Is Domain Knowledge an Aspect?

Maja D'Hondt
mjdhondt@vub.ac.be
System and Software Engineering Lab

Theo D'Hondt
tjdhondt@vub.ac.be
Programming Technology Lab

Department of Computer Science, Brussels Free University
Pleinlaan 2, B-1050 Brussels, Belgium

April 11, 1999

Abstract. Our industrial partners have difficulties coping with maintenance and reuse of their software applications, due to the hard coded presence of domain knowledge in the implementation. This paper reports on the remarkable analogy between this phenomenon and aspect-oriented programming, which leads to the interesting question: Is domain knowledge an aspect? Using a simplified example based on a real-world application, we illustrate that this problem could benefit from the AOP philosophy. Nevertheless, a number of important issues remain to be addressed.

Contents

1 Introduction

Domain knowledge indicates the collection of concepts existing in a domain, relations between these concepts and constraints on these concepts. The term domain knowledge originated in the artificial intelligence community, where it is used in knowledge-based applications such as expert systems, and in systems for natural-language understanding. Only recently the software engineering community has realized that domain knowledge is also relevant in the development of software systems, since almost all software is applied to some domain. Software developers are now (or should be) performing domain analysis in the first phases of the software life cycle, which results in an explicit description of the domain. However, this practice has not yet infiltrated the design and implementation phases: the developer still relies on a mental picture of the domain knowledge when manually hard coding it in the application code.

Cooperation with several industrial partners that produce large and complex software applications, allows us to identify some of the problems encountered in today's software development. A common complaint concerns the hard coded combination of application functionality and domain knowledge. This results in code that is difficult to maintain and even more difficult to reuse. We distinguish several activities that lead to maintenance and reuse problems:

- When existing domain knowledge is modified at a conceptual level, the relevant piece of domain knowledge has to be localized in the implementation in order to adapt it accordingly. The mixture of domain knowledge and application functionality makes this a complex and error-prone task. The broader implications are that it is almost impossible to reuse an application for a different domain.
- The competitive software business encourages the development of a suite of applications for the same domain. A lack of synchronization is experienced between the domain knowledge on the one hand and each application it is used in on the other. So, although in some cases domain knowledge is available at a conceptual level as a result of domain-analysis, it is very hard to reuse domain knowledge at the implementation level across applications.

In short, people who are involved in the development of large-scale software applications, report that the domain and the application can no longer evolve independently. This is illustrated at the hand of an example.

2 Example from a Geographic Information System

For the illustration of our ideas, we turn to a real software application, more specifically a geographic information system (GIS), developed by one of our industrial partners. A GIS can support anything from car navigation, environmental planning and control, to alarm call and dispatch. Such a system works with a vast amount of geographic data, which is actually its domain knowledge. We give an overview on how these data is modeled and zoom in on the part of the domain knowledge that is used in the example. As a model for part of a car navigation system, a simple algorithm to calculate the shortest path, is also presented to show how an application uses domain knowledge. Note that although the example is based on the domain of a real application, we have kept it simple in order to demonstrate our ideas more clearly.

2.1 The Domain Knowledge

Geographic data consists of everything related to buildings, roads and areas: gas stations, hotels, airports, highways, railways, ferries, street names, towns, states, parks, industrial areas and so on. These data are captured from satellite images, aerial pictures, reports of intensive field surveys and official source maps form local, regional and national authorities, and stored in a database. This spatial information is presented as vector data: it consists of nodes, edges and areas. An unlimited number of features can be assigned to those objects, such as a road type, a street name, a city name and so on. Vector data is also able to define relationships between nodes, edges and areas. An example is a prohibited manoeuvre, which defines a relationship between two nodes, denoting that it is prohibited to move from the first node to the second with a vehicle.

