The Open Geodata Interoperability Specification

Draft Base Document—OGIS Project Document 94-025R2

December 22, 1994

Authored by

The OGIS Project Members

Editor: Kurt A. Buehler

Copyright © 1994, Open GIS Consortium, Inc.

Open GIS Consortium, Inc.
35 Main Street, Suite 5
Wayland, Massachusetts, USA 01778
Tel: +1 508 655 5858
FAX: +1 508 655 2237

All rights reserved. no part of this document may be copied or reproduced in any manner without the expressed written permission of Open GIS Consortium, Inc.

List of OGIS Project Members:

To be supplied.

List of Contributing Non-Members:

To be supplied.

Comments and requests for information concerning change proposals should be sent to the Editor:

Mr. Kurt Buehler
USACERL
Spatial Modeling and Systems Team
2902 Newmark Drive
Champaign, Illinois 61826-9005
Email: kurt@zorro.cecer.army.mil
Tel: +1 217 373 6713
FAX: +1 217 373 7222

Table of Contents

List of Figures ____ *vii*

List of Tables ____ *ix*

Preface ____ *x*

1. Introduction ____ *1*

2. OGIS Reference Model ____ *3*

2.1 Introduction ____ **3**
2.1.1 Vision ____ 3
2.1.2 Scope ____ 4
2.1.3 Open Geodata Model ____ 5
2.1.4 Open Geoprocessing Services ____ 5
2.1.5 OGIS Reference Model ____ 6

2.2 OGIS Application Domain Model ____ **6**
2.2.1 Earth Model ____ 6
2.2.2 Geographic Application Model ____ 8

2.3 OGIS Technology Model ____ **8**
2.3.1 OGIS and the NII ____ 8
2.3.2 OGIS and OSE ____ 10
2.3.3 OGIS Architectural Framework ____ 11
2.3.4 OGIS Service Categories ____ 14

2.4 Candidate OGIS Service Requirements ____ **15**
2.4.1 Characteristics of the OGIS Computing Model ____ 16
2.4.2 OGIS and Other Computing Models ____ 17
2.4.3 Application Context ____ 17

2.5 OGIS Standards Model ____ **17**
2.5.1 General ____ 17
2.5.2 Profiles ____ 18

3. OGIS Geodata Model ____ *19*

3.1 Informative ____ **19**
3.1.1 Introduction ____ 19
3.1.1.1 Strategy ____ 19
3.1.1.1.1 Notational Conventions ____ 20
3.1.1.1.2 Functional Programming Interfaces ____ 21
3.1.1.2 Assertions and Definitions ____ 21
3.1.2 Type Definitions ____ 23
3.1.2.1 Preliminary Definitions ____ 23
3.1.2.2 Type ____ 23
3.1.2.3 Well Known Types ____ 25
3.1.2.4 Objects and Classes ____ 26
3.1.2.5 Some Common Types ____ 26

3.1.2.5.1 The Root Type 27
3.1.2.5.1.1 Interface, Type and Class: Basic Building Blocks of Metadata 27
3.1.2.5.1.2 Root Factory 29
3.1.2.5.2 Identity, Persistent and Transient Objects 29
3.1.2.5.2.1 Identity 30
3.1.2.5.2.2 Transient 30
3.1.2.5.3 Template 31
3.1.2.5.4 List or Array 31
3.1.2.5.4.1 List Factory 32
3.1.2.5.5 Multiset (or Bag) 33
3.1.2.5.5.1 MultiSetOf Factory 34
3.1.2.5.5.2 MultiSetOfIterator 34
3.1.2.5.6 SetOf 34
3.1.2.5.6.1 Set Of Factory 35
3.1.2.5.7 Tuple 35
3.1.2.5.7.1 Tuple Factory 36
3.1.2.5.7.2 Tupletype Factory 36
3.1.2.5.8 Relations 36
3.1.2.5.8.1 Relation Factory 36
3.1.2.5.9 Ordered Type 37
3.1.2.5.10 Stored Function 37
3.1.2.5.10.1 Stored Function Factory for functions with finite domains 38

3.2 Normative 38

3.2.1 The Spatial and Temporal Templates and Interfaces 38
3.2.1.1 Interval 38
3.2.1.1.1 Interval Factory 38
3.2.1.1.2 Closed Interval 39
3.2.1.1.3 Open Interval 39
3.2.1.1.4 Open Interval Factory 39
3.2.1.1.5 Mixed Intervals 39
3.2.1.2 Sorted Lists and Sorted Lists of Intervals 40
3.2.1.2.1 Sorted Lists 40
3.2.1.2.2 Sorted Lists Factory 40
3.2.1.2.3 Sorted Lists of Intervals 40
3.2.1.3 Curve of an Arbitrary Type 41
3.2.1.3.1 Curve of Type 41
3.2.1.3.2 Multi-curve of Type 41
3.2.1.3.3 Simple and Closed Curves 41
3.2.1.3.4 Temporal Domains and Attributes 42
3.2.1.3.5 Spatial attributes 42
3.2.1.3.6 Spatial Temporal Attributes 42
3.2.2 Geodata Definitions 43
3.2.3 Spatial Domains 44
3.2.3.1 Geometry 44
3.2.3.2 Point, and Point Cluster 45
3.2.3.2.1 Point 46
3.2.3.2.1.1 Point Factory 46
3.2.3.3 Curves and 1 Dimensional Geometry 46
3.2.3.3.1 Curve 46
3.2.3.3.2 Line 48
3.2.3.3.2.1 Line Factory 48
3.2.3.3.3 Line String 49
3.2.3.3.3.1 Line String Factory 49

3.2.3.4 Surfaces 49
3.2.3.4.1 Simple Surface 50
3.2.3.4.1.1 Surface Domains 50
3.2.3.4.1.2 Simple Surfaces 51
3.2.3.4.2 Complex Surfaces 52
3.2.3.5 Solids 52
3.2.3.6 Complexes 52
3.2.4 Features and Attributes 53
3.2.4.1 Spatial Object 53
3.2.4.1.1 Spatial Object Factory 55
3.2.4.2 Feature 55
3.2.4.2.1 Attributes and Attribute Descriptors 56
3.2.4.2.2 Feature Type 57
3.2.4.3 Sets of Features, Datasets, and Catlogs 58
3.2.4.3.1 Conceptual model 59
3.2.4.3.2 Components 60
3.2.4.3.2.1 FeatureSet 61
3.2.4.3.2.2 Dataset 61
3.2.4.3.2.3 Catalog 61
3.2.4.3.2.4 Catalog Service 61
3.2.4.3.3 Feature Dictionaries and Mappings 62
3.2.4.3.3.1 Feature Dictionary 62
3.2.4.3.3.2 Feature Dictionary Service 64
3.2.5 Networks and Topologies 65
3.2.6 Coverages 65
3.2.6.1 Types of Coverages 65
3.2.6.1.1 Raster Coverages and Image Processing 65
3.2.6.1.2 Grid Cell Coverages 66
3.2.6.1.2.1 Cell 66
3.2.6.1.2.2 Grid Cell 66
3.2.6.1.3 Sampled Grid Coverages 66
3.2.6.1.3.1 Post 66
3.2.6.1.3.2 Sampled Grid 66
3.2.6.1.4 ‘Polygon’ Surface Coverages 67
3.2.6.1.5 ‘Polyhedra’ Solid Coverages 67
3.2.6.2 Coverage Processing 67
3.2.6.2.1 Overlay and Value Tuples 67
3.2.6.2.2 Feature and Coverage Synthesis 68
3.2.7 Simplex, Cell, and Complex 68
3.2.7.1 Simplex 69
3.2.7.1.1 Simplex Factory 69
3.2.7.2 Cell 70
3.2.8 Method Invokation 71
3.2.8.1 Types 71
3.2.8.2 Classes 72
3.2.8.3 Gathering Methods 73
3.2.8.4 Invoking Methods 73
3.2.8.5 Example 1 73
3.2.8.6 Example 2 74
3.2.8.7 Types 74
3.2.8.7.1 Method 75
3.2.8.7.2 FunctionRegistry 75
3.3 Future Work 75

Appendix A: Type Hierarchy _____ *I*
List of Acronyms _____ *III*
Glossary _____ *V*
References _____ *X*
Index _____ *XII*

List of Figures

Figure 2-1 NII Services Model 9
Figure 2-2 OSE Reference Model (Entities and Interfaces) 11
Figure 2-3 The OGIS Computing Model 12
Figure 2-4 Pluggable Tool Interfaces 13
Figure 2-5 OGIS Service Areas 14
Figure 3-1 Low Dimension Discs and Spheres 22
Figure 3-2 Relationship Between Type and Class 24
Figure 3-3 An example of application use of well known type hierarchies 25
Figure 3-4 Subtype Relations among the Common Types 27
Figure 3-5 Basic Building Blocks of Metadata 28
Figure 3-6 Flow of a Generic Command 29
Figure 3-7 Spatial Features, Coverages and their Relationship as Types 44
Figure 3-8 A surface in E^2 Defined by Cutting 4 D^2 a line, and a point, out of a larger D^2 50
Figure 3-9 Relationship between high-level OGM abstractions and aggregations 60
Figure 3-10 Relationships between feature types, dictionaries and mappings. 62
Figure 3-11 Overlay for Polygon Coverages 68
Figure 3-12 A Surface as a 2 Simplical Complex 70

List of Tables

Table 2-1 Candidate Service Requirements 16
Table 3-1 Definition of Type, Object and Class 23
Table 3-2 Definitions of Domain Types 43
Table 3-3 Definition of Feature and Coverage 43
Table 3-4 Basic Attribute Type Definitions 43

Preface

This document reflects the work of the OGIS Technical Committee. It is not to be interpreted as the final specification, nor is it to be considered complete. It is created and released to provide a working base document for change proposals and evaluation of current status only.

1. Introduction

This section will introduce the OGIS. It will, when completed, cover the following areas:

- Why is OGIS needed?
- What problem does OGIS address?
- List the Technical Vision, Goals, and Requirements of the OGIS Project.
- Outline the contents of the following chapters and the appendices.
-

2. OGIS Reference Model

2.1 Introduction

2.1.1 Vision

The Open GIS Consortium shares the vision of the National Information Infrastructure (NII). The NII will have a profound affect on the nation, and indeed the world, by making a vast array of information, communications, and other services readily available to all citizens at affordable prices. The services enabled by the NII will provide new opportunities and may dramatically change how citizens conduct business, learn, receive health care, shop for and purchase products, and use their leisure time. There are six guiding principals that determine the shape and form of the NII, namely the NII must: serve all citizens; promote the principals of free enterprise; protect the rights of users and stakeholders; promote interoperability and open standards; provide high-quality, high-capability services; and provide an information marketplace. [2-1]

Through the collaborative efforts of government, academia and industry, the Open GIS Consortium focuses on developing the role of geospatial data and geoprocessing services as part of the NII. OGIS geoprocessing services will serve needs across a broad spectrum of institutions and enterprises with functions for the creation, movement, distribution, use and management of geospatial data. These services will be available as "componentware" that can be bundled or integrated with any other information technology. This geoprocessing capability will be extraordinarily flexible and accessible, with many levels of functionality. It will be pervasive, supporting a large variety of applications and services from which users and providers may select according to their needs, capabilities and resources. OGIS geoprocessing services will operate with a variety of information appliances, including computers, cameras, personnal communicators, telephones, televisions, and remote sensors.

The technical vision of the OGIS Project is: The Open Geodata Interoperability Specification will result in the ability of an application developer to use any geospatial data and any geospatial function or process available on "the net' within a single environment and a single work flow [2-2]. The application developer is the primary user of OGIS capabilities, consisting of geospatial data objects and geoprocessing service objects with standardized Application Programmer Interfaces (API). Geospatial data are any earth features or earth phenomena defined by spatial and temporal coordinate systems. A geospatial function or process is any function or process which handles or operates on geospatial data. The "net " defines the full extent of the network which is accessible from any given information appliance node. The expected pervasive use of OGIS technology and associated large geospatial data volumes within the NII (i.e., large data volumes in comparison with most types of information which will flow through the NII), drive yet unknown requirements for network bandwidth. The OGIS operating environment, within the context of NII, includes all devices accessible through the "net".

Similarly, an OGIS work flow extends across the "net", accessing geoprocessing services and other geoprocessing applications in a distributed, cooperative fashion.

2.1.2 Scope

OGIS addresses technology barriers that exist within the geoprocessing community and between the geoprocessing community and other segments of the information technology industry. Traditional geoprocessing systems have been called "stovepipe" or "closed" systems, since they were developed in an isolated fashion to perform limited tasks. Organizations with traditional geoprocessing applications and other traditional information technology typically have several distinct monolithic applications, often platform dependent, with limited ability to share computing and data resources. Often there are redundant functions and databases between applications. Training requirements are excessive because of the diversity of user interfaces. Designed for limited utility, these applications lack the facilities to easily accomodate new methods and data types, as they are required. These shortcomings greatly limit the potential of this technology.

In contrast to traditional geoprocessing technology, the open geoprocessing capabilities defined within OGIS represent a common technology foundation with the following major characteristics:

Interoperable—has standard interfaces to geospatial data and geoprocessing services. These interfaces support geospatial data object access, geospatial data object interchange, distributed client-server geoprocessing operations and distributed peer-to-peer geoprocessing operations. A future OGIS specification will define common human-technology interface requirements.

Ubiquitous—provides the means for all information technology applications to readily take advantage of OGIS geospatial data and geoprocessing services through standard interfaces and protocols.

Reliable—operates at a high-level of manageability and integrity. Provides a technology framework that supports GIS Certification.

Easy to use—employs logical and consistent rules and procedures for the use of geospatial data and geoprocessing services; unnecessary geospatial data and geoprocessing complexity are "hidden" from the application developer.

Portable—is independent of software environment, hardware platform and network.

Cooperative—supports shared computing and shared data resources; can be easily combined with other information technology.

Scalable—consists of "plug-and-play" geoprocessing software components which are configurable for any geoprocessing application or computing environment (e.g. platform size). Can operate with any size database.

Extensible—can assimilate new geoprocessing software and geospatial data types, or can accomodate new technologies upon which OGIS is dependent, such as distributed computing environments, as they become available.

Compatible—preserves investment in legacy technology by providing the means to seamlessly integrate this technology (i.e., in a fashion which is transparent to the user), particularly existing geoprocessing software and geospatial data and related information technology, with OGIS-compliant geoprocessing applications. Also, is compatible and does not overlap with supporting information technology, particularly distributed computing environments and database systems.

The OGIS Technical Committee is defining a two part specification and conceptual model, the Open Geodata Model (OGM) and Open Geoprocessing Services (OGS) . This

specification and model exist to enable open, flexible implementations of geoprocessing applications. This document proposes a reference model that orients the reader of the specification in understanding the conceptual model, and it's relationship to real world phenomena (specifically, the earth and geographic applications), to computing environments and to standards frameworks for information technology.

2.1.3 Open Geodata Model

The Open Geodata Model (OGM) provides a common Geodata Model for all spatio-temporal data. The model consists of two basic components, Features and Coverages. A Feature is an abstract or real-world entity that holds some interest and which may have location or extent as one of its attributes. A Coverage is a stored function which associates points to a defined type (value).

2.1.4 Open Geoprocessing Services

The Open Geoprocessing Services (OGS) define a common, consistent set of geoprocessing software interfaces. These interfaces define the behavior of geoprocessing software services which access, interchange, manage, manipulate and present geospatial data (OGM). These services have the following common features:

- Independent of operating system, programming language, human-technology interface, hardware and network.
- Operates with each of the major distributed computing environments and between distributed computing environments, with consistent interfaces to these services.
- Operates with each of the major database languages, with consistent interfaces to these languages.
- Supports dynamic interface definition for data (OGM) and process (OGS) extension.
- Supports client-server and peer-to-peer processing.
- Supports shared data and process resource utilization.
- Service modules can be bundled together in any configuration.

There are five categories of services:

- OGIS Object Access Services
- OGIS Object Interchange Services
- OGIS Object Management Services
- OGIS Object Manipulation Services
- OGIS Human-Technology Interface Services

OGIS Object Access Services provide the means to locate, access, retrieve or store distributed geospatial data (OGM), and the means to locate and invoke local or remote services that act on distributed geospatial data. OGIS Object Interchange Services provide the means to exchange geospatial data between geoprocessing systems (import and export), including legacy systems, through distributed services. OGIS Object Management Services provide the means to replicate and migrate geospatial data, manage geospatial data concurrency and provide geospatial data integrity protection. OGIS Object Manipulation Services provide the means to interpret, combine and correlate geospatial data, in a correct and consistent manner, in order to support cooperative computing and common geoprocessing application paradigms. OGIS Human-Technology Interface Services provide the means to present geospatial data to users and the means to interact with geospatial data through an information appliance.

The OGIS Geoprocessing Services fall into three project implementation phases:

Phase I—Data cataloging, locating, access and sharing through OGIS Object Access, Object Interchange and Object Management Services

Phase II—Standard geoprocessing operations through OGIS Object Manipulation Services

Phase III—Human-technology interface standardization through OGIS Human-Technology Interface Services

2.1.5 OGIS Reference Model

The OGIS Reference Model provides the complete scientific and engineering contextual framework for OGIS Geodata Model and OGIS Geoprocessing Services. The reference model defines this framework in terms of abstraction types and methods consisting of real world, mathematical, computer science and computer engineering constructs (elements, rules and behaviors). This definition extends the traditional meaning of reference model as it pertains to computer architectures, in that the geographic application context is added.

There are three parts to the OGIS Reference Model:

OGIS Application Domain Model—defines the application context for OGIS, consisting of the Earth Model and Geographic Application Model

OGIS Technology Model—defines the conceptual technology architecture for the OGIS Geodata Model and OGIS Geoprocessing Services, and their properties, behavior, and geoprocessing application context

OGIS Standards Model—defines the community standards and standards profiles which comprise complete OGIS environments, building upon a foundation of existing standards and specifications.

The three parts of the reference model are discussed in the next three sections. A glossary and list of acronyms are provided in the last two sections.

2.2 OGIS Application Domain Model

The OGIS Application Domain Model consists of the abstraction framework for geographic applications. Geographic Applications are those applications which pertain to the earth and earth phenomena, with known spatial and temporal reference frames. The range of Geographic Applications encompasses all problems for which one wishes to model the earth and then use the model to solve these problems. There are two aspects to the OGIS Application Domain Model: the Earth Model and the Geographic Application Model. These are expressed in a human context versus computer context.

2.2.1 Earth Model

The Earth Model represents the method of abstracting the earth and earth phenomena, conceptually and mathematically. By conceptual we mean how humans perceive the world around them, and in this case, in a geographic context. The conceptual view has both a visual component, i.e., graphical or imagery, and an intuitive/interpretive component which provides meaning and understanding. The nature of the conceptual view is such that it is unique for each person. But for practical

reasons, the conceptual view must be expressed in a common way. For example, consider the map, where common symbology are used to represent earth phenomena.

A goal of the OGIS Project is for the Earth Model to closely correspond with reality. For geographic applications with the most rigorous requirements, this would be a "virtual reality" model. However, models with less detail must be supported due to the practical limits of present technology and geographic databases. For example, a two-dimensional representation of surface information (e.g. map) will suffice for many applications. Yet, the focus is to model the earth, not to model maps and charts, which has been a limiting factor in some traditional GIS implementations.

The Earth Model is comprised of three spatio-temporal components: the Surface Configuration Model, Feature Model and Volume Model. The Surface Configuration Model represents the elevation, shape, slope and aspect of the earth's solid surface and water surfaces. (Bodies of water have both a solid surface, e.g., ocean floor, and a water surface, e.g., mean sea level.) The Feature Model contains three dimensional representations of earth surface and subsurface features and/or the projection of these features onto the surface. The Feature Model is also used to represent earth properties and characteristics classified into discrete categories and represented as surface features, e.g., land parcels, contaminated areas, etc. The Feature Model is comprised of one or more Feature Levels. Feature Levels contain a collection of Features or Coverages which form a logically consistent grouping, in terms of resolution, accuracy, content and context. It may or may not be possible to compare and correlate information between Feature Levels. Special OGIS Geoprocessing Services are required to deal with the ambiguities and artifacts found in comparing and correlating data between disparate Feature Levels. A subset of Features within a Feature Level, which have some distinct significance to a user, are referred to as a Feature Set. The Volume Model represents the properties and characteristics of three-dimensional spaces above or below the earth's surfaces, namely the atmosphere, bodies of water and earth's mantle.

A mathematical representation of the Earth Model is required in order to implement geographic applications on a computer. This consists of spatial and temporal coordinate systems defining the geodetic reference frame for the Earth Model. The Horizontal Geodetic Datum defines the primary earth reference figure, an ellipsoid of revolution, or spheroid. The Vertical Geodetic Datum defines the primary reference surface for defining elevation. A common geographic coordinate system, such as World Geodetic System 1984, applies to these datums. Other spatial coordinate systems can be related to geographic coordinates via coordinate transformation functions. (Also known as the geoid.) Universal Time (i.e., Greenwich Mean Time) serves as the primary temporal reference system. Other temporal reference systems, such as Local Time and Solar Time, relate to Universal Time.

An open geoprocessing application environment requires consistent treatment of the Earth Model so that applications may interoperate and share data from the same model. This leads to the definition of the Base Earth Model. The Base Earth Model provides a common, logically consistent definition of the Earth Model within an OGIS environment; it includes a geodetic and temporal mathematical reference frame, and the components of Surface Configuration, Surface Features and Volumes which are independent of OGIS applications and have unambiguous definitions. The purpose of the Base Earth Model is to limit the overhead associated with those operations within an OGIS Application Environment that deal with geospatial data inconsistencies and ambiguities, particularly when comparing and correlating geospatial data between Feature Levels. This places greater emphasis on providing rigorous methods to resolve problems as objects are introduced to an OGIS Application Environment. Thus OGIS-

compliant Geoprocessing Applications which rely on these databases may be optimized to handle a clean Earth Model.

A given OGIS Application Environment may have one or more Base Earth Models, due to the practical limitations of the available source data that are used to construct the databases, or the existance of several applications within the environment which have disparate data needs and have no requirements to interrelate these data. However, OGIS Application Environments may share data from disparate Base Earth Models. These cases mandate OGIS Geoprocessing Services which deal with geospatial data inconsistencies and ambiguities as geospatial data are exchanged, compared and correlated.

2.2.2 Geographic Application Model

This section is the subject of an upcoming change proposal.

Ed.: We need to at least state what this section is about.

2.3 OGIS Technology Model

This section places OGIS in the context of the *human-technology environment*. It describes a conceptual architecture model in terms of the interconnection of basic functional elements and the interfaces between them. The model serves to position OGIS within the larger context of Information Technology and Distributed Computing Environments (DCEs). In particular, the model defines OGM and OGS relative to DCE, database, the human-technology interface and other non-geoprocessing software.

The model is based on standards and is evolutionary in nature. Not all of the details are known. Not all of the needed standards have emerged. Not all of the functional requirements, especially future needs and context, are well understood. The objectives of the technology model are simply to:

- provide a common reference for describing and comparing existing technologies and standards, including OGIS;
- identify a set of common infrastructure services that support scalable, extensible and seamless integration of interoperable tools;
- serve as a basis for determining interfaces and relationships between environment components

2.3.1 OGIS and the NII

The goal of the National Information Infrastructure (NII) is to provide nationwide services in support of education, health care, law enforcement, public safety, manufacturing, resource management, entertainment, commerce and interpersonal communication. The NII will incorporate a wide variety of equipment with differing capabilities and services including cameras, scanners, recorders, keyboards, telephones, fax machines, computers, televisions, video and audio media, cable, wire, satellites, switches, optical fiber transmission lines, microwave nets, etc. The key elements of the NII include the following: a *bitway*, or network, and the various components required to transmit and manage digital data across the physical pathways of the infrastructure; an *information appliance*, a customer-controlled device used to access services; a *service provider to the customer* is the implementor of an information appliance access point; a

bitway-bitway crossover enables interworking between bitways; finally, a *information service provider* delivers value-added services both within and outside the NII [2-9].

The structure of the NII, in its most basic form, can be represented with three layers: applications, services and bitways. The NII services model shown in Figure 2-1 goes beyond a simplistic three-layer model to focus on services, the interrelationships among the various services and the interfaces between bitways and services and between services and applications. This is not a layered diagram but is instead an abstract partitioning of NII services and has no relation to how the listed services are implemented relative to one another [2-9].

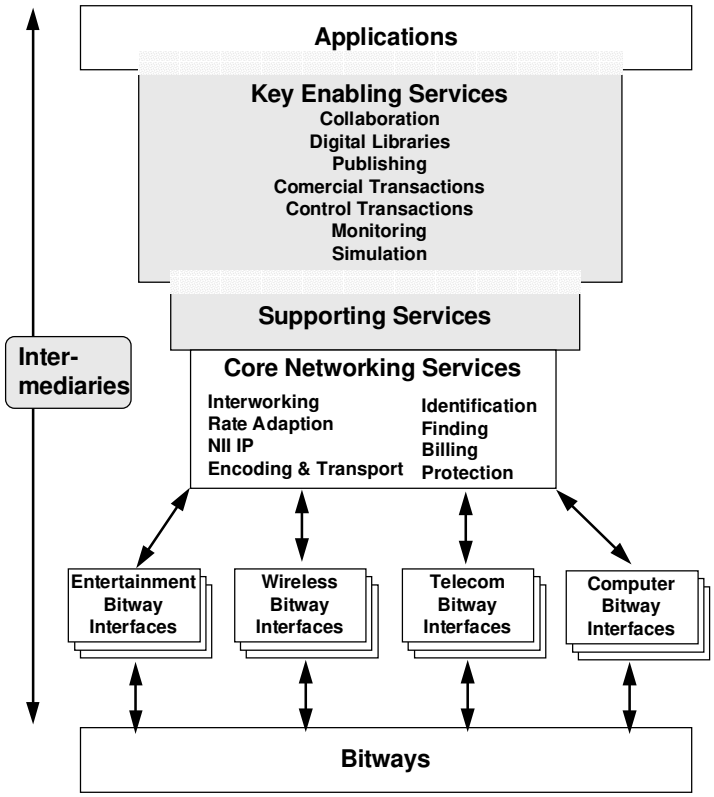


Figure 2-1 NII Services Model

In Figure 2-1, the highlighted service areas show where OGIS services (OGS) are positioned relative to the overall NII structure and the specific services and interfaces thought to be necessary for its implementation.

