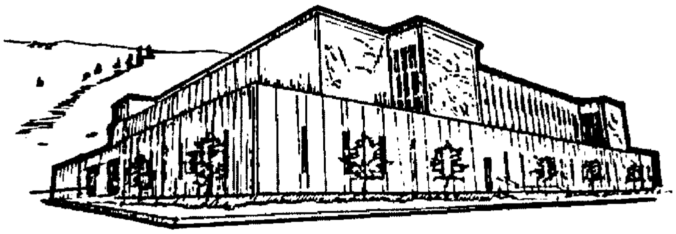
1993

An object oriented domain analysis of ecosystem modeling

Ronald Lee Righter

The University of Montana



Maureen and Mike MANSFIELD LIBRARY

The University of **Montana**

Permission is granted by the author to reproduce this material in its entirety, provided that this material is used for scholarly purposes and is properly cited in published works and reports.

*** Please check "Yes" or "No" and provide signature***

Yes, I grant permission ✓

No, I do not grant permission ___

Author's Signature Ronald L. Rigby

Date: 12/13/93

Any copying for commercial purposes or financial gain may be undertaken only with the author's explicit consent.

AN OBJECT ORIENTED DOMAIN ANALYSIS
OF
ECOSYSTEM MODELING

by

Ronald Lee Righter
B. A., Elizabethtown College, 1972

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science

The University of Montana

1993

Approved by

Chairman, Board of Examiners

Dean, Graduate School

Dec. 14, 1993
Date

Righter, Ronald Lee, M. S., December 1993 Computer Science

An Object-Oriented Domain Analysis of Ecosystem Modeling

Director: Ray Ford

An object-oriented methodology is used as the basis for a domain analysis of computer-based ecosystem modeling. Requirements analysis of the application domain serves as the basis for domain analysis, which produces a hierarchy of classes, each characterized by a set of properties. The hypothesis that such an analysis could provide a useful set of knowledge about the domain is tested.

Requirements analysis of the application domain identifies several representations of spatial phenomena and a set of significant modeling processes. A particular representation of spatial phenomena is chosen as the basis for domain analysis. A classification of 23 key modeling processes identifies several types of operations: data acquisition, cartographic transformation, creation of higher level model structure from undifferentiated datasets, algebraic manipulation of models to derive new variables, interpolation and extrapolation of data values, process simulation, and enhancement of models for visual display and analysis.

Domain analysis produces a set of class specifications, a set of diagrams showing the relationships among classes, and a set of diagrams illustrating how each key modeling activity can be performed in the context of the class hierarchy. The hierarchy contains 169 classes, which are divided into twelve groups: root entities, spatial entities, ecosystem entities, ecosystem descriptor entities, classification entitities, operations, data acquisition entities, descriptor entities, window entities, software entities, documentation entities, and human entities.

Diagrams are successfully completed, depicting the execution of 19 of the 23 key modeling processes. The hypothesis is considered to be successfully validated. Suggestions for future work are made.

ACKNOWLEDGEMENTS

Acknowledgements and thanks go to Dr. Roland Redmond, leader of the Wildlife Spatial Analysis Lab in the Montana Cooperative Wildlife Research Unit at the University of Montana, and his staff and students. Dr. Redmond also receives thanks for his contribution as a thesis committee member and for financial support. Dr. Steve Running, leader in the Numerical Terradynamics Simulation Group at the University of Montana School of Forestry and his staff and students provided much help and encouragement during work on this thesis. Dr. Running also provided financial support. Dr. Jim Ullrich of the Dept. of Computer Science at the University of Montana served as a thesis committee member. And finally, Dr. Ray Ford of the Dept. of Computer Science at the University of Montana provided many kinds of support and encouragement, including financial support. Dr. Ford also served as chairman of my thesis committee, and perhaps most importantly, invited me to participate in the development of the Ecosystem Information System.

This thesis is an attempt to integrate the results of a year and a half journey through fields far removed from computer science. This document is an illustrated account of that journey; it is hoped that the material presented here will be useful in the construction of the Ecosystem Information System, and will be useful to ecosystem scientists who are receptive enough to listen to a visitor who wandered through their world for a time.

Table of Contents

List of Tables

List of Figures

CHAPTER I

PROBLEM STATEMENT

Introduction

Researchers in the ecological sciences are developing techniques for modeling and analyzing ecosystems using digital computers. The increasing power of the computer allows researchers to work at larger spatial scales and finer spatial resolutions than was previously possible. Modeling efforts can also be carried out at varying time scales, ranging from static views of an ecosystem to century long modeling of some processes. Workers in this application domain must be well-versed in a number of disciplines. Primary among them are plant science, ecosystem ecology, community ecology, climatology, meteorology and hydrology. Understanding of fundamental geographic principles is indispensable, as many of the datasets that researchers work with are two-dimensional representations of phenomena that occur in three-dimensional space.

Ecosystem Research

Modeling and analysis of ecological phenomena can be carried out for more than one purpose. In some cases, researchers use ecosystem simulation models for hypothesis testing. In other cases, model outputs are used by land managers in their decision-making process. At the University of Montana, several labs are engaged in different aspects of ecosystem modeling . The Wildlife Spatial Analysis Lab (WSAL) at the Montana Cooperative Wildlife Research Unit is responsible for the Montana Gap Analysis project (MT-GAP). Gap Analysis is a nationwide effort by the U. S. Fish and Wildlife Service to idenfity and protect biodiversity using a computerized geographic information system (GIS). The Numerical Terradynamic Simulation Group (NTSG) in the School of Forestry is partially funded by NASA and does ecosystem simulation on spatial scales ranging from the continental to a single watershed.

Data and Software Management Problems

The focus of this thesis will be the modeling requirements for the MT-GAP and NTSG projects. The research efforts in both labs overlap, and both projects use many of the same datasets and software tools. Additionally, both labs share many of the same problems. A major problem is the enormous size and number of datasets utilized in some projects. For example, a Thematic Mapper image from the Landsat satellite will typically require 350 to 400 MB of storage. The MT-GAP project requires 31 of these images to cover the state of Montana. The size and number of these datasets presents a formidable data management and processing problem. Researchers must often devise *ad hoc* and sometimes unsatisfactory solutions to these problems.

Many different software tools - both commercial off the shelf (COTS) and custom-made - are used in ecosystem modeling. Two particular types of software package have proven especially useful. Geographic Information Systems (GIS) facilitate the acquisition, management, analysis and display of data relating to phenomena located in geographic space. Image Processing (IP) systems are used to analyze digital data that is collected from remote-sensing devices such as satellites and aircraft. Commercial products of both types are used extensively in both WSAL and NTSG. These products, however, usually are developed for fairly large markets and thus emphasize functionality that is in fairly widespread demand. Workers who are doing cutting edge research and/or dealing with unique problems may find that their software and data management needs often exceed the capacity of commercially-available software. For this reason, in-house software and data management tools are widely used for many modeling and analytical problems. Construction of these software tools often requires considerable investment of time from ecological researchers, and therefore is a distraction from the scientific work they might rather be doing. Additionally, these tools sometimes are of an *ad hoc* nature. That is, they are sometimes built as limited solutions to a particular problem, and therefore may not be applicable in a broader context.

Ecosystem Information System

The Ecosystem Information System (EIS) was conceived as a remedy to many of these data management and processing problems (Ford92). EIS will allow researchers working in a distributed environment (i.e., in physically separate labs) to share data and software. EIS will provide a researcher with a mechanism to browse a collection of data-sets and software (both COTS and in-house), and obtain the information necessary to use the data and software in an appropriate manner. A critical step in the development of EIS is construction of a hierarchical description of the entities (software and data) available in the participating labs. Construction of this hierarchy is a first step in the development of any non-trivial, ecosystem database. The EIS hierarchy will therefore serve as a guide for researchers who need to find what datasets and software tools are available for their work. It will also provide a context for further software development by facilitating reuse of existing software designs and implementations. The construction of this hierarchy is the primary objective of this thesis.

CHAPTER II

HYPOTHESIS

Statement of Hypothesis

Our hypothesis is that a domain analysis model can effectively describe the entities and relationships between entities that are found in this application domain. The technical basis for our modeling work is an object-oriented modeling methodology. In this chapter we will describe object-orientation and domain analysis; we will also specify the criteria by which we will validate our hypothesis.

The Nature of Complex Systems

In what is one of the primary references in the object-oriented software development literature, Booch describes the nature of complex systems that include software components (Booch92). He describes industrial-strength software as software for which "it is intensely difficult, if not impossible, for the individual developer to comprehend all the subtleties of design." (Booch92, pp. 13). This is true of software systems designed for the modeling and analysis of geographical data. Booch characterizes a complex system as one that has a number of attributes, such as the following.

1) A complex system is in the form of a hierarchy in which a particular subsystem is itself composed of subsystems.

2) Hierarchic systems are usually composed of only a few different kinds of subsystems that are organized in various patterns.

3) Relationships among separate components are weaker than relationships among the internal parts of a particular component.

From these attributes, Booch derives a canonical form for a complex system. Such a system must be viewed from two perspectives, each of which is formalized as a hierarchy.

The *part of* hierarchy is based on decomposing the system into parts. A component in the *part of* hierarchy is made of sub-components which are found below it in the hierarchy. The *kind* of hierarchy is based upon generalization of properties. A component in this hierarchy has properties common to all components below it in the hierarchy. Booch refers to these hierarchies as object structure and class structure, respectively.

The Nature of System Decomposition

A key notion in the design of software is that of *decomposition.* A complex software problem can be broken into smaller pieces, each of which can be dealt with independently. Booch describes two approaches to decomposition -- algorithmic and object-oriented. *Algorithmic* decomposition divides a system into units, each of which represents a major step in a process. Each of these units can also be decomposed algorithmically. A more recent approach is *object-oriented* decomposition. Here each unit represents a key abstraction in the application domain. In this approach, the world is viewed as a collection of semi-autonomous entities that interact to exhibit higher level behaviors. Booch says that although both approaches are valuable, object-oriented systems tend to be smaller, more resilient, and less risky to develop, because their development can more easily be implemented incrementally.

The Object Model

There are four essential elements for the object model (Booch92, pp. 39-40) The first is *abstraction,* which "denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer." Abstraction provides information about what an object does while it conceals the means by which the object performs operations. The abstraction of an object provides an interface for the object, and identifies its essential behavior. The behavior of an object includes operations it performs on other objects, as well as operations that other objects perform upon it. An object that uses the resources of another object is that object's *client*. An object's *protocol* is the set of operations that its clients may perform upon it.

The second element of the object model is *encapsulation*, defined as "the process of hiding all the details of an object that do not contribute to its essential characteristics." (Booch92, pp. 46) Abstraction and encapsulation are complementary concepts. Encapsulation, also known as information hiding, prevents clients from seeing the inside view or implementation of an object.

Modularity is "the property of a system that has been decomposed into a set of cohesive and loosely coupled modules." (Booch92, pp. 52) A module is a higher level abstraction than a class. It is a construct used in system design and implementation, and serves as a way to manage source code. *Coupling* is the measure of the strength of association between modules or classes. A strongly coupled system is harder to understand than one with weaker coupling. *Cohesiveness* measures the degree of connectivity between the elements of a module or class.

The last of the major concepts of the object model is *hierarchy*, which is "a ranking or ordering of abstractions." (Booch92, pp. 54)

Another attribute of object-oriented models is *persistence*, described by Booch as "the property of an object through which its existence transcends time (i.e., the object continues to exist after its creator ceases to exist) and/or space (i.e., the object's location moves from the address space in which it was created)" (Booch92, pp. 70). Though some advocates of object-oriented design consider persistence as an essential aspect of object-oriented systems, Booch considers it a minor aspect of the object model. In this application domain, persistence through time is obviously important. Virtually all software tools used by researchers (both COTS and custom-made) provide some mechanism for storing objects permanently. This mechanism may be a simple "write to disk" or may involve recording entities in a database management system. For software that functions in a distributed environment, persistence in space is critical as well.

Booch next characterizes an *object*, which is a concept fundamental to the object-oriented methodology, as an entity with state, behavior and identity. The concept of state embraces all the properties of an object where a property is a distinctive feature or quality that helps to make the object unique. Properties have a value, which may be a simple quantity or a reference to another object. An object's behavior determines how its state changes in response to the actions of other objects. Behavior also controls the way in which an object acts upon other objects. Identity is the property of an object that makes it unique. A set of objects that share a common structure and a common behavior is called a *class*. An object is an instance of a class.

In Booch's view there are two kinds of relationships among objects -- *using* and *containing*. In *using* relationships, an object uses another object. Objects involved in this kind of relationship play one of three roles. *Actors* use other objects, but are never themselves used. *Servers* are used by other objects, but never themselves use other objects. Finally, an *agent* is an object that can be both an actor and a server. In *containing* relationships, one object contains other objects.

Relationships among classes are more involved. Booch describes three basic kinds. *Generalization,* or *kind of,* relationships indicate sharing of properties. *Aggregation,* or *part of,* relationships indicate structural relationships. Finally, a semantic connection between otherwise unrelated classes is an *association.* Booch's methodology provides four relationships among classes. *Inheritance* relationships are the most powerful and can be used to express generalization, i.e., that a class shares the structure or behavior defined in one or more other classes, called *superclasses.* An inheriting class is called a *subclass.*

Inheritance is the relationship used to build the kind of hierarchy discussed earlier. *Using* relationships among classes can be of two types. A class can use another class in its *interface* or in its *implementation*. *Instantiation* relationships indicate that an instance of a class involved in this relationship cannot be created without creating instances of the other class in the relationship. Finally, *metaclass* relationships allow classes to be seen as objects. A metaclass is a class whose instances are themselves classes.

Booch states that no single diagram or model can capture all the details of a system. His methodology requires construction of two kinds of models - logical and physical. The *logical* model depicts classes, objects and the relationships between them. The *physical* model depicts higher level subdivision of the system into modules and the allocation of processes to processors, then processors to competing demands. Logical models are built first, and then used as a basis for construction of the physical models. In this work, we will build only logical models, according to the two stages of the Booch methodology as discussed in White (White93) -- requirements analysis and domain analysis.

Requirements Analysis

Requirements analysis is essentially the process of determining what the customer expects from the system that is being built. Among the products are a clear definition of a system's reason for being and a statement of overall goals. This stage comprises a contract between developer and user, recognizing that the contract will evolve over time. In some design methodologies, the only formal product of this stage is a simple statement of the primary functions of the system. In our modeling effort, however, system requirements will be described in more detail, with summaries of many of the kinds of modeling and analysis activities the system must support.

Domain Analysis

Domain analysis is the process of building an object-oriented model of the portions of the real-world enterprise that are relevant to the system being designed. White says it is through this process that developers gain the detailed knowledge of the domain needed to have the system carry out its required functions. Domain analysis identifies all major objects in the domain, including state components and major operations. The result is a central model containing all system semantics. White says this process focuses on "resolving that bewildering set of aliases, contradictory requirements, obscure policy, and varying styles of explanation and communication into a ... structure that will map directly to final implementation." (White93, pp. 7) Our domain analysis will produce three sets of deliverables:

1) class-relationship diagrams, which identify the key classes and relationships among classes in the domain;

2) class specifications, which contain all semantic definitions of the classes, attributes and key operations; and

3) object-scenario diagrams, which illustrate how the objects will interact to carry out key system functions.

White suggests the following sequence for domain analysis. First define classes, then identify major relationships between classes and objects. Next, identify key attributes. At this point, identify properties shared among classes to produce a class hierarchy, then identify operations. Finally, validate the model by appropriate means, and repeat the steps above to refine the model.

We will use the object-scenario diagrams produced by domain analysis to evaluate our model with respect to our hypothesis. That is, if we can successfully build an object-scenario diagram for each scenario identified during requirements analysis, we will have validated our hypothesis.

The Object-Oriented Methodology in this Thesis

The object-oriented methodology can be employed in several ways for this analysis. We will discuss two distinct approaches and then present a third approach, which will serve as the basis for our work. In the first approach, we can examine the collection of spatial datasets, software and other tools used by ecosystem modelers, then construct an object model depicting the nature of these objects and the relationships among them. In this approach we make no reference to the ecosystem. Our work falls more strictly under the umbrella of conventional software modeling and design. Our model will be limited because it will not capture much of the semantics of the application domain.

In the second approach we apply the object-oriented methodology to the ecosystem itself. In this case we view the ecosystem as a collection of interrelated and interacting objects (i.e., as a complex system). In this case our work falls into the domain of ecosystem modeling. This approach is being pursued by a number of ecosystem researchers (Silvert93, Raper93, Bennett93, and Klanika93). It is not our intention to build an object-oriented model of an ecosystem. Instead we will model the datasets, software and tools used in ecosystem modeling, while recognizing the ecosystem as the focus of concern.

Our approach will be as follows. We will view the ecosystem as a complex system (Booch89). The system is composed of interrelated objects. Ecosystem scientists study the objects and relationships comprising this system. As part of their research activities, scientists build models of such systems. They acquire data to serve as input and validation for their models by sampling the ecosystem. Modeling transformations, implemented in software, are used to derive new models from existing models. Scientists document their research efforts and rely on documentation of research by others.

CHAPTER III

REQUIREMENTS ANALYSIS

Introduction

Requirements analysis will provide a detailed summary of the key entities and processing operations carried out in the domain of ecosystem modeling. This material will be used as the basis for a model of the application domain.

The Real World

Ecosystem researchers study phenomena, both natural and human. Phenomena can be of two types, *continuous* and *discrete*. Continuous phenomena have a numerical value at every point of the earth's surface. Examples are precipitation and elevation. We assume that discrete phenomenon are bounded in three-dimensional space. Examples are roads, rivers, lakes and buildings. Phenomena that have a non-numerical value at every point on the surface fall into a gray area. In many cases, these phenomena are the product of human activity, e.g. land ownership, land use and political territory. In other cases they can only be measured after expert analysis of data, e.g. vegetation community type.

We measure phenomena at four levels, as summarized in (Muehrcke78). *Nominal* is the most primitive and is used strictly for classification. We may subdivide a set of data into areas of equal vegetation type, where each type is represented by a unique numerical value. *Ordinal* measurements indicate a ranking on a continuum, with each group having a value. *Interval* and *ratio* measurements indicate magnitude. In the case of an interval measure, like temperature on the Celsius or Fahrenheit scales, the numbers used don't have an absolute value. Hence, the value zero in an interval measure has no real world significance. Ratio is similar to interval in that the value indicates magnitude on a scale. However, the zero is a meaningful value. Precipitation is a ratio measure. Nominal measurements are sometimes referred to as qualitative, while ordinal, interval and ratio measurements are known as quantitative. We use the term *categorical* in place of nominal, and the term *numerical* in place of interval and ratio. We do not treat the ordinal level of measurement in our work.

Phenomena in the real world are associated with entities or structures. These entities are sometimes identifiable features of the natural and human landscape. In other cases they are the product of analysis and modeling carried out by experts in a particular application domain. Burrough says that "all geographical data can be reduced to three basic topological concepts - the point, the line and the area. Every geographical phenomenon can in principle be represented by a point, line or area plus a label saying what it is." (Burrough86, pp. 13) Unwin (Unwin81) discusses point, line and area entities, and also discusses the surface. A *surface* is a collection of points and a scalar field characterizes the surface. A scalar field is a contiguous set of positions, each of which has an associated scalar value, characterized by a magnitude measure. Thus, magnitude values represent a function of scalar field position. Unwin describes an example of a contour map of a mountain, where altitude is a function of <x,y> position. He says two critical assumptions have already been made. "The first is that of continuity, that is a z value exists (or can be imagined to exist) everywhere on the surface, and that there are no sharp discontinuities.... Second, it is assumed that the field is single-valued, that is only one value of z is present at each location." (Unwin81, pp. 154). Based on these assumptions, a scalar field can be used to approximate a surface corresponding to the surface of the earth. The two conditions above are not always perfectly met, especially in mountainous areas, but this shortcoming is often overlooked for modeling purposes. Using these assumptions allows other continuous phenomena to be treated as surfaces of some type.

We can find structure in the real world by partitioning areas into subunits. We recognize two kinds of partitioning - spatial and value. We partition an area spatially by dividing it into spatially contiguous areas. In other words, we can identify spatially contiguous sets of points as sub-areas, each with a particular geographic extent. This kind of partitioning is often used to isolate a small area for focused analysis or to provide an organizational framework for archiving and management of large spatial models. Value partitioning is used to find a more complex kind of structure. We place in one partition all areas that are of common value for a particular phenomenon. Generally, researchers choose one phenomenon as the basis for structure. They then develop criteria and a methodology for evaluating the chosen phenomenon. The result of evaluation yields a surface broken into value partitions. In somewhat more formal terms, spatial partitioning divides a set of points into spatially contiguous subsets. Value partitioning is carried out by isolating one phenomenon, and partitioning a set of points into possibly disjoint subsets that are equivalence classes with respect to the value of that phenomenon.

Models of Reality

Scientists build models of reality to facilitate their research. We define a model as an abstract representation of reality. Models can be of two kinds - spatial and temporal. Spatial models are static views of the real world; that is, they either describe the state of the ecosystem at a particular instant, or they summarize state values over a particular time interval. Spatial models represent phenomena in n dimensions. We limit ourselves to two-dimensional spatial models. Temporal models contain a logic that simulates system activity and produces an ordered set of spatial models called a time series. A particular time series has a temporal scale and resolution. Temporal scale is the time period represented by the series, whereas temporal resolution, also called time step, indicates the time interval between the spatial models comprising the series.

Geographic information systems and image processing systems utilize two different characterizations of spatial data -- *raster* and *vector*. The raster model assumes a regular grid that divides a surface into cells of uniform size, sometimes called pixels, each of which represents a discrete area of the surface. Each cell is referenced by unique <x,y> coordinates. Raster cells are spatially contiguous, and in most applications are of a standard square shape. Burrough (Burrough86) summarizes the advantages and disadvantages of the raster model. He says the data structure is simple, permitting easy simulation and analysis operations. However, this model requires that large amounts of data be stored, and will lose information about the phenomenon if the resolution of the phenomenon is smaller than the raster cell size. Any model of spatial data must be able to store topological information, that is, a description of the spatial relationships between objects. Topological information is implicit in the raster model at the level of the individual cell, in the sense that neighbors are easily identified by simple functions on <x,y> coordinates. However, for more abstract raster objects, such as groups of cells, topological information must be explicitly managed.

The vector model is based on three kinds of entities. A *node* represents point entities, an *arc* represents a line entity defined by start and end nodes and a *polygon* represents an area defined by a closed collection of arcs. The vector model gives what is sometimes called a continuous representation of reality. This means that the spatial location of nodes, arcs and polygons corresponds to their location in the real world to a certain level of precision. Each object (node, arc, polygon) on a vector layer is given a unique identifier which serves as the primary key into a relational table. Burrough (Burrough86) summarizes the advantages and disadvantages of the vector model as follows. The vector model gives a truer representation of real world entities and for sparse phenomenon provides a much more compact data structure than the raster model. However, the data structures required by vector models are complex, making many important modeling and analysis operations very difficult to implement.

The vector model can store topological information on the objects that comprise a surface. Aronoff (Aronoff89) describes the ARC-NODE model in which topological data is store for all arcs, nodes and polygons on a vector layer. Polygon topology defines a polygon in terms of the arcs by which it is bounded. Node topology defines a node in terms of the arcs in which the node is either a start or end point. Finally, arc topology identifies the start and end nodes for each arc, as well as the polygons to the left and right of the arc. While we recognize the importance of these traditional views of spatial data, we seek a representation that allows us to ignore the distinction between vector and raster.

The spatial representation that we use in this work is that of a model of the geographic database (Burrough86). Although the term geographic database implies that this model relates to data in a spatial modeling/analysis system, we can apply this model to entities and phenomena in the real world. Burrough's model provides a means of bridging the gap between the real world and the software systems used in the application domain. Implicit in Burrough's model is a view of the earth's surface as an infinite set of points, each of which has a set of attributes. Each attribute represents a phenomenon. Attributes representing continuous phenomena are present at every point on the surface. Attributes representing discrete phenomena are present only where those phenomena occur. In other words, an attribute representing streams is present only at points that lie in the stream. We will call this representation of reality the *point model.*

Burrough's model also uses a structure called an *overlay*. An overlay is a scalar field which represents the value of exactly one attribute for a particular area. There is a unique overlay for every attribute for a given area. If we examine a particular overlay, we find a structure that Burrough calls a *region*. A region is a set of <x,y> positions that share a value for a particular attribute. A region is composed of one or more spatially contiguous areas. The smallest region is one <x,y> position. The region is therefore an equivalence class with respect to one attribute.

In the work here we refine Burrough's model in several ways. Firstly, we will describe two kinds of overlays -- *numerical overlays* hold values on the numerical measurement level and *categorical overlays* hold values at the categorical level. Secondly, we note that Burrough does not treat the spatially contiguous areas that comprise a region as meaningful entities. However, these entities are spatially unique objects and should be treated as semantically meaningful. We call these entities subregions and recognize two kinds. Linear subregions are one-dimensional and are composed of at least two points, while areal subregions are two-dimensional and may be as small as one point. We realize that geometrically a point is a subspace of area "zero" in a larger two-dimensional space. In modeling terms, each point is assumed to have a non-zero area corresponding to its spatial resolution. Thus we treat the point as similar to the cell in the traditional raster model. We define a four-tiered overlay structure. The foundational structure is the point, the second level of structure is provided by the subregion, the third tier by the region and the top level by the overlay.

How Models are Used

We discuss here a subset of the key modeling activities carried out by MT-GAP and NTSG projects. In subsequent sections, we will expand on these descriptions and describe other modeling activities.

Gap analysis (Scott93) is intended to provide a more proactive approach to biodiversity protection than has been available in the past. The process relies on spatial models of spectral reflectance, topography, land ownership, land use and other phenomena. These foundational models are analyzed to predict land cover types and wildlife distributions. These higher level models are used to identify geographical areas that are important to the survival and well being of various species and communities. A vegetation model is fundamental to biodiversity assessment, because the vegetation at a site reflects many physical and biological factors that are significant to plant and animal species.

The MT-GAP project uses many traditional image processing and GIS analysis techniques, as well as modeling and analytical methods developed by the project staff. (Ma 93) discuss modeling activities being used on the Montana project to map existing vegetation and land cover. Remotely-sensed reflectance data recorded by Landsat TM scanner is used to identify units of land with similar spectral pattern. Land units smaller than five acres (i.e., the minimum map unit) are removed by absorbing them into adjacent land units. Topography is modeled for the remaining units. A subset of these units are sampled in the field to determine plant cover type and is used to develop a characterization for each cover type, called a signature. The signature is used to label the cover type of each land unit in the model by cover type.

The Numerical Terradynamic Simulation Group studies ecosystem processes at watershed and larger scales by using a suite of temporal modeling tools. We limit our discussion to the Regional Hydro-Ecological Simulation System (RHESSys) which has been developed by NTSG (Nemani92). RHESSys is comprised of submodels called Forest-BGC, Basin, MtClim and TopModel. Forest-BGC is a process model that simulates the cycling of carbon, water and nitrogen through forest ecosystems. The model requires topographic and soil parameters, along with an estimate of vegetation canopy density. The simulation is driven by climate data that is extrapolated from a weather station to the study site. Seven state variables are modeled on a daily time step: evaporation, transpiration, snowpack, soil water, stream discharge, photosynthesis and maintenance respiration. The remaining state variables are updated once a year: growth and decomposition respiration, nitrogen loss, available carbon and nitrogen, and carbon and nitrogen in leaves, stems, roots, soil and litter. A topographic landunit called a hillslope is used as the simulation unit, reflecting the strong relationship between the hillslope and forest cover patterns in mountainous regions. The submodel, Basin, is used to partition the study area into hillslopes, so that each hillslope is partitioned into areas of similar soil-water dynamics. MtClim is used to extrapolate climate data, generally on a daily time interval for the period for which the simulation is to run. TopModel (Bevan79) simulates water flow in moderate to steep topography.

The execution of a RHESSys simulation proceeds as follows. Basin partitions the study area into topographically-defined simulation units. MtClim provides climate data for each simulation unit. Next, Forest-BGC is run on each simulation unit and results are integrated across the hillslope. TopModel provides a robust model of hydrology, providing estimates of stream discharge in terms of timing and quantity, and of soil moisture patterns. These patterns are especially important in determining rates of evapotranspiration and photosynthesis.

Data Acquisition

Remotely Sensed Data

Before information about the real world can be used for modeling and analytical purposes, some method(s) for sampling the real world must be devised. A great deal of the data used in ecosystem modeling is sampled remotely -- i.e., from high above the earth -- and over large areas virtually simultaneously. Two primary vehicles for remote sensing are utilized today -- aircraft and satellites. The use of aerial photography from aircraft mounted cameras has long been a source of data about the earth, used to generate maps and other representations of the earth's surface. The use of satellite imagery is newer, but rapidly growing in importance in a wide range of applications.

Two of the products from aerial photography that are used in the MT-GAP project and NTSG research will be discussed here. First is the *digital elevation model* (DEM) which describes the elevation of points in a regular grid on the earth's surface . A similar product is called a *digital terrain model* (DTM). Burrough (Burrough86) uses the term digital elevation model in a broader context to refer to any digital representation of the elevation of the earth's surface. He uses the term *altitude matrix* for the particular kind of elevation representation that is commonly available as an actual DEM or DTM dataset. The altitude matrix has an elevation value at each intersection point of a regular grid. The U. S. Geological Survey describes three methods used in the collection of elevation data for the agency 7.5 minute DEM product (USGS86).

1) Elevation data is interpolated from digital representations of hypsography (contour lines).

2) A stereo-pair of aerial photographs are processed using a method called terrain profiling, and the resulting data is interpolated to a regular grid.

3) The Gestalt Photo Mapper is used to examine a stereo-pair of aerial photographs and sample the terrain on a regular grid.

The second product of aerial photography discussed here is a *hydrography* overlay. Hydrographic features are bodies of water, such as lakes, springs, marshes, ponds, streams, and wells. A map generated from an aerial photograph is used as source material for the digital representation of hydrographic features. The hydrography overlay is formed by digitizing the map, either manually or automatically.

Satellite based monitoring of the earth has been carried out for several decades. The LANDSAT series of satellites has provided remote sensing capabilities by carrying scanners that record electromagnetic radiation reflected off the earth's surface at various wavelengths (ERDAS91, Lillesand79). The term *scanner* is typically used to describe an entire data acquisition system. A satellite may carry more than one scanner. Each scanner carries a *sensor*, which records radiated energy and converts it to a signal suitable for further use. Each sensor contains a number of detectors. The current generation of LANDSAT satellites carries the Thematic Mapper (TM) scanner system, which records radiation from seven bands of the electromagnetic spectrum. Three bands, the lowest in terms of wavelength recorded, are in the visible spectrum. One band is in the near infrared, which lies just beyond visible red. Another two bands lie in the mid infrared range, and the final band is in the thermal infrared. The width in microns of each band is called the *spectral resolution*. A sensor's *spatial resolution* is the size of the smallest object detectable by the sensor. The spatial resolution of TM data is 30 meters for every band, except band 6, which has a resolution of 120 meters. The data captured by TM scanners is initially processed by Earth Observation Satellite Corp. (EOSAT). The result is a dataset with all bands converted to a 30 meter resolution, called a *satellite scene*. Individual scenes vary in size. The Missoula scene is a 7000 by 7000 grid of cells representing approximately 13 million acres. The *temporal resolution* of TM data (that is, the time interval between separate recordings of a particular area) is sixteen days. A sensor also has a *radiometric resolution* which indicates the number of values the signal transmitted to the earth can have. The TM sensor has a radiometric resolution of 256, i.e., the data values are encoded in 8 bits.

Ground Based Sampling

Ground-based sampling of data has been used in the MT-GAP project and in ecosystem simulation by NTSG to measure climatic phenomena. Although climate is a continuous phenomenon, current technology is limited in remote sensing of this data. Therefore, ground observations are necessary. For climate modeling, researchers at NTSG generally use data collected by the National Weather Service at primary and secondary meteorological stations, and archived at the National Climate Data Center, by the National

Oceanic and Atmospheric Administration (NOAA) is useful for climate modeling. Interpolation and extrapolation of this data are the subject of considerable modeling activity and will be discussed later.

Ground-based sampling also provides data that can be used to complement analyses that are based on remotely-sensed data. This practice, called *ground truthing*, provides an empirical check on the methods used by modelers. Field workers examine plots and record observations appropriate to the purpose of a particular study. In some cases a standard *sampling system* may be used. For example, MT-GAP utilized the Ecodata sampling system developed by the U.S. Forest Service. The Ecodata system provides over twenty field forms for recording particular observations and attributes of a site.

Model Transformations

Cartographic Transformations

Robinson, et al., (Robinson53), define *cartography* in the broad sense as work in which the use of maps is of fundamental importance. The term map must be used with care because researchers use hard copy maps only part of the time. Most of the time they work with digital representations of maps using the data models discussed in the previous section. Robinson, et al., (Robinson53), describe five conceptions of cartography, only one of which -- the geometric focus -- is important in this discussion. The aim of cartography in this view is the creation of cartographic models of reality that are used for measurement and analysis of phenomenon. The emphasis here is on accuracy of measurement. This of course is the primary concern of scientists studying earth resources and the natural environment.

The process of representing the three-dimensional surface of the earth on a two-dimensional hardcopy map or a flat computer monitor requires the use of a *map projection*. The act of projection involves the application of a function to the three-dimensional data in order to render it in two-dimensions. There are many such transformation functions. The spatial location of real-world entities is given in the *geographic coordinate system* using latitude and longitude. At this stage there is no projection system involved. After projection into two-dimensions, a *rectangular coordinate system* is used for referencing spatial locations. Some accuracy is always lost in a projection transformation. The fidelity of geographic data can be analyzed in four ways -- representation of angles, areas, distances and directions. Each projection system preserves fidelity in some of these areas better than others. The analyst must decide which projection system will best suit the needs of the analysis, based on the nature of the data, the characteristics of the study area and the purpose of the analysis. For a given study, one projection system is usually chosen as the standard. All input overlays (raster and vector) are converted to the same projection system. During the conversion from one projection to another, the analyst must provide parameters that will guide the transformation process.

Overlays representing the same geographic area must also be referenced to the same coordinate system. The same geographic location will thus be represented by the same map coordinates in both overlays. *Registration* is the procedure used to accomplish this goal. In absolute registration, different overlays are registered to a common coordinate system. In relative registration, one overlay is used as a base, or master, to which the other overlays, called slaves, are registered. The registration process typically takes three steps. First, a set of features (generally point entities) that can be identified on all the overlays are isolated. Second, the coordinates of these points are recorded. Finally, a function is generated to transform coordinates on each slave into the appropriate coordinates on the master overlay.

When performing projection transformations or registration on overlays with a regular point distribution, it is often necessary to *resample* the surface. A resampling algorithm generates a new value for a particular point by performing a calculation on the values of its neighbors. The product of resampling is an overlay with newly calculated values for all points. Resampling is often necessary after projection or registration because the functions that transform the surface distort the regular grid of points. There are a number of possible resampling algorithms, e.g., nearest neighbor, bilinear interpolation and cubic convolution. Resampling may also be used to change the spatial resolution of the points on an overlay to match the resolution of another overlay.

Partitioning Transformations

Introduction

Ecosystem modelers seek out a basic structure to use as the focus of modeling activities. The structure must be semantically meaningful - that is, it must correspond to a real world entity. Primitive overlay objects (i.e., points) may represent objects of special significance to modelers (e.g., wildlife sightings, spring locations). Usually however, overlay points are arbitrarily chosen as part of a grid of points to represent an area, and do not have this semantic meaning to ecosystem modelers. The analyst must aggregate these points into semantically meaningful units. Two methods of doing this are used by MT-GAP and NTSG : *classification* based on spectral value, which generates a surface divided into spectral land units, and *topographic partitioning*, which subdivides the surface into spatially contiguous areas representing the natural terrain.

Classification and topographic partitioning are important components of modeling because they allow researchers to seek out higher levels of structure than exist in raw data. The problem common to all classification activities is: given a set of unclassified objects, describe a set of appropriate classes, then decide which class each object should be assigned. Classification and topographic partitioning are typically treated separately, in spite of the fact that topographic partitioning can be viewed as a subtype of classification.

That is, the partitioning of a surface into topographic land units involves examining a collection of points and placing each one into a topographic landunit, just as spectral classification requires that each point be placed into a spectral class. However, the literature on classification typically excludes topographic partitioning. Both these methods are discussed further in sections below.

Classification

Burrough (Burrough86) provides two characterizations of classification methods. First he distinguishes *univariate* and *multivariate* classifications. Univariate classifications are based upon a single variable, or attribute. Multivariate classifications are based upon multiple variables. Secondly, Burrough distinguishes four different methods of generating class intervals. *Exogenous* class intervals are related to the data being classified, but are not derived from that data. They are often standard classification schemes in a discipline (e.g., land cover type, soil class). *Arbitrary* class intervals are selected without a clear aim, while *serial* intervals are chosen so that they relate to each other mathematically. Sub categories of the serial interval include class intervals based upon 1) normal percentiles, 2) a proportion of standard deviation centered on the mean, and 3) equal intervals on both arithmetic and non-arithmetic scales. *Idiographic* intervals reflect the nature of the data set being classified; there is a unique set of intervals for each set of data classified. Therefore, comparison between different data sets that are classified independently is difficult, if not impossible.

A third subdivision of classification focuses on the nature of the classes into which objects are placed. The classes can be based either on traditional or fuzzy set theory. Fedrizzi discusses fuzzy set theory noting that "analysis and modeling of real world phenomenon or processes must take into account an inherent uncertainty." (Fedrizzi87, pp. 13) He notes that in some cases, this uncertainty is due to vagueness, i.e., a lack of clearcut boundaries of the set of objects to which a label is applied. As originally described by Zadeh (Zadeh65), all sets can be described by a *characteristic function* whose value indicates whether the object belongs to the set. In traditional, or crisp, set theory the function maps every candidate object to a membership value of one or zero (i.e., the characteristic function value corresponds to either true or false). Fuzzy set theory uses a *membership function* which maps object membership into the real number interval zero to one, also called a *possibility value*. As an example, imagine a set of cover type classes into which we must sort landunits. Using traditional set theor, we generate a definition for each class corresponding to a characteristic function. We apply this function to each landunit, and the function returns a zero or one to indicate whether or not the land unit belongs to the class. We apply the characteristic function for each class to the landunit until either 1) we find a return value of one, or 2) we have exhausted all classes. If we exhaust all possibilities without finding a member, the landunit remains unclassified. Approaching the same classification problem using fuzzy set theory, we start by deriving a fuzzy membership function for each cover type class. As above, we apply the function for each class to each landunit. The interval between zero and one contains an infinite number of real numbers; therefore the set of potential possiblity values returned by a membership function is theoretically infinite. Thus, we usually find that a landunit has non-zero possibility values for more than one class. The process of *de-fuzzification* is used to select among several candidate classes.

Data cannot be classified until a set of classes, called in science a *class scheme* or *class system*, has been defined. The process of characterizing the classes into which objects will be placed is called *training*. An analyst must not only identify the set of classes; he/she must also define each class in terms of the dataset to be classified, so that the classification process will be straightforward. Training can be of two kinds. *Unsupervised* *training* examines the set of unclassified objects to determine the structure inherent in the data. This structure is used as the basis for class definitions. Unsupervised training is also called *cluster analysis* and can utilize one of four standard algorithms - sequential, statistical, isodata and rgb clustering (ERDAS91). Sequential clustering not only trains, but also classifies all points in the image. The other three clustering algorithms require the analyst to subsequently classify the image points. Cluster analysis may be an interactive process and may be repeated a number of times until the analyst is satisfied with the number of classes and their definitions. *Supervised training* requires the analyst to have *a priori* knowledge of the data set or the area represented by the image. One such source of knowledge is ground truthing. A *training sample* is a set of image points which comprise a discernible pattern that may represent a class. A *training site* is the geographic area represented by a training sample. A training sample can be used to generate a *signature*, or set of statistics that characterizes a class. Signatures can also be generated during cluster analysis.

The actual classification process requires the analyst to choose a *decision rule* that will be used to determine which class a given object is assigned to. Four decision rules are commonly used: parallelepiped, minimum distance, mahanobis distance and maximum likelihood (ERDAS91).

MT-GAP Classification Methods

The role of classification in the GAP project is described in (Ma93). Firstly, points in a TM scene are classified into *spectral class*. The product of this classification is a partitioning of the study area into regions, each representing a spectral class. Each is composed of a number of subregions (i.e., an aggregate of points) which serve as modeling units. The data is classified into spectral groups, or classes, that simulate the color composite of TM channels 3, 4 and 5. These bands record the visible red, near-infrared and mid-infrared wavelengths and are commonly used for visualization and feature identification in image processing. The algorithm identifies by similarity of color, groups of similar points in three-dimensional spectral space. Within each color group, brightness sub-groups are identified. These sub-groups are the clusters or classes into which all the points in the image are classified. Training and classification are combined in a two-pass process called "corr_dist" (Ma93). The data is trained in the first pass by randomly sampling the set of points comprising the image. Each sampled point is examined to determine if it is distinct from the previous samples, if any, to form a new cluster. If so, this newly examined point is stored in a *spectral bank* holding those points that characterize the clusters identified so far. In the second pass of the process, each point in the image is examined with respect to the "training points" stored in the *spectral bank.* Each point is labeled using the Euclidean Distance decision rule.

In the second stage of classification, the subregions produced by spectral classification become the objects of classification (i.e., instead of individual points). This is a traditional set-theoretic approach to cover type classification which utilizes the nearest member group algorithm (Ma93). Each subregion is "located" in *n*-space, where *n* equals the number of spectral and biophysical parameters used in the training and classification. The cover type for a subregion of unknown class is the cover type of the nearest labeled subregion. The class intervals used here are exogenous, and make use of a standard land cover type classification scheme. Before classification, the analyst conducts supervised training, using ground-sampled data collected at training sites. For each training site, the cover type is determined. Next, each region (i.e., a set of training sites that have the same cover type) is characterized in terms of spectral and biophysical parameters including: mean values for all seven TM bands, mean elevation, gradient and aspect. Summary climate statistics on both a yearly and growing season basis can be derived, but have not been used to date in cover type classification.

We can extend the cover type classification process beyond that used in this traditional approach by the following fuzzy set-theoretic method. For each cover type, all training sites are analyzed and summary statistics are generated for each parameter. Thus we have a signature for each *<cover type, parameter>* pair. The signature is used to derive a fuzzy set membership function. This m x n set of functions (where m equals the number of cover types, and n equals the number of parameters) is applied to each unclassified subregion. The product is an m x n matrix of possibility values for each unclassified unit. Conceptually, the matrix represents a set of rules, with one rule for each cover type. More precisely, each of m rules is a conjunction of n statements (one for each attribute). If all the statements in a given rule are true (that is, if the possibility value for all attributes is greater than zero), then the rule has a value greater than zero. We therefore derive the "truth" value of a rule by taking the minimum of all its component possibility values. At the end of evaluation, the cover type for a modeling unit, is the type whose rule has the greatest "truth" value.

Topographic Partitioning

The second approach to generation of landunits suitable for modeling is topographic partitioning. Band, et al., (Band90,92) discuss the theoretical background of this approach and a widely used algorithm they have developed to perform such partitioning. The fundamental unit for this work is the drainage basin, which is viewed as a fundamental topographic concept in geomorphology, hydrology and landscape ecolo gy. Using topography as the focus for ecological modeling offers three advantages: 1) the drainage basin has well-defined internal and external boundaries, 2) topography is more stable than either vegetation or soils, and 3) a set of unambiguous, mutually exclusive and spatially exhaustive objects can be defined from a model of topography.

An object model of the watershed is used in which the watershed is viewed as a partition of hillslopes and stream networks. "The watershed representation follows a formal geomorphic model that defines the consistent ordering and hierarchy of the topographic objects, enabling higher order processing of drainage basin structure and attribute information (Band90, p. 787). The drainage basin is seen as a collection of hillslope facets bounded by the stream and divide links.

A digital elevation model is used as the basic input to the partitioning process. Topographic information, such as gradient and aspect, is derived from the DEM. Initially, the area inside the watershed is viewed as a tree, rooted at the outlet point for the watershed. Each point in the watershed is a node on the tree. Each point has an arbitrary number of descendants. The number of descendants of point A indicates how many points are upstream of point A. The points which have no descendants (i.e., the leaves on the tree), indicate the watershed's ridgelines. Ridgelines are both external, marking the boundary of the watershed, and internal, marking the divides between catchments. The watershed is partitioned into hillslopes by pruning the tree. Thus, all points with fewer than a user-specified number of descendants are eliminated. The remaining points represent a model of the drainage's stream network. This network is comprised of the remaining drainage lines and junction points. In graph-theoretic terms, these structures are called edges or arcs, and vertices or nodes, respectively. The hillslope is defined as the aggregate of all points which exist on one side of a drainage line up to the ridgeline. If the watershed is to be partitioned at a coarser level, the tree is pruned at higher levels.

This topographic model is characterized as a three-tiered structure (Band90). The first level contains data overlays with point based data (i.e., elevation, gradient, aspect, drainage area). Above this level is the extracted stream network, and above this is the basin structure.

Re-partitioning Transformations

Partitioning operations (in particular, overlay classification) often produce an overlay with a very large number of small subregions. A high concentration of small units is commonly called "salt and pepper", or simply "noise". Some of these units may be only one point in size. This amount of detail may make any further modeling operations very compute-intensive. It also makes visualization and examination by eye much more difficult. Additionally, the units may also be too small to have meaning in an application domain. For example, Gap analysis is meant to serve as a coarse filter to identify areas above a certain size (typically 100 acres) that are potentially important habitats for native terrestrial vertebrates (Scott93). Thus, modeling units with size below this level should be aggregated somehow.

There are two common operations that perform this function. The first is commonly available in most GIS packages, and involves passing a window, called a *filter*, across the overlay. Aronoff (Aronoff89) classifies this operation as a *neighborhood operation*. The neighborhood around each point is examined and a function applied. The return value of the function becomes the value of the point. The name of the filter describes in general terms the nature of the function. For example, the *majority filter* has been used on MT-GAP to eliminate "salt and pepper" on classified imagery (Ma93). The return value from this filter is the class identifier that occurs most often in the neighborhood. The resulting overlay will have fewer subregions.

A second re-partitioning method merges any subregions below a user specified size with an appropriate neighboring unit. A modeling unit that is to be merged will be swallowed by one of its neighbors, which is selected according to a user specified function for selecting one of the set of possible neighbors.

Deriving Missing Values

Burrough (Burrough86, pp. 146-147) defines *interpolation* as the "procedure of estimating the value of properties at unsampled sites within the area covered by existing point observations" and *extrapolation* as the procedure of "estimating the value of a property at sites outside the area covered by existing observations." We interpolate by finding a model of variation and applying it to the surface. These models can be of two types. In the discrete model, we interpolate by drawing boundaries on the surface. We can use external landscape features to delineate landscape units, utilize edge-seeking algorithms or employ Thiessen polygons. All these methods assume that the important variation occurs at the boundaries defining landscape units. The second kind of interpolation model is continuous and utilizes the gradual change of values. These methods use a model that can be described by a smooth, mathematically defined surface. They are of two types, global and local. Global models are constructed from observation at all points and don't accommodate local features. Examples are trend surface analysis and Fourier series. Local models are constructed from neighborhood observations. Examples are splines and moving averages, also known as inverse distance weighting.

An important use of interpolation and extrapolation on MT-GAP and NTSG projects is as a method for modeling climate. Climate is a continuous phenomenon that is sampled at irregularly distributed locations. Researchers must often develop a model of climate for a specific location that has no directly-measured climate data. Climate data has been utilized by MT-GAP to aid in cover type classification, and is used in RHESSys to drive ecosystem simulation. Ecosystem modeling requires that this sparsely sampled data be used to infer climate conditions at many unsampled points. The primary modeling logic has been developed at the NTSG lab, in a software package called Mountain Microclimate Simulation Model, or MtClim (Hungerford89). MtClim extrapolates climate data from a site where climate data has been measured, called a *base met station*, to a *study site* where climate values are unknown. The extrapolation uses topographic data for the study site (i.e., elevation, gradient and aspect) and a model of sun-earth geometry to compute incident short-wave radiation. The base station data describes maximum and minimum temperatures and precipitation on a daily time step for a time period selected by the modeler. The input data is drawn from the Climatological Data Summary from NOAA containing data for National Weather Service (NWS) weather stations from the late 19th century to the present. MtClim logic produces data describing maximum and minimum temperatures, dew point, shortwave radiation and precipitation for a study site.

