

An Object-Oriented Frameworks-based Architecture for Decision Support Systems*

Karin Becker

Instituto de Informática

Pontifícia Universidade Católica do Rio Grande do Sul

Porto Alegre - RS - Brazil

kbecker@inf.pucrs.br

François Bodart

Institut d'Informatique

Facultés Universitaires Notre-Dame de la Paix

Namur - Belgium

fbodart@info.fundp.ac.be

Abstract

Reusability is considered to be the key for achieving productivity and quality in software, and much has been claimed about the particular contributions of the object-oriented paradigm towards the achievement of these goals. Object-oriented frameworks are coarse-grained reuse units, composed of a set of classes specifically designed to be refined and used as a group. In this paper, we discuss the nature of frameworks necessary to build a particular type of systems, namely Decision Support Systems (DSS), and their organization in a generic OO DSS multi-layer architecture. DSS are systems intended to improve the effectiveness of decision making, but information technologies can only have a major impact on decision making if techniques allowing the easy and rapid development of DSS are available. Much benefit is expected in terms of easiness and rapidity of development by constructing DSS from domain-oriented reusable components, as well as in terms of quality of DSS in this way developed.

1 Introduction

One of the persistent challenges faced by managers is to comprehend and respond in a timely manner to the opportunities and threats of the market place with decisions that will make the best use of corporate resources. Decision Support Systems (DSS) are computer-based systems intended to improve the effectiveness of decision making, by helping decision-makers use models and data to solve semi-structured to unstructured decision problems [46, 47]. Major problems identified in early DSS designs were: 1) the systems were so time consuming and costly to develop and maintain that they were difficult to adapt to rapid changes in organisation's decision support requirements, and 2) they were too user and problem specific. From these realizations, and aided by rapid advances in information technology, much of DSS

*An earlier version of this paper was presented at CLEI'96, Colombia.

research has been oriented towards the investigation of more flexible approaches for the easy and rapid development of DSS.

Despite their differences, most propositions found in the literature aim at the discovery of some degree of *genericity* and *reuse* as the foundations for the flexible and rapid development of DSS. Two major trends can then be identified since late 1970's: DSS generators and Generalized Model Management Systems (MMS). DSS generators [47], such as spreadsheets, financial modeling languages, MS/OR packages, etc., are implementation tools targeted at reducing the time, cost and effort involved in implementing and maintaining a DSS. The focus on Generalized MMS [10, 17, 33, 38] is rather to extend the scope of DSS, so as to allow the support of multiple problems using a variety of models and data sources, as well as the centralized management of models as an organizational resource. In both approaches, the underlying genericity for DSS development is *function-oriented*. DSS generators enable the reuse of high-level implementation functions targeted at the requirements for DSS development. The reuse promoted by Generalized MMS is at the level of generic facilities for development, storage, manipulation, control and effective utilization of models as a corporate resource of an organization, in an attempt of evening it up with Database Management Systems (DBMS), which offer the same facilities for data.

The easy and rapid development of computer-based systems is a concern shared by the Software Engineering community with regard to the development of applications in general. *Reuse* is regarded as the key for improving quality and productivity of software development [8], with a primary focus on the potential contributions of the object-oriented (OO) paradigm and derived methods and technologies [37, 40, 30, 31]. Frameworks are an object-oriented reuse technique. A framework [16, 52, 28, 31, 40, 22] can be regarded as a high-level application or subsystem architecture, consisting of a set of classes that are specifically designed to be refined and used *as a group*. A framework represents a generic solution for the development of applications in a specific domain (e.g. graphical editors [29], operational systems [13], graphical user interfaces [34]), which can be customized to meet the particularities of the problem at hand. Being more coarse-grained reuse units than individual classes, frameworks are expected to promote *application component-oriented reuse*, by addressing the development of applications as the activity of *refining* and *binding* pre-defined, plug-compatible components that are truly application oriented [40]. The development of specific applications based on the reuse of frameworks induces a new application development life-cycle, in which two distinct activities can be distinguished [23]: 1) *Application Engineering*, related to the development of frameworks, and 2) *Application Development*, where frameworks are customized to develop specific applications.