In the NII, *application areas* represent commonly understood spheres of activity which will make significant use of the services provided through the NII. Application areas such as manufacturing, electronic commerce, healthcare, education, government services, environmental monitoring and libraries represent the *users* of the NII. Many, if not all, of the targeted NII application areas will have a need for accessing spatially referenced information. The business of other NII application areas may directly involve producing, manipulating and disseminating spatial data.

In the NII services model, applications are collections of functions available to a user through an information appliance. Applications are based on combinations of *key enabling services*, supplemented with supporting services and value-added *core services*. Where necessary, *intermediaries* will be used to integrate services to achieve

interoperation of services within an application [2-9]. It is entirely conceivable that applications will use OGIS services to transparently produce, access, manipulate and dissiminate spatial information within the NII. Collections of OGIS services may themselves be an application within the NII.

Key enabling services (enablers) represent well-defined capabilities provided to support specific hardware/software systems within the various NII application areas. These enablers are intended to span the different application areas. Some types of enabling services are uniquely matched to specific types of applications. Alternatively, the capabilities of other enablers may be reused across many different application areas. A digital library for spatial data, such as the National Geospatial Data Clearinghouse (NGDC), is an example of a key enabling service in which OGIS services may play a critical role.

The functionality of key enabling services depend in part on the use of common supporting services. Supporting services are built upon the NII core networking services or, alternatively, may be constructed to access individual bitway interfaces directly. They may be used individually or combined within enablers and user applications. Providers of these services may bundle existing services in novel ways; they may provide tools, aids or graphical user interfaces to enhance utility of other NII services; provide services for use in applications unrelated to the NII except in using its services; and they may provide services that are specific to a particular industry, business or government activity. Providers of common services include corporations, small businesses, government agencies, industry consortia, education institutions and private individuals.

Since a wide array of applications require access and manipulation of spatially referenced information, OGIS services are natural candidates for use as common services supporting both key enabling services and user applications within the NII. For example, an OGIS data access service could be used to deliver spatio-temporal information, per the OGM, to requesting applications or services within the NII. Such a service could be used by enabling services, such as the NGDC digital library, to package spatial data for delivery to requesting applications. Alternatively, the OGIS data access service could be used directly by applications to support data fusion and interoperability.

Intermediaries are services that provide functions for interconnecting, adapting and facilitating the services offered by service providers. Intermediaries bring together component services, allowing construction of larger, more capable services. Intermediaries can be used to integrate a collection of supporting services and value-added core services in order to provide the capability of a key enabling service. Similarly, intermediaries can be used within application systems to bind selected enabling services into a common, reusable application framework. As with key enabling and supporting services, OGIS services may either support construction of intermediary services or may act as intermediaries themselves. For example, an OGIS query service, in its roles of locating and accessing relevant data and optimizing queries, may behave purely as an agent, broker, mediator and trader for other OGIS and NII services.

2.3.2 OGIS and OSE

Ed.: OSE model goes here (page8 of OSE Arch .Framework for NII Services and Standards), explain the figure and tie OGIS into it!]

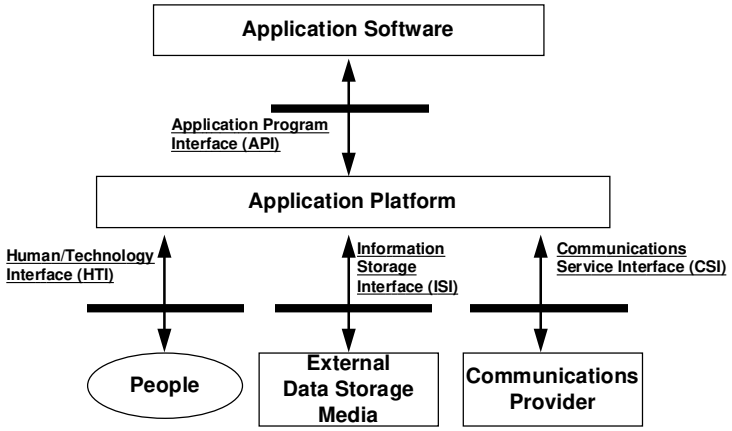


Figure 2-2 OSE Reference Model (Entities and Interfaces)

2.3.3 OGIS Architectural Framework

We have adapted the NIST/ECMA reference model [2-8] as a basis for describing the technical aspects of integrated application environments with respect to OGIS. The OGIS Computing Model, shown in Figure 2-3, divides an application environment into functional elements called *services.* Services provide specific functionality to an application. They are task-oriented entities which have a specified interface and a consistent behavior. In the computing model services are grouped to reflect basic functional areas of the computer environment:

- the objects (data) that are to be manipulated;
- the tasks (processes) that are to be performed;
- the user interactions (human interface)

The services defined in the OGIS Computing Model address the three fundamental forms of system integration:

- *data* integration, which is supported by the Object Management and Data Repository services;
- *control* integration, which is supported by Process Management and DCE services;
- *presentation* integration which is supported by Human-Technology services.

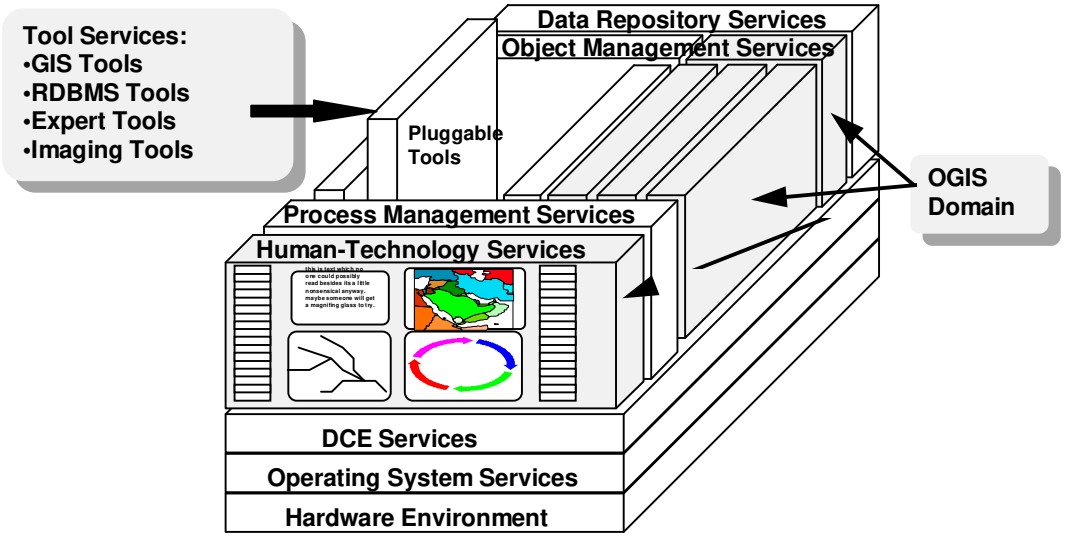


Figure 2-3 The OGIS Computing Model

The services of the model are:

- Human-Technology Services include the functionality (such as windows and a pointing device on a computer) necessary for a user to interact with an information appliance. For computer applications, a Graphical User Inteface (GUI) that adheres to a standard style guide is required to produce a consistent look and feel across applications. These services support separation of the presentation and other behaviors of functionality from its provision by the underlying tools in the environment.
- Process Management Services incorporate problem domain and organization-specific issues of workflow control, task sequencing and event notification. Services in this group may include management, documentation, evaluation, assessment, poliicy-enforcement, maintenance and other activities.
- Tool Services provide any of the additional functionality needed to support the end user and the application. Examples of services include functions for manipulation of data objects (e.g.,manipulation of spatial data, imagery, graphics, audio and video), interchange of data between external formats and environments and analysis of data in support of workflow or decision-making processes. In an *OGIS application environment*, *OGM features* are accessed, manipulated and interchanged by the OGIS Object Management, Access, Manipulation and Interchange portions of the Computing Model.
- Object Management Services deal with the management of structured data, text, imagery, graphics and other media such as voice and video that can be defined independent of the processes that create or use it, maintained indefinitely, and shared among many processes. These services broadly cover the definition, storage, maintenance, management and access of object entities and the relationships among them. In an *OGIS application environment*, *OGM features* are managed by the OGIS Object Management Services portion of the Computing Model.

- Data Repository Services represent the actual data (text, imagery, *surface features*, graphics, voice, video) that the user and application access and manipulate. In an *OGIS application environment*, this data is actually the objects that are being managed and manipulated by the OGIS Object Management, Access, Manipulation and Interchange portions of the Computing Model.
- Distributed Computing Environment (DCE) Services provide a standard communication mechanism which may be used for inter-tool and inter-service communication within a distributed network environment. Services may include message broadcasting, process invocation, remote procedure call, data communication, transparent file access and network security.
- Operating System Services include both the low-level services that manage hardware resources and the higher level software mechanisms that facilitate user and application interaction with the hardware and operating system.
- Hardware Platforms consist of the hardware components of a workstation or server. The hardware platform provides the central processor, local memory, and communications devices that make up the physical components of the system. Peripheral devices are normally attached to a hardware platform, are independent of other portions of the architecture and are accessed through other system services such as the operating system.

In addition to defining the infrastructure services of an application platform, the OGIS Computing Model describes the role of service interfaces for building tools. The service interfaces, APIs, define the behavior and isolate the implementation details of services. As shown in Figure 2-4, tools in the OGIS Computing Model interoperate with other clients and services via interfaces to infrastructure services such as Object Management, DCE and Process services. Tools in the model are structured packages of data and executable code which typically reference platform services (via their supported interfaces) in addition to their own domain-specific or private algorithms and data.

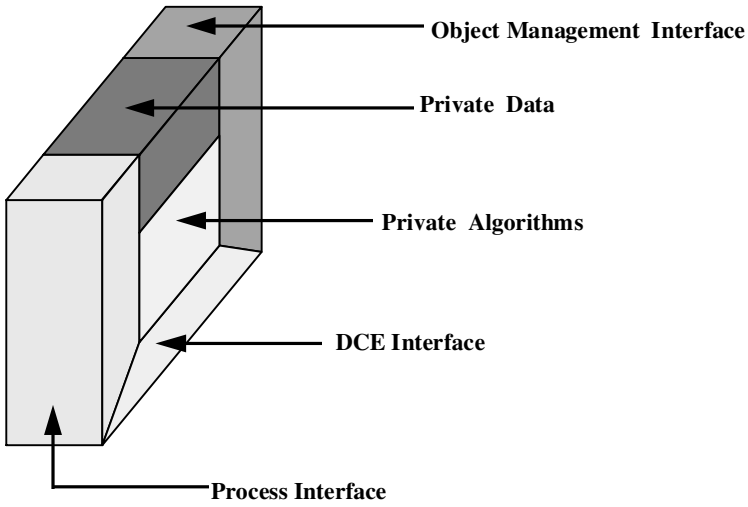


Figure 2-4 Pluggable Tool Interfaces

2.3.4 OGIS Service Categories

Figure 2-5 shows in greater detail the OGIS service categories of the OGIS Computing Model:

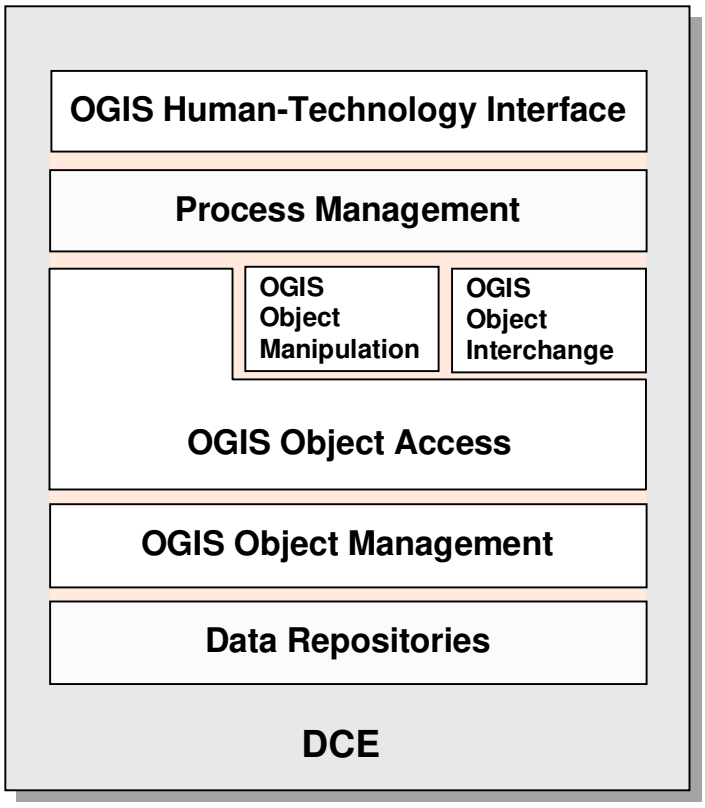


Figure 2-5 OGIS Service Areas

- OGIS Human-Technology Interface Services present geospatial data to users and allow users to interact with geospatial data through an information appliance. On a computer workstation, geospatial data are conveyed as display objects on the workstation screen, such as icons, windows or graphical features within a window. Users interact with geospatial data display objects through interactive devices provided with the workstation, including mouse and keyboard. Geospatial data display objects have a consistent look and feel across applications.
- OGIS Object Management Services deal with the management of *OGM features* within an *OGIS application environment*. These objects (data and behavior) can be defined independent of the tools that create or use them, maintained indefinitely and shared among many tools across application domains. Types of services include data dictionary and catalog services, interfaces to data repositories (e.g., database management systems), distributed data management services and data management security, but not the actual objects (data) themselves. Interfaces to data repositories and database management systems are identified, as are the interfaces to OGM object access services.

- OGIS Object Access Services provide standard interfaces supporting data transfer, interchange and transparency of *OGM features* between clients and services in the *OGIS application environment*. These services are based on a uniform conceptual model of *surface features* defined by the OGM and its component objects.
- OGIS Object Manipulation Services provide algorithms, functions and tools required for manipulation of *OGM features*. Types of services include rendering, display, quality assurance, mensuration, analysis and transformation of *OGM features*.
- OGIS Object Interchange provide specialized support for the interchange of sets of *OGM features* to other, external representations or implementations of *surface features*. Object interchanges may occur between local or external environments via an *external environment interface (EEI)*.

Implementations of these services are supported within an *OGIS application environment profile*.

2.4 Candidate OGIS Service Requirements

Table 2-1 outlines a suite of candidate OGIS service requirements. These candidate service requirements reflect basic functionality which may be required in OGIS implementations. The specification of these candidate service requirements may be phased over time. This list should not be considered to be complete. Furthermore, the existing list, including definitions and categorization, may be altered as a result of additional work in this area. The identification of these candidate service requirements carries with it no implications for how this functionality will be specified in the OGM.

Service	Definition	Service Category
Spatial Reference Transformations	Transformation of coordinates from one coordinate system to another	Manipulation, Interchange
Metric Services	Quantitative measurements in either space or time	Manipulation
Set Operations on geospatial objects (e.g. Union, intersection)	Set operations applied to geospatial and/or objects	Manipulation
Data Model Schema Transformations	Reorganization and restructuring of the conceptual, logical and physical representation of the data	Manipulation
Geospatial Object Locate	Use of the geospatial position/extent of an object as a mechanism to locate another object	Access
Geospatial Indexing	Supporting functionality for geospatial locate and/or topological services, indexing based on/limited to space and/or time.	Access, Management
Topological Services (geospatial-join)	Geospatial operations based on qualitative relationships in geometry (e.g. containment, connectivity, adjacency, overlap). Proximity is explicitly excluded.	Access, Manipulation
Catalog	A service that enables the location and identification of datasets that and catalog copy/move.	Access, Management
Feature Dictionary	A mechanism for cataloging types of features and mappings to other feature dictionaries	Access
Dataset Management	Geography based dataset management which includes both partitioned and seamless perspectives of the data. geospatial join, tiles as indexing vs. storage unit, plus organizational issues	Management
Coverage/functions (overlay,	Map algebra (arithmetic, boolean, logical), image	Manipulation

...image processing,)	processing, polygon processing, grid operations, multi-type operations	
Navigation	Topologic (navigation) operations on networks of segments, nodes, regions, surfaces	Manipulation
Query (identity, proximity)	Location/identification of objects based on attribute value, topologic relations, and metric properties and relations	Access
Service Registry	The capability to locate and invoke specific functionality	All
Presentation	Contents of human-technology interface	Human Technology Interface
Attribute Services	Services that support the manipulation of attributes	Manipulation
Geometry Services	Services that support the manipulation of geometry	Manipulation
Import Services	Translation of non-OGIS data into a format consistent with an OGIS environment (In)	Interchange
Export Services	Translation of non-OGIS data into a format consistent with an OGIS environment (Out)	Interchange
Aggregation Services	Combination of simpler features into more complex features based on some cartographic algorithms	Manipulation
Generalization Services	Geometric simplification of geometric features based on some cartographic algorithms	Manipulation
Browsing Services	Geodata discovery, higher level	Access, Human Technology Interface
Data replication (local cache)	Replication of geodata for the sake of performance	Management
Persistence	Ability to store geodata objects	Management, Access
Transactions and Concurrency	Integrity protection capabilities related to storage.	Access, Management
Negotiation	Client-server interaction to develop an accord in regard to available interfaces.	Manipulation
Artifact/Error Handling	Recognition, description, and mitigation of error occurring in data (error may be defined in terms of accuracy, consistency, and precision)	Management, Manipulation

Table 2-1 Candidate Service Requirements

2.4.1 Characteristics of the OGIS Computing Model

The OGIS Computing Model provides a consistent basis for allocating technical functions to specific portions of each service area of the model. Consistent adherence to the model will facilitate understanding of the role of OGIS in the larger context of Information Technology; permit increased resource sharing between organizations and processes; enhance data connectivity among users and applications; and improve the ability of developers and users to integrate new capabilities into existing environments as well as incorporate legacy systems into new environments. These are the characteristic benefits of *open systems* and *open system environments*.

Central to the definition of open systems are the properties of *scalability, extensibility, diversity,*and *interoperability*. These "ilities" describe the characteristics and behavior of *open system environments* (OSEs). The OGIS Computing Model identifies and describes the properties and behavior of services which support OSEs in general and *OGIS application environments* in particular.

Scalable application environments have the ability to flexibly match, or "right-size", the set of tools (Tool Services in the model) to the needs of a *geoprocessing application*. For example, not all map display applications require full spatial data management capabilities and, as such,

only a subset of OGIS services need be accessed. The configuration of tools in scalable environments should be based on the task and the characteristics of the overall application.

Extensible systems support flexible assimilation of outside data and software into an application environment. This includes the ability to leverage *legacy systems* (data and software) through encapsulation. An extensible *geoprocessing application* is one whose functionality grows as the requirements of the *geographic application* matures.

Diverse systems support a broad range of application domains either through extensibility or the inherent richness of functions in the existing tools. Diverse systems can be thought of as "horizontal" or "enterprise" application environments which support different styles of work and fuse disparate data types within an organization's workflow processes.

Interoperable systems support integrated, cooperative and interchangable tool sets. The mechanisms for enabling interoperable tools are primarily supported by the infrastructure services of the *application platform*: the Process Management Services, Object Management Services and DCE Services of the OGIS Computing Model. Interoperable systems enable tools to be replaced such that no changes to other tools in the environment are required and new tools automatically cooperate with other tools in the environment.

2.4.2 OGIS and Other Computing Models

Ed.: discuss OGIS relative to OMG, OLE/COM, etc...

2.4.3 Application Context

An *application context* is the set of hardware, software and data necessary to perform a sequence of tasks required to achieve an end-user goal. An OGIS application context is comprised of a *geographic application* (the end-user goal), a *computer environment* (humans, hardware, software, data and standards) and a *geoprocessing application*— an implementation of an *OGIS application environment* (supporting the required tasks). In the OGIS Computing Model, each tool provides the actions that are appropriate based on the type of data managed by the tool.

2.5 OGIS Standards Model

Ed.: the details of the OGIS Standards Model are TBD

2.5.1 General

As with the OGIS Computing Model, the purpose of the OGIS Standards Model is to provide an organizational framework for positioning OGIS within the broader context of existing and emerging standards efforts within the computing industry as a whole. In defining the scope of OGIS, it is valuable to identify and describe the role of other standards within the *computer environment,* regardless of whether OGIS itself directly impacts or is impacted by those other computing standards. Another important goal of the OGIS standards model is to identify other standards whose domains overlap that of OGIS for purposes of evaluation and comparison of strengths and weaknesses.

As with the OGIS Computing Model, the standards and profiles described here are evolutionary in nature. Not all of the details are known. Not all of the needed standards have emerged. Not all of the functional requirements, especially future needs and contexts, are completely understood. Again, the primary purpose of the standards model is to help understand the role of existing, evolving and emerging standards by positioning them within a uniform conceptual model of computing environments.

2.5.2 Profiles

Future implementations of OGIS application environments will be based on *profiles,* collections of standards for building complete systems. In particular, we anticipate the definition of multiple *OGIS application environment profiles* which will serve as guidelines for building OGIS application environments from collections of standards to fit specific classes of geographic applications. Examples of classes of profiles follow:

3. OGIS Geodata Model

This chapter contains the current description of the OGIS Geodata Model, as adopted by the OGIS Project Technical Committee. The Technical Committee is currently working on several change proposals that will add to and modify the content of this chapter.

This chapter includes three major sections; 1) Informative, 2) Normative, and 3) Future Work. The Technical Committee passed a motion to include the informative part in the base document, but to isolate it from the normative part of the specification pending review of the OMG object model and extensions. Currently, the OMG Object Model and OMG Interface Definition Language (IDL) does not include collections (list, set, multiset, etc.). The Object Database Management Group (ODMG) has extended the OMG Object Model and IDL to form the Object Definition Language (ODL) to include collections. Current activity within the OMG Technical Committee is heading towards defining a standard collection library. We will include section **Error! Reference source not found.** as informative and use the definitions in the normative section until there is some resolution of these matters.

3.1 Informative

This section is informative, pending resolution of the above mentioned matters. Portions of this section will likely move to the OGIS Reference Model section, when a draft is accepted and placed into this base document.

3.1.1 Introduction

Deep functional interoperability requires a basis for common dialog between processes. That basis for OGIS consists of a "jargon" built to communicate data and process models between cooperative applications. Taken in total, this language describes a conceptual model or world view for GIS

The new building blocks that are required for GIS consist of representations of subsets of space and time, and their interaction with existing modeling concepts, entities and attributes. Because of the special nature of space and time, the concept of "spatial" and "temporal" function will play a large role in the model.

3.1.1.1 Strategy

The basic strategy is to define a set of "well-known types" and common aggregates as the basic building blocks. The well-known types would include common programming types such as integers, real numbers, character strings. The aggregate types would include common programming and database aggregate constructors such as list (array), set, multiset (bag), and tuple.

Each OGIS type will include constructor interfaces or factories deriving data directly or recursively from well-known types. The only exception to this will be some basic

geodata types whose implementation will be determined during the testbed phase of the project.

In defining the interfaces to the various types, when possible, we will do the following:

1. Separate Constructor from other interfaces.
2. Isolate inconsistency in OOPLs (such as C++, Smalltalk, Objective C).
3. Define interfaces recursively with respect to type.
4. Give a mechanism to 'unroll' (recursively decompose) complex types to basic data types.
5. Use inheritance via delegation.
6. Define assertions (an algebra) that define and limit the behavior of the types.
7. Use subtyping only if needed by assertions.
8. Define modifiers and destructors only for persistent data types subtyped from 'identity'.

3.1.1.1.1 Notational Conventions

This section outlines notational conventions that we will use throughout the specification.

In defining type, we will use a C++ like syntax that uses the object in question as the first parameter. For example, the following interface statement defines a type X, which has an interface MethodName taking one parameter of type Z, and returning an object of type Y. The 'under' specifies what other interfaces must always be supported by objects supporting this one. The following is a subtype of 'Root.'

```
Interface X under Root
{
Y MethodName(Z);
}
```

Writing pseudo code using this interface would look like the following (in object-oriented languages like C++):

```
X x1;
Z z1;
Y y1;
y1 = x1.MethodName(z1);
```

Or, alternatively (as in C), the object becomes the first parameter of the function implementing the interface:

```
X x1;
Z z1;
Y y1;
y1 = MethodName(x1, z1);
```

In interface specifications, default values can be specified for parameters by appending the parameter type with an "= <default value>."

