IJCTA, 8(4), 2015, pp. 1503-1510
© International Science Press

A Structural Design for Web Application Based on Model-view-presenter View-model (Mvpvm) Pattern

Malar P* & Agnise Kala Rani X**

Abstract: In earlier times, to design a PHP based on MVC and MVP pattern which provides better performance. But, MVC pattern PHP development design stages new levels of indirection and therefore the complexity of the solution is increased. It also lifts the event-driven nature of the user-interface code, which can become more tedious for debugging. In this research work, an improved design is proposed on the basis of the Model view presenter viewmodel (MVPVM) pattern. The new approach targets at very large-scale enterprise applications with multiple views, having different business requirements that may require binding onto the same ViewModel, indicating the ViewModel cannot contain view-specific business logic. It also shows that it is definitely performing extensive unit testing and coded UI testing of the components, hence that process has to be much efficient as possible. The proposed MVPVM pattern comprises model, view, presenter and View Model. The MVPVM pattern yields better design for PHP web applications through the separation of the data view and main controls of web applications. The vital benefits of the proposed MVPVM pattern are easy maintenance; debugging and the problems are solved more easily when compared to the conventional patterns. Consequently, an improved design is obtained.

Keywords: Model view presenter, Model view controller, PHP, Web application

1. INTRODUCTION

Owing to the speedy increase corresponding to the number of Web users over the earlier two decades, the extensive opportunities and convenient software design, and the larger demand for such applications has given rise to a considerable increase in the number of people working on web applications design [1]. For some period of time not so long ago, Web applications had just been only an add-on to some other strict system and hence design of these web applications required people with lots of experience in other fields of software. The PHP, J-Query are some of the important scripting languages that were used for web applications development.

PHP is a server-side scripting language employed for web development but also considered as a general-purpose programming language. Now, PHP is installed on more than 244 million websites and 2.1 million web servers [2,3]. PHP code is decoded by a web server with a PHP processor module, which, in turn, gives the resulting web page: PHP commands can also be embedded directly into an HTML source document instead of calling an external file for data processing. It has evolved a lot more to have a command-line interface capability and can be found to be used in separate graphical applications. But in the recent times, young keen people have begun to design Web pages with the help of scripting languages without even possessing the knowledge of even the simplest principles of software design. Several authors have introduced different techniques for minimizing the complexity. One of the important methods is the introduction of the design patterns in the PHP and the creation of a new design depending on the design patterns.

One means for solving the problem is the evolution of a web application design which is evolved for supporting the development of dynamic websites, web application and web services. The goal of the design is to mitigate the

* Ph.D Research Scholar, Karpagam Academy of Higher Education, Coimbatore – 641 021, Tamil Nadu, India.*Email: malar.prp@gmail.com*

** Professor, Department of Computer Applications, Karpagam Academy of Higher Education, Coimbatore-641 021, *Email: agneskala72@gmail.com*

overhead that is associated with the usual activities involved in web development. For instance, many designs are provided with libraries for database access, templating designs and session management, and they also support code reuse often [2]. The software designs substantially minimize the amount of time, effort, and resources that are necessary for developing and maintaining web applications. In addition, a design is an open architecture which is in accordance with few generally accepted standards [3].

MVC design pattern is considered to be an efficient method for the evolution of structured modular applications, [4]. In the form of a design pattern, usually MVC is for the purpose of splitting an application into multiple independent layers that can be evaluated, and implemented occasionally, in an isolated manner. Through the decoupling models and views, MVC facilitates decreasing the complexity in architectural design and also increases the flexibility and reuse of code. Even though flexibility and reusability are gained, it has few issues that impact PHP design like new levels of indirection and therefore the complexity of the solution is increased, and the event-driven nature of the user-interface code, can turn out to be hard for debugging, Views such as graphical displays may take up some time to yield results and in case the model undergoes changes in a frequent manner, the views could be overwhelmed with requests on update.

