DECOUPLING MVC: J2EE DESIGN PATTERN INTEGRATION

Francisco Maciá-Pérez, Virgilio Gilart-Iglesias, Diego Marcos-Jorquera, Juan Manuel García-Chamizo
Departamento de Tecnología Informática y Computación, Universidad de Alicante , AP. 99, 03080, Alicante, Spain

Antonio Hernández-Sáez
Servicio de Informática de la Escuela Politécnica Superior, Universidad de Alicante, AP. 99, 03080, Alicante, Spain

Keywords: Enterprise platforms, design patterns, J2EE, frameworks

Abstract: In this paper we propose a model based on the Model-View-Controller design paradigm and built over the integration of open source frameworks, which are widely supported by the software architect community. The main contribution of this model lies in that it provides a true decoupling of the MVC paradigm's model, view and controller elements. This approach eases the horizontal development and maintenance of large-scale distributed network applications. In order to concretize our model, we have based our prototype application in the following three frameworks. First, the Struts framework in which the controller element resides. Second, the Cocoon framework which serves as the basis for the view. And, finally, the J2EE business components that constitute the model. This led us to integrate these three frameworks so as to decouple the referred MVC elements, through the use of the Cocoon-Plugin (as the View-Controller tie) and Struts-EJB (which links the Model and the Controller elements).

1 INTRODUCTION

The Internet has changed the way in which business is understood. Nowadays, organizations can find in the Internet's environment new models to compete which were previously targeted only by the largest corporations, due to its high operation costs (Harmon, 2001). These transformations imply new requirements that are not satisfied by traditional software models, therefore forcing to adopt new strategies so as to adapt the business and software processes – this is referred to as process reengineering.

In most cases, organizations resistance to take over these changes is high and sometimes traumatic, due to various facts: the effort necessary to give up the former business culture, the complexity of integrating inherited systems which hold critical enterprise information, the changes needed in the established infrastructures and the learning process required for the organization's personnel. In spite of these difficulties, new enterprises that rapidly adapt to the new technologies are arising, forcing older companies to evolve towards these environments (Harmon, 2001).

The traditional architectures do not provide a global solution to the new business models requirements, since many of these were not in their initial design: standards that support integration between applications and different devices (Hansmann, 2003), scalability to allow applications growing as the business grows (Weaver, 2004), flexibility towards new technologies and ubiquitous computing systems (Hansmann, 2003), security in not reliable environments prone to assaults (Sing, 2002), and portability over different systems (Cade, 2002). To fill in this architectural gap, a new generation of software platforms based on components over distributed (n-tier) architectures has been developed, in order to provide a complete solution allowing organizations to approach the new business models and to take advantage of the technological environment introduced by the Internet (Harmon, 2001).

Among these platforms there is J2EE, which has acquired, during the last years, a relevant role in software development based on the new business models (Sing, 2002). Its success probably resides in its specification, which is based on an open process participated by prestigious companies in the

Maciá-Pérez F., Gilart-Iglesias V., Marcos-Jorquera D., Manuel García-Chamizo J. and Hernández-Sáez A. (2005).
DECOUPLING MVC: J2EE DESIGN PATTERN INTEGRATION.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 280-287
DOI: 10.5220/0002524102800287
Copyright © SciTePress

technological sector that have contributed with their particular point of view about the new models needs (Allamaraju, 2002). This process has given birth to an infrastructure that supports the requirements demanded by enterprise applications in the new environments – diverse information needs, complex economic processes, diversity of applications, rapid developments that produce lasting designs, reliability, availability and security – (Weaver, 2004). For most organizations, creating such an infrastructure by their own means would be unfeasible due to its high costs in effort, time and budget. Besides, in case of achieving its creation, this custom infrastructure would probably lack J2EE's capabilities for integration with other distributed computing models (Weaver, 2004).

J2EE defines a programming model and supplies a set of associated services that provide the necessary elements in order to develop enterprise applications: transactions, security, administration, standard communication protocols, internationalization, scalability, integration, availability, maintenance capabilities and flexibility (Cade, 2002).

Nevertheless, while the foundations of the platform are relatively easy to describe and understand, applying them to outline an architecture for the design of distributed applications is not a trivial problem: it requires a deep understanding of the platform and a careful decision-making process.

