

*Geospatial and Imagery Access  
Services Specification*

---

*National Imagery and Mapping Agency*

*United States Imagery and Geospatial Information Service*

*Version 3.5.1*

*6 August 2001*

## **Acknowledgments**

Many individuals and organizations provided support and technical contributions to this work, Individuals from numerous government agencies, contractor organizations and vendors contributed significantly to the development of this specification. We acknowledge these contributions and hope that these individuals and organizations will continue to actively support future updates and extensions. Thanks in advance.

## Revision History

- Image Access Facility, Version 0.1 Straw 23 May 1995.
- Image Access Facility, Version 0.2 Tin 11 June 1995.
- Image Access Facility, Version 0.3 Aluminum 19 June 1995.
- Image Access Facility, Version 0.4 Copper - For USIS release June 21, 1995.
- Image Access Facility, Version 0.5 Nickel - Preliminary draft release for Image Access Working Group (IAWG) June 29, 1995.
- Image Access Facility, Version 0.6 Iron - This release contained a relatively complete description of semantics and sequencing for sample implementation prototypers. July 12, 1995.
- Image Access Facility, Version 0.7 Silver - This release addressed comments received. September 6, 1995.
- Image Access Facility and Catalog Access Facility, Version 0.8 Gold - This release contained extensions based upon the additional architecture mining. February 8, 1996.
- Image Access Facility and Catalog Access Facility, Version 0.85 Gold Interim - Update for release and comment on March 22, 1996.
- Image Access Services Specification, Version 0.9 Platinum - Revisions based upon comments from Core Team Working Group, April 24, 1996.
- Image Access Services Specification Version 1.0 - ICCB Configuration-controlled, pilot operational specification for contractor and commercial prototyping and interoperability testing, June 20, 1996.
- Image Access Services Specification Version 1.1 -
- Revised to remove TBR's and TBDs concerning the PNF and IDF. Released for comments December 6, 1996
- Image Access Services Specification Version 1.1 -
- Approved by ICCB December 20, 1996.
- Name of document changed to Geospatial and Imagery Access Services Specification. Version number set to 3.0 to reflect extensions and updates for inclusion of geospatial data and operations.
- Geospatial and Imagery Access Services Specification
- Version 3.0 - Released for NCCB submittal 22 July 1997
- Geospatial and Imagery Access Services Specification Version 3.1. Includes updates to incorporate responses and comments from additional interface prototyping tests. Released for NCCB submittal 4 February 1998.
- Geospatial and Imagery Access Services Specification Version 3.2A. Released on 2 October for 24 November NCCB. Part of RFC N01-0085.
- Geospatial and Imagery Access Services Specification Version 3.3. Released on 9 November for 24 November NCCB. Part of RFC N01-0085. Incorporates mods resulting from review and UIP WG, 3-4 November 1998.
- Geospatial and Imagery Access Services Specification Version 3.3. Approved on 24 November at NCCB. As part of RFC N01-0085. Approval date annotated to document.

- Geospatial and Imagery Access Services Specification Version 3.3. Approved on 22 June 1999 at NCCB. As part of RFC N01-0114, E2.0 As Built Baseline. Approval date annotated to document.
- Geospatial and Imagery Access Services Specification Version 3.4. Draft released on 4 June as part of RFC N01-0127 for E2.5 UIP Baseline Update. Additional mods incorporated on 5 August as part of RFC Mod Package.
- Geospatial and Imagery Access Services Specification Version 3.4. Approved on 24 August 1999 at NCCB. As part of RFC N01-0127, E2.5 Baseline. Approval date annotated to document.
- Geospatial and Imagery Access Services Specification Version 3.5. Final Draft released on 18 February 2000 as part of RFC N01-0148 for UIP Baseline Update for the NE028/NE022 era. Update released on 18 February 2000 based on results of 19 January 2000 UIP/API Final Draft
- RFC N01-0148 withdrawn. GIAS Version 3.5. Final Draft re-released on 21 April 2000 as part of RFC N01-0203 for UIP Baseline Update for NE049/NE028/NE022 effectivities.
- Geospatial and Imagery Access Services Specification Version 3.5. Final Release dated 26 June 2000. Approved by NCCB on 26 June 2000 as part of RFC N01-0203J.
- Geospatial and Imagery Access Services Specification Version 3.5.1 Final Release dated 6 August 2001. Approved by NCCB on 6 August 2001 as part of RFC N01-0268.

## **Planned Releases**

- Regular updates at approximately six-month intervals or as needed.

## **Preface**

This document defines common interfaces and datatypes that are expected to be used by many other United States Imagery and Geospatial Information Service (USIGS) interface specifications. The intent of this specification is to document the interfaces, datatypes and error conditions that are expected to most commonly occur or be most broadly applicable across the USIGS architecture. The use of these common definitions will support interoperability among the various interface specifications in the USIGS architecture.

This specification was prepared consistent with industry practices and is modeled after those being prepared by the Object Management Group (OMG) industry consortium. This approach is also consistent with guidelines and direction established by NIMA through its Architecture and Standards processes.

## Table of Contents

<b>1 OVERVIEW</b>	<b>1</b>
<b>1.1 Background</b>	<b>1</b>
<b>1.2 Overview</b>	<b>1</b>
<b>2. INTERFACE OVERVIEW</b>	<b>6</b>
<b>2.1. Overview</b>	<b>6</b>
<b>2.2. Data Types</b>	<b>7</b>
2.2.1. USIGS Common Objects	7
2.2.2. GIAS Specific Data Types (j/NPS)	8
2.2.3. GIAS Simple Data Types (j/NPS)	25
<b>2.3. Interfaces</b>	<b>32</b>
2.3.1. Library (j/NPS)	32
2.3.2. LibraryManager	34
2.3.3. RequestManager (j/NPS)	36
2.3.4. AccessManager	39
2.3.5. OrderMgr	43
2.3.6. DataModelMgr	45
2.3.7. StandingQueryMgr	52
2.3.8. CreationMgr (j/NPS)	54
2.3.9. UpdateMgr	58
2.3.10. CatalogMgr	61
2.3.11. ProductMgr	63
2.3.12. IngestMgr	66
2.3.13. QueryOrderMgr	68
2.3.14. VideoMgr	70
2.3.15. Request (j/NPS)	70
2.3.16. CreateMetaDataRequest	74
2.3.17. SetAvailabilityRequest	75
2.3.18. GetRelatedFilesRequest	76
2.3.19. CreateRequest	76
2.3.20. UpdateRequest	77
2.3.21. SubmitQueryRequest	78
2.3.22. SubmitStandingQueryRequest	80
2.3.23. HitCountRequest	86
2.3.24. GetParametersRequest	87
2.3.25. IngestRequest	88
2.3.26. OrderRequest	89
2.3.27. SubmitQueryOrderRequest	89
2.3.28. CreateAssociationRequest	91

2.3.29. UpdateByQueryRequest	92
<b>2.4. Exceptions</b>	<b>93</b>
2.4.1. Exception Model	93
2.4.2 InvalidInputParameter Exceptions (j/NPS)	93
2.4.3 ProcessingFault Exceptions (j/NPS)	100
2.4.4 SystemFault Exceptions (j/NPS)	100
<b>3. CALLBACK (J/NPS)</b>	<b>101</b>
<b>3.1. Callback (j/NPS)</b>	<b>101</b>
3.1.1. notify (j/NPS)	101
3.1.2. release (j/NPS)	102
<b>4. BOOLEAN QUERY SYNTAX</b>	<b>103</b>
<b>4.1. Overview</b>	<b>103</b>
<b>4.2. BQS Design</b>	<b>103</b>
<b>4.3. BNF definition</b>	<b>103</b>
<b>4.4. Rules and Constraints</b>	<b>108</b>
4.4.1. Operator Precedence	108
4.4.2. Units	109
4.4.3. Strings and Wildcards	109
4.4.4. BQS and UCOS/GIAS Types	109
4.4.5. Deriving attribute names from data model	113
4.4.6. Attribute Name Syntax Rule	113
<b>APPENDIX A: GIAS IDL</b>	<b>114</b>
<b>APPENDIX B: CALLBACK IDL</b>	<b>154</b>
<b>APPENDIX C UML DIAGRAMS</b>	<b>155</b>
<b>APPENDIX D REFERENCE OMG STANDARD IDL</b>	<b>157</b>
<b>APPENDIX E CORBA STANDARD EXCEPTIONS</b>	<b>157</b>
<b>APPENDIX F ACRONYMS</b>	<b>159</b>

<b>APPENDIX G: UML STATECHART DIAGRAMS</b>	<b>160</b>
--	------------

<b>APPENDIX H: POINTS OF CONTACT</b>	<b>171</b>
--------------------------------------	------------

# 1 Overview

## 1.1 Background

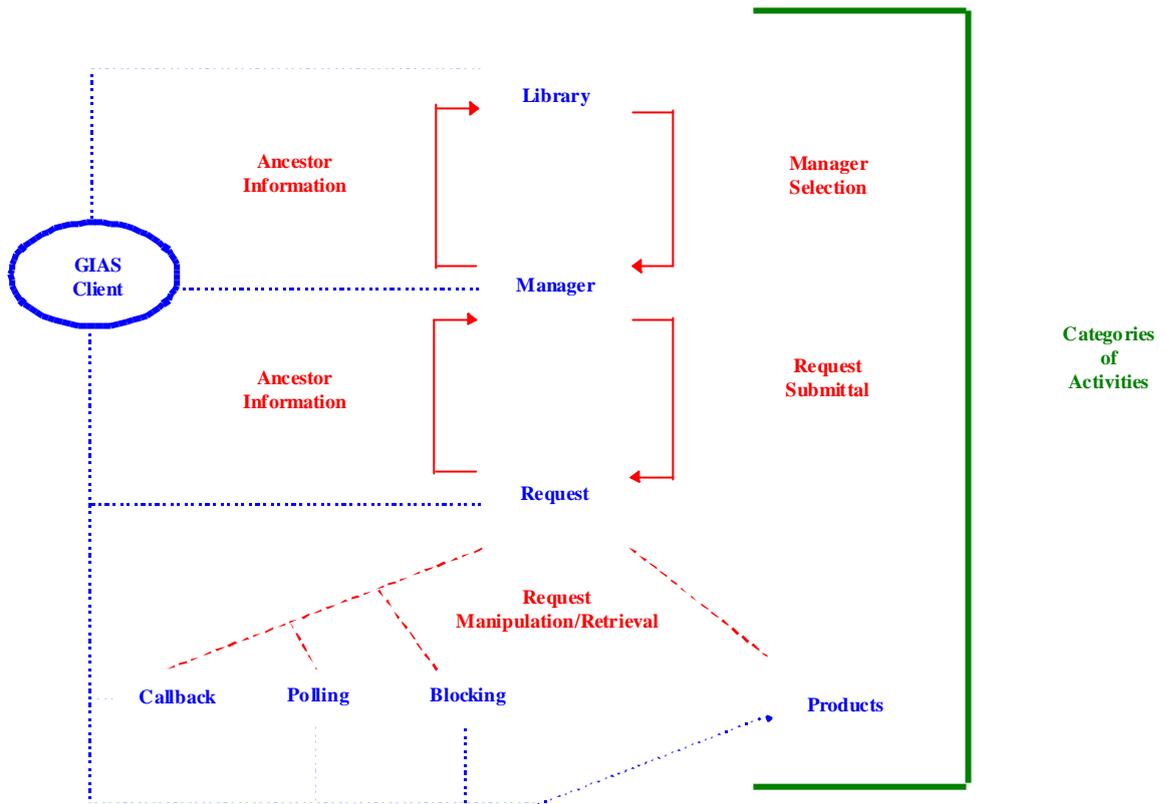
The Geospatial and Imagery Access Services (GIAS) specification defines the core interfaces of the United States Imagery and Geospatial Service (USIGS) libraries for client access to geospatial information. USIGS is a single and integrated system, which is evolving from multiple systems to support the Imagery and Geospatial Community (IGC) in the US Government's acquisition and production of imagery, imagery intelligence, and geospatial information. USIGS has a common information management framework that enables sharing of data, services, and resources among IGC members and their consumers

The GIAS provides client access, which includes search, discovery, browsing, and retrieval of information and its associated meta-data. Geospatial information is defined to include imagery and imagery-based information, maps, charts and any other data that has a well-defined association with a point or area on the Earth.

## 1.2 Overview

The GIAS interfaces are specified using the Object Management Group (OMG) Interface Definition Language (IDL). IDL is a language-independent notation for specifying software interfaces. IDL can be readily compiled into software interfaces for various programming languages including C, C++, Java, Ada95, and Smalltalk.

To help the reader assimilate the GIAS specification, a series of figures are presented providing varying levels of details concerning the GIAS interfaces structure and usage. At the highest level of abstraction, the GIAS interfaces are partitioned into four activity categories: library; request managers; request objects; and callback/product objects. Figure 1-1 shows how the GIAS interface is structured and what are its activity categories.



**Figure 1-1 GIAS Interface Structural and Activity Models**

Figure 1-2 is based on the notion that a GIAS client requires access to a Library, which is accessible through the GIAS interfaces. The GIAS client interacts with the Library to select and request access to a manager of a specific type. (“manager selection activity category”). Using the provided Manager the client can submit requests for the Library to perform tasks (“request submittal activity category”). Each request submittal returns a Request object. The GIAS client then uses the Request object to monitor progress on the task and to retrieve the results. The Request object also provides a mechanism (a Callback) to allow a client to be notified of the progress of the task. The GIAS client can also obtain information (“ ancestor information activity category”) on a specific request or manager. This allows a GIAS client to determine for any Request, the Manager that is managing it and for any Manager determine the Library(s) it services.

Figure 1-2 provides another view of the GIAS interface specification as a UML class diagram. This class diagram represents a static GIAS interface structure which is partitioned by four abstract interface classes: *LibraryManager*; *AccessManager*; *RequestManager*; and *Request*. The *LibraryManager* represents an instance of the specific Library being accessed by the client for requesting products. In addition, specializations of the abstract class *LibraryManager* provide access to search a Library catalog, query for products in the Library, discover elements of a data model in use by the Library, and archive new products into the Library.

The *AccessManager* provides operations for a client to “monitor” the status of a Library request for a specific product. In addition, specializations of the abstract class *AccessManager* provide operations for specific products or data sets such as: orders for products, retrieval of tiled products, determination of the characteristics of a specific product or data set, bulk transfer of data from a Library, and video products (*N.B.*, this capability is currently not implemented).

The *RequestManager* provides operations for requested products or data sets. The *RequestManager* provides selector and modifier operations for a request instance, which is a specialization of the abstract interface *Request*. The set of request instances is shown in Figure 1-2.

Access to request operations submitted by the client can be transmitted to the Library by polling the Library requested object (i.e., asynchronous), blocking (i.e., synchronous) the client until the Library requested object is available, or requesting a Callback event when Library requested object is available.

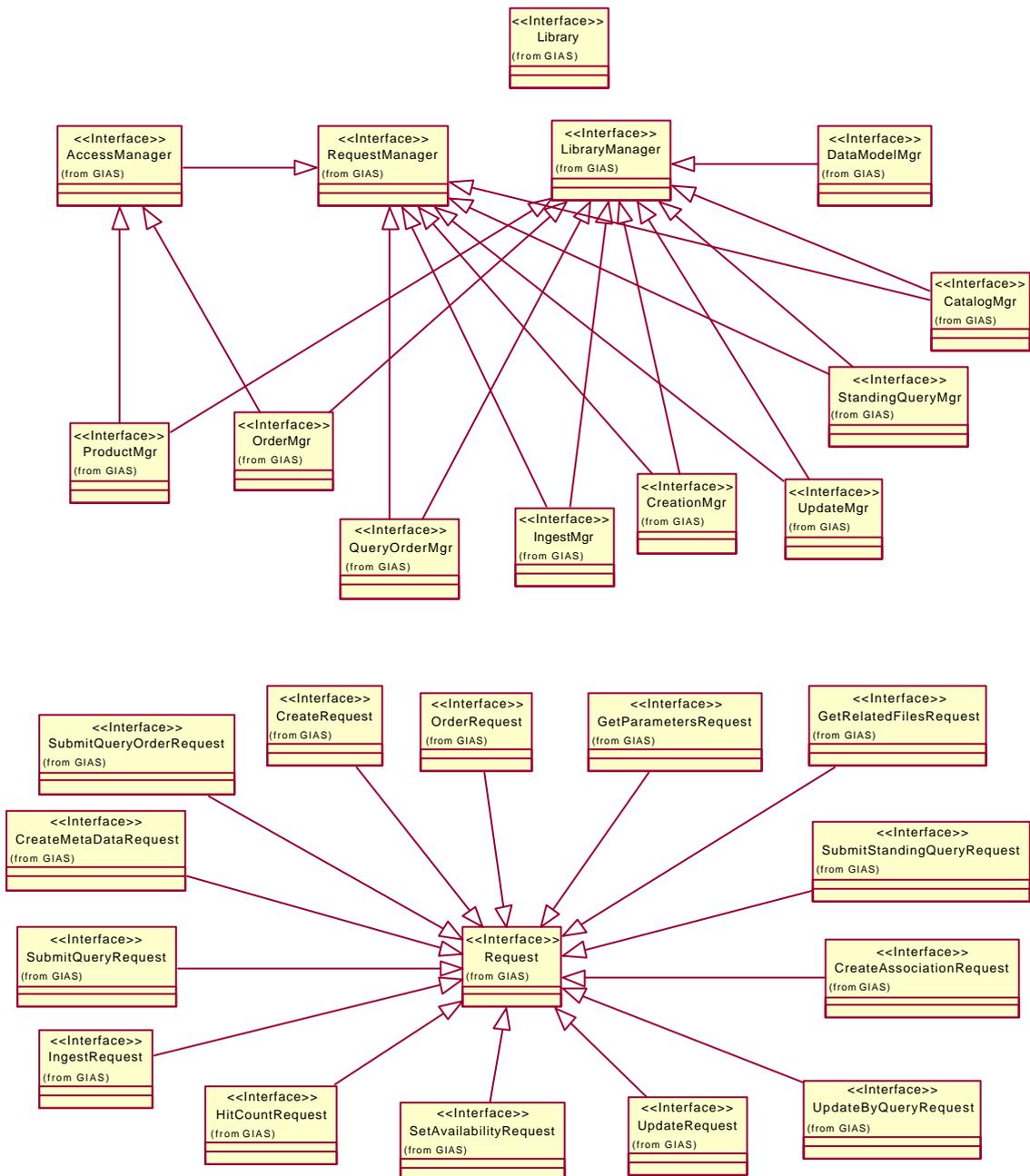


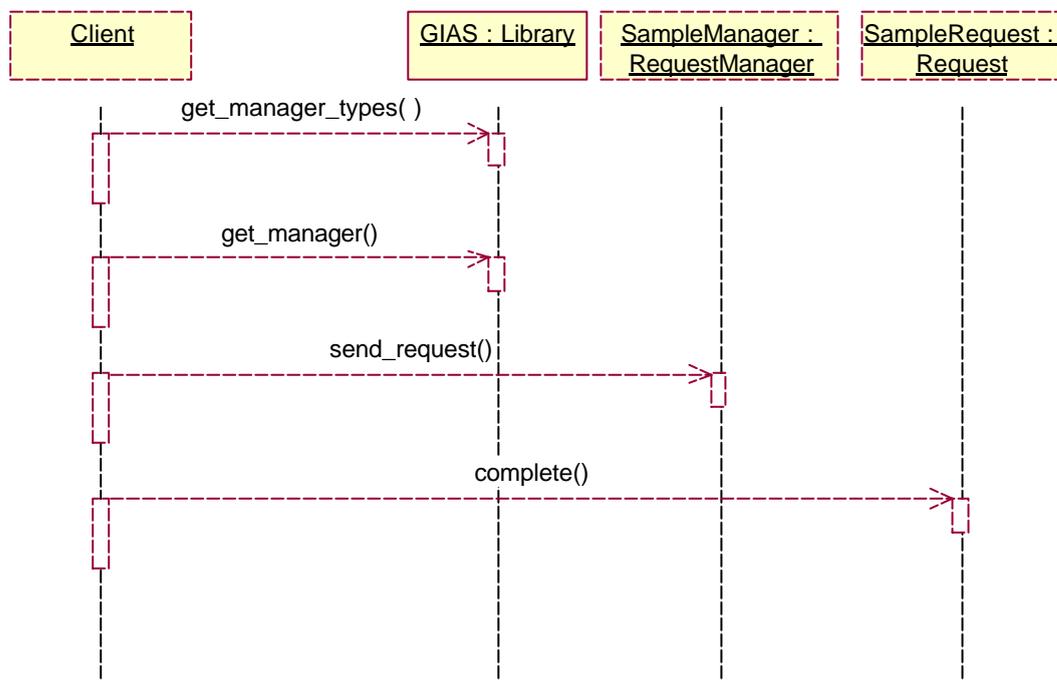
Figure 1-2 UML Static Class Diagram of GIAS Interface Structure

The following three sequence diagrams outline three scenarios for interactions between a client and GIAS Requests: Blocking, Polling and Callback.

Figure 1-3 provides the sequence for blocking requests made by GIAS clients. The scenario is initiated by the GIAS Client inquiring and obtaining a list of what types of Managers are available from the GIAS Library. Upon receiving and evaluating the list, the GIAS Client selects a Manager type (SampleMgr) and requests access to a Manager object of that type from the GIAS Library. The GIAS Client uses this Manager to submit requests (send\_request). The SampleMgr returns a SampleRequest object to the client. The client then calls “complete” on the Request and is blocked until all processing is completed.

Figure 1-4 provides the sequence for polling requests made by GIAS clients. The scenario is initiated by the GIAS Client inquiring and obtaining a list of what types of Managers are available from the GIAS Library. Upon receiving and evaluating the list, the GIAS Client selects a Manager type (SampleMgr) and requests access to a Manager object of that type from the GIAS Library. The GIAS Client uses this Manager to submit requests (send\_request). The SampleMgr returns a SampleRequest object to the client. The client then calls “get\_status” repeatedly until the status returns as COMPLETE. The client then calls “complete” to complete the Request.

Figure 1-5 provides the sequence for “callback” requests made by GIAS clients. The scenario is initiated by the GIAS Client inquiring and obtaining a list of what types of Managers are available from the GIAS Library. Upon receiving and evaluating the list, the GIAS Client selects a Manager type (SampleMgr) and requests access to a manager object of that type from the GIAS Library. The GIAS Client uses this Manager to submit requests (send\_request). The SampleMgr returns a SampleRequest object to the client. In this scenario, the GIAS Client is associated with a Callback object. It registers this Callback object with the SampleRequest. When the SampleRequest completes, SampleRequest invokes “notify” on the Callback object.



**Figure 1-3 UML Use Case Sequence Diagram for a GIAS Blocking Scenario**

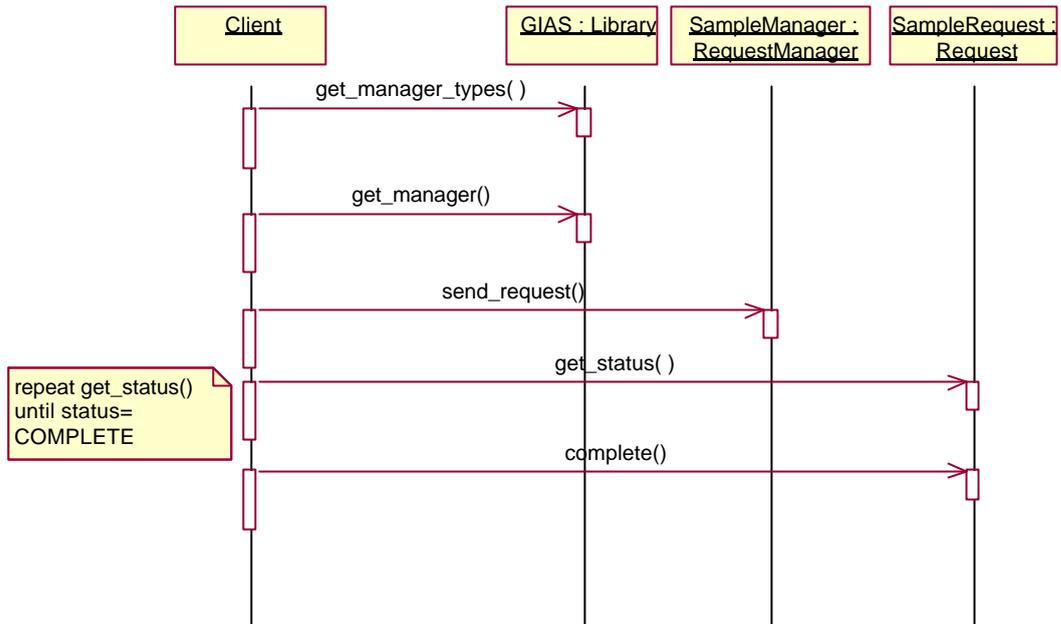


Figure 1-4 UML Use Case Sequence Diagram for a GIAS Polling Scenario

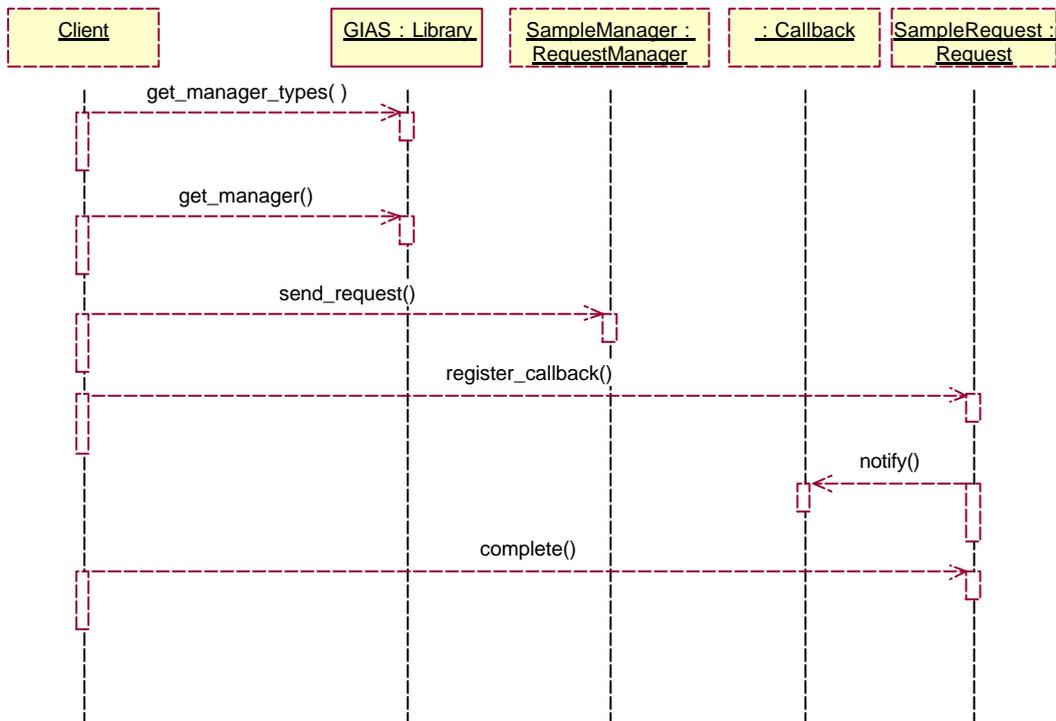


Figure 1-5 UML Use Case Sequence Diagram for a GIAS Callback Scenario

## 2. Interface Overview

### 2.1. Overview

The GIAS specification defines, through the use of OMG IDL, the interfaces, data types and error conditions that represent a geospatial information Library. A GIAS-based geospatial Library has interfaces that allow a client to search and discover information (data sets/products) contained in the Library, inquire about details of a particular data set/product and arrange for the delivery of the data set/product to another location or to another system. Also provided are interfaces to allow a client to nominate information to be included in the Library. There are also interfaces to allow Library-to-Library interchange of information as well as interfaces that support management and control of the client-library interactions.

