Isolating the Concern "Time" in Geographic Information Systems

Lucas Hahn, Arturo Zambrano, and Silvia Gordillo

LIFIA, Facultad de Informática, Universidad Nacional de La Plata
50 y 115 1er Piso
1900 La Plata, Argentina
{lhahn,arturo,gordillo}@lifia.info.unlp.edu.ar

Abstract. Time is an important feature for Geographic Information Systems (GIS). Unfortunately it is difficult to achieve a modularized model of the concern "*time*", because it impacts on several entities of the system. In this work we analyze the effects of including this concern in a GIS for continuous fields considering two different approaches. The first one is based on pure object orientation, while the second one uses aspect orientation to improve modularization. This work, also shows that it is feasible to do the modularization of a non-trivial functional concern, such as time, in the complex domain of geographic applications.

Key words: crosscutting concern, aspect-oriented programming, GIS, continuous field, time

1 Introduction

A field in the context of a GIS system, is an abstract concept used to describe several physical phenomena: electromagnetism, gravity, electrodynamics and other relativist mechanics. Fields are utilized to understand complex physical laws, which can be expressed in an accurate way by differential equations. A field is a physical space-distributed amount, which takes a different value on each space point. The set of all possible values taken by the field is called *image*. The set of space points where phenomena can be measured is named *domain*. The domain of a continuous field is a geographical area, defined by polygon-shaped positions.

A great variety of applications exists in which the required information are fields; for example any weather-related application must have the ability to analyze continuous phenomena, like temperature, humidity, and precipitations, among others. Surface and pollution are also modeled as continuous fields in environmental research. Due to the continuous nature of fields it is impossible to store the values corresponding to each space-point since they are infinite. The general approach consists of sampling the phenomenon in some selected points and then interpolating the value of the phenomenon for unsampled points. The sampled points are organized in different ways according to the interpolation algorithm. For example, if the interpolation method is linear, for a 3D space, a triangle will be needed so as to define the plane used for the interpolation. A

plane is defined by 3 points, so that the sample points are organized in a triangulated irregular network (TIN). There are other possibilities for storing sampled data, such as regular and irregular grids, tessellations, etc. These structures are known as spatial data models [3,4]. The algorithm used to calculate the value of the phenomenon at unsampled points is named *estimation* or *interpolation method*.

In many situations it is interesting to derive new fields from previous ones. This task is done by applying operations on existing fields. A trivial example of this might be obtaining the annual average precipitations in a sample field, from a set of monthly precipitation fields. Other operations on scalar fields are: sum, subtraction and multiplication. For vector fields (used to model phenomena like wind) vector and scalar products are also valid operations.

Time is an important and usefull feature of GIS. The challenge is to introduce the concept of time preserving modularization and encapsulation. These characteristics provide for flexibility for futures changes. Phenomena represented by fields use the concept of time to analyze their evolution and history. Some examples of this type of phenomena are: hurricanes, twisters, floods, etc.

GIS deal with complex and multiple concerns like location management, time, topological operations, etc. Zipf et al. [10] claims that the use of Aspect-Oriented Programming (AOP) might bring higher modularization levels to GIS. As many of the GIS concerns are crosscutting, AOP can be useful for tackling such complexity. Zipf proposes the use of AOP for modularization of features such as: (non spatial) business applications, 2D and height, 2D or 3D and time, geometry and taxonomy, GIS application and context awareness or personalization.

This work analyzes the effects of including the concern "*time*" into a GIS. We consider three different alternatives analysing advantages and disadvantages of each one. Two of them follow a pure object-oriented approach and the last one uses aspect orientation for improved modularization.

The results presented in this paper are applicable to almost any object oriented GIS. In order to illustrate our approach, we took a particular system which we will use as an example.

The paper is organized as follows: in section 2 we describe *continuous fields* in the context of GIS. Section 3 presents an object-oriented extension for time and their consequences. Section 4 introduces the concepts of AOP and presents the aspect-oriented extension. Finally, section 5 compares both approaches and concludes this paper.

2 A Model for Continuous Fields

In Zambrano et al. [9], it is presented Tilcara, an object-oriented application which enables us to define and manipulate continuous fields. Tilcara's capabilities include selection of fields' areas according to certain condition and execution of arithmetical operations involving several fields. By using such capabilities it is possible to obtain new fields derived from others.

