A GIS Editor for a Database Programming Language

YuLing Chen
School of Computer Science
McGill University, Montreal
March 2001

A Thesis Submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of Master of Science in Computer Science

Abstract

Geographical Information Systems (GIS) have become a more and more important application of database systems. Most general-purpose database systems do not contain a graphical display interface which is indispensable in GIS applications. This thesis presents the design and implementation of a GIS editor (Geditor) for a relational database programming language. It builds a graphical map display interface into the database language and integrates a set of GIS functions.

Two interfaces are built in Geditor. One is with the database programmer and the other is with the GIS End-User. The former interface implements a new syntax (gedit) into the database language for the database programmer to call and display the Geditor GUI. The latter implements a GUI with the GIS End-User to view the map and perform a series of fundamental GIS functions.

Geditor stores both spatial and non-spatial data in the relational database. The implementation utilizes the spatial capabilities of the relational database programming language to the largest extent. This demonstrates the feasibility and the simplicity of implementing GIS applications in an integrated approach using relational databases. It also provides a flexible and extendable framework by designing an extendable syntax and utilizing the event handler mechanism which is the characteristic of active databases. Java, especially the JFC Swing package is used extensively in the implementation.

Résumé

Les Systèmes d'Information Géographiques (SIG) deviennent des applications de plus en plus importantes des systèmes de bases de données. La plupart des systèmes de bases de données n'ont pas besoin d'une interface graphique utilisateur qui est pourtant indispensable pour les applications des SIG. Cette thèse présente la conception et la réalisation d'un éditeur de SIG (Geditor) pour un langage de programmation de bases de données relationnelles. Celui-ci intègre une interface d'affichage graphique de cartes ainsi que certaines fonctionnalités des SIG au langage de bases de données.

Deux interfaces sont construites pour le Geditor. Une est pour le programmeur de bases de données et l'autre pour l'utilisateur du SIG. La première interface implante une nouvelle syntaxe (gedit) dans le langage de base de données pour que le programmeur puisse appeler et afficher le GUI du Geditor. La dernière implante un GUI pour l'utilisateur du SIG pour qu'il puisse voir la carte et utiliser une série de fonctionnalités fondamentales des SIG.

Le Geditor conserve les données spatiales et non-spatiales dans la base de données relationnelle. L'implantation utilise au maximum de ses possibilités les capacités du langage de programmation à traiter les données purement spatiales. Cette thèse démontre le faisabilité et la simplicité de réaliser des applications de SIG avec une approche intégrée utilisant les bases de données relationnells. Elle apporte aussi un cadre flexible et extensible en concevant une syntaxe extensible et en utilisant un mécanisme de gestion d'événements qui est la caractéristique des bases de données actives. Le langage Java, et plus particulièrement le paquetage JFC Swing, sont utilisés de façon extensive dans cette implantation.

Acknowledgements

First and foremost, I wish to express my gratitude to my supervisor, Professor Tim H. Merrett, for his attentive guidance, invaluable advice, and continuous encouragement throughout the research and preparation of this thesis. His careful reading and constructive criticism makes this thesis better and better. In addition, I am also grateful for his generous and constant financial support during this program.

I would like to thank Ian Garton, an officemate and friend, who helped me a lot in lab operations and latex commands. I would also like to thank WeiZhong Sun for providing great information and help in the jRelix implementation, especially the Event Handlers. I am also grateful to all the secretaries and system staff for their continuous administrative help and technical assistance. Special mention should be made of Lise Minogue, Franca Cianci, Lucy St-James, Teresa De Angelis, Andrew Bogecho, and Philippe Ciaravola.

Special thanks to Julien Mazloum, who translated the abstract to French. I also thank Delize Williams, who did a great job in proofreading the thesis.

Thanks must go to my parents and brother for their endless love, support, and encouragement. Without their guidance, teachings, and advices throughout my life, this thesis would not have been possible.

Finally, I would like to send my appreciation to my husband, Hao Wu, for his love, encouragement, and support during my studies.

To my parents, *HongMing Chen* and *Ying Zhang*,
for always encouraging me to pursue higher education

and to my husband, *Hao Wu*,
for studying with me and sharing the joy together.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

GIS has developed rapidly in the past two decades. It integrates spatial and non-spatial data into one system and provides powerful tools and various operations to deal with these data. Since spatial data are usually large-scale, this makes it inevitable for GISs to use a database to manage the data. This thesis presents the implementation of a GIS application (Geditor) which is focused on utilizing database capabilities to build an independent GIS application.

Relational database systems, which provide mechanisms for managing data, achieve great success in the commercial world. Most GISs also adopt a relational database system to manage the data. Based on whether the spatial data are stored in the database management system (DBMS) or not, the data management of current GISs is divided into two categories: the integrated approach and the hybrid approach [Wor99]. In the integrated approach, the GIS puts all the data including both spatial and non-spatial (descriptive) data in the relational database. In the hybrid approach, however, only the descriptive data are stored the relational database. The benefits of using an integrated architecture are obvious. Since DBMS treats all data uniformly, in the integrated architecture, the spatial data are treated equally with the descriptive data. Furthermore, a professional DBMS enforces the integrity, concurrency and security of the data, which is another advantage of using integrated architecture. However, because SQL lacks the expressive power for spatial queries and because of performance issues, the integrated approach is not widely adopted [Wor99].

However, the integrated approach is feasible when an advanced general-purpose relational database is adopted. This thesis presents an implementation of a GIS application (Geditor) which stores both spatial and non-spatial data in jRelix — a relational database programming language developed at the School of Computer Science, McGill University. jRelix contains a DBMS and a programming language named Aldat. This Geditor implements a graphical display interface which becomes a new component of jRelix and utilizes the programming capability of Aldat to complete the implementation of GIS functions.

With this implementation, users will enter or import data into jRelix format, display the map graphically using Geditor, and perform GIS operations such as thematic mapping, spatial query, and layer control.

It is the aim of this thesis to demonstrate the feasibility of using a relational database programming language to implement an independent GIS application. Martinez [Mar98] has already proved the Aldat capability of spatial analysis which is the basic requirement for developing GIS applications. However, this implementation has to export spatial analysis results into other software packages to display the map because jRelix did not provide a graphical display interface. This is not satisfactory for a complete and independent GIS application. This Geditor implementation builds a graphical display interface into jRelix and utilizes the Aldat spatial analysis capabilities to build an independent GIS application. Geditor keeps the graphical display interface succinct and utilizes Aldat spatial analysis capability to the greatest extent. This demonstrates that the Geditor implementation is not only a feasible but also a simple approach in an integrated architecture using a relational database.

A high performance implementation is not the central issue in this work. We do not expect Geditor to be very fast. Performance improvement can be obtained by tuning the spatial analysis algorithms, choosing a different data model, and other spatial database techniques.

As a result of this Geditor implementation, we (1) built a graphical display interface to display the map based on the map data stored in jRelix format; (2) integrated a series of core GIS functionality into Geditor by utilizing Aldat spatial analysis capabilities; and (3) built the new syntax (**gedit**) into jRelix to allow jRelix programmer to call the Geditor. It is beyond the capability of this thesis to implement all possible GIS functionality into Geditor. However, our work is complete enough to perform interesting studies of implementing spatial applications using a relational database. Consequently, this thesis will provide a resource for a GIS implementation in a jRelix environment, and the fundamental framework for further studies on this issue.

The rest of this chapter is organized as follows. In Section 1.1, an overview of the relational database systems is presented. General concepts and topics related to the Geditor implementation are discussed in this section. Section 1.2 provides an overview of GIS capabilities. The GIS operations in this section are categorized as being either binary or unary according to the operands these operations have. Section 1.3 lists all the GIS functions Geditor includes. The thesis outline is given in Section 1.4.

1.1 Relational Database System

Relational database systems have developed rapidly since 1970s. Not only the strict and consistent mathematical model has been defined, but relational database systems also achieved great success in the commercial world. Nowadays, relational databases are still developing very fast to incorporate advanced properties and constructs to deal with modern data-intensive applications. In this section, we are going to review the basic concepts of relational database systems.

1.1.1 Relational Model

The relational model was first proposed by Dr. E.F. Codd in his famous paper " A Relational Model of Data for Large Shared Data Banks" [Cod70]. In his relational model, Codd uses a collection of tables that he terms relations, to model and store data about objects in the real world. Each relation resembles a table which consists of rows and columns. "tuples" is used to refer to rows and "attributes" is used to refer to the column header. The term "domain" refers to the set of legal values that an attribute can have, i.e. the data type of an attribute. Table 1.1 shows the data about different states in the U.S.A. represented in a relational model.

Mapfeature	Name	Pop	Temp
polygon1	Arizona	2350725	20
polygon2	California	29760021	20
polygon3	Nevada	1201833	-10
polygon4	Oregon	2842321	8

Table 1.1: Relational Model

As indicated in the paper, the relations in the relational model have the following characteristics:

- All tuples are distinct.
- The ordering of the tuples is immaterial.
- Each attribute is unique so that the order of columns is irrelevant.
- The domain of each attribute is of a simple type such as integer, float, etc. which cannot be further decomposed.

Operations on Relations

In the relational model, all data within a relational database are held in tables or relations. A system that supports the relational model should be able to perform well-defined operations on these relations to retrieve information. Relational algebra, which is also proposed by Codd, consists of a set of operations applied on relations. In the relational algebra, there is no operation performed on individual tuples. The relational operators take relations as operands and return a relation as a result which can be further manipulated.

The relational algebra operations are usually classified as unary or binary, according to the number of their operands. Unary operators take a single relation as operand and binary operators take two relations as operands. Both of them produce a single relation as their result.

- Unary operations
  - Projection: makes a copy of a relation with a specific subset of the attributes
  - Selection: selects tuples that satisfy a specific condition
- Binary operations
  - μ-join: join operators that generalize set-valued set operations
  - σ-join: join operators that generalize logic-valued set operations

Operations on Domains

The arithmetic and related processing of the values of attributes in individual tuples also becomes necessary. Merrett [Mer84] proposed the domain algebra which consists of a set of such operations. It allows the user to create new domains from existing ones. The generation of a new value from many values within a tuple or from values along an attribute also becomes possible. The domain algebra operations are defined as follows:

- horizontal operations: new value is generated from the values with a tuple.
  - Constant
  - Rename
  - Function
  - If-then-else
- vertical operations: new value is generated from values along an attribute.
  - Reduction
  - Equivalence Reduction
  - Functional Mapping
  - Partial Functional Mapping

Various combinations and permutations of the above operations of relational algebra and domain algebra are used in practice to retrieve information from a collection of relations in a relational database. Some of these, but notably not domain algebra, have been implemented on commercial DBMS in the form of SQL (Structured Query Language) and other specialized devices.

1.1.2 DataBase Programming Languages

The relational model has attracted much attention both in the academic world and in industry. It has proven itself exceptionally useful for many business applications. However, the commercial implementations of the relational model are lacking in expressive power and in the ability to handle complex data. Many applications are arising in science and engineering for which these implementations are inadequate tools. The relational model itself, however, is not limited to these implementaions. This has led to continued research in the field of database programming languages (DBPL).

The applications that drive the efforts in the research of database programming languages have the following properties:[Hul89]

- involve large amounts of complex, shared, concurrently accessed, persistent data
- reliability requirements
- involve distribution of data storage and processing over networks
- design orientation
- complex behavior involving inference or rule-based computation
- sophisticated graphical interfaces

Computer automated design (CAD), VLSI chips design and Geographic Information Systems are examples of such applications.

DBMS are capable of dealing with large amounts of persistent data, that is, the data stored in the secondary storage. It allows concurrent access to the data even if it is distributed among several sites. Programming languages provide well-proven and powerful techniques for creating, organizing and manipulating data that is in memory. Therefore, database programming languages seek to integrate the technologies and paradigms of programming languages and database management in order to solve the problem of developing the above data-intensive applications.

An early approach to creating a database programming language has been to embed a database query language into an existing programming language. For example, the INGRES relational database system [Sto76] embedded its query language QUEL into the C programming lanugage to produce the EQUEL language. QUEL variables and statements are inserted into a C program in lines that begin with '##'. A major disadvantage of this approach is that it requires the programmer to be fluent with both the host language and the query language. It also yields an awkward programming environment by fitting the bulk types of the query language, such as the relation, together with the typing system of the host language. This inspired the search for more integrated solutions to database programming languages.

Another approach to creating a DBPL is to add database features to an existing programming language. For example, Pascal/R [Sch77] combines the relational data model with the Pascal programming language [Jen85]. The type **record** which represents a tuple of a relation is added to the language. The constructor **relation of**, **database**, new iteration construct **for each**, and a set of operators that permit the traversal of a relation —**low, next, high and eor** are also added to the language. This extension to Pascal allows the manipulation of relations together with the mechanism to support persistence and efficiency.

An important step was the demonstration of the possibility to design a programming language with uniform persistence. A problem with conventional programming languages is that their constructs (e.g. arrays and records) do not correspond to those for persistent storage (e.g. the abstraction of a file or of a relation). The programmer must map the data from the forms used in primary memory to those used on the persistent storage. The typing mechanism provided by the programming language is usually lost across this mapping. In a Persistent Programming Language, such as PS-algol [Mor88] [Atk83] [Atk84], the mapping becomes unnecessary because data of any type may persist. The value persists whenever it has a persistent label or it is a part of some structure with a persistent label. This saves the programmer's effort of data mapping and also keeps the data type completeness which is essentially that all data types have equal rights.

At the same time, the research efforts in database programming languages are devoted to the incorporation of the facilities of a DBMS with the object-oriented computing paradigm. For example, ObjectStore [Lam91] adds persistence to the C++ programming language which makes the accessing of persistent data seamless to the programmer. A number of bulk types such as ordered lists, sets and bags are also added to the language to manage large amounts of data. Queries are contained between delimiters '[:' and ':]'. Other features such as transactions, locks and logging for recovery are also supported in ObjectStore.

Efforts have also been put into the investigation of the connection between logic and databases.

The Knowledge-Base Management Systems (KBMS) combine the traditional feature of a DBMS with the logic programming paradigm. Attempts have been made to combine database features with Prolog [Boc86] [Cha86] [Ioa94]. Datalog [Ull85] [Mor86] [Mor87] [Cer89] is one of them. It is based on Prolog and developed for use with relational databases. The predicate can be stored in a relation of the same name. Every tuple of this relation represents a fact. Some extensions of Datalog also support bulk type constructors to deal with large amount of data. An object-oriented extension to Datalog also includes methods, classes, instantiation, overloading and late binding [Abi93].

1.1.3 Active Databases

Traditional database systems are passive because commands are executed by the database when requested by the user or application programs. The system cannot respond to happenings of interest without the user intervention. An Active Database System is a conventional passive database system extended with the capability of reactive behavior. The desired behavior is expressed in rules that are defined and stored in the database [Vla98].

For example, an inventory control system needs to monitor the inventory database, so that when the quantity in stock of some item falls below a threshold, a re-ordering activity will be initiated. In an active database system, a corresponding rule can be defined and the active database system is responsible for detecting the quantity in stock. When the quantity in stock falls below a threshold, the request of order will be triggered by the active database without the user's intervention.

In active database systems, the event-condition-action (ECA) model is widely used. According to McCarthy [McC89], an event-condition-action model consists of three components.

- Event: "An event is the occurrence of pre-defined state which triggers the rule and causes the system to evaluate the condition".
- Condition: "Conditions are typically predicates or queries against the database system".
- Action: "An action is a sequence of operations which are executed when the condition of the triggering event is satisfied".

1.1.4 jRelix

jRelix (the **j**ava implementation of a **Rel**ational database programming language in Un**ix**) was developed at the Aldat lab of the School of Computer Science at McGill University. jRelix contains a database management system (DBMS) which is responsible for organizing and storing data, and a programming language Aldat — Algebraic Data Language, based on relational algebra and domain algebra [Mer77] [Hao98] [Yua98]. jRelix incorporates complex constructs such as computations (procedures) and some object-oriented paradigms, such as instantiation [Bak98]. The event handler, which is the characteristic of an active database system, is also implemented in jRelix [Sun00]. Therefore, jRelix is a full-featured modern relational database system, which is an ideal candidate for the implementation of current data-intensive applications, such as GIS.

In [Mar98], Martinez proved the spatial capability of jRelix by implementing in Aldat an essentially complete set of the spatial operations of Arc/Info and MapInfo. The Aldat codes of these operations such as polygon overlay, buffering, and spatial queries are presented in that thesis. Since jRelix has no graphical editor, Martinez exported the result of the spatial operations to Arc/Info to display the map. For a complete GIS application, this map display was obviously not satisfactory. It was very inconvenient for the user to view the map, let alone more advanced functions related with the display of the map, such as editing the color of map features and thematic mapping. This led to the motivation of building a GIS editor which provides a graphical interface that allows the user to display and edit the map. Common GIS spatial operations should also be incorporated in this editor to allow the user to call them directly from this graphical interface. This is the objective of this thesis. In the next section, an overview of GIS capabilities will be presented in order to investigate what functions the GIS editor should incorporate.

1.2 Overview of GIS capabilities

1.2.1 Introduction to GIS

Geographic Information Systems are designed to handle information relating to spatial locations [Sta90]. It is a system of hardware, software, data, people, organizations, etc. for collecting, storing, analyzing and disseminating all types of geographically referenced information [Due89]. The most common understanding of a GIS emphasizes it as a tool for storing and retrieving, transforming and displaying spatial data [Bur86]. Five essential elements must be contained in a GIS: data acquisition, preprocessing, data storage and retrieval, manipulation and analysis, and data reporting [Peu90] [Cla99]. These five elements actually capture the flow of work in a GIS system. In each stage of the continuous process, a GIS provides powerful tools for the user to complete the work. Refer to section 1.3 to see what elements Geditor currently contains.

GIS developed rapidly since 1980 and achieved great popularity in the commercial worlds in the past ten years. This is because a GIS is not only a tool for displaying and making maps, but most importantly, it is a tool for the analysis of spatial data and the creation of more interesting and real studies resulting from the combination of data from different sources. A GIS not only combines spatial and non-spatial data into one system, it combines a collection of thematic layers (coverages) that can be linked together. Each of the layers can be manipulated separately and various operations are allowed on the combination of multiple layers. Figure 1.1 shows a typical example of thematic layers stored in a GIS.

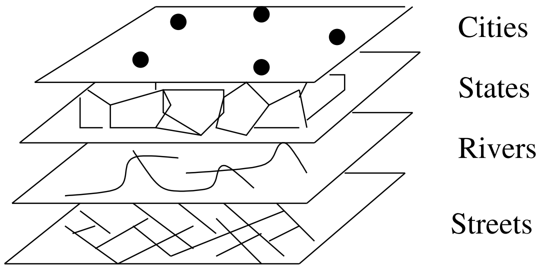


Figure 1.1: Thematic Layers

Based on the various source of data the GIS captures and stores, it provides powerful functionality for the user to process the data. An overview of GIS capabilities will be presented next.

1.2.2 GIS capabilities

GIS provides powerful tools for processing both spatial and non-spatial data. From a mathematical point of view, these tools are operators applied on spatial or non-spatial operands. Therefore, we are going to categorize the various functions of GIS packages into operators in the following discussions. According to the number of operands each function acts on, these functions are divided into *unary* and *binary* operators. For each operator, the number and types of operand(s) are discussed. The functions of each operation are also described. Moreover, after the data (including both spatial and non-spatial) have been imported and stored into the system, all the functions GIS provides are related with two parts: data manipulation and display. Since we are going to use jRelix to manage the data (both spatial and non-spatial), the data manipulation will be performed in Aldat language of jRelix. In the following discussions, we also indicate whether the operation is a pure data manipulation, therefore an Aldat problem (Aldat), or a pure display problem (Display), or both (Aldat & Display). The purpose of this indication is to identify those functions that are related with graphical display. Notice that a function is identified as a display-related problem only if a graphical interface is needed during the process of data manipulation of this function. After the data manipulation, all the results need to be displayed. This can be achieved by a common map-display module, which our Geditor will definitely include.

There is a set of operations based only on descriptive data, such as displaying the descriptive data table, browsing, editing, selecting, and joining descriptive data tables. Since this set of operations is not related to spatial data and are common operations in a conventional relational database management system, we omit the discussion of them in the following discussions.

A. Binary operators

Most operations in GIS are binary operators. For example, the polygon overlay is a typical binary operator applied on two layers with polygon topology. Other operations such as spatial query, buffering, dissolve, and thematic mapping are also binary operators. In this subsection, we discuss 1. two operations of spatial retrieval: query window generation and spatial query, 2. measurement of points, 3. buffer generation, 4. polygon overlay, 5. dissolve, and 6. thematic mapping.

In the following discussions, first, the function of the operation is explained, then an operator table is presented as the summary of this function. The first column of the table indicates whether this is a display or Aldat related function. The second and third columns describe the two operands of this operation. The fourth column presents the description of the operations on the two operands.

1. **Spatial Retrieval — Query window generation and spatial query**

   GIS packages allow the user to spatially extract both spatial and non-spatial information. This group of operations includes identifying, labeling, query window generation, and spatial query. Among these operations, query window generation and spatial query are binary operations.

(a) **Query window generation**

This function involves the ability to generate points, irregular shaped polygons, squares, circles, etc. for interactively overlaying with map features contained in certain coverage. The map features that coincide in space with these generated query windows are retrieved (or selected) according to a certain spatial relationship. Three techniques are usually used in this set of operations [Peu90]:

- Adjacency

  The map features that are adjacent to a user-generated point are selected.

- Select by polygon

  The map features that are entirely within or partially within the user-generated polygons are selected.

- Select by polygon overlay

  Only the portions of the map features which fall within the boundaries of the query window polygon are selected. In this case, all lines as well as parts of polygons which fall outside of the query window are snipped off using polygon overlay techniques (polygon overlay will be discussed in the following section).

Figure 1.2 shows examples of the above three query window generation techniques.

In Figure 1.2, the map features that are selected by the user are displayed in solid lines or shaded areas. Those that are not selected are displayed in dashed lines. User generated points are represented as dash-dotted mouse arrow. User generated polygons are shown in dash-dotted lines.

For example, in Figure 1.2 (1), the point, line, and polygon that are adjacent to the user-generated mouse arrow, are selected, that is, shown in solid lines. In Figure 1.2 (2), the point, line and polygon that are entirely within or partially within the user-generated polygons (rectangles) are selected. In Figure 1.2 (3), the point and line segment that are entirely within a user-generated polygons (rectangles) are shown in solid lines (se-

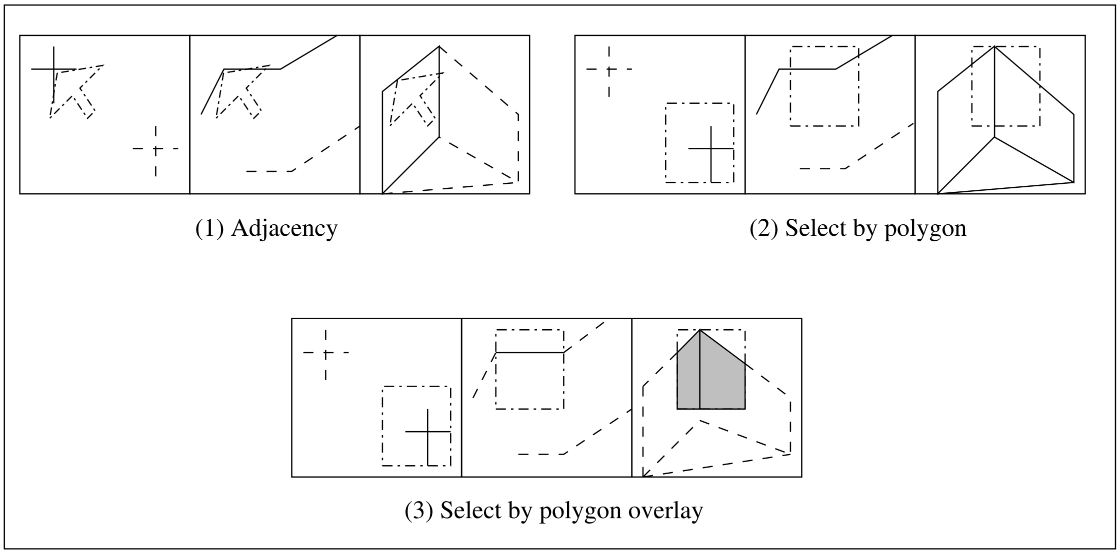


Figure 1.2: Query Window Generation

lected), and the parts of polygons that fall within the user-generated query window are shown in shaded area (selected).

(b) **Spatial query**

This operation locates map features in relation to a given existing map feature [Hut97] [Whi99] [Zei97]. For example, "show the lakes that are within Quebec province" is a query which locates all the polygons that are "entirely within" a particular polygon. See Figure 1.3. Notice that the two map features can be from two different layers.

(c) **Operator table**

From the operator point of view, query window generation and spatial query are the same. Both of them retrieve map features by investigating the spatial relationship between two graphical objects. The only difference is that the query window generation uses user-generated graphical objects to locate existing map features by studying their spatial relationship, while the spatial query uses an existing map feature to perform such an operation. Table 1.2 summarizes the spatial retrieval operations as binary operators.

The above table includes all the operations in spatial retrieval. Moreover, NOT is allowed to be added in front of each operator. For example, "not strictly contained" is

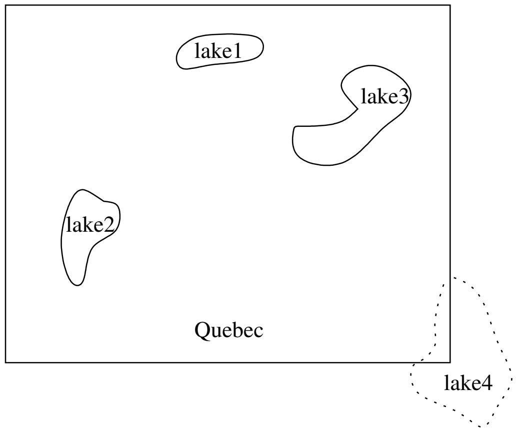


Figure 1.3: Spatial Query

	Map features	**Window Object (existing or user-generated)**	**How to combine**
Display & Aldat	Point Line Polygon	Point Line Polygon	Select the map feature if it has the following relation with the window object: **Strictly contained** **Adjacent** **Intersection** **Overlap** **Disjoint**
			Select the part of the map feature using polygon overlay

Table 1.2: Operators of Spatial Retrieval

also an operator which locates the three cases as Figure 1.4 shows.

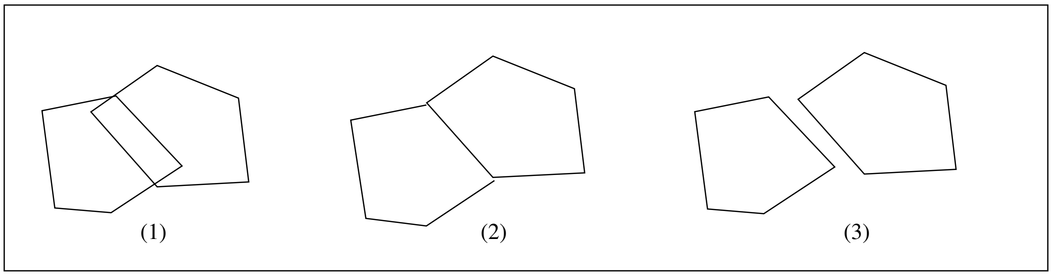


Figure 1.4: Three Cases of “not strictly continued”

To achieve this function, an interface is needed to capture the user input of the windowing objects by either selecting an existing map feature or drawing graphical objects on screen. Aldat is capable of computing and locating the map features that are in the particular relationship to the windowing objects. [Mar98] provides Aldat codes for this set of operations.

2. **Measurement of Points**

In GISs, *points, lines and polygons (or areas)* are the three basic graphical objects. The most common types of measurement tasks involve the measurement of the three basic objects. Among them, the measurement of points is a binary operator. The measurement of lines and polygons are unary operators.

Objects with zero dimension are represented as points. Cities in a province, hospitals and schools in a city are usually represented as points. There is no formal calculation of the size of points. The measurement related with points is the total number of points falling in a polygon or in a buffering area [Peu90]. See Figure 1.5.

