A Layered Architecture Sustaining Model Driven and Event Driven Software Development

(extended abstract - short talk)

Cindy Michiels, Monique Snoeck, Wilfried Lemahieu, Frank Goethals, and Guido Dedene

MIS Group, Dept. Applied Economic Sciences, K.U.Leuven,
Naamsestraat 69, 3000 Leuven, Belgium
{Cindy.Michiels, Monique.Snoeck, Wilfried.Lemahieu, Frank.Goethals, Guido.Dedene}@econ.kuleuven.ac.be

Abstract. This paper presents a layered software architecture reconciling model-driven, event-driven, and object-oriented software development. In its simplest from, the architecture consists of two layers: an enterprise layer consisting of a relatively stable business model and an information system layer, containing the more volatile user functionality. The paper explains how the concept of events is used in the enterprise layer as a means to make business objects more independent of each other. This results in an event handling sublayer, allowing to define groups of events and handling consistency and transaction management aspects. This type of architecture results in information systems with a high-level modular structure, where changes are easier to perform as higher layers will not influence the inherently more stable lower layers.

1. Introduction

Separation of concerns can be pursued at different levels of abstraction in the software development process. In this paper we present a layered software architecture that represents a separation of concerns at a high level of abstraction: it classifies specifications in different layers according to a model-driven approach such that each layer only relies on concepts of the same layer or on concepts of the layers below. Such architecture results in information systems accounting for a high-level modular structure, where changes in higher layers will not influence the inherently more stable lower layers. At the same time, an event-driven approach is advocated as interaction paradigm for objects in the lowest layer: their interaction is modeled by identifying common involvement in business events. As such, these objects can be added and deleted from the bottom layer with no risk for severe maintenance problems, as is often the case in software development methodologies using method invocation as the main interaction method.

2. Model-driven development and basic layering

Model-driven development finds its origin in a reframed version of Zachman's Information Systems Architecture [1], which states that software requirements should be captured in different models according to their origin. As such, the Zachman framework recognizes four basic levels of abstraction in the software development process: the scope model, the business model, the information system model, and the technology model. Orthogonal to these basic abstraction layers, it identifies a number of aspects interacting with them, such as goals, data, process, and people. One of the main benefits of model-driven development is that it imposes a high-level modularity on the developed system: the inherently more stable business model can act as a foundation layer for the more volatile information system model. This facilitates maintenance dramatically.

The architecture presented in this paper builds upon model-driven development and as such retains two basic abstraction levels: the business model and the information system model. The scope model, defining the enterprise's direction and business purpose, is considered as a matter of ICT strategy development and is as a consequence left out of consideration [2]. The technology model, describing how technology may be used to address the information processing needs that were identified, is considered as an aspect orthogonal to the business model and the information system model, as different technology choices can be made for the realization of different layers [3]. One approach to deal with technology aspects will be the combination of code generation facilities with the reuse of patterns and frameworks [4].

The business model defines the fundamental business requirements using a set of business objects, business methods, and business rules to capture the entirety of the business. The Business Layer does not build upon an information system and is also valid if no information system is defined. The information system model is captured by an Information System Layer that actually builds upon an information system and covers all services related to the requested input and output functionality to the information system users. More precisely, the Information System layer is considered as a layer on top of the Business layer, because only the former is allowed to invoke services on the latter, and not the other way round (see Figure 1). In adhering to this basic layering and to the principles of object-orientation, a flexible architecture is obtained, because input and output services can be plugged in and out of the Information System Layer with no side effects on the objects in the Business Layer.

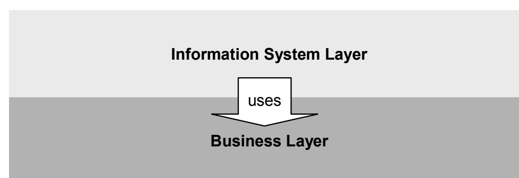

Fig. 1. Basic layers identified in architecture

3. Imposing an event-driven approach

The introduction of a Business Layer, relating to the more stable business requirements, and an Information System Layer, capturing the more volatile input and output services of an information system, imposes a basic modularity upon the information system architecture. In this section it is argued that the modularity and, as a consequence, the flexibility of the architecture can be further enhanced by imposing an event-driven approach on the basic layering. This will result in a refinement of the Business Layer and the Information System Layer to account for business events and information system events respectively.

