A Multi-Layer Framework for Enterprise Application Development

Rodrigo Soares Manhães[1,2,3], Alexandre Gomes da Silva[1,2,4], Luiz Batista de Almeida[1,4], Rogério Atem de Carvalho[2]
1 Universidade Estadual do Norte Fluminense (UENF), Brazil
2 Centro Federal de Educação Tecnológica (CEFET-Campos), Brazil
3 Faculdade Salesiana Maria Auxiliadora (FSMA), Brazil
4 Universidade Federal Fluminense (UFF), Brazil
rmanhaes@cefetcampos.br, {alexangs, lbalmeida}@uenf.br,
ratem@cefetcampos.br

Abstract. Building Enterprise Information Systems (EIS) presents, by a structural point of view, a number of somehow repetitive tasks like programming behavior and structure for Graphical User Interfaces (GUI) for user input and data retrieving; coding verbose protocols to accessing application servers for merely input and retrieve data; coding numerous routines implementing database queries for answering several kinds of user transaction requirements, and others. For most applications, these tasks are numerous, repetitive, complex and error prone. Besides, they are very similar in most enterprise applications. This work proposes an object-oriented multi-layer framework for automating basic structural tasks for enterprise applications, freeing developers from time-consuming, low aggregated value tasks and allowing them to concentrate on the actual goal of software development: the implementation of user requirements. The framework is constructed over three conceptual layers – presentation layer, business layer and service layer – each one playing a well-defined role in the architecture.

1 Introduction

The current software development, in spite of the dissemination of object technology, did not achieve yet the goal of reducing effort on repetitive and similar development codification activities in several transactional systems, like

Please use the following format when citing this chapter:

Soares Manhães, R., Gomes da Silva, A., Batista de Almeida, L., Atem de Carvalho, R., 2006, in International Federation for Information Processing, Volume 205, Research and Practical Issues of Enterprise Information Systems, eds. Tjoa, A.M., Xu, L., Chaudhry, S., (Boston:Springer), pp.285-296.

data entry forms, simple queries, data access facilities and complex but repetitive operations for getting access to application servers.

In most of the current systems, all these activities are encoded *ad hoc*, in a low style of programming, not much reusable and inclined to mistakes. Additionally, the use of object-oriented (OO) programming many times only happens within "visual inheritance", in other words, the reuse of form layout templates and some basic GUI functionality. The data access classes normally are, also, encoded case by case, and they are difficult to be reused. Also, there are the particularities of different queries and the shortcomings caused by the "impedance" between OO programming paradigm and the relational paradigm.

The present proposal aims to present an object-oriented multi-layer framework for the creation of applications where the developer will focus in the problem's business rules to be solved and not in the creation of interface functionalities, database access code or formalities of the access to application servers.

Thus, the framework has reasonable amplitude, comprising: a) a MVC (Model-View-Controller) framework [1, 2] for Java desktop and several facilities and layout creators for the Swing API (Application Programming Interface); b) a set of classes that encapsulate application servers, hiding of the developer his complexities and formalities of access; c) a framework of DAO (Data Access Objects) [3] automation, so that basic accesses, of the type CRUD (Create, Retrieve, Update, Deletes), to the DBMS (Database Management System) are done in transparent and automated way.

1.1 Overview

The work described here has been implemented under the Java platform, more specifically the J2EE (Java 2 Enterprise Edition) distribution. The proposed framework intends to be extremely flexible and extensible to the use within third part products, through the development of plug-ins (there are several of them already implemented). This plug-ins take the form of adapter classes, according to the Adapter design pattern [4].

The framework offers a solution, which intends to be complete, for the presentation layer, being directed exclusively for the desktop. However, it is simple to extend it to give support, for example, to MVC mechanisms for the web, which can substitute or act concomitantly to the standard presentation in desktop.

The framework supports, at this moment, in its business layer, the use of application servers based on the JSR-19, Enterprise JavaBeans 2.0 [5]. The use of EJB 2.0/2.1 is considered widely complex and bureaucratic, and the framework automates almost all the complexity of its use, releasing developer's time for implementing business rules. The new version of the specification, 3.0 [6], however, simplifies a lot the use of EJB; support for it is expected to be provided soon.

At the persistence layer, the framework offers support to SQL-92 and to the object-relational persistence mechanism Hibernate. Specific plug-ins for Firebird and PostGreSQL DBMS are on the way.

