Proceedings of the *VLDB Workshop on Technologies for e-Services (VLDB-TES 2000)*, Cairo, Egypt, 2000

Designing Components for *e*-Services

Barbara PERNICI
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
+39 02 23993526
pernici@elet.polimi.it

Massimo MECELLA*
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy
+39 06 49918479
mecella@dis.uniroma1.it

Abstract

Component based approaches are becoming more and more popular to support distributed application development. The concept of component itself, however, is not generally agreed upon and several definitions can be found. Moreover, different approaches to object oriented component modeling obtain different abstraction levels (conceptual vs. operational). In this paper, we discuss the concept of component in the framework of e-Service and e-Application design, where these services are based on legacy systems. We give a precise definition of stateful and stateless components, and we discuss their characteristics and their applicability in different stages of web application development.

Keywords: *e*-Service, *e*-Application, component, wrapper, legacy system, cooperation, web application development.

1. Introduction

The emergence of Internet allows the development of new interaction business paradigms, commonly referred to either as *e*-Commerce or as *e*-Business. There are many other contexts where the use of communication networks and of distributed applications can be taken into consideration in order to offer new added value services to customers. Some important initiatives for the definition of what it is referred to as *e*-Government [2][6] are undertaken in some countries.

As an example, in Italy the project of the Unitary Network of the Italian Public Administration [13][19] aims at implementing a "secure Intranet" that can interconnect the information systems of different public administrations. The emphasis of this project is on promoting cooperation among the various administrations at the application level. Besides providing the essential interconnection services (e-mail, file transfer, etc.) to administrations by supplying them with basic interoperability tools, the project defines the Unitary Information System of the Italian Public Administration as a whole, by bringing together the collection of distributed, autonomous systems of each administration into a common Cooperative Architecture. In turn, this will make it possible to reengineer global administrative processes by making more effective use of the information made available by each individual system.

In this paper, we define an *e*-Service as an application component; an *e*-Service can be used in a portal [18][23], in an *e*-Commerce application, to offer services in a public context (e.g. a service allowing citizens to access and to manage information about their retirement plans). An *e*-Application is a distributed application, possibly a complete system, which integrates in a cooperative way the *e*-Services offered by different organizations.

In the literature, the term Cooperative Information System (CIS) [4][12][15] is often used to define a large number of cooperating component systems, distributed over large, complex computer and communication networks and working together cooperatively, effectively requesting and sharing information, constraints, and goals. We consider an *e*-Application as a particular instance of CIS, in which two dimensions are relevant: it must rely on open architectures and the constraints imposed to cooperating organizations must be as loose as possible.

Building *e*-Applications requires the integration of different heterogeneous systems (at least one for each cooperating organization); these systems are different not only under the technological point of view (they are very often legacy systems), but also for the information content they export. The information content is the information owned by an organization of the CIS. The same information content can be exported with different technologies and with different models, at different abstraction levels. The abstraction level (of the exported information) is indirectly measured through the coupling, both technological and semantic, among the organizations. As an example, consider the information about a citizen in a public administration. The information conveyed in the data stored in the information system of the administration can be exported in several ways: through a record in a mainframe transaction, through a relational table, or through an object Citizen expressed in a middleware IDL. The abstraction levels are different, that is in the latter

* This work was partly supported by the Distributed Object Engineering and Research (DOER) Group in Telcordia Applied Research, Telcordia Technologies, 445 South Street, Morristown, NJ.

situation the coupling, both technological and semantic, between the client and the server organization is looser.

It must be pointed out that this is not just an interoperability problem; many technologies, referred to as Enterprise Application Integration (EAI) [16], allow the communication and data exchange among heterogeneous computing platform. The problem is conceptual, dealing with information modeling: how to export information in order to avoid same of the drawbacks present in current practices: the development of ad-hoc interfaces is costly, difficult to maintain, and limits the applicability of *e*-Applications to very few special cases.

The aim of this paper is to present two different ways of modeling services as components. The first one is *operational*, the services are exported according to the offered functionalities; the resulting software components are *stateless*. The second modeling approach is *conceptual*, it exports the information needed to provide e-Services as entities, to be linked together in a flexible way; the resulting software components are *stateful*. In discussing these two approaches, we point out the difficult case in which the cooperating systems are legacy systems and therefore they must be encapsulated, exporting as few as possible their underlying legacy structure. This situation is very common in the current *e*-Application scenarios, and it is a good starting point to consider the main differences between the two approaches. We discuss how different approaches can be appropriate in different stages of *e*-Application development, showing their characteristics.

The remainder of this paper is organized as follows. In Section 2, some fundamental concepts and background are provided, discussing components and the integration of legacy systems. In Section 3, stateful and stateless wrapper components are introduced, pointing out how they are the result of two different modeling approaches. In Section 4, a discussion on the development of component based *e*-Applications starting from legacy systems is presented. In Section 5, a complete example by using the two different approaches is outlined.

2. Background

2.1. Components and Distributed Object Computing

In general, a *reusable component* can be defined as a unit of design (at any level), for which a structure is defined, a name identifying the component is associated, and for which design guidelines, in the form of design documentation, are provided in order to support the reuse of the component and to illustrate the context where it can be reused, including constraints, for instance, indicating which other components must be used in combination with the one being considered [7].

