Paper 007-31

# Using Design Patterns to Integrate SAS/IntrNet® with Web Technologies

**Manolo Figallo-Monge**
**Freddie Mac**
**Information Technology Group**
**McLean, Virginia**

## ABSTRACT

Design Patterns are reusable design solutions for commonly occurring problems in software engineering. The purpose of this paper is to demonstrate how SAS developers can use design patterns in order to write more extensible and reusable code while deriving more enjoyment in the process. We present three simple applications related to Workflow Management that use design patterns and SAS:

1. XML Configurations using the Singleton and Observer Pattern
2. Role-based Authentication using the Chain of Responsibility and Decorator pattern
3. Model View Controller Workflow Architecture using SAS ODS

We determine that SAS/IntrNet® is a particularly suitable solution for implementing design patterns as it can be used to integrate many Internet technologies. Furthermore, by decoupling presentation and data, we also conclude that design patterns produce more robust and well-designed SAS applications.

## BACKGROUND AND OVERVIEW

Patterns occur in everyday life[1]. They help us understand, appreciate, and, most importantly, manage complexity. Software design patterns are re-usable design solutions to problems in software applications development. They are oftentimes denoted by definitions described in *Design Patterns: Elements of Reusable Object-Oriented Software*, a seminal work on the topic, which offers timeless and elegant solutions to problems in software design. Although, as the title suggests, design patterns are used typically with object-oriented languages, this paper proposes that many of these patterns are very relevant and applicable to applications developed using SAS/IntrNet® and procedural languages, such as SAS.

## BENEFITS

There are many benefits to using design patterns, and they are best realized and most palpable after project teams have adopted them[2]. These benefits include the following:

- Economies of scale. This is achieved because use of design patterns enables developers to leverage off of the design experience of others, and, as a result, costs are significantly reduced for every unit of development output. What's more, once you use a design pattern a lot of other design decisions follow automatically. This means that developers get a design "right" faster
- Managing complexity. Developers who use design patterns can tap into their design arsenal for structuring and creating SAS applications. It provides teams with a common vocabulary to facilitate team communication and describe complex ideas succinctly. In sum, a common vocabulary for tackling difficult problems is powerful; it facilitates communication between members of a team, particularly new members and especially those with an OO background.
- Increased enjoyment in the application development process. Recognizing and identifying patterns in the work we do is fun! It builds self-confidence to know that you have successfully solved a difficult problem through solutions that are tried and true. Producing clean, pleasing code leads to a sense of enjoyment and, in addition to this, more extensible, reusable, and maintainable code; in sum, design patterns are an obvious antidote to the frustrations of spaghetti code.

## EXAMPLES

SAS/IntrNet® is built on design patterns.

[1] Take, for instance, the Fibonacci series.

[2] The opposite of Design Patterns are Resign Patterns: see Bibliography

Paper 007-31

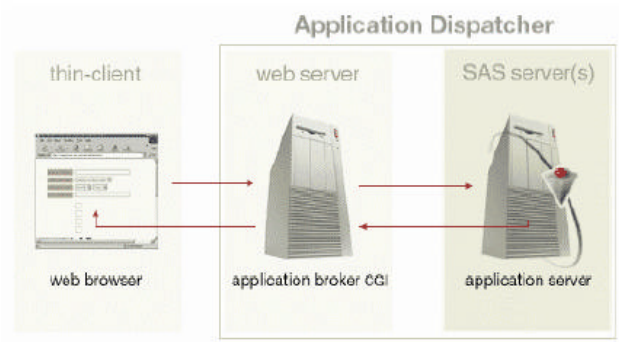



**Figure #1 SAS/IntrNet® Architecture**

The Application Broker CGI resides on the Web Server and forwards requests for processing to the SAS Application Server. This behavior is summarized by the Chain of Responsibility pattern. Using the Chain of Responsibility pattern, we decouple the request on the web server from the receiver (i.e., SAS application server). By passing the request along the chain, we can introduce a load manager object to handle the request without specifying any specific port explicitly. The upshot is a more extensible and scalable system because we have reduced coupling and have added flexibility in assigning responsibilities to objects.

