Developing Enterprise Applications with Support to Dynamic Unanticipated Evolution

Hyggo O. de Almeida, Marcos F. Pereira, Márcio de M. Ribeiro, Angelo Perkusich, Emerson Loureiro and Evandro Costa

Abstract— **This paper presents a component based framework for developing enterprise applications with support to dynamic unanticipated evolution. The framework is based on the COMPOR Component Model Specification, which provides mechanisms to manage unpredicted evolution even at runtime. We describe the framework design that is based on design patterns and aspect-oriented concepts. Finally, we present a example application of the framework in the context of electronic commerce.**

Index Terms— **Unanticipated Software Evolution, Enterprise Applications, Component-Based Development**

I. Introduction

Enterprise information systems are applications for handling company-wide information and delivering services to a wide range of users. Such systems must be: secure, to protect users and the enterprise; scalable, to ensure that users simultaneously take advantage of various services; and reliable, to ensure the consistency of the transactions processing.

Besides these features, enterprise applications change frequently. Considering the complexity of these applications, requirement changes cause a great impact on the system architecture, design and code. This impact is even more relevant when such changes are not predicted at design time. Unanticipated changes have been pointed out as the main reason of problems related to software evolution activities [1]. In the case of enterprise applications that cannot be interrupted for financial or safety reasons, it becomes even more difficult to manage unanticipated evolution at runtime.

J2EE [2] and .NET [3] are well known platforms for developing and deploying enterprise applications. Developers using such platforms save time by not looking at a diverse range of products and services, since they are already provided by those platforms. Such services include security, persistence, distribution, load balancing, and transaction management, among others. Nevertheless, J2EE and .NET do not support adequately dynamic unanticipated software evolution. This occurs due to the high coupling among components, which makes difficult to implement unpredicted changes on the fly.

To deal with this problem, in [4] is proposed a component model to develop software supporting dynamic unanticipated evolution named COMPOR Component Model Specification (CMS). Such a model allows changing any part of the software, by removing and/or adding components, even at runtime. It is also proposed a Java implementation of CMS called Java Component Framework (JCF), which is used to develop Java applications supporting dynamic unanticipated evolution.

However, the JCF framework do not provide support for developing enterprise applications. When developing software with JCF, developers have to implement all features related to enterprise applications, such as security, distribution and transactions management from scratch. In this paper we introduce an extension of JCF framework for developing enterprise applications with support to dynamic unanticipated evolution. More specifically, we describe how to extend JCF design to implement distribution, security and transaction management by using design patterns and aspect-oriented programming.

The remainder of this paper is organized as follows. In Section II, we present the extension for enterprise applications. Section III describes a example application of the framework. Section IV discusses some related works. Finally, in Section V, we present the final remarks.

II. Support for Enterprise Application

In this section we present the extension of JCF to develop enterprise applications. More specifically, we describe how to extend JCF to provide support for security, transaction management and distribution features.

A. Security

According to CMS, an alias is used to uniquely identify services and events with the same name for different components. However, such a strategy introduces a security problem into the model. For example, it is possible to interpose a provider *X* between another provider *Y* and its clients in order to intercept the client requests towards *Y*. This may represent an intrusive way to make something undesirable in the system, since the interposed provider *X* may be seen as an intruder.

As this security issue is not tackled by the component model, the JCF must provide means for dealing with security policies for the interaction and deployment models. Such policies must then be satisfied when some service is requested or an event is announced as well as a component is inserted into or removed from a container. This security infrastructure, shown in Figure 1, was developed using aspect oriented

The authors are with the Embedded System and Pervasive Computing Laboratory, Department of Electrical Engineering, Federal University of Campina Grande, C.P. 10105 - 58109-970 - Campina Grande - PB - Brazil, emails: hyggo@dsc.ufcg.edu.br, marcos@embedded.ufcg.edu.br, mmr3@cin.ufpe.br, perkusic@dee.ufcg.edu.br, evandro@ic.ufal.br

programming, with AspectJ [5]. Aspects have allowed to hide the complexity of the security mechanism from the developer as well as to simplify the development of systems without security requirements. The security mechanism illustrated in Figure 1 is explained as follows.

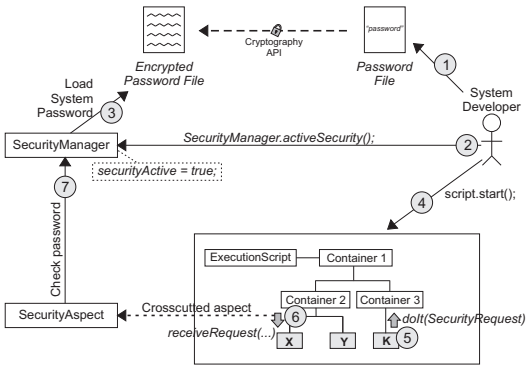

Fig. 1. Aspect oriented security architecture.