One particular exception to this rule is factory types, which need not be mentioned in the execution of the function. In C++, a class factory is most often invoked using a new syntax, which creates an object that persists longer than the scope of the statement, as follows:

```
Point p1 = new Point((30, 30), "Lat. Long");
```

If the scope of persistence is not taken into consideration (which we will determine in other manners), then the "new" is redundant, so we will use the following syntax:

```
Point p1 = Point((30, 30), "Lat. Long");
```

To distinguish a factory, its name will be either <type_name>Factory or, if the type being created is in turn a factory, <type_name>Type or <type_name>Descriptor. For example, a Feature is created through a FeatureType interface, and a FeatureType (metadata) is created through a FeatureTypeFactory interface. When a type's name is not descriptive of its use, we have used "typedef" statements, as in C, to give them more descriptive names.

Since a particular object may implement multiple interfaces, the object constructor should require all input parameters of all factory interfaces for the types it implements.

A Boolean return value is associated to every statement (or list of statements) and is a success flag, returning TRUE if the interface produced an acceptable result. For the sake of consistency, we will assume a NULL return from any interface is the equivalent of a Boolean FALSE, or process failure. Exception handling will be treated separately, and should most likely be left to the facilities of the distributed computing environment. Going back to our example:

```
Point X;
Boolean B;
B = ( X = Point((30, 30), "Lat. Long") );
```

If B is TRUE, then X is a point located at (30,30) in the Lat. Long coordinate system. If B is FALSE, the X is NULL or unchanged if it had been previously assigned. Each statement is a transaction that only has effects if it succeeds.

For interfaces with no parameters, which are often implemented as member data in programming languages, the extra empty parentheses '()' will often be omitted.

3.1.1.1.2 Functional Programming Interfaces

Where possible, we have kept the interfaces functional in that the underlying objects are not required to change in supporting an interface. For example, when we add an element to a list, we create, possibly temporally, a new list. If the intent is to modify the existing list, then an assignment must also be used.

```
ListOfReal L, M;
L = L.AddFirst(1.4);
M = L.AddFirst(1.4);
```

Note that the second statement above does not modify L, while the first does. Different implementations may choose to accomplish the same duality of function differently, but both functions must be possible.

For this reason, not all constructors are listed in the factory interfaces. Protocols from other types or the same type, whose return type is also an OGIS type (as opposed to being a well-known type) are essentially constructors for that returned type.

3.1.1.2 Assertions and Definitions

Assertions will be included, following a type definition. These assertions have the purpose of restricting the semantics of implementations so that they yield the intended result. Assertions simply list Boolean statements that must be true regardless of the type implementation. Because of the nature of some of the more complex assertions, we have used a predicate calculus notation. The following will be used for short hand notations for the most common domains:

- **R** for the real line.

- $\mathbf{R^n}$ for n-dimensional real coordinates, that is, ordered tuples of n real numbers without any metric inferred. $\mathbf{R^n}$ will often be the coordinate domain of a coordinate system with its own metric.
- $\mathbf{E^n}$ for n-dimensional Euclidean space, $\mathbf{R^n}$ with the normal orthogonal (Pythagorean) metric $\|.\|$, which is valid only for 'flat' spaces.
- **Z** for the integers, all negatives, zero and positive.
- $\mathbf{Z^+}$ for the positive integers, starting at 1, 2, 3 … .
- $\mathbf{Z^{o+}}$ for the extended positive integers, starting 0, 1, 2, … .
- For every pair x, y of real numbers in **R**, [x, y] is the closed interval of reals r with $x \le r \le y$.
- For every pair x, y of real numbers in **R**, (x, y) is the open interval of reals r with $x < r < y$.
- $\mathbf{D^n}$ is the n - disk or solid sphere in $\mathbf{E^n}$, the set of $p \in \mathbf{E^n}$ where $\| p \| \le 1$.
- $\mathbf{S^n}$ is the hollow n - sphere in $\mathbf{E^{n+1}}$, the set of $p \in \mathbf{E^{n+1}}$ where $\| p \| = 1$. The boundary of $\mathbf{D^{n+1}}$.

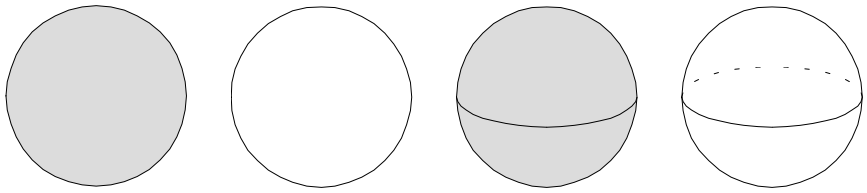


Figure 3-1 Low Dimension Discs and Spheres

The following symbols will be used in predicate calculus statements:

- $\forall$ - for every, the universal quantifier;
  used to indicate that some statement is true regardless of the value of some element variable.
- $\exists$ - there exists, the existential quantifier;
  used to indicate the existence of some element, usually with some property.
- $\neg$ - not, logical negation;
  used to indicate that the statement that follows is false.
- $\wedge$ - logical "and", Boolean intersection;
  used to create compound statements that are true if and only if both elements of the statement are true.
- $\vee$ - logical inclusive "or", Boolean union;
  used to create compound statements that are true if either one of the elements of the statement is true.
- $\bigcup$ or $\cup$ - set union;
  the larger symbol is usually subscripted to indicate that a sequence of sets is being combined,
  the smaller symbol is used as a binary operator between two sets; an element is in the union of sets if it is in at least one of the sets.

- $\bigcap$ or $\cap$ - set intersection;
  the larger symbol is usually subscripted to indicate that a sequence of sets is being intersected,
  the smaller symbol is used as a binary operator between two sets; an element is in the intersection of sets if it is in both of the sets.
- \ - set difference
  The difference of two sets A \ B is all elements in the first but not the second.
- $\oplus$ - symmetric difference (for sets).
  The symmetric difference of two sets A$\oplus$ B is all elements in one but not both sets.
- $\varnothing$ - the empty set, the unique set having no elements.
- $\rightarrow$ - mapping or function;
  used to indicate the mapping of one set to another.
- $\in$ ($\notin$)- element of (not an element of);
  use to indicate that an element is (or is not) a member of a set.
- $\Rightarrow$ ($\Leftrightarrow$) - implies (is logically equivalent to);
  used to indicate that one proposition implies another (or that two propositions each imply each other, making them equivalent statements). 'P $\Rightarrow$ Q' is the same as '($\neg$ P) $\vee$ Q.'
- :: -resolution operator
  <type_name>::<interface_name> refers to the interface name defined in the type specified. '::' can be used either where ambiguity exists, or just as a reminder from where the interface is derived.

3.1.2 Type Definitions

3.1.2.1 Preliminary Definitions

OGIS will be implemented in many different environments. Each environment will have its own set of definitions and underlying programming paradigm, for which the OGIS model will have at least one implementation. To keep the conceptual model generic, we will use the following generic terms:

Type	a well defined set of logical programming interfaces (protocols).
Type Factory	a type containing the constructor protocols for another type.
Object	a digital representation of some conceptual phenomena, supporting some set of type interfaces. If an object supports a type, it is often said to be element of that type. An object is an instance of the class for which it is a member
Class	an implementation or collection of similar objects. Member objects of a class share structure and implementation. Subclassing is an implementation specific concept and varies from one object oriented programming language to another.

Table 3-1 Definition of Type, Object and Class

3.1.2.2 Type

We have defined type as a well defined set of logical programming interfaces (protocols). Normally this will be implemented in a programming language as either a type or a class. For example, C++ uses type for primitives such as integer, real, character, and for some more complex structures such as array and character string. It uses classes for more general "programmer defined" types, such as will probably be the case for the various feature and geometry types defined later in this paper. Smalltalk is more consistent and uses class for everything we might define as a type. SQL 2 is consistent in the other direction, forcing the user to only use the types and structures

defined as part of the language. SQL3 takes a more C++ like approach, mixing the SQL 2 primitives with classes called abstract data types (ADTs).

Type is unit of behavior. Type contains no specification of data storage mechanism nor of method implementation. Class specifies how behavior is accomplished. Class contains specification of data storage mechanism and of method implementation. Types define an algebra. Class defines a set of algorithms that implements that algebra. This relationship is diagrammed in Figure 3-2.

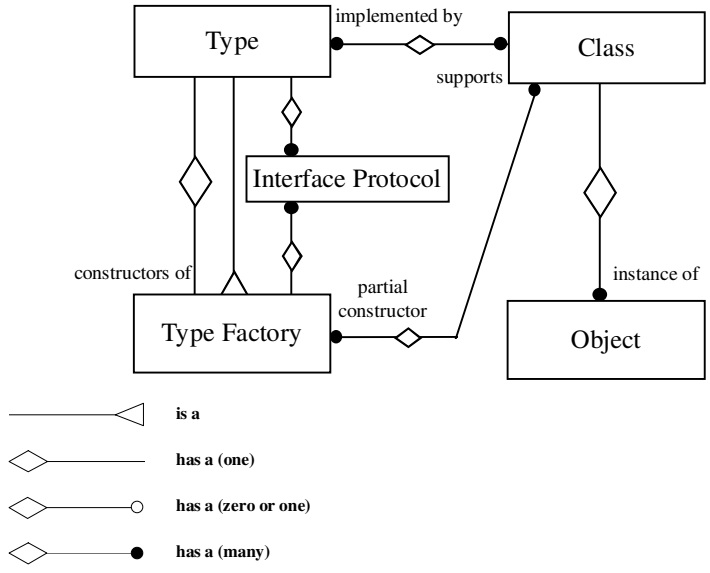


Figure 3-2 Relationship Between Type and Class

All of us learn algebra early on, and learn to manipulate symbols according to rules to solve real problems. Later, in classes for programming languages, we discover that the same techniques allow us to specify the manipulation of variable without reference to the physical structures in which they are stored and manipulated (hence the name FORTRAN for "formulae translation"). We have simply extended this concept to the level of programming language objects. The types described in the OGIS data model establish a behavioral framework, sort of an algebra for geodata, which is mapable onto different implementations.

There are a few terms we will use that are suggestive of the final implementations but which are not meant to be restrictive. One type is a subtype of another if and only if it includes (supports) all the other type's interfaces. In this manner, the types can be seen as sets of interfaces, with set inclusion acting as a reverse order: a subset of interfaces forms a supertype, and a subtype is a superset of interfaces. This does not mean that the object class that implements one type need be a subclass of another class that implements the supertype.

Because of the logical problem of creation (who creates an object?), we introduce the concept of a "type factory" as a holder of the construction protocols. Again, this does not mean that the programming language need have a separate class for factories. C++ uses variable declaration and the "new" operator as "factories" for a class, but includes the constructor in the protocols for the class being constructed. While this is a logical inconsistency, nothing in OGIS is meant to "disable" basic C++ methodology. Thus, the type factory in C++ is supported by the same classes that support the type. Other languages bootstrap the existence of objects in different manners. In addition to the type factory, other interfaces may contain constructors for the type in question (mainly type

conversion functions). The type factory is unusual because it does not otherwise have or require an associated object.

3.1.2.3 Well Known Types

In describing a model behaviorally, the one question that always arises in the mind of the practical programmer is "where do these things come from?" Since every object must be buildable somewhere, we define a minimal set of types that are special in that everyone can agree on what they are and how they are represented. Thus, a well known type has one and only one object implementation per environment (programming language, OS, source GIS, database, etc.) and a well defined transfer mechanism between environments. The obvious types include anything in the programming language's basic arsenal. Integers (16, 32 and 64 bit IEEE varieties), real numbers (probably 32 and 64 bit IEEE varieties), character and character strings will most definitely be well known types. Additional types that we will define as well known types in the case of OGIS will include simple coordinates in at least one basic geographic system (latitude, longitude, WGS 84 is a candidate), lines, line strings, polygons (with holes as needed), and polyhedral solids. As we will see below, other types should use these well known types (either directly or by recursion) in their construction. In this sense, 'well known' is a relative term. It describes the basic transfer mechanisms between a client and a server. If the client and the server have a mechanism in place to determine a level of commonality (sometimes called a 'negotiation'), then the set of well known types can shift to that level.

Data has to move between cooperating processes, both within and between environments. Processes built on the same underlying data storage or data environment can share objects directly. Other process pairs need to depend on a common data language type. To support this, every type should include a constructor which uses only well known types as inputs and a "copy" function that returns those same types (enables transfer). Processes wishing to communicate establish a common set of well known types (which must include all OGIS well known types, but can go further) based on "negotiation." Data can then be transferred using the "copy" and construct protocols for corresponding higher level types.

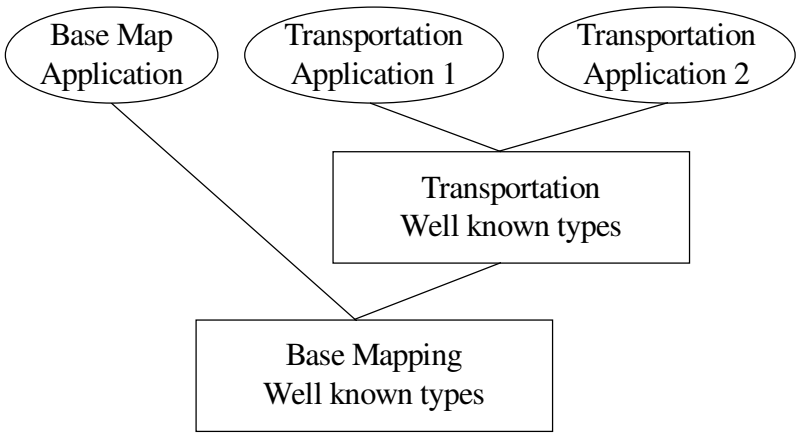


Figure 3-3 An example of application use of well known type hierarchies

Many types in OGIS, such as feature, appear to be structurally based, posing protocols for "metadata" and "attributes." These types are designed to enable some level of system interoperability through data transfer. For example, if we investigate the negotiation between a simple base mapper product and one that does network analysis

for transportation, we quickly see that the behavioral model that the two applications have of the road network is quite different. The transportation product probably includes behavior for the interaction of roads at intersections, while the base mapping product, if it deals with the intersections at all, sees them as simple points. The interoperation of these two products uses a simpler view (the type of the feature in the base mapper) to connect the more cogent network roads to the simpler base mapping roads. If the two products were both cogent, but in different manners (such as a cadastral mapper and a transportation mapper), they may agree on the simple base mapper types as their interoperability ***lingua franca***. This sort of situation is diagrammed in Figure 3-3.

The copy of one type to another through a "lesser" interface is called "forgetful" in the sense that it ignores data and behavior that are not relevant to the target application. In the next section, we will establish the baseline for well known geodata types as the "feature" and the "coverage."

3.1.2.4 Objects and Classes

Objects are where the real geo-processing happens, in that they are the instances of classes that implement the types we are describing. Since OGIS must enable "objects" to exist in many environments, we must make very few assumptions about the underlying object system. These assumptions are:

- Objects are instances of some implementation class. Thus, an object's interfaces are defined by the class in which it is implemented.
- A class implements some number of types.
- The class constructors include the type factory protocols. That is, if a type is implemented, its factory must also be implemented.
- The mapping from type interfaces to class interfaces is straight forward, consistent with the well known OGIS types, and is handled totally in software. Thus, details like functional interfaces, are usually hidden from the user of a particular environment.
- The type hierarchy and the subclass hierarchy are independent, and accounted for in the mapping between class and type.

3.1.2.5 Some Common Types

In most programming languages, there are some basic types that are commonly used. In the specification, we need to have generic interface definitions for some of the these. The subtyping tree for these common types in given in Figure 3-4.

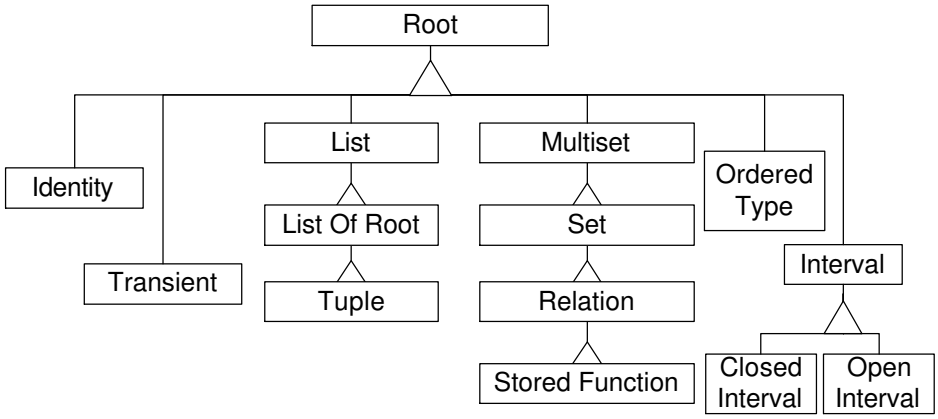


Figure 3-4 Subtype Relations among the Common Types

"Type" means "interface enumerator." Any system that will support OGIS will have some set of existing types that are well known within that system. For that reason, we will not redefine most of them, and use some common names as place holders:

- Type: any type defined either previously or later (see templates)
- Integer: any type used to represent all or some subset of integers {..., -2, -1, 0, 1, 2, 3, 4, ...} and their usual operations
- Number: any type used to represent a numerical value, including integer and real types.

Whenever a type name is used that parallels a common programming concept, then the same name will be used.

3.1.2.5.1 The Root Type

Root is a special type, from which all other types logically, but usually not interestingly, derive through subtype. We include it here as a convenience, as we will use untyped references later (pointers or references to Root). Because of its use as an untyped reference, the Root interface requires a hook so that a user or application programmer can regain information on its true types. SetOfType is defined below.

```
Interface Root
{
SetOfType Type();
}
```

- Type() returns a set of interface types that the instanciated object supports.
- All other types will be subtyped from Root, either directly or by recursion. This means that all type interfaces will have a Type() function.

3.1.2.5.1.1 Interface, Type and Class: Basic Building Blocks of Metadata

The next two types and the Root type, form the basic metadata interfaces (see Figure 3-5). 'Interface' is a name, a set of input types, a set of output types (which is most often a singleton data type and a 'error mode' Boolean). Type is a set of interfaces that use that type as the key first parameter. Tupletype (**Error! Reference source not found.**), which defines tuples and stored functions (**Error! Reference source not found.**), and FeatureType (**Error! Reference source not found.**), which defines features (**Error! Reference source not found.**) and coverages (**Error! Reference source not found.**),

act as the basic aggregations. Class is the fundamental unit of implementation, and implements some number of types. The definition of 'Tupletype' and 'SetOf<Type>' are given later in sections **Error! Reference source not found.** and **Error! Reference source not found.** respectively.

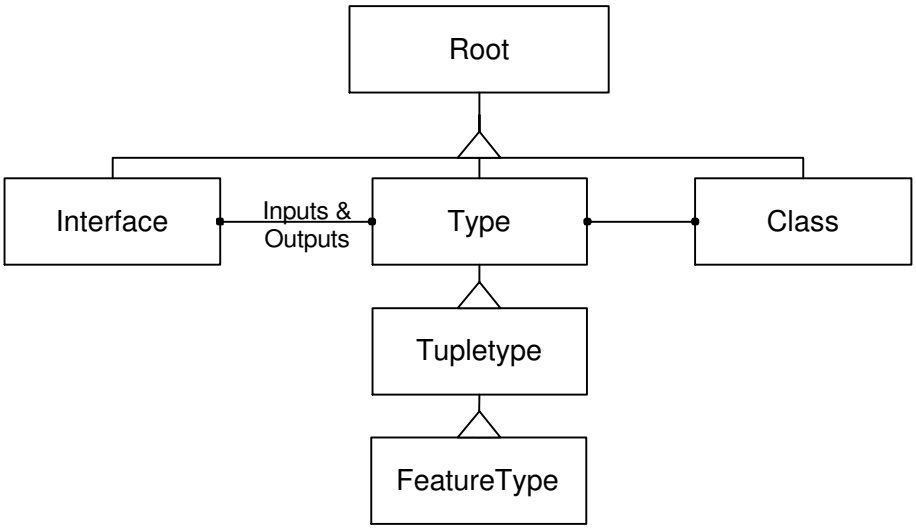


Figure 3-5 Basic Building Blocks of Metadata

```
Interface Interface under Root
{
Tupletype           ParameterList();
Tupletype           Return();
CharacterString     Name();
}
Interface Type under SetOfInterface
{
SetOfInterface      Interfaces()
CharacterString     Name()
SetOfType           Supertypes()
}
Interface Class under SetOfType
{
SetOfRoot           Query(BooleanPredicate);
SetOfType           Interfaces()
CharacterString     Name()
SetOfClass          Superclasses()
}
```

Ed.: Should Type and Class be under Identity or Root instead?

Assertions:

1. All objects (implemented types) will support all interfaces implied by the Type().
   $\forall$ r $\in$ Root [r supports all interfaces in $\bigcup$ r.Type]
2. The first parameter in an interface defined at a type is that type.
   $\forall$ T $\in$ Type, $\forall$ I $\in$ Type.Interface [I.ParameterList[0] = T]

This machinery gives a program enough metadata information from calls to an object's type function, and calls to the interfaces of that type, to construct input and output tuples for any interface supported by that object. In many ways this is equivalent

to the ability in SQL for a program to extract metadata about tables and to then construct queries on those tables. This is important to user driven interoperability because the software invoking an interface does not have to have knowledge of that interface at compile time. This allows programming for interpreters and generic commands as in the flow chart in Figure 3-6.

3.1.2.5.1.2 Root Factory

The only Root constructor that is totally generic is the copy constructor. Thus, the RootFactory is quite simple.

```
Interface RootFactory under Root
{
Root   Root(Root);
}
```

- Root() — copies the existing object to an identical one.

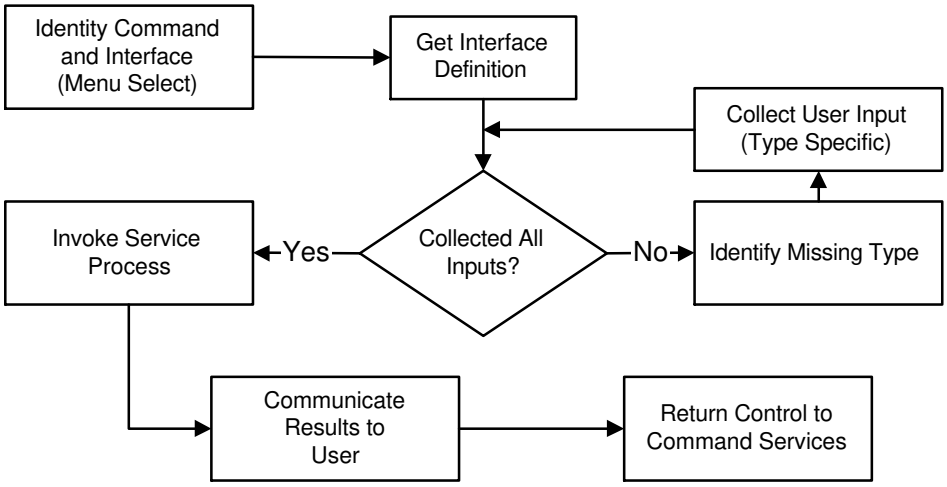


Figure 3-6 Flow of a Generic Command

3.1.2.5.2 Identity, Persistent and Transient Objects

Persistent objects are those known to the system by some mechanism other than by their values. The usual mechanism in a database is to assign one of the attributes (the 'key') unique values for each particular instance of a persistent object. We assume that KeyValue will be a well-known type in the final implementation of the specification. It will probably include information as to the source of the data (the database in which the 'official' copy is held) and an internal identifier specific to that source.

Transient objects are known only by their value. Most systems have a garbage collection routine that deletes transient objects when they are no longer needed. In OGIS, with its cross application references to objects, there must be a mechanism for tracking when no further 'external' reference to a transient object exists. Whether or not the same mechanism is used to internally track references to objects is strictly an implementation issue, and is left to the design of the data servers.

```
Typedef ReferenceValue    Integer
Typedef KeyValue    CharacterString
Typedef TimeStamp         Time
```

3.1.2.5.2.1 Identity

```
Interface Identity under Root
{
KeyValue      WhoAmI()
Boolean       EqualTo(Identity)
Boolean       EqualTo(KeyValue)
TimeStamp     BeginTime()
TimeStamp     EndTime()
}
```

- WhoAmI() — returns the identity of the object. The precise format for a key is determined by the storage system of the object. KeyValue is typed as a character string to allow a 'well-known' exchange mechanism and to maintain with the most likely form for query language interchanges.
- EqualTo() — test for equality of this object to another having the passed identity or KeyValue. This test equality by identity. Other versions of '=' test for equality by value.
- BeginTime(), EndTime() — give the range of time for which the object with this identity actually reflects the real world. Using this mechanism, databases can 'version' the same real world entity as separate objects, and still maintain the same key value. NULL values for either of these returns indicated unlimited time. 'BeginTime = NULL' means the beginning of history (as the database knows it) or that there is no other prior 'Identity.' 'EndTime = NULL' means valid for up to the current moment, and possibly beyond. More precisely, 'EndTime = Null' means that there is no other Identity for this object having a later existence than this one.
- It is assumed that destructors for objects with identity will be called when these objects are no longer required to be in the database or in the process space. Objects with identity are not subject to 'garbage collection.'

Assertions:

1. Two persistent objects are 'identical' if and only if their key values are identical.
   $\forall$ x, y $\in$ Identity [x.KeyValue = y.KeyValue $\Leftrightarrow$ x = y]

3.1.2.5.2.2 Transient

```
Interface Transient under Root
{
ReferenceValue AddReference();
ReferenceValue DeleteReference();
ReferenceValue Reference()
}
```

- AddReference() — is called each time the value of this object is held by some process.
- DeleteReference() — is called each time the value of this object is no longer needed by some process.
- Reference() — returns the current accumulated value of the reference count. If this is 0 (zero, or NULL), then the transient object is no longer being referenced by any active process, and can be deleted, and the space reclaimed for other objects (garbage collection).

