Proceedings of the 6th AGILE
April 24th-26th, 2003 – Lyon, France

IS ASPECT-ORIENTATION A NEW PARADIGM FOR GIS DEVELOPMENT?

ON THE RELATIONSHIP OF GEOOBJECTS, ASPECTS AND ONTOLOGIES

Alexander Zipf1, Matthias Merdes

European Media Laboratory, EML, Heidelberg, Germany.

<firstname.lastname>@eml.org

1. A MOTIVATION FOR ASPECT-ORIENTATED PROGRAMMING

Egenhofer and Frank [15] state that multiple inheritance is essential for object-oriented (OO) GIS modeling. Fonseca *et al*. [18] bypassed the associated problems by defining ontologies that consist of concepts with different "roles". But within GIS application development it is still often necessary to choose from these possibilities to construct actual classes. While the need for multiple inheritance has been discussed for a long time in the OO community, only now – years later – a new paradigm arises on the horizon, that has the potential to solve some of the inherent problems. This paradigm is aspect-oriented programming (AOP), or, more generally, AOSD – Aspect-Oriented Software Development [1] or AOSE – Aspect-Oriented Software Engineering. It allows the separation of multi-dimensional concerns within an application into individual aspects that have their own well-defined responsibilities, but can be weaved together at compile or run-time to generate more sophisticated applications. We will investigate and discuss the different ways of how AOP can contribute to GIS modeling and development systematically within this paper.

All of the classes defining an OO software system have well-defined responsibilities. There are however parts that cannot be viewed as being the responsibility of only one class, they crosscut the complete system and affect parts of many classes. Of course, the code that handles these parts can be added to each class separately, but that would violate the principle that each class has well-defined responsibilities. This is where AOP comes into play: AOP defines a new construct, called an aspect, which is used to capture crosscutting concerns of a software system in separate program entities. The application classes keep their well-defined responsibilities. Additionally, each aspect captures crosscutting behavior.

The article is divided into the following parts: After an introduction to relevant work in the area of GIS, we explain the basic concepts of AOP2. Then we provide examples of how this can be adopted in GIS development as well as geoobject- and domain modeling. In particular the latter one opens new possibilities. We have detected similar features in the area of ontology-based domain modeling and AOP. These can be exploited when used together. Therefore we propose research on the generation of aspects and geographical ontologies for the development of working GIS applications. We will present an elaborate example illustrating the concepts and benefits of AOP for GIS using Java and AspectJ.

---

1 Now with the University of Applied Sciences, Dep. of Geoinformatics & Surveying, Mainz, Germany. zipf@geoinform.fh-mainz.de

2 A note for clarity: AOP = "Aspect-oriented Programming" should not be mixed up with the term AOP for "Agent-oriented Programming" in the sense of developing "Software Agents" [37]. The latter is on a different – and complementary – level of abstraction. There is no contradiction, for example, to develop agent based software using aspect-oriented programming. [48] presented an agent-based mobile GIS at AGILE 2002, integrating specifications both of OGC and the software agent world (FIPA) [17].

2. HISTORY OF RESEARCH IN GIS DEVELOPMENT

The history of GIS development mirrors general trends in software development. Looking at software architectures we find a trend from monolithic GIS to modular approaches, object-orientation [45], components (e.g. [8] for 3D/4D), distributed GIS [42], web and recently mobile GIS and LBS [52] to dynamic architectures [40]. Another focus is on interoperability that lead to the foundation of the Open GIS Consortium. Their specifications deal mainly with interface definitions, not with the implementation of the components. Recent experiences with implementing OGC specifications [46], [47] or building OGC testbeds [6] are reported. Problems are reported by [36]. From this work on syntactic interoperability research in semantic interoperability through ontologies arose. The focus originally was mostly on data interoperability with few examples of work on functional ontologies [32]. On the level of abstraction of programming languages one finds a few papers, e.g., on query languages for spatial databases [14] to work on deductive or functional languages (e.g. [19], [27],), also in the context of OGC specifications [20].

While there was much work on object-orientation in GIS (e.g. [11]), the further path from objects to components has mostly been left to the industry. There are some papers regarding extensions to OO models [7] or UML for GIS modeling [31], [43]. But working on software engineering seems not to be regarded as a necessity within the GIS community. This lead to only a few papers on new software engineering paradigms in the GIS domain, e.g. the application of “design patterns” [22] to GIS [23] or the “Model Driven Architecture (MDA)” [24], [28]. We think that it is useful to reflect from time to time on new ideas developed in general computer science and software engineering. Therefore we want to investigate how such new concepts – in this case AOP - can be applied and exploited in GIS development.