1) The application developer creates a ".security" file containing the password for accessing the system as well as the service access policies. Then, uses the Java cryptography API to encrypt the file.
2) When developing the application, the security mechanism should be activated calling the `activeSecurity()` method of the `SecurityManager` singleton class. This operation defines that all service invocations, event announcements and component additions must be verified.
3) The `SecurityManager` retrieves the password and the policy information and stores them in memory.
4) After starting the root container, all of its components are also started and the application runs by means of a sequence of service invocations and event announcements.
5) A component invokes a service. With the security activated, the service requester component must forward a `SecurityServiceRequest` instance as parameter, containing the system password.
6) The component receives the request via the `receive Request` method, then the `SecurityAspect` aspect intercepts the method invocation and asks the `Security Manager` to verify the request password.
7) `SecurityManager` verifies the request password and allows the service execution. Otherwise, a `ComporSecurity Exception` is thrown.

B. Transaction

Figure 2 illustrates the component *K* requiring the execution of two services. The first (*withdraw*) is implemented by the component *X*, whereas the second (*deposit*) is implemented by the component *Y*. Because it represents a money transferring, such operation must be atomic (or indivisible), which means that the money either moves between the two accounts or it stays in the first account.

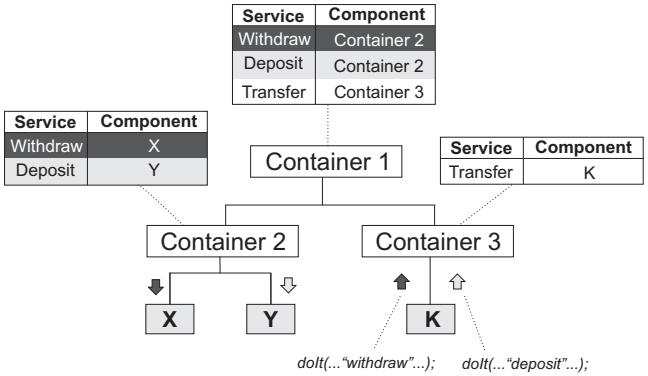

Fig. 2. Necessity of atomic operations.

In order to handle with atomic situations, a transaction mechanism is available. Such mechanism implements the two-phase commit protocol, a popular protocol used to guarantee consensus between the participating members of a transaction [6].

In the same way of the security mechanism, the Aspect-Oriented approach was used, allowing us to separate the transaction concern as well as to develop systems without it by simply removing the aspect responsible for implementing the mechanism. Therefore, the simplicity of the CMS model was maintained, since it does not depend on the transaction mechanism.

The two-phase commit protocol defines a coordinator that is responsible for governing the outcome of the transaction. In the first phase of the protocol, the participants (in our case, components) must invoke their init service. According to the all participants answers, in the second phase the coordinator decides whether it will commit or rollback the transaction by sending a message with its decision to all participants.

According to the CMS model, when clients invoke services, they must use instances of the *ServiceRequest* class. However, if clients have to execute transactional services, they must use instances of the *TransactionServiceRequest* class instead. Notice that a such class extends *ServiceRequest*.

Each CMS component must extend the *FunctionalComponent* class, which has an important method named *receiveRequest* [4]. Since *receiveRequest* is called by the framework before the execution of services, the aspect responsible for the transaction mechanism verifies the instance of the service request. If the request is an instance of the *ServiceRequest* class, the service is executed normally. Otherwise, a transaction is started.

Notice that the verification about which service (init, commit, or rollback) will be executed is weaved by the aspect in the *receiveRequest* method. Hence, the implementation of the protocol is guaranteed by this verification. In addition, the consistence of information through atomic operations is guaranteed as well.

Aiming at completing the ACID properties, the mechanism also provide isolation of transactions through synchronization

of threads. Besides, in order to guarantee the consistence of data in case of hardware crashes, each transaction is logged. When the system comes back, the aspect crosscuts its initialization and recovers the transactions automatically through the log file reading.

C. Distribution

Distribution is a desirable feature for enterprise applications, since it might provide performance increasing, economies of scale, reliability (if carefully designed), and resource sharing (through the use of a computer network).

