Lappeenranta University of Technology
School of Technology Management
Degree Program in Information Technology

Bachelor's Thesis

Matti Bragge

MODEL-VIEW-CONTROLLER ARCHITECTURAL PATTERN AND ITS EVOLUTION IN GRAPHICAL USER INTERFACE FRAMEWORKS

Examiner: Professor Kari Smolander

Supervisor: Professor Kari Smolander

ABSTRACT

Lappeenranta University of Technology
School of Technology Management
Degree Program in Information Technology

Matti Bragge

Model-View-Controller architectural pattern and its evolution in graphical user interface frameworks

Bachelor's Thesis

11.8.2013

34 pages, 9 pictures

Examiner: Professor Kari Smolander

Keywords: model, view, controller, mvc, mvp, mvvm, architectural pattern

Model-View-Controller (MVC) is an architectural pattern used in software development for graphical user interfaces. It was one of the first proposed solutions in the late 1970s to the Smart UI anti-pattern, which refers to the act of writing all domain logic into a user interface. The original MVC pattern has since evolved in multiple directions, with various names and may confuse many. The goal of this thesis is to present the origin of the MVC pattern and how it has changed over time. Software architecture in general and the MVC's evolution within web applications are not the primary focus. Fundamental designs are abstracted, and then used to examine the more recent versions. Problems with the subject and its terminology are also presented.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Tuotantotalouden tiedekunta
Tietotekniikan koulutusohjelma

Matti Bragge

Model-View-Controller arkkitehtuurimallin evoluutio graafisissa käyttöliittymissä

Kandidaatintyö

11.8.2013

34 sivua, 9 kuvaa

Tarkastaja: Professori Kari Smolander

Hakusanat: model, view, controller, mvc, mvp, mvvm, arkkitehtuurimalli

Model-View-Controller (MVC) on arkkitehtuurimalli, jota käytetään sovelluskehityksessä graafisiin käyttöliittymiin. Se oli 1970-luvun lopulla yksi ensimmäisistä tarjotuista ratkaisuista Smart UI anti-patterniin, jolla viitataan tapaan tehdä sovellusta missä kaikki logiikka kirjoitetaan suoraan käyttöliittymään. Alkuperäinen MVC-malli on sen jälkeen kehittynyt useisiin suuntaan ja useilla eri nimillä, mikä voi olla hämmentävää monille. Tämän työn tavoite on esittää MVC-mallin alkuperä ja kuinka malli on muuttunut ajan myötä. Sovellusarkkitehtuuri yleisellä tasolla ja MVC:n kehitys web-applikaatioissa eivät ole tämän työn pääfokus. Perustavaa laatua olevat suunnittelupäätökset abstraktoidaan MVC-mallista, ja niitä käytetään tarkastelemaan tuoreempia versioita. Myös ongelmia aiheen ja terminologian kanssa käsitellään.

TABLE OF CONTENTS

ABBREVIATIONS 3

1 INTRODUCTION 4

1.1 Goals 4

1.2 Restrictions 4

1.3 Structure 5

1.4 Terms 5

2 PATTERNS IN SOFTWARE DEVELOPMENT 6

2.1 Pattern Categories 6

2.2 Relevant Design Patterns 7

2.2.1 Observer Pattern 7

2.2.2 Strategy Pattern 8

2.2.3 Composite Pattern 9

3 SMART UI ANTI-PATTERN 11

4 SMALLTALK MODEL-VIEW-CONTROLLER 13

4.1 Components 13

4.1.1 Model 14

4.1.2 View 14

4.1.3 Controller 14

4.2 Communication 15

4.2.1 Passive Model 15

4.2.2 Active Model 15

4.3 Fundamental Principles 16

4.3.1 Separated Presentation or Model-View 16

4.3.2 Observer Synchronisation 18

4.3.3 View-Controller Division 18

5 VISUALWORKS MODEL-VIEW-CONTROLLER 19

6 MODEL-VIEW-PRESENTER 20

7 RELATED ARCHITECTURAL PATTERNS 23

7.1 Model-View-ViewModel..23

7.2 Model-View-Adapter...25

8 SUMMARY AND DISCUSSION 26

8.1 Tutorials on MVC 27

8.2 Repository Pattern and Object-Relational Mapping Frameworks 27

8.3 MVC as a Compound Design Pattern 28

REFERENCES 30

ABBREVIATIONS

EF	Entity Framework
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
LINQ	Language Integrated Query
MVA	Model-View-Adapter
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
ORM	Object-Relational Mapping
SQL	Structured Query Language
UI	User Interface
UML	Unified Modeling Language
WYSIWYG	What You See Is What You Get
XAML	Extensible Application Markup Language

1 INTRODUCTION

Software development is challenging, and the presence of graphical user interfaces (GUI) makes it even more challenging. User interfaces change often and are hard to test. Often there are many interfaces to the same underlying application.

Various solutions have been proposed and used over the years. Understanding the nuances between them can be a challenge. Many GUI frameworks have the letters "MVC" in their name or description, yet they hardly seem to have anything in common with the original Model-View-Controller (MVC) architectural pattern.

The architectural challenges of software and software development are not going to go away any time soon. The topic of this thesis was chosen to both satisfy the author's thirst to understand the complicated topic and apply that understanding into practise, as well as to potentially teach others interested in doing the same.

1.1 Goals

The goal of this thesis is to research and share the history and fundamental design decisions behind the Model-View-Controller architectural pattern. The thesis will work as an introduction to understanding the MVC, and also help to deconstruct its many implementations and variations, with and without the same pattern name.

Research questions are:

1. What was the situation in graphical user interfaces before MVC was invented?
2. How has MVC evolved since then?
3. Which are the fundamental designs that have lived on, and which have been dropped?

1.2 Restrictions

Software architecture in general is such a broad subject that it will not be discussed except when related to user interfaces and the author's personal experiences. No concrete software will be implemented in conjunction with this thesis.

Model-View-Controller's evolution and use within web applications is not discussed. While it is a broad subject, the resulting modifications to the pattern are minor enough that they can be presented within the other text or footnotes.

1.3 Structure

The thesis starts with the concept of patterns and their categorisation in software development context. Next comes the fundamental problem, the Smart UI anti-pattern, that all the following GUI patterns are attempting to solve.

To clearly see the evolution of these GUI patterns, they will be discussed in chronological order, starting with the original Smalltalk version of Model-View-Controller. The fundamental design decisions are abstracted out of the Smalltalk MVC, and the subsequent patterns are discussed in terms of which of these decisions have lived on and which have not. A few related patterns are also briefly presented. The thesis ends with a combined summary and discussion chapter which answers the research questions and presents common problems.

1.4 Terms

Presentation logic - Code that is tied to the presentation, such as iterating a collection of objects to be viewed as rows in a table. It is the only logic that should reside in a view in MVC.

Application logic - Code that is specific to an application or a user interface. In MVC, application logic belongs mostly to the controllers.

Domain logic - The functional core of an application. Domain logic belongs to the model component of MVC. Also known as *business logic*.

Domain class - Represents a combination of behaviour and data that is meaningful to the domain. A *domain object* is an instance of a domain class.

Domain model - A richly connected web of domain objects. Often a separate assembly from a user interface, so that it can be reused between other projects. Domain model does not know about any user interfaces.

View model - View models merely hold data that is being transferred between controllers and views. It is sometimes necessary to format the data of a domain object in another way.

2 PATTERNS IN SOFTWARE DEVELOPMENT

Christopher Alexander, an architect and urban planner, ”*developed a theory of architecture, building and planning that is based on the construction and use of patterns*” (Buschmann et al. 1996). He has been noted saying that

> *”Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”* (Alexander et al. 1977, p. x in chapter “using this book”).