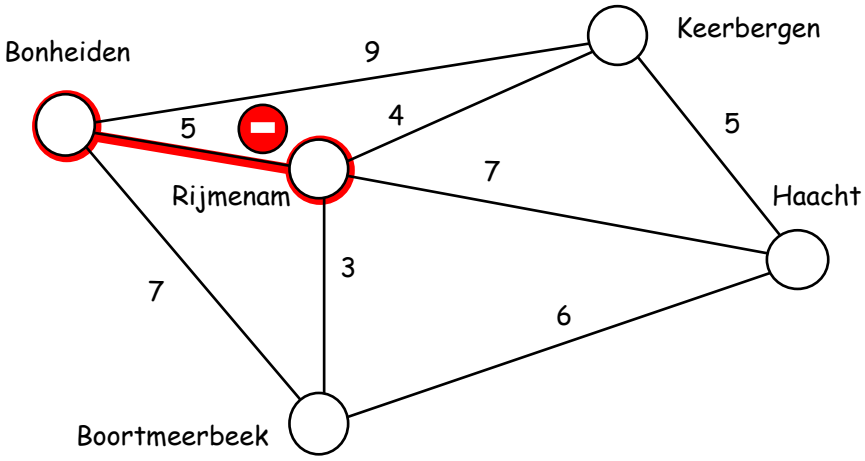

figure 1: A visual representation of a road network between towns showing distances and a prohibited manoeuvre in red.

An example of the domain of geographic data is presented in figure 1. It shows five nodes and eight edges. The nodes have features attached to them, indicating that they are villages. The edges also have features, in this case these denote distances between the connected nodes. Moreover, the villages Rijmenam and Bonheiden are part of a prohibited manoeuvre, which means that it is prohibited for vehicles to drive from the first to the second along their connecting edge.

2.2 A Car Navigation Application

An important part of the functionality of a car navigation application is an algorithm that calculates the shortest path from a starting node to a destination node. Branch and bound (B&B) is a very simple algorithm in this category which always returns the best solution and although theoretically intractable, has a reasonable observed performance. B&B is presented here to play the role of application on geographic domain knowledge in our example.

In short, the B&B algorithm performs an exhaustive search on all the nodes in the graph, selecting the shortest distance first, discarding all paths that exceed the best solution yet found, and at the same time avoiding loops. Because this algorithm is developed to run on geographic data, it also has to take prohibited manoeuvres into account. Figure 2 shows an implementation of this B&B algorithm written in `Pico` [1],

```
branch(node, action(next, dist)):
  { node[free_idx]:= false;
    display("enter ", node[name_idx], eoln);
    for(pos: link_idx, pos<=size(node), pos:=pos+1,
        { link: node[pos];
          if(prohibited(node, link[next_idx]),
             { next: link[next_idx];
               display("prohibited ", next[name_idx], eoln) },
             action(link[next_idx], link[dist_idx])) });
    display("exit  ", node[name_idx], eoln);
    node[free_idx]:= true }

branch_and_bound(start, stop):
  { bound: 999999999;
    traverse(node, sum):
      if(free(node),
         if(sum<bound,
            if(node==stop,
               { bound:= sum;
                 display("reach ", node[name_idx],
                         " after ", bound, eoln) },
               branch(node, traverse(next, sum+dist)))));
    traverse(start, 0);
    display("from ", start[name_idx],
            " to ", stop[name_idx],
            " is ", bound, eoln) }
```

figure 2: An implementation of the B&B algorithm. The domain knowledge concerning prohibited manoeuvres is shown in red.

The program in figure 2 illustrates that the domain knowledge concerning prohibited manoeuvres imposes a constraint on the functionality of the B&B algorithm. This constraint is translated into an `if`-test by a programmer, who also has to decide where to put this domain knowledge in the implementation of the B&B algorithm.

The output of the program in figure 2, when it is run on the data of figure 1 with the start node Haacht and the stop node Bonheiden, is seen in figure 3.