Recent enhancements of MtClim allow NWS data from a number of base stations within a radius of the study site to be used to improve predictions of the conditions at the study site. This new logic interpolates from surrounding base stations and produces a *virtual met station,* which occupies the same two-dimensional location as the study site. The interpolation uses a weighted average method, by determining the radius of the window and the maximum number of NWS stations to use for interpolation of any one virtual station. For each virtual station, the *n* (where *n* is the maximum number of stations to use in the weighting) nearest NWS stations within the window are selected and inverse distance weighting is used to generate a weight to apply to each chosen NWS station. If no stations are found within the window radius, the nearest station at any distance is used. For each attribute to be interpolated, and for each time step, the appropriate data value is weighted and a sum performed to derive the data value for the virtual station. MTCLIM is subsequently used to extrapolate from the virtual station to the study site.

<u>Deriving Missing Variables</u>

Overlays containing values at the numerical measurement level can be used as operands in algebraic operations. An operation can have as operands two overlays or one overlay and a scalar value. The result of the algebraic expression is of course another overlay. We call this kind of operation *overlay algebra*. We discuss two examples of overlay algebra in the two following sections.

Soils data is required as input for various ecosystem simulations, including RHESSys. Commonly used soil attributes include *soil water capacity* and *soil transmis*sivity. Soil water capacity measures the quantity of water the soil can retain per unit of land. Soil transmissivity is *soil depth* times *hydraulic conductivity*, i.e., the capacity of water to migrate through soil. Hydraulic conductivity is typically estimated from soil texture.

Many ecosystem simulations use an abstraction of actual ground cover to estimate the density of the vegetation canopy. Remotely sensed spectral reflectance is the data source typically used for this estimation. Before using the data on these overlays to estimate vegetation parameters, the analyst must compensate for two limitations of the datasets. The level of reflectance detected by the sensor is affected by dust, gases and aerosols in the atmosphere between the ground and the sensor. A widely-used technique for dealing with this phenomenon is known as the clear-lake strategy (White92). Certain objects, such as deep, clear lakes, absorb all incoming radiation. The point on each overlay with the lowest reflectance value is assumed to be a clear lake that reflects no radiation. Therefore, any reflectance value at that point is assumed to be a product of atmospheric effects. Overlay algebra is used to subtract the value at that point from all points on the overlay. The resulting dataset is taken to be atmosphere corrected. Overlay algebra is also used to compensate for the effects of sun angle. Satellite scenes are often recorded at a time when the sun angle is low enough to cast shadows across the landscape. These shadows interfere with proper imagery interpretation.

After this corrective work, overlay algebra is performed on three overlays, each representing a unique spectral band. The product of this operation is the estimate of land cover, called the *normalized difference vegetation index* (NDVI). A regression model is used to transform NDVI values into values for another abstraction called *leaf area index* (LAI) (Nemani92, White92). LAI can be measured on the ground, by dividing the total leaf surface area in a tree canopy by the ground surface area under the canopy. Ground measurements can be compared with the estimates from satellite imagery. Thus LAI serves a critical role for ecosystem modelers who rely on satellite imagery. LAI is measured in two forms: *projected LAI* estimates the leaf area on one side of a leaf and is used primarily in broadleaf forests and croplands, while *all-sided LAI* is used in needleleaf forests where the leaves are not flat. However, several transformations must be applied to the raw imagery in order to generate NDVI. First the data must be adjusted for the spectral reflectance produced by the atmosphere, called atmospheric radiance. Secondly the data must be corrected for the effects of sun angle on the image.

EcoModel Assignment

Often during modeling operations, values must be derived for attributes that describe the ecosystem units in an ecosystem model. An overlay describing a particular attribute on a point by point basis is processed to obtain representative values (e.g., the mean) for each ecosystem unit. A value is extracted from each point that falls within the ecosystem unit of interest, and a mean value is calculated. For example, preparation for execution of the dynamic simulation Forest BGC requires that representative values be computed for five attributes: elevation, gradient, aspect, available soil water and leaf area index. A topographic model serves as a template; that is, the model is "laid over" the five overlays containing values for the above attributes. For each hillslope in the topographic model, a mean value is computed for each of the five parameters. This procedure is implemented in a process called "TopCart".

A similar procedure is used to generate an ecosystem model for MT-GAP by performing spectral classification of the numerical overlays representing the area of interest. In order to classify each landunit by cover type and to facilitate further modeling activity, sets of representative values are derived for topographic, spectral and climate attributes. The topographic attributes include elevation, gradient and aspect. Average values per land unit are also generated for each of the seven bands of TM data, as well as a set of summary statistics developed from climate simulation.

Image Enhancement

Images (or overlays) often need to be modified to facilitate data interpretation. This process is called *image enhancement* and is often used for feature extraction. The techniques used depend on the nature of the data and the objective of analysis. We discuss the two categories of image enhancement that are used on single-band imagery (in our model, a single overlay). Our experience is that image enhancement operations are performed on overlays containing elevation or reflectance data. However, in theory any numerical overlay could be the subject of image enhancement operations.

Spectral enhancement deals with values of individual points in the overlay. Spectral enhancement is usually used to make certain features stand out by increasing the contrast in the image. A *linear contrast stretch* takes an overlay and modifies the values of individual points to take full advantage of the capacity of the display device. In most raw overlays, the data values fall within a narrow range. By "stretching" this range to fit that of the display device, contrast is enhanced. The operation can be described as a linear function that maps raw data values to enhanced data values. The operation uses the mean, standard deviation, and other overlay statistics. The range of the raw data is calculated by taking a distance from the mean measured in units of standard deviation. The minimum and maximum values are not used, because these measures are not usually representative of the data. In *non-linear contrast stretching* the function that describes the enhancement is non-linear. Again, it maps raw data values to enhanced data values. *Histogram equalization* is one kind of non-linear contrast stretch. This operation redistributes point data values so that there are approximately the same number of pixels with each value within a range.

Spatial enhancement determines the value of a given point by examining the values of its neighbors. A critical concept here is that of *spatial frequency*, which describes the difference between the lowest and highest data values of a neighboring set of points. An overlay in which all points have the same value has zero spatial frequency. An overlay in which alternating points have values at the highest and lowest ends of the range of data values has high spatial frequency. *Convolution filtering* averages the values of small sets of points across an overlay, and is used to change the spatial frequency of the overlay. A *convolution kernel* is a matrix (generally, a 3 x 3 matrix) of numbers that serve as the coefficients of a function that computes a weighted average. The kernel is placed on the overlay, centered on the point for which we wish to compute an enhanced data value. The return value of the function becomes the enhanced value. An entire overlay can easily be enhanced by passing the kernel across the surface, centering it on each point in turn. Convolution kernels which increase spatial frequency are called *high-pass kernels* and those which decrease spatial frequency are called *low-pass kernels.* A special kind of kernel is *zero-sum,* in which the existing spatial frequency is exaggerated. Zero-sum kernels are often called edge detectors because the resulting image often consists only of edges and zeroes. High-pass kernels serve as edge enhancers in that they highlight the edges between homogeneous groups of points. Unlike edge detectors they don't necessarily elimate other features.

Summary of Requirements Analysis

Our detailed analysis of the key abstractions and modeling operations used in ecosystem modeling provides the foundation for further analysis. Specifically, the entities and processes identified in requirements analysis will serve as the key components of a domain analysis model that is presented in the following chapter and in the appendices.

CHAPTER IV

DOMAIN ANALYSIS

Introduction

We present here the results of our domain analysis, consisting of three sets of documents: class relationship diagrams, class specifications and object diagrams depicting key scenarios. These documents are contained in full in Appendices I to III. In this chapter we discuss key aspects of our analysis, using elements from all three sets of documents.

Class Relationships and Interfaces

We begin with a discussion of class interfaces and relationships. The purpose here is to characterize the key abstractions in the application domain, and describe the relationships between these abstractions. The notation we use is discussed in (Booch89). Each class name is in boldface and class attribute and operation names are in italics.

Our complete class hierarchy contains 169 classes, making it difficult to show in a single diagram. To simplify this discussion, we subdivide classes into twelve sets, each occupying a section of the hierarchy and focusing on a related set of entities. These sets are root, spatial, ecosystem, ecosystem descriptors, operations, classification, data acquisition, descriptors, windows, software, documentation, and humans.

We discuss in this chapter classes and relationships from the groups: spatial entities, ecosystem entities and modeling operations. We have chosen these groups to discuss at length because they lie at the heart of our domain analysis model. Ecosystem modeling is fundamentally an activity in which operations are performed upon spatial representations of ecosystems. Thus, if we confine our discussion here to a subset of our hierarchy, it is sensible to choose those classes and relationships that represent the most significant entities we have discovered.

<u>Spatial Entities</u>

The most significant set of abstractions we have found are those that describe phenomena in two-dimensional space. These entities are geometric objects that provide the foundation upon which ecosystem models are created.

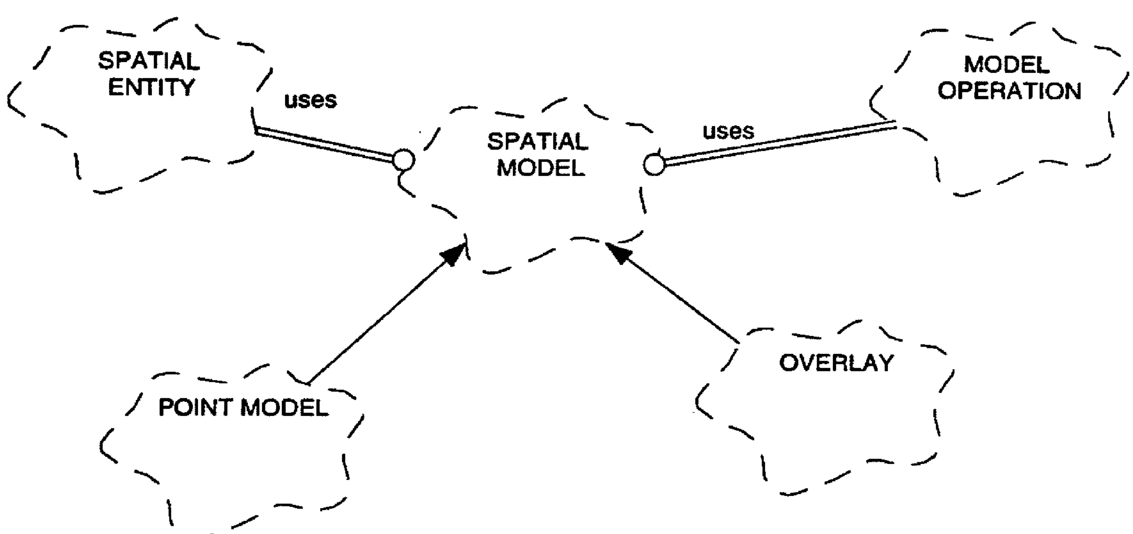


Figure 1

Figure 1 portrays a key class, **spatial model**, using as a class symbol an amorphous shape with dashed borders. **Spatial model** is a subclass of **spatial entity**, and the parent class for two subclasses - **point model** and **overlay**. The parent-subclass relationship is represented by a solid arrow from the child to the parent class.

Spatial model is also involved in a "using" relationship with class **model operation**, which is indicated by a double line from "using" to "used" class with a hollow circle adjacent to the "using" end. This notation indicates that the "used" class is referenced in the interface of the "using" class - either as a formal parameter to an operation or as a data field. In the case where two classes use each other, we have hollow circles on both ends of the line.

class

name	Spatial Model
parent	Spatial Entity
attributes	
spatial scale	String
geographic coordinate system	String
projection	Projection System
consuming operation	Spatial Model Consumer
producing operation	Model Operation
operations	
Get spatial scale (void)	**return** String
Set spatial scale (String)	**return** void
Return projection (void)	**return** Projection System

end class

Figure 2

The interface of class **spatial model** is shown in Figure 2. The class definition contains class name, parent class, attributes and operations. **Spatial model** has five attributes. Attributes are indicated by attribute name, followed by the type of the attribute. In cases where the attribute is an object of a class (i.e., rather than a simple data type), the type name will be the name of the class. In this case, the attributes of **spatial model** refer to two classes that describe processing operations discussed in requirements analysis, and these references provide the history of a particular object by providing a trace of the sequence of processing steps used to create it.

We have followed the advice of White (White93) in providing operations for our classes. We have modeled standard store and retrieve methods for all attributes that are simple data values (e.g. types Number, String) as operations. Names in parentheses are formal paremeters for an operation and indicate the type of arguments required by the operation. The word "return" in the specification is followed by the type of the value returned by the operation. The word void means that we require as an argument, or return from an operation, no value.

class	
name	Spatial Entity
parent	Entity
attributes	
number points	Number
area	Number
perimeter	Number
operations	
...	
end class	

Figure 3

The definition of **spatial entity**, the parent class of **spatial model**, is shown in Figure 3. **Spatial entity** has attributes that describe the number of points contained in the entity, its area and perimeter. Based on the parent/subclass relation, these attributes and operations defined for **spatial entity** are inherited by all subclasses of **spatial entity**.

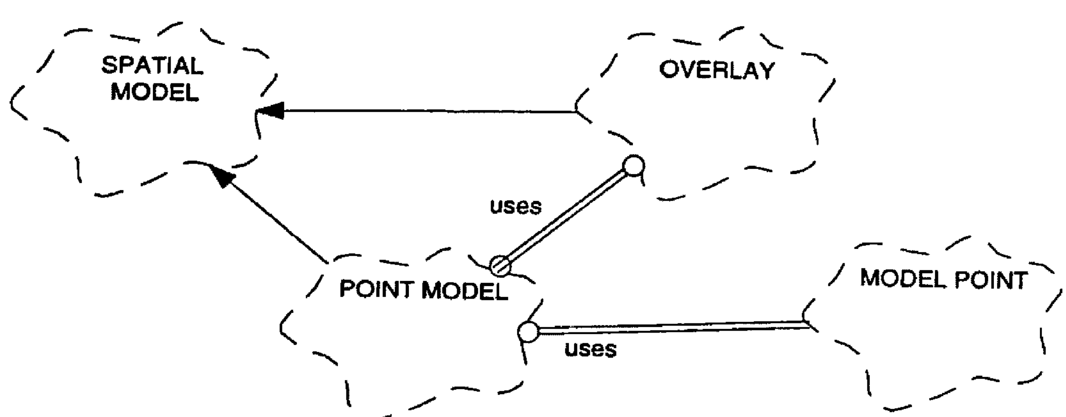


Figure 4

Our concept of **spatial model** is refined in Figure 4. The subclasses **overlay** and **point model** of **spatial model** are shown here, along with a "using" relationship between **point model** and **model point**.

class	
name	Point Model
parent	Spatial Model
attributes	
model point	SET < Model Point >
number overlays	Number
overlay	SET < Overlay >
operations	
Iterate over points (void)	**return** Model Point
Sample points (void)	**return** Model Point
end class	

Figure 5

Figure 5 shows the specification for **point model**. The bracket notation used in the type designation for attribute *model point* indicates that the value will be a set of objects of the class or type named inside the brackets. This notation is used in cases where more than one object of a "used" class can exist in a class interface.

class	
name	Overlay
parent	Point Model
attributes	
number of regions	Number
number of subregions	Number
number of points	Number
data name	Model Attribute
containing model	Point Model
consuming operation	Overlay Consumer
operations	
...	
end class	

Figure 6

Figure 6 contains the specification for **overlay.** The first three attributes indicate the number of overlay substructures that are contained in a particular overlay. The attribute, *containing model*, allows access to the **point model** that contains the **overlay**, and therefore to the other **overlay** objects in the model. *Data name* labels the data in the overlay.

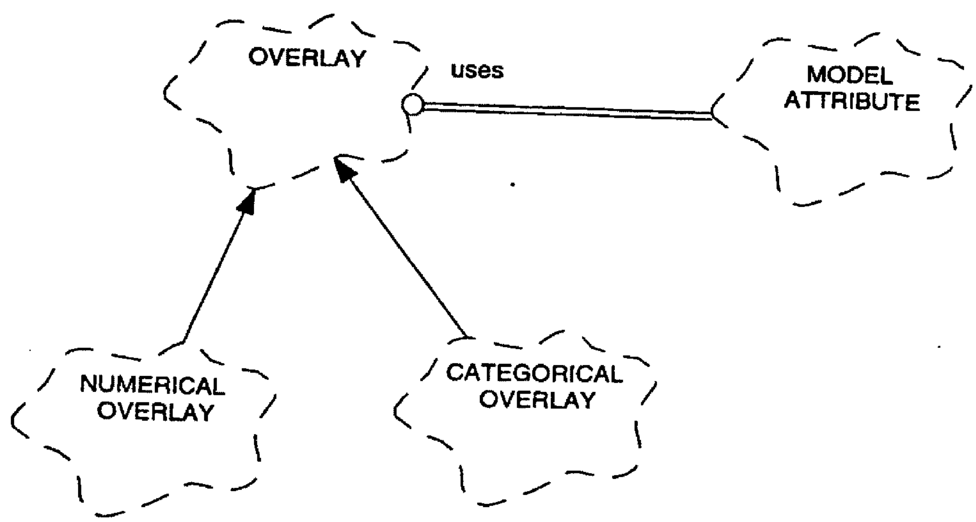


Figure 7

Figure 7 shows two subclasses of **overlay, numerical overlay** and **categorical overlay**, along with a using relationships between **model attribute** and **overlay. Model attribute** labels the **overlay** with the name of the data attribute it represents. Objects of **numerical overlay** hold data at the numerical measurement level, while **categorical overlay** objects hold values that are class identifiers. We distinguish between these two subclasses of **overlay** because of differences in the set of operations that are meaningful in the two cases. That is, **numerical overlays** can be used in overlay algebra, while such operations performed on categorical data are meaningless. **Categorical overlay** objects are also involved in relationships, discussed later, in which **numerical overlay** instances cannot take part.

It would be possible to create a subclass of **numerical overlay** or **categorical overlay** for each kind of data that these structures can hold, i.e., subclasses for elevation, gradient and aspect. However, we believe that the differences between **overlays** holding these three kinds of data do not require the creation of a separate class for each. The behaviors which are semantically meaningful are determined by the measurement level of the data. We view these three **overlays** as objects of class numerical overlay. Thus, we believe that subclasses **numerical overlay** and **categorical overlay** are adequate.

class	
name	Categorical Overlay
parent	Overlay
attributes	
regions	SET < Categorical Region >
subregions	SET < Categorical SubRegion >
points	SET < Categorical Point >
consuming operation	Categorical Consumer
eco model	Ecosystem Model
operations	
Iterate over points (void)	**return** Categorical Point
Iterate over subregions (void)	**return** Categorical Subregion
Iterate over regions (void)	**return** Categorical Region
Sample points (void)	**return** Categorical Point
end class	

Figure 8

The interface of **categorical overlay** is depicted in Figure 8. The attribute *eco model* is a reference to a higher level model which is built upon the **categorical overlay**. **Categorical overlays** are the product of a partitioning operation, and are often used as the basis upon which higher level modeling structure is built. We have called these higher level models, **ecosystem models**, and will discuss them later in this chapter. The operation *iterate* is provided for all three sets of overlay components represented by the attributes *regions*, *subregions* and *points*. The interface for **numerical overlay** is not shown here. It differs from **categorical overlay** in two ways. First, references to **categorical point, categorical subregion** and **categorical region** are replaced by **numerical point, numerical subregion** and **numerical region**. Second, a set of algebraic operations (e.g., add, subtract, multiply and divide) are permitted on **numerical overlay** and its components.

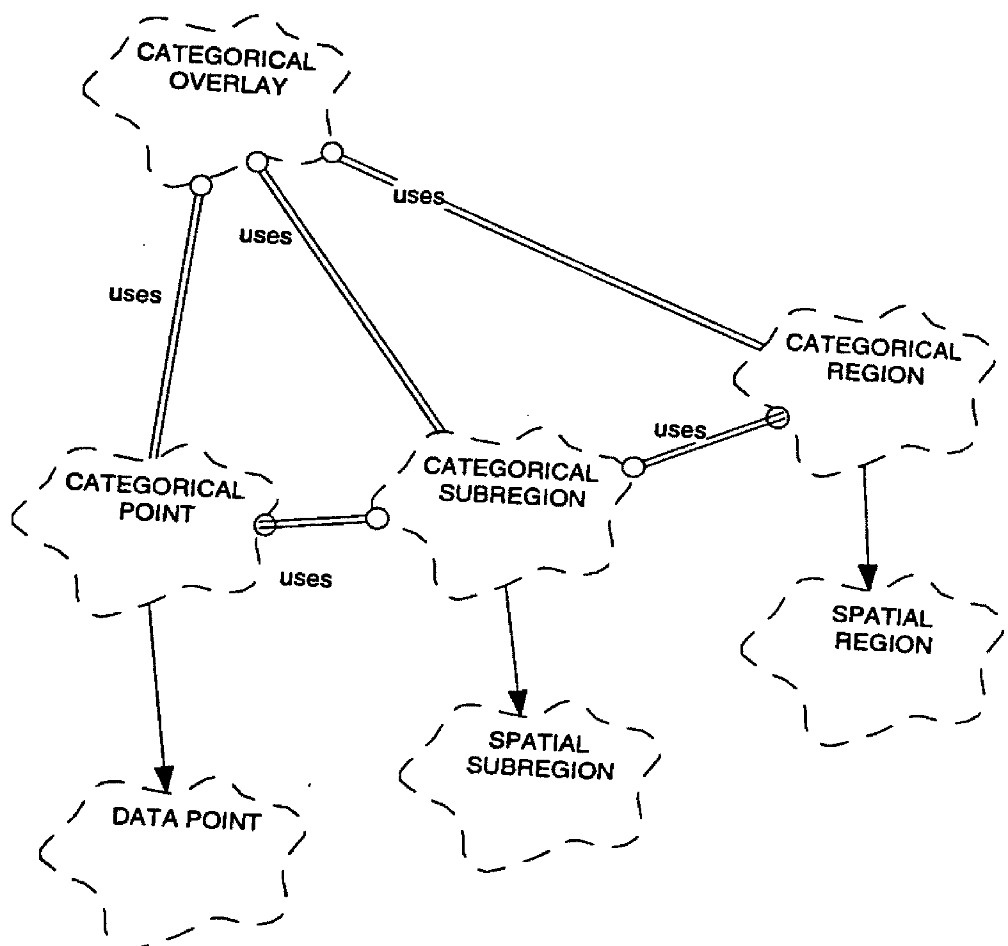


Figure 9

class	
name	Categorical Region
parent	Spatial Region
attributes	
subregions	SET < Categorical Subregion >
operations	
Iterate over subregions (void)	**return** Categorical Subregion
end class	

Figure 10

class	
name	Categorical Subregion
parent	Spatial Subregion
attributes	
points	Categorical Point
containing region	Categorical Region
neighbors	SET < Categorical Subregion >
operations	
Iterate over points	return Categorical Point
end class	

Figure 11

Figure 9 depicts the using and parent/subclass relationships of the **overlay** abstraction, as proposed in our requirements analysis. Class **categorical region,** shown in Figure 10, is a subclass of **spatial region**. **Categorical subregion,** shown in Figure 11, is a subclass of **spatial subregion**. Objects of **categorical subregion** are the spatially contiguous areas that comprise the "polygon mesh" on the overlay surface. An attribute, *containing region*, refers to the **categorical region** in which the **categorical subregion** is contained. This reference to the spatially enclosing objects can be used to move from subregion to region, and query attributes of the region, or to visit other subregions that are part of the same region.

Point objects, which have no area and perimeter, are not subclasses of **spatial**

entity. Point objects form a separate class hierarchy descended from class **named entity**. We do not show the interfaces for the ancestors of **categorical point** here because the semantics of this set of objects is much simpler than the other classes we have discussed. Instead, we summarize the attributes inherited from ancestors. **Categorical point** inherits two objects of class **coordinate** - a *geographic coordinate* and a *spatial coordinate* - and a numerical value indicating *spatial resolution*. The specification for **categorical point** is shown in Figure 12.

```
class
        name                          Categorical Point
        parent                        Data Point
        attributes
         data                         Number
         containing subregion         Categorical Subregion
        operations
         ...
end class
```

Figure 12

Data contained by an **overlay** is actually held by the point objects comprising the **overlay**. The reference to the enclosing **categorical subregion** serves the same function as the pointer to region does in the interface of **categorical subregion**.

Ecosystem Entities

Ecosystem entities are used to build the higher level structures often used in ecosystem modeling. Whereas the **spatial entities** described above are essentially geometric objects, the **ecosystem entities** have a richer semantics reflecting the information scientists obtain about the real world.

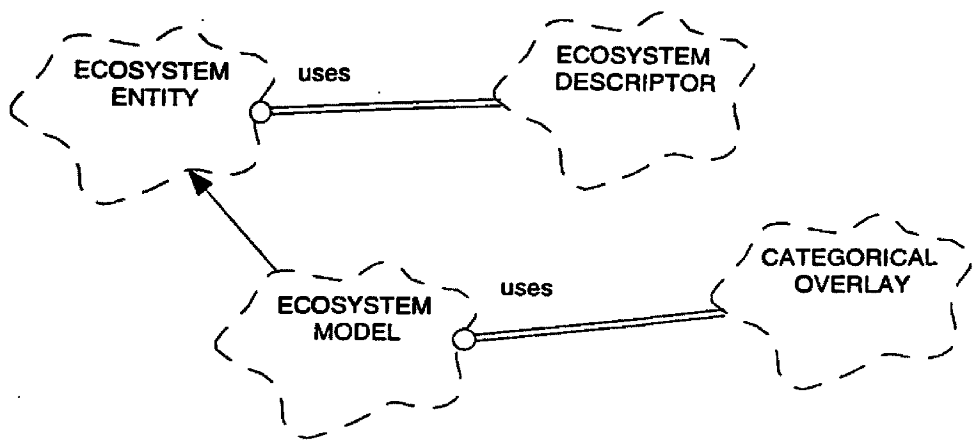


Figure 13

class	
name	Ecosystem Entity
parent	Documented Entity
attributes	
descriptor	Ecosystem Descriptor
operations	
...	
end class	

Figure 14

class	
name	Ecosystem Model
parent	Ecosystem Entity
attributes	
foundation	Categorical Overlay
consuming operation	Ecosystem Model Consumer
operations	
Return foundation	**return** Categorical Overlay
...	
end class	

Figure 15

Figure 13 depicts **ecosystem model**, its parent **ecosystem entity** and **ecosystem descriptor**, along with a using relationship from **ecosystem model** to **categorical overlay**. Figure 14 contains the specification for **ecosystem entity**. An object of class **ecosystem descriptor** can be attached to any **ecosystem entity** in order to provide information on topography, vegetation, climate and other aspects of the entity. The interface of **ecosystem model** is shown in Figure 15. We do not model attributes describing geometric properties for this class because this information is already available on the corresponding **categorical overlay**. The operation *Return foundation*, in the interface for ecosystem model, allows access to the **categorical overlay** upon which the **ecosystem model** is based.

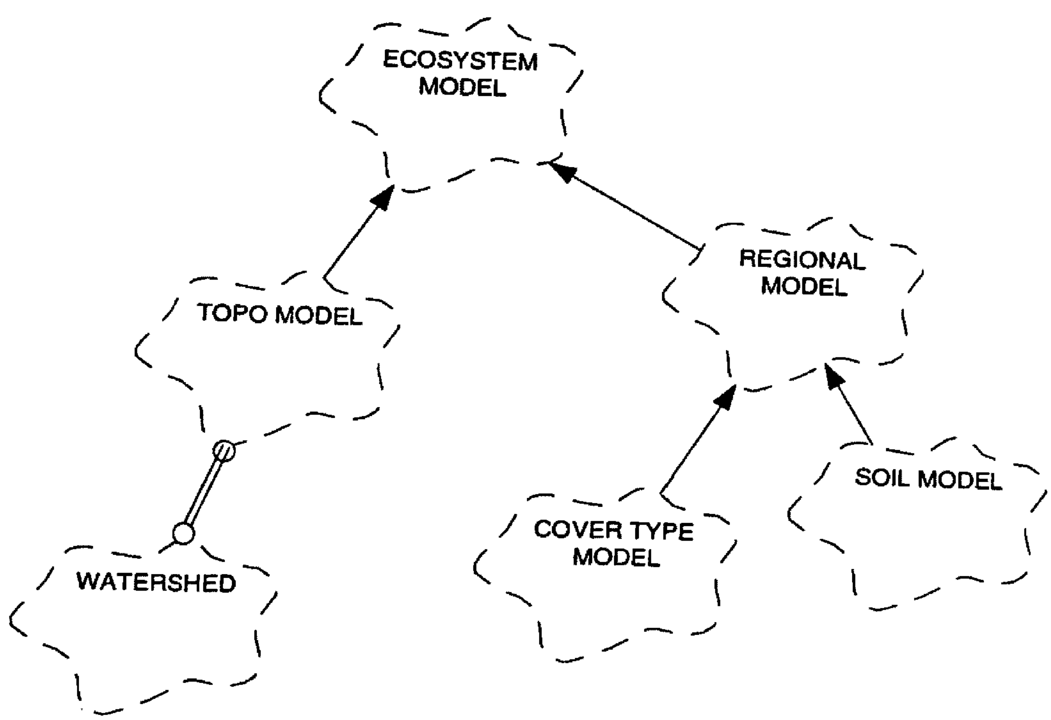


Figure 16

class	
name	Topographic Model
parent	Ecosystem Model
attributes	
number watersheds	Number
watersheds	SET < Watershed >
operations	
...	
end class	

Figure 17

class	
name	Regional Model
parent	Ecosystem Model
attributes	
number regions	Number
operations	
...	
end class	

Figure 18

class	
name	Ecosystem Region
parent	Ecosystem Entity
attributes	
number points	Number
area	Number
perimeter	Number
number subregions	Number
operations	
...	
end class	

Figure 19

class	
name	Cover Type Model
parent	Regional Model
attributes	
regions	SET < Cover Type Region >
subregions	SET < Cover Type Sub region >
consuming operation	Cover Type Model Consumer
class scheme	Cover Class Scheme
operations	
...	
end class	

Figure 20

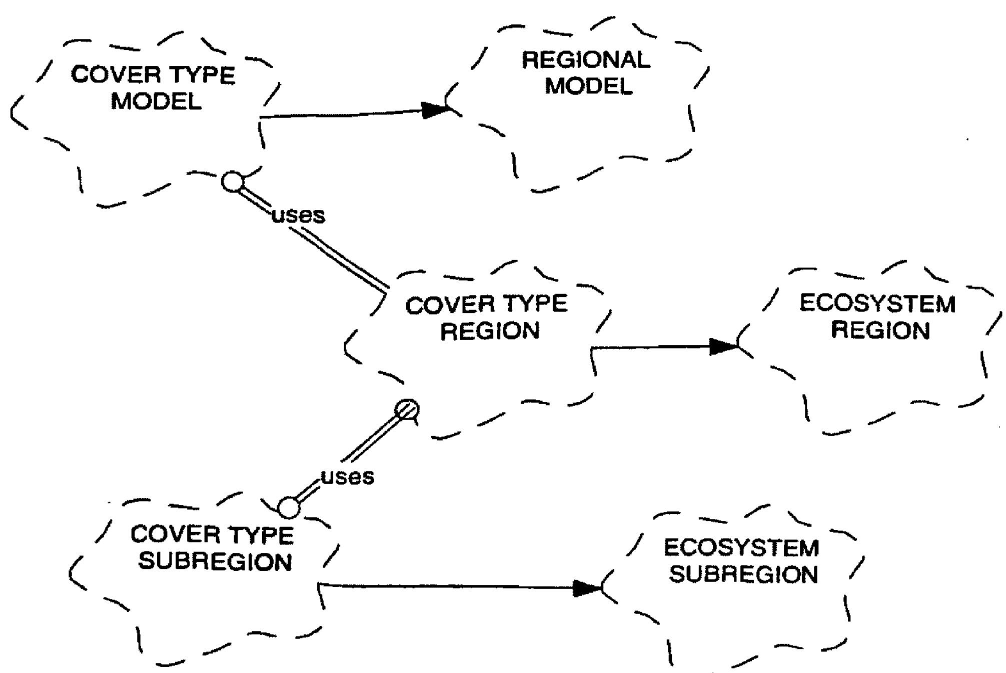


Figure 21

Figure 16 portrays the class hierarchy descended from **ecosystem model**. A **topographic model** doesn't have region/subregion structure; instead it may consist of one or more **watersheds** as illustrated in the specification in Figure 17. Figure 18 presents the interface of **regional model**, which does have the region/subregion structure. The interface of **ecosystem region** is shown in Figure 19. Generally, we do not provide attributes to describe the geometric properties of **ecosystem entity** and its subclasses because the geometry of these entities can be obtained from the components of the **categorical overlay** on which the model is built. However, the number, size and shape of **regions** on an **ecosystem model** will almost never be the same as on the corresponding **categorical overlay.** For example, a partitioning operation (e.g., spectral classification) will examine all points on a set of **numerical overlays** and create a **categorical overlay** where each point has a class identifier. Our **categorical overlay** can be described as a "polygon mesh" of **categorical subregions**, each occupying a spatially contiguous area. A second partitioning operation (e.g., cover type classification) will examine each **categorical subregion** and classifty it into a **cover type class**. It is extremely unlikely that all categorical subregions that fall in the same spectral class, will also fall in the same cover type class. Therefore, we provide attributes *number points*, *area*, *perimeter* and *number subregions* in the interface of **ecosystem region** to reflect our understanding of the difference in region "structure" between **categorical overlays** and **ecosystem models**. Figure 20 shows the interface of a subclass of **regional model, cover type model**, which characterizes land in terms of vegetative cover type. The attribute *class scheme* is a reference to an object of class **cover class scheme,** which is used to classify the model into **cover type regions.** Figure 21 illustrates the parent/subclass and using relationships of **cover type model**.

class	
name	Cover Type Region
parent	Ecosystem Region
attributes	
subregions	SET < Cover Type Subregion >
describing class	Cover Type Class
operations	
...	
end class	

Figure 22

The interface for **cover type region** is presented in Figure 22. The attribute *describing class* refers to the specific **cover type class** that describes this region. A **cover type class** referenced by an object of **cover type region** must be contained in the **cover class scheme** to which the **cover type model** refers. The attribute *subregions* is the set of **cover type subregions** which comprise **cover type region**.

class	
name	Ecosystem Landunit
parent	Ecossytem Entity
attributes	
foundation	Categorical Subregion
virtual climate station	Virtual Met Station
operations	
...	
end class	

Figure 23

class	
name	Cover Type Subregion
parent	Ecosystem Landunit
attributes	
containing region	Cover Type Region
operations	
...	
end class	

Figure 24

The classes describing the spatially contiguous portions of an ecosystem model are subclasses of **ecosystem landunit**, whose specification is shown in Figure 23. The attribute *foundation* indicates a mapping of the **cover type subregion** from corresponding unit on the **categorical overlay**. *Virtual climate station* is the product of interpolation of climate data in MtClim, discussed in requirements analysis. As illustrated in Figure 24, class **cover type subregion** adds to the properties inheritied from **ecosystem landunit**, the attribute *containing region*.

Operations

The modeling and analysis operations discussed in requirements analysis define the characteristics of a set of objects. In an object-oriented methodology, operations are seen as properties of individual classes, and not usually as objects in their own right. We do model some operations in class interfaces, but have found that these operations tend to be limited in scope. Significant operations have a larger and more complex logic that prevents them from being modeled on individual classes. Booch (Booch89) uses the term mechanism to describe a set of object behaviors that together achieve a more complex task than any one object could in isolation. We provide diagrams depicting several of these mechanisms later in this chapter. First we present class diagrams for several modeling operations.

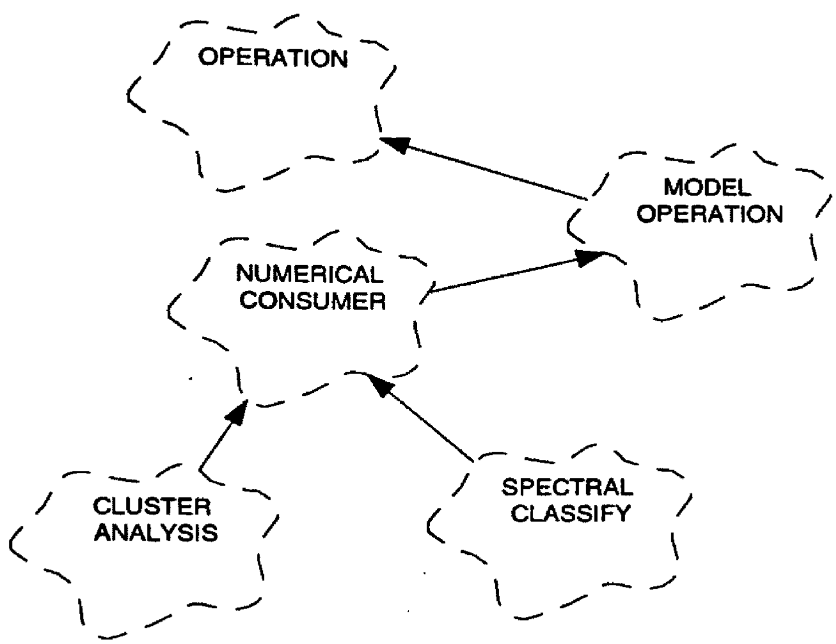


Figure 25

class	
name	Operation
parent	Documented Entity
attributes	
date	Date
time	Time
operations	
Perform (void)	Return void
end class	

Figure 26

class	
name	Model Operation
parent	Operation
attributes	
performer	Analyst
command used	Command
operations	
...	
end class	

Figure 27

```
class
        name                          Cluster Analysis
        parent                        Numerical Consumer
        attributes
          output bank                 Spectral Bank
          output scheme               Spectral Scheme
        operations
          ...
end class
```

Figure 28

```
class
        name                          Spectral Classification
        parent                        Numerical Consumer
        attributes
          input scheme                Spectral Scheme
          input bank                  Spectral Bank
          output overlay              Categorical Overlay
        operations
          ...
end class
```

Figure 29

Figure 25 shows a portion of the hierarchy containing classes representing operations. Classes **operation** and **model operation** have simple interfaces as shown in Figures 26 and 27. The behavior perform in the interface of **operation** initiates execution; attributes *date* and *time* record the date and time of execution. **Model operation** contains the attribute *performer* to indicate which **analyst** carries out the **operation;** *command used* records the software **command** utilized. The interfaces of two subclasses of **model operation, cluster analysis** and **spectral classification,** are presented in Figure 28 and 29. Both **cluster analysis** and **spectral classification** are children of class **numerical consumer** (i.e., they take **numerical overlays** as input). **Cluster analysis** (Figure 28) contains references to two objects produced during execution: **spectral bank** is a class of objects used in the spectral classification algorithm described in requirements analysis, and **spectral scheme** is a subclass of **class scheme. Spectral classification** (Figure 29) uses the **spectral scheme** and the **spectral bank** generated in **cluster analysis** and the input **numerical overlay** to produce a **categorical overlay** partitioned by **spectral class.**

Object Scenario Diagrams

It is with the illustration of scenarios that the analysis model comes to life for those who work in the application domain. Scenarios demonstrate how the activites performed by analysts are carried out by showing how objects work together to perform more complex behavior than they could in isolation. These collaborative arrangements are called mechanisms in (Booch89). Twenty three scenarioshave been identified from the material discussed in requirements analysis (Table 1). Nineteen have been successfully constructed and are depicted in full in Appendix III. The four scenarios which were not successfully constructed have a higher level logic than the scenarios for which construction was successful. The scenarios which were not constructed are discussed in Chapter 5. In this chapter we illustrate and discuss two scenarios that have been constructed.. We place object names in double quotes and use single underlining to highlight operation names.

Figure 30 is the object diagram depicting cluster analysis. Objects are indicated by amorphous shapes with solid borders, with the object name inside the shape. Solid lines between objects indicate that one object uses another. In other words, the object "a Cluster Analysis" uses object "a Numerical Point - tm3". The message Get value triggers an operation that is part of the interface of the class being called. Thus the interface of class **numerical point** must contain an operation called Get value. The passing of this message will cause "a Numerical Point - tm3" to carry out that operation and return a value.

SCENARIO LIST

Scenario Name	Constructed
Registration	Yes
Resample	Yes
Projection Transformation	No
Convolution	Yes
Contrast Stretch	Yes
Overlay Algebra	Yes
Interpolation	Yes
MtClim	No
Merge	Yes
Majority Filter	Yes
EcoModel Assignment	Yes
Cluster Analysis	Yes
Spectral Classification	Yes
Cover Training - traditional	Yes
Cover Training - fuzzy	Yes
Cover Classification - traditional	Yes
Cover Classification - fuzzy	Yes
Topographic Partitioning	No
Forest BGC	No
Elevation Data Acquisition	Yes
Spectral Data Acquisition	Yes
Climate Data Acquisition	Yes
Field Data Acquisition	Yes

Table 1

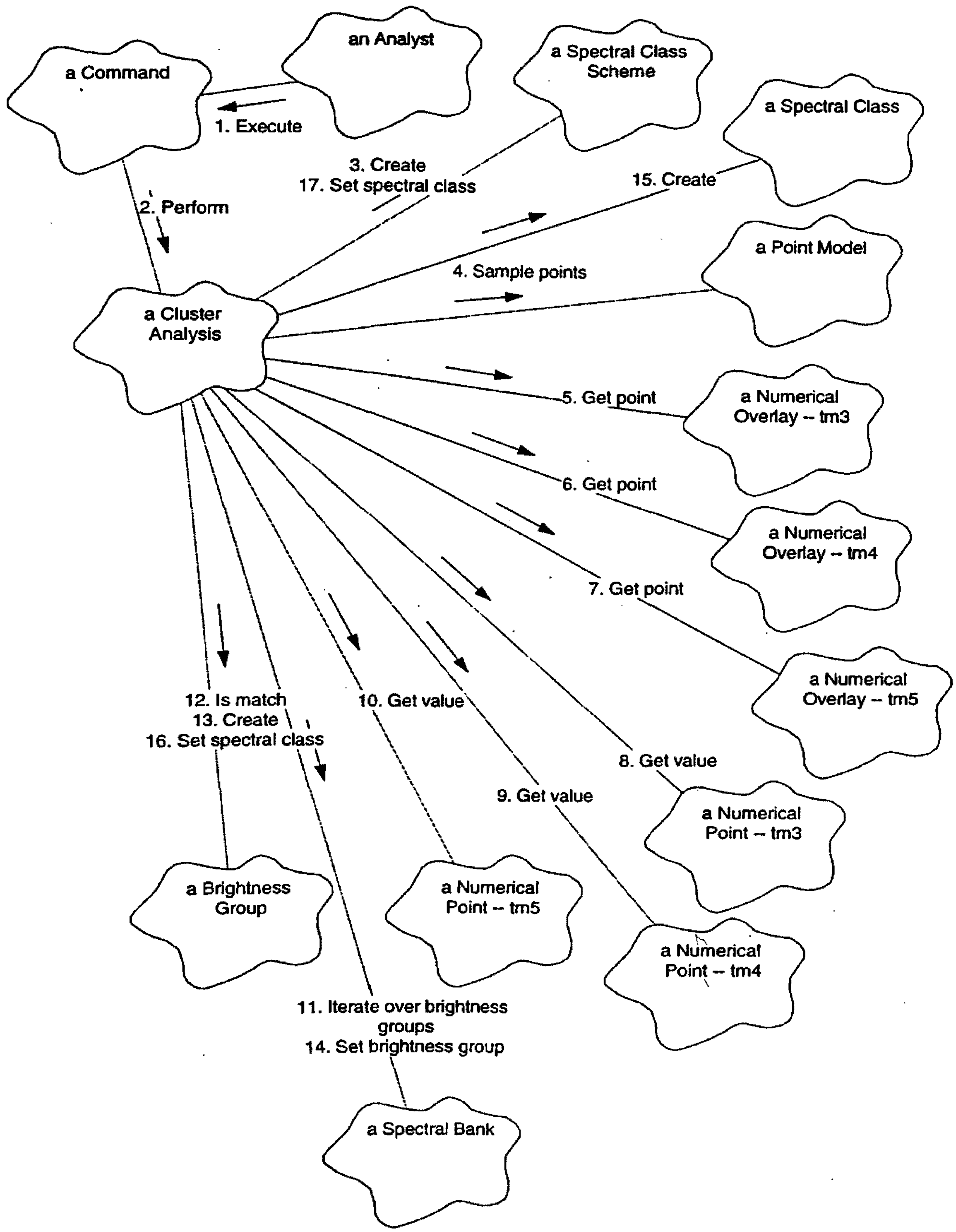

an Analyst
a Command
a Spectral Class Scheme
a Spectral Class
1. Execute
3. Create
17. Set spectral class
15. Create
2. Perform
a Point Model
4. Sample points
a Cluster Analysis
5. Get point
a Numerical Overlay -- tm3
6. Get point
a Numerical Overlay -- tm4
7. Get point
a Numerical Overlay -- tm5
12. Is match
13. Create
16. Set spectral class
10. Get value
8. Get value
9. Get value
a Numerical Point -- tm3
a Brightness Group
a Numerical Point -- tm5
a Numerical Point -- tm4
11. Iterate over brightness groups
14. Set brightness group
a Spectral Bank

Figure 30

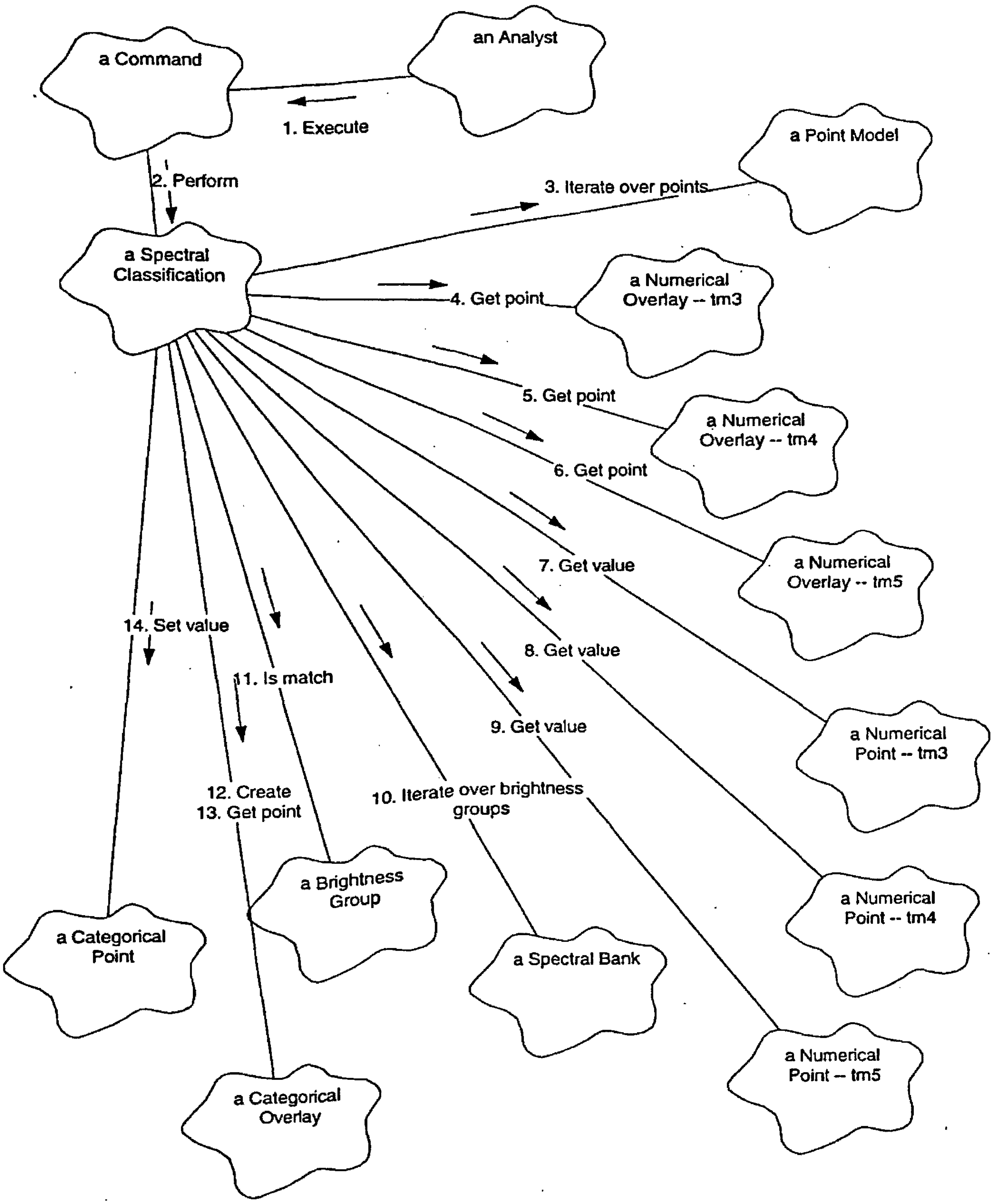

an Analyst
a Command
1. Execute
a Point Model
2. Perform
3. Iterate over points
a Spectral Classification
4. Get point
a Numerical Overlay -- tm3
5. Get point
a Numerical Overlay -- tm4
6. Get point
a Numerical Overlay -- tm5
7. Get value
8. Get value
9. Get value
14. Set value
11. Is match
12. Create
13. Get point
10. Iterate over brightness groups
a Numerical Point -- tm3
a Numerical Point -- tm4
a Numerical Point -- tm5
a Brightness Group
a Spectral Bank
a Categorical Point
a Categorical Overlay

Figure 31

All scenarios begin with object "an Analyst" passing the message Execute to "a Command", which in turn passes the message Perform to an object representing the modeling operation depicted in the scenario. "A Cluster analysis" passes message Create to "a Spectral Class Scheme" and the message Sample points to "a Point Model". The message Get point is passed to each of the three **numerical overlays** being sampled. Get value is sent to each of the **numerical points** being sampled. The message Iterate over brightness groups is passed to "a Spectral Bank". The values of each sampled point are passed to "a Brightness Group" with the message Is match, which returns a value indicating whether the newly sampled point represents the same class as "a Brightness Group". If the point represents a new **brightness group**, then an object of class **brightness group** is created, and the reference to this **brightness group** is set in "a Spectral Bank". Finally, "a Spectral Class" is created and linked to the newly created "a Brightness Group". This process continues through all sample points.