Gamma et al. (1995) brought the same kind of thinking to the world of software development. According to them, what separates the novice designers from the experts, is that novices are overwhelmed by the available options, while expert designers know how to reuse solutions that have worked for them in the past (Gamma et al. 1995). Gamma et al. (1995) started the recording and cataloging of these recurring solutions used in the development of object-oriented software.

Design patterns are generally reusable solutions to common software design problems. They have four essential elements to them: the pattern name, the problem, the solution, and the consequences (Gamma et al. 1995). **The pattern name** is the handle to be used to tie the design problem, its solutions, and consequences together for easier communication between developers (Gamma et al. 1995). **The problem** explains the design situation in which to apply the design pattern as well as its context. **The solution** is an abstract template, describing the elements that make up the design, their relationships, responsibilities, and collaborations (Gamma et al. 1995). Finally, **the consequences** describe results and trade-offs when applying the specific design pattern (Gamma et al. 1995).

2.1 Pattern Categories

Pattern-Oriented Software Architecture (Buschmann et al. 1996) provides a similar take on defining design patterns in software development and further divides them into three categories: architectural patterns, design patterns, and idioms. **Architectural patterns** are considered the highest scale of patterns providing a template for concrete software architectures (Buschmann et al. 1996). **Design patterns** are medium-scale patterns, smaller in size compared to architectural patterns, yet independent of any particular

programming languages or paradigms (Buschmann et al. 1996). Finally, **idioms** are language specific low-level patterns, addressing both design and implementation details (Buschmann et al. 1996).

Buschmann et al. (1996) categorise the Model-View-Controller pattern as an architectural pattern. Architectural patterns have an impact on the fundamental systemwide properties of an application and their selection should be done on the basis of the application at hand. (Buschmann et al. 1996)

While an application can evolve over time to implement design patterns when the need arises, it is by definition very hard to change architectural decisions. When choosing an architectural pattern, Buschmann et al. (1996) recommend exploring several alternatives before deciding on a specific pattern. Buschmann et al. (1996) also underline that *”the selection of an architectural pattern, or a combination of several, is only the first step when designing the architecture of a software system.”*

2.2 Relevant Design Patterns

Compared to architectural patterns which define how a system is divided into subsystems and how those subsystems interact, design patterns are generally applied within a subsystem. The main design patterns relevant to this thesis are the **Observer pattern**, the **Strategy pattern**, and the **Composite pattern**, which are briefly introduced.

2.2.1 Observer Pattern

A good program is supposed to be separated into different modules, which have a single, well-defined responsibility. But as soon as there are different modules, a new set of problems emerges. How do the modules interact with each other at runtime? How will they synchronise state changes? The Observer pattern is intended to resolve these issues. (Hunt & Thomas 2000, p. 157)

> *”Define a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically.”* (Gamma et al. 1995)

The observer pattern has two key objects, a **subject** and any number of **observers**, which have subscribed themselves to receive notifications when the subject changes (figure 2-1). The Observer pattern decouples the subject and the observers from each

other. The subject does not know about its observers, and a single observer does not know about the other observers (Gamma et al. 1995).

A problem with the Observer pattern is that the observers may be blind to the cost of changing the subject because it does not describe what has changed in the subject possibly resulting in much work for the observers (Fowler 2006a). A small change may also result in a cascade of updates if there are many observers (Gamma et al. 1995).

Events in many toolkits and languages are essentially just a rephrasing of the Observer pattern (Fowler 2006a). Publish-Subscribe pattern is also a similar if not identical mechanism for the same purpose (Hunt & Thomas 2000).

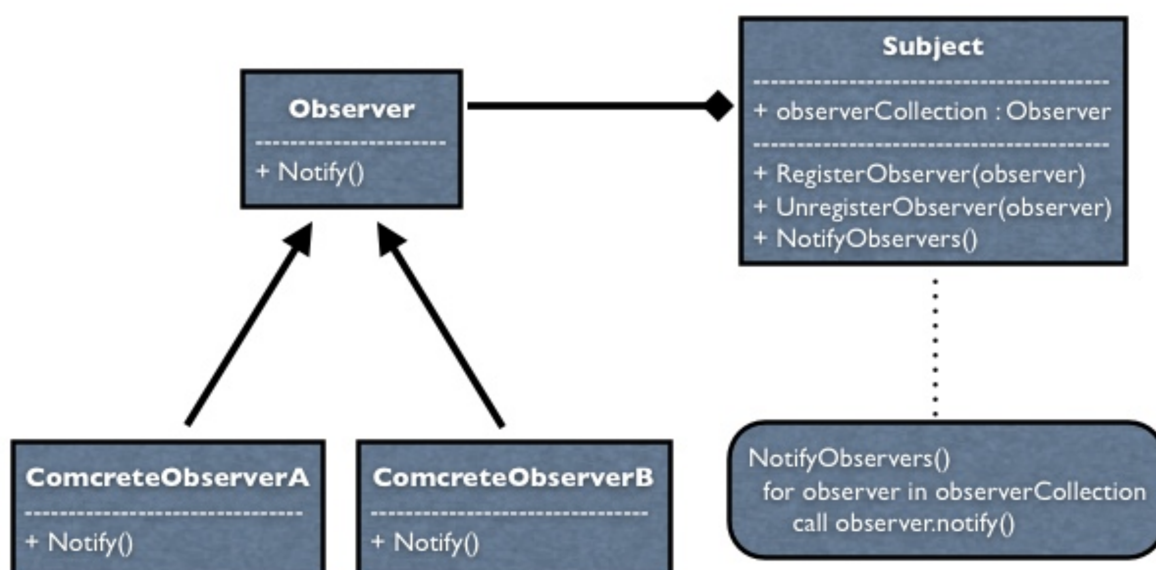

Figure 2-1. UML class diagram of the Observer pattern (Observer Wikipedia)

2.2.2 Strategy Pattern

> *"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."* (Gamma et al. 1995)

The Strategy pattern (figure 2-2) separates an algorithm or algorithms from its user (*context*). It creates a family of algorithms that may be reused and changed at runtime. The Strategy pattern can be used to eliminate conditional statements. (Gamma et al. 1995, pp. 315 - 323.)

A negative consequence of the strategy pattern is that it increases the number of classes in the software solution in development and actual objects at runtime. Clients of the algorithms provided by the Strategy pattern must also be aware of the alternatives and know their differences. (Gamma et al. 1995, pp. 315 - 323.)

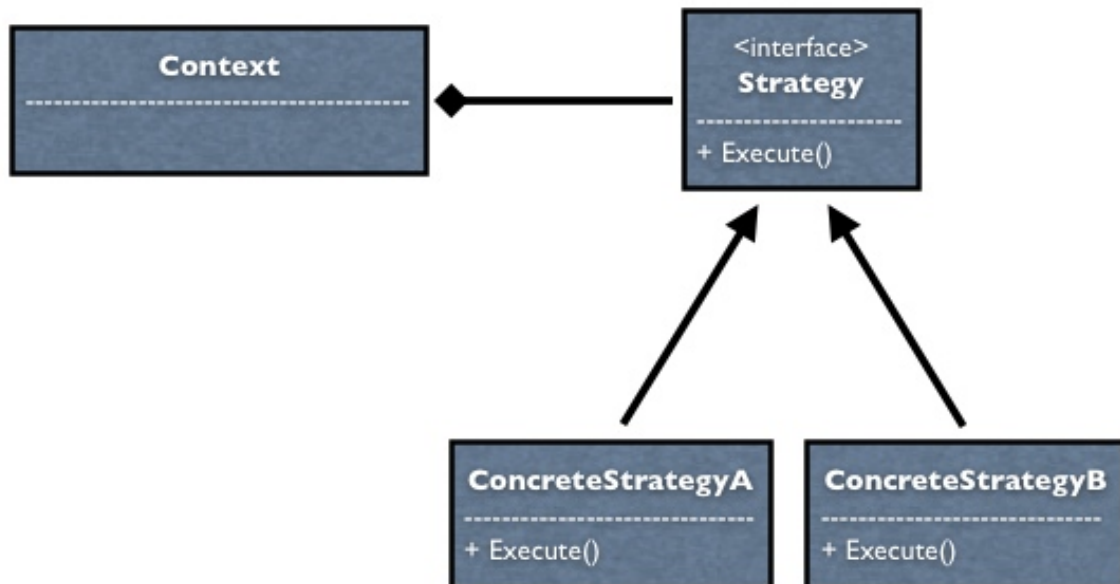

Figure 2-2. UML class diagram of the Strategy pattern (Strategy Wikipedia)

2.2.3 Composite Pattern

> *"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly."* (Gamma et al. 1995)

The Composite pattern provides an abstract class that represents both individual objects and their containers (Gamma et al. 1995). This makes it easy to use recursive composition, i.e. group components into larger and larger components. It is especially useful in user interfaces, which are made up of views containing smaller and smaller views and finally the user interface objects such as buttons.

