Paper 219-29

Architecting AppDev Studio™ Based Web Applications with Object Oriented Methodologies

Aimee Basile, Qwest, Denver, CO
Dave Hayden, Qwest, Denver, CO

ABSTRACT

SAS AppDev Studio 2.0 provides SAS custom tags to enable quick and easy creation of JavaServer Pages™ (JSP™) to access SAS data. However, extensive reliance on these tags limits an application's functional and architectural options. Fortunately, SAS AppDev Studio provides the Java™ classes behind these tags, allowing developers to leverage the full power of Java and object oriented methodologies such as component re-use and abstraction, design patterns, and functional encapsulation.

We will present a comprehensive, component based architecture for developing web based applications with SAS AppDev Studio. Key elements of the architecture that will be discussed include: JavaBeans™, Servlets, and the Model-View-Control design pattern.

Intended Audience: AppDev Studio users and SAS users of all levels with an interest in object oriented programming.

INTRODUCTION

SAS AppDev Studio 2.0 is an integrated development environment that allows a developer to use different methods to present SAS data. These methods can be purely logical, such as SAS code, or object oriented, such as Java and Java components.

As a SAS user, you are probably very familiar with the linear progression of a structural programming language. However, in order to use the full power of AppDev Studio, and the concepts presented in this paper, it is necessary to understand the object oriented (OO) programming nature of Java.

As a structural programming language, SAS takes input data, processes it, and outputs data. On the other hand, object oriented programming focuses on objects to be manipulated rather than the logic required to manipulate them. Objects can be thought of as any item that can be described with data. Everyday things can be objects, such as a car or building. For example, a building has data that describes it, number of floors, address/location, number of entries and exits, type of use, etc. On the other hand, an object can be a computer icon or button. For example, the data that describes a computer button could be, size, color, text, location, etc.

Since objects are the heart of OO programming, it is important to first define the objects, and the data contained by them, before deciding how to manipulate them. Once an object is defined, the logic of how to manipulate, or use, them can be created. This logic is called the object's methods. Methods provide instructions and the object characteristics provide relevant data. Each method is a stand-alone sequence that performs one task.

The stand-alone nature of methods and objects represents one of the benefits and important concepts of OO programming. This isolation is referred to as encapsulation in OO design. The benefit of encapsulation is that several distinct objects and methods can be fitted together in any order to meet the requirements. Each object or method can be thought of as a building block, or component, that can be interchanged and organized in many different ways.

Because of this flexibility, it can be very easy for an application to become disorganized. For example, one of the technologies AppDev Studio supports is JavaServer Pages (JSPs). Obviously, JSPs are object oriented since they are Java-based. However, JSPs are so flexible that a developer can include components from many different sources, like HTML, JavaScript™, Java and the SAS custom tags that AppDev Studio provides. It is very easy to write a page with all of these different components, and AppDev even gives developers a drag-and-drop interface with which to create such a page. However, as you can imagine, a JSP with all of these components and each of their different syntaxes can be very confusing. So, even though a JSP can perform all of the tasks of an application, it is too unmanageable for anything but an extremely simple application.

Therefore, it is very important to have an organized, well-designed plan for an application. It is best to have a plan even before deciding what objects to create and what data the objects will contain. However, even if you have an application that is quickly becoming an unwieldy monster, don't worry, you can still step back and create an elegant, manageable, even re-usable architecture. (Believe us, we did it!)

COMPONENT DRIVEN DESIGN

In the realm of software development the concept of components is a simple yet powerful one. In order to introduce the concept, we will present the component concept at work in another discipline: homebuilding.

A general contractor needs to install windows in a new home that is nearing completion. Should he try to build the windows himself from scratch or get completed units from a window manufacturer? To answer that question the contractor briefly considers these two facts before deciding to purchase pre-built windows: he has neither the time nor experience to build his own windows AND he knows that he can get high quality, inexpensive windows elsewhere that are built to industry standards and can be installed quickly and easily. He can, essentially, just go get window 'components' and plug them in. He does not need (or want) to know anything about the window manufacturing process, or the internal mechanisms by which the windows can be easily opened, closed and locked, to make use of them. He is only interested in their functionality as components. He is not interested in the details of how they achieve that functionality.

Using this example as a primer we can now define the term component and examine its value in terms of systems development.

> *A component is a self contained reusable software building block that fully exposes its functionality while abstracting the details of its implementation.*

There are really three key principles at work here, all of which, not coincidentally, have been derived from the object oriented (OO) programming paradigm. They are: Encapsulation, Component Reuse, and Abstraction.

