OpenGIS - Catalog Interface Implementation Specification (Version1.0)

The Joint Submission

Submitters:

Blue Angel Technologies

Environmental Systems Research Institute (ESRI)

Geomatics Canada (Canada Centre for Remote Sensing (CCRS))

Intergraph Corporation

Marconi Integrated Systems, inc.

MITRE

Oracle Corporation

U.S. Federal Geographic Data Committee (FGDC)

U.S. National Aeronautics and Space Administration (NASA)

U.S. National Imagery and Mapping Agency (NIMA)

Contributors:

Compusult Limited, GEODAN IT bv, HJW, JRC - European Commission, SICAD GEOMATICS

OpenGIS Project Document 99-051s

NOTICE

The companies listed above hereby grant a royalty-free, paid up license to the Open GIS Consortium, Inc. (OGC) for worldwide distribution of this document or any derivative works thereof, so long as OGC reproduces the copyright notices and the below paragraphs on all distributed copies.

The material in this document is submitted to the OGC for evaluation. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELEIVED TO BE ACCURATE, THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material. The information contained in this document is subject to change without notice.

Table of Contents

1. Preface 1
2. Overview 3
3. General Catalog Interface Model 9
4. OGC_Common Catalog Query Language 75
5. CORBA Profiles 84
6. OLEDB Profile 121
7. WWW Profile 155
8. Proposed Additional OGC Basic Data Types 176

Detailed Table of Contents

1. Preface 1
1.1 Submitting Entities 1
1.2 Contributing Entities 1
1.3 Contact Points 1
2. Overview 3
2.1 Context of Catalog Services 3
2.2 Reference Model Architecture 3
2.3 Cross Profile Interoperability 4
2.4 Catalog Object Model 5
2.5 Metadata Model Independence 6
2.6 Query Language 7
2.7 Use of XML 7
2.8 Browse 7
2.9 Interoperability and Compliance with Simple Features 7
2.10 Distributed Search 8
3. General Catalog Interface Model 9
3.1 Introduction of The General Model 9
3.2 Coarse-Grain Structural Model 11
3.2.1 Overview of The Coarse-Grained Interface Model 11
3.2.2 The Messaging Model 12
3.2.3 CG_CatalogService Interface 16
3.2.4 CG_Discovery Interface 21
3.2.5 CG_Access Interface 27
3.2.6 CG_CatalogManager Interface 30
3.2.7 Parameter Type Definitions 33
3.3 Fine-Grain Structural Model 45
3.3.1 Overview 45
3.3.2 Managers 45
3.3.3 Responses 49
3.3.4 DataTypes 53
3.3.5 Catalog Specific DataTypes 53
3.3.6 Callbacks 58
3.4 Coarse-Grain Dynamic Model 59
3.4.1 UML State Diagram Notation 59
3.4.2 Catalog Server State Machine 59

3.4.3 Discovery State 60
3.4.4 Access State Diagram 63
3.4.5 Management State 65
3.4.6 Utility State Diagram 65
3.5 Fine-Grain Dynamic Model 66
3.6 Cross-Model Interoperability 70
3.6.1 Coarse to Fine Grain Mapping 70
3.6.2 DCP Bridge Dynamics 73
4. OGC_Common Catalog Query Language 75
4.1 Assumptions during the development of OGC_Common Query Language: 75
4.2 BNF definition of OGC_Common Query Language 75
5. CORBA Profiles 84
5.1 Fine-Grain CORBA Profile 84
5.1.1 Sequence Diagrams 84
5.1.2 IDL Definition 84
5.1.3 Module FGCatalog 84
5.1.4 Module CB 94
5.1.5 Module UID 94
5.1.6 Module OGCBasic 95
5.2 Coarse-Grained CORBA Profile 98
5.2.1 Architecture - Object Model 98
5.2.2 Event Traces 98
5.2.3 Interface Definition - IDL 98
6. OLEDB Profile 121
6.1 Architecture 121
6.1.1 Mandatory OLEDB Interfaces 121
6.1.2 OGC Extensions to OLEDB 122
6.2 Sequence Diagram 122
6.3 Parameter Translation 123
6.3.1 CG_AttributeCategory 124
6.3.2 CG_BrokeredAccessRequestType 124
6.3.3 CG_Status 124
6.3.4 CG_Capability 124
6.3.5 CG_QueryLanguage 127
6.3.6 CG_CatalogEntryType 127
6.3.7 CG_CharacterSet 128
6.3.8 CG_CollectionName 128

6.3.9 CG_MessageFormat 128
6.3.10 CG_PredefinedPresentationType 128
6.3.11 CG_PresentationDescription 128
6.3.12 CG_QueryExpression 129
6.3.13 CG_QueryScope 129
6.3.14 CG_RequestID 129
6.3.15 CG_ResultType 129
6.3.16 CG_ReturnData 129
6.3.17 CG_SortField 130
6.3.18 CG_SortOrder 130
6.3.19 CG_UserInformation 130
6.3.20 CG_PaymentMethod 130
6.3.21 TupleType 130
6.3.22 Schema 130
6.3.23 CG_SchemaID 131
6.4 Detailed Implementation Guidance 131
6.4.1 Establish a catalog session 131
6.4.2 End a Catalog Session 133
6.4.3 Query the server properties 134
6.4.4 Check the status of a request 136
6.4.5 Cancel a request 138
6.4.6 Issue a Query 140
6.4.7 Present Query Results 146
6.4.8 Get the schema 151
7. WWW Profile 155
7.1 Architecture 155
7.1.1 Supported Services 155
7.1.2 Transport (HTTP) 155
7.1.3 Transport (TCP) 155
7.2 Sequence Diagrams 155
7.3 Example Sequence Diagram 156
7.4 Interface Definition – XML 158
7.5 Definition of Externals 165
7.5.1 Additional Search Info 165
7.5.2 Order Extended Service 165
8. Proposed Additional OGC Basic Data Types 176
8.1 AbsTime 176

8.2 BinData 176
8.3 Cardinality 176
8.3.1 Public Attributes: 176
8.4 DG_DirectedGraph 176
8.5 DG_DirectedGraphList 177
8.6 Date 177
8.6.1 Public Attributes: 177
8.7 Edge 177
8.7.1 Public Attributes: 177
8.8 EdgeList 177
8.9 FileLocation 177
8.9.1 Public Attributes: 177
8.10 FileLocationList 177
8.11 Name 177
8.12 NameList 178
8.13 NameValue 178
8.13.1 Public Attributes: 178
8.14 NameValueList 178
8.15 NameValueTable 178
8.16 Node 178
8.16.1 Public Attributes: 178
8.17 NodeID 178
8.18 NodeList 178
8.19 NodeType 178
8.19.1 Public Attributes: 178

Table of Figures

Figure 1 - Information Discovery Continuum....3
Figure 2 - Reference Model Architecture....4
Figure 3- Decomposition of Catalog Services....4
Figure 4- An Example of a One-Way Bridge....5
Figure 5- Catalog Object Model....6
Figure 6 - Top Level General Model....10
Figure 7 -Main Static Class Diagram of the Coarse-Grained General Interface Model....11
Figure 8 - Main Class Diagram for Message Package....12
Figure 9 - Request Message Classes and Their Attributes (Parameters) Defined for OGC Coarse-Grain Catalog Interface Model....14
Figure 10 -Response Message Classes and Their Attributes (Parameters) Defined for OGC Coarse-Grained Catalog Interface Model....15
Figure 11 - Static Class Diagram showing CG_Capability Class and Its Instantiated Subtypes.....34
Figure 12 - The Library and Managers....45
Figure 13 - Response Interfaces....50
Figure 14 - UML State Diagram Notation....59
Figure 15 - Catalog Session State Diagram....60
Figure 16 - Discovery State Diagram (without Status and Cancel)....61
Figure 17 - Discovery State Diagram (Complete)....62
Figure 18- Access State Diagram....63
Figure 19- Order Estimation State Diagram....64
Figure 20 - Order Submit State Diagram....64
Figure 21 - Management State Diagram....65
Figure 22 - Utility State Diagram....65
Figure 23 - Typical Query Sequence....66
Figure 24 - Minimal Query Sequence....67
Figure 25 - Query with Callback Sequence....68
Figure 26 - Coarse to Fine Bridge....74
Figure 27 - WWW Profile Sequence Diagram....157
Figure 28 - The DG_DirectedGraph....176

Table of Tables

Table 1 - The Profile Matrix 5
Table 2 - The Operations of the CG_CatalogService Interface 16
Table 3 - The Operations of the CG_Discovery Interface 21
Table 4 - The Operations of the CG_CatalogManager Interface 30
Table 5 - Brokered Access Request Types 33
Table 6 - Order Status Codes 38
Table 7 - Minimal Mandatory Attribute Definitions 42
Table 8 - Sort Order Operations 43
Table 9 - CG_ResultType as used in Discovery State Diagrams 61
Table 10 - Order Request Type Diagram Codes 63
Table 11 - Coarse to Fine Grain Mapping 70
Table 12 - General Model to WWW Profile Message Mapping 156
Table 13 - Order Extended Service 166

1. Preface

1.1 Submitting Entities

The following entities are pleased to jointly submit this specification in response to the OGC Request 6, Core Task Force, Catalog Working Group, A Request for Proposals: OpenGIS Catalog Interface (OpenGIS Project Document Number 98-001r2):

Blue Angel Technologies, Inc.

Environmental Systems Research Institute (ESRI)

Geomatics Canada (Canada Centre for Remote Sensing (CCRS))

Intergraph Corporation

Marconi Integrated Systems, Inc.

MITRE

Oracle Corporation

U.S. Federal Geographic Data Committee (FGDC)

U.S. National Aeronautics and Space Administration (NASA)

U.S. National Imagery and Mapping Agency (NIMA)

1.2 Contributing Entities

The submitting entities were grateful for the contributions from the following companies in the development of this response to the OGC Request 6, Core Task Force, Catalog Working Group, A Request for Proposals: OpenGIS Catalog Interface (OpenGIS Project Document Number 98-001r2):

Compusult Limited

GEODAN IT bv

Hammon, Jensen, Wallen & Associates, Inc (HJW)

JRC (Joint Research Centre), European Commission

SICAD GEOMATICS

1.3 Contact Points

All questions about the joint submission should be directed to one of the following contacts:

Yonsook Enloe
SGT, Inc.
7701 Greenbelt Rd
Greenbelt, MD 20770
yonsook.enloe@gsfc.nasa.gov
voice : +1-704-243-2085
fax : +1-704-243-2150

Doug Nebert
U.S. Federal Geographic Data Committee
USGS National Center, Mail Stop 590
12201 Sunrise Valley Drive
Reston, VA 20192
ddnebert@usgs.gov
voice : +1-703-648-4151
fax : +1-703-648-5755

Larry Stephens
National Imagery and Mapping Agency
12310 Sunrise Valley Drive
Mail Stop P-23
Reston, VA 20191
stephenl@nima.mil
voice : +1-703-262-4583
fax : +1-703-262-4599

2. Overview

Section 2 provides a descriptive overview of key issues in the development of the OGC Catalog Interface. Section 2 is non-normative.

2.1 Context of Catalog Services

The geospatial community is a very broad based community which works in many different operational environments. On one extreme there are tightly coupled systems dedicated to well defined functions in a tightly controlled environment. At the other extreme are Web based services which know nothing about the client. The initial catalog submittals addressed two parts of this continuum The DCS proposal addressed the controlled Enterprise environment where a degree of a-priori knowledge exists about the client and server. The EO/GO proposal addressed the global internet case where no a-priori knowledge exists between client and server. This document provides a specification which is applicable to the full range of catalog operating environments.

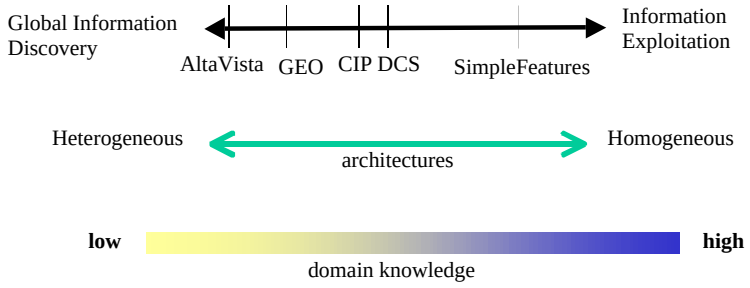


Figure 1 - Information Discovery Continuum

2.2 Reference Model Architecture

The Reference Model for the OGC Catalog Interface is composed of two parts: a Reference Architecture and a Decomposition of Catalog Services.

Figure 2 shows the Reference Architecture assumed for development of the OGC Catalog Interface. The architecture is a multi-tier arrangement of clients and servers. To provide a context, the architecture shows more than just catalog interfaces. The bold lines illustrate the scope of OGC Catalog and Features interfaces. Where appropriate, OGC Feature interfaces have been re-used in the OGC Catalog interface, as discussed in Section 2.9.

The Application shown Figure 2 interfaces with the Application Server using the OGC Catalog Interface. The Application Server may draw on one of three sources to respond to the Catalog Service request: a Metadata Store local to the Application Server, another Application Server, or a Data Store. The interface to the local metadata store is internal to the Application Server. The interface between Application Servers is the OGC Catalog Interface. The interface to the Data Store is the OGC Features Interface. In this case an Application Server is acting as both a client and server. See Section 2.10 for more about Distributed Searching. Data returned from a OGC Features query is processed by the Application Server to return the data appropriate to a Catalog request.

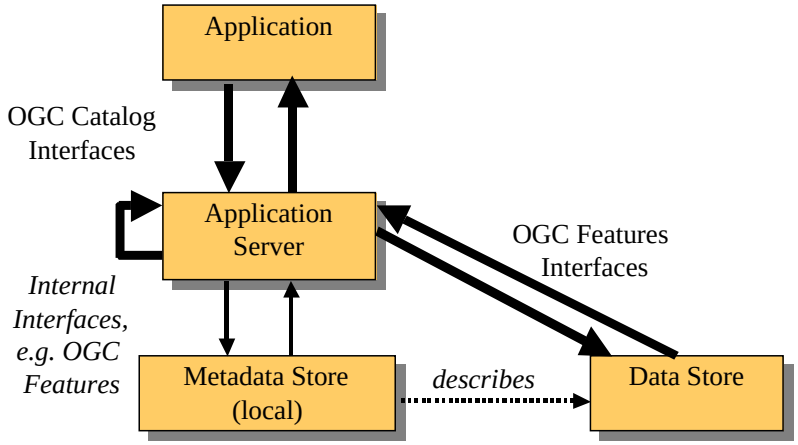


Figure 2 - Reference Model Architecture

Figure 2 shows a decomposition of the OGC Catalog Services. Discovery Services are those services which allow a client to locate metadata that describes data. Access Services provide the client with methods to request services on the data. Access Services are divided into two types. Direct Access provides the client with a handle which when used by the client will provide the data to the client. The specific definition of such a handle is outside of the scope of the OGC Catalog Interface. Brokered Access provides the client with methods to order data that will be delivered in some means outside of the Catalog Interface. The Management Service defines methods for a client to change the metadata held by a catalog.

The Discovery Service is to be provided by all Application Servers claiming compliance with the OGC Catalog Interface. The Access and Management Services are optionally required for an OGC compliant catalog. But, if an application server claims Access or Management compliance this OGC Catalog Interface specification defines how the services are to be implemented.

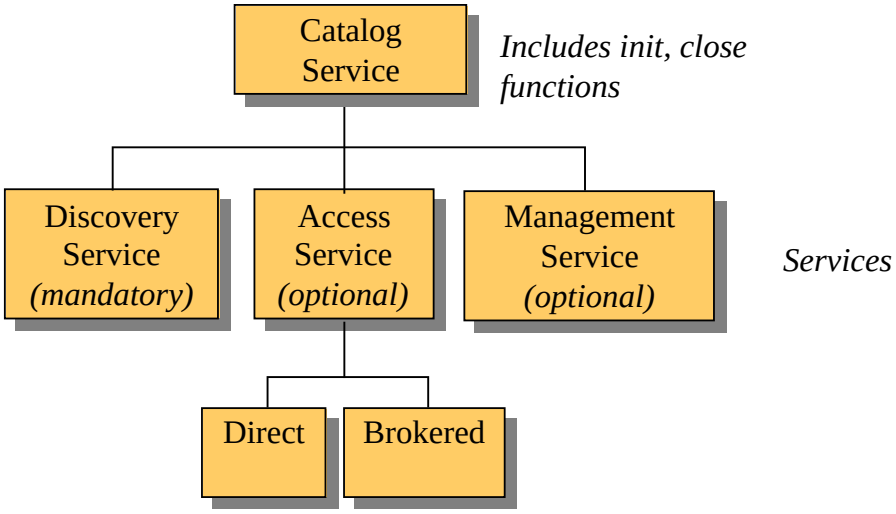


Figure 3- Decomposition of Catalog Services

2.3 Cross Profile Interoperability

The OGC Catalog General Model defines the behaviors and interfaces applicable to all implementations of OGC Catalogs. In the real world, there is no one solution that fits everyone's needs. The OGC Catalog

Profiles provide refinements of the General Model targeted toward specific implementation communities. For those communities, the Profile defines the standards for compliance.

Profiles can be categorized by their location in a matrix. The primary axis of this matrix is the Distributed Computing Platform (DCP). The DCP specifies the distributed computing environment that the Profile will operate within. This Specification defines a DCP as any set of protocols and services which allow two entities to communicate. DCPs addressed in this specification include CORBA, OLEDB and The World Wide Web. The secondary axis of the matrix describes the complexity of the interfaces. This specification applies to environments as diverse as the internet and workgroup clusters. At the one extreme, the client and server have very little knowledge of one another. The interfaces in this environment must be flexible, well defined and simple. Profiles designed for this environment are classified as coarse grained profiles. At the other extreme, the client and server are closely coupled. The standards and conventions imposed by the system design allow for a much more detailed interface between the catalog components. Profiles addressing this environment are classified as fine grained profiles.

Table 1 - The Profile Matrix

DCP	Fine Grained	Coarse Grained
CORBA	Section 5.1	Section 5.2
COM	NA	Section 6
Web	NA	Section 7

The General Model provides the glue that ties the Profiles together. Every Profile must demonstrate consistency with the General Model in terms of behaviors and interfaces. This consistency allows for the construction of Bridges between Profile implementations. A Catalog Bridge would consist of software layered over implementations of two or more Profiles. The Profile implementations would all expose the same interfaces to the Bridge code. In this way, the Bridge serves as little more than a store and forward device for Catalog request and response messages. The Profile implementations are responsible for executing those messages within their implementation domain.

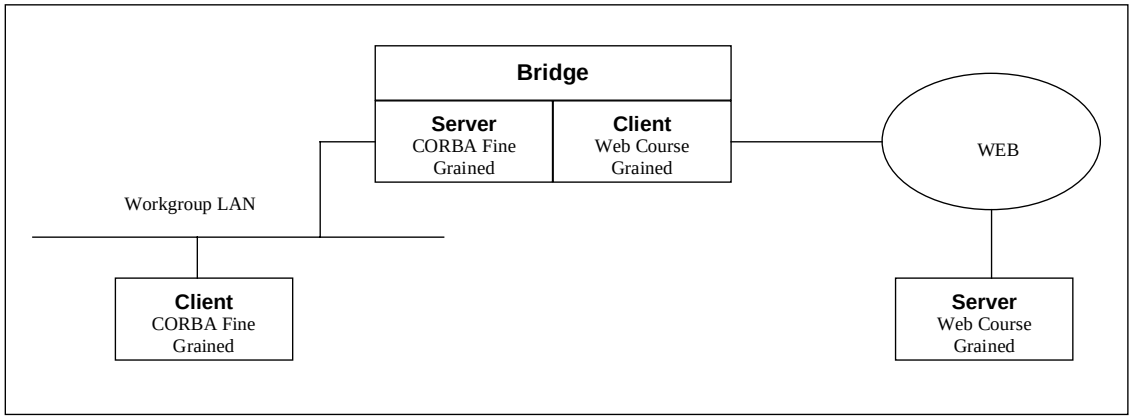


Figure 4- An Example of a One-Way Bridge

2.4 Catalog Object Model

The static class diagram of Figure 5 illustrates how the Catalog Services can utilize a OGC Features Implementation. A catalog entry" references" the data it describes through a feature-to-feature association. Since metadata can be associated at any level in the feature hierarchy, the target of this reference can be any subclass of feature, but would most commonly be associated to the feature collection, or logical data set. The catalog entry consists of an aggregation of metadata attributes, at least one of which would

describe the "footprint" of the data referenced. Thus, a catalog entry meets the fundamental definition of a feature. For this reason, the Catalog Entry class realizes the Feature interface, that is, it supports all interface protocols defined on Feature. Since the catalog entries are sub-types of feature, their aggregation, the Catalog, would be a sub-type of feature collection. Thus, the Catalog realizes the interface for Feature Collection. Assuming that the catalog has been implemented according to an OGC compliant feature datastore, it would be possible to access that datastore directly using any OGC feature data access interface. Thus, one mechanism to implement robust catalogs would be to use OGC compliant feature datastores.

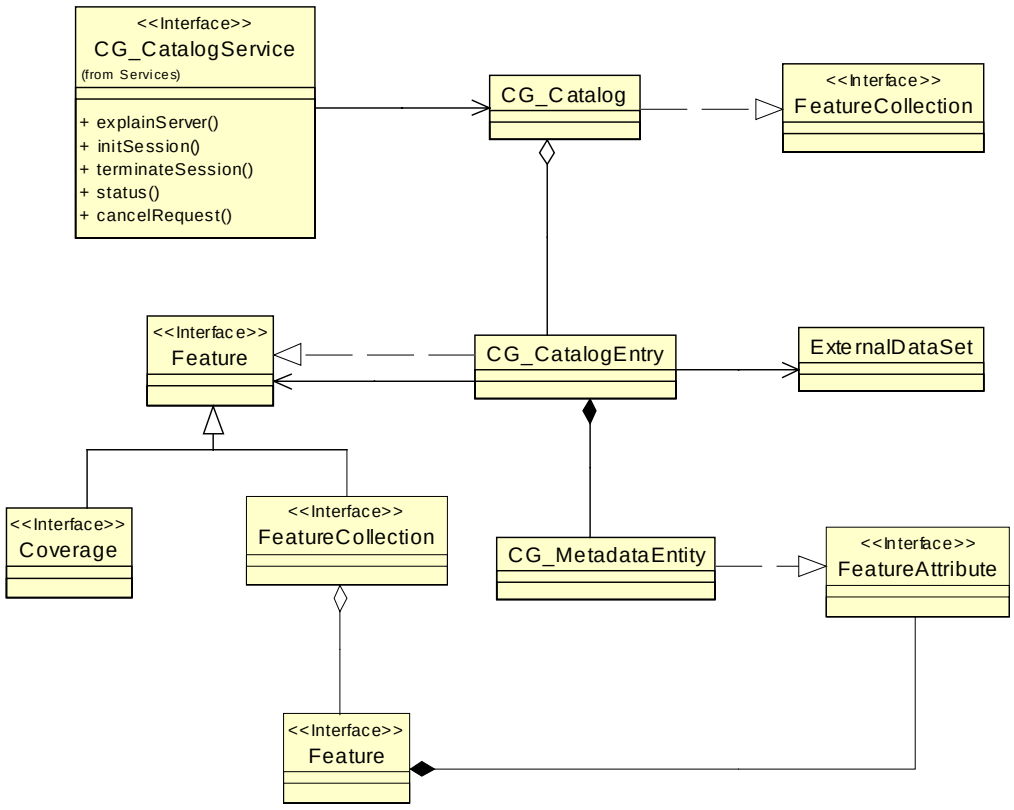


Figure 5- Catalog Object Model

2.5 Metadata Model Independence

Metadata structures, dependencies, and definitions -- known as schema -- exist for multiple information communities. For the purposes of interchange of information within an information community, a metadata schema may be defined that provides for a common vocabulary supporting search, retrieval, display, and association between the description and the object being described. Although this specification does not require the use of a specific schema, the adoption of a given schema within an information community ensures the ability to communicate and discover information

The geomatics standardization activity under Technical Committee 211 includes a formal schema for geospatial metadata that is intended to be applied to all types of information. This standard, currently in Committee Draft form (June 1999) includes a proposal for an Essential Profile of metadata elements in common use. All future registered ISO TC211 metadata profiles must include these Essential Profile elements. For the purpose of information exchange across OpenGIS/Geomatics communities, the schema

and elements within the Essential Profile of ISO 15046-15 must be implemented by conforming implementations.

2.6 Query Language

The Query Capabilities of the OpenGIS Catalog Interface are intended to provide a minimum subset of query capabilities that can be assumed at OGC Compliant Catalog implementations while providing maximum flexibility for enabling alternate styles of query, result presentation, and query languages. The flexibility goals are accomplished through the use of a query service call that contains the parameters needed to establish the query /result presentation style and a query expression parameter that includes the actual query and an indication of the query language used.

The interoperability goal is supported by the specification of a minimal query language, which must be supported at all compliant OpenGIS Catalog Service sites (defined in Section 4). This query language supports nested Boolean queries, text matching operations, temporal data types, the Simple Features "well known text representations " and Simple Features relational operators. The minimal query language syntax is based on the SQL WHERE clause in the SQL SELECT statement.

The minimal query language assists the consumer in the discovery of datasets of interest at all sites supporting the OpenGIS Catalog Services. The ability to specify alternative query languages allows for evolution and higher levels of interoperability among more tightly coupled subsets of Catalog Service Providers and Consumers

2.7 Use of XML

The eXtensible Markup Language (XML) version 1.0 is used in the implementation of certain aspects of catalog services to promote easy encoding and decoding of structured information. To facilitate translation of information between implementation profiles XML is used: 1) to package the elements of a query, and 2) to package the structured information being returned from a query.

Standard metadata schemas are expressed in this specification using eXtensible Markup Language (XML) using Document Type Declarations (DTDs) that are separate from the XML document they describe. In catalog applications, the documents marked-up in XML must include either reference to the DTD in the header line, and/or the DTD embedded in the document. XML-Schema is a proposal under development (June 1999) within the World Wide Web Consortium that is likely to define a more rigorous successor to the DTD.

2.8 Browse

In the OpenGIS community there are a significant number of non-character data items that can be used for Discovery. A good example of this type of metadata is a browse image, a reduced resolution version of an image that is used by the consumer to select the data he wishes to order. The browse image can be acquired from a service or as a standard piece of metadata based on the size and ability to accept parameterization. A very small data item such as a thumbnail image would be useful to include in the catalog results presentation. A large static browse or a dynamic browse that selected different resolutions or bands of the image based on request parameters would use the access service and be represented by the appropriate URI in the catalog results. There is a large spectrum between these two extremes and our legacy systems handle browse images in all the ways discussed.

The current advice of this specification is to encode very small browses such as thumbnail images as part of the catalog query result presentation using a common encoding supported in XML Schema. For all larger browse images treat them as an access service and place a referencing URI in the appropriate result fields.

2.9 Interoperability and Compliance with Simple Features

A functional requirement for this proposal was to use the Simple Feature types and functions wherever possible, such as Feature, Feature Collection, Geometry, and Spatial Reference System and the spatial

operators. This proposal tries to maintain conceptual compatibility with the Simple Features Implementation Specifications in the following manners:

- Query comparison operators consistent with those defined in simple features are used in the Catalog Specification Metadata query mechanism.
- The Access service of the catalog specification allows for a simple transition to Simple Feature access mechanisms.

The consistency between the query mechanism within the catalog specification and the query language within the simple features specification allows an implementation to use a simple features data store for the storing of metadata.

An alternative design would have been to use the simple features query language directly to access metadata. This design was rejected to preserve legacy implementations using Z39.50 metadata servers incapable of supporting a complex query language. Since many Z39.50 metadata servers use SQL databases as backends, a negotiation phase between a client and a particular server could promote the query language to a full object-SQL for Simple Features.

The current query model allows for three types of query language:

- A common, mandatory query language using SQL style syntax, Z39.50 Type 1 operation style, and spatial operations derived from the simple features model in the OGC Simple Features Implementation Specification.
- Any dialect of SQL conformant with the OGC Simple Features Implementation Specification.
- Z.39.50 Type 1 query.

This specification allows support of additional query languages as they are identified.

Each conformant server must support the mandatory query language. Other languages are optional. Because of the limitations placed upon the mandatory query language it will be possible to implement a service that would translate the mandatory query language syntax into either of the other two languages.

2.10 Distributed Search

The Reference Architecture for the OGC Catalog allows for catalog requests to be distributed to multiple catalogs. The architecture allows for a Catalog to accept a request from a client and distribute the request to other Catalogs. For the OGC Catalog Service, Distributed Catalog Searching is defined as a service that involves services of multiple Catalog Servers, in addition to the primary client-server interaction. Distributed Searching is accomplished by a catalog server propagating secondary catalog service requests to other catalog servers.

To enable Distributed Searching, the following items are needed:

- A multi-tier Reference Architecture as provided by this specification (as defined in Section 2.1)
- A data model to define how searches are to be distributed as defined by an information community.
- Messages with elements applicable to Distributed Searching as provided by this specification

To support distributed searching, a community develops a data model that determines how a search will be distributed to coordinated data servers. The OGC Catalog General model allows data model neutrality with respect to distributed searching.

Several of the Discovery messages defined in Section 3 contain elements that pertain to distributed searching. The query message contains elements that allow the client to request certain search behavior with respect to distribution. The request and response messages define elements that allow for the retrieval and comprehension of a distributed result set. The request and response messages contain elements that allow for understanding the status of distributed searches.

3. General Catalog Interface Model

3.1 Introduction of The General Model

The General Catalog Interface Model is composed of two equivalent views (also referred to as entry points into the object model), one being “Coarse-Grain” the other “Fine-Grain”. It also touches on the concept of having an OGC Service Architecture Framework where the Catalog Interface would be one component of such a framework. Both Structural and Dynamic models are provided for both views. The Fine Grain view provides the client with a rich set of objects and interfaces for interfacing with the server. This model supports the creation of integrated systems of OGC Catalog Clients and Servers. These client and server components exercise a fine degree of control and coordination upon each other. This environment also facilitates the integration of Catalog Management and Simple Features Access within the catalog Discovery context. The Coarse Grain model provides a more generalized interface between the client and server. Coarse Grain systems require all control and coordination client and server components to occur at a aggregated level. These profiles are appropriate where a client is accessing a number of dissimilar servers, or is not engaging in singular interactions with a server. What the Coarse Grain client gives up in detailed control it gains in flexibility. The General Model also contains a small set of interfaces that an implementer would need to provide to facilitate a mapping between these two entry points. These interfaces would be private and left up to an implementer to specify how they would be developed.

Figure 6 provides a top-level view of the structural General Model. The classes of the Coarse-Grained model are shown at the top of Figure 6. A Translator is shown in the middle of Figure 6. Elements of the Fine-Grain model are shown in the bottom of the Figure. The Coarse-Grain structural model is detailed in Section 3.2. The Fine-Grain structural model is detailed in Section 3.3.

Both the Coarse and Fine –Grained interface models provide a set of service interfaces that support the discovery, access, maintenance and organization of catalogs of geospatial information. The interfaces specified are intended to allow users or application software to find information that exists in multiple distributed computing environment, including the World Wide Web (WWW) environment.

The dynamic models take different forms for coarse grained and fine grained approaches. For the Coarse-grained approach, the dynamics are represented as transitions in the state of the CG_CatalogService object. That is, all of the behavior is expressed by the states and the state transitions of the CG_CatalogService object which is effected by the messages sent by the client (see Section 3.4). With the Fine-Grain approach, the behavior is represented as the sequence of interactions between various fine grained objects. Therefore, the Fine Grained behavior is represented by sequence diagrams (see Section 3.5).

A key aspect of interoperability for the OGC Catalog Interface is the ability to bridge across Distributed Computing Platforms (DCPs). Section 3.6 describes the dynamics of a Bridge that accepts Catalog requests in one profile and outputs the request in another profile. The simplicity of the Bridge is one measure of the ease of achieving interoperability across DCPs.

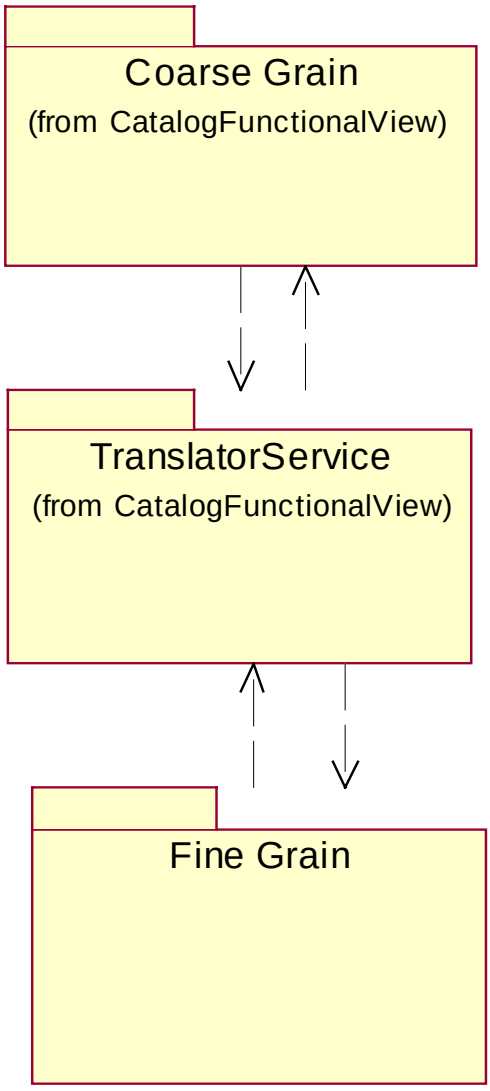


Figure 6 - Top Level General Model

3.2 Coarse-Grain Structural Model

3.2.1 Overview of The Coarse-Grained Interface Model

Figure 7 shows the Coarse-Grained service interfaces. These interfaces allow the discovery, access and management of geospatial data and services. This model is based on the concept of interface operations passing Request – Response Message Pairs between a client and a server. Stated another way, the Coarse-Grained architecture uses a messaging based structure to describe the access and invocation of Catalog services.

As seen in Figure 7 there are four major interfaces, CG_CatalogService, CG_Discovery, CG_Access and CG_CatalogManager. These are described in more detail in the following sections of this document. The taxonomy of interfaces that have been placed above the CG_CatalogService interface (i.e., OGC_Service and OGC_Stateful) have been created to put forth the idea of having an overall architectural framework for the different services that will be developed over time to populate the OGC Service Architecture. In the future, a Stateless Catalog service will be defined.

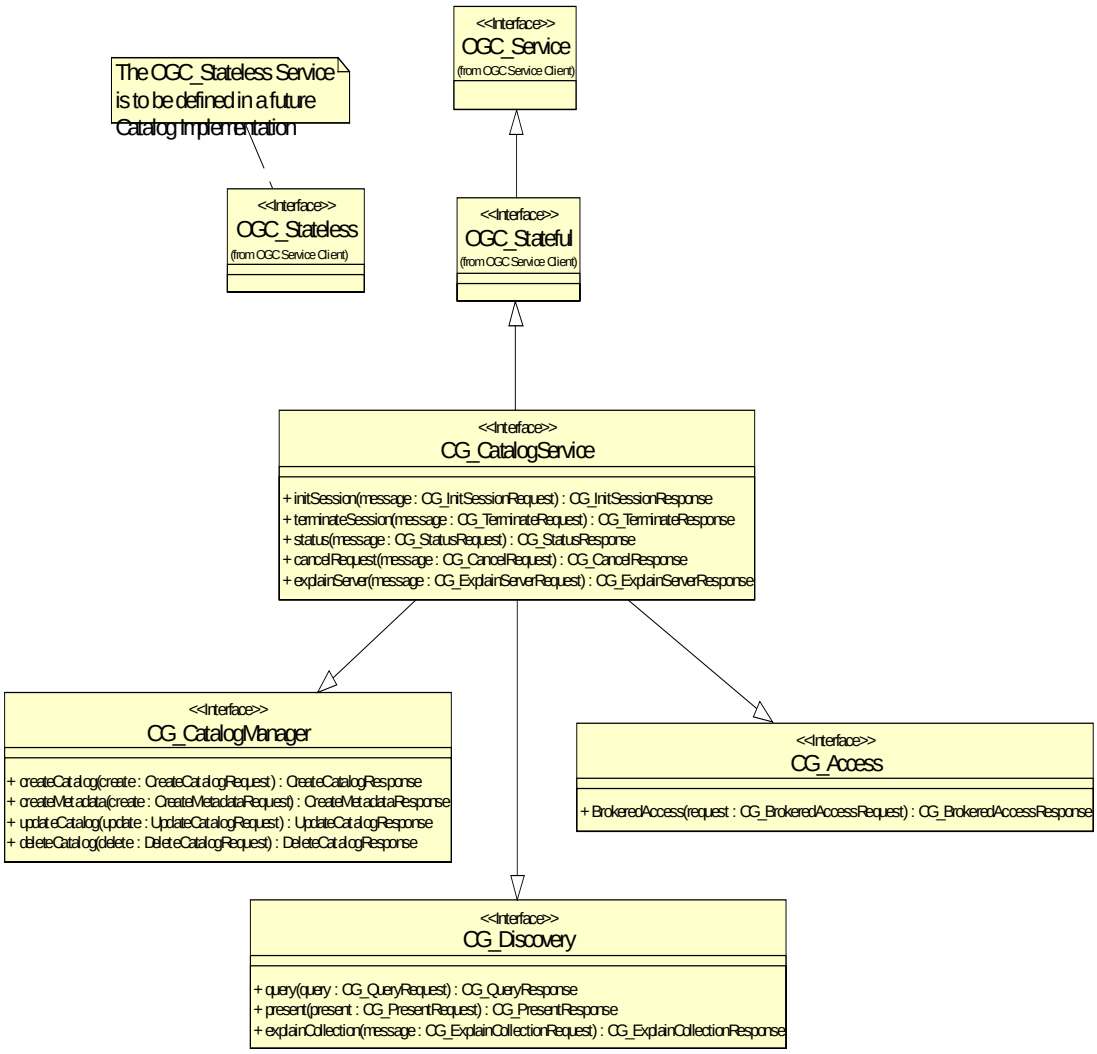


Figure 7 -Main Static Class Diagram of the Coarse-Grained General Interface Model

3.2.2 The Messaging Model

As previously noted, the Coarse-Grain general interface model is based on the passing of messages between a client and catalog server. To support this type of model, a messaging based structure has been developed to describe the access and invocation of catalog services. Figure 8, Figure 9, and Figure 10 are static class diagrams that depict this message based taxonomy.

Central to this taxonomy is the CG_Message class. CG_Message provides a consistent set of parameters that are populated for all messages. These parameters are used by the underlying implementation platform to perform message routing and session management. Subclasses of CG_Message are CG_Request and CG_Response. CG_Request messages encompass all messages from a client requesting a service from the server. CG_Response messages encompass all messages from a server generated in response to a client request. There is a one to one relationship between requests and responses. That is to say, for each request, one and only one response will be generated

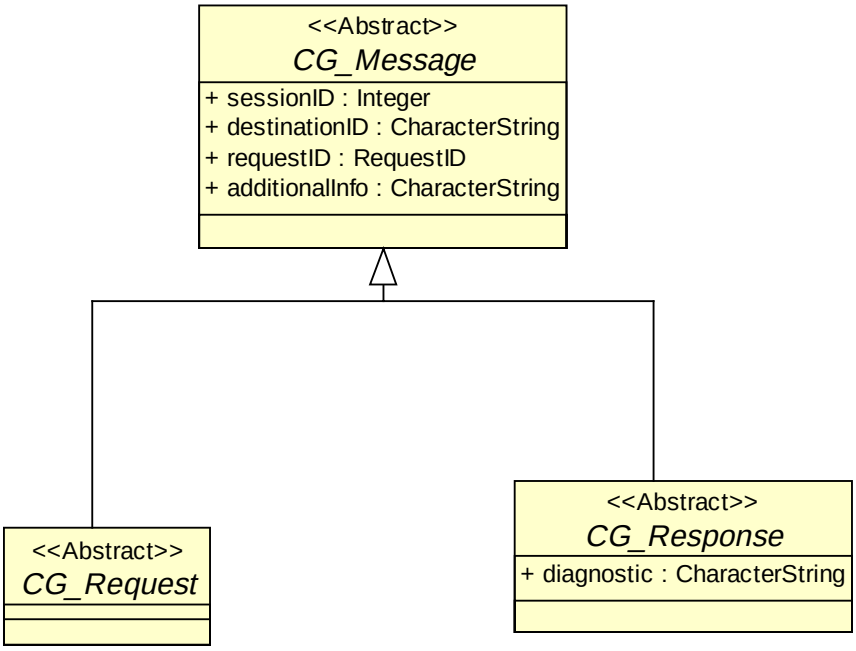


Figure 8 - Main Class Diagram for Message Package

3.2.2.1 The Message Class (CG_Message)

The CG_Message class defines the core set of parameters expected of each message exchanged between a client and server. These parameters support message routing and session management. All request and reply messages are subclasses of the CG_Message class.