The pattern consists of the *component*, *leaf*, *composite*, and *client* classes (figure 2-3). **Component** declares the common interface and implements default behaviour for all the classes. **Leaf** presents the actual objects such as rectangles, lines and text. Leafs have no children. **Composite** stores child components and implements child-related operations. **Client** manipulates objects in the composite tree through the Component interface. (Gamma et al. 1995)

Using the Composite pattern makes the client code simpler, because the client can treat leaf and composite objects uniformly (Gamma et al. 1995). It also makes adding new types easy, allowing them to work automatically with existing structures and client code (Gamma et al. 1995). The downside is that the pattern may make the design too general. To restrict a composite to contain only certain objects, it is necessary to rely on run-time checks (Gamma et al. 1995).

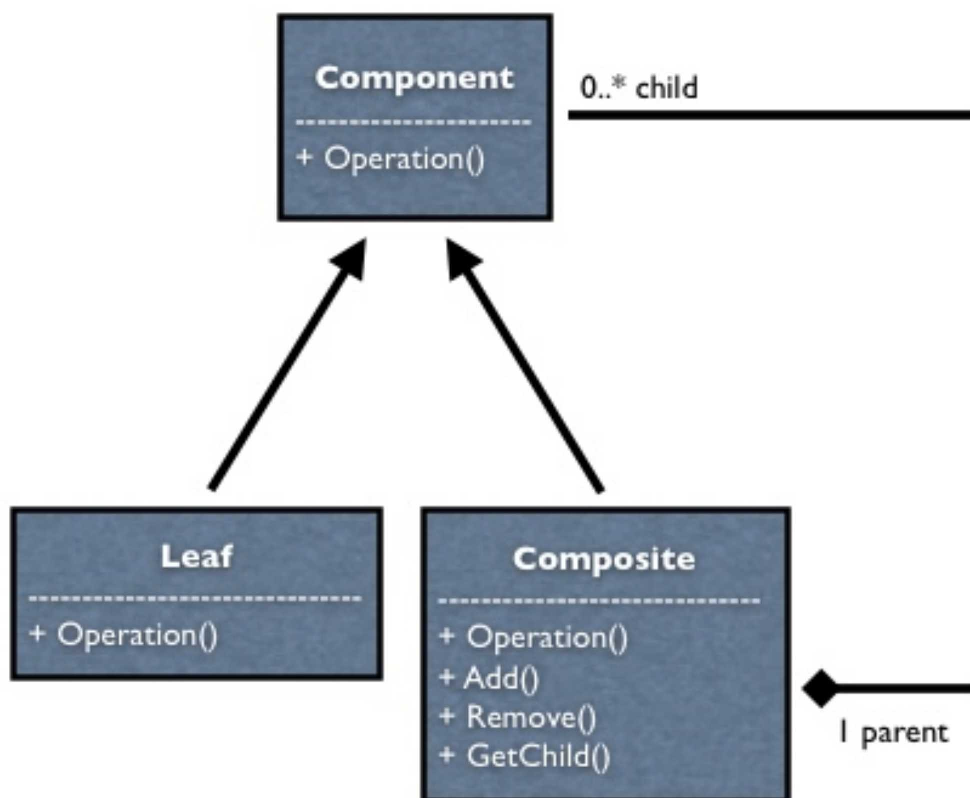

Figure 2-3. UML class diagram of the Composite pattern (Composite Wikipedia)

3 SMART UI ANTI-PATTERN

The Smart UI was the prevalent GUI "architecture" before the introduction of the Smalltalk MVC and still in wide-scale use. In essence, the Smart UI means there is no separation of concerns in an application (figure 3-1).

> *"Put all the business logic into the user interface. Chop the application into small functions and implement them as separate user interfaces, embedding the business rules into them. Use a relational database as a shared repository of the data. Use the most automated UI building and visual programming tools available." (Evans 2004, p. 77)*

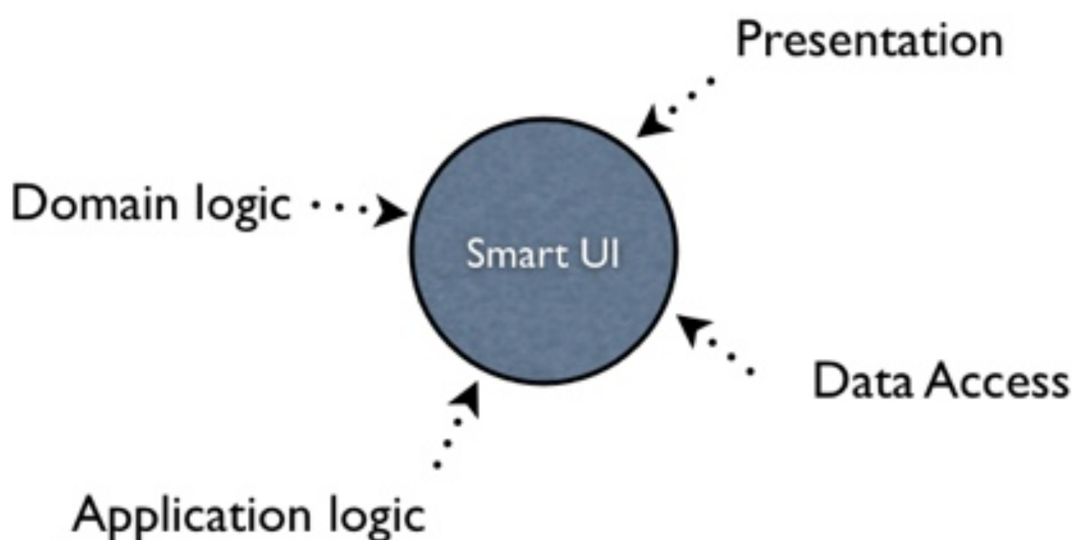

Figure 3-1. The monolithic Smart UI anti-pattern, without any separation of concerns.

Creating a Smart UI may also have benefits. It allows inexperienced developers to be immediately productive and deliver visible results quickly. It can be a legitimate design decision if the project is small and will always remain so. In that case, a sophisticated architecture would cost more than benefit. Copy-pasting source code also has a natural kind of decoupling (but not deduplicating), where changes to one part will not break others. (Sanderson, 2010)

However, a Smart UI does not work well with automated testing, except through user interface automation tools, such as Watir, WatiN or Selenium, which can still be tedious

to script and slow to run. The resulting tests are also tied to the specific user interface, and would need to be duplicated for another one. Smart UI applications become exponentially harder to maintain as they grow (Sanderson 2010) and therefore it is often called an anti-pattern, a practise to specifically avoid.