From this general definition, two more specific ones follow:

- A conceptual component is a model/schema (or a subset of) to be reused; following the object oriented approach, it may be specified with the Unified Modeling Language (UML) [3][8]; other more specific models may be adopted in other application areas, such as for instance workflow management [7].
- A software component is a coherent software package that can be independently developed and delivered, has explicit and well-specified interfaces for the services it provides and for the services it expects from the others, can be composed with other components, perhaps customizing some of their properties, without modifying the components themselves [8][21].

A modeling aspect of software components is the *granularity* [10] at which components are defined. While the classical software development was based on homegrown applications, in the 90s Enterprise Resource Planning (ERP) systems emerged, in which functionally complete subsystems are considered as basic components and an information system is developed by assembling and customizing these components. Recently, the trend is towards finer grained components and application frameworks, which include several related aspects modeled together. Other approaches consider Business Objects as the basic components [20][24], that is the classes represented in the conceptual model correspond to the software components.

The major obstacle to the definition of software components is the need for a common framework, that is the definition of "the world in which the component will live in" [9]. Until now the fields in which the component based approach was successful were those ones in which the framework is well defined: e.g., the development of graphical user interfaces, in which the framework is the operating environment or the virtual machine.

The new approach referred to as Distributed Object Computing (DOC) [17][28] is based on the merge of two trends: distribution and middleware technologies and Object Orientation. The computation is performed through messages that objects developed in different programming languages and deployed on heterogeneous hardware and software platforms exchange through the network. The technology of the Component Transaction Monitors (CTMs) offers both the bus allowing various objects to communicate in a transparent way and a standard component model; it is the middleware layer enforcing the important separation of concerns between the design of added-value, business-aware services and their actual deployment. A CTM defines the framework of services and interfaces on top of which it is effectively possible to develop and deploy software components. These components are not restricted to the presentation layer, but

they represent business logic to be reused. The component model used by CTMs is standard, thus a component is pluggable on CTMs from different vendors; it is not tied to the platform it was originally developed, but it is portable among different platforms. Currently there are three main component models: the OMG CORBA Component Model (CCM), the Enterprise JavaBeans (EJB) architecture and the Microsoft Component Object Model (COM+).

2.2. Integrating Legacy Systems in e-Applications

Many existing organizations, when they decide to develop *e*-Applications, need to address the issue of their legacy information systems: only very new organizations (e.g. start-up companies) succeed in developing from scratch both the backbone information system and Web applications; on the other hand, banks, public administrations, manufacturing companies, and so on, have already their own systems, very often with all the relevant data about customers, products and services. What they need is integration, in a seamless way, of information and procedures already running on their legacy systems.

Legacy systems are defined as applications of value (critical to the business) that have been in production for five or more years (according to this definition, most applications currently in production can be considered as legacy) [27]; in [27] it is presented a classification of architectures of legacy systems into four categories. Among the different reengineering strategies that have been proposed for dealing with legacy applications, the following two can be considered in the development of *e*-Applications [5][27]:

- Integrate: consolidate the legacies into the current and future applications.
- Gradual Migration: rearchitect and transition the legacy system gradually.

The former approach allows accessing legacy data and attempts to integrate legacy and new applications requiring only minimal modifications to the legacy system. The result of Integration is a final composite system where the old applications are not replaced. The Migration approach, conversely, produces a new system that completely replaces the old one, possibly by using intermediate and partial integration steps. In the development of *e*-Applications, Gradual Migration is a viable option if the time constraints are not very strict and only inside single organizations. Inter-organizations architectures can only be based on Integration, since it only defines interfaces and respects the autonomy of individual organizations.

A viable method for the Access and Integration of legacy systems is based on object wrapping [27][29]. Specifically, access wrappers are used to provide external access to legacy applications, while object wrappers facilitate their integration:

- Access wrappers simply provide a view of existing access functionality, by providing a new interface that corresponds exactly to the available data and application access paths.
- Object wrappers provide a higher level of abstraction, by implementing new interfaces that do not necessarily map exactly to existing information access paths. While the external view of the information appears as a self-consistent OO schema, its implementation relies on the coordinated use of multiple access wrappers in order to present an integrated view of the underlying data, which conforms to that schema.

3. Stateful vs. stateless wrapper components

e-Services can be implemented as components; with the term *component* we mean a set of object oriented (OO) classes assembled together to be deployed as a single software unit, with explicit and well-specified interfaces for the services they provide and for the services they expect from other components; a component can be composed with other components without modifying it. The term *component instance* is used to distinguish the specification of a component and the executable that implements that specification from a particular installation of that executable and a "running" incarnation of that executable that is available as a server. In particular the component instance is the object (set of objects) which is the runtime manifestation of a component when composed within a particular application [8].

In the following, we discuss *components as wrappers of legacy information systems for building e-Applications*. The context we consider is that of an organization which needs to export its information in order to cooperate with others. The organization can model this information according to two different component based approaches: (i) stateful (conceptual) and (ii) stateless (operational) components.

In order to define the two different approaches, we introduce the following definitions:

State Association Property: *The specification of a component comprises properties/attributes. A property/attribute must be considered part of the state of the component instance whether it is necessary to maintain its value between two invocations of any service/method on that component instance.*

State Management: *The mechanism used to logically store the state of a component instance, for the duration of an interaction with it (session).*

Therefore, by using these two properties, it is possible to precisely define both stateful and stateless components as follows:

Stateful Component: *component specification provides the State Association Property and the State Management of its instances is carried out by the server side of the distributed client/server application* (see class `Account` in Figure 1a).

