APPLICATION OF DESIGN AND ARCHITECTURAL PATTERNS IN AJAX PROGRAMMING

by

Paul Beaudoin

A THESIS

Presented to the Department of Computer and Information Science
and the Honors College of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Bachelor of Sciences

May 2007

An Abstract of the Thesis of

Paul Beaudoin for the degree of Bachelor of Sciences

In the Department of Computer and Information Sciences to be taken May 2007

Title: APPLICATION OF DESIGN AND ARCHITECTURAL PATTERNS IN AJAX PROGRAMMING

Approved: [illegible]
Professor Sarah Douglas

Design patterns have been a part of software engineering for nearly three decades. Ajax applications are a relatively new type of Web application that use client-side scripting languages in conjunction with methods for asynchronous data request to enable powerful applications with rich, dynamic, user-oriented interfaces. This paper addresses the question of whether the design and architectural patterns often used in other software fields are applicable and beneficial to Ajax development. Can the same patterns be applied? Are new patterns uniquely applicable? This paper attempt to answer these questions by examining 1) what design patterns are, 2) what Ajax is, and 3) how the former can be adapted to the latter. We find that many of the patterns popular in other realms have direct applicability to Ajax programming. We find also that there are opportunities for new types of patterns that are uniquely relevant to Ajax.

ACKNOWLEDGEMENTS

The author expresses sincere appreciation to Professor Sarah Douglas for her assistance in the preparation of this paper. Her advice has been invaluable. This paper has been a long time coming, and her patience through a few false starts is greatly appreciated. In addition, special thanks are due to Professor Henry Alley for participating in the committee and for his patience serving as my advisor throughout this effort. I also thank Professor Kent Stevens for joining the committee.

Table of Contents

List of Figures

1 Introduction

Websites like Google Maps[1], Kayak.com, and Amazon.com's A9 search engine[2] leverage a new collection of technologies and approaches we generally call "Ajax". Ajax enables rich, dynamic interfaces that fundamentally improve classic Web applications through the use of new, asynchronous communication channels and the promotion of rich, intelligent controls in the Web page.

Being an especially new and popular technology, a lot of Web Developers have scrambled to gain Ajax proficiency quickly. Best practices have yet to be firmly established and the notion of 'elegance'—already a nebulous concept—has little precedent in Ajax development. As a result, 'quick and dirty' solutions are more prevalent than manageable, extensible implementations. Design and architectural patterns have long been thought to aid software development by promoting proven, abstract solutions to common problems. As the Ajax approach is still maturing, an investigation of the application of patterns to Ajax development appears to be a fruitful area of inquiry.

In this paper, I will look at opportunities for the application of similar patterns to Ajax development. To lay the groundwork for specific analysis, this paper begins with a general description of design and architectural patterns as they have traditionally been applied to software development. An introduction to Ajax technologies follows this, which summarizes the history of technological development that lead to Ajax and provides a basic example application. Using an example Ajax application, we look at whether or not patterns can be applied naturally and beneficially to Ajax development. We look at whether or not the existing industry toolkits and frameworks provide support

1 See http://maps.google.com
2 See http://A9.com

to this end. Finally, we look at whether or not pattern application has a future with Ajax, and what such application we are likely to see.

2 Software Patterns

To begin, let us discuss pattern application in the software field in general. The *pattern* idea is deceptively intuitive. It is not difficult to define, but it is sometimes difficult to distinguish pattern approaches from alternatives. Let us look at what patterns are and where they came from to lay the foundation for a more detailed look at their application.

2.1 What's a Design Pattern?

We start with an examination of the peculiar popularity of Christopher Alexander's "A Pattern Language: Towns, Buildings, Construction." At face value, Alexander's book joins a vast library of architectural documentation extending several thousand years into the past. However Alexander's "A Pattern Language" embodied a fundamentally different perspective on architectural technique compared to his predecessors. The documentation efforts that preceded Alexander were comparatively pragmatic in nature, favoring the practical over the abstract. Alexander contended that the challenge of producing good architecture was more complicated than merely replicating form and function. He described a timeless "quality without a name" produced by good architecture—a quality not adequately captured in the popular language of his industry. His work produced a collection of "patterns" that described a "problem which occurs over and over again in our environment, and then describes the core of the solution to that problem" (Alexander, Ishikawa, & Silverstein, 1977, page *x*[1]).

1 This phrase occurs in the introductory chapter "Using this book", which is numbered with roman numerals.

As an example, Alexander's 88th pattern is the "Street Café". Alexander notes that a street café frequently serves as a quiet oasis on the edge of a busy path. The street café adds balance to a busy street by allowing one to "sit lazily, legitimately, be on view, and watch the world go by." (Alexander et al., 1977, page 437). One consequence of allowing an idle entity like a street café to tempt the edges of a fast-paced boulevard of strolling pedestrians is increased socialization. Healthy neighborhoods are supported by familiar neighbors, and one way to promote familiarity is by creating mediums for idle socialization (i.e., the café) immediately adjacent to active ways (i.e., the street). Alexander describes the street cafe as a pattern that accomplishes just that.

By defining about 250 such patterns, Alexander created a new language that architects could employ when addressing common challenges—like the challenge of how best to promote socialization in a commercial area.

2.2 Pattern Application in Software

Although "A Pattern Language" is one of the best selling books in architecture (Mehaffy, 2007), the book also strikes a chord with computer scientists, who find that his observations in the architectural discipline resonate with challenges in their own. In computer science and software engineering, there's a notion of *elegance* that's difficult to quantify and often expressed by example. Countless solutions exist for a given problem, yet frequently only a few produce the solution that will be optimally intuitive, flexible, and robust. Computer scientists found that Alexander's pattern approach could be used to identify these elegant practices at a high level of abstraction.

A distinction is frequently made between *design* patterns and *architectural* patterns.

Design patterns usually address a limited challenge like how two specific types of components should communicate. Architectural patterns on the other hand impose constraints on the application as a whole. The distinction is not important immediately to the discussion at hand except to demonstrate the different levels of granularity with which patterns can be applied. Let us look at design patterns, followed by architectural patterns.

2.2.1 Software Design Patterns

The seminal 1994 book by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* documented 23 design patterns in object-oriented[1] software engineering. Now in its 36th printing, the book follows the spirit of Alexander's work by laying down a common vocabulary of abstract solutions to common problems that arise in software development.

Let us consider the **Observer** pattern, which we will return to frequently throughout the paper. Observer is applied to situations where one or multiple objects—the *Observers*—wish to monitor changes in another object—the *Subject.* The pattern specifies that the Subject provide a method by which objects may register themselves as Observers of that object. Although the Subject maintains this collection of Observers, the Subject need not know anything about the Observers, with one exception: Observers must implement a single method ("notify update") by which the Subject can notify them of Subject changes. When a change occurs in the Subject that the Subject internally determines to be of interest to an object observing it, the Subject notifies all interested

1 Object-oriented programming is one of dozens of overlapping programming paradigms. This particular approach—perhaps the most popular at the time of this writing—prescribes thinking about application structure in terms of interacting "objects" that encapsulate logically related functionality and data.

Observers by calling their "notify update" methods. In this way, Observers learn that a change has occurred in the Subject, which they may subsequently reinspect. (See Figure 2.1.A for a simple example of this relationship.)

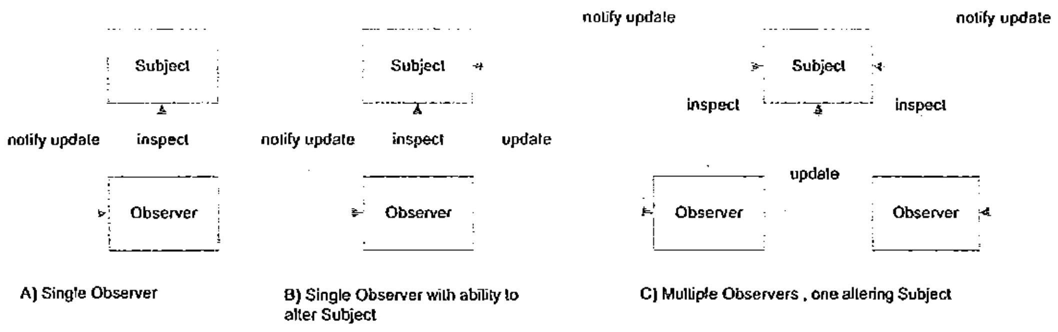


Figure 2.1: The Observer pattern can accommodate one or many objects ("Observers") that monitor a single "Subject".

Figure 2.1 represents three levels of complexity the Observer pattern can acquire. In figure 2.1.A, we see a single Subject is monitored by a single Observer. This is the simplest manifestation of the pattern. Figure 2.1.B shows a common variant of the pattern where the Observer wishes to alter the Subject in addition to monitoring it. When the Observer effects change on the Subject, the latter will notify all observing objects of the change—including the object that executed the change. In this way, the Observer is made passively aware of the effect its changes are having on the Subject. Figure 2.1.C demonstrates that the pattern allows multiple Observers to share a single Subject.