> *"This style, often the way many programmers learn to program, does not produce programs that readily scale up, can be modified or enhanced, can be reused in whole or in part in other programs, or can be worked on by multiple programmers at a time or over time." (Potel 1996)*

4 SMALLTALK MODEL-VIEW-CONTROLLER

Model-View-Controller (MVC), introduced by Trygve Reenskaug in the Smalltalk community in the late 1970s (Fowler 2003, Reenskaug 2007), was one of the first solutions to approach the Smart UI problem (Fowler 2006).[1] [2]

> *"MVC was created as an obvious solution to the general problem of giving users control over their information as seen from multiple perspectives."* (Reenskaug 2007)

4.1 Components

MVC divides an interactive application into three distinct areas (figure 4-1): *processing*, *output* and *input*. The model performs the role of processing, the view provides visual feedback and the controller handles input or interactions from the user. (Krasner & Pope 1988; Burbeck 1992)

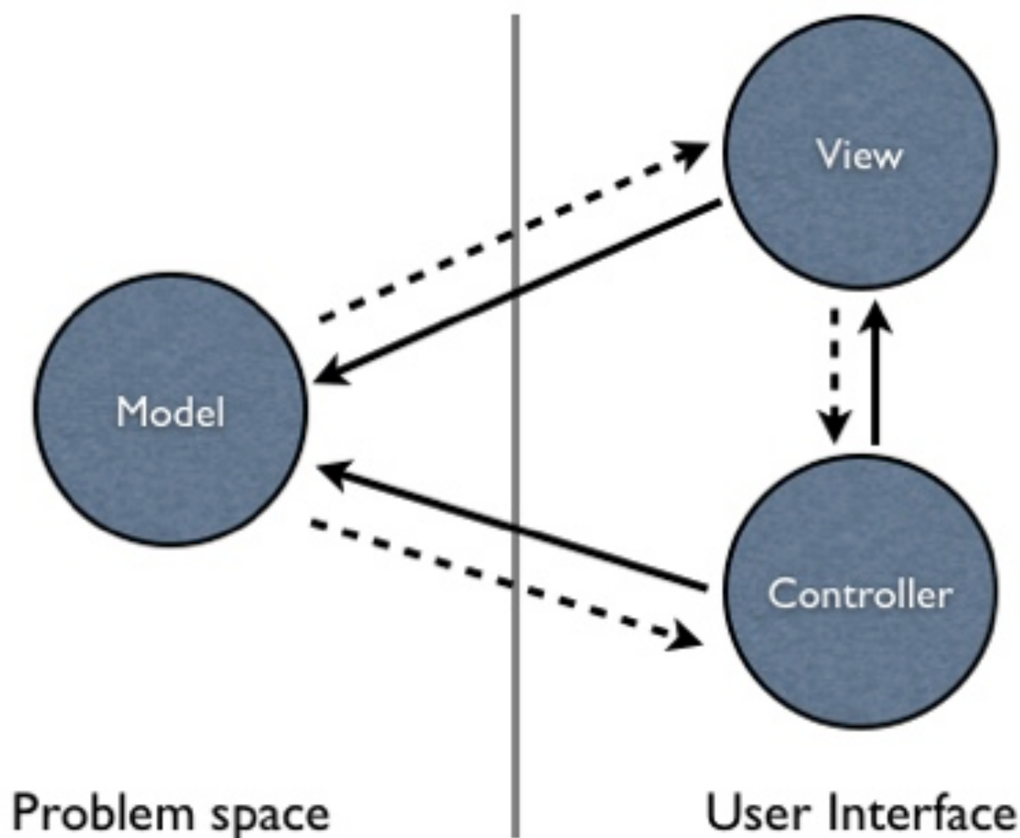

Figure 4-1. Building blocks of the Model-View-Controller architectural pattern. Solid lines represent direct connections, dashed lines represent indirect connections.[3]

4.1.1 Model

The model is a non-visual object containing all the data and behaviour other than that used for its presentation to the user (Fowler 2003). Model represents the data and state (Freeman et al. 2004) of an application and the domain logic that manipulates it (Buschmann et al. 1996). The model provides an interface for other components to access the data (Buschmann et al. 1996). To maximise data encapsulation and code reusability, The model is completely unaware of the other components and does not depend on them (Krasner & Pope 1988; Burbeck 1992; Fowler 2003).

The model can be as simple as a single integer variable, or as complex as an enterprise web service serving JSON or XML data and built with a multilayer or even multitier architecture encapsulating business rules and persistence needs.

4.1.2 View

Visual presentation of an application to the user is done by the view (Krasner & Pope 1988; Burbeck 1992). Views do not know what the data shown to the user will be, they only present it (Buschmann et al. 1996). Views hand over user actions such as mouse movement, keyboard strokes or touch gestures to the controller. Views also offer ways for the controller to manipulate itself, such as turning features on and off (Freeman et al. 2004).

Views are usually nested, so that a view contains subviews inside it (Krasner & Pope 1988). The view on the top of this hierarchy can be responsible for general aspects such as controlling the window the application is running in (Burbeck 1992). In the case of MVC, this is implemented with the Composite pattern, and many more modern UI toolkits and frameworks have also implemented the same pattern (Gamma et al. 1995).

4.1.3 Controller

The controller is what ties the views and models of MVC together (Krasner & Pope 1988). The controller accepts user input as events (Buschmann et al. 1996) from the view and makes the intelligent decisions for the view (Freeman et al. 2004)[4]. This event mechanism is done with the Strategy pattern. The Strategy pattern makes controllers changeable and may be used to provide editable and a non-editable behaviour in the same form, for example (Fowler 2003).

4.2 Communication

The components of the MVC triad must communicate between each other to maintain a coherent state inside the application (Burbeck 1992). The details of how the communication is implemented have an impact on the reusability of the components. The objective is to have a loosely coupled design, which, for example, allows the same model to be used with entirely different presentations.

It is important to note that essentially any object can be a model and partake in multiple MVC triads at the same time. Burbeck (1987) gives an architectural model of a building as an example: while the architectural model stays the same, there can be different views of the floor plan, an external perspective and a view of external heat loss. When the model changes, it will notify all of its views, which in turn will update themselves.

Unlike the model, however, each view and controller are tightly associated together. Both of them have instance variables pointing to each other (Burbeck 1992).

4.2.1 Passive Model

In a simple case where every change to the model happens only from the user's input, the model can be completely passive. It can simply change its state upon demand by the controller, and provide data for the view to present upon request. The controller will assume the role of notifying the view upon changes. (Burbeck 1992)

Burbeck (1987) gives a WYSIWYG text editor as an example where passive model is sufficient. Whenever a user modifies the document, the controller will tell the model what text string to add, remove or replace. The controller will inform the view about the change, and the view can request updated data from the model.

4.2.2 Active Model

If the model can be changed through other means than just the input from a single user, then the passive model is not sufficient. An example of such a situation would be a collaborative text editor, where multiple users can edit the same document simultaneously.

To keep other components in sync, the model needs to implement a change-propagation mechanism (Buschmann et al. 1996), which maintains a list of dependant components and notifies them when a change is triggered (Buschmann et al.1996). The views and controllers implement an update method and register to receive notifica-

tions of the changes of the model (Burbeck 1992). The synchronisation is often done with the Observer pattern.

4.3 Fundamental Principles

The Model-View-Controller architectural pattern in its core tries to tackle many of the non-functional properties of software architecture, such as changeability, testability and reusability. Non-functional properties are very important in software development, however they are hard to measure (Buschmann et al. 1996). Therefore the estimation of how well these properties are met is largely dependent on the experiences of (senior) software engineers (Buschmann et al. 1996).

Discussed next are the most fundamental principles that MVC first brought to light, which can be considered its legacy. A vocabulary given by Martin Fowler is used to discuss the same principles in later patterns.