3. INTRODUCTION TO ASPECT-ORIENTED PROGRAMMING (AOP)

While in the 1990ies object-orientation became the dominant paradigm for software development research into post-object programming (POP) has been carried out for a number of years [10]. Aspect-orientation is such a POP technology with good chances for mass adoption. While not attempting to replace object orientation it is rather an enhancement intended to complement object-oriented languages.

3.1 Modularity and Separation of Concerns

The basic idea behind AO is to provide advanced support for modularity combined with *separation of concerns* (SOC). The idea of modularity is not a new one [39] and all major programming paradigms in the past decades have strived to achieve support for it. While object-orientation has certainly been a major breakthrough proponents of AOP believe that the concept of aspects can solve some of the major problems object-oriented development is plagued by today that cannot be fully addressed by some of the recent advances in OO design such as design patterns [22].

3.2 Crosscutting concerns and Aspects

Separation of concerns can partly be achieved with object-oriented techniques, especially with the careful use of design patterns. However, there are many concerns that cannot be modularized, i.e. cleanly implemented in a single or at least limited number of classes. These concerns are called “*crosscutting*” concerns because they crosscut the structure of the class graph, that is, their implementation extends to many classes of a system, possibly to all its classes. Generic examples of crosscutting concerns are logging, tracing, performance tracking, security handling, distribution, persistence [38], quality of service (QoS) and so on. Without support for aspects the code tends to be tangled introducing severe problems, especially the non-explicit structure of the code and code redundancy. This redundancy is often deemed to be the root of all evil in software development.

An aspect is a cleanly modularized concern. An AO language must provide a possibility to specify the aspect definition easily and most of all in a single point. This is achieved by

introducing aspects as explicit language constructs. In addition, a mechanism is needed to specify rules for the execution of aspects. The process of linking aspects to traditional Java classes is called *weaving*. The weaving mechanism operates on so-called *join points*. These join points are well-defined points in the dynamic call graph at run-time, that is, points reached during the execution of a program.

4. REQUIREMENTS FOR AND IMPLEMENTATIONS OF AO TECHNOLOGIES

When evaluating some of the implementations of aspect-oriented languages several requirements should be considered. The requirements we deem important are:

- Desirable to be Java-based technology as Java is primary language for current academic (and enterprise) development
- Easy deployment, must work with standard platform-independent JVM
- tool support/integration with IDEs
- simple yet expressive

With these requirements in mind, some of the current major aspect-oriented efforts can be considered:

- Adaptive methods with Demeter/DJ [33]
- Multidimensional Separation of Concerns (MDSOC) with Hyper/J [35]
- Language extension to Java with AspectJ [30]

Although the Demeter paradigm has the advantage that it does not need a language extension to Java, it is also limited in scope and expressiveness. The most promising approach to adding support for aspects to Java seems to be the AspectJ language extension, a project of Xerox PARC. Although it does add a number of new constructs to the basic Java language these additions only matter at development time. The runtime library for AspectJ consists only of a small jar-file. The AspectJ distribution comes with its own development tool for visualizing and navigating the structure of aspect-oriented projects as well as plug-ins for integration with a number of integrated development environments.

5. ELEMENTS OF ASPECTJ

The main abstraction of AspectJ is called *aspect*. Aspects have much in common with classes: They can have methods and fields, extend normal java classes, implement interfaces, and may be abstract. In addition, they can extend other aspects and support the notion of domination, a mechanism for specifying aspect precedence.

Aspects can contain new constructs called *pointcut* and *advice*. Pointcuts provide a mechanism for specifying join points, i.e. well-defined points in the execution of the program. Examples for join points include object initialization, method calls, and field access. When defining a join point related to a method call it is possible to use powerful wildcards semantics for the method signature including name, arguments, return type as well as target object. The definition of an executable piece of functionality is called advice. An advice is defined with respect to a pointcut and can be run in a variety of ways, e.g. *before*, *after*, or even instead of a method call. The advice also contains a body, a block of code much like a normal method body. Within an advice elements of the surrounding non-aspect code such as method call parameters can be made accessible. AspectJ also has a mechanism for adding elements (fields, methods) to existing classes and change the inheritance and interface structure. This mechanism is called *introduction*. Introduction effectively changes the static structure of a program at compile-time as opposed to the dynamic nature of join points. While the dynamic join point model of AspectJ is very powerful it is still simple and accessible enough for most Java developers. It can be concluded that AspectJ is a suitable choice of technology for the implementation of aspect-oriented programming in Java.

6. INTRODUCTORY EXAMPLES FOR AOP

In order to better understand the nature of aspects and their application it is useful to consider some examples. The standard examples are taken from the domains of logging or tracing. These aspects are also called development aspects because they are more important during the development phase of a system than during production. They might even be switched off or removed totally from the system before the final product is shipped.