The GIAS specification does NOT define interfaces for functions such as: locating Libraries with specific characteristics (this is the function of a Trading service), requests for the collection or acquisition of information not in a Library (this is the function of a collection requirements system), management of the underlying communication and other infrastructure or requests for processing of information not directly related to the search or delivery of information (this is the function of the exploitation and production systems).

The definitions and semantics associated with the elements of the GIAS specification are intended to be as general and as broadly useful as possible. It is not intended to be a description of any single implementation or system but is intended to allow great latitude in the design and implementation schemes for geospatial Libraries. However, to ensure interoperability, all systems that must interoperate must make the same interpretations concerning this general specification. A *profile* of the GIAS specification for the intended community of use is a critical supplement to the GIAS specification itself. A profile is a formal documentation of the specific interpretations, limits, and conventions chosen by the community of use. The USIGS community will be producing profiles of the GIAS specification that document these factors.

The following sections detail the interfaces, data types and error conditions that compose the GIAS interface definition.

All elements of the GIAS definition are contained in the GIAS module, which identifies and defines data types, interfaces, operations, and exceptions.

The interfaces defined in GIAS use the exception model defined in USIGS Common Object Services (UCOS) Specification. That specification defines a general purpose model which the GIAS specification extends by defining a set of error condition identifiers (string constants) (see section 2.4 for these string constants). In the interface definitions that follow, the GIAS-specific error conditions that an interface may return are identified by 1) defining the set of UCOS general purpose exceptions that may be returned and 2) listing the set of GIAS-specific error condition identifiers that may be returned inside one of these UCOS defined general purpose exceptions.

```
module GIAS
{
.... all GIAS elements ...
```

```
} ;
```

## 2.2. Data Types

### 2.2.1. USIGS Common Objects

In order to support interoperability among the components of the USIGS architecture, the most common or most broadly useful data types, interfaces, operations, and exceptions (i.e., error conditions) have been defined and collected into the USIGS Common Object (UCO) Specification. The intent is for all USIGS specifications to draw upon the UCO definitions when appropriate rather than redefine a common element. In order to support interoperability, the GIAS specification uses the definitions in the UCO whenever they are appropriate. The specific UCO entities that GIAS uses are detailed below. The definitions given are descriptions of how GIAS uses these entities and are not intended to replace the definitions specified in the UCO. Only cases where the UCO data type is used as an element of a GIAS data type are detailed below. For cases where the UCO element is used in a GIAS operation, its intended use is defined in the text accompanying each operation. All GIAS operations are defined in section 2.3.

#### 2.2.1.1. PropertyList

```
typedef UCO::NameValueList PropertyList;
```

The PropertyList is a typedef of UCO::NameValueList structure, which is re-used to hold the name value pairs (Properties) that are used to augment or clarify many of the operations of the RequestManager.

#### 2.2.1.2. GeoRegion and GeoRegionType

```
typedef UCO::Rectangle GeoRegion;  
  
enum GeoRegionType {  
    LINE_SAMPLE_FULL,  
    LINE_SAMPLE_CHIP,  
    LAT_LON,  
    ALL,  
    NULL_REGION};
```

UNCLASSIFIED

The GIAS specification uses the GeoRegion data type to define geospatial subsections of products or data sets. Currently the only type of subsection allowed is rectangular. The GIAS specification thus defines GeoRegion based on the UCO:: Rectangle. The GeoRegion type upper\_left component is defined as the first column/row in the Chippable Image. The enumeration GeoRegionType indicates the type of coordinate system used by a GeoRegion:

- LINE\_SAMPLE\_FULL – An image coordinate system defined by the original full resolution image;
- LINE\_SAMPLE\_CHIP – An image coordinate system defined by an image which has been extracted from a larger image ;
- LAT\_LON – A geographic coordinate system expressed in decimal degrees (x = latitude y=longitude). The convention used in this specification is that a positive value for latitude or longitude indicates a northern/eastern direction and a negative value indicates a southern/western direction.
- ALL – The special case of a GeoRegion that includes the entire product or data set.
- NULL\_REGION – The special case of a null or empty GeoRegion

### 2.2.1.3. Status and State (j/NPS)

The GIAS specification uses the State enumeration defined in UCO paragraph 2.2.4 to identify the state of Request objects. The specific states and state transitions that the concrete Request objects may use are defined in Appendix G.

## 2.2.2. GIAS Specific Data Types (j/NPS)

The GIAS specification defines a number of data types that are specific to the GIAS. The definitions of the specific types are given in the following sections.

### 2.2.2.1. AvailabilityRequirement, and UseMode

The types described in this subsection are used exclusively by the *AccessManager*.

```
enum AvailabilityRequirement
{
    REQUIRED, NOT_REQUIRED
};
```

The enumeration `AvailabilityRequirement` is used by the `AccessManager` to determine if the request is to place a product into a certain mode (`REQUIRED`) or a request for a product to be removed from a certain mode (`NOT_REQUIRED`).

```
typedef string UseMode;
```

*UseMode* is a string that describes a purpose or intended use of a data set or product. It is used by the *AccessManager* to support client requests and monitoring of the readiness of products for their use.

#### **2.2.2.2. OrderType, ProductSpec, ProductFormat, AlterationSpec, PackagingSpec, ImageFormat, Compression, BitsPerPixel, Algorithm, SupportDataEncoding, ProductFormatList, ImageSpec, ImageSpecList, ImageUniqueIdentifier and AlternationSpecList**

The types described in this subsection are used by the `OrderMgr` to describe the details of an order. An order is a request to have one or more products delivered from a `Library` to one or more destinations in one or more specific forms.

```
enum OrderType {STANDING, IMMEDIATE};
```

This type is used to distinguish between an immediate order which is to be performed once based on the current state of the `Library` and a standing order which is to be performed repeatedly until the order lifetime expires.

```
typedef string ProductFormat;
```

This type identifies the specific product format.

```
typedef string ImageFormat;
```

This type identifies the specific image format.

```
typedef string Compression;
```

This type identifies the compression type.

```
typedef short BitsPerPixel;
```

This type identifies the number of bits of data in an uncompressed pixel.

```
typedef string Algorithm;
```

This type identifies the algorithm to be used for alteration.

```
typedef string ImageUniqueIdentifier;
```

This type identifies the unique image identification that can be used for ordering a set of images.

```
typedef any ProductSpec;
```

The ProductSpec will contain one of the following specialized product format structures: ImageSpec or ImageSpecList.

```
Enum SupportDataEncoding {ASCII, EBCDIC};
```

```
typedef sequence < ProductFormat > ProductFormatList;
```

```
struct ImageSpec
```

```
{
```

```
    ImageFormat imgform;
```

```
    ImageUniqueIdentifier imageid;
```

```
    Compression comp;
```

```
    BitsPerPixel bpp;
```

```
    Algorithm algo;
```

```
    RsetList rrds;
```

```
    GeoRegion sub_section;
```

```
    GeoRegionType geo_region_type;
```

```
    SupportDataEncoding encoding;
```

```
};
```

```
typedef sequence < ImageSpec > ImageSpecList;
```

```
struct AlterationSpec
{
    ProductFormat pf;
    ProductSpec ps;
    GeoRegion sub_section;
    GeoRegionType geo_region_type;
};
```

```
typedef sequence < AlterationSpec >
AlterationSpecList;
```

This structure describes details of how a product is to be altered before being delivered. There are three types of alterations that can be specified:

1. The element *pf* indicates the desired choice of a specific data and compression format.
2. The element *ps* contains alteration details specific to the specific product format
3. The element *sub\_section* contains a geographically defined subsection of the whole product.
4. The element *geo\_region\_type* defines the type of coordinate system in element *sub\_section*

```
struct PackagingSpec
{
    string package_identifier;
    string packaging_format_and_compression;
};
```

The PackagingSpec defines characteristics of the data package that are sent to the client. This package may contain one or more products in the requested form. The PackagingSpec allows a client to specify:

1. An identifier for the package (*package\_identifier*) so the client can identify the package when it arrives and

2. A choice of specific format and compression type for the package (packaging\_format\_and\_compression).

### 2.2.2.3. TailoringSpec

```
struct TailoringSpec {
    UCO::NameNameList specs;
};
```

The TailoringSpec structure defines the information required to describe processing or modifications to be done to an ordered product by the Library prior to being sent to the client. The TailoringSpec contains a NameNameList where each NameName pair includes an identifier for a processing step in the first Name and any parameters for that step in the second Name.

### 2.2.2.4. Destination, DestinationType, and DestinationList

```
enum DestinationType
{
    FTP, EMAIL, PHYSICAL
};
```

```
union Destination switch (DestinationType)
{
    case FTP:          UCO::FileLocation f_dest;
    case EMAIL:       UCO::EmailAddress e_dest;
    case PHYSICAL:    PhysicalDelivery h_dest;
};
```

```
typedef sequence < Destination > DestinationList;
```

These types describe the details of a destination for an order. The enumeration `DestinationType` describes the three choices for delivery modes. Mode `FTP` indicates the package will be sent electronically via an FTP (or similar) mechanism, mode `EMAIL` indicates that the package will be sent via an e-mail enclosure and mode `PHYSICAL` indicates a hardcopy or physical media was ordered and will be delivered physically.

The structure `Destination` defines the data type that must be provided when the above-defined modes are selected. Element `f_dest` indicates the location for FTP type deliveries, element `e_dest` contains an email address for EMAIL type deliveries, and `h_dest` contains a `PhysicalDelivery` structure that contains the details required for a PHYSICAL type delivery

The sequence `DestinationList` is used to provide a set of destinations as a single list.

#### 2.2.2.5. `MediaType` and `MediaTypeList`

```
struct MediaType
{
    string media_type;
    unsigned short quantity;
};

typedef sequence < MediaType > MediaTypeList;
```

This data structure specifies the media type of the data set or product ordered by a client. This attribute of an order is only relevant for a `DestinationType` type of `PHYSICAL`.

#### 2.2.2.6 `PhysicalDelivery`

```
struct PhysicalDelivery
{
    string address;
};
```

This structure defines the details for an order which is to be delivered physically to the receiver rather than electronically.

#### 2.2.2.7. ValidationResults and ValidationResultsList

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

typedef sequence < ValidationResults >
ValidationResultsList;
```

The data structure `ValidationResults` is used by the `OrderMgr` to validate orders before submitting by the *order* operation. The structure indicates the validity of a proposed *order* and information concerning the proposed *order*. A validated order will be judged to be in one of three states: valid, invalid or valid with warning. If the value of the data element “valid” is “TRUE”, then this indicates that the order is valid. If the value of the data element “valid” is “FALSE”, this indicates that the order has been judged to be invalid and the value of the data element “details” is a human readable and interpretable string that explains why the order is invalid. The additional data element “warning” is used in combination with the “valid” boolean to indicate that a warning has been noted. The table below summarizes the states of `ValidationResults`.

**Table 2-1 Validation Results Table**

Warning	Valid	
	True	False
True	Details contains warning message of proposed order	Details contains explanation of invalid order
False	Details not applicable	Details contains explanation of invalid order

**2.2.2.8. RelatedFileType, RelatedFileTypeList, RelatedFile & RelatedFileList**

```
typedef UCO::Name RelatedFileType;
```

```
typedef sequence<RelatedFileType>
RelatedFileTypeList;
```

```
struct RelatedFile
```

```
{
    RelatedFileType file_type;
    UCO::FileLocation location;
};
```

```
typedef sequence <RelatedFile> RelatedFileList;
```

The RelatedFileType is used by the ProductMgr get\_related\_files to provide access to arbitrary datasets/files related to a specified Product.

The structure RelatedFile defines a relationship between a file instance located at *location* and a RelatedFileType in *file\_type*. RelatedFileList provides a sequence of these structures. These structures are used by the CreationMgr to support the specification of related files when defining a product for creation.

**2.2.2.9. ConceptualAttributeType, Entity, DomainType, DateRange, IntegerRange, FloatingPointRange, Domain, AttributeType, RequirementMode, AttributeInformation, Association, ViewName, ViewNameList, View, ViewList, IntegerRangeList, FloatingPointRangeList, AssociationList, and AttributeInformationList**

The data types listed in this subsection are utilized by the *DataModelMgr*. The *DataModelMgr* uses the idea of a Conceptual Attribute to remain data model neutral. A conceptual attribute is an attribute that serves as a label for a concept that is likely to be present in metadata models of interest and which needs to be discovered by the client. It is in essence a minimal meta-metadata model. For each metadata model of interest, each conceptual attribute can either be mapped to a single logical attribute or is not supported.

```
enum ConceptualAttributeType
{
    FOOTPRINT, CLASSIFICATION, OVERVIEW, THUMBNAIL,
    DATASETTYPE, MODIFICATIONDATE, PRODUCTTITLE,
    DIRECTACCESS, DIRECTACCESSPROTOCOL,
    UNIQUEIDENTIFIER, DATASIZE
};
```

This data type is used by the *DataModelMgr* selector operation `get_logical_attribute_name` to obtain the logical attribute name that is the equivalent of the *ConceptualAttributeType*. The *ConceptualAttributeType* exhaustively enumerates all conceptual attribute types. The definition, valid data types and domain for each of the *ConceptualAttributes* are defined in the following table:

ConceptualAttribute	Description	Valid Data types	Domain
FOOTPRINT	Describes a products geospatial location or bounds	UCO::Coordinate2d UCO::Coordinate3d UCO::LineString2d UCO::LineString3d UCO::Polygon UCO::PolygonSet UCO::Rectangle UCO::RectangleList UCO::Circle UCO::Ellipse	Entire domain of datatype
CLASSIFICATION	Indicates the products security classification	CORBA::string	Domain defined in appropriate GIAS profile

OVERVIEW	Indicates the file name of the product's overview representation	CORBA::string	Entire domain of datatype
THUMBNAIL	Indicates the file name of the product's thumbnail representation	CORBA::string	Entire domain of datatype
DATASETTYPE	Indicates the product's type	CORBA::string	Domain defined in appropriate GIAS profile
MODIFICATIONDATE	Indicates the date the product was last modified	UCO::AbsTime UCO::Date	Entire domain of datatype
PRODUCTTITLE	Indicates the textual title of the product	CORBA::string	Entire domain of datatype
DIRECTACCESS	Indicates the file location of the product	UCO::FileLocation	Entire domain of datatype
DIRECTACCESSPROTOCOL	Indicates the transfer protocol by which the product may be retrieved	CORBA::string	Domain defined in appropriate GIAS profile
UNIQUEIDENTIFIER	Indicates the identifier which uniquely identifies the product	CORBA::long CORBA::string	Entire domain of datatype
DATASIZE	Indicates the total data size of the product	UCO::FileSize	Entire domain of datatype

```
typedef string Entity;
```

This data type represents the name of a data model entity.

```
typedef string ViewName;
```

```
typedef sequence< ViewName > ViewNameList;

struct View {

    ViewName view_name;

    ViewNameList subViews;

    boolean orderable;

};

typedef sequence < View > ViewList;
```

These data types are used to define the identifier for a view (viewName) and to express the relationships of views to other related views (subViews). It also indicates whether the contents described by the view can be ordered.

```
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

};
```

This data type defines the equivalent of a mathematical domain, which is used in the data structure Domain.

```
struct DateRange

{

    UCO::AbsTime earliest;

    UCO::AbsTime latest;

};
```

This data structure defines the mathematical range for data that expresses a calendar date. It is used in the data structure Domain.

```
struct IntegerRange
{
    long lower_bound;
    long upper_bound;
};
```

This data structure defines the mathematical range for integer, which is used in the data structure *Domain*.

```
struct FloatingPointRange
{
    double lower_bound;
    double upper_bound;
};
```

This data structure defines the mathematical range for floating point, which is used in the data structure *Domain*.

```
typedef sequence < IntegerRange > IntegerRangeList;
```

This data type definition represents a set of integer ranges, which is used in the data structure *Domain*.

```
typedef sequence < FloatingPointRange >
FloatingPointRangeList;
```

This data type definition represents a set of floating point ranges, which is used in the data structure *Domain*.

```
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 LIST: UCO::NameList l;
    case ORDERED_LIST: UCO::NameList ol;
    case INTEGER_RANGE: IntegerRange ir;
    case FLOATING_POINT_RANGE: FloatingPointRange
fr;
    case FLOATING_POINT_SET:
FloatingPointRangeList fps;
    case GEOGRAPHIC: UCO::Rectangle g;
    case GEOGRAPHIC_SET: UCO::RectangleList
gs;
    case BINARY_DATA: UCO::BinData bd;
    case BOOLEAN_VALUE: boolean bv;
};

```

This data type associates a member of the set DomainType with a defined range.

```

enum AttributeType
{
    TEXT,
    INTEGER,
    FLOATING_POINT,
    UCOS_COORDINATE,
    UCOS_POLYGON,
    UCOS_ABS_TIME,
    UCOS_RECTANGLE,
    UCOS_SIMPLE_GS_IMAGE,

```

```
UCOS_SIMPLE_C_IMAGE ,
UCOS_COMPRESSED_IMAGE ,
UCOS_HEIGHT ,
UCOS_ELEVATION ,
UCOS_DISTANCE ,
UCOS_PERCENTAGE ,
UCOS_RATIO ,
UCOS_ANGLE ,
UCOS_FILE_SIZE ,
UCOS_FILE_LOCATION ,
UCOS_COUNT ,
UCOS_WEIGHT ,
UCOS_DATE ,
UCOS_LINestring ,
UCOS_DATA_RATE ,
UCOS_BIN_DATA ,
BOOLEAN_DATA ,
UCOS_DURATION
};
```

This data type exhaustively enumerates all attribute types.

```
enum RequirementMode
{
    MANDATORY , OPTIONAL
```

UNCLASSIFIED

```
};
```

This data type defines the requirements mode for an attribute and is used as a component of the AttributeInformation struct. The elements specify whether the attribute is optional or mandatory for purposes of creation of a product.

```
struct AttributeInformation
{
    string attribute_name;
    AttributeType attribute_type;
    Domain attribute_domain;
    string attribute_units;
    string attribute_reference;
    RequirementMode mode;
    string description;
    boolean sortable;
    boolean updateable;
};
```

```
typedef sequence < AttributeInformation >
AttributeInformationList;
```

This data type represents a set of characteristics that together describe an attribute. The syntax for the attribute names in field *attribute\_name* is defined in section 4.4.6. (Attribute Name Syntax Rule)

```
struct Association {
    string name;
    ViewName view_a;
    ViewName view_b;
    string description;
```

UNCLASSIFIED

```
UCO::Cardinality card;

AttributeInformationList attribute_info;

};

typedef sequence <Association> AssociationList;
```

These data types are used by the DataModelMgr to describe the relationships between views.

#### **2.2.2.10. ProductDetails, ProductDetailsList, DeliveryDetails, DeliveryDetailsList, OrderContents & QueryOrderContents**

```
struct ProductDetails {

    MediaTypeList mTypes;

    UCO::NameList benums;

    AlterationSpec aSpec;

    UID::Product aProduct;

    string info_system_name;

};

typedef sequence <ProductDetails> ProductDetailsList;

struct DeliveryDetails {

    Destination dests;

    string receiver;

    string shipmentMode;

};

typedef sequence < DeliveryDetails > DeliveryDetailsList;
```

```
struct OrderContents {  
    string originator;  
    TailoringSpec tSpec;  
    PackagingSpec pSpec;  
    UCO::AbsTime needByDate;  
    string operatorNote;  
    short orderPriority;  
    ProductDetailsList prod_list;  
    DeliveryDetailsList del_list;  
};
```

```
struct QueryOrderContents {  
    string originator;  
    TailoringSpec tSpec;  
    PackagingSpec pSpec;  
    string operatorNote;  
    short orderPriority;  
    AlterationSpec aSpec;  
    DeliveryDetailsList del_list;  
};
```

These data structures are used to describe the details of an order and a query (standing) order.

#### **2.2.2.11. AccessCriteria (j/NPS)**

```
struct AccessCriteria {
```

UNCLASSIFIED

```
string userID;  
string password;  
string licenseKey;  
};
```

The structure AccessCriteria contains the information used for access control of GIAS Library capabilities.

### **2.2.3. GIAS Simple Data Types (j/NPS)**

The GIAS defines a number of simple data types.

#### **2.2.3.1. LibraryList, RequestList, ManagerType, ManagerTypeList, UseModeList, and RsetList (j/NPS)**

```
typedef sequence < Library > LibraryList;  
typedef string ManagerType;  
typedef sequence < ManagerType > ManagerTypeList;  
typedef sequence < Request > RequestList;  
typedef sequence < UseMode > UseModeList;  
  
typedef sequence <short> RsetList;
```

The GIAS specification defines a number of convenience structures that are a sequence of other defined types. LibraryList contains a sequence of references of type Library. ManagerType is a string. ManagerTypeList contains an unbounded sequence of ManagerType. RequestList contains a sequence of references of type Request. RsetList contains a sequence of short integers. UseModeList contains a sequence of UseMode (UseMode is type String).

#### **2.2.3.2. LibraryDescription and LibraryDescriptionList**

```
struct LibraryDescription  
{
```

```
    string library_name;  
    string library_description;  
    string library_version_number;  
};  
  
typedef sequence < LibraryDescription >  
LibraryDescriptionList;
```

The *LibraryDescription* structure contains the name of a specific Library instance in the string *library\_name*. The string *Library\_description* contains a human readable description of the Library and its holdings. The *Library\_version\_number* provides a mechanism for clients to determine the version of the GIAS specification used by this specific library implementation. *LibraryDescriptionList* contains an unbounded sequence of *LibraryDescriptions*.

### 2.2.3.3. Query

```
struct Query {  
    ViewName view_name;  
    string    bqs_query;  
};
```

The data structure *Query* is composed of a query expression for a particular view of a given data model.

### 2.2.3.5. QueryResults

```
typedef UCO::DAGList QueryResults;
```

The *QueryResults* structure is used to contain a collection of results from a catalog query. Each individual result in this collection contains metadata that describes a data set or product and a reference to that data set or product in the form of a *Product*. The *QueryResults* structure re-uses the *DAGList* type defined in the UCO specification. The set of results from a catalog query is expressed in a *QueryResults* structure by applying the following rules:

- 1) Each result (catalog record or “hit”) consists of an identifier of a data set or product and a set of metadata elements. The identifier will be in the form of a *Product* reference for that data set. The metadata elements will each consist of an attribute name and a type and value for that attribute and the relationships among the metadata elements.

- 2) Each result is placed in its own *DAG*.
- 3) Each metadata element is placed in its own node by setting the *attribute\_name* of the node to reflect the name of the metadata element and by setting the *value* of the node to reflect the type and value of that metadata element.
- 4) The number, type and name for the relationships in a *DAG* are dependent on the data model that underlies the catalog that generated the result and are thus implementation dependent.

### 2.2.3.6. LifeEventType, LifeEvent, NamedEventType, Event, EventList, DayEvent, DayEventTime, LifeEventList, and QueryLifeSpan

```
enum LifeEventType
{
    ABSOLUTE_TIME ,
    DAY_EVENT_TIME ,
    NAMED_EVENT ,
    RELATIVE_TIME
};
```

```
union LifeEvent switch (LifeEventType)
{
    case ABSOLUTE_TIME: UCO::AbsTime at;
    case DAY_EVENT_TIME: DayEventTime day_event;
    case NAMED_EVENT: string ev;
    case RELATIVE_TIME: UCO::Time rt;
};
```

```
enum NamedEventType
```

```
{
    START_EVENT,
    STOP_EVENT,
    FREQUENCY_EVENT
};

struct Event {
    string event_name;

    NamedEventType event_type;

    string event_description;
};

typedef sequence < Event > EventList;

enum DayEvent { MON, TUE, WED, THU, FRI, SAT, SUN,
FIRST_OF_MONTH, END_OF_MONTH };

struct DayEventTime
{
    DayEvent          day_event;
    UCO::Time         time;
};

typedef sequence < LifeEvent > LifeEventList;
```

```
struct QueryLifeSpan {
    LifeEvent start;
    LifeEvent stop;
    LifeEventList frequency;
};
```

These data structures are used by a client when interacting with the StandingQueryMgr and QueryOrderMgr to identify and describe events that can be used to establish the lifetime of a standing query. These Managers can be used to establish the lifetime of a standing query and how frequently it is run, by setting these elements. They allow an event in the lifetime of a query to be defined as one of the following: 1) an absolute time (e.g., start on 12 Jan 99); 2) an event; or 3) a relative time. These types of life events can be used to describe the start, stop points, and frequency of a standing query. NB: Not all combinations of absolute, relative and event references with start, stop and frequency are meaningful.

### 2.2.3.7. Polarity, SortAttribute and SortAttributeList

These data structures are used to indicate the sorting preferences of query results.