The Observer pattern is useful because it enforces a decoupled relationship between the communicating objects. The Subject need not know anything about the objects that observe it nor what they will do as a result of changes in the Subject. This makes it easy to add and remove Observers as needed. Additionally, it allows one to more easily alter the Observer without disrupting the Subject—and vice versa.

Observer exemplifies software design patterns by defining a generally applicable solution to a common problem. The solution is not associated to a specific language or environment. Like the other 23 patterns described by Gamma et al. in 1994, it merely advises a high-level component organization commonly found to succeed.

2.2.2 Software Architectural Patterns

Architectural patterns might be said to be a refinement of design patterns. Significant overlap is found between architectural patterns and design patterns, and the terms are sometimes used interchangeably.[1] Some patterns traverse these categories by being applicable as design patterns in one scenario, and just as applicable as architectural patterns in another.[2]

No discussion of patterns can escape invoking **Model-View-Controller**—or MVC, as it is known with equal familiarity among computer scientists. MVC emerged from some of the first experiments with graphical user interfaces in Smalltalk-80 (Gamma et al., 1995, page 4), It specifies a triad of subsystems:

- **Model:** The object or data that is the subject of the application
- **View:** One or more displays representing the Model
- **Controller:** The components directly **manipulable** by the user

For example, in a document editor such as Microsoft Word, the Model comprises the collection of raw data objects that stores text and formatting; the View comprises the

1 Gamma et al. refer to Model-View-Controller as a design pattern, suggesting that architectural patterns may be a subset of design patterns. (Gamma et al., 1995, page 4)

2 The Observer pattern—previously described—normally refers to two or more components within a system, but one can imagine an application whose top-down organization consists of two or more families of components that interact in much the same way.

objects responsible for producing an intelligible presentation of the Model on a user's monitor; and the Controller handles keyboard and mouse input.[1]

Organizing an application's components into the MVC triad can benefit application architecture by decoupling functionally disparate objects. This allows, for example, changes to occur in the application display—or View—without necessitating change in the Model. Conversely, changes in the Model can be made independent of the View. As Figure 2.2 demonstrates, View components share a relationship with Model objects similar to that shared by Observers and Subjects in the Observer pattern discussed above. The flexibility obtained by this organization frequently improves application quality by reducing maintenance costs and the complication of adding new features.

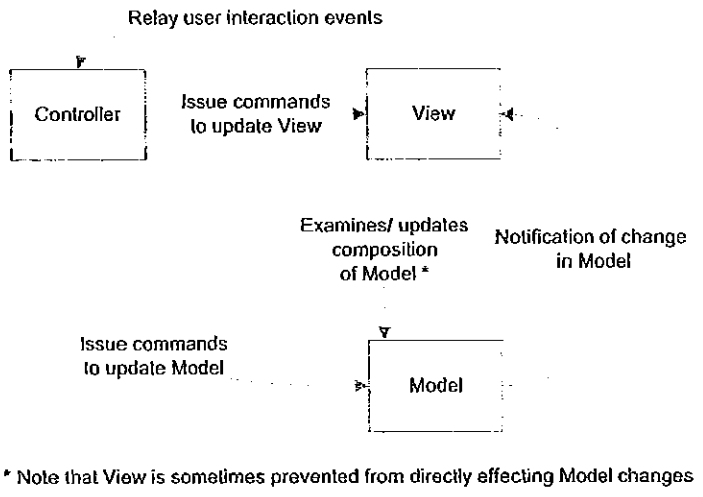

Relay user interaction events
Controller
Issue commands to update View
View
Examines/ updates composition of Model *
Notification of change in Model
Issue commands to update Model
Model
* Note that View is sometimes prevented from directly effecting Model changes

Figure 2.2: Model-View-Controller specifies the organization of application components into three subsystems.

The most important characteristic of architectural patterns is shared by design patterns: patterns describe abstract solutions to problems that happen over and over. By

1 Some additionally assign all functionality *not naturally assignable to other components* to the Controller, such as View management.

familiarizing oneself with the language of software patterns, one taps into a collective wisdom that enables the application of proven strategies—even to challenges one has never solved before.

2.2.3 Challenges to Pattern Application in Software

Not everyone appreciates the pattern movement started by Christopher Alexander in 1977. Design patterns are often criticized on the grounds that they are mere pedagogical exercises that would be more usable if laid down concretely as part of a language—or provided as a usable library. Design patterns are not code, after all. They can not be directly integrated into a program one is writing; they must be summoned and applied cognitively when one approaches common challenges. Critics claim that many of the challenges met in this way would be better served by real language support or libraries.

As an example, consider the Iterator pattern (Appendix A3.2 "Iterator Pattern"), which has been used to demonstrate that design patterns can sometimes formalize proven technique without providing the benefit of codified language support (Baker, 1992). Perl, for example, provided native language support for the Iterator concept for years before Java came along. Java's conscious omission of a language-level Iterator construct brought the Iterator pattern to the forefront because programmers no longer had a baked-in method by which to achieve Iterator functionality (Dominus, 2006). Programmers, in essence, had to apply the Iterator pattern anew every time the functionality was required as a result of a language insufficiency.

Looking even further back, Dominus points out that a frequent practice among machine language coders of the 1950s and 1960s was to designate a common block of code to be accessed in two or more places. Had there been a "patterns" buzz in the 1950s and 1960s, we might have talked about a "Subroutine" pattern. Subroutines are now an integral part of all modern functional languages.

Given this history, it is possible that design patterns serve merely to presage library or language-level codification. Ultimately, the important question is whether pattern application is a worthwhile activity—regardless of its end. If patterns benefit programs in lieu of library and language support—and if libraries and language enhancements serve often to realize the essential tenants of those patterns—patterns still benefit development in the interim.

However one feels about patterns personally, they appear frequently in software development and computer science theory. Patterns have become a major common language in computer science, since they allow us to speak at a high level about ways to approach challenges without concern for languages and platforms. We will return to patterns in the chapter that follows. For now, let us look at the set of technologies of greatest interest to this paper: Ajax.

3 What's Ajax?

Ajax is one of the latest acronyms to emerge from the acronym factory that is the tech industry. To understand the technologies involved—and the special buzz around this particular acronym—we need first to lay some historical groundwork.

3.1 History

Ajax relies on a number of Web technologies, detailed descriptions of which are beyond the scope of this discussion. Nevertheless, a brief history is important to illustrate how Ajax fundamentally departs from its technological roots. Figure 3.1 notes a few of the milestones covered in the historical summary that follows, beginning with the birth of the Web.

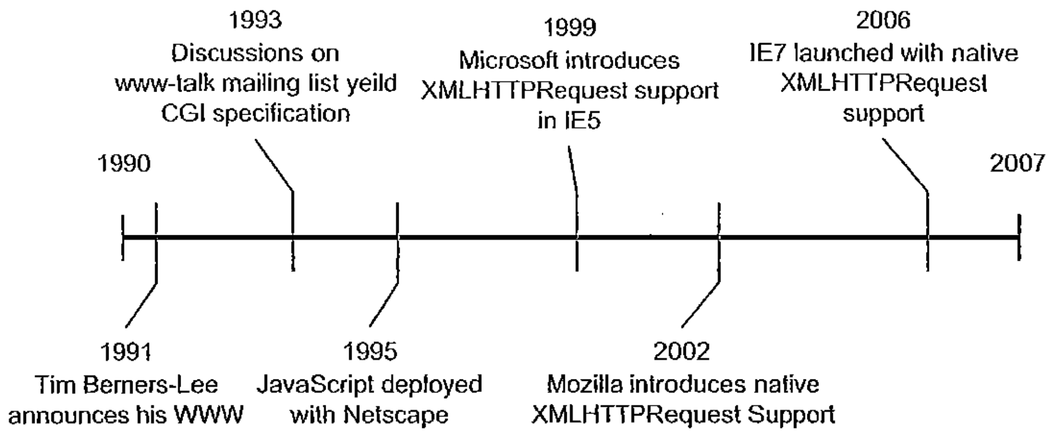

1993
Discussions on www-talk mailing list yeild CGI specification
1999
Microsoft introduces XMLHTTPRequest support in IE5
2006
IE7 launched with native XMLHTTPRequest support
1990
2007
1991
Tim Berners-Lee announces his WWW
1995
JavaScript deployed with Netscape
2002
Mozilla introduces native XMLHTTPRequest Support

Figure 3.1: Countless events contributed to the development of Ajax, but most important among them were 1) the birth of the Web, 2) the development of CGI, 3) the introduction of client-side scripting languages like JavaScript, and finally 4) the various support provided to programmers by browser vendors for executing asynchronous requests.

3.1.1 The Web

The Web arose from an internal project at CERN in Switzerland. Tim Berners-Lee worked as an independent contractor at CERN in 1980. One of his projects was to address the problem of the sharing and cross-referencing of information among physicists. Berners-Lee envisioned a new naming convention for hypertext references[1] "to allow links to be made to any information anywhere" (Berners-Lee, 1991).

Importantly, the Web's original intention was as a mechanism allowing documents written in HTML to reference other documents written in HTML on any Web server anywhere in the world. Core to this idea was the notion that Web servers set aside a "document root" as a repository for publicly accessible files. Any request submitted to the server for a document within that root folder would be returned.