4.3.1 Separated Presentation or Model-View

The most fundamental principle in the Smalltalk MVC and all the others that followed, is to separate the elements seen on the screen i.e. *presentation* or *user interface*, and the set of classes that are concerned with the core of the application, i.e. the *domain objects* or *data management* (Potel 1996; Fowler 2003). Fowler (2006b) refers to this as **Separated Presentation**, while Sanderson (2010) names it **Model-View** (figure 4-2).

Another strict rule is visibility: the presentation can call the domain objects, but not vice-versa. The domain objects should be completely self contained and unaware of any presentations. This is in fact a form of a layered architecture. The visibility rule can even be validated with build-time tools. (Fowler 2006b)

Encapsulating the model provides multiple benefits. Over time, the underlying data structures of an application can be changed to accommodate new fields or faster searching, for example. New responsibilities such as access control, authentication, and caching can be implemented. While, at first, the model might contain the data of the application inside itself, later it may be turned into a proxy and the data handling be moved to a remote server. This in turns provides sharing between multiple users. (Potel 1996)

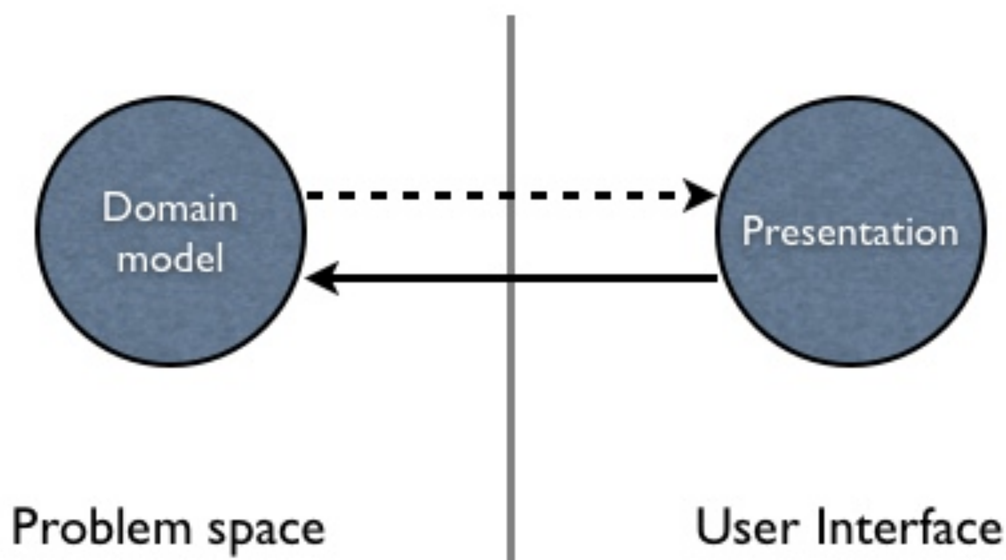

Figure 4-2. Separated Presentation. Presentation can access the domain model, but domain model can only interact with presentation indirectly.

Another benefit is the ability to make different presentations of the same data (Fowler 2003). This might mean having different parts of a rich-client application show the same data in different ways such as a pie chart and numerically. It could also mean re-using the same model (Buschmann et al. 1996) and making it available through a rich client, a Web interface, a remote API, and a command-line interface (Fowler 2003).

When developing the view, the developer will be thinking about how to layout a good user interface and not about business logic or databases. The libraries used will be different and the technologies as well. On the other hand, if the developers on the model are unaware of how it will be presented to the users it simplifies their tasks and makes it easier to add new views later. (Fowler 2003)

This separation allows specialising to one side or the other, based on the developer's preference (Fowler 2003), and enables having different programmers work in parallel on models and presentations (Potel 1996). Specialising cuts down the number of technologies a developer must learn and work with.

Separating the model and the presentation also affects the general testability of an application. Non-visual objects are often easier to test than visual ones (Fowler 2003).

As Potel (1996) so well states it, *"These benefits may appear obvious, but surprising numbers of programs do not employ such clear decomposition or interfaces."*

The weakness of Model-View for overall architecture is that data access code still remains mixed with domain logic (Sanderson 2010). The data access code may even be specific to a relational database vendor (Sanderson 2010), such as an SQL dialect for the Microsoft SQL Server or MySQL.

4.3.2 Observer Synchronisation

It was just stated that the domain model is strictly not allowed to call the presentation, but there are times when this is needed, particularly when the model changes outside of the current user's own actions. The Observer pattern is used to provide this update mechanism, while still keeping the model ignorant of any presentations. When a change occurs, the domain (subject) fires an event to all presentations (observers) noting about the change, and the view can in turn reread their data from the model when needed. Fowler (2006a) names this **Obsever Synchronisation** and it is visualised as the dashed lines in figures 4-1 and 4-2.

The mechanism has an advantage in that the controllers can stay ignorant about which other changes need to be made when a user makes actions. This becomes particularly useful if there are many presentations open. On the other hand, the core problem of the Observer pattern itself is that it is impossible to tell what is happening by just reading the code, and being required to resort to the debugger instead (Fowler 2006a).

4.3.3 View-Controller Division

The second separation after the Separated Presentation (figure 4-2), is that of the view and the controller on the presentation side, referred to as **View-Controller Division**. This separation largely has not lived on in all the newer GUI patterns, and according to Fowler (2003) is hardly ever useful in rich-client systems but has become more important with Web frameworks.

In practise, most systems end up with only one controller per view so the separation of view and controller ends up less important. A classic example in favour of the second separation is to support editable and non-editable behaviour, which can be done with one view and two interchangeable controllers. (Fowler 2003)

Greer (2007) argues that the primary motivation for the original Model-View-Controller was the separated presentation, and that the concept of a controller *"was really a by-product of addressing complexities inherent to the host platform."*

5 VISUALWORKS MODEL-VIEW-CONTROLLER

Some of the problems of Smalltalk MVC were dealing with *view logic* and *view state* (Fowler 2006). An example demonstrating the problem with view logic in MVC is how to colour a field (Fowler 2006a). The decision that more than ten percent is good and less than ten percent is bad is clearly domain logic. The decision that good is shown in green and bad is shown in red is view logic. The problem is where to put this view logic (Fowler 2006), since it does not seem to belong to either the view, which is supposed to be dumb, nor the controller which is supposed to handle input.

VisualWorks version of the MVC can be credited for the introduction of an **Application Model** (Bower & McGlashan 2000; Fowler 2006a). The role of the view is still to present the data and be dumb and the controller will handle user input. Application model (figure 5-1) is a mediator between the true problem space and the views and controllers (Bower & McGlashan 2000). It allows to separate the view logic and view state that is part of the presentation from real domain logic (Fowler 2006a).

