Architectural Patterns

Brett Lee

CPSC 545: Software Design & Architecture

Due: April 24, 2011

Introduction

In the field of Software Architecture or Software Engineering, architectural patterns describe the ways in which various elements within a software system will interact. As defined by Bass, et al.: "An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used." [1] The use of an architectural pattern in the design of an architecture provides benefits in providing a standardized method for the implementation system elements, as well as instilling in the design known quality attributes [1].

In many cases, deciding upon an architectural pattern to use is the first step for a software architect in designing upon an architecture to implement [1]. Architectural patterns provide a well-established template for the design of a system's architecture, with many best practices in terms of design patterns and implementation techniques already identified [6].

Documenting Architectural Patterns

The *Open Group Architectural Framework* explains the contents of a software architectural pattern with the following sections: name, problem, context, forces (stimuli), solution, resulting context, examples, rationale, related patterns and known uses [3]. Documenting an architectural pattern illustrates specifically why a specific pattern should be used and how using it accomplishes the goals of the problems that it supposes to resolve. Other relevant attributes that might be of importance to indicate in the documentation are the domain and subdomain in which the architectural pattern operates.

The documentation for an architectural pattern should illustrate clearly how the various subsystems interact in a way that illustrate to stakeholders how the proposed architectural pattern responds to system and business requirements. The subsystem to be indicated might include various layers, external systems, or modules. One way of providing such illustration would be to use a reference model, which is also sometimes referred to as an architectural view [6].

As defined by Bass, et al., a reference model "is a division of functionality together with data flow between the pieces" decomposing the "known problem into parts that cooperatively solve the problem."[1] This decomposition is necessary to the design of the resulting architecture in that it identifies areas that must be approached with specific design patterns or special consideration during implementation. The importance of illustrating these concerns is central in the goal of architecture as described by Bass, et al., as a "vehicle for stakeholder communication."[1]

Some ways of representing these concerns and the relationships between them are by using various types of models or views. Some examples of views include:

- Layered view: The individual parts of the system are decomposed into specific functional layers and the relationship is illustrated using a common method such as Unified Modeling Language (UML). [5] *See Figure 1.*
- Data flow view: The transformation of data is illustrated as it moves through various elements of the design. [5] *See Figure 2.*
- Data-centered view: Data is viewed in a persistent nature being accessed and manipulated by various elements. [5] *See Figure 3.*

Example Architectural View Diagrams

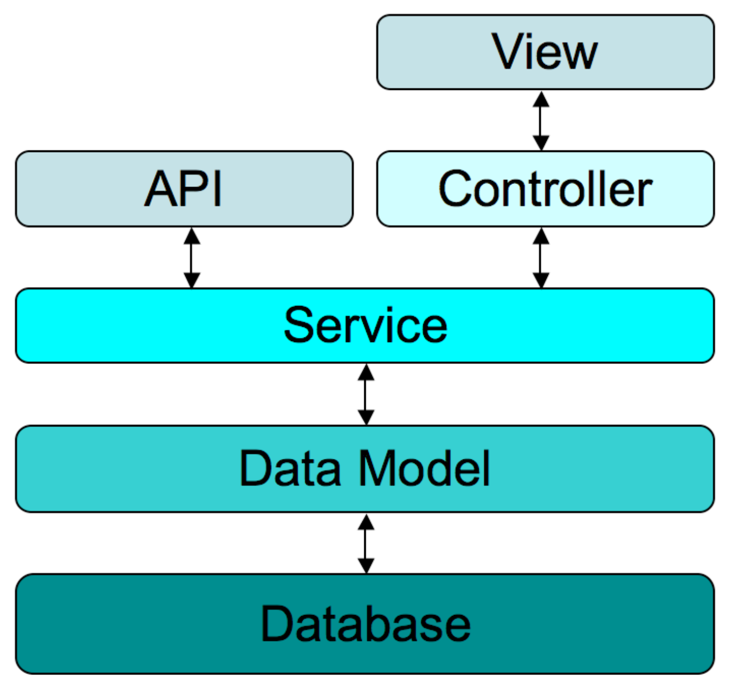

Figure 1: A simplified example of a layered view.