In order to get over the challenges and problems in the MVC and MVP design pattern, an enhanced version of design pattern referred to as Model view presenter view model (MVPVM) is utilized for the development of a new PHP design in this paper.

2. MATERIALS AND METHODS

2.1. Model view controller

The model-view-controller or MVC is a software architecture that is used generally for developing web applications or software structure. It means that, it is a structure for web applications to go behind for guaranteeing efficiency and consistency. Many of the widely-known designs make use of the MVC architecture that includes ASP.NET, CodeIgniter, Zend, Django, and Ruby on Rails. Figure 1 indicates the relationship between the Model, View, and Controller.

- Model: business logic & processing
- View: user interface (UI)
- Controller: navigation & input

In the case of MVC, the presentation layer is divided into controller and view. The most important separation is between the presentation and application logic. The split in View/Controller is less. MVC includes much of the architecture of an application than being distinguished for a design pattern. Hence the term architectural pattern may be functional, or most probably an aggregate design pattern [5, 6].

Model

The Model in MVC is the area-specific representation of the information on which the application works. Application (or domain) logic imparts significance to raw data (e.g., it is manipulative in case today is the user's birthday, or the totals, taxes and shipping charges for shopping cart items). Several applications use a persistent storage mechanism (like a database) for the storage of data. MVC does not intend to mention the resource management layer as it is understood to be beneath or enclosed by the Model.

View

Yields the model into a form suitable for interaction, View is basically a user interface element. MVC is often observed in web applications, in which the view is the HTML page and the code that collects dynamic data for the page.

Controller

Does the processing and then responds to events, generally user actions and may react to variations on the model and view. The control flow generally works in the following manner [8, 9]:

1. The user interacts with the user interface in multiple ways (e.g., user presses a button)
2. A controller gets the input event from the user interface, often by means of a registered handler or callback.
3. The controller operates on the model, possibly updating it in a way that suits the user's action (e.g., controller does the updation over user's shopping cart). Complex controllers are prepared repeatedly through the command pattern in order to encapsulate actions and have a simplification of the extension.
4. A view makes use of the model to develop an appropriate user interface (e.g., view generates a screen that lists the contents of the shopping cart). The view collects its own data from the model. The model has no direct information about the view.
5. The user interface remains for the next user interactions, which starts the cycle.

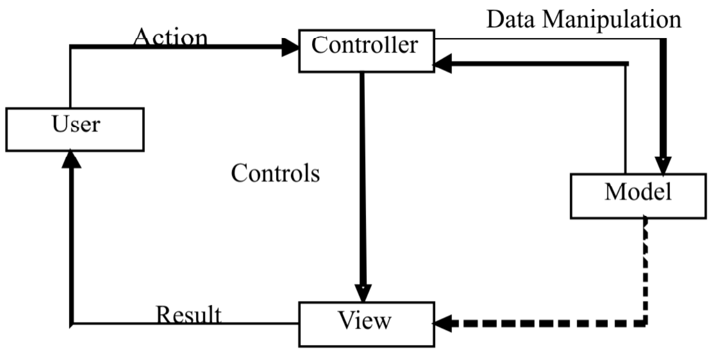


Figure 1: Model view controller architecture

Disadvantages of MVC pattern

- The MVC pattern exhibits new levels of indirection and hence results in increase in the complexity of the solution.
- In case of the model undergoing changes frequently, the views could be loaded with update requests.
- Views such as graphical displays may use up some time to produce the result. Consequently, the view may be falling behind update requests.
- Unit testing cannot be conducted.

2.2. Model View Presenter

The MVP is derived from MVC software pattern that is useful for the structuring of user interfaces. The MVP design pattern separates the view from its presentation logic to allow each to be distinguished separately [10]. In MVP, the view goes on to become an ultra-thin component whose function is just to be able to offer a presentation to the user. The view captures and deals with the events increase by the user, though forwarding these directly to the presenter who is aware on the means of treating them. The presenter then communicates with the model, and coordinates with the view's controls directly so as to present the data.