The OO paradigm has lately attracted the attention of the DSS research community, but more emphasis has been given to the benefits of the unifying characteristics of this paradigm in the design of DSS [19, 36, 38, 39]. We have been investigating the potential contributions to the DSS field of the *domain-oriented genericity* underlying OO. The issue of decision problem formulation and modeling based on domain-oriented components has been addressed in [4, 5], where frameworks were used to represent classes of decision problems (e.g. industrial activity planning, capital budgeting), expressed in terms of the prototypical decision elements. This approach is intended to decision-makers as a *modeling tool*, since a framework, regarded as a generic decision model for problems of a certain class, can be customized through an *instantiation process* to represent the particular decision problem at hand. Specific models can be constructed by selecting, adapting and combining *problem-domain concepts*. The methodological role played by a framework based modeling environment is discussed in [6].

In this paper, we go one step further, considering the development of DSS that provide such modeling capabilities, through the reuse of OO frameworks. Frameworks for DSS development in the domain of financial instruments analysis are reported in [20, 9, 1]. Their experiences confirm that application engineering is a hard task for which presently no consistent formal support exists, and therefore subject to much empiricism and creativity, and doomed to a trial-and-error process. As an attempt to manage the complexity of framework development and reuse in large scale industrial banking projects, [2] proposed the Gebos System, an architecture in which distinct types of frameworks are organized in 5 distinct layers: business domain, business section, application, technical kernel and desktop. The Gebos System makes it possible to configure and adapt new application systems in a comparatively short time [2], since it identifies and organizes frameworks with loose coupling and fewer dependencies.

In restricting our concerns to the DSS context, we have also identified a number of common criteria for DSS development based on frameworks reuse. The contribution of this paper is the analysis and description of the different natures of frameworks that can be combined for developing DSS, and their organization as a generic OO DSS multi-layer architecture, aiming at a less hazardous approach for performing the application engineering and application development activities in the DSS domain.

The rest of this paper is organized as follows. Frameworks are discussed in Section 2. The striking characteristics of target DSS, i.e. the type of DSS that can adequately be developed through the reuse of the suggested reusable components, are presented in Section 3. The distinct types of reusable components for target DSS are then detailed in Section 4. In Section 5, the application development of DSS is briefly discussed, and Section 6 draws conclusions.

2 Frameworks

Object-oriented application frameworks are a promising reuse technology for reifying proven software designs and implementations [22, 31]. A framework provides a generic solution for a given class of problems, which can be customized to meet the particularities of the problem at hand. A framework can be regarded as a high-level application or subsystem architecture, consisting of a set of classes that are specifically designed to be refined and used as a group. A framework concentrates on describing the kind of objects that make up the ensemble and their protocol of interaction, i.e. how they cooperate with each other to accomplish the overall objective of the application. Most classes composing a framework are abstract [41], i.e. they serve as template for other classes (concrete classes), and not for objects. The functional interface of abstract classes is essentially defined by abstract operations, for which code is actually provided in concrete classes. Though code can equally be provided in the classes constituting a framework, the essential role of a framework is to describe the way a system is partitioned into components and their interfaces. In this way, the users of frameworks, i.e. application developers, can concentrate on refining and combining components. This is the key insight behind frameworks [31, 40, 16].

The first and most successful frameworks were in the general domain of graphical user interfaces (GUI). Well-known frameworks in this domain are MVC [34], MacApp [45], ET++ [51], Interviews [35], Microsoft Foundation Classes, to mention just a few. Frameworks have been developed as well for a variety of other domains, such as graphical editors [29, 49], document editors [12], operational systems [13], network and distributed applications [27, 7],

manufacturing [43], banking and financial engineering [1, 2, 9, 20], etc.

Definitions for frameworks vary, as researchers and practitioners have difficulty on recognizing the similarities and differences with regard to other reuse techniques in general, and OO reuse techniques in particular [48, 32, 22, 31]. For the purposes of this paper, it is interesting to adopt the proposition of [22], which classifies frameworks according their purpose as *system infrastructure frameworks* (e.g. operating systems, communications, GUI), *middleware integration frameworks* (e.g. implementations of OMG standards, message-oriented middleware) and *enterprise application frameworks*, targeted at specific application domains (e.g. telecommunications, manufacturing, financial engineering). The first two focus largely on internal software development concerns and are basically application domain-independent, whereas the latter is targeted directly to support the development of end-users applications and products in specific domains.

Relative to system infrastructure and middleware integration frameworks, enterprise frameworks are the most difficult and expensive to develop and commercialize [22, 31]. Indeed, designing a framework is like developing a theory about application development, that can be tested only by trying to reuse the framework. While developing software is difficult enough, developing high quality, extensible and reusable frameworks for complex applications domains is even harder [24, 52], in particular when no standards or normative ways of developing applications exist for such domains.