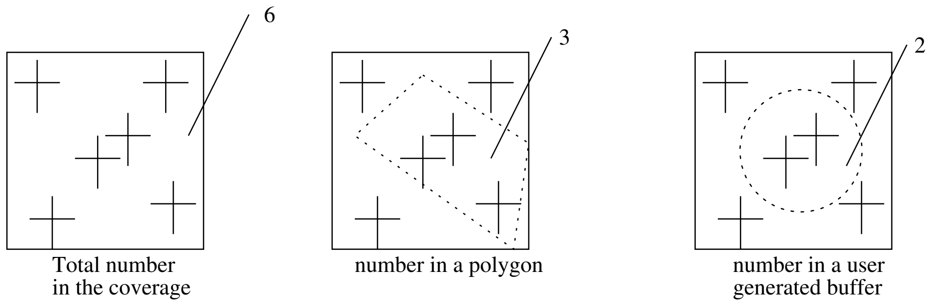


Figure 1.5: Point Counting

The counting of points can be represented as the binary operator as Table 1.3 shows.

An interface is needed to capture the user input of the window polygons. Aldat is capable of locating the points entirely within the window polygon and counting the number of such points. The operation "entirely within" is one of the spatial operators discussed in the last section. The counting of the points is a conventional database operation.

3. **Buffer generation**

	Map features	Window polygon (existing or user generated)	How to combine
Display & Aldat	Points	Polygon (the whole coverage, user generated buffer, etc.)	Strictly contained

Table 1.3: Point Counting Operator

Buffering is a commonly used operation in the GIS world. It creates new polygons from points, lines, and polygon features within a specific distance. Figure 1.6 shows the summary of different types of buffers [Peu90].

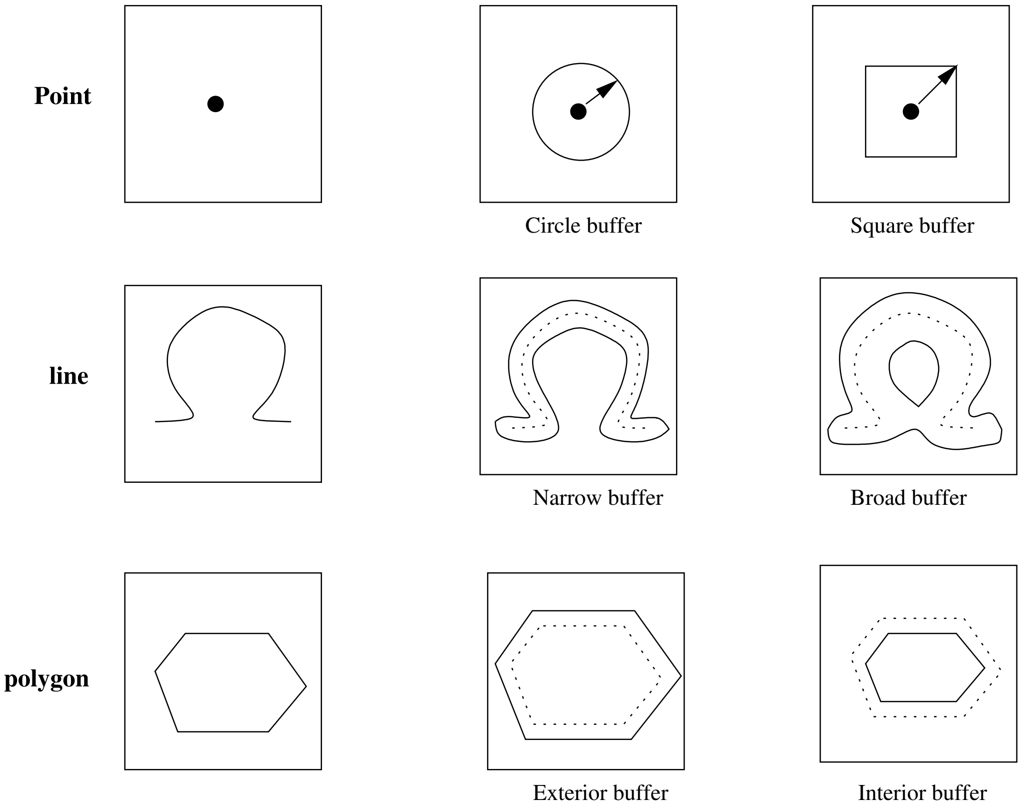


Figure 1.6: Buffering

In Table 1.4, all the buffering operations are summarized as different operators. Except for the routine display of the results, this operation is a pure data manipulation. Aldat is responsible

	Map features	Distance Value	Operations
Aldat	Points	dist	**Square_buff**(p1,dist) **Circle_buff**(p1,dist)
	Lines	dist	**Narrow_buff**(line1,dist) **Broad_buff**(line2,dist)
	Polygons	dist	**Interior_buff**(poly1,dist) **Exterior_buff**(poly1,dist)

Table 1.4: Buffering Operators

for generating all the graphic data of all sorts of buffers of different map features. [Mar98] also provides such Aldat codes.

4. **Polygon Overlay**

Overlay operations play an important role in GIS applications. This is because most applications of geographic information must integrate information from different resources. In the output coverage, all the attribute values from different parents of the map feature (polygon in this case) can be accessed. The most commonly used polygon overlay operations in commercial products include *Clip, Split, Erase, Update, Identity,* etc.

Actually, polygon overlay involves two phases: *geometric phase* and *non-spatial phase* [Chr97]. In the *geometric phase*, two layers are combined to produce a composite geometric representation where each area has a key linking to the attribute tables of the two source layers. For example, in Figure 1.7, suppose polygon 1 is from coverage A, and polygon 2 is from coverage B. Through geometric phase, a composite geometric representation is generated as Figure 1.7(b) shows. For each area (1,2, or 3), the link to the attribute table is indicated. Then to produce the output coverage, either select polygon in Figure 1.7(b) by indicating the source of layer, or select polygons with the attribute values these areas link to. For example, the user can select areas that are from both layer A and B as Figure 1.7(c) does. The user can also select areas that are banana field and rainforest at the same time. Since the data manipulation in this phase only involves non-spatial (descriptive) data, it is called *non-spatial phase*.

Based on the above understanding, the polygon overlay operations can be summarized as indicated in Table 1.5.

	Input coverage	**Overlay coverage**	**How to combine**
Aldat	Polygon	Polygon	Compose new polygon coverage from two different layers using geometric intersection processing techniques
Aldat	descriptive data table	descriptive data table	Select polygons in the composite coverage by indicating the source layer using AND and OR Boolean operators.
			Select polygons in the composite layer using the descriptive data table each polygon links to.

Table 1.5: Operators of Polygon Overlay

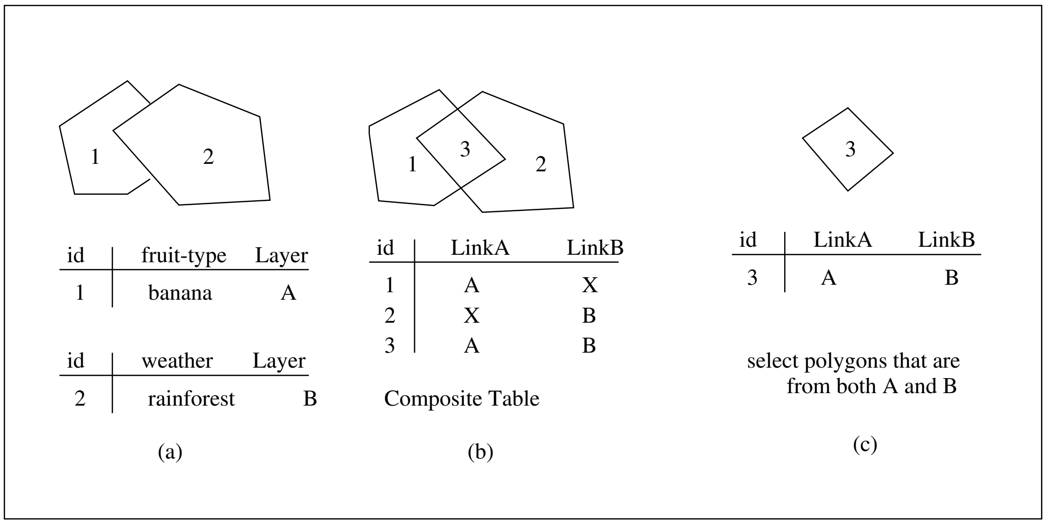
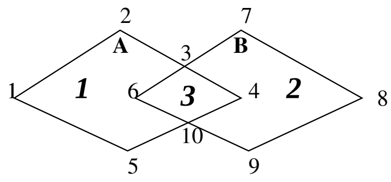


Figure 1.7: Two Phases of Polygon Overlay

All polygon overlay operations can be achieved using the above binary operators. For example, as listed in Figure 1.9, the seven overlay operations that are supported in Arc/Info [Zei97] can be implemented by first generating the composite coverage and then selecting the areas by indicating the source of layers the areas are from.

In the example, two polygons are from coverage A and B respectively. In the geometric phase, new vertices 3 and 10 are generated and the composite table is also produced as Figure 1.8 shows. 1,2, and 3 areas are generated after this phase.

In the non-spatial phase, different selection conditions produce the result of different polygon



	fromA?	fromB?
1	A	
2		B
3	A	B

Figure 1.8: Geometric Phase

overlay operations of Arc/Info. See Figure 1.9. In this figure, there are four columns. The first column lists the name of each operation. The second column presents the visualization of the corresponding operation. The operation results are presented in the shaded area. The third column lists the selection condition to produce the corresponding result. The fourth column describes the operation results using area numbers(1, 2, and 3).

Obviously, Aldat is capable of the operations in the non-spatial phase because only conventional database operations are involved in this phase. Aldat is also capable of data processing in the geometric phase. Related Aldat codes are already available in [Mar98].

5. **Dissolve**

   This operation is an inverse operation of polygon overlay. Instead of splitting polygons to generate new areas by overlaying two map layers, it merges adjacent polygons based on the similar attributes the polygons have [Hut97]. For example, the merging of Federal Republic of Germany and Democratic Republic of Germany can be achieved by merging all the adjacent polygons that have the same attribute value of political_name. After the merge of the two countries, the polygons 1, 2 and 3 have the same political_name. Consequently, the three adjacent polygons are merged to produce a single polygon as Figure 1.10 shows.

   This operation involves the modification of the geometric data based on identical attribute values. Only those adjacent polygon data needs to be updated. Table 1.6 represents the dissolve operation as a binary operator. Aldat is capable of performing this operation. [Mar98] provides the related codes.

6. **Thematic mapping**

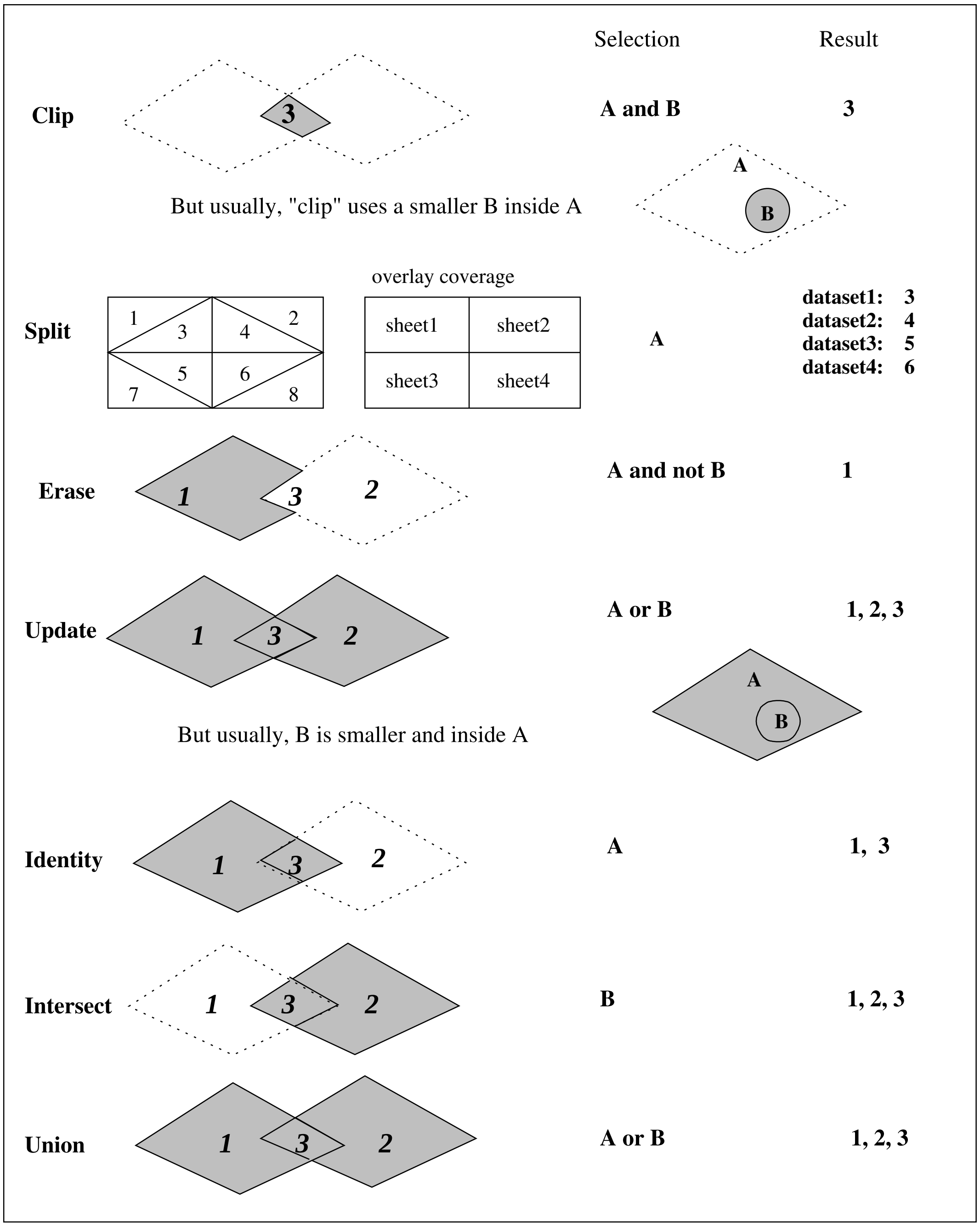


Figure 1.9: Polygon Overlay Operations in Arc/Info (Non-spatial phase)

One of the GIS's capabilities is to generate different maps based on the same map data. Thematic mapping helps the user to achieve this goal. *Individual Value map, Range Map, Dot*

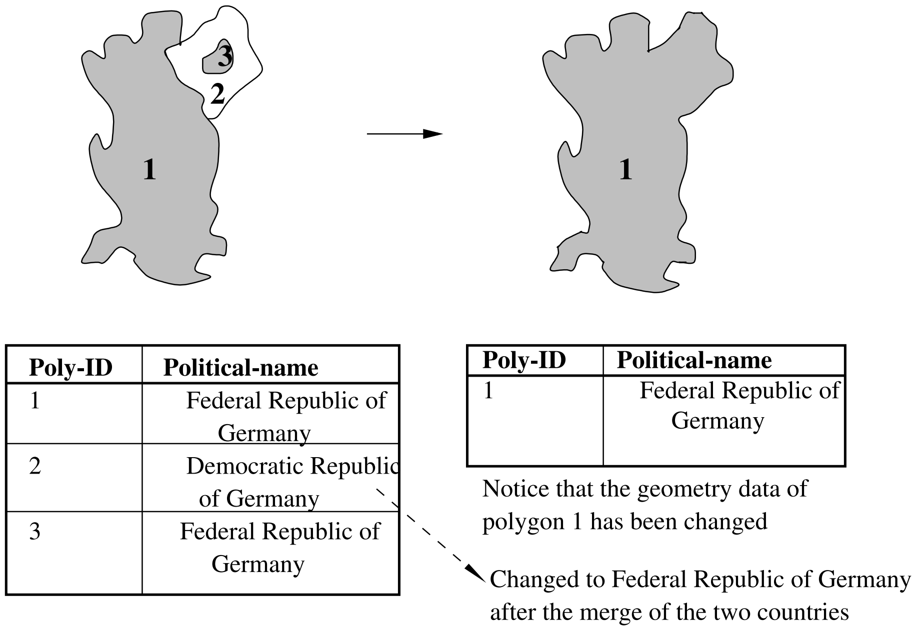


Figure 1.10: Dissolve

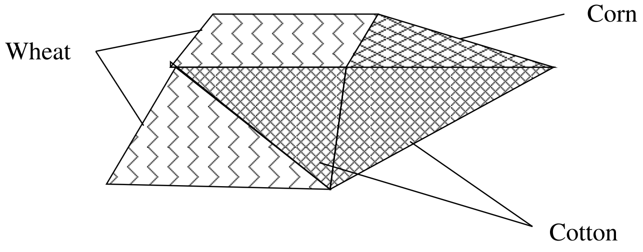
	Map features	Attribute	Operations
Aldat	Polygons	Attribute name	Put all the polygons with the same attribute value into one group and update their coordinates to dissolve the shared boundaries

Table 1.6: Dissolve Operator

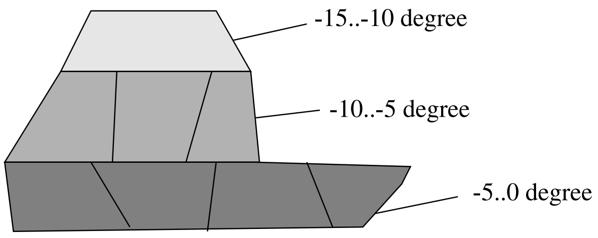
density, Graduate Symbol, and chart symbols are commonly used thematic maps in commercial products [Whi99] [Zei97] [Hut97].

Individual value maps classify map objects by different colors or fill patterns. The map objects that have the attribute with a certain value are filled in the same color or pattern. For example, in the following figure, the polygons filled in the same pattern show the same type of field.

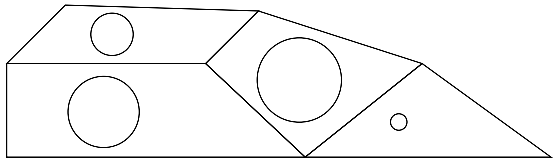
Range maps allow the user to group map objects according to the range of values the map objects linked to. The map objects that have the attribute in a certain range are filled in the



same color or pattern. For example, in the following figure, the polygon filled in the same color shows a certain range of temperature, say -10 to -5 Celsius degrees.



Graduated symbol maps display symbols in graduated size for polygons or points according to the value of an attribute. For example, in the following figure, the larger the circle, the greater the population in the polygon.



Dot Density thematic maps display randomly dispersed dots within a region. The dots depict the amount of the selected data each region contains by the number of dots placed within the region. For example, in the following figure, 1 dot represents 100 people.

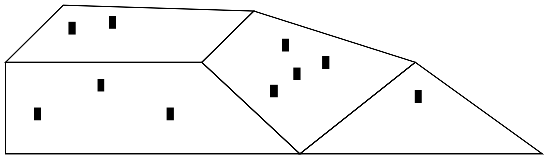
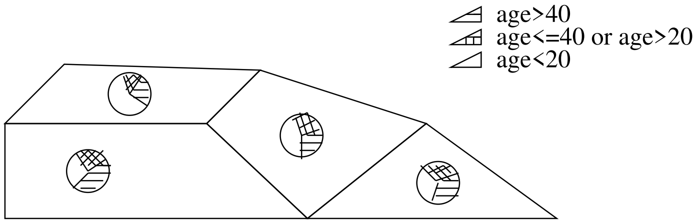


Chart symbol maps display pie charts or bar charts within a polygon. The chart in each polygon shows the distribution of an attribute value. For example, in the following figure, the pie chart in each polygon shows the age distribution of the population in the area.



The operations of thematic mapping can be summarized as binary operations as Table 1.7 shows.

	Map coverage	**Attribute**	**Operations**
Aldat & display	Map with any type of map features (usually polygons)	Attribute name	**update** command in Aldat
			Commands in Aldat to generate tuples in relations

Table 1.7: Operators of Thematic Mapping

An interface is needed to allow the user to specify the attribute name that the thematic mapping is based on. The other part of this set of operations is related to the manipulation of the map data. Individual value and range maps change the color of the map features based on the value of an attribute. This can be achieved by the **update** command in Aldat. Graduate symbol maps generate new graphical objects, such as circles and squares, according to a certain attribute linked to existing map features. This can be done by commands in Aldat to create graphic data (tuples) in map relations. Dot density thematic maps generate points based on an attribute and chart symbol maps generate pie or bar charts based on an attribute value. These operations can also be obtained using commands in Aldat to create graphic data (tuples) in map relations.

7. **Drawing Charts**

When two attribute names are specified, a graph of their relation to each other can be presented as *point charts, bar charts, pie charts, line charts*, and *shaded charts*.

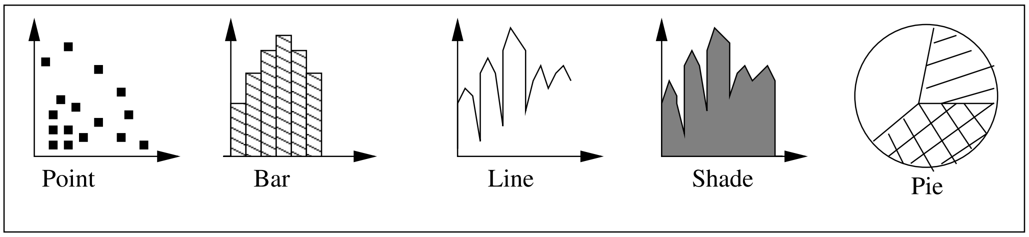


Figure 1.11: Different Charts

Although this set of functions is not found only in GIS packages, they are widely used in them. The value of the two attributes needs to be retrieved and the display interface needs to draw the charts according to the values. This can be summarized as the following binary operator shown in Table 1.8.

	Attribute	Attribute	Operations
Aldat& Display	Attribute name	Attribute name	Retrieve the attribute values according to the given attribute names and display different type of charts: Bar, Point, Pie, Shade, Line, etc.

Table 1.8: Charts Operators

For example, if attribute names *year* and *population* are given, a bar chart can be generated showing the population of each year as Figure 1.12 shows.

Aldat needs to retrieve the value of the two attributes. This is a conventional database operation. Then a display interface needs to draw the chart according to the values.

B. Unary operators

Other operations in GISs are unary operators. This includes some operations in spatial retrieval, measurement, map sheet manipulation, map generalization and some map display functions, such

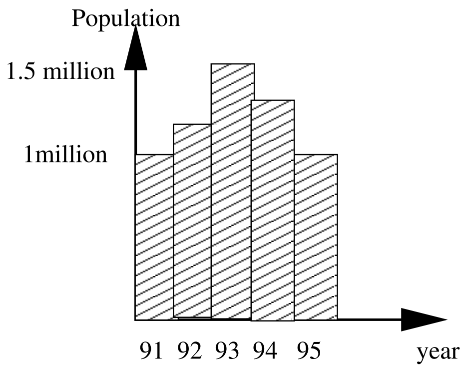


Figure 1.12: Bar Chart

as legend display, print layout, and report generating. In this subsection, we discuss 1. two other operations of spatial retrieval: identifying and labeling, 2. the measurement of lines and polygons, 3. map sheet manipulation, 4. spatial data editing, and 5. some display functions.

In the following discussions, first, the function of the operation is explained, then an operator table is presented as the summary of this function. The first column of the table indicates whether this is a display or Aldat related function. The second column describes the two operands of this operation. The third column presents a description of the operations on the two operands.

1. **Spatial Retrieval — Identifying and labeling**

   (a) **Identifying**

   Most GIS packages allow the user to specify the position of a map object (usually by placing the mouse close to the map feature and clicking the mouse) and then display all the linked attributes of this map feature. This is called *identifying* [Hut97]. This operation can be represented as the following unary operator:

	Position of the map feature	**Operations**
Display & Aldat	User input by clicking mouse close to the map feature	Obtain the X,Y coordinate from the user input and then retrieve all the attributes related with this map feature

Table 1.9: Identifying Operator

An interface is needed to obtain the position of the map feature and then translate it into the X and Y coordinates. Aldat operations can complete the attribute value retrieval. Finally, a display interface needs to display the result of all the attribute values in a proper place on the screen.

(b) **Labeling**

Labeling is another common operation in GIS packages [Zei97] [Whi99] [Hut97]. It is also called *automatic labeling*. By specifying an attribute name, the value of this attribute will be displayed as labels close to corresponding map features, for example, inside the polygon if the map feature is a polygon. This operation can be represented as the unary operator indicated in Table 1.10.

	Attribute	**How to display**
Display & Aldat	Attribute name	Retrieve the value of the attribute name as the label value and display properly on the map.

Table 1.10: Labeling Operator

Aldat commands are needed to retrieve the attribute values as labels and the display function is needed to calculate the proper position to show the label.

2. **Measurement - lines and polygons**

Besides points, measurement operations involve the lines and polygons [Peu90].

(a) **Lines**

Lines have one single dimension of length. Rivers, roads, rails are usually represented as lines. The calculation of length includes the length of the whole line and the length of a single edge.

Calculation of line length can be represented as the unary operators shown in Table 1.11.

	Map features	**Computation**
Display & Aldat	line or edge (existing map feature or drawn by user)	**Line_len**(lineID) **Edge_len**(edgeID)

Table 1.11: Line Measurement Operators

Interface is needed to allow the user to specify the line or edge and display the result of the measurement. Aldat operations can compute the length of a line or an edge.

(b) **Polygons**

Polygons have two dimensions, length and width. Provinces, lakes and parks are usually represented as polygons. The two basic measurement types of a polygon are the perimeter and the area of the polygon.

The calculation of area and perimeter can be represented as unary operators as Table 1.12 shows.

	Map features	**Computation**
Display & Aldat	Polygons (existing or user generated)	**Perimeter**(polyID)
Display & Aldat	Polygons (existing or user generated)	**Area**(polyID)

Table 1.12: Polygon Measurement Operators

An interface is needed to allow the user to specify the polygon and display the result of measurement. Aldat is capable of the computation of the perimeter and area of the polygon.

3. **Map Sheet Manipulation**

A series of techniques manipulate the x,y coordinates for a given map. This includes projection change, coordinate translation, scale change and rotations [Peu90]. These can be summarized as Table 1.13 shows.

	Value	**Comutations**
Aldat	Scale value	**Scale change:** X,Y multiply a coefficient
	Projection name	**Projection change:** change X, Y according to a projection formula
	Constant value	**Coordinate translation:** X, Y plus or minus a constant
	Rotation angle	**Rotation:** change X, Y according to a formula

Table 1.13: Operators of Map Sheet Manipulation

The **update** or assignment operator in Aldat can be used to update X, Y coordinates in the above series of operations. Therefore, this set of operations is a pure Aldat computation.

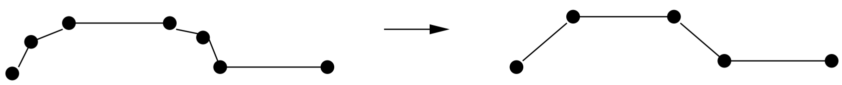
4. **Spatial data editing**

After digitizing, a lot of editing operations need to be performed on the map features to correct the errors in the data capture stage. Map generalization, rubber sheeting, and snapping are common operations of spatial data editing [Hey98] [DeM97].

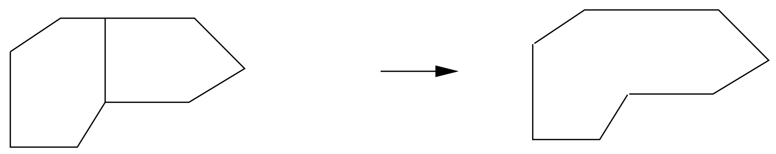
(a) **Map generalization**

Map generalization tools are frequently used when map scales are changed. This series of operations is also used to edit the map features that are digitized during the data capture stage.

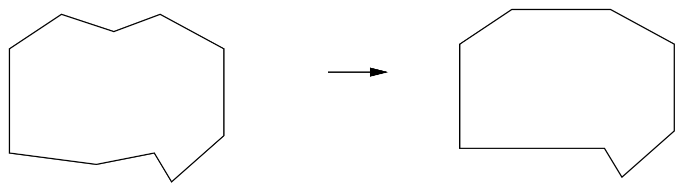
- *Line coordinate thinning* — This is a technique for reducing the number of coordinates defining a given line.



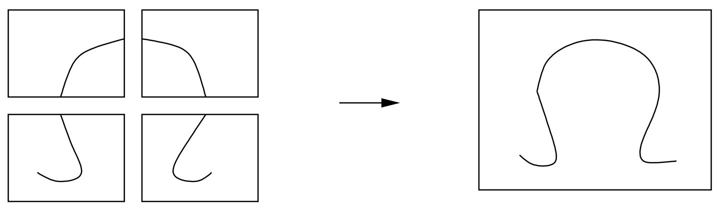
- *Dropline* — This is a technique to drop the line which is shared by two polygons. The remaining line segments of the two polygons make up a new polygon.