The Model

This is the data on which the user interface will work upon. It is generally a domain object and the goal is that such objects are supposed to have no idea of the user interface. Here the M in MVP is different from the M in MVC. As discussed previously, it is, in fact, an Application Model, which touches onto multiple aspects of the domain data but also does the implementation of the user interface to make an impact over it. In MVP, the model is merely a domain object and there is no anticipation of (or link to) the user interface at all.

The View

The performance of a view in MVP is similar to that of MVC. It is the view's obligation to have to display the contents of a model. The model is then projected to initiate appropriate change notifications every time its data is changed and these allow the view to "hang off" the model next to the standard Observer pattern. Similar to the way as MVC does, this permits multiple views to be linked to a single model [9].

One significant difference in MVP is the elimination of the controller. As an alternate, the view is supposed to manage the raw user interface events that are generated by the operating system (in Windows these are observed as WM_xxxx messages) and like this kind of operation suits more apparently into the style of many of the recent operating systems. In some cases, like a Text View, the user input is held directly by the view and utilized to make modifications to the model data. However, in most cases the user input events are generally routed through the presenter and this is responsible for the way the model gets modified 5.

The Presenter

When it is the responsibility of the view to display model data, it is actually the presenter that controls how the model can be manipulated and hence can be varied by the user interface. In several ways, a MVP presenter is based on the application model in MVC; most of the code starts with how a user interface works which is built into a presenter class. The important difference is that a presenter is directly connected to its associated view such that the two can closely coordinate in their roles of providing the user interface for a specific model.

Benefits of MVP

Therefore the impact of this "twist" is that the presenter, which is the data manipulation part of a user interface, is also permitted direct access to the view, where the data display is realized. This can be very convenient at times and is one of the most apparent advantages over MVC where the application model simply has an indirect link to its associated view. The Dolphin implementation of MVP also handles the dispensing of the idea of a controller, which is aimed at making the design "fit" better along with the Windows operating system underneath.

When compared with the actual widget design, MVP provides a much greater isolation between the visual presentation of an interface and the code necessary for implementing the interface functionality. The latter lives in one or more presenter classes which are coded as normal making use of a standard class browser. The window layouts for several applications are created by a tool called the View Composer which is again utilized for creating an instance of the view required 6. These view instances reside in an internal binary form by a Resource Manager.

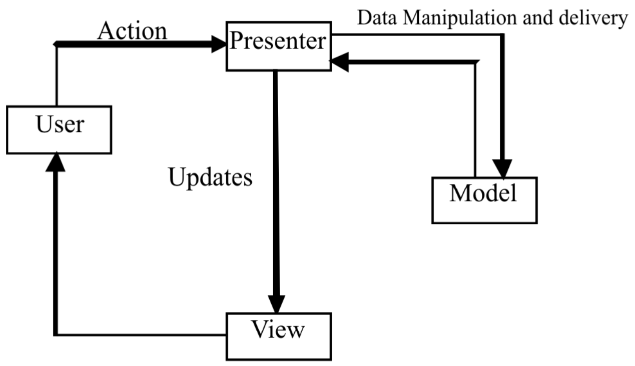


Figure 2: Model view presenter architecture

Generally, one or more view instances can get associated with any presenter class and a presenter can define which specific view is needed when it is launched. Therefore it is easier for an MVP application to be with one or more "skins" that can be chosen as necessary.

Disadvantages

The disadvantages pertaining to Model-View-Presenter are identical to the disadvantages of Model-View-Controller as below:

- The pattern is complicated and may be not really necessary for simple screens.
- The pattern is one more thing to be learnt for busy developers: there's surely an overhead.
- It can be difficult to debug events that are being fired in active Model-View-Presenter.
- The 'Passive View' version of Model-View-Presenter can result in a particular amount of boilerplate code that has to be written in order to get the interface into the View to work.