1 Hypertext is the name given the convention of including links inside text documents to other documents. The concept precedes these events by almost 40 years. It first appeared in Vannevar Bush's Atlantic Monthly article "As We May Think," in which he described the *Memex*, a mechanical device capable of calling up comprehensive cross-referenced information from multiple mediums on demand. (Bush, 1945)

Server developers quickly identified new possibilities in this simple idea. In particular they noticed that a request for a file on a server need not refer to an existing file at all. A request is just a path[2] (e.g. /currency_converter), so why not let a program running on the server dynamically generate the document to be returned every time the path is requested? This breakthrough marked the beginning of server-side Web programming, which paved the way for the fifteen years of Web application development that followed.

3.1.2 The Birth of Client-Side Scripting

When one considers that Tim Berners-Lee had a lot of trouble generating interest in his World Wide Web in 1984, one finds the growth of the Web following 1990 astonishing. Early text-only browsers on limited platforms were replaced by multimedia capable browsers like the ViolaWWW browser in 1992 and Mosaic[2] in 1993. In addition to multimedia, some browsers began to support a new concept called client-side scripting. Client-side scripting allowed one to use a special lightweight language to program behaviors that execute on the client's computer alongside display of the web page. One could, for example, detect the user's browser and dynamically write content into the web page.

The rise of client-side scripting was a foundational achievement because it created a new symmetry between server and client. Both server and client now had the capacity for dynamic behavior. With a decent programming environment and a little creativity,

2 HTTP requests actually comprise quite a bit more information, but for the sake of the discussion at hand it suffices to think about them as containing just a path.

2 Mosaic became an early version of the Netscape browser still in production (although it retains only marginal popularity at writing). The Netscape code base was made public in 1998 as the open source browser Mozilla, an early predecessor to today's Firefox. (Metzger, 2007)

almost anything is possible. This was now true on both sides of the client-server divide.

3.1.3 Enter Ajax

Even with the explosion in availability of multimedia capable browsers and dynamic elements at both client- and server-sides, a glaring challenge prevented Web applications from achieving the rich, dynamic interactivity possible in desktop applications. That challenge was the document-modeled architecture of the Web. Recall that Berners-Lee designed the Web as a method to request richly interlinked documentation; he didn't foresee an interest in rich, dynamic interfaces such as Web email and Google Maps. Web applications emulated desktop application functionality within the context of the Web's original document-request model. A Web application had to be designed as a series of pages. Many activities one hoped to achieve with the Web application had to be mapped to a specific type of document request. Application functionality could exist at both ends of the client-server divide, but active communication between elements was constrained to large, jarring page requests that typically tore down the current page, replacing it with an entirely new page—an obnoxious and sometimes confusing experience for the user of the application.

Programmers began experimenting with ways to bridge the client-server divide and allow components on one side to communicate actively with those on the other. Early attempts employed brilliantly quirky concoctions of hidden frames, hidden images and cookies, as well as browser plugins like Java applets. Browser vendors noticed these efforts and began catering to the developers by providing new programming objects directly suited to the purpose of asynchronous server requests.

In February 2005, Jesse-James Garret published an article called "Ajax: A New Approach to Web Applications". The article gave a name to the collection of technologies that had been employed in varying degrees of elegance since client-side scripting was first introduced. Garret chose "Ajax" from the acronym AJAX, which stands for Asynchronous Javascript And XML. The term is usually written "Ajax" rather than "AJAX" to encourage people to think of the concept as a methodology rather than as the specific collection of technologies that the acronym represents (Mahemoff, 2006).

3.2 State of the Industry

To date more than 160 frameworks and toolkits have been developed that address the challenges of Ajax programming in some way (Mahemoff, 2006). These efforts share a goal of simplifying Ajax development by providing reusable components that solve common challenges. Even after significant standardization work by the W3C and IETF[1] standards bodies, many incompatibilities between browsers plague forward development. Toolkits address this by abstracting the difficult cross-browser work behind common simple-to-use interfaces, letting developers proceed with the real business of developing their applications. Many of these toolkits also provide convenient abstractions for asynchronous server requests.[2]

One major Ajax toolkit effort is GWT (Google Windowing Toolkit). The approach exemplifies the Server-Side Code Generation pattern described by Mahemoff, which

1 World Wide Web Consortium and Internet Engineering Task Force respectively. These groups strive to produce open standards recommendations for Web and Internet technologies. Tim Berners-Lee holds the director position at the W3C.

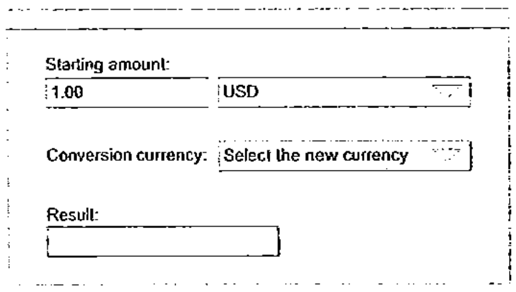
2 Most toolkits define something along the lines of an "AjaxRequest" object, which provides an intuitive interface for executing an asynchronous server request and parsing the response. The object is always built with browser capability in mind, so the technology employed depends on the browser it runs in. For very old browsers, some even fall back to the hidden frame technique of asynchronous request popular in 2000.

prescribes using a framework on the server to generate code for the client-side. Since the API exists entirely on the server-side, the developer is essentially free to ignore client-side concerns altogether and develop as one would a standalone desktop application.

Other major efforts include the Dojo and YUI (Yahoo User Interface) toolkits. These toolkits are written in JavaScript, which means that, unlike GWT, they provide support at the client side. Dojo and YUI demonstrate the growing appreciation for JavaScript as a powerful language for driving rich, complicated client-side applications.

3.3 Example Ajax Application: CurrencyConverter

For the purpose of demonstration, consider an application to convert an arbitrary amount from one major currency to another. Figure 3.2 demonstrates one possible interface.


Starting amount: 1.00 USD

Conversion currency: Select the new currency

Result:

Figure 3.2: The CurrencyConverter interface lets a user enter a starting amount and currency as well as a conversion currency to retrieve a result.

The functional requirements of our application follow:

1. *Amount*

   a) User can enter an arbitrary value into an *amount* field

   b) If the user enters an invalid value for *amount* without correcting, amount should revert to 1

c) The *amount* field should default to 1 at startup

2. *Starting Currency*

a) User can select a starting currency from a drop-down

b) The starting currency should default to USD

3. *Conversion Currency*

a) User can select a conversion currency from a drop-down

b) The default conversion currency at startup should be a null entry that reads "Select the new currency"

4. *Results*

a) Results should be displayed in a reserved *result area*

b) The *result area* should always reflect the current problem space:

i. If the two currencies and a valid amount have been entered, the result area should express the result.

ii. If the result is actively being computed, the result area should communicate that state.

iii. If no result is computable given the state of the controls (e.g. Conversion currency has null selection), the results area should be blank

5. *Accuracy*

a) Conversions should be computed at the server so that the most up-to-date currency rates can be used when computing conversions.

These requirements describe a very simple application, yet there are countless possible implementations using Web technologies. Let us see how Ajax improves the application.

3.3.1 The 'Just Make It Work' Approach

To illustrate the Ajax distinction, we will begin with an approach that does not use Ajax. Of the many possible implementations of CurrencyConverter, most will probably agree that the fastest route to requirements fulfillment lies in a straightforward Web application approach sans Ajax. Let's program like it's 1999.

The basic Web application approach produces a simple web page with a form as specified above. The form features rudimentary event detection to determine when the user has supplied enough information to perform a conversion. When such occurs, we cause the form to post to the server, which gathers the submitted values and returns the same page—this time with a result inserted into the result area.

This approach has at least two nice qualities; it is simple, and it relies on old technologies. Because Web browsers have remained largely backwards-compatible, the safest route to cross-browser stability has always been to code with yesterday's technology. This approach fulfills that, but not a lot else.

The 'Just Make It Work' version of CurrencyConverter exhibits many of the pitfalls of classic Web applications before Ajax. It places unbalanced focus on the server-side by treating the client-side as a mere presentation layer—ignoring the potential for dynamic interaction that the Javascript enables there. Furthermore, it produces an undesirable user experience by forcing a page refresh for each conversion. These shortcomings were notoriously common in Web applications before developers began experimenting with Ajax.

This implementation works, but we can do better.

