne>
```

F.2.1 Request

```
// assume
const QualifiedCodeStr PROGRESS_NOTE =
    "DNS:/omg.org/Sample/ProgressNote";

// “who” parameter
ObservedSubjectId who = new ObservedSubjectId(
     new AuthorityId( RegistrationAuthority.DNS, "myHospital.org/pids"),
     "450023"
);

// “what” parameter
String[] what = new String[1];
what[0] = PROGRESS_NOTE;

// “when” parameter (don’t care)
TimeSpan when = new TimeSpan(
     EARLIEST_TIME,
     LATEST_TIME
);

// “the_rest” parameter (a returned iterator, if # observations > max_sequence)
ObservationDataIteratorHolder() the_rest = new ObservationDataIteratorHolder();

// “qualifiers” parameter

// the following XML string is displayed on separate lines for readability
// assume that we have inputXML filled as
//

String inputXML =
<?xml version="1.0"?>
<!DOCTYPE LevelOne SYSTEM "LevelOne.dtd"[]>
<?xml-stylesheet type="text/xsl" href="himssdemo.xsl"?>
<LevelOne>
    <header>
        <document>
            <document.type value="progress.note"/>
        </document>
```

```
        <patient>
            <patient.id>
                <id.value>
                    450023
                </id.value>
            </patient.id>
        </patient>
        <practitioner>
            <practitioner.id>
                <id.value>
                    phys124
                </id.value>
            </practitioner.id>
        </practitioner>
    </header>
</LevelOne>
```

```
// put inputXML into an Any
CORBA.Any qualAny = orb.create_any();
qualAny.insert_string( inputXML );

ObservationData[] qualifiers = new ObservationData[1];
qualifiers[0] = new ObservationData(
    PROGRESS_NOTE,        // same code for input qualifier as output--an XML doc
    new ObservationData[0], // no composite members
    new ObservationData[0], // no qualifiers of this qualifier
    qualAny
);

// "the_rest" parameter (a returned iterator, if # observations > max_sequence)
ObservationDataIteratorHolder() the_rest = new ObservationDataIteratorHolder();

