Journal of Theoretical and Applied Electronic Commerce Research
ISSN 0718–1876 Electronic Version
VOL 5 / ISSUE 1 / APRIL 2010 / 25-38

This paper is available online at
www.jtaer.com
DOI: 10.4067/S0718-18762010000100004

An Assessment of Design Patterns' Influence on a Java-based E-Commerce Application

Maria Mouratidou[1], Vassilios Lourdas[2], Alexander Chatzigeorgiou[3] and Christos K. Georgiadis[4]

University of Macedonia, Department of Applied Informatics, Thessaloniki, Greece
[1] mai0502@uom.gr, [2] mai0501@uom.gr, [3] achat@uom.gr, [4] geor@uom.gr

Received 23 April 2009; received in revised form 23 December 2009; accepted 8 March 2010

Abstract

Design patterns, acting as recurring solutions to common problems, offer significant benefits such as avoiding unnecessary complexity, and promoting code reuse, maintainability and extensibility. This paper describes how four not technology-specific or language-specific design patterns (Front Controller, Model View Controller, Transfer Object and Service to Worker) can be applied to one typical e-commerce application developed using Java EE platform. The first goal is to evaluate the improvement of design properties after the implementation of each design pattern using software metrics. Another goal is to assess the influence of design patterns on the maintainability of the e-commerce application under study by examining the evolution of software metrics when performing certain extensions. The results indicate that the application of patterns positively influences design properties such as coupling, complexity and messaging implying a possible improvement in high-level quality attributes such as flexibility, extensibility and reusability.

Key words: Design Pattern, e-Commerce Application, Java Enterprise Edition, Software Metrics, Software Quality

Journal of Theoretical and Applied Electronic Commerce Research
ISSN 0718–1876 Electronic Version
VOL 5 / ISSUE 1 / APRIL 2010 / 25-38

This paper is available online at
www.jtaer.com
DOI: 10.4067/S0718-18762010000100004

1 Introduction

The Java Enterprise Edition (Java EE) platform provides the ground for the development, deployment and execution of applications in a distributed environment [10]. It is a popular platform for the development of enterprise applications, and is actually ideal for the easy development of complex, demanding projects of large scale, such as those usually required in e-commerce environment [5], [9], [11].

Java EE is based on well-defined components to provide server-side and client-side support for developing multi-tier e-commerce applications. Although Java EE applications may comprise of three or four tiers, they are generally considered as parts of three-tier architecture, because of their distribution in three layers: a client tier, a middle tier and a back-end tier [18]. The client tier may offer support for a multiplicity of client types, such as HTML pages generated by JavaServer Pages (JSP), or Java applets. The middle tier, a Java EE server, has two roles: with its Web-oriented component (Web tier), it maintains client services through Web containers (e.g. Servlets, JSP). Additionally, with its business-oriented component (EJB tier), it supports business logic component services through Enterprise JavaBeans (EJB) containers. On the back-end tier, the enterprise information systems are accessible by the way of standard APIs (e.g. JDBC) [12], [24]. The server on this tier hosts the database and the enterprise's data. The three-tier applications extend the two-tier client-server model by placing an application server between the client application and the place where the enterprise data are stored.

As modern e-commerce applications become very complex, the need for lower maintenance costs becomes more and more obvious. A part of this solution would be to develop the application in such a way, that it will be easy for someone to extend and further develop it. The design patterns are one approach for solving this problem and they are applicable in object-oriented programming languages, such as Java. A design pattern is a well defined solution to a repeatedly occurring problem. From the programmer's point of view, a design pattern comprises a set of specific interactions that can be applied in common objects to solve a known problem [8].

This paper describes the implementation of specific design patterns (*Front Controller*, *Model View Controller*, *Transfer Object*, and *Service to Worker*) in a typical e-commerce application, such as the electronic bookstore. Moreover, we present the resulting benefits after the implementation of each pattern. The benefits are described as quantities by "measuring" the improvement in quality of source code after the several aspects of implementation of each pattern. The quality of code is evaluated by appropriate metrics and the differences in quality are measured by comparing the alterations of these specific metrics' values. The paper also shows how design patterns influence maintainability by presenting three potential extensions of this application.

Although a number of researchers have studied the effect of design pattern application on extensibility and reusability employing conventional applications (without reaching the same conclusions however), to the best of our knowledge there hasn't been any empirical study that investigates the impact of design patterns on the maintainability of Web applications by comparing a pattern and a non-pattern version and by assessing their evolution when implementing the same changes.

It should be noted that metrics cannot be considered as a fully reliable way to assess the quality of a software system. However, hierarchical models for the assessment of quality attributes in object-oriented systems, such as the QMOOD model proposed by Bansiya and Davis [2], relate design properties such as coupling, cohesion and complexity to high-level quality attributes such as reusability and flexibility. In such models each of the design properties represents attributes of the design that can be assessed by one or more metrics. Although the metrics that are used in each model differ, there is a consensus that quality attributes such as flexibility, extensibility, reusability and understandability (all of which are related to the generic term of maintainability which our study aims to investigate) depend on coupling, messaging and complexity properties. The metrics that have been selected in our study directly assess the aforementioned design properties.

The rest of the paper is organized as follows: in section 2, we present the research work in the area of Java EE design patterns and their influence on the quality of code so far. In section 3, we describe the e-commerce application under the study that we developed, while in section 4, we describe four design patterns and how they are implemented in our e-shop application. In section 5, we discuss the influence of the implementation of the patterns in our application using software metrics. In section 6, we discuss how the design patterns influence the extensions of the e-shop application and the paper ends with conclusions in section 7.

2 Related Work

In the recent years, there have been numerous published papers on Java EE design patterns and their influence on the quality of code. This section briefly reviews recent significant approaches and related research work in this area. A main aspect of the quality of code is its extensibility, which is the ability to change easily its behavior, to accommodate future requirements. In fact, an objective measurement of software extensibility is an open controversial issue. But, many researchers argue that the "Open-Closed" principle (meaning that software entities

This paper is available online at
www.jtaer.com
DOI: 10.4067/S0718-18762010000100004

should be open for extension, but closed for modification), is a general design tactic capable to achieve extensible object-oriented software.

