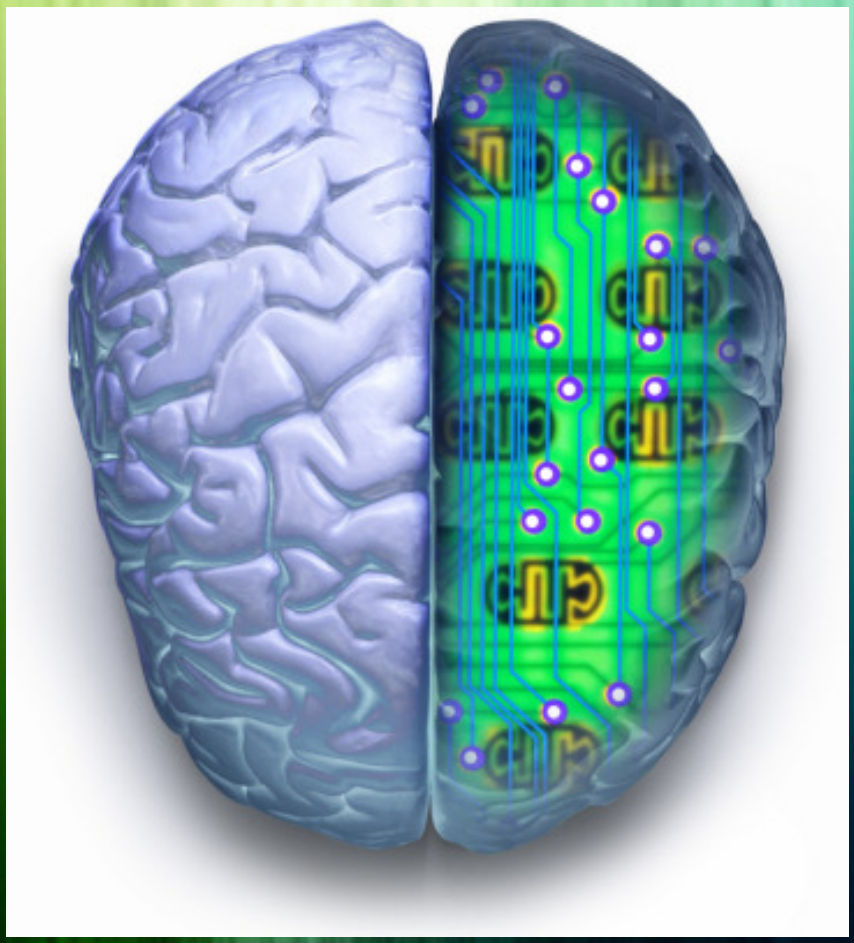
Critical Reflections on Software Architecture and Architectural Design



An Overview of the Model-View-Controller Pattern

Carianne Cowley

February 2012

Contents

Introduction

The aim of this essay is to explore the Model-View-Controller Pattern of software architecture. In order to achieve this, the fundamentals of what constitutes software architecture will firstly be identified and discussed, and furthermore compared with architectural design. Secondly, the concept of a pattern in software architecture will be examined and culminate in the final discussion involving the Model-View-Controller Pattern itself.

Software Architecture and Architectural Design

When conceiving of the notion such as that of software architecture, it is often useful to make use of an analogy that relates to the subject matter at hand. An analogy of obvious suitability is that of building architecture. According to Perry & Wolf (1992), software architecture is analogous to regular architecture in their use of views of particular aspects of systems as well as their use of style in constraining the design elements and their interrelationships. This corresponds with the view of Philip, Afolabi, Adeniran, Ishaya & Oluwatolani (2010), where the architecture of the system is seen as an abstraction of the reality of the system rather than a focus on or allusion to design and implementation issues. The software architect works together with stakeholders by providing an abstract picture of the intended outlay, while avoiding the details of specific implementation design as does a building architect with his/her clients. Thus, with a basic conceptual understanding in hand, a formalised definition of software architecture can be brought forward. Philip et al. (2010, 933) defines software architecture as “a view of the system that includes the system’s major components, the behaviour of those components as visible to the rest of the system, and the ways in which the components interact and coordinate to achieve the overall system’s goal”. Components, behaviour and relationships are thus the three major aspects addressed by software architecture.

