A Pattern Language for MVC Derivatives

SAMI LAPPALAINEN, Aalto University
TAKASHI KOBAYASHI, Tokyo Institute of Technology

General Terms: UI-Patterns

Additional Key Words and Phrases: Model-View-Controller, Model-View-Viewmodel, Model-View-Presenter, decision-tree, etc.

ACM Reference Format:

Sami Lappalainen, Takashi Kobayashi. 2017. A Pattern Language for MVC Derivatives *jn* 0, 0, Article 1 (May 2017), 8 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Creating an application user interface and binding it to the underlying application logic involves many considerations such as testability, maintainability, and modifiability of the solution. Over time a number of software developmental patterns have emerged to aid programmers solve problems dealing with coupling user interface to application domain.

For this research paper we have conducted a literature review on the Model-View-* (MV*) family of patterns using the family definition outlined by Syromiatnikov and Weyns in their research on MV* design patterns [1]. The purpose of this paper is to serve as an overview to MV* design patterns and propose a rough method for choosing the most suitable pattern based on application size and requirements. The structure of this paper is the following; An overview of the patterns is given in chapter 2, part 3 contains pattern comparison and definition of the pattern language, part4 is a summary of study.

2. OVERVIEW OF THE PATTERNS

In this section we give a brief overview of the four main MV* patterns; Model-View-Controller (MVC), Model-View-Presenter (MVP), MVP (Supervising Controller) and Model-View-ViewModel (MVVM).

2.1 Model-View-Controller

One of the first patterns in the MV* family is the Model-View-Controller (MVC) pattern introduced at the end of 1970's for the Smalltalk programming language by Trygve Reenskaug [2]. The pattern was developed with the aim to separate visual feedback, user input and application model of each other [3]. The intent of all MV* patterns is to separate the aforementioned responsibilities to different objects. In MVC the three separate objects are called Model, View and Controller. The relations between these objects has been described in figure 1.

The View component is responsible for instantiating the controller, displaying model data and forwarding input commands to the controller object. The controller object is responsible for handling user gestures and input and modifying the model based on that. The model is responsible for holding data and notifying the view of changes in itself. There are several minor variations that can be called the MVC pattern. They differ mainly in the division of labor between the controller and model.

The MVC pattern provides clear separation of concerns and makes it possible to bind multiple different views to the same model. However, by separating the user input and view into separate entities, it is a somewhat unsuitable pattern for modern desktop user interface frameworks, that usually combine these responsibilities by using widgets for creating rich user interfaces [4]. Testability is also of concern