Stateless Component: *either one or the other of the following two situations can occur:*

- *The component specification provides the State Association Property, but the State Management of its instances is carried out by the client side* (see class `Payment` in Figure 1b).
- *The component does not provide the State Association Property (a pure function).*

```
class Account {

    // attributes
    m_accountNumber As Integer;

    // methods
    void withdraw (money As Currency);
    void deposit (money As Currency);
}


// client code
Account C1;
Account C2;
Currency money;

// omitted code to obtain the two objects
… …

C1.withdraw(money);
C2.deposit(money);
```

(a). Stateful design: definition of the conceptual class `Account` and client code

```
class Withdraw {

    // methods
    void withdraw (accountNumber As Integer,
                   money As Currency);
}

class Deposit {

    // methods
    void deposit (accountNumber As Integer,
                  money As Currency);
}
```

```
class Payment {

    // methods
    void pay (accountNumberFrom As Integer,
              accountNumberTo As Integer,
              money As Currency) {
    … …
    Withdraw w = new Withdraw;
    Deposit d = new Deposit;
    … …
    w.withdraw(accountNumberFrom, money);
    d.deposit(accountNumberTo, money);
    … …
    }
}



// client code
Integer accountNumber_numC1;
Integer accountNumber_numC2;
Payment payment;
Currency money;

// omitted code to obtain the two numbers
… …

payment = new Payment;
payment.pay(accountNumber_numC1,
            accountNumber_numC2, money);
```

(b). Stateless design: definition of the operational classes `Withdraw`, `Deposit` and `Payment` and client code

Figure 1. An *e*-Service for the payment, which consists in withdrawing money from an account C1 and deposit them into an account C2[1]

We want to remark that the main difference between the two design choices is that in the former case a conceptual object (as in the real world) is identified, and it is directly represented in the software component (the class `Account` in Figure 1a). In the latter case, there is not an immediate object of the world, but the operations required are modeled (the classes `Withdraw`, `Deposit` and `Payment` in Figure 1b). Note that when considering the two approaches, we are not dealing with the data used by the system or stored in the back-end databases: clearly data about the payment must be present in both approaches. The second design reproduces quite naturally the procedural transactions of the legacy system, while the first one requires an integration layer which resolve the mismatch among the OO conceptual view and the actual access mechanisms.

Stateless components differ from the stateful ones for different point of views: they represent collections of ser-

[1] The example uses a pseudocode Java-like. In particular it is available a base type Currency for money variables.

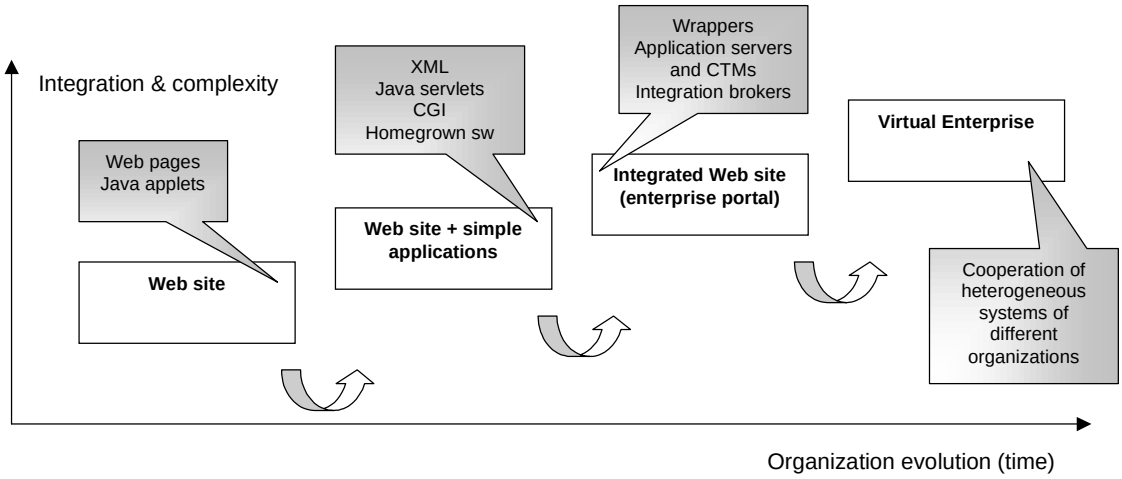

Figure 2. The 4 stages of evolution towards virtual enterprise and *e*-Applications

vices (1:1 relationship between a service and a method), instead stateful components represents "concepts", that is things of the real world. A stateless component is a way to provide an object oriented interface to existing legacy functionalities, without a real underlying object oriented model. The method invocations on the same stateless component instances are not related, since the component does not preserve a state, possibly it obtain it embedded in the method invocation itself (e.g. `accountNumber` in Figure 1b): two different invocations of the same methods executed by the same client could be served from different instances. Instead a stateful component instance typically is associated with a particular client, because it represents a view on concepts relevant only for that client.

Stateless components are "operational", that is the operation they offer are functions, taking input information, elaborating it and returning a result, without visible side-effects on the state of the component. Note that this does not mean that the back-end systems are not modified; on the contrary, the state of the legacy information base has been modified, typically through a legacy transaction. The absence of visible state changes is with respect at the component interface: the component does not offer attributes and therefore its state is unmodified by the method invocations. This can be explained considering that the component is a wrapper.

