

Object-Orientation and Evolutionary Software Engineering

Martine Devos, IS Manager, Argo (mdevos@gate.argo.be)

Michel Tilman, Senior Systems Specialist, Unisys Belgium (mtilman@gate.argo.be)

1. Abstract

The first half of this paper describes some of the issues regarding re-use and evolution of software within the context of the ever-changing organization. It draws on experiences which arose during the development of an object-oriented framework (BoA) for modeling the Argo organization, and which have led to the following observations:

- Different organizations or applications within the same organization have specific requirements which must be met at the appropriate level.
- Different aspects of a project may have specific degrees of and solutions for re-use, and may exhibit quite distinct evolutionary needs.
- A suitable architectural design is needed in order to support each aspect of the project implementation in an appropriate way.
- Straightforward OOA/OOD approaches will, in general, yield less than optimal solutions with regards to re-use and evolutionary needs, as these tend to be identified too late or too isolated during the project implementation, with little support for applying more advanced techniques during analysis and design phases.

The remainder of the paper will focus on some of the solutions which resulted from the project implementation, and on some issues to be handled in the forthcoming years.

2. Context

The BoA framework was developed by Argo and Unisys Belgium. Argo is responsible for the overall management of Public Schools (non-denominational) within the Flemish community of Belgium. Argo consists of a central administration and about a hundred semi-autonomous local boards and 730 schools.

The organization itself is subject to a Business Process Re-engineering project, which aims at restructuring the organization and its major processes, in addition to re-thinking its services towards the customers. Furthermore, the overall requirements of the Argo project are typical for many large administrations, which need support for both structured and unstructured information and for both formal and more groupware-like process structures.

Given this situation, it should come as no surprise that a suitable architecture could and should be designed, in which the customer's requirements for an evolutionary solution closely parallel the needs for a solution which is re-usable across different organizations.

3. Some problems with current approaches

Traditional object-oriented project implementations tend to flatten out the differentiation between various aspects of the problem domain when moving to a computer model. This applies as well to problems to be solved for customers with similar needs as to problems to be solved within the context of a continuously changing organization.

3.1 Problem domain issues

While Business Process Re-engineering (BPR, although Business Process Engineering would often seem to be more appropriate) has been heralded as a means to increase the business productivity or quality to varying degrees, or even to re-assess the business goals and objectives, little has been done in the way of software engineering to support the process of re-thinking and re-implementing the changes implied by the BPR process. Approaches such as using, say, graphical interfaces for defining workflow procedures won't really solve the problems. Indeed, whenever a BPR process is set up, several things may change within the organization, at different degrees and possibly in different time-frames, both with regards to the organization structure, the kinds of problems the software should support, and the way the software should be used. For instance, in the process of moving from a

traditional hierarchical structure to a flatter, less static organization with more local responsibilities, existing processes within the Argo administration were analyzed and re-designed. The original overly sequential processes were replaced by much simpler processes, with more emphasis on delegation of responsibilities and parallel teamwork. At the software level, this does not only require a change of the business model, but a move towards a more groupware-like approach at supporting business processes.

This implies that the project implementation must carefully differentiate between:

- what is specific for a particular project or organization and the more generic functionality
- what parts of the solution are re-usable and in which context
- what parts of the solution are dependent on evolutionary needs and to what degree
- how each part of the software solution contributes to increased support for re-use and evolution of the various parts of the problem domain.

Suitable methods and software tools should not only help manage the process changes, but could play an equally important enabler role as well: given the right degree of flexibility they could allow organization and users alike to discover new ways of contributing to the business goals, be it at the level of more global processes or of the end-user's way of working.