6.1 A Simple Tracing Aspect

During the development of a system programmers often insert debugging messages such as notifications of start or end of method or constructor execution in their code in order to better be able to follow the control flow of the system at run-time. While this is a simple debugging mechanism it has a number of drawbacks: The implementation of this feature is scattered over many classes of the system, the code of individual classes is cluttered with the message generation code, it is difficult to globally turn it off and on again, and worst of all, there is a lot of duplication of structurally similar code. In short, the implementation of this conceptually simple feature leads to severely tangled code. This solution can be improved by other means than aspects, e.g., in the case of Java, through the use of JPDA, the JVM's debugging interface. It can however much easier be achieved by using a simple aspect that does not have the undesired properties of the naïve Java-only solution. This simple tracing aspect consists of two parts: the definition of a pointcut and of an advice. The pointcut states which points in the execution of the program the aspect should affect. The advice instructs AspectJ at run-time what to do when such a joint point is reached in the control flow of the program. The following is a simplified adoption from [29]:

Listing 1:

```
aspect SimpleTracing
{
        pointcut methodCall():
                within( org.eml.* ) && call( public * *(..) );

before(): methodCall()
{
        log("entry:"+thisJoinPointStaticPart.getSignature());
        }

after(): methodCall()
{
        log("exit:"+thisJoinPointStaticPart.getSignature());
        }

void log(String message)
{
                //for simplicity;
                //could as well log to file/socket etc.
        System.out.println( message );
        }
}
```

This *aspect* is a container for other program constructs, much like a Java class. It defines a *pointcut*, which will be used to identify certain points in the execution flow of the program at run-time. In this case we want to pick the execution of all public methods regardless of their name, return type, and argument list. The methods must be defined on classes within the `org.eml` package.

We then declare two pieces of advice: *before-advice* and *after-advice*. This will execute the advice body before or after the condition defined in the pointcut occurs, respectively. The last item in the aspect is a traditional method used to factor out common behavior. When this aspect is compiled together with an aspect-unaware java application the whole program will be affected, in this case tracing messages for all (public) methods in `org.eml.*` classes will be generated. Although the behavior achieved is a global one its definition is nicely localized in a single aspect definition.

6.2 Advanced Aspects

As stated earlier aspects can capture a lot of other concerns that cut across a whole system or a large part of it. After an example of a development aspect we will now briefly introduce some “middleware aspects”, such as aspect-based support for persistence and distribution concerns.

In [38] a distributed system is restructured to capture the distribution-related code in a modularized aspect. The system uses Java Remote Method Invocation (RMI) to communicate between remote and server objects [41]. In the original program – as in many client-server applications – client-side user-interface objects call methods on a server object, which resides on a different computer.

While the persistence concerns comprising connection handling, transaction control, caching, object state synchronization, and others are more complex the essence of the distribution aspect is easier to understand. We will therefore give a simplified example of such an aspect that captures the spirit of the original example but ignores technical difficulties such as RMI or AspectJ limitations and exception handling.

Consider an object that bundles the intended services, say `ServiceObject`. Let this class implement some interface, say `ServiceInterface` that specifies the different remote services but doesn’t know anything about remote access. The fact that the behavior of an object unaware of aspects can still be influenced by aspects is sometimes referred to as obliviousness [21]. In order to remote-enable this service object according to the RMI specification we use the following aspect:

Listing 2:

```
aspect ServerDistribution
{
declare parents:
                ServiceInterface extends java.rmi.RemoteInterface;

declare parents:
        ServiceObject extends java.rmi.UnicastRemoteObject;

declare parents:
        BusinessObject implements java.io.Serializable;
}
```

This will make the `ServiceObject` a remote server giving remote access to the methods that are specified in the `ServiceInterface`. Note the third declare-parents-clause: Here we declare some business object to be serializable, a requirement for objects used as parameters or return values in RMI remote object invocations. This aspect is an example for AspectJ’s static crosscutting facility called introduction.

The aspect code for the client side is somewhat more complex, but basically AspectJ’s dynamic pointcut model allows intercepting calls made by the user interface objects. With the help of around advice these intercepted calls would then be redirected to the appropriate remote implementation of the service interface. Again, the client objects are unaware of the fact that their method calls on other objects are intercepted and redirected.

Although much simplified (and therefore not strictly correct), this example should give an impression of the possibilities offered by AspectJ to capture a truly crosscutting concern such as distribution in an aspect.

The two examples from the tracing and distribution areas illustrate some of the powerful features of AspectJ. The expressiveness of such an aspect-oriented language is greater than that of a purely object-oriented one. The support for modularizing crosscutting concerns as aspects is an important feature intended not to replace but to complement object-oriented languages such as Java.

