International Journal of Scientific Research in Information Systems and Engineering (IJSRISE)
Volume 1, Issue 2, December-2015. ISSN 2380-8128

The Performance Analysis of Applications Written Using MVP and MVC

Coşkun Aygun, Computer Engineering Department,
Turgut Ozal University Gazze cad. Etlik-Keçiören, Ankara, TURKEY
E-mail: caygun@turgutozal.edu.tr

Emre Kazan, Computer Engineering Department,
Turgut Ozal University Gazze cad. Etlik-Keçiören, Ankara, TURKEY
E-mail: ekazan2012@stu.turgutozal.edu.tr

Abstract - Model View Controller (MVC) is an architectural pattern introduced at the end of the 70s. The aim of MVC is the interaction of components in the view layer among themselves and with the rest of the system as well as the placements of these components. It is described as the separation of business logic from user interface code. In this way, if we want to make any changes to the view layer, we can do easily without causing any problems or changes to business logic. Model View Presenter (MVP) is the structure that retrieves user interface code (flow between pages, functioning within the user interface, etc.) in view class and carries into a different Presenter class. Thus, it can be operated and tested independently from the creation and rendering of code user interface related to the presentation. In the present study, two applications have been written using Model View Presenter (MVP) and Model View Controller (MVC) with the same requirements. To compare applications written using MVP and MVC, JMeter and speedy framework monitoring comparing tools and analysis results have been presented. MVP has been found to be more advantageous than MVC. In addition, during the encoding process by way of mocking the layers in model class, the necessity of acquisition of model data has been eliminated. The development process has been maintained without the need for database, network connection, file access, etc. And thanks to the mocking of view classes, testing of applications during the development and the creation of user interface has been unnecessary. Changes in the user interface can be made in a much easier and safe manner. After all, the changes to be made here will not affect the process in any way.

The advantage of MVP over MVC can be summarized as the ease of testability and less code dependency

Keywords *– Model View Controller (MVC), Model View Presenter (MVP), Performance analysis.*

1.INTRODUCTION

Design patterns are functionally-proven general solution proposals developed to solve similar problems frequently encountered during software design [1].

Model-view-controller (MVC) is a design pattern used in software engineering [2]. It is based on model and view abstraction in complex applications where large amounts of data are presented to users. So model and view can be organized without affecting each other. Model-view-controller solves this using a mediator called controller by isolating the data access and business logic from data display and user interaction.

Corresponding Author
Coşkun Aygun, Computer Engineering Department, Turgut Ozal University Gazze cad. Etlik-Keçiören,
Ankara, TURKEY
E-mail: caygun@turgutozal.edu.tr

Model View Presenter (MVP) basically divides the application into three parts as model, view and presenter [3].

The difference in performance between the software written using MVC and the software written using MVP will be analyzed. In the second part, architecture and benefits of the MVC is discussed. In the third part, architecture and benefits of the MVP is discussed. In the fourth part, statistical data of the software written using MVC and the software written using MVP is compared. Fifth part includes the results.

2. WHAT MVC IS AND ITS ARCHITECTURE

Model View Controller is an architectural pattern that Norwegian scientist Trygve Reenskaug introduced while visiting Xerox labs in the 1970s [4]. It is shortly called MVC. The model refers to the data displayed by the view. For example, like on/off state of a check-

box component or text data of a textfield component. The View accesses the data it needs by way of the model and performs the GUI rendering process by using this data. The controller makes it possible to change the model by the use of events from user inputs like mouse clicks, keyboard input etc. The change in the model is detected by the view by means of notifications and displayed on the screen.

