GeoInformatica 3:1, 7–32 (1999)

Developing GIS Applications with Objects: A Design Patterns Approach

SILVIA GORDILLO*, FEDERICO BALAGUER, CATALINA MOSTACCIO*, AND FERNANDO DAS NEVES
LIFIA- Departamento de Informática, Facultad de Ciencias Exactas,
Universidad Nacional de La Plata—Argentina
e-mail: [gordillo,fede,catty,babel17]@sol.info.unlp.edu.ar

Received March 24, 1998; Revised November 25, 1998; Accepted November 30, 1998

Abstract

In this paper we present an object-oriented approach for designing GIS applications; it combines well known software engineering practices with the use of design patterns as a conceptual tool to cope with recurrent problems appearing in the GIS domain. Our approach allows the designer to decouple the conceptual definition of application objects from their spatial representation. In this way, GIS applications can evolve smoothly, because maintenance is achieved by focusing on different concerns at different times. We show that our approach is also useful to support spatial features in conventional applications built with object-oriented technology. The structure of this paper is as follows: We first introduce design patterns, an efficient strategy to record design experience; then we discuss the most common design problems a developer of GIS applications must face. The core of our method is then presented by explaining how the use of decorators helps in extending objects to incorporate spatial attributes and behavior. Next, we analyze some recurrent design problems in the GIS domain and present some new patterns addressing those problems. Some further work is finally discussed.

Keywords: spatial data models, object-orientation, design patterns

1. Introduction

Geographic Information Systems deal with many complex aspects: data acquisition, accuracy, representation of spatial relationships, topological features and interface design are some of the characteristics designers must consider when developing geographic applications.

One problem with this kind of applications is that there is no design method that covers all these aspects. The lack of a well-grounded design approach leads to some undesirable consequences in the final products in terms of reusability, modularity, modifiability, etc.

Object-oriented methodologies have proved to be a good solution for the design of non-conventional applications where the complexity of data and the underlying relationships are critical. Using this technology, we obtain not only reusable, modular and modifiable software, but also we can get interoperable systems, as objects encapsulate knowledge that can be customized to different needs.

*Also Comisión de Investigaciones Científicas de la Pcia. de Buenos Aires.

Based on object-oriented technology, design patterns appear as a powerful strategy to represent good design solutions to recurrent problems. By using objects and patterns we define a two-step design approach aimed at representing the application in terms of entities of the real world and to provide general solutions to solve situations appearing during the development process, like the use of different reference systems, locations definition, etc. Though this architecture is restricted to handle vector data, we have already defined some extensions for working with continuous information [8].

In Section 2 we introduce design patterns. In Section 3 we describe some problems appearing in the design of GIS applications. In Section 4 we explain our design method while in Section 5 we explain how we use the Decorator, the Reference System, the Roles and the Appearance design patterns to obtain a geographic model. Finally, in Section 6 we present some conclusions and future work.

2. Design patterns. An introduction

The use of object technology is a growing trend in the design of GIS applications [9], [10]. We claim, however, that object-oriented design methods and class libraries are only part of the solution for designing GIS applications. In this domain there are many recurrent problems involving the use of spatial information: object locations, coordinate manipulation, computation of geographic functions, and so on.

Christopher Alexander [1] defined the purpose of design patterns as follows: ''*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of a solution to that problem, in such a way that you can use this solution a million times over, without ever doing the same thing twice*''.

Design patterns are usually described by stating the problem in which the pattern may be applied, the elements that make up the design, their relationships, responsibilities and collaborations [6]. These elements are described in an abstract way because patterns are like templates that can be applied in many different situations. The consequences and tradeoffs of applying patterns are also important because they allow evaluating design alternatives.

Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

A typical pattern has four essential elements [6]:

- The **pattern name** which is used to describe a design problem, its solution and consequences in a concise way;
- The description of the **problem** in which the pattern could be applied;
- The **solution** of that problem describing the elements (classes and objects) conforming the design, their relationships, responsibilities and collaborations;
- Finally, the **consequences** explaining the results of applying the pattern; consequences are useful for the evaluation of different design alternatives and benefits of the adopted solution.

In order to clarify the notation used along the paper we present the basic elements of the OMT notation [14] that we have adopted to express the structure of each pattern. We also present an example of a design pattern defined in [6] as a way to illustrate the purpose and description of patterns. We use a Smalltalk-like syntax (where operations that accept parameters ends with ':') to define attributes and methods of classes.

There are two kinds of diagrams we use to denote relationships and interactions between classes and objects:

- A **class diagram** depicts classes, their structure and the static relationships among them. Figure 1 shows the notation of this kind of diagrams. Figure 2 shows different kinds of relationships between classes presented in [6].

In figure 2 the reader can see three types of relationships used commonly in object-oriented designs. The inheritance relationship, that allows a subclass to inherit the behavior of its parent class; the aggregation relationship meaning that an object owns another one; and the acquaintance representing that an object ''knows of'' another one (this is the common association between two classes).

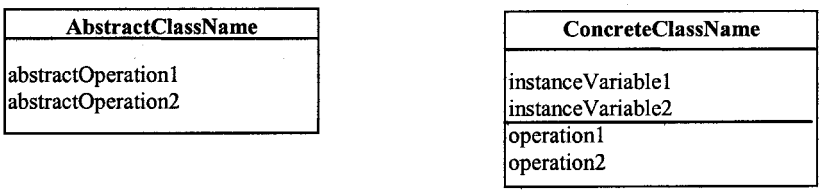


Figure 1. Class diagram notation.

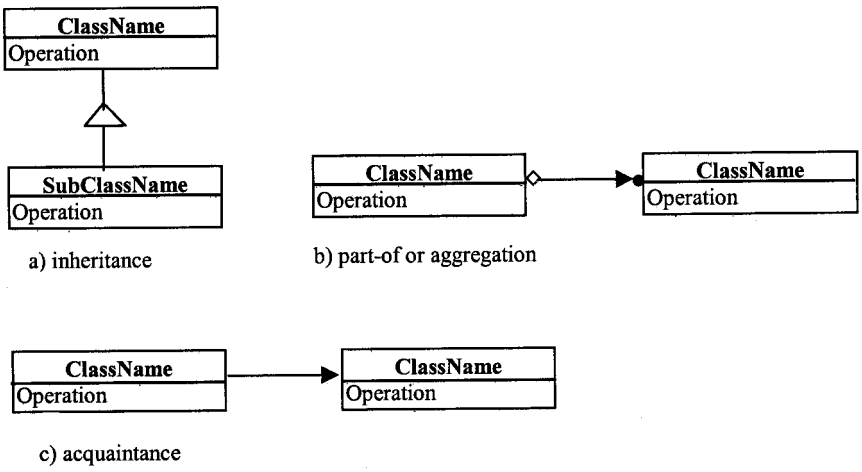


Figure 2. Relationships between classes.

- An **object diagram** depicts a particular object structure at run-time. Its notation is shown in figure 3.

2.1. An example: The composite design pattern

Composite is a pattern for composing objects in such a way that both individual objects and the composition are treated uniformly. It works by implementing an abstract class that defines the common interface for both the individual objects and the containers.

Figure 4 shows the general **Composite** structure.

Composite defines the interface to manipulate its components and delegates pertinent operations to each one of its components (see the **operation** behavior in figure 4).

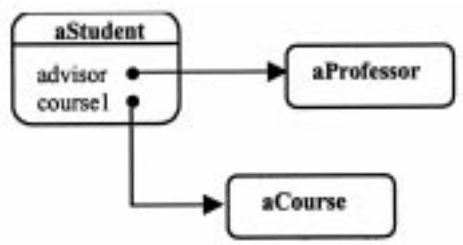


Figure 3. Object diagram notation.