In most object-oriented approaches events are considered as subordinate to objects, because they only serve as a trigger for an object's method. The object interactions themselves are modeled by means of sequence and/or collaboration diagrams often involving long sequences of method invocations. As such, the specification of the dynamic behavior of a system can result in a complex network of method invocations between interacting objects. The main problem of this approach is that the addition or deletion of an object can involve severe maintenance problems, because all interaction sequences the object participates in are to be reconsidered.

In contrast, an event-driven approach raises events to the same level of importance as objects, and recognizes them as a fundamental part of the structure of experience [5]. A business event is now defined as an atomic unit of action that represents something that happens in the real world, such as the creation of a new customer, an order placement, etc. Without events nothing would happen: they reflect how information and objects come into existence (the creating events), how information and objects are modified (the modifying events), and how they disappear from our universe of discourse (the ending events). Object interaction can now be modeled by defining which objects are concurrently involved in which events. Object-event participations can be denoted by means of an Object-Event Table [6]. When an object participates in an event, it implements a method that defines the effect of the event on the object. On occurrence of the event all corresponding methods in the participating objects are executed in parallel. As such, the addition or deletion of an object will only involve the addition or deletion of all event participations of that object, without implications for other objects.

4. Implications on basic layering

The Business Layer can now be divided in two sublayers: the business objects, encapsulating data and behavior and defining business rules, constitute the lower sublayer. The business events, managing business object interaction, constitute the upper sublayer (see lower part of Figure 2). Also the Information System Layer is split in two sublayers. The lower one is composed of information system transactions, managing a logical grouping of business events and of output services, implementing queries on business objects. The upper one consists of the user interface and its information system events triggering the input and output services (see upper part of Figure 2).

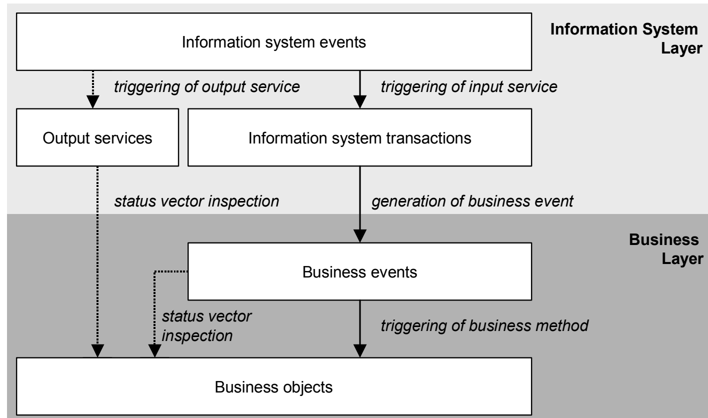

Fig. 2. Refinement of basic layers to sustain event-driven development

Business events reflect those events that occur in the real world, even if there is no information system around. Examples of business events are the creation of a new course and the enrolment of a student. In contrast, information system events are inextricably linked with an information system and allow the external user to register the occurrence of a real world event in the information system by generating input for the information system, or to request data from the information system by generating output from the information system. An example of the former is a click on a Create button in a CreateNewCourse form. An example of the latter is a click on a ViewAllCourses button, resulting in the display of a list of courses. In fact, information system events do not relate only to user interface events but also to timer and sensor signals, etc.

The business objects of the Business Layer interact by means of common business events. For each business event a business object participates in, it implements a method. The triggering of this method will induce modifications in the attributes of the business object. On invocation of the business event, all related methods are executed in parallel. In addition to a method specification, a participating object can also specify a number of constraints that should be satisfied for the event to occur. For example, the class COURSE can specify that in order to accept an *enroll* event, the course should be in the state 'open_for_registration'. Such method preconditions and also class invariants can prevent the execution of the business event. Therefore, a business event will have to check the conditions imposed by all involved business objects before it is broadcast: all class invariants and sequence constraints should be met. This synchronization is performed by means of attribute inspections on all related business objects. If all constraints are met, the business methods can be triggered and as a consequence the modifications of the attributes will be made persistent. The invoking class is notified accordingly of the rejection, acceptation, and (un)successful execution of the event.