3. PROPOSED METHODOLOGY

The definitions of the MVPVM components that follow have extracts from "Twisting the Triad" as and when applicable. The proposed Model-View-Presenter View-Model (MVPVM) Pattern for PHP design is illustrated in figure 3. In this new design the View model concept is presented, which is about not making reuse of the views. It

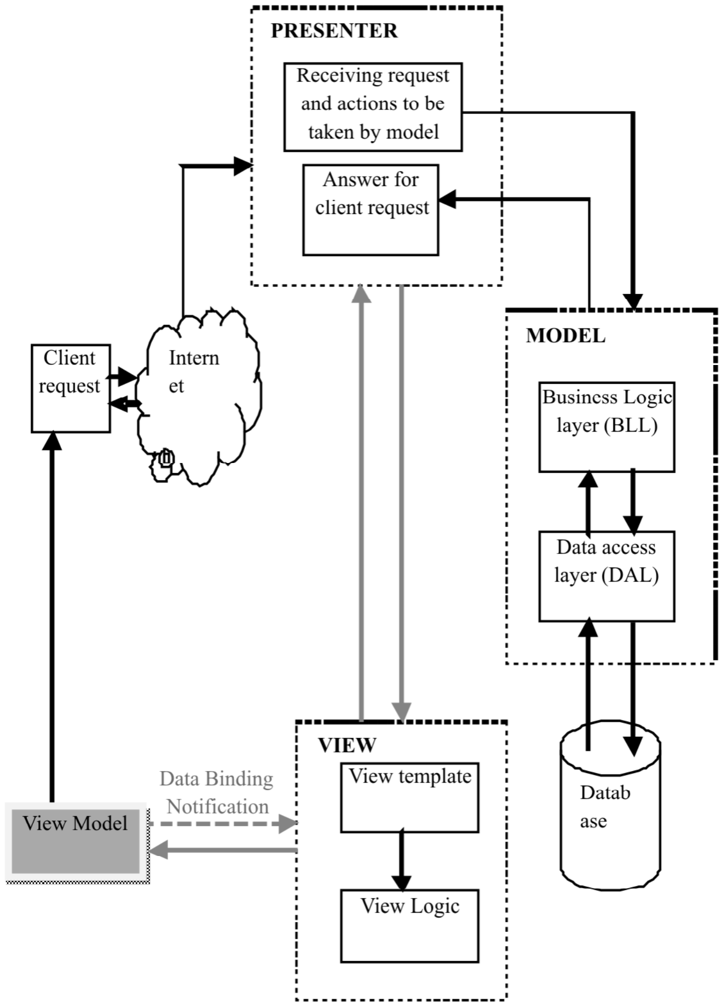

PRESENTER
Receiving request and actions to be taken by model
Answer for client request
MODEL
Business Logic layer (BLL)
Data access layer (DAL)
Database
Client request
Internet
VIEW
View template
View Logic
View Model
Data Binding Notification

Figure 3: Proposed design using MVPVM pattern

assists in reducing the scalability issues etc. The View is held accountable for generating the user interface; it can see the View Model, initiate its methods and can make modifications to its properties whenever necessary. View preserves a one-way reference to the View Model. When a View Model property is modified, View is notified by means of observer synchronization. On the contrary, when a user interacts with the View, View Model properties are modified directly. View Model holds the responsibility for managing the view state and user interaction; it possesses access to the domain Model, so that it can work with domain data and summon business logic. View Model has no knowledge of View. Model is accountable for dealing with the domain data and quite has no awareness about the View Model. This approach permits the creation of various different views for the same data, and observer synchronization renders these views to operate simultaneously. For instance, let the previous case of the financial report be extended. The new target is providing a user two views of the report: a table and a pie chart. When data is altered in the Model, the View Model receives the notification. The View Model then invokes the Model for getting the data required (e.g. the list of objects describing the subject and an amount) and sets the data in a property Data. Two Views (one for the table and one for the pie chart) see the View Model and have updations made to their representations when the Data is modified. Each View has its own logic on how the data has to be presented. The first view generates a table with columns, subject and amount; the second makes use of the amount to plot a pie chart and the subject to have a legend.