When considering distributed software, each module of the software might reside in different computers in the network. The communication among those modules is based on sending messages to each other. In the component based development context, these modules consist of components of software.

Similarly to the J2EE and CORBA, JCF containers play a fundamental role in the distribution implementation as well. In this context, each JCF container extension is responsible for sending requests and event announcements to their distributed components children. Figure 3 illustrates the distribution mechanism, which relies on the Decorator [7] and Proxy [7] design patterns. Notice that it is an extension of the CMS model. This way, the simplicity of the model remains, since it does not depend on the referred mechanism.

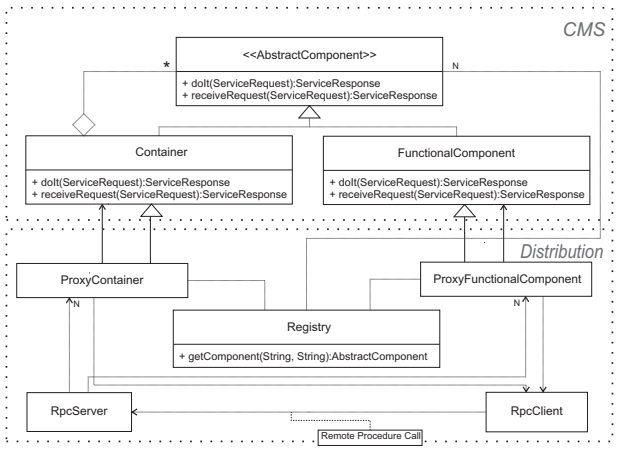

Fig. 3. Distribution architecture.

The architecture of the distribution mechanism is explained as follows. Containers have children which consist of representative entities (relying in Proxy [7] implementation). Notice that such entities point out to the remote functional component and the parent of the remote components is also a representative entity. Nevertheless, it points out to the remote container instead.

In order to get started with a distributed application, as illustrated in Figure 4, the application developer must deploy the desired part of the hierarchy into each participant host. For each host, such a developer must execute the following steps to configure the distribution:

1) In the host *192.168.10.6*, he must add an instance of `ProxyFunctionalComponent` retrieved from the host *192.168.10.1* (by a remote procedure call), such instance is a proxy to the real component which resides in the host *192.168.10.1*. Notice that the real component is child of the container localized in the host *192.168.10.6*.
2) In the host *192.168.10.6* the real component is added as a child of the `ProxyCont1`, which is a `ProxyContainer` instance retrieved from the host *192.168.10.6* (also a remote procedure call).

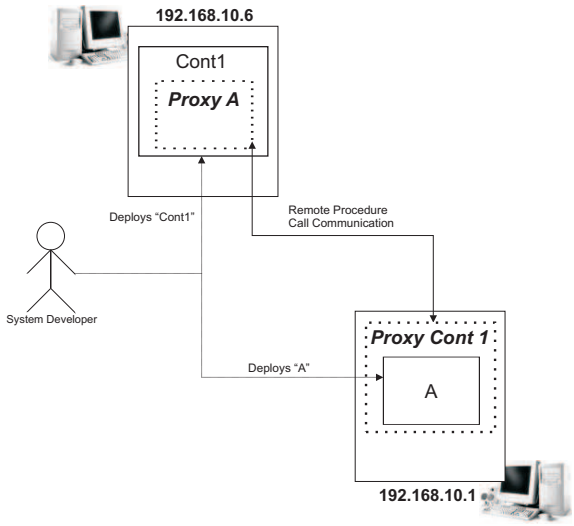

Fig. 4. Deploy and interaction of distribution mechanism.

Each instance of `ProxyFunctionalComponent` and also of the `ProxyContainer` class automatically register itself into an instance of the `Register` class. Such operation is necessary because the `Register` class is used to find proxy instances from other hosts.

In the distributed hierarchy, the component model exchanges service requests and event announcements between two computers in a transparent way. In order to implement the network communication, the JCF distribution mechanism relies on the Apache [8] implementation of the XML-RPC specification [9].

III. Example Application

The e-commerce application is a proof concept of our enterprise mechanisms. Such application provides a list of products to be purchased. In order to buy items, the user might select them. After confirming the operation, the system shows the total price of the selected items to be bought (Figure 5). When the user decides to buy something, the system invokes the *buy* service implemented in an CMS based hierarchy. This service withdraws the needed money from the user's account of a bank and deposits it into the system's account of another bank. By the presence of the transaction mechanism (described in Section II-B), the application executes the *withdraw* and *deposit* services atomically. In addition, our mechanism guarantees hardware crashes by logging the operations done by the system.