Thus, having the platform's technical knowledge – about components, services and communications – is not enough in order to design good quality applications (Johnson, 2003). We also need to know when to use a particular solution for certain problem, and the reasons that motivate this choice; this can only be achieved through experience, which necessarily requires time. There are also solutions and advices – patterns, best practices, bad practices – based on third party developers that share their experience, pointing when and how to use each technology (Berry, 2002).

In this article we propose a framework that provides a true decoupling of the MVC paradigm's model, view and controller elements. This approach eases the horizontal development and maintenance for this type of applications, based on the J2EE platform. To begin with, we will introduce the current works related to the proposed model. Secondly, we will describe the research tasks performed in order to obtain our framework's general model. In the next stage, we will explain the model's integration and implementation process. Finally, we will present the corresponding conclusions and future research lines.

2 BACKGROUND

In B2C (business to customer) J2EE applications, clients have access to the business logic through the World Wide Web by means of the Web tier, which manages the communication between the Web client and the mentioned business logic (Sing, 2002).

For this type of applications, it is recommended to use the Model-View-Controller (MVC) pattern (Stelting, 2002). This pattern divides applications in three parts, decoupling the specific responsibilities for each one: the model represents the business information and defines the business rules or operations that manage the access to this information, and its modification; the view takes charge of the information presentation and allows the interaction between the user and the model; finally, the controller manages the requests in order to update the model (Moczar, 2003). MVC is therefore a generic design pattern oriented to systems architecture, which is widely spread in object-oriented programming. It provides ease of maintenance, component reutilization and adaptation capabilities (Stelting, 2002). It is also one of SUN's main recommendations for designs using the J2EE platform (Johnson, 2003).

In (Sing, 2002), SUN defined a model (Model 2) that establishes the design guides for J2EE components on the MVC.

SUN introduces two recommendations considered to be best practices:

- In first place, the use of frameworks in order to develop applications based on the proposed models. *A framework is a generic architecture that provides the basis for applications development.* It is an implementation of a set of patterns by means of a particular technology.
- In second place, SUN recommends developers to use existing frameworks instead of designing and building one of their own, since developing a framework incurs a large time cost, and it might take several years to reach the required technological maturity (Sing, 2002). Frameworks' main advantage consists in providing an infrastructure and a set of functionalities to the development team, so as to let them concentrate in the application's implementation, abstracting them from the design aspects. A good framework should incorporate the following characteristics:
  - The precise degree of flexibility
  - The support of a wide, solid community.
  - It should be easily learned
  - It should contribute with power and robustness to the applications developed on its base.

There are different implementations (frameworks) of Model 2, each one having its own

degree of recognition in the Java community. Among these, the most representative are Struts, Maverick, WebWork, Spring, WAF and JSF. In SUN's Blueprints, it is proposed a model for Web applications which is sustained on the Model 2. This paradigm describes the patterns and best practices that should be used in the design of such applications.

Besides, in order to solve model 2's limitations, there has been an evolution towards Model 2X described in (Mercay, 2002). The last model replaces JSP pages with XML documents and XSLT transformation sheets as the technology used for view presentation, allowing developers to truly separate the business logic's results from the presentation code.

Following the same line, we reach the scenario proposed in (Giang, 2003), where the business model resides in an EJB container, therefore achieving a complete decoupling of the three parts that form the MVC paradigm.

In the next point we describe the proposed model for the development of J2EE enterprise applications, based on the MVC paradigm. Unlike the MVC, our goal is not to obtain a generic model. Instead, our model points where to place the J2EE components in a typical scenario of Web applications that interact with the end user – similarly to Model 2's proceeding –, while establishing a design based on patterns and best practices so as to ease the development in real production environments. In contrast to the mentioned models, our model seeks for a complete independence of the view from the rest of the MVC components. In order to achieve this independence, our model is based on standards (XML and XSLT), similarly to Model 2X but in a more generic way, and focusing on providing a complete solution to integrate the controller in a distributed model, thus improving the scenario proposed in (Giang, 2003).

3 MODEL

The primary target of the proposed model is to simplify the development of large applications based on the J2EE platform (Gilart-Iglesias, 2005), thus providing a well structured architectural design, which allows for a complete decoupling of the system's main elements and synthesizes existing models, patterns and frameworks in the best way.

In this model, the controller serves as the application's entry point. It is implemented using only two patterns: the Intercepting Filter and the *Service To Worker*.

