The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

PATENT

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

NOTICE

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

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013 OMG® and Object Management are registered trademarks of the Object Management Group, Inc. Object Request Broker, OMG IDL, ORB, CORBA, CORBAFacilities, CORBAservices, COSS, and IIOP are trademarks of the Object Management Group, Inc. X/Open is a trademark of X/Open Company Ltd.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the OMG web page http://www.omg.org, under Documents & Specifications. Report a Bug/Issue.
## Contents

Preface ................................................................. iii

1. Genomic Maps Overview ........................................ 1-1
   1.1 Specification Overview .................................... 1-1
   1.2 Compliance Points ........................................ 1-1
      1.2.1 “Base Maps” ........................................ 1-1
      1.2.2 “Nested Maps” ...................................... 1-2
      1.2.3 “Correlated Maps“ .................................. 1-2
      1.2.4 “LQSLink” ........................................ 1-2
   1.3 Document Structure ......................................... 1-2
      1.3.1 Module DsLSRControlledVocabularies ............ 1-3
      1.3.2 Module DsLSRGenomicMaps .......................... 1-4

2. General Description .............................................. 2-1
   2.1 Objects-by-value .......................................... 2-1
   2.2 Iterators .................................................. 2-2
   2.3 Controlled Vocabularies ................................. 2-3
   2.4 Identifier Strings ...................................... 2-4
   2.5 Mappable .................................................. 2-6
   2.6 Mappable and Map ........................................ 2-6
   2.7 Mappables and Assignments .............................. 2-6
   2.8 Nested Maps .............................................. 2-7
   2.9 Retrievals and Queries .................................. 2-8
      2.9.1 Retrievals ......................................... 2-8
      2.9.2 Queries ........................................... 2-8
Preface

About the Object Management Group

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

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

What is CORBA?

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

The CORBA documentation is organized as follows:

- **Object Management Architecture Guide** defines the OMG’s technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

- **CORBA: Common Object Request Broker Architecture and Specification** contains the architecture and specifications for the Object Request Broker.

- **CORBA services: Common Object Services Specification** contains specifications for OMG’s Object Services.

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

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

OMG Headquarters  
250 First Avenue, Suite 201  
Needham, MA 02494  
USA  
Tel: +1-781-444-0404  
Fax: +1-781-444-0320  
pubs@omg.org  
http://www.omg.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

- **Helvetica bold** - OMG Interface Definition Language (OMG IDL) and syntax elements.

- **Courier bold** - Programming language elements.

- **Helvetica** - Exceptions

Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.
Acknowledgments

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

- EMBL-EBI
- Genomica Corp.
- Infobiogen
- Millennium Pharmaceuticals, Inc.
- NetGenics, Inc.
- Technische Universität Berlin
Genomic Maps Overview

Contents

This chapter contains the following sections.

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Specification Overview”</td>
<td>1-1</td>
</tr>
<tr>
<td>“Compliance Points”</td>
<td>1-2</td>
</tr>
<tr>
<td>“Document Structure”</td>
<td>1-3</td>
</tr>
</tbody>
</table>

1.1 Specification Overview

This document describes a standard for representing genomic maps and their contents. It is able to deal with practically any type of chromosome map and marker that is likely to occur in the fast growing field of molecular genetics. Situations that are not catered for explicitly can be addressed by extending the types and using the conventions described in this document. The standard was developed starting from the practical need to represent complex bodies of data in a natural way. Existing practice and terminology is used wherever this was available and practical.

In this section, a synopsis of the data model and data types is given. Chapter 2 introduces the general design and the general rules that apply to the components in the standard, while also providing the rationale for the design. Chapter 3 presents the standard in detail. The full IDL is provided in Appendix A.
1.2 Compliance Points

There are four compliance points to this specification: (1) “Base Maps,” (2) “Nested Maps,” (3) “Correlated Maps,” and (4) “LQSLink.” These compliance points make use of the interfaces described in IDL modules *DsLSRControlledVocabularies*, *DsLSRGentricMaps*, and *DsLSRLQSLink*, but to varying degrees. The modules are described in detail in Chapter 3; the compliance points are given below.

A compliant Genomic Maps implementation must satisfy compliance point (1) “Base Maps” and may satisfy none or more of compliance points (2) “Nested Maps” and (3) “Correlated Maps” and (4) “LQSLink.” LQSLink is an optional compliance point.

1.2.1 “Base Maps”

Implements all modules and interfaces described in modules *DsLSRControlledVocabularies* and *DsLSRGentricMaps*, with the following exclusions:

1. “Base Maps” implementations ignore the *recursion_depth* parameter for query operations (implicitly setting it to 0).
2. “Base Maps” implementations support only the *MappableAssignment* subtype of *Assignment*.
3. “Base Maps” implementations support *Mappable* as the type of *left_flanking_entity* and *right_flanking_entity* in the interface *RelativePosition*. They are not required to support *Map* as a type for *left_flanking_entity* and *right_flanking_entity*.
4. The additional exclusions listed under the “Nested Maps” compliance point below.

“Base Maps” is a mandatory compliance point.

1.2.2 “Nested Maps”

Implements all modules and interfaces as described in modules *DsLSRControlledVocabularies* and *DsLSRGentricMaps*, with the following exclusions:

1. The interface *MapCorrelation* and its factory *MapCorrelationFactory*.
2. The interface *AssignmentPairIterator* and the data types *AssignmentPair* and *MapPair*.

“Nested Maps” is an optional compliance point.

1.2.3 “Correlated Maps“

Implements all modules and interfaces described in modules *DsLSRControlledVocabularies* and *DsLSRGentricMaps*, with the following exclusions:
1. Exclusions (1), (2), and (3) listed under “Base Maps” above.
“Correlated Maps” is an optional compliance point.

1.2.4 “LQSLink”

The interface described in the DsLSRLQSLink module is optional.

1.3 Document Structure

The specification is composed of three modules, DsLSRControlledVocabularies, DsLSRGenomicMaps, and DsLSRLQSLink.

Module DsLSRGenomicMaps defines mainly domain specific data types, such as Mappables, Map, Assignment, and Position, MapCorrelation.

DsLSRGenomicMaps needs an auxiliary module, DsLSRControlledVocabularies, that is used to define the contents of so-called controlled vocabularies. This module is described first. A separate, optional module DsLSRLQSLink can provide connectivity between DsLSRControlledVocabularies and the LexExplorer interface from the Lexicon Query Service.

UML diagrams of most of the data types are presented in Figures 1, 2, and 3. These diagrams are meant to provide an overview, and are not complete in the sense that code could be generated from them. For instance, not all iterators or factories are represented.

1.3.1 Module DsLSRControlledVocabularies

A diagram of the data model for this module is given in Figure 1-1. Its data types are briefly discussed.

![UML diagram of data types of module DsLSRControlledVocabularies](image)

Figure 1-1 UML diagram of data types of module DsLSRControlledVocabularies

1.3.1.1 VocabularyString

This data type presents a notion intermediate between an enum and a string. They are used to represent relatively fixed string values that are more or less well known and usually specific to the domain. Their values are defined as the contents of Vocabulary.
a type defined in the \texttt{DsLSRControlledVocabularies} module. To make the intended use of a string variable clearer, a \texttt{typedef} \texttt{string} \texttt{VocabularyString} is provided and used in this specification.

1.3.1.2 \textit{VocabularyEntry}

This \texttt{valuetype} represents the contents of a \texttt{Vocabulary}, and consists of the vocabulary string along with a description.

1.3.1.3 \textit{Vocabulary}

This interface represents a set of strings that are valid in a particular context. It is a container for \texttt{VocabularyEntry}s. Objects of this data type are candidates for registration with a Naming Server or a Trader.

1.3.1.4 \textit{VocabularyFinder}

An interface that gives access to \texttt{Vocabulary}s. Each context can have a number of different relevant \texttt{Vocabulary}s, all of which can be represented by one \texttt{VocabularyFinder}.

1.3.2 Module \textit{DsLSRGenomicMaps}

UML diagrams of the data model of the mapping data types in this module are given in Figure 1-2 on page 1-5; the types for representing map correlations are depicted in Figure 1-3 on page 1-7. The data types are briefly discussed below.

1.3.2.1 \textit{Identifier}

Many entities in molecular biology require ID strings, usually to uniquely identify them in a certain context. The current specification also uses strings for ID attributes, but constrains their syntax and semantics to improve interoperability. To make the intended use of such string variables clearer, \texttt{typedef} \texttt{string} \texttt{Identifier} is provided and used in this specification.

1.3.2.2 \textit{QueryString}

This data type (again, a \texttt{typedef} \texttt{string}, for the same reasons as described earlier) is used to represent fixed query types to the \texttt{evaluate()} method that is inherited from \texttt{CosQuery::QueryEvaluator}.

1.3.2.3 \textit{Mappable}

The \texttt{valuetype} \texttt{Mappable} is used to represent the contents of a \texttt{Map}. It represents the \texttt{MappedEntity} mentioned in the RFP. Mostly, \texttt{Mappables} will be simple markers. However, since maps can be nested, a nested map or sub-map is also 'map content' (namely of the enclosing, or nesting map). The nestability of maps allows clones,
contigs or even genes and sequences to be both markers as well as maps. In other words, Map could be regarded as a specialization of Mappable. This relationship is however not represented by IDL inheritance but by delegation as will be explained below.

Mappable has no information on where it is located on a map; this is the task of the Assignment data types. There are no sub-types of Mappable defined.

Figure 1-2  UML diagram of mapping data types in module DsLRSGenomicMaps

1.3.2.4 Map

The data type Map and its sub-types are used to represent genomic maps. It has retrieval methods, and a number of fixed QueryStrings that can be used as the input to the evaluate() method inherited from CosQuery::QueryEvaluator. A number of specific sub-types of Map are provided. As described in the previous paragraph on Mappable, Map can be regarded as being a specialization of Mappable. Although
this is-a relationship is an inheritance relationship, this standard represents it as a
delegation. The reason is that IDL syntax (i.e., sub-entity: super-entity { ... }) cannot be used to make interface Map inherit from the valuetype Mappable.

For clarity, the remainder of this specification uses the term sub-Map whenever referring to a Map that is nested inside another Map. However, in any other respect, sub-Map is exactly the same type as Map.

### 1.3.2.5 MapFactory

Map objects can be obtained from a MapFactory object. Objects of this data type are candidates for registration with a Naming Server or a Trader. Methods for the retrieval of Maps as well as a number fixed query strings are provided.

### 1.3.2.6 Assignment

An Assignment is the placement of a particular Mappable or sub-Map on a
particular Map; it holds the information concerning the location(s) of a Mappable on
a Map. The difficulty of representing either a Mappable or a sub-Map in an
Assignment is solved by having corresponding sub-types MappableAssignment
and SubMapAssignment. Assignments are never used directly; only their
specializations MappableAssignment or SubMapAssignment are.

