Code Improvement: Implementing Design Patterns to Java EE Applications

M. Mouratidou, V. Lourdas, A. Chatzigeorgiou, C. K. Georgiadis

Department of Applied Informatics, University of Macedonia,
Egnatia 156, GR-54006 Thessaloniki, Greece
{mai0502, mai0501, achat, geor} @uom.gr

Abstract

Design patterns, acting actually 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 implemented to Java EE applications. It also calculates the code improvement after the implementation of each design pattern using software metrics. The improvement of the quality of the code is considered by measuring the decrease of complexity, coupling or/and response size.

Keywords: Java Enterprise Edition, design pattern, software metrics, software quality

1. Introduction

The Java Enterprise Edition (Java EE) platform comprises a platform for development, deployment and execution of applications in a distributed environment [Ball et al. (2006)]. It is a popular platform for the development of enterprise applications, and is actually ideal for the easy development of complex, demanding projects of big scale. Java EE is based on well-defined components to provide server-side and client-side support for developing multi-tier applications [Mesbah et al. (2005)]. 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. 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) [Singh et al. (2002), Johnson (2002)]. 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.

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 in order to solve a known problem [Gamma et al. (1995)]. The design patterns of good quality manage to maintain a balance between the problem's size they solve and the specific way they face the problem.

This paper describes the implementation of specific design patterns (Front Controller, Model View Controller, Transfer Object, and Service to Worker) in a typical enterprise 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 the quality of source code after the implementation of each pattern. The quality of code is measured by proper metrics and the differences in quality are measured by comparing the alterations of these specific metrics' values.

2. Description of the system under study

Four design patterns are implemented in the demonstration of an e-commerce application. The application utilizes two enterprise beans, a stateful session bean and a stateless session bean. The former implements the functionality of a customer's shopping cart during his/her visit in the e-shop and the latter provides methods that implement the business logic. A stateful session bean has been selected to implement the functionality of a customer's shopping cart because a stateful session bean, in contrary with a stateless, has the ability to maintain information during a session. So, using a stateful session bean, the contents of the customer's shopping cart can be maintained during his/her navigation to 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).

The stateless session bean 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 buys 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, writer'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 buy of all books currently contained in a customer’s shopping cart.

Furthermore, the application utilizes the following servlets:

a) AdminCatalogServlet, which fulfils all user requests that relate to the administrative section of the application,

b) BookDetailsServlet, which displays the book’s information per user request,

c) BookStoreServlet, which handles the user’s login in the bookstore,

d) CachierServlet, which completes a user’s buy by asking from the user all required information (e.g. credit card details),

e) 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,

f) ChangeUserServlet, which handles all requests for a user’s profile change,

g) CreateUserServlet, which handles all requests for the creation of a new user,

h) LogoutServlet, which handles all requests for the termination of a user’s session,

i) ReceiptServlet, which completes the buy procedures of a user by displaying a message stating the success or not of the buy, and

j) ShowCartServlet, which handles all requests for the display of the shopping cart contents and the removal of an item or all from it.

3. 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.

3.1 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 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 design pattern Front Controller, any common functionality is gathered into one application element, e.g. a servlet, which will be called before any other servlet [Alur et al. (2003), Crawford et al. (2003), Martin (2003)]. The UML class diagram of the design pattern is shown in fig. 1.

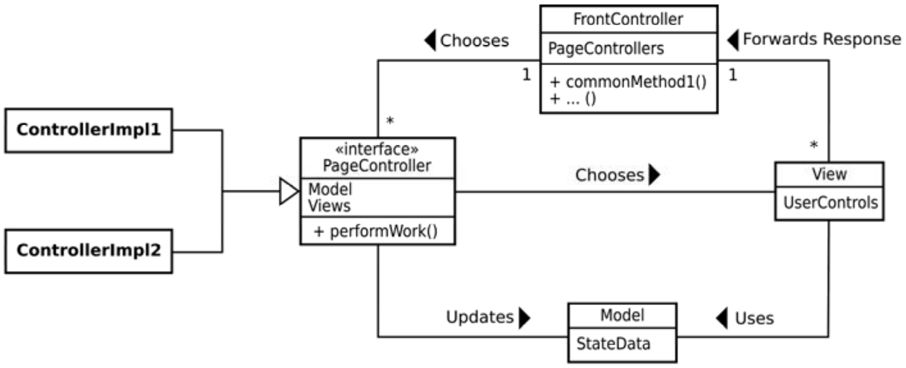


Figure 1. *The UML class diagram of the Front Controller pattern*

After the implementation of the design pattern, any source code repetition is avoided, making easier the process of code maintenance 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).

