Transforming a company with OOP

Andrea Rinaldi
Director, Business Software Development Group
Microarea S.p.A.
Via Renata Bianchi, 36
16152 Genova
Italy
+39-010-60371
andrea.rinaldi@microarea.it

ABSTRACT

The object-oriented software development paradigm has reached the mainstream application market very soon after its boom in research institutions. However, the lack of an organized work plan has spelt the doom of those who jumped on corporate object orientation without a precise growth plan. Many are the choices that have to be undertaken in such an environment, and some of them affect the fabric of software development in the very core of the business. Object orientation, however, promises to relieve the company from the ever-increasing necessity to adapt every product as soon as a requirement change occurs, by limiting the changes to a restricted number of well-identified code fragments.

This paper presents an account of how object-oriented software development was introduced in our company, the foundation of our understanding object orientation, and the hurdles we overcame in the process of building a reusable application framework.

1. INTRODUCTION

Object-oriented programming is an everyday “chore” for research institutions, but how is it going to happen in a business institution? Which applications are really object-oriented and which aren't? And, above all, is this approach really a winning approach?

Office automation software and basic office software has recently known a very rapid evolution, which has favored downsizing and rightsizing of information systems in the great majority of companies, and that imposed windows as a client platform for all business software. This tendency has created a request for business applications that are more and more versatile and sophisticated, and integrated with office automation applications. Above all, the expectations of medium-level users have grown to the point at which they now require their software to adapt to the workflow in the company in a very rapid and cost-effective ways. Users demand that their word processor, which allows them to typeset pages at their best, be used also for other documents besides letters to their clients. Specifically, they want to be free to send formatted letters regarding all financial measures including cash flow and balances. Conversely, a program as versatile as a spreadsheet, should be able to be customized in order to provide better client handling policies such as discounts and price application politics.

These expectations reflect the modern marketing schemes for software companies that consider software and hardware as a plastic and easily reconfigurable continuum. This is completely different from what was normal to expect in the information systems just about ten years ago.

Corporate software developers need to be able to cope with the current reality of office automation. The new generation of products has to cope with the issues of versatility, cost-effectiveness, continuous improvement, adaptation to company workflow, and easy configurability. These issues are very difficult to cope with, since the price of the products must not increase. In addition to this, there is the additional complication of mass production driven by global economy, thus making the task of software development nearly impossible to accomplish.

Office automation products are realized by big corporations like Microsoft for example, and as such they can benefit from large research and development departments. Business software, on the other hand, does not have the same marketability and as such does not have the possibility to be researched for such a long period. In addition to this issue, there are other more complex issues like for example the necessity to deal with constantly changing rules and regulations like fiscal regulations, which also change from state to state, and from country to country. Business software is requested to integrate seamlessly with any of the applications for office automation, and more, since the business application is the "engine" that needs to support all aspects of the company workstyles. As opposed to using

"just a word processor," or electronic mail, business applications have to deal with the specific workflow and information flow that derives from working in a specific company. As if this were not enough, the continuing lowering of prices for both hardware and software creates the expectation of a comparable price reduction for acquisitions, start-up, and maintenance for business software.

The scenario for business software of these last five to six years has seen the sudden born of the "Second Republic" for business software, with the net distinction between operators that have abandoned the technological challenge, continuing to propose traditional solutions (destined to inevitable oblivion), and those who have tried and found adequate instruments to keep the pace with times (Fig. 1).

Today's solution, we hear saying, is object-oriented programming. It is no longer possible to develop software from scratch. There is a tremendous advantage in taking parts that are already developed, and to modify them, assembling them to obtain the final result. For a single developer, this “recipe” seems simple, and since there are increasingly many object-oriented frameworks being sold, business software houses are tempted to go that way. Although at the beginning the experience is always positive, later development mutates into an impossibly hard task: how is it possible? Is the new object-oriented "religion" treacherous? In order to understand the correct avenue to undertake at our software house, we decided to evaluate the various avenues.

This paper presents the experience of an Italian business software house that has come to have to develop daily updates to their programs in order to comply with the increasingly difficult demands of Italy’s fiscal system. The approach chosen for object-oriented software construction will be presented, and evaluated. At the end of the paper, there will be a conclusion section illustrating the result of applying object orientation to our selected software development group.