ObservationData[] results = myCoasServer.get_observations_by_qualifier(
    who,
    what,
    when,
    qualifiers,
    100,          // max_sequence, largest number of observations allowed in
   returned sequence
    the_rest      // iterator for observations > max_sequence
);
```

F.2.2 Result

From the request example above, we have

ObservationData[] results

returning from the call. Assuming that just one record was returned, and that the **ObservationData** was an atomic observation, the array of results would be unpacked as follows:

String theXML_result = results[0].value[0].extract_string();

We know to unpack a string from the **CORBA::any** because the code returned,

results[0].code

contains **PROGRESS_NOTE**, our requested observation code, which is associated with exactly one return type, a string.

The content of **theXML_result** would be along the lines of the first, full-length XML sample given above.

F.3 Non-empiric Antibiotic Decision Support

F.3.1 Usage Scenario and Example

A patient is in the Intensive Care Unit (ICU) and has been treated empirically for a pneumonia (i.e., given antibiotics without knowledge of the bacterial cause of the pneumonia) with Ceftazadime. Since the inception of antibiotic therapy, the patient has not improved. Laboratory results, which include the microbiology results (bacterium and associated sensitivities to varied antibiotics), CBC, and serum creatinine, become available.

The physician uses a web browser to run a user interface to an antibiotic decision support system. The physician selects the patient. The patient's demographic, laboratory (microbiology, serum creatinine, and CBC), and vital statistics are accessed from a centralized clinical data repository. If this data is not accessible, the physician or a surrogate has the option to manually enter this data. In this example, the weight is 159 lbs, the height is 72 inches, the age is 60, the sex is Male, and the serum creatinine is 1.7.