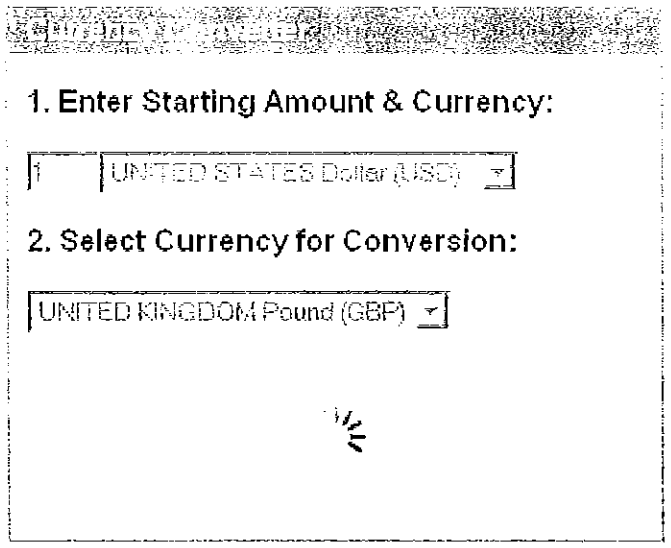
3.3.2 The Ajax Approach

Mahemoff calls Ajax a user-oriented phenomenon because it arose from a desire to improve the experience of Web application use (Mahemoff, 2006, page 5). Ajax does not enable tasks that were not—in some clunky way—possible before, but it does improve the experience of performing those tasks by speeding things up and reducing jarring and unnecessary interface reloading.

The Ajax form of CurrencyConverter uses the base Web application as a starting point. We then rewrite the code that affects the form submission so that it routes the computation request through a hidden asynchronous request to the server. While the application waits for the computation to return, the application communicates its state via a "spinner" animation. When the result is obtained, it is dynamically inserted by client-side code into the result pane.

This approach has a few advantages over the basic Web application approach. Most prominently, we remove the jarring form post and page reload that previously occurred for every conversion request. Secondly, the "spinner" animation used during computation communicates the "processing" state more naturally than the page reload could.[1] See figure 3.3 for a screen capture of the application in a processing state.

1 To be fair, with a little work, the same spinner animation could be used immediately prior to the page reload in the "Just Make It Work" version. The effect would be diminished, however, because the spinner would only appear until the point the page was visibly destroyed and reloaded. Furthermore, the CPU cost of a form post is often so great that the spinner may not render at all.


1. Enter Starting Amount & Currency:

1 UNITED STATES Dollar (USD)

2. Select Currency for Conversion:

UNITED KINGDOM Pound (GBP)

Figure 3.3: The Ajax implementation of CurrencyConverter enters a "processing" state when computing a new result.

You can interact directly with my Ajax implementation of Currency Converter at the following Web address:

```
http://thesis.nonword.com/currency_converter
```

4 Pattern Use In Ajax

Having acquainted the reader with the fundamentals of patterns and Ajax, let us try applying pattern concepts to Ajax technologies. Can pattern concepts—traditionally applicable to Architecture and later to software engineering—be applied to the relatively new field of Ajax Web application development? I will discuss the opportunities available in the technologies in general, as well as support for that effort in available toolkits. I will explore use of patterns in the CurrencyConverter Ajax application described earlier. Finally, I will assess the state of the industry in terms of its design and architectural maturity and summarize my feelings on where pattern application in Ajax may grow.

4.1 General Opportunities For Pattern Application

Recall that pattern application in software design was preceded by pattern application in building construction. Given the apparent flexibility of the *patterns* concept to traverse disciplines—and given the respect being gained by Javascript as an application programming language—it seems possible that pattern application is possible in Ajax programming as well. Indeed, as we will see, opportunities are present for pattern application at many levels of Ajax development. This pattern application can improve the overall shape of the implementation by employing proven strategies to new technology.

4.1.1 Design Pattern Application

As summarized earlier, the Observer pattern becomes useful in applications any time one object wishes to passively monitor another object. This manifests in a few aspects common to Ajax applications, notably in the relationships between display and data objects, control and display objects[1], and the relationship of AjaxRequest (or whatever object the application relies on to perform asynchronous requests) to the components that rely on it. These relationships are represented in figure 4.1 on the following page.

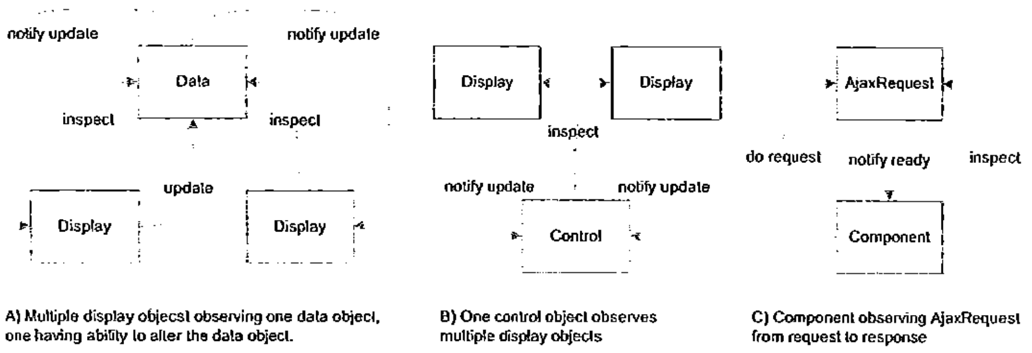

notify update
notify update
Data
inspect
inspect
update
Display
Display
A) Multiple display objecst observing one data object, one having ability to alter the data object.
Display
Display
inspect
notify update
notify update
Control
B) One control object observes multiple display objects
AjaxRequest
do request
notify ready
inspect
Component
C) Component observing AjaxRequest from request to response

Figure 4.1: The Observer pattern is found in multiple aspects of Ajax programming.

Ajax applications frequently feature rich compound controls whose state must be monitored by the Controller for changes. This View-Controller relationship is sometimes best mediated by application of the Observer pattern.

As an example of this View-Controller Observer relationship, consider a compound application control that enables a user to select a major US city and state from two drop-downs. (Figure 4.2) There being only 50 states and 12 US territories, populating the state drop-down is a simple matter. Populating the cities drop-down is not so easy since there are upwards of 680 major US cities and towns (Rarevu, 2007). Pre-filling the cities drop-

1 I refer generally to "display", "data", and "control" objects because these concepts are natural characterizations of components that arise in multiple patterns. In MVC, these categories refer, respectively, to View, Model, and Controller.

down with that number of options will render the drop-down cumbersome to use since the user will have to scroll past many cities before discovering the desired option. Furthermore, we wish to enforce data integrity at the interface level by ensuring that the selected city is consistent with the selected state. We choose therefore to delay populating the cities drop-down until after the user has selected a state. Since states contain, at most, only a few dozen major cities, selecting a city after selecting a state will be fairly easy for the user and will require a minimal amount of new data.

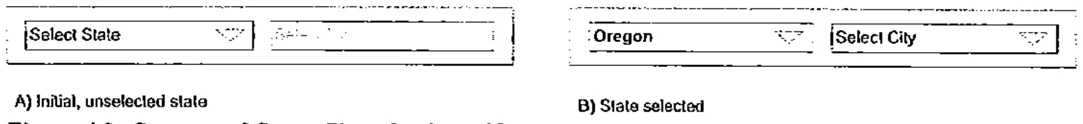


Figure 4.2: Compound State, City selection widget

We build the state drop-down as an observable object. Accordingly, any component interested in changes to the state drop-down need only register itself as an observer of that object. In this example, a Controller object performs that registration and waits for the user to cause the drop-down's state to change. When this occurs, the Controller is notified of a state change in the object it observes. The Controller decides to issue an asynchronous request to determine the valid cities for the selected state, which it uses to populate the cities drop-down. Since the Controller continues to be a registered observer of the state drop-down, further changes to the state drop-down selection by the user will trigger the same procedure; the second drop-down will be re-populated with the cities valid for the new state selection.

By applying the Observer pattern to this compound widget, we enforce a relationship between the View object and the Controller that frees the View object of the responsibility of data request and management. This sort of gain is characteristic of design pattern application. By separating logically distinct components and restricting methods of communication, we obtain a component relationship that favors extension and maintenance. We will see the same is true when we raise the scope of the pattern to the application level.

4.1.2 Architectural Pattern Application

There are a few opportunities to apply architectural patterns to Ajax development. One commonly applied architectural pattern has roots in the **Multitier Architecture** pattern. This pattern probably guides more Ajax application design than the pattern is given credit for since the pattern is a fairly intuitive organizational approach—and one suggested implicitly by the client-server divide. Multitier Architecture prescribes an application organization having multiple layers of component families organized by purpose. In a **Three-Tier Architecture**, the tiers are 1) User System Interface Tier ("Presentation"), 2) Process Management Tier ("Logic"), and 3) Database Management Tier ("Data"). Rules govern communication between tiers. For example, in a Three-Tier Architecture, components in the Presentation tier can only communicate with members in the Logic tier (Sadoski, 2000).

We can imagine a modified version of the Three-Tier organization that adds a fourth tier—a "Data Service Tier" that interfaces with the Data Tier on the server side. See figure 4.3 for a depiction of how the request for city data mentioned in the compound widget example above would occur in such an organization. The added Data Service Tier is not always necessary[1], but its use allows the logic concerning request validation to

1 A number of technologies attempt to provide direct—or somewhat direct—access to a server-side database via

remain separate from that of raw data retrieval.