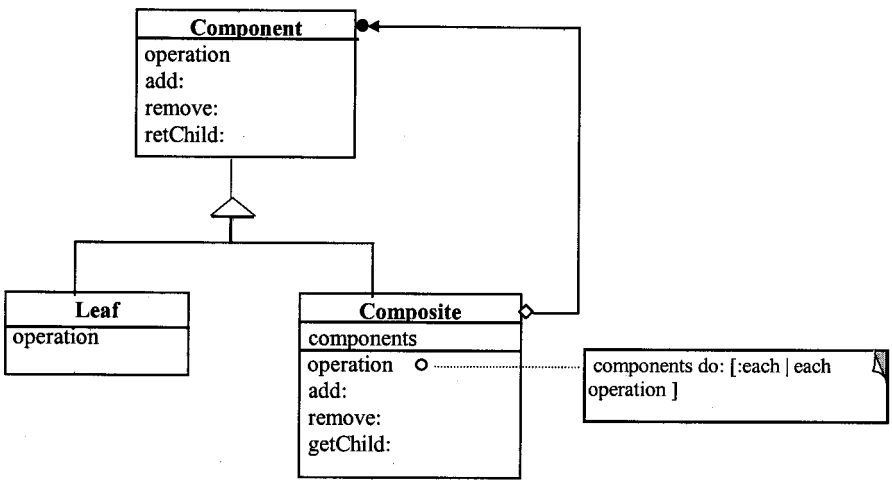


Figure 4. The structure of the composite design pattern.

Leaf represents primitive objects in the composition and defines their behavior.

In order to understand how this pattern can be used consider the case of a graphic application like a drawing editor that lets users build complex diagrams out of simple elements. Using this editor the users can group components to form larger ones. This can be made using a technique called recursive composition. The **Composite** design pattern captures the essence of recursive composition in object-oriented terms. For example, suppose we have, as it is shown in figure 5, an area that is conformed by a set of blocks, which in turn are composed by basic graphical elements, representing empty lots, buildings and other blocks.

The structure of classes in the example is shown in figure 6.

Participants. **Area** defines the interface for all objects in the composition. It declares primitive operations like *draw*: and operations for accessing and managing its child components like *getChild*: or *add*:

Classes for graphical primitives such as **Building** and **EmptyBlock** represent basic objects in the composition. Each subclass implements the necessary primitive operations:

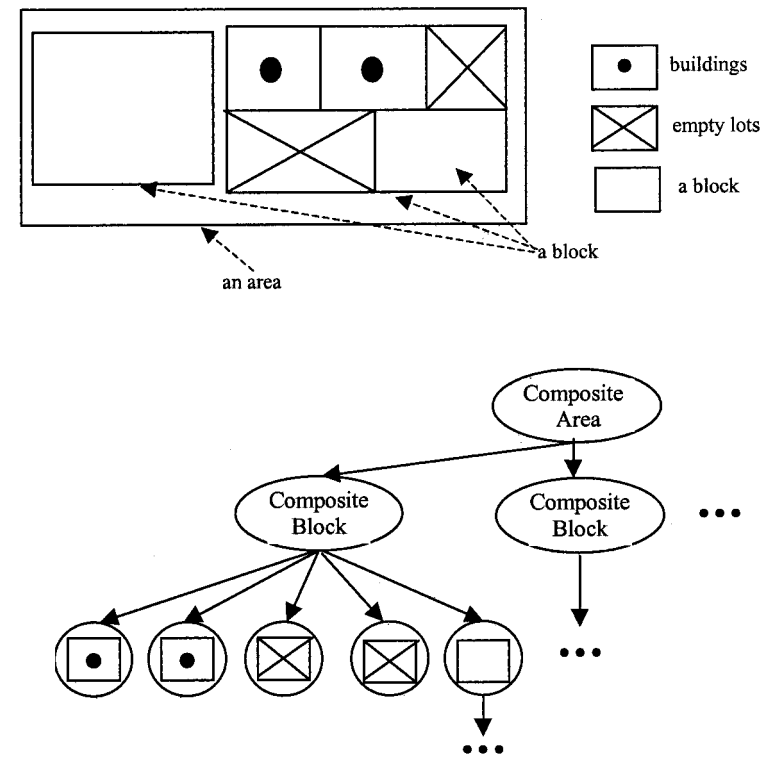


Figure 5. Composition of empty lots and buildings.

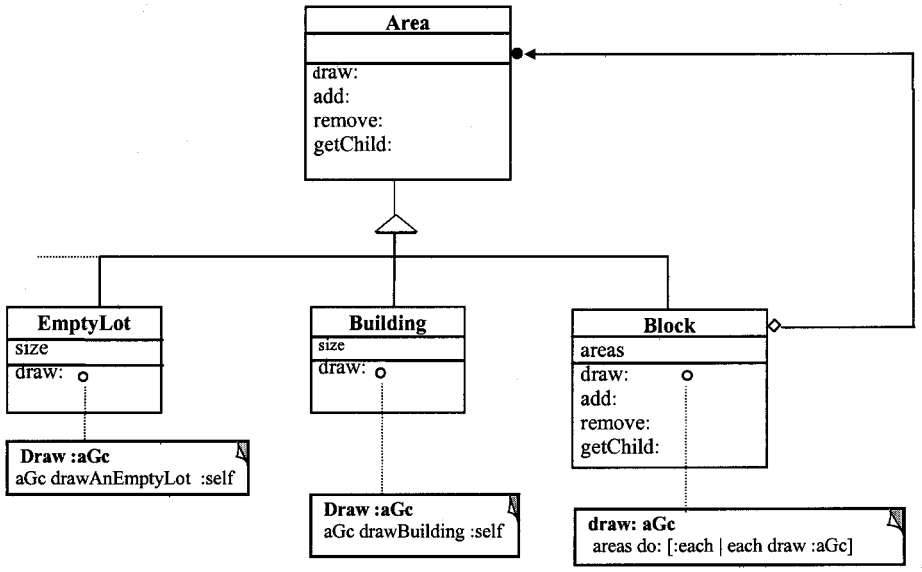


Figure 6. An instance of the composite design pattern.

for example the *draw*: operation is defined in each subclass and it draws the corresponding element in a given graphic context.

Block defines the behavior for any composite structure and stores the child components and implements the way to handle them.

Collaborations. Clients use the interface defined in the **Area** class to interact with the composite object. If the receiver is a basic element the corresponding primitive class directly handles the request. If it is a composite element, then it delegates the request to its child components.

Consequences. Clients deal uniformly with simple or composite structures by using the abstract interface.

The addition of new kinds of elements, (composite or leaf subclasses), can be made without important modifications to both, the composite structure and the design of the application.

Because of the flexibility to add new elements, it becomes difficult to restrict the quantity of components in the composition.

Design patterns are a useful mean for specifying and documenting an application. Similar applications can be built by instantiating the same patterns, which can eventually be specialized according to user needs [4]. Also, they provide a concise and powerful vocabulary that allows designers to improve communication within the development team, because of the higher level of abstraction of the discourse [15].

3. Problems in the design of GIS applications

The complexity of the underlying domain and the great variety of data that characterize GIS applications, make the design process of this kind of systems difficult and complex. In this context, modeling and design activities require a deep expertise on the application domain. Usually, no systematic design method is used to record design documents and rationales. A cyclic problem is thus generated, since designers are more concerned in obtaining efficient implementations than good designs of applications. However, when modifications and/or extensions have to be made, the lack of good design documents forces designers to do an untidy work, generally loosing the semantics of changes due to the weakness of the model; as a result, performance is also compromised.