Since the algorithm chooses the earest node first, it enters Keerbergen from Haacht. By pursuing this tactic it enters Rijmenam, then Boortmeerbeek and eventually reaches Bonheiden for the first time after 19 kilometers. The algorithm backtracks to Rijmenam, after which it wants to select the edge to Bonheiden. However, this option is discarded since it is prohibited to take the road from Rijmenam to Bonheiden. B&B continues its search, skipping routes that exceed the best intermediate solution, and reports that the shortest path is 13 kilometers. Note that the prohibited manoeuvre obstructs an even shorter path, Haacht - Rijmenam - Bonheiden, of only 12 kilometers.

```
branch_and_bound(Haacht, Bonheiden)
:enter Haacht
:enter Keerbergen
:enter Rijmenam
:enter Boortmeerbeek
:reach Bonheiden after 19
:exit  Boortmeerbeek
:prohibited Bonheiden
:exit  Rijmenam
:reach Bonheiden after 14
:exit  Keerbergen
:enter Boortmeerbeek
:enter Rijmenam
:enter Keerbergen
:exit  Keerbergen
:prohibited Bonheiden
:exit  Rijmenam
:reach Bonheiden after 13
:exit  Boortmeerbeek
:enter Rijmenam
:enter Boortmeerbeek
:exit  Boortmeerbeek
:enter Keerbergen
:exit  Keerbergen
:prohibited Bonheiden
:exit  Rijmenam
:exit  Haacht
:from Haacht to Bonheiden is 13
```

figure 3: The output of the program in figure 2, when it is run on the data in figure 1.

2.3 Modifying the Domain Knowledge

As domain knowledge evolves, the application has to be updated to reflect this evolution. Consider a new kind of manoeuvre in the example: a priority manoeuvre is added to the domain knowledge shown in figure 1. A priority manoeuvre is a sequence of three or more nodes, as is shown in figure 4. A vehicle moving on the road made up of the edges between the nodes of a priority manoeuvre, has priority over all vehicles coming from other roads (edges) connected to one of these nodes. In figure 4, the nodes Haacht, Boortmeerbeek and Bonheiden make up a priority manoeuvre.

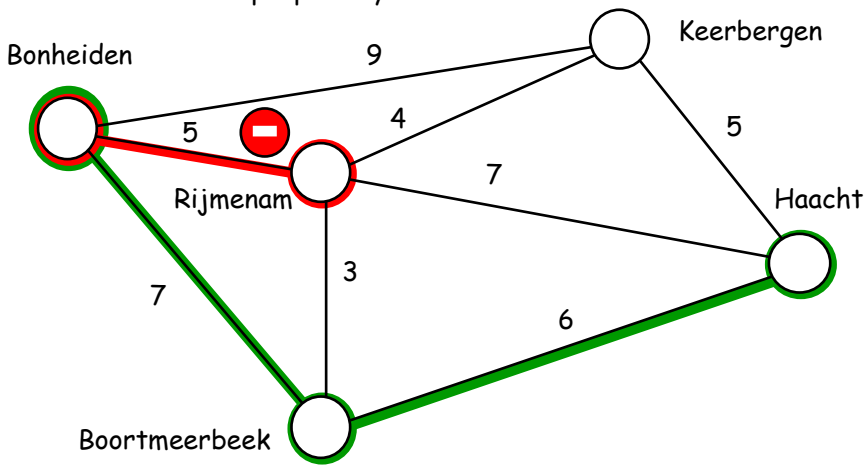

figure 4: A visualization of the domain knowledge extended with a priority manoeuvre in green.

It is useful to incorporate this domain knowledge in a car navigation application, more specifically in the B&B algorithm: Studies have shown that, once on a priority road, vehicles often reach their destinations faster when they stay on this road, although this route might not actually be the shortest in distance. This means that when Boortmeerbeek is reached and the vehicle is coming from Haacht, Bonheiden is preferred over Rijmenam as the next destination.