ENCAPSULATION

Encapsulation in this case means functional encapsulation and is referred to above as 'self containment'. The principle holds that components should not rely on any external mechanism in order to perform their function. In other words, components should be functionally independent of the larger application that they operate within. For example, you have developed an authentication component for your web application that displays a form, captures a username and password from a user, and searches your corporate database to validate that user as a member of a certain group before they are allowed to use the application. If the component was poorly designed – for instance it violates encapsulation rules by relying on the presence of a certain application specific variable in a user's web session before it will validate their group membership – then it is not really a "self contained reusable software building block" at all. The component is nothing more than some application specific code that has no value outside of that particular application. This leads us to the issue of component reuse.

COMPONENT REUSE

Long thought of as the holy grail of object oriented software development, code reuse is, unfortunately, much more often cited in principle than it is applied in practice. The idea is simple enough: build objects (components) that provide well defined, generic functionality and then reuse them by combining the generic building blocks in myriad ways to create highly specialized applications. Consider the authentication component, and assume its encapsulation issues have been resolved (it no longer relies on any external components to properly perform its own function). Does that mean that it is a quality component? Not necessarily. Take for example the instance of a good old fashioned hard code. What if the specific membership group that the component authenticates users against is written somewhere in the component itself? If you need to build another application that happens to use the same group authentication scheme, no problem. There is no need to build an authentication component from scratch since you already have a perfectly functional, well encapsulated one that is ready to be plugged into your new application as is. If, on the other hand, you need to authenticate against a different group, you are going to need to build another component. In this case then, an ideal authentication component would not only be functionally encapsulated, but it would also support reuse by being redesigned to accept an initialization parameter that specifies the required membership group. The result would be a very valuable component – one that offered 'plug-and-play' authentication functionality to *any* web application.

ABSTRACTION

Now imagine that, sure enough, some other development group wants to plug your authentication component into their new application. After all, user authentication is a standard functional requirement for many web applications. Do you need to sit down with them and review every line of code so they can understand the mechanics of how your authenticator component actually works? Not at all. In fact, they need not ever look at any of your underlying code or understand much of anything about the way it internally operates to make use of it. The concept that makes this specific component transaction so easy, and code reuse so manageable in general when it comes to component driven design, is abstraction. Abstraction is the idea that, especially when it comes to complex systems, any given component's specific implementation of its exposed functionality is essentially inconsequential. This is a powerful concept to application architects, as it frees them from a focus on the minutia of particular functional implementations

and allows them to focus on the bigger picture. Think back to our homebuilder. If he decided to build his own windows he would have to, among other things, take the time to learn how to build windows, and then actually build them. But if he buys pre-built windows, then all he has to do is be sure to leave a space in the walls where the windows should go, plug them in, and move on to other things.

At this point the value that component driven design principles bring to an application's architecture should be clear. However, at least one big question remains. How do we organize the components? Is each web application so unique that no discernable design pattern exists to use as a template? Or is the opposite true – that viewed from a high enough level, the vast majority of well designed web applications have numerous architectural features in common that we can borrow from?

MVC: THE MODEL-VIEW-CONTROLLER DESIGN PATTERN

A design pattern is nothing more than a common approach to solving a common software engineering design problem. Ideally, patterns represent "best practices" for performing standard software engineering tasks. Patterns exist to address just about any recurring systems development challenge, and as that implies, pattern granularity varies right along with the scope of the task that the pattern is being used to address. There are patterns for tackling very specific problems such as where to encapsulate your presentation-related formatting logic, as well as more abstract design patterns that can be employed as templates for architecting entire J2EE applications. MVC is a classic example of this latter category.

The Model-View-Controller design pattern is primarily concerned with separating an application's information (model) from its presentation (view). MVC accomplishes this functional encapsulation by grouping objects (components) into three distinct categories, each with a well defined set of responsibilities:

The Model objects represent the application logic layer. They are responsible for doing things like performing calculations, connecting to remote databases, and in general, managing an application's data by acting as a proxy to persistent data stores. An example Model object from AppDev Studio is the MultidimensionalTableInterface class.

The View objects collectively represent the presentation layer of an application. They make up the "front end" and are relied on for controlling an application's look and feel, prompting user action, gathering user input, and displaying results to users. An example View object from AppDev Studio is the MDTable class.

The Controller layer objects' primary responsibility is to act as an intermediary between the application logic (Model) and presentation (View) layers. The controller defines application behavior by accepting user requests, initiating activity within the application/business logic layer, and selecting appropriate views for presentation to the client. The most common implementation of the controller apparatus within AppDev Studio is to create servlets that extend the HttpServlet class.

