

# An Enterprise Layer based Approach To Application Service Integration

Wilfried Lemahieu, Monique Snoeck, Cindy Michiels

Department of Applied Economic Sciences  
Naamsestraat 69, 3000 Leuven, Belgium  
email: {wilfried.lemahieu, monique.snoeck, cindy.michiels}@econ.kuleuven.ac.be

*This paper is forthcoming in Business Process Management Journal, Special issue on ASP*

**Abstract.** *Today many companies rely on third party applications and application services for (part of) their information systems. When applications from different parties are used together, an integration problem raises. In this paper we describe an integration approach based on the construction of an enterprise layer. This approach is deliberately kept away from a document based, flow-oriented approach where business processes are hard coded into the application architecture. Interaction is based on the concurrent update of a shared underlying enterprise layer. At the same time, the application architecture becomes easily adaptable to re-engineered business processes.*

**Keywords:** *Business Process Modelling, Domain Modelling, Application Integration, Object Oriented Analysis*

## 1. Introduction

The dramatic advance in Information and Communication Technology over the past decade allows for innovative organisational forms that were unthinkable before. Whereas most companies used to develop all the required business information systems support in-house, the need for shorter software life-cycles, the shortage of IT skills and the increased affordability and ability to amortise the cost of business solutions drives more and more companies to buy or rent the required software as applications or application services from Application Providers. In the best case, a single software product line is available for the entire company. In many cases however, the company will have to rely on the services of different providers for one or more functional domains.

When a company obtains its software from different providers a number of problems must be solved:

- Software has a significant impact on the ways of working of an enterprise and shapes the possible business processes of the organisation. An approach of one software package per functional unit suits the more functionally oriented organisational forms. A more process-oriented organisational form with enterprise-wide integrated processes is much less well supported. If a process-based organisation wants to successfully integrate applications from different providers, it first must integrate the business processes from the different applications.

- Besides integrating applications from an organisational point of view, the information management aspects must also be integrated. Each application maintains its own data, but often data will be replicated across different functional domains. For example, data about "products" and "customers" will appear in the Sales & Marketing, Service Provisioning, Customer Services and Financial domains. An integrated process-based approach requires an integrated information management support (Selsikas, 1999). Replication of data must be carefully managed and avoided whenever possible as it is likely to cause errors in the processes and complicates successful automation of business processes.

- Third-party applications that support (part of) the core business processes of a company also maintain valuable core information. If after some time the company wishes to switch to another application service provider, the question arises how to recover this core information and how to merge it into the new application.

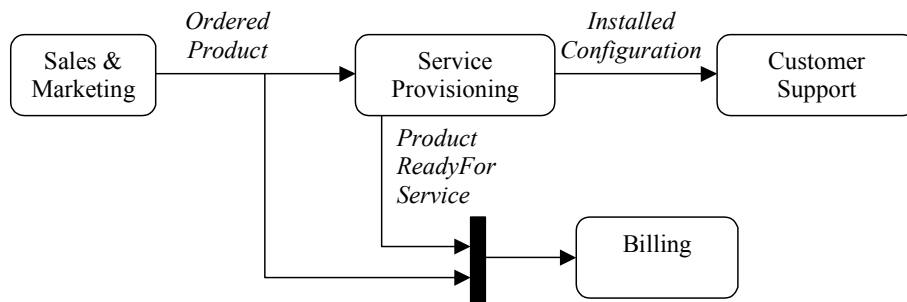
Similar problems arise when different companies want to integrate their business processes to a certain level: the integration of the business processes must be supported by integrated information processing across companies (Leclerc, 2000).

In the remainder of this paper we present an integration approach based on the concept of an enterprise layer. This kind of approach allows to stay away from a document-based, flow-oriented "stove-pipe"-like system where information is vehicled from one application to another by feeding the output of one application as input to the next in line. Rather, the common information is stored into a shared object database that can be accessed through an event-handling layer. This event-handling layer shapes the manipulation of enterprise layer objects through the definition of business events, their effect on enterprise objects and the related business rules. This "primary copy" is then used to synchronise all information in the third-party applications by means of co-ordination agents. The enterprise layer also allows companies to remain more independent from their ASPs as they remain the owners of the core company data.

The remainder of the paper is organised as follows. The second section describes the problem of application integration more in detail, illustrated by means of a real life case-study of a company in the telecommunications business area. The third section presents the overall architecture and the methodology used to build the enterprise layer. Section four zooms into the details of the implementation and section five explains how exactly information in the enterprise layer and the different applications can be synchronised. Finally, the last section draws some conclusions.

## 2. Business units with stand-alone software packages

To illustrate the problem of application integration we describe the real-life case of a company in the telecommunications business area (Lemahieu et al., 2002). The company positions itself as a broadband application provider for the SME market. In this environment one of the key factors is the ability to handle large volumes of small orders. This is only possible by means of business processes that are properly automated. The company is organised around four key business units: *Sales & Marketing*, *Service Provisioning*, *Finance* and *Customer Services*. The main business process is shown in Figure 1.