A typical mistake in this kind of applications appears when entities are designed in terms of their spatial representation; the emphasis is thus focused in ''how the information is represented'' instead of ''what information or behavior is needed''. A consequence of this approach is that entities with different representations (for example with different scales) are usually implemented as different objects while they are not different from a conceptual point of view. In this context is understandable that concepts like reuse, maintenance and evolution become difficult to apply [17].

There is a growing trend in using object-oriented concepts to design the abstract structure of geographic applications [9], [10]. Although these methods build a valuable foundation for what is needed to support the application development process, there are some problems that designers must consider:

- Specifying application entities by mixing their physical representations (points, polygons or lines) with their conceptual behavior, forces the designer to prematurely assign spatial features to those entities; moreover, this practice makes it difficult to work with different representations of the same object, e.g., for displaying it at different scales.
- The semantics of the domain is usually difficult to express, and textual annotations must be added to the resulting model. Most of the times, object-oriented GIS design methods are extensions of design methods that were conceived for other kinds of applications. Generally the semantics of those extensions are poorly specified, weakening the ability of the design notation to embrace the GIS domain.

Nevertheless, extending conventional applications with spatial features has also become a need. In the last times, more and more designers are building GIS applications based on open systems instead of using a particular GIS product (like ARCInfo, GENAMAP, etc.). The rapid growing of the WWW as a host for different kinds of applications, and the emergence of Java as a programming language well suited for developing distributed applications, show us the need to find a systematic approach for recording and reusing our design experience. This need becomes crucial when a designer must face hybrid applications, i.e., those dealing with conventional transaction-based systems that must be upgraded to include spatial features that were not considered in the underlying software. Examples of extensions of traditional applications with spatial features are shown in [13].

We find real estate agencies, telecommunications companies, hotel chains, news organizations, and survey entities usually struggling to add geographic information to their products or legacy information systems. For example, real estate agencies can browse maps finding alternatives that satisfy customer preferences (cost, distance to downtown, neighborhood style, etc.). All these applications have to support geographic queries dealing with objects within the geographic domain (downtown, neighborhood), but also with non-spatial objects like houses, cities, etc., which could have been defined in the conceptual domain.

Web applications providing access to geographic data captured from legacy systems are another example of this kind of applications. Most of the times, developers have to write customized code to visualize the required information.

In this paper we propose an object-oriented method to design GIS applications. The method is divided in two main steps: during the first one (conceptual modeling) we model conceptual features of the application, delaying the spatial features to the second step (geographic modeling). This approach makes possible not only the development of new applications, but also the extension of existing ones with geographic features, since they can be treated as the result of the first step.

4. Modeling domain entities. The conceptual model

When designers build GIS applications, they deal with two different kinds of data types. One represents conceptual data describing entities in terms of descriptive attributes. For example, if they are modeling a country in the context of a geographic application, typical descriptive attributes are the name of the country, its first language, or its government system. The other kind of data is that representing all aspects related to geographic features, like the country boundaries. These two data types define two separate databases (in most GIS environments) at the implementation level: one of them contains spatial information and the other one stores the conceptual characteristics (usually stored in a relational database).

At the functional level something similar happens; designers have to face different aspects of the same problem. Not only conventional operations like tax-payment or traffic statistics records have to be defined, but also operations involving spatial attributes, such as areas that are influenced by some phenomena, or entities holding a particular spatial property.

The above discussion shows that geographic and conceptual features must be treated in a different way: dealing with data captured with a remote sensing method is not the same as dealing with string, character or numerical data. However, in a geographic application commonly both aspects have to be covered and integrated in a consistent way.

Since the complexity of the development resides in the definition and use of the spatial information, we propose a two step iterative design process as a way to make this task easier and clearer [7]. The method leads to a strong and transparent integration between both spatial and conceptual information.

During the first step, Conceptual Modeling, we describe the application domain in terms of entities in the real world. The intent is to understand the problem by providing an abstract representation in which geographic features and implementation details are ignored; only those class responsibilities that are space independent are defined here. In this step, we use the object-oriented model to represent the application. Therefore, the result of this step is not different from those obtained in conventional (non-GIS) applications solved with object-oriented design techniques.

The method allows us to manipulate different abstraction levels by separating the comprehension of the descriptive world from the geographic one. Moreover, this process also provides a flexible way to extend conventional applications built with object-oriented technology, by taking advantage of the conceptual model just defined and thinking in terms of adding spatial characteristics instead of redefining the whole system.

5. Describing spatial features. The geographic model

In this step the goal is to define the spatial features for those classes defined in the conceptual model. We propose enriching the conceptual model instead of modifying it. We first identify which classes in the conceptual model will contain spatial features; then, for each one of these classes, we define a new one that wraps it with the spatial behavior by applying the Decorator design pattern.

In the following section we describe in detail the use of Decorators to design geographic applications.

5.1. Using decorators to specify the geographic model

Decorator is a design pattern defined in [6]. Its intent is ''to attach additional responsibilities to an object dynamically''. Decorators provide a flexible alternative to subclassing for extending a class' functionality. The substantial difference between decorating and subclassing is that while the former is dynamic, the latter is static; by using decorators we can add functionality to some objects without re-defining the conceptual hierarchy.

A decorator replicates the interface of the decorated object; it also defines additional behavior. It forwards requests to the component and may perform additional actions before or after forwarding [6].

In this way, we can define objects and decorate them with different behaviors, for example spatial behavior, and refer either to the original object or to any of their decorators. In our approach, since decorators mimic the protocol of the conceptual object, they delegate the implementation of the non-spatial protocol to that object.

We use the idea of this design pattern to construct the basic geographic model, by adding spatial features to each object in a dynamic and transparent way. The same schema is useful both as a conceptual tool for new designs and to leverage existing designs in order to include geographic information.

To show an example, suppose that we have a **Country** class in the context of an application that models different aspects of a region. Countries contain conceptual attributes such as name, first language, government system, currency, etc. Later, when geographic information about countries is needed, we will have to upgrade the **Country** class to include spatial information, in order to perform operations like width of a country in a particular latitude, its location, neighbor countries, etc. Figure 7 shows both conceptual and geographic definitions; additional features have been defined in the second one.

Countries have more than one possibility to solve the enhancement of the conceptual class with geographic features. The most obvious (and also the less convenient) implies building the whole application again in terms of the geographic classes. The second possibility is to use subclassification by defining additional features and behavior but, as we said before, this is not the best solution in order to build a flexible application. Moreover, by using this mechanism we obtain a mixed schema where conceptual and geographic classes are strongly coupled. In figure 8 shows the resulting schema for this solution.

The use of decorators helps to upgrade system functionality without modifying the existing class schema. The solution is to define a **GeoCountry** class, whose instances will work as decorators of **Country** objects. **GeoCountry** defines the geographic behavior that we need in the new application; for example, it includes a method to calculate the width of a country. Since there is a relationship between the decorator (**GeoCountry**) and the object, it decorates (**Country**); the former can delegate to the conceptual object, the execution of the already defined behavior (*name, firstLanguage, stateList, etc.*). Figure 9 shows the schema of the modification using decorators.

Decorators area more dynamical solution than subclassification since responsibilities can be added and removed in run-time, by attaching and detaching components into the decorated object.

With Decorators the resulting application manipulates two kinds of objects, conceptual and geographic ones, defining two separate levels. In this way, the model remains flexible

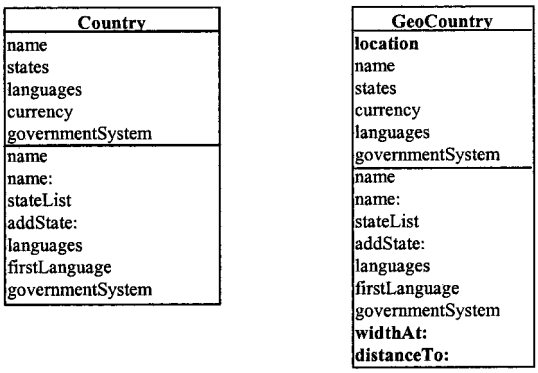


Figure 7. Conceptual and geographic classes.