The physician selects a formulary to be utilized by the decision support system. The user hits the run button and the decision support is invoked on the server. The above data is modeled in the following features. The server-side decision support system accesses the above data to create a list of drug, sensitivity, dose, dosing interval, and daily cost information for drugs in the formulary, where sensitivities are known. These results are prioritized by sensitivity and cost.

The results of the decision support are presented to the user. In the given example, the bacterium is the resistant Streptococcus Pneumonia, which is sensitive only to Vancomycin. The output suggests to the physician that his treatment should include one gram of Vancomycin every 24 hours.

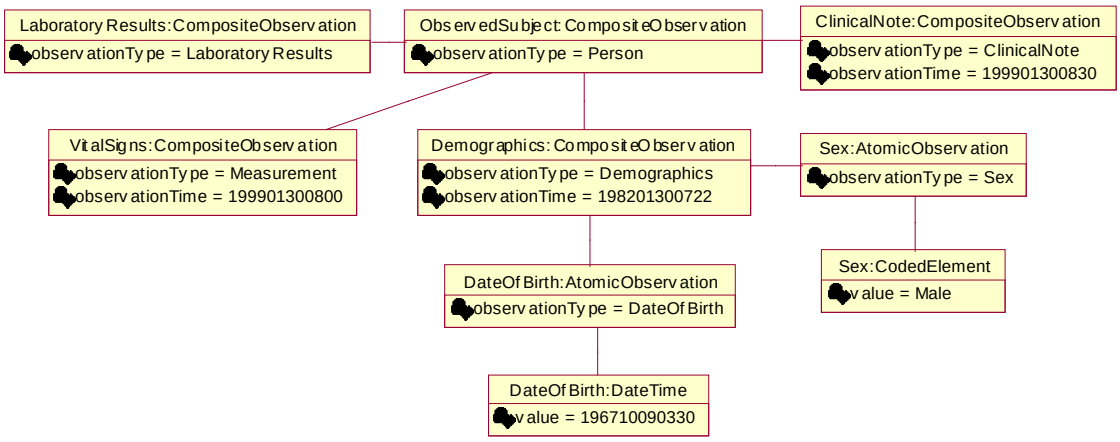

Figure 4-108 Antibiotic Decision Support System - Example

This is an Object Diagram for what might be a way to represent an Antibiotic Decision Support Systems input information.

F.3.2 ObservedSubject:CompositeObservation

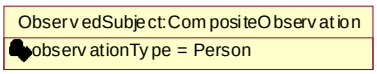

Figure 4-109 ObservedSubject:CompositeObservation