- *Polygon thinning* — This is a similar exercise to line thinning.



- *Edge matching* — Edge matching consists of a series of procedures for bringing together a large number of map sheets and composing them into one continuous map. Problems which must be resolved are: joining lines and polygons from adjacent maps, matching of the boundaries between the maps and dropping the lines which separate polygons having the same characteristics.

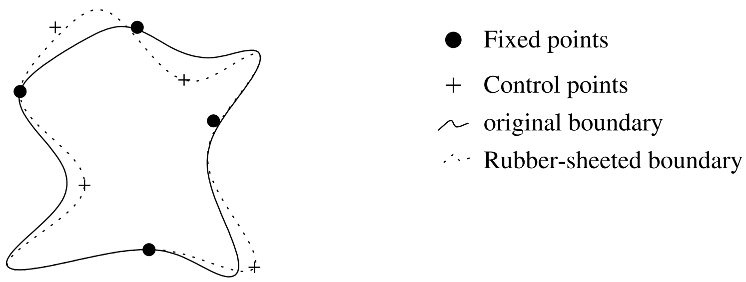


(b) **Rubber sheeting**

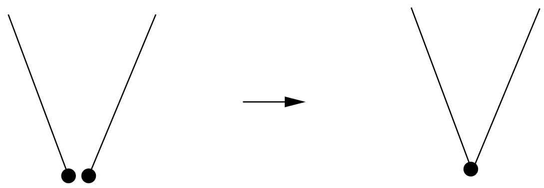
Rubber sheeting involves stretching the map in various directions as if it were drawn on a rubber sheet. Fixed points and control points are believed to be correct.

In operation, fixed points are kept still while others are stretched to fit the control points.

(c) **Node snapping**



In node snapping, points that are close to each other that should indeed be the same point are merged to generate an identical point in the graphic database.



All the above operations need an interface for the user to specify the map feature to be edited and to edit the map feature interactively. After that, the result of editing needs to be saved into the graphic database using the **update** command in Aldat.

Therefore, the above operations can be summarized as a unary operator as Table 1.14 shows.

	Map features	**Operations**
Display & Aldat	Lines Polygons	Using display interface to capture the user editing of the map feature and Aldat **update** command to update the graphic database

Table 1.14: Operators of Spatial Editing

5. **Some display functions**

Besides thematic mapping, display issues involve the legend display, printing layout and exporting graphic maps to different graphic file formats.

These display functions can be represented as unary operators as Table 1.15 shows.

		Operations
Display	Attribute name	Display the legends and labels based on the attribute value
Display	Graphic objects such as the title, north arrow, scale bar and legend	Move and edit the title, north arrow, scale bar and legend
Data Conversion	Map coverage	Convert data of maps to graphic files

Table 1.15: Operators of Some Display Functions

1.3 Geditor functions

From the above discussions about the relational database concepts and the overview of GIS capabilities, we can conclude that all the GIS functions can be implemented using Aldat capabilities and a display interface. In the above discussions, some more sophisticated functions such as spatial interpolation in terrain analysis, network analysis and image processing are omitted. In commercial products, they are often incorporated in extended packages such as ARC Network (for network analysis), ARC TIN (for terrain analysis), and MGE Grid Analyst (for image and raster analysis) [Kor97]. However, we believe that from the observation of the whole family of the GIS core functions, all the GIS functions including those extended ones can be implemented using Aldat capabilities and a graphical display interface.

It is beyond the scope of this thesis to incorporate all the possible GIS functions into our GIS editor. Our purpose is to implement the typical common functions of GIS packages, especially those display-related functions. More importantly, we are going to provide a flexible and dynamic implementation to allow the further extension of this editor. The event handler, which is the characteristic of active databases, is adopted in this Geditor. Proper events are generated when the user requests for an operation and the corresponding event handlers are invoked. Customized event handlers written by Aldat programmers become possible, which makes the Geditor flexible and dynamic.

The following functions are incorporated in Geditor:

1. Multi-layering

- Add Layer — allows the user to add layers to the current desktop.
- Layer Control — allows the user to control the states of the layers.

2. Changing the view of the map

   - Editing uniform layer — allows the user to change the color and symbol of a uniform layer.
   - Thematic mapping — allows the user to create thematic maps.
   - Legend Editor — allows the user to edit the legends.

3. Queries

   - Identifying — displays all the attribute values of a particular map feature selected by the user.
   - Expression Builder — allows the user to query the spatial database by creating expressions.
   - Spatial Query — provides sorts of spatial operators to retrieve map features according to proper spatial relationships.

The following functions are left out in Geditor either because they are not display-related or because they are often left out in GIS core functions.

- Buffer Generation

  This function is left out because it is not a display-related problem. Only the result needs to be displayed and the other part of the function can be achieved by Aldat capabilities.

- Polygon Overlay

  This is also an Aldat only operation.

- Dissolve

  Aldat is capable of completing this function.

- Measurement (of points, lines and polygons)

  This function is a both Aldat and Display related operation. We omit it because it is not as close to the core of GIS functions as those we include in Geditor.

- Spatial data editing

  This set of functions are used during the data capture stage which is not as close to the core of GIS functions as those we included in Geditor.

- Map Sheet Manipulation

  This set of functions can be achieved by Aldat capabilities only.

  Because of the above reasons, according to the work flow mentioned in Section 1.2.1, Geditor incorporates the functions of data storage and retrieval, manipulation and analysis, and data reporting, but leaves out the functions of data acquisition and preprocessing elements.

1.4 Thesis Outline

This chapter has discussed the fundamentals for the implementation of the Geditor. Now, we are ready to discuss the implementation in detail.

Chapter 2 provides an overview of jRelix, containing the basic elements to understand the subsequent discussions about the Geditor implementation. It covers all commands and statements used in the implementation.

Chapter 3 presents the User's Manual of Geditor. The usage of all the functions are explained in detail in the order that they appear on the menu bar of Geditor window from the left to the right. Examples are provided with graphics and relations data to illustrate the operation usage.

Chapter 4 describes the implementation of Geditor. The system architecture is presented and the algorithms of the main classes are explained in detail. The sequence of the algorithm description is kept the same as that of the description of the corresponding functions in the User's Manual.

Chapter 5 presents the conclusions of this work and suggestions for related future work.

Chapter 2

jRelix Overview

jRelix, a relational database programming language, was developed at the Aldat lab of the School of Computer Science at McGill University. It contains a database management system (DBMS) which is responsible for organizing and storing data, and a programming language Aldat — Algebraic Data Language, based on relational algebra and domain algebra [Mer77]. This chapter presents a tutorial on jRelix so that the user will understand the rest of the thesis. This tutorial focuses on the parts of jRelix that are relevant to the implementation and use of the Geditor.

Section 2.1 explains how to declare a domain and relation in jRelix. Relation initialization is also discussed in this section. Section 2.2 discusses assignments and relational expression in relational algebra. Section 2.3 describes domain algebra. Section 2.4 briefly explains computations. Section 2.5 discusses update commands in jRelix. Section 2.6 describes the event handler which will be used in the implementation of Geditor.

In the following discussions, the jRelix syntax and examples will be given when necessary. The syntax will be presented in typewriter font. Terminals will be quoted and non-terminals will be otherwise. The sign | means or. (...|...|...) means choosing one of the components separated by | inside the brackets. (...)? repeats the component inside the brackets zero or one time.

2.1 Declarations

Declarations of attributes and relations must be made before any further operations can be performed on them. This section describes both domain and relation declarations, initialization of relations, and some system commands to do the house-keeping work.

2.1.1 Domain Declaration

Domain Declaration declares the data type of attributes used in relations. The syntax is as follows:

Syntax

```
"domain" IDList data_type ";"
```

`IDList` specifies the list of attributes being declared separated by comma, and `data_type` specifies the type of the attributes.

jRelix provides eight atomic data types as Table2.1 shows.

Data Type	Short Form	Size
Boolean	bool	1 byte
Short		2 bytes
Integer	intg	4 bytes
Long		8 bytes
Float	real	4 bytes
Double		8 bytes
String	strg	variable
Text		variable

Table 2.1: Data Types in jRelix

Examples:

```
>domain G strg;
>domain T strg;
>domain S, X, Y, C intg;
>domain L strg;
>domain Name strg;
>domain Temp intg;
>domain Pop intg;
```

Complex data types such as computation and nested relational domain are also supported in the current jRelix. For further details, refer to [Bak98] and [Yua98].

To show the information of a specific domain or all the domains currently declared in the system, use the following command:

Syntax

```
"sd" (Identifier)?  ";"
```

When `Identifier` is specified, the above syntax shows the information about this particular domain; otherwise, it shows all the currently declared domains.

To delete a domain from the current system, use the **dd** command:

Syntax

```
"dd" IDList ";"
```

Notice that if any of the attributes specified in the `IDList` are being used in any existing relation, the command will fail. This requires the user to delete all the relations associated with the specified attributes before deleting the attributes.

Examples:

>sd Symb;

>sd;

>dd Pop;

2.1.2 Relation Declaration

The syntax of relation declaration is as follows:

Syntax

```
"relation" IDList "(" IDList ")" (Initialization)?  ";"
```

The first `IDList` specifies the relation being created, and the second `IDList` specifies the attributes of this relation. Relations must have at least one attribute and all these attributes must have been previously declared.

Initialization is optional. The following is the syntax of initialization:

Syntax

```
Initialization:= "<-" ( "{" ConstantTupleList "}" | Identifier)
```

According to the Initialization syntax, there are two ways of initializing relations: providing a list containing the constant tuples, or providing a file name specified by Identifier.

Examples:

>relation MapRelation1(G,T,S,X,Y,C,Symb,L,Name,Temp,Pop);

>relation MapRelation1(G,T,S,X,Y,C,Symb,L,Name,Temp,Pop)<−

{

("S1","Polygon",1,-11481,3257,153204,0,"States","AZ",20,2350725),

("S1","Polygon",2,-11471,3271,153204,0,"States","AZ",20,2350725),

("S1","Polygon",3,-11453,3275,153204,0,"States","AZ",20,2350725),

("S2","Polygon",1,-11475,3271,153204,0,"States","CA",20,29760021),

("S2","Polygon",2,-11449,3300,153204,0,"States","CA",20,29760021),

("S2","Polygon",3,-11462,3343,153204,0,"States","CA",20,29760021),

("S3","Polygon",1,-11963,3401,153204,0,"States","CA",20,29760021),

};

>relation MapRelation2(G,T,S,X,Y,C,Symb,L,Name,Temp,Pop)<−"MapData";

File "MapData" is a jRelix relation file containing the data of all the attributes of MapRelation1. Figure 2.1 shows the content of this file.

```
C62^FPoint^F1^F-11840^F3393^F51000051^F201^FCities^FLos Angeles^F25
^F3485398^F
C87^FPoint^F1^F-12238^F3761^F255255000^F201^FCities^FLos Angeles^F25
^F723959^F
C89^FPoint^F1^F-12193^F3736^F255051051^F201^FCities^FSan Jose AP^F25
^F782248^F
```

Figure 2.1: Contents of MapData

Table 2.2 and 2.3 shows the result of MapRelation1 and MapRelation2. Notice that there are two Los Angeles in MapRelation2. This means that at least one of them is a county of Los Angeles.

To show information on relations, use the **sr** command:

Syntax

```
"sr" (Identifier)?  ";"
```

G	T	S	X	Y	C	Symb	L	Name	Temp	Pop
S1	Polygon	1	-11481	3257	153204	0	States	AZ	20	2350725
S1	Polygon	2	-11471	3271	153204	0	States	AZ	20	2350725
S1	Polygon	3	-11453	3275	153204	0	States	AZ	20	2350725
S2	Polygon	1	-11475	3271	153204	0	States	CA	20	29760021
S2	Polygon	2	-11449	3300	153204	0	States	CA	20	29760021
S2	Polygon	3	-11462	3343	153204	0	States	CA	20	29760021
S3	Polygon	1	-11963	3401	153204	0	States	CA	20	29760021

Table 2.2: MapRelation1

G	T	S	X	Y	C	Symb	L	Name	Temp	Pop
C62	Point	1	-11840	3393	51000051	201	Cities	Los Angeles	25	3485398
C87	Point	1	-12238	3761	51000051	201	Cities	Los Angeles	20	723959
C89	Point	1	-12193	3736	51000051	201	Cities	San Jose AP	25	782248

Table 2.3: MapRelation2

`Identifier` specifies a particular relation name. If a relation name is specified, the **sr** command shows the information on this relation; otherwise, it shows information on all the relations currently defined in the system.

To print the content of a relation on screen, use the **pr** command:

Syntax

```
"pr" Expression ";"
```

The command **dr** is used to remove the relations specified in `IDList` from the system:

Syntax

```
"dr" IDList ";"
```

Examples:

>sr MapRelation1;

>sr;

>pr MapRelation2;

>dr MapRelation1, MapRelation2;

2.2 Relational Algebra

Relational algebra consists of a set of functional operations on one or two relations and produces a result relation. jRelix first constructs expressions by using various operators and then produces the result relation by assignment or incremental assignment. In this section, we first examine how to use assignment and incremental assignment, and then we discuss relational expressions.

2.2.1 Assignment and Incremental Assignment

An assignment (<−) creates a relation using the result of a relational expression. An incremental assignment (<+) adds the result of a relational expression to an existing relation. The syntax is as follows:

Syntax

```
Identifier ( "<-" | "<+" ) Expression |
Identifier "[" IDList( "<-" | "<+" )ExpressionList "]" Expression
```

`Identifier` specifies the name of the result relation. `Expression` indicates the source relation. For assignment operation, jRelix creates a new relation named by `Identifier` which consists of the same domain and data as the source relation. If the result relation has the same name as an existing relation in the current system, the existing relation will be removed first. The source relation remains unaffected.

Examples:

>MapCopy<−MapRelation1;

>MapRelation1<+MapRelation2;

In the above examples, MapCopy obtains a copy of original MapRelation1. The result MapRelation1 is a merge of the original MapRelation1 and MapRelation2.

2.2.2 Relational Expressions

Relational Expressions can be divided into two categories: unary operations and binary operations. Unary operations take one relation as input and generate one relation as output. Binary operations take two relations as input and produce one result relation.

Unary Operations

Projection, selection and T_selection are unary operations.

- Projection

  Projection creates a subset of the source relation specified by Expression. It extracts a subset of the attributes of the source relation by `IDList`. Duplicate tuples will be removed from the result relation. The syntax is as follows:

  Syntax

  ```
  "[" (IDList)? "]" in Expression
  ```

- Selection

  Selection also returns a subset of the source relation specified by Expression. Unlike projection, the result relation contains all the attributes of the source relation. However, the tuples in the result relation are those satisfying the condition of the SelectionClause. The syntax is as follows:

  Syntax

  ```
  "where" SelectClause "in" Expression
  ```

- T_selection

  Projections and selections can be combined into one expression to form T_Selections. In a T_Selection, first perform the selection, and then perform the projection. The syntax is as follows:

  Syntax

  ```
  "[" (IDList)? "]" "where" SelectClause "in" Expression
  ```

  Examples:

  >LayerName<−[L,Name] in MapRelation1;

  >MapAZ<−where Name="AZ" in MapRelation1;

>LargePop<−[Name,Pop]where Pop>10000000 in MapRelation1;

In the above examples, the LayerName obtains a relation containing all the layers and all the different map feature names in MapRelation1. MapAZ contains a subset of MapRelation1 that contains only the tuples of Arizona state. LargePop contains the Name and Pop of the state(s) whose population is over 10,000,000 (Pop>1000000).

L	Name
States	AZ
States	CA

LayerName

Name	Pop
CA	29760021

LargePop

G	T	S	X	Y	C	Symb	L	Name	Temp	Pop
S1	Polygon	1	-11481	3257	153204	0	States	AZ	20	2350725
S1	Polygon	1	-11471	3271	153204	0	States	AZ	20	2350725
S1	Polygon	1	-11453	3275	153204	0	States	AZ	20	2350725

MapAZ

Figure 2.2: Results of LayerName, MapAZ, and LargePop

Binary Operations

Binary operations in jRelix include μ_joins and σ_joins. The result of μ_joins and σ_joins are also relations.

The syntax of join is as follows:

Syntax

```
Expression JoinOperator Expression |
Expression "[" ExprList ":" JoinOperator ( ":" )? ExprList
   "]" Expression
```

Since Geditor does not use σ_joins, we only discuss μ_joins in this section.

μ_joins are a generalization of set operations on relations. The most popular two μ_joins are natural join (**ijoin**) and union join (**ujoin**). In general, μ_join operators can be defined in terms of three components — center, left and right. Given two relations R(X,Y), S(Y,Z), the three components are defined as follows:

$$\text{center}(R,S)=\{(x,y,z)\mid(x,y)\in R \text{ and} (y,z)\in S\}$$

$$\text{left}(R,S)=\{(x,y,dc)\mid(x,y)\in R \text{ and } \forall z((y,z)\notin S)\}$$

$$\text{right}(R,S)=\{(dc,y,z)\mid(y,z)\in S \text{ and } \forall x((x,y)\notin S)\}$$

where dc is a null value.

For **ijoin**, we have R ijoin S=center(R,S).

For **ujoin**, we have R ujoin S=center(R,S)$\bigcup$left(R,S)$\bigcup$right(R,S).

Example:

```
>domain Humid intg;
>relation Humidity(Name,Humid)<−{("AZ",80),("CA",90)};
>MapNew<−MapRelation1 ijoin Humidity
```

G	T	S	X	Y	C	Symb	L	Name	Temp	Pop
S1	Polygon	1	-11481	3257	153204	0	States	AZ	20	2350725
S1	Polygon	2	-11471	3271	153204	0	States	AZ	20	2350725
S1	Polygon	3	-11453	3275	153204	0	States	AZ	20	2350725
S2	Polygon	1	-11475	3271	153204	0	States	CA	20	29760021
S2	Polygon	2	-11449	3300	153204	0	States	CA	20	29760021
S2	Polygon	3	-11462	3343	153204	0	States	CA	20	29760021
S3	Polygon	1	-11963	3401	153204	0	States	CA	20	29760021

MapRelation1

Name	Humid
AZ	80
CA	90

Humidity

G	T	S	X	Y	C	Symb	L	Name	Temp	Pop	Humid
S1	Polygon	1	-11481	3257	153204	0	States	AZ	20	2350725	80
S1	Polygon	2	-11471	3271	153204	0	States	AZ	20	2350725	80
S1	Polygon	3	-11453	3275	153204	0	States	AZ	20	2350725	80
S2	Polygon	1	-11475	3271	153204	0	States	CA	20	29760021	90
S2	Polygon	2	-11449	3300	153204	0	States	CA	20	29760021	90
S2	Polygon	3	-11462	3343	153204	0	States	CA	20	29760021	90
S3	Polygon	1	-11963	3401	153204	0	States	CA	20	29760021	90

MapNew

Figure 2.3: Example of ijoin

In the above example, the result MapNew obtains a new attribute, Humid, which indicates the humidity of each state. In MapNew, the humidity of AZ and CA is 80 and 90 respectively. This is exactly the same as relation Humidity specifies. See Figure 2.3.

In this example, the common attribute of MapRelation1 and Humidity is Name. Since this attribute has the same name in both relations, it is not necessary to specify the attribute name explicitly in the above **ijoin** expression. If the common attribute names are different in the two source relations, for example, the attribute Name in Humidity relation changes to State_Name, we must specify the common attribute in the **ijoin** expression as follows:

Example: MapNew<−MapRelation1[Name ijoin State_Name] Humidity;

As a result, both Name and State_Name appear in the MapNew relation and the values of the two attributes for all tuples in MapNew are identical.

Example: MapUnion<− MapRelation1 ujoin MapRelation2;

Since the number, names, and types of the attributes in MapRelation1 are the same as those in MapRelation2, the MapUnion in the above example is a merge of MapRelation1 and MapRelation2 as Table 2.4 shows. This is not the full **ujoin**, which can combine relations with different attributes.

G	T	S	X	Y	C	Symb	L	Name	Temp	Pop
C62	Point	1	-11840	3393	51000051	201	Cities	Los Angeles	25	3485398
C87	Point	1	-12238	3761	51000051	201	Cities	Los Angeles	20	723959
C89	Point	1	-12193	3736	51000051	201	Cities	San Jose AP	25	782248
S1	Polygon	1	-11481	3257	153204	0	States	AZ	20	2350725
S1	Polygon	2	-11471	3271	153204	0	States	AZ	20	2350725
S1	Polygon	3	-11453	3275	153204	0	States	AZ	20	2350725
S2	Polygon	1	-11475	3271	153204	0	States	CA	20	29760021
S2	Polygon	2	-11449	3300	153204	0	States	CA	20	29760021
S2	Polygon	3	-11462	3343	153204	0	States	CA	20	29760021
S3	Polygon	3	-11963	3401	153204	0	States	CA	20	29760021

Table 2.4: Result of MapUnion from the ujoin Example

2.3 Domain Algebra

Domain Algebra provides a set of operations applied on attributes. A thorough description of domain algebra can be found in [Mer84]. In this section, we are going to discuss virtual domains, horizontal operations and vertical operations.

- Virtual domains

  Virtual domains are declared on a set of actual domains or virtual domains which are subsequently based on actual domains. They belong to no relation until they are actualized by a *projection* or *selection* operation.

  The syntax is as follows:

  Syntax

  ```
  "let" Identifier Expression ";"
  ```

  Examples:

  >let State_Name be Name; <<virtual domain declaration

  >StateNames<−[State_Name] where L="States" in MapRelation1;
  <<virtual domain actualization

- Horizontal Operations

  Horizontal operations of domain algebra work on a single tuple of a relation. When the Expression in the above syntax is a horizontal_expression, such as constant definition, renaming, arithmetic functions, and conditional expression(if-then-else), it becomes a horizontal operation.

  Examples:

  >let twopie be 3.1415926; <<constant definition

  >let angle be acos(twopie); <<arithmetic functions

  >let sign be if X<0 then -1 else if X>0 then 1 else 0;
  <<conditional expression

  >let px be X; <<renaming

  >let POP be Pop/100;

- Vertical Operations

Vertical operations of domain algebra work on attribute values of all tuples in a relation. Four types of vertical operations are defined in jRelix. (Only the first two are implemented in the current version.)

- Simple reduction
- Equivalence reduction
- Functional mapping
- Partial functional mapping

Simple reduction produces a single result from the values of all tuples of a single attribute in a relation. Equivalence reduction first divides all the tuples into groups based on the grouping expression (by), and then generates one result from the values of tuples of an attribute in each group.

The syntax is as follows:

Syntax

```
"let" Identifier "be" "red" AssoCommuOperator "of" Expression";"
"let" Identifier "be" "equiv" AssoCommuOperator "of" Expression
   "by" ExpressionList ";"
```

The `AssoCommuOperator` can be one of the following associative and commutative operators: (“or”|“|”)|(“and”|“&”)|“min”|“max”|“+”|“∗”|(“ijoin”|“natjoin”)|“ujoin”|“sjoin”

Examples:

>let tot_pop be red + of Pop;

>let sub_tot be equiv + of Pop by Name;

>R<−[Name, tot_pop] in MapRelation2;

>S<−[Name, sub_tot] in MapRelation2;

The tot_pop in the first example calculates the total population of all the cities in MapRelation2. The sub_tot in the second example calculates the sub-totals of the population in each city.

Name	tot_pop
Los Angeles	4991607
San Jose AP	4991607

R

Name	sub_tot
Los Angeles	4209357
San Jose AP	782248

S

Figure 2.4: Results of R and S

- Functional mapping

  Functional mapping processes a relation by first sorting it according to a specified set of attributes. Then instead of working with a set of tuples as a whole, it manipulates individual tuples according to the specified operator.

  The syntax is:

  Syntax

```
"let" Identifier "be" "fun" Operator "of" Expression
      order ExpressionList ";"
```

  The Operator in the above syntax includes the `AssoCommuOperator` discussed above and the following ordered operators: “cat”|“−”|“/”|“mod”|“∗∗”|“pred”|“succ”

  Examples:

  >R<− where Name=“AZ” in MapRelation1;

  >let X’ be fun succ of X order S;

  >T<− [S,X,X’] in R;

  In the above example, first, all the tuples in MapRelation1 are sorted by S. Then for each tuple, generate X’ as the successor of X. The calculation of successor is cyclic, that is, for each tuple except the last one, the successor of X is the X value of the next tuple. For the last tuple, the successor of X is the X value of the first tuple. The result of T is displayed in Figure 2.5.

- Partial functional mapping

S	X	X’
1	-11481	-11471
2	-11471	-11453
3	-11453	-11481

T

Figure 2.5: Result of Relation T

Partial functional mapping first divides the tuples into groups based on the grouping expression (by). Then in each group, it sorts the tuples according to the ordering expression (order). Finally, according to the `Operator`, it manipulates each tuple in each different group similar to functional mapping.

The syntax is as follows:

Syntax

```
"let" Identifier "be" "par" Operator "of" Expression
    "order" ExprssionList "by" ExpressionList ";"
```

Examples:

>let X’ be par succ of X order S by G;

>R<−[G,S,X,X’] in MapRelation1;

G	S	X	X’
S1	1	-11481	-11471
S1	2	-11471	-11453
S1	3	-11453	-11481
S2	1	-11475	-11449
S2	2	-11449	-11462
S2	3	-11462	-11475
S3	1	-11963	-11963

R

Figure 2.6: Result of Relation R

2.4 Computations

Computations are user-defined constructs that implement procedural abstraction in jRelix. Each computation contains a group of statements that perform a specific task. The formal syntax is as follows:

Syntax

```
"comp" Identifier "("(ParameterList)?")" is ComputaionBody ";"
```

In this section, we briefly discuss computation with an example relevant to Geditor implementation, which does not use parameters. Please refer to [Bak98] for a complete explanation of computations in jRelix.

Example:

```
comp AssignComp () is
{
    MapCopy<−MapRelation1;
};
```

This computation can be invoked by means of a top_level call in jRelix as indicated below:

```
>AssignComp();
```

As a result, MapCopy obtains a copy of the MapRelation1.

2.5 Updates

The **update** operation allows the user to change values of specified attributes in certain tuples. These attributes could be selected by a **using** clause that selects tuples from the relation by relational algebra operations. **Update**s could also be used to add or delete tuples from the relation. The syntax for **update** statements is as follows:

Syntax

```
"update" Identifier ("add" | "delete" )Expression ";" |
"update" Identifier "change" (StatementList)?  (UsingClause)? ";"
 UsingClause:="using" JoinOperator Expression |
              "using" "[" IDList ":" JoinOperator (":")?  ExpressionList
               "]" Expression |
              "using" Identifier|
              "using" "(" Expression ")"
```

Examples:

>update MapRelation1 add MapRelation2;
<< MapRelation1 becomes a merge of MapRelation2 (Cities) with original MapRelation1 (States)

>MapCity<−where L="Cities" in MapRelation1;
<< MapCity contains the Cities tuples in MapRelation1

>update MapRelation1 delete MapCity;
<< MapRelation1 now does not contain any Cities tuples.

>relation CA(Name)<−{("CA")};
<< a relation only has one attribute and one tuple

>update MapRelation1 change Temp<−18 using ijoin CA;
<< The temperature (Temp) of CA in MapRelation1 changes from 20 to 18 degree.

When using **update** to add tuples, the number, types and positions of all the attributes of the two relations must be the same. When using **update** to delete tuples, the attributes in the **using** clause can be a subset of the relation being updated. In the above example, MapRelation1 and MapRelation2, MapRelation1 and MapCity have the same attribute number, types and positions. When using **update** to change the relation MapRelation1, first perform the relational algebra specified in the **using** clause with MapRelation1: MapRelation1 ijoin CA. Then in the MapRelation1, change the temperature (Temp) of those tuples that participated in the **ijoin** (having common attribute values in MapRelation1 and CA) from 20 to 18 degrees. The result of this operation is shown in Table 2.5.