The *Intercepting Filter* is used in our model to implement the requests pre-processor; this system initially manages the entry requests from clients in the presentation layer. There are different types of requests, each one needing a particular processing scheme. Therefore, when a request arrives to the application, it should pass through a set of verifications before reaching the main processing phase – called the Front Controller –: authentication, session validation, client IP address checking, request authorization, data codification, auditory or browser type used (Alur, 2001). The Intercepting Filter pattern is a flexible and highly decoupled way to intercept a request, applying a set of filters, thus rejecting or allowing the request to arrive to the initial process (Berry, 2002).

This initial process plays the controller's role: it analyzes each request to identify the operation to perform, thus invoking the business logic associated to each particular request and controlling the flow to the following view (Sing, 2002). In the proposed model (see fig 1), our controller is designed following the Service To Worker pattern (see fig 2), which combines a set of smaller patterns that provide a complete and flexible solution to fulfil the requirements for an MVC controller while allowing the separation between actions – the model –, the view and the controller (Crawford, 2003).

The Front Controller pattern (Alur, 2001) describes a central point that manages the requests. In order to reach a greater flexibility and independence between the view and the model, the Front Controller assumes only the request analysis task, delegating in the Request Dispatcher the selection of the view and the action to perform. After the analysis phase, the Request Dispatcher will be in charge to select the command that encapsulates the operation to perform. Once this command has generated the result, the Request Dispatcher will select the next view to be shown to the user (Crawford, 2003). Delegating these tasks in the RequestDispatcher gives our model a greater flexibility since we can introduce new views or models in the scenario by altering the component's behaviour.

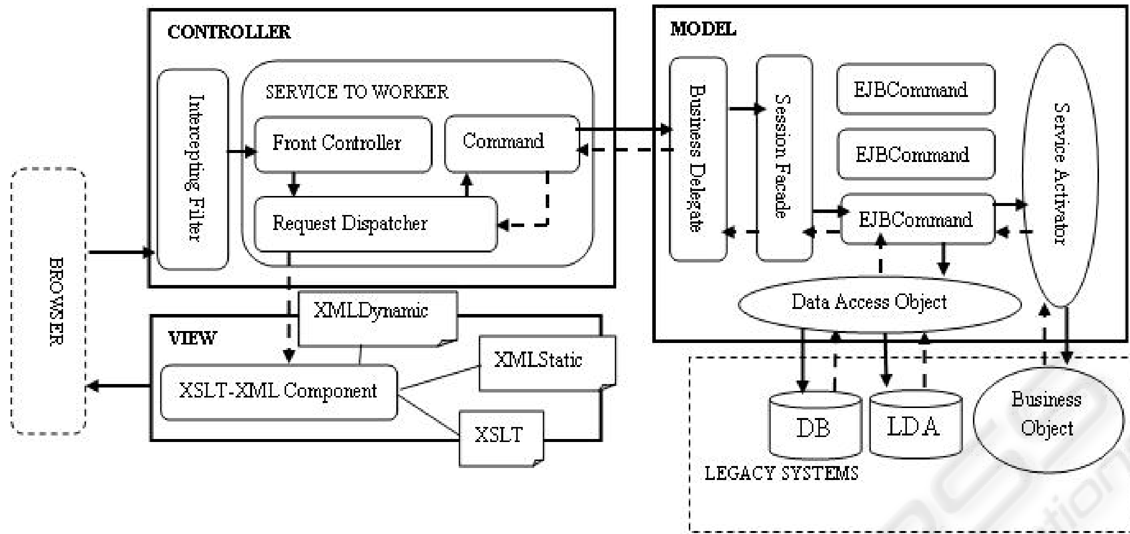

Figure 1: Model architecture

The Command pattern represents each request by means of an object, therefore providing a very simple way to introduce new operations (Berry, 2002). In our model, the Command pattern is responsible for encapsulating the request information, parameters and the current state into a command object that contains the business logic. This command is then sent through the network to the model, where it is finally executed (EJBCommand). By using this approach, we achieve a complete decoupling between the controller and the model, which represents the business data and implements the rules to operate them (Sing, 2002). Following the same approach, in the model shown in figure 1, we introduce the business layer as a set of patterns that completely disconnect the controller and the view from the model, thus achieving MVC paradigm's objective. On the other hand, we define another set of patterns in order to integrate our model with inherited systems or other business models.