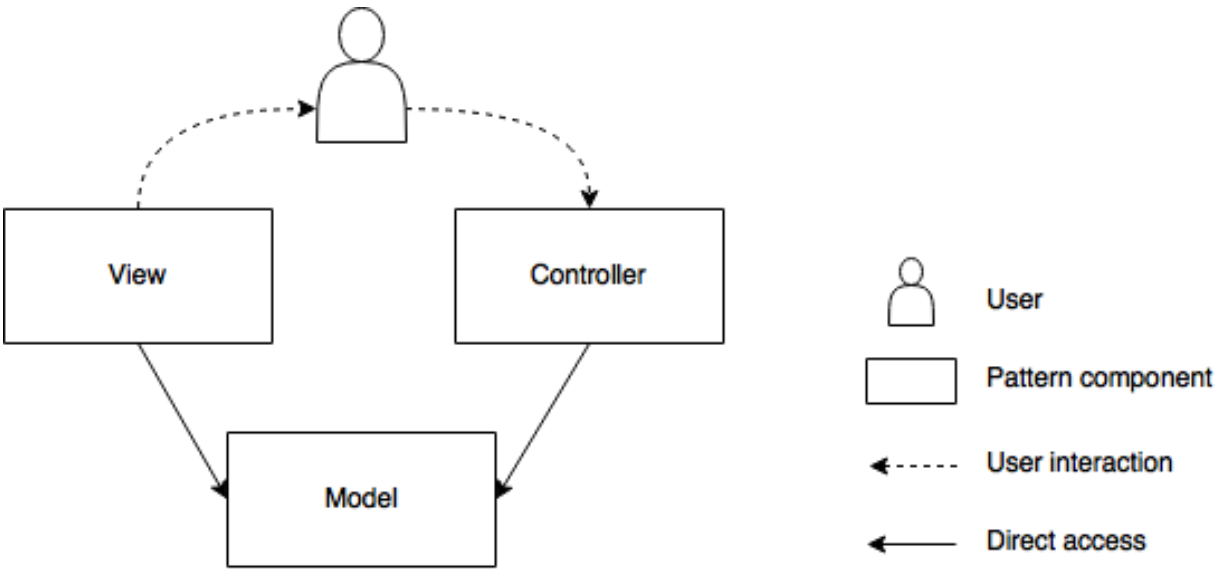

Fig. 1: MVC pattern. Controller and View both have direct access to model. The controller handles user input and changes the model. The view is responsible for observing changes in the model and updating itself to reflect the changes.

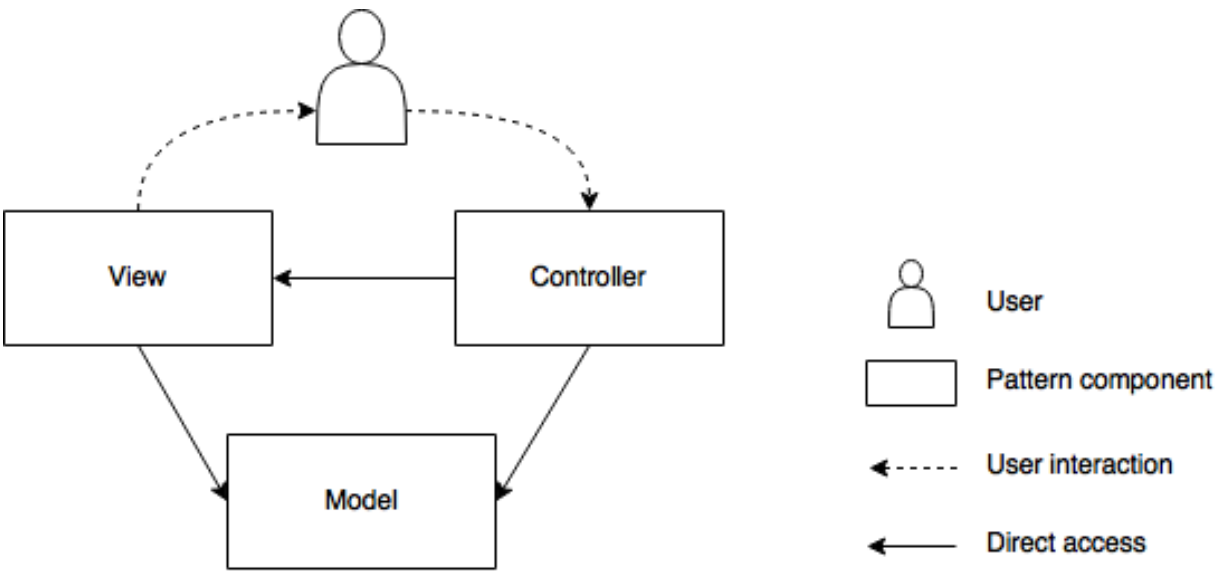

Fig. 2: MVC2 pattern. The controller has access to model and view. It handles user input and decides which view should render the request it received.

since the view needs to handle updating itself leading to untestable code-behind, this can however be somewhat remedied by following Michael Feathers' Humble Dialog box approach and creating a non GUI-class that has a one-to-one mapping of the view functions. This way the view can be substituted

by a mock and tested more thoroughly since the view has no logic in the code-behind. [5] This approach somewhat resembles the Presentation Model or MVVM architectures.

MVC has multiple derivations, most notable being the J2EE MVC Model 2 as shown in figure 2, that was developed for Java web applications to address problems with the MVC pattern. [6] The difference between MVC2 and MVC is that the controller is responsible for processing user input and modifies the model and view accordingly. It then passes the data to a view that renders the data. It provides a clearer separation of concerns, better testability and better maintainability than normal MVC. It is also more suitable for medium and large sized applications.