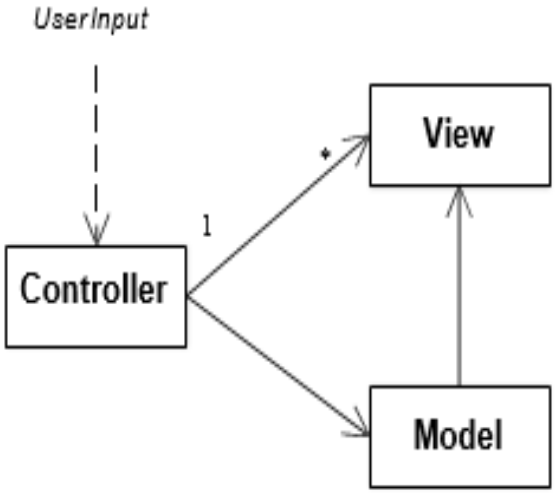

Figure 1. *The architecture of MVC*

In many current documents MVC's goal is described as "the separation of business logic from the GUI code" [5]. In this way, if we want to make any changes to the view layer, it is highlighted that we can do it easily without causing any problems or changes to business logic. However, the inventor of the MVC, Reenskaug emphasizes that the essential purpose of MVC, as can be seen in Figure 2 below, is to bridge the gap between human user's mental model and digital model that exists in the computer. With this solution, domain data, in other words, the model will be accessible, reviewable and updateable directly by the user. Changing the application into a modular structure and separating different tasks into different layers was not the first goal of MVC. As can be seen in Figure 2, model, view and controller parts are in solution, but they are parts shaped towards the main objective mentioned above. In original MVC paper "Separation of Concerns" is not a goal but a result. It is concerned with making adaptations to the MVC pattern in order to develop application in a modular way and make layers fulfill their duties independently of other layers. One of the basic reasons is that presentation in view is generally interwoven with code and business logic. There is a need for a structure that will separate two layers clearly.

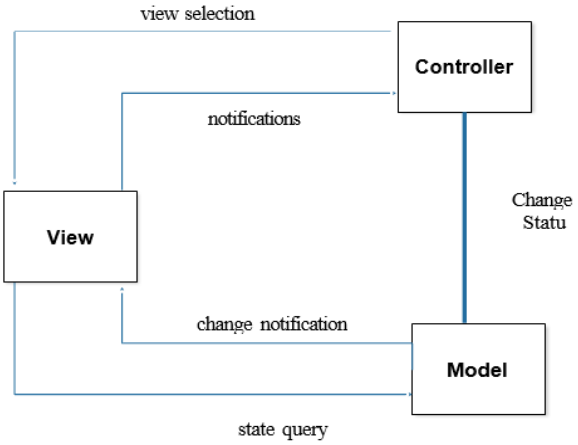

Figure 2. *The architecture of MVC*

3. WHAT MVP IS AND ITS ARCHITECTURE

The essence of Model View Presenter (MVP) is to retrieve user interface code (flow between pages, functioning within the user interface, etc.) in view class and carries into a different Presenter class [6]. Thus, it can be operated and tested independently from the creation and rendering of code user interface related to the presentation. Presenter, by getting user input by the view, transfers the work to the model layer for the execution of the relevant business logic. As a result of behavior in model, a number of state changes is very likely to occur. Presenter is informed of these state changes by way of events. Presenter reflects these state changes to view part by using appropriate methods [7]. With MVP "separation of concerns" goal can be easily realized.

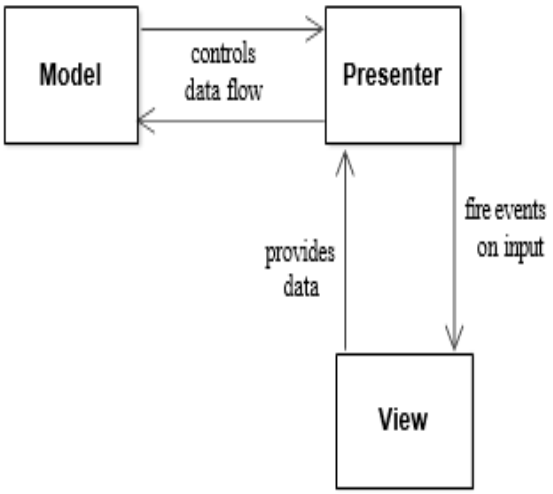

Figure 3. The architecture of MVP

In this way, the behavior of the application can also be tested easily independently

of view. MVP presents an architectural infrastructure that will help multiple groups working together at the same time develop a large application by dividing it into functional groups. It is as shown in Figure 3.