2. PROPOSED APPROACH

First of all, we had to make a distinction between real object-oriented programming and what is proposed as object-oriented programming but in reality it is not. Traditional object-oriented programming is based on three main concepts: inheritance, encapsulation and polymorphism. This means that new objects can always be derived from pre-existing objects by inheritance. This would give us a very good advantage in that we would be able to improve objects simply by driving them. Also, it would be possible for us to create abstract objects that encapsulate the basic rules for business procedures. This would allow us to conceive an object as combination of two parts: the first part is the fixed part, that is contained in the abstract class. The second part is the variable part, which is implemented through the new object, changing as soon as the rules change (for example, fiscal regulations change every year or so).

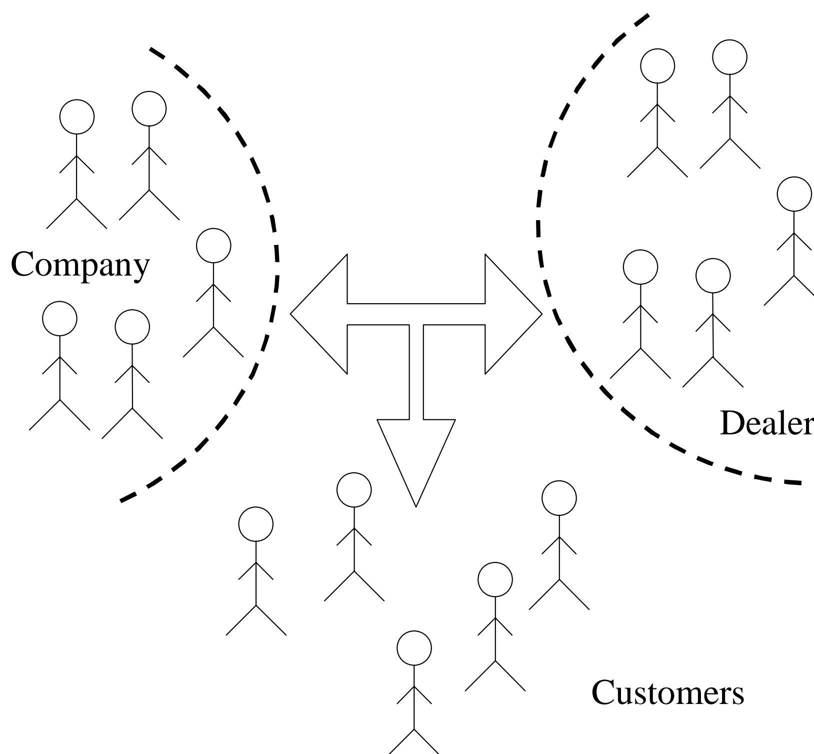

Figure 1: Market driven enterprise and the three-tier market

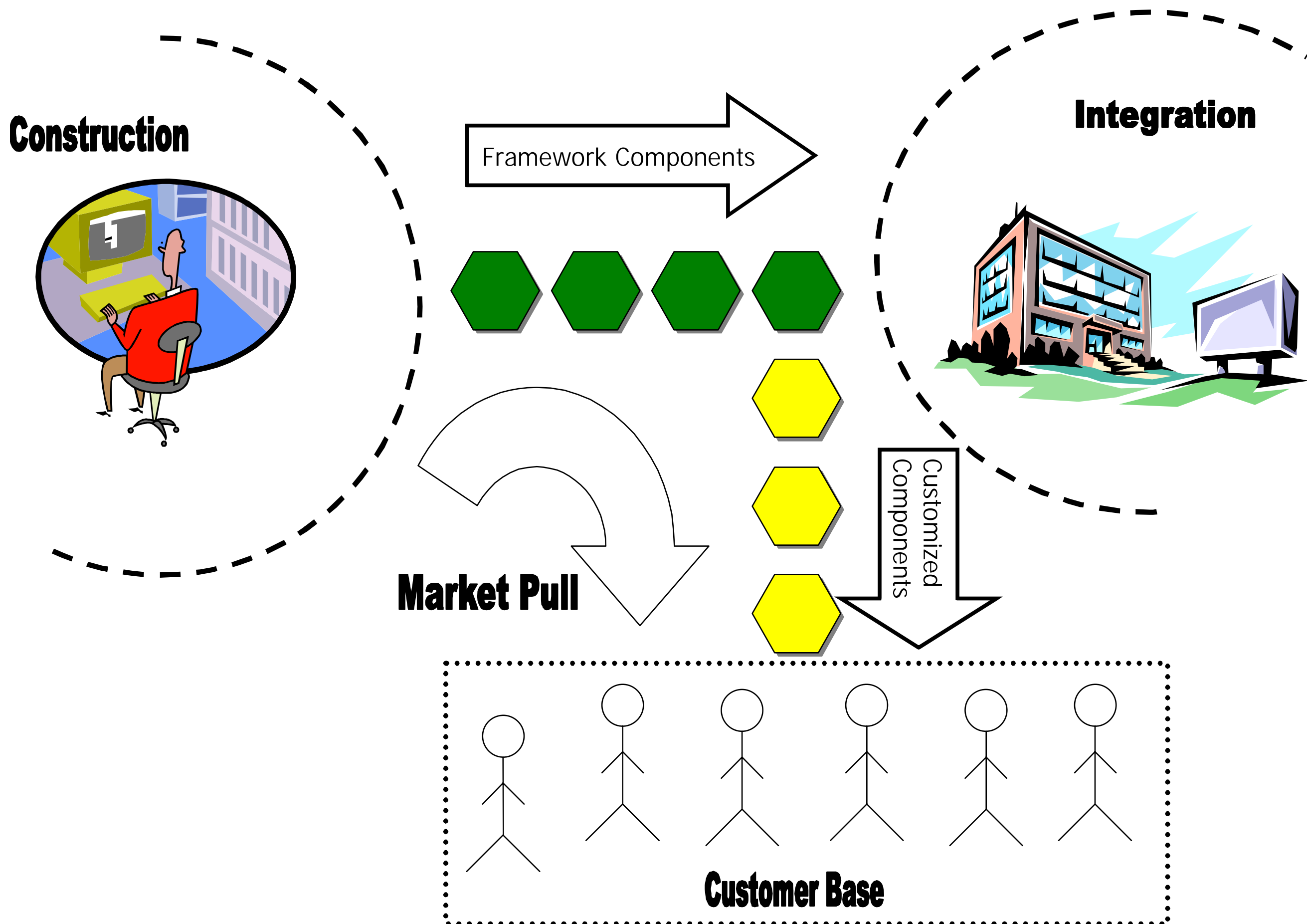

Figure 2: Proposed approach

These possibilities, however, would be useless without a means to capture and exchange experience. This was made possible, at least in the beginning, by libraries. In object-oriented programming, libraries become frameworks. Application frameworks take some time to get used to, as they are philosophically different from the library concepts previously used. In order to effectively begin the construction of a software program that complies with object orientation, it is necessary to choose appropriately not only the library that will build the program, but also the framework into which the program will be contained. We had no other way but to choose to build our own framework. However, this was very difficult to realize until much later in the development stage. In the beginning, in fact, it was even very difficult to choose the correct language for object-oriented development. There are at least three different languages that one can use to develop software using the object-oriented paradigm: C++, Smalltalk, and Java.

For any single object-oriented programmer, it seems very easy to choose one of those programming languages. However, in corporate development, it is very important to choose a strategy that leads to the most cost-effective solution. At the same time, we wanted to continue and transmit and transfer the experience that was proper to all the programmers in our group. Therefore, the most obvious solution for development was C++. Most of the development in the previous products was made either in assembly language, or in C++ already. The C language had been used for some time and had been abandoned after its imitations had become apparent.

Many were the issues that we had to consider while trying to develop a corporate basis for object-oriented development. To begin with, we were really concerned with having a language that allowed us to develop not just single components, but also full-fledged applications. However, it was very important to maintain complete flexibility, to integrate in a market that in general for Italy is divided into many segments. Most of these segments have internal corporate cultures where there has been some in-house software development. Some of that software needs to be preserved since it is usually the only repository for corporate culture and, sometimes, databases. Thus, one of the first and foremost requirements in these kind of applications, is to develop software that is fully integrated with the existing applications. The most important word is "flexibility" (Fig. 2).