MVPVM: The Model

This is the data the user interface works upon. It is generally a domain object and the intent is that such objects should have no information regarding the user interface. The isolation of the Model from the View Model is necessary for addressing the concerns related to dependency injection and its usage within the Business Logic Layer (BLL) and Data Access Layer (DAL) to Create, Read, Update, Delete and List (CRUDL) persisted data. Only the DAL has access to the persisted Model objects in MVPVM.

MVPVM: The View

"The behavior of a view in MVP is much similar to that in MVC. It is the responsibility of the view to display the contents of a model. The model is supposed to do the triggering of the suitable change notifications whenever its data is changed and these permit the view to 'hang off' the model adhering to the standard Observer pattern. In the identical way as MVC does, this permits multiple views to be linked to a single model.

It has to be emphasized that MVP is not entirely a new pattern and that it is valid today also as it was MVP springed from MVC. But, the quote references the "Model," while MVPVM makes use of the "ViewModel".

With MVPVM, there's never a necessity to have code in the code that is behind. The Presenter has access to the View and can sign up for notification on events, manipulate controls and the UI as needed. This was advantageous when developing the multi-targeted application that is associated with this article.

A View has no information about the ViewModel, hence they aren't tightly coupled. Until a ViewModel provides support to all of the properties to which a View is bound, it can conveniently make use of the View. The Presenter has the responsibility for wiring up Views to their View Models by fixing their Data Context to the View Model that is applicable.

MVPVM: The Presenter

"While the view is responsible for displaying model data, it is the presenter that tells how the model can be influenced and modified by the user interface. This is where the heart of an application's behavior is residing. In several ways, an MVP presenter equals the application model in MVC; most of the code that deals with the means by which a user interface operates is stitched into a presenter class. The important difference is that a presenter is directly connected to its associated view such that the two can closely coordinate in their roles of rendering the user interface for a specific model." —"Twisting the Triad".

The Presenter will be dependent on the interfaces for the BLLs from which it has to recollect domain objects (data). It will make use of the resolved instances as it is configured in the module or bootstrapper in order to access the data and fill up the ViewModel. Generrally, only the Presenter will be tightly coupled to the components of MVPVM. It will be tightly coupled to the View, ViewModel and the BLL inter faces. Presenters aren't supposed to be reused; they deal with particular concerns along with the business logic/rules related to those concerns. In the cases where a Presenter can be reused across enterprise applications, there are huge chances that a module would be more appropriate for the task—that is, a login module (project) could be created and that could be reused by all of the enterprise applications. If it is coded against an interface, the module can be easily reused employing dependency injection technologies like the MEF or Unity.

