# OSHADHI
# A Biodiversity Information System

A Thesis

Submitted For the Degree of

**Master of Engineering**

in the Department of Computer Science and Automation

by

**R. VIDYA**

Department of Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012

June 1995

# Contents

# Acknowledgements

I would like to thank my project guide, *Dr. Jayant Haritsa*, for his excellent guidance throughout the course of this project. I am grateful to him for his constant support and encouragement, without which this project would not have materialized. I would like to thank Prof. Madhav Gadgil, Dr. K. Gopinath and Dr. Swami Manohar for their valuable suggestions at various stages of this project.

I would also like to thank Mr. Janardhan Pillai, Mr. Utkarsh, Mr. Suresh, and Mr. Subash for providing me the specification and data required for the project, and Nirmala for providing me several references for the project. I would like to express my thanks to Mr. K. V. Subbarao for providing us with the name for the project (OSHADHI).

I take this opportunity to convey my special thanks to my friends Tulika, Vardhini, Surekha, Rupa, Joyjeet and Prakash, who have made my stay here a very pleasant and memorable one.

Finally, I would like to thank my parents and my sister for their love and encouragement.

# Abstract

The biodiversity conservation of the large number of plant species of India has become very essential with the rapid growth in the number of plant species which strengthen the genetic base of India due to their rich economic importance. Several organizations undertaking the measures to conserve the biodiversity such as identification of species and monitoring the climatic conditions, have realized the need for efficient management of the large amount of biodiversity information which is currently available in various forms. Traditional database systems (built on relational, hierarchical and network models) which are widely used for commercial applications such as banking, fail to meet the modeling and processing requirements of the biodiversity data. Recent developments in the database technology have resulted in new approaches such as the object oriented databases, which cater to the requirements of the biodiversity application.

In this project, we have addressed the problem of biodiversity data management, and designed and developed OSHADHI, an object oriented biodiversity information system. OSHADHI incorporates a variety of biodiversity information such as the information about species, their characteristics, habitats and information about the various geographical regions, the climatic conditions, etc. Apart from maintaining the database, OSHADHI also handles various kinds of queries such as identification, viewing the climatic conditions of geographic regions, etc. OSHADHI is built on top of EXODUS, an extensible object oriented database management system. OSHADHI also provides a graphical user interface, built using X and Motif, for data entry and querying.

# Chapter 1

# Introduction

## 1.1 Motivation

The genetic base of India is represented by a large number of plant species which include the “local” species (species that grow naturally in the country) and the “foreign” species (species that have been brought in from various parts of the world). Over the past few years, several new local species have been identified and there has also been a rapid growth in the number of foreign species. The rich economic potential and uses of these plant species make it essential for their *biodiversity* to be conserved. Several measures for biodiversity conservation have been suggested by researchers and scientists in the area of environmental sciences. These measures include identification of species, monitoring the status of endangered species and monitoring the climatic conditions of various geographic locations [Decc95].

Several organizations all over the country are working towards the implementation of these measures in order to conserve the biodiversity of the species. However, one of the major hurdles faced by the scientists and researches working in this area is the management of the large amount of biodiversity data. The biodiversity data includes a variety of information such as information about the species and their characteristics (such as flower, fruit, stem, habitat etc.), information about endangered species and species of medicinal value, climatic conditions such as temperature, soils and land use, distribution of various species in different geographic locations, the details about botanical gardens and herbariums [1], references (bibliography) for various species and geographic regions, details about the experts in the related fields, etc. Biodiversity data may therefore be classified into four inter-related groups, namely, the *taxonomy* data (which includes details of all the species, their classifications and characteristics, etc.), the *geographic* data (which includes the details of the geographic locations, their environmental conditions, etc.), the *bibliographic* data (which includes the details of the journals, articles and textbooks in the field of biodiversity) and the *experts* data. The organization of the biodiversity data is shown in Figure 1.1.

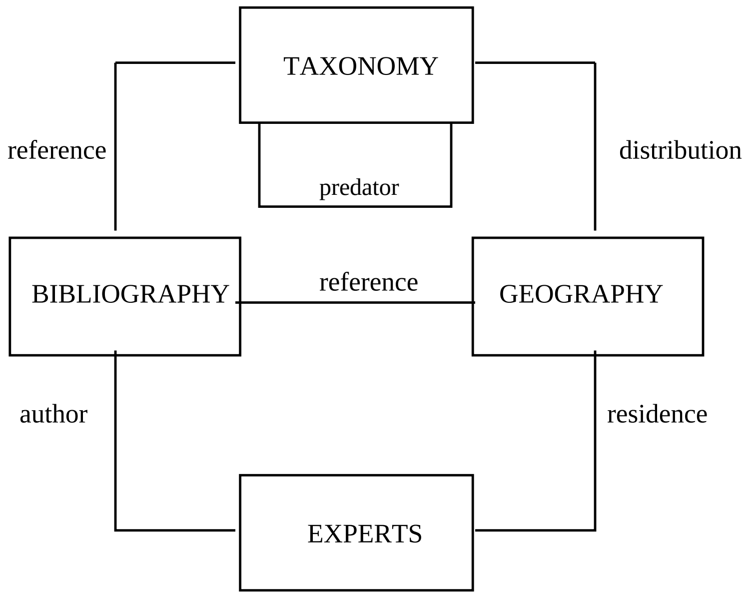



Figure 1.1: The biodiversity data organization

The biodiversity information mentioned above is currently available in various forms and there are several sources for this information. For example, a large amount of information is available in the form of textbooks and herbarium sheets [2] , and in various places like colleges, universities and botanical gardens. However, maintaining the information in these forms makes its management extremely slow and inefficient. Hence, what

[1]Places where plant specimens are stored
[2]These are sheets that contain a plant specimen and its details.

is required is a system to manage the biodiversity data in an efficient way and to provide quick retrieval facilities. Apart from the need to aid the process of biodiversity conservation, maintenance of information such as whether a species is a local or foreign species is essential in the light of the *General Agreement on Trade and Tariffs (GATT)*.

In this project, we have developed an information system for biodiversity data. We have focussed primarily on the biodiversity information of the plants of Western Ghats with the aim of meeting the requirements of the researchers and scientists at the *Center for Ecological Sciences*, Indian Institute of Science, Bangalore.

## 1.2 Related Projects

Several projects have been undertaken by various organizations with the aim of tackling some subsections of the biodiversity data management problem. For example, *INMED PLAN (Indian Medicinal Plants National Network of Distributed Database)* is a database built for the medicinal plants of India. The other related projects are the *Mangrove Ecosystems Information Service*, the *Indo-based PGR project*, the *ERIN (Environmental Resources Information Network)*, the *PLATO (PLAnt TOxins)-UK* project [Poun95] and the Australian *INTKEY* project.

Most of these projects have been built either using commercial database systems such as ORACLE [Orac88], or using other technologies like Hypermedia, etc. Some projects like the PLATO-UK and INTKEY use a database in the DELTA (DEscriptive Language for TAxonomy) format [Poun95].

Database systems built on relational, network and hierarchical models perform very well when used for commercial applications such as banking and inventory control. But they are not well suited for applications such as GIS (Geographic Information Systems), CAD (Computer Aided Design) and biodiversity database systems. This is because these applications differ significantly from the commercial applications both in their modeling as well as processing requirements. The following points highlight some of the requirements of the biodiversity application which are not satisfied by the commercial database systems:

- Most of the objects dealt with in the biodiversity application are complex. For example, a country contains several states, a state is made up of districts which in turn is composed of several cities. Mapping such complex objects on to simple abstractions like tables (as in the relational model) would require breaking the complex objects into simpler atomic objects. This process can be very cumbersome and may lead to a degradation in the performance of the system [Gunt94].

- The biodiversity data also includes spatial objects like the maps of various geographic locations. Conventional systems do not provide adequate support for managing this type of data. For example, we might require a different indexing mechanism to provide fast access to spatial data [Gunt94].

- The activities such as monitoring of climatic conditions in a location require *Versioning*, i.e., maintenance of the variations in the state of an object over a period of time. Conventional systems do not provide adequate support for maintaining different versions of an object [Bert91].

- Commercial systems do not provide facility for easy schema evolution [Jose91] which is essential in the biodiversity application where new object types could evolve with time.

- The types of queries in the biodiversity application are significantly different from those in commercial applications. For example, a typical query in the biodiversity application would be to locate the state that contains a location (the location is specified by its latitude and longitude). This query would require determining if the point is inside the polygon that approximates the state boundary. This query cannot be handled by commercial systems. Similarly, commercial systems cannot handle the other spatial queries that are common in the biodiversity database.

From the above discussion it is clear that commercial database systems are inadequate for the biodiversity information system. Among the other techniques used in existing projects, DELTA is a free format text which is flexible. However, its application is limited to taxonomy and identification.

## 1.3 The OSHADHI System

In order to meet the requirements of biodiversity database systems, the use of a different technology appears to be necessary. Recent research in the field of databases has resulted in the emergence of new approaches such as the *Object Oriented Databases* which support the desired features to cater to the requirements of the advanced applications. With this motivation, we have adopted an object oriented approach to build a biodiversity information system for the plants of Western Ghats. The users of the system would mainly be the botanists and researchers working in the area of environmental sciences.

OSHADHI[3] is an object oriented biodiversity information system that has been built with an *Extensible Object Oriented Database Management System* [DBE87], **EXODUS** [Care86a] as the back-end. The OSHADHI architecture is composed of three major components, namely, the *Exodus Storage Manager*, the *OSHADHI Database Server* and the *Graphical User Interface.*

The exodus storage manager forms the lowest layer of OSHADHI. The storage manager serves requests from the database server such as creating objects, beginning transactions, etc. The database server built on top of the storage manager, implements the object model of OSHADHI and operates on the database by making calls to the exodus storage manager. The database server is also responsible for parsing and processing the queries that it receives from the user. The user interacts with the database server through the graphical user interface. The OSHADHI database server is written using the E language [Rich89a]. E is a database programming language that is provided by EXODUS. The graphical user interface is built using X and Motif [Youn90]. The architecture of OSHADHI is shown in Figure 1.3.

The OSHADHI database includes the taxonomy details of various species, their characteristics, classifications and textual descriptions, the details of various geographic locations, their climatic conditions, etc. OSHADHI provides data entry facility through built-in forms and allows form-based querying. The OSHADHI server handles queries

[3] 'plant kingdom' in sankrit

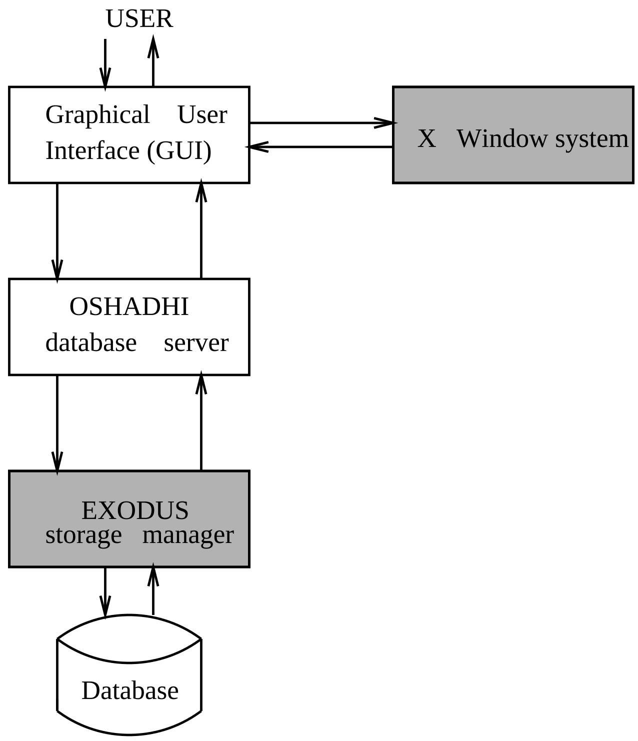



Figure 1.2: The architecture of OSHADHI

such as the identification of species given the flower, fruit, leaf, stem or seed characters (usually one or more of these characters are specified for identification), viewing the various characters of a species, listing species with specific characters in a geographic location, viewing maps of various geographic locations etc. OSHADHI uses techniques such as $B^+$tree indexing [Baye72, Come79], Quadtree indexing [Same89] and pointer intersection method [Kort91], to provide fast access to data.

In the current status of implementation, the OSHADHI database consists primarily of taxonomy data and geographical data. However, since the data model has been developed for the bibliographic and experts data also, we expect the addition of these two types of data to OSHADHI to be a simple task.

## 1.4 Organization of the Report

The rest of the report is organized as follows: In Chapter 2, we discuss the basic features of object oriented databases and how the object oriented approach applies to the problem of biodiversity data management. We also describe briefly the stages involved in the development process of an object oriented database. In Chapter 3, the problem is defined formally and the data model of OSHADHI is discussed in detail. Chapter 4 discusses the design of the database. The system architecture of OSHADHI is described in Chapter 5. Chapter 6 covers the transaction model and the implementation details of OSHADHI. In Chapter 7, we highlight some extensions that can be made to OSHADHI and conclude in Chapter 8 by presenting a summary of the features of OSHADHI.

# Chapter 2

# The Object Model

A real world problem could be visualized as a collection of related objects with certain constraints on their relationships. For example, in the biodiversity data, a country contains several states and this could be viewed as a relationship between a country object and a state object with the constraint that a state can belong only to one country. A *Data Model* is a logical organization of real world objects, their constraints and relationships between these objects. A *Database System* implements a Data Model. A *Database Programming Language* could be defined as a syntax for data modeling and manipulation [Jose91]. In this chapter, we study the requirements of the biodiversity information system in detail and discuss the features of object oriented data model which make it suitable for the biodiversity application.

## 2.1 Features of the Biodiversity Database

As mentioned in the previous chapter, the biodiversity application requires management of taxonomy data, geographic data, bibliographic data and experts data. A study of the data provided by the *Centre for Ecological Sciences* reveals the following prominent features:

- *Model*

There is an inherent nesting in most of the objects. For example, the taxonomy of species results in a four level hierarchy depending on their characteristics. Similarly, a country is composed of several states. A state is made up of districts, which in turn is composed of cities. Similar hierarchy is also exhibited by the spatial data (polygons, lines, etc.). The species show extensive sharing of data (characteristics) among them and the characters themselves exhibit a hierarchy. For example, a flower is composed of parts like petals, sepals, stamen, etc. each of which is a characteristic that could be associated with any species. A species could be found in one or more geographic locations (this is an example of a relationship between the taxonomy data and the geographic data) and a geographic location could be a city, country, state, district, forest or a bio-geographical province.

- *Types of operations*

  The types of operations done on the biodiversity database are also different from those in a commercial application. While typical operations in commercial databases would be selecting groups of data, updating and storing them back, most queries in the biodiversity database would require picking a single object and navigating through a web of related objects, the path originating from the selected object. For example, displaying the map of a country would require traversing the following objects: the country object, the spatial data (or map) associated with the country (the boundary of the country is usually approximated by a polygon), the lines constituting the polygon and finally, the end points of each of the lines. Similarly, another useful query on geographical data would be to locate a state which contains a location (specified by its latitude and longitude). This requires searching through the polygons that represent states and applying the *point in polygon* [Hear90] algorithm to it. Other spatial queries like identifying regions that satisfy certain conditions would require overlapping two or more polygons [Gunt94].

- *Versions*

Another important requirement of the application is the need for *Versioning* or maintenance of the variations in the state of an object over a certain period. Versioning would be required in order to monitor the status of endangered species and to monitor the climatic conditions in geographic locations of interest.

- *Large Volume of Data*

  The amount of biodiversity data that need to be managed is very large. The taxonomy information includes several thousand species, each having a large number of characteristics. The spatial data which includes the points, polygons and lines that are used to represent the map of a country, also accounts for the large size of the data.

- *Infrequent Updates*

  The biodiversity database is characterized by infrequent updates.

- *Schema Evolution*

  In the biodiversity application, new object types may evolve with time. For example, new types of characteristics may have to be added to the schema.

## 2.2 Drawbacks of Traditional Database Systems

Having seen the features of the biodiversity databases, we will now discuss why traditional systems are unsuitable for developing a biodiversity database [Jose91].

- *Lack of rich modeling power*

  The biodiversity data is represented by a rich model. However, conventional database systems are built on simple models and a complex problem cannot be easily modeled using simple abstractions (for example, tables). This mapping would require the complex objects to be broken into objects of atomic types thereby flattening the natural hierarchy and this would have a negative impact on the performance of the system [Gunt94].

- *Lack of adequate querying facility*

  Conventional systems do not provide adequate facilities to process queries such as those on spatial data. Also, conventional systems do not provide indexes for the spatial data such as R-trees [Gutt84] or Quadtrees [Same89].

- *Performance problems*

  Apart from the lack of rich data modeling power, traditional systems might show poor performance due to the type of operations that are performed on a biodiversity database [Jose91].

  - Let us consider the process of displaying the map of a country. We have to read the object that represents the country and seek the spatial data attribute of the object. If a relational database system is used, this operation would correspond to selecting the country tuple and projecting the spatial data attribute of the tuple, where the select and project are executed by making procedure calls to the DBMS. The overhead due to the procedure calls is not too much if the operations are done on a group of objects. However, it is considerable when done on a single object, as in the case of displaying the map. The overhead could be avoided if seeking an attribute of a single object could be done as in programming languages, i.e., by using the offset of the attribute.
  - When complex objects are broken into atomic objects, the number of tables (assuming that a relational database system is being used) in the database would be large, and the process of *Normalization* [Kort91] would result in additional tables and keys. The translation from keys to address space would place additional burden on the system.
  - In a relational database system, every tuple *must* belong to a table and for every table the system needs to maintain certain information. Due to this constraint, objects such as maps of countries, drawings of species and the text associated with species would also be tuples in a relation. This is clearly an

overhead, since these objects are never going to be accessed as elements of a set.

- Conventional systems also fail to provide adequate support for features like versioning and schema evolution which are required by the application.

Recent research in the area of database systems has resulted in emergence of a new type of modeling, the *object oriented modeling*, which caters to the requirements of biodiversity application. Before discussing how the object oriented approach suits the biodiversity application, we briefly describe some of the basic concepts of object oriented model for databases.

## 2.3 The Object Oriented Model

Many of the concepts in the object oriented model for databases are similar to those in object oriented programming languages. The object model is characterized by the concepts of *objects*, *identity*, *classes*, *aggregation*, *inheritance*, *class hierarchy*, *encapsulation* and *overloading*. We now describe each of these concepts [Kim89].