The updated B&B algorithm, in figure 5, has undergone quite a few changes. In addition to the first loop (the `for`-loop in black), another loop is added (in green) in order to detect a priority manoeuvre. The new B&B proceeds as follows: If the previously visited node and the current node are part of a priority manoeuvre, then the next node in this manoeuvre is the best choice in finding the shortest path. This strategy has major repercussions for the implementation of the B&B, since an extra parameter `previous_node` holding the previously visited node has to be passed around.

```
branch(previous_node, node, action(curr, next, dist)):
  { node[free_idx]:= false;
    display("enter ", node[name_idx],
            " from ", previous_node[name_idx], eoln);
    for(pos: link_idx, pos<=size(node), pos:=pos+1,
        { link: node[pos];
          if(priority(previous_node, node, link[next_idx]),
             { display("priority ",
                       link[next_idx, name_idx],
                       eoln);
               action(node, link[next_idx], link[dist_idx]) }) });
    for(pos: link_idx, pos<=size(node), pos:=pos+1,
        { link: node[pos];
          if(!priority(previous_node, node, link[next_idx]),
             if(prohibited(node, link[next_idx]),
                { next: link[next_idx];
                  display("prohibited ", next[name_idx], eoln) },
                action(node, link[next_idx], link[dist_idx]))) });
    display("exit  ", node[name_idx], eoln);
    node[free_idx]:= true }

branch_and_bound(start, stop):
  { bound: 999999999;
    traverse(previous, node, sum):
      if(free(node),
         if(sum<bound,
            if(node==stop,
               { bound:= sum;
                 display("reach ", node[name_idx],
                         " after ", bound, eoln) },
                 branch(previous,
                        node,
                        traverse(curr, next, sum+dist))))));
    traverse(empty_node, start, 0);
    display("from ", start[name_idx],
            " to ", stop[name_idx],
            " is ", bound, eoln) }
```

figure 5: The B&B algorithm from figure 3 extended with domain knowledge about priority manoeuvres, shown in green.

The output shown in figure 6 is the result of running this program on the data from figure 4. Note that when the algorithm has backtracked to the start node Haacht and then chooses Boortmeerbeek, it has entered a priority manoeuvre. The next selection is Bonheiden, since the algorithm takes advantage of being on the priority manoeuvre by staying on it.

```
branch_and_bound(Haacht, Bonheiden)
:enter Haacht from no_name
:enter Keerbergen from Haacht
:enter Rijmenam from Keerbergen
:enter Boortmeerbeek from Rijmenam
:reach Bonheiden after 19
:exit  Boortmeerbeek
:prohibited Bonheiden
:exit  Rijmenam
:reach Bonheiden after 14
:exit  Keerbergen
:enter Boortmeerbeek from Haacht
:priority Bonheiden
:reach Bonheiden after 13
:enter Rijmenam from Boortmeerbeek
:prohibited Bonheiden
:exit  Rijmenam
:exit  Boortmeerbeek
:enter Rijmenam from Haacht
:enter Boortmeerbeek from Rijmenam
:exit  Boortmeerbeek
:enter Keerbergen from Rijmenam
:exit  Keerbergen
:prohibited Bonheiden
:exit  Rijmenam
:exit  Haacht
:from Haacht to Bonheiden is 13
```

figure 6: The output of the B&B extended with priority manoeuvres, when it is run on the data in figure 4.

3 Applying AOP Principles

The example illustrates that domain knowledge is often hard coded in algorithms, not only in the program statements, but also at the level of parameter passing. Because of this fact, our industrial partners have problems maintaining and reusing their software applications. Therefore, we are currently engaged in projects where it is our task to tackle this problem. After investigating the problem and observing what is illustrated in the example, we could not help but notice the analogy with aspect-oriented programming [2]: The component program, an algorithm, is cross-cut with the aspect domain knowledge.

3.1 Introducing the Domain Aspect

What are the benefits of considering domain knowledge as an aspect? Based on the argumentation used in [3] for separation of concerns, we get the following:

- When domain knowledge is treated as an aspect of its application, it can be dealt with separately, thus making the programming process less complex. The programmer no longer has to intertwine the domain aspect with the component program manually, depending on an often incorrect mental picture of the domain knowledge.
- Describing domain knowledge in a separate aspect language results in applications that are easier to understand and maintain, since the implementation ceases to be cluttered with domain-specific code.