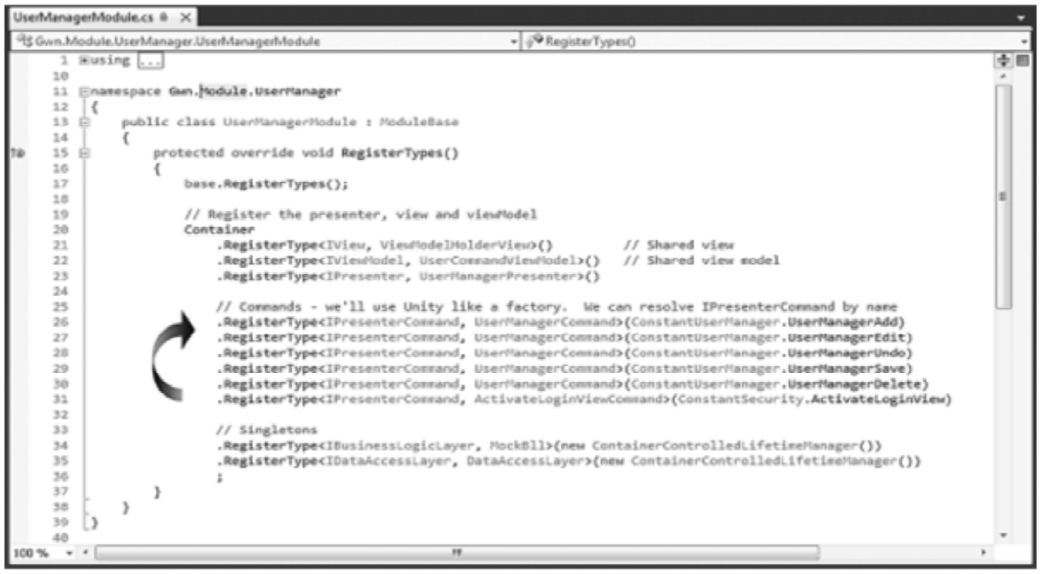
MVPVM: The View Model

"In MVP, the model is truly a domain object and there is no anticipation of (or link to) the user interface at all. This avoids the View Model from being tightly coupled to a single View, allowing it to be reused by several Views. In a similar manner, the View Model will possess no application business logic, hence it is easy to share View Models across enterprise applications. This supports reuse and application integration. For instance, the Security Command View Model is in the Gwn. Library. Mvp Vm.xxxx project (where xxxx = Silverlight, desktop or Windows Phone). Since a User View Model will be needed in most of the applications, this is a reusable component. Caution has to be taken so as not to pollute it with business logic that is specific to the demo application. This is not an issue with MVPVM, as the business logic will be managed by the Presenter, and not within the View Model (shown in Figure 3).

The Business Logic Layer

BLLs have no information regarding the persisted Model data. Their behavior is strict with respect to domain objects and interfaces. Generally, dependency injection can be applied in order to resolve the DAL interface within the BLL, so it is possible to swap out the DAL later without impacting any downstream code., A MockBll implementation for IBusinessLogicLayer for the demo application is to be noted in Figure 5, line 34. Later it can be easily substituted with an implementation during production of the interface with a single line of code since it is developed against an interface.

Business logic isn't restricted to the BLLs. In Figure 4, the register named the types (for IPresenter Command) in such a way that it can make use of dependency injection as a factory. When the user clicks on a button on the



```
1  using ...
10
11 namespace Gwn.Module.UserManager
12 {
13     public class UserManagerModule : ModuleBase
14     {
15         protected override void RegisterTypes()
16         {
17             base.RegisterTypes();
18
19             // Register the presenter, view and viewModel
20             Container
21                 .RegisterType<IView, ViewModelHolderView>()          // Shared view
22                 .RegisterType<IViewModel, UserCommandViewModel>()   // Shared view model
23                 .RegisterType<IPresenter, UserManagerPresenter>()
24
25                 // Commands - we'll use Unity like a factory.  We can resolve IPresenterCommand by name
26                 .RegisterType<IPresenterCommand, UserManagerCommand>(ConstantUserManager.UserManagerAdd)
27                 .RegisterType<IPresenterCommand, UserManagerCommand>(ConstantUserManager.UserManagerEdit)
28                 .RegisterType<IPresenterCommand, UserManagerCommand>(ConstantUserManager.UserManagerUndo)
29                 .RegisterType<IPresenterCommand, UserManagerCommand>(ConstantUserManager.UserManagerSave)
30                 .RegisterType<IPresenterCommand, UserManagerCommand>(ConstantUserManager.UserManagerDelete)
31                 .RegisterType<IPresenterCommand, ActivateLoginViewCommand>(ConstantSecurity.ActivateLoginView)
32
33                 // Singletons
34                 .RegisterType<IBusinessLogicLayer, MockBll>(new ContainerControlledLifetimeManager())
35                 .RegisterType<IDataAccessLayer, DataAccessLayer>(new ContainerControlledLifetimeManager())
36                 ;
37         }
38     }
39 }
40
```

Figure 4: Registering BLL, DAL and Commands

command parameter it is taken care of (instantiated) by the base class and the relevant command is executed. All that was necessary for wiring this up was a Button command in XAML and an entry in the SecurityModule [12, 13].