3.2 Object-orientation issues

Object-orientation has been widely promoted as the means to tackle the complexity and scale of today's and tomorrow's applications, relying on techniques such as encapsulation, inheritance and more 'natural' OOA/OOD methods as the basis for increased software engineering quality and productivity, through increased modularity, re-use and abstraction. Yet, in several regards, many object-oriented project implementations are rather low-level:

- While the proper use of object-orientation has clear benefits, the end-user is not really interested in objects, but wants a solution -now and in the future- to his or her problems, based on requirements stated in relevant terminology. To illustrate this, when concepts such as views are being discussed, OO-developers often tend to focus too soon on low-level -at least from the end-user's perspective- implementation issues, such as a new paradigm for object-oriented languages, rather than into requirements for, say, a more tangible framework component or functionality. Hence, in general, the end-user must understand the language of the developer, typically in the form of an OMT-schema or the like. And while the use of scenarios indicates a trend towards a more direct capture of the end-user requirements, more domain-specific analysis techniques with suitable terminology and tools may be appropriate. This may help narrow the gap between end-user and developer regarding current and future needs.
- Little support (and even explicit know-how) exists for the evolutionary character of software engineering, be it in the form of re-usable software which can be applied across similar applications or in the form of easily maintainable solutions. This is exemplified by a typical straightforward OOA/OOD process, in which a class is being defined for about each concept in the universe of discourse, and where re-use is often applied at too late a stage, i.e. at the actual implementation level, when programmers start looking for re-usable assets. However, re-iterating over these straightforward approaches, for instance through the proper use of meta-level or reflection techniques, or through appropriate techniques for detecting less evident architectural patterns (such as hidden typing and generic behavior), often helps identify more abstract architectural and re-usable components as well as more dynamic and long term relationships between those components. Current OOA/OOD methods do not adequately support such more architecturally oriented ways of working.
- As already mentioned, re-use can (and should) be applied much sooner in a project implementation, for instance within the context of a BPR project through scenarios about an organization's likely evolutionary paths and the resulting impact on requirements. Such an approach will typically need to be substantiated by an architecturally sound approach towards solving a customer's needs.
- Often, people are looking for re-use in what we feel to be the wrong place, for instance, by searching for generic and re-usable business objects in the form of THE generic organizational structure. Typically, this results in objects which are too abstract to be of any real use within a particular organization or which require lots of parameters. Specific types of organizational structures, on the other hand, can be more readily re-usable within particular application domains, such as well-defined vertical market segments.

- As already noted, iterative re-factoring may yield a cleaner, more orthogonal design, offering better opportunities for re-use and evolution. Failure to do so at appropriate times in the course of a project often causes objects to accumulate more or less spurious responsibilities, resulting in fuzzily defined behaviors. These objects, however, may absorb some degree of change more easily than over-engineered designs: factoring out responsibilities too soon, for instance, may result in artificial designs, which are too much tied to the current situation, and hence, not very resilient to change. Given decent measure theories, however, appropriate tools can supplement the traditional software engineering process, by keeping track of changes and their ramifications on the design, leading to hints on when a particular design starts to degenerate and to which degree.
- The software design should contribute in a more discriminate way towards the end-result. That is, some aspects of the problem domain could be handled better by, say, a high-level model, a framework or a set of rules, depending on current requirements and evolutionary needs. In practice, this will often lead to the elaboration of an architectural design with a clear target, where all the components and tools contribute to the various aspects in a suitable way, and where each component can evolve in a timely manner with the rest of the software. Once more, we feel straightforward OOA/OOD implementation to flatten out the roles and specific requirements for the major components of the problem domain.

4. The Argo project implementation

Within the context of the Argo project, an architectural solution has been designed and implemented, which we feel solves some of the problems mentioned above. This is exemplified by the fact that, somewhere in the second half of the project (when several applications were already up and running), the long-pending reorganization of the central administration was effectively carried through. This implied a major revision of the business logic. Yet these changes were put into place without any impact on the actual application software.

The use of a meta-repository in combination with a generic framework (as described below) allows to shorten the gap between end-users and developers, which leads to a rather drastic change in the development life-cycle. In the longer term, we feel it to offer better support for both 'natural' software evolution and Business Process Re-engineering projects.

4.1 The BoA architecture

BoA is an object-oriented framework developed in VisualWorks\Smalltalk for modeling the business logic of a wide variety of organizations and application domains, making as few assumptions as possible about particular business models. It offers a set of generic tools for defining, customizing, managing and maintaining the business objects pertaining the workings of a particular organization, be it in the context of database applications, electronic document management or workflow. Its main business areas are large administrations which need to cope with several applications sharing a common business logic.