**Figure 1. Main Business Process**

The business unit *Sales & Marketing* is responsible for identifying emerging trends in the telecom industry and offering new telecom services in response. They are in charge of P.R. activities, contact potential customers and complete sales transactions. They notify the Service Provisioning and Finance/Billing department of ordered products.

The business unit *Service Provisioning* is responsible for the delivery of the sales order and organises the provisioning of all telecommunication services the customer ordered. They have to coordinate the installation of network components at the customer's site and the configuration of these components according to the type of service requested. They notify the Billing department of terminated installations and the Customer Support department of installed configurations.

The business unit *Finance* takes care of the financial counterpart of sales transactions and keeps track of the payments for the requested services.

The business unit *Customer Services* is responsible for the service after sales. They can consult the entire network infrastructure built at a specific site and can inform a customer about the progress of specific service provisioning activity or about a network problem on request.

Apart from the Sales business unit that relies only on elementary office software, each of these business units relies on different software packages from application service providers. These packages are stand-alone tools tailored to the problems at hand. Although each software package is very well suited for supporting a specific business unit, the lack of integration between the different stand-alone applications causes problems. The main reason for this is that each application looks only after its own data storage and business rules, resulting in a state of huge data duplication and severe risks for data inconsistency on a company-wide level. We will illustrate the drawbacks of this state by means of two examples.

The first example is related to the storage of people data. In the *Sales & Marketing* business unit data is stored on in-house sales people, out-house distributors and commercial contacts. The *Service Provisioning* application and the *Customer Services* application maintain data on technical contacts. The *Finance* application keeps track of financial contacts. Since the company mainly deals with SME, an individual often takes several of these roles simultaneously, so that data about one person will be distributed and replicated across several business units. The company could benefit from an approach ensuring that data on a single individual is stored and maintained in one place. For example, the company can then ensure that when a person misbehaves in one of the above roles, this person will not be accepted to take on the role of say financial contact in the future. Such a policy is only possible when all information on individuals is centralised.

The second example is related to the storage of product data. The *Sales & Marketing* business unit is responsible for conducting market research and offering appropriate telecom solutions to keep pace with evolving business opportunities. What they sell as one single product can be further decomposed into a number of parts to install and parameters to configure by the *Service Provisioning* business unit. An example of this is depicted in Table 1.

<i>Product</i>	<i>parts and parameters</i>
bi-directional link of 256 kbps	installation of an unbundled line (2x)
	installation and configuration of a router (2x)
	configuration of a virtual circuit with bandwidth of 256 kbps

**Table 1. commercial and technical view on products**

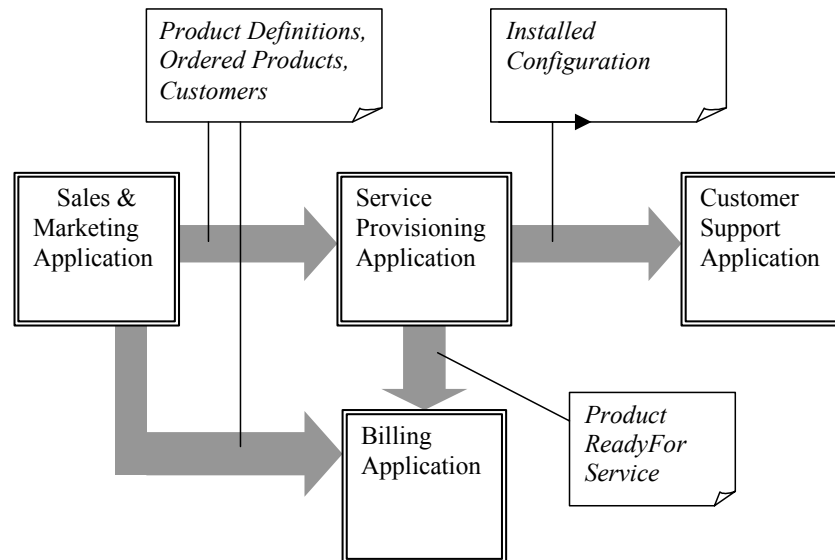
These business units need a different view on products. *Sales & Marketing* and *Finance* need a high-level product view and are not interested in low-level issues related to the installation and configuration activities. *Service Provisioning* needs to know which parts where to install and to configure and does not bother about the high-level sales and marketing issues.

In an attempt to keep a unified view on products while accommodating for their different needs, people of the business units tend to twist the product definitions in their respective software packages. By abusing attributes and fields for cross-referencing purposes, they try to maintain a more or less integrated approach. However, as the set of products in the product catalogue will increase, a unified view is no longer sustainable. Also in an international set-up with multiple business units a unified view can be held no longer: what if a single product from a sales point of view requires different technical configuration activities depending on the business unit. An example of the latter is Internet access: whereas it can be implemented by means of an unbundled line in the Netherlands and the UK, it must be provided with a leased line in Belgium where unbundling of the local loop is not (yet) possible.