- Due to weak coupling of domain knowledge and application, they can evolve independently form each other in a flexible and manageable way. Moreover, both can be reused for other applications and on other domains respectively.

This setup has other advantages in the context of a single software system, where a specific domain and application are cooperating. Consider again the B&B algorithm in the geographic domain. Suppose this algorithm needs to be fine-tuned for different circumstances: In a busy area with a lot of traffic it is faster to take a priority road and stay on it, but in the countryside where traffic is low one is better off using the shortest route. In the first case, the B&B algorithm has to take the domain knowledge of priority manoeuvres into account, whereas the second case does not require this extra knowledge. Representing the domain knowledge in an aspect language that supports sufficiently fine-grained manipulation of concepts, relations and constraints, allows one to select the required subset of domain knowledge.

3.2 The Aspect Program

At this stage in our research, we are not able to define the properties of an aspect language for domain knowledge suficiently clearly. Therefore we cannot display the domain aspect of the example in a straightforward way. But let us examine the structure of this domain knowledge more closely. Figure 7 presents another view on the geographic domain, modeled in a diagram similar to a UML class diagram. The yellow post-it notes represent the constraints concerning prohibited and priority manoeuvres. They are written in natural language, because it is simple and avoids commitment to a specific representation for now.

Since the notion of domain knowledge originated with artificial intelligence, theories and techniques that were developed and validated for its representation, should also be considered as candidates for an aspect language for domain knowledge. Ontologies can be used for describing domain knowledge on a conceptual level. According to [4], ontologies are "content theories about the sorts of objects, properties of objects, and relations between objects that are possible in a specified domain of knowledge". An ontology can be specified using formalisms such as the Knowledge Interchange Format (KIF) or Open Knowledge Based Connectivity (OKBC). KIF is basically a first order predicate logic, whereas OKBC extends this with frame-based language constructs, to improve the representation of classes and their attributes.

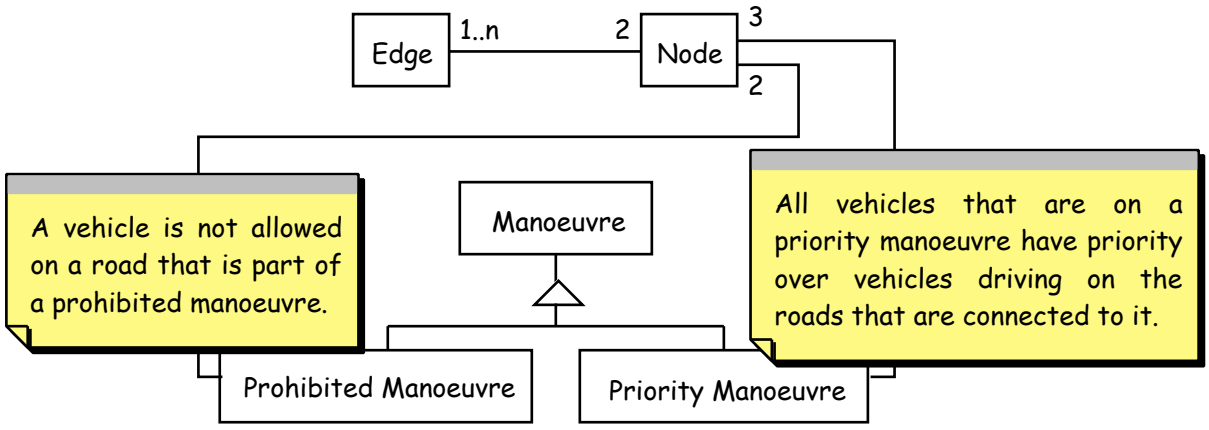

figure 7: Geographic domain knowledge modeled using an UML-like class diagram. The post-it notes indicate the constraints that are imposed on the domain.