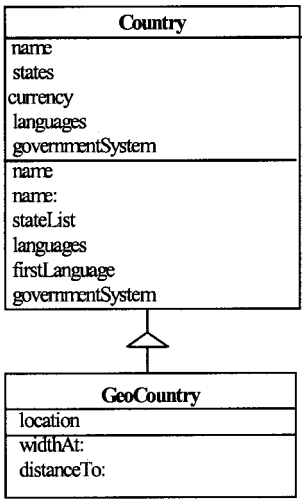


Figure 8. Adding spatial features by subclassifying.

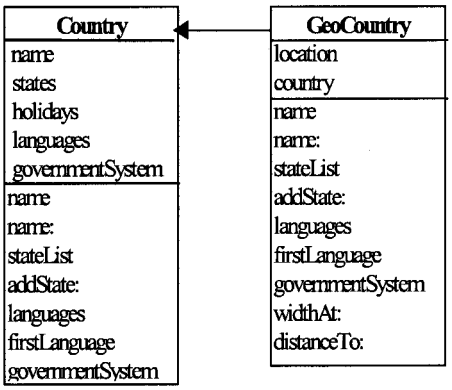


Figure 9. The GeoCountry class adds spatial features to the Country class.

because we can define many decorators for a specific object or even decorate a decorator, thus nesting geographic specifications. When we are extending existing applications, we can do it transparently because the conceptual model does not suffer modifications; therefore, the original application can still be used as it was originally conceived.

One restriction we have when using this method appears when we add or erase a conceptual feature. In this case we have to add or delete the corresponding feature in the decorator.

Geographic objects in our approach always know their location and have an associated geometry. Location has been defined as in [11]; it contains the position of the georeferenced object and some behavior that is used with temporal data, i.e., data that

changes its position as a function of time. Location has an associated reference system, which is explained in detail in Section 5.3. In figure 10 we show the structure of an instance of the **GeoCountry** class, and its relationships with both the **Country** and its **Location**.

In addition to decorators wrapping conceptual objects, purely geographic objects may also appear; they do not represent a view of any conceptual object but are characterized by their geographic relevance. In other words, these objects have the same spatial characteristics a Decorator has, but they are not related to any conceptual object. Based on the similarities between decorators that add spatial information to conceptual objects, and pure geographic objects it is possible to define an abstract class, which groups the common behavior of those ''purely'' geographic objects, plus those wrappers of the conceptual model. Figure 11 shows the schema that constitutes the basic building block to define the object-oriented architecture of GIS applications.

As it is shown in figure 11, the abstract class **AbstractGeoObject** defines the protocol to manipulate geographic functions (a point belonging to an area, perimeter, etc), to know the

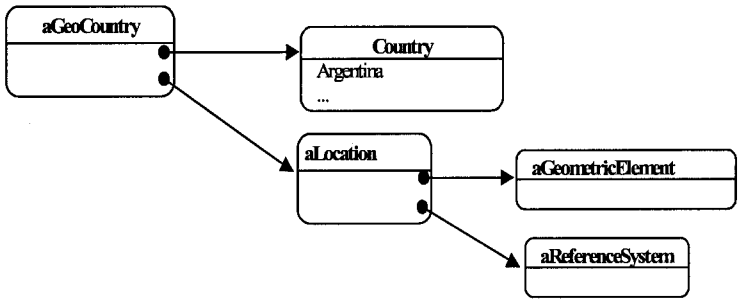


Figure 10. Relationships among Conceptual Objects, Geo-Decorators and Locations.

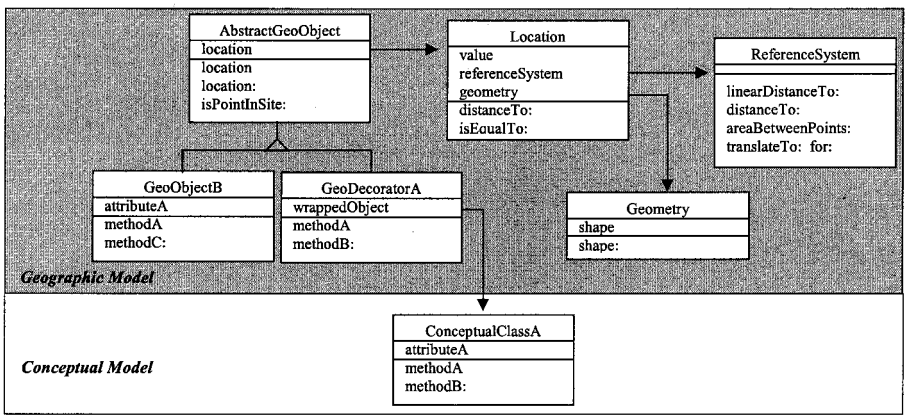


Figure 11. The basic schema to build geographic applications.

object locations and, through the **Location**, their shapes, given by the **Geometry** class and the reference system in use.

Geometry is the name of a class hierarchy defining the basic spatial elements, e.g., point, line and polygon. In this hierarchy all spatial operations are defined according to the kind of element we are manipulating. For example: one element defined as a point will be able to perform behavior like: ''do you intersect this line?'' or ''are you included in this polygon?'' and, in general, operations about adjacency, intersections, inclusion, etc.

In the following section we describe different kinds of composition relationships describing some spatial relationships.

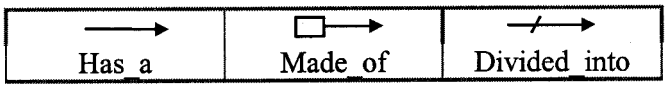
5.2. Using composite objects in the geographic model

Many geographic elements that appear in the real world are in essence aggregations of other objects. Consider for example a state that is divided into districts, which are in turn made of counties, while every county has one or more cities. According to the problem we are modeling, we find that different cases may have different restrictions about the composition semantics: sometimes, the existence of the whole depends on the existence of its parts; this restriction could be unsuitable in other cases. Defining and understanding the exact semantics of the composition is very important since it will allow us to establish proper design and implementation constraints.

We have defined three part-whole relationships that usually appear in GIS applications, figure 12 shows graphical examples of them. These relationships define restrictions about their cardinality, their geometry, and about the lifetime of all involved objects:

- **has_a**: A *has a* B. There exists b in B, a in A such that b is part of a. This relationship has no particular restrictions over A and B. Lifetimes of those are independent. Geometry of A and B could be different.
- **made_of**: A *made of* B. For every a in A, there exists b in B such that b is part of a. A and B share the same geometry.
  For example, consider that A is a range of mountains and B represents a mountain. It would not make sense that an instance in A exists without any instance in B. A is covered by all instances of B. Different instances of B may or may not be overlapped.
- **divided_into**: A *divided into* B. For every b in B, there exists a in A such that b is part of a. For example a country is divided into states and there are not states outside of a country. A is covered by all instances of B. There is not overlapping between A and B, and A and B share the same geometry.

To manage these kinds of composition relationships, we have defined an additional notation to OMT primitives [14], to distinguish each kind of composite:



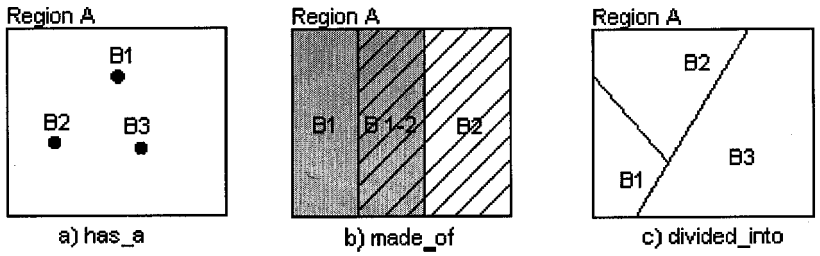


Figure 12. Examples of has_a, made_of and divided_into relationships.