G	T	S	X	Y	C	Symb	L	Name	Temp	Pop
S1	Polygon	1	-11481	3257	153204	0	States	AZ	20	2350725
S1	Polygon	2	-11471	3271	153204	0	States	AZ	20	2350725
S1	Polygon	3	-11453	3275	153204	0	States	AZ	20	2350725
S2	Polygon	1	-11475	3271	153204	0	States	CA	18	29760021
S2	Polygon	2	-11449	3300	153204	0	States	CA	18	29760021
S2	Polygon	3	-11462	3343	153204	0	States	CA	18	29760021
S3	Polygon	1	-11963	3401	153204	0	States	CA	18	29760021

Table 2.5: Result of MapRelation1 from: update MapRelation1 change Temp<-18 using ijoin CA;

2.6 Event Handler

Event handlers are procedures (computations) to process events. Events are system-generated procedure calls. In the jRelix version before Geditor was implemented, events were generated by **update**s. However, generally speaking, events may arise from operations such as executing a command (such as update), user's mouse click, a database read, and a notice from the Internet. In this section, we discuss the event handlers for **updates**. In chapter 4, the new event handler that includes those processing events generated by user's mouse click will be discussed in detail.

Current events are generated by **updates**. When jRelix meets an **update** command, an event is generated. Before jRelix executes the **update** command, it searches the system table to find the particular event handler that should be invoked before executing the command. If such an event handler exists, it will be triggered. After the **update** command is executed, jRelix searches the system table again to find the particular event handler that should be invoked after the executing of this **update** command. If such an event handler is found, it will be invoked and executed.

Computations are used to define event handlers in jRelix. To distinguish between event handlers and user-defined computations, special names must be assigned to event handlers.

2.6.1 Naming Event Handlers

The syntax of Computations is expanded to include event handlers as follows:

According to the above syntax, there are four components of EventName:

- **prefix**

Syntax

```
Computation:= "Comp" CompName "(" (ParameterList)? ")" "is"
               ComputationBody
CompName    := Identifier | EventName
EventName   := ( prefix ":" )?  action ":" relation ( "[" attribute-list
               "]" )?
```

`prefix` could be either *pre* or *post*. If `prefix` is omitted, it is *post* by default. *Pre* means that the event handler could be invoked before the execution of the corresponding event. *Post* means the event handler could be executed after the execution of the corresponding event.

- **action**

  `action` specifies what **update** operation this event handler would process. There are three possible types of action: *add*, *delete* or *change*.

- **relation**

  `relation` specifies the name of the relation to be updated.

- **attribute-list**

  `attribute-list` is optional. For action *add* or *delete*, it is always omitted. For action *change*, the `attribute-list` could be any subset of the original attributes. If the subset is not empty, *change* action on this subset of attributes will trigger the event handler. If the subset is empty, *change* action on any attribute(s) in the relation will trigger this event handler.

  The following are some examples of valid event names for relation R with attributes a,b,c.

  Examples:

  pre: add: R

  post:delete:R

  add: R

  delete: R

  pre:change:R[a,b,c]

  post:change:R

Now it is ready to define an event handler.

2.6.2 Defining an Event Handler

Since computation is used to define an event handler, defining an event handler is similar to defining a computation with a special event_name.

Syntax

```
"comp" event_name() "is"
{
    statements;
}
```

For **update** events, the event handlers do not take parameters.

2.6.3 Event Handler On/Off

When an event handler is defined, its state is set to *On* which means it will be executed when the corresponding event occurs. To turn an event handler *Off*, use the following command:

Syntax

```
"eventoff" event_name ";"
```

When an event handler is turned *Off*, it will not be invoked when the corresponding event happens even if this event handler exists in the system.

To turn an event handler *On* again, use the following command:

Syntax

```
"eventon" event_name ";"
```

2.6.4 Printing Event Handlers

The command

Syntax

```
"pr" command ";"
```

will print out the definition of the event handler with the specified eventname.

2.6.5 Deleting Event Handlers

To delete an event handler from the current system, use the command

Syntax

```
"dr" event_name ";"
```

For the **update** operation, the affected relation would be separated into three pieces: *Trigger*, *New* and *Rest*. These three pieces are useful to achieve the *undo* operation which recovers the former state of the database before the **update** command is executed. For more details, please refer to [Sun00].

Here is a simple example of event handlers:

Examples:

```
>relation CA(Name)<−{("CA")};
>comp pre:change:MapRelation1() is
  {
      MapCopy<−MapRelation1;
  }
>update MapRelation1 change Temp<−18 using ijoin CA;
```

The above example defines an event handler for the *change* `action` of the **update** command of MapRelation1. Since the `prefix` in the event_name of this event handler is *pre*, before updating MapRelation1, the relation MapCopy obtains a copy of the original MapRelation1.

Chapter 3

User's Manual for Geditor

This chapter is a tutorial that describes how to use the GIS editor (Geditor) in jRelix to display and edit maps. Section 3.1 describes the functions in the Layers menu which are related to starting and exiting Geditor. Section 3.2 discusses how to add layers and change the states of the layers. Section 3.3 explains how to change colors and symbols in a particular layer. The operations of editing the titles and labels of the legends are also discussed in this section. In section 3.4, the three different ways of querying the map are presented. Some details such as changing the default color of selected map features are also explained. Section 3.5 briefly explains the Help message the Geditor provides for the user.

Users of Geditor can be classified into two categories: *Programmer-User* and *End-User*. The Programmer-Users work in the jRelix environment. They are responsible for the preparation of map data in relations, using the correct syntax to call the Geditor and writing the correct event handlers to perform the spatial queries. The End-Users work in the Geditor window. They edit the map, generate thematic maps, perform spatial queries, and so on. They have different interests in the data. Programmer-Users are interested in the manipulation of the map data using jRelix statements or commands. End-Users are interested in viewing the map data graphically and generating new meaningful and real data by using the functions provided in the Geditor window.

3.1 Starting and Exiting Geditor

3.1.1 Starting jRelix

Geditor must be invoked from jRelix as a **gedit** operator. Therefore, jRelix must be started first. Suppose both the Java run-time system and jRelix software are successfully installed on the user system. To start jRelix, the following command is typed on the command line of the operating system:

$>$ java JRelix

As a result, jRelix copyright information is displayed as illustrated in Figure 3.1 . After that, jRelix shows its prompt sign "$>$" and waits for the user input.

```
[roxy][/localhome/ychen54/jRelix/Code] java JRelix
+-------------------------------------+
|          RelixJava version 0.7      |
|Copyright (c) 1997, 1998, Aldat Lab  |
|     School of Computer Science      |
|          McGill University          |
+-------------------------------------+
>
```

Figure 3.1: init Screen of jRelix

3.1.2 Map Relation

Before starting the Geditor, the map relation that stores both the *graphical* and *descriptive* data of the map must also be created. In the map relation, the attributes that represent graphical information of the map, such as the shape, color, x and y coordinates are called *graphical attributes*. Other attributes such as population, temperature and income, are called *descriptive attributes* because they are non-spatial or descriptive data that describe the features of the corresponding map feature. Moreover, the graphical attributes are divided into two categories: the *basic graphical attributes* and the *additional graphical attributes*. The five attributes representing group, type, sequence, x coordinate and y coordinate of the points that will be drawn on the screen are called the *basic graphical attributes*. They are required by every map relation in this implementation of Geditor. Other graphical attributes representing items such as color, symbol, and layer, are the *additional graphical* *attributes*. They are optional. The following is an example of the correct attributes definition of a map relation used by Geditor:

Attribute_category	**Attribute_type**
Group	string
Type	string
Seq	int
x	int
y	int
Color	int
Symb	int
Layer	string

In the above list, the attribute names could be chosen arbitrarily by the user, but it is the user's responsibility to provide correct types for these attributes. Table 3.1 shows a sample map relation which contains all the eight graphical attributes: G(Group), T(Type), S(Seq), X(x), Y(y), C(Color), S(Symb) and L(Layer). The other three attributes: Name, Temp and Pop are descriptive attributes.

G	T	S	X	Y	C	Symb	L	Name	Temp	Pop
C1	Point	1	-11840	3393	51000051	201	Cities	Los Angeles	25	3485398
C2	Point	1	-12238	3761	51000051	201	Cities	San Francisco AP	20	723959
R1	Polyline	1	-12062	3483	153000000	0	Rivers	Mississipi	14	0
R1	Polyline	2	-11561	3280	153000000	0	Rivers	Mississipi	14	0
S1	Polygon	1	-11481	3257	153204	0	States	AZ	20	2350725
S1	Polygon	2	-11471	3271	153204	0	States	AZ	20	2350725
S2	Polygon	1	-11450	3396	153204	0	States	CA	20	29760021
S2	Polygon	2	-11445	3399	153204	0	States	CA	20	29760021
S3	Polygon	1	-12000	3906	153204	0	States	NV	-10	1201833
S3	Polygon	2	-12000	3900	153204	0	States	NV	-10	1201833
S4	Polygon	1	-11689	4415	153204	0	States	OR	8	2842321
S4	Polygon	2	-11691	4412	153204	0	States	OR	8	2842321

Table 3.1: Example of Map Relation:MapRelation1

As indicated in Table 3.1, Color must be an integer showing its RGB values. The first, second and third three digits show the red value R, green value G, and blue value B respectively. The value range of every three digits is 0..255. For example, if the color has R value 51, G value 0, and B value 51, the value of the color attribute in the map relation should be 51000051. Notice that the leading 0 must be omitted.

The type of symbol is integer. In the current version of Geditor, only point map features can be represented by different symbols. Geditor ignores all the symbol values for polylines and polygons. Six different symbols are implemented for points. They are: 201—solid oval, 202—double hollow circle, 203—hollow circle with a solid oval inside, 204—hollow triangle, 205—solid triangle, 206—flag. When the value of the symbol attribute of a point is none of the above, Geditor assigns value 201 to this point. As a result, the point will be displayed as a solid oval.

3.1.3 Starting Geditor

Now, we are ready to start the Geditor using **gedit** operator. In general, the syntax to start the GIS editor is as follows:

basic_graphical_attribute_list[*additional_graphical_attribute_list*] gedit *rel_name* ;

The first list is required by every **gedit** expression. It contains the five basic graphical attributes by positions. From the left to the right, the sequence of the five attributes must be those representing group, type, sequence, x coordinate and y coordinate respectively. The user is responsible for providing the correct number and order of the attributes in the first list. For the MapRelation1 indicated by Figure 3.2, the only possible correct basic_graphical_attribute_list is [G,T,S,X,Y].

```
>R<-[G,T,S,X,Y][Color=C,Symbol=Symb,Layer=L]gedit MapRelation1;
```

Figure 3.2: gedit Example (1)

The second list is optional. It shows the additional graphical attributes in the map relation by category–name pair. Arbitrary number of equations are allowed in the second list, but we have implemented three: Color=*attr_name*, Symbol=*attr_name*, and Layer=*attr_name*. For example, in Figure 3.2, the second list of the **gedit** operator contains three equations: Color=C, Symbol=Symb, and Layer=L. The positions of these equations do not matter, and some or all may be omitted. When the Color and/or Symbol equation in the second list is missing, Geditor assigns default color and/or symbol to each layer of the map; when the Layer equation is missing in the second list, Geditor assumes that the map relation contains only one layer. For example, in Table 3.2, MapRelation2 has only one layer so that it has no layer attribute. Therefore, the corresponding **gedit** expression contains only two equations: Symbol=Symb and Color=C as Figure 3.1.3 shows. (Note that Figure 3.1.3 should also work for MapRelation1.) In an extreme case as Table 3.3 shows, MapRelation3 has only one layer and no color or symbol attributes. The gedit expression omits the whole second list as illustrated in Figure 3.4.

G	T	S	X	Y	C	Symb	Name	Temp	Pop
S1	Polygon	1	-11481	3257	153204	0	AZ	20	2350725
S1	Polygon	2	-11471	3271	153204	0	AZ	20	2350725
S2	Polygon	1	-11450	3396	153204	0	CA	20	29760021
S2	Polygon	2	-11445	3399	153204	0	CA	20	29760021
S3	Polygon	1	-12000	3906	153204	0	NV	-10	1201833
S3	Polygon	2	-12000	3900	153204	0	NV	-10	1201833
S4	Polygon	1	-11689	4415	153204	0	OR	8	2842321
S4	Polygon	2	-11691	4412	153204	0	OR	8	2842321

Table 3.2: Example of Map Relation:MapRelation2

```
>R<-[G,T,S,X,Y][Symbol=Symb,Color=C]gedit MapRelation2;
```

Figure 3.3: gedit Example (2)

```
>R<-[G,T,S,X,Y] gedit MapRelation3;
```

Figure 3.4: gedit Example (3)

G	T	S	X	Y	Name	Temp	Pop
S1	Polygon	1	-11481	3257	AZ	20	2350725
S1	Polygon	2	-11471	3271	AZ	20	2350725
S2	Polygon	1	-11450	3396	CA	20	29760021
S2	Polygon	2	-11445	3399	CA	20	29760021
S3	Polygon	1	-12000	3906	NV	-10	1201833
S3	Polygon	2	-12000	3900	NV	-10	1201833
S4	Polygon	1	-11689	4415	OR	8	2842321
S4	Polygon	2	-11691	4412	OR	8	2842321

Table 3.3: Example of Map Relation:MapRelation3

After the user inputs the statement containing the gedit expression, a Geditor window appears.(See Figure 3.5.) Initially, the internal frame "Map View" is empty.

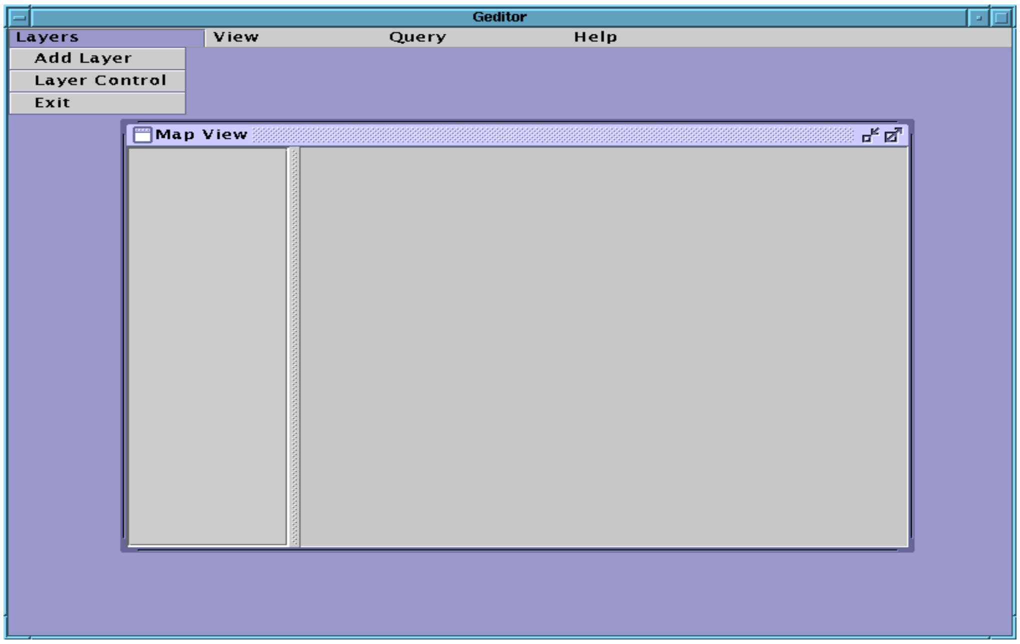


Figure 3.5: Geditor Window

As a unary operator, **gedit** is functional. It does not change the content of its operand, that is, the map relation. Furthermore, the value of the **gedit** expression is also the same as its original operand (the map relation). Therefore, R will be assigned to the original map relation, which is MapRelation1, MapRelation2 or MapRelation3 in the above examples. Actually, as soon as **gedit** calls and displays the Geditor window, the statement containing the **gedit** expression returns. The jRelix system continues to accept commands and statements in the command line at the same time the End-User performs GIS operations in the Geditor window.

3.1.4 Exiting Geditor and jRelix

To exit the Geditor Window, the user clicks the Close command at the upper left corner of the window or clicks Exit in the “Layers” menu. Upon receiving this user operation, jRelix closes the Geditor window and returns to the command line environment with a “>” prompt sign waiting for user input.

To further exit the jRelix system, the user types “quit;”after the system prompt sign. The jRelix performs its clean-up procedures and returns to the original operating system.

3.2 Layers

A layer is a logical separation of map data, such as city, road, and river. Usually the map relation contains more than one layer and Geditor allows the user to work on a single layer or multiple layers. As described in section 3.1, initially, the Map View window is empty. The user needs to add layer(s) to let the Geditor know what layer(s) should be displayed in the Map View window. After a layer is added, a set of switches is attached to the layer to indicate its current state. This section describes how to add layers and change their states through “Add Layer” and “Layer Control”. These two functions are included in the leftmost pull-down menu “Layers” as Figure 3.6 shows.

Figure 3.6: Layers Menu

3.2.1 Add Layer

On the top of the Geditor window is a menu bar containing the names of four pull-down menus. Click the Layers pull-down menu and select the Add Layer item. A dialog box titled “Add Layer”

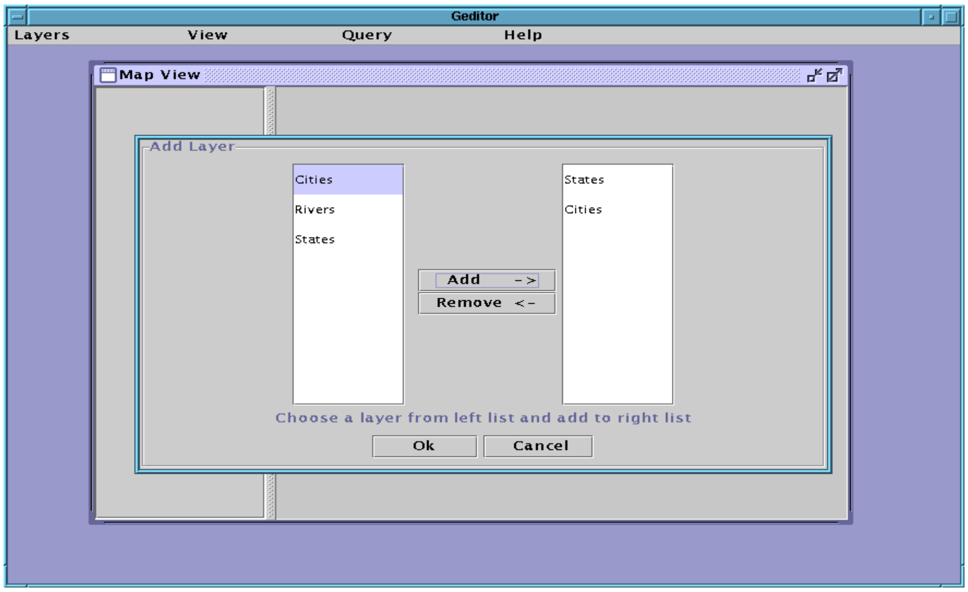


Figure 3.7: Add Layer Dialog

pops up as illustrated in Figure 3.7. In the dialog box, there are two lists and four buttons. The left list shows all the layers in the map relation and the right list displays the layers that have been added by the user. Clicking an item in the left list and then clicking the Add button adds this layer to the right list. The user can also select an item from the right list and click Remove button to remove this layer. Only one item can be selected in any list at a time. Under the two lists, there is a message line reminding the user what to do next. When the user's operations are not in a proper sequence, error messages are displayed on this message line. Clicking on OK button records the user's input and closes the dialog. The Cancel button discards the user's input and closes the dialog. Notice that the functions of the OK and Cancel buttons are the same in every dialog box in Geditor. Therefore, the description of their functions will not be mentioned again in the following discussions.

After the Add Layer dialog box is closed with the OK button, the Geditor updates the Map View window. It draws all the layers added by the user in a sequence exactly the same as the sequence of the layers that appear on the right list of Add Layer dialog. A latter layer could overwrite part or all of a former layer on the screen. Figure 3.8 shows the updated Map View resulting from the Add Layer dialog in Figure 3.7. There are three layers in the map relation: Cities, States and Rivers. The user adds two layers to the right list: States and Cities respectively. Therefore, Geditor draws

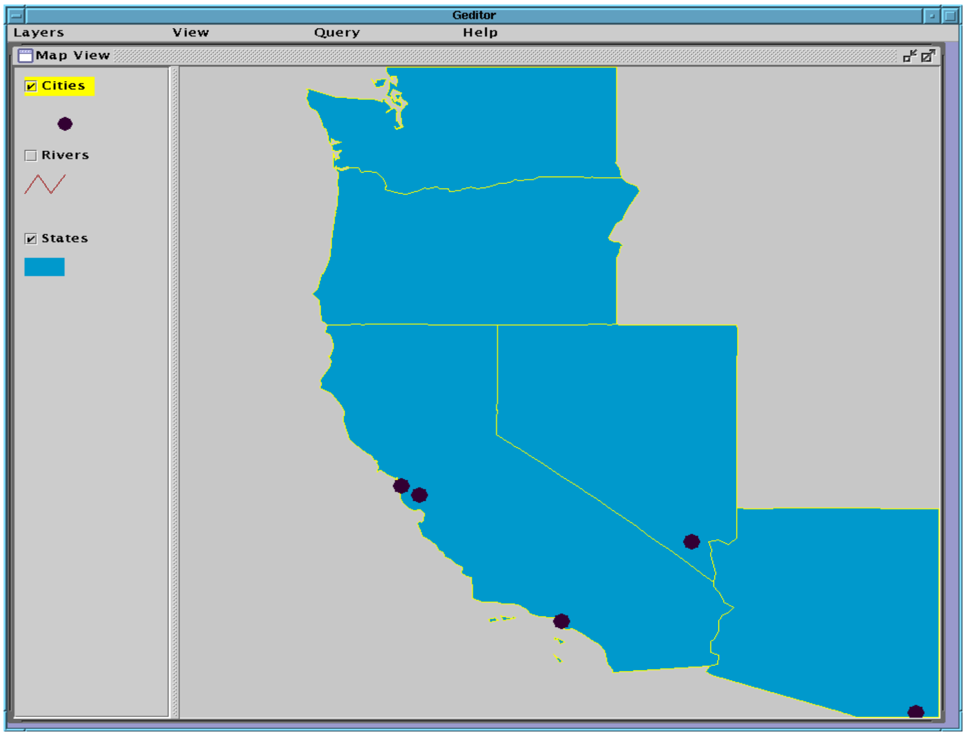


Figure 3.8: Updated Map View Window after Add Layer

states first and then the cities. As a result, both Cities and States are displayed properly in Map View as Figure 3.8 shows. However, if the user adds Cities and then the States in the Add Layer dialog, Geditor will draw the cities first and then the states. As a result, the user can only see one States layer in the Map View window because all the cities are overwritten by the states. Almost all the standard GIS products perform this operation in the same way. Therefore, the user should pay attention to the sequence of adding layers in the Add Layer dialog.

After the Add Layer operation is performed, every layer in the map relation obtains a pair of Boolean switches showing its state: Show/Hide and Active/Inactive. At any time, zero, single or multiple layers can be shown, but among the shown layer(s), only one layer can be active. When the Add Layer operation has just been performed, the states of these layers are as follows: Those that appeared in the right list of the Add Layer dialog are currently "Show"; others are "Hide". The last one in the right list of Add Layer dialog is currently "Active"; others are "Inactive". The user can tell the current state of each layer from the legend split window of Map View. Each layer has a title and a graphical sample legend in the legend window. If the layer is "Show", the title of the layer is checked with a tick; otherwise it is unchecked. If the layer is "Active", the title is highlighted in the legend window. See the legend split window in Figure 3.8 for the results from the Add Layer operation in Figure 3.7. The default highlight color is yellow and this can be changed if the user uses the "Options" function. Details will be discussed in Section 3.4.5.

An active layer may be used for thematic mapping, expression builder, spatial query, etc. in the View and Query menus. In this case, the purpose of activating a layer is to select that layer for subsequent functions.

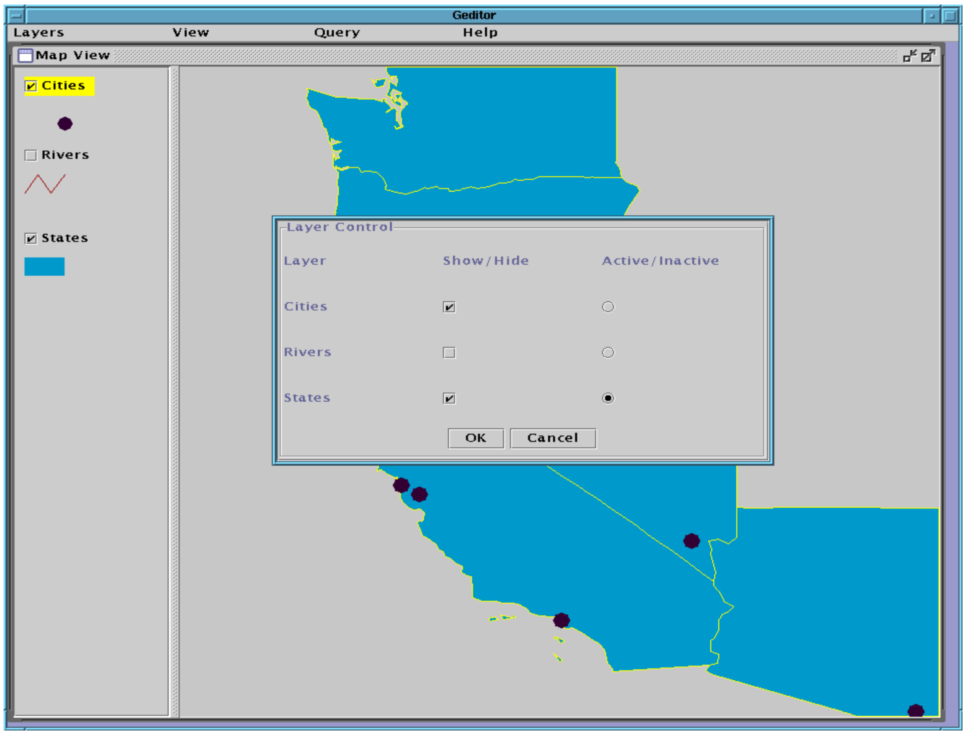


Figure 3.9: Layer Control dialog box

3.2.2 Layer Control

On the menu bar, click the Layers menu and select the Layer Control item. A Layer Control dialog box pops up as Figure 3.9 shows. All the layers in the original map relation are listed in this Layer Control dialog. There are three columns showing the layer names, the Show/Hide and the Active/Inactive state of each layer. If the checkbox under the Show/Hide column is checked, the corresponding layer is currently shown; otherwise it is hidden. If the radio box under the Active/Inactive column is checked, the corresponding layer is active; otherwise, it is inactive. Since only one layer can be active at any time, the radio boxes are mutual exclusive. The user can change the current active layer by simply clicking the radio boxes. The Show/Hide state cannot be changed using Layer Control. If users need to change a "Show" layer to "Hide", they must remove a layer using the Add Layer function. Similarly, if users need to change a "Hide" layer to "Show", they must add it using the Add Layer dialog. Click the OK button after finishing the input in the Layer Control dialog.

If the Layer Control is closed with the OK button, Geditor updates the current state of each layer according to the user input. The Map View window does not need to be updated because no Show/Hide state was changed in the Layer Control dialog. The legend window will be updated accordingly. For example, if the user changes the current active layer to States in the Layer Control dialog as Figure 3.9 shows, the legend window of Map View will be updated as Figure 3.10 shows. The user will notice the yellow bar changed from Cities to States in this legend window.

3.3 View

Up to now, we can display a map from a relation using the Layers menu. The next operation the user may need is to change the view of the current map, such as to change the color of map features. In this section, we are going to discuss how to change the color and symbol in a uniform layer and how to perform thematic mapping in a non-uniform layer. Legend Editor will also be explained in this section. All these functions are included in the View pull-down menu of Geditor as Figure 3.11 shows.