In [20] particularly, the authors adopt this conjecture and they explore the relations between design patterns and the Open-Closed principle. They present an empirical experimentation to evaluate whether the theme of patterns will result in improvements on the code regarding its extensibility. They conclude that although in most cases the design patterns improve the maintenance productivity, there are also particular cases in which the design patterns are impediments in maintenance. Similar results are also presented in [3], where an investigation of five evolving programs has taken place to identify the benefits of deploying patterns for software design and maintenance. Thus, the need to evaluate if design patterns are effectively deployed, and to understand the underlying issues of effective pattern deployment, is emphasized.

In [21], authors conducted a case study to explore (among other issues) how far design patterns using server page technology may lead to better code quality. They focus on reducing code duplication (cloning), as repeated similar program structures apparently making the code larger than needed. Unifying clones with single standard representations decrease the code size and the possibility of update irregularities. In Web applications, server page technology is capable of dynamic page generation, and thus, a server page can represent many similar Web pages in a generic form, providing an alternative to cloning. The authors reorganized their first design (a first-cut solution) around widely used design patterns (such as *Model View Controller* and *Front Controller*, patterns that actually are part of the set of design patterns we study in our article). They conclude that although code size may be reduced, certain trade-offs appear. Performance, ease of indexing by search engines, platform/framework conformance and ability to use of multiple content types, are the major of those trade-offs. According also to [13], it is not always appropriate to remove clones which exist because of missing design patterns. Consequently, understanding these implications provides valuable criteria about the actual code improvement.

3 Description of the System under Study

The system that has been used as a vehicle in order to investigate the usefulness of design patterns in e-commerce applications is an electronic bookstore. It is a rather typical application, variations of which can be found in any introductory textbook on developing Web applications. Its design follows the common three-tier architecture and as such it can be considered common enough to allow generalizing the findings of this study. Four design patterns have been implemented on the selected e-commerce application.

The application utilizes two enterprise beans, a stateful session bean and a stateless session bean. Stateful session beans are components that need to maintain a conversational state on a per-client, per-session basis. That is, a different instance of a stateful session bean implementation class is used for each client, and its state is maintained between method invocations until the session is terminated. For stateful beans, the life cycle is complicated, since the instance must be associated either with a particular client across multiple method calls or with a particular persistent entity [22]. In our application, a stateful bean implements the functionality of a customer's shopping cart during his/her visit into the e-shop.

A stateful session bean has been selected to implement the functionality of a customer's shopping cart because, in contrary with a stateless, it has the ability to maintain information during a session. Therefore, using a stateful session bean, the contents of the customer's shopping cart can be maintained during his/her navigation through the on line bookstore. The stateful session bean provides methods that add and remove a book in the cart, empty the customer's shopping cart, return the collection of books that the cart contains and the amount that the customer must pay (which is calculated as the sum of prices of all books currently contained in the cart).

Stateless session beans are components that do not maintain any state between invocations. Essentially, stateless session beans provide in general utility functions to clients. Stateless session beans have a very simple life cycle, since the same instance can repeatedly service requests from different clients without the special handling that is required for stateful beans. In our application, the stateless session bean implements the business logic.

It provides methods that create a new user, update a user's personal information, delete an existing user, add, remove or update a book, return a subset of the books, users or purchases that have occurred, return a user's personal information, the book category which a customer prefers to buy books from and the books that match the query entered by the user in their title, author's last name or category name. Additionally, the stateless session bean provides methods that check the validity of the username and password that a user enters for his/her login in the bookstore, the validity of a credit card's number and the validity of a book's ISBN and carry out all required actions for the purchase of all books currently contained in a customer's shopping cart.

Furthermore, the application utilizes the following servlets:

- *AdminCatalogServlet*, which fulfils all user requests that relate to the administrative section of the application

Journal of Theoretical and Applied Electronic Commerce Research
ISSN 0718–1876 Electronic Version
VOL 5 / ISSUE 1 / APRIL 2010 / 25-38

This paper is available online at www.jtaer.com
DOI: 10.4067/S0718-18762010000100004

- *BookDetailsServlet*, which displays the book's information per user request
- *BookStoreServlet*, which handles the user's login in the bookstore
- *CachierServlet*, which completes a user's buy by asking from the user all required information (e.g. credit card details)
- *CatalogServlet*, which handles the display and search functions for a user's books, along with the functionality of adding a book in his/her shopping cart
- *ChangeUserServlet*, which handles all requests for a user's profile change
- *CreateUserServlet*, which handles all requests for the creation of a new user
- *LogoutServlet*, which handles all requests for the termination of a user's session
- *ReceiptServlet*, which completes the buy procedures of a user by displaying a message stating the success or not of the buy
- *ShowCartServlet*, which handles all requests for the display of the shopping cart contents and the removal of an item or all from it

An overview of the system's architecture is shown in the following simplified class diagram (figure 1).