3.1. Passive View

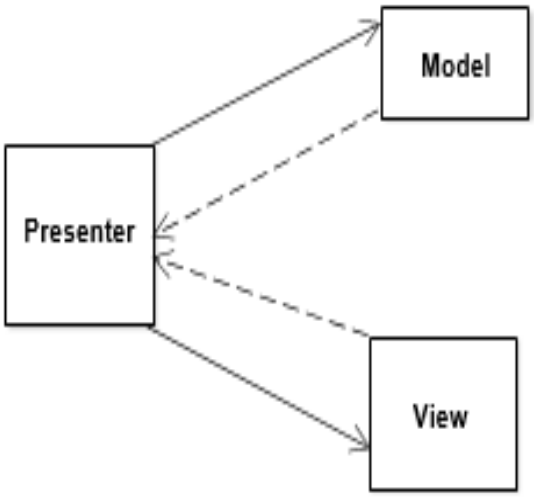

Figure 4. Passive View

The biggest difference of the variation in Figure 4 from MVC is that it is independent and unaware of view model. Presenter or Controller objects provide the coordination between model and view [8]. After Presenter carries out the necessary procedures by addressing the UI events, it is responsible for reflecting the changes to the view side.

3.2. Supervising Controller

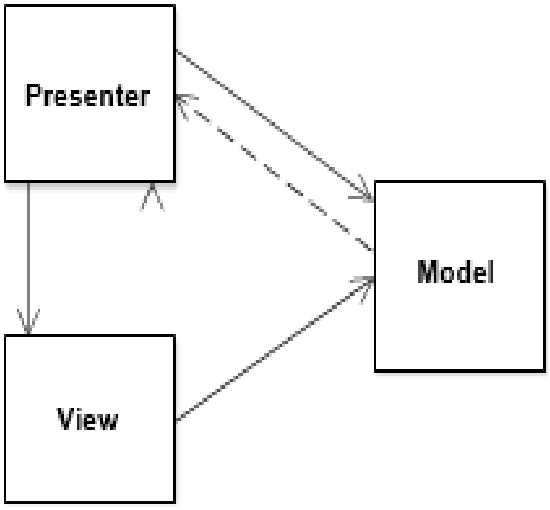

Figure 5. *Supervising Controller*

In Figure 5, the relationship between View and model is limited to data binding. The changes in Model can be reflected to view side with data binding. More complex behaviors are realized by way of Presenter [9]. To start with the Model may cause to focus on parts where user cannot see in the first place or will not interact. Bottom-up development is the case. Model can be developed without fully understanding the domain. It might be more useful if the development of the Model is postponed until having a broader idea of the functionality of the system by piling user scenarios for a while. To start with the View is often the case. As a result, user scenarios describe some functions; one can start with the view for the realization of these functions and for the user to see them in a short time and use them to provide feedback. However, in the first stage of the development process, it will lead to a focus on the user interface. That users focus on user interface more than enough will cause the user interface to change frequently and this will keep the developer team from focusing on the more important parts. Another risk is the possibility of the accumulation of too much business logic in the view layer. Also not being easily testable of the GUI interface and disrupting the process of TDD is another disadvantage.

The best starting point is the Presenter part. Development starts from the implementation of the Presenter class by selecting any of the user scenarios. User statements in user scenarios direct the structure of the method in Presenter. Therefore, Presenter methods are created by retaining user statements as much as possible in the user scenarios. This makes tracking the functional requirements demanded by users in the code easier.

When you implement Presenter class, mock objects are created from the interfaces corresponding to the model and view class required by Presenter. Thus, behaviors in model and view interfaces will be shaped as user scenarios are implemented. After unit tests corresponding to the scenarios are completed, actual classes that correspond to the model and view are implemented and user scenario becomes fully operational.

3.3. Presenter First

In this way, the development of applications involving especially GUI is called Presenter First approach. In GUI applications any application behavior is often triggered by a user action [10]. Thus, when the save button is clicked in user scenarios, statements like “when selected a record from the query

results", "when deleted a record" are the key statements in this approach. They indicate what the methods in the present classes should do and which model and viewer interface they will interact.

The operations that users perform on GUI trigger a number of events. These events are dealt by Presenter and the necessary action is implemented and as a result some changes and results in the GUI side are reflected to the user. These events come out of view classes. Presenter objects come into play when they are informed of these events. Communication from view class to Presenter is always carried out over the events. The resulting changes when Presenter comes into play is again reflected to the GUI through the methods offered by the view class. There is no connection between View and Model classes. Any changes in View are transmitted to Presenter through events. These changes are reflected to model by Presenter. Likewise, any changes in Model are transmitted to Presenter through events. Necessary changes are reflected to View over Presenter again.