After the implementation of Front Controller, the new servlet FrontControllerServlet is created, which includes the common code from the above indicated eight servlets and handles the common functionality. FrontControllerServlet is executed every time the user requests one of the above mentioned servlets, and after it completes execution, it calls for execution the initially requested by the user servlet.

3.2 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.

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 [Alur et al. (2003), Crawford et al. (2003), Martin (2003)]. The MVC design pattern separates user interface code into three distinct classes: 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 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 [Moock (2004), Singh et al. (2002)].

The UML class diagram of the pattern is displayed in figure 2. 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 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 are created. 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.

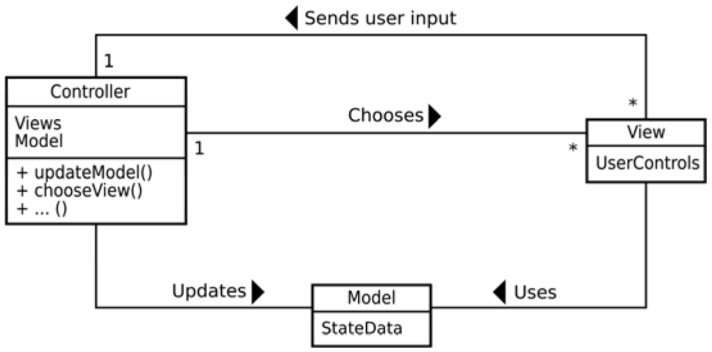


Figure 2. *The UML class diagram of the Model View Controller pattern*

3.3 Implementation of Transfer Object

The principle behind the design pattern Transfer Object 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 [Alur et al. (2003), Crawford et al. (2003), Martin (2003)]. 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 pattern's UML class diagram is displayed in fig. 3.

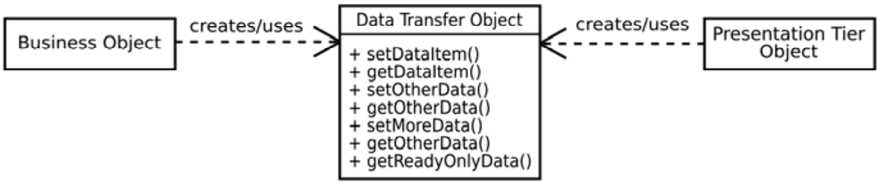


Figure 3. *The UML class diagram of the Transfer Object pattern*

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 buys. The display of each list is done partially in pages, so that they are easy to read due to their great length. 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 buys can occur. So, 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 class TransferList was created, having as 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 is 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.

3.4 Implementation of Service to Worker

The goal of the design pattern Service to Worker is to maintain a separation between the actions that must be executed, the display and the functionality of the application [Alur et al. (2003), Crawford et al. (2003), Martin (2003)]. 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 displayed in fig 4.

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.java. 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 for the user section of the application, whereas the latter does the same for the administrative section.

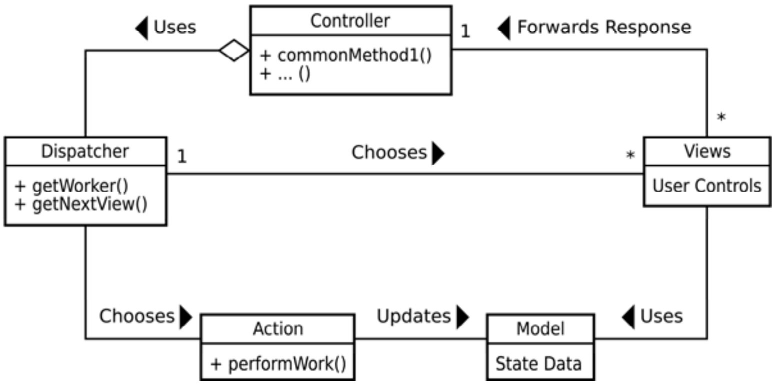


Figure 4. *The UML class diagram of the Service to Worker pattern*

After the implementation of the pattern, the servlets are replaced by "normal" classes. This replacement enables the ability to call the perfomAction() method without the existence of a web container, as is the case with servlets. The object Dispatcher 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).

