

Integration of third-party applications and web-clients by means of an Enterprise Layer

**Wilfried Lemahieu, Monique Snoeck, Cindy Michiels,
K.U.Leuven**

Published in Annals of Cases on Information Technology, Vol5/2003, pp. 213-233

EXECUTIVE SUMMARY

This Case Study presents an experience report on an Enterprise Modelling and Application Integration project for a young company, starting in the telecommunications business area. The company positions itself as a broadband application provider for the SME market. Whereas its original information infrastructure consisted of a number of stand-alone business and operational support system (BSS/OSS) applications, the project's aim was to define and implement an Enterprise Layer, serving as an integration layer on top of which these existing BSS/OSS would function independently and in parallel. This integration approach was to be non-intrusive, and was to use the business applications as-is. The scope of the case entails the conception of a unifying Enterprise Model and the formulation of an implementation architecture for the Enterprise Layer, based on the Enterprise JavaBeans framework.

ORGANIZATION BACKGROUND

This Case Study deals with a company acting as supplier of fixed telephony and of broadband data communication and Internet services. A particular feature of the company is that all the telecom services it offers are facilitated via 'Unbundling of the Local Loop' (ULL).

ULL is the process where the incumbent operator makes its local network (the copper cables that run from customers premises to the telephone exchange) available to other operators. These operators are then able to use individual lines to offer services such as high speed Internet access directly to the customer. The European Union regulation on ULL requires incumbents to offer shared access (or line sharing). Line sharing enables operators and the incumbent to share the same line. Consumers can acquire data services from an operator while retaining the voice services of the incumbent. Some operators may choose to offer data services only, so with line sharing consumers can retain their national PTT service for voice calls while getting higher bandwidth services from another operator without needing to install a second line.

The regulation on ULL can have a significant impact on the competing forces in the telecom industry: it offers a great opportunity for new companies to enter the telecom market and compete with the incumbent operator. Indeed, by means of ULL the sunk cost of installing a countrywide network infrastructure is not an obstacle any more for new entrance in the telecom market.

A large Telecommunication Company immediately understood the business opportunities behind this new regulation and decided to exploit the ULL benefits in all European countries. In a first step, it has created a starter company that is the subject of this Case Study. As a means to differentiate from the services offered by the incumbent operator, the new company focuses on telecom services for the business market, the small and medium-sized sector in particular. The main headquarters are located in the first European country where ULL is possible. The starter company was set up in September 1999. In March 2000, it succeeded in acquiring two major investors from the US, which are both specialists in new media. The company has evolved rapidly and in August 2001 it already surpassed 2000 customers and employed about 150 people. Until now, the company offers its services in two countries. Gradually, the company will extend its coverage of the European Union by opening new premises in the countries that enable ULL.

SETTING THE STAGE

Business units with stand-alone software packages

The company is organised around four key business units: Sales & Marketing, Service Provisioning, Finance and Customer Services. 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. 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. 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 have access to the entire network infrastructure and on request they can inform a customer about the progress of a service provisioning activity, about the network status or possible network breakdowns. The main business process is shown in Figure 1.

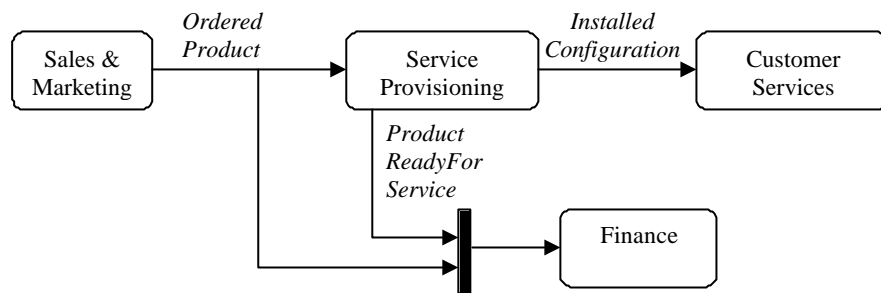


Figure 1. Main Business Process

It is a company policy that the amount of in-house developed software must be limited. The company has therefore acquired a number of off-the-shelf software packages. Apart from the Sales & Marketing business unit that relies only on elementary office software, each of these business units relies on different business and operational support systems (BSS/OSS) that are tailored to the problems at hand. Until recently, all BSS/OSS were functioning independently from each other. The company quickly understood that the integration of the BSS/OSS could improve its competitive position in three ways. On the short run, the company could benefit both from a *better performance* in transaction processing and *more transparency* in its business process. On the long run, the activities of the company should become *better scalable* to an international set-up with multiple business units. Now is presented in some more detail how the integration of the BSS/OSS can realise these benefits.

Better performance

To differentiate from the services offered by the incumbent operator, the company has targeted her services towards the SME market segment. As opposed to the segment of multinational companies, the SME segment typically involves large volumes of relatively small sales orders. But the performance of the present information infrastructure with stand-alone BSS/OSS deteriorates significantly when a large number of transactions have to be processed simultaneously. A main factor explaining this performance decrease is that the BSS/OSS applications are essentially used as stand-alone tools: each application supports only part of the value chain and is treated in isolation. Although each package is very well suited for supporting a specific business unit, the lack of integration between the different applications causes problems: 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. As a result a substantial amount of manual work is required to co-ordinate the data stored in the different packages. To avoid these problems in the future, an integration approach should be adopted guaranteeing a centralized data storage and organizing the automatic coordination between the different BSS/OSS.

More transparent

By integrating the BSS/OSS applications, company-wide business rules can be maintained that give a more formal structure to the company's business organisation. As a result, the business process will become more transparent and better controllable. Previously, the cooperation between different business units was rather ad hoc and not yet formalised on a company-wide level. As a result, a number of business opportunities/risks could often not be detected by the company. For example, the company had problems with a customer company not paying her bills. Later on the commercial contact of this company reappeared as financial contact person in another customer company and again bills didn't get payed. This repetition of the same problem could have been avoided if information on people had been centralised. In the current system, data on people is stored in all four business units. Sales & Marketing stores data on in-house sales people, out-house distributors and commercial contacts. Service Provisioning and Customer Services both 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 with the current information infrastructure data about one person will be distributed and replicated across several business units. Nevertheless, the company could benefit from an approach

ensuring that data on a single person is stored and maintained in one place. Indeed, if a person is not loyal in its role of commercial contact, the company should take extra care in the future when assigning this person the role of financial contact. Such a policy is only maintainable if all information on individuals is centralised. (In the integrated information infrastructure such a person will be altered to the state 'blacklisted' and as long as he resides in this state he cannot be assigned to one of the above roles.)