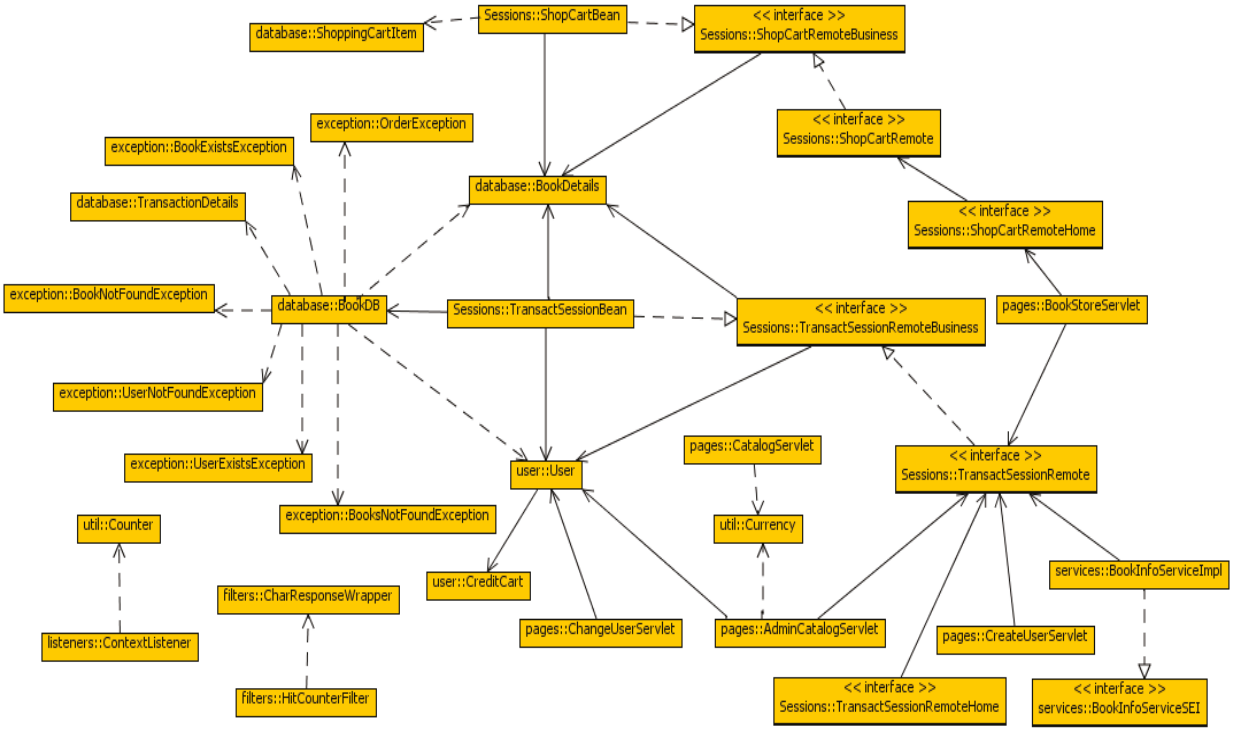

Figure 1: The Simplified UML Class Diagram of the Application

4 Design Patterns

A description of the implementation of four specific design patterns follows. We will present, as a case study, how these particular design patterns can be implemented to the bookstore pilot application.

4.1 Implementation of Front Controller

After an inspection of servlets a) *BookDetailsServlet*, b) *BookStoreServlet*, c) *CashierServlet*, d) *CatalogServlet*, e) *ChangeUserServlet*, f) *LogoutServlet*, g) *ReceiptServlet*, and h) *ShowCartServlet*, it is realized that a common

Journal of Theoretical and Applied Electronic Commerce Research
ISSN 0718–1876 Electronic Version
VOL 5 / ISSUE 1 / APRIL 2010 / 25-38

This paper is available online at
www.jtaer.com
DOI: 10.4067/S0718-18762010000100004

function exists in all of them. This common function is the check whether a user browses the bookstore either anonymously or with his/her username and password. If the user has not logged in using one of the previous two methods, then he/she is forwarded to the book-store's login page in order to login properly.

According to the principle behind the *Front Controller* design pattern, any common functionality is gathered into one application element, e.g. a servlet, which will be called before any other servlet [1], [7], [17]. The UML class diagram of the design pattern is shown in figure 2.

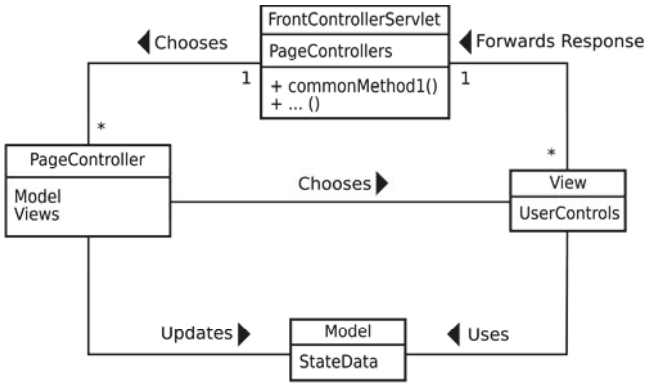

Figure 2: The UML Class Diagram of the Front Controller Pattern

After the implementation of the design pattern, any source code repetition is avoided, making the process of code maintenance easier and reducing the possibility of mistakes in case of a change in the application. In this particular situation, the same portion of code is repeated eight times (in the eight servlets).

In our application, the *FrontControllerServlet* class acts as the gathering point of the common functionality, in other words, the concept of the *FrontController*. The *FrontControllerServlet* servlet includes the common code from the aforementioned eight servlets and handles the common functionality. *FrontControllerServlet* is executed every time the user requests one of the eight servlets, and after it completes execution, it calls the appropriate *PageController* for execution, according to what was initially requested by the user servlet. Each of the servlets has the role of *PageController*.

4.2 Implementation of Model View Controller

During the use of an e-shop or any application in general, the need to change or extend it often arises. These changes might affect either the application functionality or its user interface. If no explicit boundaries between the functionality and the user interface elements in an application are defined, problems could arise. If, for example, a change in the application's display is needed, this could affect the portions of the code that implement its functionality and vice versa.

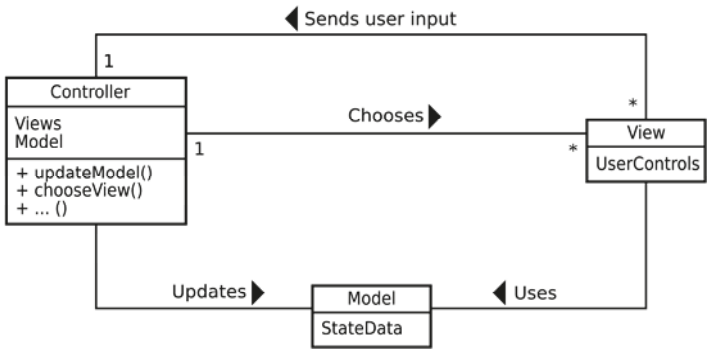

Figure 3: The UML Class Diagram of the Model View Controller Pattern

The goal of the *Model View Controller* (MVC) pattern is to set explicit boundaries between the elements of an application that implement its functionality, display and model [1], [7], [17]. As the UML class diagram of the MVC design pattern shows (figure 3), code is separated into three distinct segments: Model (stores the data and application logic for the interface), View (renders the interface, usually to the screen), and Controller (responds to user input by modifying the model). The basic principle of MVC is the separation of responsibilities. In an MVC application, the model class concerns itself only with the application's state and logic. It has no interest in how that state is represented to the user or how user input is received. By contrast, the view class concerns itself only with

Journal of Theoretical and Applied Electronic Commerce Research
ISSN 0718–1876 Electronic Version
VOL 5 / ISSUE 1 / APRIL 2010 / 25-38

This paper is available online at
www.jtaer.com
DOI: 10.4067/S0718-18762010000100004

creating the user interface in response to generic updates it receives from the model. It does not care about application logic or about the processing of input. Finally, the controller class is occupied solely with translating user input into updates that it passes to the model. It does not care how the input is received or what the model does with those updates [19], [24].

Data and classes that act as data represent the Model. The functionality is handled by classes or servlets and the display by HTML or Java Server Pages (JSP).