```
enum Polarity { ASCENDING, DESCENDING };

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

typedef sequence < SortAttribute >
SortAttributeList;
```

### 2.2.3.8 DelayEstimate

```
struct DelayEstimate {  
    unsigned long time_delay;  
    boolean valid_time_delay};
```

This structure returns an approximate time delay (i.e., *time\_delay*) when *valid\_time\_delay* is true. *Time\_delay* is not valid when *valid\_time\_delay* is false and should be ignored by the client application.

### 2.2.3.9 PackageElement, PackageElementList, and DeliveryManifest

```
struct PackageElement {  
    UID::Product prod;  
    UCO::NameList files;  
};  
  
typedef sequence< PackageElement >  
PackageElementList;  
  
struct DeliveryManifest {  
    string package_name;  
    PackageElementList elements;  
};  
  
typedef sequence<DeliveryManifest>  
DeliveryManifestList;
```

These structures are used to describe the contents of a delivery. The DeliveryManifest contains the names of the package (*package\_name*) and a list of PackageElement structures. Each of these PackageElement structures describes an element included in the package. A PackageElement describes a package element by containing the UID::Product identifier and a list of the file names that make up that product as delivered.

### 2.2.3.10 CallbackID (j/NPS)

```
typedef string CallbackID;
```

The data type CallbackID is used as an identifier for an instance of a Callback. The specific details of this data type are found in the appropriate GIAS profile.

## 2.3. Interfaces

### 2.3.1. Library (j/NPS)

```
interface Library
{
    ManagerTypeList get_manager_types ()
        raises (UCO::ProcessingFault,
UCO::SystemFault);

    LibraryManager get_manager
        (in ManagerType manager_type,
        in AccessCriteria
access_criteria)
        raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

    LibraryDescription get_library_description ()
        raises (UCO::ProcessingFault,
UCO::SystemFault);

    LibraryDescriptionList get_other_libraries
        (in AccessCriteria access_criteria)
        raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
};
```

The *Library* interface serves as the starting point for any interaction with the rest of the Library. All capabilities of a library system are accessed through the Manager objects it supports. The *Library* interface

is the mechanism by which a client discovers and requests access to Manager objects. The operations defined in the *Library* interface are described in the following subsections.

### 2.3.1.1. `get_manager_types`

```
ManagerTypeList get_manager_types()

raises (UCO::ProcessingFault, UCO::SystemFault);
```

This selector operation allows a client to discover which Managers are supported by a particular GIAS library. A *ManagerTypeList* structure is returned from a successful invocation of this operation. The *ManagerTypeList* returned by this operation will contain the names of all Manager types supported by this implementation. The Manager names contained in this list are used with the *get\_manager\_types* operations defined below to specify the type of Manager desired.

### 2.3.1.2. `get_manager (j/NPS)`

```
LibraryManager get_manager

                (in ManagerType manager_type,

                in AccessCriteria

access_criteria)

                raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

This operation is a Request to be given access to a Manager object. The client supplies the type of Manager desired in *manager\_type* and supplies information used for access control in *access\_criteria*. (See the *get\_manager\_types* operations for details on determining acceptable values). A successful invocation will return a reference to an object of type *LibraryManager*. This reference should then be narrowed (cast) into a reference to an object of the specific Manager type requested in *manager\_type*. It can be assumed that all Manager types supported by a GIAS implementation are derived (inherited) from type *LibraryManager*. The client must know the correlation between the names given in the *ManagerTypeList* and the object type to which that corresponds. Subsequent calls to *get\_manager* by the same client will result in the return of the same instance of a Manager or a new instance that has exactly the same state as the first instance (i.e., the state of the Manager is persistent).. Also calls to *get\_manager* by different clients will always result in different instances of Managers being returned. That is, the library system will not force clients to share an instance of a Manager.

The standard exception identifier *UnknownManagerType* is returned by this operation if the client has supplied a value of *manager\_type* unknown or unsupported by this implementation. Supplying an unknown criteria in *access\_criteria* will result in the *BadAccessCriteria* standard exception identifier. Supplying an unacceptable value for an OPTIONAL attribute in *access\_criteria* will result in the *BadAccessValue* standard exception identifier. Supplying incorrect or unacceptable values for one or more MANDATORY attributes in *access\_criteria* will result in the NO\_PERMISSION system exception being returned. (See Appendix E for a list of other system exceptions)

### 2.3.1.3. `get_library_description`

```
LibraryDescription get_library_description()  
  
    raises (UCO::ProcessingFault,  
          UCO::SystemFault);
```

This selector operation returns some descriptive information about the Library. A successful invocation of this operation will return a populated *LibraryDescription* structure.

### 2.3.1.4. `get_other_libraries`

```
LibraryDescriptionList get_other_libraries  
  
    (in AccessCriteria access_criteria)  
  
    raises ( UCO::InvalidInputParameter,  
          UCO::ProcessingFault, UCO::SystemFault);
```

This selector operation returns some descriptive information about other Libraries known to this Library that are accessible to the requesting user. *access\_criteria* holds any identifying or access control information needed. A successful invocation of this operation will return an unbounded list of Library descriptions.

## 2.3.2. LibraryManager

```
interface LibraryManager  
  
{  
  
    UCO::NameList get_property_names ()  
  
    raises (UCO::ProcessingFault, UCO::SystemFault);  
  
    PropertyList get_property_values  
  
        (in UCO::NameList desired_properties)  
  
        raises ( UCO::InvalidInputParameter,  
              UCO::ProcessingFault, UCO::SystemFault);
```

```

    LibraryList get_libraries ()

        raises (UCO::ProcessingFault,
UCO::SystemFault);

};

```

The *LibraryManager* interface serves as the (abstract) root for all types of Manager objects in the GIAS definition. 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 GIAS. Because these operations are common to all Manager types, a client can use these common operations to interact with Managers of unfamiliar type.

The operations defined in the *LibraryManager* interface are described in the following subsections.

### 2.3.2.1. get\_property\_values

```

PropertyList get_property_values

    (in UCO::NameList desired_properties)

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

```

This operation allows a client to discover the properties and the current values of those properties that describe a Manager. A client supplies the names of the properties of interest in the *NameList desired\_properties*. A successful invocation of this operation returns a *PropertyList*, which contains the current values of the requested properties. The *PropertyList* will contain one *NameValue* pair for each element supplied in the *NameList desired\_properties*. The *name* in that *NameValue* pair will be the name as specified in *desired\_properties*. The *value* associated with that *name* will be the current value of that property. The specific set of properties supported by a Manager is defined in the appropriate GIAS profile.

The standard exception identifier *UnknownProperty* will be returned if the client has supplied one or more properties unknown or unsupported by this Manager..

### 2.3.2.2. get\_libraries

```

LibraryList get_libraries ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);

```

This selector operation allows a client to determine which GIAS-based Library system(s) this Manager supports. A successful invocation of this operation will return a *LibraryList* structure. This structure will contain an object reference of type *Library* for each Library this Manager supports. There will always be at least one Library object reference in this list.

### 2.3.2.3. `get_property_names`

```
UCO::NameList get_property_names ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);
```

This selector operation allows a client to obtain a list of property names. A property name is the *name* component of a *NameValue* pair. The *NameList* returned by this selector operation identifies all the property names supported or known by this Manager.

### 2.3.3. RequestManager (j/NPS)

```
interface RequestManager:
{
    RequestList get_active_requests ()

    raises ( UCO::ProcessingFault, UCO::SystemFault);

    void set_default_timeout (in unsigned long
new_default)

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

    unsigned long get_default_timeout ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);

    void set_timeout (in Request aRequest,
in unsigned long new_lifetime)
```

```
        raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

    unsigned long get_timeout (in Request aRequest)

        raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

    void delete_request (in Request aRequest)

        raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

};
```

The *RequestManager* interface serves to define operations common to all Managers that use Request objects as part of their operations. This interface is abstract. Also, these common operations allow a client to interact with unfamiliar forms of *RequestManagers*. The operations defined on *RequestManager* serve to allow clients to identify active Requests and control their lifetimes.

Each Request being managed by a *RequestManager* has a limited lifetime. This lifetime is considered to begin when the processing it represents reaches the COMPLETE state and ends when the timeout set for that particular Request has elapsed. After a Request's lifetime has expired a *RequestManager* is free to (but is not required to) delete that Request as well as all resources associated with that *Request*.

The operations defined in the *RequestManager* interface are described in the following subsections.

### 2.3.3.1. get\_active\_requests (j/NPS)

```
RequestList get_active_requests ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);
```

This operation allows a client to determine what Requests are being managed by this *RequestManager*. A successful invocation of this operation will return a *RequestList* structure. This structure will contain an object reference of type *Request* for each Request currently being managed by this *RequestManager*.

### 2.3.3.2. set\_default\_timeout

```
void set_default_timeout (in unsigned long
new_default)

raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

This operation allows a client to set a default value (in seconds) of the lifetime of the Requests being managed by this *RequestManager*. The client supplies the desired lifetime in *new\_default*. Following successful invocation of this operation, all new Requests managed by this *RequestManager* will have a lifetime of *new\_default* seconds. This operation has no effect on the lifetime of Requests that already exist at the time of invocation of this operation.

The standard exception identifier *ImplementationLimit* will be returned if the client attempts to set a default lifetime that exceeds the maximum lifetime supported by this *RequestManager* implementation. The value of this maximum is implementation dependent and may vary over time.

#### 2.3.3.3. get\_default\_timeout

```
unsigned long get_default_timeout ()

raises (UCO::ProcessingFault,
UCO::SystemFault);
```

This operation allows a client to determine the current default lifetime for Requests initiated by this *RequestManager*. Successful invocation of this operation will return the current default lifetime of Requests in seconds.

#### 2.3.3.4. set\_timeout

```
void set_timeout (in Request aRequest,
in unsigned long new_lifetime)

raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

This operation allows a client to modify the currently set value for the lifetime of a Request. The client supplies the Request that is to have its lifetime modified in *aRequest* and the desired value of its new lifetime in *new\_lifetime*. Following successful invocation of this operation, the lifetime of Request *aRequest* will be

*new\_lifetime* seconds. If the Request *aRequest* has not reached a COMPLETE state, the lifetime will be *new\_lifetime* seconds beginning from the time it reaches the COMPLETE state. If Request *aRequest* is already in the COMPLETE state when this operation is invoked (that is a portion of its lifetime has already elapsed), the lifetime of Request *aRequest* will be *new\_lifetime* seconds beginning from the time the *set\_timeout* operation successfully completes.

The standard exception identifier *UnknownRequest* will be returned if the client has supplied a Request unknown to this instance of *RequestManager*. The standard exception identifier *ImplementationLimit* will be returned if the client attempts to set a default lifetime that exceeds the maximum lifetime supported by this *RequestManager* implementation. The value of this maximum is implementation dependent and may vary over time.

### 2.3.3.5. delete\_request (j/NPS)

```
void delete_request (in Request aRequest)

    raises ( UCO::InvalidInputParameter,
            UCO::ProcessingFault, UCO::SystemFault);
```

This operation allows a client to destroy a Request and free all resources associated with that Request. A client supplies the Request to be destroyed in *aRequest*. Following successful invocation of this operation, the *RequestManager* is free to (but is not required to) immediately destroy Request *aRequest* and to free all resources associated with that Request.

The standard exception identifier *UnknownRequest* will be returned if the client has supplied a Request unknown to this instance of *RequestManager*. After the *RequestManager* has destroyed the Request, attempts to invoke operations on that Request will return the *OBJECT\_NOT\_EXIST* system exception.

### 2.3.3.6. get\_timeout

```
unsigned long get_timeout (in Request aRequest)

    raises ( UCO::InvalidInputParameter,
            UCO::ProcessingFault, UCO::SystemFault);
```

The selector operation *get\_timeout* provides the client with the amount of time that remains on Request *aRequest* before the *RequestManager* deletes the Request.

## 2.3.4. AccessManager

```
interface AccessManager:RequestManager
```

```
{
  UseModeList get_use_modes ()
      raises (UCO::ProcessingFault,
UCO::SystemFault);

  boolean is_available (in UID::Product product,
                        in UseMode use_mode)
      raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

  // Returns the time (in seconds) estimated to put the
  // requested product into the requested UseMode.
  // DOES NOT request a change in the availability of
  // the product.

  unsigned long query_availability_delay
      (in UID::Product product,
       in AvailabilityRequirement
availability_requirement,
       in UseMode use_mode)
      raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

  short get_number_of_priorities()
      raises (UCO::ProcessingFault,
UCO::SystemFault);

  SetAvailabilityRequest set_availability
      (in UID::ProductList products,
```

```

        in AvailabilityRequirement
        availability_requirement,

        in UseMode use_mode,

        in short priority)

        raises ( UCO::InvalidInputParameter,
        UCO::ProcessingFault, UCO::SystemFault);

};

```

The *AccessManager* is an abstract interface that serves to define operations common to Managers that allow clients to determine and control the “availability” of a data set or product. “availability” is defined as the readiness of a data set or product to be used by the other operations on the Manager. An *AccessManager* describes “availability” by defining one or more *UseModes*. A *UseMode* is a state or condition of a data set or product that indicates its readiness to be used by the *AccessManager* for a specific purpose.

The operations defined in the *AccessManager* are described in the following subsections.

#### 2.3.4.1. get\_use\_modes

```

UseModeList get_use_modes ()

        raises (UCO::ProcessingFault,
        UCO::SystemFault);

```

This operation allows a client to discover the *UseModes* supported by this *AccessManager*. A successful invocation of this operation returns a *UseModeList* containing all of the *UseModes* supported or known to this *AccessManager*.

#### 2.3.4.2. is\_available

```

boolean is_available (in UID::Product product, in
UseMode use_mode)

        raises ( UCO::InvalidInputParameter,
        UCO::ProcessingFault, UCO::SystemFault);

```

This operation allows a client to determine whether a data set or product is ready for a specific purpose. A client indicates the data set or product of interest and its desired use by supplying both a reference of type *Product* in *product* and its intended use as a *UseMode* in *use\_mode*. A successful invocation of this operation will return a boolean that indicates whether or not the requested data set or product is currently available for the requested use. A boolean value of "TRUE" indicates the product is available. A boolean value of FALSE indicates that the product is not currently available for the requested use. This operation does **not** affect the current availability of the requested data set or product.

The standard exception identifier *UnknownProduct* will be returned if the client supplied a product reference unknown to this *AccessManager*. The standard exception identifier *UnknownUseMode* will be returned if the client supplied a *UseMode* unknown or unsupported by this *AccessManager*. The standard exception identifier *BadUseMode* is returned if the client supplied a *UseMode* that is inappropriate or unsupported for the particular data set or product supplied in *product*.

#### 2.3.4.3. get\_number\_of\_priorities

```
short  get_number_of_priorities()

        raises (UCO::ProcessingFault,
UCO::SystemFault);
```

This operation returns the number of priority levels this *OrderMgr* recognizes. Priorities are ordered from 1 (one) to N, where 1 is the highest priority and N the lowest. This operation returns the N for this *AccessManger*.

#### 2.3.4.4. set\_availability

```
SetAvailabilityRequest set_availability

        (in UID::ProductList products,

         in AvailabilityRequirement
availability_requirement,

         in UseMode use_mode,

         in short priority)

        raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

This operation allows a client to submit a Request to make one or more products available for a specific purpose or to indicate that the products are no longer needed for the specific purpose. A client indicates data sets or products of interest and their desired use by supplying both a list of references of type *Product*

in the productList *products* and its intended use as a *UseMode* in *use\_mode*. The client also includes a parameter of type *AvailabilityRequirement*. If that parameter holds the value REQUIRED it indicates the client wishes to have the products put into the requested UseMode. If the parameter holds the value NOT\_REQUIRED, it indicates the client no longer needs the products in that mode and the server is free to remove it from that mode. The client also supplies a priority as a short in the parameter *priority*.

The standard exception identifier *UnknownProduct* will be returned if the client supplied one or more product references unknown to this *AccessManager*. The standard exception identifier *UnknownUseMode* will be returned if the client supplied a *UseMode* unknown or unsupported by this *AccessManager*. The standard exception identifier *BadUseMode* is returned if the client supplied a *UseMode* that is inappropriate or unsupported for the particular data set or product supplied in *product*.

The *BadUseMode* standard exception identifier will also occur if the requested data set or product can never be made available in the requested *UseMode*.

#### 2.3.4.5. query\_availability\_delay

```

unsigned long query_availability_delay (in
    UID::Product product,
                                     in AvailabilityRequirement
                                     availability_requirement,
                                     in UseMode use_mode)
    raises ( UCO::InvalidInputParameter,
            UCO::ProcessingFault, UCO::SystemFault);

```

This selector operation allows a client to get an estimate of the time in seconds for a product to be placed in the requested mode. Invocation of the operation does NOT submit a Request to actually place the product in the specified mode, it only returns an estimate of the time required to do so. The parameters and standard exception identifiers are identical to those of *set\_availability* defined above.

#### 2.3.5. OrderMgr

```

interface OrderMgr:LibraryManager, AccessManager
{
    UCO::NameList get_package_specifications()
        raises (UCO::ProcessingFault,
                UCO::SystemFault);
}

```

```
ValidationResults validate_order
    (in OrderContents order,
     in PropertyList properties)
    raises ( UCO::InvalidInputParameter,
            UCO::ProcessingFault, UCO::SystemFault);

OrderRequest order (in OrderContents order,
                    in PropertyList properties)
    raises ( UCO::InvalidInputParameter,
            UCO::ProcessingFault, UCO::SystemFault);
};
```

The *OrderMgr* allows a client to submit orders for data sets or products from a GIAS Library. The *OrderMgr* provides an operation to validate an order specification prior to submitting the order to a GIAS Library. Both operations on this Manager re-use the *OrderContents* structure to describe an order. The operations defined in the *OrderMgr* interface are described in the following subsections.

### 2.3.5.1. get\_package\_specifications

```
UCO::NameList get_package_specifications()
    raises (UCO::ProcessingFault,
            UCO::SystemFault);
```

This operation returns a *NameList* containing all packaging specifications known or acceptable to this *OrderMgr*. These packaging specifications are used as values for the element *packaging\_format\_and\_compression* in *PackagingSpec* structures submitted in orders.

### 2.3.5.2. validate\_order

```
ValidationResults validate_order
```

```
(in OrderContents order,  
    in PropertyList properties)  
  
    raises ( UCO::InvalidInputParameter,  
            UCO::ProcessingFault, UCO::SystemFault);
```

The operation *validate\_order* is invoked to determine if an order Request for a data set or product from a GIAS 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*.

### 2.3.5.3. order

```
OrderRequest order (in OrderContents order,  
    in PropertyList properties)  
  
    raises ( UCO::InvalidInputParameter,  
            UCO::ProcessingFault, UCO::SystemFault);
```

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

### 2.3.6. DataModelMgr

```
interface DataModelMgr:LibraryManager  
  
    {  
  
        UCO::AbsTime get_data_model_date (in PropertyList  
            properties)  
  
            raises ( UCO::InvalidInputParameter,  
                    UCO::ProcessingFault, UCO::SystemFault);  
  
        UCO::NameList get_alias_categories(in PropertyList  
            properties)
```

```
    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

UCO::NameNameList get_logical_aliases(in string
category, in PropertyList properties)

    raises( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

    string get_logical_attribute_name (in ViewName
view_name, in ConceptualAttributeType attribute_type,
in PropertyList properties)

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

    ViewList get_view_names (in PropertyList
properties)

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

AttributeInformationList get_attributes (in ViewName
view_name,

                                in PropertyList
properties)

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

AttributeInformationList get_queryable_attributes

                                (in ViewName view_name,

                                in PropertyList
properties)

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

```
UCO::EntityGraph get_entities (in ViewName view_name,
                                in PropertyList properties)
    raises ( UCO::InvalidInputParameter,
            UCO::ProcessingFault, UCO::SystemFault);

AttributeInformationList get_entity_attributes
    (in Entity aEntity,
     in PropertyList properties)
    raises ( UCO::InvalidInputParameter,
            UCO::ProcessingFault, UCO::SystemFault);

AssociationList get_associations(in PropertyList
properties);

    raises ( UCO::InvalidInputParameter,
            UCO::ProcessingFault, UCO::SystemFault);

    unsigned short get_max_vertices(in PropertyList
properties)
    raises ( UCO::InvalidInputParameter,
            UCO::ProcessingFault, UCO::SystemFault);

};
```

The *DataModelMgr* allows a client to discover and access the data model being used by the Library. This capability allows a client to be constructed without “hard-coding” a specific data model into its design. The *DataModelMgr* interface operations can be partitioned into two sets: access to ancillary data and access to the data model itself. The ancillary set of selector operations provides the following:

- the last date and time the data model was updated (*get\_data\_model\_date*);
- a list of communities that define their own set of aliases data model attribute names (*get\_alias\_categories*);
- the aliases defined by a specific community (*get\_logical\_aliases*) and

- the logical attribute names that are the equivalent of the *ConceptualAttributeType* (*get\_logical\_attribute\_name*)

The data model set of selector operations provides the following descriptions and associated interface operations:

- the set of data views known by the Library (*get\_view\_names*);
- the set of attributes that describe a specific data view (*get\_attributes*);
- the subset of attributes of a data view which are queryable (*get\_queryable\_attributes*);
- the set of entities that compose a specific data view (*get\_entities*);
- the set of attributes for a specific *Entity* (*get\_entity\_attributes*);
- the set of associations available for use among views (*get\_associations*) and
- the maximum number of vertices supported in a geospatial query (*get\_max\_vertices*)

The operations defined for the *DataModelMgr* interface are described in the subsections below.

### 2.3.6.1. *get\_data\_model\_date*

```
UCO::AbsTime get_data_model_date (in PropertyList
properties)

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

This selector operation returns the last date the Library's data model was updated.

### 2.3.6.2. *get\_alias\_categories*

```
UCO::NameList get_alias_categories(in PropertyList
properties)

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

This selector operation returns a *NameList* containing user communities known to this Library. A user community contained in the *NameList* is used as a parameter for the selector operation *get\_logical\_aliases*.

### 2.3.6.3. *get\_logical\_aliases*

```
UCO::NameNameList get_logical_aliases(in string
category, in PropertyList properties)
```

```
    raises(UCO::InvalidInputParameter,  
          UCO::ProcessingFault, UCO::SystemFault);
```

This selector operation returns a *NameNameList*, which contains a mapping of a specific community's aliases to names based on the logical data model of the Library. The syntax for the attribute names contained in the *NameNameList* is defined in section 4.4.6. (Attribute Name Syntax Rule)

The standard exception identifier *UnknownCategory* is returned if the category requested is unknown or unsupported by this *DataModelMgr*.

#### 2.3.6.4. **get\_logical\_attribute\_name**

```
string get_logical_attribute_name (in ViewName  
view_name,in ConceptualAttributeType attribute_type,  
in PropertyList properties)  
  
    raises ( UCO::InvalidInputParameter,  
          UCO::ProcessingFault, UCO::SystemFault);
```

This selector operation returns the name of the logical attribute that is the equivalent of the requested *ConceptualAttributeType* in the requested view *view\_name*. The syntax for the attribute names contained in the string returned is defined in section 4.4.6. (Attribute Name Syntax Rule)

#### 2.3.6.5. **get\_view\_names**

```
ViewList get_view_names (in PropertyList properties)  
  
    raises ( UCO::InvalidInputParameter,  
          UCO::ProcessingFault, UCO::SystemFault);
```

This selector operation returns a *DAG* structure which provides a hierarchy of data views supported by the Library for use by the client. The *DAG* is composed of a hierarchical set of nodes, where each node contains a string identifying a data view.

#### 2.3.6.6. **get\_attributes**

```
AttributeInformationList get_attributes (in ViewName  
view_name,in PropertyList properties)
```

```
        raises ( UCO::InvalidInputParameter,  
                UCO::ProcessingFault, UCO::SystemFault);
```

This selector operation returns an *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. The syntax for the attribute names contained in the *AttributeInformation* structure is defined in section 4.4.6. (Attribute Name Syntax Rule)

The standard exception identifiers raised by this operation denote an invocation that submits one or more parameters that provide an *UnknownViewName*, *UnknownProperty* or *BadPropertyValue*.

### 2.3.6.7. *get\_queryable\_attributes*

```
AttributeInformationList get_queryable_attributes  
                        (in ViewName view_name,  
                        in PropertyList  
properties)  
  
        raises ( UCO::InvalidInputParameter,  
                UCO::ProcessingFault, UCO::SystemFault);
```

This selector 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. The syntax for the attribute names contained in the *AttributeInformation* structure is defined in section 4.4.6. (Attribute Name Syntax Rule)

The standard exception identifiers raised by this operation denote an invocation that submits one or more parameters that provide an *UnknownViewName*, *UnknownProperty* or *BadPropertyValue*.

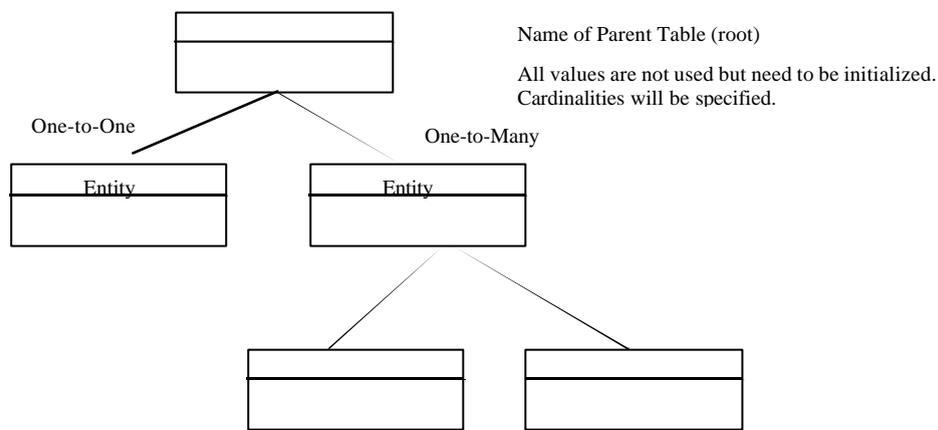
### 2.3.6.8. *get\_entities*

```
UCO::EntityGraph get_entities (in ViewName view_name,  
                               in PropertyList properties)  
  
        raises ( UCO::InvalidInputParameter,  
                UCO::ProcessingFault, UCO::SystemFault);
```

This selector operation returns an *EntityGraph*, which represents a set of entities and their relationships that compose a specific data view. Note that the cardinality is defined within the Edge structure of the graph.

The standard exception identifiers raised by this operation denote an invocation that submits one or more parameters that provide an UnknownViewNameUnknownPropertyor BadPropertyValue.

**DAG: Entities**



Note: All nodes of type Entity\_Node

**Figure 2-1 Structure of Data View DAG**

**2.3.6.9. get\_entity\_attributes**

```

AttributeInformationList get_entity_attributes
    (in Entity aEntity,
     in PropertyList properties)
    raises ( UCO::InvalidInputParameter,
            UCO::ProcessingFault, UCO::SystemFault);
    
```

This selector operation returns an `AttributeInformationList`, which represents a set of attributes that describes a specific entity. The `AttributeInformationList` contains elements of type `AttributeInformation`. The syntax for the attribute names contained in the `AttributeInformation` structure is defined in section 4.4.6. (Attribute Name Syntax Rule)

The standard exception identifiers raised by this operation denote an invocation that submits one or more parameters that provide an `UnknownViewNameUnknownProperty` or `BadPropertyValue`.