CG_Message ::= sessionID destinationID requestID additionalInfo format

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

3.2.2.1.1 Message Parameters:

sessionID: Type = Integer

This is a unique identifier for this client/server session. The session identifier value is assigned in response to a CG_InitSessionRequest. All further messages within that session will contain that identifier in the sessionID parameter.

destinationID: Type = CharacterString

The DestinationID parameter identifies the target for this message. It can identify a server, service, or a process within a service.

requestID: Type = CG_RequestID

The RequestID parameter is an identifier unique to this message. In the case of a request message, this identifier can be used to monitor and control the processing resulting from the request message. The formal definition of the CG_RequestID data type is in Section 3.2.7.

additionalInfo: Type = CharacterString

This parameter provides a means of passing additional data that may only be relevant within the context of a specific message exchange.

3.2.2.1.2 Message Operations: None

3.2.2.2 Request Messages (CG_Request)

Catalog services are invoked by a client through request messages. Request messages include the parameters of the message class but do not add any of their own. All messages to invoke specific catalog services are subclasses of the CG_Request class.

CG_Request ::= sessionID destinationID requestID additionalInfo format

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

3.2.2.2.1 Message Parameters: None

3.2.2.2.2 Message Operations: None

3.2.2.3 Response Messages (CG_Response)

Response messages are used by the server to reply to client requests. The CG_Response class is the root class for all response messages constructed by the server in response to a client request.

CG_Response ::= sessionID destinationID requestID additionalInfo format diagnostic

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

diagnostic ::= CharacterString

3.2.2.3.1 Message Parameters:

diagnostic: Type = CharacterString

This parameter provides a means of passing diagnostic data relevant within the context of the specific message exchange.

3.2.2.3.2 Message Operations: None

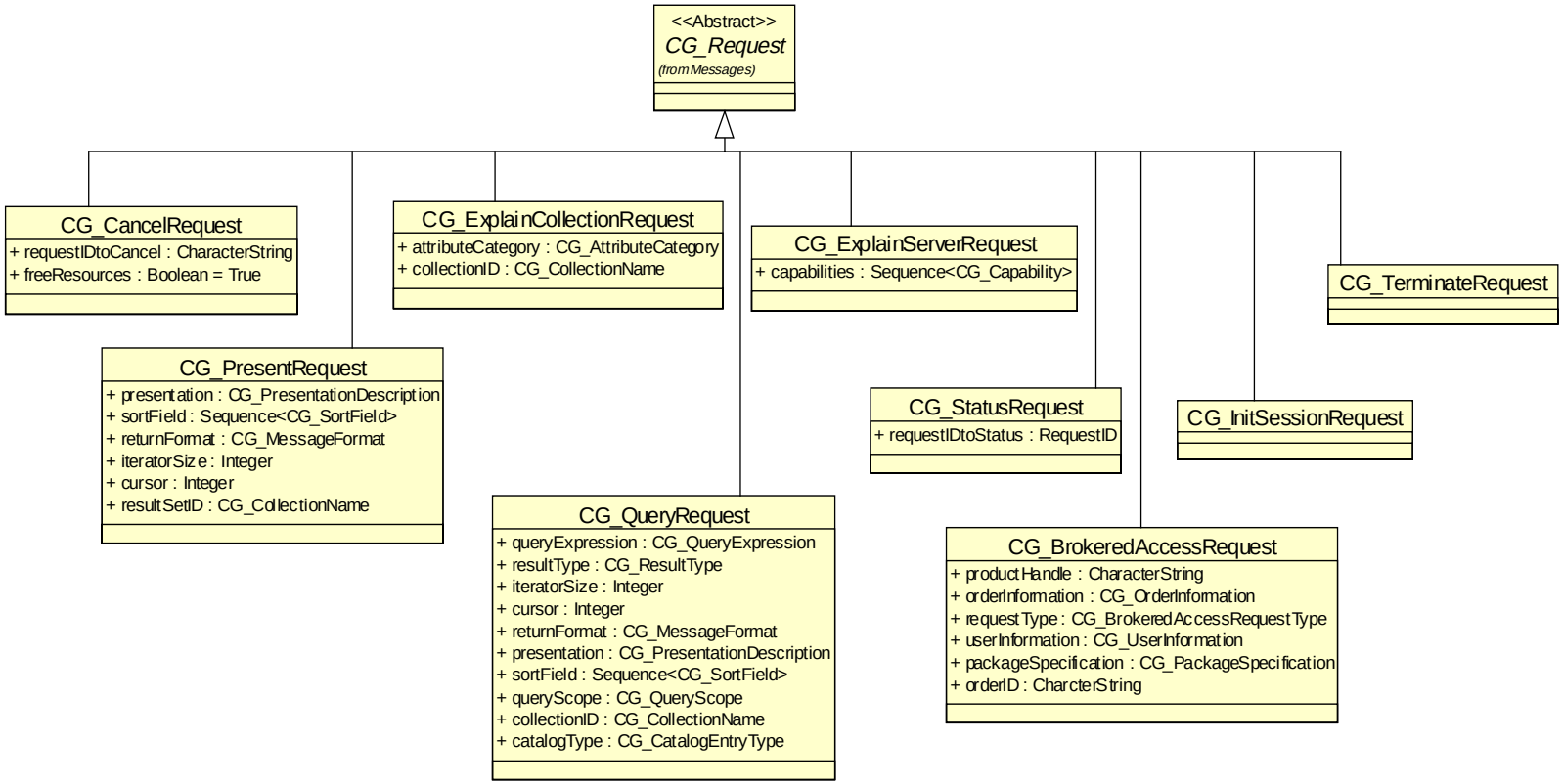


Figure 9 - Request Message Classes and Their Attributes (Parameters) Defined for OGC Coarse-Grain Catalog Interface Model

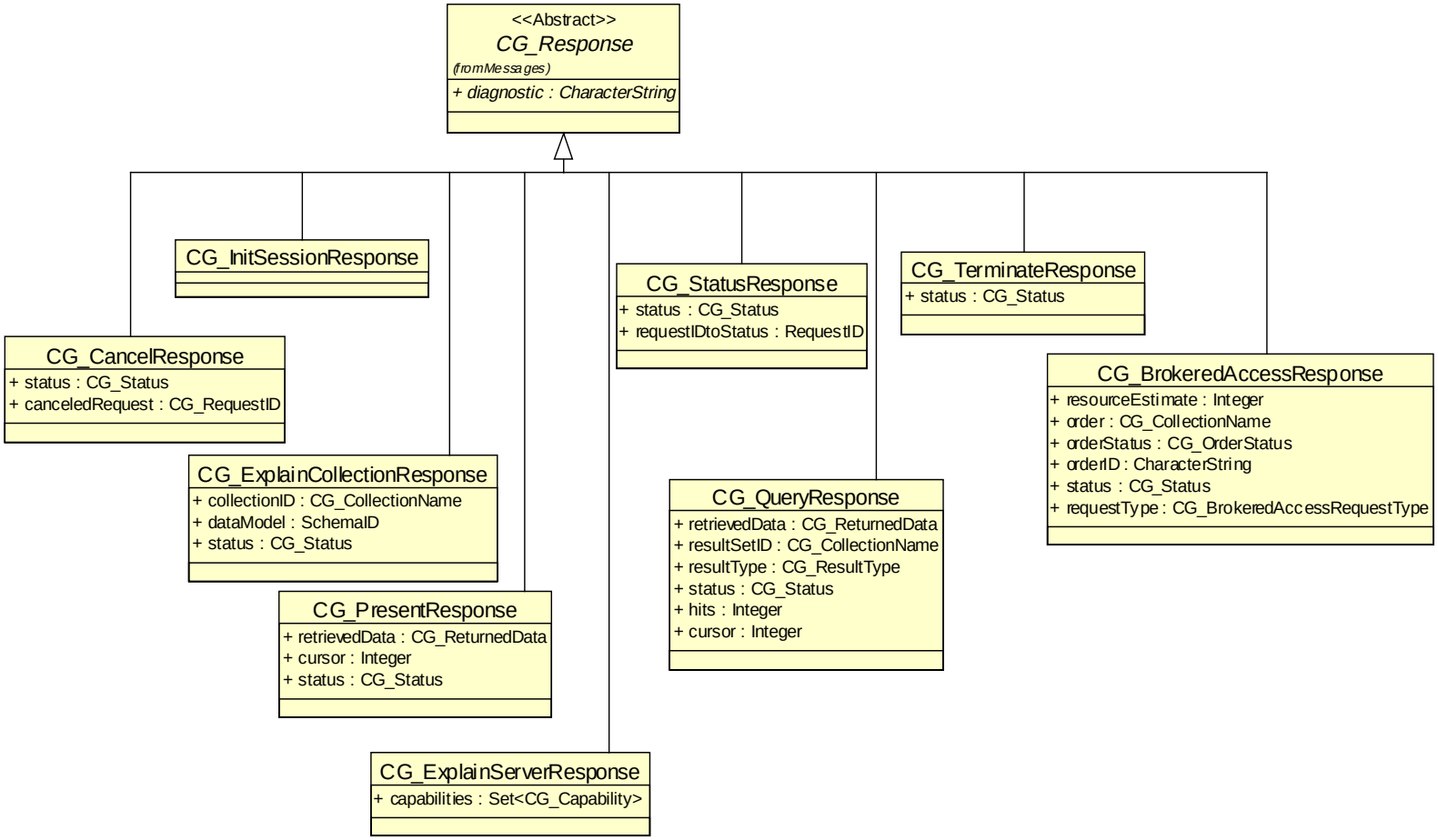


Figure 10 -Response Message Classes and Their Attributes (Parameters) Defined for OGC Coarse-Grained Catalog Interface Model.

3.2.3 CG_CatalogService Interface

Server level interfaces (i.e., those provided in the interface CG_CatalogService) provide access to the services that support the establishment and management of a user session. Core capabilities include the discovery of server capabilities, session initialization and termination and request status and termination. The specific operations put forth in the coarse-grained general model supporting the CG_CatalogService Server are listed in Table 2.

Table 2 - The Operations of the CG_CatalogService Interface

Operation Name	Input Message Type	Returned Message Type	Function Provided
InitSession	CG_InitSession-Request	CG_InitSessionResponse	This operation generates a unique identifier used to track the context of session.
TerminateSession	CG_TerminateRequest	CG_TerminateResponse	This operation terminates the session
Status	CG_StatusRequest	CG_StatusResponse	This operation is used to check on the status of a current pending request.
CancelRequest	CG_CancelRequest	CG_CancelResponse	This operation is used to terminate any request.
ExplainServer	CG_ExplainServer-Request	CG_ExplainServer-Response	This operation lists all the conventions and services available during the current session.

All request messages generated for these interfaces must specify the Catalog Server in the destinationID parameter, and the sessionID is also needed in some instances..

Catalog services are accessed through request messages, and the results returned through response messages. Errors in processing a request message are reported to the client by returning the appropriate response message using the diagnostic parameter to return the error status and error message and with all service specific parameters unpopulated.

3.2.3.1 CG_InitSessionRequest

The CG_InitSessionRequest message is used to establish a session between the Catalog Server and the Catalog Client. SessionID may be null in CG_InitSessionRequest. If a sessionID is supplied in CG_InitSessionRequest, the server is not obliged to accept the sessionID. The server is free to supply any SessionID in CG_InitSessionResponse.

CG_InitSessionRequest ::= sessionID destinationID requestID additionalInfo

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

Message Parameters: None

Message Operations: None

3.2.3.2 CG_InitSessionResponse

The CG_InitSessionResponse message is used to acknowledge the establishment of a session between the Catalog Server and the Catalog Client. This message provides the session identifier that will be used to establish the session context for each subsequent message.

CG_InitSessionResponse ::= sessionID destinationID requestID additionalInfo diagnostic

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

diagnostic ::= CharacterString

Message Parameters: None

Message Operations: None

3.2.3.3 CG_TerminateRequest

The CG_TerminateRequest message is used to terminate the current session. These messages originate at the client and are addressed to the catalog server. Upon receipt of the message, the Catalog Server will validate the message, stop all processing for that session and delete any queries and result sets.

CG_TerminateRequest ::= sessionID destinationID requestID additionalInfo

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

Message Parameters: None

Message Operations: None

3.2.3.4 CG_TerminateResponse

The CG_TerminateResponse message is used by the server to deliver back to a client the completion status of a CG_TerminateRequest.

CG_TerminateResponse ::= sessionID destinationID requestID additionalInfo diagnostic status

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

diagnostic ::= CharacterString

status ::= CG_Status

3.2.3.4.1 Message Parameters:

status: Type = CG_Status

The Status parameter conveys the success or failure of the terminate request.

3.2.3.4.2 Message Operations: None

3.2.3.5 CG_ExplainServerRequest

The CG_ExplainServerRequest message is used to expose and negotiate the services and conventions governing this session. CG_ExplainServerRequest messages originate at the client. They are initially populated with the properties desired by that client using the capabilities parameter. Each capabilities component can be populated with either a value or a "wildcard". When populated with a wildcard, the client is requesting the server to report on the options available for that capability. In response to a request, the server can confirm, deny or report on each capability. Capabilities requested by value are confirmed by returning the same capability/value pair as requested. Capabilities requested by value that are not supported by the server are denied by not returning that capability. When reporting the server returns all of the values supported for a requested capability. The CG_Capability data type is described in Section 3.2.7.3

CG_ExplainServerRequest ::= sessionID destinationID requestID additionalInfo capabilities

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

capabilities ::= Set<CG_Capability>

3.2.3.5.1 Message Parameters:

capabilities: Type = Set<CG_Capability>

The capabilities parameter passes a list of CG_Capability data types specifying the capabilities and conventions of interest to the user. CG_Capability is a complex data type that is described in Section 3.2.7.

3.2.3.5.2 Message Operations: None

3.2.3.6 CG_ExplainServerResponse

The CG_ExplainServerResponse message is used to expose and negotiate the services and conventions governing this session. The capabilities parameter is received from the Explain Server request and populated with the data desired. The details of populating the response are given in the CG_ExplainServerRequest. This parameter is then inserted into the response message and returned to the user.

CG_ExplainServerResponse ::= sessionID destinationID requestID additionalInfo

diagnostic capabilities

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

diagnostic ::= CharacterString

capabilities ::= Set<CG_Capability>

3.2.3.6.1 Message Parameters:

capabilities: Type = Set<CG_Capability>

The capabilities parameter contains a list of CG_Capability data types detailing the capability and convention information requested by the user. CG_Capability is a complex data type that is described in Section 3.2.7.

3.2.3.6.2 Message Operations: None

3.2.3.7 CG_StatusRequest

The CG_StatusRequest message is used by the client to discover the current status of any processing taking place as a result of a specific request. A CG_StatusResponse message is generated by the server returning the current status of the request.

CG_StatusRequest ::= sessionID destinationID requestID additionalInfo requestIDtoStatus

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

requestIDtoStatus ::= CG_RequestID

3.2.3.7.1 Message Parameters:

requestIDtoStatus: Type = CG_RequestID

The identifier of the Request about which the user desires information.

3.2.3.7.2 Message Operations: None

3.2.3.8 CG_StatusResponse

The CG_StatusResponse message is used by the server to deliver to the client the current status of any processing taking place relating to the specified request. These messages are generated by the server in response to a CG_StatusRequest message.

CG_StatusResponse ::= sessionID destinationID requestID additionalInfo requestIDtoStatus status

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

requestIDtoStatus ::= CG_RequestID

status ::= CG_Status

3.2.3.8.1 Message Parameters:

requestIDtoStatus: Type = CG_RequestID

The identifier of the Request which this message is delivering information about.

status: Type = CG_Status

The status parameter conveys the current status of the selected request.

3.2.3.8.2 Message Operations: None

3.2.3.9 CG_CancelRequest

The CG_CancelRequest message is used to terminate any request. It is assumed that in terminating a request that any result set or other resources associated with the request will be “garbage collected” by the server if freeResources is true. Upon receipt of the message, the Catalog Server will validate the message, stop all processing for the target and release appropriate resources dependent on the request.

CG_CancelRequest ::= sessionID destinationID requestID additionalInfo

requestIDtoCancel freeResources

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

requestIDtoCancel ::= CG_RequestID

freeResources ::= Boolean

3.2.3.9.1 Message Parameters:

requestIDtoCancel: Type = CG_RequestID

The identifier of the Request to be canceled.

freeResources: Type = Boolean

If set to FALSE, the partial result set is not deleted until the client terminates the session. Default value is TRUE.

3.2.3.9.2 Message Operations: None

3.2.3.10 CG_CancelResponse

The CG_CancelResponse message is used by the server to report on the success or failure of an attempt to cancel a request.

CG_CancelResponse ::= sessionID destinationID requestID additionalInfo diagnostic

status canceledRequest

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

diagnostic ::= CharacterString

status ::= CG_Status

canceledRequest ::= CG_RequestID

3.2.3.10.1 Message Parameters:

status: Type = CG_Status

Status indicator for whether the cancel request was successful or if there was an error.

canceledRequest: Type = CG_RequestID

Identifier for the request object that was the target of the cancel request.

3.2.3.10.2 Message Operations: None

3.2.4 CG_Discovery Interface

The CG_Discovery Interface provides users a way to discover what data, services and other resources are available to them. These interfaces do not provide access to the resources themselves, rather, they provide information on what the resources are and how to access them. The specific operations of CG_Discovery are found in Table 3.

Table 3 - The Operations of the CG_Discovery Interface

Operation Name	Input Message Type	Returned Message Type	Function Provided
Query	CG_QueryRequest	CG_QueryResponse	This operation is used to search for data/services from a given catalog server and may return records from the result set.
Present	CG_PresentRequest	CG_PresentResponse	This operation is used to retrieve records from a result set created from the issuance of a query.
ExplainCollection	CG_ExplainCollection Request	CG_ExplainCollection-Response	This operation is used to explain the data model of the catalog.

3.2.4.1 CG_QueryRequest

The CG_QueryRequest message is used to request that the Catalog Server create a subset (Result Set) of the catalog holdings or to further subset an existing Result Set. CG_QueryRequest messages originate at the client. They are populated with the criteria to be used to select the Result Set and parameters governing the scope of the query and the format of the response. Upon receipt of the message, the Catalog Server will identify those elements of the query space to be included in the Result Set and create a Result Set containing those elements. Response to the client will be through the CG_QueryResponse message. Timing of the response message is governed by the resultType parameter.

CG_QueryRequest ::= sessionID destinationID requestID additionalInfo queryExpression resultType
iteratorSize cursor returnFormat presentation sortField queryScope
collectionID catalogType

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

queryExpression ::= CG_QueryExpression

resultType ::= CG_ResultType

iteratorSize ::= Integer

cursor ::= Integer

returnFormat ::= CG_MessageFormat

presentation ::= CG_PresentationDescription

sortField ::= Set<CG_SortField>

queryScope ::= CG_QueryScope

collectionID ::= CG_CollectionName

catalogType ::= CG_CatalogEntryType

3.2.4.1.1 Message Parameters:

queryExpression: Type = CG_QueryExpression

The queryExpression parameter contains the criteria used to subset the search space. CG_QueryExpression is formally defined in Section 3.2.7

resultType: Type = CG_ResultType

The resultType parameter is used to specify how the user wants the result set to be presented. CG_ResultType is formally defined in Section 3.2.7

iteratorSize: Type = Integer

The iteratorSize parameter indicates the maximum number of result set entries to be returned in the CG_QueryResponse.

cursor: Type = Integer

The Cursor parameter identifies the first result set entry to be returned in the CG_QueryResponse.

returnFormat: Type = CG_MessageFormat

This parameter specifies the encoding standard to be used for returning the result set. CG_MessageFormat is formally defined in Section 3.2.7

presentation: Type = CG_PresentationDescription

The Presentation parameter is only valid when results are requested to be returned directly in the CG_QueryResponse. This parameter informs the server which of the attributes in the result set elements are to be returned to the client. The CG_PresentationDescription parameter is defined in Section 3.2.7

sortField: Type = Set(CG_SortField)

The sortField parameter specifies how the result set data is to be sorted prior to presentation. The CG_SortField type is defined in Section 3.2.7.

queryScope: Type = CG_QueryScope

The queryScope parameter is used to specify the size of the query space for distributed catalogs. CG_QueryScope is formally defined in Section 3.2.7.

See Section 2.10 for a discussion about distributed searching.

collectionID: Type = CG_CollectionName

This parameter identifies the search space for this query. A search space can be the catalog holdings, a result set, or a named subspace of the catalog holdings. CG_CollectionName is formally defined in section 3.2.7

catalogType: type = CG_CatalogEntryType

The catalogType parameter specifies the types of catalog entries to query. CG_CatalogEntryType is an enumerated code list formally defined in Section 3.2.7.

3.2.4.1.2 Message Operations: None

3.2.4.2 CG_QueryResponse

The CG_QueryResponse message is used by the server to report back to a client on the status of a CG_QueryRequest. The behavior of the CG_QueryResponse depends on the result type parameter as shown in Figure 16. Additionally, the contents of the CG_QueryResponse depends on the result type parameter.

CG_QueryResponse ::= sessionID destinationID requestID additionalInfo diagnostic
retrievedData resultSetID resultType status hits cursor

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

diagnostic ::= CharacterString

retrievedData ::= CG_ReturnData

resultSetID ::= CG_CollectionName

resultType ::= CG_ResultType

status ::= CG_Status

hits ::= integer

cursor ::= Integer

3.2.4.2.1 Message Parameters:

retrievedData: Type = CG_ReturnData

The retrievedData parameter contains a subset of the results of this query request. It is organized and formatted as specified in the presentation, messageFormat, and sortField parameters. This parameter is only populated if resultType = Results. A formal definition of the CG_ReturnData type can be found in Section 3.2.7.

resultSetID: Type = CG_CollectionName

This parameter identifies the Result Set generated for the query. Further query, present and cancel requests for this Result Set will supply this value through the collectionID parameter. The CG_CollectionName type is defined in Section 3.2.7.

resultType: Type = CG_ResultType

The resultType parameter indicates how the server responded to the query request. CG_ResultType is formally defined in Section 3.2.7.

status: Type = CG_Status

The Status parameter conveys the success or failure of the query request. The CG_Status type is formally defined later in Section 3.2.7.

hits: Type = Integer

Indication of the number of entries in the result set.

cursor: Type = Integer

The Cursor parameter identifies the last item in the result set that was returned in this retrieved data set.

3.2.4.2.2 Message Operations: none

3.2.4.3 CG_PresentRequest

The CG_PresentRequest message is used to request that the Catalog Server deliver a portion of a Result Set. CG_PresentRequest messages originate at the client. CG_PresentRequest messages are populated with the identifier for the Result Set and parameters governing the format of the response. Upon receipt of the message, the Catalog Server will build a subset of the Result Set based on the specified cursor location, the iterator size, and the attributes defined in the presentation parameter. This subset will then be returned to the client through the CG_PresentResponse message.

CG_PresentRequest ::= sessionID destinationID requestID additionalInfo resultSetID presentation
sortField returnFormat iteratorSize cursor

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

resultSetID ::= CG_CollectionName

presentation ::= CG_PresentationDescription

sortField ::= Set<CG_SortField>

returnFormat ::= CG_MessageFormat

iteratorSize ::= Integer

cursor ::= Integer

3.2.4.3.1 Message Parameters:

presentation: Type = CG_PresentationDescription

The Presentation parameter informs the server which of the attributes in the result set elements are to be returned to the client. Presentation serves the same function and has the same format as the corresponding parameter in the CG_QueryRequest message. The CG_PresentationDescription parameter is defined in Section 3.2.7.

sortField: Type = Set(CG_SortField)

The sortField parameter specifies how the result set data is to be sorted prior to presentation. The CG_SortField type is defined in Section 3.2.7

returnFormat: Type = CG_MessageFormat

This parameter specifies the encoding standard to be used for returning the result set. CG_MessageFormat is formally defined in Section 3.2.7.

iteratorSize: Type = Integer

The iteratorSize parameter indicates the maximum number of result set entries to be returned at one time.

cursor: Type = Integer

The Cursor parameter identifies the first result set entry to be accessed when traversing the result set.

3.2.4.3.2 Message Operations: None

3.2.4.4 CG_PresentResponse

The CG_PresentResponse message is used by the server to deliver to a client a subset of the Result Set. These messages are generated by the server in response to a CG_PresentRequest message.

CG_PresentResponse ::= sessionID destinationID requestID additionalInfo diagnostic retrievedData
cursor hits status

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

diagnostic ::= CharacterString

retrievedData ::= CG_ReturnData

cursor ::= Integer

hits ::= Integer

status ::= CG_Status

3.2.4.4.1 Message Parameters:

retrievedData: Type = CG_ReturnData

The retrievedData parameter contains a subset of the results of the query request. It is organized and formatted as specified in the presentation, returnFormat, and sortField parameters. A formal definition of the CG_ReturnData type can be found in Section 3.2.7.

cursor: Type = Integer

The Cursor parameter identifies the last item in the result set that was returned in this retrieved data set.

hits: Type = Integer

Indication of the number of entries in the result set.

status: Type = CG_Status

The Status parameter conveys the success or failure of the query request. The CG_Status type is formally defined in Section 3.2.7

3.2.4.4.2 Message Operations: None

3.2.4.5 CG_ExplainCollectionRequest

The CG_ExplainCollectionRequest inquires for information on the data taxonomy (model) of a particular catalog or catalog collection.

CG_ExplainCollectionRequest ::= sessionID destinationID requestID additionalInfo

attributeCategory collectionID returnFormat

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

attributeCategory ::= CG_AttributeCategory

collectionID ::= CG_CollectionName

returnFormat ::= CG_MessageFormat

3.2.4.5.1 Message Parameters:

attributeCategory: Type = CG_AttributeCategory

This parameter allows the client to specify the types of attributes that they want data about. Currently defined values are queriable, presentable and all. CG_AttributeCategory is formally defined in Section 3.2.7.

collectionID: Type = CG_CollectionName

This parameter specifies the collection for which the client wants the data structure explained. CG_CollectionName is formally defined in Section 3.2.7.

3.2.4.5.2 Message Operations: None

3.2.4.6 CG_ExplainCollectionResponse

The CG_ExplainCollectionResponse returns the requested information on the data taxonomy of the selected catalog collection.

CG_ExplainCollectionResponse ::= sessionID destinationID requestID additionalInfo diagnostic

collectionID dataModel returnFormat

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

diagnostic ::= CharacterString

collectionID ::= CG_CollectionName

dataModel ::= CG_SchemaID

returnFormat ::= CG_MessageFormat

3.2.4.6.1 Message Parameters:

collectionID: Type = CG_CollectionName

This parameter specifies the collection from which the dataModel parameter was derived. CG_CollectionName is formally defined in Section 3.2.7.

dataModel: Type = CG_SchemaID

This parameter provides the data model information requested by the client. CG_SchemaID is formally defined in Section 3.2.7.

status: Type = CG_Status

The Status parameter conveys the success or failure of the request. The CG_Status type is formally defined in Section 3.2.7

3.2.4.6.2 Message Operations: None

3.2.5 CG_Access Interface

The CG_Access Interface provides the user with a means to access the items located through the Discovery service. Access is divided into two categories, direct and brokered. Direct access is for those resources that are readily available over public interfaces such as the OGC Simple Features and Catalog. Methods for Direct Access are outside of the scope of the Catalog Interface, although the Catalog Interface will return a "handle" to the client to allow Direct Access. Not all resources can be accessed directly. Brokered access provides interfaces for gaining access to resources that are controlled. Controlled resources could include those for which the following applies: 1) a fee is charged, 2) have security limitations, 3) require additional processing or 4)are not available electronically. The brokered access operation provides a means for the user to provide the necessary information to request access to a resource (i.e., order) and for the owner to provide the data necessary to achieve that access.

3.2.5.1 CG_BrokeredAccessRequest

The CG_BrokeredAccessRequest is a service requesting data that cannot be made available directly.

CG_BrokeredAccessRequest ::= sessionID destinationID requestID additionalInfo
productHandle orderInformation orderID requestType
userInformation statusOrderUpdateType packageSpecification

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

productHandle ::= CharacterString

orderInformation ::= CG_OrderSpecification

orderID ::= CharacterString

requestType ::= CG_BrokeredAccessRequestType

userInformation ::= CG_UserInformation

statusOrderUpdateType ::= CG_StatusUpdateType

packageSpecification ::= CG_PackageSpecification

3.2.5.1.1 Message Parameters:

productHandle: Type = CharacterString

The product handle is the identifier for a specific product taken from the catalog metadata for that product.

orderInformation: Type = CG_OrderSpecification

The specification of the order request as provided as input by the client if CG_BrokeredAccessRequestType = orderEstimate or OrderQuoteAndSubmit. For CG_BrokeredAccessRequestType = orderMonitor or orderCancel, CG_OrderSpecification shall be empty.

orderID: type = CharacterString

The orderID parameter provides a unique identifier for an order in progress. This ID can be used to inquire about the status of the order as it is being processed. For CG_BrokeredAccessRequestType = orderMonitor or orderCancel, orderID shall be supplied. For requestType = orderEstimate or OrderQuoteAndSubmit, orderID shall be empty.

requestType: Type = CG_BrokeredAccessRequestType

The request type parameter identifies the type of service the client needs from the server. Valid values are estimate, submit, monitor and cancel. Estimate is used to check if the order is valid and to request an estimate of resources required to fill the order. Submit is a request to order and deliver the products(s). Monitor provides the current status of the order. Cancel requests that the order be cancelled. Cancellation of the order must be granted by the server.. CG_BrokeredAccessRequestType is formally defined in Section 3.2.7.

userInformation: Type = CG_UserInformation

To receive products it is necessary to provide requester identification, billing and delivery data as part of the order. This parameter is used to provide that data.

statusOrderUpdateType : Type = CG_StatusUpdateType

How a given client would like to be kept informed about the status of a given order.

packageSpecification: Type = CG_PackageSpecification

The specification of a single package of a requested order, or multiple packages.

3.2.5.1.2 Message Operations: None

3.2.5.2 CG_BrokeredAccessResponse

The CG_BrokeredAccessResponse message is generated by the server in response to a CG_BrokeredAccessRequest.

CG_BrokeredAccessResponse ::= sessionID destinationID requestID additionalInfo diagnostic format orderStatus resourceEstimate order orderID status requestType

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

diagnostic ::= CharacterString

format ::= CG_MessageFormat

orderStatus ::= CG_OrderStatus

resourceEstimate ::= Sequence<Integer>

order ::= CG_CollectionName

orderID ::= CharacterString

status ::= CG_Status

requestType ::= CG_BrokeredAccessRequestType

3.2.5.2.1 Message Parameters:

orderStatus Type ::= CG_OrderStatus

This parameter indicates the status of the order. The status of the order is different than the status of a CG_Access message. The status of the message is reported in the response in the status parameter. The CG_OrderStatus type is formally defined in Section 3.2.7 of this specification.

resourceEstimate: Type = Sequence<Integer>

This parameter reports back on the resources needed to process and/or deliver the requested resource.

order: Type = CG_CollectionName

The order parameter returns a name or id of the requested product object which can be used for direct access (such as through simple features). The CG_CollectionName type is formally defined in Section 3.2.7 of this specification.

orderID: type = CharacterString

The orderID parameter provides a unique identifier for an order in progress. This ID can be used to inquire about the status of the order as it is being processed.

status: Type = CG_Status

The Status parameter conveys the status of the requested product. The CG_Status type is formally defined in Section 3.2.7.

requestType: Type = CG_BrokeredAccessRequestType

The request type parameter identifies the type of service the client needs from the server. CG_BrokeredAccessRequestType is formally defined in Section 3.2.7.

3.2.5.2.2 Message Operations: None

3.2.6 CG_CatalogManager Interface

The CG_CatalogManager Interface provides for the maintaining and updating of a catalog service. The operations defined for this interface are listed in Table 4

Table 4 - The Operations of the CG_CatalogManager Interface

Operation Name	Input Message Type	Returned Message Type	Function Provided
createCatalog	CG_CreateCatalogRequest	CG_CreateCatalogRequest	This operation would be done to start the process of creating a new catalog or a new set of catalog entries of an existing catalog service.
createMetadata	CG_CreateMetadataRequest	CG_CreateMetadataResponse	This operation would be initiated to create metadata about a given set of products held in a catalog.
updateCatalog	CG_UpdateCatalogRequest	CG_UpdateCatalogResponse	This operation is used to update the contents of a given catalog service.
deleteCatalog	CG_DeleteCatalogRequest	CG_DeleteCatalogResponse	This operation is used to delete the contents of a given catalog service entry (entries).

3.2.6.1 CG_CreateCatalogRequest

This message is used by a client with the appropriate user privileges to add new information to a catalog service.

CG_CreateCatalogRequest ::= sessionID destinationID requestID additionalInfo

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

Message Parameters: TBD

Message Operations: TBD

3.2.6.2 CG_CreateCatalogResponse

This message is used/sent by the server to acknowledge/accept the request to add new information to the catalog.

CG_CreateCatalogResponse ::= sessionID destinationID requestID additionalInfo diagnostic

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

diagnostic ::= CharacterString

Message Parameters: TBD

Message Operations: TBD

3.2.6.3 CG_CreateMetadataRequest

This message is used to by a client that has the appropriate user privileges to add metadata entries to a catalog.

CG_CreateCatalogRequest ::= sessionID destinationID requestID additionalInfo

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

Message Parameters: TBD

Message Operations: TBD

3.2.6.4 CG_CreateMetadataResponse

This message is used/sent by the server to acknowledge/accept the request to add new metadata entries to the catalog.

CG_CreateCatalogResponse ::= sessionID destinationID requestID additionalInfo diagnostic

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

diagnostic ::= CharacterString

Message Parameters: TBD

Message Operations: TBD

3.2.6.5 CG_UpdateCatalogRequest

This message is used to by a client that has the appropriate user privileges to update various types of information (e.g., data or metadata) to a catalog.

CG_UpdateCatalogRequest ::= sessionID destinationID requestID additionalInfo

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

Message Parameters: TBD

Message Operations: TBD

3.2.6.6 CG_UpdateCatalogResponse

This message is used/sent by the server to acknowledge/accept the request to update various types of information (e.g., data or metadata) to a catalog.

CG_UpdateCatalogRequest ::= sessionID destinationID requestID additionalInfo diagnostic

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

Message Parameters: TBD

Message Operations: TBD

3.2.6.7 CG_DeleteCatalogRequest

This message is used to by a client that has the appropriate user privileges to delete various types of information (e.g., data or metadata) to a catalog.

CG_UpdateCatalogRequest ::= sessionID destinationID requestID additionalInfo

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

Message Parameters: TBD

Message Operations: TBD

3.2.6.8 CG_DeleteCatalogResponse

This message is used/sent by the server to acknowledge/accept the request to delete various types of information (e.g., data or metadata) to a catalog.

CG_UpdateCatalogRequest ::= sessionID destinationID requestID additionalInfo diagnostic

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

Message Parameters: TBD

Message Operations: TBD

3.2.7 Parameter Type Definitions

This section provides definitions for all of the parameter data types used in Request-Response Message Pairs. These definitions assume the use of the OGC well known data types where applicable.

3.2.7.1 CG_AttributeCategory

Type: Code_List

Used By: CG_ExplainCollectionRequest

CG_AttributeCategory is a code list for selecting the types of catalog entry attributes to be exposed by an explain request. The valid values for this type are the following:

queriable

presentable

both

3.2.7.2 CG_BrokeredAccessRequestType

Type: Code_List

Used By: CG_BrokeredAccessRequest

CG_BrokeredAccessRequestType is a code list for identifying the nature of a brokered access request. Valid values for this type are shown in Table 5

Table 5 - Brokered Access Request Types

Value	**Explanation**
orderEstimate	Validate and obtain the estimate of an order specification
orderQuoteAndSubmit	Obtain a quote and subsequently submit an order specification
orderMonitor	Monitor the progress of an order request
orderCancel	Cancel an order request

3.2.7.3 CG_Capability

Type: Complex data structure

Used By: CG_ExplainServerRequest, CG_ExplainServerResponse

Uses: CG_AllSupportedRequest, CG_Defaults, CG_Explain, CG_Query, CG_Messaging, CG_Session, CG_SoftwareInformation, CG_SupportedCollections

CG_Capability is a super class for organizing descriptions of catalog capabilities and session conventions. It is a collection of subtypes (subclasses) which can be aggregated together into a single data entity in the CG_ExplainServerRequest and CG_ExplainServerResponse. Each subtype addresses a specific piece of data relating to the interactions between a client and the server Figure 11 shows the Capability Class and its subclasses that have been defined for the Coarse-Grained Catalog General Interface Model.

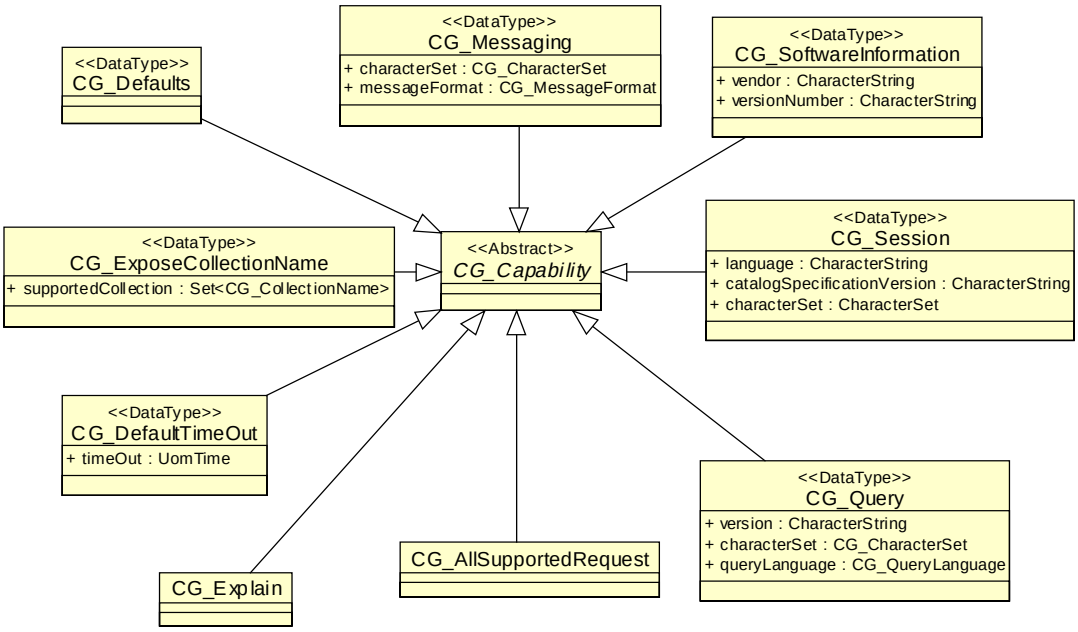


Figure 11 - Static Class Diagram showing CG_Capability Class and Its Instantiated Subtypes.

3.2.7.3.1 CG_AllSupportedRequest

Type: Boolean

Used By: CG_Capability

This parameter is only submitted as part of a request message from the client. When this parameter is set, the response shall include a complete list of all capabilities supported by the server.