The object "a cluster analysis" is the executive (i.e., the controlling entity) in this mechanism since it controls the sequence of messages and makes decisions based on values returned from other objects. The object representing an operation will serve as the executive in all of our scenarios.

In Figure 31, the scenario for spectral classification is shown. "A spectral classification" passes message Iterate over points to "a Point Model". The messages Get point and Get value are used to obtain the values for the **numerical overlays** being used in the classification. For each point, the message Iterate over brightness groups is sent to "a Spectral Bank". The message Is match, sent to each **brightness group**, until a match is found. An object of class **categorical overlay** is created, a reference to the appropriate **categorical point** is obtained, and the message Set value assigns the appropriate class identifier to the point.

Summary of Domain Analysis

The appendices to this document contain the complete results of domain analysis: a set of class/relationship diagrams (Appendix I), a set of formal class specifications (Appendix II) and a set of object scenarios (Appendix III).

Although this chapter focuses on three groups of classes -- spatial entities, ecosystem entities and operations -- there are many significant entities in those groups that are not described. For example, a set of relationships and operations are defined to give a point or subregion access to its neighbors in "modeling space". Additionally, many of the classes describing operations are not described here. However, a great deal of similarity among these classes leads us to conclude that depicting the interfaces of a few key classes in the operation portion of the hierarchy is adequate for the overview provided in this chapter. A significant class group that we do not discuss deals with classification entities. This class group includes the components of **class schemes**, the parent/subclass hierarchy depicting different kinds of **class** (e.g., **cover type class, soil class**), and the many entities used in the classification operations described in requirements analysis. Although this class group is important, many of its members are discussed in our treatment of spatial and ecosystem entities and model operations. Additional class groups include ecosystem descriptor entities, data acquisition entities, and window and descriptor entities. The remaining class groups - software, documentation and humans - are less central to this discussion. They represent the human actors that carry out modeling operations, the tools that make those operations possible, and the knowledge used in support of, and obtained during, modeling activities. However, the classes and relationships describing spatial and ecosystem entities, modeling operations and classification entities provide the core of significant abstraction in this domain.

The object scenario diagrams (Appendix III) show how the classes defined in this analysis will actually be used in performing the key processing operations used by eco-system modelers. This is where the "action" is, in understanding how an object-oriented approach to modeling can contribute to our knowledge of a specific application domain.

CHAPTER V

HYPOTHESIS EVALUATION

We have constructed scenarios depicting most of the processes discussed in our requirements analysis. These scenarios show how modeling processes can be carried out given the structure revealed in our class hierarchy. During scenario construction, we found that two actions were necessary with respect to the hierarchy -- we had to modify the interface of class by adding attributes and operations, and less often, we created a new class.

The only scenarios not completed -- for the extrapolation of climate data performed by MtClim, the simulation of ecosystem processes by Forest BGC, projection transformation and topographic partitioning -- are for processes for which the available literature discusses the internal logic in adequate detail. In the interest of providing a concise characterization of the domain, we did not incorporate this knowledge into our requirements analysis. In general, the classes depicted in the domain analysis model, with the exception of classes representing key modeling activities, are "primitive" in nature. That is, the logic of their behavior is at a fairly simple level. Many of the modeling activities for which scenarios have been successfully constructed have a higher level logic than these "primitive" classes. However, in these cases, the "logic gap" between the modeling activity and the classes used in the activity is not great (e.g. Merge). However, the modeling activities captured in Mtclim, Forest BGC and topographic partitioning exhibit a logic that is considerably higher than that of the classes used in these activities. This greater "logic gap" prevented scenario construction in these cases.

We believe that many of the scenario construction problems associated with Mtclim and topographic partitioning can be handled by decomposing these high-level operations into procedural components. MtClim (Hungerford89) is composed of four sub processes which compute solar radiation, temperature, humidity and precipitation. Each of these sub-processes can be modeled as component operations of class **mtclim**. Topographic partitioning consists of three major sub-processes. The first defines the drainage by recursively examing points in the **watershed** to determine, for each point, the number of upstream points. Secondly, the **stream network** must be delineated, and thirdly the **watershed** must be partitioned into **hillslopes**. Each of these sub-processes can be represented as a component of class **topo partition.**

Forest BGC is a complex operation that computes flows of carbon, nitrogen and water through forest ecosystems. The procedural decomposition discussed above for MtClim and topographic partitioning may not be appropriate in this case. Instead, a deeper examination of the logic of this modeling activity may be needed before a productive approach can be found. In particular, the treatment of the compartments, state variables, state equations and flows comprising Forest BGC may yield an object-oriented model that can be applied to a wide range of process models.

We conclude that our ability to define scenarios for most processes, without radically changing our existing hierarchy, indicates that our domain model is capable of supporting the necessary modeling activities, and is resilient enough to absorb many changes. The fact that not all necessary scenarios are constructed here is not a failure of our modeling methodology or of the concept of domain analysis. Rather it reflects that even more detailed evaluation of some parts of the application domain may be necessary for the more detailed requirements analysis needed in the development of a particular application. We believe our success in scenario construction where our requirements analysis provides adequate information constitutes proof of our hypothesis.

CHAPTER VI

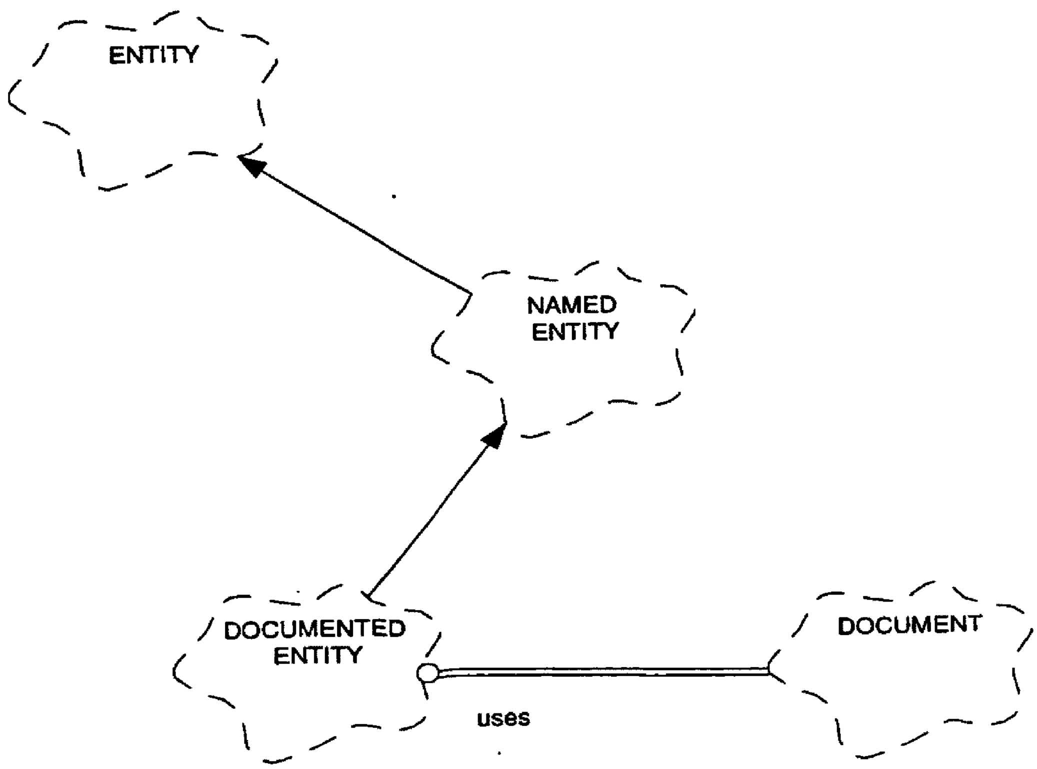
CONCLUSIONS

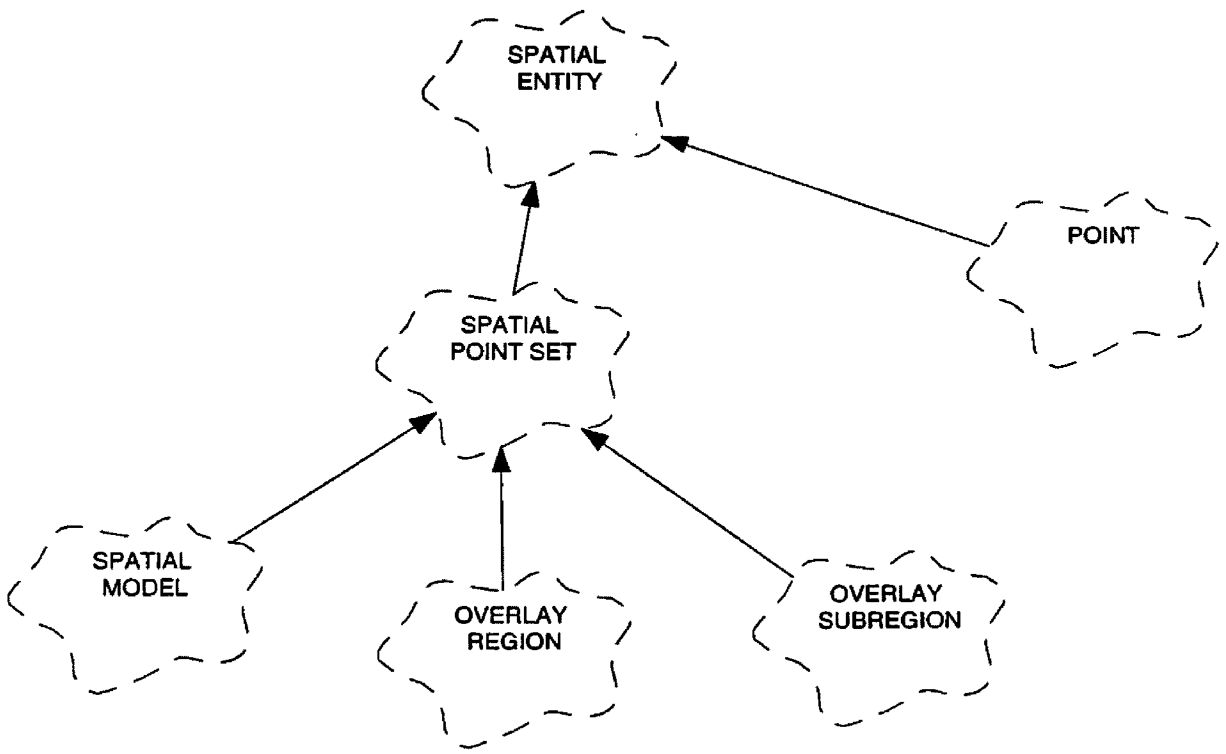
In keeping with the principles of the spiral model of design (Boehm88), this work is essentially our "first draft" in the attempt to provide a comprehensive view of ecosystem modeling and will no doubt be challenged, revised, and extended in future work. It has been suggested that the hierarchy be extended so that **numerical** and **categorical overlay** subclasses are defined on the basis of data content (e.g., class elevation overlay, aspect overlay). Currently, an overlay containing elevation data is an object of class **numerical overlay**. This extension would provide a much richer set of class names that reflect more closely the terminology used by workers in this application domain. We believe that our handling of modeling processes is a useful approach, but we are aware that additional work is needed, especially in cases where a significant level of human judgement is involved (e.g., supervised training). Additionally, a deeper examination of ecosystem process modeling is needed to adequately handle the needs of process modelers.

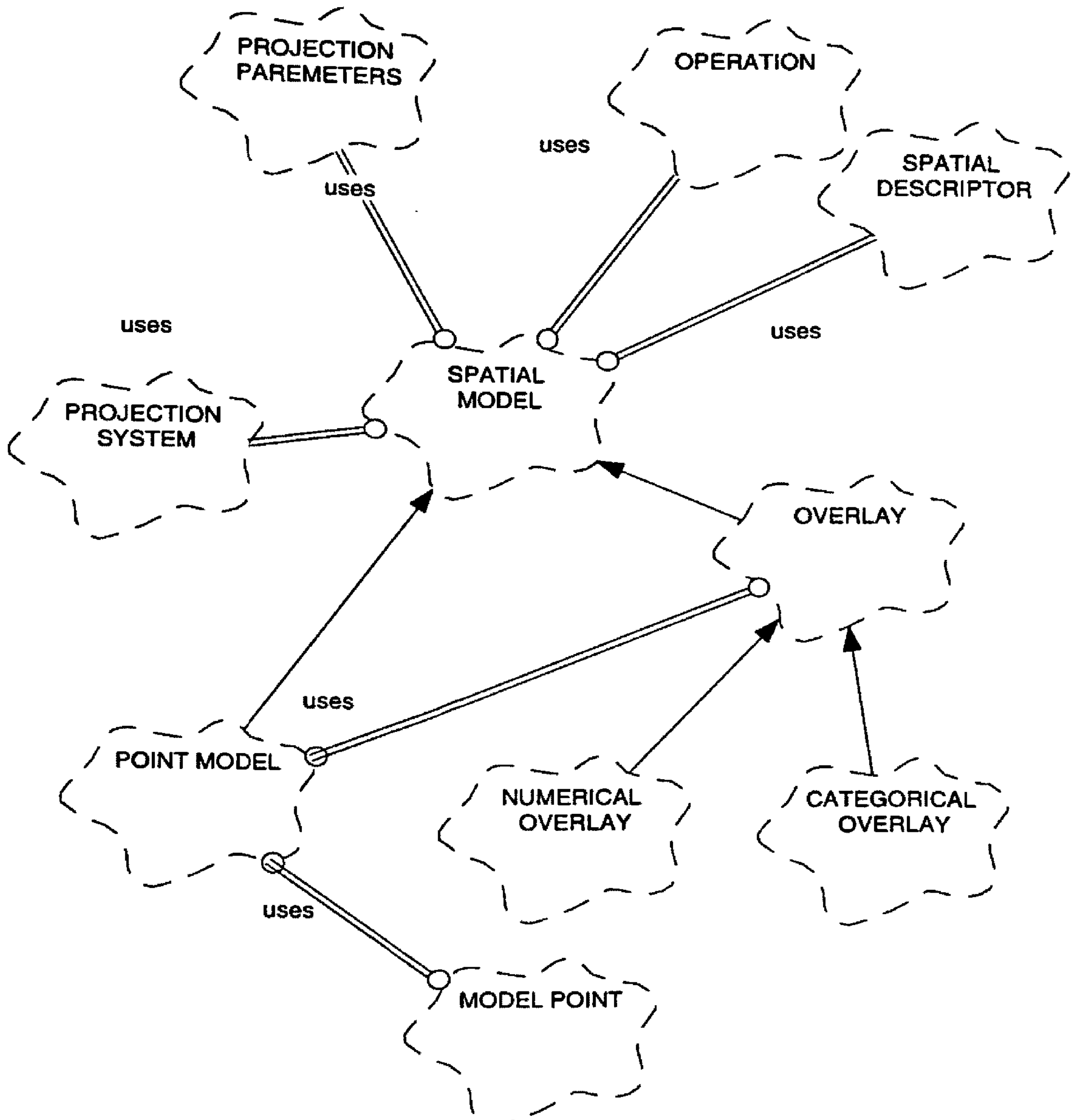
The work presented here provides a graphical and textual model of the domain of ecosystem modeling. This depiction of the modeling entities used by ecosystem modelers will serve as the foundation of the information design for the Ecosystem Information System (EIS) and will help them to better manage the models, software, modeling activities, and other tools used in their daily work. We hope that this work will help modelers working in this application domain to increase their productivity and facilitate the advance of their discipline.

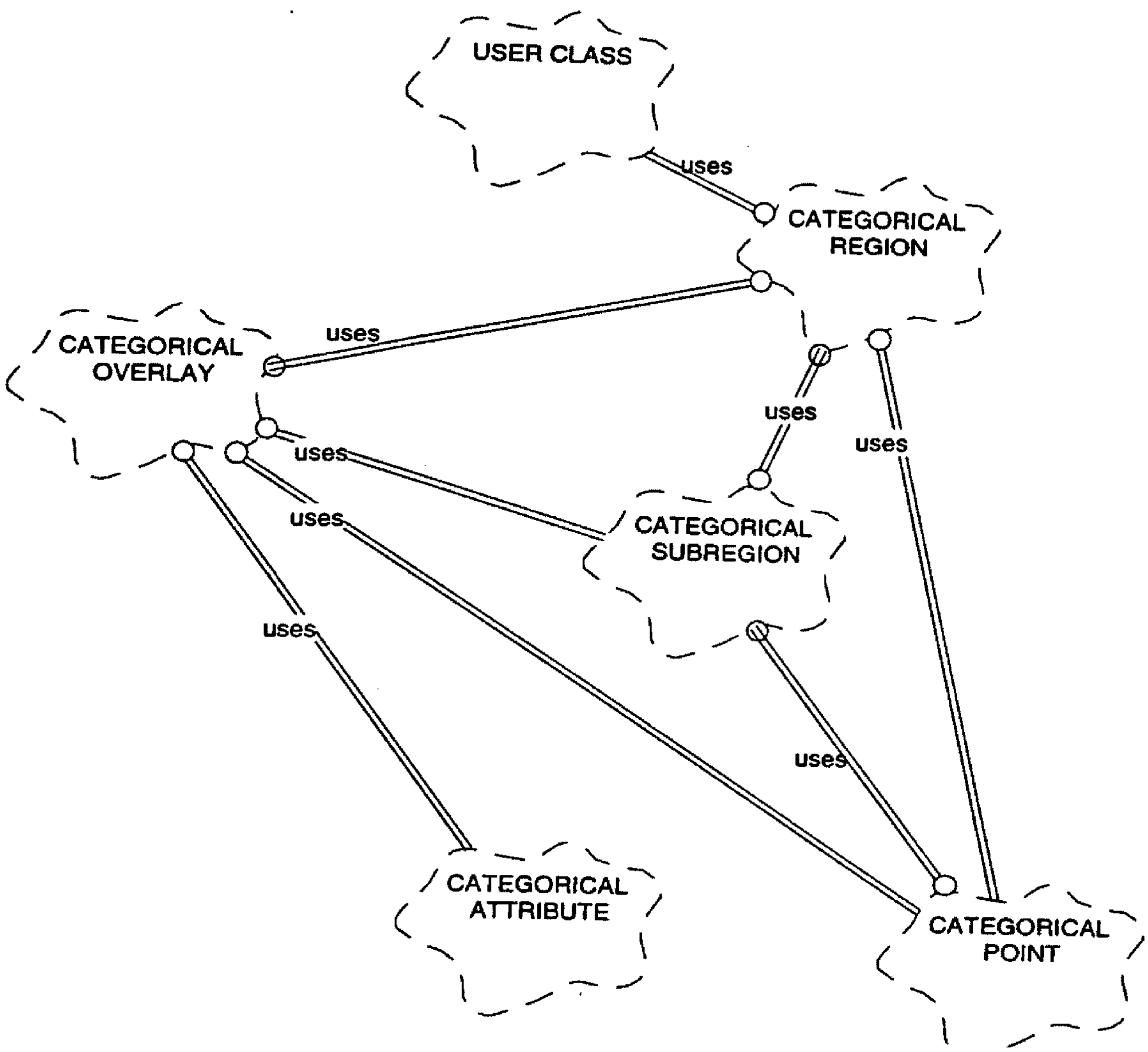
APPENDIX I
CLASS AND RELATIONSHIP DIAGRAMS

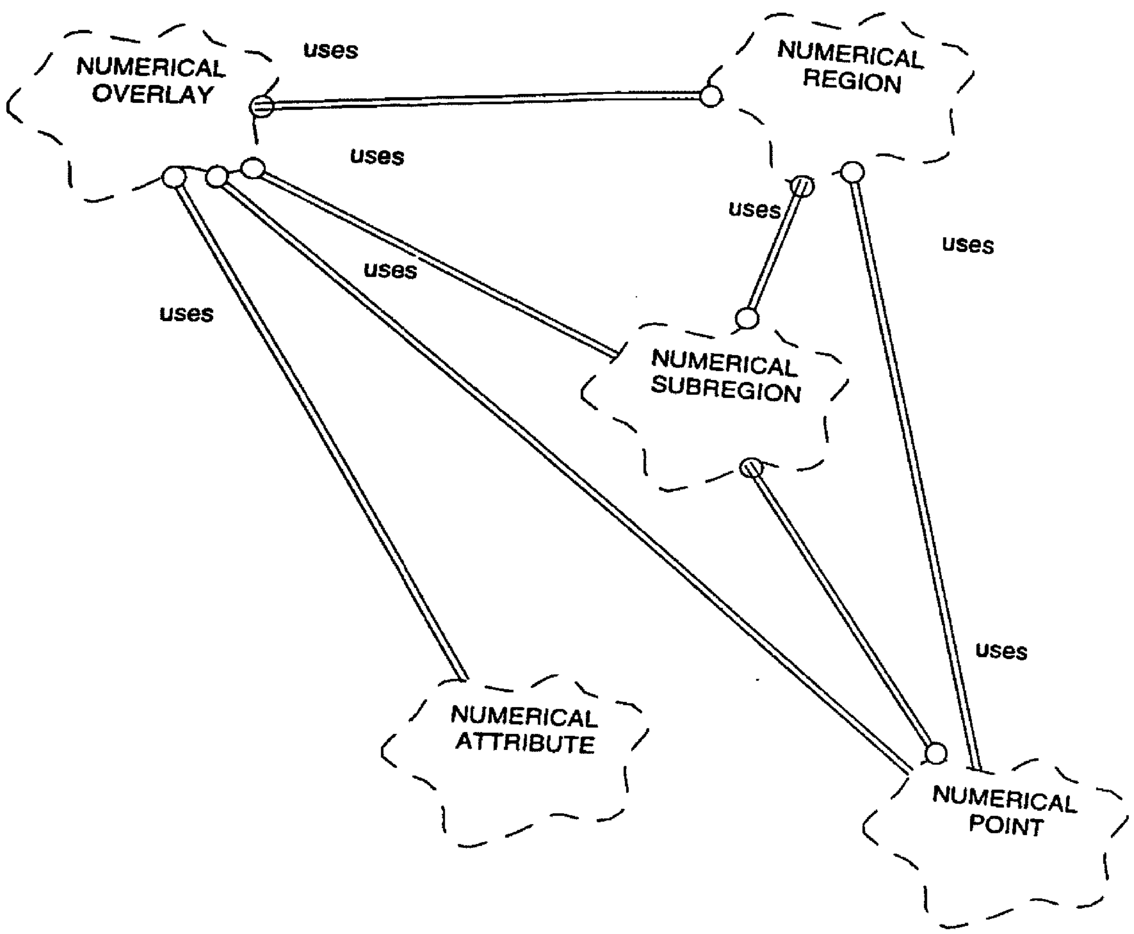
TABLE OF CONTENTS


ENTITY
NAMED ENTITY
DOCUMENTED ENTITY
DOCUMENT
uses

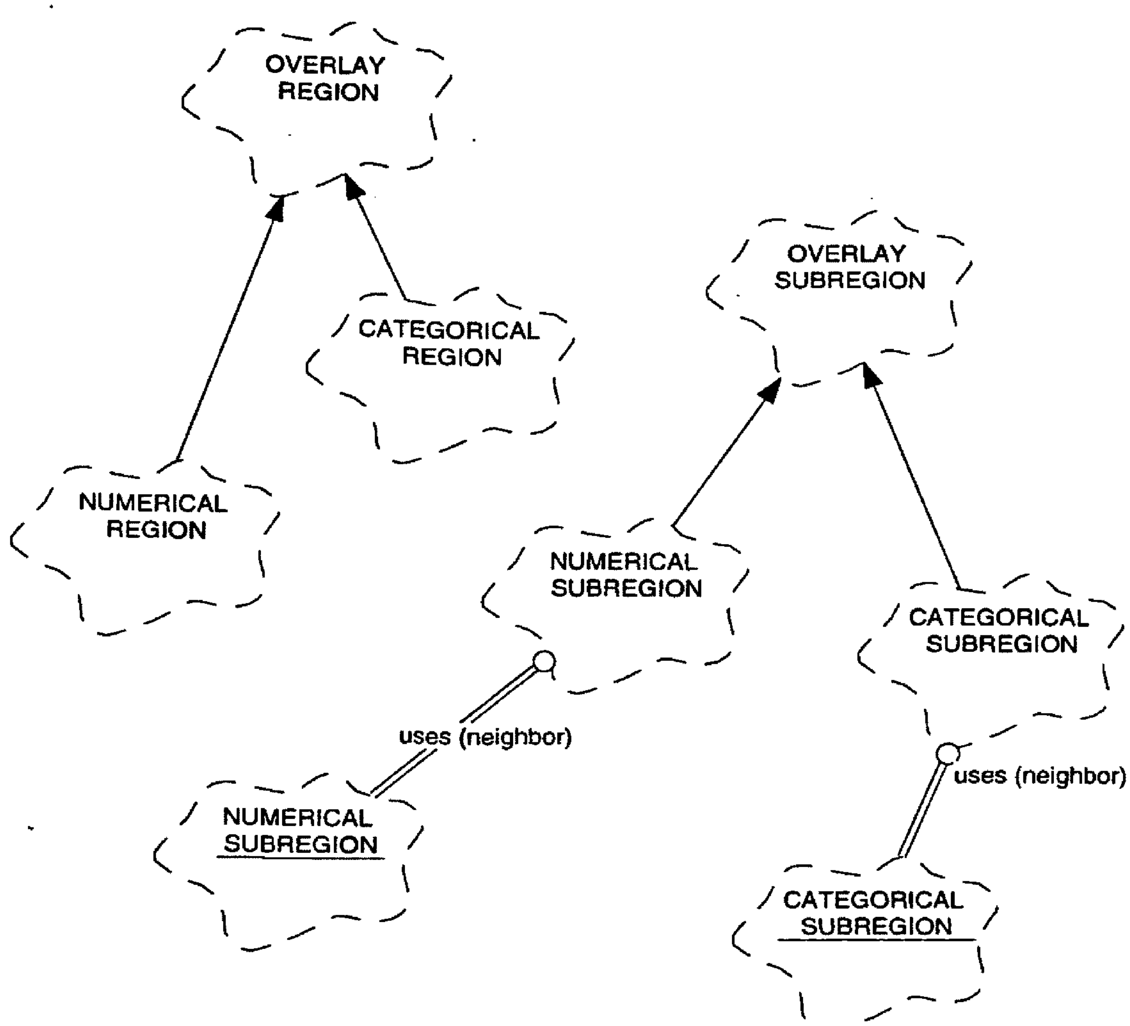

SPATIAL
ENTITY
POINT
SPATIAL
POINT SET
SPATIAL
MODEL
OVERLAY
REGION
OVERLAY
SUBREGION

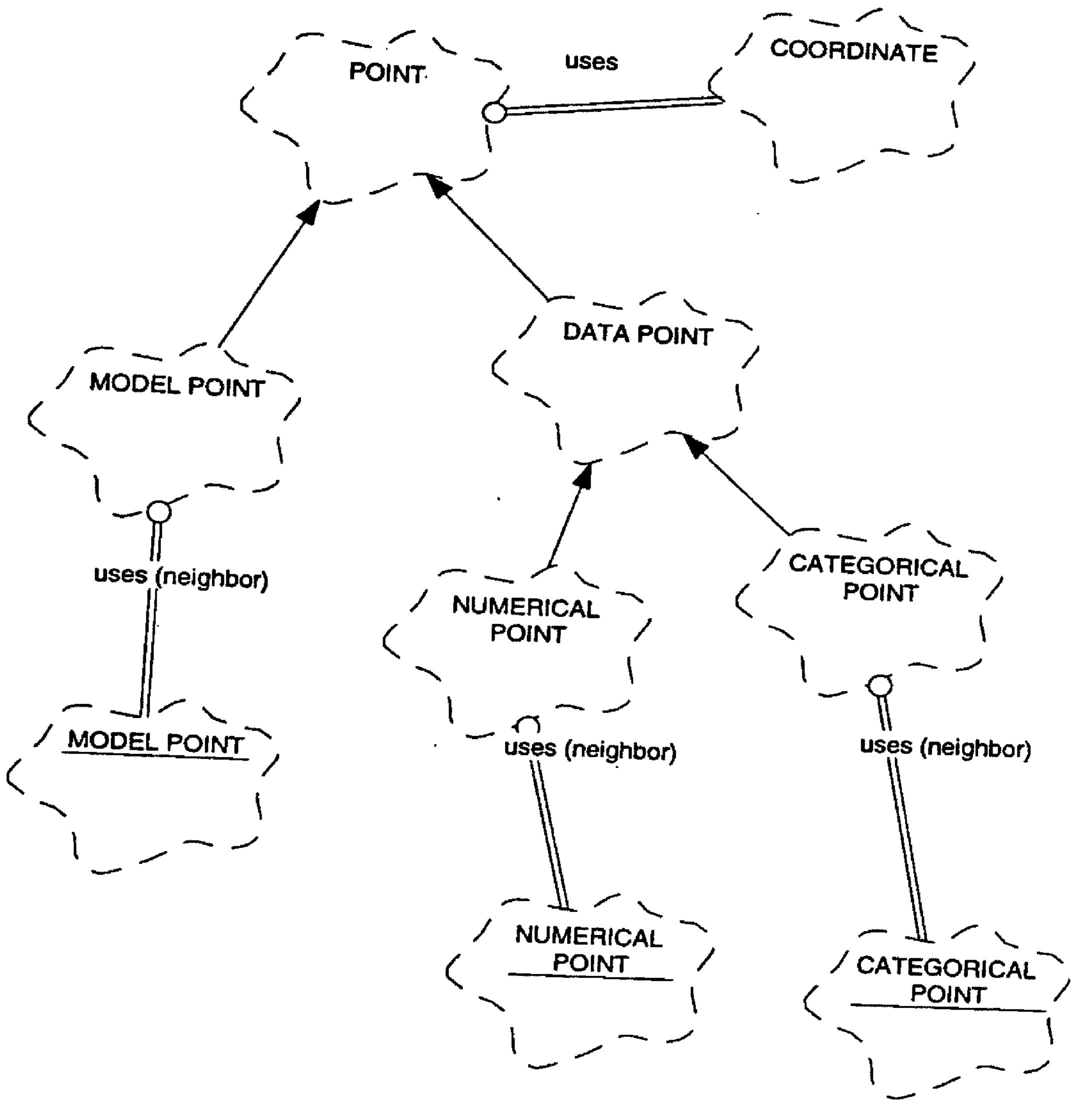

PROJECTION PAREMETERS
OPERATION
SPATIAL DESCRIPTOR
uses
uses
uses
uses
SPATIAL MODEL
PROJECTION SYSTEM
OVERLAY
uses
POINT MODEL
NUMERICAL OVERLAY
CATEGORICAL OVERLAY
uses
MODEL POINT

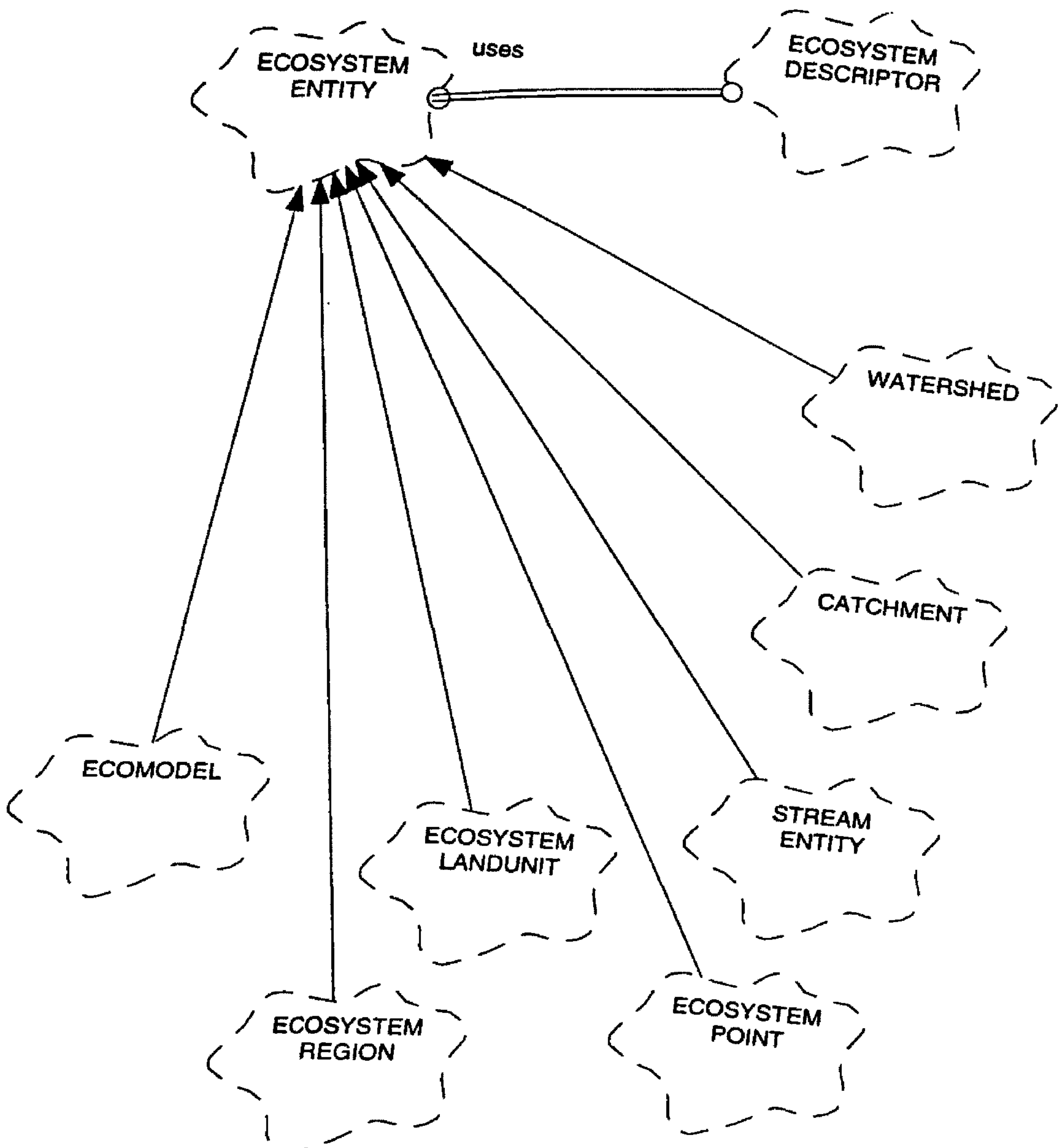

USER CLASS
uses
CATEGORICAL REGION
CATEGORICAL OVERLAY
uses
uses
uses
uses
uses
uses
CATEGORICAL SUBREGION
uses
CATEGORICAL ATTRIBUTE
CATEGORICAL POINT

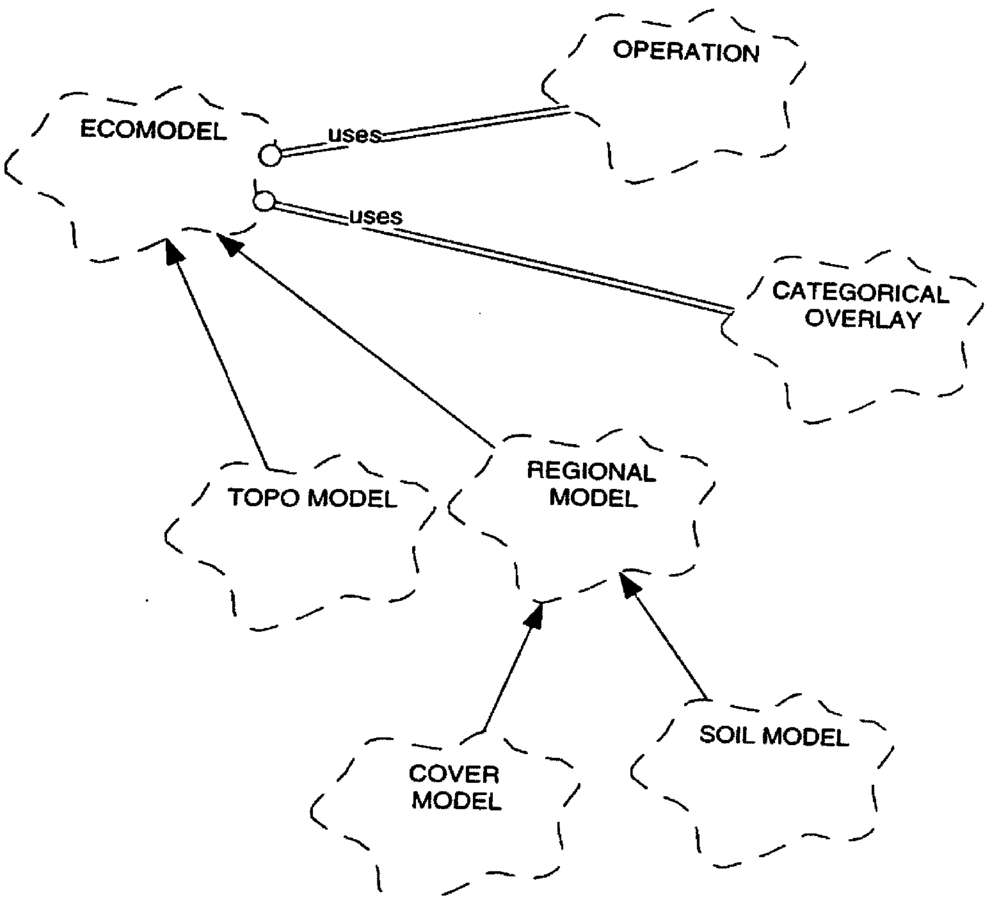


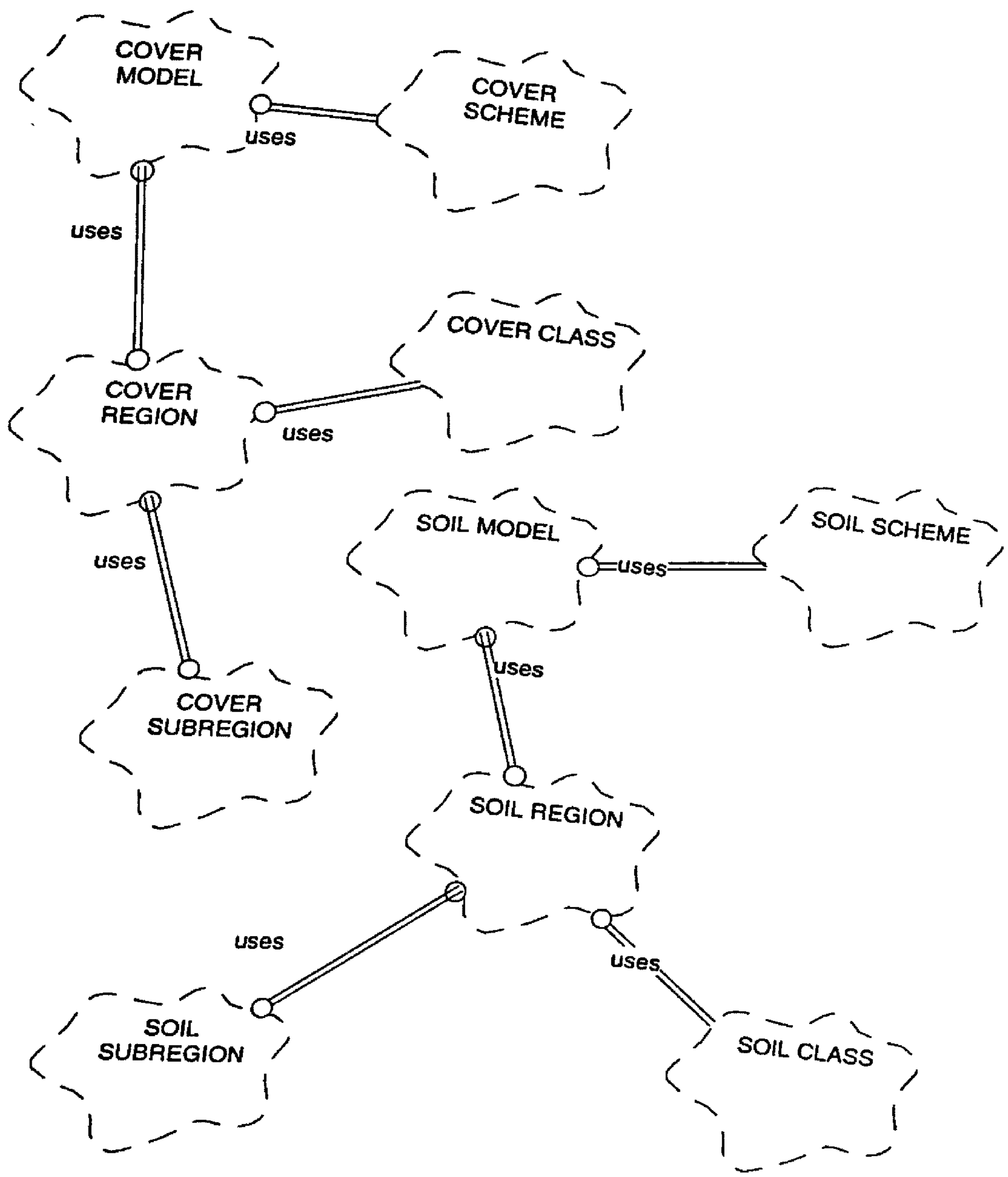
NUMERICAL OVERLAY
uses
NUMERICAL REGION
uses
uses
uses
uses
uses
NUMERICAL SUBREGION
uses
NUMERICAL ATTRIBUTE
NUMERICAL POINT

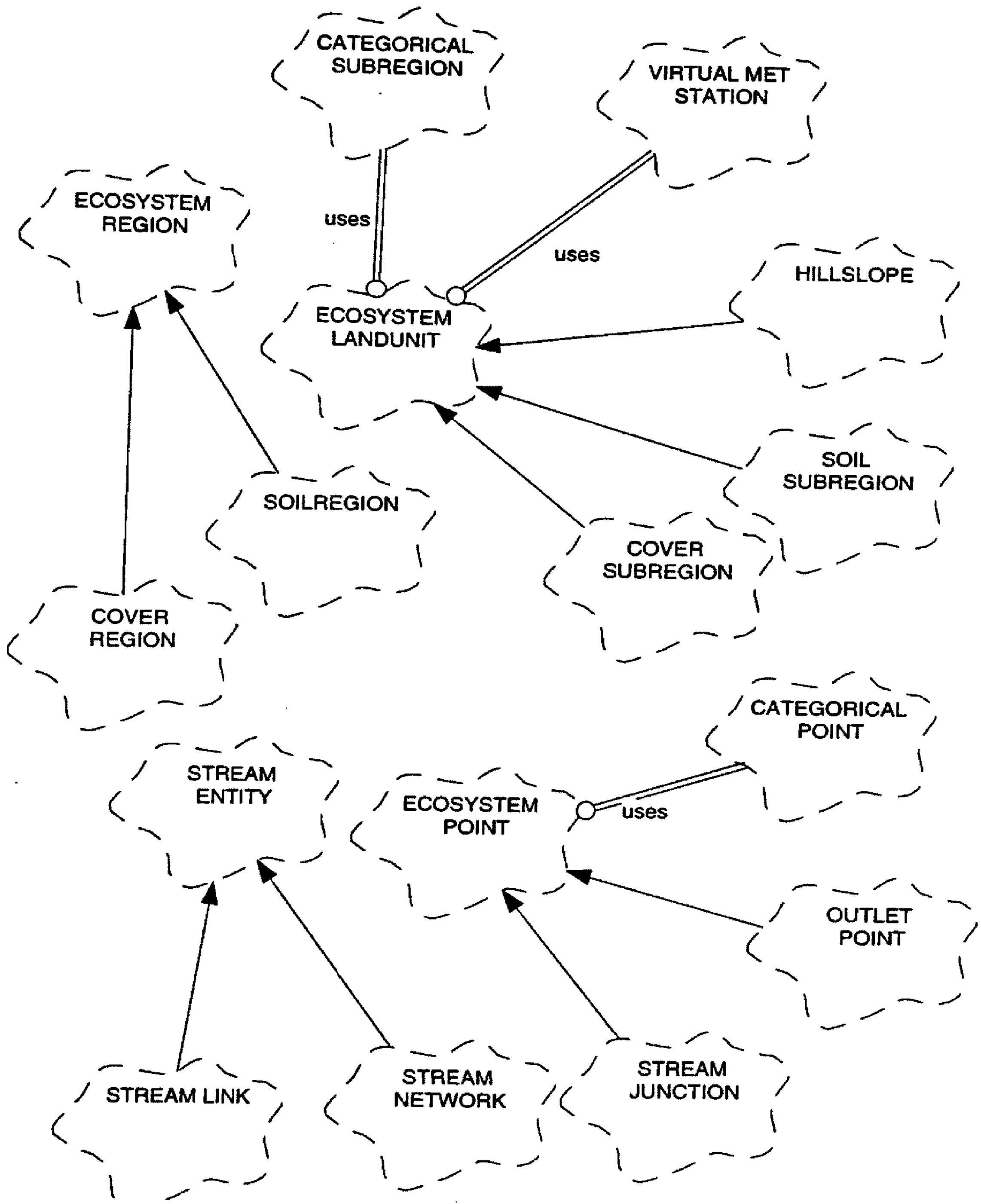

OVERLAY REGION
CATEGORICAL REGION
NUMERICAL REGION
OVERLAY SUBREGION
NUMERICAL SUBREGION
CATEGORICAL SUBREGION
uses (neighbor)
NUMERICAL SUBREGION
uses (neighbor)
CATEGORICAL SUBREGION