Better scalable

As an early adopter of the ULL opportunity, the company is very likely to become one of the major players targeted to the telecom SME market in the European Union. Flexibility in both the adoption of new products, to keep pace with the evolving technology, and in the adaptation of existing products, to conform to different company standards, will be one of the cornerstones to realise this objective. However, the current information infrastructure cannot guarantee this level of flexibility.

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 the table below.

| <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

In fact, the 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 ULL in the Netherlands and the UK, it must be provided with a leased line in Belgium where ULL 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.

CASE DESCRIPTION

Goals of the project

The goal of the project was to develop a solution for the integration of the BSS/OSS applications, so as to achieve fully automated processes that are able to handle scalable business processes, systems and transaction volumes. The same software will be used in all European divisions of the company. The company has contacted a team of researchers at a university to consult them for the development of a solution. A consultancy company, unknown at the start of the project, will do the implementation. The time frame of the project is roughly 5 months for requirements engineering, about 2 months for the technical design of the solution's architecture and another 4 months for implementation. During the requirements engineering period, the company will look for an adequate implementation environment and

contract a consultancy company for the implementation part. The remainder of this section outlines the proposal as the university team developed it.

Choosing the right integration approach

1. A bridging 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).

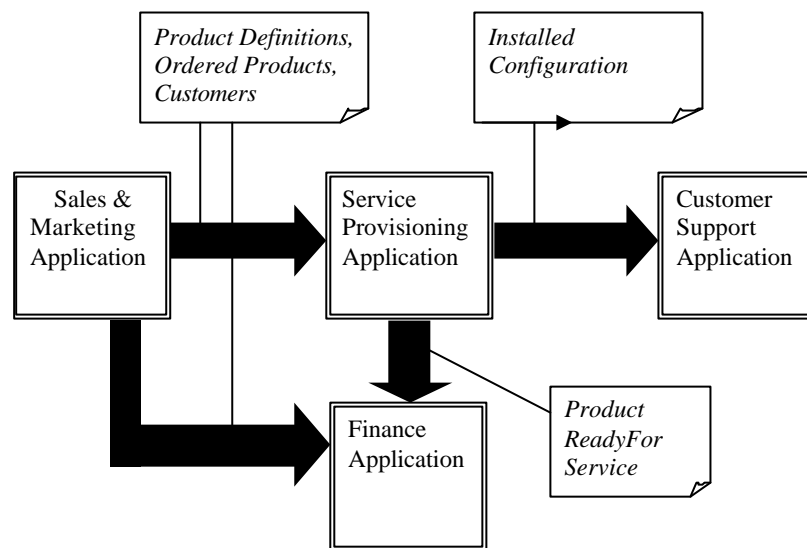


Figure 2: "Stove-pipe" architecture derived from the information flow defined in the business process

The major advantage of this approach is its simplicity. If one chooses to automate the existing manual procedures, the amount of required requirements analysis is limited. In addition, one can rely on the expertise of the providers of the BSS/OSS software to build the bridges. Such architecture does however 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 task and impedes the swift adaptation of a company to the ever changing environment it operates in.

2. An integration approach based on an Enterprise Layer

An approach that does not hard code the underlying business processes is to define a common layer serving as a foundation layer on top of which the stand-alone BSS/OSS can function independently and in parallel (see Figure 3). This so called *Enterprise Layer* has to co-ordinate the data storage by defining a unified view on key business entities such as CUSTOMER and PRODUCT. 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. From an integration point of view this is certainly a more flexible approach: it does not hard-code the current business processes what more easily allows for business process redesign. In addition, the replacement of a particular BSS/OSS service will not affect the

other packages: all interaction is accomplished through the Enterprise Layer, they never interact directly with each other. In this way the company is more independent of the vendors of the packages. On the other hand, this solution has a number of factors that will make it more costly than the bridging approach:

- it requires a thorough domain analysis in order to integrate the concepts of the four functional domains;
- relying on the expertise of the software vendors will be more difficult;
- there is no experience in the company with such an approach.

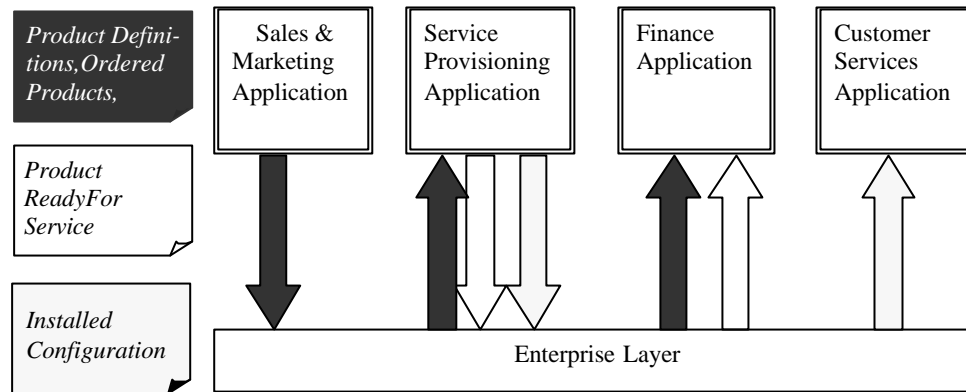


Figure 3: "Enterprise layer " architecture derived from the information flow defined in the business process

Phases of the automation strategy and global architecture

After considering the advantages and disadvantages of both the bridging and the Enterprise Layer approach, the company has opted for the latter. An additional motivation is that apart from the integration aspects, the Enterprise Layer will allow the future development of e-business functionality directly on top of the Enterprise Layer, rather than via one of the BSS/OSS applications. The automation strategy of the company is hence summarised by the following four steps:

- Step 1: Roll out of industry proven BSS/OSS applications with out of the box functionality. Each application is treated in isolation (total lack of integration).
- Step 2: Specification and development of an Enterprise Layer that will support all applications and user interfaces. The Enterprise layer is by definition passive and not aware of its users.
- Step 3: Integration of the existing BSS/OSS applications by 'plugging' them in on the Enterprise Layer. The interface between these applications and the Enterprise Layer will be realised by the definition of agents, responsible for co-ordinating the information exchange.
- Step 4: Development of user interfaces on top of the BSS/OSS applications or directly on top of the Enterprise Layer.