3.2.7.3.2 CG_Defaults

Type: Boolean

Used By: CG_Capability

CG_Defaults is a Boolean parameter which when set in a request message causes the return of all of the default settings of that particular catalog server.

3.2.7.3.3 CG_DefaultTimeOut

Type: DataType

Used By: CG_Capability

The default time out that a client can set for a session. After a period of no activity in a session, the server may unilaterally close a session without notification to the client (see Section 3.4.2). The server should be prepared to respond to client request for a session which has timed out by returning the paired response containing a diagnostic indicating that the session does not exist. The single parameter in CG_DefaultTimeOut is the default time out and it is specified using the UomTime data type from the OGC Basic Package[1], Unit of Measure.

[1] OGC Basic Package: see OpenGIS project document 99-005r3, January 1993.

3.2.7.3.4 CG_Explain

Type: Boolean

Used By: CG_Capability

CG_Explain is a Boolean type for expressing whether or not the explain Collection facility is provided by a catalog server.

3.2.7.3.5 CG_Messaging

Type: Data structure

Used By: CG_Capability

Uses: CG_CharacterSet, CG_MessageFormat

The CG_Messaging parameter is a data structure containing data describing the messaging conventions a particular server observes. Sub-types of the messaging type are:

characterSet: (type = CG_CharacterSet) describes the character sets supported

messageFormat: (type = CG_MessageFormat) describes the formatting of the messages.

3.2.7.3.6 CG_Query

Type: data structure composed of version, characterSet and queryLanguage fields

Used By: CG_Capability

Uses: CG_QueryLanguage, CG_CharacterSet

This parameter provides information on one of the query languages supported by the server. This is a data structure composed of the following elements:

version: (type = character string) specifies the version of queryLanguage supported.

characterSet: (type = CG_CharacterSet) specifies the expected character set.

queryLanguage: (type = CG_QueryLanguage) specifies the query language supported.

3.2.7.3.7 CG_QueryLanguage

Type: Code_List

Used By: CG_Query, CG_QueryExpression

This code list contains the query languages supported by a given catalog server that the client has initiated a session with. OGC_Common is the default for all implementations. The list of query languages follows:

OGC_Common

Z3950_TypeOne

SQL3_SimpleFeature

SQL2_SimpleFeature

The OGC_Common query language is defined in Section 4. All implementations must support OGC_Common.

3.2.7.3.8 CG_Session

Type: Data Structure

Used By: CG_Capability

Uses: CG_CharacterSet

The CG_Session parameter contains data describing the constraints on any sessions supported by a server. This is a data structure containing the following elements:

language: (type = Character String) – language supported by the interface

catalogSpecificationVersion: (type = Character String) – OGC Catalog compliance version

characterSet: (type = CG_CharacterSet) – character set used for text encoding

3.2.7.3.9 CG_SoftwareInformation

Type: Data structure

Used By: CG_Capability

This parameter is a CG_Capability type used to identify the vendor and version number of the server software suite. CG_SoftwareInformation is a data structure containing the following elements:

vendor: (type = Character String) – name of the software manufacturer

SWversionNumber: (type = Character String) – version number of this release

IFversionNumber (type = Character String) – version number of OGC Catalog Interface supported by the software suite.

3.2.7.3.10 CG_SupportedCollections

Type: set(CG_CollectionName)

Used By: CG_ExposeCollectionName

Uses: CG_CollectionName

A capability used for requesting and returning the collections that the server has knowledge of and can provide access to a client request.

3.2.7.4 CG_CatalogEntryType

Type: Code_List

Used By: CG_QueryRequest, CG_PresentResponse

A catalog contains several different types of data. This parameter provides for the selection of one of those types for processing. It is implemented as a code list that takes the following values:

product - the lowest level Feature accessible

collection - a Feature Collection

catalog - a server which provides metadata about Features and Collections

service - a implementation of software that acts on Features.

3.2.7.5 CG_CharacterSet

Type: Code_List

Used By: CG_Messaging, CG_Query, CG_Session

This parameter type represents one of the standard computer character representation systems. It is implemented as a code list that takes the following values:

ASCII

UniCode

Shift-JIS

3.2.7.6 CG_CollectionName

Type: Union data

Used By: CG_QueryRequest, CG_QueryResponse, CG_ExplainCollectionRequest, CG_ExplainCollectionResponse, CG_BrokeredAccessResponse, CG_ReturnData

Collection Name is a type that identifies a catalog data resource. It can point to a catalog, catalog entry, named catalog subspace, named catalog superspace or a result set. This type is a union of two base types; collection ID (character string) and collection Name (character string).

3.2.7.7 CG_Dictionary

Type: Interface

Used By: CG_Schema

A dictionary is a simple look-up and query mechanism that associates keys (of a selected type) to values (of another selected type). Most commonly used for finding attributes by name within a tuple or tuple-like associative memory entity, but may also be used in other similar cases where structure is not important.

select – return a value using a key

insert – add a value using a key

delete – delete a value using a key

keylist – list the keys

3.2.7.8 CG_MessageFormat

Type: Code_List

Used By: CG_QueryRequest, CG_PresentRequest, CG_Messaging

CG_MessageFormat is an enumerated code list of the available formats for encoding a returned data set. Valid values for this type are:

XML

HTML

TXT

3.2.7.9 CG_OrderItem

Type: Data Structure

Used by: GC_BrokeredAccessRequestType

This data structure contains the specification of a single order item (i. e. the product that is ordered and that is to be delivered):

- productId, which is the identifier of the ordered product.
- productPrice, which is the price of the product.
- productDeliveryOptions, which contains delivery options for the product.
- processingOptions, which specifies the processing options that are to be applied on the product before delivery.
- sceneSelectionOptions, which specifies the selection of the scene from the whole product that is to be delivered.

3.2.7.10 CG_OrderSpecification

Type: Data Structure

Used By: CG_BrokeredAccessRequest

The specification of the order request as provided as input by the client if CG_BrokeredAccessRequestType = orderEstimate or OrderQuoteAndSubmit.

The structure contains the following information about the product specification:

orderCentreID – identifies the order center at which the order will be performed

orderPrice –the price for the whole order

orderDeliveryDate - the latest date at which the order can be expected to be delivered to the user.

orderCancellationDate – the latest date at which the user can cancel the order.

deliveryMethod – how the order will be delivered to the user: e-mail, ftp or mail.

package – contains the definition of how the packages which compose the order

3.2.7.11 CG_OrderStatus

Type: Code_List

Used By: CG_BrokeredAccessResponse

CG_ OrderStatus is a code list for identifying the status of an order. Valid values for this type are:

Table 6 - Order Status Codes

Value	Explanation
orderBeingEstimated	the order is currently being estimated by the target order handling system. An Estimate is an approximation only.
orderEstimated	indicates that the order has been successfully validated and that an estimate is provided.
orderBeingQuoted	the order is currently being quoted by the target order handling system. A Quote shall be considered contractually binding.
orderBeingProcessed	the order is currently being processed by the target order handling system.
orderCompleted	processing of order has been completed.
orderNotValid	the order has not been successfully validated.
orderCancelled	the order has been cancelled

3.2.7.12 CG_PackageSpecification

Type: Data Structure

Used By: CG_BrokeredAccessRequest, CG_OrderSpecification, CG_PackagingType

The specification of a single package, or multiple packages.

The structure contains the following information about the packaging order:

packageId – the identifier of the ordered package

packagePrice –the price for the package

package – the detailed information concerning the specification of package. (see packagingType)

packageMedium –the medium on which the package will be delivered to a user.

packageSize – the size of the package in kilobytes.

3.2.7.13 CG_PackagingType

Type: Code List

Used By: CG_PackageSpecification, CG_BrokeredAccessRequest

The specification of the packaging method being used to deliver an order to a user.

predefinedPackage: A package predefined by the given catalog service

adhocPackage: A package constructed of OrderItems to fulfil a particular order

3.2.7.14 CG_PaymentMethod

Type: Code_List

Used By: CG_UserInformation

This code list contains the payment methods for an order secured through using a CG_Access operation. The supported methods are the following:

credit

cash

purchaseOrder

3.2.7.15 CG_PredefinedPresentationType

Type: Code_List

Used By: CG_PresentationDescription

This parameter is a code list defining pre-defined query presentation descriptions supported by a data server. Current values that this parameter can take are:

full - includes all defined standard elements from the information community schema. This is a large set of elements, but it ensures that clients receive everything their users may need to evaluate the retrieval record for further processing. Note that, while all schema elements are returned, some elements may be meaningless for the record that is actually returned, and may contain undefined values.

brief - includes a minimal subset of the defined standard information community schema elements available from the appropriate database schema.

3.2.7.16 CG_PresentationDescription

Type: Data Union

Used By: CG_QueryRequest, CG_QueryResponse

Uses: CG_PredefinedPresentationType, TupleType

This parameter type contains the name and types of the requested attributes that will be returned by a query or present request. Alternately, this parameter may be the name of a Predefined Presentation Type.

attributes: (type = sequence<TupleType>) – list of attribute name/type pairs

name: (type = CG_PredefinedPresentationType) –), identifying a predefined presentation type.

3.2.7.17 CG_QueryExpression

Type: Data Structure

Used By: CG_QueryRequest

Uses: CG_QueryLanguage

CG_QueryExpression contains a description of the query language being used and the query string. The query string is a character string. The query language is specified using the CG_QueryLanguage type.

theQuery: (type = CharacterString) – the text defining the query

theNamespace: (type = CharacterString) – where the attributes used in theLanguage are defined.

theLanguage: (type = CG_QueryLanguage) – the query language being used

3.2.7.18 CG_QueryScope

Type: Code_List

Used By: CG_QueryRequest

CG_QueryScope is a code list describing the size of the search space for a query. Current valid values for this type are:

distributed

local

See Section 2.10 for a discussion of distributed search behavior.

3.2.7.19 CG_RequestID

Type: Data Structure

Used By: CG_Message, CG_StatusRequest, CG_CancelRequest, CG_CancelResponse

CG_RequestID is a compound number used to uniquely identify a specific request in a global context. These parameters are created by the client from two values, the SessionID and a counter. The Session ID provides a globally unique identifier for this request context. A counter provides a session unique identifier. Joined together, they form a globally unique identifier for a request.

sessionID: (type = uint) – globally unique session identifier

counter: (type = uint) – session unique identifier

3.2.7.20 CG_ResultType

Type: Code_List

Used By: CG_QueryRequest, CG_QueryResponse

CG_ResultType is a code list describing the type of data to be returned in a query response message and the behavior of the message response (see Figure 16). Current valid values for this type are:

validate - the CG_QueryResponse is returned as soon as CG_QueryRequest has been determined to be valid. Query processing continues after the CG_QueryResponse is returned. CG_Status will be set to 'failure' in case of an invalid query and to 'processing ' in case of a valid query. Reasons for failure are provided in the diagnostic of CG_QueryResponse.

resultSetID - the CG_QueryResponse is returned as soon as the resultSetID is available and the query has completed processing.

hits- the CG_QueryResponse is returned as soon as the query has completed processing and the number of hits has been determined. Metadata records are not returned in the CG_QueryResponse

results - the CG_QueryResponse is returned as soon as the query has completed processing and the results have been formatted for return. Metadata records are returned in the CG_QueryResponse

3.2.7.21 CG_ReturnData

Type: Data Structure

Used By: CG_QueryResponse, CG_PresentResponse

Uses: CG_MessageFormat

CG_ReturnData is a data type for packaging result set elements for return to the client. This data structure contains two components. The encoding component identifies the technique used to encode the result set data. The payload component contains the actual encoded data.

encoding: (type = CG_MessageFormat) – this component identifies the encoding technique used to package the catalog data. It is of type CG_MessageFormat which is defined in Section 3.2.7.8.

payload: (type = CharacterString) – payload is a "blob" for holding the returned catalog data. The structure of this component is defined by the encoding parameter.

3.2.7.22 CG_Schema

Type: Complex Data

Used By: CG_SchemaID

An OGC class containing a FeatureClassDescription. FeatureClassDescriptions are dictionaries with character string keys and TupleType values. This allows name-type pairs (TupleTypes) to be aggregated into a larger named context.

Sample Feature Collection CG_Schema: A Feature Collection is a class with a set of attributes and a set of aggregated features. Note that a Feature Collection is a type of feature. A Feature Collection is represented using CG_Schema as follows:

CG_Schema(featureCollection) = Dictionary (Attribute1, TupleType(Attribute1);
Attribute2, TupleType(Attribute2);
Feature1, TupleType(Feature1);
Feature2, TupleType(Feature2);
Feature3, TupleType(Feature3))

TupleType(Atttribute1) = (Attribute1, CG_Schema(Attribute1))

TupleType(Feature1) = (Feature1, CG_Schema (Feature1))

The other attributes and features in the feature collection have the same Tuple Types as shown here.

Sample Feature CG_Schema: A Feature is a class with a set of attributes. A Feature is represented using CG_Schema as follows:

CG_Schema(feature) = Dictionary (Attribute3, TupleType(Attribute3);
Attribute4, TupleType(Attribute4))

Sample Attribute schema: The minimal schema for an Attribute is defined in Table 7 which is based on ISO/IEC 11179 - Information technology -- Specification and standardization of data elements. An Attribute Schema is represented using CG_SchemaID as follows:

CG_Schema(Attribute1) = Dictionary (Name, valueName;
Definition, valueDefinition;
Representation Category, valueRepresentationCategory;
Form of Representation, valueFormRepresentation;
Datatype of data element values, valueDatatype)

Table 7 - Minimal Mandatory Attribute Definitions

Name	Single or multi word designation assigned to a data element.
Definition	Statement that expresses the essential nature of a data element and permits its differentiation from all other data elements.
Representation Category	Type of symbol, character or other designation used to represent a data element.
Form of Representation	Name or description of the form of representation for the data element, e.g. 'quantitative value', 'code', 'text', 'icon'.
Datatype of data element values	A set of distinct values for representing the data element value.

3.2.7.23 CG_SchemaID

Type: Union Data

Used By: CG_ExplainCollectionResponse

Uses: CG_Schema, SchemaName

The CG_SchemaID is a data type used to represent the schema of a data, feature or catalog collection. It is a union of two elements, a named identifier for a well known schema, or an element of type CG_Schema.

schemaName : (type = CharacterString)

schema := (type = CG_Schema)

3.2.7.24 CG_SortField

Type: Data Structure

Used By: CG_QueryRequest, CG_PresentRequest

Uses: CG_SortOrder

CG_SortField provides sorting information to the server for formatting data returned to the client. This type consists of an attribute name and sort order descriptor. The attribute name identifies the result set attribute type to be sorted on. The sort order descriptor is of the CG_SortOrder type.

attributeName: (type = character string) – name of attribute to sort on

sortOrder: (type = CG_SortOrder) – how the attributes are to be ordered by the sort

3.2.7.25 CG_SortOrder

Type: Code_List

Used By: CG_SortField

CG_SortOrder is an enumerated code list for defining how a value is to be sorted. The current valid values for this type are shown in Table 8

Table 8 - Sort Order Operations

OPERATOR	DESCRIPTION
ascending	Sort in ascending alphanumeric order based on the attribute
descending	Sort in descending alphanumeric order based on the attribute

3.2.7.26 CG_Status

Type: Code_List

Used By: CG_TerminateResponse, CG_StatusResponse, CG_CancelResponse, CG_PresentResponse, CG_BrokeredAccessResponse

CG_Status is a code list for representing the current status of a resource or request. The valid values for this type are the following:

success: the request has been processed without error.

successResultsAvailable: the request has been processed without error and outputs of the processing can be retrieved.

processingNormal: the requested operations have begun but are not completed. No errors have been identified.

processingQueued: the requested operations have begun but are not completed. No errors have been identified. The processing has been temporally suspended and will resume when other processing has been completed.

processingPausedOrSuspended: the requested operations have begun but are not completed. No errors have been identified. The processing has been temporally suspended and will resume when triggered by an external event.

failure: the request could not be completed due to errors being encountered. On a best effort basis the server has returned to the state prior to the request.

failureAccessDenied : the request could not be completed because the privileges of the client did not permit the operation. On a best effort basis the server has returned to the state prior to the request.

3.2.7.27 CG_StatusUpdateType

Type: Code List

Used By: CG_BrokeredAccessRequest

This parameter defines how the user requesting the order desires to be kept informed about the order processing.

manual: The user performs the status request using the Catalog Interface

automatic: The OHS filling the order provides status updates for the user via email

3.2.7.28 CG_UserInformation

Type: Data Structure

Used By: CG_BrokeredAccessRequest

This parameter type is a data structure used to provide information about the user.

userName: (type = Character String) – name of the user

userAddress: (type = CharacterString) – billing, home or delivery address of user

phoneNumber: (type = CharacterString) – home or office phone number for user

faxNumber: (type = CharacterString) – home or office fax number for user

emailAddress: (type = CharacterString) – e-mail address for the user

NetAddress: (type = CharacterString) – Address of the users' primary computer.

PaymentMethod: (type = CG_PaymentMethod) – defines the payment method

3.2.7.29 TupleType

Type: MetaClass

Used By: CG_PresentationDescription

A set of AttributeName - AttributeType pairs. A structural metadata entity for controlling the instances of the class Tuple.

3.3 Fine-Grain Structural Model

3.3.1 Overview

This section define the Fine Grain section of the Catalog General Model. The Fine Grain portion is divided for purposes of explanation into 4 sections:

1) The Library and Manager interfaces – Used to place requests with the Catalog Service
2) The Responses – Used to retrieve the results of a request
3) The Datatypes – the data types used as parameters in the requests and responses operations
4) Callback – Used to notify clients of the status of a request

These elements are used together in a simple 3 step pattern to provide the Catalog Service capabilities:

1) The Library object provides a Manager object to the client.
2) This manager object provides one or more operations for a specific capability such as query or access. These operation when successfully invoked by the client, these operations return a Response object.
3) That Response object provides one or more operations allow retrieval of the results of the request, such as the results of a query.

3.3.2 Managers

The Manager segment of the Fine Grain General Model is composed of the Library interface, which acts as a Factory for the Managers, two abstract interfaces (LibraryManager and ResponseManager) which define operations common to the concrete Managers and five concrete Managers (CatalogMgr, OrderMgr, CreationMgr, UpdateMgr and DataModelMgr) each specialized to provide a specific capability. Figure 12 shows the UML describing these interfaces, relationships and their operations. Details for each interface are given below.

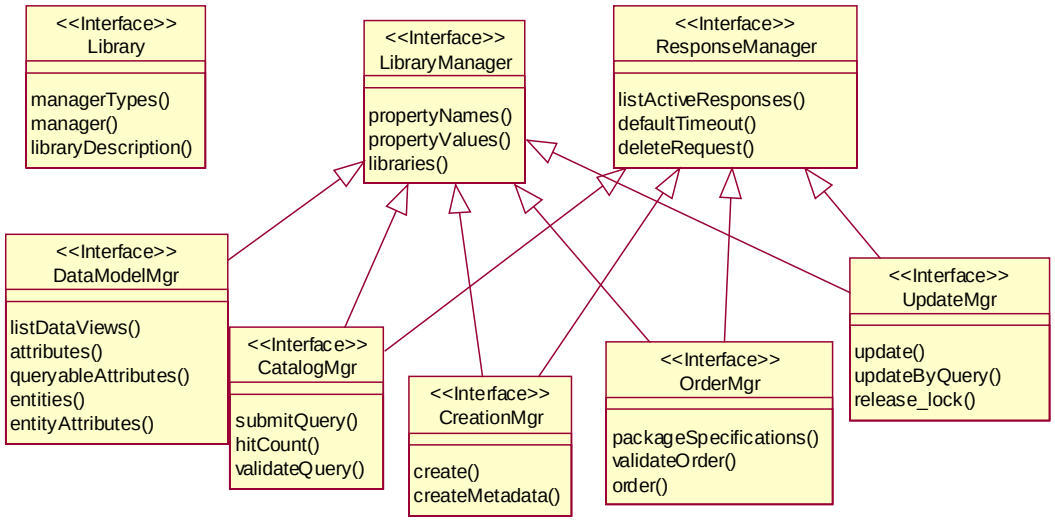


Figure 12 - The Library and Managers

3.3.2.1 Library

The Library interface is intended to serve as the starting point for any interaction with the rest of the fine-grained interfaces. All capabilities of a library system are accessed through the concrete manager objects. The Library interface is the mechanism by which a client discovers and requests access to manager objects.

3.3.2.1.1 Public Operations:

managerTypes () : ManagerTypeList

This operation allows a client to discover which managers are supported by a particular library. A ManagerTypeList structure is returned from a successful invocation of this operation.

manager (manager_type : in ManagerType, access_criteria : in OGCBasic::NameValueList) : LibraryManager

This operation is a request to be given access to a manager object. A successful invocation will return a reference to an object of type LibraryManager. The client then can use this Manager to make requests for specific Catalog services.

libraryDescription () : LibraryDescription

This operation returns descriptive information about the library. A successful invocation of this operation will return a populated LibraryDescription structure. See the Datatypes section for details on the LibraryDescription structure.

3.3.2.2 LibraryManager

The LibraryManager interface serves as the (abstract) root for all types of manager objects in the Fine Grained Model. It is abstract in the sense that a concrete LibraryManager object by itself would serve no real purpose. Its real purpose is to define certain operations that are common to all types of manager objects in the Fine Grained Model.

3.3.2.2.1 Public Operations:

propertyNames () : OGCBasic::NameList

This selector operation allows a client to obtain a list property names. A property name is the name component of a NameValue pair. The NameList returned by this operation identifies all the property names supported or known by this manager. These properties are used to describe any characteristics of a Manager.

propertyValues (desired_properties : in OGCBasic::NameList) : PropertyList

This operation allows a client to discover the properties and the current values of those properties that describe a Manager.

libraries () : LibraryList

This selector operation allows a client to determine the specific geospatial library system(s) this Manager supports.

3.3.2.3 ResponseManager

The RequestManager is an abstract interface that defines operations common to all managers that use Response objects as part of their operations.

3.3.2.3.1 Public Operations:

listActiveResponses () : ResponseList

This operation allows a client to determine what responses are being managed by this RequestManager. A successful invocation of this operation will return a ResponseList structure

defaultTimeout () : OGCBasic::RelativeTime

This operation allows a client to get the default value of the lifetime of the Requests being managed by this RequestManager. This is the length of time the Response will be maintained by the Manager before it is deleted. This is a count down timer which is reset with each invocation of a method on the object. When the count down is expired, i.e., no invocations in the defaultTimeout, the object may be "garbaged collected".

deleteRequest (response : in Response) : void

This operation allows a client to destroy a Responses and free all resources associated with that Response.

3.3.2.4 CatalogMgr

The CatalogManager Interface allows a client to submit queries to search the catalog of holdings of a geospatial library.

3.3.2.4.1 Derived from LibraryManager, ResponseManager

3.3.2.4.2 Public Operations:

submitQuery (view_name : in ViewName, query : in Query, result_attributes : in OGCBasic::NameList, sort_attributes : in SortAttributeList, properties : in PropertyList) : SubmitQueryResponse

This operation allows a client to submit a query to search a catalog of products. The client indicates the product type of interest by supplying the desired value in view_name. The client indicates the view of the catalog of interest in view_name, the query expression itself in *query*, the set of attributes to be returned in *result_attributes* and any sorting to be done in *sort_attributes*. The parameter *properties* is used to supply any implementation specific parameters. A successful invocation returns a SubmitQueryResponse object, which is used to retrieve the query results. If the property list contains the property "lock" (type = Boolean) and the lock is set to true, the products that are returned by the query are locked for update.

hitCount (view_name : in ViewName, query : in Query, properties : in PropertyList) : HitCountResponse

This operation allows a client to determine the number of results ("hits") that would be returned from a particular query. The parameters used are the same as used in the submitQuery operation.

validateQuery (view_name : in ViewName, query : in Query, result_attributes : in OGCBasic::NameList, sort_attributes : in SortAttributeList, properties : in PropertyList) : boolean

This operation allows a client to verify that a specific query is valid. the parameters used are the same as used in the submitQuery operation.

3.3.2.5 OrderMgr

The OrderMgr Interface allows a client to submit orders for data sets or products from a geospatial library. The OrderMgr provides operations to place an order (order), specify how it is to packaged and delivered (i.e., packageSpecifications), and to validate an order specification prior to submitting the order to a library (validate).

3.3.2.5.1 Derived from LibraryManager, ResponseManager

3.3.2.5.2 Public Operations:

packageSpecifications () : OGCBasic::NameList

This operation returns a NameList containing all packaging specifications known or acceptable to this OrderMgr. The details of the packageSpecifications are implementation dependent.

validateOrder (order : in OGCBasic::DG_DirectedGraph, properties : in PropertyList) : ValidationResults

This operation is used to determine if an order request for a data set or product from a geospatial library is valid. The operation returns a data structure indicating the validity of the order and information concerning details specific to the validation of the order.

order (order : in OGCBasic::DG_DirectedGraph, properties : in PropertyList) : OrderResponse

This operation is used to request delivery of one more products (i.e. place an order). The client defines the order by assembling a DG_DirectedGraph containing all necessary elements of the desired order.

3.3.2.6 CreationMgr

The CreationMgr interface allows a client to nominate a data set or product to a library(s) for inclusion in the library holdings. This interface also allows a client to nominate the metadata of a data set or product for inclusion without supplying the data set or product itself.

3.3.2.6.1 Derived from LibraryManager, ResponseManager

3.3.2.6.2 Public Operations:

create (new_product : in OGCBasic::FileLocationList, creation_metadata : in OGCBasic::DG_DirectedGraph, properties : in PropertyList) : CreateResponse

This operation allows a client to nominate a data set or product for inclusion in the holdings of a library(s). The data set or product nominated must be accompanied by the appropriate metadata. The metadata may be in the product itself in the DG_DirectGraph or a combination of the two.

createMetadata (creation_metadata : in OGCBasic::DG_DirectedGraph, view_name : in ViewName, properties : in PropertyList) : CreateMetaDataResponse

This operation allows a client to nominate the metadata of a data set or product for inclusion in a library(s) without supplying the data set or product itself. The client nominates the metadata by supplying all metadata elements in the DG_DirectedGraph creation_metadata.

3.3.2.7 UpdateMgr

The UpdateManager Interface provides the capability for a client to modify existing catalog entries.

3.3.2.7.1 Derived from LibraryManager, ResponseManager

3.3.2.7.2 Public Operations:

update (view : in ViewName, changes : in UpdateDG_DirectedGraphList, properties : in PropertyList) : UpdateResponse

This operation allows a client to modify existing catalog entries that match a specific que

updateByQuery (updated_attribute : in OGCBasic::NameValue, query : in Query, view_name : in ViewName, properties : in PropertyList) : UpdateByQueryResponse

This operation allows a client to update one or more catalog entries by supplying a query to select the entries to be changes in *query* and a NameValue pair containing the attribute to be updated in and its new value in *updated_attribute*.

release_lock (lockedProduct : in UID::Product) : void

This operation manually release a lock that has been placed on a Product. The Product reference for the locked Product is provided in the parameter lockedProduct. A product is locked through the Catalog Manager.

3.3.2.8 DataModelMgr

The DataModelManager Interface allows a client to discover and access the metadata model being used by a given Geospatial Library.

3.3.2.8.1 Derived from LibraryManager

3.3.2.8.2 Public Operations:

listDataViews (properties : in PropertyList) : ViewList

This operation exposes the hierarchy of data views types recognized by this library. See the Data types section for details of the ViewList structure.

attributes (view_name : in ViewName, properties : in PropertyList) : AttributeInformationList

This operation returns a AttributeInformationList, which describes the requested data view. The AttributeInformationList is composed of elements of type AttributeInformation. The AttributeInformationList contains both queryable and non-queryable attributes. See the data types section for the details of the AttributeInformation structure.

queryableAttributes (view_name : in ViewName, properties : in PropertyList) : AttributeInformationList

This operation returns an AttributeInformationList , which describes a specific data view. The AttributeInformationList is a sequence of elements of type AttributeInformation. The AttributeInformationList contains the subset of all attributes that are queryable. See the data types section for the details of the AttributeInformation structure.

entities (view_name : in ViewName, properties : in PropertyList) : OGCBasic::DG_DirectedGraph

This operation returns a DG_DirectedGraph, which represents a set of entities and their relationships that compose a specific data view.

entityAttributes (entity : in Entity, properties : in PropertyList) : AttributeInformationList

This operation returns a AttributeInformationList, which represents a set of attributes that describes a specific entity. The AttributeInformationList contains elements of type AttributeInformation. See the data types section for the details of the AttributeInformation structure.

3.3.3 Responses

The Response segment of the Fine Grain General Model is composed of one abstract interface (Response) and six concrete Responses. Each of these concrete responses is returned by a specific Manager operation. . Figure 13 shows the UML describing these interfaces, relationships and their operations. Details for each interface are given below.

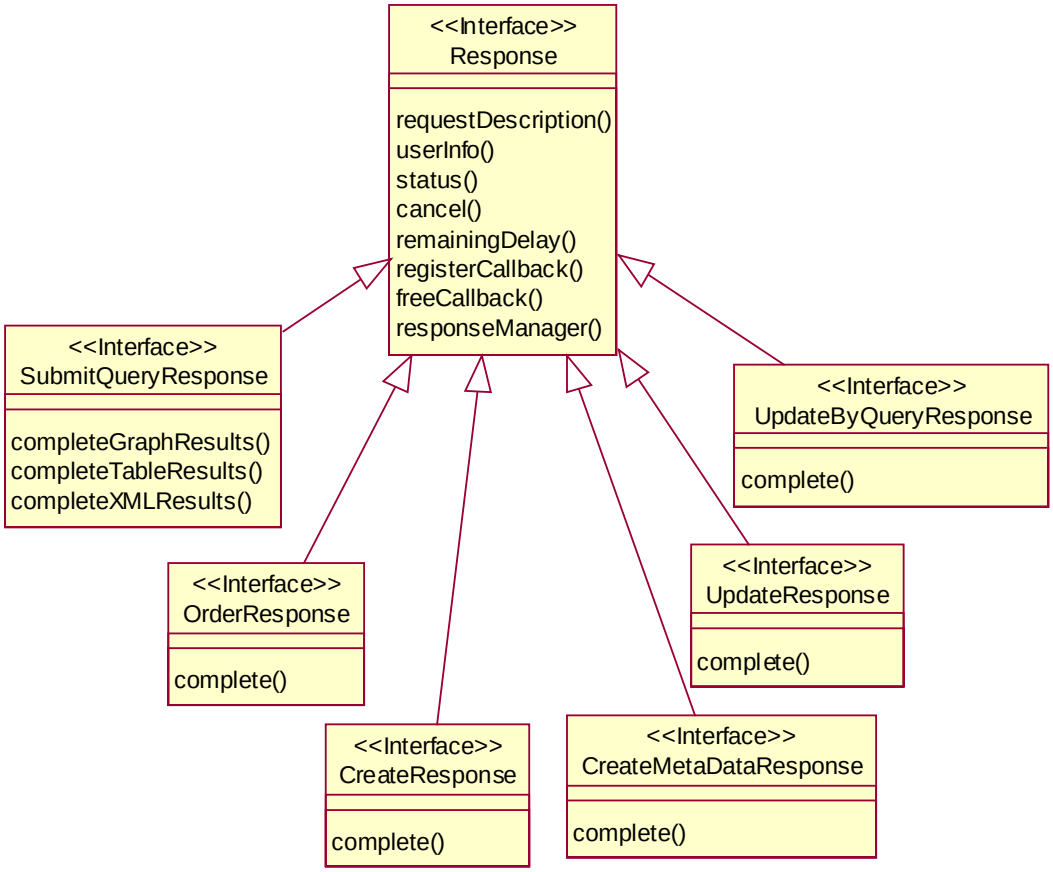


Figure 13 - Response Interfaces

3.3.3.1 Response

The Response Interface is an abstract interface that defines those operations that are common to all concrete Response objects.

3.3.3.1.1 Public Operations:

requestDescription () : OGCBasic::RequestDescription

This operation returns a populated RequestDescription structure that describes the Request that generated this Response. See the data types section for details on the RequestDescription structure.

userInfo (message : in string) : void

This operation allows a user to provide information that describes the Response. The client supplies this information, in the form of a string in a message form. A successful invocation of this operation associates the client's message with the Request.

status () : OGCBasic::Status

This operation returns the current status of the Response. This operation can be used to poll the Response to determine whether or not it has completed processing.

cancel () : void

This operation is used to terminate further processing of a Response.

remainingDelay () : OGCBasic::RelativeTime

This operation returns an estimate of the time until the Response reaches completion.

registerCallback (callback : in CB::Callback) : void

This operation allows a client to register a callback object with a Response. The purpose of a callback object is to provide a method to allow the Request to notify the client that processing of a Response has reached a terminal state.

freeCallback (callback : in CB::Callback) : void

This operation allows a client to remove a callback previously registered with a Response. The client supplies a reference to the Callback that is to be de-registered

responseManager () : ResponseManager

This operation allows a client to discover which RequestManager is managing the Request

3.3.3.2 SubmitQueryResponse

The SubmitQueryResponse Interface is used to obtain the results from submitting a query to the catalog service of a geospatial library. This Response is returned from the submitQuery operation of the CatalogMgr.

3.3.3.2.1 Derived from Response

3.3.3.2.2 Public Operations:

completeGraphResults (start_point : in unsigned long, length : in unsigned long, results : out QueryResults) : OGCBasic::State

This operation returns a set of query results expressed as a Directed Graph structure. See the data types section for details on the DG_DirectedGraph structure.

completeTableResults (start_point : in unsigned long, length : in unsigned long, results : out OGCBasic::NameValueTable) : OGCBasic::State

This operation returns a set of query results expressed as a Name Value table structure. See the data types section for details on the NameValueTable structure.

completeXMLResults (start_point : in unsigned long, length : in unsigned long, results : out string) : OGCBasic::State

This operation returns a set of query results expressed as an XML document.

3.3.3.3 OrderResponse

The OrderResponse Interface is used to return the status of the processing of an order. This Response is returned from the order operation of the OrderMgr.

3.3.3.3.1 Derived from Response

3.3.3.3.2 Public Operations:

complete () : OGCBasic::State

This operation allows a client to check if processing of the OrderResponse is complete.

3.3.3.4 CreateMetaDataResponse

This Interface is used to create new metadata entries for the catalog holdings of a geospatial library. This Response is returned from the createMetadata operation of the CreationMgr.

3.3.3.4.1 Derived from Response

3.3.3.4.2 Public Operations:

complete (new_product : out UID::Product) : OGCBasic::State

This operation allows a client to check if processing of the CreateMetaDataResponse is complete. It returns the identifier for the product just created.

3.3.3.5 CreateResponse

The CreateResponse interface is used to create new product entries in a geospatial library. This Response is returned from the create operation of the CreationMgr.

3.3.3.5.1 Derived from Response

3.3.3.5.2 Public Operations:

complete (new_product : out UID::Product) : OGCBasic::State

This operation returns a ProductList containing the references to all newly created product(s).

3.3.3.6 UpdateByQueryResponse

The UpdateByQueryRespsonse is used to complete the processing of an update of a catalog entry operation. This Response is returned from the updateByQuery operation of the UpdateMgr.

3.3.3.6.1 Derived from Response

3.3.3.6.2 Public Operations:

complete () : OGCBasic::State

Allows a client to check if processing of the of an update operation is complete.

3.3.3.7 UpdateResponse

The UpdateResponse Interface is used to complete the processing of an update operation of a catalog entry. This Response is returned from the update operation of the UpdateMgr.

3.3.3.7.1 Derived from Response

3.3.3.7.2 Public Operations:

complete () : OGCBasic::State

This operation completes the processing of a catalog update operation. It returns the status of the update operation.

3.3.4 DataTypes

This section defines the data types used by the operations of the Managers and Response interfaces. This section is broken into two subsections: Catalog Types and OGC Types. Catalog Types are types defined specifically for the use of the Catalog Service. OGC Types are used by the Catalog Service but are very general in nature i.e. could be used by other OGC services.

3.3.5 Catalog Specific DataTypes

3.3.5.1 AttributeInformation

A collection of elements that together describe an attribute used in a metadata model.

3.3.5.1.1 Public Attributes:

attribute_name : string
The name of the attribute being described

attribute_units : string
The units of measure for this attributes

description : string
A human readable description of the attribute.

sortable : boolean
A flag indicating whether this attribute is sortable.

updateable : boolean
A flag indicating whether this attribute is updateable by clients

3.3.5.2 AttributeInformationList

A sequence of AttributeInformation structures.

3.3.5.3 AttributeType

Defines the list of all possible Attribute Types

3.3.5.3.1 Public Attributes:

text

integer

floating_point

ogcbasic_coordinate

ogcbasic_polygon

ogcbasic_abs_time

ogcbasic_rectangle

ogcbasic_image

ogcbasic_height

ogcbasic_elevation

ogcbasic_distance

ogcbasic_percentage

ogcbasic_ration

ogcbasic_angle

ogcbasic_file_size

ogcbasic_file_location

ogcbasic_count

ogcbasic_weight

ogcbasic_date

ogcbasic_linestring

ogcbasic_data_rate

ogcbasic_bin_data

boolean_data

ogcbasic_duration

3.3.5.4 Change

Indicate which node of a DG_DirectGraph is to be changed and what type of change is to be performed.

3.3.5.5 ChangeList

A sequence of Change structures

3.3.5.6 ChangeType

Indicates the type of change to an attribute value being requested.

3.3.5.6.1 Public Attributes:

add_change :
Indicates a change to add a new node to a DG_DirectedGraph

update_change :
Indicates a change that will update an existing Node in a DG_DirectedGraph

delete_change :
Indicates a change that will delete a Node from a DG_DirectedGraph

3.3.5.7 DateRange

Defines a range of dates.

3.3.5.8 Domain

Defines a container to hold a domain value.

3.3.5.8.1 Public Attributes:

t : unsigned long

bv : boolean

3.3.5.9 DomainType

This enumeration defines the set of all possible data types expected to be used in metadata model. It is used by the DataModelMgr to describe an attribute in a metadata model.

3.3.5.9.1 Public Attributes:

date_value :

text_value :

integer_ value:

floating_point_ value:

list :

ordered_list :

integer_range :

floating_point_range :

geographic :

integer_set :

floating_point_set :

geographic_set :

binary_data :

boolean_value :

3.3.5.10 Entity

An identifier for an entity in a metadata model.

3.3.5.11 FloatingPointRange

Defines a range of floating point numbers.

3.3.5.11.1 Public Attributes:

lower_bound : double
The lower limit of the range.

upper_bound : double
The upper limit of the range.

3.3.5.12 FloatingPointRangeList

Defines a set of floating point ranges.

3.3.5.13 IntegerRange

Defines a range of integers.

3.3.5.13.1 Public Attributes:

lower_bound : long
The lower limit of the range

upper_bound : long
The upper limit of the range.

3.3.5.14 IntegerRangeList

Defines a set of integer ranges.

3.3.5.15 LibraryDescription

A human readable description of a Library.

3.3.5.15.1 Public Attributes:

library_name : string
An identifier for this instance of a Library.

library_description : string
A human readable description of this Library. This may contain information such as a description of its holdings and ordering or pricing schemes.

library_version_number : string
A field that indicates the version of the Library system software i.e. N.N.N

3.3.5.16 LibraryDescriptionList

A sequence of LibraryDescriptions.

3.3.5.17 LibraryList

A sequence of Library identifiers.

3.3.5.18 ManagerType

An identifier for a type of Manager. The current valid values are "CatalogMgr" , "OrderMgr" , "DataModelMgr", "CreationMgr" and "UpdateMgr"

3.3.5.19 ManagerTypeList

A sequence of ManagerTypes

3.3.5.20 Polarity

Indicates the direction of a sort.

3.3.5.20.1 Public Attributes:

ascending :

descending :

3.3.5.21 PropertyList

A list of properties and their values.

3.3.5.22 Query

A string that contains a query expression.

3.3.5.23 QueryResults

A structure that is used as one of the three means to encode a set of query results, the others being NameValueTable and XML. See the operations of the SubmitQueryResposnse interface.

3.3.5.24 RequirementMode

Defines a flag to indicate whether the attribute is required to be present in every catalog entry.