- *Object and Object Identity*

  In the object oriented model, every entity is treated as an *Object* and every object is characterized by its *Identity*, *State* and *Behavior*. The state of an object is determined by the values of the attributes, the behavior is represented by the set of *Methods* that can be used to interact with the object and identity is a feature that distinguishes two objects having the same state and behavior. Each object is assigned an unique object identifier(OID) which does not change with time.

- *Class*

  A *Class* is an user-defined type that represents all objects sharing the same attributes and methods . An object could be viewed as an **INSTANCE-OF** the class to which it belongs. For example, *Country* is a class and *India* is an object that belongs to the class.

- *Aggregation*

  Object oriented model supports construction of complex objects or objects which *contain* other objects. In terms of the class definition, this means that the class of a complex object may have attributes that are of composite type. This feature captures the **PART-OF** relationship between objects. For example, a country object contains many state objects or a state is **PART-OF** a country. This type of nesting gives rise to a hierarchy of objects, where an object in one level contains the objects in the next level. This hierarchy is termed as the *aggregation* or the *containment hierarchy*.

- *Inheritance and Class Hierarchy*

  The inheritance feature of the object model distinguishes it from other models such as the hierarchical model. Object oriented model allows new classes to be *derived* from existing classes. The derived classes inherit the attributes and methods of their parent class. The derived classes can also have new methods or refine the methods of their parent class. Inheritance in the object model, captures the **IS-A** relationship between objects. Classes may form a hierarchy in which, objects in a given class automatically become members of all its superclasses. This hierarchy is termed as the *Class hierarchy*. For example, a polygon is a spatial data . Therefore, the class representing a polygon is derived from the class representing spatial objects.

- *Encapsulation and Overloading*

  In an object oriented approach, it is possible to define classes so that the state of an object belonging to the class can be accessed or modified only through its methods. This is referred to as *Encapsulation*. Encapsulation causes *localization* of code that eases the task of debugging. Also, application programming becomes easier, since the user using a predefined class need to examine only the definitions of the methods(functions) in order to use it [Stro91].

  *Overloading* refers to attaching different implementations to the same function name. The implementation that is to be used is determined by the object for which the method was invoked. For example, *Display* is a method that has different implementations for different spatial objects like point, line and polygon.

## 2.4 Object Oriented Database Management Systems

Object oriented database management systems (OODBMS) are database systems built on the object model. Therefore, OODBMS combine the features of database systems and those of the object model. In this section, we highlight the features of OODBMS that make them suitable for the biodiversity application.

- *Rich Modeling Power*

  By the virtue of the object model on which OODBMS are built, OODBMS provide a rich data modeling facility. Hence any complex problem could be easily modeled. For example, objects such as countries (a country contains several states, a state in turn, contains several districts which are made up of cities) can be modeled easily using aggregation. Features like inheritance allow the model to be organized and simplified [Rumb91].

- *Persistent Programming*

  The notions of classes, attributes, class hierarchy, inheritance, aggregation, encapsulation and overloading are the same as those in object oriented programming languages. Hence there is no *Impedance Mismatch* [Bert91] between the programming languages and the object oriented model. Most OODBMS (object oriented database systems) extend object oriented programming languages to have database constructs in addition to features mentioned earlier. For instance, database programming languages built on the object model [Maie87], provide the capability of declaring some objects as *persistent*. The database is thus implemented as a collection of *persistent objects*. Relationships between objects are treated as inter-object references and applications can navigate through the web of objects using these inter-object references, just as they would in the case of in-memory objects. This is referred to as *Persistent Programming* [Atki87]. This feature allows an attribute of an object to be accessed in the same way as a structure field is accessed in programming languages. Also, the use of inter-object references eliminates the need for keys to access a non-atomic attribute of an object.

- *Complex queries*

  OODBMS can handle queries such as those on spatial data. For example, in order to determine if a point is inside it, a *polygon* class with a *IsPointInside* method can be created. The method can then be invoked on the instances of the polygon class. Other spatial queries can also be handled similarly.

- OODBMS also provides adequate support for versions and schema evolution (due to inheritance).

The above mentioned features of an OODBMS enable them to support complex applications that require rich modeling and efficient processing. Hence, the object oriented model seems to be the most appropriate data model for the biodiversity databases.

## 2.5 Development Stages

Having decided on the data model to be used for the design of the biodiversity databases, we will now describe in brief, the various stages in the development of an object oriented database. The development of an object oriented database involves five major stages, namely, *Analysis*, *Database Modeling*, *Database Design*, *System Design* and *Implementation* [Rumb91]. The following is a brief description of each of the stages.

1. **Analysis of the problem**

   In this stage, we collect information regarding all types of data, their constraints and relationships that need to be maintained in the database. This is formally presented in the form of a problem statement.

2. **Modeling the Database**

In the modeling stage, the real world objects, their constraints and relationships are represented using abstractions such as classes, associations and inheritance. The construction of an object model involves the following steps:

- *Identifying objects and classes.*
- *Identifying associations (including aggregations) between objects.*
- *Identifying attributes of objects and relationships.*
- *Organizing and simplifying the object classes using inheritance.*
- *Verifying access paths for likely queries.*

3. **Designing the Database**

   This design stage marks the beginning of the implementation phase. This stage involves the following steps.

   - *Identifying additional classes that would be required for the implementation.*
   - *Designing aggregations and other associations.*
   - *Identifying indexes and designing classes to represent them.*

4. **System Architecture**

   Once the design of the database is completed, we have the database model with classes that stem from the problem definition and those that were added as a part of the implementation requirement, their attributes, constraints, associations and methods. The implementation is that phase during which the real mapping is done between the model and the data structures of a programming language. In the field of databases, this means a mapping from the model to the data structures of a database programming language supported by an OODBMS.

   Thus, in order to implement the data model that has been developed, we need to design the *system architecture.* System architecture here refers to the overall organization of the database in terms of the back-end *OODBMS*, the *programming* *language* and the *user interface* though which an user interacts with the database system. This stage involves making the choice of the OODBMS that is most suitable for the application. Various factors need to be considered in making this choice. For example, the query processing requirements impose certain restrictions on the choice of OODBMS. This will be seen in Chapter 5, when the choice of OODBMS for OSHADHI is discussed.

5. **Implementation**

   This is the last stage in the development process when the model is converted to computer programs. This involves implementing the classes and relations of the model using the corresponding data structures of the language. Instances of the classes, the *objects*, constitute the database. The implementation includes *query processing*, which is one of the most important features of a database system. Query processing is treated as a part of the implementation, since the way a query can be processed depends not only on the model, but also on the power of the programming language that is used. This will be made clear when we discuss about the choice of OODBMS in Chapter 5.

   Several iterations need to be done at different stages of the database development, until all the requirements are met. The following chapters discuss each of these stages for OSHADHI in detail.

# Chapter 3

# Analysis and Modeling

In this chapter, we look at the first two stages in the development of OSHADHI, namely, analysis and modeling.

## 3.1 Analysis

In the analysis stage, we study the problem and collect details regarding the types of data that need to be maintained in the database, their constraints and relationships. This information is presented in the form of a problem statement. The following is the statement of the biodiversity data management problem.

### 3.1.1 The Problem Statement

The biodiversity data could be classified into four major types – the taxonomy data, the geographic data, the bibliography data and the experts data.

#### Taxonomy Data

According to Cronquist, taxonomy could be defined as

> *"a study aimed at producing a system of classification of organisms which best reflects the totality of the similarities and differences."*

Each unit of classification is called a *taxon*. The four major taxons in the classification of species are *Order*, *Family*, *Genera* and *Species*. An order contains several related families, which means that families that exhibit certain similar characteristics are grouped into an order. Likewise, a family is represented by a collection of genera and a genera is a collection of related species [Vasi74]. Each taxon has a biological name. The classification hierarchy of species is shown in Figure 3.1.

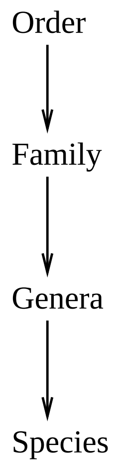



Figure 3.1: The taxonomy of species

Species are identified based on their leaf, stem, flower and fruit characters. Hence, these characters could be referred to as identifying characters. There are other characters, for example, the root character, which do not aid in identification of a species. These characters could be referred to as non-identifying characters. A flower is composed of several parts, namely, petals, sepals, inflorescence, carpels, ovaries, anthers, style and stigma. Each of these characters could be used for identification of a species and hence they are also identifying characters. Similarly, a leaf contains parts such as blades, sheath, etc. Hence, every order, family, genus and species is associated with a set of identifying and non-identifying characters which differentiate it from the other orders, families, genera and species respectively. A species may have synonyms, line drawings and a large amount of textual information associated with it. The textual information may include extracts from journals and textbooks. A species may also have other species as its predators.

#### Geographic Data

There are essentially four types of geographical organizations – the *bio-geographic provinces* ( e.g., Malabar, South Karnataka, etc.), the *biomes* (e.g., evergreen, decidous, etc.), the *administrative organizations* (e.g., district, state, etc.) and the *forests*. These organizations along with the botanical gardens and herbariums, represent the geographical data. Every Country, state, district and city has a political map associated with it. The environmental conditions ( soils, temperature, pressure humidity, etc.) in the countries, states, districts and cities need to be recorded in two forms. One is the spatial variation (distribution) of the condition and the other is the temporal variation ( variation of the condition over a period of time). Similarly, for all the species found in a geographic region, the distribution and temporal variation in population are to be recorded. The information about the distribution of a species could be available at any geographic organization level. For example, we might have the information that a species is found in the Malabar province or that a species is found usually in evergreen forests or that a species is found in Karnataka.

#### Bibliographic and Experts Data

The bibliographic data includes a collection of biodiversity related journals and textbook names. Each journal is referred to by its title and date. A journal has a collection of articles which refer to a species or a geographical region. An article has a group of authors.

The experts in the area of biodiversity, their area of specialization, affiliation and publications to their credit constitute the experts database.

### 3.1.2 The Queries

In the last section, we described the types of data and their relationships that need to be maintained in the database. We now focus on the query processing requirements of the biodiversity database. The following are some of the queries that need to be answered by the system:

- *Select an order, genus, family or species and list some/all of its characters*
- *Identify a species based on one or more of the flower/fruit/seed/leaf characters*
- *Display the map of a geographic location*
- *List all species in a given geographic location*
- *List all species with specific characters in a given geographic location*
- *Find out the experts in a specific field*
- *Display the distribution of a species in a geographic location*
- *Display the distribution of various climatic conditions like temperature, pressure and humidity in a geographic location*
- *List all predators of a given species*

Summarizing the problem statement, we can state the requirement of the system as follows :

- The database should incorporate all types of data that are specified and the system should provide facility for data entry.
- The system should be able to efficiently process queries such as the ones listed.

This completes the analysis phase of the database development process. Having identified the goals of the project, we now discuss the object model of OSHADHI.

## 3.2 Object Model of OSHADHI

The development of an object model involves various stages such as identifying the classes, associations, attributes and operations, simplifying the classes using inheritance and finally verifying the existence of access paths for likely queries. We have used the Rumbaugh [Rumb91] notation (given in Appendix A) to pictorially represent the classes, associations and inheritance. We now discuss each of the steps in the object model development of OSHADHI.

### 3.2.1 Identifying Classes

The taxonomy data includes instances of all taxons and their characteristics. The four classes – *Order*, *Family*, *Genera* and *Species* correspond to the four major taxons – the order, the family, the genus and the species. Classes are required to represent the identifying and non-identifying characters. Therefore, we have the classes *Flower*, *Fruit*, *Seed*, *Leaf*, *Inflorescence*, *Sepal*, *Petal*, *Anther*, *Ovary*, *Style*, *Stigma*, etc., to represent the corresponding identifying characters and a class *OtherChar* to represent any non-identifying character.

In the geographic data model, the classes *Country*, *State*, *District*, *City*, *Forest*, *Biogeographic province* and *Botanical Garden* represent the countries, states, districts, cities, forests, provinces and botanical gardens respectively. In OSHADHI, a map associated with a geographic location is modeled as a collection of polygons, lines and points [Banc92]. The classes *Point*, *Line* and *Polygon* are used to represent points, lines and polygons respectively and the classes *Points*, *Lines* and *Polygons* correspond to a group of points, a group of lines and a group of polygons respectively [Banc92]. The *Map* class represents the maps of geographic locations. The *Attribute Temporal Record* class represents the temporal records of an environmental condition in a geographic region and the *Species Temporal Record* class represents the temporal records of a species in a geographic region. Similarly, the *Attribute Spatial Record* and *Species Spatial Record* classes represent the spatial records of environmental conditions and species in a geographic location respectively. The *Value* class corresponds to the values that are associated with these records.

*Journal*, *Article*, *Author* and *University* are the major classes in the bibliographic and the experts data model.

### 3.2.2 Identifying Aggregations

Most of the objects dealt with in biodiversity databases are composite, i.e., they exhibit an aggregation hierarchy. The following are some of the aggregations in the biodiversity databases :

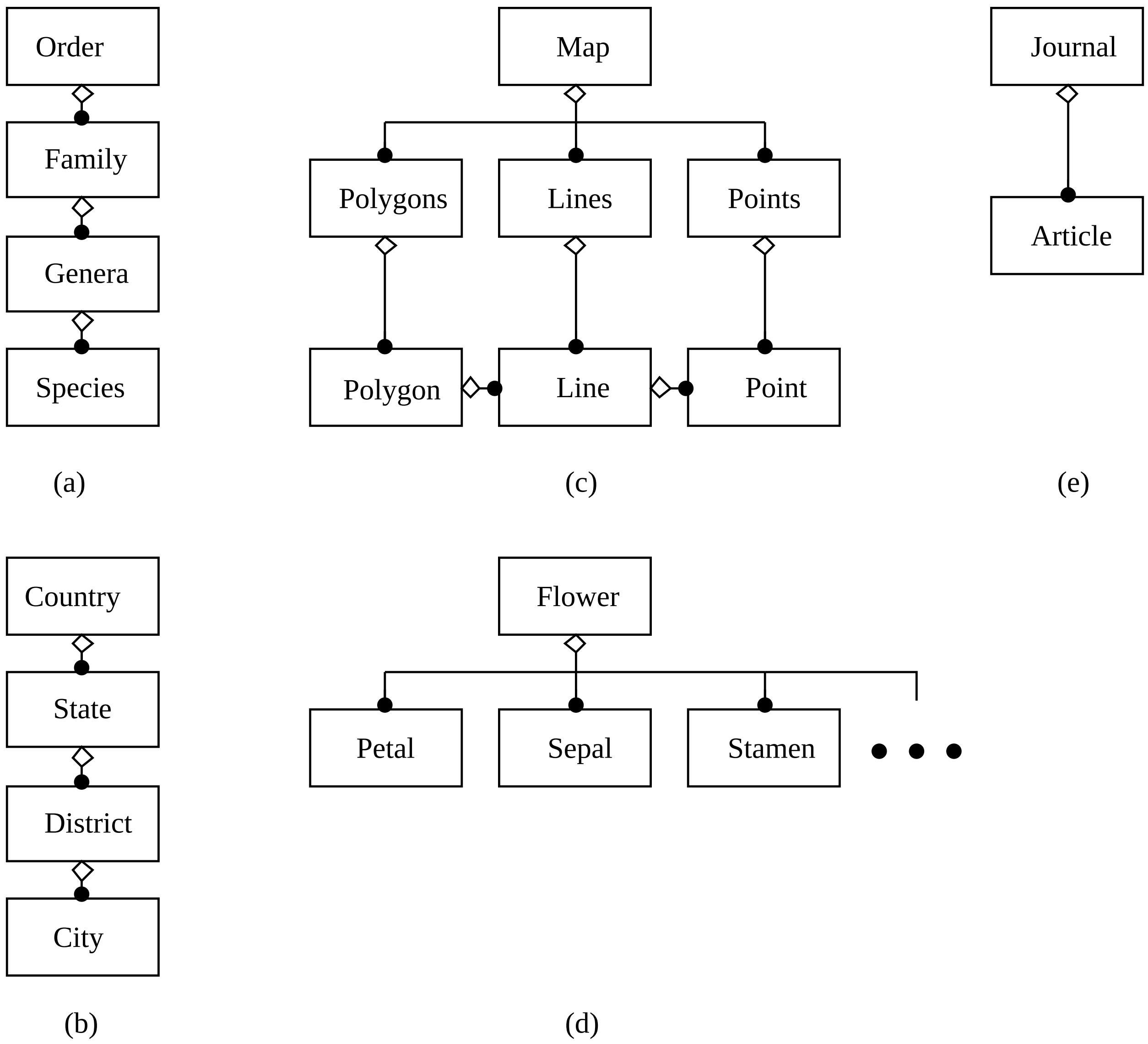



Figure 3.2: Aggregations in OSHADHI data model

- An order *contains* several families. A family in turn *contains* several genus and a genus *contains* closely related species. Since a family cannot belong to two orders, the order-family relationship is modeled as an aggregation rather than an association. Similar arguments hold for genera and species also. Hence, the taxonomy hierarchy is modeled as a four level containment hierarchy in OSHADHI (Figure 3.2a).

- A country *contains* a number of states, each state is made up of districts and districts are made up of cities. This relationship is also modeled as a four level containment hierarchy. Similarly, the association between a city and the botanical gardens in the city and the association between a country and a forest are modeled as aggregations (Figure 3.2b).

- The spatial data also exhibit aggregation. For instance, a map is a collection of polygons, lines and points. In turn, a polygon is composed of lines and a line is made up of points (Figure 3.2c).

- A flower is an aggregation of inflorescence, sepal, petal, anther, stamen, style and stigma(Figure 3.2d).

- A journal is an aggregation of several articles (Figure 3.2e).

### 3.2.3 Identifying other Associations

Aggregation is in some sense an association [Rumb91]. We have seen the various relationships that are modeled as aggregations in OSHADHI. The other associations between object classes are described below:

- Every order, family, genus and species is associated with some identifying characters. Since taxons exhibit a large amount of sharing of characteristics, an identifying character could be associated with many taxons. Therefore, an identifying character cannot be said as *belonging to* a specific taxon. Hence, the relationship between the taxons and the identifying characters is modeled as an association rather than an aggregation between the corresponding classes. Modeling the relationship this way helps in the process of *identification*. This will be seen in Chapter 6. Every taxon is also associated with the text class. The text object models the textual information about any entity (Figure 3.3a).