The general methodology of applying AOP is as follows:

- Identify concerns that cut across class / object boundaries

- Write code (an aspect) that encapsulates that concern
- Define join points that specify how the aspect will be weaved into traditional code

In the following paragraphs the usage of AOP in GIS modeling and development are examined. In particular we discuss the modeling of domain knowledge as aspects. Modeling domain knowledge in a domain ontology helps for the first point in the list above. It would even allow to automate point two by developing a suitable "*ontology2aspect* generator". The crucial part that needs further investigation is point three. These three steps will be explained further in the remainder of the paper.

7. GIS AND AOP

When we investigate how AOP could support GIS development we can distinguish several types of applications where the use of AOP might be beneficial for GIS: The main idea is to find domains or parts of applications (algorithms) that can be handled and maintained independently from each other. Another point is that there might be existing applications (e.g. business applications or GIS applications) that should or cannot be changed, but need functional enhancements. Examples include to make GIS applications - like tour planning or map production - aware of new context or personalization parameters, in order to be able to deliver context-aware and personalized tour proposals (51, 26) or maps [53]. We discovered the following AOP & GIS application typology:

- (non spatial) business applications + GIS
- 2D + height
- 2D or 3D + time
- geometry + taxonomy
- GIS application + context awareness or personalization

7.1 Geographic Domain Modeling and Aspect Orientation

As already pointed out, we want to propose the integration of research on ontologies and aspect-oriented programming in the GIS domain. In [12] it is argued that separating the domain from the algorithm in software applications could solve some reuse and maintenance problems. Applying AOP to model the domain as an aspect program and the algorithm as the base program, allows them to evolve independently from one another. By algorithm the D'Hondts mean the functionality of a software application, whereas a domain or domain knowledge denotes concepts and constraints that model the real world and which an algorithm can be applied to. Therefore [12] propose to express domain knowledge in an appropriate environment acting in an aspect-oriented way. Based on this [9] developed an automatic code generator that translates the declarative quality constraints to pieces of executable code that are managed in a quality check module within a tour planning scenario, where a basic tour planning algorithm is enhanced through domain knowledge like priority or prohibited maneuvers.

Current software engineering practices result in software applications that contain implicit domain knowledge tangled with the core application functionality. Domain knowledge consists of concepts and relations between the concepts, as well as constraints on the concepts and the relations, and rules that state how to infer or calculate new concepts and relations. There is an analogy between ontologies (defining the domain knowledge in terms of concepts, rules and restrictions) and aspect-oriented programming.

7.2 Dynamic enhancements of geoobject-models with new aspects

Let us assume, we have a class hierarchy for geoobjects – e.g. based on the GML or OGC SFS standards. Now we want to extend this through a "time"-aspect, to make it aware of temporal changes. Time can be modeled on its own in several more or less complex ways. [49] illustrate how it is possible to extend, for example, GML through such temporal constructs, using a sophisticated object-oriented temporal model that handles a wide range of temporal constructs. See [50] how this temporal model can be combined with a geometric

3D model to a “4D” GIS data model. But in order to do this, is it necessary to change the original model by either enhancing the super-class of that model or defining coupling classes. This might not be desirable or possible in all situations for a number of reasons (e.g. “interoperability”). In such cases AOP can help to dynamically enhance existing class libraries of certain domain models to enrich them with new aspects.

In order to better understand the benefits of aspect-orientation for the GIS domain we will discuss an extended yet simple example. This example combines most of AspectJ’s language constructs and illustrates its capabilities for handling crosscutting concerns.

The starting point for our example is a situation where we have two different and independent class libraries, one is an implementation of a detailed object-oriented temporal model [50], the other is a GIS library based on the OGIS GML standard. This Java library is generated automatically from the GML XML schema definition with the help of the CASTOR data-binding framework (www.castor.org). As it is generated automatically – and must be regenerated if the GML specification changes – we cannot manually change these classes without creating serious maintenance problems. We further assume to have the temporal library in compiled form, i.e., as a Java jar file. Our goal is now to use aspects to introduce time-dependency into the GIS library without (explicitly) modifying its sources. We also want to define the time-dependency localized in one single place, that is, in a single AspectJ file. The aspect will affect structure and behavior of many - possibly unrelated - classes. Thus the time-dependency is a crosscutting concern, and an aspect is the new concept to encapsulate such a crosscutting concern.