4. Results and Discussion

The quality of the source code is calculated using metrics [Li et al. (1993), Lorenz et al. (1994), Chidamber et al. (1994)]. 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. So, the metrics that are used in order to be defined 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 represents the number of other classes to which a class is coupled to,

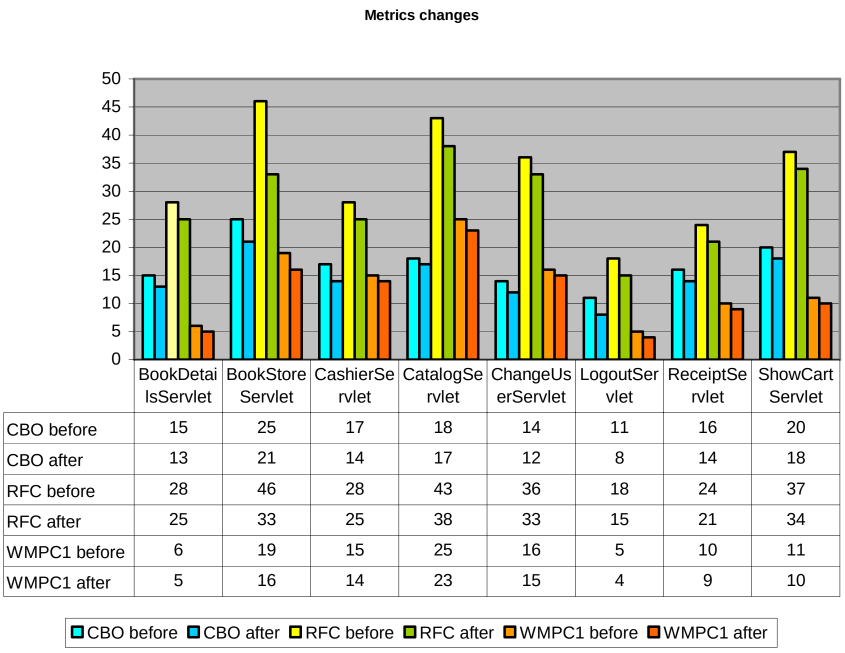
- Changing Methods (CM), which represents the number of distinct methods in the system that would be potentially affected by changes operated in the measured class,
- Changing Classes (ChC), which represents the number of client-classes where the changes must be operated in result of a change in the server-class,
- Response For Class (RFC), which represents the size of the response set for the class 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 represents the sum of the (algorithm) complexity of all methods for a class, where each method is weighted by its cyclomatic complexity[1], and
- 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.

Generally, the values of the above metrics should be as low as possible. According to Salehie et al. (2006), the high values of those 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, means that the quality of the source code has increased.

4.1 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. Figure 5 shows the changes of CBO, RFC and WMPC1 metrics after the implementation of the pattern. The metrics 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.