In stateful components, the interface export objects, which correspond to the conceptual entities considered in the information base of the legacy systems. That is the component exports an OO schema which is the "conceptual" model of the information asset of the organization: the information is modeled as classes and associations, using the typical notation of the UML class diagrams. The schema offers integrated views over data and services, as if the system were a "virtual" object oriented database. The component instances offer objects and links which are instances of the classes and associations from the conceptual model. A client organization, willing to access the information asset, needs to access some objects and to follow the links among different objects. These objects and links are the instances of the classes and associations represented in the OO schema, and through their properties and operations they export the information. The act of determining which classes to use and which associations to follow is referred to as the "navigation" of the conceptual model. After determining the particular paths, a software application to effectively access the objects will be developed (*e*-Application). The "navigation" of the OO schema by the client organization is a conceptual step: during the development of a new cooperative application, the client organization considers the OO schema in order to identify how to access the exported information.

Both component types can be designed and deployed by using OO principles and languages. Stateful components are more suited to be implemented by OO languages, while stateless components could be implemented either through OO languages or procedural ones, because the interfaces they expose are "procedural". The principles of late binding and polymorphism are valid for both, while the design principle that software objects corresponds to conceptual real ones is valid only for stateful component.

4. Development of e-Applications

4.1. Stages of e-Applications

The development of *e*-Applications leads to the notion of virtual enterprise [1][25], that is an enterprise whose

business processes are constructed by combining the services provided by different organizations. In this way, the information and communication technologies allow the business processes to go beyond the organizational boundaries: tools and services of different organizations are the building blocks (*e*-Services) of a higher level system supporting cooperative processes and data flow within the cooperating organizations.

An organization which aims at cooperating with others through *e*-Applications typically gets to the virtual enterprise concept evolving through 4 stages [14][22][26], shown in Figure 2.

The first stage represents the simple Web site, in which the only *e*-Service offered is advertising through Web pages, possibly embedding Java applets. The second stage adds to the simple Web site the opportunity of exploiting simple applications (*e*-Services), as the retrieval of information from back-end databases of the organization and remote data entry of some information about customers and orders. Typically the technologies used at this stage are based on Web pages, XML, Java applets and servlets, CGI and homegrown software for the communication with the back-end databases. In these two stages the organization is completely independent, the only cooperation with other organizations consists possibly in links among their sites.

The third stage integrates the legacy applications of the organization with the Web front-end, allowing customers to effectively use *e*-Services. Typical examples are enterprise portals; the technologies used at this stage are based both on the technologies used in the previous stages for the Web front-end, and on a set of middleware technologies (i.e. integration brokers, mediators and legacy wrappers, application servers and component transaction monitors, etc.) for the integration and interoperation of the back-end systems.

Finally the last stage is the virtual enterprise, that is a network of organizations in which heterogeneous information systems cooperate in order to offer services to customers. The technologies used are mainly the same used in the previous stage, the main difference is that in the third stage the integration is mainly intra-organization, while in the last one is completely inter-organizations.

4.2. Stateless and Stateful component based development of e-Services

e-Services can be developed as software components, that is as a set of related distributed objects. In particular, when the back-end applications are legacy systems, such components are wrappers over the existing functionalities.

In Figure 3 a comparison among the two different approaches for designing wrapper components presented in Section 3 is shown.

Components / Characteristics	Stateful	Stateless
Wrapping type	Object	Access
***e*-Service design complexity**	High	Low
Component development time	Long	Short
Integration logic	Distributed	Centralized
***e*-Application composition**	Easy	Difficult
***e*-Application development time**	Short	Long

Figure 3. Characteristics of component based development starting from legacy systems

From the complexity of stateful design stems that required *e*-Service development time is longer than in stateless design. This issue must be considered when the time constraints are very strict.

Conversely, as regards the characteristics of the overall *e*-Application, it must be pointed out that the main difference between the two designs is the "fatness" of the *e*-Application: in the stateless case, it carries out the integration logic, it is a "fat" application, while in the stateful case the *e*-Application is "thin", simply managing the "navigation" among objects. In the stateless design, the *e*-Service interfaces exactly reproduce the "legacy" features, therefore, as legacy applications are very often vertical and non-integrated, it is necessary a centralized integration logic. In the stateful design, the interfaces exported by the different organizations are semantically rich (they carry out the integration logic pertaining to the particular organization), thus the *e*-Application has only to "navigate" among them, because each object hides its "legacy" features; we can say that the integration logic is distributed among the *e*-Services assembled to build the *e*-Application.

Clearly it stems that the complexity of the *e*-Application and its development time result inferior than in stateless design. Moreover, the *e*-Application flexibility is increased by stateful design: adding a new organization is not very complex, because each organization carries out its own integration logic and the objects exported are always the same (for the same kind of organization). In stateless design, every time it is necessary to add a new organization, all the integration logic must be modified, because it is based on an organization-to-organization basis, while in the stateful situation the integration at the conceptual level encapsulates all the differences.

From the previous comparison, we draw the conclusions shown in Figure 4.