Necessary behavior for View and model classes will emerge spontaneously with the development of Presenter class. These interfaces serve as specification for user scenarios. After View interfaces come out, feedback from users can be taken by developing views. The only task of the view classes is to promptly notify Presenter of any changes [11].

Apart from that, any behavior in View classes is out of the question. View classes, therefore, do not have any other functionality other than bringing together and rendering GUI components.

3.4. Presenter

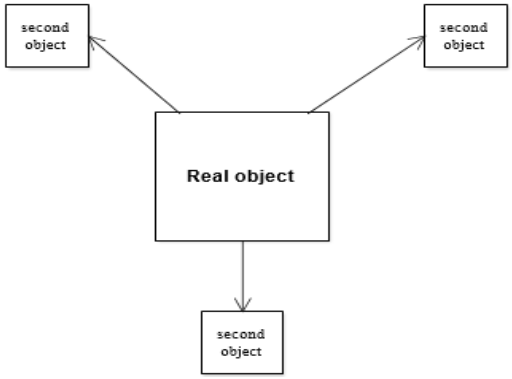

Figure 6. Presenter structure

Presenter First approach also makes it much easier to implement the practice of TDD in developing applications. In this approach, mock derivatives of view, model and other needed service components are created and given to Presenter object. In this way, Presenter can be developed independently of the view, the model and the service layer.

In TDD practice, in the creation of unit tests of original object, two approaches are generally used [12]. These are:

- Interaction-based approach
- State-based approach

In TDD practice, other objects needed for the actual object exposed to unit test to run are called secondary objects. In interaction-based approach, it is checked whether behavioral methods tested on secondary mock objects are called by actual object in appropriate number and way. There might be lots of reasons of the creation of mock derivatives of secondary objects.

- Real implementations may not be ready.
- Even though they are ready, the creation and operation in test environment may be difficult or may run very slowly. It may be related to the network or file system.
- GUI connection may be the case.
- Due to these and similar reasons, instead of actual secondary objects fake ones are used. These are called "mock" objects.

In the second approach, it is checked whether primary and secondary objects reflect the true state values after the related behavior [13]. In this approach, usually actual objects themselves are used not fake object derivatives as secondary objects.

3.5. Mediator

Developing a more improved user interface by integrating different view-presenter-model trio forms is the general logic in developing GUI-based applications.

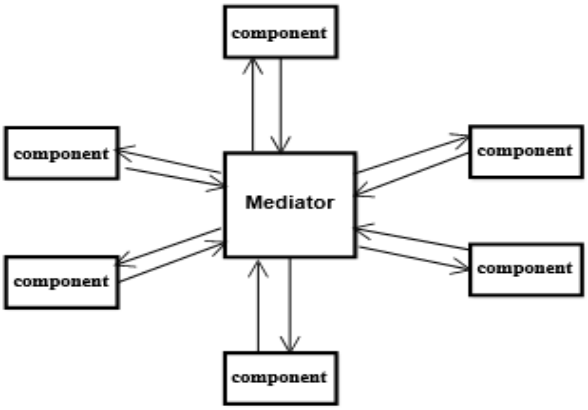

Figure 7. Mediator structure

At this stage it arises the need for different components to communicate with each other [14]. This requirement results in a common architectural problem that different components become dependent on each other. Mediator, in a sense, can be likened to a communication of a group of people over messenger. A member in the group uses mediator to send a message to other group members. Message is transmitted to other group members via mediator [15]. There is not a direct relation or connection between group members. Group members are not aware of the people communicating over messenger at the same time.

After mediator, communication network between components change into a structure as shown in Figure 7. This paves the way that components are reused in the same application or different applications.

4. EXPERIMENTAL RESULTS

The general aim of this study is to analyze the performance difference between the application developed using Model-View-Controller and the application developed using Model-View-Presenter. Steps to be followed in the system is as follows:

- Both applications have been written using Java programming language according to object- oriented software principles [16].
- Creating requirements report for applications: In this application, one usecase has been determined. First of all, steps for usecases have been identified. [17].

For the second application, Class diagrams have been drawn [18].