## NON-SOFTWARE EXAMPLES

To increase the communicative power of Patterns, Michael Duell introduced non-software examples of Design Patterns. For example, The Chain of Responsibility described above is demonstrated in the military where some underling asks for approval from superior to superior until someone finally makes a decision (see Figure 2).

**Figure #2 Chain of Responsibility Design Pattern**

SAS/IntrNet® also exhibits the Façade Pattern through the web browser which acts as a higher-level, unified interface to subsystems below. Similarly, a Customer Service Representative, as Duell notes, acts a façade to several departments in an organization (e.g., billing, order fulfillment, shipping). This is captured in Figure 3.

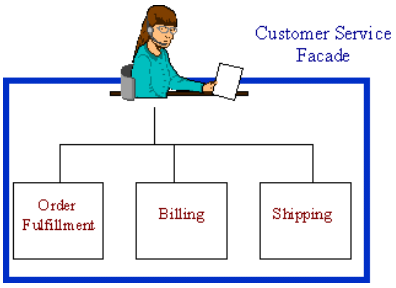



**Figure #3 Façade Design Pattern**

Other noteworthy patterns, which we will explore with SAS code, include:

- Observer Pattern.
- Singleton Pattern
- Decorator Pattern
- Chain of Responsibility
- Factory Pattern.

In the next section, we use these patterns in the SAS examples mentioned above, or, to be succinct:

1. XML Configurations
2. Role-based Authentication
3. SAS XML ODS reports

We then provide a real-world case-study application for workflow management using these simple examples.

## XML CONFIGURATIONS USING THE SINGLETON AND OBSERVER DESIGN PATTERNS

The Singleton pattern allows us to provide global access to a single instance of an object. By creating a single, unique instance of an object we avoid all sorts of problems such as incorrect program behavior, overuse of resources, and inconsistent results.

In the real world, this pattern is exhibited by the presidency where there can only be one active president at a time, and the title "President of the United States" is a global point of access that identifies that person in the office. See Figure #4

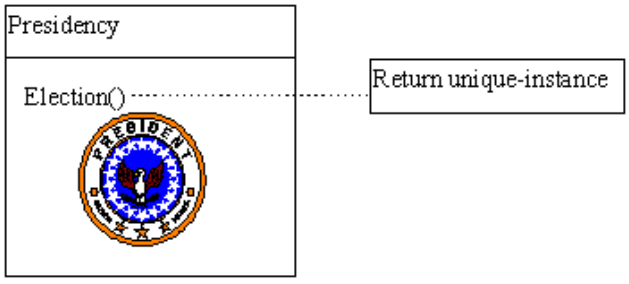


**Figure #4 Singleton Design Pattern**

The Observer pattern defines a publisher-subscriber relationship. Using this pattern, we define a one-to-many relationship where an object notifies others of any changes and they are subsequently updated automatically.

The dynamic relationship between publisher and subscriber is epitomized in some auctions where an auctioneer *pushes* price information to bidders. When the price changes, all bidders (i.e., subscribers or observers) are notified immediately (see Figure 5).

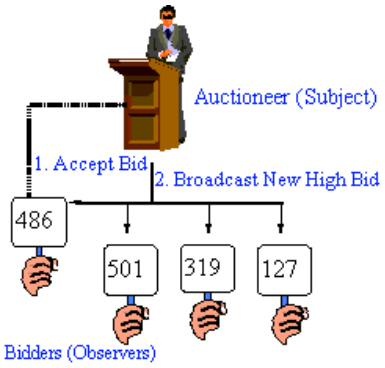



**Figure #5 Observer Pattern**

The interface we provide between subject (i.e., publisher) and observer (i.e., subscribers) is also depicted in Figure 6. A report *pulls* configuration information from data *"published"* in an XML configuration file.