Since the identity of a persistent object must be assigned by the database system, and the transient's reference count is always 1 at construction time, no additional factory parameters are needed in their construction.

3.1.2.5.3 Template

We will often define a broad range of related types using a "template." A template will carry one or more variables (usually type names) set off in angle brackets "< >" which are allowed to range over any acceptable value. Restrictions on values will be given with a template (see assertions below). For example, the next section gives a template for lists of a given type as ListOf<Type>. By substituting Number where Type appears, we get an interface definition ListOfNumber, or a list of numbers. We similarly get ListOfRoot, ListOfInteger, and even ListOfType.

3.1.2.5.4 List or Array

A list (or array) is an ordered set of objects (usually of a common base type, but not always);

```
Interface template ListOf<Type> under Root
{
<Type>              [](Integer);
Integer             Length();
<Type>              Last();
<Type>              First();
ListOf<Type>        Rest();
ListOf<Type>        AddLast(<Type>);
ListOf<Type>        AddFirst(<Type>);
ListOf<Type>        Add(<Type>, Integer);
ListOf<Type> Delete(<Type>);
ListOf<Type> Delete(Integer);
Integer             InPosition(<Type>);
SetOf<Type>  SetOf();
MultiSetOf<Type>    MultiSetOf();
}
```

- []()—Return the i^{th} element of the list. Lists are indexed from 0 an in C (as we will see in the assertions that follow). The usually notation for "[](i)" is to drop the redundant parameter list separators and simply write [i].
- Length()—Return the length of the list.
- Last()—Return the last element in the list.
- First()—Return the first element in the list.
- Rest()—Return the "tail" or "cdr" of the list. In other words, all but the first element in the list.
- AddLast()—Add the given item to the beginning of the list. Lengthen the list by 1.
- AddFirst()—Add the given item to the end of the list. Lengthen the list by 1.
- Add() — Add the given item to the list in the position indicated. Shift the remaining elements of the list upward, lengthening the list by 1.
- Delete(<Type>)—Delete the given item from the list. If an item is in the list more than once, the deletion is of the first instance. The index is collapsed
- Delete(Integer) — Delete the item at the given offset from the list. The index is collapsed
- InPosition()—Return the index of the given item in the list. A NULL return indicates that the given item is not in the list. If an item is in the list more than once, the index returned is of the first instance of that item.
- SetOf(), MultiSetOf() — returns the list to a set or a multiset respectively. The SetOf operation will remove duplicates in the list, the MultiSetOf of operation will not.

Assertions:

1. When an element is added to the beginning of a list the length is increased by 1.
   $\forall$ t $\in$ Type $\forall$ List $\in$ ListOfType [List.AddFirst(t).Length = 1 + List.Length]
2. After an element is added to the beginning of a list, the first element will be the element that was added.
   $\forall$ t $\in$ Type $\forall$ List $\in$ ListOfType [t = List.AddFirst(t).First]
3. When an element is added to the end of a list, the length is increased by 1.
   $\forall$ t $\in$ Type $\forall$ List $\in$ ListOfType [List.AddLast(t).Length = 1 + List.Length()]
4. After an element is added to the end of a list, the last element will be the element that was added.
   $\forall$ t $\in$ Type $\forall$ List $\in$ ListOfType [t = List.AddLast(t).Last]
5. After an element is added to the beginning of a list, the rest of the list will be the list as it was before the addition.
   $\forall$ t $\in$ Type $\forall$ List $\in$ ListOfType [List = List.AddFirst(t).Rest]
6. Lists are indexed consecutively from 0 to the length of the list minus 1.
   $\forall$ i $\in$ **Z,** $\forall$ L $\in$ ListOf [(i < 0) $\Rightarrow$ (L[i] = NULL)]
   $\forall$ i $\in$ **Z,** $\forall$ L $\in$ ListOf [((L.Length - 1) < i)) $\Rightarrow$ (L[i] = NULL)]
7. The rest interface does not return the first element, but it returns all of the other elements of the list.
   $\forall$ i $\in$ **Z,** $\forall$ L $\in$ List [(0 < i) $\Rightarrow$ (L[i] = L.Rest[i-1])]
8. When an element is deleted from the list the length is decreased by 1.
   $\forall$ t $\in$ Type, $\forall$ List $\in$ ListOfType [List.Delete(t).Length = List.Length - 1]

3.1.2.5.4.1 List Factory

```
Interface ListOf<Type>Factory under Root
{
ListOf<Type> ListOf(<Type>);
ListOf<Type> ListOf(ListOf<Type>, <Type>);
ListOf<Type> ListOf(<Type>, ListOf<Type>);
ListOf<Type> ListOf(ListOf<Type>, ListOf<Type>);
}
```

- ListOf() — Constructs a list of type '<Type>', returning the list. If a NULL is passed for type, then an empty (length 0) list is produced.
- The two protocols that mix a list and a singleton add the singleton to the beginning or end of list as would be suggested in the parameter list.
- The protocol with two lists does a concatenation of those lists.

Assertions:

1. When a list is constructed from a single, non-NULL element, its length is 1.
   $\forall$ t $\in$ Type [ListOf(t).Length = 1 $\wedge$ ListOf(t).First = t]
   $\forall$ L $\in$ List [L[0] = L.First]
   $\forall$ L $\in$ List [L[L.Length -1] = L.Last]

For convenience, we will often write a list in a LISP like "()" notation. This is consistent with the groupings within the interface definitions or assertions which are lists of statements. It is also consistent with the use of "()" for the passing of parameters to functions, since the parameters are set off with "( )" as list of passed parameters.

3.1.2.5.5 Multiset (or Bag)

A multiset is like an array of objects except order does not matter. As in an array, a multiset can contain an object more than once.

```
Interface MultiSetOf<Type> under Root
{
Boolean             Finite();
ListOf<Type>        MakeList();
Integer             Size();

\\ obtain an iterator
MultiSetOfIterator<Type>  Iterator();

\\ The following functions also appear in the factory
MultiSetOf<Type>    Delete(<Type>);
MultiSetOf<Type>    Add(<Type>);

\\ Containment information
Integer             Count(<Type>);
Boolean             IsIn(<Type>);
Boolean             IsSubset(MultiSetOf<Type>);
Boolean             Equals(MultiSetOf<Type>);
}
```

- Finite() — Returns true if the multiset has a finite number of elements, false otherwise. Most implementations that actually store the elements of a set or multiset will by necessity be finite. Others that are, for example, stored as spatial objects or as Boolean operations capable of testing an element, can be infinite. Infinite sets or multi-sets will not support the interfaces that assume 'finite' such as List, Size, Rest, InitIterator, etc.
- MakeList() — Construct a list from this multiset. The list will have length Size() and have multiple entries for items in the multiset multiple times.
- Size() — Return the size of the multiset (that is, the number of elements).
- Iterator() — Return an iterator over the multiset.
- Add() — Insert an element into the multiset (or increments the count if the element is already in the multi-set). Remember, there are no guarantees about ordering.
- Delete() — Removes an element from the multiset (or decrements the count if the element is in the multi-set multiple times).
- Count() — Return the number of times the given element appears in the multiset. The Boolean equal '=' used to determine the uniqueness of items in a multiset or set is the one defined for the type.
- IsIn() — Is this given element in the MultiSetOf? The equivalent of '$\in$'.
- IsSubset() — Is this given multi-set a subset of MultiSetOf? The equivalent of '$\subset$'.
- Equals() — Is this given multi-set equal to the MultiSetOf? The equivalent of '='.

Assertions:

1. 'Element of' and 'count' are consistent
   $\forall$ t $\in$ Type, $\forall$ M $\in$ MultiSetOfType [M.IsIn(t) $\Leftrightarrow$ t $\in$ M $\Leftrightarrow$ M.Count(t) > 0]
2. 'Subset' means that it has smaller or equal counts for everything in Type. 'Subset' and 'element of' are consistent.
   M.IsSubset(L) $\Leftrightarrow$ L $\subset$ M $\Leftrightarrow$ [$\forall$ t $\in$ Type, t $\in$ L $\Rightarrow$ t $\in$ M $\wedge$ L.Count(t) $\leq$ M.Count(t)]

3.1.2.5.5.1 MultiSetOf Factory

```
Interface MultiSetOf<Type>Factory under Root
{
MultiSetOf<Type> MultiSetOf(<Type>);
MultiSetOf<Type> MultiSetOf(ListOf<Type>);
MultiSetOf<Type> MultiSetOf(Boolean<Type>);
MultiSetOf<Type> MultiSetOf(MultiSetOf<Type>, <Type>);
}
```

- MultiSetOf() — Construct a multi-set of Type, returning the multi-set. As in list, a NULL parameter may result in an empty multi-set.

A Boolean<Type> is a Boolean expression with one free variable. Elements of the returned multi-set are all of those that satisfy the Boolean. Multiple entries may occur if the Boolean can be interpreted as having a 'group by' style 'count.' For example, requesting all roads that cross rivers may return a multi-set where the count for each road indicates the number of times the road crossed some river.

```
Interface MultiSetOf<Type>Operator under Root
{
MultiSetOf<Type>    Union(MultiSetOf<Type>, MultiSetOf<Type>);
MultiSetOf<Type>    Intersection(MultiSetOf<Type>,
                          MultiSetOf<Type>);
MultiSetOf<Type>    SymmetricDifference(
                          MultiSetOf<Type>, MultiSetOf<Type>);
MultiSetOf<Type>    Difference(MultiSetOf<Type>,
                          MultiSetOf<Type>);
}
```

- Union() ... Difference() — Perform the named operations on multi-sets.

3.1.2.5.5.2 MultiSetOfIterator

```
      Interface MultiSetOfIterator<Type> under Root
      {
      MultiSetOfIterator<Type>  Initialize();
      <Type>                    GetElement();
      MultiSetOfIterator<Type>  Rest();
      Boolean                   AtEnd();
      }
```

- Initialize()—returns a iterator over the multiset that has the potential to iterate over all the elements in the multiset. This has the same effect as calling the 'Iterator()' method for MultiSetOf'.
- GetElement()—will return the same element no matter how many times it is called for a particular iterator. No ordering of the MultiSetOf is implied.
- Rest()—Return a new iterator that will iterate over all elements in the multiset that the previous iterator would iterator over excluding the element GetElement() returned for the previous iterator.
- AtEnd()—The current iterator has just the one element that GetElement() returns to iterate over. If true a call to Rest() would be undefined.

3.1.2.5.6 SetOf

A Set is a group of objects, similar to List, except that order does not matter. It is similar to a MultiSetOf, except that each object is unique.

```
Interface SetOf<Type> under MultiSetOf<Type>
{\\ Note: no new interfaces, just a new assertion
}
```

3.1.2.5.6.1 Set Of Factory

```
Interface SetOf<Type>Factory under MultiSetOf<Type>Factory
{
SetOf<Type> SetOf(MultiSetOf<Type>);
}
```

- SetOf()—Construct a set of Type, returning the set. Again, if a NULL is passed in, an empty set will result.
- MultiSetOf protocols are valid for Sets, since the above protocol is a 'cast' operator.

```
Interface SetOf<Type>Operator under MultiSetOf<Type>Operator
{
SetOf<Type>  Union(SetOf<Type>, SetOf<Type>);
SetOf<Type>  Intersection(SetOf<Type>, SetOf<Type>);
SetOf<Type>  SymmetricDifference(SetOf<Type>, SetOf<Type>);
SetOf<Type>  Difference(SetOf<Type>, SetOf<Type>);
}
```

- Union() ... Difference() — Perform the named operations on sets.

Assertions:

1. Each element in the set is unique.
   $\forall t \in Type, \forall S \in SetOfType\ [\ t \in S\ \Rightarrow S.Count(t) = 1\ ]$
2. Adding a particular instance to a set twice has no effect.
   $\forall t \in Type, \forall S \in SetOfType\ [\ t \in S\ \Leftrightarrow S = Set(S, t)]$

Ed.: Assertions about Set and MultiSetOf will also contain all of the applicable set theory, that is, the interfaces act like sets.

3.1.2.5.7 Tuple

A tuple is a list where each element could be of a different type. Since List is a typed template, to use interface inheritance, we go back to ListOfRoot that are untyped pointers. To type the offsets in the list, the tuple must be able to return a list of types (type ListOfType). The columns of the tuple are usually associated to names.

```
Typedef Name CharacterString;

Interface Tupletype under Type
{
ListOfType          MetaData();
ListOfName          Name();
Tupletype           SetName(ListOfName);
}

Interface Tuple under ListOfRoot
{
Tupletype           Type();
Root                [](Name);
Tuple               SetName(ListOfName);
}
```

- Metadata() — lists all the types in order of the offsets
- Name() — lists the names of the offsets; this should be a unique index into the tuple
- [] — Returns the value, and is inherited from the 'list of root' interface. The type of the return should be consistent with the metadata.
- SetName — renames all of the offsets of the tuple. The interfaces in tuple and tuple type are consistent, in that a set name returns a tuple whose type would be the result of a set name on the type of the original tuple.

Assertions:

1. The name index is unique.
   $\forall$ T $\in$ Tupletype, $\forall$ i, j $\in$ $\mathbf{Z^{0+}}$, i, j < T.Length
   (i = j $\Leftrightarrow$ T.Name[i] = T.Name[j])
2. The i^{th} element of the tuple is a subtype of the i^{th} type listed in the metadata
   $\forall$ T $\in$ Tuple $\forall$ i $\in$ $\mathbf{Z^{0+}}$, i < T.Length [T.Metadata[i] $\subset$ T[i].Type]
3. The numerical index and the name index are equivalent.
   $\forall$ T $\in$ Tuple, $\forall$ i $\in$ $\mathbf{Z^{0+}}$, i < T.Length $\Rightarrow$ T [T.Name[i]] = T[i]

3.1.2.5.7.1 Tuple Factory

```
Interface TupleFactory under Root
{
Tuple        Tuple(Tupletype, ListOfRoot);
}
```

3.1.2.5.7.2 Tupletype Factory

```
Interface TupleTypeFactory under Root
{
Tupletype  Tupletype(Name, ListOfType, ListOfName);
}
```

3.1.2.5.8 Relations

One way to express relations is as sets of tuples:

```
Interface Relation under SetOfTuple, Tupletype
{
Tupletype    Type();
Name         Name()
}
```

- Type() — returns the metadata of the relation, just as The metadata as represented for the Relation must be consistent with the metadata for each of the tuples.
- Tupletype interfaces are inherited through delegation to the Type().
- The name list of all the tuples in the relation are the same as that of the relation.

Ed: There is no reason for this, but there seems to be no good reason why not since doing a set type on each tuple as it comes in should accomplish the same effect.

Assertions:

1. Each tuple in a relation has metadata, and names consistent with that of the relation.
   $\forall$ R $\in$ Relation, $\forall$ t $\in$ Tuple [t $\in$ R $\Rightarrow$ R.Metadata $\subset$ t.Metadata $\wedge$ t.Name = R.Name]

3.1.2.5.8.1 Relation Factory

For finite relations, the factory is listed below. Not all relations are finite, as for example the '$\in$' ('element of') relation between points and geometry objects. These types of relations would be able to satisfy the 'relation' interface, but would require special factory interfaces.

```
Interface RelationFactory under Root
{
Relation Relation(Name, TupleType, SetofTuple);
```

```
}
```

3.1.2.5.9 Ordered Type

In several places in the specification, we will have to assume that a type is ordered, such as the integers or reals, under an order function that acts like "<." Since we are only looking at existing ordered types, no factory specification is needed.

```
Interface OrderedType under Root
{
Boolean LessThan(OrderedType);
Boolean LessThanOrEqual(OrderedType);
Boolean IsEqual(OrderedType);
Boolean GreaterThan(OrderedType);
Boolean GreaterThanOrEqual(OrderedType);
}
```

- LessThan() or '<' — This Boolean interface will return TRUE if the given OrderedType is less than the type instance.
- LessThanOrEqual() '≤' — This Boolean interface will return TRUE if the given OrderedType is less than or equal the type instance.
- '>,' and '≥' are similarly defined as the inverses of '<' and '≤.'

Assertions:

When there is little chance for confusion, we will use "x < y" for "x.LessThan(y)."

1. Two values have one and only one relationship to one another
   $\forall$ x, y $\in$ OrderedType $[\neg (x = y) \Leftrightarrow ((x \prec y) \vee (y \prec x))]$
   $\forall$ x, y $\in$ OrderedType $[\neg ((x \prec y) \wedge (y \prec x))]$
2. The ordering is transitive
   $\forall$ x, y $\in$ OrderedType $[((x \prec y) \wedge (y \prec z)) \Rightarrow (x \prec z)]$
3. The three comparisons are related in the way the names imply.
   $\forall$ x, y $\in$ OrderedType $[(x \leq y) \Leftrightarrow ((x = y) \vee (x \prec y))]$

3.1.2.5.10 Stored Function

Actually, the 'stored' adjective is of no meaning, since the interface does not specify how a thing is to be done. The adjective is simply to remind us that the functions defined here are not implemented code but a type.

```
Interface template StoredFunction<DomainType><RangeType>
            under Relation
{\\ value is NULL if the parameter is not in domain
<RangeType>         Value(<DomainType>);
SetOf<DomainType>   Domain()
}
```

Assertions:

1. The function is defined only for its domain
   $\forall$ f $\in$ StoredFunction $[t \notin f.Domain \Rightarrow f(t) = NULL]$
2. As a relation, the metadata is consistent with the function definition
   $\forall$f $\in$ StoredFunction [f.Relation::Metadata = (DomainType, RangeType)]
   $\forall$f $\in$ StoredFunction [f.Relation::Name = (Domain, Range)]
3. The relation tuple and the value function are consistent.
   $\forall$F $\in$ StoredFunction, $\forall$ t $\in$ Tuple, [t $\in$ (Relation) F $\Rightarrow$ F(t[Domain]) = t[Range]]

Stored function interfaces will be placed on most of what we consider to be 'coverages.' For example, a digital terrain model, such as a TIN, grid, or Vornoi tesselation can all be thought of as mapping points on a map (2D points) to those same points with elevation added (3D points).

3.1.2.5.10.1 Stored Function Factory for functions with finite domains

Construtors for stored functions will depend heavily on the storage mechanism, so there is no generic stored function factory. On the other hand, many of the functions used in geo-processing have finite domains, and thus can be defined as a set of tuples. Thus the stored function factory is the relation factory.

```
Interface template StoredFunction<DomainType><RangeType>Factory
              under RelationFactory
{}
```

3.2 Normative

The following section is the normative portion of the OGIS.

3.2.1 The Spatial and Temporal Templates and Interfaces

In this section, we outline some basic interfaces for spatial and temporal primitives. It is not the intent to specify implementation, but a basic level of interface that would allow systems to build their own internal representations. We begin with some basic definitions to lay out abstract parameter spaces. The idea is to define space and time by its coordinate nature without specifying the specific coordinate system. This works especially well with time which is inherently 1 dimensional. We begin by defining what a "real" interval is.

Once we have a type that admits an ordering, we can define interval to be all values in the type that lie in the order between the two endpoints. Here we are defining closed intervals, which is what we normally associate to 'lines.'

3.2.1.1 Interval

```
Interface template Interval<OrderedType> under Root
{
<OrderedType>       Low();
<OrderedType>       High();
Boolean             IsIn(<OrderedType>);
}
```

- Low() — This interface returns the ordered type value that represents the low end of the interval (as determined by the interface of OrderedType).
- High() — This interface returns the ordered type value that represents the high end of the interval (as determined by the interface of OrderedType).
- IsIn() — This Boolean interface returns TRUE if the given ordered type is in the interval. Again, this the element of operator '∈.'

3.2.1.1.1 Interval Factory

```
Interface template Interval<OrderedType>Factory under Root
{
```

```
Interval<OrderedType>
        Interval<OrderedType>(<OrderedType>, <OrderedType>);
}
```

- Interval()—Constructor for interval types.

Assertions:

1. If an interval is constructed with t as the low, then Low()invoked on the result will return t.
   $\forall$ t, s $\in$ OrderedType [t = Low(IntervalType(t, s))]
2. If an interval is constructed with s as the high, then High() invoked on the result will return s.
   $\forall$ t, s $\in$ OrderedType [s = High(IntervalType(t, s))]

3.2.1.1.2 Closed Interval

```
Interface template ClosedInterval<OrderedType> under Interval
{
}
```

Assertions:

1. The first argument must always be ordered "less than" the second for a non-empty interval.
   $\forall$ t, s $\in$ OrderedType [(t > s) $\Rightarrow$ ($\varnothing$ = ClosedIntervalType(t, s))]
2. A value is in the interval if and only if it lies between the low and high value.
   $\forall$ t, s, x $\in$ OrderedType [(t $\leq$ x) $\wedge$ (x $\leq$ s)) $\Leftrightarrow$ (x $\in$ ClosedInterval(t, s))]

3.2.1.1.3 Open Interval

An open interval is defined similarly.

```
Interface template OpenInterval<OrderedType> under Interval
{
}
```

3.2.1.1.4 Open Interval Factory

```
Interface template OpenInterval<OrderedType>Factory under Root
{
OpenInterval<OrderedType>
        OpenInterval<OrderedType>(<OrderedType>, <OrderedType>);
}
```

- OpenInterval()—Constructor for interval types.

Assertions:

1. The first argument must always be ordered "less than" the second for a non-empty interval.
   $\forall$ t, s $\in$ OrderedType [(t $\geq$ s) $\Rightarrow$ ($\varnothing$ = OpenIntervalType(t, s))]
2. A value is in the interval if and only if it lies between the low and high value.
   $\forall$ t, s, x $\in$ OrderedType [(t < x) $\wedge$ (x < s)) $\Leftrightarrow$ (x $\in$ OpenInterval(t, s))]

3.2.1.1.5 Mixed Intervals

It is possible to define half open, half closed intervals such as (t, s] = {x | t < x $\leq$ s}.

```
Interface template OpenClosedInterval<OrderedType> under Interval
{
}
Interface template ClosedOpenInterval<OrderedType> under Interval
```

```
{
}
```

3.2.1.2 Sorted Lists and Sorted Lists of Intervals

Most of the protocols for various forms of sorted lists do not change from that of ordinary lists.

3.2.1.2.1 Sorted Lists

```
Interface SortedListOf<OrderedType> under ListOf<OrderedType>
{
}
```

Assertions:

1. Items in a sorted list are sorted.
   $\forall$ L $\in$ SortedList $\forall$ i, j $\in$ L.Domain [i < j $\Rightarrow$ L[i] $\leq$ L[j]]

3.2.1.2.2 Sorted Lists Factory

```
Interface SortedListOf<OrderedType>Factory under
        ListOf<OrderedType>Factory
{
SortedListOf<OrderedType>
        SortedListOf<OrderedType>(MultiSetOf<OrderedType>);
}
```

3.2.1.2.3 Sorted Lists of Intervals

```
Interface template SortedListOfInterval<OrderedType> under
        ListOfClosedInterval<OrderedType>
{
<OrderedType>       Low();
<OrderedType>       High();
SortedListOfInterval<OrderedType>
              IsInIntervals(<OrderedType>);
Boolean             IsIn(<OrderedType>);
}
```

Assertions:

1. The smallest element is the smallest of the start points of the included intervals.
   $\forall$ L $\in$ SortedListOfClosedIntervalType
   [L.Low = Minimum{ L[i].Low | i $\in$ L.ListOf::Domain}]
2. The largest element is the largest of the end points of the included intervals.
   $\forall$ L $\in$ SortedListOfClosedIntervalType
   [L.High = Maximum{ L[i]High | i $\in$ L.ListOf::Domain}]
3. The intervals are, in fact, sorted by their position as sets.
   $\forall$ L $\in$ SortedListOfClosedIntervalType, $\forall$ i, j $\in$ L.ListOf::Domain
   [(i < j) $\Rightarrow$ (L[i].High $\leq$ L[j].Low)]
4. A sorted list is sorted.
   $\forall$ L $\in$ SortedListOfType, $\forall$ i, j $\in$ L.ListOf::Domain [(i < j) $\Rightarrow$ L[i] < L[j]]

The only restriction on type above, is that it satisfy a ordering. We could probably live with a partial ordering, but this goes beyond the need. The usual types used will be integer and various precisions of reals. In the following definitions, the type is normally a spatial coordinate, or a Point.

3.2.1.3 Curve of an Arbitrary Type

Curves can be used in many aspects other than geographically related geometry. For example, the graph of the population for a city may be considered a curve in <time, population> 2D space. For this reason, we have defined a generic interface for curve that is valid for any arbitrary type. Most of the time, geodata curves refer to curves of point, so when there is no chance of misunderstanding, we will write CurvePoint as simply Curve.

3.2.1.3.1 Curve of Type

```
Interface template Curve<Type> under StoredFunction
{
<Type>                    Value(Real);
ClosedIntervalReal  Domain();
}
```

- Value(r) — the value of the curve at the input point r. We will write this as C(r) for curve C.
- ClosedIntervalReal Domain()—returns a closed interval for which the curve value function is valid.

Assertions:

1. It the type is in a topological space (where continuity can be defined) the curve is continuous.

   $\forall$ Type $\in$ Topology, $\forall$ C $\in$ CurveType [C is continuous]

3.2.1.3.2 Multi-curve of Type

Often the parameterization along a curve requires 'resetting.' In most drawing packages, this is accomplished by using complex elements or graphic groups.

```
Interface template MultiCurve<Type> under Curve<Type>
{
ListOfClosedIntervalReal  Domain();
}
```

- ListOfClosedIntervalReal Domain() — returns a list of open and closed intervals for which the curve value function is valid.