2 Presentation Layer

In Java platform, browser-based web interfaces are well used as technology for information systems presentation. For doing that, there is an abundant offer of open-source frameworks, such as Apache Struts, MyFaces (open-source and certified JSR-127 (JavaServer Faces) implementation), Spring, WebWork, Jakarta Tapestry, VRaptor, JSenna, e-Gen and others. For a presentation layer based in desktop, however, the offer of open-source frameworks is quite reduced. Basically, we have knowledge of the MVC framework JForms, which, however, did not pass from version 0.0.1, release candidate.

Besides making possible a very richer interactivity with the user, since several interface functionalities are quite difficult – if not impossible – to be implemented in a browser, there are several reasons to develop desktop applications: visual handling of graphic elements – diagrams, graphics, images etc; distributed processing, in the cases in which it is desirable to have code running in the clients; local cache of information, to increase access performance, and others. Even traditional disadvantages, like the necessity of installing software locally in several client machines, have been significantly minimized by the use of technologies like Java Web Start.

The proposed model is an adaptation of the Model-View-Controller architectural pattern [2]. In this pattern, the View layer contains only the code for the assembly of the user interface, the Controller contains the logic for the treatment of events occurring in the system, and Model is the layer responsible for the treating and supplying data for the other application layers. The View and Controller lay on framework's Presentation Layer; Model works on framework's Business Layer.

Besides providing the construction of visions and controllers *ad hoc*, the functionalities offered by the presentation layer offer interfaces with almost functional specific characteristics, and the framework user will only extend some few hot spots.

2.1 Design Issues

The presentation layer was developed based on the Java Foundation Classes (JFC) toolkit, with emphasis in the set of functionalities known as Swing [7], the more powerful and popular graphic toolkit for desktop in Java. However, the great power provided by Swing has a high price, which is complexity, being criticized because of being exaggeratedly academic and carrying out in extreme way all the

requisites of a strictly object-oriented project [8]. However, we think this is a quality, since it gives the programmer the possibility to build virtually anything in the user interface, in a strictly object oriented way.

To avoid dealing directly with Swing API complexities, the framework offers classes that work as Façades [4], hiding complexity and supplying a lighter and dry API. Therefore, one of the main requisites of the presentation layer is to use the power and flexibility of the API and, at the same time, create facilities so that the user productivity is maximized.

2.2.1. Presentation Layer Internals

Presentation layer architecture is adapted from MVC, inspired in the standard J2EE View Helper [3]. Thus, there is, for each item of interface, a controller that assumes the function of helper in the foregoing standard, assuming the tasks of process data for visualization, storing intermediary data models, and working as a business data adapter. Therefore, in the presentation layer, the hot spots consist of only a controller/helper and a viewer, that will contain only the visualization components, which are essentially *ad hoc*, and all the processing and adaptation responsibility of business data and the communication with the business layer stay with the controller/helper. Besides that, controllers and viewers abstracts classes automate, by default, repetitive tasks like reading and writing component data, allowing very little codification for the hot spot implementation in most cases.

A visualization screen, however, has not only the customized viewer, but a series of other supportive visual components, like menus, toolbars, navigators, search bars and others, that have fixed or little customized appearance and behavior. Such components are considered frozen spots, though they could be eventually extended without any type of impediment related to couplings. The communication between several visual classes that compose the frozen structure of the presentation layer is implemented by the use of the Mediator design pattern [4], making possible a decoupled communication between the components and giving, even for the frozen spots, great flexibility and extensibility.

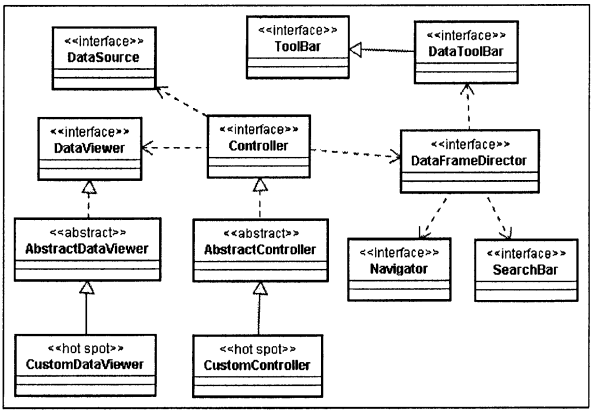


Fig. 1. Presentation layer schema

The Controller also has another responsibility in the framework architecture, which is to carry out the communication between the hot spots and the DataFrameDirector (the entity that does the mediation between the participant colleagues of the Mediator design pattern) that manages the standard components. This communication also is done in completely automated way, so that the developer of the hot spots will be able to ignore even the existence of the frozen spots. The described schema is shown at Figure 1.