This instance of an ObservedSubject is typed as a Person (patient) and has a CompositeObservation link of type LaboratoryResults, a CompositeObservation link of type ClinicalNote and a CompositeObservation link of type VitalSigns and . a CompositeObservation link of type Demographic. This diagram is not meant to be normative but rather to show an example of what an ObservedSubject of type Person (patient) may have associated with it.

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the ObservedSubject. For example, Person, Organ, Epidemic, etc.

F.3.3 LaboratoryResults:CompositeObservation

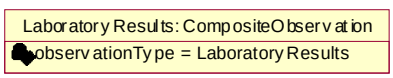

Figure 4-110 LaboratoryResults:CompositeObservation

A Person (patient) in a health care information environment usually has a link to some LaboratoryResults information.

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the CompositeObservation. In this case LaboratoryResults.

F.4 LaboratoryResults:CompositeObservation

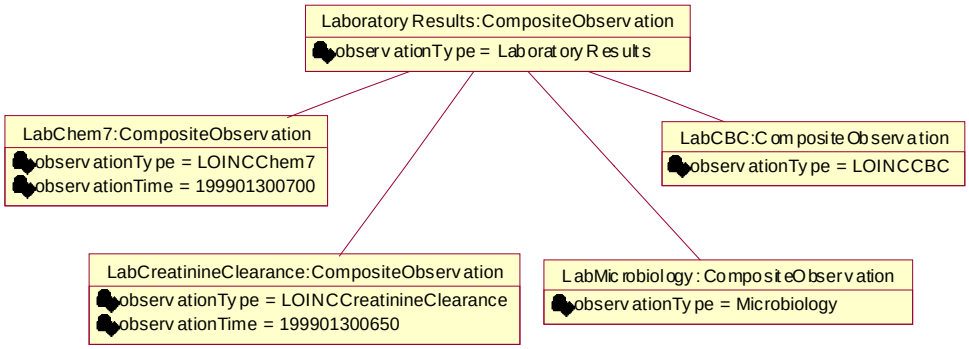

Figure 4-111 LaboratoryResults:CompositeObservation

The LaboratoryResults has a CompositeObservation link of type LabChem7, LabCreatinineClearance, LabMicrobiology and a LabCBC.