3.3.5.24.1 Public Attributes:

mandatory :

optional :

3.3.5.25 ResponseList

A sequence of Response identifiers.

3.3.5.26 SortAttribute

Indicates the attribute to be sorted upon and its direction.

3.3.5.27 SortAttributeList

3.3.5.28 UpdateDG_DirectedGraph

A structure that defines changes to another DG_DirectedGraph. It includes the new values (data) and how these changes are to be applied to the other DG_DirectedGraph (changes)

3.3.5.29 UpdateDG_DirectedGraphList

A sequence of UpdateDG_DirectedGraph structures.

3.3.5.30 ValidationResults

This type is returned to indicate if a requested operation is valid. It is used in the CatalogMgr::validateQuery and OrderMgr::validateOrder operations.

The field valid

3.3.5.30.1 Public Attributes:

valid : boolean
: If TRUE requested operation is valid. If FALSE requested operation is not valid.

warning : boolean
: If TRUEwarning field contains a description of a warning condition associated with the validity of the requested operation (i.e. Valid but ...). If FALSE no warning is given.

details : string
: The text describing the warning.

3.3.5.31 ValidationResultsList

A sequence of ValidationResult structures.

3.3.5.32 View

This structure is used to define the relationship between views and other views (sub-views)

3.3.5.33 ViewList

A sequence of Views

3.3.5.34 ViewName

An identifier for a view. A view is used to denote a specific set of attributes which may be used together in queries.

3.3.5.35 ViewNameList

A sequence of ViewName

3.3.6 Callbacks

Callbacks are an optional portion of the Fine Grain General Model that allow clients to monitor the status of their requests without either blocking or polling for status. This is done by the client implementing the Callback interface and supplying a pointer to this interface in the registerCallback operation of the Response interface. See the Response interface for the details of this operation.

3.3.6.1 Callback

The Callback interface is implemented by clients wishing to be notified of changes in state of their requests.

3.3.6.1.1 Public Operations:

notify (description : in OGCBasic::RequestDescription) : void

This operation is invoked by a server to notify the client which owns this Callback that the state of a request has changed. A description of the Request that has changed is provided in the RequestDescription parameter.

release () : void

This operation is invoked by the server to indicate to the client that the callback resources may be released.

3.4 Coarse-Grain Dynamic Model

The Coarse-Grain Catalog Interface defines a stateful server (a stateless interface will be added in future versions of the Implementation Specification). This section defines the states of the server and the allowed transitions between the states. All other state transitions are disallowed and are consider errors if exhibited by a server.

3.4.1 UML State Diagram Notation

The state diagrams in the following sections use the UML notation[2]. Figure 14 provides a summary of the UML notation used in the following sections. Both the Sequential Composite State and the Concurrent Composite State types are used. In a Sequential Composite State only one state in the composite is active at any given time. UML defines that when a transition enters Concurrent Composite State all of the concurrent states are active, although some of the concurrent states may only be in the Initial State.

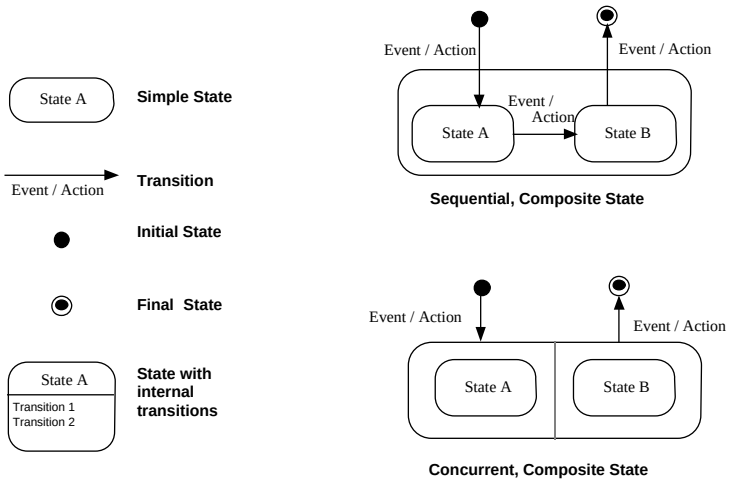


Figure 14 - UML State Diagram Notation

3.4.2 Catalog Server State Machine

The top level state diagram for the Coarse Grained Catalog Interface is shown in Figure 15. A session can only begin with an initialization that leaves the Server in the "Session Ready" state. While in the Session Ready state, if any request (except a terminate request) is received, the Server transitions to the concurrent, composite state containing four substates: Discovery, Access, Management, and Utility. Details of the four substates is provided in subsequent sections. When a termination is received, the session will transition from any current state to the end state. The state machine allows for the server to session to end after a designated time, i.e., timeout. When a session times-out, the server closes the session without notification to the client. The server should be prepared to respond to client request for a session which has timed out by returning the paired response containing a diagnostic indicating that the session does not exist.

---

[2] "The Unified Modeling Language Reference Manual", J. Rumbaugh, I. Jacobson, G. Booch, Addison-Wesley, 1999.

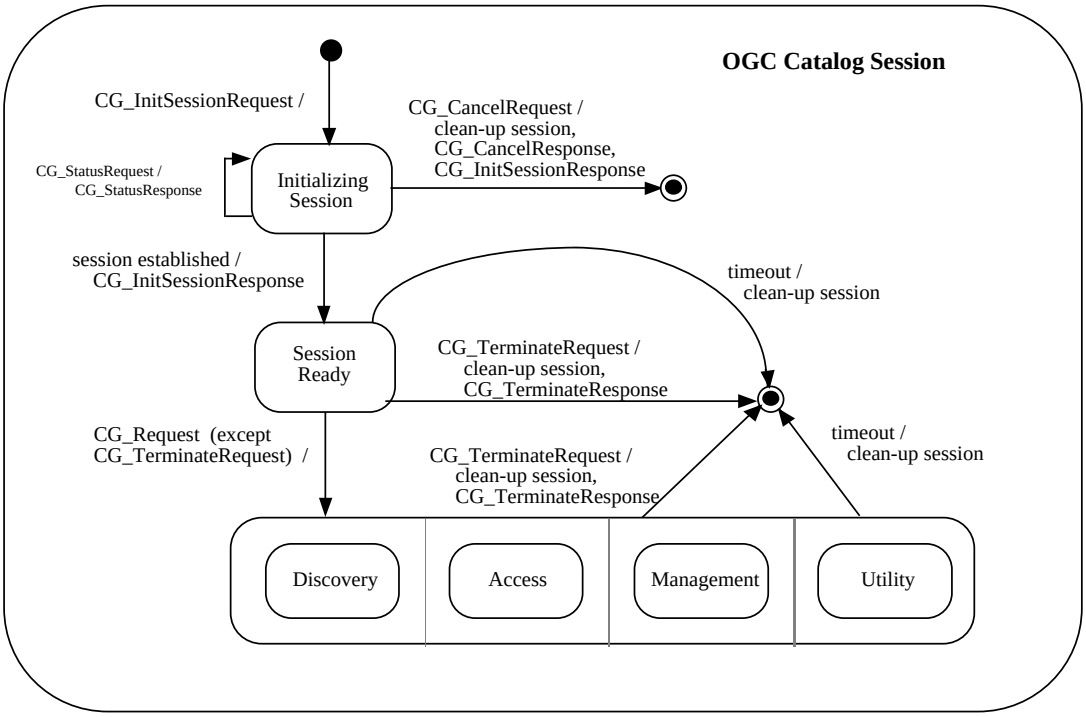


Figure 15 - Catalog Session State Diagram

3.4.3 Discovery State

Two views of the Discovery State diagram are provided: Figure 16 shows an abbreviated version, Figure 17 shows the complete Discovery state diagram. The abbreviated version is only provided to assist the reader in understanding the complete diagram. When a CG_QueryRequest message is received by the Server, the state transitions to one of three states depending upon the value of the CG_ResultType in the CG_QueryRequest. For CG_ResultType equals Validate, the transition is to the Type 1 state. A transition out of the Type 1 state occurs when the CG_QueryResponse is ready, which happens relatively quickly because the CG_QueryResponse contains little information. A CG_QueryResponse is returned for Type 1 prior to the server completing the query, therefore there is a Processing Query state. Transitions out of the Types 2& 3 and Type 4 states occur after the query is complete. Type 4 has the additional constraint that the records be formatted before transitioning to the Wait with Result Set state.

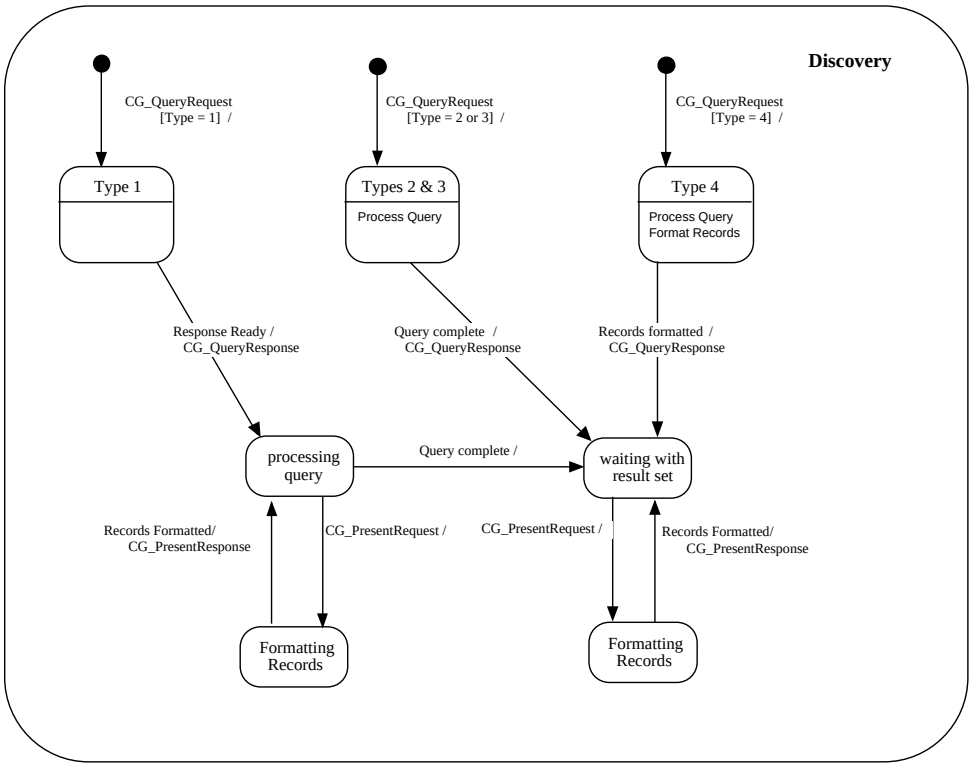


Figure 16 - Discovery State Diagram (without Status and Cancel)

Table 9 - CG_ResultType as used in Discovery State Diagrams

1	Validate
2	Result set ID
3	Hits
4	Results

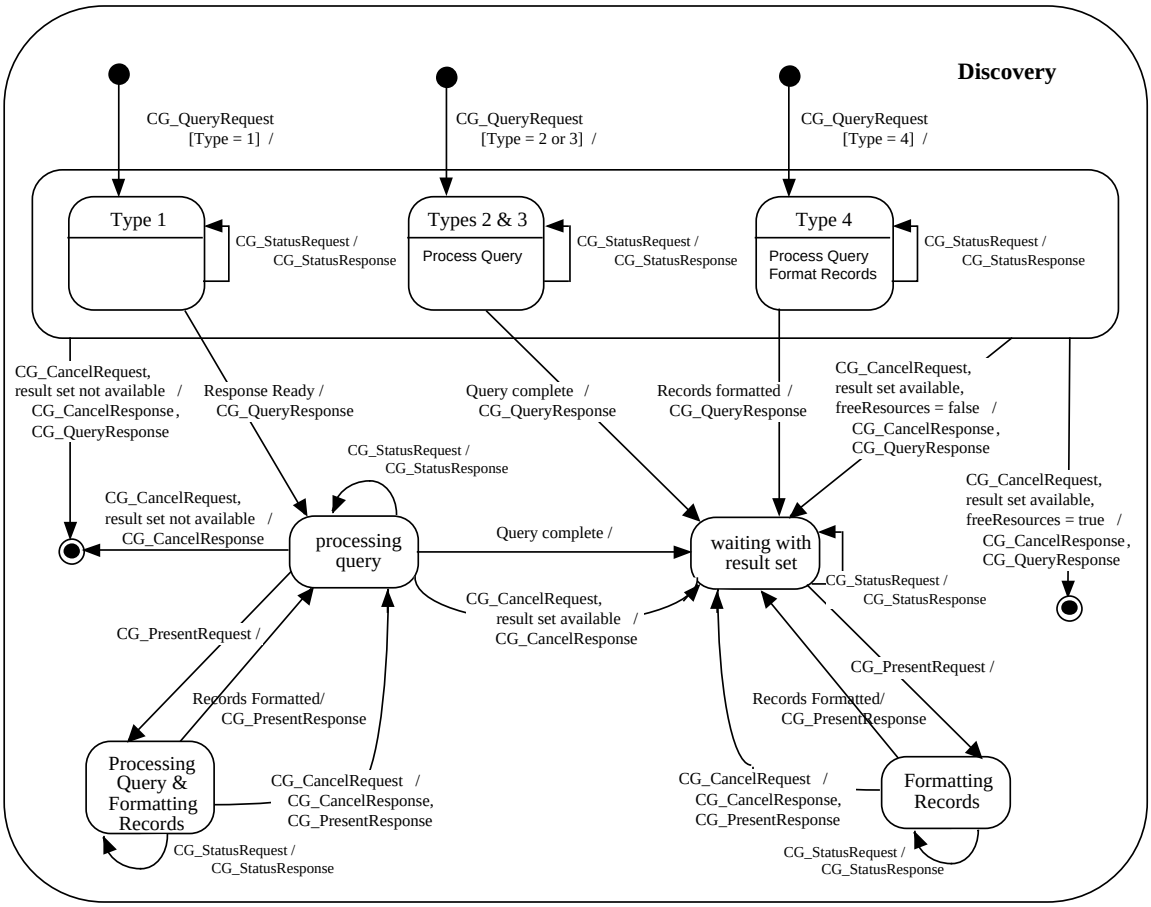


Figure 17 - Discovery State Diagram (Complete)

3.4.4 Access State Diagram

The Access State Diagram is shown in Figure 18. There is only one substate of the Access State. The Processing Request state is entered whenever a CG_BrokeredAccessRequest is received. During the Processing Request state the state of an Order may be modified based on the contents of the CG_BrokeredAccessRequest. The state of the Order is a separate state machine, see Figure 19 and Figure 20. Transitions in the Order state may occur independent of OGC Catalog Interface requests, e.g., order fulfilled is a transition which occurs without a CG_BrokeredAccessRequest. Orders may be deleted by the server. The server should be prepared to respond to client request for an order that has been deleted by returning the paired response containing a diagnostic indicating that the order does not exist.

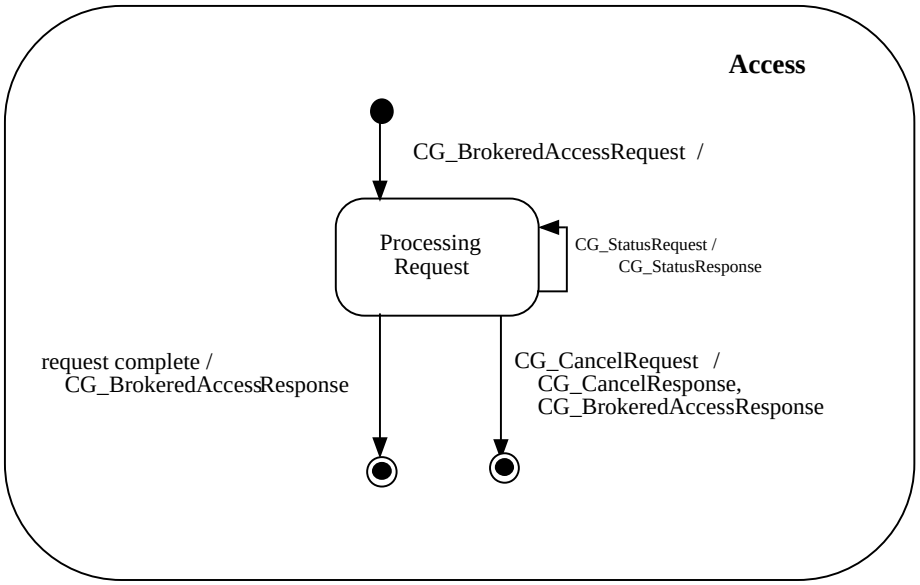


Figure 18- Access State Diagram

Table 10 - Order Request Type Diagram Codes

Value	Code in Diagrams
orderEstimate	1
orderQuoteAndSubmit	2
orderMonitor	3
orderCancel	4

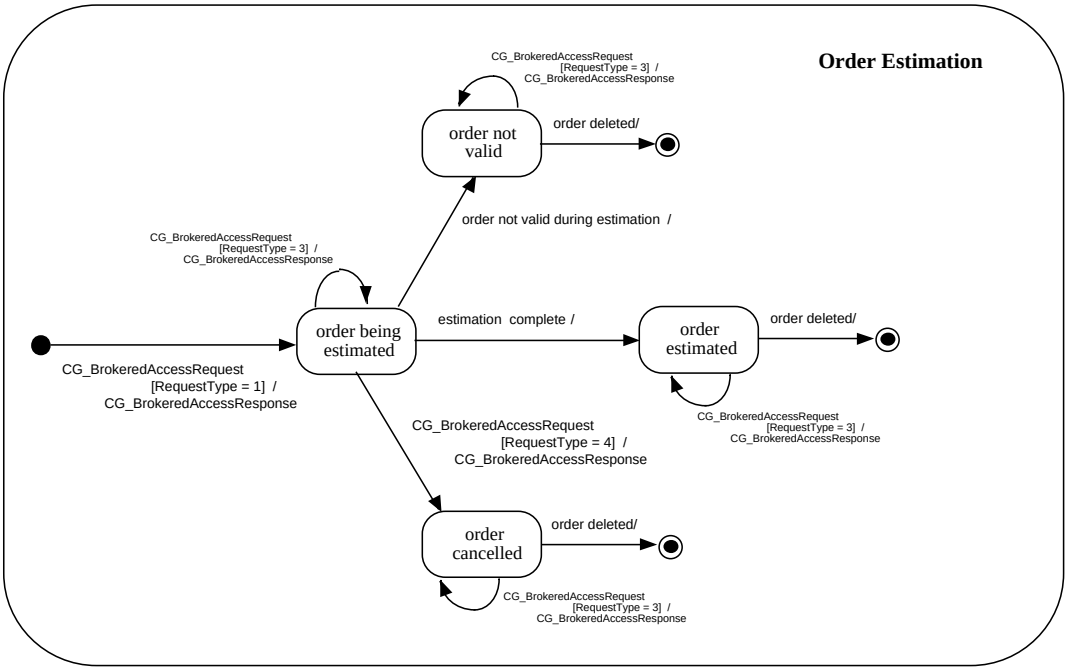


Figure 19- Order Estimation State Diagram

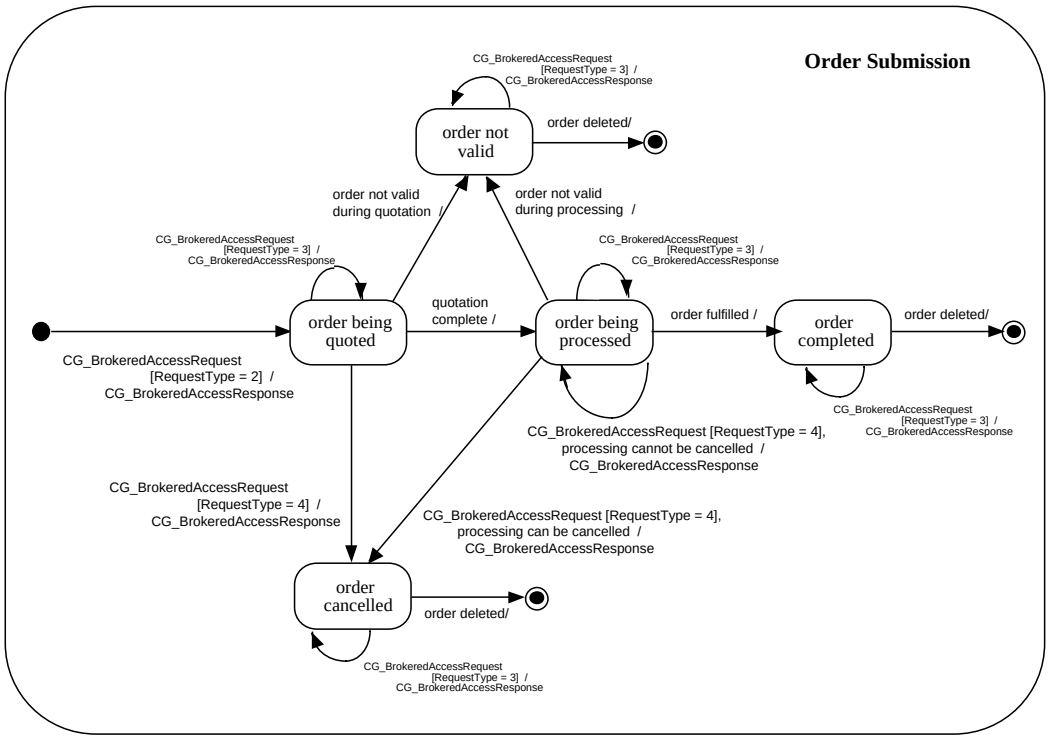


Figure 20 - Order Submit State Diagram

3.4.5 Management State

The Management State Diagram is shown in Figure 21.

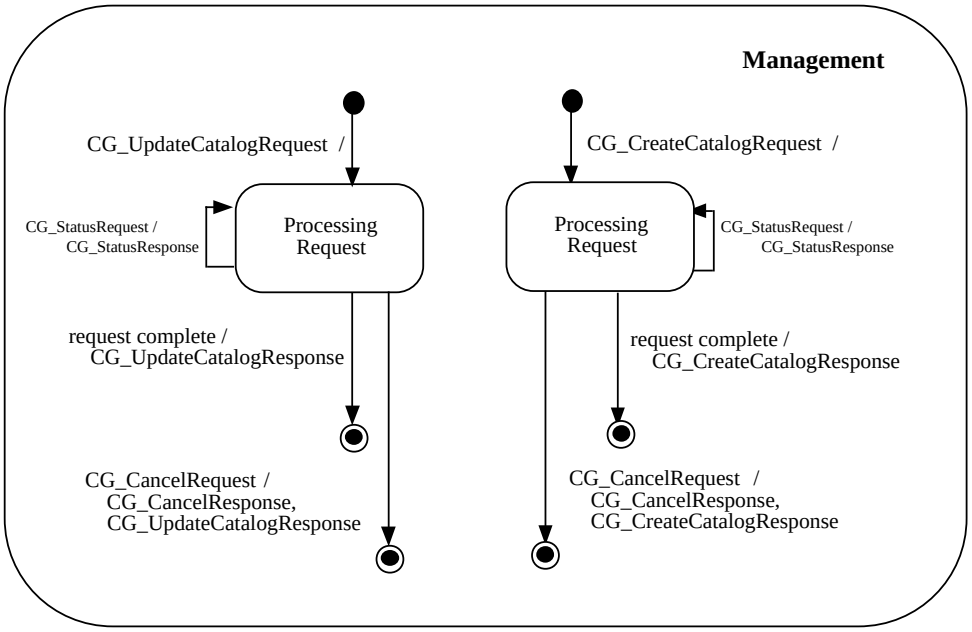


Figure 21 - Management State Diagram

3.4.6 Utility State Diagram

Requests which do not fit in the previous states are handle as part of the Utility State Machine (See Figure 22).

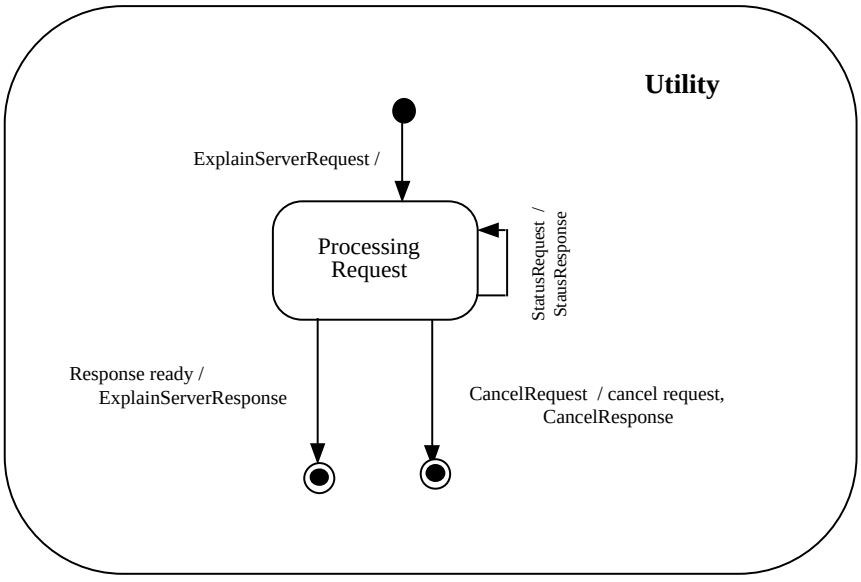


Figure 22 - Utility State Diagram

3.5 Fine-Grain Dynamic Model

This section contains sequence diagrams that show how the Fine Grain General Model is used.

3.5.1.1 Typical Query Sequence

Figure 23 - Typical Query Sequence

The typical query sequence diagram shows how a client may submit a query against a Catalog Service and retrieve the results.

1. The client retrieves a description of the Library. This could contain such information as a summary of the libraries holdings, its capabilities or its pricing model.

2. The client retrieves a list of the Manager types supported by this implementation. Using this list, the client software can determine what set of capabilities this implementation offers (i.e., discovery, access/order, creation etc). The client selects one Manager value from this list (in this case the value "CatalogMgr") and uses it in a call to the requestManager operation.

3. The client requests a specific Manager type, passing in the desired ManagerType ("CatalogMgr") and a set of name value pairs that are used as access criteria. User name and password would be the most common examples of access criteria. A successful invocation of this operation returns a reference (pointer) to a Manager of the requested type. The client can now interact directly with that Manager.

4. The client submits a query to the CatalogManager, which includes the query expression itself (comparable to a SQL where clause), a set of attributes to be returned (comparable to a SQL select clause) and any desired sorting of the result set. A successful invocation of this operation returns a reference to a SubmitQueryResponse object. The results of the query can be accessed through this Response object.

5. The client can retrieve the results of the query via the complete operation. Each invocation returns the specified sequence of "hits".

6. The client calls complete to retrieve as many hits as needed or desired.

7. The client deletes the SubmitQueryRequest when its no longer has need of it.

3.5.1.2 Minimal Query Sequence

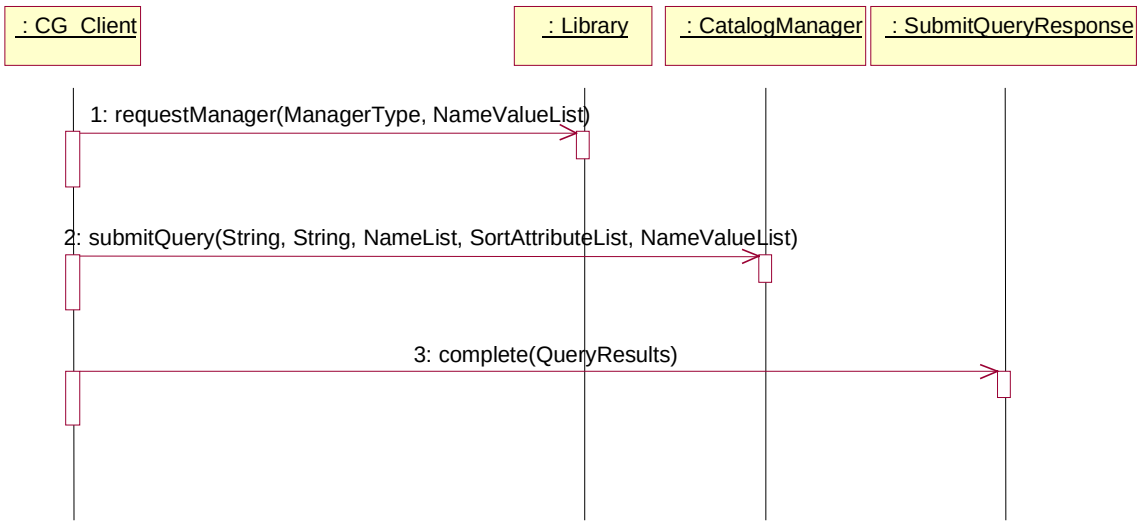


Figure 24 - Minimal Query Sequence

This is the minimum set of operations required to perform a single query and retrieve some results.

1. The client requests access to a Manager passing in the desired ManagerType (in this case a CatalogManager) and a set of name value pairs that are used as access criteria.(User name and password would be common examples of access criteria). A successful invocation of this operation returns a reference (pointer) to a Manager of the requested type. The client can now interact directly with that Manager.
2. The client submits a query to the CatalogManager, including the query expression itself (comparable to a where clause), a set of attributes to be returned (a select clause) and any desired sorting of the result set. A successful invocation of this operation returns a reference to a SubmitQueryResponse object. The results of the query can be accessed through this object.
3. The client can retrieve the results of the query via the complete operation. Each invocation returns the specified sequence of “hits”. This operation is invoked repeatedly to return as many hits as needed or desired.

3.5.1.3 Query With Callback Sequence

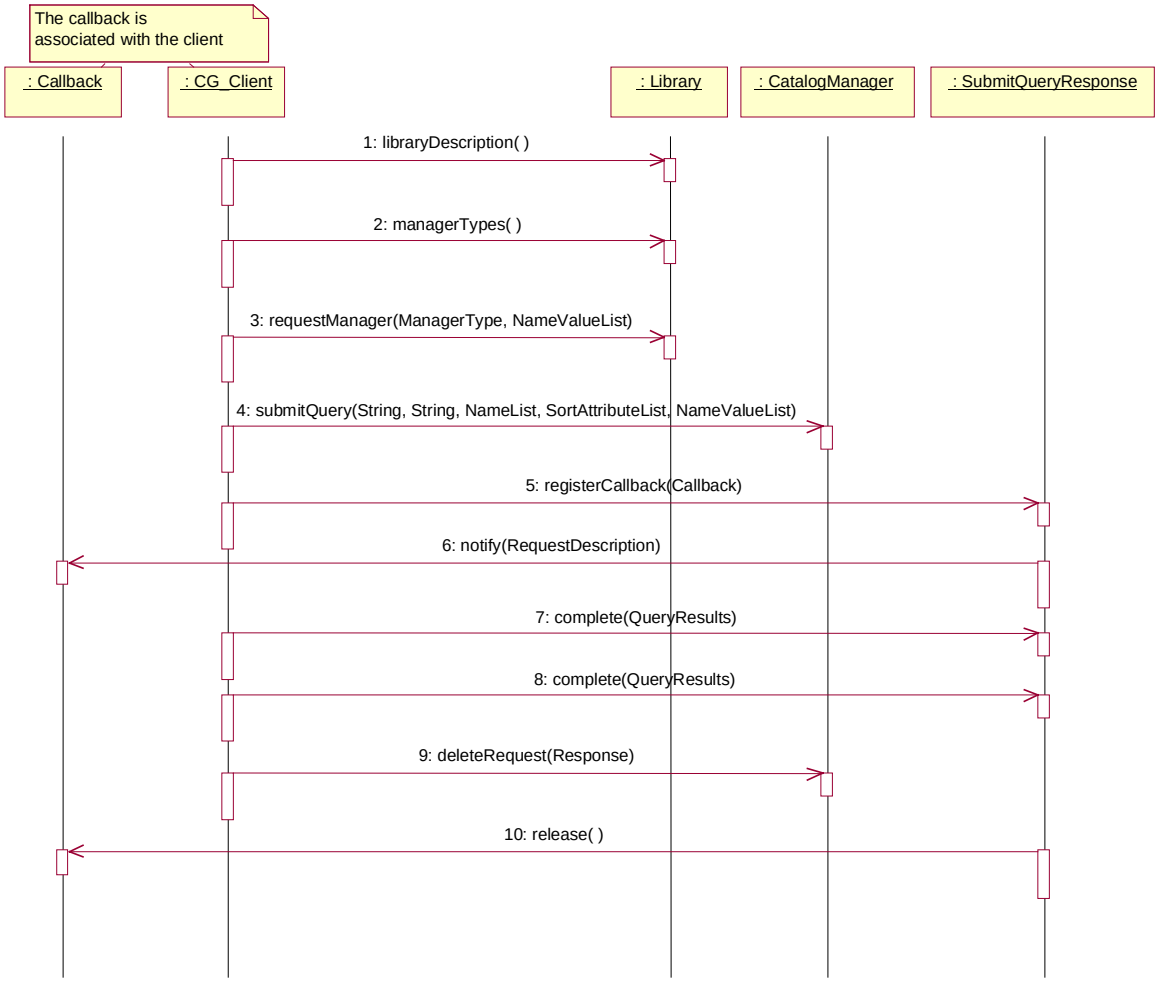


Figure 25 - Query with Callback Sequence

The query with callback sequence diagram shows how a client may use a callback object to be notified that a query has been completed.

1. The client retrieves a description of the Library. This could contain such information as a summary of the libraries holdings, its capabilities or its pricing model.
2. The client retrieves a list of the Manager types supported by this implementation. Using this list, the client software can determine what set of capabilities this implementation offers (i.e., discovery, access/order, creation etc). The client selects one Manager value from this list (in this case the value "CatalogMgr") and uses it in a call to the requestManager operation.
3. The client requests a specific Manager type, passing in the desired ManagerType ("CatalogMgr") and a set of name value pairs that are used as access criteria. User name and password would be the most common examples of access criteria. A successful invocation of this operation returns a reference (pointer) to a Manager of the requested type. The client can now interact directly with that Manager.

4. The client submits a query to the CatalogManager, which includes the query expression itself (comparable to a SQL where clause), a set of attributes to be returned (comparable to a SQL select clause) and any desired sorting of the result set. A successful invocation of this operation returns a reference to a SubmitQueryResponse object. The results of the query can be accessed through this Response object.
5. The client registers a callback object with the SubmitQueryResponse object.
6. The Catalog Service invokes the notify operation on the Callback object indicating the Response has completed processing.
7. The client can retrieve the results of the query via the complete operation. Each invocation returns the specified sequence of "hits".
8. The complete operation is called repeatedly to retrieve as many results as desired.
9. The client deletes the Response object
10. The Catalog Service invokes the release operation on the Callback since it will not be called again.

3.6 Cross-Model Interoperability

This section addresses interoperability between the Coarse-Grained and Fine-Grain models.

3.6.1 Coarse to Fine Grain Mapping

Table 11 provides a mapping between the capabilities of the Fine Grain Interfaces and their operations to the capabilities put forth in the Request-Response Message Pairs of the Coarse Grain General Model View. The mapping has been done at the Request/Response to Interface Operation level, a more detailed mapping to the Request/Response Message Class parameters will be provided with revisions of this initial specification.

Table 11 - Coarse to Fine Grain Mapping

Fine Grain Interfaces and Their Operation(s)	Coarse Grain Message Classes
Library	
managerTypes	CG_ExplainServerRequest
manager	CG_ExplainServerRequest
libraryDescription	CG_ExplainServerRequest
LibraryManager (abstract)	
propertyValues	CG_ExplainServerRequest
propertyNames	CG_ExplainServerRequest
libraries	CG_ExplainServerRequest
RESPONSEMANAGER (ABSTRACT)	
listActiveResponses	CG_StatusRequest
defaultTimeout	CG_Timeout
deleteRequest	CG_CancelRequest
CatalogManager	
submitQuery	CG_QueryRequest
hitCount	CG_QueryRequest
validateQuery	CG_QueryRequest
DataModelManager	
listDataViews	CG_ExplainCollectionRequest
attributes	CG_ExplainCollectionRequest
queryableAttributes	CG_ExplainCollectionRequest
entities	CG_AttributeCategory

Fine Grain Interfaces and Their Operation(s)	Coarse Grain Message Classes
entityAttributes	CG_AttributeCategory
OrderManager	
packageSpecifications	CG_PackageSpecification
validateOrder	CG_BrokeredAccessResponse
order	CG_BrokeredAccessRequest
UpdateManager	
update	CG_UpdateCatalogRequest
releaseLock	N/A
updateByQuery	
CreationManager	
create	CG_CreateCatalogRequest
createMetadata	CG_CreateMetadataRequest
Callback	
notify	N/A
release	N/A
RESPONSE (ABSTRACT)	
requestDescription	N/A
userInfo	CG_UserInformation
status	CG_StatusRequest
remainingDelay	N/A
cancel	CG_CancelRequest
registerCallback	N/A
freeCallback	N/A
responseManager	N/A
SubmitQueryResponse	
One of the three complete operations depending on what the type of data is returned in the CG_QueryResponse message parameter “retrievedData””	CG_QueryResponse

Fine Grain Interfaces and Their Operation(s)	Coarse Grain Message Classes
OrderResponse	
complete	CG_BrokeredAccessResponse
SubmitQueryResponse	
One of the three complete operations: completeGraphResults; completeTableResults; completeXMLResults.	CG_QueryResponse CG_PresentResponse
UpdateResponse	
complete	CG_UpdateCatalogResponse
UpdateByQueryResponse	
complete	CG_UpdateCatalogResponse
CreateResponse	
complete	CG_CreateCatalogResponse
CreateMetaDataResponse	
complete	CG_CreateMetadataResponse

3.6.2 DCP Bridge Dynamics

To achieve cross-DCP interoperability, a Bridge between Coarse and Fine Grain implementation is needed. It's basic function is to translate the receipt of a message (the basic element of the Coarse Grain General Model) into one or more object invocations (the basic element of the Fine Grain General Model) and then inversely translate one or more object invocations into the appropriate message. The Bridge is composed of a server from one profile, a translator, and a client for another profile. A Coarse-to-Fine translator exposes only Coarse Grain interfaces and maps these to the appropriate Fine Grain operations. Likewise a Fine-to-Coarse Translator exposes only Fine Grain interfaces and maps these to the appropriate message transfers. Figure 26shows a sequence diagram that traces a query request from the Coarse side to the Fine and its return.