MVC AND APPDEV STUDIO

The advantages of developing AppDev Studio based web applications under the MVC paradigm are numerous. First and foremost is the fact that SAS recognizes the ubiquity of MVC and has already organized their Java classes along standard MVC lines. For example, AppDev Studio provides a series of classes that they call "transformation beans" that are solely dedicated to presentation functionality. But there are other significant reasons. MVC applications typically require less code by eliminating duplication, and are easier to maintain/enhance/extend by nature of the fact that they are so compartmentalized. (Imagine being able to tell your boss that yes, you can build her another brand new OLAP web application that has the same look, feel, and functionality as your original, and that you'll be done in "a couple of minutes" since all you need to do is make an exact copy of your existing application and change a single configuration parameter that controls which data source your Model points to.)

FROM THEORY TO PRACTICE

Much of this paper to this point has been dedicated to a theoretical discussion of components, object oriented design principles, and the MVC design pattern. Here, we will explore the practical implications of those theories.

Many initial development efforts in support of Java based AppDev Studio projects produce page centric (not MVC) applications that are functional, yet inefficient and limited with respect to their incorporation of custom functionality. The reason for this is fairly straightforward: SAS provides an extensive tag library and drag and drop capabilities within AppDev Studio that enable developers too quickly and easily add Java components and their attendant out-of-the-box functionality to JavaServer Pages. In other words, it is easy to create quick and dirty web applications without writing much code. But these applications and their development approach have a downside.

Web applications that utilize a page centric architecture suffer from problems that stem from the inherently 'stateless' nature of the web. Without the single central point of access control that MVC enforces, users are free to access any

JSP, at any time, in any order that they see fit – effectively circumventing any application flow that you may have envisioned. There are of course ways to program around these issues, but the solutions are necessarily ungainly since every page would have to perform every control function that could otherwise be performed by a single centralized component under MVC. With all that duplication of effort, and the tangle of code that JSPs can become with the HTML, JavaScript, JavaBeans, SAS Custom Tags and raw Java code needed to satisfy model, view and control functionality, it is no wonder that page centric applications are notoriously difficult to maintain, enhance, and reuse.

Figure 1 illustrates the responsibilities and complexities that JavaServer Pages attain under two distinct architectures.

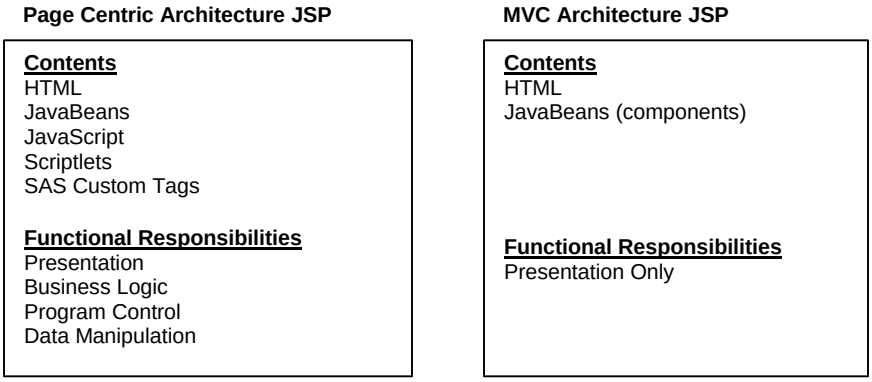

Figure 1 – Page-centric vs. MVC JSPs

Perhaps the most interesting feature of the Java Server Pages role within our MVC architecture lies not in what they are asked to do, but in what they are not asked to do. As depicted in the above graphic, their list of functional responsibilities has been limited to a single item: presentation. What about all the business logic, application flow and control, and database communication functionality? If JSPs are simply going to act as the View layer of our web applications, what technologies are going to be employed to fulfill the other roles, and how will they be organized?

JAVABEANS

At the lowest level, JavaBeans are nothing more than Java objects that can be referenced within JavaServer Pages. Sun Microsystems™ has published an entire set of specifications (the JavaBean API) for creating these stand-alone, reusable software components that can then be referenced via special tags within JSPs, but assuming that you are willing to forgo the shortcut to object method calls that these tags represent, strict adherence to those specifications is not necessary to access the full power of Java within web applications.