Architectural design, though related to software architecture, is a different concept. It is expressed as the process through which elements of the proposed system are designed in finer detail in the context of the selected software architecture. This is achieved through activities such as development of procedures, algorithms, and interface design as well as selection of data types (Perry & Wolf, 1992).

Why Software Architecture?

The foundations for the idea of software architecture and architectural design have been laid, but this doesn't state what it hopes to achieve in practice. Perry & Wolf (1992) assert that the gains to be had from software architecture are that it: serves as a framework in order to fulfil user requirements, provides a foundation for technical as well as managerial aspects, provides a platform for reuse of code and that it allows consistency and relational tests to be performed amongst the elements of the system. Therefore, it is clear that applying the framework of software architecture to the software development process is not only useful, but enables one to ensure that user requirements are actually met as well as to test effectiveness and efficiency of the software solution. It is useful in that it allows one to set a clear boundary as to where the system must exist and what it must contain by restricting the technical aspects to the features representing the architecture in use (Philip et al., 2010).

The Problem with Software Architecture

According to Bosch (2004), the greatest difficulties when utilising architectural design arise from inadequate documentation of the design decisions that are taken at various points throughout the planning process. This can result in the decisions appearing indistinguishable in the final architecture due to their entwinement with each other and the architecture itself. Furthermore, he argues, these gaps in documentation result in a much higher cost of reversal of decisions, both in terms of effort and finances, as changes need to be made on the multiple elements that the specified decision shaped and influenced. If these haven't been documented effectively and accurately, merely attempting to locate these change areas consumes resources. Additionally, the author claims, designers and architects may inadvertently breach the boundaries and conditions of the architecture laid down in these prior undocumented decisions. It is clear that a technique must be implemented to overcome these documentation concerns. One such technique is that of patterns, and this will be discussed in the subsequent section.

Patterns Explained

In terms of software architecture, patterns are "solutions to general architectural problems that developers have verified through multiple uses and then documented" (Harrison, Avgeriou & Zdun, 2007, 2). Patterns, therefore, aid documentation in providing a framework for common problems that arise during architectural design along with their

solutions. In this way the act of documentation is simplified and structure is provided to enable designers to document the design decisions taken at each juncture. According to Gamma, Helm, Johnson & Vlissides (1995), patterns have four crucial components. These are: its name, problem, solution and consequences. The name of the pattern describes the other three components of the pattern in a short phrase, increasing the ease with which a particular pattern can be selected for the given scenario. The problem and solution describe the scenario and context during which the pattern would be relevant and the arrangement of elements that constitute the design respectively. The consequences of the pattern materialise as the advantages and disadvantages of implementing the specific pattern, which assists designers in deciding when a solution may be appropriate above and beyond its resolution of the problem situation (Gamma et al., 1995). This is a simplistic outline of a pattern's identity. However, Gamma et al. (1995) advocate a far more complex means of classifying a pattern in the software architecture community by means of a template. The headings for this more widely used classification of patterns are: "Pattern Name", communicating the fundamental nature of the pattern; "Intent", communicating the problem situation that the pattern hopes to address; "Also Known As", aliases for the pattern; "Motivation", providing a scenario that the pattern is designed to solve; "Applicability", naming a number of problem situations for which the pattern would possibly provide a solution; "Structure", representing the classes in the pattern by means of diagrams and notation; "Participants", naming and listing the purposes of the classes and objects involved in the pattern; "Collaborations", explaining the interactions between the participants in the pattern to bring about the solution; "Consequences", listing the trade-offs and effects of using this pattern; "Implementation", providing tips and hints regarding the practical aspects of using the pattern; "Sample Code", providing snippets of code illustrating use of the pattern in a specific language; "Known Uses", outlining known instances of use of the pattern in existing system and finally "Related patterns", noting which design patterns exhibit resemblance to or are valuable in conjunction with this particular pattern (Gamma et al., 1995).