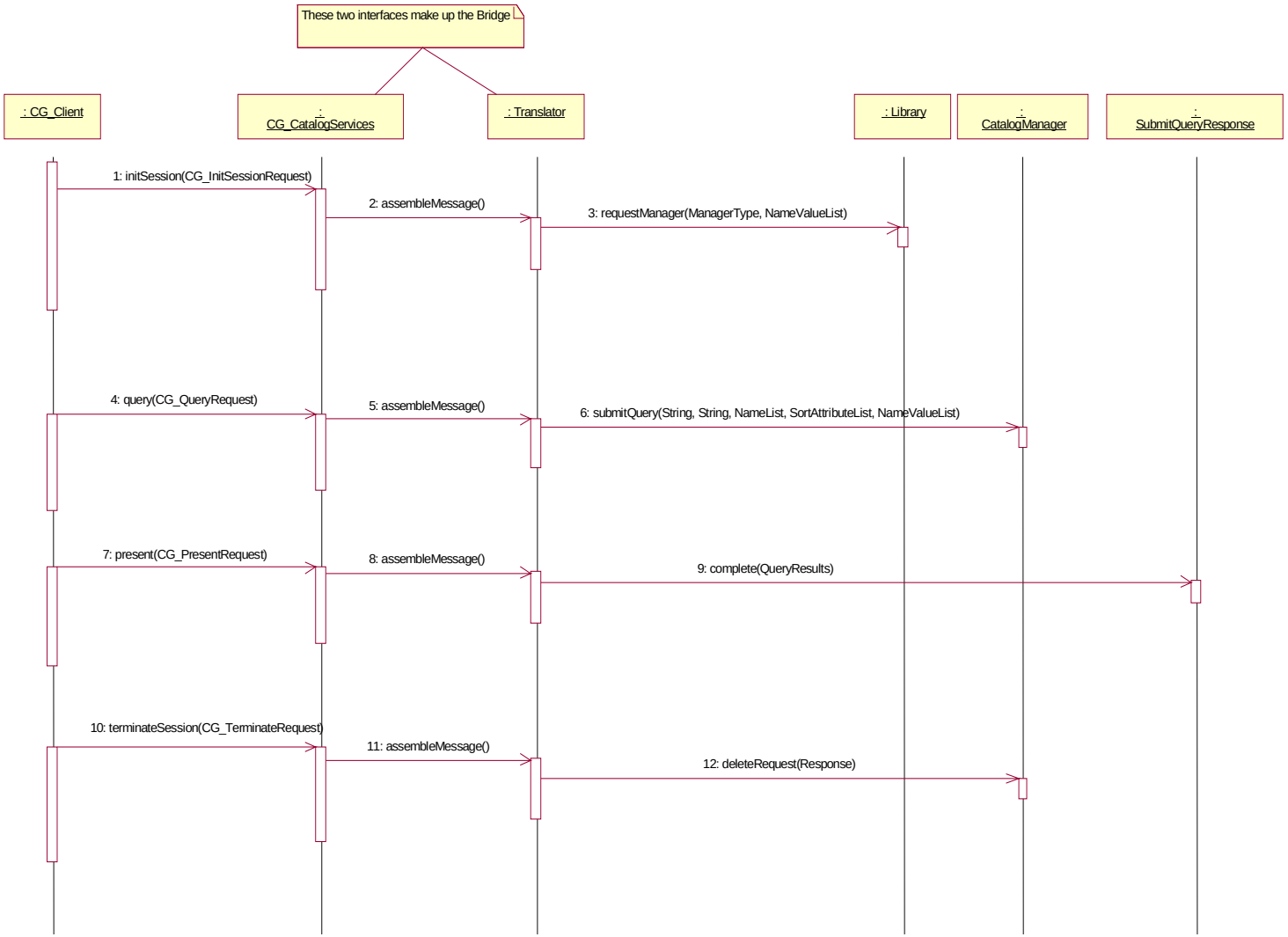


Figure 26 - Coarse to Fine Bridge

4. OGC_Common Catalog Query Language

This section defines the OGC_Common Catalog Query Language. OGC_Common is the query language to be supported by all OGC Catalog Interfaces in order to support search interoperability.

4.1 Assumptions during the development of OGC_Common Query Language:

- The query will have a syntax similar to the SQL "Where Clause"
- The expressiveness of the query will not require extensions to various current query systems used in geospatial catalog queries other than the implementation of some geo operators.
- The query language should be extensible
- OGC_Common should support both tight and loose queries. A tight query is defined where if a catalog doesn't support an attribute/column specified in the query, no entity/row can match the query and the null set is returned. In a loose query, if an attribute is undefined, it is assumed to match

4.2 BNF definition of OGC_Common Query Language

```
<SQL terminal character> ::=
    <SQL language character>

<SQL language character> ::=
      <simple Latin letter>
    | <digit>
    | <SQL special character>

<simple Latin letter> ::=
      <simple Latin upper case letter>
    | <simple Latin lower case letter>

<simple Latin upper case letter> ::=
     A | B | C | D | E | F | G | H | I | J | K | L | M | N | O
    | P | Q | R | S | T | U | V | W | X | Y | Z

<simple Latin lower case letter> ::=
          a | b | c | d | e | f | g | h | i | j | k | l | m | n | o
    | p | q | r | s | t | u | v | w | x | y | z

<digit> ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<SQL special character> ::=
      <space>
    | <double quote>
    | <percent>
    | <ampersand>
    | <quote>
    | <left paren>
    | <right paren>
    | <asterisk>
    | <plus sign>
    | <comma>
    | <minus sign>
    | <period>
    | <solidus>
```

```
    | <colon>
    | <semicolon>
    | <less than operator>
    | <equals operator>
    | <greater than operator>
    | <question mark>
    | <left bracket>
    | <right bracket>
    | <circumflex>
    | <underscore>
    | <vertical bar>
    | <left brace>
    | <right brace>

<space> ::=  /*space character in character set in use In ASCII it would be
          40*/

<double quote> ::= "

<percent> ::= %

<ampersand> ::= &

<quote> ::= ’

<left paren> ::= (

<right paren> ::= )

<asterisk> ::= *

<plus sign> ::= +

<comma> ::= ,

<minus sign> ::= -

<period> ::= .

<solidus> ::= /

<colon> ::= :

<semicolon> ::= ;

<less than operator> ::= <

<equals operator> ::= =

<greater than operator> ::= >

<question mark> ::= ?

<left bracket> ::= [

<right bracket> ::= ]

<circumflex> ::= ^

<underscore> ::= _

<vertical bar> ::= |
```

```
<left brace> ::={

<right brace> ::=}

<separator> ::= { <comment> | <space> | <newline> }...


/* The next section of the BNF defines the tokens available to the language.
          I have deleted the concepts of bit string, hex string and national
          character string literal, since those types do not have
          equivalents in GIAS or CIP/GEO. Also a significant number of the
          keywords have been  removed with Keywords have been added to
          support the geo literals. */

<token> ::=
      <nondelimiter token>
    | <delimiter token>

<nondelimiter token> ::=
      <regular identifier>
    | <key word>
    | <unsigned numeric literal>

<regular identifier> ::= <identifier body>

<identifier body> ::=
    <identifier start> [ { <underscore> | <identifier part> }... ]

<identifier start> ::= <simple latin letter>

<identifier part> ::=
      <identifier start>
    | <digit>

<key word> ::=
      <reserved word>

<reserved word> ::=
    AND | POINT  |LINESTRING
    |POLYGON |MULTIPOINT  |MULTILINESTRING  |MULTIPOLYGON
    | EMPTY  | DATE | TIME | TIMESTAMP|FALSE| TRUE| UNKNOWN |LIKE  | MINUTE
          | MONTH
    |  NOT | NULL |

<unsigned numeric literal>::=
      <exact numeric literal>
    | <approximate numeric literal>

<exact numeric literal> ::=
      <unsigned integer> [ <period> [ <unsigned integer> ] ]
    | <period> <unsigned integer>

<unsigned integer> ::= <digit>...

<approximate numeric literal> ::= <mantissa> E <exponent>

<mantissa> ::= <exact numeric literal>

<exponent> ::= <signed integer>

<signed integer> ::= [ <sign> ] <unsigned integer>

<sign> ::= <plus sign> | <minus sign>
```

```
< character string literal> ::=
   <quote> [ <character representation>... ] <quote>

 <character representation> ::=
      <nonquote character>
    | <quote symbol>

<quote symbol> ::= <quote><quote>


/*End of non delimiter tokens*/

/* I have limited the delimiter tokens by eliminating, interval strings and
          delimited identifiers BNF and simplifying the legal character set
          to the characters to a single set so no identification of
          character set would be needed decision. */



<delimiter token> ::=
      <character string literal>
    | <SQL special character>
    | <not equals operator>
    | <greater than or equals operator>
    | <less than or equals operator>
    | <concatenation operator>
    | <double greater than operator>
    | <right arrow>
    | <left bracket>
    | <right bracket>

< character string literal> ::=
<quote> [ <character representation>... ] <quote>

<character representation> ::=
      <nonquote character>
    | <quote symbol>

<quote symbol> ::= <quote><quote>

<not equals operator> ::= <>

<greater than or equals operator> ::= >=

<less than or equals operator> ::= <=


/*The following section is intended to give context for identifier and
          namespaces.  It assumes that the default namespace is specified in
          the query request and does not allow any overrides of the namepace
          */


< identifier> ::=
    < identifier start [ { <underscore> | <identifier part> }... ]

< identifier start> ::= <simple Latin letter>

< identifier part> ::=
      <simple Latin letter>
    | <digit>.
```

```
<attribute name> ::= <simple attribute name>
     | <compound attribute name>

<simple attribute name>::=<identifier>

<compound attribute name>::= < identifier><period> [{<identifier><period>}…]
       <simple attribute name>



/*The rest of the BNF is the real BNF for the query capabilities.*/

<search condition> ::=<boolean value expression>

<boolean value expression> ::=
      <boolean term>
    | <boolean value expression> OR <boolean term>

<boolean term> ::=
      <boolean factor>
    | <boolean term> AND <boolean factor>

<boolean factor> ::=
    [ NOT ] <boolean primary>

<boolean primary> ::=
      <predicate>
    | < routine invocation>

<predicate> ::=
      <comparison predicate>
    | <text predicate>
    | < null predicate>


<comparison predicate> ::= <attribute name> <comp op> <literal>


<text predicate> ::= <attribute name> [ NOT ] LIKE <character pattern>


<null predicate> ::= <attribute name> IS [ NOT ] NULL


<character pattern> ::= <character string literal> /* In a character pattern
          the character percent is used as a wildcard to represent an
          arbitrary string. This allows LIKE to implement the effect of many
          characters matching operations, such as: contains, begins with,
          ends with, not contains, not begins with, not ends with, and so
          forth. For example:
    attribute like '%contains_this%'
     attribute like 'begins_with_this%'
     attribute like '%ends_with_this'
    attribute like ‘d_ve’ will match ‘dave’ or “dove””
     attribute not like '%will_not_contain_this%'
     attribute not like 'will_not_begin_with_this%'
     attribute not like '%will_not_end_with_this’*/


<comp op> ::=
      <equals operator>
    | <not equals operator>
    | <less than operator>
```

```
    | <greater than operator>
    | <less than or equals operator>
    | <greater than or equals operator>

<literal> ::=
      <signed numeric literal>
     | <general literal>

<signed numeric literal> ::=
    [<sign> ] <unsigned numeric literal>

<general literal> ::=
      <character string literal>
    | <datetime literal>
    | <boolean literal>
    | <geography literal

<boolean literal> ::=
      TRUE
    | FALSE
    | UNKNOWN

<routine invocation> ::=
     | <geoop name>< georoutine argument list>
     | <relgeoop name><relgeoop argument list>
     | <routine name> <argument list>


<routine name> ::= < attribute name>

<geoop name> ::= EQUAL |DISJOINT |INTERSECT |TOUCH |CROSS |WITHIN |CONTAINS
          |OVERLAP |RELATE

<relgeoop name> ::= DWITHIN |BEYOND
<argument list> ::=
      <left paren> [ <positional arguments>]  <right paren>

<positional arguments> ::=
    <argument> [ { <comma> <argument> }... ]

<argument> ::= <literal> | <attribute name>

<georoutine argument list> ::=
    <left paren> <attribute name> <comma> <geometry literal>  <right paren>

<relgeoop argument list> ::=
   <left paren> <attribute name> <comma> <geometry literal> <comma>
          <tolerance>
   <right paren>

<tolerance> ::=
    <unsigned numeric literal> <comma> <distance units>

<distance units> ::= = “feet” | “meters” | “statute miles” | “nautical miles”
          | “kilometers”

/*this set of units is just an example. The real list of distance unit must
          be developed*?

<geometry literal> :=
     <Point Tagged Text>
…. | <LineString Tagged Text>
     | <Polygon Tagged Text>
```

```
     | <MultiPoint Tagged Text>
     | <MultiLineString Tagged Text>
     | <MultiPolygon Tagged Text>
     | <GeometryCollection Tagged Text>
     | <Envelope Tagged Text>

<Point Tagged Text> :=
    POINT <Point Text>

<LineString Tagged Text> :=
    LINESTRING <LineString Text>

<Polygon Tagged Text> :=
    POLYGON <Polygon Text>

<MultiPoint Tagged Text> :=
    MULTIPOINT <Multipoint Text>

<MultiLineString Tagged Text> :=
    MULTILINESTRING <MultiLineString Text>

<MultiPolygon Tagged Text> :=
    MULTIPOLYGON <MultiPolygon Text>

<GeometryCollection Tagged Text> :=
    GEOMETRYCOLLECTION <GeometryCollection Text>

<Point Text> := EMPTY | <left paren> <Point> <right paren>

<Point> := <x> <space><<y>

<x> := numeric literal

<y> := numeric literal

<LineString Text> := EMPTY
     | <left paren> <Point >  {<comma> <Point > }… <right paren>

<Polygon Text> := EMPTY
     | <left paren> <LineString Text >  {<comma> < LineString Text > }…<right
          paren>

<Multipoint Text> := EMPTY
     | <left paren> <Point Text >  {<comma> <Point Text > }…  <right paren>

<MultiLineString Text> := EMPTY
     | <left paren> <LineString Text >  {<comma> < LineString Text > }…
          <right paren>

<MultiPolygon Text> := EMPTY
     | <left paren> < Polygon Text >  {<comma> < Polygon Text > }…  <right
          paren>

<GeometryCollection Text> := EMPTY
     | <left paren> <Geometry Tagged Text> {<comma> <Geometry Tagged Text> }…
            <right paren>
<Envelope Tagged Text> ::=
    ENVELOPE <Envelope Text>

<Envelope Text> := EMPTY
| <left paren> > <WestBoundLongitude> <comma> EastBoundLongitude> <comma>
NorthBoundLatitude <comma> <SouthBoundLatitude> < <right paren>
```

```
<WestBoundLongitude> := numeric literal

<EastBoundLongitude> := numeric literal

<NorthBoundLatitude> := numeric literal

<SouthBoundLatitude> := numeric literal
<datetime literal> ::=
<date literal>
| <time literal>
| <timestamp literal>

<date literal> ::=
    DATE <date string>
<date string> ::=
    <quote> <unquoted date string> <quote>

<unquoted date string> ::= <date value>

<date value> ::=
     <years value> <minus sign> <months value><minus sign> <days
          value>

<years value> ::= <datetime value>

<datetime value> ::= <unsigned integer>

<months value> ::= <datetime value>

<days value> ::= <datetime value>

<time literal> ::=
TIME <time string>

<time string> ::=
    <quote> <unquoted time string> <quote>
 unquoted time string> ::=
    <time value> [ <time zone interval> ]
<time value> ::=
    <hours value> <colon> <minutes value> <colon> <seconds value>

<hours value> ::= <datetime value>

<minutes value> ::= <datetime value>

<seconds value> ::=
<seconds integer value> [ <period> [ <seconds fraction> ] ]

<seconds integer value> ::= <unsigned integer>

<seconds fraction> ::= <unsigned integer>

<time zone interval> ::=
<Z>|<sign> <hours value> <colon> <minutes value> /*  Z= Coordinated
          Universal Time, signed numerics are offsets from UTC*/

<timestamp literal> ::=
```

```
TIMESTAMP <timestamp string>
<timestamp string> ::=
<quote> <unquoted SQL timestamp string> <quote>
|<quote> <unquoted ISO timestamp string> <quote>

<unquoted  SQL timestamp string> ::=
<unquoted date string> <space> <unquoted time string>

<unquoted  ISO timestamp string> ::=
<unquoted date string> <T> <unquoted time string>
```

5. CORBA Profiles

The CORBA Profile contains two profiles, the Fine-Grain that maps to the Fine-Grain General Interface Model and a Coarse-Grained CORBA Profile that maps to the Coarse-Grained Discovery Interfaces of the General Model.

5.1 Fine-Grain CORBA Profile

This profile defines a mapping of the Fine Grained segment of the Catalog General Model into a form suitable for implementation using CORBA technology. Specifically, it provides an interpretation of the General Model into OMG Interface Definition Language (OMG-IDL). Since the mapping described here is based on the standardized UML to IDL mapping[3] the architecture of the Fine Grained CORBA Profile is identical to the Fine Grained segment of the General Model with the following exception: All General Model elements stereotyped as "DataType" where modified to have a more specific CORBA-specific stereotype (i.e. "CORBAStruct", "CORBAEnum", "CORBAUnion" or "CORBATypedef" as appropriate). Given these more specific stereotypes, the OMG-IDL was auto-generated from the General Model.

5.1.1 Sequence Diagrams

The sequence diagrams that indicate how the elements of the Fine Grained CORBA profile are used are identical to those included in the Fine Grained segment of the General Model (see Figure 23, Figure 24, and Figure 25).

5.1.2 IDL Definition

The Fine Grained CORBA profile is broken into four CORBA Modules:

1) FGCatalog module – The Primary Catalog Service interfaces and data types
2) CB module– The Callback interface
3) UID module – A unique identifier handle
4) OGCBasicTypes module – Some fundamental data types used by the Catalog Service but which are not specific to Catalog Services.

5.1.3 Module FGCatalog

```
#ifndef __FGCATALOG_DEFINED
#define __FGCATALOG_DEFINED

#include "ogcBasic.idl"
#include "cb.idl"
#include "uid.idl"

module FGCatalog {

/* Forward References */
  interface Library;
  interface Response;
```

[3] As defined in OMG Unified Modeling Language Specification (Draft), Version 1.3 alpha R5, March 1995.

```
/* The Catalog Specific Data Types */
  typedef OGCBasic::NameValueList PropertyList;

  struct ValidationResults {
    boolean valid;
    boolean warning;
    string details;
  };

  typedef sequence <ValidationResults> ValidationResultsList;

  typedef string ViewName;

  typedef sequence <ViewName> ViewNameList;

  struct View {
    ViewName view_name;
    ViewNameList sub_views;
  };

  typedef sequence <View> ViewList;

  enum DomainType {
    DATE_VALUE,
    TEXT_VALUE,
    INTEGER_VALUE,
    FLOATING_POINT_VALUE,
    LIST,
    ORDERED_LIST,
    INTEGER_RANGE,
    FLOATING_POINT_RANGE,
    GEOGRAPHIC,
    INTEGER_SET,
    FLOATING_POINT_SET,
    GEOGRAPHIC_SET,
    BINARY_DATA,
    BOOLEAN_VALUE
  };

  struct DateRange {
    OGCBasic::AbsTime earliest;
    OGCBasic::AbsTime latest;
  };

  struct IntegerRange {
    long lower_bound;
    long upper_bound;
  };

  struct FloatingPointRange {
    double lower_bound;
    double upper_bound;
  };

  typedef sequence <IntegerRange> IntegerRangeList;

  typedef sequence <FloatingPointRange> FloatingPointRangeList;
```

```
union Domain switch(DomainType) {
   case DATE_VALUE: DateRange d;
   case TEXT_VALUE: unsigned long t;
   case INTEGER_VALUE: IntegerRange iv;
   case INTEGER_SET: IntegerRangeList is;
   case FLOATING_POINT_VALUE: FloatingPointRange fv;
   case FLOATING_POINT_SET: FloatingPointRangeList fps;
   case LIST: OGCBasic::NameList l;
   case ORDERED_LIST: OGCBasic::NameList ol;
   case INTEGER_RANGE: IntegerRange ir;
   case FLOATING_POINT_RANGE: FloatingPointRange fr;
   case GEOGRAPHIC: OGCBasic::Rectangle g;
   case GEOGRAPHIC_SET: OGCBasic::RectangleList gs;
   case BINARY_DATA: OGCBasic::BinData bd;
   case BOOLEAN_VALUE: boolean bv;
};

enum AttributeType {
   TEXT,
   INTEGER,
   FLOATING_POINT,
   OGCBasic_COORDINATE,
   OGCBasic_POLYGON,
   OGCBasic_ABS_TIME,
   OGCBasic_RECTANGLE,
   OGCBasic_IMAGE,
   OGCBasic_HEIGHT,
   OGCBasic_ELEVATION,
   OGCBasic_DISTANCE,
   OGCBasic_PERCENTAGE,
   OGCBasic_RATIO,
   OGCBasic_ANGLE,
   OGCBasic_FILE_SIZE,
   OGCBasic_FILE_LOCATION,
   OGCBasic_COUNT,
   OGCBasic_WEIGHT,
   OGCBasic_DATE,
   OGCBasic_LINESTRING,
   OGCBasic_DATA_RATE,
   OGCBasic_BIN_DATA,
   BOOLEAN_DATA,
   OGCBasic_DURATION
};

enum RequirementMode {
   MANDATORY,
   OPTIONAL
};

struct AttributeInformation {
   string attribute_name;
   AttributeType attribute_type;
   Domain attribute_domain;
   string attribute_units;
   RequirementMode mode;
   string description;
```

```
      boolean sortable;
      boolean updateable;
    };

    typedef string Entity;

    typedef sequence <AttributeInformation> AttributeInformationList;

    typedef string ManagerType;

    typedef sequence <ManagerType> ManagerTypeList;

    struct LibraryDescription {
      string library_name;
      string library_description;
      string library_version_number;
    };

    typedef sequence <LibraryDescription> LibraryDescriptionList;

    typedef string Query;

    typedef OGCBasic::DG_DirectedGraphList QueryResults;

    enum Polarity {
      ASCENDING,
      DESCENDING
    };

    struct SortAttribute {
      OGCBasic::Name attribute_name;
      Polarity sort_polarity;
    };

    typedef sequence <SortAttribute> SortAttributeList;

    enum ChangeType {
      ADD_CHANGE,
      UPDATE_CHANGE,
  DELETE_CHANGE
    };

    struct Change {
      OGCBasic::NodeID changed_node;
      ChangeType change_type;
    };

    typedef sequence <Change> ChangeList;

    struct UpdateDG_DirectedGraph {
      OGCBasic::DG_DirectedGraph data;
      ChangeList changes;
    };

    typedef sequence <UpdateDG_DirectedGraph>
           UpdateDG_DirectedGraphList;
```

```
/* The Exceptions */

   struct ExceptionInfo {
     string exception_details;
   };

   exception BadAccessCriteria {
     ExceptionInfo info;
   };

   exception BadAccessValue {
     ExceptionInfo info;
   };

   exception BadCreationAttributeValue {
     ExceptionInfo info;
   };

   exception BadLocation {
     ExceptionInfo info;
   };

   exception BadPropertyValue {
     ExceptionInfo info;
   };

   exception BadQuery {
     ExceptionInfo info;
   };

   exception BadQueryAttribute {
     ExceptionInfo info;
   };

   exception BadQueryValue {
     ExceptionInfo info;
   };

   exception ImplementationLimit {
     ExceptionInfo info;
   };

   exception UnknownCallBack {
     ExceptionInfo info;
   };

   exception UnknownManagerType {
     ExceptionInfo info;
   };

   exception UnknownProduct {
     ExceptionInfo info;
   };

   exception UnknownProperty {
     ExceptionInfo info;
   };
```

```
exception UnknownResponse {
  ExceptionInfo info;
};

exception UnregisteredCallback {
  ExceptionInfo info;
};

exception ProductUnavailable {
  ExceptionInfo info;
};

exception BadOrder {
  ExceptionInfo info;
};

exception UnknownViewName {
  ExceptionInfo info;
};

exception BadSortAttribute {
  ExceptionInfo info;
};

exception NonUpdateableAttribute {
  ExceptionInfo info;
};

exception LockUnavailable {
  ExceptionInfo info;
};

exception UnsafeUpdate {
  ExceptionInfo info;
};

exception BadFileType {
  ExceptionInfo info;
};

exception BadResultAttribute {
  ExceptionInfo info;
};

exception UnknownEntity {
  ExceptionInfo info;
};

exception UnknownCreationAttribute {
  ExceptionInfo info;
};

exception BadUpdateAttribute {
  ExceptionInfo info;
};
```

```
   typedef sequence <Library> LibraryList;

/* The interfaces */

   interface LibraryManager {
     OGCBasic::NameList propertyNames ();

     PropertyList propertyValues (
       in OGCBasic::NameList desired_properties
       )
       raises (UnknownProperty);

     LibraryList libraries ();

   };

   interface Library {
     ManagerTypeList managerTypes ();

     LibraryManager manager (
       in ManagerType manager_type,
       in OGCBasic::NameValueList access_criteria
       )
       raises (UnknownManagerType,BadAccessCriteria,BadAccessValue);

     LibraryDescription libraryDescription ();

   };

   interface DataModelMgr : LibraryManager {
     ViewList listDataViews (
       in PropertyList properties
       )
       raises (UnknownProperty,BadPropertyValue);

     AttributeInformationList attributes (
       in ViewName view_name,
       in PropertyList properties
       )
       raises (UnknownViewName,UnknownProperty,BadPropertyValue);

     AttributeInformationList queryableAttributes (
       in ViewName view_name,
       in PropertyList properties
       )
       raises (UnknownViewName,UnknownProperty,BadPropertyValue);

     OGCBasic::DG_DirectedGraph entities (
       in ViewName view_name,
       in PropertyList properties
       )
       raises (UnknownViewName,UnknownProperty,BadPropertyValue);

     AttributeInformationList entityAttributes (
       in Entity entity,
       in PropertyList properties
       )
```

```
      raises (UnknownEntity,UnknownProperty,BadPropertyValue);

  };

  typedef sequence <Response> ResponseList;

  interface ResponseManager {
    ResponseList listActiveResponses ();

    OGCBasic::RelativeTime defaultTimeout ();

    void deleteRequest (
      in Response response
      )
      raises (UnknownResponse);

  };

  interface Response {
    OGCBasic::RequestDescription requestDescription ();

    void userInfo (
      in string message
      )
      raises (ImplementationLimit);

    OGCBasic::Status status ();

    void cancel ();

    OGCBasic::RelativeTime remainingDelay ();

    void registerCallback (
      in CB::Callback callback
      )
      raises (UnknownCallBack);

    void freeCallback (
      in CB::Callback callback
      )
      raises (UnknownCallBack,UnregisteredCallback);

    ResponseManager responseManager ();

  };

  interface OrderResponse : Response {
    OGCBasic::State complete ();

  };

  interface OrderMgr : LibraryManager, ResponseManager {
    OGCBasic::NameList packageSpecifications ();

    ValidationResults validateOrder (
      in OGCBasic::DG_DirectedGraph order,
      in PropertyList properties
```

```
      )
      raises (UnknownProperty,BadPropertyValue);

    OrderResponse order (
      in OGCBasic::DG_DirectedGraph order,
      in PropertyList properties
      )
      raises
        (UnknownProduct,BadOrder,UnknownProperty,BadPropertyValue,Pr
        oductUnavailable);

};

interface CreateResponse : Response {
  OGCBasic::State complete (
    out UID::Product new_product
    );

};

interface CreateMetaDataResponse : Response {
  OGCBasic::State complete (
    out UID::Product new_product
    );

};

interface CreationMgr : LibraryManager, ResponseManager {
  CreateResponse create (
    in OGCBasic::FileLocationList new_product,
    in OGCBasic::DG_DirectedGraph creation_metadata,
    in PropertyList properties
    )
    raises
      (BadLocation,UnknownCreationAttribute,BadCreationAttributeVa
      lue,UnknownProperty,BadPropertyValue);

  CreateMetaDataResponse createMetadata (
    in OGCBasic::DG_DirectedGraph creation_metadata,
    in ViewName view_name,
    in PropertyList properties
    )
    raises
      (UnknownCreationAttribute,BadCreationAttributeValue,UnknownP
      roperty,BadPropertyValue);

};

interface UpdateResponse : Response {
  OGCBasic::State complete ();

};

interface SubmitQueryResponse : Response {
  OGCBasic::State completeGraphResults (
    in unsigned long start_point,
    in unsigned long length,
```

```
      out QueryResults results
      );

    OGCBasic::State completeTableResults (
      in unsigned long start_point,
      in unsigned long length,
      out OGCBasic::NameValueTable results
      );

    OGCBasic::State completeXMLResults (
      in unsigned long start_point,
      in unsigned long length,
      out string results
      );

};

interface HitCountResponse : Response {
    OGCBasic::State complete (
      out unsigned long number_of_hits
      );

};

interface CatalogMgr : LibraryManager, ResponseManager {
    SubmitQueryResponse submitQuery (
      in ViewName view_name,
      in Query query,
      in OGCBasic::NameList result_attributes,
      in SortAttributeList sort_attributes,
      in PropertyList properties
      )
      raises
        (UnknownViewName,BadQuery,BadQueryAttribute,BadQueryValue,Ba
        dResultAttribute,BadSortAttribute,UnknownProperty,BadPropert
        yValue);

    HitCountResponse hitCount (
      in ViewName view_name,
      in Query query,
      in PropertyList properties
      )
      raises
        (UnknownViewName,BadQuery,BadQueryAttribute,BadQueryValue,Un
        knownProperty,BadPropertyValue);

    boolean validateQuery (
      in ViewName view_name,
      in Query query,
      in OGCBasic::NameList result_attributes,
      in SortAttributeList sort_attributes,
      in PropertyList properties
      );

};

interface UpdateByQueryResponse : Response {
```

```
      OGCBasic::State complete ();

    };

    interface UpdateMgr : LibraryManager, ResponseManager {
      UpdateResponse update (
        in ViewName view,
        in UpdateDG_DirectedGraphList changes,
        in PropertyList properties
        )
        raises (NonUpdateableAttribute,UnsafeUpdate,UnknownViewName);

      UpdateByQueryResponse updateByQuery (
        in OGCBasic::NameValue updated_attribute,
        in Query query,
        in ViewName view_name,
        in PropertyList properties
        )
        raises
          (NonUpdateableAttribute,BadUpdateAttribute,LockUnavailable,U
          nknownViewName,BadQuery,BadQueryAttribute,BadQueryValue,Unkn
          ownProperty,BadPropertyValue);

      void release_lock (
        in UID::Product lockedProduct
        );

    };

  };
  #endif
```

5.1.4 Module CB

```
  #ifndef __CB_DEFINED
  #define __CB_DEFINED

  #include "ogcBasic.idl"

  module CB {

    interface Callback {
      void notify (
        in OGCBasic::RequestDescription description
        );

      void release ();

    };

  };
  #endif
```

5.1.5 Module UID

```
  #ifndef __UID_DEFINED
  #define __UID_DEFINED
```

```
module UID {

   interface Product {
   };

};
#endif
```

5.1.6 Module OGCBasic

```
#ifndef __OGCBASIC_DEFINED
#define __OGCBASIC_DEFINED

module OGCBasic {

   typedef string Name;

   typedef sequence <Name> NameList;

   struct NameValue {
      OGCBasic::Name theName;
      any theValue;
   };

   typedef sequence <NameValue> NameValueList;

   typedef sequence <NameValueList> NameValueTable;

   /* Placeholder - What is OGC definition for Rectangle? */
   typedef string Rectangle;

   typedef sequence <Rectangle> RectangleList;

   struct FileLocation {
      string user_name;
      string password;
      string host_name;
      string path_name;
      string file_name;
   };

   typedef sequence <FileLocation> FileLocationList;

   struct Date {
      unsigned short year;
      unsigned short month;
      unsigned short day;
   };

   struct Time {
      unsigned short hour;
      unsigned short minute;
      float second;
```

```
};

struct AbsTime {
   Date date;
   Time time;
};

typedef AbsTime RelativeTime;

typedef sequence <octet> BinData;

typedef unsigned long NodeID;

enum NodeType {
   ROOT_NODE,
   ENTITY_NODE,
   RECORD_NODE,
   ATTRIBUTE_NODE
};

struct Node {
   NodeID id;
   NodeType node_type;
   string attribute_name;
   any value;
};

enum Cardinality {
   ONE_TO_ONE,
   ONE_TO_MANY,
   MANY_TO_ONE,
   MANY_TO_MANY
};

struct Edge {
   NodeID start_node;
   NodeID end_node;
   string relationship_type;
   Cardinality start_to_end_card;
   Cardinality end_to_start_card;
};

typedef sequence <Node> NodeList;

typedef sequence <Edge> EdgeList;

struct DG_DirectedGraph {
   NodeList nodes;
   EdgeList edges;
};

typedef sequence <DG_DirectedGraph> DG_DirectedGraphList;

enum State {
   COMPLETED,
   IN_PROGRESS,
   ABORTED,
```

```
      CANCELED,
      PENDING,
      SUSPENDED,
      RESULTS_AVAILABLE,
      TRANSFER_COMPLETE
    };

    struct Status {
      State completion_state;
      boolean warning;
      string status_message;
    };

    struct RequestDescription {
      string user_info;
      string request_type;
      string request_info;
      NameValueList request_details;
    };

    typedef sequence <RequestDescription> RequestDescriptionList;

};
#endif
```

5.2 Coarse-Grained CORBA Profile

5.2.1 Architecture - Object Model

This paragraph describes the coarse grained CORBA profile. The intention of the coarse grained CORBA profile is to follow the general model closely. This enables the building of lightweight bridges between the coarse-grained CORBA profile and the coarse-grained WWW profile.

The CORBA profile is described in IDL (interface definition language) of OMG (the Object Management Group).

5.2.2 Event Traces

The interfaces in the IDL follow the general model as closely as possible. Therefore all conventions, operation names and cases are borrowed from the general model. An alternative is using the conventions of the CORBA IDL for Simple Features, in which all names are in lower case. This alternative is rejected to stay close to the general model.

The core of the coarse grained CORBA profile consists of only one interface: CG_CatalogService. The separate services of the general model (discovery, access and management) are defined in separate interfaces to reflect the general model. They are all realized by the central interface CG_CatalogService. The operations of CG_CatalogService take without exception a request message as an input parameter and return a response parameter. All messages are filled with standard or compound CORBA structures. Name value pairs, an optional way to transfer meta information, are borrowed from the Simple Feature specification.

5.2.3 Interface Definition - IDL

This section describes the CORBA IDL. First enumerations are described. Then structures and unions will be treated. After that the messages are described. Finally the core of the profile, the CG_CatalogService interface and other interfaces are discussed.

All enumerations, structures, unions, messages and interfaces are part of the module OGC_CatalogService.

```
module OGC_CatalogService
{
...
};
```

5.2.3.1 Enumerations

Enumerations can be modeled by a direct translation of all code-lists of the general model. The following enumerations are borrowed literally:

```
enum CG_AttributeCategory {queriable, presentable, both};
enum CG_CatalogEntryType {product, collection, catalog, service};
enum CG_CharacterSet {ASCII, uniCode, shiftJIS, uniCodeJ};
enum CG_PredefinedPresentationType {full, brief};
enum CG_QueryLanguage {OGC_Common, Z3950_TypeOne, SQL3_SimpleFeature,
        SQL2_SimpleFeature};
enum CG_QueryScope {distributed, local};
enum CG_ResultType {validate, resultSetID, hits, results};
```

```
enum CG_SortOrder {ascending, descending, ascendingSize,
          descendingSize, none};
enum CG_Status {success, processing, failure, canceled, queued,
          pausedOrSuspended, resultsAvailable, accessDenied};
```

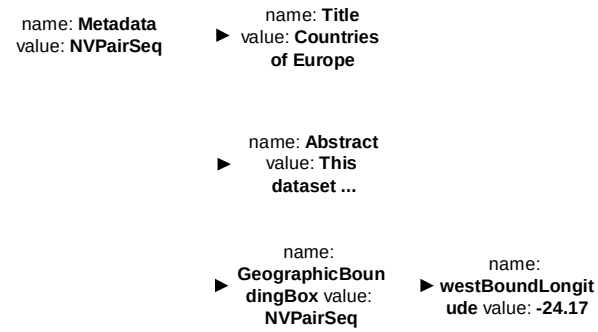
The coarse grained CORBA profile adds a **NV** entry to the message format enumeration. Specifying NV let the server give results back as name value pairs. Name-value pairs are specified in the simple feature specification, but to be complete the definition is repeated below.

```
enum CG_MessageFormat {XML, DG_DirectedGraph, HTML, TXT, NV};

module OGIS
{
   ...
   struct NVPair
   {
      string name;
      any value;
   };

   typedef sequence<NVPair> NVPairSeq;
   ...
};
```

So if the server gives the results back as XML in the next example:

```
<?xml version="1.0"?>
<!DOCTYPE Metadata SYSTEM "min.dtd" >
<Metadata><Title>Countries of Europe</Title>
<Abstract>This dataset contains the countries of Europe</Abstract>
<GeographicBoundingBox><westBoundLongitude>-
          24.17</westBoundLongitude>
<eastBoundLongitude>40.71</eastBoundLongitude>
<northBoundLatitude>71.26</northBoundLatitude>
<southBoundLatitude>27.63</southBoundLatitude>
</GeographicBoundingBox>
</Metadata>
```

In name-value pairs the results are as follows:



The advantage is that pure CORBA environments do not have to parse the XML to get the results. They receive them in a suitable general structure. If the CORBA server is combined with another type of client, e.g. a web client, then probably XML (the default) will be preferred.

5.2.3.2 Structures and unions

Also most of the structures and unions from the general model can be translated directly into CORBA structs and unions.

```
union CG_CollectionName
  switch(long)
  {
    case 1 : string collectionID;
    case 2 : string collectionName;
  };

struct CG_QueryExpression
{
  string theQuery;
  string theNamespace;
  CG_QueryLanguage theLanguage;
};

struct CG_RequestID
{
  long sessionID;
  long counter;
};

struct CG_SortField
{
  string attributeName;
  CG_SortOrder sortOrder;
};
```

The General Model specifies for the CG_ReturnData the structure member **payload** as string, indicating it as a 'blob'. For CORBA it is more correct to specify an **any** structure member here, in where strings or directed graphs, name-value pairs or sequences can be stored.

```
struct CG_ReturnData
{
  CG_MessageFormat encoding;
  any payload;
};
```

The CG_PresentationDescription union in the General Model contains a sequence of tuple-types in the presentation description. For the coarse grained CORBA profile it is not necessary to have tuple-types here, a sequence of attribute names is sufficient. The tuple-types are not defined in the coarse grained CORBA profile.

```
union CG_PresentationDescription
  switch(long)
  {
    case 1 : sequence<string> attributes; // CG_TupleType in GM
```

```
      case 2 : CG_PredefinedPresentationType presentationType;
    };
```

The CG_SchemeID structure uses a structure member CG_Schema. This is in the CORBA profile defined as a sequence of name-value pairs from the Simple Feature Specification for CORBA. In name-value pairs all names, types, used sequences can be specified. A schema, tuple-type or a dictionary is not needed here. If a schema or anything like that is specified in a general OGIS module later on, it could be taken over here.

```
typedef OGIS::NVPairSeq CG_Schema;
struct CG_SchemaID
{
  string schemeName;
  CG_Schema schema;
};
```

5.2.3.3 Definitions for brokered access

The general model defines some code-lists and structures for brokered access. These definitions are directly translated into their CORBA counterparts:

```
enum CG_BrokeredAccessRequestType {orderEstimate,
         orderQuoteAndSubmit,
       orderMonitor, orderCancel};

struct CG_OrderItem
{
  // Note: datatypes not provided by GM
  any productID;
  any productPrice;
  any productDeliveryOptions;
  any processingOptions;
  any sceneSelectionOptions;
};

struct CG_OrderSpecification
{
  // Note: datatypes not provided by GM
  any orderCentreID;
  any orderPrice;
  any orderDeliveryDate;
  any orderCancellationDate;
  any deliveryMethod;
  any package;
};

enum CG_OrderStatus {orderBeingEstimated, orderEstimated,
  orderBeingQuoted, orderBeingProcessed,
  orderCompleted, orderNotValid, orderCancelled};

enum CG_PackagingType {predefinedPackage, adhocPackage};

struct CG_PackageSpecification
{
  // Note: datatypes not provided by GM
  any packageId;
```

```
    any packagePrice;
    CG_PackagingType package;
    any packageMedium;
    long packageSize;
};

enum CG_PaymentMethod {credit, cash, purchaseOrder};
enum CG_StatusUpdateType {manual, automatic};
struct CG_UserInformation
{
    string userName;
    string userAddress;
    string phoneNumber;
    string faxNumber;
    string emailAddress;
    string netAddress;
    CG_PaymentMethod paymentMethod;
};
```

5.2.3.4 Capabilities

The capabilities in the general model are designed with inheritance. In CORBA designing capabilities as interfaces can reflect this, but this is not useful. Capabilities (like messages, see below) have to be transferred over the network. Therefore they are defined as either type definitions or structures.

```
typedef boolean CG_AllSupportedRequest;

typedef boolean CG_Defaults;

struct CG_DefaultTimeOut
{
    OGC_Basic::UomTime timeOut;
};

typedef boolean CG_Explain;

struct CG_Messaging
{
    CG_CharacterSet characterSet;
    CG_MessageFormat messageFormat;
};

struct CG_Query
{
    string version;
    CG_CharacterSet characterSet;
    CG_QueryLanguage queryLanguage;
};

struct CG_Session
{
    string language;
    string catalogSpecificationVersion;
    CG_CharacterSet characterSet;
};
```

```
struct CG_SoftwareInformation
{
  string vendor;
  string SVversionNumber;
  string IFversionNumber;
};

typedef sequence<CG_CollectionName> CG_SupportedCollections;
```

To be able to make a sequence of different capabilities, a union CG_Capability is created, encompassing all derived capabilities.