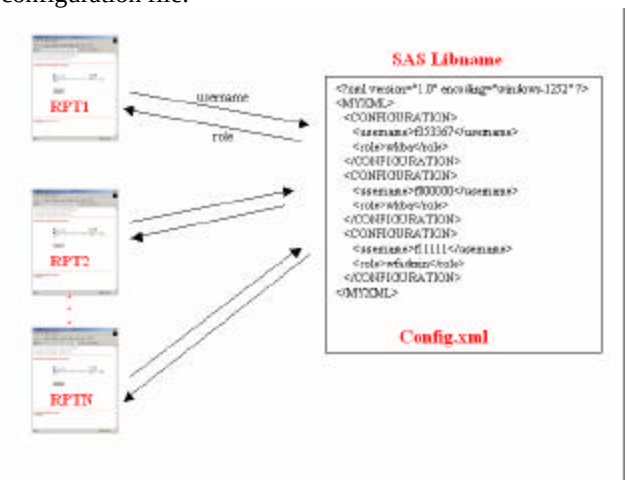



**Figure #6 SAS XML Libname is used to implement the Observer and Singleton Design Patterns.**

Many SAS reports use the same configuration information (e.g., server name, database name, passwords, user roles) but this information is often hard-coded in SAS reports. Using the Singleton and Observer patterns, we can design SAS reports so that a global point of entry for this configuration information exists (in an XML file), and all reports are notified in case any configuration information changes.

Using XML Libname to store name-value pairs for configuration information results in substantial reductions in costs, errors, and time. Indeed, some estimates are as high as 86% costs savings and avoidance due to XML configuration files.

Changes, in sum, to configuration information used by X reports means that we no longer need to maintain and change manage SAS code but can focus our efforts in maintaining XML configuration files; this significantly reduces overhead costs in maintenance and deployment. We, in sum, we can experience economies of scale vis-à-vis system maintenance ---i.e., because XML configurations allows us to "kill X birds with one stone".

Code examples are shown in Appendix B.

## ROLE-BASED AUTHENTICATION USING THE DECORATOR AND CHAIN OF RESPONSIBILITIES DESIGN PATTERNS

The decorator pattern is used to surround and sometimes group code in order to produce new capabilities to it dynamically; it creates, in other words, a wrapper.

This is exhibited in a framed painting where paintings can be "layered" with a mat and a frame to form a single visual component with new capabilities (e.g., "hanging"). See Figure 7.

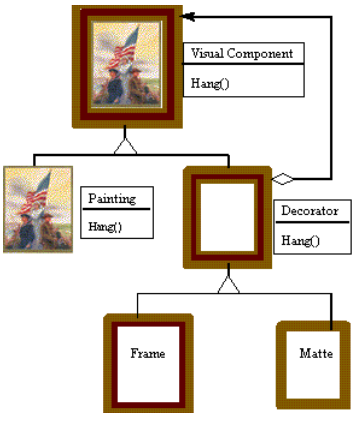



**Figure #7 Decorator Pattern**

In this series of put statements, we have broken up the header of a page, body of a page, and page footer into separate components (see Figure #8):

Paper 007-31

```
put '<%@ include file="sugi-header.jsp" %>';
put "&decorator_body";
put '<%@ include file="sugi-footer.jsp" %>';
```

**Figure #8 Decorator Pattern Code (see Appendix for link to the complete code)**

Figure 9 demonstrates this code visually with actual output from SAS XML ODS. The variable &decorator_body represents the name of body "skin" that should be used, depending on the user's role. That is, we can use different XSL skins to wrap the SAS-XML report embedded between the header and footer. Our catalog of "skins" implements different functionality depending on the user's role (as we shall see)—e.g.,traffic-lighting, pagination, moveable columns, sorting, highlighting rows, etc. Figure 9 demonstrates functionality useful to a Business Analyst.