In the Information System Layer, information system events will trigger input and output services. Output services relate to information that is extracted from the business objects and are implemented by means of a set of corresponding attribute inspections (e.g. by means of a query). As attribute inspections do not involve

changes in a business object, they are not the subjects of transaction management. In contrast, input services relate to modifications of business objects and are therefore only allowed by means of the intermediary of a business event. The handling of business events is subject to transaction management.

5. Layered architecture in some more detail

To manage consistency of the Business Layer, business objects and business events are both modeled according to the principle of Design By Contract [7]. For each method they include a precondition clause, defining when method invocation is allowed, and a postcondition clause, defining what results are guaranteed after method invocation. As a result, a business event will first synchronize all involved business objects, i.e. check whether all method preconditions are fulfilled, and only if all agree trigger the corresponding business methods. In a similar way one layer up, an information system transaction will first check whether all involved business events agree before actually generating the business events.

However, the business events as considered so far actually all pertain to atomic business events, it is to say, they relate to atomic units of action. As atomic business events first synchronize all involved business objects, all class invariants and method sequence constraints are met before broadcasting the business event. If implemented well, the business objects are also in a consistent state after execution of the business event. Nevertheless, there remain some business rules that cannot be enforced at the level of atomic events but only by combining multiple atomic business events. As such, the layered architecture depicted in Figure 2 can be refined somewhat further in distinguishing between two different kinds of business events, as explained in the example below.

Assume a course administration with a business rule stating that a student is mandatory enrolled for at least one course. In abstract terms, the atomic business event *create_student* is never allowed to happen alone, but should always be accompanied with at least one atomic business event *enroll*. This business rule can be formulated as a class invariant in the business class STUDENT, but cannot be dealt with by the atomic business event *create_student* or *enroll* alone. Actually, to enforce this mandatory relationship the atomic business events *create_student* and *enroll* are to be grouped in one consistent business event *C_create_student*. To ensure consistency, the consistent business event will be responsible for first checking whether its elementary atomic events agree and only in that case agree itself with a transaction.

Similar to the consistent event *C_create_student*, a consistent event *C_end_student* is defined, grouping the atomic business events *end_enrolment*, relating to the student's last enrolment, and *end_student*, relating to the student him/herself. Next to the consistent event *C_end_student*, a consistent event *C_end_enrolment* has to be defined, imposing an extra constraint on atomic event *end_enrolment*, in stating that the event can only be invoked alone if STUDENT has more than one enrolment. Before and after the execution of consistent events *C_end_student,* and *C_end_enrolment*, all business objects are in a consistent state and thus all business rules are satisfied.

In this way, consistent events can be considered as a layer on top of part of the atomic events with a twofold responsibility: grouping of atomic events and/or imposing extra pre- and postconditions on atomic events. Now, information system transactions are considered as a layer invoking consistent events and those atomic events that are inherently consistent, for example *modify_student*. Transactions define a logical grouping of atomic and/or consistent events. For example, the transaction 'End Student' will involve the ending of the life cycle of a student together with all his/her enrolments because of referential integrity constraints. More precisely, the transaction will invoke a set of consistent events *C_end_enrolment* and one consistent event *C_end_student* the latter being composed of one atomic event *end_enrolment* and one atomic event *end_student* (see Figure 3).