The initial bookstore application does not define any explicit boundaries between the elements that implement its functionality or display. In fact, the servlets are the application elements that implement both. After the application of MVC pattern for each servlet, besides *FrontControllerServlet* – which was created after the application of the *Front Controller* pattern and it does not implement any display part of the application – one or more JSPs have been created. The JSPs implement the role of the View and the servlets implement the role of the Controller in the MVC pattern. More specifically, a separation of code that relates to the functionality and code that relates to the display is applied. The latter is moved into JSPs.

4.3 Implementation of Transfer Object

The principle behind the *Transfer Object* design pattern is the creation of an object that carries multiple data from one application layer to another, e.g. from the enterprise to the presentation layer [1], [7], [17]. All class attributes are defined public, so that no getter and setter methods are created, as is the case with Java Beans. The use of *Transfer Object* minimizes the effort and time to transfer data from one layer to another and the communication in general of a Java EE application's layers. The UML class diagram of the design pattern is shown in figure 4.

Figure 4: The UML Class Diagram of the Transfer Object Pattern

In the bookstore application, there is a transfer of a great deal of data between the enterprise and the presentation layer. Some of those are the list of available books, the list of users and the list of purchases. The display of each list is done partially in pages, so that they are easy to read due to their large size. However, the latter is not fixed, because of the fact that at any time books can be added or deleted, a book's supply can reach zero, users can be added or deleted or new purchases can occur. Therefore, during a page retrieve process the length of each list should be known in advance, besides the list's elements that correspond to the currently displayed page, in order to create the links to the other pages.

For this purpose, the *TransferList* class was created which acts as the *TransferObject*. The class has two properties, an *ArrayList* object that holds the portion of the list that is requested and an integer that holds the length of the list. The *TransferList* class is not different when it holds books, user or buy objects, because of the *ArrayList's* feature to hold any kind of object. *TransferList* allows for the concurrent transfer of a great deal of the same kind of objects and their count number, which are currently stored in the database. By utilizing this, the number of database accesses has decreased, since all required data are retrieved with a single function, whereas before the pattern's implementation two databases accesses were necessary, one for the page's objects and one for the total number of objects.

4.4 Implementation of Service to Worker

The goal of the *Service to Worker* design pattern is to maintain a separation between the actions that must be executed, the display and the functionality of the application [1], [7], [17]. *Service to Worker* is an extension to *Front Controller* and uses an object that is called dispatcher. The dispatcher encloses the appropriate action to be executed and the corresponding display page according to the user's request. The UML class diagram of the pattern is depicted in figure 5.

The pattern is applied at two points, a) at the section which targets the bookstore's customers, and b) at the administrative section. The interface *Action* is used, containing the method *performAction*(), that takes an *HttpServletRequest* object as parameter and represents the user's request. This interface is implemented by the classes a) *admincatalogAction*, b) *adminlogoutAction*, c) *bookdetailsAction*, d) *cashierAction*, e) *catalogAction*, f) *changeuserAction*, g) *insertbookAction*, h) *logoutAction*, i) *receiptAction*, j) *showcartAction*, k) *transactionsAction*, and l) *usersAction*. The interface *Dispatcher* is also used, which contains the method *getNextPage*() that takes as parameter an *HttpServletRequest* object and represents the user's request. This interface is implemented by the classes *UserDispatcher* and *AdminDispatcher*. The former takes on the responsibility of executing the appropriate action according to user's request and calls the appropriate JSP (View) for the user section of the application, whereas the latter does the same for the administrative section.

Journal of Theoretical and Applied Electronic Commerce Research
ISSN 0718–1876 Electronic Version
VOL 5 / ISSUE 1 / APRIL 2010 / 25-38

This paper is available online at
www.jtaer.com
DOI: 10.4067/S0718-18762010000100004

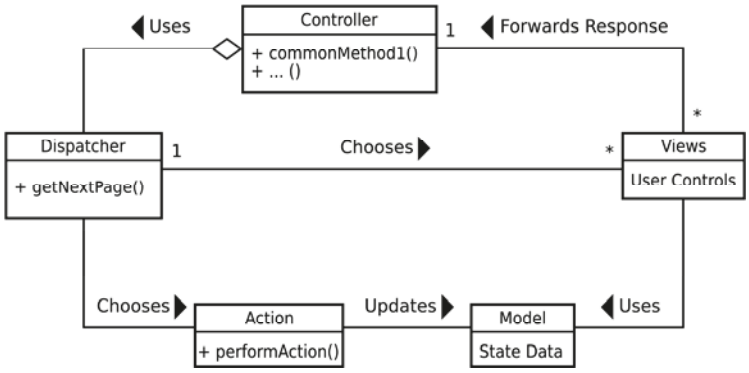

Figure 5: The UML Class Diagram of the Service to Worker Pattern

After the implementation of the pattern, the servlets are replaced by “normal” classes. This replacement enables the ability to call the *performAction*() method without the existence of a Web container, as is the case with servlets. The *Dispatcher* object also takes on the responsibility of choosing the appropriate action that must be executed and the call of the JSP that renders the response. If for any reason, the logic behind the selection of the appropriate action or JSP is changed, the implementation of *Dispatcher* is all that needs to be changed, without affecting other classes (e.g. the class that implements *Front Controller*).

5 Results and Discussion

The quality of the source code has been evaluated using a set of metrics [6], [14], [15]. Simple classes with few methods are easy to change and extend. Classes that are not tightly coupled with other classes or have low response set, cause few or no changes to other classes when they are subject to change. The metrics that are used to evaluate how easily the source code can be maintained and extended before and after the implementation of each design pattern are:

- *Coupling Between Objects (CBO),* which measures the number of other classes to which a class is coupled to
- *Changing Methods (CM),* which measures the number of distinct methods in the system that would be potentially affected by changes operated in the measured class
- *Changing Classes (ChC),* which measures the number of client-classes where the changes must be potentially operated in result of a change in the server-class
- *Response For Class (RFC)*, which measures the size of the response set for the class [6] which includes methods in the class's inheritance hierarchy and methods that can be invoked on other objects
- *Weighted Methods Per Class 1 (WMPC1),* which measures the sum of the (algorithm) complexity of all methods for a class, where each method is weighted by its cyclomatic complexity (note that cyclomatic complexity measures the number of linearly independent paths through a program's source code using a graph that describes the control flow of the program)
- *Weighted Methods Per Class 2 (WMPC2)*, which measures the complexity of a class, assuming that a class with more methods than another is more complex, and that a method with more parameters than another is also likely to be more complex. The metric simply counts methods and parameters for a class excluding any inherited methods

Generally, the values of the above metrics should be as low as possible. According to [23], high values of these metrics may be caused by a ‘design flaw’. A design flaw is possible to cause difficulties or problems during the maintenance and extension of an application. A low value or a decrease of a metric value after a change in the source code, possibly by the implementation of a design pattern, is an indicator that quality of the source code has improved.