Fig. 5. Purchasing a list of items.

As illustrated in Figure 6, for security reasons, each bank is responsible for developing and maintaining both *withdraw* and *deposit* operations. The communication between the banks and the e-commerce application occurs through the distribution described in Section II-C.

This way, the application must trust in each bank components. Beyond the components, each bank must maintain a encrypted password file as illustrated in Figure 6. This password is used by the security mechanism, which is demonstrated in Section II-A.

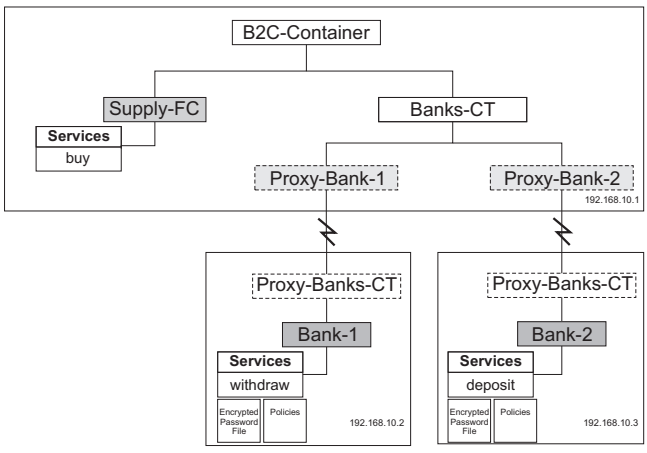

Fig. 6. E-Commerce component hierarchy.

IV. Related Works

There are two main component models used for developing enterprise applications currently, Enterprise JavaBeans (EJB) in the J2EE platform and Microsoft's .NET. The main reason is the support to the following required services in software development for enterprise applications: security, distribution, transaction, web server, etc. These features are essential and must be present into almost all applications, mainly enterprise ones.

Moreover, enterprises demand on-the-fly changes of running applications, because the downtime of their software directly causes big looses. This demand includes a new feature: The dynamic unanticipated software evolution. However neither, EJB nor .NET provide native support for this feature. This feature might be implemented in these component models, but it is a difficult task because the design of them has a high coupling among components.

There are some works which could be used to add dynamic unanticipated evolution support to these models. One of them have created a new class loader type for use in J2EE platform. This class loader might minimize the barrier between two or more class loaders [10]. Another work [11] proposes a new way to load DLL libraries in .NET.

V. Final Remarks

In this paper we presented a component based framework for developing enterprise applications supporting dynamic unanticipated evolution. Such a framework is an extension of the COMPOR Java Component Framework, which implements a component model called CMS that promotes software evolution even at runtime.

We described the framework design that is based on design patterns and aspect-oriented programming. The framework includes support to security, distribution and transaction management, but still maintaining the CMS simplicity. We describe also an e-commerce case study of the application of our approach.

As future work, we plan to develop other features of enterprise applications as extensions of JCF, such as load balancing, persistence, logging, and integration to legacy systems. Also, we are working on applying the proposed framework to develop a web-based e-commerce application.

References

[1] G. Kniesel, J. Noppen, T. Mens, and J. Buckley, "1st Int. Workshop on Unanticipated Software Evolution," in *ECOOP Workshop Reader*, ser. LNCS, vol. 2548. Springer Verlag, 2002.

[2] SUN, "Sun developer network (sdn)," July 2007, http://java.sun.com/javaee/.

[3] Microsoft, ".net framework developer center," August 2007, http://msdn.microsoft.com/netframework/.

[4] H. Almeida, G. Ferreira, E. Loureiro, A. Perkusich, and E. Costa, "A Component Model to Support Dynamic Unanticipated Software Evolution," in *Proceedings of International Conference on Software Engineering and Knowledge Engineering*, vol. 18, San Francisco, USA, 2006, pp. 262–267.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An Overview of AspectJ," in *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 2001, pp. 327–353.

[6] M. Little, J. Maron, and G. Pavlik, *Java Transaction Processing*, 1st ed. Prentice Hall PTR, 2004.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

[8] Apache, "Apache software foundation," August 2007, http://www.apache.org/.

[9] XML-RPC, "Xml-rpc home page," August 2007, http://www.xmlrpc.com/.

[10] Y. Sato and S. Chiba, "Negligent class loaders for software evolution." in *RAM-SE*, 2004, pp. 53–58.

[11] S. Eisenbach, V. Jurisic, and C. Sadler, "Managing the evolution of .net programs." [Online]. Available: citeseer.ist.psu.edu/728246.html