3.2.1.3.3 Simple and Closed Curves

```
Interface template SimpleCurve<Type> under Curve<Type>
{
}

Interface template ClosedCurve<Type> under Curve<Type>
{
}

Interface template SimpleMultiCurve<Type> under MultiCurve<Type>
{
}

Interface template ClosedMultiCurve<Type> under MultiCurve<Type>
{
}
```

- Simple curves — do not pass through the same point twice.
- Closed curves — begin and end at the same point.

Assertions:

1. Closed curves close
   ∀ C ∈ ClosedCurveType [C(C.Domain.Low) = C(C.Domain.High)]
2. Simple (nonclosed) curves are one to one.
   ∀ C ∈ SimpleCurveType C ∉ ClosedCurveType ∀ s, t ∈ C.Domain
   [C(s) = C(t) ⇒ s = t]
3. Simple closed curves are one to one, except where they join.
   ∀ C ∈ SimpleCurveType C ∈ ClosedCurveType ∀ s, t ∈ C.Domain - {C.Domain.Low}
   [C(s) = C(t) ⇒ s = t]

3.2.1.3.4 Temporal Domains and Attributes

Temporal versions of any attribute or other value based object type can be expressed as functions of time. Here we assume that the environment has a Time well-known data type.

Ed.: Lets decide to "borrow" the SQL3 time "well-known" types, unless we think we can do it better.

```
Typedef TemporalDomain SortedListOfClosedIntervalTime

Interface template Temporal<Type> under StoredFunctionTime<Type>
{
<Type>              Value(Time);
TemporalDomain      Domain();
}
```

3.2.1.3.5 Spatial attributes

```
Interface template Spatial<Type> under StoredFunctionPoint<Type>
{
<Type>              Value(Point);
SpatialObject       Domain();
}
```

3.2.1.3.6 Spatial Temporal Attributes

Spatial temporal attributes are covered by the two templates above, but we reiterate the interfaces here in their combination.

```
Typedef SpatialTemporalDomain TemporalSpatialObject

Interface template SpatialTemporal<Type> under StoredFunction
{
<Type>                      Value(Time, Point);
SpatialTemporalDomain       Domain();
}

SpatialTemporal<Type>       :== Spatial<Temporal<Type>>
                            :== Temporal<Spatial<Type>>
```

These templates basically establish orthogonal projection functions for time and space.

3.2.2 Geodata Definitions

This section gives the descriptive definitions of features and coverages and their associated attribute types.

Spatial domain	a well-defined subset of space.
Temporal domain	a well-defined subset of time.
Spatial temporal domain	a well-defined subset of space and time..

Table 3-2 Definitions of Domain Types

Domains may be used to describe the location of a feature or the range of applicability of a coverage function. Examples of pure spatial domains are points, lines, curves, surfaces, Solids, and complex domains such as networks, grids, rasters, and triangulated irregular networks (TIN). Spatial domains can be implemented as spatial object types.

A temporal domain consists of some well-defined subset of time, usually a time interval. A full spatial temporal domain can be thought of as a space domain valued function of time.

Feature	representation of a real world entity or an abstraction of the real world. It has a spatial domain, temporal domain, or spatial temporal domainas an attribute.
Coverage	is an association of points within a spatial temporal domain to a value (of a defined type). It is a function from the spatial temporal domain to the type.

Table 3-3 Definition of Feature and Coverage

A feature has an identity that can be recognized through time. It is a specific instance of a feature type, such as the Brooklyn Bridge (a feature whose type is Bridge), Mt. Everest (a feature whose type is Mountain), or the stop sign at the corner of Main and Prospect in Champaign, Illinois (a feature whose type is RoadSign). We will be most often be interested in spatial feature that are associated to spatial temporal domains, called extents or representations.

Attribute descriptor	Data about an attribute (metadata), such as name and type, for instance: 'Elevation, integer,' or 'Deckmaterial, character string,' or 'SpatiaslObject'.
Attribute	A specific instance corresponding to an atribute descriptor, such as 29,495 feet,' opr 'concrete,' or an instance of a spatial object.
Spatial Attribute	An attribute whose value is a function of the point within a spatial domain at which it is measured, such as elevation, soil type, vegetation coverage, color. A coverage whose domain is the extent of the feature. A spatial function.
Temporal attribute	An attribute whose value is a function of time, such as land ownership, land value, illumination, or temperature.
Spatial temporal attribute	A spatial attribute whose value is a function of time, such as land use. A spatial / temporal function.

Table 3-4 Basic Attribute Type Definitions

Since coverages are associated with a spatial temporal domain (the domain of the function) they can be also considered as a type of feature with a distinguished spatial / temporal attribute. The internal representation of the function does not affect the interface definitions (as we will see later).

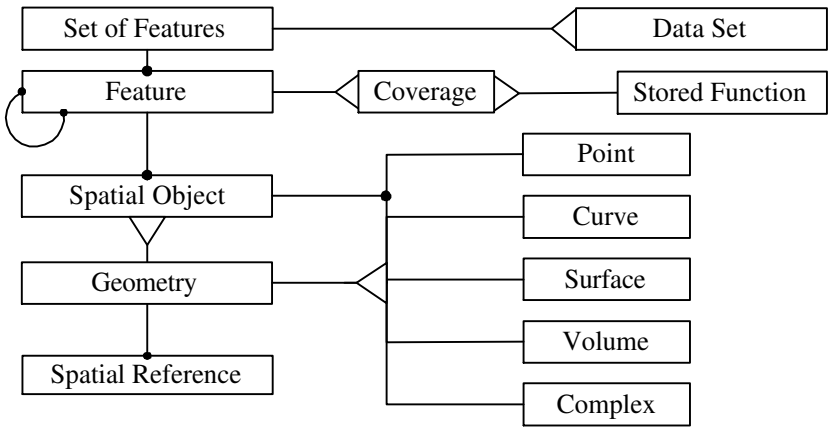


Figure 3-7 Spatial Features, Coverages and their Relationship as Types

3.2.3 Spatial Domains

The most common realizations of a spatial domain are from computer graphics. These are associated with a specific coordinate system. Using this as a model, we see that a spatial domain consists of a geometric (usually coordinate-based) description and a mechanism for mapping the underlying geometry to a real world spatial reference (usually a cartographic projection or coordinate system).

3.2.3.1 Geometry

Every piece of geo-spatial geometry must be able to return its spatial reference so that it can tie itself to its position on the ground, and transform itself to another spatial reference so that it can enable the maximum number of intersystem transfers. Because of this, and the need for 'spatial object' to later share these interfaces, we define a minimal class for geometry in geodata. This class will not be used for "graphics" data that does not represent the actual position of objects. Notice that even though geometry is subtyped from set of points, it will not inherit any constructor (except for in point and point cluster) because of the finite restrictions on sets.

```
Interface Geometry under SetOfPoint
{
SpatialReference    GetSpatialReference();
Geometry            SetSpatialReference(SpatialReference);
Boolean             Equals(Geometry);
Boolean             Locate(Point, Distance);
Boolean             LocateWithin(Geometry);
Boolean             LocateOverlap(Geometry);
Geometry            Boundary();
Boolean             Open();
Boolean             Closed();
Distance            Resolution();
Error               Accuracy();
}
```

- GetSpatialReference() — Returns the spatial reference for the geometry.
- SetSpatialReference() — Returns a new geometry object (usually of the same base type but not always - see line and line string) which has the a different spatial reference.
- Equals() — determines whether two geometries are equal as sets of Points

- Locate()— will retrun whether or not the geometry is within the given distance of the given point
- LocateWithin()—will return whether or not the geometry is within the given geometry
- LocateOverlap()—will return whether or not the geometry overlaps the given geometry
- Boundary() — will return the boundary of the geometry. Since geometry is an abstract class, the precise meaning of boundary can be deferred to its various subclasses.
- Open() and Closed() return whether the object in question contains its boundary (is closed) or is disjoint from it (is open). These are used in topological operators. A NULL return from either of these can imply that the source system does not support this interface.
- Resolution() — Returns a distance expressing the smallest significant linear distance measured in the coordinates of the geometry.
- Accuracy() — Returns a data structure describing the accuracy of the geometry.

Under any spatial reference, a coordinate point is given by an ordered list of numeric values. Here Number is a supertype for all the common types of numbers used as coordinates, usually either reals, integers, or angles.

```
Typedef CoordinatePoint ListOfNumber
```

For the sake of notational convenience we will assume that the "[i]" operator on a list returns the i^{th} element of the list (NULL if the list is not that long). We will further assume that the ListOfNumber type acts like a vector of Number for vector addition and scalar multiplication.

Ed.: At this point, we need to include types for spatial referencing.

3.2.3.2 Point, and Point Cluster

The simplest example of a spatial domain is a set of points. A realization could be a list of latitude and longitude coordinates. The reference coordinate system would consist of the normal interpretation of latitude, longitude and a choice of ellipsoid. To form a line string from these points, we would have to add a mechanism for linear interpolation between points. A Mercator system would give us rhumb lines, and a spherical system would give us great circles, assuming a geodesic interpolation scheme.

The following two interfaces, Point and PointFactory, form the basis for all the geometry interfaces that follow. Point is a coordinate (of some dimension determined by the Spatial Reference[1]) in space. The well-known types are Number (which is probably either a real number or an integer) and the SpatialReference (or coordinate system). In Point::Coordinate, the SpatialReference needs to be specified only if it is different from the SpatialReference returned by the Geometry interface. Recall that the "factory" type is simply a holder for the type constructor, which is handled by object systems differently. The first constructor creates points directly from the coordinates (a list of numbers) and a spatial reference for those coordinates. The second constructor uses an existing point to determine location, and allows for a change in spatial reference.

[1] Ed.: Note that this places a requirement on SpatialReference that we should remember to include in its definition.

3.2.3.2.1 Point

```
Interface Point under Geometry
{
CoordinatePoint Coordinates();
CoordinatePoint Coordinates(SpatialReference);
Point SetSpatialReference(SpatialReference);
}
```

- Coordinates() — Will return the list of coordinates for the point. The version of this interface with an argument will first convert the coordinates to the spatial reference provided.
- SetSpatialReference() — Which is inherited from geometry, returns a Point.

3.2.3.2.1.1 Point Factory

```
Interface PointFactory under Root
{
Point Point(CoordinatePoint, SpatialReference);
Point Point(Point, SpatialReference);
}
```

- Point() — The constructors for points. The first constructs a point from the well-known types CoordinatePoint and SpatialReference. The second is a "copy" function, it will create a new point from an existing one under the given spatial reference.

We will use simple "linear" geometry for some of the higher dimension well-known types. Several data types are built from sets of points, such as triangulated irregular networks, (TIN), Vornoi diagrams, grids, etc. The PointCluster type will be used in many of these as a well-known type for the constructor. Recall that Point is not a well known type, but has a constructor (in PointFactory) that uses the well known types Number and SpatialReference, and has "copy" functions Coordinates (in Point) and GetSpatialReference (in Geometry) that will return these types, thus enabling copy functions across the most primitive interfaces. Point, and Point Cluster can be thought of as being "recursively well known."

3.2.3.3 Curves and 1 Dimensional Geometry

3.2.3.3.1 Curve

We already have a template for CurvePoint that gives us the basic one dimensional nature of curves of points. Since CurveType will be seldom used for anything but standard coordinate defined points, we will use Curve to mean CurvePoint (unless there is a reason for confusion).

We also need interfaces that will allow to define some basic mechanisms to move curves from one system to another. To export a curve, the export system will have to express the data in a well-known type understood by the import system. We need a mechanism to determine accuracy of the representation. Two types of measurements will be needed; one for distance along the curve and another for distance between the curve and its approximation.

Note that the assertions still hold, and, in particular, curves are continuous. Multicuves are piecewise continuous, and the subsegments join at their endpoints.

```
Interface Curve under Geometry, StoredFunctionRealPoint
{
Point               Value(Real);
ClosedIntervalReal  Domain();
```

```
ScaledCurve         SetScale(Real);
Curve               ChangeScale(Real)
Real                      Length(Real, Real);
Real                      LocalScaleFor(Real);
}
```

- Value(r) — the value of the curve at the input point r. We will write this as C(r) for curve C.
- Domain() — returns the real interval for which the curve is valid
- SetScale() — returns the scaled curve that traces the given curve
- ChangeScale() — returns a curve for which the parameter has been changed by the factor given.
- Length() — returns the length of the curve between the two parameter values. If one of the values is not in the domain, then Null is returned. If the first value is greater than the second, a negative arc length is returned.
- LocalScaleFor() — returns the local scale for the curve. If the curve is differentiable the this is the length of the tangent vector, LocalScaleFor(t) = ‖C'(t)‖. If the curve is parameterized by arc length, then this value is 1.

```
Interface MultiCurve under Curve
{
Point                       Value(Real);
ListOfClosedIntervalReal    Domain();
}
```

- ListOfClosedIntervalReal Domain() — returns a list of open and closed intervals for which the curve value function is valid.

```
Interface SimpleCurve under Curve
{
}
Interface ClosedCurve under Curve
{
}
Interface ScaledCurve under Curve
{
Real Scale();
}
```

- Simple curves — do not pass through the same point twice.
- Closed curves — end where they began.
- Scaled curves — the parameter is a multiple of arc length.

Assertions:

1. The parameter in a scaled curve is a multiple of curve length.
   $\forall C \in$ ScaledCurve, $\forall s, t \in$ C.Domain, $s < t$ [C.Length(s, t) = (t - s)C.Scale]
2. A line string constructed from a curve passes through the integer points of the curve.
   $\forall C \in$ Curve, $\forall i \in \mathbf{Z}^{0+}$ [L= C.LineString $\wedge$ i < C. Length $\Rightarrow$ L(i) = C(i)]
3. Set scale 'stretches' the parameter value to arc length times the scale
   $\forall C \in$ Curve, $r \in \mathbf{R}$, S = C.SetScale(r) [a, b] = C.Domain,
   [$t \in$ C.Domain, x = r (C.Length(a, t)) $\Rightarrow$ $x \in$ S.Domain $\wedge$ C(t) = S(x)]

4. Change scale ‘stretches’ the parameter value.

$\forall$ C $\in$ Curve, r $\in$ **R**, S = C.ChangeScale(r)

[t $\in$ C.Domain $\Rightarrow$ rt $\in$ S.Domain $\wedge$ C(t) = S(rt)]

3.2.3.3.2 Line

We can define a type for a simple two point line for each SpatialReference. We can define a curve on the interval [0,1] between two coordinate points by:

```
CoordinatePoint FromPoint, ToPoint;
c(t)[i] = (1-t) * FromPoint[i] + t * ToPoint[i];
```

And, we can map this canonical representation into any other by an appropriate change of parameter.

Using this idea, we can define a 2 point Line as a Curve of Points that satisfy certain Assertions.

```
Interface Line under Curve
{
Line          Canonical();
Curve         SetSpatialReference(SpatialReference);
}
```

- Canonical() — returns an equivalent line parameterized on [0, 1].
- SetSpatialReference() — Which is inherited from geometry, cannot always return a line (unless the transformation involved is affine) It will return a curve of a type defined and chosen by the implementation of the interface. This will most likely change the Geometry::Resolution and Geometry::Accuracy.

3.2.3.3.2.1 Line Factory

```
Interface LineFactory under Root
{
Line Line(Point, Point);
}
```

- Line() — creates a line parameterized on [0, 1], with the two endpoints in the given order.

Assertions:

1. The domain of a line is a non-trivial interval.

   $\forall$ L $\in$ Line [$\neg$ (L.Domain.Low = L.Domain.High)]
2. The domain of any canonical line is [0,1]

   $\forall$ M $\in$ Line [L = M.Canonical $\Rightarrow$ (L.Domain = ClosedIntervalReal(0.0 1.0))]
3. The function that defines the mapping of a line from its domain to the associated coordinate system is linear.

   $\forall$ M $\in$ Line [L = M.Canonical $\Rightarrow$ [(t $\in$ L.Domain) $\Rightarrow$

   L(t).Coordinate = (1-t) L(0.0).Coordinate() + t L(1.0).Coordinate())]

   ]
4. The function from the domain of a line to the domain of its canonical representation is linear.

   $\forall$ M $\in$ Line [L = M.Canonical $\Rightarrow$ [L(z) = M((z-x)/(y-x))]
5. Two lines are equal if and only if their canonical representations are equal.

   $\forall$ K,M $\in$ Line [(K = M)) $\Leftrightarrow$ (K.Canonical = M.Canonical)]
6. $\forall$ s, t $\in$ Point [Line(s, t)(0.0) = s]
7. $\forall$ s, t $\in$ Point [Line(s, t)(1.0) = t]

We have used a canonical parameterization of 1 integer per vertex. We could have also used an arc length parameterization, which would have simplified some of the differential geometry, but made the simple geometry harder.

3.2.3.3.3 Line String

Continuing on with LineString, we define a canonical representation to be one that uses the parameterization to distinguish the vertexes.

```
Interface LineString under Curve
{
LineString   Canonical();
ListOfPoint  Vertexes();
Curve        SetSpatialReference(SpatialReference);
}
```

- Canonical() — returns an equivalent line parameterized so that integer parameters fall on vertexes of the line string.
- SetSpatialReference() — Which is inherited from geometry, cannot always return a line string (unless the transformation involved is affine) It will return a curve of a type defined and chosen by the implementation of the interface. This will most likely change the Geometry::Resolution and Geometry::Accuracy.

Assertions:

1. A line string and its canonical representation are equal as geometries.
   $\forall$ M $\in$ LineString, [L= M.Canonical $\Rightarrow$ L = M]
2. A canonical line string is parameterized from 0, to one less than the number of vertexes.
   $\forall$ M $\in$ LineString, [L= M.Canonical $\Rightarrow$ L.Domain = [0.0, M.Vertexes.Length -1]]
3. Between the
   $\forall$ M $\in$ LineString, i $\in \mathbf{Z}^+$, t $\in \mathbf{R}$
   [((L= M.Canonical) $\wedge$ (i < t < M.Domain.High)) $\Rightarrow$
   (L(t).Coordinate = (i+1-t)L(i).Coordinate + (t-i)L(i+1).Coordinate)]

3.2.3.3.3.1 Line String Factory

```
Interface LineStringFactory under Root
{
LineString LineString(ListOfPoint, OrderedListOfReal);
LineString LineString(Curve);
}
```

Assertions:

1. $\forall$ i, n $\in \mathbf{Z}^{0+}$, s $\in$ ListOfPoint, r $\in$ OrderedListOfReal
   [(n = s.Length) $\wedge$ (L = LineString(s, r) $\wedge$ i < n $\Rightarrow$ L(i) = r[i])
2. $\forall$ n $\in \mathbf{Z}^{0+}$, s $\in$ ListOfPoint, r $\in$ ListOfReal = (0,1,2,3,...,n-1)
   [(n = s.Length) $\wedge$ (L = LineString(s, r) $\Rightarrow$ (L = L.Canonical)

3.2.3.4 Surfaces

The basic strategy for defining 2 dimensional objects is to decompose them into what is commonly called 2 ½ D surfaces (called here 'surfaces'). These surfaces are isomorphic to planar surfaces. Surfaces are 'stitched' together along boundary components to form more complex surfaces.

3.2.3.4.1 Simple Surface

Because of the Jordan Curve Theorem, we can define an n-cell $\mathbf{D^n}$ in any $\mathbf{E^n}$ by specifying its boundary (a $\mathbf{S^{n-1}}$, a sphere of 1 less dimension). What that means for surfaces, is that any simple closed curve (which by definition is a $\mathbf{S^1}$ sphere) defines a $\mathbf{D^2}$ surface. This is exactly what SDTS uses in defining surfaces through the use of closed chains.

For one dimension curves, the parameter space is an interval on the real line. For simple surfaces (and for complex surfaces later) the canonical parameter space is a closed set in 2-space. We will define these surfaces as the difference of 2-cells. Each surface will be associated to one 'exterior boundary' from which we cut any number of holes by specifying 'interior boundaries'. In the parameter space $\mathbf{E^2}$, this gives pictures like that in Figure 3-8. Note that there are no restrictions on the manner in which the 'holes' interact with the exterior boundary of the surface (in the diagram, one of them is tangent). An alternative would be to allow the exterior boundary to self intersect, but this requires complex logic at the point of intersection and makes some of the assertions quite a bit more complex.

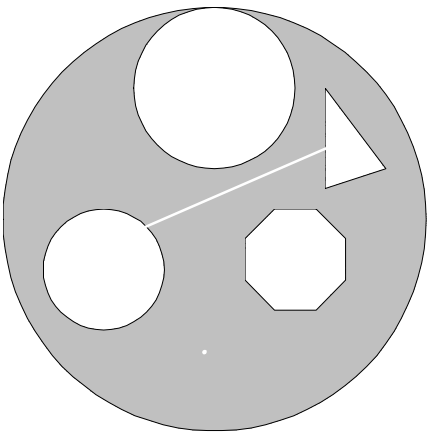


Figure 3-8 A surface in $\mathbf{E^2}$ Defined by Cutting 4 $\mathbf{D^2}$ a line, and a point, out of a larger $\mathbf{D^2}$

Note that some of the cells are defined by smooth curves, while others are polygons.

3.2.3.4.1.1 Surface Domains

```
Typedef E2point (Real, Real)
Interface SurfaceDomain under SetOfE2point
{
SimpleClosedCurveE2point  ExteriorBoundary();
SetOfSimpleClosedCurveE2point InteriorBoundary();
SetOfCurveE2point               CutLines();
SetOfE2point             Punctures();
}
```

- ExteriorBoundary() — returns the exterior boundary of the surface domain.
- InteriorBoundary() — returns a set of interior boundaries.
- CutLines() — are curves that have been sliced out of the surface domain.
- Punctures() — are point holes that have been punched out of the surface domain.

```

Interface SurfaceDomainFactory under Root
{
```

```
SurfaceDomain SurfaceDomain(     SimpleClosedCurveE2point,
                            SetOfSimpleClosedCurveE2point
                            SetOfCurveE2point, SetOfE2point);
}
```

The 'element of' function for surface domain, inherited from 'set of Euclidean 2D points' is the one defined by the Jordan curve theorem. For our purposes, we can say that a point is in the surface if it is interior to the 'exterior boundary,' and exterior to the 'interior boundaries' by, say, a point in polygon style test.

Assertions:

1. Surface domains are closed.
   $\forall$ A $\in$ SurfaceDomain; $\forall$ c $\in$ A.InteriorBoundary $\bigcup$ {A.ExteriorBoundary}
   $\forall$ p, $\in$ E2point, [p $\in$ c $\Rightarrow$ p $\in$ A]
2. No two boundaries 'cross.' They may intersect at a point, but only at a tangent.
   $\forall$ A $\in$ SurfaceDomain; $\forall$ c1, c2 $\in$ A.InteriorBoundary $\bigcup$ {A.ExteriorBoundary}
   $\forall$ p, q $\in$ E2point, p, q $\in$ c1
   [p $\in$ SurfaceDomain(c2, NULL) $\Rightarrow$ q $\in$ SurfaceDomain(c2, NULL)]
3. All points interior to the surface are interior to the exterior boundary.
   $\forall$ A $\in$ SurfaceDomain, $\forall$ p $\in$ E2point,
   [p $\in$ A $\Rightarrow$ p $\in$ SurfaceDomain(A.ExteriorBoundary, NULL)]
4. All points interior to the surface are exterior to or on interior boundaries.
   $\forall$ A $\in$ SurfaceDomain, $\forall$ p $\in$ E2point, $\forall$ c $\in$ A.InteriorBoundary
   [p $\in$ A $\Rightarrow$ p $\notin$ SurfaceDomain(c , NULL) \ c]
5. All points on a cut line, or at a puncture are outside the domain.

3.2.3.4.1.2 Simple Surfaces

Using this we will define a surface to be an object for which this sort of parameterization exists. Not all surfaces will qualify as surfaces, because a surface must be mapable into a plane (the plane of its parameterization). In other words, you can fold, bend, stretch or cut a surface, but pasting it back together (say as a cylinder or torus) can cause problems.

```
Interface SimpleSurface under Geometry,
      StoredFunctionE2pointPoint
{
Point        Chart(E2point);
SurfaceDomain             Domain();
SetOfSimpleCurve    Boundary();
SetOfSimpleCurve    CutLines();
SetOfPoint          Punctures();
}
```

- Chart() — maps the surface domain into points in the surface's spatial reference. Surface::Chart is the same as StoredFunction::Value.
- Domain() — the primitive planar domain that is the preimage of the surface in question. In the usual context of 2 ½ D surfaces, this is the projection of the surface onto a horizontal plane (usually the plane of the map projection).
- Boundary() returns a set of closed curves that comprise the 'boundaries' of the surface as defined by the image of the boundary of the surface domain. Each boundary of the surface domain can be mapped onto one or more curves in the surface's boundary. This set of boundary curves will be used to define complex surfaces in a later section.

- CutLines() and Punctures() are the images of the corresponding items in the surface domain. They are not part of the boundary.

Assertions:

1. SimpleSurface::Chart is the same as StoredFunctionE2pointPoint::Value.
2. Charts are continuous, one to one maps from the surface domain in Euclidean 2 space to the spatial reference of the surface, including points on the cut lines and at punctures.
3. The boundary of a surface (the set union of all the boundary curves) is the image under the chart of the boundary of the surface domain.