5.1 Metric Changes after the Implementation of the Front Controller Design Pattern

The classes that are influenced the most by the implementation of the *Front Controller* design pattern are the servlets: a) *BookDetailsServlet*, b) *BookStoreServlet*, c) *CashierServlet*, d) *CatalogServlet*, e) *ChangeUserServlet*, f) *LogoutServlet*, g) *ReceiptServlet*, and h) *ShowCartServlet*. Table 1 shows the changes of CBO, RFC and WMPC1

Journal of Theoretical and Applied Electronic Commerce Research
ISSN 0718–1876 Electronic Version
VOL 5 / ISSUE 1 / APRIL 2010 / 25-38

This paper is available online at
www.jtaer.com
DOI: 10.4067/S0718-18762010000100004

metrics after the implementation of the pattern. These metric values have been decreased for all servlets. The average decrease of CBO metric for the eight servlets is 15%, though *LogoutServlet* servlet has the biggest decrease, 27% (11 to 8). The average decrease of RFC metric is 13%, though *BookStoreServlet* servlet has the biggest decrease, 28% (46 to 33). The average decrease of WMPC1 metric is 12%, though *LogoutServlet* servlet has the biggest decrease, 20% (5 to 4). The values of CM, ChC and WMPC2 metrics do not notably change.

Table 1: Changes of the CBO, RFC and WMPC1 metrics after the implementation of the Front Controller pattern

	BookDetails Servlet	**BookStore Servlet**	**Cashier Servlet**	**Catalog Servlet**	**ChangeUser Servlet**	**Logout Servlet**	**Receipt Servlet**	**Front Controller Servlet**	*System average*
CBO before	15	25	17	18	14	11	16	-	7,45
CBO after	13	21	14	17	12	8	14	22	7,34
CBO % change	-13,33	-16,00	-17,65	-5,56	-14,29	-27,27	-12,50	-	-1,5
RFC before	28	46	28	43	36	18	24	-	24,53
RFC after	25	33	25	38	33	15	21	30	23,8
RFC % change	-10,71	-28,26	-10,71	-11,63	-8,33	-16,67	-12,50	-	-3
WMPC1 before	6	19	15	25	16	5	10	-	10,88
WMPC1 after	5	16	14	23	15	4	9	9	10,53
WMPC1 % change	-16,67	-15,79	-6,67	-8,00	-6,25	-20,00	-10,00	-	-3,2

The decrease of the complexity of the servlets, extracting the common source code from them, is shown by the decrease of WMPC1 metric. Extracting source code from the servlets has resulted in the decrease of servlet "response", which is shown by the decrease of RFC metric and the decrease of coupling, which is shown by the decrease of CBO metric. The common source code of servlets that was moved to *FrontControllerServlet* servlet included coupling between classes that no longer exists in the new servlets. Additionally, the lines of source code were decreased by 0.8% (4562 to 4525 lines). If the common functions of the application were more than one, the decrease would be much more.

5.2 Metric Changes after the Implementation of the Model View Controller Pattern

Before the implementation of the MVC design pattern, a number of servlets had the dual role of a controller and a view and the same time. After the introduction of the MVC pattern, the servlets act only as the controller, which process the request and update the model, while JSPs take over the role of the view part.

Moving the source code that is responsible for the presentation of the application to JSPs has resulted to the decrease of servlet response, which is shown by the decrease of RFC metric. The decrease of complexity of the servlets is shown by the large decrease of WMPC1 and WMPC2 metrics.

The JSPs are not included in table 2 because the metrics RFC, WMPC1, WMPC2, CM and ChC for JSPs are too low. It must be noted that JSPs in general are considered as Web elements of a simple structure.

There is, also, notable decrease to CM and ChC metrics for the classes: *BookDetails*, *ShoppingCartItem*, *TransactionDetails* and *Currency*. The decrease percentage of CM and ChC metrics is not calculated because the values of the metrics before the implementation of the MVC pattern were too low and a decrease by one, e.g. from value 2 to value 1, would mean 50% decrease. If one of the above classes changes the need for changes to the system decreases. Additionally, the lines of source code have decreased by 45% (4525 to 3121 lines), because the code that has to do with the presentation has moved to JSPs. Although the number of code lines of JSPs is approximately the number of code lines that have been removed from the servlets, the distinction between presentation code and functionality code simplifies our application a lot and conceptually its extensibility.

Journal of Theoretical and Applied Electronic Commerce Research
ISSN 0718–1876 Electronic Version
VOL 5 / ISSUE 1 / APRIL 2010 / 25-38

This paper is available online at www.jtaer.com
DOI: 10.4067/S0718-18762010000100004

Table 2: Changes of the RFC, WMPC1 and WMPC2 metrics after the implementation of the MVC design pattern

	Admin Catalog Servlet	Book Details Servlet	BookStore Servlet	Cashier Servlet	Catalog Servlet	Change User Servlet	CreateUser Servlet	Logout Servlet	Receipt Servlet	Show Cart Servlet	*System average*
RFC before	88	25	33	25	38	33	35	15	21	34	23,8
RFC after	58	16	30	17	28	19	27	12	20	27	21,536
RFC % change	-34,09	-36,00	-9,09	-32,00	-26,32	-42,42	-22,86	-20,00	-4,76	-20,59	-4,6
WMPC1 before	71	5	16	14	23	15	14	4	9	10	10,56
WMPC1 after	32	5	14	5	10	5	6	4	8	7	8,51
WMPC1 % change	-54,93	0,00	-12,50	-64,29	-56,52	-66,67	-57,14	0,00	-11,11	-30,00	-19,4
WMPC2 before	15	10	11	10	10	14	14	10	10	10	11,15
WMPC2 after	11	10	11	10	10	10	11	10	10	10	10,88
WMPC2 % change	-26,67	0,00	0,00	0,00	0,00	-28,57	-21,43	0,00	0,00	0,00	-2,4

5.3 Metric Changes after the Implementation of the Transfer Object Design Pattern

The implementation of *Transfer Object* design pattern decreases the need for changes of the system in case the *TransferSessionRemoteBusiness*, *BookDetails*, *BookDB* and *BooksNotFoundException* classes change. The decrease of CM metric shows the decrease of the need for changes to the system. The average decrease of CM metric is 13%, though BookDB and BooksNotFoundException classes have the biggest decrease, 16%.

