INFOTEH-JAHORINA Vol. 12, March 2013.

Groundwork for Presentation Pattern Metamodels

Nina Turajlić, Marko Petrović, Milica Vučković, Ivana Dragović
Faculty of Organizational Sciences
University of Belgrade
Belgrade, Serbia
{niiv, mpetrovic, milica, pavlovici}@fon.rs

Abstract—**A number of Presentation Patterns exist providing common solutions for the design and implementation of the presentation layer of a business application. The differences among them stem from the manner in which the concerns, associated with the functionality of the presentation layer, are separated into components, as well as the interaction between these components. By identifying the set of concepts comprising a given Presentation Pattern the pattern's metamodel can be defined. This paper presents an analysis of the most frequently applied Presentation Patterns, with the intention of setting the grounds for future automated development of the presentation layer of a business application through the transformation of the chosen pattern, defined as a Platform Independent Model, into an implementation of that pattern on the chosen development platform - a Platform Specific Model.**

Keywords-Pattern; Presentation Layer; Metamodel

I. INTRODUCTION

Patterns provide common solutions for the design and implementation of business applications. In other words, they enable the use of collective knowledge and experience for solving certain classes of problems. On the other hand, the Separation of Concerns principle states that an application should be designed in a manner that results in the minimal overlapping in the functionality of its components. The goal is to design the application in such a way that each of its components can be replaced without affecting the others. First the principle is applied to the design of the application itself, which is consequently, developed as a multi-tier application comprised of a presentation layer, a business layer, a data layer etc. Subsequently, the principle should be applied to each of the layers. In object-oriented software development a large number of patterns have been introduced e.g. design patterns such as GRASP and Gang of Four (GoF) patterns, standard software architectures (e.g. the well-known three - tier architecture - a combination of the Model-View-Controller MVC and Persistent Broker patterns), business patterns, implementation patterns…, all of which have incorporated the high cohesion and loose coupling principles into the solution they provide. Coupling is the degree of dependency of one element (class, component, subsystem, etc.) on other elements of the system. Loose coupling enables better readability, facilitates maintenance and increases the possibility of component reuse and thus is one of the goals of OO software design. Cohesion is primarily related to the functional cohesion of an element and indicates how tightly related the responsibilities of an element are. High cohesion basically means that an element executes one or more sequentially related functions. Low cohesion results in a tight coupling of elements.

One of the main issues with patterns is their automation. The usual informal descriptions of patterns have proven to be effective at communicating design experience to developers, but they lack the formality needed to support rigorous use of design patterns. Precise specification of pattern solutions enables the development of pattern-based development techniques and supporting tools that can be used to (1) systematically build solutions from pattern specifications, (2) verify the presence of pattern solutions in designs, and (3) systematically incorporate a pattern solution into a design [1]. This limitation poses a serious problem for automated software development. The Model Driven Architecture [2] defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality for a specific technology platform. The key elements of the MDA are models. A model is defined as a representation of a part of the structure and/or behavior of a system. The main goal of Model Driven Development is to automate software development through the successive application of model transformations, starting from the model representing the specification of the system and ending in a model representing the detailed description of the physical realization, from which the executable code can ultimately be generated. To this end, two types of models are defined: Platform Independent Models (PIM) and Platform Specific Models (PSM). The PIM provide formal specifications of the structure and function of the system that abstracts away technical details, while PSM is expressed in terms of the specification model for the target platform. How the functionality defined in a PIM is realized depends on the chosen platform and is specified in a PSM, which is derived from the PIM via some transformation.

Presentation Patterns (PP) provide common solutions for problems inherent to the presentation layer and the main goal of most PPs, in accordance with the Separation of Concerns principle, is the clear separation of the code which renders the user interface and accepts user interaction from the code responsible for business logic and state management.

This paper presents a continuation of the work presented in [3] in which a review of the most frequently cited patterns for presentation layer design was given to facilitate the selection of the best-suited pattern for a given problem. The differences among the patterns are due to different responsibilities being assigned to different components as well as the various means of interaction between these components.