A union normally has a discriminator. This can be a **long** value, but this is generally not preferred because you have to remember the value indicating the intended capability. Therefore an enumeration of capabilities is included in the CORBA profile.

```
enum CG_CapabilityType
  { ctAllSupportedRequest, ctDefaults, ctDefaultTimeOut, ctExplain,
        ctMessaging,  ctQuery, ctSession, ctSoftwareInformation,
        ctSupportedCollections };

union CG_Capability
  switch(CG_CapabilityType)
  {
    case ctAllSupportedRequest : CG_AllSupportedRequest
         allSupportedRequest;
    case ctDefaults : CG_Defaults defaults;
    case ctDefaultTimeOut : CG_DefaultTimeOut timeOut;
    case ctExplain : CG_Explain explain;
    case ctMessaging : CG_Messaging messaging;
    case ctQuery : CG_Query query;
    case ctSession : CG_Session session;
    case ctSoftwareInformation : CG_SoftwareInformation
         softwareInformation;
    case ctSupportedCollections : CG_SupportedCollections
         supportedCollections;
  };
```

5.2.3.5 General messages

The general model (coarse grained) is a message based model, where messages are designed in the form of a class hierarchy. In CORBA IDL the messages are translated as structs. Writing them in the form of interfaces is not useful. In CORBA the objects (instances of interfaces) stay on a remote server machine and are referred to by a client machine. They are not transferred over the network. This is definitely not the intention for messages.

All messages have the same form as the messages described in the general model. However, messages in the form of structs cannot inherit from each other in CORBA. Therefore the CG_Message class is also included in the CORBA profile and a member of all other messages, called 'base'.

```
struct CG_Message
{
```

```
    long sessionID;
    string destinationID;
    CG_RequestID requestID;
    string additionalInfo;
    CG_MessageFormat format;
};
```

All other messages, which in the general model inherit from CG_Message, have in the CORBA profile the CG_Message as a structure member. The next messages do not add extra structure members. They could also have been modeled by a typedef. But to be consistent with the rest of the messages also these message have **base** as a structure member.

Note that the response in the general model also contains a string structure member **diagnostic**. This parameter is not specified in the CORBA profile. Error handling will be handled by exceptions, the standard CORBA facility. Exceptions are described below. WWW/CORBA bridges can catch these exceptions and convert them into diagnostic info if necessary.

```
struct CG_InitSessionRequest
{
  CG_Message base;
};

struct CG_InitSessionResponse
{
  CG_Message base;
};
struct CG_TerminateSessionRequest
{
  CG_Message base;
};
struct CG_TerminateSessionResponse
{
  CG_Message base;
};
```

The status and cancel messages add a few structure members in addition to the **base** structure member.

```
struct CG_StatusRequest
{
  CG_BaseMessage base;
  CG_RequestID requestIDtoStatus;
};

struct CG_StatusResponse
{
  CG_BaseMessage base;
  CG_RequestID requestIDtoStatus;
  CG_Status status;
};
struct CG_CancelRequest
{
  CG_BaseMessage base;
  CG_RequestID requestIDtoCancel;
  boolean freeResources;
};
```

```
struct CG_CancelResponse
{
  CG_BaseMessage base;
  CG_RequestID requestIDtoCancel;
  CG_Status status;
  CG_RequestID canceledRequest;
};
```

5.2.3.6 Discovery messages

There are three request/response message pairs in the discovery service. To enhance distributed searching, an additional structure member for the query message is provided, this member is not included in the general model. This structure member **asynchronous** can be set to **true** to force asynchronous searching. The query method will return immediately, setting structure member **hits** in the response to zero. Query results can be retrieved later on, when the query is ready. The progress of the query can be examined with the status messages. The query can be cancelled with the cancel messages.

Another structure member, **maxLevel**, is added to have more control in the range of the distribution. If one catalog contains another one, that other one contains a third one, and so on, you will possibly specify that only two levels of sub-catalogs will be searched through. Setting the **maxLevel** member to two will force this. Setting **maxLevel** to -1 forces searching all sub-catalogs. Note that if the **queryScope** is **Local** there is no distributed search at all.

```
struct CG_QueryRequest
{
  CG_BaseMessage base;
  CG_QueryExpression queryExpression;
  CG_ResultType resultType;
  long iteratorSize;
  long cursor;
  CG_MessageFormat returnFormat;
  CG_PresentationDescription presentation;
  sequence<CG_SortField> sortField;
  CG_QueryScope queryScope;
  CG_CollectionName collectionID;
  CG_CatalogEntryType catalogType;
  boolean asynchronous;
  long maxLevel;
};
struct CG_QueryResponse
{
  CG_Message base;
  CG_ReturnData retrievedData;
  CG_CollectionName resultSetID;
  CG_Status status;
  long hits;
  long cursor;
};

struct CG_PresentRequest
{
  CG_Message base;
  CG_PresentationDescription presentation;
  sequence<CG_SortField> sortField;
  CG_MessageFormat returnFormat;
```

```
    long iteratorSize;
    long cursor;
    CG_CollectionName resultSetID;
  };


  struct CG_PresentResponse
  {
    CG_Message base;
    CG_ReturnData retrievedData;
    long cursor;
    long hits;
    CG_Status status;
  };

  struct CG_ExplainCollectionRequest
  {
    CG_Message base;
    CG_AttributeCategory attributeCategory;
    CG_CollectionName collectionID;
  };

  struct CG_ExplainCollectionResponse
  {
    CG_Message base;
    CG_CollectionName collectionID;
    CG_SchemaID dataModel;
    CG_Status status;
  };
```

Note that this asynchronous behavior is only specified for the query request message. All other operations (e.g. init, terminate, status, cancel, explain, present) are not considered as time-consuming and return immediately after processing.

5.2.3.7 Management messages

The general model defines messages for managing catalogs. These messages are translated to the CORBA coarse grained profile literally. The general model must still define the contents of the messages. Therefore not all messages are taken described here, they are described in the full IDL below.

```
  struct CG_CreateCatalogRequest
  {
    CG_Message base;
    // tbd
  };
  struct CG_CreateCatalogResponse
  {
    CG_Message base;
    // tbd
  };
  ...
```

5.2.3.8 Access messages

The general model specifies direct access and brokered access. Direct access is provided by interfaces such as the OGC Simple Features and Coverage interfaces for CORBA. If a catalog entry denotes an OGC Feature, a Feature Collection or a Coverage, the meta-information of this entry can be populated with an **ior** (interoperable object reference). This meta-information entity is called **ior** and is filled with the standard representation of an **ior**, specified by the OMG (Object Management Group), the creators of CORBA. In XML this looks like the following (abbreviated) example:

```
<ior>IOR:010631002800000049444c3a6f6d672e6f...</ior>
```

Brokered access is specified by a request and a response message, conform all operations of the general model. The messages are listed below.

```
struct CG_BrokeredAccessRequest
{
  CG_Message base;
  string productHandle;
  CG_OrderSpecification orderInformation;
  string orderID;
  CG_BrokeredAccessRequestType requestType;
  CG_UserInformation userInformation;
  CG_StatusUpdateType statusOrderUpdateType;
  CG_PackageSpecification packageSpecification;
};

struct CG_BrokeredAccessResponse
{
  CG_Message base;
  CG_OrderStatus orderStatus;
  sequence<long> resourceEstimate;
  CG_CollectionName order;
  string orderID;
  CG_Status status;
  CG_BrokeredAccessRequestType requestType;
};
```

5.2.3.9 Exceptions

Exceptions are not specified in the general model because they are profile specific. In CORBA exceptions are considered as an appropriate way to notify error situations to clients. The coarse grained CORBA profile specifies exceptions. The **diagnostic** structure member of the response messages are not used in the CORBA profile, their role is taken over by the exceptions. Some exceptions specify the **diagnostic** string as an exception parameter. By other exceptions this is not necessary, as the exceptions are self-explaining.

```
exception InvalidRequest{};
exception InvalidSession{};
exception InvalidCollection{ string diagnostic; };
```

The exception InvalidQuery is thrown if the client specifies an invalid query. Note that the exception is not thrown if the **resultType** field is set to **validate**.

```
exception InvalidQuery{ string diagnostic; };
```

The exception NotImplemented is defined in cases where not-implemented behavior is asked by the client. This might occur by requesting the optional access or management services.

```
exception NotImplemented{ string diagnostic; };
```

The NotSupported exception is thrown if the client specifies something in a request parameter that is not implemented by the server. For example the client can specify its query in Z3950_TypeOne but the server can only interpret OGC_Common queries.

```
exception NotSupported{ string diagnostic; };
```

The last exception, CatalogError, is to notify error situations where none of the above exceptions is appropriate.

```
exception CatalogError{ string diagnostic; };
```

5.2.3.10 Catalog Service interfaces

The interface CG_Discovery implements methods for discovery: **query**, **present** and **explainCollection**. These methods take a request message as input parameter and return a response message as output parameter.

```
interface CG_Discovery
{
  CG_QueryResponse query(in CG_QueryRequest request)
      raises(InvalidSession, CatalogError);
  CG_PresentResponse present(in CG_PresentRequest request)
      raises(InvalidSession, CatalogError);
  CG_ExplainCollectionResponse explainCollection(in
         CG_ExplainCollectionRequest request)
      raises(CatalogError);
};
```

The next interface describes the CG_Manager interface, which defines catalog management functions. All methods are taken literally from the general model. These methods create, update or delete catalog entries. In the request messages the appropriate meta information will be provided.

By specifying a CORBA **ior** (interoperable object reference) in the meta-information, the following functions are possible:

- direct access to OGC Simple Features, OGC Feature Collections or OGC Coverages
- distributed search through multiple catalog services

To enable this functionality the field **ior** must be filled with the correct ior in the standard OMG ior string representation.

```
interface CG_Manager
{
  CG_CreateMetadataResponse
       createMetadata(in CG_CreateMetadataRequest request)
         raises(NotImplemented, CatalogError);
  CG_CreateCatalogResponse
```

```
        createCatalog(in CG_CreateCatalogRequest request)
          raises(NotImplemented, CatalogError);
    CG_UpdateCatalogResponse
        updateCatalog(in CG_UpdateCatalogRequest request)
          raises(NotImplemented, CatalogError);
    CG_DeleteCatalogResponse
        deleteCatalog(in CG_DeleteCatalogRequest request)
          raises(NotImplemented, CatalogError);
};
```

The interface CG_Access is the interface for access messages. It describes only one operation: the brokeredAccess function which has the request as input and which returns the response. Direct access is provided by interfaces as the Simple Feature interface and the Coverage interface. These interfaces are not described here. The client can get a reference to these interfaces by examining the **ior** field in the meta-information.

```
interface CG_Access
{
  CG_BrokeredAccessResponse
       brokeredAccess(in CG_BrokeredAccessRequest request)
         raises(NotImplemented, CatalogError);
};
```

The CG_CatalogService interface is the core of the coarse grained CORBA profile. All operations have a comparable form of the operations specified in the general model. This consists of a request message as an input parameter and a response message as a return value.

The CG_CatalogService inherits from the interfaces CG_Discovery, CG_Access and CG_Manager. In this way these services are realized. Note that access and manager services are optional. If a server does not implement these services it throws the exception **NotImplemented**. The CG_CatalogService also inherits from OGC_StatefulService that is described below.

```
interface CG_CatalogServices : OGC_StatefulService, CG_Discovery,
         CG_Access, CG_Manager
{
  CG_InitSessionResponse initSession(in CG_InitSessionRequest
         request)
       raises(CatalogError);
  CG_TerminateSessionResponse terminateSession(in
         CG_TerminateSessionRequest request)
       raises(InvalidSession, CatalogError);
  CG_ExplainServerResponse explainServer(in CG_ExplainServerRequest
         request)
       raises(CatalogError);

  CG_StatusResponse status(in CG_StatusRequest request)
       raises(InvalidSession, InvalidRequest, CatalogError);
  CG_CancelResponse cancel(in CG_CancelRequest request)
       raises(InvalidSession, InvalidRequest, CatalogError);
};
```

5.2.3.11 Basic interfaces

Because of the asynchronous behavior of the query operation, a callback notifying the termination of the query could be useful. The Observer Design Pattern [GAMMA97] describes a standard mechanism for notifications to one or more clients. We envision that such a mechanism will be useful for many operations in the OpenGIS world. Therefore the OGC_Observer and the OGC_Subject interfaces are modeled separately. These interfaces could be moved to an OGC general module, in the same or a similar form. The next interfaces describe the mechanism. Note that they are not mentioned in the general model, as this is a CORBA specific behavior.

```
interface OGC_Observer;

interface OGC_Subject
{
  void attachObserver(in OGC_Observer Observer);
  void detatchObserver(in OGC_Observer Observer);
  void notifyObserver();
};

interface OGC_Observer
{
  void updateSubject(in OGC_Subject ChangedSubject);
};
```

The CG_CatalogService interface inherits from **OGC_Service**. This is envisioned as the basic interface for all OpenGIS services. As it does not exist yet, the content of this interface is not clear.

```
interface OGC_Service : OGC_Subject
{
};

interface OGC_StatefulService : OGC_Service
{
};
```

5.2.3.12 Complete IDL

```
//------------------------------------------------------------------
// Module       : OGC_CatalogService
//------------------------------------------------------------------
// Purpose      : Coarse grained CORBA profile for catalog services
//------------------------------------------------------------------
// Authors      : Barend Gehrels, Geodan IT b.v., the Netherlands
//                Joined Catalog Response Team
// Date         : july 13, 1999
//------------------------------------------------------------------

module OGC_CatalogService
{

  //---------------------------------------------------------------
  // Parameter type definitions
  //---------------------------------------------------------------

  // 3.2.7.1
  enum CG_AttributeCategory {queriable, presentable, both};
```

```
// 3.2.7.2
enum CG_BrokeredAccessRequestType {orderEstimate,
       orderQuoteAndSubmit,
       orderMonitor, orderCancel};

// 3.2.7.3 capabilities see below

// 3.2.7.4
enum CG_CatalogEntryType {product, collection, catalog, service};

// 3.2.7.5
enum CG_CharacterSet {ASCII, uniCode, shiftJIS, uniCodeJ};

// 3.2.7.6
union CG_CollectionName
  switch(long)
  {
    case 1 : string collectionID;
    case 2 : string collectionName;
  };

// 3.2.7.7 CG_Dictionary see CG_Scheme

// 3.2.7.8
enum CG_MessageFormat {XML, DG_DirectedGraph, HTML, TXT, NV};

// 3.2.7.9
struct CG_OrderItem
{
  // Note: datatypes not provided by GM
  any productID;
  any productPrice;
  any productDeliveryOptions;
  any processingOptions;
  any sceneSelectionOptions;
};

// 3.2.7.10
struct CG_OrderSpecification
{
  // Note: datatypes not provided by GM
  any orderCentreID;
  any orderPrice;
  any orderDeliveryDate;
  any orderCancellationDate;
  any deliveryMethod;
  any package;
};

// 3.2.7.11
enum CG_OrderStatus {orderBeingEstimated, orderEstimated,
  orderBeingQuoted, orderBeingProcessed,
  orderCompleted, orderNotValid, orderCancelled};

// 3.2.7.13
enum CG_PackagingType {predefinedPackage, adhocPackage};
```

```
// 3.2.7.12
struct CG_PackageSpecification
{
  // Note: datatypes not provided by GM
  any packageId;
  any packagePrice;
  CG_PackagingType package;
  any packageMedium;
  long packageSize;
};

// 3.2.7.14
enum CG_PaymentMethod {credit, cash, purchaseOrder};

// 3.2.7.15
enum CG_PredefinedPresentationType {full, brief};

// 3.2.7.16
union CG_PresentationDescription
  switch(long)
  {
    case 1 : sequence<string> attributes; // CG_TupleType in GM
    case 2 : CG_PredefinedPresentationType presentationType; //
      name in GM
  };


// 3.2.7.3.7
enum CG_QueryLanguage {OGC_Common, Z3950_TypeOne,
      SQL3_SimpleFeature, SQL2_SimpleFeature};

// 3.2.7.17
struct CG_QueryExpression
{
  string theQuery;
  string theNamespace;
  CG_QueryLanguage theLanguage;
};

// 3.2.7.18
enum CG_QueryScope {distributed, local};

// 3.2.7.19
struct CG_RequestID
{
  long sessionID;
  long counter;
};

// 3.2.7.20
enum CG_ResultType {validate, resultSetID, hits, results};

// 3.2.7.21
struct CG_ReturnData
{
  CG_MessageFormat encoding;
```

```
  any payload;

  // XML,HTML,TXT will return a string
  // DAG will return a DAG-structure
  // NV will return a OGIS::NVPairSeq (from CORBA SF)
};

// 3.2.7.22 CG_Scheme
typedef OGIS::NVPairSeq CG_Schema;

// 3.2.7.23 CG_SchemeID
struct CG_SchemaID
{
  string schemeName;
  CG_Schema schema;
};

// 3.2.7.25
enum CG_SortOrder {ascending, descending, ascendingSize,
       descendingSize, none};

// 3.2.7.24
struct CG_SortField
{
  string attributeName;
  CG_SortOrder sortOrder;
};

// 3.2.7.26
enum CG_Status {success, processing, failure, canceled, queued,
     pausedOrSuspended, resultsAvailable, accessDenied};

// 3.2.7.27
enum CG_StatusUpdateType {manual, automatic};

// 3.2.7.28 CG_TupleType

// 3.2.7.29
struct CG_UserInformation
{
  string userName;
  string userAddress;
  string phoneNumber;
  string faxNumber;
  string emailAddress;
  string netAddress;
  CG_PaymentMethod paymentMethod;
};

//-------------------------------------------------------------------
// Capabilities, 3.2.7.3
//-------------------------------------------------------------------
enum CG_CapabilityType
  { ctAllSupportedRequest, ctDefaults, ctDefaultTimeOut,
    ctExplain, ctMessaging,  ctQuery, ctSession,
    ctSoftwareInformation, ctSupportedCollections };
```

```
// 3.2.7.3.1
typedef boolean CG_AllSupportedRequest;

// 3.2.7.3.2
typedef boolean CG_Defaults;

// 3.2.7.3.3
struct CG_DefaultTimeOut
{
  OGC_Basic::UomTime timeOut;
};

// 3.2.7.3.4
typedef boolean CG_Explain;

// 3.2.7.3.5
struct CG_Messaging
{
  CG_CharacterSet characterSet;
  CG_MessageFormat messageFormat;
};

// 3.2.7.3.6
struct CG_Query
{
  string version;
  CG_CharacterSet characterSet;
  CG_QueryLanguage queryLanguage;
};

// 3.2.7.3.8
struct CG_Session
{
  string language;
  string catalogSpecificationVersion;
  CG_CharacterSet characterSet;
};

// 3.2.7.3.9
struct CG_SoftwareInformation
{
  string vendor;
  string SVversionNumber;
  string IFversionNumber;
};

// 3.2.7.3.10
typedef sequence<CG_CollectionName> CG_SupportedCollections;

// 3.2.7.3
union CG_Capability
  switch(CG_CapabilityType)
  {
    case ctAllSupportedRequest : CG_AllSupportedRequest
      allSupportedRequest;
    case ctDefaults : CG_Defaults defaults;
```

```
      case ctDefaultTimeOut : CG_DefaultTimeOut timeOut;
      case ctExplain : CG_Explain explain;
      case ctMessaging : CG_Messaging messaging;
      case ctQuery : CG_Query query;
      case ctSession : CG_Session session;
      case ctSoftwareInformation : CG_SoftwareInformation
        softwareInformation;
      case ctSupportedCollections : CG_SupportedCollections
        supportedCollections;
    };


//---------------------------------------------------------------------
// Messages
//---------------------------------------------------------------------

struct CG_Message
{
   long sessionID;
   string destinationID;
   CG_RequestID requestID;
   string additionalInfo;
   CG_MessageFormat format;
};

struct CG_InitSessionRequest
{
   CG_Message base;
};

struct CG_InitSessionResponse
{
   CG_Message base;
};
struct CG_TerminateSessionRequest
{
   CG_Message base;
};
struct CG_TerminateSessionResponse
{
   CG_Message base;
};

struct CG_ExplainServerRequest
{
   CG_Message base;
   sequence<CG_CapabilityType> capabilities;
};

struct CG_ExplainServerResponse
{
   CG_Message base;
   sequence<CG_Capability> capabilities;
};

struct CG_StatusRequest
{
```

```
  CG_Message base;
  CG_RequestID requestIDtoStatus;
};

struct CG_StatusResponse
{
  CG_Message base;
  CG_RequestID requestIDtoStatus;
  CG_Status status;
};

struct CG_CancelRequest
{
  CG_Message base;
  CG_RequestID requestIDtoCancel;
  boolean freeResources;
};

struct CG_CancelResponse
{
  CG_Message base;
  CG_RequestID requestIDtoCancel;
  CG_Status status;
  CG_RequestID canceledRequest;
};
struct CG_QueryRequest
{
  CG_Message base;

  CG_QueryExpression queryExpression;
  CG_ResultType resultType;
  long iteratorSize;
  long cursor;
  CG_MessageFormat returnFormat;
  CG_PresentationDescription presentation;
  sequence<CG_SortField> sortField;
  CG_QueryScope queryScope;
  CG_CollectionName collectionID;
  CG_CatalogEntryType catalogType;

  boolean asynchronous;
  long maxLevel;
};

struct CG_QueryResponse
{
  CG_Message base;
  CG_ReturnData retrievedData;
  CG_CollectionName resultSetID;
  CG_Status status;
  long hits;
  long cursor;
};

struct CG_PresentRequest
{
  CG_Message base;
```

```
  CG_PresentationDescription presentation;
  sequence<CG_SortField> sortField;
  CG_MessageFormat returnFormat;
  long iteratorSize;
  long cursor;
  CG_CollectionName resultSetID;
};
struct CG_PresentResponse
{
  CG_Message base;
  CG_ReturnData retrievedData;
  long cursor;
  long hits;
  CG_Status status;
};

struct CG_ExplainCollectionRequest
{
  CG_Message base;
  CG_AttributeCategory attributeCategory;
  CG_CollectionName collectionID;
};
struct CG_ExplainCollectionResponse
{
  CG_Message base;
  CG_CollectionName collectionID;
  CG_SchemaID dataModel;
  CG_Status status;
};

// Messages for access
// 3.2.5.1
struct CG_BrokeredAccessRequest
{
  CG_Message base;
  string productHandle;
  CG_OrderSpecification orderInformation;
  string orderID;
  CG_BrokeredAccessRequestType requestType;
  CG_UserInformation userInformation;
  CG_StatusUpdateType statusOrderUpdateType;
  CG_PackageSpecification packageSpecification;
};

// 3.2.5.2
struct CG_BrokeredAccessResponse
{
  CG_Message base;
  CG_OrderStatus orderStatus;
  sequence<long> resourceEstimate;
  CG_CollectionName order;
  string orderID;
  CG_Status status;
  CG_BrokeredAccessRequestType requestType;
};
```

```
// Messages for managing functions
struct CG_CreateCatalogRequest
{
  CG_Message base;
  // tbd
};
struct CG_CreateCatalogResponse
{
  CG_Message base;
  // tbd
};
struct CG_UpdateCatalogRequest
{
  CG_Message base;
  // tbd
};
struct CG_UpdateCatalogResponse
{
  CG_Message base;
  // tbd
};
struct CG_DeleteCatalogRequest
{
  CG_Message base;
  // tbd
};
struct CG_DeleteCatalogResponse
{
  CG_Message base;
  // tbd
};
struct CG_CreateMetadataRequest
{
  CG_Message base;
  // tbd
};
struct CG_CreateMetadataResponse
{
  CG_Message base;
  // tbd
};



//------------------------------------------------------------------
// Exceptions
//------------------------------------------------------------------
exception InvalidSession{};
exception InvalidRequest{};
exception InvalidCollection{ string diagnostic; };
exception InvalidQuery{ string diagnostic; };
exception NotImplemented{ string diagnostic; };
exception NotSupported{ string diagnostic; };
exception CatalogError{ string diagnostic; };

//------------------------------------------------------------------
// Interfaces
```

```
//--------------------------------------------------------------------

interface OGC_Observer;

interface OGC_Subject
{
  oneway void attachObserver(in OGC_Observer Observer);
  oneway void detachObserver(in OGC_Observer Observer);
  oneway void notifyObserver();
};

interface OGC_Observer
{
  void updateSubject(in OGC_Subject ChangedSubject);
};

interface OGC_Service : OGC_Subject
{
};
interface OGC_StatefulService : OGC_Service
{
};


interface CG_Discovery
{
  CG_QueryResponse query(in CG_QueryRequest request)
       raises(InvalidSession, InvalidQuery, InvalidCollection,
       NotSupported, CatalogError);
  CG_PresentResponse present(in CG_PresentRequest request)
       raises(InvalidSession, InvalidCollection, NotSupported,
       CatalogError);
  CG_ExplainCollectionResponse explainCollection(in
       CG_ExplainCollectionRequest request)
       raises(CatalogError);
};

interface CG_CatalogServices;

interface CG_Access
{
  // Direct access is provided by the IOR fields in the meta-
       information
  // itself

  // Brokered access
  CG_BrokeredAccessResponse
       brokeredAccess(in CG_BrokeredAccessRequest request)
       raises(NotImplemented, CatalogError);
};

interface CG_Manager
{
  CG_CreateMetadataResponse
       createMetadata(in CG_CreateMetadataRequest request)
       raises(NotImplemented, CatalogError);
  CG_CreateCatalogResponse
```

```
            createCatalog(in CG_CreateCatalogRequest request)
            raises(NotImplemented, CatalogError);
        CG_UpdateCatalogResponse
            updateCatalog(in CG_UpdateCatalogRequest request)
            raises(NotImplemented, CatalogError);
        CG_DeleteCatalogResponse
            deleteCatalog(in CG_DeleteCatalogRequest request)
            raises(NotImplemented, CatalogError);

    };

    interface CG_CatalogServices : OGC_StatefulService, CG_Discovery,
            CG_Access, CG_Manager
    {
        CG_InitSessionResponse initSession(in CG_InitSessionRequest
            request)
            raises(CatalogError);
        CG_TerminateSessionResponse terminateSession(in
            CG_TerminateSessionRequest request)
            raises(InvalidSession, CatalogError);
        CG_ExplainServerResponse explainServer(in
            CG_ExplainServerRequest request)
            raises(CatalogError);

        CG_StatusResponse status(in CG_StatusRequest request)
            raises(InvalidSession, InvalidRequest, CatalogError);
        CG_CancelResponse cancel(in CG_CancelRequest request)
            raises(InvalidSession, InvalidRequest, CatalogError);
    };
};
```

6. OLEDB Profile

6.1 Architecture

The COM Profile uses OLEDB as the mechanism for accessing catalog data. OLEDB is the standard within the Microsoft developer community for locating and exchanging data. As such, this profile addresses two classes of catalog environment, those using pure OLEDB and those using OGC extensions. The majority of this profile will address the first case as pure OLEDB will address most of the functional needs. Extensions will be detailed where they are appropriate.

6.1.1 Mandatory OLEDB Interfaces

The following OLEDB interfaces are mandatory for a data server to act as an OGC Catalog server.

Datasource:

IDBCreateSession	(OLEDB Mandatory)
IDBInitialize	(OLEDB Mandatory)
IDBProperties	(OLEDB Mandatory)
IDBAsyncStatus	(OLEDB optional)

Session:

IDBCreateCommand	(OLEDB optional)
IDBSchemaRowset	(OLEDB optional)

Commands:

IAccessor	(OLEDB Mandatory)
IColumnsInfo	(OLEDB Mandatory)
ICommand	(OLEDB Mandatory)
ICommandProperties	(OLEDB Mandatory)
ICommandText	(OLEDB Mandatory)

Rowsets:

IAccessor	(OLEDB Mandatory)
IColumnsInfo	(OLEDB Mandatory)
IRowset	(OLEDB Mandatory)
IRowsetView	(OLEDB optional)
IDBAsyncStatus	(OLEDB optional)

Views:

IColumnsInfo	(OLEDB Mandatory)
IAccessor	(OLEDB optional)
IViewRowset	(OLEDB optional)
IViewSort	(OLEDB optional)

6.1.2 OGC Extensions to OLEDB

OLEDB only supports HTML, text and binary formats. To support XML, DAG and SGML return formats the following flags have been defined for the dwFlag parameter of the DBBINDING entry in the Accessor.

DBBINDFLAG_XML -> 0x4

DBBINDFLAG_DAG -> 0x8

DBBINDFLAG_SGML -> 0x10

OLEDB only supports SQL dialects. Support for non-SQL query languages requires the addition of the DBPROP_OGCLANG property to Command objects. This property can take the following values:

OGC_Common -> 1

Z3950_TYPEONE -> 2

SQL3_SIMPLEFEATURE -> 3

SQL2_SIMPLEFEATURE -> 4

The DBPROP_OGCLANG property is set instead of the DBPROP_SQLSUPPORT property through the SetProperties interface on Command objects.

The AttributeCategory Parameter of the Explain Collection Request is not directly supported in the pure OLEDB environment. To support this parameter, the parameter ATTRIBUTECATEGORY will be added to the property set supported by the GetRowset interface.

6.2 Sequence Diagram

InitSessionRequest

Initializer::CreateDBInstance()

Datasource->IDBInitialize::Initialize()

Datasource->IDBProperties::SetProperties()

Datasource->IDBCreateSession::CreateSession()

InitSessionResponse

TerminateSessionRequest

Session->Release()

Datasource->Release()

TerminateResponse

ExplainServerRequest

Datasource->IDBProperties::GetProperties()

Datasource->IDBProperties::SetProperties()

Session->QueryInterface(IID_IDBSchemaRowset)

ExplainServerResponse

StatusRequest

	Rowset->IDBAsynchStatus::GetStatus()
StatusResponse	
CancelRequest	
	Rowset->IDBAsynchStatus::Abort()
CancelResponse	
QueryRequest	
	Session->IDBCreateCommand::CreateCommand()
	Command->ICommandProperties::SetProperties()
	Command->ICommandText::SetCommandText()
	Command->ICommand::Execute()
	Rowset->IRowsetView::CreateView()
	View->IColumnsInfo::GetColumnInfo()
	View->IViewSort::SetSort()
	View->IAccessor::CreateAccessor()
	View->IViewRowset::OpenViewRowset()
	Rowset->IRowset::GetNextRows()
	Rowset->IRowset::GetData()
QueryResponse	
PresentRequest	
	Rowset->IRowsetView::CreateView()
	View->IColumnsInfo::GetColumnInfo()
	View->IViewSort::SetSort()
	View->IAccessor::CreateAccessor()
	View->IViewRowset::OpenViewRowset()
	Rowset->IRowset::GetNextRows()
	Rowset->IRowset::GetData()
PresentResponse	
ExplainCollectionRequest	
	Session->IDBSchemaRowset::GetRowset()
	Rowset->IColumnsInfo::GetColumnInfo()
	Rowset->IAccessor::CreateAccessor()
	Rowset->IRowset::GetNextRows()
	Rowset->IRowset::GetData()
ExplainCollectionResponse	

6.3 Parameter Translation

This section addresses how catalog message parameter types defined in the General Model can be translated into and out of OLEDB equivalents.

6.3.1 CG_AttributeCategory

Type: Code_List

Used By: CG_ExplainCollectionRequest

CG_AttributeCategory is a code list for selecting the types of catalog entry attributes to be exposed by an explain collection request. These values are used by the client code to select the subset of the schema to return.

- Queriable
- Presentable
- Both

CG_AttributeCategory is supported through an extension to the GetRowset interface. ATTRIBUTECATEGORY is added to the property set supported by this interface. ATTRIBUTECATEGORY can take one of two bit values; queriable (0x01) and presentable (0x02). Both is the inclusive or of Queriable and presentable (0x03).

6.3.2 CG_BrokeredAccessRequestType

Type: Code_List

Used By: CG_BrokeredAccessRequest

Not currently mapped

6.3.3 CG_Status

Type: Code_List

Used By: CG_TerminateResponse, CG_StatusResponse, CG_CancelResponse, CG_PresentResponse, CG_BrokeredAccessResponse

CG_Status type variables are used to return status information to the general model. This is a direct mapping of the OLE DB HRESULT values in most cases. More detailed information will be provided with each message description.

6.3.4 CG_Capability

Type: Complex data structure

Used By: CG_ExplainServerRequest, CG_ExplainServerResponse

Uses: CG_AllSupportedRequest, CG_Defaults, CG_ExplainCollection, CG_Query, CG_Messaging, CG_Session, CG_SoftwareInformation, CG_SupportedCollections

CG_Capability is an aggregate of the following parameter types.

6.3.4.1 CG_AllSupportedRequest

Type: Boolean

Used By: CG_Capability

When this parameter is set within a capabilities structure all other capabilities will be ignored and the server will be queried for the all of the capabilities supported.

6.3.4.2 CG_Defaults

Type: Boolean

Used By: CG_Capability

When this parameter is set within a capabilities structure, all other capabilities will be ignored and the server will be queried for the default capabilities supported.

6.3.4.3 CG_ExplainCollection

Type: Boolean

Used By: CG_Capability

CG_ExplainCollection is supported in OLE DB by the IDBSchemaRowset interface. This parameter will be set to TRUE for servers that support that interface.

6.3.4.4 CG_Query

Type: data structure composed of version, characterSet and queryLanguage fields

Used By: CG_Capability

Uses: CG_QueryLanguage, CG_CharacterSet

The CG_Query capability structure can be populated from the DBPROP_SQLSUPPORT property of the Data Source object. This is a read only property that is read from the Data Source object through the IDBProperties interface. This interface only reports on the variations of SQL supported. The versions of SQL supported are:

DBPROPVAL_SQL_NONE – no SQL support

DBPROPVAL_SQL_ODBC_MINIMUM

DBPROPVAL_SQL_ODBC_CORE

DBPROPVAL_SQL_ODBC_EXTENDED – cumulative based on ODBC 2.5 definitions

DBPROPVAL_SQL_ESCAPECLAUSES – ODBC escape clause syntax supported

DBPROPVAL_SQL_ANSI92_ENTRY

DBPROPVAL_SQL_FIPS_TRANSITIONAL

DBPROPVAL_SQL_ANSI92_INTERMEDIATE

DBPROPVAL_SQL_ANSI92_FULL - cumulative based on ANSI SQL 92 definitions

DBPROPVAL_SQL_ANSI89_IEF – supports ANSI 89 Integrity Enhancement Facility

DBPROPVAL_SQL_SUBMINIMUM – uses SQL rules but less capable than ODBC minimum.

Support for non-SQL query languages requires the addition of the DBPROP_OGCLANG property to Dataset objects. This property can take the following values:

OGC_Common -> 1

Z3950_TYPEONE -> 2

SQL3_SIMPLEFEATURE -> 3

SQL2_SIMPLEFEATURE -> 4

The DBPROP_OGCLANG property is set instead of the DBPROP_SQLSUPPORT property through the SetProperties interface on Dataset objects.

The components of the CG_Query structure are populated as follows:

Version == derived from the DBPROP_SQLSUPPORT property.

CharacterSet == only UNICODE or ASCII is valid. Client code must know what it can support.

QueryLanguage == always SQL.

6.3.4.5 CG_Messaging

Type: Data structure

Used By: CG_Capability

Uses: CG_CharacterSet, CG_MessageFormat

OLE DB only supports binary, text and HTML formatting, UNICODE and ASCII character sets. To support XML, DAG and SGML return formats the following flags have been defined for the dwFlag parameter of the DBBINDING entry in the Accessor.

DBBINDFLAG_XML -> 0x4

DBBINDFLAG_DAG -> 0x8

DBBINDFLAG_SGML -> 0x10

6.3.4.6 CG_Session

Type: Data Structure

Used By: CG_Capability

Uses: CG_CharacterSet

This capability provides information that is specific to the Catalog service. These properties can be added to a server product but are not currently available.

Language == not available

CatalogSpecificationVersion == not currently available

CharacterSet == limited to UNICODE or ASCII

6.3.4.7 CG_SoftwareInformation

Type: Data structure

Used By: CG_Capability

This capability structure can be populated from two of the Data Source Information properties. These are read only properties that can be read from the Data Source object through the IDBProperties interface.

Vendor == DBPROP_DBMSNAME (the name of the server product)

VersionNumber == DBPROP_DBMSVER (the version of the server product)

6.3.4.8 CG_SupportedCollections

Type: set(CG_CollectionName)

Used By: CG_Capability

Uses: CG_CollectionName

The DBPROP_DATASOURCENAME property can be queried using the IDBProperties interface on the data source but cannot be set. Only the current catalog data set name can be returned at this point.

6.3.5 CG_QueryLanguage

Type: Code_List

Used By: CG_Query, CG_QueryExpression

The CG_QueryLanguage parameter type can be mapped into the DBPROP_SQLSUPPORT property of the Data Source object. This is a read only property that is read from the Data Source object through the IDBProperties interface. This interface only reports on the variations of SQL supported. The versions of SQL supported are:

DBPROPVAL_SQL_NONE – no SQL support

DBPROPVAL_SQL_ODBC_MINIMUM

DBPROPVAL_SQL_ODBC_CORE

DBPROPVAL_SQL_ODBC_EXTENDED – cumulative based on ODBC 2.5 definitions

DBPROPVAL_SQL_ESCAPECLAUSES – ODBC escape clause syntax supported

DBPROPVAL_SQL_ANSI92_ENTRY

DBPROPVAL_SQL_FIPS_TRANSITIONAL

DBPROPVAL_SQL_ANSI92_INTERMEDIATE

DBPROPVAL_SQL_ANSI92_FULL - cumulative based on ANSI SQL 92 definitions

DBPROPVAL_SQL_ANSI89_IEF – supports ANSI 89 Integrity Enhancement Facility

DBPROPVAL_SQL_SUBMINIMUM – uses SQL rules but less capable than ODBC minimum.

Support for non-SQL query languages requires the addition of the DBPROP_OGCLANG property to Command objects. This property can take the following values:

OGC_Common -> 1

Z3950_TYPEONE -> 2

SQL3_SIMPLEFEATURE -> 3

SQL2_SIMPLEFEATURE -> 4

The DBPROP_OGCLANG property is set instead of the DBPROP_SQLSUPPORT property through the SetProperties interface on Command objects.

6.3.6 CG_CatalogEntryType

Type: Code_List

Used By: CG_QueryRequest

There is no direct way to use this parameter in OLEDB. It may be passed as a command parameter to some servers or included in query text.

6.3.7 CG_CharacterSet

Type: Code_List

Used By: CG_Messaging, CG_Query, CG_Session

OLE DB only supports ASCII and UNICODE character sets. Specific providers may only support one or the other.

6.3.8 CG_CollectionName

Type: Union data

Used By: CG_QueryRequest, CG_QueryResponse, CG_ExplainCollectionRequest, CG_ExplainCollectionResponse, CG_BrokeredAccessResponse, CG_ReturnData

CG_CollectionName can be mapped into several types of OLEDB parameters based on the message and message parameter. Specific mapping details can be found in each message section.

6.3.9 CG_MessageFormat

Type: Code_List

Used By: CG_QueryRequest, CG_PresentRequest, CG_Messaging

OLE DB only supports binary, text and HTML formatting. This parameter is used to build accessors for retrieving data from a Rowset. To support XML, DAG and SGML return formats the following flags have been defined for the dwFlag parameter of the DBBINDING entry in the Accessor.

DBBINDFLAG_XML -> 0x4

DBBINDFLAG_DAG -> 0x8

DBBINDFLAG_SGML -> 0x10

6.3.10 CG_PredefinedPresentationType

Type: Code_List

Used By: CG_PresentationDescription

Named presentations are not directly supported by OLEDB.