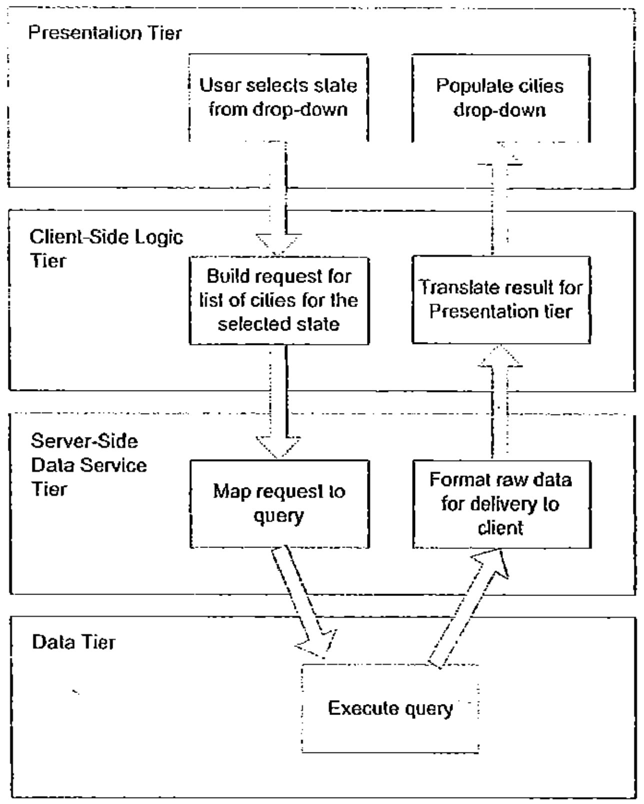

Presentation Tier
User selects state from drop-down
Populate cities drop-down
Client-Side Logic Tier
Build request for list of cities for the selected state
Translate result for Presentation tier
Server-Side Data Service Tier
Map request to query
Format raw data for delivery to client
Data Tier
Execute query

Figure 4.3: Multitier Architecture an an Ajax application

Patterns can be applied to Ajax either by focusing on the small-scale interactions of components using design patterns like Observer or by widening one's focus to the whole system as in the Multitier application example above. The examples given so far could be implemented by any capable Ajax programmer. We will now look at some of the support given to this effort by popular frameworks and toolkits.

client-side components. ASP.net's Data Binding technology is one such technology (Microsoft, 2007)

4.2 Pattern Acknowledgment By Toolkits

Many use Ajax toolkits like Yahoo's YUI, Google's GWT, or the Dojo Toolkit, as these toolkits provide useful cross-browser abstractions and general-purpose utilities for driving rich, dynamic interfaces and communicating with back-end services. But do these toolkits encourage use of design patterns?

4.2.1 Design Patterns in Toolkits

One common pattern-aware achievement of popular Ajax toolkits is the taming of the Web browser event model. Insufficient standards and proprietary implementations plague client-side event handling by forcing script writers to look several places at once when examining user events. For example, when a user clicks a keyboard key, Internet Explorer places information about the event in a different object than FireFox. Many toolkits— notably Dojo and YUI—solve these problems by building abstractions on top of the native event model. These abstractions typically emulate the Observer pattern. An object that wishes to be notified when a click event occurs on a button with id "button1" can use YUI's Event utility to register a callback function:

```
YAHOO.util.Event.addListener("button1", "click", callback);
```

A global event handler will fire the supplied callback function any time a click event occurs on the button.

The event subscription mechanism provided by YUI and other toolkits is a strong vote of support for the Observer pattern. Its formalization is a relief to developers familiar with what can go wrong when scripts compete to overwrite element event callback handlers. Previous to widespread use of Event utilities like those mentioned above, programmers often wrote code that directly assigned the callback as a property of the observed object. Consider this code, which might seem equivalent to the code given above:

```
document.getElementById("button1").onClick = callback;
```

This code is almost sufficient, but it enforces a one-to-one relationship of Subject and Observer. What recourse does a second Observer have if it wants to be notified of click events on the button? If the second object uses code similar to that above, the original Observer will cease to observe.

The benefits of Observer pattern acknowledgment in Ajax toolkits is not simply practical. Use of these pattern-modeled methods familiarizes programmers with the patterns represented, encouraging programmers to apply those patterns in other situations.

4.2.2 Architectural Patterns in Ajax Toolkits

Few Ajax toolkits claim to be architectural in scope—probably because at writing a competition is waging among vendors for leading community acceptance. This competition obliges toolkits to be somewhat lightweight and pluggable. If a toolkit were to take on architectural framework aspirations, the learning curve might be prohibitively steep. So for the moment, most Ajax toolkits embody basic design patterns rather than all-encompassing architectural patterns.

One notable exception to this is Freja. Freja claims to be an Ajax framework designed around the Model-View-Controller pattern. Freja employs aptly named Model and View classes that allow programmers to load XML and XSL documents asynchronously. The object/procedure that instantiates the Model and View objects serves as the Controller. The overall shape of a Freja application is dictated by the Freja framework—everything from the methods by which the application display is manipulated to the way the application data is represented (Savarese, 2007).

Freja exemplifies the power of patterns by applying one of the oldest known architectural patterns (MVC) to Ajax, one of the newest popular programming technologies. The fundamental power of MVC is as relevant to Smalltalk-80 in the 1970s —a system providing one of the first graphical user interfaces—as it is to Freja and the modern, real-world Ajax Web applications that rely on it.

4.3 Testing the Idea of Patterns With CurrencyConverter

We' now narrow our focus further to explore another case of demonstrated Ajax pattern application. The CurrencyConverter application introduced in 2.3 does more than demonstrate Ajax capability. It also realizes an MVC approach to Ajax application design.

4.3.1 MVC Organization in CurrencyConverter

CurrencyConverter consists of 7 principal components: 5 Model classes and 1 each of View and Controller classes. See figure 4.4 (following page) for a high-level view of the general relationships of these components with respect to how they communicate with one another.

The solitary component comprising the View subsystem—the "View" class—is concerned solely with the representation of the Model on the screen. The View class produces the containers, form inputs, and *results* area that express the current problem space.

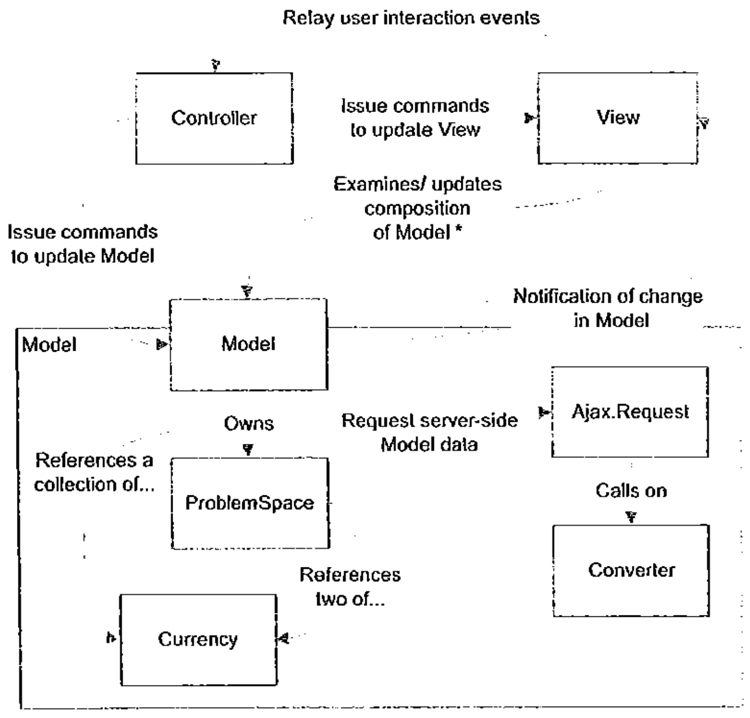

Relay user interaction events
Controller
Issue commands to update View
View
Examines/ updates composition of Model *
Issue commands to update Model
Notification of change in Model
Model
Model
Ajax.Request
Owns
Request server-side Model data
References a collection of...
ProblemSpace
Calls on
Converter
References two of...
Currency

Figure 4.4: CurrencyConverter demonstrates an MVC component organization with one component fulfilling each of the Controller and View subsystems and a cluster of components fulfilling the Model subsystem.

Similar to the View subsystem, one solitary "Controller" class comprises the Controller subsystem. The Controller is the main thread of execution and is responsible for initiating creation of Model and View objects. In this implementation, the Controller also serves as a dispatcher for user input events. For example, when the value of a form input changes, the View object representing that form input immediately and unconditionally relays that event to the Controller for consideration.[1]

The Model subsystem comprises five components on server- and client-side. I decided early that the Model element of the application should include a "ProblemSpace" component, which contained the current state of computation. This reflects a choice to define the "subject" of the application as comprising more than mere domain data concerns; The "subject" of CurrencyConverter also includes the user's workspace. The ProblemSpace thus contains the starting and conversion currencies as well as the entered amount. A fourth property of the ProblemSpace stores the *result.* The ProblemSpace is designed to be self-adjusting such that updates to the input parameters immediately effect change on a *result* property—also contained within the ProblemSpace.

For full implementation details, see Appendix A2: "CurrencyConverter MVC Breakdown."