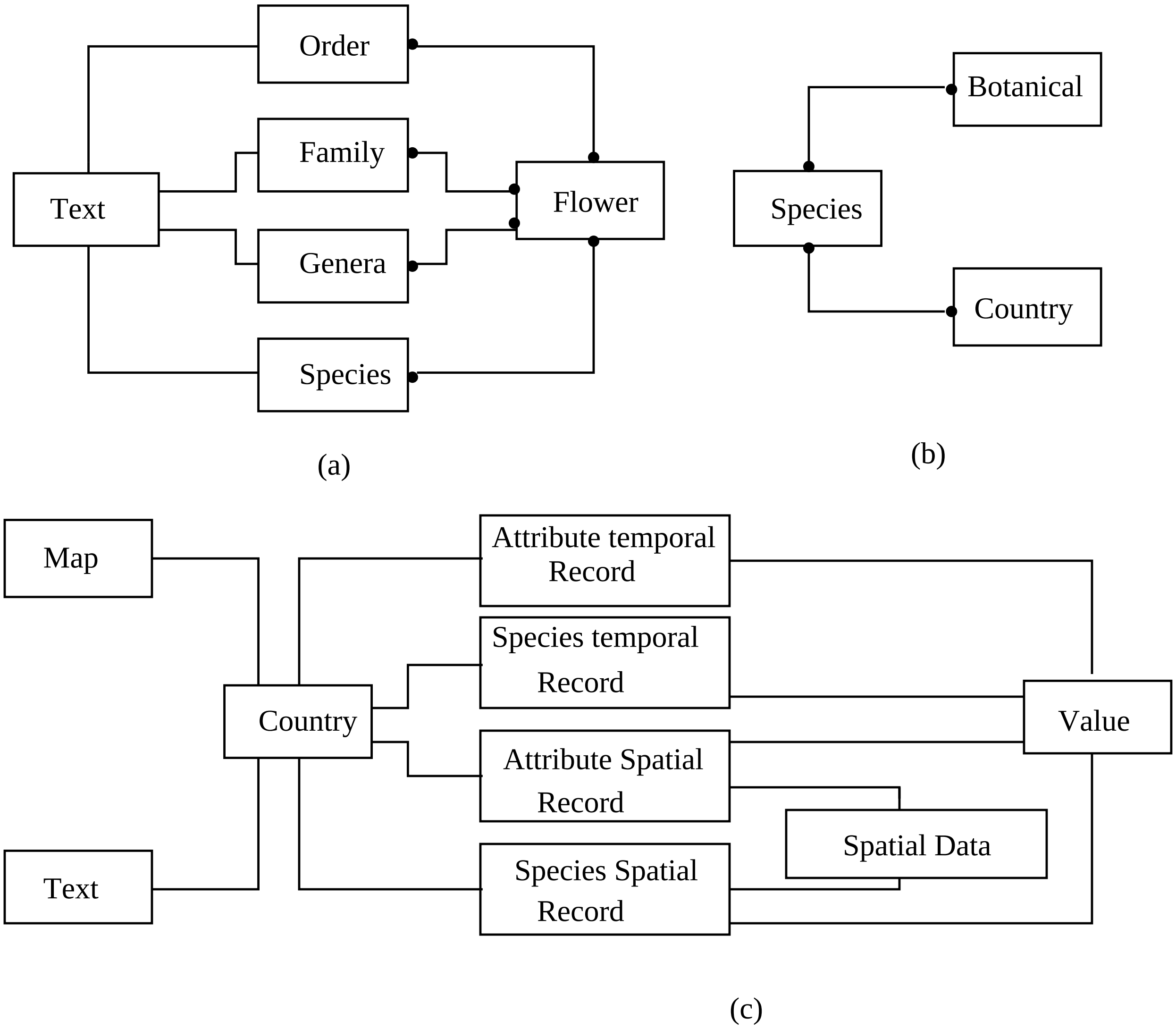
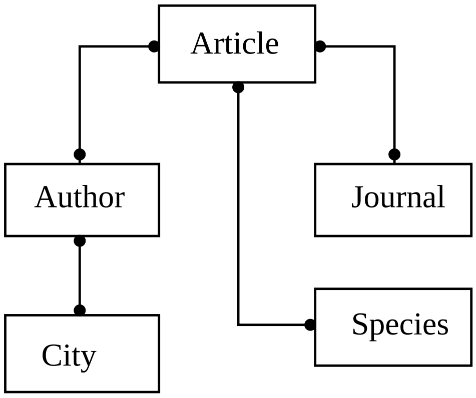



Figure 3.3: Other associations in OSHADHI data model

- A species is found in many geographic organizations and botanical gardens. This feature is modeled as two associations, one between species and geographic organization (could be a country, city, state, etc.), and the other between species and botanical gardens. A geographic organization or botanical garden may have several species associated with it (Figure 3.3b).

- Countries, states, districts and cities have maps associated with them. The textual information about a country, state, district or city is modeled as an association between the geographic location and the text. Countries, states, districts and cities




(d)

Fig 3.3

also have several temporal and spatial records associated with them. A species spatial record is associated with a value, a species and the spatial data that represent regions. Similarly, an attribute spatial record is associated with a value, an attribute name and the spatial data that represent regions or contours (Figure 3.3c).

- Information about a species is found in many articles and this is represented by the association between the species class and the article class. An article is associated with its authors, and authors have an association with geographic location (Figure 3.3d).

### 3.2.4 Identifying the Attributes and Operations of Classes

Attributes could be viewed as properties of objects that are not objects by themselves [Rumb91]. For example, a taxon has *name* as an attribute. A petal has attributes such as aestivation, colour, apex form, shape, etc., and a sepal has attributes such as symmetry, form, etc. Most of the methods for the classes are for reading, updating the attributes or objects contained in them ( in case of aggregations). The spatial classes such as the polygon, line and point have the *Display* method to draw the corresponding object. A polygon has a method *IsPointInside* to determine if a point is inside the polygon or not. A line has a method *DoesLineIntersect* to determine if a line intersects it or not. Methods are also provided to traverse all associations between object classes. The attributes and methods of some of the object classes are shown in the object diagram given in Appendix B.

### 3.2.5 Organizing and Simplifying Classes using Inheritance

In this phase, we organize the existing classes by identifying those classes that share similar attributes, methods and associations.

- **The Taxon Class and Ident Taxon Class**

  Order, family, genera and species are *Taxons*. The Taxon class generalizes order, family, genera and species. A taxon class has *OtherChar* and *text* associated with it. Order, family and Genera have identifying characters. Hence the three are generalized by *Ident Taxon* class (Figure 3.4a). The classes *Tree*, *Climber* and *Shrub* are specializations of species. Tree has additional attributes such as the bark characteristics. Similarly, climber and shrub have additional attributes.

- **The Geographical Region Class**

  The *Geographical Region* class generalizes country, state, city, bio-geographic province, forest and botanical garden. Every geographical region has a spatial data (map) and records associated with it (Figure 3.4b).

- **The Spatial Data Class**

  A map is a spatial data. The *Spatial Data* class generalizes all graphical entities. We derive two classes from the *Spatial Data* class. The class *Geos* and class *Geo*. *Geos* is a collection of *Geo*. A Geo generalizes the basic graphical entities – *Polygon*, *Line* and *Point*. *Polygons*, *Lines* and *Points* are specializations of Geos [Banc92] (Figure 3.4c).

- **The Record Class**

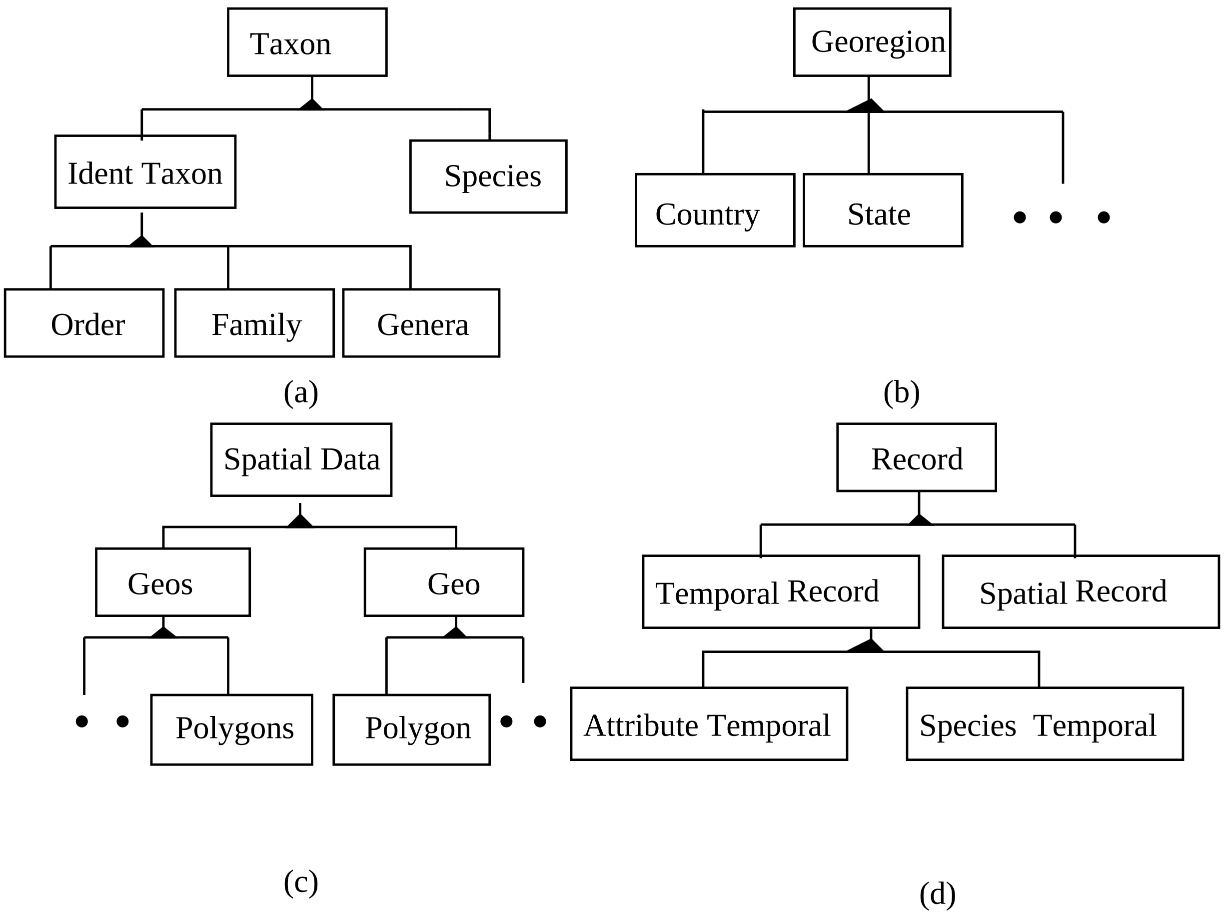



Figure 3.4: Inheritance in OSHADHI data model

*Temporal record* generalizes *species temporal record* and *attribute temporal record.* A temporal record has a time measure ( year, month, date or a time range) and a value associated with it. A species temporal record specializes the temporal record by having an additional association with species. Similarly, an attribute temporal record specializes the temporal record by having an additional property, the name of attribute (e.g population). Similarly, a *Spatial record* generalizes *species spatial record* and *attribute spatial record.* A spatial record has a value and a spatial data associated with it ( since a spatial data generalizes all graphical entities). A species spatial record specializes the spatial record by having additional association with species. Similarly, an attribute spatial record has an additional property, the name of the attribute. A *Record* generalizes *Spatial Record* and *Temporal Record*

(Figure 3.4d).

- **The Value Class**

  The Value class forms the super-class for the three primitive classes - *Integer*, *Float* and *String*. A *Range* class is derived from the Value class.

  The object diagram of the data model of OSHADHI is given in Appendix B.

### 3.2.6 Verifying Access Paths

This is the last step in the data modeling phase. Here, we verify the access paths for the queries that need to be processed. Most of the queries require traversing one or more associations. For example, let us consider the following query:

> Display the map of country X.

This query may be processed by picking the country object X, obtain the spatial data associated with it and invoke the *Display* method in the spatial data. The display method of spatial data would invoke the display method of all its components like polygons, lines and points.

In this chapter, we have discussed the analysis and modeling of OSHADHI. The problem was stated formally and classes, aggregations and associations were identified. The classes were then organized using inheritance.

So far, the model has been analyzed only in the problem domain and we now shift to the implementation domain. The implementation phase of development begins with the object design which we will be discussing in the following chapter.

# Chapter 4

# The Design

The design phase involves identifying the classes that would be required for the implementation, deciding the index structures to be used and designing associations considering the types of queries to be processed. In this chapter, we describe the design process of OSHADHI in detail.

## 4.1 Design of Associations

By "Design" of associations, we mean deciding on whether as association is to be implemented as unidirectional or bidirectional. This decision depends on the way the association will be traversed in the application [Rumb91].

### 4.1.1 One Way Associations

Associations that are traversed only in one direction in the application are implemented as one way associations, i.e., one object contains a reference or *pointer* to the object associated with it. The object that contains the reference is the one from which the association traversal is initiated. For example, the association between a taxon and text is an one way association which would be traversed from the taxon object. Hence, the taxon object contains an object reference to text. The following associations are modeled as one-way associations for the same reason:

- *The association between a geographic location and the record.*
- *The association between a spatial record and the spatial data.*
- *The association between a record and value.*
- *The association between a species and the species spatial record.*

### 4.1.2 Two Way Associations

Many associations in the biodiversity database model are two way associations, i.e., the association is likely to be traversed in both directions. There are three ways of implementing two way associations [Rumb91]:

1. The association is implemented as a one way association. This allows the association to be traversed in one direction. When the traversal in the opposite direction is required, a search is performed. This kind of an implementation is useful if the traversal in one direction is more frequent than the traversal in the opposite direction.

2. The association is implemented as a two way association, i.e., the two associated objects have object references to each other as attributes. This implementation consumes space and needs additional maintenance while updating the database (two pointers have to be updated to maintain consistency and to avoid dangling references). However, this implementation has an advantage of fast access.

3. An association object is used. An association object is a table of pairs of object references to the objects that are associated. When an association object is used, we need to search the table for both forward traversals as well as backward traversals. This implementation is useful when the associations are sparse ( only a few objects participate in the associations).

The most efficient implementation depends on the nature of the data and queries. The second implementation is the most effective in terms of access time. The disadvantage of additional pointer maintenance does not cause too much burden in a database where the accesses outnumber the updates. Since the biodiversity database is characterized by low frequency of updates, the second would be the most efficient implementation of most of the associations. For example, the association between an identifying character and a taxon is a two way association. Let us consider the following queries that require traversal of the association:

- *Given a taxon, find its identifying characters.*
- *Given a character, identify all taxons that possess the character.*

While the first query requires the traversal of the association from a taxon to its characters, the second query requires the traversal of the association from a character to the taxons. Since both these queries are equally frequent, implementing this association as a one way association (method 1) would result in a skewed response for one of the queries. The third method requires a search for both traversals and is therefore slower in access than the second method. Moreover, the associations are not sparse. Hence the second method is preferred over the first and third. This association is therefore implemented using object references to each other. The following are the other two way associations in the data model:

- *The association between a geographical organization and its map.*
- *The association between species and a geographic organization.*
- *The association between species and botanical gardens.*
- *The association between a species and an article.*
- *The association between a taxon and other characters.*

### 4.1.3 Aggregations

Aggregation is implemented by embedding the *contained object* in the object that contains it, i.e., by declaring an attribute in the aggregate object to be of the contained object type. This differs from an association implementation, where the attribute is declared as a reference to the associated object. Although aggregations are usually traversed from the composite object to the objects it contains, it may sometimes be useful for the contained objects to know the composite object which it belongs to. This is treated as an association and is implemented as an object reference from the contained object to the object that contains it. For example, a typical query on the taxonomy data could be:

> Which family does a given species belong to?

A family contains several genera and a genus contains several species. Although this has been modeled as an aggregation, it is useful in this case, to maintain an object reference from the species to the genera it belongs to and similarly, from genera to the family that the genera belongs to.

#### Encapsulating the implementation

The implementation of associations and aggregations are encapsulated by providing methods to traverse them [Rumb91].

## 4.2 Classes Evolved for Implementation

The data model that was presented in the previous chapter deals only with the problem domain. The classes there represent the types of data that need to be maintained and the data model represents the relationships between the classes in the real world. However, there is a need for more classes to implement the data model. In this section, we will the see the major classes that are required for implementation. The class definitions of these classes are given in Appendix C.

### The Databases

A database could be treated as a collection of all the objects represented in the data model. For convenience, the biodiversity data objects are grouped into four major databases:

- **Taxonomy database**

  This includes all the taxons.

- **Characters database**

  This includes all the identifying and non identifying characters.

- **Geographic database**

  This includes all countries, cities, states, districts, botanical gardens, forests, the records, etc.

- **Spatial Database**

  This includes spatial data such as the polygons, lines and points used to represent the maps of geographic locations.

  The data model for bibliographic data and experts data have not been implemented in OSHADHI. However, we expect the addition of these two types of data to be simple.

A database could be viewed as an object. The class that represents the object provides an encapsulation for all the objects in the database and interactions to the database are through its methods. Correspondingly, we have four classes to represent them:

- *TaxonomyDatabase Class*
- *CharDatabase Class*
- *GeoDatabase Class*
- *SpatialDatabase Class*

The databases has objects of the *Pointer Collection* type, of the *Encoded Collection* type and of the *Indexed Collection* type. The databases also contain several index objects such as $B^+$tree index and QuadTree index, which are instances of the $B^+$*tree Index* class and the *Quadtree Index* class respectively. These classes are discussed below:

## The $B^+$tree Index

geographic regions have their names as primary keys. A $B^+$tree could be used to access these objects by their names. Hence it is useful to have a class for the $B^+$tree index. The actual index structure is private and the methods provided to manipulate the index are *Insert*, *Remove* and *Retrieve*.