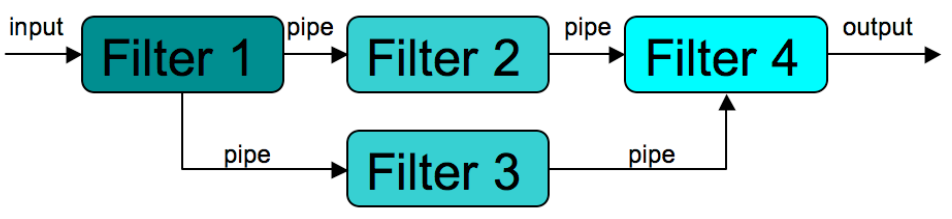

Figure 2: A simplified example of a data flow view.

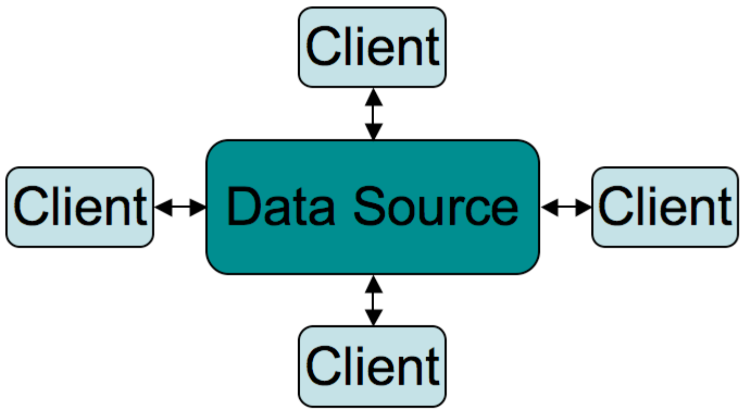

Figure 3: A simplified example of a data-centered view.

Quality Attributes

Software quality attributes describe a system's expected behavior in response to given stimuli, either external or internal to the software system [2]. Quality attributes come in a few flavors: system, business and architectural. System quality attributes include: *availability, modifiability, performance, security, testability* and *usability*, [1] among others. Business attributes are those relevant to the greater business need for an architecture, including but not limited to areas such as *time to market, cost and benefit*, *projected lifetime of the system*, and *integration with legacy systems* [1]. Architectural quality attributes may center on areas such as conceptual integrity, correctness and buildability [1].

Documentation of Quality Attributes

Quality attributes are typically documented in terms of specific scenarios. These scenarios will typically have a source, stimulus, artifact, environment, response (or tactic) and a measurement of response [1]. The source of the stimulus may be internal to the system, or an external action or event. The artifact is an element or resource related to the specific scenario. The response or tactic is the expected behavior of the system corresponding to the stimulus. The response measure relates to the how to quantify, or qualify the systems response to related stimuli. An assessment of the quality attribute can be made based on these factors to determine how well the scenario plays out.

Quality Attributes and Architecture

Quality attributes are related to system architecture in that a system's architecture (as well as the architectural patterns) provides responses to quality concerns [1]. An architectural pattern prescribes specific approaches to the implementation of architecture. As such, the quality attributes and their responses have typically been identified and documented before development begins. This also provides a means of measuring the effectiveness of the architectural design before and during implementation.

Types of Architectural Patterns

Architectural patterns exist for a variety of software architecture subdomains; some are domain-specific and others can have cross-domain applications. Business domains for software architectures can be for any area of software need in an industry. Some possible domains might be financial, avionics, artificial intelligence or data management, among infinite others. Domain models can lead to the distilling of generic subdomains [4]. Generic subdomains can be just as plentiful, focusing on areas such as data architecture, business intelligence, data modeling and web application development among others.

Some architectural patterns are better suited for certain subdomains than others. In the case of web application development (and GUI application development in general), one of the architectural patterns that have become very popular is the Model-View-Controller (MVC) pattern. Many architectural patterns are often used in conjunction with other architectural patterns. A popular architectural pattern commonly used in conjunction with the MVC pattern in the web service subdomain is the Service-Oriented-Architecture (SOA) pattern. In such a marriage, SOA provides the mechanism for the construction of the model component in MVC [5].

An Architectural Pattern Example – Model-View-Controller (MVC)

In the following example, I will be highlighting the previously mentioned Model-View-Controller (MVC) architecture pattern. I will be discussing the problem MVC supposes to resolve, how it resolves said problem within the subdomain it operates.