As shown by N. Delessy and E.B. Fernandez [7] The MVC patterns can also be used for building secure web applications. By implementing several other security patterns and adding them to MVC, they created the secure MVC pattern. It allows users to securely access and modify sensitive information in the model.

2.2 Model-View-Presenter

Pattern	Project size	Separation of Concerns	Code complexity	Modifiability	Testability	Requires view synch.	Platform
MVC	Small/Medium	Fair	Low	Low	Fair	Yes	Web/ Mobile
MVC2	Medium/ Enterprise	Good	Medium/ High	Low	Good	No	Web
MVP (SC)	Medium/ Enterprise	Fair	High	High	Fair	Yes	All
MVP (PV)	Small/Medium/ Enterprise	Good	High	High	Good	No	All
MVVM	Medium/ Enterprise	Good	Medium/ High	Low	Very good	Yes	All

Table I. : Studied pattern characteristics and their differences

The Model-View-Presenter is a derivation of the original MVC pattern published in 1996 by Taligent for C++ and Java. The goal was to improve the MVC separation of concerns and provide a way to facilitate automated unit tests for the user interface [8]. There are two main variations of the pattern called Passive View and Supervising Controller. The two variations have been visualised in figures 3 and 4. The main difference when compared to the MVC-pattern is that the presenter has a direct access to the view through an interface, whereas in MVC the controller doesn't know about the view.

In Supervising Controller MVP the presenter is a slightly modified controller that is responsible for updating the model and notifying the view of changes in the model. The view still holds a direct link to the model, making UI-testing hard. This problem has been addressed in the passive view version where the presenter acts as a connection between the view and the model. The presenter acts on the model when it detects user input and updates the view through an interface when the model changes. This also helps reduce the amount of testable code in the views code-behind.

The Supervising Controller pattern has a similar testability footprint as MVC but it provides a way to handle view state logic without extra fields in the model.

Passive View MVP displayed in figure 4 was developed to address the issues of testability in MVP (SC) and MVC. It contains a presenter that handles user input and communication between the view

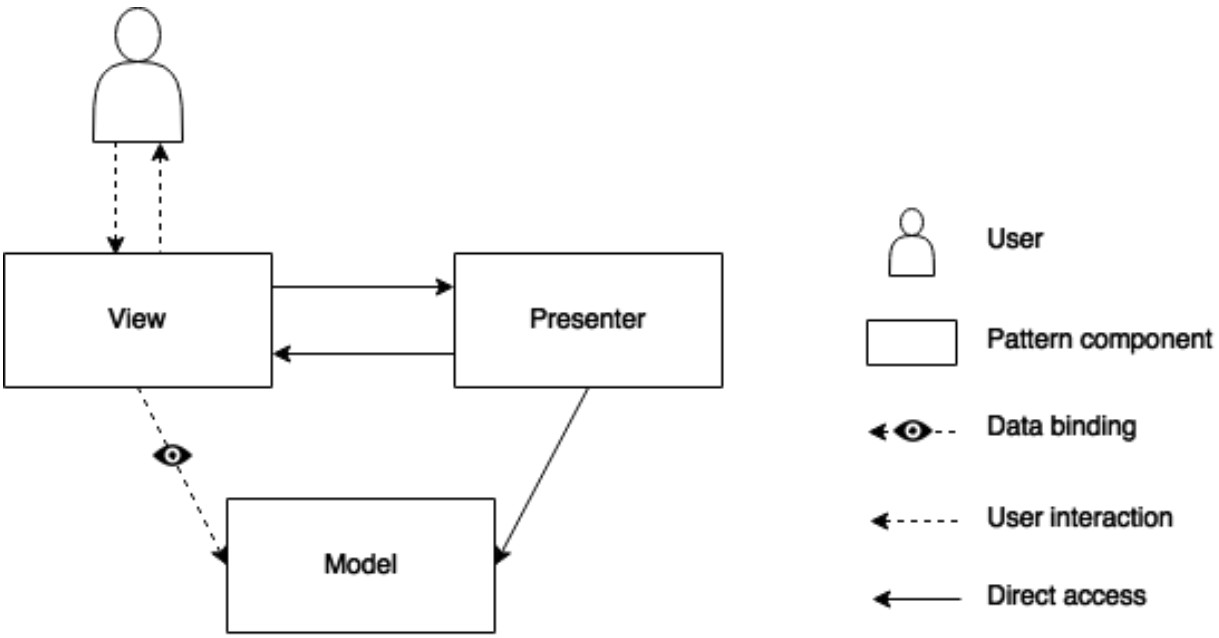

Fig. 3: MVP (SC) pattern. The presenter has direct access to view and model. The view also has a 1-way observer synchronisation binding to the underlying model. Presenter is responsible for handling view state logic and calling business logic methods on the model.

and model. It also handles updating the view through an interface defined by the view. This removes the coupling between view and model and allows for a larger testability footprint.