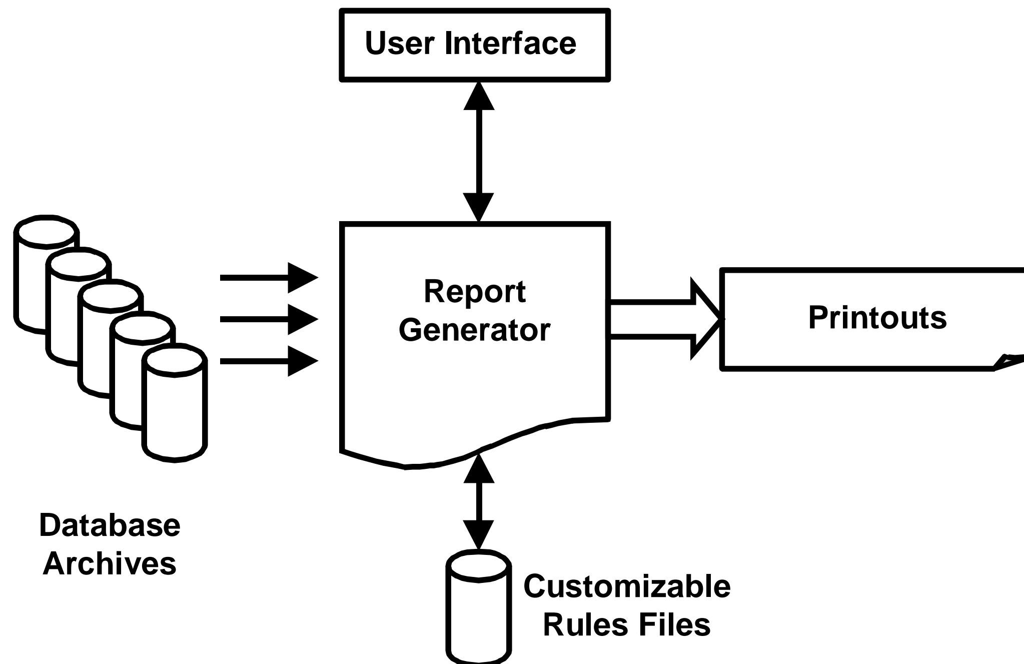

Figure 3: "TASK," first phase

This demand configures a schism: on the one side, there is the development for the market; on the other side, there is the development for the final user. These two development paradigms are different. User-oriented development has to answer the demand of the public very quickly. It is not possible to achieve that level of flexibility just with object-oriented programming. In general, user-oriented development tends to appeal to companies that produce customized software for specific users. Market-oriented software development, on the other hand, is more complex, because it requires careful consideration of the market, possibly involving thorough market analysis, and the designation of a strategy to satisfy the largest customer base by means of a careful analysis of the component base and the development prioritized by component requirement. However, market-oriented software can take advantage of the traditional distribution channel structure without having to be constrained like the traditional package development.

When we began the development of applications in graphical environments (the Italy of 1990 still meant Windows 386 and OS/2 presentation manager 1.1) we decided to take advantage of our experience by creating a series of instruments that would the help us from repetitive work, freeing us from a their repetitive day-to-day necessities. The project was code-named "TASK", acronym of "The Accounting Software Killer." Our intent was to build the starting point that lets us free to concentrate on single problems reusing the work already done. In modern terms, we needed a collection of reusable business software components.

We chose C++ as the computer language to develop all the software. At the time there was no other language available for object-oriented software development, but however even afterwards the platform wasn't abandoned, as it was judged the most powerful and flexible development platform we could find.

3. THE FIRST STEPS

The first problem with tackled was that of reporting. We chose this problem because it is the most general for personalization. The result was WOORM, a report generator put on the market starting from 1993 and to date integrated with all our Windows software solutions. Someone might think that it was a wasted effort, since there are a very large number of report generators available today on the market, but to have not just built it, but also designed it, allowed us to incorporate the many domain-specific characteristics for business software. The result is that WOORM produces all printouts in our applications, from simple customer directories to consolidated balance sheets, with little effort. This specialized component helped us save nearly one thousand development hours in the following years (Fig. 3).

The second problem we decided to solve was that of data entry. The key concept around which all business data entry has to revolve is the business document. We created an abstract class that encompassed the main features of the business document, these being a header, a variable number of lines, each with their own type, and one or more footer information packages. The class must

