Making Software Knowledgeable

[Extended Abstract]

Maja D’Hondt
System and Software Engineering Lab
Vrije Universiteit Brussel
Belgium
mjdhondt@vub.ac.be

The complexity of software domains – for example the financial industry, television and radio broadcasting, hospital management and rental business – is steadily increasing and knowledge management of businesses is becoming more important with the demand for capturing business processes. Software domains such as the ones mentioned above, are inherently knowledge-intensive but this domain knowledge is often hard to detect in the resulting application. This is because the domain knowledge is not separately and explicitly dealt with, but implemented together with the implementation strategy of the application. This also results in changes to the domain knowledge propagating to the implementation strategy. Hence, current software engineering practices result in software applications where domain knowledge is implicit and tangled in the implementation strategy, thus encumbering understandability, maintenance, adaptability, and reuse. A second problem is that the development of software where domain knowledge and implementation strategy are tangled is a very complex task: the software developer who is typically not a domain expert has to concentrate on two aspects of the software at the same time, and moreover manually compose them. This violates the principle of separation of concerns [5] [9], that states that the basic algorithm should be separated from other concerns or aspects.

A category of applications where separating explicit domain knowledge would be advantageous is concerned with software for the support of a real-world task and management of real-world information, such as software for banks, hospitals or television broadcasters. For example, in the latter domain there exists software for the support of seasonal and daily planning of the broadcasts. Such an application typically consists of domain knowledge about the kinds of programs that exist (films, series, commercials) and planning rules and constraints (children’s programs should be scheduled before 8pm, moving a program results in moving the commercials that are scheduled in its breaks, a film should be broadcast in the period for which its contract is valid). This domain knowledge is typically tangled with the implementation strategy, which is concerned with for example the user interface and persistency. These are appropriately implemented using object-oriented programming and techniques such as design patterns (model-view-controller, bridge) [8], whereas the domain knowledge ideally should be expressed in a declarative medium using rules and constraints.

Another example is distributed applications, where the domain knowledge of the application and distribution issues such as transactions, replication, remote procedure calling and concurrency control are also difficult to separate.

The goal of this research is to express domain knowledge in software applications explicitly and as separated as possible from the implementation strategy. Although some (domain) knowledge is notoriously hard to elicit and capture, as was discovered in building expert systems, the domain knowledge we intend to make explicit is quite tangible as is illustrated by the aforementioned examples. In fact, the domain knowledge is currently “implemented” using a (object-oriented) programming language. When expressed in a suitable medium, domain knowledge consists of concepts and relations between the concepts, constraints on the concepts and the relations, and rules that state how to infer new concepts and relations [12].

We are inspired by Aspect-Oriented Programming [10] [1], where aspects such as error handling, error reporting, synchronisation and so on, are expressed in an aspect language separately from the implementation strategy. A weaver composes the aspect with the implementation strategy which results in an executable program. A weaver uses join points which indicate places in the implementation strategy where the aspect should be inserted. An original contribution of this research is to consider domain knowledge as an aspect [6] [7]. We believe that in some cases the “weaving” of domain knowledge and implementation strategy will be quite straightforward, but that in others it will benefit from applying ideas if not actual techniques from aspect-oriented programming.

Our research hypothesis is Expressing domain knowledge of a software application explicitly and separately in a suitable medium alongside the implementation strategy of the software application which is expressed in a standard (object-oriented) programming language will improve software understandability, software maintenance and software reuse. Although it is difficult to test subjective properties such as improved understandability, it was already shown in [15] that an explicit model of the domain knowledge achieves
1. REFERENCES