### 1.3.2.7 Position

The geometric information of an Assignment. A number of sub-types are provided to
deal with different kinds of maps and assignments.

### 1.3.2.8 MapCorrelation

This data type is used to hold the information needed to correlate two different maps.

A UML diagram of the data types that represent correlations between two maps is
given in Figure 1-3 on page 1-7. The data types Map and Assignment are the same
as those in Figure 1-2 on page 1-5.

### 1.3.2.9 MapCorrelationFactory

MapCorrelations are obtained by querying a MapCorrelationFactory object. As with
Map and MapFactory, fixed query strings to be used as input for the evaluate() method inherited from CosQuery::QueryEvaluator are given as well. Like
MapFactory, it will usually be registered with a Naming or Trader service.
Figure 1-3  UML diagram of correlation data types in module DsLSRGenomicMaps
General Description

Contents

This chapter contains the following sections.

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Objects-by-value”</td>
<td>2-2</td>
</tr>
<tr>
<td>“Iterators”</td>
<td>2-2</td>
</tr>
<tr>
<td>“Controlled Vocabularies”</td>
<td>2-3</td>
</tr>
<tr>
<td>“Identifier Strings”</td>
<td>2-5</td>
</tr>
<tr>
<td>“Mappable”</td>
<td>2-6</td>
</tr>
<tr>
<td>“Mappable and Map”</td>
<td>2-7</td>
</tr>
<tr>
<td>“Mappables and Assignments”</td>
<td>2-7</td>
</tr>
<tr>
<td>“Nested Maps”</td>
<td>2-7</td>
</tr>
<tr>
<td>“Retrievals and Queries”</td>
<td>2-8</td>
</tr>
<tr>
<td>“Lifecycle Issues”</td>
<td>2-12</td>
</tr>
</tbody>
</table>

This chapter describes the principles that are used by many of the components in this specification, along with explanations of the design rationale. The more detailed descriptions provided in the next section, Modules and Interfaces, refer to those provided here. For a UML diagram of some of the data types in this document, the reader is referred to the figures located in Chapter 1. The full IDL specification can be found in Appendix A.
2.1 Objects-by-value

The CORBA 2.3a specification provides the concept `valuetype`, an IDL data type intermediate between `struct` and `interface`. They are part of the so-called Objects by Value (OBV) specification. Although the OBV standard is relatively new and not widely available yet, the current work uses `valuetypes`, as they offer definite advantages. In the context of this standard, the benefit of `valuetypes` over `interfaces` is scalability (a single round trip transfers the whole state of the object). The benefit of `valuetypes` over `structs` is their extendibility through inheritance.

In the interest of scalability, the contents of a map should preferably be local to the client. For this reason the most abundantly used types (`Mappable`, `Position`, and `Assignment`) are represented using `valuetypes`. They are used essentially as extendible `structs`, by applying the following constraints:

- all members (‘attributes’) are `public`,
- there are no methods,
- inheritance is only of other `valuetypes` (i.e., no “supports SomeInterface”),
- all inheritance uses `truncatable` (i.e., “casting” a sub-type to its super-type by simply omitting the extra members is a semantically valid operation).

2.2 Iterators

If a method has to return a multi-valued result to the caller, there is a design choice of returning these elements directly as a list, or through an iterator, or using a combination of both.

This standard mostly uses the iterator approach. Iterators are objects that ‘point to’ elements in a set, and which can be used to ‘step through’ the set. During this stepping process, each element is visited once. If the underlying set is ordered, this ordering is also preserved in the output of the iterator methods. If, during the iteration, the underlying result set changes (by another process), an exception is thrown. Iterators allow the client to choose between the scalability of iterators and the convenience of the lists returned by the iterators’ `next_n()` methods.

This standard has iterators for the data types `Map`, `Assignment`, and `VocabularyEntry`. An iterator provides a pointer or cursor to step through a set of entities. The iterators all look as follows:
exception IteratorInvalid {
    string reason;
};

interface ThingIterator {
    boolean next(out Thing the_thing) raises(IteratorInvalid);
    boolean next_n(in unsigned long how_many, out ThingList thing_list)
        raises(IteratorInvalid);
    void reset();
    void destroy();
};

Iteration using these objects can be in steps of one entry using the \texttt{next()} method, which are returned as the \texttt{out} parameter. Alternatively, when using the \texttt{next\_n()} method, a batch of at most \texttt{how\_many} entities are returned in the \texttt{out} parameter. If the retrieval was successful, the \texttt{out} parameter contains the next entity or entities. \texttt{TRUE} is returned if the call did not yet exhaust the iteration (i.e., if more elements are available for subsequent calls to \texttt{next()} or \texttt{next\_n()}). Conversely, a \texttt{FALSE} return value signifies that no more elements are available from the iterator. If, in a call to \texttt{next\_n()}, less than the requested \texttt{how\_many} elements can be returned, the \texttt{out} parameter contains as many elements as were available, and the return-value is \texttt{FALSE}. The \texttt{next()} and \texttt{next\_n()} methods can fail (e.g., if the underlying set changed). In this case, the \texttt{IteratorInvalid} exception is raised. Its \texttt{reason} member can be used to provide human-readable information on details of the failure.

Calls to \texttt{reset()} re-position the iterator such that subsequent calls to \texttt{next()} or \texttt{next\_n()} start at the beginning of the result set. In this case, nothing is implied about the contents of the underlying result set, or of their ordering; both may have changed.

Empty result sets (such as from queries yielding no matches) are not represented by NULL objects, but by real iterators that are ‘empty’ (i.e., invoking their \texttt{next()} or \texttt{next\_n()} methods only ever return \texttt{FALSE}).

The \texttt{destroy()} method is used to indicate that the iterator is no longer needed, and deletes the iterator object.

### 2.3 Controlled Vocabularies

When describing and representing domain-specific systems, there is frequently a need for a string type that can only assume a limited set of allowed values, a set however that is allowed to change over time (as values are added or removed) or space (different servers accepting different sets of strings). Such strings are called \textit{controlled vocabulary} strings (“vocabulary strings” for brevity). A particular set of such strings, valid in some context, is called a controlled vocabulary. Vocabulary strings typically denote domain-specific concepts, usually as a short descriptive string or common abbreviation, rather than as a code. An example from the mapping domain would be the strings “unknown,” “genetic,” “EST,” and “RFLP” as valid marker types in a particular map.
To specify a satisfactory standard for vocabulary strings, the usage of an `enum` is too inflexible, as it would require approval and re-compilation of new IDL, possibly rendering existing clients and/or servers incompatible. Conversely, `string` is too lenient, as there is no mechanism to list or control the values that a particular variable of such a type can assume. As a result, the definition of a system that needs vocabulary strings becomes less precise and less interoperable. This loss of semantics is frequently due to things as trivial as misspellings and issues of type-case and white-space usage.

Some of the controlled vocabulary functionality could be provided by the Lexicon Query Services standard (LQS; corbamed/98-03-22). For reasons described in Appendix B, the current specification includes the module `DsLSRControlledVocabularies`, which describes a standard for representing, listing and checking vocabulary strings. This module is general, and may be of use in contexts other than that of genomic maps. A mapping between `DsLSRControlledVocabularies` and LQS is also given in Appendix B.

To provide a standard way of offering access to the functionality LQS, a specialization of `VocabularyFinder` called `LQSVocabularyFinder` is provided in module `DsLSRLQSLink`. This optional interface has a `TerminologyServices::LexExplorer` attribute that yields access to the LQS functionality.

IDL strings are used to represent vocabulary strings. The `typedef string VocabularyString` is provided, and used to indicate that the values of an attribute or member are constrained. `VocabularyStrings` are contained in `VocabularyEntry`s (along with a `description`). There are no syntactic restrictions on the value of `VocabularyStrings`, but the following guidelines are suggested:

- vocabulary strings should not contain superfluous white-space (i.e., no leading or trailing white-space); internal white-space should be represented by single spaces only.

- vocabulary strings should be short yet descriptive. Common abbreviations often serve this purpose well.

- empty strings are allowed but discouraged. The semantics are typically “unknown,” “not applicable,” “missing,” “miscellaneous,” “default,” etc. It is considered cleaner to define dedicated vocabulary strings for this purpose.

The `VocabularyEntry`s are served by `Vocabulary` objects, which can in turn be obtained from `VocabularyFinder` objects. The vocabulary names presented by the `VocabularyFinder` are typically identical to the names available from the `Vocabulary` interface. However, this is not necessarily the case, because names provided by individual `Vocabulary` objects cannot be guaranteed to be unique.

The anticipated use of `VocabularyFinder` objects is to contain a relatively limited number of `Vocabulary`s (say, less than 50). If there is a need for managing larger numbers of `Vocabulary`s, a more general and powerful facility akin to a knowledge base would be more appropriate.
2.4 Identifier Strings

There is frequently a requirement for a simple data type to indicate an entity’s identity. In most cases, this need is or can be addressed by using a string type. The advantages are that it is simple, lightweight, and used universally throughout the realm of computing (and indeed outside). However the risk of using strings is that they can be too flexible, both in terms of syntax and semantics. This easily results in the lack of interoperability. To allow strings, yet mitigate their potential for abuse, this standard uses the syntax convention of `CosNaming::StringName` as described in the Interoperable Naming service. This convention is mainly a syntactical one; in no way is the use of a naming service implementation required or implied (but it is not precluded either).

A brief description of `CosNaming::StringName` is as follows. `CosNaming::Name` is a list of `struct NameComponent`. For the purpose of illustration, a `NameComponent` can be likened to a directory or filename, whereas `CosNaming::Name` constitutes a full path-name. The `struct NameComponent` has string members `id` and `kind`. To transform a `CosNaming::Name` into a string, all its `NameComponent`s are represented as strings “`id.kind`”. If the `kind`-field is empty, this becomes simply “`id`”; if the `id`-field is empty, this becomes “`.kind`”; finally, the Naming service also allows both the `id`- and `kind`-fields to be empty, which is represented as “`.`”. The full `stringified CosNaming::Name` is then obtained by concatenating all the `NameComponent`s using “`/`” as a separator character. The character “`\`” is designated as an escape character; if it precedes any of the special characters “`.`”, “`/`” and “`\`”, these special characters are taken as literal characters.

The typedef `string CosNaming::StringName` is provided for strings used as object names using this convention.

The genomic maps specification adopts the same syntax convention, but requests that the components of our `Identifier` data type adhere to some additional semantic constraints. These rules do not follow from, nor are implied by any semantics of the Naming Service. The additional constraints make this data type sufficiently different from `CosNaming::StringName` to warrant the dedicated typedef `string Identifier`.