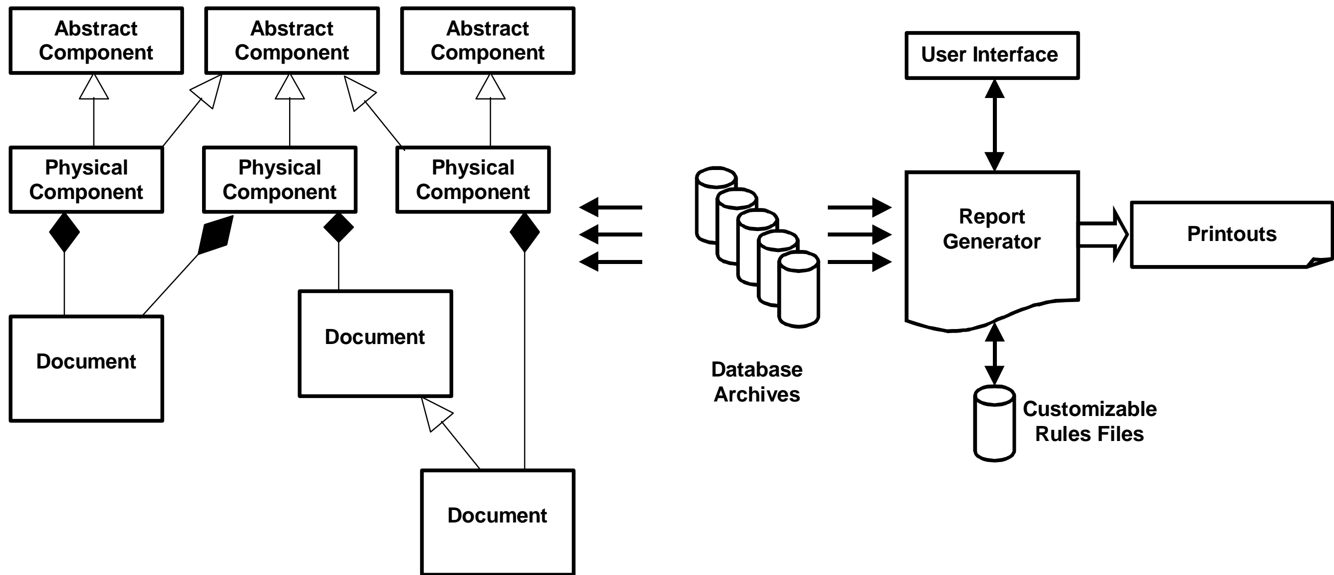

Figure 4: "TaskBuilder," second phase

be able to create a document, modify it, connect it to a database and save it, print it, in full autonomy. This abstract class then is specialized in concrete classes that give "birth" to the concrete business documents. Inheritance insures that the properties of the base class remain, while polymorphism insures that special behaviour be encapsulated where it is needed. In turn, every concrete object can be further specialized, so that a deferred invoice, for example, is identical to an interactive invoice, with the added capability to build itself from the bills and slips already recorded in the database. Other smart objects manipulate field entry, which can involve dates, quantities, currency amounts, or codes linked to extensible tables that allow further manipulation and so on. The availability and high reusability of such components and the presence of the underlying framework saved several hundred hours per development cycle.

At this stage of development, we realized that we had reached a crucial step: the creation of a reusable code base flexible enough to be integrated in every product we produced. This by-product was a major result by itself. We called the resulting framework TaskBuilder (Fig. 4). TaskBuilder was finished in 1996, and it immediately became apparent that it was a valuable commercial asset by itself. It was then decided to commercialize it through our regular market channel, targeting it to the advanced mid- to large-sized company that has a software development unit and highly customized business procedures (or *processes*, as many were starting to say). It was 1996.

4. PUTTING IT ALL TOGETHER

The final step we needed to make is the creation of complex business objects. There are many similarities in simple accounting documents, mortgaging definitions, and invoices, for example: in every case the debit must match the credit, fiscal and internal enumerators must be incremented, and so forth. Many documents also require very similar cross-checks in the company's databases. Most of the times, these cross-checks are nearly the same, while what varies is the subset of the databases that are joined to make the check.

We decided to generate a new set of abstract documents whose specializations would be the business documents we usually work with. This gave birth to a two-level structure: on the one hand, there are two abstract levels, one for documents, and the other for business objects. On the other hand, concrete manifestations of the abstract levels become specialized documents or business objects, ready to work together in the final product (Fig. 5).

This allows to encapsulate all the business components together. The initial framework is fully available to developers, who can specialize every class for particular requirements, or compose a new object by means of the framework's primitives. We can then reach vertical markets with comparatively small effort, building the vertical component on the general-purpose elements we have in the framework. This reduced our time to market, making us highly competitive in the current market.

If we compare this approach to the "componentware" approach described in Section 1, it is apparent that the additional structure that the traditional object-oriented approach provides pays off handsomely. Component-based approaches do not have the cohesive structure that inheritance provides, and as such often mislead developers. Specialization, on the other hand, can be coupled with composition, including the component-based approach, with no hassles.

CONCLUSIONS

We have developed a fully object-oriented architecture that allows us to decouple our software development teams from the competitive peer pressure and from sudden changes in fiscal regulations. Our approach benefits from inheritance and composition, and relies on a framework that assigns precise responsibilities to every business object. This makes it possible to avoid the lack of direction in the componentware approach, giving us a unity of purpose in the software development, and a unity of scope in component augmentation.