The average decrease of RFC metric is 5%, though *TransactSessionRemoteBusiness* class has the biggest decrease, 15%. At the same time, the average decrease of WMPC1 metric is 5%, though *TransactSessionRemoteBusiness* class has the biggest decrease, 15%. The average decrease of WMPC2 metric is 3%, though *TransactSessionRemoteBusiness* class has also the biggest decrease, 4.4%. The values of CBO and ChC metrics have not notably changed. Additionally, the lines of source code have decreased by 1% (3121 to 3087 lines).

5.4 Metric Changes after the Implementation of the Service to Worker Design Pattern

After the implementation of the two instances of the *Service to Worker* design pattern, one for the administration section and one for the customer section of the e-shop, all servlets, except *AdminCatalogServlet* and *FrontControllerServlet* which have the role of *Front Controller* for the two sections, no longer exist. The two *Front Controllers*, the classes that implement the *Dispatcher* interface and the classes that implement the *Action* interface share the role of those servlets.

Table 3: Changes of the CBO, RFC and ChC metrics after the implementation of the Service to Worker pattern

	Admin Catalog Servlet / admin catalog Action	Front Controller Servlet	Book Details Servlet / bookdetails Action	Cashier Servlet / cashier Action	Catalog Servlet / catalog Action	Change User Servlet / change user Action	Logout Servlet / logout Action	Receipt Servlet / receipt Action	ShowCart Servlet / showcart Action	*System average*
CBO before	26	22	14	16	18	13	10	15	18	7,45
CBO after	24	26	9	11	13	8	3	10	13	5,9
CBO % change	-7,69	18,18	-35,71	-31,25	-27,78	-38,46	-70,00	-33,33	-27,78	-20,8
RFC before	56	30	16	17	26	18	12	21	26	20,02
RFC after	36	46	10	11	18	12	4	15	19	16,74
RFC % change	-35,71	53,33	-37,50	-35,29	-30,77	-33,33	-66,67	-28,57	-26,92	-16,4
ChC before	2	2	2	2	2	2	2	2	2	103
ChC after	2	2	1	1	1	1	1	1	1	103
ChC % change	0,00	0,00	-50,00	-50,00	-50,00	-50,00	-50,00	-50,00	-50,00	0

Journal of Theoretical and Applied Electronic Commerce Research
ISSN 0718–1876 Electronic Version
VOL 5 / ISSUE 1 / APRIL 2010 / 25-38

This paper is available online at
www.jtaer.com
DOI: 10.4067/S0718-18762010000100004

Before the implementation of this pattern the administration section was implemented only by the *AdminCatalogServlet* servlet. After the implementation of the pattern the realization of the administration section of the bookstore is shared by *AdminCatalogServlet* that has only the role of *Front Controller* for this section, the *AdminDispatcher* class (that implements the *Dispatcher* interface) and the *admincatalogAction*, *adminlogoutAction*, insertbookAction, transactionsAction and usersAction classes (that implement the Action interface). At the customer section, the *FrontControllerServlet* servlet preserves the role of *Front Controller* for this section and with the *UserDispatcher* class and the *bookdetailsAction*, *cashierAction*, *catalogAction*, *changeuserAction*, *logoutAction*, *receiptAction* and *showcartAction* classes share the role of the servlets.

Table 3 shows the changes of the CBO, RFC and ChC metrics for the servlets after the implementation of the design pattern. The average decrease of CBO metric is 28%, though the *LogoutServlet* class has the biggest decrease, 70%. The average decrease of RFC metric is 27%, though the *LogoutServlet* class has also the biggest decrease, 67%. The ChC metric for all servlets except *AdminCatalogServlet* and *FrontControllerServlet* that remains 2, becomes 1. The value of CM metric does not notably change.

Moving the functionality from the servlets to classes that implement the *Dispatcher* and the *Action* interfaces results in the decrease of the response of the classes that implement the *Action* and the decrease of the coupling betweens classes. The decrease of the RFC metric shows the decrease of the response and the decrease of the CBO metric show the decrease of the coupling. The decrease of the ChC metric shows the decrease of the need for changes if one of those classes changes.

Table 4: Changes of the WMPC1 and WMPC2 metrics after the implementation of the Service to Worker pattern

	Admin Catalog Servlet	Front Controller Servlet	BookDetails Servlet	Cashier Servlet	Catalog Servlet	Change User Servlet	Logout Servlet	Receipt Servlet	Show Cart Servlet	*System average*
WMPC1 before	32	9	5	5	10	5	4	8	7	8,24
WMPC1 after	11	17	2	2	7	2	1	5	4	6,5
WMPC1 % change	-65,63	88,89	-60,00	-60,00	-30,00	-60,00	-75,00	-37,50	-42,86	-21,1
WMPC2 before	11	11	10	10	10	10	10	10	10	10,4
WMPC2 after	11	12	2	2	2	2	2	2	2	7,78
WMPC2 % change	0,00	9,09	-80,00	-80,00	-80,00	-80,00	-80,00	-80,00	-80,00	-25,2

Table 4 shows the changes of the WMPC1 and WMPC2 metrics. The average decrease of WMPC1 metric is 38%, though the *LogoutServlet* class has the biggest decrease, 75%. The average decrease of WMPC2 metric is 61%. This metric for the *AdminCatalogServlet* servlet has not been altered; for the *FrontControllerServlet* servlet has had 9% increase and the remaining servlets have had a decrease of 80%.

The big decrease of the WMPC1 and WMPC2 metrics shows the big decrease of the complexity (WMPC1) and the number of methods of the servlets that no longer exist (WMPC2) after the implementation of the Service to Worker pattern. Additionally, the lines of source code have decreased by 8.4% (3087 to 2829 lines).

6 The Influence of the Design Patterns on Application's Extensibility

In addition to the absolute metric differences between the two systems, the influence of the design patterns in an application can be demonstrated more clearly by extending this application. In order to validate the influence of the design patterns in this particular application, we will consider a few extensions regarding user requirements in e-commerce setting. Each extension is usually influenced the most by one of the design patterns. In the following section we will discuss the changes that have to be made for each hypothetical extension for the non-pattern and the pattern version of the code.