## The QuadTree Index and Quadtree Node

The spatial database includes several points, lines and polygons. Some queries require finding the geographic location corresponding to a point in a map. This requires an indexing on the geographic locations based on their spatial data. A quadtree index [Same89] is used for this purpose. The quadtree provides member functions such as *Insert* and *Search*. The index structure itself is private and is implemented as a tree of nodes. The nodes are objects of quadtree node [Same89] class. A quadtree node has a point field, a pointer to an object, and four children which correspond to the north-east, north-west, south-east and south-west region of the point associated with the quadtree node. The node has the operations of *Insert* and *Search*. The quadtree indexing is discussed in detail in chapter 6.

## The Indexed Collection

Most of the objects in the database have names. For example, orders, genera, families, species, countries, states, districts and cities have names. The database consists of a collection of objects of each type. Many queries pertain to an object selected by name from these collections. Therefore, these collections are implemented as objects of type *Indexed Collection*. The indexed collection class contains a collection of objects and a $B^+$tree index built with the name of the object as the key. Both of them are maintained as private members. The methods for indexed collection are *Add*, *Delete*, *Search*, *Iterate*. The add and delete functions update the $B^+$tree index and the search member function uses the $B^+$tree index for searching by name. This is implemented as a template class [Stro91].

### The Encoded Collection

The database has a collection of character objects. These include flower objects, fruit objects, inflorescence objects, etc. For identification, we need to select objects from these collections which have certain values for certain attributes. We then traverse the association to get the taxons that possess those characters. However, the attributes for which the values would be specified are not known *apriori*. Maintaining an index on each attribute would be too expensive, since the number of values that most of the attributes can take is typically small (between 5 and 10) and indexes are generally useful when attribute values have large domains. Also, we have to do multiple intersections to get the character objects required when more than one attribute is specified. We therefore use a numeric encoding method to build an index for objects of this type. In this method each attribute is assigned a numeric code based on its attribute values. This numeric code forms the key for a $B^+$tree index. This implementation is encapsulated by creating a new class, the Encoded Collection. Hence collections of flower objects, fruit objects, petal objects etc., are objects of the encoded collection type. The numeric encoding method and the way it is used in query processing are discussed in detail in Chapter 6. The encoded collection is also implemented as a template class. The methods provided by this class are *Add*, *Delete*, *Search*, *GetRange*. *GetRange* yields a stream of object references of objects whose numeric code lies between two specified values.

### The Pointer Collection

Most of the associations are one to many. These are implemented as object references from one object to a group of objects. Therefore, an object of one type associated with many objects of another type will have a collection of object references as private members. We identify the pointer collection class which forms an abstraction for the collection of object references. The class encapsulates the actual pointers through the methods like *Add*, *Delete*, *Search* and *Iterator*. Iterators iterate through the collection and yield a stream of pointers. This class is also implemented as a template class.

In this chapter, we have seen the complete design of the database, the new classes that evolved due to implementation requirements, the design of associations and aggregations and the index classes. This completes the design phase in the database development.

# Chapter 5

# System Architecture

So far, we have seen the object modeling and design of OSHADHI. We now move on to discuss the next phase in the development of OSHADHI, namely, the system architecture design. In this phase, we organize the database in terms of the back-end OODBMS, the programming language and the user interface through which the user interacts with the system. The decisions regarding the back-end OODBMS and the programming language are made taking the query processing requirements into consideration.

## 5.1 The Platform

Object oriented database management systems could be classified into two broad categories: *Complete* and *Extensible*. A Complete DBMS is an end-user DBMS with built-in query language, indexes and access mechanisms. In a complete DBMS there is no way by which an user can add new access methods or query language. This approach is well suited for commercial applications. This is because query languages such as SQL, QUEL and indexes such as $B^{+}$tree, hash tables are suitable for all commercial applications. But when we consider applications such as CAD/VLSI, AI or scientific applications it is not possible to design a query language or access method which suits all such applications. This difference stems from the nature of data and queries that were discussed in chapter 2. For such applications, we can classify the components of a DBMS into application independent and application dependent components. While transaction management, concurrency control, recovery and storage management are application independent features of a DBMS, the query language, indexes and access methods are application dependent [Care86a]. For example, while a $B^+$tree index would be appropriate for searching through a collection of species (the name of the species will be the primary key), Quadtree [Same89] would be a suitable index structure for spatial objects. Similarly, a query language developed for one application may not be suitable for another as the nature of operations might differ from one application to another. The new application areas require facilities by which application specific access methods, indexes and query language may be added.

One of the approaches to satisfy this requirement is to make a DBMS *Extensible* [DBE87] There are two approaches to achieving extensibility [Care86a]. One of the approaches is to provide a complete DBMS, with its query language and access methods, but provide ways by which an user can add new data types and access methods. This approach is adopted in projects such as POSTGRES [Rowe87]. The second approach is the *toolkit* approach, where the user is provided the basic building blocks of an OODBMS and a programming language interface to build application specific query language and access methods. Thus, the user can build an application specific DBMS by combining the components that are already provided and the components that were constructed for the application. This approach is used in EXODUS [Care86a] and GENESIS [Bato88]. Since the application that is being discussed requires implementing index structures such as the quadtrees, we have chosen EXODUS as the back-end for OSHADHI.

The following section briefly discusses the architecture of EXODUS.

## 5.2 The Architecture of EXODUS

EXODUS achieves extensibility by providing the basic components of a DBMS. EXODUS is composed of the five major components [Care86a] – The storage manager, the E programming language and its compiler, a library of access methods, a query optimizer generator and Tools for constructing the query language front-ends.

### 5.2.1 The Storage Manager

The storage manager is the lowest level software which is responsible for creating and operating on storage objects. The storage manager also ensures concurrency and recovery on the storage objects. An object is viewed as a variable-length byte sequence which can be read and updated by the storage manager [Care86b, Exod93]. The storage manager provides a procedural interface which can be invoked by applications.

### 5.2.2 The E Programming Language

The E programming language [Rich89a, Rich89b] is provided for implementing application specific access methods and operator methods for application specific query language. E extends C++ [Stro91] by adding features like persistence, iterators and certain other database constructs.

#### Persistence

E provides database types called *DB Types* analogous to the C++ data types. A db type could be one of the following : *dbshort, dbint, dblong, dbfloat, dbdouble, dbchar and dbvoid.* Every C++ data type has a corresponding db type. Objects of dbtypes are similar to their non-db counterparts except that an object of a non-db type is volatile (whose lifetime is bound by the program run) whereas an object of db-type could be declared **persistent**. A persistent object survives across program runs and crashes.

#### Iterators

This is a feature that aids database querying. Iterators are control abstractions which have two agents that work together. One is an iterate loop and another is the iterator function. The iterate loop invokes the iterator function which yields a stream of values one-at-a-time to the client loop which then processes it. The iterator function therefore needs to retain its state between two yields. This feature is particularly useful in performing operations on a group of related objects since the iterator function could be written independent of the client loops.

#### Collections

E provides a way of grouping related objects into *Collections*. A collection may also be defined to be persistent.

#### Transactions

E provides built-in functions to begin, commit and abort transactions.

The features of E discussed above provide a neat interface between the storage manager and the user. The constructs in E are translated into appropriate calls to the storage manager routines. These features allow E to be used as a database programming language also [Atki87]. More details on the programming constructs of E are given in [Rich87].

### 5.2.3 Access Methods in EXODUS

EXODUS provides certain built-in index structures such as $B^{+}$tree index and linear hashing. New access methods could be written using E. Implementing a new access method would require declaring the required index structure to be of db type and creating an object of that type. The application code can then manipulate the index structure just as any other volatile object. Persistence in E thus shields the user from having to worry about the internal structure of the persistent objects [Care86a].

### 5.2.4 Constructing an Application Specific Query Language

EXODUS provides a rule-based query optimizer generator. This generator takes in an input specification and produces an optimizer. The input specification contains a set of predicates, the methods to perform the operations, the cost factors and a set of transformation rules. The optimizer can then take in a query, transform it using the set of rules and then select the access paths for the transformed query. EXODUS also provides tools to generate front-ends for query languages that parse a query and yield an operator tree. This operator tree is then optimized by the generated optimizer, converted into E program, and then compiled into an executable [Care86a].

## 5.3 Architecture of OSHADHI

OSHADHI is based on a client server model. The following are the various components of the system.

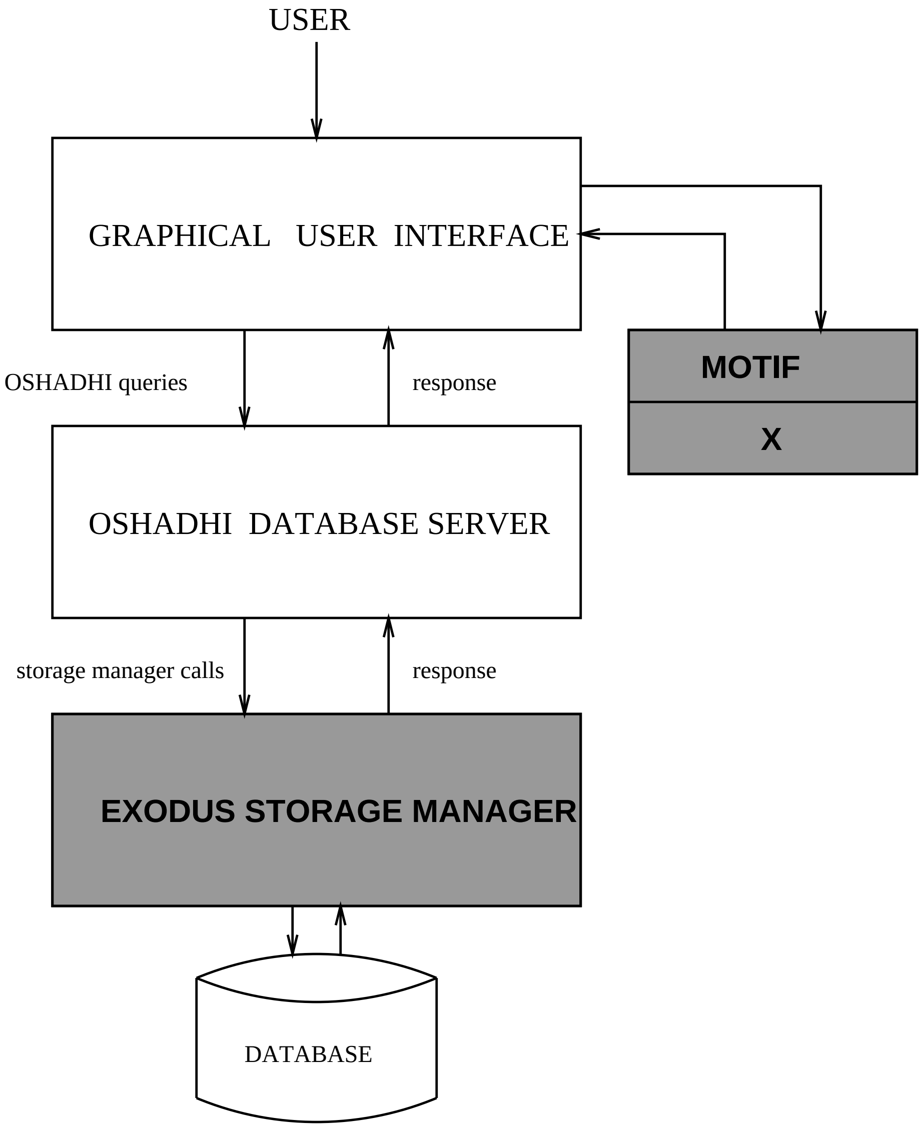



Figure 5.1: System Structure of OSHADHI

- **The EXODUS Storage Manager Server**

  The storage manager server of EXODUS forms the lowest layer of OSHADHI. The server receives requests from clients for various operations on objects and collections.

The server is responsible for concurrency control, recovery, transactions and storage management.

- **The Database Server**

  The database server acts as a client of the EXODUS storage manager server. The database server is responsible for query processing and for all the operations on the biodiversity database. After initialization, the database server waits for a query from its client. The results of the query that are obtained by processing it are sent back to the client.

- **The Interface**

  This is a graphical user interface between the user and the database server. The user interactions with the GUI results in a query which is sent to the database server. The interface waits for the result of the query from the database server and displays it in a suitable form when it is received. The GUI is built using X and Motif [Youn90]. The system architecture of OSHADHI is shown in Figure 5.1.

# Chapter 6

# Implementation

The Database Server and the Graphical Interface are the two components of OSHADHI that are built on top of the EXODUS storage manager. The implementation details of the two are discussed in this chapter. The database server is responsible for query processing, while the interface is responsible for generating the queries from the user interactions and displaying the result of the query. In this chapter, we discuss the implementation of the database and the algorithms that are used by the database server for processing the queries.

## 6.1 Implementing the Database

As the first step in implementing the database that was designed as shown in chapter 4, we should define the schema of the database. The Db types of E (5.2.2) are used for this purpose. Therefore, every class that was identified in the data model is implemented as a dbclass and inter-object references are implemented as pointers of objects of a dbclass. Populating the database would then be writing code to add persistent objects belonging to these classes. Some of the class definitions written in E are given in Appendix C.

## 6.2 E as a Database Programming Language

Although the main purpose of E is to ease the development of internal system software, its features allow it to be used as a database programming language also. We have used E as a database programming language in the development of the database server of OSHADHI. We do not build a query language as mentioned in Section 5.2.4, but define some fixed set of operations and arguments for them. The server then calls appropriate procedures to process the queries using optimized algorithms that are designed for them. Hence, we can say that the query processing in OSHADHI is built into the database server code. In the following sections, we describe the various operations on the database and how they are processed.

## 6.3 The Access Methods in OSHADHI

Before we discuss the query processing methods, we briefly discuss the indexes that are built into the database. The types of indexes that are used were mentioned during the object design phase in Section 4.2. We now discuss about these indexes in detail.

- **Index for objects with name : The $B^+$tree Index**

  Objects like the orders, genera, families, species, countries, states etc., have names. A query most often would select an object by name and perform operations on it. We use a $B^+$tree index with the name of the object as the key and this allows retrieval by name. We use the *StrTypedBtreeIndex* class provided by E to instantiate $B^+$tree objects in the database. The StrTypedBtreeIndex class encapsulates the storage manager routines that are used. The StrTypeBtreeIndex is a template class with the object type as the template.

- **Index for the 'character' objects**

  The character objects include flower, fruit, inflorescence, etc. Unlike the named objects, these objects do not have a primary key. This is because searches on the collection of flowers or inflorescence could be based on one or more attributes of the objects. Building a $B^+$tree index on every attribute would be expensive both in terms of space and efficiency, since there are only a few possible values that any attribute would take. Also, searches involving more than one attribute would need several intersections. We therefore use a numeric encoding scheme along with the $B^+$tree index. We build a $B^+$tree index on the character objects with the key as a numeric encoding of all the attribute values.

### Numeric Encoding

Let us consider a character class with certain attributes. The values each attribute can take are enumerated and the numeric value of each of the attributes of an object are obtained based on this enumeration. The values of all attributes are concatenated to yield a number. This number forms an **unique code** for the object which can form the primary key. However the number cannot be used as a key directly since the concatenation might yield a number that cannot be represented by the integer variable. We therefore obtain a string code corresponding to that number and use that as the key. Let us consider the class defined for a sepal character. A sepal has the following attributes: nerve character (which specifies whether the sepal is nerved towards apex, or nerved, but not towards apex or not nerved), symmetry character (which specifies whether the sepal is symmetrical, slightly asymmetrical or strongly asymmetrical), form ( which specifies whether a sepal is oval, triangular, lanceolate or some other form) and fusion ( which specifies whether a sepal is free, fused less than half its length, fused greater than half its length or fused entire length). The class definition of SepalChar is given below:

```
dbclass SepalChar : public IdentChar
{
    NERVECHAR nerves;
    SYMMETRYCHAR symm ;
    FORM form ;
    FUSIONCHAR fusion;
```

```
    public :
      ...
};
```

The attributes nerves, symm, form and fusion are of enumerated type. The enumeration is given below:

```
dbenum NERVECHAR    {NOTNERVED =1, NERVE_TO_APEX, NERVED};
dbenum SYMMETRYCHAR {SYMMETRICAL=1, SLIGHT_ASSYM, STRONG_ASSYM};
dbenum FORM         {OVATE=1, LANCEOLATE, TRIANGULAR, OTHERFORMS};
dbenum FUSIONCHAR   {FREE=1, FUSED_LT_HALF, FUSED_GT_HALF, FUSED_ENTIRE};
```

The enumeration starts from 1 rather than from 0 because the value 0 for an attribute is used for unspecified attributes in the identification query. This is discussed in detail in Section 6.4.

Now let us consider a SepalChar object with the following values for the attributes:

```
nerves = NERVED_TO_APEX ;
symm   = SLIGHT_ASSYM ;
form   = LANCEOLATE ;
fusion = FREE ;
```

The values of the attributes are 2,2,2 and 1 respectively. Therefore, the key for this object is "2221". We will see how this index is used for searching through the characters database for objects when one/more attributes are specified, in Section 6.4.

- **Index for Spatial Data**

  We use the Quadtree [Same89] for indexing the spatial data. Most of the queries on spatial data involve one of the following sub-queries:

- Given a point determine the corresponding city.
- Given a point determine the state/district containing the point.

The details of how a quadtree index is used to fasten the access of spatial data structures are given in Section 6.4

## 6.4 Some Important Queries and their Processing

In this section, we will look at some of the typical queries on the database and how they are processed.

1. **Data Entry**

   Objects in the database are added, deleted or modified by sending a query to the database server. The query specifies the operation - *Add* or *Delete*, the object type, for example, order, genera, species etc. and the attribute names and values. The following is an example of a data entry operation:

   - **Add species** *Magnolia-champa* **in genera** *Magnolia arrangement polystichou stalk sessile petalstruct fleshy.*

     This query searches the species collection for "Magnolia-champa". If the species already exists, this query is viewed as a "modify" operation and the attributes of the species are modified. If no such species is found, a new species object with name "Magnolia-champa" is created. The database server then creates a flower object (if it does not exist already) with attribute arrangement having a value POLYSTICHOU and attribute stalk having a value SESSILE. The searching of the flower object is done using the $B^+$tree index and the numeric code for the object. A petal object is then created (if it does not exist already) with its structure attribute having a value FLESHY. The three objects are then linked by creating object references from species object to the flower object and the flower object to the petal object. The back pointers from flower to species are also updated by adding the object reference of the species to the collection of pointers to taxons in the flower object. Setting up the object references are done through invoking the appropriate methods.