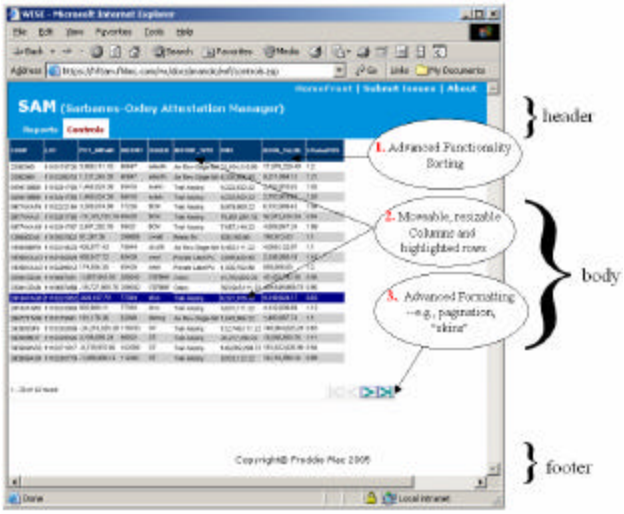



**Figure #9 Decorator implemented with SAS ODS XML.[3] Note that the report "body" contains advanced functionality: 1) Column sorting on-the-fly; 2) Moveable and resizable columns and highlighted rows; and, 3) pagination**

From a software management point of view, we can delegate the development and maintenance of these skins to front-end developers. The upshot is more maintainable, modular code and clearer "separation of duties" in software development.

[3] Resulting GUI includes: 1.Advanced Functionality (Sorting); 2. Moveable, resizable columns and highlighted rows; 3.Advanced Formatting—e.g., pagination.

Recall that the body in the code in Figure #8 is a variable. How is this variable assigned? Based on a user's role (as defined in an XML configuration file), we determine what "body" is appropriate for any given user. This is the first step towards implementing role-based authentication, and it is representative of the Chain of Responsibility pattern.

The Chain of Responsibility pattern allows us to decouple the sender of a request for the receiver and, furthermore, it passes a request until it is recognized and handled.

This is exhibited by the military example above, but it is also exhibited by a mechanical coin sorting bank a single slot is used and any coin that's deposited is routed to the appropriate receptacle by the mechanical mechanisms within the bank. See Figure 10.

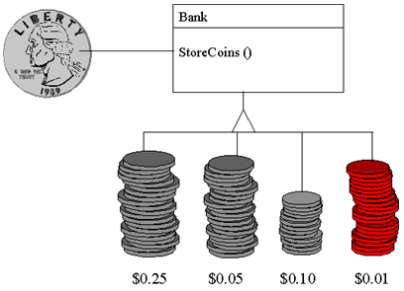



**Figure #10 Chain of Responsibility Pattern**

SAS/IntrNet®, fortunately, has a variable we can use, _RMTUSER, to capture the username in session. This username can then be routed to an XML configuration file to derive a user's role. In our coding examples, there are only two roles:

1. Supervisor or Admin
2. Business Analyst

With this role information we can then determine a host of information pertinent to any given role—e.g., XSL style sheets ("Skins"), body file, header information, etc.

The flow is a follows:

GetUsername→DetermineRole→Assign &decorator_body

As in the sorting bank example, we use a single interface to get a user's role, and then derive the proper body file based on this information.

## SAS XML ODS REPORTS USING THE FACTORY PATTERN

Paper 007-31

Our role-based system relies heavily on XML. With our body "skin" determined by the Chain of Responsibility pattern, we are ready to create an XML report based on any given user's role; this is done by using a Factory pattern. In other words, we provide data (i.e., role) to the Factory to return an instance of an XML/XSL report suited to that role.

Injection molding presses demonstrate this Pattern since they provide an interface to inject plastic into a mold. The mold, however, ultimately determines the final output. See Figure 11.