At the start of the project, step 1 had been realised: three out of the four main business domains (as represented in Figure 1) were supported by a standalone software package. There existed no automated support for the sales domain: sales business processes were mainly paper-based or used standard office software such as word processors and spreadsheets. The university team was responsible for requirements engineering and technical design for steps 2 and 3.

Modelling the Enterprise Layer

Development of an Enterprise Model: choice of the methodology

In order to specify an Enterprise Layer that acts as an integration layer of the existing information infrastructure, it is key that all the company's business rules are formalised and modelled as a subset of the Enterprise Model. This requires the choice of a systems analysis method, preferably one that is suited for domain modelling. Two options were considered:

1. Enterprise modelling with UML

UML has become a de facto standard for Object Oriented Analysis. It has a very broad scope of application and the major techniques (class diagram, state machines, Use Cases) can be used for enterprise modelling purposes. Many CASE tools are available that facilitate the transition to design and implementation. The language is known to practically all IT-people, which will facilitate the communication with the implementation team.

On the other hand, UML is not an approach but a *language*. It is not specifically tailored to enterprise modelling either. As a result, UML is not a great help for the modelling task. An other disadvantage of UML is its informal semantics. Because of the informal semantics, UML offers no support for the consistency checking and quality control of specifications. In addition, the informal semantics also increase the risk of misinterpretation by the implementation team.

2. Enterprise modelling with MERODE

A particular feature of the object oriented enterprise modelling method MERODE (Snoeck & al., 1999; Snoeck & al., 2000) 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 systems services and the Enterprise Model. The information systems services are defined as a layer on top of the Enterprise Model, what perfectly fits with the set-up of the project. The Enterprise Layer can be considered as a foundation layer on which the acquired off-the-shelf software can act as information services.

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 (Booch & al., 1999; D'Souza & Wills, 1999; Jacobson & al., 1997). Instead, *business events* are identified as independent concepts. An object-event table (OET) allows to define which types of objects are affected by which types of events. 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. An interesting consequence is that this allows to implement the Enterprise Layer both with object-oriented and non-object-oriented technologies. Since some of the acquired software is built with non-object oriented technology, this is a substantial advantage.

A third feature of MERODE is that the semantics of the techniques have been formalised by means of process algebra. This allows checking the semantic integrity of the data model with the behavioural models.

On the negative side, MERODE has a narrow scope of application: it is only intended for enterprise modelling purposes. Although some hints are given on how to implement an Enterprise Layer, the MERODE approach gives but very little support for the specification and implementation of services on top of the Enterprise Layer. A second major disadvantage of the method is that it is unknown to most people. Because of this and of the peculiar way of treating events and inter object communication, difficulties are to be expected for communicating the requirements to implementation team. It is likely that the implementation team will not be familiar with the approach and that additional MERODE-training will be required.

Based on the above considerations, the project manager decided to go on with MERODE as modelling approach. The major reasons for this choice are the consideration that quality of the Enterprise Model is a key factor for the success of the Enterprise Layer approach and the possibilities for implementation with a mixed OO and non-OO approach.

Key concepts of the Enterprise Model

To better illustrate the responsibilities of the different layers, objects in the domain layer and the event handling layer are exemplified 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 (Booch & al., 1999) Class diagram is given in Figure 3.

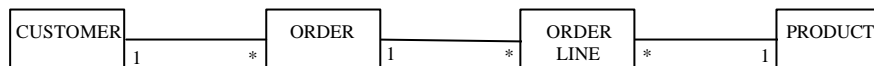


Figure 3. Class-diagram for the order handling system

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, the *create_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 one does 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 case study but can be found in (Snoeck & Dedene, 1998; Snoeck & al., 1999).

| | 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

The Enterprise Layer developed in the project covers the company's four main business domains: People, Products, Orders and Configuration.

The *People* domain concerns both the customers and the sales persons. Information about people is found in all four business processes. The Sales & Marketing process stores data on sales people (both in-house and distributors) and on commercial contacts for customers. The Service Provisioning application and the Customer Services application both maintain data on technical contacts. Finally, the Financial application

keeps track of data about financial contacts. Since the company mainly deals with SME, a single person often takes several roles simultaneously, so that information about the same person can be distributed and replicated across several business processes. The Enterprise Layer ensures that information about an individual person is stored and maintained in a single place.

The *Products* domain describes the products sold by the company. The four business processes have their own particular view on the definition of the concept "product", and each BSS/OSS maintains its own product catalogue. Again, the Enterprise Layer will be responsible for tying together the description of products. Initially, it was assumed that a single company-wide definition per product would be possible, but soon it appeared that each functional domain had to work with a particular view on this definition. Therefore, the final Enterprise Layer describes products both from a marketing/sale perspective (what products can be sold and for which price) and from a technical perspective (what are the technical elements needed to deliver an item). The sales perspective is variable over time depending on sales marketing campaigns, both from a price and descriptive standpoint. The technical description is constant over time, but new technology may cause a redefinition without changing sales description.

The *Orders* domain encompasses all business objects related to sales orders. The sellable products are defined and registered in the *Products* domain; the actual ordered products are registered in the *Orders* domain. Finance will use this domain.

The *Configuration* domain keeps track of all business objects related to the technical configuration that is build at a customer's site during provisioning activities. The parts to install and the parameters to configure are defined and registered in the *Products* domain, the actual installed parts and configured parameters are registered in the *Configuration* domain. The information of this domain is provided by the Service Provisioning application and is consulted by the Customer Services application.

Business rules and constraints

Whereas the above mainly dealt with information *structure*, the MERODE Enterprise Model also incorporates *behavioural* specifications. Indeed, as can be seen from the description of the business process, activities of the different departments must be co-ordinated in a proper way. Notice how in this type of approach the business process is not hard-coded in the architecture. All information flows through the Enterprise Layer. In this way, the integration approach is deliberately kept away from a document-based, flow-oriented "stovepipe"-like system. Interaction between respective applications 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. 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.