Listing 3:
TimeDependency.aj

```
1    package org.eml.aop;
2
3    import org.eml.modell_4d.temporal.structure.interv;
4    import org.eml.deepmap.gml.objects.*;
5
6    //very simple example interface for time dependency
7    public interface TimeDependent
8    {
9       public long getBegin();
10      public long getEnd();
11   }
12
13   aspect TimeDependency
14   {
15      //static introduction for interface implementation
16    declare parents: (Feature || Geometry) implements TimeDependent;
17
18      //field introduction
19      public interv (Feature || Geometry).validTime = null;
20
21      //constructor introduction
22      public (Geometry+ || Feature+).new(interv interval)
23      {
24         //call to existing constructor
25         this();
26         //additional initialization
27         this.validTime = interval;
28      }
29
30      //method introductions
31      public long (Feature || Geometry).getBegin()
32      {
33         return this.validTime.getBeginChronons();
34      }
35
36      public long (Feature || Geometry).getEnd()
37      {
38         return this.validTime.getEndChronons();
39      }
```

```
40
41      //pointcut definitions
42      pointcut AllGetterMethods(TimeDependent object):
43               call( public * get*(..) ) &&
44               target( object );
45
46      pointcut DesiredGetterMethods(TimeDependent object):
47               AllGetterMethods( object ) &&
48               !call( public long getBegin() ) &&
49               !call( public long getEnd() );
50
51      //around advice
52      Object around(TimeDependent object): DesiredGetterMethods(object)
53      {
54         long now = System.currentTimeMillis();
55
56         if (( object.getBegin() < now ) && ( object.getEnd() > now ))
57            return proceed( object );
58         else
59            return null;
60      }
61   }
62
```

In the following we will discuss listing 3 in detail. First, we define a standard Java interface named `TimeDependent`, which is greatly simplified for the sake of clarity. This interface contains two methods to model access to the beginning and end of a continuous period of time. It is unknown to and independent of either the geographic or the temporal class hierarchies.

After this conventional Java interface we declare an AspectJ aspect named `TimeDependency`. This aspect will contain all program elements relevant to the temporal modeling. Here these elements are static introductions such as parent, constructor, method, and field introductions, as well as pointcut and advice definitions.

We plan the aspect to affect all members of the two separate class hierarchies with the root classes `Geometry` and `Feature`. We thus start to introduce new parents into both classes in line 16. These classes now implement the interface `TimeDependent` as if it was declared in the original source code. In order for this interface introduction to be valid and compilable we need to introduce the two methods of the interface into both classes, which is done in lines 31 through 39. Both methods reference an instance variable named `validTime` of type `interv`, which we have introduced into the `Feature` and `Geometry` classes in line 19.

The last example of static introduction is the constructor introduction in lines 22 through 28. The implementation of this constructor uses the existing no-argument constructor and additionally initializes the previously introduced instance variable `validTime`. Note that this constructor is introduced into all members of both class hierarchies individually not just into the two root classes by virtue of the '+'-wildcard in the type list. This enables us to use the new constructors as if they were declared within the respective class definitions: `Box box = new Box( new interv() ).Box`es can now be created with a time interval constructor argument just like any other subclass of `Geometry` or `Feature`. These introductions are called static introductions because they change the class structure, hierarchy, and dependencies at compile-time. They also do this in a crosscutting manner, that is, they affect many different, and potentially unrelated files from a single aspect definition.

The remaining elements of the aspect are pointcut and advice definitions, which change the behavior of the affected classes at run-time. In this case we define a composite pointcut named `DesiredGetterMethods` in lines 46 through 49 which selects certain getter-methods of objects of type `TimeDependent`, that is, instances of `Geometry`, `Feature`, or any of their respective subclasses. This pointcut is defined with the help of a simpler pointcut, which selects all public getter-methods of the mentioned objects. The

composite pointcut needs to exclude the two methods `getBegin()` and `getEnd()` in order to prevent undesired recursion.

We can now use the pointcut `DesiredGetterMethods` to define a piece of advice, i.e., the functionality that is to be executed before, after, or instead of the methods selected by the pointcut. Here, we want to replace the existing behavior of all getter-methods (that is, all access methods) by some new, time-dependent behavior. This can be achieved by the *around advice* in lines 52 through 60. In this simple example we only check if the time of the method invocation is within the time span defined as valid. If the method call occurs at an invalid time, we return `null` to signal this invalid situation. If the invocation time is valid we return the unaltered result of the invocation of the replaced method by using the `proceed` keyword in line 57. This mechanism is similar to calling a super method when overriding a method in the case of standard object-oriented inheritance.

After reviewing this example we can summarize the advantages of using aspects to enhance GIS data models and class libraries as follows:

- Extension of the functionality of an existing GIS library without modification of its sources
- Combination of an existing GIS library with an unrelated library from a different domain
- Modification of the behavior of classes scattered over many places in the source code in a single aspect

Similarly, it is for example possible to enhance other existing non-spatial-aware domain-models (e.g. from business-oriented domains) with spatial “aspects” in order to spatially enrich them. Another example is to add a new spatial dimension (e.g. the height dimension) to existing GIS applications for analysis purposes.