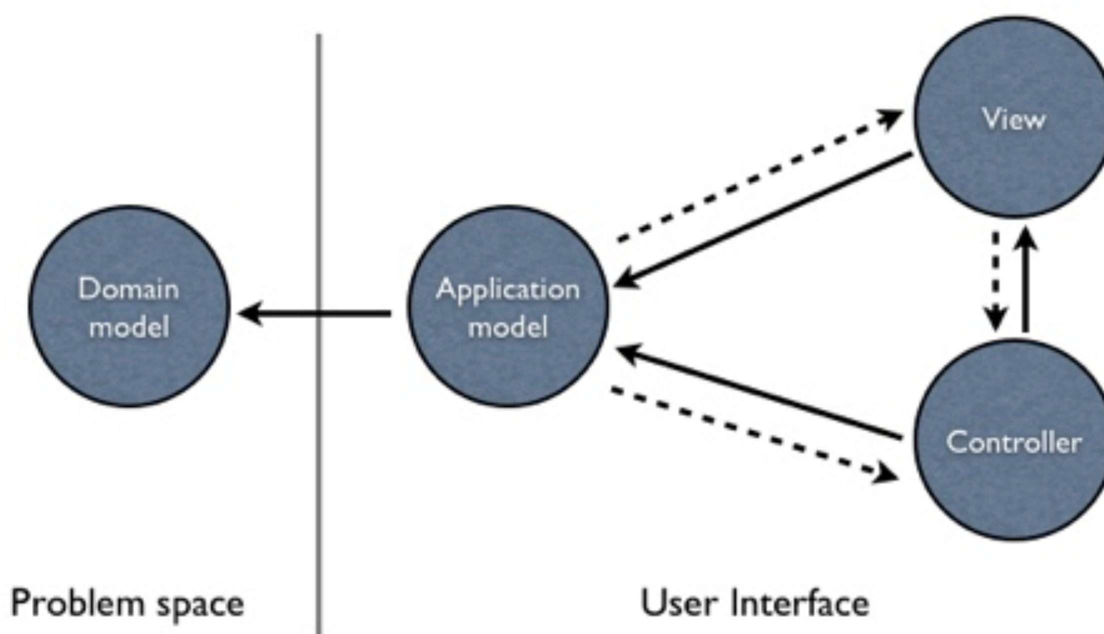

Figure 5-1. The VisualWorks version of MVC, which separated the concepts of domain model and application model. (Bower & McGlashan 2000)

The problem is that in complex cases Observer Synchronisation is not enough for the Application Model, which needed to access the view directly. According to Fowler (2006a), this was seen as a dirty hack and helped develop the Model-View-Presenter approach.

6 MODEL-VIEW-PRESENTER

The Model-View-Presenter (MVP) architectural pattern first appeared from Taligent with the [Potel 1996] paper (Fowler 2006a). It will be referred to as *Taligent MVP*. Another paper on the subject is [Bower & McGlashan 2000], which uses the [Potel 1996] paper as baseline and will be referred as *Dolphin MVP*. The Dolphin MVP compares the Taligent MVP to the VisualWorks MVC but leaves open many of the finer details.

MVP incorporated the principles that were seen to clearly work in MVC: **Separated Presentation** and **Observer Synchronisation** (Fowler 2006a). Compared to MVC, the views in bot the Taligent MVP and the Dolphin MVP are a structure of **widgets**, which do not make the View-Controller Division (Fowler 2006a). In other words the role of the MVC controller is now part of the view. The views still do not contain the active reaction, but leave it for the presenter (Fowler 2006).

The presenter decides how the model can be manipulated and changed by the user interface (Bower & McGlashan 2000, Fowler 2006a). It can be thought of as absorbing the role of the **Application Model** from VisualWorks MVC (Bower & McGlashan 2000).

It is unfortunate that the figure for MVP (figure 6-1) looks essentially the same as the one for Smalltalk MVC (figure 4-1). However, as previously stated the View contains the responsibilities of the MVC controller, and the Presenter is essentially the application model from VisualWorks MVC.

Essentially, differences between the MVP versions come done to the degree of how much the presenter controls the view. Potel (1996) leaves all view logic to the view and the presenter does not get involved in the rendering. Bower & McGlashan (2000) advocates an approach in which the view handles most of the view logic, but the presenter can intervene in complex cases. At the other extreme is the **Passive View** approach, in which the presenter does all the manipulation of the view. This can mainly be helpful with testability. (Fowler 2006)

Fowler (2006) also states that since the MVP presenters and MVC controllers seem so similar, some designs use controller as a synonym for presenter. MVP has been implemented in some .NET (e.g. ASP.NET Web Forms MVP) and Java (e.g. Swing, Vaadin) frameworks, for example. (MVP Wikipedia)

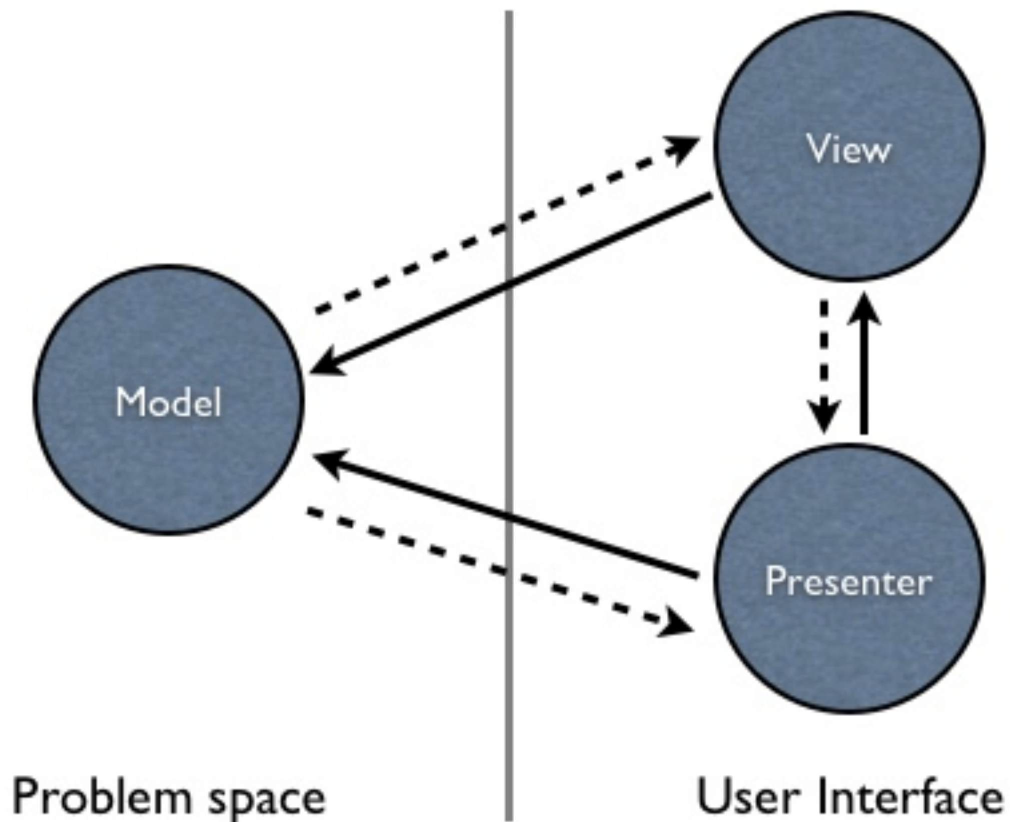

Figure 6-1. The Model-View-Presenter triad. (Bower & McGlashan 2000)

Potel (1996) makes a strong case that the Taligent MVP is well suited to client-server applications. The model represents typical server-side and the view typical client-side functionality. The presenter's responsibilities can then be divided as needed between the server and the client. Most of the presenter code on the client side would result in a *fat client*, and conversely most of it on the server side would be a *thin client*. The responsibilities of some or all the MVP components could even be implemented on both the client and the server. (Potel 1996)

The Taligent MVP, along with the model, view, and presenter components has three more concepts: *selections*, *commands*, and *interactors*. A **selection** is used for picking the data to be manipulated. A selection might be a single database row, or the whole table; and in a text editor it could be a single character, a sentence or the full document. **A command** is used to define an operation on the selected part of a model. Commands could include copying, pasting, deleting, changing the font or its size, or printing. **Interactors** map user's actions like mouse movement and clicks and keyboard keystrokes into events. (Potel 1996)

In [Potel 1996] version, presenter manipulates the model through the Command pattern, and the views are updated with the Observer Synchronisation. In the [Bower & McGlashan 200] MVP, there is no presence of the Command pattern in particular, and the presenter is allowed to manipulate the view directly. (Fowler 2006)

7 RELATED ARCHITECTURAL PATTERNS

7.1 Model-View-ViewModel

Model-View-ViewModel (MVVM) originated at Microsoft in 2005 with Windows Presentation Foundation (WPF) and Silverlight technologies (Sanderson 2010). Compared to MVC applications which are developed with one environment and one language, Model-View-ViewModel is targeted at UI development platforms which have a user experience developer working with the view along with a traditional developer working with domain logic and back end. (Gossman 2005.)