The Data Access Layer

Persisted domain data could be maintained in SQL databases or XML or text files, or returned from a REST service. With MVPVM, only the DAL contains specific information that is necessary for retrieving the data. The DAL will only retrieve domain objects to the BLL. This prevents the necessity for the BLL to be knowledgeable about the connection strings, file handles, REST services and so on. This improves the scalability and extensibility; DAL can be switched from an XML file to a cloud service without any impact over any existing code residing out of the DAL and the application configuration file. Until the new DAL unit tests work fine for the CRUDL processes, the application can be configured in order to utilize the new DAL without any impact over current applications and their utilization.

4. CONCLUSION

The PHP is a highly efficient language for developing dynamic and interactive web applications. One among the defining characteristics of PHP is the ease with which developers can connect and manipulate a database. PHP aids the preparation of the functions for manipulation of the database. But, database management is performed by the Structure Query Language (SQL). Many new programmers frequently have trouble with SQL syntax. In this paper, the PHP design is introduced for database management on the basis of the MVPVM pattern. The MVPVM pattern is very helpful for the architecture of web applications, isolating the model, view, presenter and Modelview of a web application. The PHP design encapsulates general database operations which are INSERT, UPDATE, DELETE and SELECT. The user or programmer is capable of easily programming web application software projects or projects of their own by means of customization in the development. This PHP design takes the best advantages of its loose coupling, expansibility, and reusable quality.

References

[1] Andris Paikens, Guntis Arnicans "Use of Design Patterns in PHP-Based Web Application Frameworks" LATVIJAS UNIVERSITATES RAKSTI. (2008), 733.

[2] Gamrat B., "PHP and preprocessed Web pages," Dr. Dobb'S Journal, January (2006), 31(1), pp. 46-48.

[3] Stig Saether Bakken, Alexander Aublach, Egnn Schmid et al., "PHP Manual," The PHP Documentation Group, (2007).

[4] Hofmeister C., Nord R.L., Soni D., "Applied Software Architecture,". Addison Wesleyÿ (2000) ÿ

[5] Song Wei, "Approach to Web application program based on MVC and J2EE," Hebei Journal of Industrial Science & Technology, July (2005) ,22(4), pp. 189-191.

[6] Ni L.P. Pravina Utpatadevi, A. A. K. Oka Sudana, A. A. Kt. Agung Cahyawan "Implementation of MVC (Model-View-Controller) Architectural to Academic Management Information System with Android Platform Base" International Journal of Computer Applications (0975 – 8887) Volume 57– No.8, November 2012.

[7] Zamah Sari, Moechammad Sarosa, Heru Nurwasito "Concept of Designing an Optimized Pull Model View Controller Type Content Management Framework" International Journal of Computer Applications, (2012).

[8] Nelly Delessy-Gassant, Eduardo B. Fernandez "The Secure MVC pattern" International Symposium on Software Architecture and Patterns, July 23-27, (2012).

[9] R.Anusha, Ch.Sivaramamohana Rao "Secured N-Tier Attack Detection and Prevention Mechanism" International Journal of Computer Trends and Technology (IJCTT) – volume 4 Issue 8–August (2013).

[10] Andy Bower "Twisting The Triad" Camp Smalltalk 1, San Diego, March 2000 http://aviadezra.blogspot.in/2007/07/twisting-mvp-triad-say-hello-to-mvpc.html

[11] Veglis A., Leclercq M., Quema V., "PHP and SQL made simple, Distributed Systems Online," Aug 2005, 6(8), pp. 15-22.

[12] http://www.codeproject.com/Articles/42830/Model-View-Controller-Model-View-Presenter-and-Mod.

[13] Bill Kratochvil, The Model-ViewPresenter-ViewModel Design Pattern for WPF, MVPVM DESIGN PATTERN, 2011.