More classical examples include to “weave in” the mentioned middleware-style features (e.g. logging, tracing, performance monitoring, persistence, error-handling, security, distribution) into existing GIS class libraries.

7.3 Generating aspect-oriented application code from geographic ontologies

Based on the concepts explained we developed the idea, that it should be possible to derive aspects from an available explicit domain ontology automatically. The result would be runnable aspects (e.g. as AspectJ constructs). Exploiting this idea could make it possible to derive aspects that can be weaved into existing application code automatically. This allows the enrichment of basic algorithms through advanced domain concepts, as well as the traditional application level improvements like distribution, logging, persistence etc. This is illustrated in figure 1. The use of an ontology for the development of information systems leads to Ontology driven Information Systems [25] and from these to Ontology Driven Geographic Information Systems (ODGIS) [18]). While in most examples of AOP [2] its usage for middleware concerns has been stressed, [13] found that also domain knowledge crosscuts the core application, and that domain knowledge can be modeled as aspects. Crosscutting of domain knowledge and application code leads to the typical problems encountered when there is no loose coupling at design and implementation levels between conceptually separate components.

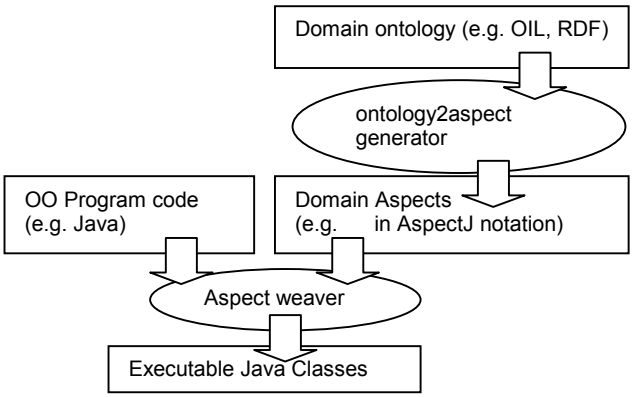

Fig. 5 Framework for deriving executable Aspects from formal domain ontologies

Real-world domains are subject to change. Therefore, it is necessary to identify and locate the software's domain knowledge easily and change it accordingly. Similarly, due to the rapid evolution of technology, we want to update or replace the core application in a well-localized and controlled manner. When the core application is tangled with domain knowledge development is more complex than necessary: the developer has to concentrate on two aspects of the software at the same time and compose them manually.

8. CONCLUSIONS

We have presented the first systematic discussion of the applicability of AO in the GIS domain. A detailed example illustrates the potential of the AO approach in GIS models and systems. The nature of GIS models as a composition of spatial, temporal, and thematic aspects makes this an ideal domain to further research and exploit this new paradigm.

AOP provides interesting new concepts, many of which are borrowed from metalevel programming. But as AOP is easier to understand it seems likely that it will find a broader acceptance. The benefits of considering domain knowledge as an aspect can be summarized as follows: The separation of domain knowledge from its application makes the programming process less complex and frees the programmer from intertwining the domain aspect with the component program manually. This leads to applications that are easier to understand and maintain, since they are less cluttered with domain-specific code. Due to weak coupling of domain knowledge and application, they can evolve independently from each other in a flexible and manageable way. The potential for reusing them for other applications and in other domains can been enhanced.

9. ACKNOWLEDGEMENTS

This work has been funded by the Klaus Tschira Foundation (KTS) and the German Ministry of Education and Research (01IL905C).

10. REFERENCES

[1] Aspect-Oriented Software Development http://www.aosd.org

[2] AOP Bibliography (2002): http://www.aosd.org

[3] AspectJ Programming Guide http://www.aspectj.org/doc/dist/progguide/index.html

[4] AspectJ Tutorial http://www.aspectj.org/doc/dist/tutorial.pdf

[5] AspectJ FAQ http://www.aspectj.org/doc/dist/faq.html

[6] Bernard L.(2002): Experiences from an implementation Testbed to set up a national SDI. AGILE 2002. International Conference on Geographic Information Science of the Association of Geographic Information Laboratories in Europe (AGILE). 04.2002. Palma. Spain.

[7] Borges K. A.V., Clodoveu A. Davis, Alberto H.F. Laender (2001): OMT-G: An Object-Oriented Data Model for Geographic Applications. Geoinformatica. Volume 5, Issue 3 pp. 221-260.