3.2.3.4.2 Complex Surfaces

A complex surface (following the lead of cellular complex) will be a set of surfaces stitched together along common curves on their boundary.

```
Interface Surface under Geometry
{
SetOfSimpleSurface  Components()
SetOfCurve   Boundary().
}
```

- Components() will return a set of simple surfaces that are used for the various parts of the surface. The surfaces will intersect one another only along their common boundaries. Each surface boundary curve for this set will either intersect no other boundary curve in the set, or be equal to one and only one other boundary curve in the set.
- Boundary() returns a set of curves that is the subset of surface boundaries not covered twice by the surface components of the surface. Each boundary curve from the surface components will either be in the boundary of the surface, or be in precisely two boundaries of surface components of the surface.
- A surface is closed if its boundary is empty.

3.2.3.5 Solids

Solids become easy again (as opposed to surfaces) because of the limitations of 3 dimensional space. If we have a closed surface in three space, it has an interior that it separates from the rest of the space. This is the defining criteria for solids.

```
Interface Solid under Geometry
{
SetOfSurface        Boundary();
Surface             ExteriorBoundary();
SetOfSurface        InteriorBoundary();
Boolean             Equals(Solid);
}
```

- Each Surface in the boundary is closed. The solid is the interior of the largest (exterior) boundary surface, with the others removed as holes.

3.2.3.6 Complexes

The same mechanism for defining complex combinations of surfaces can be used to define complex combinations of other spatial objects (see section **Error! Reference source not found.**). Spatial object is defined below (**Error! Reference source not found.**).

```
Interface Complex under Geometry
{
```

```
SetOfSpatialObject  Components()
SetOfSpatialObject  Boundary().
}
```

- Components() will return a set of geometries that are used for the various parts of the complex. The set of component geometries will be closed under the boundary operators. The components of a fixed dimension will intersect one another only along their common boundaries. Each component's interior will either intersect no other component in the set, or be contained in the common boundary of some number of other component of one higher dimension in the set.
- Boundary() returns a set of geometries that is the subset of boundaries not covered on all sides by the components of the complex.

3.2.4 Features and Attributes

This section discusses feature types, features and attribute descriptors (including their relationship to features).

Informative Note: Because the OGIS is being designed to yield interoperability, we felt it necessary to include a general mechanism for what we will call *schema interoperability*. Schema interoperability is defined as the ability to determine whether features in one application context are mapped to those in other contexts and the ability to perform queries that can access features from these differing contexts without end user intervention. We realize that the goal of schema merger and/or integration is a different (probably impossible) problem. In thinking about our problem, we concluded that we would need to represent both feature types as well as features. In other words, we had to maintain metadata about our feature objects.

The next-generation GIS software will need a mechanism to create new features, and even feature types, dynamically (that is, without recompilation of application source code). Thus the concept of maintaining metadata about features is necessary.

Several types used below can be implemented as primitives, but may require reconsideration later. We will use the following type definitions to cover possible changes later, and to make the specification more readable:

```
Typedef FeatureName       string
Typedef AttributeName     string
Typedef AttributeType     Type
Typedef Constraint  BooleanPredicate
Typedef CatalogName       string
Typedef Dimension         integer
```

3.2.4.1 Spatial Object

A spatial object will act as a representation of a spatial feature (its extent attribute). Essentially, a spatial object is geometry used by a feature to express extent. The basic interfaces to SpatialObject are given here. The spatial relationships between objects can get quite complex, but the basic metric, set, and topological operators can cover most important cases (see Egenhofer and Herring [3-23-2]).

```
Interface SpatialObject under Geometry
{
SetOfGeometry       Geometries();
Boolean             IsIn(Point);
}
```

- Geometries()—returns a list of the geometries that make up this spatial object.
- IsIn()—is the operator '∈.'

```
Interface SpatialOperator under SetOperator
{
SpatialObject Aggregate(SetOfSpatialObject);
SetOfSpatialObject Disaggregate(SpatialObject);
SpatialObject Union(SetOfSpatialObject);
SpatialObject Intersection(SetOfSpatialObject);
SpatialObject SymmetricDifference(SpatialObject, SpatialObject);
SpatialObject Difference(SpatialObject, SpatialObject);
}
```

- Aggregate()—Combines several SpatialObjects into one (by a point set union).
- Disaggregate()—Separates the SpatialObject into one or more simple geometry types (as SpatialObjects). Although the process is undefined, we would suggest to any implementation that the Disaggregate return a minimal answer in terms of the number of primitives.
- Union() ... Difference() — Perform the named operations on spatial objects.

```
Interface SpatialMetric under Root
{
Distance     Diameter(SpatialObject);
Distance     Distance(SpatialObject, SpatialObject);
}
```

- Diameter() — finds the diameter of a spatial object.
- Distance() — finds the distance between two spatial objects

```
Interface SpatialBoolean under Root
{
Boolean      Intersects(SpatialObject, SpatialObject);
Boolean      Contains(SpatialObject, SpatialObject);
Boolean      Equals(SpatialObject, SpatialObject);
}
```

- Intersects() ... Equals() — Perform the named Boolean operations on the spatial objects.

```
Interface TopologicalOperator under SetOfStoredFunction
{
SpatialObject       Boundary(SpatialObject);
SpatialObject       ExteriorBoundary(SpatialObject);
SpatialObject       EnclosedSpace(SpatialObject);
Integer             Dimension(SpatialObject);
Integer             NumberConnectedComponents(SpatialObject);
SetOfSpatialObject  ConnectedComponents(SpatialObject);
SpatialObject       Interior(SpatialObject);
SpatialObject       TopologicalSkeleton(SpatialObject,
      Dimension);
}
```

- Boundary() — finds the topological boundary of a spatial object.
- ExteriorBoundary() — finds the exterior boundary of a spatial object, which makes sense only for an object of the same dimension as the containing coordinate system (such as a solid in 3 space or a surface in 2 space).
- EnclosedSpace()—Returns a surface (or solid) without holes whose exterior boundary is the same as the passed one.
- Dimension() — returns the largest dimension of the components of the spatial object.

- NumberConnectedComponents()—Returns the number of connected components in the set.
- Connected Components()—Returns a (constructed) list of the connected components.
- Interior()—Returns a SpatialObject that represents the "open" part of the object. Remember that "open" is a function of the dimension of the components.
- TopologicalSkeleton()—Returns a "skeletonized" version of the spatial object, including either objects of the given dimension (or smaller) or boundaries of larger dimension objects.

Assertions:

1. The set theoretic operators behave as they should for $\in$, $\subset$ and =.
   $\forall$ A, B $\in$ SpatialObject [A $\subset$ B $\Leftrightarrow$ ($\forall$ p $\in$ Point, p $\in$ A $\Rightarrow$ p $\in$ B)]
   $\forall$ A, B $\in$ SpatialObject [A = B $\Leftrightarrow$ (A $\subset$ B $\wedge$ B $\subset$ A)] …

3.2.4.1.1 Spatial Object Factory

```
Interface SpatialObjectFactory under Root
{\\ Constructors for simple geometry in geometry
SpatialObject       SetSpatialReference(SpatialReference);
SpatialObject       SpatialObject(SetOfPoint);
SpatialObject       SpatialObject(SetOfPointCluster);
SpatialObject       SpatialObject(SetOfCurve);
SpatialObject       SpatialObject(SetOfSurface);
SpatialObject       SpatialObject(Complex);
}
```

Note that protocols to get and set the spatial reference are inherited from geometry. SetSpatialReference is included here only to redefine its return type to SpatialObject.

3.2.4.2 Feature

Features are the first specification that deals with persistence. This means that features will have identity and must have destructors for removal from persistence. Other objects, which are not persistent, are only known by value.

The constructor for Feature is in FeatureType (which acts as a type factory for Feature).

```
Interface Feature under Identity, Tuple
{
ListOfAttribute     Attributes();
Attribute           Attribute(AttributeDescriptor);
SetOfFeatureType    Type();
Feature             SetSpatialObject(SpatialObject, Time);
Feature             SetAttributes(SetOfAttribute, Time);
Boolean             CheckConstraint(FeatureType);
Boolean             Delete(Time);
Feature             Version(Time);
}
```

- Extent() — returns a list of features making up the component features of a complex feature. Extent() is assumed to be an acyclic relation, but is not necessarily hierarchical . The geometry of a complex feature is assumed to be the combination of all the geometries of the subfeatures in the extent.
- Attributes() — return the list of attributes, or the requested attribute for the feature The Attribute type is defined in section **Error! Reference source not found.**.

- Type() — return the set of feature types for this feature. It is inherited from Root, and is respecified here to restrict its return type.
- Tuple::Metadata() — returns the list of attribute types and the extent type.
- Tuple::Names() — returns the names of the attributes and 'Extent' for the geometry (or feature collection.
- SetSpatialObject() — set the spatial object for the feature. The time stamp is a specification to the database of the 'official' time at which the change is to take place. In the specification, this creates a new feature, which may differ from the old one by only the time code of its identity.
- SetAttributes() — set the attribute list for the feature. It does this by replacement. This method should invoke the CheckConstraint method on the AttributeDescriptor for each attribute. In the specification, this creates a new feature, which may differ from the old one by only the time code of its identity.
- Delete() — specifies a time at which the feature is considered to be deleted from persistence. In the specification, this does not delete the existing feature as much as change the time code of its identity.
- Version() — returns a feature copy as it existed at the time indicated.
- All geometry interfaces are inherited from the unique spatial object or set of feature extent.

```
Interface SimpleFeature under Feature
{
SpatialObject       Extent();
}
```

- Extent() — returns the SpatialObject of a simple feature.

```
Interface ComplexFeature under Feature, SetOfFeature
{
SetOfFeature        Extent();
}
```

- Complex feature supports set of feature interfaces through delegation to its extent.

There is only one SpatialObject for Feature. The rationale is that there will be SpatialObjects that are unions of many SpatialObjects and thus there is no need for an aggregation at that level. Also, OGIS is a specification dealing with things that have a georeferenced geometry. The general mechanism for defining this geometry is the SpatialObject. We should be able to access this SpatialObject in a straight forward manner, which is why we did not just assume that SpatialObject would be one of the attributes of Feature. The mechanism used to access the SpatialObject is the Extent() operation. This differes from the mechanism used to access attributes in general, which is the purpose of the Attributes() operation.

The Type() operation on the FeatureType interface (described in section **Error! Reference source not found.**) provides the mechanism for obtaining metadata about the specific feature. This will be extremely useful when building generic browsers and other high-level tools. It is also used to access the attribute descriptions.

3.2.4.2.1 Attributes and Attribute Descriptors

The AttributeDescriptor type is defined as follows:

```
Interface AttributeDescriptorFactory under Root
{\\ class factory for Attribute Descriptor
AttributeDescriptor AttributeDescriptor(
      AttributeName, AttributeType, Constraint);
```

```
}
Interface AttributeDescriptor under Root
{\\ class factory for attribute
AttributeName       Name();
AttributeType       Type();
Constraint          Constraint();
Boolean             CheckConstraint(Root);
}
```

- AttributeDescriptor()—this is the constructor for an attribute descriptor. The name should indicate a general family of values, such as width or thickness. The type that specific values will have once realized is AttributeType.
- Name()—returns the name of the attribute descriptor.
- Type()—returns the type of the corresponding attribute object.
- Constraint()—returns the constraint for this attribute descriptor.
- CheckConstraint()—returns whether the given Root meets the constraint, if one exists.

The attribute interfaces are:

```
Interface Attribute under Root
{
AttributeDescriptor GetDescriptor();
Root                Value();
}
Interface AttributeFactory under Root
{
Attribute    Attribute(Root, AttributeDescriptor);
}
```

Assertion:

1. Attribute::Value() returns a transient object whose type is consistent with the descriptor.
2. AttributeFactory::Attribute() must be passed a transient object whose type can be converted to that of the attribute descriptor.

3.2.4.2.2 Feature Type

The FeatureType is a type that defines what a feature is, in terms of its attribution. Every Feature object will be able to return its corresponding FeatureTypes. Each of these types will contain a name, and a list of attribute descriptors. Every Feature will normally contain a SpatialObject to define its extent and a list of attribute Objects. The attribute descriptors of FeatureType will correspond to and describe the Objects of Feature. If a feature supports multiple types, then the commonly named attributes should have the same value. If this is not the case, then attribute names may need a type resolution prefix of <typeName>::<attributeName>. Here is the interface for FeatureType.

```
Interface FeatureType under Tupletype
{
// add a FeatureType to an existing instance
Feature      Feature(FeatureType, SpatialObject,
                    ListOfAttribute, Time);
Feature      NewFeatureType(Feature, FeatureType,
                    ListOfAttribute, Time);
Feature      DeleteFeatureType(Feature, FeatureType, Time);
Name         Name();
ListOfAttributeDescriptor
             AttributeDescriptors();
```

```
Constraint    Constraint();
Boolean       CheckConstraint(Feature);
Boolean       Delete();
}
```

- Feature() — Constructs an instance of Feature. The SpatialObject should already have been realized. If it has not been realized, then the SpatialObject argument will be NULL. The SpatialObject method on Feature can be invoked to initialize or replace the spatial object at a later time. The list of attributes should also have been realized, or a NULL list should be passed for this argument. The Attributes method on Feature may be called at any time to initialize or replace the list of attributes. This method should invoke the CheckConstraint method on AttributeDescriptor for each attribute.
- NewFeatureType() — adds attributes to an existing feature that allow it to support another feature type interface. Note that since the same named attribute can appear in multiple types, not all attributes of the added feature type need be added. If the feature already supports this type, then the attribute values are updated. (Time stamps are updated as appropriate.)
- DeleteFeatureType() — deletes attributes from an existing feature that is 'giving up' it role as a particular feature type. Note that since the same named attribute can appear in multiple types, not all attributes of the deleted feature type need be deleted.
- Name() — returns the name of the feature type.
- AttributeDescriptors() — returns a list of AttributeDescriptor objects for the feature type.

```
Interface FeatureTypeFactory under Root
{// a FeatureType constructor
FeatureType FeatureType(
      FeatureName, ListOf<AttributeDescriptor>,Constraint);
}
```

- FeatureType() — the constructor for FeatureType. The name should indicate a general class of features, such as "bridge" or "mountain peak".

3.2.4.3 Sets of Features, Datasets, and Catlogs

Geographic information is almost universally defined in terms of higher-order collections of entities and phenomena rather than the underlying spatial representation of those entities and phenomena. That is, while geography may be delineated and described in terms of polygons, grid cells, or x, y, z, t coordinates with certain attribute descriptors, effective geographic understanding is typically based on structured aggregations of such spatial objects into "maps," "images," or simply "datasets."

Clearly, synoptic views of a landscape, in the most general sense, are predicated on this level of representation. Application users need to see, for example, landcover maps and Landsat scenes, not individual landcover polygons or image pixels. Even query for individual entities within a data collection is usually executed to provide spatial and thematic context to the query result. For example, coniferous forest landcover polygons are returned as a set (perhaps with only one member) which have specific geographic relationships (spatial and thematic) to other landcover polygons.

This document describes features and coverages as one of simple or compound spatial objects with a set of attributes versus functions mapping elements within a spatial domain to attribute values or descriptions. This distinction will not be furthered

explored here, except to note that any form of underlying spatial representation—feature lists, boundary maps, n-dimensional raster images, numeric fields, or network graphs—is amenable to higher level aggregations, subject to certain constraints of the spatial model (e.g., networks are linked into larger networks, images are mosaicked together, and so forth).

All feature, coverage, or network objects contain one or more elements typically corresponding to some real-world, recognizable entity. This correspondence may be explicit, such as the relationship between a coordinate point and a named place; implicit, such as the identification of an ownership parcel based on a boundary line; or inferred, such as the assembly of similarly valued image pixels into landscape phenomena. Structured information pertaining to individual elements within an object comprises *attributes*. Information pertaining to the entire object or a collection of objects comprises *metadata*, as distinct from attributes.

Metadata serve to inform the user of information necessary to determine the following [3-8]:

availability—data needed to determine the sets of data that exist for a geographic location.

fitness for use—data needed to determine if a set of data meets a specific need.

access—data needed to acquire an identified set of data.

transfer—data needed to process and use a set of data.

A *simple dataset* is defined as the combination of a set of identically typed complex objects and their related metadata. Conversely, the metadata alone for a group of datasets comprise the *catalog* about those datasets.

Complex datasets are those whose member objects are of different types, but which nonetheless share metadata relating to dataset identification, spatial reference, data quality, and other dataset characteristics. *Compound datasets* comprise two or more simple or complex datasets, but have common metadata that can be expressed within a catalog (as is also the case with a simple or complex dataset).

At a higher level of aggregation, a documented collection of datasets, whose members may be of many different types, is a *catalog*. Typically, a catalog would be identified as those datasets relevant to a specific problem or issue, or alternatively as the set of information maintained by a data provider. The catalog contains the key metadata for the datasets within a collection.

This section addresses aspects of the OGIS Geodata Model that pertain to aggregation of OGIS features into collections. It also addresses some issues associated with the attributes of individual elements, attribute catalogs and data dictionaries, for the purpose of facilitating schema merger and federation. However, it does not address operational implications of dataset interoperability and transfer mechanisms, since these are the subject of testbed applications of OGIS.

3.2.4.3.1 Conceptual model

Figure 3-9 depicts the OGIS Geodata Model's highest levels of abstraction and aggregation. The Catalog encapsulates (or collects) datasets. Datasets are a FeatureSet, meaning that they embody some collection of Features (remember that Coverages are also Features). The additional information that Dataset encapsulates is information as to the physical storage location, or Host. Note also, that, because it implements the FeatureSet interface, Dataset also encapsulates Metadata. The content of a Dataset, in terms of the schema of contained Features is described by the contained

FeatureDictionary. This FeatureDictionary is a collection of FeatureTypes, which describes the attribution of individual Features. FeatureSet serves to collect Features, including other FeatureSets. This self-collecting relationship is possible because FeatureSet implements the interface of Feature, as well as the interface for a set. FeatureSet is described by Metadata.

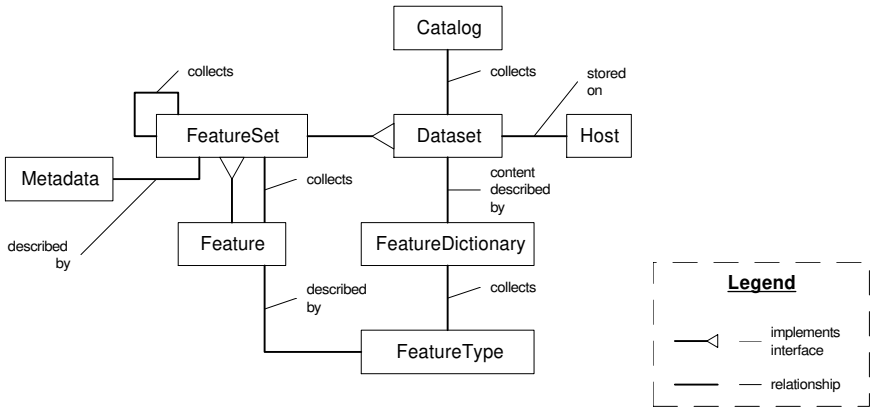


Figure 3-9 Relationship between high-level OGM abstractions and aggregations.

Metadata included with a Dataset includes relevant identification material. The default content of Metadata is as specified in the FGDC Content Standards for Digital Geospatial Metadata, Final Draft. However, there must exist an mechanism for attaching (and defining) additional metadata that follows other standards, such as a corporate or organizational standard metadata description.

Catalogs are comprised of Datasets which implement the FeatureSet interface. Metadata at the FeatureSet level includes information addressing identification, data quality, spatial data organization information, spatial reference information, metadata reference information, citation information, and possibly temporal information.

At the level of Feature, metadata derive (i.e., are not necessarily stored explicitly but are accessible via some interface) from private dictionaries, application dictionaries (e.g., DLG), and other geomatics standards. These are also sometimes referred to as the data dictionary for a feature, network, or coverage. The point of distinction with element attributes is that per-element metadata where these metadata items are not pervasive and identical across an object (e.g. polygons in a coverage) are structured as attributes, whereas metadata are uniform across some meaningful set of information and can be used to inform the creation of a catalog or directory. This aspect is facilitated in part by the “documentation” about Feature found in the FeatureType and the aggregate FeatureDictionary interfaces.

3.2.4.3.2 Components

As already defined, Features encapsulate georeferenced geometry and attribution and are the first level of persistence in OGIS. One or more Features comprise a FeatureSet, which is not necessarily peristent. Dataset, on the other hand, is persistent and thus must be locatable. In common practice, Datasets are uniform in the Features that they collect

— i.e., a Dataset contains one or more Features comprising similar geometry, spatial referencing, and attribution.

3.2.4.3.2.1 FeatureSet

The FeatureSet collects Features and it should implement the Feature interface. Its interface is defined below:

```
Interface FeatureSet under Feature, SetOfFeature
{
Metadata Metadata();
}
```

3.2.4.3.2.2 Dataset

The Dataset interface should provide information about storage (because it must be persistent). It must also provide access to data dictionary information about its contained Features. While this information is available via the Feature interface, it is much more accessible for browsing and other similar functions at this level.

```
Interface Dataset under FeatureSet
{
CharacterString     Name();
Host                Host();
FeatureDictionary FeatureDictionary();
}
Interface DatasetFactory under Root
{
Dataset      Dataset(Name, Host, FeatureDictionary);
}
```

3.2.4.3.2.3 Catalog

The Catalog interface is the primary interface to collections of data (Datasets). It is a high-level interface that provides access to datasets that might reside on hosts in a local or wide area network.

```
Interface Catalog under SetOfDataset
{
Boolean      DeleteDataset(Name);
Dataset      FindDataset(Name);
}
Interface CatalogFactory under Root
{
Catalog      Catalog(Name, SetOfDataset);
}
```

3.2.4.3.2.4 Catalog Service

The Catalog Service is an interoperability aid. The interface will enable applications to federate multiple catalogs for some application context.

```
Interface CatalogService under Identity, SetOfCatalog
{
SetOfCatalogName    GetCatalogNames();
Catalog             FindCatalog(Name);
}
```

Ed.: Should there be a factory for CatalogService or should we assume that OGIS implementations provide a mechanism for creating such a thing and making it available via some interface over the net??

3.2.4.3.3 Feature Dictionaries and Mappings

Having defined feature types, features, and attribute descriptors, we can move to the definition of feature dictionaries. These named dictionaries will contain instances of FeatureType and mappings from the dictionary to other dictionaries. The purpose of mappings is to provide the beginnings of support for federation and schema merger. While there is no real possibility of sovling the schema merger problem in the general case, there are mechanisms for simplifying the problem using techniques like the one proposed here. Figure 3-10 depicts the relationship between types relevant to dictionaries and mappings. It shows that feature mappings are contained in feature dictionaries and that the specific mapping between the feature types in the two dictionaries is defined by the mapping. There may be many such mappings per dictionary and feature types within a dictionary may be mapped to many other dictionaries.

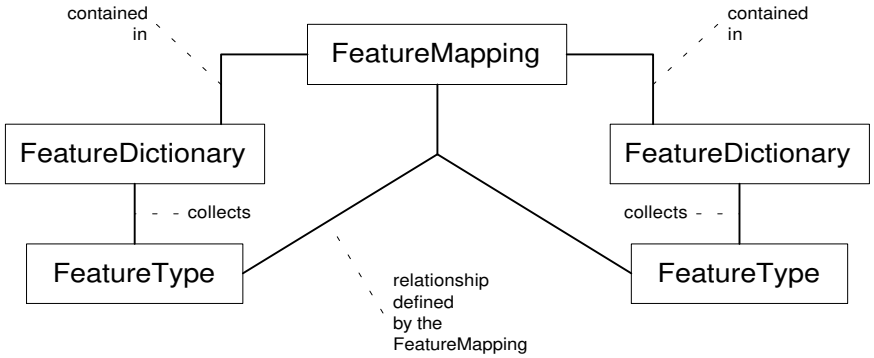


Figure 3-10 Relationships between feature types, dictionaries and mappings.

The Feature Dictionary Service is proposed as an interoperability aid. It defines a special feature dictionary called the Master Dictionary. This dictionary serves the role of providing a central schema for some application context (i.e, it contains a global conceptual schema in federated database terms). This is necessary because many application contexts will require integration across schemas. For the most part, the feature mappings will be to the Master dictionary, but this is not a strict requirement as we have defined FeatureDictionary and FeatureMapping, meaning that mappings to dictionaries other than the Master are possible.

3.2.4.3.3.1 Feature Dictionary

```
typdef FeatureTypeName CharacterString

Interface FeatureDictionary under SetOfFeatureType
{
DictionaryName            GetName();
FeatureType               GetFeatureType(FeatureTypeName);
ListOfFeatureTypeName     GetFeatureTypeNames();
ListOfFeatureMapping      GetFeatureMappings();
ListOfFeatureMapping      GetToMappings(DictionaryName);
ListOfFeatureMapping      GetFromMappings(DictionaryName);
Boolean                   SetName(DictionaryName);
Boolean                   DeleteFeatureType(FetureTypeName);
```

```
Boolean
            SetFeatureMappings(ListOfFeatureMapping);
Boolean
            AddFeatureMapping(ListOfFeatureMapping);
Boolean                   AddFeatureMapping(FeatureMapping);
Boolean                   AddFeatureMapping(FeatureTypeName,
      FeatureTypeName, FeatureDictionary);
Boolean                   DeleteFeatureMapping(
      ListOfFeatureMapping);
Boolean                   DeleteFeatureMapping(FeatureMapping);
Boolean                   DeleteFeatureMapping(FeatureTypeName,
      FeatureTypeName, FeatureDictionary);
}
```

GetName()—return the name of this feature dictionary.