In the first case we apply the Business Delegate, Session Facade and EJBCommand patterns. The EJBCommand pattern (Marinescu, 2002) is a special case of the Command pattern (Berry, 2002), where the business logic is encapsulated into a serializable object created by a remote client – the controller's Command – and sent through the network to the EJB container where it will be executed – by invoking its execute method (Johnson, 2003). This scheme provides the advantages of the Command pattern in an environment where the business logic is distributed, therefore allowing the execution of business rules without overloading the application by a massive usage of EJBs. The EJBCommand's only requirement for its execution is a distributed component that processes it. The other two patterns – the Business Delegate and the Session Façade – jointly provide a good solution in order to decouple the model from the view and the controller. These patterns also hide the business rules' implementation details and allow the execution of commands in the Session Facade (Alur, 2001).

For the second case, we have defined two patterns to ease the integration between the business model and the inherited systems or other business models. The Data Object Access pattern supplies a mechanism to abstract and encapsulate access to the data sources, therefore achieving warehouse independency (Alur, 2001). It also achieves a clear separation between the business logic and the data logic, increasing the applications' maintenance capabilities (Berry, 2002). The Service Activator pattern describes a way to access other business models and services in an asynchronous manner. When a message is received, the Service Activator locates and invokes the business methods necessary to resolve the request asynchronously (Alur, 2001).

The view is responsible for showing the data output by the MVC model. One of our models' goals is to decouple the presentation from the controller and the model, and to achieve this the model's output is first produced in XML format, for its later transformation by XSLT sheets into the final presentation shown to the client. XML/XSLT is an elegant way to separate the data from the presentation and to free it from any particular technology (Johnson, 2003). In this scheme, the Request Dispatcher returns the data initially output by the model, as an XML document. In the next step, at the view stage, the XML data is transformed by XSLT. Since XSLT can return any format, it is a notably flexible technology, supporting multiple types of clients (Mercay, 2002). Besides, it provides

a simple way to modify the view without changing the model, and facilitates parallel application development.

In the next point we describe the implementation of the proposed model, based on SUN's recommendations (Sing, 2002) –use of existing frameworks in the market–. We will detail the process of framework selection and the integration tasks carried out, finalizing with the advantages and the power offered by the obtained implementation.

4 IMPLEMENTATION

We can find in the market different frameworks based on Model 2 that facilitate the development of J2EE applications. Some of these are integrated with servers and tools specific of their corresponding J2EE providers. There are also open source frameworks supported by a solid community and widely spread in the last years (Sing, 2002). In order to select the most appropriate framework for our purposes, we carried out a survey which focused on open source frameworks, due to its inexpensive costs and the technological maturity reached by some of them.

A suitable framework must achieve the following two objectives: first, it must adapt to our model's specifications, and second it must be a good framework – as described in the background. After analyzing the most used, widely spread open source frameworks in the Java community (Struts, Cocoon, Maverick, SOFIA, Spring, WebWork, Tapestry, Turbina and JSF), we observed that none of these completely satisfied the established requirements. So we decided to use a different framework for each of the model's parts (the Model, the View and the Controller): we chose Struts as the Controller, Cocoon for the View and StrutsEJB for the Model. In the next point, we describe each of these frameworks and how they fit into our model.

4.1 Struts

Struts is a framework that implements a powerful and flexible controller based on the *Service To Worker* pattern. Struts' main advantages are:

Integration flexibility: Struts' architecture provides flexibility for choosing the view and the model to be used. The view is based on the plug-ins concept. A plug-in is a dynamic mechanism by means of which a component or set of components that implement certain functionality in our application can be replaced by another ones, by simply modifying the application's configuration. This model is implemented through JavaBeans, thus allowing its integration with other frameworks.

- It is supported by a solid community: Struts is a project from the Apache Software Foundation which has been consolidated as one of the most important organizations in the open source scope. In (Sing, 2002), SUN recommends using Struts as the framework for the Web tier.
- Performance: Struts is a lightweight and mature framework, recommended for production environments.

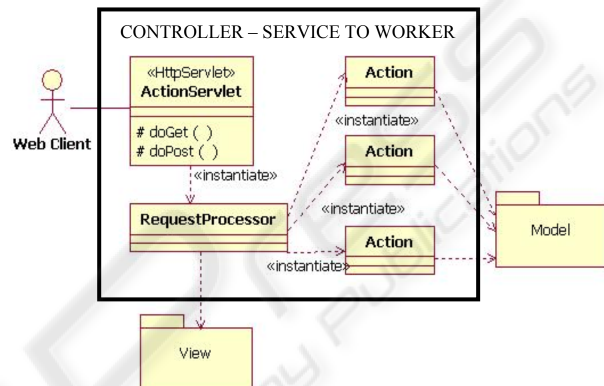

Figure 2: Struts's class diagram

For these reasons, Struts is the ideal choice in order to implement our model.