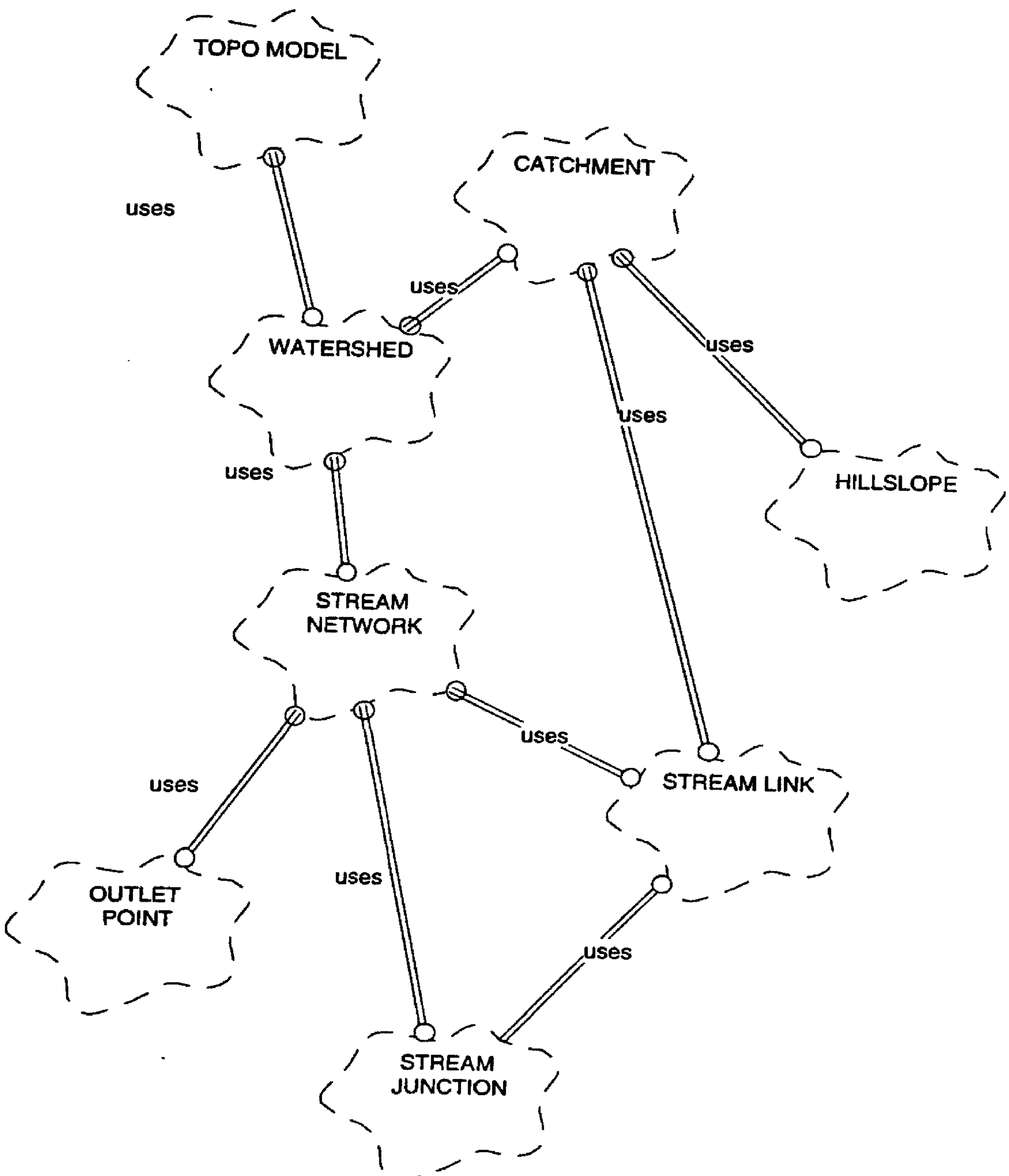

POINT
uses
COORDINATE
MODEL POINT
DATA POINT
uses (neighbor)
NUMERICAL POINT
CATEGORICAL POINT
MODEL POINT
uses (neighbor)
uses (neighbor)
NUMERICAL POINT
CATEGORICAL POINT

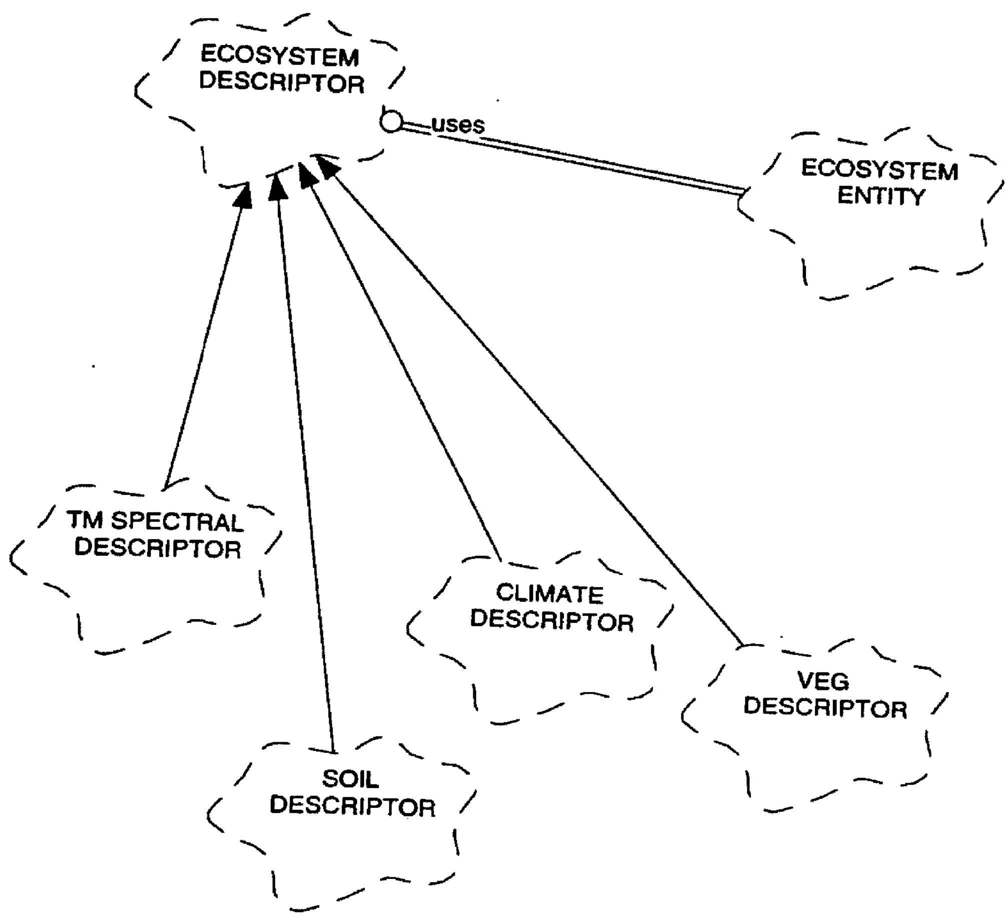

ECOSYSTEM ENTITY
uses
ECOSYSTEM DESCRIPTOR
WATERSHED
CATCHMENT
ECOMODEL
ECOSYSTEM LANDUNIT
STREAM ENTITY
ECOSYSTEM REGION
ECOSYSTEM POINT

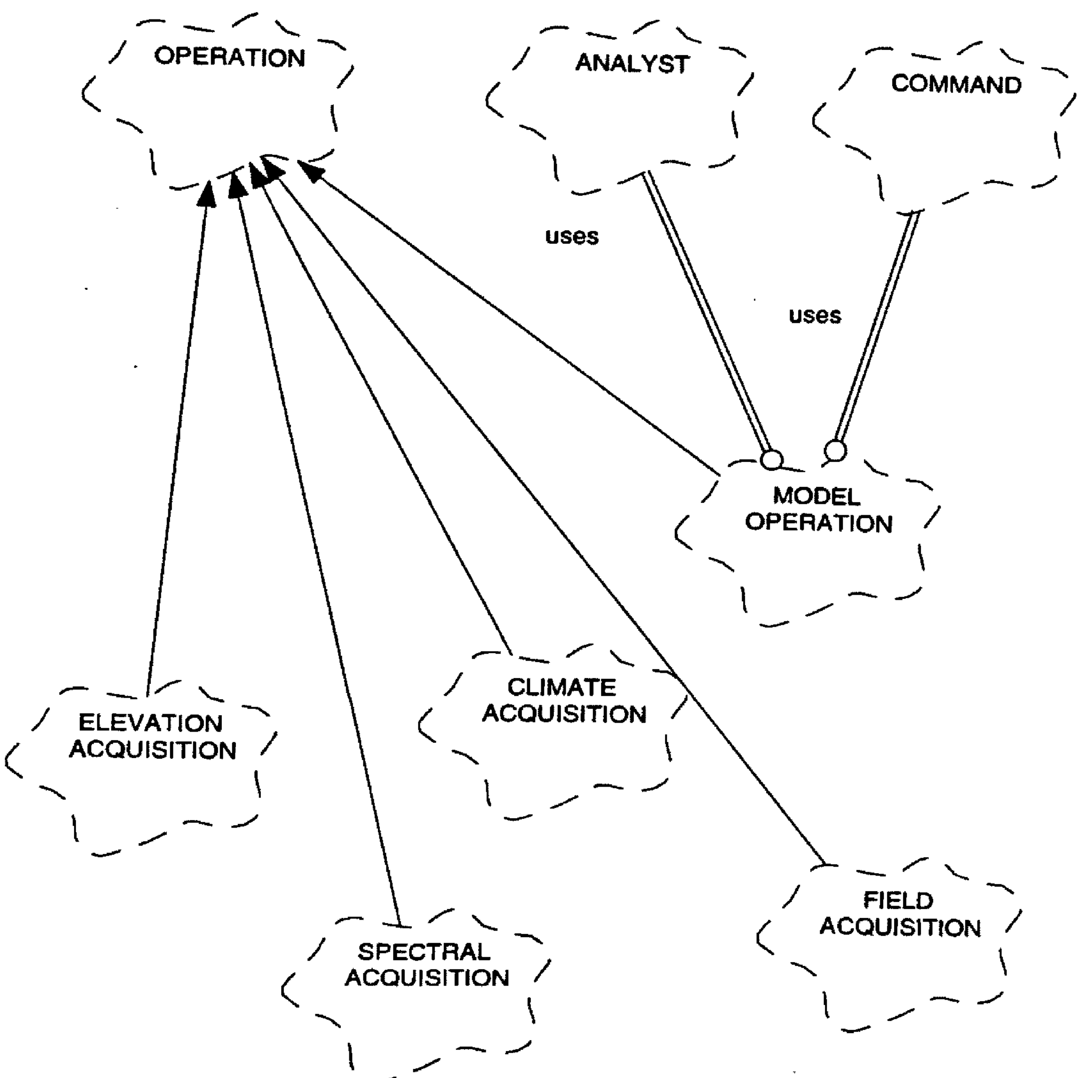

OPERATION
ECOMODEL
uses
uses
CATEGORICAL
OVERLAY
TOPO MODEL
REGIONAL
MODEL
COVER
MODEL
SOIL MODEL

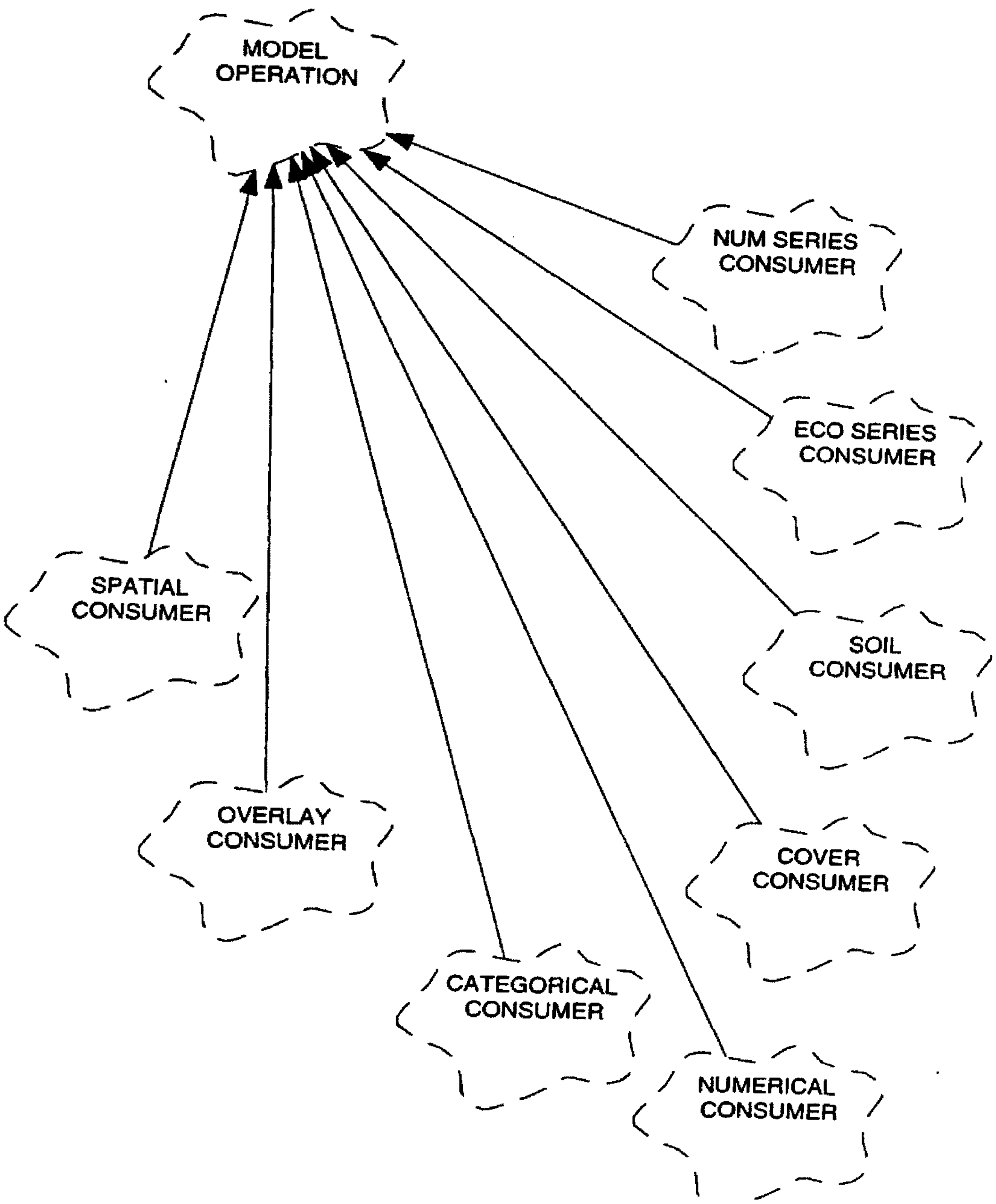

COVER MODEL
uses
COVER SCHEME
uses
COVER CLASS
COVER REGION
uses
uses
SOIL MODEL
uses
SOIL SCHEME
uses
COVER SUBREGION
SOIL REGION
uses
uses
SOIL SUBREGION
SOIL CLASS

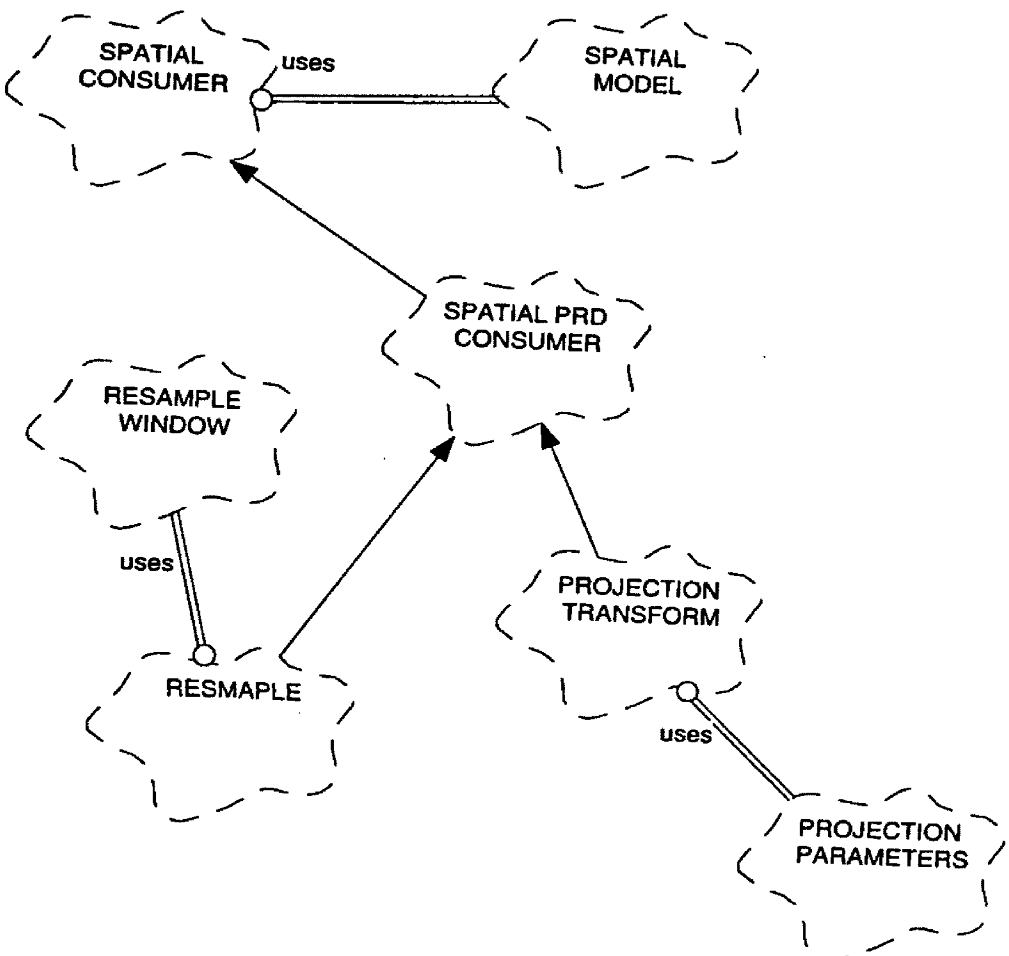

CATEGORICAL SUBREGION
VIRTUAL MET STATION
ECOSYSTEM REGION
uses
uses
HILLSLOPE
ECOSYSTEM LANDUNIT
SOIL SUBREGION
SOILREGION
COVER SUBREGION
COVER REGION
CATEGORICAL POINT
STREAM ENTITY
ECOSYSTEM POINT
uses
OUTLET POINT
STREAM LINK
STREAM NETWORK
STREAM JUNCTION

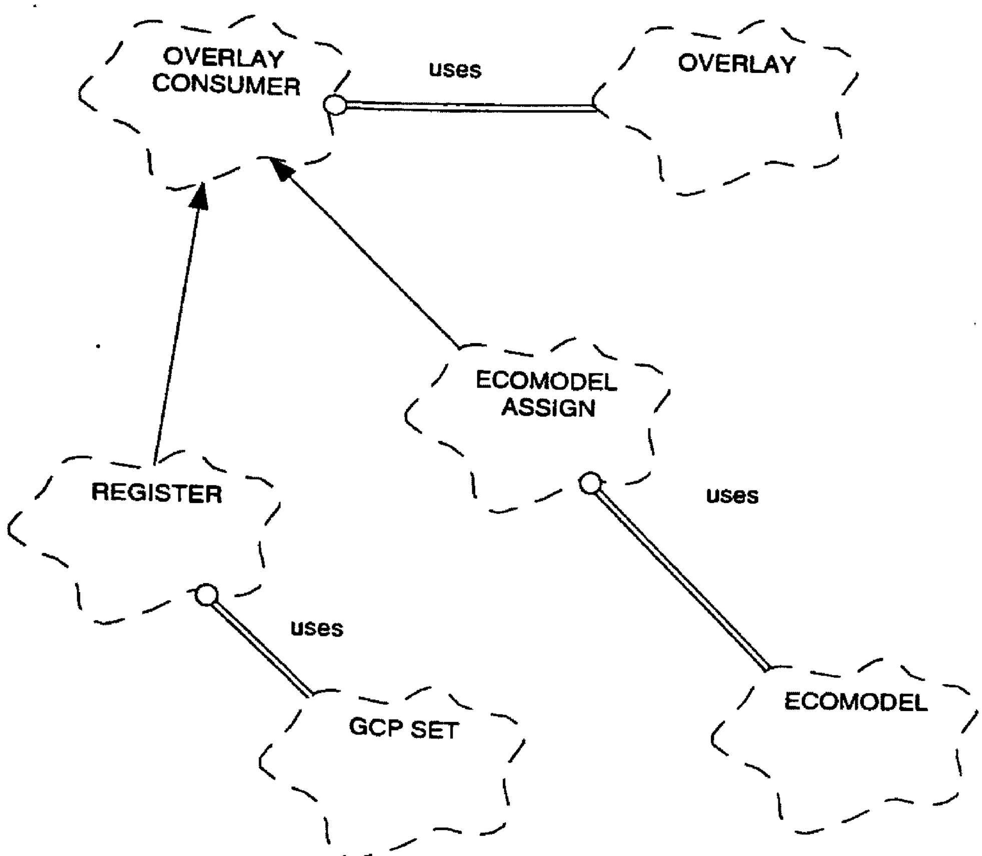

TOPO MODEL
uses
CATCHMENT
uses
WATERSHED
uses
uses
uses
HILLSLOPE
STREAM
NETWORK
uses
uses
STREAM LINK
OUTLET
POINT
uses
uses
STREAM
JUNCTION

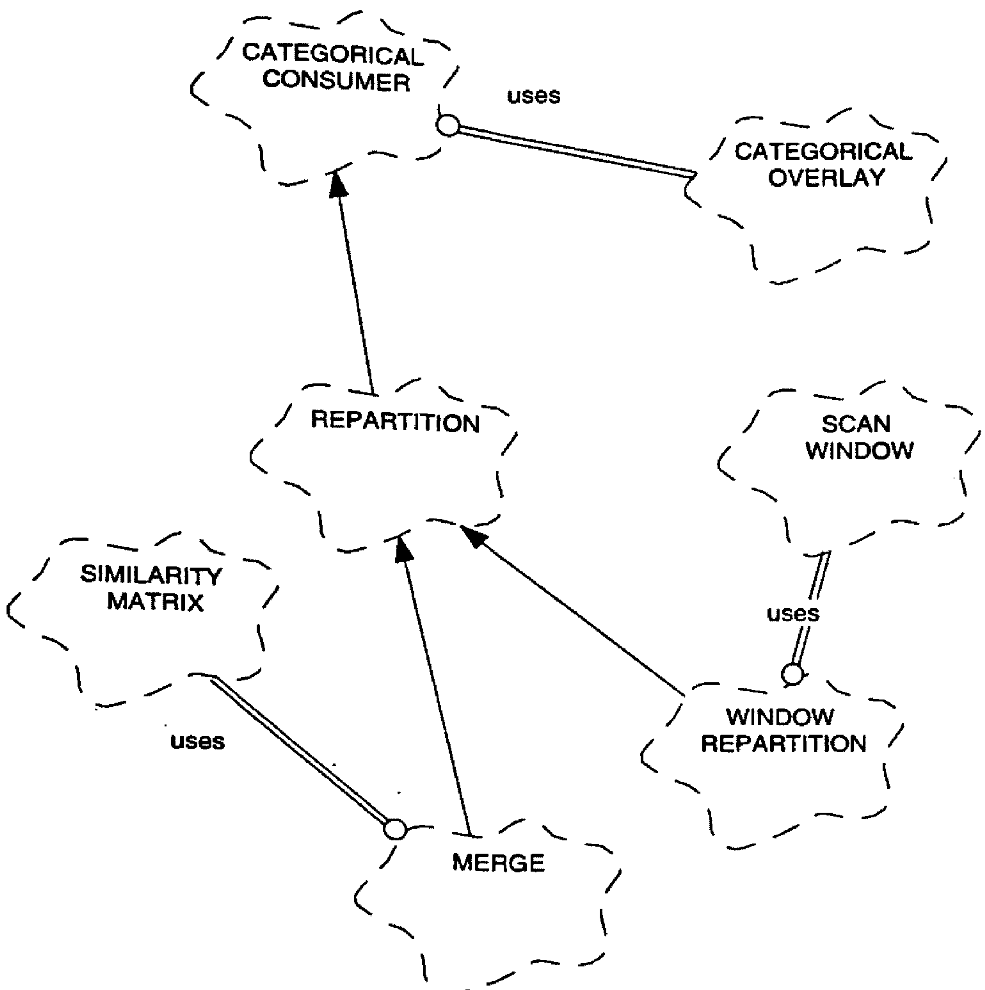

ECOSYSTEM DESCRIPTOR
uses
ECOSYSTEM ENTITY
TM SPECTRAL DESCRIPTOR
CLIMATE DESCRIPTOR
VEG DESCRIPTOR
SOIL DESCRIPTOR

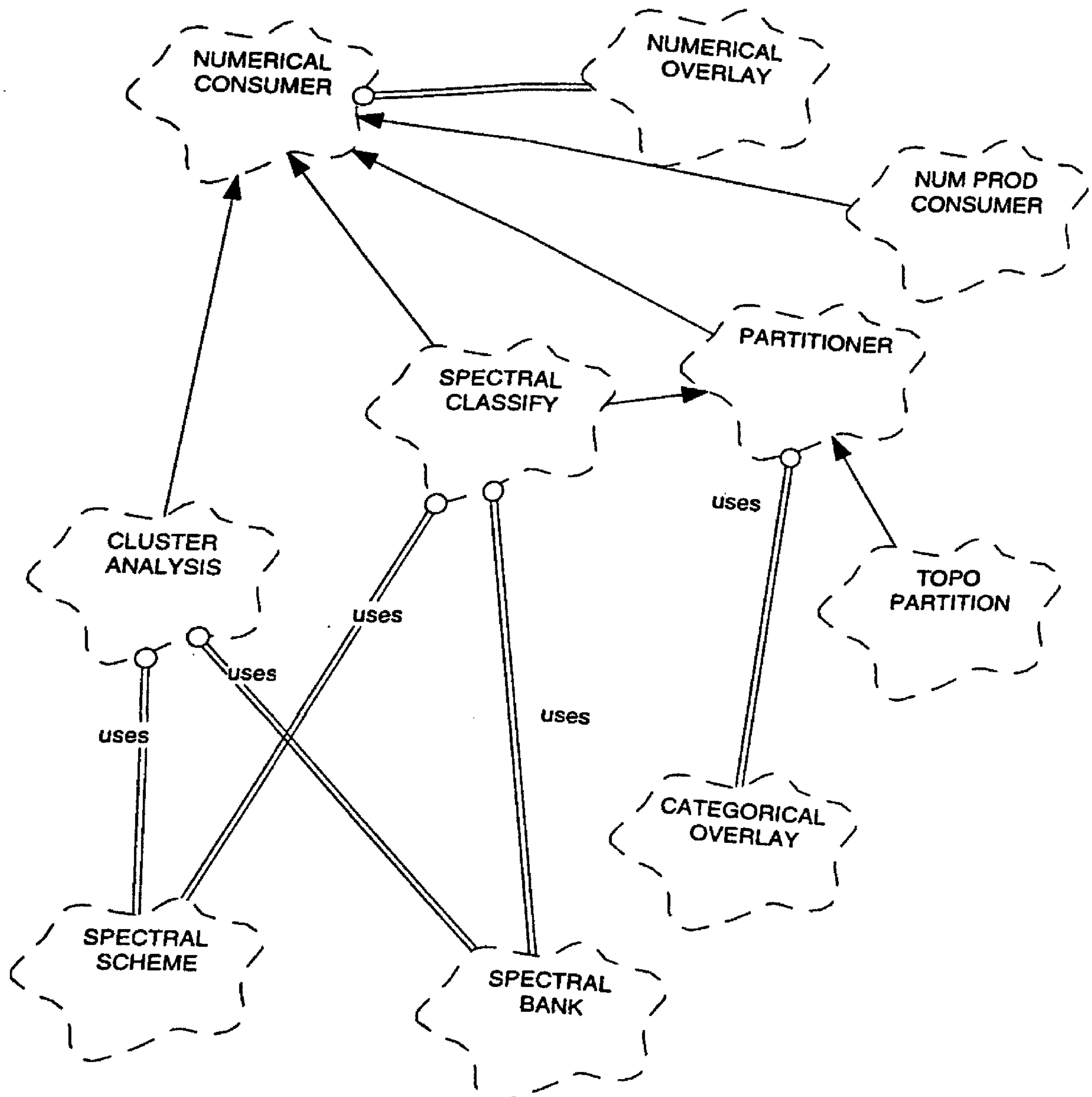

OPERATION
ANALYST
COMMAND
uses
uses
MODEL
OPERATION
ELEVATION
ACQUISITION
CLIMATE
ACQUISITION
SPECTRAL
ACQUISITION
FIELD
ACQUISITION

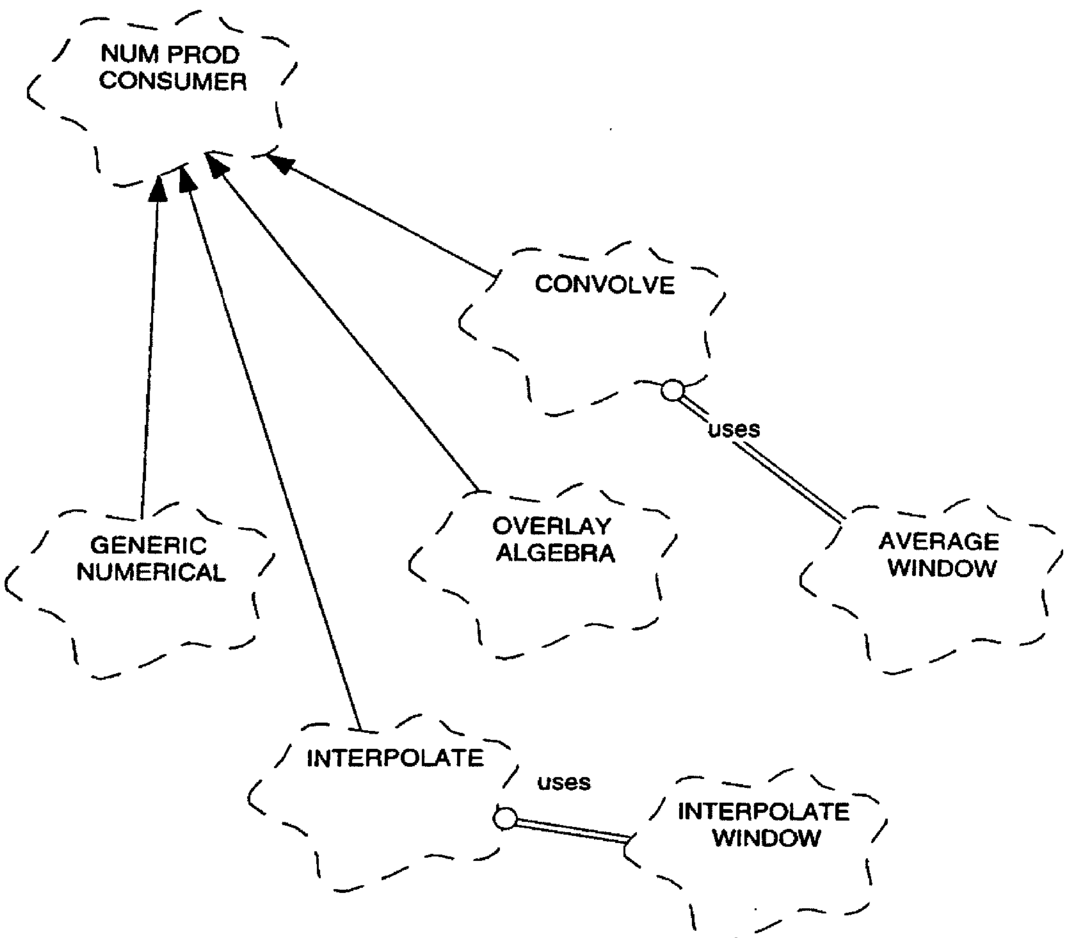

MODEL
OPERATION
NUM SERIES
CONSUMER
ECO SERIES
CONSUMER
SPATIAL
CONSUMER
SOIL
CONSUMER
OVERLAY
CONSUMER
COVER
CONSUMER
CATEGORICAL
CONSUMER
NUMERICAL
CONSUMER

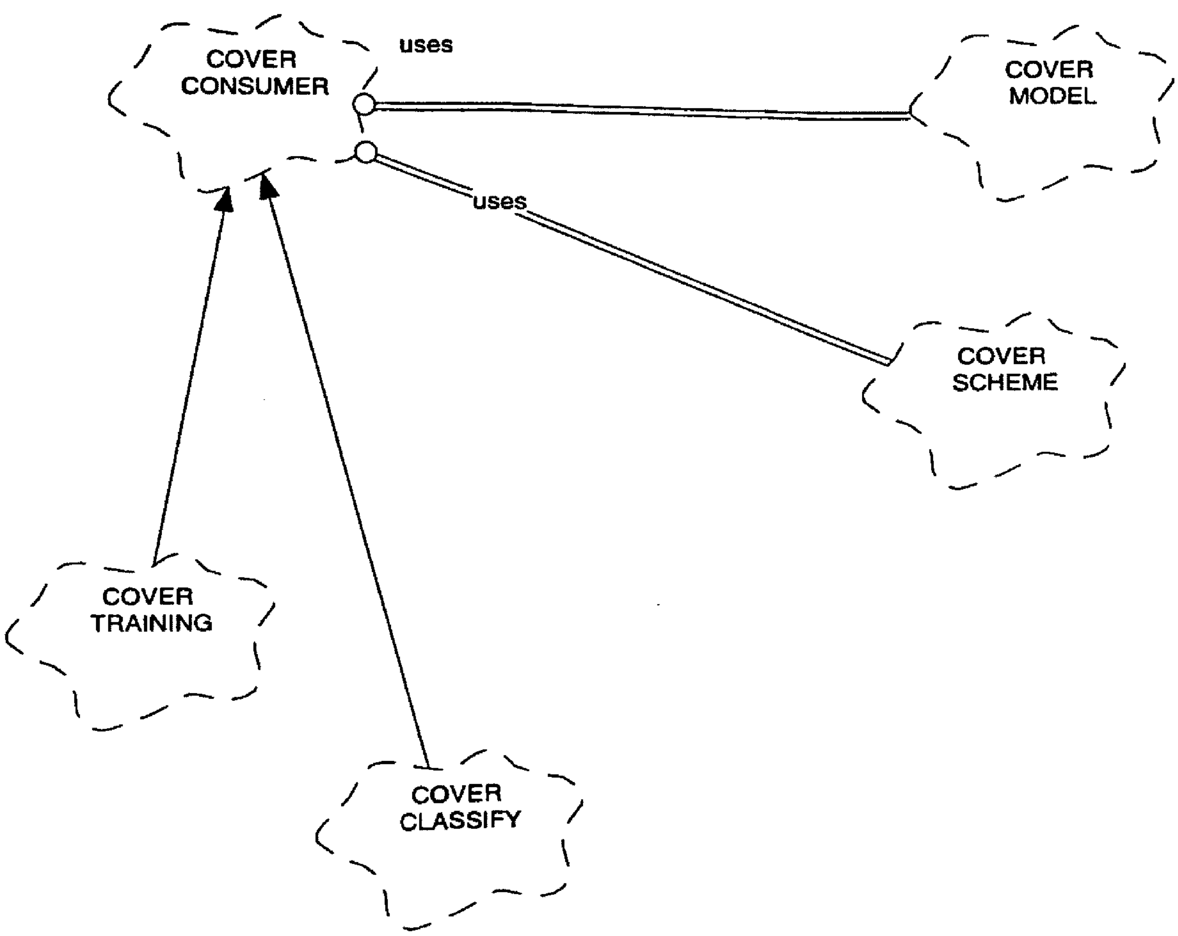

SPATIAL CONSUMER
uses
SPATIAL MODEL
SPATIAL PRD CONSUMER
RESAMPLE WINDOW
uses
RESMAPLE
PROJECTION TRANSFORM
uses
PROJECTION PARAMETERS

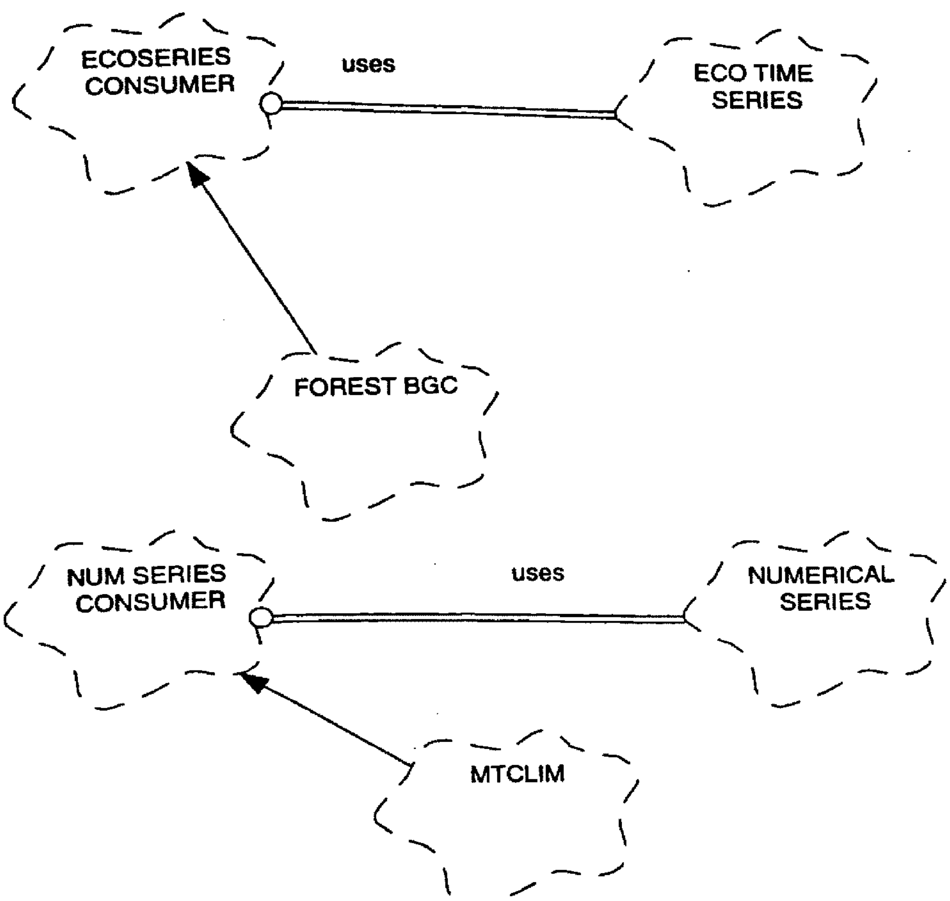

OVERLAY CONSUMER
uses
OVERLAY
ECOMODEL ASSIGN
REGISTER
uses
uses
GCP SET
ECOMODEL

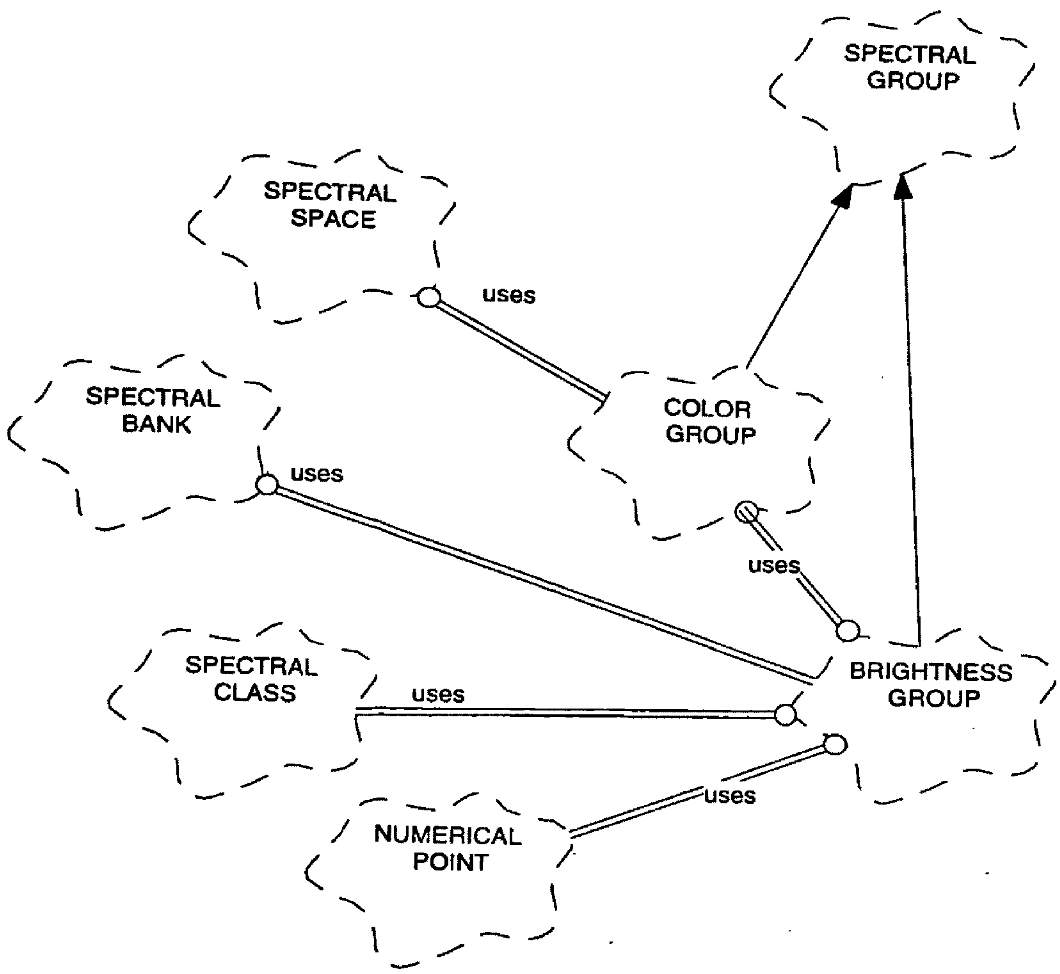

CATEGORICAL
CONSUMER
uses
CATEGORICAL
OVERLAY
REPARTITION
SCAN
WINDOW
SIMILARITY
MATRIX
uses
uses
WINDOW
REPARTITION
MERGE

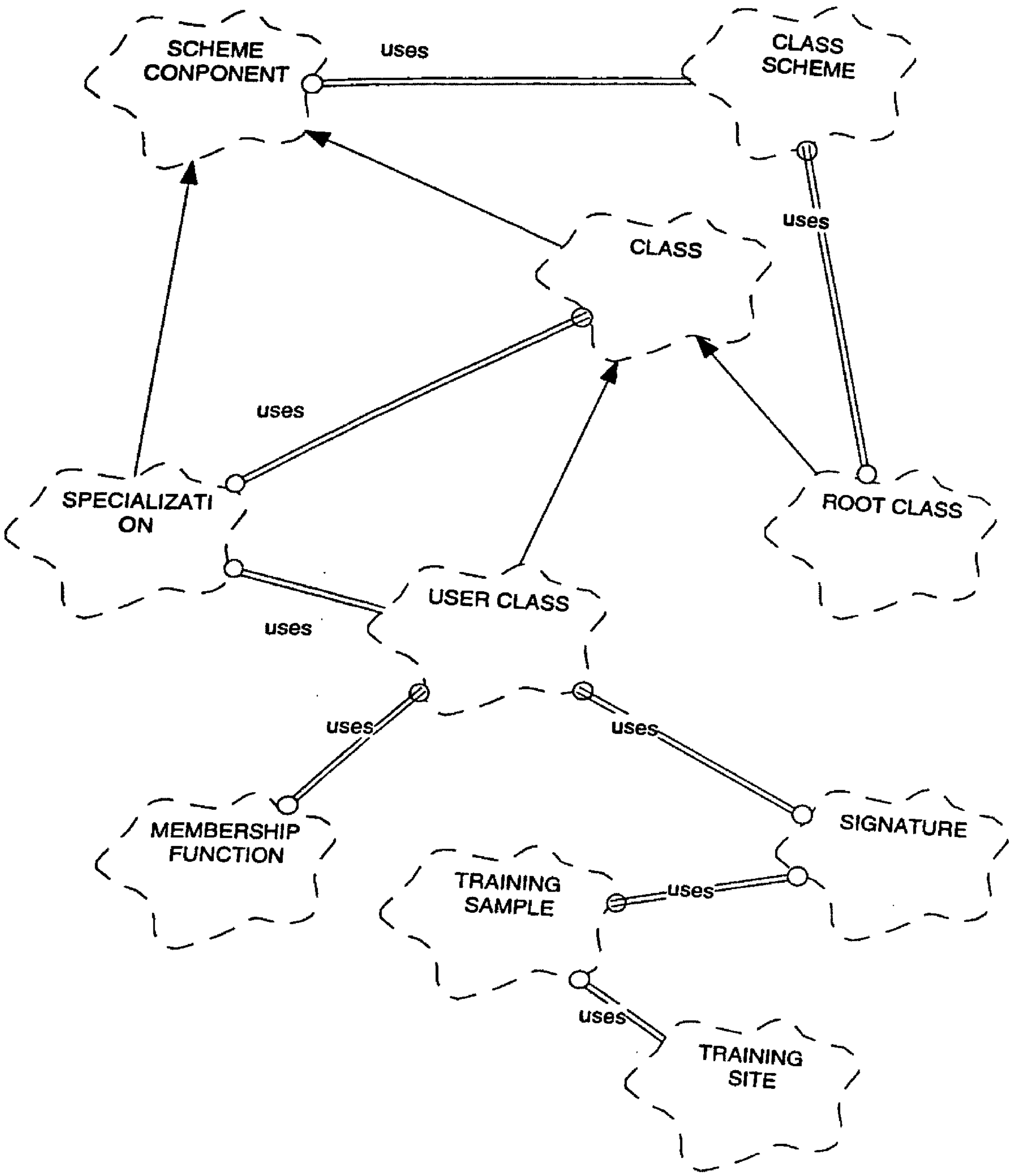

NUMERICAL CONSUMER
NUMERICAL OVERLAY
NUM PROD CONSUMER
SPECTRAL CLASSIFY
PARTITIONER
CLUSTER ANALYSIS
TOPO PARTITION
uses
uses
uses
uses
uses
CATEGORICAL OVERLAY
SPECTRAL SCHEME
SPECTRAL BANK