F.4.1 LabChem7:CompositeObservation

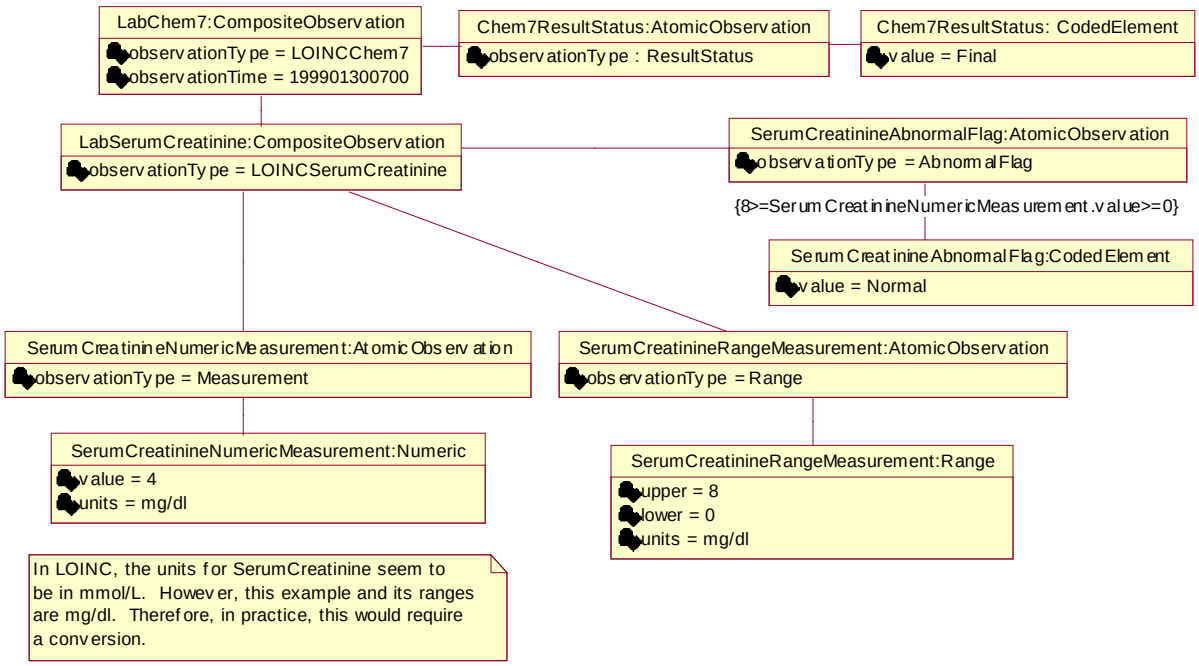

Figure 4-112 LaboratoryResults:LabChem7

F.4.2 LabCreatinineClearance:CompositeObservation

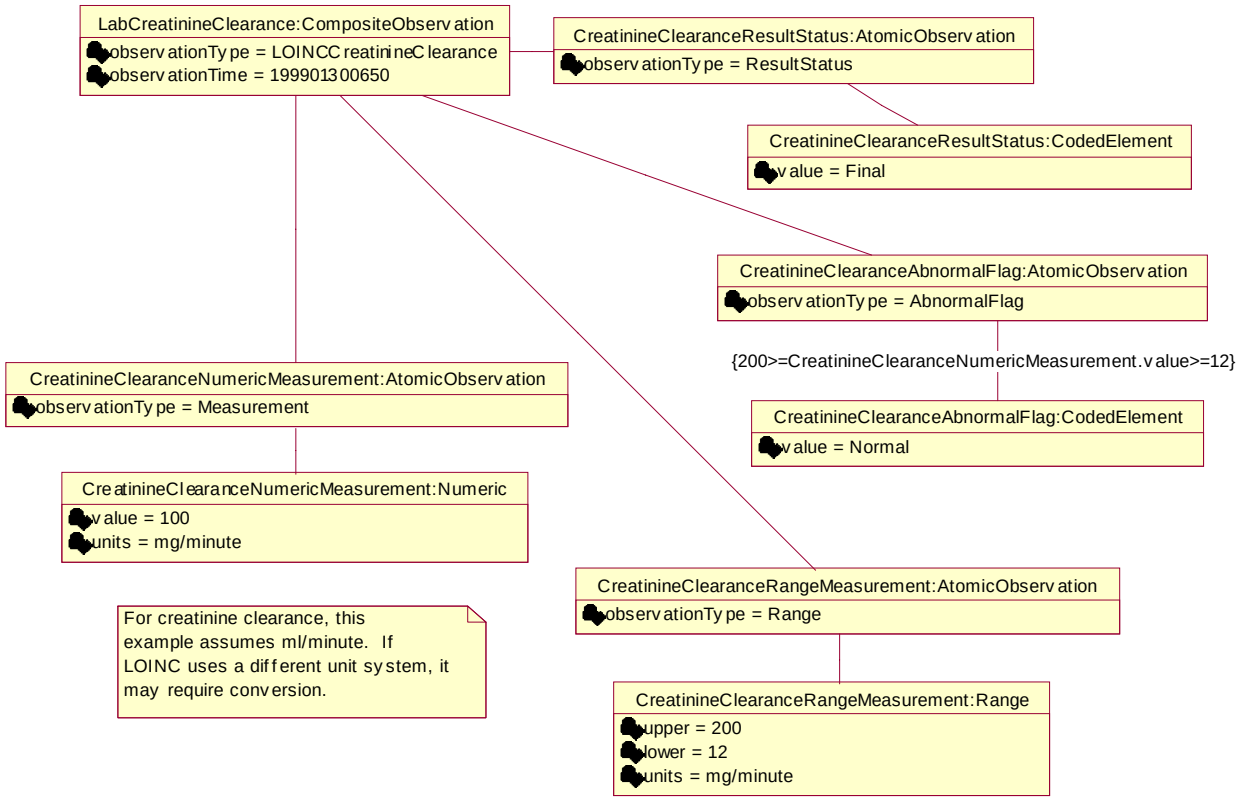

Figure 4-113 LaboratoryResults:LabCreatinineClearance

F.4.3 LabMicorbiology:CompositeObservation

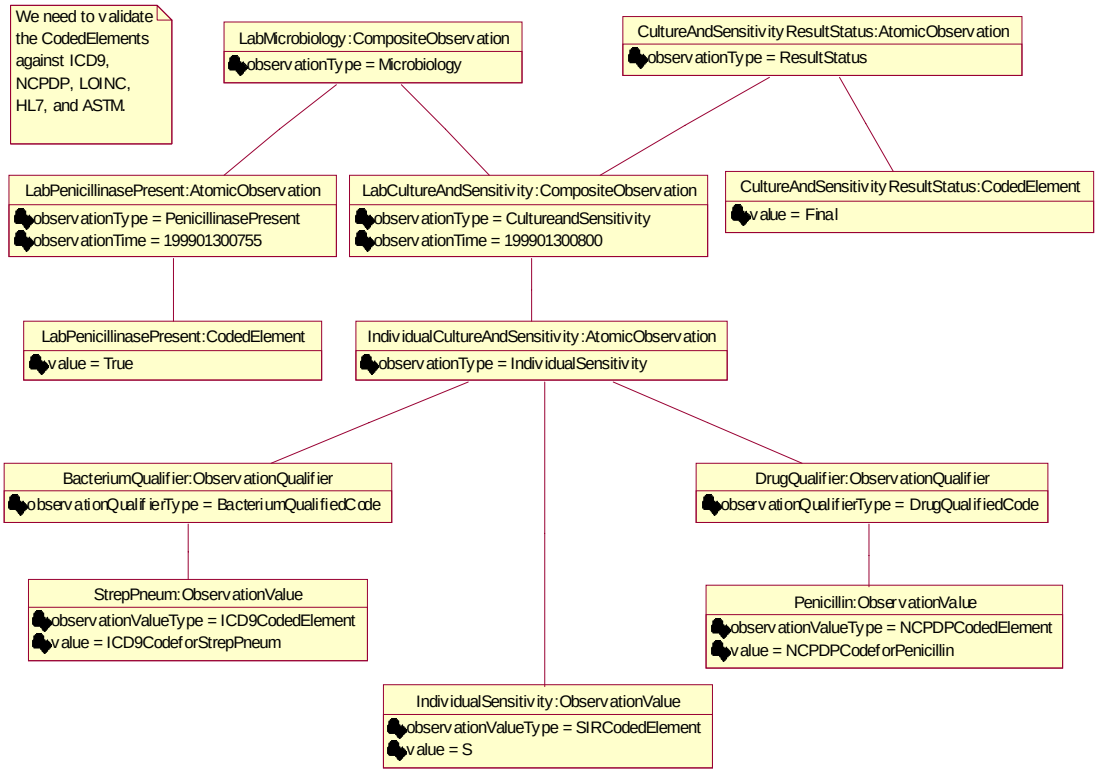

Figure 4-114 LaboratoryResults:LabMicorobiology

F.4.4 LabCBC:CompositeObservation

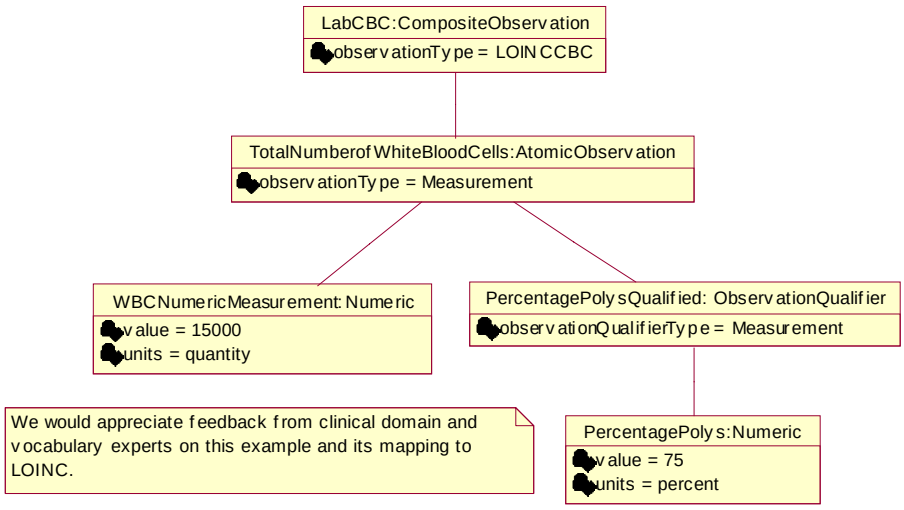

Figure 4-115 LaboratoryResults:LabCBC

F.5 ClinicalNote:CompositeObservation

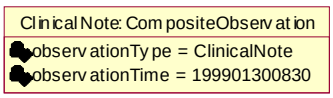

Figure 4-116 ClinicalNote:CompositeObservation

A Person (patient) in a health care information environment usually has a link to some ClinicalNote information.

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the CompositeObservation. In this case ClinicalNote.

observationTime:TimeSpan

This is a TimeSpan that provides the time of the CompositeObservation. In this case 199901300830.

F.5.1 ClinicalNote:CompositeObservation