#### 2.3.6.10. `get_associations`

```
AssociationList get_associations(in PropertyList
properties);

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

This selector operation returns a list of `Association` structures that contains the descriptions of the associations that are used by the `DataModelMgr`.

#### 2.3.6.11. `get_max_vertices`

```
unsigned short get_max_vertices(in PropertyList
properties);

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

This selector operation returns the maximum number of vertices supported in geospatial queries.

### 2.3.7. `StandingQueryMgr`

```
interface StandingQueryMgr:LibraryManager,
RequestManager

{

    EventList get_event_descriptions()

raises ( UCO::ProcessingFault, UCO::SystemFault);

    SubmitStandingQueryRequest submit_standing_query (
```

```

        in Query aQuery,
        in UCO::NameList result_attributes,
        in SortAttributeList sort_attributes,
        in QueryLifeSpan lifespan,
        in PropertyList properties)

        raises ( UCO::InvalidInputParameter,
        UCO::ProcessingFault,
        UCO::SystemFault);

};

```

The *StandingQueryMgr* allows a client to place a query with a Library that will monitor the Library for new products arriving in the Library and notify the requester.

### 2.3.7.1. get\_event\_descriptions

```

EventList get_event_descriptions()

        raises (UCO::ProcessingFault,
        UCO::SystemFault);

```

This selector operation returns a list of events that can be used by the client in the lifespan parameter of *submit\_standing\_query* to set the details of the lifetime of a standing query such as start, duration, and end.

### 2.3.7.2. submit\_standing\_query

```

SubmitStandingQueryRequest submit_standing_query (

        in Query aQuery,

```

```

in UCO::NameList result_attributes,
in SortAttributeList sort_attributes,
in QueryLifeSpan lifespan,
in PropertyList properties)

raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault,
UCO::SystemFault);

```

This operation allows a client to establish a standing query, that is, a query that is run repeatedly for a set period of time rather than simply once. The parameters, semantics and standard exception identifiers for this operation are the same as for a single query. (see the `submit_query` operation of the `CatalogMgr`). In addition to those parameters, the client specifies the period of time the query is to be run in the parameter *lifespan*. The standard exception identifier `InvalidEvent` is returned if an event contained in *lifespan* is inappropriate or unknown. The standard exception identifier `ImplementationLimit` is returned if a submitted value in the `QueryLifeSpan` parameter exceeds this Manager's capabilities. These limits are implementation specific.

### 2.3.8. CreationMgr (j/NPS)

```

interface CreationMgr:LibraryManager, RequestManager
{

CreateRequest create (in UCO::FileLocationList
new_product, in RelatedFileList related_files, in
UCO::DAG creation_metadata,
in PropertyList properties)

raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

CreateMetaDataReader create_metadata (in UCO::DAG
creation_metadata, in ViewName view_name, in
RelatedFileList related_files, in PropertyList
properties)

```

```

        raises ( UCO::InvalidInputParameter,
                UCO::ProcessingFault, UCO::SystemFault);

```

```

CreateAssociationRequest create_association( in string
assoc_name,

        in UID::Product view_a_object,

        in UID::ProductList view_b_objects,

        in UCO::NameValueList assoc_info)

        raises ( UCO::InvalidInputParameter,
                UCO::ProcessingFault, UCO::SystemFault);

};

```

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. The operations defined in the *CreationMgr* interface are described in the following subsections.

### 2.3.8.1. create (j/NPS)

```

CreateRequest create (in UCO::FileLocationList
new_product, in RelatedFileList related_files, in
UCO::DAG creation_metadata, in PropertyList
properties)

        raises ( UCO::InvalidInputParameter,
                UCO::ProcessingFault, UCO::SystemFault);

```

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 client nominates a data set or product by supplying a *FileLocationList* *new\_product* that points to the data set or product being nominated. The client also indicates any related files that accompany this product by specifying them

in the parameter *related\_files*. The metadata that must accompany this nomination may be supplied in one of two ways: 1) the file at the location *new\_product* contains the data set **and all** the appropriate metadata 2) the file at location *new\_product* contains the data set **and some** (to include none) of the metadata and *creation\_metadata* contains the remainder of the appropriate metadata. If the first method of metadata submission is chosen a NULL value is supplied for *creation\_metadata*. All metadata for the nominated product is then expected to be in file at location *new\_product*. If a non-NULL value is supplied for *creation\_metadata* this indicates that the second metadata submission method has been chosen and that the metadata for the nominated product is to be found in the file at location *new\_product* **and** in the *DAG* *creation\_metadata*. If the same metadata element appears in both the file and in the *DAG*, the value appearing in the *DAG* takes precedence and will be used for the nomination. Note that it is implementation dependent whether the server “edits” products submitted with conflicting metadata i.e a server may choose NOT to edit these products and thus serve out products with metadata that doesn’t match the metadata in the catalog. The definition of the metadata elements (their names and acceptable values or ranges, whether mandatory or optional and their mapping into and out of various file formats that may be nominated) to be described in the file or in the *DAG* are defined in the appropriate GIAS profile. The client also describes any properties that further refine, effect or amplifies this Request by supplying their names and values in the *PropertyList* *properties*. (The properties that are available or applicable to this operation are defined in the appropriate GIAS profile.) A successful invocation of this operation will return a reference to a *CreateRequest* object.

The standard exception identifier *BadLocation* will be returned if the client supplies a location description which is syntactically invalid, incomplete or specifies a location unknown or inaccessible by the *CreationMgr*. This does **not** require the *CreationMgr* to determine the validity of the *user\_name* - password combination specified in *location* or the availability of space at *location* to return successfully. The standard exception identifier *UnknownCreationAttribute* will be returned if the client has supplied a metadata element in the *DAG* *creation\_metadata* that is unknown or unsupported by this *CreationMgr*. Note that a server will ignore unknown attributes in a nominated file. The standard exception identifier *BadCreationAttributeValue* will be returned if the client supplies a metadata element, whether in a file or in the *DAG* *creation\_metadata*, with an inappropriate or invalid value.. The standard exception identifier *UnknownProperty* will be returned if the client has supplied one or more properties unknown or unsupported by this *CreationMgr*. The standard exception identifier *BadPropertyValue* is returned if the client has supplied one or more values for properties which are inappropriate or exceed the allowed or expected values of that property.

### 2.3.8.2. create\_metadata

```
CreateMetaDataRequest create_metadata (in UCO::DAG
creation_metadata, in ViewName view_name, in
RelatedFileList related_files, in PropertyList
properties)

raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

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 *DAG creation\_metadata*. The client also indicates which view this metadata pertains to by supplying the name of the relevant view in *view\_name*. The client also supplies any related files by providing a RelatedFileList which contains the file location and the file relationship. (See section 2.2.2.8 for the description of the RelatedFileList) The client also describes any properties that further refine, effect or amplifies this Request by supplying their names and values in the PropertyList *properties*. (The properties that are available or applicable to this operation are defined in the appropriate GIAS profile.) A successful invocation of this operation will return a reference to a CreateMetaDataReader object.

The standard exception identifier *UnknownCreationAttribute* will be returned if the client has supplied a metadata element in the *DAG creation\_metadata* that is unknown or unsupported by this *CreationMgr*. The standard exception identifier *BadCreationAttributeValue* will be returned if the client supplies a metadata element in the *DAG creation\_metadata* with an inappropriate or invalid value. The standard exception identifier *UnknownProperty* will be returned if the client has supplied one or more properties unknown or unsupported by this *CreationMgr*. The standard exception identifier *BadPropertyValue* is returned if the client has supplied one or more values for properties which are inappropriate or exceed the allowed or expected values of that property.

### 2.3.8.3. create\_association

```
CreateAssociationRequest create_association( in string
assoc_name,

                                     in UID::Product view_a_object,

                                     in UID::ProductList view_b_objects,

                                     in UCO::NameValueList assoc_info)

                                     raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

This operation allows a client to create an association of a specified type between a set of Products that exist in a Library. The client identifies the desired association in *assoc\_name*, the Product which has this association in *view\_a\_object* and the Product(s) to be associated with the *view\_a\_object* in the ProductList *view\_b\_objects* and the metadata that describes this association in *assoc\_info*. This method will return a CreateAssociationRequest reference which can be used to monitor the status of this Request.

The standard exception identifier *InvalidCardinality* will be raised if the number of Products in the ProductList *view\_b\_objects* does not match the cardinality of the association *assoc\_name*. The standard exception identifier *UnknownAssociation* will be raised if the value of *assoc\_name* is unknown to the *CreationMgr*. The standard exception identifier *InvalidObject* will be raised if one or more of the Products identified are inappropriate for the association requested.

### 2.3.9. UpdateMgr

```
interface UpdateMgr: LibraryManager, RequestManager
{
void set_lock(in UID::Product lockedProduct)
           raises (
UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

    UpdateRequest update (in ViewName view, in
UCO::UpdatedDAGList changes, in RelatedFileList
relfiles, in PropertyList properties)

           raises( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

UpdateByQueryRequest update_by_query(in UCO::NameValue
updated_attribute,

                                     in Query bqs_query,

                                     in PropertyList
properties)

           raises( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

void release_lock(in UID::Product lockedProduct)

           raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

void delete_product(in UID::Product prod)
           raises( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

};
```

UNCLASSIFIED

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

### 2.3.9.1. set\_lock

```
void set_lock(in UID::Product lockedProduct)
    raises( UCO::InvalidInputParameter,
           UCO::ProcessingFault, UCO::SystemFault);
```

This operation locks a Product to allow it to be safely updated. The Product reference for the Product to be locked is provided in the parameter *lockedProduct*. Attempts to lock a Product which is already locked will generate a LockUnavailable standard exception identifier.

### 2.3.9.2. update

```
UpdateRequest update (in ViewName view, in
                     UCO::UpdatedDAGList changes, in RelatedFileList
                     relfiles, in PropertyList properties)
    raises( UCO::InvalidInputParameter,
           UCO::ProcessingFault, UCO::SystemFault);
```

This operation allows a client to modify existing catalog entries. The entries to be modified are first retrieved via the ProductMgr::get\_parameters operation. (see section 2.3.11. ProductMgr) The desired modifications are described by providing the type of view that is being updated along with an UpdatedDAGList that contains the entries with the modified values. (see the UCOS specification for the design of the UpdateDAGList) The UpdateDAGList, containing the new entries, is provided in the parameter *changes*. The client also supplies any updated related files in *relfiles*. Note that in the case where related files are being updated only one catalog entry (Product) can be updated per invocation. The client also describes any properties that further refine, effect or amplifies this Request by supplying their names and values in the *PropertyList properties*. (The properties that are available or applicable to this operation are defined in the appropriate GIAS profile.) A successful invocation of this operation will return a reference to an *UpdateRequest* object. A successful invocation of this operation also releases the lock on the updated catalog entries.

The standard exception identifier NonUpdateableAttribute is returned if the client attempts to modify a non-updateable attribute. The standard exception identifier UnsafeUpdate is returned if the client attempts to update entries that are not locked. The standard exception identifier ProductLocked is returned if the client attempts to update entries that are locked by another client.

### 2.3.9.3 update\_by\_query

```

UpdateByQueryRequest update_by_query(in UCO::NameValue
updated_attribute,

                                     in Query bqs_query,

                                     in PropertyList properties)

    raises( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

```

This operation allows a client to modify existing catalog entries that match a specific query. The entries to be modified are defined as those that match the query defined by the parameter *bqs\_query*. The desired modifications are described by providing a name value pair (*updated\_attribute*) that contains the name of the attribute to be changed and its new value. The client also describes any properties that further refine, effect or amplifies this Request by supplying their names and values in the *PropertyList properties*. (The properties that are available or applicable to this operation are defined in the appropriate GIAS profile.) A successful invocation of this operation will return a reference to an *UpdateByQueryRequest* object.

The standard exception identifier *NonUpdateableAttribute* is returned if the client attempts to modify a non-updateable attribute. The standard exception identifier *BadUpdateAttribute* is returned if the attribute in *updated\_attribute* is unknown. The standard exception identifier *LockUnavailable* is returned if the items to be modified cannot be safely locked prior to modification. The standard exception identifier *UnknownViewName* is returned if the data view specified is unknown. The standard exception identifier *BadQuery* is returned if the query specified is malformed. The standard exception identifier *BadQueryAttribute* is returned if one or more of the attributes in the query is unknown. The standard exception identifier *BadQueryValue* is returned if one or more of the attributes in the query have an inappropriate value. The standard exception identifier *UnknownProperty* is returned if one or more of the properties specified in *properties* is unknown. The standard exception identifier *BadPropertyValue* is returned if one or more of the values of properties specified in *properties* is inappropriate.

#### 2.3.9.4. release\_lock

```

void release_lock(in UID::Product lockedProduct)

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

```

This operation manually releases a lock that has been placed on a Product. The Product reference for the locked Product is provided in the parameter *lockedProduct*. Attempts to release a lock on a Product which is not locked will be silently ignored.

#### 2.3.9.5. delete\_product

```
void delete_product(in UID::Product prod)

    raises( UCO::InvalidInputParameter,
           UCO::ProcessingFault, UCO::SystemFault);
```

This operation is used to delete a Product. Attempts to delete a Product which is locked will raise the ProductLocked standard exception identifier.

### 2.3.10. CatalogMgr

```
interface CatalogMgr:LibraryManager, RequestManager

{

SubmitQueryRequest submit_query (

    in Query aQuery,

    in UCO::NameList result_attributes,

    in SortAttributeList sort_attributes,

    in PropertyList properties)

    raises ( UCO::InvalidInputParameter,
           UCO::ProcessingFault, UCO::SystemFault);

HitCountRequest hit_count (in Query aQuery, in
PropertyList properties)

raises ( UCO::InvalidInputParameter,
        UCO::ProcessingFault, UCO::SystemFault);

raises ( UCO::InvalidInputParameter,
        UCO::ProcessingFault, UCO::SystemFault);

};
```

UNCLASSIFIED

The *CatalogMgr* allows a client to submit queries to search the catalog of holdings of a GIAS Library. The operations defined in the *CatalogMgr* interface are described in the following subsections.

### 2.3.10.1. submit\_query

```
SubmitQueryRequest submit_query (  
    in Query aQuery,  
    in UCO::NameList result_attributes,  
    in SortAttributeList sort_attributes,  
    in PropertyList properties)  
    raises ( UCO::InvalidInputParameter,  
            UCO::ProcessingFault, UCO::SystemFault);
```

This operation allows a client to submit a query to search a catalog of products (The specific data views available and acceptable to a *CatalogMgr* are available through the *DataModelMgr*.) The query, which defines the selection criteria for the products of interest as well as the view of interest, is defined by the Query *aQuery*. The format of this query is defined by the Boolean Query Syntax (BQS) (See chapter 4). The client indicates the attributes desired in the results in the *NameList* *result\_attributes*. The client also indicates any desired sorting by including a *SortAttribute* for each attribute to be sorted in the element *sort\_attributes*. The format for the attributes used in the query, result attributes and sort attributes is the same and are defined by application of a rule defined in Chapter 4. The client also describes any properties that further refine, effect or amplifies this Request by supplying their names and values in the *PropertyList* *properties*. (The properties that are available or applicable to this operation are defined in the appropriate GIAS profile.) A successful invocation of this operation will return a reference to a *SubmitQueryRequest* object.

The standard exception identifier *UnknownViewName* will be returned if the client has supplied a data view unknown or unsupported by this *CatalogMgr*. The standard exception identifier *BadQuery* will be returned if the Query specified by *aQuery* is syntactically invalid. The standard exception identifier *BadQueryAttribute* will be returned if the query contains an attribute unknown to the *CatalogMgr*. The standard exception identifier *BadQueryValue* is returned if the client has supplied one or more values for query attributes which are inappropriate or exceed the allowed or expected values of that attribute. The standard exception identifier *UnknownProperty* will be returned if the client has supplied one or more properties unknown or unsupported by this *CatalogMgr*. The standard exception identifier *BadPropertyValue* is returned if the client has supplied one or more values for properties which are inappropriate or exceed the allowed or expected values of that property.

**2.3.10.3. hit\_count**

```

HitCountRequest hit_count (in Query aQuery, in
PropertyList properties)

raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

```

This operation allows a client to determine the number of results (“hits”) that would be returned from a particular query. The operation parameters, properties and exceptions for this operation are identical in form and meaning to those of the *submit\_query* operation defined above. A successful invocation of this operation returns a reference to a *HitCountRequest* object.

**2.3.11. ProductMgr**

```

interface ProductMgr:LibraryManager,AccessManager
{

  GetParametersRequest get_parameters (in UID::Product
product,
                                     in UCO::NameList
desired_parameters,
                                     in PropertyList properties)

                                     raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

  RelatedFileTypeList get_related_file_types( in
UID::Product prod)

                                     raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

  GetRelatedFilesRequest get_related_files(
                                     in UID::ProductList products,
                                     in UCO::FileLocation location,

```

UNCLASSIFIED

```

        in RelatedFileType type,
        in PropertyList properties)

    raises ( UCO::InvalidInputParameter,
            UCO::ProcessingFault, UCO::SystemFault);

};

```

The *ProductMgr* interface provides operations that allow a client to determine characteristics about a specific data set or product. The operations defined in the *ProductMgr* interface are described in the following subsections.

### 2.3.11.1. get\_parameters

```

GetParametersRequest get_parameters

        (in UID::Product product,
         in UCO::NameList
         desired_parameters,
         in PropertyList properties)

    raises ( UCO::InvalidInputParameter,
            UCO::ProcessingFault, UCO::SystemFault);

```

This operation allows a client to submit a Request to determine the characteristics of a specific data set or product. The client supplies a reference to the data set of interest in Product *product*. The client also indicates which parameters are of interest in the NameList *desired\_parameters*. The client also describes any properties that further refine, effect or amplifies this Request by supplying their names and values in the PropertyList *properties*. (The properties that are available or applicable to this operation are defined in the appropriate GIAS profile.) A successful invocation of this operation will return a reference to a *GetParametersRequest* object.

The standard exception identifier *UnknownProduct* will be returned if the client supplied a product reference unknown to this *ProductMgr*. The standard exception identifier *UnknownProperty* will be returned if the client has supplied one or more properties unknown or unsupported by this *ProductMgr*. The standard exception identifier *BadPropertyValue* is returned if the client has supplied one or more values for properties, which are inappropriate or exceed the allowed or expected values of that property. The standard

exception identifier `LockUnavailable` is returned if the product is already locked or the client is not allowed to lock this Product.

### 2.3.11.2. `get_related_file_types`

```
RelatedFileTypeList get_related_file_types( in
  UID::Product prod)

      raises ( UCO::InvalidInputParameter,
              UCO::ProcessingFault, UCO::SystemFault);
```

This operation is used to obtain a list of acceptable `RelatedFileType` values for each Product. These values are used with the `get_related_files` operation (see section 2.3.11.3 below). The standard exception identifier `UnknownProduct` is returned if the client supplied a Product which is unknown or unusable by this `ProductMgr`.

### 2.3.11.3. `get_related_files`

```
GetRelatedFilesRequest get_related_files(

      in UID::ProductList products,

      in UCO::FileLocation location,

      in RelatedFileType type,

      in PropertyList properties)

  raises ( UCO::InvalidInputParameter,
          UCO::ProcessingFault, UCO::SystemFault);
```

This selector operation allows a client to submit a Request for a specified type of related file/dataset for a set of products. The client supplies a reference to the set of data sets of interest in `ProductList products`. The location in which the related files are to be placed is described in the parameter `location`. This `FileLocation` data type is populated to describe the path to the directory level. The field `file_name` of this parameter is left blank (empty string). The client also indicates the type of related file desired in the parameter `type`. The acceptable values for this parameter can be determined via the `get_related_file_types` operation (see section 2.3.11.2. above). The client also describes any properties that further refine, effect or amplifies this Request by supplying their names and values in the `PropertyList properties`. (The properties that are available or applicable to this operation are defined in the appropriate GIAS profile.) A successful invocation of this operation will return a reference to a `GetRelatedFilesRequest` object.

The standard exception identifier `UnknownProduct` will be returned if the client supplied a product reference unknown to this `ProductMgr`. The standard exception identifier `BadLocation` will be returned if

the client supplied a location that was incomplete or inaccessible. The standard exception identifier *BadFileType* will be returned if the client supplied a *RelatedFileType* that is not valid for the Product requested. The standard exception identifier *UnknownProperty* will be returned if the client has supplied one or more properties unknown or unsupported by this *ProductMgr*. The standard exception identifier *BadPropertyValue* is returned if the client has supplied one or more values for properties, which are inappropriate or exceed the allowed or expected values of that property.

### 2.3.12. IngestMgr

```
interface IngestMgr:LibraryManager,RequestManager
{
// FileLocation contains a directory

IngestRequest bulk_pull(in UCO::FileLocation
location, in PropertyList property_list)

raises( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

// FileLocation contains a directory

IngestRequest bulk_push(in Query aQuery, in
UCO::FileLocation location, in PropertyList
property_list )

raises( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

};
```

The *IngestMgr* provides operations that allow a Library to exchange large amounts of metadata with another library. The exchange takes place by exchanging (pushing or pulling) a set of files containing the metadata between the Libraries. The format of the files exchanged and the mapping of those file formats into and out of the Library's implementation are outside the scope of the GIAS. The details of this file format and its mappings will be detailed in the appropriate GIAS profile. The operations defined in the Request interface are described in the following subsections.

#### 2.3.12.1. bulk\_push

```
// FileLocation contains a directory

IngestRequest bulk_push(in Query aQuery, in
UCO::FileLocation location, in PropertyList
property_list )

raises( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

This operation places a Request to push all metadata concerning a specified data view. The initiating Library also supplies a query to further refine the desired metadata. Both of these elements are supplied in the in parameter *aQuery*. The format of the query is defined by the BQS (see chapter 4). The initiating library describes any properties that further refine, effect or amplify this Request by supplying their names and values in the PropertyList *property\_list*. (The properties that are available or applicable to this operation are defined in the appropriate GIAS profile.).

The standard exception identifier *UnknownViewName* will be returned if the initiating Library has supplied a data view unknown or unsupported by this *IngestMgr*. The standard exception identifier *BadLocation* will be returned if the client supplies a location description which is syntactically invalid, incomplete or specifies a location unknown or inaccessible by the *IngestMgr*. This does **not** require the *IngestMgr* to determine the validity of the user\_name - password combination specified in *location* or the availability of space at *location* to return successfully. The standard exception identifier *UnknownProperty* will be returned if the client has supplied one or more properties unknown or unsupported by this *IngestMgr*. The standard exception identifier *BadPropertyValue* is returned if the client has supplied one or more values for properties, which are inappropriate or exceed the allowed or expected values of that property.

### 2.3.12.2. bulk\_pull

```
// FileLocation contains a directory

IngestRequest bulk_pull( in UCO::FileLocation
location, in PropertyList property_list)

raises( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

This operation allows a Library (the initiating Library) to notify another Library (the receiving library) that a block of metadata is available to be ingested. The initiating Library also describes any properties that further refine, effect or amplify this Request by supplying their names and values in the PropertyList *property\_list*. (The properties that are available or applicable to this operation are defined in the appropriate GIAS profile.) A successful invocation of this operation will return a reference to an *IngestRequest* object.

The standard exception identifier *BadLocation* will be returned if the client supplies a location description which is syntactically invalid, incomplete or specifies a location unknown or inaccessible by the *IngestMgr*. This does **not** require the *IngestMgr* to determine the validity of the user\_name - password combination specified in *location* or the availability of space at *location* to return successfully. The standard exception identifier *UnknownProperty* will be returned if the client has supplied one or more properties unknown or unsupported by this *IngestMgr*. The standard exception identifier *BadPropertyValue* is returned if the client has supplied one or more values for properties which are inappropriate or exceed the allowed or expected values of that property

### 2.3.13. QueryOrderMgr

```
interface QueryOrderMgr:LibraryManager,
RequestManager

{

EventList get_event_descriptions()

        raises (UCO::ProcessingFault,
UCO::SystemFault);

SubmitQueryOrderRequest submit_query_order (

        in Query aQuery,

        in QueryLifeSpan lifespan,

        in OrderType o_type,

        in QueryOrderContents order,

        in PropertyList properties)

        raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

};
```

The *QueryOrderMgr* allows a client to place a query with a Library that will monitor the Library for existing (immediate orders) or new products arriving in the Library (standing orders) and then automatically deliver these products to the requestor. The details of the method to submit a standing order (using the operation

*submit\_query\_order*) are identical to submitting a *CatalogMgr* query. (See either of this query operation of the *CatalogMgr* Section 2.3.10). In addition, a query order is defined by two additional parameters: a flag which defines whether this is an immediate or standing order and a lifespan which is described by including a completed *QueryLifeSpan* structure (note that for immediate orders, the *QueryLifeSpan* is ignored).

In addition to submitting a query, a client must submit a *QueryOrderContents* structure that describes the details of how the products are to be delivered. The information contained in the *QueryOrderContents* will be applied to all products generated by the order. The client invokes the operation *submit\_query\_order*, passing in the query and order and receives a *SubmitQueryOrderRequest* object to track the order. The client establishes a *QueryOrder* in exactly the same way as establishing a *StandingQuery*, with the addition of a *QueryOrderContents*, which defines the delivery details. As hits are generated against the query, the associated products are delivered as defined in the order. Calling *complete* on the *SubmitQueryOrderRequest* can be used to determine if the order has been completed or the client can ignore the status of the order.

### 2.3.13.1 *get\_event\_descriptions*

```
EventList get_event_descriptions()

raises (UCO::ProcessingFault, UCO::SystemFault);
```

*get\_event\_descriptions* returns a list of events that can be used by the client in the lifespan parameter of *submit\_query\_order* to set the details of the lifetime of a query such as start, duration, and end.

### 2.3.13.2 *submit\_query\_order*

```
SubmitQueryOrderRequest submit_query_order (

    in Query aQuery,

    in QueryLifeSpan lifespan,

    in OrderType o_type,

    in QueryOrderContents order,

    in PropertyList properties)

raises ( UCO::InvalidInputParameter,
        UCO::ProcessingFault, UCO::SystemFault );
```

This operation allows a client to establish a simple query with a Library that will monitor the Library for existing (“immediate” orders) or new products arriving in the Library (“standing” orders) and then automatically deliver these products to the requestor. The basic parameters, semantics, and standard exception identifiers for this operation are identical to those for submitting a CatalogMgr query. (See the *submit\_query* operation of the CatalogMgr in Section 2.3.10.) Additional parameters provide specific details for establishing the lifespan, type, and delivery details of this query order. The parameter OrderType *o\_type* provides a flag that defines whether this is an “immediate” or “standing” order. For “immediate” orders, the query is performed just once on the Library. For “standing” orders, the parameter QueryLifeSpan *lifespan* determines the time period that the query is to be run. (Note: For “immediate” orders, the *lifespan* parameter is ignored.) The parameter QueryOrderContents *order* describes the details of how the requested products are to be delivered.