6.3.11 CG_PresentationDescription

Type: Data Union

Used By: CG_QueryRequest, CG_QueryResponse

Uses: CG_PredefinedPresentationType, TupleType

The list of attribute names is used to build accessors for retrieving data from a Rowset. Named presentations are not directly supported by OLEDB.

6.3.12 CG_QueryExpression

Type: Data Structure

Used By: CG_QueryRequest

Uses: CG_QueryLanguage

CG_QueryExpression maps directly into two parameters used for building queries in OLE DB. Queries are built using the IcommandText::SetCommandText interface the parameters are:

dialect == which is similar to the theLanguage element of CG_QueryExpression

command == a pointer to a text string such as the theQuery element

6.3.13 CG_QueryScope

Type: Code_List

Used By: CG_QueryRequest

There is no OLE DB equivalent to this parameter at this time.

6.3.14 CG_RequestID

Type: Data Structure

Used By: CG_Message, CG_StatusRequest, CG_CancelRequest, CG_CancelResponse

CG_RequestID is mapped by the client software into a Rowset handle.

6.3.15 CG_ResultType

Type: Code_List

Used By: CG_QueryRequest, CG_QueryResponse

CG_ResultType is a code list describing the type of data to be returned in a query response message. These values are used by the OLEDB client code to select the interfaces to exercise.

- resultSet
- results
- validate
- hits

6.3.16 CG_ReturnData

Type: Data Union

Used By: CG_QueryResponse, CG_PresentResponse

Uses: CG_CollectionName, CG_CatalogEntry

Packaging of data into a CG_ReturnData format is performed by the Rowset::GetData() method. The format of the returned data is determined by the dwFlag parameter of the DBBINDING entry in the Accessor.

6.3.17 CG_SortField

Type: Data Structure

Used By: CG_QueryRequest, CG_PresentRequest

Uses: CG_SortOrder

CG_SortField parameters can be mapped directly into OLD DB data types with a little processing.

attributeName == map into column information structure (DBCOLUMNINFO). Retrieve the column information using IcolumnsInfo::GetColumnInfo, identify the proper attribute by comparing attributeName to the DBCOLUNINFO entry pwszName, get the ordinal for that column from the Iordinal entry. The ordinal values will be used to identify the sort attributes.

sortOrder == OLE DB type DBSORT

6.3.18 CG_SortOrder

Type: Code_List

Used By: CG_SortField

CG_SortOrder is similar to the OLE DB type DBSORT. DBSORT variables can indicate ascending or descending sorting only.

6.3.19 CG_UserInformation

Type: Data Structure

Used By: CG_BrokeredAccessRequest

Not yet mapped.

6.3.20 CG_PaymentMethod

Type: CodeList

Used By: CG_UserInformation

Not yet mapped.

6.3.21 TupleType

Type: Complex Data

Used By: CG_PresentationDescription

Maps into a data structure consisting of a key (character string) and a type (codelist).

6.3.22 Schema

Type: Complex Data

Used By: CG_SchemaID

Maps into an array of TupleType

6.3.23 CG_SchemaID

Type: Complex Data

Used By: CG_ExplainCollectionResponse

Uses: Schema, SchemaName

CG_SchemaID maps into a data structure consisting of:

schemaID ::= character string

schema ::= Schema

6.4 Detailed Implementation Guidance

6.4.1 Establish a catalog session

6.4.1.1 Request

CG_InitSessionRequest ::= sessionID destinationID requestID additionalInfo

sessionID ::= Integer

destinationID ::= CG_CollectionName

requestID ::= CG_RequestID

additionalInfo ::= String

6.4.1.2 Response

CG_InitSessionResponse ::= sessionID destinationID requestID additionalInfo diagnostic

sessionID ::= Integer

destinationID ::= CG_CollectionName

requestID ::= CG_RequestID

additionalInfo ::= XMLString

diagnostic ::= CharacterString

6.4.1.3 Pure OLEDB Processing

// Marshall the input parameters

sessionID == not used

destinationID == Name of data source, map to clsid through iterator, Data Links UI or Directory

requestID == not used

additionalInfo == not used

// Create a Data Source object

// clsid is an identifier for the data source. It can be found through the iterator, data links UI or Active Directory.

myInitialize->CreateDBInstance

(

clsid // clsid generated from destinationID

pUnkOuter // NULL

dwClsContext // CLSTX_INPROC_SERVER (in process server)

pwszReserved // NULL

riid // IID_IDBInitialize

&myDataSource // returned pointer Data source object

)

// Map myDataSource to destinationID. This mapping will be persistent for use in all further messages within this session.

// Initialize it

myData Source->IDBInitialize::Initialize()

// Set the properties, the following properties are recommended:

// DBPROP_ASYNCTXNABORT

// DBPROP_INIT_ASYNCH

// DBPROP_MULTIPLERESULTS

// DBPROP_MULTIPLESTORAGEOBJECTS

myData Source->IDBProperties::SetProperties

(

cPropertySets // Number of entries in rgPropertySets (4)

rgPropertySets // an array of DBPROPSET data structures

)

// Create a session

myDataSource->IDBCreateSession::CreateSession

(

pUnkOuter // NULL

riid // IID_IOpenRowset

mySession // Pointer to the session object

)

// Map mySession to sessionID. This mapping will be used for all further messages in this session.

// Marshal the output parameters

sessionID == map from mySession

destinationID == map from myDataSource

requestID == NULL

additionalInfo == NULL unless an error occurred

6.4.1.4 Relevant OLE DB Properties

DBPROP_ASYNCTXNABORT – (Data source) select whether transactions can be aborted asynchronously

DBPROP_INIT_ASYNCH – (Initialization) select asynchronous processing.

DBPROP_INIT_DATASOURCE – (Initialization) the name of the database to connect to.

DBPROP_INIT_LOCATION – (Initialization) the name of the catalog server.

DBPROP_MULTIPLERESULTS – (Data source) set the DBPROPVAL_MR_SUPPORTED and DBPROPVAL_MR_CONCURRENT flags to allow access to multiple result sets.

DBPROP_MULTIPLESTORAGEOBJECTS – (Data Source) set if access to more than one catalog at a time is supported

6.4.1.5 OGC OLEDB Extensions

none

6.4.2 End a Catalog Session

6.4.2.1 Request

CG_TerminateRequest ::= sessionID destinationID requestID additionalInfo

sessionID ::= Integer

destinationID ::= CG_CollectionName

requestID ::= CG_RequestID

additionalInfo ::= XMLString

6.4.2.2 Response

CG_TerminateResponse ::= sessionID destinationID requestID additionalInfo diagnostic status

sessionID ::= Integer

destinationID ::= CG_CollectionName

requestID ::= CG_RequestID

additionalInfo ::= XMLString

diagnostic ::= CharacterString

status ::= CG_Status

6.4.2.3 Pure OLEDB Processing

// Marshall the input parameters

sessionID == maps to the session handle “mySession”
destinationID == maps to the Data Source handle “myDataSource”
requestID == not used
additionalInfo == not used

// Terminate the session

mySession->Release()
myDataSource->Release()

// Marshall the output parameters

sessionID == mapped from mySession
destinationID == mapped from myDataSource
requestID == NULL
additionalInfo == NULL
diagnostic == NULL unless an error occurred
status == mapped from HRESULT

6.4.2.4 Relevant OLE DB Properties

None

6.4.2.5 OGC OLEDB Extensions

none

6.4.3 Query the server properties

6.4.3.1 Request

CG_ExplainServerRequest ::= sessionID destinationID requestID additionalInfo capabilities

sessionID ::= Integer
destinationID ::= CG_CollectionName
requestID ::= CG_RequestID
additionalInfo ::= XMLString

capabilities ::= Sequence<Capability>

6.4.3.2 Response

CG_ExplainServerResponse ::= sessionID destinationID requestID additionalInfo
diagnostic capabilities

sessionID ::= Integer

destinationID ::= CG_CollectionName

requestID ::= CG_RequestID

additionalInfo ::= XMLString

diagnostic ::= CharacterString

capabilities ::= Sequence<Capability>

6.4.3.3 Pure OLEDB Processing

// Marshall the input parameters

sessionID == maps to the session handle “mySession”
destinationID == maps to the Data Source handle “myDataSource”
requestID == not used
additionalInfo == not used
capabilities == mapping of capabilities to OLE DB properties is described in section ----

// If CG_AllSupportedRequest or CG_Default specified
// Get all or the properties of the Data Source

myDataSource->IDBProperties::GetProperties
(
cPropertyIDSets // number of entries in rgPropertyIDSets
rgPropertyIDSets // DBPROPIDSET array
pcPropertySets // number of entries returned in rgPropertySets
&rgPropertySets // Pointer to property set buffer
)

// Else set all writeable properties and read them back

myDataSource->IDBProperties::SetProperties
(
cPropertySets // Number of entries in property set buffer
&rgPropertySets // Pointer to property set buffer

```
        )
myDataSource->IDBProperties::GetProperties
        (
        cPropertyIDSets         // number of entries in rgPropertyIDSets
        rgPropertyIDSets        // DBPROPIDSET array
        pcPropertySets          // number of entries returned in rgPropertySets
        &rgPropertySets         // Pointer to property set buffer
        )

        // See if this server supports CG_ExplainCollection

mySession->QueryInterface
        (
        riid                            // IID_IDBSchemaRowset
        (void **)&mySchemaRowset        // pointer to Schema Rowset interface
        )

        // If mySchemaRowset == NULL, then set CG_ExplainCollection to FALSE
        // ELSE set CG_ExplainCollection to TRUE

        // Marshall the output parameters

sessionID == mapped from mySession
destinationID == mapped from myDataSource
requestID == copied from input parameter
additionalInfo == NULL
diagnostic == NULL unless an error occurred
capabilities == remap as described in section ---
```

6.4.3.4 Relevant OLE DB Properties

DBPROP_DATASOURCENAME – (Data source) the name of the data source

DBPROP_MAXSORTCOLUMNS – (View) maximum number of columns that can be supported in a sort.

DBPROP_SQLSUPPORT – (Data Source) specifies level of SQL support provided by server.

6.4.3.5 OGC OLEDB Extensions

None

6.4.4 Check the status of a request

6.4.4.1 Request

CG_StatusRequest ::= sessionID destinationID requestID additionalInfo requestIDtoStatus

sessionID ::= Integer

destinationID ::= CG_CollectionName

requestID ::= CG_RequestID

additionalInfo ::= XMLString

requestIDtoStatus ::= CG_RequestID

6.4.4.2 Response

CG_StatusResponse ::= sessionID destinationID requestID additionalInfo diagnostic status requestIDtoStatus

sessionID ::= Integer

destinationID ::= CG_CollectionName

requestID ::= CG_RequestID

additionalInfo ::= XMLString

diagnostic ::= CharacterString

status ::= CG_Status

requestIDtoStatus ::= CG_RequestID

6.4.4.3 Pure OLEDB Processing

```
// Marshall the input parameters

sessionID == maps to the session handle “mySession”
destinationID == maps to the Data Source handle “myDataSource”
requestID == not used
additionalInfo == not used
requestIDtoStatus == map into myCommand

// Request the status

myCommand->QueryInterface
(
riid                          // IID_IDBAsynchStatus
(void **)&myAsynchStatus      // pointer to asynch status interface
)
myAsynchStatus->GetStatus
(
HCHAPTER hChapter             // DB_NULL_HCHAPTER
ULONG ulOperation             // DBASYNCHOP_OPEN
```

ULONG * pulprogress // current progress toward completing this phase
ULONG * pulProgressMax // returned maximum value of pulprogress
ULONG * pulAsynchPhase // Phase – can be initializing, populating or complete
ULONG * ppwszStatusText // supporting text
)

// percentage complete is pulprogress / pulProgressMax
// Marshall the output parameters

sessionID == mapped from mySession
destinationID == mapped from myDataSource
requestID == copied from input parameter
additionalInfo == NULL
diagnostic == copy from ppwszStatusText
requestIDtoStatus == copied from input parameter
status == mapped from pulprogress, pulProgressMax and pulAsynchPhase

6.4.4.4 Relevant OLE DB Properties

DBPROP_CONNECTIONSTATUS – (Data source) gets the status of the catalog connection

6.4.4.5 OGC OLEDB Extensions

None

6.4.5 Cancel a request

6.4.5.1 Request

CG_CancelRequest ::= sessionID destinationID requestID additionalInfo
requestIDtoCancel freeResources
sessionID ::= Integer
destinationID ::= CG_CollectionName
requestID ::= CG_RequestID
additionalInfo ::= XMLString
requestIDtoCancel ::= CG_RequestID
freeResources ::= Boolean

6.4.5.2 Response

CG_CancelResponse ::= sessionID destinationID requestID additionalInfo diagnostic
Status canceledRequest
sessionID ::= Integer
destinationID ::= CG_CollectionName

requestID ::= CG_RequestID

additionalInfo ::= XMLString

diagnostic ::= CharacterString

status ::= CG_Status

canceledRequest ::= CG_RequestID

6.4.5.3 Pure OLEDB Processing

// Marshall the input parameters

sessionID == maps to the session handle “mySession”

destinationID == maps to the Data Source handle “myDataSource”

requestID == not used

additionalInfo == not used
requestIDtoCancel == map into myCommand

freeResources == not sure we can do this here

// Terminate the session

```
myCommand->QueryInterface
        (
        riid                        // IID_IDBAsynchStatus
        (void **)&myAsynchStatus    // pointer to asynch status interface
        )
myAsynchStatus->Abort
        (
        HCHAPTER hChapter           // DB_NULL_HCHAPTER
        ULONG ulOperation           // DBASYNCHOP_OPEN
        )
```

// Marshall the output parameters

sessionID == mapped from mySession

destinationID == mapped from myDataSource

requestID == copied from input parameter

additionalInfo == NULL

diagnostic == NULL

status == mapped from HRESULT

canceledRequest == copied from input parameter requestIDtoCancel

6.4.5.4 Relevant OLE DB Properties

DBPROP_ABORTPRESERVE – Rowset property to preserve or delete results after an abort

6.4.5.5 OGC OLEDB Extensions

None – freeResources not currently supported

6.4.6 Issue a Query

6.4.6.1 Request

CG_QueryRequest ::= sessionID destinationID requestID additionalInfo queryExpression resultType
iteratorSize cursor returnFormat presentation sortField queryScope
collectionID catalogType

sessionID ::= Integer

destinationID ::= CG_CollectionName

requestID ::= CG_RequestID

additionalInfo ::= XMLString

queryExpression ::= CG_QueryExpression

resultType ::= CG_ResultType

iteratorSize ::= Integer

cursor ::= Integer

returnFormat ::= CG_MessageFormat

presentation ::= CG_PresentationDescription

sortField ::= Sequence<sortField>

queryScope ::= CG_QueryScope

collectionID ::= CG_CollectionName

catalogType ::= CG_CatalogEntryType

6.4.6.2 Response

CG_QueryResponse ::= sessionID destinationID requestID additionalInfo diagnostic
retrievedData resultSetID resultType status hits cursor

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

diagnostic ::= CharacterString

retrievedData ::= CG_ReturnData

resultSetID ::= CG_CollectionName

resultType ::= CG_ResultType

status ::= CG_Status

hits ::= integer

cursor ::= Integer

6.4.6.3 Pure OLEDB Processing

// Marshall the input parameters

sessionID == maps to the session handle “mySession”

destinationID == maps to the Data Source handle “myDataSource”

requestID == not used

additionalInfo == not used

queryExpression

theQuery == local LPCOLSTR variable “string”

theLanguage == local REFGUID variable “dialect”

resultType == used by this client to control query processing

resultSet == only return the resultSet ID

results == Return result data

validate == Only confirm that the query was accepted

hits == Only return the size of the result set

iteratorSize == used directly by IRowSet::GetNextRows

cursor == used directly by IRowSet::GetNextRows

returnFormat == Used to generate the Accessor.

presentation == Used to generate the Accessor

sortField

attributeName == maps into the column ordinal for this attribute

sortOrder == OLEDB type = DBSORT which can indicate ascending or descending sorts only

queryScope == used to indicate distributed query. May be ignored or included as a command parameter.

collectionID == may be included in the query string as an SQL FROM clause or included as a command parameter

catalogType == may be passed as a command parameter or included in query text.

// Create a command object

```
mySession->IDBCreateCommand::CreateCommand
        (
         pUnkOuter                  // NULL
        riid                        // IID_ICommand
        (void **)&myCommand         // pointer to the command object
```

```
        )

        // Set the query language

myCommand->QueryInterface
        (
        riid                        // IID_ICommandProperties
        (void **)&myCommandProps    // pointer to command properties interface
        )
myCommandProps->SetProperties
        (
        cPropSets                   // Number of property sets (1)
        rgPropSets                  // the DBPROP_SQLSUPPORT property
         )

        // Insert the query text

myCommand->QueryInterface
        (
        riid                        // IID_ICommandText
        (void **)&myCommandText     // pointer to command text interface
        )
myCommandText->SetCommandText
        (
        DBGUID_SQL                  // allows use of the DBPROP_SQLSUPPORT property
        string                      // from queryExpression::theQuery
         )

        // Execute the command

myCommand->QueryInterface
        (
        riid                        // IID_ICommand
        (void **)&myCommandInterface // pointer to command interface
        )
myCommandInterface->Execute
        (
```

```
        NULL
        IID_IRowset
        NULL
        NULL
        (void **) &myRowset
        )

        // If resultType parameter is resultSet or validate then skip to marshalling
        // If resultType parameter is hits then -------
        // If resultType parameter is results then process the Rowset data

        // Create a view from the Rowset

myRowSet->IRowsetView::CreateView
        (
        pUnkOuter
        riid                    // IID_IView
        myView
        )

        // get the column information

myView->IColumnsInfo::GetColumnInfo
        (
        ULONG * pcColumns               // number of columns returned
        DBCOLUMNINFO * prgInfo          // array of column information
        OLECHAR ** ppStringsBuffer      // string data pointed to by prgInfo elements
        )

        // apply sorting

myView->IViewSort::SetSort
        (
        ULONG cColumns                  // Number of entries in rgColumns and rgOrders
        Const ULONG rgColumns[]         // column ordinals from prgInfo[].iOrdinal
        Const DBSORT rgOrders[]         // can be DBSORT_ASCENDING or
                                        //      DBSORT_DESCENDING
```

```
        )

        // Build an Accessor
        // The Accessor defines how the data returned by this query will be processed.  This is
        // where the presentation and returnFormat parameters come into play.  To build the Accessor,
        // traverse the list of attributes in the presentation parameter and add to the Accessor the
        // instructions for appending that attribute to the end of the retrievedData parameter.

myView->QueryInterface
        (
        riid                                    // IID_Iaccessor
        &myIAccessor                            // pointer to the Accessor interface
        )

        // For each attribute on the Presentation list, find the column
        // and build a new DBBINDING entry for the Accessor.  Key entries are:
        //    iOrdinal = ordinal defines the location of the attribute in the Rowset
        //    obValue = offset in retrievedData where the value for this attribute is to be stored
        //    dwFlag = set DBBINDFLAG_HTML if returnFormat is HTML
        //    wtType = data format of copied data.  If messageFormat is HTML or TXT, set to
                DBTYPE_STR for ASCII and DBTYPE_WSTR for UNICODE text output.

myIAccessor->CreateAccesor
        (
        DBACCESSORFLAGS flags       //DBACCESSOR_ROWDATA
        ULONG pcBindings            // number of entries in prBindings
        Const DBBINDING prBindings[]  // an array of DBBINDING structures, one for each attribute
        ULONG rowsize               // not used
        HACCESSOR * myAccessor      // returned handle of the Accessor
        DBBINDSTATUS rgstatus[]     // An array of status values, one for each rbindings entry
        )

        // Create a Rowset with the sorting applied

myView->QueryInterface
        (
        REFIID riid                 // IID_IViewRowset
```

```
        Iunknown ** &myIViewRowset    // pointer to the ViewRowset interface
        )
myIViewRowset->OpenViewRowset
        (
        IUnknown ** outer             // NULL
        REFIID Riid                   // IID_IRowSet
        &myRowset                     // Sorted Rowset
        )

        // retrieve the data

myRowset->GetNextRows
        (
        HCHAPTER chapter              // DB_NULL_HCHAPTER
        LONG cursor                   // from input parameter
        LONG iteratorSize             // from input parameter
        ULONG * rowsreceived          // number of rows actually returned
        HROW ** rowbuffer             // memory containing the row data
        )
myRowset->GetData
        (
        HROW rowbuffer                // memory containing the row data
        HACCESSOR myAccessor          // the data Accessor object
        Void * retrievedData.payload  // payload portion of the returned data parameter
        )

        // Marshall the output parameters

sessionID == mapped from mySession
destinationID == mapped from myDataSource
requestID == maps to myRowSet
additionalInfo == NULL
diagnostic == NULL unless an error occurred
retrievedData == populated by IRowset::GetData
resultSetID == maps to myRowSet
resultType == copied from input parameter
status == map from HRESULT values
```

hits == TBD

cursor == input parameter + rowsreceived from Irowset::GetNextRows

6.4.6.4 Relevant OLEDB Properties

DBPROP_ACCESSORDER – (Rowset) set to DBPROPVAL_AO_RANDOM to enable presentation specification.

DBPROP_CANFETCHBACKWARDS – (Rowset) Boolean to allow backup the cursor

DBPROP_CANSCROLLBACKWARDS _ (Rowset) Boolean to allow backward scrolling of the Rowset

DBPROP_ROWSET_ASYNCH – (Rowset) governs how the Rowset is generated – maps to result type

DBPROP_MAXROWS – (Rowset) maps to iterator size?

DBPROP_SERVERCURSOR (Rowset) sets the cursor location

6.4.6.5 OGC OLEDB Extensions

ReturnFormat: standard OLEDB only supports HTML, text and binary formats. To support XML, DAG and SGML formats the following flags have been defined for the dwFlag parameter of the DBBINDING entry in the Accessor.

DBBINDFLAG_XML

DBBINDFLAG_DAG

DBBINDFLAG_SGML

QueryExpression: OLEDB only supports SQL dialects. Support for non-SQL query languages requires the addition of the DBPROP_OGCLANG property. This property can take the following values:

OGC_Common -> 1

Z3950_TYPEONE -> 2

SQL3_SIMPLEFEATURE -> 3

SQL2_SIMPLEFEATURE -> 4

The DBPROP_OGCLANG property is set instead of the DBPROP_SQLSUPPORT property through the SetProperties interface on command objects.

queryScope == Add optional command parameter

collectionID == SQL FROM clause equivalent, add optional command parameter or include in query string.

catalogType == may be passed as a command parameter or included in query text.

6.4.7 Present Query Results

6.4.7.1 Request

CG_PresentRequest ::= sessionID destinationID requestID additionalInfo presentation sortField returnFormat iteratorSize cursor

sessionID ::= Integer

destinationID ::= CG_CollectionName

requestID ::= CG_RequestID

additionalInfo ::= XMLString

presentation ::= CG_PresentationDescription

sortField ::= Sequence<SortField>

returnFormat ::= CG_MessageFormat

iteratorSize ::= Integer

cursor ::= Integer

6.4.7.2 Response

CG_PresentResponse ::= sessionID destinationID requestID additionalInfo diagnostic retrievedData
cursor hits status

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

diagnostic ::= CharacterString

retrievedData ::= CG_ReturnData

cursor ::= Integer

hits ::= Integer

status ::= CG_Status

6.4.7.3 Pure OLEDB Processing

// Marshall the input parameters

sessionID == maps to the session handle “mySession”
destinationID == map to Rowset (myRowSet) created by previous query
requestID == not used
additionalInfo == not used
presentation == Used to generate the Accessor
sortField

attributeName == maps into the column ordinal for this attribute

sortOrder == OLEDB type = DBSORT which can indicate ascending or descending sorts only

returnFormat == Used to generate the Accessor.
iteratorSize == used directly by IRowSet::GetNextRows
cursor == used directly by IRowSet::GetNextRows

// Create a view from the Rowset

myRowSet->IRowsetView::CreateView

```
        (
        pUnkOuter
        riid                    // IID_IView
        myView
        )

        // get the column information

myView->IColumnsInfo::GetColumnInfo
        (
        ULONG * pcColumns               // number of columns returned
        DBCOLUMNINFO * prgInfo          // array of column information
        OLECHAR ** ppStringsBuffer      // string data pointed to by prgInfo elements
        )

        // apply sorting

myView->IViewSort::SetSort
        (
        ULONG cColumns                  // Number of entries in rgColumns and rgOrders
        Const ULONG rgColumns[]         // column ordinals from prgInfo[].iOrdinal
        Const DBSORT rgOrders[]         // can be DBSORT_ASCENDING or
                                        //      DBSORT_DESCENDING
        )

        // Build an Accessor
        // The Accessor defines how the data returned by this query will be processed.  This is
        // where the presentation and returnFormat parameters come into play.  To build the Accessor,
        // traverse the list of attributes in the presentation parameter and add to the Accessor the
        // instructions for appending that attribute to the end of the retrievedData parameter.

myView->QueryInterface
        (
        riid                            // IID_Iaccessor
        &myIAccessor                    // pointer to the Accessor interface
        )
```

```
        // For each attribute on the Presentation list, find the column
        // and build a new DBBINDING entry for the Accessor.  Key entries are:
        //    iOrdinal = ordinal defines the location of the attribute in the Rowset
        //    obValue = offset in retrievedData where the value for this attribute is to be stored
        //    dwFlag = set DBBINDFLAG_HTML if returnFormat is HTML
        //    wtType = data format of copied data.  If messageFormat is HTML or TXT, set to
                DBTYPE_STR for ASCII and DBTYPE_WSTR for UNICODE text output.

myIAccessor->CreateAccesor
        (
        DBACCESSORFLAGS flags        //DBACCESSOR_ROWDATA
        ULONG pcBindings             // number of entries in prBindings
        Const DBBINDING prBindings[] // an array of DBBINDING structures, one for each attribute
        ULONG rowsize                // not used
        HACCESSOR * myAccessor       // returned handle of the Accessor
        DBBINDSTATUS rgstatus[]      // An array of status values, one for each rbindings entry
        )

        // Create a Rowset with the sorting applied

myView->QueryInterface
        (
        REFIID riid                  // IID_IViewRowset
        Iunknown ** &myIViewRowset   // pointer to the ViewRowset interface
        )
myIViewRowset->OpenViewRowset
        (
        IUnknown ** outer            // NULL
        REFIID Riid                  // IID_IRowSet
        &myRowset                    // Sorted Rowset
        )

        // retrieve the data

myRowset->GetNextRows
        (
        HCHAPTER chapter             // DB_NULL_HCHAPTER
```

```
        LONG cursor                 // from input parameter
        LONG iteratorSize           // from input parameter
        ULONG * rowsreceived        // number of rows actually returned
        HROW ** rowbuffer           // memory containing the row data
        )
myRowset->GetData
        (
        HROW rowbuffer              // memory containing the row data
        HACCESSOR myAccessor        // the data Accessor object
        Void * retrievedData.payload // payload portion of the returned data parameter
        )

        // Marshall the output parameters
```

sessionID == mapped from mySession

destinationID == mapped from myRowset

requestID == mapped from myRowSet

additionalInfo == NULL

diagnostic == NULL unless an error occurred

retrievedData == populated by IRowset::GetData

cursor == input parameter + rowsreceived from Irowset::GetNextRows

hits == rowsreceived
status == map from HRESULT values

6.4.7.4 Relevant OLEDB Properties

DBPROP_ACCESSORDER – (Rowset) set to DBPROPVAL_AO_RANDOM to enable presentation specification.

DBPROP_CANFETCHBACKWARDS – (Rowset) Boolean to allow backup the cursor

DBPROP_CANSCROLLBACKWARDS _ (Rowset) Boolean to allow backward scrolling of the Rowset

DBPROP_ROWSET_ASYNCH – (Rowset) governs how the Rowset is generated – maps to result type

DBPROP_MAXROWS – (Rowset) maps to iterator size?

DBPROP_SERVERCURSOR (Rowset) sets the cursor location

6.4.7.5 OGC OLEDB Extensions

ReturnFormat: standard OLEDB only supports HTML, text and binary formats. To support XML, DAG and SGML formats the following flags have been defined for the dwFlag parameter of the DBBINDING entry in the Accessor.

DBBINDFLAG_XML

DBBINDFLAG_DAG

DBBINDFLAG_SGML

6.4.8 Get the schema

6.4.8.1 Request

CG_ExplainCollectionRequest ::= sessionID destinationID requestID additionalInfo attributeCategory collectionID

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

attributeCategory ::= CG_AttributeCategory

collectionID ::= CG_CollectionName

6.4.8.2 Response

CG_ExplainCollectionResponse ::= sessionID destinationID requestID additionalInfo diagnostic collectionID dataModel

sessionID ::= Integer

destinationID ::= CharacterString

requestID ::= CG_RequestID

additionalInfo ::= CharacterString

diagnostic ::= CharacterString

collectionID ::= CG_CollectionName

dataModel ::= CG_SchemaID

6.4.8.3 Pure OLEDB Processing

// Marshall the input parameters

sessionID == maps to the session handle “mySession”

destinationID == maps to the Data Source handle “myDataSource”

requestID == not used

additionalInfo == not used

attributeCategory == See extensions

collectionID == not yet used

// a local data item to hold schema data

Schemadata == an array of structure

Schema_name – character string

Table – character string

```
        Column_name – character string
        Ordinal – integer
        Data_type – code list (see OLEDB Programmer's Reference Appendix A)

        // Get the COLUMNS table from the schema Rowsets

mySession->QueryInterface
        (
        riid                            // IID_IDBSchemaRowset
        (void **)&myISchemaRowset       // pointer to Schema Rowset interface
        )
myISchemaRowset->GetRowset
        (
        Iunknown * punkOuter            // NULL
        REFGUID rguidschema             // DBSCHEM_COLUMNS
        ULONG crestrictions             // 0
        Const VARIANT rgrestrictions[]  // NULL
        REFIID riid                     // IID_IRowSet
        ULONG cpropertysets             // 0
        DBPROPSET rgpropertysets[]      // NULL
        Iunknown ** myRowSet            // pointer to the schema Rowset
        )

        // get the column information for this Rowset

myRowSet->IColumnsInfo::GetColumnInfo
        (
        ULONG * pcColumns                   // number of columns returned
        DBCOLUMNINFO * prgInfo              // array of column information
        OLECHAR ** ppStringsBuffer          // string data pointed to by prgInfo elements
        )

        // create an Accessor collecting the schema name, table, column name, ordinal and data type
        //      Table == TABLE_NAME
        //      Schema_name == TABLE_SCHEMA
        //      Column_name = COLUMN_NAME
        //      Data_type == DATA_TYPE
```

```
        //          Ordinal == ORDINAL_POSITION

myRowSet->QueryInterface
        (
        riid                                    // IID_Iaccessor
        &myIAccessor                            // pointer to the Accessor interface
        )
myIAccessor->CreateAccesor
        (
        DBACCESSORFLAGS flags       //DBACCESSOR_ROWDATA
        ULONG pcBindings            // number of entries in prBindings
        Const DBBINDING prBindings[]  // an array of DBBINDING structures, one for each attribute
        ULONG rowsize               // not used
        HACCESSOR * myAccessor      // returned handle of the Accessor
        DBBINDSTATUS rgstatus[]     // An array of status values, one for each rbindings entry
        )

        // get the data from each COLUMNS Rowset

myRowSet->GetNextRows
        (
        HCHAPTER chapter            // DB_NULL_HCHAPTER
        LONG cursor                 // 0
        LONG iteratorSize           // number of rows that rowbuffer can hold
        ULONG * rowsreceived        // number of rows actually returned
        HROW ** rowbuffer           // memory containing the row data
        )
myRowset->GetData
        (
        HROW rowbuffer              // memory containing the row data
        HACCESSOR myAccessor        // the data Accessor object
        Void * Schemadata           // temporary storage for schema data
        )

        // Marshall the output parameters

sessionID == mapped from mySession
```

destinationID == mapped from myDataSource

requestID == mapped from myRowSet
additionalInfo == NULL

diagnostic == NULL unless an error occurred

collectionID == copy from schemadata.table

dataModel == composed of

schemaName == copy from schemadata.schema_name

schema == composed of

key == copy from schemadata.column_name

type == map from schemadata.data_type

6.4.8.4 Relevant OLEDB Properties

DBPROP_COL_DEFAULT – (column) VARIANT specifying the default value for the column

DBPROP_COL_DESCRIPTION – (column) Human readable description of the column

6.4.8.5 OGC OLEDB Extensions

AttributeCategory Parameter: This parameter is not supported by the pure OLEDB environment. To support this parameter, the parameter ATTRIBUTECATEGORY will be added to the property set supported by the GetRowset interface.

7. WWW Profile

7.1 Architecture

The WWW Profile uses a message-based client server architecture. The profile maps each of the general model operations to a corresponding service specified in the ANSI/NISO Z39.50 Application Service Definition and Protocol Specification [*ISO 23950*] [*http://lcweb.loc.gov/z3950/agency/document.html*]. For conformance, clients and servers must support Z39.50 Version 3.

The WWW Profile specifies the use of the following transport mechanisms:

- HyperText Transport Protocol (HTTP) where services are encoded in XML using the XML Encoding Rules (XER) [*http://asf.gils.net/xer*].
- Directly over TCP where services are encoded using the Basic Encoding Rules (BER) [*ISO 8825*].

7.1.1 Supported Services

Each operation specified in this profile corresponds to a Z39.50 Service, and consists of a client request message followed by a server response message. The Z39.50 Services used in this profile include the Init, Search, Present, Resource Control, Trigger Resource Control, Sort, Extended Services and Close.

7.1.2 Transport (HTTP)

The client transmits request messages to the server and the server returns responses to the client over HTTP version 1.0 or 1.1. A logical session is maintained between the client and server using state management as specified in *IETF RFC 2109: HTTP State Management Mechanism* [*http://www.w3.org/Protocols/rfc2109/rfc2109*], where the SessionID is maintained in a cookie named "XERSessionId".

Request messages are transmitted using the HTTP POST method. As other HTTP methods become widely available, other HTTP methods may be used (such as the HTTP SEARCH method). The content of the HTTP method contains the request message, and the content of the HTTP response contains the response message. In both cases, the message content is encoded in XML and the Content-Type is application/x-xer-z3950. Once the Content Type is registered, the Content Type will become application/xer-z3950.

7.1.3 Transport (TCP)

The client transmits request messages to the server and the server returns response messages to the client directly over TCP as specified in *IETF RFC 1729: Using the Z39.50 Information Retrieval Protocol in the Internet Environment* [*ftp://ftp.ietf.org/rfc/rfc1729.txt*], where all request and response messages are encoded using BER.

7.2 Sequence Diagrams

Table 12 provides a mapping between general model operations and the WWW Profile services. The WWW profile messages are defined in Section 7.4.

Table 12 - General Model to WWW Profile Message Mapping

General Model Operation	WWW Profile Service
CG_InitSessionRequest	initRequest[1]
CG_InitSessionResponse	InitResponse[1]
CG_TerminateRequest	close[2]
CG_TerminateResponse	close
CG_ExplainServerRequest	searchRequest[3, 4]
CG_ExplainServerResponse	searchResponse
CG_StatusRequest	triggerResourceControlRequest
CG_StatusResponse	resourceControlRequest
CG_CancelRequest	triggerResourceControlRequest
CG_CancelResponse	none[5]
CG_QueryRequest	searchRequest[3,6] and sortRequest
CG_QueryResponse	searchResponse and sortResponse
CG_PresentRequest	presentRequest
CG_PresentResponse	presentResponse
CG_ExplainCollectionRequest	searchRequest[7]
CG_ExplainCollectionResponse	searchResponse[7]
CG_BrokeredAccessRequest	extendedServicesRequest[8]
CG_BrokeredAccessResponse	extendedServicesResponse[8]

[1] The following init Options are used in this profile: search, present, sort, extended-services, trigger-resource-control, named result sets, and resource-control.

[2] Although Z39.50 permits both the client and server to initiate a Close request, for conformance with the general model, only the client is permitted to initiate a Close request. In practice, a server may terminate a session after a reasonable amount of idle client activity.

[3] Note that the CG_ResultType values of results and hits are supported in this profile. The CG_ResultType values of result set ID and validate are unsupported.

[4] The CG_ExplainServerRequest is implemented using a searchRequest on the Explain Database with ExplainCategory = TargetInfo and DatabaseInfo.

[5] For HTTP transport, a message with no content is returned.

[6] The CG_CatalogEntryType and CG_QueryScope parameters in the CG_QueryRequest are implemented in the WWW Profile as external elements of the SearchRequest. The externals are defined in Section 7.5.1.

[7] The CG_ExplainCollectionRequest is implemented using a searchRequest on the Explain Database with ExplainCategory = TargetInfo and RetrievalRecordDetails.

[8] Brokered Access is implemented in the WWW Profile using the Order Extended Service defined in Section 7.5.2. The Order Extended Service uses the Z39.50 Extended Service mechanism.

7.3 Example Sequence Diagram

The following sequence diagram illustrates a typical set of transactions that may occur between a client and server, and between the server and its interface to an external catalog system. The client sends an

initRequest message to the server, the external system processes the initRequest message by initializing a session with the client and the server returns an initResponse message to the client. This interaction establishes a session in which all subsequent interactions occur.

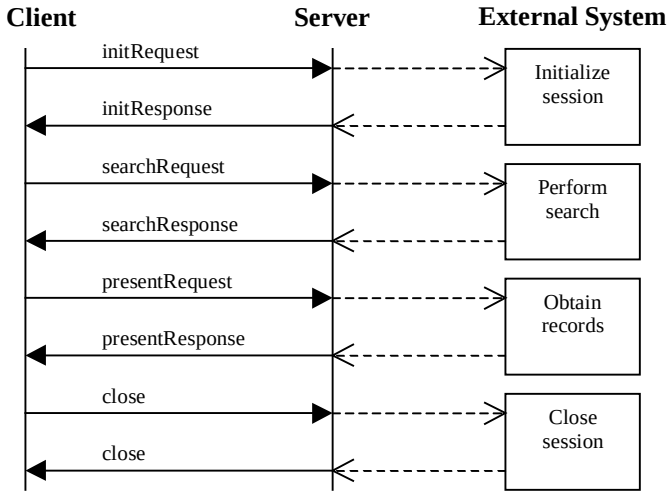


Figure 27 - WWW Profile Sequence Diagram

Next the client constructs a query and sends the query in the searchRequest message to the server. The server runs the search on the external catalog system, and returns the requested results in the searchResponse message. If the search was successful, a virtual result set is created and the client may request records from the result set using the presentRequest message. In the presentRequest, the client may request any contiguous set of records from the result set (e.g., records 10 through 20). The server returns the records to the client in the presentResponse message. The client may continue to perform additional searches and record retrievals, or may close the session with the server by sending a close message. Optionally, the server may respond with a close message.

7.4 Interface Definition – XML

For HTTP transport the XML messages are defined by the XML encoding rules. The specification for the XML encoding rules can be found at http://asf.gils.net/xer . This specification derives the encoding of the Application Protocol Data Units (APDUs) from the ASN.1 specification of Z39.50 available from http://lcweb.loc.gov/z39.50/agency/document.html .