The benefit of the passive view pattern is high separation of view and model, high testability and flexibility in division of responsibilities. However, it suffers from low re-usability and separation of concerns since the presenter is responsible for updating the view. This may also affect the complexity of code and hamper maintainability [1]. The Passive View and Supervising Controller MVP patterns are suitable for all platforms. [9]

2.3 Model-View-Viewmodel

Model-View-Viewmodel is a derivation of Presentation Model, which is a further derivation of the MVP pattern, introduced in 2004 by Martin Fowler [10]. The pattern aims to decouple the view and viewmodel even further by introducing data bindings between the view and viewmodel. The aim of the pattern is to enforce separation of concerns, support multiple views on the same viewmodel and allow unit testing on the UI. The viewmodel works as an abstract of the model and thus allows decoupling view and model as shown in figure 5.

The viewmodel works as a binder between the view and the model. It holds information about the view state, allows interaction with the model, handles view logic and notifies the view of changes in the model and view state.

The advantage of the MVVM pattern is high testability and low amount of logic in the views code-behind. It also allows the UI to be unit-tested since the view just binds to the viewmodel. The disadvantages of the pattern are for example; high complexity for small projects, debugging data-bidings during development due to their imperative nature and performance issues caused by excessive amounts of

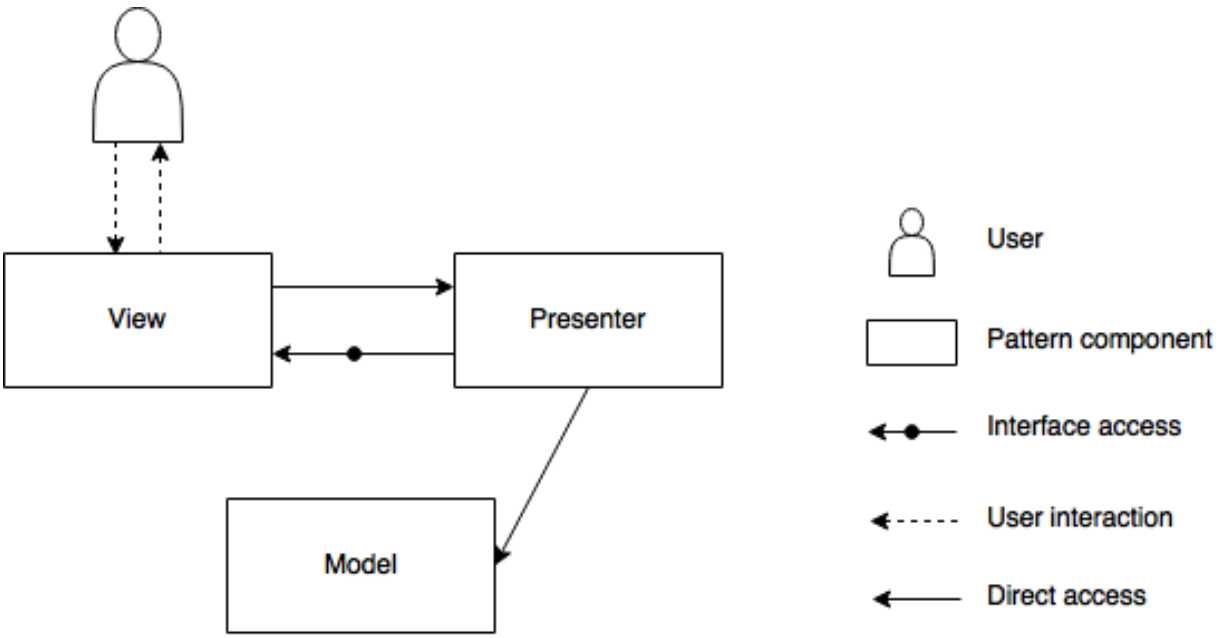

Fig. 4: Passive View MVP pattern. The presenter has direct access to view and model. It is responsible handling view events and model changes.

data-bindings [11]. The pattern is suitable for desktop, mobile and web development, but mainly when using a framework that provides native data-bindings between view and viewmodel.