The original architecture of Tilcara allows representing continuous fields in several ways. In order to model the different representations that a continuous field can adopt in this architecture, the Bridge [1] pattern is used. Therefore, there is a separation between the sampled data and the general information about the field. A continuous field can also alter the algorithm for calculation of phenomenon's values at unsampled points (known as estimation method). In this case the Strategy [1] design pattern was applied. Also, the original architecture has an object (an evaluator of operations) that is responsible for coordinating how the operations on fields are performed. So, this "evaluator" is responsible for: establishing the resulting continuous field's domain, defining the resulting continuous field's sample, selecting a representation type, establishing the estimation method according to the previous components.

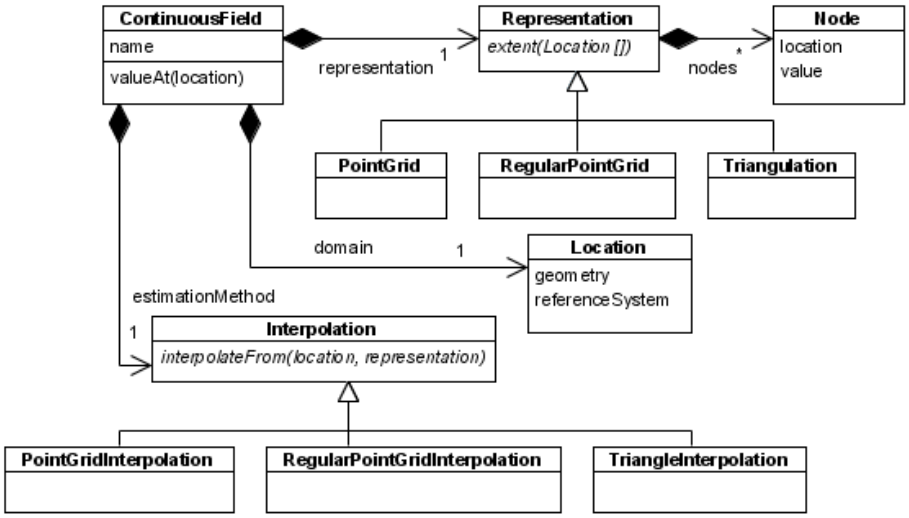

Fig. 1. Class diagram illustrating the design of Tilcara

Figure 1 depicts the design of Tilcara, which enables us to define continuous fields using different spatial data models (`Representation` class hierarchy) and estimation methods (`Interpolation` class hierarchy).

3 Extending Continuous Fields with Time Using OO

Adding the concern "*time*" to an existing application involves modifying the application's design; then its implementation must be altered to reflect this new design. Obviously, this is an error prone operation that involves the possibility of

breaking the code which was previously working properly. As we will see in this section, including this concern by using only object orientation implies making changes in several classes of the application.

In order to narrow the risk of introducing new bugs in the application, it is desirable to add extensions in a non intrusive manner. At the same time, it is important for the new extension to be as modular as possible, so that it can be easily included or removed from the application.

In this section we present two possible object-oriented designs for extending Tilcara. One of them is inspired by an existing work by Won Kim [8] and involves embedding time-related behavior in continuous fields; whereas the latter is good for contrasting the place where time behavior is located, since it proposes placing time behavior in Wrappers [1].

3.1 Alternative 1. Embedding Time into the Structure of Continuous Field

Modeling the state of objects over time through versions, encapsulating version management behavior in a class, is proposed in [8]. Following this approach, the original model of Tilcara evolves as depicted in Fig. 2. The new class, responsible for version management is called `NodeVersionManager`.

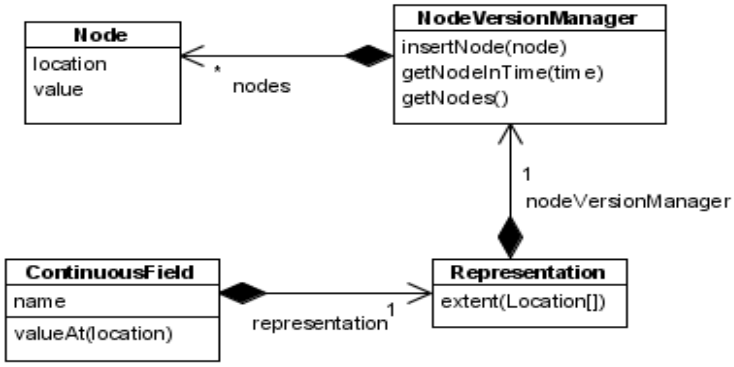

Fig. 2. Class diagram for internal representation of time.