The translation of those relationships among objects, which are defined in the Conceptual Model to the Geographic Model, is application-dependent; it is not sure that the composition semantics will be preserved.

In Sections 5.3, 5.4 and 5.5, we present some design patterns that can be used to solve recurrent problems appearing during the geographic modeling.

5.3. The reference system design pattern

A geographic object has a location referencing its geographic position in a particular moment. Locations are critical in geographic systems since they constitute the main characteristics in this domain; they are also one of the most difficult concepts to manipulate.

There are a great number of map projection techniques to transform spatial information extracted from the Earth's surface to a planar coordinate system [16]. The use of each one of them, however, depends on the data type we are manipulating, on the way in which the spatial data was acquired or even on the final user profiles.

The most common way to represent locations is the latitude/longitude system, but this is not always the more convenient way to specify them; for example the latitude/longitude system makes it difficult to calculate distances and areas.

Another usual reference system is the Universal Transverse Mercator. UTM produces a secant projection to the Earth surface, and divides it into zones of 6 degrees of longitude and 8 degrees of latitude providing a mechanism to locate areas in a coarse grid.

There are also many kinds of projections either based on the geometrical model of the projection or involving patterns of distortions in the map. Azimuthal, Conic and Cylindrical are examples of the former and Equivalents and Perspectives of the latter (see [16] for more details).

When we are dealing with geographic objects, we have to keep the information about the reference system being used. Moreover, when we need to combine information from many different objects, we must be sure that the same reference system is being used in all of them, otherwise some conversions will be needed. The same happens when the

reference system that we use to present information to the user has to be different from the reference system used in the input data.

The Reference System design pattern helps to define the context of a location. It provides a set of legal operations that conform to the corresponding reference system arithmetic (including time-related operations).

Consider an application in which we want to track cars or trucks carrying goods from one city to another one. We may have two kinds of positions: one explained in terms of latitude/longitude when a truck is traveling by a route, and the other one in terms of streets' names and numbers, when the truck is in a particular city. In this example we can see that not only we may have to describe what the location means, but also the way in which that location changes depending on the place where the truck is. From the implementation point of view, the application must provide operations to dynamically change from one reference system to other. From the design point of view, we need to express this design decision clearly and unambiguously.

Without the information that the reference system provides, the set of points representing the location of a geographic object does not make sense. Working with several reference systems is also complex since even when it is possible to implement algorithms to translate locations from one reference system to another one, the semantics of these changes are not explicitly recorded during design.

Figure 13 shows the structure of the Reference System design pattern.

Please notice that we have decoupled **Location** from **ReferenceSystem** to allow to dynamically configuring a location object with different reference systems during the lifetime of a geographic object. This design solution is clearly better than sub-classifying **Location** according to different reference systems, since the resulting design would be more rigid and would prevent changing the reference system dynamically.

Location implements the basic behavior to support the representation of the geographic coordinates within a reference system. **ReferenceSystem** defines an abstract protocol that

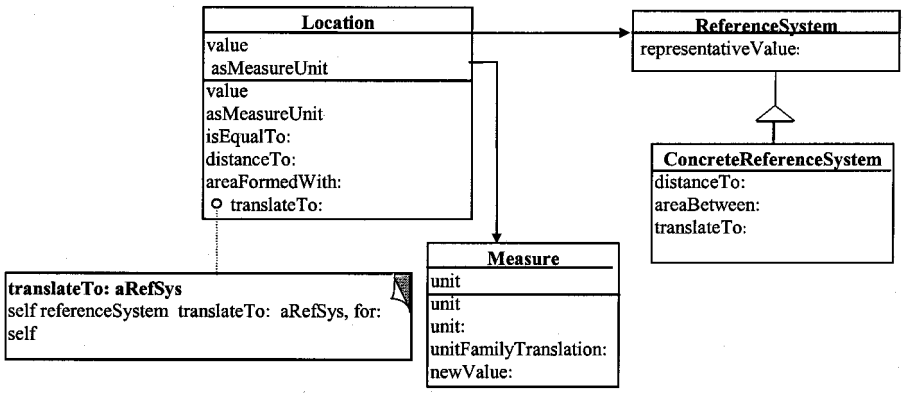


Figure 13. Structure of the Reference System.

is used to describe the context where a Location is defined. It also defines the set of legal operations in this context. In other words each ReferenceSystem describes how measures behave in the defined Location. **Measure** describes the units of the Location values. These values could be measured following the approach proposed in [5].

This architecture defines an object (location), which encapsulates the relationship between a measure and a reference system. **Location** knows the value of its position and the used reference system. This pattern provides a great flexibility since multiple ways to represent geo-referenced entities of the real world, based on different earth abstractions can be supported. Furthermore, it makes possible to change or translate locations from one system to another one without affecting the referenced object.

Each reference system implements a set of legal operations such as computing distances between points, comparing elements, and calculating areas. It also specifies translation operations to other reference systems. Distances, areas and any other operations that depend on the geographic object position, are expressed by collaborations between the location of that object and the location reference system.

There is also the problem of defining distances. The meaning of distance depends on the space. While the distance between two points in a three- dimensional space in spherical coordinates is the usual distance function

$$\sqrt{\sum_{k \in N} (x_{ik} - x_{jk})^2}$$

we may want to define distance only with respect to the length of the shortest segment that goes from the point to the sphere surface. Thus the definition of what is the center of the space may vary. For computing purposes we can think of the center either as a point or as the whole surface of the sphere.

To be able to implement different kinds of reference systems we associate an object representing the origin to the **ReferenceSystem** class. In this way we can define the origin as a point or as an equation without affecting other objects.

Figure 14, shows the structure of the Reference System where the origin is defined.

In order to calculate the distance between two points, the **ReferenceSystem** collaborates with its origin to obtain the information about its position values.

5.4. The roles design pattern

In this section we describe the **Roles** design pattern, which can be used in GIS applications. We define it in terms of the problem it solves, its structure and the advantages of its use.

Objects within a geographic model can handle different kinds of information, addressing completely different concerns. In the design step, all information related to an object is usually included in that object. Describing different unrelated themes in the same object is an unsuitable approach, as we show in the example below.

Suppose that we are modeling a country divided into states. We could define their

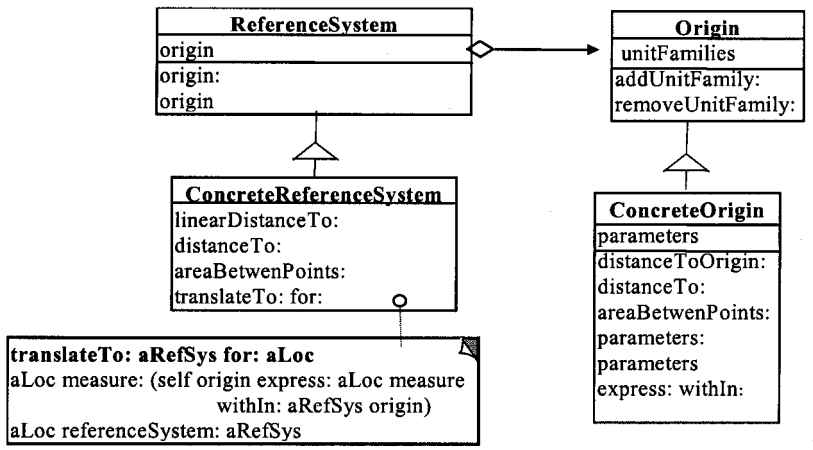


Figure 14. Structure of the Reference System including its origin.