The standard exception identifier *InvalidEvent* is returned if an event contained in the *lifespan* parameter is inappropriate or unknown. The standard exception identifier *ImplementationLimit* is returned if a submitted value in the *order* parameter exceeds this Manager’s capabilities. These limits are implementation specific.

### 2.3.14. VideoMgr

```
//interface VideoMgr : LibraryManager, AccessManager
{
    //};
```

The *VideoMgr* is intended to provide operations that allow a client to access a video data set as a temporal stream as well as a geospatial data set. The requirements and design of this interface and operations are TBR.

### 2.3.15. Request (j/NPS)

```
interface Request
{
    UCO::RequestDescription get_request_description ()
    raises ( UCO::ProcessingFault, UCO::SystemFault);

    void set_user_info (in string message)
```

```
    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

UCO::Status get_status ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);

DelayEstimate get_remaining_delay()

    raises (UCO::ProcessingFault,
UCO::SystemFault);

void cancel ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);

CallbackID register_callback (in CB::Callback
acallback)

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

void free_callback (in CallbackID id)

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

RequestManager get_request_manager ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);

};
```

The *Request* interface is an abstract interface that defines those operations that are common to Request objects. Most operations of a *RequestManager* return a reference to a specialized Request object. All specialized Request objects are derived (inherited) from *Request*. This interface defines the operations in the following subsections.

#### **2.3.15.1. get\_request\_description (j/NPS)**

```
UCO::RequestDescription get_request_description()

    raises (UCO::ProcessingFault,
UCO::SystemFault);
```

This selector operation returns a populated *RequestDescription* structure that describes the *Request*.

#### 2.3.15.2. set\_user\_info

```
void set_user_info (in string message)

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

This modifier operation allows a client to provide information that describes the *Request*. The client supplies this information, in the form of a string in *message*. A successful invocation of this operation associates the client's message with the *Request*. This client-supplied information can be accessed in the *user\_info* element of the *RequestDescription* structure returned by the *get\_request\_description* operation (see above).

The *ImplementationLimit* standard exception identifier will be returned if the client supplies a message that exceeds the maximum length allowed by the implementation. This maximum length is implementation dependent.

#### 2.3.15.3. get\_status (j/NPS)

```
UCO::Status get_status ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);
```

This selector operation returns the current status of the *Request*. A successful invocation returns a *Status* structure (see the UCO document for details).

#### 2.3.15.4. cancel

```
void cancel ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);
```

This modifier operation is used to terminate further processing of a *Request*. After successful invocation of this operation, all current and future processing associated with this *Request* is terminated.

Before	After
COMPLETED	COMPLETED
IN_PROGRESS	CANCELED
ABORTED	ABORTED
CANCELED	CANCELED
PENDING	CANCELED
SUSPENDED	CANCELED
RESULTS_AVAILABLE	CANCELED

### 2.3.15.5. register\_callback (j/NPS)

```
CallbackID register_callback (in CB::Callback
acallback)
```

```
raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault );
```

This operation allows a client to register a Callback object with a *Request*. The purpose of a Callback object is to provide a mechanism to allow the Request to notify the client that processing of a *Request* has transitioned into a state which is specified to trigger a Callback. The states which trigger a Callback are specific to each concrete Request and are defined in Appendix G. A client can register zero or more *Callback* objects with a *Request*. The client indicates the Callback object to be registered by supplying a reference to a Callback object in a *callback*. A successful invocation of this operation returns a CallbackID that uniquely identifies that instance of Callback. The details of this CallbackID are defined in the appropriate GIAS profile. Note that registering the SAME Callback object twice results in two callbacks being registered with different CallbackIDs. Following successful invocation of this operation the Callback specified will be associated with this Request (registered). When this *Request* reaches a state which triggers a Callback, the appropriate operation(s) on the specified Callback object will be invoked. (See section 3 for details of the operations invoked on the Callback object). Note that if a Callback is registered with a Request which is already in a state that triggers a Callback that Callback will be triggered immediately.

The standard exception identifier *UnknownCallback* will be returned if the client supplies a reference to a *Callback* object that is unknown or unreachable by the *Request*.

#### 2.3.15.6. free\_callback (j/NPS)

```
void free_callback (in CallbackID id)

raises ( UCO::InvalidInputParameter,
        UCO::ProcessingFault, UCO::SystemFault );
```

This operation allows a client to remove a *Callback* previously registered with a *Request*. The client supplies a reference to the *Callback* that is to be de-registered. Following successful invocation of this operation, the *Callback* specified will no longer be registered with this *Request*.

The standard exception identifier *UnknownCallback* will be returned if the client supplies a reference to a *Callback* object that is unknown or unreachable by the *Request*. The standard exception identifier *UnregisteredCallback* will be returned if the client attempts to free a *Callback*, which has not previously been registered with this *Request*.

#### 2.3.15.7. get\_request\_manager

```
RequestManager get_request_manager ()

raises (UCO::ProcessingFault,
        UCO::SystemFault);
```

This operation allows a client to discover which *RequestManager* is managing the *Request*. A successful invocation of this operation returns a reference to the *RequestManager* that is managing this *Request*. This reference can be narrowed (cast) into a more concrete type.

#### 2.3.15.8. get\_remaining\_delay

```
DelayEstimate get_remaining_delay ()

raises (UCO::ProcessingFault,
        UCO::SystemFault);
```

This operation returns an estimate in seconds (*time\_delay* field of the returned *DelayEstimate* structure ) until the *Request* reaches the COMPLETE (or the RESULTS\_AVAILABLE state if applicable to the Request) state. The delay is valid only if the *valid\_time\_delay* component of the *DelayEstimate* is true. If the *valid\_time\_delay* is false, the *time\_delay* should be ignored by the client.

### 2.3.16. CreateMetaDataRequest

```

interface CreateMetaDataRequest:Request
{
    UCO::State complete (out UID::Product new_product)
        raises (UCO::ProcessingFault,
UCO::SystemFault);
};

```

This *Request* is returned by the operation `create_metadata` of the *CreationMgr*. This *Request* defines the following operation:

### 2.3.16.1. complete

```

UCO::State complete (out UID::Product new_product)
    raises (UCO::ProcessingFault,
UCO::SystemFault);

```

This operation allows a client to complete processing of the *CreateMetaDataRequest*. It returns an identifier in the form of a *Product* for the newly created *Product*. It also returns a *State* indicating details of the completed operation. See Appendix G for a description of the state transitions defined for the *Request*.

### 2.3.17. SetAvailabilityRequest

```

interface SetAvailabilityRequest:Request
{
    UCO::State complete ()
        raises (UCO::ProcessingFault,
UCO::SystemFault);
};

```

This *Request* is returned by the operation `set_availability` of the *AccessMgr*. This *Request* defines the following operation:

### 2.3.17.1. complete

```

UCO::State complete ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);

```

This operation allows a client to complete processing of the *SetAvailabilityRequest*. This operation blocks until the requested products are placed in the requested UseMode. It also returns a *State* indicating details of the completed operation. See Appendix G for a description of the state transitions defined for the Request.

### 2.3.18. GetRelatedFilesRequest

```

interface GetRelatedFilesRequest:Request
{
    UCO::State complete (out UCO::NameList locations)

        raises (UCO::ProcessingFault,
UCO::SystemFault);
};

```

This *Request* is returned by the operation *get\_related\_files* of the *ProductMgr*. This Request defines the following operation:

#### 2.3.18.1. complete

```

UCO::State complete (out UCO::NameList locations)

    raises (UCO::ProcessingFault,
UCO::SystemFault);

```

This operation allows a client to complete processing of the *GetRelatedFilesRequest*. This operation blocks until the requested related files have been made available. It returns a sequence of *names* in the parameter *locations*, which holds the file names of the related files. The names in this sequence are in the same order as specified in the ProductList submitted in the *get\_related\_files* operation. It also returns a *State* indicating details of the completed operation. See Appendix G for a description of the state transitions defined for the Request.

### 2.3.19. CreateRequest

```
interface CreateRequest:Request
{
    UCO::State complete (out UID::ProductList
new_products)
    raises (UCO::ProcessingFault, UCO::SystemFault);
};
```

The *CreateRequest* is returned by invocations of the create operation of the *CreationMgr*. This operation defined in the *CreateRequest* interface, is described in the following subsection.

#### **2.3.19.1. complete**

```
UCO::State complete (out UID::ProductList
new_products)
    raises (UCO::ProcessingFault, UCO::SystemFault);
```

This operation allows a client to complete processing of the *CreateRequest*. This operation blocks until the requested operation reaches a complete state. A successful invocation of this operation returns a *ProductList* containing the references to the newly created product or the composite product for multi-part products. It also returns a *State* indicating details of the completed operation. See Appendix G for a description of the state transitions defined for the Request.

#### **2.3.20. UpdateRequest**

```
interface UpdateRequest:Request
{
```

```

    UCO::State complete ()

    raises (UCO::ProcessingFault, UCO::SystemFault);

};

```

The UpdateRequest is returned by invocations of the UpdateMgr::update operation. It is used to complete the processing of an update of a catalog entry.

### 2.3.20.1. complete

```

    UCO::State complete ()

    raises (UCO::ProcessingFault, UCO::SystemFault);

```

This operation completes the processing of a catalog update operation. It returns the status of the update operation. See Appendix G for a description of the state transitions defined for the Request.

### 2.3.21. SubmitQueryRequest

```

interface SubmitQueryRequest:Request

{
    void set_number_of_hits (in unsigned long hits)

    raises ( UCO::InvalidInputParameter,
    UCO::ProcessingFault, UCO::SystemFault);

    UCO::State complete_DAG_results (out QueryResults
    results);

        raises (UCO::ProcessingFault,
    UCO::SystemFault);

    UCO::State complete_stringDAG_results (out
    UCO::StringDAGList results);

        raises (UCO::ProcessingFault,
    UCO::SystemFault);

    UCO::State complete_XML_results (out UCO::XMLDocument
    results)

```

```
        raises (UCO::ProcessingFault, UCO::SystemFault)
    };
```

The *SubmitQueryRequest* is returned by invocations of the *submit\_query* operation of the *CatalogMgr*. It provides operations to retrieve the results of the submitted query in three forms: as a DAG, as a StringDAG or as a XMLDocument. This interface defines the following operations:

#### 2.3.21.1. set\_number\_of\_hits

```
void set_number_of_hits (in unsigned long hits)

raises ( UCO::InvalidInputParameter,
        UCO::ProcessingFault, UCO::SystemFault);
```

This operation allows a client to set the number of results (“hits”) that are returned by invocations of the operation complete (see below). This operation also sets the number of hits accumulated by this *SubmitQueryRequest* before a Callback is triggered.

#### 2.3.21.2. complete\_DAG\_results

```
UCO::State complete_DAG_results (out QueryResults
results)

raises (UCO::ProcessingFault, UCO::SystemFault);
```

This operation allows a client to complete processing of the *SubmitQueryRequest*. The operation blocks until the number of results set by *set\_number\_of\_hits* has been accumulated or all results have been processed. A successful invocation of this operation returns a *QueryResults* structure containing results from the query. Subsequent invocations of this operation can be used to retrieve any remaining results. Once a set of results have been returned from this operation they are no longer accessible, that is, there is no mechanism to retrieve the same set of results a second time. It is the client’s responsibility to hold any retrieved results. The number of results returned in this structure per invocation is determined by the value set in an invocation of *set\_number\_of\_hits* (see above). A retrieval that returns a number of results less than the value previously set by *set\_number\_of\_hits* indicates that all results have been retrieved. If *set\_number\_of\_hits* has not been called prior to the invocation of complete, the number of results returned in the *QueryResults* structure is determined by a default value, which is implementation dependent. See Appendix G for a description of the state transitions defined for the Request.

#### 2.3.21.3. complete\_stringDAG\_results

```
UCO::State complete_stringDAG_results (out
UCO::StringDAGList results)

raises (UCO::ProcessingFault, UCO::SystemFault);
```

This operation allows a client to complete processing of the *SubmitQueryRequest*. The operation blocks until the number of results set by *set\_number\_of\_hits* has been accumulated or all results have been processed. A successful invocation of this operation returns a *UCO::StringDAGList* structure containing results from the query. Subsequent invocations of this operation can be used to retrieve any remaining results. Once a set of results have been returned from this operation they are no longer accessible, that is, there is no mechanism to retrieve the same set of results a second time. It is the client's responsibility to hold any retrieved results. The number of results returned in this structure per invocation is determined by the value set in an invocation of *set\_number\_of\_hits* (see above). A retrieval that returns a number of results less than the value previously set by *set\_number\_of\_hits* indicates that all results have been retrieved. If *set\_number\_of\_hits* has not been called prior to the invocation of *complete*, the number of results returned in the *StringDAGList* structure is determined by a default value, which is implementation dependent. See Appendix G for a description of the state transitions defined for the Request

#### 2.3.21.4. complete\_XML\_results

```
UCO::State complete_XML_results (out UCO::XMLDocument
results)

raises (UCO::ProcessingFault, UCO::SystemFault);
```

This operation allows a client to complete processing of the *SubmitQueryRequest*. The operation blocks until the number of results set by *set\_number\_of\_hits* has been accumulated or all results have been processed. A successful invocation of this operation returns a *UCO::XMLDocument* structure containing results from the query. Subsequent invocations of this operation can be used to retrieve any remaining results. Once a set of results have been returned from this operation they are no longer accessible, that is, there is no mechanism to retrieve the same set of results a second time. It is the client's responsibility to hold any retrieved results. The number of results returned in this structure per invocation is determined by the value set in an invocation of *set\_number\_of\_hits* (see above). A retrieval that returns a number of results less than the value previously set by *set\_number\_of\_hits* indicates that all results have been retrieved. If *set\_number\_of\_hits* has not been called prior to the invocation of *complete*, the number of results returned in the *XMLDocument* structure is determined by a default value, which is implementation dependent. See Appendix G for a description of the state transitions defined for the Request.

#### 2.3.22. SubmitStandingQueryRequest

```
interface SubmitStandingQueryRequest:Request
{
    void set_number_of_hits (in unsigned long hits)
```

```
        raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

    unsigned long  get_number_of_hits()

        raises (UCO::ProcessingFault,
UCO::SystemFault);

    unsigned long  get_number_of_hits_in_interval(in
unsigned long  interval)

        raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

    unsigned long  get_number_of_intervals()

        raises (UCO::ProcessingFault,
UCO::SystemFault);

    void  clear_all()

        raises (UCO::ProcessingFault,
UCO::SystemFault);

    void  clear_intervals(in unsigned long
num_intervals)

        raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

    void  clear_before(in UCO::Time relative_time)

        raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

    void  pause()

        raises (UCO::ProcessingFault,
UCO::SystemFault);

    void  resume()
```

```
        raises (UCO::ProcessingFault,
UCO::SystemFault);

        UCO::AbsTime get_time_last_executed()

        raises ( UCO::ProcessingFault,
UCO::SystemFault);

        UCO::AbsTime get_time_next_execution()

        raises ( UCO::ProcessingFault,
UCO::SystemFault);

        UCO::State complete_DAG_results (out QueryResults
results)

        raises (UCO::ProcessingFault,
UCO::SystemFault);

        UCO::State complete_stringDAG_results (out
UCO::StringDAGList results)

        raises (UCO::ProcessingFault,
UCO::SystemFault);

        UCO::State complete_XML_results (out
UCO::XMLDocument results)

        raises (UCO::ProcessingFault,
UCO::SystemFault);

};
```

The SubmitStandingQueryRequest is returned by invocations of the submit\_standing\_query operation of the StandingQueryMgr.

### **2.3.22.1. set\_number\_of\_hits**

```
void set_number_of_hits (in unsigned long hits)

        raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

UNCLASSIFIED

This operation allows a client to set the number of results (“hits”) that are returned by invocations of the operation complete (see below). This operation also sets the number of hits accumulated by this *Request* before a Callback is triggered.

#### 2.3.22.2. **get\_number\_of\_hits**

```
unsigned long  get_number_of_hits()  
  
    raises (UCO::ProcessingFault,  
           UCO::SystemFault);
```

This operation returns the current total number of hits collected by this Request.

#### 2.3.22.3. **get\_number\_of\_hits\_in\_interval**

```
unsigned long  get_number_of_hits_in_interval(in  
unsigned long  interval)  
  
    raises ( UCO::InvalidInputParameter,  
           UCO::ProcessingFault, UCO::SystemFault);
```

This operation returns the number of hits in the specified interval.

#### 2.3.22.4. **get\_number\_of\_intervals**

```
unsigned long  get_number_of_intervals()  
  
    raises (UCO::ProcessingFault,  
           UCO::SystemFault);
```

This operation returns the number of intervals for which this Request has collected hits.

#### 2.3.22.5. **clear\_all**

```
void  clear_all()  
  
    raises (UCO::ProcessingFault,  
           UCO::SystemFault);
```

This operation clears all hits currently held by this Request.

#### **2.3.22.6. clear\_intervals**

```
void clear_intervals(in unsigned long num_intervals)

raises ( UCO::InvalidInputParameter,
        UCO::ProcessingFault, UCO::SystemFault);
```

This operation clears all hits in the specified interval.

#### **2.3.22.7. clear\_before**

```
void clear_before(in UCO::Time relative_time)

raises ( UCO::InvalidInputParameter,
        UCO::ProcessingFault, UCO::SystemFault);
```

This operation clears all hits collected before the specified time.

#### **2.3.22.8. pause**

```
void pause()

raises (UCO::ProcessingFault, UCO::SystemFault);
```

This operation suspends processing of the Request.

#### **2.3.22.9. resume**

```
void resume()

raises (UCO::ProcessingFault, UCO::SystemFault);
```

This operation resumes processing of a suspended Request.

#### **2.3.22.10 get\_time\_last\_executed and get\_time\_next\_execution**

```
UCO::AbsTime get_time_last_executed()

raises (UCO::ProcessingFault, UCO::SystemFault);

UCO::AbsTime get_time_next_execution()
```

```
raises (UCO::ProcessingFault, UCO::SystemFault);
```

These operations allow access to the time of the last performed execution of this standing query and the time of the next scheduled execution of this standing query.

The standard exception identifier `InvalidEvent` will be raised if the standing query has not yet been run (`get_time_last_executed`) or will not run again (`get_time_next_execution`).

### 2.3.22.11 complete\_DAG\_results

```
UCO::State complete_DAG_results (out QueryResults  
results)
```

```
raises (UCO::ProcessingFault,  
UCO::SystemFault);
```

This operation allows a client to complete processing of the `SubmitStandingQueryRequest`. This operation blocks until the requested operation reaches a `COMPETE` or `RESULTS_AVAILABLE` state. A successful invocation of this operation returns a “results” value which represents a *DAGList*. It also returns a `State` indicating details of the completed operation. See Appendix G for a description of the state transitions defined for the Request.

### 2.3.22.12 complete\_stringDAG\_results

```
UCO::State complete_stringDAG_results (out  
UCO::StringDAGList results)
```

```
raises (UCO::ProcessingFault,  
UCO::SystemFault);
```

This operation allows a client to complete processing of the `SubmitStandingQueryRequest`. This operation blocks until the requested operation reaches a `COMPETE` or `RESULTS_AVAILABLE` state. A successful invocation of this operation returns a “results” value which represents a *StringDAGList*. It also returns a `State` indicating details of the completed operation. See Appendix G for a description of the state transitions defined for the Request.

### 2.3.22.13 complete\_XML\_results

```
UCO::State complete_XML_results (out UCO::XMLDocument
results)

    raises (UCO::ProcessingFault,
UCO::SystemFault);
```

This operation allows a client to complete processing of the SubmitStandingQueryRequest. This operation blocks until the requested operation reaches a COMPLETE or RESULTS\_AVAILABLE state. A successful invocation of this operation returns a “results” value in the form of an *XMLDocument*. It also returns a State indicating details of the completed operation. See Appendix G for a description of the state transitions defined for the Request.

### 2.3.23. HitCountRequest

```
interface HitCountRequest:Request
{
    UCO::State complete (out unsigned long
number_of_hits)

    raises (UCO::ProcessingFault,
UCO::SystemFault);
};
```

The HitCountRequest is returned by invocations of the hit\_count operation of the CatalogMgr. This operation defined in the HitCountRequest interface is described in the following subsection.

#### 2.3.23.1. complete

```
UCO::State complete (out unsigned long
number_of_hits)
```

```

        raises (UCO::ProcessingFault,
UCO::SystemFault);

```

This operation allows a client to complete processing of the HitCountRequest. This operation blocks until the requested operation reaches a COMPETE state. A successful invocation of this operation returns a value that indicates the total number of results (“hits”) that would be returned if the query was executed. It also returns a State indicating details of the completed operation. See Appendix G for a description of the state transitions defined for the Request.

### 2.3.24. GetParametersRequest

```

interface GetParametersRequest:Request
{
    UCO::State complete (out UCO::DAG parameters)
        raises (UCO::ProcessingFault,
UCO::SystemFault);
    UCO::State complete_StringDAG (out UCO::StringDAG
parameters)
        raises (UCO::ProcessingFault,
UCO::SystemFault);
};

```

The ParametersRequest is returned by invocations of the get\_parameters operation of the ProductMgr. This operation defined in the GetParametersRequest interface is described in the following subsection.

#### 2.3.24.1. complete

```

UCO::State complete (out UCO::DAG parameters)
    raises (UCO::ProcessingFault,
UCO::SystemFault);

```

This operation allows a client to complete processing of the ParametersRequest. This operation blocks until the requested operation reaches a COMPLETE state. A successful invocation of this operation returns a

UCO::DAG structure that contains the properties and current values of those properties of the product or data set requested. It also returns a State indicating details of the completed operation. See Appendix G for a description of the state transitions defined for the Request.

### 2.3.24.2. complete

```
UCO::State complete_StringDAG (out UCO::StringDAG
parameters)

    raises (UCO::ProcessingFault,
UCO::SystemFault);
```

The details of this operation are identical to those of the *complete* operation described above except that this operation returns the results as a String DAG.

## 2.3.25. IngestRequest

```
interface IngestRequest:Request

{

UCO::State complete ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);

};
```

The IngestRequest is returned by invocations of the *bulk\_pull* and *bulk\_push* operations of the IngestMgr. This operation defined in the IngestRequest interface is described in the following subsection.

### 2.3.25.1. complete

```
UCO::State complete ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);
```

This operation allows a client to complete processing of the *IngestRequest*. This operation blocks until the requested operation reaches a complete state. A successful invocation of this operation indicates that the files containing the metadata to be exchanged are available. For the *bulk\_pull* operation this indicates that the metadata file has been delivered to the location specified and is ready to be ingested by the pulling Library. For the *bulk\_push* operation this indicates that the metadata file has been found by the receiving Library at the location specified. This operation does NOT indicate that the receiving Library has successfully ingested the metadata file. It merely indicates successful transfer of and access to the metadata file. See Appendix G for a description of the state transitions defined for the Request.

### 2.3.26. OrderRequest

```
interface OrderRequest:Request
{
    UCO::State complete (out DeliveryManifest prods)
        raises (UCO::ProcessingFault,
UCO::SystemFault);
};
```

The *OrderRequest* is returned by an invocation of the *order* operation of the *OrderMgr*. The operation defined in the *OrderRequest* interface is described in the following subsection.

#### 2.3.26.1. complete

```
UCO::State complete (out DeliveryManifest prods)
    raises (UCO::ProcessingFault,
UCO::SystemFault);
```

This operation allows a client to complete processing of the *OrderRequest*. This operation blocks until the requested operation reaches a complete state. A successful invocation of this operation indicates that the client's *order* is available. The parameter *prods* contains the package and its contents as delivered. This operation also returns a State indicating details of the completed operation. See Appendix G for a description of the state transitions defined for the Request.

### 2.3.27 SubmitQueryOrderRequest

```
interface SubmitQueryOrderRequest:Request
{
    void pause()
        raises (UCO::ProcessingFault,
UCO::SystemFault);

    void resume()
        raises (UCO::ProcessingFault,
UCO::SystemFault);

    UCO::State complete_list (out DeliveryManifestList
prods)
        raises (UCO::ProcessingFault,
UCO::SystemFault);

    UCO::State complete (out DeliveryManifest prods)
        raises (UCO::ProcessingFault,
UCO::SystemFault);
};
```

This *SubmitQueryOrderRequest* is returned by the operation *submit\_query\_order* of the *QueryOrderMgr*. This *Request* defines the following operations:

### **2.3.27.1 Pause**

```
void pause()
    raises (UCO::ProcessingFault,
UCO::SystemFault);
```

This operation temporarily suspends the Request.

### **2.3.27.2 Resume**

This operation causes a suspended Request to continue.

```

void resume( )

    raises (UCO::ProcessingFault,
UCO::SystemFault);

```

### 2.3.27.3 Complete\_list

```

UCO::State complete_list (out DeliveryManifestList
prods)

    raises (UCO::ProcessingFault,
UCO::SystemFault);

```

This operation blocks until the *submit\_query\_order* reaches a COMPLETE state, indicating that the order is ready. The parameter *prods* contains the package(s) and their contents as delivered. This DeliveryManifestList contains the description of all packages which have been delivered from this QueryOrder Request since the last time the *complete* operation was called, which may include deliveries from one or more intervals. That is, the contents of the DeliveryManifestList accumulates until the *complete* operation is invoked, at which point the DeliveryManifestList is cleared and begins to accumulate again. There is one DeliveryManifest in the DeliveryManifestList for each separate package delivered. See Appendix G for a description of the state transitions defined for the Request.

### 2.3.27.4 Complete

```

UCO::State complete (out DeliveryManifest prods)

    raises (UCO::ProcessingFault,
UCO::SystemFault);

```

This operation blocks until the *submit\_query\_order* reaches a COMPLETE state, indicating that the order is ready. The parameter *prods* contains the package(s) and their contents as delivered. This DeliveryManifest contains a concatenated description of all packages which have been delivered from this QueryOrder Request since the last time the *complete* operation was called, which may include deliveries from one or more intervals. That is, the contents of the DeliveryManifest accumulate until the *complete* operation is invoked, at which point the DeliveryManifest contents are cleared and begins to accumulate again. See Appendix G for a description of the state transitions defined for the Request.

## 2.3.28. CreateAssociationRequest

```
interface CreateAssociationRequest:Request
{
    UCO::State complete ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);
};
```

The *CreateAssociationRequest* is returned by an invocation of the *create\_association* operation of the CreationMgr. The operation defined in this interface is described in the following subsection.