The intention of this paper is to identify and clarify the set of concepts comprising different Presentation Patterns in order to enable the defining of their metamodels describing both the structure and behavior of the corresponding PPs. In this way a vocabulary (a set of concepts including their syntax and semantics) would be defined which could be used in software design for creating the appropriate PIM. Subsequently, the development could be automated by transforming a PIM of a chosen PP to a PSM, an actual implementation of the Pattern for a chosen platform.

The paper is structured as follows: In Section II a selection of Presentation Patterns will be presented. In Section III, a comparative analysis of the selected patterns will be given. In addition to conclusions, in Section IV further steps will be proposed towards the automated generation of the presentation layer for different platforms.

II. PRESENTATION PATTERNS

The presentation layer typically obtains the necessary domain data and presents it in a user interface (the user interface is comprised of controls which display the data). The user can then view and change the data. Once the user changes the data, these changes are processed and then forwarded through the application to the data storage and can, in turn, bring about the modification of the user interface controls.

The first idea that comes to mind is to tie the data sources with the user interface (UI) to minimize coding and enhance the performance of the application. This would result with the entire presentation logic, controlling the outlined process, being located in the user interface. However, this approach entails certain problems, which were given in detail in [3]. In order to avoid or at least minimize these problems the basic idea of the PPs is the clear separation of the concerns associated with the functionality of the presentation layer into components. The common goal is to separate the presentation functionality from the component that encapsulates the business and data manipulation logic (Model).

What is common to all of the considered PPs is that the *Model* manages the behavior, state and data of the application domain [4]. It encapsulates both data and business logic, i.e. application logic for managing the data. Thus the Model is not just a collection of data related to a certain concept, it also encapsulates business logic (i.e. application logic for managing the data) which implements the business rules associated with that concept. The Model has no knowledge of how it will be displayed or updated. The rationale behind the extraction of the Model into a separate component is the enabling of its reuse, thereby eliminating the duplication of code. Moreover, the Model could then support different presentations of the same data.

However, the further separation of the presentation concerns into a component that encapsulates the UI and a component that encapsulates the presentation logic (which is independent of the actual implementation of the UI) can be accomplished in several ways. According to [5] the main concerns of the presentation layer are related to: State (represents the current data picture of the UI), Logic (represents the behavior associated with the presentation of the data as well as the manipulation of that presentation) and Synchronization (the data presented in the UI should correspond to the data in the domain model. Therefore, part of the presentation functionality is the synchronization of data between its components and the domain model).

Correspondingly, PPs differ depending on which of these concerns is associated with which component as well as the different manners of interaction between these components.

The PPs discussed in this paper are: Autonomous View, Model View Controller, Model View Presenter (Supervising Presenter and Passive View) and Presentation Model.

A. *Autonomous View Pattern*

A *View*, in the most general sense, is a collection of controls of a user interface, while it can, sometimes, also include behavior.

The Autonomous View Pattern is one of the most simple presentation layer patterns. The presentation logic is directly implemented in the View. The Autonomous Views manage their states and communicate with each other when necessary. Therefore, the state and logic are in the View. Even business logic can be implemented in the View. While the advantage of this pattern is its simplicity, the disadvantage is that it can result in code that is difficult to read, maintain, build upon, and especially test, since all of the functionalities are in the same View. This is particularly true for complex Views.

All of the following PPs attempt to overcome the limitations of this approach. They adopt the Humble View Philosophy which stipulates that the user interface should be as simple as possible and that to that end the logic associated with the behavior of the user interface (i.e. the presentation logic) should be relocated from the View into other non-visual components. A Humble View should be the smallest possible wrapper around the actual presentation code. The view is also "passive," meaning that it doesn't really take any actions on its own without some sort of stimulus from outside the view. The view simply relays user input events to somewhere else with little or no interpretation [6].

B. *Model View Controller Pattern (MVC)*

COMPONENTS: This pattern proposes that the functionality of the user interface be split into: View and Controller. *View* is responsible for the rendering the elements of the user interface to display the data contained in the Model and should contain as little logic as possible. *Controller* is responsible for reacting to user actions, processing them and then forwarding the data changes to the Model and/or View.