In the remainder of this description, ‘component’ means: the sub-string of an `Identifier` that corresponds to one `CosNaming::NameComponent`; likewise, `id`-field and `kind`-field correspond to the equivalent fields of `NameComponent`.

The rules are as follows:

- Names can refer to collections of entities (such as databases), or to entities within such collections. Names referring to collections consist of exactly one component; names referring to entities within collections consist of at least two components.

- The first component represents the data source. Data sources can be anything: transient collections, local databases, public repositories, etc. It is up to the implementation to document the accepted names for the data source.

- The empty name (“`.`”) is valid for the first component, and represents the ‘local’ or ‘default’ collection. It is up to the implementation to document what the semantics of ‘local’ or ‘default’ is.
Names that refer to entities within collections consist of two or more components. The second component of such names represents an identifier that is unique in the context of the data source. No empty id-fields are allowed in this or any further components.

If two components are not enough to uniquely identify an entity, an Identifier can contain more than two components, but no more than necessary to make the identification unique. That is, an Identifier may not be used to freely attach textual information.

The only characters valid in a component are “a” through “z”, “0” through “9”, and “-” (hyphen), “_” (under_score), “$” and “.” (period). Use of the latter is discouraged since it has a special meaning in the stringifying convention, and has therefore to be escaped.

To comply with existing practice in the field of public data repositories, it is strongly advised that implementations do string comparisons in a case-insensitive manner. The CosNaming Service standard fails to mention whether type-case is, for string comparison purposes, significant or not. Implementations that use a third-party implementation of the Naming service may therefore wish to restrict Identifiers to only use one type-case. It is up to an implementation to state whether mixed type-case is allowed, and whether type-case is significant in comparisons.

The id and kind parts of the string components of Identifier are used as follows:

- The id-field of a component contains the principal value that makes it unique in the scope provided by the preceding component. It may only be empty in the case of the first component of an Identifier (see above).

- The kind-field of a component is used to represent information indicating the release (for a data source) or version (for an entry) of an entity, and can be empty. If kind is empty and entities with non-empty kind-fields exist, an empty kind field becomes synonymous with ‘the latest release or version’. It is up to the implementation to document the syntax and semantics of the version information.

The adoption of this convention has the following advantages:

- it is simple and lightweight,
- it has a well-defined and ‘re-used’ syntax,
- it is compatible with existing practice,
- it is sufficiently flexible to allow for sub-IDs if necessary.

The LSR Biomolecular Sequence Analysis standard uses the same Identifier type and semantics.

2.5 Mappable

The data type Mappable defined in this specification is used to represent things that can be mapped. This includes ordinary markers as well as sub-maps (see below). The choice for this unusual, novel term was motivated by the desire to reduce the risk of confusion with existing terms.
2.6 Mappable and Map

As indicated above, a Map should be considered a special kind of Mappable to allow for the nesting of maps. Moreover, it is desirable that map contents be local to clients. Yet at the same time, the data type Map needs methods to serve and query its contents, so Map should be an object local to the server.

This inheritance relationship is represented as a delegation: the interface Map has a readonly attribute Mappable the_mappable, which contains (and transfers) the whole state of the Mappable aspects of a Map. One could call this the ‘header information’ of a map (such as the id, the chromosome, etc.).

2.7 Mappables and Assignments

The relationship between maps and mapped entities is a many-to-many association. That is, one map can contain many markers, and conversely, one marker can have been mapped on several different maps. An assignment of one marker on one map is one instance of this map-marker relationship. Therefore, a map is a container (or factory) of Assignments, rather than of (references to) Mappables. For this reason, the query methods of Map and its sub-types yield assignments, never Mappables. If the Mappables contained in a Map are required, they can always be obtained by inspecting the mapped_entity members of the sub-types of Assignment.

Since an assignment can be both of a Mappable and of a sub-Map, the valuetype Assignment data type is never returned directly; only its more specific sub-types MappableAssignment and SubMapAssignment are returned. Only instances of these sub-types contain the mapped entity that makes them meaningful.

2.8 Nested Maps

Nested maps are a potentially powerful concept for the integration of mapping data from different (and potentially distributed) sources. Another area where nesting is desirable is the case where resolution of a map increases over the course of a mapping project. For example, a contig may be treated as a point-like entity at first; later, when the mapping effort proceeds, the contig may become a map in its own right.

There are three potential problem areas with nested maps: infinite recursion, representation and coordinate systems, and querying.

Infinite recursion occurs when a map contains itself as a sub-map. However this standard is concerned with representation only, and it is the implementor’s responsibility to prevent such errors. Hence, this does not pose a problem.

Coordinate systems of nested maps are generally different from that of the containing map. Such nested maps must retain their own ordering and coordinate system, as sub-maps may not be under the control of the implementation, and transforming the coordinates would be cumbersome. Perhaps more importantly, currently no widely accepted general coordinate system is available that would make such a transformation possible or meaningful. For the same reasons, maps and map sections that are returned from queries always retain the nesting structure of the underlying map.
The problem with queries into possibly nested maps is whether the query method is expected to satisfy the criteria by inspecting the immediately contained mapped entities only, or alternatively, should delve into any contained nested maps to find matches. A related issue is how a query method should return ‘nested hits’ (i.e., queries that actually are satisfied at a nested level).

The first question is addressed by having a parameter `recursion_depth` for query operations where this is relevant. If this parameter is zero, no descending into nested maps takes place, and only ‘top-level’ mapped elements are inspected for matches (although these top-level elements themselves may in fact be maps). If `recursion_depth` is greater than zero, the query descends into a nesting level no deeper than its value. The value of the `recursion_depth` parameter only determines how deep a query should descend; it does not determine the way in which the obtained match is returned.

This is the second issue, and it is resolved as follows: each query for mapped entities shall only return directly contained Assignments (whether this concerns entities of Map type or not). Entities that match at a deeper level (‘nested hits’) are not returned directly; instead, the Assignment that contains it (either directly or through yet further nesting) is returned. In other words, each query either returns the sought entity (the usual case) or the Assignment that contains it at some deeper nesting level.

In the latter case (the ‘nested hit’ case), only the first step in the access path to an entity contained in a nested map is given as a result. Therefore, in this situation, the same query has to be effected on the returned map, possibly several times, until the sought entity itself is obtained.

The rationale for this design is that it is sufficient, simple and unambiguous. Moreover in client implementations, explicit representations of the nested map ‘tree’ will have to be established before the map can be rendered anyway. Therefore, the usefulness of a method that returns the complete access path to the mapped entity in one call is debatable.

Finally, it can be remarked that iterators are well suited to dealing with recursive structures. By always invoking their `next(out_arg)` method on nodes in the tree structure, depth-first traversal of a nested map structure can be effected. Conversely, breadth-first traversal can be had by using the `next_n(n, out_args)` method with `n` being a very large number.

### 2.9 Retrievals and Queries

The current specification offers limited query capability by two means. Firstly, a distinction is made between retrieval and querying. Retrieval, in this context, includes both the resolving of a known entity given some designator such as name or an id, as well as the listing and/or obtaining of all entities in a given space. In contrast, querying corresponds to the semantically different (and richer) concept of searching among entities in a given space.
2.9.1 Retrievals

Retrievals are available as specific methods, typically get_thing_by_id() for the resolution case, and get_all_things() for the listing case. The resolution methods can raise the CannotResolveID exception:

```java
exception CannotResolveID {
    Identifier id;
    string reason;
};
```

This rationale for using an exception rather than returning nothing is that resolution should normally be considered to succeed.

2.9.2 Queries

The more general query functionality is provided by inheriting from CosQuery::QueryEvaluator. Its evaluate() method is used for the expression of a number of queries, and has the following signature:

```java
any evaluate(in string query, in QLTypeql_type, in ParameterList params) raises (QueryTypeInvalid, QueryInvalid, QueryProcessingError);
```

A number of const QueryString (with descriptive names such as GET_ASSIGNMENTS) are defined that represent fixed queries. To effect a query, one such predefined QueryString is passed as the query argument to evaluate(). The contents of the params argument contain the criteria for the query, and are described below. In the remainder of this document, the term “query” is often synonymous with “passing a particular pre-defined QueryString into the evaluate() method.”

The argument ql_type denotes the query language type. Its formal type is a CORBA::InterfaceDef corresponding to one of the (empty) sub-classes of the (empty) QueryLanguageType interface defined in CosQuery. All interfaces that extend CosQuery::QueryEvaluator must accept MapsQL. This query language type is defined as the CORBA::InterfaceDef of the MapsQueryLanguageType interface (the ‘value’ of this CORBA::InterfaceDef cannot be defined in IDL; hence this textual description). In contrast to the formal CosQuery module, the current specification does not require that at least one of the OQL or SQL query languages be supported.

The semantics of MapsQL are as follows. The params argument to evaluate() contains name-value pairs that correspond to criteria that have to be fulfilled. It is essentially a query in conjunctive normal form (albeit a very restricted version: just two levels of clauses are possible, and there is no logical NOT operator). That is, queries such as the following:

```
( (criterion1 = value1 OR criterion1 = value2 OR criterion1 = ...)
 AND
 (criterion2 = value3 OR criterion2 = value4 OR criterion2 = ...)
 AND
```

(criterion\textsubscript{3} = value\textsubscript{5} OR criterion\textsubscript{3} = value\textsubscript{6} OR criterion\textsubscript{3} = \ldots )
AND
\ldots

are represented by a list of \texttt{struct \{ string name; any value; \}} pairs with the following values:

\begin{verbatim}
{ { “criterion1” , {“value1”, “value2”, \ldots } },
 { “criterion2” , {“value3”, “value4”, \ldots } },
 { “criterion3” , {“value5”, “value6”, \ldots } },
 \ldots
}
\end{verbatim}

(The name-value pairs are defined as a \texttt{CosQueryCollection::ParameterList}). Each name-value pair represents one criterion; its value (packaged as an IDL \texttt{any}) is a list of terms that each would constitute a valid match for the criterion. That is, the ‘local’ match for one criterion is the logical OR of all the values in the list. The final result of a query is simply the logical AND of the local matches obtained for each of the separate criteria.

As a convenience, a criterion queried with a single particular value (rather than a list) may be represented as that value, rather than as a list of values of length one. Likewise, separate criteria having the same criterion name are treated as one criterion that matches any of the values of all the separate criteria. For example:

\begin{verbatim}
{ { “criterion1” , {“value1”, “value2” } },
 { “criterion1” , {“value3”} }
}
\end{verbatim}

is equivalent to

\begin{verbatim}
{ { “criterion1” , {“value1”, “value2”, “value3” } }
}
\end{verbatim}

Strings are always allowed as search values, but an implementation may offer querying on types other than \texttt{string}. In some cases, the required type of the query is indicated (e.g., \texttt{long} for parameter “recursion_depth”, \texttt{float} for “from,” “to,” and “range” for the queries of \texttt{LinearMap}; see below).