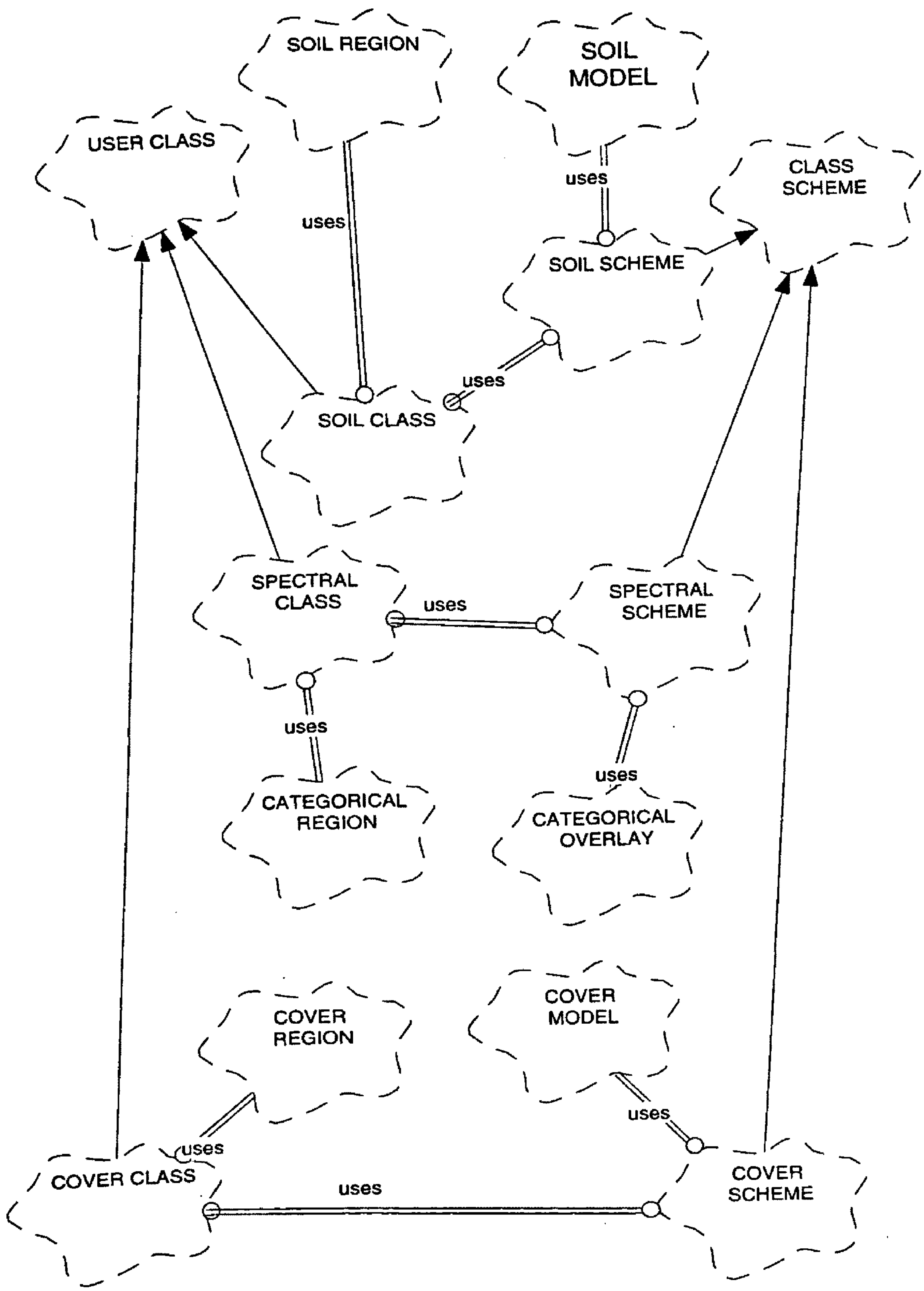

NUM PROD CONSUMER
CONVOLVE
uses
GENERIC NUMERICAL
OVERLAY ALGEBRA
AVERAGE WINDOW
INTERPOLATE
uses
INTERPOLATE WINDOW

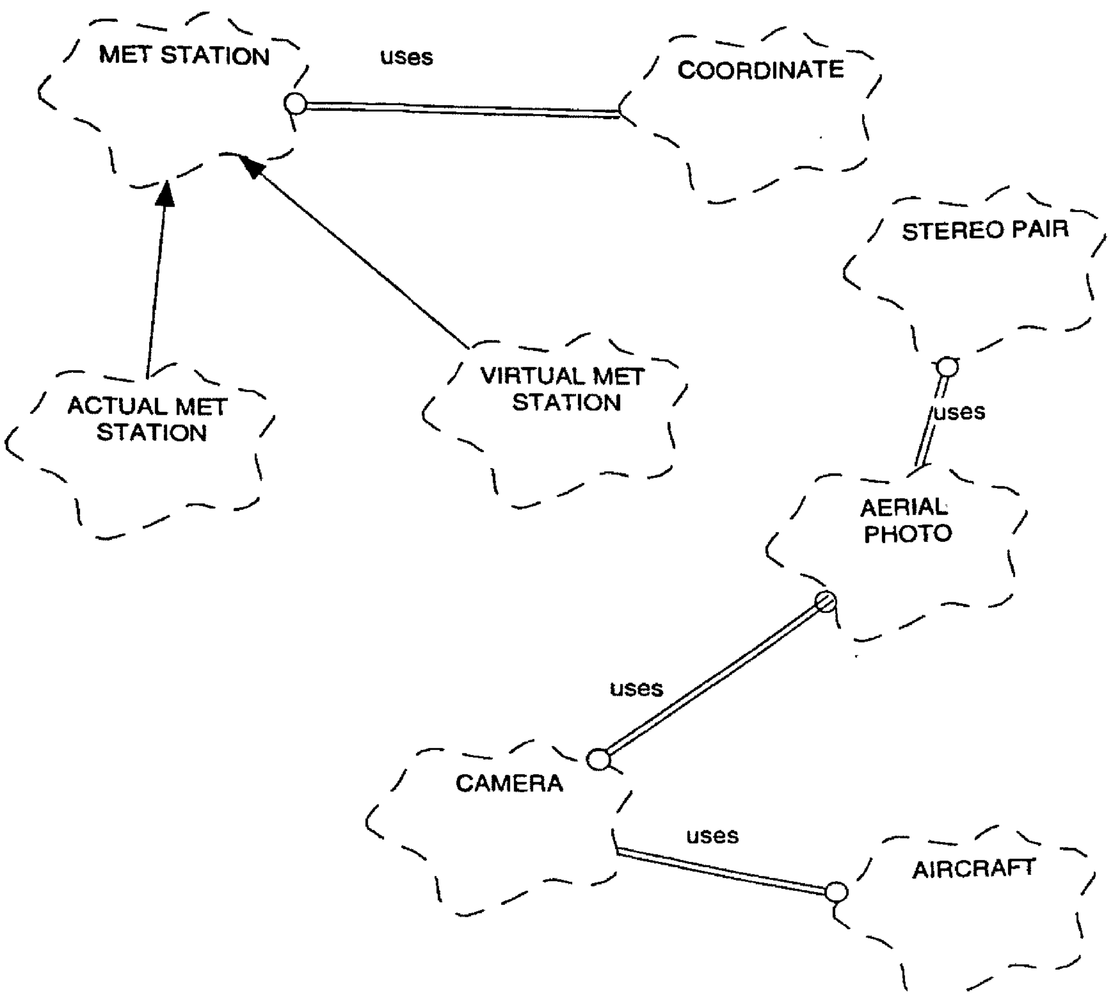

COVER
CONSUMER
uses
COVER
MODEL
uses
COVER
SCHEME
COVER
TRAINING
COVER
CLASSIFY

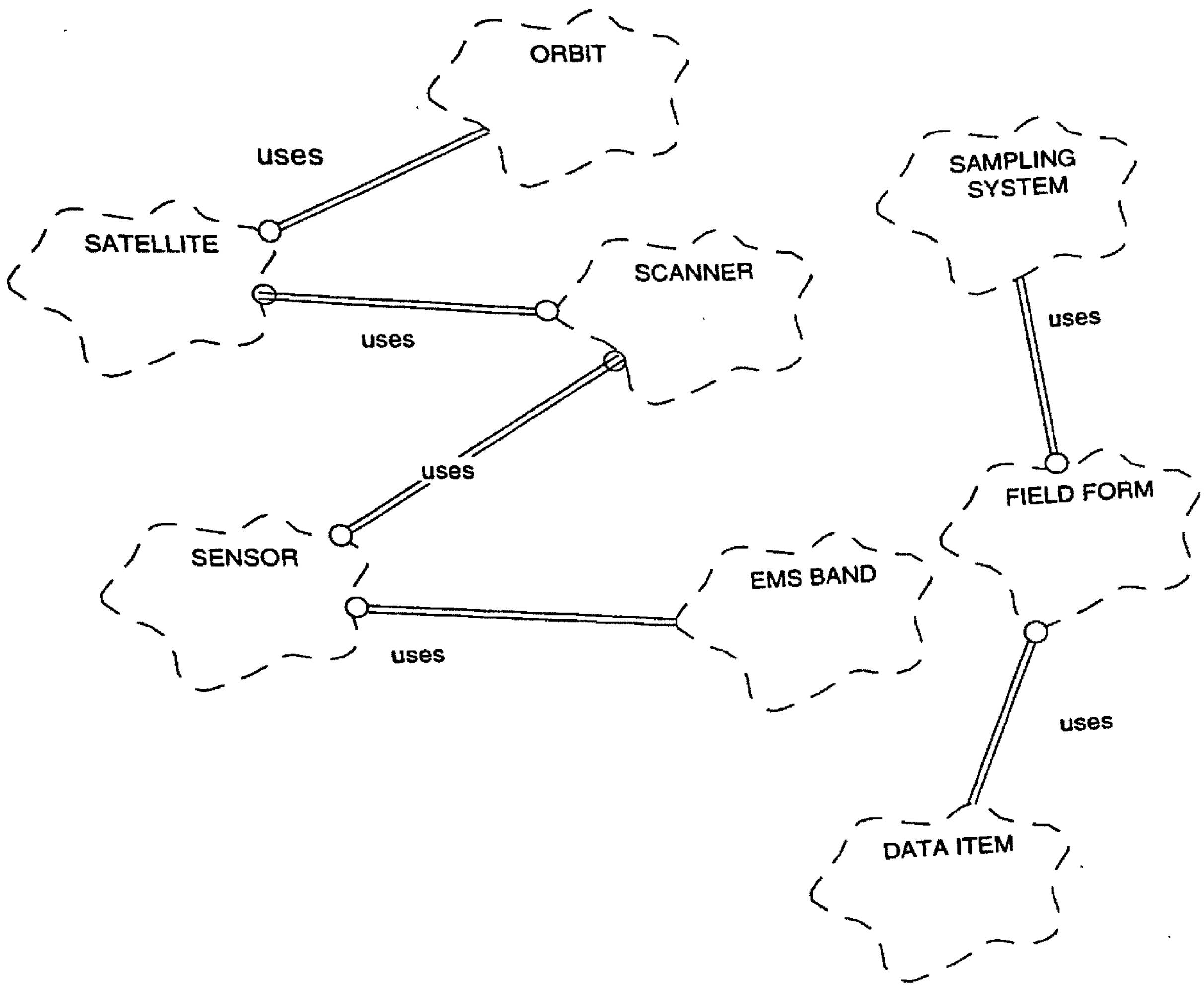

ECOSERIES CONSUMER
uses
ECO TIME SERIES
FOREST BGC
NUM SERIES CONSUMER
uses
NUMERICAL SERIES
MTCLIM

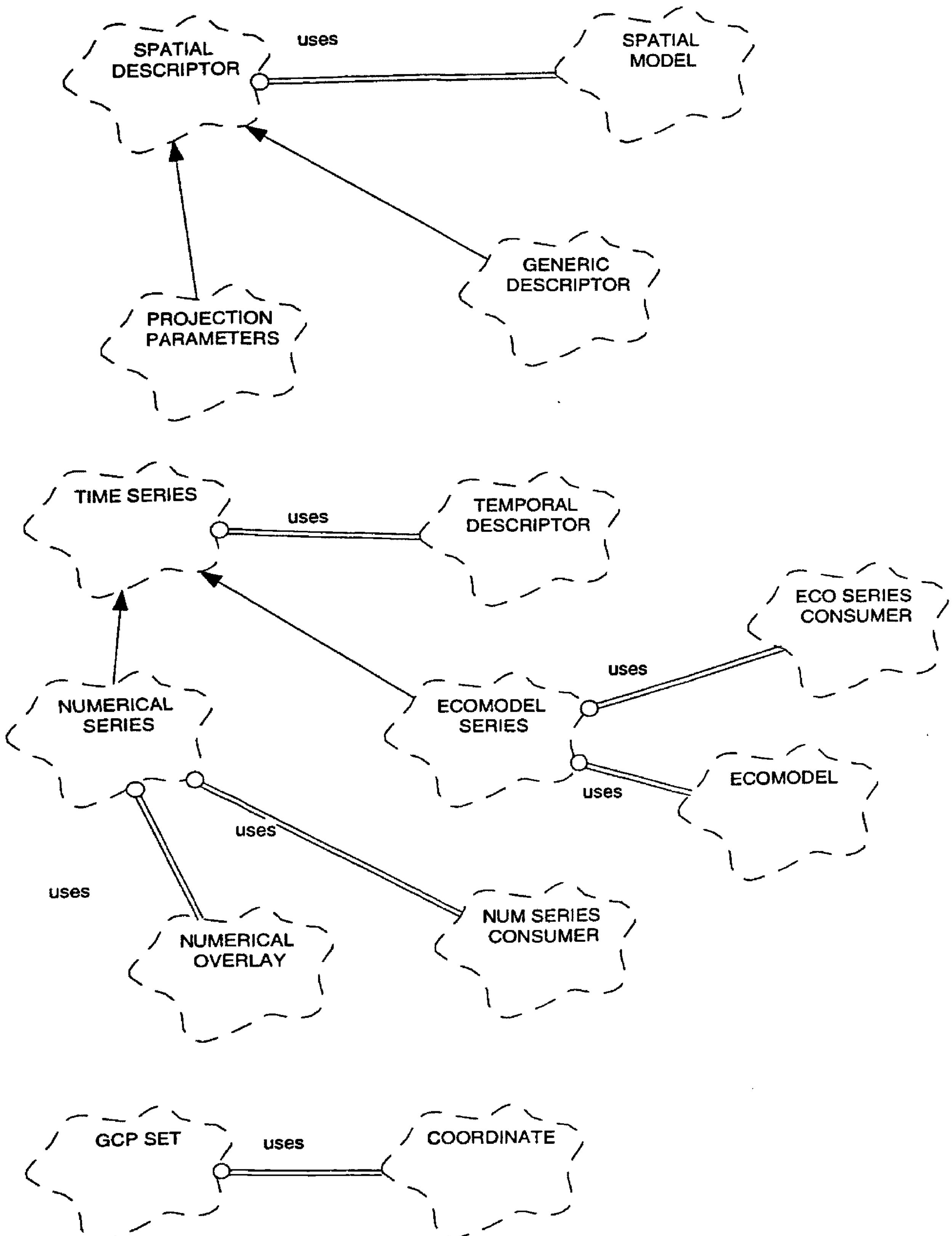

SPECTRAL GROUP
SPECTRAL SPACE
uses
SPECTRAL BANK
uses
COLOR GROUP
uses
SPECTRAL CLASS
uses
BRIGHTNESS GROUP
uses
NUMERICAL POINT

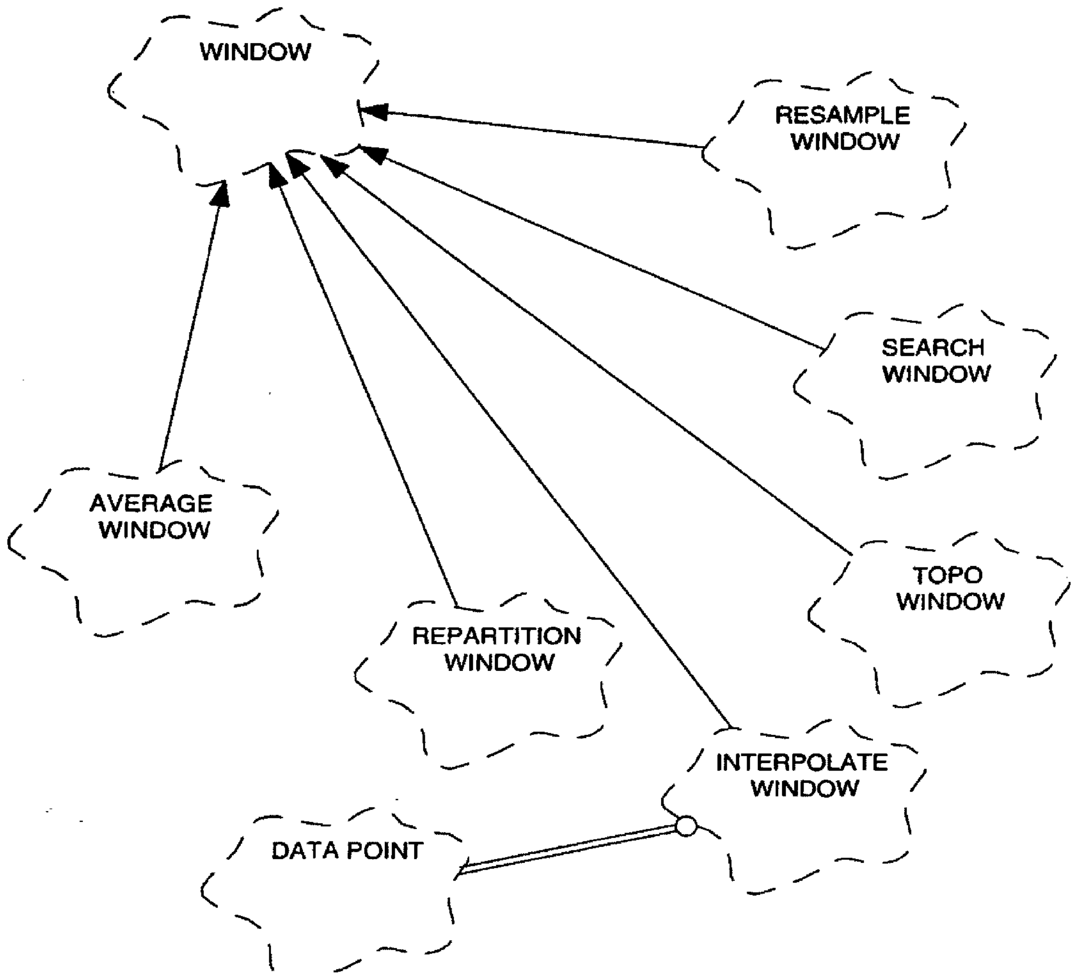

SCHEME
CONPONENT
uses
CLASS
SCHEME
uses
CLASS
uses
SPECIALIZATI
ON
ROOT CLASS
USER CLASS
uses
uses
uses
MEMBERSHIP
FUNCTION
SIGNATURE
TRAINING
SAMPLE
uses
uses
TRAINING
SITE

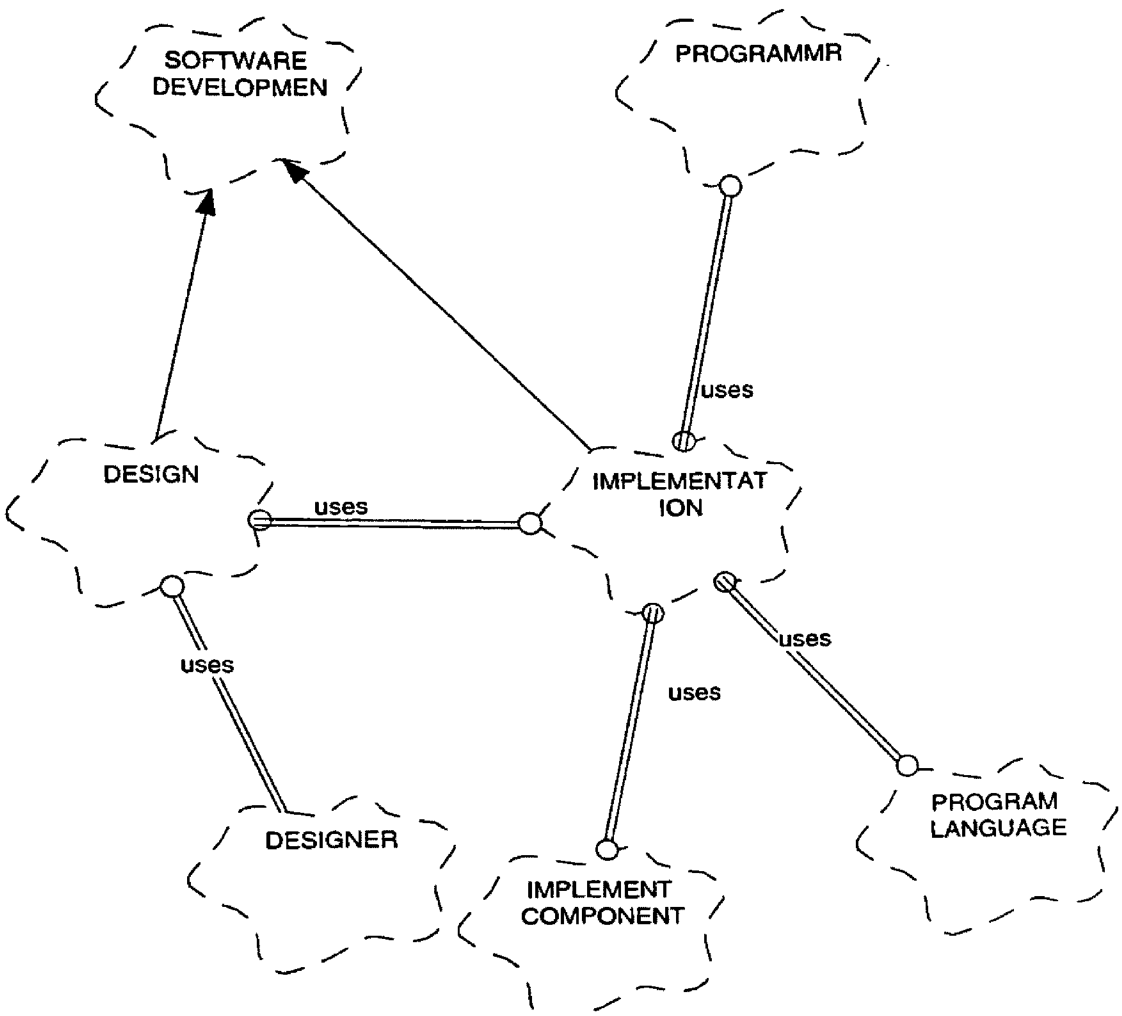

SOIL REGION
SOIL MODEL
USER CLASS
uses
uses
CLASS SCHEME
SOIL SCHEME
uses
SOIL CLASS
SPECTRAL CLASS
uses
SPECTRAL SCHEME
uses
uses
CATEGORICAL REGION
CATEGORICAL OVERLAY
COVER REGION
COVER MODEL
uses
uses
COVER CLASS
uses
COVER SCHEME

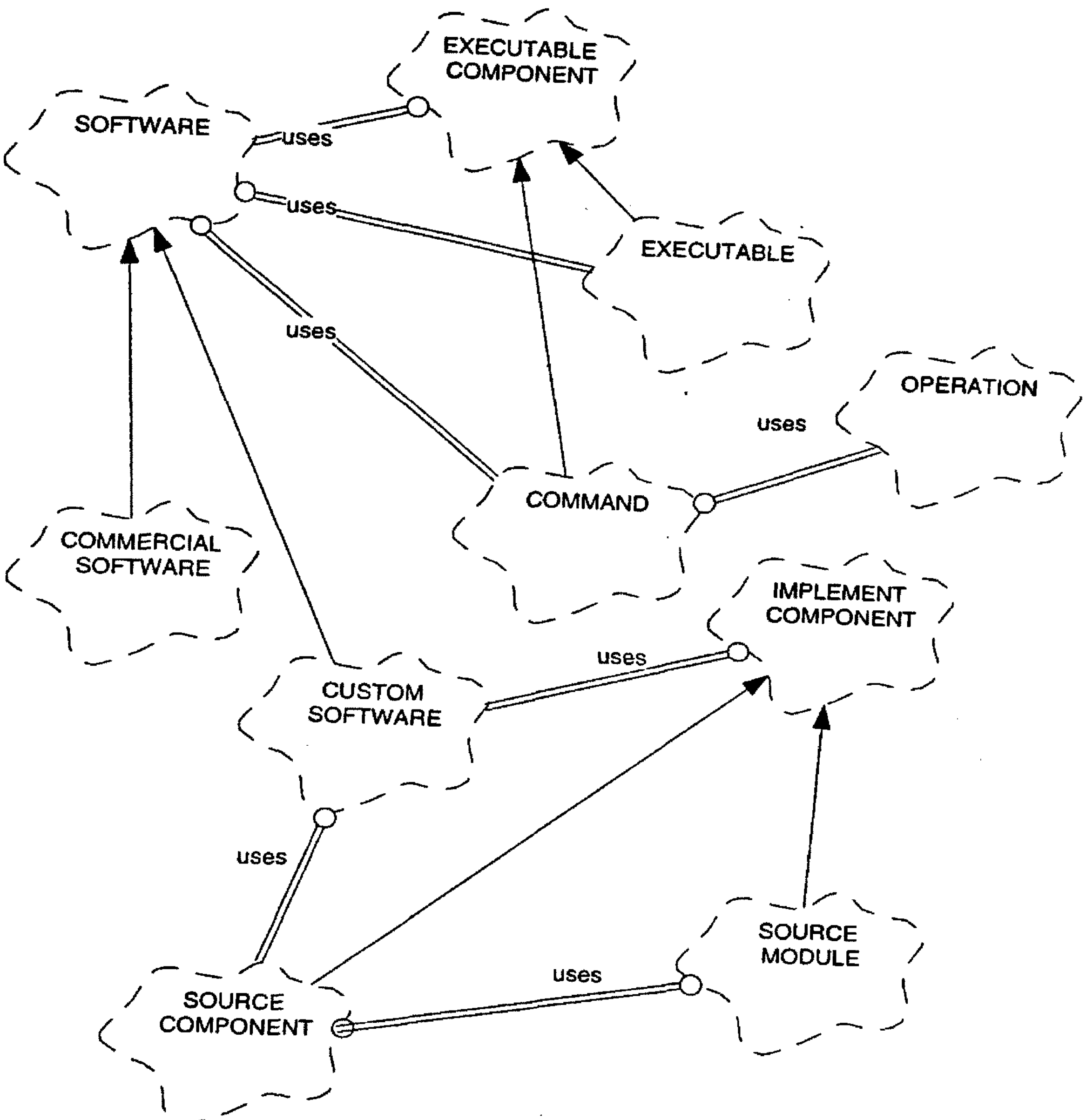

MET STATION
uses
COORDINATE
STEREO PAIR
ACTUAL MET STATION
VIRTUAL MET STATION
uses
AERIAL PHOTO
uses
CAMERA
uses
AIRCRAFT

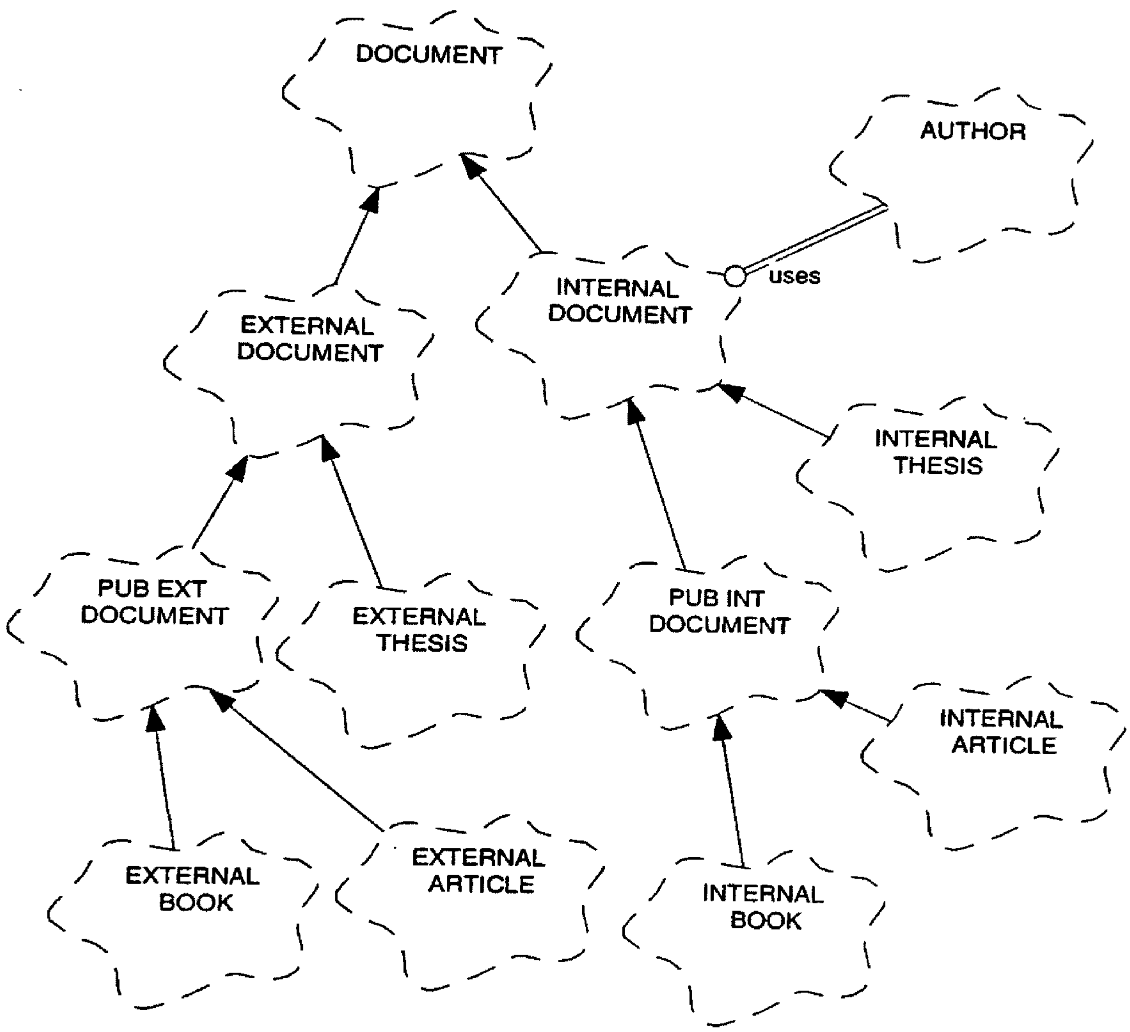

ORBIT
uses
SATELLITE
SCANNER
uses
uses
SENSOR
EMS BAND
uses
SAMPLING SYSTEM
uses
FIELD FORM
uses
DATA ITEM

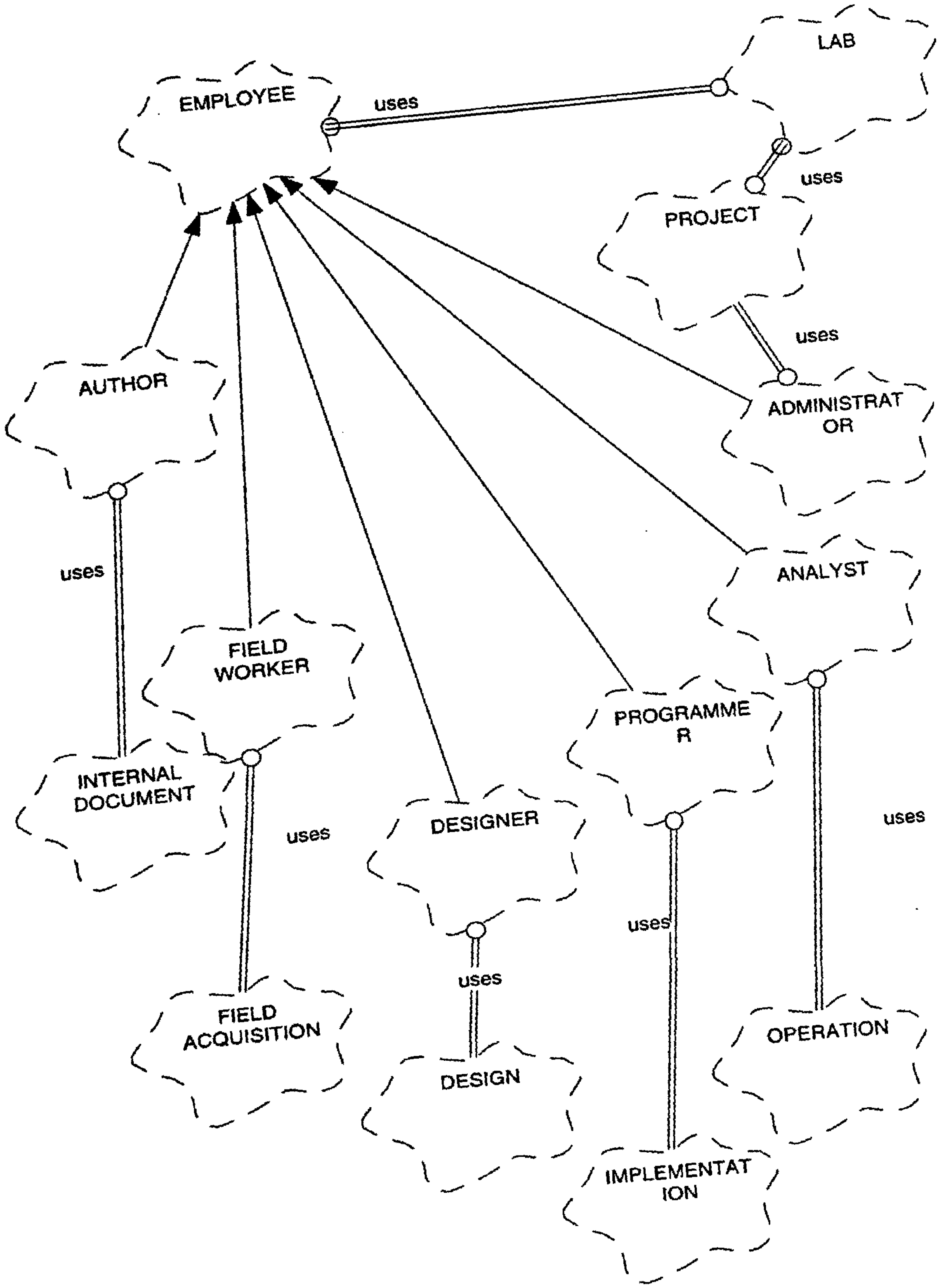

SPATIAL DESCRIPTOR
uses
SPATIAL MODEL
GENERIC DESCRIPTOR
PROJECTION PARAMETERS
TIME SERIES
uses
TEMPORAL DESCRIPTOR
ECO SERIES CONSUMER
uses
NUMERICAL SERIES
ECOMODEL SERIES
ECOMODEL
uses
uses
uses
NUM SERIES CONSUMER
NUMERICAL OVERLAY
GCP SET
uses
COORDINATE


WINDOW
RESAMPLE WINDOW
SEARCH WINDOW
AVERAGE WINDOW
TOPO WINDOW
REPARTITION WINDOW
INTERPOLATE WINDOW
DATA POINT


SOFTWARE
DEVELOPMEN
PROGRAMMR
uses
DESIGN
uses
IMPLEMENTAT
ION
uses
uses
uses
DESIGNER
IMPLEMENT
COMPONENT
PROGRAM
LANGUAGE


EXECUTABLE COMPONENT
SOFTWARE
uses
uses
EXECUTABLE
uses
OPERATION
uses
COMMAND
COMMERCIAL SOFTWARE
IMPLEMENT COMPONENT
uses
CUSTOM SOFTWARE
uses
SOURCE MODULE
uses
SOURCE COMPONENT


DOCUMENT
AUTHOR
uses
INTERNAL DOCUMENT
EXTERNAL DOCUMENT
INTERNAL THESIS
PUB EXT DOCUMENT
EXTERNAL THESIS
PUB INT DOCUMENT
INTERNAL ARTICLE
EXTERNAL BOOK
EXTERNAL ARTICLE
INTERNAL BOOK


EMPLOYEE
uses
LAB
uses
PROJECT
uses
AUTHOR
ADMINISTRAT
OR
uses
ANALYST
FIELD
WORKER
PROGRAMME
R
INTERNAL
DOCUMENT
uses
DESIGNER
uses
uses
uses
FIELD
ACQUISITION
OPERATION
DESIGN
IMPLEMENTAT
ION

APPENDIX 2

CLASS SPECIFICATIONS

TABLE OF CONTENTS

2a. *Root Classes*

```
class   ENTITY
{

        /* attributes */
        /* none */

        /* operations */

        Create                                  (void) ;
        Destroy                                 (void) ;

} ;

class   NAMED_ENTITY : public       ENTITY
{

        /* attributes */
        STRING                                  name ;
        STRING                                  description ;

        /* operations */

        STRING                                  Get_name (void) ;
        void                                    Set_name (STRING) ;
        STRING                                  Get_description (void) ;
        void                                    Set_description (STRING) ;

} ;

class   DOCUMENTED_ENTITY       : public        NAMED_ENTITY
{

        /* attributes */
        SET < DOCUMENT >                        documents ;

        /* operations */

        DOCUMENT                                Iterate_over_documents (void) ;

} ;
```

2b. *Spatial Classes*

```
class   SPATIAL_ENTITY : public              DOCUMENTED_ENTITY
{
        /* attributes */
        NUMBER                               area ;
        NUMBER                               perimeter ;

        /* operations */
        NUMBER                               Get_area (void) ;
        void                                 Set_area (NUMBER) ;
        NUMBER                               Get_perimeter (void) ;
        void                                 Set_perimeter (NUMBER) ;
} ;

class   SPATIAL_POINT_ENTITY      : public   SPATIAL_ENTITY
{
        /* attributes */
        NUMBER                               number_points ;

        /* operations */
        NUMBER                               Get_number_points (void) ;
        void                                 Set_number_points (NUMBER) ;
} ;
```

```
class  SPATIAL_MODEL                    : public        SPATIAL_POINT_SET
{
        /* attributes */
        STRING                                          spatial_scale ;
        STRING                                          geog_coord_system ;
        STRING                                          point_distribution ;
        PROJECTION_SYSTEM                               projection_system ;
        PROJECTION_PARAMETERS                           projection_parameters ;
        SET < SPATIAL_MODEL_CONSUMER >                  consuming_operations ;
        OPERATION                                       producing_operation ;
        SET < SPATIAL_DESCRIPTOR >                      spatial_descriptors ;

        /* operations */
        STRING                                          Get_spatial_scale (void) ;
        void                                            Set_spatial_scale (STRING) ;
        STRING                                          Get_geog_coord_system (void) ;
        void                                            Set_geog_coord_system (STRING) ;
        STRING                                          Get_point_distribution (void) ;
        void                                            Set_point_distribution (STRING) ;
        PROJECTION_SYSTEM                               Get_projection_system (void) ;
        void                                            Set_projection_system
                                                            (PROJECTION_SYSTEM) ;
        PROJECTION_PARAMETERS                           Get_projection_parameters (void) ;
        void                                            Set_projection_parameters
                                                            (PROJECTION_PARAMETERS) ;
        SPATIAL_MODEL_CONSUMER                          Iterate_over_consuming_operations (void) ;
        OPERATION                                       Get_producing_operation (void) ;
        void                                            Set_producing_operation (OPERATION) ;
        SPATIAL_DESCRIPTOR                              Iterate_over_spatial_descriptors
} ;
```

```
class   POINT_MODEL : public        SPATIAL_MODEL
{
        /* attributes */
        SET < MODEL_POINT >                 model_points ;
        NUMBER                              number_numerical_overlays ;
        NUMBER                              number_categorical_overlays ;
        SET < OVERLAY >                     overlays ;

        /* operations */
        MODEL_POINT                         Iterate_over_model_points (void) ;
        NUMBER                              Get_number_numerical_overlays (void) ;
        void                                Set_number_numerical_overlays
                                                (NUMBER) ;
        NUMBER                              Get_number_categorical_overlays (void) ;
        void                                Set_number_categorical_overlays
                                                (NUMBER) ;
        OVERLAY                             Iterate_over_overlays (void) ;
        MODEL_POINT                         Get_point (COORDINATE) ;
        void                                Set_point (MODEL_POINT) ;
        MODEL_POINT                         Sample_points (void0 ;
} ;

class   OVERLAY     : public        SPATIAL_MODEL
{
        /* attributes */
        NUMBER                              number_regions ;
        NUMBER                              number_subregions ;
        NUMBER                              number_points ;
        POINT_MODEL                         contained_in ;
        SET < OVERLAY_CONSUMER >            consuming_operations ;

        /* operations */
        NUMBER                              Get_number_regions (void) ;
        void                                Set_number_regions (NUMBER) ;
        NUMBER                              Get_number_subregions (void) ;
        void                                Set_number_subregions (NUMBER) ;
        NUMBER                              Get_number_points (void) ;
        void                                Set_number_points (NUMBER) ;
        POINT_MODEL                         Get_point_model (void) ;
        void                                Set_point_model (POINT_MODEL) ;
} ;
```

```
class   NUMERICAL_OVERLAY        : public        OVERLAY
{
        /* attributes */
        SET < NUMERICAL_REGION >                 regions ;
        SET < NUMERICAL_SUBREGION >              subregions ;
        SET < NUMERICAL_POINT >                  points ;
        NUMERICAL_ATTRIBUTE                      describing_attribute ;
        SET < NUMERICAL_CONSUMER >               consuming_operations ;

        /* operations */
        NUMERICAL_REGION                         Iterate_over_regions (void) ;
        NUMERICAL_SUBREGION                      Iterate_over_subregions (void) ;
        NUMERICAL_POINT                          Iterate_over_points (void) ;
        NUMERICAL_ATTRIBUTE                      Get_describing_attribute (void) ;
        void                                     Set_describing_attribute
                                                     (NUMERICAL_ATTRIBUTE) ;
        NUMERICAL_CONSUMER                       Iterate_over_consuming_operations (void) ;
        NUMERICAL_OVERLAY                        Add (NUMERICAL_OVERLAY) ;
        NUMERICAL_OVERLAY                        Subtract (NUMERICAL_OVERLAY) ;
        NUMERICAL_OVERLAY                        Multiply (NUMERICAL_OVERLAY) ;
        NUMERICAL_OVERLAY                        Divide (NUMERICAL_OVERLAY) ;
        NUMERICAL_POINT                          Get_point (COORDINATE) ;
        void                                     Set_point  (NUMERICAL_POINT) ;
        NUMERICAL_POINT                          Sample_points (void) ;
} ;
```

```
class   CATEGORICAL_OVERLAY        : public     OVERLAY
{
        /* attributes */
        SET < CATEGORICAL_REGION >              regions ;
        SET < CATEGORICAL_SUBREGION >           subregions ;
        SET < CATEGORICAL_POINT >               points ;
        CATEGORICAL_ATTRIBUTE                   describing_attribute ;
        SET < CATEGORICAL_CONSUMER >            consuming_operations ;

        /* operations */
        CATEGORICAL_REGION                      Iterate_over_regions (void) ;
        CATEGORICAL_SUBREGION                   Iterate_over_subregions (void) ;
        CATEGORICAL_POINT                       Iterate_over_points (void) ;
        CATEGORICAL_ATTRIBUTE                   Get_describing_attribute (void) ;
        void                                    Set_describing_attribute
                                                    (CATEGORICAL_ATTRIBUTE) ;
        CATEGORICAL_CONSUMER                    Iterate_over_consuming_operations (void) ;
        CATEGORICAL_POINT                       Get_point (COORDINATE) ;
        void                                    Set_point (CATEGORICAL_POINT) ;
        CATEGORICAL_POINT                       Sample_points (void) ;

} ;

class   OVERLAY_REGION             : public     SPATIAL_POINT_ENTITY
{
        /* attributes */
        NUMBER                                  number_subregions ;

        /* operations */
        NUMBER                                  Get_number_subregions (void) ;
        void                                    Set_number_subregions (NUMBER) ;
} ;
```

```
class   CATEGORICAL_REGION        : public        OVERLAY_REGION
{
        /* attributes */
        CATEGORICAL_OVERLAY                       containing_overlay ;
        SET < CATEGORICAL_SUBREGION >             subregions ;
        SET < CATEGORICAL_POINT >                 points ;
        USER_CLASS                                class ;

        /* operations */
        CATEGORICAL_OVERLAY                       Get_containing_overlay (void) ;
        void                                      Set_containing_overlay
                                                      (CATEGORICAL_OVERLAY) ;
        CATEGORICAL_SUBREGION                     Iterate_over_subregions (void) ;
        CATEGORICAL_POINT                         Iterate_over_points (void) ;
        USER_CLASS                                Get_class (void) ;
        void                                      Set_class (USER_CLASS) ;
        void                                      Set_subregion
                                                      (CATEGORICAL_SUBREGION) ;
} ;

class   NUMERICAL_REGION          : public        OVERLAY_REGION
{
        /* attributes */
        NUMERICAL_OVERLAY                         containing_overlay ;
        SET < NUMERICAL_SUBREGION >               subregions ;
        SET <NUMERICAL_POINTS >                   points ;

        /* operations */
        NUMERICAL_OVERLAY                         Get_containing_overlay (void) ;
        void                                      Set_containing_overlay
                                                      (NUMERICAL_OVERLAY) ;
        NUMERICAL_SUBREGION                       Iterate_over_subregions (void) ;
        NUMERICAL_POINT                           Iterate_over_points (void) ;
} ;
```

```
class   OVERLAY_SUBREGION        : public     SPATIAL_POINT_ENTITY
{
        /* attributes */
        NUMBER                                number_neighbors ;

        /* operations
        NUMBER                                Get_number_neighbors (void) ;
        void                                  Set_number_neighbors (NUMBER) ;
} ;

class   CATEGORICAL_SUBREGION : public        OVERLAY_SUBREGION
{
        /* attributes */
        CATEGORICAL_REGION                    containing_region ;
        SET < CATEGORICAL_POINT >             points ;
        SET <CATEGORICAL_SUBREGION >          neighbors ;

        /* operations */
        CATEGORICAL_REGION                    Get_containing_region (void) ;
        void                                  Set_containing_region
                                                  (CATEGORICAL_REGION) ;
        CATEGORICAL_POINT                     Iterate_over_points (void) ;
        CATEGORICAL_SUBREGION                 Iterate_over_neighbors (void) ;
} ;

class   NUMERICAL_SUBREGION : public SPATIAL_POINT_SET
{
        /* attributes */
        NUMERICAL_REGION                      containing_region ;
        SET < NUMEGORICAL_POINT >             points ;
        SET < NUMERICAL_SUBREGION >           neighbors ;

        /* operations */
        NUMERICAL_REGION                      Get_containing_region (void) ;
        void                                  Set_containing_region
                                                  (NUMERICAL_REGION) ;
        NUMERICAL_POINT                       Iterate_over_points (void) ;
        NUMERICAL_SUBREGION                   Iterate_over_neighbors (void) ;
} ;
```

```
class   POINT           : public        SPATIAL_ENTITY
{
        /* attributes */
        COORDINATE                              coord ;

        /* operations */
        COORDINATE                              Get_coord (void) ;
        void                                    Set_coord (COORDINATE) ;
} ;

class   DATA_POINT   public :           POINT
{

        /* attributes */
        NUMBER                                  value ;

        /* operations */
        NUMBER                                  Get_value (void) ;
        void                                    Set_value (NUMBER) ;
} ;

class   NUMERICAL_POINT                 : public        DATA_POINT
{
        /* attributes */
        NUMERICAL_SUBREGION                     containing_subregion ;

        /* operations */
        NUMERICAL_SUBREGION                     Get_containing_subregion (void) ;
        void                                    Set_containing_subregion
                                                     (NUMERICAL_SUBREGION) ;
        NUMBER                                  Add (NUMERICAL_POINT) ;
        NUMBER                                  Subtract (NUMERICAL_POINT) ;
        NUMBER                                  Multiply (NUMERICAL_POINT) ;
        NUMBER                                  Divide (NUMERICAL_POINT) ;
} ;
```

```
class   CATEGORICAL_POINT      : public    VALUE_POINT
{
        /* attributes */
        CATEGORICAL_SUBREGION              containing_subregion ;

        /* operations */
        CATEGORICAL_SUBREGION              Get_containing_subregion (void) ;
        void                               Set_containing_subregion
                                               (CATEGORICAL_SUBREGION) ;
} ;

class   MODEL_POINT            : public    POINT
{
        /* attributes */
        POINT_MODEL                        containing_point_model ;

        /* operations */
        POINT_MODEL                        Get_point_model (void) ;
        void                               Set_point_model (POINT_MODEL) ;
} ;
```