4.3.2 Practical Implications of Pattern Use in CurrencyConverter

The choice to use MVC when I designed CurrencyConverter encouraged the program to take a unique shape. MVC encourages grouping components by their tendency to vary. One of the measures of an architecture's robustness is the ease with which it can adapt to accommodate new functionality. Say, for example, that we add an additional View that subscribes to the Model to display the country flags of the selected

1 Note that the choice to have the View blindly relay user interaction "upward" to the Controller is an effort to approximate a purist implementation of MVC in spite of the limitations inherent in pre-built form elements. The purist view of MVC dictates that all user interaction is detected and processed by the Controller. Modern user interface toolkits like Java Swing frequently provide pre-built form controls that encapsulate View, Controller, and Model functionality in a single, immutable entity. The pre-built controls defined by the HTML specification engender the same challenge. These elements contain within them control over their display as well as the privilege of first notification regarding user interaction. The method described here is one way to remain optimally true to the MVC ideal in spite of the constraints imposed by these toolkits.

currencies. Since 1) user interaction is bubbled directly to the Controller, 2) meaningful changes are subsequently applied to the ProblemSpace object in the Model, and 3) changes to the Model are broadcast to all subscribed views, it follows that implementing this "flag view" would be fairly simple. Such a new View would simply subscribe as an Observer of the Model and then display the country flag appropriate for the selected currencies, as reflected by the ProblemSpace. By a similar means, one could add two clickable maps by which major currencies could be selected by region. The new View need only bubble the appropriate user interaction events to the parent Controller object to effect the change on the ProblemSpace. Clicking on England in the second map would fire an "updated currency2" event, passing "GBP." Thanks to the many-to-one Observer relationship shared by the two views with the Model, selecting currencies via one View would effect immediate changes on the other control—a desired effect since it enforces the relationship of the two methods presented to the user for currency selection.

Another benefit of MVC application in CurrencyConverter is the governance placed on communication by members of one subsystem with those of another. The flavor of MVC chosen does not, for example, permit the View to directly alter the Model. User interaction intended to update the Model must first pass through the Controller, which performs validation by ensuring that the new value is meaningful (i.e., an integer greater than 0). The responsibility of validation could have been assigned to the ProblemSpace object—or to the Model itself—however, two considerations make this choice less ideal. Firstly, allowing the Controller to perform user input validation is closer to the original spirit of the Controller subsystem. Secondly, the user input broadcast system provides a unified interface for user input in general; User input can be intended for other purposes besides Model updates after all. Suppose one of the views contains a button to close the application. Such an event is best mediated by a single central entity—such as the Controller embodies.

Finally, there's the benefit of a common language. Web applications of even moderate size frequently require many collaborating developers. Telling a developer that CurrencyConverter is an MVC-modeled application communicates a lot about the structure of the application, which reduces the challenge of working in foreign code.

There are notable negative consequences to applying MVC to CurrencyConverter. One is the simple matter of code size and development time. CurrencyConverter is an intentionally simple application with trivially simple functionality. It would not be difficult to develop a functionally identical application in less time, with less code. One could, for example, omit the ProblemSpace object, which abstracts the data represented by the form away from the form elements themselves. Values to be used in the currency conversion could be collected directly from the form inputs and posted to the server for a result. The same component could subsequently parse the returned response and directly update the display. Indeed, for such a simple application, this approach may be perfectly practical. The loss of architectural and component communication robustness—benefits noted above—would be compensated by a gain of simplicity.

One can imagine, however, the issues one will have if new functionality is added to the application. Suppose one wants to add the flag view described earlier so that two View objects interface one Model object. Without a ProblemSpace object, user interaction via the View widgets would have to directly trigger the currency conversion. A procedural[1] approach could employ a central "compute" function that accepted three arguments: currency1, currency2, and and amount. The function would validate the three parameters, compute a result, and update the displayed result (altering the View directly). Note that without special handling, the two Views would fall out of sync the first time one of them was used to select a currency, because the two Views no longer share the common ProblemSpace abstraction. One can imagine how similar problems would quickly hinder manageability and make the addition of more Views increasingly difficult.

1 Procedural programming is a paradigm that favors using global procedures—or "functions"—without object-orientation.

5 Conclusion

Although young, pattern theory in Architecture has profoundly changed the way we think about documenting technique. Patterns implore us to overlook the finer points of execution to focus on the essential characteristics of successful practice. This simple idea finds an audience among computer scientists who are familiar with the pursuit of an intangible *elegance* not easily quantified or guided.

At the same time, Web application development is exploding. The initial fervor around server-side programming produced a wealth of pattern application and frameworks. When methods for asynchronous client-server communication surfaced around the turn of the century, a similar fervor erupted in client-side programming. Many of the cross-browser compatibility issues initially apparent have been resolved through standardization work and the development of JavaScript toolkits. Ajax arose as a collection of technologies employed to bridge the client-server gap. This new communication channel enables rich, frequent communication between client and server, making rich, dynamic Web applications possible.

Given the suitability of pattern theory to other software fields, it seems natural to apply pattern concepts to Ajax. We find that this effort is not only possible; it has already begun. Many toolkits and frameworks already employ pattern concepts including direct implementations of architectural patterns (e.g. Freja). Application of patterns ideas generally improves Ajax development by promoting component grouping and reducing confusion through restrictive component communication channels. This application can come at a cost of added development time. Furthermore, it is not yet clear how beneficial larger architectural patterns will be to Ajax applications. We see efforts to employ them (e.g. Freja and CurrencyConverter), but use of architectural patterns in very large, complicated applications is yet unseen.

We have seen that use of design and architectural patterns in Ajax development is both possible and prevalent. We now look at the future of pattern application in Ajax. Firstly, we ask whether or not pattern application efforts can coexist with the codification efforts embodied by frameworks and toolkits. Secondly, we look at the one area of growth uniquely available to Ajax—server-inclusive architecture patterns.

5.1 Patterns Versus Toolkits & Frameworks

Given the success of Ajax frameworks and toolkits, one might ask if it is better to focus efforts on toolkit development or pattern application. In a recent count, over 160 toolkits and frameworks were found that address the challenges of developing rich Ajax applications (Mayemoff, page 567). Although, as we've seen, these frameworks and toolkits can promote pattern use through pattern-oriented APIs, the goal of frameworks and toolkits is sometimes thought to run opposed to that of patterns. After all, frameworks and toolkits provide ready-to-use, pluggable solutions to common challenges, whereas patterns merely provide advice. Given this apparent conflict, one might wonder whether it is better to focus on patterns or on codifying pattern ideas in directly applicable frameworks and toolkits.

Frameworks and patterns really represent two paths to the same goal. That goal is the promotion of code that is strong, maintainable, and extensible through the reuse of proven technique. The technique encouraged by frameworks and toolkits is directly applicable whereas the technique provided by patterns is more cognitive in nature.

One might choose a framework solution over a pattern approach for an obvious reason: Using pre-written code saves development time and increases the probability of correctness. However integrating a pre-written solution via a framework or toolkit is not always practical. Doing so may increase the overall size of a program to an unacceptable level, for instance. Often, one only hopes to acquire one aspect of a library, but use of the part necessitates use of the whole. Another reason one might choose to implement a pattern manually rather than rely on framework/toolkit support is that they wish to maximize the efficiency of the implementation by coding the pattern into the piece in a tightly integrated manner not possible with pre-built code.

Finally, it is worth keeping in mind that the Ajax development industry is young. The term *Ajax* still referred solely to an industrial detergent as recently as 2005, and the collection of technologies that it served to represent preceded that date by perhaps five years. We are very much still figuring out what *elegance is* in Ajax. The frameworks and toolkits writers still disagree about what shape an Ajax application should take. Although Ajax developers can join this discussion by using available frameworks and toolkits, they can contribute arguably more by looking outside the Ajax arena to the abstract solutions demonstrated in the language of software patterns.

But pattern application is not purely backwards-looking. New patterns can arise to address new industry challenges.

5.2 Server-Inclusive Architectural Patterns

One possible area of growth for future pattern application in Ajax is the development of server-inclusive architectural patterns. Architectures that span the client-server divide do not frequently appear in discussions of Ajax design. This fact may owe much to the common perception that Ajax is a user-interface-centered technology. Indeed, Ajax arose out of efforts to bring dynamic behaviors to flat user interfaces, so it follows that Ajax might be considered a mere extension of those early efforts. The Ajax application, in this view, is the collection of components executing in the user's browser on the client-side. Server-side components are frequently excluded from the model as external *services* that the client-side application periodically calls upon.

Yet it is also possible to think of the Ajax application as comprising both client- and server-side concerns. Originally, the focus of Web applications was the server-side. With the standardization of JavaScript and associated XMLHTTPRequest technologies in recent years, the same fervor has been raised for client-side activity. Now that rich, communicating components are possible on both sides of the client-server divide, one might ask whether or not a holistic client- and server-inclusive approach is possible. Will we see such architectural patterns in the future?

Traditionally, most pattern application has been confined to the organization of components within a single program on a single platform. Industry support for a client-server inclusive architecture therefore seems unlikely. When an Ajax architecture is considered, focus is typically given to the cloud of components on one or the other side of the divide.