Indeed, 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_salesorder*, *modify_salesorder*, *create_orderline*, *modify_orderline*, *end_orderline*. 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 kinds 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 anymore. This means that the events *create_orderline*, *modify_orderline* and *end_orderline* are no longer possible for this sales order. The resulting finite state machine is shown in Figure 3.

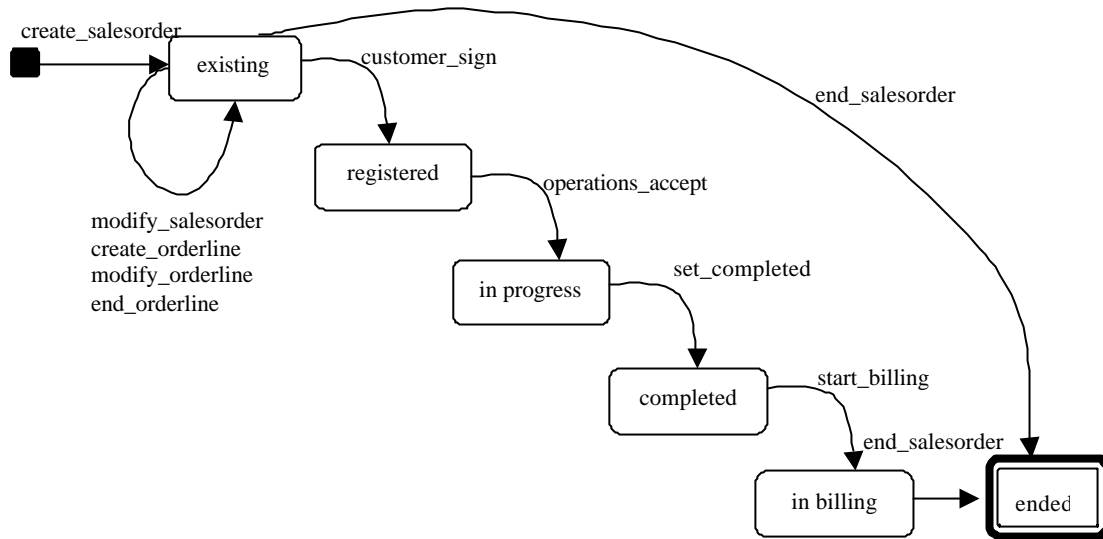


Figure 3. State Machine for Sales Order

A layered infrastructure: co-ordination agents and user interfaces

Once the Enterprise Layer is developed, the integration of the business unit applications is realised by plugging them in on the Enterprise Layer. The previously stand-alone BSS/OSS applications now constitute the middle 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 had to be defined. This interface can be realised by developing agents responsible for co-ordinating both sides. Such *co-ordination agents* 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 example, 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 (event *set_completed*) can 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 requires the re-design of the event sequencing in the Enterprise layer, but leaves the co-ordination mechanisms unchanged.

The top layer is established by the development of user interfaces that offer a Web interface to the BSS/OSS applications or directly to the Enterprise Layer. For example, functions related to the sales process (entering of Sales Order Forms) and to customer self care functions (access and maintenance of personalised content) are input and output functions performed directly on the Enterprise Layer. The resulting layered infrastructure is depicted in Figure 4.

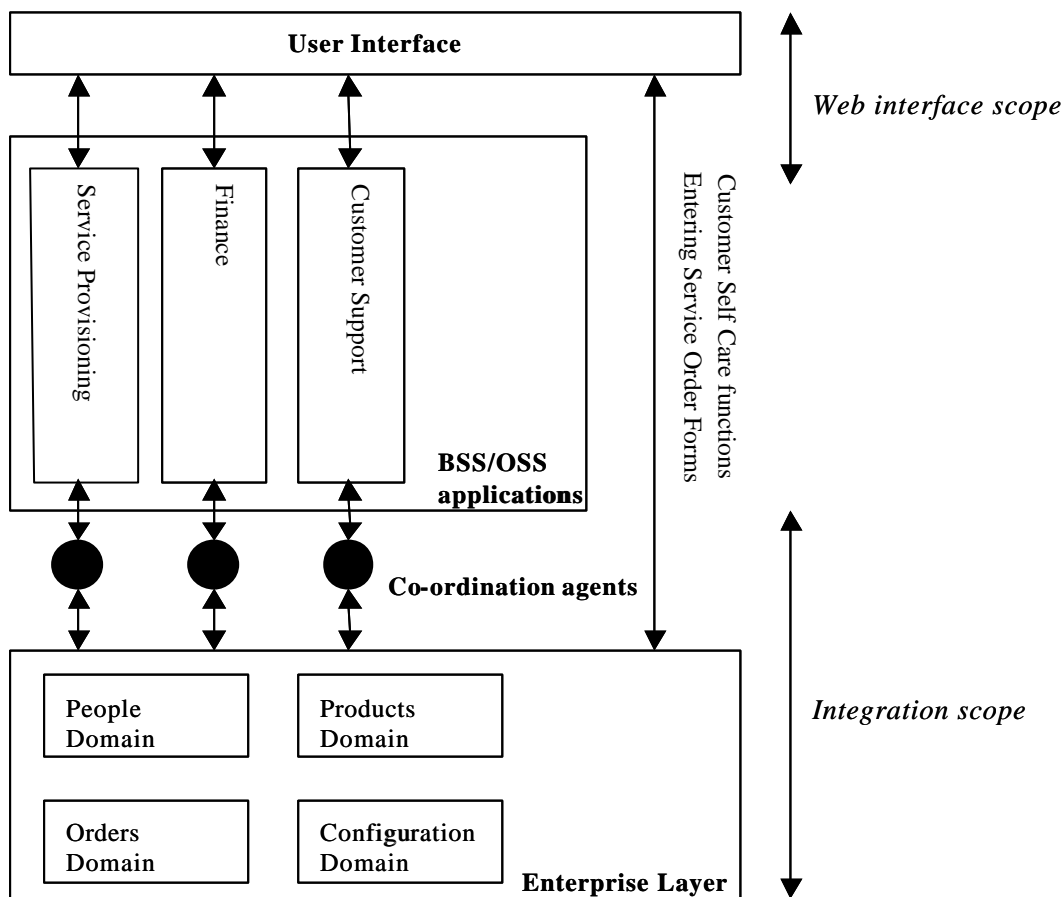


Figure 4. Enterprise Layer integration approach

Implementation of the Enterprise Layer

Choosing the implementation architecture

The Enterprise Layer consists of the MERODE enterprise objects, as defined above. These objects embody a combination of data and behaviour. As to the pure data, a relational database (Oracle) seemed to be the obvious implementation choice. However, to implement the entirety of data and behaviour, three alternatives were considered:

1. A combination of stored procedures and relational tables

A first option consisted of a combination of *stored procedures* (for the functionality) and *relational tables* (for the data). Indeed, although MERODE is an object-oriented Analysis & Design methodology, the implementation is not restricted to an object-oriented environment. The object definitions can be mapped to relational structures, whereas the object-event table can be translated to stored procedures. Each horizontal line in the object-event table will define a single stored procedure, resulting in one procedure per event. Each such procedure enforces the necessary constraints upon all objects participating in the event and manipulates the relational database tables to which these objects are mapped.

2. An object-relational approach

A second possibility was to build upon the *object-relational capabilities of Oracle*. Here, the MERODE enterprise objects would be implemented as user-defined types in Oracle, which combine data and behaviour into “object types”, stored in the Oracle database.

3. A distributed object architecture

A third option was to turn to a *component-based distributed object architecture* such as EJB, CORBA or DCOM. In this approach, the business objects would be represented as distributed objects, which are mapped transparently into relational tables. The external view, however, remains fully object-oriented, such that interaction can be based on method invocation or event handling.

The eventual preference went out to an approach based on a *component-based distributed object architecture* for several reasons. First, it allows for the Enterprise Layer to be perceived as real objects. In this way, all constraints defined in MERODE can be mapped directly to their implementation. The constraints are not only attributed to *events*, but also to *objects* and can be enforced and maintained at the object level (i.e. the vertical columns of the MERODE object-event table: each column represents an object that reacts to different events). This is not possible with a purely relational design, nor with an object-relational implementation, where object-oriented concepts are still tightly interwoven with relational table structures. As a consequence, a distributed object approach offers the best guarantee with regard to maintainability and scalability. Moreover, a number of “low-level” services such as concurrency control, load balancing, security, session management and transaction management are made virtually transparent to the developer, as they can be offered at the level of the application server.

As to the choice between CORBA, EJB and DCOM, the Java based EJB (Enterprise JavaBeans) was chosen, offering a simpler architecture and being easier to implement than CORBA. Moreover, unlike DCOM, it is open to non-Microsoft platforms. The EJB architecture guarantees *flexibility*: all enterprise Beans are components and can be maintained with little or no consequences for other Beans. Also, such environment is easily *scalable*, as enterprise Beans can be migrated transparently to another server, e.g. for load balancing purposes. Finally, in this way, the Enterprise Layer would easily exploit Java’s opportunities for Web based access such as JSP (Java Server Pages) and servlets. The proposed architecture conformed the *n-tier* paradigm, utilizing the database server only for passive data storage and moving all actual functionality to a separate application server, where the enterprise Beans represent the “active” aspects of the Enterprise Layer (see Figure 5). The MERODE enterprise objects are implemented as so-called *entity Beans*. These are a specific type of enterprise JavaBean, that essentially denote object-oriented representations of relational data, which are mapped transparently into relational tables. Updates to an entity Bean’s attributes are propagated automatically to the corresponding table(s). 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.

All functionality is available in the enterprise Beans on the application server. External applications are to make calls upon the enterprise Beans by means of Java’s RMI (Remote Method Invocation). A business event is dispatched to each entity Bean that represents a business object that participates in the event. An enterprise object handles such event by executing a *method*, in which also constraints pertaining to that particular (object type, event type) combination are enforced. Hence, rather than enforcing integrity constraints at the level of the relational database, they are enforced in the entity Beans’ *methods*. Clients only interact with the enterprise Beans; the relational database is never accessed directly. In this way, the MERODE specification can be mapped practically one-to-one to the implementation architecture.

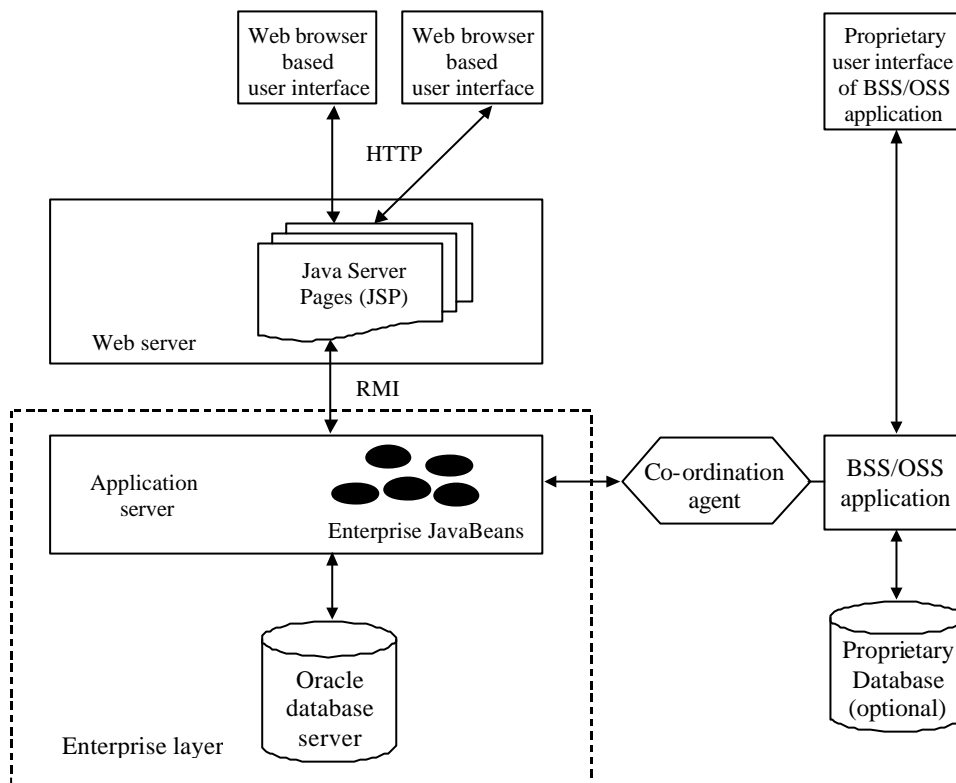


Figure 5. Proposed architecture

Interaction between applications and Enterprise Layer

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 BSS/OSS applications, to which the Enterprise Layer serves as a foundation layer.