GetFeatureType()—return the feature type matching the name passed as an argument.

GetFeatureTypeNames()—return the list of names of feature types stored in this dictionary.

GetFeatureMappings()—return the list of feature mapping contained in this feature dictionary.

GetToMappings()—return the list of feature type names that the argument name is mapped to in the feature dictionary.

GetFromMappings()—return the list of feature type names that are mapped to the argument name in the feature dictionary.

SetName()—set the name of the feature dictionary.

DeleteFeatureType()—delete the given (list of) feature types from the feature dictionary.

SetFeatureMappings()—replace the feature mappings in the feature dictionary with those in the argument list.

AddFeatureMappings()—append the given list of feature mappings to the feature dictionary.

AddFeatureMapping()—append the given list of feature mappings to the feature dictionary.

DeleteFeatureMappings()—delete the given list of feature mappings from the feature dictionary.

DeleteFeatureMapping()—delete the given list of feature mappings from the feature dictionary.

```
Interface FeatureDictionaryFactory under Root
{
FeatureDictionary   FeatureDictionary(
      DictionaryName, ListFeatureType,ListOfFeatureMapping);
}
```

A feature mapping simply maps one name (the *from* name) to another name (the *to* name) in a feature dictionary (the *to* dictionary). Because feature mappings are associated with a feature dictionary, the *from* dictionary is implicitly the containing feature dictionary. The names that are mapped are the names of feature types. Here is the definition of FeatureMapping:

```
Interface FeatureMapping under Root
```

```
{
FeatureTypeName     GetFromName();
FeatureTypeName     GetToName();
FeatureDictionary   GetToDictionary();
Boolean             SetFromName(FeatureTypeName);
Boolean             SetToName(FeatureTypeName);
Boolean             SetToDictionary(FeatureDictionary);
}
```

FeatureMapping()—this is the constructor for feature mappings.

GetFromName()—this method returns the *from* name of the feature mapping.

GetToName()—this method returns the *to* name of the feature mapping.

GetToDictionary()—this method returns the *to* dictionary of the feature mapping.

SetFromName()—this method sets the *from* name of the feature mapping.

SetToName()—this method sets the *to* name of the feature mapping.

SetToDictionary()—this method sets the *to* dictionary of the feature mapping.

```
Interface FeatureMappingFactory under Root
{
FeatureMapping FeatureMapping(
       FeatureTypeName, FeatureTypeName,FeatureDictionary);
}
```

3.2.4.3.3.2 Feature Dictionary Service

The Feature Dictionary Service (FDS) is defined to provide access to a set of feature dictionaries that will be available in any OGIS-environment. Every FDS will have exactly one Master dictionary. All other feature dictionaries will have a set of feature mappings that map entries in them into the Master dictionary. If any feature dictionary contains items that do not map into the Master dictionary, then they must be inserted into the Master dictionary. The following is the interface for the `FeatureDictionaryService`:

```
Interface FeatureDictionaryService under Indentity
{
Boolean AddMasterDictionary(FeatureDictionary);:
Boolean AddDictionary(FeatureDictionary);
Boolean DeleteDictionary(FeatureDictionary);
ListOfDictionaryName GetDictionaryNames();
ListOfFeatureDictionary GetDictionaries();
FeatureDictionary FindDictionary(DictionaryName);
}
```

AddMasterDictionary()—make the given feature dictionary the master dictionary.

AddDictionary()—add the given dictionary to the list of dictionaries managed by this feature dictionary service.

DeleteDictionary()—delete the given dictionary from the list of dictionaries managed by this feature dictionary service.

GetDictionaryNames()—return the list of names of feature dictionaries managed by this feature dictionary service.

GetDictionaries()—return the list of feature dictionaries managed by this feature dictionary service.

FindDictionary()—return the feature dictionary that matches the given name, if it is managed by this feature dictionary service.

Ed.: Should there be a factory for FeatureDictionaryService or should we assume that OGIS implementations provide a mechanism for creating such a thing and making it available via some interface over the net??

3.2.5 Networks and Topologies

```
Interface NetworkFeature under ComplexFeature
{
}
Interface TopologyFeature under ComplexFeature
{
}
```

- Extent() for a network is assumed to be composed of only curves and the points at which they terminate.
- Extent() for a topology is assumed to be composed of primitives of the dimension of its extent (see the Topological Operator interface) and those of lower dimension that are contained in the boundary of the highest dimension primitives.

3.2.6 Coverages

Coverages are features with at least one spatial or spatial temporal attribute.

```
Interface template Coverage<Type>
      under Feature, StoredFunctionPoint<Type>
{
<Type> Value(Point);
}
```

- Value() gives the value of the spatial attribute (of type 'Type') at points in the extent of the coverage.

Assertions:

1. The stored function domain is the extent of the feature
   $\forall$ C $\in$ Coverage $\Rightarrow$ [C.Feature::Extent = C.StoredFunction::Domain]
   $\forall$ p $\in$ Points; [p $\notin$ C.Extent $\Rightarrow$ C(p) = NULL]

3.2.6.1 Types of Coverages

3.2.6.1.1 Raster Coverages and Image Processing

Raster coverages can be described in two distinct manners, as a set of point samples, or as a set of cellular averages. The Extent of a raster coverage thus can be a point cluster consisting of the centers of the pixels, or a 2D topological feature consisting of $\mathbf{D^2}$ cells centered on those pixel locations. Both complex geometries would be arranged in a regular order (a rectilinear grid in the pixel based coordinate space of the image). The type of the coverage would depend on the encoding of the pixel values, either RGB (red, green, blue), HIS (hue, intensity, saturation), or some other combination, such as multispectral.

Image processing interfaces will be defined for standard filter based raster processes.

```
Interface RGBpixel under Root
{
```

```
Number Red();
Number Green();
Number Blue();
}
Interface RGBraster under CoverageRGBpixel
{
RGBpixel     Value(Point);
SetOfPoint   PixelCenters()
}
```

3.2.6.1.2 Grid Cell Coverages

Grid cell coverages do not differ structurally from images based on D^2 cells, but their value types vary widely (see Tomlin [3-6], or Mark and Frank [3-4], Section 5.2). The spatial value of a point is determined by the cell in which that point falls. Points on the cell grid itself are either 'Null valued' or are interpolated in a well-defined manner.

3.2.6.1.2.1 Cell

```
Interface Cell under Surface
{
SetOfPoint Corners()
}
```

Assertions:

1. A cell has exact 4 coplanar corner

3.2.6.1.2.2 Grid Cell

```
Interface GridCell<Type> under Coverage<Type>
{
<Type>       Value(Point);
SetOfCells   Grid()
}
```

Assertions:

1. The cells of the grid are arranged in a rectangular array.
2. Adjacent cells have exactly two corners in common.
3. All points within a cell have the same value.

3.2.6.1.3 Sampled Grid Coverages

Sampled grid coverages are the 'point sample' equivalent for grid cell coverages. Unlike grid cells, the samples are assumed to be valid at the individual cell point locations. Other points within the extent of the coverage are interpolates via well-defined algorithms.

Sampled grid processes will follow the same interfaces as grid cell coverages.

3.2.6.1.3.1 Post

```
Interface Post<Type> under Point
{
<Type>       Value()
}
```

3.2.6.1.3.2 Sampled Grid

```
Interface SampledGrid<Type> under GridCell<Type>
```

```
{
<Type> Value(Point);
SetOfCell    Grid();
SetOfPost    Post();
}
```

Assertions:

1. The Posts are the corners of the cells.
2. The value of the sampled grid at a post is the value of the post.
3. The values for points within a cell are interpolated from some number of adjacent posts.

3.2.6.1.4 'Polygon' Surface Coverages

Polygon surface coverages are defined by a topological complex feature, for which the value of the coverage is constant for the interior of the subsurfaces in the topology. Interpolation at the surface boundaries is left to the implementation.

```
Typedef Polygon Surface
Interface PolygonCoverage<Type> under Coverage<Type>, Surface
{
<Type>       Value(Point);
SetOfPolygon Polygons()
}
```

Assertions:

1. The polygons cover the surface.
2. All points within a polygon have the same value.

3.2.6.1.5 'Polyhedra' Solid Coverages

Polyhedra solid coverages are defined by a topological complex feature, for which the value of the coverage is constant for the interior of the solids in the topology. Interpolation at the solid boundaries is left to the implementation.

```
Typedef Polyhedron Solid
Interface PolyhedraCoverage<Type> under Coverage<Type>, Solid
{
<Type>       Value(Point);
SetOfPolyhedron    Polyhedra()
}
```

Assertions:

1. The polyhedra fill the solid.
2. All points within a polyhedron have the same value.

3.2.6.2 Coverage Processing

Ed. Grid cell processing will be defined for Tomlin's algebra, and for cell by cell manipulation via algebraic expression on the values of the cells, similar to a multiple layered spread sheet.

3.2.6.2.1 Overlay and Value Tuples

The standard 'polygon overlay' process is actually applicable to overlays of any type. See, for example Burrough[3-1]. The '(<Type 1>, <Type 2>)' is a tuple whose columns are of the indicated types.

```
Interface Overlay under Root
{
Coverage(<Type 1>, <Type 2>)
             Overlay(Coverage<Type 1>, Coverage<Type 2>)
}
```

1. Overlay has the classical meaning.
   $\forall$ S, T $\in$ Coverages, $\forall$P $\in$ Point, U = Overlay(S,T) $\Rightarrow$ [U(P) [0] = S(P) $\wedge$ U(P)[1] = T(P)]

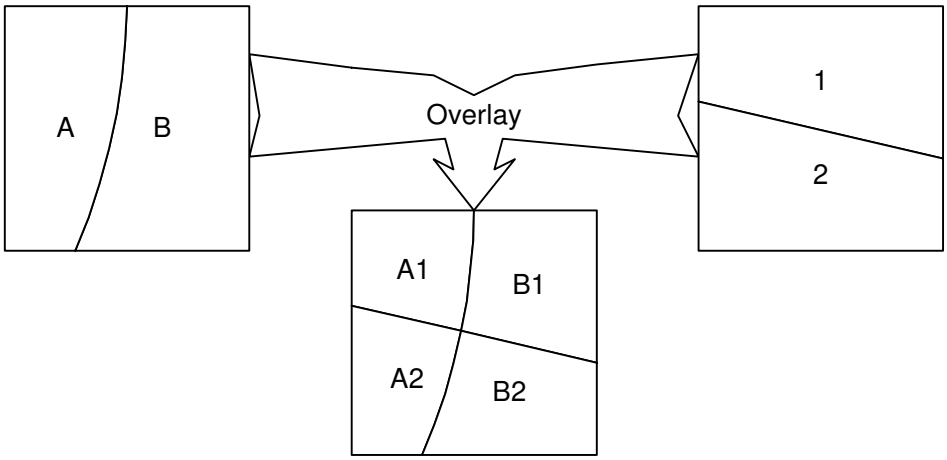


Figure 3-11 Overlay for Polygon Coverages

3.2.6.2.2 Feature and Coverage Synthesis

```
Interface Synthesis under Root
{
SetOfFeature FeatureSynthesis(Coverage,
                              FeatureType,
       ListOfAttributeDescriptor);
Coverage            CoverageSynthesis(SetOfFeature,
                              ListOfAttributeDescriptor);
Coverage            Classification(Coverage, StoredFunction)
}
```

- FeatureSynthesis() — takes a coverage and creates a set of features of a specified using the coverage value function to populate various attributes in these features (the list of attribute types should correspond to the tuple type of the coverage value function. Multiple coverages can be used by first applying the overlay function in section **Error! Reference source not found.**.
- CoverageSynthesis() — takes a set of features and reverses the feature synthesis process, creating a multivalued function.
- Classification() — takes a coverage and a stored function on the tuple type of the value of that function, and creates a new coverage on the range type of the stored function. One such function would be the projection of the tuple type onto any of its columns.

3.2.7 Simplex, Cell, and Complex

This section is provided for background. It outlines some of the more rigid approaches to defining geometry that are in the mathematical literature, but have been

eliminated from the specification in favor of simpler, more generic approaches. They are included here because they may indicate ways that the surface and solid interfaces might be implemented. An expanded discussion can be found in Mark and Frank [3-4], Section 5.1.

We must build complex geometric structures up from simple ones, because there is not a good direct way to define them via their parameterization, as we can for curves. There are basically two methods, one using linear constructs directly in the coordinate space (called 'simplicial complexes') and the other building from generic discs and sphere of various dimensions (called 'cellular complexes'). A modified version of this second method is used in the specification.

3.2.7.1 Simplex

A simplex of any dimension "n" is normally though of as the convex hull of "n+1" points. So a 0-simplex is a point, a 1-simplex is a line, a 2-simplex is a triangle with its interior and a 3 simplex is a tetrahedron with its interior. From this it is fairly obvious that if we drop one of the points from a simplex, we get a simplex of one lower dimension. The "sides" of a 1-simplex are its endpoints (two 0-simplexes). The sides of a 2-simplex are the sides of the triangle (three 1-simplexes). This operation is the boundary operation for the simplex.

```
Typedef 0-simplex Point;
Typedef 1-simplex Line;          \\ Convex hull of two points
Typedef 2-simplex Triangle;      \\ Convex hull of 3 non-
      collinear points
Typedef 3-simplex Tetrahedron; \\ Convex hull of 4 non-coplanar
      points
```

For every dimension n > 0, we define a simplex as follows:

```
Interface <n>-simplex under Root
{
SetOf<n-1>-simplex Boundary();
}
Interface 0-simplex under Root
{
Point Location();
}
```

3.2.7.1.1 Simplex Factory

```
Interface SimplexFactory under Root
{
Simplex Simplex(SetOfPoint);
}
```

Assertions:

1. Each set of n+1 points in 'general position' (in a coordinate system of dimension at least n) can define a simplex of dimension n
   $\forall$ P $\in$ SetOfPoint, $p_0 \in P$, n +1 = P.Count
   [{(p.Coordinate) - p_0 | p $\in$ P - {p_0} } are independent as vectors $\Rightarrow$
   Simplex(P) $\in$ n-simplex]
2. A point is in a simplex if and only if its coordinates are in the convex hull of the defining 'corner' points
   $\forall$ P $\in$ SetOfPoint = {p_0, p_1, …, p_{n-1}}, P.Count = n + 1,
   Simplex(P) $\in$ n-simplex, $\forall$ q $\in$ Point
   [q $\in$ Simplex(S) $\Leftrightarrow \exists\ x_i \in [0, 1]$, $0 \le i \le n$,
   $\sum x_i = 1.0 \wedge$ q.Coordinate = $\sum x_i\ p_i$.Coordinate

3. The boundary of an n-simplex is the collection of n (n-1)-simplexes that are spanned by the size n subsets of the corners of the original simplex.

$\forall P \in \text{SetOfPoint} = \{p_0, p_1, \ldots, p_{n-1}\},\ P_i = P - \{p_i\},$

$[S = \text{Simplex}(P) \in \text{n-simplex} \Rightarrow$

$(\forall i, 0 \le i < n\ ,\ \text{Simplex}(P_i) \in \text{(n-1)-simplex}) \wedge$

$(S.\text{boundary} = \{\ \text{Simplex}(P_i) \mid 0 \le i < n\ \}\ )\ ]$

A simplicial complex (complex of simplexes) is a set of simplexes (beginning with 0-simplexes, or points), in which the complex is closed under the boundary operator (the boundary of each simplex is a set of simplexes in the same complex.

```
Interface <n>-simplicialComplex under Root
{
SetOfSimplex Extent();
}
```

1. Every k-simplex in the complex has it boundary in the complex

$\forall k \in \mathbf{Z}^{+0}, 0 \le k \le n, \forall c \in \text{k-simplex}, c \in C \in \text{n-simplicialComplex} \Rightarrow$

$\{\partial c\} \subset C.\text{Extent}$

2. Every k-simplex in the complex is disjoint except at boundaries

$\forall k \in \mathbf{Z}^{+0}, 0 \le k \le n, \forall c, d \in \text{k-simplex}, c, d \in C \in \text{n- simplicialComplex},$

$c \ne d \Rightarrow c \bigcap d = \partial c \bigcap \partial d$

3. The intersection of two k-simplexes in C is a (k-1)-simplexes in C or is empty.

$\forall k \in \mathbf{Z}^{+0}, 0 \le k \le n, \forall c, d \in \text{k-simplex}, c, d \in C \in \text{n- simplicialComplex},$

$c \ne d \Rightarrow \{\partial c \bigcap \partial d\} = \{\partial c\} \bigcap \{\partial d\} \subset C.\text{Extent} \bigcap \text{(k-1)-simplex} \wedge$

$\{\partial c \bigcap \partial d\}$ contains at most 1 simplex

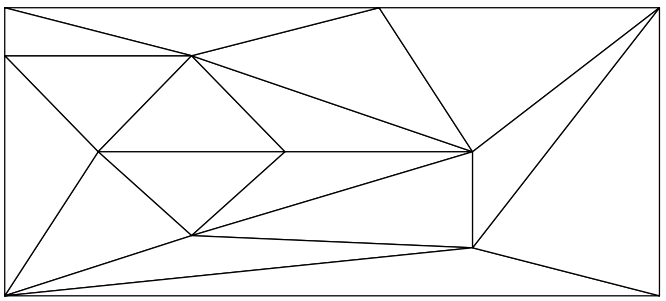


Figure 3-12 A Surface as a 2 Simplical Complex

3.2.7.2 Cell

A cell of any dimension is a set of points that can be isomorphically mapped into the appropriate standard cell of that dimension. “Isomorphic map” is one that is one to one, onto, and bicontinuous (it and its inverse are both continuous). The standard cells are a point, a line, a circle and its interior, a sphere and its interior, and the higher dimension discs and sphere defined earlier. The boundary of a cell can be expressed as almost any number of cells of one less dimension. For every dimension, we define the cell interface as follows:

```
Interface <n>-cell under Root
{
Point Chart(Coordinate)
SetOf<n-1>-cells Boundary();
}
```

- Chart() maps the standard $\mathbf{D}^{\mathbf{n}}$ to the cell

- Boundary() — returns the set of smaller cells that make up the boundary of the n-cell, written as ∂.

A cellular complex is a set of cells in which the boundary map is closed.

```
Interface <n>-Complex under Root
{
SetOfCells Extent();
}
```

1. Every $\mathbf{D^k}$ cell in the complex has it boundary in the complex
   $\forall k \in \mathbf{Z^{+0}}, 0 \le k \le n, \forall c \in$ n-cell, $c \in C \in$ n-complex $\Rightarrow \{\partial c\} \subset$ C.Extent
2. Every $\mathbf{D^k}$ cell in the complex is disjoint except at boundaries
   $\forall k \in \mathbf{Z^{+0}}, 0 \le k \le n, \forall c, d \in$ k-cell, $c, d \in C \in$ n-complex, $c \ne d \Rightarrow c \bigcap d = \partial c \bigcap \partial d$
3. The intersection of two $\mathbf{D^k}$ cells in C is a set of $\mathbf{D^{k-1}}$ cells in C
   $\forall k \in \mathbf{Z^{+0}}, 0 \le k \le n, \forall c, d \in$ k-cell, $c, d \in C \in$ n-complex,
   $c \ne d \Rightarrow \{\partial c \bigcap \partial d\} = \{\partial c\} \bigcap \{\partial d\} \subset$ C.Extent

3.2.8 Method Invokation

OGIS supports functions on geographical data. Geoprocessing functions can be defined either as part of a database or separate from any database. A database might define functions on new data types that it contains. A vendor might sell a set of geoprocessing functions that users can perform on their own data. Users might define a few new geoprocessing functions as part of the applications that they are building. But a user will not want to know where a function is defined. One geoprocessing function might call another without knowing whether it was built-in or defined by the user.

Most object models make the definition of operations part of the definition of objects. This makes it hard to define operations to work on existing data. These models also choose an operation based on the class of one object, but many geoprocessing functions depend on the classes of more than one object. For example, intersecting a circle and a line is different from intersecting a polyhedron and a line, which is different from intersecting a point and a fractal cloud, which is different from intersecting an arc and a chain. In the worse case, introducing a new kind of SpatialObject requires defining how it intersects with each of the other SpatialObjects. But in a distributed system, it is not even possible to know all the kinds of SpatialObjects, since there might be many new kinds of SpatialObjects being defined simultaneously.

The multi-method object model solves these problems. OGIS supports an object model in which methods are not necessarily part of objects and can be selected based on the classes of more than one object, i.e. it is a multi-method object model. It uses a type system developed by Chambers and Leavens [3-7].

3.2.8.1 Types

Types in OGIS are interface specifications. They are different from the implementation of an object, which is a Class.

Each type has a unique ID. A type can be the subtype of one or more other types, but the subtype graph is acyclic. The union of two types is their common supertype; the intersection of two types is their common subtype.

The type system also contains a set of interfaces. Each interface has a name, a return type, and a sequence of parameter types. Methods do not have to have unique names. There are usually many methods with the same name. A method *t* is compatible with a method *s* if *s* and *t* have different names, if the return type of *t* is a subtype of the return type of *s*, or if one of the parameter types of *s* is not a subtype of the corresponding parameter type in *t*. A set of methods is compatible if each method in the set is compatible with every other method in the set.

For example, suppose SpatialObject has subtype Surface, which has subtype Curve, which has subtype Point. The following set of methods would be compatible:

```
Intersect(SpatialObject, SpatialObject): SpatialObject
Intersect(Point, Point): Point
Intersect(Point, Curve): Point
Intersect(Curve, Curve): Curve
Intersect(Curve, Surface): Curve
Intersect(Surface, Surface): Surface
```

Each program has a current set of methods. A function call is only legal if there is a method in the current set with the same name and whose parameter types are supertypes of the types of the actual arguments of the function call.

3.2.8.2 Classes

Each class has a unique ID. A class can inherit from one or more other classes, but the inheritance graph is acyclic.

Each class conforms to one or more types. When you define a new class, you can assert the types to which it conforms. However, you can also assert that it conforms to a new type when you define the type.

Method definitions usually only work for particular classes. Thus, a method definition can specify the classes of its arguments. A method specifies the type of each parameter and optionally the class. Methods can be implemented in any programming language, so there is no particular syntax for methods. For the sake of concreteness, we will use the following notation:

```
function_name(arg1@type1: class1, ..., argk@typek: classk)
       :ret_type
```

Each method must match a method in the current set of methods. Its name must be the same, its parameter types must be a supertype of the method's parameter type, and its return type must be a subtype of the method's return type.

Note that anyone can define a method on any type and class. When you define a class and a set of methods for it in a particular language, you will often be able to use private information in the methods. Someone elsewho defines a method on that class will not be able to use that private information. For example, suppose you define a new kind of raster in C. Because you know the byte-order of the contents of the raster, and because you know that the raster and the methods you have defined for the raster are in the same address space, you can iterate over the raster very efficiently, and can perform operations on it very efficiently. However, anyone can define methods that access the new raster. The type "Raster" probably defines functions that someone could use to iterate over it, though it would be a lot slower to use these than to access the raster directly. You might define a new subtype of Raster when you define the new class. This new subtype might permit a few more operations than had existed before. However, even providing new operations for your new class will not make client programs as efficient as those that can access the class directly.

3.2.8.3 Gathering Methods

Methods can be defined in databases, be predefined by the implementation of OGIS, or be defined by the user. There must be a central FunctionRegistry that can tell which method to select for a particular set of arguments. The FunctionRegistry not only registers methods, it also registers methods. Whenever an application uses a dataset, it adds methods and methods from the dataset to the FunctionRegistry. It invokes services using the FunctionRegistry and can define new services or add to existing services using the FunctionRegistry.

Each dataset has a set of dictionaries that describes its contents.A dictionary can describe types, classes, methods, or methods. A dataset that has new methods or methods will include them in one of its dictionaries. The FunctionRegistry is essentially the union of the dictionaries of all the datasets that are being accessed. Thus, a programmer defines a new method by putting it in a dictionary. When an OGIS application accesses the dictionary containing the method (probably by opening the dataset using the dictionary) then the method is available to be invoked.

The FunctionRegistry must ensure that its information is consistent. It checks that the set of methods in the FunctionRegistry is compatible. If a method in the FunctionRegistry has the same name as a method and its parameter types are subtypes of the method's parameter types then the method's return type must be a subtype of the method's return type. For each method in the FunctionRegistry, there must be at least one method with the same name and with compatible parameter types. This can all be checked as soon as information from a dictionary is loaded into the FunctionRegistry.

3.2.8.4 Invoking Methods

The FunctionRegistry is responsible for invoking methods. Suppose that a function is being invoked. First, it must select all the methods with the same name and with parameter types that are each a supertype of the types of the actual arguments. From these, it selects those methods that have no other methods whose parameter types are all subtypes and at least one is not equal.

If there is no selected methods then the function call is in error. If there is only one such method, it is invoked.

If there are several such methods with identical types then the FunctionRegistry must pick one based on location or assigned priority. This part of the specification is still incomplete.

If there are several such methods but they do not have identical types then there are at least two methods M and N whose parameter types are supertypes of the actual argument types but M has a parameter type that is strictly a subtype of the corresponding parameter type of N, and N has a parameter type that is strictly a subtype of the corresponding parameter type of M. For the time being, we will consider these to be errors. Many multi-method based systems pick an arbitrary order, such as choosing the method whose first argument is the most specific.

3.2.8.5 Example 1

Consider how to implement an operation Display that displays a SpatialObject on a window. It doesn't make sense to make this operation be a method on SpatialObjects, because it will depend on the kind of imaging system being used, such as X, Windows, Display Postscript, or Quickdraw. Moreover, it doesn't make sense to make color and line width be attributes of SpatialObjects, yet something must define those attributes before a polygon can be rendered in a window. Thus, Display should be defined

elsewhere, yet it must work differently for lines than it does for filled polygons or circles.