- **Add country** *X* **with spatial-data** *15.6 12.4 4.5 6.7 10.9 6.6 7.8 9.9.*

  This query creates a new country object (if it does not exist) with the name X. The list of numbers that follow the spatial data attribute are the coordinates of the end points of the polygon used to represent the map of the country. The coordinate of a point refers to the longitude and latitude respectively. The database server creates a polygon (as a collection of lines that bound the polygon) in the database and sets up an object reference from the country to the polygon and the quadtree indexes are updated.

  **Updating the quadtree index for polygons**

  Two quadtree indexes are maintained for entities like country, state or district whose spatial data is a polygon. The bounding rectangle of the polygon is first computed and this rectangle has two representative points - ($C_x$, $D_x$) and ($C_y$, $D_y$), where ($C_x$,$C_y$) is the centroid of the rectangle and $D_x$ and $D_y$ are the half distances measured from ($C_x$,$C_y$). We add the two representative points in the two quadtrees. The searching method using these quadtrees will be discussed shortly.

- **Add species** *Magnolia-champa* **to state** *Karnataka.* This query is creates a link between two objects, the species object with name "Magnolia-champa" and the state object "Karnataka". This query requires the two objects to be present in the database. The server searches for both the objects and sets up the object references to each other.

- **Delete species** *Magnolia-champa*

  The server searches for the species "Magnolia-champa" in the species database. All attributes such as text, which are not shared between taxons are deleted. However, the attributes such as the flower character could be shared between taxons and therefore are not deleted unless this is the only species which refers to it. Information regarding the number of taxons that share a character object is maintained by updating *Count*, a private variable in the character objects. If the count is zero, the object is deleted and if the count is greater than zero, the object reference of the species is deleted from the flower object, but the flower object itself is not deleted. Finally, the species object is deleted from the database.

2. **List** *family.*

   This gives a listing of existing families. This query is processed by invoking an iterator( 5.2.2) in the taxonomy database class.

3. **View species** *Magnolia-champa flowerChar.*

   This query allows the various attributes of a taxon to be viewed. The user can select the attribute that is required. It could be one of identifying characters, textual information, genera name or other characters. This query is processed by locating the species objects and traversing the appropriate associations to get the required attribute.

4. **Identify taxon with** *petalstruct fleshy petalunion polypetalous*

   This query is for identifying a taxon based on one or more of its characters. For example, the above query is for identifying all taxons which are polypetalous and have a fleshy petal structure. In order to process the query, the $B^+$tree index discussed in Section 6.3 is used. A petal object with the attributes specified is created and its numeric code is obtained. In this case, the numeric code would be "2020". The petal objects in the database would then be those which have numeric codes are of the form "2x2y" where x and y can take values between 1 and the maximum value that the corresponding field may take. In this case, x can take value from 1 to 3 and y can take values from 1 to 7. In order to determine all such objects we generate a range query and obtain all the $B^+$tree nodes whose key values lie between "2020" and "2327". However, these would include a node whose key is "2030". We therefore select those nodes whose key is of the form "2x2y" from the resulting set of $B^+$tree index nodes. The petal objects contain references to the taxons that possess that petal character. We repeat this procedure for all the required character objects like inflorescence, sepal etc. At the end of the procedure, we have sets of object references to taxons, each set corresponding to a specific identifying character like petal, sepal, inflorescence, etc. Each set is a collection of pointers that are sorted by the name of the taxon which they point to. An intersection is performed on these sets of pointers to get the result of the query.

5. **Pick state in country** $X$ **containing point** *12.5 16.6.*

   This query is for choosing a state from a country given the latitude and longitude of a point. The country specified is first located and we can use the quadtree indexes built on the spatial data of the states in that country to locate the state which contains this point. We use the algorithm proposed in [Same89] for this purpose. As mentioned in 6.3 we represent every polygon (state map) by two representative points, the X representative point and the Y representative point. Let A, B, C, D and E correspond to five states in a country. Figure 6.1a shows the X representative points in the $C_x - D_x$ plane and Figure 6.1b shows the Y representative points plotted in the $C_y - D_y$ plane.

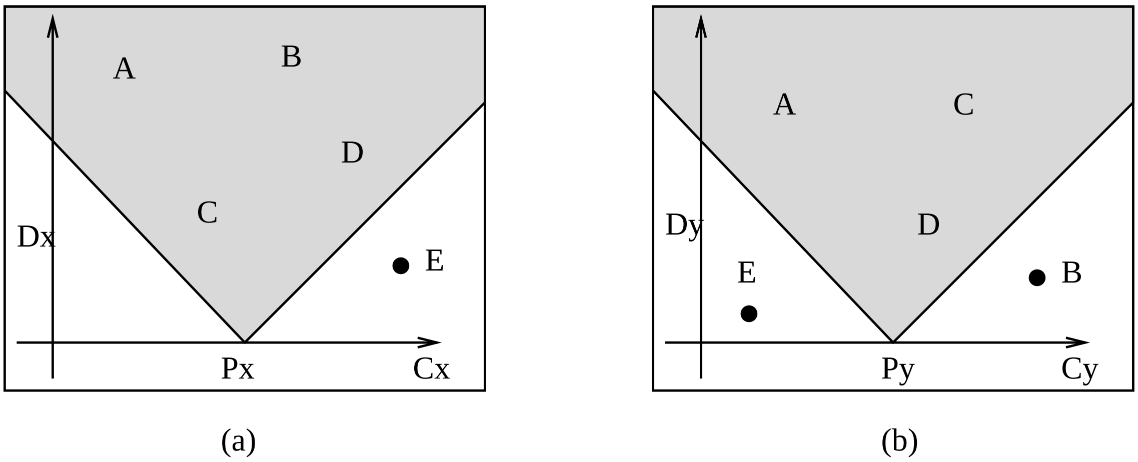



Figure 6.1:

Let us consider a point P whose coordinates are $P_x$ and $P_y$. The rectangles (represented by the points A, B, C, D and E) that intersect a vertical line through P are those that are contained in the shaded region of Figure 6.1a. Similarly, the rectangles that intersect a horizontal line through P are those that are contained in the shaded region of Figure 6.1b. The rectangles that contain the point P are those that are inside the shaded regions of both the figures. Hence we can obtain those rectangles by intersecting the two sets. In the above example, A, C and D are the three rectangles that contain P. However, what we need is the polygon that contains P. We therefore iterate through all the polygons that correspond to the rectangles obtained and determine which polygon contains the point (using the point in polygon algorithm). Since we are dealing with state boundaries, the polygons representing them are non-overlapping and hence we stop the iteration once we find one polygon that contains P. The state object that corresponds to that polygon is the required state.

**The point in polygon algorithm** [Hear90]

In order to determine whether a point lies inside a polygon or not, we consider a horizontal line from a point outside the polygon to the point considered. The number of intersections of this line with the boundaries of the polygon are calculated. If this number is odd, the point is inside the polygon and if the number is even, the point is outside.

6. **List species in state** *Karnataka* **with** *arrangement polystichou.*

   This query implies listing species in Karnataka with certain characteristics. This could be viewed as an extension of identification. This query is processed by obtaining the pointers to all species in Karnataka and pointers to all species which have the characteristics. The result of the query is obtained by performing a pointer intersection between the two sets of pointers.

7. **Display states in country** *India* **where** *Magnolia-champa* **is found.**

We locate the species “Magnolia-champa” and obtain references to all the geographic locations where the species is found. We also obtain the object references of all the states in the country. A pointer intersection between the two sets yields references to the desired states. The spatial data of these states can then be displayed.

8. **Display** *soils* **of country** *India.*

   This query requires obtaining the attribute spatial records associated with the country, having the attribute name as "Soil". The spatial data with the corresponding values can then be displayed. Similarly other attributes can be displayed. Conditions such as "soil is alluvial" may also be specified.

Although the object model permits several possible queries, we have currently limited them to the type of queries mentioned above. However, other queries can be added easily to OSHADHI.

So far, we have discussed the role of the database server in OSHADHI. The following section describes the graphical interface, which serves as the link between the user and the server.

## 6.5 The Graphical User Interface

The Graphical Interface is the only way by which an user can interact with the OSHADHI database server. The interface is responsible for generating the queries from the user interactions and for displaying the results of queries that are obtained from the server.

### 6.5.1 The Data Entry

Most of the data entry for textual data are through standard forms. This minimizes the typing errors and reduces the work that needs to be done by the end user. The interface is menu driven and as the user clicks on various buttons, the queries are formed. Once the user completes the data entry, the formed query is sent to the server. An example form for the entry of a species is shown in Figure 6.2. This form uses a combination of text widgets, list widgets and toggle buttons [Youn90]. The text widgets are used for entering the species name and genera name. The characteristics are entered by the user by clicking the buttons which change state (between selected and not selected) after every click on them. Since the set of characters might evolve with time, we have not included the character names in the interface code. Instead, the characters are read from a text file before building the form. This enables the user to modify the names of characters or add new characters in the form, without having to modify the code. The text information related to the species is entered in a simple text window that pops up when the user clicks on the TEXT button. The geographic information is entered by clicking on the appropriate menu button. Figure 6.2 shows the details of the species "champa" being edited. The genera name is "Magnolia" and the characters that have been entered are: Arrangement – Distichous, Stalk – Sessile and Petal Shape – Orbi. The ACCEPT button is clicked when the form entry is completed and this creates the following queries:

1. Add species champa to genera Magnolia arrangement polystichou stalk sessile petal-struct fleshy.
2. Add text to species champa ...(the text matter).

This query is then sent to the database server which then processes this query. However data entry for spatial data is handled differently. Although the most effective way would be to have map images and automate the process of entering spatial data by extracting the contours and points [Kast89], we have used the manual method for entering spatial data. We have adopted this method because our application does not require too much accuracy with respect to the contours or point coordinates. We therefore provide a drawing area with simple drawing facilities. The latitude and longitude ranges are marked to aid the user to draw the map in the right position (although it may not be accurate). The user is also provided with a facility to have an existing map in the background, so that the relative positions of geographical locations can be determined. Non spatial data could then be entered using standard forms.

### 6.5.2 Querying

OSHADHI uses form based querying instead of ad-hoc querying. This is because, the users of OSHADHI would primarily be researchers and scientists in the field of ecological sciences and we expect a form based querying facility to be more useful than an ad-hoc query language interface. Querying on textual data is similar to data entry. The user goes through pop-up menus to form a query. Most of the menus have options from which the user has to select one. The result of the textual queries are displayed using the Motif widgets.

For querying on the geographic database, the user first has to select a geographic location of interest. For example, when India is selected, the map of India is displayed. The options on the side menu allow different queries to be generated. The user may select any state by clicking on the state. This highlights the state and the queries formed will be on the selected state. If no state is selected, the query will be on the country, whose map is on display. Figure 6.3 shows the map of India being displayed. The user has selected the state Karnataka (highlighted).

### 6.5.3 Error Messages and Other Features

Several validations are done during data entry and querying. For example, when we want to add a species to a genera, an error message is printed if no such genera exists.

During data entry of a species, when the genera name is entered, the characteristics of the genera are highlighted. This reduces the amount of data to be entered since the characteristics of the genera would also be characteristics of species. However, we allow these characteristics to be edited, so that exceptions may be handled.

## 6.6 Other Implementation Details

The OSHADHI code size is around 13,000 lines (this includes the server and the graphical interface code). The data model is implemented using around 65 classes and the database server uses 20 additional classes for query processing.

Figure 6.2: An example form

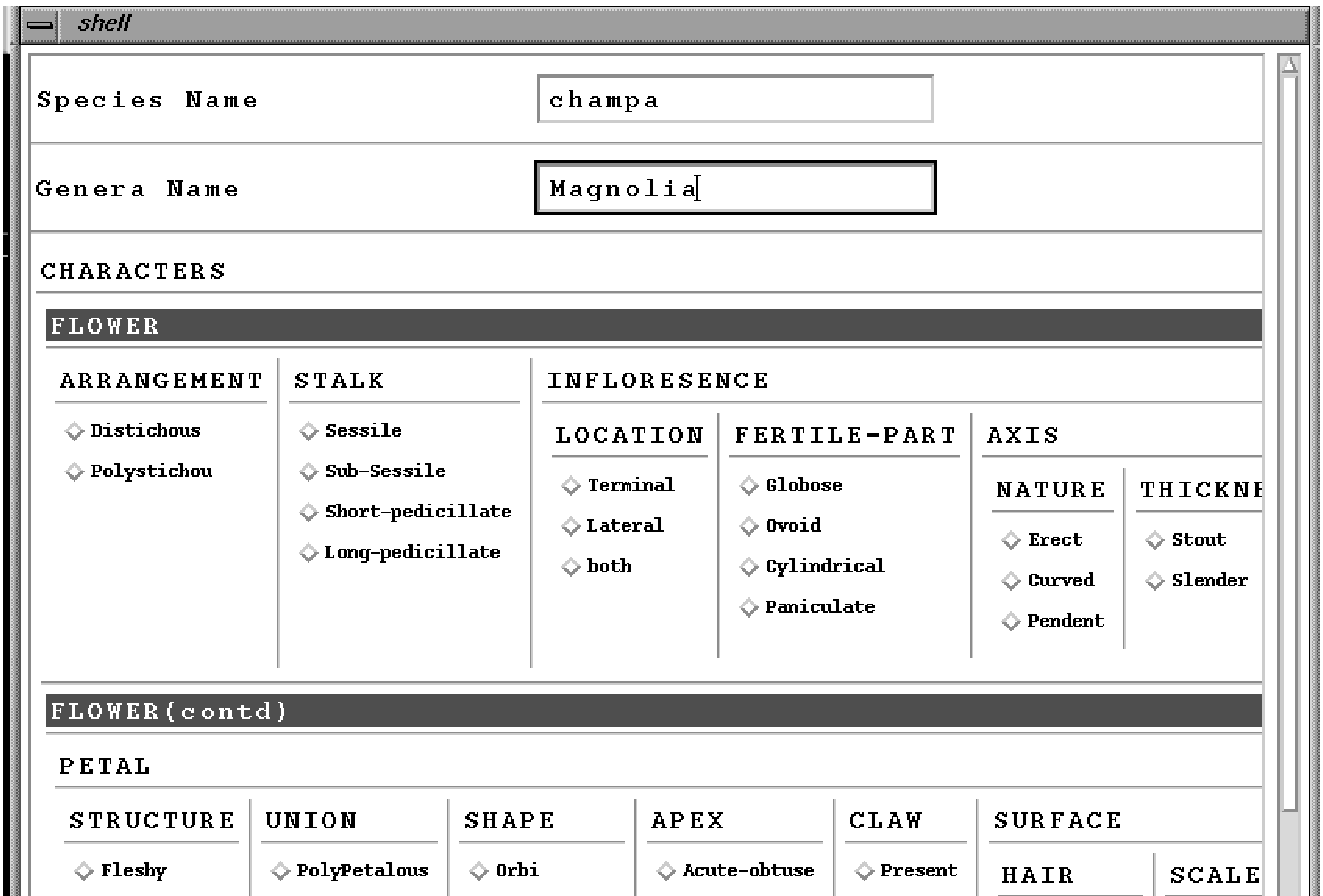


Figure 6.3: A sample query on geographic data

# Chapter 7

# Future Work on OSHADHI

In the last few chapters we have gone through the development process of OSHADHI in detail. This involved various stages like the analysis, data modeling, design and implementation. We discussed a few queries that are processed by OSHADHI. In this chapter, we highlight some of the special features of OSHADHI and also suggest some extensions to OSHADHI.

## 7.1 Special Features of OSHADHI

Since OSHADHI has been built using an object oriented model, we get the benefits of an object oriented approach. One important benefit is *Extensibility*. We shall consider an example to see how extensibility is achieved in OSHADHI. We have about 50 characters that are encoded in the schema of the database. If a new character is to be added in the database, it would require minimal effort since such a character could be derived from the identifying character class. The rest of the application code needs very little alteration.

The graphical user interface is isolated from the database server. This allows the interface to be modified or replaced without having to worry about the database related code. It is also possible to add a formal query language interface between the database server and the graphical interface.

## 7.2 Extensions

- **More Information**

  Although the data model includes several types of information, we have implemented the most important ones in OSHADHI. For example, the temporal records have not been implemented in OSHADHI.

- **A Query Language**

  As mentioned in chapter 5, the main purpose of E is to ease the task of writing internal systems software. However, we have used E as a database programming language. Although E is as powerful as any database programming language, having the query processor built into the database server has the following drawbacks:

  - It limits the types of queries to the few mentioned in Section 3.1.2.
  - It also requires the designer to write code for each type of query that is to be processed.

  An useful extension to OSHADHI would be design a general query language using E's language specification and query optimizer. This would increase the capabilities of OSHADHI.

- **Hypertext**

  The textual information associated with species is currently implemented as simple text that can only be viewed or edited. This could be converted to a hypertext like document which has links within the database. The links could be to any type of object and since it would be within the database, the process of forming the links could be automated, once the user specifies the key words. This could be done by adding code to the text class without affecting the rest of the schema or application code. The link traversal could be added as an operation in the query language.

- **Map Processing**

As mentioned in Section 6.5.1, the data entry for spatial data is manual. This is tedious and error prone. Although our application does not require too much accuracy, it would be useful to automate the data entry process. This would require extracting contours and points from the images, approximating the contours by lines and polygons and finally associating the names with them. More details on map processing are given in [Kast89].

- **Conversion from DELTA Format**

  DELTA, the DEscriptive Language for TAxonomy [Poun95], is a data format designed for use by non-computer specialists working in the area of biological sciences. Since this format is widely used by the biologists, it would be useful to provide an automatic conversion tool from the DELTA format to the OSHADHI database objects. DELTA is a text format that recognizes five main types of character states, namely, unordered multistate, ordered multistate, integer numeric, real numeric and text. The existing DELTA format could be read and the schema could be generated automatically by deriving appropriate classes from the identifying character class. The descriptions of actual plant species can then be converted into appropriate OSHADHI data entry query and processed. For example, the following is a DELTA description of a leaf character:

```
#4. leaf structure/
1. thin/
2. chartaceous/
3. fleshy/
```

  This description would correspond to the following class definition in OSHADHI:

```
dbenum LEAF_STRUCTURE {THIN=1, CHARTACEOUS, FLESHY};
```

```
dbclass leaf : public IdentChar
{
    LEAF_STRUCTURE structure;
};
```

Now, we will consider a species A whose leaf structure is thin. The DELTA format of this description is given below:

```
#Species A/ 4,1
```

The corresponding query in OSHADHI would be:

```
Add species A leaf_structure thin.
```

We have discussed only a simple construct in the DELTA format. However, extending this idea to the other constructs used in the DELTA format would not be difficult.