### 2.3.28.1. complete

```
UCO::State complete ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);
```

This operation allows a client to complete processing of the *CreateAssociationRequest*. This operation blocks until the requested operation reaches a COMPLETE state. A successful invocation of this operation indicates that the client's association has been successfully created. It also returns a State indicating details of the completed operation. See Appendix G for a description of the state transitions defined for the Request.

### 2.3.29. UpdateByQueryRequest

```
interface UpdateByQueryRequest:Request
{
    UCO::State complete ()

    raises (UCO::ProcessingFault,
UCO::SystemFault);
};
```

The *UpdateByQueryRequest* is returned by an invocation of the *update\_by\_query* operation of the UpdateMgr. The operation defined in this interface is described in the following subsection.

### 2.3.29.1. complete

```
UCO::State complete ()  
  
    raises (UCO::ProcessingFault,  
UCO::SystemFault);
```

This operation allows a client to complete processing of the *UpdateByQueryRequest*. This operation blocks until the requested processing has been performed or an error condition occurs. A successful invocation of this operation indicates that the requested updates have been successfully performed. It also returns a State indicating details of the completed operation. See Appendix G for a description of the state transitions defined for the Request.

## 2.4. Exceptions

### 2.4.1. Exception Model

The GIAS specification uses the exception model that is defined in Section 2.4 of the UCO Specification. This section defines a set of identifiers (string constants) that are used to identify the specific exception conditions of the GIAS interfaces. As such, they would be used in the *exception\_name* field of the *UCO::exception\_details* structure. When an exception that uses one of these standard identifiers is returned, the boolean *standard\_exception* field of the *UCO::exception\_details* structure should be set to TRUE. The UCO exception model defines three exceptions (*InvalidInputParameter*, *ProcessingFault* and *SystemFault*) which each represent a broad category of possible error conditions. The GIAS-specific error conditions defined below are grouped into one of these three categories.

### 2.4.2 InvalidInputParameter Exceptions (j/NPS)

#### 2.4.2.1 BadAccessCriteria (j/NPS)

```
const string BadAccessCriteriaConst =  
"BadAccessCriteria";
```

This exception indicates the client has supplied incomplete, invalid or otherwise unacceptable access criteria. The *exception\_details* structure will identify the unacceptable access criteria submitted.

#### 2.4.2.2 BadAccessValue (j/NPS)

```
const string BadAccessValueConst = "BadAccessValue";
```

This exception indicates that one or more values supplied for the access criteria was missing, incorrect or otherwise unacceptable. The *exception\_details* structure will identify which access criteria element(s) submitted were unacceptable.

#### **2.4.2.3 BadCreationAttributeValue (j/NPS)**

```
const string BadCreationAttributeValueConst =  
"BadCreationAttributeValue";
```

This exception indicates the client supplied a value for one or more creation attributes with an inappropriate type or invalid value (i.e., exceeded the allowed or expected range). The *exception\_details* structure will contain an explanation containing the names of all the unacceptable creation attributes.

#### **2.4.2.4 BadGeoRegion**

```
const string BadGeoRegionConst = "BadGeoRegion";
```

This exception indicates that a GeoRegion data structure supplied by the client is incomplete or describes a region that is inappropriate for the processing requested (i.e., region is not contained in the requested product).

#### **2.4.2.5 BadLocation (j/NPS)**

```
const string BadLocationConst = "BadLocation";
```

This exception indicates the client supplied a FileLocation structure that is syntactically invalid, incomplete or specifies a location unknown or inaccessible by the server.

#### **2.4.2.6 BadPropertyValue (j/NPS)**

```
const string BadPropertyValueConst =  
"BadPropertyValue";
```

This exception indicates the client supplied a value for one or more properties, which are inappropriate or exceed the allowed or expected values of that property. The *exception\_details* structure will contain an explanation containing the names of all the unacceptable properties.

### 2.4.2.7 BadQuery

```
const string BadQueryConst = "BadQuery";
```

This exception indicates that a query submitted by the client has improper syntax. This would include missing or mismatched delimiters, use of undefined operators or use of an operator inappropriate for an attribute. See Chapter 4 for a description of the BNF that describes the syntax for queries.

### 2.4.2.8 BadQueryAttribute

```
const string BadQueryAttributeConst =  
"BadQueryAttribute";
```

This exception indicates the client supplied one or more attributes unknown or unsupported by the server. The *exception\_details* structure will contain the unacceptable attributes.

### 2.4.2.9 BadQueryValue

```
const string BadQueryValueConst = "BadQueryValue";
```

This exception indicates the client supplied one or more values for query attributes, which are inappropriate or exceed the allowed or expected values for that attribute. The *exception\_details* structure will contain an explanation containing the names of all the unacceptable attributes.

### 2.4.2.10 BadTime

```
const string BadTimeConst = "BadTime";
```

This exception indicates the client supplied a time value that is incomplete or exceeds the allowed or expected range of times. The *exception\_details* structure will contain the unacceptable time value supplied.

### 2.4.2.11 BadUseMode

```
const string BadUseModeConst = "BadUseMode";
```

This exception indicates the client requested a UseMode that is inappropriate or unsupported for the product or conditions requested.

### 2.4.2.12 ImplementationLimit (j/NPS)

```
const string ImplementationLimitConst =  
"ImplementationLimit";
```

This exception indicates the client requested an operation with a parameter that exceeds an implementation specific limit for that parameter. The *exception\_details* structure will contain the name of the parameter exceeded.

#### **2.4.2.13 UnknownCallback (j/NPS)**

```
const string UnknownCallbackConst =  
"UnknownCallback";
```

This exception indicates the client supplied a reference to a Callback object that is unknown or unreachable by the Request.

#### **2.4.2.14 UnknownCreationAttribute (j/NPS)**

```
const string UnknownCreationAttributeConst =  
"UnknownCreationAttribute";
```

This exception indicates the client supplied a creation attribute that is unknown or unsupported by the server. The *exception\_details* structure will contain an explanation containing the names of the entire unknown or unsupported elements.

#### **2.4.2.15 UnknownManagerType (j/NPS)**

```
const string UnknownManagerTypeConst =  
"UnknownManagerType";
```

This exception indicates the client requested a Manager type, which was unknown or unsupported by this implementation. The *exception\_details* structure will contain an explanation containing the name of the unknown or unsupported Manager type.

#### **2.4.2.16 UnknownProduct**

```
const string UnknownProductConst = "UnknownProduct";
```

This exception indicates that the client requested a product reference unknown to the server.

#### **2.4.2.17 UnknownProperty (j/NPS)**

```
const string UnknownPropertyConst =  
"UnknownProperty";
```

This exception indicates the client supplied one or more properties unknown or unsupported by the server. The *exception\_details* structure will contain an explanation containing the names of all the unacceptable properties supplied.

#### **2.4.2.18 UnknownRequest (j/NPS)**

```
const string UnknownRequestConst = "UnknownRequest";
```

This exception indicates the client supplied a reference to a Request that is unknown to the server.

#### **2.4.2.19 UnknownUseMode**

```
const string UnknownUseModeConst = "UnknownUseMode";
```

This exception indicates the client supplied a UseMode unknown or unsupported by the server. The *exception\_details* structure will contain an explanation containing the name of the unacceptable UseMode supplied.

#### **2.4.2.20 UnregisteredCallback (j/NPS)**

```
const string UnregisteredCallbackConst =  
"UnregisteredCallback";
```

This exception indicates the client attempted an operation that requires a registered Callback with a reference to a Callback that has not been previously registered.

#### **2.4.2.21 BadOrder**

```
const string BadOrderConst = "BadOrder";
```

This exception indicates that an order placed by a client to a Library is not valid.

#### **2.4.2.22 UnknownViewName**

```
const string UnknownViewNameConst =  
"UnknownViewName";
```

This exception indicates that a specified view requested by a client is unknown by the Library.

#### **2.4.2.23 UnknownEntity**

```
const string UnknownEntityConst = "UnknownEntity";
```

This exception indicates that a client-requested entity is unknown.

#### **2.4.2.24 NoValuesRequested**

```
const string NoValuesRequestedConst =  
    "NoValuesRequested";
```

This exception indicates that a client did not request any values.

#### **2.4.2.25 UnsupportedConceptualAttribute**

```
const string UnsupportedConceptualAttributeConst =  
    "UnsupportedConceptualAttribute";
```

This exception indicates that a conceptual attribute specified in a Request is unsupported.

#### **2.4.2.26 BadResultAttribute**

```
const string BadResultAttributeConst =  
    "BadResultAttribute";
```

This exception indicates that a Request specified a results attribute that is unsupported.

#### **2.4.2.27 BadSortAttribute**

```
const string BadSortAttributeConst =  
    "BadSortAttribute";
```

This exception indicates that the sort attribute specified in the Request is unsupported.

#### **2.4.2.28 NonUpdateableAttribute**

```
const string NonUpdateableAttributeConst =  
    "NonUpdateableAttribute";
```

This exception indicates an attempt to update an attribute which the client is not allowed to modify.

#### **2.4.2.29 BadFileType**

```
const string BadFileTypeConst = "BadFileType";
```

This exception indicates the use of a RelatedFileType that is unknown or inappropriate for the context for which it was used.

#### **2.4.2.30 InvalidCardinality**

```
const string InvalidCardinalityConst =  
"InvalidCardinality";
```

This exception indicates the mismatch of the cardinality defined in an association and the number of products to be associated.

#### **2.4.2.31 UnknownAssociation**

```
const string UnknownAssociationConst =  
"UnknownAssociation";
```

This exception indicates the use of an association name that is unknown.

#### **2.4.2.32 InvalidObject**

```
const string InvalidObjectConst = "InvalidObject";
```

This exception indicates the use of a object that is inappropriate in the attempted context.

#### **2.4.2.33 UnknownCategory**

```
const string UnknownCategoryConst =  
"UnknownCategory";
```

This exception indicates the use of a category that is inappropriate in the attempted context.

#### **2.4.2.34 InvalidEvent**

```
const string InvalidEventConst = "InvalidEvent";
```

This exception indicates the use of an event that is inappropriate or unknown.

#### **2.4.2.35 BadUpdateAttribute**

```
const string BadUpdateAttributeConst =  
"BadUpdateAttribute";
```

This exception indicates an attempt to update an attribute that is inappropriate.

#### **2.4.2.36 BadEmailAddress**

```
const string BadEmailAddressConst =  
"BadEmailAddress";
```

This exception indicates the client supplied an unusable email address.

## **2.4.3 ProcessingFault Exceptions (j/NPS)**

### **2.4.3.1 ProductUnavailable**

```
const string ProductUnavailableConst =  
"ProductUnavailable";
```

This exception indicates that a product requested by a client is *currently* unavailable for access

### **2.4.3.2 ProductLocked**

```
const string ProductLockedConst = "ProductLocked";
```

This exception indicates that an attempt was made to update or delete a locked product.

### **2.4.3.3 UnsafeUpdate**

```
const string UnsafeUpdateConst = "UnsafeUpdate";
```

This exception indicates an attempt was made to perform an update without first locking the entry or entries being modified.

### **2.4.3.4 LockUnavailable**

```
const string LockUnavailableConst =  
"LockUnavailable";
```

This exception indicates that the lock requested is not allowed or supported.

## **2.4.4 SystemFault Exceptions (j/NPS)**

### **2.4.4.1 GeneralSystemFault (j/NPS)**

```
const string GeneralSystemFaultConst =  
"GeneralSystemFault";
```

This exception indicates that a server encountered an internal error possibly unrelated to the Request.

### 3. Callback (j/NPS)

The module Callback is compiled as a separate IDL file.

#### 3.1. Callback (j/NPS)

```
module CB
{
interface Callback
{
void notify (in UCO::State theState, in
UCO::RequestDescription description)

raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

void release ()

raises (UCO::ProcessingFault, UCO::SystemFault);

};
};
```

The *Callback* interface explicitly provides one of three mechanisms for client-server communication between Request objects (i.e., call back, polling/asynchronous, or blocking/synchronous). The other communication mechanisms are implicit. The Callback interface is contained in its own module (CB) . It is not contained within the GIAS module.

##### 3.1.1. notify (j/NPS)

```
void notify (in UCO::State theState, in
UCO::RequestDescription description)

raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);
```

This operation notifies the *Callback* that it has been triggered or activated. A Request that has reached a state which has been defined to trigger a Callback (see Appendix G for the states which trigger Callbacks) invokes the notify operation on all Callbacks registered with it. A Request will activate a Callback by invoking this operation and will supply a description of the triggering Request in *description*. It will also indicate the state which the Request has entered (which triggered the notify) in the parameter *theState*.

### 3.1.2. release (j/NPS)

```
void release ()  
  
    raises (UCO::ProcessingFault, UCO::SystemFault);
```

This operation is invoked by the Request to indicate that the Callback will no longer be used (will not be notified in the future). This allows a client to release any resources associated with this Callback.

## 4. Boolean Query Syntax

### 4.1. Overview

The Boolean query syntax (BQS) is a key part of the specification of the GIAS. The intent of the BQS is to formally define the syntax for queries made on geospatial catalogs. It is necessary to define the BQS in the GIAS specification to “decouple” the interfaces used for querying from the implementation details of the catalog. For example, the BQS allows a client to interact with a geospatial catalog in a uniform way regardless of the database or database type underlying the catalog implementation, the native query language of the database and the physical schema or data model of the database. This approach has the dual benefit of simplifying the generation of queries by the client while not constraining the catalog developers in the design choices for the implementation. The catalog implementers must, however, provide the capability to translate the BQS into whatever query language and physical schema they have chosen.

### 4.2. BQS Design

The BQS is based upon the concept of an attribute-operator-value triplet called a factor. Each factor represents a condition of interest to the client. These factors can be assembled into a complete query by relating the factors with the Boolean operators “and” and “or”. BQS constructs are case insensitive and BQS logical operators on strings and expressions are case sensitive.

The formal definition of the syntax of the BQS, described in Backus-Naur Form (BNF), is detailed below.

### 4.3. BNF definition

The Backus-Naur Form (BNF) for the Boolean query syntax is show below. Note that the individual BQS tokens in a BQS statement are separated from each other by a space.

```

query ::= term { "or" term }

term ::= factor { "and" factor }

factor ::= ["not"] primary

primary ::= ( simple_attribute_name comp_op constant_expressior
              | ( geo_attribute_name geo_op geo_element )
              | ( geo_attribute_name rel_geo_op number
                dist_units "of" geo_element )
              | ( text_attribute_name [ "not" ] "like"
                quoted_string )

```

```
| ( attribute_name "exists" )  
| ( "(" query ")" )
```

attribute ::= a member of the set of queryable attribute names (defined in the appropriate GIAS profile)

attribute\_name ::= attribute | {entity ":"}entity.attribute

simple\_attribute\_name ::= member of subset of attribute\_name for which boolean operators (comp\_op) are allowed

geo\_attribute\_name ::= member of subset of attribute\_name for which geospatial operators are allowed

text\_attribute\_name ::= member of subset of attribute\_name for which string operators are allowed ("free text search")

comp\_op ::= "=" | "<" | ">" | "<>" | "<=" | ">="

constant\_expression ::= number | quoted\_string

date ::= "'" year "/" month "/" day "[<blank>" hour ":" minute ":" second]"'

year ::= digit digit digit digit

month ::= digit digit

day ::= digit digit

hour ::= digit digit

minute ::= digit digit

second ::= digit digit [ "." digit {digit} ]

geo\_op ::= "intersect" | "outside" | "inside"

rel\_geo\_op ::= "within" | "beyond"

dist\_units ::= "feet" | "meters" | "statute miles" |  
"nautical miles" | "kilometers"

geo\_element ::= point | polygon | rectangle | circle  
| ellipse | line | polygon\_set | 3dpoint

sign ::= "+" | "-"

number ::= [sign] n [ "." [ n ] ]

```
n ::= digit { digit }
```

```
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |  
"7" | "8" | "9"
```

```
quoted_string ::= "'" { character } "'" // Single  
quotes
```

```
character ::= "a"|"b"| .... // All printable ASCII  
characters To use a "'"  
(single quote) use "'" (two single quotes)
```

```
Del = "," // Delimiter
```

```
latitude ::= number
```

```
longitude ::= number
```

```
altitude ::= number
```

```
hemi ::= "N" | "S" | "E" | "W"
```

```
DMS ::= [digit] digit digit ":" digit digit ":"  
digit digit "." digit hemi
```

```
latitude_DMS ::= DMS
```

```
longitude_DMS ::= DMS
```

```
latlon ::= latitude Del longitude | latitude_DMS Del  
longitude_DMS
```

```
coordinate ::= latlon
```

```
point ::= "POINT" "(" coordinate ")"
```

```
3dpoint ::= "3DPOINT" "(" coordinate Del altitude  
")"
```

```
polygon ::= "POLYGON" "(" coordinate Del coordinate  
Del coordinate {Del coordinate} ")"
```

```
rectangle ::= "RECTANGLE" "(" upper_left Del  
lower_right  ")"
```

```
upper_left ::= coordinate
```

```
lower_right ::= coordinate
```

```
circle ::= "CIRCLE" "(" coordinate Del radius ")"
```

```
units ::= "METERS" | "FEET"
```

```
radius ::= number units
```

```
ellipse ::= "ELLIPSE" "(" coordinate Del  
major_axis_len Del minor_axis_len Del north_angle ")"
```

```
major_axis_len ::= number units
```

```
minor_axis_len ::= number units
```

```
north_angle ::= number
```

```
line ::= "LINE" "(" coordinate Del coordinate { Del  
coordinate} ")"
```

```
polygon_set ::= "POLYGON_SET" "(" polygon { Del  
polygon} ")"
```

## 4.4. Rules and Constraints

The BNF rules are augmented by the following rules and constraints:

### 4.4.1. Operator Precedence

The order of precedence for the operators defined in the BQS , from high (evaluated first) to low (evaluated last) is :

( ) - parentheses – highest precedence

comp\_op, geo\_op, rel\_geo\_op, “like”, “not like”, exists

not

and

UNCLASSIFIED

or – lowest precedence

Operators of equal precedence are evaluated from left to right within an expression.

#### 4.4.2. Units

The units applicable to the attributes in a BQS query are those specified by the server implementation. This information is available via the AttributeInformation structure for each attribute. The AttributeInformation structures are accessed through the DataModelMgr.

#### 4.4.3. Strings and Wildcards

Wildcard expressions are allowed using the character "%" to denote a match with 0 or more characters and the character "?" to match with exactly one character. For example the query:

```
name like 'rob%'
```

would match the following strings:

```
'rob' 'robert' 'robin'
```

where the query

```
name like 'mik?'
```

would match the following strings

```
'mike' 'miki' 'miko'
```

The "like" and "not like" operators are the only operators used for text expressions and the only operators supporting wildcards.

Wildcards can be used 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 not like '%will_not_contain_this%'
```

```
attribute not like 'will_not_begin_with_this%'
```

```
attribute not like '%will_not_end_with_this'
```

#### 4.4.4. BQS and UCOS/GIAS Types

To abet developers implementing BQS query parsers, the following tables (See Table 4-1 and Table 4-2) provide a mapping between the above BNF and UCOS data structures.

UNCLASSIFIED



**Table 4-1 Mapping between BQS BNF and UCOS Data Structures**

<b>BQS</b>	<b>UCOS</b>
Point	Coordinate2d
Coordinate	
Latlon	
Latitude	x
Longitude	y
3dpoint	Coordinate3d
Coordinate	
Latlon	
Latitude	x
Longitude	y
Altitude	z
Polygon (collection of lines)	Polygon (collection of Coordinate2d)
Coordinate	
Latlon	
Latitude	x
Longitude	y
Rectangle	Rectangle
	Coordinate2d
upper_left	upper_left
Coordinate	
Latlon	
	Coordinate2d
Latitude	x
Longitude	y
Lower_right	lower_right
Coordinate	
Latlon	
Latitude	x
Longitude	y

**Table 4-2 Mapping between BQS BNF and UCOS Data Structures**

<b>BQS</b>	<b>UCOS</b>
Circle	Circle
	centerpoint
Coordinate	
Latlon	
Latitude	x
Longitude	y
Radius	radius
Number	dimension
	units
	Reference_system
Ellipse	Coordinate3d
	centerpoint
Coordinate	
Latlon	
Latitude	x
Longitude	y
Minor_axis_len	minor_axis_len
Number	dimension
	units
	Reference_system
Major_axis_len	major_axis_len
Number	dimension
	units
	Reference_system
north_angle	north_angle
Number	float
Line	LineString2d
Coordinate	
Latlon	
Latitude	x
Longitude	y

Coordinate	
Latlon	
Latitude	x
Longitude	y
polygon_set (collection of polygon's)	PolygonSet (collection of Polygon's)

Also note that attributes of type Boolean should be described as a *constant\_expression* in its quoted string form i.e.

attribute = 'TRUE'

#### 4.4.5. Deriving attribute names from data model

The GIAS query services (CatalogMgr, StandingQueryMgr and QueryOrderMgr services) require the identification of attributes both as elements of queries and as elements to be returned from a query. The BQS defined above allows for the specification of the query terms, or selection criteria, using the queryable attribute names from the data model that underlies the implementation. These attributes are available through the methods on the DataModelMgr. However, the BQS does not allow for the specification of relationship routes. Hence, when there are two or more relationship routes between entities containing queryable attributes that the user wishes to query, the BQS and queryable attribute list is insufficient to resolve the route ambiguity. The preferred solution to the route ambiguity would not require the user to have knowledge of the potential routes, nor of the underlying data model structure.

The GIAS allows for the identification of a queryable attribute by a unique attribute name and a name that is familiar to the user (an alias), via the Data Model Manager services. The current Data Model Manager meta-model accommodates only one unique attribute name for each queryable attribute. However, a single user-selectable attribute is insufficient to identify the route and/or role context. Therefore in order to allow an attribute to be used in the correct context (i.e. the route) the following syntax rule for the queryable attribute names is specified that conveys route/role information. Applying this rule to elements of the underlying data model generates attributes that can be used by the CatalogMgr services without route ambiguity. Note that this rule is used for attribute query submittal, result attributes, sort attributes and parameters associated with ProductMgr::get\_parameters operation.

#### 4.4.6. Attribute Name Syntax Rule

The syntax for the form of attribute names is defined in the appropriate GIAS profile.

## Appendix A: GIAS IDL

```

//*****
//*
//*           The Geospatial and Imagery Access Service
//*
//*
//* Description: Defines the data types and interfaces needed
//* to support search, retrieval and access to geospatial
//* data such as images, maps charts and their supporting
//* data
//*
//*
//*
//* History:
//* Date           Author           Comment
//* -----
//* 15 May 97      D. Lutz           Initial release for review
//* 2 July 97      D. Lutz           Released for TEM Review
//* 11 July 97     D. Lutz           Changes based on 2 July TEM
//* 18 July 97     D. Lutz           Released for NIMA CCB
//* 24 Oct 97      D. Lutz           Changes based on 7 Oct TEM
//* 14 Nov 97      D. Lutz           Changes based on 4 Nov TEM
//* 17 Dec 97      D. Lutz           Changes based on 9 Dec TEM
//* 15 Apr 98      J. Baldo           changes based on Mar TEM
//* 7 May 98       D.Lutz           Changes based on 1 May TEM
//* 2 Jul 98       J. Baldo/D. Lutz Changes based
//*                on 22-23 Jun TEM Requests - GIAS 3.2
//* 2 Jul 98 (J. Baldo): Callback module has been removed
//*                from previous GIAS 3.2 specification release
//* 5 June 1998 and will be included in GIAS 3.3
//* 5 Nov 98       D. Lutz           Added first version of
UpdateMgr

```

UNCLASSIFIED

```

/**
/** 10 Mar 99          J. Baldo          Changes based on March 99 TEM
/**
/**      5 August      D. Lutz          Mods from 3-4 August UIP WG.
/** 18 Februray 2000  D. Lutz          New Generic Exception Model
/**
/**
/**
/**
/**
/*******

```

```

/*******
/**      The USIGS Common Object Specification (UCOS) contains
/**      all the basic data types and interfaces common across
/**              USIGS
/*******

```

```

#include "uco.idl"
#include "cb.idl"
#include "uid.idl"

```

```

/*******
/**
/**      Module GIAS
/**
/**
/**      Description: The main module for the Geospatial & Imagery
/**              Access Service
/**
/**
/*******

```

```

module GIAS

```

```
{

//Forward references for all interfaces, just for convenience

// The Library itself
    interface Library;

// Abstract classes that help define the managers
    interface LibraryManager;
    interface RequestManager;
    interface AccessManager;

// Specific managers defined
    interface OrderMgr;
    interface CreationMgr;
    interface UpdateMgr;
    interface CatalogMgr;
    interface StandingQueryMgr;
interface ProductMgr;
    interface IngestMgr;
    interface QueryOrderMgr;
    interface DataModelMgr;
//interface VideoMgr;

// The abstract request objects
    interface Request;

// Specific requests defined
    interface OrderRequest;
    interface CreateRequest;
    interface CreateMetaDataRequest;
    interface UpdateRequest;
```

```
interface SubmitQueryRequest;
interface SubmitStandingQueryRequest;
interface SetAvailabilityRequest;
interface HitCountRequest;
interface GetParametersRequest;
interface IngestRequest;
interface SubmitQueryOrderRequest;
interface GetRelatedFilesRequest;
interface CreateAssociationRequest;
interface UpdateByQueryRequest;

//*****
//*      DataTypes re-used from UCOS
//*****

typedef UCO::NameValueList PropertyList;

typedef UCO::Rectangle GeoRegion;
enum GeoRegionType {
    LINE_SAMPLE_FULL,
    LINE_SAMPLE_CHIP,
    LAT_LON ,
    ALL,
    NULL_REGION};

//*****
//*      GIAS specific data types
//*****
```

```
enum AvailabilityRequirement
{
    REQUIRED, NOT_REQUIRED
};

typedef string UseMode;

typedef sequence <short> RsetList;

enum OrderType {STANDING, IMMEDIATE};

typedef any ProductSpec;

typedef string ProductFormat;
typedef string ImageUniqueIdentifier;
typedef string ImageFormat;
typedef string Compression;
typedef short BitsPerPixel;
typedef string Algorithm;
enum SupportDataEncoding {ASCII, EBCDIC};

typedef sequence < ProductFormat > ProductFormatList;
struct ImageSpec
{
    ImageFormat imgform;
    ImageUniqueIdentifier; imageid;
    Compression comp;
    BitsPerPixel bpp;
    Algorithm algo;
    RsetList rrds;
    GeoRegion sub_section;
    GeoRegionType geo_region_type;
    SupportDataEncoding encoding;
```