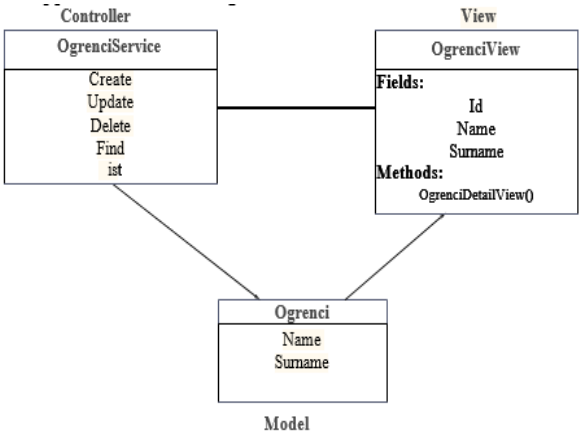

Figure 8. Class diagram of MVC application

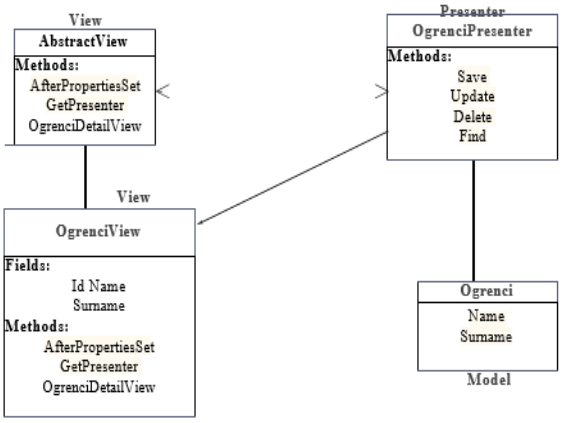

Figure 9. Class diagram of MVP application

- Applications have been coded considering class diagrams drawn for the second application.
- Comparing two applications by the digital data: Digital data have been obtained as a result of performance measurement of applications. The following tool has been used to obtain digital data.

Comparing the performance of two applications by digital data, performance measurements of applications have been made and digital data have been obtained. Jmeter testing tool has been used to obtain these digital data. Jmeter is a software which is used to test and measure performance and to make a graphical analysis of performance [19].

By using Jmeter, at a specified usecase, same operation has been done by one, fifty, one hundred and fifty users and average time values have been displayed. By obtaining average time values graphical comparison has been made as follows. The same scenario has enabled us to achieve the following data for different number of users.

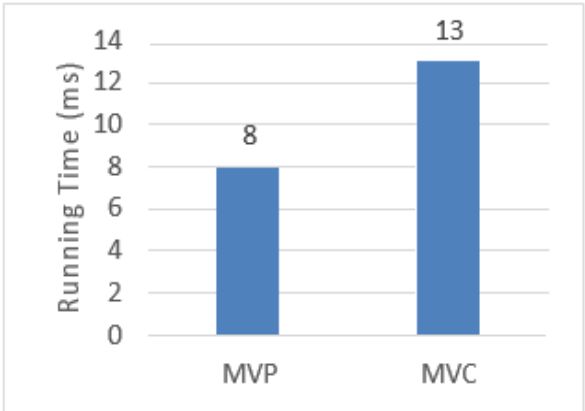

Figure 10. The graph of data obtained by using 1 user

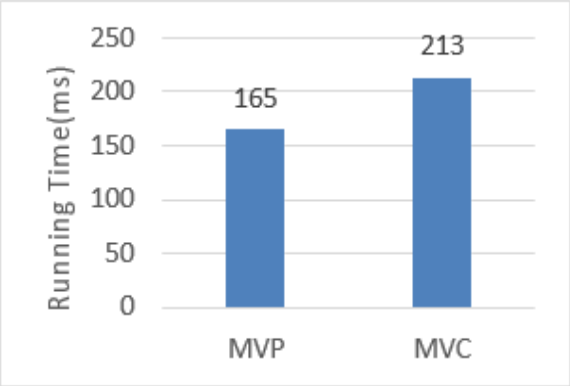

Figure 11. The graph of data obtained by using 50 users

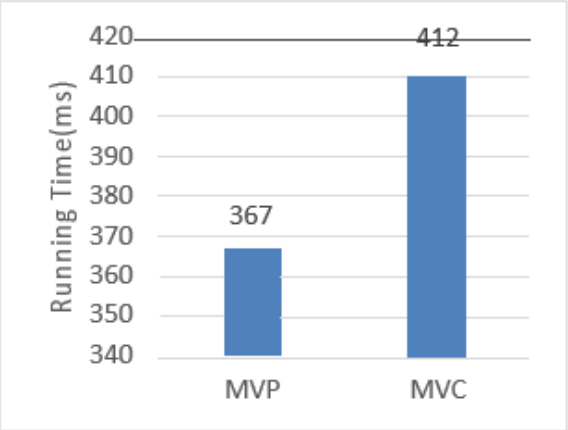

Figure 12. The graph of data obtained by using 250 users

When we examine the above data, we reach the conclusion that the application written using MVP is easier to test than the application written using MVC and the waiting period is less, when user carry out operations.