3 Business Layer

The business layer is responsible for containing all application logic, in other words, it is the layer that implements the rules concerning the application domain, controlling the stream of work and maintaining the consistence of user sessions state.

The separation of business layer from other layers brings several benefits for the application, for instance, the possibility of diverse systems using different graphical interfaces (desktop and web) share the same business logic resident on a centralized or distributed repository, like an application server. But it is not only the detachment of implementation of the business layer from the graphic interface that became this kind of solution so popular and efficient in the modern object-oriented software development. The clear separation of different concerns obtained from software partition makes technical issues concerning one layer be transparent to others. For instance, in the business layer, problems as data security and concurrent transaction management must be considered by solution developers, but ignored (or, at least, minimized) by user interface designers.

3.1. Design Issues

The primary approach of the framework is oriented to reduce the complexity of J2EE architecture's distributed components development, separating the business logic implementation itself, isolating it from technological low level details.

The function of the framework is to automate the codification of the underlying server application management and the EJB component that will be installed in the server. For this goal, the framework uses a set of classes that hide from the developer all details concerning the access, from client, to EJB components and hiding the EJB classes that connect with data services to provide persistence capabilities, by example.

Repetitive tasks performed for accessing an EJB from a desktop or web application, like the adjust of JNDI (Java Naming and Directory Interface) API for searching and obtaining the home interface of EJBs running in the application server, are automated by the framework, through the use of a simple XML configuration file. An immediate consequence is to transform JNDI into a transparent service for the user. Some design patterns are used for achieve these facilities. One of the patterns used is *EJBHomeFactory* [9], whose objective is to provide a factory that will search and obtain the home object (object responsible by the EJB component life cycle management in the container, e.g., its creation or its destruction) only in the first request of the client, storing and becoming it automatically available for future requisitions. The framework makes the factory transparent for the developer, being enough that he/she informs the name of home interface to the factory. The framework will use Java's introspection mechanism for invoking the search service and creating the home object in application server, taking off an extremely repetitive and error-inclining work from the responsibility of the application developer. The class diagram that represents the described structure is shown at Figure 2.

The framework also contemplates the automation of the relationship between the EJBs and the data access service resident in the integration layer. In a multi-layer application, the business layer interacts, directly or not, with the persistence services for storing and retrieving data. The architecture of framework intended to automate CRUD operations, because these operations are recurrent in all applications that need to persist data, becoming primordial to hide from the developer the task to define and handle the linking between the two layers for these kinds of operations. For achieving this objective, *Session Façade* [3] design pattern was implemented in the framework to interact with another J2EE design pattern, *Data Access Object* [3]. The DAO pattern belongs to the integration layer and is responsible for controlling the framework's persistence mechanism. The only task reserved to developer is to inform to *façade* what DAO will be instantiated and, according to its polymorphic behavior, all CRUD operations provided by DAO will be immediately available to the business layer and, consequently, to the presentation layer. Other benefit offered by automation of the

façades is the encapsulation of EJB architecture technical details, e.g., the handling of exceptions referring to network communication, reducing once more the developer responsibilities with non-trivial technical issues related to the use of J2EE technology.

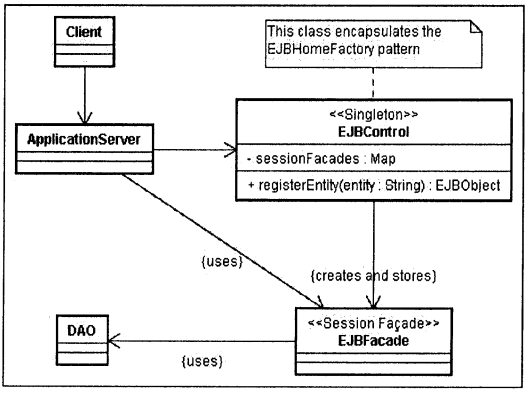


Fig. 2. EJB Interfacing Scheme Class Diagram

3.1.1. Data Transport and Decoupling

All data transport between layers, in this framework, is performed by objects that implement the *Value Object*, also known as *Transfer Object* [3], that are objects containing the same data of business objects, but only acessor members. Here, the main objective of the use of *ValueObjects* is to remove the direct dependency of presentation and persistence layers related to the domain application model.