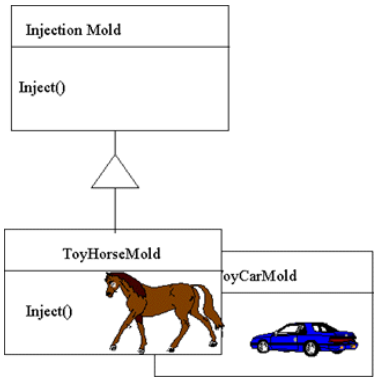



**Figure #11 Factory Design Pattern**

Similarly, our COR (Chain of Responsibility) macro assigns values to the XML/XSL variables that are then used by the XMLFactory macro to create report output. The code for our XMLFactory is in the Appendix, as is a link to XSL code for creating different reports based on any given user's role. This is demonstrated in Figure 12.

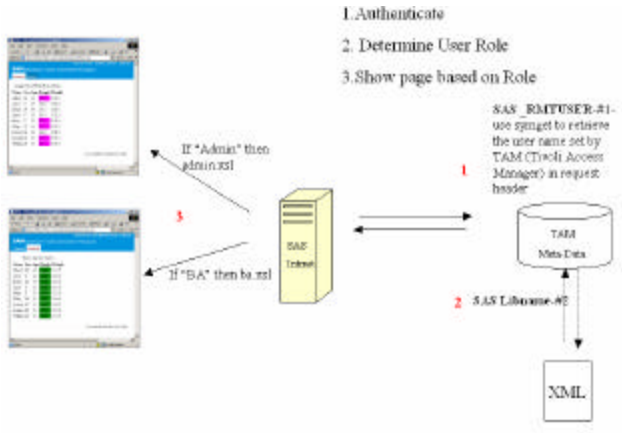



**Figure #12 Role-based authentication using SAS/IntrNet®**

## A WORKFLOW APPLICATION USING THE MVC AND SAS: PUTTING THE PIECES TOGETHER

Now that we have defined implementations of design patterns using SAS, we can put our macros together to create an architecture that integrates SAS with Web technologies such as XML and XSL. This architecture is rooted in design patterns and is referred to as a Model View Controller (MVC) Architecture, as described in Appendix A.

The goal of the MVC architecture is to separate or decouple data from presentation and facilitate reuse and use of "Skins". Furthermore, it's useful in providing the end-user with the ability to affect changes to the data and presentation through the use of a "controller". This is summarized in Figure 13.

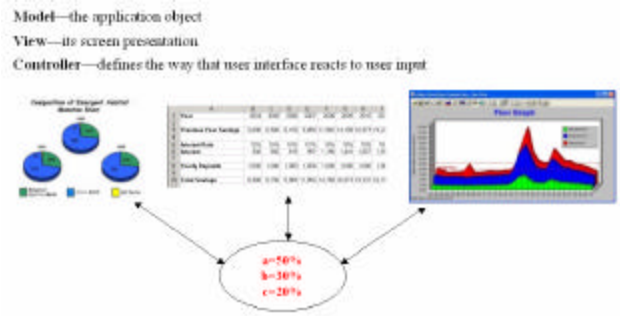



**Figure #13 Model View Controller Architecture**

The MVC Architecture ensures the following segregation of duties:

- Model: Encapsulates data
- View: Provides presentation of the Model
- Controller: Handles business logic and input

A workflow application is particularly well suited for this model. This type of application enables an end-user to execute a particular action based on their role in an organization. In today's business climate where Sarbanes-Oxley rules require a "segregation of duties", many controls, and approval steps while an "artifact" is promoted to a production state, workflow is key.

Workflow is, in other words, the automation of business process, in whole or in part, during which documents, information, data, or tasks are passed from one participant to another for action. It thus ensures that any action is traceable to any given role (another Sarbanes-Oxley rule).