3. PATTERN LANGUAGE FOR MVC DERIVATIVES

By studying literature four defining features could be found for the patterns; separation of concerns, code complexity, modifiability and testability. Three more characteristics were chosen to aid in making a decision based on project requirements; platform, project size and framework synchronisation requirements. By using these seven characteristics a table I was constructed to display differences in the patterns.

Recommended project size was defined to be either small, medium or enterprise. Small projects were defined as small tools or single view programs where the use of complex UI-patterns such as MVVM might make the application more complex than needed for the task. Medium applications are programs that require multiple views and models, usually commercial applications that require testing before delivery to the customer and benefit from the added complexity of some patterns. Enterprise applications are programs that require extensive and thorough testing, vast amounts of complex models and views.

Separation of concerns is the principle of dividing a program into separate parts that address different aspects of the program. In our scenario the concern is to separate user interface, input and business logic from each other. MVP (PV), MVC2 and MVVM patterns achieve higher separation of concerns [1] and were therefore assigned a higher score than for MVC and MVP (SC).

Code complexity was based on the amount of work required by a programmer to couple the different parts together and the amount of solution elements needed due to pattern usage as explained in Google Android Architecture Blueprints [12]. MVP scores lower than the other two patterns since it re-

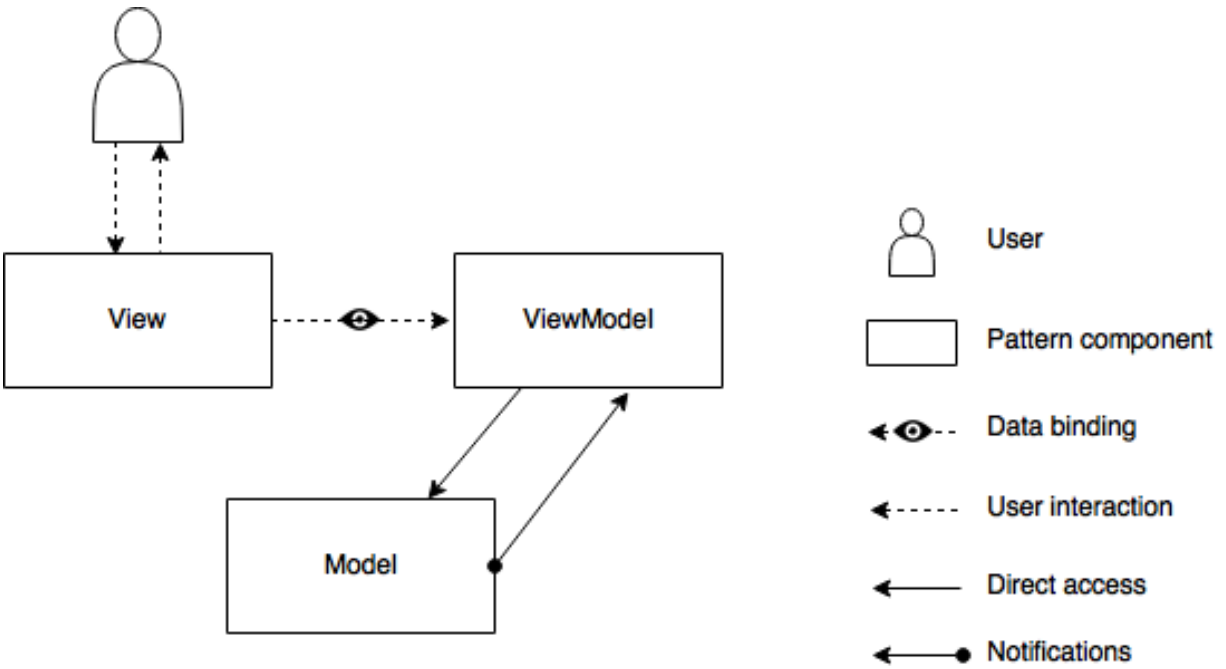

Fig. 5: MVVM pattern. The viewmodel is responsible for calling model methods and works as a proxy of the model for the view. The view is bound to the viewmodel through observer synchronisation.

quires programmers to write more code to implement view updates manually in the presenter, whereas MVVM uses a binding system for coupling the views to the viewmodel and MVC uses observer synchronisation to inform the view of changes in the model [9] [13].