Name

Model-View-Controller (MVC)

Problem

Complex applications become increasingly difficult to maintain as new features are added. This is especially true of graphical or web applications when user interfaces become highly dependent on business or data store concerns.

Context

The MVC pattern is applicable for user-facing systems having a user interface, business logic and a requirement to access and/or modify a data store of some form (e.g., database, memory, flat-file, etc.)

Forces

- Modifiability
- Modularity
- Manageability

Solution

The MVC pattern works by effectively separating the user interface, input processing and data model elements into separate layers or components [7]. The controller interprets user input and applies application logic. The view correlates to the user interface and works more cooperatively with the controller, together comprising the presentation element of MVC [8]. The model handles data abstraction, ignorant of UI elements, modeling the data source into model objects to be accessed and manipulated by the view and controller (respectively) [8]. *See Figure 4.*

Example Model-View-Controller Diagram

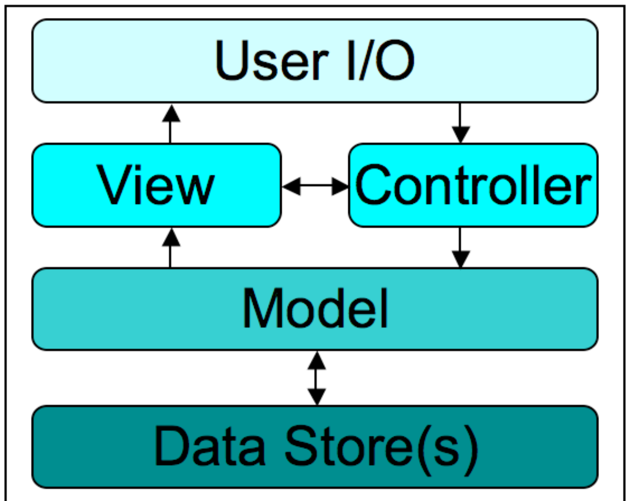

Figure 4: Example of typical MVC implementation.

Resulting Context

Application of the MVC pattern results in explicit separation of model, view and controller elements [7]. This separation of concerns can result in further modularization of elements. For example, within the model, a set of database tables containing related data may be queried for information to be assembled into relevant model objects, which correspond to specific business logic. Within the view and controller, interfaces may be developed with specific business logic in mind, which in turn map well to the model objects being accessed and manipulated.

Examples

There are many examples of MVC application within a number of frameworks. The following is a list of a few:

- The Rails framework (i.e., Ruby on Rails, RoR) [10]
- The Struts2 J2EE framework [11]
- The Yii PHP framework [12]
- Apple Cocoa development framework [13]

Rationale

The rationale for why one would choose to adopt the MVC pattern over others may be specific to the project, or it may be that the MVC pattern is inherent in a framework that is chosen for some other combination of quality attributes. I will highlight rationale for implementing the MVC pattern in relation to forces discussed earlier:

- Modifiability
  Application of MVC results in a highly modifiable solution, in that once the data model is implemented; any number of view or controller modules can be implemented without having to rewrite any code in the data model. If changes

only need to be made to the presentation layer, in many cases all that may need to be changed are the view components.

- Modularity
  The MVC architectural pattern forces modularity upon the developer by requiring that modules be developed for the data model, the controller and the view layers.
- Manageability
  MVC makes it easier to manage complexity through its application of the separation of concerns (SoC) principle. The SoC principle can be summarized as "… system elements should have exclusivity and singularity of purpose … no element should share in the responsibilities of another or encompass unrelated responsibilities." [9]

Related Architectural Patterns

- Model-View-Presenter (MVP) [8]
- SOA Model-View-Controller or N-Tier MVC [5]
- Model View ViewModel (MVVM) [14]

Known uses

MVC started as a development technique for the Smalltalk GUI applications [7]. Since then however, it has been used widely in web application development, [12] which is where it is probably used most heavily presently. Even still, the MVC pattern is seeing wider application in areas of mobile application development [15].

Architectural Patterns vs. Design Patterns

By this point, the term *architectural pattern* has been pretty well defined, however the term *design pattern* has also come up. Some resources use these terms interchangeably, [13] and in some regard they are similar, but they are not quite synonymous.