The values of match criteria typically can assume only a limited number of string values. That is, they are vocabulary strings. For this reason, each match criterion corresponds to a \texttt{Vocabulary}, with the name of the criterion being the \texttt{name} attribute of its \texttt{Vocabulary}. The different criteria and corresponding \texttt{Vocabulary}s that apply to a particular queriable object are available from its \texttt{VocabularyFinder}. An easy way to obtain the criterion names is to invoke its \texttt{get_all_vocabulary_names()} method. Typical criteria are “id,” “type,” “specie,” “chromosome,” “sex,” as these criteria correspond to the fixed members of the data types \texttt{Mappable} (and \texttt{Map}). The \texttt{name}-parts of the contents of the \texttt{properties} of \texttt{Mappable} are other likely candidates for match criteria.

Criteria that can assume an unlimited number of values (e.g., \texttt{length}, \texttt{recursion_depth}) are also represented by a \texttt{Vocabulary}, but this \texttt{Vocabulary} is degenerate, in that it contains no \texttt{VocabularyEntry}s.
The return type of `evaluate()` is `any`. In current specification, each of the queriable objects shall return only one particular type, which is documented below.

Some queries have mandatory parameters. Such parameters and their values must be passed when invoking `evaluate()`, and their semantics as documented below must be implemented. An example is “recursion_depth” for queries of `Map`. In the description of the queries below, the mandatory parameters are indicated. The current standard does not specify a general solution to dynamically inquire whether a parameter is mandatory. (A suggestion is to put this information in the `description` attribute of the `Vocabulary` that represents the parameter.)

The exceptions `QueryTypeInvalid`, `QueryInvalid`, and `QueryProcessingError` may be raised by `evaluate()`. `QueryTypeInvalid` is raised if the query language type is not understood. To be compliant with this standard, this is not allowed to happen when the `MapsQL` is used. `QueryInvalid` is raised whenever query parameter is not valid. The most important cases are:

- the value passed as in query string in a call to `evaluate()` is not one of the `const strings` defined in this specification, and is not recognized by the implementation.
- the list passed as in `ParameterList params` in a call to `evaluate()` uses wrong parameter names (i.e., they are not among the names of the vocabularies contained in the `vocabulary_finder`, and hence cannot be queried for.
- the list passed as in `ParameterList params` in a call to `evaluate()` does not include the mandatory parameters (e.g., `recursion_depth` in some queries).
- the list passed as in `ParameterList params` in a call to `evaluate()` uses wrong parameter value strings. For example, they are not among the vocabulary strings contained in the corresponding vocabulary (e.g., querying for sex = “red”).
- the list passed as in `ParameterList params` in a call to `evaluate()` uses wrong parameter value types (that is, if they are other than `string`).
- The `why` string of the `QueryInvalid` exception should document the details of the failure in human readable form; at least the above five cases must be distinguished.
- The `QueryProcessingError` is raised to signal a ‘run-time’ of the query execution.

The advantages of using `CosQuery` are re-use, and the ability to easily extend the range of query capabilities by providing additional query language types and/or additional predefined `QueryString`.

### 2.9.3 Wildcards

Implementations can, but are not required to offer the use of wildcards in queries. If wildcards are offered, they should follow the convention used for Posix filename wildcards (ISO/IEC 9945-2:1993):

- ‘?’ is taken as meta-character that represents any single character;
- ‘*’ is the meta-character that represents a string of any length (including 0), consisting of any characters;
2.9.4 Ordering

The ordering of map contents returned by a query is not strictly necessary, as the client could reconstruct it from the position information. However, it is obviously more natural and convenient for clients if the results are ordered. This specification requires that the Assignments in an AssignmentList (see below) be by increasing Assignment.positions[0].rank.

If an AssignmentList contains an Assignment having a compound position, the ordering of the AssignmentList is by definition not total, but it is unique, predictable, and repeatable.

The elements in the positions attribute of the Assignment data type are ordered by position.rank. The Positions (and their ranks) of one Assignment must be distinct.

As discussed under the description of multi-valued return types, the ordering of results does not depend on the way the results are retrieved.

2.10 Lifecycle Issues

A number of interfaces used in this specification have a method to delete the object. This can be through a destroy() method (the iterators), or by inheritance from CosLifeCycle::LifeCycleObject (Map and its sub-types). The interfaces using the latter approach (Map and Vocabulary) can, in addition to the inherited remove() method, provide the move() and copy() functionality. If they do not implement these methods, the standard system exception CORBA::NO_IMPLEMENT should be raised with an exception minor code of 7.
This chapter describes the types, methods as well as the semantics of the standard in detail. For an overview and a description of the design rationale, refer to Chapter 2. For brevity, not all the required forward declarations, typedefs and iterators are provided in the boxes containing IDL, as they will be clear from the context, and have been described earlier. For the full IDL specification, the reader is referred to Appendix A.

### 3.1 Module DsLSR ControlledVocabularies

Controlled vocabularies essentially represent ‘dynamic enums’. The need for and usage of controlled vocabularies is described in more detail in Section 2.3, “Controlled Vocabularies,” on page 2-3.

#### 3.1.1 Exceptions

**exception NotFound { string reason; };**

This exception is raised by Vocabulary::get_entry_by_name() and Vocabulary-Finder::get_vocabulary_by_name() if the desired entry or vocabulary could not be found.

**exception IteratorInvalid { string reason; };**

This exception is raised by VocabularyEntryIterator::next() and VocabularyEntryIterator::next_n() if the iterator has become invalid. For a description of the semantics, see Section 2.2, “Iterators,” on page 2-2.
3.1.2 Typedef VocabularyString

typedef string VocabularyString;

VocabularyString is the data type used for attributes that can only assume a limited set of string values. For a detailed description of the semantics of this data type, refer to Section 2.3, “Controlled Vocabularies,” on page 2-3.

3.1.3 Valuetype VocabularyEntry

valuetype VocabularyEntry {
    public VocabularyString vocabulary_string;
    public string description;
};

The contents of a controlled vocabulary are represented by the VocabularyEntry data type.

Members

vocabulary_string – an actually allowed value in a particular context

description – descriptive text; the anticipated use is to contain the full text of vocabulary_string if that string is an abbreviation.

3.1.4 Interface VocabularyEntryIterator

interface VocabularyEntryIterator {
    boolean next(out VocabularyEntry the_entry)
        raises(IteratorInvalid);
    boolean next_n(n(in unsigned long how_many, out VocabularyEntryList list)
        raises(IteratorInvalid);
    void reset();
    void destroy();
}; // interface VocabularyEntryIterator;

The semantics of the iterators were described in Section 2.2, “Iterators,” on page 2-2. The reset() operator can raise the CORBA::NO_IMPLEMENT exception (minor code 7) if the underlying implementation does not allow resetting the iterator.

3.1.5 Interface Vocabulary

interface Vocabulary: CosLifeCycle::LifeCycleObject {
    readonly attribute string name;
    readonly attribute string description;
    readonly attribute boolean case_sensitive;
    readonly attribute unsigned long num_entries;
3. VocabularyEntryIterator

VocabularyEntryIterator get_all_entries();
boolean is_contained(in string test_string);
VocabularyEntry get_entry_by_name(in string test_string)
    raises (NotFound);
};

Controlled vocabularies are represented by objects of the type Vocabulary.

The name attribute holds the name of the Vocabulary, and should be unique in the con-
text of the VocabularyFinder that serves it (see below). It is suggested that if a Vocabulary is
used to represent a query criterion, it should have the name of that criterion. The description attribute is provided to hold descriptive information. When Vocabulary is
used to represent a query criterion, this field could be used to indicate the semantics of the
criterion, such as its type and whether it is mandatory. The attribute case_sensitive is
used to indicate whether type case of VocabularyStrings in the VocabularyEntry of the
current Vocabulary is, for the purpose of comparison, significant or not; it is TRUE if
type case is significant, FALSE if not. Attribute num_entries contains the total number
of entries in a Vocabulary; they can be retrieved using the get_all_entries() method.

For establishing whether a given string belongs to a Vocabulary, the is_contained() method can be employed. The method get_entry_by_name() provides lookup func-
tionality, allowing the retrieval of single VocabularyEntry (e.g., to inspect their
description member).

3.1.6 Interface VocabularyFinder

Vocabularies are obtained from a VocabularyFinder object.

interface VocabularyFinder {
    readonly attribute string name;
    readonly attribute unsigned long num_vocabularies;
    StringList get_all_vocabulary_names();
    VocabularyList get_all_vocabularies();
    Vocabulary get_vocabulary_by_name(in string name);
};

The name attribute can be used to identify a VocabularyFinder; num_vocabularies is the number of all the Vocabularys served by it. Their names are available from the
get_all_vocabulary_names() method, and the Vocabularys themselves can be
obtained from the get_vocabulary_by_name() method.

On purpose, the iterator pattern is not used in this case, as the anticipated use of the
DsLSRControlledVocabularies module is to represent a relatively limited number
(say, less than 50) of controlled vocabularies. For such small numbers, iterators are not
needed.

Since objects of type VocabularyFinder are entry points into servers that provide
Vocabularys, it is likely that they will be registered with a CosTrader service. If they
are, the following Trader Service Type shall be used:

service omg.lsr.ControlledVocabularyFinder {
    interface DsLSRControlledVocabularies::VocabularyFinder;
mandatory property string provider;
mandatory property StringList vocabularies_served;
}

Likewise, if a VocabularyFinder is registered with a CosNamingService, this shall be done as follows (where the NamingContexts are separated by '/'):

/DSLSRControlledVocabularies/provider/VocabularyFinder

3.2 Module DsLSRLQSLink

This module offers connectivity to the Lexicon Query Service. It contains one interface, LQSVocabularyFinder, which is optional.

3.2.1 Interface LQSVocabularyFinder

interface LQSVocabularyFinder:
   DsLSRSControlledVocabularies::VocabularyFinder {
      readonly attribute TerminologyServices::LexExplorer lex_explorer;
   };

The optional LQSVocabularyFinder interface is a specialization of VocabularyFinder, and can be used instead of its super-type when it is desirable to offer access to the full functionality of the LexExplorer interface defined in the LQS TerminologyServices module. The attribute lex_explorer provides this link.

3.3 Module DsLSRGenomicMaps

3.3.1 Typedef Identifier

typedef string Identifier;

Identifier is used as data type for IDs. The syntax (which follows the new CosNaming standard) and semantics of the string Identifier are described in detail in Section 2.4, “Identifier Strings,” on page 2-5. The same type, syntax and semantics are used in the LSR Biomolecular Sequence Analysis standard.