```
};  
typedef sequence < ImageSpec > ImageSpecList;  
  
struct AlterationSpec  
{  
    ProductFormat pf;  
    ProductSpec ps;  
    GeoRegion sub_section;  
    GeoRegionType geo_region_type;  
};  
typedef sequence < AlterationSpec > AlterationSpecList;  
  
struct PackagingSpec  
{  
    string package_identifier;  
    string packaging_format_and_compression;  
};  
  
struct TailoringSpec {  
    UCO::NameNameList specs;  
};  
  
struct MediaType  
{  
    string media_type;  
    unsigned short quantity;  
};  
  
typedef sequence < MediaType > MediaTypeList;  
  
struct PhysicalDelivery  
{
```

```
string address;
};

enum DestinationType
{
FTP, EMAIL, PHYSICAL
};

union Destination switch (DestinationType)
{
case FTP:          UCO::FileLocation f_dest;
case EMAIL:       UCO::EmailAddress e_dest;
case PHYSICAL:    PhysicalDelivery h_dest;
};

typedef sequence < Destination > DestinationList;

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

typedef sequence < ValidationResults > ValidationResultsList;

typedef UCO::Name RelatedFileType;
typedef sequence<RelatedFileType> RelatedFileTypeList;
struct RelatedFile
{
RelatedFileType file_type;
```

```
    UCO::FileLocation location;
};
typedef sequence <RelatedFile> RelatedFileList;

enum ConceptualAttributeType
{
    FOOTPRINT, CLASSIFICATION, OVERVIEW, THUMBNAIL, DATASETTYPE,
    MODIFICATIONDATE, PRODUCTTITLE, DIRECTACCESS,
    DIRECTACCESSPROTOCOL, UNIQUEIDENTIFIER, DATASIZE};

typedef string Entity;
typedef string ViewName;
typedef sequence< ViewName > ViewNameList;
struct View {
    ViewName    view_name;
    boolean    orderable;
    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
{
    UCO::AbsTime earliest;
    UCO::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:                       UCO::NameList l;
    case ORDERED_LIST:               UCO::NameList ol;
    case INTEGER_RANGE:              IntegerRange ir;
    case FLOATING_POINT_RANGE:       FloatingPointRange fr;
    case GEOGRAPHIC:                 UCO::Rectangle g;
```

```
    case GEOGRAPHIC_SET:                UCO::RectangleList gs;
    case BINARY_DATA:                   UCO::BinData bd;

    case BOOLEAN_VALUE:                 boolean bv;

};
```

```
enum AttributeType
{
    TEXT,
    INTEGER,
    FLOATING_POINT,
    UCOS_COORDINATE,
    UCOS_POLYGON,
    UCOS_ABS_TIME,
    UCOS_RECTANGLE,
    UCOS_SIMPLE_GS_IMAGE,
    UCOS_SIMPLE_C_IMAGE,
    UCOS_COMPRESSED_IMAGE,
    UCOS_HEIGHT,
    UCOS_ELEVATION,
    UCOS_DISTANCE,
    UCOS_PERCENTAGE,
    UCOS_RATIO,
    UCOS_ANGLE,
    UCOS_FILE_SIZE,
    UCOS_FILE_LOCATION,
    UCOS_COUNT,
    UCOS_WEIGHT,
    UCOS_DATE,
    UCOS_LINestring,
    UCOS_DATA_RATE,
    UCOS_BIN_DATA,
    BOOLEAN_DATA,
    UCOS_DURATION
}
```

```
};

enum RequirementMode
{
    MANDATORY, OPTIONAL
};

struct AttributeInformation
{
    string attribute_name;
    AttributeType attribute_type;
    Domain attribute_domain;
    string attribute_units;
    string attribute_reference;
    RequirementMode mode;
    string description;
    boolean sortable;
    boolean updateable;
};

typedef sequence < AttributeInformation > AttributeInformationList;
struct Association {
    string name;
    ViewName view_a;
    ViewName view_b;
    string description;
    UCO::Cardinality card;
    AttributeInformationList attribute_info;
};

typedef sequence <Association> AssociationList;
typedef sequence < Library > LibraryList;
```

```
typedef string ManagerType;
    typedef sequence < ManagerType > ManagerTypeList;

typedef sequence < Request > RequestList;

typedef sequence < UseMode > UseModeList;

struct LibraryDescription
{
    string library_name;
    string library_description;
    string library_version_number;
};
typedef sequence < LibraryDescription > LibraryDescriptionList;

struct Query{
    ViewName view;
    string bqs_query;
};

typedef UCO::DAGList QueryResults;

enum NamedEventType
{
    START_EVENT,
    STOP_EVENT,
    FREQUENCY_EVENT
};
```

```
struct Event {
    string event_name;
    NamedEventType event_type;
    string event_description;
};

typedef sequence < Event > EventList;

enum DayEvent { MON, TUE, WED, THU, FRI, SAT, SUN, FIRST_OF_MONTH,
END_OF_MONTH };

struct DayEventTime
{
    DayEvent      day_event;
    UCO::Time     time;
};

enum LifeEventType {ABSOLUTE_TIME, DAY_EVENT_TIME, NAMED_EVENT,
RELATIVE_TIME};

union LifeEvent switch ( LifeEventType)
{
    case ABSOLUTE_TIME: UCO::AbsTime at;
    case DAY_EVENT_TIME: DayEventTime day_event;
    case NAMED_EVENT: string ev;
    case RELATIVE_TIME: UCO::Time rt;
};

typedef sequence < LifeEvent > LifeEventList;

struct QueryLifeSpan {
    LifeEvent start;
    LifeEvent stop;
    LifeEventList frequency;
};
```

```
};
```

```
enum Polarity { ASCENDING, DESCENDING };
```

```
struct SortAttribute  
{  
    UCO::Name    attribute_name;  
    Polarity    sort_polarity;  
};
```

```
typedef sequence < SortAttribute > SortAttributeList;
```

```
struct DelayEstimate {  
    unsigned long time_delay;  
    boolean valid_time_delay;  
};
```

```
struct ProductDetails {  
    MediaTypeList mTypes;  
    UCO::NameList benums;  
    AlterationSpec aSpec;  
    UID::Product aProduct;  
    string info_system_name;  
};
```

```
typedef sequence <ProductDetails> ProductDetailsList;
```

```
struct DeliveryDetails {  
    Destination dests;  
    string receiver;  
    string shipmentMode;  
};
```

```
typedef sequence < DeliveryDetails > DeliveryDetailsList;
```

```
struct OrderContents {
    string originator;
    TailoringSpec tSpec;
    PackagingSpec pSpec;
    UCO::AbsTime needByDate;
    string operatorNote;
    short orderPriority;
    ProductDetailsList prod_list;
    DeliveryDetailsList del_list;
};

struct QueryOrderContents {
    string originator;
    TailoringSpec tSpec;
    PackagingSpec pSpec;
    string operatorNote;
    short orderPriority;
    AlterationSpec aSpec;
    DeliveryDetailsList del_list;
};

struct AccessCriteria {
    string userID;
    string password;
    string licenseKey;
};

struct PackageElement {
    UID::Product prod;
    UCO::NameList files;
};

typedef sequence< PackageElement > PackageElementList;
```

```
struct DeliveryManifest {
    string package_name;
    PackageElementList elements;
};

typedef sequence<DeliveryManifest>
DeliveryManifestList;

typedef string CallbackID;

//*****
//*
//*          The Exceptions Identifiers
//*
//*    Note: Three sets of IDL Strings Constants are being used as
//*
//*    the Exceptions for the GIAS
//*****

//          UCO::InvalidInputParameter Exceptions
const string BadAccessCriteriaConst = "BadAccessCriteria";
const string BadAccessValueConst = "BadAccessValue";
const string BadCreationAttributeValueConst =
"BadCreationAttributeValue";
const string BadEmailAddressConst = "BadEmailAddress";
const string BadGeoRegionConst = "BadGeoRegion";
const string BadLocationConst = "BadLocation";
const string BadPropertyValueConst = "BadPropertyValue";
const string BadQueryConst = "BadQuery";
const string BadQueryAttributeConst = "BadQueryAttribute";
const string BadQueryValueConst = "BadQueryValue";
const string BadTimeConst = "BadTime";
const string BadUseModeConst = "BadUseMode";
const string UnknownCallBackConst = "UnknownCallBack";
const string UnknownCreationAttributeConst = "UnknownCreationAttribute";
const string UnknownManagerTypeConst = "UnknownManagerType";
const string UnknownProductConst = "UnknownProduct";
const string UnknownPropertyConst = "UnknownProperty";
const string UnknownRequestConst = "UnknownRequest";
const string UnregisteredCallbackConst = "UnregisteredCallback";
const string UnknownUseModeConst = "UnknownUseMode";
const string BadOrderConst = "BadOrder";
const string UnknownViewNameConst = "UnknownViewName";
const string UnknownEntityConst = "UnknownEntity";
```

```

const string UnsupportedConceptualAttributeConst =
"UnsupportedConceptualAttribute";
const string NoValuesRequestedConst = "NoValuesRequested";
const string BadSortAttributeConst = "BadSortAttribute";
const string NonUpdateableAttributeConst = "NonUpdateableAttribute";
const string BadFileTypeConst = "BadFileType";
const string InvalidCardinalityConst = "InvalidCardinality";
const string UnknownAssociationConst = "UnknownAssociation";
const string InvalidObjectConst = "InvalidObject";
const string UnknownCategoryConst = "UnknownCategory";
const string InvalidEventConst = "InvalidEvent";
const string BadResultAttributeConst = "BadResultAttribute";
const string BadUpdateAttributeConst = "BadUpdateAttribute";
const string ImplementationLimitConst = "ImplementationLimit";

//                UCO::ProcessingFault Exceptions
const string ProductUnavailableConst = "ProductUnavailable";
const string LockUnavailableConst = "LockUnavailable";
const string UnsafeUpdateConst = "UnsafeUpdate";
const string ProductLockedConst = "ProductLocked";

//                UCO::SystemFault Exceptions
const string GeneralSystemFaultConst = "GeneralSystemFault";

```

```

//*****
//*                The Interfaces
//*****

//*****
//*                interface GIAS::Library.
//*
//*                Description: This object represents a Library. It
//*                provides operations to discover and acquire manager objects,
//*                which provide access to all the functionality of this
//*                Library.
//*

```

```

//*****

interface Library
{

    ManagerTypeList get_manager_types ()
        raises (UCO::ProcessingFault, UCO::SystemFault);

    LibraryManager get_manager (in ManagerType manager_type,
                                in AccessCriteria access_criteria)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

    LibraryDescription get_library_description ()
        raises (UCO::ProcessingFault, UCO::SystemFault);

    LibraryDescriptionList get_other_libraries (in AccessCriteria
access_criteria)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

};

//*****

/**      Interface GIAS::LibraryManager
/**
/**      Description: This (abstract) object defines the basic
/**      functions common to all types of managers.
/**
/**
/**
//*****

interface LibraryManager

```

```

{
UCO::NameList get_property_names ()
    raises (UCO::ProcessingFault, UCO::SystemFault);

    PropertyList get_property_values (in UCO::NameList
        desired_properties)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

    LibraryList get_libraries ()
        raises (UCO::ProcessingFault, UCO::SystemFault);
};

//*****
//*      Interface GIAS::RequestManager
//*
//*      Description: This (abstact) object defines the basic
//*      functions common to managers that use operations that
//*      generate request objects.
//*
//*
//*****

interface RequestManager
{

    RequestList get_active_requests ()
        raises (UCO::ProcessingFault, UCO::SystemFault);

    unsigned long get_default_timeout ()
        raises (UCO::ProcessingFault, UCO::SystemFault);

    void set_default_timeout (in unsigned long new_default)

```

```
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

        unsigned long get_timeout (in Request aRequest)

        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

        void set_timeout (in Request aRequest, in unsigned long
                new_lifetime)

        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

        void delete_request (in Request aRequest)

        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);
};

//*****
//*      interface GIAS:: AccessManager
//*
//*      Description: Provides functions to check and request the
//*      availability of Library products for specific purposes
//*
//*****

interface AccessManager:RequestManager
{

        UseModeList get_use_modes ()

        raises (UCO::ProcessingFault, UCO::SystemFault);

        boolean is_available (in UID::Product product, in UseMode use_mode)

        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);
```

```
// Returns the time (in seconds) estimated to put the requested product
// into the requested UseMode. DOES NOT request a change in the
// availability of product.
```

```
    unsigned long query_availability_delay (in UID::Product product,
        in AvailabilityRequirement availability_requirement,
        in UseMode use_mode)
```

```
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);
```

```
short get_number_of_priorities()
```

```
    raises (UCO::ProcessingFault, UCO::SystemFault);
```

```
SetAvailabilityRequest set_availability (in UID::ProductList products,
in AvailabilityRequirement availability_requirement, in UseMode
use_mode, in short priority)
```

```
    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);
```

```
};
```

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

```
/**                               The Managers
```

```
/**
```

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

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

```
/**      interface GIAS::QueryOrderMgr
```

```
/**      Derived from GIAS::LibraryManager and
```

```
/**      GIAS::RequestManager
```

```
/**
```

```
/**      Description: Provides operations to submit a
```

```
/**      query based order.
```

```
/**
```

```
/**
```

```

//*****

interface QueryOrderMgr:LibraryManager, RequestManager
{

    EventList get_event_descriptions()
        raises (UCO::ProcessingFault, UCO::SystemFault);

SubmitQueryOrderRequest submit_query_order (

        in Query aQuery,
        in QueryLifeSpan lifespan,
        in OrderType o_type,
        in QueryOrderContents order,
        in PropertyList properties)
    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

};

//*****
/*      interface GIAS:: VideoMgr
/*      Derived from GIAS::LibraryManager and GIAS::AccessManager
/*
/*      Description: Provides operations to retrieve video data
/*
/*      NOTE: This interface is TBR.
//*****

//interface VideoMgr : LibraryManager, AccessManager {
//};

//*****

```

UNCLASSIFIED

```

//*****
//*      interface GIAS:: OrderMgr
//*      Derived from GIAS:: LibraryManager and GIAS::AccessManager
//*
//*      Description: Provides operations to submit orders for Products
//*      contained in the Library:
//*
//*
//*
//*****

interface OrderMgr:LibraryManager, AccessManager
{

    UCO::NameList  get_package_specifications()
        raises (UCO::ProcessingFault, UCO::SystemFault);

    ValidationResult validate_order (in OrderContents order, in
PropertyList properties)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

    OrderRequest order (in OrderContents order, in PropertyList
properties)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

};

//*****
//*      interface GIAS:: DataModelMgr
//*      Derived from GIAS:: LibraryManager
//*

```

```
    /**      Description: Provides operations to discover the elements of
the
    /**      data model in use by the library
    /**
    /**
    /**
    /*******

interface DataModelMgr:LibraryManager
{
UCO::AbsTime get_data_model_date (in PropertyList properties)
    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

UCO::NameList get_alias_categories(in PropertyList properties)
    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

UCO::NameNameList get_logical_aliases(in string category, in
PropertyList properties)
raises( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

string get_logical_attribute_name (in ViewName view_name,in
ConceptualAttributeType attribute_type, in PropertyList properties,)
    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

ViewList get_view_names (in PropertyList properties)
    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

AttributeInformationList get_attributes (in ViewName view_name,in
PropertyList properties)
    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);
```

```

AttributeInformationList get_queryable_attributes (in ViewName
view_name,in PropertyList properties)

    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

    UCO::EntityGraph get_entities (in ViewName view_name,in PropertyList
properties)

    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

    AttributeInformationList get_entity_attributes (in Entity aEntity,in
PropertyList properties)

    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

AssociationList get_associations(in PropertyList properties)

    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

unsigned short get_max_vertices(in PropertyList properties)

    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);
};

//*****
/*      interface GIAS:: CreationMgr
/*      Derived from GIAS::RequestManager and
/*      GIAS::LibraryManager
/*      Description: Provides operations to request/nominate the
/*      archiving and cataloging of a new product to a Library.
/*
/*
//*****

interface CreationMgr:LibraryManager, RequestManager
{

```

```

CreateRequest create (in UCO::FileLocationList new_product, in
RelatedFileList related_files, in UCO::DAG creation_metadata, in
PropertyList properties)

```

```

    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

```

```

CreateMetaDataRequest create_metadata (in UCO::DAG
creation_metadata, in ViewName view_name, in RelatedFileList
related_files, in PropertyList properties)

```

```

    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

```

```

CreateAssociationRequest create_association( in string assoc_name,
                                           in UID::Product view_a_object,
                                           in UID::ProductList view_b_objects,
                                           in UCO::NameValueList assoc_info)

```

```

    raises ( UCO::InvalidInputParameter,
UCO::ProcessingFault, UCO::SystemFault);

```

```

};

```

```

//*****
//*      interface GIAS:: UpdateMgr
//*      Derived from GIAS:: LibraryManager, and GIAS::RequestManager
//*      Description: Provides operations to modify, extend or delete
//*      existing catalog entries in a GIAS Library.
//*
//*
//*****

```

```

interface UpdateMgr: LibraryManager, RequestManager
{
    void set_lock(in UID::Product lockedProduct)

```

```

        raises( UCO::InvalidInputParameter, UCO::ProcessingFault,
              UCO::SystemFault);

UpdateRequest update (in ViewName view, in UCO::UpdateDAGList changes,
in RelatedFileList relfiles, in PropertyList properties)

    raises( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

UpdateByQueryRequest update_by_query(in UCO::NameValue updated_attribute,
                                     in Query bqs_query,

                                     in PropertyList properties)

raises ( UCO::InvalidInputParameter, UCO::ProcessingFault, UCO::SystemFault);

void release_lock(in UID::Product lockedProduct)

    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

void delete_product(in UID::Product prod)

    raises(UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault
);
};

//*****
//*      interface GIAS:: CatalogMgr
//*      Derived from GIAS::LibraryManager and
//*      GIAS::RequestManager
//*
//*      Description: Provides operations to submit a query for
//*      processing.
//*
//*
//*****

interface CatalogMgr:LibraryManager, RequestManager
{

```

```

SubmitQueryRequest submit_query (
    in Query aQuery,
    in UCO::NameList result_attributes,
    in SortAttributeList sort_attributes,
    in PropertyList properties)

raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

HitCountRequest hit_count (in Query
                            aQuery, in PropertyList properties)
    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

};

//*****
//*      interface GIAS::StandingQueryMgr
//*      Derived from GIAS::LibraryManager and
//*      GIAS::RequestManager
//*
//*      Description: Provides operations to submit a
//*      standing query.
//*
//*
//*****

interface StandingQueryMgr:LibraryManager, RequestManager
{

```

```

    EventList get_event_descriptions()
        raises (UCO::ProcessingFault, UCO::SystemFault);

    SubmitStandingQueryRequest submit_standing_query (
        in Query aQuery,
        in UCO::NameList
result_attributes,
        in SortAttributeList
sort_attributes,
    in QueryLifeSpan lifespan,
    in PropertyList properties)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

};

//*****
//*      interface GIAS:: ProductMgr
//*      Derived from GIAS::LibraryManager and GIAS::AccessManager
//*
//*      Description: Provides operations to retrieve data about a
//*      specific data set.
//*
//*
//*****

interface ProductMgr:LibraryManager, AccessManager
{

    GetParametersRequest get_parameters (in UID::Product product, in
UCO::NameList desired_parameters, in PropertyList properties)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

```

```

RelatedFileTypeList get_related_file_types( in UID::Product prod)
    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

```

```

GetRelatedFilesRequest get_related_files ( in UID::ProductList
    products, in UCO::FileLocation location, in RelatedFileType
    type, in PropertyList properties )
    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault );

```

```
};
```

```

/*****
//*      interface GIAS:: IngestMgr
//*      Derived from GIAS::LibraryManager and
//*      GIAS::RequestManager
//*
//*      Description: Provides operations to perform bulk transfers
//*      of data between Libraries.
//*
//*
//*****

```

```

interface IngestMgr:LibraryManager, RequestManager
{

```

```
// FileLocation contains a directory
```

```

    IngestRequest bulk_pull (in UCO::FileLocation location, in
PropertyList property_list)
    raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

```

```
// FileLocation contains a directory
```

```

    IngestRequest bulk_push (in Query aQuery, in UCO::FileLocation
location, in PropertyList property_list)

```

```
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

};

//*****
//*      interface GIAS:: Request
//*
//*      Description: An (abstract) object that provides operations
//*      common to all forms of requests.
//*
//*
//*
//*****

interface Request
{

    UCO::RequestDescription get_request_description ()
        raises (UCO::ProcessingFault, UCO::SystemFault);
    void set_user_info (in string message)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

    UCO::Status get_status ()
        raises (UCO::ProcessingFault, UCO::SystemFault);
    DelayEstimate get_remaining_delay ()
        raises (UCO::ProcessingFault, UCO::SystemFault);
    void cancel ()
        raises (UCO::ProcessingFault, UCO::SystemFault);
    CallbackID register_callback (in CB::Callback acallback)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);
    void free_callback (in CallbackID id)
```

```

        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

```

```

RequestManager get_request_manager ()
    raises (UCO::ProcessingFault, UCO::SystemFault);
};

```

```

//*****
/**      interface GIAS:: OrderRequest
/**      Derived from GIAS::Request
/**      Description: Returned by calls to order.
/**
/**
//*****

```

```

interface OrderRequest:Request
{
    UCO::State complete (out DeliveryManifest prods)
        raises (UCO::ProcessingFault, UCO::SystemFault);
};

```

```

//*****
/**      interface GIAS:: SubmitQueryOrderRequest
/**      Derived from GIAS::Request
/**      Description: Returned by calls to submit_query_order.
/**
/**
//*****

```

```

interface SubmitQueryOrderRequest:Request
{

void pause()
    raises (UCO::ProcessingFault, UCO::SystemFault);

```

```

void resume()
    raises (UCO::ProcessingFault, UCO::SystemFault);

UCO::State complete_list (out DeliveryManifestList prods)
    raises (UCO::ProcessingFault, UCO::SystemFault);

UCO::State complete (out DeliveryManifest prods)
    raises (UCO::ProcessingFault, UCO::SystemFault);
};

//*****
//*****
/**      interface GIAS:: CreateRequest
/**      Derived from GIAS::Request
/**
/**      Description: Returned by calls to create
/**
/**
//*****

interface CreateRequest:Request
{
UCO::State complete (out UID::ProductList new_products)
    raises (UCO::ProcessingFault, UCO::SystemFault);
};

//*****
/**      interface GIAS:: CreateMetaDataRequest
/**      Derived from GIAS::Request
/**
/**      Description: Returned by calls to create_metadata
/**

```

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

interface CreateMetaDataRequest:Request
{
UCO::State complete (out UID::Product new_product)
    raises (UCO::ProcessingFault, UCO::SystemFault);
};

/*******

/**      interface GIAS:: UpdateRequest
/**      Derived from GIAS::Request
/**
/**      Description: Returned by calls to update
/**
/**
/*******

interface UpdateRequest:Request
{

UCO::State complete ()
    raises (UCO::ProcessingFault, UCO::SystemFault);

};

/*******

/**      interface GIAS:: SubmitQueryRequest
/**      Derived from GIAS::Request
/**
/**      Description: Returned by calls to query
```

```

/**
/**
/*******

interface SubmitQueryRequest:Request
{
    void set_number_of_hits (in unsigned long hits)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);
    UCO::State complete_DAG_results (out QueryResults results)
        raises (UCO::ProcessingFault, UCO::SystemFault);
    UCO::State complete_stringDAG_results (out UCO::StringDAGList results)
        raises (UCO::ProcessingFault, UCO::SystemFault);
    UCO::State complete_XML_results (out UCO::XMLDocument results)
        raises (UCO::ProcessingFault, UCO::SystemFault);

};

/*******

/**      interface GIAS:: SubmitStandingQueryRequest
/**      Derived from GIAS::Request
/**
/**      Description: Returned by calls to submit_standing_query
/**
/**
/*******

interface SubmitStandingQueryRequest:Request
{
    void set_number_of_hits (in unsigned long hits)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

    unsigned long  get_number_of_hits()

```

```
        raises (UCO::ProcessingFault, UCO::SystemFault);

    unsigned long  get_number_of_hits_in_interval(in unsigned long
interval)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

    unsigned long  get_number_of_intervals()
        raises (UCO::ProcessingFault, UCO::SystemFault);
    void  clear_all()
        raises (UCO::ProcessingFault, UCO::SystemFault);

    void  clear_intervals(in unsigned long num_intervals)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

    void  clear_before(in UCO::Time relative_time)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

    void  pause()
        raises (UCO::ProcessingFault, UCO::SystemFault);
    void  resume()
        raises (UCO::ProcessingFault, UCO::SystemFault);
    UCO::AbsTime  get_time_last_executed()
        raises (UCO::ProcessingFault, UCO::SystemFault);
    UCO::AbsTime  get_time_next_execution()
        raises (UCO::ProcessingFault, UCO::SystemFault);
    UCO::State  complete_DAG_results (out QueryResults results)
        raises (UCO::ProcessingFault, UCO::SystemFault);
    UCO::State  complete_stringDAG_results (out UCO::StringDAGList results)
        raises (UCO::ProcessingFault, UCO::SystemFault);
    UCO::State  complete_XML_results (out UCO::XMLDocument results)
        raises (UCO::ProcessingFault, UCO::SystemFault);
};
```

UNCLASSIFIED

```

//*****
//*      interface GIAS:: SetAvailabilityRequest
//*      Derived from GIAS::Request
//*
//*      Description: Returned by calls to makeAvailable
//*
//*      `
//*****

interface SetAvailabilityRequest:Request
{
UCO::State complete ()
    raises (UCO::ProcessingFault, UCO::SystemFault);
};

//*****
//*      interface GIAS:: HitCountRequest
//*      Derived from GIAS::Request
//*
//*      Description: Returned by calls to Hitcount
//*
//*
//*****

interface HitCountRequest:Request
{
UCO::State complete (out unsigned long number_of_hits)
    raises (UCO::ProcessingFault, UCO::SystemFault);
};

//*****
//*      interface GIAS:: GetParametersRequest

```

```

/**      Derived from GIAS::Request
/**
/**      Description: Returned by calls to get_parameters
/**
/**
/*******

interface GetParametersRequest:Request
{
UCO::State complete (out UCO::DAG parameters)
    raises (UCO::ProcessingFault, UCO::SystemFault);
UCO::State complete_stringDAG (out UCO::StringDAG parameters)
    raises (UCO::ProcessingFault, UCO::SystemFault);
};

/*******

/**      interface GIAS:: IngestRequest
/**      Derived from GIAS::Request
/**
/**      Description: Returned by calls to bulk_push and bulk_pull
/**
/**
/*******

interface IngestRequest:Request
{
UCO::State complete ()
    raises (UCO::ProcessingFault, UCO::SystemFault);
};

/*******

/**      interface GIAS:: GetRelatedFilesRequest

```

```

/**      Derived from GIAS::Request
/**
/**      Description: Returned by calls to get_related_files
/**
/**
/*******

interface GetRelatedFilesRequest:Request
{
    UCO::State complete (out UCO::NameList locations)
        raises (UCO::ProcessingFault, UCO::SystemFault);
};

/*******

/**      interface GIAS:: CreateAssociationRequest
/**      Derived from GIAS::Request
/**
/**      Description: Returned by calls to create_association
/**
/**
/*******

interface CreateAssociationRequest:Request
{
    UCO::State complete ()
        raises (UCO::ProcessingFault, UCO::SystemFault);

};

/*******

/**      interface GIAS::UpdateByQueryRequest
/**      Derived from GIAS::Request
/**      Description: Returned by calls to update_by_query
/**

```

```
//*****  
interface UpdateByQueryRequest:Request  
{  
    UCO::State complete ()  
        raises (UCO::ProcessingFault, UCO::SystemFault);  
  
};  
  
};                                // end of module GIAS
```

## Appendix B: Callback IDL

```
/**
*****
/**  interface GIAS::Callback
/**
/**  Description: General callback interface
/**
/**  NOTE: The Callback interface is implemented on the
/**  "client" side to allow "servers" to notify clients of
/**  completion of requests.
/**
/**  NOTE: Callback module is now compiled as a separate IDL file. This will
/**  be changed in GIAS 3.3
*****
#include "uco.idl"
module CB
{
    interface Callback
    {
        void notify (in UCO::State theState, in
UCO::RequestDescription description)
        raises ( UCO::InvalidInputParameter, UCO::ProcessingFault,
UCO::SystemFault);

        void release ()
            raises (UCO::ProcessingFault, UCO::SystemFault);
    };
};
```

## Appendix C UML Diagrams

The GIAS IDL interface has been modeled using Unified Modeling Language (UML). A brief description of the notation used for the GIAS class diagrams was described in section 1.2. The purpose of this section is to provide a more detailed overview of UML to show the reader the “what” and “how” of the use of the various UML diagrams for analysis and modeling.