3.3.1 Uniform Layer

The layers can be divided into *uniform layers* and *non-uniform layers*. If all the map features in the same layer have the same color and symbol, this layer is called *uniform layer*; otherwise, it is called *non-uniform layer*. The user can use the Uniform Layer submenu to generate a uniform

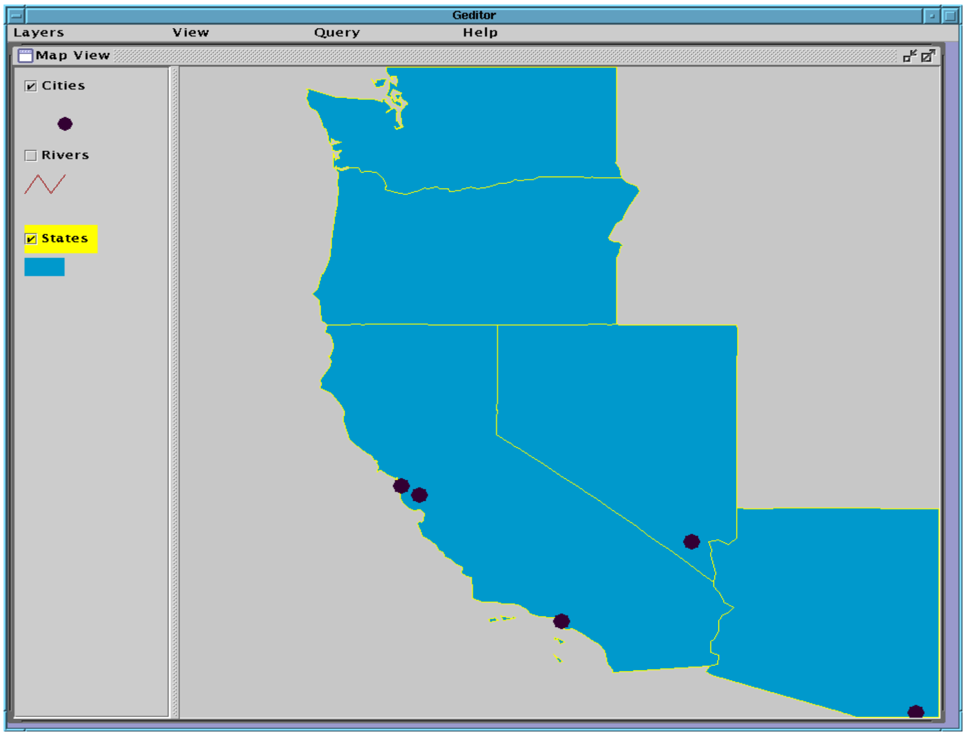


Figure 3.10: Updated Map View after Layer Control

Figure 3.11: View Menu

layer and the Thematic Mapping submenu to generate a non-uniform layer. This section explains the functions of Color Editor and Symbol Editor in the Uniform Layer submenu.(See Figure 3.12.)

Figure 3.12: Uniform Layer Submenu

Color Editor

Suppose there are two layers shown in Map View: Cities and States. No matter whether the States layer is a uniform layer or not, the user can make the States layer a red uniform layer using Color Editor. First, the user must activate the States layer using the Layer Control function as discussed in Section 3.2. Then click View menu from the menu bar and select the Color Editor from the Uniform Layer submenu. A dialog box pops up as Figure 3.13 shows.

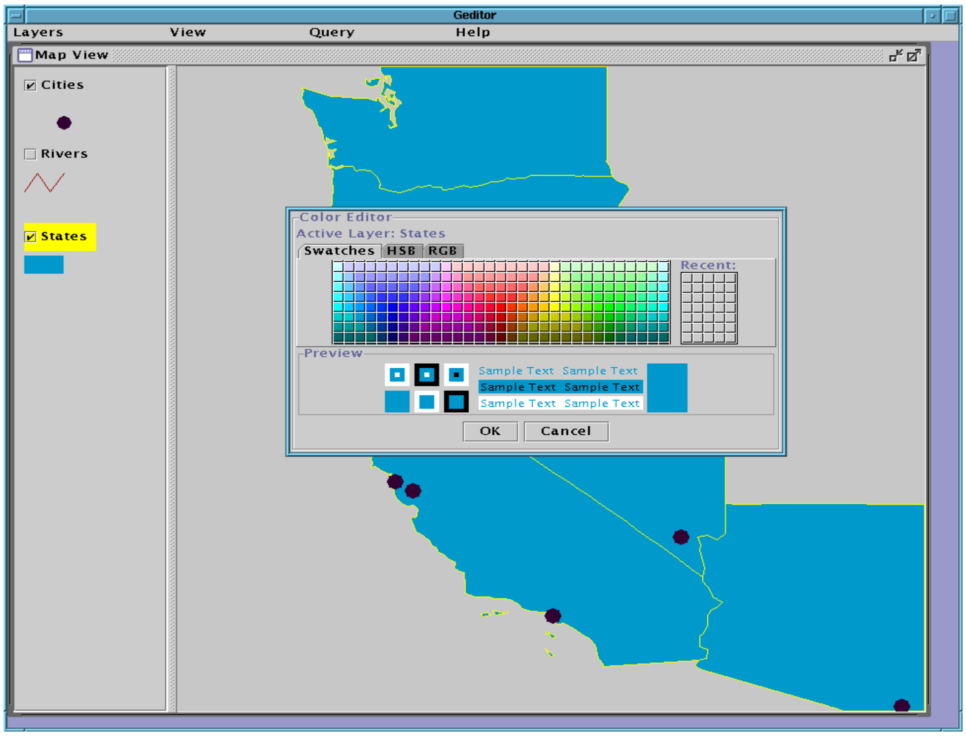


Figure 3.13: Color Editor Dialog

At the top of the dialog box, the name of the current active layer is displayed. In the middle is the color chooser. "Swatches" that allow the user to click color patches on a palette is the default way of choosing a new color. "HSB" and "RGB" are the other two alternative ways. Click the corresponding tabs when these color choosing methods are needed. In the Preview section, the original color and the new color selected by the user are displayed. After finishing the selection of new color, click OK to close the dialog.

If the dialog was closed using the OK button, Geditor updates Map View according to the new color. For example, if the user changes the color of the States layer from original blue to red, all the states polygons will be filled in red. The legend split window will also be updated accordingly.

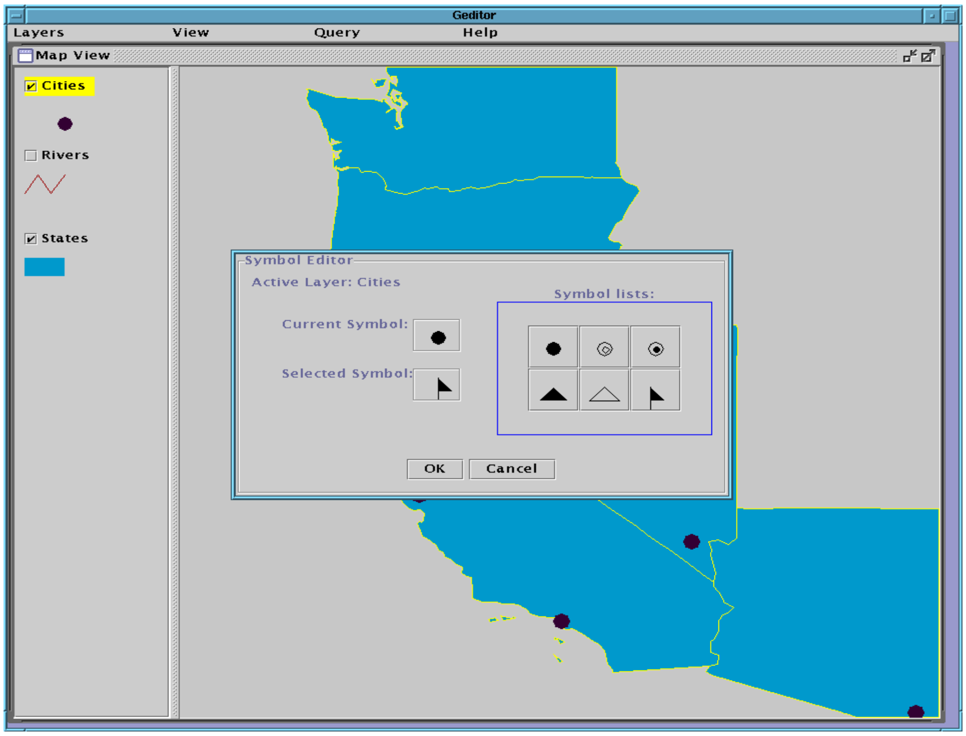


Figure 3.14: Symbol Editor

Symbol Editor

The user can also change the symbol of points using Symbol Editor. For example, suppose that the user would like to change the cities in current Map View from solid ovals to flags. First, activate the Cities layer using Layer Control. Then click the Symbol Editor item from the Uniform Layer submenu. A dialog titled as Symbol Editor appears as Figure 3.14 shows.

As usual, the name of the current active layer is displayed at the top of the dialog. The current symbol and the new selected symbol are displayed on the left part of the dialog. The right box contains the six symbols that could be chosen. We click the flag symbol and the OK button to close the dialog. After that, the Map View window will be updated as Figure 3.15 shows. Notice that the cities have been changed to flags in the figure.

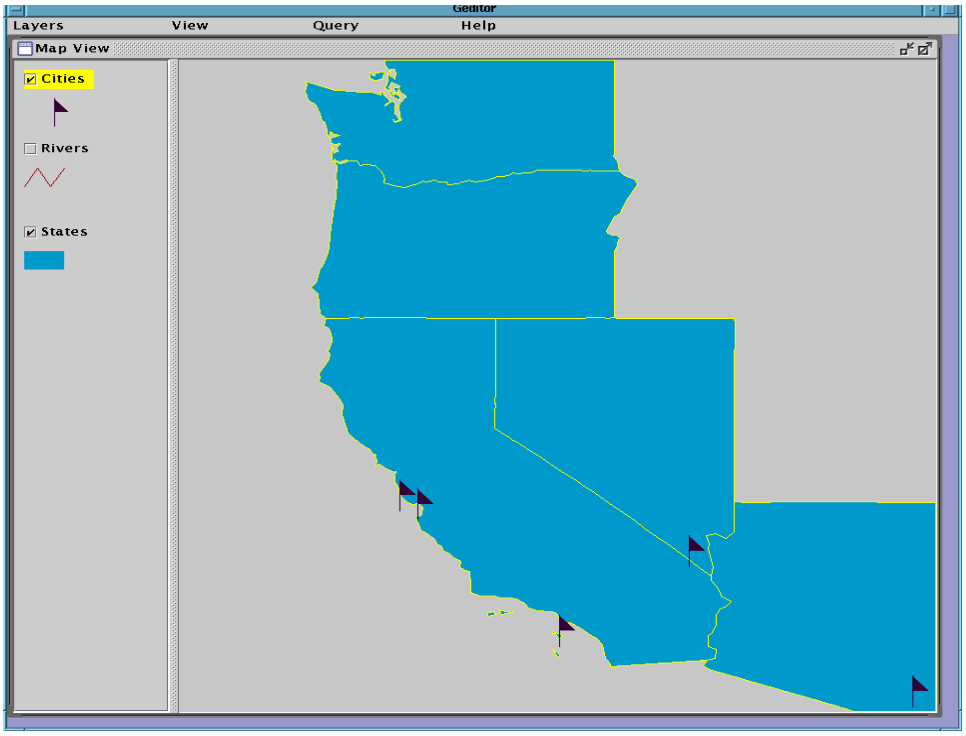


Figure 3.15: Updated Map View after Symbol Editor

3.3.2 Thematic Mapping

Non-uniform layers can be generated through Thematic Mapping. Two thematic maps are provided in current Geditor: Individual Value map and Range map. An Individual Value map maps individual values of a certain descriptive attribute to different colors. A Range map maps ranges of values to a color ramp. Figure 3.16 is the Thematic Mapping submenu.

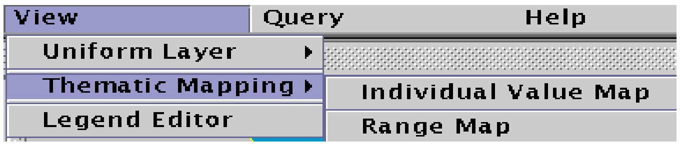


Figure 3.16: Thematic Mapping Submenu

Individual Value Map

Individual value maps are useful when the user needs to group map features by the individual values of a certain descriptive attribute. For example, in the States uniform layer displayed in Figure 3.15, all the state polygons have the same color. We cannot tell whether any of the states have the same value of temperature or not. However, we can show this using the Individual Value Map.

First, activate the States layer using Layer Control. Then click the View menu and then select the Individual Value Map from the Thematic Mapping submenu. A dialog box pops up as Figure 3.17 shows.

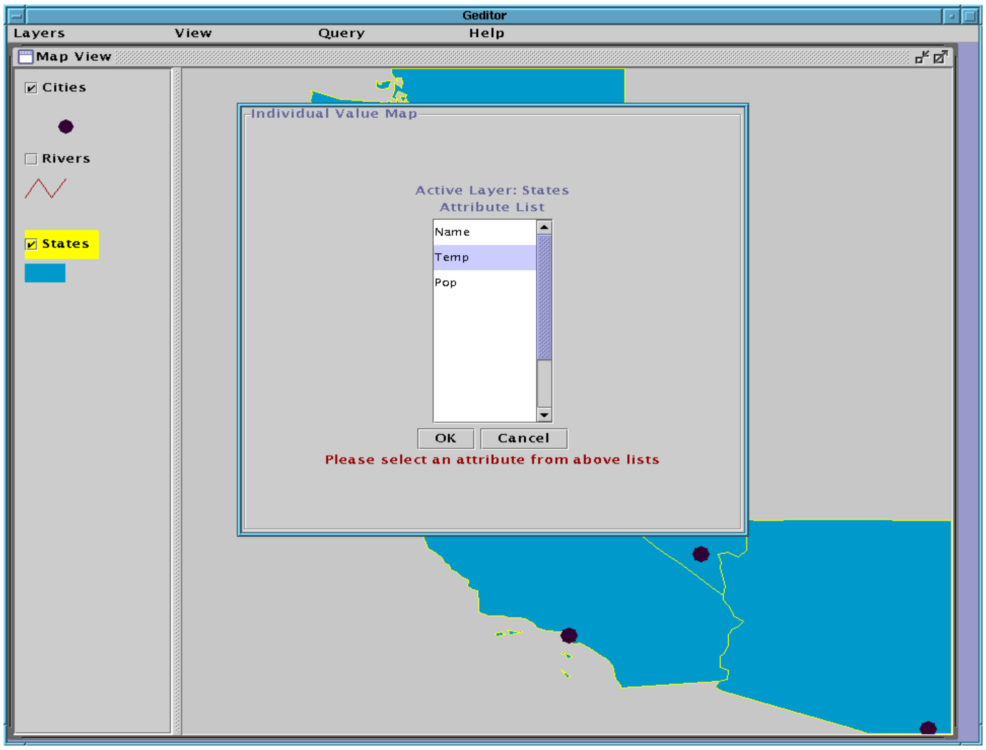


Figure 3.17: Individual Value Map Dialog

On the top line of the dialog, the current active layer name States is displayed. The middle list shows all the descriptive attributes in the map relation. We choose Temp in the list and click OK to close the dialog box. The Cancel button discards the user input and closes the dialog.

After the dialog box is closed with the OK button, the Geditor assigns different colors to the map

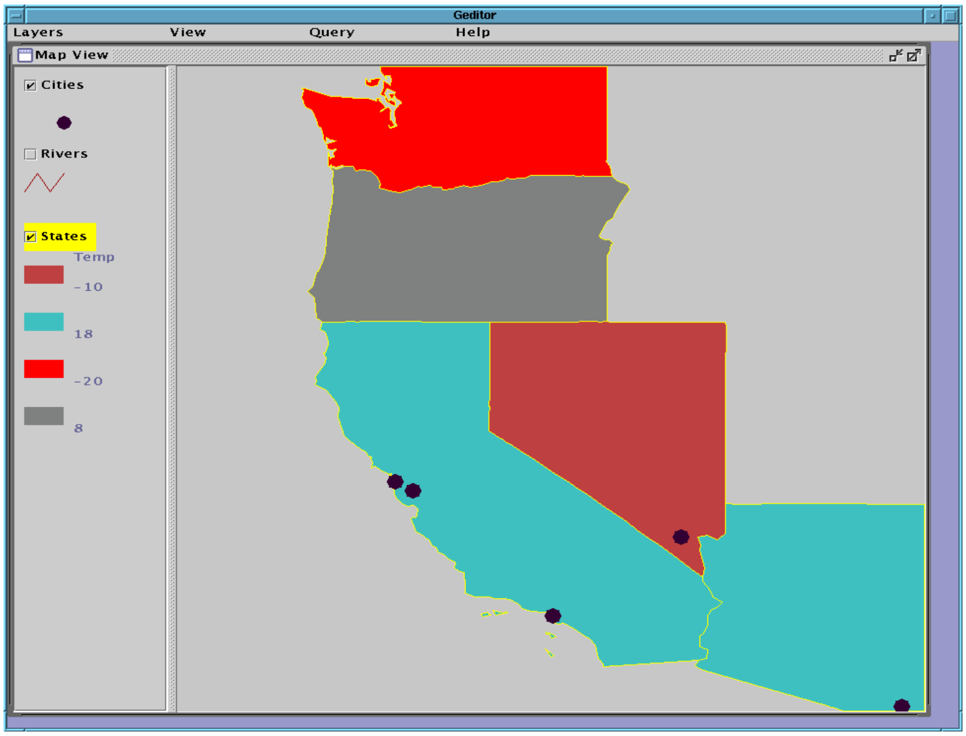


Figure 3.18: Updated Map View after Individual Value Map Dialog

features according to the different values of the Temp attribute. Then Geditor updates the Map View including the legend split window as Figure 3.18 shows. The user will notice that California(CA) and Arizona(AZ) appear in the same color which means they have the same temperature. Others are in different colors indicating that their temperatures are different. In the legend split window, a label showing the temperature value is displayed next to each graphical legend of the States layer. The labels can be changed by Legend Editor that will be discussed in Section 3.3.4.

Notice that the number of different colors is limited. When the number of different values of the attribute is over 256*256*256, Geditor will display an error message indicating that there are too many different values of this attribute in the current map relation.

3.3.3 Range Map

Although the largest number of different colors allowed in the individual value map is 256*256*256, when the number of colors is more than, say, 40, the map will become overwhelming. For example, if every state in the United States has a different temperature in January, there will be 51 different colors in an individual value map. In this case, users might need to reduce the number of colors by regrouping the map features using a range map.

Range maps assign different colors to different ranges of values of a descriptive attribute. For example, a range map showing the distribution of temperature for January in the United States fills the state polygons with white if the temperature is in range [-20.0, -11.0), light green if in range [-11.0, -2.0), green if in range [-2.0, 7.0), dark green if in range [7.0, 16.0), and deep green if in range [16.0, 25.0]. Notice that the descriptive attribute that the range map is based on must be of a quantitative type. Otherwise, the range is meaningless.

To generate the above range map, first activate the States layer using Layer Control. Then click the View menu and select Range Map from the Thematic Mapping submenu to open the Range Map dialog as Figure 3.19 shows. The first line of the dialog shows the current active layer name: States. The left list contains all the quantitative descriptive attributes of the map relation. Next to this attribute list is a list of pre-defined color ramps. They are red ramp, blue ramp, green ramp, and gray ramp. When the user selects a color map in the list, the colors are displayed as five patches in the right most list of the dialog. These colors can be edited by the user. When the user clicks on a color patch, a color chooser pops up for the user to choose a new color.

After the user selects an attribute in the attribute list, Geditor generates four numbers in the third list . The four numbers divide the range [min,max] into five equal ranges, where min and max are the minimal and maximal values of the selected attribute in the map relation. All the range numbers including min, max and the four auto-generated numbers are listed in the third column in a non-decreasing order from top to bottom. The color patches between two range numbers show that all the map features with the values between these two numbers will be displayed with this color. For example, in Figure 3.19, the color patch between the two numbers -2.0 and 7.0 is green. Therefore, all the state polygons with temperatures in [-2.0,7.0) will be filled in green. The min and max on the top and bottom are not editable, but the other four auto-generated range numbers can be edited by the user. Remember to keep all the range numbers in a non-decreasing order from top to bottom and in the range [min,max]. Type Return after editing. Geditor will perform the checking and if there is any error such as those with the order of the range numbers, the error message will be displayed on the message line. In an extreme case, when all the values of this selected attribute are the same in the whole map relation, all the range numbers are equal. Geditor assigns one color to all the map features in this layer.

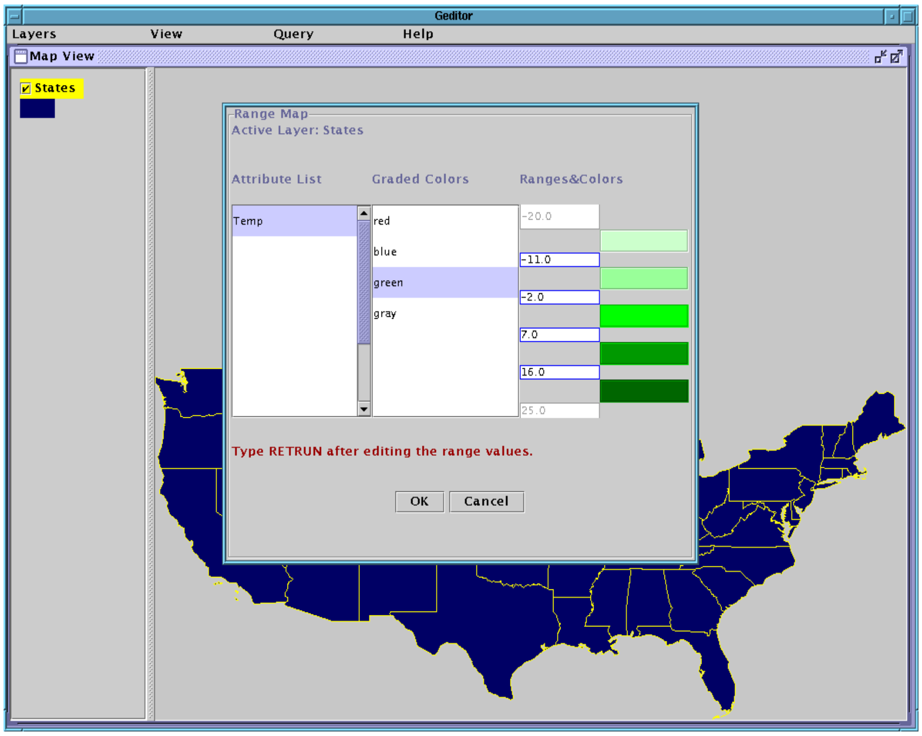


Figure 3.19: Range Map Dialog

To generate the range map of the January temperature distribution of the United States, we select Temp from the attribute list, then the green ramp from the color ramp list. Keep the ranges generated automatically by Geditor, and finally click on the OK button to close the dialog. Different from the examples in other sections, we choose the whole map instead of the West Coast of the United States in this section. This is because range maps are usually used when many map features with different attribute values exist in the map.

The Map View will be updated as Figure 3.20 shows. Notice that the legend window is also updated accordingly. The labels next to each legend color patch show the temperature range the color represents. The labels can be changed using Legend Editor that will be discussed in the next Section.

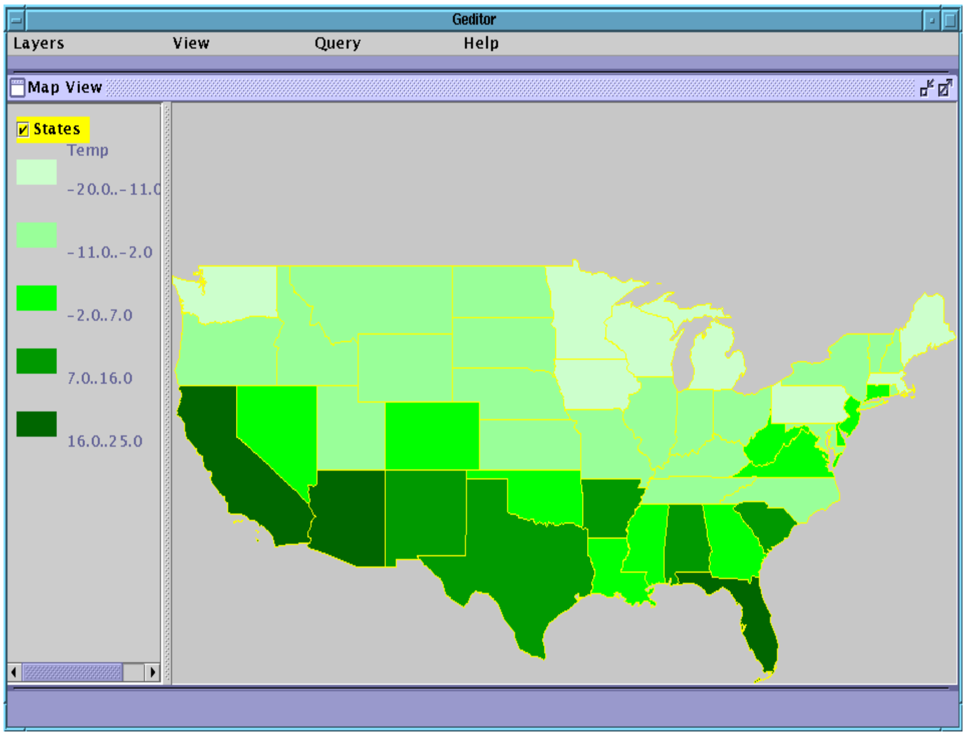


Figure 3.20: Updated Map View after Range Map Dialog

3.3.4 Legend Editor

In this section, we are going to discuss how to edit the legend window. In the legend window, each layer has a title, one or a series of graphical legends, and a label next to each graphical legend to show what this graphical legend represents. The graphical legend cannot be changed arbitrarily using Legend Editor because it shows the type, color and symbol of the map features of the corresponding layer in the current Map View window. However, the title and label can be edited by the user.

Similar to the other functions in the View menu, only the legend of the active layer can be changed. Therefore, first, the user needs to activate the layer whose legend needs to be edited. Then, click the View menu on the menu bar and select the Legend Editor item. A dialog box with the title Legend Editor pops up as Figure 3.21 shows. There are five columns in the dialog. The first column lists the name of the active layer in the map. The second column displays the title of this active layer that is currently displayed in the legend split window. The third and fourth

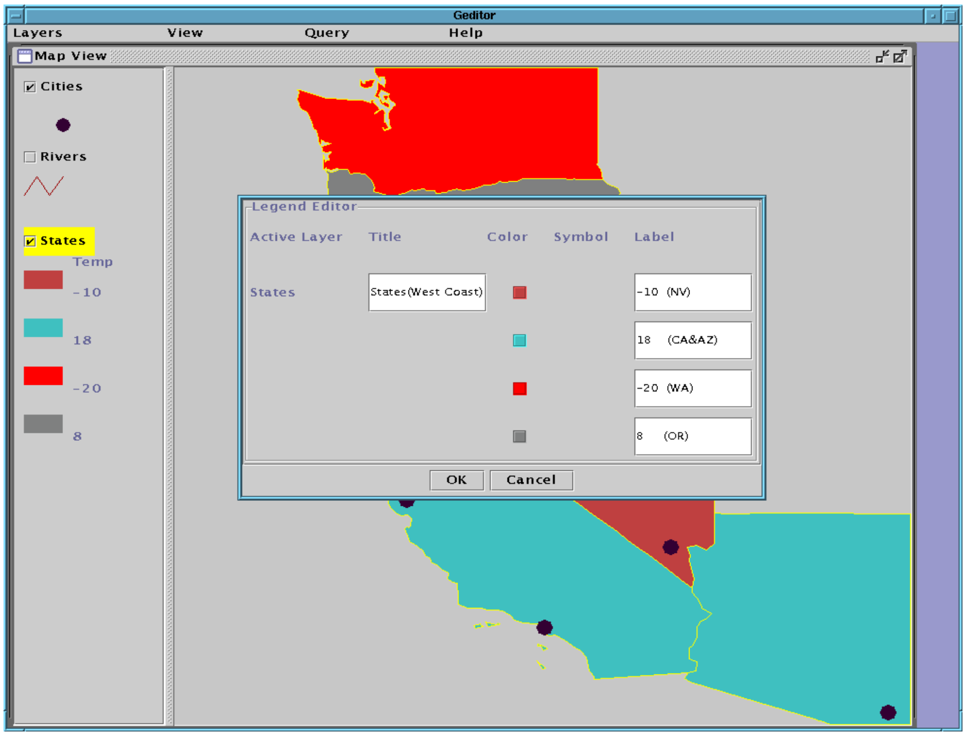


Figure 3.21: Legend Editor Dialog Box

columns display the graphical color and symbol of this layer. The fifth column shows the labels corresponding to the pair of color and symbol. If the layer is a uniform layer, only one pair of color and symbol will be displayed and the label of this layer is initially blank; if the layer is a non-uniform layer, more than one pair of color and symbol values are displayed for the layer and the default labels that have been attached by Geditor are also displayed in the fifth column.