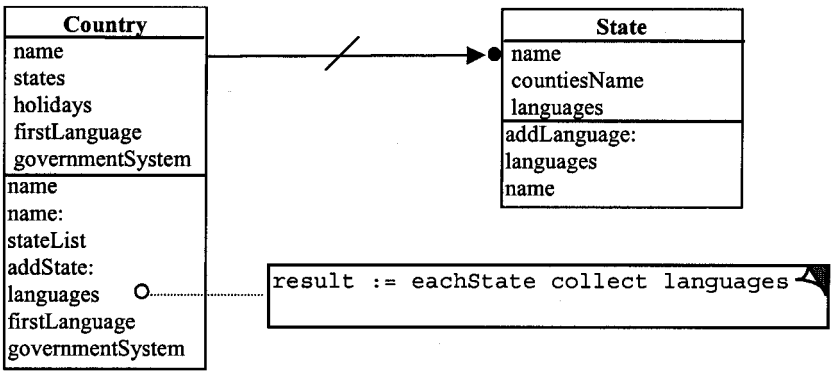


Figure 15. Structure of Country and State.

behavior as allowing to answer the country/state name, answer all states which conform the country, return the country national language, explain its government system, answer its independence day, list all national holidays, and so on. Figure 15 shows the basic structure of classes **Country** and **State**.

A country could also need to record additional information about Demography such as population, average longevity, active population, and average earn; or Sociological such as the name of the ethnic groups, regions in which each ethnic group lives, etc. Since different instances of Country could be analyzed from different points of view, each instance would need to exhibit different behaviors; for example an instance of Country can answer information about its ethnology, while another instance has information about its demographic population. If all information is included in the same class, we end with a monolithic class that will be difficult to maintain and extend.

Another inadequate solution could be defining subclasses of Country by trying to model each aspect (e.g., Demographic) as a sub-class. Figure 16 shows the resulting architecture, which does not allow the same country to be viewed from more than a point of view or even to dynamically change the point of view.

As it is shown in figure 16, when a **Country** is instantiated (**Demographic** or **Sociological**) it is quite difficult or impossible (for most of the OO languages) to change the instance class.

Actually, demographic and sociological information constitutes different points of view to see the same object. Moreover, under different views, the same information could be interpreted in different ways.

A better solution is achieved when each subject is defined as a role; each role will be an object, which has a close relationship with the main object. Figure 17 shows a simple diagram of the relationships among **GeoCountry** and its roles as they are expressed during the design step.

The relationship between a geographic class and each one of its roles is implemented with a specific **Roles** subclass.

Each role acts as a wrapper over the object that plays that role, due to its polymorphic interface. This architecture allows an object to dynamically change some of its roles and to interact with them to perform specific computations.

Country roles could need to collaborate with the country states in order to compute a geographic aspect. In the case of states with a Demographic Role, that provides information about population in that state, a country population could then be computed by summing up all states' populations. This means that the Demographic Role of the country

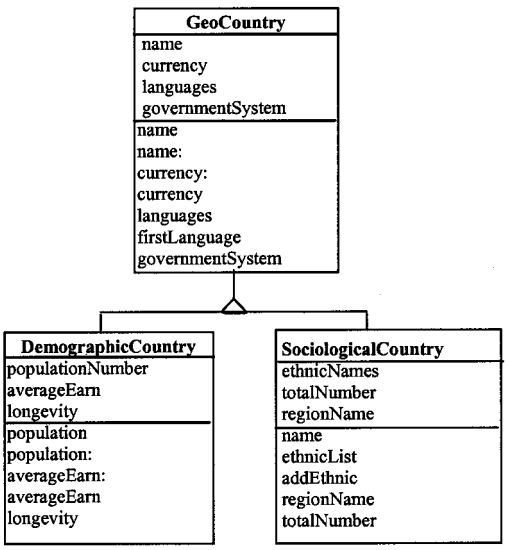


Figure 16. Resulting architecture with Country subclasses.

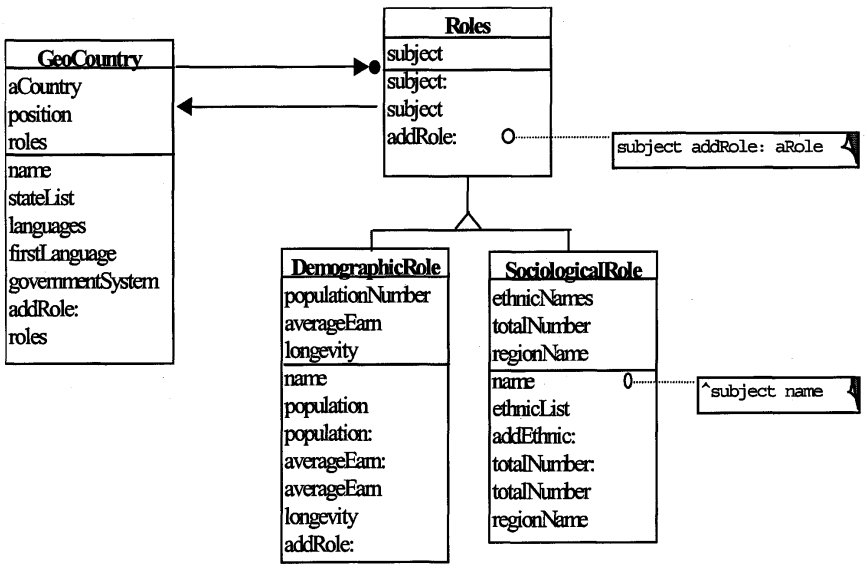


Figure 17. Relationship between the Country and the Roles hierarchy.

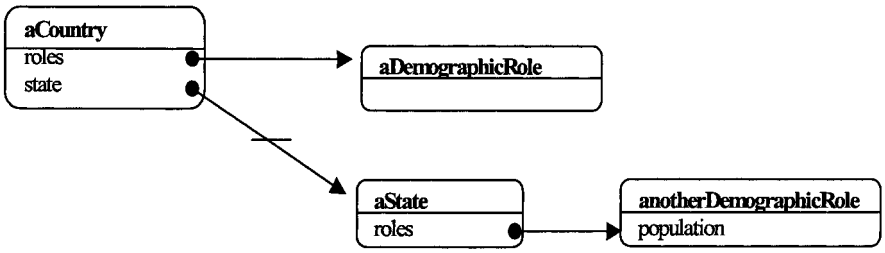


Figure 18. Relationships among instances of Country, DemographicRole and State.

must know how to collaborate with the demographic role of each state, in order to get the partial populations and to compute the final result.

While the **Demographic Role** applied over a **State** is atomic, since it handles concrete data, the **Demographic Role** of the **Country** is derived because it must collaborate with others in order to calculate population. Figure 18 presents an object diagram, which shows the relationships among a country, its states and the demographic role.

Since any role could be characterized as derived or atomic, it is necessary to abstract this condition into a new hierarchy, which models different strategies that may be used to obtain the desired information. We can do that by defining a **Strategy** hierarchy as it is shown in figure 19.