Figure 2 shows the design, where `NodeVersionManager` sits between `Representation` and `Node`. In this way, the representation delegates on the new class the responsibility of organizing and retrieving the nodes according to the desired version. The presented design leads to the following consequences:

- There is one field for each phenomenon. The field uses different sets of nodes according to the time set; this can be done by instructing `NodeVersionManager` so that the representation uses the proper set of nodes.

– Manipulation of the subsets of nodes makes it unnecessarily difficult to generate new fields.
– Extensive changes must be done in existing operations (those that create new derived fields).

3.2 Alternative 2. Managing Time Externally to Continuous Fields

An alternative to the previous subsection idea is to place *time*-related behavior outside the structure of the continuous field. As shown in Fig. 3, the class `TemporalField` represents temporal continuous field. This kind of field can be seen as a Decorator [1], because it adds a new responsibility (handling time) to the existing `ContinuousField` class. On the other hand, it can also be interpreted as Composite [1], because many single fields can compose a temporal one.

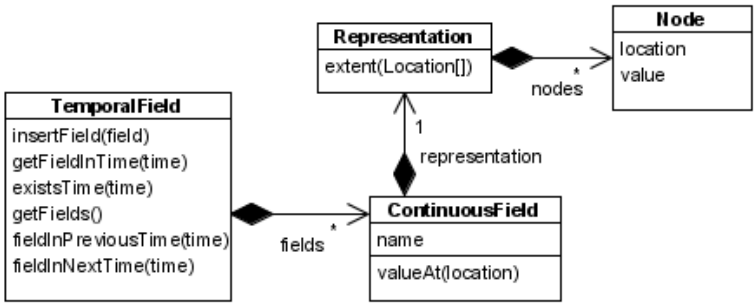

Fig. 3. Class diagram for external representation of time

This alternative design involves the following consequences:

– There is a new class that represents fields with time, without altering the original structure of the field.
– The use of Decorators can cause problems of object identity [5]. A decorator acts as a transparent enclosure but, from an object identity point of view, a decorated component is not identical to the component itself.
– If the time for all simple continuous fields is managed in the container temporal continuous field, it is impossible to know the timestamp for an isolated simple continuous field.

3.3 Impact and Discussion

The alternatives for representing time in continuous fields using just the mechanisms provided by object orientation involve substantial changes in both the design and the code. Thus, the concern "*time*" implementation is scattered in

the code of the modules (representation, nodes, continuous fields, etc), making it harder to comprehend and evolve. Fig. 4 shows classes affected by the inclusion of the concern "*time*". From this, we conclude that *time* is a crosscutting concern.

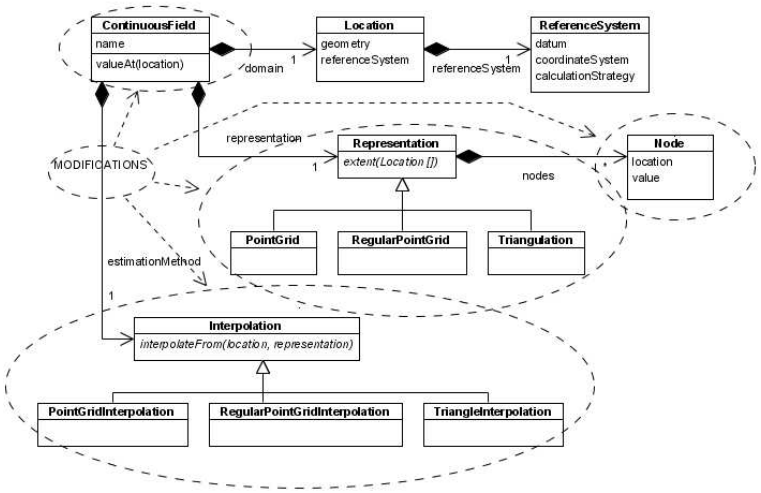

Fig. 4. Impact of the object-oriented based extension (dotted lines indicate the affected classes).

We found the second alternative as a more appealing design, but it still involves changing code in many existing classes.

The two alternatives we have shown are instances of an inherent limitation which is present in any imperative paradigm, where calls (method invocations, procedure calls, etc) need to be done in order to perform some computation.