To enhance and expand the number of instruments available enriches the dealer's choices, who can better exploit the consolidated know-how, or integrate the technologies from separate vendors, in view of personalizing the application for the needs of specific clients, with reduced times and costs. The installation of the business components on distributed architectures enhances their scalability and improves the interoperability and openness of the business information system.

The future (and by "future" in business we really need to mean a few months) will see the integration of standard interfaces such as COM and CORBA in our components, so as to make them usable also outside of our framework, and make them work in a distributed architecture either in a client/server structure, or in an Intranet (local components for full control) / Extranet (Business to Business integration) / Internet (third party support).

REFERENCES

[1] G. Booch, "Object-Oriented Analysis and Design with Applications," Cummings, 1991.

[2] G. Booch, J. Rumbaugh, and I. Jacobson, "The Unified Modeling Language User Guide," Second Edition, Addison-Wesley, 1999.

[3] P. Coad and E. Yourdon, "Object-Oriented Analysis," Prentice-Hall, 1989.

[4] A. Cockburn, "Surviving Object-Oriented Projects: A Manager's Guide," Addison-Wesley, 1997.

[5] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, "Object-Oriented Modelling and Design," Prentice-Hall 1991.

[6] I. Jacobson, M. Griss, P. Johnsson, "Software Reuse," Addison-Wesley 1997.

[7] Microsoft Corporation, "MSDN Library," DVD-ROM edition, 2000.

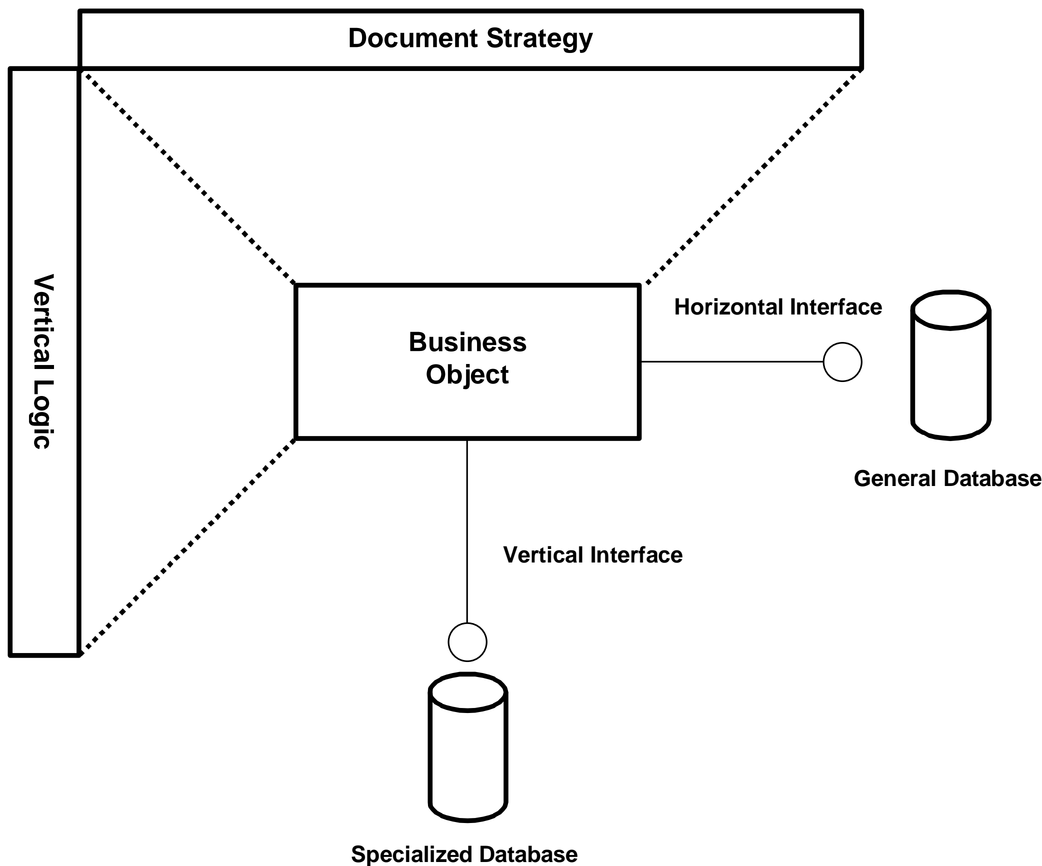

Figure 5: Business Objects architecture