# Chapter 8

# Conclusions

In this report, we have addressed a problem which many environmental scientists and biologists face - the requirement of a single source of various kinds of biodiversity related data, and a system to manage this data efficiently and process a variety of queries on them. We have designed and developed OSHADHI, an object oriented biodiversity database, with the purpose of meeting the requirement of the researchers and scientists at the *Centre for Ecological Sciences.*

Since traditional database systems fail to meet the modeling and processing requirements of the the biodiversity database, we have adopted an object oriented approach to develop this database.

The modeling and processing requirements of the biological and environmental data were studied and an object model was designed. Extensions to the object model were made by adding classes that would be required by the implementation. The system architecture of OSHADHI has the EXODUS storage manager as the back-end. The database server which processes the queries, is built on top of the storage manager. The server is written in E which is an extension of C++ with additional features such as persistence, iterators and database constructs like **EBeginTransaction()**. All interactions between the user and the database system are through a graphical user interface.

OSHADHI uses access methods like quadtrees (for spatial data) and the pointer intersection technique for fast retrieval. B+tree indexes that are provided by the storage manager of EXODUS are also used.

OSHADHI provides data entry facility for textual data through standard forms and pop-up menus. Queries are also entered through menus and text entry. OSHADHI provides a simple graphical editor that allows the user to enter data and associate spatial data with non-spatial data. Queries on geographic data are entered by clicking on the displayed maps and the menu buttons provided.

The OSHADHI database is mainly composed of the taxonomy data and the geographic data. We have not added the bibliography data and the experts data into our database. However, since the model has been developed for these data types, we expect their addition to OSHADHI to be a simple task. Currently, apart from data entry, OSHADHI can process queries such as identification of a species given its flower, fruit, stem or leaf characters (one or many at a time), listing the various characters of a specific taxon, displaying the map of a country, listing all the species in a specific geographic location, displaying the states in a country where a species is found, etc.

The object model adds features such as *Extensibility* to OSHADHI in terms of being able to add new identifying characters or new types of geographic locations without having to alter much of the application code.

Although the object model which is implemented in OSHADHI is powerful in terms of modeling, the absence of a formal query language limits its capabilities in terms of the types of queries that may be asked. This limitation can be overcome by designing and implementing a formal query language for the application using E and the query optimizer generator that is provided by EXODUS. Other extensions such as incorporating hypertext documents in the database and automating the data entry process of the spatial data were also proposed.

With these extensions provided, we expect OSHADHI to be a good first step in serving the needs of biologists and scientists in the field of environmental sciences.

# Bibliography


[Afsa85] Afsarmanesh, H., et. al., "An Extensible Object Oriented Approach to Databases for VLSI/CAD", *Proceedings of the 1985 VLDB Conference*, Stockholm, August 1985.

[Atki87] Atkinson, M., and Buneman, O.P., "Types and Persistence in Database Programming Languages", *ACM Computing Surveys* 19(2), June 1987.

[Banc92] "Object Oriented Database Systems for Geographic Applications: An experiment with O2", *The Story of O2*, Banchillon, F., Delobel, C., and Kanellakis, P., Eds., Morgan Kaufmann, San Mateo, CA, 1992.

[Bane87] Banerjee, J., et. al., "Data Model Issues for Object Oriented Applications", *ACM Transactions on Office Information Systems* 5(1), January 1987.

[Bato88] Batory, D., et. al., "GENESIS: An Extensible Database Management System", *IEEE Transactions on Software Engineering* 14(11), November 1988.

[Baye72] Bayer, R., "Symmetric Binary B-Trees, Data Structures and Maintenance Algorithms", *Acta Information* 1(4), 1972.

[Bert91] Bertino, E., and Martino, L., "Object Oriented Database Management Systems : Concepts and Issues", *IEEE Computer*, April 1991.

[Booc94] Booch, G., *Object Oriented Analysis and Design with Applications, 2nd Edition*, Benjamin/Cummings, 1994.

[Care86a] Carey, M., et. al., "The EXODUS Extensible DBMS Project : An Overview", Technical report, University of Wisconsin, 1986.

[Care86b] Carey, M., et. al., "Object and File Management in the EXODUS Extensible Database System", *Proceedings of the 1986 VLDB Conference*, Kyoto, Japan, August 1986.

[Care88] Carey, M., De Witt, D., and Vandenberg, S., "A Data Model and Query Language for EXODUS", *Proceedings of the 1988 SIGMOD Conference*, Chicago, IL, June 1988.

[Come79] Comer, D., "The Ubiquitous B-Tree", *ACM Compuing Surveys* 11(2), June 1979.

[DBE87] *Database Engineering* 10(2), Special Issue on Extensible Database Systems, M. Carey, ed., June 1987.

[Decc95] Noronha, F., "Plant Genesis of Western Ghats", *Deccan Herald*, January 22, 1995.

[Deux91] Deux, O., et. al., "The O2 System" *Communications of the ACM* 34(10), October 1991.

[Exod93] "Overview of the Exodus Storage Manager", unpublished, included in the EXODUS storage manager software release, April 1993.

[Fren90] French, J.C., et. al., "Summary of Final Report of NSF Workshop on Scientific Databases", *Sigmod Record* 19(4), December 1990.

[Gunt94] Gunther, O., and Lamberts, J., "Object Oriented Techniques for Geographic Data Management", *The Computer Journal*, 37(1), 1994.

[Gutt84] Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proceedings of the 1984 SIGMOD Conference*, Boston,MA, May 1984.

[Hear90] Hearn and Baker, "GRAPHICS", Wiley Eastern, 1990.

[Ishi93] Ishikawa, H., et. al., "The Model, Language and Implementation of an Object Oriented Multimedia Knowledge Base Management System", *ACM Transactions on Database Systems* 18(1), March 1993.

[Jose91] Joseph, J.V., et. al., "Object Oriented Databases : Design and Implementation", *Proceedings of the IEEE* 79(1), January 1991.

[Kast89] Kasturi, R., et. al., "Map Data Processing in Geographic Information Systems", *IEEE Computers*, December 1989.

[Kim89] *Object Oriented Concepts, Databases and Applications*, Kim, W., and Lochovosky, J., Eds., Addison Wesley, 1989.

[Kim90] Kim, W., "Object Oriented Databases : Definition and Research Directions" *IEEE Transactions in Knowledge and Data Engineering*, September 1990.

[Kort91] Korth, H.F., and Silberschatz, A., "Database System Concepts", Second Edition, McGrawHill, 1991.

[Maie87] Maier, D., and Stein, J., "Development and Implementation of an Object-Oriented DBMS", *Research Directions in Object-Oriented Programming*, Shiver, B., and Wegner, P., Eds., MIT Press, 1987.

[Orac88] *SQL Language Reference Manual, ORACLE RDBMS Version 6.0*, 1988.

[Poun95] Pountain, D., "The Poison Disc", *BYTE*, May 1995.

[Rich87] Richardson, J.E., and Carey, M.J., "Programming Constructs for Database System Implementation in EXODUS", *Proceedings of ACM SIGMOD Conference*, San Fransisco, CA, May 1987.

[Rich89a] Richardson, J.E., Carey, M.J., and Schuh, D., "The Design of the E Programming Language", Computer Sciences Technical Report, University of Wisconsin-Madison, February 1989.

[Rich89b] Richardson, J.E., and Carey, M.J., "Persistence in the E Language: Issues and Implementation", *Software - Practise and Experience* 19(12), December 1989.

[Rowe87] Lawrence A. Rowe and Michael R. Stonebraker, "The Postgres Data Model" *Proceedings of the 1987 VLDB Conference*, Brighton, England, August 1987.

[Rumb91] Rumbaugh, J., et. al., *Object Oriented Modeling and Design*, Prentice Hall, 1991.

[Same89] Samet, H., *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, 1989.

[Samp85] *Proceedings of the Discussion Meeting on Development of Scientific Databases in India*, Sampath, D., Balakrishnan, N., Rajaraman, V., Eds., 1985.

[Stro91] Stroustrop, B., *The C++ Programming Language, 2nd edition*, Addison-Wesley, 1991.

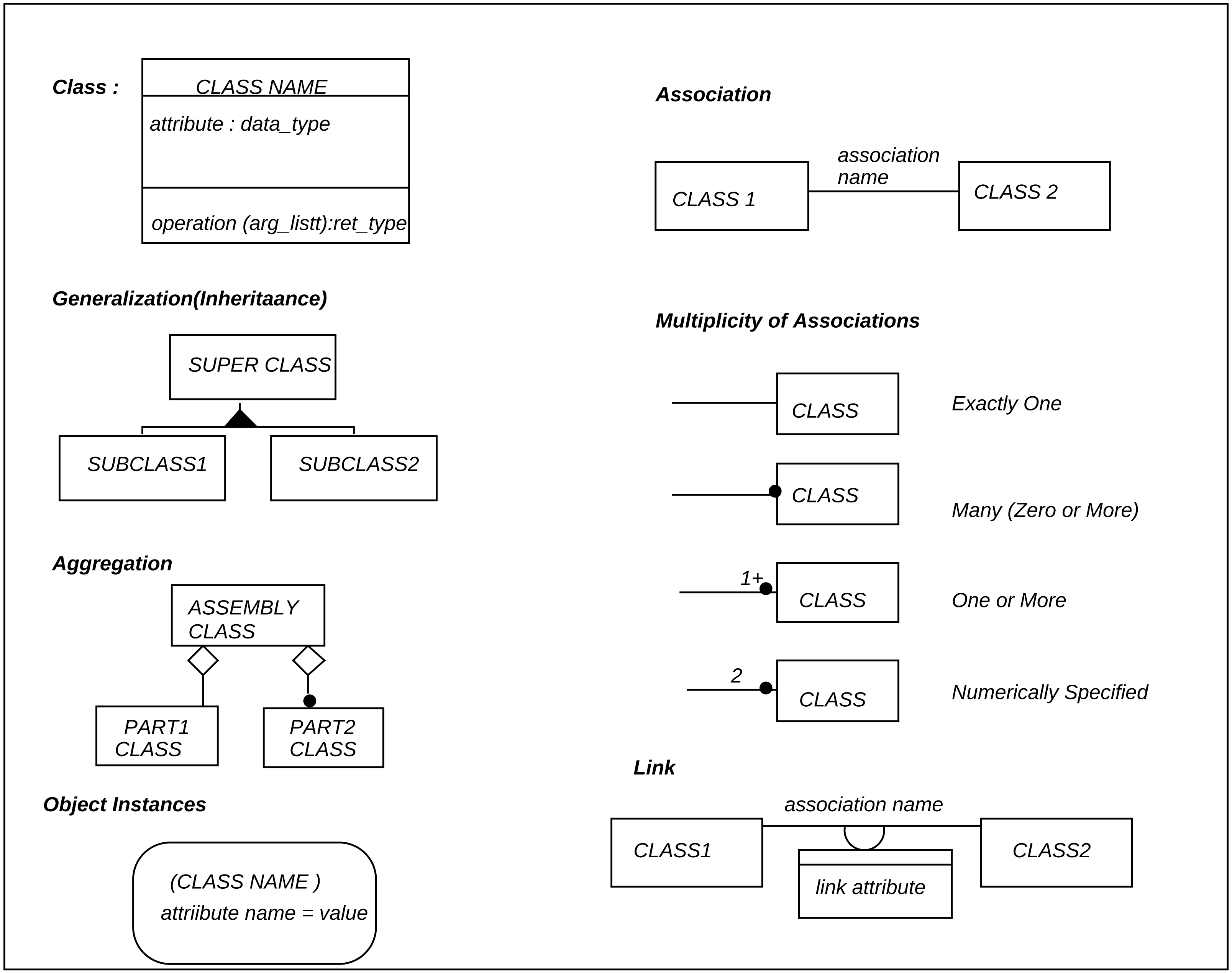
[Vasi74] Vasishta, P.C., *Taxonomy of Angiosperms*, S. Chand and Co., Delhi, 1974.

[Woel86] Woelk, D., "An Object Oriented Approach to Multimedia Databases", *Proceedings of the 1987 VLDB Conference*, Brighton, England, August 1987.

[Youn90] Young. D.A., *The X Window System* Englewood Cliffs: Prentice Hall, 1990.

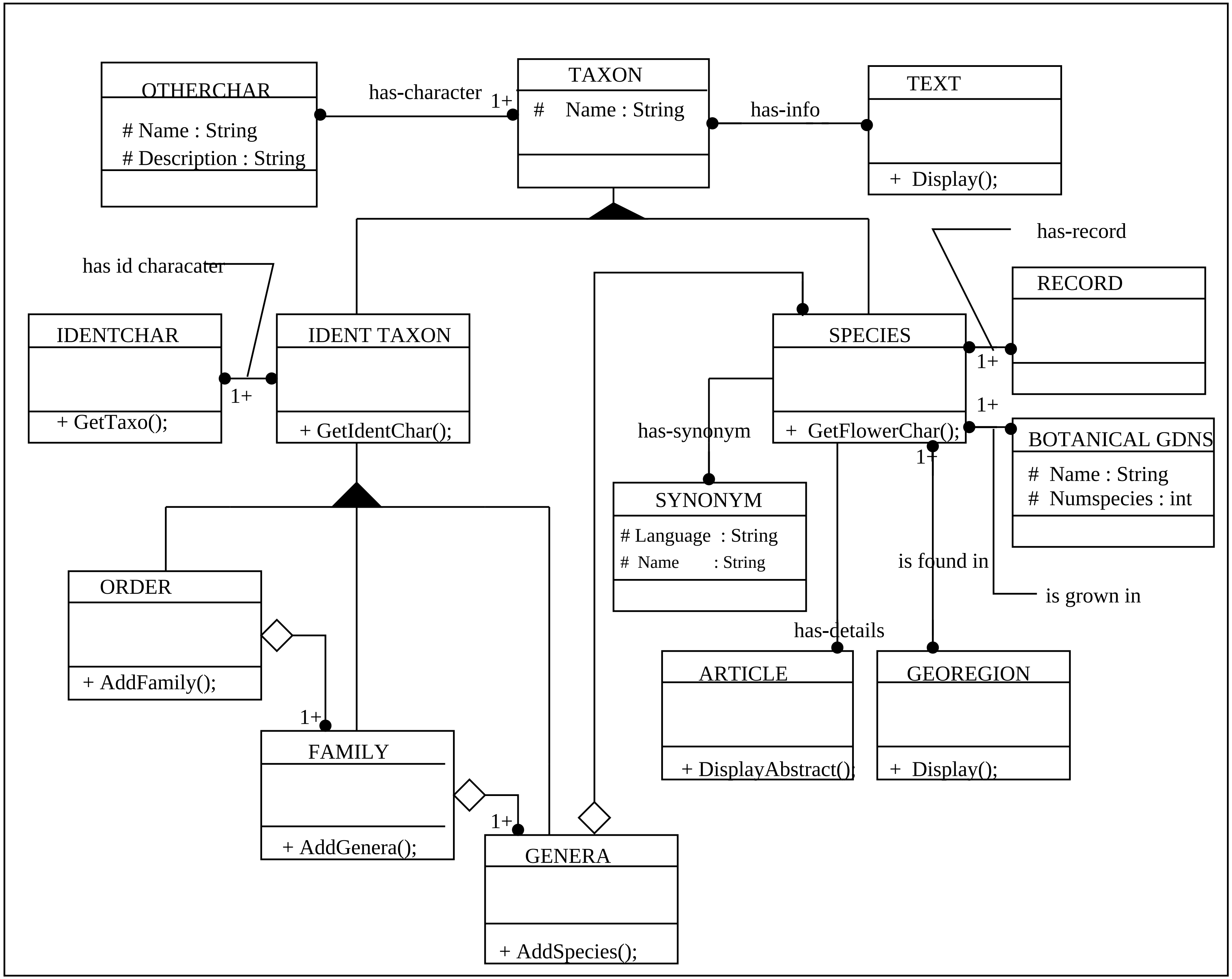
# Appendix A

# The Rumbaugh Notation for Object Model

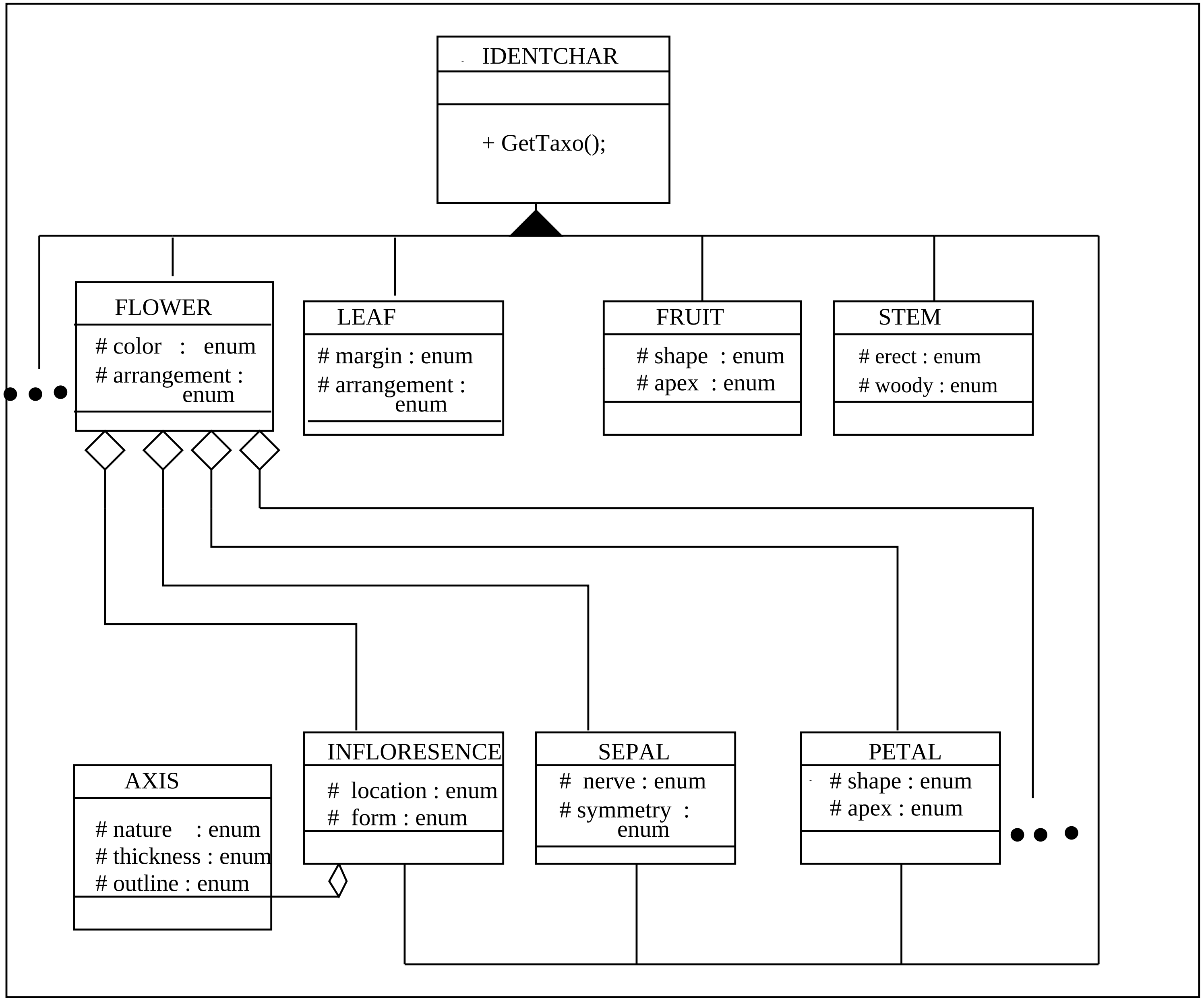

Class :
CLASS NAME
attribute : data_type
operation (arg_listt):ret_type
Association
association
name
CLASS 1
CLASS 2
Generalization(Inheritaance)
SUPER CLASS
SUBCLASS1
SUBCLASS2
Multiplicity of Associations
CLASS
Exactly One
CLASS
Many (Zero or More)
1+
CLASS
One or More
2
CLASS
Numerically Specified
Aggregation
ASSEMBLY
CLASS
PART1
CLASS
PART2
CLASS
Link
association name
CLASS1
CLASS2
link attribute
Object Instances
(CLASS NAME )
attriibute name = value