The company would certainly benefit from a scalable design reconciling the commercial view and the technical view that can be taken on a single product, the former important for the *Sales & Marketing* and *Finance* business units and the latter important for the *Service Provisioning* business unit.

### 3. Enterprise Layer as an integration approach

A possible approach to the integration problem would be to build "bridges" between the different software packages. Such a bridging approach is usually based on the "flows" of information through the different business units. From the business process shown in Figure 1 we can derive such an architecture for the case at hand (see Figure 2).



**Figure 2: "Stove-pipe" architecture derived from the information flow defined in the business process**

Such an architecture does not resolve data mapping and data duplication problems: information about customers, products, and other common entities is still found in different places. Although it is unlikely that data replication can be completely avoided, another major problem with this kind of architecture is that the business process is hard-coded into the information management architecture. Re-engineering of the business processes inevitably leads to a reorganisation of the information management architecture. Such a reorganisation of IT systems is a time consuming tasks and impedes the swift adaptation of a company to the ever changing environment it operates in.

An approach to overcome the data synchronisation problems and that in addition is not hard coding the underlying business processes is to define a common layer serving as a foundation layer on top of which the stand-alone business applications can function independently and in parallel. This common layer has to encompass all relevant business objects, relations between business objects and business rules for modelling their behaviour. The so developed common layer will be called the *Enterprise Layer*; it will co-ordinate the data storage by unifying definitions.

For the development of the *Enterprise Layer* the object oriented analysis method MERODE (Snoeck & Dedene, 1998; Snoeck et al. 1999) was adopted. A particular feature of this method is that it is specifically tailored towards the development of enterprise models. MERODE advocates a clear separation of concerns, in particular a separation between the information system services and the enterprise model. The information systems services are defined as a layer on top of the enterprise model, what fits well with the set-up of the project.

A second interesting feature of MERODE is that, although it follows an object-oriented approach, it does not rely on message passing to model interaction between domain object classes as in classical approaches to object-oriented analysis (Jacobson et al., 1997; Booch et al., 1999; Fowler & Kendall, 1998; D'Souza & Wills, 1999). Instead, business events are identified as independent concepts. An object-event table (OET) allows defining which types of events affect which types of objects. When an object type is involved in an event, a method is required to implement the effect of the event on instances of this class. Whenever an event actually occurs, it is broadcast to all involved domain object classes. This broadcasting paradigm requires the implementation of an event-handling layer between the information system services and the Enterprise Layer.

To better illustrate the responsibilities of the different layers, we exemplify objects in the domain layer and the event handling layer by considering an example of an order handling system. Let us assume that the domain model contains the four object types CUSTOMER, ORDER, ORDER LINE and PRODUCT. The corresponding UML (Fowler & Kendall, 1998; Booch et al., 1999; Marshall, 1999) Class diagram is given in Figure 3. It states that a customer can place 0 to many orders, each order being placed by exactly one customer. Orders consist of 0 to many order lines, each order line referring to exactly one product. Products can appear on 0 to many order lines.



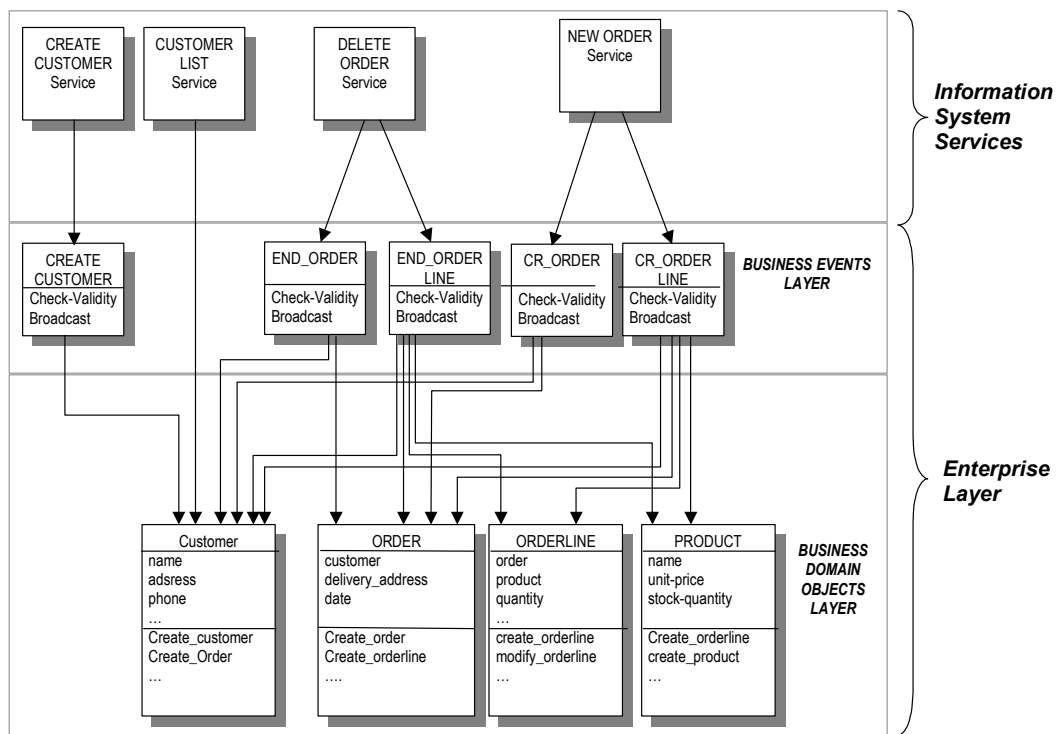
**Figure 3. Class-diagram for the order handling system**