3.3.2 Exception CannotResolveID

exception CannotResolveID {
   Identifier id;
   string reason;
};

This exception is thrown by methods that take an Identifier as an input parameter. If the entity denoted by the Identifier cannot be found or resolved by the method that takes it as an input argument, this constitutes an exceptional situation, as the usage of an Identifier
implies that the server can be expected to contain the desired entity. The id member holds the offending Identifier, and is provided as a convenience. The string reason holds details (if any can be provided) as to why the resolution failed. Suggested contents are: “syntax invalid” if the form of the Identifier to be resolved was not acceptable, and “entity unknown” if the syntax was right, but the entity it denotes could not be found.

3.3.3 Valuetype Mappable

```java
valuetype Mappable {
  public Identifier id;
  public IdentifierList cross_references;
  public StringList aliases;
  public VocabularyString type;
  public VocabularyString species;
  public VocabularyString chromosome;
  public VocabularyString sex;
  public CosPropertyService::Properties properties;
  public float length;
  public VocabularyString units;
};
```

Mappable is the central data type of this standard, and is used to represent map contents. It typically is a simple marker of some kind, but as explained in Section 2.6, “Mappable and Map,” on page 2-7, sub-Maps can also be regarded as Mappables. Mappable does not contain information on where it is located on a map, as one Mappable may be placed on several maps.

id is the unique identifier of the Mappable, and should comply with the convention described in Section 2.4, “Identifier Strings,” on page 2-5. The cross_references member can be used to hold references to other entities, of either the same or of a different type, from either the same or from a different data source. Elements in this list must comply with the conventions described in Section 2.4, “Identifier Strings,” on page 2-5. The list can be empty. It is anticipated that a common usage of the cross_references member is to contain Identifiers of entities of types specified in the BioMolecular Sequence Analysis standard, thus providing a point of contact between these two standards.

The member aliases can be used to provide other names for the current Mappable; this field is provided as a convenience. In contrast to the cross_references field, no constraints to syntax or semantics are imposed on the contents of this field.

The members type, chromosome, and sex contain information that characterizes the current Mappable. For each of these attributes the most specific value that applies should be used, although the empty string is also allowed. All these fields are vocabulary strings.

properties is a field that can be used to attach any additional characteristic not a fixed member of Mappable.

length and units describe the extent of the marker, if any. Markers that can be considered point-like have length = 0.0, and units is the empty string. Units that are integer-valued (such as base pairs) have an integer-valued length (e.g., 1243.00 if the integer length is 1243).
Mappables cannot be null. There are no sub-types defined for Mappable.

Queries for Mappables can be against the fixed members (type, chromosome, etc.) as well as against the additional characteristics contained as properties. Implementations should document which query criteria are provided.

3.3.4 Interface Map

interface Map: CosQuery::QueryEvaluator, CosLifeCycle::LifeCycleObject {
  const QueryString GET_ASSIGNMENTS = "get_assignments";

  readonly attribute Mappable the_mappable;
  readonly attribute VocabularyFinder vocabulary_finder;

  readonly attribute unsigned long num_assignments;
  readonly attribute boolean circular;

  Assignment get_assignment_by_mappable_id(  
    in Identifier the_mappable,  
    in unsigned long recursion_depth);
}

Map is the data type that represents a full or partial genomic map or anything that is used as such. Maps are also Mappables; this inheritance is specified as delegation for reasons explained in Section 2.8, “Nested Maps,” on page 2-7. All the Mappable aspects of Map, including information such as the ID or name, the species and the chromosome, are available from the the_mappable attribute. This attribute yields all the Mappable aspects of the Map in one round-trip, and can be regarded as containing its the header information.

A Map consists of Assignments; their number is provided in the num_assignments attribute. If a map is circular, the circular attribute is TRUE.

The method get_all_assignments() returns all the Assignments that constitute the map. The Assignments are returned by an iterator. For a description of the semantics of this approach see Section 2.2, “Iterators,” on page 2-2.

get_assignment_by_mappable_id() returns the Assignment that contains the Mappable identified by argument the_mappable. If possible and needed, the query descends into sub-maps to a recursion-depth of no more than recursion_depth to find the Mappable.

One query is provided, and represented by the GET_ASSIGNMENTS QueryString which can be passed as the in string query argument to the evaluate() method inherited from CosQuery::QueryEvaluator. The any returned by queries of MapFactory objects must be of type AssignmentIterator. Details of this mechanism are described in Section 2.9.2, “Queries,” on page 2-9.

The GET_ASSIGNMENTS query has three mandatory parameter: “start,” “end,” and “recursion_depth.” “start” and “end” are the Identifiers of Mappables that bracket the segment of the of the Map in which the Assignments are searched. An empty Identifier (i.e., the empty string) is legal, and implies the corresponding end-point of the map.
The “recursion_depth” parameter must be of type long or a string that evaluates to one.

In an invocation of the method, the actual start and end arguments may appear to be in the wrong order. This is the case if the Mappable designated by start is assigned closer to the end of the map than that denoted by argument end. In this situation, the arguments start and end are taken as if their values were exchanged (i.e., making their order reflect that of the underlying map). The reason for this silent correction is that it is simple and unambiguous, and preserves the ordering of the underlying map. For a description of the iterator mechanism, see Section 2.2, “Iterators,” on page 2-2; for a description of the “recursion_depth” parameter, see Section 2.8, “Nested Maps,” on page 2-7; for a description of the query specification, see Section 2.9.2, “Queries,” on page 2-9.

The assignments of Mappables denoted by the start and end arguments may be compound assignments (that is, if their Assignments have more than one Position). In this case, start and end are interpreted so as to return the maximum number of markers possible: the left-most of the Positions of start’s Assignment and/or the right-most Position of end’s Assignment are taken when calculating which Assignments to return.

The any returned by the evaluate() method must be of type AssignmentIterator. This also applies to the sub-types of Map, which are described below.

### 3.3.5 Interface OrderedMap

```
interface OrderedMap:Map {
    readonly attribute float LOD_score;
};
```

**OrderedMap** is a data type to represent maps for which only ordering information is known. The overall LOD-score (a measure of the quality of the map) is available in the attribute LOD_score.

### 3.3.6 Interface CytogeneticElement

```
interface CytogeneticElement;
typedef sequence <CytogeneticElement> CytogeneticElementList;
interface CytogeneticElement: Map {
    readonly attribute long rank;

    exception NoSuperBand { string reason; }

    CytogeneticElement get_super_band() raises (NoSuperBand);
    CytogeneticElementList get_sub_bands();
    CytogeneticElementList get_siblings();
};
```

Cytogenetic elements (chromosome banding patterns) are represented using the dedicated type **CytogeneticElement**. Theoretically, **Map**’s machinery for traversing and querying nested maps could be used to implement the functionality of cytogenetic maps, but common usage calls for the simpler methods provided by this interface. The exception NoSu-
perBand is raised if the traversal has reached the top of the hierarchy. The contents of its reason member are unspecified.

### 3.3.7 Interface LinearMap

```typescript
interface LinearMap: Map {
  const QueryString GET_INTERVAL = "get_interval";
  const QueryString GET_RANGE_AROUND = "get_range_around";

  readonly attribute float min_coordinate;
  readonly attribute float max_coordinate;
}
```

The **LinearMap** interface represents a fully metric map (i.e., one where the locations of all markers are expressed as distances, be they to the beginning of the map, or relative to other markers). It is an extension of Map that allows retrieval of Map sections specified by geometry.

Attributes **min_coordinate** and **max_coordinate** specify the end points of the map, with **min_coordinate < max_coordinate**.

Query by geometry is provided by the two QueryStrings **GET_INTERVAL** and **GET_RANGE_AROUND**.

The **GET_INTERVAL** query selects a geometric span of the **LinearMap**. To this end, two mandatory parameters are needed: “from” and “to,” which correspond to the beginning and end of the map section that is desired. The data type of these parameters can be **float** or a **string** that evaluates to one. If the “from”-parameter is less than **min_coordinate**, the beginning of the map is assumed; if the “to”-parameter is greater than **max_coordinate**, the end of the map is assumed. If “from” is greater than “to,” they are silently exchanged, for reasons outlined in the description of the **GET_ASSIGNMENTS** query of the super-type. Further criteria can be applied to the contents of the selected span by using additional parameters.

The **GET_RANGE_AROUND** query is similar to previous one, but bases its selection on the distance relative to a given marker. The distance is specified as the mandatory parameter “range” (which can be a **float** or a **string** that evaluates to one); the centre of this segment is specified as the mandatory parameter “mapped_entity.” The span from **mapped_entity - range** to **mapped_entity + range** is selected. If either end of this span runs off the map, the end point of the map in that part is assumed. Again, further criteria can be applied to the contents of the selected span by using additional parameters.

The assignment of the mapped entity denoted by the **mapped_entity** parameter may be compound (that is, if its **MappableAssignment** or **SubMapAssignment** has more than one **Position**). In this case, the location is to be interpreted such that the maximum number of markers possible is returned: the span runs from left-most of the **Positions - range** to right-most of the **Positions + range**.

Neither of the methods has the **recursion_depth** argument that determines recursion, as its usefulness is debatable, and the semantics are too difficult to specify.
3.3.8 Interface MapsQueryLanguageType

```java
interface MapsQueryLanguageType: CosQuery::QueryLanguageType {};
```

The query method `evaluate()` inherited from `CosQuery::QueryEvaluator` requires that a `CORBA::InterfaceDef` be passed into it as the `ql_type` argument. The `CORBA::InterfaceDef` of the above `MapsQueryLanguageType` can be used for this purpose. An implementation may offer more query languages, but to be compliant with the standard, at least `MapsQueryLanguageType` must be supported by all the interfaces that extend `CosQuery::QueryEvaluator`. The semantics of this `query_language_type` are described in detail in Section 2.9.2, “Queries,” on page 2-9.

3.3.9 Interface MapIterator

```java
interface MapIterator {
    boolean next(out Map the_Map)
    raises(IteratorInvalid);
    boolean next_n(in unsigned long how_many, out MapList map_list)
    raises(IteratorInvalid);
    void reset();
    void destroy();
}
```

This object is used to step through a set of `Maps`. It is the only valid return type to be contained in the `any` returned by the `evaluate()` method of `interface MapFactory`. The details of the semantics of iterators were described in Section 2.2, “Iterators,” on page 2-2.