For information a DTD for Z39.50 encoded using XER is given below.

```
<!-- The ISO23950 namespace is the specification in ASN.1
     maintained at "http://lcweb.loc.gov/z3950/agency/asn1.html" -->

<!ELEMENT Search (
    initRequest |
    initResponse |
    searchRequest |
    searchResponse |
    presentRequest |
    presentResponse |
    resourceControlRequest |
    resourceControlResponse |
    sortRequest |
    sortResponse |
    extendedServicesRequest |
    extendedServicesResponse |
    close
    )>

<!-- Initialization service definitions -->
<!ELEMENT initRequest (
    referenceId?,
    protocolVersion,
    options,
    preferredMessageSize,
    exceptionalRecordSize,
    idAuthentication?,
    implementationId?,
    implementationName?,
    implementationVersion?,
    userInformationField?,
    otherInfo?
    )>
<!ELEMENT initResponse (
    referenceId?,
    protocolVersion,
    options,
    preferredMessageSize,
    exceptionalRecordSize,
    result,
    implementationId?,
    implementationName?,
    implementationVersion?,
    userInformationField?,
    otherInfo?
    )>

<!-- Search service definitions -->
<!ELEMENT searchRequest (
    referenceId?,
    smallSetUpperBound,
    largeSetLowerBound,
```

```
    mediumSetPresentNumber,
    replaceIndicator,
    resultSetName,
    databaseNames,
    smallSetElementSetNames?,
    mediumSetElementSetNames?,
    preferredRecordSyntax?,
    query,
    additionalSearchInfo?,
    otherInfo?
    )>
<!ELEMENT searchResponse (
    referenceId?,
    resultCount,
    numberOfRecordsReturned,
    nextResultSetPosition,
    searchStatus,
    resultSetStatus?,
    presentStatus?,
    records?,
    additionalSearchInfo?,
    otherInfo?
    )>

<!-- Present service definitions -->
<!ELEMENT presentRequest (
    referenceId?,
    resultSetId,
    resultSetStartPoint,
    numberOfRecordsRequested,
    recordComposition?,
    preferredRecordSyntax?,
    otherInfo?
    )>
<!ELEMENT presentResponse (
    referenceId?,
    numberOfRecordsReturned,
    nextResultSetPosition,
    presentStatus,
    records?,
    otherInfo?
    )>

<!-- Resource control service definition -->
<!ELEMENT resourceControlRequest (
    referenceId?,
    suspendedFlag?,
    resourceReport?,
    partialResultsAvailable?,
    responseRequired,
    triggeredRequestFlag?,
    otherInfo?
    )>
<!ELEMENT resourceControlResponse (
    referenceId?,
    continueFlag,
    resultSetWanted?,
    otherInfo?
    )>

<!-- Close service definition -->
<!ELEMENT close (
    referenceId?,
```

```
    closeReason,
    diagnosticInformation?,
    resourceReportFormat?,
    resourceReport?,
    otherInfo?
    )>

<!-- Sort service definition -->
<!ELEMENT sortRequest (
    referenceId?,
    inputResultSetNames,
    sortedResultSetName,
    sortSequence,
    otherInfo?
    )>
<!ELEMENT sortResponse (
    referenceId?,
    sortStatus,
    resultSetStatus?,
    diagnostics?,
    otherInfo?
    )>

<!-- extendedServices service definition -->
<!ELEMENT extendedServicesRequest (
    referenceId?,
    function,
    packageType,
    packageName?,
    userId?,
    retentionTime?,
    permissions?,
    description?,
    taskSpecificParameters?,
    waitAction,
    elements?,
    otherInfo?
    )>

<!ELEMENT extendedServicesResponse (
    referenceId?,
    operationStatus,
    diagnostics?,
    taskPackage?,
    otherInfo?
    )>


<!-- Auxiliary initialization service definitions -->
<!ELEMENT protocolVersion (#PCDATA)> <!-- values: version-1 version-2
                                          version-3 -->
<!ELEMENT options (#PCDATA)>         <!-- values: search present delSet
                                          triggerResourceCtrl resourceCtrl sort
                                          extendedServices namedResultSets -->
<!ELEMENT preferredMessageSize (#PCDATA)>   <!-- integer -->
<!ELEMENT exceptionalRecordSize (#PCDATA)>  <!-- integer -->
<!ELEMENT result (#PCDATA)>                 <!-- values: true | false -->
<!ELEMENT implementationId (#PCDATA)>       <!-- general string -->
<!ELEMENT implementationName (#PCDATA)>     <!-- general string -->
<!ELEMENT implementationVersion (#PCDATA)>  <!-- general string -->
<!ELEMENT userInformationField (External)>

<!-- Auxiliary search service definitions -->
```

```
<!ELEMENT smallSetUpperBound (#PCDATA)>     <!-- integer -->
<!ELEMENT largeSetLowerBound (#PCDATA)>     <!-- integer -->
<!ELEMENT mediumSetPresentNumber (#PCDATA)> <!-- integer -->
<!ELEMENT replaceIndicator (#PCDATA)>       <!-- values: true | false -->
<!ELEMENT resultSetName (#PCDATA)>          <!-- general string -->
<!ELEMENT smallSetElementSetNames (genericElementSetName | databaseSpecific)>
<!ELEMENT mediumSetElementSetNames (genericElementSetName | databaseSpecific)>
<!ELEMENT preferredRecordSyntax (#PCDATA)>  <!-- object identifier -->
<!ELEMENT additionalSearchInfo (#PCDATA)>   <!-- subelement omitted -->
<!ELEMENT resultCount (#PCDATA)>            <!-- integer -->
<!ELEMENT searchStatus (#PCDATA)>           <!-- values: true | false -->

<!-- Query definition -->
<!ELEMENT query (type-0 | type-1 | type-2 | type-100 | type-101 | type-102)>
<!ELEMENT type-0 (#PCDATA)>                 <!-- any -->
<!ELEMENT type-1 (attributeSet, rpn)>       <!-- RPN query -->
<!ELEMENT type-2 (#PCDATA)>                 <!-- octet string -->
<!ELEMENT type-100 (#PCDATA)>               <!-- octet string -->
<!ELEMENT type-101 (attributeSet, rpn)>     <!-- RPN query -->
<!ELEMENT type-102 (#PCDATA)>               <!-- octet string -->

<!-- Query operand definitions -->
<!ELEMENT rpn (op | rpnRpnOp)>              <!-- op is Operator -->
<!ELEMENT rpn1 (op | rpnRpnOp)>             <!-- op is Operator -->
<!ELEMENT rpn2 (op | rpnRpnOp)>             <!-- op is Operator -->
<!ELEMENT rpnRpnOp (rpn1,
                    rpn2,
                    op)>                    <!-- op is Operand -->
<!ELEMENT op (
    (attrTerm | resultSet | resultAttr ) |
    (and | or | and-not)
    )>                                      <!-- op is Operand & Operator -->
<!ELEMENT attrTerm (attributes, term)>      <!-- AttributesPlusTerm -->
<!ELEMENT resultSet (#PCDATA)>              <!-- general string -->
<!ELEMENT resultAttr (#PCDATA)>             <!-- subelements omitted -->
<!ELEMENT numeric (#PCDATA)>                <!-- integer -->
<!ELEMENT string (#PCDATA)>                 <!-- general string -->
<!ELEMENT general (#PCDATA)>                <!-- octet string -->
<!ELEMENT complex (#PCDATA)>                <!-- subelements omitted -->

<!-- Query operator definitions -->
<!ELEMENT and EMPTY>                        <!-- null -->
<!ELEMENT or EMPTY>                         <!-- null -->
<!ELEMENT and-not EMPTY>                    <!-- null -->

<!-- Auxiliary present service definitions -->
<!ELEMENT resultSetStartPoint (#PCDATA)>       <!-- integer -->
<!ELEMENT numberOfRecordsRequested (#PCDATA)>  <!-- integer -->
<!ELEMENT recordComposition (simple)>          <!-- complex omitted -->
<!ELEMENT simple (genericElementSetName | databaseSpecific)>

<!-- Auxiliary search and present service definitions -->
<!ELEMENT numberOfRecordsReturned (#PCDATA)>   <!-- integer -->
<!ELEMENT nextResultSetPosition (#PCDATA)>     <!-- integer -->
<!ELEMENT presentStatus (#PCDATA)>             <!-- values: success |
                                                    partial-1 | partial-2 |
                                                    partial-3 | partial-4 |
                                                    failure -->

<!-- Record and diagnostic definitions -->
<!ELEMENT records (
    responseRecords |
    nonSurrogateDiagnostic |
```

```
    mutipleNonSurDiagnostics
    )>
<!ELEMENT responseRecords (Item*)>          <!-- sequence of NamePlusRecord -->
<!ELEMENT name (#PCDATA)>                   <!-- general string -->
<!ELEMENT record (retrievalRecord | surrogateDiagnostic)>
<!ELEMENT retrievalRecord (External)>
<!ELEMENT surrogateDiagnostic (defaultFormat | externallyDefined)>
<!ELEMENT nonSurrogateDiagnostic (diagnosticSetId | condition | addinfo)>
<!ELEMENT mutipleNonSurDiagnostics (Item*)>

<!-- Auxiliary resource control definitions -->
<!ELEMENT suspendedFlag (#PCDATA)>           <!-- values: true | false -->
<!ELEMENT partialResultsAvailable (#PCDATA)> <!-- values: subset | interim
                                                  | none -->
<!ELEMENT responseRequired (#PCDATA)>        <!-- values: true | false -->
<!ELEMENT triggeredRequestFlag (#PCDATA)>    <!-- values: true | false -->
<!ELEMENT continueFlag (#PCDATA)>            <!-- values: true | false -->
<!ELEMENT resultSetWanted (#PCDATA)>         <!-- values: true | false -->

<!-- Auxiliary close service definitions -->
<!ELEMENT closeReason (#PCDATA)>            <!-- values: finished | shutdown |
                                                systemProblem |  costLimit |
                                                resources | securityViolation |
                                                protocolError | lackOfActivity|
                                                peerAbort | unspecified -->
<!ELEMENT diagnosticInformation (#PCDATA)>  <!-- general string -->
<!ELEMENT resourceReportFormat (#PCDATA)>   <!-- object identifier -->

<!-- Auxiliary sort definitions -->
<!ELEMENT inputResultSetNames (Item*)>      <!-- sequence of general string -->
<!ELEMENT sortedResultSetName (#PCDATA)>    <!-- general string -->
<!ELEMENT sortSequence (Item*)>             <!-- SeqOf SortRequ.sortSeq -->
<!ELEMENT sortElement (generic | databaseSpecific)>
<!ELEMENT sortRelation (#PCDATA)>           <!-- values: ascending | descending
                                                 ascendingByFrequency |
                                                 descendingByFrequency -->
<!ELEMENT caseSensitivity (#PCDATA)>        <!-- values: caseSensitive |
                                                 caseInsensitive -->
<!ELEMENT missingValueAction ((abort | null), missingValueData)>
<!ELEMENT abort EMPTY>                      <!-- null -->
<!ELEMENT null EMPTY>                       <!-- null -->
<!ELEMENT missingValueData (#PCDATA)>       <!-- octet string -->
<!ELEMENT generic (sortfield | elementSpec | sortAttributes)>
<!ELEMENT databaseName (#PCDATA)>           <!-- general string -->
<!ELEMENT dbSort (sortfield | elementSpec | sortAttributes)>
<!ELEMENT sortfield (#PCDATA)>              <!-- general string -->
<!ELEMENT elementSpec (
    (schema?, elementSpec?) |
    (elementSetName | externalSpec)
    )>                                      <!-- SortKey, Specification -->
<!ELEMENT schema (#PCDATA)>                 <!-- object identifier -->
<!ELEMENT elementSetName (#PCDATA)>         <!-- general string -->
<!ELEMENT externalSpec (External)>
<!ELEMENT sortAttributes (id, list)>
<!ELEMENT id (#PCDATA)>                     <!-- object identifier -->
<!ELEMENT list (Item*)>                     <!-- SeqOf AttributeElement -->
<!ELEMENT sortStatus (#PCDATA)>             <!-- values: success | partial-1 |
                                                 failure -->


<!-- Auxiliary extendedServices definitions -->
<!ELEMENT function (#PCDATA)>               <!-- values: create | delete |
                                                 modify -->
```

```
<!ELEMENT packageType (#PCDATA)>           <!-- object identifier -->
<!ELEMENT packageName (#PCDATA)>           <!-- general string -->
<!ELEMENT retentionTime (value | unitUsed)>
<!ELEMENT permissions (Item*)>
<!ELEMENT allowableFunctions (Item*)>      <!-- values: delete |
                                                modifyContents |
                                                modifyPermissions | present |
                                                invoke -->
<!ELEMENT description (#PCDATA)>
<!ELEMENT taskSpecificParameters (External)>
<!ELEMENT waitAction (#PCDATA)>            <!-- values: wait | waitIfPossible|
                                              dontWait | dontReturnPackage -->
<!ELEMENT elements (#PCDATA)>              <!-- general string -->
<!ELEMENT operationStatus (#PCDATA)>       <!-- values: done | accepted |
                                                failure -->
<!ELEMENT taskPackage (External)>

<!ELEMENT referenceId (#PCDATA)>           <!-- octet string -->
<!ELEMENT resultSetId (#PCDATA)>           <!-- general string -->
<!ELEMENT diagnostics (Item*)>             <!-- sequence of DiagRec -->
<!ELEMENT databaseNames (Item*)>
<!ELEMENT otherInfo (#PCDATA)>             <!-- subelement omitted -->
<!ELEMENT resourceReport (External)>
<!ELEMENT resultSetStatus (#PCDATA)>       <!-- values: empty | subset |
                                                interim | unchanged | none -->

<!-- Definition of additional components -->

<!-- Authentication (initRequest) -->
<!ELEMENT idAuthentication (open | idPass | anonymous | other)>
<!ELEMENT open (#PCDATA)>                  <!-- visible string -->
<!ELEMENT idPass (groupId?, userId?, password?)>
<!ELEMENT groupId (#PCDATA)>               <!-- general string -->
<!ELEMENT userId (#PCDATA)>                <!-- general string -->
<!ELEMENT password (#PCDATA)>              <!-- general string -->
<!ELEMENT anonymous EMPTY>                 <!-- null -->
<!ELEMENT other (External)>

<!-- AttributesPlusTerm -->
<!ELEMENT attributes (Item*)>              <!-- SeqOf AttributeElement -->
<!ELEMENT term (#PCDATA)>                  <!-- data types omitted -->

<!-- AttributeElement -->
<!ELEMENT attributeSet (#PCDATA)>          <!-- object identifier -->
<!ELEMENT attributeType (#PCDATA)>         <!-- integer -->
<!ELEMENT attributeValue (numeric | complex)>

<!-- ElementSetNames -->
<!ELEMENT genericElementSetName (#PCDATA)>  <!-- general string -->
<!ELEMENT databaseSpecific (Item*)>        <!-- sequence of (dbName, esn) -->
<!ELEMENT dbName (#PCDATA)>                <!-- general string -->
<!ELEMENT esn (#PCDATA)>                   <!-- general string -->

<!-- DiagRec -->
<!ELEMENT defaultFormat     (diagnosticSetId | condition | addinfo)>
<!ELEMENT diagnosticSetId   (#PCDATA)>     <!-- object identifier -->
<!ELEMENT condition         (#PCDATA)>     <!-- integer -->
<!ELEMENT addinfo           (v2AddInfo | v3AddInfo)>
<!ELEMENT v2AddInfo         (#PCDATA)>     <!-- visible string -->
<!ELEMENT v3AddInfo         (#PCDATA)>     <!-- general string -->
<!ELEMENT externallyDefined (External)>

<!-- IntUnit -->
```

```
<!ELEMENT value (#PCDATA)>                  <!-- integer -->
<!ELEMENT unitUsed (unitSystem | unitType | unit | scaleFactor)>
<!ELEMENT unitSystem (#PCDATA)>             <!-- general string -->
<!ELEMENT unitType (string | numeric)>
<!ELEMENT unit (string | numeric)>
<!ELEMENT scaleFactor (#PCDATA)>            <!-- integer -->

<!-- Elements added by the XER specification -->
<!-- the Item tag is used for many things:
AttributesPlusTerm                      (attributes, term)
AttributeList                           (attributeSet?, attributeType,
                                         attributeValue)
Records.responseRecords                 (name, record)
DiagRec                                 (defaultFormat | externallyDefined)
SortRequest.sortSequence                (sortElement, sortRelation,
                                         caseSensitivity, missingValueAction?)
SortElement.databaseSpecific            (databaseName, dbSort)
ElemenSetnames.databaseSpecific         (dbName, esn)
Permissions                             (userId, allowableFunctions)>
InternationalString                     (#PCDATA)
-->
<!ELEMENT Item (#PCDATA)>

<!-- Global auxiliary definitions -->
<!ELEMENT External (direct-reference, encoding)>
<!ELEMENT direct-reference (#PCDATA)>
<!ELEMENT encoding (single-ASN1-type | octet-aligned)>
<!ELEMENT single-ASN1-type (#PCDATA)>
<!ELEMENT octet-aligned   (#PCDATA)>
```

7.5 Definition of Externals

7.5.1 Additional Search Info

This section contains the parameters used in the "otherInfo" part of a Z39.50 searchRequest in order to implement the CG_CatalogEntryType and CG_QueryScope parameters in the CG_QueryRequest of the General Model.

"otherInfo" in a SearchRequest may be used by the origin to specify the scope of a search, i. e. whether the search domain is wide or restricted to a local search. This is achieved using the SearchControl EXTERNAL in otherInfo. SearchControl is defined below using ASN.1 notation. If otherInfo is not provided, the type of item descriptors to be searched shall be derived from the query definition and/ or the content of the collection and the default scope of a local search shall be assumed.

The Search Control structure contains two items: itemDescriptorType which maps to CG_CatalogEntryType and searchScope which maps to CG_QueryScope. The CIP-Release-B-APDU {Z39.50-CIP-B-APDU 1} defines the following items:

```
SearchControl ::= SEQUENCE
     {
          itemDescriptorType [1] IMPLICIT INTEGER
               {
                    collectionDescriptorSearch (1),
                    productDescriptorSearch (3),
                    serviceDescriptorSearch (4),
                    catalogDescriptorSearch (5)
               }
          searchScope [2] IMPLICIT INTEGER
               {
                    localSearch (1),
                    wideSearch (2)
               }
     }
```

For further information, see Section 3.5.2.5 and Appendix E. 6.1. of Catalogue Interoperability Protocol (CIP) Specification - Release B, CEOS/WGISS/PTT/CIP-B, June 1998, Issue 2.4, Committee on Earth Observation Satellites (CEOS) (ftp://harp.gsfc.nasa.gov/incoming/fed/cip_spec24.pdf)

7.5.2 Order Extended Service

The *Order Extended Service*, which is a custom Z39.50 Extended Service, allows an *origin* to order products previously queried. The *Order ES* is presented in Table 13.

Further information describing the Order Extended Service can be found in Catalogue Interoperability Protocol (CIP) Specification - Release B, CEOS/WGISS/PTT/CIP-B, June 1998, Issue 2.4, Committee on Earth Observation Satellites (CEOS) (ftp://harp.gsfc.nasa.gov/incoming/fed/cip_spec24.pdf)

Table 13 - Order Extended Service

ASN.1 Definition	Meaning

{Z39.50-CIP-Order-ES} DEFINITIONS ::=
BEGIN
IMPORTS OtherInformation, InternationalString, IntUnit
FROM Z39.50-APDU-1995;
 	
CIPOrder ::= CHOICE
{
esRequest [1] IMPLICIT SEQUENCE{
toKeep [1] OriginPartToKeep,
notToKeep [2] OriginPartNotToKeep},
taskPackage [2] IMPLICIT SEQUENCE{
originPart [1] OriginPartToKeep,
targetPart [2] TargetPart}
}
	The Order Extended Serivce uses the Z39.50 Extended Servi ce Facility.

OriginPartToKeep ::= SEQUENCE
{
action [1] IMPLICIT INTEGER {
orderEstimate (1),
orderQuoteAndSubmit (2),
orderMonitor (3),
orderCancel (4)},
orderId [2] InternationalString OPTIONAL,
orderSpecification [3] OrderSpecification OPTIONAL,
statusUpdateOption [4] StatusUpdateOption OPTIONAL,
userInformation [5] UserInformation OPTIONAL,
otherInfo [6] OtherInformation OPTIONAL
}
	The **OriginPartToKeep** contains the following:
	• **action**, which indicates the type of operation that is requested to be performed for the order request. The supported operations are the following:
	  • **orderEstimate**, which is used to validate and obtain the estimate of an order specification.
	  • **orderQuoteAndSubmit**, which is used to quote[4] and submit an order specification.
	  • **orderMonitor**, which is used to monitor the progress of the processing of an order request.
	  • **orderCancel**, which is used to cancel an order request.

[4] The estimate for an order is approximate and non-binding, whereas the quote for an order is precise and binding.

ASN.1 Definition	Meaning
	• **orderId**, which is the identifier of the order request as provided as input by the origin. • **orderSpecification**, which is the specification of the order request as provided as input by the origin. Note that, in principle, the order request specified by the origin is unstructured, i.e. it contains a list of item descriptor identifiers and the order options related to them, but does not attempt to group them into packages and delivery units. • **statusUpdateOption**, which indicates how the origin wishes to be kept up to date as to the status of the order processing. • **userInformation**, which contains the personal user information as provided as input by the origin. • **otherInformation**, which contains additional information not specified by the CIP.

`OriginPartNotToKeep ::= SEQUENCE`
`{`
`orderId [1] InternationalString OPTIONAL,`
`orderSpecification [2] OrderSpecification OPTIONAL,`
`userInformation [3] UserInformation OPTIONAL,`
`otherInfo [4] OtherInformation OPTIONAL`
`}`
	The **OriginPartNotToKeep**[5] contains the following:
	• **orderId**, which is the identifier of the order request.
	• **orderSpecification**, which is the specification of the order request.
	• **userInformation**, which contains the personal user information.
	• **otherInformation**, which contains additional information not specified by the CIP.

[5] The definitions used in *OriginPartNotToKeep* are strictly identical to the ones provided in *OriginPartToKeep*. The former is used as input by the *target* (which may overwrite some values as appropriate) for the definition of *TargetPart*, whereas the latter remains unmodified and is stored in the *task package*. This duplication therefore allows the comparison of the order as specified by the *origin* (*OriginPartToKeep*) with the order as returned by the *target* (*TargetPart*).

ASN.1 Definition	Meaning

`TargetPart ::= SEQUENCE`
`{`
`orderId [1] InternationalString,`
`orderSpecification [2] OrderSpecification OPTIONAL,`
`orderStatusInfo [3] OrderStatusInfo OPTIONAL,`
`userInformation [4] UserInformation OPTIONAL,`
`otherInfo [5] OtherInformation OPTIONAL`
`}`
	The **TargetPart** contains the following:
	• **orderId**, which is the identifier of the order request as provided as output by the target.
	• **orderSpecification**, which is the specification of the order request as provided as output by the target. This order specification provided by the target overrides the specification provided as input by the origin in **originPartNotToKeep.** It contains the item descriptors and order options supplied as input, with any necessary modifications or additions, in a structured manner, i.e. the item descriptors are grouped into packages and delivery units.
	• **orderStatusInfo**, which indicates the status of the order request being performed[6].
	• **userInformation**, which contains the personal user information.
	• **otherInfo**, which contains additional information not specified by the CIP

`StatusUpdateOption ::= CHOICE`
`{`
`manual [1] NULL,`
`automatic [2] IMPLICIT INTEGER {`
`eMail (1)}`
`}`
	The **StatusUpdateOption** provides options for how the user will receive updates on the status of an **extended service** request. The parameters are:
	• **manual** the user performs the status request.
	• **automatic** where the OHS filing the order provides status updates for the user via **email**[7].

[6] Note the difference between the *operationStatus*, which is provided in the *ES Response*, and the *orderStatusInfo*, which is included in the *task package*. *operationStatus* provides status information for the *ES operation* as a whole and indicates whether the *ES operation* has been performed successfully or not by the *target*. *orderStatusInfo* provides status information for the order specified in the *task package* and indicates the state of the order or the process being performed for an order at the LOHS.

[7] This could be expanded in the future to include, for example, automatic update via the *origin*.

ASN.1 Definition	Meaning

UserInformation ::= SEQUENCE
{
userId [1] InternationalString,
userName [2] InternationalString OPTIONAL,
userAddress [3] PostalAddress OPTIONAL,
telNumber [4] InternationalString OPTIONAL,
faxNumber [5] InternationalString OPTIONAL,
emailAddress [6] InternationalString OPTIONAL,
networkAddress [7] InternationalString OPTIONAL,
billing [8] Billing OPTIONAL
}
	The **Userinformation** structure is presented by the origin part of a request to a target. The information provided contains mandatory fields (the user identifier) and optional fields. The target will allow the **Userinformation** structure contents to be used as an input to the delivery specification for elements which can be altered by the user. The target will refer to the local database contents for the user and will use the contents of the database, or the **Userinformation** structure depending on the privilege of the user to offer alternative information. The **UserInformation** structure consists of the following attributes:
	• **userId** the user identifier, the identifier which the user provides as part of an **InitializeRequest.**
	• **userName** the full name of the user.
	• **userAddress** a structure to hold the users address.
	• **telNumber** the users telephone number.
	• **faxNumber** the fax number for the user.
	• **emailAddress** the electronic mail address for the user.
	• **networkAddress** the network address to send files to electronically. For Internet addresses, the address is written in URL format to allow directories as well as domain's to be specified.
	• **billing** the method of payment (and hence of billing) available for the user.

OrderSpecification ::= SEQUENCE
{
orderingCentreId [1] InternationalString,
orderPrice [2] PriceInfo OPTIONAL,
orderDeliveryDate [3] InternationalString OPTIONAL,
orderCancellationDate [4] InternationalString OPTIONAL,
deliveryUnits [5] SEQUENCE OF DeliveryUnitSpec,
otherInfo [6] OtherInformation OPTIONAL
}
	The **OrderSpecification** is the specification of the order request and contains the following:
	• **orderingCentreId**, which identifies the ordering centre at which the order will be performed.
	• **orderPrice**, which is the price for the whole order.
	• **orderDeliveryDate**, which is the latest date at which the order can be expected to be delivered to the user.
	• **orderCancellationDate**, which is the latest date at which the user can cancel the order.
	• **deliveryUnits**, which contains the definition of the delivery units which compose the order.
	• **otherInfo**, which may be used to provide additional information not specified by the CIP.

ASN.1 Definition	Meaning

DeliveryUnitSpec ::= SEQUENCE
{
deliveryUnitId [1] InternationalString OPTIONAL,
deliveryUnitPrice [2] PriceInfo OPTIONAL,
deliveryMethod [3] DeliveryMethod OPTIONAL,
billing [4] Billing OPTIONAL,
packages [5] SEQUENCE OF PackageSpec,
otherInfo [6] OtherInformation OPTIONAL
}
	The **DeliveryUnitSpec** contains the specification of a single delivery unit (i.e. part of an order that is delivered as a unit):
	• **deliveryUnitId**, which is the identifier of the delivery unit.
	• **deliveryUnitPrice**, which is the price of the delivery unit.
	• **deliveryMethod**, which is the method with which the delivery unit is delivered to the user.
	• **billing**, which is the method with which the user is going to be billed.
	• **packages**, which contains the definition of the packages which compose the delivery unit.
	• **otherInfo**, which may be used to provide additional information not specified by the CIP.

DeliveryMethod ::= CHOICE
{
eMail [1] InternationalString,
ftp [2] FTPDelivery,
mail [3] PostalAddress,
otherInfo [4] OtherInformation
}
	The **DeliveryMethod** defines the method with which a delivery unit is delivered to the user and is one of the following:
	• **eMail**, which specifies the email address that the order will be delivered to
	• **ftp**, which specifies that the order will be delivered via ftp, the type of transfer and the ftp address
	• **mail**, which specifies that the order will be delivered via mail and provides the postal address
	• **otherInfo**, which may be used to provide additional information (such as an alternative delivery method) not specified by the CIP.

ASN.1 Definition	Meaning

FTPDelivery ::= SEQUENCE
{
transferDirection [1] IMPLICIT INTEGER
{
push (0),
pull (1)
},
ftpAddress [2] InternationalString
}
	The **FTPMethod** defines the method with which a delivery unit is delivered to the user and is one of the following:
	• **transferDirection**, which specifies that the order will be delivered via e-mail.
	• **ftpAddress**, which specifies that the order will be delivered via ftp.

Billing ::= SEQUENCE
{
paymentMethod [1] PaymentMethod,
customerReference [2] IMPLICIT CustomerReference,
customerPONumber [3] IMPLICIT InternationalString OPTIONAL
}
	The **Billing** structure[8] contains attributes which describe the method by which a user will pay for a service, together with supporting information regarding the payment. The attributes are:
	• **paymentMethod** indicates the method of payment used.
	• **customerReference** is the customer provided reference for the order.
	• **customerPONumber** is the purchase order provided by the customer for the order.

PaymentMethod ::= CHOICE
{
billInvoice [0] IMPLICIT NULL,
prepay [1] IMPLICIT NULL,
depositAccount [2] IMPLICIT NULL,
privateKnown [3] IMPLICIT NULL,
privateNotKnown [4] IMPLICIT EXTERNAL},
}
	The **PaymentMethod** structure contains attributes which describe the method by which a user will pay for a service. The attributes are:
	• **billInvoice** indicates that an invoice is to be sent to the user (or payee).
	• **prepay** indicates that payment has already been agreed/performed.
	• **depositAccount** indicates that there is a deposit account for the payment.
	• **privateKnown** indicates that the payment method is private and known.
	• **privateNotKnown** contain private unknown payment method information.

CustomerReference ::= SEQUENCE
{
customerId [1] InternationalString,
accounts [2] SEQUENCE OF InternationalString
}
	The **CustomerReference** structure contains attributes which provide a customer reference for the order. The attributes are:
	• **customerId** indicates the customer identifier at the LOHS.
	• **accounts** is the name of the account(s) available to apply charges to on behalf of the user.

[8] The Billing structure used by the Order Extended Service is derived from the *addlBilling* structure defined in the *Item Order ES*.

ASN.1 Definition	Meaning

PostalAddress ::= SEQUENCE
{
streetAddress [1] InternationalString,
city [2] InternationalString,
state [3] InternationalString,
postalCode [4] InternationalString,
country [5] InternationalString
}
	PostalAddress contains the postal address for a user and consists of:
	• **streetAddress**, which is the street name and number.
	• **city**, which is the name of the city (or nearest city).
	• **state**, which is the name of the state or county.
	• **postalCode**, which is the country specific postal code.
	• **country**, which is the name of the country.

PackageSpec ::= SEQUENCE
{
packageId [1] InternationalString OPTIONAL,
packagePrice [2] PriceInfo OPTIONAL,
package [3] CHOICE
{
predefinedPackage [1] PredefinedPackage,
adHocPackage [2] AdHocPackage
},
packageMedium [4] InternationalString,
packageKByteSize [5] INTEGER,
otherInfo [6] OtherInformation OPTIONAL
}
	The **PackageSpec** contains the specification of a single package (i.e. part of an order that is delivered on a single medium):
	• **packageId**, which is the identifier of the package.
	• **packagePrice**, which is the price of the package.
	• **package**, which contains the specification of the package. The package is one of the following:
	  • **predefinedPackage**, which is a package pre-defined by the data provider.
	  • **adHocPackage**, which is a package constructed ad-hoc by the data provider to fulfil the order request.
	• **packageMedium**, which is the medium on which the package will be delivered to the user.
	• **packageKByteSize**, which contains the size of the **package** in kilobytes.
	• **otherInfo**, which may be used to provide additional information not specified by the CIP.

PredefinedPackage ::= SEQUENCE
{
collectionId [1] InternationalString,
orderItems [2] SEQUENCE OF OrderItem,
otherInfo [3] OtherInformation OPTIONAL
}
	A **PredefinedPackage** contains the definition of a package that is pre-defined by the data provider. A PredefinedPackage is a collection that is stored in advance (i.e. not to fulfil a specific order) on a medium and is defined as follows:
	• **collectionId**, which is the identifier of the pre-packaged collection. Must be formatted according to the naming convention for collection identifiers specified in Appendix E.
	• **orderItems**, which contains the list of the order items contained in the package.
	• **otherInfo**, which may be used to provide additional information not specified by the CIP.

ASN.1 Definition	Meaning
`AdHocPackage ::= SEQUENCE OF OrderItem`	An **AdHocPackage** is a package that is defined ad-hoc by a data provider to fulfil a specific order. An **AdHocPackage** contains the list of the order items contained in the package.
`OrderItem ::= SEQUENCE { productId [1] InternationalString, productPrice [2] PriceInfo OPTIONAL, productDeliveryOptions [3] ProductDeliveryOptions OPTIONAL, processingOptions [5] ProcessingOptions OPTIONAL, sceneSelectionOptions [6] SceneSelectionOptions OPTIONAL, orderStatusInfo [7] OrderStatusInfo OPTIONAL, otherInfo [8] OtherInformation OPTIONAL }`	The **OrderItem** contains the specification of a single order item (i.e. the product that is ordered and that is to be delivered): • **productId**, which is the identifier of the ordered product. • **productPrice**, which is the price of the product. • **productDeliveryOptions**, which contains delivery options for the product. • **processingOptions**, which specifies the processing options that are to be applied on the product before delivery. • **sceneSelectionOptions**, which specifies the selection of the scene from the whole product that is to be delivered. • **orderStatusInfo**, which indicates the status of the order item[9]. • **otherInfo**, which may be used to provide additional information not specified by the CIP.
`ProductDeliveryOptions ::= SEQUENCE { productByteSize [1] INTEGER OPTIONAL, productFormat [2] InternationalString OPTIONAL, productCompression [3] InternationalString OPTIONAL, otherInfo [4] OtherInformation OPTIONAL }`	The **ProductDeliveryOptions** contains the specification of the options regarding the delivery of a product: • **productByteSize**, which contains the size of the product in bytes. • **productFormat**, which specifies the format of the product. • **productCompression**, which specifies the compression mechanism applied to the product. • **otherInfo**, which may be used to provide additional information not specified by the CIP.
`ProcessingOptions ::= CHOICE { formattedProcessingOptions [1] EXTERNAL, unformattedProcessingOptions [2] InternationalString }`	The **ProcessingOptions** specifies the processing options that are to be applied on the product before delivery and is one of the following: • **formattedProcessingOptions**, which specifies the processing options according to the format specified in [ORD]. • **unformattedProcessingOptions**, which specifies the processing options in a free-text form.

[9] Note the difference between the *orderStatusInfo* in *TargetPart*, which indicates the state, or the process being performed for, an order as a whole at the LOHS, and the *orderStatusInfo* in *OrderItem*, which indicates the state, or the process being performed for, a specific order item within an order at the LOHS.

ASN.1 Definition	Meaning

SceneSelectionOptions ::= CHOICE
{
formattedSceneSelectionOptions [1] EXTERNAL,
unformattedSceneSelectionOptions [2] InternationalString
}
	The **SceneSelectionOptions** specifies the selection of the scene from the whole product that is to be delivered and is one of the following:
	• **formattedSceneSelectionOptions**, which specifies the scene selection options according to the format specified in [ORD].
	• **unformattedSceneSelectionOptions**, which specifies the scene selection options in a free-text form.

PriceInfo ::= SEQUENCE
{
price [1] IntUnit,
priceExpirationDate [2] InternationalString,
additionalPriceInfo [3] InternationalString OPTIONAL
}
	The **PriceInfo** contains the information related to the price of an item:
	• **price**, which contains the price of the item.
	• **priceExpirationDate**, which specifies the latest date at which the price provided is valid (i.e. until the expiration date the origin is guaranteed that the price will not vary. However, after the expiration date the price may change).
	• **additionalPriceInfo**, which may be used to provide a textual explanation when the price of a item differs from the sum of the elements which compose this item (e.g. it can be used to explain why the price of a delivery unit differs from the sum of the prices of the packages which compose the delivery unit).

OrderStatusInfo ::= SEQUENCE
{
orderState [1] CHOICE
{
staticState [1] StaticState,
dynamicState [2] DynamicState
},
additionalStatusInfo [2] InternationalString OPTIONAL
}
	OrderStatusInfo describes the status of an extended service order request. The different status values are:
	• **orderState** indicates the state of the order request or the processing being performed for the order:
	  • **staticState** indicates the state of the order when no order request is being performed.
	  • **dynamicState** indicates the processing that is currently performed for an order request.
	• **additionalStatusInfo** contains additional status information provided by the LOHS (e.g. to clarify the meaning of the **orderState**).

StaticState ::= [1] IMPLICIT INTEGER
{
orderNotValid (1),
orderEstimated (2),
orderCompleted (3)
}
	StaticState describes the state of an order when no order request is active. The possible states are:
	• **orderNotValid** indicates that the order has not been successfully validated.
	• **orderEstimated** indicates that the order has been successfully validated and that an estimate is provided.
	• **orderCompleted** indicates that the order has been completed.

ASN.1 Definition	Meaning

`DynamicState ::= [2] IMPLICIT INTEGER`
`{`
`orderBeingEstimated (4),`
`orderBeingQuoted (5),`
`orderBeingProcessed (6),`
`orderBeingCancelled (7),`
`orderBeingDeleted (8)`
`}`
 	
`END`
	DynamicState describes the state of an order when an order request is active and thus being process. The possible states are:
	• **orderBeingEstimated** the order is currently being estimated by the target order handling system.
	• **orderBeingQuoted** the order is currently being quoted by the target order handling system.
	• **orderBeingProcessed** the order is currently being processed by the target order handling system.
	• **orderBeingCancelled** the order request which was previously sent to the target is being cancelled.
	• **orderBeingDeleted** the order is being deleted.

8. Proposed Additional OGC Basic Data Types

This is an informational annex.

8.1 AbsTime

A structure define an absolute time.

8.2 BinData

A "blob" of binary data

8.3 Cardinality

An enumeration that defines the possible values of cardinality.

8.3.1 Public Attributes:

one_to_one :

one_to_many :

many_to_one :

many_to_many :

8.4 DG_DirectedGraph

A Directed Graph structure. See Figure 28 - The DG_DirectedGraph for details of this structure.

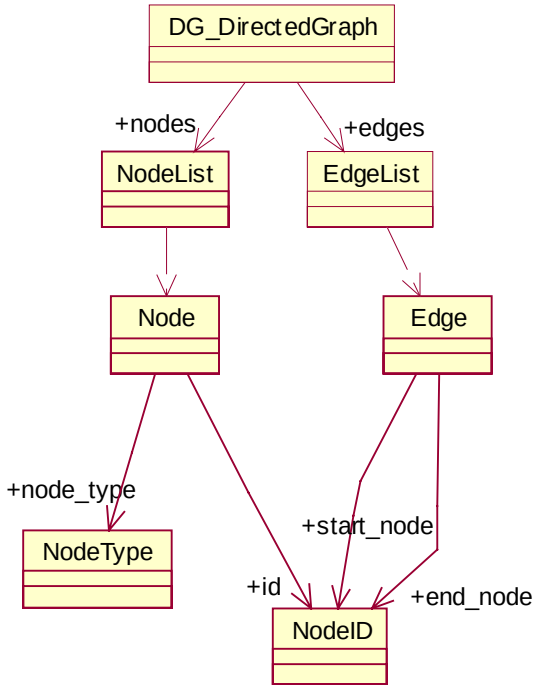


Figure 28 - The DG_DirectedGraph

8.5 DG_DirectedGraphList

A sequence of Directed Graphs

8.6 Date

A structure describing a single date with month and year.

8.6.1 Public Attributes:

year : unsigned short
The year stated as a 4 digit number.

month : unsigned short
The month stated as an unsigned short whose valid range is 1-12, where 1=January and 12=December.

day : unsigned short
The day of the month.

8.7 Edge

8.7.1 Public Attributes:

relationship_type : string
Defines the type of relationship if any between two Nodes of a DG_DirectedGraph.

8.8 EdgeList

A sequence of Edges

8.9 FileLocation

This structure contains the location and access information for a file.

8.9.1 Public Attributes:

user_name : string
An identifier for a user that has access to this file.

password : string
A password associated with the user_name field

host_name : string
The host on which this file resides.

path_name : string
The complete path to the directory containing this file.

file_name : string
The name of the file.

8.10 FileLocationList

A sequence of FileLocation structures

8.11 Name

A generic string identifier.

8.12 NameList

A sequence of generic identifiers.

8.13 NameValue

A structure that associates an identifier with a value.

8.13.1 Public Attributes:

theValue : any

theName : string

8.14 NameValueList

A sequence of Name Value pairs

8.15 NameValueTable

A 2D table where each cell is a Name Value structure.

8.16 Node

A Node of a DG_DirectedGraph.

8.16.1 Public Attributes:

attribute_name : string
The attribute being described.

value : any
The value of the attribute

8.17 NodeID

A numeric identifier for a Node in a DG_DirectedGraph

8.18 NodeList

A sequence of Nodes

8.19 NodeType

The type of node in a DG_DirectedGraph.

8.19.1 Public Attributes:

root_node :

entity_node :

record_node :

attribute_node :