Business event types are *create\_customer*, *modify\_customer*, *end\_customer*, *create\_order*, *modify\_order*, *end\_order*, *create\_orderline*, *modify\_orderline*, *end\_orderline*, *cr\_product*, *modify\_product*, *end\_product*. The object-event table (see Table 2) shows which object types are affected by which types of events and also indicates the type of involvement: C for creation, M for modification and E for terminating an object's life. For example, *cr\_orderline* creates a new occurrence of the class ORDERLINE, modifies an occurrence of the class PRODUCT because it requires adjustment of the stock-level of the ordered product, modifies the state of the order to which it belongs and modifies the state of the customer of the order. Notice that Table 2 shows a maximal number of object-event involvements. If we do not want to record a state change in the customer object when an order line is added to one of his/her orders, it suffices to simply remove the corresponding object-event participation in the object-event table. Full details of how to construct such an object-event table and validate it against the data model and the behavioural model is beyond the scope of this paper but can be found in (Snoeck et al., 1999; Snoeck & Dedene, 1998).

	CUSTOMER	ORDER	ORDER LINE	PRODUCT
<i>create_customer</i>	C			
<i>modify_customer</i>	M			
<i>end_customer</i>	E			
<i>create_order</i>	M	C		
<i>modify_order</i>	M	M		
<i>end_order</i>	M	E		
<i>create_orderline</i>	M	M	C	M
<i>modify_orderline</i>	M	M	M	M
<i>end_orderline</i>	M	M	E	M
<i>create_product</i>				C
<i>modify_product</i>				M
<i>end_product</i>				E

**Table 2. Object-event table for the order handling system**

Information system services are considered as a layer on top of the event-handling layer. Output services query (read) the domain objects and send this information in the appropriate format to a user interface, a printer or another external element of the system. Input services are services that add, modify or delete information in the domain model by generating the appropriate events. Figure 4 illustrates how objects in the different layers interact. The event-handling layer can, for example, be implemented by one class per type of event. This class is responsible for checking the validity of the invoked event and for broadcasting the event to all involved objects. Input services can update information in the enterprise layer by invoking the relevant event. For example, the create customer service will invoke the *create\_customer* event. Output services, such as for example the customer list service can directly inspect the state of enterprise objects.

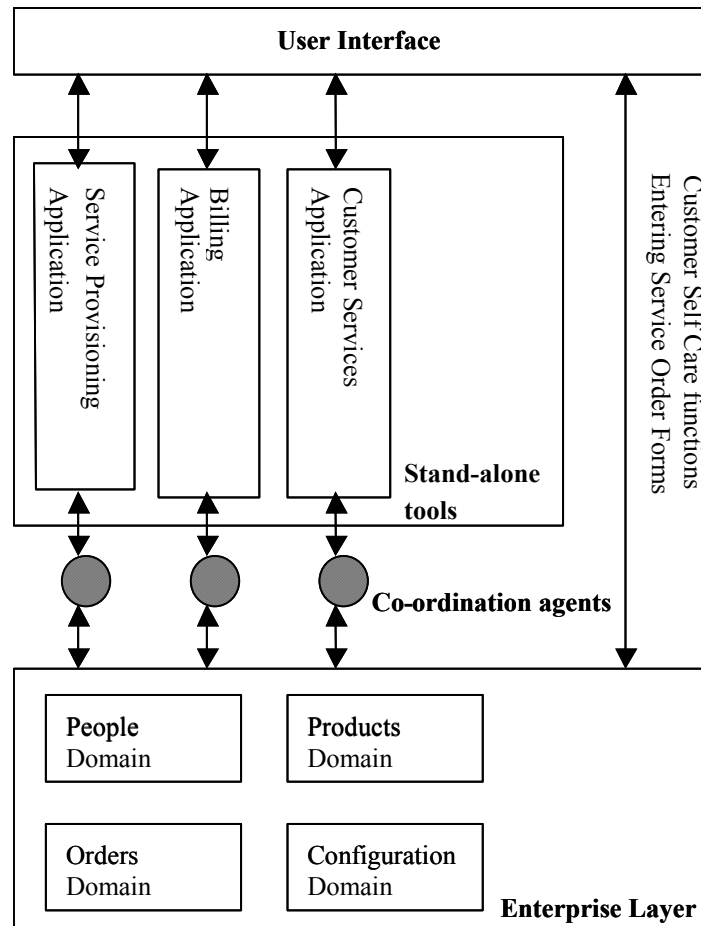


**Figure 4. Sublayers within the Enterprise layer**

The event-broadcasting approach allows implementing the Enterprise Layer (together with the event-handling layer) by means of an object-oriented implementation technology as well as a relational or object-relational technology. When not all applications are object-oriented, there is still enough freedom to choose the most appropriate implementation technology for the Enterprise Layer, without danger for a paradigm mismatch between specification and implementation.