The key benefits of automated process flow include:

- Improved efficiency—automation of many business processes resulting in the elimination of many unnecessary steps and integration of disparate processes.

- Better process control—i.e., improved management of business processes achieved through standardized or uniform working methods. This focus on business processes also leads to their streamlining and simplification
- Improved customer service—consistency and uniformity in the processes leads to greater predictability in levels of response to customers

Our workflow model contains states, events, and transitions:

- **Role**: A role defines who can enter a given state and execute any given event.
- **State**: A state defines the current status of an item in the process flow, such as Published.
- **Trigger**: A trigger or event defines the operations that can be performed on an item, such as upload or "press ok". In addition, process flow events can be used to trigger scripts.
- **Transition**: A transition moves an item from one state to another. Change is an example of an event that can be used to cause a transition. When creating a transition, in addition to selecting the event, the next state must also be specified.

This, as well as the MVC architecture to implement our workflow model, is summarized in Figure 14.

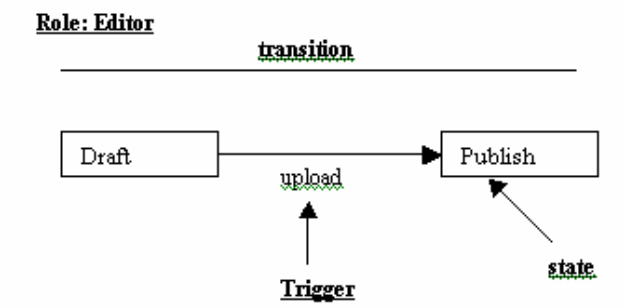



**Figure #14 The design components of our workflow mode: role, transition, event/trigger, state**

As mentioned, our workflow model consists of only two roles: Supervisor/Admin and Business Analyst. The role of an Admin (much like the Editor in Figure 14) is to move a report to the production or "publish" state using the "Admin" tab in our application, which contains a "Press OK to Publish" button. Admins can only view a report with limited functionality. The Business Analyst, on the other hand, can view a very functional report (using the "Controls" tab). The "Controls" tab makes use of the design patterns we have used so far to produce a very robust report for analysis. See Figure 9, once again, for a demonstration of the report.

As shown in the Architecture diagram in Appendix A, our Controller is implemented using SAS/IntrNet®. It plays a critical part in:

- Responding to user requests by first determining the role of a user
- Determining the appropriate XSL file, header file, and XML to use
- Generating XML (using SAS ODS)
- Redirecting the end-user to the appropriate XML report

Our code for the controller is available in the link mentioned in Appendix B.

## CONCLUSIONS

In sum, the upshot of this architecture is not only vastly improved maintenance, but also "separation of duties" among members of a software team. SAS programmers can now focus on the business logic of an application, and the presentation layer can be delegated to a team of front-end developers.

Although the code we have seen so far has been designed with proven techniques, it is still incumbent on SAS developers to become familiar with them. Team communication can only be facilitated if SAS programmers learn the current trends in software engineering and, as a result, share a common vocabulary with, for instance, front-end developers. It is because of this reason, among others, that design patterns are a critical and valuable piece of knowledge for SAS application developers.

## REFERENCES

Ben, Anver. "Design Pattern Dictionary" http://www.skilldesign.com/pattern/1/2/0.html

Duell, Michael. "Non-Software Examples of Software Design Patterns," Object Magazine, Vol. 7, No. 5, July 1997, pp. 52-57.

Duell, Michael. "Resign Patterns: Ailments of Unsuitable Project-Disoriented Software". http://www.agcs.com/patterns/papers/respat.htm

Freeman, Eric and Freeman, Elizabeth. *Head First Design Patterns.* O'Reilley 2004

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

Henderson, Donald et al. "The SAS/IntrNet Application Dispatcher " http://www2.sas.com/proceedings/sugi22/INTERNET/PAPER182.PDF

Huston, Vince. "Design Patterns: The Sacred Elements of the Faith" http://home.earthlink.net/~huston2/dp/patterns.html

SAS Institute Inc.. "SAS 9.1.3 XML Libname Engine User's Guide" http://support.sas.com/rnd/base/topics/sxle913/usersguide913.htm

Shalloway, Alan and Trott, James. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley 2005.