Modifiability is the ease of assigning different roles for different components. The MVC, MVC2 and MVVM patterns don't allow as much choice in this by definition. Whereas the MVP patterns allow a leaner separation of concerns and a thus a greater modifiability [14]

Testability was ranked based on the amount of testable code each of the patterns allowed. A lower level of coupling and logic in views allows for a larger testability footprint. MVVM achieves the highest testability score by allowing programmers to write logic completely independent of the view [10] [14]. The MVP (PV) pattern allows for a large testability footprint by handling view updating through an interface but still requiring some minimal view logic. MVC2 achieves quite a good testability footprint, almost at the level with MVVM and MVP. MVC and MVP (SC) require view logic for communicating with the model, thus scoring lowest on the testability scale.

MVC, MVP (SC) and MVVM require observer synchronisation for coupling the view to the model or viewmodel. Especially in the case of MVVM this can cause a large amount of code overhead which must be taken into account when choosing a pattern when a framework has already been chosen for the application. [10]

Due to low depth of interaction, the MVC patterns are not widely supported by desktop frameworks. However, MVC is widely supported in web and mobile applications, whereas MVC2 is mainly limited to Java web applications . All the other patterns can be used in web, desktop and mobile environments.

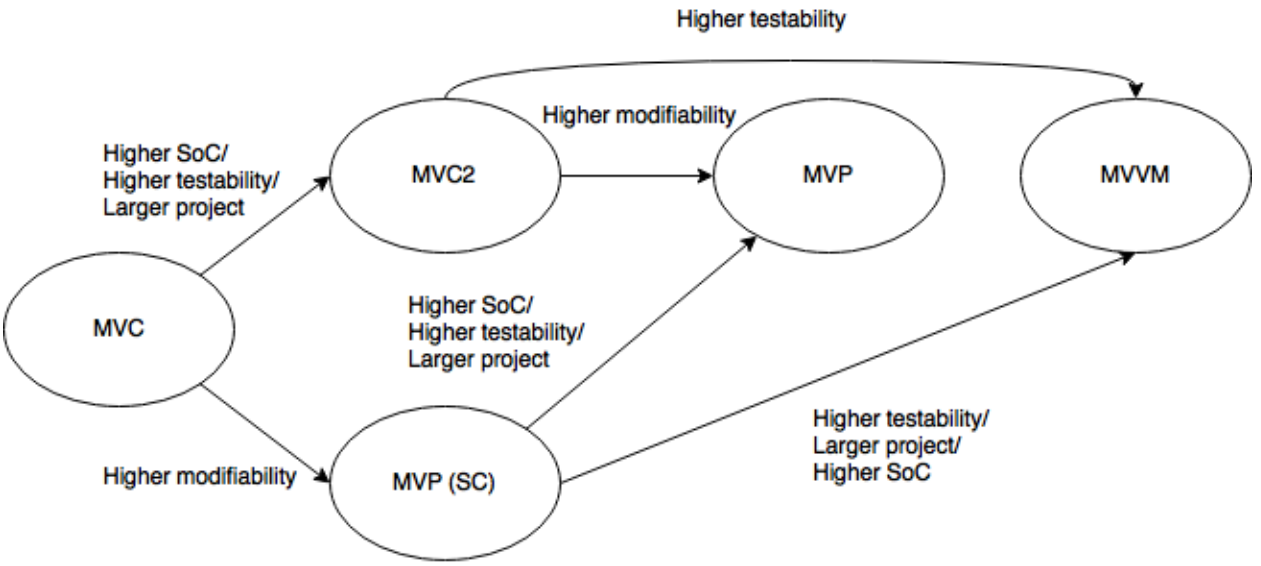

Fig. 6: A pattern language for MVC variants

Using table I as a reference the simple pattern language in figure 6 can be developed. Due to the improvements to MVC achieved in MVC2, it has been added to the pattern language chart, especially since it enables use of MVC on modern medium and large web applications.