Advantages of Patterns

It has already been noted that patterns are beneficial in the sense that they assist with documentation. However, it is useful to also understand the manner in which patterns ease documentation of design decisions by exploring the difficulties that they address. Harrison et al. (2007) argues that patterns assist in providing an outline from which the architect can draw, thereby reducing the effort required to document the decision. The pattern name

itself provides a minimal documentation at the very least. Additionally, they argue, patterns make explicit some of the more implicitly reasoned architectural decisions, thereby enabling architects to document elements inherent in the architecture more effectively. The authors also state that developers often wish to postpone documentation until completion of the design process, after which many design decisions and their reasoning, may have been forgotten. Patterns enable architects to document naturally as the design process unfolds. Finally, Harrison et al. (2007) reason, patterns enable architects to document design decisions in a structured way, absorbing the strain of them having to decide how the documentation process should ensue. All of these issues in documentation, therefore, are argued to be viably solved by the implementation of architectural design patterns.

The Model-View-Controller Pattern

Having moved from an introduction outlining the basic concepts regarding software architecture and architectural design to a discussion of what constitutes an architectural pattern, we now turn to a discussion of the specific pattern at hand – the Model-View-Controller Pattern. This pattern will be explored in terms of the template described above in the section on pattern classification, as outlined by Gamma et al. (1995).

Pattern Name

The name of the pattern under scrutiny is the Model-View-Controller Pattern.

Intent

This pattern is useful in the context of interactive applications, where there is a desire to have flexibility in interactions with the user interface. The problem is that various users request different things from an application at various time, and the application interface thus needs to respond appropriately to these requests. Therefore, the user interface must be able to change at runtime when necessary, and without affecting the internal workings and code of the application, and it must also be possible for different representations of the same data to exist for users (Buschmann, Meunier, Rohnert, Sornmerlad, & Stal, 1996).

Also Known As

The Model-View-Controller Pattern is simply abbreviated to MVC (Buschmann et al., 1996).

Motivation and Applicability

According to Burbeck (1987), the one of the most influential writers concerning use of the Model-View-Controller Pattern, the defining characteristic of the Model-View Controller Pattern is that the modelling of the input from the user, external reality model and graphical output to the user are handled in complete isolation from each other, and by three separate object types which have been specifically created for the responsibility each has been granted. Thus, the Model-View-Controller Pattern is useful in scenarios when the application must be divided into its input, processing, and output mechanisms. This is significant when input can vary based on the user's preference or access to peripherals, such as textual commands via the keyboard, or point-and-click input via a mouse or trackpad. It is also relevant when the output can vary based on the selection of the user. This can appear in a context as simplistic as choosing the toolbars that are visible on the screen to the format of a created object displayed to the user (Buschmann et al., 1996). In both instances, the actual code for processing should not be modified. However, if the code for input and output as well as processing is intertwined, complications are caused in modification of code. This is due to the possibility that the code would have to be modified in multiple locations. The processes themselves should be able to be used in various contexts with an assortment of inputs, culminating in a variety of outputs (Buschmann et al., 1996).