When the scope of the *e*-Application is restricted to a single organization (2nd and 3rd stages) stateless design is appropriate, because the centralization and the "fatness" of the integration logic are not a dramatic issue, being involved only one organization. On the other hand, in this

case stateless design results in shorter development time. Stateful design can be chosen as an alternative in the 3rd stage with two goals: (i) a flexible *e*-Application (if several modifications can be anticipated) and (ii) preparation of the next stage.

On the contrary, in the 4th stage, when *e*-Services are mainly concerned with different organizations, which need to cooperate with as loose constraints as possible, the encapsulation, abstraction level and distributed integration logic offered by stateful components is absolutely needed. Therefore we argue that the 4th stage requires stateful (conceptual) components.

Stages of *e*-Applications	Suitable Component type(s)
1st (*Web site*)	n.a.
2nd (*Web site + simple applications*)	Stateless
3rd (*Integrated enterprise portal*)	Stateless / Stateful
4th (*Virtual enterprise*)	Stateful

Figure 4. Suitable wrapper component types for *e*-Application stages

5. A complete example

In this section a complete example will be carried out, starting from the same initial situation and designing both stateful wrapper components and stateless wrapper components. The context is the one of a simple explanatory *e*-Application for buying goods in remote way. The involved organizations are a Bank, which holds the accounts of the customers, and the Virtual Shop, which offers its catalog and the opportunity to customers to buy goods. These items will be delivered by a third organization, the Deliver Company, which is actually part of the cooperative system, but it will be no further considered in this example.

The back-end legacy information systems are "program decomposable"[27], that is semistructured systems in which applications can be separated into two units: the interface processing unit, and a combination of business logic and database processing unit. For systems in this class, the data layer can only be accessed through a set of predefined functions, and never directly (i.e. through a direct query interface). In the legacy jargon, these functions (a mix of data and business logic) are called transactions. The word "transaction" should not be confused with the database-centric meaning of the same word in a modern client/server DBMS based environment: on a mainframe, every programming entity running under the control of a software manager is called a transaction. Refer to [11] for a brief and precise overview of the mainframe infrastructure.

The Bank legacy information system provides some transactions: the first one, namely Tran_CUST, taking in input an identifier (proprietary of the bank system) of the customer, returns the personal data about the customer, formatted in a proprietary way according to the record I/O layout of the transaction. The second transaction, namely Tran_LIST, taking in input the same identifier used in the previous transaction, returns the list of identifiers (proprietary of the bank system) of all the accounts owned by the customers, together with few other information about each account. Note that for different causes, to be found in the evolution of the legacy information system, not all the accounts are stored in the same database; therefore it happens that according to the identifier of the account, different transactions must be invoked to perform account management (i.e., each transaction performs the same logical operations on different non-integrated databases). These transaction will be referred to as Tran_ACC_i, all functionally equivalent but with slightly different record I/O layouts.

The Virtual Shop owns a legacy database, with a procedure for the order entry, i.e., it takes in input the list of codes of the ordered goods and triggers the process of deliver preparation. The Virtual Shop was specialized in telephone shopping, this is the cause why this procedure is already running: it was used by the operators at the call center to receive customer orders. The customer management is provided through a modern relational database, but it is completely non-integrated with the other legacy applications.

Starting from this situation, we want to outline the design of an *e*-Application allowing customers to buy goods through a computer network (e.g. Internet) and automatically have the money transferred from the customer account to the Virtual Shop account (assuming, for simplicity, that the Virtual Shop owns an account in the Bank). *e*-Applications of this kind are nowadays very common, we want to point out how a conceptual design can obtain better flexibility and a looser coupling among cooperating organizations with respect to an operational design.

5.1. Stateful design

With a stateful design, it is needed to design a conceptual OO schema of the information on top of which the organizations must cooperate. This model is shown in Figure 5.

The interface `Customer` is used as gateway among the two systems of the Bank and of the Virtual Shop[2]. The

[2] In practice, each organization exports a "bootstrap" object (e.g. the Bank offer an object of class `BankServer`), a Factory through which clients can obtain instances of `Customer` that are specialized for a particular organization The Factory provided for each organization supplies specialized objects through the standard method `FindCustomer()` (e.g. it returns `BankCustomer` objects). The common shared interface for the Customer entity is central to the interoperation across different organizations. In a typical flow, a client

Bank system offers `BankCustomer` and `BankAccount` classes; when a `BankCustomer` object is allocated, the transactions of the legacy systems are invoked, all the data are collected and integrated, the related `BankAccount` objects are allocated and the links among them are created[3]. The same is true when a `ShopCustomer` object is allocated. Note that the allocation of an object consists of retrieving all the needed data from the back-end systems and fill in the opportune attributes of the objects; these objects "live" for the session duration in the wrapping components exporting the information. The *e*-Application simply manages the link among these objects; for example, when a customer wants to buy some goods, through the Web interface he introduces some data identifying himself (e.g. userID and password). The related `ShopCustomer` object is allocated, together with a new `ShoppingBag` object. During the session, the products acquired are added to the `ShoppingBag` object. Finally, when he stops shopping, the *e*-Application stores permanently the data (through the transactions wrapped in the component) and asks to the Bank for him, passing the object. The Bank, through the common interface, is able to allocate its own `BankCustomer` object with its `BankAccount` objects. The *e*-Application asks to the Bank also for the VirtualShop, in order to have the `BankAccount` object of the VirtualShop. Finally the methods on the `BankAccount` objects for the billing are invoked (Figure 6).