Why distinguish between client- and server-side components in an Ajax application? There are clear differences dividing the two. Obviously client- and server-side components can execute at a great geographical disparity—potentially joined by thousands of miles of buried fiber optic cable. Additionally, there may be thousands of computers executing client-side components, yet server-side instantiations frequently occupy a single centralized server. But these differences are not barriers to inclusive architectures; they are unique characteristics of the Ajax application, which can be thought of as executing in a single composite client-server environment.

Client-server inclusive architectural patterns seem inevitable for Ajax for the following reasons: 1) Ajax developers frequently engineer components on both sides of the client-server divide; 2) New methods of client-server communication enable richer and more frequent client-server communication; and 3) Ajax applications are only likely to increase in complexity, and complex applications are best tamed through high-level thinking such as embodied in pattern application.

One client-server inclusive Ajax pattern is provided in Appendix A1: "Client-Server Redundant Objects Pattern." This new pattern enjoys admittedly little industry support, yet it should approach a proof of concept for client-server inclusive architectural patterns.

5.3 Closing Remarks

It is an exciting time for Web application development. Modern browsers provide largely standardized mechanisms for executing asynchronous calls to the server. JavaScript toolkits provide reliable abstractions on top of browser incompatibilities. The course is now cleared of the largest obstacles, allowing programmers to develop both sides of the client-server divide without constraint and to bridge those environments with a new frequency and richness thanks to XMLHTTPRequest support. The time may be ideal to take a step back and survey Ajax development practice using the collective wisdom of the computer science community. Nowhere is that wisdom more accessible nor more generally applicable than in the vocabulary of design and architectural patterns.

6 References

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A pattern language: Towns, buildings, construction.* New York: Oxford University Press.

Baker, H.G. *Signs of weakness in object-oriented languages.* Retrieved January 29, 2007 fromhttp://home.pipeline.com/~hbaker1/Iterator.html

Berners-Lee, T. (1991). Posting to alt.hypertext 6 August, 1991. Retrieved May 6 2007 from http://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt

Bush, V. (1945). *As we may think.* Retrieved on March 5 2007 from http://www.theatlantic.com/doc/194507/bush

Dominus, M. (2007). *Design patterns of 1972.* Retrieved on January 29, 2007 from http://blog.plover.com/prog/design-patterns.html

Fielding, R. (2000). Architectural styles and the design of network-based software architectures. Retrieved March 24, 2007 from http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*, Reading, MA: Addison-Wesley.

Mahemoff, M. (2006). *Ajax design patterns.* Sebastopol, CA: O'Reilly Media, Inc. See http://ajaxify.com

Mehaffy, M. (2004). *Towards a new science of architecture, and a new architecture of science,* KATARXIS No 3. Retrieved April 28, 2007 from http://www.katarxis3.com/Review_Nature_Order.htm

Metzger, H. *Netscape history.* Retrieved May 6 2007 from http://www.holgermetzger.de/Netscape_History.html

Microsoft Corporation. *Asp.NET data binding overview.* Retrieved May 1 2007 from http://support.microsoft.com/kb/307860/

RareVu, *Rarevu Personalized Travel Site.*Retrieved April 25 2007 from Internal database of major airports. See http://rarevu.com

Sadoski & Comella-Dorda , *Three tier software architectures.* Retrieved January 11, 2007 from http://www.sei.cmu.edu/str/descriptions/threetier_body.html

Savaresa, C., Freja. Retrieved May 1 2007 from http://www.csscripting.com

Appendices

A1. Client-Server Redundant Objects Pattern

A1.1 Intent

Create a relationship between data access objects on the client- and server-side that maintains compositional synchronism by automating the generation of client-side class declarations.

A1.2 Motivation

We like building lightweight objects to interface raw data objects. These objects provide a simple, intuitive interface to a confusing, flat data pool as well as a way to think about domain data in a truly object oriented manner. Since the objects are intentionally lightweight, the added memory use is minimal. Our appreciation for lightweight data objects extends to both sides of the client server divide since the languages at both ends natively support the object-oriented paradigm. We find that we often code the same lightweight interface objects twice—once on the server side, and once on the client side—because components at both ends need to store and manipulate instances of the classes at both ends. Since these lightweight objects represent the same data, the class declarations employed on the client and server are semantically similar.

The Client-Server Redundant Objects pattern describes one approach we can take to save work and formalize the relationship of like classes across the client-server divide.

Not all objects on the server side will have a natural client-side translation—a property we will call "JavasScript castable." Indeed, we are likely only interested in the lightweight objects used to interface raw domain data. Furthermore, we are probably only interested in the objects of interest to the user interface. We define a mechanism to identify JavaScript castable objects by interface, package, folder, or any other well-defined rule. We then use reflection[1] to generate a class declaration in JavaScript, copying all properties, constructors, and methods as best as the reflection tools allow. This JavaScript output is inserted into a server-generated JavaScript library that can be included in the Web application together with the rest of the JavaScript Model code.

A1.3 Applicability

Use the Client-Server Redundant Objects Pattern when:

- a set of domain data can be simplified by lightweight interfacing classes
- the domain data needs to be accessed/manipulated on both server and client
- changes to the domain data require occasional/frequent changes to the lightweight interfacing classes (This assumes an implementation such as described in the RESTful Service discussion that follows.)

A1.4 Participating Patterns

Server-Side Code Generation

Overview: This pattern describes efforts to minimize developer overhead by generating all HTML, CSS, and Javascript at the server-side via one of the

1 Reflection is a convention of advanced OO languages like Java and PHP that enables one to programatically inspect the composition of instances and Class objects. Through reflection one can, for example, inspect Class to identify all of the publicly available methods.

growing number of frameworks available (Mahemoff, 2006, page 275).

Relationship: Although the pattern describes the generation of JavasScript by server-side components, the similarities end there. The pattern bears a sharp difference of philosophy with that of Client-Server Redundant Objects. The purpose of the former is to remove Javascript design from the equation altogether, freeing the developer to focus on the application from a server-side perspective. In doing so, naturally the developer loses direct control over the client-side code. With JavaScript undergoing a renaissance among developers, and new methods for improving user interface at the client-side arising every day, Server-Side code generation likely benefits the developer more than it does the user. Client-Server Redundant Objects, on the other hand, merely reflects a portion of the server-side down to the client. The developer is free to script those components at the client-side in any sophisticated way he/she chooses.

RESTful Service

The REST concept was developed by Roy Thomas Fielding at UC Irvine in 2000. REST—or Representational State Transfer—includes a number of guidelines for deploying Web services. Primary among them is that URLs be treated as unique indicators representing objects or collections of objects exposed by the server. The specification calls for a common language by which HTTP request methods are assumed to relate to objects. Issuing an HTTP Get request at a URL is presumed to return a view of that object. For example, issuing a GET at http://nonword.com/projects/1 should logically return information regarding my project number 1. Since POST calls are specified by REST to indicate a desire to change the composition of the object posted to, issuing a POST at the same URL will attempt to update the object. (Security concerns naturally enter the picture pretty quickly.)

REST need not necessarily be involved in Client-Server Redundant Objects, but I bring it up to suggest that REST concepts could be employed to extend the pattern's idea. Client-Server Redundant Objects is primarily a Development pattern in that it mostly benefits the development of the application without bearing directly on the runtime features. Suppose, however, we were to create a set of REST compliant URLS to retrieve JavaScript snippets containing server-generated object declarations. In the CurrencyConverter application, by this mechanism, the method one could use to dynamically load a currency object would be to issue a GET call to http://thesis.nonword.com/currency_converter/dao/currencies/GBP. Imagine the elegance if this call returned the following JavaScript code snippet:

```
new Currency('Great Britain', 'Pound', 'GBP');
```

Note that this particular approach strongly relates to Mahemoff's "On-Demand JavaScript" pattern (Mahemoff, 2006, page 122).

A1.5 Consequences

Applying the Client-Server Redundant Objects pattern can yield an architecture that spans the client-server divide. Components on either side are bound logically by a common set of lightweight data access objects. Although these objects are not technically shared at run-time, their static relationship should nevertheless improve the top-down architecture of the system by automating a redundancy that previously had to be manually maintained.

This approach has notable shortcomings. For one, reflection capabilities are often incomplete in server-side programing languages. In PHP5, for instance, methods exist to derive a class object from a class instance, and to iterate over the various methods and properties defined by that class, but fine-grained analysis of method bodies is not possible at writing. This means that when methods are translated from PHP5 to JavaScript class methods, the method body must be intuited from its name. *Set* methods can, for example, be identified by a combination of 1) the method name beginning with "set" and 2) the method taking just one predictably named parameter. The composition of the "equals" method can also be intuited by assuming that the same parameters passed into the constructor can also be used to determine equality. These assumptions are not unreasonable if the programmer remains mindful of the constraints and exploits the opportunity to override improperly generated code.

A2. CurrencyConverter MVC Breakdown

The seven principal components that compose the application belong to one of the three subsystems defined by the pattern—Model, View, and Controller. Figure A2.1 provides a class-level breakdown of the system.

Communication between components in one subsystem with those of another is kept to a minimum—and constrained to the channels of communication identified by the pattern. Each subsystem, as implemented by the CurrencyConverter application, is summarized below.