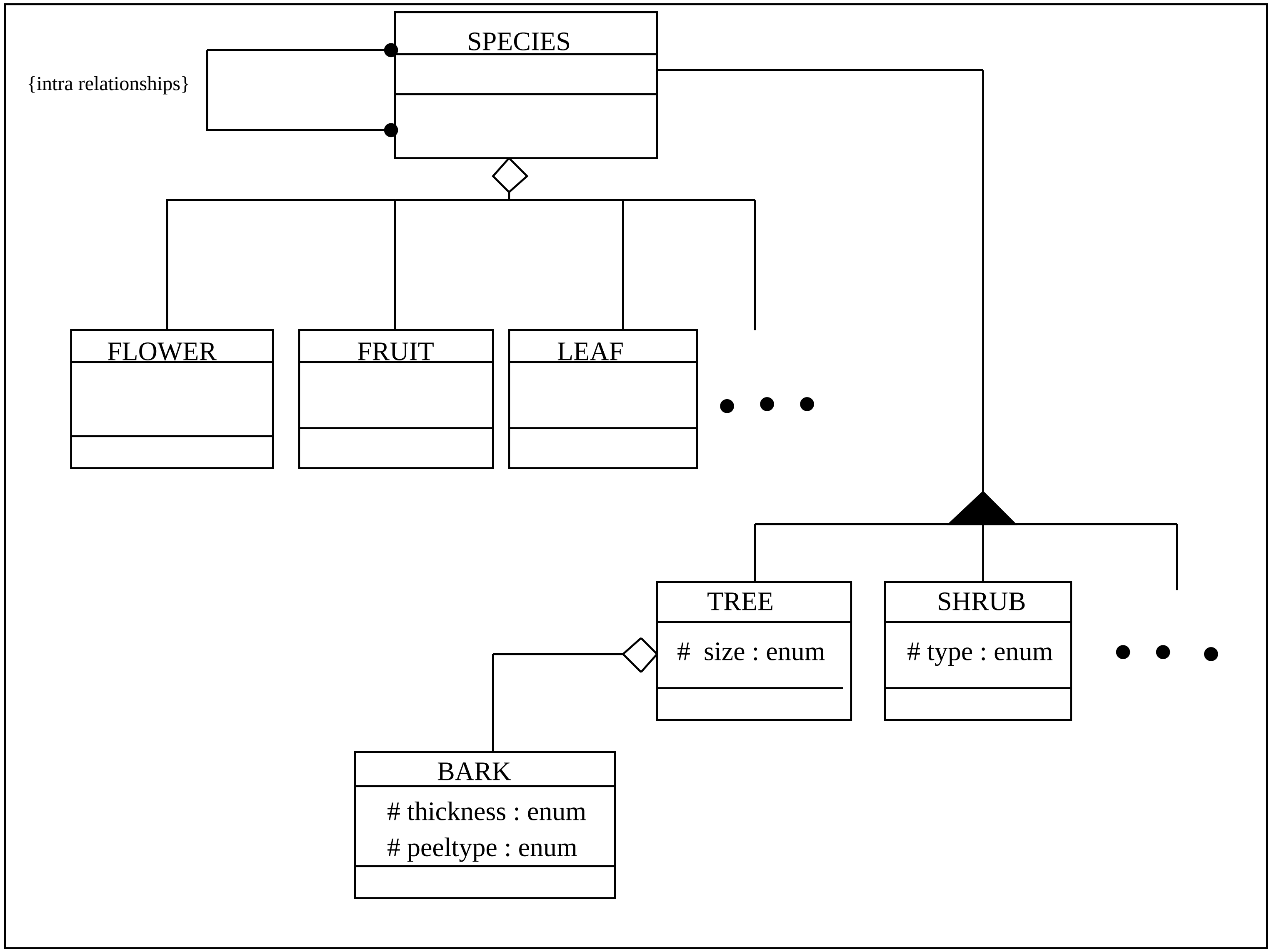
# Appendix B

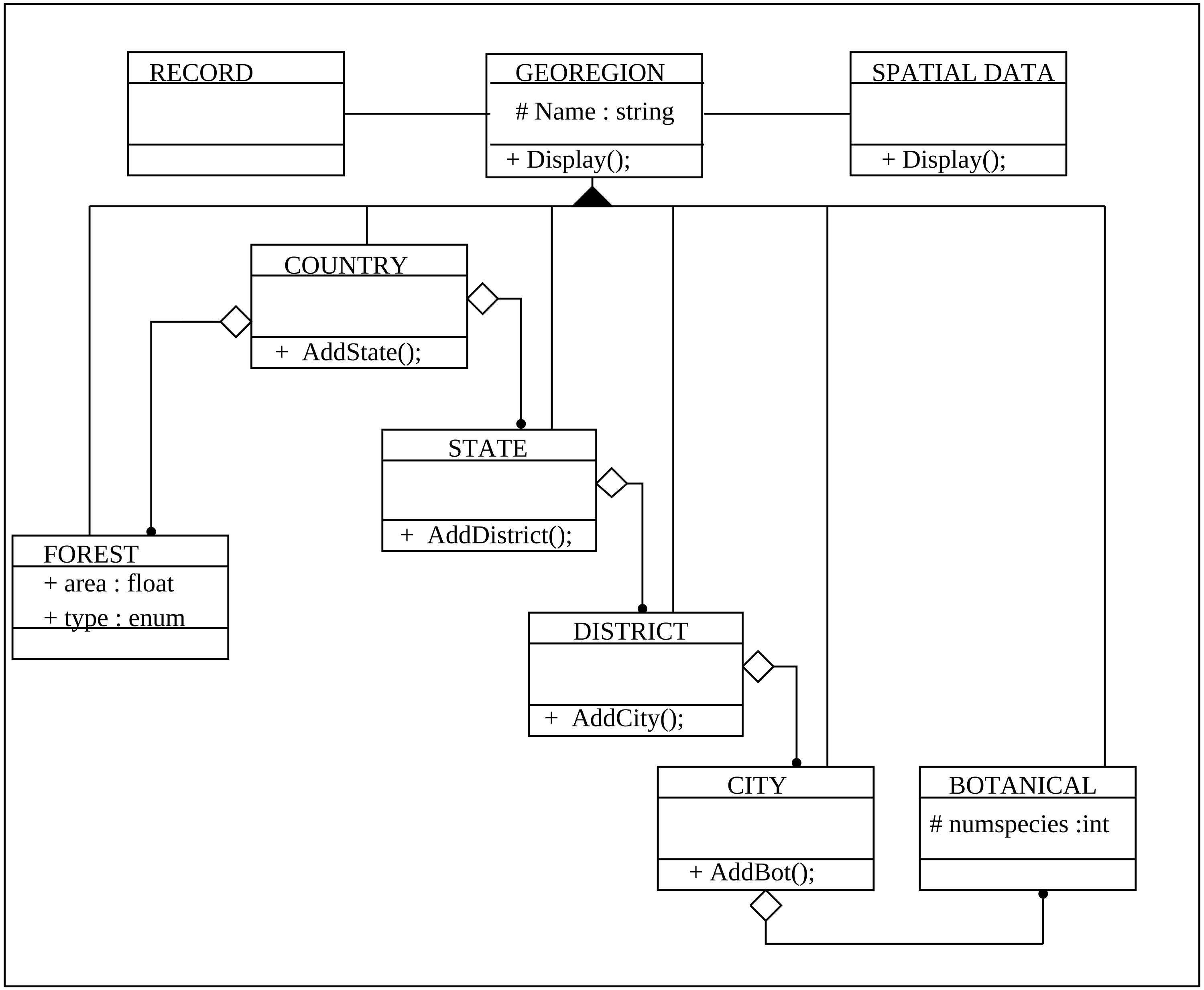
# The Data Model of OSHADHI

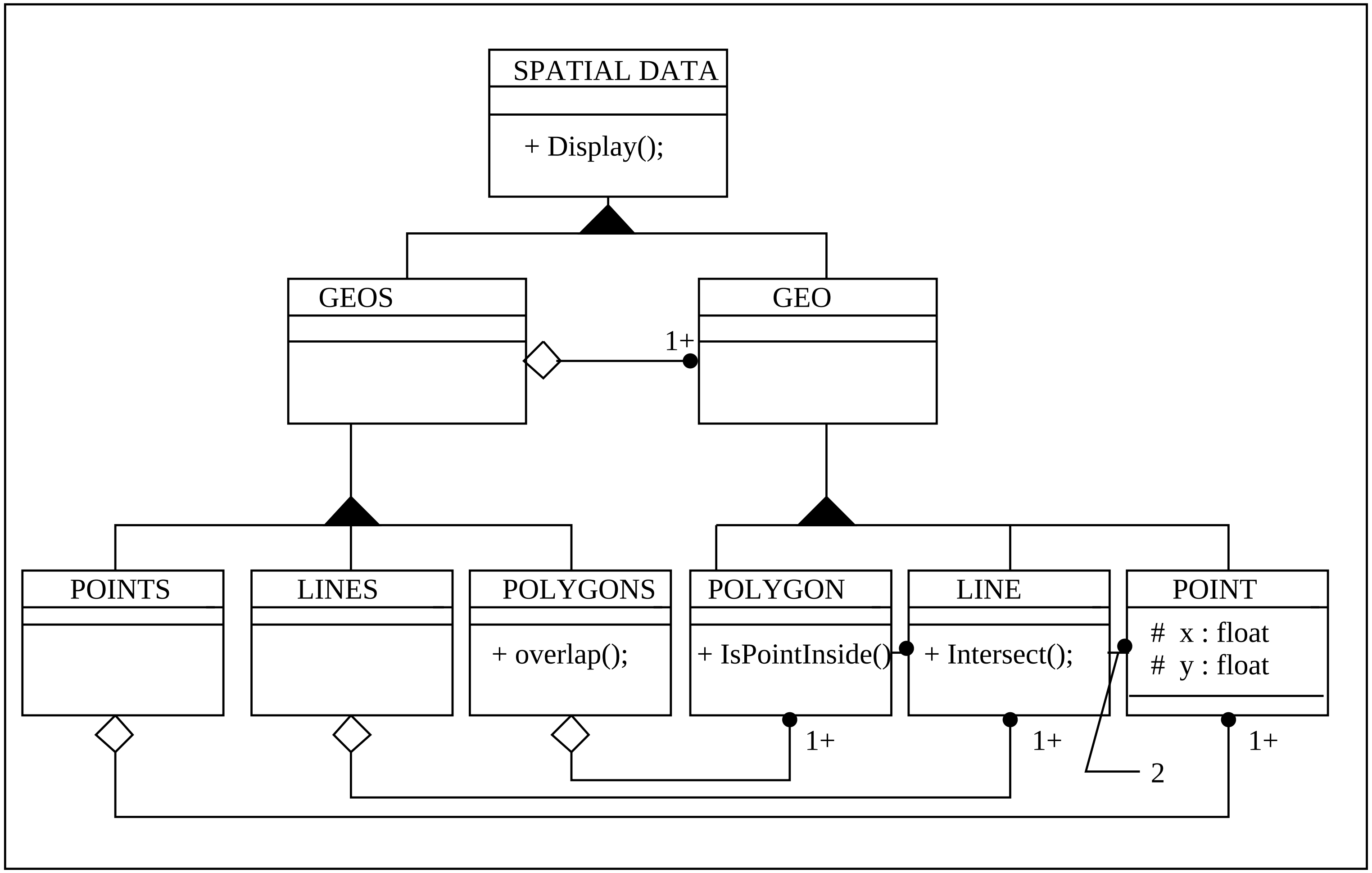


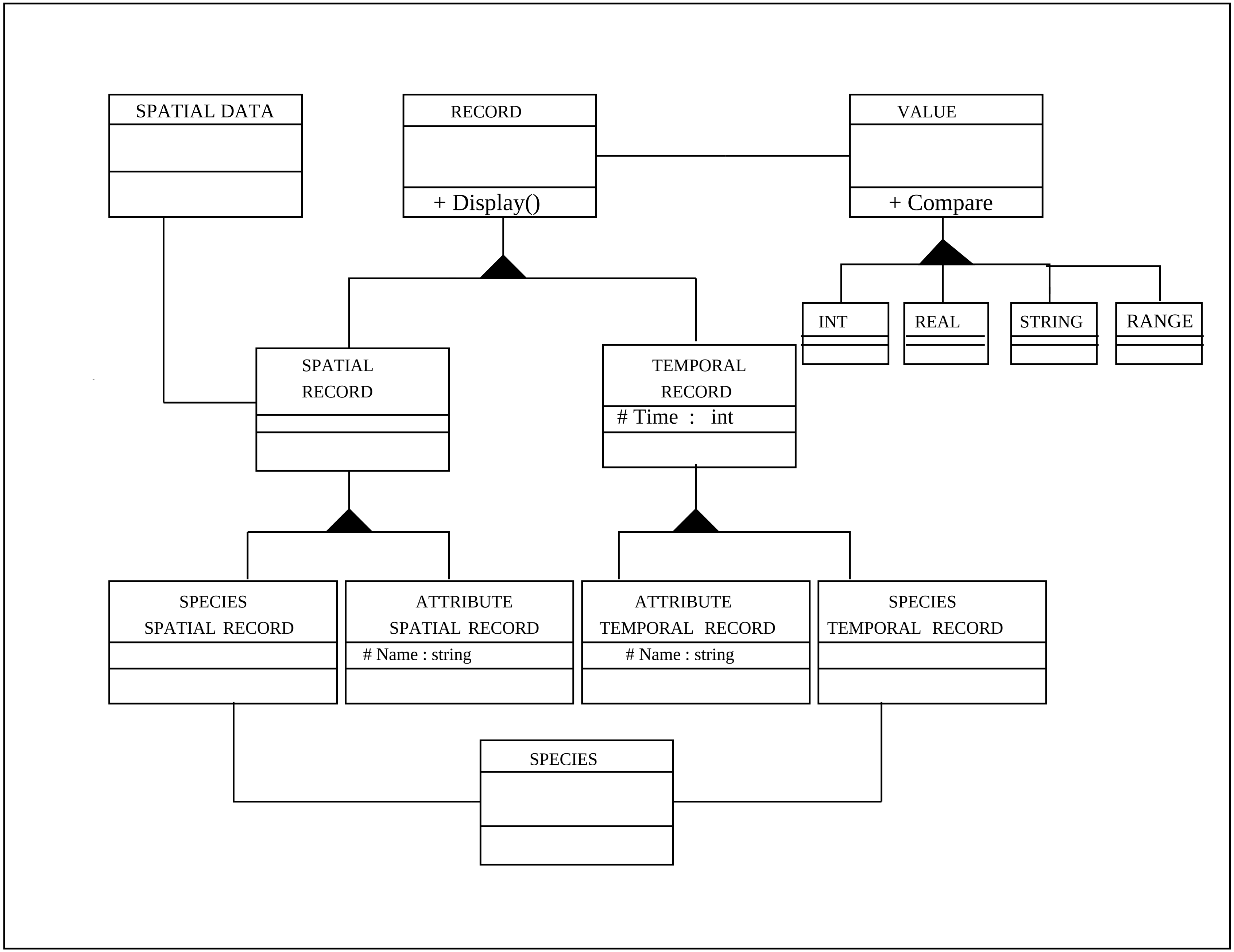
OTHERCHAR
# Name : String
# Description : String
has-character
1+
TAXON
# Name : String
has-info
TEXT
+ Display();
has-record
has id characater
RECORD
IDENTCHAR
+ GetTaxo();
1+
IDENT TAXON
+ GetIdentChar();
SPECIES
+ GetFlowerChar();
1+
1+
BOTANICAL GDNS
# Name : String
# Numspecies : int
has-synonym
1+
SYNONYM
# Language : String
# Name : String
is found in
is grown in
ORDER
+ AddFamily();
has-details
ARTICLE
+ DisplayAbstract();
GEOREGION
+ Display();
1+
FAMILY
+ AddGenera();
1+
GENERA
+ AddSpecies();