3.3.10 Interface MapFactory

```java
interface MapFactory: CosQuery::QueryEvaluator {
    const QueryString MAP_BY_MAP_PROPERTY = "map_by_map_property";
    const QueryString MAP_BY_CONTENT_PROPERTY = "map_by_content_property";

    readonly attribute unsigned long num_maps;
    readonly attribute VocabularyFinder vocabulary_finder;

    MapIterator get_all_maps();
    Map get_map_by_id(in Identifier id) raises(CannotResolveID);
};
```

The data type `MapFactory` allows the retrieval of `Maps`. Queries are represented by the `QueryStrings` `MAP_BY_MAP_PROPERTY` and `MAP_BY_CONTENT_PROPERTY`. These strings should be used as the `in string query` argument to the `evaluate()` method inherited from `CosQuery::QueryEvaluator`. The `any` returned by queries of `MapFactory` objects must be of type `MapIterator`. Details of this mechanism are described in Section 2.9.2, “Queries,” on page 2-9.
**MAP_BY_MAPPROPERTY** queries for maps based on their properties (that is, those of the Map ‘header,’ rather than those of the contained Mappables). Both ‘top-level’ Maps and sub-Maps can be returned, and there is no need for a “recursion_depth” parameter to this query (see Section 2.8, “Nested Maps,” on page 2-7).

**MAP_BY_CONTENT_PROPERTY** yields Maps for which the contained Mappables satisfy the query criteria. This query has “recursion_depth” (of type long or as a string that evaluates to one) as a mandatory parameter that determines how deep the recursion can be. This topic is discussed in Section 2.8, “Nested Maps,” on page 2-7.

The attribute num_maps contains the number of Maps that are available from the get_all_maps() method. get_map_by_id() is a retrieval method to fetch a known map from a server.

The vocabulary_finder attribute contains the VocabularyFinder that holds the Vocabularys corresponding to the search criteria.

Since objects of type MapFactory are entry points into servers that provide Maps, it is likely that they will be registered with a CosTrader service. If they are, the following Service Type shall be used:

```java
service org.irs.MapFactory {
    mandatory property string provider;
    mandatory property StringList map_databases_served;
}
```

Likewise, if a MapFactory is registered with a CosNamingService, this shall be done as follows (where the NamingContexts are separated by '/'):

```
/DsLSRGenomicMaps/provider/MapFactory
```

### 3.3.11 Valuetypes Assignment, MappableAssignment and SubMapAssignment

```java
enum AssignType { SINGLE, NOT, ALL, ONE, SOME, NONE };

value type Assignment {
    public boolean framework_assignment;
    public VocabularyString evidence;
    public PositionList positions;
    public AssignType assign_type;
};

value type SubMapAssignment : Assignment {
    public Map mapped_entity;
};

value type MappableAssignment : Assignment {
    public Mappable mapped_entity;
};
```

As discussed above, an assignment is an instance of the many-to-many association
between maps and the mapped entities, and holds the positional information that is the objective of mapping in general (see also Figure 1-1 on page 1-3). Assignments must always be returned as either a MappableAssignment or as a SubMapAssignment. Only these sub-types have the mapped_entity member (of different type) that make them meaningful. The mapped_entity member refers to the Mappable or sub-Map respectively, that is described by the Assignment. The positions field describes where it has been mapped. In the case of compound assignments, this is at more than one location (see below). A Map cannot have two different Assignments for the same Mappable or sub-Map: in this situation, a single Assignment with multiple Positions should be used.

assign_type is used to describe the following information. Assignments may be compound (e.g., if experimental information is ambiguous, or genes are detected in multiple copies). An assignment may also be a negative one, in the sense that a marker is known not to be at a certain location. These semantics can be expressed using the assign_type member:

**value of assign_type member**

- **SINGLE**
  - the Mappable or sub-Map is at the single position given
- **NOT**
  - the Mappable or sub-Map is not at the single position given;
- **ALL**
  - the Mappable or sub-Map is at all of the several positions given;
- **ONE**
  - the Mappable or sub-Map is at one, unknown, of the several positions given;
- **SOME**
  - the Mappable or sub-Map is at more than one, unknown, of the several positions given;
- **NONE**
  - the Mappable or sub-Map is at none of the several positions given.

SINGLE is probably the most commonly used value of this enum. The values SINGLE and NOT can only apply to single Positions. The usage of the types ALL, ONE, SOME, and NONE only apply if there is more than one Position.

The framework_assignment field of valuetype Assignment indicates whether the assignment was of a framework marker or not.

An Assignment cannot be null. The positions member of Assignment must contain at least one Position. The list positions may not contain duplicates. Their ordering is by increasing positions[0].

### 3.3.12 Interface AssignmentIterator

```java
interface AssignmentIterator {
    boolean next(out Assignment the_assignment)
    raises(IteratorInvalid);
    boolean next_n(in unsigned long how_many,
        out AssignmentList assignment_list)
    raises(IteratorInvalid);
    void reset();
}
```
Objects of this type are used to step through a list of Assignment. It is the only valid return type to be contained in the any returned by the evaluate() method of interface Map and its sub-types. The semantics of the iterator mechanism are described in more detail in Section 2.2, “Iterators,” on page 2-2.

3.3.13 Valuetype Position

valuetype Position {
    public long rank;
    public float LOD_score;
};

Position is the base-type of a family of types that hold the location information of an Assignment. In general, the positional information of an assignment includes or implies a point(s) of reference, units, and a measure of the quality of the assignment. These factors and their usage vary widely across different types of maps. Position has two members: rank and LOD_score. rank represents the most elementary position information: the index of an entity in an ordered list (ties are allowed). Ranks have usually a significance measure attached in the form of a LOD score; this is the role of the LOD_score member.

Neither Position, nor any of its sub-types is allowed to be null.

The unextended type Position is likely (but not required) to be used in Assignment of OrderedMaps.

valuetype MetricPosition: truncatable Position {
    public float left_end;
    public float right_end;
};

MetricPosition specializes Position for situations where the real distance to the beginning of the map is known. This distance is contained in the members left_end and right_end. If the mapped entity is segment-like, left_end and right_end denote the location of the entity’s end-points as a distance to the beginning of the map.

If left_end is greater than right_end, the segment is placed on the map in reversed direction.

If the mapped entity is considered to be point-like and the error associated with the placement can be represented as a distance, then left_end and right_end represent the endpoints of the interval in which the Mappable is believed to lie.

If a Mappable or sub-Map is considered point-like and the error of the placement is unknown, negligible, or cannot be represented as a distance, then left_end and right_end have identical values, again being the distance to the beginning of the map.

MetricPositions are likely to be useful in Assignments of LinearMaps.
3.3.14 Valuetype RelativePosition

valuetype RelativePosition: truncatable Position {
    public any left_flanking_entity;
    public any right_flanking_entity;
};

RelativePosition represents location information that is relative to (an)other Mappable(s) or sub-Map(s). The field left_flanking_entity is the point of reference to the left of the mapped entity, and right_flanking_entity is that to the right. Either but not both of these members can be null, in case there is just one flanking entity. Only Mappable or Map are valid types for the any.

If the mapped entity is considered segment-like, and left_flanking_entity lies, on the current map, to the right of right_flanking_entity, the placement of the mapped entity on the current map is in reverse direction.

RelativePosition offers no location information more precise than indicating the flanking entities; for this purpose, the type RelativeMetricPosition can be used.

3.3.15 Valuetype RelativeMetricPosition

valuetype RelativeMetricPointPosition: truncatable RelativePosition {
    public float distance_left;
    public float distance_right;
};

RelativeMetricPosition is used to represent location information that is relative to (an)other Mappable(s) or sub-Map(s), but where also a real distance to the flanking entities is known. distance_left is the distance to the left_flanking_entity; distance_right that to the right_flanking_entity. If either of the flanking entities is null, the corresponding distance is undefined.

No data-type or convention is provided to deal with the exceptional case of an entity lying to one side of both flanking entities.

3.3.16 Interface MapCorrelationFactory

interface MapCorrelationFactory: CosQuery::QueryEvaluator {
    const QueryString GET_CORRELATION = "get_correlation";
    const QueryString GET_ALL_CORRELATIONS = "get_all_correlations";

    readonly attribute unsigned long num_correlations;
    readonly attribute VocabularyFinder vocabulary_finder;
};

The data type MapCorrelationFactory provides the methods to obtain cross-correlations of maps.

As with the other factories described in this document, the queries are represented as fixed
pre-defined query strings which are passed as the \texttt{in string query} argument to the \texttt{evaluate()} method inherited from \texttt{CosQuery::QueryEvaluator}. Details of this are described in Section 2.9.2, “Queries,” on page 2-9. The \texttt{MapCorrelationFactory} interface has two such queries: \texttt{GET\_CORRELATION} and \texttt{GET\_ALL\_CORRELATIONS}.

The \texttt{GET\_CORRELATION} query has the mandatory input parameter “map,” which is an \texttt{Identifier} string. It returns all the correlations known for the map designated by the given identifier. The \texttt{GET\_ALL\_CORRELATIONS} query has the mandatory input parameters “map1” and “map2,” both \texttt{Identifier} strings. This query returns the correlations known between the two maps denoted by the identifiers given. As with the other query methods, these queries may take additional query criteria using the parameter mechanism described in Section 2.9.2, “Queries,” on page 2-9.

Only entities of type \texttt{MapCorrelationList} are valid as the type of the \texttt{any} returned by the \texttt{evaluate()}. \texttt{MapCorrelationFactory} objects are likely to be registered with a CosTrader service. If they are, they shall do so with the following Service Type:

\begin{verbatim}
service omg.isr.MapCorrelationFactory {
    interface DsLSRGenomicMaps::MapCorrelationFactory;
    mandatory property string provider;
};
\end{verbatim}

Likewise, if a \texttt{MapCorrelationFactory} is registered with a CosNamingService, it shall be done as follows (where the NamingContexts are separated by '/'):

\begin{verbatim}
/DsLSRGenomicMaps/provider/MapCorrelationFactory
\end{verbatim}

### 3.3.17 Typedef AssignmentPair

\begin{verbatim}
typedef sequence<AssignmentPair> AssignmentPairList;
typedef Assignment AssignmentPair[2];
\end{verbatim}

An \texttt{AssignmentPair} represents one correspondence between assignments on two maps.

### 3.3.18 Interface AssignmentPairIterator

\begin{verbatim}
interface AssignmentPairIterator {
    boolean next(out AssignmentPair the_assignment_pair)
    raises(IteratorInvalid);
    boolean next_n(in unsigned long how_many,
                   out AssignmentPairList assignment_list)
    raises(IteratorInvalid);
    void reset();
    void destroy();
};
\end{verbatim}

This iterator is used to step through a set of \texttt{AssignmentPairs}. The semantics of the iterator is described in detail in Section 2.2, “Iterators,” on page 2-2.
3.3.19 **Typedef MapPair**

```c
typedef Map MapPair[2];
```

When maps are correlated, the current standard represents this using pairs of maps. This data type is defined for that purpose.