[1] 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



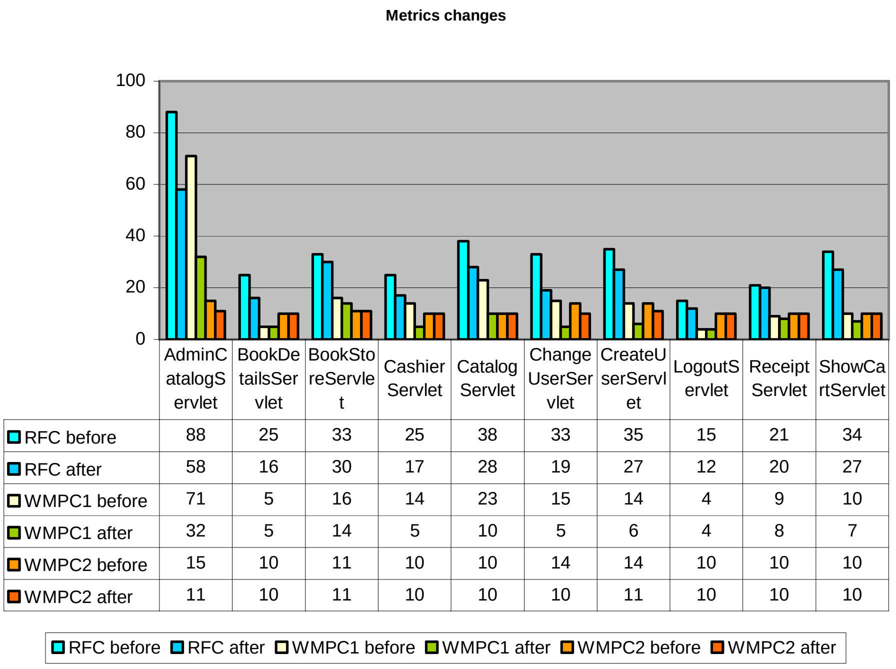
	BookDetailsServlet	BookStoreServlet	CashierServlet	CatalogServlet	ChangeUserServlet	LogoutServlet	ReceiptServlet	ShowCartServlet
CBO before	15	25	17	18	14	11	16	20
CBO after	13	21	14	17	12	8	14	18
RFC before	28	46	28	43	36	18	24	37
RFC after	25	33	25	38	33	15	21	34
WMPC1 before	6	19	15	25	16	5	10	11
WMPC1 after	5	16	14	23	15	4	9	10

Figure 5. *Changes of the CBO, RFC and WMPC1 metrics after the implementation of the Front Controller design pattern*

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 results 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 decrease by 0.8% (4562 to 4525 lines). If the common functions of the application were more than one the decrease would be much more.

4.2 Metric changes after the implementation of the Model-View-Controller design pattern

After the implementation of the MVC design pattern, the servlets that had the role of the controller and the role of the view, they now act only as the controller, which process the request and update the model, while JSPs have the role of the view.



	AdminCatalogServlet	BookDetailsServlet	BookStoreServlet	Cashier Servlet	Catalog Servlet	ChangeUserServlet	CreateUserServlet	LogoutServlet	Receipt Servlet	ShowCartServlet
🟦 RFC before	88	25	33	25	38	33	35	15	21	34
🟦 RFC after	58	16	30	17	28	19	27	12	20	27
⬜ WMPC1 before	71	5	16	14	23	15	14	4	9	10
🟩 WMPC1 after	32	5	14	5	10	5	6	4	8	7
🟧 WMPC2 before	15	10	11	10	10	14	14	10	10	10
🟧 WMPC2 after	11	10	11	10	10	10	11	10	10	10

Figure 6. *Changes of the RFC, WMPC1 and WMPC2 metrics after the implementation of the Model View Controller design pattern*

Fig. 6 shows the changes of RFC, WMPC1 and WMPC2 metrics. The average decrease of RFC metric is 25%, though ChangeUserServlet servlet has the biggest decrease, 42% (33 to 19). The average decrease of WMPC1 metric is 35%, though servlet ChangeUserServlet servlet has also the biggest decrease, 67% (15 to 5). The average decrease of WMPC2 metric is 7,6%, though ChangeUserServlet servlet has, again, the biggest decrease, 28,6% (14 to 10). This particular metric changes in only three of the ten servlets, however, for those three servlets the decrease is over 21%. The value of CBO metric does not notably change.

Moving the source code that is responsible for the presentation of the application to JSPs results 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 big decrease of WMPC1 and WMPC2 metrics.

There is, also, notable decrease to CM and ChC metrics for the classes: BookDetails (CM: 14 to 11, ChC: 9 to 6), ShoppingCartItem (CM: 6 to 5, ChC: 4 to 3), TransactionDetails (CM: 2 to 1, ChC: 3 to 2) and Currency (CM: 7 to 6, ChC: 7 to 6). The decrease percentage of CM and ChC metrics is not calculated because of the fact that 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 decrease by 45% (4525 to 3121 lines), because the code that has to do with the presentation is moved to JSPs.