2c. *Ecosystem Classes*

```
class   ECOSYSTEM_ENTITY            : public        DOCUMENTED_ENTITY
{
        /* attributes */
        SET < ECOSYSTEM DESCRIPTOR >                ecosystem_descriptors ;

        /* operations */
        ECOSYSTEM_DESCRIPTOR                        Iterate_over_ecosystem_descriptors (void) ;
} ;

class   ECOMODEL      : public        ECOSYSTEM_ENTITY
{
        /* attributes */
        NUMBER                                      number_landunits ;
        CATEGORICAL_OVERLAY                         corresponding_overlay ;
        OPERATION                                   producer ;
        SET  < ECOMODEL_CONSUMER >                  ecomodel_consumers ;

        /* operations */
        NUMBER                                      Get_number_landunits (void) ;
        void                                        Set_number_landunits (NUMBER) ;
        CATEGORICAL_OVERLAY                         Get_correspondin_overlay (void) ;
        void                                        Set_corresponding_overlay
                                                        (CATEGORICAL_OVERLAY) ;
        OPERATION                                   Get_producer (void) ;
        void                                        Set_producer (OPERATION) ;
        ECOSYSTEM_MODEL_CONSUMER                    Iterate_over_ecosystem_model_consumers
                                                        (void) ;
} ;

class   REGIONAL_MODEL              : public        ECOMODEL
{
        /* attributes */
        NUMBER                                      number_regions ;

        /* operations */
        NUMBER                                      Get_number_regions (void) ;
        void                                        Set_number_regions (NUMBER) ;
} ;
```

```
class   SOIL_MODEL : public          REGIONAL_MODEL
{
        /* attributes */
        SET < SOIL_REGION >                  soil_regions ;
        SOIL_CLASS_SCHEME                    describing_class_scheme ;
        SET < SOIL_MODEL_CONSUMER >          soil_model_consumers ;

        /* operations */
        SOIL_REGION                          Iterate_over_soil_regions (void) ;
        SOIL_CLASS_SCHEME                    Get_soil_class_scheme (void) ;
        void                                 Set_soil_class_scheme
                                                 (SOIL_CLASS_SCHEME) ;
        SOIL_MODEL_CONSUMER                  Iterate_over_soil_model_consumers (void) ;
} ;

class   COVER_MODEL : public         REGIONAL_MODEL
{
        /* attributes */
        SET < COVER_REGION >                 cover_regions ;
        COVER_CLASS_SCHEME                   describing_class_scheme  ;
        SET < COVER_MODEL_CONSUMER >         cover_model_consumers ;
        SET < COVER_SUBREGION >              training_sites ;
        SET < COVER_SUBREGION >              cover_subregions ;

        /* operations */
        COVER_REGION                         Iterate_over_cover_regions (void) ;
        COVER_CLASS_SCHEME                   Get_cover_class_scheme (void) ;
        void                                 Set_cover_class_scheme
                                                 (COVER_CLASS_SCHEME) ;
        COVER_MODEL_CONSUMER                 Iterate_cover_model_consumers (void) ;
        COVER_SUBREGION                      Iterate_over_training_sites (void) ;
        COVER_SUBREGION                      Iterate_over_cover_subregions (void) ;
} ;
```

```
class   TOPO_MODEL : public        ECOMODEL
{
        /* attributes */
        NUMBER                              number_watersheds ;
        SET < WATERSHED >                   watersheds ;

        /* operations */
        NUMBER                              Get_number_watersheds (void) ;
        void                                Set_number_watersheds (NUMBER) ;
        WATERSHED                           Iterate_over_watersheds (void) ;
} ;

class   ECOSYSTEM_REGION        : public    ECOSYSTEM_ENTITY
{
        /* attributes */
        NUMBER                              number_subregions ;

        /* operations */
        NUMBER                              Get_number_subregions (void) ;
        void                                Set_number_subregions (NUMBER) ;
} ;

class   COVER_REGION            : public    ECOSYSTEM_REGION
{
        /* attributes */
        COVER_MODEL                         containing_model ;
        SET < COVER_SUBREGION >             cover_subregions ;
        COVER_CLASS                         describing_class ;

        /* operations */
        COVER_MODEL                         Get_containing_model (void) ;
        void                                Set_containing_model (COVER_MODEL) ;
        COVER_SUBREGION                     Iterate_over_cover_subregions (void) ;
        COVER_CLASS                         Get_cover_class (void) ;
        void                                Set_cover_class (COVER_CLASS) ;
        void                                Set_cover_subregion
                                                (COVER_SUBREGION) ;
} ;
```

```
class   SOIL_REGION  : public       ECOSYSTEM_REGION
{
        /* attributes */
        SOIL_MODEL                                  containing_model ;
        SET < SOIL_SUBREGION >                      soil_subregions ;
        SOIL_CLASS                                  describing_class ;

        /* operations */
        SOIL_MODEL                                  Get_containing_model (void) ;
        void                                        Set_containing_model (SOIL_MODEL) ;
        SOIL_SUBREGION                              Iterate_over_soil_subregions (void) ;
        SOIL_CLASS                                  Get_soil_class (void) ;
        void                                        Set_soil_class (SOIL_CLASS) ;
} ;

class   ECOSYSTEM_LANDUNIT          : public        ECOSYSTEM_ENTITY
{
        /* attributes */
        CATEGORICAL_SUBREGION                       categorical_subregion ;
        VIRTUAL_MET_STATION                         virtual_climate ;

        /* operations */
        CATEGORICAL_SUBREGION                       Get_categorical_subregion (void) ;
        void                                        Set_categorical_subregion
                                                         (CATEGORICAL_SUBREGION) ;
        VIRTUAL_MET_STATION                         Get_virtual_met_station (void) ;
        void                                        Set_virtual_climate
                                                         (VIRTUAL_MET_STATION) ;
} ;

class   COVER_SUBREGION             : public        ECOSYSTEM_LANDUNIT
{
        /* attributes */
        COVER_REGION                                containing_region ;

        /* operations */
        COVER_REGION                                Get_region (void) ;
        void                                        Set_region  (COVER_REGION) ;
} ;
```

```
class   SOIL_SUBREGION          : public        ECOSYSTEM_LANDUNIT
{
        /* attributes */
        SOIL_REGION                             containing_region ;

        /* operations */
        SOIL_REGION                             Get_containing_region (void) ;
        void                                    Set_containing_region (SOIL_REGION) ;

} ;

class   WATERSHED   : public        ECOSYSTEM_ENTITY
{
        /* attributes */
        TOPO_MODEL                              containing_model ;
        SET < CATCHMENT >                       catchments ;
        STREAM_NETWORK                          stream_network ;

        /* operations */
        TOPO_MODEL                              Get_containing_model (void) ;
        void                                    Set_containing_model (TOPO_MODEL) ;
        CATCHMENT                               Iterate_over_catchments (void) ;
        STREAM_NETWORK                          Get_stream_network (void) ;
        void                                    Set_stream_network
                                                    (STREAM_NETWORK) ;

} ;
```

```
class   CATCHMENT    : public      ECOSYSTEM_ENTITY
{
        /* attributes */
        WATERSHED                          containing_watershed ;
        HILLSLOPE                          left_hillslope ;
        HILLSLOPE                          right_hillslope ;
        STREAM_LINK                        stream_link ;

        /* operations */
        WATERSHED                          Get_containing_watershed (void) ;
        void                               Set_containing_watershed
                                               (WATERSHED) ;
        HILLSLOPE                          Get_left_hillslope (void) ;
        void                               Set_left_hillslope (HILLSLOPE) ;
        HILLSLOPE                          Get_right_hillslope (void) ;
        void                               Set_right_hillslope (HILLSLOPE) ;
        STREAM_LINK                        Get_stream_link (void) ;
        void                               Set_stream_link (STREAM_LINK) ;

} ;

class   HILLSLOPE    : public      ECOSYSTEM_LANDUNIT
{
        /* attributes */
        CATCHMENT                          containing_catchment ;

        /* operations */
        CATCHMENT                          Get_containing_catchment (void) ;
        void                               Set_containing_catchment
                                               (CATCHMENT) ;

} ;
```

```
class   STREAM_ENTITY : public ECOSYSTEM_ENTITY
{
        /* attributes */
        NUMBER                                  length ;
        NUMBER                                  order ;

        /* operations */
        NUMBER                                  Get_length (void) ;
        void                                    Set_length (NUMBER) ;
        NUMBER                                  Get_order (void) ;
        void                                    Set_order (NUMBER) ;
} ;

class   STREAM_NETWORK          : public        STREAM_ENTITY
{
        /* attributes */
        WATERSHED                               containing_watershed ;
        OUTLET_POINT                            outlet_point ;
        SET < STREAM_JUNCTION >                 stream_junctions ;
        SET < STREAM_LINK >                     stream_links ;
        NUMBER                                  number_junctions ;
        NUMBER                                  number_links ;

        /* operations */
        WATERSHED                               Get_containing_watershed (void) ;
        void                                    Set_containing_watershed
                                                    (WATERSHED) ;

        OUTLET_POINT                            Get_outlet_point (void) ;
        void                                    Set_outlet_point (OUTLET_POINT) ;
        STREAM_JUNCTION                         Iterate_over_stream_junctions (void) ;
        STREAM_LINK                             Iterate_over_stream_links (void) ;
        NUMBER                                  Get_number_junctions (void) ;
        void                                    Set_number_junctions (NUMBER) ;
        NUMBER                                  Get_number_links (void) ;
        void                                    Set_number_links (NUMBER) ;
} ;
```

```
class   STREAM_LINK : public        STREAM_ENTITY
{
        /* attributes */
        STREAM_JUNCTION                     flows_to ;
        STREAM_JUNCTION                     flows_from ;
        CATCHMENT                           containing_catchment ;
        STREAM_NETWORK                      containing_network ;

        /* operations */
        STREAM_JUNCTION                     Get_flows_to (void) ;
        void                                Set_flows_to (STREAM_JUNCTION) ;
        STREAM_JUNCTION                     Get_flows_from (void) ;
        void                                Set_flows_from (STREAM_JUNCTION) ;
        CATCHMENT                           Get_catchment (void) ;
        void                                Set_catchmetn (CATCHMENT) ;
        STREAM_NETWORK                      Get_containing_network (void) ;
        void                                Set_containing_network
                                                (STREAM_NETWORK) ;
} ;

class   ECOSYSTEM_POINT         : public    ECOSYSTEM_ENTITY
{
        /* attributes */
        CATEGORICAL_POINT                   categorical_point ;

        /* operations */
        CATEGORICAL_POINT                   Get_categorical_point (void) ;
        void                                Set_categorical_point
                                                (CATEGORICAL_POINT) ;
};
```

```
class   STREAM_JUNCTION         : public    ECOSYSTEM_POINT
{
        /* attributes */
        STREAM_NETWORK                      containing_stream_network ;
        NUMBER                              number_inflows ;
        SET < STREAM_LINK >                 inflows ;
        STREAM_LINK                         outflow ;

        /* operations */
        STREAM_NETWORK                      Get_containing_stream_network (void) ;
        void                                Set_containing_stream_network
                                                (STREAM_NETWORK) ;
        NUMBER                              Get_number_inflows (void) ;
        void                                Set_number_inflows (NUMBER) ;
        STREAM_LINK                         Iterate_over_inflows (void) ;
        STREAM_LINK                         Get_outflow (void) ;
        void                                Set_outflow (STREAM_LINK) ;
} ;

class   OUTLET_POINT            : public    ECOSYSTEM_POINT
{
        /* attributes */
        STREAM_NETWORK                      containing_stream_network ;

        /* operations */
        STREAM_NETWORK                      Get_containing_stream_network (void) ;
        void                                Set_containing_stream_network
                                                (STREAM_NETWORK) ;

} ;
```

2d. *Ecosystem Descriptor Classes*

```
class   ECOSYSTEM_DESCRIPTOR    : public    DOCUMENTED_ENTITY
{
        /* attributes */
        ECOSYSTEM_ENTITY                    described_ecosystem_entity ;

        /* operations */
        ECOSYSTEM_ENTITY                    Get_described_ecosystem_entity (void) ;
        void                                Set_described_ecosystem_entity
                                                (ECOSYSTEM_ENTITY) ;
} ;

class   TM_SPECTRAL_DESCRIPTOR  : public    ECOSYSTEM_DESCRIPTOR
{
        /* attributes */
        NUMBER                              tm_1 ;
        NUMBER                              tm_2 ;
        NUMBER                              tm_3 ;
        NUMBER                              tm_4 ;
        NUMBER                              tm_5 ;
        NUMBER                              tm_6 ;
        NUMBER                              tm_7 ;

        /* operations */
        NUMBER                              Get_tm_1 (void) ;
        void                                Set_tm_1 (NUMBER) ;
        NUMBER                              Get_tm_2 (void) ;
        void                                Set_tm_2 (NUMBER) ;
        NUMBER                              Get_tm_3 (void) ;
        void                                Set_tm_3 (NUMBER) ;
        NUMBER                              Get_tm_4 (void) ;
        void                                Set_tm_4 (NUMBER) ;
        NUMBER                              Get_tm_5 (void) ;
        void                                Set_tm_5 (NUMBER) ;
        NUMBER                              Get_tm_6 (void) ;
        void                                Set_tm_6 (NUMBER) ;
        NUMBER                              Get_tm_7 (void) ;
        void                                Set_tm_7 (NUMBER) ;
} ;
```

```
class   SOIL_DESCRIPTOR          : public        ECOSYSTEM_DESCRIPTOR
{

        /* attributes */
        NUMBER                                   depth ;
        NUMBER                                   texture ;
        NUMBER                                   hydraulic_conductivity ;
        NUMBER                                   transmissivity ;
        NUMBER                                   water_capacity ;
        NUMBER                                   available_water ;
        NUMBER                                   temperature ;
        NUMBER                                   carbon ;
        NUMBER                                   nitrogen ;

        /* operations */
        NUMBER                                   Get_depth (void) ;
        void                                     Set_depth (NUMBER) ;
        NUMBER                                   Get_texture (void) ;
        void                                     Set_texture (NUMBER) ;
        NUMBER                                   Get_hydraulic_conductivity (void) ;
        void                                     Set_hydraulic_conductivity (NUMBER) ;
        NUMBER                                   Get_transmissivity (void) ;
        void                                     Set_transmissivity (NUMBER) ;
        NUMBER                                   Get_water_capacity (void) ;
        void                                     Set_water_capacity (NUMBER) ;
        NUMBER                                   Get_available_water (void) ;
        void                                     Set_available_water (NUMBER) ;
        NUMBER                                   Get_temperature (void) ;
        void                                     Set_temperature (NUMBER) ;
        NUMBER                                   Get_carbon (void) ;
        void                                     Set_carbon (NUMBER) ;
        NUMBER                                   Get_nitrogen (void) ;
        void                                     Set_nitrogen (void) ;
} ;
```

```
class   CLIMATE_DESCRIPTOR      : public    LANDUNIT_DESCRIPTOR
{
        /* attributes */
        NUMBER                              maximum_temperature ;
        NUMBER                              minimum_temperature ;
        NUMBER                              dew_point ;
        NUMBER                              shortwave_radiation ;
        NUMBER                              precipitation ;

        /* operations */
        NUMBER                              Get_maximum_temperature (void) ;
        void                                Set_maximum_temperature (NUMBER) ;
        NUMBER                              Get_minimum_temperature (void) ;
        void                                Set_minimum_temperature (NUMBER) ;
        NUMBER                              Get_dew_point (void) ;
        void                                Set_dew_point (NUMBER) ;
        NUMBER                              Get_shortwave_radiation (void) ;
        void                                Set_shortwave_radiation (NUMBER) ;
        NUMBER                              Get_precipitation (void) ;
        void                                Set_precipitation (NUMBER) ;
} ;

class   VEG_DESCRIPTOR          : public    ECOSYSTEM_DESCRIPTOR
{
        /* attributes */
        NUMBER                              gross_primary_production ;
        NUMBER                              net_primary_production ;
        NUMBER                              maintenance_carbon ;
        NUMBER                              growth_carbon ;

        /* operations */
        NUMBER                              Get_gross_primary_production (void) ;
        void                                Set_gross_primary_production
                                                (NUMBER) ;
        NUMBER                              Get_net_primary_production (void) ;
        void                                Set_net_primary_production (NUMBER) ;
        NUMBER                              Get_maintenance_carbon (void) ;
        void                                Set_maintenance_carbon (NUMBER) ;
        NUMBER                              Get_growth_carbon (void) ;
        void                                Set_growth_carbon (NUMBER) ;
} ;
```

2e. *Operation Classes*

```
class   OPERATION   : public   DOCUMENTED_ENTITY
{
        /* attributes */
        DATE                           date_performed ;
        TIME                           time_performed ;

        /* operations */
        DATE                           Get_date_performed (void) ;
        void                           Set_date_performed (DATE) ;
        TIME                           Get_time_performed (void) ;
        void                           Set_time_performed (TIME) ;
        void                           Perform (void) ;
} ;

class   MODEL_OPERATION   : public   OPERATION
{
        /* attributes */
        ANALYST                        performed_by ;
        COMMAND                        used_command ;

        /* operations */
        ANALYST                        Get_analyst (void) ;
        void                           Set_analyst (ANALYST) ;
        COMMAND                        Get_command (void) ;
        void                           Set_command (COMMAND) ;
} ;

class   SPATIAL_CONSUMER : public MODEL_OPERATION
{
        /* attributes */
        SPATIAL_MODEL                  input ;

        /* operations */
        SPATIAL_MODEL                  Get_input (void) ;
        void                           Set_input (SPATIAL_MODEL) ;
} ;
```

```
class   SPATIAL_PRD_CONSUMER : public          SPATIAL_CONSUMER
{
        /* attributes */
        SPATIAL_MODEL                          output ;

        /* operations */
        SPATIAL_MODEL                          Get_output (void) ;
        void                                   Set_output (SPATIAL_MODEL) ;
} ;

class   PROJECTION_TRANSFORM  : public SPATIAL_PRD_CONSUMER
{
        /* attributes */
        PROJECTION_PARAMETERS                  parameters ;

        /* operations */
        PROJECTION_PAREMETERS                  Get_paramters (void) ;
        void                                   Set_parameters
                                                   (PROJECTION_PAREMETERS) ;
} ;

class   RESAMPLE      : public       SPATIAL_PRD_CONSUMER
{
        /* attributes */
        RESAMPLE_WINDOW                        window ;

        /* operations */
        RESAMPLE_WINDOW                        Get_window (void) ;
        void                                   Set_window (RESAMPLE_WINDOW) ;

} ;

class   OVERLAY_CONSUMER        : public       MODEL_OPERATION
{
        /* attributes */
        OVERLAY                                input ;

        /* operations */
        OVERLAY                                Get_overlay (void) ;
        void                                   Set_overlay (OVERLAY) ;
} ;
```

```
class   REGISTER            : public    OVERLAY_CONSUMER
{
        /* attributes */
        OVERLAY                         input_slave ;
        GCP_SET                         ground_control_points ;
        OVERLAY                         output_slave ;

        /* operations */
        OVERLAY                         Get_input_slave (void) ;
        void                            Set_input_slave (OVERLAY) ;
        GCP_SET                         Get_ground_control_points (void) ;
        void                            Set_ground_control_points
                                            (GCP_SET) ;
        OVERLAY                         Get_output_slave (void) ;
        void                            Set_output_slave (OVERLAY) ;
} ;

class   ECOMODEL_ASSIGN             : public    OVERLAY_CONSUMER
{
        /* attributes */
        ECOMODEL                        output ;

        /* operations */
        ECOMODEL                        Get_output (void) ;
        void                            Set_output (ECOMODEL) ;
} ;

class   CATEGORICAL_CONSUMER        : public    MODEL_OPERATION
{
        /* attributes */
        CATEGORICAL_OVERLAY             input ;

        /* operations */
        CATEGORICAL_OVERLAY             Get_input (void) ;
        void                            Set_input (CATEGORICAL_OVERLAY) ;
} ;
```

```
class   REPARTITION  : public         CATEGORICAL_CONSUMER
{
        /* attributes */
        CATEGORICAL_OVERLAY                     output ;

        /* operations */
        CATEGORICAL_OVERLAY                     Get_output (void) ;
        void                                    Set_output
                                                    (CATEGORICAL_OVERLAY) ;
} ;

class MERGE            : public         REPARTITION
{
        /* attributes */
        SIMILARITY_MATRIX                       similarity_matrix ;

        /* operations */
        SIMILARITY_MATRIX                       Get_similarity_matrix (void) ;
        void                                    Set_similarity_matrix
                                                    SIMILARITY_MATRIX) ;
} ;

class   WINDOW_REPARTITION      : public        REPARTITION
{
        /* attributes */
        REPARTITION_WINDOW                      window ;

        /* operations */
        REPARTITION_WINDOW                      Get_scan_window (void) ;
        void                                    Set_scan_window
                                                    (REPARTITION_WINDOW) ;
} ;

class   NUMERICAL_CONSUMER      : public        MODEL_OPERATION
{
        /* attributes */
        NUMERICAL_OVERLAY                       input ;

        /* operations */
        NUMERICAL_OVERLAY                       Get_input (void) ;
        void                                    Set_input (NUMERICAL_OVERLAY) ;
} ;
```

```
class   NUMERICAL_PROD_CONSUMER          : public        NUMERICAL_CONSUMER
{
        /* attributes */
        NUMERICAL_OVERLAY                output ;

        /* operations */
        NUMERICAL_OVERLAY                Get_output (void) ;
        void                             Set_output (NUMERICAL_OVERLAY) ;
} ;

class CONVOLUTION   : public    NUMERICAL_PROD_CONSUMER
{
        /* attributes */
        AVERAGE_WINDOW                   window ;

        /* operations */
        AVERAGE_WINDOW                   Get_window (void) ;
        void                             Set_window (AVERAGE_WINDOW) ;
} ;

class   OVERLAY_ALGEBRA     : public    NUMERICAL_PROD_CONSUMER
{
        /* attributes */
        NUMERICAL_OVERLAY                addend ;

        /* operations */
        NUMERICAL_OVERLAY                Get_addend (void) ;
        void                             Set_addend (NUMERICAL_OVERLAY) ;
} ;

class   INTERPOLATE : public    NUMERICAL_PROD_CONSUMER
{
        /* attributes */
        INTERPOLATE_WINDOW               window ;

        /* operations */
        INTERPOLATE_WINDOW               Get_window (void) ;
        void                             Set_window
                                             (INTERPOLATE_WINDOW) ;
} ;
```

```
class   GENERIC_NUMERICAL          : public     NUMERICAL_PROD_CONSUMER
{
        /* attributes */
        STRING                                  operation_type ;

        /* operations */
        STRING                                  Get_operation_type (void) ;
        void                                    Set_operation_type (STRING) ;
} ;

class   PARTITIONER  : public      NUMERICAL_CONSUMER
{
        /* attributes */
        CATEGORICAL_OVERLAY                     output ;

        /* operations */
        CATEGORICAL_OVERLAY                     Get_output (void) ;
        void                                    Set_output
                                                    (CATEGORICAL_OVERLAY) ;
} ;

class   SPECTRAL_CLASSIFICATION  : public       PARTITIONER
{
        /* attributes */
        SPECTRAL_SCHEME                         input_class_scheme ;
        SPECTRAL_BANK                           input_spectral_bank ;
        CATEGORICAL_OVERLAY                     output ;

        /* operations */
        SPECTRAL_SCHEME                         Get_input_class_scheme (void) ;
        void                                    Set_input_class_scheme
                                                    (SPECTRAL_SCHEME) ;
        SPECTRAL_BANK                           Get_input_spectral_bank (void) ;
        void                                    Set_input_spectral_bank
                                                    (SPECTRAL_BANK) ;
        CATEGORICAL_OVERLAY                     Get_categorical_overlay (void) ;
        void                                    Set_categorical_overlay
                                                    (CATEGORICAL_OVERLAY) ;
} ;
```

```
class   CLUSTER_ANALYSIS        : public    NUMERICAL_CONSUMER
{
        /* attributes */
        SPECTRAL_BANK                       output_bank ;
        SPECTRAL_SCHEME                     output_scheme ;

        /* operations */
        SPECTRAL_BANK                       Get_output_spectral_bank (void) ;
        void                                Set_output_spectral_bank
                                                (SPECTRAL_BANK) ;
        SPECTRAL_SCHEME                     Get_output_class_scheme (void) ;
        void                                Set_output_class_scheme
                                                (SPECTRAL_SCHEME) ;
} ;

class   TOPO_PARTITION          : public    PARTITIONER
{
        /* attributes */
        NUMERICAL_OVERLAY                   output_digital_area_transform ;
        CATEGORICAL_OVERLAY                 output_partitions ;

        /* operations */
        NUMERICAL_OVERLAY                   Get_output_digital_area_transform (void) ;
        void                                Set_output_digital_area_transform
                                                (NUMERICAL_OVERLAY) ;
        CATEGORICAL_OVERLAY                 Get_output_partitions (void) ;
        void                                Set_output_partitions
                                                (CATEGORICAL_OVERLAY) ;
} ;

class   COVER_CONSUMER          : public    MODEL_OPERATION
{
        /* attributes */
        COVER_MODEL                         input_model ;
        COVER_SCHEME                        input_scheme ;

        /* operations */
        COVER_MODEL                         Get_input_model (void) ;
        void                                Set_input_model (COVER_MODEL) ;
        COVER_SCHEME                        Get_input_scheme (void) ;
        void                                Set_input_scheme (COVER_SCHEME) ;
} ;
```

```
class   COVER_TRAINING          : public        COVER_CONSUMER
{
        /* attributes */
        COVER_SCHEME                            output_scheme ;

        /* operations */
        COVER_SCHEME                            Get_output_scheme (COVER_SCHEME) ;
        void                                    Set_output_scheme (COVER_SCHEME) ;
} ;

class   COVER_CLASSIFY          : public        COVER_CONSUMER
{
        /* attributes */
        COVER_MODEL                             output_model ;

        /* operations */
        COVER_MODEL                             Get_output_model (void) ;
        void                                    Set_output_model (COVER_MODEL) ;
} ;

class   SOIL_CONSUMER           : public        MODEL_OPERATION
{
        /* attributes */
        SOIL_MODEL                              input_model ;
        SOIL_SCHEME                             input_scheme ;

        /* operations */
        SOIL_MODEL                              Get_input_model(void) ;
        void                                    Set_inputt_model (SOIL_MODEL) ;
        SOIL_SCHEME                             Get_input_scheme (void) ;
        void                                    Set_input_scheme (SOIL_SCHEME) ;
} ;
```

```
class   SOIL_TRAINING            : public      SOIL_CONSUMER
{
        /* attributes */
        SOIL_SCHEME                            output_scheme ;

        /* operations */
        SOIL_SCHEME                            Get_output_scheme (SOIL_SCHEME) ;
        void                                   Set_output_scheme (SOIL_SCHEME) ;
} ;

class   SOIL_CLASSIFY            : public      SOIL_CONSUMER
{
        /* attributes */
        SOIL_MODEL                             output_model ;

        /* operations */
        SOIL_MODEL                             Get_output_model (void) ;
        void                                   Set_output_model(SOIL_MODEL) ;
} ;

class   ECOSERIES_CONSUMER       : public      MODEL_OPERATION
{
        /* attributes */
        ECOMODEL_TIME_SERIES                   input ;

        /* operations */
        ECOMODEL_TIME_SERIES                   Get_input (void) ;
        void                                   Set_input (ECOMODEL_TIME_SERIES) ;

} ;
class   FOREST_BGC   : public ECOSERIES_CONSUMER
{
        /* attributes */
        ECOMODEL_TIME_SERIES                   output ;

        /* operations */
        ECOMODEL_TIME_SERIES                   Get_output (void) ;
        void                                   Set_output
                                                   (ECOMODEL_TIME_SERIES) ;
} ;
```

```
class   NUMSERIES_CONSUMER      : public        MODEL_OPERATION
{
        /* attributes */
        NUMERICAL_TIME_SERIES                   input ;

        /* operations */
        NUMERICAL_TIME_SERIES                   Get_input (void) ;
        void                                    Set_input (NUMERICAL_TIME_SERIES) ;
}

class   MTCLIM          : public        NUMSERIES_CONSUMER
{
        /* attributes */
        NUMERICAL_TIME_SERIES                   output ;

        /* operations */
        NUMERICAL_TIME_SERIES                   Get_output (void) ;
        void                                    Set_output
                                                    (NUMERICAL_TIME_SERIES) ;
};

class   ELEVATION_ACQUISITION   : public        OPERATION
{
        /* attributes */
        STEREO_PAIR                             air_photo_pair ;
        NUMERICAL_OVERLAY                       output_elevation_model ;
        ANALYST                                 performed_by ;

        /* operations */
        STEREO_PAIR                             Get_air_photo_pair (void) ;
        void                                    Set_air_photo_pair (STEREO_PAIR) ;
        NUMERICAL_OVERLAY                       Get_output_elevation_model (void) ;
        void                                    Set_outptu_elevation_model
                                                    (NUMERICAL_OVERLAY) ;
        ANALYST                                 Get_analyst (vodi) ;
        void                                    Set_analyst (ANALYST) ;
};
```

```
class   SPECTRAL_ACQUISITION      : public      OPERATION
{
        /* attributes */
        SENSOR                                  sensor ;
        POINT_MODEL                             output_spectral_model ;
        NUMBER                                  solar_zenith ;
        NUMBER                                  solar_azimuth ;

        /* operations */

        SENSOR                                  Get_sensor (void) ;
        void                                    Set_sensor (SENSOR) ;
        POINT_MODEL                             Get_output_spectral_model (void) ;
        void                                    Set_output_spectral_model
                                                    (POINT_MODEL) ;
} ;

class   CLIMATE_ACQUISITION       : public      OPERATION
{
        /* attributes */
        NWS_MET_STATION                         climate_station ;
        NUMERICAL_OVERLAY                       output_climate_model ;
        CLIMATE_RECORDER                        data_recorder ;

        /* operations */
        NWS_MET_STATION                         Get_climate_station (void) ;
        void                                    Set_climate_station
                                                    (NWS_MET_STATION) ;
        NUMERICAL_OVERLAY                       Get_output_climate_model (void) ;
        void                                    Set_output_climate_model
                                                    (NUMERICAL_OVERLAY) ;
        CLIMATE_RECORDER                        Get_data_recorder (void) ;
        void                                    Set_data_recorder
                                                    (CLIMATE_RECORDER) ;
} ;
```

```
class   FIELD_ACQUISITION        : public        OPERATION
{
        /* attributes */
        FIELD_WORKER                             field_worker ;
        SAMPLING_SYSTEM                          sampling_system ;
        AERIAL_PHOTO                             air_photo ;
        TRAINING_SITE                            training_sitet ;
        FIELD_FORM                               field_form  ;

        /* operations */
        FIELD_WORKER                             Get_field_worker (void) ;
        void                                     Set_field_worker (FIELD_WORKER) ;
        SAMPLING_SYSTEM                          Get_sampling_system (void) ;
        void                                     Set_sampling_system
                                                     SAMPLING_SYSTEM) ;
        AERIAL_PHOTO                             Get_air_photo (void) ;
        void                                     Set_air_photo (AERIAL_PHOTO) ;
        TRAINING_SITE                            Get_training_site (void) ;
        void                                     Set_training_site (TRAINING_SITE) ;
        FIELD_FORM                               Get_field_form (void) ;
        void                                     Set_field_form (FIELD_FORM) l;
} ;
```

2f. *Classification Classes*

```
class   SPECTRAL_SPACE : public     DOCUMENTED_ENTITY
{
        /* attributes */
        NUMBER                              dimensionality ;
        SET < COLOR_GROUP >                 color_groups ;

        /* operations */
        NUMBER                              Get_dimensionality (void) ;
        void                                Set_dimensionality (NUMBER) ;
        COLOR_GROUP                         Iterate_over_color_groups (void) ;
} ;

class   SIGNATURE    : public       DOCUMENTED_ENTITY
{
        /* attributes */
        NUMBER                              value ;
        TRAINING_SAMPLE                     derived_from ;
        USER_DEFINED_CLASS                  characterized_class ;

        /* operations */
        NUMBER                              Get_value (void) ;
        void                                Set_value (NUMBER) ;
        TRAINING_SAMPLE                     Get_training_sample (void) ;
        void                                Set_training_sample
                                                (TRAINING_SAMPLE) ;
        USER_DEFINED_CLASS                  Get_characterized_class (void) ;
        void                                Set_characterized_class
                                                (USER_DEFINED_CLASS) ;
        void                                Compute_value  (NUMBER) ;
        BOOLEAN                             Is_match (NUMBER) ;
} ;
```

```
class   SPECTRAL_BANK : public      DOCUMENTED_ENTITY
{
        /* attributes */
        SET < BRIGHTNESS_GROUP>                   brightness_groups ;

        /* operations */
        BRIGHTNESS_GROUP                          Iterate_over_brightness_groups (void) ;
        void                                      Set_brightness_group
                                                      (BRIGHTNESS_GROUP) ;
} ;

class   SPECTRAL_GROUP          : public          DOCUMENTED_ENTITY
{

        /* attributes */
        NUMBER                                    correlation_coefficient ;
        /* operations */
        NUMBER                                    Get_correlation_coefficient (void) ;
        void                                      Set_correlation_coefficient (NUMBER) ;
} ;

class   COLOR_GROUP : public        SPECTRAL_GROUP
{
        /* attributes */
        NUMBER                                    number_brightness_groups ;
        SET  < BRIGHTNESS_GROUP >                 brightness_groups ;

        /* operations */
        NUMBER                                    Get_number_brightness_groups (void) ;
        void                                      Set_number_brightness_groups
                                                      (NUMBER) ;
        BRIGHTNESS_GROUP                          Iterate_over_brightness_groups (void) ;
} ;
```

```
class   BRIGHTNESS_GROUP : public   SPECTRAL_GROUP
{
        /* attributes */
        COLOR_GROUP                 containing_color_group ;
        SPECTRAL_CLASS              spectral_class ;
        NUMERICAL_POINT             point_1 ;
        NUMERICAL_POINT             point_2 ;
        NUMERICAL_POINT             point_3 ;

        /* operations */
        COLOR_GROUP                 Get_containing_color_group (void) ;
        void                        Set_containing_color_group
                                        (COLOR_GROUP) ;
        SPECTRAL_CLASS              Get_spectral_class (void) ;
        void                        Set_spectral_class
                                        (SPECTRAL_CLASS) ;
        NUMERICAL_POINT             Get_point_1 (void) ;
        void                        Set_point_1 (NUMERICAL_POINT) ;
        NUMERICAL_POINT             Get_point_2 (void) ;
        void                        Set_point_2 (NUMERICAL_POINT) ;
        NUMERICAL_POINT             Get_point_3 (void) ;
        void                        Set_point_3 (NUMERICAL_POINT) ;
        BOOLEAN                     Is_match
                                        (NUMBER, NUMBER, NUMBER) ;
} ;

class   TRAINING_SAMPLE         : public        DOCUMENTED_ENTITY
{
        /* attributes */
        SET < TRAINING_SITE >       training_sites ;
        SIGNATURE                   signature ;

        /* operations */
        TRAINING_SITE               Iterate_over_training_sites (void) ;
        SIGNATURE                   Get_signature (void) ;
        void                        Set_signature (SIGNATURE) ;
} ;
```

```
class   SIMILARITY_MATRIX          : public        DOCUMENTED_ENTITY
{
        /* attributes */
        MATRIX < SIMILARITY >                      similarities ;

        /* operations */
        SIMILARITY                                 Get_similarity
                                                       (USER_CLASS, USER_CLASS) ;
        void                                       Set_similarity
                                                       (USER_CLASS, USER_CLASS) ;
} ;

class   SIMILARITY : public        DOCUMENTED_ENTITY
{
        NUMBER                                     value ;
        USER_DEFINED_CLASS                         from_class ;
        USER_DEFINED_CLASS                         to_class ;

        /* operations */
        NUMBER                                     Get_value (void) ;
        void                                       Set_value (NUMBER) ;
        USER_CLASS                                 Get_from_class (void) ;
        void                                       Set_from_class
                                                       (USER_CLASS) ;
        USER_CLASS                                 Get_to_class (void) ;
        void                                       Set_to_class (USER_CLASS) ;
} ;
```

```
class   MEMBERSHIP_FUNCTION     : public        DOCUMENTED_ENTITY
{
        /* attribute */
        STRING                                  attribute_defined ;
        USER_DEFINED_CLASS                      class_defined ;
        FUNCTION                                function ;

        /* operations */
        STRING                                  Get_attribute_defined (void) ;
        void                                    Set_attribute_defined (STRING) ;
        USER_CLASS                              Get_class_defined (void) ;
        void                                    Set_class_defined  (USER_CLASS) ;
        FUNCTION                                Get_function (void) ;
        void                                    Set_function (FUNCTION) ;
        NUMBER                                  Compute_possibility (NUMBER) ;
} ;

class   SCHEME_COMPONENT        : public DOCUMENTED_ENTITY
{
        /* attributes */
        CLASS_SCHEME                            containing_class_scheme ;

        /* operations */
        CLASS_SCHEME                            Get_containing_class_scheme (void) ;
        void                                    Set_containing_class_scheme
                                                    (CLASS_SCHEME) ;
} ;

class   CLASS           : public        CLASS_SCHEME_COMPONENT
{
        /* attributes */
        SPECIALIZATION                          child_specialization ;
        NUMBER                                  number_children ;
        NUMBER                                  id ;

        /* operations */
        SPECIALIZATION                          Get_child_specialization (void) ;
        void                                    Set_child_specialization
                                                    (SPECIALIZATION) ;
        NUMBER                                  Get_id (void) ;
        void                                    Set_id (NUMBER
} ;
```

```
class   ROOT_CLASS   : public        CLASS
{
        /* attributes */
        CLASS_SCHEME                           containing_class_scheme ;

        /* operations */
        CLASS_SCHEME                           Get_containing_class_scheme (void) ;
        void                                   Set_containing_class_scheme
                                                   CLASS_SCHEME) ;
} ;

class   USER_CLASS   : public        CLASS
{
        /* attributes */
        SPECIALIZATION                         parent_specialization ;
        SIGNATURE                              signature ;
        MEMBERSHIP_FUNCTION                    membership_function ;

        /* operations */
        SPECIALIZATION                         Get_parent_specialization (void) ;
        void                                   Set_parent_specialization
                                                   (SPECIALIZATION) ;
        SIGNATURE                              Get_signature (void) ;
        void                                   Set_signature (SIGNATURE) ;
        MEMBERSHIP_FUNCTION                    Get_membership_function (void) ;
        void                                   Set_membership_function
                                                   (MEMBERSHIP_FUNCTION) ;

} ;
class   SOIL_CLASS   :               public        USER_DEFINED_CLASS
{
        /* attributes */
        SOIL_REGION                            described_region ;

        /* operations */
        SOIL_REGION                            Get_described_region (void) ;
        void                                   Set_described_region (SOIL_REGION) ;

} ;
```

```
class   COVER_CLASS : public        USER_DEFINED_CLASS
{
        /* attributes */
        COVER_REGION                            described_region ;

        /* operations */
        COVER_REGION                            Get_described_region (void) ;
        void                                    Set_described_region (COVER_REGION) ;
} ;

class   SPECTRAL_CLASS          : public        USER_DEFINED_CLASS
{
        /* attributes */
        CATEGORICAL_REGION                      described_region ;

        /* operations */
        CATEGORICAL_REGION                      Get_described_region (void) ;
        void                                    Set_described_region
                                                    (CATEGORICAL_REGION) ;
} ;

class   SPECIALIZATION          : public        CLASS_SCHEME_COMPONENT
{
        /* attributes */
        CLASS                                   parent_class ;
        USER_DEFINED_CLASS                      child_class ;

        /* operations */
        CLASS                                   Get_parent_class (void) ;
        void                                    Set_parent_class (CLASS) ;
        USER_DEFINED_CLASS                      Get_child_class (void) ;
        void                                    Set_child_class
                                                    (USER_DEFINED_CLASS) ;
} ;
```

```
class   CLASS_SCHEME              : public       DOCUMENTED_ENTITY
{
        /* attributes */
        ROOT_CLASS                               root ;

        /* operations */
        ROOT_CLASS                               Get_root (void) ;
        void                                     Set_root (ROOT_CLASS) ;
} ;

class   SOIL_SCHEME  : public     CLASS_SCHEME
{
        /* attributes */
        SOIL_MODEL                               described_model ;
        SET < SOIL_CLASS >                       classes ;

        /* operations */

        SOIL_MODEL                               Get_described_model (void) ;
        void                                     Set_described_model (SOIL_MODEL) ;
        SOIL_CLASS                               Iterate_over_classes (void) ;
        void                                     Set_class (SOIL_CLASS) ;
} ;

class   COVER_SCHEME              : public       CLASS_SCHEME
{
        /* attributes */
        COVER_MODEL                              described_model ;
        SET < COVER_CLASS >                      classes ;

        /* operations */
        COVER_MODEL                              Get_described_model (void) ;
        void                                     Set_described_model (COVER_MODEL) ;
        COVER_CLASS                              Iterate_over_classes (void) ;
        void                                     Set_class (COVER_CLASS) ;
} ;
```

```
class   SPECTRAL_SCHEME              : public        CLASS_SCHEME
{
        /* attributes */
        CATEGORICAL_OVERLAY                          described_overlay ;
        SET < SPECTRAL_CLASS >                       clases ;

        /* operations */
        CATEGORICAL_OVERLAY                          Get_described_overlay (void) ;
        void                                         Set_described_overlay
                                                         (CATEGORICAL_OVERLAY) ;
        SPECTRAL_CLASS                               Iterate_over_classes (void) ;
        void                                         Set_class (SPECTRAL_CLASS) ;
} ;
```

2g. *Data Acquisition Classes*

```
class    MET_STATION : public        DOCUMENTED_ENTITY
{
         /* attributes */
         NUMBER                                elevation ;
         COORDINATE                            coord ;

         /* operations */
         NUMBER                                Get_elevation (void) ;
         void                                  Set_elevation (NUMBER) ;
         COORDINATE                            Get_coord (void) ;
         void                                  Set_coord (COORDINATE) ;
} ;

class    ACTUAL_MET_STATION        : public     MET_STATION
{
         /* attributes */
         DATE                                  start_date ;
         DATE                                  end_date ;

         /* operations */
         DATE                                  Get_start_date (void) ;
         void                                  Set_start_date (DATE) ;
         DATE                                  Get_end_date (void) ;
         void                                  Set_end_date (DATE) ;
} ;

class    VIRTUAL_MET_STATION       : public     MET_STATION
{
         /* attributes */
         DATE                                  date_generated ;

         /* operations */
         DATE                                  Get_date_generated (void) ;
         void                                  Set_date_generated (DATE) ;
} ;
```

```
class   AIRCRAFT : public          DOCUMENTED_ENTITY
{
        /* attribute */
        CAMERA                              carries ;

        /* operations */
        CAMERA                              Get_camera (void) ;
        void                                Set_camera (CAMERA) ;
} ;

class   AERIAL_PHOTO          : public      DOCUMENTED_ENTITY
{
        /* attributes */
        STRING                              type ;
        STRING                              scale ;
        DATE                                acquisition_date ;
        TIME                                acquisition_time ;
        CAMERA                              camera ;

        /* operations */
        STRING                              Get_type (void) ;
        void                                Set_type (STRING) ;
        STRING                              Get_scale (void) ;
        void                                Set_scale (STRING) ;
        DATE                                Get_acquisition_date (void) ;
        void                                Set_acquisition_date (DATE) ;
        TIME                                Get_acquisition_time (void) ;
        void                                Set_acquisition_time (TIME) ;
        CAMERA                              Get_camera (void) ;
        void                                Set_camera (CAMERA) ;
} ;
```

```
class   STEREO_PAIR  : public         DOCUMENTED_ENTITY
{
        /* attributes */
        AERIAL_PHOTO                            photo_1 ;
        AERIAL_PHOTO                            photo_2 ;


        /* operations */
        AERIAL_PHOTO                            Get_photo_1 (void) ;
        void                                    Set_photo_1 (AERIAL_PHOTO) ;
        AERIAL_PHOTO                            Get_photo_2 (void) ;
        void                                    Set_photo_2 (AERIAL_PHOTO) ;
} ;

class   CAMERA       : public         DOCUMENTED_ENTITY
{
        /* attributes */
        STRING                                  manufacturer ;
        NUMBER                                  focal_length ;

        /* operations */
        STRING                                  Get_manufacturer (void) ;
        void                                    Set_manufacturer (STRING) ;
        NUMBER                                  Get_focal_length (void) ;
        void                                    Set_focal_length (NUMBER) ;
        AERIAL_PHOTO                            Read_data (void) ;
} ;
```

```
class   SATELLITE   : public   DOCUMENTED_ENTITY
{
        /* attributes */
        DATE                launch_date ;
        NUMBER              life_expectancy ;
        NUMBER              ground_track_speed ;
        NUMBER              orbital_period ;
        NUMBER              ground_track_distance ;
        SCANNER             scanner ;
        ORBIT               orbit ;

        /* operations */
        DATE                Get_launch_date (void) ;
        void                Set_launch_date (DATE) ;
        NUMBER              Get_life_expectancy (void) ;
        void                Set_life_expectancy (NUMBER) ;
        NUMBER              Get_ground_track_speed (void) ;
        void                Set_ground_track_speed (NUMBER) ;
        NUMBER              Get_orbital_period (void) ;
        void                Set_orbital_period (NUMBER) ;
        NUMBER              Get_ground_track_distance (void) ;
        void                Set_ground_track_distance (NUMBER) ;
        SCANNER             Get_scanner (void) ;
        void                Set_scanner (SCANNER) ;
        ORBIT               Get_orbit (void) ;
        void                Set_orbit (ORBIT) ;

} ;
```

```
class   ORBIT        : public      DOCUMENTED_ENTITY
{
        /* attributes *
        NUMBER                           altitude ;
        NUMBER                           inclination_angle ;

        /* operations *
        NUMBER                           Get_altitude (void) ;
        void                             Set_altitude (NUMBER) ;
        NUMBER                           Get_inclination_angle (void) ;
        void                             Set_inclination_angle (NUMBER)
} ;

class   SCANNER : public               DOCUMENTED_ENTITY
{
        /* attributes */
        NUMBER                           scan_angle ;
        NUMBER                           instantaneous_field_of_view ;
        NUMBER                           swath_width ;
        SATELLITE                        carried_on ;
        SENSOR                           carries ;

        /* operations */
        NUMBER                           Get_scan_angle (void) ;
        void                             Set_scan_angle (NUMBER) ;
        NUMBER                           Get_instantaneous_field_of_view (void) ;
        void                             Set_instantaneous_field_of_view
                                             (NUMBER) ;
        NUMBER                           Get_swath_width (void) ;
        void                             Set_swath_width (NUMBER) ;
        SATELLITE                        Get_satellite (void) ;
        void                             Set_satellite (SATELLITE) ;
        SENSOR                           Get_sensor (void) ;
        void                             Set_sensor (SENSOR) ;
} ;
```

```
class   SENSOR          : public        DOCUMENTED_ENTITY
{
        /* attributes */
        STRING                                          temporal_resolution ;
        NUMBER                                          radiometric_resolution ;
        SET < EMS_BAND >                                ems_bands ;
        SCANNER                                         scanner ;

        /* operations */
        STRING                                          Get_temporal_resolution (void) ;
        void                                            Set_temporal_resolution (STRING) ;
        NUMBER                                          Get_radiometric_resolution (void) ;
        void                                            Set_radiometric_resolution (NUMBER) ;
        EMS_BAND                                        Iterate_over_ems_bands (void) ;
        SCANNER                                         Get_scanner (void) ;
        void                                            Set_scanner (SCANNER) ;
        POINT_MODEL                                     Read_data (void) ;
} ;

class   EMS_BAND        : public        DOCUMENTED_ENTITY
{
        /* attributes */
        STRING                                          spectral_resolution ;

        /* operations */
        STRING                                          Get_spectral_resolution (void) ;
        void                                            Set_spectral_resolution (STRING) ;
} ;

class   FIELD_FORM      : public        DOCUMENTED_ENTITY
{
        /* attributes */
        SAMPLING_SYSTEM                                 sampling_system ;
        DATA_ITEM                                       data ;

        /* operations */
        SAMPLING_SYSTEM                                 Get_sampling_system (void) ;
        void                                            Set_sampling_system
                                                            (SAMPLING_SYSTEM) ;
        DATA_ITEM                                       Get_data_item (void) ;
        void                                            Set_data_item (DATA_ITEM) ;
} ;
```

```
class   SAMPLING_SYSTEM         : public        DOCUMENTED_ENTITY
{
        /* attributes */
        STRING                                  developer ;

        /* operations */
        STRING                                  Get_developer (void) ;
        void                                    Set_developer (STRING) ;
} ;

class   DATA_ITEM     : public     DOCUMENTED_ENTITY
{
        /* attributes */
        STRING                                  data_type ;
        NUMBER                                  value ;

        /* operations */
        STRING                                  Get_data_type (void) ;
        void                                    Set_data_type (STRING) ;
        NUMBER                                  Get_value (void) ;
        void                                    Set_value (NUMBER) ;
} ;
```