Slaughter, Susan and Truong, Sy and Delwiche, Lora. "ODS MEETS SAS INTRNET" www2.sas.com/proceedings/sugi27/p009-27.pdf

Stelting, Stephen and Maassen, Olav. *Applied Java Patterns*. Sun Microsystems 2002

## TRADEMARKS



## AUTHOR CONTACT INFORMATION

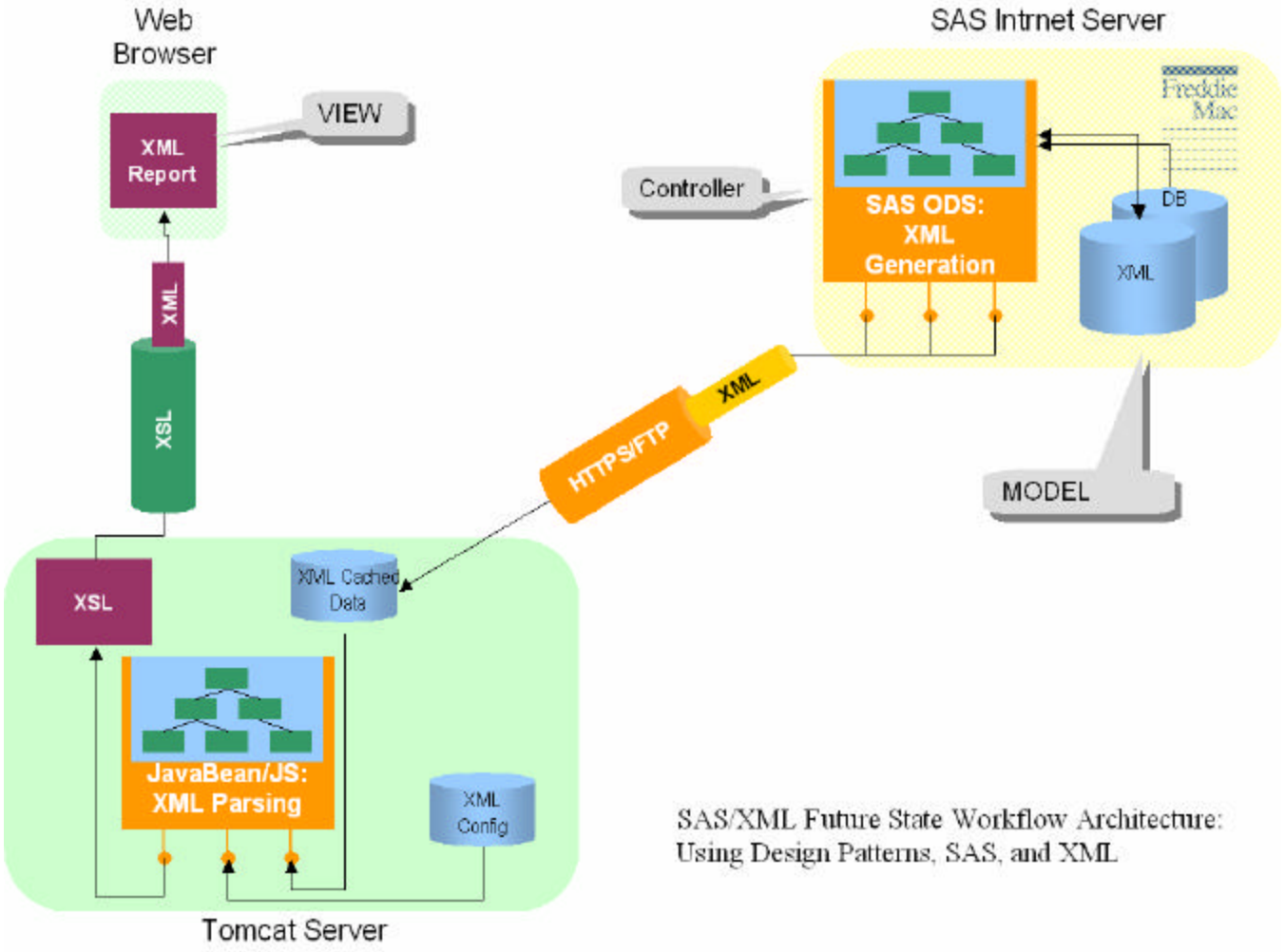
Your comments and questions are welcome.