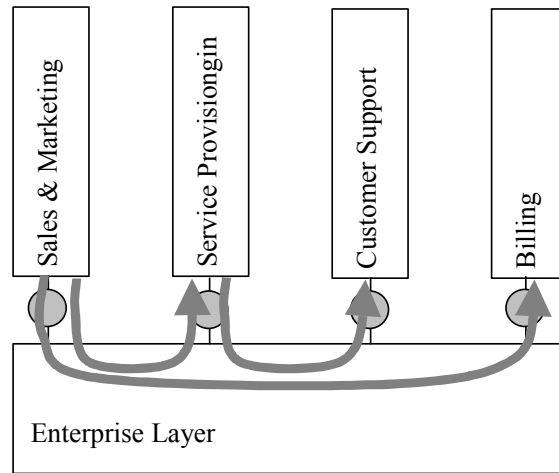
Once the Enterprise Layer is developed, the integration of the business unit applications will be realised by plugging them in on the Enterprise Layer as part of the Information Systems Services Layer. An important feature of the Enterprise Layer is that it is a *passive* layer that is not aware of its possible users. Therefore, an *active* interface between the Enterprise Layer on the one hand and the business unit applications on the other hand has to be defined. This interface can be realised by developing agents responsible for co-ordinating both sides. They will listen to the applications and generate the appropriate events in the Enterprise Layer so that state changes in business classes are always reflected in the Enterprise Model.

For the case at hand, the Enterprise Layer contains all the common objects for the acquired software packages and the domain objects required for the Sales & Marketing business area. Apart from co-ordination agents, user interfaces will be developed that interact directly with the Enterprise Layer. An example of the latter is the entering of sales orders and customer self care functions such as access and maintenance of personalised content. The resulting infrastructure is depicted in Figure 5.



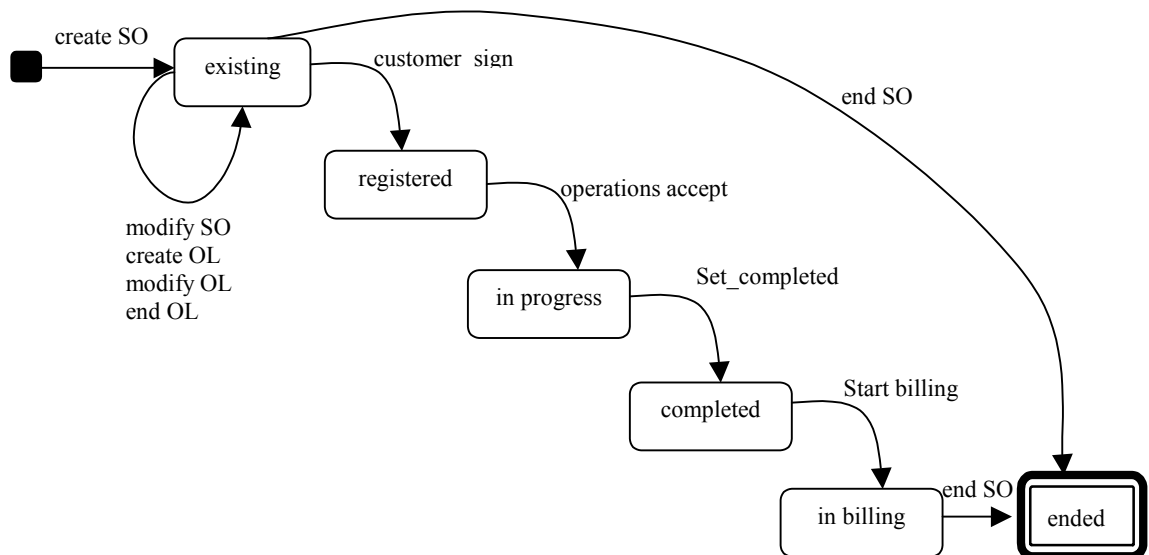
**Figure 5. Enterprise Layer integration approach**

Notice how in this type of application the Business Process is not hard-coded in the architecture. All information flows through the enterprise layer (see Figure 6). In this way, the integration approach is deliberately kept away from a document-based, flow-oriented “stovepipe”-like system. Interaction between respective application services is not based on feeding the output document of one application as input to the next in line, but on the concurrent updating of data in the shared, underlying enterprise layer. The latter defines a unified view on key business entities (e.g. CUSTOMER and PRODUCT), it encompasses relationships between business entities and formulates their behaviour in terms of *business events*, apart from other business rules. This results in a maximum of flexibility in the interaction between users and system. However, wherever certain workflow-related aspects in the business model necessitate a strict flow of information, the correct consecution of business events can be monitored by the sequence constraints enforced in the enterprise layer.