The key features of the BoA architecture are its open-ended nature, relying on an extensible framework of re-usable components containing the generic application logic and on a meta-repository containing descriptions of the actual business logic, which is used at run-time by virtually all components of the framework, such as the authorization rules. This allows applications to be modeled and configured, rather than hard-coded, and to let applications evolve more easily with changing needs or organization, be it at the level of functionality or business model. At this, it relies on the following observations:

- Whereas the business logic is particular to each organization (or a well-defined vertical market segment, in which case we can actually re-use the business logic), the end-user or administration functionality (such as data-entry, querying, reporting, document management and in / out baskets) is all the more generic.
- Extensions or enhancements to the functionality can often be made into re-usable assets, independent of the actual business model.
- The business model should be allowed to evolve, with limited impact on the application logic and vice versa.

This has resulted into the following approach for the BoA framework, which:

- allows to formalize business structures, data, relationships, processes and rules particular to a specific organization (the *datamodel* of the organization) by modeling and configuration
- uses a meta-repository to store this datamodel

- separates this formalized description of the datamodel from the application logic
- offers a generic end-user application as well as administration, authorization and configuration tools for managing both structured and unstructured data as well as workflow, requiring only the business model to get the system up and running.

The benefits are manifold:

- Applications, as perceived by end-users, represent coherent views on the common business logic, which allows to share and re-use common business objects within the organization.
- Using a meta-repository in combination with a ready-built generic application containing the necessary functionality bridges the gap between development team and end-users. This results in an iterative and incremental style of working, and supports rapid application deployment. As such, setting up the specifications for a default application is, in fact, synonymous to building the working application itself. This allows project implementers to focus more on business consultancy, e.g. within the context of a BPR project, rather than on development.
- In contrast to most CASE-tool approaches, the datamodel is not used to *generate* an end-user application, but is *consulted* in a dynamic way by generic end-user application, tools and framework components, which allows for increased flexibility (such as enhancing the functionality of the end-user application without having to modify the existing business model or vice versa) and maintainability.
- As all components, tools and end-user application, as well as structure and functionality of the meta-repository are part of the same framework environment, they can evolve over time within the framework as needed. This allows, for instance, to add new components or functionality to be incorporated in the framework, and to cope with future technologies and media.

Another important design goal of the framework was to allow a kind of *bootstrapping*, whereby administration and configuration tools can gradually be expressed in terms of the framework itself.

4.2 Analysis and design

From the analysis point of view, up until now, rather traditional methods have been used, supplemented with user scenarios (not based on any explicit method, such as USE Cases, however).

In the future, more emphasis will be placed on the analysis and design aspects, including the possibility of using evolutionary scenarios within the context of Business Process Re-engineering. In particular, the framework will be used to a much larger degree as a learning tool, whereby users and management can discover new ways of setting up and managing business processes through the use of the system. Additional high-level modeling and configuration functionality, augmented by the bootstrapping process, will be provided to further support this particular use of the system.

4.3 Documentation

Another important issue is the need for decent documentation, and to keep it up-to-date.

One of the approaches consists of generating (at least part of) the documentation out of the available information (such as the datamodel and semi-structured class comments). Some work has already been done (e.g. generating NIAM-schema diagrams out of the datamodel specification), but in the future, more work remains to be done in this area (for instance, generating explicit workflow process diagrams out of rule-like descriptions of the workflow processes). One of the main themes in this approach, is to generate as much useful documentation -and, more generally, information about a project implementation's state, such as its consistency and various metrics- as possible out of a minimum of explicit user input (i.e. not having to use tools for tools' sake).

We plan to use our own system to model a technical documentation application, which keeps track of all our technical documentation, be it analysis, design, implementation or test reports. While the end-user functionality allows to manage relationships and versions, using the system itself will, in addition, help to achieve support for a larger degree of consistency. For instance, by setting up appropriate rules, specific documents could be invalidated or new versions created whenever some part of the datamodel changes. This becomes all the more interesting as the bootstrapping process evolves, whereby, for instance, hard-wired administration tools are gradually being replaced by configured applications: in this way, consistency of administration tools documentation can partly be supported by the system as well.

5. Acknowledgments

Thanks to Argo and Unisys Belgium, and to Thanh Son Du, Els Goossens and Danny Ureel (Unisys Belgium) in particular for commenting on this paper.