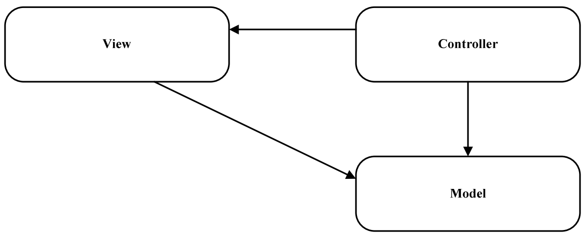

Figure 1. MVC Pattern

INTERACTION (Fig. 1): The View is responsible for the state and the Controller is responsible for the logic. Together they are responsible for interacting with the Model. Each View is associated with only one Controller and vice versa. The

Controller and View communicate via the Model and do not reference each other (both reference the Model). On the other hand, the Controller connects the Model and the View. The Model may be updated by the View, the Controller or by other components in the system. Once the state of the Model changes it fires an event to notify the View and Controller. Therefore the View and the Controller must observe the Model. Once the user inputs the data, it is necessary to first determine the appropriate Controller since a View-Controller pair exists for every element in the user interface. Then the request is passed to that Controller and it determines how to process the request. Some user actions may require updates in the Model, while other may require visual changes to the View. If the View should be updated, the Controller notifies the Model and since the View is observing the Model it will update and render itself accordingly.

VARIATIONS: Passive Model and Active Model [7]. In the *Passive Model* one Controller exclusively manipulates the Model. The Controller updates the Model and then informs the View to refresh. The Model is completely independent from both the View and the Controller so the Model cannot inform them when its state changes. In the *Active Model* the state of the Model changes independently of the Controller, for instance when other sources change the data and these changes should be reflected in the View. Once the state of the Model changes the Model notifies the View to refresh. This would, however, make the Model dependent of the View, but it could be avoided by employing the Observer pattern.

The original MVC pattern is rarely used nowadays, though variations of it have been developed in accordance with new development platforms (e.g. the Front Controller Pattern for web applications). Another variation of this pattern is the following MVP pattern.

C. *Model View Presenter Pattern (MVP)*

COMPONENTS: This pattern proposes that the functionality of the user interface be split into: View and Presenter. *View* represents the structure of the controls in the user interface, and is responsible for the presentation as well as for maintaining its state. The View should contain as little logic as possible, that is to say, it should not contain any behavior concerning the reaction to the user actions. *Presenter* contains the logic for reacting to user actions and is responsible for the behavior.

INTERACTION: The View manages the controls of the user interface. Consequent to a user action the View transfers control to the Presenter. The Presenter will then decide how to react to that action. There are several ways in which the View can forward the user actions to the Presenter: (1) The View can contain a reference to a Presenter and when the user performs an action the View merely invokes a method of that Presenter. This requires implementing additional methods in the Presenter and results with the tight coupling of the View and the Presenter, but on the flip side the code is easier to read and follow (ideally the behavior of the View should be clear just by looking at the Presenter); (2) The View raises events when the user performs an action and the Presenter subscribes to these events. The advantage is that it requires less coupling between the View and the Presenter then the previous approach.

When the View requests the Presenter to process a user action it doesn't provide the Presenter with any details, so the Presenter must then ask the View and/or Model for the data necessary to process that request.

Furthermore, since the View reflects the state of the Model, once the Model has been updated the View must be update as well; therefore it is necessary to perform their synchronization. The View and the Model can also communicate in different ways:

- The View observes the Model for updates (the View is aware of the Model) and/or
- The Presenter observes the Model for updates and updates the View accordingly (the View is not aware of the Model).