4.3 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%.

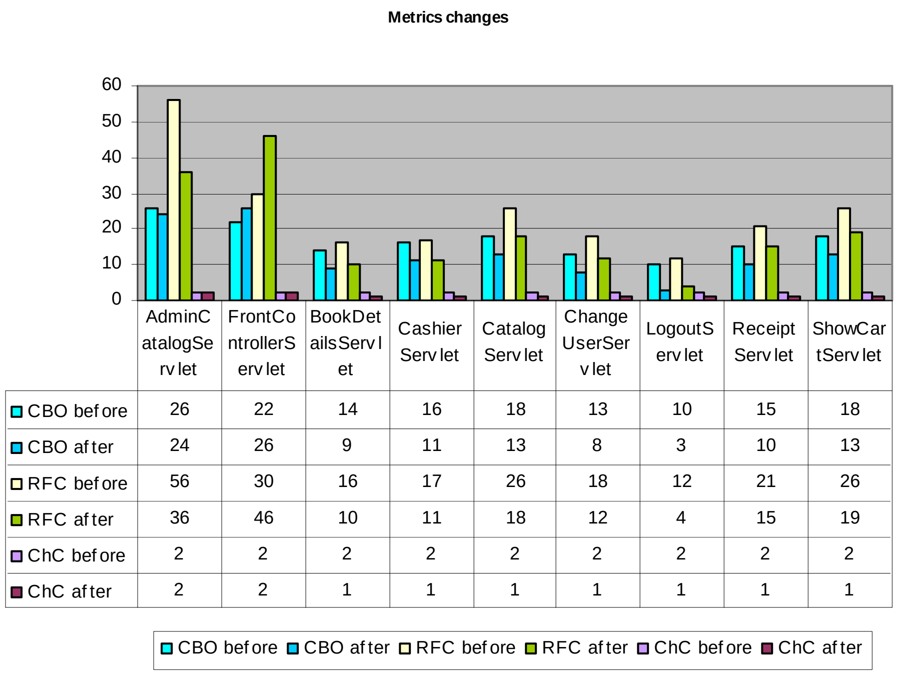
The average decrease of RFC metric is 5%, though TransactSessionRemoteBusiness class has the biggest decrease, 15%. At the same time, the average decrease of WMPC1 metric is 5%, though TransactSessionRemoteBusiness class have the biggest decrease, 15%. The average decrease of WMPC2 metric is 3%, though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

4.4 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.

Before the implementation of this pattern the administration section was implemented only by the AdminCatalogServlet servlet. After the implementation of the pattern the implementation 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.

Fig. 7 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 FrontController that remains 2, becomes 1. The value of CM metric does not notably change.



	AdminCatalogServlet	FrontControllerServlet	BookDetailsServlet	CashierServlet	CatalogServlet	ChangeUserServlet	LogoutServlet	ReceiptServlet	ShowCartServlet
🟦 CBO before	26	22	14	16	18	13	10	15	18
🟦 CBO after	24	26	9	11	13	8	3	10	13
🟨 RFC before	56	30	16	17	26	18	12	21	26
🟩 RFC after	36	46	10	11	18	12	4	15	19
🟪 ChC before	2	2	2	2	2	2	2	2	2
🟥 ChC after	2	2	1	1	1	1	1	1	1

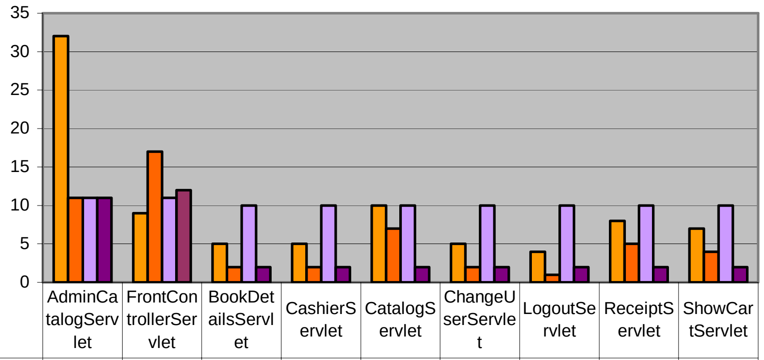
***Figure 7**. Changes of the CBO, RFC and ChC metrics after the implementation of the Service to Worker design pattern*

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.