From a technological viewpoint, the classes of the components are distributed middleware classes, for example EJBs or CORBA classes. The *e*-Application consists of a simple Web front-end, e.g. a servlet, with which the human customer interacts; it manages the "navigation" among the objects. Note that each system maintain all the business logic needed to allocate its own objects, and the exchanges among systems are carried out through objects modeled in a conceptual way.

organization A starts by obtaining an instance α of class `CustomerA`, filling in enough information to identify a particular customer. `CustomerA` implements the generic `Customer` interface for organization A. In order to obtain information and services about that customer from organization B, the client invokes method `FindCustomer()` on the well-known bootstrap object of B, providing α as an input parameter. B uses α (in particular the interface `Customer`) to derive key data that can be used to obtain the information about the customer from its back-end systems, resulting in a new instance β of class `CustomerB` that can be returned to the client. Using β, the client can then further "navigate" organization B to obtain additional information about that customer [13].

[3] In particular, it is firts invoked Tran_CUST, then Tran_LIST and for each account the relative Tran_ACC_i.

5.2. Stateless design

In stateless design, the classes and objects exported are simple access wrappers over the preexisting transactions. This means that objects are at low abstraction level, not very integrated among them and it is needed to exchange also the proprietary identifiers. Each class simply reproduces the transaction it is a wrapper for, and the *e*-Application carries out all the integration logic. When a customer wants to buy some goods, through the Web interface he introduces some data identifying himself (e.g. userID and password); the acquired goods are temporarily stored and finally the transaction is invoked. Note that all these operations are performed by the *e*-Application, while in the previous design each of them was the result of the cooperation of different objects, possibly located on different components. As regards the billing, the bank is directly invoked, that is the class `BankServer` offers directly the method `pay(accountID,money)`. In this case the *e*-Application must know which codes to use, at least the identifier of the customer.

The used technologies are the same, but there is a big amount of exchange of proprietary information. This data exchange can be conveniently carried out through the use of XML as interoperable language, or through the use of flat middleware structures.

We want to remark that the final result is the same, the difference is the abstraction level of the interfaces exported by the different organizations: in the former case they are semantically rich, the *e*-Application has only to "navigate" among them, because each object hides its "legacy" features. Instead in the latter design the interfaces exactly reproduce the "legacy" features, therefore, as legacy applications are very often vertical and non-integrated, it is necessary a centralized integration logic.

6. Concluding remarks

In the present paper, we have introduced the concept of stateless wrapper component, as the basis for providing access to existing functionalities, and stateful wrapper component, providing a more abstract and conceptual object oriented view to existing legacy applications.

The general requirements of development in the domain of web based applications is rapid application development and a flexible composition of different *e*-Services, subject to frequent modifications. To achieve this goal, both stateful and stateless components can be used to build an *e*-Application, and in the paper we have discussed the principal characteristics of each approach to create *e*-Services and *e*-Applications.

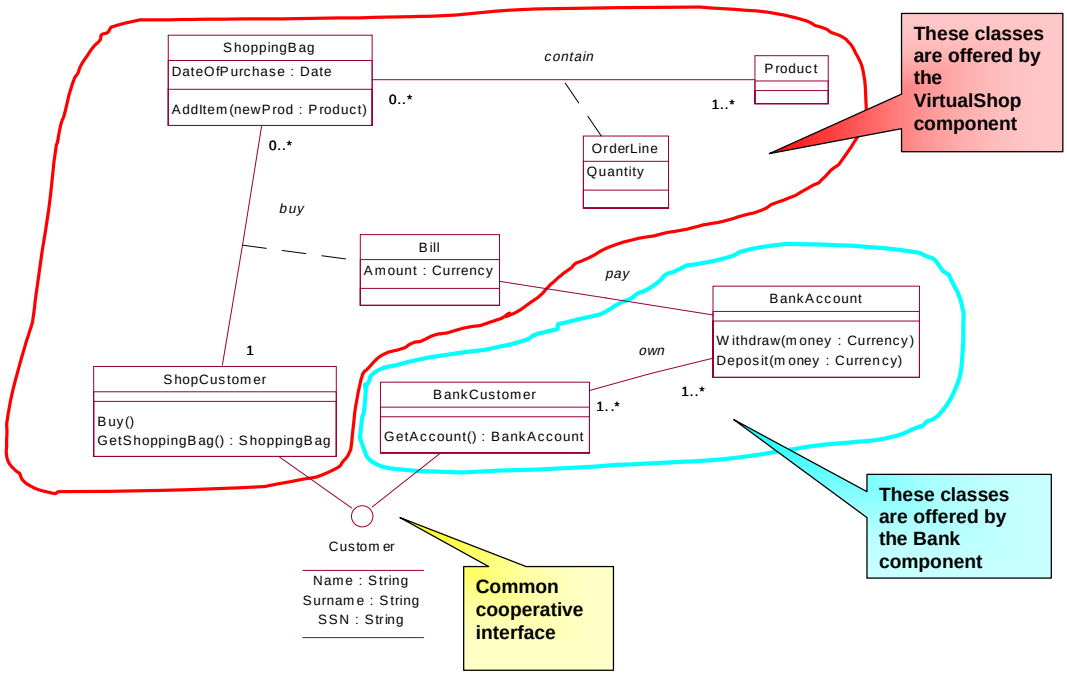

Figure 5. The OO schema of the stateful design

In this paper, we have showed that the general requirements mentioned above can be better focused examining the stage of web development in which the organizations, willing to develop *e*-Applications, can be positioned. For each stage, different design and development solutions can be more appropriate to realize effective *e*-Applications based on components exploiting legacy systems.