[8] Breunig, M. (2000): On the Way to Component-Based 3D/4D Geoinformation Systems. Lecture notes in Earth Sciences. 94. Springer. Berlin, Heidelberg, New York.

[9] Casanova, M, D'Hondt, M. and Thomas Wallet, T. (2001): Explicit Domain Knowledge in Geographic Information Systems. In proceedings of SEKE, June 2001, Buenos Aires, Argentina.

[10] Czarnecki, K. and Eisenecker, U.W. (2000): Generative Programming: Methods, Tools, and Applications. Addison Wesley, Boston.

[11] Das Neves, F., Gordillo, S., Mostaccio, C., and Levato, A. (1997): Toward a foundation for Object Oriented GIS Design". Proceedings of the Joint European Conference On Geographical Information. Vienna, Austria. April 16-18, pp. 58-63.

[12] D'Hondt, M. and D'Hondt, T. (1999): Is domain knowledge an aspect? ECOOP99, Aspect-Oriented Programming Workshop.

[13] D'Hondt, M. and M. A. Cibran (2002): Domain Knowledge as an Aspect in Object-Oriented Software Applications. ECOOP 2002, Workshop on Knowledge-Based Object-Oriented Software Engineering. In Association with the 16th European Conference on Object-Oriented Programming. Málaga, Spain. June 10 - 14, 2002.

[14] Egenhofer, M. (1992): Why not SQL! International Journal of Geographical Information Systems, 6 (2): 71-85, 1992.

[15] Egenhofer, M.J. and A. Frank (1992): Object-oriented modeling for GIS. Journal of the Urban and Regional Information Systems Association, 4, 3-19.

[16] Elrad, T; R. E. Filman and A. Bader (2001): Aspect-Oriented Programming. Communications of the ACM, 44(10):29-32.

[17] FIPA - Foundation for Intelligent Physical Agents: http://www.fipa.org/specifications/index.html

[18] Fonseca, F., M. Egenhofer, P. Agouris, and G. Câmara (2002): Using Ontologies for Integrated Geographic Information Systems. Transactions in GIS, 6(3): 231-257, 2002.

[19] Frank, A.U. (1984); Extending a Database with Prolog. In: First international Workshop on Expert Database Systems. Edited by L. Kerschberg, (Kiawah Island SC), pp. 665-676.

[20] Frank , A.U.; Kuhn, W., 1995. Specifying Open GIS with Functional Languages. In: Egenhofer, M.J.; Herring, J.R. (Eds.), : Advances in Spatial Databases. LNCS. 951. Springer, Berlin, 184-195.

[21] Filman R. E. and D. P. Friedman (2000): Aspect-Oriented Programming is Quantification and Obliviousness. In Workshop on Advanced Separation of Concerns, OOPSLA'00.

[22] Gamma, E., R.Helm, R.Johnson, J.Vlissides. "Design Patterns. Elements of Reusable Object-Oriented Software". Addison-Wesley 1995.

[23] Gordillo, S., F. Balaguer, C. Mostaccio, F. Das Neves (1998): Developing GIS Applications with Objects. A design Patterns Approach. GeoInformatica Vol 3, Number 1, pp 7-32.

[24] Grønmo, R. (2001): Supporting GI standards with a model-driven architecture. In: Walid G. A. (Ed.): ACM-GIS 2001, Proceedings of the Ninth ACM International Symposium on Advances in Geographic Information Systems, Atlanta, GA, USA, November 9-10, 2001. ACM. pp. 100-105

[25] Guarino, N. (1998): Formal Ontology and Information System. In: N. Guarino (Ed): Proceedings of the 1st International Conference on Formal Ontologies in Information Systems, FOIS'98, Trento, Italy,. IOS Press, June 1998. pp 3-15.

[26] Jöst, M. and W. Stille (2002): A User-Aware Tour Proposal Framework Using a Hybrid Optimization Approach, In: Proc of ACM-GIS 2002, the 10th ACM Int'l Symp. Advances in Geographic Information Systems, ACM Press, New York, 2002, pp. 329–338.

[27] Jones, C.B. (1989): Cartographic name placement with Prolog. IEEE Computer Graphics and Applications, 9(5), pp 36-47.

[28] Keller S. F. (2002): Towards Open Geo-Visualization Services SVG Open / Carto.net Developers Conference Zurich, Switzerland - July 15-17, 2002.

[29] Kiczales G., J Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin (1997): Aspect-Oriented Programming. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag, LNCS 1241.

[30] Kiczales, G.; E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. (2001): Getting Started with AspectJ. Communications of the ACM, 44(10):59,65.

[31] Kolodziej, K. & Winter, S., (2001): UML for Building Interoperable, Distributed GIS Service Components. UCGIS Summer Assembly, SUNY Buffalo.