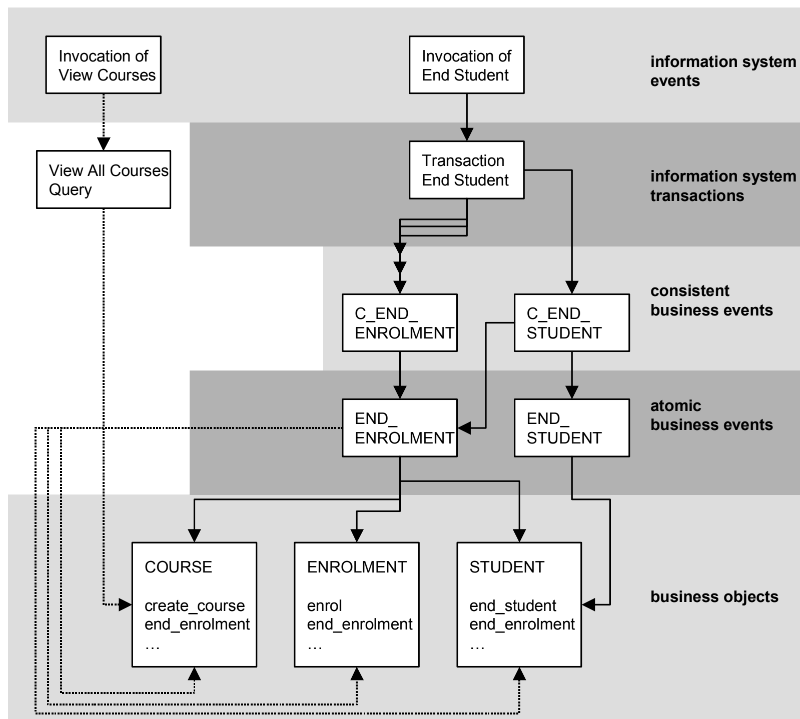

Fig. 3. Layered architecture applied to an example

6. Conclusions and future research

The architecture presented above adheres to the principles of model-driven and event-driven software development. The adoption of model-driven development resulted in the introduction of a Business Layer and an Information System Layer imposing a high-level modularity on the architecture. By adopting an event-driven

approach the basic layers are further refined and deal respectively with business events and information system events. The modeling of business object interaction by means of business events allows for a loose coupling between the business objects. Atomic business events synchronize on business objects before they are triggered, i.e. all class invariants and method preconditions should be fulfilled. In order to guarantee full consistency of the Business Layer, business events are subdivided in atomic and consistent business events, because the former cannot deal with all kinds of business rules. Therefore, consistent events will be responsible for grouping and/or imposing extra business rules on part of the atomic events.

Another advantage of the event-driven approach is that it follows the Query-Command separation Principle [7]: each service in the Information System layer is either a "command" which invokes business events but does not return values, or it is a "query" that inspects business object attributes, but does not modify or delete any object. Only commands are subject to transaction management whereas for queries simpler implementation patterns can be adopted.

Further research will concentrate on the elaboration of event-based transaction management in a distributed and loosely-coupled environment [8]. In such an environment the assumption of one global statically defined business model cannot be maintained: a set of distributed business models will exist of whom the business objects can interact in an ad hoc basis. Furthermore, a business event can no longer be considered as an atomic unit of action, as this approach would be too limiting in a loosely-coupled environment with long-standing transactions. Instead, an event should account for compensating actions if part of its execution fails.

References

1. Sowa J.F., Zachman J.A., Extending and Formalizing the Framework for Information Systems Architecture, IBM Systems Journal, 31(3), 1992, 590-616.
2. Maes R., Dedene G., Reframing the Zachman Information System Architecture Framework, Tinbergen Institute, discussion paper TI 96-32/2, 1996
3. Snoeck M., Poelmans S., and Dedene G., 2001, A Layered Software Specification Architecture, Lecture Notes in Computer Science 1920, in Laendler A.H.F., Liddle S.W., and Storey V.C., ed.: Conceptual Modeling - ER2000, 19th International Conference on Conceptual Modeling, Salt Lake City, Oct. 2000 (Springer Verlag), pp.454-469
4. Goebl W., Improving Productivity in Building Data-Oriented Information Systems – Why Object Frameworks are not Enough, Proc. of the 1998 Int. Conf. On Object-Oriented Information Systems, Paris, 9-11 September, Springer, 1998.
5. Cook S., Daniels J., Designing Object Systems: Object-Oriented Modeling with Syntropy, Prentice Hall, 1994.
6. Snoeck M., Dedene G., Existence Dependency: They Key to Semantic Integrity Between Structural and Behavioral Aspects of Object Types, IEEE Transactions on Software Engineering, Vol. 24, No. 24, April 1998, pp. 233-251.
7. Meyer B., Object-Oriented Software Construction, Prentice Hall, second edition, 1998
8. Lemahieu W., Snoeck M., Michiels C., and Goethals F., An Event Based Approach to Web Service Design and Interaction, Lecture Notes in Computer Science, accepted for APWeb2003.