6.1 First extension

One of the extensions that are common to expect in a Web application is the addition of a utility function that is available in several parts of the application. In order to investigate the influence of design patterns on our application we choose the first extension to be the addition of a common utility and more specifically the display of day and time in a particular format in every single page of the e-bookstore.

Non-pattern version
In the version without the patterns, the following code has to be added in each of the ten servlets:

Journal of Theoretical and Applied Electronic Commerce Research
ISSN 0718–1876 Electronic Version
VOL 5 / ISSUE 1 / APRIL 2010 / 25-38

This paper is available online at
www.jtaer.com
DOI: 10.4067/S0718-18762010000100004

```
Date date = new Date();

SimpleDateFormat sdf = new SimpleDateFormat("EEE d MMM yyyy HH:mm:ss Z",
request.getLocale());

String sdate = sdf.format(date);

out.println("<p align=\"right\">" + sdate + "</p>");
```

An object of type Date and an object of type *SimpleDateFormat* to format the Date object have been created to display the date and time in a particular format. Two import statements are also necessary for the creation of these two objects:

```
import java.util.Date;

import java.text.SimpleDateFormat;
```

As a result, 60 new lines of code (6 in each of the 10 servlets) have been added.

Pattern version
On the contrary, in the version with the design patterns, 6 lines of code have to be added, only in three servlets, the *CreateUserServlet* servlet that creates a new user and the two front controllers (*AdminFController* and *FrontControllerServlet*). The use of the MVC design pattern separates the functionality from the presentation and as a result, the line that displays the date and time:

```
out.println("<p align=\"right\">" + sdate + "</p>");
```

is replaced with the line that adds a new attribute with the current date and time value to the Request object:

```
request.setAttribute("sdate", sdate);
```

Therefore, the JSP that displays the contents of each page gets the exact date and time of the user requests. This also means that each JSP that uses the *sdate* attribute should include two more lines. The first one is:

```
<jsp:useBean id="sdate" scope="request"  class="java.lang.String" />
```

and the second one is:

```
out.print(sdate);
```

The former is used to indicate to the JSP the attribute *sdate* and the latter is used to display the date and time.

It becomes clear that in the version with the design patterns the lines of code that have to be added are significantly less. They are only 38 lines (6 in each of the three servlets and 2 in each of the 10 JSPs). This benefit comes out especially by the two instances of the *Front Controller* design pattern. It is worth mentioning that the benefit of the implementation of this design pattern increases as the common source increases.

6.2 Second extension

Changes in an application's interface are often common place during the use by the end-users. These changes may be related only to the interface styling or even to hiding some functionality from e.g. a non authenticated user. As a result, in our application, the second change deals with the administrative section of the e-bookstore. The menu on the left should be displayed only when the administrator has logged in successfully.

Non-pattern version
In the version without the design patterns, the administrative section is only controlled by a servlet. Therefore, a new boolean variable must be defined, e.g. *logoutSelected*, which will be true when the administrator logs out or he is not logged in and false when he is logged in. Based on the value of this variable, the menu will be displayed as needed. The code that must be added is the following:

```
boolean logoutSelected = false;

if ((request.getParameter("action") != null) &&
     (request.getParameter("action").equals("logout"))){

              logoutSelected = true;
```

Journal of Theoretical and Applied Electronic Commerce Research
ISSN 0718–1876 Electronic Version
VOL 5 / ISSUE 1 / APRIL 2010 / 25-38

This paper is available online at
www.jtaer.com
DOI: 10.4067/S0718-18762010000100004

```
}

if (( ! logoutSelected ) && ( adminlogin != null ) ){
...

//display menu//

}
```

Additionally, the following code should be added in the section that checks if the administrator login is valid. This code requests the same servlet with new parameters. After a successful login the page will show the menu and the book catalog of the e-bookstore.

```
String nextPage="/admin/catalog";

RequestDispatcher dispatcher = getServletContext().getRequestDispatcher(nextPage);

dispatcher.forward(request, response);
```

Pattern version
In the version with the design patterns, all that should be done is to erase the code that displays the menu from the two corresponding JSPs (AdminLogin.jsp and adminlogout.jsp) and to add one line to the *Front Controller* of the administrative section, *AdminFController*. This line requests the display of the book catalog after a successful login. This benefit derives by the MVC design pattern because of its separation between the functionality and the presentation.

6.3 Third extension

It is expected that during the use of an application, the need for a number of new functionalities arises, which the developer must add to the application. The third change is an extension to the administrative section of the application with new functionality. The administrator will be able to see which book category each user prefers to buy.

In the version without the patterns, a new else if statement that checks if the administrator chose this functionality, should be added. This block of code includes approximately 48 lines of new code. A new line will also be needed to add this option in the menu.

In the version with the design patterns there is no need to change any of the existing files, in accordance to the Open-Closed principle which states that changes should be made by adding new code rather than modifying existing code. In this case, two new files will be created, favcatAction.java and favcat.jsp. The former is 33 lines long and implements the functionality and the latter implements the display and has approximately the same size with the other JSP files of the application. A new line will also be needed to add this option in the menu in 4 of the JSP files.

In this case, the code that we had to add in the version with the design patterns is more but the advantage is that we don't have to change or add code in existing files. The new functionality has been added by creating new files. This way, possible errors or problems are eliminated in the application.

7 Conclusion

In this article we have attempted to investigate the effect of applying well-accepted design patterns to a Java EE application. Four neither technology-specific nor language-specific design patterns have been implemented on a typical e-commerce application. The non-pattern and the pattern version of the system have been compared using appropriate metrics. The patterns that have been employed are: *Front Controller*, *Model View Controller*, *Transfer Object* and *Service to Worker*. According to the metric changes, the implementation of the *Front Controller*, which gathers common functionality in a single application element, reduces complexity, response and coupling by 12% to 15%. The response set and complexity of the servlets in the application is also reduced drastically by the introduction of the MVC pattern whose goal is to separate application logic from the presentation of data and handling of user input. The implementation of the *Transfer Object* pattern, which introduces objects for transferring multiple data from one application layer to another, decreased mainly the number of potential propagated changes if the involved classes are subject to change. Finally, the implementation of the two instances of the *Service to Worker* design pattern, which help to avoid the duplication of control code scattered throughout various views, has an impressive improvement of the code of the application. The coupling decreases 28%, the response 27% and the complexity more than 38%.