**Figure 6. Information flows in the Enterprise Layer integration approach**

Such sequences between activities are enforced in the Enterprise Layer by allowing domain objects to put event sequence constraints on their corresponding business events. For example, when a customer orders a product, a new order line is created. In terms of enterprise modelling, this requires the following business events: *create SO* (Sales Order), *modify SO*, *create OL* (Order Line), *modify OL*, *end OL*. The *customer-sign* event models the fact that a final agreement with the customer has been reached (signature of sales order form by the customer). At the same time this event signals that installation activities can be started. These kind of sequence constraints can be modelled as part of the life-cycle of business objects. In the given example this would be in the life cycle of the sales order domain object. As long as it is not signed, a sales order stays in the state "existing". The *customer\_sign* event moves the sales order into the state "registered". From then on the sales order has the status of a contract with the customer and it cannot be modified any more. This means that the events *create OL*, *mod OL* and *end OL* are no longer possible for this sales order. The *operations-accept* event signals that the sales order has been planned for installation. When installation is successfully terminated, this is recorded by means of the *set-completed* event. Finally, the *start\_billing* event signals the start of the billing process. The resulting finite state machine is shown in Figure 7. In this way, the sequence constraints mimic the general business process defined in Figure 1.



**Figure 7. State Machine for Sales Order**

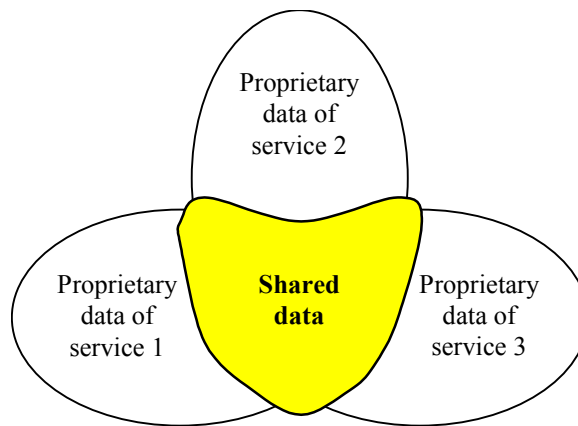
By listening to the *customer\_sign* event, the co-ordination agent for the Service Provisioning application knows when a sales order is ready for processing for Service Provisioning. In a similar

way, the Enterprise Layer allows to monitor signals from the Service Provisioning area (such as the completion of installations) and to use these signals to steer the completion of sales orders. At its turn, the completion of a sales order will be used by the billing agent to start the billing process. In this way, the Enterprise Layer allows for an automated co-ordination of behavioural aspects of different business areas. A redesign of the business process now requires the re-design of the event sequencing in the Enterprise layer, but leaves the co-ordination mechanisms unchanged.

#### 4. Implementation aspects

A MERODE-based enterprise layer can be implemented in several possible ways. The case described above uses a relational database for “passive” data storage and enterprise JavaBeans for the “active” aspects of the enterprise layer. The MERODE enterprise objects are represented as distributed objects, which are mapped transparently into relational tables. Hence, although a relational database is used for object persistence, the external view is fully object-oriented, such that interaction can be based on (remote) method invocation or event handling. External applications and services only interact with the enterprise Beans, the relational database is never accessed directly. However, this is not an absolute requirement: other implementations were built around a purely relational design, augmented with stored procedures for the active components.

Although the data in the enterprise layer can be queried directly by means of purpose-built user-interface components, its primary focus is to offer a unified view on the data objects observed by the respective application services, to which the Enterprise Layer serves as a foundation layer. The information exchange between these services and the inherently passive enterprise layer is mediated by the co-ordination agents. Each application service deals with two potential types of data (see Figure 8): its *proprietary data* that are only relevant to that particular service and the *shared data* that are relevant to multiple applications and that are also present as attribute values to the objects in the enterprise layer. Whereas the proprietary data are handled by means of the application service’s own mechanisms (such data are not relevant outside the application service anyway), it is the task of a co-ordination agent to provide the application with the relevant shared data and to ensure consistency between the application service’s view on these data and the enterprise layer’s. Such agent can be rewritten when an application is replaced, hence leaving both the application services themselves and the enterprise layer virtually unaffected.



**Figure 8. Proprietary data and shared data of application services**

A co-ordination agent supports the interaction between an application (service) and the enterprise layer in two ways: by *inspecting attribute values* of enterprise objects and by *generating business events* that affect the state of one or more enterprise objects. These two mechanisms correspond roughly to “*reading from*” and “*writing to*” the enterprise layer (see Figure 9).

“Reading” from the enterprise layer, i.e. information is passed from the enterprise layer to an application service, is rather straightforward: the co-ordination agent inspects the relevant attributes of one or more enterprise objects and passes these values to the service. The situation where information

is passed from the application to the enterprise layer (the application “writes” to the enterprise layer) is a bit more complex: because the updates that result from a given business event are to be coordinated throughout the entirety of the enterprise layer (they can be considered as a single transaction), co-ordination agents should never just *update* individual attributes of enterprise objects. Changes to the enterprise layer are only to be induced by generating business events, as stated in the MERODE specification. A business event corresponds to a row in the OET and affects all enterprise objects whose column is marked for this row.