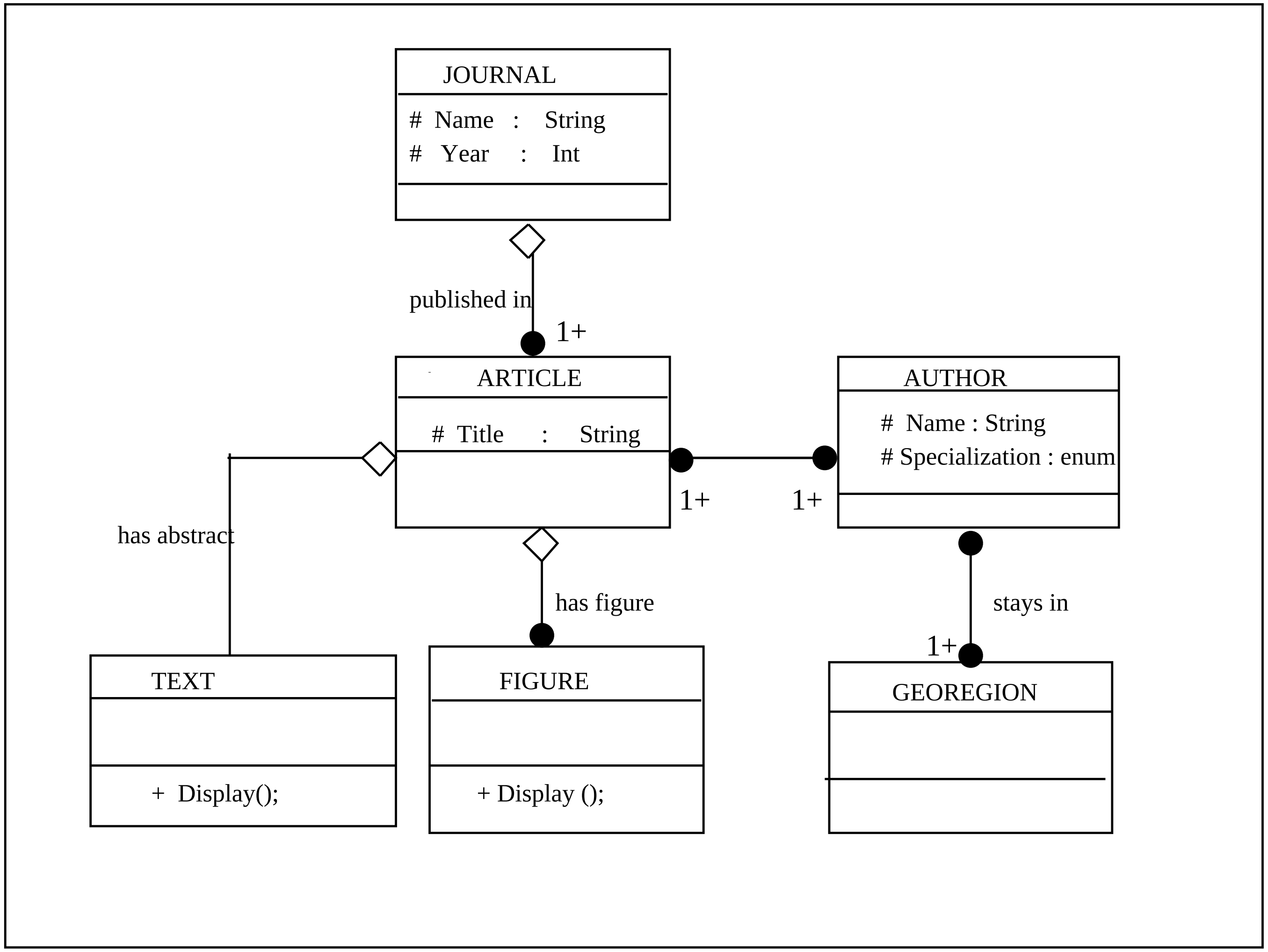

IDENTCHAR
+ GetTaxo();
FLOWER
# color : enum
# arrangement : enum
LEAF
# margin : enum
# arrangement : enum
FRUIT
# shape : enum
# apex : enum
STEM
# erect : enum
# woody : enum
AXIS
# nature : enum
# thickness : enum
# outline : enum
INFLORESENCE
# location : enum
# form : enum
SEPAL
# nerve : enum
# symmetry : enum
PETAL
# shape : enum
# apex : enum


SPECIES
{intra relationships}
FLOWER
FRUIT
LEAF
TREE
# size : enum
SHRUB
# type : enum
BARK
# thickness : enum
# peeltype : enum


RECORD
GEOREGION
# Name : string
+ Display();
SPATIAL DATA
+ Display();
COUNTRY
+ AddState();
STATE
+ AddDistrict();
FOREST
+ area : float
+ type : enum
DISTRICT
+ AddCity();
CITY
+ AddBot();
BOTANICAL
# numspecies :int


SPATIAL DATA
+ Display();
GEOS
GEO
1+
POINTS
LINES
POLYGONS
+ overlap();
POLYGON
+ IsPointInside()
LINE
+ Intersect();
POINT
# x : float
# y : float
1+
1+
1+
2


SPATIAL DATA
RECORD
+ Display()
VALUE
+ Compare
INT
REAL
STRING
RANGE
SPATIAL RECORD
TEMPORAL RECORD
# Time : int
SPECIES SPATIAL RECORD
ATTRIBUTE SPATIAL RECORD
# Name : string
ATTRIBUTE TEMPORAL RECORD
# Name : string
SPECIES TEMPORAL RECORD
SPECIES


JOURNAL
# Name : String
# Year : Int
published in
1+
ARTICLE
# Title : String
AUTHOR
# Name : String
# Specialization : enum
1+
1+
has abstract
has figure
stays in
1+
TEXT
+ Display();
FIGURE
+ Display ();
GEOREGION

# Appendix C

# Definitions of Classes in OSHADHI

The following are the class definitions of some of the major classes in the design of OSHADHI. The definitions are in the E language.

```
/* The Indexed Collection Class */

template<dbclass T> dbclass i_collection
{
    collection<T> coll ;
    StrTypedBtreeIndex<T> coll_index ;

     public :
         T * Add (STRING s) ;
         void Delete (STRING s);
         void Modify (STRING s, const T & el) ;
         T* Search ( STRING  s);
         ITER_DECL ( T*, GetNext (void)) ;
};

/* The encoded Collection Class */
```

```
template<dbclass T> dbclass e_collection
{
       collection<T>  hcoll ;
       StrTypedBtreeIndex<T> hcol_index;

      public :
          T * Add ( const T& el);
          void Delete ( T * el);
          T * Search ( STRING s);
          ITER_DECL ( T *, GetRange (char *, char *)) ;
};

/* The String B-tree Index Class */

template <dbclass T> dbclass strindex
{
       StrTypedBtreeIndex<T> n_index ;

       public :
            void insert ( STRING s, T * objptr );
            void remove ( STRING s, T * objptr);
            T * retrieve ( STRING s);

};

template<dbclass T> dbclass ptr_collection
{
      collection<T *> coll ;
```

```
      StrTypedBtreeIndex<T *>  coll_index  ;

      public :
          void Add ( T * pt);
          void Delete ( const T * pt );
          T ** Search ( const T * pt ) ;
          int IsEmpty (void );
          ITER_DECL (T*, IndGetNext(void)) ; /* generates in sorted order*/

};

dbclass TaxonomyLevel
{
      protected :
       dbchar Name[MAXLENGTH] ;
       i_collection<SpecialChar> SpecialCol ;
       Text  Text_info ;
       ptr_collection<Article> ArtCol ;

      public :
          TaxonomyLevel ( char *) ;
          void AddSpecialChar ( STRING );
          void DeleteSpecialChar (STRING);
          void AddText (char *) ;
          ITER_DECL ( Word *, GetTextWord());
          void AddArticle (const Article *) ;
          void DeleteArticle (STRING ) ;
          char * GetName () const;
};
```

```
dbclass IdentLevel : public TaxonomyLevel
{
         ptr_collection<IdentChar> IdentCol ;

         public :
            void AddIdentChar (IdentChar *);
            void DelIdentChar ( IdentChar *);
            ITER_DECL ( IdentChar *, GetIdentChar()) ;
};

dbclass Species : public TaxonomyLevel
{
         ptr_collection<GeoRegion> Locations ;
         ptr_collection<Botanical> Botanicals ;
         ptr_collection<Record>  Records ;
         strindex<Synonym>      Synonyms ;
         FlowerChar *    Flower ;
         FruitChar  *      Fruit ;
         LeafChar   *      Leaf ;
         SeedChar   *      Seed ;
         Genera *         genera ;
 ...

         public :
           Species ( char *) ;
           Genera * GetGenera ();
           void AddLocation ( GeoRegion *);
           void DeleteLocation (STRING );
```

```
          void AddBotanical ( Botanical *);
          void DeleteBotanical ( STRING );
          void AddFlowerChar (const FlowerChar &);
          void DeleteFlowerChar () ;
          FlowerChar * GetFlowerChar (void) const
   . . .

};

dbclass Genera : public IdentLevel
{
        set<Species> SpeciesCol ;
        Family * family ;

        public :
          Species * AddSpecies ( STRING);
          Family * GetFamily() { return family ;}
          void DeleteSpecies ( STRING );
          ITER_DECL ( Species *, GetSpecies()) ;
};

dbclass Family : public IdentLevel
{
        set<Genera> GeneraCol ;
        Order * order ;

        public :
           Genera * AddGenera ( STRING );
           Order * GetOrder();
```

```
            void DeleteGenera ( STRING );
            ITER_DECL ( Genera *, GetGenera());
};

dbclass Order :public IdentLevel
{
         set<Family>  FamilyCol ;

         public :
             Family * AddFamily ( STRING);
             void DeleteFamily ( STRING);
             ITER_DECL ( Family *, GetFamily()) ;
};

dbclass IdentChar
{
    ptr_collection<TaxonomyLevel>  taxo ;

    public :
        int NumPtrs ;  // for avoiding dangling references
        IdentChar ();
        ITER_DECL ( TaxonomyLevel *, GetTaxo()) ;
        virtual char *EncodedValue (void ) const =0 ;
        int  NotPointedTo (void );
};

dbclass SepalChar : public IdentChar
{
```

```
   public :
     NERVECHAR  nerves ;
     SYMMETRYCHAR symm ;
     FORM  form ;
     FUSIONCHAR fusion ;

     SepalChar();
     char * EncodedValue() const;
     ...

};

/* other identifying characters are not listed here */

dbclass TaxonomyDatabase
{
      i_collection<Order>  orders ;
      strindex<Family>  family_index ;
      strindex<Genera>  genera_index ;
      strindex<Species> species_index ;

      public :
         Order  * SearchOrder  ( STRING );
         Family * SearchFamily ( STRING);
         Genera * SearchGenera (STRING);
         Species * SearchSpecies (STRING);
         Order *  AddOrder ( STRING );
         void DeleteOrder ( STRING );
         Family * AddFamily (STRING,STRING );
```

```
         void DeleteFamily ( STRING );
         Genera * AddGenera ( STRING,STRING );
         Species * AddSpecies ( STRING ,STRING );
         void DeleteGenera ( STRING );
         void DeleteSpecies ( STRING );
         ITER_DECL ( Order *, GetOrder()) ;
};

dbclass CharDatabase
{
      e_collection<FlowerChar>  Flowers ;
      e_collection<FruitChar> Fruits ;
      e_collection<LeafChar> Leaves ;
      e_collection<SeedChar> Seeds ;
      ...

      public :
         void DeleteChar (IdentChar *) ;
         IdentChar * AddChar (IdentChar *) ;
         IdentChar * GetChar (IdentChar *);
 ITER_DECL ( IdentChar *, GetRange (char *, char *));
};

dbclass Spatial
{
    public :
       virtual void Display() const;
};
```

```
dbclass Geo : public Spatial
{
    public :
       virtual void Display (void ) const = 0;
};

dbclass Geos : public Spatial
{
     collection<Geo *> geocol ;

     public :
         void Display(void) const;
         void AddGeo (Geo *);

};

typedef float Longitude;
typedef float Latitude ;

dbclass Point : public Geo
{
      Longitude longi ;
      Latitude lat ;

      public :
        Point & operator=(const Point & );
        Point ( Longitude, Latitude);
        Point ();
        Latitude Lat();
```

```
        Longitude Long();
        void Display () const;
};

dbclass Line : public Geo
{
       Point p1 ;
       Point p2 ;
       float slope ;
       float intercept ;
       Boolean LieOppositeSide ( const Point &, const Point &) const;
       float ValWhenSubstituted (const Point &) const;

      public :
         Point & operator=(const Line & );
         Line (Point, Point );
         Boolean DoesIntersect ( const Line &) const;
         void Display() const;
};

dbclass Polygon : public Geo
{
      collection<Line>  lines ;
      void AddLine (Line);
      float minlong ;
      float maxlong ;
      float minlat ;
      float maxlat ;
```

```
     public :
         Polygon ( ElemList<Line *> &) ;
         Point GetXIntervalRep(void);
         Point GetYIntervalRep (void);
         Boolean IsPointInside (const Point &) const;
         void Display() const ;
};

dbclass Polygons : public Geos
{
    collection<Polygon *> geocol ;

    public :
       Polygons Overlap (Polygons &);
};

dbenum Quadrant { SW, NW, SE, NE };

template <dbclass T> dbclass QuadTreeNode
{
       float x,y ;
       Point pt ;
       QuadTreeNode<T> * SWNode ;
       QuadTreeNode<T> * NWNode ;
       QuadTreeNode<T> * SENode ;
       QuadTreeNode<T> * NENode ;
       T * ElemPtr ;

       Quadrant PtCompare ( Point p );
```

```
       int Equal ( Point p, int margin );
       QuadTreeNode ** Child ( Quadrant q );
       int FindRegion (float pt);

      public :
          QuadTreeNode(Point p1, T * el )
          void Insert ( QuadTreeNode * newNode );
          T * Search ( Point p1, int margin );
          ITER_DECL (T *, SearchPoint (float));
};

#define  qtn QuadTreeNode<T>

template <dbclass T> dbclass QuadTree
{
      collection <T> allNodes ;
      QuadTreeNode<T>  * root ;

      public :
        QuadTree() { root = NULL ;};
        T * Search (Point, int );
        void Insert ( Point pt, T *EntityPtr);
        ITER_DECL ( T *, SearchPoint(float));
};

dbclass GeoRegion
{
      Spatial * spat ;
      StrTypedBtreeIndex<AttrRecord> attrindx ;
```

```
      ptr_collection<TaxonomyLevel>  taxcol ;
      Text txt ;
      dbchar Name[MAXLENGTH] ;
      GeoRegion * par ;

     protected :
      QuadTree<GeoRegion> Xindex ;
      QuadTree<GeoRegion> Yindex ;

     public :
         GeoRegion ( char * sname );
         GeoRegion * SearchPoint (Point);
         void AddRecord (AttrRecord *);
         ITER_DECL ( AttrRecord *, GetRec(char *name ));
         virtual void UpdateIndex (const GeoRegion *);
         virtual int Type() {return ERRORNO ;};
         void AddTaxon (TaxonomyLevel * taxon) ;
         ITER_DECL ( TaxonomyLevel *, GetTaxon());
         virtual void Display () ;
         void AddSpatial( Spatial *);
         Spatial * GetSpatial () const;
         char * GetName() const;
};

dbclass Country : public GeoRegion
{
      collection <State> states ;

     public :
```

```
       Country ( char *s );
       void Display() ;
       void UpdateIndex(const GeoRegion *);
       State * AddState (const State &);
       ITER_DECL (State *, GetState());
};

/* The class definitions of City, State etc. are similar to Country
   and hence we have not listed them */

dbclass Value
{
        public :
          virtual int compare (Value *) =0;
          virtual void Print () =0;
};

dbclass IntVal : public Value
{
        int val ;

        public:
          IntVal (int v);
          int compare (Value *v);
          void Print ();
};

dbclass FloatVal : public Value
{
```

```
        float val ;

        public:
        FloatVal(float v) {val = v;};
        int compare (Value *v );
        void Print ();
};

dbclass StringVal : public Value
{
       dbchar val[MAXLENGTH] ;

       public:
       StringVal (char *s);
       int compare (Value * s);
       void Print();
};

dbclass RangeVal : public Value
{
       Value * valmin ;
       Value * valmax ;

       public :
          RangeVal (Value *v1, Value *v2 );
          int compare ( Value * val);
          void Print ();
};
```

```
dbclass AttrRecord
{
        protected :
         Value *val ;
         dbchar attrname[MAXLENGTH] ;

         public :
           Value * GetVal();
           char *GetName();
           AttrRecord (char *n, Value *v );
};

dbclass SpatialAttrRecord : public AttrRecord
{
          Spatial * spat ;

          public:
            Spatial * GetSpatial();
            void Print();
            SpatialAttrRecord(char *n, Value *v,Spatial *sp);
};

dbclass  SpatialDatabase
{
      collection<Point> pointdb ;
      collection<Polygon> polygondb ;
      collection<Line> linedb ;
      collection<Points> pointsdb ;
      collection<Lines> linesdb ;
```

```
      collection<Polygons> polygonsdb ;
      collection<Geos> geosdb ;

      public :
          Point * GetPoint ( Latitude , Longitude);
          Line * GetLine ( Point, Point );
          Polygon * GetPolygon ( ElemList<Line *> &) ;
          Points * GetPoints (void);
          Lines * GetLines (void);
          Polygons * GetPolygons (void);
          Geos * GetGeos (void);
};

dbclass  RecDatabase
{
         collection <SpatialAttrRecord> records ;
         collection <IntVal> ivalues ;
         collection <FloatVal> fvalues ;
         collection <StringVal> svalues ;
         collection <RangeVal>  rvalues ;

         public :
            SpatialAttrRecord * GetSpatialRecord (char * , Value *, Spatial *);
            IntVal * GetIval( int ) ;
            FloatVal * GetFval (float);
            StringVal * GetSval (char *);
            RangeVal * GetRval ( Value *, Value *);
};
```

```
dbclass  GeoDatabase
{
         collection <Country> countries ;

         strindex <Country> country_index ;
         strindex <State> state_index ;
         strindex <District> district_index ;
         strindex <City> city_index ;

        public :
           Country * SearchCountry (STRING);
           State * SearchState (STRING );
           District * SearchDistrict(STRING);
           City * SearchCity (STRING );

           Country * AddCountry (STRING);
           State * AddState (STRING, STRING);
           District * AddDistrict (STRING,STRING);
           City * AddCity (STRING, STRING);

           GeoRegion * SearchGeoRegion ( STRING );
           ITER_DECL ( Country *, GetCountry());
};
```