> *"In short, the UI part of the application is being developed using different tools, languages and by a different person than is the business logic or data backend."* (Gossman 2005.)

Roles of the model and the view in MVVM are similar to MVC, with model acting as the completely UI independent data and the view presenting it (Gossman 2005; Sanderson 2010). The View-Controller Division is not present and the role of the MVC controller is the responsibility of the view (Gossman 2005).

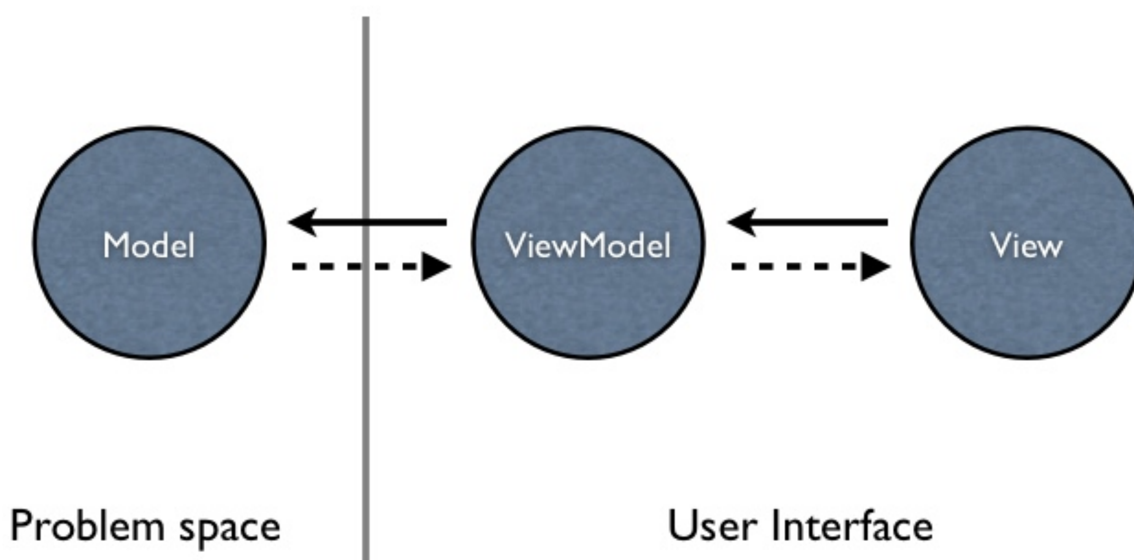

Figure 7-1. The Model-View-ViewModel architecture.

ViewModel acts as the "*Model of the View*", reducing the amount of business logic or glue code stuck in code-behind or event driven code (Gossman, 2006) that is present in the traditional Windows Forms or ASP.NET Web Forms applications. ViewModel in

MVVM context should not be confused with view models in MVC context (Sanderson 2010).

The ViewModel can be unit tested easily without resorting to UI automation techniques (Gossman 2006). The ViewModel is developed by the traditional developers, leaving designers to use HTML, XAML, and other UI technologies. (Gossman, 2005)

MVVM is based on Martin Fowler's similar **Presentation Model** pattern which pulls presentation state and behaviour out from the view (Fowler 2004). *"The essence of a Presentation Model is of a fully self-contained class that represents all the data and behavior of the UI window, but without any of the controls used to render that UI on the screen"* (Fowler 2004). MVMM can be thought of as a specialisation of the Presentation Model pattern, tailor-made for the WPF and Silverlight platforms (Smith 2009).

Presentation Model is essentially the same as an Application Model (Fowler 2004). The difference seems to be the viewpoint. While the term Application Model was introduced with VisualWorks MVC, when there still was a separate MVC controller, the view logic ended up in the model and thus, when it was separated from the model was called Application Model. With MVP and MVVM, where this problem has been resolved with the Presenter and ViewModel components taking the same responsibilities, and clearly being on the presentation side is called Presentation Model.

Unlike the Presenter in MVP, a reference is not needed from ViewModel to a view (Smith 2009). The view uses **data binding** to stay in sync with a ViewModel and property changes in the ViewModel automatically propagate to the view (Smith 2009; Sanderson 2010). There is no code in the ViewModel to directly update the view, and the data binding mechanism also supports standardised input validation (Smith 2009).

A model is oblivious to both the ViewModel and the view, the ViewModel does not know about the view and the view does not know about the model, as seen in figure 7-1. (Smith 2009).

The creator of the MVVM (Gossman 2005) himself has critisised that for simple UIs the MVVM pattern *"can be overkill"* (Gossman 2006). Memory consumption also needs special attention (Gossman 2006).

7.2 Model-View-Adapter

Model-View-Adapter (MVA) modifies the Smalltalk MVC by removing the connection between the view and the model, so that all communication between them goes through the mediating controller, mediator or adapter. As a consequence, the view knows nothing about the model and vice versa. (MVA Wikipedia.)

In practise, many MVC frameworks have evolved towards the MVA design, but in the process have not changed their name. Therefore the term is not widely used and may confuse the situation even more.

For example, while it is possible to develop a Mac OS X application with the Cocoa framework in the traditional sense of the Smalltalk MVC, it is not recommended by Apple's developer documentation (Apple Inc. 2010). Instead, notifications of model state changes usually go through the controller. Thus, from a puristic point of view, the Cocoa implementation should be called MVA and not MVC.[5]

8 SUMMARY AND DISCUSSION

Before the Model-View-Controller architectural pattern, graphical user interfaces were often implemented in the midst of the rest of the application. This type of simplistic design is often called the Smart UI anti-pattern. It has some merits, but will not scale for larger applications and may become a big burden.

The Model-View-Controller pattern was developed in the Smalltalk community to resolve issues with the Smart UI anti-pattern. It separated the domain part of the application into a component called the model, and the presentation side into the view and the controller components. Controllers handle input to an application and decide which methods call from the model. The view is simple and only presents whatever is passed to it. The Model-View-Controller pattern has since been modified many times and the original Smalltalk version is not widely used. The Separated Presentation is so fundamental that it has remained in all versions. The View-Controller Division has not been seen as fundamental and has been dropped in some versions.

Some of the implementations are called MVC, while some have other names. It is more appropriate to talk about MVC and the related patterns in the context of a specific design or implementation, such as referring to the Smalltalk MVC, or the Taligent MVP, or the (Ruby on) Rails MVC.

Communication between the three components depends more on the underlying environment instead of being an integral part of MVC. Observer Synchronisation may be useful, but in certain scenarios it may not be available and other mechanisms are needed. Data binding mechanisms such as the one used in MVVM between View-Model and View may make components even more decoupled than Observer Synchronisation.

The correct location for a particular line of source code depends on the application, it's programming language, chosen frameworks, overall architecture and personal preference. Views should stay dumb and contain presentation logic, controllers should only be about application logic and the rest should be a part of a separate domain project. When there are several equally good options, or no clear place at all, the DRY principle[6] (Don't Repeat Yourself) and questions such as "Should this be reusable between projects?" and "If the code is written here, will it be reusable between projects?" may help.

8.1 Tutorials on MVC

Many available tutorials on MVC, even ones created by framework authors, skip the model part completely opting for a simpler example. In these tutorials the authors write all the domain logic and data access code directly in to the controller classes, and end up with a view-controller architecture instead of MVC (Sanderson 2010).