Fig. 8 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 remains the same, for the FrontControllerServlet servlet has 9% increase and the remaining servlets have a decrease of 80%.

Metrics changes



	AdminCatalogServlet	FrontControllerServlet	BookDetailsServlet	CashierServlet	CatalogServlet	ChangeUserServlet	LogoutServlet	ReceiptServlet	ShowCartServlet
🟧 WMPC1 before	32	9	5	5	10	5	4	8	7
🟧 WMPC1 after	11	17	2	2	7	2	1	5	4
🟪 WMPC2 before	11	11	10	10	10	10	10	10	10
🟪 WMPC2 after	11	12	2	2	2	2	2	2	2

🟧 WMPC1 before 🟧 WMPC1 after 🟪 WMPC2 before 🟪 WMPC2 after

Figure 8. *Changes of the WMPC1 and WMPC2 metrics after the implementation of the Service to Worker design pattern*

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 decrease by 8.4% (3087 to 2829 lines).

5. Conclusion

The application of four not technology-specific nor language-specific design patterns on a typical Web Application has been examined by using appropriate metrics. According to the metric changes the implementation of the Front Controller influences eight servlets out of the ten servlets. After the implementation of the Front Controller, the servlets and consequently a great part of the application’s functionality have a 12% to 15% decrease of complexity, response and coupling between classes. The implementation of the MVC design pattern influences almost all servlets of the application. The response of the servlets decreases by 25% and the complexity decreases by 35%. The implementation of the Transfer Object decreases by 13% the need for changes to the system if classes that implement the design pattern change. The implementation of the two instances of the Service to Worker design pattern has an impressive improvement of the code of the application. The coupling decreases 28%, the response 27% and the complexity more than 38%.

The implementation of a design pattern to an application, like an e-shop, influences the classes that implement the specific design pattern positively and a significant improvement of a set of classes improves the application as a whole. Additionally, the lines of the source code decrease, although the number of the classes increases.

In case more than one design patterns are implemented in the same part of an application the results are better. For example, the gradual implementation of the patterns: Front Controller, MVC and Service to Worker, to the e-shop, results the continually increasing quality of the servlet code and consequently the maintenance and extensibility of the application is remarkably improved.

References

Alur, D., Crupi, J. and Malks (2003), D. *Core J2EE Patterns Best Practices and Design Stategies*, California USA, Sun Microsystems.

Ball, J. et al. (2006), *The Java EE 5 Tutorial*, California USA, Sun Microsystems Press.

Chidamber, S.R., Kemerer, C.F. (1994), *“A Metrics Suite for Object Oriented Design,”* IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476-493.

Crawford W. and Kaplan J. (2003). *J2EE Design Patterns*, USA, O'Reilly.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Johnson R. (2002). Expert *One-on-One J2EE Design and Development*, USA, John Wiley & Sons, Inc.

Li, W., Henry, S. (1993), *“Object-Oriented Metrics that Predict Maintainability,”* J. Systems and Software, vol. 23, no. 2, pp. 111-122.

Lorenz, M., Kidd, J. (1994), *Object-Oriented Software Metrics. Object-Oriented Series*, Prentice Hall.

Martin, R.C. (2003), *Agile Software Development: Principles, Patterns and Practices*. Prentice Hall.

Mesbah A. and Deursen A. (2005). *Crosscutting Concerns in J2EE Applications*. In Proceedings of the 7th IEEE International Symposium on Web Site Evolution (WSE 2005), IEEE Computer Society, pp. 14-21.

Moock C. (2004). *Essential ActionScript 2.0*, USA, O’ Reilly Media.

Salehie M., Li S. and Tahvildari L. (2006). *A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws*, 14th IEEE International Conference on Program Comprehension (ICPC'06), pp. 159-168.

Singh I., Stearns B., Johnson M. and the Enterprise Team (2002). *Designing Enterprise Applications with the J2EE Platform*, Second Edition, Sun Microsystems, Inc.