A2.1 CurrencyConverter Controller Subsystem

This subsystem represents the main "thread" of execution in the application. The Controller is the first application-specific component to be created in the application environment. The Controller is responsible for creating an instance of a Model subsystem. The Controller also creates and manages one or many View subsystems, providing each with a reference to the Model so that the former can subscribe to the latter via the Observer pattern.

The **Controller** class of components contains only one instantiated component in the CurrencyConverter application. Accordingly, the single component is named "Controller" and comprises a reference to a model, a collection of View subsystems, and a method by which the user interaction that is intercepted by View subsystems can be bubbled up to the Controller.

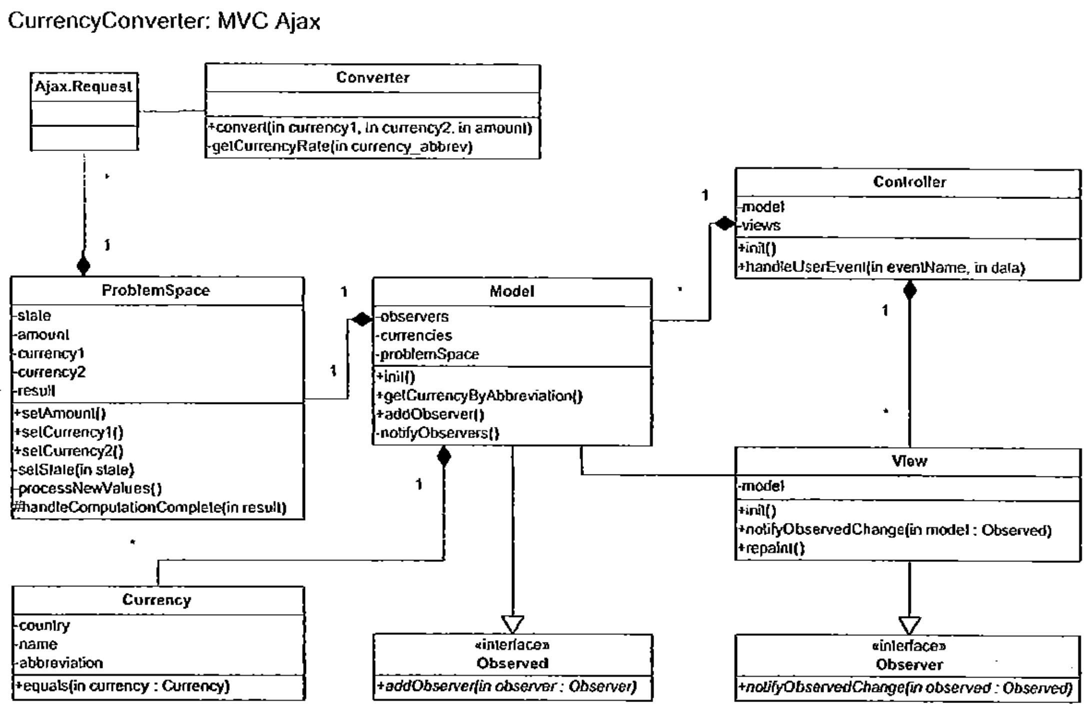

CurrencyConverter: MVC Ajax
Ajax.Request
Converter
+convert(in currency1, in currency2, in amount)
-getCurrencyRate(in currency_abbrev)
Controller
-model
-views
+init()
+handleUserEvent(in eventName, in data)
ProblemSpace
-state
-amount
-currency1
-currency2
-result
+setAmount()
+setCurrency1()
+setCurrency2()
-setState(in state)
-processNewValues()
#handleComputationComplete(in result)
Model
-observers
-currencies
-problemSpace
+init()
+getCurrencyByAbbreviation()
+addObserver()
-notifyObservers()
View
-model
+init()
+notifyObservedChange(in model : Observed)
+repaint()
Currency
-country
-name
-abbreviation
+equals(in currency : Currency)
«interface»
Observed
+addObserver(in observer : Observer)
«interface»
Observer
+notifyObservedChange(in observed : Observed)
1
*

Figure A2.1: CurrencyConverter Static Class Diagram

A2.2 CurrencyConverter Model Subsystem

The Model represents the subsystem of components that exclusively provide access to the data and problem domain. Typically, Model components include Data Access Objects (Appendix A3.1 "Data Access Object Pattern") as well as the data store itself.

The CurrencyConverter application Model subsystem contains five major components:

Converter: This is the server-side component interfacing the server-side data store.

The component accepts requests to convert a given amount from one currency to another, and returns the result.

Ajax.Request: This is the bridge between the sole server-side component—the Database—and other Model components. The Ajax.Request object is a transient object created once to perform a single HTTP request on the Database whose response is returned via callback to the initiating object.

ProblemSpace: The ProblemSpace component encapsulates the current state of computation within the application. Information such as the user's desired conversion amount and selected currencies are stored here. Access to ProblemSpace properties is constrained to Set methods, which ensures that the ProblemSpace component can update its own state internally in response to updated properties. For example, when the amount and starting currency are recorded in the ProblemSpace, application of a new conversion currency will trigger the ProblemSpace to internally update its fourth property—the result. Since this method relies on an external component to compute said result—and because that component executes asynchronously to the ProblemSpace—the ProblemSpace maintains a 'state' property to indicate whether or not its properties are consistent (i.e., internal state is consistent when the stored result is correctly computed given the stored amount and currencies). When sufficient user input is gained via the View, the ProblemSpace enters a "processing" state while the new result is computed and returned from the Converter component.

Currency: Currency objects represent single instances of international currencies. In CurrencyConverter, these objects contain only relatively permanent data—country, name, and abbreviation—leaving more volatile data like currency value at the server-side where changes at the data source can be more-quickly acknowledged. (Notably, without this distinction, I would have no need for Ajax.) The specification of Data Access Objects such as the Currency class occurs frequently in Ajax programming where large and/or highly dynamic objects on the server side are represented by lightweight "display" objects that are not sufficient on their own for business processes.

Model: The so-called Model component that belongs to its namesake MVC subsystem provides the sole point of contact for all components under the Model umbrella. The Model manages the relationships specified by the Observer pattern by providing a means for Observers to subscribe to changes in the Model. When such a change occurs (e.g. a new result is computed in the ProblemSpace), all listening Observers are notified of the change. (In this implementation, only one View object exploits this interface.)

A2.3 CurrencyConverter View Subsystem

Classically, the View class of components in the MVC pattern isolates those objects solely concerned with representing aspects of the Model to the user. Modern implementations of MVC must adapt this idea to allow some degree of Control responsibilities to occur in View elements. This occurs because many of the toolkits relied upon by View components provide widgets that pack Model, View, and Controller into a single immutable object. Modern MVC implementations work around this—and, arguably, maintain the spirit of MVC—by allowing Control behaviors to occur in View components, but only via a restrictive broadcast mechanism. One such implementation is taken up by the CurrencyConverter.

One View component occupies the View subsystem. Although this View implementation contains the most code, it is fronted by one of the simplest interfaces owing to the Observer pattern-modeled relationship it shares with the Model. The View learns about changes to the Model when notified via View's notifyObservedChange method (required by the Observer interface). When such occurs, the View reinspects the Model and selectively updates its displayed effects accordingly.

The View contains a number of form elements obtained via the Web browser's native factory methods. As noted above, use of controllable display widgets obtained from modern toolkits poses a problem in that user interaction with said widgets (classically a responsibility of the Controller) must be handled by the View. This apparent issue is overcome—or arguably minimized—by "blindly" bubbling selected user interaction events to the Controller. User interaction events are represented by two parameters: a name and a value. When the user selects British Pounds as the conversion currency, an "updated currency2" event is fired with the data "GBP." Four form elements comprise the CurrencyConverter interface. A handler attaches to each form element when it is created to facilitate gathering and broadcasting the field data unparsed to the Controller for consideration. Only upon arriving at the Controller is the data parsed for validity and selectively used to update the ProblemSpace in the Model.

Note also that this broadcast behavior lays groundwork for additional Views. Say for example that we add an additional view that subscribes to the Model to display the country flags of the selected currencies. Since 1) user interaction is bubbled directly to the Controller, 2) meaningful changes are subsequently applied to the ProblemSpace object in the Model, and 3) changes to the Model are broadcast to all subscribed views, it follows that implementing this "flag view" would be fairly simple. Such a view must simply subscribe as an Observer and then display the country flag appropriate for the selected currencies, as reflected by the ProblemSpace.

A3. Related Patterns

A3.1 Data Access Objects Pattern

Data Access Objects Pattern specifies the creation of objects whose sole purpose is to interface raw data sources. A single object typically represents a single logical domain entity. The object provides a unified interface to the entities that use the represented data. When changes to the object occur, the object may personally handle the commitment of those changes to the data store.

A3.2 Iterator Pattern

The Iterator pattern describes a way to access the elements of a list sequentially without knowing anything about the elements contained therein.

When one wants to traverse a list, one creates an Iterator that references that list. The Iterator subsequently becomes the sole interface one uses when traversing the list. Four core methods are provided to this end: 1) First, 2) Next, 3) IsDone, and 4) CurrentItem. (Gamma et al., 1995, page 257)