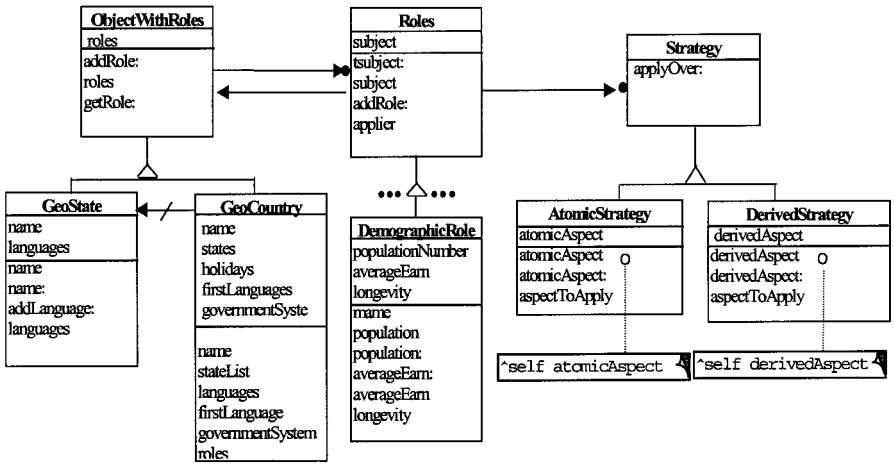


Figure 19. Resulting structure with ObjectWithRoles, Roles and Strategy hierarchies.

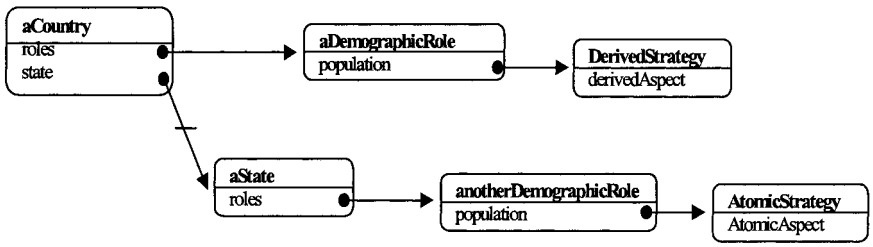


Figure 20. Relationships among instances of Country, State, DemographicRole, AtomicStrategy and DerivedStrategy.

In the example above, when a query for information about population is received by an instance of GeoCountry, it will satisfy the request by collaborating with all its states. The relationship among State, **DemographicRole**, and Country can be modeled by creating a new object, the **Strategy**, which represents the strategy used to answer or calculate the population. When an object corresponding to the demographic role of the country is created, the specific **Strategy** that will be used for a particular calculation, have to be associated with that role. Figure 20 shows the resulting relationships among instances of **Country**, **State**, **DemographicRole**, **AtomicStrategy** and **DerivedStrategy**.

While an instance of **AtomicStrategy** specifies the way in which a state answers demographic requirements, an instance of **DerivedStrategy** models and implements the interaction between a country and a state in order to elicit the data (i.e., the population). In the end, the instance of **DemographicRole** is the responsible for the final calculations. The role abstraction is well known in other areas such as: security models [18], financial applications, etc.; within the GIS area, it makes possible to dynamically assign different

views to a geographic object. Finally, it allows GIS designers to take advantage of the '*is divided into*' relationship; it is possible to build views which are based on components' knowledge and behavior as we have shown in the country and states example.

5.5. The appearance design pattern

All geographic information systems produce graphical outputs; these outputs have to satisfy a wide range of user requirements, which in many cases implies displaying the same entity in different ways according to the actual context; or showing different objects have to be in a similar fashion. For example, transportation maps of a city usually deal with subway lines, pedestrian path, airports, etc. In this context, there are a lot of symbols, which aid to explain different features of the city [12]. A tiny airplane could indicate a regional airport while an image of a commercial jet represents an international airport. In order to locate subway stations, it is common to use points with different colors or shapes. Additionally, the intersections of subway lines are marked by combining the color and the shape of the representing points, figure 21 shows some examples of different symbols which usually appear in city maps.

Geographic information is generally used for both performing spatial analysis and producing outputs. Symbols, colors and the desired level of abstraction for a visual analysis are defined according to the needs of the users. Usually, some tradeoffs must be made between the amount of data and the accuracy of information in order to obtain a legible map in the output [2]. The selection of the output properties (symbols, colors, scale, etc.) is done at the moment in which the map is shown. In fact, spatial analysis and information outputs are very different concerns; they have to be dealt by separating the definition of the geographic object from its presentation in the interface.

A first attempt to design the user interface of a GIS would consider assigning the representation responsibilities to the **GeoObject**; then **GeoObject** will have to implement methods to manage visual properties. This solution does not address the dynamic essence

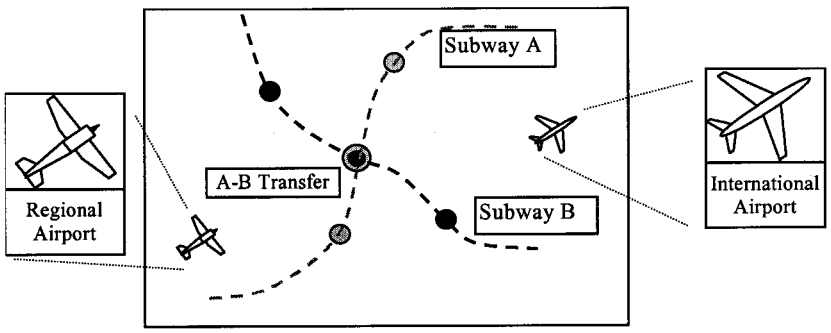


Figure 21. Different interface symbols for similar objects.

of the problem. Some entities will need to handle more than one interface feature, which perhaps will have to be shown at the same time. As it is shown in figure 21, the station, which is located in the intersection of Subways A and B, is shown as the composition of symbols of both lines.

The most effective way to relate objects with their graphical representations is the Model/View schema, described as the Observer pattern in [6]. This solution implies modeling the representation itself as an object. In our model this object is called Appearance and it knows how to display the corresponding **GeoObject** according to the scale and the selected shape. It also manages visual properties such as color, definition level, and line thickness. For many objects, these properties will have a default value and will not change.

Given the large number of objects than can appear in any GIS system, this pattern is useful to separate the properties from the object displaying those properties. All objects that have the default appearance will share the same **AppProperties** object. Only those objects with a distinctive appearance will have their own properties.

In figure 22 the structure of the Appearance design pattern is shown.

The **Appearance** class specifies the protocol to receive notification of changes from the related object. The **AppProperties** class defines the set of properties that establishes how the Shape is displayed by the Appearance object. The **ConcreteAppearance** class implements the protocol of **Appearance** to display **ConcreteGeoObjects** according to a set of properties (modeled by AppProperties). Finally, each **Shape**'s instance is an arbitrary object whose protocol is known by the **ConcreteAppearance** object.

All geographic objects that are going to be displayed have one or more associated **Appearance** objects, that are recorded by the **GeoObject** by calling the *attach*: method. An **Appearance** object knows a **GeoObject** instance that works as its model when its *assign*: method is called.

Every time the **ConcreteGeoObject** changes its state, it calls its *update*: method, which in turn calls the *notify*: method of every **Appearance** object associated to the **GeoObject**. The *showOn*: method of a **ConcreteAppearance** object is tailored to display the **Shape** object as the representation of the **GeoObject**, taking into account the appearance modifiers stored in the properties object.

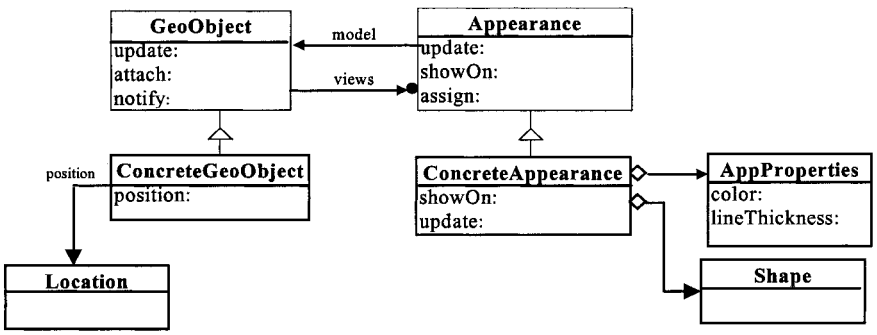


Figure 22. The Appearance design pattern.