Even if we have changed the number of tests, the number of users or scenarios, the performance results between two applications, we have seen few changes and we have achieved results similar to the above data.

5. CONCLUSION

As a result of data with the same scenarios with 1, 50 and 250 users, we have observed that the application written using MVP is easier to test than the application written using MVC. MVC pattern separates the system functionally from each other. However, it shows how user interactions are changed into functional behavior. MVP allows developers to address functional behaviors and the display of user interface independently.

Thanks to the MVP approach, developers focus more on functionality rather than thinking about GUI components. Developing a fully testable form of behavior becomes possible. Developing interface and business logic can be completely separated from each other and can be carried out by different teams. It becomes possible to develop components that need communication between Mediator and each other modularly and independently.

Since functionality is under control with unit tests, problems resulting from any changes made to the user scenario are detected early. Changes in the user interface can be performed much more easily and safely. After all, changes to be made here is known to have no effect on the function in any way.

6. REFERENCES

[1] Freeman, E., Freeman, E., Bates, B. ve Sierra, K. (2004). Head First Design Patterns, O'Reilly. Available: http://www.sws.bfh.ch/~amrhein/ADP/HeadFirstDesignPatterns.pdf

[2] Reenskaug, T. "The Model-View-Controller (MVC) Its Past and Present", 2003, University of Oslo. Available: https://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf

[3] Potel, M. "MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java", 1996, Taligent. Available: http://www.wildcrest.com/Potel/Portfolio/mvp.pdf [4] Reenskaug, MVC XEROX PARC 1978-79 Available: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

[5] Geoffroy Warin, Mastering Spring MVC 4, Packt Publishing - ebooks Account, 2015

[6]Yang Zhang, Yanjing Luo, "An Architecture and Implement Model for Model-View-Presenter Pattern", Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on, Chengdu, 9-11 July 2010, Volume: 8, pp. 532 - 536

[7] Andy Bower, Blair McGlashan, "Twisting the Triad: The evolution of the Dolphin Smalltalk MVP application framework", European Smalltalk User Group (ESUG), 2000.

[8] M.Fowler, "Passive View " Available: http://www.martinfowler.com/eaaDev/PassiveScreen.html, 1996 [9] M. Fowler, "Supervising Controller" Available:http://www.martinfowler.com/eaaDev/SupervisingPresent, 2006

[10]M. Marsiglia,B. Harleton, C. Erickson, "Presenter First: TDD for Large, Complex Applications with Graphical User Interfaces"

[11] http://en.wikipedia.org/wiki/Presenter_First,2010

[12]M. Alles,D. Crosby,Presenter First: Organizing Complex GUI Applications for Test-Driven Development Available: https://atomicobject.com/uploadedImages/archive/files/PF_March2005.pdf [13] J Halife, "HOW-TO: WRITE MVP USING TDD" Available:http://blogs.southworks.net/jhalife/2006/09/01/how-to-write-mvp-using-tdd/, 2006 [14] Bevis, T., Java Design Pattern Essentials, Ability Firs. 2012.

[15] Freeman, A. Pro Design Patterns in C#, Apress. , 2015. [16]R. C. Martin, "Design Principles and

Design Patterns", 2002.
[17]Bennett, S., McRobb, S. ve Farmer, R. "Object-Oriented Systems Analysis and Design Using UML, McGraw-Hill.", 2010.
[18]Lethbridge, Timothy C. ve Laganiere, R. "Object-Oriented Software Engineering: Practical Software Development Using UML and Java", McGraw-Hill, 2001.
[19] Nevedrov, D. "Using JMeter to Performance Test Web Services" 2006. Available:http://loadstorm.com/files/Using-JMeter-to-Performance-Test-Web-Services.pdf