Here is a simple example of how JavaBeans can be used to produce highly customized, dynamic content within component driven web applications. Suppose you want to add a greeting to your application's main page that displays a personalized welcome message along the lines of "Welcome Stella". All you would need to do is create a bean that captures a user's first name when they log in, and then write that bean to the user's session. On the main page of your application you can then add a reference to your 'Greeter' bean that simply asks the bean (Java object instance) to write its message out to the page. This works perfectly by respecting the fact that JSPs have been asked to perform presentation functions only. The bean is not created by the page - it already exists in the user's session - it is only referenced by the page. Under this design, regardless of what else is happening within the application (think functional encapsulation, components), your users will be personally greeted by name each time they visit the main page. However, this begs the question: "How does a customized 'Greeter' bean get created and written to each user session?" Well we already know that the JSP is not creating the bean since our pages are not allowed to create content under MVC, they can only be used to present it. Alternatively, the bean cannot instantiate, or create, itself, so a servlet must be used.

SERVLETS

Java Servlets often play a critical role under the MVC design pattern by serving as the Control component. They do this by acting as the single, centralized request handler for ALL incoming client requests to an application, and can be thought of as something of a gatekeeper. Servlets also have the added advantage of being highly configurable via initialization parameters – which makes them valuable tools in the creation of generic application frameworks from which specific web applications can be easily derived.

Servlets, much like beans, are just Java objects. Since we are working in the world of the web though, our servlets are all derived from a particular Java class, the HttpServlet class. Aside from a few create, destroy and initialization

methods, HttpServlets can really be boiled down to one function; handling http (client web browser) requests. Under the web-tier architecture that will be presented in the next section, the control servlet has simplified the handling of client requests by abstracting a particular request's action code out of the servlet itself and into generic command beans. So the control servlet in this case is deceptively simple. It lies there waiting for client requests, and when it receives one it parses the request to expose its embedded command, creates an instance of that command type, and tells that instance to execute. The reason that it can be thought of as deceptively simple is that the control servlet, consisting of about a dozen lines of code, essentially *is* the web application in that a single instance of it manages all client interaction, database connectivity, business logic, user authentication, etc. It does so via a heavy reliance on command beans (which is where most of the code resides, including code that directs clients to a specific JSP for presentation after the command's execute method has completed).

This heavily abstracted, simplistic control servlet brings us full circle back to component driven design principles. As long as you are creating (or better yet, utilizing someone else's) fully encapsulated beans/components, they can be easily plugged into this architecture. Looking back at the 'Greeter' bean example from the previous section it is easy to imagine a request coming into the control servlet that has embedded within it, among other things, a "createGreeter command". The control servlet would parse the incoming request for its commands, realize that it was being asked to instantiate a createGreeter bean, and then create that bean and tell it to execute. Some pseudo code for the createGreeter's execute method follows:

```
//createGreeter.java
  public void execute ( HttpRequest, HttpResponse )
  {
      get the user's session from the passed request
      get the user's authenticated first name from the session
      create a Greeter bean using the first name
      write that Greeter bean to the user's session
  }
```

Now any time the user visits the main page (the JSP that references the Greeter bean in the user's session) they will be greeted by name. Keep in mind that the functionality you are adding with the Greeter bean is very simple, but the mechanism that adds it does not change no matter how complex the bean you are adding. So how about a graphing bean? Or a customized menu bar bean? How about a bean that can be toggled to point at Oracle or SAS data as needed? The possibilities are truly endless.

The point is this: servlet centric MVC architectures created using AppDev Studio make it possible to achieve component oriented plug and play capabilities for web applications, regardless of a component's complexity or functionality. The next section will briefly present a practical example of one such architecture.

PUTTING IT ALL TOGETHER

At this point a comprehensive system architecture for developing Java based web applications should start to become visible. As shown in Figure 2, JavaServer Pages will be utilized for the View layer since they excel at application presentation tasks and are so well suited to a component orientation via their ability to make use of JavaBeans. Servlets will act as the Controller mechanism since they effectively separate an application's information from its presentation and give us the ability to run all user interaction through a single control point, which in turn allows for the centralization of administrative functions such as user authentication and logging. And finally, the Model will consist of various JavaBeans that the control servlet will manage and use as building blocks to affect the required business logic.