Struts is composed of three main components:

- The *ActionServlet* (the model's *Front Controller*), which is responsible for the application configuration and for receiving and analyzing the clients' requests. This component extracts from the configuration file (*strut-config.xml*) the general configuration parameters, the set of components that defines its behaviour (plug-ins) and the properties of each request. After performing these tasks, it delegates the control in the *RequestProcessor*.
- The *RequestProcessor* (*Request Dispatcher* in the model), that creates an instance of the action (*Command* pattern) associated to the received request and executes it.
- The *Action* (*Command* in the model). For each operation or use case, the developer creates an action (object) that inherits from the Action component. Each action is associated to a request type in Struts' configuration file.

4.2 Cocoon

Cocoon is a framework that performs transformations on XML documents using XSLT

stylesheets. It is a mature technology (from 1999), created and maintained by the Apache Software Foundation. It is considered as one of the most powerful and robust frameworks in the transformation of XML documents.
When Cocoon receives a request, it is analyzed based on the configuration parameters stored in the site-map.conf file. These parameters point to the steps that Cocoon's engine must follow in order to present the final view. Generally, this process will consist of three steps:

- The first step is to generate the XML document associated to the request. For this purpose, Cocoon uses the Generator (Moczar, 2003) interface, which allows specifying different source types to get the XML document from (a static file, JSP page, Servlet, data stream, session variable or request parameter). It also provides a simple mechanism in order to develop our own generators.
- The second step consists in defining the XSLT sheet that will guide the transformation. To achieve this, Cocoon uses the Transformer (Moczar, 2003) interface, which is implemented depending on the desired transformation type –like in the generators' case. By default, the XSLT sheet is obtained from a static file.
- The third, final step consists in formatting the final output. Cocoon invokes the Serializer (Moczar, 2003) interface, which is responsible for applying a format based on the client's needs (for example FOP, XML, HTML or WML). It is also possible to create new serializers based on the application's needs.

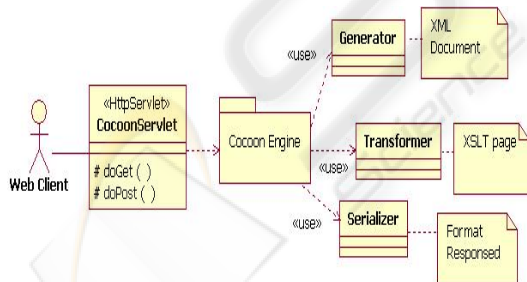

Figure 3: Cocoon's class diagram

4.3 Cocoon plug-in for Struts

In order to generate the view, Struts uses the *Tiles framework* (Sam-Bodden, 2004), which is based on the JSP technology. To replace this framework we must substitute the associated plug-in for a new one. In (Brown, 2003), Cocoon Plug-in for Struts is described as a framework that allows the integration between Struts and Cocoon. In order to use the mentioned plug-in, we must replace the Tiles plug-in with it (this can be configured in the struts-config.xml file).
Cocoon plug-in for Struts consists of three components:

- The *CocoonPlugin*: this component replaces Struts' *RequestProcessor* with the *CocoonRequestProcessor*, thus modifying the behaviour so that the view is generated by Cocoon.
- The *CocoonRequestProcessor*: this component inherits from Struts' *RequestProcessor* object class, implementing the functionality required to generate the view in Cocoon. Like the *RequestProcessor*, it creates and executes the corresponding action. After obtaining the action's result, it delegates in the *CocoonHandler* component.
- The *CocoonHandler* is responsible for communicating with Cocoon's engine, which starts the view generation process.