2h. *Descriptor Classes*

```
class   SPATIAL_DESCRIPTOR          : public    DOCUMENTED_ENTITY
{
        /* attributes */
        SPATIAL_MODEL                           described_model ;

        /* operations */
        SPATIAL_MODEL                           Get_described_model (void) ;
        void                                    Set_described_model
                                                    (SPATIAL_MODEL) ;
} ;

class   PROJECTION_PARAMETERS       : public    SPATIAL_DESCRIPTOR
{
        /* attributes */
        NUMBER                                  easting ;
        NUMBER                                  northing ;

        /* operations */
        NUMBER                                  Get_easting (void) ;
        void                                    Set_easting (NUMBER) ;
        NUMBER                                  Get_northing (void) ;
        void                                    Set_northing (void) ;
} ;

class   GENERIC_SPATIAL_DESCRIPTOR              : public        SPATIAL_DESCRIPTOR
{

        /* attributes */
        STRING                                  descriptor_type ;
        STRING                                  descriptor_value ;

        /* operations */
        STRING                                  Get_desc_value (void) ;
        void                                    Set_desc_value (STRING) ;
        STRING                                  Get_desc_type (void) ;
        void                                    Set_desc_type (STRING) ;
} ;
```

```
class   TEMPORAL_DESCRIPTOR       : public     DOCUMENTED_ENTITY
{
        /* attributes */
        STRING                                 temporal_descriptor_type ;
        STRING                                 temporal_descriptor_value ;
        TIME_SERIES                            described_time_series ;

        /* operations */
        STRING                                 Get_temporal_descriptor_type (void) ;
        void                                   Set_temporal_descriptor_type (STRING) ;
        STRING                                 Get_temporal_descriptor_type (void) ;
        void                                   Set_temporal_descriptor_type (STRING) ;
} ;

class   GCP_SET                   : public     DOCUMENTED_ENTITY
{
        /* attributes */
        SET < COORDINATE >                     coordinate_set_1 ;
        SET < COORDINATE >                     coordinate_set_2 ;

        /* operations */
        COORDINATE                             Iterate_over_coordinates ;
        FUNCTION                               Compute_transformation (void) ;
} ;

class   TIME_SERIES   : public    DOCUMENTED_ENTITY
{
        /* attributes */
        TEMPORAL_DESCRIPTOR                    temporal_scale ;
        TEMPORAL_DESCRIPTOR                    temporal_resolution ;

        /* operations */
        TEMPORAL_DESCRIPTOR                    Get_temporal_scale (void) ;
        void                                   Set_temporal_scale
                                                   (TEMPORAL_DESCRIPTOR) ;
        TEMPORAL_DESCRIPTOR                    Get_temporal_resoluiton (void) ;
        void                                   Set_temporal_resolution
                                                   (TEMPORAL_DESCRIPTOR) ;
} ;
```

```
class   NUMERICAL_TIME_SERIES   : public   TIME_SERIES
{

        /* attributes */
        SET < NUMERICAL_OVERLAY >                  numerical_overlays ;
        SET < NUMERICAL_SERIES_CONSUMER >          consuming_operations;

        /* operations */
        NUMERICAL_OVERLAY                          Iterate_over_numerical_overlays (void) ;
        NUMERICAL_SERIES_CONSUMER                  Iterate_over_consuming_operations (void) ;
} ;

class   ECOMODEL_TIME_SERIES   : public   TIME_SERIES
{
        /* attributes */
        SET < ECOMODEL >                           ecomodels ;
        SET < ECOMODEL_SERIES_CONSUMER >           consuming_operations;

        /* operations */
        ECOMODEL                                   Iterate_over_ecomodels (void) ;
        ECOMODEL_SERIES_CONSUMER                   Iterate_over_consuming_operations (void) ;
} ;

class   COORDINATE   : public   ENTITY
{
        /* attributes */
        NUMBER                                     x_coord ;
        NUMBER                                     y_coord ;
        STRING                                     coord_type ;

        /* operations */
        NUMBER                                     Get_x_coord (void) ;
        void                                       Set_x_coord (NUMBER) ;
        NUMBER                                     Get_y_coord (void) ;
        void                                       Set_y_coord (NUMBER) ;
        STRING                                     Get_coord_type (void) ;
        void                                       Set_coord_type (STRING) ;
} ;
```

2i. *Window Classes*

```
class   WINDOW                  : public    DOCUMENTED_ENTITY
{
        /* attributes */
        NUMBER                              dimension_in_x ;
        NUMBER                              dimension_in_y ;

        /* operations */
        NUMBER                              Get_dimension_in_x (void) ;
        void                                Set_dimension_in_x (NUMBER) ;
        NUMBER                              Get_dimension_in_y (void) ;
        void                                Set_dimension_in_y (NUMBER) ;
} ;

class   AVERAGE_WINDOW          : public    WINDOW
{
        /* attributes */
        MATRIX                              coefficient_matrix
        NUMBER                              mean ;

        /* operations */
        MATRIX                              Get_coefficient_matrix (void) ;
        void                                Set_coefficient_matrix (MATRIX) ;
        NUMBER                              Get_value (void) ;
        NUMBER                              Set_value (NUMBER) ;
        NUMBER                              Compute_mean (MATRIX) ;
} ;

class   REPARTITION_WINDOW      : public    WINDOW
{
        /* attributes */
        NUMBER                              majority ;
        NUMBER                              minority ;

        /* operations */
        NUMBER                              Get_majority (void) ;
        void                                Set_majority (NUMBER) ;
        NUMBER                              Get_minority (void) ;
        void                                Set_minority (NUMBER) ;
        NUMBER                              Calculate_majority (void) ;
        NUMBER                              Calculate_minority (void) ;
} ;
```

```
class   TOPO_WINDOW                   : public    WINDOW
{
        /* attributes */
        NUMBER                                    gradient ;
        NUMBER                                    aspect ;

        /* operations */
        NUMBER                                    Get_gradient (void) ;
        void                                      Set_gradient (NUMBER) ;
        NUMBER                                    Get_aspect (void) ;
        void                                      Set_aspect (NUMBER) ;
        NUMBER                                    Calculate_gradient (void) ;
        NUMBER                                    Calculate_aspect (void) ;
} ;

class   SEARCH_WINDOW                 : public    WINDOW
{
        /* attributes */
        NUMBER                                    number_points ;

        /* operations */
        NUMBER                                    Get_number_points (void) ;
        void                                      Set_number_points (NUMBER) ;
        POINT                                     Get_nearest_neighbor (void) ;
} ;

class INTERPOLATE_WINDOW              : public    WINDOW
{
        /* attributes */
        SET < NUMERICAL_POINT>        target_points ;
        NUMBER                                    value ;

        /* operations */
        NUMERICAL_POINT                           Iterate_over_target_points (void) ;
        NUMBER                                    Get_value (void) ;
        void                                      Set_value (NUMBER) ;
        void                                      Compute_weight (void) ;
} ;
```

2j. *Software Classes*

```
class   SOFTWARE_DEVELOPMENT   : public   DOCUMENTED_ENTITY
{
        /* attributes */
        DATE                              start_date ;
        DATE                              end_date ;

        /* operations */
        DATE                              Get_start_date (void) ;
        void                              Set_start_date (DATE) ;
        DATE                              Get_end_date (void) ;
        void                              Set_end_date (DATE) ;
} ;

class   DESIGN   : public   SOFTWARE_DEVELOPMENT
{
        /* attributes */
        SET < DESIGNER >                  designers ;
        SET < IMPLEMENTIONS >             implementations ;

        /* operations */
        DESIGNER                          terate_over_designers (void) ;
        IMPLEMENTATION                    Iterate_over_implementations (void) ;
} ;
```

```
class   IMPLEMENTATION : public     SOFTWARE_DEVELOPMENT
{
    .   /* attributes */
        SET < PROGRAMMER >                      programmers ;
        DESIGN                                  design ;
        IMPLEMENTED_COMPONENT                   product ;
        SET < PROGRAMMING_LANGUAGE >            programming_languages ;

        /* operations */
        PROGRAMMER                              Iterate_over_programmers (void) ;
        DESIGN                                  Get_design (void) ;
        void                                    Set_design (DESIGN) ;
        IMPLEMENTATION_COMPONENT                Get_implementation_component (void) ;
        void                                    Set_implementation_component
                                                  (IMPLEMENTATION_COMPONENT) ;
        PROGRAM_LANGUAGE                        Iterate_over_programming_languages
                                                  (void) ;
} ;

class   SOFTWARE    : public        DOCUMENTED_ENTITY
{
        /* attributes */
        STRING                                  version_number ;
        STRING                                  revision_number ;
        EXECUTABLE                              executable ;
        SET < COMMAND >                         commands ;

        /* operations */
        STRING                                  Get_version_number (void) ;
        void                                    Set_version_number (STRING) ;
        STRING                                  Get_revision_number (void) ;
        void                                    Set_revision_number (STRING) ;
        EXECUTABLE                              Get_executable (void) ;
        void                                    Set_executable  (EXECUTABLE) ;
        COMMAND                                 Iterate_over_commands (void) ;
} ;
```

```
class   CUSTOM_SOFTWARE            : public    SOFTWARE
{
        /* attributes */
        SOURCE_COMPONENT                       source ;

        /* operations */
        SOURCE_COMPONENT                       Get_source (void) ;
        void                                   Set_source (SOURCE_COMPONENT)
} ;

class   COMMERCIAL_SOFTWARE        : public    SOFTWARE
{
        /* attributes */
        STRING                                 producer ;
        STRING                                 vendor ;
        NUMBER                                 cost ;

        /* operations */
        STRING                                 Get_producer (void) ;
        void                                   Set_producer (STRING) ;
        STRING                                 Get_vendor (void) ;
        void                                   Set_vendor (STRING) ;
        NUMBER                                 Get_cost (void) ;
        void                                   Set_cost (NUMBER) ;
} ;

class   IMPLEMENTED_COMPONENT      : public    DOCUMENTED_ENTITY
{
        /* attributes */
        CUSTOM_SOFTWARE                        containing_software ;

        /* operations */
        CUSTOM_SOFTWARE                        Get_containing_software (void) ;
        void                                   Set_containing_software
                                                   (CUSTOM_SOFTWARE) ;
} ;
```

```
class   SOURCE_COMPONENT          : public    IMPLEMENTATION_COMPONENT
{
        /* attributes */
        SET < SOURCE_MODULE >                 source_modules ;

        /* operations */
        SOURCE_MODULE                         Iterate_over_source_modules (void) ;
} ;

class   SOURCE_MODULE             : public    IMPLEMENTATION_COMPONENT
{
        /* attributes */
        SOURCE_COMPONENT                      ontaining_source_component ;

        /* operations */
        SOURCE_COMPONENT                      Get_containing_source_component (void) ;
        void                                  Set_containing_source_component
                                                  (SOURCE_COMPONENT) ;
} ;

class   EXECUTABLE_COMPONENT      : public    DOCUMENTED_ENTITY
{
        /* attributes */
        SOFTWARE                              containing_software ;

        /* operations */
        SOFTWARE                              Get_software (void) ;
        void                                  Set_software (SOFTWARE) ;
} ;

class   COMMAND : public          EXECUTABLE_COMPONENT
{
        /* attributes */
        OPERATION                             operation_used_for ;

        /* operations */
        OPERATION                             Get_operation_used_for (void) ;
        void                                  Set_operation_used_for (OPERATION) ;
} ;
```

```
class   EXECUTABLE  : public            EXECUTABLE_COMPONENT
{
        /* attributes */
        DATE                                        creation_date ;

        /* operations */
        DATE                                        Get_creation_date (void) ;
        void                                        Set_creation_date (DATE) ;
} ;

class   PROGRAMMING_LANGUAGE  : public      DOCUMENTED_ENTITY
{
        /* attributes */
        SET < IMPLEMENTATION >                      implementations ;

        /* operations */
        IMPLEMENTATION                              Iterate_over_implementations (vodi) ;
} ;
```

2k. *Documentation Classes*

```
class   DOCUMENT          : public   NAMED_ENTITY
{
        /* attributes */
        NUMBER                       length ;

        /* operations */
        NUMBER                       Get_length (void) ;
        void                         Set_length (NUMBER) ;
} ;

class   EXTERNAL_DOCUMENT   : public   DOCUMENTED_ENTITY
{
        /* attributes */
        STRING                         writer ;

        /* operations */
        STRING                         Get_writer (void) ;
        void                           Set_writer (STRING) ;
} ;

class   PUB_EXT_DOCUMENT    : public   EXTERNAL_DOCUMENT
{
        /* attributes */
        DATE                           publication_date ;

        /* operations */
        DATE                           Get_publication_date (void) ;
        void                           Set_publication_date (DATE) ;
} ;

class   EXTERNAL_BOOK       : public   PUB_EXT_DOCUMENT
{
        /* attributes */
        STRING                         publisher ;

        /* operations */
        STRING                         Get_publisher (void) ;
        void                           Set_publisher (STRING) ;
} ;
```

```
class   EXTERNAL_ARTICLE        : public    PUB_EXT_DOCUMENT
{
        /* attributes */
        STRING                              journal ;

        /* operations */
        STRING                              Get_journal (void) ;
        void                                Set_journal (STRING) ;
} ;

class   EXTERNAL_THESIS         : public    EXTERNAL_DOCUMENT
{
        /* attributes */
        STRING                              degree ;

        /* operations */
        STRING                              Get_degree (void) ;
        void                                Set_degree (STRING) ;
} ;

class   INTERNAL_DOCUMENT : public          DOCUMENT
{
        /* attributes */
        SET <AUTHOR >                       writers ;

        /* operations */
        AUTHOR                              Iterate_over_writers (void) ;
} ;

class   PUB_INT_DOCUMENT        : public    INTERNAL_DOCUMENT
{
        /* attributes */
        DATE                                publication_date ;

        /* operations */
        DATE                                Get_publication_date (void) ;
        void                                Set_publication_date (DATE) ;
} ;
```

```
class   INTERNAL_BOOK       : public    PUB_INT_DOCUMENT
{

        /* attributes */
        STRING                          publisher ;

        /* operations */
        STRING                          Get_publisher (void) ;
        void                            Set_publisher (STRING) ;
} ;

class   INTERNAL_ARTICLE    : public    PUB_INT_DOCUMENT
{
        /* attributes */
        STRING                          journal ;

        /* operations */
        STRING                          Get_journal (void) ;
        void                            Set_journal (STRING) ;
} ;

class   INTERNAL_THESIS     : public    INTERNAL_DOCUMENT
{
        /* attributes */
        STRING                          degree ;

        /* operations */
        STRING                          Get_degree (void) ;
        void                            Set_degree (STRING) ;
} ;
```

2j. *Human Classes*

```
class   EMPLOYEE   : public   NAMED_ENTITY
{
        /* attributes */
        STRING                          home_address ;
        STRING                          home_phone ;
        STRING                          office ;
        STRING                          office_phone ;
        STRING                          email ;
        STRING                          fax ;
        DATE                            birth_date ;
        DATE                            hire_date ;
        LAB                             employed_in ;

        /* operations */
        STRING                          Get_home_address (void) ;
        void                            Set_home_address (STRING) ;
        STRING                          Get_home_phone (void) ;
        void                            Set_home_phone (STRING) ;
        STRING                          Get_office (void) ;
        void                            Set_office (STRING) ;
        STRING                          Get_office_phone (void) ;
        void                            Set_office_phone (STRING) ;
        STRING                          Get_email (void) ;
        void                            Set_email (STRING) ;
        STRING                          Get_fax (void) ;
        void                            Set_fax (STRING) ;
        DATE                            Get_birth_date (void) ;
        void                            Set_birth_date (DATE) ;
        DATE                            Get_hire_date (void) ;
        void                            Set_hire_date (DATE) ;
        LAB                             Return_lab (void) ;
} ;
```

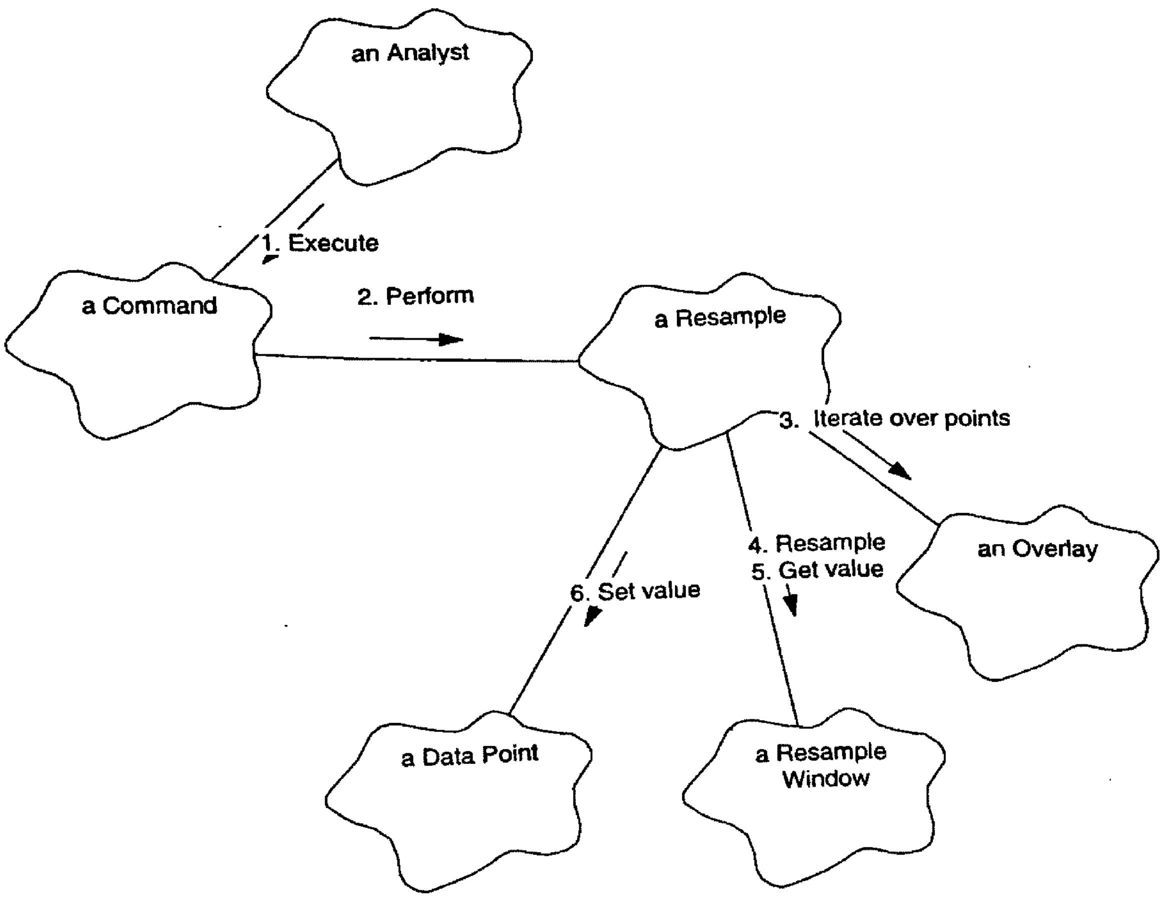
```
class   ADMINISTRATOR        : public     EMPLOYEE
{
        /* attributes */
        SET  < PROJECT>                   projects ;
        LAB                               administers ;

        /* operations */
        PROJECT                           Iterate_over_projects (void) ;
        LAB                               Return_lab (void) ;
} ;

class   AUTHOR      : public      EMPLOYEE
{
        /* attributes */
        SET < INTERNAL_DOCUMENT >         documents ;

        /* operations */
        INTERNAL_DOCUMENT                 Iterate_over_documents (void) ;
} ;

class   ANALYST     : public      EMPLOYEE
{
        /* attributes */
        SET < OPERATION >                 operations ;

        /* operations */
        OPERATION                         Iterate_over_operations (void) ;
} ;

class   PROGRAMMER: public        EMPLOYEE
{
        /* attributes */
        SET < IMPLEMENTATION >            implementations ;

        /* operations */
        IMPLEMENTATION                    Iterate_over_implementation (void) ;
} ;
```

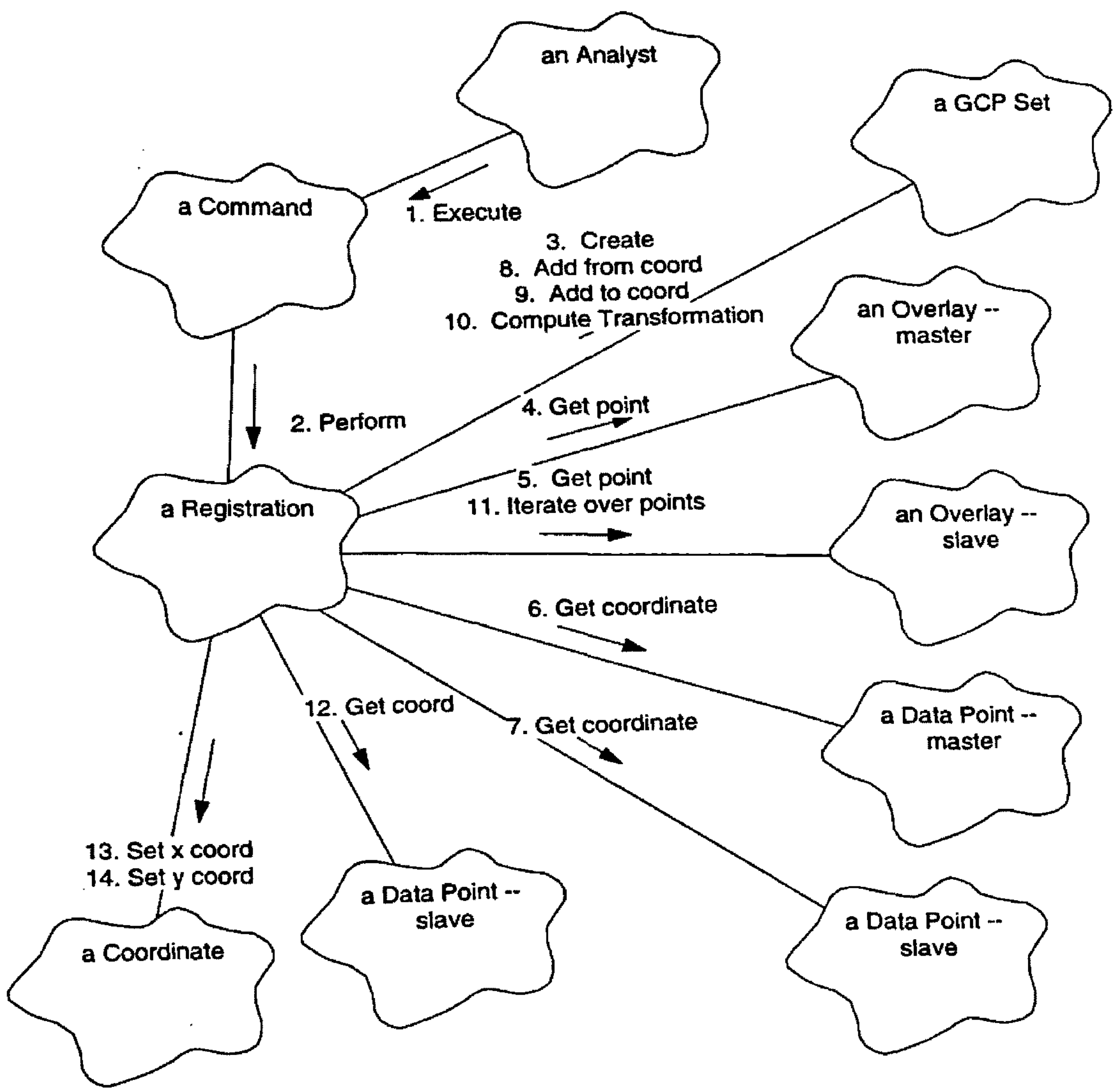
```
class   DESIGNER        : public        EMPLOYEE
{

        /* attributes */
        SET < DESIGN >                          designs ;

        /* operations */
        DESIGN                                  Iterate_over_designs (void) ;
} ;

class   FIELD_WORKER                : public        EMPLOYEE
{
        /* attributes */
        SET < FIELD_DATA_ACQUISITION >          field_data_acquisitions ;

        /* operations */
        FIELD_DATA_ACQUISITION                  Iterate_over_field_data_acquisitions (void) ;
} ;

class   LAB             : public        DOCUMENTED_ENTITY
{
        /* attributes */
        STRING                                  lab_address ;
        STRING                                  lab_phone ;
        STRING                                  lab_email ;
        STRING                                  lab_fax ;
        SET < PROJECT >                         projects ;
        SET < EMPLOYEE >                        employees ;


        /* operations */
        STRING                                  Get_lab_address (void) ;
        void                                    Set_lab_address (STRING) ;
        STRING                                  Get_lab_phone (void) ;
        void                                    Set_lab_phone (STRING) ;
        STRING                                  Get_lab_email (void) ;
        void                                    Set_lab_email (STRING) ;
        STRING                                  Get_lab_fax (void) ;
        void                                    Set_lab_fax (STRING) ;
        PROJECT                                 Iterate_over_projects (void) ;
        EMPLOYEE                                Iterate_over_employees (void) ;
} ;
```

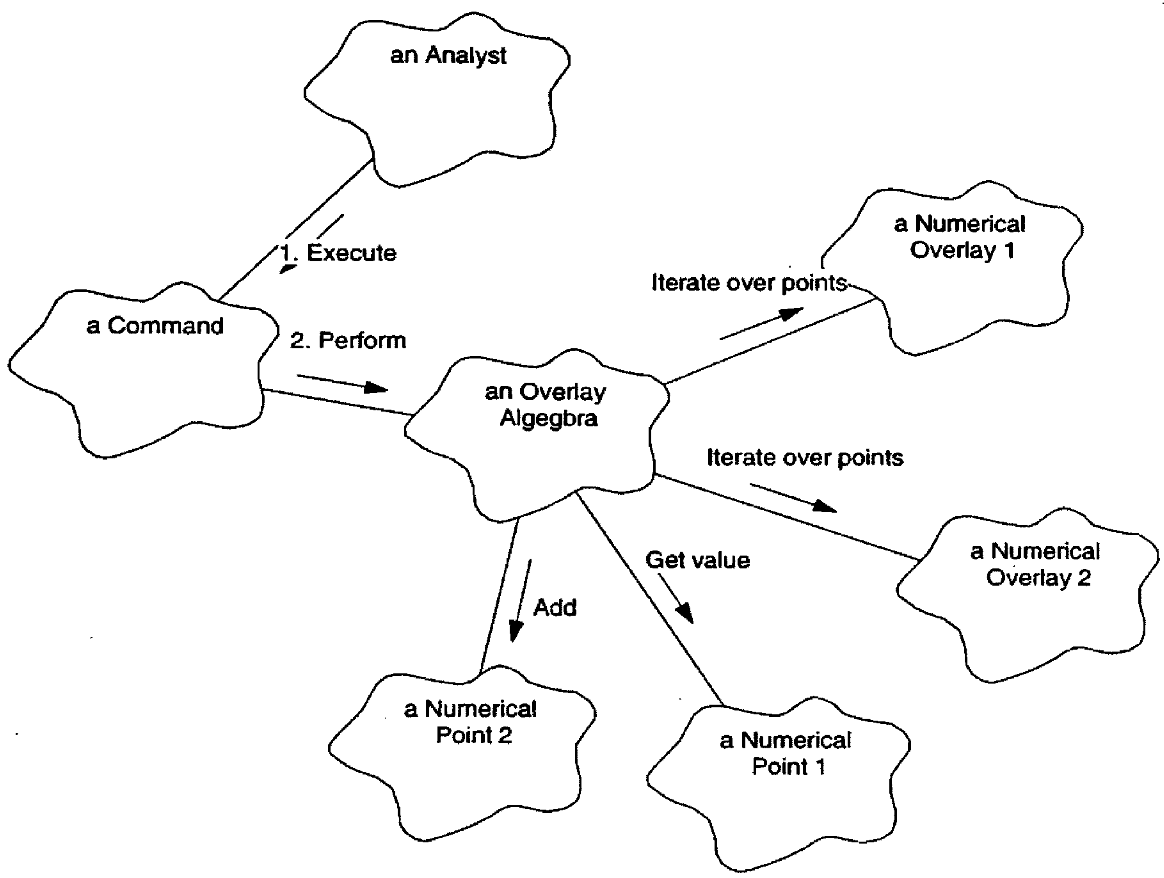
```
class   PROJECT   : public   DOCUMENTED_ENTITY
{
        /* attributes */
        STRING                  funding_source ;
        LAB                     managed_by ;

        /* operations */
        STRING                  Get_funding_source (void) ;
        void                    Set_funding_source (STRING) ;
        LAB                     Return_lab (void) ;
};
```

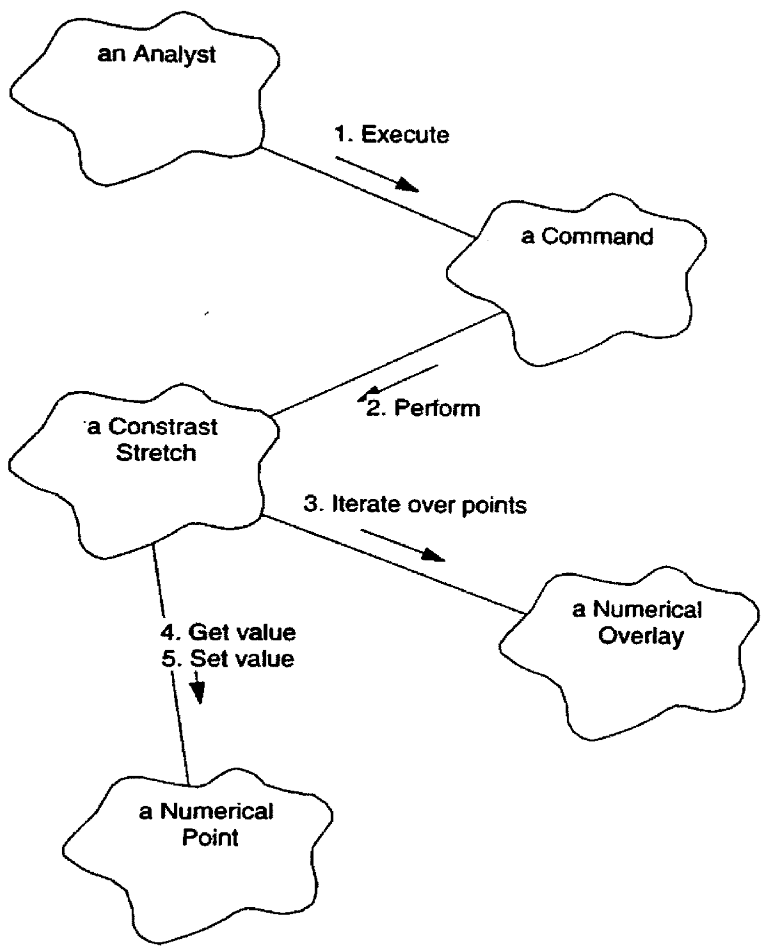
APPENDIX III
SCENARIOS

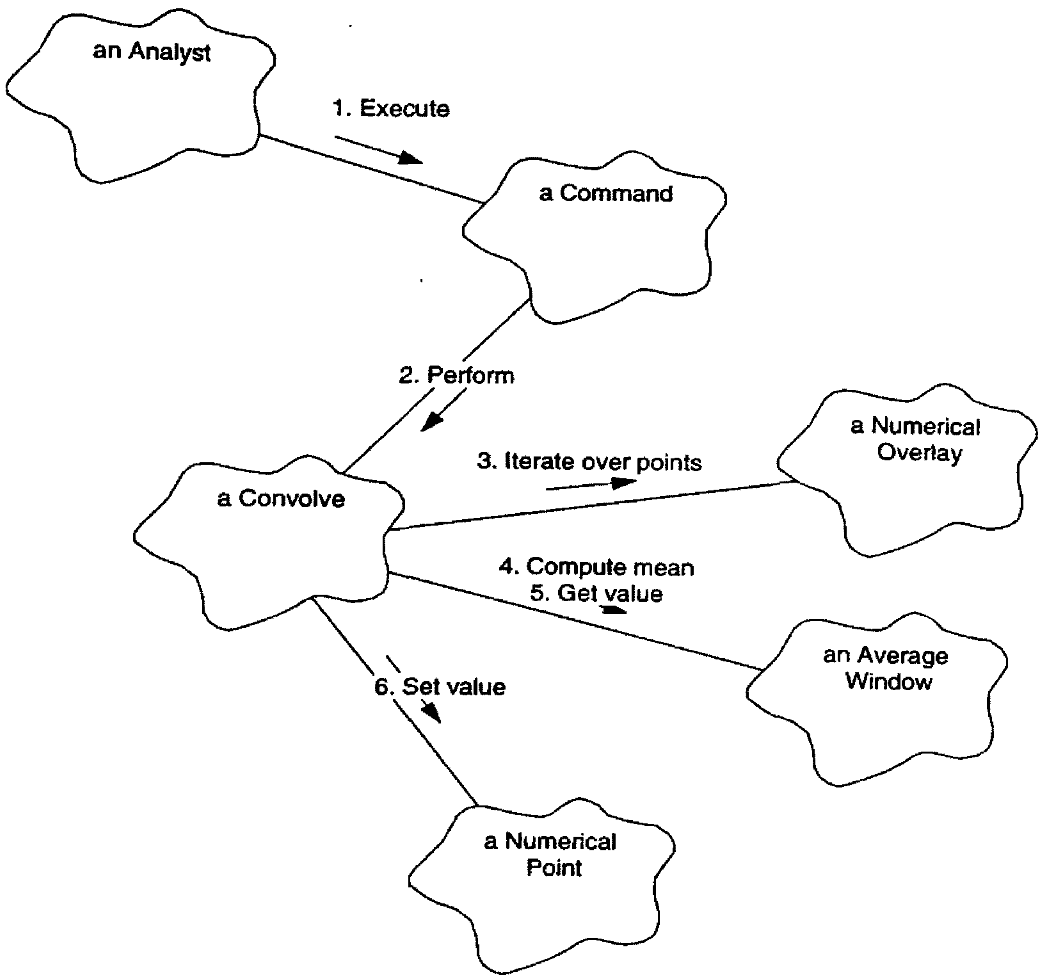
TABLE OF CONTENTS

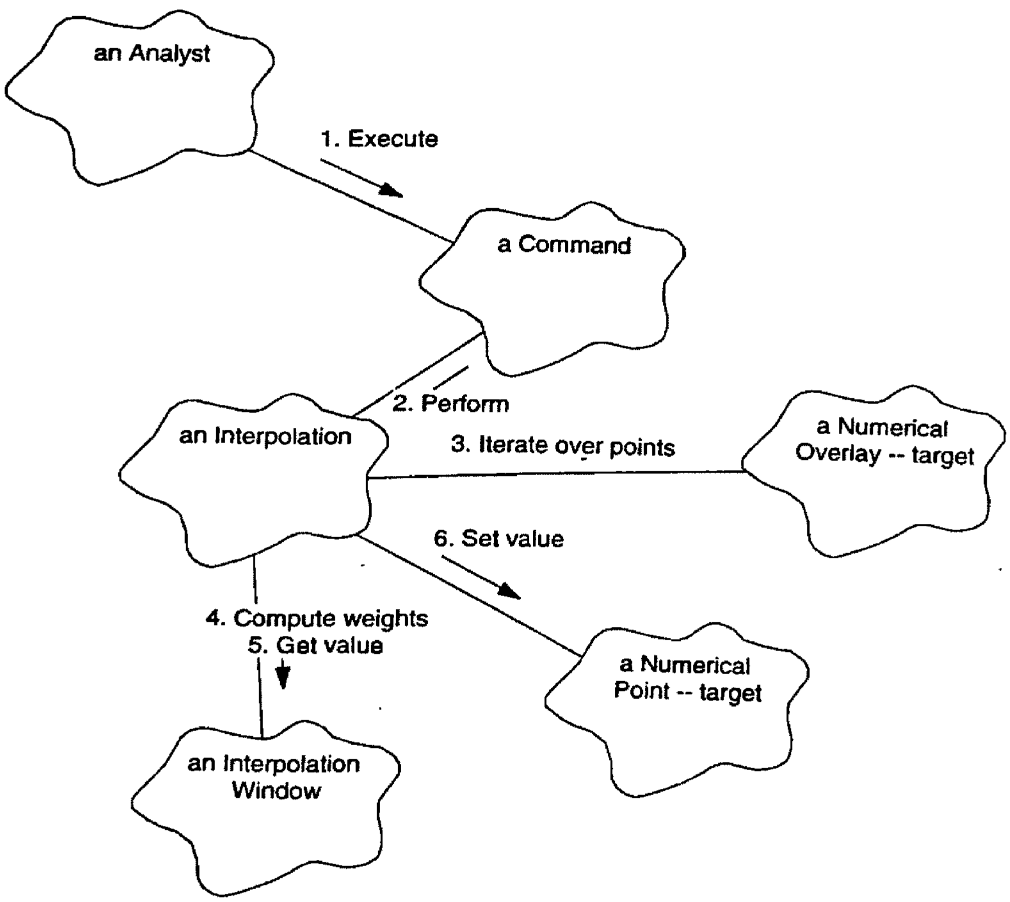

an Analyst
1. Execute
a Command
2. Perform
a Resample
3. Iterate over points
an Overlay
4. Resample
5. Get value
6. Set value
a Data Point
a Resample
Window

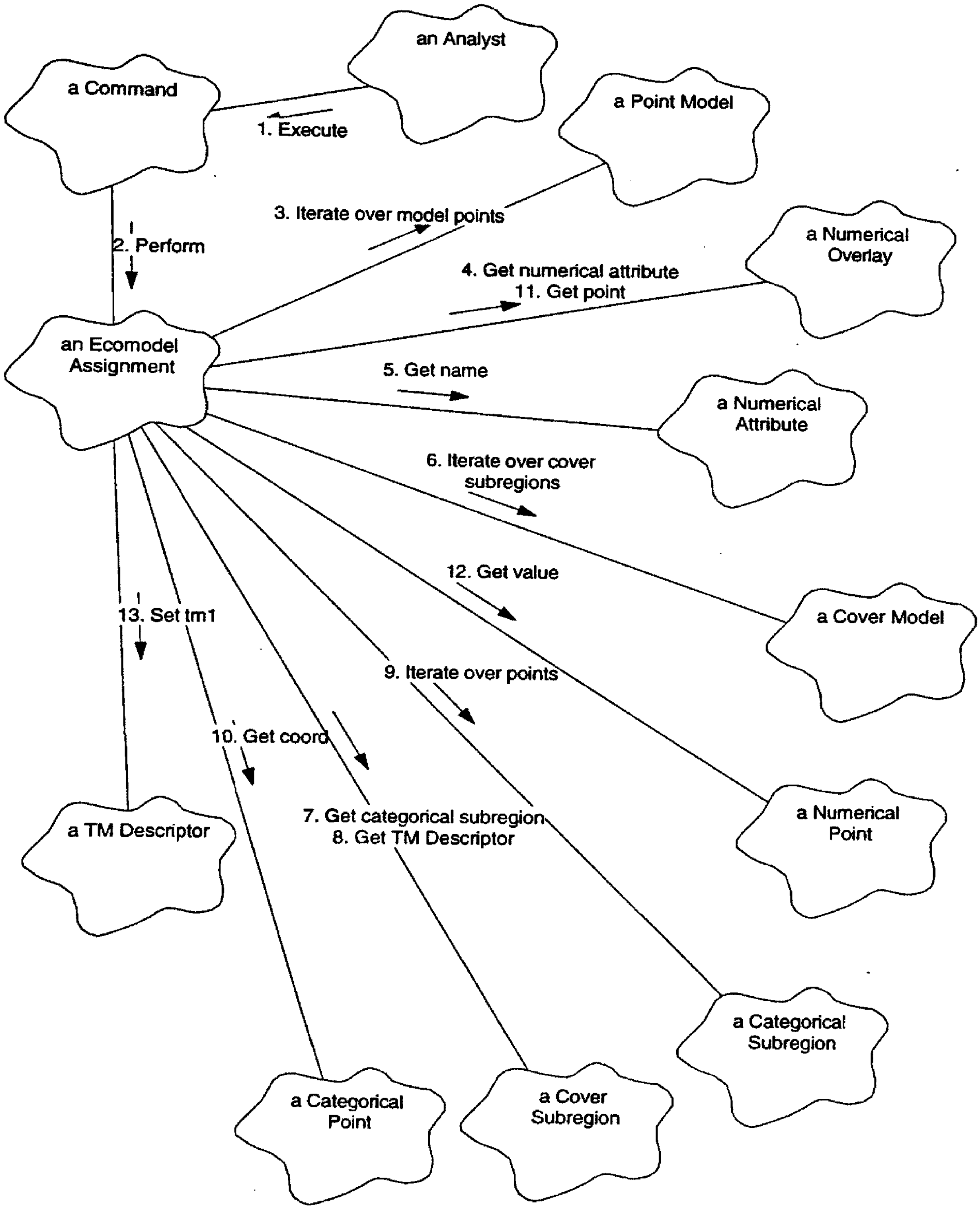

an Analyst
a GCP Set
a Command
1. Execute
3. Create
8. Add from coord
9. Add to coord
10. Compute Transformation
an Overlay -- master
2. Perform
4. Get point
a Registration
5. Get point
11. Iterate over points
an Overlay -- slave
6. Get coordinate
12. Get coord
7. Get coordinate
a Data Point -- master
13. Set x coord
14. Set y coord
a Data Point -- slave
a Data Point -- slave
a Coordinate


an Analyst
1. Execute
a Command
2. Perform
an Overlay
Algegbra
a Numerical
Overlay 1
Iterate over points
Iterate over points
a Numerical
Overlay 2
Get value
Add
a Numerical
Point 2
a Numerical
Point 1

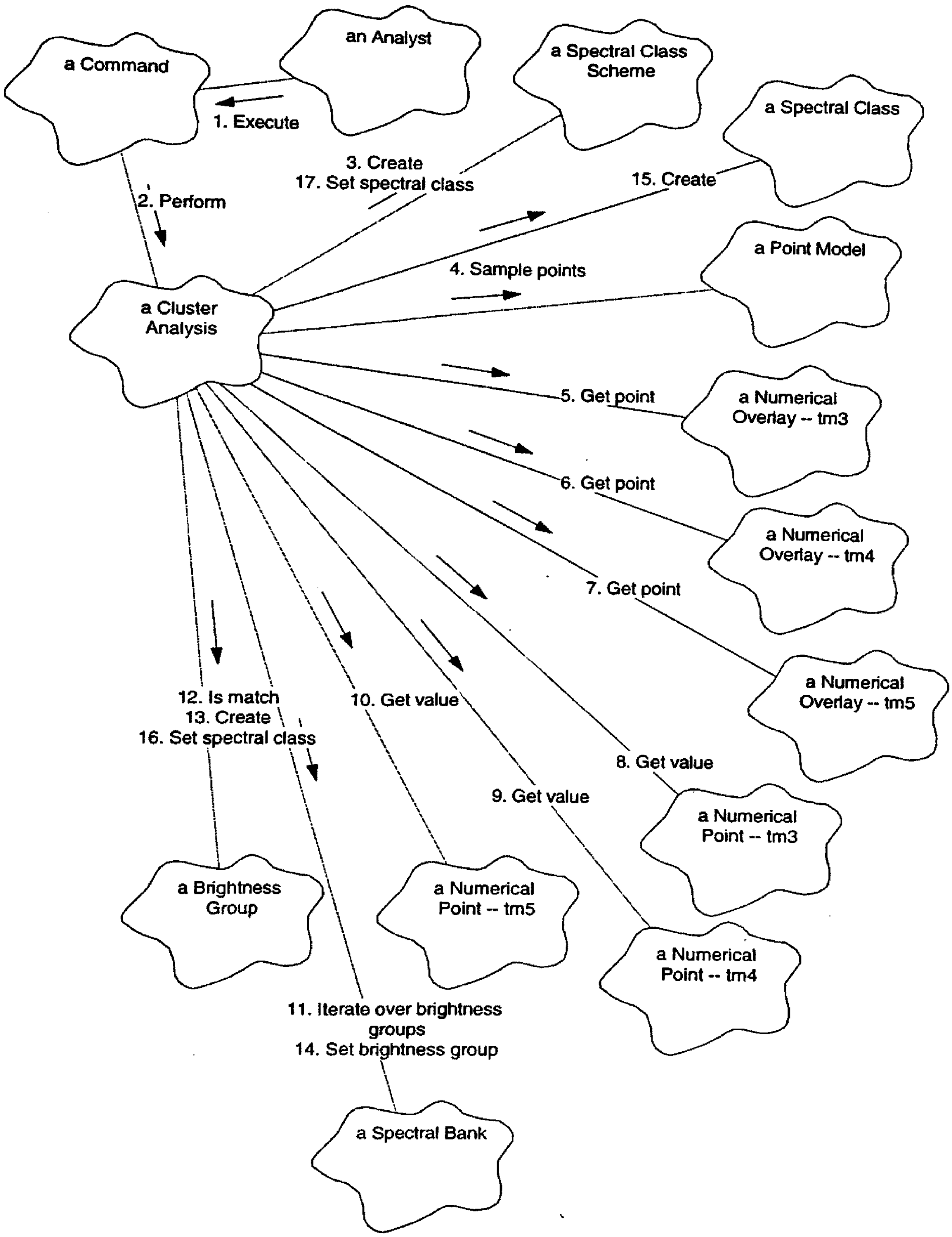

an Analyst
1. Execute
a Command
2. Perform
a Constrast
Stretch
3. Iterate over points
a Numerical
Overlay
4. Get value
5. Set value
a Numerical
Point

