
Business Modeling for OT Systems – Workshop 28 OOPSLA'97

Organizers: Steffen Schaefer, Simon Horner

Position paper by Michel Tilman, System Architect, Argo Belgium

mtilman@argo.be

<http://www.argo.be/OoFrame>

Papers of related interest

Martine Devos and Michel Tilman, Design and Implementation of a Business Modeling Framework using Smalltalk, Object Technology'96, 1996

Martine Devos and Michel Tilman, Business Modeling using an Object-Oriented Framework and Meta-Repository Architecture, OOPSLA'96 Demonstrations, 1996

Martine Devos and Michel Tilman, Object-Orientation and Evolutionary Software Engineering, Position paper for the OOPSLA'96 Workshop on Object-Oriented Software Evolution and Reengineering, 1996

Martine Devos and Michel Tilman, Application development using a Meta-repository based Framework, ECOOP'97 Demonstrations, 1997

Martine Devos and Michel Tilman, Application development using a Meta-repository based Framework, OOPSLA'97 Demonstrations, 1997

Martine Devos and Michel Tilman, A Framework for Adaptive Design, OOPSLA'97 Posters, 1997

Martine Devos, System Envisioning, Position paper for the OOPSLA'97 Workshop on System Envisioning, 1997 (submitted)

Context

This paper describes some experiences derived from the design, development and use of our framework to support the Argo administration with database, electronic document management, workflow and Internet applications. We refer the reader to our web-site for more background information, in particular to the 'A Framework for Adaptive Design' and 'System Envisioning' papers.

Experiences

Each organization has unique and evolving requirements. And even within a single organization we have to accommodate both global consistency and tools adapted to the skills and needs of individuals and teams. In our framework we separate the business model from the functionality of end-user, configuration and modeling tools. These tools consult the meta-information of the business model (stored in a central repository) at run-time, leading to easier maintenance, increased re-use and better support for evolution of both business model and tools. The reflective nature of our approach gives us considerable freedom and flexibility in modeling (e.g. in the use of static vs. dynamic types), yet how to achieve this power is poorly supported by object-modeling techniques, and but to a limited degree by some pattern languages.

Although we use Smalltalk throughout the framework implementation, we do not 'think' object-oriented for all parts of the framework. For instance, we specify authorizations by means of a rule-base with a dedicated inference engine. And we do not request extensions to the basic object model, such as subjectivity, to give our users appropriate views on the object model. Rather, we combine our tools to filter or enhance the user's perspective on the business logic.

Sometimes we want to enforce global constraints that, ideally, should be satisfied at all times, but do not turn out to be workable when we take into account business policies or actual working practices. These constraints need to be satisfied in due time, however, hence they must be checked by an auditing tool. This is typical of the static nature of most object-modeling techniques, as is the absence of business rules. We regularly use business rules to add more application-specific semantics to the global object model. We

implement these rules by means of 'action' rules, which are essentially object-oriented in nature, with a slight declarative flavor.

Not all knowledge of the business organization, processes and rules can equally well be formalized into a (consistent) model. In this context, our repository approach plays another important role, by keeping track of both formal (e.g. the object model) and informal (e.g. custom scripts) knowledge of the business 'model'. A continuous process of stepwise structuring enables us to evolve meta-model, business model and framework tools by 'formalizing' latent patterns in the knowledge base. For instance, by using the existing configuration and modeling tools, we can extend the meta-model by adding new types of constraints or we can enhance the basic authorization mechanism. This idea of bootstrapping the system is important, and even feels very natural in our approach. At this, we rely heavily on the reflective powers of our framework.

Many users have no clear idea of what (new) technology can achieve, hence they tend to describe their problems in terms of familiar concepts. To support our users in this learning process, we basically rely on two techniques. System envisioning encompasses several creativity techniques, such as the use of metaphors. The other, better known to software engineers, is the use of prototypes, but, even so, prototyping is mostly applied to validate the design, rather than to act as idea enabler. Since our framework diffuses the boundaries between prototyping, application specification and development to a large degree, we can build 'prototypes' rather cheaply. And since we do not need to throw away these prototypes later on, we consistently apply a multi-prototype approach for all our end-user application development.

In developing end-user applications we have largely dispensed with class diagrams as a primary communication medium with the users, partly because our prototyping approach has proven to yield more useful and valid results, partly because many users failed to fully grasp the real meaning or impact of the model presented. We do use class diagrams of the business model: they are generated automatically, and are mainly used to validate or fine-tune our design and to document the applications. We increasingly use scenario diagrams as they reflect more familiar and directly observable (especially from the user's point of view) aspects of the business processes. In this context we are also exploring the use of CRC-like techniques.

We apply several object-modeling techniques within the development of the framework, such as class, interaction, scenario and state diagrams. However, we do not use all of these techniques to analyze, design and document each and every class in the framework. Rather we use the most appropriate techniques to help us better understand the global design (roles, responsibilities, inheritance, collaborations, important message sequences and lifecycles) or to highlight particular technical issues. For architectural purposes, we often use 'abstract' diagrams, defined in terms of 'components' rather than actual classes, and in terms of abstract responsibilities and messages. This liberal use of familiar object-modeling techniques is sometimes useful to hide irrelevant detail or technical issues, for instance when interface and implementation inheritance hierarchies conflict, as this distinction is not well supported by most object-modeling techniques. Our primary criterion to use a particular technique is to get a good understanding in a clear way. When the diagrams become too detailed (or sometimes even too concrete), their complexity tends to increase and their usability drops accordingly.

Existing object-modeling techniques offer a rather black-and-white view on interfaces (access being e.g. either public, private or protected), thus failing to give support for more sophisticated contracts. Security aspects and trusted components, for instance, are increasingly becoming important (e.g. in distributed computing and when dealing with performance issues as in open implementations).

We increasingly incorporate pattern language vocabulary as part of our regular (textual) documentation, typically by using pattern names as verbs. This helps us write more concise documentation with a well-understood meaning, provided the patterns are (re-) useable and have a clearly understood application 'domain'.

References

We refer the reader to the references in the papers listed above. These papers are available on our web-site.