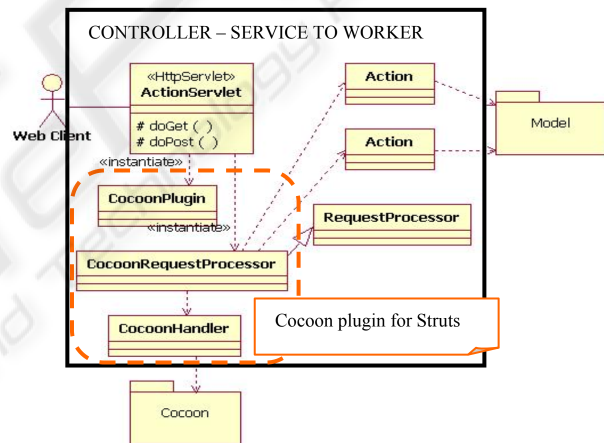

Figure 4: Integration between Cocoon and Struts

4.4 StrutsEJB

StrutsEJB is a mini-framework that implements the patterns specified in our model (Business Delegate, Session Façade and *EJBCommand*), therefore completely decoupling the model from the view and the controller (Yoshikawa, 2003). This framework provides the architecture and design necessary to integrate Struts with a distributed model based on EJBs.
StrutsEJB's main components are:

- *DefaultAction*: it is a Struts Action that represents a generic action which avoids creating an action for each operation in the Web tier. Its function

consists in creating an instance of the StrutsEJB command (*EJBCommand*) associated to the request and to store in this command the parameters and session variables required to run its business logic.

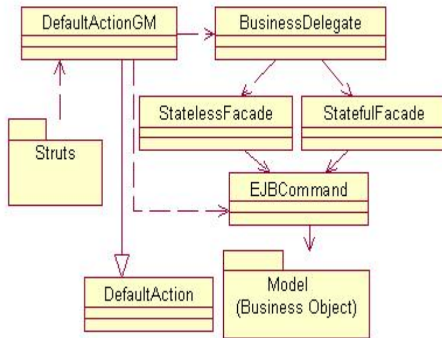

Figure 5: StrutsEJB class diagram

- *BusinessDelegate*: this component implements the *BussinessDelegate* pattern. It resides in the Web tier and acts as a model abstraction at the client's side, hiding the details of the business services implementation, the search for these services and the access to the EJB architecture. The *ServiceLocator* is the helper component – an implementation of the Service Locator pattern – used to look for services.
- *StatelessFacade* and *StatefulFacade*: these components implement the *Session Facade* pattern, providing a facade to access the business components, thus offering a uniform communications system to clients. Usually, a particular *SessionFacade* is created for each set of use cases. Since StrutsEJB is based in the *EJBCommand* pattern, this function is transferred to the commands sent from the Web tier. In this way, *SessionFacade* provide a distributed environment in order to execute such commands. The *BusinessDelegate* will invoke the corresponding *SessionFacade*, passing the command to this façade, which will in turn execute the given command inside the EJBs container. The *StatefulFacade* is to be used in applications where it is necessary to maintain the state; otherwise, the *StatelessFacade* will be used.
- *EJBCommand*: this component implements the *EJBCommand* pattern, jointly with *BusinessDelegate* and *SessionFacade*. It is an abstract component that encapsulates the business logic for each use case (or set of use cases). Therefore, it is necessary to extend this component for each operation (or set of related operations), creating a new command. In this way, developers do not need to use EJBs in their applications (although they can still use them voluntarily), but the business logic is kept inside the EJBs container. *EJBCommands* are created from the Web tier by means of an action (*DefaultAction*), which sends them to the model where they will be executed by a *SessionFacade*.

StrutsEJB was designed for its integration with the Struts controller, but using JSP as the view generator technology. In the proposed model we base the view generation on XML/XSLT, which implied modifying StrutsEJB in order to adapt it to Cocoon. To achieve this task, we created a new generic action that extended StrutEJB's *DefaultAction*'s functionality, *DefaultActionGM*. This modification solved the problem of passing the XML document generated in the model to the *CocoonRequestProcessor*, as well as other internationalization, multiple devices support and error managing problems.

4.5 Filters

In the model's specification we commented the Intercepting Filter pattern's importance. In order to implement this pattern, we have not used any framework. We have used the J2EE Filter, which provides a mechanism similar to a Servlet. The Filter is controlled by a Web container and can be inserted, in a declarative way, into the request-reply HTTP (Allamaraju, 2002) process. For each Filter needed, the developer creates a new class that extends the Filter interface and incorporates it to the application adding the corresponding directives to the configuration file, web.xml

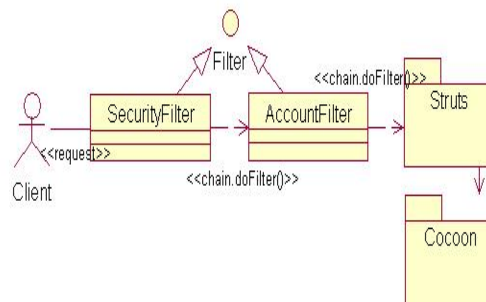

Figure 6: Intercepting Filter class diagram

5 CONCLUSIONS

It is advisable for organizations to face changes in the business processes with caution. These changes are introduced mostly by the Internet's evolution.