The Presenter should actually contain a reference to an interface of the View and not to an actual implementation of the View. This would enable the reuse of the Presenter in that, several Views (perhaps implemented using different technologies) could then share the same Presenter. It would also allow for the replacement of the actual View with a "mock" implementation to facilitate testing. Hence, the Presenter and the View should not be tightly coupled so that one View may be completely replaced with a different one. There may be several Views and Presenters for a single UI. Regarding the manner in which the View and the Model can communicate there are two variants of this pattern: Supervising Presenter Pattern and Passive View Pattern. In the Supervising Presenter Pattern the View directly communicates with the Model. The View contains logic that can be described declaratively, while the Presenter is involved in more complex cases. In the *Passive View Pattern* the View is not aware of the changes in the Model and the communication with the Model is solely through the Presenter.

1) *Supervising Presenter/Supervising Controller Pattern*

CLARIFICATION: Since this pattern is a variant of the MVP pattern we consider that Supervising Presenter is a more appropriate name than Supervising Controller, though both names can be found in the relevant literature.

COMPONENTS: This pattern proposes that the functionality of the user interface be split into: View and Presenter. *View* holds the state as well as simple mappings to the Model. Therefore, controls in the user interface can be directly bound to the domain Model. *Presenter* (often dubbed the Controller) contains the presentation logic. It has two main responsibilities: reacting to user actions and partial synchronization between the View and Model. The Presenter must observe its associated View and, if necessary, react by updating the View and/or Model.

INTERACTION (Fig. 2): Once the Model is updated the View should also be updated to reflect these changes and vice versa, when the user interacts with the UI in the View the Model should be updated accordingly. In terms of synchronization:

- The View usually uses some sort of binding technology for simple mappings, so the updates to the Model can be automatically reflected in the user interface without the intervention of the Presenter. Therefore, the View is aware of the Model and observes it. Conversely, as the user interacts with the user interface updates can be made to the Model without the intervention of the Presenter.
- More complex logic is left to the Presenter. The Presenter should interpret the updates to the Model so that in can update the View in a more complex fashion, while in other cases the Presenter can update the Model based on updates to the View.

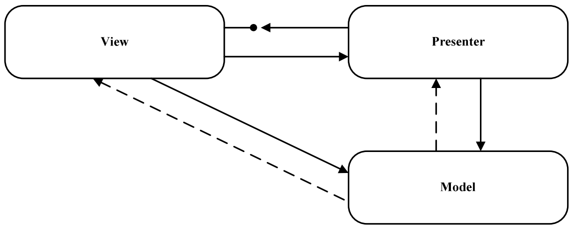

Figure 2. Supervising Presentation Pattern

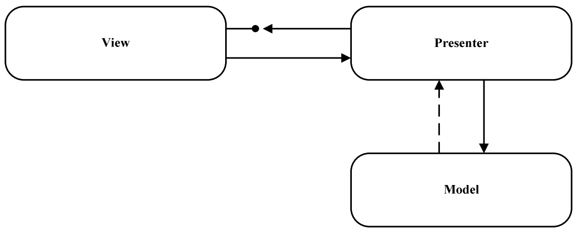

Figure 3. Passive View Pattern

Since the View forwards user actions to the Presenter it must reference the Presenter. On the other hand, the Presenter must reference the View so that it can update the View when the Model is updated. Seeing how the Supervising Presenter is dependent on the View that is assigned to it, while testing the Presenter, an instance of the actual View is required, or an object simulating the behavior of the View. Some of the disadvantages of this pattern are that it is difficult to determine what part of the presentation logic is the responsibility of the Presenter, which may lead to inconsistencies in the project. Furthermore, as the logic is contained in the Presenter it is still tightly coupled to the View and must know its details of the View.

2) *Passive View Pattern*

COMPONENTS: This pattern proposes that the functionality of the user interface be split into: View and Presenter. *View* only holds the state. *Presenter* contains the complete presentation logic including the mapping. In addition to reacting to user actions the Presenter is also responsible for the complete synchronization between the View and the Model. The Presenter must observe its associated View and, if necessary, react by updating the View and/or Model.

INTERACTION (Fig. 3): The View is completely passive, hence, there is no dependency between the View and the Model. The Presenter contains a reference to the Model. Once the Presenter obtains the data from the Model it directly updates the properties of the View, thereby eliminating the need for the View to have any knowledge of how to display the object data correctly.