UML is based on three object-oriented modeling languages: 1) Object Modeling Technique (OMT) by James Rumbaugh; 2) Booch Method by Grady Booch; and 3) Object-oriented Software Engineering Method by Ivar Jacobson. Although all three methods had a large critical mass of users the authors were motivated to merge their modeling methods based on the following rationale:

- Their methods were evolving toward each other and already shared many commonalities.
- A common modeling language would greatly enhance communication between designers and implementers.
- A common modeling language would greatly enhance portability amongst object-oriented analysis and design tool vendors.
- A combination of the three methods would have a synergistic effect of combining lessons learned and addressing problems that the former method did not address well.

UML has been submitted as a standard modeling language to OMG and can be obtained as OMG documents ad/97-01-01 — ad/97-01-14. Based on the above rationale and potential for standardization the justification was used for using UML as the modeling language for GIAS.

UML distinguishes between the notions of model and diagram. A model contains all of the underlying elements of information about a system under consideration and does so independently of how those elements are visually presented. A diagram is a particular visualization of certain kinds of elements from a model and generally exposes only a subset of those elements' detailed information. A given model element might exist on multiple diagrams, but there is but one definition of that element in the underlying model.

UML defines notation and semantics for the following diagrams:

- class diagrams - Is a collection of (static) declarative model elements, such as classes, types, and their relationships, connected as a graph and to each other and to their contents. Class diagrams may be organized into packages either with their underlying models or as separate packages that build upon the underlying model packages.
- use-case diagrams - Is a graph of actors, a set of use cases enclosed by a system boundary communication (participation) associations between the actors and the use cases, and generalizations among the use cases.
- interaction diagrams

- sequence diagrams - Shows objects participating in a set of interactions based on their “lifelines” and the messages that they exchange arranged in time sequence.
- collaboration diagrams - Shows interactions amongst a set of objects.
- state diagrams - Is a bipartite graph of states and transitions. It shows the sequences of states that an object or an interaction goes through during its life in response to received stimuli, together with its responses and actions.
- component diagrams - Is a graph of components connected by dependency relationships. It shows aspects of implementation, including source code structure and run-time implementation structure.
- deployment diagrams - Is a graph of nodes connected by communication associations. It shows the configuration of run-time processing elements and the software components, processes, and objects that live on them.

## Appendix D Reference OMG Standard IDL

## Appendix E CORBA Standard Exceptions

```
#define ex_body {unsigned long minor;
completion_status completed;}

enum completion_status {COMPLETED_YES, COMPLETED_NO,
COMPLETED_MAYBE};

enum exception_type {NO_EXCEPTION, USER_EXCEPTION,
SYSTEM_EXCEPTION};

exception "UNKNOWN ex_body;

exception BAD_PARAM ex_body;

exception NO_MEMORY ex_body;

exception IMP_LIMIT ex_body;

exception COMM_FAILURE ex_body;

exception INV_OBJREF ex_body;

exception NO_PERMISSION ex_body;

exception INTERNAL ex_body;

exception MARSHAL ex_body;

exception INITIALIZE ex_body;

exception NO_IMPLEMENT ex_body;

exception BAD_TYPECODE ex_body;

exception BAD_OPERATION ex_body;

exception NO_RESOURCES ex_body;
```

UNCLASSIFIED

```
exception NO_RESPONSE ex_body;  
exception PERSIST_STORE ex_body;  
exception BAD_INV_ORDER ex_body;  
exception TRANSIENT ex_body;  
exception FREE_MEM ex_body;  
exception INV_IDENT ex_body;  
exception INV_FLAG ex_body;  
exception INTF_REPOS ex_body;  
exception BAD_CONTEXT ex_body;
```

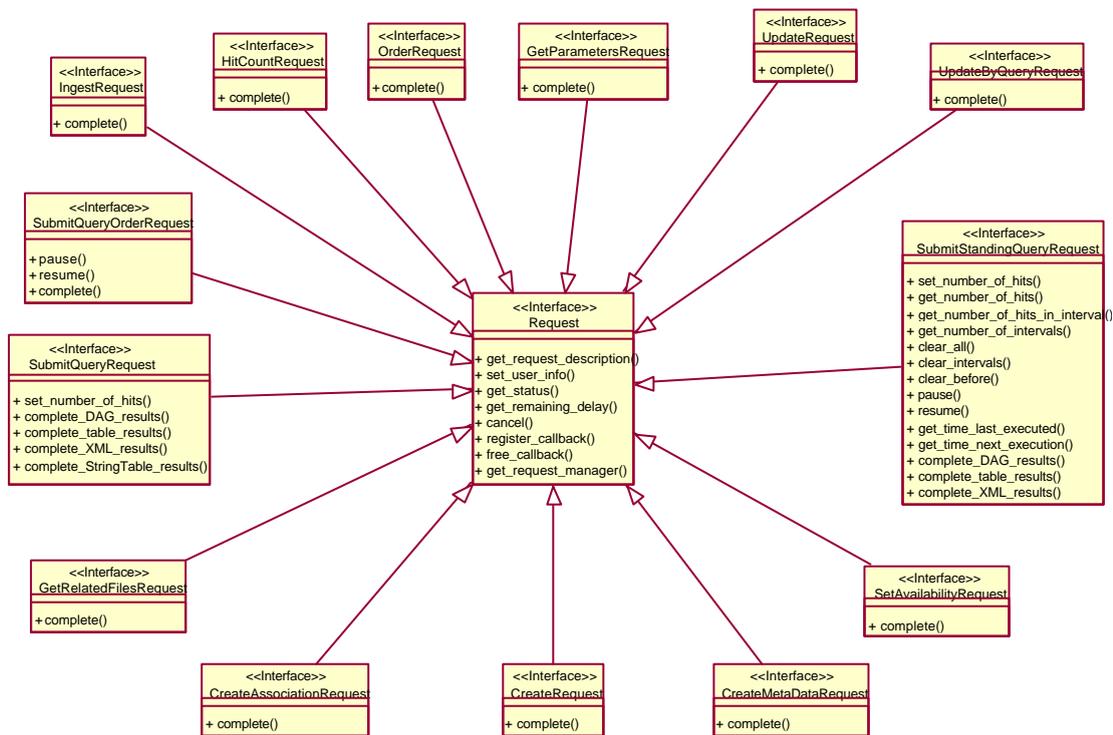
## Appendix F Acronyms

API	Application Program Interface
CIIF	Common Imagery Interoperability Facilities
CIIP	Common Imagery Interoperability Profile
CIIWG	Common Imagery Interoperability Working Group
CORBA	Common Object Request Broker Architecture
GIAS	Geospatial & Imagery Access Services
IASS	Image Access Services Specification
IDL	Interface Definition Language
ISO	International Standard Organization
NIMA	National Imagery and Mapping Agency
OGC	Open GIS Consortium
OMG	Object Management Group
TBD	To Be Determined
TBR	To Be Resolved
UCOS	USIGS Common Object Specification
UIP	USIGS Interoperability Profile
USIGS	United States Imagery and Geospatial Information System

## Appendix G: UML Statechart Diagrams

This appendix provides a set of UML statechart diagrams that describe a particular aspect of the GIAS's implementation behavior. The audience for these diagrams are targeted for developers implementing GIAS clients and developers implementing GIAS services. GIAS client developers will use these diagrams to infer GIAS service behavior and GIAS service developers will use these diagrams as a specification for behavior of GIAS services.

This appendix will evolve based on changes to the GIAS OMG IDL specification and expanded documentation of GIAS UML design and implementation descriptions. At present this section includes UML statecharts for interface generalizations of the GIAS IDL Interface Request, as shown in figure G-1<sup>1</sup>.

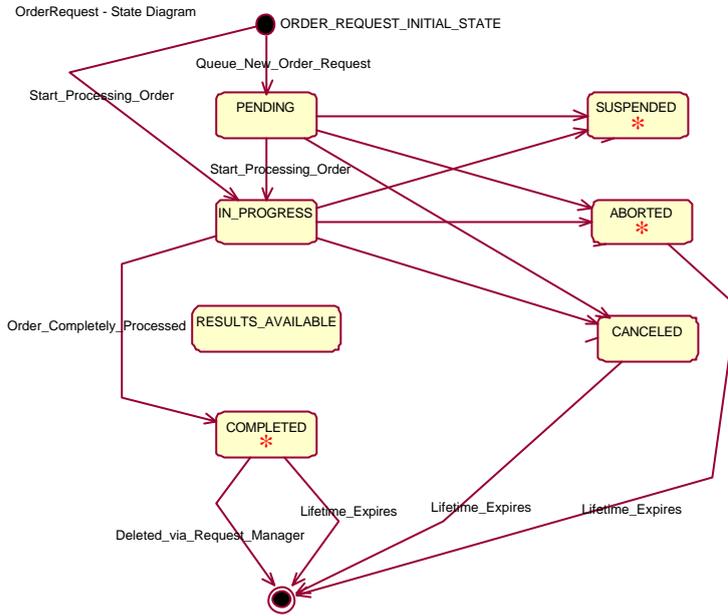


**G-1 Interface Generalization for the GIAS Interface Request**

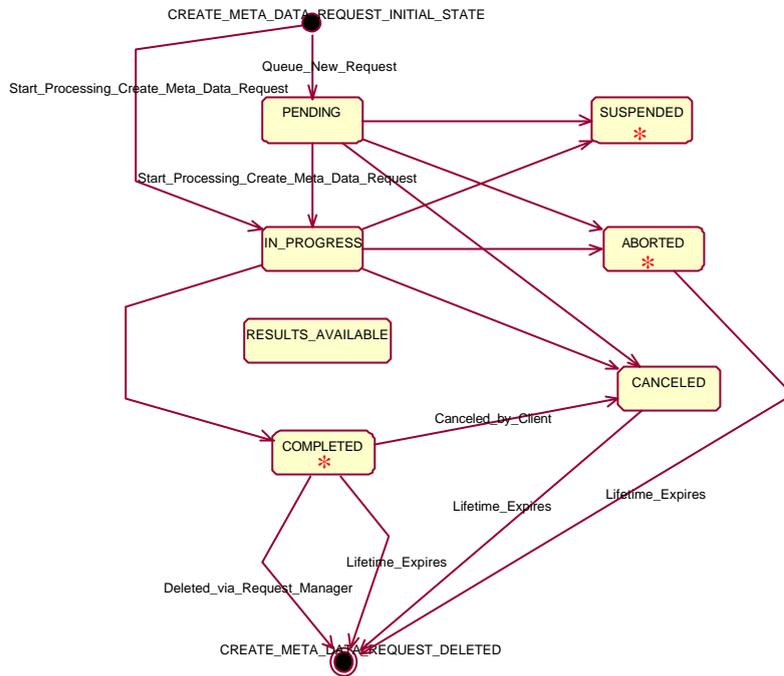
The following state diagrams reflect the state machine for each generalization of the GIAS Interface Request. States marked with an asterisk indicate that a Callback (if one has been registered with this Request) is triggered when that state is entered.

<sup>1</sup> N.B., these new State Diagrams supercede UCO:State/Status Terminal/Non-Terminal details described in previous versions of the UCOS specifications.

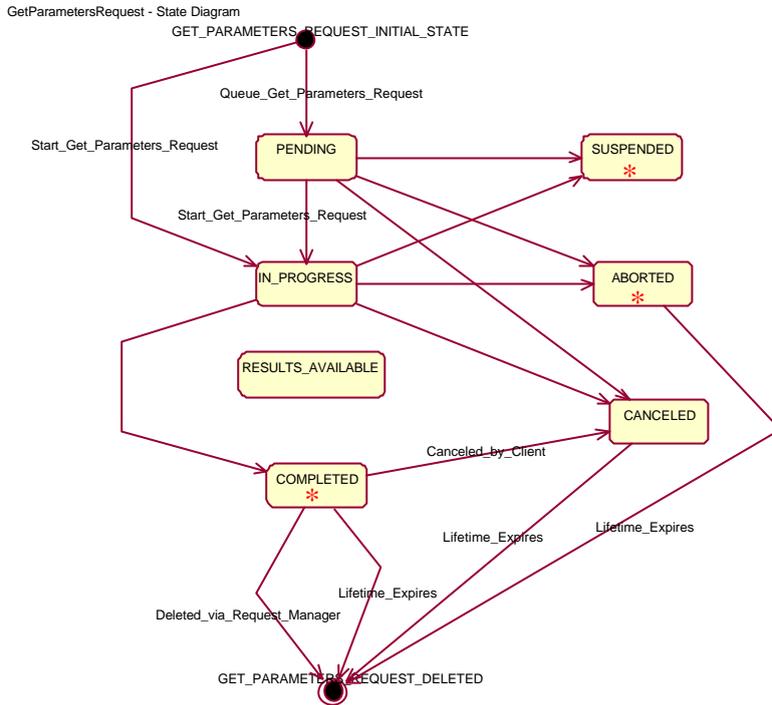




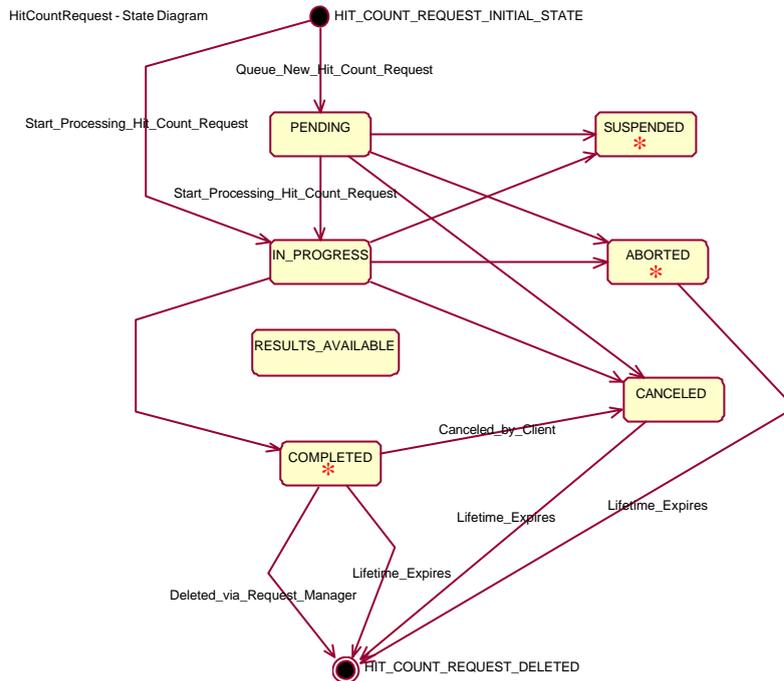
G-3 UML Statechart – OrderRequest



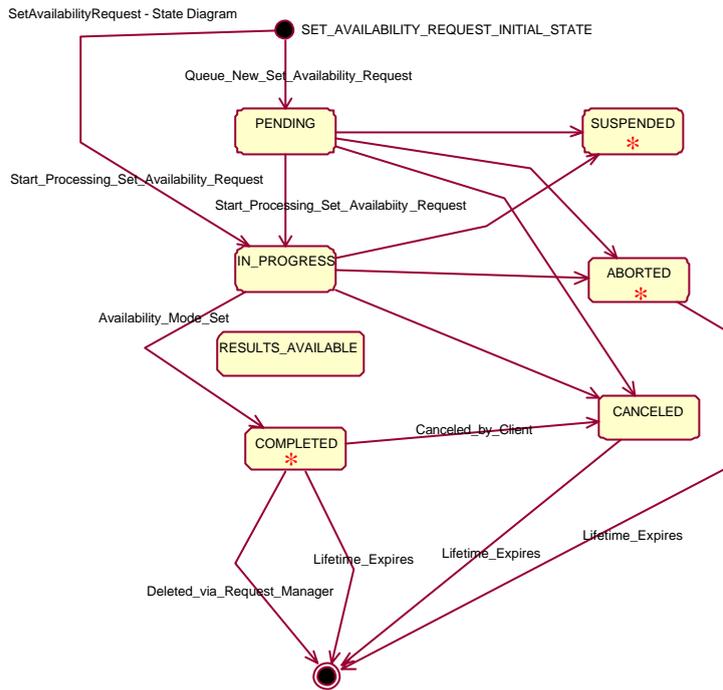
**G-4 UML Statechart CreateMetadataRequest**



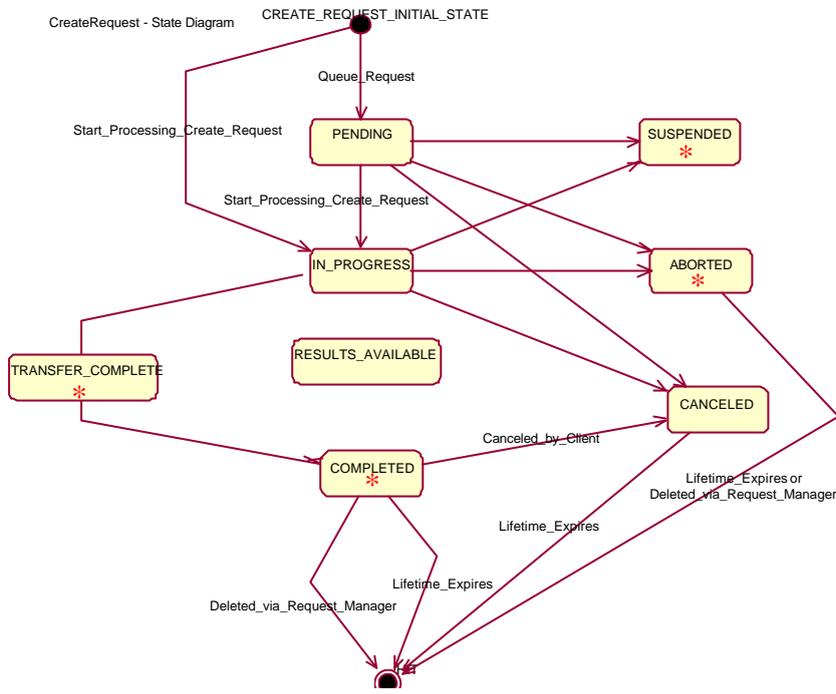
G-5 UML Statechart GetParametersRequest



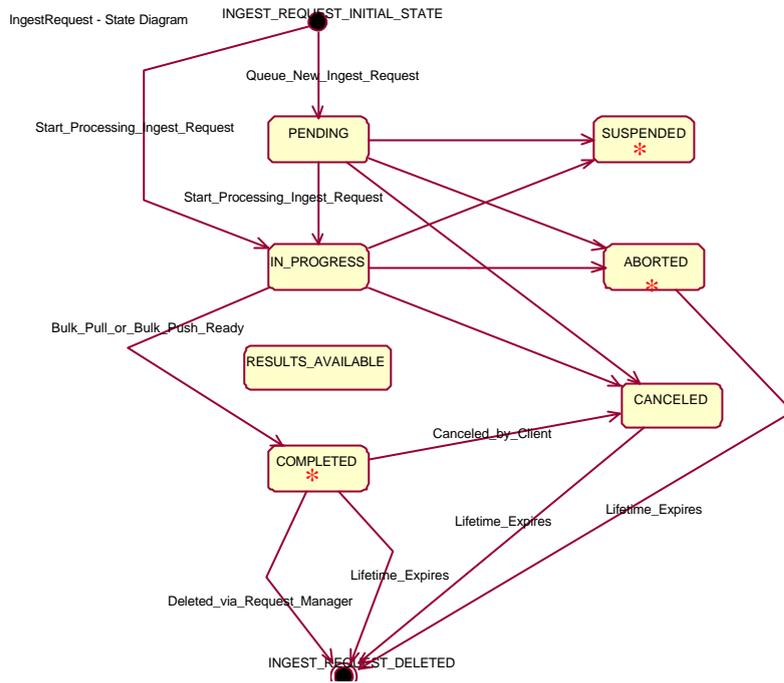
G-6 UML Statechart HitCountRequest



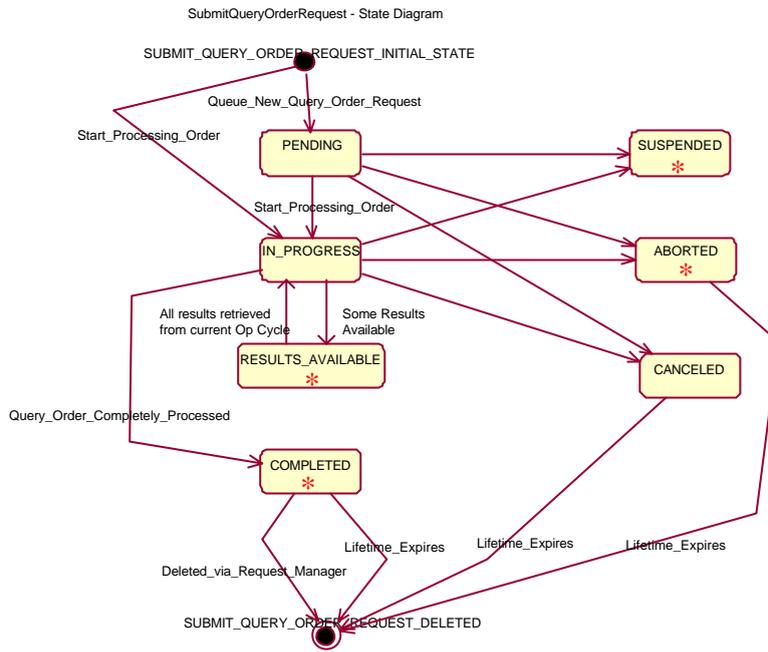
G-7 UML Statechart SetAvailabilityRequest



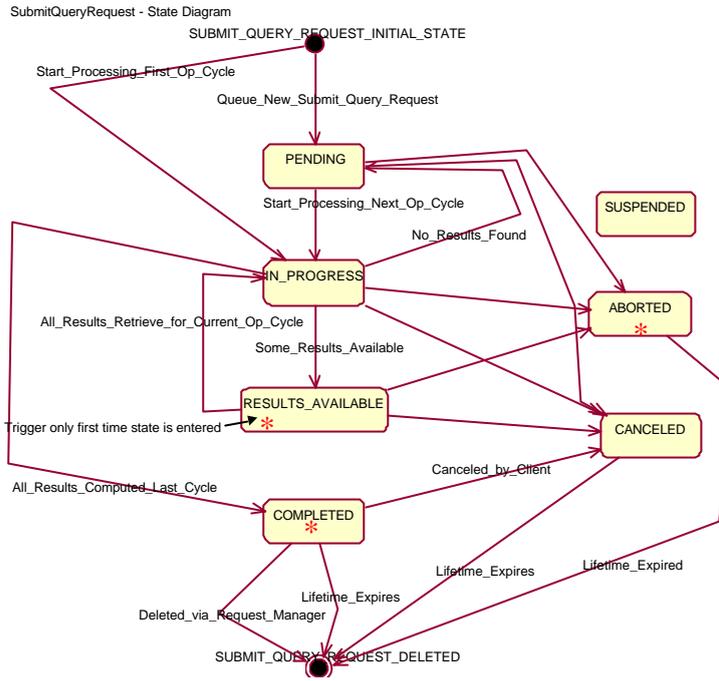
G-8 UML Statechart CreateRequest (j/NPS)



G-9 UML Statechart IngestRequest



G-10 UML Statechart SubmitQueryOrderRequest



G-11 UML Statechart SubmitQueryRequest

## Appendix H: Points of Contact

### *NIMA/ATSR*

Ron Burns, National Imagery and Mapping Agency

Phone: 703.755.5630

Email: [BurnsR@nima.mil](mailto:BurnsR@nima.mil)

### *NIMA/ATSRI*

**Bill Young**, National Imagery and Mapping Agency

Phone: 703.755.5644

Email: [YoungW@nima.mil](mailto:YoungW@nima.mil)

### *USIGS Interface Definition & Implementation*

**Charlie Green**, SI, Engineering Edge Alliance (Sierra Concepts, Inc).

Phone: 610.347.0602

Email: [cpg.sci@mindspring.com](mailto:cpg.sci@mindspring.com)

### *UCOS & GIAS Specifications, RFCs & Support*

**Dave Lutz**, The MITRE Corporation

Phone: 703.883.7848

Email: [dlutz@mitre.org](mailto:dlutz@mitre.org)

### *USIGS Interoperability Profile (UIP)*

**Bradley Bretzin**, SI, Engineering Edge Alliance (Booz•Allen & Hamilton)

Phone: 703.375.2034

Email: [bretzinb@bah.com](mailto:bretzinb@bah.com)