A co-ordination agent is to acknowledge relevant events that occur in its associated application service. The co-ordination agent “writes” to the enterprise layer by triggering the corresponding business event in the enterprise layer. The enterprise objects can subscribe to business events that are relevant to them (as denoted in the OET). The enterprise objects have a method for each event type in which they participate. If a relevant event occurs, they execute the corresponding method. This method checks constraints pertinent to the (object instance, event type) combination and executes the necessary updates to attributes of that particular object if all constraints are satisfied. If not all constraints are satisfied, an exception is generated. For example, when a *create\_orderline* event is triggered four domain objects are involved that each might impose some preconditions on the event. For example:

- the order line checks that the line number is unique
- the product it refers to, checks its availability
- the order it is part of checks whether it is still modifiable
- the customer object validates the event against a limit for total cost of outstanding orders.

The global result of the business event corresponds to the combined method executions in the individual objects. The transaction is only committed if none of the objects that take part in the event have generated an exception. Otherwise, a rollback is induced.

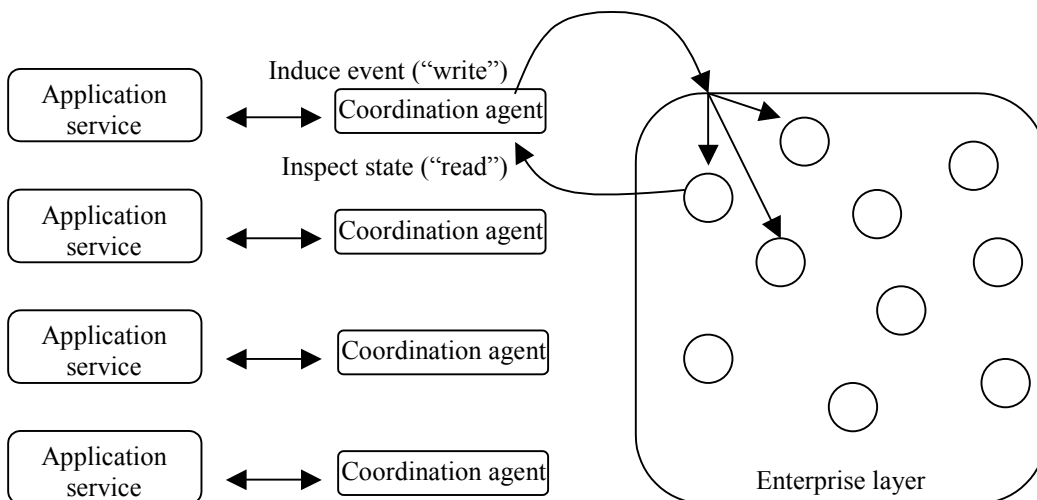


Figure 9. Interaction between application services and enterprise layer

## 5. Preservation of consistency between the replicated data by means of the co-ordination agents

The implementation modalities of “reading from” and “writing to” the enterprise layer in an actual co-ordination agent may vary, depending on an application or application service’s architecture and the middleware. The interaction mechanism is sufficiently flexible to cater for configurations with a tight coupling between (local) applications and the enterprise layer, e.g. by means of CORBA (Siegel, 1996), RMI (Pitt & McNiff, 2001) or COM (Sessions, 1997) as well as for configurations where (remote) application services and the enterprise layer are very loosely coupled, e.g. by means of SOAP (Seely & Sharkey, 2001). A mixture of both is equally feasible.

Also, not every application or application service's interaction mechanism will allow for the shared data to be accessed directly from the enterprise layer in real time. If an application (service) allows for external data (i.e. beyond the proprietary data that possibly exists in a local database) to be accessed through a gateway, all shared data can be accessed directly from the enterprise layer, without the need for replication in the application's proprietary database. The co-ordination agent interacts directly with the application's in-memory data structures for storage and retrieval to/from the enterprise layer. In terms of data integrity, this is the preferable approach, as the shared data are not replicated, hence cannot give rise to any inconsistencies between application service and enterprise layer. The interaction is synchronous, with all updates becoming visible in both the enterprise layer and the application service without any delay.

In the case where an application can only read from a local, proprietary database, the co-ordination agent is responsible for "pumping" the relevant shared data from the enterprise layer into the local database and vice versa. The enterprise layer contains the "primary" copy of the shared data, but a subset of the enterprise layer's business objects is replicated in the proprietary database of the application (service). A crucial task of the co-ordination agent will be to guarantee a satisfactory degree of consistency between enterprise layer and replicated data, especially given the situation of concurring business events. For that purpose, co-ordination agents can subscribe to insert-update-delete events (caused by other applications triggering business events) to the enterprise layer objects that are relevant to the agent's corresponding application service (using e.g. the observer pattern (Gamma et al., 1999)). When an insert-update-delete event occurs, the updated data can be transferred to the proprietary database by the agent. Such "general" events are sufficient for this purpose, as the agent only needs to know *that* an update has happened, not *because of which business event*. The enterprise layer itself can never take the initiative to call upon an application service: it is not aware of its existence.