Manolo Figallo-Monge
Freddie Mac
8250 Jones Branch Dr.
McLean, VA 22102-3107

Phone: 703 407-1775
Email: manolo_figallo-monge@freddiemac.com
Email2: mfigallo@ocf.berkeley.edu
Web: www.sabiotechconsulting.com
Code: www.sabiotechconsulting.com/sugi31

Paper 007-31

## Appendix A SAS/XML Future State Workflow Architecture




SAS/XML Future State Workflow Architecture: Using Design Patterns, SAS, and XML

Paper 007-31

## Appendix B Code Example (XMLFactory.sas) and Code Link

```
%macro getXMLReport;
        /*XML and ODS.  VERSION 2*/

        ods path work.templates(update) sashelp.tmplmst(read);


        /*step 1: define template*/
        proc template;
          define tagset tagsets.MYXML;

          define event doc;
              start:
            put "<?xml version=""1.0""";
            putq " encoding=" encoding;
            put "?>" NL;
            put "&xsl_skin_depending_on_role";
            put NL;
            put "<XML xmlns:dt=""urn:schemas-microsoft-com:datatypes"">";
            put NL;
             put "<caption>Logged in as";
             put "&my_role";
             put "</caption>";
             put NL;
              finish:
                put NL;
            put "</XML>";
          end;

              define event table_head;
                 start:
                    put "<headers>" NL;

                 finish:
                    put "</headers>" NL;
              end;
              define event table_body;
                 start:
                    put "<output-body>" NL;

                 finish:
                    put "</output-body>" NL;
              end;
              define event row;
                 start:
                    put "<row>" NL;

                 finish:
                    put "</row>" NL;
              end;
              define event rowcol;
                 putq " row=" ROW;
                 putq " column=" COLSTART;
                 putq " column-end=" COLEND;
              end;
             define event header;
                 start:
                    put "<headerrow";
                    put ">" NL;

                    put "<value>";
```

```
                    put VALUE;
                           put "</value>" NL;

                    put "<class>";
                    put HTMLCLASS;
                    put "</class>" NL;

                    put "<type>";
                           put TYPE;
                    put "</type>" NL;

                finish:

                    put "</headerrow>" NL;
             end;


            define event attr_out;
                putq " name=" NAME;
                putq " label=" LABEL;
                putq " clabel=" CLABEL;
                putq " type=" TYPE;
                putq " function=" @FUNCTION;
                putq " path=" PATH;
                putq " title=" FLYOVER;
                putq " class=" HTMLCLASS;
                putq " id=" HTMLID;
             end;
            define event data;
                start:
                   put "<data";
                   *putq " Row=" row;
                   *putq " Column=" colstart;
                   putq " raw-value=" RAWVALUE;
                   putq " missing-value=" MISSING;
                   trigger attr_out;
                   trigger rowcol;
                   put ">";
                   put VALUE NL;
                   trigger spanning / if any( COLSPAN , ROWSPAN );
                finish:
                   put "</data>" NL;
             end;
          end;
       run;


%mend getXMLReport;
```

The complete code for this Future State Architecture can be found at:

http://www.sabiotechconsulting.com/sugi31