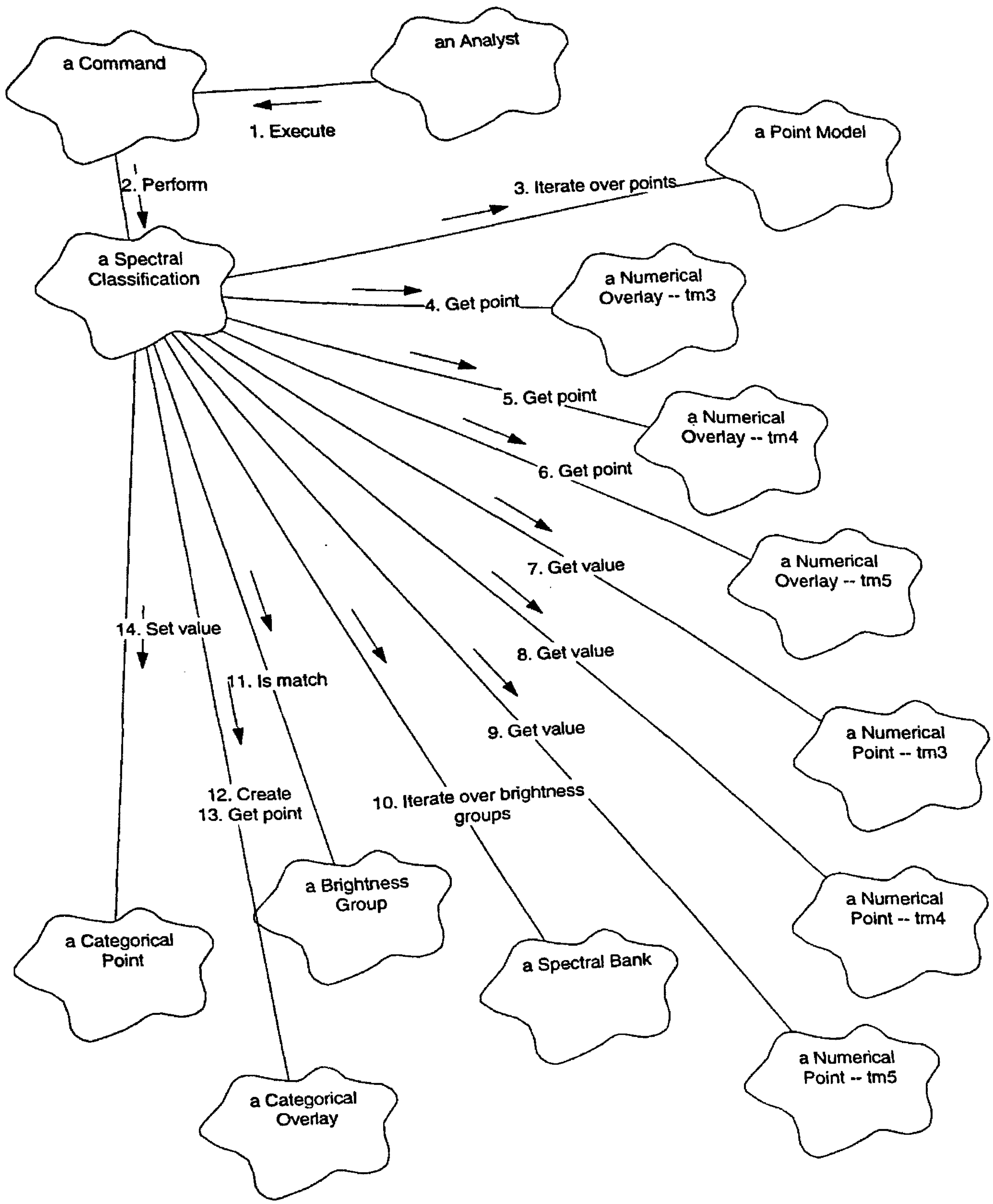

an Analyst
1. Execute
a Command
2. Perform
a Numerical
Overlay
3. Iterate over points
a Convolve
4. Compute mean
5. Get value
an Average
Window
6. Set value
a Numerical
Point

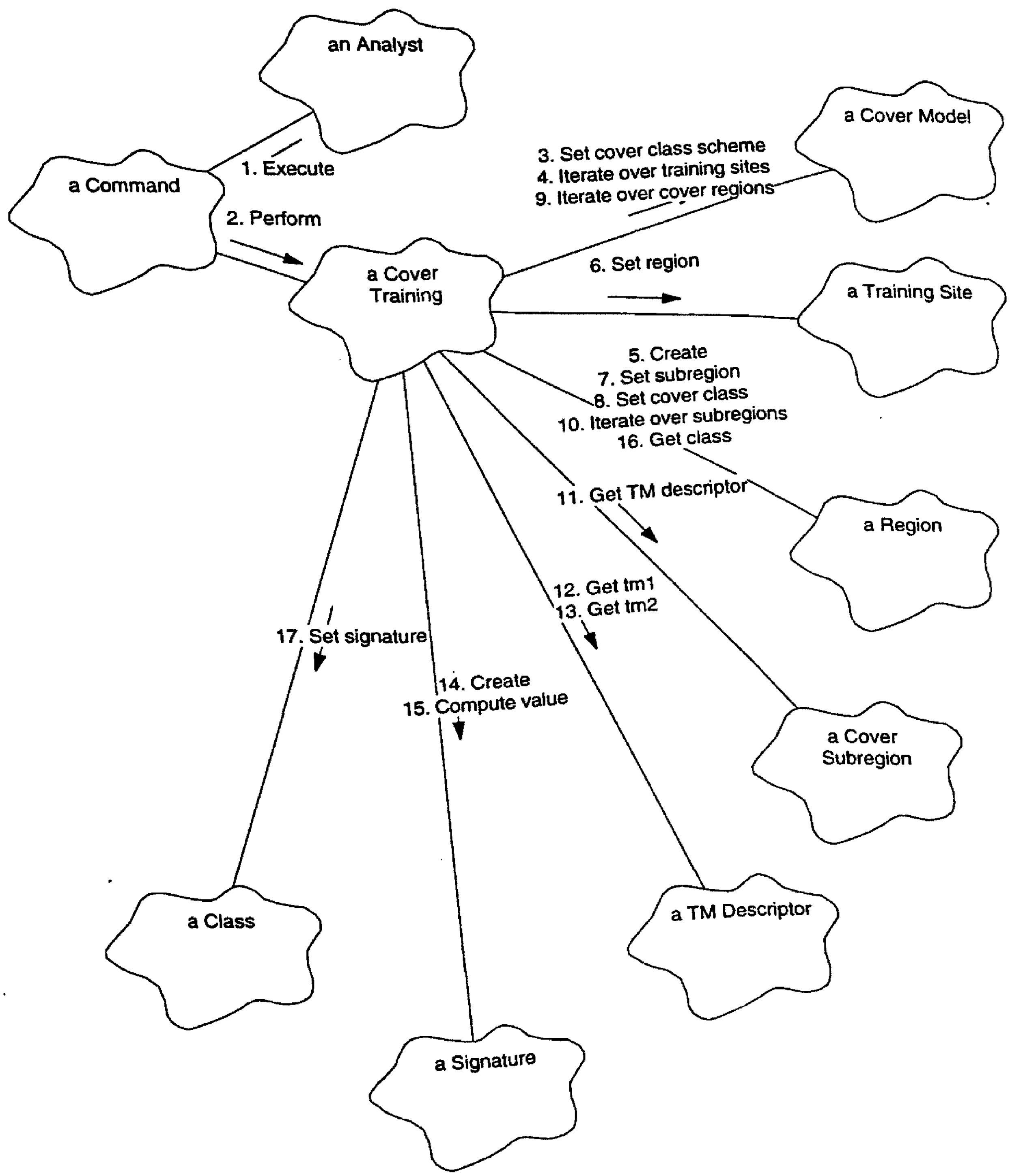

an Analyst
1. Execute
a Command
2. Perform
an Interpolation
3. Iterate over points
a Numerical
Overlay -- target
6. Set value
4. Compute weights
5. Get value
a Numerical
Point -- target
an Interpolation
Window

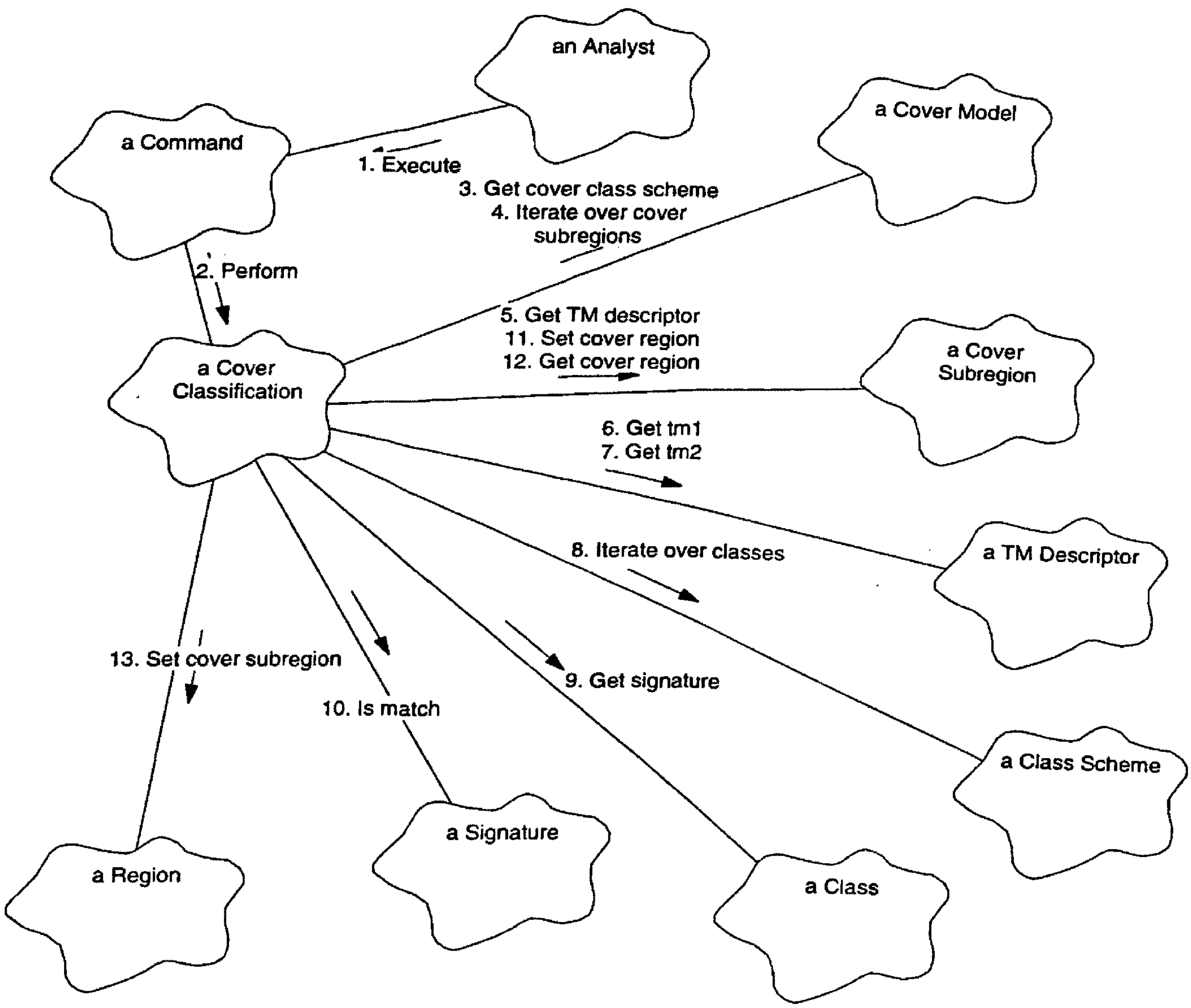


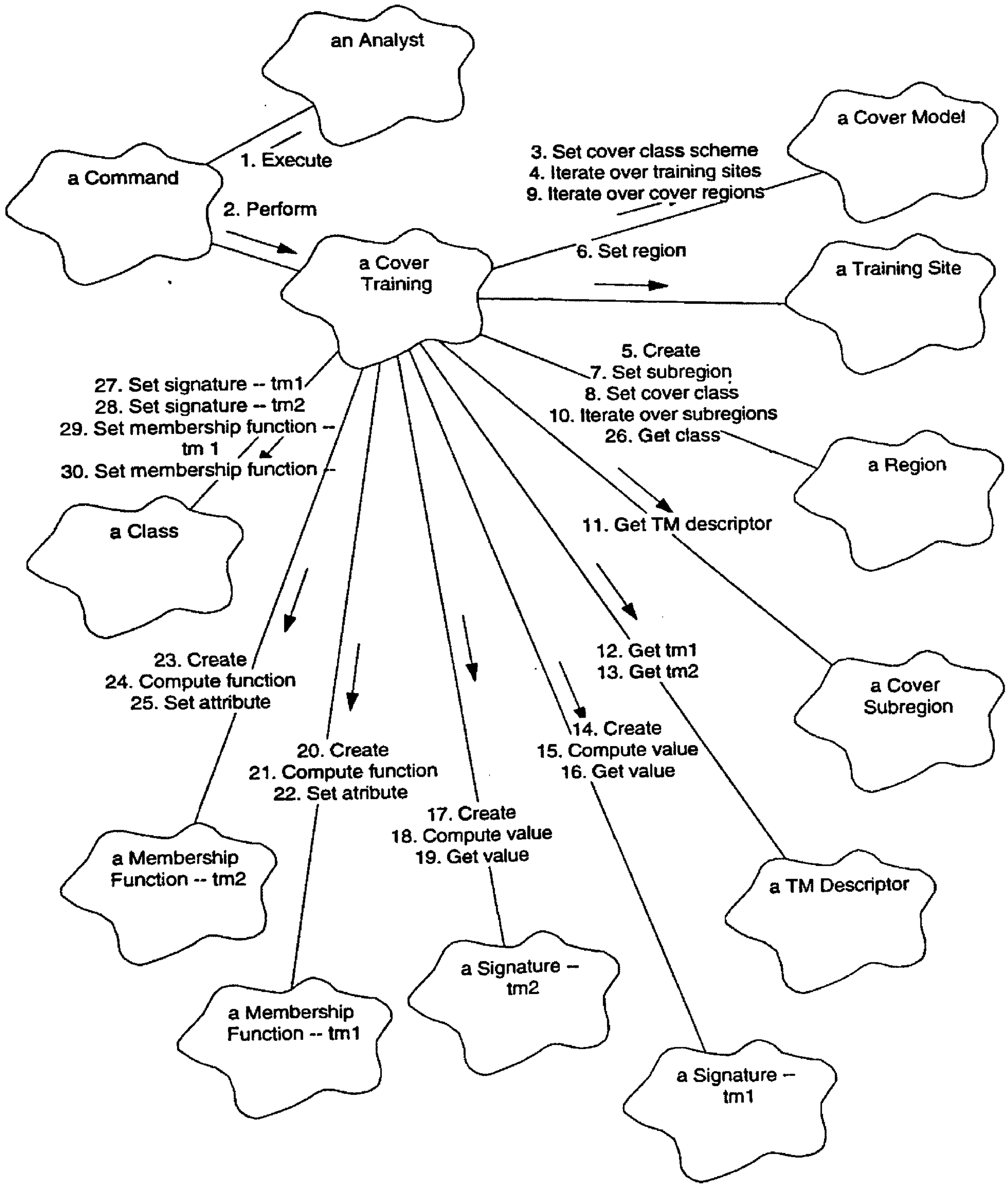
an Analyst
a Command
1. Execute
a Point Model
3. Iterate over model points
2. Perform
a Numerical Overlay
4. Get numerical attribute
11. Get point
an Ecomodel Assignment
5. Get name
a Numerical Attribute
6. Iterate over cover subregions
12. Get value
a Cover Model
13. Set tm1
9. Iterate over points
10. Get coord
7. Get categorical subregion
8. Get TM Descriptor
a TM Descriptor
a Numerical Point
a Categorical Subregion
a Categorical Point
a Cover Subregion


a Command
an Analyst
a Spectral Class Scheme
a Spectral Class
1. Execute
2. Perform
3. Create
17. Set spectral class
15. Create
a Point Model
4. Sample points
a Cluster Analysis
5. Get point
a Numerical Overlay -- tm3
6. Get point
a Numerical Overlay -- tm4
7. Get point
a Numerical Overlay -- tm5
12. Is match
13. Create
16. Set spectral class
10. Get value
8. Get value
9. Get value
a Numerical Point -- tm3
a Brightness Group
a Numerical Point -- tm5
a Numerical Point -- tm4
11. Iterate over brightness groups
14. Set brightness group
a Spectral Bank

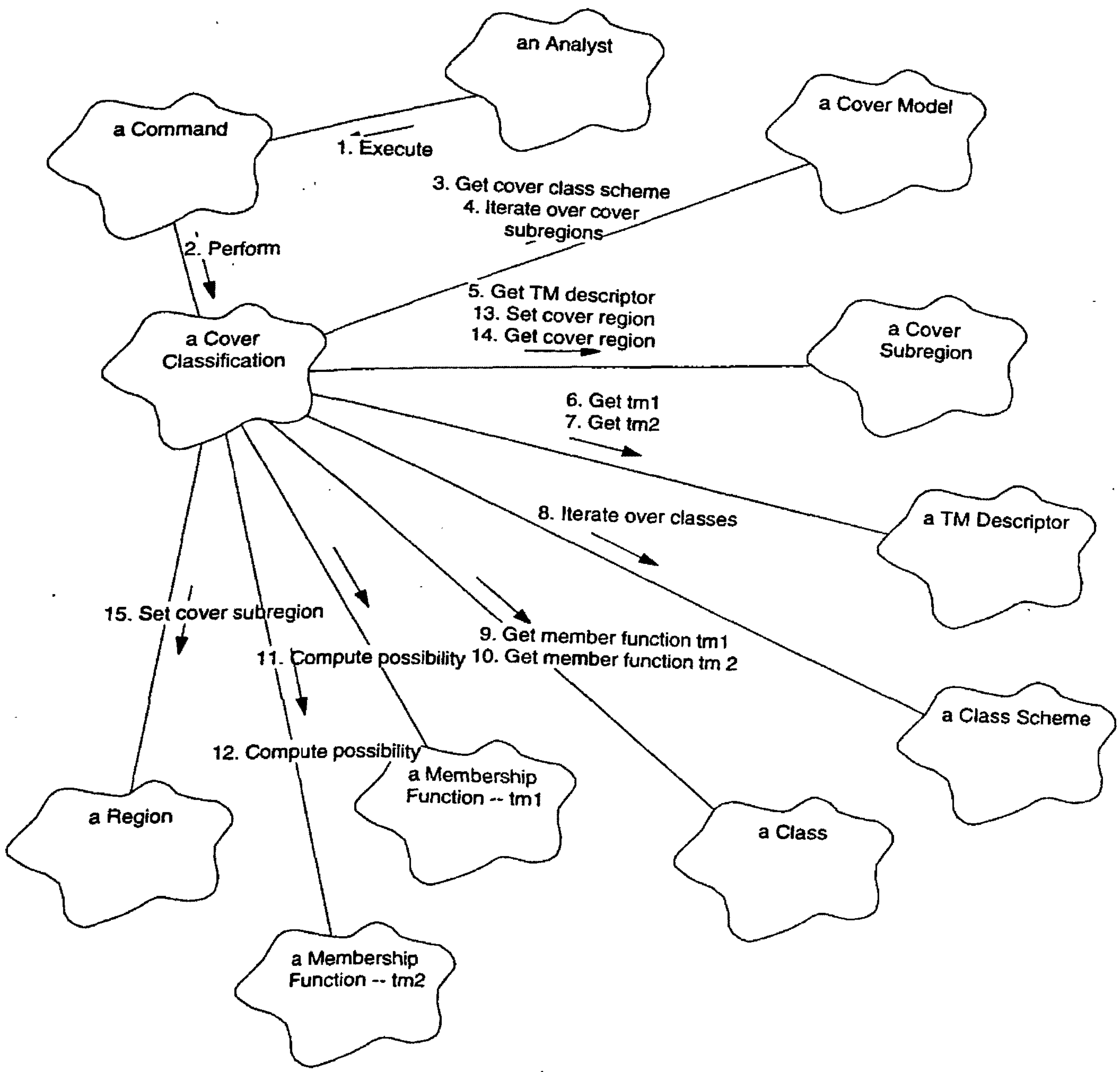

a Command
an Analyst
1. Execute
2. Perform
a Point Model
3. Iterate over points
a Spectral Classification
4. Get point
a Numerical Overlay -- tm3
5. Get point
a Numerical Overlay -- tm4
6. Get point
a Numerical Overlay -- tm5
7. Get value
8. Get value
9. Get value
14. Set value
11. Is match
12. Create
13. Get point
10. Iterate over brightness groups
a Numerical Point -- tm3
a Numerical Point -- tm4
a Numerical Point -- tm5
a Brightness Group
a Spectral Bank
a Categorical Point
a Categorical Overlay

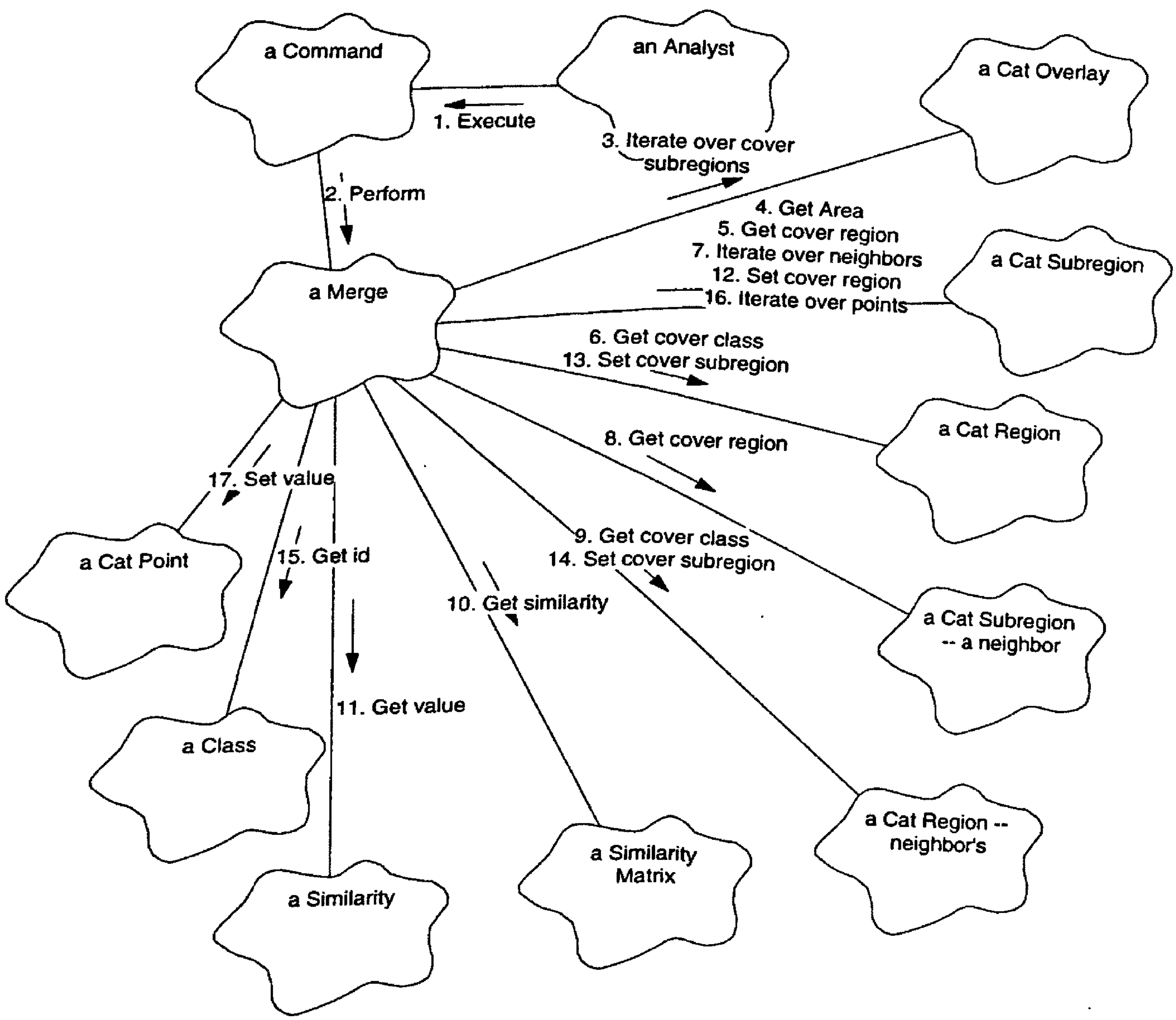

an Analyst
a Command
1. Execute
2. Perform
a Cover Model
3. Set cover class scheme
4. Iterate over training sites
9. Iterate over cover regions
a Cover Training
6. Set region
a Training Site
5. Create
7. Set subregion
8. Set cover class
10. Iterate over subregions
16. Get class
11. Get TM descriptor
a Region
12. Get tm1
13. Get tm2
17. Set signature
14. Create
15. Compute value
a Cover Subregion
a Class
a TM Descriptor
a Signature

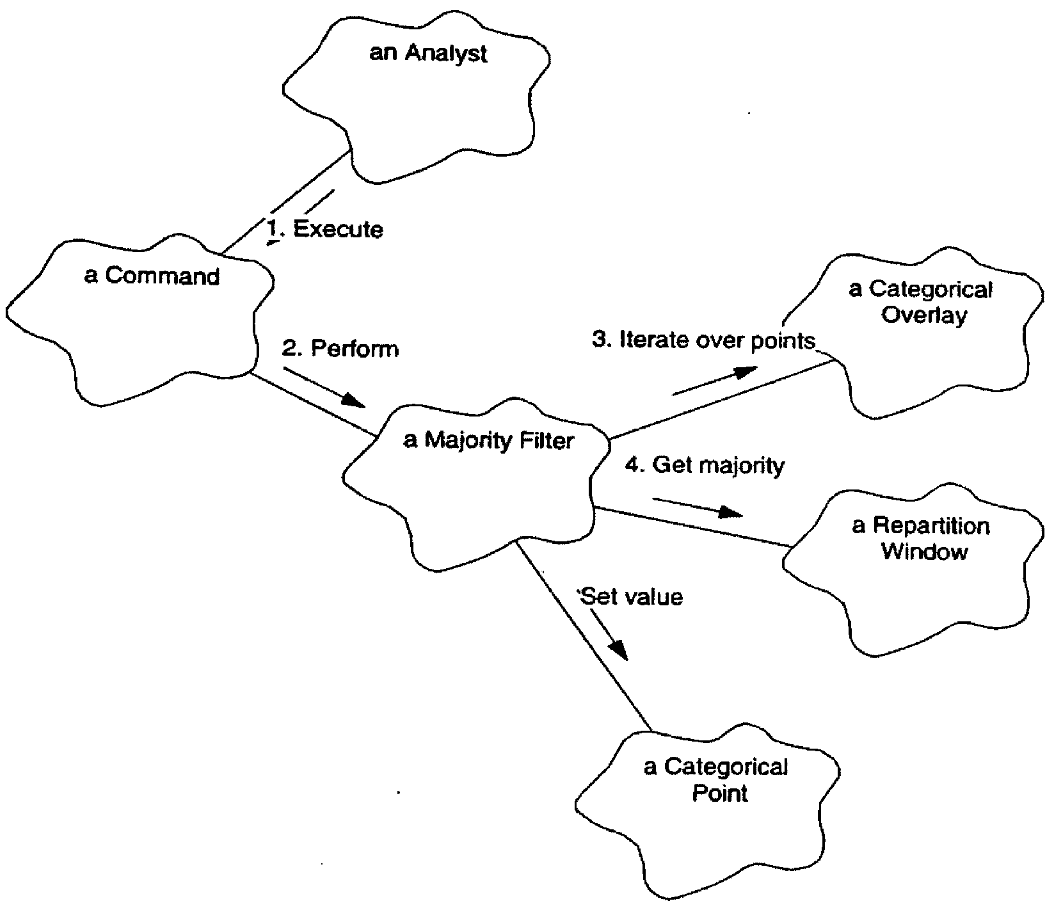

an Analyst
a Command
a Cover Model
1. Execute
3. Get cover class scheme
4. Iterate over cover
subregions
2. Perform
5. Get TM descriptor
11. Set cover region
12. Get cover region
a Cover
Subregion
a Cover
Classification
6. Get tm1
7. Get tm2
a TM Descriptor
8. Iterate over classes
13. Set cover subregion
9. Get signature
10. Is match
a Class Scheme
a Signature
a Region
a Class

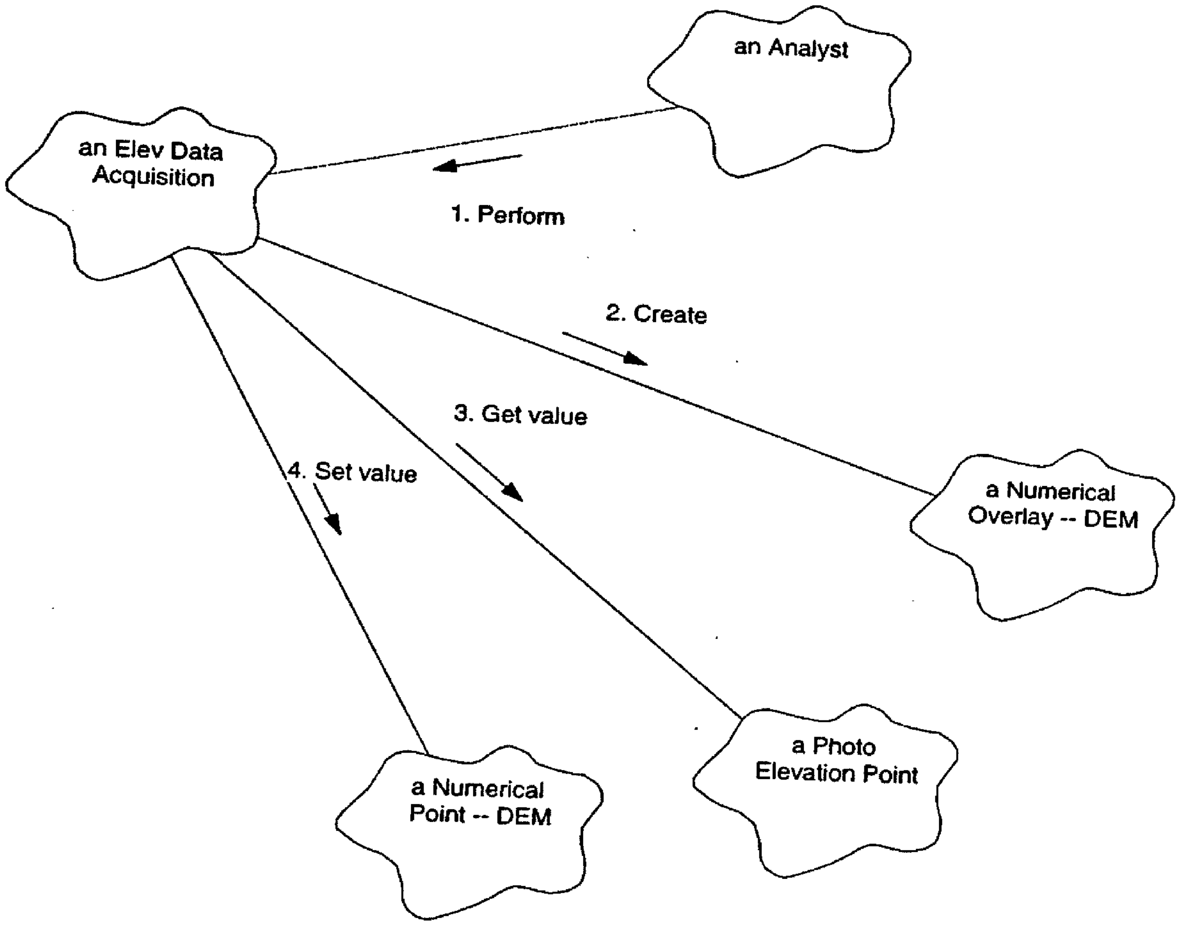


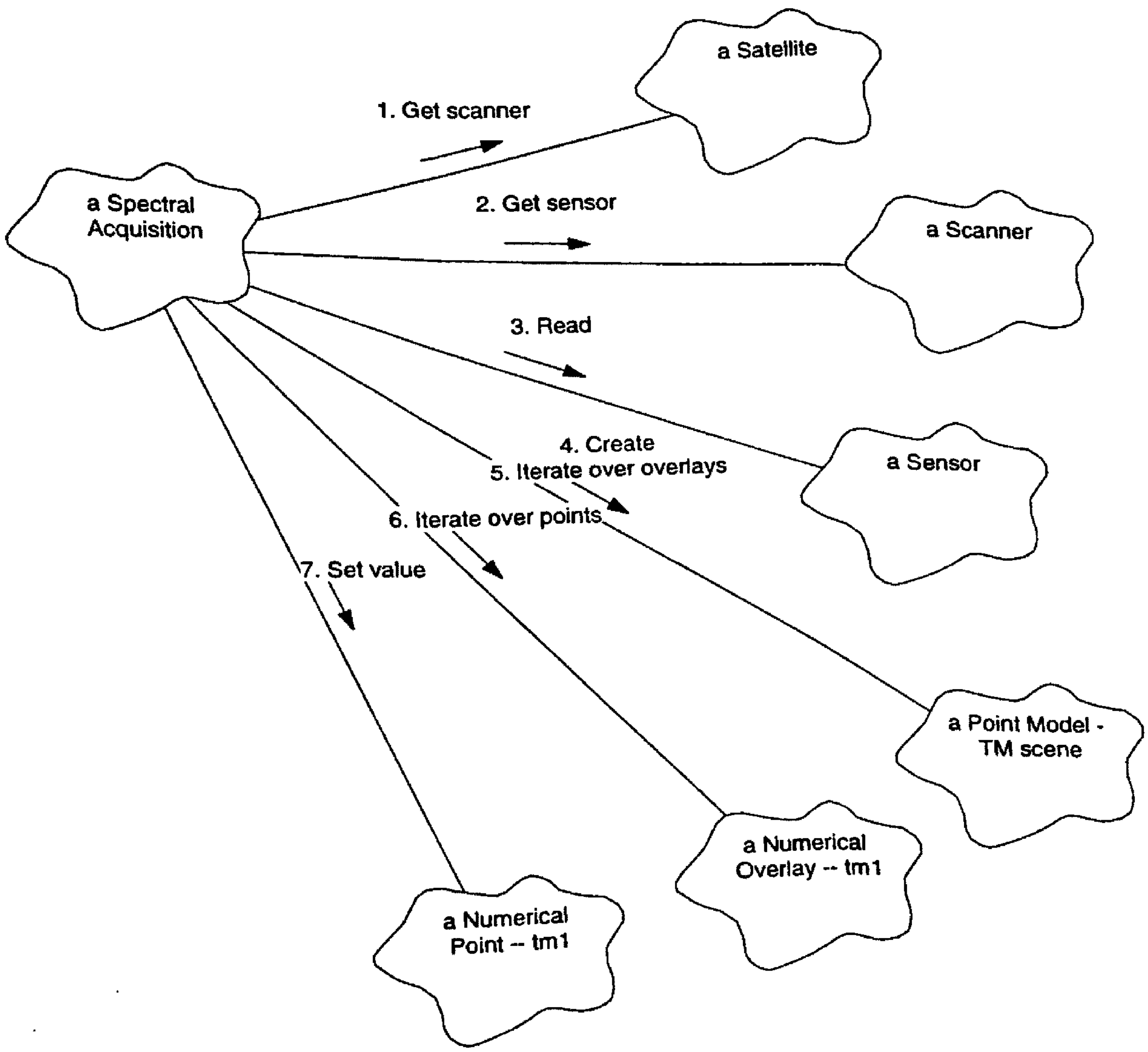
an Analyst
a Command
1. Execute
2. Perform
a Cover Training
a Cover Model
3. Set cover class scheme
4. Iterate over training sites
9. Iterate over cover regions
6. Set region
a Training Site
5. Create
7. Set subregion
8. Set cover class
10. Iterate over subregions
26. Get class
a Region
11. Get TM descriptor
a Cover Subregion
12. Get tm1
13. Get tm2
a TM Descriptor
14. Create
15. Compute value
16. Get value
a Signature -- tm1
17. Create
18. Compute value
19. Get value
a Signature -- tm2
20. Create
21. Compute function
22. Set atribute
a Membership Function -- tm1
23. Create
24. Compute function
25. Set attribute
a Membership Function -- tm2
27. Set signature -- tm1
28. Set signature -- tm2
29. Set membership function -- tm 1
30. Set membership function --
a Class

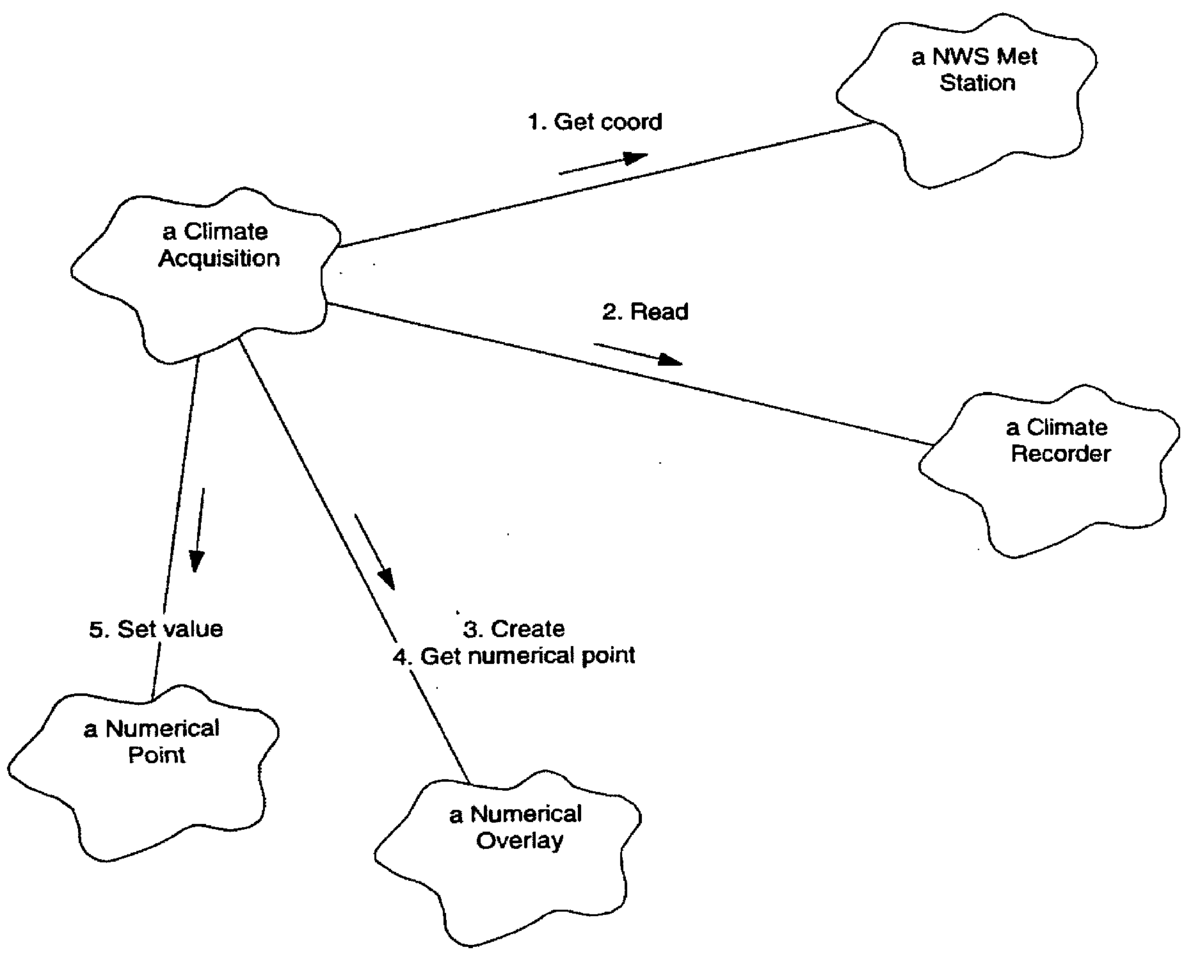

an Analyst
a Command
1. Execute
a Cover Model
3. Get cover class scheme
4. Iterate over cover subregions
2. Perform
5. Get TM descriptor
13. Set cover region
14. Get cover region
a Cover Classification
a Cover Subregion
6. Get tm1
7. Get tm2
a TM Descriptor
8. Iterate over classes
15. Set cover subregion
9. Get member function tm1
10. Get member function tm 2
11. Compute possibility
a Class Scheme
12. Compute possibility
a Membership Function -- tm1
a Region
a Class
a Membership Function -- tm2

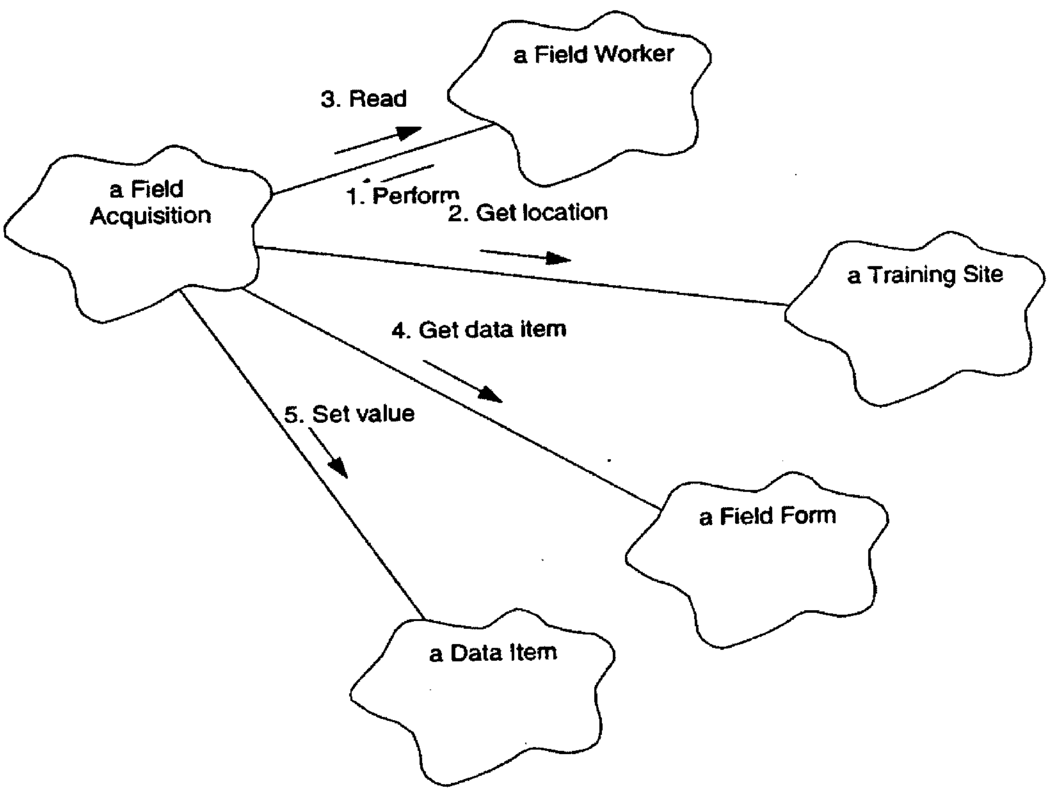

a Command
an Analyst
a Cat Overlay
1. Execute
3. Iterate over cover subregions
2. Perform
4. Get Area
5. Get cover region
7. Iterate over neighbors
12. Set cover region
16. Iterate over points
a Cat Subregion
a Merge
6. Get cover class
13. Set cover subregion
a Cat Region
8. Get cover region
17. Set value
a Cat Point
15. Get id
9. Get cover class
14. Set cover subregion
10. Get similarity
a Cat Subregion -- a neighbor
11. Get value
a Class
a Cat Region -- neighbor's
a Similarity Matrix
a Similarity


an Analyst
1. Execute
a Command
2. Perform
a Majority Filter
3. Iterate over points
a Categorical Overlay
4. Get majority
a Repartition Window
Set value
a Categorical Point


an Analyst
an Elev Data Acquisition
1. Perform
2. Create
3. Get value
4. Set value
a Numerical Overlay -- DEM
a Photo Elevation Point
a Numerical Point -- DEM


a Satellite
1. Get scanner
a Spectral Acquisition
2. Get sensor
a Scanner
3. Read
4. Create
5. Iterate over overlays
a Sensor
6. Iterate over points
7. Set value
a Point Model - TM scene
a Numerical Overlay -- tm1
a Numerical Point -- tm1


a NWS Met
Station
1. Get coord
a Climate
Acquisition
2. Read
a Climate
Recorder
5. Set value
3. Create
4. Get numerical point
a Numerical
Point
a Numerical
Overlay


a Field Worker
3. Read
a Field Acquisition
1. Perform
2. Get location
a Training Site
4. Get data item
5. Set value
a Field Form
a Data Item

BIBLIOGRAPHY

(Aronoff89) S. Aronoff, Geographic Information Systems : A Management Perspective, Ottawa, Canada : WDL Publications 1989.

(Band86) L. E. Band, "Topographic Partition of Watersheds with Digital Elevation Models", *Water Resources Research*, Vol. 22 pp. 15-24 1986.

(Bennett93) D. A. Bennett, M. P. Armstrong, F. Weirich, "An Object-Oriented Model Base Management System for Environmental Simulation", *Proceedings of the Second International Conference/Workshop on Geographic Information Systems and Environmental Modeling*, Breckenridge, CO. September 1993.

(Boehm88) B. W. Boehm, "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, Vol. 21 No. 5 pp. 61-72 1988.

(Booch91) G. Booch, Object Oriented Design with Applications, Redwood City, CA : The Benjamin/Cummings Publishing Company, Inc. 1991.

(Burrough86) P. A. Burrough, Principles of Geographical Information Systems for Land Resources Assessment, Oxford : Clarendon Press 1986.

(ERDAS91) ERDAS, ERDAS Field Guide. ERDAS, Inc 1991.

(Fedrizzi87) M. Fedrizzi, "Introduction to Fuzzy Sets and Possiblity Theory" in Optimization Models Using Fuzzy Sets and Possiblity Theory ed. J. Kacprzyk, S. A. Orlovski, Kluwer Academic Publishers 1987.

(Ford92) R. Ford, "Ecosystem Information System Development", National Science Foundation Grant Proposal 1992.

(Hungerford87) R. D. Hungerford, R. Nemani, S. W. Running, J. C. Coughlan, "MTCLIM: A Mountain Microclimate Simulation Model" USDA, INT-414 November 1989.

(Jennings93) M. D. Jennings,. "A Land Cover Classification for GAP Analysis Terrestrial Vegetation", draft 1993.

(Klanika93) K. Klanika, M. Mohite, A. D. Whittaker, M. L. Wolfe, "A Distributed Parameter, Object-Oriented Hydrologic Model", *Proceedings of the Second International Conference/Workshop on Geographic Information Systems and Environmental Modeling*, Breckenridge, CO. September 1993.

(Lammers90) R. B. Lammers, L. E. Band, "Automating Object Representation of Drainage Basins", *Computers and GeoSciences*, Vol. 16 pp. 787-810 1990.

(Lillesand79) T. M. Lillesand, R. W. Kiefer, Remote Sensing and Image Interpretation, John Wiley & Sons 1979

(Ma93) Z. Ma, R. L. Redmond, "Using Landsat TM Data and a GIS to Clasify and Map Existing Vegetation Across the State of Montana", draft 1993.

(Muehrcke78) P. C. Muehrcke, J. O. Muehrcke, Map Use : Reading, Analysis and Interpretations, Madison, WI : JP Publishing 1978.

(Nemani92) R. Nemani, S. W. Running, L. Band, D. Peterson, "Regional Hydro-Ecological Simulation System: An Illustration of the Integration of Ecosystem Models in a GIS" in Integrating GIS and Environmental Modeling ed. M. Goodchild, B. Banks, L. Steyvert, Oxford, London 1992.

(Raper93) J. Raper, D. Livingstone "High Level Coupling of GIS and Environmental Process Modeling", *Proceedings of the Second International Conference/Workshop on Geographic Information Systems and Environmental Modeling*, Breckenridge, CO. September 1993.

(Robinson53) H. Robinson, R. D. Sale, J. L. Morrison, P. C. Muehrcke, Elements of Cartography, New York, NY : John Wiley & Sons 1953.

(Scott93) M. Scott et, al., "GAP Analysis : A Geographical Approach to Protection of Biological Diversity", *Wildlife Monograph* No. 123 January 1993.

(Silvert93) W. Silvert, "Object-Oriented Ecosystem Modeling", Elsevier Inc. 1993.

(Unwin81) D. Unwin, Introductory Spatial Analysis, Methuen 1981.

(USGS86) U. S. Geological Survey, "Standards for Digital Elevation Models", Open File Report 86-004. Dept. of the Interior, U. S. Geological Survey, National Mapping Division 1986.

(White93) I. White, The Booch Method : A Case Study for Rational Rose for Windows, Santa Clara, CA : Rational 1993.

(White92) J. White, "RHESSys Runs on Watershed Scale Data", unpublished document 1992.

(Zadeh65) L.A. Zadeh, "Fuzzy Sets", *Inf. and Control* Vol. 8 pp. 338-353 1965.