OO and framework design principles [28, 42, 44, 15, 32] and design patterns [24, 31, 41, 14, 50] are important trends towards reducing the complexity of frameworks development and (re)use. It has been recognized that black-box frameworks are easier to reuse than white-box frameworks [28, 22]. Whereas white-box frameworks rely heavily on OO programming language features like inheritance and dynamic binding, black-box frameworks support extensibility by defining interfaces for components that can be plugged into the framework via object composition and delegation. Black-box frameworks are easier to reuse because they do not require from application developers an extensive knowledge about how the framework was developed in order to be reused, but they are much harder to design since interfaces for a wide range of use cases must be anticipated. Patterns have become a popular way to reuse design experience that cannot be expressed in terms of components, and have been extensively used as a guide for the development of more reusable frameworks. As for frameworks, definitions for patterns vary, influenced by issues such as granularity, domain (in)dependency and notations for patterns description. Other research issues on frameworks are documentation, learning curve, development and maintenance processes, framework economics, framework standards, interoperability, etc [22, 31].

3 Striking Features of Target DSS

No generic solution can be constructed without a clear understanding of the nature and characteristics of the specific solutions that one wishes to be derived from it. DSS are difficult to define, and many are the sources of disagreements for their definition [46]. A generally agreed upon functional architecture for DSS is the frame of reference proposed by [47], which identifies 3 main functional components, namely the *model component* (MMS), the *data component* (DBMS) and the *dialogue component*. DSS following this functional architecture are referred to as *model-oriented DSS*.

Acceptance of (model-oriented) DSS by managers as a decision-making supporting tool is

still limited. Model formulation is typically performed by experts, and managers often feel reluctance on using models they do not fully understand, and in which development they had little participation [18]. One of the major goals in the DSS research is the provision of modeling environments for users who are not modeling specialists [21, 46]. The reusable components discussed in this paper are targeted at the development of *DSS supporting modeling capabilities* for non-experts.

The basic idea is that, by using application component-oriented components to construct the modeling capabilities of DSS, the domain concepts represented by those components become also apparent to *DSS users* (i.e. decision-makers), such that model formulation, from their point of view, becomes the simple process of choosing and applying a set of special-purpose, domain-oriented concepts for describing the decision situation at hand, in a process fully compatible with their profiles, as described in [4, 5, 6]. This and other features of target DSS are further considered below. For illustration purposes, a very simple example of specific DSS that could be constructed according to the proposed approach is considered. Figure 1 presents some elements of the modeling facilities of a simplified DSS for capital budgeting decision problems.

- **modeling paradigm:** decision-makers should be able to express their specific problems in a conceptual level, directly in terms of decision elements of the problem and independently of resolution and implementation concerns. This is possible if domain-oriented building blocks used to construct the DSS are offered to decision-makers as *modeling concepts*. Model modeling becomes then the activity of customizing a generic decision model for classes of decision problems, by the instantiation of concepts and binding of those instances [5]. For example, the Capital Budgeting class of problems [11] involves decisions about investments whose returns are expected to extend beyond a year. The lifetime of a project can thus span over several years, and it implies forecasting all incomes and expenditures incurred in this period. The profitability of an investment can be evaluated by a number of criteria, such as the Net Present Value (NPV), Internal Rate of Return, Payback Period, etc. Models for this class of decision problems can be structured and formulated in terms of concepts such as *lifetime* of the investment project, *time-schedules* of *incomes* and *expenses*, *cash flow*, *profitability criteria*, *cost of capital*, etc. Figures 1.(a,c) show some of those concepts, and Figure 1.(b) depicts the instantiation of an object *time* by a user.
- **visual representation:** domain concepts are given a graphical representation, using familiar presentation structures, such as table, graph, report, chart, etc. It is through this visual representation that users can create and manipulate instances of model concepts. For example, a tabular form is the most frequent graphical representation for representing Capital Budgeting problems, with columns representing the concept of time, whereas the rows would represent various concepts such as cash flow, profitability criteria, etc (Figure 1.(a)). Instances of those concepts would be created as a result of the creation of columns and rows (Figure 1.(f)). The inherent hierarchical structure among budgetary items, used to describe the incomes and expenditures of an investment project, can be represented using a tree structure. The generic item tree presented in Figure 1.(c) shows to the user the nature of the items that can be specified, and the valid relationships between them. A customized item tree is shown in Figure 1.(d). For instance, the “working cash flow” is the balance between the “results” of the project, and the “taxes” that must be paid.