The advantages of this pattern are that testing can be focused on the Presenter since the View is passive. In addition, while binding gives the best results when dealing with nonhierarchical objects, when there is a hierarchy or aggregated data present, the Passive View provides better control of the synchronization. On the other hand, the disadvantages are that the Presenter could become as bulky as the Autonomous View and there it also requires frequent View Presenter communication.

D. Presentation Model Pattern (PM)

COMPONENTS: This pattern proposes that the functionality of the user interface be split into: View and Presentation Model. *View* represents the display on the user interface. It also holds the details pertaining to the chosen technology and graphical components. *Presentation Model* which on the one hand represents an abstraction of the View independent of the actual user interface technology, i.e. it represents the state and behavior (logic) of the View without going into the specifics of it rendering. On the other hand, the Presentation Model customizes the data for presentation in the user interface, so it could be said that it performs a specialization of the Model.

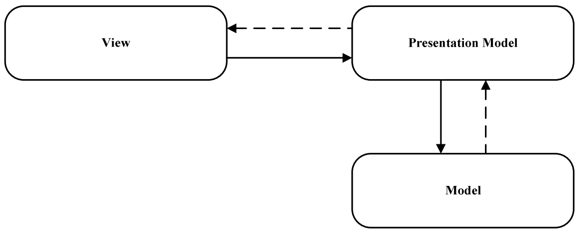

Figure 4. Presentation Model Pattern

INTERACTION (Fig. 4): According to Martin Fowler there exist two variants of this pattern [8], [9]: In the first, which will be referred to as PM1, the View is aware of the Presentation Model - then the View is responsible for synchronization. In the second, which will be referred to as PM2, the Presentation Model is aware of the View - then most of the synchronization is performed by the Presentation Model. Therefore in PM2 the View is simple, it just contains properties through which it exposes its states and it raises events in response to user actions.

The View simply displays the state of the Presentation Model. In accordance with the changes in the Presentation Model the View updates the display. The Presentation Model just changes its state and relies on the binding mechanism or a similar technology to update the View. Therefore, the Presentation Model alone is responsible for the display while the View is simple. The Presentation Model also contains all the dynamic information of a View. The View must therefore, frequently synchronize its state with the Presentation Model, since it holds the information the View needs to display the controls. This synchronization is usually performed through the use of the Observer pattern. The Presentation model synchronizes with the Model and represents an interface towards the View. Thus changes in the Model are reflected in the View via the Presentation Model. The Presentation Model may be linked to several domain objects which implies that a one-to-one relationship between the Presentation Model and the Model isn't required. Furthermore, several Views may use the same Presentation Model, while on the other hand, each View should correspond to a single Presentation Model.

The advantages of this pattern are the possibility of writing logic which is completely independent of the View used to display the data, and in addition, since the details concerning the chosen technology are not a part of the Presentation Model it can be reused. The disadvantage is that it requires a mechanism for the synchronization of the View and the Presentation Model.

III. Comparative Analysis

The comparative analysis of the presented PPs will be given through two tables illustrating the main sources of differences between these patterns, namely the concerns associated with each of the components and the different manners of interaction between the components. Therefore, the first table (Tab. 1) will depict the different responsibilities the components have in each of the patterns regarding the identified concerns, and the latter (Tab. 2) will depict the interactions between the components. Regarding component interactions it should be pointed out that even within a certain pattern there numerous variations are present. The figures given in this paper represent component interactions most frequently found in literature.

Finally, certain variations are the result of different implementations of the patterns. All of the patterns in this paper were presented without giving the specifics of their actual implementation (how many classes will a component contain, in which assembly will they be defined, which class will be created first, and so on), since the goal was to identify the key concepts of the patterns thereby enabling the representation of these patterns as Platform Independent Models.