Each BSS/OSS application deals with two potential types of data: its *proprietary data* that are only relevant to that particular application 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's own mechanisms (such data are not relevant outside the application 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's view on these data and the Enterprise Layer's. External applications can interact with the Enterprise Layer in two ways: by *inspecting attribute values* of enterprise objects and by generating *business events that affect the state of enterprise objects*. These two mechanisms correspond roughly to "reading from" and "writing to" the Enterprise Layer (see Figure 6).

"Reading" from the Enterprise Layer, i.e. information is passed from the Enterprise Layer to an application, is rather straightforward: the co-ordination agent inspects the relevant attributes of one or more enterprise objects and passes these values to the application. As already mentioned, the enterprise objects are deployed as *entity Beans*. These can be seen as persistent objects, i.e. they are an object-oriented representation of data in an underlying relational database. Entity Beans have predefined *setAttribute()* and *getAttribute()* methods, which can be published in their remote interface. In this way, these methods can be called remotely through RMI for respectively reading and updating the value of a particular attribute. Hence, attribute inspections come down to calling a *getAttribute()* method on the corresponding entity

Bean. Applications (or their co-ordination agents) can call directly upon published *getAttribute()* methods to retrieve data from the Enterprise Layer.

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 co-ordinated 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. Hence, in EJB terminology, *setAttribute()* methods should never be published in an entity Bean’s public interface. Changes to the Enterprise Layer are only to be induced by generating *business events* that affect the state of the enterprise objects, as stated in the MERODE specification.

A business event corresponds to a row in the object-event table and affects all enterprise objects whose column is marked for this row. They are generated by means of another kind of enterprise JavaBeans: *session Beans*. These represent non-persistent objects that only last as long as a particular client session. Each business event type is represented by a session Bean. The latter publishes a method for triggering an event of the corresponding type. A co-ordination agent acknowledges relevant events that occur in its associated application and that have to be propagated to the Enterprise Layer. The agent “writes” to the Enterprise Layer through a single (*remote*) *method invocation* on a session Bean. The session Bean’s method generates a “real” Java event, which represents the business event, to which the entity Beans that take part in the event (as denoted in the object-event table) can subscribe.

The entity Beans 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 (the effect may also be the creation or deletion of objects). If not all constraints are satisfied, an exception is raised.

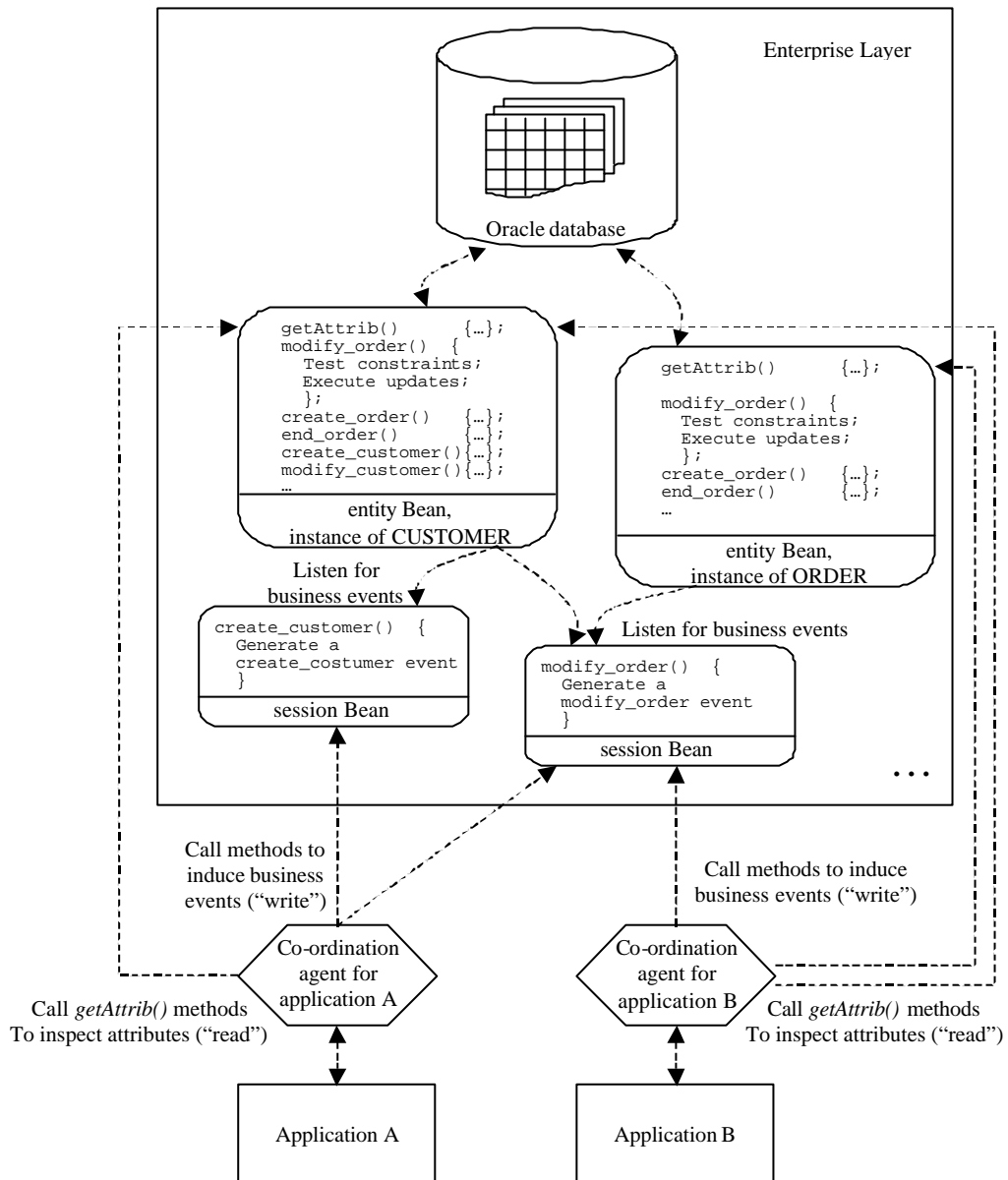


Figure 6. Interaction between applications and Enterprise Layer

For example, when a `create_orderline` event is triggered, four domain objects are involved that each might impose some preconditions on the event:

- 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 update is only committed if none of the objects that take part in the event have raised an exception. Otherwise, a rollback is induced.