4 Persistence Layer

The persistence layer is a highlighting point in the architecture of any application with needs to store and retrieve data, even if it is contained in DBMS, XML files, serialized object files, and LDAP repositories. For each data access mechanism, the variety of APIs that support it becomes more and more abundant and diversified. In the case of relational DBMS, there are many solutions, like J2EE CMP entity beans, the Hibernate framework for object-relational mapping, or even the direct use of JDBC or the underlying DBMS' API.

It's very natural that in this so complex and sensible piece of the application, there is an evolution of APIs for increasing the productivity and the facility of implementing the permanent storing devices. Besides, handling legacy data

schema concurrently with new data schema is commonplace. Thus, it is extremely important the existence of a very weak coupling between the persistence mechanisms and the rest of the application. In this way, any future changes, even caused by the natural evolution of APIs or by the replacing of a solution, have very low (or, preferentially, none) impact in the rest of the application.

4.1. Design Issues

Aiming for the objectives of low coupling and high cohesion in the persistence layer, ensuring the maintainability and the portability of the remaining architecture and preserving the freedom of the developer in adopting various data persistence solutions, the framework adopts a sophisticated interface schema. Many design patterns are widely implemented in this layer to ensure the necessary flexibility, facilitating the operations on this admittedly heterogeneous system structure, beyond centering the access to its functionalities.

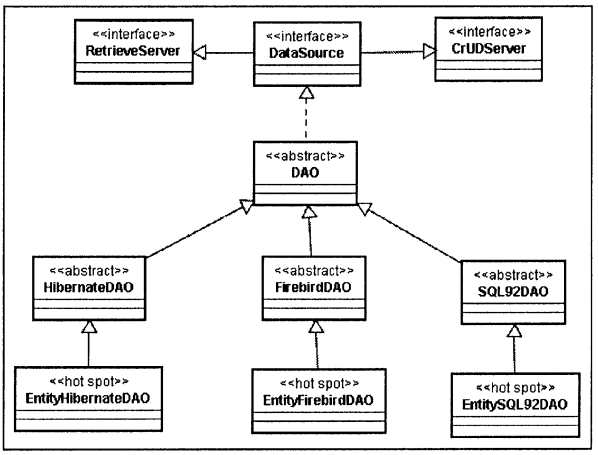


Fig. 3. Framework's data access architecture

On top of data access architecture's hierarchy, we have CrUDServer and RetrieveServer interfaces. Both interfaces comprise the four basic functions of the persistence layer, known as CRUD – create, retrieve, update and delete data. The CrUD server interface is responsible to provide operations for three of four persistence functions – including, updating and deleting. The CrUDServer interface has a lowercase "r" in its name for expliciting the retrieve operations are not defined in this interface. All three operations take as parameter a Value Object, representing the persistent entities of the system. The RetrieveServer

interface provides operations that comprise almost all searching possibilities that an application can need, through the use of auxiliary classes representing the full set of relational algebra operations (select, project, union, intersection, difference and Cartesian product) [10], but under a higher and object-oriented abstraction level. This makes possible that all queries be requested, at presentation level, only in terms of objects and its attributes. Consequently, all queries are generated dynamically from the relational algebra encapsulator objects, with no need for creating *ad hoc* querying methods on DAO classes, except for very special cases. The schema containing the described interfaces is shown at Figure 3.

The DataSource interface extends CrUDServer and RetrieveServer, being a starting point for the components that will handle all persistent operations present in these two interfaces. These interfaces are fundamental for the framework, due to their presence on the other layers of the architecture, do not restricted to persistence layer. This happens because the CRUD operations are requested by the application user start from the presentation layer and pass through the layers until achieve the persistence layer for executing the required operation. To reach this goal, it was adopted the *Chain of Responsibility* design pattern [4], whose advantage is to provide a low coupling level between an object that sends a request and the receiver one. In the particular case, the sender objects are resident on presentation level and the receiver objects belong to persistence level. From the graphical interface to the persistence services, the request goes through all layers.

4.1.1. Data Access Objects' (DAO) role

Among the DAO implementation possibilities, the framework adopted a factory strategy for Data Access Objects, whose implementation is based on the *Abstract Factory* design pattern [4]. This pattern is convenient because its objective is to deal with a hierarchy of classes composed by a set of subclasses with a parent class in common. In that way, the framework automates and speeds up the implementation of the DAOs, defining the main methods that will have to be considered by the developer for the correct communication between the actual layers in the whole architecture. The factory responsible for creating, effectively, the correct DAO in the application business layer, according to the design pattern Abstract Factory, will have to be implemented by the developer, since each application will define, in a unique fashion, its set of applicable DAOs.