One solution is to introduce a Window object and to make Display be a function with both a SpatialObject and a Window as an argument. The Window will keep track of a default color and line width. Vendors of window systems will provide display services for all the standard SpatialObjects on their subclasses of Window. GIS vendors will provide Display methods for their complex SpatialObjects on Window in terms of the standard SpatialObjects. This will make it possible for any SpatialObject to be displayed on any Window.

3.2.8.6 Example 2

Consider how to implement the Intersect operation on SpatialObjects. It must be able to compute the intersection of any SpatialObject with any other, in spite of the fact that we do not know all the kinds of SpatialObjects and the algorithm for computing an intersection depends on the kinds of SpatialObjects being computed.

One solution is to make a new kind of SpatialObject called an Intersection. An Intersection has a set of components that are SpatialObjects, and it represents the intersection of those components. Every SpatialObject must know how to intersect itself with a few basic kinds of SpatialObjects, such as Points, Curves, Surfaces, and Volumes. Intersecting other kinds of SpatialObjects results in an Intersection of them. The Intersection must implement all the operations of a SpatialObject. It intersects itself with the basic kinds of SpatialObjects by intersecting them with each of its components and returning whatever is left. It intersects itself with other SpatialObjects by adding them to the Intersection.

Unfortunately, it can be less efficient to build an Intersection and repeatedly evaluate it than to just perform the intersect operation. The intersection of two areas is often best represented by an Intersection object, but the intersection of a area and line is best represented by a a set of points (depending on the nature of the intersection). Therefore, we will need to define the intersect operation on many different combinations of objects. We will need to let vendors and users define how to intersect the objects that they create with other objects. If users get different SpatialObjects from different vendors, it is likely that the vendors didn't know about the SpatialObjects created by the other vendors. They can use the default "create an Intersection object" definition for intersection, which will often be good enough. But if it isn't good enough, the users will have to define intersection on these new combinations.

Note that methods are always picked by class, not by type. Consequently, it will be common to define an abstract class for a type so that method can be defined for that class and inherited by every class that conforms to the type. However, it is often more efficient to define intersection directly for the new class. For example, a RadiusCircle is a Circle that is specified by its center point and a radius. The standard interface to Circle is the diameter and the center. Converting from one to the other requires calculation. A vendor that introduced a RadiusCircle might want to redefine the intersect operation to avoid doing the conversion.

3.2.8.7 Types

Given the background above, the following sections describe the interface for the Method and FunctionRegistry.

3.2.8.7.1 Method

A method is a specialization of the Interface type (see 3.1.2.5.1.1). The only additional information needed is the host on which the implementation exists.

```
Interface Method under Interface
{
Host   Host();
Root   Invoke(Name, ListOfTuple, Tuple);
}
Interface MethodFactory under Root
{
Method Method(Name, Tupletype, Tupletype, Host);
}
```

3.2.8.7.2 FunctionRegistry

```
Interface FunctionRegistry under ListOfMethod
{
CharacterString     Name();
ListOfMethod        GetMethods();
Boolean             DeleteMethod(Name);
ListOfMethod        FindMethod(Name);
Root                Invoke(Name, ListOfTuple, Tuple);
}
```

3.3 Future Work

The specification will require the filling in of details for specific application areas. Especially important are:

- plain language descriptions of the intent of the interfaces and assertions,
- development or adoption of a reasonable syntax for expressing interfaces and assertions formally and unambiguously,
- definition of well-known types for coverages, especially for BooleanPredicate, raster images and 'Tomlin' cartographic coverages.

There is obviously more work to do on methods and the method registry. Some of the issues are listed here.

i) There should be a mechanism to provide invokation policy to the FunctionRegistry when more than one method matches the invokation parameters.

ii) Do we need a FunctionRegistryFactory or do we assume its existence on all hosts that are providing support for OGIS?

iii) What is the exact policy within OGIS for registering methods?

  a) Do all OGIS implementations create one when they are installed on a host?

  b) Do the append all available methods to an existing registry if other OGIS implementations are already installed?

  c) Do object instances of OGIS types install all of their interfaces in the method registry?

iv) Should there be an interface that stringizes methods for transfer between machines? e.g.,

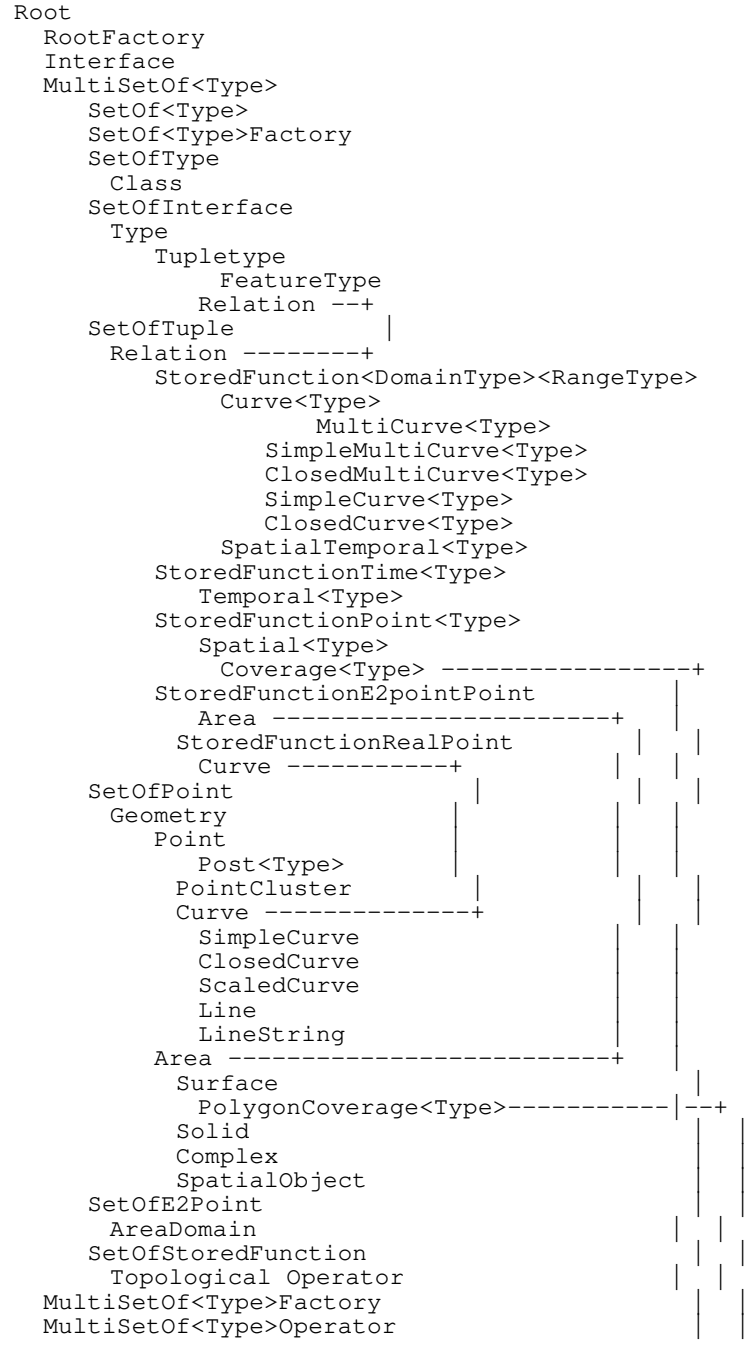
```
Name(Tupletype1.Name,Tupletype2.Name,...,Tupletypen.Name):Tuplety
       pereturn.N
```

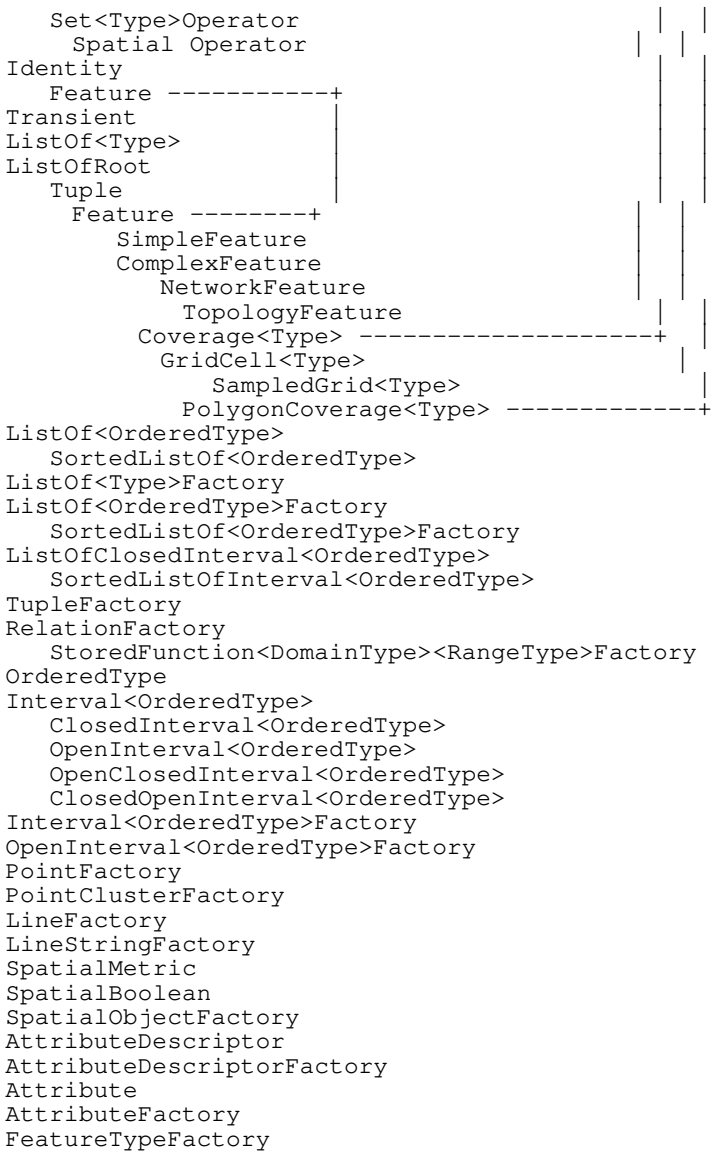
Appendix A: Type Hierarchy

The following is the type hierarchy as described in Chapter 3.

Ed. Note: This appendix needs to be arranged in a more readable way. Expect to see this change for the better soon.

The connected Types indicate multiple inheritance.




Set<Type>Operator
Spatial Operator
Identity
Feature
Transient
ListOf<Type>
ListOfRoot
Tuple
Feature
SimpleFeature
ComplexFeature
NetworkFeature
TopologyFeature
Coverage<Type>
GridCell<Type>
SampledGrid<Type>
PolygonCoverage<Type>
ListOf<OrderedType>
SortedListOf<OrderedType>
ListOf<Type>Factory
ListOf<OrderedType>Factory
SortedListOf<OrderedType>Factory
ListOfClosedInterval<OrderedType>
SortedListOfInterval<OrderedType>
TupleFactory
RelationFactory
StoredFunction<DomainType><RangeType>Factory
OrderedType
Interval<OrderedType>
ClosedInterval<OrderedType>
OpenInterval<OrderedType>
OpenClosedInterval<OrderedType>
ClosedOpenInterval<OrderedType>
Interval<OrderedType>Factory
OpenInterval<OrderedType>Factory
PointFactory
PointClusterFactory
LineFactory
LineStringFactory
SpatialMetric
SpatialBoolean
SpatialObjectFactory
AttributeDescriptor
AttributeDescriptorFactory
Attribute
AttributeFactory
FeatureTypeFactory

List of Acronyms

This section will contains a list of acronyms used within the document.

API - Application Program Interface

DCE - Distributed Computing Environment

COM - Common Object Model

CORBA - Common Object Request Broker Architecture

CSI - Communication Service Interface

EEI - External Environment Interface

GII - Global Information Infrastructure

HTI - Human-Technology Interface

ISI - Information Storage Interface

NGDC - National Geospatial Data Clearinghouse

NII - National Information Infrastructure

OGIS - Open Geodata Interoperability Specification

OGM - Open Geodata Model

OGS - Open Geoprocessing Services

OMG - Object Management Group

OSE - Open System Environment

OSF - Open Software Foundation

RPC - Remote Procedure Call

SII - System Internal Interface

Glossary

This section contains definitions for as many terms as possible, as utilized in the text of the specification.

Agent - A kind of intermediary service which acts on behalf of another service (service provider or requester) according to rules established upon its invocation. Also known as an "intelligent agent". [2-9].

Application - The use of capabilities, including hardware, software and data, provided by an information system specific to the satisfaction of a set of user requirements [2-10]. See Geographic Application and Geoprocessing Application.

Application Developer -

Application Platform - The collection of hardware and software components that provide the infrastructure services used by application programs; APIs make the specific characteristics of the platform transparent to the application.

Application Portability - The ability to move software among computers without rewriting it. This may be provided in three ways, as source code portability, pseudocode portability, or binary code portability. See User Portability

Application Program Interface (API) - An interface definition that permits writing OGS services for application programs without details of their internal implementation; defines the behavior of a OGS service. In the OSE model, the interface between the application software and the application platform, across which services are provided [2-11].

Application Software - The computing elements supporting users' particular needs; includes data, documentation, and training, as well as programs.

Architecture - An abstract technical description of a system or collection of systems. Conceptually based. Does not contain the level of detail needed for construction.

Architectural Framework - Identifies key interfaces and services, and provides a context for identifying and resolving policy, management and strategic technical issues; constrains implementation by focusing on interfaces, but does not dictate design or specific technical solutions [2-12]. [Ed. Note: not to be confused with a Framework!]

Architecture - A documented implementation of an architectural framework designed and constucted by an engineering organization from selected technologies [2-12].

Base Earth Model - A common, logically consistent definition of the Earth Model within OGM; includes a geodetic and temporal mathematical reference frame, and the components of Surface Configuration, Surface Features and Volumes which are independent of OGIS applications and have unambiguous definitions.

Base Standard - An approved International Standard, Technical Report, CCITT Recommendation or National Standard [2-10].

Broker - A kind of intermediary service whose responsibility is only to bring other services together (typically a service requester and a service provider) and has no responsibility for satisfactory completion of the "contract" established between the requester and provider. [2-9]

Communications Service Interface (CSI)- The interface where an application platform accesses external entities which provide data transport services. The service provided is data transport among application platforms. [2-12]

Computer Architecture - The functional composition of a system and its components, the interfaces between components, and interfaces with the external environment, including users and other systems.

Computer Environment - The general term describing the people, hardware, software, and databases comprising a single computer system or several network-connected computer systems, and the associated standards.

Distributed Computing Environment (DCE) - A collection of software components that support interoperability and transparency across multiple heterogeneous platforms in various network configurations.

Diversity - The ability for a system or components of a system to support diversity in behavior, function and data types.

Domain - System context: A class of systems which have similar requirements and capabilities. Application context: The body of knowledge defining the range and scope of an application in terms of elements, rules and behaviors.

Earth Model - The approach(es) to abstracting the earth; the data model for the earth.

Emerging Standard - A specification that is under consideration by an accredited standards development organization, but has not completed the process of approval by the sponsoring body [2-10].

Extensibility - The ability for a system or components of a system to expand by assimilating new data, software or hardware components.

External Environment - A set of entities external to the application platform with which services are provided. External entities include people, exchangable media not mounted in the platform, communication wiring and other platforms [2-11]

External Environment Interface (EEI) - An interface for data interchange with OGM. In the OSE model, the interface between the application platform and the external environment across which services are provided [2-11]

Flexibility - The ability for information technology to be modified, extended or reconfigured to accomodate new application requirements.

Framework - A reusable software template, or skeleton, from which key enabling and supporting services can be selected, configured and integrated with application code. [2-9] [Ed. Note: not to be confused with an Architectural Framework!]

Geographic Application - Applications which pertain to the earth and earth phenomena, with known spatial and temporal reference systems. Expressed in a human context versus computer context.

Geoprocessing Application - Computer applications which model, interpret and use earth information; the implementation of a Geographic Application on a computer.

Geospatial Data - Any earth features or earth phenomena defined in terms of spatial and temporal coordinate systems.

Geospatial Function (or Process) - Any function or process which handles or operates on geospatial data.

Human Technology Environment - The environment within which people interact with information technology.

Human Technology Interface (HTI)- The interface across which people interact with information technology. The service provided through this interface is access to the information infrastructure and to other people. [2-12]

Information Appliance - end-user equipment having input and display capabilities for communication with other users or service providers in the NII.

Information Storage Interface (ISI)- The interface across which information technology interacts with external storage media. The service provided through this interface is persistent storage of data, where the physical storage media is often removeable [2-12].

Implementation - A software package that conforms with a standard or specification.

Interface - A shared boundary between two functional entities. A standard specifies the services in terms of the functional characteristics and behavior observed at the interface. The standard is a contract in the sense that it documents a mutual obligation between the service user and provider and assures stable definition of that obligation [2-10].

Intermediary - A service that provides functions by which to interconnect, adapt and facilitate services offered by other parties, components or environments. Common forms of intermediaries include agent, broker, mediator and trader services [2-9].

Interoperability - The ability for a system or components of a system to provide information portability and interapplication, cooperative process control.

Language Independence - A standard or specification which is not specified in terms of a specific programming language.

Legacy System - Software or database components inherited from a previous computing model which do not fit into an open system environment without some modification; in the case of OGIS, modified to include an OGIS-compliant interface.

Manageability - The ability to configure, reconfigure, manage and control a system or the components of a system.

Mediator - A service which acts as an intermediary capable of negotiating aspects of the service to be provided impartially to meet goals of both a service requester and a service provider. The mediation function often follows broker, trader or agent functions [2-9].

OGIS Application - an application platform for geoprocessing applications that references the OGM, OGM features and OGIS services.

OGIS Application Domain - The part of the OGIS Reference Model which defines the Earth Model and Geographic Application Model for OGM.

OGIS Application Environment - A computer environment for OGIS applications; an OGIS environment, including the associated standards.

OGIS Application Environment Profile - A standards profile that specifies an OGIS environment and supports a specific class of Geographic Applications.

OGIS Computing Model - The part of the OGIS Reference Model which defines the conceptual computer architecture, properties, behavior, and geoprocessing application context for OGM and its environment.

OGIS Geodata Model (OGM) -

OGIS Reference Model - There are three parts to the OGIS Reference Model: (1) OGIS Application Domain Model; (2) OGIS Computing Model; and (3) OGIS Standards Model.

OGIS Standards Model - The part of the OGIS Reference Model which defines the community standards and standards profiles which comprise the complete OGIS framework.

OGM Feature - The OGM representation of a Surface Feature.

OGM Feature Layer - A subset of a OGM Feature Level.

OGM Feature Level - A collection of OGM Features which form a logically consistent grouping, in terms of resolution, accuracy, content and context.

OGIS Service - A software component (possibly a service of an application platform) providing object management, access, manipulation, interchange, or human-technology services for OGM features; also Open Geoprocessing Services (OGS).

Open Geoprocessing Services (OGS) - See OGIS Service.

Open Specification - A specification consistent with international standards, updated to conform with new standards, accomodates new technologies, and maintained by a public open consensus process. An open specification is usually also a de facto standard.

Open System - A system that implements open interface specifications and standards that promote application portability, scalability, interoperability, diversity, manageability, extensibility, compatibility with legacy components, and user portability.

Open System Environment (OSE) - A computer environment specified by a set of standards and profiles for interfaces, services, and formats for an open system [2-11].

Platform External Environment - Consists of those system elements which are external to the application software and the application platform; EEIs enable information exchange between systems executing on other platforms [2-11].

Profile - A collection of standards, with parameters, options, classes, or subsets, necessary for building a complete computer system, application or function.

Protocol - A set of semantic and syntactic rules that determine the behavior of entities that interact. [2-10].

Reference Model - Provides the complete scientific and engineering contextual framework for a technology area. The underlying elements, rules and behaviors.

Remote Procedure Call (RPC) - An API for remote execution of detailed functions.

Scalability - The ability to change the component configuration of a system to fit desired application contexts.

Service - A computation performed by an entity on one side of an interface in response to a request made by an entity on the other side of the interface [2-10]

Specification - A document that specifies a technological area with a well-defined scope, by a consortium, vendor or user, and is not a formal standard.

Standard - A document that specifies a technological area with a well-defined scope, by a formal standardization body and process.

Surface Configuration Model - Defines the geometric characteristics of the earth's surface, exclusive of features which fall upon the surface; defined in terms of elevation, shape, roughness, slope, and aspect, with the later properties possibly derived from elevation.

Surface Feature - An entity which lies on the earth's surface or is referenced to the earth's surface.

System Internal Interface (SII) - An interface between components within an application platform.

Tool - A software component, sometimes called an application object, which can act as either a service provider or service requester within an application platform.

Trader - A kind of intermediary service which acquires services from one or more providers for "resale" to a service requester. The trader service insulates requester and provider services from having to interact directly with one another. The trader is responsible to the requester for all aspects of the requested service [2-9].

Transparency - The ability of systems or components of systems to hide the details of their implementations from other client or server systems or components of systems.

User Portability - The ability of a user to move from one system to another without having to learn everything again.

Validation - The process of testing an application or system to ensure that it conforms to its specification [2-10].

References

This section contains references mentioned in the body of the specification. They are arranged by the chapter in which they appear.

Chapter 2: OGIS Reference Model

[2-1] An Architectural Framework for the National Information Infrastructure 1994. *An Architectural Framework for the National Information Infrastructure*, Technical Report of the Cross-Industry Working Team.

[2-2] OGIS Project Document 94-002. OGIS Technical Vison, Goals, and Requirements, The Open GIS Foundation, 1 Kendall Square, Suite 2200, Cambridge, Massachusetts.

[2-3] OGIS Project Document 94-007. OGIS Features and Coverages, The Open GIS Foundation, 1 Kendall Square, Suite 2200, Cambridge, Massachusetts.

[2-4] Quarterman 1993. Quarterman, J. S., and Wilheim, S., *UNIX, POSIX, and Open Systems*, Addison-Welsey Publishing Company, New York, 1993.

[2-5] Common Object Model Specification 1994. Draft Version 0.1, Microsoft Corporation and Digital Equipment Corporation.

[2-6] Common Object Request Broker Architecture 1994. Version 2.0, Object Management Group.

[2-7] DODIIS Reference Model 1991. Department of Defense Intelligence Information Systems Reference Model, Defense Intelligence Agency.

[2-8] Reference Model for Frameworks of Software Engineering Environments. 1991. Technical Report ECMA TR/55, 2nd Edition. National Institute of Standards and Technology, U.S. Department of Commerce.

[2-9] Framework for National Information Infrastructure Services. 1994. NISTIR 5478. National Institute of Standards and Technology, U.S. Department of Commerce.

[2-10] NII/GII Requirements Input from IEEE P1003.0. 1994. IEEE P1003.0/D17. pp. 11-19.

[2-11] Application Portability Profile - The Government's Open System Environment Profile OSE/1 Version 2.0, 1990. NIST Special Publication 500-210.

[2-12] Open System Environment Architectural Framework for National Information Infrastructure Services and Standards, Draft 1.0, August 5, 1994. Developed by NIST/DSEP and the Information Infrastructure Task Force (IITF) Technology Policy Working Group.

3573 The references below have been chosen both for specific citations and for their
3574 general applicability to the subject of features and coverages. Especially useful is
3575 Maguire, Goodchild and Rhind [3-5] which has extensive bibliographies.

3576 [3-1] Burrough, Peter A. (1986), **Principles of Geographical Information Systems**
3577 **for Land Resources Assessment**, Clarendon Press, Oxford, 194 pp.

3578 [3-2] Egenhofer, Max J., and John R. Herring (1990) "A Mathematical Framework
3579 for the Definition of Topological Relationships," **Fourth International**
3580 **Symposium on Spatial Data Handling**, K. Brassel, ed., Zurich, Switzerland, pp
3581 803-813.

3582 [3-3] Laurini, Robert and Derek Thompson (1992), **Fundamentals of Spatial**
3583 **Information Systems**, Academic Press, London, 680 pp.

3584 [3-4] Mark, David M., and Andrew U. Frank (1991), **Cognitive and Linguistic**
3585 **Aspects of Geographic Space**, Proceedings of the NATO Advanced Study
3586 Institute on Cognitive and Linguistic Aspects of Geographic Space, Las Navas
3587 del Marquis, Spain, July 9-20, 1990, Kluwer Academic Publishers, Dordrecht,
3588 The Netherlands, 519 pp.

3589 [3-5] Maguire, David J., Michael F. Goodchild, and David W. Rhind (1991),
3590 **Geographic Information Systems**, Longman Scientific & Technical, Harlow,
3591 England, 2 volumes, 649pp + 447 pp

3592 [3-6] Tomlin, C. Dana, (1990), **Geographic Information Systems and**
3593 **Cartographic Modeling**, Prentice Hall, Englewood Cliffs, NJ, 249 pp.

3594 [3-7] Chambers, Craig and Gary T. Leavens, (1994). "Type checking and Modules
3595 for Multi-Methods", to appear in Proceedings of OOPSLA'94, Portland,
3596 October.

3597 [3-8] Federal Geographic Data Committee, (1994). "Content Standards for Digital
3598 Geospatial Metadata," June 8, 1994 version, available via anonymous ftp from
3599 fgdc.er.usgs.gov:gdc/metadata

Index

This section will contain an index of important terms and concepts within the specification.