Implementation of the co-ordination agents

The BSS/OSS applications that are currently used have very diverse APIs and ways of interacting with external data. The latter will affect the way in which an agent actually mediates between the application and the Enterprise Layer. Some applications only feature an in-memory representation of their data, whereas others have their own local database with proprietary data. Also, not every application's API will allow for the shared data to be accessed directly from the Enterprise Layer in real time. This section discusses three concrete categories of implementations for a co-ordination agent and, consequently, interaction mechanisms between an application and the Enterprise Layer. Virtually all kinds of applications will fit in at least one of these (as, obviously, did the applications that were used in the project). The categories can be characterised by means of the *moment in time* on which the reading from and writing to the Enterprise Layer takes place. One can discern between *synchronous interaction without replication of shared data*, *synchronous interaction with replication of shared data* and *asynchronous interaction*.

1. Synchronous interaction, without replication in a local database

If the application's API allows for data beyond its local database to be accessed *directly* through a database gateway, all shared data can be accessed straight from the Enterprise Layer, without the need for replication in the application's local database. The co-ordination agent interacts directly with the application's in-memory data structures for storage to and retrieval from the Enterprise Layer. This is the preferable approach, as the shared data are not replicated, hence cannot give rise to any inconsistencies between application and Enterprise Layer. The interaction is *synchronous*, as all updates are visible in both the Enterprise Layer and the application without any delay.

Information is passed from the Enterprise Layer to the application on the actual moment when the application wants to access the shared data: the application issues a request, which is translated by the co-ordination agent to attribute inspections on the entity Beans representing the enterprise objects. The result is returned to the application.

An information flow from the application to the Enterprise Layer exists on the moment when the application updates (part of) the shared data: the application issues a request, which is translated by the co-ordination agent to a method call on a session Bean that corresponds to the appropriate business event. The session Bean generates an event in the Enterprise Layer, to which each entity Bean that participates in the event responds by executing a corresponding method, which executes the necessary updates to that particular entity Bean. These updates are only committed if none of the entity Beans raises an exception because of a constraint violation.

2. Synchronous interaction, with replication in a local database

If the application's API does not allow for data beyond its local, proprietary database to be accessed directly through a database gateway, all shared data have to be replicated in the local database for access by the application. The Enterprise Layer then contains the "primary" copy of the shared data and the co-ordination agent is responsible for "pumping" the relevant shared data from the Enterprise Layer into the local database and vice versa. 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.

Even in this case, the interaction can still be synchronous if the local database can be kept in consistency with the Enterprise Layer at any time, i.e. all updates are visible in both the Enterprise Layer and the application without any delay. However, the moment of reading from the Enterprise Layer differs from the situation without replication: when the application accesses shared information, all data (both shared and proprietary) are retrieved from the local database. An information flow from the Enterprise Layer to the application now takes place *when shared attributes in the Enterprise Layer are updated by another application*. For that purpose, the application's co-ordination agent *listens* to events (generated by another application) in the Enterprise Layer that cause updates to data relevant to its corresponding application. The mechanism can be implemented by means of the observer pattern (Gamma & al., 1999). When a create, modify or end event on relevant data in the Enterprise Layer is detected by the agent, the

information is propagated by issuing inserts, updates or deletes in the application's local database. Note that the Enterprise Layer itself can never take the initiative to propagate updates to an application: it is not aware of its existence.

Similarly to the case without replication, information is passed from the application to the Enterprise Layer when the application updates shared data. These updates are initially issued on the replicated data in the local database. The co-ordination agent listens for updates in the local database. If such update concerns data that is also present in the Enterprise Layer, it triggers a business event (again by means of a session Bean) in the Enterprise Layer, which provokes the necessary updates.

3. Asynchronous interaction

When the interaction between application and Enterprise Layer is *asynchronous*, a continuous consistency between updates in the application and updates in the Enterprise Layer cannot be guaranteed. Again, shared data will have to be replicated in a local database but this time, updates on one side will only be propagated *periodically* instead of in real time. In terms of the risk of inconsistencies, this is of course the least desirable approach. However, it has the advantage of allowing for a very loose coupling between application and Enterprise Layer, whenever a synchronous approach with a much tighter coupling is not feasible. One can distinguish two situations that call for asynchronous interaction; a first cause, when "writing" to the Enterprise Layer, could be the fact that the application's API doesn't allow the co-ordination agent to listen for the application's internal events. Consequently, the co-ordination agent is not immediately aware of the application updating shared data replicated in the local database. These updates can only be detected by means of periodical polling. Hence, an information flow from the application to the Enterprise Layer exists when, during periodical polling, the co-ordination agent detects an update in the shared data and propagates this update to the Enterprise Layer.

With regard to "reading" from the Enterprise Layer, a co-ordination agent can always listen in the Enterprise Layer for relevant events (triggered by another application) causing updates to shared data. This is the same situation as for synchronous interaction with replicated data, where it not that the updates cannot be propagated right away to the application's local database. Indeed, one could imagine situations where the corresponding application is not always able to immediately process these updates, e.g. because it is too slow, because the connection is unreliable etc. This is a second cause for asynchronous interaction. In that case, the propagation of updates will be packaged by the co-ordination agent as (*XML-*) *messages*, which are *queued* until the application is able to process the updates (in particular, this situation would occur if in the future part of the functionality is delegated to an application service provider, who publishes its software as a Web Service). Information is passed from the Enterprise Layer to the application when the queued updates are processed by the application, resulting in updates to the local database.