Future research work is needed in this direction. One important issue concerns the adaptation of stateful components to specific needs of different *e*-Applications. For instance, the `Customer` object can present different attributes if used in a medical application domain or in a virtual bookstore. An application specific view of each object might be needed to provide applications with services which are tailored to their needs, from different point of views: both the semantics and quality of information and performance. Even in a stateless approach some sort of filtering of provided functionalities could help in providing better *e*-Services.

Further investigation is also needed to provide more detailed criteria to evaluate at design time the more appropriate choices concerning whether to develop stateless or stateful components.

Acknowledgements

Some of the ideas exposed in this paper stems from the summer internship that Massimo Mecella held in Telcordia Technologies. Thanks go to Amjad Umar, Paolo Missier and Francesco Caruso.

Thanks to Carlo Batini (AIPA) for important discussions about many issues dealt in this paper.

Special thanks to Monica Scannapieco who studied some issues about distributed components during her master thesis and suggested some definitions and examples.

References

[1] Alonso G., Fielder U., Hagen C., Lazcano A., Schuldt H., Weiler N.: "WISE: Business to Business E-Commerce". Proceedings of the *9th International Workshop on Research Issues on Data Engineering (RIDE'99): Information Technology for Virtual Enterprises*, Sydney, Australia, 1999.

[2] Autorità per l'Informatica nella Pubblica Amministrazione (AIPA): `http://www.aipa.it/english[4/`.

[3] Booch G., Rumbaugh J., Jacobson I.: *The Unified Modeling Language User Guide*. Addison Wesley, 1998.

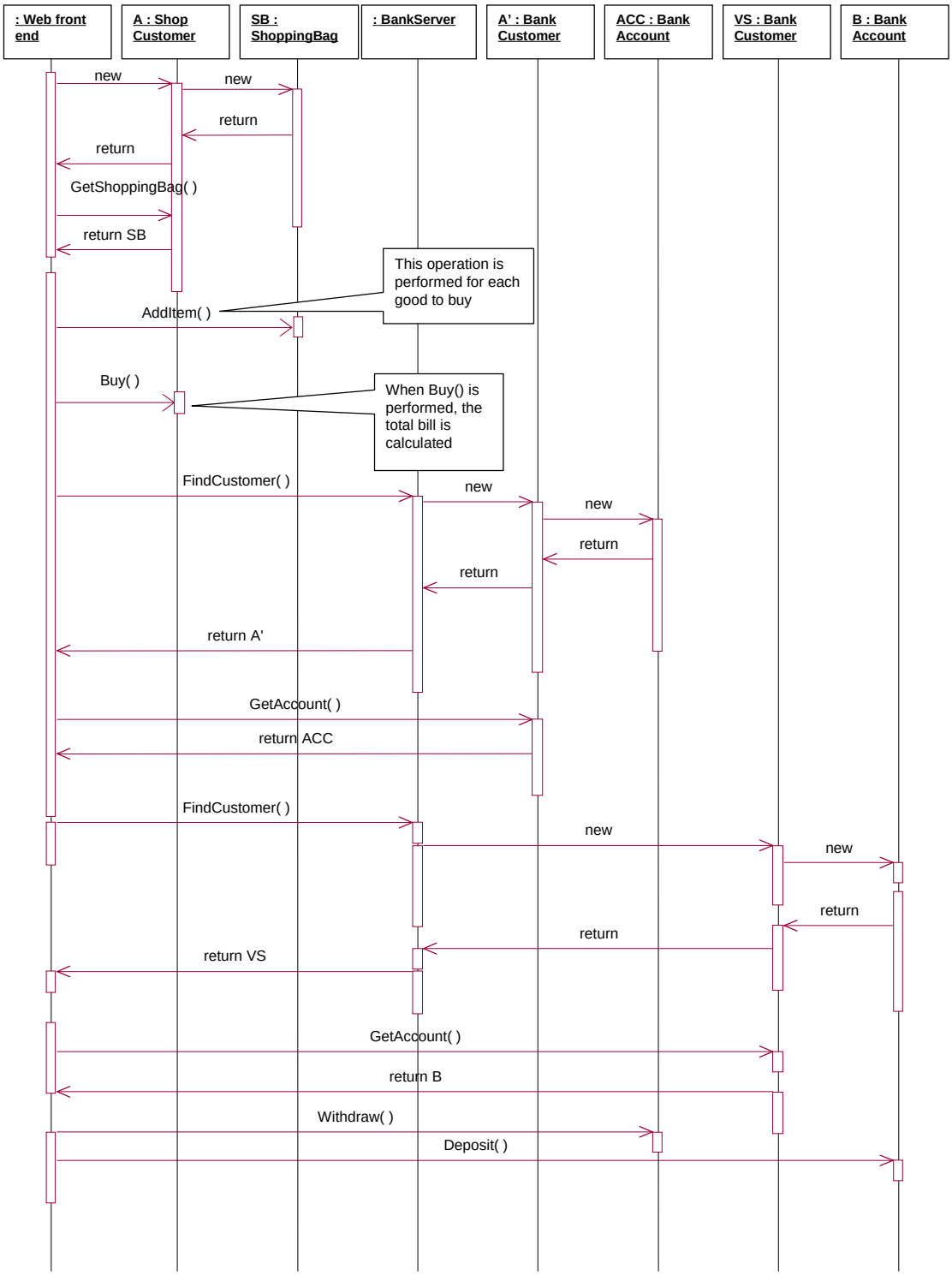

Figure 6. Sequence diagram of an *e*-Application scenario

[4] Brodie M.L.: *The Cooperative Computing Initiative. A Contribution to the Middleware and Software Technologies*. The Cooperative Computing Initiative, January 1998.