Such updates can be propagated synchronously or asynchronously. Asynchronous interaction may be inevitable in two cases. With regard to "reading" from the enterprise layer, an application service may not always be able to immediately process an update, e.g. because it is too slow, because the connection is unreliable etc. In that case, the update propagations will be packaged by the co-ordination agent as *messages* and are *queued* until the application is able to process the updates. Another cause for asynchronous interaction, when "writing" to the enterprise layer, could be the fact that the application service's interface doesn't allow the co-ordination agent to adequately acknowledge the application's internal events. Consequently, the co-ordination agent is not immediately aware of updates to shared data replicated in the local database. These updates can only be detected by means of periodical polling, hence will only be propagated to the enterprise layer periodically.

If the interaction between application service and enterprise layer is asynchronous, a continuous consistency between updates in the application and updates in the enterprise layer cannot be guaranteed. Shared data are replicated in the service's local database and updates are propagated periodically. However, an asynchronous approach has the advantage of allowing for a very loose coupling between application and enterprise layer, whenever a synchronous approach with a tighter coupling is not feasible.

## 6. Conclusions

In this paper we have presented an integration approach based on the construction of an Enterprise Layer. This Enterprise Layer is more than a shared object database. It also contains an event-handling layer that accepts business events generated by the application layer above and broadcasts the events to the involved enterprise object types.

The business events are the basic mechanism by means of which co-ordination agents manage the consistency of shared data. The event mechanism supports a variety of co-ordination policies: direct access to the Enterprise Layer, and synchronous and asynchronous update mechanisms for replicated data.

An important advantage of the concept of event broadcasting is that it allows implementing the Enterprise Layer together with the event-handling layer by means of object-oriented as well as relational or object-relational technology. This ensures enough freedom for implementation technology choices, which is interesting when not all applications are based on the same technology.

Because all applications interact through the Enterprise Layer and never interact directly with each other, the removal of an existing application or the addition of a new application does not affect the other applications.

Finally, the Enterprise Layer architecture does not hard-code a particular business process, such that the application architecture offers the much required flexibility to adapt business processes to the ever changing environment of today's enterprises.

## 7. References

1. Booch, G., Rumbaugh, J., Jacobson, I. (1999), The unified modeling language user guide, Addison-Wesley, Reading, MA.
2. D'Souza, D. F., Wills, A. C. (1999), Objects, Components and Frameworks with UML, The Catalysis Approach, Addison-Wesley, Reading, MA.
3. Fowler, M., Kendall, S. (1998), UML Distilled: applying the standard object modeling language, Addison-Wesley, Reading, MA.
4. Gamma E., Helm R., Johnson R. (1999), Design patterns: elements of reusable object-oriented software, Addison-Wesley, Reading, MA.
5. Jacobson, I., Christerson, M., Jonsson P. et al. (1997), Object-Oriented Software Engineering, A use Case Driven Approach, Addison-Wesley, Reading MA, Rev. 4th pr.
6. Leclerc, A. (2000), Distributed Enterprise Architecture, Integrating the Enterprise, Vol. III, no. 5.
7. Lemahieu, W., Snoeck M., Michiels, C. (2002), Integration of third-party applications and web-clients by means of an Enterprise Layer, Annals of Cases on Information Technology, Idea Group Publishing, Hersley, PA.
8. Marshall C. (1999), Enterprise Modeling with UML: Designing Successful Software through Business Analysis, Addison-Wesley Professional, Reading MA.
9. Pitt, E., McNiff, K. (2001), Java.rmi: The Remote Method Invocation Guide, Addison-Wesley, Reading, MA.
10. Seely, S., Sharkey, K. (2001), SOAP: Cross Platform Web Services Development Using XML, Prentice Hall PTR, Upper Saddle River, NJ.
11. Seltsikas, P. (1999), Information Management in process-based organisations: a case study at Xerox Ltd, Information Systems Journal, Vol 9, pp. 181-195.
12. Sessions, R. (1997), Understanding COM and DCOM: Microsoft's Vision for Distributed Objects, John Wiley & Sons, New York, NY.
13. Siegel, J. (1996), CORBA – fundamentals and programming, John Wiley & Sons, New York, NY.
14. Snoeck M., Dedene G. (1998), Existence Dependency: The key to semantic integrity between structural and behavioral aspects of object types, IEEE Transactions on Software Engineering, Vol 24 No. 24, pp.233-251.
15. Snoeck M., Dedene G., Verhelst M, Depuydt A. M. (1999), Object-oriented Enterprise Modeling with MERODE, Leuven University Press, Leuven.

### Acknowledgement

The case described in this paper has resulted from a project executed with the company NOVAXESS based in Amsterdam, the Netherlands.