Table 3 summarizes the different interaction modalities between application and Enterprise Layer. "A \rightleftharpoons E" represents an information flow from application to Enterprise Layer. "A \leftarrow E" represents an information flow from Enterprise Layer to application.

| | Data access by application | Update by application | Update in Enterprise Layer, caused by other application | Polling on application's local database | Application processes message in queue |
|---|----------------------------|-------------------------------|---|---|--|
| Synchronous interaction, no replication | A \rightleftharpoons E | A \rightleftharpoons E | | | |
| Synchronous interaction, with replication | | A \rightleftharpoons E * | A \rightleftharpoons E | | |
| Asynchronous interaction | | | | A \leftarrow E * | A \leftarrow E * |

Table 3. Summary of the possible interaction mechanisms between application and Enterprise Layer

Transaction management

Transaction management can easily be directed from the session Bean where a business event was triggered. The processing of a single MERODE event is to be considered as one atomic transaction, which is to be executed in its entirety or not at all. The event is generated by a method call on a session Bean and results in methods being executed in all objects that participate in the event. If none of these raises an exception, i.e. no constraints are violated in any object, the transaction can be committed. If at least one such method *does* generate an exception, the entire transaction, and not only the updates to the object where the exception was raised, is to be rolled back. In this way, the Enterprise Layer itself always remains in a consistent state.

However, if the business event is the consequence of an application first updating shared data replicated in its local database, the co-ordination agent is to inform the application about the success or failure of the propagation of the update to the Enterprise Layer by means of a return value. In case of the event that corresponds to the update being refused in the Enterprise Layer, the application should roll back its own local updates too, in order to preserve consistency with the Enterprise Layer, which contains the primary copy of the data. Cells marked with a “*” in Table 3 denote situations where there is a risk of inconsistency between application and Enterprise Layer. As can be seen, the issue is particularly critical in the case of asynchronous interaction.

Applications and user interfaces

The existing BSS/OSS applications interact with the Enterprise Layer through the co-ordination agents. They are virtually unaffected by being plugged into the Enterprise Layer. Co-ordination agents are to be rewritten when an application is replaced, such that the applications themselves and the Enterprise Layer can remain untouched.

The existing user interfaces of the third-party BSS/OSS applications are proprietary to these applications and were of no concern during the integration project. Apart from these, new web-browser based user interfaces have to be developed in-house for direct interaction with the Enterprise Layer. This can be accomplished fairly easily by means of Java Server Page technology. This issue is, however, beyond the scope of this case study.

CURRENT CHALLENGES/PROBLEMS FACING THE ORGANIZATION

Once the solution was specified, the main challenge resided in the successful implementation of the proposed solution. The choice of the implementation environment was a key influencing factor for success. It is the company's policy to outsource IT development efforts as much as possible. The requirements engineering task is done in-house, but system realisation should be done by buying off-the-shelf software or by outsourcing. In addition, coding efforts should be limited by utilising code generators where possible.

Considerable complications resulted from the implementation environment that was chosen by the company: an application development tool that generates Enterprise Bean code, but that is nonetheless not fully object-oriented. In fact, by no means can it conceal its origin in the *relational* paradigm. Its so-called “objects” actually behave like single rows in a relational table: they are interrelated by means of foreign keys (instead of object identifiers) and lack the concepts of subtyping, inheritance and real behaviour. Moreover, whereas MERODE defines *shared participation in business events* as the enterprise objects' means of interacting, the tool only supports event notification at the level of *inserts, updates* and *deletes*. These are not very suitable for modelling business events: they pertain to the actual changes to rows in the database, but do not carry information about *which business event* induced the change, ignoring the possibility of different underlying semantics, preconditions and constraints. Therefore, whereas an almost perfect mapping existed between concepts from the MERODE based Enterprise Model and the proposed EJB architecture, several of the MERODE model's subtleties were lost when being translated into the design environment.

Furthermore, MERODE advises a "transformation"-based approach to implementation: the transformation is defined by developing code templates that define how specifications have to be transformed into code. The actual coding of an Enterprise Model is then mainly a "fill in the blanks" effort. MERODE provides a number of examples for transformations to purely object-oriented program code. Also a lot of experience exists with transformations to a COBOL+DB2 environment. Transformation with a code-generator had never been done before.

Two additional elements were important in further evolution of the project. The company had explicitly opted for the development method MERODE by asking the MERODE-team for assistance in the project. Although the method is in use by several companies and has in this way proven its value, it is not widespread enough to assume that it is known by the programmers of the implementation team. In fact, no members of the implementation team were acknowledged with the chosen modelling approach MERODE. The same was true for the members of the consultancy team of the company providing the implementation environment.

Finally, on the longer term there might be a problem of domain model expertise. Initially, it was agreed that the company would appoint one full-time analyst to actually do the requirements engineering. The university team would advise this person. However, due to shortage on the labour market, the company did not succeed to find a suitable candidate. As a result, the requirements engineering has completely been done by the university team. Consequently, valuable domain model expertise resides outside the company.

REFERENCES

Booch, G., Rumbaugh, J., Jacobson, I. (1999). The unified modeling language user guide. Reading: Addison-Wesley.

D'Souza, D.F. & Wills, A. C. (1999). Objects, Components and Frameworks with UML. The Catalysis Approach. Reading: Addison-Wesley.

Fowler, M. & Kendall, S. (1998). UML Distilled: applying the standard object modeling language. Reading: Addison-Wesley.

Gamma E., Helm R., Johnson R. (1999). Design patterns: elements of reusable object-oriented software. Reading: Addison-Wesley.

Jacobson, I., Christerson, M., Johnsson P. (1997). Object-Oriented Software Engineering. A use Case Driven Approach. Reading: AddisonWesley.

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, 24(24), 233-251.

Snoeck M., Dedene G., Verhelst M., Depuydt A.M. (1999). Object-oriented Enterprise Modeling with MERODE. Leuven: Leuven University Press.

Snoeck M., Poelmans S., Dedene G. (2000). A Layered Software Specification Architecture. in Laendler A.H.F., Little S.W., Storey V.C. Conceptual Modeling -ER2000, 19th International Conference on Conceptual Modeling. Salt Lake City, UTAH, USA. Lecture Notes In Computer Science 1920. Springer. 454-469.