[5] Brodie M.L., Stonebraker M.: *Migrating Legacy Systems: Gateways, Interfaces & The Incremental Approach*. Morgan Kaufmann, 1995.

[6] Cabinet Office CITU Central IT Unit: *UK Government Interoperability Framework*. London, 28th March 2000.

[7] Casati F., Castano S., Fugini M.G., Mirbel-Sanchez I., Pernici B.: "Using Patterns to design rules in workflows". To be published on *IEEE Transactions on Software Engineering*.

[8] D'Souza D.F., Wills A.C.: *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison Wesley, 1999.

[9] Fowler M.: *Analysis Patterns. Reusable Object Models*. Addison Wesley, 1997.

[10] Johnson R.: "Framework = (Components + Patterns)". *Communications of the ACM*, vol. 40, no. 10, October 1997.

[11] Koch T., Murer S.: "Service Architecture Integrates Mainframes in a CORBA Environment". Proceedings of the *3rd International Enterprise Distributed Object Computing Conference (EDOC'99)*, Mannheim, Germany, 1999.

[12] Laufmann S., Spaccapietra S., Yokoi T.: "Foreword". Proceedings of the *3rd International Conference on Cooperative Information Systems (CoopIS'95)*, Vienna, Austria, 1995.

[13] Mecella M., Batini C.: "Cooperation of Heterogeneous Legacy Information Systems: a Methodological Framework". Proceedings of the *4th International Enterprise Distributed Object Computing Conference (EDOC 2000)*, Makuhari, Japan, 2000.

[14] Mecella M., Umar A., Missier P.: "Electronic Commerce Workbench (A Decision Support System for Building Web Architectures). A Java Prototype". Presentation of the Summer Internship Project, Telcordia Applied Research, Morristown, NJ, August 1999.

[15] Mylopoulos J., Papazoglou M.: "Cooperative Information Systems". *IEEE Expert*, 1997.

[16] Morgenthal J.P.: "Enterprise Application Integration Tutorial". Presentation at the *OMG EAI Workshop*, Lake Buena Vista, FL, 2000.

[17] Orfali R., Harkey D., Edwards J.: *The Essential Distributed Object Survival Guide*. Wiley & Sons, 1996.

[18] PA Consulting Group: *CITU Portal Feasibility Study*. London, 29th June 1999.

[19] Pernici B., Mecella M., Batini C.: "Conceptual Modeling and Software Components Reuse: Towards the Unification". In Sølvberg A., Brinkkemper S., Lindencrona E. (eds.): *Information Systems Engineering: State of the Art and Research Themes*. Springer Verlag, 2000.

[20] Persson E.: "Shibboleth of Many Meanings. An Essay on the Ontology of Business Objects". Proceedings of the *3rd International Enterprise Distributed Object Computing Conference (EDOC'99)*, Mannheim, Germany, 1999.

[21] Persson E.: "The Quest for the Software Chip. The Roots of Software Components. A Study and Some Speculations". Proceedings of the *1st Nordic Workshop on Software Architecture (NOSA'98)*, Ronneby, Sweden, 1998.

[22] Pezzini M.: "From Business to E-Commerce: Integration Infrastructure as the Enabling Factor". Presentation at the BEA Systems Seminar with Gartner Group, Roma, Italy, 2th March 2000.

[23] Phifer G.: "Enterprise Portals". Symposium Documentation of the *Gartner Group ITxpo'99*, Cannes, France, 1999.

[24] Sims O.: *The OMG Business Object Facility and the OMG Business Object*. OMG Document cf/96-02-03, OMG, Framingham, MA, 1996.

[25] Umar A., Missier P.: "A Framework for Analyzing Virtual Enterprise Infrastructure". Proceedings of the *9th International Workshop on Research Issues on Data Engineering (RIDE'99): Information Technology for Virtual Enterprises*, Sydney, Australia, 1999.

[26] Umar A., Telcordia Applied Research, Morristown, NJ, personal communication, 1999.

[27] Umar A.: *Application Reengineering. Building Web-Based Applications and Dealing With Legacy*. Prentice Hall, 1997.

[28] Wallnau K., Weiderman N., Northrop L.: "Distributed Object Technology with CORBA and Java: Key Concepts and Implications". Technical Report CMU/SEI-97-TR-004, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, 1997.

[29] Weiderman N., Northrop L., Smith D., Tilley S., Wallnau K.: "Implications of Distributed Object Technology for Reengineering". Carnegie Mellon University, Software Engineering Institute Technical Report CMU/SEI-97-TR-005, 1997.