3.3 The Component Program

In the example, filtering the domain aspect out of the B&B algorithm results in a general, domain-independent B&B algorithm that can be seen in figure 8.

```
branch(node, action(next, dist)):
  { node[free_idx]:= false;
    display("enter ", node[name_idx], eoln);
    for(pos: link_idx, pos<=size(node), pos:=pos+1,
        { link: node[pos];
          action(link[next_idx], link[dist_idx]) });
    display("exit  ", node[name_idx], eoln);
    node[free_idx]:= true }

branch_and_bound(start, stop):
  { bound: 999999999;
    traverse(node, sum):
      if(free(node),
         if(sum<bound,
            if(node==stop,
               { bound:= sum;
                 display("reach ", node[name_idx],
                         " after ", bound, eoln) },
               branch(node, traverse(next, sum+dist)))));
    traverse(start, 0);
    display("from ", start[name_idx],
            " to ", stop[name_idx],
            " is ", bound, eoln) }
```

figure 8: The general B&B algorithm without the domain aspect.

It is clear that this standard algorithm can be applied to any other domain that requires the calculation of the shortest path, and that it can evolve independently from it.

4 Other Issues

First of all we remark on the presence of nodes and edges in the general B&B algorithm in figure 8, although these concepts are part of the domain knowledge of a GIS. This raises a question: How do we demarcate the domain knowledge we want to remove from the component program? Naturally even a general algorithm should have access to some knowledge, and in this case this includes nodes and edges. This suggests that a domain and an algorithm, in order for them to be combined successfully, should fulfill a condition: The domain knowledge inherent to a general and otherwise domain-independent algorithm, should be compatible with the domain knowledge it is applied to. Additionally, this might indicate how the join points between concepts of the domain aspect and concepts of the component program could be determined.

```
branch(previous_node, node, action(curr, next, dist)):
  { node[free_idx]:= false;
    display("enter ", node[name_idx], eoln);
    link: select_next_node(node);
    while(link != void,
          { action(node, link[next_idx], link[dist_idx]);
            link:= select_next_node(node) });
    display("exit  ", node[name_idx], eoln);
    node[free_idx]:= true }
```

figure 9: The selection of the next node is made explicit (shown in blue).

Another issue is establishing the join points at the level of domain knowledge constraints. Again, we turn to the example to illustrate how this might be accomplished. Figure 9 shows a fragment of the B&B implementation where the selection of the next node is made explicit, in order to facilitate the coordination of the aspect program with the component program. Note that this approach does not deal with the extra parameter.

Finally, we wish to make a few notes on the aspect weaver. Since we are still unsure of the properties of a domain aspect language as well as of how to determine the join points, the design of the aspect weaver also remains undecided. Nevertheless, weaving the domain aspect at compile-time with the algorithm would ensure performance, which is certainly a necessity in the case of the B&B algorithm .

5 Conclusions

In this paper we pointed to domain knowledge as a possible aspect of programs. Domain knowledge is hard coded in the implementation of software applications, thereby introducing difficulties in maintaining and reusing both the domain and the application. This problem shows remarkable similarities to the problems that are tackled with AOP, which prompted us to investigate this observation more closely. Through our contacts with industry, we constructed an example based on an existing geographic information system, although simplified for the purpose of demonstrating our ideas. This example illustrated that considering domain knowledge as an aspect of software applications could solve the maintenance and reuse problems mentioned above. However, many issues remain to be investigated, such as finding a suitable aspect language, determining and describing the joint points and the design of the aspect weaver.

6 References

[1] http://pico.vub.ac.be/

[2] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, ECOOP'97--Object-Oriented Programming, 11th European Conference, volume 1241 of Lecture Notes in Computer Science, pages 220-242, Jyväskylä, Finland, 9-13 June 1997. Springer.

[3] Walter L. Hürsch and Cristina Videira Lopes. Separation of Concerns. North Eastern University, February 1995.

[4] B. Chandrasekaran, John R. Josephson and V. Richard Benjamins. What Are Ontologies, and Why Do We Need Them? IEEE Intelligent Systems, pages 20-26, January/February 1999.