Design Patterns

As defined by the so-called gang-of-four,—Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides—a design pattern "names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design." [16] Though this definition sounds similar to the definition for architectural pattern; as noted by Avgeriou and Zdun, "it is hard to draw the line between architectural patterns and design patterns," [6] continuing however:

> *"We consider a pattern to be architectural, if it refers to a problem at the architectural level of abstraction; that is, if the pattern covers the overall system structure and not only a few individual subsystems."* [6]

Given the above explanation of what a design pattern is not, it seems that the distinction is one of granularity. Indeed, the patterns focused on by the gang-of-four in *Design Patterns* relate mostly to communication of objects between classes and specific implemented structures. For instance, the design pattern referred to as the "Abstract Factory" has the expressed propose of providing "an interface for creating families of related or dependent objects without specifying their concrete classes." [16] An abstract factory relates to a specific problem encountered within the development of a class, not on an architectural level.

Contrast the resulting artifacts of the abstract factory design pattern with the MVC architectural pattern: the abstract factory results in a piece of code developed to provide a solution to the problem of generating objects without knowledge of their specific class at development time, whereas MVC results in an architectural design to be implemented as part of an architectural framework. Design patterns in this sense apply at the code level and architectural patterns apply at a much higher level.

Relationship to Architectural Patterns

The fact that design patterns and architectural patterns differ in definition and function does not mean there is not a relation between the two. It may be possible for some architectural patterns to prescribe the use of certain design patterns (or a set of design patterns) in the implementation. This could potentially result from an architectural pattern's requirement for a certain level of abstraction of object types, encapsulation or some other design consideration. Typically, the architectural pattern puts in place constraints around which the system is developed, whereas the design patterns represent methods of working within those constraints.

References

1. Len Bass, Paul Clements and Rick Kazman. *Software Architecture in Practice*, Second Edition. (2003) Addison-Wesley, pp. 24-25, 73, 79-90.
2. *Quality Attributes*, from *SoftwareArchitectures.com* (2010). Available: http://www.softwarearchitectures.com/go/Discipline/DesigningArchitecture/QualityAttributes/tabid/64/Default.aspx
3. *Architectural Patterns*, from *The Open Group Architectural Framework* (2006). Available: http://pubs.opengroup.org/architecture/togaf8-doc/arch/toc.html
4. Abel Avram and Floyd Marinescu. *Domain-Driven Design Quickly* (2006). C4Media Inc., InfoQ.com Enterprise Software Development Community. Available: http://www.infoq.com/minibooks/domain-driven-design-quickly
5. Franc Stratton. *SOA Model-View-Controller (MVC)*. Available: http://francstratton.com/SOA.aspx
6. Paris Avgeriou and Uwe Zdun. *Architectural Patterns Revisited – A Pattern Language*. Available: http://www.infosys.tuwien.ac.at/staff/zdun/publications/ArchPatterns.pdf
7. Steve Burbeck. *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)*. (1992) Available: http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html
8. Martin Fowler. *Model View Controller*, from *GUI Architectures* (2006). Available: http://www.martinfowler.com/eaaDev/uiArchs.html
9. Derek Greer. *The Art of Separation of Concerns*, from *Aspiring Craftsman* (2008). Available: http://www.aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/
10. *Getting Started with Rails* (2010). Available: http://guides.rubyonrails.org/getting_started.html
11. Donald Brown, Chad M. Davis and Scott Stanlick. *Struts2 in Action*. (2008) Manning, p. 11.
12. Qiang Xue and Xiang Wei Zhuo. *The Definitive Guide to Yii 1.1* (2010). Available: http://yii.googlecode.com/files/yii-docs-1.1.7.r3135.tar.gz
13. *Cocoa Design Patterns*, from *Cocoa Fundamentals Guide* (2010). Available: http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html#//apple_ref/doc/uid/TP40002974-CH6-SW1
14. Josh Smith. *WPF Apps With The Model-View-ViewModel Design Pattern*, from *MSDN Magazine*(Feb 2009). Available: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx
15. Paul Krill. *MVC development gets a new life in mobile apps*, from *InfoWorld* (2011). Available: http://www.infoworld.com/d/developer-world/mvc-development-gets-new-life-in-mobile-apps-643
16. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. (1994) Addison-Wesley.