Among the five columns, the Title and Label columns are editable. For example, the user can change the labels of the States layer as shown in Figure 3.21. The States abbreviations are added next to the temperature numbers. The title of States can also be changed to "States(West Coast)" as Figure 3.21 shows. Finally, click on the OK button to close the dialog. The legend window will be updated as Figure 3.22 shows. Notice the change of the title and labels of the States layer.

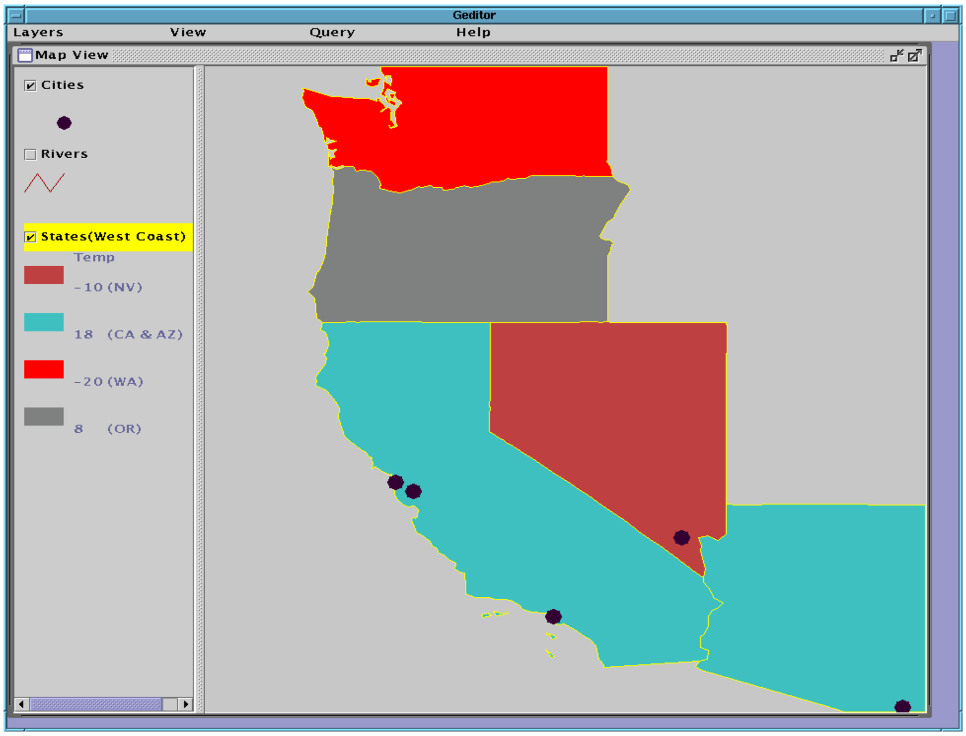


Figure 3.22: Updated Map View after Legend Editor

3.4 Query

The information displayed on the screen is always limited. As a GIS editor, Geditor provides the function to allow the user to query the map to obtain additional data that is not currently displayed on the map. For example, the user may need to know the population of San Francisco city. Using the Query function of Geditor, the user can obtain the population number by simply clicking on the city. Geditor provides three ways of querying the map: Identify Tool, Expression Builder, and Spatial Query. This section describes how to use Geditor to query the map using these three methods. Figure 3.23 shows the Query menu on the menu bar.

3.4.1 Identify Tool

Click the Query menu and then select the Identify Tool item. An internal frame titled as Identify Tool pops up. The name of the current active layer is displayed at the top of the frame. On the

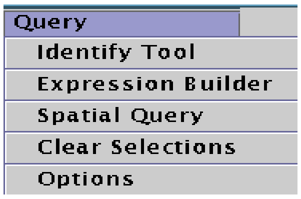


Figure 3.23: Query Menu

bottom of the frame, there is an Identify toggle button. Initially, this button is pressed, which means the Identify operation is currently active. At this time, if the user clicks the map feature in the current active layer, the data of all descriptive attributes will be displayed in the center part of the Identify Tool. To deactivate the Identify operation, the user just needs to release the Identify toggle button by clicking on it. Then no data will be displayed in the frame when the user clicks the map feature in the current active layer. The user can keep the Identify Tool on the desktop or close it by clicking on the Close button on the right top corner of the frame. See Figure 3.24. The data in the Identify Tool shows the results of clicking on San Francisco city in the Map View.

3.4.2 Expression Builder

The user can also obtain information by creating a query expression. For example, if users need to know which state has a population over 29760021 and the January temperature over 8 Celsius degrees, they can build an expression using Expression Builder.

First, activate the States layer using Layer Control. Then select the Expression Builder item from the Query menu to open the dialog as Figure 3.25 shows.

As usual, the first line indicates the current active layer: States. The upper left list contains all the descriptive attribute names. In the middle is a list of operators and brackets for the user to build the expression. The right list is the values of the selected attribute. The left lower part is a text area showing the expression being built according to the user input. The user can also type an expression from the keyboard in this area. There are three buttons in the right lower part: Clear, OK and Cancel. Clicking on the Clear button clears the text area. The OK button records the user input and closes the dialog, and the Cancel button discards the user input and closes the dialog.

To build the expression we mentioned at the beginning of this section, we first click on the Pop

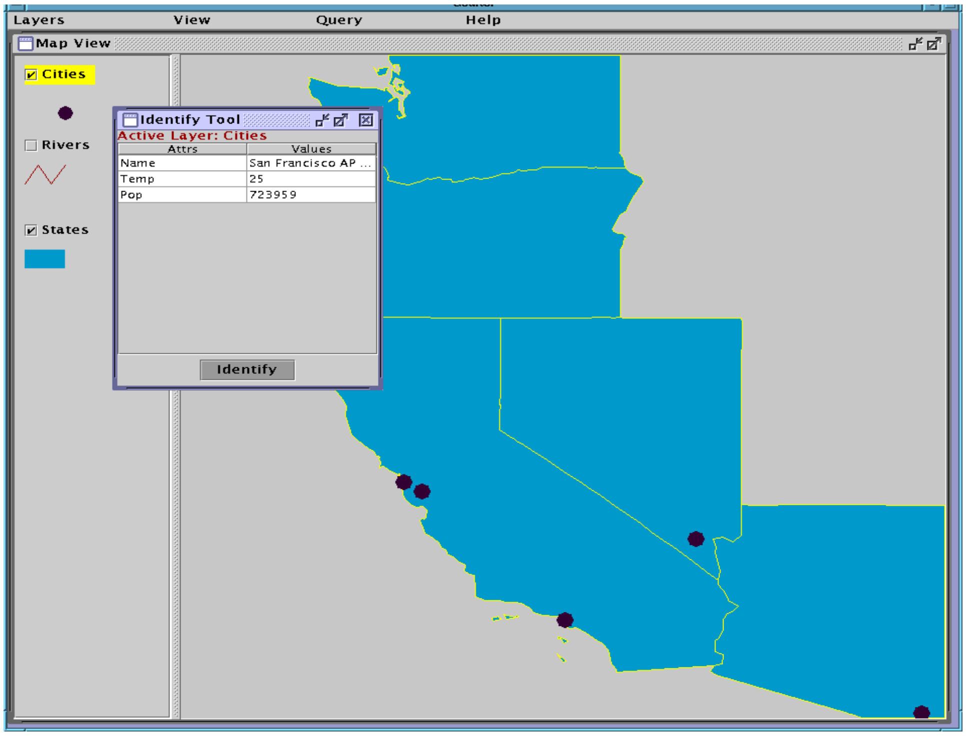


Figure 3.24: Identify Tool

attribute in the attribute list. Notice that the value list changes according to the selected item in the attribute list. Click on the “>=” operator and 2976021 in the values list. Click on the “and” operator in the middle list. After that, choose the Temp attribute, the “>” operator and 8 in the valuelist. The text area shows “Pop $>=$ 2976021 and Temp $>$ 8” which is exactly the expression we need to query the map. Finally, we close the dialog by clicking on the OK button.

After the dialog is closed, Geditor updates the Map View as Figure 3.26 shows. The California state is highlighted as yellow. This is the default color for selected map features by Expression Builder. The user can change it by using the Options menu that will be discussed in Section 3.4.5.

All the expressions built using Expression Builder must observe the following syntax:

```
Expression:=Conjunction(( ``or'' | ``|'' )Conjunction)*
Conjunction:=Comparison(( ``and'' | ``&'' )Comparision)*
Comparison:=Primary | Primary ComparativeOperator Primary
Primary:=Literal | Identifier | ``('' Expression ``)''
```

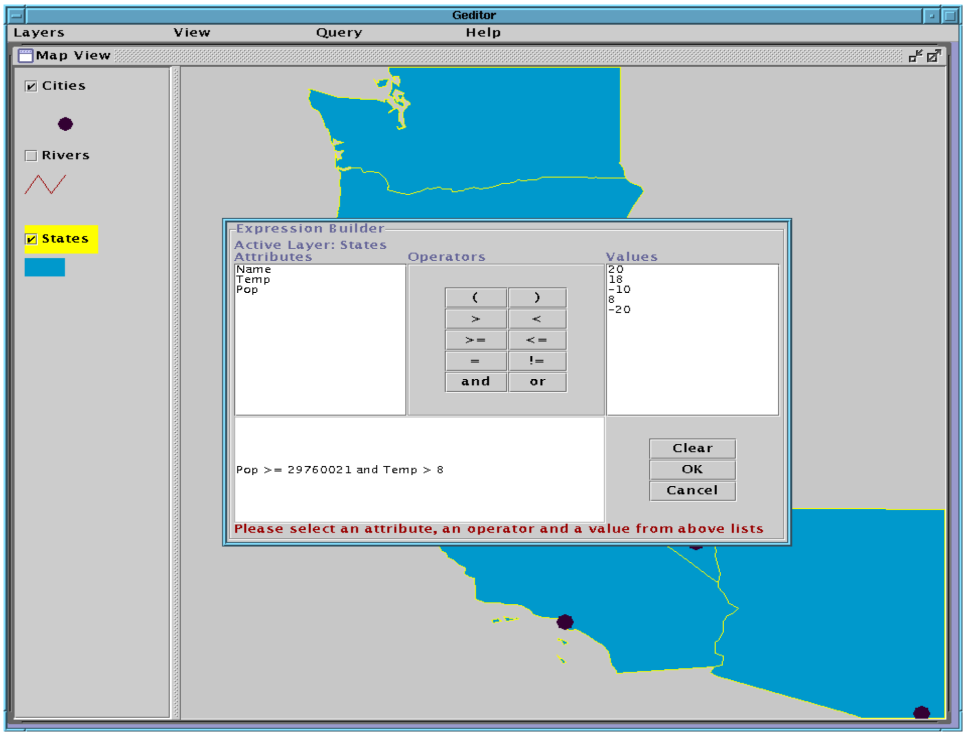


Figure 3.25: Expression Builder Dialog Box

`ComparativeOperator:=` ``=''|``!=''|``>''|``<''|``>=''|``<=''

Identifier specifies the attribute name and Literal specifies the value of the attributes. ( )* in the first and second formulae means repeating the component inside the brackets zero or more times.

The following are some examples of legal expressions:

Exmaple1: Pop>10000000

This is a simple `Comparison` composed of `Primary ComparativeOperator Primary`, where the first `Primary` is Pop — an `Identifier`, and the second `Primary` is 10000000 — a `Literal`. According to the second formula, a single `Comparison` is a `Conjunction`, and according to the first formula, a single `Conjunction` is an `Expression`. Therefore, it is a legal expression.

Example2: Pop>10000000 and Temp>8

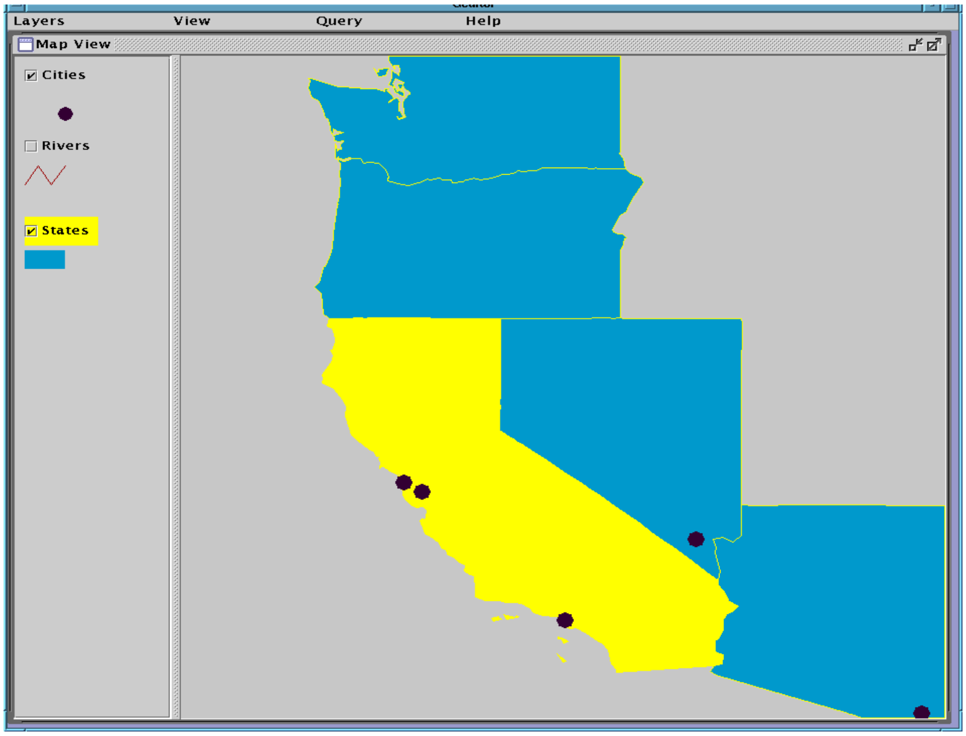


Figure 3.26: Updated Map View after Expression Builder

This is a `Conjunction` composed of two `Comparisons` combined by "and". According to the first formula, a `Conjunction` is an `Expression`.

Example3: Pop>10000000 and Temp>8 or Name="Arizona"

This is an expression composed of two `Conjunctions`:"Pop>10000000 and Temp>8", and "Name="Arizona"". The two `Conjunctions` are combined by "or". According to the first formula, two `Conjunctions` combined by "or" forms a legal `Expression`.

Example4: (Pop>10000000 and Temp>8 or Name="Arizona") and Temp<18

"Pop>10000000 and Temp>8 or Name="Arizona"" is an `Expression`. Therefore, "(Pop>10000000 and Temp>8 or Name="Arizona")" is a `Primary`. A `Primary` is also a `Comparison`. "Temp<18" is a `Comparison` too. According to the second formula, two `Comparisons` combined by "and" is a `Conjunction`. According to the first formula, a `Conjunction` is a legal `Expression`.

Notice that all the above expressions are also legal jRelix expressions in the **where** clause of selections. Actually, the Geditor expressions form a subset of jRelix expressions.

3.4.3 Spatial Queries

Spatial queries are unique in GIS editors. The user extracts data from the map relation by indicating the spatial relationships between map features. For example, the user may ask:" Show me all the cities that are within 50 miles of San Francisco". This is a typical spatial query. The given map feature is San Francisco, and the query is searching the map features in the Cities layer that satisfy the following spatial relationship with San Francisco: within 50 miles. Therefore, this query must be performed in the following steps: First, select the given map feature with the expression, "Name=San Francisco", using Expression Builder. As a result, San Francisco will become yellow as Figure 3.27 shows. Then activate the Cities layer that the query result belongs to. After that, click on the Spatial Query item in the Query menu to open the Spatial Query dialog as Figure 3.27 shows.

In the upper part of the dialog, Geditor shows the current active layer. In this example, it is the Cities layer. A list of spatial operators is displayed in the middle of the dialog. We choose the "Within Distance of" operator from the list. Next to the spatial operators list is a text field that allows the user to input a value. This area is only enabled when the selected operator is "Within Distance Of" since this is the only operator that needs a value. We type 50 in this value area. Finally, click on the OK button to close the window.

After the dialog is closed with the OK button, Geditor updates the Map View as Figure 3.28 shows. The user will notice that San Jose city is turned red in the Map View. Red color is the default color of the selected map features from Spatial Query. Users can change it using the Options menu.

In the implementation of Geditor, when the user selects a spatial operator, Geditor generates a jRelix event. An event handler for such an event should be already written and submitted by the Aldat programmer in the current jRelix system. If the corresponding event handler is not there, an error message is displayed. A detailed discussion about event handers used in Geditor will be presented in section 3.6.

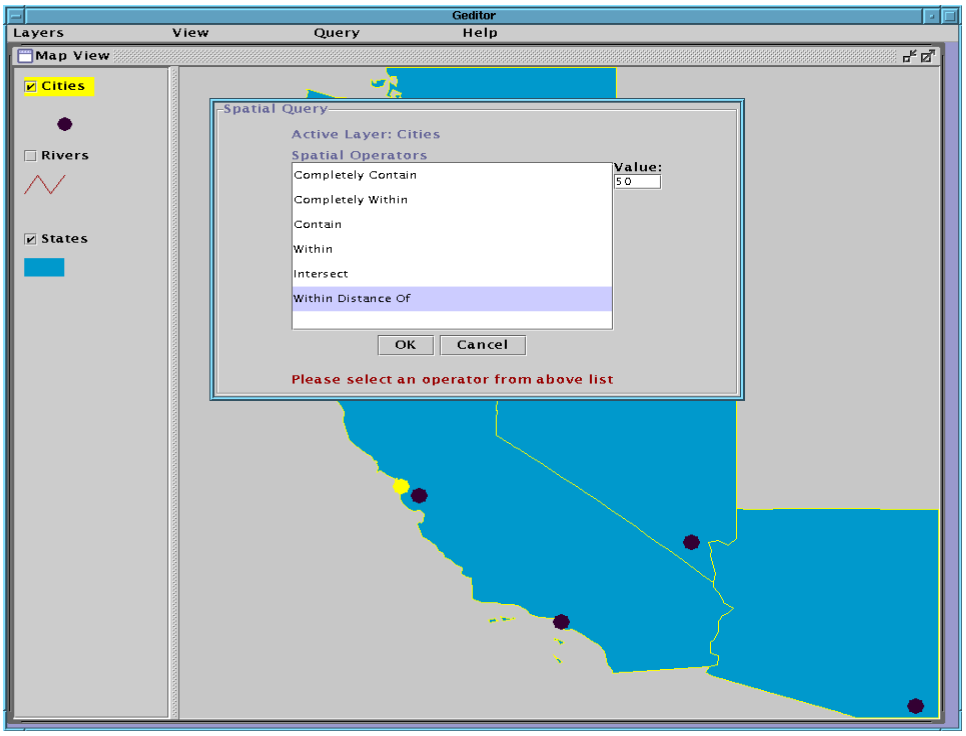


Figure 3.27: Spatial Query Dialog Box

3.4.4 Clear Selection

This function clears the user selections from Expression Builder and Spatial Query. It displays the map that was shown before the Expression Builder and Spatial Query operations were performed. To clear the current selections, select the Clear Selection item from the Query menu. The Map View will be updated and the highlighted map features will disappear.

3.4.5 Options

There are three default colors used in Geditor: the highlight color used in the legend window to show the current active layer, the color to show the selected map features from Expression Builder, and the color to show the selected map features from Spatial Query. All of them can be changed using the Options function in Geditor.

Select the Options item from the Query menu. A dialog box titled as "Options" pops up as Figure 3.29 shows. Three default colors are displayed as color patches in the dialog. Clicking on

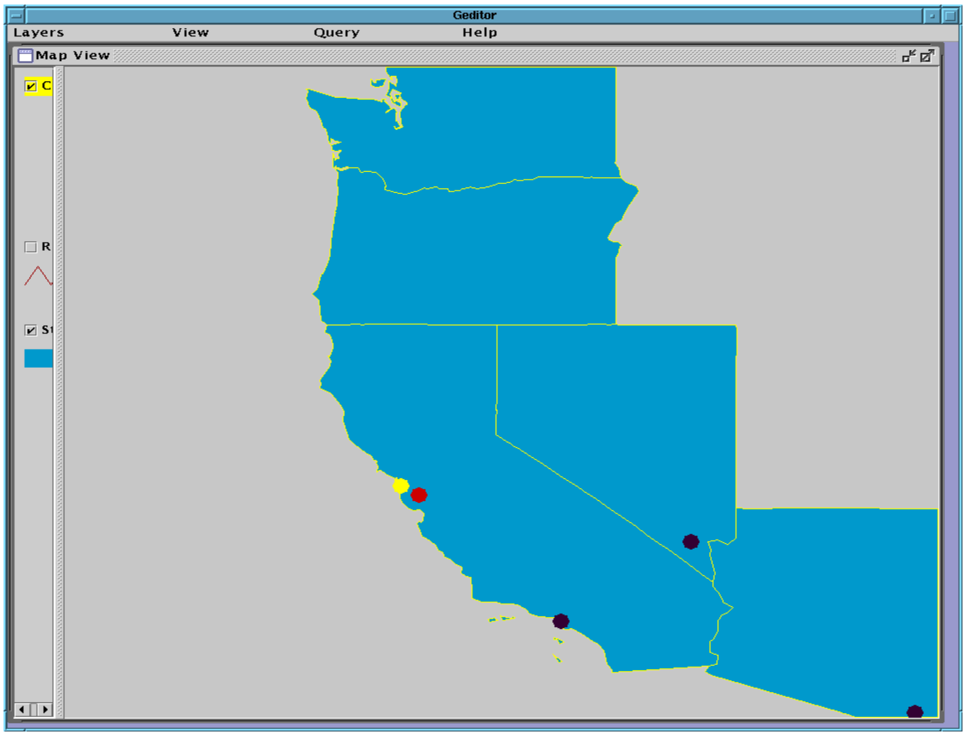


Figure 3.28: Updated Map View after Spatial Query Dialog

the color patches causes a color chooser to be displayed, which allows the user to choose a new color. After choosing a new color, click on the OK button to close the color chooser dialog. Click on the OK button again in the Options dialog. The default color(s) will be changed to the new color(s) the user selected. Clicking on the Cancel button discards user input.

For example, if the user changes the default color of the Expression Builder from yellow to cyan, all the map features that satisfy the user's expression will be turned to cyan the next time the user queries the map with Expression Builder.

3.5 Help

The Help menu in Geditor provides the current Geditor version information and a link to this User's manual. The Help contains two menu items as Figure 3.30 shows.

When User's Manual is selected, ghostview shows the text of this chapter in a separate window.

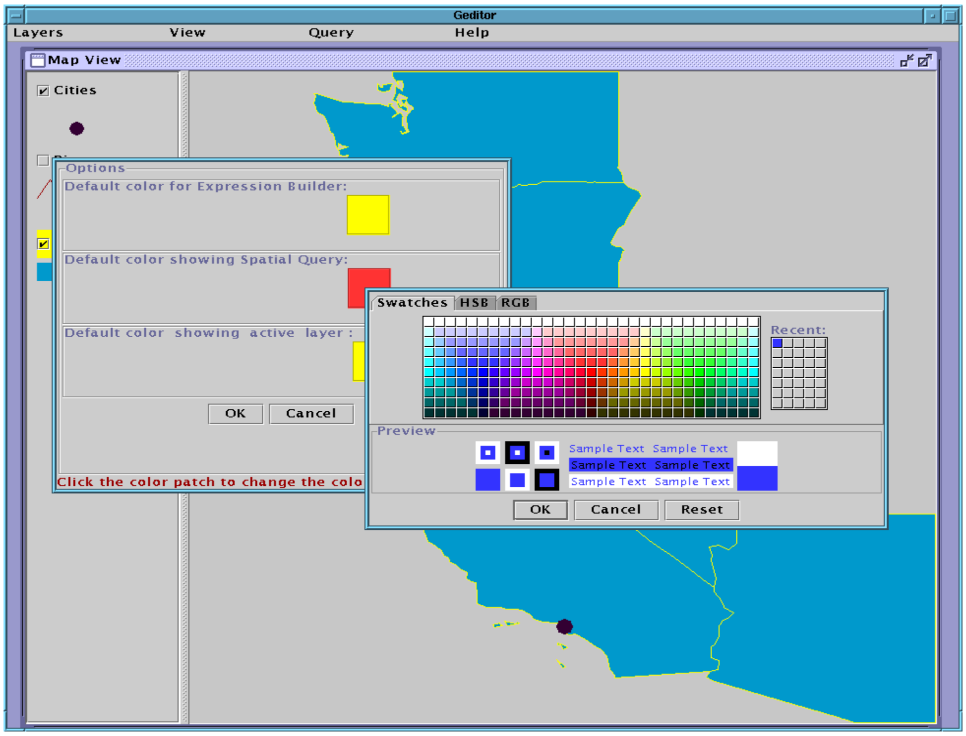


Figure 3.29: Options Dialog Box

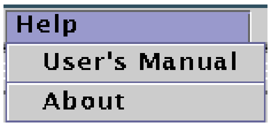


Figure 3.30: Options Dialog Box

If the About menu item is selected, a window is displayed that shows the version description of the current Geditor.

3.6 Event Handler for Geditor

As discussed in section 3.4.3, when a spatial query needs to be performed, an event is generated. For example, if the user selects the "Within Distance Of" operator in the Spatial Query dialog box as Figure 3.27 shows, Geditor generates a "withindist:" event. If the user selects "Contain" as the spatial operator, Geditor generates a "contains:" event. Corresponding to the six spatial operators built in Geditor, there are six events: "contains:", "cmpcontains:", "within:", "cmpwithin:", "intersect:", and "withindist:". Different from the Update events [Sun00], no prefix "post:" or "pre:" is needed in front of these event names because all the above Geditor event handlers should be invoked after the user's mouse click. Since the parameter list has not been built in the current version of jRelix computation package, Geditor cannot pass any relation as a parameter to the event handler. Therefore, Geditor makes an agreement with the Aldat Programmer on the four relation names: .ActLRel, .SelRel, .SpqRel, and .ValueRel. Notice that all of these relation names have a leading ".". This indicates that they are system relations instead of user-defined relations.

- **.ActLRel:** This relation contains all the map features in the current active layer.
- **.SelRel:** This relation contains the map features selected by the user using Expression Builder.
- **.ValueRel:** This relation contains the value needed by the "Within Distance Of" operator.
- **.SpqRel:** This relation contains the results of the event handler. It should contain all the map features that satisfy this spatial query.

Figure 3.31 shows a simple example of the event handler for the "contains:" event generated by the spatial operator: "Contain". It checks if polygons in the active layer (.ActLRel) **contain** points selected by the user (.SelRel). To handle the more complex events generated by the "Contain" spatial operator on all possible map objects including points-in-polygons, lines-in-polygons, and polygons-in-polygons, a more complicated polymorphism event handler should be written. This is out of the scope of this thesis.

```
<< Check if Points are in the Polygon
comp contains:() is
{ <<.SelRel: relation of the selected points
  <<.ActLRel: relation of the current active layer
  <<.SpqRel: relation of the polygons in the active layer that contains points in .SelRel
  <<the structures of the above three relations are the same:
  <<[G,T,S,X,Y,C,Symb,L,Name,Temp,Pop]
  let px be X;
  let py be Y;
  T2 ← [px,py] in .SelRel;
  T3 ← .ActLRel ijoin T2;  <<create the Cartesian product of .ActLRel and .SelRel

  let X' be par succ of X order S by G;
  let Y' be par succ of Y order S by G;
  let asquare be ((px-X)*(px-X)+(py-Y)*(py-Y));
  let bsquare be ((px-X')*(px-X')+(py-Y')*(py-Y'));

  <<calculate the sign through the determinant
  let det be px*Y*1+X*Y'*1+X'*py*1-px*1*Y'-X*py*1-X'*Y*1;
  let sig be if det< 0 then -1 else if det >0 then 1 else 0;
  let csquare be (X'-X)*(X'-X)+(Y'-Y)*(Y'-Y);
  let twoab be 2*sqrt(asquare)*sqrt(bsquare);
  let up be asquare+bsquare-csquare;
  let val be (real up)/twoab;
  let angle be acos(val)*sig;
  let sum be equiv + of angle by px,py, G;

  let D1 be 6.28319;  <<inside the polygon
  let D2 be 3.14159;   <<on the boundary of the polygon

  .SpqRel ← [G,T,S,X,Y,C,Symb,L,Name,Temp,Pop] where abs(sum-D1)<=0.03
     or abs(sum-D2)<= 0.03 in T3;
};
```

Figure 3.31: Event Handler for Spatial Operator: “Contain”

Chapter 4

Geditor Implementation

In this chapter, we are going to describe the implementation of Geditor in detail. Section 4.1 gives the overview of the Geditor implementation. Two interfaces of Geditor are explained in this section. In Section 4.2, the implementation of the interface for the jRelix Programmer-User is described. The new **gedit** syntax in jRelix will be discussed. Section 4.3 explains the implementation of the interface for the GIS End-User. This includes the Geditor GUI and a series of GIS functions.