4 Extending Continuous Fields with Time Using AOP

4.1 Crosscutting Concerns and Aspect-Oriented Programming

In the application development process, it is common to find a set of concerns that affect many objects beyond their classes constituting (in object-oriented programming) the natural units to define functionality. They are called crosscutting concerns. A crosscutting concern is one that is spread along many of the modules of a system. Typical crosscutting concerns are persistence, synchronization, error handling, and, in our case, time. As it is said in [2]: *"...existing software formalisms support separation of concerns only along a* ***predominant dimension*** *neglecting other dimensions... with negative effects on re-usability, locality of changes, understandability..."*. These secondary dimensions correspond to

crosscutting concerns. In our case, a new dimension is added when the application must be extended with time.

AOP [7] is one of many approaches resulting from the effort to modularize crosscutting concerns. The goal of AOP is to decouple those concerns, so that the system's modules can be easily maintained, evolved and seamlessly integrated. To do that AOP introduces a set of concepts:

Joinpoint: is a well-defined point in the program flow (for instance a method call, an access to a variable, etc.).
Pointcut: a set of joinpoints, and data associated to them.
Advice defines code that can be executed when a joinpoint is hit.
Intertype declarations: changes to the structure of an existing program.

The program whose behavior is affected by aspects is called *base program* (in this case it is *Tilcara*). A *joinpoint* specifies a point in the execution of the base program that will be affected by an aspect. One or more of these joinpoints (from one or different classes) are matched pointcuts in the aspect, which may have associated advices. In this way, when one joinpoint, referred in a pointcut, is reached in the program execution, the additional code, defined in the proper advice is executed. The code of the aspect is composed of advices and the pointcuts where those advices must be applied.

4.2 Including Time Using Aspects

As we have shown, it is not possible to achieve a well modularized extension for *time* in Tilcara. Hence, it seems reasonable to apply AOP [7] to accomplish a modular extension. In this section we present an extension based on a combination of aspects and objects. It extends the Tilcara's object-oriented model with a new kind of continuous field capable of managing the notion of time. Aspects are used to inject the behavior and structure for handling *time* in the application. As Tilcara was originally developed in Java, we have chosen AspectJ [6] as the aspect language for our prototype implementation.

The class `TemporalContinuousField` (Fig. 5) models a continuous field defined over a period of time. It keeps an instance of `ContinuousField` for each point in time when the field is sampled.

In order to keep the existing code working with original continuous fields implementation and with our new temporal field, we have introduced an interface (`IContinuousField`), which is implemented by both kinds of fields.

The time when a continuous field was created is considered relevant information for such field. For this reason it was decided to model the time as an attribute in the class `ContinuousField`. To avoid altering original source code, we have introduced the declaration of the new attribute using AspectJ's introductions [6]. Introductions allow static crosscutting, i.e. to add attributes or methods within existing classes.

When a message is sent to a field, it should react according to its state at a particular point in time. This functionality can be added by using the

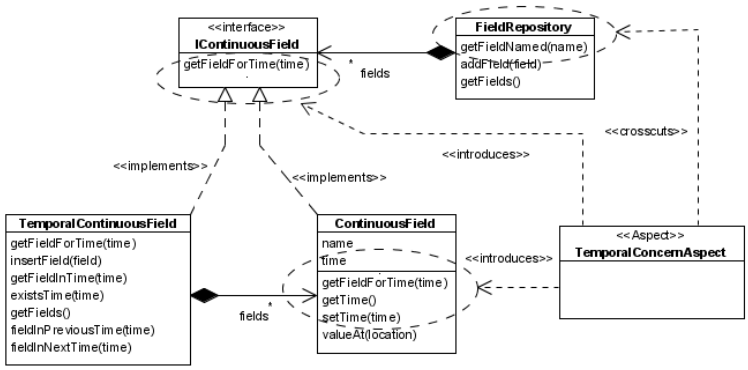

Fig. 5. Class diagram of the aspect based extension.

```
1  public aspect TemporalConcernAspect {
2     pointcut retrieveField(String s)
3        : call (* FieldRepository.getFieldNamed(String))
4           && this(Tilcara) && args(s);
5
6     IContinuousField around(String s): retrieveField(s)  {
7       IContinuousField  field = proceed(s);
8       return field.getFieldFor(TemporalAspect.getTime());
9     }
10    ...
```

Listing 1.1. Pointcut and advice for obtaining continuous fields.

mechanisms for dynamic crosscutting provided by AspectJ, i.e. the execution of advices when (before, after, around) a pointcut is matched. In this way it is possible to add extra logic corresponding to the concern of the time, for instance returning the proper field or value for a given timestamp. Listing 1.1 shows part of the aspect in charge of adding time to the process of field retrieval:

- From line 2 to line 4 the pointcut `retrieveField` is declared, which matches the calls to method `getFieldNamed(String)` of the class `FieldRepository` (line 3). Line 4 shows captures of the arguments for the method call.
- Lines 6 to 9 defines an around advice, which means that the code in lines 7 to 8 will be executed instead of the original method.
- Line 7 does the actual call to the adviced method. The return value is stored in a local variable (`field`). Then in line 8 the advice returns the result of calling `getFieldFor(time)` on the original retrieved field.