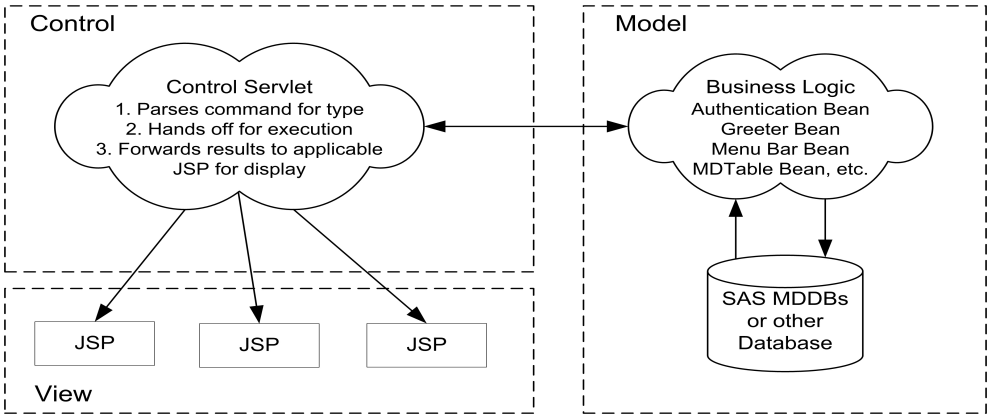

Figure 2 – MVC Application Architecture

APPLICATION FRAMEWORKS

In the Servlets section of this paper we briefly mentioned that one of the benefits of servlets is that they are configurable via initialization parameters, and that this feature enabled the creation of application frameworks. But what are application frameworks? To answer that question we need to revisit another previously presented section of this paper one last time – our homebuilder example. Imagine that our homebuilder had so much success using his pre-built components (windows) that he decided to see how far he could push the component concept. For his next project, he is pretty sure he will be able to utilize completely pre-built walls (windows and all) to save himself some time and money. And after that, he is thinking of getting into the development of subdivisions, so he figures he might just be able to find some quality pre-built modular homes (walls with windows, floors, a roof, the whole works) and really save himself some time and money. Well, if he is not there already, our homebuilder is pretty quickly going to realize that because of the limits inherent to physical architecture, he can only reasonably expect to carry his concept so far. However, if he was working within the software architecture paradigm, he would not have any such limits.

Application frameworks are the result of applying the configurable component concept to applications themselves, not just their individual parts. For example, much like we pass a user's first name as a parameter to our generic Greeter component and get a custom JavaBean in return, application frameworks allow us to pass a series of parameters to an application framework and have it return a series of customized servlets that each act as an individual web application.

The mechanism that makes this possible is a web application's deployment descriptor, which is nothing more than an XML file. Deployment descriptors serve many functions (declare error pages, perform servlet mapping, point to tag libraries, etc.) but the following XML example highlights the key areas.

```
//web.xml
...
<servlet>
        <servlet-name>
                Application 1
        </servlet-name>

        <servlet-class>
                controlServlet
        </servlet-class>

        <init-param>
                <param-name>
                        app_title
                </param-name>
                <param-value>
                        SUGI Application One
                </param-value>
        </init-param>
        ...
</servlet>

<servlet>
        <servlet-name>
                Application 2
        </servlet-name>

        <servlet-class>
                controlServlet
        </servlet-class>

        <init-param>
                <param-name>
                        app_title
                </param-name>
                <param-value>
                        SUGI Application Two
                </param-value>
        </init-param>
        ...
</servlet>
```

The above example reveals that two servlet driven web applications will be created (Application 1 and Application 2) that both utilize the same controlServlet class for their controller mechanism, and that each one has a specific application title associated with it. But remember, this is only an example. The full text of the deployment descriptor

likely contains dozens of initialization parameters for each servlet driven web app that is defined. Some examples of other likely initialization parameters include: the address of the database that this web app will connect to, the port it should connect on, the style sheet that should be applied to all presentation content, default settings for various variables, and just about anything else you care to think of. The deployment descriptor can even be used to signal which pieces of functionality (beans) should be made available to users of the application. In this manner, by combining highly configurable components to create applications and then treating applications as highly configurable components themselves, multiple web applications (each with its own customized look and feel, data source, and functionality) can be created out of a single set of resources.

CONCLUSION

AppDev Studio 2.0 is an integrated development environment that supports many different kinds of projects and many different architectural styles. However, to take advantage of all the capabilities that the AppDev Studio class library has to offer, the right architectural approach is vital. This paper has hopefully demonstrated that by applying sound object oriented design principles, some architectural forethought, and the power of Java, almost anything that you can imagine is possible.

REFERENCES

Fields, Duane K., and Mark A. Kolb. *Web Development with Java Server Pages.* Greenwich, CT: Manning Publications Co., 2000.

Johnson, Mark, Inderjeet Singh, Beth Stearns, et al. *Designing Enterprise Applications with the J2EE™ Platform, Second Edition*. Santa Clara, CA: Sun Microsystems, 2002. java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/titlepage.html

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Aimee Basile
Qwest Communications, Inc.
1801 California, Suite 650
Denver, Colorado 80202
(303) 965-0658
Aimee.Basile@qwest.com

Dave Hayden
Qwest Communications, Inc.
1801 California, Suite 650
Denver, Colorado 80202
(303) 896-1641
Dave.Hayden@qwest.com