3.3.20 **Interface MapCorrelation**

The data types used to represent correlations between two maps were depicted in Figure 1-3 on page 1-7.

```c
interface MapCorrelation {
    readonly attribute Identifier id;
    readonly attribute MapPair map_pair;
    readonly attribute AssignmentPairIterator correspondences;
};
```

`MapCorrelation` is a data type that contains all the information of a map cross-correlation. Member `id` provides an identification tag. `map_pair[0]` and `map_pair[1]` contain the two maps that are cross-correlated. A correspondence between an `Assignment` on `map_pair[0]` and one on `map_pair[1]` forms an `AssignmentPair`, with each first assignment of the pair being on `map_pair[0]` and each second one on `map_pair[1]`. The full list of correspondences is available from the `correspondences` attribute, which is an iterator.

Nothing is implied about the identity of `map_pair[0]` and `map_pair[1]`; they could even be the same map. The `correspondences` list is sorted by `positions[0]` of the first `Assignment` of each `AssignmentPair`. 
A.1 File: DsLSRControlledVocabularies.idl

```
//File: DsLSRControlledVocabularies.idl
#ifndef _DS_LSR_CONTROLLED_VOCABULARIES_IDL_
#define _DS_LSR_CONTROLLED_VOCABULARIES_IDL_

#pragma prefix "omg.org"
#include <CosLifeCycle.idl>

module DsLSRControlledVocabularies {
  // typedefs:
  typedef sequence <string> StringList;
  typedef string Identifier;
  typedef string VocabularyString;
  typedef sequence<string> VocabularyStringList;

  valuetype VocabularyEntry {
    public VocabularyString vocabulary_string;
    public string description;
  };
  typedef sequence<VocabularyEntry> VocabularyEntryList;

  exception IteratorInvalid { string reason; };

  interface VocabularyEntryIterator {
    boolean next(out VocabularyEntry the_entry)
    raises(IteratorInvalid);
    boolean next_n(in unsigned long how_many, out VocabularyEntryList list)
    raises(IteratorInvalid);
    void reset();
    void destroy();
  } // interface VocabularyEntryIterator;
```
interface Vocabulary: CosLifeCycle::LifeCycleObject {
    readonly attribute string name;
    readonly attribute string description;
    readonly attribute unsigned long num_entries;

    VocabularyEntryIterator get_all_entries();
    boolean is_contained(in string test_string);
}; // interface Vocabulary;
typedef sequence<Vocabulary> VocabularyList;

interface VocabularyFinder {
    readonly attribute string name;
    readonly attribute unsigned long num_vocubularies;

    StringList get_all_vocabulary_names();
    VocabularyList get_all_vocubularies();
    Vocabulary get_vocabulary_by_name(in string name);
    void destroy();
}; // interface VocabularyFinder
}; // module DsLSRControlledVocabularies
#endif // #ifdef _DS_LSR_CONTROLLED_VOCABULARIES_IDL_

A.2 File: DsLSRLQSLink.idl

//File: DsLSRLQSLink.idl
#ifndef _DS_LSR_LQS_LINK_IDL_
#define _DS_LSR_LQS_LINK_IDL_

#pragma prefix "omg.org"

#include "TerminologyService.idl"
#include "DsLSRControlledVocabularies.idl"

module DsLSRLQSLink {
    interface LQSVocabularyFinder:
    DsLSRControlledVocabularies::VocabularyFinder {
        readonly attribute TerminologyServices::LexExplorer lex_explorer;
    };
};

#endif // _DS_LSR_LQS_LINK_IDL_

A.3 File: DsLSRsGenomicMaps.idl

//File: DsLSRGenomicMaps.idl
#ifndef _DS_LSR_GENOMIC_MAPS_IDL_
#define _DS_LSR_GENOMIC_MAPS_IDL_


#pragma prefix "omg.org"

#include <CosLifeCycle.idl>
#include <CosPropertyService.idl>
#include <CosQuery.idl>

#include "DsLSRControlledVocabularies.idl"

module DsLSRGenomicMaps {
    // simple typedefs:
    typedef sequence<string> StringList;
    typedef string QueryString;
    typedef string Identifier;
    typedef sequence<Identifier> IdentifierList;

    // shorthands for imported types:
    typedef CosPropertyService::Properties Properties;
    typedef DsLSRControlledVocabularies::VocabularyFinder VocabularyFinder;
    typedef DsLSRControlledVocabularies::VocabularyString VocabularyString;
    typedef sequence<string> VocabularyStringList;

    // forward declarations:
    valuetype Assignment;
    interface AssignmentIterator;
    typedef sequence<Assignment> AssignmentList;

    interface AssignmentPairIterator;

    valuetype Position;
    typedef sequence<Position> PositionList;

    interface MapFactory;
    interface Map;
    interface MapIterator;
    typedef sequence<Map> MapList;

    exception IteratorInvalid { string reason; }; 

    exception CannotResolveID { Identifier id; string reason; }; 

    interface MapsQueryLanguageType : CosQuery::QueryLanguageType {};

    valuetype Mappable {
        public Identifier id;
        public StringList aliases;
        public IdentifierList cross_references;
        public VocabularyString type;
        public VocabularyString species;
        public VocabularyString chromosome;
    }
}
public VocabularyString sex;
public Properties properties;

public float length;
public VocabularyString units;
}; // interface Mappable

interface MapFactory: CosQuery::QueryEvaluator {
    const QueryString MAP_BY_MAPPROPERTY = "map_by_map_property";
    const QueryString MAP_BY_CONTENTPROPERTY = "map_by_content_property";

    readonly attribute unsigned long num_maps;
    readonly attribute VocabularyFinder vocabulary_finder;

    MapIterator get_all_maps();
    Map get_map_by_id(in Identifier id) raises(CannotResolveID);
}; // interface MapFactory

interface Map: CosQuery::QueryEvaluator, CosLifeCycle::LifeCycleObject {
    const QueryString GET_ASSIGNMENTS = "get_assignments";

    readonly attribute Mappable the_mappable;
    readonly attribute VocabularyFinder vocabulary_finder;

    readonly attribute unsigned long num_assignments;
    readonly attribute boolean circular;

    Assignment get_assignment_by_mappable_id (in Identifier the_mappable,
        in unsigned long recursion_depth)
        raises(CannotResolveID);
    AssignmentIterator get_all_assignments();
}; // interface Map

interface MapIterator {
    boolean next(out Map the_Map)
        raises(IteratorInvalid);
    boolean next_n(in unsigned long how_many, out MapList map_list)
        raises(IteratorInvalid);
    void reset();
    void destroy();
}; // interface MapIterator

interface OrderedMap: Map {
    readonly attribute float LOD_score;
};

interface CytogeneticElement;
typedef sequence <CytogeneticElement> CytogeneticElementList;
interface CytogeneticElement: Map {
    exception NoSuperBand { string reason; }
    readonly attribute long rank;

    CytogeneticElement get_super_band() raises (NoSuperBand);
    CytogeneticElementList get_sub_bands();
    CytogeneticElementList get_siblings();
}; // interface CytogeneticElement

interface LinearMap:Map {
    const QueryString GET_INTERVAL = "get_interval";
    const QueryString GET_RANGE_AROUND = "get_range_around";

    readonly attribute float min_coordinate;
    readonly attribute float max_coordinate;
}; // interface LinearMap

denum AssignType { SINGLE, NOT, ALL, ONE, SOME, NONE };

valuetype Assignment {
    public boolean framework_assignment;
    public VocabularyString evidence;
    public PositionList positions;
    public AssignType assign_type;
}; // valuetype Assignment

interface AssignmentIterator {
    boolean next(out Assignment the_assignment)
    raises(IteratorInvalid);
    boolean next_n(in unsigned long how_many,
        out AssignmentList assignment_list)
    raises(IteratorInvalid);
    void reset();
    void destroy();
}; // interface AssignmentIterator

valuetype SubMapAssignment : Assignment {
    public Map mapped_entity;
};

valuetype MappableAssignment : Assignment {
    public Mappable mapped_entity;
};

valuetype Position {
    public long rank;
    public float LOD_score;
};

valuetype MetricPosition: truncatable Position {
public float left_end;
public float right_end;
};

valuetype RelativePosition: truncatable Position {
    public any left_flanking_entity;
    public any right_flanking_entity;
};

valuetype RelativeMetricPointPosition: truncatable RelativePosition {
    public float distance_left;
    public float distance_right;
};

typedef Assignment AssignmentPair[2];
typedef sequence<AssignmentPair> AssignmentPairList;
typedef Map MapPair[2];

interface AssignmentPairIterator {
    boolean next(out AssignmentPair the_assignment_pair)
        raises(IteratorInvalid);
    boolean next_n(in unsigned long how_many,
        out AssignmentPairList assignment_list)
        raises(IteratorInvalid);
    void reset();
    void destroy();
}; // interface AssignmentPairIterator

interface MapCorrelation {
    readonly attribute Identifier id;
    readonly attribute MapPair map_pair;
    readonly attribute AssignmentPairIterator correspondences;
    readonly attribute unsigned long num_correspondences;
}; // interface MapCorrelation

typedef sequence<MapCorrelation> MapCorrelationList;

typedef MappingRelationFactory: CosQuery::QueryEvaluator {
    const QueryString GET_CORRELATION = "get_correlation";
    const QueryString GET_ALL_CORRELATIONS = "get_all_correlations";

    readonly attribute unsigned long num_correlations;
    readonly attribute VocabularyFinder vocabulary_finder;
}; // interface CorrelationFactory
}; // module DsGenomicMaps

#endif // #ifdef _DS_LSR_GENOMIC_MAPS_IDL_
**Relation to Lexicon Query Service**

The CORBAmed Lexicon Query Service is an OMG standard for representing medical terminology systems in a comprehensive framework. This includes such things as naming authorities, presentation (formats, language), conversion between different coding schemes, general description of relationships between concepts (including hierarchies), and different versions of coding schemes and value domains. None of these are deemed relevant for the domain of genomic maps. The **DsLSRControlledVocabularies** module of the Genomic Maps specification essentially offers a ‘dynamic enum’, and parts of the **ValueDomain** aspects of LQS could be used to address some of these needs.

This appendix describes a mapping between the types in the **DsLSRControlledVocabularies** module of the current standard, and the **ValueDomain** aspects of the Lexicon Query Service (LQS). This mapping may prove useful if implementors want to base their implementation of **DsLSRControlledVocabularies** on an implementation of LQS.

The list below follows the order of definitions given in the **DsLSRControlledVocabularies.idl** file, which can be found in Section A.1, “File: DsLSRControlledVocabularies.idl,” on page A-17. For each item, the **DsLSRControlledVocabularies.idl** type is given first, the LQS equivalent second. All the relevant LQS types are in the **TerminologyServices** module; therefore the types are not scoped by their module name. Attributes and methods are scoped by their class name using dot-notation where necessary.

- **VocabularyString** corresponds to a **QualifiedCode**, but with a human-readable **ConceptCode** and an empty **CodingSchemeID**. The functionality of the latter is not needed, as it is implied by the context.