The method `getFieldFor(Date)` is the one which allows to retrieve a field for a timestamp. This method makes sense for the class `TemporalContinuousField`

```
1   ...
2   public Date ContinuousField.time;
3   public ContinuousField
4               ContinuousField.getFieldFor (Date time){
5                       if (this.getTime().equals(time))
6                                       return this;
7                               else return null;}
8   public abstract ContinuousField
9               IContinuousField.getFieldFor (Date time);
10  public Date ContinuousField.getTime() {return time;}
11  public void ContinuousField.setTime(Date t) {time = t;}
12  ...
```

Listing 1.2. Introductions for time.

but, for class `ContinuousField` it only returns the field itself (if appropriate). To reuse of the rest of Tilcara it is necessary for both continuous fields classes to be polymorphic. To minimize the impact on the existing code, it was decided to introduce it using an intertype declaration. The method was also introduced in the interface `IContinuousField`. The code listing 1.2 shows such introductions:

- In line 2 the attribute that represents time in the field is introduced.
- From line 3 to 7 the method `getFieldFor(time)` is introduced in the class `ContinuousField`.
- From line 8 to 9 the message `getFieldFor(time)` is introduced in the interface `IContinuousField`.
- From line 10 to line 11 the getter and setter of the attribute are introduced.

When the system needs to perform an operation on the field, it sends the message `getFieldNamed(name)`, which seeks the desired field. The message is captured by the aspect (`TemporalConcernAspect`), which carries the search operation out in the repository system and then send the message `getFieldFor(time)` to the field. The message `getFieldFor(time)` returns the resulting field.

5 Conclusions

As shown in Section 3.3 an extension based exclusively on the use of object orientation would have required changes in many of the existing classes. This would involve a mixture of original and new code, with the consequent danger of spoiling code. In addition, the *time*-related behavior would have been spread in many classes.

We have used both object and aspect orientation to obtain a modular design and implementation of the concern "*time*". On the one hand, object orientation was used to implement the temporal continuous field concept, encapsulating internal state and behavior. On the other hand, aspect orientation was used

to introduce the new behavior in an existing application. Aspects have been used to declare new instance variables, inheritance relationships and methods on existing classes. They have also been used to introduce new behavior with around advices.

Some of the benefits of the presented approach are:

- The original code of the application remains untouched and fully functional.
- Implementation of the concern "*time*" is not tangled with the rest of the application; so changes for maintenance or evolution can be safely performed.
- The new feature can be easily removed from the application, it is enough not to weave the new aspects.

Another contribution of this work is the application of aspect orientation to a functional concern. Many times aspect orientation is underestimated by arguing that it is useful only for non-functional concerns, such as logging, persistence, security, etc. In this case, we have shown how a functional non-trivial concern in the domain of GIS can be properly modularized by using aspect orientation.

References

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
2. S. Herrmann and M. Mezini. PIROL: A case study for multidimensional separation of concerns in software engineering environments. In *OOPSLA*, pages 188–207, 2000.
3. K. Kemp. Fields as a framework for integrating gis and environmental process models: Representing spatial continuity. *Transactions in GIS*, 1(3), 1997.
4. K. Kemp. Fields as a framework for integrating gis and environmental process models: Specifying field variables. *Transactions in GIS*, 1(3), 1997.
5. S. Khoshafian and G. P. Copeland. Object identity. In *OOPSLA*, pages 406–416, 1986.
6. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.
7. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.
8. W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, 1990.
9. A. Zambrano, L. Polasek, and S. Gordillo. Tilcara: An oo perspective to handle continuous fields in gis. In *Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 185–186, 2000.
10. A. Zipf and M. Merdes. Is aspect-orientation a new paradigm for gis development? -on the relationship of geoobjects, aspects and ontologies. *AGILE 2003*, 2003.