Structure

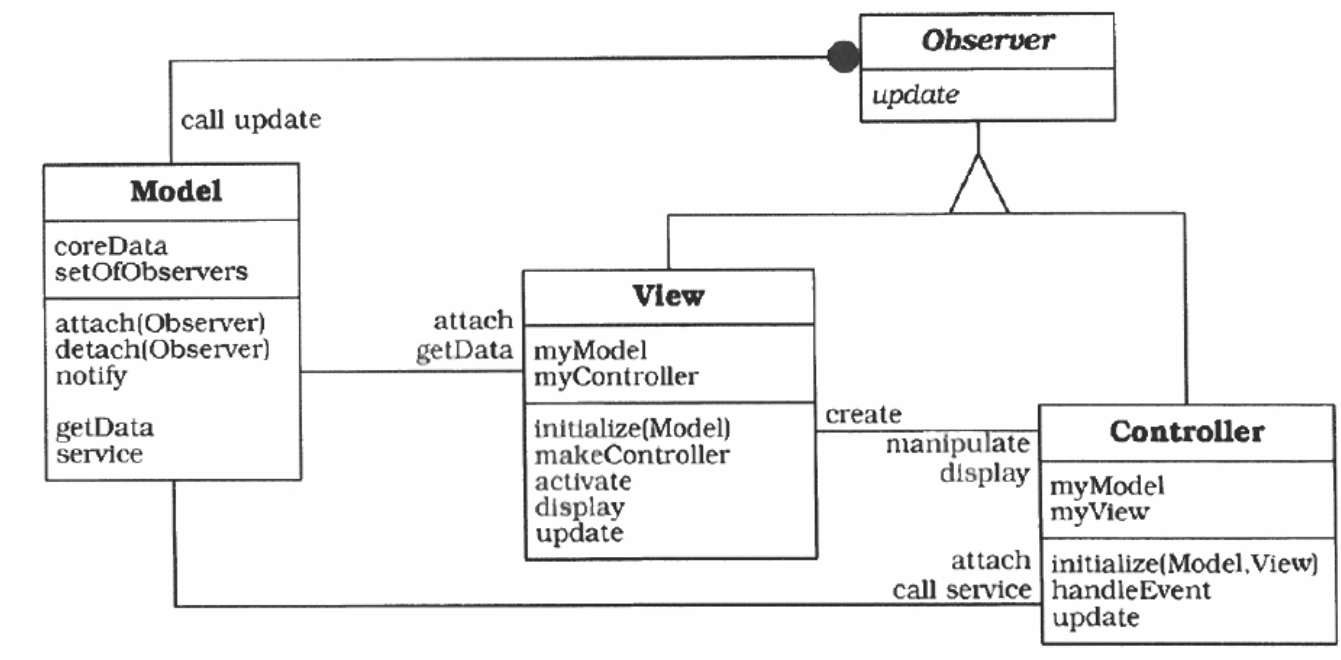


Fig1: Diagram depicting the Model-View-Controller Pattern (Buschmann et al., 1996)

Participants

These three object types used in the Model-View-Controller Patter are the model, view, and controller objects, and make up the interactive application (Burbeck, 1987). These will each be discussed hereunder.

Model

The model contains core functionality of the application, independent of the input and any output to the user (Buschmann et al., 1996).

View

The view is responsible for the output to the user through the screen display that the application is currently utilising. This can either be by means of visual or textual information (Burbeck, 1987). It gains this information from the model, and there can be multiple views of the same model. Each view has a separate controller linked to it (Buschmann et al., 1996).

Controller

The controller is responsible for coordinating between the view and the model. It achieves this by translating the inputs from the user from external peripherals into events which are translated as instructions for the model or the view to change as commanded to (Burbeck, 1987 & Buschmann et al., 1996).

Collaborations

As apparent in Fig 1, flow of control begins with the user interacting with the view by means of input. The controller recognises the input from the user by means of an event-handling procedure, and then calls a procedure that activates the appropriate service for the request from the user. The model then executes the required service, changing the model data. The model commuinicates with all the views and their associated controllers in order to inform them of the change by calling their update functions. Each view then responds by requesting the data that the previous change affected from the model and displays this data on the screen. The controller that was originally involved in the user input begins the listening process once again (Burbeck, 1987 & Buschmann et al., 1996). From this, it is evident that the model element is thus the chief component, and has two main tasks. Its first task is to handle everything occurring within the application itself, such as procedures and data within it. Secondly, it communicates with the other two elements in

order to retrieve information for the view about the current state of the model, or else to change to state of the model as instructed by the controller (Burbeck, 1987 & Buschmann et al., 1996).

Consequences

The benefits of the Model-View-Controller Pattern according to Burbeck (1987) & Buschmann et al. (1996) are as follows:

Firstly, it is possible to implement and display multiple views of the same model. This is due to the separation of the model from the user interface, and is useful at runtime when multiple instances of the model will be utilised.

Secondly, the multiple views are all synchronised and represent the same data. This is due to the model notifying all views and controllers of any changes that do occur and directing them to update to match the new model data.

Finally, the user interface is easily modifiable. This is due to views existing independently of the model and controllers.

The drawbacks of the Model-View-Controller Pattern according to Buschmann et al. (1996) are as follows:

Firstly, the pattern increases the complexity of the application structure. In smaller applications, the benefits of isolating the three elements may be exceeded by the additional effort to develop such an application.

Secondly, the notification system of the model may result in extraneous communication and resulting inefficiencies when the change in the model does not actually make a change to the view at that time.