- **valuetype VocabularyEntry** corresponds to **PickListEntry**. Inside this aggregate type, the member **vocabulary_string** corresponds to the **aQualifiedCode** member, whereas **description** corresponds to **pick_text**.

- **VocabularyEntryIterator** corresponds to **PickListIter**.
• **interface Vocabulary** corresponds roughly to **ValueDomainId**. The latter is `typedef`-ed to **struct QualifiedCode**. That is, **ValueDomain** itself is not a CORBA object, but is represented by an ID. Its methods can be found in the **LexExplorer** interface: the **Vocabulary** methods correspond to methods in **LexExplorer** that have **ValueDomainId** input arguments (see below). They all may raise the **UnknownValueDomain** exception.

• **Vocabulary.name** corresponds to **ValueDomainId**. That is, **ValueDomainId** corresponds to both a **Vocabulary** object as well as to its own (human-readable) name.

• **Vocabulary.description** is not represented in LQS; this attribute is for convenience only, and can be left empty.

• **Vocabulary.get_all_entries()** corresponds to **get_pick_list(in ValueDomainId value_domain_id, ...)** in the **LexExplorer** interface. This methods returns a **PickListIter**; the **Vocabulary.num_entries** attribute corresponds to the quantity obtained from **PickListIter.max_left()** when invoked appropriately.

• **Vocabulary.is_contained(in string test_string)** corresponds to **LexExplorer.is_concept_in_value_domain(in QualifiedCode qualified_code, in ValueDomainId value_domain_id)**.

• **interface VocabularyFinder** corresponds to **LexExplorer**.

• **VocabularyFinder.name** corresponds to **LexExplorer.terminology_service_name**.

• **get_all_vocabulary_names()** and **get_all_vocabularies()** in **interface VocabularyFinder** correspond to **LexExplorer.list_value_domain_ids()**. This method returns a **ValueDomainIdIter**, which is an iterator. The **VocabularyFinder.num_vocabulary** attribute corresponds to the quantity obtained from **ValueDomainIdIter.max_left()** when invoked appropriately.

• **VocabularyFinder.get_vocabulary_by_name()** is not represented in LQS, since **ValueDomains** are not CORBA objects, but are represented by a **ValueDomainId**. Instantiating a **ValueDomainId** would require a **ValueDomainId** as input argument, which obviates the need for this method in LQS.
## Glossary

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Data type to represent assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>Placement of a Mappable or sub-Map on a Map; contains position information. Can be compound. See also MappableAssignment and SubMapAssignment.</td>
</tr>
<tr>
<td>bin</td>
<td>One of an ordered set of collections of unordered markers.</td>
</tr>
<tr>
<td>clone</td>
<td>In the context of large-scale sequencing: long sequence used in genome sequencing. Clones or sub-clones are assembled into contigs.</td>
</tr>
<tr>
<td>compound assignment</td>
<td>A non-unique placement of a Mappable on a Map.</td>
</tr>
<tr>
<td>contig</td>
<td>A genomic sequence fragment assembled from an overlapping group of clones.</td>
</tr>
<tr>
<td>controlled vocabulary</td>
<td>A set of strings that are valid as the values of a vocabulary string. The standard specifies a Vocabulary data type that represents such sets.</td>
</tr>
<tr>
<td>cytogenetic map</td>
<td>Map (or image) of chromosome banding patterns. See also idiogram.</td>
</tr>
<tr>
<td>EST</td>
<td>Expressed Sequence Tag.</td>
</tr>
<tr>
<td>factory</td>
<td>An object that is capable of ‘producing’ other objects (simply by returning them as a result of a method call). These objects may or may not be entirely new and/or shared with others.</td>
</tr>
<tr>
<td>framework</td>
<td>In the context of mapping: a map consisting of well known and high-quality ‘anchor points’ (framework markers), relative to which other markers are placed.</td>
</tr>
<tr>
<td>gene</td>
<td>Unit of inheritance; also: the DNA sequence coding for a particular protein sequence. Also: unit of independently regulated transcription. No definition is generally accepted and the issue is somewhat contentious.</td>
</tr>
<tr>
<td>genome</td>
<td>The full volume of information contained in the genetic material of a species.</td>
</tr>
<tr>
<td>genomic</td>
<td>Belonging/applying to the genome as a whole.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>genomic map</td>
<td>Map of chromosome content obtained by any means. This is as opposed to genetic map, which is generally used for maps obtained from linkage analysis. The term map is used more frequently in the domain of molecular genetics, but is too general.</td>
</tr>
<tr>
<td>genomic sequence</td>
<td>Sequence such as existing in the chromosomes themselves.</td>
</tr>
<tr>
<td>idio gram</td>
<td>Simplified drawing of a chromosome that highlights certain aspects such as banding patterns. See also cytogenetic map.</td>
</tr>
<tr>
<td>linkage analysis</td>
<td>The calculation of maps based on the observed patterns of occurrences of traits in families of individuals.</td>
</tr>
<tr>
<td>locus</td>
<td>A location on the chromosome (as opposed to the contents of such a location, such as a gene). In the current standard, these entities are best represented as Mappables.</td>
</tr>
<tr>
<td>LOD score</td>
<td>Logarithm of odds score; statistical measure of the quality of a placement on a map. The higher, the better.</td>
</tr>
<tr>
<td>LQS</td>
<td>Lexicon Query Service; an OMG CORBA standard (formal/99-03-01) that could be used to deal with representations of controlled vocabularies.</td>
</tr>
<tr>
<td>Map</td>
<td>Data type that represents maps.</td>
</tr>
<tr>
<td>map</td>
<td>A summary of chromosome content. The ultimate map is the full sequence of a chromosome (in which case ‘summary’ is a misnomer). See also genomic map.</td>
</tr>
<tr>
<td>Mappable</td>
<td>Term used in this standard to represent anything that can be placed on a map. This includes maps themselves, in the case of nested, or sub-maps.</td>
</tr>
<tr>
<td>MappableAssignment</td>
<td>An assignment of a simple Mappable (as opposed to a sub-Map).</td>
</tr>
<tr>
<td>marker</td>
<td>Any experimentally identifiable element on a chromosome. Examples include genes, ESTs, polymorphisms.</td>
</tr>
<tr>
<td>nested map</td>
<td>A map placed, at a certain location, within another map. Same as sub-Map.</td>
</tr>
<tr>
<td>OBV</td>
<td>Objects-by-value; see valuetype.</td>
</tr>
<tr>
<td>PIDS</td>
<td>Person Identification Service; an OMG CORBA standard (formal/99-03-05) for uniquely identifying persons.</td>
</tr>
<tr>
<td>placement</td>
<td>See assignment.</td>
</tr>
<tr>
<td>polymorphism</td>
<td>Any variation in chromosome content that can be used to distinguish between individuals; used in linkage analysis.</td>
</tr>
<tr>
<td>ordered</td>
<td>Having an ordering; in the context of molecular genetics, indicates that only the order is known, rather than more precise distances.</td>
</tr>
<tr>
<td>sequence</td>
<td>Biologically, a string of nucleotides (DNA building blocks) or amino acids (protein building blocks. Often the term sequence is used as including additional information.</td>
</tr>
<tr>
<td>STS</td>
<td>Sequence Tagged Site. An example is EST.</td>
</tr>
<tr>
<td>sub-Map</td>
<td>A Map that is contained in another Map. Same as nested map.</td>
</tr>
<tr>
<td><strong>SubMapAssignment</strong></td>
<td>An <strong>assignment</strong> of a <strong>sub-Map</strong> inside another <strong>Map</strong></td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td><strong>valuertype</strong></td>
<td>IDL keyword from the Objects-by-Value specification, designating an entity that lies halfway between an IDL <strong>struct</strong> and an IDL <strong>interface</strong>.</td>
</tr>
<tr>
<td><strong>vocabulary string</strong></td>
<td>A string that can only assume a limited set of values; the contents of a <strong>controlled vocabulary</strong>.</td>
</tr>
</tbody>
</table>
Index

A
Assignment 1-6
AssignmentIterator 3-11
AssignmentPair 3-14
AssignmentPairIterator 3-14

B
Base Maps 1-2

C
CannotResolveID 3-4
Chromosome map 1-1
Compliance points 1-2
Controlled Vocabularies 2-3
Controlled vocabularies 3-1
CORBA
  contributors iii
  documentation set ii
Correlated Maps 1-3
CytogeneticElement 3-7

E
Exception CannotResolveID 3-4
Exceptions 3-1

I
Identifier 1-4
Identifier Strings 2-5
id-field 2-6
Interface AssignmentIterator 3-11
Interface AssignmentPairIterator 3-14
Interface CytogeneticElement 3-7
Interface LinearMap 3-8
Interface LQSVocabularyFinder 3-4
Interface Map 3-6
Interface MapCorrelation 3-15
Interface MapCorrelationFactory 3-13
Interface MapFactory 3-9
Interface MapIterator 3-9
Interface MapsQueryLanguageType 3-9
Interface OrderedMap 3-7
Interface Vocabulary 3-2
Interface VocabularyEntryIterator 3-2
Interface VocabularyFinder 3-3
Iterators 2-2

K
kind-field 2-6

L
Lifecycle Issues 2-12
LinearMap 3-8
LQSLink 1-3
LQSVocabularyFinder 3-4

M
Map 1-5, 3-6
MapCorrelation 1-6, 3-15
MapCorrelationFactory 1-6, 3-13
MapFactory 1-6, 3-9
MapIterator 3-9
Mappable 1-4, 2-6, 3-5
Mappable and Map 2-7
MappableAssignment 3-10
Mappables and Assignments 2-7
MapPair 3-15
MapsQueryLanguageType 3-9
Module DsLSRGenomicMaps 1-4, 3-4
Module DsLSRLQSLink 3-4
Modules 1-3
Molecular genetics 1-1

N
Nested Maps 1-2
Nested maps 2-7

O
Object Management Group i
  address of ii
Objects-by-value 2-2
OrderedMap 3-7
ordering 2-12

P
Position 1-6, 3-12

Q
Queries 2-9
QueryString 1-4

R
RelativeMetricPosition 3-13
RelativePosition 3-13
Retrievals 2-9
Retrievals and Queries 2-8

S
SubMapAssignment 3-10

T
Typedef AssignmentPair 3-14
Typedef Identifier 3-4
Typedef MapPair 3-15
Typedef VocabularyString 3-2

V
Valuetype Mappable 3-5
Valuetype Position 3-12
Valuetype RelativeMetricPosition 3-13
Valuetype RelativePosition 3-13
Valuetype VocabularyEntry 3-2
Valuetypes Assignment 3-10
Vocabulary 1-4, 3-2
vocabulary strings 2-3
VocabularyEntry 1-4, 3-2
VocabularyEntryIterator 3-2
VocabularyFinder 1-4, 3-3
VocabularyString 1-3, 3-2

W
wildcards 2-11
Index