The enterprise platforms are complex and their learning curves are really steep. The amount of time and the effort required in order to use them correctly are seldom available in most projects' planning, often submitted to strong time constraints. In the present work, we have analyzed these problems and revised the main models, platforms and frameworks that seek to facilitate the convergence towards the new business models. The main conclusion derived from this study is that there is no proposal that satisfies all the requirements demanded by these models. However, there are partial solutions for them, which can be of great value. For this reason, we propose a model based on the integration of design patterns and standard technologies, a model that keeps the independence proposed by MVC, facilitating the analysis and development of complex applications, focusing on the developers' effort in the implementation stage and allowing a progressive learning of the technology involved.

We are currently applying this model and its associated methodology to our own developments, therefore allowing us to validate the proposal, to refine the model and finally to take into account new aspects. For example, we are currently working on the integration of a security infrastructure called Single Sign-On, called JOSSO. In future research lines we would like to improve our proposal so that it supports other business models and technologies, and to extend our model in order to use Web Services to further decouple the MVC elements.

REFERENCES

Allamaraju, S., Beust, C., Davis, J., Jewell, T., Johnson, R., Longshaw, A., Nagappan, R., Dr. Sarang, P.G., Toussaint, A. Tyagi, S., Watson, G., Wilcox, M., Williamson, A., O'Connor, D., 2002. *Programación Java Server con J2EE Edición 1.3*. Anaya Multimedia.

Alur, D., Crupi, J., Malks, D., 2001. *Core J2EE patterns, best practices and design strategies.* Prentice Hall.

Berry, A. C., Carnell, J., Juric, M.B., Moidoo Kunnumpurath, M., Nashi, N., Romanosky, S., 2002 *J2EE design patterns applied, real world development with pattern frameworks*. Wrox press

Brown, D., 2003. *Cocoon Plugin For Struts 1.1.* http://struts.sourceforge.net/struts-cocoon.

Cade, M., Roberts, S., 2002. *Sun certified enterprise architecture for J2EE technology, studie guide*. Prentice Hall.

Crawford, W., Kaplan, J., 2003. *J2EE design patterns*. O'Reilly.

Giang. Z., 2003. http://www2.tw.ibm.com/developerWorks/tutorial/SelectTutorial.do?tutorialId=77.

Gilart-Iglesias, V., Maciá-Pérez, F., Hernández-Sáez, A., Marcos-Jorquera, D., García-Chamizo, J. M. *A model for developing J2EE applications based on design patterns.* Proceedings of IADIS International Conference on Applied Computing 2005. Algarve, Portugal, 2005.

Gong, L., 1999. *Inside Java 2 paltform security.* Addison-Wesley.

Harmon, P., Rosen, M., Guttman, M., 2001. *Developing E-business Systems and Architectures: A Manager's Guide.* Morgan Kaufmann Publishers.

Hansmann, U., Merk, L., Niklous, M. S., Stober, T., 2003. *Pervasive Computing, second edition.* Springer.

Johnson, R, 2003. *Expert one-on-one, J2EE design and development*. Wrox press.

Marinescu, F., 2002. *EJB design patterns, advanced patterns, processes and idioms.* Wiley.

Mercay, J., Bouzeid, G., 2002. *Boost Struts with XSLT and XML.* http://www.javaworld.com/javaworld/jw-02-2002/jw-0201-strutsxslt.html

Moczar, L., Aston, J., 2003. *Cocoon, Developer's handbook.* Developer's Library.

Sam-Bodden, B., Judd, C. M., 2004. *Enterprise java development on a budget, leveraging java open source technologies.* Apress.

Sing, I., Stearns, B., Jonson, M., 2002. *Design Enterprise Applications with J2EE Plantaform, Second Edition*. Addison-Wesley.

Stelting, S., Maassen, O., 2002. *Patrones de diseño aplicados a java.* Prentice Hall.

Weaver, J.L., Mukhar, K., Crume, J., 2004. *Beginning J2EE 1.4, from novice to professional*. Apress

Yoshikawa, K., 2003. *StrutsEJB.* https://strutsejb.dev.java.net/