4.1 Overview

The purpose of this implementation is to build a Graphical GIS editor (Geditor) into the current jRelix so as to allow the jRelix Programmer-User to invoke Geditor from jRelix and allow the GIS End-User to perform a series of GIS operations in the graphical interface.

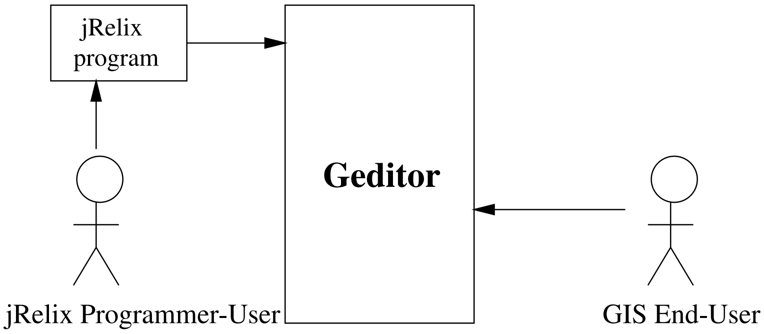


Figure 4.1: Two Interfaces of Geditor

Therefore, there are two interfaces of Geditor. One is the interface for the jRelix Programmer-User and the other is the Geditor graphical user interface (GUI) for the GIS End-User. The goal of the interface with jRelix Programmer-User is to build the new syntax (**gedit**) into jRelix to allow the programmer to call and display the Geditor GUI interface. The Geditor GUI provides a series of GIS operations for the End-User to view and edit the map, generate thematic maps, perform spatial queries, etc.

JDK1.2.2 is used in this implementation. This is because the JDK1.2.2 contains JFC/Swing components, which are used extensively in our GUI implementation. The Java Foundation Classes, or JFC, is a collection of Java APIs for developing graphical user interfaces [Gea99]. It contains the following APIs:

- Abstract Window Toolkit (versions 1.1 and beyond)
- 2D API
- Swing Components
- Accessibility API

The Abstract Window Toolkit, or AWT, is Java's original toolkit for developing user interfaces. The AWT provides the foundation upon which the rest of the JFC is built. The original AWT was designed to develop simple user interfaces. Swing, however, has more components expected in an object-oriented UI toolkit, is more platform independent, more stable and bug-free, and is capable of supporting the development of a high-powered user interface. Therefore, in this implementation, the Geditor GUI is built with Swing components. The 2D API offers two-dimensional rendering capabilities, such as providing a variable-sized pen for graphical operations. The Accessibility API consists of a set of classes enabling Swing components to interact with assistive technologies for users with disabilities. These two APIs are not used in our Geditor implementation.

4.2 Interface for the jRelix Programmer-User

To build the interface for the jRelix Programmer-User, we must first understand the original jRelix system architecture and add the Geditor to it. Section 4.2.1 explains the original system architecture and displays the updated system architecture with the new Geditor component. Section 4.2.2 discusses the building of the **gedit** syntax into the jRelix system. Section 4.2.3 explains the algorithm of executeRelixCommand() method which is added to the Interpreter to allow Geditor to call and execute jRelix commands or statements.

4.2.1 jRelix System Architecutre

The current jRelix system contains three main parts, the Parser, the Interpreter, and the Execution Engine. The Parser is created using JavaCC [San96], a Java Compiler Compiler that automatically generates parsers by compiling a high-level grammar stored in a text file. The JJTree [San96] preprocessor is used to build a syntax tree while parsing. The Interpreter (implemented in class Interpreter.java) repeatedly calls the Parser, gets the syntax tree generated by the Parser, traverses the syntax tree and decomposes it into a set of method calls executed by the Execution Engine.

In order to allow the jRelix programmer to call Geditor from jRelix, we must build the new **gedit** syntax into jRelix. The overall system architecture of jRelix is as Figure 4.2 shows.

In the above architecture, the Execution Engine is the same as that in the original version. The new **gedit** syntax needs to be added to the Parser and Interpreter. Notice that in Figure 4.2, there are two arrows of method call from Geditor to the Interpreter and Execution Engine. This is because Geditor needs to modify the map data (copy) during its running time. There are three ways to complete the modification: 1. Generate the method calls and pass them to the Execution Engine. Then the Execution Engine methods operate directly on the map data. 2. Call the Interpreter with executeRelixCommand() method and pass the jRelix commands or statements as arguments to it. Refer to Section 4.2.3 for the algorithm of executeRelixCommand(). 3. Generate events and pass them to the Interpreter that searches the system table for the predefined event handler to handle the events. Section 4.3.6(C) discusses the related algorithm in detail. These are represented by the arrows of method calls from Geditor to the Interpreter and Execution Engine in Figure 4.2.

Since the **gedit** is a functional operator, Geditor cannot change the original map data. Therefore, a copy of the map data is made and Geditor performs the data modification only on this copy. In our implementation, when Interpreter calls Geditor, it spawns a new thread for Geditor to run. As soon as Geditor is called, the statement which contains the **gedit** expression returns. As a unary

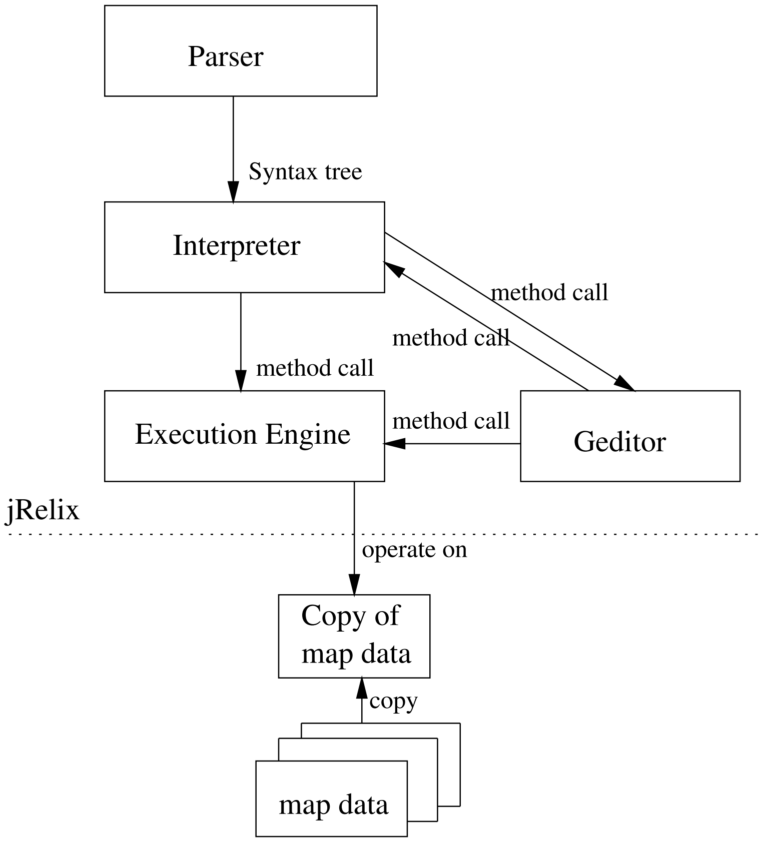


Figure 4.2: System Architecture

functional operator, the value of the **gedit** expression is the same as its operand: the original map relation. Since Geditor runs on an independent thread, the Interpreter can still accept the jRelix programmer's commands and statements while the GIS End-User is working interactively in the Geditor GUI.

4.2.2 Building the gedit Syntax

The syntax of **gedit** has already been specified in Section 3.1.3. Here, we summarize it as follows:

basic_graphical_attribute_list[*additional_graphical_attribute_list*] gedit rel_name

The first list specifies the five attributes representing group, type, sequence, x coordinate, and y coordinate respectively. The second list is optional. It specifies the attribute names of some predefined attributes. An arbitrary number of equations are allowed in the second list and we have implemented the following three: Color=*attr_name*, Symbol=*attr_name*, Layer=*attr_name*.

To implement the above syntax, we first modify the grammar specification text file by adding the new specification of the **gedit** syntax. Then we generate the new Parser using JavaCC and JJTree. In Interpreter.java, we add the EvaluateGedit() method to analyze the syntax tree and then call and display Geditor.

The algorithm of EvaluateGedit() is as follows:

EvaluateGedit()

- Analyze the expression tree to obtain the attribute lists and the map relation of **gedit**.
- Create the array of domains: doms[0..7], where doms[0] .. doms[4] record the following five domains of the map relation respectively: group, type, sequence, x coordinate, and y coordinate. These are the five attributes specified in the first list of **gedit**.
- doms[5].. doms[7] record color, symbol and layer domains of the map relation respectively. If the second list specifies all the three attributes, the corresponding domains of the three attributes will be stored in doms[5].. doms[7]. If any of them is missing, a default attribute with a default value will be first appended to the map relation and then the domain of the appended attribute will be stored in doms[5].. doms[7].
- Call Geditor(doms, map_relation) to create the Geditor object. The graphical Geditor window will be displayed.
- Return the relation object of map_relation as the return value.

In the above algorithm, when any one of the three attributes in the second list is missing, a default attribute with a default value will be appended to the map relation. For example, Figure 4.3 shows the data of a map relation called maprel.

The gedit expression is as follows:

R<−[G,T,S,X,Y][Symbol=Symb,Layer=L] gedit maprel;

Then, EvaluateGedit() appends a default color attribute to the maprel as Figure 4.4 shows.

G	T	S	X	Y	Symb	L	Name
S1	Polygon	1	12286	3369	203	States	California
S1	Polygon	2	12278	3345	203	States	California
S1	Polygon	1	13588	3399	203	States	Washington
S1	Polygon	2	14000	3379	203	States	Washington

Figure 4.3: Example of Map Relation:maprel

G	T	S	X	Y	Symb	L	Name
S1	Polygon	1	12286	3369	203	States	California
S1	Polygon	2	12278	3345	203	States	California
S2	Polygon	1	13588	3399	203	States	Washington
S2	Polygon	2	14000	3379	203	States	Washington

maprel

ijoin

.C
204055055

COLOR

G	T	S	X	Y	Symb	L	Name	.C
S1	Polygon	1	12286	3369	203	States	California	204055055
S1	Polygon	2	12278	3345	203	States	California	204055055
S2	Polygon	1	13588	3399	203	States	Washington	204055055
S2	Polygon	2	14000	3379	203	States	Washington	204055055

updated maprel

Figure 4.4: Append Color Attribute to maprel

In Figure 4.4, COLOR is the default relation name used in appending the color attribute to the map relation. .C is the default attribute name. The default color value is 204055055 (red). After the appending, the domain of .C will be recorded in the array doms[0..7] and be passed to Geditor object.

Similarly, when the attribute of symbol is missing in the second list, a default attribute will be appended to the map relation as well. SYMBOL is the default relation name used in appending and .Symb is the default attribute name. The default symbol value is 201 (solid oval). The default relation for appending the layer attribute is LAYER and the default attribute name is .L. The default layer value is "defaultlayer".

Since the three attributes we implemented in the second list (Color, Symbol, and Layer) are required for the map display, when any one of them is missing, a default value must be appended. However, in future work, when more attributes are implemented in the second list, if they are not related to the map display, it will not be necessary to append the default attribute values.

4.2.3 executeRelixCommand() algorithm

In this section, the algorithm of executeRelixCommand() is explained. This is a method written in Interpreter.java. With this method, other classes can execute jRelix commands or statements by calling the executeRelixCommand() and passing these commands or statements as arguments to this method.

executeRelixCommand(jRelix_command_or_statement)

- Call parser.Start() to parse the jRelix_command_or_statement and build the syntax tree.
- Call interpret() to analyze the syntax tree and decompose it into method calls and pass them to the Execution Engine.

4.3 Interface for the GIS End-User

Building the interface for the GIS End-User includes the displaying of the GUI with menus of GIS functions, performing the GIS operations according to the user-input in the GUI, and displaying the results of these operations as graphical maps on screen. Section 4.3.1 presents the architecture of Geditor. Section 4.3.2 explains the algorithm of Geditor.java that behaves as the controller of Geditor and lays out the Geditor GUI. Section 4.3.3 describes the algorithm of the map display function of Geditor. Section 4.3.4 explains the classes used to implement the GIS functions in the Layers menu. Section 4.3.5 describes the classes to implement functions in View menu. The classes to build the functions in Query menu are discussed in Section 4.3.6.

4.3.1 Geditor architecture

The main class of Geditor is implemented in Geditor.java. The architecture is shown as Figure 4.5.

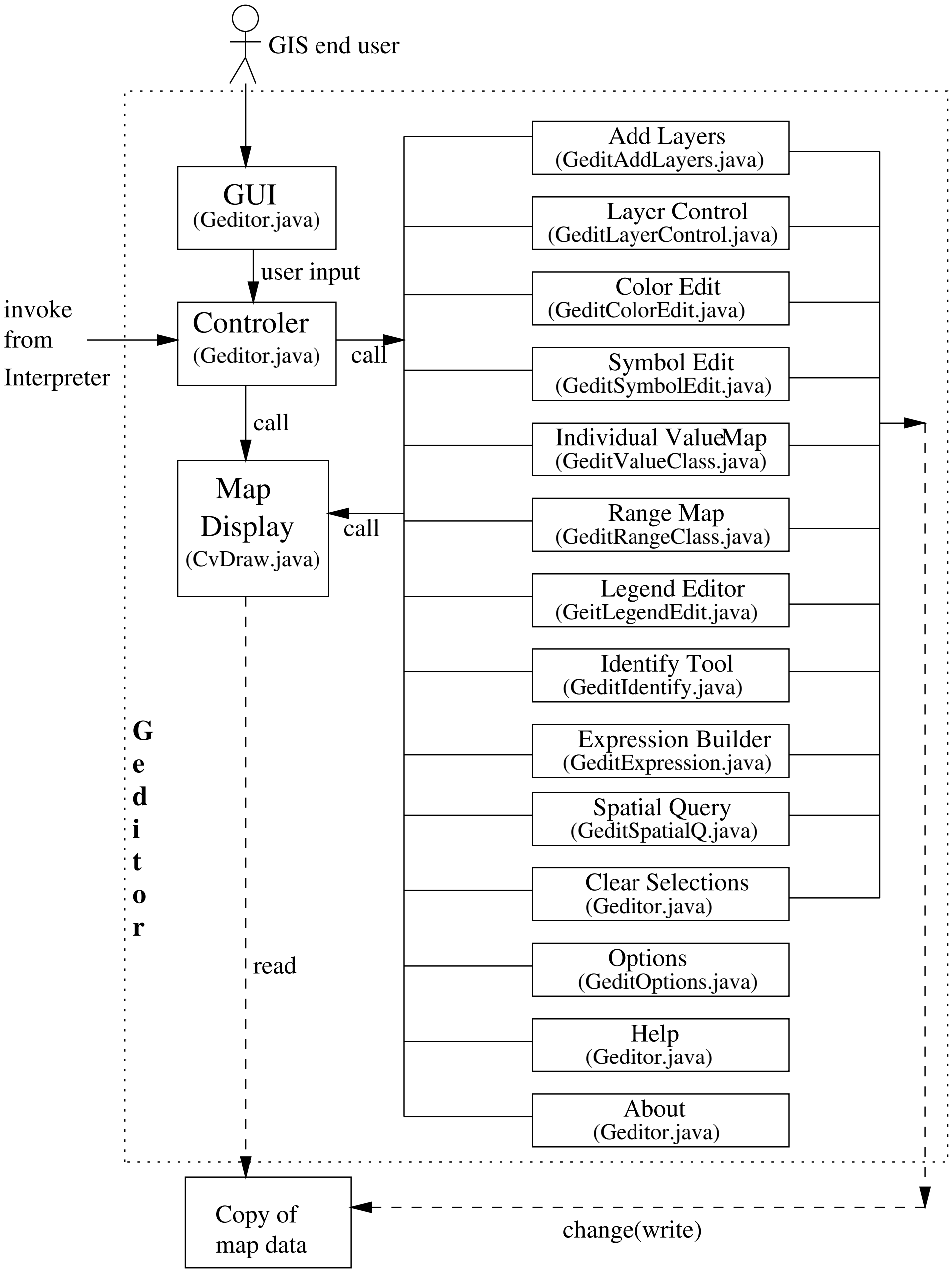


Figure 4.5: Geditor Architecture

As Figure 4.5 shows, Geditor is invoked from Interpreter (Interpreter.java) with the parameters of the map relation and the array of domains:doms[0..7]. After Geditor is called, it displays a GUI with menus containing a set of GIS operations. Corresponding to the GIS End-User's input in the GUI, Geditor calls the related class to perform the GIS operations. These operations include Add Layers, Layer Control, Color Edit, ..., etc. These GIS functions modify the copy of the map data (in relation format) when necessary and call the map display class to show the most recently updated Map View on screen. The map display class is implemented in CvDraw.java. When it is called, it reads the most recently updated data from the copy of the map relation and displays it on screen. Notice in Figure 4.5, the read and write opeations of the map data are displayed using dashed lines. This means the map data is not directly accessed by Geditor. Instead, as discussed in Section 4.2.1, Geditor calls the Execution Engine to read and write the map data copy.

As indicated in Figure 4.5, Geditor is implemented using more than a dozen classes. Table 4.1 summarizes the main classes of Geditor and their functions.

Class Name	Function Description
Geditor.java	Geditor Controller. It lays out the Geditor GUI, waits for the user input and calls the corresponding classes to perform the user required tasks.
CvDraw.java	Draws the map in the Map View window.
GeditAddLayers.java	Adds layers to the Map View window.
GeditLayerControl.java	Changes the state of each layer shown in the Map View window.
GeditColorEdit.java	Changes the color of the active layer to form a uniform layer.
GeditSymbolEdit.java	If the active layer has the Point type, changes the symbols of the map features in this layer to form a uniform layer.
GeditValueClass.java	Creates an individual value thematic map based on a descriptive attribute.
GeditRangeClass.java	Creates a range thematic map based on a descriptive attribute.
GeditLegendEdit.java	Edits the title and label of the active layer in the legend window.
GeditIdentify.java	Lists the values of all the descriptive attributes of a map feature close to the mouse in a pop-up window.
GeditExpression.java	Allows the user to build a Boolean expression and highlights the map feature that satisfies the expression.
GeditSpatialQ.java	Highlights the map feature having particular spatial relationship with a given map feature.
GeditOption.java	Allows the user to change the default colors used in Geditor.

Table 4.1: Geditor Classes

With the exception of Geditor.java and CvDraw.java, all the other classes in Table 4.1 build the GIS functions corresponding to the menu items in Geditor GUI. These classes are implemented in a similar way. First, they lay out components such as text fields, lists, color choosers, and radio buttons inside the dialog box and display the dialog. The purpose of the dialog is to allow the user to input the values required as parameters for completing the GIS functions. The classes wait for the user's input until the user closes the dialog with the OK or Cancel button. If the user closes the dialog with Cancel, these classes return to their parent: Geditor.java. If the user closes the dialog with OK, these classes use methods such as getText(), getElementAt(), getSelectedValue(), and getColor() to obtain the user input in the dialog. After updating the map data copy according to the user input, these classes update the map in the Map View window.

There are other classes that are used to facilitate the implementation of the classes listed in Table 4.1. GeditSpatial.java provides methods which determine the spatial relationship of the map features used in GeditSpatialQ.java. GeditSymbolIcon.java draws the symbol icons used in GeditSymbolEdit.java . GeditLegend.java defines the GeditLegend class used by Geditor.java to build the legend array.

In the rest of this chapter, the implementation of the Geditor classes listed in Table 4.1 will be discussed in detail.

4.3.2 Geditor Controller

The controller of Geditor is implemented in Geditor.java. The main task of this class is to record the system state with a set of variables, lay out and display the Geditor GUI, wait for the user input, and call the corresponding functions to perform the required GIS operations. As described in Chapter 3, the main part of Geditor GUI contains a Geditor window with a menu bar on the top. From the left to the right, the menu bar contains the Layers, View, Query and Help menus. In the middle of the Geditor window is the Map View internal frame that displays the map. The left split pane of the Map View window shows the legends of the map.

The following are some important variables that record the Geditor system state:

- showlayers[] array: records the layers that are shown in the Map View window.
- legend array: records the necessary data of each layer for displaying both the text and graphical elements in the legend window. The data of each array element include the *name, title, color value, symbol value, label, labelattr, visible, active,* and *type* of each layer.

The *labelattr* is used in thematic mapping. It uses the same attribute name as that used by the map feature classification. It is used in the legend window to show the attribute name where the labels of different graphical legends come from. When the thematic maps are generated, the labels of different graphical legends are generated using this attribute value.

Show/Hide are two values the *visible* variable can have. Active/Inactive are two possible values of the *active* variable. The three possible values of *type* are Polygon, Point, or Polyline.

The algorithm of Geditor.java is as follows:

1. Obtain a copy of the map relation to tmprel.
2. tprel$<-$[layer, type, color, symbol] in tmprel.
3. Use tprel to create the legend array.
4. Lay out the components inside the Geditor window. Display the Geditor window with the menu bar and an empty Map View window. Wait for the user input in the Geditor window.
5. If the user selects the Add Layer or Layer Control menu item in the Layers menu, call Add layers or Layer Control functions by invoking GeditAddLayers or GeditLayerControl classes accordingly. After that, call the updateMapView(maprelation_name) to update the map in Map View window. The updateMapView method selects a subset of the map relation according to the showlayers[] array and calls the Map Display function to display the map with shown layers. Then, call updateLegend() to update the legend window according to the legend array which has been updated using the Add Layer function.
6. If the user selects a menu item in the View menu, call the related function by creating the corresponding object. For example, if the user the chooses the Color Editor menu item in the View menu, call the Color Editor function by creating the GeditColorEdit object. After that, call the updateMapView(maprelation_name) to update the map in the Map View window. The map relation should be the updated one with the changed color. Then, also call updateLegend() to update the legends.

7. If the user selects a menu item (except the Clear Selection) in the Query menu, call the related function and update the map and legends in the Map View window. Before the query is performed, store the current map relation to "stored_maprelation". When the user chooses the Clear Selection function, call the updateMapView(stored_maprelation_name) to restore the original Map View before the query is performed.

8. If the user selects User's Manual in the Help menu, use "gv helpfilename" to call the ghostview to show the text of the User's Manual chapter of this thesis. If the user selects the About item in the Help menu, display a window with the version message of the current Geditor.

4.3.3 Map Display

The map display function is implemented in CvDraw.java. This function is called to display the result of the set of GIS functions that are going to be discussed in the following sections. For example, at the end of the Add Layer function, this map display function is called to display the map with the layers selected by the user in the Add Layer dialog. In another case, when a new color is assigned to a map feature using the Color Editor function, the map display function is called to display the map with the updated color. To make it possible for this map display function be reused by all GIS functions, we design the following algorithm.

1. Get the positions of the attributes: group, sequence, x coordinate, y coordinate, color, symbol, and layer in the map relation.

2. According to the maximum and minimum x, y coordinates in the map relation, calculate the factor for mapping them to the screen pixels.

3. Read the first tuple in the map relation (sorted relation).

4. If the type value of the tuple is point, draw a symbol in the Map View window. The shape of the symbol is determined by the symbol value of the tuple, the position is determined by the x, y coordinates, and the pen color is determined by the color value of the tuple.

5. If the type of the tuple is polygon, read the next tuple (sorted relation) until it is not in the same group as the former one. Each tuple in the same group is one of the vertices of the polygon.

Store all the vertices in an array. Then draw and fill a polygon in the Map View window. The x, y coordinates of each vertex determines the shape and position of the polygon. The filling color is determined by the color value.

6. If the type of the tuple is polyline, read the next tuple (sorted relation) until the next tuple is not in the same group as the former one. Each tuple is one of the vertices of the polyline. Draw a line from the former vertex to the next. The shape and position of the polyline is determined by the x, y coordinates of each vertex. The color value determines the pen color.

7. Repeat step 4, 5, 6 until the end of the map relation is reached.

The above algorithm is based on the following assumptions:

- The map relation that is passed to the CvDraw class must contain at least the following eight attributes: group, type, sequence, x coordinate, y coordinate, color, symbol, and layer. The relation must be sorted by group, type, sequence, x coordinate, y coordinate as well. The tuples in the same polygon group must represent the vertices of the polygon in a counter-clockwise sequence. Otherwise, the map would be displayed in an unexpected shape.

- The color values of the tuples in the same polygon group are the same. Therefore, CvDraw picks up a random one as the filling color. In this implementation, CvDraw picks up the color value of the last tuple in the same group. The color values of the tuples in the same group of polyline could be different. When CvDraw draws a line from the former vertex to the next, the color value of the start vertex is used as the pen color.

In the above algorithm, when the polygons and polylines are drawn, the symbol values are ignored. Strictly speaking, the symbol values of polygons and polylines could have meanings. The symbol values of polygons could represent the filling patterns and the symbol values of polylines could represent the line thickness or line styles, such as the dashed line and dotted line. Since we use JDK 1.2.2 to implement Geditor and it only contains the AWT and Swing components of JFC (refer to Section 4.1 for more details), it is very complicated to implement the filling patterns of polygons as well as the line thickness and styles of polylines. For further implementation, the JFC 2D package could be used for the map display. Please refer to Chapter 5 for further information.

4.3.4 Layers

There are two functions in the Layers menu: Add Layer and Layer Control. When the Geditor is called, Geditor displays the Geditor window with a menu bar on the top and a Map View window in the middle. Initially, the Map View window is empty. The user needs to add layer(s) to let the Geditor know what layer(s) should be displayed. After a layer is added to Map View window, two switches are attached to this layer to record its state: Show/Hide, Active/Inactive. The Active/Inactive state can be changed using Layer Control function.

A. Add Layer

The following is the algorithm of Add Layers function.

1. Lay out the components inside the Add Layer dialog and display it. Wait for the user input until the user closes the dialog with OK or Cancel button.

2. If the user closes the dialog with the OK button, store the layer names that the user selects in the Geditor.showlayers[] array.

3. Update the legend array. Change the visible variable of each layer added by the user to "Show". The last layer added by the user in the Add Layer dialog is the current active layer. Therefore, change the active variable of this layer to "Active".

When the Add Layer function returns to the Geditor controller (Geditor.java), according to the updated showlayers[] array and the legend array, the controller calls updateMapView and updateLegend methods to update the map and legends in the Map View window.

B. Layer Control

Layer Control changes the Active/Inactive state of each shown layer in the Map View window. The algorithm is as follows:

1. Lay out the components inside the Layer Control dialog and display it. Wait for the user input until the user closes the dialog with OK or Cancel button. The two switches of all the layers are displayed in the dialog.

2. If the user closes the dialog with the OK button, change the *active* variable of the Geditor legend array of the corresponding layer to "Active" according to the user input in the dialog.

There is no need for the Geditor controller to update the map or legend in the Map View window when the Layer Control function returns.

4.3.5 View

There are three main functions in this menu: color and symbol editing in a uniform layer, thematic mapping, and legend editor.

1. Uniform Layer

In a uniform layer (refer to Section 3.3.1 for the definition), the color and symbol of all the map features in this layer can be changed by Color Editor and Symbol Editor.

A. Color Editor

The Color Editor changes the color of the map features in the current active layer. The algorithm is as follows:

1. Lay out the components inside the Color Editor dialog and display it. The current active layer name and a color palette are displayed. The active layer name is obtained from the Geditor legend array. Wait for the user input until the user closes the dialog with the OK or Cancel button.

2. If the user closes the dialog with OK, change the color value of the tuples of the current active layer in the map relation to what the user chose in the dialog.

3. Update the color variable of the current active layer in the legend array with the new color value the user input in the dialog.

When the Color Editor function returns to the Geditor Controller, it updates the map and legend in the Map View window according to the updated map relation and legend array.

B. Symbol Editor

The Symbol Editor function is enabled only when the current active layer has the "Point" type. It changes the symbol value of the points in the current active layer. The algorithm is as follows:

1. Lay out the components inside the Symbol Editor dialog and display it. The current active layer name and a list of predefined graphical symbols are displayed. Wait for the user input until the user closes the dialog with the OK or Cancel button.

2. If the user closes the dialog with OK, change the symbol value of the tuples of the current active layer in the map relation to what the user chose in the dialog.