TABLE I. Comparative analysis of presentation patterns regarding the separation of concerns

CONCERNS	**View**	**Controller/Presenter/Presentation Model**	**Model**
Autonomous View	Holds the state.	N/A	Manages the behavior, state and data of the application domain.
	Contains all of the presentation logic.		
MVC	Holds the state.		
	Contains a minimum of presentation logic.	Contains most of the presentation logic.	
	Synchronizes the View with the Model.	Synchronizes the Model with the View.	
Supervising Presenter	Holds the state.		
		Contains most of the presentation logic.	
	Performs simple mapping.	Performs complex synchronization.	
Passive View	Holds the state.		
		Contains all of the presentation logic.	
		Performs synchronization.	
Presentation Model (PM 1)		Holds the state.	
	Contains the logic tied to the chosen graphical components.	Contains most of the presentation logic.	
	Performs data binding with the Presentation Model.		
Presentation Model (PM 2)		Holds the state.	
	Contains the logic tied to the chosen graphical components.	Contains most of the presentation logic.	
		Performs synchronization.	

TABLE II. Comparative analysis of presentation patterns regarding the interaction between components

INTERACTION	**View**	**Controller/Presenter/Presentation Model**	**Model**
Autonomous View	Aware of the Model.	N/A	The Model is intended to be completely independent of the presentation functionality. Therefore it is unaware of both the View and the Controller/Presenter/ Presentation Model. In some implementations it can be Observable so either the View or the Presenter can subscribe to it.
MVC	Aware of the Model.	Aware of the Model.	
	Unaware of the Controller.	Aware of the View.	
Supervising Presenter	Aware of the Model. For simple mapping communicates directly with the Model.	Aware of the Model. For complex mapping performs the communication between the View and the Model.	
	Unaware of the Presenter.	Aware of the View.	
Passive View	Unaware of the Model.	Performs all of the communication between the View and the Model.	
	Unaware of the Presenter.	Aware of the View.	
Presentation Model (PM 1)	Unaware of the Model. Completely responsible for the synchronization with the Presentation Model.	Aware of the Model. Coordinates the communication between the View and the Model.	
	Aware of the Presentation Model.	Unaware of the View.	
Presentation Model (PM 2)	Unaware of the Model. Performs simple mapping with the Presentation Model.	Aware of the Model. Performs most of the synchronization between itself and the View.	
	Unaware of the Presentation Model.	Aware of the View.	

IV. Conclusion and Further Work

There exist a number of well-documented and verified patterns that aid the resolution of problems inherent to the presentation layer. Though the choice of the best suited pattern is not often straightforward, comprehending, choosing and adopting the best suited one (which entails the separation of business logic, presentation logic and the user interface-UI) will result in numerous *benefits*: (1) the code becomes easier to read, maintain and alter; (2) designers of the UI can focus on the visual aspects of the application, and easily create and modify the UI, while the programmers can focus on the logic and structure of the application; (3) the amount of code that can be automatically tested (independently of the UI) is increased; (4) the possibility of code reuse is increased, so that certain behaviors can be used in different parts of the same application, while the UI can be customized to different roles and localizations. In other word, the logic remains the same while its visual presentation can differ, thus enabling multiple Views requiring the same behavior to share the same code; (5) the same data can be displayed at the same time through different Views; (6) adding a new View does not need to affect the rest of the application, which is significant considering that UI requirements usually change more often than business rules. On the other hand, the use of PPs also entails certain *problems*: (1) patterns introduce several levels of connection which increases the complexity of the solution; (2) if communication is based on events, debugging and reading the code is more difficult, etc.

If the main criterion for PP selection is the degree to which it facilitates testing, then the following should be taken into account: the automated testing of behavior through the UI may be complicated and time consuming, and it may be difficult to figure out in which component the errors occurred. By relocating some or all of the logic from the View into other components, testing can be facilitated. If automated testing of the UI is required then the MVP pattern and the Presentation Model Pattern are superior to the MVC pattern. With the Presentation Model and Passive View patterns by testing the Presenter or Presentation Model, respectively, most of the functionality is tested without the need for testing the UI itself, since most of the presentation logic resides in them. In the case of the Presentation Model the only potential source of errors is the mapping of the controls in the View into the Presentation Model. With the Passive View even this potential source of errors is removed since the View does not contain any behavior, not even the mapping. Both the Passive View and Supervising Presenter require a mock view to mimic the actual View during the testing. The key advantage of the Passive View pattern in comparison to the Supervising Presenter and Presentation Model patterns is that in both of the latter patterns the View performs part of the synchronization, and that part is difficult to test.