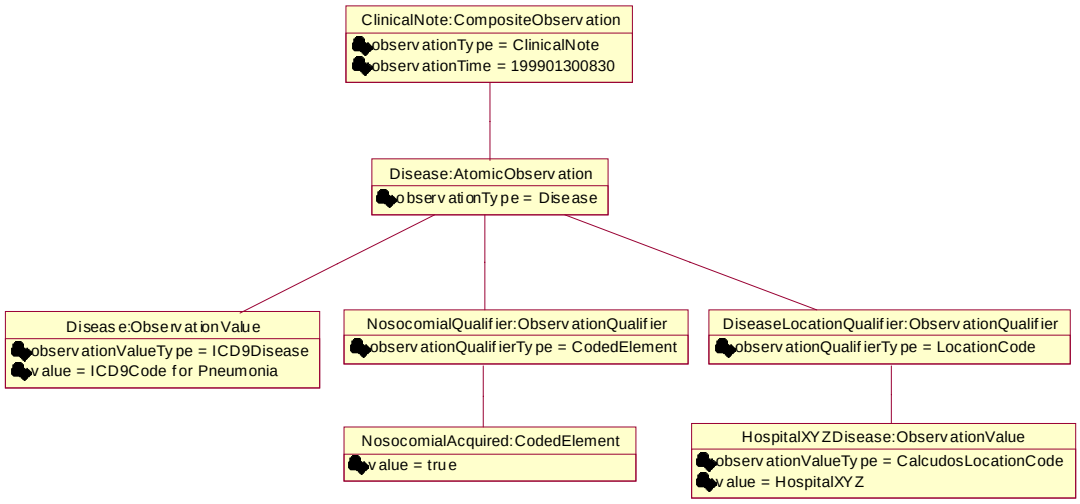

Figure 4-117 ClinicalNote:CompositeObservation

F.6 VitalSigns:CompositeObservation

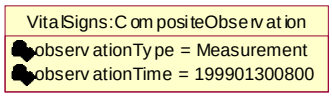

Figure 4-118 ClinicalNote:CompositeObservation

A Person (patient) in a health care information environment usually has a link to some ClinicalNote information.

observationType:QualifiedCode

This is a QualifiedCode that provides the type of the CompositeObservation. In this case Measurement.

observationTime:TimeSpan

This is a TimeSpan that provides the time of the CompositeObservation. In this case 199901300800.

F.6.1 VitalSigns:CompositeObservation

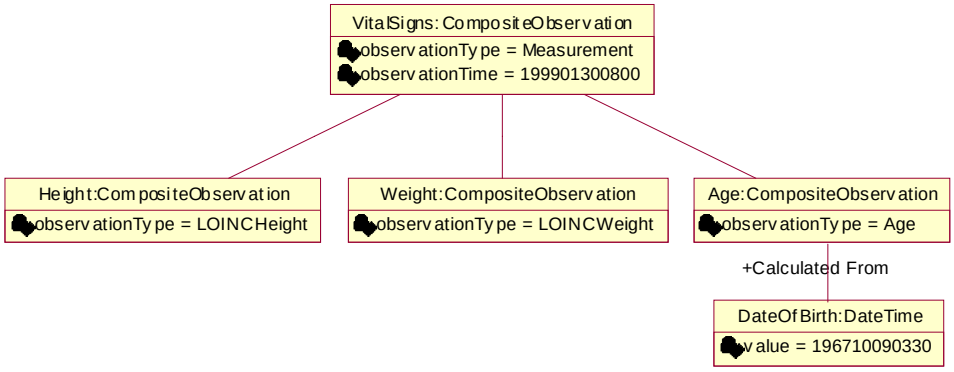

Figure 4-119 VitalSigns:CompositeObservation

F.6.2 Height:CompositeObservation

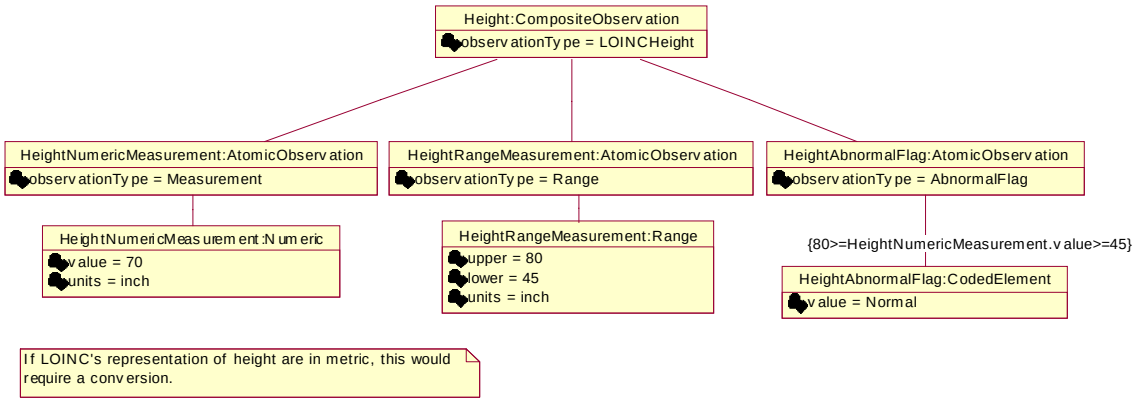

Figure 4-120 Height:CompositeObservation

F.6.3 Weight:CompositeObservation

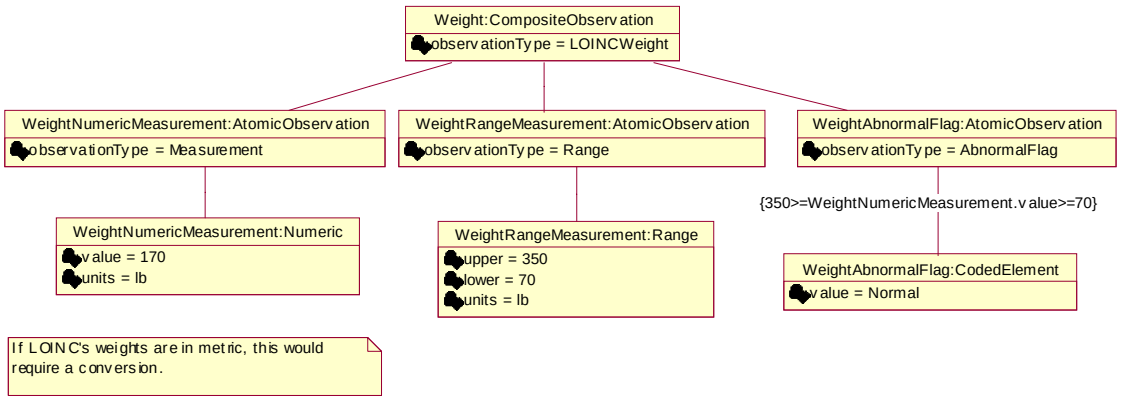

Figure 4-121 Weight:CompositeObservation

F.6.4 Age:CompositeObservation

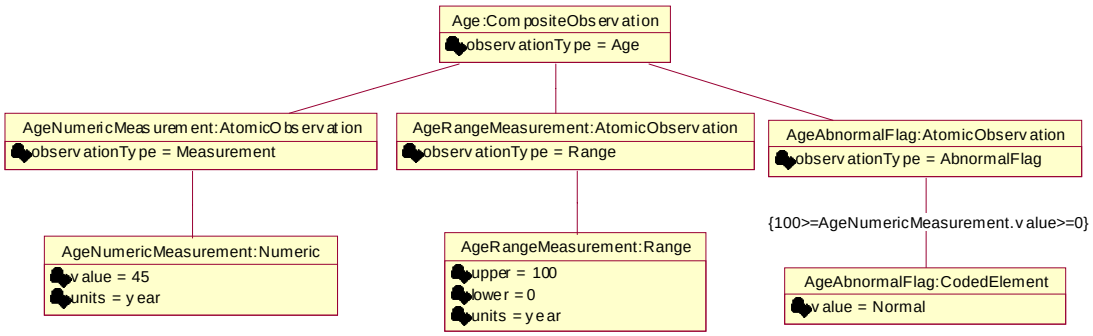

Figure 4-122 Age:CompositeObservation