Having the domain logic and/or data access code directly in the controllers prevents reusing that knowledge between different controllers in the same UI, resulting in duplication even inside the same application. Going further, doing so prevents reusing the same code in a different UI because controller classes are usually tied to the MVC framework by a controller base class. Without a UI independent domain project there is nothing to build a new UI on, resulting in more duplication.

8.2 Repository Pattern and Object-Relational Mapping Frameworks

Another common theme in the tutorials is to use the Repository pattern for hiding data access and actual database queries. *"A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection"* (Fowler 2003, p. 322). Provided that the repositories are in a separate project from the MVC GUI, this accomplishes that data access code is separated and reusable between different projects. However, if the repositories are accessed directly from MVC controllers, without a domain layer in between, the domain logic will still be in the UI. Such a design often results in an anemic domain model, where the domain model is nothing but objects with getters and setters for accessing their database values.

With modern Object-Relational Mapping (ORM) frameworks, such as Microsoft's .NET Entity Framework (EF), a layer of repositories is questionable at best. The ObjectContext API (or the newer DbContext API in EF 4.1+) already provides repository-like collections for free, and an implementation of Unit Of Work[7] for handling transactions. This type of wrapping a repository with another repository design, which seems to be an industry standard in .NET, is often justified with two arguments:

1. Fake versions of the repositories can be created, which instead of using the database will manage in-memory collections, and allow unit testing the upper layers

2. Will make changing the ORM framework and/or database vendor easier

The problem with faking repositories is that it will result in a huge amount of source code. With .NET, EF and LINQ (Language Integrated Query) queries, LINQ-to-Objects will be used instead of LINQ-to-Entities. This will create subtle bugs, because they support slightly different features even though the same queries can be used. When creating fake repositories, the actual repositories will also not be unit tested.

The second argument is ridiculed because it is very rare to change an ORM framework or a database vendor after they have been initially chosen. Preparing for such a small probability may not be worth the added complexity.

8.3 MVC as a Compound Design Pattern

The Model-View-Controller is often presented as being composed of several design patterns, usually the Composite pattern for the views, the Strategy pattern for user input events and the Observer pattern for keeping the components in a coherent state (Freeman et al. 2004, Apple Inc. 2010). While this may be accurate for the classic Smalltalk MVC, modern MVC implementations may be radically different.

The *Cocoa Fundamentals Guide* argues that there is a theoretical problem with the view objects directly observing model changes and that such a design decreases reusability of the view objects. *"In most Cocoa applications, notifications of state changes in model objects are communicated to view objects through controller objects."* The Cocoa MVC controllers incorporate the Mediator pattern along with the Strategy pattern, and the views implement the Command pattern in addition to the Composite pattern. (Apple Inc. 2010)

With MVC web applications, the user's input comes to the application through HTTP Get or Post requests and this might not be according to the Strategy pattern. The Observer Synchronisation may also be infeasible in web scenarios, because the view is an HTML document rendered by a client browser (Sanderson 2010).

These examples alone should demonstrate that while a specific *MVC* implementation may be considered as being a compound design pattern, the abstract architectural pattern itself is not.

---

[1] *Java Server Pages (JSP) Model 2 architecture can be "seen as a server-side implementation of the popular Model-View-Controller (MVC) design pattern" (Seshadri 1999) and seems to be the earliest introduced MVC architecture for the web. Modern MVC or MVC-based web frameworks include Ruby on Rails, ASP.NET MVC and Backbone.js.*

[2] *"While MVC is typically taught in the context of GUI development, it is really a general-purpose programming technique." (Hunt & Thomas 2000)*

[3] *Greer (2007) states that the View-Controller link was a direct instead of an indirect connection in the original Smalltalk MVC, but argues that the direct association "was largely a byproduct of its implementation rather than an inherent part of the MVC pattern".*

[4] *In MVC web applications, a routing engine (also called front controller) decides which controller to invoke based on the HTTP request. Implementations vary on how the model, view and controller components are distributed between the client (browser) and server.*

[5] *Most MVC web frameworks have also dropped the view-model connection and Observer Synchronisation, because changes to the model on the server can not notify the view (which is an HTML page in the client browser). Ajax techniques are slowly changing this.*

[6] *"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." (Hunt & Thomas 2000)*

[7] *"A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work." (Fowler 2003, p. 184)*

REFERENCES

Alexander, C., Ishikawa, S., Silverstein, M. 1977. *A Pattern Language: Towns, Buildings and Construction.* Oxford University Press.

Burbeck, S. 1992. *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC).* Accessed 5 July 2013.
<http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>

Buschmann, F., Meunier, R. Rohnert, H., Sommerlad, P., Stal, M. 1996. *Pattern-Oriented Software Architecture*. John Wiley & Sons.

Apple Inc. 2010. *Cocoa Fundamentals Guide*. Accessed 1 August 2013.
<http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html>

Composite Wikipedia. Accessed 11 August 2013.
<http://en.wikipedia.org/wiki/File:Composite_UML_class_diagram_%28fixed%29.svg>

Evans, E. 2004. *Domain-Driven Design — Tackling Complexity in the Heart of Software*. Addison-Wesley.

Fowler, M. 2003. *Patterns of Enterprise Application Architecture*, pp. 330-333. Pearson Education.

Fowler, M. 2004. *Presentation Model.* Accessed 6 July 2013.
<http://martinfowler.com/eaaDev/PresentationModel.html>

Fowler, M. 2006a. *GUI Architectures*. Accessed 6 July 2013.
<http://www.martinfowler.com/eaaDev/uiArchs.html>

Fowler, M. 2006b. *Separated Presentation*. Accessed 6 July 2013.
<http://www.martinfowler.com/eaaDev/SeparatedPresentation.html>

Freeman, Er., Freeman, El., Sierra, K., Bates, B. 2004. *Head First Design Patterns*, pp. 526-576. O'Reilly Media.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*, pp. 1-6. Addison-Wesley.

Gossman, J. 2005. *Introduction to Model/View/ViewModel pattern for building WPF apps*. Accessed 7 July 2013.
<http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx>

Gossman, J. 2006. *Advantages and disadvantages of M-V-VM*. Accessed 7 July 2013. <http://blogs.msdn.com/b/johngossman/archive/2006/03/04/543695.aspx>

Greer, D. 2007. *Interactive Application Architecture Patterns*. Accessed 19 July 2013. <http://aspiringcraftsman.com/2007/08/25/interactive-application-architecture/>

Hunt, A., Thomas, D. 2000. *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley.

Krasner, G., Pope, S. 1988. *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System*. Journal of Object-Oriented Programming 1.3 (1988): 26-49. <http://www.math.sfedu.ru/smalltalk/gui/mvc_krasner_and_pope.pdf>

MVA Wikipedia. Accessed 7 July 2013. <http://en.wikipedia.org/wiki/Model-view-adapter>

MVP Wikipedia. Accessed 26 July 2013. <http://en.wikipedia.org/wiki/Model_View_Presenter>

Observer Wikipedia. Accessed 10 August 2013. <http://en.wikipedia.org/wiki/File:Observer.svg>

Potel, M. 1996. *MVP: Model-View-Presenter. The Taligent Programming Model for C++ and Java.* Accessed 26 July 2013. <http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>

Reenskaug, T. 2007. *The original MVC reports*. Accessed 1 August 2013. <http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf>

Sanderson, S. 2010. *Pro ASP.NET MVC 2 Framework, pp. 43-50.* Second Edition. Apress.

Seshadri, G. 1999. *Understanding JavaServer Pages Model 2 architecture.* Accessed 19 July 2013. <http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html>

Smith, J. 2009. *WPF Apps With The Model-View-ViewModel Design Pattern*. MSDN magazine, February 2009. <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>

Strategy Wikipedia. Accessed 11 August 2013. <http://upload.wikimedia.org/wikipedia/commons/3/39/Strategy_Pattern_in_UML.png>