Thirdly, the use of and allocated of controllers to a single view prevents reuse of controllers, since they are inextricably linked to the functioning that of that particular view. This creates excess code and therefore inefficiency.

Fourth, as views are abstractions of data in the model, frequent updates can slow the application significantly if views are not cached.

Finally, the structure of this pattern makes it a challenging task to make use of toolkits to change the look and feel of the user interface. This is due to isolation of the user interface mechanisms in the view, where toolkits often build certain user interface actions into the model by default.

Implementation

The following fundamental steps involve the implementation of the Model-View-Controller Pattern according to Buschmann et al. (1996):

Step One

Separate the user interface from the core procedures and functionality. In an object-oriented language, create the classes that will represent the objects in use.

Step Two

Implement the mechanism that allows for notification of the views and controllers of changes to the model. This is known as the “change-propagation mechanism” (132). It might be useful to create a separate class to represent an “Observer” that listens for changes to the model.

Step Three

Design and create the views that will be used. This will also involve creating the procedures that create the views on the screen, display the data from the model, and update the views from the model when appropriate.

Step Four

Design and create the controllers that will be used. This involves creating each of the responses to user input by recognising and interpreting the behaviour and then directing the information correctly.

Step Five

Design and create the relationship specifics that will exist between views and their associated controllers. This will involve implementing the code that will create the controller when the specific view is initialised.

Step Six

Implement code that sets up the Model-View-Controller at initialisation. This should appear somewhere in the “main program” or “main method” of a program depending on the language in use. This main code functions as the model.

Further steps can be taken in the process of implementing a Model-View-Controller application, however, these has been selected as the most generic to all applications employing this pattern.

Known Uses

These are the following known uses of the Model-View-Controller Pattern according to Buschmann et al. (1996):

Smalltalk

This is the most well-known, as well as the very first application of this pattern. It was designed to enable reuse of components in the user interface.

MFC

This stands for Microsoft Foundation Class Library. This version of the Model-View-Controller Pattern, known as the Document-View variant, forms part of the Visual C++ environment for developing Windows Applications.

ET++

This framework also uses the Document-View variant. User interface is encapsulated in its own class.

Related patterns

The Presentation-Abstraction-Control pattern is similar to the Model-View-Controller Pattern in that it separates the model from the user interface. However, the views and

controllers are combined to form a singular presentation element (Buschmann et al., 1996).

Conclusion

In this text, the concepts of software architecture and architectural design were explored in detail as a backdrop to a discussion involving software design and development in a commercial setting involving multiple stakeholders. Additionally, the notion of a pattern was offered as a solution to a number of the known issues inherent in architectural design due to its ability to provide a framework for documentation of design decisions. One particular pattern, the Model-View-Controller Pattern, was subsequently discussed and applied to the problem of multiple views in the user interface of an interactive application. It is furthermore deduced, in light of its benefits and limitations, that the Model-View-Controller Pattern is effective as an approach to software development involving encapsulation of application data from its user interface and inputs. This is due to the fact that it inherently provides class types that make it feasible to split input, processing and output into controllers, models and views respectively.

Notes

This assignment has been completed using the APA method of referencing.

References

Bosch, J. 2004. Software Architecture: The Next Step. *Lecture Notes in Computer Science, 3047*, 194-199.

Burbeck, S. 1987. *Applications Programming in Smalltalk-80™: How to use Model-View-Controller (MVC)*. Retrieved from http://www.math.sfedu.ru/smalltalk/gui/mvc.pdf.

Buschmann, F., Meunier, R., Rohnert, H., Sornmerlad, P. & Stal, M. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. England: John Wiley & Sons.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. New York, NY: Addison-Wesley.

Harrison, N.B., Avgeriou, P.A., & Zdun, U. 2007. Using Patterns to Capture Architectural Decisions. *IEEE Software, 24*, 4, 38-45.

Perry, D.E. & Wolf, A.L. 1992. Foundations for the Study of Software Architecture. *Software Engineering Notes, 17*, 4, 40-52.

Philip, A., Afolabi, B.,Adeniran, O., Ishaya, G. & Oluwatolani, O. 2010. Software Architecture and Methodology as a Tool for Efficient Software Engineering Process: A Critical Appraisal. *Journal of Software Engineering and Applications, 3*, 10, 933-938.