3. Update the symbol variable of the current active layer in the legend array with the new symbol value the user input in the dialog.

When the Symbol Editor function returns to the Geditor Controller, it updates the map and legends in the Map View window according to the updated map relation and legend array.

2. Thematic Mapping

Two types of thematic maps are provided in this Geditor: Individual Value Map and Range Map.

A. Individual Value Map

Individual value maps group map features by the individual values of a certain descriptive attribute (refer to Section 3.1.2 for the definition). In this Geditor implementation, the grouping is shown using different colors. All the map features in the same group are displayed in the same color. The map features in different groups are displayed in different color. The algorithm goes as follows:

1. Lay out the components inside the Individual Value Map dialog and display it. The current active layer name and a list of all the descriptive attributes are displayed in the dialog. The current active layer name can be obtained by reading the Geditor legend array. Wait for the user input in the dialog and wait until the user closes the dialog with the OK or Cancel button.

2. If the user closes the dialog with OK, update the color value of the active layer in the map relation as follows:

   - Get different values of the selected attribute by running the jrelix statement:

     tmprel$<-$[selectedattribute] where layer=activelayer in maprelation;

     We use executeRelixCommand() as described in Section 4.2.3 to run the above statement.

   - Read the tmprel and construct the following array of values:

     $\{$(value[1], color[1]), (value[2], color[2]), ..., (value[n], color[n])$\}$, where color[i]=(255-i*colorgap,i*colorgap, i*colorgap) and colorgap=256/(number of tuples in tmprel). The above three values for a color represents the R, G, and B respectively.

- Use the above array to update the original map relation. For example, if the value of the selected attribute is value[i], the color value of this tuple must be changed to color[i]. Therefore, the tuples with different values of the selected attribute obtain different color values.

3. Update the Geditor legend array accordingly.

When this function returns to the Geditor Controller, the controller will update the map and legend in the Map View window with the updated map relation and legend array.

B. Range Map

Range maps assign different colors to different ranges of values of a descriptive attribute. The algorithm goes as follows:

1. Lay out the components inside the Range Map dialog and display it. The active layer name, the list of descriptive attributes, a list of color ramps, the range values and the sample of the selected color ramp are displayed in the dialog. The active layer name is also obtained from the Geditor legend array. The range values of the selected attribute can be obtained by dividing the range of [min,max] into five equal ranges, where min and max are the minimum and maximum values of the selected attribute. Then wait for the user input in the dialog until the user closes the dialog with the OK or Cancel button.

2. If the user closes the dialog with OK button, update the color value of the tuples of the active layer in the map relation. Suppose there are five ranges: r[1], r[2], r[3], r[4], r[5] which have been calculated according to the user input in the dialog. The color values clr[1], clr[2],clr[3],clr[4],clr[5] for the five ranges can be obtained from the user choice of the predefined color ramp. Therefore, change the color value of the tuples in the map relation as follows: if the value of the selected attribute is in range r[1], change the color value to clr[1], if the value of the selected attribute is in range r[2], change the color value to clr[2], and so on.

When this function returns to the Geditor Controller, the controller will update the map and legend in the Map View window with the updated map relation and legend array.

3. Legend Editor

The legend window can be changed using Legend Editor. The title of each layer and the label beside each graphical legend can be changed. The algorithm is as follows:

1. Lay out the components inside the Legend Editor dialog and display it. The name, title, color, symbol and the label of the active layer are displayed in the dialog. Notice that if the current active layer is a uniform layer, there is only one row displayed in the dialog. If the current active layer is a non-uniform layer, there are multiple rows showing different colors, symbols and labels of map features in this layer.

2. Wait for the user input in the dialog until the user closes the dialog with the OK or Cancel button.

3. If the user closes the dialog with OK button, update the Geditor legend array according to the user input.

When this function returns, the Geditor controller updates the legend according to the updated legend array. The map does not need to be updated in the Map View window.

4.3.6 Query

Geditor provides three ways of querying the map: Identify Tool, Expression Builder, and Spatial Query. With Identify Tool, when the user clicks the mouse close to a map feature, all the values of descriptive attributes will be listed in a pop-up window. With Expression Builder, the map feature that satisfies the user's expression will be turned yellow (default color). With Spatial Query, the map feature that satisfies the user-defined spatial relationship with a given map feature will be turned red (default color). The user selection in both Expression Builder and Spatial Query can be cleared using Clear Selections. The Options menu item provides the capability of changing the default color values used in Geditor.

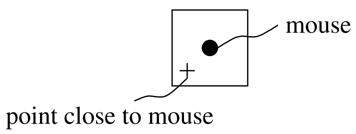
A. Identify Tool

Identify Tool lists the descriptive attribute values (refer to Section 3.1.2 for the definition) of the map feature in the current active layer when the user clicks the mouse close to this map feature. The algorithm is as follows:

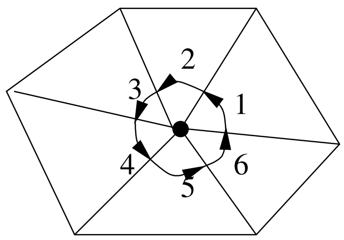
1. Lay out the components inside the Identify Tool window and display it. Obtain the active layer name from the Geditor legend array and display it on the top of the pop-up window. In the middle of the pop-up window, display an empty list that will be filled with attribute values. An "Identify" toggle button is displayed on the bottom of the window.

2. Then, check the type of the map features of the current active layer.

- If it is Point type, find the point that is within a small distance of the mouse position on screen.

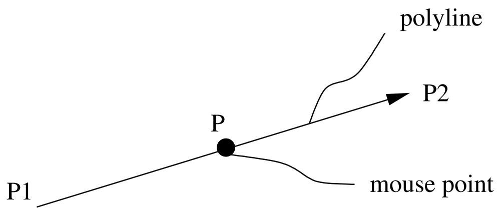


- If it is Polygon type, find the polygon that contains the mouse (the calculation is approximate).



if the angles 1+2+3+4+5+6=2pi, then the mouse point is in the polygon

- If it is Polyline type, find the polyline that passes the mouse point (the calculation is approximate).



If the determinant of P, P1, and P2 equals zero, the mouse point P is on the line of (P1, P2)

3. When the map feature is found, if it is a point, display all the descriptive attribute values of this point in the pop-up window. If it is a polygon or polyline, we assume that all the descriptive attribute values in the same polygon or polyline group in the map relation are the same. Therefore, a tuple is picked randomly in the same group and the corresponding attribute values are displayed in the pop-up window. If there is no map feature that is close enough to the mouse, no value is displayed in the pop-up window.

4. In the pop-up window, the "Identify" button is initially pressed. If the user needs to keep the pop-up window on the desktop, but disable the Identify function, click the "Identify" button to bounce it up. After that, nothing will be displayed in the pop-up window when the user clicks the map feature. To enable the Identify function again, just click on the "Identify" button.

B. Expression Builder

The Expression Builder provides a dialog box which enables the user to select the map features in the current active layer with an expression. In the dialog, the active layer name, a list of descriptive attributes, a list of operators, and the list of values of the selected attribute are displayed. The expression is also displayed in the text area of the dialog while the user is building the expression. The algorithm of this function is as follows:

1. Lay out the components inside the dialog and display it. The active layer name is obtained from the Geditor legend array. Wait for the user input and display the user's input in the bottom text area of the dialog.

2. If the user closes the dialog with the OK button, obtain the user's input from the text area. This is the expression the user created.

3. Assemble the following jRelix statement: .SelRel<−where *expression* in maprelation;

   Then call executeRelixCommand() to execute the above statement.

   .SelRel contains the selected map feature.

   If the execution fails, display an error message showing that the expression the user built was illegal. Otherwise, make a copy of the current map relation and update this map copy by changing the color value of the selected map feature to yellow.

When this function returns to the Geditor Controller, it updates the map in the Map View window with the above updated map copy. The legends do not need to be updated.

C. Spatial Query

Spatial Query allows the user to select a map feature that has a particular spatial relationship with a given map feature. The given map feature is the one that is selected by Expression Builder, that is, the current yellow map feature in the Map View window. The algorithm is as follows:

1. Lay out the components inside the Spatial Query dialog and display it. The current active layer name is displayed on the top. A list of spatial operators is displayed in the middle. The value field is only enabled when the spatial operator is "Within Distance Of".

2. Wait for the user input until the user closes the dialog with the OK or Cancel button.

3. If the user closes the dialog with the OK button, generate the EventName according to the user's choice in the spatial operator list. For example, if the user chose "Within Distance Of", the EventName will be "withindist:".

4. Generate the four necessary relations needed in the execution of the event handler: .ActLRel, .SelRel, .ValueRel (only needed when the operator is "Within Distance Of"), and .SpqRel. The definition of these relations can be found in Section 3.6.

5. Call executeEventH(EventName, env) to execute the event hander. The executeEventH searches the current system table to find the corresponding event handler. If the event handler has been defined, it will be executed. Otherwise display an error message reminding the jRelix programmer to write the event handler before performing this Spatial Query.

When the Spatial Query function returns, the Geditor updates the map in the Map View window according to the result of the event handler (.SpqRel relation). The legends do not need to be updated.

D. Clear Selections

This function restores the Map View window before the query is performed. Section 4.3.2 has discussed about the implementation of this function.

E. Options

The Options function provides the dialog for the user to change the three default colors used in Geditor: the highlight color used in the legend window to show the current active layer, the color to show the selected map features from Expression Builder, and the color to show the selected map features from Spatial Query. The values of the above three colors are stored as three variables in the Geditor class. These are selcolor (for Expression Builder), spqcolor (for Spatial Query) and activecolor (for highlighting the active layer).

Therefore, the algorithm is as follows:

1. Lay out the components inside the Options dialog and display it. Obtain the current value of the three default colors from the Geditor variables: selcolor, spqcolor and activecolor. Three color patches shows these three color values. A color palette is displayed when the user clicks on a color patch. This allows the user to change the corresponding default colors. Wait for the user input until the user closes the dialog with the OK or Cancel button.

2. Change the corresponding Geditor variables (selcolor, spqcolor, or activecolor) to record the changed default color according to the user input in the dialog.

Chapter 5

Conclusion

This chapter begins with a summary of the work that has been accomplished. It concludes with suggestions for possible extensions and future enhancements.

5.1 Summary

This thesis presents the design and implementation of a GIS editor (Geditor) which becomes a jRelix component that allows the user to view and edit the map graphically. With this Geditor, a series of GIS functions related to the map can also be performed.

Geditor builds two interfaces for the user. One is the interface for the jRelix Programmer-User which builds the **gedit** syntax that allows the jRelix programmer to call and display the Geditor GUI. The other is a graphical interface for the End-User that allows the End-User to display the map and complete GIS functions.

The principle of the design and implementation of Geditor is to provide a flexible and extendable framework. Firstly, the second attribute list of the **gedit** syntax is extendable, therefore further implementations can accommodate more attributes in the second list. Secondly, by utilizing the event handler machanism of jRelix, Geditor generates proper events according to the users' requirements and transfer the task to the event handlers written by jRelix programmers. This makes it possible to customize the implementation of the corresponding GIS functions and change it dynamically during the running time.

Geditor incorporates a series of GIS core functions that are implemented using Aldat capabilities and a graphical display interface. This accomplished the goal of building GIS applications in an integrated architecture using a relational database. All the data including both spatial and non-spatial data are treated equally in jRelix. Some DBMS do not offer the flexibility of implementing the necessary spatial operations needed for GIS functions. However, jRelix definitely has the capabilities and the Geditor implementation is a good test. It is beyond the scope of this thesis to integrate all the possible GIS functions within Geditor. However, this thesis builds the fundamental elements and provides an extendable framework for future work on this issue.

5.2 Future Work

5.2.1 Extension of the second attribute list of gedit

As discussed in Section 3.1.3 and 4.2.2, the second attribute list of **gedit** could be extended. Currently, three attributes: color, symbol, and layer are implemented. The following is an example of a legal syntax:

[G,T,S,X,Y][Color=C, Symbol=Symb, Layer=L] gedit maprelation1;,

where maprelation1 is the same as what Table 3.1 shows.

In the future, Geditor may need more data to accomplish more advanced GIS applications. For example, if the more sophisticated map display requires the capital of each country to flash on screen, a flash speed attribute might be needed. In this case, a fourth attribute could be added to the second list of the above example:

[G,T,S,X,Y][Color=C, Symbol=Symb, Layer=L, Fspeed=F] gedit maprelation;,

where maprelation should include the attribute F that indicates the flash speed of each map feature.

As another example, if the user needs to display the map in three dimensions, a third coordinate Z might be needed. Therefore, another attribute that shows the Z coordinate must be added to the second attribute list of **gedit** as follows:

[G,T,S,X,Y][Color=C, Symbol=Symb, Layer=L, Fspeed=F, Zvalue=Z] gedit maprelation;

In maprealtion, the attribute Z indicates the Z coordinate of each map feature.

If more thematic maps are built in Geditor, more attributes need to be added to the second list of **gedit**. Recall the example about the Dot Density map in Section 1.2.2.A.6. The dots inside each polygon are based on the number of people in that area. Therefore, the second attribute list of **gedit** could be extended as follows:

[G,T,S,X,Y][Color=C, Symbol=Symb, Layer=L, Fspeed=F, Zvalue=Z, DotDens=Pop] gedit maprelation;

This is similar to the Graduate Symbol map and the Chart Symbol map discussed in Section 1.2.2.A.6.

When other functions are added to Geditor, more attributes might also be needed in the second attribute list of **gedit**. For example, if Geditor implements the contour map, the attribute for drawing the contour lines needs to be added to the second attribute list. Suppose the contour lines are drawn based on the altitude of each polygon vertex, the **gedit** syntax becomes:

[G,T,S,X,Y][Color=C,Symbol=Symb, Layer=L, Fspeed=F, Zvalue=Z, DotDens=Pop, Contour=Hgt] gedit maprelation;,

where maprelation contains the attribute Hgt that shows the altitude of each polygon vertex.

When the contour lines are drawn, interpolation techniques must be used to generate these lines.

The examples could go on and on and the second attribute list could become very long. However, regardless of the number of attributes that are added, the Parser does not have to be rewritten because the **gedit** syntax allows the second attribute to be extended. Only the Interpreter needs to be modified accordingly.

5.2.2 Enhancement of Map Display

In this version of Geditor, all polygons are filled with colors and all polylines are drawn in solid lines with the same line thickness (one pixel). The different filling patterns of polygons and the different polyline styles are not implemented. In the further implementation, the values of the symbol attribute could be used to indicate the line types and polygon filling patterns. As mentioned in Section 4.3.3, in future work, the JFC 2D package could be used. There are existing methods for filling polygons with different patterns and drawing lines with vaiable thickness and different styles in the JFC 2D package.

After these display features are added, different line styles can show different road types, such as highway, freeway, streets, railroad, and subway. Polygons can also be filled with different patterns, which is important when black and white pictures are needed.

The current version of Geditor displays maps in two dimensions. However, 3D display is becoming increasingly popular in GIS applications. To achieve this goal, firstly, in addition to the X,Y coordinates already stored in the map relation, the Z coordinate would have to be added. Secondly, the second attribute list of **gedit** would have to be extended as discussed in Section 5.2.1. Thirdly, a 3D display package needs to be used to achieve the 3D visualization. Java 3D[tm] API which enables programmers using Java[tm] technology to do 3D visualization might be a good choice.

5.2.3 Integration of more GIS functions

As discussed in Section 1.3, Geditor implements the typical common functions of GISs, especially those display-related functions. Some GIS functions are left out and it would be nice to incorporate them into Geditor in the future.

A. Polygon Overlay, Dissolve, and Buffer Generation

Polygon overlay, dissolve, and buffer generation functions are left out in Geditor because they are not display-related except for the results display. These functions can be achieved within Aldat capabilities and Martinez has already provided the Aldat codes for these functions [Mar98]. However, since the above three functions play an important role in GIS applications, perhaps in the future, Geditor can integrate them into Geditor window. This can be achieved easily by utilizing the event handler mechanism of jRelix. Similar to the implementation of Spatial Query function, when the user requires these functions, say polygon overlay, Geditor generates a proper event. Then the corresponding event handler written by the jRelix programmer will be called and executed to complete the operation. The results can be displayed using the map display class (CvDraw.java) already implemented in this Geditor.

B. Measurement

- Measurement of Points

  As discussed in Section 1.2.2.A.2, the measurement of points is related to the counting of points in a user defined windowing polygon. Therefore, an interface is needed to capture the user input of the windowing polygon. Then the Aldat capabilities can locate the points entirely within the windowing polygon and count the number of such points. When integrating this function into Geditor, a component needs to be added to capture the user's windowing polygon. Aldat codes which locate and count the points entirely within the windowing polygon, can be executed using executeRelixCommand() method. Finally, the result (number of points) can be displayed in a pop-up window or a fixed text field in the Geditor window.

- Measurement of Lines and Polygons

  As discussed in Section 1.2.2.B.2, an interface is needed to allow the user to specify the polygon or the polyline to be measured. This can be achieved by the selection of map features using Expression Builder, which has already been implemented in the current Geditor. The user might also need another selection tool to highlight the map feature using the mouse. A minor modification of Identify Tool can accomplish this. When the user clicks the mouse close enough to the map feature, instead of showing all the descriptive attribute values of the map feature, the selection tool may simply highlight the map feature on screen. Then, Aldat codes which calculate the area or perimeter of the polygon, edge length or the whole length of the polyline can be passed to executeRelixCommand(). Finally, the results of the measurement can be displayed in a pop-up window or a fixed text field in the Geditor window.

C. Spatial data editing

This set of editing operations is used to correct the errors in the data capture stage. It includes map generalization, rubber sheeting, and snapping that are discussed in detail in Section 1.2.2.B.4. A Graphical editing interface needs to be added to Geditor to allow the user to specify the map feature and edit it interactively. Different specific editing interfaces need to be implemented for different editing functions. For example, for polygon thinning, an interface for the End-User should allow the user to delete vertices by clicking them and show the result polygon with the remaining vertices. Finally, when the user is satisfied with the modified map feature, the results need to be saved for future use.

To implement the saving of the graphic data with the modified map features, there are two options. One is keeping **gedit** functional, the other is changing **gedit** to a non-functional operator.

Currently, **gedit** is a functional unary operator, that is, gedit does not change the value of its operand (the map relation). Furthermore, the value of the **gedit** expression is also the same as its operand: the original map relation. When the spatial editing is implemented, if we keep **gedit** functional, we still do not change the value of the operand of gedit, but let gedit return a different value from the original operand (the original map relation). In this case, Geditor cannot run on an independent thread from the Interpreter as Section 4.2.1 describes. When the Interpreter calls Geditor, it has to wait when Geditor interacts with the End-User in its GUI. As discussed in Section 4.2.1, Geditor obtains a copy of the original map relation and updates this copy when necessary. In the spatial editing case, when the user is satified with the modified map features, the result will be saved in this map relation copy. When Geditor finishes and returns to the Interpreter, the gedit expression obtains the map relation copy as its value. This value can be assigned to a new map relation for future use. The original map relation is kept unchanged during the whole process.

The other approach to implementing the saving is to change the gedit to a non-functional unary operator. In this case, gedit must fit into **update** syntax, not **expression**. **gedit** will be allowed to change the value of its operand, that is, the original map relation. The Interpreter still has to wait until Geditor finishes and returns, but the content of the original map relation can be changed during the Geditor interaction with the End-User in its GUI. As far as spatial editing is concerned, when the user is satisfied with the modified map features, the result will be saved into the original map relation directly. This implementation approach changes **gedit** to a non-functional operation.

D. Map Sheet Manipulation

Map sheet manipulation includes projection change, coordinate translation, scale change and rotations. With the exception of the result display, these functions can be achieved using Aldat capabilities. Dialogs with the user need to be added to Geditor to obtain the parameter data for completing those operations. Then the corresponding Aldat codes can be executed using event handler mechanism or the executeRelixCommand() method. For example, for the scale change, Geditor needs to know how much percent smaller or larger the user wants the map to be. This can be captured by displaying a list of percentages in a dialog or on the toolbar to allow the user to choose. Then the X,Y coordinates can be updated in the map relation by multiplying by the chosen percentage. Aldat codes with the Update command can achieve this. The executeRelixCommand()

method or event handler mechanism can be used to run these Aldat codes. Finally, Geditor calls the map display class (CvDraw.java) to display the map with the updated map relation. As a result, the map will be zoomed in or out.

There are other functions that could be added to Geditor. For example, more thematic maps (such as dot density map and chart symbol map), spatial interpolation in terrain analysis, network analysis, image processing, and so on. With the development of GIS applications, more and more functions will emerge and therefore need to be included in Geditor. By implementing this Geditor with GIS core functions, this thesis leads the way for the implementation of more complicated GIS applications using Aldat capabilities and a graphical interface.

Appendix A

Backus-Naur Form for gedit

This appendix presents the Backus-Naur form (BNF) of the grammar in our implementation. Only the new added syntax (**gedit**) and the modified syntax will be provided here. A complete documentation of the original jRelix grammar/syntax in BNF format is given in [Sun00].

The grammar is created from the grammar specification (in file Parser.jjt), using the JavaCC documentation generator called `jjdoc`. In the BNF definition, terminals will be quoted and non-terminals will be otherwise. The sign | means or. (...|...|...) means choosing one of the components separated by | inside the brackets. (...)? repeats the component inside the brackets zero or one time.

In this Geditor implementation, we created the **gedit** grammar/syntax and modified the syntax of event handlers in jRelix. The following is the BNF notation of the new **gedit** syntax and the updated event handler syntax.

```
Projection := Projector ( ( "in" | "from" ) Projection
                          | Projector "gedit" Expression
                          | "gedit" Expression | Selection )

EventName := ( Prefix ":" )?  Action ":" ( Identifier )?

Action := "add" | "delete" | "change" | "contains" | "cmpcontains"
          | "cmpwithin" | "intersect" | "within" | "withindist"
```

Bibliography

[Abi93] Serge Abiteboul, Georg Lausen, Heinz Uphoff, and Emmanuel Waller. Methods and rules. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):32-41, June 1993.

[Atk83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrision. PS-algol: A language for persistent programming. In *10th Australian National Computer Conference*, pages 70-79, Melbourne, Australia, 1983.

[Atk84] M. P. Atkinson, W. P. Cockshott, P. Bailey, K, J. Chisholm, and R. Morrison, *PS-algol reference manual*, Technical Report PPR-4-83, Department of Computer Science, Universities of Edinburg and St. Andrews, January 1984.

[Bak98] Patrick Baker. *Java Implementation of Computations in a Database Programming Language*, Master's thesis, McGill University, 1998.

[Boc86] J. Bocca. EDUCE: A marriage of convenience: Prolog and a relational DBMS. In *Proceedings of the International Symposium on Logic Programming,* pages 36-45. IEEE Computer Society, The Computer Society Press, September 1986.

[Bur86] Burrough P.A.. *Principles of Geographical Information Systems for Land Resources Assessment*, Oxford:Clarendon Press, 1986.

[Cer89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog(and never dared to ask). *IEEE Transactions on knowledge and Data Engineering*, 1(1): 146-166, March 1989.

[Cha86] C. L. Chang and A. Walker. PROSQL: A Prolog programming interface with SQL/DS. In L. Kerschberg, editor, *Expert Database Sys.*, page 233. Benjamin/Cummings, Menlo Park, CA, 1986.

[Chr97] Nicholas Chrisman. *Exploring Geographic Information systems.* John Wiley&Sons, 1999.

[Cla99] Keith C. Clarke. *Getting Started with Geographic Information Systems*, Prentice Hall, Upper Saddle River, New Jersy, 1999.

[Cod70] E.F.Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377-387, June 1970.

[DeM97] Michael N. DeMers. *Fundamentals of Geographic Information Systems*, John Wiley&Sons, Inc., 1997.

[Due89] K. J., Dueker and D. Kjerne. *Multipurpose cadastre: Terms and definitions*, Falls Church, VA:ASPRS and ACSM., 1989.

[Gea99] David M. Geary. *Graphic JAVA: Mastering the JFC, third Edition*, Sun Microsystems Press, Java Series, Palo Alto, California, 1999.

[Hao98] Biao Hao. *Implementation of the nested relational algebra in Java*, Masters thesis, McGill University, Montreal, Canada, 1998.

[Hey98] Ian Heywood, Sarah Cornelius, Steve Carver. *An Introduction to Geographical Information Systems*, Longman, New York, 1998.

[Hul89] Richard Hull, Ron Morrison, and David Stemple. *Proc. of the 2nd workshop on Database Programming Languages*, Salishan Lodge, Oregon, page xi, June 1989.

[Hut97] Scott Hutchinson, Larry Daniel. *Inside ArcView GIS*, OnWord Press, U.S.A., 1997.

[Ioa94] Y. E. Ioannidis and M. M. Tsangaris. The design, implementation, and perfomance evaluation of BERMUDA. *IEEE Transactions on Knowledge and Data Eng*, 6(1):38, February 1994.

[Jen85] Kathleen Jensen and Niklaus Wirth. *PASCAL User Manual and Report (third edition)*, Springer - Verlag, New York, N.Y., 1985. Revised to ISO Standard by Andrew B. Mickel and James F. Miner.

[Kor97] George B. Korte, P.E. *The GIS Book*, OnWord Press, U.S.A., 1997.

[Lam91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system, *Communications of the ACM*, 34(10):50-63, October 1991.

[Mar98] Martinez Angelica Valdivia. *Implementing of G.I.S. Spatial Operations in a Database System*, Master's thesis, School of Computer Science, McGill University, Montreal, 1998.

[McC89] D. McCarthy and U. Dayal. The architecture of an active data base management system, *Proceedings of ACM SIGMOD,* Portland, Oregon 1989, 215-224.

[Mer77] T. H. Merrett. Relations as programming language elements. Information Processing Letters, 6(1):29-33, 1977.

[Mer84] T. H. Merrett. *Relational Information Systems*, Reston Publishing Co., Reston, VA, 1984.

[Mor86] K. Morris, J.D. Ullman, and A. Van Gelder. Design overview of the Nail! system, *Proc. of International Conference of Logic Programming*. New York: Academic, 1986.

[Mor87] K. Morris, J. Naughton, Y. Saraiya, J. Ullman, and A. Van Gelder. YAWN!(Yet another window on NAIL!), Special Issue on Databases and Logic, *IEEE Data Engineering,* vol. 10, Dec. 1987.

[Mor88] R. Morrison. PS-algol reference manual. Technical Report 12, University of St. Andrews, St. Andrews, Scotland, Febrauary 1988.

[Peu90] Donna J. Peuquet and Duane F. Marble. *Introductory readings in Geographic Information Systems*, Taylor&Francis, London, 1990.

[San96] Sriram Sankar, Rob Duncan, and Screenivasa Viswanadha. Java Compiler compiler(JavaCC)-The Java Parser Generator, JavaCC web site at: http://www.suntest.com/JavaCC/, 1996. The web site contains documentation, FAQs, newsgroups, and software for JavaCC and JJTree.

[Sch77] Joachmim W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247-261, September 1977.

[Sta90] Jeffrey Star and John Estes. *Geographic Information Systems: An Introduction*, Prentice Hall, New Jersey, 1990.

[Sto76] M.R.Stonebraker, E. Wong, P. Kreps, and G.D. Held. *The design and implementation of INGRES.* ACM Transactions on Database Systems, 1(3):189-222, September 1976.

[Sun00] Weizhong, Sun. *Updates and Events in a Nested Relational Programming Language,* Master's thesis, McGill University, Montreal, Canada, 1998.

[Ull85] J. D. Ullman. Implementation of logic query languages for databases, *ACM Trans. Database Syst.*, vol. 10, no. 3, 1985.

[Vla98] I. Vlahavas and N. Bassiliades. *Parallel, Object-oriented, and Active Knowlege Base Systems*, Kluwer Academic Publishers, Boston/Dordrecht/London, 1998.

[Whi99] Anela Whitener, Paula Loree, and Larry Daniel. *Inside MapInfo Professional*, OnWord Press, 1999.

[Wor99] M.F. Worboys. Relational databases and beyond, in *Geographical Information Systems, Volume 1*, John Wiley & Sons, Inc., New York/Chichester/Weinheim/Brisbane/Singapore/Toronto, 1999.

[Yua98] Zhongxia Yuan. *Implementation of the domain algebra in Java*, Master's thesis, McGill University, Montreal, Canada, 1998.

[Zei97] Michael Zeiler. *Inside Arc/Info*, OnWord Press, U.S.A., 1997.