Using the **Appearance** design pattern we can separate the representation from the geometry of an object; we also reduce storage by avoiding repeated information in different objects and allow many views of the same **GeoObject** to co-exist seamlessly.

AppProperties and **Shape** can be shared; as stated above, many objects can share a common shape that will be displayed in the same way.

Different **Appearance** instances are needed for different shapes, even for the same **GeoObject**. There are three kinds of relationships between a **GeoObject** and its appearance. It can be a literal representation of the **GeoObejct** position; it can be a shape derived from the position; or it can be an arbitrarily assigned shape. Then, an Appearance object must know the protocol of the **GeoObject** and the **Shape** in order to display it in a specific media.

6. Conclusions

In this paper we have presented a two-step method for designing GIS applications. The goal of the process is to deal with a different concern in each step. In the Conceptual Modeling step we concentrate our efforts on the identification and abstract definition of domain elements, thus allowing us a better understanding of the problem. As the definition of spatial features is delayed until the geographic design step, this approach leads to more modular and understandable designs. We have presented a basic architecture for building the Geographic Model; it is based on the use of the Decorator design pattern to decouple conceptual from geographic objects. As GeoObjects wrap conceptual objects it is possible to create more than a geographic view for the same object. We discussed the Reference System design pattern to address the problem of dealing with multiple ways of understanding the location of an object. The Appearance design pattern allows defining more than one user interface for the same geographic object. We showed that the interaction among these three simple patterns yield an evolvable and easy to modify and reuse object structures.

We then presented a new design pattern, Role, yielding an elegant and powerful solution to recurrent problems in the GIS domain. We also showed how the basic architecture could be easily combined with this micro-architecture in a more comprehensive example.

Besides, we are building an object-oriented application framework providing the basic hierarchies and abstract collaboration models that will allow us to extend an existing application with geographic features by systematically applying previously mentioned design patterns. We intend to define an environment in which the designer will be able to visually compose application and geographic objects by following those design patterns.

Different reference systems have been implemented by instantiating the ***Reference System*** pattern. In particular we have implemented Rectangular (2-D and 3-D) and Ellipsoidal (2-D and 3-D) systems including cylindrical and conic projections, using an object-oriented language.

Interface aspects of geographic objects (instantiations of the ***Appearance*** pattern) and the abstract object collaborations needed to interpret locations in the model have been also implemented.

We are now working on the definition of topological features of geographic objects. In this way, each object in the model knows its main spatial characteristics: the location, the reference system in which the location has been defined and its topology. Topologies define geographical operations, such as distances, neighborhood, etc.

From our experience in modeling geographic applications using patterns and from the feedback obtained from prototypical implementations, we believe that our approach is quite promising for obtaining interoperable GIS applications, either built from scratch or by using existing legacy applications.

References

1. C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. A Pattern Language. Oxford University Press: New York, 1977.
2. S. Aronoff. Geographic Information Systems: A Management Perspective. WDL Publications: Ottawa, Canada, 1989.
3. A. Dallari Bonfatti and P.D. Monari. ''Capturing more knowledge for the design of geological information systems,'' in *Proc of ACM-GIS'95*, Baltimore, Maryland, USA, 1–7, 1995.
4. Coplien and Schmidt (Eds.). Pattern Languages of Program Design. Addison Wesley, 1995.
5. M. Fowler. Analysis Patterns: Reusable Object Models. Addison Wesley, 1997.
6. R. Helm Gamma, R. Johnson, and J. Vlissides. Design Patterns. Elements of reusable Object-Oriented Software. Addison Wesley, 1995.
7. S. Gordillo, F. Balaguer, and F. Das Neves. ''Generating the architecture of GIS applications with design patterns,'' in *Proc. of the ACM-GIS'97: Advances in Geographic Information Systems*, Las Vegas, USA, November 13–14, 1997.
8. S. Gordillo and F. Balaguer. ''Refining an object-oriented GIS design model: Topologies and field data,'' in *Proc. of the ACM-GIS'98: Advances in Geographic Information Systems*, Washington, USA, November 6–7, 1998.
9. B.U. Pagel Kosters and H.W. Six. ''Object-Oriented requirements engineering for GIS applications,'' in *Proc. of the ACM 3rd ACM International Workshop on Advances in Geographic Information Systems, ACM-GIS'95,* Baltimore, Maryland, USA, 61–68, 1995.
10. M.A. Casanova Medeiros and G. Camara. ''The Domus project. Building an OODB GIS for environmental control,'' in *Proc. of IGIS'94, International Workshop on Advanced Research in GIS*, Springer Verlag LNCS, N. 884, 45–54, 1994.
11. Open GIS Consortium (OGC). 1996B, *The Open GIS Guide-A Guide to Interoperable Geo- processing*, Available at http ://ogis.org/guide/guide1.htm
12. G. Plumb. ''Cartography and map design workshop book,'' Urban and Regional Information System Association 1460 Renaissance Dr.; Suit 305 Park Ridge, IL 60068, July 1997.
13. M. Postmesil. ''Maps alive: viewing geospatial information on the WWW,'' in *Proc. of the six International World Wide Web Conference, 1997*, Available at http ://www6.nttlabs.com/Hypernews/get/PAPER130.htm.
14. M. Rumbaugh, M. Blaha, M. Premerlani, and W. Lorensen. Object-Oriented Modeling and Design. Prentice Hall: Englewoods Cliff, New Jersey, 1991.
15. D. Schmidt. ''Using design patterns to develop reusable object-oriented communication software,'' *Comm. of the ACM*, 65–74, October 1995.
16. J. Star and J. Estes. Geographic Information Systems. An Introduction. Prentice Hall, 1990.
17. N. Tryfona and T. Hadzilacos. ''Geographic applications development: models and tools for the abstract level,'' in *Proc. of ACM-GIS'95*, Baltimore, Maryland, USA, 19–28, 1995.
18. J. Yoder and J. Barcalow. ''Application security,'' in *Proc. of Pattern Languages of Programming*, Vol. 2: Roles and Analysis. Urbana-Champaign, Monticello, Illinois, USA, 1997.

Silvia Gordíllo received an M.Sc. in Software Engineering at La Plata University in Argentina. She is a full Professor at the same University and member of LIFIA (Laboratorio de Investigación y Formación en Informática Avanzada). Her areas of work include Object Oriented Databases and Geographic Information Systems. She has presented results of research projects at many Computer Science Conferences.

Catalina Mostaccio is an associate Professor at La Plata University in Argentina. She is completing her M.Sc. in Software Engineering at the same University. She is a member of LIFIA (Laboratorio de Investigación y Formación en Informática Avanzada). Her areas of work include Geographic Information Systems. She has presented results of research projects at many Computer Science Conferences.

Federico Balaguer is member of LIFIA-University of La Plata-, since 1994. He has experience using Geographical Information Systems for supporting cadastral and water management projects. He has consulted for organizations in different fields, as ''object expert'' and gave several courses about Object Technology. His interests include GIS evolution, frameworks and design patterns. Currently he is enrolled in the Ph.D. program in the University of Illinois at Urbana-Champaign.

Fernando Das Neves got his Licenciate degree at the National University of La Plata in Argentina. He worked there in 1996 and 1997 with Prof. Gorillo to develop an object oriented design model for GIS. He is currently in the MS program at Virginia Polytechnic Institute and State University. His interest areas include object-oriented design, information visualization and retrieval, and digital libraries.