The choice of the best suited PP also depends on the nature of the business application being developed. If the same Views can be applied to different data the MVC pattern is a good choice. If the application contains complex Views that involve numerous user interactions the MVC pattern may be difficult to implement since each of the interactions requires a separate Controller. In this case the MVP pattern is a better alternative since the complex logic can be encompassed into a single class that can be tested independently. Both the MVP pattern and the Presentation Model pattern support multiple UIs so they are a good choice for applications requiring the use of various UI development technologies. The MVP pattern is favored if the data supports binding and doesn't require conversion and modification prior to being displayed, while if the opposite holds true the Presentation Model is preferred. Finally, some technologies providing automated architecture development require the use of particular patterns.

The chosen PP might further enable the automated development of the presentation layer of a business application through the transformation of the chosen Pattern, defined as a Platform Independent Model, into an implementation of that Pattern on the chosen development platform.

The set of concepts (components and means of interaction among them) comprising the chosen PP represent elements of the Presentation Pattern metamodel (M2 layer of the OMG MDA [2]). These metamodels can be specified as specialized UML metamodels [1], [10]. Consequently, based on these identified concepts, UML profiles [11] could be created and then utilized in the presentation layer design phase. Moreover, for the development of a Platform Independent Model of the presentation layer, a UML profile could be defined for each of the Patterns discussed in this paper. Finally, it would be necessary to define the corresponding rules of transformation. The automatic transformation from PIM to PSM could then be accomplished using CASE tools.

In conclusion, the design of a business application's presentation layer should commence with choice of the best suited PP, followed by the creation of an actual model of the presentation layer using the corresponding UML profile which would finally be transformed into concepts of the implementation environment.

ACKNOWLEDGMENT

This paper was supported by the Ministry of Education and Science of the Republic of Serbia, Grant III-44010.

REFERENCES

[1] R. France, D. Kim, S. Ghosh and E. Song, "A UML-Based Pattern Specification Technique", IEEE Transactions on Software Engineering, v.30 n.3, p.193-206, March 2004.

[2] Model Driven Architecture - A Technical Perspective, OMG Document ormsc/01-07-01, Architecture Board, Available at: http://www.omg.org/

[3] Petrović M., Turajlić N and Dragović I, "A Review and Comparative Analysis of Presentation Patterns", Journal of Information technology and multimedia systems Info M, Vol. 34/2010, pp. 35-41, 2010.

[4] http://msdn.microsoft.com

[5] S. Koirala, "Comparison of Architecture presentation patterns MVP(SC), MVP(PV), PM, MVVM and MVC", http://www.dotnetfunda.com/articles/article830-comparison-of-architecture-presentation-patterns-mvpscmvppvpmmvvm-an-.aspx

[6] J. Miller, "Building your own CAB", http://www.jeremydmiller.com/ppatterns

[7] S. Burbeck, "Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC)", UIUC Smalltalk Archive, http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html.

[8] M. Fowler, "Development of Further Patterns of Enterprise Application Architecture", http://martinfowler.com/eaaDev/index.html

[9] M. Fowler, "Patterns of Enterprise Application Architecture", Addison-Wesley, 2003.

[10] X Pang, K Ma, and B Yang, "Design Pattern Modeling and Implementation Based on MDA", LCNS, Vol. 6988, pp 11-18, 2011.

[11] N.C. Debnath, A. Garis, D. Riesco and G. Montejano, "Defining Patterns Using UML Profiles", in Proc. of IEEE Intl. Conf. on Computer Systems and Applications 2006, pp.1147-1150, 2006