[32] Kuhn, W. (2001): Ontologies in support of activities in geographical space. International Journal of Geographical Information Science, 15(7): 613-631.

[33] Lieberherr, K. Orleans D., Ovlinger J. (2001): Aspect-Oriented Programming with Adaptive Methods. Communications of the ACM, 44(10): 39-41.

[34] OpenGIS Consortium (1999): The Geographic Markup Language - GML. www.opengis.org

[35] Osher, H and P. Tarr (2001): Using Multi-Dimensional Separation of Concerns to (Re)shape Evolving Software. Communications of the ACM, 44(10): 43-50.

[36] Ostman, A., J. Nogueras, S. Winter (2002): Barriers for the implementation of GI standards and interoperability. Proc. of the 5th AGILE Conference, Mallorca (Spain).

[37] Shoham, Y. (1993): Agent-Oriented Programming, Artificial Intelligence (60), pp. 51-92.

[38] Soares, S.; E. Laureano and P. Borba (2002): Implementing distribution and persistence aspects with AspectJ, Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications, ACM Press, Seattle, Washington, USA, pp. 174-190.

[39] Parnas, D.L. (1972): On the criteria to be used in decomposing systems into modules. Commun. ACM 15, 2.

[40] Tsou M. & Buttenfield B.P. (2002): A Dynamic Architecture for Distributing Geographic Information Services. Transactions in GIS, October 2002, vol. 6, no. 4, pp. 355-381(27).

[41] Sun (2002): Java Remote Method Invocation (RMI). At: http://java.sun.com/products/jdk/rmi/

[42] Vckovski, A. (1998): Interoperable and distributed processing in GIS. Taylor & Francis. London.

[43] Wallet, T., M. Casanova, and M. D'Hondt (2000): Ensuring quality of geographic data with uml and ocl. In Third Int. Conf. on the Unified Modeling Language (UML 2000), 225–239. Springer.

[44] Winter, S. & Nittel, S. (forthcoming): Formal Information Modeling for Standardisation in the Spatial Domain. Accepted for publication in the International Journal of Geographical Information Science.

[45] Worboys, M., Hearnshaw, H., and Maguire, D., (1990): Object-Oriented Data Modeling for Spatial Databases. International Journal of Geographical Information Systems, 4. 369-383.

[46] Zarazaga F.J.1 , R.López, J. Nogueras, O. Cantán, P. Álvarez, P.R.Muro-Medrano (2000): First Steps to Set Up Java Components for the OpenGIS Catalog Services and its Software Infrastructure. 3rdAGILE Conf. on Geographic Information Science. Helsinki. May 25.- 27.05.2000.

[47] Zipf, A. und Aras, H. (2001): Realisierung verteilter Geodatenserver mit der OpenGIS SFS für CORBA. In: GIS. 3/2001. GIS - Geo-Informations-Systeme. Zeitschrift für raumbezogene Information und Entscheidungen. Heidelberg. pp 36-41.

[48] Zipf, A. and Aras, H. (2002): Proactive Exploitation of the Spatial Context in LBS - through Interoperable Integration of GIS-Services with a Multi Agent System (MAS). AGILE 2002. Int. Conf. on Geographic Information Science. 04.2002. Palma. Spain.

[49] Zipf, A. and Krüger, S. (2001a): TGML - Extending GML by Temporal Constructs - A Proposal for a Spatiotemporal Framework in XML. In: Proceedings of ACM GIS 2001. Atlanta USA.

[50] Zipf, A. und Krüger, S. (2001b): Ein OO-Framework für temporale 3D-Geodaten. AGIT 2001, Symp. für Angewandte Geog. Informationsverarbeitung, 04.-06.06.2001, Salzburg. Austria.

[51] Zipf, A und Röther, S. (2000): Tourenvorschläge für Stadttouristen mit dem ArcView Network Analyst. In: Liebig (Hrsg.)(2000): ArcView Arbeitsbuch. Hüthig Verlag. Heidelberg.

[52] Zipf, A. (2001): Interoperable GIS-Infrastruktur für Location-Based Services (LBS) - M-Commerce und GIS im Spannungsfeld zwischen Standardisierung und Forschung. In: GIS, Geo-Informations-Systeme. Zeitschrift für raumbezogene Information und Entscheidungen. 09/2001. 37-43.

[53] Zipf, A. (2002): User-Adaptive Maps for Location-Based Services (LBS) for Tourism. In: K. Woeber, A. Frew, M. Hitz (eds.), Proc. of the 9th Int. Conference for Information and Communication Technologies in Tourism, ENTER 2002. Innsbruck, Austria. Springer Computer Science. Heidelberg, Berlin.