From the figure we can note that MVVM and MVP are highly recommended for medium and large applications on almost any platform. MVC is a versatile pattern that can be used in small and medium projects, but with the adaptation of MVC2 it can also be used in large projects requiring quite extensive testing and a more distinct separation of concerns. The MVP (SC) pattern is a good upgrade from MVC, since it offers higher modifiability regarding controller responsibilities.

4. SUMMARY

The Model-View-design patterns are a great and established solution for solving the problem of separation of user interface, user input and application domain in an application. These patterns have multiple differing characteristics that make them suitable for different types of applications on multiple platforms. The MVC-pattern, especially the MVC2 variant, is suited well for a wide range of web applications of all sizes while the MVVM and MVP patterns are viable choices for all platforms rich user interfaces, high testability and long term maintainability are required. The final decision of a pattern should be well grounded on application requirements, framework and the use case at hand.

It should also be noted that since these patterns offer some amount of flexibility, the most suitable pattern for a project might be found somewhere in-between these.

REFERENCES

Artem Syromiatnikov and Danny Weyns. A Journey through the Land of Model-View-Design Patterns. *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on. IEEE*, 2014. http://www.diva-portal.org/smash/get/diva2:738269/fulltext01.pdf.

Trygve Reenskaug. MVC Pattern Language. 2003. http://heim.ifi.uio.no/ trygver/2003/javazone-jaoo/MVC_pattern.pdf [Online; accessed 15-January-2017].

Steve Burbeck. Applications programming in smalltalk-80 (tm): How to use model-view-controller (mvc). *Smalltalk-80 v2 5*, 1992. https://www.researchgate.net/profile/Steve_Burbeck/publication/238719652_Applications_programming_in_smalltalk-80_how_to_use_model-view-controller_mvc/links/5575a00508ae7536375024c7.pdf.

Hans Rohnert Peter Sommerlad Frank Buschmann, Regine Meunier and Michael Stal. *Pattern - Oriented Software Architecture; A System of Patterns*. John Wiley and Sons, 1996.

Michael Feathers. The Humble Dialog Box. *Object Mentor*, 2002. http://home.arcor.de/fbdiplom/lit_pdf/fea02_thehumbledialogbox.pdf.

John Crupi Deepak Alur and San Malks. *Core J2EE Patterns Best Practices and Design Strategies*. Prentice Hall PTR, 2003.

N. Delessy and E.B. Fernandez. The Secure MVC pattern. July 23-27, 2012,. First International Symposium on Software Architecture and Patterns, in conjunction with the 10th Latin American and Caribbean Conference for Engineering and Technology, Panama City, Panama.

Mike Potel. MVP: Model-View-Presenter the Taligent programming model for C++ and Java. *Taligent Inc*, 1996. http://www.wildcrest.com/Potel/Portfolio/mvp.pdf.

Microsoft. The Model-View-Presenter (MVP) Pattern. https://msdn.microsoft.com/en-us/library/ff649571.aspx [Online; accessed 05-January-2017].

Martin Fowler. Presentation Model. 2004. https://martinfowler.com/eaaDev/PresentationModel.html [Online; accessed 10-January-2017].

John Gossmann. Advantages and disadvantages of M-V-VM. 2006. https://blogs.msdn.microsoft.com/johngossman/2006/03/04/advantages-and-disadvantages-of-m-v-vm/ [Online; accessed 13-January-2017].

Android Architecture Blueprints. 2016. Github repository, https://github.com/googlesamples/android-architecture.

Steve Ng and Bimlesh Wadhwa. A Pattern Language for Mobile Application Development. 2014. http://patterns-wg.fuka.info.waseda.ac.jp/asianplop/proceedings2014/asianplop2014_submission_19.pdf.

Lou Tian. A Comparison of Android Native App Architectures - MVC, MVP and MVVM. September 2016.