These code improvements were actually expected as most of the design patterns for Java EE Web-based applications have the goal of improving performance in their distributed execution environment [1]. According to our results, the introduction of a design pattern in a Web application providing typical e-shop functionality seems to

This paper is available online at
www.jtaer.com
DOI: 10.4067/S0718-18762010000100004

influence low-level design properties in a positive way implying easier maintenance for implementing changes that occur as new requirements arise. Our results confirm previous studies (regarding the development of an ERP/support chain system), concluding that based on software patterns, the quality can be enhanced, software development costs be reduced, and software maintenance be improved [4]. Indeed, the outcomes of our research are in line with the concluding remarks in [16]: good use of pattern information in programming systems reduces programmer effort.

In case more than one design patterns are implemented in the same part of our e-shop demonstration application the results are cumulative. For example, the gradual implementation of the patterns: *Front Controller*, *MVC* and *Service to Worker*, to the e-shop, results in a continuous quality improvement for the servlets' code and consequently for the maintainability and extensibility of the application.

Three extensions of the application that consist of typical examples of new requirements have been studied. The quantitative analysis has shown that the use of design patterns can make the ground easier for the implementation of changes in an application and practically reflects a quality improvement with regards to maintainability.

However, we believe that the decision whether a specific design pattern should be applied or not, should be taken considering the goal of each pattern and the problems that it can solve, rather than solely considering its effect on design metrics. In other words, the effect of the application of design pattern on low level design properties is worth exploring, but should not be the primary reason for applying or avoiding them.

References

[1] D. Alur, J. Crupi, and D. Malks, Core J2EE Patterns: Best Practices and Design Strategies (2nd Edition). Santa Clara, CA: Prentice Hall / Sun Microsystems, 2003.
[2] J. Bansiya, and C. G. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Transactions on Software Engineering, vol. 28, no. 1, pp. 4-17, 2002.
[3] J. M. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander, Design patterns and change proneness: An examination of five evolving systems, in Proceedings 9th International Software Metrics Symposium, IEEE Computer Society Press, Sydney, Australia, Sep. 2003, pp. 40−49.
[4] C. Chang, C. Lu, W. C. Chu, N. Hsueh, and C. Koong, A case study of pattern-based software framework to improve the quality of software development, in Proceedings of the ACM Symposium on Applied Computing, Honolulu, Hawaii, 2009, pp. 443-447.
[5] Q. Chen, J. Yao, and R. Xing, Middleware components for e-commerce infrastructure: An analytical review, Issues in Informing Science & Information Technology, vol. 3, pp. 137-146, 2006.
[6] S. R. Chidamber, and C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, 1994.
[7] W. Crawford and J. Kaplan, J2EE Design Patterns. Sebastopol, CA: O'Reilly Media, 2003.
[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Upper Saddle River, NJ: Addison-Wesley, 1995.
[9] S. He, Function point metrics improvement and application in e-commerce, in IFIP Volume 254, Research and Practical Issues of Enterprise Information Systems II, vol. 1 (Xu, Tjoa, Chaudhry, and Sohail, Eds.), TC 8 WG 8.9 International Conference on Research and Practical Issues of Enterprise Information Systems, China, Springer, pp. 395-398, 2007.
[10] E. Jendrock, J. Ball, D. Carson, I. Evans, S. Fordin, and K. Haase, The Java EE 5 Tutorial. Santa Clara, CA: Sun Microsystems Press, 2008.
[11] M. Jiang, L. Li, M. Hu, and Y. Ding, Design and model analysis of the e-commerce development platform for 3-tiered web application, in Proceedings of the International Conference on Advanced Language Processing and Web Information Technology (ALPIT), 2008, pp. 581-584.
[12] R. Johnson, Expert One-on-One J2EE Design and Development. Indianapolis, IN: Wiley Publishing, 2002.
[13] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, An empirical study of code clone genealogies, in Proceedings 10th European Software Engineering Conference & the 13th Foundations of Software Engineering, 2005, pp. 187-196.
[14] W. Li, and S. Henry, Object-oriented metrics that predict maintainability, Journal of Systems and Software, vol. 23, no. 2, pp. 111-122, 1993.
[15] M. Lorenz, and J. Kidd, Object-Oriented Software Metrics. Upper Saddle River, NJ: Prentice Hall, 1994.
[16] S. MacDonald, K. Tan, J. Schaeffer, and D. Szafron, Deferring design pattern decisions and automating structural pattern changes using a design-pattern-based programming system, ACM Transactions on Programming Languages and Systems, vol. 31, no. 3, pp. 1-49, 2009.
[17] R. C. Martin, Agile Software Development: Principles, Patterns and Practices. Prentice Hall. 2003.
[18] A. Mesbah, and A. Deursen, Crosscutting concerns in J2EE applications, in Proceedings of the 7th IEEE International Symposium on Web Site Evolution, IEEE Computer Society, 2005, pp. 14-21.
[19] C. Moock, Essential ActionScript 2.0. Sebastopol, CA: O' Reilly Media, 2004.
[20] T. H. Ng, S. C. Cheung, W. K. Chan, and Y. T. Yu, Toward effective deployment of design patterns for software extension: A case study, in Proceedings ACM Workshop on Software Quality, 2006, pp. 51-56.

Journal of Theoretical and Applied Electronic Commerce Research
ISSN 0718–1876 Electronic Version
VOL 5 / ISSUE 1 / APRIL 2010 / 25-38

This paper is available online at
www.jtaer.com
DOI: 10.4067/S0718-18762010000100004

[21] D. C. Rajapakse, and S. Jarzabek, Using server pages to unify clones in web Applications: A trade-off analysis, in Proceedings 29th International Conference on Software Engineering, IEEE Computer Society Press, 2007, pp. 116-126.
[22] M. J. Rutherford, K. M. Anderson, A. Carzaniga, D. Heimbigner, and A. L. Wolf, Reconfiguration in the enterprise JavaBean component model, in Proceedings of the IFIP/ACM Working Conference on Component Deployment, 2002, pp. 67-81.
[23] M. Salehie, S. Li, and L. Tahvildari, A metric-based heuristic framework to detect object-oriented design flaws, in Proceedings 14th IEEE International Conference on Program Comprehension, 2006. pp. 159-168.
[24] I. Singh, B. Stearns, M. Johnson, and the Enterprise Team, Designing Enterprise Applications with the J2EE Platform, Second Edition. Upper Saddle River, NJ: Addison-Wesley / Sun Microsystems, 2002.