5 Case Study

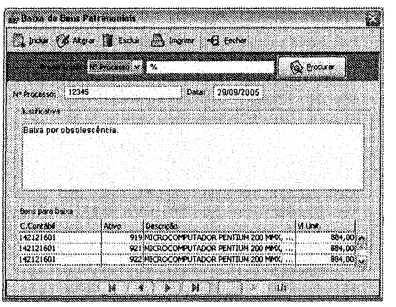
The framework has being developed on context of an enterprise application for a public university in Brazil. All features explained already were implemented and they are working at production. Figure 4 shows an example viewer and the

controller code manipulating all frozen spots and hot spots, including the data obtained from a database.

In the presentation layer, for each CRUD screen, the developer will create a panel containing data components (through programming, tools or IDEs). The framework provides a set of programmed components to perform several kinds of validation, like dates and personal document numbers. These components can be easily extended or customized according to the developer needs.

The controller, that also will be unique for each CRUD screen, will contain a small quantity of code to register the data components that will be automatically managed by the framework. These components, after being registered, have their behavior determined by the framework in a transparent way, thus the control of the edition mode, reading, color, among others, of the components in the diverse screen states (e.g. data inclusion, edition or browsing) is exclusive responsibility of the framework, do not having need of application developer interference. The controller class can be defined programmatic, or through a simple XML file, according to the programmer convenience.

Another controller responsibility is to inform the class of the persistent objects that will be used in the process. From this information, the framework makes possible the interaction of the presentation layer with the business and persistence layers, in order to automate the CRUD operations. With that, the screen does not be concerned of how data will be obtained or treated in the application, since the framework standardizes the procedures of obtaining, manipulating and recording these data in the persistence layer.




```
public class BaixaBemController extends
    AbstractController <BaixaBemViewer> {
    public BaixaBemController(BaixaBemViewer viewer) {


        super(viewer);
       this.addModel(this.getViewer().getModel("nrDoc",
          ModelFamily.DOCUMENT), "Nº Processo");
       this.addModel(this.getViewer().getModel("data",
          ModelFamily.DOCUMENT), "Data");
       this.addModel(this.getViewer().getModel("just",
          ModelFamily.DOCUMENT), "Justificativa");
       this.addModel(this.getViewer().getModel("bens",
          ModelFamily.TABLE), "Bens para baixa");
    }
    protected Class<? extends ValueObject> getVOClass()
    {
        return BemPatrimonialVO.class;
    }
    public String getFrameTitle() {
        return "Baixa de Bem Patrimonial";
    }
}
```

Fig. 4. A viewer and its controller

6 Conclusions

The strong interlayer decoupling guaranteed by the framework, beyond the evident and well-known advantages of low coupling [11], avoids dependency of providers, when it allows the different layers of framework are used in independent way, being able to work together with another supplier products. For instance, the connector architecture allows the presentation layer to have its role played for MVC web frameworks, in place of the default desktop presentation of framework.

Moreover, the automation of the tasks related to data manipulation is a very valuable factor for productivity increase and software maintainability, releasing developer time for modeling and implementation of the user business requirements.

References

1. G. Krasner and S. Pope, A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, *Journal of Object-Oriented Programming* **1**(3), (1988).
2. F. Buschmann, *Pattern-Oriented Software Architecture: Volume 1 – A System of Patterns* (John Wiley & Sons, New York, 1996).
3. D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies*, 2nd edition (Prentice-Hall, New York, 2003).
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (New York: Addison-Wesley, 1995).
5. L. DeMichiel (leader), *JSR 19 – Enterprise JavaBeans 2.0*, (2002); http://www.jcp.org/en/jsr/detail?id=19.
6. L. DeMichiel and M. Keith (leaders), *JSR 220 – Enterprise JavaBeans 3.0*. (2005); http://www.jcp.org/en/jsr/detail?id=220.
7. M. Robinson, and P. Vorobiev, *Swing*, 2nd edition (Manning, Greenwich, 2003).
8. J. Marinacci, *Swing Has Failed. What Can We Do?* (2003); http://weblogs.java.net/blog/joshy/archive/2003/11/ swing_has_faile.html.

9. F. Marinescu, *EJB Design Patterns*: *Advanced Patterns, Processes and Idioms* (John Wiley and Sons, New York, 2002).
10. E. Codd, Relational Completeness of Data Base Sublanguages, in: R. Rustin, Data *Base Systems* (Prentice-Hall, New York, 1972).
11. I. Sommerville, *Software Engineering* (Addison-Wesley, New York, 1995).